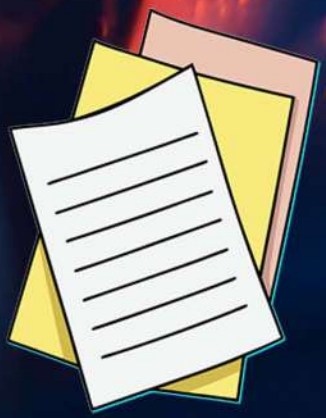
GeeksforGeeks

GATE फर्स्ट

CSE

ALGORITHMS

SHORT NOTES



ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

ALGORITHMS

GATE फर्रे

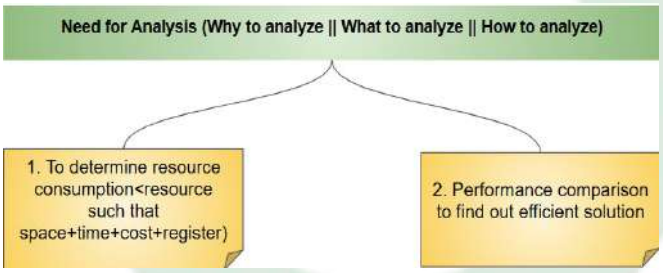
Module 1: Analysis of Algorithm

Aim : The goal of analysis of algorithms is to compare algorithms mainly in terms of running time but also in terms of other factors like memory, developer effort.

Need for Analysis (Why to analyze || What to analyze || How to analyze)

1. To determine resource consumption <resource such that space+time+cost+register> Resources may differ from domain to domain.

2. Performance comparison to find out efficient solution



Methodology of algorithm

- Depends on language
- Operating system
- Hardware (CPU, processor, memory, Input/output)

Types of analysis

1. **Aposteriori analysis(platform dependent)** : It gives exact value in real units.
2. **Apriori analysis(platform independent)** : It allows us to calculate the relative efficient performance of two algorithms in a way such that it is platform independent. It will not give real values in units.

Asymptotic Notations

θ-Notation

Let f(n) and g(n) be two positive functions
f(n) = θ(g(n)) if and only if
f(n) ≤ c1 . g(n) and f(n) ≥ c2 . g(n)
∀ n ≥ n0 such that there exists three positive constant c1 > 0, c2 > 0 and n0 ≥ 1

O-Notation [Pronounced "big-oh"]

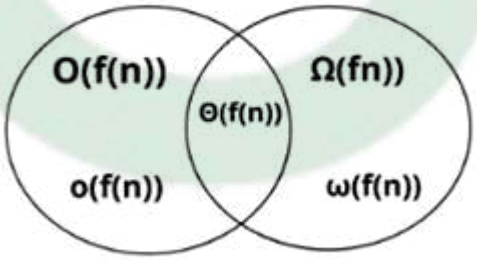
Let f(n) and g(n) be two positive functions
f(n) = O(g(n)), if and only if
f(n) ≤ c . g(n), ∃ n, ≥ n0
such that $ two positive constants c > 0, n0 ≥ 1.

Ω-Notation: [Pronounced "big-omega"]

Ω notation provides an asymptotic lower bound for a given function g(n), denoted by Ω(g(n)). The set of functions f(n) = Ω(g(n)) if and only if f(n) ≥ c . g(n), ∀ n ≥ n0 such that two positive constants c > 0, n0 ≥ 1.

Analogy between real no & asymptotic notation : Let a, b are two real no & f, g two positive functions
If f(n) is O(g(n)) : a ≤ b

- If f(n) is Ω(g(n)) : a ≥ b
- If f(n) is Θ(g(n)) : a = b
- If f(n) is o(g(n)) : a < b
- If f(n) is ω(g(n)) : a > b



Rate of growth of function

[Highest] 2^{2^n} —> n! —> 4^n —> 2^n —> n^2 —> nlogn —>log(n!) —> n —> $2^{\log n}$ —> $\log^2 n$ —> log logn —> 1 [lowest]

Trichotomy property:

For any two real numbers (a, b) there must be a relation between them
(a > b, a < b, a = b)

Asymptotic notation does not satisfy trichotomy property
ex: f(n) = n, g(n) = n * |sin(n)|, n > 0
∴ These two functions cannot converge

Example

1. Loop

```
for ( i = 1; i <= n; i++) {
x=y+z;
}
```

T(n) = O(n)

2. Nested loop

```
for(i=1; i<=n; i++){
for(j=1; j<=n; j++){
k= k+1;
}
}
```

$T(n) = O(n^2)$

3. Logarithm

```
for(i=1; i<=n ; i *= 2){
k=k+1;
}
```

```
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j *= 2) {
      printf("GFG");
   }
}
```

T(n) = O(n log n)

4. Linear recursion:

```
void fun(int n) {
   if (n > 0) {
      fun(n - 1);
   }
}
```

T(n) = T(n-1) + C
T(n) = O(n)

5. Recursive with logarithmic loop

```
void fun(int n) {
   if (n > 1) {
      fun(n / 2);                  // Recursive call first
      for (int i = 1; i <= n; i *= 2) {
         printf("Hello\n");
      }
   }
}
```

