MIT OpenCourseWare & Other Course Notes

Contents

MIT OpenCourseWare & Other Course Notes

This collection of pages contains notes for all the MIT OCW and other online courses I have taken, mainly for my own reference. You can view the source files for these notes on GitHub.

These notes may contain errors, including formatting errors, especially in the auto-generated PDF versions of these notes. If you notice an error, please raise an issue on GitHub (@hrushikeshrv/ocw) and I will be happy to fix it.

Course List

1. MIT 6.004 - Computation Structures
2. MIT 6.006 - Introduction to Algorithms - Fall 2011 (Complete)
3. MIT 6.006 - Introduction to Algorithms - Spring 2020
4. MIT 6.042J - Mathematics for Computer Science (Complete) (External)
5. MIT 6.045J - Automata, Computability, Complexity
6. MIT 6.046J - Design and Analysis of Algorithms (Complete)
7. MIT 6.858 - Computer Systems Security
8. CMU 15-445 - Database Systems
9. UCB CS 162 - Operating Systems & Systems Programming (Complete)

MIT 6.004 - Computation Structures

Taught by Professors Arvind, Daniel Sanchez, Silvina Hanono Wachman, and Song Han

Course Description

This course introduces architecture of digital systems, emphasizing structural principles common to a wide range of technologies. It covers the topics including multilevel implementation strategies, definition of new primitives (e.g., gates, instructions, procedures, processes) and their mechanization using lower-level elements. It also includes analysis of potential concurrency, precedence constraints and performance measures, pipelined and multidimensional systems, instruction set design issues and architectural support for contemporary software structures.

Course Website (Spring 2017)
Lecture Videos (YouTube, Fall 2018)

Lectures

1. The Digital Abstraction
2. Combinational Devices and Boolean Algebra

Lecture 1 - The Digital Abstraction

This course covers 3 main modules. The first module discusses digital design, combinational and sequential circuits. The second module teaches assembly language, processor architecture, caches, pipelining, etc. The third module talks about entire computer systems like operating systems, virtual memory, I/O, etc.

Each of these modules represents one level of abstraction. Layers of abstraction higher on the hierarchy hide a lot of the complexity of the lower layers of the hierarchy. And even though the exact implementation of any one of these layers may change in the future, their underlying concepts remain the same and are long-lived.

Digital and Analog Systems

Analog values are values that vary continuously over time. Digital values are values that vary discretely over time. Most computing systems are digital systems because digital systems are just a lot more resilient to noise. Analog computing systems are rare.

Since a computer is a digital system and voltage is an analog value, we need to have convention to convert an analog input into a digital one. We could simply decide a threshold value V_{th}, and say that anything above the threshold is a 1 and below it is a 0, but in this case a little bit of noise added to a signal that is close to the threshold can end up flipping bits.

Instead, we define two thresholds - V_l and V_h. Any signal below V_l is a 0, and any signal above V_h is a 1. Any signal in between V_l and V_h is *undefined*, and the system is allowed to behave however it wants in that case (even in an undefined manner).

However, there is one more problem. If a system sends a signal very close to V_l or V_h, a small amount of noise could push that signal into the undefined zone. We solve this by having different high and low thresholds for input and output - V_{ol}, V_{oh}, V_{il}, V_{ih}. This means that if a system outputs a 0 that is close to V_{ol} and some noise pushes it to be higher, the value can still be lower that V_{il}, and the system that takes that signal as an input will still correctly interpret it as a 0. This gives us a much better tolerance towards noise. Of course, this means that $V_{ol} < V_{il} < V_{ih} < V_{oh}$.

Voltage Transfer Characteristic

The VTC is a plot of V_{out} vs V_{in}. Essentially, it is a plot showing how the output voltage changes as the input voltage changes for a particular device. It shows you static behaviour, which means it does not tell you anything about how fast the device responds.

This is the VTC for a buffer, which is essentially a repeater. It outputs a 0 when it gets a valid 0, and a 1 when it gets a valid 1.

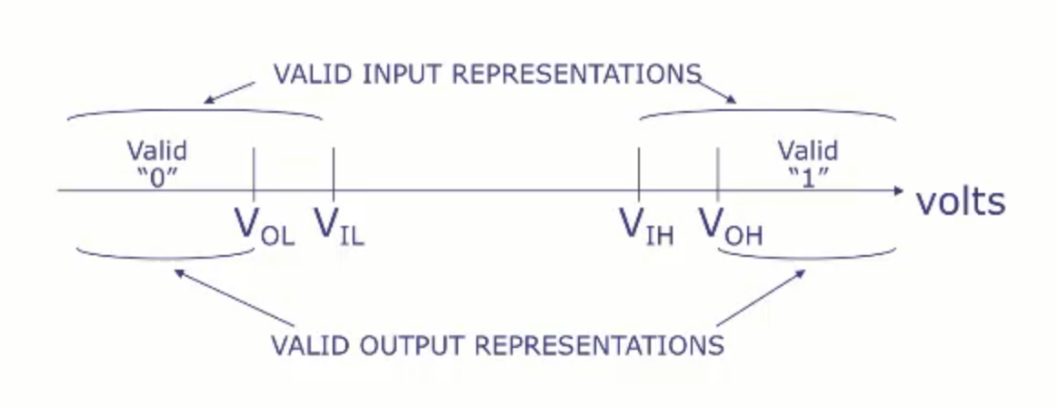


Figure 1: Noise margins

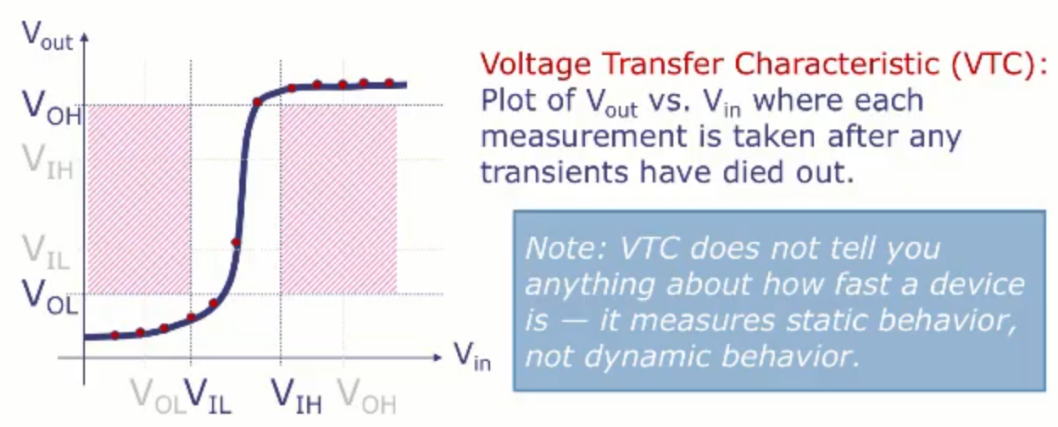


Figure 2: Buffer VTC

One interesting observation is that in the center of the input range (i.e. invalid input ranges), a small change in the input causes a large change in the output. This means that there has to be some kind of amplification in the device.

Also note that the VTC is allowed to do whatever it wants for invalid inputs. So the below VTC is also a valid VTC for a buffer. It is not a VTC that you would want to have, but it is valid.

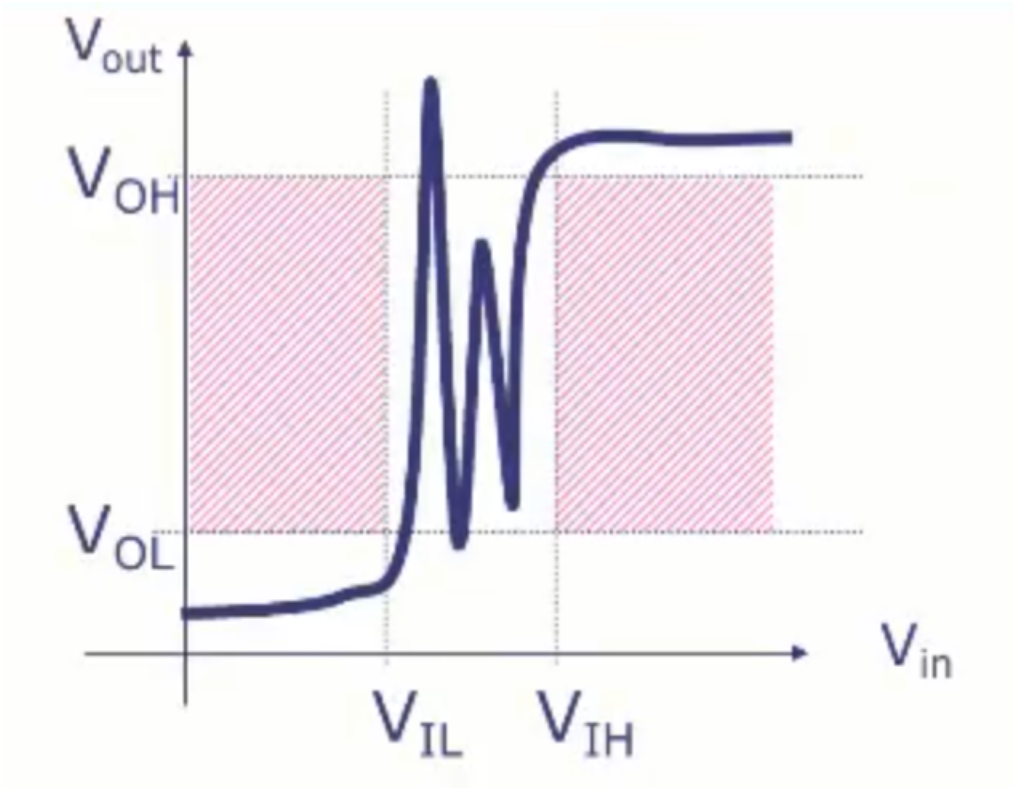


Figure 3: Buffer VTC still valid

The next few lectures (lecture 2-5) talk about combinational circuits, which are circuits where the output only depends on the current inputs, and not the past inputs/outputs. These circuits have no memory. Lectures 6-8 will talk about sequential circuits, which have memory and the current output depends on current input and past inputs/outputs.

Lecture 2 - Combinational Devices and Boolean Algebra

Combinational Devices

A combinational device has to satisfy the following criteria in order to be a valid combinational device:

1. It has to have one or more digital inputs
2. It has to have one or more digital outputs
3. It has to have a functional specification that details the value of each output for every possible combination of valid inputs
4. It has to have a timing specification consisting at minimum of a propagation delay (t_{pd}), which is the maximum amount of time the device needs to produce a valid output on receiving valid inputs

These criteria are called the static discipline. Moreover, a combinational device consisting of many devices also has to satisfy the following:

1. Each sub-device has to be combinational
2. Every input to a sub-device must either be driven by a single previous output or a constant 1 or 0
3. The circuit should have no cycles

Functional Specifications

There are many ways to give a functional specification of a combinational device. The two main ones are:

- Truth tables
- Boolean expressions

When implementing the functional specification of a circuit, we usually convert the truth table representation to a boolean expression, and implement that. However, there are many ways to optimize a standard sum-of-products type boolean expression into a minimal form that uses the fewest gates, so we use tools to synthesize circuits. These tools take as input the high-level circuit specification (in an HDL language), the set of available gates to be used, and the optimization goals (area, delay, power, etc.), and return an optimized circuit.

> **Note**
> The NAND and NOR gates together are universal gates, which means you can represent any boolean expression using just those two gates.

In CMOS technology, inverting gates (NAND, NOR, NOT) are faster and smaller than non-inverting gates (AND, OR).

MIT 6.006 - Introduction to Algorithms

Taught by Prof. Erik Demaine, Prof. Srini Devadas in Fall, 2011

Course Description

This course provides an introduction to mathematical modeling of computational problems. It covers the common algorithms, algorithmic paradigms, and data structures used to solve these problems. The course emphasizes the relationship between algorithms and programming, and introduces basic performance measures and analysis techniques for these problems.

Course website
Lecture Videos (YouTube)

Lectures

1. Algorithmic Thinking, Peak Finding
2. Models of Computation, Document Distance
3. Insertion Sort, Merge Sort
4. Heaps and Heap Sort
5. Binary Search Trees, BST Sort
6. AVL Trees, AVL Sort
7. Counting Sort, Radix Sort, Lower Bounds for Sorting
8. Hashing with Chaining
9. Table Doubling, Karp Rabin
10. Open Addressing, Cryptographic Hashing
11. Integer Arithmetic, Karatsuba Multiplication
12. Square Roots, Newton's Method
13. Breadth-First Search
14. Depth-First Search, Topological Sort
15. Single Source Shortest Paths Problem
16. Dijkstra
17. Bellman-Ford
18. Speeding up Dijkstra
19. Dynamic Programming 1: Fibonacci, Shortest Paths
20. Dynamic Programming 3: Parenthesization, Edit Distance, Knapsack

Lecture 1 - Algorithmic Thinking, Peak Finding

1D Peak Finding

In a one dimensional case, peak finding is trivial. You just parse through all the elements in the array, look to the left and the right element, if both of them are lesser than the current element, we have found a peak. For the edges of the array we just check the one neighbor.

This algorithm is clearly O(n).

A faster algorithm would use binary search. We jump to the middle of the array, compare the left and right neighbors, and recurse on the half of the array which has the higher number. If both the elements are smaller than the middle element, we found a peak. We can prove the correctness of this algorithm, and its time complexity is clearly O(log n). This recursive algorithm returns one peak, and not all.

For this problem, and most others in this course, O(log n) is the optimal complexity.

2D Peak Finding

In two dimensions, we define a peak if all the neighbors (left, right, top, bottom) of a cell are less than or equal to the current cell. You could also define a peak to be strictly less than the current cell, which makes more sense to me.

An efficient and correct algorithm for a 2D peak finder is as follows.

Start with the middle column in the 2D matrix, and find the global peak on the single column. This is O(log n) complexity. We then look to the elements on the left and the right of the global maximum, and we recurse on the half-matrix which has the higher element. If both the left, and the right elements are smaller or equal to the global max, we have found a peak, and we return.

The recurrence for the time complexity of this algorithm is -

$$T(n, m) = \Theta(n) + T(n, m/2)$$

Where the matrix has n rows and m columns. Solving this recurrence gives us the time complexity of this algorithm as O(n log m). The proof of correctness is left as an exercise, and as a question in the problem set. Intuitively, it makes sense because it follows the same pattern as the recursive algorithm for 1D peak finding.

Lecture 2 - Models of Computation, Document Distance

Models of Computation

A model of computation gives you the set of operations that you can perform in constant time. It is like the set of axioms or physical laws that govern our algorithm's execution. It gives you the set of operations you can use to run an algorithm, and is just the mathematical analog of a computer. There are 2 main models of computation - the random access machine, and the pointer machine. The random access machine gives you constant time access to any memory location in an array. The pointer machine is simpler and doesn't give you random access.

Random Access Machine

The random access machine model gives you a constant number of registers (O(1)). In O(1) time, you can -

- Access or load O(1) machine words
- Perform O(1) computations on words that are already loaded
- Write or store O(1) machine words

A word is the quantum of data you can store in one memory location, usually 64 bits as of the 2020s.

Pointer Machine

A pointer machine model corresponds very closely to OOP. It gives you dynamic memory allocation, and it consists of dynamically allocated objects which can have a constant (O(1)) number of fields. Each field either contains a word of data or contains a pointer to some other object.

A good example of an application of this model is an implementation of a linked list. In this model, in O(1) time, you can -

- Read or access a field of an object
- Follow a pointer
- Change a pointer
- Change a field if it contains a value

You can implement a pointer machine model using a RAM model.

Python Cost Model

Python implements both the random access machine, and the pointer machine models for different data structures. Some important properties of python's built in data structures are -

1. Accessing any element in a list takes O(1) time
2. Appending to a list uses table doubling and so takes amortized O(1) time
3. Adding 2 lists and returning takes linear time or O(|L1| + |L2|) time
4. Extending a list L1 with a list L2 takes O(1+|L2|) time (l1.extend(l2))
5. Retrieving the length of a list takes O(1) time because python keeps track of the length of a list as it's being built
6. The in operator takes linear time
7. The sorted() function and List.sort() takes O(nlogn) time, and uses a comparison sort (quick sort)
8. Inserting a key into a dictionary takes O(1) time
9. Checking if a key is in a dictionary takes O(1) time

Document Distance

We define a document as a sequence of words, and a word as a sequence of alphanumeric characters. A document does not have to be a huge list of words, any list of words is a document (for our purposes in this lecture). The document distance problem is about finding if two given documents are "similar."

We can have different definitions of similarity, and if you implement a document distance function you may want to choose a definition that fits your use case, but for now we can call two documents similar if they have a lot of words in common. We can set the threshold for the exact number of common words for two documents to classify as similar dynamically.

There are a lot of motivations for solving this problem, most of which involve searching, duplicate matching, and now also machine learning and NLP.

We find the distance between two documents using a vector representation of the documents. We represent each document as an n dimensional vector, where n is the number of words in your dictionary. Each element of the vector represents the frequency of that word in that document. So we represent document D1 as an n dimensional vector - $< d_1, d_2, d_3, ..., d_n >$ where di is the number of times the ith word in the dictionary appeared in D1. We then calculate the angle between the two documents by taking the inverse cosine of the dot product of the two document vectors and define the angle as the similarity. A smaller angle means closely similar documents and a larger angle means dissimilar documents.

Implementing this algorithm has 3 steps - 1. Split the document into words 2. Count the frequency of the words 3. Compute the dot product and the angle

The first two steps are the slowest steps, and asymptotic as well as constant factors in the different algorithms you could write to do these steps will affect performance dramatically. Eric tested 8 different implementations, and the difference between the least efficient and the most efficient algorithm was a 1000x improvement.

P.S - Don't use the `re` module unless you absolutely need to use regexes, and when you do, be sure to know the time complexities of the methods you use, since the `re` module has a lot of functions that need exponential time to run.

Lecture 3 - Insertion Sort, Merge Sort

Insertion Sort

Insertion sort is an $O(n^2)$ algorithm. The pseudocode for insertion sort is -

1. Consider you have already sorted a prefix of the array A[:i-1] and are considering key i.
2. Take key i and insert it into the right place in the sorted array A[:i-1] by doing pairwise swaps with adjacent elements.

For example, if you had the array $A = [5, 3, 4, 1, 2, 6]$ and you were somewhere in the middle of your sorting process -

$$A = [3, 4, 5, 1, 2, 6]$$

If you were considering the key 1, then you would first swap its place with 5, then compare with 4, then swap it with 4 (since 1 belongs to the left of 4), then compare 1 with 3, then swap it with 3, then stop since 1 would be in the right place. At the end of this step you would get

$$A = [1, 3, 4, 5, 2, 6]$$

You would then do the same thing for 2 and 6.

Complexity

Since insertion sort requires an entire iteration through the array, and since in each step of the iteration you may have to do $\Theta(n)$ compares and swaps, insertion sort is a $O(n^2)$ algorithm.

One optimization we could perform is to the number of comparisons. To find the right place in which we have to insert the current key, we could use binary search to find the right index and then do pairwise swaps to insert it into the right place. This would reduce the complexity w.r.t. the number of comparisons but we would still need to do a linear number of swaps to get the key into the right place.

This still doesn't change the asymptotic time complexity of insertion sort, because we still have to do $O(n)$ swaps for each step. So we have a total of $O(n \cdot log\, n)$ comparisons but we have a total of $O(n^2)$ swaps. This helps you improve the total runtime if comparing two items in the array is expensive (like one or more database lookups), but if comparison and swapping are both $O(1)$ operations, insertion sort remains a $O(n^2)$ algorithm.

Merge Sort

Merge sort is a recursive sorting algorithm. Instead of sorting an array as is, it splits the array into 2 halves, sorts them, and then merges them. Of course, it doesn't explicitly sort the two half arrays either - it splits them into two more arrays, tries to sort them, and merges them together.

The pseudocode for merge sort is - 1. Given an array, split it into 2 half-arrays. 2. If the arrays are not of size 1, split them further and recurse on both halves of the array. 3. Else, the two half-arrays have been sorted (since they only have 1 element), combine them together by running the merge routine.

The pseudocode for the merge routine is - 1. Consider the first element in both the sorted arrays. 2. Add the minimum of the considered elements to the merged sorted array. 3. Remove the added element from the half-array.

```
## Untested, rough merge sort implementation
def merge_sort(arr):
    if len(arr) == 1:
        return arr
    left = arr[:len(arr)//2]
    right = arr[len(arr)//2:]
    return merge(merge_sort(left), merge_sort(right))

def merge(left, right):
    i, j = 0, 0
    ans = []
    while True:
        if i >= len(left) and j >= len(right):
            break
        elif i >= len(left):
            ans.append(right[j])
            j += 1
        elif j >= len(right):
            ans.append(left[i])
            i += 1
        else:
            if left[i] > right[j]:
                ans.append(right[j])
                j += 1
            else:
                ans.append(left[i])
                i += 1
    return ans
```

Complexity

Merge sort involves dividing the arrays $log_2(n)$ times, and the merge routine for each division takes $\Theta(n)$ steps. Therefore, merge sort is a $\Theta(n \cdot log\, n)$ algorithm. More formally, we can write the recurrence for merge sort as -

$$T(n) = \Theta(1) + \Theta(n) + 2 \cdot T(n/2)$$

The $O(1)$ comes from splitting the array into two halves, and the $O(n)$ comes from the work required to merge the two half-arrays after they have been sorted. Solving this recurrence using the techniques given in 6.042J lecture 15, we get the time complexity of merge sort to be $O(n \cdot log\, n)$.

One disadvantage of merge sort is that it is not in place. To sort an array you have to essentially create a sorted copy of it in memory, which may not be possible if the array is too big to fit twice into memory. In this case, insertion sort may be preferred, since even though it is $O(n^2)$, it is in place. There does exist an in place version of merge sort, but it doesn't do too well on the constant factors, so we prefer quick sort, or any other in place algorithm instead of merge sort.

Heaps and Heap Sort

Heaps

Heaps are an implementation of a priority queue. Heaps can do the following operations -
1. Insert(x) - Insert element x into the heap 2. Max(x) - Return the max element in the heap 3. Extract_max(x) - Return the max element and remove it from the heap

Heaps are just an array visualized as a nearly complete binary tree. We map an array to a binary tree by setting the first element as the root, then the next element as the first child, the next as the right child, and so on, in order. So for any node i in the tree, its parent is node i/2, its left child is node 2i, its right child is node 2i + 1.

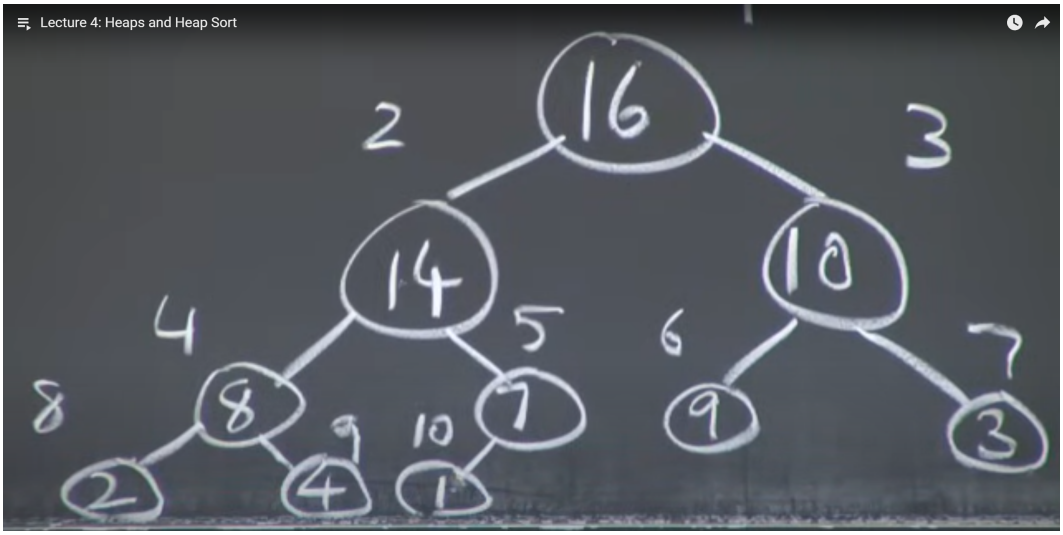


Figure 4: Heap representation of an array

There are 2 types of heaps - min heaps and max heaps. Min heaps have the property that the root of their tree representation is the smallest element in the array, and the key of every node i is less than or equal to the keys of its children. Max heaps have the property that the root of their tree representation is the largest element in the array, and the key of every node i is greater than or equal to the keys of its children.

For heaps to be useful, we need the heaps to maintain their max/min property as we perform extract max or extract min on them. We also need a function to build a max or min heap from a normal array.

Heap Methods

1. max_heapify() or min_heapify() - Assumes that the heap violates the max/min property at exactly one node and fixes it. More specifically, it assumes that

the violation occurs at the root of a subtree, and that the subtree rooted at the violation satisfies the max/min property.

Max or min heapify starts at the violation, compares the violating element with its children, and then swaps the violating element with the bigger child if max, smaller child if min, and then calls itself on the node with which it was just swapped to see if it didn't create a new violation. Max/min heapify takes in 2 arguments - the tree visualization of the array, and the index in the tree at which the violation occurs.

2. build_max_heap() or build_min_heap() - Builds a max/min heap given a normal array. The pseudocode for build_max_heap() of build_min_heap() can be written as -

```
def build_heap(A, n, max = True):
    for i in range(n/2, 1, -1):
       if max:
           max_heapify(A, i)
       else:
           min_heapify(A, i)
    return A
```

The complexity of max_heapify or min_heapify is $O(log\, n)$, which isn't hard to see because for one call of max/min_heapify, we may be doing at most $log(n)$ swaps and recursions. The complexity of build_heap() is $O(n)$, since we call max/min_heapify n/2 times, but each call of max/min_heapify does not take $O(log\, n)$ time. The upper limit of one call of max/min heapify at a level of the tree is log(i), where i is the level of the tree, which is not n. Doing a careful analysis and taking the sum of the complexities of max/min heapify over the levels, we find that the complexity of build_heap() is linear.

Heap Sort

Since we can retrieve the max or the min of the array in constant time, we can use this property to sort a given array. We first build a min heap from the array. Then, while there are elements left in the heap, we take the root element (i.e. the smallest element), swap it with the last element (i.e. the largest element). We then remove this last element of the heap from the heap and add it to our sorted list. But we have now violated the min heap property of our heap, since the root element is now the biggest element. So we run min heapify on our heap. Repeat.

The complexity of this algorithm is $O(n \cdot log\, n)$, since building a min heap is $O(n)$, but we extract n elements, and do min heapify each time, which is $O(log\, n)$. So our complexity is $O(n \cdot log\, n)$.

Binary Search Trees, BST Sort

Scheduling problem

Imagine we have an airport with 1 runway that needs to schedule the landing and take off times of a lot of planes. We start at t = 0, and increase time monotonically. No plane can land or take off k minutes within any other plane. We have to write an algorithm to take as input a landing/take off time and insert it into a table of landing/take off times if it is a valid time.

There are a variety of data structures we could use to solve this problem, but the optimal data structure is a binary search tree. A BST can solve this problem in $O(log\, n)$ time, while an unsorted or even sorted array needs at least linear time. This is because it would take us linear time to check if the given time is valid for an unsorted array, and it would take us log n time to check for validity but would take us linear time to insert for a sorted array. Heaps also don't work because finding the insertion point would take $O(n)$ time.

BST Specification

A BST structure has nodes. Each node has a parent pointer, a left child pointer, and a right child pointer. A BST maintains the invariant that any descendant node in the left subtree of an ancestor has a key less than or equal to the key of the ancestor, and any descendant node in the right subtree of an ancestor has a key greater than or equal to the key of the ancestor.

BSTs are useful because they let you perform fast insertion into a sorted array. They support all operations (search, insert, delete, pop, etc) in $ O(log,n) $ time.

However, a vanilla BST doesn't always solve the scheduling problem with $O(log\, n)$ time. It solves the problem in $O(h)$ time, where h is the height of the tree. In some cases, h can be $log\, n$, but as you keep on inserting into the tree, there is a high probability that h will be much greater than $log\, n$ and approach n. This still leads to an $O(n)$ complexity in the long run. However, there are some operations that are still useful on a BST.

BST Methods

1. find_min() - finding the minimum element of the tree takes $O(h)$ time, just keep going towards the left child starting from the root. The leftmost child of the tree (which is a leaf) will be the minimum element of the tree.
2. find_max() - same as find min, just keep going right.
3. next_larger() - finding the smallest element larger than a given x. Do binary search. $O(h)$ time.
4. next_smaller() - same as next_larger. $O(h)$ time.

AVL Trees, AVL Sort

AVL Trees were the original way to keep BSTs balanced.

The height of a tree is defined as the length of the longest path from the root of the tree down to a leaf. The height of a node is defined as the length of the longest path from the node down to a leaf.

The height of a node is 1 + max(height of left child, height of right child).

The invariant in AVL Trees is that the height of the left and the right subtrees of any node differ by at most 1.

In the worst case, when one subtree is always 1 deeper than the other subtree for all nodes in the tree, we want the height of the tree to still be $O(log\, n)$. In the worst case, the number of nodes of an AVL tree of height h, N_h, is

$$N_h = 1 + N_{h-1} + N_{h-2}$$

This is because the number of nodes at height h is 1 + the number of nodes in the left child and the right child. One of the children has a height h-1 and one has a height h-2. We can solve this recurrence to get an expression for n, but we don't need to. This recurrence is going to be larger than the fibonacci recurrence at each step, so we can use the fibonacci recurrence's closed form as an approximation.

$$Nh > F(h)$$

$$F(h) = h/5$$

$$Nh = n > h/5$$

$$h = 1.44 * log_2\,(n)$$

$$h = \Theta(log\, n)$$

So we call an AVL tree a balanced tree since $h = \Theta(log\, n)$.

AVL Insert

There are 2 steps to insert a key into an AVL tree - 1. Find the location where you want to insert using binary search. 2. Insert and then fix the AVL property that we just violated.

We fix the AVL property after we insert by using rotations.

Rotations

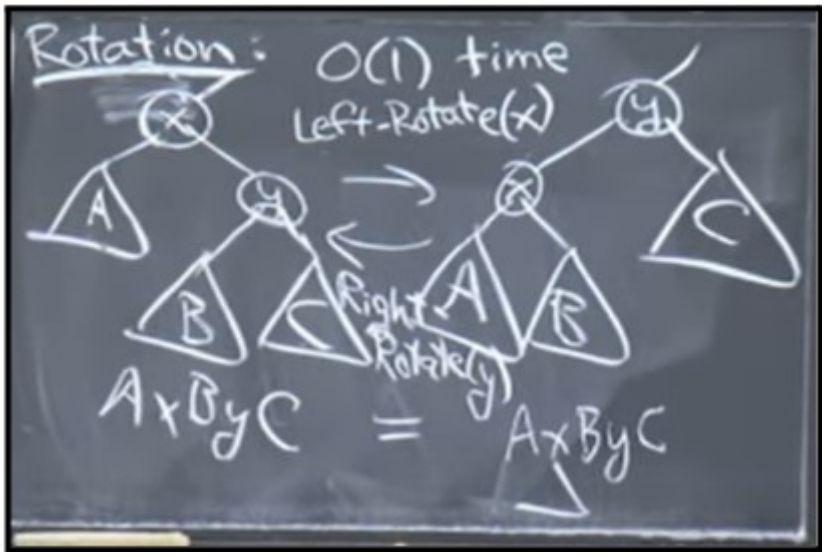


Figure 5: AVL rotations

If you insert a node and end up getting a straight line path from the node which violates the AVL property to the newly added node, then we can fix it in one rotation. If we insert a node and get a zig-zag path from the node which violates the AVL property to the newly added node, we first rotate the middle node in the zig-zag path, which results in a straight line path as in the first case, which we can fix in one rotation.

