Algorithm Complexity and Analysis

Complexity of Algorithm

- measurement of efficiency of an algorithm in terms of time and space
- helps in understanding the behavior of an algorithm as the input size grows
- two types-

1. time complexity
2. space complexity

Time Complexity

- amount of time an algorithm takes to complete as a function of the input size (n)
- algorithm with time complexity $O(n)$ indicates that the time to complete will grow linearly with the input size

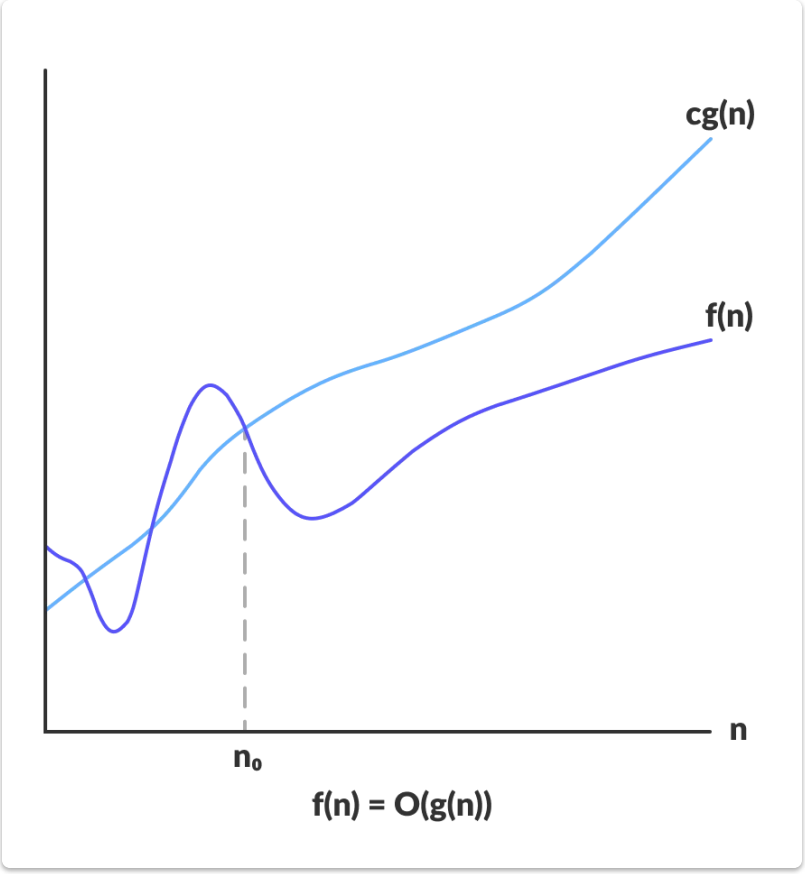
Space Complexity

- amount of memory an algorithm uses as a function of the input size (n).
- algorithm with space complexity $O(1)$ uses a constant amount of memory regardless of the input size.

Big Oh notation

Big-O is a way to express the upper bound of an algorithm's time complexity, since it analyses the worst-case situation of algorithm.

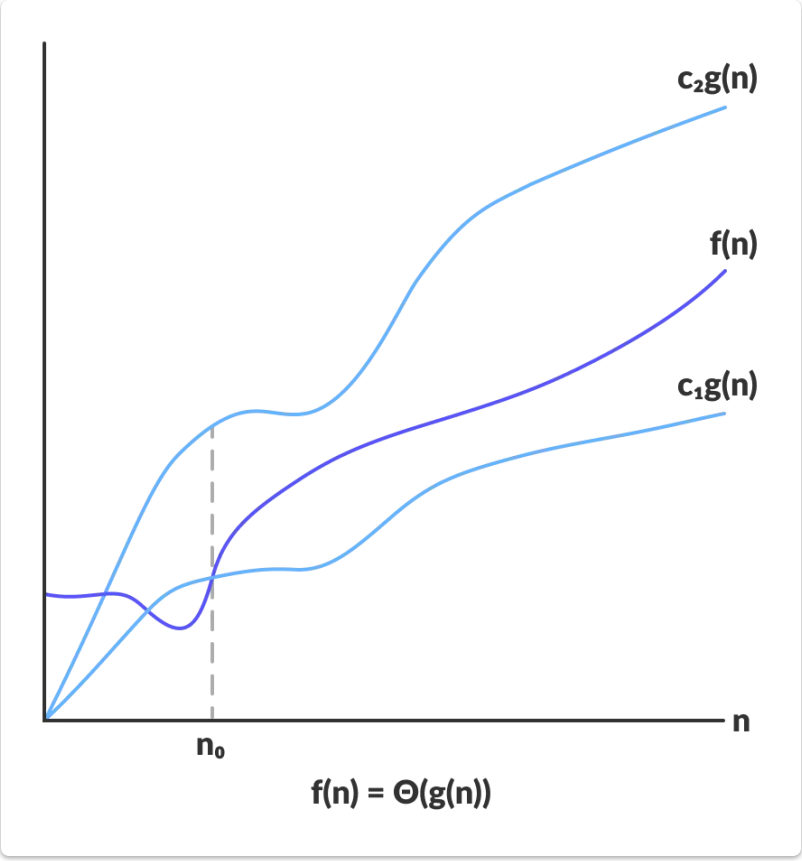
Given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 >= 0$ such that $f(n) <= c * g(n)$ for all $n >= n_0$.



