Advanced Algorithmic Techniques (COMP523)

Dynamic Programming 2

Recap and plan

- **Last lecture:**
  - Dynamic Programming
  - Weighted Interval Scheduling
- **This lecture:**
  - Subset Sum
  - Knapsack

The subset sum problem

- We are given a set of n items {*1*, *2*, … , *n*}.
- Each item *i* has a non-negative weight w_i.
- We are given a bound W.
- Goal: Select a subset S of the items such that $\sum_{i \in S} w_i \leq W$

and $\sum_{i \in S} w_i$ is maximised.

Greedy Approaches

Greedy Approaches

- Ideas?

Greedy Approaches

- Ideas?
- Sort items in terms of *decreasing weight* and put them in S one by one.

Greedy Approaches

- Ideas?
- Sort items in terms of *decreasing weight* and put them in S one by one.
- Sort items in terms of *increasing weight* and put them in S one by one.

Greedy Approaches

- Sort items in terms of *decreasing weight* and put them in S one by one.

Greedy Approaches

- Sort items in terms of *decreasing weight* and put them in S one by one.
- Example where this fails?

Greedy Approaches

- Sort items in terms of *decreasing weight* and put them in S one by one.
- Example where this fails?
  - $W = 4$, $w_1 = 3$, $w_2 = 2$, $w_3 = 2$.

Greedy Approaches

- Sort items in terms of *increasing weight* and put them in S one by one.

Greedy Approaches

- Sort items in terms of *increasing weight* and put them in S one by one.
- Example where this fails?

Greedy Approaches

- Sort items in terms of *increasing weight* and put them in S one by one.
- Example where this fails?
  - $W = 4$, $w_1 = 1$, $w_2 = 2$, $w_3 = 2$.

Dynamic Programming

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.
- Recall the weighted interval scheduling problem. Similar approach.

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.
- Recall the weighted interval scheduling problem. Similar approach.
- Let OPT(i) be the optimal solution to the subset sum problem, using a subset of $\{1, 2, \ldots, i\}$.

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.
- Recall the weighted interval scheduling problem. Similar approach.
- Let $OPT(i)$ be the optimal solution to the subset sum problem, using a subset of $\{1, 2, \ldots, i\}$.
  - Let O_i be its value and hence O is O_n.

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.
- Recall the weighted interval scheduling problem. Similar approach.
- Let $OPT(i)$ be the optimal solution to the subset sum problem, using a subset of $\{1, 2, \ldots, i\}$.
    - Let O_i be its value and hence O is O_n.
- Should item n be in the optimal solution O or not?

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.
- Recall the weighted interval scheduling problem. Similar approach.
- Let OPT(i) be the optimal solution to the subset sum problem, using a subset of {1, 2, … , i}.
  - Let O_i be its value and hence O is O_n.
- Should item n be in the optimal solution O or not?
  - If no, then OPT(n-1) = OPT(n)

Dynamic Programming

- We need to identify the appropriate subproblems to use in order to solve the main problem.
- Recall the weighted interval scheduling problem. Similar approach.
- Let OPT(i) be the optimal solution to the subset sum problem, using a subset of $\{1, 2, \ldots, i\}$.
  - Let O_i be its value and hence O is O_n.
- Should item n be in the optimal solution O or not?
  - If no, then OPT(n-1) = OPT(n)
  - If yes, ?

If n is in O

If *n* is in O

- What information do we get about the other items?

If *n* is in O

- What information do we get about the other items?
- In weighted interval scheduling, we could remove all intervals overlapping with *n*.

If n is in O

- What information do we get about the other items?
- In weighted interval scheduling, we could remove all intervals overlapping with n.
- Can we do something similar here?

If n is in O

- What information do we get about the other items?
- In weighted interval scheduling, we could remove all intervals overlapping with n.
- Can we do something similar here?
  - There is no reason to a-priori exclude any remaining item, unless adding it would exceed the weight.

If *n* is in O

- What information do we get about the other items?
- In weighted interval scheduling, we could remove all intervals overlapping with *n*.
- Can we do something similar here?
  - There is no reason to a-priori exclude any remaining item, unless adding it would exceed the weight.
  - The only information that we really get is that we now have weight $W - w_n$ left.

What we really need

What we really need

- To find the optimal value of OPT(n), we need

What we really need

- To find the optimal value of OPT(n), we need
  - The optimal value of OPT(n-1) if n is not in O.