Fixing the AVL Property

Let us assume that the highest node that violates the AVL property is x (in the diagram above), and that x is doubly heavy towards the right. Now we have 2 cases. x's right child can be either right heavy or balanced, or x's right child can be left heavy.

If x's right child is right-heavy or balanced, then we can fix the AVL property by doing left rotate(x) (Eric wrote RR(x), which is wrong. It should be LR(x)). This is

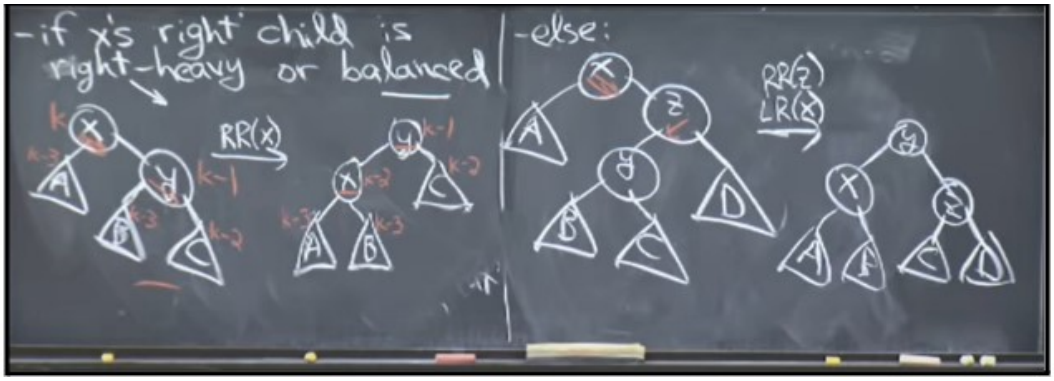


Figure 6: Fixing the AVL property

the simple one rotation case.

If x's right child is left-heavy, we first right rotate x's right child, which makes it balanced and brings it into case 1, and then we left-rotate x.

Counting Sort, Radix Sort, Lower Bounds for Sorting

Lower bounds for sorting

We can represent the execution of an algorithm as a decision tree. The leaves of the tree are the endpoints or the "answers" of the algorithm, and the path to a leaf is one execution of the algorithm. In this case, we can define the worst case running time of an algorithm to be the height of the tree.

Using this representation, we can prove that we need at least $O(log,n) $ time for searching and at least $O(n \cdot log_2 n)$ time for sorting (except for some special cases in which we can sort in linear time).

The number of leaves in the tree for a searching algorithm is at least n, since in a data structure containing n items, the item to search for can be in any of the n different spots. For a binary tree containing at least n leaves, the height of the tree has to be at least $log_2 n$.

Similarly for sorting, the number of leaves has to be at least $\Theta(n!)$. So the height of the tree has to be at least $log_2 (n!)$, which, if you calculate, is at least $\Omega(n \cdot log\, n)$.

Counting Sort

For sorting n integers in `range(k)`, when k is not too big, we can sort in linear time.

Let there be n items that you want to sort. Initialize a list L of length k, where each element in the list is an empty list, and k is the maximum key that can be assigned to any item (using a hashing function). The key is assigned by a key() function which maps the given item to a positive integer.

Go through the n items you want to sort one by one, convert them into their keys, and append the item itself to the list L in the index given by the key. Then go through the list L and keep concatenating the lists that you see.

```
def counting_sort(A, key):
    """
    A is the array to be sorted, key is a function that maps the
    elements in A to a non-negative integer.
    """
    L = [[] for i in range(k)]
    for j in range(n):
        L[key(A[j])].append(A[j])

    output = []
    for k in range(k):
        output.extend(L[k])
```

```
return output
```

The time complexity of this algorithm is $O(n + k)$, most of the time being taken up by the last for loop (since extend takes linear time). If k is $O(n)$, then counting sort is practical, if k gets even a little bigger than n, then counting sort starts becoming sub-optimal.

Radix Sort

Radix sort can deal with a bigger range of k than counting sort. Radix sort requires k to be $O(n^{O(1)})$, i.e. k can be to the order of a polynomial in n, and it uses counting sort as a subroutine.

The setup is the same. Let there be n items you want to sort. Visualize whichever integer the key function will give you as an integer in some base b. In this representation, the number of digits in the base b number is $log_b\, k$.

We then consider only the least significant bits of the keys and use counting sort to sort the items. We then consider only the next most significant bit and use counting sort to sort the items. We do this for all the digits in the representation, which is $log_b\, k$ iterations of counting sort. Therefore, the time complexity of this algorithm is $O((n + b) \cdot log_b\, k)$. To get the minimum complexity, we want $(n + b) \cdot log_b\, k$ to be minimum, which is when we choose b = n. When b = n and k is $O(n^{O(1)})$, the complexity reduces to

$$O((n + b) \cdot log_b\, k) = O((n + n) \cdot log_n\, n^{O(1)}) = O(n + n) = O(n)$$

Which is linear time.

Hashing with Chaining

Dictionaries

Dictionaries in python are an implementation of hash tables. A hash table is an ADT which has the following specifications - 1. Insert a value at a given key - should be $O(1)$ time 2. Delete a key from the table - should be $O(1)$ time 3. Search if a key exists in the table - should be $O(1)$ time

Hash tables are used in a variety of applications from cryptography, to document distance, to programming compilers, and much more. Given the specifications above, one naive way to implement hash tables is to use a normal array as the data structure, limit the keys to be non-negative integers, and treat the keys as indices into the array. This would give us constant time search, insert, and delete.

There are 2 problems with this approach - the keys are limited to non negative integers, and that if the possible space of keys is huge but there aren't many items in the dictionary, then a huge amount of space is wasted. There is also a bottleneck when initializing a giant array in memory, but that problem is secondary.

We can solve the first problem by just using a hash function. Python has a `hash()` function which returns the prehash of its argument. By default, if the argument is an integer, it just returns that integer. If the argument is a custom object, then it uses the `__hash__()` magic method to compute the hash. If the `__hash__()` method is not implemented then it just returns the `id` of the object, which is its position in memory.

Hash functions should not change value over time. This means that the hash of a value should not depend on time. This also means that you cannot use mutable objects (like lists) as keys in python dictionaries, because there is a good chance that the list you are using as a hash changes value. This solves the first problem. This is actually prehashing.

To solve the second problem of unused space, we use hashing. Let the space of all possible keys be K. Let the number of elements/keys that exist in the hash table be n. We can then limit our array to be of some size m such that $m = \Theta(n)$. This would limit the space that the array takes up in memory so that we no longer hog space, and it would make sure that we still have some room to insert and delete items. However, since there are now only m spaces in our hash table, the hash function will need to map K keys to m slots, which means that there will be a lot of collisions since K » m. We can solve this problem in 2 ways - by chaining or by open addressing.

Chaining

Chaining is a way to deal with collisions that occur during hashing. If the hash function gives you 2 keys that map to the same address, we just store both of the

keys in a linked list. All the items that map to the same address are just stored in a linked list (or normal list/array) at that address.

One disadvantage of chaining is that you could get really unlucky and map all of the keys that you have into the same address, which would reduce your entire hash table to just a single list. However, the probability of that happening is satisfactorily small. A good hash function randomizes things enough that this almost never happens, so in practice, the length of the list at each address is constant.

Simple Uniform Hashing

Simple uniform hashing is an assumption we make about our hashing function to analyze the complexity of the operations we can perform on our hash table. Simple uniform hashing states that any key has an equal chance of being hashed to any address in the table and the hash of any key is independent of the hash of the other keys.

This assumption is not completely true, in particular the second part about the hash function being independent, but the analysis that we can do using this assumption is correct.

If there are n items in the table, and there are m slots in the table, then the average length of the linked list at each address is n/m. If $m = \Theta(n)$, then $n/m = \Theta(1)$. This number is called α, the load factor of the table. As long as α is $\Theta(1)$, hashing with chaining is a good idea, since the complexity of insert and delete would still be $O(1)$, but the complexity of search is $O(1 + \alpha)$, which is constant if α is $\Theta(1)$.

Hash Functions

A hash function does not convert any arbitrary object into an integer. That is the pre-hash function. A hash function maps keys from the giant possible space of keys K to a small subset of keys m. It takes as input an integer which could be any integer from the set K and outputs another integer in set m. A few example of hash functions are -

1. Division method - Just take the modulus(m) of the integer you are given.

```
def division_hash(x, m):
    return x % m
```

This gives you an integer between 0 and m. This method is not too good because the performance of this hash function depends on our choice of K and m. It performs well in practice when m is prime or at least not a power of 2 or 10, but this solution does not seem so robust.

2. Multiplication method - The multiplication method feels a lot more like hashing.

```
def multiplication_hash(x, m):
    a = os.environ.get('a')
    w = os.environ.get('w')
    r = math.log2(m)
    return ((a*x) % 2**w) >> (w-r)
```

Here a is any random number that we choose for our hash function, and w is the word size for our machine. 'r' is log2m. The reason this works is as follows - a·x first gives us a number that is 2 words long (since multiplying two words will give us a number that is two words long). This number is randomized because a is random. Taking the modulus to the base 2w gets rid of the upper half bits. Shifting the number to the right by (w-r) bits leaves us with r bits, which is a number between 0 and m-1.

3. Universal hashing - Universal hashing is used in practice and is a good hash function.

```
def universal_hash(x, m):
    a = os.environ.get('a')
    b = os.environ.get('b')
    p = os.environ.get('p')
    return ((a*k + b) % p) % m
```

Here p is a big prime number, larger than the size of your universe of keys, and a and b are any random numbers between 0 and p. Using linearity of expectation and a similar technique which we used to prove that simple uniform hashing gave us a reasonable hash function, we can prove that universal hashing gives us a O(1) time hash table implementation.

Table Doubling, Karp Rabin

A key piece we are missing if we want to complete our implementation of hash tables is our choice of m. We will map from the set of all keys K, which could be infinite, to a small subset m, and we want to choose an m such that the load factor α remains O(1), but we also don't want to make m too big and waste space. We want m to be $\Theta(n)$, where n is the number of elements in our hash table.

This is where table doubling comes in. We choose a small size for our table, say 8. We set m to be 8. Then, as elements keep coming in and going out, we double the size of our table if it gets completely occupied. In general, if we want to grow the table from size m to size m1, we need to follow 3 steps -

1. Allocate a new table to size m1
2. Choose a new hash function h1
3. Rehash all items from the old table of size m to the new table of size m1

The time we need to perform this operation is O(n + m + m1), which is O(n). Now, if we double the size of the table during an insert, that insert is going to take O(n) time and not O(1) time. However, since we only double the size of the table log2n times as we go from a table of size 0 to n, we can distribute the cost of doubling over all n inserts and calculate the average time needed to insert n items into the hash table.

Amortized Cost For Inserts

You can think of the amortized cost kind of like the average cost of n operations. One way to calculate it is by redistributing the cost of all operations uniformly. For example, if you have to pay a monthly rent, you usually pay nothing for most days of the month and pay all of it at once on one day of the month. This is kind of how most inserts are O(1) and only a few need O(n) time. What you can do instead is redistribute your rent and instead of paying nothing for most days of the month, you can pay a smaller amount every day of the month. The total cost would be the same, and you would call the rent you pay each day to be the amortized cost of one day (operation).

The cost of doubling the table is O(n), and only happens log2n times. So the total cost of table doubling for n insertions is O(1 + 2 + 4 + 8 + ... + n), which is O(n). The total cost of the insertions themselves is O(n). So the total cost of inserting n items into the hash table is O(n)/n, which is O(1). Hence we can say that inserting into a hash table takes constant time amortized.

We use a similar logic for deletions. If we delete items such that m = 4n, we halve the size of the table to be 2n. Using the same amortized analysis, the cost for a deletion is O(1) amortized.

This is how lists work in python. They use table doubling and table halving to provide constant time append and pop operations. So appending to a list or removing the last item from a list are actually O(1) time amortized.

Table Doubling for Real Time Applications

For some high performance real time applications, spending O(n) time for a few operations is not ideal and can lead to spikes in performance. One way to prevent this is to keep track of the size of your table. As soon as you notice that you are starting to reach the threshold where you would want to double the size of the table, you start to build a table twice the size on the side. Then, every time you insert, you copy over some constant amount of items over to the new table. This constant needs to be big enough to be able to copy over all of the items by the time you need to double.

While this technique is not used too often in practice since it is hard to implement robustly, it does allow you to get O(1) insertions and deletions non-amortized.

String Matching & Karp Rabin

String matching is a common problem, you want to search for a string s in a corpus or document t. The naive algorithm of checking all possible substrings of length s from the corpus is something like this -

```
def search(s, t):
    return any(s == t[x:x+len(s)] for x in range(len(t)-len(s))
```

The == operator in python checks for the equality of two strings by comparing all characters of the strings and returns true if all of them match. This is clearly a $ O(|s|) $ operation, which we do $ O(|t|) $ times. So the cost of searching for a substring match this way is at least $ O(|t||s|) $, which is polynomial time. This would be really bad if we were searching for a big substring in a big corpus. We would like to bring this time down using hashing.

If, instead of comparing two strings character by character, we could hash them down to the size of a word, we will be able to compare two strings in constant time. However, hashing a string of length s still takes $ O(|s|) $ time, so we're not saving any time using this approach. To fix this, we use a rolling hash function. We make use of the fact that when we are checking for substring matches in a corpus, we are effectively checking each substring of the corpus by sliding a "window" of size $ |s| $ and checking if the substring of the corpus in that window matches our string. However, when we slide that window over by 1 character, we don't need to recalculate the hash value of the new string from scratch. We can use the information we had from the previous string, and define a deletion and an addition operation on the hash function that would let us compute the hash of a new string by deleting one

character from the front of the string and inserting a character at the end of the string.

Now we only need to design a data structure that would let us perform the above operations in constant time. This is called the Karp Rabin algorithm, and it runs with time complexity $ O(|s| + |t| + #matches*|s|) $ in expectation, #matches is the number of matches you want to report.

Open Addressing, Cryptographic Hashing

Open Addressing & Probing

Open addressing is another method of dealing with collisions in a hash table. Instead of simply chaining collisions, we use the notion of probing to compute a slightly different hash if the first hash caused a collision.

We modify our hash function, h, a little such that it now takes the universe of keys as well as the 'trial count', which is an integer between 1 and m-1, as input and returns an integer between 1 and m-1 as output. Also, we have to make sure that the set of hashed values {h(k,1), h(k,2), ..., h(k,m-1)} is a permutation of {0, 1, 2, ..., m} so that each item has an opportunity to be hashed into any slot of the table.

Inserting into this table would then involve first hashing the given key into a slot. If that slot is occupied, we hash again by incrementing the trial count, otherwise we insert the item into the slot we found. We store each item in the table as a tuple of the form (key, value), where key is the original key from the universe of keys, and value is the value to be stored at that key. This is to make search possible.

Searching for a key ks in this table would then involve two steps. We hash ks into a slot and look at the key stored in that slot. If that key is not ks, we hash ks once more by incrementing the trial count. This then hashes ks into a new slot. If we look at the new slot and find that ks is indeed stored in that slot, we return the value stored in that slot. Otherwise, if we find an empty slot, we conclude that ks does not exist in the table.

This approach would work if we were not deleting any keys. However, if we inserted a key into a slot after, say, the third trial, and we end up deleting the item stored at the key we hashed into in the second trial, search would wrongly return that the key we are searching for does not exist.

To fix this we introduce a "deleted" flag. When we delete a key from the table, we replace it with a `(deleted, value)` tuple instead of the `(None, value)` tuple. Then, during search, if you encounter a deleted flag you just keep going and treat it as an occupied slot. During insert, we treat the deleted flag the same as None and overwrite the value stored in that slot.

Choosing a Hash Function

We now need to choose a good hash function that will take the trial count as an additional input and produce an integer between 0 and m-1. It also needs to produce a permutation of {0, 1, ..., m-1} as the trial count goes from 1 to m.

A bad hash function to use would be

```
## m = size of table
```

```
def linear_probing_hash(x, i):
    return (some_other_hash(x) + i) % m
```

This just computes the hash of the key with a standard hash function, adds the trial count to it and takes its modulus to the base m. This approach is bad because it leads to clustering. What this will do in case of a collision is just hash the key (in subsequent executions) to the empty slot closest to the collision. Over time, this will lead to there being clusters of items near the very first collisions of the table, and these clusters will continue to grow. This is bad for load balancing of the table, and will lead to poorer search and delete run times.

A better choice of hash function is called the double hash function.

```
def double_hash(x, i):
    return (h1(x) + i*h2(x)) % m
```

Here h1 and h2 are ordinary hash functions. There is a restriction on h2 to make sure that you get a permutation of {0, 1, . . . , m-1}, which is that h2(x) and m should be relatively prime. To ensure this, we can define m to be 2i and h2(x) to be odd for all x. This function works pretty well in practice.

With this, if you also implement table doubling (which works a little differently for an open addressing table than a chaining table), you can prove that you can have O(1) insert, delete, and search time w.h.p, as long as the load factor is a small number (<0.5).

Lecture 11 - Integer Arithmetic, Karatsuba Multiplication

This lecture focuses on performing arithmetic operations on large integers. We first cover Newton's method for approximation, followed by Karatsuba multiplication.

Newton's Method

Newton's method is a method for finding the root of a function through successive approximation . It is based on the idea of linear approximation. Given a function $ f(x) $, we can approximate it's root using the following steps -

- Start with an initial guess $ x_i $
- Compute the tangent line at $ x_i $
- Find the x-intercept of the tangent line
- Repeat the process with the new x-intercept

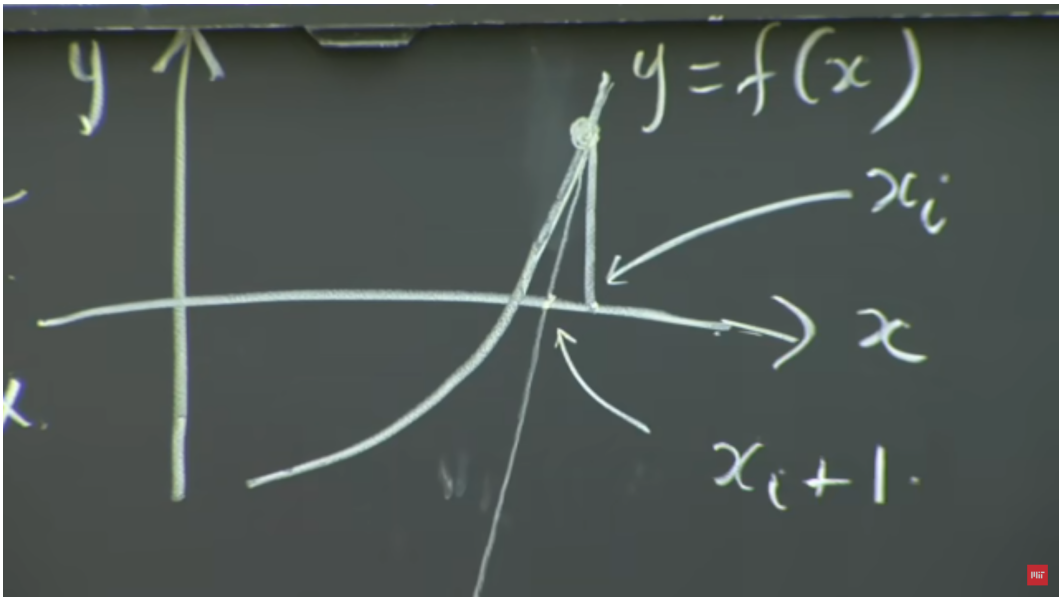


Figure 7: Newton's Method

The formula for Newton's method is given by -

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

$ x_{i+1} $ is an approximation of the root of the function $ f(x) $, which gets better on every iteration. Newton's method converges quadratically, which means that the number of correct digits doubles on every iteration. This means that to

get an approximation with $ n $ correct digits, we need to perform $ O(log,n) $ iterations.

Karatsuba Multiplication

Karatsuba multiplication is a fast multiplication algorithm that uses divide and conquer to multiply two numbers. It is defined as follows -

- Given two numbers $ x $ and $ y $, we can write them as -
  - $ x = 10^{n/2}a + b $
  - $ y = 10^{n/2}c + d $
- The product of $ x $ and $ y $ can be written as -
  - $ xy = 10^nac + 10^{n/2}(ad + bc) + bd $
- We can recursively compute the product of $ a $ and $ c $, $ b $ and $ d $, and $ (a+b) $ and $ (c+d) $ to get the final product.

Lecture 12 - Square Roots, Newton's Method

Lecture 13 - Breadth-First Search

BFS is a graph search algorithm that explores a graph one "level" at a time. It starts at some pre-defined start node and explores all the neighbours of that node. Then it moves on to the neighbours of the neighbours and so on. It is a very simple and intuitive algorithm that is used to find the shortest path in an unweighted graph. It runs in $O(V + E)$ time, where V is the number of vertices and E is the number of edges in the graph.

This is a simple implementation in Python:

```
def BFS(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            queue.extend(graph[node])
    return visited
```

This is the Python code covered in the lecture that gives you the "level" of each node as well as the actual path in the form of parent pointers:

```
def BFS(start, graph):
    level = {start: 0}
    parent = {start: None}
    i = 1
    frontier = [start]
    while frontier:
        next = []
        for u in frontier:
            for v in graph[u]:
                if v not in level:
                    level[v] = i
                    parent[v] = u
                    next.append(v)
        frontier = next
        i += 1
    return level, parent
```

{:.language-python}

Lecture 14 - Depth-First Search, Topological Sort

Depth-First search is a fundamental algorithm for traversing or searching tree or graph data structures. It starts at the root node and explores as far as possible along each branch before backtracking. It does visit all nodes reachable from a vertex, but tt does not find shortest paths. However, it has a few properties that make it useful in other applications.

```
def dfs_visit(graph: dict[str, list[str]], start: str, parents={}):
    """
    Recursively visit all nodes in the graph starting from the given node.
    """
    for neighbor in graph[start]:
        if neighbor not in parents:
            parents[neighbor] = start
            dfs_visit(graph, neighbor, parents)

def dfs(graph: dict[str, list[str]]):
    """
    parents = {}
    for node in graph:
        if node not in parents:
            parents[node] = None
            dfs_visit(graph, node, parents)
```

DFS Edge Classification

DFS can be used to classify edges in a graph. There are four types of edges in a graph: 1. **Tree edges**: Edges that are in the DFS tree (formed by the parent pointers). 2. **Back edges**: Edges that point from a node to one of its ancestors in the DFS tree. 3. **Forward edges**: Edges that point from a node to one of its descendants in the DFS tree. 4. **Cross edges**: Edges that do not fall into any of the above categories.

These edges can be used to detect cycles in a graph. If there is a back edge in the graph, then there is a cycle in the graph. This gives us a simple algorithm for cycle detection. We just run DFS on the graph and check if there are any back edges.

Topological Sort

Topological sort is an ordering of the nodes in a directed acyclic graph (DAG) such that for every edge (u, v), u comes before v in some ordering that we care about. Topological sort can be used to solve a variety of problems, such as scheduling tasks with dependencies, and solving linear programming problems. For task scheduling

with dependencies, it makes sure that all tasks that depend on other tasks are only done after the tasks they depend on are completed.

All we have to do to get a topological ordering is to run DFS on the graph and output the nodes in reverse order of when they finish. This is because the nodes that finish last are the ones that have no outgoing edges, and thus can be done first.

Lecture 15 - Single Source Shortest Paths Problem

This lecture discusses the single source shortest paths problem and presents the general strategies algorithms follow to solve it.

The single source shortest paths problem is defined as follows: given a graph $G = (V, E)$ and a source vertex $s \in V$, find the shortest path from s to every other vertex in V.

General Strategies

Given that your graph does not have any negative weight cycles, there is a general approach to solve the single source shortest paths problem:

```
Initialize the distance of the source vertex to 0 and the distance of all other vertices
Repeat until no more updates are possible:
    For each edge (u, v) in E:
        Relax the edge (u, v)
```

"Relaxing" an edge means updating the distance of the destination vertex v to the minimum of its current distance and the sum of the distance of the source vertex u and the weight of the edge (u, v).

```
if d[v] > d[u] + w(u, v):
    d[v] = d[u] + w(u, v)
```

Dijkstra

Graph Notation

- s - The start vertex
- d[v] - The current shortest path distance from s to v
- Π[v] - The predecessor of v on the shortest path from s to v
- δ[u, v] - The actual shortest path length from u to v
- w(u, v) - The weight function w gives us the weight of the path going from u to v, given that (u,v) $\in$ E, the set of edges

Relaxation of Edges

We "relax" an edge if we find a shorter path to the vertex the edge is pointing to than we currently know.

We look at an edge (u, v). If we find that the path from u to v is shorter than d[v], we update the value d[v] to be this new shorter path. We keep doing this for all edges and eventually, we end up with the shortest path to every vertex v starting from the start vertex s.

```
Relax(u, v, w):
    if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w(u, v)
        predecessor[v] = u
```

The algorithm we talked about in the last lecture just picked edges to relax randomly, but that algorithm does very poorly complexity-wise. Keeping the general idea the same, if we are a little smarter about how we pick the edges we want to relax, we get a much better algorithm (Dijkstra or Bellman-Ford).

The General Algorithm

The general algorithm works only for directed acyclic graphs which may or may not have negative edges. The algorithm works in 2 main steps

1. Sort the graph topologically using depth first search.
2. If we use DFS to sort topologically, we can say that a shortest path exists from u to v iff u comes before v in the ordering that DFS gives us.
3. Represent the sorted graph linearly and choose a start vertex.
4. Go through the sorted linear graph in order from left to right, relax each edge you encounter as you go, and by the end, you have found all the shortest paths.
5. Every vertex to the left of s gets a δ(s, v) of ∞.

The depth first search takes O(V + E) time, and we only look at each edge once, which takes O(E) time, so our general algorithm takes O(V + E) time.

Dijkstra's Algorithm

Dijkstra is a greedy algorithm, and it only works for undirected graphs without negative edges. Dijkstra starts with a vertex, relaxes all the edges emanating from it, and assumes that the shortest path to that vertex has been found.

```
Dijkstra(G, w, s):
    solved = set() # S contains the set of solved vertices
    Q = priority_queue()
    Q.add(V) # Q contains the set of vertices of the graph
    d = dict() # d will contain the d values of all the vertices as we go on relaxing th

while Q:
        # While there are still elements in Q, do
        u = extract_min(Q) # return the element with the lowest priority in Q and remove
        solved.add(u) # Assume that u has been solved
        for v in Adj[u]:
            relax(u, v, w)
```

The priority queue Q will be ordered by the d values in the dictionary d. We implement the priority queue using a min-heap, and calculating the cost of operations required in the implementation above gives us a complexity of $\Theta(V \cdot log\, V + E \cdot log\, V)$. If we implement the priority queue using a fibonacci heap, we get a complexity of $\Theta(V \cdot log\, V + E)$.

Bellman-Ford Algorithm

Bellman-Ford should be able to- 1. Correctly calculate the (u,v) values for all vertices v that are not part of negative cycles and can be reached without encountering a negative cycle. 2. Mark all vertices which are part of a negative cycle or can be reached from a negative cycle as having a (u,v) value of undefined. 3. Mark all the vertices that cannot be reached from the start vertex as having a (u,v) value of infinity.

```
bellman_ford(G, w, s):
    for i in range(1, len(V)-1):
        # For all the vertices in the graph except the first and the last, do
        for edge in E:
            # For each edge in the set of edges
            # assume the edge goes from u to v
            relax(u, v, w)

    # Check for negative cycles
    for edge in E:
        # If we can relax further, that means that a -ve cycle exists, and we have just
        if d[v] > d[u] + w(u, v):
            print("Negative weight cycle(s) exist")
            return False
```

The first nested for loop would be Θ(VE) and the second for loop would be Θ(E), so the overall dominating complexity is Θ(VE).

Proof of Correctness

Let v be a vertex in the graph G, and let a path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ be a simple shortest path from s to v. Then $k \leq |V| - 1$, because any simple path cannot visit a vertex more than once, and there are $|V|$ vertices in the graph.

Now, during every iteration of the outer for loop, we relax all the edges in the graph. This means that after the first iteration, we relax the edge $\langle s, v_1 \rangle$ at some point. This means that we have found the shortest path from s to v_1, since it is a subset of a shortest path from s to v, which means it is also a shortest path by itself. Similarly, after the kth iteration, we will have found the shortest path upto v_{k-1}. This means that by $|V|$ iterations, we will have found the shortest path from s to v.

However, there could be negative weight cycles in the graph. To detect these, we run one more iteration. If we notice that we can still relax an edge, that means that there is a negative weight cycle in the graph. This is because if we can still relax an edge, it means that the shortest path from s to v can be made shorter by going through the negative weight cycle. In that case, we fail and stop.

Speeding Up Dijkstra

We can perform some optimizations to our dijkstra implementation to improve the average runtime. The worst case complexity remains the same, but the average case complexity goes down.

Optimization 1 - Early Stopping

We add one check to vanilla dijkstra -

```
Dijkstra(G, w, s):
    solved = set() # S contains the set of solved vertices
    Q = priority_queue()
    Q.add(V) # Q contains the set of vertices of the graph
    d = dict() # d will contain the d values of all the vertices as we go on relaxing th
    while Q:
        # While there are still elements in Q, do
        u = extract_min(Q) # return the element with the lowest priority in Q and remove
        solved.add(u) # Assume that u has been solved
        for v in Adj[u]:
            relax(u, v, w)
        # t is our destination vertex
        if u == t:
            return
```

If you know what the destination vertex is, you don't need to keep relaxing edges after you are done with the destination vertex. So you return if you are done with the destination vertex.

Optimization 2 - Bidirectional Search

Instead of the search for shortest paths going in just one direction, we search for shortest paths in both directions. We do one step of forward search starting from s, then we do one step of backward search starting from t. In each step we do just what dijkstra usually does. We stop when we find a vertex that has been searched both forwards and backwards.

To implement this search, we need to maintain 3 more data structures. We have Q, d[v], and Π for the forward search like we did before, but now we also have another Q, d[v], and Π for the backward search. We call them Qf and Qb, df and db, and Πf and Πb. Qf is the processing queue for vertices going in the forward direction, Qb is the same going in the backward direction. Πf is the predecessor relationship dictionary going in the forward direction and Πb is the same going in the backward direction. df[v] is the length of the current shortest path to v in the forward direction and db[v] is the same for the backward direction.

Once we terminate, i.e. find a node t that has been processed both ways, we start to find the shortest paths. The shortest path might not always be the path joining s to w to t. Once we terminate, we must check all the vertices who have a finite db[v] and df[v] value, and check to see if they form a shorter path than $s \to w \to t$.

