COMP8620
Lecture 8

Dynamic Programming

Dynamic Programming

Principle of optimality

- If an intermediate state is visited in an optimal sequence of decisions, then the decisions leading to that state must also be optimal.

Bellman's equation

$$f_{n+1}(x) = \min_{u \in S(n)} \{ f_n(u) + c(u,x) \}$$

- to reach optimally *state x* in *stage n+*1 it suffices to consider those policies that reach optimally each possible *state u* in previous *stage n*)

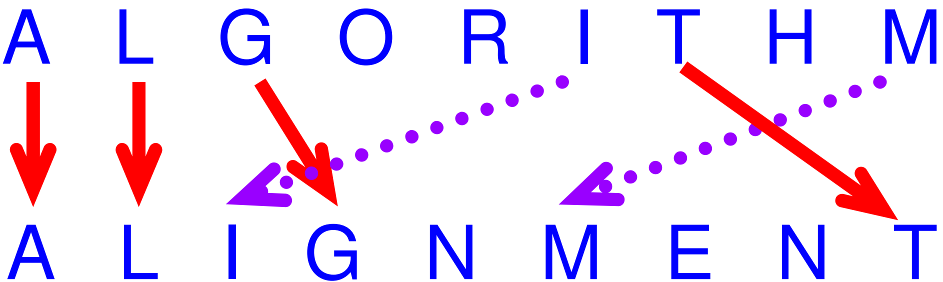
(Bellman,1960)

Properties of DP

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

Longest Common Subsequence (LCS)

- Problem: Find the longest pattern of characters that is common to two text strings X and Y
- Subproblem: find the LCS of *prefixes* of X and Y.
- *Optimal substructure*: LCS of two prefixes is always a part of LCS of bigger strings



Longest Common Subsequence (LCS)

- Define X_i, Y_j to be prefixes of X and Y of length i and j; m = |X|, n = |Y|
- We store the length of LCS(X_i, Y_j) in c[i,j]
- Trivial cases: LCS(X_0 , Y_j) and LCS(X_i, Y_0) is empty (so c[0,j] = c[i,0] = 0)
- Recursive formula for c[i,j]:

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

c[m,n] is the final solution

Longest Common Subsequence (LCS)

- Need separate data structure to retrieve answer

- Algorithm runs in $O(m*n)$,
- *cf* Brute-force algorithm: $O(n\ 2^m)$

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 , b_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

This is a knapsack
Max weight: W = 20

W = 20

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

0-1 Knapsack problem

$$\max \sum_{i \in T} x_i b_i$$

$$\text{subject to} \sum_{i \in T} x_i w_i \leq W_i$$

$$x_i \in \{0,1\}$$

- 0-1 problem: each item is entirely accepted or rejected.

0-1 Knapsack problem: Brute force

- n items: 2^n possible combinations of items.
- Go through all combinations, find the one with the most total value and with total weight less or equal to W
- Running time: $O(2^n)$

Dynamic Programming

- Consider DP

- What subproblem?

Defining a Subproblem

- Subproblem(?): find an optimal solution for S_k = *{items labeled 1, 2, .. k}* within weight 20

- Either
  - Use item *k* with one of the S_i for *i < k or*
  - Don't use *k* and use the best S_i so far

$$S_k = \max_{i=1..k-1}\{S_{i-1} + b_k \mid W_{i-1} + w_k < 20\} \; or$$

$$\max_{i=1..k-1}\{S_{i-1}\}$$

$$W_k = W_{\arg\max} + w_k \; or \; W_{\arg\max}$$

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1		
2		
3		
4		
5		

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1	2	3
2		
3		
4		
5		

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1	2	3
2	5	7
3		
4		
5		

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1	2	3
2	5	7
3	9	12
4		
5		

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1	2	3
2	5	7
3	9	12
4	14	20
5		

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1	2	3
2	5	7
3	9	12
4	14	20
5		

Defining a subproblem

k	b_k	w_k
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

k	S_k	W_k
1	2	3
2	5	7
3	9	12
4	14	20
5	14	17

But sol 1,3,4,5

$\Sigma w = 20$

$\Sigma b = 26$!

What went wrong?

- Wrong subproblem

- The best way to pick *k-1* items with a budget of 20 does NOT necessarily chose the best way to choose the next item

Subproblem Mk II

- State includes how much weight I have used so far
- Consider filling the knapsack to weight *w* with up to *k* items
- Either I choose item *k* or not
- If I choose item *k*, I need to have filled my knapsack up to weight $w\text{-}w_k$ optimally, and then add item *k*, OR
- I fill my knapsack up to weigh *w* and skip item *k*

Recursive Formula for subproblems

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

The best way to pack up to item *k* using at most weight *w* is either

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

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

0-1 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 sol
              if b_i + B[i-1,w-w_i] > B[i-1,w]
                    B[i,w] = b_i + B[i-1,w- w_i] // use i
              else
                    B[i,w] = B[i-1,w] // don't use i
        else
              B[i,w] = B[i-1,w]  // w_i > w
```

Running time

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

$O(W)$

Repeat n times

$O(W)$

Running time: O(n*W)

Brute-force algorithm: $O(2^n)$

Example

Data:

n = 4 (# of elements)
W = 5 (max weight)

Item	w_i	b_i
1	2	3
2	3	4
3	4	5
4	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 $w_i \le w$ // 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 $B[i,w] = B[i-1,w]$ // $w_i > w$

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:

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:

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:

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	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:

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	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

```
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
```

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 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
```

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 $w_i \le w$ // 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)

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		

Items:

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 $w_i <= w$ // 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 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
```

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	0	3	7	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 (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	

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

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**

$i=3$

$b_i=5$

$w_i=4$

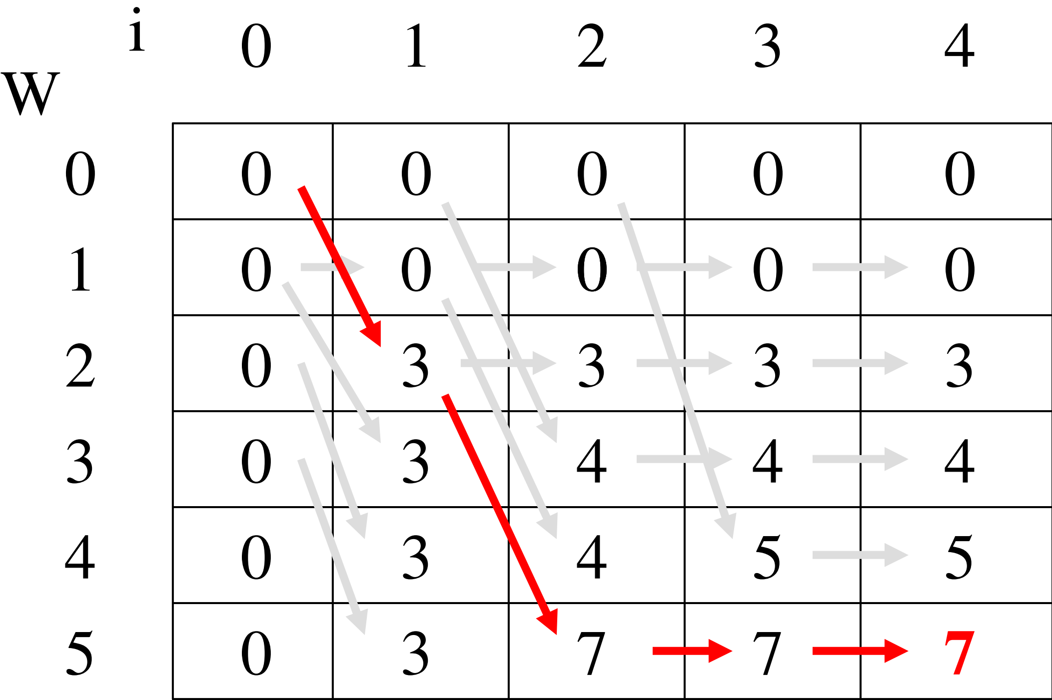
$w=5$

```
if 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
```

Solution

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**

Solution



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**

Dynamic Programming

Three basic components:

- The recurrence relation (for defining the value of an optimal solution);
- The tabular computation (for computing the value of an optimal solution);
- The traceback (for delivering an optimal solution).

- Useful for solving certain types of problem

Exercise

Do it yourself:

- Find the edit distance between two strings
  - Cost of adding a letter: 1
  - Cost of deleting a letter: 1
  - Cost of modifying a letter: 1
  - Cost of transposing 2 letters: 1
- What is state?
- What is the recursion?
- Try on DYNAMIC → DNAMCI

Idea

- Either
  - Copy a letter (cost 0)
  - Add a letter (cost 1)
  - Delete a letter (cost 1)
  - Modify a letter (cost 1)
  - Transpose 2 letters
- S(i,j) = edit cost of turning X[0..i] to Y[0..j]

Recursion

$$S(i,j) = \min \begin{cases} S(i-1, j-1) & \text{if } x_i = y_j \\ S(i-1, j) + 1 \\ S(i, j-1) + 1 \\ S(i-1, j-1) + 1 \\ S(i-2, j-2) + 1 & \text{if } x_{i-1} = y_j \ \&\ x_i = y_{j-1} \end{cases}$$

```
1. public static int editDist (String a, String b)
2. {
3.     int n = a.length() + 1;
4.     int m = b.length() + 1;
5.     int[][] dist = new int [n][m];
6.
7.     for (int i = 0; i < n; i++)
8.         dist[i][0] = i;
9.     for (int j = 0; j < m; j++)
10.        dist[0][j] = j;
```

```
11.       for (int i = 1; i < n; i++) {
12.           for (int j = 1; j < m; j++) {
13.               // Assume modify
14.               int best = dist[i-1][j-1] + 1;
15.               if (a.charAt(i-1) == b.charAt(j-1))
16.                   best = dist[i-1][j-1];
17.               if (dist[i-1][j] + 1 < best)
18.                   best = dist[i-1][j] + 1; // Add
19.               if (dist[i][j-1] + 1 < best)
20.                   best = dist[i][j-1] + 1; // Delete
21.               if (
22.                   i > 1 && j > 1 &&
23.                   a.charAt(i-1) == b.charAt(j-2) &&
24.                   a.charAt(i-2) == b.charAt(j-1) &&
25.                   dist[i-2][j-2] + 1 < best
26.               )
27.                   best = dist[i-2][j-2] + 1; // Transpose
28.               dist[i][j] = best;
29.           }
30.       }
31.       return dist[n-1][m-1];
32.   }
```

Running it

```
$ java -cp . EditDist dynamic dnamci
   0  1  2  3  4  5  6
   1  0  1  2  3  4  5
   2  1  1  2  3  4  5
   3  2  1  2  3  4  5
   4  3  2  1  2  3  4
   5  4  3  2  1  2  3
   6  5  4  3  2  2  2
   7  6  5  4  3  2  2
dynamic dnamci 2
```

References

- Short section (Section 2.4) in “Search Methodologies”, E.K. Burke and G Kendall (Eds), Springer 2005
- In depth in “Combinatorial Data Analysis : Optimization by Dynamic Programming”, L.J. Hubert, P Arable and J. Meulman, SIAM, 2001 (electronic book)

- Next week:
  Constraint Programming (Jason Li)