CSE 431/531: Algorithm Analysis and Design (Fall 2024)
Dynamic Programming

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Paradigms for Designing Algorithms

Greedy algorithm

- Make a greedy choice
- Prove that the greedy choice is safe
- Reduce the problem to a sub-problem and solve it iteratively
- Usually for optimization problems

Divide-and-conquer

- Break a problem into many independent sub-problems
- Solve each sub-problem separately
- Combine solutions for sub-problems to form a solution for the original one
- Usually used to design more efficient algorithms

Paradigms for Designing Algorithms

Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse

Recall: Computing the n-th Fibonacci Number

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$

Fib(n)

```
F[0] ← 0
F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Recall: Computing the n-th Fibonacci Number

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$

Fib(n)

```
F[0] ← 0
F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

- Store each $F[i]$ for future use.

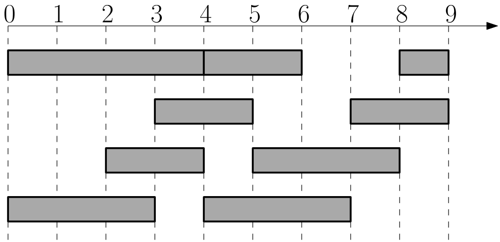
Outline

Recall: Interval Schduling

Input: n jobs, job i with start time s_i and finish time f_i

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-size subset of mutually compatible jobs

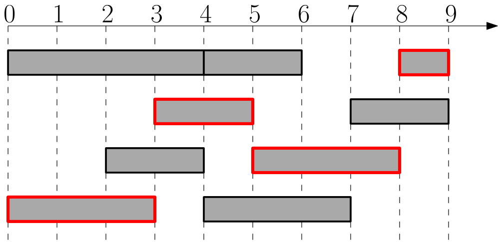


Recall: Interval Schduling

Input: n jobs, job i with start time s_i and finish time f_i

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-size subset of mutually compatible jobs



Weighted Interval Scheduling

Input: n jobs, job i with start time s_i and finish time f_i

each job has a weight (or value) $v_i > 0$

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-weight subset of mutually compatible jobs

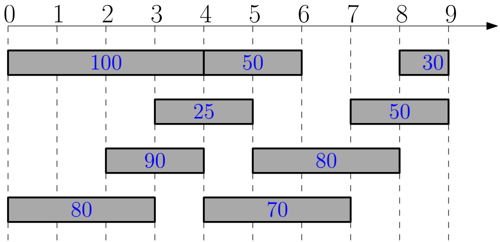
Weighted Interval Scheduling

Input: n jobs, job i with start time s_i and finish time f_i

each job has a weight (or value) $v_i > 0$

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-weight subset of mutually compatible jobs



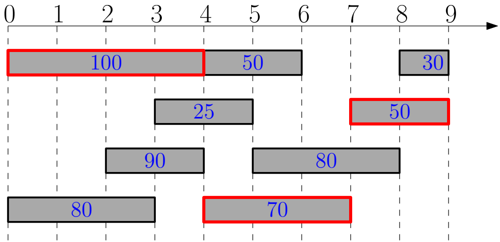
Weighted Interval Scheduling

Input: n jobs, job i with start time s_i and finish time f_i

each job has a weight (or value) $v_i > 0$

i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

Output: a maximum-weight subset of mutually compatible jobs



Optimum value = 220

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time?

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight?

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
- Job with the largest $\frac{\text{weight}}{\text{length}}$?

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
- Job with the largest $\frac{\text{weight}}{\text{length}}$?

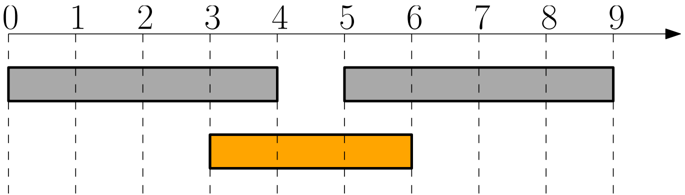
  No, when weights are equal, this is the shortest job

Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

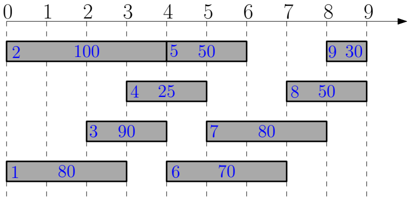
- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
- Job with the largest $\frac{\text{weight}}{\text{length}}$?

