Dynamic Programming II

Longest Common Subsequence
& Edit Distance
& Longest Increasing Subsequence

Application 1: Longest Common Subsequence

Subsequence

Given two sequences of characters $X=<x_1 x_2 .. x_n>$ and $Z=<z_1 z_2 .. z_k>$, Z is called a subsequence of X if there is a strictly increasing sequence of k indices ($1 \leqq i_1 < i_2 < .. < i_k \leqq n$) such that

$$z_j = x_{i_j}$$

(e.g. X=<ABRACADABRA> and Z=<AADAA> Z is a subsequence of X)

Longest Common Subsequence (LCS)

Problem: Given two strings

$X = \langle x_1\, x_2 \,..\, x_m \rangle$ and $Y = \langle y_1\, y_2 \,..\, y_n \rangle$,

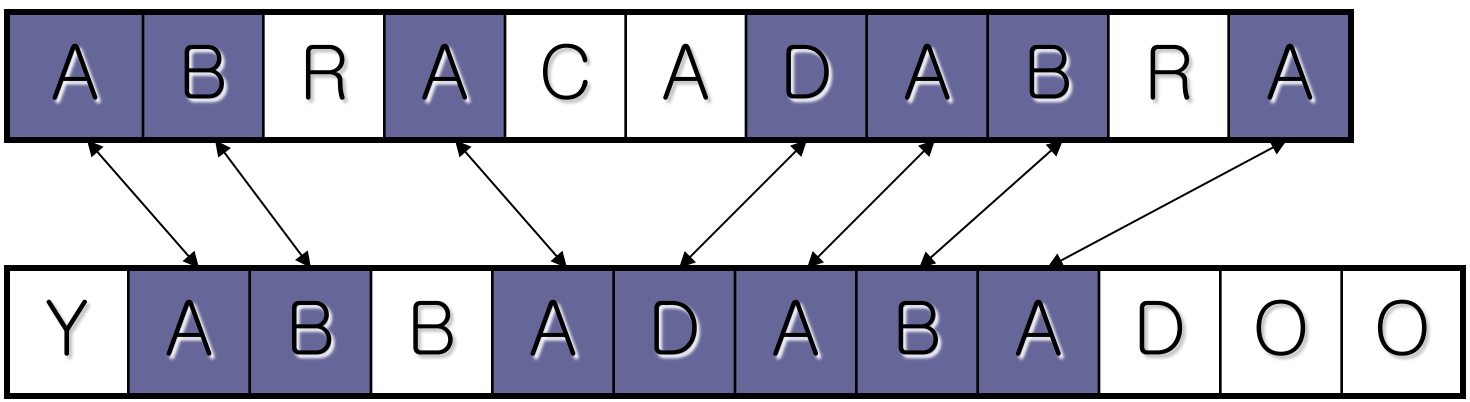
determine a longest common subsequence Z such that

Z is longest subsequence of

X and Y

Example

- X=<ABRACADABRA>
  Y=<YABBADABADOO>



LCS =

A	B	A	D	A	B	A

Brute-force algorithm

- For every subsequence of X, check whether it's a subsequence of Y
  - 2^m subsequences of X to check
  - Each subsequence takes $\Theta(n)$ time to check; scan Y for first letter, for second..

  => $\Theta(n\ 2^m)$ time

Optimal Substructure

- Notation:
  - $X_i = <x_1\ x_2\ ..\ x_i>$
  - $Y_j = <y_1\ y_2\ ..\ y_j>$
- Observation:
  - What happens if $x_i = y_j$?
  - What happens if $x_i \neq y_j$?

Theorem

Let $Z = <z_1\ z_2\ ..\ z_k>$ be LCS of X_i and Y_j

- $x_i = y_j$:
  - $z_k = x_i = y_j$
  - Z_{k-1} is an LCS of X_{i-1} and Y_{j-1}
- $x_i \neq y_i$:
  - $z_k \neq x_i$: Z is an LCS of X_{i-1} and Y_j
  - $z_k \neq y_j$: Z is an LCS of X_i and Y_{j-1}

Recursive Formulation

Let $c[i,j]$ be the length of LCS of X_i and Y_j

- $i=0, j=0 : c[i,j] = 0$
- $i, j > 0$ and $x_i = y_j$

  $c[i,j] = c[i-1][j-1]+1$
- $i, j > 0$ and $x_i \neq y_j$

  $c[i,j] = \max(c[i-1][j], c[i][j-1])$

```
LCS-length(X,Y,m,n)
for i=1.. m
    c[i][0]=0
for j=1.. n
    c[0][j]=0
for i=1..m
for j=1..n
   if x_i = y_j
     c[i][j]= c[i-1][j-1]+1
     b[i][j]= NW
   else if (c[i-1][j]>=
              c[i][j-1])
      c[i][j]= c[i-1][j]
      b[i][j]= N
   else
      c[i][j]=c[i][j-1]
      b[i][j]= W
```

```
Print-LCS(b,X,i,j)
if i=0 or j=0
   return
if b[i][j]=NW
   Print-LCS(b,X,i-1,j-1)
   print x_i
else if b[i][j]=N
   Print-LCS(b,X,i-1,j)
else
   Print-LCS(b,X,i,j-1)
```

