MIT 6.046 - Design & Analysis of Algorithms

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.

```python
# 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 $\mathrm{T(n)} = 2\mathrm{T(n/2)} + \mathrm{O(n)}$, 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.

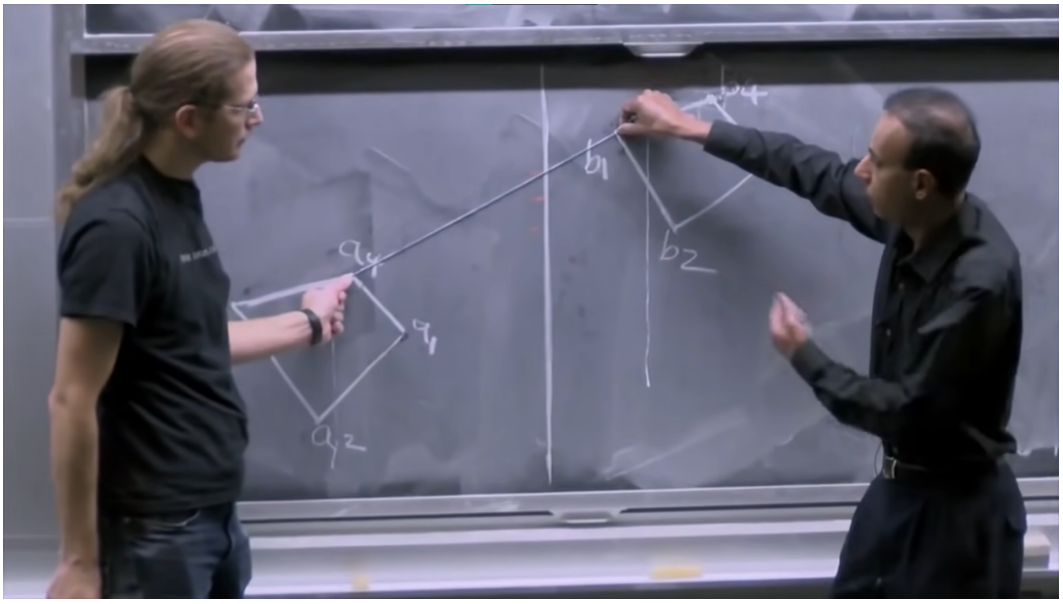

Figure 1: Image

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.

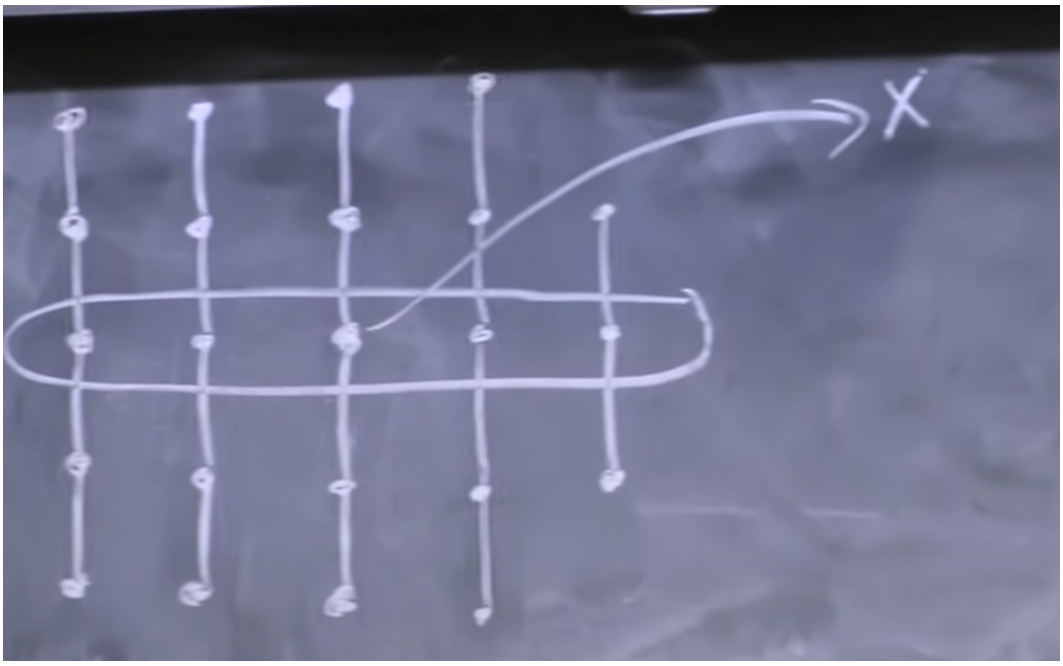

Figure 2: Image

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).