Lecture 19 - Dynamic Programming 1: Fibonacci, Shortest Paths

Dynamic programming is a very powerful algorithm design paradigm. Dynamic programming can be thought of as a smart way of applying brute force. Dynamic programming breaks a problem into smaller sub-problems, solves the sub problem, and uses the same solution to solve the higher problem. It is well suited to optimization problems.

`Dynamic programming = recursion + memoization + guessing`

Guess an algorithm and all the possible steps for that algorithm. Guess all possible algorithms and then take the best one.

Fibonacci Numbers

A naive algorithm to calculate fibonacci numbers is

```
def fibonacci(n):
    if n <= 2:
        f = 1
    else:
        f = fibonacci(n-1) + fibonacci(n-2)
    return f
```

This is a very inefficient algorithm because the recursion is exponential. The time complexity of this algorithm can be written as -

$$T(n) = T(n-1) + T(n-2) + O(1)$$

The solution to this complexity is exponential, $\Theta(2n/2)$, which is pretty bad. A better way to do this is by dynamic programming.

Memoization

Memoization is basically storing the solutions to smaller problems in a data structure (usually a hash table) and using memory to speed up computation.

Calculating fibonacci numbers using memoization can be implemented as -

```
memo = {}
def fibonacci(n):
    if n in memo:
        return memo[n]
    if n <= 2:
        f = 1
    else:
```

```
        f = fibonacci(n-1) + fibonacci(n-2)
        memo[n] = f
    return f
```

Looking at the dictionary makes sure that we don't repeat calculations we have already done. You can do this with any recursive algorithm. You may not be able to do this in contexts, however, because of memory limits, but you can still store a few examples to help your computation.

It is easy to see how this optimization helps our computation time. For example, if we draw a recursion tree, it would look something like this -

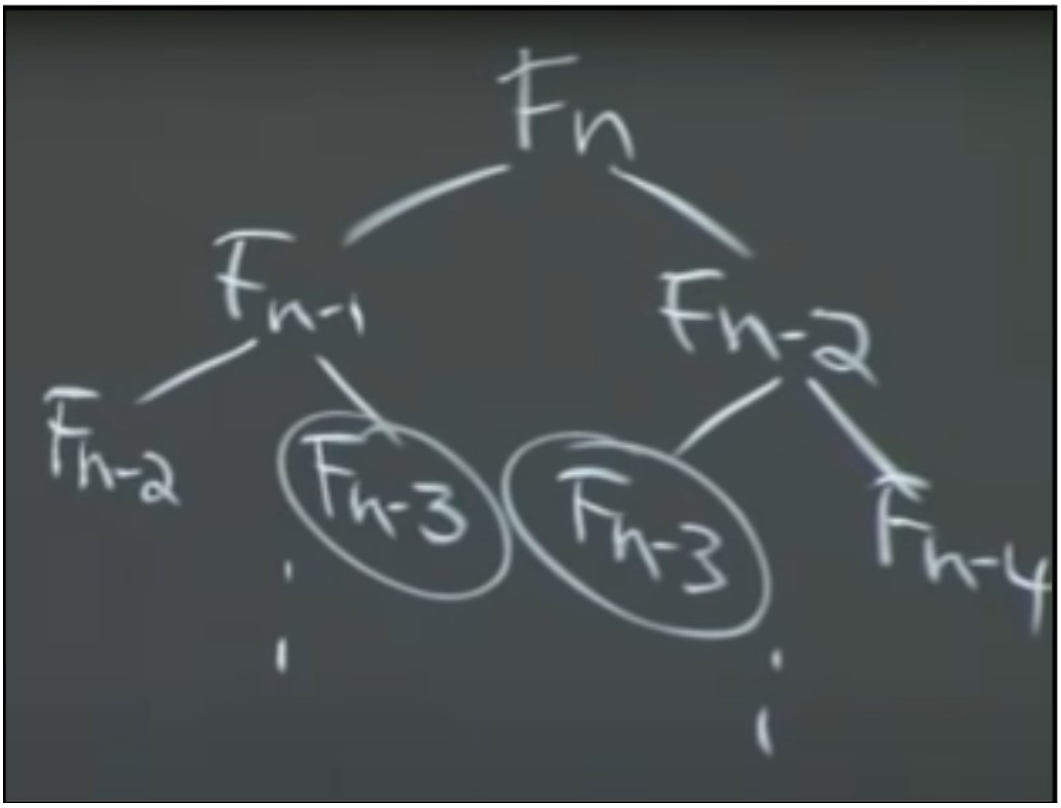


Figure 8: Memoization recursion tree

To compute Fn, we have to compute Fn-1 and Fn-2 However, once we have computed Fn-2, we should already have computed Fn-3, and therefore we should be able to compute Fn-1 in constant time if only we remembered the result of Fn-2 and Fn-3. This means we can completely drop one half of the recursion tree (the right half).

Another way of thinking about it is that Fn recurses only the first time it is called, and every other time it returns in constant time after the dictionary lookup. So we

make n calls to compute Fn, and each call takes constant time (because we ignore the recursion). Therefore, the time complexity of F is now $\Theta(n)$. This is a general technique used to calculate the total cost for any DP with memoization.

DP Cost With Memoization

To see why each call takes constant time and why we can ignore the time taken by recursive calls, imagine an execution of the algorithm.

If you want to compute Fn, you need to compute Fn-1 + Fn-2. We recurse once (on the Fn-1 call) and now we need to compute Fn-2 + Fn-3. We recurse again (on the Fn-2 call) and now we need to compute Fn-3 + Fn-4, and so on. We eventually get to the base case of n=2, which immediately returns 1. Now we move up one level in the recursion tree, which requires us to compute F3, which we can return in constant time since we have F2 and F1 in the memo table (and also by definition). We move up one more level and now we need to compute F4, which also returns in constant time because we have F3 and F2 in the memo table. Similarly, F5, F6, ..., Fn all return in constant time. Therefore, we had n recursive calls or subproblems, and each sub problem takes O(1) to solve.

This is a general method to calculate the running time of a dynamic program.

```
Running Time = #subproblems * time per subproblem
```

When calculating the time per subproblem, we ignore all recursive calls and assume they take constant time, as we shown above.

Another paradigm of writing the fibonacci algorithm (and all dynamic programming algorithms), is to start from the bottom up.

```
fib = {}
for k in range(1, n+1):
    if k <= 2:
        f = 1
    else:
        f = fib[k-1] + fib[k-2]
    fib[k] = f
return f
```

Here, we start calculating from the bottom and go up, but this algorithm performs exactly the same calculations as the memoized algorithm. This version of the algorithm allows you to save space on your dictionary/list as well. Once you calculate a fibonacci number, you only need to store the last 2 fibonacci numbers to calculate the next one. So you don't need a big dictionary with fibonacci numbers for all the problems you have solved so far. The bottom up approach saves space compared to

the memoized approach. It's also obvious that the running time of this algorithm is linear.

Shortest Paths

Suppose we are trying to solve a single source shortest path problem from vertex s to vertex v. We can use naive recursion and reduce our current problem down to a smaller subproblem

1. Consider all edges going to v. The shortest path has to include one of them.
2. The edges going to v will all be connected to some vertex u. The shortest path will then have the weight $\delta(s, u) + w(u, v)$for all u connected to v.
3. The problem to be solved is now reduced from (s, v) to $min_{(u,v)inE}(\delta(s, u) + w(u, v))$.

But this algorithm does not work on graphs with cycles (we get stuck in an infinite loop).

The intuition is this - suppose you want to calculate the shortest path from s to v, $\delta(s, v)$. We know that the shortest path has to include one of the edges coming into v. So we choose a random edge coming into v, and assume that it is included in the shortest path. Then, we calculate the shortest path from u to that node instead of from u to v. We then do this for all nodes connected to v and take the minimum value.

$$\delta(s, v) = min_{(u,v)\in E}(\delta(s, u) + w(u, v))$$

```
def shortest_path(s, v):
    return min(shortest_path(s, u) + weight(u, v) for u in v.neighbours)
```

This is a $O(V + E)$ algorithm. This algorithm is essentially doing a depth-first search.

Lecture 20 - Dynamic Programming 2 - Text Justification, Blackjack

5 General Steps To Writing a Dynamic Program

Lecture 21 - Dynamic Programming 3: Parenthesization, Edit Distance, Knapsack

Dynamic programming is a powerful algorithm design paradigm that can be used to solve a variety of problems efficiently. In this lecture, we will explore three important applications of dynamic programming: parenthesization, edit distance, and the knapsack problem.

Parenthesization

The parenthesization problem involves determining the optimal way to parenthesize a sequence of matrices to minimize the total number of scalar multiplications required to compute the product. The key idea is to use dynamic programming to find the optimal split points for the matrices.

Consider you have n matrices $A_1, A_2, \ldots, A_n$ with dimensions $d_0 \times d_1$, $d_1 \times d_2$, $\ldots$, $d_{n-1} \times d_n$. The cost of multiplying two matrices A_i and A_j is given by the product of their dimensions: $d_i \times d_j \times d_k$, where k is the index of the matrix that splits the multiplication. We want to find the optimal way to parenthesize the product of these matrices to minimize the total cost.

For example, if you have matrices A_1, A_2, A_3 with dimensions $n \times 1$, $1 \times n$, and $n \times 1$. Then it is better to multiply A_2 and A_3 first, as it results in just one scalar, and then multiply that scalar with A_1.

Let $DP[i, j]$ be the minimum cost of multiplying matrices from A_i to A_j. The recurrence relation can be defined as follows:

$$DP[i, j] = \min_{i<k<j} (DP[i, k] + DP[k, j] + cost(A_i, A_k) + cost(A_k, A_j))$$

The base case is when $i = j$, where the cost is zero since there is only one matrix.

MIT 6.006 - Introduction to Algorithms

Taught by Prof. Erik Demaine, Dr. Jason Ku, and Prof. Justin Solomon

Course Description

This is an introductory course covering elementary data structures (dynamic arrays, heaps, balanced binary search trees, hash tables) and algorithmic approaches to solve classical problems (sorting, graph searching, dynamic programming). Introduction to mathematical modeling of computational problems, as well as common algorithms, algorithmic paradigms, and data structures used to solve these problems. Emphasizes the relationship between algorithms and programming, and introduces basic performance measures and analysis techniques for these problems.

Course Website Lecture Videos (YouTube)

Lectures

1. Algorithms and Computation
2. Data Structures and Dynamic Arrays
3. Problem Session 1
4. Sets and Sorting
5. Hashing

Lecture 1 - Algorithms and Computation

Lecture 2 - Data Structures and Dynamic Arrays

Lecture 3 - Sets and Sorting

Lecture 4 - Hashing

Problem Session 1

Refer to the problems here.

Problem 1-1

(a)

$$(f_1, f_5, f_2, f_3, f_4)$$

This is because $(\log n)^a = o(n^b)$ for any value of a and b (proved in recitation).

(d) Using Stirling's approximation, $n! = \Theta\left(\sqrt{n}\left(\frac{n}{e}\right)^n\right)$.
Using Stirling's approximation in the expansion of $\binom{n}{n/2}$, we get $\binom{n}{n/2} = \Theta\left(\frac{2^n}{\sqrt{n}}\right)$.

$$(\{f_5, f_2\}, f_3, f_1, f_4)$$

Problem 1-2

Obvious solution.

MIT 6.042J - Mathematics For Computer Science

Taught by Prof. Tom Leighton and Dr. Marten van Dijk

Course Description

This course covers elementary discrete mathematics for computer science and engineering. It emphasizes mathematical definitions and proofs as well as applicable methods. Topics include formal logic notation, proof methods; induction, well-ordering; sets, relations; elementary graph theory; integer congruences; asymptotic notation and growth of functions; permutations and combinations, counting principles; discrete probability. Further selected topics may also be covered, such as recursive definition and structural induction; state machines and invariants; recurrences; generating functions.

Course Website Lecture Videos (YouTube)

Lectures

MIT 6.045J - Automata, Computability, Complexity

Taught by Prof. Scott Aaronson

Course Description

This course provides a challenging introduction to some of the central ideas of theoretical computer science. It attempts to present a vision of "computer science beyond computers": that is, CS as a set of mathematical tools for understanding complex systems such as universes and minds. Beginning in antiquity, with Euclid's algorithm and other ancient examples of computational thinking, the course will progress rapidly through finite automata, Turing machines and computability, decision trees and other concrete computational models, efficient algorithms and reducibility, NP-completeness, the P versus NP problem, the power of randomness, cryptography and one-way functions, computational learning theory, interactive proofs, and quantum computing and the physical limits of computation. Class participation is important, as the class will include discussion and debate about many of these topics.

Course website
Lecture Videos (YouTube)

Lectures

1. Introduction
2. Logic, Circuits, and Gates
3. Finite State Automata, Regular Languages
4. Regular Expressions, Context-Free Grammars
5. Turing Machines
6. Decidability
7. Turing Recognizability, Oracles
8. More Reducibility, Dovetailing, Godel

Lecture 1 - Introduction

What is Computer Science?

This course justifies why computer science deserves to be called a "science". It doesn't give you any skills that you can apply in your daily job, or your projects, but it introduces you to some Great Ideas in Computer Science.

It starts with a set of rules that we assume to be true, just as we assume axioms to be true in mathematics, and shows what we can and cannot do with this set of rules. It shows us the limits of what is possible and what is definitely impossible.

Factoring is Hard

The first "idea" we look at is that factoring a composite number into its prime factors is hard, at least with our current computational model. Factoring will be easy with a quantum computer. For now, the best we can hope to do is spend exponential time finding the factors, and the best algorithm we know takes $ O(2^{cN/3}) $ time to find the factors, where N is the number of digits in the number.

However, checking if a number is prime or composite is much easier - possible to do in polynomial time. This is called primality testing, and we have primality testing algorithms that run in around $ O(n^3) $ time.

These two ideas are the foundation of today's cryptographic systems, and if we are able to build a true quantum computer, that will basically be the end of all privacy.

Euclidean Geometry

Compass and straightedge geometry can be considered the first model of computation in theoretical computer science. The course explores this model as an example of what we will be doing further along with other, more modern models.

The compass and straightedge geometry model says that given any two points on a plane, we can perform the following operations on them -

1. Draw a line through them
2. Bisect the line
3. Extend the line arbitrarily
4. Draw a circle with the line segment as a diameter

Furthermore, you can mark the intersections of any two curves as a new point, and use that point for further constructions.

Using this model of computation, we can reduce it to the following model -

Starting with the numbers 1 and 0, perform the following operations on them -, +, x, ÷, and come up with new numbers. The limits of this model are then seen in

the numbers (or types of numbers) that we can't come up with. For example, we can come up with 2 by using the pythagorean theorem in the plane. However, we can't come up with 32 using just 1s, 0s, and the above operations. The proof of this requires Galois theory, but it is a proven limitation.

This example is more mathematical than computational, but this illustrates the type of reasoning we perform in this course. We start with a set of well defined rules, and then extend them to find out exactly what we can and cannot do, as well as give reasons as to why we definitely cannot do something, regardless of how clever we are or how much computational power we have.

Lecture 2 - Logic, Circuits, and Gates

Gates and Universality

This lecture introduces the fundamental logic gates - AND, OR, and NOT, and then talks about their universality.

Universality in this case means that we can construct any boolean function using a combination of these gates. In computer science, universality is the norm, and most sets are usually universal. You have to work a little to get a set that is not universal. Some examples for non-universal sets in logic gates are - {AND, OR}, and {XOR, NOT}.

This is because AND gates and OR gates are monotonic functions. That is, changing one of the inputs from a 0 to a 1 can only change the output from a 0 to a 1, or leave it unaffected. Whereas to represent a NOT gate, we would need to change the output from a 1 to a 0 when the input went from 0 to 1, which is not possible to do with monotonic gates.

An example of a universal set of gates is {AND, OR, NOT}. To prove that this set is universal, we need to prove that we can use these gates to produce an arbitrary boolean expression of any number of bits. We can do this using a truth table.

For any arbitrary boolean expression, draw a truth table or all possible input combinations. Then, only consider the input combinations which output a 1. For each such input combination, build a circuit that would output a 1. That is, if one input combination that outputs a 1 for 3 inputs is $ {X, Y, Z} = {0, 0, 1} $, then we would build a circuit for this combination as $\neg X \wedge \neg Y \wedge Z$.

Once we have circuits for each such row (that outputs a 1 in the final boolean expression), we just OR all of those circuits to get our expression.

This is the same technique I learned in college. However, this is pretty inefficient, because it produces a circuit of size $O(n \cdot 2^n)$. The size of a circuit is the same as the number of gates in the circuit. That's pretty bad, so we need to improve our approach, which we will do shortly.

Another set of universal gates we can use is {AND, NOT}. This is because we can simulate the OR gate using an AND gate and a NOT gate. Another set of universal gates is {NAND} . We can use the NAND gate to simulate any other gate, hence it is universal (so is the NOR gate). But these sets don't help us either, because using the same approach as above with these sets of universal gates would only affect the size of the circuit by a constant factor.

Gates and P vs NP

Now you might wonder if there is a bound on the number of gates we will need to represent any arbitrary boolean expression of n bits. That is, if we have any arbitrary boolean function of n bits, what is the smallest size of the circuit we can build to represent it? Using the approach above, we definitely know that it can be done in at most $O(n \cdot 2^n)$ bits, but can we come up with a better bound?

Proving that a specific circuit definitely needs an exponential number of gates and it cannot be represented in any fewer is an open problem. If you think about it, it's the same as P vs NP. If you build a circuit to represent an NP-complete problem, you can't prove that it needs an exponential number of gates (at least no one has proven it yet), and proving it would be the same as proving $P \neq NP$.

However, we can prove that a majority of boolean expressions need a circuit of exponential size. Sure, there are some boolean expressions for which we can have as small circuit, but most require a large circuit. This was proven by Claude Shannon in 1949 using a counting argument.

The gist of the argument is as follows - we count the number of boolean expressions which take n bits as input, we count the number of circuits that we can build using n inputs of size S, and we compare the two quantities to realize how large S has to be to equal the number of boolean expressions with n inputs.

The number of boolean expressions that take n bits as input is 2^{2^n}. This is because for an n bit boolean expression, the truth table has 2^n rows. Now for each row, the output value could be 0 or 1, so the number of possible truth tables is 2^{2^n}.

Now, let's say we want to build a circuit of size S, which takes n bits as input. The number of circuits of size 1 which take n bits as input is just $nC2$, since any 2 input bits could be used as the input to a gate. Similarly, the number of circuits of size 2 is $\binom{n}{2} \cdot \binom{(n+1)}{2}$, since adding the first gate creates a new input to which we can attach the second gate. If we keep going, we find that the number of circuits of size S using n inputs is $\binom{n}{2} \cdot \binom{(n+1)}{2} \cdot ... \cdot \binom{(n+S-1)}{2}$. We can approximate this to be at most $(n+S)^{2S}$.

Now we want

$ (n+S)^{2S} = 2^{2^{n}} $

Which gives us

$S = \Theta(\frac{2^n}{n})$

This means that to represent a boolean expression with n inputs, most circuits will have to be of size $S = \Theta(\frac{2^n}{n})$.

Lecture 3 - Finite State Automata & Regular Languages

Finite Automata

A finite automaton is the same as a finite state machine. It has a finite set of states that it can be in, and it takes input from a set called its alphabet. It has a transition function that takes one alphabet and the current state as input and transitions to the next state.

The notation we will use for these quantities is the following -

Q | Finite set of states |
Σ | Finite alphabet |
$\delta : Q \times \Sigma \to Q$ | Transition function |
q_0 | Start state |
$F \subset Q$ | Accepted states |

We call

$$L = 0, 1^*$$

a *language*. In this case, a language is the set of all binary strings of length 0 or more. A language is just any set of strings. We say that a language is *accepted* or *recognized* by an automaton if the automaton ends up in an accepting state for all strings in the language, and we denote that as $L(M)$, meaning L is a language recognized by the finite automaton M. We say that L is a *regular* language if we can design an automaton to accept that language.

Regular languages have the following properties - 1. The intersection of regular languages is regular 2. Complement of a regular language is regular 3. Regular languages are closed under union, intersection and complement 4. All finite languages are regular

An example of a language that is not regular is the language of all palindromes. We can prove this using the pigeonhole principle. A finite state automaton has a finite number of states, and this language can have strings of infinite length. Since the amount of "memory" a finite automaton has is essentially equal to the number of states it has, it cannot store inputs whose length is more than the number of states it has. Therefore, we cannot design a finite state automaton for this language, so it is not regular. This was just a rough proof, but the lecture gives a detailed proof.

Nondeterministic Finite State Automata

A non-deterministic finite state automaton is an extension of DFAs. It has the property that it can have any number of ways to transition from one state to another, even when it reads the same symbol. That means that if it is in a state q_i, it can have any number of paths to follow when it reads the symbol 0, for example, whereas a DFA only has one path per symbol.

For example, we have the NDFA that accepts the language where the 3rd to last bit is a 1, which can be drawn as given below -

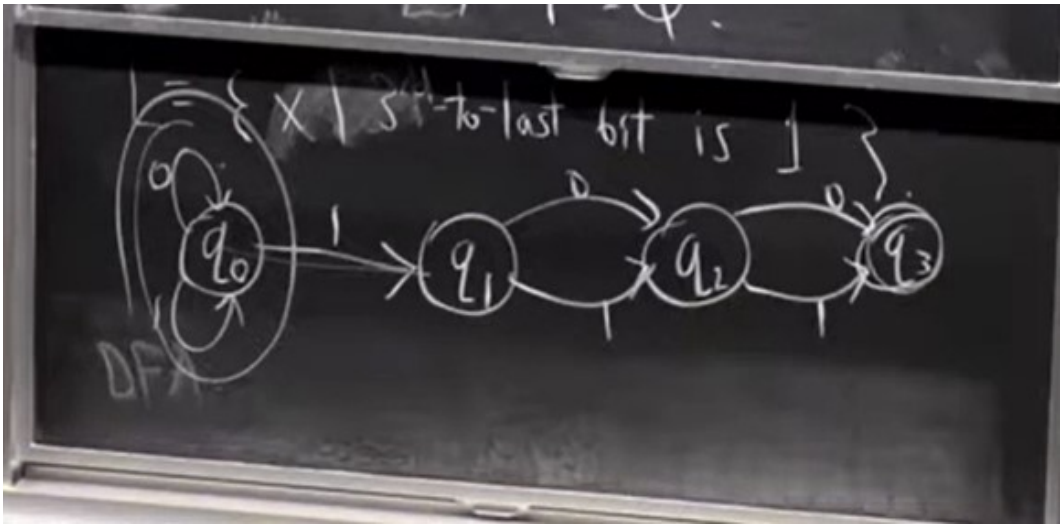


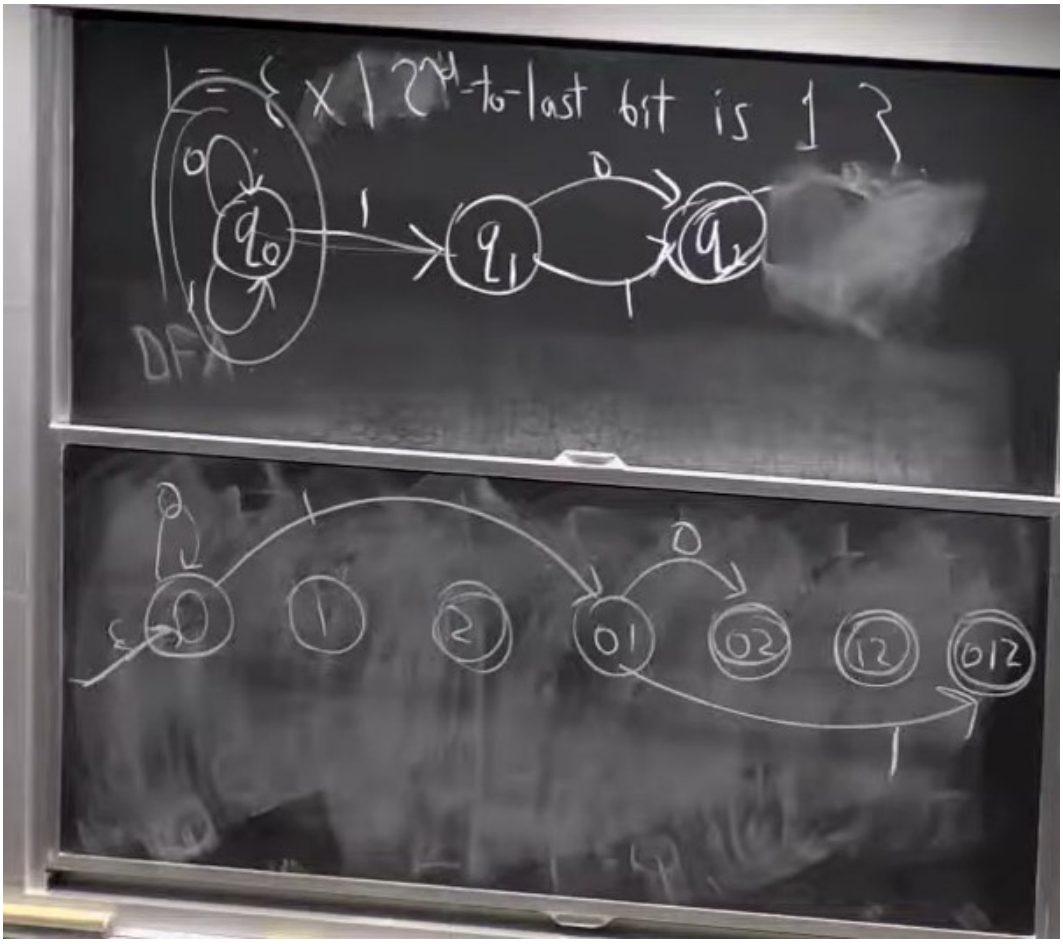
Figure 9: NDFA accepting language where 3rd to last bit is 1

It turns out that NDFAs are essentially equivalent to DFAs. For every NDFA that accepts a language L, we can construct a DFA that accepts the same language, and for every DFA that accepts a language L, we can construct an NDFA that accepts the same language.

We do this by power set construction. First of all, converting from a DFA to an NDFA is trivial. All DFAs are NDFAs. To convert from an NDFA to a DFA, we construct a power set of the states the NDFA can be in. For example, if the NDFA can be in states {0, 1, 2}, then we construct a power set {0, 1, 2, 01, 02, 12, 012} (we exclude the null element in the power set). We can then convert from an NDFA to a DFA by following transitions from the start state of the NDFA, and whenever we read a character that can transition to multiple states, we jump to that element in the power set that has all of those states.

For example, if we consider the same NDFA given above, and we start at the start state and read a 1, we can end up in either the start state itself (state 0) or state 1. So in the DFA we transition to state 01.

An example conversion is shown below for the NDFA that accepts languages where the second to last bit is a 1 to its corresponding DFA.


{x | 2nd-to-last bit is 1}
0
1
q0
q1
q2
DFA
0
1
2
01
02
12
012

Lecture 4 - Regular Expressions, Context-Free Grammars

Regular Expressions to DFAs

Regular expressions are basically equivalent to DFAs. To show this, we have to show that, given a regular expression, we can design a DFA that accepts the same language as the regular expression, and given a DFA, we can write a regular expression that accepts the same language as the DFA.

To prove the first part of the implication, regular expression $\rightarrow$ DFA, we first show that we can design an NFA to represent a regular expression. We then know how to convert from an NFA to a DFA.

Regular expressions have three important operations - union, concatenation, and the * operation (zero or more). Suppose we have a regular expression without any of these operations, then this regular expression can be represented by a simple NFA that looks like a simple path. If the regular expression is `0110`, for example, the NFA would just be a start node, followed by 4 other nodes, which it reaches if it reads the characters `0110` in sequence, the last of which is the accepting state.

So we can easily design an NFA for a regular expression that doesn't use any operations. Now we show that if any two regular expressions are combined with an operation, then we can combine the corresponding NFAs for the two expressions using a simple method. The methods are shown in the photo below -

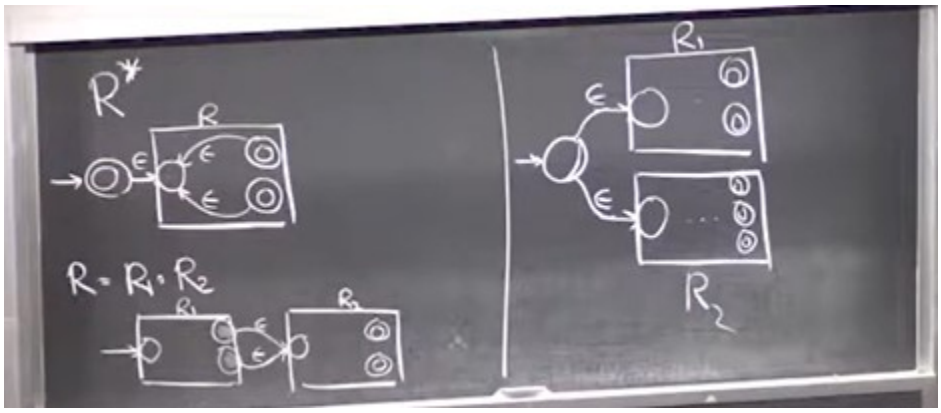


Figure 10: Converting from regular expressions to NFAs

If the expressions are combined using the union operation - `01|10`, then we just take the two NFAs for the two expressions and put a common start state before both of them with null transitions to the start states of the two NFAs. Think of this as similar to an OR gate.

If the expressions are concatenated, think of this as similar to an AND gate.

If an expression has the * operation, we add transitions from the accepting state back to the start state with null transitions, as shown above.

DFAs to Regular Expressions

To convert from a DFA to a regular expression, we first convert from a DFA to a GNFA. A GNFA stands for General NFA. A general NFA is just like an NFA, but its transitions function operates on regular expressions instead of an alphabet. So you can have regular expressions on the arrows between states.

For our purposes, we will make GNFAs satisfy the constraint that they only have one start state with only outgoing edges, and only one accepting state with only incoming edges. Once we convert from a DFA to a GNFA with some number of states k, we reduce the k states down to 2 (as shown in the lecture, pretty straightforward). Once we are left with 2 states in our GNFA, we can write it as a regular expression.

Context-Free Grammars

A context-free grammar is a set of rules for generating any string in a language. It can be defined by the following tuple - (V, Σ, R, S). V is the set of variables for the grammar, Σ is the alphabet of the language we are going to generate, R is the set of rules for generating the language, and S is the start variable from which we start generation.

An example of a context-free grammar is

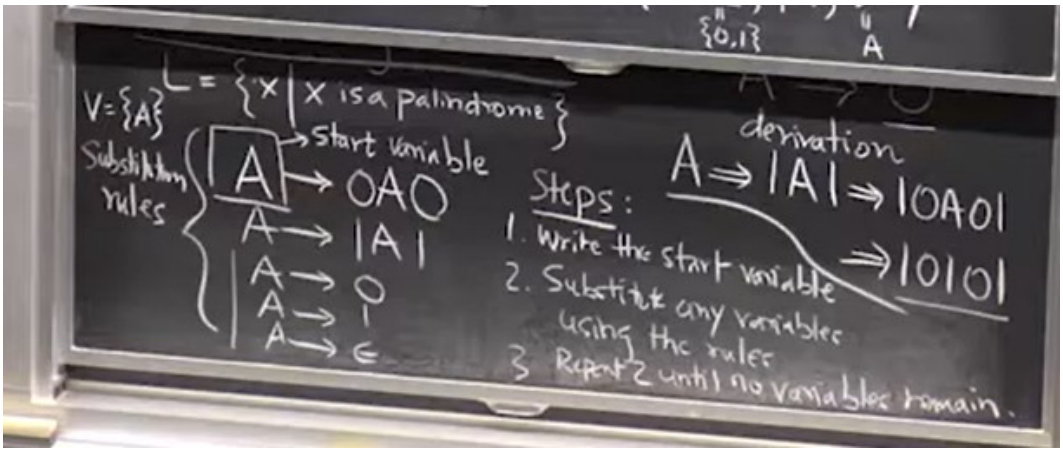


Figure 11: Context-free grammar

Languages that are generated from a context-free grammar are called context-free languages. As you might expect, since CFG is a more powerful model of computation as compared to regular expressions or DFAs, regular languages are a subset of

context-free languages. That is, every regular language is a context-free language (i.e. can be expressed with a CFG).