T(n) = T(n/2) + O(log n)

$T(n) = O(\log^2 n)$

6. Time is infinite

```
c= 0;
while(1)
C += 1;
```

7. Mutually Exclusive Loops

1. **For** i←1 **to** n: C=C+1;
2. **For** j←1 **to** m: K=K∗2;

Time = O(max(n, m))

8. Nested loop analysis

```
for (i = 1; i <= n; ++i) // Executes 'n' times
for (j = 1; j <= n; ++j) // Executes 'n' times
for (k = n/2; k <= n; k += n/2) // Executes 2 times (n/2, n)
C = C + H;
```

Time $= O(n^2)$

```
for(i=1;i<n;i = i+a)
```

Time : O(n/a) = O(n)

9. For a loop with a multiplicative increment:

for (i=1;i<=n;i=i∗2): This loop's complexity is $Log_2 n$.
for (i=1;i<=n;i=i∗3): This loop's complexity is $Log_3 n$.
for (i=1;i<=n;i=i∗a);

General formula: The time complexity is $O(\log_a n)$.

```
k=1, i=1
while(k<=n){
i++;
k=k+i;
}
```

Time complexity : $T(n) = O(\sqrt{n})$

```
for (i = n; i >= 2; i = sqrt(i))
```

Time complexity: **O(log log n)**.

10. for (i = 2; i <= n; i++) { // log logn

```
for (j = 1; j <= i; j++) {
for (k = 1; k <= n; k +=j) { // n/j ,  where j = 1,2,3,..n
x = y + z;
...
}
}
}
```

T(m) = O(log logn (n logn))

Master Theorem:

Let f(n) is a positive function and T (n) is defined recurrence relation:

$T(n) = aT(n/b) + f(n)$

Where a >= 1 and b > 1 are two positive constants.

Case 1:

If $f(n) = O(n^{(\log_b a - \epsilon)})$ for some constant $\epsilon > o$ 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 constant $\epsilon > 0$, and if a $f(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then **$T(n) = \theta(f(n))$**

Master theorem for subtract and conquer recurrence:

Let T(n) be a function defined on possible n:

$T(n) = aT(n-b) + f(n)$, if $n > 1$

$T(n) = C$, if $n <= 1$

For some constant C, $a>0$, $b>0$, and $f(n) = O(n^d)$

1. $T(n) = O(n^d)$, if $a < 1$
2. $T(n) = O(n^d+1)$, if $a = 1$
3. $T(n) = O(n^d * a^{(n/b)})$, if $a > 1$

Common Recurrence Relation

Recurrence relation	Time complexity
T (n) = C; n = 2 T (n) = 2 T(√n) + C; n > 2	O(logn)
T (n) = C; n = 2 T (n) = T(n – 1) + C ; n > 2	O(n)
T (n) = C; n = 1 T (n) = T(n – 1) + n + C ; n > 2	O(n^2)
T(n) = C ; n = 1 T(n) = T(n-1) * n + C ; n > 2	O(n^n)
T(n) = C ; n = 1 T(n) = 2T(n/2) + C ; n > 1	O(n)
T(n) = C ; n = 1 T(n) = 2T(n/2) + C ; n > 2	O(nlogn)
T(n) = C ; n = 1 T(n) = T(n/2) + C ; n > 1	O(logn)
T (n) = 1; n = 2 T(n) = T(n) + C; n > 2	Θ(loglogn)
T(n) = T(n/2) + 2^n if n > 1	O(2^n)

Analogy between real no & asymptotic notation

Let a, b are two real no & f, g two positive functions

- If f(n) is O(g(n)) : **a ≤ b** (f grows slower than some multiple of g)
- If f(n) is Ω(g(n)) : **a ≥ b** (f grows faster than some multiple of g)
- If f(n) is Θ(g(n)) : **a = b** (f grows at same rate of g)
- If f(n) is o(g(n)) : **a < b** (f grows slower than any multiple of g)
- If f(n) is ω(g(n)) : **a > b** (f grows faster than any multiple of g)

Analysis

1. f(n) = n!
   n! <= $c*n^n$: n >= 2
   n! = $O(n^n)$ with c = 1, n0 = 2.
   using stirling's approximation : $n! \approx \sqrt{(2n\pi)}\, n^n * e^{-n}$

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

Discrete Properties of Asymptotic Notation

Property	Big Oh(O)	Big Omega()	Theta()	Small oh(o)	Small omega ()
Reflexive	✓	✓	✓	×	×
Symmetric	×	×	✓	×	×
Transitive	✓	✓	✓	✓	✓
Transpose symmetric	✓	✓	×	✓	✓

Analogy between real no & asymptotic notation

Let a, b are two real no & f, g two positive functions

- If f(n) is O(g(n)) : **a ≤ b** (f grows slower than some multiple of g)
- If f(n) is Ω(g(n)) : **a ≥ b** (f grows faster than some multiple of g)
- If f(n) is Θ(g(n)) : **a = b** (f grows at same rate of g)
- If f(n) is o(g(n)) : **a < b** (f grows slower than any multiple of g)
- If f(n) is ω(g(n)) : **a > b** (f grows faster than any multiple of g)

