Lecture 9

Dynamic Programming Methods for

1. Longest Common Subsequence
2. Shortest Common Supersequence
3. 0-1 Knapsack Problem.

What we learned so far:

1. Euler Circuit Problem. Theorem and algorithm
2. Greedy Algorithms
   1. Interval scheduling and interval partitioning problems. Algorithms, correctness, running time.
   2. MST: Kruskal and Prim Algorithms, theorem, running times with different data structures.
   3. Single-source shortest path: Dijkstra algorithm: algorithm, running time and correctness.
3. Divide and Conquer: divide, recur, and conquer
   1. Merge sort: algorithm and running time
   2. Counting Inversions: algorithms and running: two jobs are easier than one.

Dynamic Programming

* Break problems into subproblems and combine their solutions into solutions to larger problems.

In contrast to divide-and-conquer, dynamic programming uses memorization: each sub-problem is solved only once and the result of each sub-problem is stored in a table

1. Longest common subsequence (LCS)

A string : A = b a c a d

A subsequence of A: deleting zero or more symbols from A (not necessarily consecutive)

e.g. ad, ac, bac, acad, bacad, bcd.

Common subsequences of A = b a c a d and B = a c c b a d c b:

ad, ac, bac, acad.

The longest common subsequence (LCS) of A and B: acad.

Longest common subsequence problem

Input: Two sequences $X=x_1x_2...x_m$, and $Y=y_1y_2...y_n$.

Output: a longest common subsequence of X and Y.

A brute-force approach:

Suppose that $m \geq n$. Try all subsequence of X (There are 2^m subsequence of X), test if such a subsequence is also a subsequence of Y, and select the one with the longest length.

Charactering a longest common subsequence

Theorem (Optimal substructure of an LCS)

Let $X=x_1x_2...x_i$, and $Y=y_1y_2...y_j$ be two sequences, and Z=be a LCS of X and Y.

1. If $x_i=y_j$, $Z=LCS(x_1x_2...x_{i-1}, y_1y_2...y_{j-1}) + x_i$

2. If $x_i \neq y_j$, Z is the longer one among $LCS(x_1x_2...x_{i-1}, y_1y_2...y_j)$ and $LCS(x_1x_2...x_i, y_1y_2...y_{j-1})$.

The recursive equation

Let c[i,j] be the length of an LCS of X[1...i] and Y[1...j].
c[i,j] can be computed as follows:

$$c[i,j]=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j, \\ \max\{c[i,j-1],c[i-1,j]\} & \text{if } i,j>0 \text{ and } x_i\neq y_j. \end{cases}$$

Computing the length of an LCS

There are $n \times m$ c[i,j]'s. So we can compute them in a specific order.

The algorithm to compute an LCS

```
for i=1 to m do
        c[i,0]=0;
for j=0 to n do
        c[0,j]=0;
for i=1 to m  do
     for j=1 to n do
     {
           if x[i] ==y[j]  then
                c[i,j]=c[i-1,j-1]+1;
                b[i,j]=1;
           else   if c[i-1,j]>=c[i,j-1] then
                          c[i,j]=c[i-1,j]
                          b[i,j]=2;
                   else c[i,j]=c[i,j-1]
                          b[i,j]=3;
     }
```

Example 3: X=BDCABA and Y=ABCBDAB.

	y_i	B	D	C	A	B	A
x_i							
A							
B							
C							
B							
D							
A							
B							

Example 3: X=BDCABA and Y=ABCBDAB.

	y_i	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D	0	↑ 1	↖ 2	↑ 2	2	↑ 3	↑ 3
A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	4 ←

Constructing an LCS (back-tracking)

We can find an LCS using b[i,j]'s.
We start with b[n,m] and track back to some cell b[0,i] or b[i,0].
The algorithm to construct an LCS (backtracking)

```
1. i=m
2. j=n;
3. if i==0 or j==0 then exit;
4. if b[i, j]==1 then
    {
          i=i-1;
          j=j-1;
          print "x_i" ;
    }
5.  if b[i, j]==2            i=i-1
6.  if b[i, j]==3            j=j-1
7. Goto Step 3.
```

The time complexity: $O(nm)$.

2. Shortest common super-sequence

Shortest common supersequence

Definition: Let X and Y be two sequences. A sequence Z is a **supersequence** of X and Y if both X and Y are subsequences of Z.

Shortest common supersequence problem:

Input: Two sequences X and Y.

Output: a shortest common supersequence of X and Y.

Example: X=**abc** and Y=**abb**. Both **abbc** and **abcb** are the shortest common supersequences for X and Y.

