Optimization Problems Recap

ACTIVITY SCHEDULING (chapter 16)

Scheduling (potentially overlapping) requests of different duration, such that the following constraints are satisfied:

A. Requests must not overlap.
B. Number of requests must be maximal.

Example:

1 - 2:00
1:30 - 2:30
2:00 - 3:00

Assumption:

We have all information available to us.

fictitious

start request

(-∞, 1) (1, 2) (1.5, 2.5) (2, 3) (3, ∞)

fictitious

end request
① **Brute Force Approach**

- Sort list, e.g., by start time. $O(n)$ w/ Counting Sort or by finish time.
- Exhaustively enumerate all possible answers (valid + invalid) and get rid of invalid ones. $O(2^n \cdot n)$
  
  # of lists

  \[-\infty, 1\) (1, 2) (1.5, 2.5) (2, 3) (3, \infty) \]

  \[-\infty, 1\) (1, 2) (2, 3) (3, \infty) \]

  | - Find list with max # of requests $O(2^n)$
  
  **TOTAL**: $O(2^n \cdot n)$

  Can we do better? We want polynomial.

  Can we apply D+Q?
Dynamic Programming Approach

- Sort list by end time. (Break ties by sorting on start time)
- Add fictitious requests: \( \text{request}_0, \text{request}_{n+1} \)
- Find largest non-conflicting subset of \( \text{request}_0, \text{request}_{n+1} \) using divide and conquer.

\[ S_{i,k} \]
\[ S_{k,j} \]
\[ \text{Defn: } S_{i,j} = \text{set of requests between } i \text{ and } j. \]
\[ = \{ r_k \in S | f_i \leq s_k \leq f_k \leq s_j \} \]

Algorithm:

\[
\text{OPTIMAL-COUNT}[i,j] = \max (\text{OPTIMAL-COUNT}[i,k] + 1 + \text{OPTIMAL-COUNT}[k,j])
\]
Compute sum for all possible values of \( k \) and find max. Top-level recursive call:

\[
\text{OPTIMAL-COUNT}[\emptyset, n+1]
\]

**Dynamic programming:**

- Store optimal-count values from sub-problems and use these to solve bigger sub-problems.

- \( C[i,j] = \max \# \text{ requests from } S_{ij} \text{ that are compatible with each other.} \)

- If \( S_{ij} \neq \emptyset \), then request \( r_k \) exists s.t.:
  \[
  C[i,j] = C[i,k] + 1 + C[k,j]
  \]

- Try all \( r_k \)'s:
  \[
  C[i,j] = \begin{cases} 
  0 & \text{if } S_{ij} = \emptyset \\
  \max_{i \leq k \leq j} (C[i,k] + 1 + C[k,j]) & \text{otherwise}
  \end{cases}
  \]
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 $c[i,j] = c[i,k]+1+c[k,j]$;

Let's look at our example again:

$$(-\infty, 1) (1, 2) (1.5, 2.5) (2, 3) (3, \infty)$$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What does this chart tell you?
- The max number of requests that can be fulfilled.

Does it tell you how to fill them?
- No! Would need additional bookkeeping.
- Need to save information along the way.

- In the homework, you will need to determine how to return a schedule from the resulting chart.

Important: There could be a different ways to grant requests. You should focus on extracting one valid schedule.
(Reverse engineer the solution at the end.)
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 $(r[i].f <= r[k].s)$
        if $(r[k].f <= r[j].s)$
           if $(c[i,k]+1+c[k,j] > c[i,j])$
              then $c[i,j] = c[i,k]+1+c[k,j]$;

What is the runtime of this algorithm?

for $d = 1$ to $n+1$ $O(n)$
for $i = 0$ to $n-d+1$ $O(n)$
for $k = i+1$ to $j-1$ $O(n)$

$O(n^3)$ worst case

$< O(n \cdot 2^n)$

Can we do better?
The DP solution is overkill
Can we do better than $O(n^3)$?
- Make decisions as data streams in.
- Never look back.

Greedy approach!

- Can't just make Y/N decision as data comes in. That's too greedy. (Adversary could mess things up.)

- So, do some work first: Sort! (Sort by finish time.)
- $O(n) \Rightarrow$ linear sorting; our data falls in a particular range!

| 1-2 | 1:30-2:30 | 3-4 |

Which do we throw out?

Pick first! 2PM gives us more time to schedule other things today! Don't choose 1:30-2:30 because we lose 30 min with no gain.

This is why greedy works!!
3. **GREEDY APPROACH**

// Look for "locally optimal" choice and take it.

- Sort list of requests by finish time. $O(n)$
- Take first request in sorted list and put it in the result list. $O(1)$
- Remove everyone who conflicts with that request $O(n)$
- Repeat on remaining requests until sorted list is empty.
- Return the result list.

---

- When you eliminate a choice, you don’t have to examine it again!!

- Amortized linear time.

Can this possibly be optimal?
The Greedy Solution is Optimal!

- By taking the first request, we only eliminate:

  ▪ Other requests that end at the same time as this one. (Sorting by finish time is key!)

     ↓

     This is fine because we could only have chosen one of all these overlapping requests anyway.

  ▪ Other requests that overlapped at some time period.

     ↓

     Again, this is fine for the same reason.

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