Dynamic Programming

→ Dynamic Programming can be defined as the method of solving optimization problem which exhibit property of overlapping subproblem and optimal sub-structoral.

→ Dynamic Programming solves the ~~problem~~ each sub-problem once and stores the result into table so that it can be repeatedly retrieved if needed again.

→ Dynamic Programming is buttom-up approach where we solve all the possible sub-problem and combine to obtain solution.

Characteristis of Dynamic Programming

- Overlapping Substructure
  - ↳ if optimal solution contain optimal-sub. solution, then problem exhibit overlapping sub-strustrure.

- Overlapping Sub-problem
  - ↳ when recursive problem visit the same sub-problem repeadly, then problem exhibit overlapping sub-problom.

Dynamic Programming	Greedy Method
→ It solves the overlapping sub-problem and combining there sub-solution to get optimal global solution.	→ It solves the problem by making locally optimal choice at each step hoping that it lead to global solution.
→ It guarantee the optimal solution of that problem.	→ It doesn't guarantee the optimal solution.
→ It is comparetaively slower since it need to store the solution of sub-problem and later combine them.	→ Greedy method are generally faster since they make quick decision based on current best option.
→ It uses more memory.	→ It require less memory.
→ It store solution in a table.	→ It doesn't store / require table
→ It is suitable for those problem that consist of overlapping sub-problem.	→ It is suitable for the problem that make locally optimal choice that lead to global optimal solution such as huffman.
→ It is more efficient and reliable	→ It is less efficient and reliable.
→ example :- 0/1 knapsack	example :- fractional knapsack.

Dynamic Programming	Divide and Coquer Method
→ Defination ...	→ Break down into subproblem, find there solution/conquer and merge them.
→ It is not recursive	→ It is recursive
→ It is Bottom-up approach	→ It is top-down approach
→ It solve sub-problem only once.	→ It may solve sub-problem multiple time.
→ The sub-problem are dependent to eachother.	→ The sub-problem are not dependent to eachother.
→ It often require more memory as it store solution to subproblem in table.	→ It typically require less memory as it solve sub-problem independently.
→ It uses DP table	→ It doesn't require any table.
example:- 0/1 knapsack	example :- Merge Sort.

Dynamic Programming

Matrix Chain Multiplication

Algorithm

```
Matrix-Chain(p)
 {
   n = length(P);

   for(i=1; i<=n; i++)
       m[i,i]=0;

   for(l=2; l<=n; l++)
     {
       for(i=1; i<=n-l+1; i++)
         {
           j = i+l-1;
           m[i,j] = ∞;

           for(k=i; k<=j-1; k++)
             {
               c = m[i,k]+m[k+1,j]+ p[i-1]*p[k]*p[j];
               if(c < m[i,j])
                 {
                   m[i,j] = c;
                   s[i,j] = k;
                 }
             }
         }
     } }
   return m and s;
 }
```

Time Complexity

The above algorithm have 3 nested loop, thus Time Complexity is given by

$$T(n) = O(n^3)$$

Q1) Consider matrix A_1, A_2, A_3, A_4 order of 3×4, 4×5, 5×2 and 2×3

soln

let,

$P_0 = 3$

$P_1 = 4$

$P_2 = 5$

$P_3 = 2$

$P_4 = 3$

Now, constructing M table and S table

M-table

	1	2	3	4
1	0	60	64	82
2		0	40	64
3			0	30
4				0

S table

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

$M[1,1] = 0$
$M[2,2] = 0$
$M[3,3] = 0$
$M[4,4] = 0$

since, if $i=j$, the $m[i,j] = 0$

for,
$M[1,2] = \min\{M[i,k] + M[k+1,j] + P_{i-1}P_kP_j\} = 60$

similarly,
$M[2,3] = 40$
$M[3,4] = 30$

Now,

$$M[1,3] = \min\left\{\begin{array}{l} \overset{0}{M[1,1]} + \overset{40}{M[2,3]} + \overset{24}{P_0P_1P_3} \\ \overset{60}{M[1,2]} + \overset{0}{M[3,3]} + \overset{30}{P_0P_2P_3} \end{array}\right\} = 64$$

$$M[2,4] = \min\left\{\begin{array}{l} \overset{0}{M[2,2]} + \overset{30}{M[3,4]} + \overset{60}{P_1P_2P_4} \\ \overset{40}{M[2,3]} + \overset{0}{M[4,4]} + \overset{24}{P_1P_3P_4} \end{array}\right\} = 64$$