What we really need

- To find the optimal value of OPT(n), we need
  - The optimal value of OPT($n-1$) if n is not in O.
  - The optimal value of the solution on input $\{1, 2, \ldots, n-1\}$ and $w = W - w_n$.

What we really need

- To find the optimal value of OPT(n), we need
    - The optimal value of OPT(n-1) if n is not in O.
    - The optimal value of the solution on input $\{1, 2, \ldots, n-1\}$ and $w = W - w_n$.
- How many subproblems do we need?

What we really need

- To find the optimal value of OPT(n), we need
  - The optimal value of OPT($n-1$) if n is not in O.
  - The optimal value of the solution on input $\{1, 2, \ldots, n-1\}$ and $w = W - w_n$.
- How many subproblems do we need?
  - One for each initial set $\{1, 2, \ldots, i\}$ of items and each possible value for the remaining weight w.

Subproblems

Subproblems

- Assumptions:

Subproblems

- Assumptions:
  - W is an integer.

Subproblems

- Assumptions:
  - W is an integer.
  - Every w_i is an integer.

Subproblems

- Assumptions:
  - W is an integer.
  - Every w_i is an integer.
- We will have one subproblem for each $i=0,1, \ldots ,n$ and each integer $0 \leq w \leq W$.

Subproblems

- Assumptions:
  - W is an integer.
  - Every w_i is an integer.
- We will have one subproblem for each $i=0,1, \dots ,n$ and each integer $0 \leq w \leq W$.
- Let OPT(i,w) be the value of the optimal solution on subset {1, 2, … , i} and maximum allowed weight w.

Subproblems

Subproblems

- Using this notation, what are we looking for?

Subproblems

- Using this notation, what are we looking for?
  - $\mathrm{OPT}(n, W)$

Subproblems

- Using this notation, what are we looking for?
  - OPT(n,W)
- Should item n be in the optimal solution O or not?

Subproblems

- Using this notation, what are we looking for?
  - OPT(n,W)
- Should item n be in the optimal solution O or not?
  - If no, then OPT(n,W) = OPT(n-1,W).

Subproblems

- Using this notation, what are we looking for?
  - $OPT(n, W)$
- Should item n be in the optimal solution O or not?
  - If no, then $OPT(n,W) = OPT(n-1,W)$.
  - If yes, then $OPT(n,W) = w_n + OPT(n-1,W-w_n)$.

Subproblems

Is j in **O** ?