Big Theta notation

Big-O is a way to express the upper and lower bound of an algorithm's time complexity, since it analyses the average-case situation of algorithm.

Given two functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$ and $n_0 >= 0$ such that $c_1 * g(n) <= f(n) <= c_2 * g(n)$ for all $n >= n_0$.



Efficiency Classes

1. Constant Time - $O(1)$:
2. Logarithmic Time - $O(\log n)$:
3. Linear Time - $O(n)$:
4. Linearithmic Time - $O(n \log n)$:
5. Quadratic Time - $O(n^2)$:
6. Cubic Time - $O(n^3)$:
7. Exponential Time - $O(2^n)$:
8. Factorial Time - $O(n!)$:

Proof of $7n^2 + 8n - 9 = \Theta(n^2)$

Statement:

$7n^2 + 8n - 9 = \Theta(n^2)$

Proof:

To show $7n^2 + 8n - 9 = \Theta(n^2)$, we prove both $O(n^2)$ and $\Omega(n^2)$.

Step 1: Prove $7n^2 + 8n - 9 = O(n^2)$

1. Upper bound:

$$7n^2 + 8n - 9 \leq 7n^2 + 8n^2 \quad (\text{since } 8n \leq 8n^2 \text{ for } n \geq 1)$$

$$7n^2 + 8n - 9 \leq 15n^2$$

2. Conclusion:
   $7n^2 + 8n - 9 \leq 15n^2$ for $n \geq 1$.
   Hence, $7n^2 + 8n - 9 = O(n^2)$ with $c = 15$ and $n_0 = 1$.

Step 2: Prove $7n^2 + 8n - 9 = \Omega(n^2)$

1. Lower bound:

$$7n^2 + 8n - 9 \geq 6n^2$$

2. Conclusion:
   $7n^2 + 8n - 9 \geq 6n^2$ for $n \geq 1$.
   Hence, $7n^2 + 8n - 9 = \Omega(n^2)$ with $c = 6$ and $n_0 = 1$.

Combining Both Proofs:

Since $7n^2 + 8n - 9 = O(n^2)$ and $7n^2 + 8n - 9 = \Omega(n^2)$, we conclude:

$$7n^2 + 8n - 9 = \Theta(n^2)$$

Recurrence Relations

Recurrence Relations

- equation that defines a sequence of values
- where each term is formulated as a function of its preceding terms.
- For example, in the recurrence relation $T(n) = 2T(n/2) + 1$:
  - $T(n)$ is defined in terms of $T(n/2)$.
  - The relation involves splitting the problem into two sub-problems of half the size and combining their solutions with an additional constant work $+1$.

Write Recurrence Relation

Fibonacci Series

The Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The Fibonacci sequence is defined as follows:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

This definition provides the recurrence relation for the Fibonacci series:

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

Factorial Function

The recurrence relation for factorial can be written as:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot T(n-1) & \text{if } n > 1 \end{cases}$$

Here:

- $T(n)$ represents the time complexity or the number of operations to compute the factorial of n.
- When $n = 1$, the function returns 1 directly, representing the base case.
- For $n > 1$, the function recursively computes $T(n-1)$ and multiplies it by n.

Strassen's Method of Matrix Multiplication

Let $T(n)$ represent the time complexity of multiplying two $n \times n$ matrices using Strassen's method. The recurrence relation for Strassen's matrix multiplication is:

$$T(n) = 7T(n/2) + O(n^2)$$

where:

- $7T(n/2)$ represents the 7 recursive multiplications of $(n/2) \times (n/2)$ sub-matrices.
- $O(n^2)$ represents the time complexity of the additions and subtractions of matrices of size $n \times n$.

Solve Recurrence Relation using Recursion Tree method

Steps

1. Draw a recursive tree for given recurrence relation
2. Count the total no of levels in the recursion tree
3. Count the total number of nodes in the last level
4. Calculate the cost at each level
5. Sum up the cost of all the levels in the recursive tree

Solve Recurrence Relation using Master method

Master Method

Given a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

for constants $a \geq 1$ and $b > 1$ with $f(n)$ asymptotically positive, the following cases apply:

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a}).$$

Case 2: If $f(n) = \Theta(n^{\log_b a})$, then

$$T(n) = \Theta(n^{\log_b a} \log n).$$

Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ for all n sufficiently large, then

$$T(n) = \Theta(f(n)).$$

In simple terms:

- If $f(n)$ is polynomially smaller than $n^{\log_b a}$, then $n^{\log_b a}$ dominates, and the runtime is $\Theta(n^{\log_b a})$.
- If $f(n)$ is polynomially larger than $n^{\log_b a}$, then $f(n)$ dominates, and the runtime is $\Theta(f(n))$.
- If $f(n)$ and $n^{\log_b a}$ are asymptotically the same, then the runtime is $\Theta(n^{\log_b a} \log n)$.

