Dynamic Programming

Viet Tung Hoang

The slides are loosely based on those of Dr Kevin Wayne, Princeton University
Recall: Divide & Conquer

- Divide the problem into multiple **independent** subproblems
- Solve each subproblem and then combine them to form solution for original problem
Today: Dynamic Programming

- Break up a problem into a series of overlapping subproblems
- Build up solutions to larger and larger subproblems
Some Applications of Dynamic Programming

Speech recognition

Unix diff to compare files

Triangulation
Agenda

1. Fibonacci numbers

2. Weighted Activity Selection

3. Binary Knapsack
Fibonacci Numbers

Ubiquitous in nature

\[ F_1 = 1, \quad F_2 = 1 \]
\[ F_n = F_{n-1} + F_{n-2}, \quad \text{for} \quad n \geq 3 \]

Leaves per height in sneezewort

Nautilus size

Pineapple scales
Naïve Algorithm

**procedure** $\text{Fib}(n)$

*if* $n = 1$ or $n = 2$ *then* return 1

*else* return $\text{Fib}(n - 1) + \text{Fib}(n - 2)$

Running time escalate *exponentially*
**How Slow Is It?**

```plaintext
procedure Fib(n)
    if n = 1 or n = 2 then return 1
    else return Fib(n - 1) + Fib(n - 2)
```

\[
T(n) = T(n - 1) + T(n - 2) + O(1)
\]

\[
\Omega \left( \left( \frac{1 + \sqrt{5}}{2} \right)^n \right)
\]

So \( T(n) \) is at least as big as the \( n \)-th Fibonacci number \( F_n \)
Why Is It So Slow?

When we run $\text{Fib}(6)$:

Should have **reused** the prior value

**Observation:** $\text{Fib}(4)$ and $\text{Fib}(3)$ are **recomputed**
Approach 1: Top Down with Memoization

Cache result of each sub-problem; lookup as needed

procedure Fib(n, M)

if $M[n] \neq \text{NULL}$ then return $M[n]$

if $n = 1$ or $n = 2$ then $M[n] \leftarrow 1$

double else $M[n] \leftarrow \text{Fib}(n - 1, M) + \text{Fib}(n - 2, M)$

return $M[n]$
How Fast Is It?

When we run $\text{Fib}(6)$:

Only **two** nodes per level
Height $\leq n$  

$O(n)$ time
Approach 2: Bottom Up

Unwind recursion, and compute $F_i$ for every $i \leq n$

```
procedure Fib(n)
  if n = 1 or n = 2 then F[n] ← 1
  for i ← 3 to n do F[n] ← F[n - 1] + F[n - 2]
  return F[n]
```

$O(n)$ time
Graphical View of Execution

When we run $\text{Fib}(6)$:

Nodes of the same value now collapse to one
Agenda

1. Fibonacci numbers

2. Weighted Activity Selection

3. Binary Knapsack
**Weighted Activity Selection**

Job $i$ starts at time $s_i$, finishes at time $f_i$, and has value $v_i$.

**Goal:** find a set of non-overlapping jobs that maximizes the total value.
Finding Optimal Value

Label jobs by finishing time \( f_1 \leq f_2 \leq \cdots \leq f_n \)

\( U[i] \) : value of optimal solution if one considers just jobs 1, 2, .., \( i \)

\( U[n] \) : value of optimal solution of the original problem

Base case: \( U[0] \leftarrow 0 \)
Finding Optimal Value: Recursion

Use job $i$

Don’t use job $i$

$U[i] = \max\{U[i - 1], v_i + U[p_i]\}$
Using Dynamic Programming

\[ O(n \log(n)) \text{ time} \]

\[ O(n \log(n)) \text{ time} \quad \text{via binary search} \]

\[ O(n) \text{ time} \]

**procedure** \( \text{Interval}(n, s_1, f_1, v_1, \ldots, s_n, f_n, v_n) \)

Sort by finishing time so that \( f_1 \leq f_2 \leq \cdots \leq f_n \)

