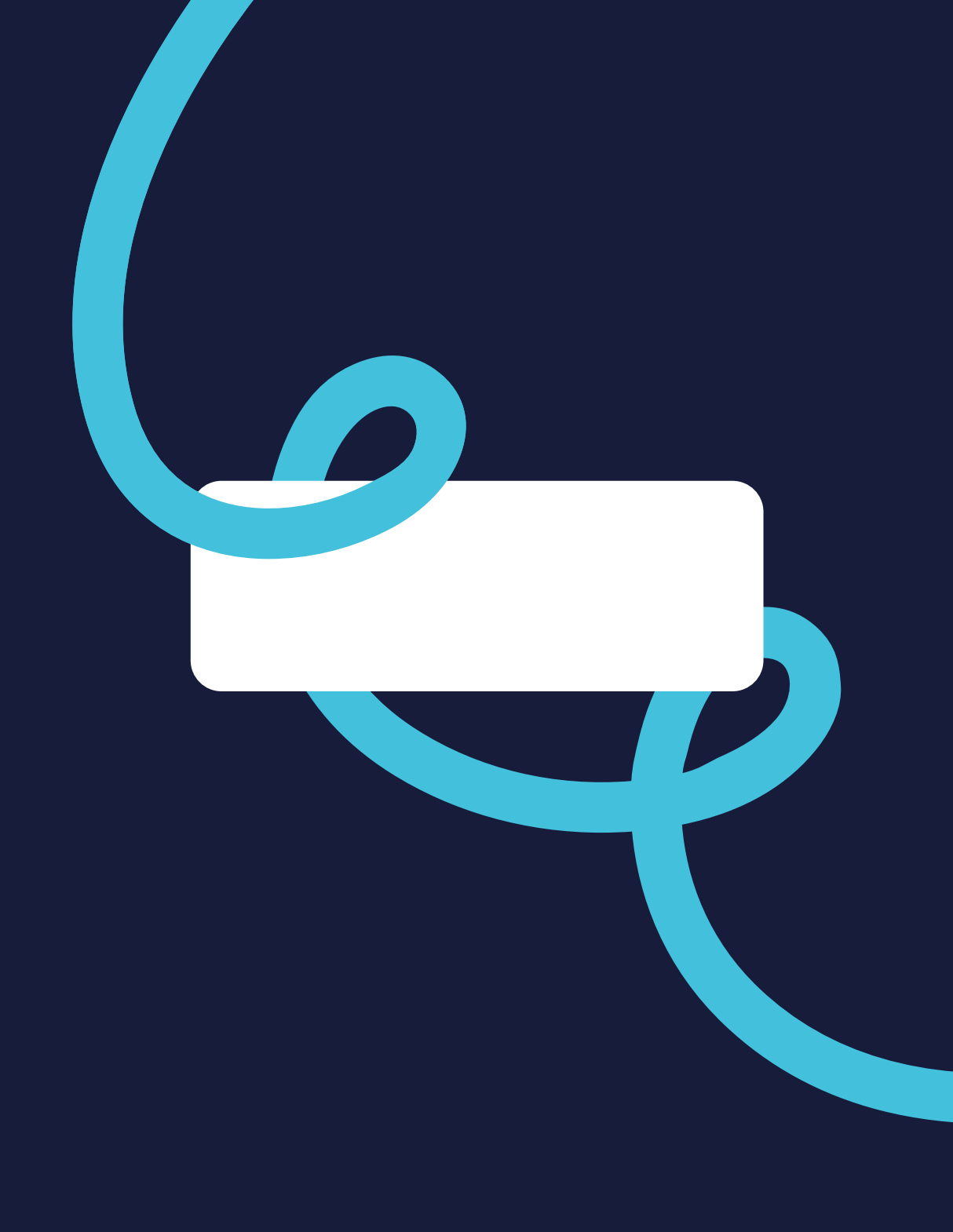


04/18/24

↔ **Set**: a built-in data type that stores an unordered collection of unique elements.

↳ commonly used when the presence of an element in a collection is more important the order of frequency of elements.

↳ Being implemented as hash tables, sets provide average time complexity of O(1) for lookups, which makes them highly efficient for certain operations like checking for memberships, removing duplicates from a sequence, and performing mathematical operations like intersection, union, difference, and symmetric difference

↳ Creat set using: {} or set() constructor.

