Optimization Problems

Problems we've seen so far:
- selection
- sorting
- matrix multiplication

only one valid answer

new kind of problem:
optimization problem
- many valid answers
  but some of those answers are better
  than other answers
- we want the optimal answer

usually optimality is measured
by minimizing or maximizing
something.
→ a “metric”
Example: room reservation.

A valid answer consists of a set of reservation requests that don't overlap in time.

Say there is a fee for making a reservation.

An optimal answer is valid and gives the maximum amount of fees.

Alternative optimization criterion: minimize cleaning costs.

Then an optimal answer is to accept no reservation requests.

— Still valid and there will be nothing to clean!

Which answer is optimal depends on the optimization criterion.
types of algorithms for optimization problems

- dynamic programming
- greedy algorithms

we want to find an optimal solution quickly.

That means we may try to make a series of choices each based on only a little info (looking at all info may be slow) to lead to a hopefully optimal solution.
Examples of Optimization Problems

**Shortest Path**
- find an optimal route
  - maps.google.com
  - minimize distance
  - minimize time

- find an optimal flight itinerary
  - kayak.com, expedia
  - minimize time
  - minimize cost

**Minimum Spanning Tree**
- given a graph, find the cheapest tree that touches all nodes

**Bin Packing**
- items of different sizes and containers
- goal: minimize # of containers used

**Scheduling**
- room reservations
- doctor appointment requests
Activity Scheduling

single resource cannot be shared.

n requests to use resource.

\( \text{each request has form } \quad \text{request: } [s, f] \)

output: list of approved request such that you max the # of requests serviced.

example: doctor appointments

patient A requests 2 - 3:30 PM
patient B requests 2:15 - 2:45 PM
patient C requests 3:00 - 3:15 PM
patient D requests 1:45 - 2:20 PM

max # of patients that can be seen if you only see one at a time.
One way to do this: \((\text{BRUTE-FORCE})\)

1. Exhaustively enumerate all possible answers (valid and invalid):

\[
\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \ldots + \binom{n}{n}
\]

\[
= \sum_{i=1}^{n} \binom{n}{i} \quad \text{hard to simplify!}
\]

So try Truth Table Approach...

<table>
<thead>
<tr>
<th>request</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>in or not in returned list</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

How many rows? \(2^n\)
BRUTE FORCE
algorithm for activity scheduling

1. exhaustively enumerate all possible lists.
   \((2^n \text{ lists})\)

2. Identify if each answer is valid. (no requests overlap)
   - compare every request in a list to all other requests.
     \(n^2 \cdot \#\text{lists} = n^2 \cdot 2^n\)
   - sort each list, look for overlap
     \(O(n) \cdot \#\text{lists} = n \cdot 2^n\)
   - compare all requests to each other
     \(n^2 + 2^n(n-1) = O(n \cdot 2^n)\)

3. Find valid list with max # of request in it: \(2^n - 1\) to do max

TOTAL runtime: \(O(n2^n)\)
a faster way?

Note worst case: every request conflicts with some other request.

In many optimization problems, the exponential approach is the straightforward way, but we want to try to find polynomial approaches.

Let's try divide and conquer.

— How to divide the problem into subproblems??

Given list of requests REQUESTS,

REQUESTS[ij] = set of all requests that can fit between request i and request j,

\[ \{ \text{request } k \mid \text{finish}_i < \text{start}_k < \text{finish}_k < \text{start}_j \} \].
faster? activity scheduling algorithm.

1) sort all requests by finish time

2) Add fake requests
   \[
   \text{requests} \cup \text{request}_{n+1} \}
   \]
   boundary markers

3) The largest non-conflicting
   subset of \text{REQUESTS}_p, n+1
   is our answer.

How to do step #3?
- divide and conquer!
Finding The Largest Non-Conflicting Sublist of REQUESTS ij

① If $i \geq j$ (we have sorted requests by finish time), then REQUESTS $ij$ is empty.

② If REQUESTS $ij$ is empty for any reason, then there is no conflicts to remove.

③ If REQUESTS $ij$ is not empty, then that means there must exist some request $k$ where $i < k < j$ that will appear in the optimal solution.

If only one request $k$, that request will appear in optimal solution.
If many requests fit between request $i$ and request $j$, we have to find the largest conflicting subset.

How to find it?

Divide and conquer!

Let request $k$ = a request in the optimal solution between request $i$ and request $j$.

$$\text{optimal} \left| \text{REQUESTS}_{i,j} \right| = \text{optimal} \left| \text{REQUESTS}_{i,k} \right| + 1 + \text{optimal} \left| \text{REQUESTS}_{k,j} \right|$$
Let $\text{OPTIMAL-COUNT}[i, j]$

$$= \text{number of requests in the optimal REQUESTS}_{i,j}$$

$(\text{max # of events possible between request } i \text{ and request } j)$

**algorithm:**

$$\text{OPTIMAL-COUNT}[i, j] = \max \left( \text{OPTIMAL-COUNT}[i, k] + 1 + \text{OPTIMAL-COUNT}[k, j] \right).$$

Or $\emptyset$ if no requests.

(Compute sum for all possible values of $k$ and find max)

**algorithm:**

$$\text{compute OPTIMAL-COUNT}[\emptyset, n+1]$$
Dynamic Programming

Once you compute a value in \( \text{OPTIMAL-COUNT}(i, j) \), store it and reuse if you need it later.

To get max reuse,

- Start with small subproblems and use answers to solve bigger subproblems ("opposite" of recursion)
- Start at bottom of recursion tree.

Like one of our Fibonacci solutions... memoization
OPTIMAL-COUNT MATRIX

or "C" for short

\[
C_{i,j} \text{ where } i \geq j = \emptyset
\]

\[
C_{i,j} \text{ where } j \geq i+3
= \max \left[ C_{i,k} + 1 + C_{k,j} \right]
\]

you can fill in the table using previously computed values in the table.
OPTIMAL-COUNT[i,j]. or C[i,j] for short

1. If \( i = j \), then \( C[i,j] = \emptyset \).
2. If \( i > j \), then \( C[i,j] \) is invalid.
3. If \( j = i + 1 \) request \( i \) request \( i + 1 \)
   \( C[i,j] = \emptyset \)
4. If \( j = i + 2 \) for (i = \( \emptyset \) to \( n - 1 \))
   compute \( C[i, i+2] \)
5. If \( j = i + 3 \) for (i = \( \emptyset \) to \( n - 2 \))
   \( j = i + 3 \)
   compute \( C[i,j] \)

and so on...
Pseudo code (use on homework 7)

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];