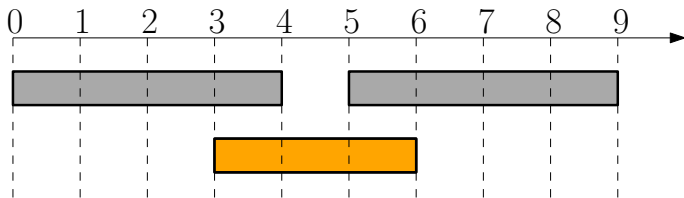


Hard to Design a Greedy Algorithm

Q: Which job is safe to schedule?

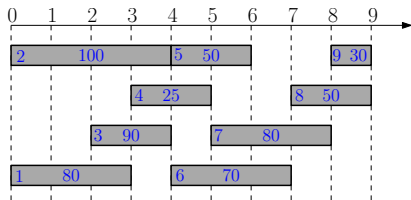
- Job with the earliest finish time? No, we are ignoring weights
- Job with the largest weight? No, we are ignoring times
- Job with the largest $\frac{\text{weight}}{\text{length}}$?

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



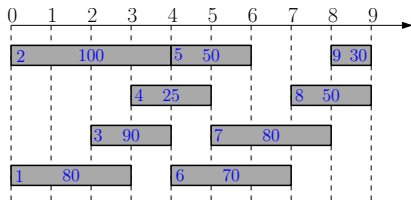
Designing a Dynamic Programming Algorithm

Designing a Dynamic Programming Algorithm



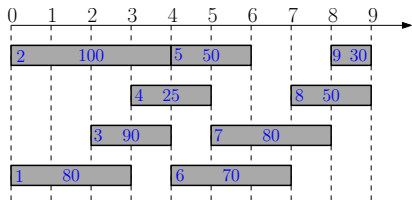
- Sort jobs according to non-decreasing order of finish times

Designing a Dynamic Programming Algorithm



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

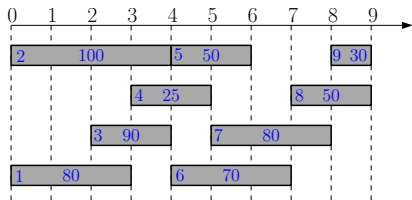
Designing a Dynamic Programming Algorithm



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

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

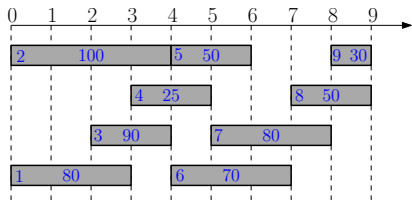
Designing a Dynamic Programming Algorithm



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

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

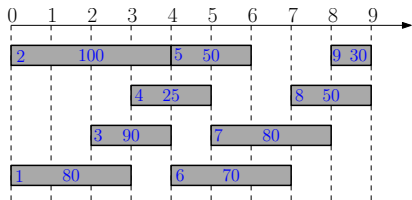
Designing a Dynamic Programming Algorithm



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

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

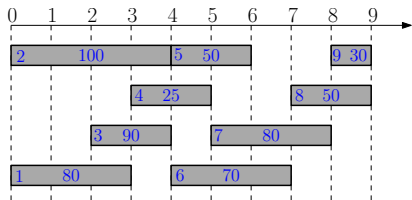
Designing a Dynamic Programming Algorithm



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

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

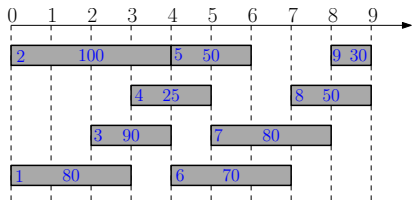
Designing a Dynamic Programming Algorithm



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

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

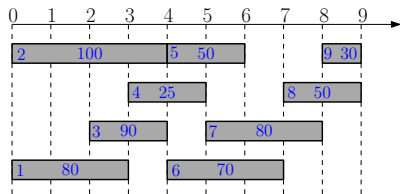
Designing a Dynamic Programming Algorithm