Pushdown Automata

Pushdown automata (PDAs) are a more powerful variation of NFAs. Think of them as basically NFAs with access to a stack onto which they can push items and pop items. They allow us to express a larger set of languages.

Even though NFAs are equivalent to DFAs, PDAs are not equivalent to their deterministic counterparts. That is, DPDAs $\neq$ PDAs.

Lecture 5 - Turing Machines

Alan Turing, in 1936, proposed a more powerful model of the finite automata, called the Turing machine (On Computable Numbers, A.M. Turing, 1936).

A Turing machine is a finite automata which has the ability to read a symbol and transition to another state, just like a DFA, but it can also write a symbol on the same paper it is reading from, and it can further move the paper it is reading from either to the right or to the left. It turns out that this model is powerful enough to be able to make arbitrary finite computations.

For example, we can have a Turing machine that accepts the language of palindromes. Let's say we have an infinitely long tape with an input. The start and end of the input are marked with special symbols, say `#`, so we know that the input has started or ended. Then we can design a Turing machine to accept the input only if it is a palindrome by the following (rough) algorithm -

1. Start in a some start state.
2. If you read a one in the start state, overwrite it with a `#` and keep moving to the right till you reach the end of the input.
3. If the last symbol read is also 1, replace it with a `#` and come back to the start of the input (the 1 that we overwrote with a #), and repeat.

Church-Turing Hypothesis

Now that we have such a powerful model of computation, you may wonder why not keep making it even more powerful. What if we have random access in our model of computation, or multiple tapes to read from and write to instead of just one. It turns out that all of these models of computation are no more powerful than a Turing machine, because they can be simulated by a Turing machine.

This is called the Church-Turing thesis. It is a hypothesis and doesn't have a proof, and could be disproved some time in the future, but for now we haven't found a counterexample.

The Church-Turing thesis says that any model of computation you can conceive that follows the physical laws of our universe can be simulated by a regular Turing machine.

The lecture gives two examples of such "reductions". A Turing machine that has a one-way infinite tape instead of a two-way infinite tape is equivalent to a two-way infinite tape Turing machine. A Turing machine that can read from two tapes simultaneously is equivalent to a regular Turing machine.

Computable Language

Finally, we define a language that a Turing machine can accept. Just as a language that was accepted by a DFA was called a regular language, and a language that was accepted by a context-free grammar was called a context-free language, a language that is accepted by a Turing machine is called a computable language.

Lecture 6 - Decidability

An important theorem regarding Turing machines is that a Turing machine can simulate other Turing machines, which can potentially be more complicated than itself. That is, a Turing machine A can act as an *interpreter* for Turing machine B, and B can be much more complicated than A.

Alan Turing actually gave such an example of an interpreter in his 1936 paper, but it turned out to have a mistake, so he had to publish a correction in 1937. Regardless, it can be done.

Halting Problem

A question you might have now is what can Turing machines not do? One famous example of a problem Turing machines can't solve is the halting problem - given a description of a Turing machine, return whether the machine will halt on an empty input.

For some Turing machines (and some computer programs), we can easily tell if the machine will halt or get stuck in an infinite loop. However, we cannot know that for sure for all Turing machines.

If we did know how to solve the halting problem, we would also be able to solve many unsolved problems in mathematics, like the Riemann hypothesis, or Goldbach's conjecture.

The proof that we can't solve the halting problem is given by contradiction as follows. Say you have a Turing machine P that takes the description of a Turing machine as input and tells you whether the input will halt on an empty input. This is not a special case, as we can always convert a Turing machine that takes some non-empty input into another Turing machine that takes an empty input but performs the same operations as the first Turing machine on the specific non-empty input.

Now, say we have another machine Q that takes the description of a Turing machine M, runs P on it, and goes into an infinite loop if M halts, and halts if M loops. If we pass the description of Q into Q itself, we get a paradox. Therefore, we contradict ourselves, and we cannot have a Turing machine P.

Busy Beaver & Infinities

Busy Beaver is a sequence that grows faster than any other countable sequence. Scott introduces it in the lecture as a way to introduce countability, sets of infinities, and prove that most languages are not countable.

The Busy Beaver sequence is defined as follows, `BB(n)` is the maximum number of steps an `n` state Turing Machine can take on an empty input before halting. Of

course, the maximum number of steps an `n` state Turing Machine can take is infinity, since it can just go into an infinite loop, so we restrict our discussion only to those Turing machines that halt on an empty input.

The first few values of the `BB` sequence are as follows -

`BB(1)` | 1 |
`BB(2)` | 6 |
`BB(3)` | 21 |
`BB(4)` | 107 |
`BB(5)` | >= 47,176,870 |
`BB(6)` | $>3 \cdot 10^{1730}$ |

We are not yet sure about the values of `BB(5)` and above. It turns out that this sequence grows faster than any computable sequence. We prove this by contradiction. Let's say that the Busy Beaver sequence was a computable sequence. Then we could design a Turing Machine to take an integer N as input and output `BB(N)`. However, if we knew `BB(N)`, we could use that fact to solve the halting problem. We could just run a Turing Machine (with N states), and if it took more than `BB(N)` steps, then we could be sure that the Turing Machine never halted.

Since we *can't* solve the halting problem, this is a contradiction, and the Busy Beaver sequence is not countable.

Scott then introduces the different types of infinities - countable and uncountable.

A countable infinity is any set that can be shown to have a one-to-one mapping with the set of integers. For example, the set of all even numbers can be shown to have a one-to-one mapping with the set of integers. This means that the set of all even numbers is countably infinite (interestingly, it also means that there are as many even numbers as integers). Similarly, the rational numbers are countably infinite.

Also, the set of all Turing Machines is countable. This is because each Turing machine can be represented as a series of transitions from each of its states, and all of these transitions can be encoded as a binary string. Since the set of all binary strings is just the set of all integers, the set of all Turing Machines is countable.

An important observation is that the set of all Languages is not countable. The lecture has a simple proof. This means that if we choose a language at random, the probability that a Turing Machine recognizes that language is 0%.

The set of countable infinity is called $\aleph$ (Aleph), and the set of uncountable infinities is called $2^{\aleph}$. In general, there is no largest set of infinity. You can always keep constructing sets that have more elements than the largest infinity you have by just taking the set of all subsets of that infinity.

Lecture 7 - Turing Recognizability, Oracles

Turing Recognizability

We now define a new kind of language, a Turing Recognizable language - very similar to a decidable language, with one key difference.

A Turing Recognizable language is a language for which we can construct a Turing Machine such that, for all inputs in the language, the Turing Machine halts and accepts, and for all inputs not in the language, the Turing Machine either halts and rejects or goes into an infinite loop.

The only difference between a decidable and recognizable language is the very last clause, which allows the machine to go into an infinite loop. This means that the halting problem can be phrased in terms of a Turing Recognizable language.

$$HALT = \{< M >, x : \text{M halts on x}\}$$

We can clearly see that for all strings in the language, we can construct a Turing machine to either halt and accept (just run M on x), or go into an infinite loop (just run M on x again, and it will go into an infinite loop if x is not in the language).

We have now seen a few classes of languages of increasing generality, shown in Figure 1.

As you would expect, the halting problem is in the set of recognizable languages but not in the set of decidable languages. Now if we define a new language, $\overline{HALT}$, which is the *complement* of $HALT$, it lies even outside the class of recognizable languages.

Reductions

Reductions are a way of solving a problem that you don't know how to solve using a solution to a problem that you do know how to solve. Reductions were seen before in 6.006 and 6.046. Reductions are seen everywhere in computer science, and are especially useful while proving the NP-completeness (or incompleteness) of some problem. Similarly, you can also use them to show that some problems are decidable or undecidable.

Formally, we represent a reduction as $A \leq_T B$, where the T stands for Turing reducibility. This denotes that problem A can be reduced to problem B, where A is the problem that we want to solve, and B is the problem that we know how to solve.

We can also write this as A is decided by M^B, where we call B an "oracle", because we get to assume that the Turing Machine M has access to Turing Machine B, which already knows how to solve its own problem

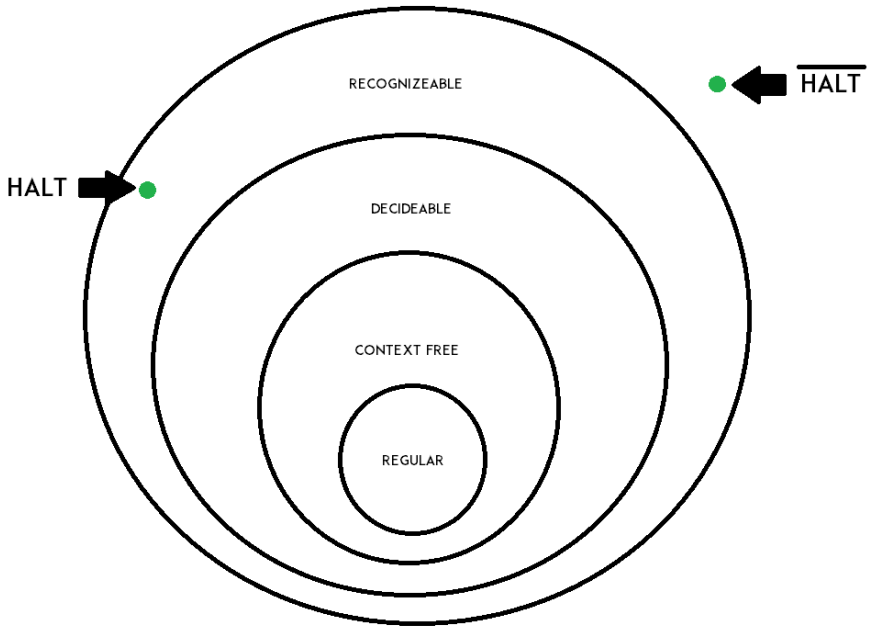


Figure 12: Classes of languages

In a Turing reduction, we can call the oracle any number of times, even zero. We are not forced to use the oracle. This is not the case for all kinds of reductions, for example in case of mapping reductions, you must call the oracle exactly once, and you must output the result of the oracle.

Mapping reductions can be a little more helpful that Turing reductions, because a mapping reduction between two problems means that the two problems are in the same class of languages, which is not guaranteed by a Turing reduction.

For example, you can come up with a Turing reduction to reduce $HALT$ to $\overline{HALT}$, but they are in complementary classes. However, we can come up with a mapping reduction from the halting problem with no input to the halting problem that takes any general input, which means that the two problems are in the same class of languages.

Fermat's last theorem was proved this way around 1995, when it was shown that that halting problem can be reduced to Fermat's last theorem using a mapping reduction. In other words, if you had an oracle that solved Fermat's last theorem, you could solve the halting problem.

Turing Degrees

A Turing degree is a set of problems that have the same difficulty. For example, the set of all decidable languages forms a Turing degree. The set of all undecidable problems also forms a Turing degree.

Each Turing degree has $\aleph$ languages. This is because each pair of languages in a degree can be reduced to each other, and you need a Turing machine to do reduce any problem to another. And since there are only $\aleph$ number of Turing machines, there can be at most $\aleph_0$ number of problems in a degree.

Another interesting fact is that there are problems even harder than the halting problem. To come up with a problem harder than the halting problem, consider a Turing machine that has access to an oracle that solves the halting problem. Now we ask whether this new Turing machine will halt on an empty input.

Using the same proof that we used to prove that the halting problem was undecidable (i.e, not solvable by a Turing machine), we can prove that this new halting problem is not solvable by our augmented Turing machine that has access to a halting problem oracle. This creates a new class of languages that form another Turing degree.

You can see that this means there are an infinite number of Turing degrees, since we can always come up with a harder halting problem.

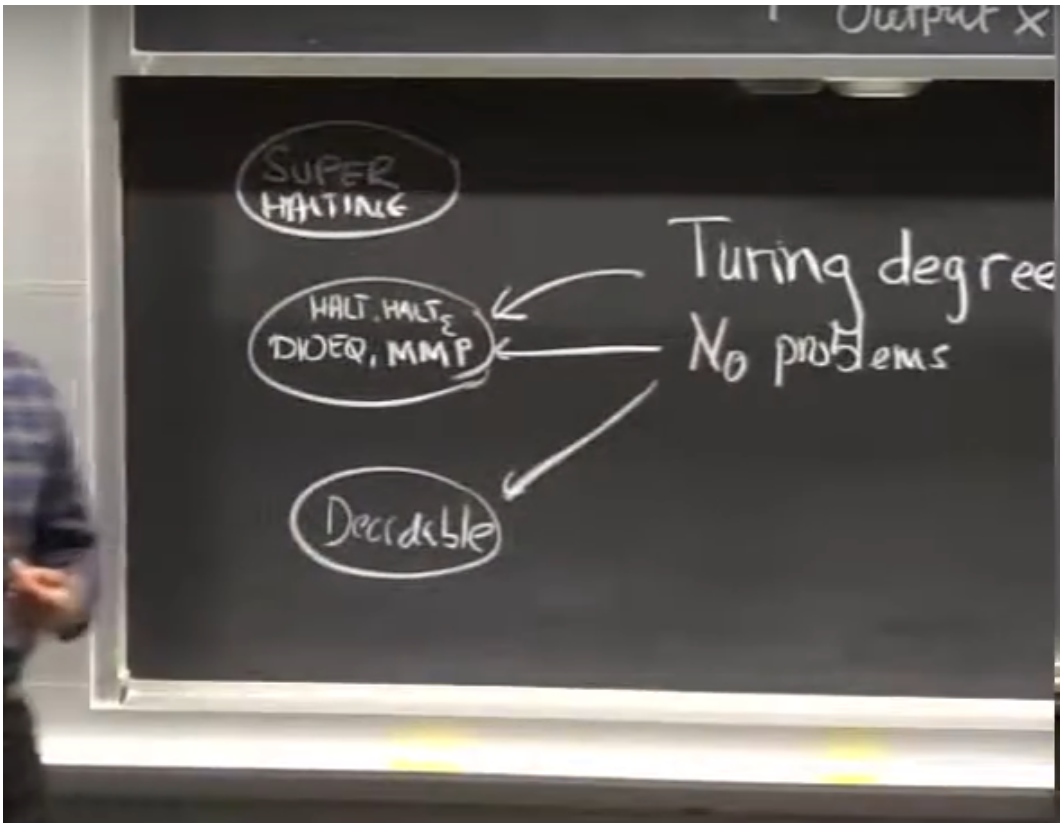


Figure 13: Turing degrees

Lecture 8 - Turing Degrees, Dovetailing, Godel

In the last lecture we defined Turing degrees, which are basically sets of languages. A Turing degree consists of a set of languages that can all be reduced to another language in the same set using a Turing reduction.

The set of computable languages is called degree 0. The set of languages equivalent to the halting problem is just above degree 0. A natural question to now ask is are there degrees between these two?

It turns out the answer is yes. There are infinitely many Turing degrees between degree 0 and the halting degree. What it means to be "between" these two degrees is that if we have a language L in this intermediate degree, the following relation holds -

$$L \leq_T HALT$$

$$L \not\leq_T \text{degree0}$$

$$HALT \not\leq_T L$$

This means that the language L is reducible to the halting problem. That is, you can build a Turing machine to recognize L given an oracle for the halting problem. However, the language is not reducible to any language in degree 0, and the halting problem is not reducible to language L.

MIT 6.046 - Design & Analysis of Algorithms

Taught by Prof. Erik Demaine, Prof. Srini Devadas, and Prof. Nancy Lynch

Course Description

This is an intermediate algorithms course with an emphasis on teaching techniques for the design and analysis of efficient algorithms, emphasizing methods of application. Topics include divide-and-conquer, randomization, dynamic programming, greedy algorithms, incremental improvement, complexity, and cryptography.

Course Website
Lecture Videos (YouTube)

Lectures

1. Interval Scheduling
2. Convex Hull, Median Finding
3. FFT: Divide & Conquer

Lecture 1 - Interval Scheduling

P vs NP

A small change in the problem specification can dramatically change the running time of the optimal solution. A small change in the specification can turn a problem from having a linear time solution to a $O(n^2)$ solution, or even worse.

We can divide all problems we would like to solve into two classes - P and NP.

P - The class of problems that can be solved in polynomial time.
NP - The class of problems whose solution can be verified in polynomial time.

For example, take the Hamiltonian cycle problem. The Hamiltonian cycle problem asks you to find a simple cycle in a directed graph that touches each vertex only once. Verifying a solution to this problem is easily done in polynomial time - you just walk along the path in the solution and check if you touched each vertex only once. However, there is no known algorithm to find such a path in polynomial time. Such a problem is called an NP complete problem.

An NP complete problem is a problem which does not have a solution that runs in polynomial time, but a solution to this problem can be verified in polynomial time. In a sense, we can say that an NP complete problem is one of the hardest problems in NP, and solving any NP complete problem in polynomial time would need a technique that will be able to solve any NP complete problem in polynomial time.

Interval Scheduling Version 1

Let's say you have one resource that you are receiving requests for - it may be a physical resource like a shared printer, it could be your personal time, or computation time on a server. Your goal is to schedule requests by choosing a subset of the requests you receive such that no two requests conflict with each other, and you fulfil the maximum number of requests possible.

Each request has a start time and an end time, and the definition of a conflict between two requests is if one request starts before the other request finishes.

This is an easy problem to solve with a greedy algorithm. Our greedy algorithm will have 3 steps -

1. Choose a request to serve according to some rule
2. Discard all the requests that are incompatible with the request you just chose
3. Repeat till you have no requests left

Our choice of the rule which chooses a request will determine if this algorithm will perform correctly. We go through a few different rules in the lecture, but the correct rule is to just choose the request with the earliest finish time. So the algorithm looks something like this - look at all the requests you have and choose the request that finishes first. Then eliminate all of the requests that conflict with the request you chose. We "move forward" a step and repeat this process till we have no requests left.

Interval Scheduling Version 2

We change the problem definition to include a "weight" for each request. Each request has a weight, and we want to schedule requests for our resource such that the total weight of requests we select is maximum.

This small change in problem specification means that our greedy algorithm no longer works, no matter the rule you use to select a request. We need to change our approach from greedy to dynamic programming.

Let us say we have a list L of requests that we have to schedule. We first sort this list according to the start times of the requests. We define the set R^x to be the set of all requests that come after some time x. We then consider each request in L and construct a list of requests that come after this request ends. We can store this as a dictionary with the keys as the finish times of each request and the values as the index in L which contains the first request that comes after this request finishes. The idea behind our algorithm is that we then consider each request in L as a possible *first* request for our resource. This means there are n sub-problems (suffixes). Writing out the recursion is then fairly obvious -

$ opt(R) = max(w_i + opt({R}^{f(i)})$ `for i in range(n))`

This is the solution for one sub problem, and it takes O(n) time. Since we have n sub problems, our algorithm runs in $O(n^2)$ time.

Lecture 2 - Convex Hull, Median Finding

Divide & Conquer

The general ideology behind divide and conquer is a simple idea - we divide a problem into "a" subproblems of size n/b, where a is an integer greater than or equal to 1 and b is an integer greater than 1. We then usually use the solutions to all other problems and combine them into a single solution for the original problem. We can write a recurrence for the time complexity of a general divide and conquer algorithm as -

$T(n) = a \cdot T(n/b) + [w]$

Where w is the work required to merge the "a" solutions to the subproblems.

Convex Hull

We are given a set of points S = $\{(x_i, y_i)\}$. Our objective is to find a convex polygon with the smallest number of sides that uses the points in S as vertices and contains all of the points in S inside it. We can assume that no two points in the set S have the same x coordinate, no two points in S have the same y coordinate, and no three points in S are collinear.

We call the convex hull CH(S). CH(S) can be a doubly linked list containing a sequence of points in clockwise order.

There is a simple algorithm that we can use to solve this problem that is not too bad. Given the set of points, we choose a pair of points and connect them with a line segment. If all the other points in the set lie on one side of this line segment, then this line segment has to be a part of our convex hull. Otherwise, we just drop that line segment and move to the next pair of points. This is a really simple algorithm. There are nC2 pairs of points, i.e. $O(n^2)$, and it takes $O(n)$ time to check if all points lie on the same side of the line segment. So the worst case running time of this algorithm is $O(n^3)$.

But we can do better with divide and conquer.

Convex Hull w/ Divide & Conquer

We first sort the set of points according to the x coordinate of the points. We then pick the median point, and divide the set of points into two sets - left and right. We then find the convex hull of the two smaller subproblems by recursing on both subproblems, which gives us two convex hulls that we need to merge together to get our solution. The division into two sets is straightforward, but the merge step is interesting, as it is not immediately obvious.

```
## Rough, untested solution
def CH(S: list) -> list:
    if len(S) < 10:
        # If there are less than 10 points left, we just use
        # the simple brute force algorithm
        return brute_force_ch(S)
    left = S[:len(S)//2]
    right = S[len(S)//2:]
    return merge(CH(left, CH(right))
```

The merge routine uses the two finger algorithm to merge the two hulls together. We are given two convex hulls on the opposite side of some dividing line (because we chose the two subproblems to be on different sides of some dividing line).

We start with one finger on the rightmost point of the convex hull on the left and one finger on the leftmost point of the convex hull on the right. We then set up a while loop in which we keep trying to move our fingers up (one after the other), and calculate the y intercept each time. So in the first iteration, we move our right finger clockwise and keep our left finger in the same spot. We then calculate the y intercept of the line joining our two chosen points. If the y intercept went down, then this line is not a part of the merged hull, so we go back to our previous point. We then move our left finger counterclockwise and keep our right finger in the same spot and calculate the y intercept. We keep doing this until we try moving both our left and right fingers, and the y intercept decreases each time. At this point, we can say that we have found the upper tangent for the merged hull. We then repeat this procedure for finding the lower tangent.

The recurrence for this algorithm, as you can clearly see, is T(n) = 2T(n/2) + O(n), which is the same as the merge sort recurrence, and so the time complexity of this algorithm is $O(nlogn)$. This is the optimal time complexity for the general convex hull problem.

There is a final step left of removing the edges of the two sub hulls now that we have merged them. For doing that we follow a "copy and paste" procedure. Let us say that we have found the upper tangent going from point a_i to point b_j, where a_i is a point in the left sub hull and b_j is a point on the right sub hull, and we have found a lower tangent going from point a_k to point b_m, where a_k is a point on the left sub hull and b_m is a point on the right sub hull. We select the edge (a_i, b_j) to be the first edge on our merged hull. We then go clockwise around the right hull and add all the points we see to our merged hull, till we get to b_m. Once we get to b_m, we jump to the left sub hull at point a_k and keep going clockwise around the left hull and adding all the points we see until we get to a_i. This process gives us a closed, merged convex hull, as you can verify. This is also an O(n) subroutine.

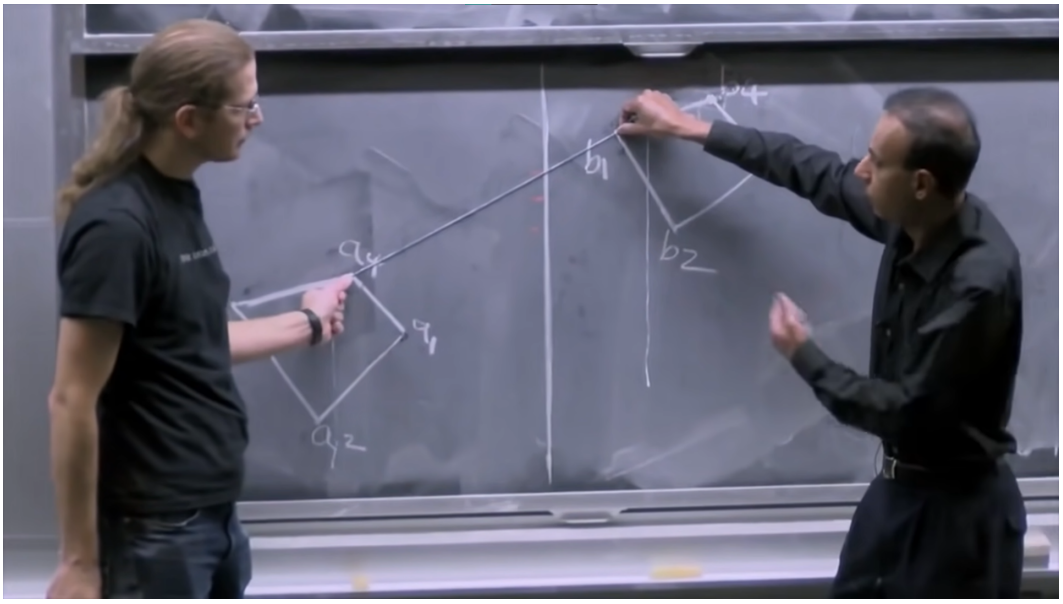


Figure 14: Image

Median Finding

The problem is simple, to find the median faster than having to sort the list of numbers and look at the median position.

So we want to find the median of a given set of numbers in linear time, if possible, and we will use the divide and conquer approach to do so. We first define the rank of an element, which is the number of elements in the given set smaller than or equal to the given element.

Our lower median is the element with rank floor(n+1 / 2) and our upper median is the element with rank ceiling(n+1 / 2). We define a selection routine that will select the element with some rank. We can then find the median by just calling the select routine on the rank of the median.

The select routine first chooses an element in the array, x. We don't choose this number randomly, but use a complex subroutine to choose x cleverly. Once we have chosen x, we divide the set of elements into two sub arrays. We iterate through the array and compare each element with x. We put all the elements less than or equal to x in the left array, and the elements greater than x in the right array.

We can then calculate the rank of x, which is the length of the left sub array. If the rank of x is the rank we are looking for, we return x, otherwise, If we want to find an element of rank less than the rank of x, we recurse on the left sub array, otherwise, we recurse on the right sub array to find the element of rank(x) - the rank we are

searching for.

However, you can see that we are doing O(n) work here, and if we just chose x randomly, we would end up with an $O(n^2)$ algorithm. We define a complex subroutine to choose x that will make sure that we recurse on a problem that is geometrically smaller than the current problem.

We take the set of elements and arrange them into a 2D array with 5 rows, that is, we divide the set of elements into columns of size 5.

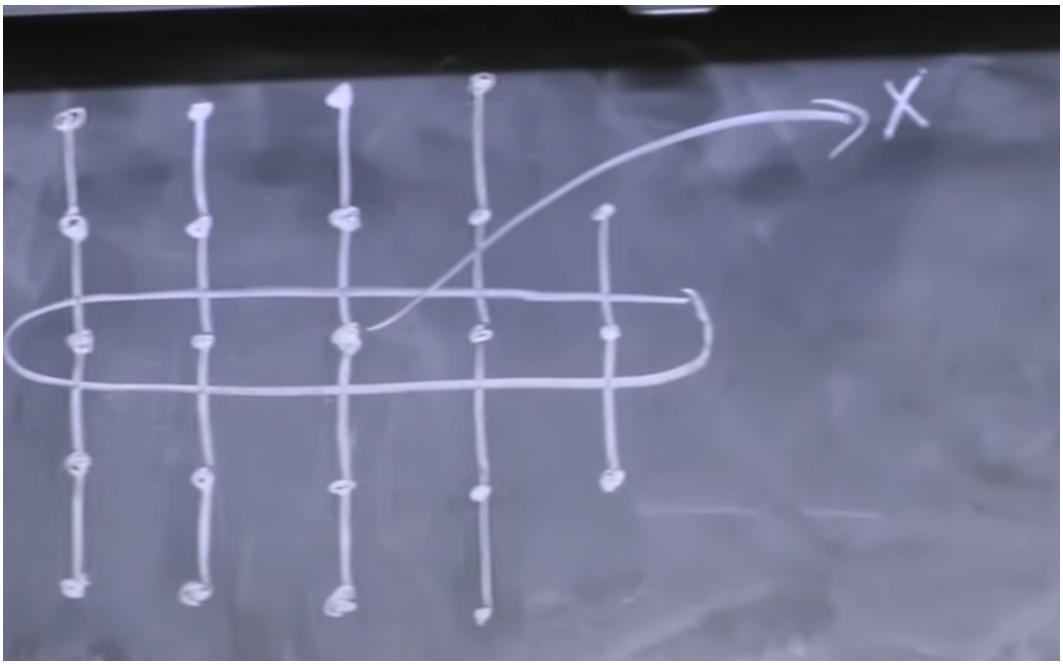


Figure 15: Image

We then sort each column of 5 elements, and since all columns are of a fixed size (5) for all subproblems, this takes constant time. We then look at the medians of all of these columns, as shown in the figure. We then find the median of these columns by recursing on the slice we have selected. That median that we find is the x that we will choose. Moreover, this x has the property that the elements to the right and up of x are greater than x, and elements to the left and below x are less than x.

This means that choosing this as x will give us fairly balanced subproblems. If we do the math to calculate how many elements will be in each problem, we get that the recurrence for the complete algorithm is

$T(n) = T(n/5) + T(7n/10 + 6) + \Theta(n)$

This recurrence solves to be O(n).

