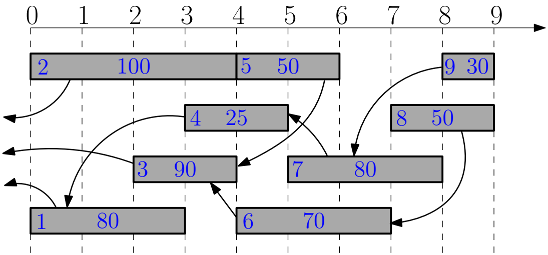
How Can We Recover the Optimum Schedule?

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

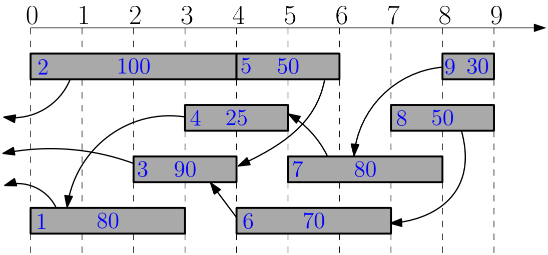
Recovering Optimum Schedule: Example

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	



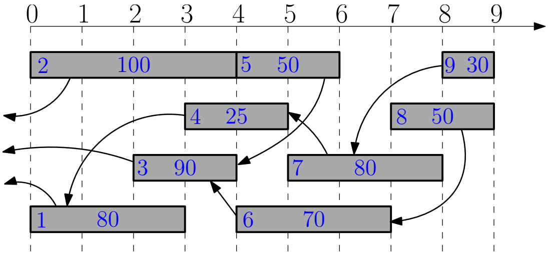
Recovering Optimum Schedule: Example

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	



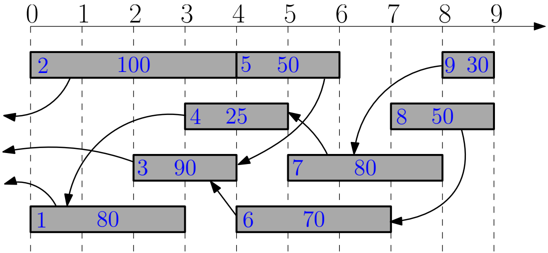
Recovering Optimum Schedule: Example

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	



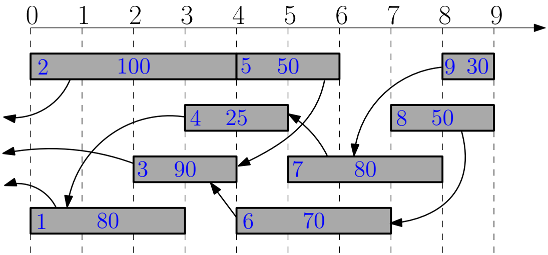
Recovering Optimum Schedule: Example

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	



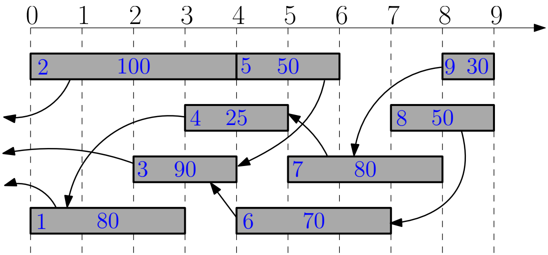
Recovering Optimum Schedule: Example

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	



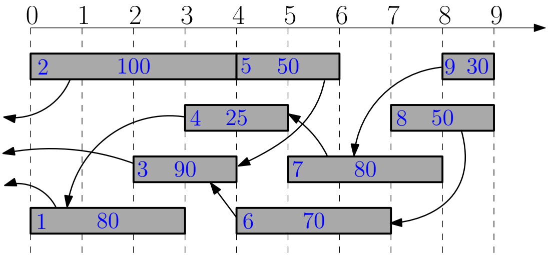
Recovering Optimum Schedule: Example

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	



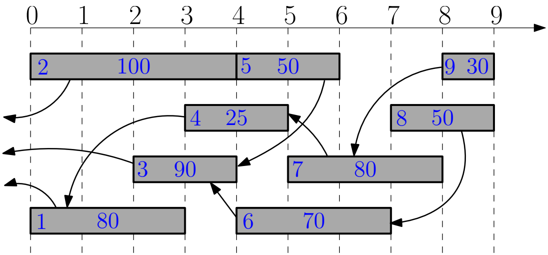
Recovering Optimum Schedule: Example

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	



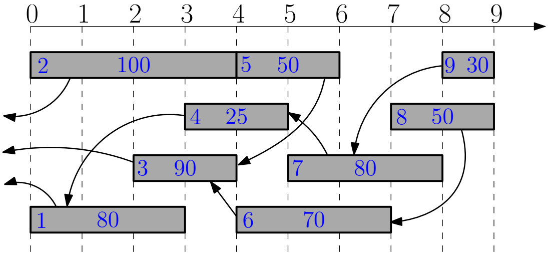
Recovering Optimum Schedule: Example

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	



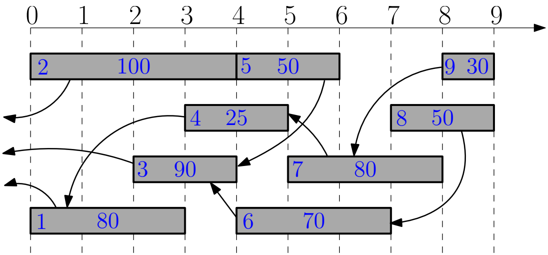
Recovering Optimum Schedule: Example

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	