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

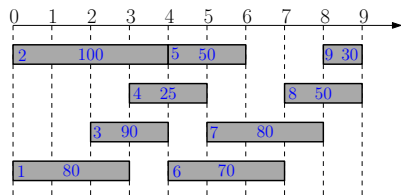
i	$opt[i]$
0	0
1	80
2	100
3	100
4	105
5	150
6	170
7	185
8	220
9	220

Designing a Dynamic Programming Algorithm



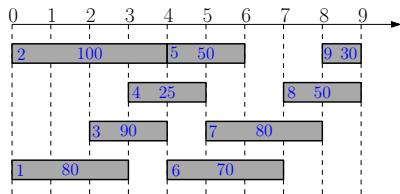
- Focus on instance $\{1, 2, 3, \dots, i\}$,
- $opt[i]$: optimal value for the instance

Designing a Dynamic Programming Algorithm



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

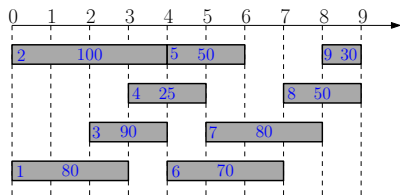
Designing a Dynamic Programming Algorithm



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

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

Designing a Dynamic Programming Algorithm

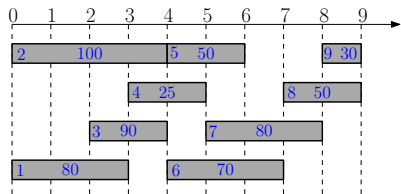


- Focus on instance $\{1, 2, 3, \dots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \dots, opt[i - 1]$

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

A: $opt[i - 1]$

Designing a Dynamic Programming Algorithm



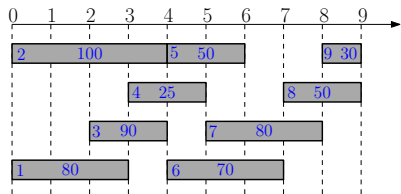
- Focus on instance $\{1, 2, 3, \dots, i\}$,
- $opt[i]$: optimal value for the instance
- assume we have computed $opt[0], opt[1], \dots, opt[i - 1]$

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

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

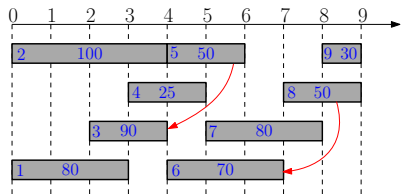
A: $opt[i - 1]$

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

A: $v_i + opt[p_i]$,

$p_i =$ the largest j such that $f_j \leq s_i$

Designing a Dynamic Programming Algorithm



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

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: $v_i + opt[p_i]$,

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

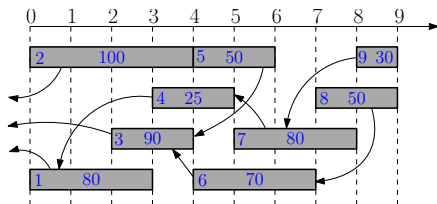
Recursion for $opt[i]$:

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

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

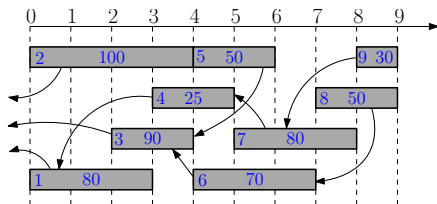


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] =$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

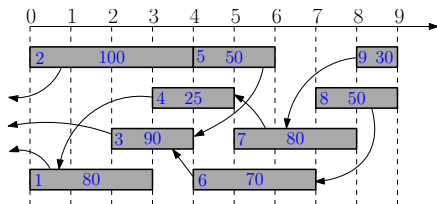


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] =$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

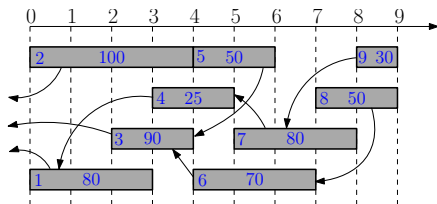


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\}$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