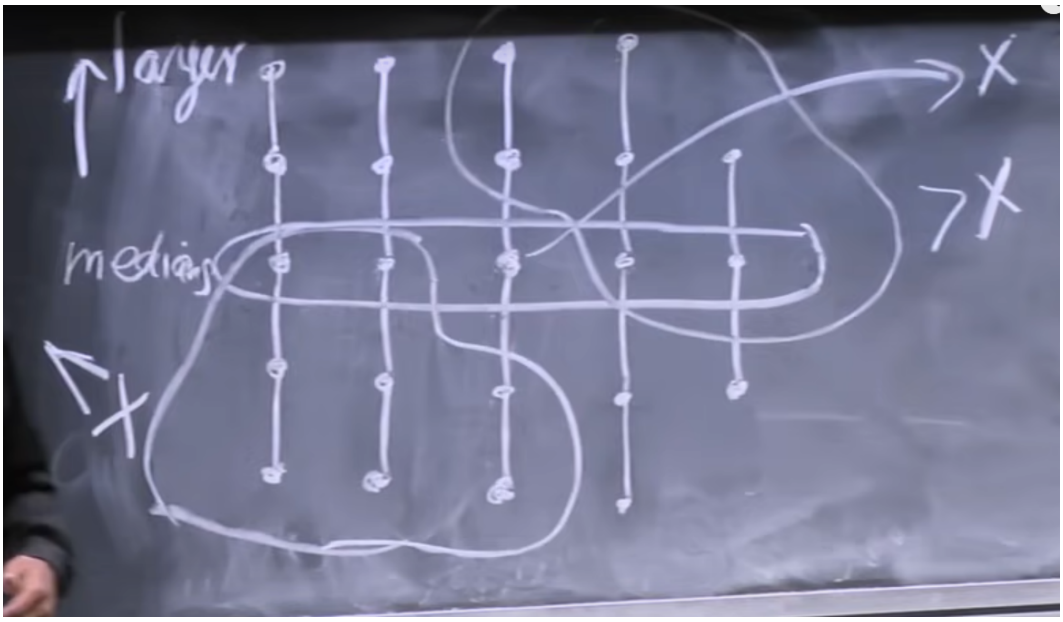


Figure 16: Image

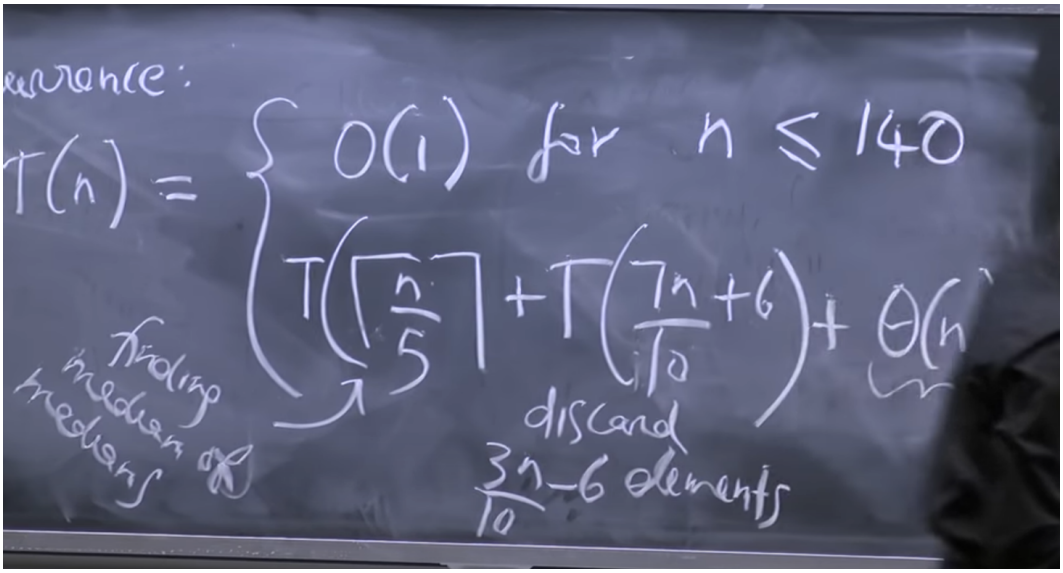


Figure 17: Image

Lecture 3 - FFT: Divide & Conquer

Operations on Polynomials

This lecture focuses on fast fourier transforms and operations on polynomials. We represent polynomials as a one dimensional vector of real numbers, where the ith term represents the coefficient of x^i. Some operations like evaluation and addition of polynomials are easy to do in linear time, but doing multiplication of two polynomials efficiently is what we care about, since it is possible to achieve multiplication faster than $O(n^2) time with a non-trivial algorithm.

We care about multiplication because multiplication of two vectors finds important applications in signal & image processing, and many other areas. We will be able to do one dimensional vector multiplication in $O(n \cdot logn)$ time.

We have three operations we care about - evaluation, addition, and multiplication. We want to be able to do these operations as quickly as possible, and we have three representations of polynomials available. Each representation has some advantages and a disadvantage related to the operations we care about.

1.Coefficient Representation - This is the standard representation of polynomials where we state the coefficient of each power of x. 2.Root Representation - We can uniquely specify a polynomial by specifying all of its roots. 3.Sample Representation - We can uniquely specify a polynomial of degree k by specifying the value the polynomial takes at at least k distinct points.

The advantages and disadvantage of each representation is shown in the figure -

Multiplying two polynomials represented by samples is easy, since they are just multiplications of the corresponding sample points. Doing the same in coefficient representation is harder because it involves $O(n^2)$ operations.

We can't use the workaround of sampling two polynomials and multiplying them because it takes $O(n^2)$ time to sample n points in coefficient form.

We would like to have a representation that would give us the best of both worlds. We don't have such a representation. Instead, we will be able to develop a procedure to convert between the coefficient representation and sample representation in O(n log n) time. This algorithm is called the fast fourier transform - taking n samples from a polynomial or transforming a polynomial from the coefficient representation to the sample representation and vice versa.

Fast Fourier Transform

We first represent the polynomial as a matrix along with n points (x values) we want to sample the polynomial at. The way we represent the n points we want to sample at is a little different than just x values, and it is called the Vanderbonde matrix.

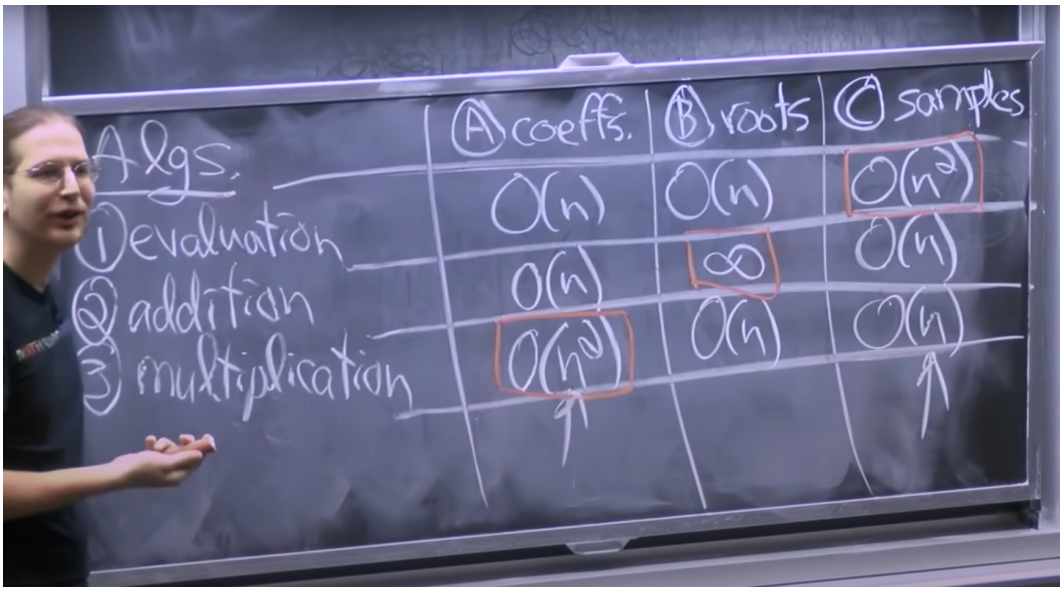


Figure 18: Image

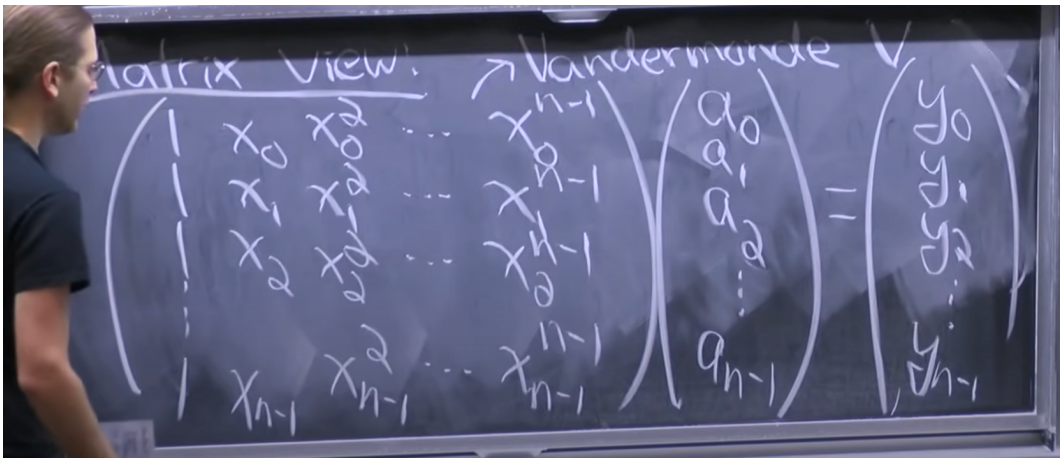


Figure 19: Image

In the photo above, the Vanderbonde matrix is the matrix on the left. Each row of the matrix is one point x_i, and each row contains all the powers of x_i we will need. We can represent the Vaderbonde as $V_{jk} = {a}_{j}^{k} $. The column vector on the right has the coefficients of the polynomial written in a column. To compute the samples we just have to compute the product. This also gives us a way to convert back from samples to coefficients - we just multiply both sides by the inverse of the Vanderbonde matrix.

The problem is that performing the above multiplication will take $O(n^2)$ time. Computing the Vanderbonde matrix and its inverse can be considered free because we only have to do it once, but the product takes quadratic time, which is not of any use to us.

Choosing Sample Points

We go back to the coefficient representation of the polynomial. Let us say we have a polynomial A(x) of degree n that we want to sample for all points in some set X, where X has n points.

We can use a divide and conquer algorithm to solve this problem recursively. First, we divide the polynomial A(x) into two smaller polynomials of half the size - $A_{even}(x)$ and $A_{odd}(x)$. $A_{even}(x)$ is constructed by considering only the coefficients of the polynomial at even positions (a_0, a_2, a_4, ...), so it is a different polynomial of half the degree of A(x). Similarly, $A_{odd}(x)$ is constructed by considering only the coefficients at odd positions (a_1, a_3, a_5, ...), and is another polynomial of degree n/2. We can represent these two polynomials arithmetically as -

$A_{even}(x) = \sum_{k=0}^{n} a_{2k} \cdot x^k$

and

$A_{odd}(x) = \sum_{k=0}^{n} a_{2k+1} \cdot x^k$

(The limits of the sum, especially the sum representing A_{even} may be off by one)

Now, if we somehow had computed these two polynomials, we could combine them to get A(x) using the following relation -

$A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$

So if we wanted to evaluate A(x) for all n points in X, this would take linear time. Now we can set up the recurrence. We recursively divide the polynomial into smaller and smaller polynomials till we get to the base case of degree one. We then have to evaluate these polynomials over a set with n elements, X^2, where X^2 is the set of all element wise squares of X (this is because we have to evaluate the odd and even polynomials at x^2). The recurrence is -

$T(n, |X|) = 2 \cdot T(n/2, |X|) + O(n + |X|)$

This is not solvable by the master method, but if you draw a recursion tree, you can see that this recurrence solves to being $O(n^2)$. This is the same complexity we got before, and this doesn't help us at all. We got this complexity because the size of the set X does not decrease. This is where the choice of X comes in. We could choose X to be a special set, such that the number of elements in the set halves when we square each element in the set.

You can probably see that choosing X to be the set of nth roots of unity will give us this property. For that to work, n has to be a power of 2, but if it isn't we can just take it to be the closest higher power of two. That will only double the number of sample points we will get in the worst case, and having more than n sample points will still uniquely identify the polynomial. The nth roots of one are represented by

$$e^{2i\pi \cdot k/n}$$

where k goes from 0 to n-1. Verify for yourself that squaring the 4th roots of one gives you the 2nd roots of one, and so on for all powers of two.

When we choose this set as the set X, it's size halves as we go down each level of recursion, and the recurrence we set up above simplifies to

$$T(n) = 2 \cdot T(n/2) + O(n)$$

which is the classic merge sort recurrence, and which solves to $O(n \cdot logn)$. This algorithm, with this particular choice of X, is called the fast fourier transform.

The divide and conquer algorithm we outlined above is called the discrete fourier transform. When we set the Vanderbonde matrix to the nth roots of unity, we get the fast fourier transform algorithm.

Fast Polynomial Multiplication

To convert back from the sample representation to coefficient representation, we have to multiply the sample representation with the inverse Vanderbonde matrix. The inverse Vanderbonde matrix for the nth roots of unity is given by simply taking the complex conjugate of each element. We then apply the same algorithm using this inverse, and we can convert from sample representation to coefficient representation.

MIT 6.840 - Theory of Computation

Taught by Prof. Michael Sipser in Fall 2020.

This course emphasizes computability and computational complexity theory. Topics include regular and context-free languages, decidable and undecidable problems, reducibility, recursive function theory, time and space measures on computation, completeness, hierarchy theorems, inherently complex problems, oracles, probabilistic computation, and interactive proof systems.

Course website Lecture Videos (YouTube)

Lectures

1. Introduction, Finite Automata, Regular Expressions

Lecture 1 - Introduction, Finite Automata, Regular Expressions

Finite Automata

A finite automaton is a model of computation. It consists of a set of "states", and transitions between those states based on inputs. Finite automata are used to recognize patterns in strings, and they are the basis for regular expressions.

An automaton can be described formally as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where: 1. Q is a finite set of states. 2. Σ is a finite set of input symbols (the alphabet). 3. $\delta : Q \times \Sigma \to Q$ is the transition function, which takes a state and an input symbol and returns the next state. 4. $q_0 \in Q$ is the start state. 5. $F \subseteq Q$ is the set of accept states.

A string is accepted by the automaton if, starting from the start state q_0 and processing each symbol of the string in order, the automaton ends in an accept state after reading the entire string. Therefore, an automaton can either accept or reject a string. We call the set of all strings accepted by an automaton A its language:

$$L(A) = \{w \in \Sigma^* | A \text{ accepts } w\}$$

We say that A is the language of an automaton M and that M recognizes A, or $A = L(M)$. We say that a language is a regular language if there is a finite automaton that recognizes it. That is, a language is regular if we can build a finite automaton such that the automaton accepts all the strings in the language and only the strings in the language.

MIT 6.851 - Advanced Data Structures

Taught by Prof. Erik Demaine in Fall 2012.

Data structures play a central role in modern computer science. You interact with data structures even more often than with algorithms (think Google, your mail server, and even your network routers). In addition, data structures are essential building blocks in obtaining efficient algorithms. This course covers major results and current directions of research in data structure.

Course website Lecture Videos (YouTube)

Lectures

1. Persistent Data Structures

Lecture 1 - Persistent Data Structures

The first two lectures discuss temporal data structures, which are data structures that allow you to access past versions of the data structure. This is useful for applications like undo functionality in text editors or version control systems. It is kind of like a time machine for data structures.

The first lecture introduces the concept of persistent data structures, which are data structures that preserve their previous versions when modified, and for which you can access any version at any time.

Types of Persistence

There are four types of persistence:

1. **Partial Persistence**: You can access any version of the data structure, but you can only modify the most recent version. This means that the versions are ordered linearly.
2. **Full Persistence**: You can access any version and modify any version. This means that the versions form a tree structure, where each version can have multiple children.
3. **Confluent Persistence**: You can access any version and modify any version, and you can also merge two versions together. This means that the versions form a directed acyclic graph (DAG) structure, where each version can have multiple parents.
4. **Functional Persistence**: This is a special case of full persistence where the data structure is immutable, meaning that once it is created, it cannot be modified. Instead, you create a new version of the data structure with the modifications.

We will look at how to implement partial and full persistence, and discuss features of confluent and functional persistence, but some aspects of confluent and functional persistence are still open problems.

Partial Persistence

We use the pointer machine model to implement all types of persistence. This means that the data structures we consider will have nodes and pointers, such as trees, linked lists, etc. We can implement partial persistence efficiently if we are guaranteed that the number of pointers going *to* a node is bounded by a constant.

To convert any pointer machine data structure to a partially persistent data structure, we can use the following approach. For each node in the original data structure, we create a new node that contains:

- A read-only area of the fields of the node containing their original values (values that the fields were initialized to).
- An area for back-pointers, such that for every pointer that comes into this node, we store a pointer going back to the node that points to this node. This is needed so that when this node eventually gets "full", we can move this data to a new node, in which case the pointers pointing to this node will need to point to the new node instead.
- A constant amount of space for storing modifications to the fields of this node. This is like a log which will be written to every time a field of this node is modified.

We claim that using this approach, we can implement partial persistence in $O(1)$ amortized time per operation.

1. Reading a field is simple: we just read the value of the field, and then go through the mods log and see if the value was changed.
2. To write to a field, we first check if the mods log is full. If it is not, we just write the new value to the mods log by storing a tuple of the form `(version, field, new_value)`. If the mods log is full, we create a new node `n'` that contains the same fields with their latest values, the same back pointers, and a new empty mods log. We then look at all the pointers coming into the old node `n`, and for each pointer to `n` from a different node `m`, we recursively call a write on `m` to change the given pointer to point to `n'` instead of `n`.

Writes can clearly take a lot of work if the mods log is full, but it turns out that the amortized cost of writes is still $O(1)$, because we only create a new node when the mods log is full, and we only do this once per node. The number of nodes created is proportional to the number of writes, so the amortized cost is still $O(1)$.

Full Persistence

To implement full persistence, we can use a similar approach to the one we used for partial persistence, but now the versions are not linear, but rather form a tree structure. Most of the process for converting a pointer machine data structure to a fully persistent data structure is the same as for partial persistence, but we need to make some changes to handle the tree structure.

We "linearize" this tree of versions by simply using its inorder traversal. Furthermore, we use a special data structure called an order-maintenance data structure, which allows us to tell if a version is an ancestor of another in $O(1)$ time, and also lets us insert versions before or after any other version in $O(1)$ time. This is useful because we need to be able to insert new versions into the tree structure as we modify the data structure.

Now for each node we store `2(d + p + 1)` mods, where `d` is the number of fields in

the node, and `p` is the number of back pointers from this node to other nodes. The `+1` is only for the amortized analysis to work.

1. Reading a field is the same as in partial persistence: we read the value of the field and then go through the mods log to see if the value was changed. However, we need to check the order-maintenance data structure to see if the version we are reading is an ancestor of the current version. We only consider the mods that are in the current version or in an ancestor of the current version.
2. To write to a field, we first check if the mods log is full. If it is not, we just write the new value to the mods log by storing a tuple of the form `(version, field, new_value)`. If the mods log is full, we create a new node `n'` and essentially split `n` into two nodes: `n` and `n'`. We store half of the modes in `n` and the other half in `n'`. We then determine all nodes which need to point to `n'` instead of `n`, and recursively call a write on those nodes to change the pointers to point to `n'` instead of `n`.

MIT 6.858 - Computer Systems Security

Taught by Prof. Nikolai Zeldovich at MIT in Spring, 2020.

6.858 Computer Systems Security is a class about the design and implementation of secure computer systems. Lectures cover threat models, attacks that compromise security, and techniques for achieving security, based on recent research papers. Topics include operating system (OS) security, capabilities, information flow control, language security, network protocols, hardware security, and security in web applications.

6.858 on OCW
2020 course website
Lecture Videos (YouTube)

Lectures

1. Lecture 1: Introduction
2. Lecture 2: Security Architecture
3. Lecture 3: User Authentication
4. Lecture 4: Buffer Overflows
5. Lecture 5: Privilege Separation

Lecture 1 - Introduction

This course is about building secure computer systems. Security is a property or behaviour of a system that is achieved despite attacks from adversaries. This property could be data confidentiality, data integrity, access control, etc.

There are three main aspects to building secure systems -

1. Policy - The plans and/or rules you will implement to make sure that your system is secure. For example, require users to enable 2FA.
2. Threat model - Assumptions about what the attacker can and cannot do. For example, the attacker can try to guess passwords, but cannot try to physically break into the server room (this assumption is fine in most cases).
3. Mechanism - The specific software and hardware techniques you will use to achieve security.

Building secure systems is hard, because security is a *negative goal*. For example, if our goal is to make sure that only the author of a file can access that file, implementing access control is a positive goal - only one code path is needed to check if the accessing user is the owner, and if yes, access is granted. However, to make sure that *no one else* accesses the file, we may need many checks, techniques, etc. to make sure that no one else is granted access to that file. We must make sure that there is no way in our system for a user that is not the owner to pretend to be the owner.

This is a very hard task, and there is no guarantee that we will get it right the very first time. There is also no guarantee that we have thought of and considered all the possible attack vectors for a system. Therefore, we must constantly be vigilant and iterate. We must be vigilant not just of our system, but open source databases of documented security vulnerabilities, such as cve.mitre.org.

Perfect security is not possible. However, perfect security is not required, either. We just have to make sure that the cost to an attacker is higher than the potential reward they may gain from breaking into our system. Also, implementing some security techniques can ensure that some types of attacks are completely nullified.

The rest of the lecture presents examples on security failures to highlight the ways in which a security engineer must think. They are very interesting and fun reads, you can find them here.

Lecture 2 - Security Architecture

Security architecture is structuring entire systems in order to defend against large classes of attacks, prevent as-yet unknown attacks, and limit damage from successful attacks. This lecture analyzes a paper written by Google describing their security architecture for Google Cloud Platform. It gives a good overview of the techniques and principles Google uses to implement their security architecture.

You can read the paper here. You should read the paper before going through these notes. The paper describes the architecture Google uses, these notes discuss *why* Google uses that architecture and why it is good.

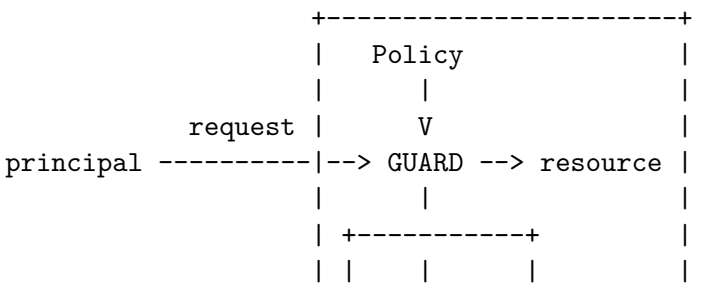
Google has the following broad security goals - 1. Protect customer data 2. Ensure availability of Google services 3. Track down what went wrong if something does go wrong 4. Limit the damage an attacker can do in case of a successful attack

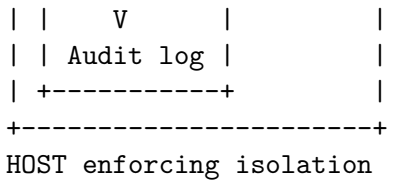
Google has the following hardware/software services in its architecture - 1. Data centers 2. Physical machines in those data centers (servers) 3. Virtual machines in those servers 4. Services running on those virtual machines 5. Applications running on those services 6. Remote Procedure Calls (RPCs) between applications and services 7. Requests coming in through the internet producing RPCs

Isolation

A fundamental principle used by Google's security architecture is isolation. Each service and/or application is isolated from each other (sometimes virtually, sometimes physically), such that by default no service can access the data and/or resources belonging to any other service. This means that even if one service is compromised or has bugs, it is not able to damage any other service. Without isolation, we cannot hope to build secure systems.

Now, we don't want every service to be 100% isolated all the time. We want services to be able to talk to each other, without which we cannot build a distributed system. We need to allow inter-service communication in a controlled way. In order to control communication between services, a common pattern is the guard pattern. A guard is placed in the front of every service, who's job is to monitor incoming requests from other services and only let them in if the service is allowed to talk to it.





In the diagram above, a principal can be a user, a device, a service, etc. Resources can be services, like Gmail or Drive, or files and user data.

The guard performs three functions -

1. Authenticate - Identifies who is issuing the request.
2. Authorize - Determines whether that request should be allowed.
3. Audit - Records each request along with its authentication and authorization information, and other useful metadata.

Lecture 3 - User Authentication

User authentication is an important problem which has a variety of issues that are hard to deal with. Even though the premise of the problem is not complicated, user authentication continues to be challenging because it is not just a technical problem. All schemes we have come up with for user authentication so far are imperfect because the end users are ultimately human and imperfect.

The bigger problem of user authentication can be broken down into three separate sub problems - 1. User registration - The process of registering the identity of a user with a service so that the identity can be verified in the future. This sets up a shared secret between the user and the service (usually a password). 2. Identity verification - The process of authenticating a user and verifying their identity. This process verifies the shared secret. 3. Recovery - The process of recovering the secret if the user loses it. If overlooked, this step can undermine the security of the first two steps, no matter how strong they are.

Identity Verification

The most common way of verifying an identity is a password. The benefit of passwords as a shared secret is that they are easy to remember and very convenient to use. However, the challenge with passwords is that the users of passwords are human, and they tend to use passwords in a way that makes them weaker, such as by reusing passwords, or by using weak or easy to guess passwords.

Passwords are never stored in plaintext. They are combined with a large random number called a salt, and hashed using a cryptographic hash function. The service then stores tuples containing `(username, salt, Hash(salt || password))`. The hash function is also purposefully made to be slow, to make it harder for attackers to brute force passwords.

However, even if passwords can be imperfect, their security can be greatly improved by augmenting them with two-factor authentication. You are already familiar with common ways of introducing two-factor authentication such as OTPs through SMS or an authentication app. These are all good approaches, but they don't protect against MITM attacks and phishing attacks.

Universal Two Factor Protocol (U2F)

The U2F protocol being developed by Google is a form of two-factor authentication which is resistant to phishing and even MITM attacks. It uses public-key encryption and a dedicated 2FA device to verify a user's identity.

It is a great improvement over OTPs over SMS or authentication apps, but it still makes some assumptions which make it vulnerable in some extremely rare situations.

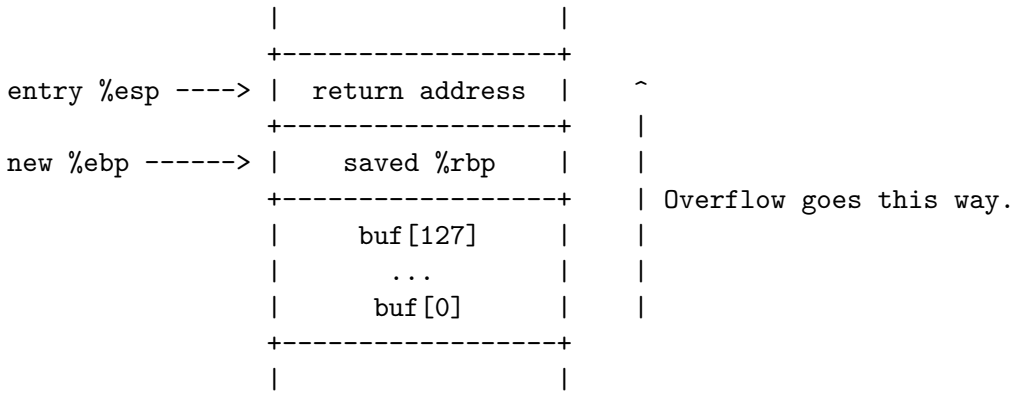
During authentication, the browser sends a login request to the server. The server sends back a "challenge", which is a long random number, for the U2F device to sign using its private key. The user plugs in the U2F device via USB, which receives the challenge, signs it using its private key, and sends it back to the browser. The browser forwards the signed challenge to the server. The server can now verify if the U2F device's identity using its public key.

This is the basic premise of the protocol. To protect against MITM and phishing attacks, the browser actually forwards not just the server's challenge, but also the origin and the TLS connection ID. The U2F device then verifies the signature for all three instead of just the challenge.

Lecture 4 - Buffer Overflows

Buffer overflows are an old but still relevant attack path. A buffer overflow is an attack which uses poorly sanitised/validated user input to overwrite data on the buffer. When a function is called, the registers and the return address are stored on the stack. Sometimes, a buffer is allocated to the function if the function needs to store some data. The buffer is allocated just under the pushed registers.

If the program accepts and stores user input in the buffer and does not check the length of the input, the user input can possibly overwrite the data in the stored registers if the programming language does not check for buffer bounds before writing. For example, whenever a function is called, the stack may contain the return address, the base pointer of the stack, and finally the buffer itself. If a buffer overflow attack exists, the attacker may write data that is so long that it fills the buffer and also overwrites the base pointer of the stack and the return address.



This way, the attacker can supply any return address, which means the attacker can run any function they want. In the classic buffer overflow attack, the attacker supplies the code they want to run in the buffer itself, and the specify the return address as the first address of the buffer.

Now, this attack could be completely mitigated if the programming language simply checked if the address being written to is part of the data structure being accessed. Most modern programming languages do that. However, C doesn't. If you are not using C, then you are probably safe from buffer overflows in your code. For example, if you are using a language like Java or Rust which performs bounds checking, then you most likely won't get buffer overflows. However, you can't always control the programming language you use, and you will, at some point, have to use libraries that are written in C. Even if you don't, you will probably want to run your code on Linux, which is written in C. Therefore, it is important to learn about buffer overflows even if you don't work with C directly.

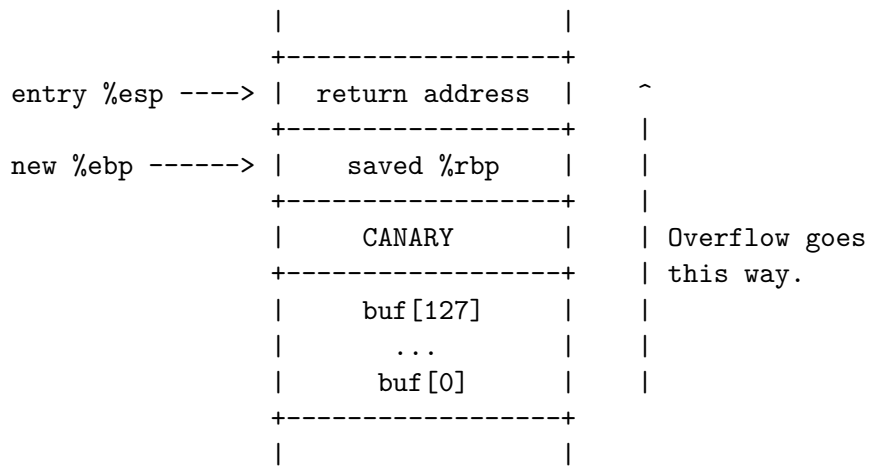
Defenses Against Buffer Overflows

The lecture takes an iterative approach to defending against buffer overflows. In computer security, we rarely have solutions that perfectly solve a problem. It is more common that we have solutions that make it harder to pull of a particular attack successfully. As we develop such solutions, attackers modify their attacks to get around these solutions, forcing us to come up with better solutions, and so on.

NX bit One approach we can take is to tell the hardware not to execute any instructions in pages where the stack is stored. This is done by the operating system, since it is responsible for paging and memory management. This is a simple solution, and it definitely works, but there is a way to bypass it.

Instead of the attacker providing the code they want to run in the buffer itself, the attacker can overwrite the return address to be a `libc` function, like `exec`, and achieve the same effect of running arbitrary code. This is called a "return-to-libc" type of buffer overflow.

Stack Canaries The next approach to protect against "return-to-libc" attacks is called a stack canary. A stack canary is a number that is stored between the stack and the return addresses and saved registers. This is a randomly generated number and is different for every process. It is generated by the operating system and stored off the stack.



When the attacker overwrites addresses outside the buffer, the canary is also overwritten. Before returning, the compiler checks the canary to make sure it is correct. If the canary is not correct, then the compiler knows that a buffer overflow has occurred and crashes the program.

This is a pretty good solution, but it isn't completely foolproof since the canary could be guessed, either by reverse engineering how the canary is generated or just

by brute force.

Address Space Layout Randomization This is a common strategy and pretty effective in combating return-to-libc attacks. Instead of storing libc binaries in the same address space everytime, we randomize the addresses where we store system libraries on every startup, as well as the location of the application stack, code, and heap. This means that the attacker cannot guess where libc functions are stored (easily), and makes it much harder to pull off return-to-libc attacks.

