DP Optimization

- Used for **optimization problems**
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is built in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information

Knapsack problem

Given a set of items, each with a weight and a benefit (value), pack a knapsack with a subset of items to achieve the maximum total benefit (value). Total weight that can be carried in the knapsack is no more than some fixed number W.

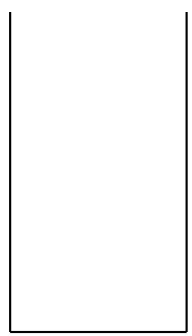
There are two versions:

1. “0-1 knapsack problem” use DP
   Items are indivisible: you either take an item or not.

2. “Fractional knapsack problem” Use a Greedy Method
   Items are divisible: you can take any fraction of an item

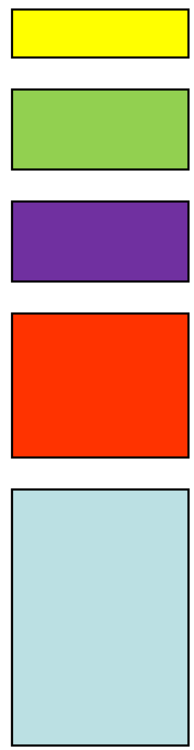
0-1 Knapsack problem:

This is a knapsack
Max weight: W = 13



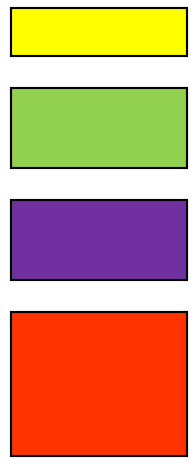
Benefit = 0
Weight = 0

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16



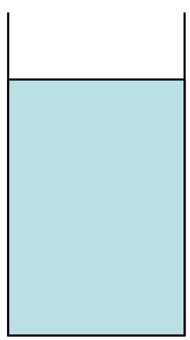
0-1 Knapsack problem:

This is a knapsack
Max weight: $W = 13$



Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Benefit = 16
Weight = 10



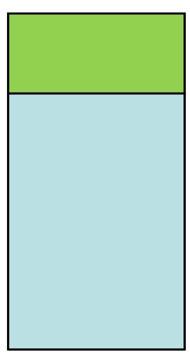
0-1 Knapsack problem:

This is a knapsack
Max weight: W = 13

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Is this maximum ?

Benefit = 20
Weight = 13

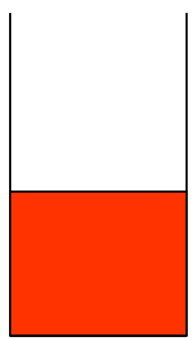


0-1 Knapsack problem:

This is a knapsack
Max weight: W = 13

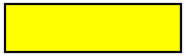
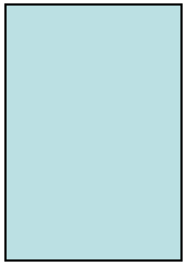
Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Benefit = 15
Weight = 5

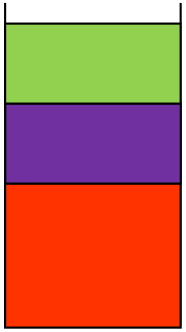


0-1 Knapsack problem:

This is a knapsack
Max weight: W = 13

Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	15
	10	16

Benefit = 24
Weight = 12



0-1 Knapsack Problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total benefit of packed items?

0-1 Knapsack problem

Let S be the set of items represented by the ordered pairs (w_i, b_i) and W be the capacity of the knapsack.

Find a T $\subseteq$ S such that

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.

0-1 Knapsack Brute-Force

Let's first solve this problem with a straightforward algorithm

- Since there are *n* items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to *W*
- Running time will be $O(n2^n)$

Can we do better?
Yes, with an algorithm based on dynamic programming
We need to carefully identify the subproblems

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for

S_k = *{items labeled 1, 2, .. k}*

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that. ….Why???

Defining a Subproblem

w_1 =2 b_1 =3	w_2 =4 b_2 =5	w_3 =3 b_3 =4	w_4 =5 b_4 =8	**?**

Max weight: W = 20

For S_4: {1, 2, 3, 4}

Total weight: 14;

total benefit: 20

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

w_1 =2 b_1 =3	w_3=4 b_3 =5	w_4 =5 b_4 =8	w_5 =9 b_5 =10

For S_5: { 1, 3, 4, 5 }

Total weight: 20

total benefit: 26

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: *w*, which will represent the exact weight for each subset of items
- The subproblem then will be to compute B[k,w] which is the maximum benefit for total capacity w and items {1, 2, ..k}

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & w_k \le w \end{cases}$$

The best subset of S_k that has the total weight w, either contains item k or not.

- **First case**: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be > w. So we select the "optimal" using items 1,.., k-1
- **Second case**: $w_k \le w$. Then the item k can be in the solution, and we choose the case with greater value

Recursive Formula for subproblems

$$B[k,w] = \begin{cases} \boxed{B[k-1,w]} & \text{if } w_k > w \\ \max\{\boxed{B[k-1,w]}, \boxed{B[k-1,w-w_k]+b_k}\} & \text{if } w_k \le w \end{cases}$$

It means, that the best subset of S_k that has total weight w is one of the two:

Item k is too big to fit in the knapsack with capacity w

Do not use item k: the best subset of S_{k-1} that has total weight w, **or**

Use item k: the best subset of S_{k-1} that has total weight w-w_k plus the item k *with benefit* b_k

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0      // 0 item's
for i = 0 to n
    B[i,0] = 0       // 0 weight
    for w = 1 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w item i is too big
```

Running time

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 1 to W
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(nW) pseudo-polynomial

Remember that the brute-force algorithm takes $O(n2^n)$.
Better than Brute force if $W << 2^n$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):

S = {(2,3), (3,4), (4,5), (5,6) }

w \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

for w = 0 to W

$B[0,w] = 0$

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

for i = 0 to n

    B[i,0] = 0

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0				
3	0				
4	0				
5	0				

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=1$
$b_i=3$
$w_i=2$
$w=1$
$w-w_i=-1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Item : (w, b)

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w-w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	0				

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w-w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	0				

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w-w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	**3**			

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution

if $b_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = b_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3 →	**3**		
3	0	3			
4	0	3			
5	0	3			

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Item : (w, b)

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]   // w_i > w
```

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w-w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	**7**		

Item : (w, b)

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $\mathbf{B[i,w] = b_i + B[i-1,w- w_i]}$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →	**0**	
2	0	3	3 →	**3**	
3	0	3	4 →	**4**	
4	0	3	4		
5	0	3	7		

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=3$
$b_i=5$
$w_i=4$
$w=1..3$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w- w_i=0$

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

if $w_i <= w$ // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	**5**
5	0	3	7	7	

$i=4$

$b_i=6$

$w_i=5$

$w=1..4$

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Item : (w, b)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	7

i=4

b_i=6

w_i=5

w=1..4

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]   // w_i > w
```

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- See LCS algorithm for the example how to extract this data from the table we built using “parent pointers”.

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time (Dynamic Programming algorithm vs. naïve algorithm):
  - LCS: **$O(mn)$** vs. **$O(n\ 2^m)$**
  - 0-1 Knapsack problem: **$O(Wn)$** vs. **$O(n2^n)$**

Pseudo-polynomial