Example 1:

X=humans, Y=chimpanzees

SCS(X, Y)=SCS(human, chimpanzee)+s

Example 2:

X=human, Y=chimpanzee

SCS(X, Y)=shorter(

SCS(huma, chimpanzee)+n,

SCS(human, chimpanze)+e)

Exmple 3:

X=human, Y=“”

SCS(X, Y)=human

Recursive Equation:

Let c[i,j] be the length of an SCS of X[1...i] and Y[1...j].

c[i,j] can be computed as follows:

$$c[i,j] = \begin{cases} j & \text{if } i=0 \\ i & \text{if } j=0, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j, \\ \min\{c[i,j-1]+1, c[i-1,j]+1\} & \text{if } i,j>0 \text{ and } x_i \neq y_j. \end{cases}$$

	y_i	B	D	**C**	A	**B**	**A**
x_i	0	1	2	3	4	5	6
A	1	←2	←3	←4	↖4	←5	↖6
B	2	↖2	←3	←4	←5	↖5	←6
C	3	↑3	←4	↖4	←5	←6	←7
B	4	↖4	←5	↑5	←6	↖6	←7
D	5	↑5	↖5	←6	←7	↑7	←8
A	6	↑6	↑6	←7	↖7	←8	↖8
B	7	↖7	↑7	←8	↑8	↖8	↑9

The pseudo-codes

```
for i=0 to n do
   c[i, 0]=i;
for j=0 to m do
    c[0,j]=j;
for i=1 to n do
   for j=1 to m do
       if (xi == yj)  c[i ,j]= c[i-1, j-1]+1;  b[i.j]=1;
       else {
               c[i,j]=min{c[i-1,j]+1, c[i,j-1]+1}.
                if (c[i,j]=c[i-1,j]+1 then b[i,j]=2;
                else b[i,j]=3;
               }
 p=n, q=m;  / backtracking
 while (p≠0 or q≠0)
    { if (b[p,q]==1) then {print x[p]; p=p-1; q=q-1}
       if (b[p,q]==2) then {print x[p]; p=p-1}
       if (b[p,q]==3) then {print y[q]; q=q-1}
    }
```

Example SCS for X=BDCABA and Y=ABCBDAB.

	y_i	B	D	C	A	B	A
x_i	0	1	2	3	4	5	6
A							
B							
C							
B							
D							
A							
B							

Example SCS for X=BDCABA and Y=ABCBDAB.

	y_i	B	D	C	A	B	A
x_i	0	1	2	3	4	5	6
A	1	← 2	← 3	← 4	↖ 4	← 5	↖ 6
B	2	↖ 2	← 3	← 4	← 5	↖ 5	← 6
C	3	↑ 3	← 4	↖ 4	← 5	← 6	← 7
B	4	↖ 4	← 5	↑ 5	← 6	↖ 6	← 7
D	5	↑ 5	↖ 5	↑ 6	← 7	↑ 7	← 8
A	6	↑ 6	↑ 6	← 6	↖ 7	← 8	↖ 8
B	7	↖ 7	↑ 7	↑ 7	↑ 8	↖ 8	← 9

Dynamic Programming

Step 1: Structure and subproblem definition:

- find a relation between the original problem and the smaller problems.

Step 2: Bottom-up computation: Compute an optimal solution in a bottom-up fashion by using a table structure.

Step 3: Construction of optimal solution: Construct an optimal solution from computed information.

3. 0-1 Knapsack Problem

Knapsack Problem *0-1 version*

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w), \quad v_i + OPT(i-1, w - w_i) \,\} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
φ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

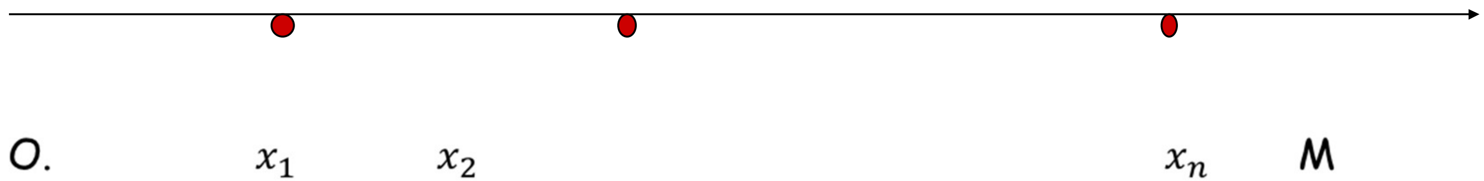
Chanllenge Q1: Billboard placement (Not Required)

You are trying to decide where to place billboards on a highway that goes East-West for M miles. The possible sites for billboards are given by numbers $x_1, \ldots, x_n$ each in the interval [0, M]. If you place a billboard at location x_i, you get a revenue r_i.

You have to follow a regulation: no two of the billboards can be within less than or equal to 5 miles of each other.

You want to place billboards at a subset of the sites so that you maximize your revenue subject to this constraint.

How?



Solution:

Two sites are compatible if their distance are larger than 5km.

Def: P(i) is the largest index such that P(i)<i and sites i and P(i) are compatible. If no such index, define P(i)=0.

Let M(i) be the optimal revenue if only the first i sites x_1, …, x_i are considered.

$$M(0)=0,$$

$$M(i) = \max\{M(i-1), r_i + M(P(i))\} \quad for\ i = 1, \dots, n$$

Dynamic Program Method:

M(0)=0

For i=1 to n, do

If $M(i-1) > r_i + M(P(i))$

B(i)=0, M(i)=M(i-1)

Otherwise

B(i)=1, $M(i) = r_i + M(P(i))$

Backtracking:

```
p=n
While{p isn't 0} do
{If B(p)=1,
 select site p, and  p=P(n),
 Otherwise,
 p=p-1,
}
```