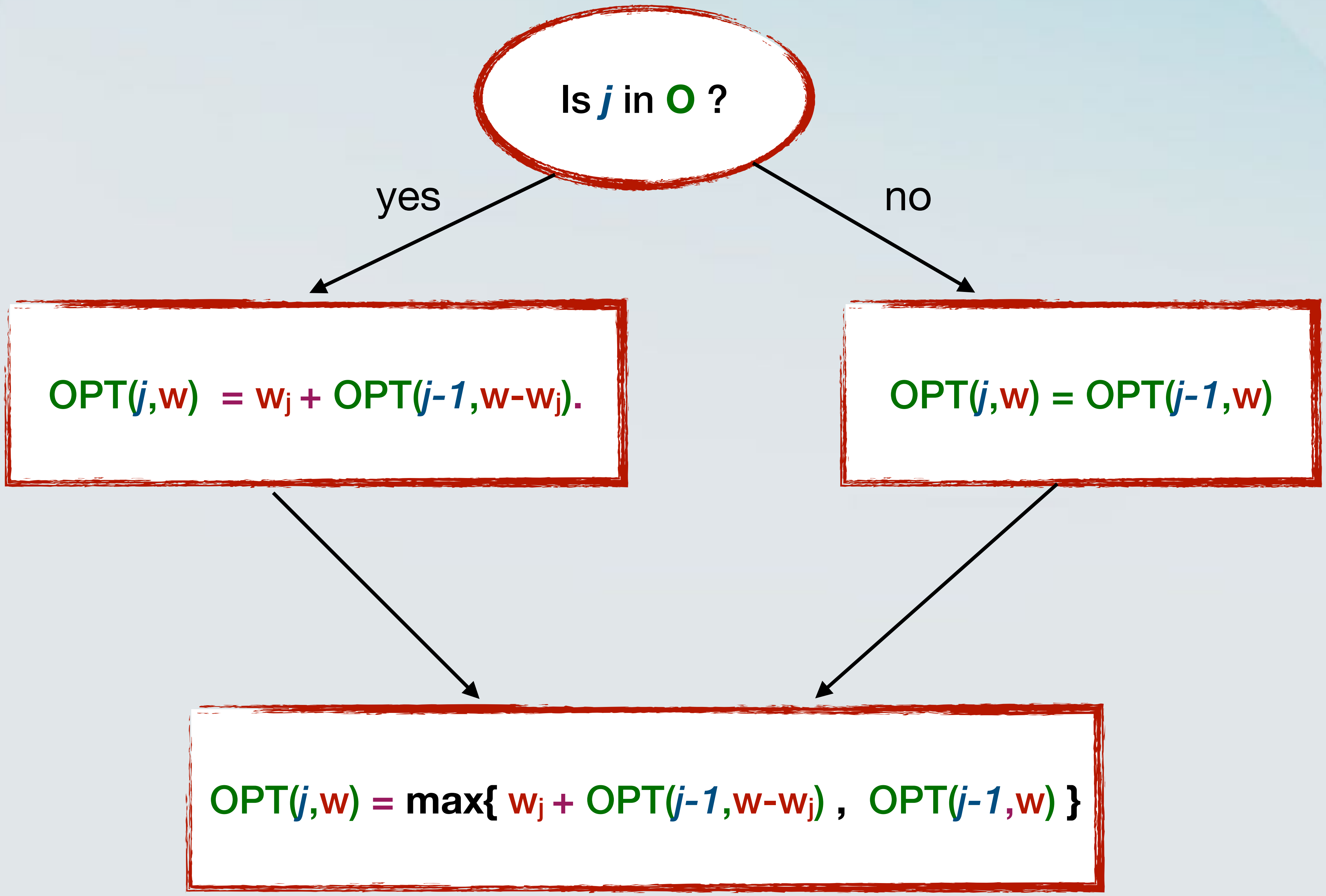
yes

$OPT(j,w) = w_j + OPT(j-1,w-w_j)$.

no

$OPT(j,w) = OPT(j-1,w)$

Subproblems



Subproblems

Is j in **O** ?

Unless $w_j < w$

yes

$\mathrm{OPT}(j,w) = w_j + \mathrm{OPT}(j-1,w-w_j)$.

no

$\mathrm{OPT}(j,w) = \mathrm{OPT}(j-1,w)$

$\mathrm{OPT}(j,w) = \max\{\, w_j + \mathrm{OPT}(j-1,w-w_j)\,,\ \mathrm{OPT}(j-1,w) \,\}$

Algorithm

Algorithm **SubsetSum**(n,W)

Array M=[$0 \ldots n$, 0 … W]
Initialise M[0, w] = 0, for each w = 0, 1 , … , W

For i = 1, 2, … , n
For w = 0 , … , W
If (w_i > w)
M[i, w] = M[i-1, w]
Else
M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
EndIf

Return M[n, W]

Two dimensional array

n	0										
n-1	0										
…	0										
…	0										
i	0										
i-1	0										
…	0										
2	0										
1	0										
0	0	0	0	0	0	0	0	0	0	0	0
	0	1	…	$w-w_i$			w				W

Example

- *n*=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Array M=[*0 … n*, 0 … W]

Initialise M[*0*, w] = 0, for each w = 0, 1 , … , W

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[*i*, w] = M[*i-1*, w]
    Else
        M[*i*, w] = **max{**M[*i-1*, w] , w_i + M[*i-1*, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
$\boxed{1}$	0						
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[*i*, w] = M[*i-1*, w]
    Else
        M[*i*, w] = **max{**M[*i-1*, w] , w_i + M[*i-1*, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
$\boxed{1}$	0	0					
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For $w = 0, \ldots, W$
    If ($w_i > w$)
        $M[i, w] = M[i\text{-}1, w]$
    Else
        $M[i, w] = \mathbf{max}\{M[i\text{-}1, w], w_i + M[i\text{-}1, w\text{-}w_i]\}$
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1	0	0	2				
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
  If (w_i > w)
    M[i, w] = M[i-1, w]
  Else
    M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
  EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
$\boxed{1}$	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

```
For w = 0 , … , W
        If (wi > w)
          M[i, w] = M[i-1, w]
        Else
          M[i, w] = max{M[i-1, w] , wi + M[i-1, w-wi ]}
        EndIf
```

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

```
For w = 0 , … , W
        If (wi > w)
            M[i, w] = M[i-1, w]
        Else
            M[i, w] = max{M[i-1, w] , wi + M[i-1, w-wi ]}
        EndIf
```