Sorting Algorithms

Bubble Sort

```
bubbleSort(A, n):
    for i = 0 to n-1:
```

```
        for j = 0 to n-i-2:
            if A[j] > A[j+1]:
                # Swap A[j] and A[j+1]
                temp = A[j]
                A[j] = A[j+1]
                A[j+1] = temp
```

Selection Sort

```
selectionSort(A, n):
    for i = 0 to n-1:
        # Find the minimum element in the unsorted part of the array
        minIndex = i
        for j = i+1 to n-1:
            if A[j] < A[minIndex]:
                minIndex = j
        # Swap the found minimum element with the first element of the unsorted part
        temp = A[minIndex]
        A[minIndex] = A[i]
        A[i] = temp
```

Insertion Sort

```
insertionSort(A, n):
    for i = 1 to n-1:
        key = A[i]
        j = i - 1
        # Move elements of A[0..i-1], that are greater than key,
        # to one position ahead of their current position
        while j >= 0 and A[j] > key:
            A[j + 1] = A[j]
            j = j - 1
        A[j + 1] = key
```

Merge Sort

```
mergeSort(A, p, r):
    if p >= r:
```

```
        return
    q = (p + r) // 2
    mergeSort(A, p, q)
    mergeSort(A, q + 1, r)
    merge(A, p, q, r)

merge(A, p, q, r):
    # Create temporary arrays to hold the left and right subarrays
    n1 = q - p + 1
    n2 = r - q
    left = new array of size n1 + 1
    right = new array of size n2 + 1

    # Copy data to temporary arrays left[] and right[]
    for i = 0 to n1 - 1:
        left[i] = A[p + i]
    for j = 0 to n2 - 1:
        right[j] = A[q + 1 + j]

    # Append sentinel values (infinity) at the end of the temporary arrays
    left[n1] = infinity
    right[n2] = infinity

    # Initial indexes of the temporary arrays
    i = 0
    j = 0

    # Merge the temporary arrays back into A[p..r]
    for k = p to r:
        if left[i] <= right[j]:
            A[k] = left[i]
            i = i + 1
        else:
            A[k] = right[j]
            j = j + 1
```

Quick Sort

```
quickSort(A, p, r):
    if p < r:
        # Partition the array and get the pivot index
```

```
        q = partition(A, p, r)
        # Recursively sort elements before and after partition
        quickSort(A, p, q - 1)
        quickSort(A, q + 1, r)

partition(A, p, r):
    pivot = A[r]
    i = p - 1
    for j = p to r - 1:
        if A[j] <= pivot:
            i = i + 1
            # Swap A[i] and A[j]
            temp = A[i]
            A[i] = A[j]
            A[j] = temp
    # Swap A[i + 1] and A[r] (or pivot)
    temp = A[i + 1]
    A[i + 1] = A[r]
    A[r] = temp
    return i + 1
```

Searching Algorithms

Binary Search

```
do until the pointers low and high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > arr[mid]) // x is on the right side
        low = mid + 1
    else                        // x is on the left side
        high = mid - 1
```

Best Case Complexity:

- The best case occurs when the target element is the middle element of the array on the first attempt.
- The time complexity in this case is $O(1)$.

Worst Case Complexity:

- The worst case occurs when the target element is not in the array or is located at the ends of the array, requiring the maximum number of divisions.
- The time complexity in this case is $O(\log n)$.

Graph Algorithms

Basic Concepts

General Graph Terminology

- Graph: A collection of vertices (or nodes) and edges connecting pairs of vertices.
- Vertex (Node): A fundamental unit of a graph, representing an entity.
- Edge: A connection between two vertices in a graph.
- Adjacent Vertices: Two vertices connected by an edge.
- Path: A sequence of vertices where each adjacent pair is connected by an edge.
- Cycle: A path that starts and ends at the same vertex, with all edges and vertices being distinct (except the start/end vertex).

Undirected Graphs

- Undirected Graph: A graph where edges have no direction; they simply connect two vertices.
- Degree: The number of edges incident to a vertex.
- Connected Graph: An undirected graph in which there is a path between every pair of vertices.
- Complete Graph: A complete graph is an undirected graph in which every pair of distinct vertices is connected by a unique edge.
- Tree: A connected undirected graph with no cycles.
- Forest: A disjoint set of trees.