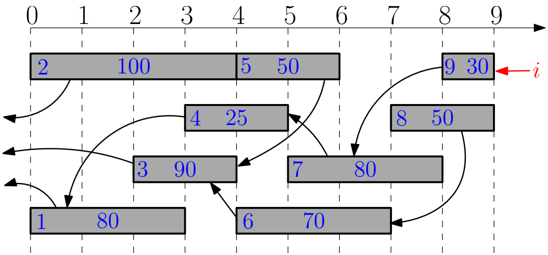
Recovering Optimum Schedule: Example

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



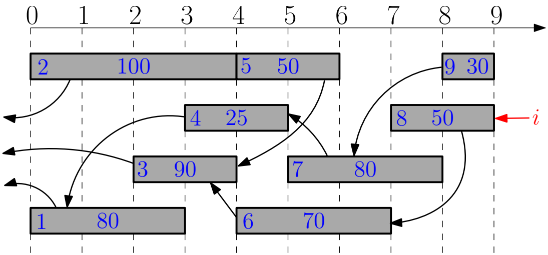
Recovering Optimum Schedule: Example

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



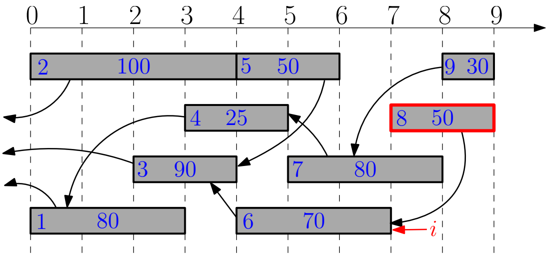
Recovering Optimum Schedule: Example

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



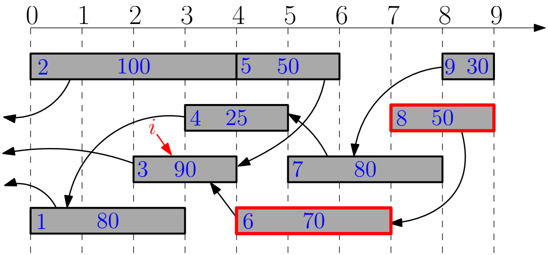
Recovering Optimum Schedule: Example

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



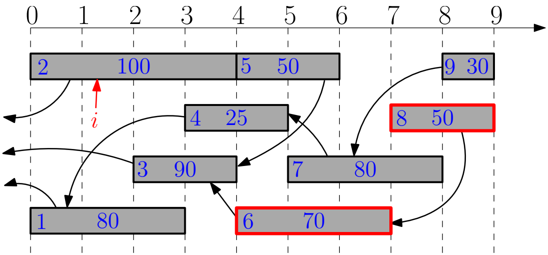
Recovering Optimum Schedule: Example

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



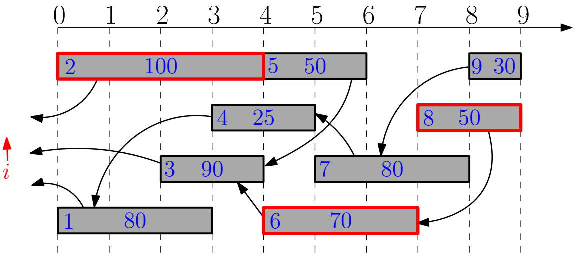
Recovering Optimum Schedule: Example

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

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

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

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

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

- 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

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

- 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

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

- 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. $W = 100, n = 3, w = (1, 50, 50)$