  No, when weights are equal, this is the shortest job



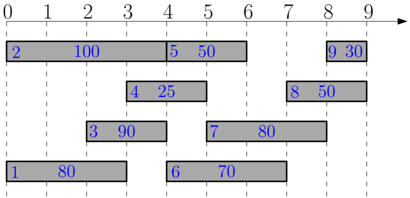
Designing a Dynamic Programming Algorithm

Designing a Dynamic Programming Algorithm



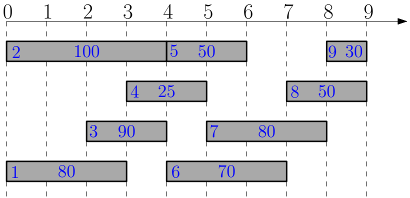
- Sort jobs according to non-decreasing order of finish times

Designing a Dynamic Programming Algorithm



- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

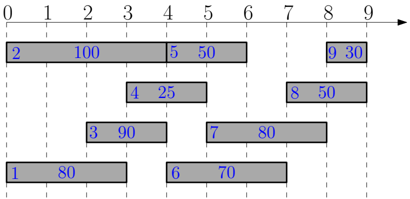
Designing a Dynamic Programming Algorithm



- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

i	$opt[i]$
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

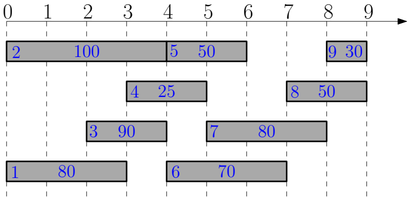
Designing a Dynamic Programming Algorithm



- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

i	$opt[i]$
0	0
1	
2	
3	
4	
5	
6	
7	
8	
9	

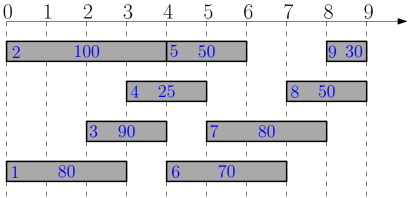
Designing a Dynamic Programming Algorithm



- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

i	$opt[i]$
0	0
1	80
2	
3	
4	
5	
6	
7	
8	
9	

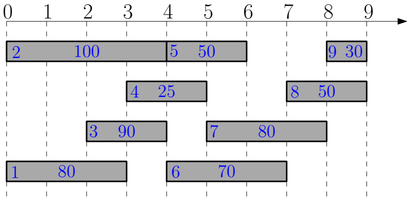
Designing a Dynamic Programming Algorithm



i	$opt[i]$
0	0
1	80
2	100
3	
4	
5	
6	
7	
8	
9	

- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

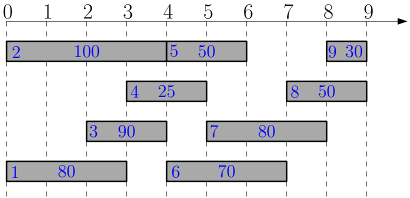
Designing a Dynamic Programming Algorithm



i	$opt[i]$
0	0
1	80
2	100
3	100
4	
5	
6	
7	
8	
9	

- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

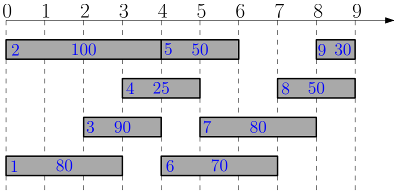
Designing a Dynamic Programming Algorithm



- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

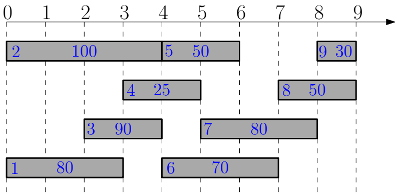
i	$opt[i]$
0	0
1	80
2	100
3	100
4	105
5	150
6	170
7	185
8	220
9	220

Designing a Dynamic Programming Algorithm



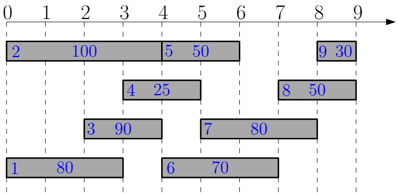
- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance

Designing a Dynamic Programming Algorithm



- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

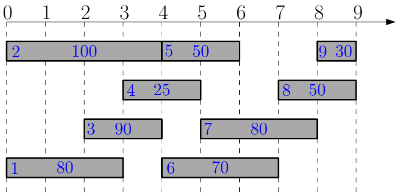
Designing a Dynamic Programming Algorithm



- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

Q: The value of optimal solution that does not contain i?