↳ Add elements : • add() method

↳ Remove " : • remove() (raises an error if the element is not present.
• discard() (doesn't raise error)

↳ Membership test : 'in' keyword to check if an element exists in a set

** Recursion: fundamental programming concept and technique in which a function calls itself directly or indirectly to solve a problem.
↳ It breaks down a large problem into smaller, more manageable parts, each of which is solved by a recursive call.

Key Concepts:

① Recursive Case: part of recursion that splits the problem into sub-problems and calls the same func to solve these sub-problems

② Base Case: Also known as terminating condition, this stops the recursion.

③ Stack: Each recursive call adds a layer to the stack. When a recursive call reaches the base case, the stack begins to unwind as each call completes and returns its result.

Used in Tree traversal and sorting algorithms.

Optimize by:
↳ Tail recursion: → special kind
→ recursive call is the last operation in the func.
↳ Memoization.

Dynamic Programming :-

↳ The basic idea of DP is :
- identifying and solving subproblems.
- using subproblems together to solve larger problem.

5 simple steps for solving DP problems :

↳ using 2 problems :

1. Longest Increasing Subsequence (LIS) :

For a sequence $a_1, a_2, \ldots, a_n$, find the length of the longest increasing subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$

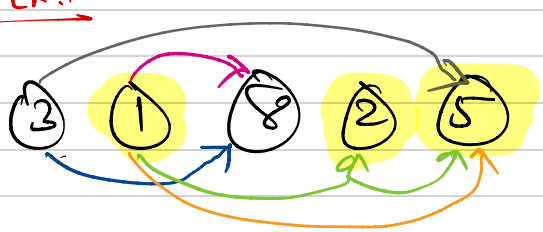
Constraints: $i_1 < i_2 < \ldots < i_k$;
$a_{i_1} < \ldots < a_{i_k}$

[focus on length of LIS]

Ex: LIS([3 1 8 2 5]) → 3

LIS([5 2 8 6 3 6 9 5]) → 4

Step 1. Visualize Ex :-



LIS = Longest Path in DAG + 1

2. Find an appropriate subproblem

↳ simpler version of our main problem.

→ Here, All increasing subsequences are subsets of original sequence.
→ All increasing subsequences have a start and end.
→ Let's focus on the end index of an increasing subsequence

So, here : subproblem : LIS[k] = LIS ending at index k

Ex: LIS[3] = 2

3. Find relationship among subproblems

→ Dynamic Programming : Abdul Bari

Diff b/w Greedy Method & DP:

↳ Both are used for solving optimization problems
↳ both are different strategies but the purpose is same.

Optimization problems are those which requires minimum result or maximum result.

→ In Greedy Method, we try to follow predefined procedure to get optimal result.

↳ The procedure is known to be optimal, we follow the procedure to get best result like Kruskal's method to find minimum cost spanning tree → always select a minimum cost edge and that gives us best result.

** Mostly DP problems are solved by using recursive formulas.

↳ though we won't use recursion of programming but the formulas are recursive.

→ DP → follows Principle of Optimality

↓

it says that a problem can be solved by taking sequence of decisions to get the optimal solution.

In Greedy → decision taken only one time.

Difference b/w Tabulation and Memoization:

A recursive ex: Memoization

Fibonacci Series:

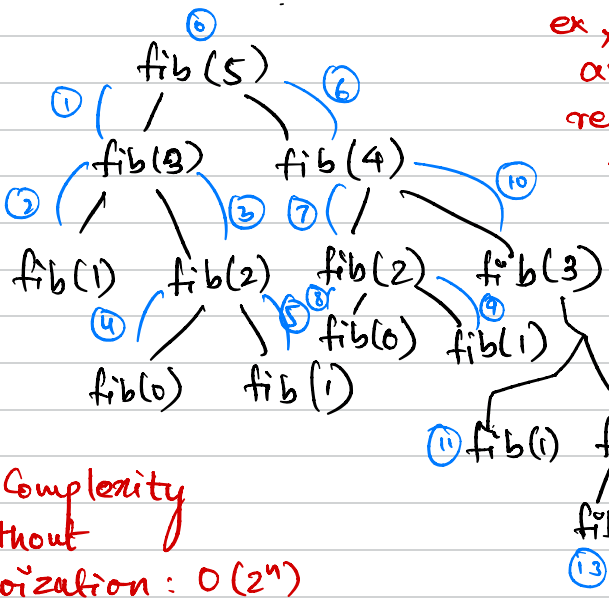
$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-2) + fib(n-1) & \text{if } n > 1 \end{cases}$$

fib. seq = 0, 1, 1, 2, 3, 5, 8, 13 . . .