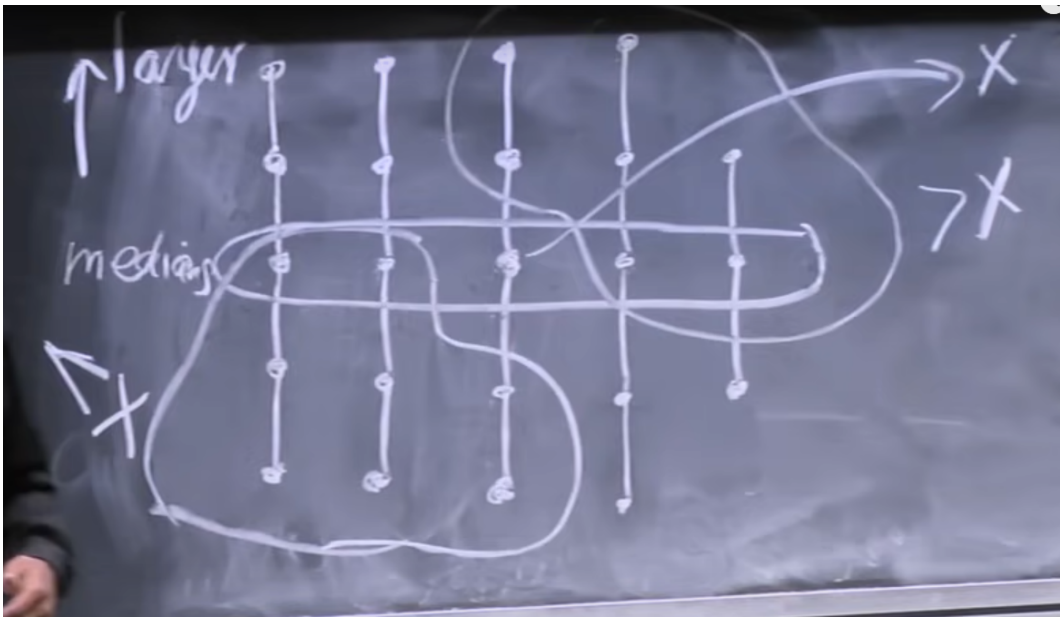

Figure 3: Image

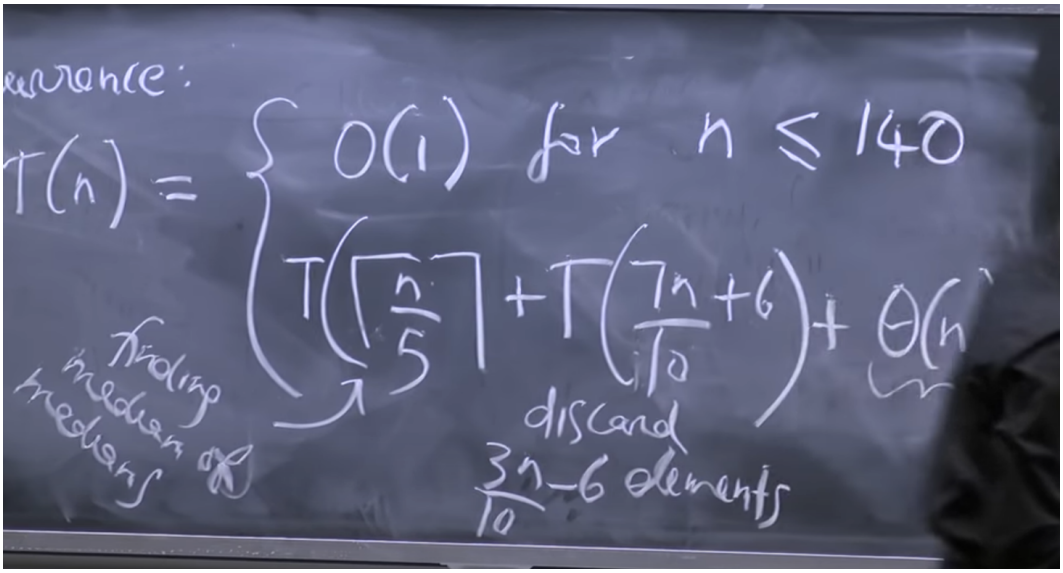

Figure 4: Image

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.