Designing a Dynamic Programming Algorithm

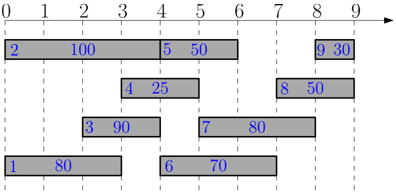


- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Designing a Dynamic Programming Algorithm



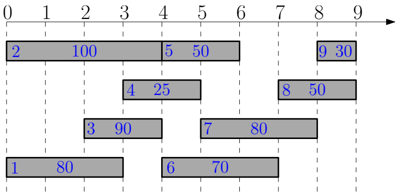
- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

Designing a Dynamic Programming Algorithm



- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

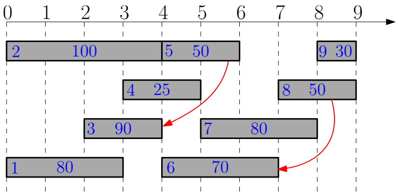
Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

A: $v_i + opt[p_i]$, $\quad p_i =$ the largest j such that $f_j \leq s_i$

Designing a Dynamic Programming Algorithm



- Focus on instance $\{1, 2, 3, \cdots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \cdots, opt[i-1]$

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

A: $v_i + opt[p_i]$, $\quad p_i =$ the largest j such that $f_j \leq s_i$

Designing a Dynamic Programming Algorithm

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

A: $v_i + opt[p_i]$, $\quad p_i =$ the largest j such that $f_j \le s_i$

Designing a Dynamic Programming Algorithm

Q: The value of optimal solution that does not contain i?

A: $opt[i-1]$

Q: The value of optimal solution that contains job i?

A: $v_i + opt[p_i]$, $p_i =$ the largest j such that $f_j \le s_i$

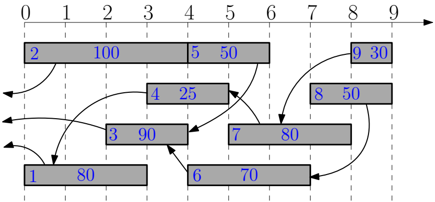
Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

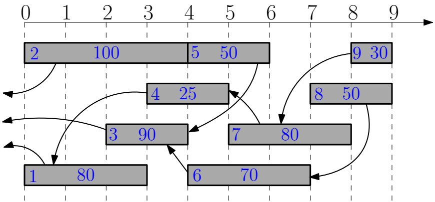


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] =$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

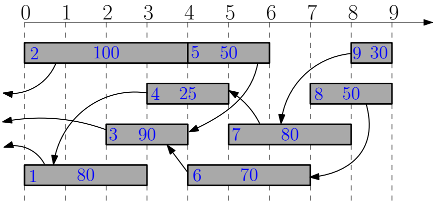


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] =$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

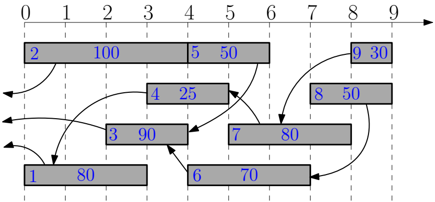


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\}$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

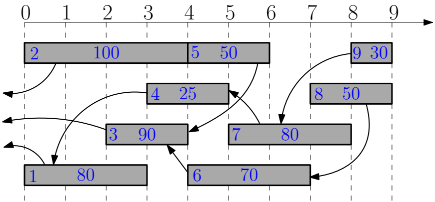


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

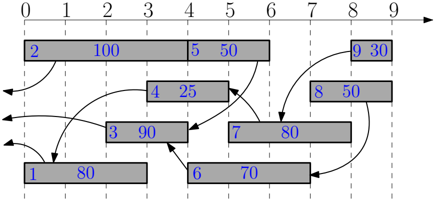


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\}$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

