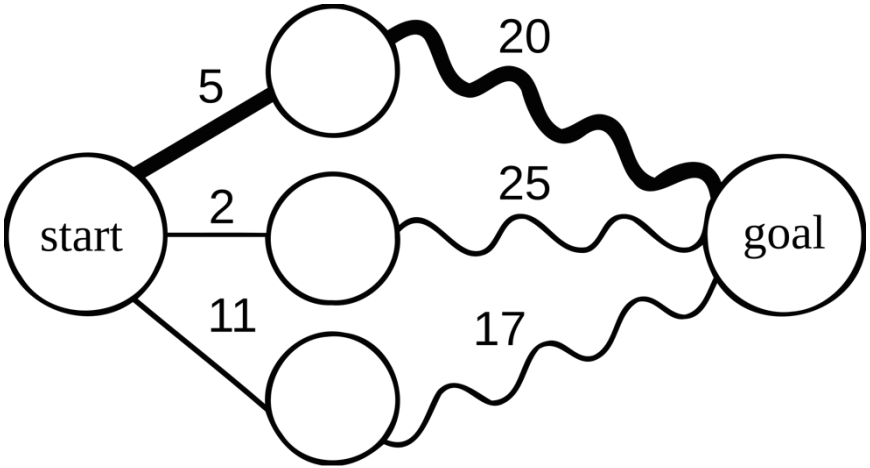
Lecture 16

Dynamic Programming I



Announcements

- Homework 5 Reflection due this Sunday night (4/12)
- Homework 6 due this Sunday night (4/12)
- Group Meetings continue

Algorithmic Paradigms

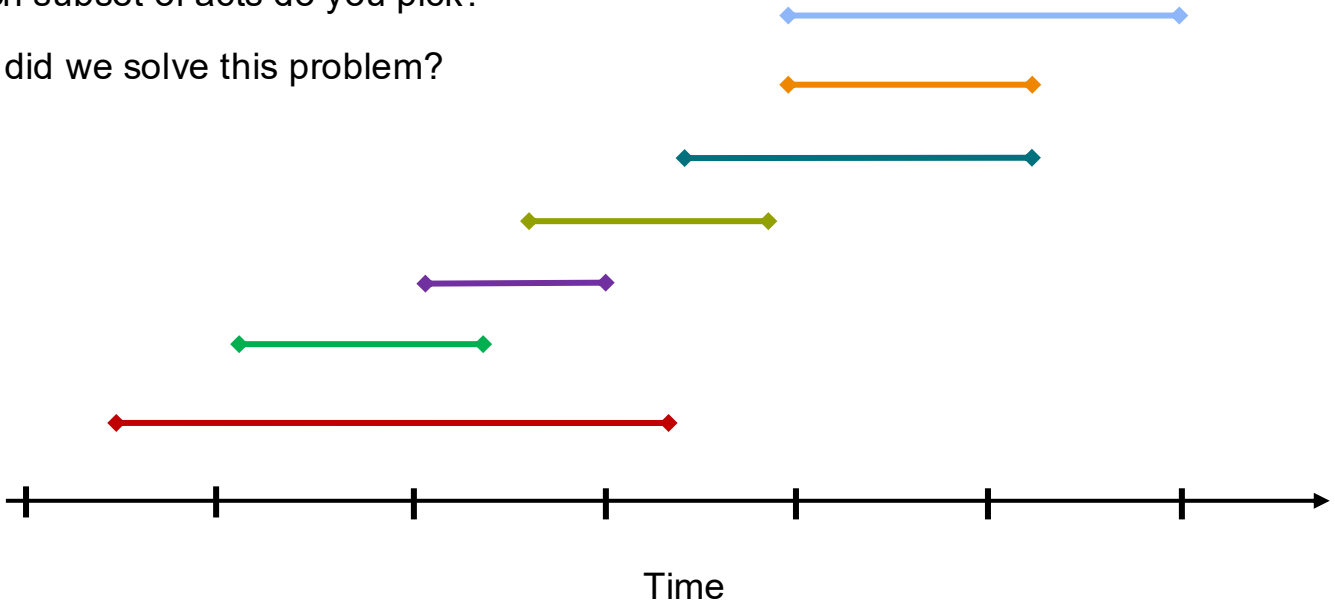
We are moving on to our study of algorithm design

- Greedy
- Divide-and-conquer
- **Dynamic Programming**
- Network Flow

Interval Scheduling

Imagine you are going to a music festival and want to see as many different acts as possible

