38 Dynamic Programming

→ Most popularly used design technique

→ Dynamic programming is similar to divide & conquer, the problem will be divided into subproblem

→ Dynamic programming, the overlapping subproblem is solved only once.

→ In dynamic programming, the solutions of the subproblems are stored in a table

→ To solve an problem

i) Optimal substructure

ii) Recurrence

iii) Approach — Top-down
\ Bottom-up

Binomial-coefficient

The objective of binomial coefficient is to find the no. of combinations of 'k' elements

→ It is denoted as $C(n, k)$ or ${}^{n}C_{k}$

$${}^{n}C_{k} = \frac{n!}{k!(n-k)!}$$

→ The recurrence definition to find

$$C(n,k) = \begin{cases} 1 & \text{if } k=0 \text{ or } k=n \\ 0 & k>n \\ C(n-1,k) + C(n-1,k-1), & k<n \text{ \& } k \neq 0 \end{cases}$$

Eg:- ${}^4C_3 = \frac{4!}{3! \times 1!} = 4$

$n=4, k=3$

Table is constructed with $n+1$ rows & $k+1$ columns

→ k

n ↓	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	(4)

2) ${}^6C_4 = 15$

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1
5	1	5	10	10	5
6	1	6	15	20	(15)

Algorithm Binomial coefficient (n, k)

// Computes $c[n,k]$ by dynamic programming

// Input: Two non-negative integers n & k

// Output: Value of $C(n,k)$

```
for i ← 0 to n do
    for j ← 0 to min(i,k)
        if j=0 or j=i
            C[i,j] ← 1
        else
            c[i,j] = c[i-1,j] + c[i-1,j-1]
return c[n,k]
```

Basic operation is addition

$$T(n) = \sum_{i=0}^{n} \sum_{j=0}^{k} 1 = O(nk)$$

Recurrence relation

$A(n,k) = 0$ when $k=0$, $k=n$

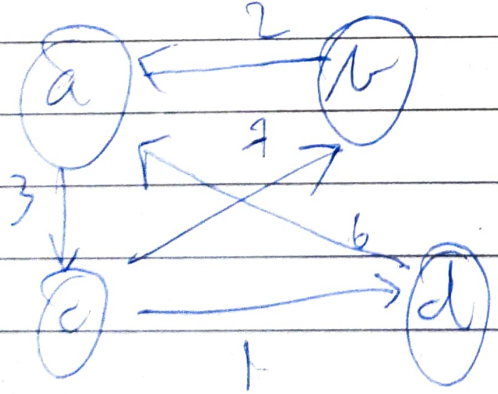
$A(n,k) = A(n-1,k) + A(n-1,k-1) + 1$, otherwise

39) Floyd's Algorithm

Goal: All pair shortest paths algorithm

Input: A weighted graph

n as no. of vertices



n = 4

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

D_1

0	∞	3	∞	
0	∞	3	∞	
2	2	0	5	∞
∞	∞	7	0	1
6	6	∞	9	0

D_2

2	0	5	∞	
∞	20	∞	3	∞
0	2	0	5	∞
7	9	7	0	1
∞	6	∞	9	0

D_3

9	7	0	1	
3	0	10	3	4
5	2	0	5	6
0	9	7	0	1
9	6	16	9	0

D_4

6	16	9	0	
4	0	10	3	4
6	2	0	5	6
1	7	7	0	1
0	16	16	9	0

Algorithm Floyd's(W[1..n, 1..n])

// Implements Floyd's algo, using dynamic programming

// Input: A weight graph represented through a matrix

// Output: D matrix which gives all-pair shortest path

```
D ← W
for i ← 1 to n do
    for j ← 1 to n do
        for k ← 1 to n do
            D[i,j] ← min{D[i,j], D[i,k] + D[k,j]}

return D
```

$\theta(n^3)$

40 (Program to implement Floyd's Algorithm

```
#include <stdio.h>
#include <stdlib.h>
int min (int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}

int main()
{
```

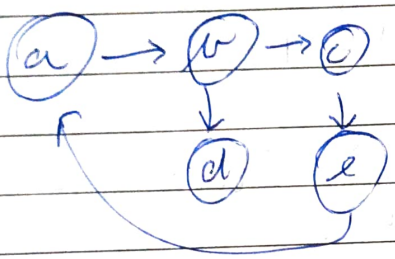
```
int n, i, j, D[10][10];
printf("\n Read the no. of nodes:");
scanf("%d", &n);
printf("\n Read the weighted matrix \n");

for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        scanf("%d", &D[i][j]);

Floyds(D, n);

printf("\n The all pair shortest path is \n");
for (i=1; i<=n; i++)
{
    for (j=1; j<=n; j++)
    {
        printf("%d\t", D[i][j]);
    }
    printf("\n");
}
return 0;
}

void Floyds(int D[10][10], int n)
{
    int i, j, k;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            for (k=1; k<=n; k++)
                D[i,j] = min(D[i][j], D[i,k] +
                    D[k, j]);
}
```