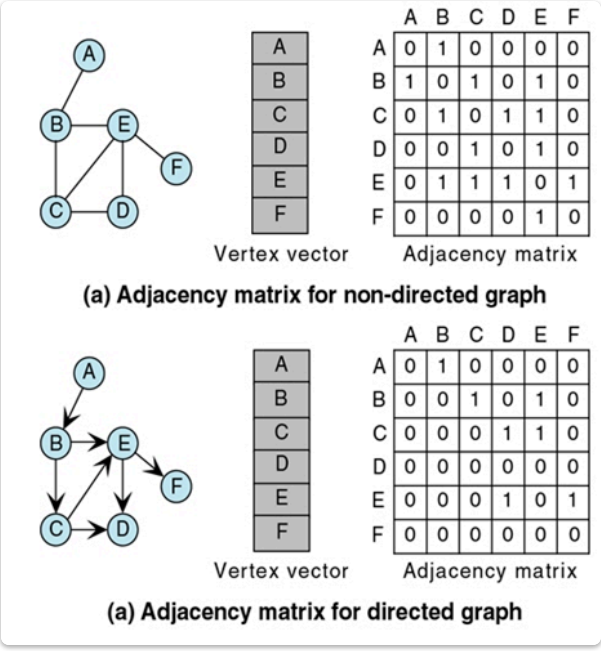
Directed Graphs (Digraphs)

- Directed Graph (Digraph): A graph where each edge has a direction, going from one vertex to another.
- In-Degree: The number of incoming edges to a vertex.
- Out-Degree: The number of outgoing edges from a vertex.
- Strongly Connected Graph: A directed graph in which there is a directed path from any vertex to every other vertex.

- Weakly Connected Graph: A directed graph where replacing all directed edges with undirected edges results in a connected undirected graph.
- Directed Acyclic Graph (DAG): A directed graph with no cycles.

Adjacency Matrix

An adjacency matrix is a square matrix of N x N size where N is the number of nodes in the graph and it is used to represent the connections between the edges of a graph.



Spanning Trees

Difference in Kruskal's and Prim's Algorithm

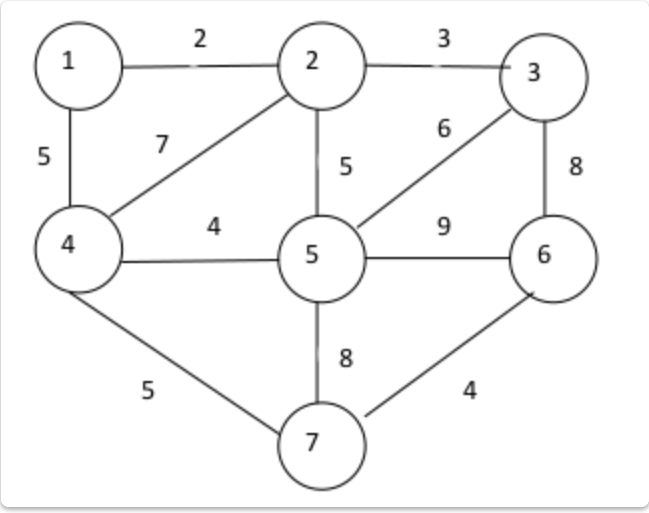
Aspect	Kruskal’s Algorithm	Prim’s Algorithm
Selection Criterion	Always selects an edge (u, v) of minimum weight to find MCST.	Always selects a vertex (say, v) to find MCST.
Adjacency Requirement	It is not necessary to choose adjacent vertices of already selected vertices.	It is necessary to select an adjacent vertex of already selected vertices.
Intermediate Steps	There may be more than one connected component at intermediate steps.	There will be only one connected component at intermediate steps.
Time Complexity	$O(\|\|E\|\| \log \|\|V\|\|)$	$O(\|\|V\|\|^2)$

Kruskal's MCST Algorithm

Input: A undirected connected weighted graph $G = (V, E)$.
Output: A minimum cost spanning tree $T = (V, E')$ of G.

1. Sort the edges E in order of increasing weight.
2. Initialize the set A to be empty: $A \leftarrow \emptyset$.
3. For each vertex v in $V[G]$:
    1. `MAKE_SET(v)`
4. For each edge (u, v) in E, taken in increasing order of weight:
    1. If `FIND_SET(u) ≠ FIND_SET(v)`:
        1. $A \leftarrow A \cup \{(u, v)\}$
        2. `MERGE(u, v)`
5. Return A.

Time complexity: `O(E log V)`.