Now,

$$M[1,4] = \min\left\{\begin{array}{l} \overset{0}{M[1,1]} + \overset{64}{M[2,4]} + \overset{36}{P_0P_1P_4} \\ \overset{60}{M[1,2]} + \overset{30}{M[3,4]} + \overset{45}{P_0P_2P_4} \\ \overset{64}{M[1,3]} + \overset{0}{M[4,4]} + \overset{18}{P_0P_3P_4} \end{array}\right\} = 82$$

from table we obtain optimal cost $= 82$

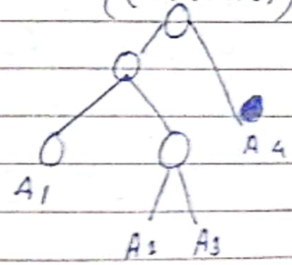
from the S-table,

$(A_1\ A_2\ A_3)(A_4)$

$((A_1)(A_2A_3))(A_4)$

Here, we 1st multiply $A_2 \times A_3$, then the result to A_1 then finally A_4.

$((A_1)(A_2A_3))(A_4)$



0/1 KnapSack / Dynamic Knapsack

```
DynaknapSack ( W, n, v, w )
{
  for( w = 0 ; w <= W ; w++ )
      c[0,w] = 0 ;

  for( i = 1 ; i <= n ; i++ )
      c[i,0] = 0 ;

  for ( i = 1 ; i <= n ; i++ )
  {
    for ( w = 1 ; w <= W ; w++ )
    {
      if ( w[i] <= w )
      {
        if ( v[i] + c[i-1, W-W[i]] > c[i-1, W] )
            c[i,w] = v[i] + c[i-1, W-W[i]] ;
        else
            c[i,w] = c[i-1,W] ;
      }
      else
          c[i,W] = c[i-1,w] ;
    }
  }
}
```

Time Complexity
$T(n) = O(n \cdot W)$

Q.
$w = \{3, 5, 7, 4, 3, 9, 2\}$ $\quad W = 9$
$v = \{2, 3, 3, 4, 4, 5, 7\}$
$n = 7$

Now,

i \ w	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2
2	0	0	0	2	2	3	3	3	5	5
3	0	0	0	2	2	3	3	3	5	5
4	0	0	0	2	4	4	4	6	6	7
5	0	0	0	4	4	4	6	8	8	8
6	0	0	0	4	4	4	6	8	8	8
7	0	0	7	7	7	11	11	11	13	15

Max profit = 15.

Item picked = I_7, I_5, I_4

Longest Common Sequence

```
LCS(X,Y)
{
  m = length(X)
  n = length(Y)

  for(i=1; i≤m; i++)
     c[i,0]=0;

  for(j=0; j≤n; j++)
     c[0,j]=0;

  for(i=1; i≤m; i++)
  {
    for(j=1; j≤n; j++)
    {
      if(X[i]==Y[j])
      {
        c[i,j]= c[i-1,j-1]+1; b[i,j] = "upleft";
      }
      elseif(c[i-1,j] ≥ c[i,j-1])
      {
        c[i,j] = c[i-1,j]   ; b[i,j] = "up";
      }
      else
        c[i,j] = c[i,j-1] ; b[i,j] = "left";
    }
    return b and c
}
```