Analysis

1. f(n) = n!
   n! <= c*n^n : n >= 2
   n! = O(n^n) with c = 1, n0 = 2.
   using stirling's approximation : n! ≈ √(2nπ) n^n * e ^ -n

Trichotomy property:

For any two real numbers (a, b) there must be a relation between them
(a > b, a < b, a = b)

Asymptotic notation does not satisfy trichotomy property

Ex: f(n) = n, g(n) = n ^ |sin(n)|, n > 0
∴ These two functions cannot converge

1. **Reflexive**
   f(n) = O(f(n))
   f(n) = Ω(f(n))
   f(n) = Θ(f(n))

2. **Symmetric**
   f(n) = Θ(g(n)), iff g(n) = Θ(f(n))

3. **Transitive**
   f(n) = Θ(g(n)) & g(n) = O(h(n))
   f(n) = O(h(n))
   Note : Ω and Θ also satisfy transitivity

4. **Transpose Symmetric**
   f(n) = O(g(n)) iff g(n) = Ω(f(n))

Best case(n) ≤ Average case(n) ≤ Worst case(n)

Space Complexity

Space required by algorithm to solve an instance of the problem, excluding the space allocated to hold input.
Space complexity : C + S(n)
C - Constant space
S(n) - Additional space that depends on input size n

Space Complexity VS Auxiliary space

Space Complexity = Total space used **including** input
Auxiliary space = Extra space used **excluding** the input

Space Complexity (Memory)

Example:

```
Algo sum(A, n){
int n, a[], i;
int sum=0;
for(i=0; i<n; i++)
sum = sum+arr[i];
}
```

Time complexity : O(n)

Space complexity : O(1)

```
Algo swapNum(int a, int b){
int temp = a;
a = b;
b = temp;
}
```

Time complexity : O(1)

Auxiliary space : O(1)

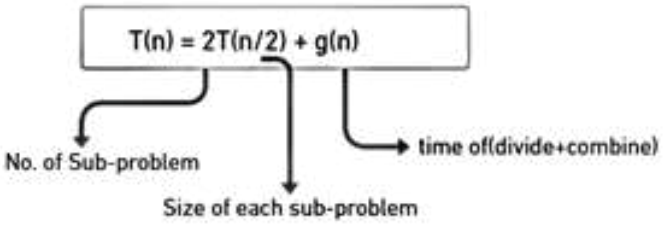
Module 2 : Divide and Conquer (DAC)

Note : In DAC, divide and conquer is mandatory but combine is optional.

- Algorithm DAC(A, 1, n)
- if(small(1, n)
- return (S(A, 1, n);
- Else
- m ←– Divide(1, n)
- S1 ←– DAC(A, 1, m)
- S2 ←– DAC(A, m+1, n)
- Combine (S1, S2);

Time Complexity for DAC Problem
T(n) = F(n), if n is small,

T(n) = 2T(n/2) + g(n) : if n is large



Generalized Form : T(n) = aT(n/b) + g(n)
g(n) – +ve, a > 0, b > 0;

I Symmetric form

T(n) = aT(n/b) + g(n)
a : number of sub-problem
b : size shrink factor(each sub-problem n/b)
g(n) : cost to divide and combine
eg. Merge sort : T(n)=2·T(2n)+O(n)

II Asymmetric form 1

T(n) = T(αn) + T((1-α)n) + g(n) provide that : 0 < α < 1
eg: T(n) = T(n/3) + T(2n/3) + g(n)

III Asymmetric form 2

T(n) = T(n/2) + T(n/4) + g(n)
ex. Quick sort with asymmetric partitioning

Divide and conquer problem

1. Finding minimum and maximum

T(n) = 2T(n/2) + 2 , n>2
Time complexity using DAC : T(n) = O(n)
Space complexity using DAC :
T(n) = O(logn)

2. Power of an element

Recurrence relation
T(n) = 1 if n = 1
T(n) = T(n/2) + C if n > 1
Time complexity : O(logn)
Space complexity : O(logn)

3. Binary Search Algorithm

Note: Provided that list of elements already sorted.
T(n) = c : n = 1
T(n) = a + T(n/2) : n > 1
Time complexity : T(n) = O(logn)
Space complexity : T(n) = O(1)

4. Merge Sort Algorithm

Comparison based sorting
Stable sorting but outplace
Recurrence relation: T(n) = c if n = 1
T(n) = T(n/2) + T(n/2) + cn if n > 1

Time complexity = O(n log n)
= Ω(n log n)
= Θ(n log n)
Space complexity : O(n + logn) = O(n)

5. Quick Sort Algorithm

Best Case / Average Case
T(n) = 1 ; if n = 1.
T(n) = 2T(n/2) + n + C, if n>1
Time complexity : O(n logn)
Worst case : T(n) = n + T(n-1) + C ; if n > 1
Note: Quick sort behaves in worst case when element are already sorted
Time complexity : O(n^2)

6. Matrix Multiplication

I. Using DAC: T(n) = 8T(n/2)+O(n^2), for n>1

T(n) = O(n^3)