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

This code follows recursion

for ex: fib(5) = ?



In memoization, in this ex, we create an array to store the results of each fib(n) we need to get the ultimate result.
If we do not use memoization, it will take 15 calls to add each stage and move up to finally get the o/p.

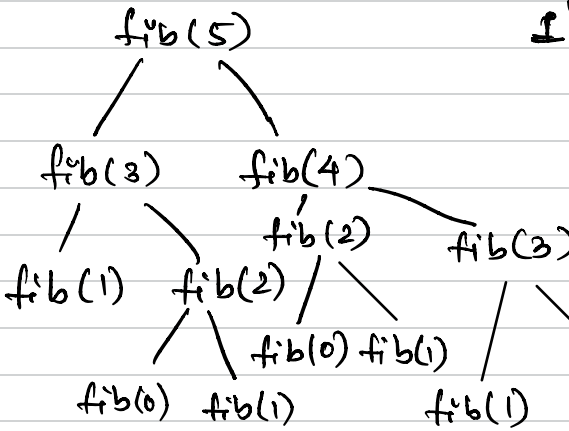
Time Complexity without memoization : $O(2^n)$

Now : memoization :

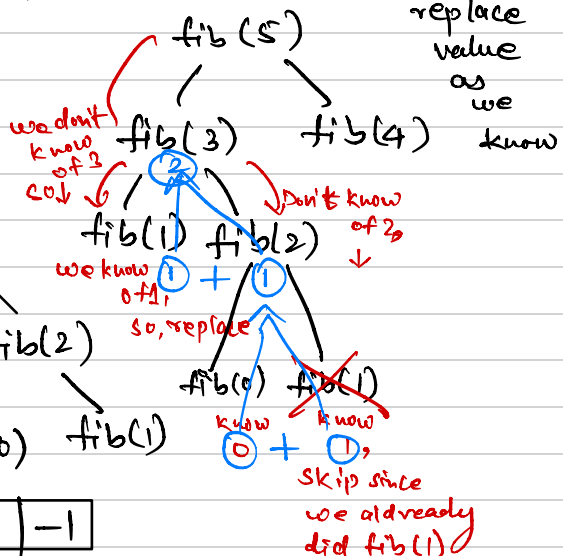
F	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5

OK, so we'll initially fill the array with -1, since we do not know any value right now and we'll replace the values as we know.

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



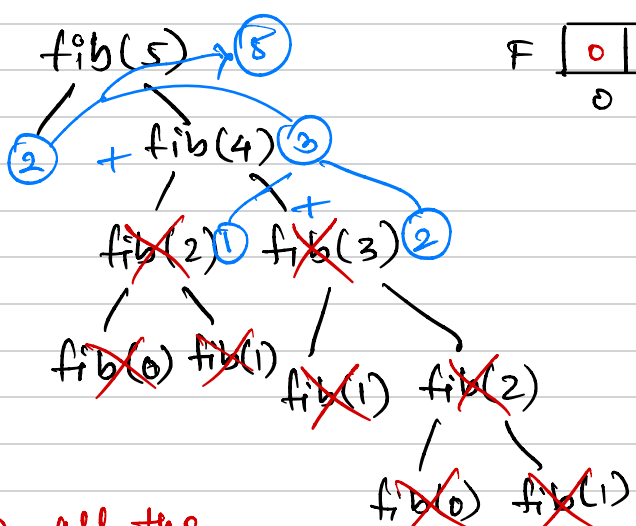
1'Step: We travel down and replace value as we know



∴

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

2nd step: we travel remaining side.



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

Here, the calls made are only 5

Time Complexity = O(n)

Memoization use Top-Down approach.

we skip all the fib(n), we already calculated, just use the value from array.

Tabulation method : use iterative approach

Bottom-Top approach

We create a table and fill it as we get the values to get the ultimate o/p.

```
int fib(int n)
{ if (n<=1)
      return n;
  f[0]=0; f[1]=1;
  for (int i=2; i<=n; i++)
  {
      f[i] = f[i-2] + f[i-1];
  }
  return f[n];
}
```

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

1st step, 2nd, 3rd, 4th, 5th, 6th

Multistage Graph : Problem

- is a directed weighted graph where vertices are divided into stages such as the edges are connecting vertices from one stage to next stage.
  First stage and last stage have only one single vertex to represent the starting point (aka source) or ending point (aka sink of a graph).
- usually useful for representing resource allocation.