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2	0	0					
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
$\boxed{2}$	0	0	2				
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

```
For w = 0 , … , W
        If (wi > w)
          M[i, w] = M[i-1, w]
        Else
          M[i, w] = max{M[i-1, w] , wi + M[i-1, w-wi ]}
        EndIf
```

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
$\boxed{2}$	0	0	2	2			
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
$\boxed{2}$	0	0	2	2	4		
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
If (w_i > w)
M[i, w] = M[i-1, w]
Else
M[i, w] = **max**{M[i-1, w] , w_i + M[i-1, w-w_i]}
EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

$\boxed{3}$							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

$\boxed{3}$	0	0	2				
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

```
For w = 0 , … , W
        If (wi > w)
            M[i, w] = M[i-1, w]
        Else
            M[i, w] = max{M[i-1, w] , wi + M[i-1, w-wi ]}
        EndIf
```

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

$\boxed{3}$	0	0	2	3			
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

$\boxed{3}$	0	0	2	3	4		
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

```
For w = 0 , … , W
        If (wi > w)
            M[i, w] = M[i-1, w]
        Else
            M[i, w] = max{M[i-1, w] , wi + M[i-1, w-wi ]}
        EndIf
```

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

$\boxed{3}$	0	0	2	3	4	5	
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For $w = 0, \ldots, W$
    If ($w_i > w$)
        $M[i, w] = M[i-1, w]$
    Else
        $M[i, w] = \max\{M[i-1, w], w_i + M[i-1, w-w_i]\}$
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

$\boxed{3}$	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If ($w_i > w$)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

Example

- n=3, W=6, $w_1 = w_2 = 2$ and $w_3 = 3$.

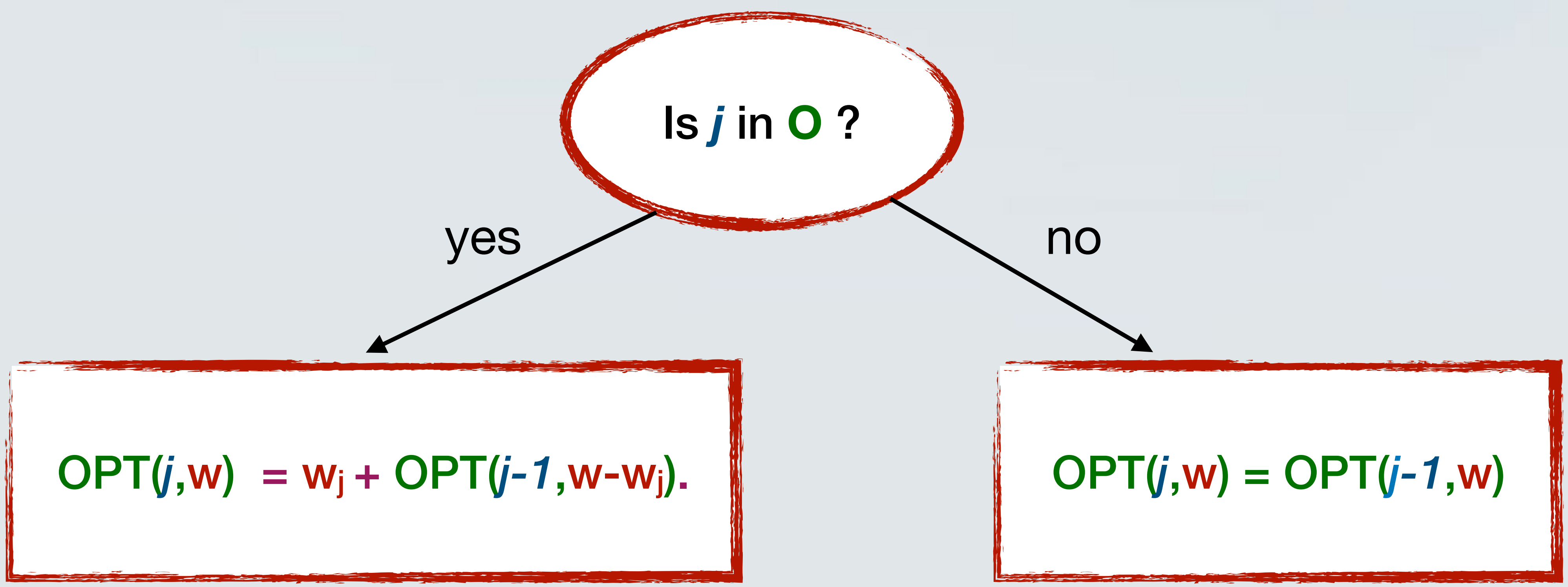
Optimal value

$\boxed{3}$	0	0	2	3	4	5	$\boxed{5}$
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

For w = 0 , … , W
    If (w_i > w)
        M[i, w] = M[i-1, w]
    Else
        M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
    EndIf

From values to solutions

- Very similar idea to weighted interval scheduling



Running Time

Running Time

- Similar to weighted interval scheduling.