II. Strassen's matrix multiplication :

T(n) = 7T(n/2) + a.n^2, for n > 1
Time complexity : O(n^2.81) (by Strassen's)
Time complexity : O(n^2.37) (by Coppersmith and winograd)
Space complexity :

7. Selection Procedure (Find kth smallest on given an array of element and integer k)
Time complexity : O(n^2)
Space complexity : O(n)

8. Counting Number of Inversion (An inversion in an array is a pair(i, j) such that **i<j and arr[i] > arr[j])**
Time complexity : O(nlogn)
Space complexity : O(n)(due to merges)

9. Closest pair of points (Find the minimum Euclidean distance between any two points in a 2D plane.)
Recurrence relation => T(n) = 2T(n/2) + O(n)
Time complexity = O(nlogn)
Sorting copies (x-sorted, y-sorted): O(n)
Auxiliary space (recursion stack): O(log n)
Space complexity : O(logn)

10. Convex hull (Find smallest convex polygon that encloses a point in a 2D plane)
T(n) = 2T(n/2) + O(n)
Time complexity = O(nlogn)
Space complexity : O(logn)

Note: In GATE exam if merge sort given then always consider outplace.

- If array size is very large, merge sort preferable.
- If array size is very small, then prefer insertion sort.
- Merge sort is a stable sorting technique.

11. Longest Integer multiplication(LIM)

int data type, can stores digits max 32767
long int data types, can store 4B/8B (8-10 digits number) not more than that.
Solution : We can store long integer multiplication in an array.
T(n) = 4T(n/2) + b.n ; if n>1
Time complexity : O(n^2)
Space complexity : O(logn)

Karatsuba optimization :

T(n) = 3T(n/2) + bn ; if n > 1
Time complexity : O(n^1.58)
Karatsuba is better but still not fast enough

Toom cook optimization :

Toom-Cook is a generalization of **Karatsuba's algorithm** that splits the input numbers into **three parts**.

Toom-3 (3-way split)

I. T(n) = 9T(n/3) + bn.
Time complexity : O(n^2)

II. T(n) = 8T(n/3) + bn.
Time complexity :
T(n) = xT(n/3) + bn
for x = 5,
T(n) = 5T(n/3) + bn
Time complexity : O(O(n^1.464)

Generalised equation of time complexities of k-ways split

1. DAC : T(n) = kT(n/k) + bn
2. Karatsuba : T(n) = (k^2 - 1)T(n/k) + bn
3. Toom-cook : T(n) = (2k-)T(n/k) + bn

Toom-4 exists but is less practical due to overhead.

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

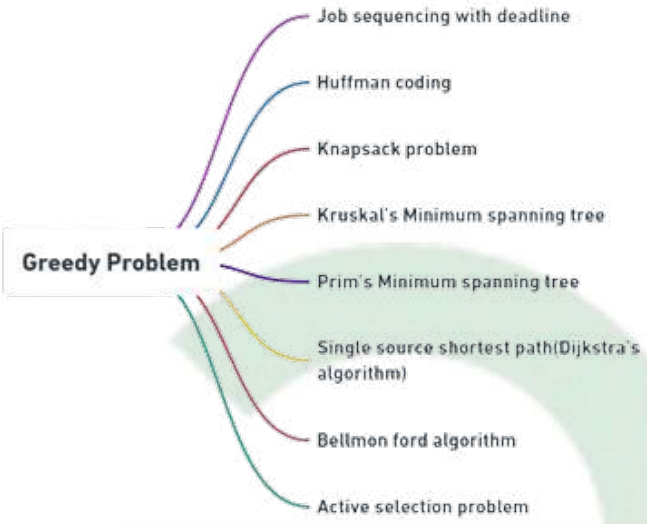
ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

Module 3 : Greedy Algorithm

The greedy technique algorithm is a method that makes the locally optimal choice at each step with the hope of finding a global optimum, without reconsidering previous choices.



Application of Greedy algorithm

1. Job sequencing with deadline

Schedule jobs to **maximize profit** before their deadlines (one job per time slot).
Sort jobs by profit (descending).
Find maximum deadline in the given array of n-deadlines and take the array of maximum deadlines size
Schedule each job in the **latest available slot** before its deadline.

Time complexity

Best case : O(nlogn)
Worst case : $O(n^2)$

2. Optimal merge pattern (Huffman coding is one of its application)

Merge n sorted files with minimum total cost (record movements).
Always merge the **two smallest** files. Repeat until one file remains.

Huffman coding is a direct application of the **optimal merge pattern**.

Step to solve a problem : Create a min-heap (priority queue) of all characters based on their frequencies.

Repeat until the heap contains only one node:

(i) Extract the two nodes with the **smallest frequencies**.

(ii) Create a new internal node with:
Frequency = sum of the two nodes.
Left child = node with smaller frequency.
ight child = node with larger frequency.