Lecture 5 - Privilege Separation

Privilege separation is the practice of dividing your software into multiple independent services which each run in isolation. This way one service being compromised minimizes the chance of the attacker being able to compromise another service. It also reduces the attack surface.

It is easy to describe the idea of privilege separation, but hard to actually implement it in practice. It is similar to how it is always recommended to make your software modular, but writing modular code is easier said than done.

This week's paper describes the working of a web server - OKWS - which is written with the principle of privilege separation at its core.

Recitation 1 - Linux Containers

CMU 15-445/645 - Database Systems

Taught by Prof. Andy Pavlo in Fall, 2019 and Fall, 2023.
The first 7 lectures are from the course taught in Fall 2019, the rest are from the course taught in Fall 2023.

Course Description

This course is on the design and implementation of database management systems. Topics include data models (relational, document, key/value), storage models (n-ary, decomposition), query languages (SQL, stored procedures), storage architectures (heaps, log-structured), indexing (order preserving trees, hash tables), transaction processing (ACID, concurrency control), recovery (logging, checkpoints), query processing (joins, sorting, aggregation, optimization), and parallel architectures (multi-core, distributed). Case studies on open-source and commercial database systems are used to illustrate these techniques and trade-offs. The course is appropriate for students with lit systems programming skills.

Course Website (2019) Course Website (2023) Lecture Videos (YouTube)

Lectures

1. Introduction & Relational Model
2. Advanced SQL
3. Database Storage 1
4. Database Storage 2
5. Buffer Pools
6. Hash Tables
7. Tree Indexes

Introduction & Relational Model

DBMSs

A database is an organized collection of related data that models some aspect of the real world (e.g., modeling the students in a class, or a digital music store). A DBMS is a software that allows applications to store and analyze the information in a database. It allows the definition, creation, querying, updating, and administration of databases.

DBMSs have 2 layers - the logical layer and the physical layer. The logical layer defines which entities and attributes the data stored in the database will have, and the physical layer is how those entities are actually stored (like the particular data structure used to store the data).

Before relational databases, the physical layer was defined in the application code itself, which made the database much harder to manage. If the structure of the data or the requirements of the application changed, the whole physical layer needed to be rewritten.

Relational Model

Ted Codd proposed the relational model in 1970. The relational model has 3 key points -

1. Store database in simple data structures.
2. Access data through a high-level query language.
3. The physical storage of data is left up to the implementation.

The relational model is an example of a *data model*. A data model is a collection of concepts for describing the data stored in a database. A *schema* is a description of a particular collection of data using a particular data model.

The relational model defines three concepts - 1. Structure - Data is distributed into tables and tables can be related to each other. The structure also describes what kind or type of data each table can hold. 2. Integrity - The database's contents always satisfy some constraints, which can either be defined by the designer or the database itself. 3. Manipulation - Describes how the contents of the database can be modified.

Relational Algebra

Relational algebra gives a set of operations we can perform on relations to produce new relations. You can think of a relation as a table in a database. Relational algebra operators take a relation as an input and produce another relation.

The basic few relational algebra operators are -

1. Select ($\sigma_{\text{predicate}}$(R)) - Acts like a filter. Selects tuples from the relation that satisfy the predicate.
2. Projection ($\pi_{\text{A1,A2,...}}$(R)) - Returns the same relation but only with the attributes A1, A2, ... selected from the relation.
3. Union (R $\cup$ S) - Produces a new relation that contains all the tuples in at least one of the relations (the relations need to have the same attributes).
4. Intersection (R $\cap$ S) - Produces a new relation that contains only those tuples that exist in both R and S (the relations need to have the same attributes).
5. Difference (R - S) - Produces a new relation that contains all tuples that appear in the first relation but not in the second relation (the relations need to have the same attributes).
6. Product (R $\times$ S) - Product takes in two relations and outputs a relation that contains all possible combinations for tuples from the input relations.
7. Join (R $\bowtie$ S) - outputs a relation that contains all the tuples that are a combination of two tuples where for each attribute that the two relations share, the values for that attribute of both tuples is the same.

Advanced SQL

Aggregates

SQL has a few aggregate functions that take a bag of tuples as the input and produce a single scalar value as the output. These can only be used on the left-hand side of a query, in the output list of the `SELECT` clause.

Here is the example database we will be using for this part of the lecture -

```
CREATE TABLE student (
    sid INT PRIMARY KEY,
    name VARCHAR(16),
    login VARCHAR(32) unique,
    age SMALLINT,
    gpa FLOAT
);

CREATE TABLE course (
    cid VARCHAR(32) PRIMARY KEY,
    name VARCHAR(32) NOT NULL
);

CREATE TABLE enrolled (
    sid INT REFERENCES student (sid),
    cid VARCHAR(32) REFERENCES course (cid),
    grade CHAR(1)
);
```

For example, we have this query which gets the number of students in the students table with a '@cs' login -

```
SELECT COUNT(*) FROM student WHERE login LIKE '%@cs';
/* or */
SELECT COUNT(login) FROM student WHERE login LIKE '%@cs';
/* or */
SELECT COUNT(1) FROM student WHERE login LIKE '%@cs';
```

We can use multiple aggregate functions in a single query -

```
SELECT AVG(gpa), COUNT(sid) FROM student WHERE login LIKE '%@cs';
```

If you use an aggregate function in your query and you also select some other columns, you will need to `GROUP BY` those columns. For example, if you `AVG(student.GPA)` and `student.id` in the same query, you will need to `GROUP BY student.id`. This is because you want to get one row for each student, containing that student's average

GPA, which means you want to group each student into a row of their own, look at all of their GPA values, and take the average.

Another example - ~~sql SELECT AVG(s.gpa), e.cid FROM enrolled AS e, student AS s WHERE e.sid = s.sid GROUP BY e.cid;~~

Database Storage 1

Memory Hierarchy

We will assume the database management system will have to deal with 2 levels of memory - volatile and non-volatile. This lecture and the next few lectures are focussed on how the data is actually stored in the file system.

File Storage

We want to give the user the impression that the entire database is stored in the main memory, and not stall when the user requests data from outside the main memory, even though calls to disk are slow.

We want to implement something like a virtual memory.

Now, the operating system can already do this well, but we almost never want to leave it up to the OS to handle this for us, because we can manage memory much better since we have extra information about the queries that are being run and that can be run, and we can use that information to perform spatial and temporal optimizations. Moreover, some queries may require specialized memory management for correctness.

A DBMS stores a database in memory as a file (sqlite) or multiple files. The OS does not know anything about what is inside those files, only the DBMS can interpret them correctly.

The DBMS has a *storage manager* that is responsible for managing these files. It stores the database in these files in blocks of data called pages. A page is just a block of data that can contain any kind of information. There are three kinds of pages -

1. Hardware pages - The blocks exposed by the SSD/disk itself on the hardware level. Usually 4kB.
2. OS page - The blocks as defined by the OS (4kB).
3. Database page - The blocks as defined by the DBMS (0.5 to 16kB).

Heap File Organization

There are a few ways to organize pages in memory, and the one covered in the lecture is the heap file organization. A heap file just means that pages are stored in memory in an unordered way, and tuples are stored inside those pages.

We maintain something called a page directory, which is basically a hash table that maps page ids to the location of the pages in memory. It can also contain metadata about the page.

Page Structure

The data contained in the page itself can be stored in two ways - slotted pages and log-structured pages.

Slotted pages are pages which contain "slots" for tuples to go into. The page has a header that stores metadata about the page, and the top of the page contains a "slots" array that maps slot locations to an offset in the page. The tuples are stored starting from the bottom of the page working up. The page is said to be full when the slots array and the tuples touch.

Log structured pages store only logs, i.e, records of how the page was modified. The data present in the page can then be inferred from the logs during a read. This results in fast writes, but could lead to slow reads. However, there are a few techniques to make reads practical as well.

Database Storage 2

Data Representation

This section describes how the DBMS stores the values inside the tuples in memory. Tuples' data is essentially just byte arrays. It is up to the DBMS to know how to interpret those bytes to derive the values for attributes. A data representation scheme is how a DBMS stores the bytes for a value. There are four main types that can be stored in tuples: integers, variable precision numbers, fixed point precision numbers, variable length values, and dates/times.

Integers, Floats, Reals These are usually just stored in the same way C/C++ store them, as specified by the IEEE-754 standard.

Fixed Point Precision Numbers The DBMS defines how it will handle fixed point precision numbers, and implements an API to access them itself. It does not rely on CPU instructions, since they have rounding errors. Instead, the entire precision of the number is stored in the representation, along with additional metadata telling the DBMS how to interpret the value.

These types are used when rounding errors are unacceptable, for example in scientific or financial applications. However, handling these types is much slower than floating point numbers.

Variable Length Data

These are data types that consist of a varying amount of data that needs to be stored. For example, `TEXT`, `VARCHAR`, `BLOB`, `VARBINARY`. Has a header that keeps track of the length of the string to make it easy to jump to the next value. Most DBMSs do not allow a tuple to exceed the size of a single page, so they solve this issue by writing the value on an overflow page and have the tuple contain a reference to that page.

Some systems will let you store these large values in an external file, and then the tuple will contain a pointer to that file. For example, if our database is storing photo information, we can store the photos in the external files rather than having them take up large amounts of space in the DBMS. One downside of this is that the DBMS cannot manipulate the contents of this file.

Datetime Data These are usually just represented as an integer equal to the number of milli or micro seconds passed since the UNIX epoch.

Workload Types

Depending on the type of work you want to do with your data, systems can be divided into two types - Online Transaction Processing (OLTP), and Online Analytical

Processing (OLAP).

OLTP

OLTP basically means you want transactions on your database to be fast. It is usually used when your application is read heavy, and you want to collect potentially large amounts of data from your frontend efficiently.

Characteristics of OLTP systems are - 1. Fast, short running operations. 2. Queries usually only operate on a single entity at a time (hence joins are rare). 3. Application is read-heavy. 4. Usually collect more data than they need to analyze.

OLAP

OLAP means that you want joins and other complex operations involving multiple parts of your database to be fast, usually because you want to perform analytics.

Characteristics of OLAP systems are - 1. Long running, complex queries accessing multiple columns, tuples, and tables. 2. Long scans through entire tables. 3. Analytical and exploratory queries. 4. Usually analyze and read more data than they write.

Storage Models/Methods

The relational model does not specify that you should store your tuples contiguously. There are two types of storage models - Decomposition storage model (column storage) and N-Ary storage model (row storage).

NSM

NSM stores all the columns of a single tuple contiguously in the same page. This approach is ideal for OLTP workloads where transactions tend to operate only an individual entity and insert heavy workloads. It is ideal because it takes only one fetch to be able to get all of the attributes for a single tuple.

Its advantage is fast inserts and updates, and it is good for queries that need the entire tuple. Its disadvantage is queries that use only a small portion/number of columns of the tuple.

Therefore, row stores are usually not used for OLAP systems.

DSM

DSM stores a single attribute or column for all tuples contiguously in a single page.

Locks vs. Latches

Locks and latches mean different things in a DBMS than in an OS.

Locks are - 1. Used to protect the logical contents of the database from other transactions. 2. Held for transaction duration 3. Help to roll back changes

Latches are - 1. Used to protect internal data structures from other threads 2. Held only for operation duration 3. Do not need to be able to roll back changes

Buffer Pool

A buffer pool is just an area in memory that the DBMS reserves to read pages from disk. The buffer pool consists of discrete "slots" of memory we call frames that are the same size as a page. The DBMS reads pages from disk and brings them into frames.

A buffer pool maintains the following metadata - 1. Page table - A hash table mapping page ids to memory location in the buffer pool 2. Dirty flag - A flag indicating whether a page has been modified. 3. Pin counter - The number of threads currently accessing that page. This counter prevents pages that are being accessed from being evicted.

Now that we have introduced buffer pools, we can make the following optimizations - - Multiple buffer pools - We can have multiple buffer pools for different purposes. Each buffer pool can have its own replacement policy and some other properties that may help. For example, having a dedicated buffer pool for each table can help with some particular queries. - Pre-fetching - The DBMS can pre-fetch pages into the buffer pool if it decides that it needs to based on the query plan. The OS can also do this for simple queries, but not in all situations (e.g, with indexes on a query that operates on a certain range of pages). - Scan sharing - If a query is performing a scan over a table, and another query comes in from another thread that also needs to perform the same scan, it can join the scan that the first query is performing from the page that the scan has reached now.

Replacement Policies

Once we fill our buffer pool, which can happen pretty quickly for large queries, we will need to evict an existing page from the pool to make space for a new page. The way we decide which page to evict is called the replacement policy. This is an extremely important component of the DBMS, because it can make a huge difference in performance. Commercial database systems have a very complex replacement policy as compared to open source systems.

There are a few common replacement policies, but the most popularly used ones are LRU and clock. You already know what LRU is so I won't write about it.

Clock is an approximation of LRU that doesn't need you to store the last accessed time stamps. You maintain a circular queue with a pointer pointing to the "current" page. Each page has a single bit of metadata that indicates whether that page has been accessed since the last time the pointer was here. The pointer always moves forward (or "clockwise"), and when it wants to evict a page, it checks the reference bit for that page. If that bit is 1, that means that the page was accessed fairly recently, and we don't evict that page, but we set its bit to 0. We evict the first page we find that has its reference bit set to 0.

B-Trees and Indexes

A table index is a replica of a subset of a table's columns that is stored in such a way that it allows the DBMS to find tuples faster than sequential scans.

Indexes dramatically speed up some types of queries, but they also take up space, and need to be kept in sync with the data. Too many indexes can make reads fast but writes slower.

The query optimizer decided which index to use when it receives a query.

B+ Tree

A B+ Tree is a self-balancing tree data structure that keeps data sorted and gives $O(logn)$ search, sequential access, insertion, and deletion.

A B+ Tree is used in almost all DBMSs which support order-preserving indexes.

A B+ Tree comes from a class of data structures called B Trees. It has the following properties - 1. It is perfectly balanced (every leaf is at the same depth). 2. Every inner node other than the root is always at least half full. 3. Every inner node with k keys has k+1 children.

That is, if we have a B+ Tree of degree M, that means its nodes can contain at most $M-1$ keys and M children. Property 2 says that every inner node has between $M/2-1$ to $M-1$ keys.

Andy then explains the insertion and deletion algorithms in a B+ Tree, which I won't describe here. But they are pretty much what you would expect.

CS162 - Operating Systems and Systems Programming

Taught by Prof. John Kubiatowicz at UC Berkeley in Fall, 2020

Course Website
Fall 2020 Course Website
Lecture Videos (YouTube)

Lectures

1. Lecture 1 - What is an Operating System?
2. Lecture 2 - Four Fundamental OS Concepts
3. Lecture 3 - Abstractions 1: Threads and Processes
4. Lecture 4 - Abstractions 2: Files and I/O
5. Lecture 5 - Abstractions 3: IPC, Pipes, and Sockets
6. Lecture 6 - Synchronization 1: Concurrency, Mutual Exclusion
7. Lecture 7 - Synchronization 2: Semaphores, Lock Implementation
8. Lecture 8 - Synchronization 3: Atomic Instructions, Monitors, Readers/Writers
9. Lecture 10 - Scheduling 1: Concepts & Classic Policies
10. Lecture 11 - Scheduling 2: Case Studies, Real Time, Forward Progress
11. Lecture 12 - Scheduling 3: Deadlock
12. Lecture 13 - Memory 1: Address Translation and Virtual Memory
13. Lecture 14 - Memory 2: Virtual Memory, Caching and TLBs
14. Lecture 15 - Memory 3: Caching & TLBs, Demand Paging
15. Lecture 16 - Memory 4: Demand Paging Policies
16. Lecture 17 - Demand Paging, General I/O, Storage Devices
17. Lecture 18 - General I/O, Storage Devices, Performance
18. Lecture 19 - Filesystems 1: Performance, Queueing Theory, Filesystem Design
19. Lecture 20 - Filesystems 2: Filesystem Design, Case Studies
20. Lecture 21 - Filesystems 3: Case Studies, Buffering, Reliability, Transactions
21. Lecture 22 - Transactions, End-to-End Arguments, Distributed Decision Making
22. Lecture 23 - Distributed Decision Making, Networking, TCP/IP

Lecture 1: What is an Operating System?

An operating system is a piece of software that provides other software access to the system's hardware resources, and makes it seem much more convenient and less complex. It also gives applications isolated environments with useful abstractions of hardware resources.

It is like an illusionist. It provides clean, easy-to-use abstractions of physical resources and makes it seem to the user as if their machine has infinite memory, and makes it seem to the application that it has the entire machine to itself.

The operating system doesn't directly expose the processor and its registers to an application. Instead, it exposes "threads", which an application runs on. An application can run on one or more than one thread(s). It does not directly expose the memory to an application. Instead, it exposes "address spaces", "files", and "directories". It does not directly expose the network adapters and cards to the application. Instead, it exposes "sockets".

On top of threads, the OS packages each running application into its own "process". This makes it seem to the application that it has the entire machine to itself, because it isolates the application from every other application running on the machine. You can think of a process as and isolated execution environment with restricted rights provided by the operating system.

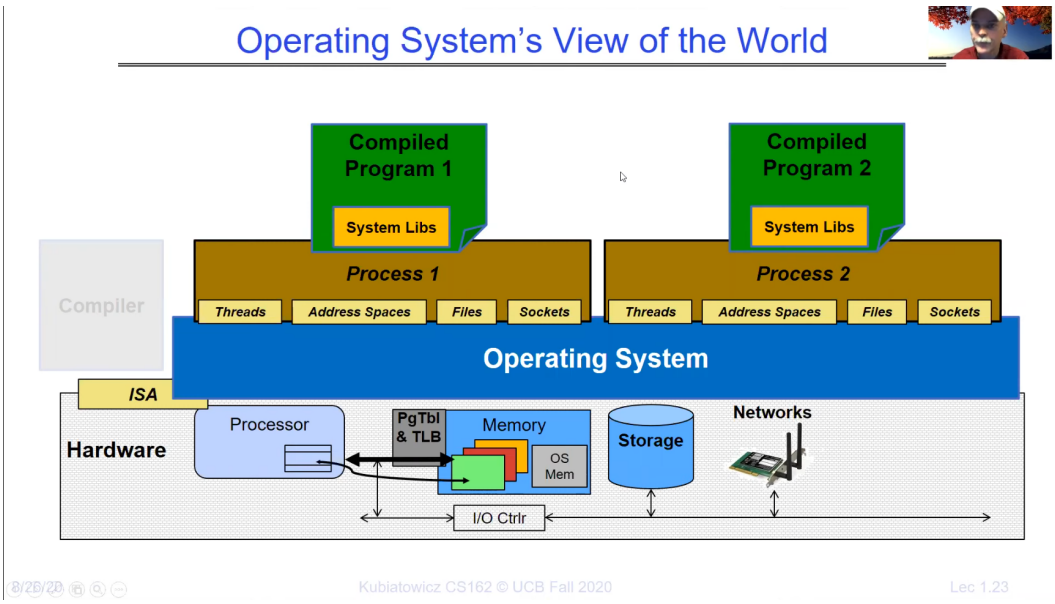


Figure 20: Image

Lecture 2: Four Fundamental OS Concepts

This lecture introduces four fundamental concepts of an operating system - 1. Threads: Execution Context 2. Address Spaces 3. Processes 4. Dual mode operation

Threads

A thread is a single unique execution context that has its own program counter, registers, execution flags, stack, and so on. This means that the block diagram you have studied for how a processor executes a program is really one single thread executing its own code.

A thread is said to be executing when it is "resident" in a core. Resident means that the PC of the core is pointing at the next executable instruction for the thread and the instructions of the thread are stored in memory. A resident thread's stack pointer holds the address of the top of the stack for that thread.

A thread is said to be suspended when its data (everything we have seen above) is not in memory.

The OS gives the user the illusion of multiple processors by multiplexing the execution of threads in time. All threads that want to run on the processor are stored in a chunk of memory called the Thread Control Block, and the data for each thread is swapped in and out of the hardware registers over time.

Swapping out the data of one thread for another takes some time, which is called context switching. We have to be careful not to swap between executing threads so often that majority of the time is spent in context switching, since the more context switching we perform, the more time is "wasted". However, if we don't context switch fast enough, we might get the feeling that our machine is unresponsive, since only one process would be running on the CPU for a long time, making all other processes hang.

Context switching can also happen because the thread voluntarily gave up control, or it asked for some i/o, and so on.

Address Spaces

An address space is a set of addresses that a thread or process can access. A 32-bit processor can access 2^{32} spaces, which is around 4 billion. A 64-bit processor can access 2^{64} address spaces, which is 10^{18}

When a thread or a process writes to an address space, one of many things can happen, depending on the type of memory location. A memory location can behave as -

1. A regular memory location, i.e. data written is stored in memory

2. Memory-mapped I/O, i.e. data written is mapped to I/O ports
3. Communication between two or more programs

An address space for a thread or process contains the parts shown above. The code segment contains the instructions that are being executed, the static data contains data that does not change throughout the life of the program (e.g. global variables), the heap is used to store dynamically allocated memory, and the stack is used to store local variables.

However, since this address space lives along with other address spaces around it, we want to make sure that no other thread or process can access this address space. We also want to make sure that this thread cannot access any other address space, especially address/memory where the OS lives.

This can be done using a simple check during each memory access. When the thread accesses a memory location, the OS first checks if the memory location is within the address space allocated to this thread. If it is not, then the thread or process can be killed. The address space therefore has a "base" (the start of the address space), and a "bound" (the end of an address space).

This simple model has lots of drawbacks, but the biggest one is fragmentation.

To fix that, we use a paged virtual address space. We divide the entire virtual address space into equal size chunks called pages. Each page has a base, and the bound is the global page size. All pages are of the same size, so it is easy to manipulate memory this way.

The hardware then translates addresses using a page table. A special hardware register stores a pointer to the page table. This lets us treat memory as a set of pages that are put into page frames. The thread or process operates on virtual addresses, and the virtual addresses are translated to hardware addresses by the page table.

Processes

A process is an isolated execution environment with restricted rights. It has an address space and contains one or more threads. You can think of it as an encapsulation of a running program. The process has an address space that is isolated from every other address space, but the threads inside the process share everything, including memory, hardware resources, network, i/o, etc.

This lets us do concurrency easily. Multiple threads in a single process can run on different cores (if available).

But one question that needs to be addressed is what stops a process from changing some data that it owns, that should not be changed (for example, the page table pointer for this process)? If a process could change its own page table pointer, the security offered by the concept of address spaces would mean nothing.

This is solved by the fourth concept: privilege levels and dual mode operation.

Dual Mode Operation

The hardware itself has at least two modes, kernel mode and user mode, which essentially give us privilege levels. Almost all processes installed in the machine can only run in user mode, which stops them from accessing/changing sensitive data. Certain operations are prohibited when a process is running in user mode, for example changing the page table pointer, disabling interrupts, interacting with the hardware directly, or accessing any memory outside its address space.

If the process needs to perform a "sensitive" operation, the OS provides a controlled transition between user mode and kernel mode.

The OS can itself switch between user mode and kernel mode if any of the following happen -

1. syscall
2. Interrupt
3. Exception

Lecture 3 - Abstractions 1: Threads and Processes

Multiprocessing vs. Multiprogramming

Multiprocessing vs. Multiprogramming is similar to the idea of concurrency vs. parallelism. Multiprocessing is when you have multiple physical cores, and you use them to perform operations at the same time completely independently. Multiprogramming is when you divide a program into multiple jobs/processes/threads.

Concurrency is when you are *handling* multiple things at once, not necessarily *doing* them. Parallelism is when you do multiple things *simultaneously*.

Threads

A thread can be in one of 3 states - running, ready, or blocked. A running thread is a thread whose instructions are actually being executed, a ready thread is a thread that is waiting for CPU time, and can be swapped out with the current running thread, and a blocked thread is a thread that is waiting for something to happen before it can run (typically i/o).

To write a multithreaded program in C, we usually use pthreads. The "P" in pthreads stands for POSIX. POSIX is an attempt at standardizing the programming interface that different operating systems provide to programs running in user mode.

Usually the structure of a multithreaded program is that we have a "main" thread that is created when the process is created. The main thread creates as many new threads as it needs. These threads run, do their own thing, and then "join" with the main thread. The main thread then continues to run.

All threads in a process share the process's address space, but they have their own independent stack. These stacks grow and shrink independently, and sometimes may run into each other. In this case, the OS decides what to do. The thread may be killed, or its stack may be reallocated.

The important part about multithreaded programming is correctness. The program you write must be able to handle any interleaving of thread execution that the OS scheduler gives you. One way to ensure correctness is mutual exclusion.

If more than one thread is going to access some data, the thread accessing that data acquires a "lock" over that data. This ensures that only that thread can read or write to that location. Once the thread has done whatever it needs to do with that data, it releases the lock. The part of code that exactly one thread can access at once is called the critical section.

Processes

All processes are created by other processes. If that is the case, how is the first process created? The first process is called the "init" process, and it is created by the kernel.

The OS also provides a process management API, that allows our programs to manage other processes. The common functions provided by this API are -

1. `exit` - Terminate a process
2. `fork` - Create a complete copy of the current process
3. `exec` - Change the program being run by the current process
4. `wait` - Wait for a process to finish
5. `kill` - Send an interrupt-like notification to another process
6. `sigaction` - Set handlers for signals

Lecture 4 - Abstractions 2: Files and I/O

Semaphores

Semaphore are a kind of generalized locking mechanism. A semaphore is just a number associated with critical section of code. It was first defined by Dijkstra in the 60s. A semaphore is a non-negative integer that supports the following operations -

1. `P()` or `down()` - An atomic operation that waits for the semaphore to become positive, then decrements it by 1.
2. `V()` or `up()` - An atomic operation that increments the semaphore by 1, and then wakes up a waiting `P`, if any.

This concept of semaphores can be used in two patterns. One is for mutual exclusion, which is easy to see. We set the initial value of the semaphore to be 1. Each thread first checks the value of the semaphore before entering the critical section. If the value is greater than 0, the thread enters the section and decrements the semaphore. If the value is 0, the thread is put to sleep and waits for the semaphore to increment.

The other pattern is for waiting for a thread to finish execution and join the main thread (or some other thread). We set the initial value of the semaphore to 0. When the main thread tries to decrement it, it is put to sleep. Then, another thread finishes its execution and increments the semaphore. This wakes up the main thread, and the main thread then takes over the execution. Essentially, we made the main thread wait for some other thread to join.

Processes & Signals

A common way of having one program launch or quit other programs is using `exec()` and signals from the process management API. `exec()` stops the execution of the current process, and gives up control to some other process that is specified as an argument to `exec()`. This is how the shell starts new programs. Look at the example code below -

```
pid = fork();
if (pid == 0) {
    // This is the child process launched by the parent
    exec(...);      // Start some other (child) process
}
else {
    wait(&stat);   // Wait for the child process to finish and exit
}
```

Files

A key idea in Unix/POSIX is that everything is a file. This means that (almost) all system calls have an identical interface. POSIX gives an identical interface for -

1. Files
2. Devices (terminals, printers, etc.)
3. Networking sockets
4. Interprocess communication (pipes, sockets)

and more. This standard interface is based around the system calls `open()`, `read()`, `write()`, `close()`. If you are writing a device driver that doesn't necessarily fit into this interface, you can use `ioctl()`, which stands for I/O control. A "folder" is just a special type of file that maps to a set of other files/folders, instead of having any arbitrary content.

The high level abstractions of files is streams. A stream is just an unformatted sequence of bytes. Kubi then explains the API C gives us for working with streams/-files, which I won't write about here, since it is fairly simple and can be looked up if you need it.

File Descriptors

A file descriptor is just an integer that corresponds to an entry in a system-wide file description table that is maintained by the kernel. The kernel maintains a list of open files, and a list of processes and which files they have access to.

On a successful call to `open()`, which is a low level function to open files (or anything that behaves like a file, which is everything), a file descriptor for the corresponding file is returned to the user, and an open file description is created in the kernel.

For each process, the kernel maintains a mapping from the file descriptor to the open file description. On performing system calls (e.g. `read()`), the kernel looks up the open file description using the file descriptor and services the system call.

Since everything in POSIX is a file, stdin, stdout, and stderr also have file descriptors associated with them. The file descriptors associated with these are -

- stdin -> 0
- stdout -> 1
- stderr -> 2

Not only that, but this means that other hardware resources like sockets also have file descriptors associated with them.

File descriptors is how pipes can be implemented. If you take the file descriptor for the stdout for a process, and map it to the file descriptor for stdin for another

process (using the pipe function), then the output of one process is fed to the input of another process.

All calls to high-level file handling APIs like `fread()`, `fwrite()`, etc. are buffered at the user level. This is how they are able to provide us functionality like "read until the next newline". Moreover, all reads and writes are also buffered at the kernel level by the OS. Another reason for having buffers at both the user and kernel level is because buffering in user level is faster than having to access the kernel buffer, for which the process would need to go into kernel mode (or make a syscall).

When you `fork()` a process, all of its file descriptors are also copied. This means that both the parent and the child process can read and write to and from the same file. This is a useful way to have the processes communicate with each other. This works not only for files, but also for hardware resources like network sockets.

Lecture 5: Abstractions 3: IPC, Sockets

IPC

IPC stands for inter-process communication. The process model we have seen so far is specifically designed so that one process is not able to affect another in any way. This is by design, but it does make inter-process communication more difficult.

Therefore, we have to have a specialized way for two processes to communicate with each other, if both processes agree. One initial idea may be to communicate through reading and writing to files that are shared by processes. We have seen that forking a process creates file descriptors that are shared by both the parent and the child, so maybe one process can write to a shared file descriptor, and another process could read from it. The drawback to this approach is, as you might imagine, efficiency. Communicating with disk is really slow, and this approach becomes completely impractical as the number of processes increases.

The approach we use is to have an in-memory queue maintained by the kernel, which provides the same POSIX interface for reading and writing as reading and writing to a file. This is called a "pipe". You can create a pipe using the `pipe(int filedes[2])` syscall.

Sockets

When we extend the principle of IPC to two processes that are not running on the same machine, we get communication across the network. This involves a "network connection", which is essentially a pair of sockets connected to each other (for a bidirectional connection).

Actually transmitting data over the network is a whole another task, and this course does not deal with that. We assume that the messages we send over the network are correctly transmitted and reach the other device across the network.

Aside from the fact that these two sockets are on different physical devices, they work in the exact same way that a local socket (or pipe) works.

The client has a socket that is connected to a socket on the server. The client opens the socket (which may establish a TCP connection using a 3-way handshake), and can then read from it or write to it. Data written to the socket is transmitted over the network, arrives at the server, which reads it, and sends a response. The server then closes the socket once the client's request has been served.

In order to serve multiple requests at once, the server may create either a new process or a new thread for each request. Creating a new process for each request gives us better isolation, but is more heavyweight. Creating a new thread is faster, but doesn't offer the same isolation.

If a server creates a different thread for each request, then there is a chance that a spike in traffic can create so many threads that the throughput actually decreases. To avoid that, we create a group of threads called the "thread pool". The thread pool has a certain number of threads waiting to serve requests. We maintain a queue of requests that are waiting to be served, and when a thread from the thread pool becomes free, it dequeues the next request from the request queue. This way, there is an upper limit on how many threads can be created, and requests are still served concurrently.