Running Time

- Similar to weighted interval scheduling.
- We are building up a table M of solutions (instead of an array).

Running Time

- Similar to weighted interval scheduling.
- We are building up a table M of solutions (instead of an array).
- We compute each value M(i, w) of the table in O(1) time using the previous values.

Running Time

- Similar to weighted interval scheduling.
- We are building up a table M of solutions (instead of an array).
- We compute each value $M(i, w)$ of the table in $O(1)$ time using the previous values.
- What is the running time overall?

Running Time

- Similar to weighted interval scheduling.
- We are building up a table M of solutions (instead of an array).
- We compute each value $M(i, w)$ of the table in $O(1)$ time using the previous values.
- What is the running time overall?
    - How many entries does the table M have?

Running Time

Running Time

- **SubsetSum**(n,W) runs in time O(nW).

Running Time

- **SubsetSum**(n,W) runs in time O(nW).
- Is this a polynomial time algorithm?

Running Time

- SubsetSum(n,W) runs in time O(nW).
- Is this a polynomial time algorithm?
  - No, because it depends on W.

Running Time

- **SubsetSum**(n,W) runs in time O(nW).
- Is this a polynomial time algorithm?
  - No, because it depends on W.
  - It is *pseudopolynomial*, as it runs in time polynomial in n and W.

Running Time

- SubsetSum(n,W) runs in time O(nW).
- Is this a polynomial time algorithm?
  - No, because it depends on W.
  - It is *pseudopolynomial*, as it runs in time polynomial in n and W.
  - It is fairly efficient, if in the number involved in the input are reasonably small.

Should we be happy?

Should we be happy?

- Pseudopolynomial is good in some cases.

Should we be happy?

- Pseudopolynomial is good in some cases.
- But why not polynomial?

Should we be happy?

- Pseudopolynomial is good in some cases.
- But why not polynomial?
- How hard is it to design a polynomial time algorithm for subset sum?

Should we be happy?

- Pseudopolynomial is good in some cases.
- But why not polynomial?
- How hard is it to design a polynomial time algorithm for subset sum?
    - Hard enough to justify a reward of 1 million dollars!

Should we be happy?

- Pseudopolynomial is good in some cases.
- But why not polynomial?
- How hard is it to design a polynomial time algorithm for subset sum?
  - Hard enough to justify a reward of 1 million dollars!
  - Subset sum is NP-hard!

Should we be happy?

- Pseudopolynomial is good in some cases.
- But why not polynomial?
- How hard is it to design a polynomial time algorithm for subset sum?
    - Hard enough to justify a reward of 1 million dollars!
    - Subset sum is NP-hard!
    - More about that later on in the module.

The subset sum problem

- We are given a set of n items {*1*, *2*, … , *n*}.
- Each item *i* has a non-negative weight w_i.
- We are given a bound W.
- Goal: Select a subset S of the items such that $\sum_{i \in S} w_i \leq W$

and $\sum_{i \in S} w_i$ is maximised.

The knapsack problem

- We are given a set of n items {*1*, *2*, … , *n*}.
- Each item *i* has a non-negative weight w_i and a non-negative value v_i.
- We are given a bound W.
- Goal: Select a subset S of the items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximised.

The knapsack problem

- The subset sum problem is a specific instance of the knapsack problem (why?)
- You have actually seen this problem before!

The fractional knapsack problem

- We are given a set of n items {*1*, *2*, … , *n*}.
- Each item *i* has a non-negative weight w_i and a non-negative value v_i.
- We are given a bound W.
- Goal: Select a fraction x_i from each item *i* to maximise

$$\sum_{i \in n} x_i \cdot v_i \quad \text{such that} \quad \sum_{i \in n} x_i \cdot w_i \leq W$$

A greedy solution

- Sort the items in terms of their "bang-per-buck" value v_i / w_i : v_1 / w_1, v_2 / w_2, … , v_n / w_n.
- Put as much as possible from the first item in the knapsack, then as much as possible from the second item, … and so on.
- This algorithm solves the fractional knapsack problem optimally.