(iii) Insert this new node back into the heap.
The remaining node is the **root** of the Huffman tree.
Time complexity : **O(n log n)**
Space complexity : **O(n)**

3. Fractional Knapsack

Maximize profit with given weight capacity.
Fractional items allowed.
Sort by value/weight ratio. Pick greedy until capacity is full.
for(i=1;i<=n;i++)
a[i] = Profit(i)/weight(i)
Take one by one object from a and keep in knapsack until knapsack becomes full arrange array a in ascending order
Time complexity : **O(n log n)**

4. Activity selection problem (You are given **n activities**, each with a **start time** and **finish time**.

The goal is to select the **maximum number of activities** that can be performed by a **single person**, under the constraint that the person can work on **only one activity at a time** (i.e., no overlapping activities).
Sort the activities by their finishing time (in ascending order).
Select the first activity in the sorted list and include it in the final solution.
Iterate through the remaining activities in the sorted list:

- For each activity, **check if its start time** is **greater than or equal to** the **finish time of the last selected activity**.
- If the condition holds, **select the activity** and update the last selected finish time.

Time complexity

If activities are not sorted by finish time:
- Sorting takes O(nlog⁡n)
- Selecting activities takes O(n)O(n)O(n)
- **Total time = O(nlog⁡n)**

If activities are sorted by finish time:
- Only the selection loop runs → O(n)
- **Total time = O(n)**

5. Minimum cost spanning tree

I. Kruskal's Minimum spanning tree algorithm

It builds the Minimum Spanning Tree by always choosing the next **lightest edge** that doesn't form a cycle.

Sort all edges of the graph in **non-decreasing order** of their weights.

Initialize an empty set for the **MST**.

For each edge in the sorted list:

- If the edge **does not form a cycle** with the MST formed so far, **include it** in the MST. Otherwise discard the edge.

Repeat until the MST includes V−1V - 1V−1 edges (where VVV is the number of vertices).

Time complexity : O(E log E) or O(E log V)

Note : Works well with **sparse graphs** (fewer edges). May produce a **forest** if the graph is not connected.

II. Prim's minimum spanning tree algorithm

It builds the MST by growing it one **vertex** at a time, always choosing the **minimum-weight edge** that connects a vertex inside the MST to one outside.

Start with a random vertex, Initialize a **MST set** (vertices included in MST), and a **priority queue** (or min-heap) of edge weights.

While the MST set does not include all vertices:

- Select the **minimum-weight edge** that connects a vertex in the MST to a vertex outside.
- Add the selected edge and vertex to the MST.
- More efficient for dense graph

Time complexity :

Adjacency matrix + linear search = O(V^2)
Adjacency list + binary heap = O(E log V)
Adjacency list + Fibonacci heap = O(E + log V)

6. Single source shortest path algorithm

I. Dijkstra's algorithm

Using min heap & adjacency list = O(E + V)logV
Using adjacency Matrix & min heap = O(V^2 * E * logV)
Using adjacency list & Unsorted array = O(V^2)
Using adjacency list & sorted Doubly linked list = O(EV)

II. Bellman Ford algorithm

It finds the shortest path from source to every vertex, if the graph doesn't contain a negative weight edge cycle.

If a graph contains a negative weight cycle, it does not compute the shortest path form source to all other vertices but it will report saying "negative weight cycle exists".

It finds shortest path from source to every vertex,

Input : A weighted, directed graph G = (V+E), with edge weights w(u, v), and a source vertex s.

Output : Shortest path distance from source s to all other vertices, or detection of a negative-weight.

Time complexity : O(EV)

Module 4 : Dynamic Programming

Use case of Tabulation and memoization method

- If the original problem **requires all subproblems** to be solved, then **tabulation** is usually more efficient than memoization.
- **Tabulation** avoids the **overhead of recursion** and can use a **preallocated array**, leading to better performance in both time and space in some cases.
- If **only some subproblems** are needed to solve the original problem, then **memoization** is preferable, because it solves **only the required subproblems** (solved *lazily*, i.e., on-demand).

1. Longest common subsequence (LCS)

Given two strings, find the length of their longest subsequence that appears in both. Subsequence must be linear only not necessarily contiguous.

Input : x= <ABCD>, y = <BDC>
Output : 2 <BC>

Let, i & j denote indices of x & y. L(i, j) denote the LCS of string x & y , n & m are length respectively.
L(i, j) = 1 + L(i-1, j-1) ; if x[i] = y[j]
L(i, j) = max(L(i-1, j) , L(i , j-1) ; if x[i] != y[j]
L(-i, j) = 0
L(i, -j) = 0
Time complexity : O(n * m)
Space complexity : O(n * m)

2. 0/1 Knapsack problem

Input : N items, each item has weight W[i], profit[i] and a knapsack with a minimum capacity M
Objective : Total weight <= M, and total profit maximized. Each item can be either 1 (include) or 0 (exclude)
Recurrence relation
KS(M, N) = 0 ; if M=0 or N=0
KS(M, N) = 0 ; if W[N]>M
KS(M, N) = max(K(M - W[N], N-1) + P[N], K(M, N-1)) ; otherwise

Time complexity : O(M * N) (We compute and store results in a 2D of table of size M* N)