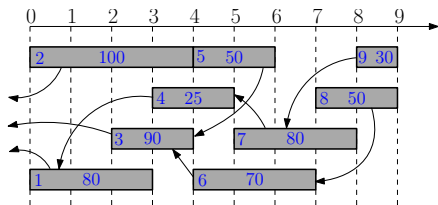


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] =$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

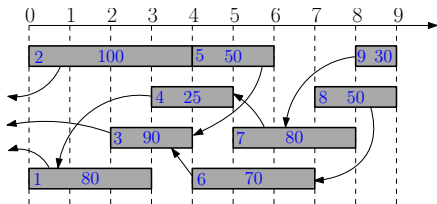


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\}$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

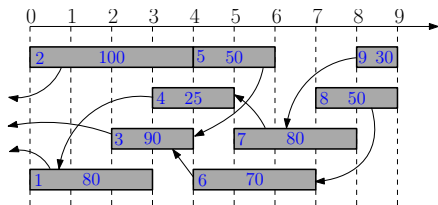


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] =$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

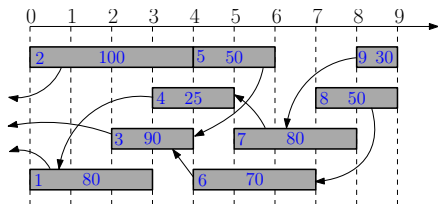


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\}$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

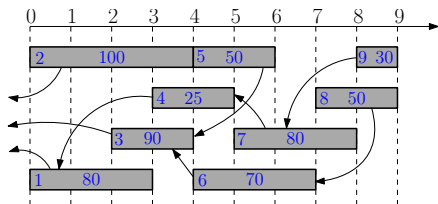


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] =$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

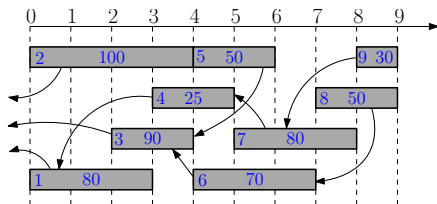


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] = \max\{opt[4], 50 + opt[3]\}$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

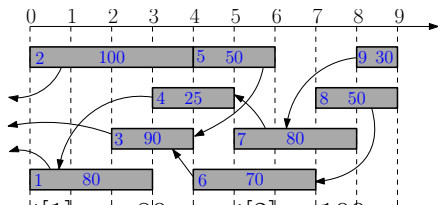


- $opt[0] = 0$
- $opt[1] = \max\{opt[0], 80 + opt[0]\} = 80$
- $opt[2] = \max\{opt[1], 100 + opt[0]\} = 100$
- $opt[3] = \max\{opt[2], 90 + opt[0]\} = 100$
- $opt[4] = \max\{opt[3], 25 + opt[1]\} = 105$
- $opt[5] = \max\{opt[4], 50 + opt[3]\} = 150$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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

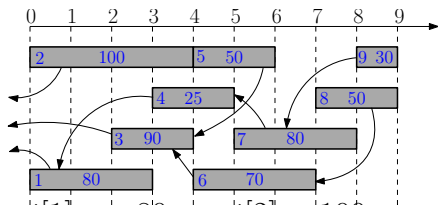


- $opt[0] = 0$, $opt[1] = 80$, $opt[2] = 100$
- $opt[3] = 100$, $opt[4] = 105$, $opt[5] = 150$

Designing a Dynamic Programming Algorithm

Recursion for $opt[i]$:

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



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

Dynamic Programming

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

Dynamic Programming

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

- 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?

```
1: sort jobs by non-decreasing order of
   finishing times
2: compute  $p_1, p_2, \dots, p_n$ 
3:  $opt[0] \leftarrow 0$ 
4: for  $i \leftarrow 1$  to  $n$  do
5:     if  $opt[i - 1] \geq v_i + opt[p_i]$  then
6:          $opt[i] \leftarrow opt[i - 1]$ 
7:
8:     else
9:          $opt[i] \leftarrow v_i + opt[p_i]$ 
10:
```