The knapsack problem

The 0/1 knapsack problem

The 0/1 knapsack problem

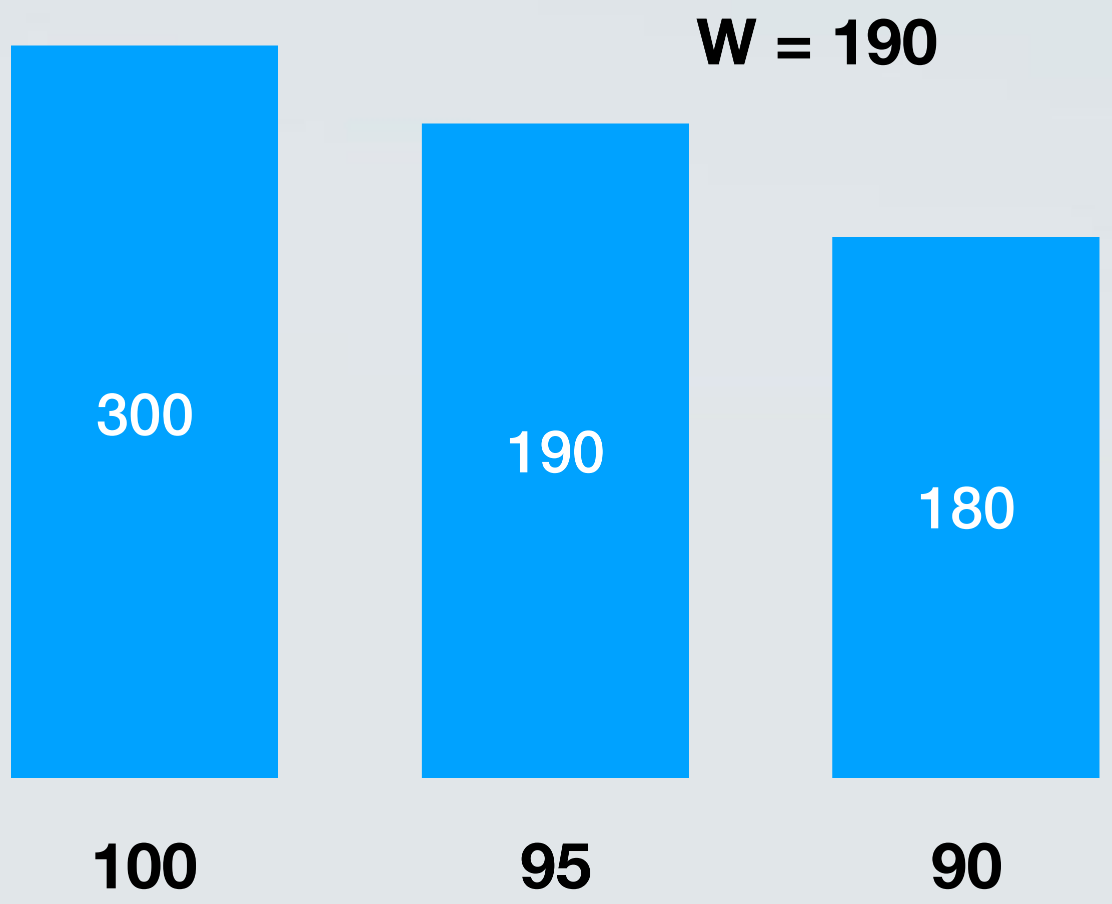
- Could the greedy algorithm solve the 0/1 knapsack problem optimally?

The 0/1 knapsack problem

- Could the greedy algorithm solve the 0/1 knapsack problem optimally?
- Counter-example:

The 0/1 knapsack problem

- Could the greedy algorithm solve the 0/1 knapsack problem optimally?
- Counter-example:



7 minute exercise

Design a dynamic programming algorithm for 0/1 knapsack.

Algorithm **SubsetSum**(n,W)

Array M=[$0 \ldots n$, 0 … W]
Initialise M[0, w] = 0, for each w = 0, 1 , … , W

For i = 1, 2, … , n
For w = 0 , … , W
If (w_i > w)
M[i, w] = M[i-1, w]
Else
M[i, w] = **max{**M[i-1, w] , w_i + M[i-1, w-w_i]**}**
EndIf

Return M[n, W]