Time Complexity,

$T(n) = O(m \cdot n)$

Q.No

X = A B C B D A B

Y = B D C A B A

Now, Constructing Table

X \ Y	Ø	B	D	C	A	B	A
Ø	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

Hence,

Longest Sequence length = 4

ie BDAB

Floyd Warshwall Algorithm

Algorithm :-

W = weight , D = adj Matrix , n = no. of nodes

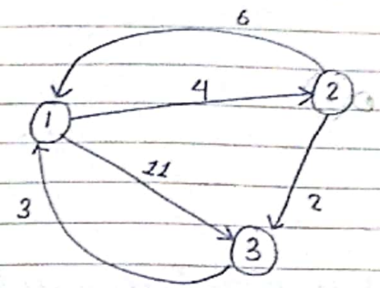
```
Floyd (W, D, n)
{
for ( i= 1 ; i≤n ; i++ )
{
  for ( j= 1 ; j≤n ; j++ )
  {
     D[i][j] = W[i][j] ;     // D⁰
  }
}

for ( k=1 ; k≤n ; k++ )
{
 for ( i=1 ; i≤n ; i++ )
  {
   for ( j=1 ; j≤n ; j++ )
   {
    if ( D[i][j] > D[i][k] + D[k][j] )
      then
      D[i][j] = D[i][k] + D[k][j] ;
   }
  }
 }
}
```


Time Complexity

$T(n) = O(n^3)$

eg



$$D^0 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & 4 & 11 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & \infty & 0 \end{array}$$

$$D^1 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & 4 & 11 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

$$D^2 = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$D^3 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & 0 & 4 & 6 \\ 2 & 5 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

Traveling Salesman

Algorithm:-

TSP(W, n)

{

c[j, { }, j] = 0

for s = 2 to n

{

for all subset S belongs to {1, 2, ...n} of size s

{

c [s, S, j] = ∞

}

for all i ∈ S and i ≠ j

{

c [i, j] = min { w[i, j] + c[i, S - {i}, j] }

}

}

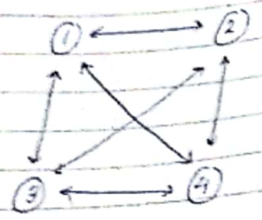
Return min(1) c[1, {2, 3. . . .n}, j + w[i, j];

}

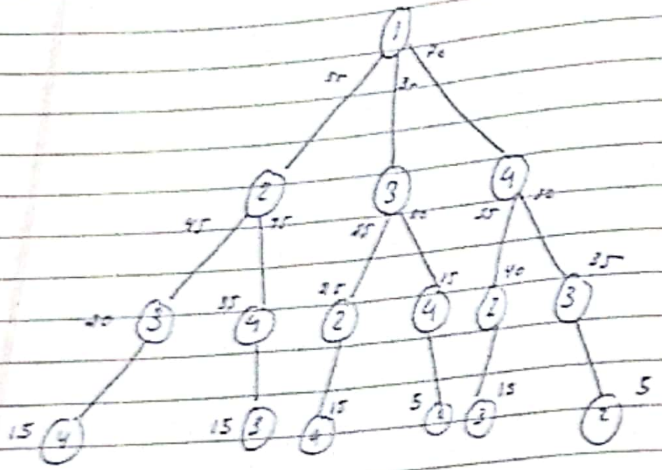
Time Complexity : $T(n) = O(2^n)$

classmate

PAGE



	1	2	3	4
1	0	10	15	20
2	5	0	25	10
3	15	30	0	5
4	15	10	20	0



We know,

$$g(i, S) = \min(w[i,j] + g(j, S-\{j\}))$$

Now,

$$g(1,\{2,3,4\}) = \min\begin{pmatrix} w[1,2] + g(2,\{3,4\}) \\ w[1,3] + g(3,\{2,4\}) \\ w[1,4] + g(4,\{2,3\}) \end{pmatrix}$$

$$g(2,\{3,4\}) = \min\begin{pmatrix} w[2,3] + g(3,\{4\}) \\ w[2,4] + g(4,\{3\}) \end{pmatrix} = 45$$

