CS3000: Algorithms & Data
Jonathan Ullman

Lecture 6:
• Dynamic Programming:
  Fibonacci Numbers, Interval Scheduling

Sep 25, 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. -Bellman

• Dynamic programming is careful recursion
  • Break the problem up into small pieces
  • Recursively solve the smaller pieces
  • **Key Challenge:** identifying the pieces
Warmup: Fibonacci Numbers
Fibonacci Numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- $F(n) = F(n - 1) + F(n - 2)$
- $F(n) \to \phi^n \approx 1.62^n$
- $\phi = \left(\frac{1+\sqrt{5}}{2}\right)$ is the golden ratio
Fibonacci Numbers: Take I

FibI(n):
    If (n = 0): return 0
    ElseIf (n = 1): return 1
    Else: return FibI(n-1) + FibI(n-2)

• How many recursive calls does \textbf{FibI(n)} make?
  \[ T(n) = 2^n \]
  \[ T(n) = T(n-1) + T(n-2) \]
  \[ T(n) \approx \phi^n \approx 1.62^n \]
Fibonacci Numbers: Take II

"Memoization" "Top-Down"

M ← empty array, M[0] ← 0, M[1] ← 1
FibII(n):
  If (M[n] is not empty): return M[n]
  ElseIf (M[n] is empty):
    M[n] ← FibII(n-1) + FibII(n-2)
  return M[n]

• How many recursive calls does \textbf{FibII}(n) make?
  • Only have to fill \( n-1 \) entries
  • Each pair of recursive calls fills one entry

\[ \Rightarrow \quad 2n-2 \text{ recursive calls} \quad O(n) \]
Fibonacci Numbers: Take III

```
FibIII(n):
    M[0] ← 0, M[1] ← 1
    For i = 2, ..., n:
        M[i] ← M[i-1] + M[i-2]
    return M[n]
```

- What is the running time of $\text{FibIII}(n)$?

$$0(n^2) \text{ time algorithm (b/c } \text{Fib}(n) \text{ has } \Omega(n) \text{ digits)}$$

$$\text{Fib}(n) \approx \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^n$$
Fibonacci Numbers

- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$
- $F(n) = F(n - 1) + F(n - 2)$

- Solving the recurrence recursively takes $\approx 1.62^n$ time
  - Problem: Recompute the same values $F(i)$ many times

- Two ways to improve the running time
  - Remember values you’ve already computed (“top down”)
  - Iterate over all values $F(i)$ (“bottom up”)

- **Fact:** Can solve even faster using Karatsuba’s algorithm!
Dynamic Programming:
Interval Scheduling
Interval Scheduling

• How can we optimally schedule a resource?
  • This classroom, a computing cluster, ...

• **Input:** $n$ intervals $(s_i, f_i)$ each with value $v_i$
  • Assume intervals are sorted so $f_1 < f_2 < \cdots < f_n$

• **Output:** a compatible schedule $S$ maximizing the total value of all intervals
  • A schedule is a subset of intervals $S \subseteq \{1, \ldots, n\}$
  • A schedule $S$ is compatible if no $i, j \in S$ overlap
  • The total value of $S$ is $\sum_{i \in S} v_i$
Interval Scheduling

\[ S = \{1, 5\} \]

value\( (S) = v_1 + v_5 = 4 \)

Index

1 \( v_1 = 2 \)
2 \( v_2 = 4 \)
3 \( v_3 = 4 \)
4 \( v_4 = 7 \)
5 \( v_5 = 2 \)
6 \( v_6 = 1 \)
A Recursive Formulation

• Let $O$ be the **optimal** schedule

• **Case 1:** Final interval is not in $O$ (i.e. $6 \notin O$)
  • Then $O$ must be the optimal solution for $\{1, \ldots, 5\}$

If $O$ were not the optimal of $\{1, \ldots, 5\}$ and $6 \notin O$,
then the opt of $\{1, \ldots, 5\}$ is better than $O$.

Index

1
$v_1 = 2$

2
$v_2 = 4$

3
$v_3 = 4$

4
$v_4 = 7$

5
$v_5 = 2$

6
6 $\notin O$

$O$ is the opt of these intervals
A Recursive Formulation

• Let $O$ be the **optimal** schedule

• **Case 2:** Final interval is in $O$ (i.e. $6 \in O$)
  • Then $O$ must be $6 +$ the optimal solution for $\{1, \ldots, 3\}$

  $$O \text{ is either } 63 + \text{opt} (31, 2, 53)$$

  $$\text{opt} (31, \ldots, 53)$$

Index

1. $v_1 = 2$
2. $v_2 = 4$
3. $v_3 = 4$
4. $v_4 = 7$
5. $v_5 = 2$
6. $v_6 = 1$

\[ \text{optimal schedule for } 31, \ldots, 53 \]

\[ \text{not in } O \]
A Recursive Formulation

On is the thing we want

• Let $O_i$ be the optimal schedule using only the intervals \{1, ..., i\}

• **Case 1:** Final interval is not in $O_i$ ($i \notin O_i$) \[ O_i = O_{i-1} \]
  • Then $O_i$ must be the optimal solution for \{1, ..., i − 1\}

• **Case 2:** Final interval is in $O$ ($i \in O_i$) \[ O_i = s_i; \exists + O_{p(i)} \]
  • Assume intervals are sorted so that $f_1 < f_2 < \cdots < f_n$
  • Let $p(i)$ be the largest $j$ such that $f_j < s_i$
  • Then $O_i$ must be $\exists i^2+$ the optimal solution for \{1, ..., $p(i)$\}

If $\text{value}(O_{i-1}) > v_i + \text{value}(O_{p(i)})$ then $i$ is not in $O_i$
Else $i$ is in $O_i$
A Recursive Formulation

\[ \text{OPT}(i) = \text{value}(O_i) \]

