Recap

We have studied the divide and conquer technique and Fri you did a worksheet about these kinds of problems.

What are some of the characteristics of this technique?
Recap

We have studied the divide and conquer technique and Fri you did a worksheet about these kinds of problems.

What are some of the characteristics of this technique?

We worked through a rank-analysis problem using similarity to demonstrate this technique, modifying a merge-sort to count inversion pairs.
What about the Gale-Shapley algorithm? Could it be solved using divide and conquer?
Dynamic Programming
CS 320, Fall 2017

Dr. Geri Georg, Instructor
georg@colostate.edu
DP Strategy

1. Characterize **structure:**
   - show how optimum to larger problem can be constructed from optima of "small set" (polynomial) of smaller problems

2. **Recursively** define the optimum

3. Compute the optimum **bottom up:**
   - store solutions of sub problems

4. Construct optimum from **stored data**
Weighted Intervals

Find the set of compatible intervals with the maximum sum of values (weights)
Sort by Finish Time

Latest Predecessor:
\[ p(j) = \text{largest index } i < j \mid \text{job } i \text{ is compatible with job } j, \]
\[ p(j) = 0 \text{ no predecessors} \]
Recurrence

OPT(j): intervals that contribute to the optimum when we’re considering interval j

**Case 1:** OPT(j) includes j
- add $v_j$ to total value
- include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $p(j)$

**Case 2:** OPT(j) does not include j
- include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., $j-1$

$$OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + OPT(p(j)), OPT(j-1) \} & \text{otherwise}
\end{cases}$$
Recursive Solution

**input:** $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $v_1, \ldots, v_n$

**sort** jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

**compute** $p(1), p(2), \ldots, p(n)$

```
Compute-Opt(j) {
    if (j == 0)
        return 0
    else
        return max($v_j + Compute-Opt(p(j))$, Compute-Opt(j-1))
}
```

What $j$ do we start with?
Worst Case
What if we “layer” the intervals?

Where do we start? 5, and we have to call Compute-Opt for both 4 (in case we don’t use 5), and 3 (in case we do), ditto with 4, ....
Worse Case Recursive

Note this is top-down

Exponential growth
Memoization – also top-down

input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

compute \( p(1), p(2), \ldots, p(n) \)

for \( j = 1 \) to \( n \)

\[ M[j] = \text{empty} \]

\[ M[0] = 0 \]

\[ M-\text{Compute-Opt}(j) \{ \]

if \( (M[j] \text{ is empty}) \)

\[ M[j] = \max(v_j + M-\text{Compute-Opt}(p(j)), M-\text{Compute-Opt}(j-1)) \]

return \( M[j] \} \]
What about intervals??

The array M[] only stored the value, not the nodes we used to get it. We can recursively use it to figure out the intervals used to get the maximum.

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
Bottom-up dynamic programming

Unwind the recursion:

**input**: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n$

sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

**compute** $p(1), p(2), \ldots, p(n)$

Iterative-Compute-Opt {
    \[ M[0] = 0 \]
    \[ \text{for } j = 1 \text{ to } n \]
    \[ M[j] = \max(v_j + M[p(j)], M[j-1]) \]
Subset Sums

Here’s another example:

Given a set of \( n \) objects with weights \( w_i \) and a capacity \( W \), find a subset \( S \) with the largest sum of weights such that total weight is less equal \( W \)

Recursive approach

Either take object \( i \) or don't

Assume the current available capacity is \( W \)

If we take object \( i \), leftover capacity is \( W - w_i \)

If we don't, leftover capacity is \( W \)
Recurrence

Assume: \( \text{OPT}(i, w) \)

\( \text{OPT} \) is the weight of the max weight subset with the set of objects 1..\( i \)

Case 1: Don’t include \( i \)
  subset includes best of \{ 1, 2, ..., \( i-1 \) \} with \( w \)

