Lecture 7:
• Dynamic Programming: Fibonacci Numbers, Interval Scheduling

Jan 30, 2018
Dynamic Programming

• Don’t think too hard about the name
  • “I thought dynamic programming was a good name. It was something not even a congressman could object to. So I used it as an umbrella for my activities.”
    –Richard Bellman

• Dynamic programming is careful recursion
  • Identify a “small” number of “subproblems”
  • Relate these problems via a recurrence
  • Carefully solve the recurrence
Warmup: Fibonacci Numbers
Fibonacci Numbers

The recurrence is given "for free"

\[ F(0) = 0, \ F(1) = 1 \]

\[ F(n) = F(n - 1) + F(n - 2) \]

\[ F(n) \approx \phi^n \approx 1.62^n \]

\[ \phi = \left( \frac{1 + \sqrt{5}}{2} \right) = \text{the "golden ratio"} \]

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
Fibonacci Numbers Alg I

FibI(n):
  If \( n = 0,1 \): return \( n \)  // Base Cases
  Else: return FibI(n − 1) + FibI(n − 2)  // Recursion

• How many total recursive calls does this algorithm make when computing \( F(n) \)?

  If \( R(n) \) is the # of recursive calls to FibI on input \( n \)
  \[ R(n) = R(n-1) + R(n-2) \]

  \[ R(n) = F(n) = 1.62^n \] (over a million calls to compute \( F(30) \))
Fibonacci Numbers Alg I

FibI(n):
   If $n = 0,1$: return $n$
   Else: return FibI(n - 1) + FibI(n - 2)
Fibonacci Numbers Alg II ("Memoization")

Let $M$ be an array (initialize each cell to “empty”) 

FibII($n$):

If $n = 0, 1$: return $n$  // Base case 

Else if $M[n]$ is not empty: return $M[n]$  // Checking for a solution 

Else: 

\[ M[n] \leftarrow \text{FibII}(n - 1) + \text{FibII}(n - 2) \]

return $M[n]$ 

• How many total recursive calls does this algorithm make when computing $F(n)$?

  M only has $n-1$ elements at any point in the algorithm 

  Every time we make two recursive calls, we fill one element of $M$. 

  $R(n) \leq 2^{(n-1)}$ WAY less than $1.62^n$ recursive calls
Fibonacci Numbers Alg III ("Bottom-Up / Iterative")

\[ \text{FibIII}(n): \]
\[ M[0] \leftarrow 0, M[1] \leftarrow 1, \]
For \( i = 3, \ldots, n: \)
\[ M[i] \leftarrow M[i - 1] + M[i - 2] \]
return \( M[n] \)

- How many additions does this algorithm take to compute \( F(n) \)?

\[ O(n) \text{ additions} \times O(n) \text{ time per addition} = O(n^2) \text{ time} \]
Fibonacci Numbers Alg III

\[
\text{FibIII}(n):
\]
\[
M[0] \leftarrow 0, M[1] \leftarrow 1,
\]
For \( i = 3, \ldots, n: \)
\[
M[i] \leftarrow M[i - 1] + M[i - 2]
\]
return \( M[n] \)
Summary

• The “obvious” recursive algorithm for Fibonacci numbers takes exponential time

• Problem: recursing on the same problem many times
  • Idea I: Remember solutions (aka “top-down”)
  • Idea II: Solve subproblems in order (aka “bottom-up”)

• Dynamic programming is careful recursion
  • Identify a “small” number of “subproblems”
  • Relate these problems via a recurrence
  • Carefully solve the recurrence
Interval Scheduling
Interval Scheduling

• How can we optimally schedule a resource?
  • This classroom, a computing cluster, the Large Hadron Collider, etc...

• Input: a \( n \) intervals \((s_i, f_i)\) with value \(v_i\)
• Output: a compatible schedule \(S\) with the largest possible total value
  • A schedule is a subset of intervals \(S \subseteq \{1, \ldots, n\}\)
  • A schedule \(S\) is compatible if no two \(i, j \in S\) overlap
  • The total value of \(S\) is \(\sum_{i \in S} v_i\)
Interval Scheduling

<table>
<thead>
<tr>
<th>Index</th>
<th>$s_i$</th>
<th>$v_i$</th>
<th>$f_i$</th>
<th>$v_i$</th>
<th>$f_i$</th>
<th>compatible?</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>No</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Yes</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$s_1, 9, 5, 63$
$s_1, 63$
$s_1, 3, 5$

Value

Time
Possible Algorithms

• Choose most valuable interval first

- take most valuable
- eliminate conflicts

\[ 1 \quad 2 \quad 1 \quad 2 \quad 1 \]

this rule gives value 3
opt value is 4
Possible Algorithms

• Choose interval with earliest start time first
Possible Algorithms

• Choose interval with earliest finish time

1

[1, 10]
Possible Algorithms

• Choose shortest interval first

The optimal set of intervals requires some "global reasoning"
A Recursive Algorithm

• Let $O$ be the **optimal** solution

• Case I: Last interval is not in $O$ ($6 \notin O$)
  • Then $O$ must be the optimal solution for $\{1, \ldots, 5\}$

Why? If there were a better solution $O'$ containing only intervals in $\{1, \ldots, 5\}$ then $O'$ is better than $O$.

Step 1: Post an optimal solution and think about it.

Step 2: Classify $O$ $O$ either does or does not contain 6.