↳ Objective of the problem: is to find the shortest path blw starting point and sink giving the minimum cost.

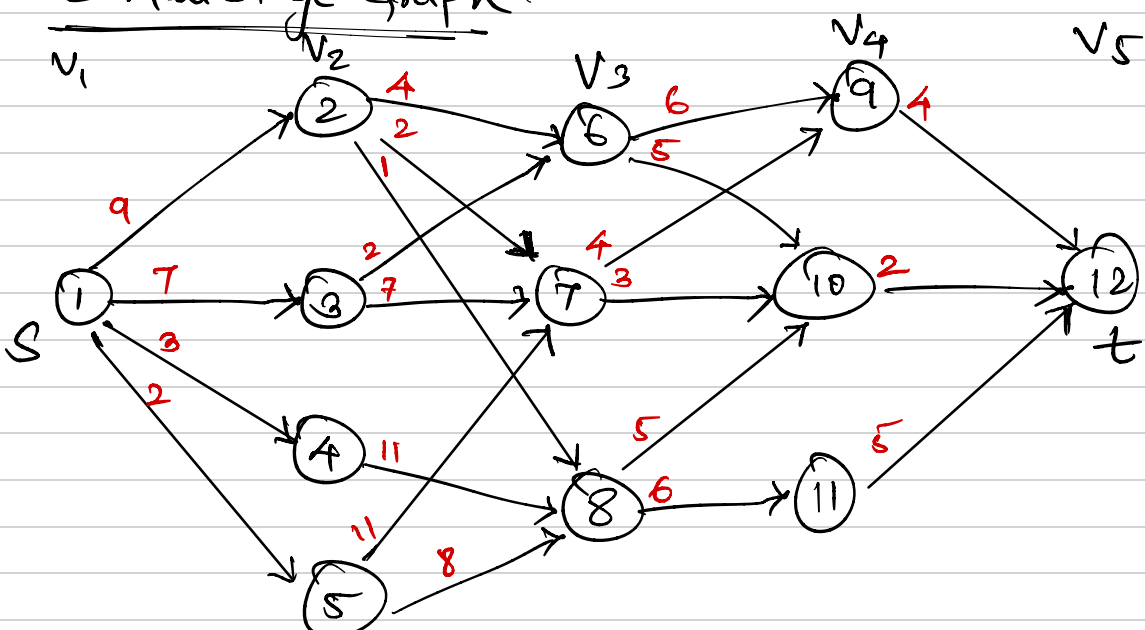
↳ A minimization problem → an optimization problem

↓ can be solved with DP.

1st

↳ Check if the problem can be solved by DP. DP works on principle of optimality stating a problem must be solved in sequence of decisions.

The Multistage Graph:



Since, we travel stage by stage, we have to choose the most optimal route → stating principle of optimality ∴ DP can be used.

↳ We'll use tabulation method to get/fill data in the table.

for each vertex, calculate the cost and to do that start with sink because if we start from earlier stages, we have to calculate all the possible cost (by travelling through all possible edges to the sink).

V	1	2	3	4	5	6	7	8	9	10	11	12
cost						7	5		4	2	5	0
d						10	10		12	12	12	

↳ write down the vertex that gave shortest path (min. cost)

To calculate cost: (stage, vertex)

$cost(5, 12) = 0$

$cost(4, 9) = 4$
$cost(4, 10) = 2$
$cost(4, 11) = 5$

$cost(3, 6) = \min\{c(6,9) + cost(4,9), c(6,10) + cost(4,10)\}$ (vertex, stage vertex)
$= \min\{6+4, 5+2\}$
$= \min(10, 7)$
$= 7$

$cost(3, 7) = \min\{c(7,9) + cost(4,9), c(7,10) + cost(4,12)\}$
$= \min\{(4+4), (3+2)\}$
$= \min(8, 5)$
$= 5$