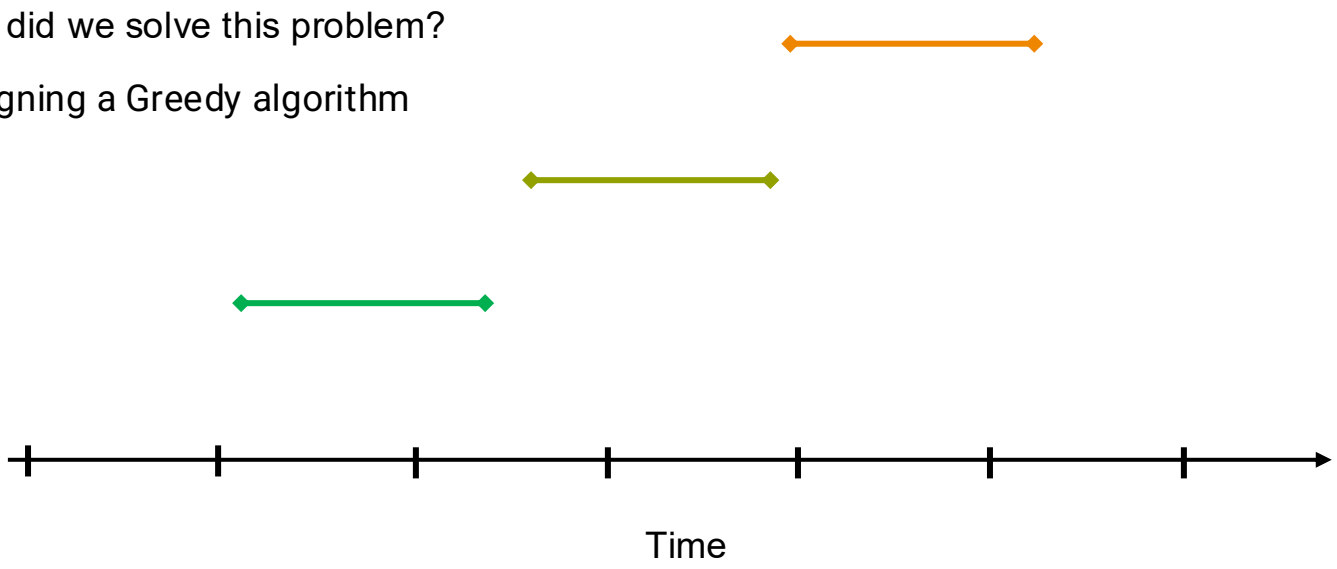
- Which subset of acts do you pick?
- How did we solve this problem?



Interval Scheduling

Imagine you are going to a music festival and want to see as many different acts as possible

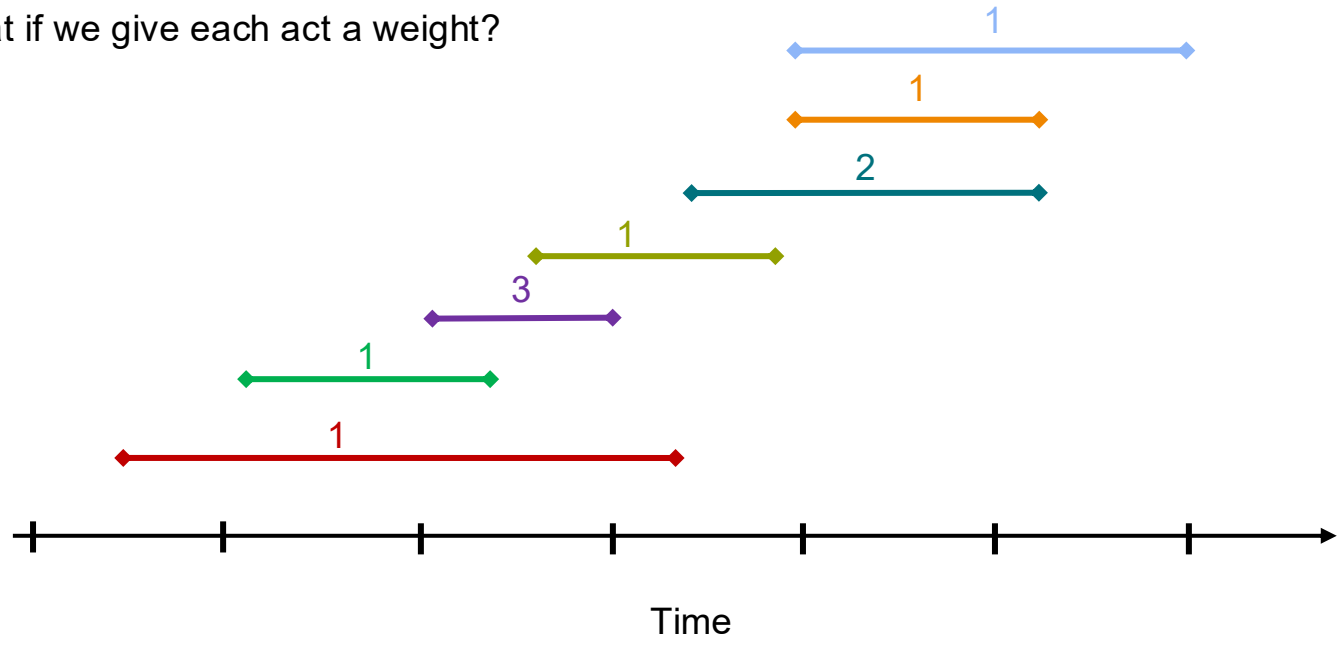
- Which subset of acts do you pick?
- How did we solve this problem?
- Designing a Greedy algorithm