Design a Dynamic Programming Algorithm

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

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

Design a Dynamic Programming Algorithm

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

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

A: $opt[i-1, W']$

Design a Dynamic Programming Algorithm

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

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

A: $opt[i-1, W']$

Q: The value of the optimum solution that contains i?

Design a Dynamic Programming Algorithm

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

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

A: $opt[i-1, W']$

Q: The value of the optimum solution that contains i?

A: $opt[i-1, W'-w_i] + w_i$

Dynamic Programming

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

$$opt[i, W'] = \begin{cases} & i = 0 \\ & i > 0, w_i > W' \\ & i > 0, w_i \le W' \end{cases}$$

Dynamic Programming

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ & i > 0, w_i > W' \\ & i > 0, w_i \leq W' \end{cases}$$

Dynamic Programming

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

$$
opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ & i > 0, w_i \le W' \end{cases}
$$

Dynamic Programming

- Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;
- $opt[i, W']$: the optimum value of the instance

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max\left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W'-w_i] + w_i \end{array} \right\} & i > 0, w_i \leq W' \end{cases}$$

Dynamic Programming

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
return $opt[n, W]$

Recover the Optimum Set

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        $b[i, W'] \leftarrow$ N
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
            $b[i, W'] \leftarrow$ Y
return $opt[n, W]$

Recover the Optimum Set

```
i ← n, W' ← W, S ← ∅
while i > 0 do
    if b[i, W'] = Y then
        W' ← W' − w_i
        S ← S ∪ {i}
    i ← i − 1
return S
```

Running Time of Algorithm

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
return $opt[n, W]$

Running Time of Algorithm

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
return $opt[n, W]$

- Running time is $O(nW)$

Running Time of Algorithm

for $W' \leftarrow 0$ to W **do**
    $opt[0, W'] \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $W' \leftarrow 0$ to W **do**
        $opt[i, W'] \leftarrow opt[i-1, W']$
        if $w_i \leq W'$ and $opt[i-1, W'-w_i] + w_i \geq opt[i, W']$ **then**
            $opt[i, W'] \leftarrow opt[i-1, W'-w_i] + w_i$
return $opt[n, W]$

- Running time is $O(nW)$
- Running time is pseudo-polynomial because it depends on value of the input integers.

Running Time of Algorithm

```
for W' ← 0 to W do
    opt[0, W'] ← 0
for i ← 1 to n do
    for W' ← 0 to W do
        opt[i, W'] ← opt[i − 1, W']
        if w_i ≤ W' and opt[i − 1, W' − w_i] + w_i ≥ opt[i, W'] then
            opt[i, W'] ← opt[i − 1, W' − w_i] + w_i
return opt[n, W]
```

- Running time is $O(nW)$
- Running time is pseudo-polynomial because it depends on value of the input integers.
- Game's running time: https://courses.csail.mit.edu/6.5440/fall23/

Avoiding Unncessary Computation and Memory Using Memoized Algorithm and Hash Map

compute-opt(i, W')

if $opt[i, W'] \neq \bot$ **then return** $opt[i, W']$
if $i = 0$ **then** $r \leftarrow 0$
else
    $r \leftarrow$ compute-opt$(i-1, W')$
    if $w_i \leq W'$ **then**
        $r' \leftarrow$ compute-opt$(i-1, W'-w_i) + w_i$
        if $r' > r$ **then** $r \leftarrow r'$
$opt[i, W'] \leftarrow r$
return r

- Use hash map for opt

Outline

Knapsack Problem

Input: an integer bound $W > 0$

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

a value $v_i > 0$ for each item i

Output: a subset S of items that

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