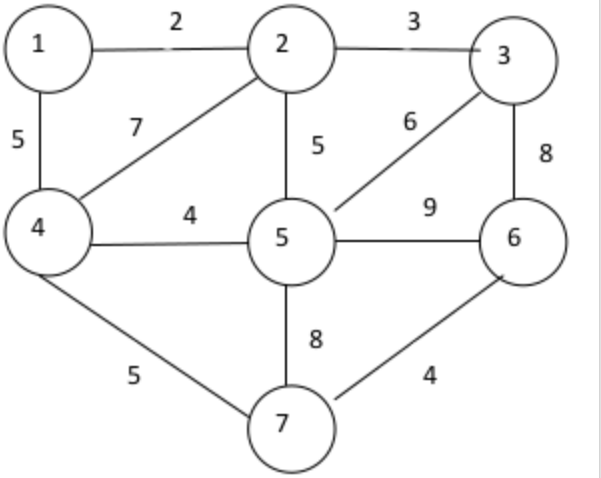
STEP	EDGE CONSIDERED	CONNECTED COMPONENTS	SPANNING FORESTS (A)
Initialization	___	{1}{2}{3}{4}{5}{6}{7} (using line 3-4)	(1) (2) (3) (4) (5) (6) (7)
1.	(1, 2)	{1, 2},{3},{4},{5},{6},{7}	(1) – (2) (3) (4) (5) (6) (7)
2.	(2, 3)	{1,2,3},{4},{5},{6},{7}	(1)–(2)–(3) (4) (5) (6) (7)
3.	(4, 5)	{1,2,3},{4,5},{6},{7}	(1)–(2)–(3) (6) (7) (4)–(5)
4.	(6, 7)	{1,2,3},{4,5},{6,7}	(1)–(2)–(3) (4)–(5) (6) (7)
5.	(1, 4)	{1,2,3,4,5},{6,7}	(1)–(2)–(3) | (4)–(5) (6) (7)
6.	(2, 5)	Edge (2,5) is rejected, because its end point belongs to same connected component, so create a cycle.	
7.	(4, 7)	{1,2,3,4,5,6,7}	(1)–(2)–(3) | (4)–(5) (6) (7)

Prim's MCST Algorithm

Input: An undirected connected weighted graph $G = (V, E)$.
Output: A minimum cost spanning tree $T = (V, E')$ of G.

1. Initialize the set $T \leftarrow \emptyset$ // T contains the edges of the MST
2. Initialize the set $A \leftarrow \{\text{Any arbitrary member of } V\}$
3. While $A \neq V$:
    1. Find an edge (u, v) of minimum weight such that $u \in V - A$ and $v \in A$
    2. $T \leftarrow T \cup \{(u, v)\}$
    3. $A \leftarrow A \cup \{u\}$
4. Return T

Time complexity: `O(V^2)`.



STEP	EDGE CONSIDERED (4, v)	CONNECTED COMPONENTS (Set A)	SPANNING FORESTS (set T)
Initialization	—	{1}	(1)
1	(1,2)	{1,2}	(1)–(2)
2	(2,3)	{1,2,3}	(1)–(2)–(3)
3	(1,4)	{1,2,3,4}	(1)–(2)–(3) | (4)
4.	(4,5)	{1,2,3,4,5}	(1)–(2)–(3) | (4)–(5)
5	(4,7)	{1,2,3,4,5,7}	(1)–(2)–(3) | (4)–(5) \ / (7)
6	(6,7)	{1,2,3,4,5,6,7}	(1)–(2)–(3) | (4)–(5) (6) \ / (7)

Traversal

DFS

```
DFS(graph, start_node):
    Create an empty stack S
    Create an empty set visited

    Push start_node onto S
```

```
    While S is not empty:
        node = Pop S
        If node is not in visited:
            Add node to visited
            For each neighbor in graph[node]:
                If neighbor is not in visited:
                    Push neighbor onto S
```

Time Complexity: $O(V + E)$

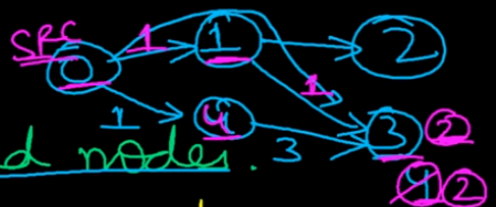
BFS

```
BFS(graph, start_node):
    Create an empty queue Q
    Create an empty set visited

    Enqueue start_node onto Q
    Add start_node to visited

    While Q is not empty:
        node = Dequeue Q
        For each neighbor in graph[node]:
            If neighbor is not in visited:
                Add neighbor to visited
                Enqueue neighbor onto Q
```

Time Complexity: $O(V + E)$

Dijkstra's Shortest Path Algorithm

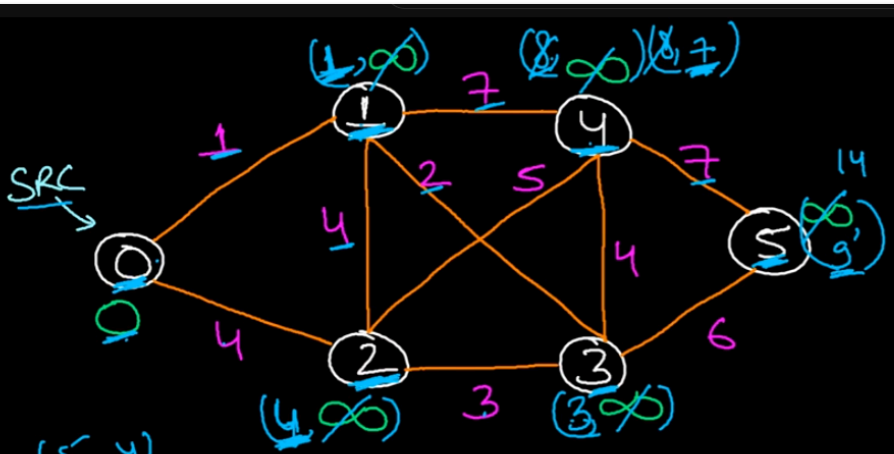
ALGORITHM



- Maintain a SET of processed nodes.
- ASSIGN all Nodes with distance value = ∞ except SOURCE Node (0).
- Repeat Following :- (unless all vertices are included)
  1. Pick min. value vertex which is not already processed
  2. Include this selected Node in processed SET.
  3. Update all the adjacent Node distances.