Lecture 6: Synchronization 1: Concurrency, Mutual Exclusion

Concurrency

One of the most important and central parts of the operating system is the scheduler. The scheduler maintains a queue of the active processes in the system and chooses which process gets to go next.

The scheduler maintains a data structure containing the process control blocks, and distributes CPU time to the different PCBs it contains according to some policy. It can also distribute access to other hardware resources like memory, IO, etc.

The lifecycle of a process is shown in the diagram below -

The pseudocode of the scheduler basically looks like this -

```
while (1) {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(currPCB);
    LoadStateOfCPU(nextPCB);
 }
```

One can even argue that this is all that the operating system does.

Context Switching

When the scheduler decides to run a thread, it essentially gives up control of the CPU to the process it decided to run. It needs some way to get that control back from the process.

This can happen because of internal events, i.e. the thread yields control voluntarily, or because of external events like interrupts. The POSIX API provides a syscall `ptherad_yield()` or `sched_yield()`, that give up control of the CPU.

When a process yields, it goes into the kernel, tells the scheduler to run a new thread, and then performs a context switch. The context switch basically saves the state of all registers and the stack of the first thread/process into the TCB/PCB, and loads in the registers and the stack of the new thread/process.

Switching between threads is much faster than switching between processes (about 30x faster).

What happens if a process never yields or never does any operation (like waiting for I/O or disk read/writes) that causes it to voluntarily give up control of the CPU? In primitive operating systems like Windows 3.1, this would cause the OS to crash. Since the OS has completely given up control of the CPU to the currently executing process, there is no way for the OS to regain control of the CPU.

In that case, we use external interrupts. Today's computers have hardware timers that are configured to interrupt the CPU at a fixed interval. A hardware interrupt is designed to stop the currently executing process and transfer control back to the OS.

Mutual Exclusion

We have seen mutual exclusion in a previous lecture. This lecture goes over the concept again and shows some examples. When we write multithreaded code, a big problem is correctness. Since threads share memory, if multiple threads access the same memory at the same time, and if at least one of those threads is doing a write, then we run into a race condition. The result of the program in that case is unpredictable.

The way we fix this is by using locks or semaphores. Before entering a critical section of code where multiple threads are going to access the same memory, we acquire a lock over that memory. This lock only lets one thread access that memory at a time. Once the thread is done using that part of memory, it releases that lock.

Semaphores

Semaphores are a higher level abstraction than locks. They are a little different, and can be used in a scenario where using locks is cumbersome. The lecture gives an example where using a semaphore is better than using a lock.

Consider a publish-subscribe application with producers and consumers connected with a buffer. We want the enqueue and dequeue operations on the buffer to be atomic, and we want a producer to go to sleep when it tries to publish an event when the buffer is full, and we want a consumer to go to sleep when it tries to read from an empty queue.

Using a lock to enforce this behaviour is not really ideal (you can try to write the pseudocode for it and see what problems arise). Instead, we use semaphores. The constraints we have on the resource (the buffer) are the following -

1. If the buffer is empty, the consumer must go to sleep (if it tries to read)
2. If the buffer is full, the producer must go to sleep (if it tries to write)
3. Only one thread can read or write at a time (standard mutual exclusion)

The general rule of thumb is that you have one semaphore for each constraint. So we would have to following semaphores -

1. `Semaphore fullBuffers`
2. `Semaphore emptyBuffers`
3. `Semaphore mutex`

Then the pseudocode for producers and consumers would look as follows -

```
Semaphore fullSlots = 0;            // No full slots initially
Semaphore emptySlots = buffSize;    // The max size of the buffer
Semaphore mutex = 1;                // Standard mutex

Producer(item) {
    semaphoreDown(emptySlots);      // Decrement empty slots or wait until we have space
    semaphoreDown(mutex);           // Mutex for actual enqueue/dequeue
    Enqueue(item);
    semaphoreUp(mutex);             // Release mutex after enqueue/dequeue
    semaphoreUp(fullSlots);         // Signal that an item has been enqueued. This wakes
}

Consumer() {
    semaphoreDown(fullSlots);       // Decrement full slots or wait until there is somet
    semaphoreDown(mutex);           // Mutex for actual enqueue/dequeue
    item = Dequeue();
    semaphoreUp(mutex);             // Release mutex after enqueue/dequeue
    semaphoreUp(emptySlots);        // Signal that an item has been dequeued. This wakes
}
```

Notice that the `semaphoreDown()` operation is used to keep track of the amount of available resources, in this case the space in the buffer, and the `semaphoreUp()` operation is used to signal that some change has taken place in the resource. This is why we have two semaphores. The `fullSlots` semaphore signals that at least one item has been enqueued, and wakes up any sleeping consumers. The `emptySlots` semaphore signals that the queue has some space left, and wakes up any sleeping producers.

Lecture 7 - Synchronization 2: Semaphores, Lock Implementation

This lecture gives an intuition about how locks are actually implemented in an operating system. We want an interface which lets us acquire a lock on a critical section of code, run the critical section, and then release the lock.

The simple implementation is described by the acquire and release pseudocode given below -

```
int value = FREE;   // Flag describing if the critical section is being executed or not
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread to sleep;
        go to sleep();      // careful!
    }
    else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (thread is sleeping on this lock) {
        wake up thread;
    }
    else {
        value = FREE;
    }
    enable interrupts;
}
```

A small problem with this implementation is in the `Acquire()` code, where if the `value` is `BUSY`, we put the thread (that is trying to enter the critical section) to sleep, and then we go to sleep ourselves, without ever re-enabling interrupts!

This problem is solved by following the convention that whichever thread wakes up after we go to sleep re-enables interrupts.

Another problem with this implementation is that we have to run this code at the kernel level, since it involves disabling and enabling interrupts. This means that the user level code has to go into the kernel by making a system call in order to acquire and release a lock.

Making a system call is around 25x costlier than a normal function call. So if we have

hundreds of threads all acquiring and releasing locks, this can make the computer unusably slow. Therefore, we need a locking implementation that can acquire and release locks without having to go into the kernel.

Lecture 8 - Synchronization 3: Atomic Instructions, Monitors, Readers/Writers

The previous lock implementation we saw was impractical because it involved a system call every time we wanted to acquire or release a lock. A system call is at least 25x more expensive than a normal function call, so this approach limited our performance severely.

What we need is just to be able to read a value (of the lock), check if it is 0 or 1, and then write to the value atomically. If we removed the need to disable and re-enable interrupts for this operation, we would be able to implement locks at the user level.

Therefore, all architectures provide atomic instructions for this purpose. These instructions can read, modify, and write data atomically, such that they cannot be interleaved. Some of these instructions are -

```
// Return the value stored at address, and store a 1 there
test&set(&address) {
    result = M[address];
    M[address] = 1;
    return result;
}

// Swap "value" with value stored at address
swap(&address, value) {
    temp = M[address];
    M[address] = value;
    value = temp;
}

// If value at address == reg1, store reg2 there and return success, else failure
compare&swap(&address, reg1, reg2) {
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    }
    else return failure;
}
```

Implementing Locks With test&set

The lecture gives a few ways in which we can implement locks using `test&set()`, but I won't write about them here because they are very intuitive to come up with but result in busy waiting, so we don't use those implementations anyway.

Monitors

Monitors are a concurrent programming paradigm that are, in a way, cleaner and more intuitive than semaphores. A monitor is a lock and zero or more *condition variables* associated with it. A condition variable is just a variable that represents some constraint on the system.

We have the following functions to use with monitors - 1. `Wait(&lock)` - Automatically release the lock and go to sleep 2. `Signal()` - Wake up one waiter, if present 3. `Broadcast()` - Wake up all waiters, if present

For example, let us look at a database with multiple readers and writers. The constraints we have are that the readers can access the database only when there are no active writers or waiting writers, and the writers can only access the database when there are no active readers.

We maintain the following state variables - 1. `int activeReaders = 0` 2. `int waitingReaders = 0` 3. `int activeWriters = 0` 4. `int waitingWriters = 0` 5. `Condition okayToRead = NULL` 6. `Condition okayToWrite = NULL`

Then the code for a reader will look as follows -

```
Reader() {
    // First check self into system
    acquire(&lock);
    while ((AW + WW) > 0) { // Is it safe to read?
        WR++; // No. Writers exist
        cond_wait(&okToRead,&lock);// Sleep on cond var
        WR--; // No longer waiting
    }
    AR++; // Now we are active!
    release(&lock);
    // Perform actual read-only access
    AccessDatabase(ReadOnly);
    // Now, check out of system
    acquire(&lock);
    AR--; // No longer active
    if (AR == 0 && WW > 0) // No other active readers
        cond_signal(&okToWrite);// Wake up one writer
    release(&lock);
}
```

Lecture 10 - Scheduling 1: Concepts & Classic Policies

In a modern OS, all IO operations have the same interface, usually consisting of `read()`, `write()`, `open()`, `close()`, etc. This applies to file IO, network IO, peripheral device communication, etc. This is possible because of device drivers.

A device driver is software that runs on the kernel thread that corresponds to the user-level thread, and is responsible for providing the standard interface that the OS requires. This way, the device driver takes care of implementing each operation, and exposes it under the standard names like `read()`, `write()`, etc.

The life cycle of an IO request is shown below -

The flow chart is pretty self-explanatory in itself. What is important w.r.t. scheduling is that the thread requesting IO may be put to sleep either by the kernel in the second layer, when the kernel decides that the thread is requesting a large amount of data from some device, or by the top half of the device driver, when it receives the IO request, sends it to the device to process, and then puts the thread to sleep as it waits for the response to come back from the device.

Scheduling is the process of taking all the threads and processes that are waiting to run and choosing which thread or process should get the CPU next. Implementing a first-in-first-out policy may seem like the right thing to do, but that is not always the case. If one user has just one process running and another user has 5 processes running on a shared machine, the second user would get more CPU time just because they have more processes running.

Therefore, the scheduling algorithm that we decide to use has to be chosen carefully depending on the kind of workload we are trying to support. For example, consider the CPU burst model. Most programs use the CPU for a short burst of time, then perform some IO, then use the CPU, and so on. So each program has alternate "bursts" of CPU time and IO time. Now we can decide to schedule programs which are expected to have shorter CPU bursts first, or longer CPU bursts first. Each policy has its own advantages and disadvantages.

FIFO Scheduling

FIFO scheduling, also known as first-come-first-serve scheduling, is the most obvious scheduling policy. You run every process in the ready queue to completion, and then pick the first waiting process in the ready queue.

However, as you may expect, this is a bad strategy because a process that runs really long can take over the CPU. If a process runs infinitely long, it may take over the computer entirely. Also, if some processes come along which need very little CPU time to finish, they get stuck behind a long-running process and have to wait a long time.

This is a huge problem for responsiveness of the system. If the thread that handles user input via the keyboard has to wait for a few seconds before a long-running process finishes, then the user may not see what they typed on the screen for a few seconds after they type it.

This effect is called the convoy effect, where a short-running job is stuck behind a long-running job.

Round Robin Scheduling

An improvement over FIFO or FCFS is Round Robin. Round Robin puts an upper limit on the amount of time any process can run, and if it exceeds that time limit, it is preempted and swapped out with another process. This is good, because now the thread that handles user input via the keyboard can still get to run if a long-running process is trying to hog the CPU.

This is a simple but effective solution for the convoy effect, but for it to work well in the real world, we need to tune time quantum that is given to each process. If the time quantum is too large, we effectively revert back to FCFS, and if it is too small, we spend so much time context switching that even the short-running jobs are swapped out multiple times, which ultimately *reduces* the response time. In the lecture, Kubi shows the effect of different choices of time quantum on the average completion time and the average waiting time for each job.

Shortest Job First Scheduling

This is a hypothetical version of scheduling that puts the jobs with the shortest running time first. It is hypothetical because it requires us to know the future and know exactly how long each job is going to run. A preemptive variation of this scheduling policy is called Shortest Remaining Time First (SRTF), where if a new job comes along which has an even shorter time to completion that the currently running job, the currently running job is preempted and the new job starts running.

Although this policy is proven to be the best for improving response times, it has another problem (besides needing to know the future). If short jobs keep coming along, long jobs keep getting stuck behind short jobs and potentially never run. If the short jobs never stop coming, long jobs never run. This issue makes this scheduling policy extremely unfair.

In the real world, we use various heuristics to predict how long each job will take, along with trends observed from the previous runs of the job. So we can have an imperfect version of SRTF scheduling.

When writing a scheduler for a real world OS, you may have to come up with and test multiple scheduling policies to see which one works best for the type of workload you have.

Lecture 11 - Scheduling 2: Case Studies, Real Time, Forward Progress

Multi-level Feedback Scheduling

This is a scheduling scheme in which we maintain multiple ready queues instead of just one. Each queue has a different time quantum for round-robin and represents different priority levels.

In the image above, the topmost queue has a time quantum of 8 and is for real-time, short-running jobs. The bottom most queue is for long-running jobs. Each new job gets added to the highest priority ready queue when it arrives for the first time. Then, every time its time slice expires, if it still hasn't finished, it gets dropped one level to the queue below with a longer time slice.

The CPU is shared among all the queues according to some percentage - each queue gets a certain amount of CPU time. That is, the topmost queue could get 70% of the CPU time, the one below it gets 20%, and the bottom most queue gets 10%. That means that 70% of the time, tasks in queue 1 are executing with a time quantum of 8, for 20% of the time, tasks in queue 2 are executing with a time quantum of 16, and for the remaining 10% of the time, tasks in queue 3 are executing FCFS.

Case Study: Linux O(1) Scheduler

The Linux scheduler for a while was a variant of the Multi-level feedback scheduler you saw above. Instead of 3 read queues, it had 140, of which the first 100 (of highest priority) were for real time tasks, and the remaining 40 were for user tasks.

It was a O(1) time scheduler, which means that it took constant time to run and figure out which job should be scheduled next. It achieved this by computing a large amount of heuristics every time it was called and maintaining a copy of all the 140 ready queues. Once a job was dequeued and started to run, it was put into the copy queue, and then the pointer to the active queue was swapped to point to the copy queue.

However, even though the scheduler worked well, it because impossible to maintain because the number of heuristics it used and their complexity grew too much, to the point that it was just discarded and completely replaced with a new policy.

Real-Time Scheduling

Real-time scheduling refers to the scheduling for a real-time operating system, which often needs to ensure that jobs that arrive are run before a certain "deadline". For example, the embedded system in a car that controls the brakes needs to ensure that once the driver presses the brake, the brakes are applied within a certain deadline.

These deadlines are critical to the safety of the application, and thus we need predictability of performance. A common scenario in real-time scheduling is scheduling

periodic tasks that occur with a period P and have a burst time of C. The condition in this case is that a task should finish its computation of C before its period P is over. In this case, round robin scheduling does not work - we end up missing deadlines.

Earliest Deadline First (EDF)

This scheduling policy simply takes the task which has the earliest deadline and runs it. Every time it has to switch, it selects the job which has the earliest deadline.

EDF is proven to be optimal if you have a reasonable number of jobs to run. If the number of jobs keeps increasing, at one point EDF will not be able to guarantee deadlines. That point is given by -

$$\sum_{i=1}^{n} \frac{Ci}{Di} \leq 1$$

Where C_i is the burst time of task i and D_i is the deadline or period of task i. This inequality just says that the percentage of CPU used by each task should add up to be less than or equal to 100%. If it is greater than 100%, the scheduler obviously won't be able to meet all deadlines.

Ensuring Progress

We have already seen the problem of starvation - when a thread or a job fails to make progress for a long time because of the scheduling policy. This is not the same as deadlock, since starvation could possibly resolve under the right conditions.

Lecture 12 - Scheduling 3: Deadlock

Linux Completely Fair Scheduler

The scheduler used in Linux (as of the lecture in August 2021) is called the "Completely Fair Scheduler", and it approaches scheduling differently than a traditional round-robin or real-time scheduler. It tries to give every thread the same amount of time on the CPU, so if there are N threads, each thread gets 1/N of CPU time.

As each thread is running, the scheduler keeps track of how long the thread ran. Each thread has an allocated time slice, and if a thread gives up the CPU and goes to sleep before its time slice has expired, then it gets more priority to run when it wakes up, because the scheduler wants to make sure that each thread gets the same amount of time on the CPU.

The Linux CFS has a target latency, which is the period of time over which it wants to make sure that all threads are run at least once. For example, if the target latency is 20ms, and there are 5 threads to be scheduled, it will schedule each thread to run for 4ms, and keep track of the threads that run for a shorter (or longer) time.

However, if the target latency is 20ms, and there are 200 threads to be scheduled, then each thread gets a 0.1ms time slice, which may be too small, since we have to spend some time context-switching. Therefore, the CFS scheduler has a lower bound on the time slice, say 1ms. So in this case, all 200 processes will get a 1ms time slice, and we will not be able to meet our latency goal of 20ms.

So far this looks like a slightly modified version of round-robin, but the difference becomes clear when we see how CFS handles job priorities. It allows you to give each job (thread) a priority, and then instead of calculating the time slice for each thread equally, it calculates the time slice to give based on the weight of the job. The weight of the job is decided by the "nice" value of the job. A nice value typically ranges between -20 to 19, and a higher nice value means lower priority.

The weight of a job is calculated as

$$weight = \frac{1024}{1.25^{nice}}$$

The virtual time slice of a job is then calculated as

$$max(minslice, \frac{Wi}{\sum_p Wp} * targetlatency)$$

Where w_i is the weight of the ith job. CFS then tries to give every job this virtual time slice instead of real time.

Deadlock

A deadlock is a type of starvation that never resolves, and only occurs non-deterministically, when the schedule creates a cycle of accessing resources. A deadlock can be caused due to any resource - locks, terminals, memory, printers, sockets, pipes, etc.

There are 4 conditions that are necessary (but not sufficient) for deadlock to occur -

1. Mutual exclusion - There has to be a resource that only one thread can access at a time.
2. Hold and wait - A thread holding at least one resource is waiting to acquire more resources held by other threads.
3. No preemption - Resources are only released voluntarily by the thread, and the kernel cannot preempt it into giving up the resource
4. Circular wait - There exists a set of threads {T1, T2, . . . , Tn} such that
   - T1 is waiting for a resource held by T2
   - T2 is waiting for a resource held by T3
   - . . .
   - Tn-1 is waiting for a resource held by Tn
   - Tn is waiting for a resource held by T1

In order to detect deadlock, we can build a resource-allocation graph. A resource-allocation graph is a graph with 2 types of vertices - one representing threads and one representing resources. It has directed edges that signify the following -

- A directed edge from a thread vertex T to a resource vertex R means that the thread T is requesting resource R.
- A directed edge from a resource vertex R to a thread vertex T means that the thread T has ownership of resource R.

The slides show a few examples of resource allocation graphs. Below, a resource vertex has one or more dots which represent an instance of the resource.

Detecting Deadlock Just because there is a cycle in this graph does not mean that there will be deadlock. To detect deadlock, we have a simple algorithm. Instead of describing it here, I'll just attach a screenshot of the lecture slide.

Dealing with Deadlock There are 4 common ways of dealing with deadlock -

- Deadlock prevention - Write your code in such a way that deadlock won't occur. This means that you trust the apps running on your machine to be written this way as well.
- Deadlock recovery - Let deadlock happen, and then figure out how to recover from it, maybe by rolling back a transaction (if transactions and rollbacks are supported).

- Deadlock avoidance - Dynamically delay resource requests so that deadlock doesn't happen. Requires you to do extra work while writing and maintaining the kernel.
- Deadlock denial - Don't deal with deadlock

The Linux operating system makes sure that the kernel is not involved in any deadlock, but ignores any deadlock that happens in applications.

There is an algorithm that can detect if allocating a resource to a particular thread would lead to deadlock called the banker's algorithm (described in the lecture and the slides).

Lecture 13 - Memory 1: Address Translation and Virtual Memory

Virtual Address Spaces

An address space is a set of protected memory addresses that are accessible to a program. These are virtual addresses distinct from actual physical memory. Not all processors have virtual address spaces. In order to have virtual address spaces, you need support for address translation from hardware.

The mapping between physical addresses and virtual addresses is done by a unit called the Memory Management Unit (MMU). This mapping lets every process assume that it has the entire RAM to itself, and is able to access any address. Whenever the processes accesses a particular (virtual) address, the MMU maps it to the specific physical address assigned to it, and gives the process data from the actual physical address. However, the process never knows exactly which physical address the data it accessed came from.

Memory Translation

The most important function of the MMU is to be able to allocate non-contiguous chunks of memory to a single process. Without the MMU, in very early operating systems, when a process was loaded into memory, it was assigned a big contiguous chunk of memory. However, allocating contiguous chunks of memory quickly leads to fragmentation of the memory.

To get around this problem, the MMU only allocates contiguous chunks of memory to different "segments" of a process. For example, each process needs memory for its code, static data, heap, and stack. The MMU only allocates contiguous memory for a process's code, static data, heap, and stack, but all of these four parts may be stored in different chunks of physical memory as well as virtual memory.

This approach has another advantage. If we allocate a contiguous chunk of memory for a process' heap, we can map that same chunk of memory to another process' heap and get inter-process communication.

The information about where each segment has been stored is contained in the segment map. For each virtual address, we take the first few bits of the virtual address. These bits are used to index into the segment map and find the base of the chunk of physical memory allocated to this segment. These first few bits are usually the first 2 or 3 bits. Then, these first few bits are replaced by 0s, and the resulting virtual address is considered to be the offset from the base that we got from the segment map. So this offset is added to the base, and we get a physical address.

In the lecture Kubi steps through an assembly program and goes through all the steps I described above in order to get a physical address from a virtual address, around the 1:07:00 timestamp.

This segment table is unique to a process, and when a process is swapped out during a context-switch, this segment table also needs to be swapped out.

Now, if we have so many processes that we run out of room for storing each segment for each process, we can actually use disk to store these segments for a process. This is called swapping, and this lets us use the huge size of the disk as DRAM. However, accessing disk has a huge cost in terms of time, and each disk access takes roughly on the order of 1M cycles. Therefore, this is not really a practical strategy, but only used when there is no other option.

So far, this strategy helped us a little bit with fragmentation. Instead of allocating a huge chunk of contiguous memory to a process, we were able to divide the memory a process requires into different segments, and only allocate those contiguously. However, when the size of the segments keeps growing, especially with the process stack and heap, fragmentation starts coming up again.

Fragmentation can be of two types - external and internal. External fragmentation is the type we have been talking about so far. But internal fragmentation is when we allocate a small(er) chunk of memory to a segment, but the process doesn't need that entire chunk. This leads to internal fragmentation.

In order to better deal with fragmentation (both external and internal), we need a smaller quanta than a segment of a process. We can come up with a smaller quanta, and call it a "page".

Paging

Now each process gets a page table which maps a virtual page number into a physical page number. If you have a virtual address, the last few bits of the address are used to represent an offset into the page. So if a page is of 1024 bytes, we will need to use the last 10 bits of the virtual address to specify the offset into the page. The remaining bits of the address are used to represent the virtual page number.

Now that you have this virtual address, we use the virtual page number as an index into the page table. The page table gives you the physical page number. You then access that physical page and go to the offset you have, and you get to the address you wanted.

However, this is still not quite what we want. Next lecture extends this idea to better deal with fragmentation.

Lecture 14 - Memory 2: Virtual Memory, Caching and TLBs

So far, the implementation of paging that we have seen is not very efficient. If we assume that a page is 4KB, then we need 12 bits to represent the offset into each page. If we have a 32 bit machine, then the remaining 20 bits will be used as an index into the page table, which means that the page table has to have 2^20 entries, each of 4 bytes, which means we use around 4MB to store the page table for a process. That may not be too much for today's machines to handle, but if we have a 64 bit machine, then using a 12 bit offset, we use 52 bits as an index into the page table, which means our page table takes around 36 exabytes.

How do we fix this? We can split the page table into multiple levels. Instead of just having one "level" of virtual address translation, we can have two. Considering the example of a 32 bit machine, we would have 12 bits for the offset into the page, and then instead of using the remaining 20 bits as an index into the page table, we will use only the top 10 bits. These 10 bits will be used to index into a table which will point to the next "level" of the page table. Then the next 10 bits will give us the actual physical page.

As processor cores get bigger and bigger and we start getting to 64-bit architectures, even multi-level page tables start becoming infeasible. In that case, we can switch to using a hash table. A hash table has the advantage that its size only depends on the number of pages used by the process, and not the size of the virtual address space. However, this means that we don't have cache locality, and managing this hash table is more complex than a simple page table.

TLBs

There is one issue with the concept of paging that we must address. The issue is that we have effectively made all caching useless because in order to check if an address is in the cache, we have to translate it to a physical address, for which we have to go to the page table, which is stored in DRAM!

A simple solution to this is to add another cache for storing virtual addresses and their mapping to physical addresses. This is called the TLB - Translation Look-aside Buffer.

Lecture 15 - Memory 3: Caching & TLBs, Demand Paging

Demand paging is a simple but powerful idea. Most modern programs require a lot of memory, but they don't use all of their memory at once. On average, they follow the 90-10 rule - 90% of the time the program executes 10% of its code, and 90% of the time, the program only needs 10% of its memory.

We can use this trend to our advantage in order to keep only the most used data in DRAM and cache, and the rest out on disk, instead of trying to load the entire program and the data it needs into the DRAM.

In demand paging, we don't fully populate the page table when we load a program into memory. As the program runs, it tries to access a page that is not mapped by the page table. In this case, we get a page fault, which is equivalent to a cache miss. We then go out to disk and load the page into memory, update the page table, and run the instruction again.

Lecture 16 - Memory 4: Demand Paging Policies

Demand paging aims to improve the average memory access time. The average memory access time (AMAT) can be calculated as -

```
AMAT = Hit Rate x Hit Time + Miss Rate x Miss Time
```

Which can be simplified to

```
AMAT = Hit Time + Miss Rate x Miss Penalty
```

You can think of demand paging as being a cache managed in software. The pages live on the disk, and only the pages that are needed are brought from the disk into the DRAM. The mechanism of demand paging is as follows -

1. Processor tries to access an address
2. It checks the page table entry for the address
    1. If the entry is valid, we go ahead and access that physical address
    2. If it is invalid, we get a page fault, and we need to get that page from the disk into DRAM

In case we get a page fault, the OS gets the page from the disk into DRAM -

1. Choose an old page to replace (assuming we don't have free space in DRAM)
2. If the old page is modified, write it back to disk
3. Change the page table entry and TLB entry to be invalid
4. Load new page from disk into DRAM
5. Update the page table entry and invalidate the TLB entry to force the MMU to fetch the new address
6. Retry the memory access that faulted before

Memory Access Behaviors

There are two common types of memory access behaviors that a process follows -

1. A process has a specific set of pages that it accesses, and it doesn't access any other pages. This working set can change over time. This is called the working set model.
2. A process does not have a specific set of pages it accesses, but it accesses some pages more frequently than others. In this case we can say that some pages are popular for a process, and give each page it accesses a rank depending on the number of times it is accessed. This is called the Zipf model.

Cache hit rate under the working set model grows in steps as the cache size increases.

Cache hit rate under the Zipf model grows logarithmically as the cache size increases.

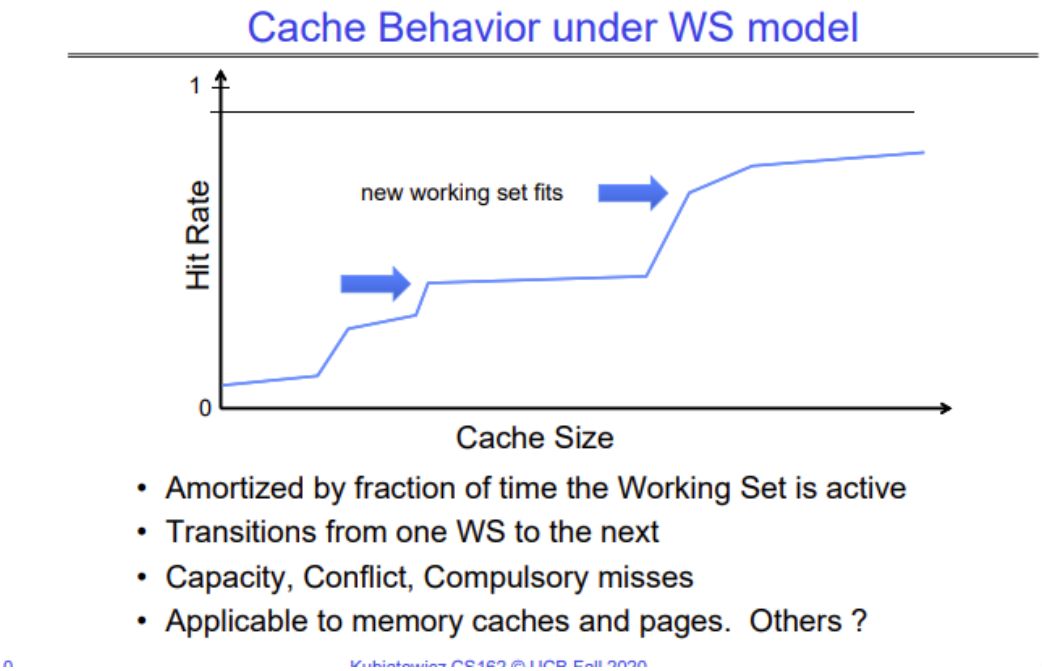


Figure 21: Cache behavior under the working set model

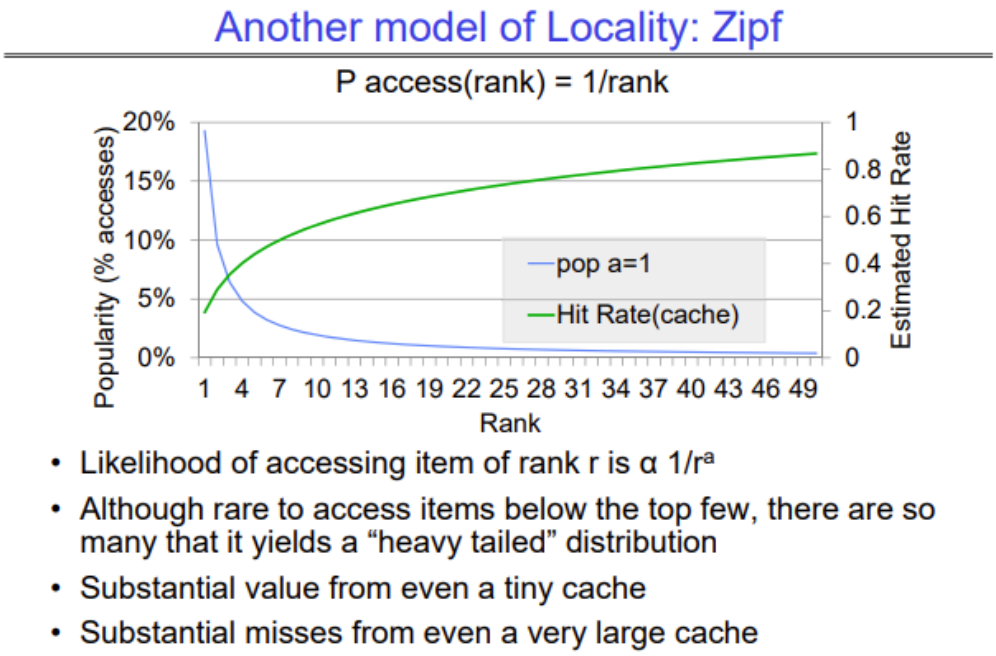


Figure 22: Cache behavior under the Zipf model

Demand Paging Replacement Policies

When we implement demand paging, we have to take a special care about the replacement policy we use to choose which page to evict from the DRAM to make space for the new page we will bring in from the disk. This is because the cost of a miss/page fault is huge, since going out to the disk takes around 1M cycles. This means that if you start page faulting, your performance comes to a grinding halt very fast.

