1. Divide and Conquer

2. Greedy Algorithms

3. Dynamic Programming

Dynamic Programming is an algorithmic paradigm that solves the complex problems breaking it into sub-problems

(i) overlapping sub-problems

(2) optimal substructure

Overlapping sub-problems

5

6

7

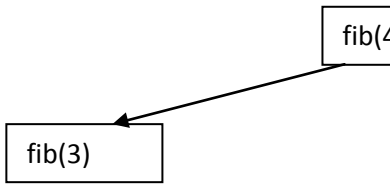
8

9

70

Recursive Algorithm for Fibinacci Sequence

```
int fib( int n)
{ if n< =1 then return 1;
   else return fib(n-1) + fib(n-2);
}
```



$T(n) = T(n-1)+T(n-2)+O(1)$

$= O(2^n)$

Dynamic Programming

(1) Memoization(Top Down)

(2) bottom up (Tabulation)

Memoization(Top Down)

....To store the solutions of the sub-problems in the memory

and while solving the relatively complex problems which requires the solutions of the previously solved subproblems, we

consult with the memory to find if any solution is available or not.

Lookup[ ]= nil

```
int fib (int n)

{ if (lookup[n]==nil)
    {
        if (n<=1)   lookup[n]=n;
        else lookup[n]= fib(n-1) + fib(n-2);

    }

return lookup(n);

}
```

Many recursive calls

(2) bottom up (Tabulation)

```
int fib (int n)


{ f[0]=0;
f[1]=1;
for (i=2; i<=n; i++)
  f[i]=f[i-1]+ f[i-2];

return [n];
}
```

O(n)

2
1
1
0

Finding Longest increasing Sub-sequences

input : 5 2 8 6 3 6 9 7

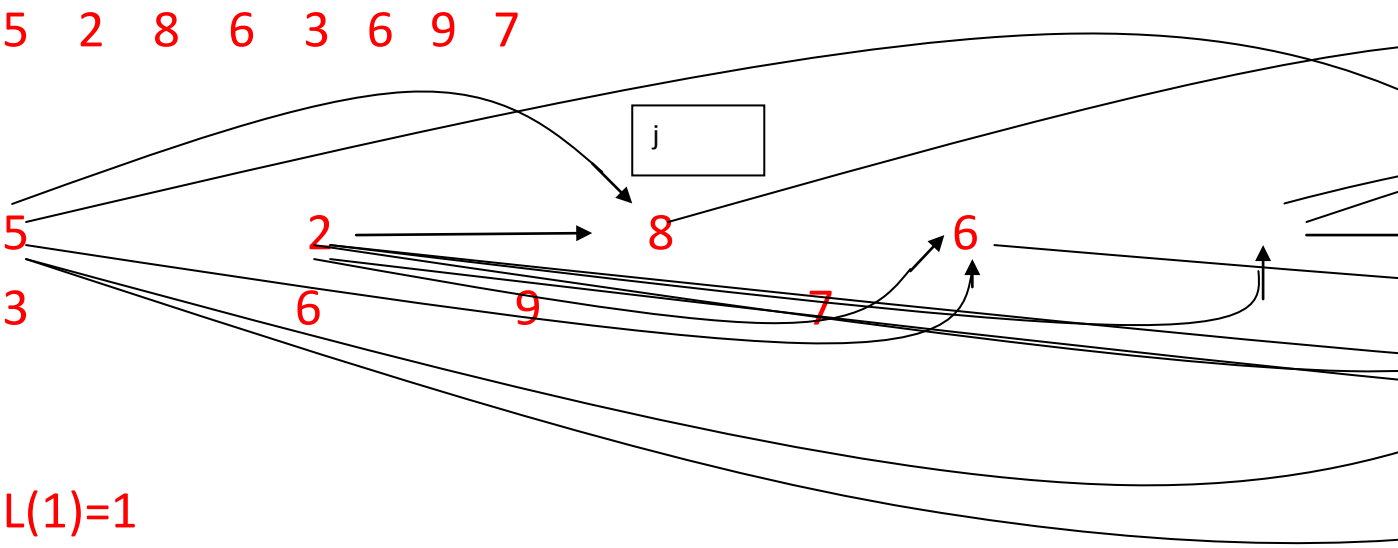
Longest increasing Subsequences: 2 3 6 9

5 2 8 6 3 6 9 7

Brute Force method: Consider all possible candidate solutions and find the best

1. Generate all possible subsets
2. Discard the sub-sequences which are not increasing
3. Among the valid candidates, find the sub-sequence with max length.

Dynamic Programming Solution:



L(1)=1

L(2)=1

L(3)=1+max(L(1), L(2))=1+1=2

L(4)=1+ max(L(1), L(2))= 1+1=2

L(5)=1+1=2

L(6)=1+ max(L(1), L(2), L(5))= 1+2=3

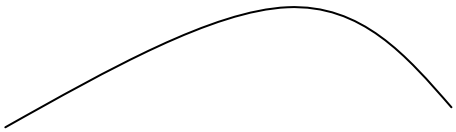
Recursive implementation : Exponential number of function calls

Dynamic Programming implementation with Memoization

The longest increasing sub-sequence upto node j, L(j)

for j= 1 to n

L(j)=1+max{L(i): (i,j) belongs to E}



O(m), number of edges

return $\max_j L(j)$

O(n)