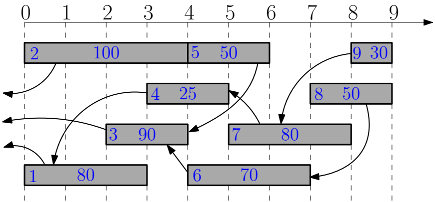


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

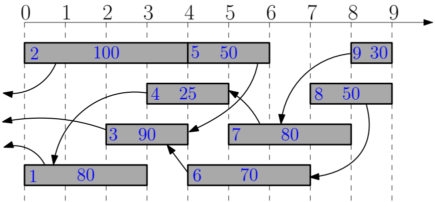


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\}$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

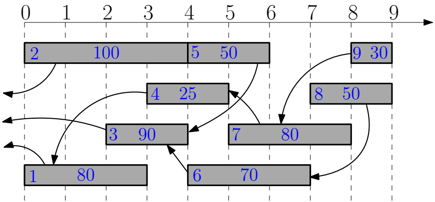


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

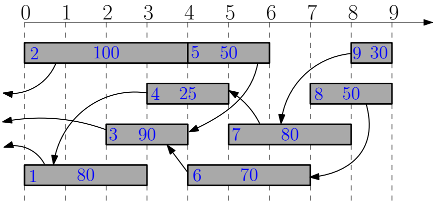


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] = \max\{opt[4], 50 + opt[3]\}$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

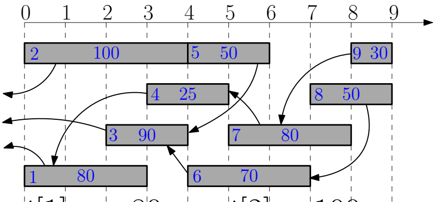


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] = \max\{opt[4], 50 + opt[3]\} = 150$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$

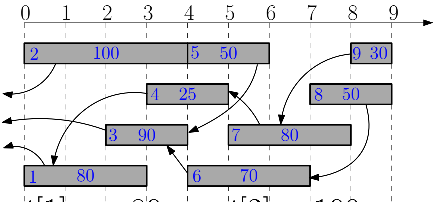


- $opt[0] = 0$, $opt[1] = 80$, $opt[2] = 100$
- $opt[3] = 100$, $opt[4] = 105$, $opt[5] = 150$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

$$opt[i] = \max\{opt[i-1], v_i + opt[p_i]\}$$



- $opt[0] = 0, \quad opt[1] = 80, \quad opt[2] = 100$
- $opt[3] = 100, \quad opt[4] = 105, \quad opt[5] = 150$
- $opt[6] = \max\{opt[5], 70 + opt[3]\} = 170$
- $opt[7] = \max\{opt[6], 80 + opt[4]\} = 185$
- $opt[8] = \max\{opt[7], 50 + opt[6]\} = 220$
- $opt[9] = \max\{opt[8], 30 + opt[7]\} = 220$

Dynamic Programming

sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    $opt[i] \leftarrow \max\{opt[i-1], v_i + opt[p_i]\}$

Dynamic Programming

```
sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
**for** $i \leftarrow 1$ to $n$ **do**
    $opt[i] \leftarrow \max\{opt[i-1], v_i + opt[p_i]\}$
```

- Running time sorting: $O(n \lg n)$
- Running time for computing p: $O(n \lg n)$ via binary search
- Running time for computing $opt[n]$: $O(n)$

How Can We Recover the Optimum Schedule?

sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    if $opt[i-1] \geq v_i + opt[p_i]$ **then**
        $opt[i] \leftarrow opt[i-1]$
    else
        $opt[i] \leftarrow v_i + opt[p_i]$

How Can We Recover the Optimum Schedule?

sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    if $opt[i-1] \geq v_i + opt[p_i]$ **then**
        $opt[i] \leftarrow opt[i-1]$
        $b[i] \leftarrow$ N
    else
        $opt[i] \leftarrow v_i + opt[p_i]$
        $b[i] \leftarrow$ Y

How Can We Recover the Optimum Schedule?

sort jobs by non-decreasing order of finishing times
compute $p_1, p_2, \cdots, p_n$
$opt[0] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  if $opt[i-1] \geq v_i + opt[p_i]$ **then**
    $opt[i] \leftarrow opt[i-1]$
    $b[i] \leftarrow$ N
  else
    $opt[i] \leftarrow v_i + opt[p_i]$
    $b[i] \leftarrow$ Y

$i \leftarrow n, S \leftarrow \emptyset$
while $i \neq 0$ **do**
  if $b[i] =$ N **then**
    $i \leftarrow i - 1$
  else
    $S \leftarrow S \cup \{i\}$
    $i \leftarrow p_i$
return S