In the lecture, Kubi shows a rough calculation which shows that even if you have a miss rate of 1 in 1000, you slow down by a factor of 40! If you want to keep your effective memory access time to be only around 10% more than accessing a page from DRAM, you need a miss rate of less than 0.00025% or 1 in 400,000.

The lecture covers a few replacement policies -

1. FIFO - Replace the page that came in first. This is a bad policy, since it is likely that the page that came in first could be a page that is accessed very frequently.
2. Random - This works well for caches since the miss penalty of a cache means going to the DRAM, which is not as bad. However, when the miss penalty is going out to the disk, we cannot risk randomly choosing a very frequently accessed page.
3. Min (Optimal) - Replace the page that will be used farthest in the future. This strategy is provably optimal, but we obviously can't implement it because we can't know the future.
4. LRU - LRU is a good approximation for Min, but it is not practical to implement since it involves operations that have high time complexity.

Approximating LRU - The Clock Algorithm

Since LRU is impractical to implement, we approximate LRU using something called the Clock algorithm.

We first link all the pages in memory in a circular linked list. We then keep a pointer to a node in this list, and call it the clock hand. On every page fault, we check if this node has been set as "used" or accessed by the processor. If it has, we reset its used bit to zero, and move the clock hand to the next node. We eventually reach a node which is not marked as used. This means this page is an old page which has not been used in a while, so we can replace it.

This is an approximation to LRU because we evict an old page, but not necessarily the oldest page. In the worst case, we could loop around the circular linked list and not find a single unused page. In this case, we eventually come back to the start node, whose used bit we had cleared, and evict that page. Therefore, in the worst case, the clock algorithm becomes FIFO.

Instead of keeping track of a single used bit, we could keep a counter, and give each page N chances instead of just 1 before we choose to evict it. This is a variation of the clock algorithm.

Lecture 17 - Demand Paging, General I/O, Storage Devices

An important design decision is how many page frames we should give to each process/thread. Too few page frames (memory) allocated to a process will lead to thrashing, where the process just spends all or most of its time swapping pages in and out of memory. Too many frames allocated to a process will waste memory that could have gone to a process that needs the extra memory.

We can use a static policy to decide the number of page frames, but it is better to have a dynamic policy that changes the amount of memory allocated to a process based on its behaviour. We can detect that a process needs more memory and that it is thrashing by monitoring the number of page faults it is encountering. A large page fault rate means a process is likely thrashing. We want to keep the page fault rate between an upper limit and a lower limit. We don't want the page fault rate to get too low, because that means we could be allocating too much memory to a process that doesn't need it.

Meltdown

In 2017, a bug was discovered that allowed a user level program to read kernel code, because of the way memory was mapped by operating systems. The lower addresses of virtual memory were mapped to user addresses, and the upper addresses of virtual memory were mapped to kernel addresses. The image below shows the organization for 32 bit and 64 bit machines. The memory map for 64 bit machines has a big hole (called the Canonical Hole), simply because the address space that can be mapped by 64 bits is huge, and no machine has that much RAM yet.

A user program accessing a kernel address would cause a page fault, and ultimately a core dump.

However, processors have something called speculative execution, which is when an instruction is executed even though it performs an illegal operation (like accessing kernel level addresses), or when a branch prediction is needed, etc, and when the illegal operation (or error) is discovered, all the registers relating to that instruction are cleared so that no data is leaked. This technique is usually beneficial and improves the speed of the processor.

Meltdown exploited this technique in a very clever way -

```
uchar array[256 * 4096];
flush(array);       // Make sure the array is not in cache

try {
    uchar result = *(uchar *) kernel_address;       // Try to access kernel memory
    uchar dummy = array[result * 4096];             // Force the processor to store (res
} catch () {;}
```

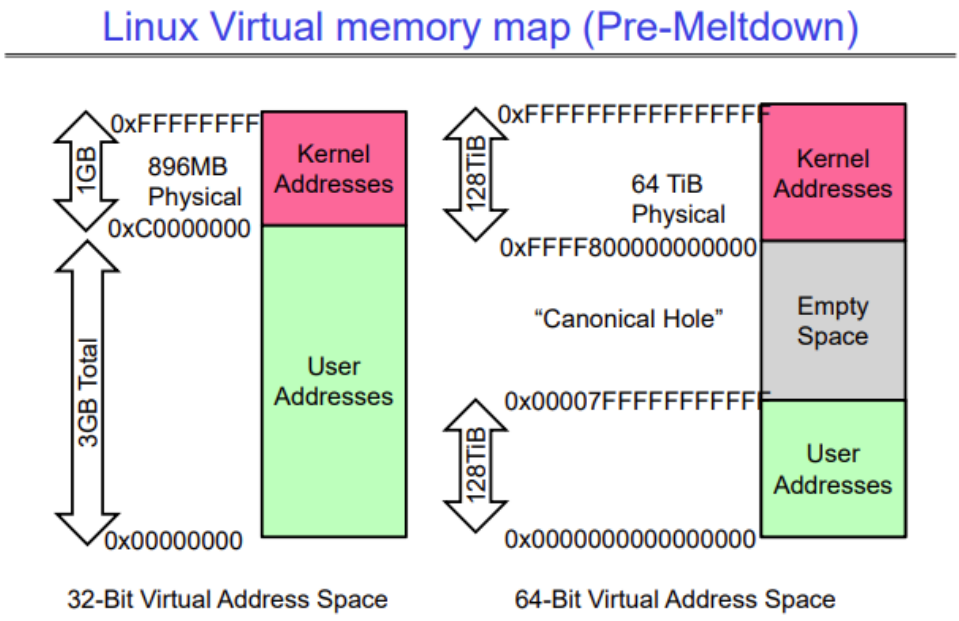


Figure 23: Pre-meltdown Linux Memory Map

We first try to access some kernel memory, which the processor does successfully because of speculative execution. We then immediately force the processor to cache the value we got from the kernel address in the next instruction. When the processor realizes that we performed an illegal operation, it clears the value of the registers and throws an error, which we catch. However, the value is still in cache. We then try to access each item in the `array` array, and measure the time it takes to access each item. When we get to the index that was in cache, our access time is very fast, so we can tell what value we got back from the kernel address we just accessed.

Lecture 18 - General I/O, Storage Devices, Performance

In previous lectures we have seen that the OS has a standardized interface for communicating with the external world. This means that even though there are hundreds of different types of devices, each of which has a different internal structure, the operating system can work with all of them because the devices support this standardized interface.

This interface is provided to the OS by the device driver. The device driver is a process that knows how to work with the specific hardware of the device, and can work with the device in order to provide the standard API calls to the OS.

There are three main types of devices -

1. Block oriented devices - Devices which access blocks of data, e.g. disk drives. When you read from or write to these devices you can only do so in "blocks" (typically 4KB), and not individual bytes.
2. Character oriented devices - Devices which communicate individual bytes or character, e.g. mouse, keyboard, etc.
3. Network oriented devices - Devices which deal with packets and are slightly different from block or character oriented devices, e.g. ethernet, wireless, bluetooth, etc.

To each of these devices, there are three main types of interfaces -

1. Blocking interface - When a process requests some data from this device, the process is blocked and possibly put to sleep.
2. Non-blocking interface - When a process requests some data from this device, the device returns whatever it can return immediately, and the process continues executing. If the process needs more data, then it must keep asking the device for more data.
3. Asynchronous interface - When a process request some data from this device, it continues executing and the device starts fetching the data. Once the data is ready, the device signals to the process saying that the data it had requested is ready.

Hard Disk Drives

Although SSDs are becoming more common now, HDDs are still around. HDDs work by storing information on magnetic disks. A track positioned on top of the disk can read or write to the disk.

The disk latency (time taken for a read/write request to be completed) is the sum of the queueing time, disk controller's execution time, seek time, rotational delay, and transfer time.

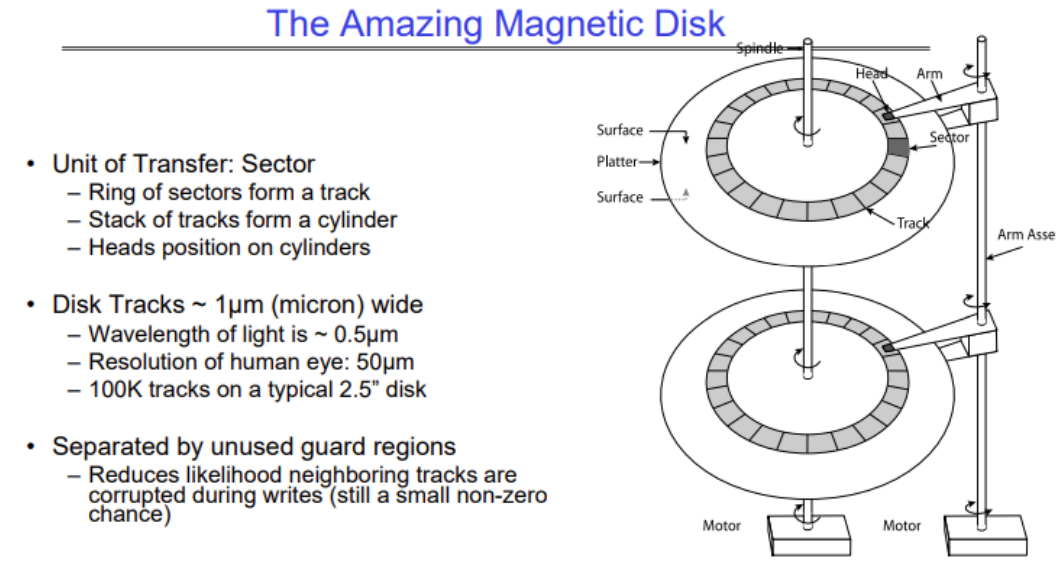


Figure 24: Magnetic Disks

All of these factors means that HDDs are painfully slow. Their data transfer rate is usually 50 to 250 MB/s.

Lecture 19 - Filesystems 1: Performance, Queueing Theory, Filesystem Design

When we measure the performance of an I/O system, we have to consider queueing time, because the data can pass through multiple queues before our I/O request is satisfied. These queues are present in the OS itself (the ready queue, wait queue, etc. that the process has to go through), and the I/O device itself.

Each queue has a latency L and a bandwidth B. In a simple system, the (maximum) bandwidth is the inverse of the latency.

$$B = \frac{1}{L}$$

If we consider a pipelined system, if we have a pipeline of K stages, our bandwidth increases K times, as expected.

$$B = \frac{K}{L}$$

Another important parameter could be the number of things/operations inside the system at any time. If the latency is L, and the current bandwidth is B, then there are $ B*L $ things/operations inside the system at this time (roughly, assuming the bandwidth isn't constantly changing). This is known as Little's Law.

Lecture 20 - Filesystems 2: Filesystem Design, Case Studies

The filesystem is responsible for bridging the gap between the API exposed by the operating system to the user (which is byte-oriented), and the API provided by the device driver for the disk to the OS (which is block oriented). The disk only lets you read and write data in blocks which can range from 512 bytes upto 4KB. However, the user is given the illusion that they can access any byte on the disk as they need.

The filesystem provides two entities - files and directories. Files are just a collection of bytes, and a directory is an index that maps a name to a set of files. Each sector of the disk is given an integer address, and the filesystem remembers which files are stored at which addresses.

The filesystem also needs to track additional information such as -

- Free block addresses, in case it has to create new files.
- Which blocks belong to which file, in order to read and write existing files.
- Which directories contain which files.

Since we want all of this data to be persistent after the machine is shut down, we will need to store this data somewhere on the disk. This may become a recursive problem! We can solve this using a small set of fixed addresses for this metadata, or some standard positions for the root of the filesystem.

Components of a File System

A file system has two components - a directory structure map, and a file header structure. The directory structure map is a mapping in a directory that maps a file name to a file number (called an inumber). This inumber is used to find the inode of a file (also called the file header). The inode of a file contains a list of all the blocks that belong to the file.

When a process opens a file, a name resolution is performed in the directory structure to first find the inumber of the file. This inumber is used to look up the inode of the file. This inode describes all the blocks that belong to the file, and now we can read/write these blocks as we want.

A directory in a file system is actually a file that contains a list of mappings of the format `<file_name: file_number>` for all the files the directory contains. Each of these `<file_name: file_number>` mapping is called a directory entry, and these are stored in a format that the OS decides. The OS doesn't let a user process read the raw bytes of a directory (the directory entries), because if these entries are corrupted by an incorrect user process, all the data in the directory could be lost.

When we perform actions on these directories like creating them, moving them, or deleting them, we are actually just modifying this structure. So syscalls like `open`,

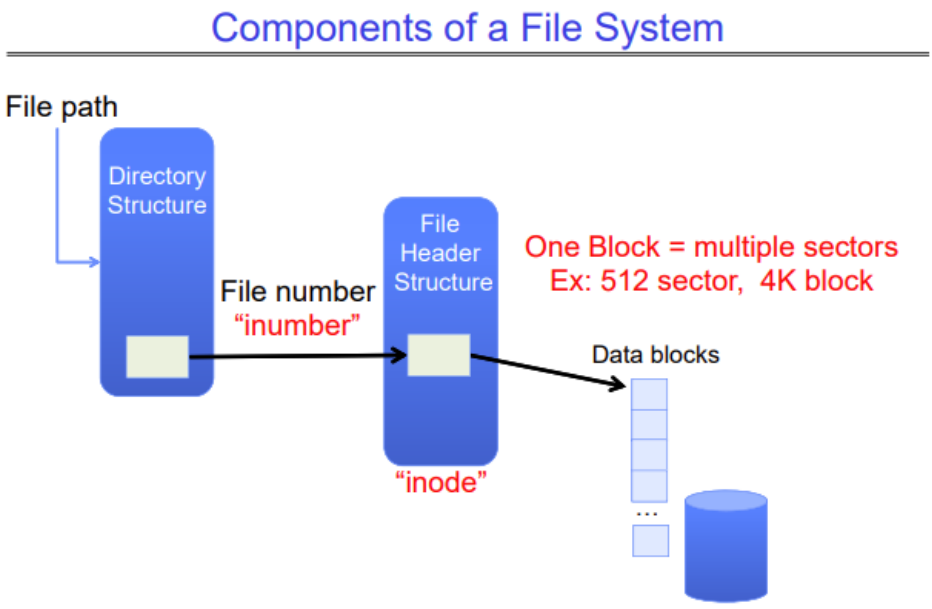


Figure 25: Components of a file system

`create`, `readdir` traverse this structure, and `mkdir`, `rmdir` add/remove entries from this structure.

For example, say you wanted to open a file `/my/book/count`. The OS follows the following steps -

1. Read the file header (inode) for the root `/`.
2. Using the inode, read the first data block for the root.
3. In this data block, find the directory entry for `my`.
4. If we find it in this block, we proceed, otherwise we read in the next block and look for it there.
5. Once we find the directory entry for `my`, we read the first block for `my`.
6. We look for the directory entry for `book`.
7. We read the first block for `book`.
8. We look for the directory entry for `count`.
9. We get the inode for `count`, and now we can read the blocks for `/my/book/count` as we need.

This is all stored on the disk, however, when we try to open a file, these structures are also created in memory -

Each process has a table that contains the file descriptors of the files the process has opened. When the `open` syscall is executed, a file descriptor is created for the

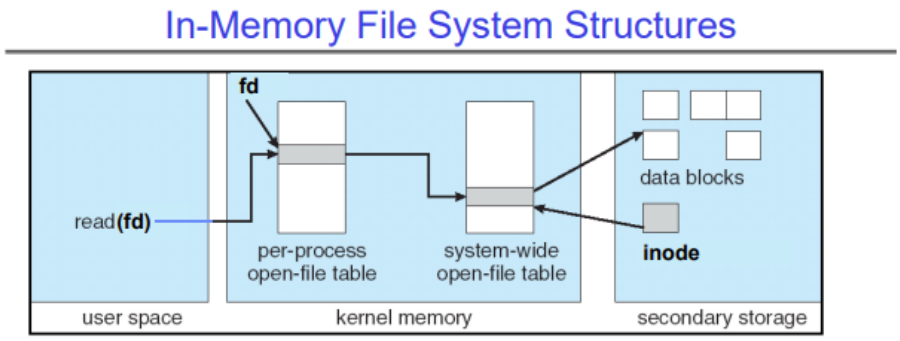


Figure 26: In-memory file system structures

file and stored in this table. This file descriptor points to an entry in a system-wide open file table, which contains the inode of this file, fetched from the disk and put into RAM.

Case Study: File Allocation Table (FAT)

FAT is a very simple file system. It basically just maintains a table that maps from a file number to a disk block number. Assuming we have a way to turn a file path into a file number, we just look up the file number in the FAT, and it gives us the disk block for the start of that file. If a file spans across multiple blocks, each FAT entry stores a pointer to the FAT entry for the next block of the file.

FAT stores directories in the same way, since directories are just files that store file name to file number mappings. The root directory is stored in FAT entry 2.

The FAT filesystem is simple by design, and is therefore not the most efficient. It is prone to fragmentation, random access is slow, and sequential access can also become slow over time because of fragmentation.

Case Study: Berkley Fast File System

Instead of a file allocation table, the Berkeley Fast File System uses inodes. Inodes first appeared in BSD 4.1 (Berkeley Standard Distribution). An inode is a structure

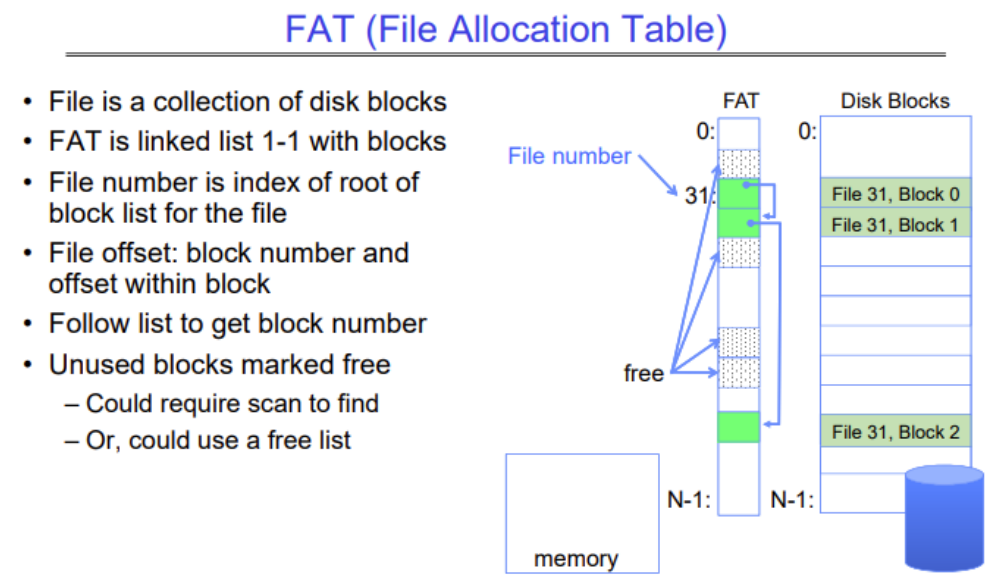
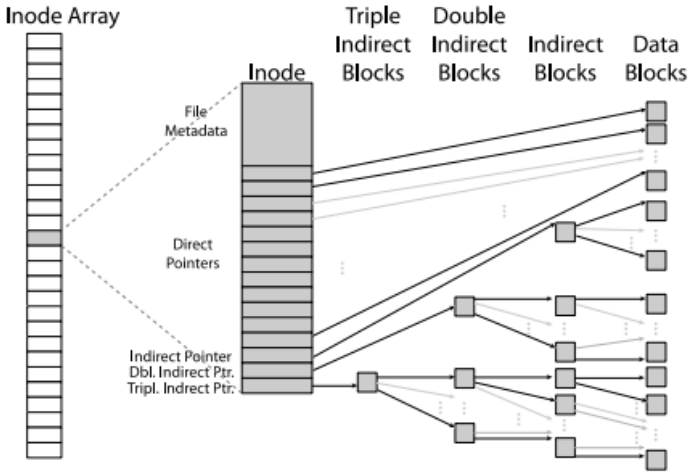


Figure 27: FAT Filesystem

that holds information about a file. Every file has an inumber, which is an index into an array of inodes. The inodes store some metadata about the file, and then they have a few pointers -

1. Direct pointers - These pointers point directly to the first few blocks of the file. Each pointer points to one block. There are around 10 to 12 direct pointers, which point to the first 10 to 12 blocks of the file. These pointers are maintained because most files are small, so those small files can be accessed directly.
2. Indirect pointers - Indirect pointers point to indirect blocks, which are one level above the data blocks. These indirect blocks mean that you have to follow a tree-like structure to get to the data blocks. This allows the file system to store very large files efficiently.

Inode Structure



Filesystems 3: Case Studies, Buffering, Reliability, Transactions

In previous lectures, we saw that a directory is just a file mapping file names to file numbers. There are two such types of mappings -

1. Hard links - Maps a file name to a file number. The fist hard link is created when the file is created. There can be multiple hard links, and hard links can be created and removed using syscalls like `link()` and `unlink()`. When all hard links to a file are removed, the file is deleted.
2. Soft links - These are also called symbolic links (symlinks). They map a file name to a different file name.

Memory Mapped Files

Traditional I/O involves reading and writing to regions of memory on disk, usually with buffers and caches in between. Memory mapping of files is a technique in which we "map" the file into RAM by giving it a virtual address, and then we can use the RAM as a cache without needing buffers, and the RAM is backed by the file in memory. This technique is often used to set up inter process communication.

Buffer Cache

The buffer cache is just a region in memory that is used to cache (hash table) disk blocks, inodes, directory entries, etc. It is a write back cache with LRU replacement, and is completely implemented in software by the kernel. It also supports prefetching for sequential disk access. It also writes blocks back to the disk periodically, in case of a crash.

It improves the performance a lot, but it also means that we have to worry about reliability. In the lecture, Kubi describes 3 important properties of a system -

1. Availability - The probability that the system is responsive, but not necessarily working correctly.
2. Durability - The ability of the system to recover despite faults.
3. Reliability - The ability of a system to perform correctly under the stated conditions.

Transactions, End-to-End Arguments, Distributed Decision Making

Transactions in a file system are a mechanism to improve the reliability of the system. They extend the concept of atomic updates from memory to persistent storage. A transaction is an atomic sequence of writes that takes the file system from one state to another. If the system crashes before the transaction is complete, none of the writes in the transaction make it through and the file system is restored to the state it was in before the transaction.

Transactions are generally implemented using a log-based approach. There is a log which supports only one atomic operation - appending to it. When we write to a file as part of a transaction, all the individual steps of the write are written as instructions in the log. Once all the instructions are written, we write a "commit" instruction to the log, which says that the transaction is complete and the changes can be applied to the file system.

The OS then goes through the instructions in the log and applies them one by one. If we crash at any time before reaching the commit instruction, we just start from the first instruction and perform the writes again. This is okay because we're just overwriting the same data. If we crash before the commit message is written to the log, all the instructions for this transaction are just thrown out from the log and we say we rolled back.

We use NVRAM for the log to make sure unapplied commits are not lost in case of power failures.

A file system that uses a log in this way to support transactions is called a transactional file system. A transactional file system can be log structured or journaled. In a log structured file system, the data stays in log form. In a journaled file system, the log is only used for recovery and all data is written directly to the disk.

Log Structured File System (LFS)

If we take the concept of a log further, we can make the log itself the file system. It is one continuous sequence of blocks that data is just appended, not modified. As actions are taken (files or directories are created, read, deleted), those actions are also appended to the log. The present state of the file system is then inferred by replaying the actions stored in the log.

As you might expect, this type of filesystem is efficient for writes but not so efficient for reads. However, a good block cache with enough space can make this file system feasible, especially when used with flash memory instead of a hard disk drive. An implementation of a log structured file system is F2FS, a flash file system used in many mobile devices, including the Pixel 3.

Distributed Decision Making, Networking, TCP/IP

General's Paradox

The general's paradox is a problem in which two generals have to decide on a particular time to coordinate their attack. They can only exchange physical messages through messengers, who can be captured by the enemy. If they successfully decide on a time to attack, they will succeed. If they cannot decide on a time and attack at different times, they will fail.

The general's paradox doesn't have a solution. Since the messengers can be captured, you can never be sure that the last message you sent was received. So even if you send an ACK, you can never be sure that the other person got the ACK.

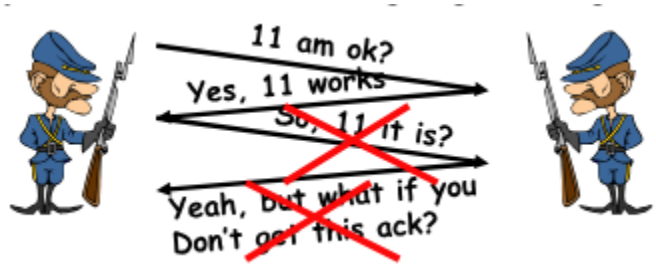


Figure 28: General's Paradox

Two-Phase Commit

The two-phase commit protocol was developed by Jim Gray, a Turing award winner and Berkeley alumni. The two-phase commit protocol solves a problem related to the General's paradox, since we cannot solve the general's paradox itself.

The 2PC algorithm has one coordinator node and n workers who are trying to decide whether to commit or abort a transaction. The transaction can be a database transaction, a distributed file system transaction, or any other action that can be committed or aborted.

The high-level overview of the algorithm is -

1. The coordinator asks all workers if they want to commit or abort
2. All workers decide individually if they are ready to commit or if they want to abort
3. All workers reply with their desired action
4. If the coordinator receives "commit" from all workers, it sends a "global-commit" message to each worker, which causes each worker to commit their transaction
5. If the coordinator receives at least one "abort", it sends a "global-abort" message to each worker, which causes each worker to abort their transaction

All workers and the coordinator have an internal log in which they record their decision so that they will remember their decision even after a crash.

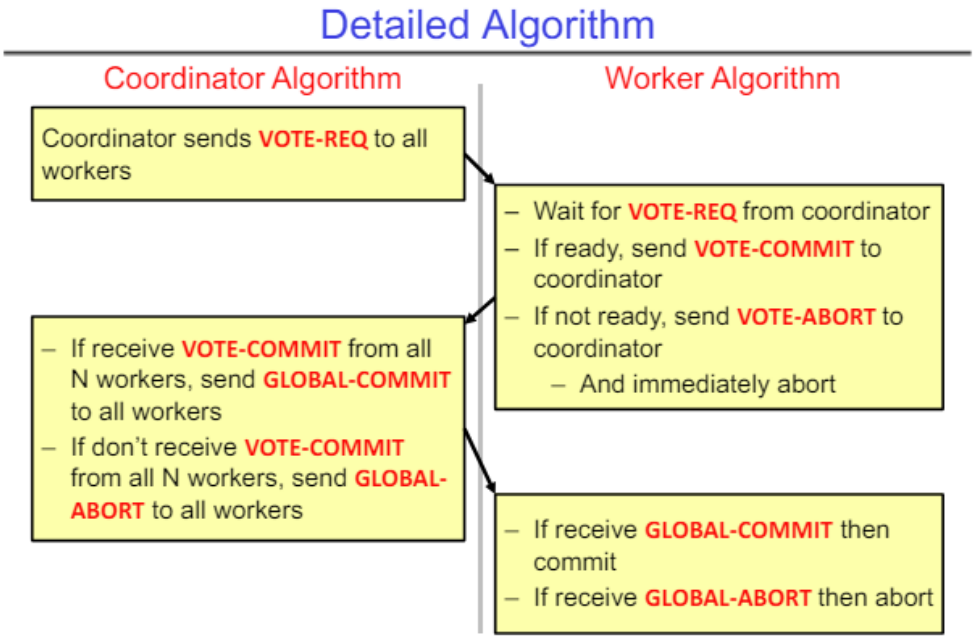


Figure 29: 2PC algorithm

A key drawback of the 2PC algorithm is that it is blocking. If a node fails or keeps failing, the other nodes are blocked and cannot make progress. A blocked node can also hold resources like locks, which make the blocking problem worse. There are some alternatives to the 2PC algorithm, such as the 3 phase commit, which has 3 phases instead of 2, PAXOS, a complicated distributed decision making algorithm used by Google in its internal systems, etc.

Another limitation of the 2PC algorithm (and also the alternatives to it) is that it assumes all nodes are working correctly, and no node is malicious. If a distributed system can have malicious nodes, you need a different class of algorithms. This situation is demonstrated by the Byzantine general's problem.

Byzantine General's Problem

The Byzantine general's problem has n players (nodes), out of which one or more can be malicious. One of the nodes is the General (leader), and has to send a boolean message to all nodes. The outcome must be that all correctly working nodes perform the same action, regardless of how the malicious nodes try to throw them off.

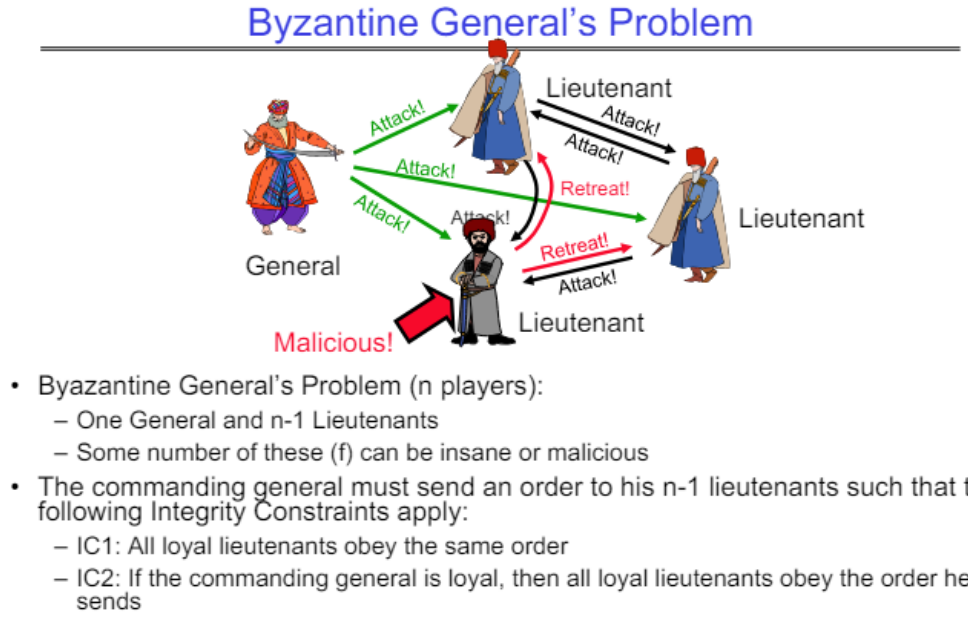


Figure 30: Byzantine General's Problem

The leader itself could be compromised. In this case, all the correctly functioning nodes still perform the same action, but that action may not be what the leader communicates to each node.

The lecture doesn't actually describe the algorithms that solve this problem, but today's algorithms are $ O(nˆ2) $, and block chain based algorithms can even be $ O(n) $ in the number of messages that need to be exchanged between the nodes.