Interval Scheduling

Imagine you are going to a music festival and want to see as many different acts as possible

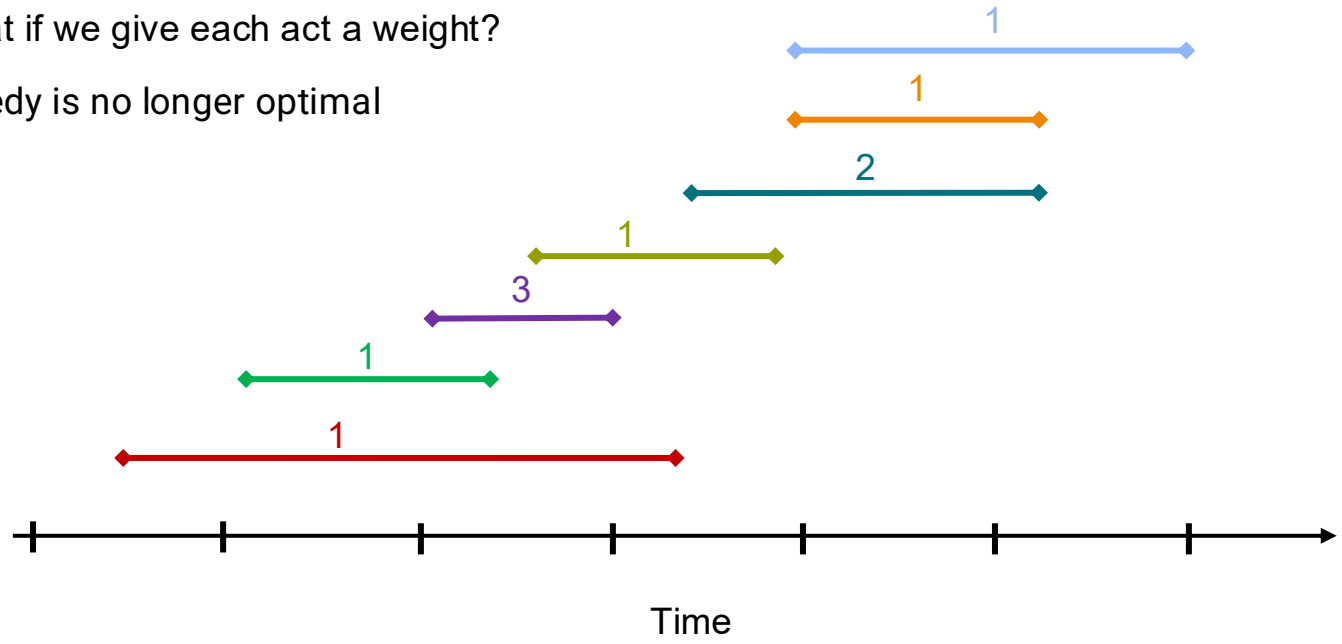
- What if we give each act a weight?