4) Warshall's Algorithm

→ Used to compute the transitive closureness

of a directed graph

→ The transitive closureness of a directed graph with 'n' vertices can be defined as an n×n boolean matrix $T = \{t_{ij}\}$ in which the element in the i^{th} row & the j^{th} column is (1) if there exists a non-trivial directed path from i^{th} vertex to j^{th} vertex; otherwise it is (0)

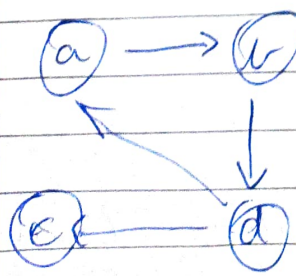


Input

	a	b	c	d	e
a	0	1	0	0	0
b	0	0	1	1	0
c	0	0	0	0	1
d	0	0	0	0	0
e	1	0	0	0	0

Output → Adjacency matrix

	a	b	c	d	e
a	1	1	1	1	1
b	1	1	1	1	1
c	1	1	1	1	1
d	0	0	0	0	0
e	1	1	1	1	1



R^e

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Old value of R1 and R2

R^1

0	0	1	0	0
0	0	1	0	0
0	0	0	0	1
0	0	0	0	0
0	1	1	1	0

R^2

1	0	1	0	1
0	0	0	0	1
0	0	0	0	0
1	1	1	1	1

R^3

0	0	0	0	1
0	0	0	0	1
0	0	0	0	0
1	1	1	1	1

R^4

1	1	1	1	1
1	1	1	1	1
0	0	0	0	0
1	1	1	1	1

Algorithm Warshalls $(A[1 \ldots n, 1 \ldots n])$

// Implements Warshall Algo to compute transitive closureness

// Input : Adjacency matrix A of the directed graph

// output : Transitive closureness of a diglaph

$R^0 \leftarrow A$

for $i \leftarrow 1$ to n do
  for $j \leftarrow 1$ to n do
    for $k \leftarrow 1$ to n do
      $R^k[i,j] = R^{k-1}[i,j]$ or $R^{k-1}[i,k]$ and $R^{k-1}[j,k]$

return R^n

Time complexity :- $O(n^3)$

2 0-1 Knapsack Problem

→ In the knapsack problem there will be n objects of known weights denoted by $w_1, w_2, w_3 \ldots w_n$ and profits denoted by $v_1, v_2, v_3 \ldots v_n$ and a knapsack capacity denoted by W

→ Here the aim is to add more profitable objects into the knapsack with a constraints that weight of the added objects should not exceed the knapsack capacity.

→ Let x_i denote whether the object is included or not

$x_i = 1$ (object is included)

$x_i = 0$ (object is not included)

→ The knapsack problem can be denoted as

Maximise $\sum_{i=1}^{n} x_i v_i$ subjected to

$$\sum_{i=1}^{n} x_i w_i \leq W$$

Through brute-force,

$n = 3$

wt = 2 3 2

pt = 5 10 8

1 - 5 Max profit = 18

2 - 10

3 - 8

1, 2 - 15

1, 3 - 13

2, 3 - 18

1, 2, 3 - 23

To solve knapsack problem using dynamic programming, we construct a table of n+1 rows (n = no. of objects) 0 ... n and W+1 columns (W = Knapsack capacity) 0 ... W
Let the table be denoted by V, the entries in V are calculated as

$$V[i,j] = \begin{cases} V[i-1,j], & \text{if } j - w_i < 0 \\ \max\{V[i-1,j], V[i-1, j-w_i] + v_i\}, & \text{if } j - w_i \geq 0 \end{cases}$$

Eg.- n = 4 W = 5

→ Knapsack capacity

objects ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	8	8	8	8
2	0	6	8	14	14	14
3	0	6	8	16	22	24
4	0	6	11	17	22	27

(w) Weight = {2, 1, 3, 2}
(v) Profits = {8, 6, 14, 11}

V[1,1] = 0 | 1-2 = -1
V[1,2] = 8 | 2-1 = 1
max{V[0,2], V[0,0]+8}
= 8
V[1,3] = 8 | 3-1 = 2
V[1,4] = max{0, 0+8} = 8

V[2,1] = | 1-1 = 0
V[2,2] = max{V[1,1], V[1,0]+6} = 6
V[2,3] = | 2-1 = 1
max{V[1,2], V[1,1]+6} = 14
V[2,4] = | 3-1 = 2
max{V[1,3], V[1,2]+6} = 14

V[3,1] = | 1-3 = -2
V[3,2] = V[2,1] = 0
V[3,3] = | 2-3 = -1
V[3,4] = V[2,2] = 8
V[3,5] = | 3-3 = 0
max{V[2,3], V[2,0]+16}
= max{14, 16} = 16