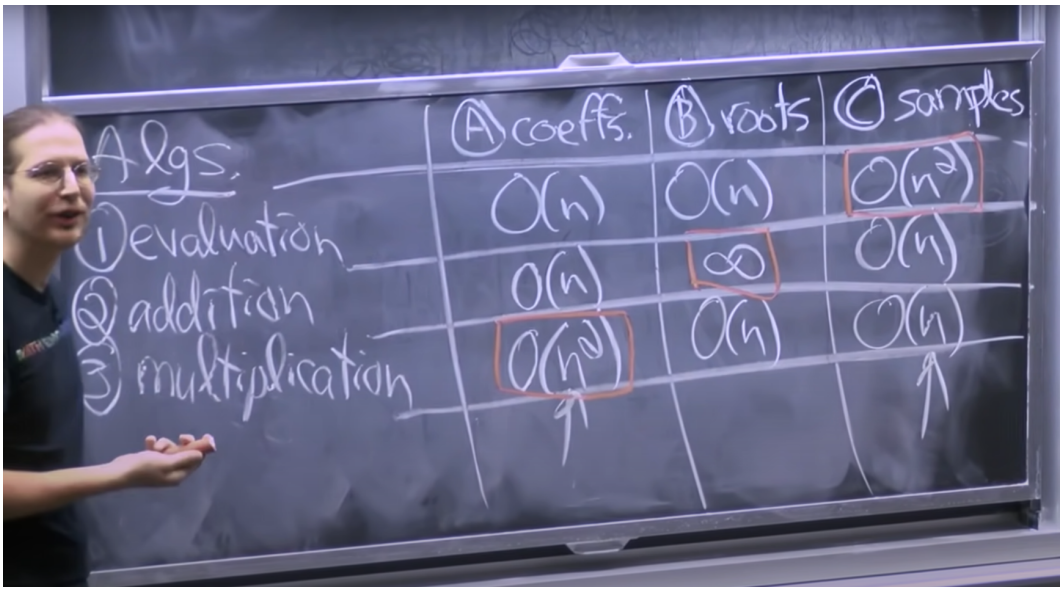

Figure 5: Image

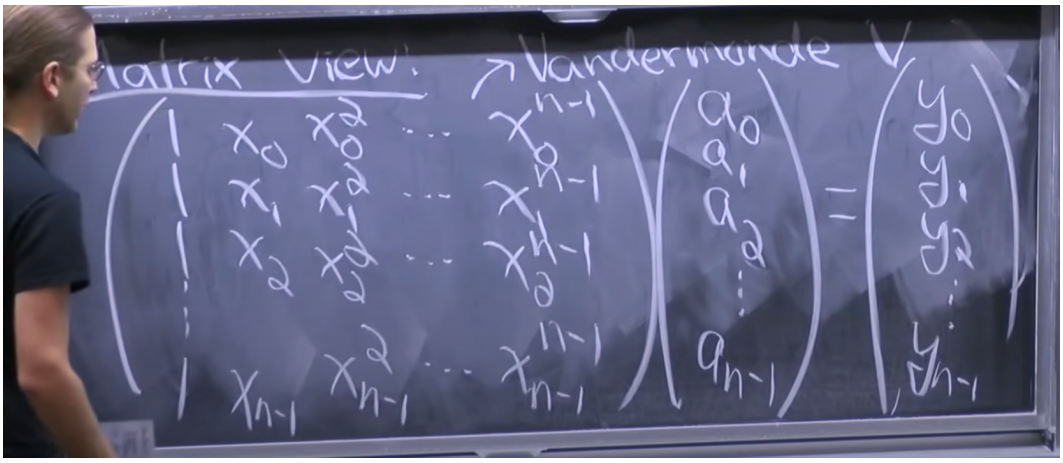

Figure 6: Image

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.