Interval Scheduling

Imagine you are going to a music festival and want to see as many different acts as possible

- What if we give each act a weight?
- Greedy is no longer optimal



Problem Formulation

- Act j has value v_j, start time s_j, finish time f_j
- Assume acts are sorted by finish time $f_1 \leq f_2 \leq \cdots \leq f_n$
- Acts i, j are **compatible** if they do not overlap
- **Goal**: find subset of compatible jobs with maximum total value

Dynamic Programming Recipe

- Step 1: Devise simple recursive algorithm for the value of the optimal solution
- Step 2: Write recurrence relation for optimal value
- Step 3: Design bottom-up iterative algorithm
- Step 4: Recover optimal solution

Step 1: Recursive Algorithm

Idea: Let A be an optimal solution. Either act $n \in A$ or $n \notin A$. In either case, we can reduce the problem to a *smaller* instance of the same problem.

Step 1: Recursive Algorithm

Idea: Let A be an optimal solution. Either act $n \in A$ or $n \notin A$. In either case, we can reduce the problem to a *smaller* instance of the same problem.

Let's write a recursive algorithm to compute value of optimal subset of first j acts

```
ComputeValue(j):
    if j = 0 do                    Base case
        return 0
```

Step 1: Recursive Algorithm

Idea: Let A be an optimal solution. Either act $n \in A$ or $n \notin A$. In either case, we can reduce the problem to a *smaller* instance of the same problem.

Let's write a recursive algorithm to compute value of optimal subset of first j acts

$\text{ComputeValue}(j)$:

if $j = 0$ **do** Base case

return 0

Let $i < j$ be the highest-numbered act compatible with j Case 1: $j \in A$

$val1 \leftarrow v_j + \text{ComputeValue}(i)$

Step 1: Recursive Algorithm

Idea: Let A be an optimal solution. Either act $n \in A$ or $n \notin A$. In either case, we can reduce the problem to a *smaller* instance of the same problem.

Let's write a recursive algorithm to compute value of optimal subset of first j acts

$\text{ComputeValue}(j)$:

if $j = 0$ **do** Base case

return 0

Let $i < j$ be the highest-numbered act compatible with j Case 1: $j \in A$

$val1 \leftarrow v_j + \text{ComputeValue}(i)$

$val2 \leftarrow \text{ComputeValue}(j-1)$ Case 2: $j \notin A$

Step 1: Recursive Algorithm

Idea: Let A be an optimal solution. Either act $n \in A$ or $n \notin A$. In either case, we can reduce the problem to a *smaller* instance of the same problem.

Let's write a recursive algorithm to compute value of optimal subset of first j acts

$\text{ComputeValue}(j)$:

if $j = 0$ **do** Base case

return 0

Let $i < j$ be the highest-numbered act compatible with j Case 1: $j \in A$

$val1 \leftarrow v_j + \text{ComputeValue}(i)$

$val2 \leftarrow \text{ComputeValue}(j-1)$ Case 2: $j \notin A$

return $\max(val1, val2)$

Exercise 1

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

a. $O(n \log n)$

b. $O(n^2)$

c. $O(n)$

d. $O(2^n)$

$\text{ComputeValue}(j)$:

if $j = 0$ **do**

return 0

Let $i < j$ be the highest numbered act compatible with j

$val1 \leftarrow v_j + \text{ComputeValue}(i)$

$val2 \leftarrow \text{ComputeValue}(j - 1)$

return $\max(val1, val2)$

Exercise 1

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

a. $O(n \log n)$

b. $O(n^2)$

c. $O(n)$

d. $O(2^n)$

$\text{ComputeValue}(j)$:

if $j = 0$ **do**

return 0

Let $i < j$ be the highest numbered act compatible with j

$val1 \leftarrow v_j + \text{ComputeValue}(i)$

$val2 \leftarrow \text{ComputeValue}(j - 1)$

return $\max(val1, val2)$

Recursion Tree

Let's consider the case of $\text{ComputeValue}(n)$ for our example

$\text{ComputeValue}(j)$:

if $j = 0$ **do**

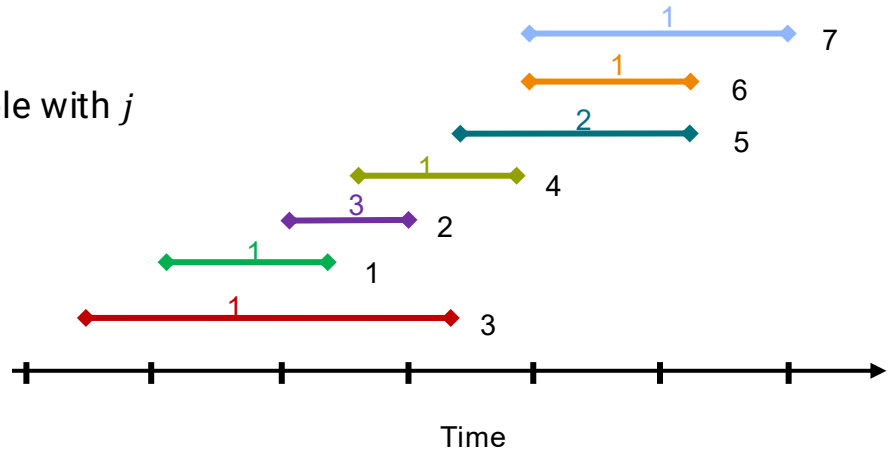
return 0

Let $i < j$ be the highest-numbered act compatible with j

$val1 \leftarrow v_j + \text{ComputeValue}(i)$

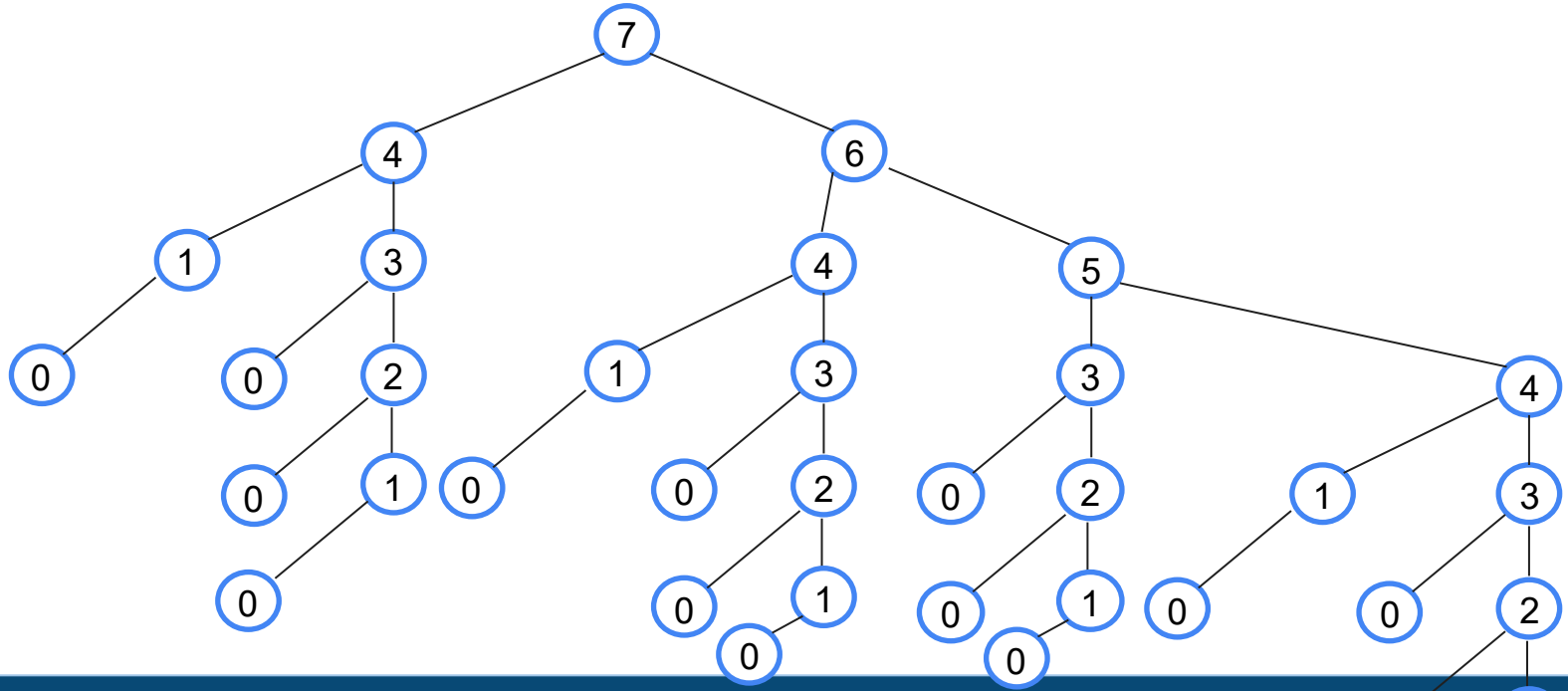
$val2 \leftarrow \text{ComputeValue}(j-1)$

return $\max(val1, val2)$



Recursion Tree

Let's consider the case of $\mathrm{ComputeValue}(n)$ for our example



Recursion Tree

- This recursion tree is really awful!
- Exponential time in the worst case
- However: there are only n unique subproblems
- **Idea**: we can save work by only solving each problem once and storing the value!