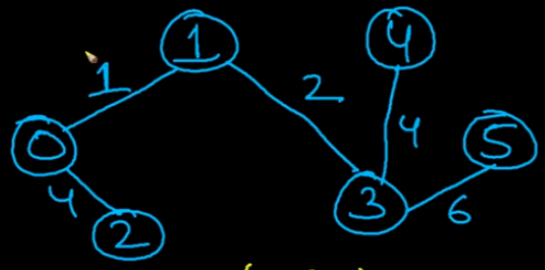
(V-1) times

If (New distance < old distance) then UPDATE
ELSE SKIP



SET

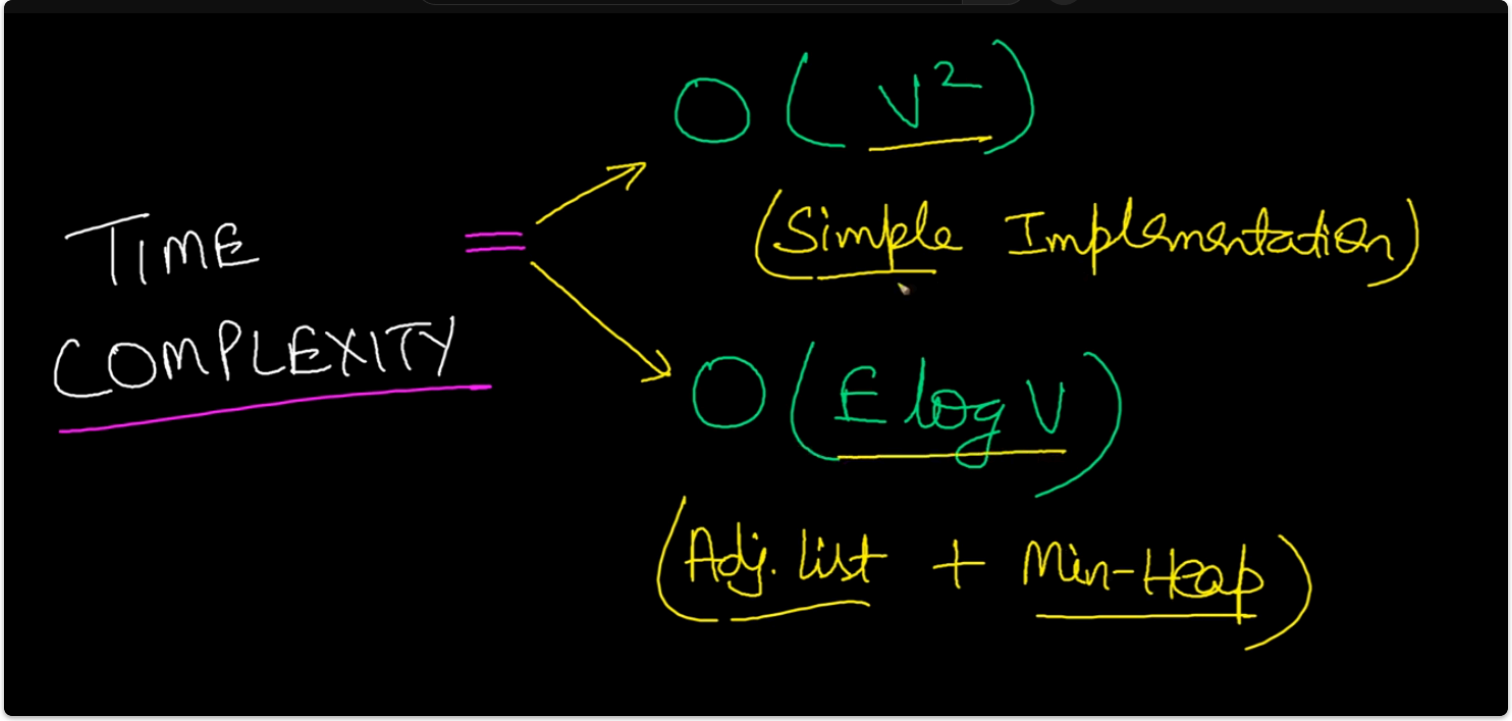
0, 1, 3, 2, 4



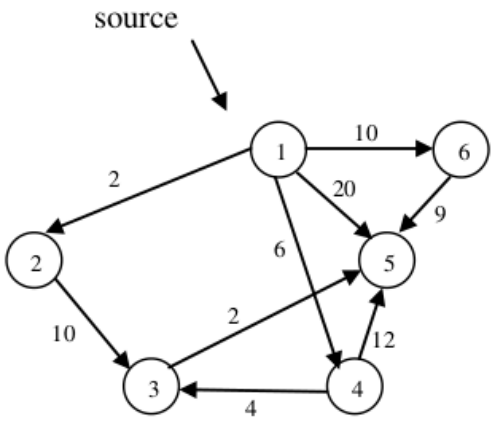
(SPG)