3. Travelling salesman problem

Given a set of cities and distances between every pair of cities, the goal is to find the shortest possible tour that visits each city exactly once and returns to the starting city.
This is equivalent to finding the minimum cost Hamiltonian cycle.
A cost/distance function C(i, j) representing the cost to travel from city i to city j.
TSP(A, R) be the minimum cost of visiting all cities in the set R, starting from city A.
TSP = C(A, S) ; if R = 0
TSP = min(C(A, K) + TSP(K, R-{K}) ; otherwise.
Where A is current city
R : set of unvisited cities
S : Starting city
C(A, K) : cost from city A to city K

Time complexity : (without dynamic programming) O (n^n) with dynamic programming O(2^n * n^2)
Space complexity : O(2^n * n^2)

4. Matrix chain multiplication

Given a sequence of matrices, find the most efficient order of multiplication of these matrices together in order to minimize the number of multiplications.
Let MCM(i, j) denote the minimum number of scaler multiplication required to multiply matrices from Ai to Aj.
MCM(i, j) = 0 ; i = j
MCM(i, j) = min(MCM(i, k) + MCM(k + 1, j) + Pi-1 * Pk * Pj) ; if i < j
The cost of multiplying the resulting matrices is $Pi-1 \cdot Pk \cdot Pj$
The total number of ways to parenthesize the matrix chain of n matrices :
$T(n) = \sum_{n-1}^{i-1} \quad T(i) \cdot T(n-i)$
Number of parenthesizing for a given chain represented by catalan number : $[1 / (n+1) \ (2nCn)]$

Time complexity:

- Without DP : O(n!)
- With DP : O(n^3)

Space complexity:

- Without DP : O(n)
- With DP : O(n^2)

5. Sum of subset problem

Given a set of numbers W[1...N] and a value M, determine if there exists a subset whose sum is exactly M.

Recursive Relation:

SoS(M, N, S) =
return(S) ; if M = 0
return(-1) ; if N = 0
SoS(M, N - 1, S) ; if W[N] > M
min(
SoS(M - W[N], N - 1, S ∪ {W[N]}),
SoS(M, N - 1, S)
) ; otherwise

Time complexity by brute force: O(2^N)
Time complexity with DP: O(M × N) where M is the target sum, N is the number of elements
Space complexity : without optimization O(M × N)
With space optimization : O(M)

6. Floyd-warshall's : All pair shortest path

Used to find the shortest distances between every pair of vertices in a weighted graph. Works with positive and negative edge weights (but no negative weight cycles allowed).
Recurrence relation
A^0(i, j) = C(i, j)
A^k(i, j) = min(A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j))
where : C(i,j): initial weight of the edge from iii to jjj

A^k(i,j)A^k(i, j)Ak(i,j): shortest path from iii to jjj using vertices {1,2,...,k}\{1, 2, \dots, k\}{1,2,...,k} as intermediate nodes
Time complexity O(n^3)
Space complexity O(n^2)

7. Optimal binary search tree

Given a sorted array of keys[0..n-1] and their frequencies:
- p[i]: frequency of successful searches for keys[i]
- q[i]: frequency of unsuccessful searches between keys
Goal: Construct a binary search tree that minimizes the total expected cost of searches.

Recurrence relation

cost(i, j) =
if j < i → return 0
else → min over k ∈ [i..j] of:
cost(i, k-1) + cost(k+1, j) + w(i, j)
Where: w(i, j) = sum of p[i..j] + sum of q[i-1..j]
Time complexity : O(n^3)
Space complexity : O(n^2)

8. Multistage graph

A **multistage graph** is a **directed acyclic graph (DAG)** in which the set of vertices is partitioned into **stages** (e.g., S1,S2,...,Sk) such that:
Every edge connects a vertex from stage iii to stage i+1
The goal is to find the **shortest path** from the **source vertex** in stage S1to the **destination vertex** in stage Sk
MSG(si, vj) =
0 if si = F and vj is the destination
min over all K in si+1 where (vj, K) ∈ E:
cost(vj, K) + MSG(si+1, K)

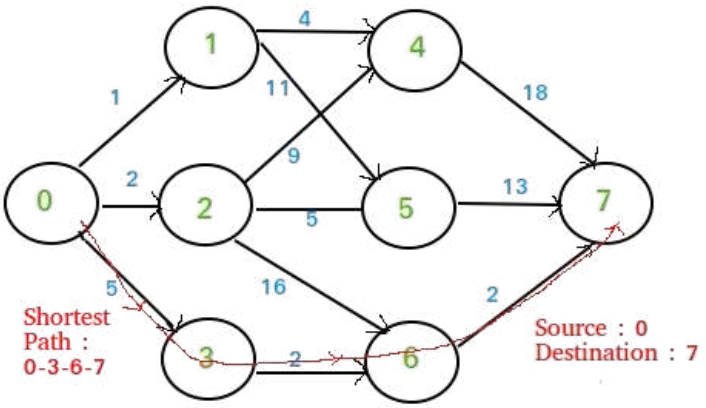
Time Complexity:

- Without dynamic programming: O(2^n)
- With dynamic programming: O(V + E)

Space Complexity:

- Without dynamic programming: O(V^2)

- With dynamic programming: O(V^2)