$$g(3,\{4\}) = \min(w[3,4] + g(4,\{\emptyset\})) = 20$$

$$g(4,\{3\}) = \min(w[4,3] + g(3,\{\emptyset\})) = 35$$

$$g(3,\{2,4\}) = \min\begin{pmatrix} w[3,2] + g(2,\{4\}) = 55 \\ w[3,4] + g(4,\{2\}) = 20 \end{pmatrix} = 20$$

$$g(2,\{4\}) = \min(w[2,4] + g(4,\emptyset)) = 25$$

$$g(4,\{2\}) = \min(w[4,2] + g(2,\emptyset)) = 15$$

$$g(4,\{2,3\}) = \min\begin{pmatrix} w[4,2] + g(2,\{3\}) = 50 \\ w[4,3] + g(3,\{2\}) = 55 \end{pmatrix} = 50$$

$$g(2,\{3\}) = \min(w[2,3] + g(3,\emptyset)) = 40$$

$$g(3,\{2\}) = \min(w[3,2] + g(2,\emptyset)) = 35$$

finally,

$$g(1,\{2,3,4\}) = \min\begin{pmatrix} w[1,2] + g(2,\{3,4\}) = 55 \\ w[1,3] + g(3,\{2,4\}) = 35 \\ w[1,4] + g(4,\{3,2\}) = 70 \end{pmatrix} = 35$$

Memorization v/s Dynamic Programming

Memorization

→ Memorization can be defined as optimization technique, we memorize the previously computed result, which will be used whenever the same result is required.

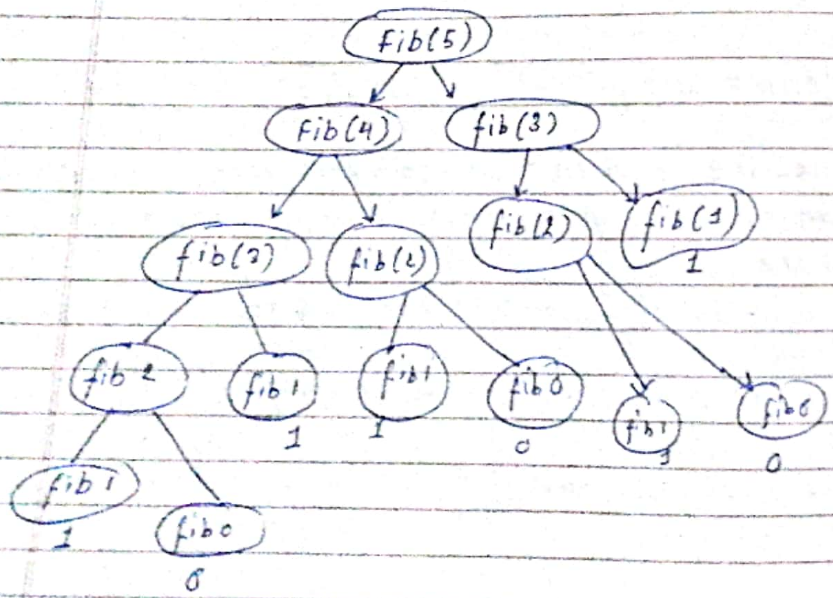
→ It caches the result of function call in data structure.

Memorization	Dynamic Programming
→ It caches the required function call in a data structure.	→ It solve/store subproblem in array or table.
→ It is top-down approach	→ It is buttom-up approach.
→ Have lower space-complexity.	→ Have higher space-complexity.
→ It only solve sub-problem once.	→ It may solve sub-problem repeadtly.

For example :-

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

using Dynamic Programming



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

So,

$TC = O(2^n)$

using Memoization

0	1	2	3	4	5
0	1	-1	-1	-1	-1

0	1	2	3	4	5
0	1	1	-1	-1	-1

0	1	2	3	4	5
0	1	1	2	-1	-1

0	1	2	3	4	5
0	1	1	2	3	-1

0	1	2	3	4	5
0	1	1	2	3	5