MIT 6.046 - Design & Analysis of Algorithms Lecture 1 - Interval Scheduling

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.