	0	1	2	3	4	5
Value	0	∞ 1	∞ 4	∞ 3	∞ 8 7	∞ 9

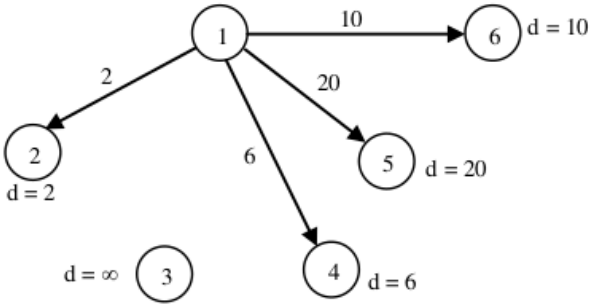
NOTE: We need only (V-1) STEPS like in MST algo.



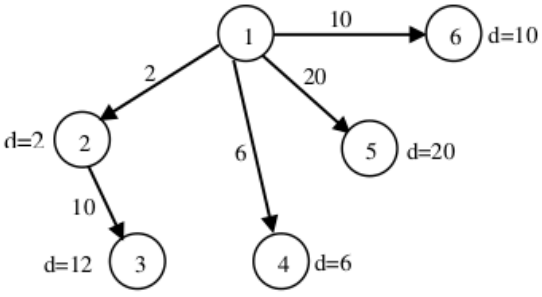
Example2: Apply dijkstra's algorithm on the following digraph (1 is starting vertex)



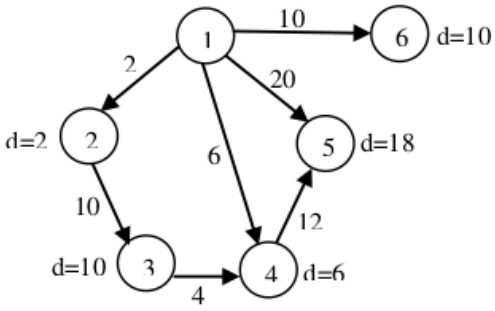
S={ }



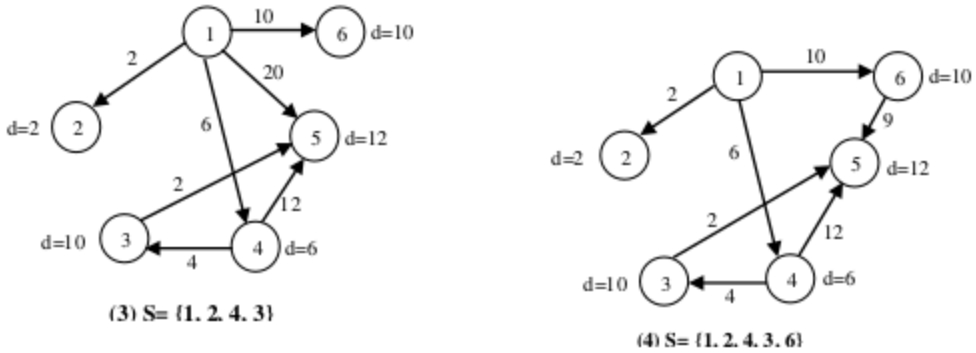
(initial) S={1}



(1) S= {1, 2}



(2) S={1, 2, 4}



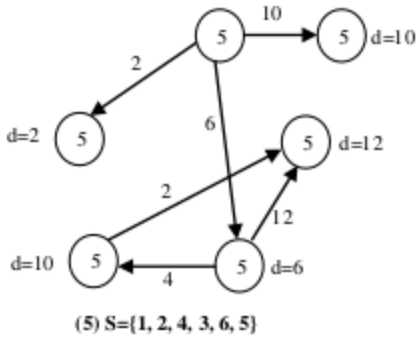


Figure 2: Stages of Dijkstra's algorithm

Iterations	S	Q (Priority Queue)						EXTRACT_ MIN(Q)
		d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	
Initial	{ }	0	∞	∞	∞	∞	∞	1
1	{1}	**[0]**	2	∞	6	20	10	2
2	{1,2}		[2]	**12**	**6**	**20**	**10**	4
3	{1,2,4}			10	[6]	18	10	3
4	{1,2,4,3}			**[10]**		**18**	**10**	6
5	{1,2,4,3,6}					**12**	**[10]**	5
	{1,2,4,3,6,5}					**[12]**		

Table2: Computation of Dijkstra's algorithm on digraph of Figure 2

Backtracking

- problem-solving algorithm that uses a brute force approach for finding the desired output.