Time Complexity:

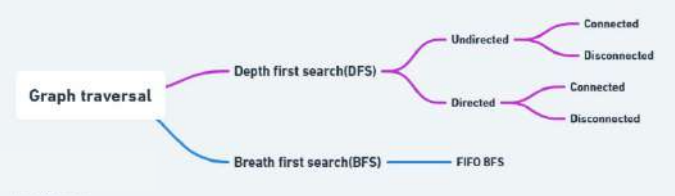
- Without DP: O(2^n) (due to exponential combinations)
- With DP: O(V + E)
  (because each vertex and edge is processed only once)

Space Complexity:

- Without DP: O(V^2)
- With DP: O(V^2)

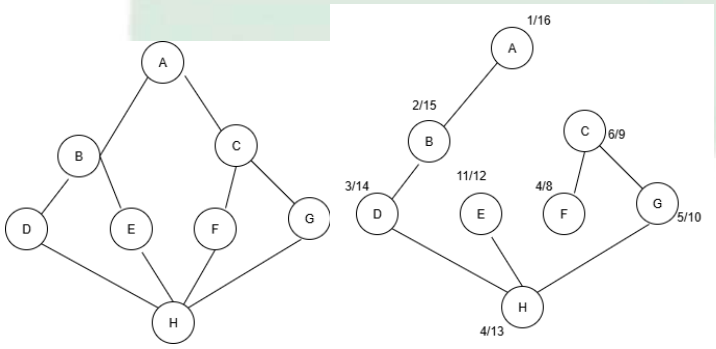
Module 5 : Graph traversal Techniques

Visiting all nodes of the tree/Graph in a specified order and processing the information only once.



DFS in Undirected graph:

a. **Connected graph**
   Structure of node
   E-node: Exploring node
   Live node: Node which is not fully explored
   Dead node: Node which is fully explored
   Time associated with the node, during traversal
   Discovery time: The time at which the node is visited for the first time.
   Finishing time: The time at which nodes become dead.



b. **Disconnected/Disjoint graph : Depth forest tree**

DFS in Directed graph: DFS when carried out on a directed graph leads to following types of edge.

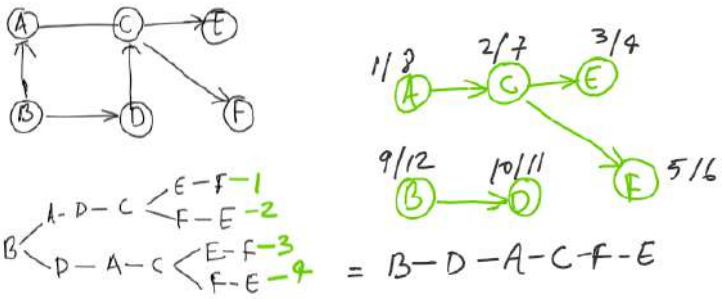
1. Tree edge : it is part of DFS spanning tree or forest
2. Forward edge: Leads from a node to its non child descendant in the spanning tree
3. Back edge: Leads from a node to its ancestors
4. Cross edge: Leads to a node which is neither ascending nor descending.

DFS in Directed graph acyclic graph

Topological Sort:

Linear order of the vertices representing the activities maintaining precedence.

Example below.



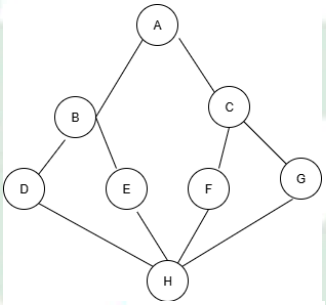
Topological sort(){

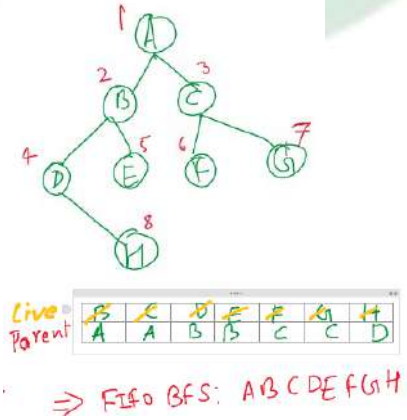
1. **DFS(v).**
2. **Arrange all the nodes of traversal in decreasing order of finishing time.**

}

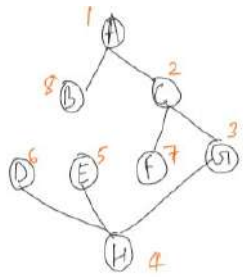
BFS : Level by level order traversal

1. **FIFO BFS: (BFS spanning tree) A B C D E F G H**





2. **LIFO BFS: A C G H E D F B**



Application of DFS & BSF

Time complexity of DFS and BFS depends upon representation of Graph:

(i) **Adjacency matrix**: $O(V^2)$

(ii) **Adjacency list**: $O(V + E)$

Both **DFS** and **BFS** can be used to detect the presence of **cycle** in the graph.

Both **DFS** and **BFS** can be used to know whether the given graph is **connected** or not.

Both **DFS** and **BFS** can be used to know whether the two vertices **u** and **v** are **connected** or not.