- Let \( \text{OPT}(i) \) be the value of the optimal schedule using only the intervals \( \{1, \ldots, i\} \)

- **Case 1:** Final interval is not in \( O_i \) \( (i \notin O_i) \)
  - Then \( O \) must be the optimal solution for \( \{1, \ldots, i-1\} \)

- **Case 2:** Final interval is in \( O_i \) \( (i \in O_i) \)
  - \( \text{OPT}(i) = v_i + \text{OPT}(p(i)) \)
    - Assume intervals are sorted so that \( f_1 < f_2 < \cdots < f_n \)
    - Let \( p(i) \) be the largest \( j \) such that \( f_j < s_i \)
    - Then \( O \) must be in the optimal solution for \( \{1, \ldots, p(i)\} \)

- \( \text{OPT}(i) = \max\{\text{OPT}(i-1), v_i + \text{OPT}(p(i))\} \)

- \( \text{OPT}(0) = 0, \text{OPT}(1) = v_1 \)

Algorithmically the same as computing Fib numbers.
Interval Scheduling: Take I

Assuming values $p(n)$ are already computed

// All inputs are global vars
FindOPT(n):
    if (n = 0): return 0
    elseif (n = 1): return $v_1$
    else:
        return max{FindOPT(n-1), $v_n + \text{FindOPT}(p(n))$}

• What is the running time of $\text{FindOPT}(n)$?

At least 1.62$^n$ recursive calls
// All inputs are global vars
M ← empty array, M[0] ← 0, M[1] ← \$ \cup_\perp

FindOPT(n):
  if (M[n] is not empty): return M[n]
  else:
    M[n] ← \max \{FindOPT(n-1), v_n + FindOPT(p(n))\}
  return M[n]

• What is the running time of \textbf{FindOPT}(n)\
  \((n-1)\text{ entries to fill}) \times (2 \text{ calls per entry}) = 2n-2\)

\(O(n)\) time \(\left(\begin{array}{c}
+ O(n\log n)\text{ to sort if necessary} \\
+ O(n)\text{ to compute } p(1) \ldots p(n)
\end{array}\right)\)
Interval Scheduling: Take II

\[ M[i] = \text{OPT}(i) \]

\[ M[2] = \max \{ M[1], 4 + M[0] \} \]

\[ M[3] = \max \{ M[2], 4 + M[1] \} \]

\[ M[4] = \max \{ M[3], 7 + M[0] \} \]

\[ M[5] = \max \{ M[4], 2 + M[3] \} \]

\[ M[6] = \max \{ M[5], 1 + M[3] \} \]

\[ p(1) = 0 \]

\[ p(2) = 0 \]

\[ p(3) = 1 \]

\[ p(4) = 0 \]

\[ p(5) = 3 \]

\[ p(6) = 3 \]
Interval Scheduling: Take III

“Bottom-Up Dynamic Programming”

// All inputs are global vars
FindOPT(n):
M[0] ← 0, M[1] ← 2 v1
for (i = 2,...,n):
    M[i] ← \max \{M[i-1], v_i + M[p(i)]\}
return M[n]

• What is the running time of \textbf{FindOPT (n)}?

\(O(n) + \text{time to sort if needed} + \text{time to compute } p(i)'s\)
Finding the Optimal Solution

• Let \( OPT(i) \) be the **value of the optimal schedule** using only the intervals \( \{1, \ldots, i\} \)

• **Case 1:** Final interval is not in \( O \) \( (i \notin O) \)

• **Case 2:** Final interval is in \( O \) \( (i \in O) \)

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

If \( \) \( OPT(i-1) > v_i + OPT(p(i)) \):
\[
O_i = O_{i-1}, \quad (i \in O_i)
\]

Else if \( v_i + OPT(p(i)) > OPT(i-1) \):
\[
O_i = \xi3 + O_{p(i)}, \quad (i \in O_i)
\]

Else: \( O_i \) could be either \( \xi3 + O_{p(i)} \) or \( O_{i-1} \)
Interval Scheduling: Take II

Indices

<table>
<thead>
<tr>
<th>Index</th>
<th>$v_1 = 2$</th>
<th>$v_2 = 4$</th>
<th>$v_3 = 4$</th>
<th>$v_4 = 7$</th>
<th>$v_5 = 2$</th>
<th>$v_6 = 1$</th>
</tr>
</thead>
</table>

$M[i] = \text{OPT}(i)$

$O_6 = \{3, 1, 3, 5\}$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

$O_1 = \{3\}$

$3 \in O_0$

$5 \in O_0$

$6 \in O_0$
Interval Scheduling: Take III

Completed table with value of optimum

```c
// All inputs are global vars
FindSched(M,n):
    if (n = 0): return Ø
    elseif (n = 1): return {1}
    elseif (v_n + M[p(n)] > M[n-1]):
        return {n} + FindSched(M,p(n))
    else:
        return FindSched(M,n-1)
```

• What is the running time of \textbf{FindSched(n)}?
Now You Try

1. $v_1 = 3$
2. $v_2 = 5$
3. $v_3 = 9$
4. $v_4 = 6$
5. $v_5 = 13$
6. $v_6 = 3$

$p(1) = 0$
$p(2) = 1$
$p(3) = 0$
$p(4) = 2$
$p(5) = 1$
$p(6) = 4$
Dynamic Programming Recap

• Express the optimal solution as a recurrence
  • Identify a small number of subproblems
  • Relate the optimal solution on subproblems

• Efficiently solve for the value of the optimum
  • Simple implementation is exponential time
  • Top-Down: store solution to subproblems
  • Bottom-Up: iterate through subproblems in order

• Find the solution using the table of values