Step 2: Recurrence

Let $\mathrm{OPT}(j)$ be the value of the optimal solution on the first j acts

- $\mathrm{OPT}(0) = 0$
- $\mathrm{OPT}(j) = \max\{v_j + \mathrm{OPT}(p_j), \mathrm{OPT}(j-1)\}$
- p_j is the highest-numbered act $i < j$ that's compatible with j

Step 2: Recurrence

Let $\mathrm{OPT}(j)$ be the value of the optimal solution on the first j acts

- $\mathrm{OPT}(0) = 0$
- $\mathrm{OPT}(j) = \max\{v_j + \mathrm{OPT}(p_j), \mathrm{OPT}(j-1)\}$
- p_j is the highest-numbered act $i < j$ that's compatible with j

Notice that this recurrence directly corresponds to the recursive algorithm

- **Tip**: start by writing the recursive algorithm and translating it to a recurrence
- After some practice, you can skip straight to forming the recurrence

Step 3: Iterative Bottom-Up Algorithm

Idea: compute the optimal value of every unique subproblem in order from smallest (base case) to largest (original problem). Use recurrence for each subproblem.

- We call this "inductively" computing the optimal value

$\text{WeIS}(n)$:

Initialize list M of size n to hold optimal values

$M[0] \leftarrow 0$

for $j = 1$ to n **do**

$$M[j] = \max\left(v_j + M\left[p_j\right], M[j-1]\right)$$

return M

Step 3: Iterative Bottom-Up Algorithm

Idea: compute the optimal value of every unique subproblem in order from smallest (base case) to largest (original problem). Use recurrence for each subproblem.

- We call this "inductively" computing the optimal value

$\text{WeIS}(n)$:

Initialize list M of size n to hold optimal values

$M[0] \leftarrow 0$

for $j = 1$ to n **do**

$$M[j] = \max\left(v_j + M\left[p_j\right], M[j-1]\right)$$

return M

- Running time?

Step 3: Iterative Bottom-Up Algorithm

Idea: compute the optimal value of every unique subproblem in order from smallest (base case) to largest (original problem). Use recurrence for each subproblem.

- We call this "inductively" computing the optimal value

$\text{WeIS}(n)$:

Initialize list M of size n to hold optimal values

$M[0] \leftarrow 0$

for $j = 1$ to n **do**

$$M[j] = \max\left(v_j + M\left[p_j\right], M[j-1]\right)$$

return M

- Running time? $O(n)$

Memoization

Alternate approach

- Keep the recursive function structure, but store value in array on first computation and reuse it
- Can be useful if we are unsure of the right iteration order
- Typically, same running time as Iterative Bottom-Up approach
- In practice: memoization is slower and uses more memory due to function call overhead and stack management

Step 4: Recover the Solution

Idea:

- Save best choice for each subproblem
- Track back from end

Step 4: Recover the Solution

- Save best choice for each subproblem

$\text{WeIS}(n)$:

Initialize list M of size n to hold optimal values

Initialize list B of size n to hold choices

$M[0] \leftarrow 0$

for $j = 1$ to n **do**

$M[j] = \max\left(v_j + M\left[p_j\right], M[j-1]\right)$

if $v_j + M\left[p_j\right] > M[j-1]$ **do**

$B[j] = 1$

else **do**

$B[j] = 0$

return M

Step 4: Recover the Solution

- Track back from end

Given lists M, B:

$A \leftarrow \{\}$

$j \leftarrow n$

while $j > 0$ **do**

  if $B[j] == 1$ **do**

    $A = A \leftarrow \{j\}$

    $j \leftarrow p_j$

  else do

    $j \leftarrow j - 1$

Review

Recursive algorithm → find recurrence → write iterative algorithm

Three ways of expressing value of optimal solutions of subproblems

- $\mathrm{ComputeValue(j)}$: recursive algorithm
- $\mathrm{OPT}(j)$: used in recurrence; matches recursive algorithm
- $M[j]$: array to hold optimal values for each distinct subproblem; filled in during the iterative algorithm

Next Time

- Subset sum problem (SSP) & Knapsack problem