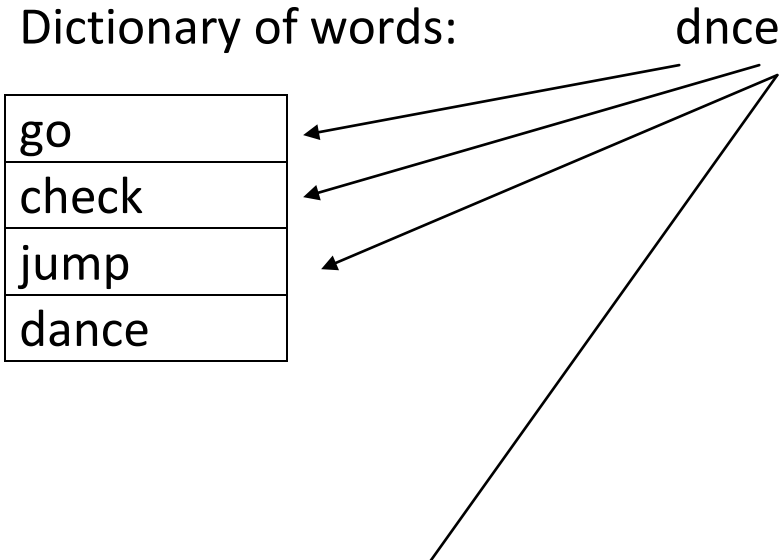
Computation of L(j) is proportional to in-degree of node j

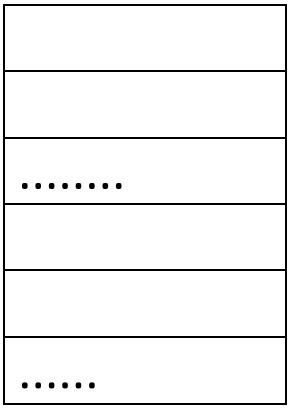
Total complexity of this algorithm is proprotional to the sum of in-degrees of all nodes

Running time is O(m) + O(n) = O(m) (m>n)

To find minimum edit distance:

Spell checking, DNA sequence alignment





Possible typing mistakes: danci (substitution)

daance (insertion)

dnce (deletion)

Edit operations: insertion, deletion, substitution

Edit distance: number of insertions, deletions and substitutions that I might have made so that the first string has been written as the second string

Cost: number of insertions, deletions and substitutions are made to convert the first string to the second

We do not exact positions where such operations have been made.

So, we will focus on finding the minimum number of edit operations (Hence the name is

minimum edit distance)

SNOWY

SUNNY

S N O W Y -

S U N N Y -

n!= omega(2^n) n. (n-1) . n-2.....1

Cost: number of insertions, deletions and substitutions are made to convert the first string to the second

Cost:3

- S N O W -
Y

S U N - - N
Y

ins subs subs(with 0 cost) dels dels ins subs(with 0 cost)

cost:5

Edit distance:

min is 3?

Minimum Edit distance

Brute force method:

1. Consider all possible alignments
2. Calculate the cost of each alignment
3. Choose the alignment with minimum value

Dynamic Programming Approach

Formulation of the problem..

x[1..i] and y [1...j]

x(i) and y[j)

x[i] - x[i]

- or y[j] or y[j]

EXPO

POL

O or - or O

- L L

3 possibilities are:

(1) Delete "O" from EXPO and align EXP with POL
cost: cost of deletion + E(3, 3)

(2) Insert "L" from POL at the end of EXPO and align EXPO with PO (cost of insertion + E(4, 2))

(3) Substitute O in EXPO by L in POL and align EXP and PO (cost of substitution + E(3,2))

E(4,3)= the cost of alignment between string: x(1)---x(4) (EXPO) and the string: y(1)...y(3) (POL)

E(4,3)=minimum edit distance between the first string of length 4 and the second string of length 3

= min(cost of deletion + E(3,3), cost of insertion + E(4,2), cost of substitution + E(3,2))

E(i, j)= min (1 + E(i-1,j), 1 + E(i,j-1), diff(i,j) + E(i-1,j-1))

cost of deletion= cost of insertion=1

cost of substitution=diff(i,j), diff(i,j)=0 if x[i]=y[j] else diff(i,j)=1

Algorithm(m, n) (m is the len of 1st string and n is the length of second string)

for i = 0, 1, 2, . . . , m:

E(i, 0) = i /* the second one is an empty string*/

for j = 1, 2, . . . , n:

E(0, j) = j /* the first one is empty string*/

for i = 1, 2, . . . , m:

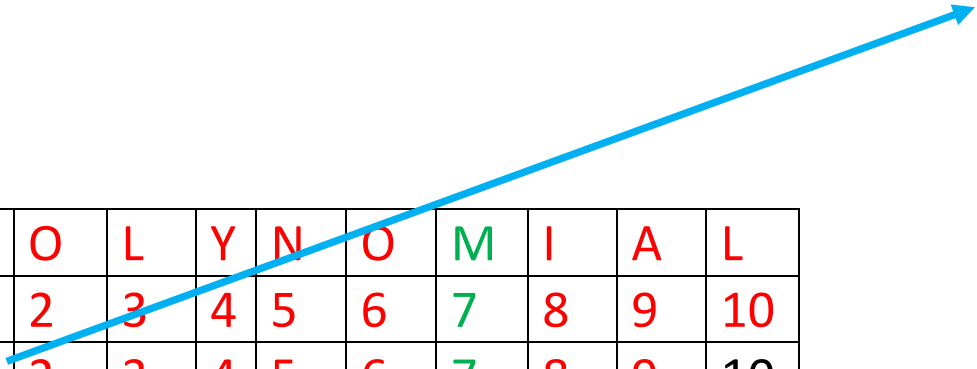
for j = 1, 2, . . . , n:

E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}

return E(m, n)

P

E



	-	P	O	L	Y	N	O	M	I	A	L
-	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	7	8	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Running time = O(mn)

E X P O N E N - T I A L

- - P O L Y N O M I A L

DNA sequence alignment: Modified version of this algorithm is used