How Can We Recover the Optimum Schedule?

```
1: sort jobs by non-decreasing order of
   finishing times
2: compute  $p_1, p_2, \dots, p_n$ 
3:  $opt[0] \leftarrow 0$ 
4: for  $i \leftarrow 1$  to  $n$  do
5:     if  $opt[i - 1] \geq v_i + opt[p_i]$  then
6:          $opt[i] \leftarrow opt[i - 1]$ 
7:          $b[i] \leftarrow N$ 
8:     else
9:          $opt[i] \leftarrow v_i + opt[p_i]$ 
10:         $b[i] \leftarrow Y$ 
```

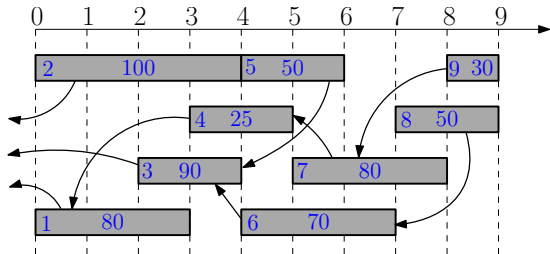
How Can We Recover the Optimum Schedule?

```
1: sort jobs by non-decreasing order of
   finishing times
2: compute  $p_1, p_2, \dots, p_n$ 
3:  $opt[0] \leftarrow 0$ 
4: for  $i \leftarrow 1$  to  $n$  do
5:     if  $opt[i - 1] \geq v_i + opt[p_i]$  then
6:          $opt[i] \leftarrow opt[i - 1]$ 
7:          $b[i] \leftarrow N$ 
8:     else
9:          $opt[i] \leftarrow v_i + opt[p_i]$ 
10:         $b[i] \leftarrow Y$ 
```

```
1:  $i \leftarrow n, S \leftarrow \emptyset$ 
2: while  $i \neq 0$  do
3:     if  $b[i] = N$  then
4:          $i \leftarrow i - 1$ 
5:     else
6:          $S \leftarrow S \cup \{i\}$ 
7:          $i \leftarrow p_i$ 
8: return  $S$ 
```