Initial call is

`Print-LCS(b,X,m,n)`

		a	m	p	u	t	a	t	i	o	n
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4

p a i n

Application 2:
Edit Distance

Edit Operations

- Change one letter
  - computer => commuter
- Delete one letter
  - sport => sort
- Insert one letter

  sort => sport

Edit Distance

- Edit distance of two strings X and Y
  = minimum number of edit operations
    required to change X to Y

Recursive Formulation

- $d(X_0, Y_0) = 0$
- $d(X_i, Y_0) = i$ and $d(X_0, Y_j) = j$
- $d(X_i, Y_j) =$ minimum of
  - $d(X_{i-1}, Y_{j-1}) + c$
    - if $x_i = y_j$ then $c=0$
    - else $c=1$ (edit X_{i-1} to Y_{j-1} and change x_i to y_j)
  - $d(X_i, Y_{j-1}) + 1$ (edit X_i to Y_{j-1} and insert y_j)
  - $d(X_{i-1}, Y_j) + 1$ (delete x_i and edit X_{i-1} to Y_j)

```
EditDistance(X,Y,m,n)
d[0][0]= 0
for i=1.. m
    d[i][0]=i
for j=1.. n
    d[0][j]=j
for i=1..m
for j=1..n
    val = (x_i == y_j) ? 0 : 1
    d[i][j] = min { d[i-1][j-1]+val,
                    d[i-1][j]+1,
                    d[i][j-1]+1 }
```

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0					
M	1					
A	2					
T	3					
H	4					
S	5					

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0				
M	1					
A	2					
T	3					
H	4					
S	5					

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1			
M	1					
A	2					
T	3					
H	4					
S	5					

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2		
M	1					
A	2					
T	3					
H	4					
S	5					

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1				
A	2	↑ 2				
T	3	↑ 3				
H	4	↑ 4				
S	5	↑ 5				

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1	↖ 1			
A	2	↑ 2				
T	3	↑ 3				
H	4	↑ 4				
S	5	↑ 5				

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1	↖ 1	↖← 2		
A	2	↑ 2				
T	3	↑ 3				
H	4	↑ 4				
S	5	↑ 5				

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1	↖ 1	↖← 2	↖← 3	↖← 4
A	2	↑ 2	↖ 1	↖← 2	↖← 3	↖← 4
T	3	↑ 3				
H	4	↑ 4				
S	5	↑ 5				

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	←1	←2	←3	←4
M	1	↑1	↖1	↖←2	↖←3	↖←4
A	2	↑2	↖1	↖←2	↖←3	↖←4
T	3	↑3	↑2	↖2	↖2	←3
H	4	↑4				
S	5	↑5				

Computation of d(X,Y)

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1	↖ 1	↖← 2	↖← 3	↖← 4
A	2	↑ 2	↖ 1	↖← 2	↖← 3	↖← 4
T	3	↑ 3	↑ 2	↖ 2	↖ 2	← 3
H	4	↑ 4	↑ 3	↖↑ 3	↖↑ 3	↖ 3
S	5	↑ 5	↑ 4	↖↑ 4	↖↑ 4	↖ 3

The traceback

Y			A	R	T	S
X		0	1	2	3	4
	0	0	←1	←2	←3	←4
M	1	↑1	↖1	↖←2	↖←3	↖←4
A	2	↑2	↖1	↖←2	↖←3	↖←4
T	3	↑3	↑2	↖2	↖2	←3
H	4	↑4	↑3	↖↑3	↖↑3	↖3
S	5	↑5	↑4	↖↑4	↖↑4	↖3

The solutions - #1

1	0	1	1	0	=	3
D	***M***	***R***	***R***	***M***		
M	A	T	H	S		
-	A	R	T	S		

The traceback

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1	↖ 1	↖← 2	↖← 3	↖← 4
A	2	↑ 2	↖ 1	↖← 2	↖← 3	↖← 4
T	3	↑ 3	↑ 2	↖ 2	↖ 2	← 3
H	4	↑ 4	↑ 3	↖↑ 3	↖↑ 3	↖ 3
S	5	↑ 5	↑ 4	↖↑ 4	↖↑ 4	↖ 3

The solutions - #2

1	0	1	0	1	0	=	3
D	***M***	***I***	***M***	***D***	***M***		
M	A	-	T	H	S		
-	A	R	T	-	S		

The traceback

Y			A	R	T	S
X		0	1	2	3	4
	0	0	← 1	← 2	← 3	← 4
M	1	↑ 1	↖ 1	↖← 2	↖← 3	↖← 4
A	2	↑ 2	↖ 1	↖← 2	↖← 3	↖← 4
T	3	↑ 3	↑ 2	↖ 2	↖ 2	← 3
H	4	↑ 4	↑ 3	↖↑ 3	↖↑ 3	↖ 3
S	5	↑ 5	↑ 4	↖↑ 4	↖↑ 4	↖ 3

The solutions - #3

1	1	0	1	0	=	3
R	***R***	***M***	***D***	***M***		
M	A	T	H	S		
A	R	T	-	S		

“Life must be lived forwards and understood backwards.”