Compute \( p_1, \ldots, p_n \)

\[ U[0] \leftarrow 0; \]

\[ \text{for } i \leftarrow 1 \text{ to } n \text{ do } U[i] \leftarrow \max\{v_i + U[p_i], U[i - 1]\} \]

return \( U[n] \)

Totally \( O(n \log(n)) \) time
Finding Actual Solution: Backtracking

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>U[i]</td>
<td>0</td>
<td>1</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>p[i]</td>
<td>---</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Don’t use job $i$ if $U[i] = U[i - 1]$

value = 99

value = 1
Finding Actual Solution: Backtracking

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U[i]$</td>
<td>0</td>
<td>1</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>$p[i]$</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Don’t use job $i$ if $U[i] = U[i - 1]$

value = 1

value = 99
Finding Actual Solution: Backtracking

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U[i]$</td>
<td>0</td>
<td>1</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>$p[i]$</td>
<td>___</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Don’t use job $i$ if $U[i] = U[i - 1]$

value = 1

value = 99
Finding Actual Solution: Backtracking

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U[i]$</td>
<td>0</td>
<td>1</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>$p[i]$</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Don’t use job $i$ if $U[i] = U[i - 1]$

Optimal solution: {2}

value = 1

value = 99
Agenda

1. Fibonacci numbers

2. Weighted Activity Selection

3. Binary Knapsack
Binary Knapsack

- Have \( n \) items: item \( i \) has weight \( w_i \) and value \( v_i \)

- Have a knapsack with capacity \( C \)

Goal: Pack knapsack to maximize value without exceeding capacity

Example: Capacity = 11

<table>
<thead>
<tr>
<th>Item:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight:</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Value:</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

Best solution: Pick \{3, 4\} for value 40
Dynamic Programming: False Start

Let $A[i]$ be the optimal profit if we consider just items 1, ..., $i$ (with capacity $C$).

$A[n]$ : optimal profit of the original problem

Problem: One we pack item $i$, the remaining capacity is no longer $C$. 

Item we pack  |  Item we don’t pack  |  Item we may pack
Taking Capacity Into Consideration

Let $A[i, c]$ be the optimal profit if we consider just items 1, ..., $i$ with capacity $c$

$A[n, C]$ : optimal profit of the original problem

Base case: $A[0, c] \leftarrow 0$ and $A[i, 0] \leftarrow 0$

If $c < w_i$ (impossible to pack item $i$ if capacity is $c$):

$A[i, c] \leftarrow A[i - 1, c]$
If $c \geq w_i$ : $A[i, c] \leftarrow \max\{A[i - 1, c], v_i + A[i - 1, c - w_i]\}$
Dynamic Programming for Knapsack

```
procedure Knapsack\(n, C, v_1, w_1, \ldots, v_n, w_n\)
for \(i \leftarrow 0\) to \(n\) do \(A[i, 0] \leftarrow 0\)
for \(c \leftarrow 0\) to \(C\) do \(A[0, c] \leftarrow 0\)
for \(i \leftarrow 1\) to \(n\), \(c \leftarrow 1\) to \(C\) do
    if \(c < w_i\) then \(A[i, c] \leftarrow A[i - 1, c]\)
    else \(A[i, c] \leftarrow \max\{A[i - 1, c], v_i + A[i - 1, c - w_i]\}\)
return \(A[n, C]\)
```

Running time: \(O(nC)\)
# Backtracking to Find the Solution

Don’t take item $i$ if $A[i, c] = A[i - 1, c]$

**Optimal solution:** {3, 4}

<table>
<thead>
<tr>
<th></th>
<th>$i$</th>
<th>$v_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>{1}</td>
<td>0 1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>0 1 6 7 7 7 7 7 7 7 7 7</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>0 1 6 7 7 18 19 24 25 25 25 25</td>
</tr>
<tr>
<td>{1, 2, 3, 4}</td>
<td>0 1 6 7 7 18 22 24 28 29 29 40</td>
</tr>
<tr>
<td>{1, 2, 3, 4, 5}</td>
<td>0 1 6 7 7 18 22 28 29 34 35 40</td>
</tr>
</tbody>
</table>

![Table with values and arrows indicating the backtracking process to find the solution]