V[4,1] = | 1-2 = -1
V[4,2] = V[3,1] = 0
V[4,3] = | 2-2 = 0
max{V[3,2], V[3,0]+11}
= max{8, 11} = 11
V[4,4] = | 3-2 = 1
max{V[3,3], V[3,1]+11}
= max{16, 11} = 16
V[4,5] = | 4-2 = 2
max{V[3,4], V[3,2]+11}
= max{8, 11} = 11

$V[3,4]$ = | $4-3=1$

$\max\{V[2,4], V[2,1]+16\}$

$= \{14, 6+16\} = 22$

$V[3,5]$ = | $5-3=2$

$\max\{V[2,5], V[2,2]+16\}$

$= \max\{14, 8+16\} = 24$

$V[4,1]$ = | $1-2=-1$

$V[4,2]$ = | $2-2=0$

$\max\{V[3,2], V[3,0]+11\}$

$= \{8, 11\}$

$= 11$

$V[4,3]$ = | $3-2=1$

$\max\{V[3,3], V[3,1]+11\}$

$= \{16, 6+11\} = 17$

$V[4,4]$ = | $4-2=2$

$\max\{V[3,4], V[3,2]+11\}$

$= \{22, 8+11\} = 22$

$V[4,5]$ = | $5-2=3$

$\max\{V[3,5], V[3,3]+11\}$

$= \{24, 24\} = 24$

Max profit : $V[4,5] = 27$

$V[4,5] \neq V[3,5]$

(4th object should be included)

weight of 4th object

$V[3,5-2] = V[3,3]$

$V[3,3] \neq V[2,3]$

(3rd object should be included)

$V[2,3-3] = V[2,0]$

Should be stopped

∴ Objects included are 3rd & 4th object

Knapsack Problem using memory function

* Memory function uses the concept of both the top down & bottom-up approach

* Memory functions are generally recursive in nature

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

MFKP

Algorithm ~~Memory Function Knapsack~~ (i, j)

// Implements memory function method to solve the knapsack problem

// Input: A non-negative integers
i - no. of first objects being considered
j - knapsack capacity

// Output: The values of an optimal feasible subset of first i items

// Note: 1st row ← 0, 1st column ← 0, rem ← -1

```
if V[i, j] < 0
    if j < wt[i]
        value ← MFKP(i-1, j)
    else
        value ← max(MFKP(i-1, j),
                    MFKP(i-1, j-wt[i]) + v[i])
```

V[i,j] ← Value

return V[i,j]

Of the previous example

V[4,5] = max(V[3,5], V[

n=4, W=5

wt: 2 1 3 2

V: 8 6 16 11

(i,j) = (4,5)

5,2

V[4,5] = max(V[3,5], V[3,3] + 11)

5,3

V[3,5] = max(V[2,5], V[2,2] + 16)

V[3,3] = max(V[2,3], V[2,0] + 16)

V[2,5] = max(V[1,5], V[1,4] + 6)

V[2,2] = max(V[1,2], V[1,1] + 6)

V[2,3] = max(V[1,3], V[1,2] + 6)

V[2,0] = 0

V[1,5] = max(0, 0 + 8) = 8

V[1,4] = 8

V[1,2] = 8

V[1,1] = 0

V[1,3] = 8 | V[1,2] = 8

V[2,3] = 14

V[2,4] = 8

V[2,5] = 14

V[3,3] = max(14, 16) = 16

V[3,5] = max(14, 24) = 24

V[4,5] = max(24, 16+11) = 27

43) C Program to Solve Knapsack Problem using Dynamic Programming

```
#include <stdio.h>
#include <stdlib.h>
int n, W, w[10], v[10], V[10][10], x[10];
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int main()
{
    int i;
    printf("\n Read no. of objects: ");
    scanf("%d", &n);
    printf("\n Read knapsack capacity");
    scanf("%d", &W);
    printf("Read weights of the object \n");
    for (i = 1; i <= n; i++)
        scanf("%d", &w[i]);
```

(j-w[i]

11

```
printf("\n Read Profits of the objects \n");
for(i=1;i<=n;i++)
    scanf("%d", &V[i]);
knapsack();
printSolution(); // This will tell which objects
                    are there in the knapsack
for(i=1;i<=n;i++)
    if(x[i]==1)
        printf("%d\t%d\t%d\n", i, w[i],
                V[i]);

printf("\n Max profit is %d \n", V[n][W]);
return 0;
}

void knapsack()
{
  int i,j;
  for(i=0;i<=n;i++)
  {
    for(j=0;j<=n;j++)
    {
      if(i==0||j==0)
          V[i][j]=0;
      else if(j<w[i])
          V[i][j]=V[i-1][j];
      else
          V[i][j]=max(V[i-1][j], V[i-1]
                  [j-w[i]]+W[i]);
      printf("%d\t", V[i][j]);
    }
    printf("\n");
  }
}
```

```
void printSolution()
{
    int i,j;
    i = n;
    j = W;
    while(i!=0 && j!=0)
    {
        if(V[i][j] != V[i-1][j])
        {
            x[i] = 1;
            j -= w[i];
        }
        i--;
    }
}
```