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	

↳ write down the vertex that gave you shortest path

Continuing

$$c(3,8) = \min\{c(8,10) + cost(4,10), c(8,11) + cost(4,11)\} = \min\{(5+2), (6+5)\}$$
$$= \min(7, 11) = 7$$

$$c(2,2) = \min\{c(2,6) + cost(3,6), c(2,7) + cost(3,7), c(2,8) + cost(3,8)\} = \min\{(4+7), (2+5), (1+7)\}$$
$$= \min(11, 7, 8) = 7$$

$$c(2,3) = \min\{c(3,6) + cost(3,6), c(3,7) + cost(3,7)\} = \min\{(2+7), (7+5)\}$$
$$= \min(9, 12) = 9$$

$$c(2,4) = c(4,8) + c(3,8)$$
$$= 11 + 7 = 18$$

$$c(2,5) = \min\{c(5,7) + cost(3,7), c(5,8) + cost(3,8)\} = \min\{(11+5), (8+7)\}$$
$$= \min(16, 15) = 15$$

$$cost(1,1) = \min\{c(1,2) + cost(2,2), c(1,3) + cost(2,3), c(1,4) + cost(2,4), c(1,5) + cost(2,5)\} = \min\{(9+7), (7+9), (3+18), (2+15)\}$$
$$= \min(16, 16, 21, 17)$$

This is called forward method.

formula we used:

$$cost(i, j) = \min_{\langle j, l\rangle \in E,\ l \in V_{i+1}} \{ c(j, l) + cost(i+1, l) \}$$

(i → stage, j → vertex no., E → set of Edges)

Ex: $cost(2, 3) = \min \{ c(3, 6) + cost(3, 6),\ c(j, l) + cost(i+1, l),\ c(3, 7) + cost(3, 7) \}$

(i, j)

Now, DP:

↳ based on data available:

↳ we take decisions of where to go from each stage.

we use this:

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	

↳ write down the vertex that gave you shortest path

↳ We start from vertex 1. (in forward direction)

(stage ↓, vertex ↓)

$d(1, 1) = 2$ $\quad d(4, 10) = 12$ → shortest path

$d(2, 2) = 7$ ← if we choose 2 out of 2&3

$d(3, 7) = 10$

if we choose 3

d(1,1) = 3
d(2,3) = 6
d(3,6) = 10
d(4,10) = 12

→ Shortest path

Two paths with same cost

Now, Same with Program:

To solve this, we use two formulas:

- Calculate the cost of vertex:

$$cost[i] = \{ c[i][k] + cost[k] \}$$

$c[i][k]$: this symbolizes cost from i to k

i: vertex

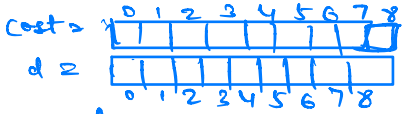
k: $(i+1 \to n)$

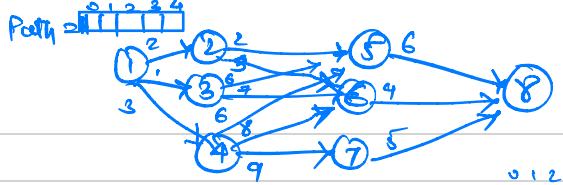
$cost[k]$: cost of k i.e cost of $i+1 \to n$

- $p[i] = d[p[i-1]]$ To find the Shortest Path

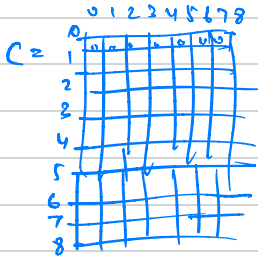
- We will use graph, cost adjancey matrix representing the cost of the edges, arrays as data structures.

- Three arrays: cost, d, path (d: shortest path vertex)





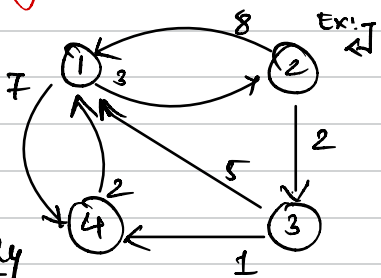
Ex code:




```
main()
{ int stages=4, min;
  int n=8;
  int cost[9], d[9], Path[9];
  int c[9][9]={ {0,0,0,0,0,0,0},
                { fill the values of the row in C },
                {  ,,                              },
                :
                :
                };
  cost[n]=0;
for(int i=n-1 ; i>=1 ; i--)
  { min=32767;
    for(k=i+1; k<n; k++)
    {
      if(c[i][k]!=0 && c[i][k]+c[k]<min)
      { min=c[i][k]+c[k];
        d[i]=k;
      }
    }
    cost[i]=min;
  }
  P[1]=1; P[stages]=n;
for(i=2; i<stages; i++) P[i]=P[d[i-1]];
```

Time Complexity = $O(n^2)$

All Pairs Shortest Path (Floyd-Warshall algorithm)



↳ Objective: we need to find the shortest paths blw all pairs of nodes in a weighted graph.