DFS is used to determine **connected**, **strongly connected**, **biconnected components**, and **articulation points**.

Connected Component (Undirected graph) : It is a maximal set of vertices such that there is a path between any pair of vertices in that set.

Strongly connected component (Directed graph): A Strongly Connected Component (SCC) of a directed graph is a maximal set of vertices such that for every pair of vertices u and v in the set, there is a path from u to v and a path from v to u.

Properties of Strongly Connected Components

1. Every directed graph is a D.A.G. of strongly connected components.

2. Let C and C′ be distinct strongly connected components in directed graph G = (V, E). Let u, v ∈ C and u′, v′ ∈ C′. Suppose that there is a path u → u′ in G, then there cannot be a path v′ → v in G.

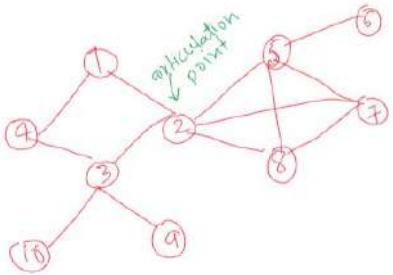
3. If C and C′ are strongly connected components of G, and there is an edge from a node in C to a node in C′, then the highest post number in C is bigger than the highest post number in C′.

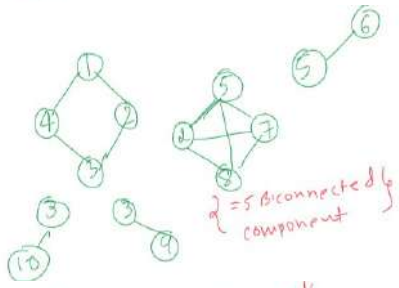
Articulation point (cut vertex)

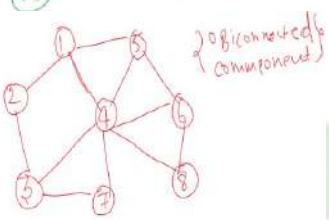
Articulation Point: A vertex whose removal increases the number of connected components in a graph.

Bi-connected Graph: A graph with **no articulation points**.

Bi-connected Component: A **maximal subgraph** that is bi-connected.







Searching and Sorting

Classification	Explanation
Internal vs External sorting	**Internal:** All data fits into main memory (RAM). **External**: Used when data is too large to fit into memory and uses external storage
Comparison vs Non-comparison Based	**Comparison:** Sorting is done using comparisons between elements **Non-comparison:** Uses digit-based or counting approaches(radix sort, counting sort)
Recursive vs Iterative	**Recursive:** The function calls itself to divide and conquer (e.g., Merge Sort, Quick Sort).
In-place vs Not-in-place	Space required is generally O(1) or O(log n) at most (for recursion stack) Merge Sort → O(n) space
Stable vs Unstable	Relative order of same elements is maintained (Stable)

1. Comparison based sorting algorithm

Algorithm	Time complexity			Stable sorting	In place sorting
	Best	Average	Worst		
Quick sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	No	Yes
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	Yes	No
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	Yes	Yes
Selection sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	No	Yes
Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	Yes	Yes
Heap sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	No	Yes

Selection sort takes the least number of swaps overall i.e. **(n – 1) swaps** — no matter how unsorted the input is.

2. Non Comparison based sorting:

Algorithm	Time complexity			Stable sorting	In place sorting
	Best	Average	Worst		
Radix sort	Ω(d * (n + k))	Θ(d * (n + k))	O(d * (n + k))	Yes	No
Counting sort	Ω(n + k)	Θ(n + k)	O(n + k)	Yes	No
Bucket sort	Ω(n + k)	Θ(n + k)	$O(n^2)$	Yes(if stable sort used inside buckets)	No

GeeksforGeeks

GATE CSE BATCH

KEY HIGHLIGHTS:

- 300+ HOURS OF RECORDED CONTENT
- 900+ HOURS OF LIVE CONTENT
- SKILL ASSESSMENT CONTESTS
- 6 MONTHS OF 24/7 ONE-ON-ONE AI DOUBT ASSISTANCE
- SUPPORTING NOTES/DOCUMENTATION AND DPPS FOR EVERY LECTURE

COURSE COVERAGE:

- ENGINEERING MATHEMATICS
- GENERAL APTITUDE
- DISCRETE MATHEMATICS
- DIGITAL LOGIC
- COMPUTER ORGANIZATION AND ARCHITECTURE
- C PROGRAMMING
- DATA STRUCTURES
- ALGORITHMS
- THEORY OF COMPUTATION
- COMPILER DESIGN
- OPERATING SYSTEM
- DATABASE MANAGEMENT SYSTEM
- COMPUTER NETWORKS

LEARNING BENEFIT:

- GUIDANCE FROM EXPERT MENTORS
- COMPREHENSIVE GATE SYLLABUS COVERAGE
- EXCLUSIVE ACCESS TO E-STUDY MATERIALS
- ONLINE DOUBT-SOLVING WITH AI
- QUIZZES, DPPS AND PREVIOUS YEAR QUESTIONS SOLUTIONS

ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW