Optimization Problems

Optimization Problems

We have been looking at problems such as selection and sorting. These problems don't really involve the notion of making a choice, or more specifically, the best choice.

We haven't been looking at problems where we want to choose the optimal answer. There isn't really an "optimal" sorted list. There are good and bad ways to ***achieve*** the right ordering, but there is not a concept of the "best sorted order".

Optimization Problems: Examples

Typically problems with choices:

- Shortest Path ("short" could be cheap, etc.)
  - Internet traffic
  - driving directions
  - air travel
- Minimum Spanning Tree
  - given a graph, build the cheapest tree that touches all nodes
- Bin Packing
  - given a group of items of different sizes, and a set of containers, how can you place items in containers to minimize the number of containers used?
- Scheduling
  - given the Colony Ballroom at the Stamp Union and a list of events, with the goal of getting as many events as possible in the room, how do you select the events to OK?
  - given list of courses you'd like to take, what's the most that you can fit into a schedule?

Activity Scheduling

We have a single resource and have ***n*** requests to use that resource.

- the resource can not be shared
- the requests each have a start and finish time $request_i[s_i,f_i)$

Your function must take the requests and return a list of approved requests such that the most possible number of requests are granted.

Consider the following questions…

- How many potential answers are there?
- How long does it take to determine whether a candidate solution would actually work?
- Given a list of valid candidates, how long would it take to choose the ***best*** solution?

Finding Solutions Faster

Can we do better than $O(2^n)$?

Let's think about creating sub-problems that (if solved) could help us solve the larger problem.

Given S as the set of requests, define S_{ij} as $\{r_k \in S | f_i \leq s_k < f_k \leq s_j\}$

always true

- We can call this the set of all requests that could fit between $request_i$ and $request_j$.

We can now do the following:

- sort our list of requests by finish time (when there is a tie, sort by start time)
- Add fictitious requests $request_0$ and $request_{n+1}$ as boundary markers.

Find the largest non-conflicting subset of $S_{0,n+1}$

- If we find the largest **non-conflicting** subset of $S_{0,n+1}$, then we have our solution.
- If $i \geq j$ then S_{ij} is empty.
- If Sij is empty for any reason, then there's nothing to compute.
- If it is non-empty, then there must exist some request $request_k$ where $i<k<j$ that will appear in the optimal solution.
- Using $request_k$ as part of the solution creates two sub-problems to optimize: S_{ik} and S_{kj}.

How can we implement this?

We could write this as a recursive definition for matrix c (for **c**ount):

- define c[i,j] as the maximum number of requests that are Compatible with each other from S_{ij}.
- If S_{ij} is non-empty, then there is a request $request_k$ that is used, and c[i,j] can be expressed as c[i,k]+1+c[k,j]

We now have a definition, but we don't know who $request_k$ is yet!

We could just try all of them…

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} \text{ is empty} \\ \max_{i<k<j} \left(c[i,k] + 1 + c[k,j] \right) & \text{otherwise} \end{cases}$$

Dynamic Programming

Notice that once we compute a particular c[a,b] value, we could just store that solution and look it up later if it's ever needed again…

- This is very much like one of our Fibonacci improvements from earlier in the semester.

In dynamic programming, we often work from the bottom of a recursion tree, back up towards the top.

- Start by solving for sub-problems of size 1, then of size 2, then of size 3, etc…

Pseudo Code

```
Initialize the matrix c to all zeros.
Assume we have an array r of request
   records.
for d=1 to n+1
    for i=0 to n-d+1
        j=i+d
        if (r[i].f<=r[j].s)
            for k=i+1 to j-1
                if (
                 ((r[i].f<=r[k].s)
                       &&
                  (r[k].f<=r[j].s)
                  )
                       &&
                  (c[i,k]+1+c[k,j]>c[i,j])
                 )
                 then c[i,j]= c[i,k]+1+c[k,j];
```

What is the runtime of this algorithm?

A Faster Solution

Greedy algorithms look for the "locally optimal" choice and take it.

For this problem, a greedy approach would be:

- Sort the list of requests.
- Take the first request in the sorted list and assign it to the room.
- Remove everyone who conflicts with that request.
- Repeat on remaining requests until
- Announce the solution.

The Greedy Solution is Optimal!

By taking the first request, we only eliminate:

- Other requests that end at the same time as this one.
  - This is fine since we could only have chosen one of all of these overlapping events anyway.
- Other requests that overlapped this one at some time period.
  - Again, this is fine for the same reason.

In the next homework, you will need to determine the runtime of this solution and compare it to the two previous solutions' runtimes.