Floyd-Warshall algo. is particularly useful for graphs with negative weights (as long as there are no negative cycles).

↳ Basically, this problem can also be solved using Dijkstra's algo (which finds all the shortest paths from one single point) but this will lead to time complexity of $O(n^2) \times n$, which is costly.
↳ all points available.

Thus, we use F-W algo. using dynamic Programming.

↳ so, basically, how it works is:

↳ We create matrices of the shortest paths from each point. Suppose we start from ①, we'll check the shortest path to every point directly and indirectly. [Ex: if ①, then ① → ②, & ② → ① → ③ Ex: we mark ①,① as 0, and when there's no path blw any two points, we'll mark it as ∞.
Moreover, once one matrix is completed, we'll move to next matrix and fill it up using the previous matrix. The last matrix will all the shortest paths possible.
To check shortest path, we'll use this formula:

distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])

where, distance = matrix

distance[i][j] = weight of the edge from node 'i' to node 'j'.

(again) if no direct edges connecting 'i' and 'j',

distance[i][j] = ∞

k = intermediate node

So, to summarise the algorithm, we are using a triple nested loop to systematically check whether a given intermediate node 'k' can act as a shortcut to reduce the distance b/w each pair of nodes 'i' and 'j' using the formula. After iterating through all possible intermediate nodes 'k', the 'distance' matrix will contain the shortest paths b/w all pair of nodes.

Time Complexity: $O(n^3)$

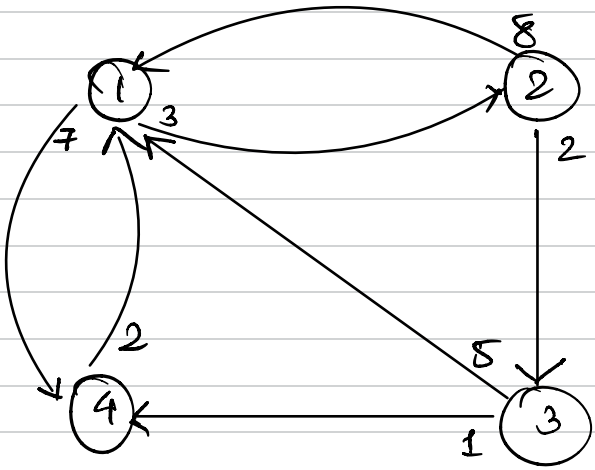
This algo is effective for dense graphs or when all pairs shortest paths are needed.

For sparse graphs with a large no. of nodes, algorithms like Dijkstra's, possibly run from each node, might be more efficient.

↳ It can handle negative weight edges (but not negative cycles), which can be a challange for shortest path algo like Dijkstra's (without modifications).

Calculation

Ex:



→ ① Start with adjacency matrix: $A^0 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

② $A^1 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	5	∞	0

this remains as it is.

$A^1[2,3] = \min(A^0[2,3], A^0[2,1] + A^0[1,3])$

$\Rightarrow \min(2, 8+\infty) = \min(2, \infty)$

$= 2$

$A^1[2,4] = \min(A^0[2,4], A^0[2,1] + A^0[1,4])$

$= \min(\infty, 8+7) = \min(\infty, 15)$

$= 15$

111 sy, we fill all values

Lastly, we'll fill all matrices.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array} \qquad A^3 = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix}$$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

⇒ This is the final matrix with all the shortest paths.

In code: (pseudo code)

```
for(k=1 ; k<=n ; k++)        → for intermediate node
{
    for(i=1 ; i<=n ; i++)
    {
        for(j=1 ; j<=n ; j++)
            A[i,j] = min(A[i,j], A[i,k]+A[k,j]);
    }
}
```

This is for checking all shortest path

In python:

```
def floyd_warshall(weights, n):
    # weights is adjacency matrix of the graph
    # n is no. of vertices
    dist = list(map(lambda i: list(map(lambda j:
                    j, i)), weights))
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i]
                                 [k] + dist[k][j])
    return dist

# Example usage:
# Create a graph with 4 nodes
inf = float('inf')
graph = [
        [0, 3, inf, 7],
        [8, 0, 2, inf],
        [5, inf, 0, 1],
        [2, inf, inf, 0]
    ]

shortest_paths = floyd_warshall(graph, 4)
print(shortest_paths)
```

creating a copy of weights (simpler) or

```
dist = [[j for j in i] for i in weights]
```