- Søren Kierkegaard

Application 3: Longest Increasing Subsequence

Longest Increasing Subsequence (LIS)

Problem:

Given a string $X = <x_1\ x_2\ ..\ x_n>$, find a longest increasing subsequence

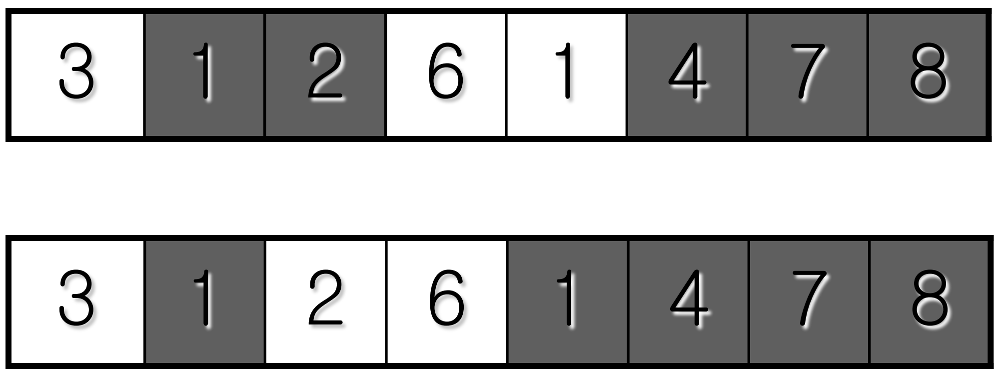
$Z = <z_1\ z_2\ ..\ z_k>$ such that

$$z_1 <= z_2 <= .. <= z_k$$

Example

- X=<3,1,2,6,1,4,7,8>

LIS =



Brute-force algorithm

- For every subsequence of X, check whether it's an increasing subsequence of X
  - 2^n subsequences of X to check
  - Each subsequence takes $\Theta(n)$ time

=> $\Theta(n\ 2^n)$ time

Recursive Formulation

L[i]: Length of LIS of $X_i = <x_1\ x_2\ ..\ x_i>$

that ends with x_i

P[i]: Index of the element before x_i

in the lis ending at x_i

- L[i] >= 1 for all i
- L[i] = 1 +

  max{ L[j]: $1 <= j < i$ and $x_j <= x_i$ }

```
LIS(X,L,P,n)
for(i=1.. n){
    L[i]=1;
    P[i]=0;
    for(j=1.. i-1){
      if((x[j]<=x[i])
     &&(L[j]+1 > L[i])){
          L[i]=L[j]+1;
          P[i]=j;
      }
    }
}
```

```
Print-LIS(X,L,P,n)

L[i] := max {L[j]:
  1<=j<=n}
RecoverHelper(X,L,P,i);
```

```
RecoverHelper(X,L,P,k)
if (k>0){
RecoverHelper(X,L,P,P[k]);
print X[k];
}
```

O(n log n) algorithm

Step i:

- LIS[j]= Smallest X[k] (k<=i) having an increasing subsequence of length j ending at this value
- P[i] = LIS에서의 위치

```
L = 0; LIS[0]=-infty; LIS[1..n+1]=infty;
for i=1, 2, .., n
    binary search for the largest j
    such that LIS[j] <= X[i];
    P[i] = j+1;
    if (j==L or X[i] < LIS[j+1]){
        LIS[j+1] = X[i]; L = max(L, j+1);
    }
```

STL – lower_bound

- 정렬되어 있는 배열에서
  - lower_bound
    : **크거나 같은 수 중**에 정열 상태 유지하면서 들어가도 되는 **첫 번째** 위치
  - upper_bound
    : **큰 수 중**에 정열 상태 유지하면서 들어가도 되는 **첫 번째** 위치

	1	3	3	6	7
2		L,U			
3		L		U	
4				LU	

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P								

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	i	i	i	i	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1							

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	i	i	i	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2						

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	10	i	i	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2					

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	9	i	i	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2	2				

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	2	i	i	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2	2	3			

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	2	3	i	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2	2	3	4		

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	2	3	8	i	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2	2	3	4	5	

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	2	3	8	8	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2	2	3	4	5	2

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	1	3	8	8	i	i	i

i = infinity

O(n log n) LIS Algorithm

idx	1	2	3	4	5	6	7	8
input	-7	10	9	2	3	8	8	1

P	1	2	2	2	3	4	5	2

(LIS 에서의 위치)

		1	2	3	4	5	6	7	8
LIS	-i	-7	1	3	8	8	i	i	i

i = infinity