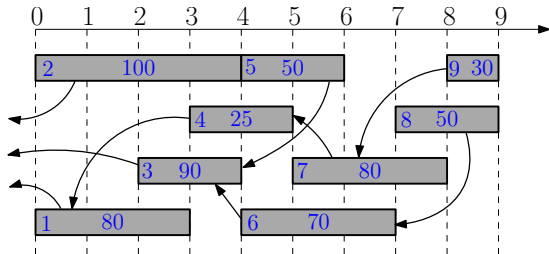
Recovering Optimum Schedule: Example

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



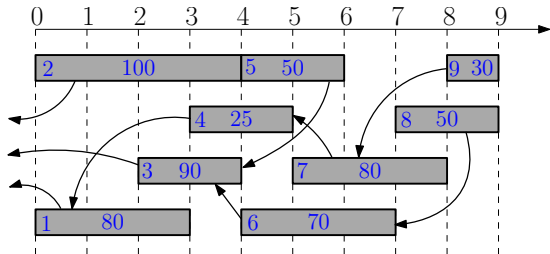
Recovering Optimum Schedule: Example

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



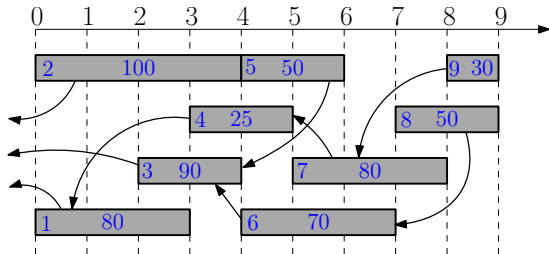
Recovering Optimum Schedule: Example

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



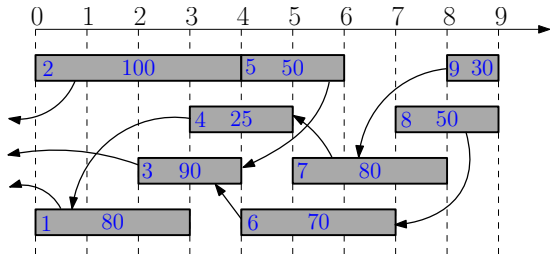
Recovering Optimum Schedule: Example

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



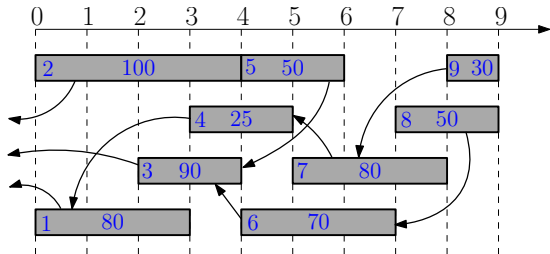
Recovering Optimum Schedule: Example

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



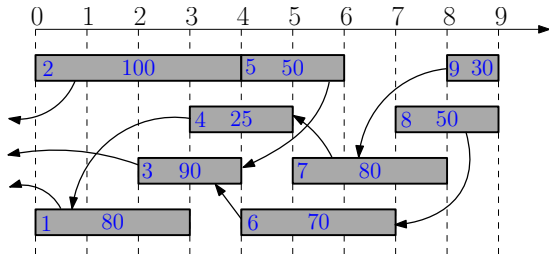
Recovering Optimum Schedule: Example

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



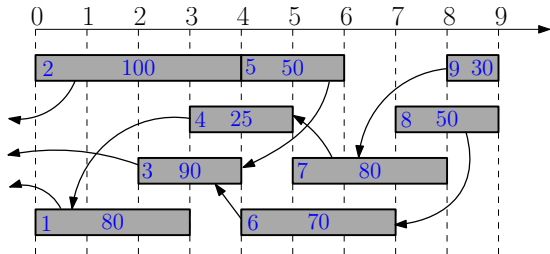
Recovering Optimum Schedule: Example

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



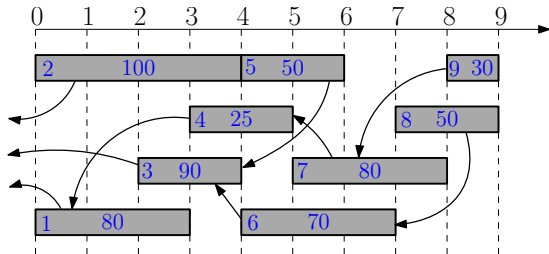
Recovering Optimum Schedule: Example

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



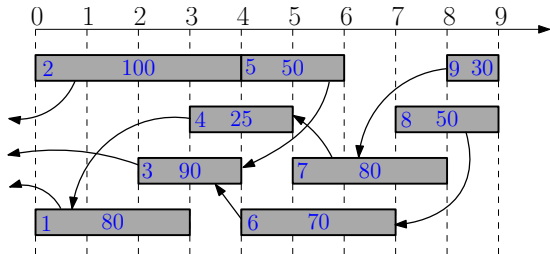
Recovering Optimum Schedule: Example

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



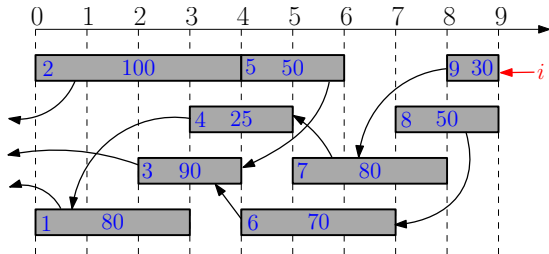
Recovering Optimum Schedule: Example

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



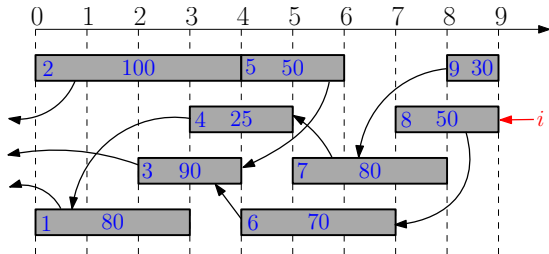
Recovering Optimum Schedule: Example

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



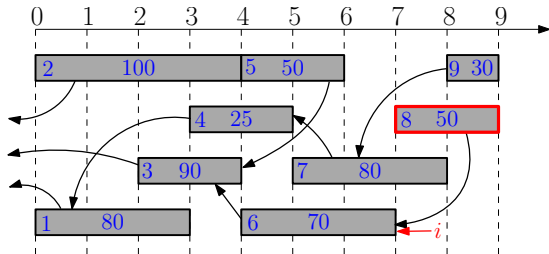
Recovering Optimum Schedule: Example

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



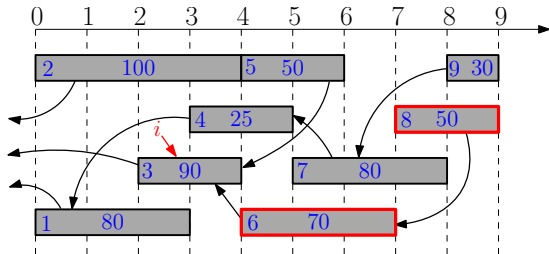
Recovering Optimum Schedule: Example

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



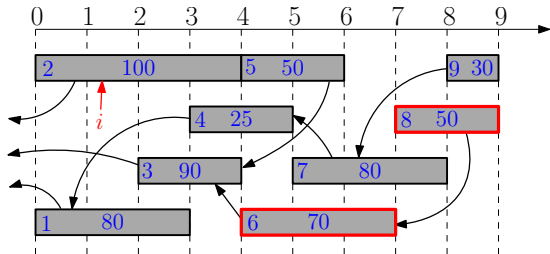
Recovering Optimum Schedule: Example

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



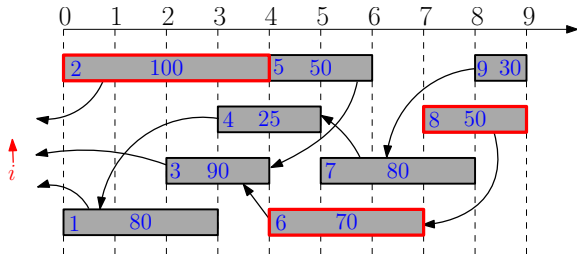
Recovering Optimum Schedule: Example

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



Recovering Optimum Schedule: Example

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



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

Outline

- 1 Weighted Interval Scheduling
- 2 Subset Sum Problem**
- 3 Knapsack Problem
- 4 Longest Common Subsequence
 - Longest Common Subsequence in Linear Space