Case 2: Do include \( i \)
  new weight limit is \( w - w_i \)
  subset include best of \{ 1, 2, .., \( i-1 \) \} with \( w-w_i \)
Recurrence

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i - 1, w) & \text{if } w_i > w \\
\max\{OPT(i - 1, w), w_i + OPT(i - 1, w - w_i)\} & \text{otherwise}
\end{cases}
\]

OPT is the weight of the max weight subset with the set of objects 1..i
Case 1: Don’t include i
    subset includes best of \(\{1, 2, ..., i - 1\}\) with \(w\)
Case 2: Do include i
    new weight limit is \(w - w_i\)
    subset include best of \(\{1, 2, .., i - 1\}\) with \(w - w_i\)
Bottom-up Dynamic Prog

1. Unwind the recursion so we can have a loop that goes from 1 to $n$.
2. Save the **weights** in a table of $(i, w)$ where $i$ is the object set and $w$ is the capacity. Initialize weights to 0. Row $(i)$ label shows what’s in the set.

**We can start at obj$_1$ since the recurrence only needs the previous row for calculations**
Bottom-up Dynamic Prog

Capacity: 5
Object: 1 2 3
Weight: 2 3 4

What’s the weight if o_1 is in the set and the capacity is 1? In terms of the recurrence – what entries are needed to calculate M[i,w]?

\[
i=0: \quad M[i,w]=0 \\
w_i>w: \quad M[i,w]=M[i-1,w] \\
\text{else: } \quad M[i,w]=\max(M[i-1,w], w_i + M[i-1,w-w_i])
\]

<table>
<thead>
<tr>
<th>n=3, W=5</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>i ↓ c→</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Objects are in the Set?

Start at $i = 3$ items, $c = 5$ wt:

- if $M[i,c] - M[(i-1),(c-w_3)] = w_3$ then $i$ included
- and $i = i-1$ and $c = c - w_i$
- else $i = i-1$

<table>
<thead>
<tr>
<th>n=3, W=5</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i \downarrow$</td>
<td>c→</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{1}</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Object: 1 2 3
Weight: 2 3 4
Traveling

You have 8 gallons of gas, and need to travel as far as you can.

You can change speed every hour if you want, but when your 8 gallons is gone, you’re done.

<table>
<thead>
<tr>
<th>mph→</th>
<th>8</th>
<th>50</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>170</th>
<th>170</th>
<th>196</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg→</td>
<td>8</td>
<td>25</td>
<td>27</td>
<td>22.5</td>
<td>20</td>
<td>28</td>
<td>24</td>
<td>24.5</td>
</tr>
</tbody>
</table>
**First** question – what are we trying to “optimize”?
Maximize the distance we go.

**Second** question – what data are we given?
Starting number of gallons, miles/hour, and miles/gallon

**Third** question – what data do we need?
How many gallons we use in an hour, at each speed
We essentially have the data in “miles” and need it in “gallons”

<table>
<thead>
<tr>
<th>mph</th>
<th>8</th>
<th>50</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>170</th>
<th>170</th>
<th>196</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>8</td>
<td>25</td>
<td>27</td>
<td>22.5</td>
<td>20</td>
<td>28</td>
<td>24</td>
<td>24.5</td>
</tr>
</tbody>
</table>

We can change speed every hour, so use mph to get how far in 1 hr and then divide by mpg to get number of gallons used. (Round to closest integer.)

<table>
<thead>
<tr>
<th>gals</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>miles</td>
<td>8</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>100</td>
<td>170</td>
<td>170</td>
<td>196</td>
</tr>
</tbody>
</table>
What problem can we think about that can be divided into smaller instances of the same problem?
How about choosing the number of gallons to use?
We could decide how many gallons to use, then decide how many gallons to use of what is left, ....
We can do this till there is 1 gallon left, and that’s the smallest we can choose so we’re done.
choose $x$

... $x$ ... $z$

... 1 gal ... and we go 8 miles

10 gal 10 - $x$ gal
Image Credits