CSEP 521 - Applied Algorithms

Dynamic Programming

Reading:

Skiena, chapter 3

CLRS: chapter 16 (1st ed.)

chapter 15 (2nd ed.)

Dynamic Programming

- A strategy for designing algorithms.
- A technique, not an algorithm.
- The word "programming" is historical and predates computer programming.
- Use when problem breaks down into recurring small subproblems.

Dynamic Programming Example: Longest Common Subsequence

- *Longest common subsequence* (*LCS*) problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  - Example: x = {A B C B D A B }, y = {B D C A B A}
  - {B C} and {A A} are both subsequences of both
    - *What is the LCS?* BCAB, BCBA
  - Brute-force algorithm: For every subsequence of x, check if it's a subsequence of y
    - *How many subsequences of x are there?*
    - *What will be the running time of the brute-force alg?*

LCS Algorithm

- Brute-force algorithm: 2^m subsequences of x each takes O(n) to search in y: $O(n\,2^m)$
- We can do better: for now, let's only worry about the problem of finding the *length* of LCS
  - When finished we will see how to backtrack from this solution back to the actual LCS
- Notice LCS problem has optimal substructure
  - Subproblems: LCS of pairs of *prefixes* of x and y

Finding LCS Length

- Define c[i,j] to be the length of the LCS of X_i =x[1..i] and Y_j=y[1..j]
  - *What is the length of LCS of x and y?*
    C[m,n]
- Theorem:

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- *What is this really saying?*

LCS recursive solution

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- When we calculate c[i,j], we consider two cases:
- **First case:** x[i]=y[j]: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1.

LCS recursive solution

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- **Second case:** x[i] != y[j]
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

LCS recursive solution

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

Answer: Let x=abc y=db

c[3,2]=max(c(3,1), c(2,2))=max(0,1)=1

c[3,2]$\neq$ c(2,1)=0

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define c[i,j] to be the length of LCS of X_i and Y_j
- The length of LCS of X and Y is *c[m,n]*

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

LCS Algorithm

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- We start with i = j = 0 (empty substrings of X and Y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. c[0,0] = 0)
- LCS of empty string and any other string is empty, so for every i and j: c[0, j] = c[i,0] = 0
- We can now calculate for all j c[1,j], c[2,j], etc.

LCS Length Algorithm

LCS-Length(X, Y)

1. m = length(X) // # of symbols in X
2. n = length(Y) // # of symbols in Y
3. for i = 1 to m c[i,0] = 0 // special case: Y_0
4. for j = 1 to n c[0,j] = 0 // special case: X_0
5. for i = 1 to m // for all X_i
6. for j = 1 to n // for all Y_j
7. if (X_i == Y_j)
8. c[i,j] = c[i-1,j-1] + 1
9. else c[i,j] = max(c[i-1,j], c[i,j-1])
10. return c

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB
X = A **B** **C** **B**
Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

i	Xi \ j	0 Yj	1 B	2 D	3 C	4 A	5 B
0	Xi						
1	A						
2	B						
3	C						
4	B						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

LCS Example (1)

ABCB
BDCAB

i	Xi \ j	0 Yj	1 B	2 D	3 C	4 A	5 B
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**					
2	B	**0**					
3	C	**0**					
4	B	**0**					

for i = 1 to m   c[i,0] = 0
for j = 1 to n   c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

i	Xi \ j	0 Yj	**1** B	2 D	3 C	4 A	5 B
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**	**0**				
2	B	**0**					
3	C	**0**					
4	B	**0**					

i=1
j=1

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB
BDCAB

i	Xi \ j	0 Yj	1 B	2 D	3 C	4 A	5 B
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**	**0**	**0**	**0**		
2	B	**0**					
3	C	**0**					
4	B	**0**					

i=1
j=2,3

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

i=1
j=4

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

i=1
j=5

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**				
3	**C**	**0**					
4	**B**	**0**					

i=2
j=1

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

i=2
j=2,3,4

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

i		j=0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

i=2
j=5

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (9)