- Brute force approach tries out all the possible solutions and chooses the desired/best solutions.
- backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

Example Problems:

1. N-Queens Problem: Place n queens on an $n \times n$ chessboard such that no two queens threaten each other. The algorithm tries placing a queen in each row and backtracks if it finds that placing a queen in a certain position leads to a conflict.
2. Sudoku Solver: Fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids contain all of the digits from 1 to 9. The algorithm tries placing each number in each cell and backtracks if it finds that placing a number in a certain cell violates the rules of Sudoku.

Branch and Bound

- general algorithmic technique for solving optimization problems.
- divide the problem (branch) into smaller subproblems and evaluate bounds to discard subproblems that cannot lead to a better solution (bound).

Example Problem:

Knapsack Problem: Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by filling a knapsack with a limited weight capacity. Unlike the Greedy approach, B&B can be used to solve the 0/1 Knapsack Problem optimally by exploring all possible combinations of items and using bounds to prune non-promising branches.

Greedy

- builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit
- This technique assumes that by choosing a local optimum at each step, the overall solution will be a global optimum.

Example Problems:

1. Fractional Knapsack Problem: Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by filling a knapsack with a limited weight capacity. Items can be divided to maximize the total value.
2. Activity Selection Problem: Given a set of activities with start and end times, select the maximum number of activities that do not overlap.

Miscellaneous

Euclid's Algorithm for finding GCD

```c
// a < b
int gcd(int a, int b)
{
        if (a == 0)
                return b;
        return gcd(b % a, a);
}
```

Time Complexity: O(log min(a, b))

Horner's rule for polynomial evaluation

Algorithm

1. Input:
   - A list of coefficients of the polynomial $a_n, a_{n-1}, \ldots, a_1, a_0$.
   - A value x at which the polynomial is to be evaluated.
2. Initialize:
   - Set $result$ to the leading coefficient a_n.
3. Iterate:
   - For each coefficient a_{i-1} from a_{n-1} to a_0 (in reverse order):
     - Update $result$ as $result \times x + a_{i-1}$.
4. Output:
   - The value of $result$ is the evaluated polynomial at x.

Pseudocode

1. Start
2. Input: List of coefficients $[a_n, a_{n-1}, \ldots, a_1, a_0]$ and value x

3. Initialize: $result = a_n$
4. For i from $n-1$ down to 0:
    - $result = result \times x + a_i$
5. End For
6. Output: $result$
7. End

This algorithm efficiently evaluates a polynomial at a given point x using Horner's method, reducing the computational complexity to $O(n)$.

Binary Exponentiation

Right-to-Left Binary Exponentiation

1. Input: Base a, Exponent n
2. Initialize: $result = 1$, $base = a$
3. While $n > 0$:
    - If $n \mod 2 == 1$ (if n is odd):
        - $result = result \times base$
    - $base = base \times base$
    - $n = \lfloor n/2 \rfloor$ (integer division)
4. End While
5. Output: $result$
6. End

This approach processes the exponent from least significant bit (right) to most significant bit (left).

Left-to-Right Binary Exponentiation

1. Input:
    - Base a
    - Binary string of length s representing the exponent n as an array $A[s]$
2. Initialize:
    - Set $result = a$
3. Iterate:
    - For i from 1 to $s-1$:
        - $result = result \times result$
        - If $A[i] = 1$:

- $result = result \times a$

4. Output:
    - Return $result$

Strassen's Algorithm to multiply two matrices

Karatsub's method

Fibonacci series

```c
#include <stdio.h>

void fibonacciIterative(int n) {
    int t1 = 0, t2 = 1, nextTerm;

    printf("Fibonacci Series: %d, %d", t1, t2);

    for (int i = 1; i <= n-2; ++i) {
        nextTerm = t1 + t2;
        printf(", %d", nextTerm);
        t1 = t2;
        t2 = nextTerm;
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    fibonacciIterative(n);
    return 0;
}
```

language-c

```c
#include <stdio.h>

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

void printFibonacciSeries(int n) {
    printf("Fibonacci Series: ");
    for (int i = 0; i < n; ++i) {
```

language-c

```
        printf("%d", fibonacciRecursive(i));
        if (i < n - 1) {
            printf(", ");
        }
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of terms: ");
    scanf("%d", &n);
    printFibonacciSeries(n);
    return 0;
}
```

Add two matrices

```
AddMatrices(matrix1, matrix2):
    rows = number of rows in matrix1
    columns = number of columns in matrix1

    Create a new matrix result with dimensions [rows][columns]

    For i from 0 to rows - 1:
        For j from 0 to columns - 1:
            result[i][j] = matrix1[i][j] + matrix2[i][j]

    Return result
```

Note

Search for complexity analysis once in textbook