Index

\begin{align*}
1 & \quad v_1 = 2 \\
2 & \quad v_2 = 4 \\
3 & \quad v_3 = 4 \\
4 & \quad v_4 = 7 \\
5 & \quad v_5 = 2 \\
6 & \quad \text{(crossed out)}
\end{align*}

Opt solution for $\{1, \ldots, 6\}$ is an opt solution for a smaller problem
A Recursive Algorithm

Let $O$ be the **optimal** solution

Case I: Last interval is not in $O$ ($6 \notin O$)
- Then $O$ must be the optimal solution for $\{1, \ldots, 5\}$

Case 2: Last interval is in $O$ ($6 \in O$)
- Then $O$ must be the optimal solution for $\{1, \ldots, 3\} + \{6\}$

Suppose $O = \{2, 6\}$ on intervals $1, 2, 3$,
$O' = \{1, 3\}$ beats $\{2\}$.
A Recursive Algorithm

Assume that the intervals are sorted by $f_j$, so $f_1 < f_2 < \ldots < f_n$.

- Let $O$ be the optimal solution

- **Case I:** Last interval is not in $O$ ($n \notin O$)
  
  - Then $O$ must be the optimal solution for \{1, \ldots, n − 1\}

- **Case 2:** Last interval is in $O$ ($n \in O$)
  
  - Sort intervals by end time so $f_1 < f_2 < \ldots < f_n$
    
  - Let $p(i)$ be the last interval not overlapping $i$
    
  - Then $O$ must be the optimal solution for \{1, \ldots, p(n)\} union of \{n\}

Let $OPT(i)$ be the value of the optimal schedule using \{1, \ldots, i\}

\[ OPT(0) = 0 \]

\[ OPT(i) = \max \{ OPT(i-1), v_i + OPT(p(i)) \} \]
Recursive Algorithm I

FindOPT(i):
If $i = 0$: return 0
Else: return $\max\{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i-1)\}$

- Claim: FindOPT(i) computes $OPT(i)$  $\forall i=0\ldots,n$
  
  **Proof:** Base case ($i=0$): easy

  Inductive Step: if $\text{FindOPT}(j) = \text{OPT}(j)$ $\forall j<i$
  
  then $\text{FindOPT}(j) = \max\{v_i + \text{FindOPT}(p(j)), \text{FindOPT}(j-1)\}$

  $\quad = \max\{v_i + \text{OPT}(p(j)), \text{OPT}(j-1)\}$ (IH)

  $\quad = \text{OPT}(j)$
Recursive Algorithm I

FindOPT(i):
   If $i = 0$: return 0
   Else: return $\max\{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i - 1)\}$

- Sad Fact: $\text{FindOPT}(n)$ might run in $\geq F(n)$ time

$F(n) = n^2$ (Sad Fact)

Makes at least $1.62^n$ recursive calls.
Recursive Algorithm II: Memoization

M is an array to hold subproblems we’ve solved. (Initially empty)

MFindOPT(i):

If \( i = 0 \): return 0
Elseif \( M[i] \) is not empty: return \( M[i] \)
Else:

\[
M[i] \leftarrow \max \{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i - 1)\}
\]

return \( M[i] \)

• Happy fact: MFindOPT(\( n \)) runs in time \( O(n) \)
  - Only \( n \) elements of \( M \) get filled in
  - Every two recursive calls fill one elem

\[ \Longrightarrow \text{At most 2n recursive calls, } T(n) = O(n) \]
Finding the Solution
Finding the Solution

- \( v_6 + \text{OPT}(3) = 7 \)
- \( \text{OPT}(5) = 8 \)
- We know 6 \( \notin O \)!
Finding the Solution

FindSOL(i):
  If $i = 0$: return $\emptyset$
  Else:
    If $v_i + M[p(i)] \geq M[i - 1]$:
      return: $\{i\} \cup \text{FindSOL}(p(i))$
    Else
      return: FindSOL($i - 1$)
Recursive Algorithm III: Bottom-Up

IterFindOPT(i):

\[ M[0] \leftarrow 0 \]

For \( i = 1, \ldots, n \):

\[ M[i] \leftarrow \max\{v_i + \text{FindOPT}(p(i)), \text{FindOPT}(i - 1)\} \]
Recursive Algorithm III: Bottom-Up

Index

\[
\begin{align*}
1 & \quad v_1 = 2 \quad p(1) = 0 \\
2 & \quad v_2 = 4 \quad p(2) = 0 \\
3 & \quad v_3 = 4 \quad p(3) = 1 \\
4 & \quad v_4 = 7 \quad p(4) = 0 \\
5 & \quad v_5 = 2 \quad p(5) = 3 \\
6 & \quad v_6 = 1 \quad p(6) = 3
\end{align*}
\]

0 2 4 6 7 8 8

\[\text{OPT(6) OPT(1) \ldots OPT(6)}\]
Now You Try

\[ v_1 = 3 \quad \text{and} \quad p(1) = 0 \]
\[ v_2 = 5 \quad \text{and} \quad p(2) = 1 \]
\[ v_3 = 9 \quad \text{and} \quad p(3) = 0 \]
\[ v_4 = 6 \quad \text{and} \quad p(4) = 2 \]
\[ v_5 = 13 \quad \text{and} \quad p(5) = 1 \]
\[ v_6 = 3 \quad \text{and} \quad p(6) = 4 \]
Dynamic Programming Recap

• What did we do:
  • Identified a “small” number of “subproblems”
  • Related these problems via a recurrence
  • Carefully solved the recurrence

• Defining the subproblems and finding a recurrence can be challenging