ABCB
BDCAB

i		j=0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

i=3
j=1,2

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

i		j=0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

i=3
j=3

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB
BDCAB

i		j=0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

i=3
j=4,5

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

i=4
j=1

LCS Example (13)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

i=4
j=2,3,4

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

i=4
j=5

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(mn)

since each c[i,j] is calculated in constant time, and there are mn elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output the Longest Common Subsequence of X and Y

Each c[i,j] depends on c[i-1,j] and c[i,j-1] or c[i-1, j-1]

For each c[i,j] we can say how it was acquired:

2	2
2	3

For example, here c[i,j] = c[i-1,j-1] +1 = 2+1=3

How to find actual LCS - continued

- Remember that

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- So we can start from c[m,n] and go backwards
- Whenever c[i,j] = c[i-1, j-1]+1, remember x[i] (because x[i] is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order.

Finding LCS

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Finding LCS (2)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**
(this string turned out to be a palindrome)

Dynamic programming- summary of LCM example:

- We were able to use DP since the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*).
- We were able to find solutions to subproblems and to store them in memory for later use
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again.

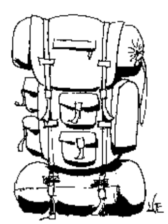
Properties of a problem that can be solved with dynamic programming

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

The Knapsack problem

- You are about to go to a camp.
- There are many items you want to pack.
- You have one knapsack. The total weight you can carry is some fixed number W.
- Every item in your list has has some weight, w_i, and some value, b_i, that measures how much you really need it.
- You need to pack the knapsack in a way that maximizes the total value of the packed items.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5
4	4	6



Can we use Dynamic Programming?

- Does the solution of the problem includes solutions to subproblems?
- Can we find a recursive formula for the solution?
- Can we recursively solve subproblems, starting from the trivial case, and save their solutions in memory?
- Does it mean that at the end we'll get the solution of the whole problem?

The Knapsack problem:

Max weight:
W = 20

Items	Weight w_i	Benefit b_i
	2	3
	3	4
	4	5
	5	8
	9	10

The Knapsack problem: brute-force approach

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 $O(2^n)$ – not acceptable

The Knapsack problem

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

Let's try this:
If items are labeled *1..n*, then a subproblem would be to find an optimal solution for
S_k = {items labeled 1, 2, .. k}

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 next question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that.
- Here is why….

Defining a Subproblem

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

- Max weight: W = 20
- Best solution for S_1 is {1}
  weight=2, benefit =1
- Best solution for S_2 is {1,2}
  weight=5, benefit =7
- Best solution for S_3 is {1,2,3}
  weight=9, benefit =12
- Best solution for S_4 is {1,2,3,4}
  weight=14, benefit =20
- Best solution for S_5 is {1,3,4,5}
  weight=20, benefit =26

Best solution for S_4 is not part of the best solution for S_5!!!

Defining a Subproblem (cont.)

- 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 weight for each subset of items
- The subproblem then will be to compute *B[k,w]* – the best value of a subset of S_k with total weight at most *w*.

Recursive Formula for subproblems

- Recursive formula for subproblems:

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

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

1) the best subset of S_{k-1} that has total weight *w*, **or**

2) the best subset of S_{k-1} that has total weight $w-w_k$ plus the item *k*

Recursive Formula

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

- 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*, which is unacceptable
- Second case: $w_k \leq w$. Then the item *k* <u>can</u> be in the solution, and we choose the case with greater value

Knapsack Algorithm

```
for w = 0 to W
  B[0,w] = 0
for i = 0 to n
  B[i,0] = 0
  for w = 0 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
```


Running time

```
for w = 0 to W          O(W)
  B[0,w] = 0
for i = 0 to n            Repeat n times
  B[i,0] = 0
  for w = 0 to W          O(W)
     < the rest of the code >
```

What is the running time of this algorithm?