- 5 Shortest Paths in Directed Acyclic Graphs
- 6 Matrix Chain Multiplication
- 7 Optimum Binary Search Tree
- 8 Summary

Subset Sum Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} w_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$

Subset Sum Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} w_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$

- Motivation: you have budget W , and want to buy a subset of items, so as to spend as much money as possible.

Subset Sum Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} w_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$

- Motivation: you have budget W , and want to buy a subset of items, so as to spend as much money as possible.

Example:

- $W = 35, n = 5, w = (14, 9, 17, 10, 13)$

Subset Sum Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} w_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$

- Motivation: you have budget W , and want to buy a subset of items, so as to spend as much money as possible.

Example:

- $W = 35, n = 5, w = (14, 9, 17, 10, 13)$
- Optimum: $S = \{1, 2, 4\}$ and $14 + 9 + 10 = 33$

Greedy Algorithms for Subset Sum

Candidate Algorithm:

- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below W

Greedy Algorithms for Subset Sum

Candidate Algorithm:

- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below W

Q: Does candidate algorithm always produce optimal solutions?

Greedy Algorithms for Subset Sum

Candidate Algorithm:

- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below W

Q: Does candidate algorithm always produce optimal solutions?

A: No. $W = 100, n = 3, w = (51, 50, 50)$.

Greedy Algorithms for Subset Sum

Candidate Algorithm:

- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below W

Q: Does candidate algorithm always produce optimal solutions?

A: No. $W = 100, n = 3, w = (51, 50, 50)$.

Q: What if we change “non-increasing” to “non-decreasing”?

Greedy Algorithms for Subset Sum

Candidate Algorithm:

- Sort according to non-increasing order of weights
- Select items in the order as long as the total weight remains below W

Q: Does candidate algorithm always produce optimal solutions?

A: No. $W = 100, n = 3, w = (51, 50, 50)$.

Q: What if we change “non-increasing” to “non-decreasing”?

A: No. $W = 100, n = 3, w = (1, 50, 50)$