$O(n*W)$

Remember that the brute-force algorithm takes $O(2^n)$

Example (1)

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

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
```

Example (3)

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

Example (4)

Items:
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	0				
4	0				
5	0				

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

Example (5)

Items:
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

Example (6)

Items:
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

Example (7)

Items:

$\boxed{1: (2,3)}$

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				

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

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

Example (8)

Items:

$\boxed{1: (2,3)}$

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

Example (9)

Items:

$\boxed{\begin{array}{l}1: (2,3)\\ 2: (3,4)\end{array}}$

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	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

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

Example (10)

Items:

$\boxed{\begin{array}{l}1: (2,3)\\ 2: (3,4)\end{array}}$

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	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

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

Example (11)

Items:
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

Example (12)

Items:
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	**4**		
5	0	3			

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

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

Example (13)

Items:
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	4		
5	0	3	**7**		

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

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]
    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

Example (14)

Items:
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	**0**	
2	0	3	3	**3**	
3	0	3	4	**4**	
4	0	3	4		
5	0	3	7		

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

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

Example (15)

Items:
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	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

$i=3$
$b_i=5$
$w_i=4$
$w=4$
$w- w_i=0$

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

Example (15)

Items:
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	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	**7**	

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

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

Example (16)

Items:
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	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	

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

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

Example (17)

Items:
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	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**

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

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
- Similar to what we did in the LCS algorithm, this can be done by keeping some more information while building the table.

All-pair Shortest Path

- Input: a directed graph $G=(V,E)$ with $V = \{1, 2, ..., n\}$. The length of edge e is denoted by c(e), and it may be negative.
- Output: All-pair shortest path: for any two vertices v,u in V, what is the shortest path from v to u
  - we will only be interested in the *length* of that path.

All-pair Shortest Path

- We can solve this problem using single-source shortest path algorithms. For example, we can run Bellman-Ford |V| times (one time for each possible selection of the source vertex s).
- Time complexity: $|V| * O(|V||E|) = O(|V|^2|E|)$
- We will see a solution using Dynamic Programming.

All-pair Shortest Path

Define

$$\delta^0(i,j) = \begin{cases} c(e) & \text{if } i \xrightarrow{e} j, \\ \infty & \text{if there is no edge from } i \text{ to } j. \end{cases}$$

Let $\delta^k(i,j)$ be the length of a shortest path from i to j among all paths which may pass through vertices $1,2,\dots,k$ but do not pass through vertices $k+1, k+2,\dots,n$.

Floyd Algorithm (1962)

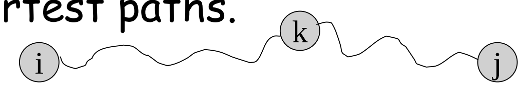
1. Init $\delta^0(i, j)$ as defined earlier
2. $k \leftarrow 1$
3. For every $1 \leq i, j \leq n$ compute
   $\delta^k(i, j) \leftarrow \text{Min}\{\delta^{k-1}(i, j),$
   $\delta^{k-1}(i, k) + \delta^{k-1}(k, j)\}.$
4. If $k = n$, stop. If not, increment k and go to step 3.

Floyd Algorithm

$$\delta^k(i, j) \leftarrow \text{Min}\{\delta^{k-1}(i, j),\ \delta^{k-1}(i, k) + \delta^{k-1}(k, j)\}.$$

The shortest path from i to j which may pass through vertices 1,2,...,*k* but do not pass through vertices *k+1, k+2,...,n:*

1. Might not pass through vertex k, or
2. Might pass through k, and then it is composed by two already-computed shortest paths.



Floyd Algorithm

$$\delta^k(i, j) \leftarrow \text{Min}\{\delta^{k-1}(i, j),\ \delta^{k-1}(i, k) + \delta^{k-1}(k, j)\}.$$

Theorem: The value of $\delta^n(i, j)$ is the shortest path from i to j

Proof idea: By induction on k, the value of $\delta^k(i, j)$ is correct. In particular, for $\delta^n(i, j)$ we get the shortest path.

Floyd Algorithm

- The value of $\delta^n(i, j)$ is meaningful only if there are no negative cycles in *G*.
- The existence of negative cycles is detected by having $\delta^k(i, i) < 0$ for some i and k.
- Each application of step 3 requires n^2 operations, and step 3 is repeated n times. Thus, the algorithm is of complexity $O(n^3)$.

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. naive algorithm):
  - LCS: **$O(m*n)$** vs. **$O(n*2^m)$**
  - Knapsack problem: **$O(W*n)$** vs. **$O(2^n)$**
- This is called 'pseodo-polynomial'