

Shri Vile Parle Kelavani Mandal’s

**Institute of Technology, Dhule**

**Deapartment of Information Technology**

Design and Analysis of Algorithms Lab

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr No:** | **Students Name** | **Roll No** | **PRN No** | **Class** |
| 1 | Anam Arif Shaikh | 63 | 2254491246053 | Second Year |

**Question 1 :** Explain Polynomial time reduction with example

**Answer :** **Reductions**

Reduction algorithm reduces a problem to another problem. The input of a problem is called instance. Problem A is reduced to another problem B, if any instance of A “can be rephrased” as instance of B, the solution of which provides a solution to the instance of A .  There are two types of reduction. One is called Turing reduction and another is called Karp reduction.

A **polynomial-time reduction** is a method for solving one problem using another. A polynomial-time reduction proves that the first problem is no more difficult than the second one, because whenever an efficient algorithm exists for the second problem, one exists for the first problem as well.

For input size n, if worst-case time complexity of an algorithm is O(n ), where k is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time

However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the NP-Complete problems, whose status is unknown.

We want prove some problems are computationally difficult. As a first step, we settle for relative judgements: Problem X is at least as hard as problem Y .To prove such a statement, we reduce problem Y to problem X

* If problem Y can be reduced to problem X, we denote this by Y ≤P X.
* This means “Y is polynomal-time reducible to X.”
* It also means that X is at least as hard as Y because if you can solve X, you can solve Y .

Note: We reduce to the problem we want to show is the harder problem.

If Y ≤P X and Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

**Question 2**: Explain Class P, Class NP and Class NPC problems in detail .

**Answer:**

1. **Class P**

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems (problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.

**Features:**

* The solution to **P problem**s is easy to find.
* **P** is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many problems:

1. **Calculating the greatest common divisor.**
2. **Finding a maximum matching.**
3. **Merge Sort**
4. **Class NP :**

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

**Features:**

* The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
* Problems of NP can be verified by a Turing machine in polynomial time.

1. **Boolean Satisfiability Problem (SAT).**
2. **Hamiltonian Path Problem.**
3. **Graph coloring.**

NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time. This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time. Example Integer factorisation is in NP. This is the problem that given integers n and m, is there an integer f with 1 < f < m, such that f divides n (f is a small factor of n)? This is a decision problem because the answers are yes or no. If someone hands us an instance of the problem (so they hand us integers n and m) and an integer f with 1 < f < m, and claim that f is a factor of n (the certificate), we can check the answer in polynomial time by performing the division n / f.

1. **NPC Class :**

NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time. Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Yis reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

**Question 3** : Differentiate between DFS and BFS along with example

**Answer** :

|  |  |
| --- | --- |
| DFS | BFS |
| DFS stands for Depth First Search . | BFS stands for Breadth First Search. |
| DFS(Depth First Search) uses Stack data structure. | BFS(Breadth First Search) uses Queue data structure for finding the shortest path. |
| It works on the concept of LIFO (Last In First Out). | It works on the concept of FIFO (First In First Out). |
| DFS builds the tree sub-tree by sub-tree. | BFS builds the tree level by level. |
| It is comparatively faster than the BFS method. | It is comparatively slower than the DFS method. |
| You need to follow a backtrack in DFS. | You don’t need to backtrack in BFS. |

DFS example :

2

1

0

3

4

5

8

7

6

|  |  |  |  |
| --- | --- | --- | --- |
| Pop | Traversing vertex | Push | Stack |
| 0 | 0 | 4,3,1 | 4,3,1 |
| 1 | 1 | 2,4 | 4,3,4,2 |
| 2 | 2 | 5 | 4,3,4,5 |
| 5 | 5 | - | 4,3,4 |
| 4 | 4 | 7 | 4,3,7 |
| 7 | 7 | 8 | 4,3,8 |
| 8 | 8 | - | 4,3 |
| 3 | 3 | 6 | 4,3,6 |
| 6 | 6 | - | 4,3 |
| 3 | - | - | 4 |
| 4 | - | - | - |

**Final answer by DFS : 0,1,2,5,4,7,8,3,6**

By solving BFS through same example we can get the output through following steps :

* First we visit the starting vertex and then we will visit all the adjacent vertices of this starting vertex.
* Then we pick these adjacent vertices one by one and visit their adjacent vertices and this process goes on.

**Final answer by BFS : 0,1,3,4,2,6,5,7,8.**

**Question 4** : Explain different asymptotic notations.

**Answer** :

**Asymptotic notations:**

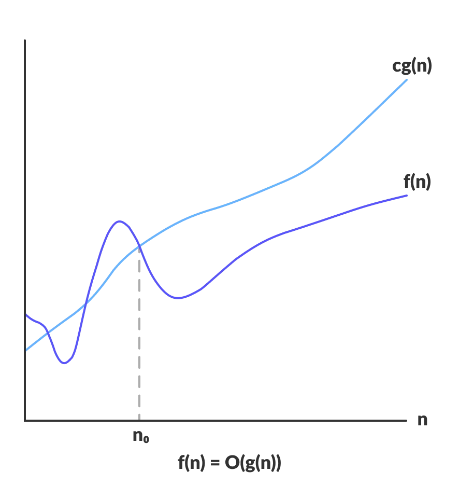
The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations. Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value. There are several asymptotic notations, including Big O, Omega, and Theta.

**Big O Notation (O):**

Big O notation is used to describe the upper bound or worst-case scenario of an algorithm's running time. It essentially answers the question: "How bad can it get?"

Characteristics:

1. Upper Bound: Big O provides an upper limit on the growth rate of a function.
2. Worst-Case Scenario: It focuses on the maximum time an algorithm takes to run.
3. Simplified Analysis: It simplifies the analysis by ignoring constant factors and lower-order terms.
4. Bounding Function: It helps in understanding how the algorithm's performance scales with the size of the input.
5. Not Necessarily Tight: The upper bound provided by Big O may not always be tight or exact but gives an idea of the algorithm's behavior.



*O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }*

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Example:

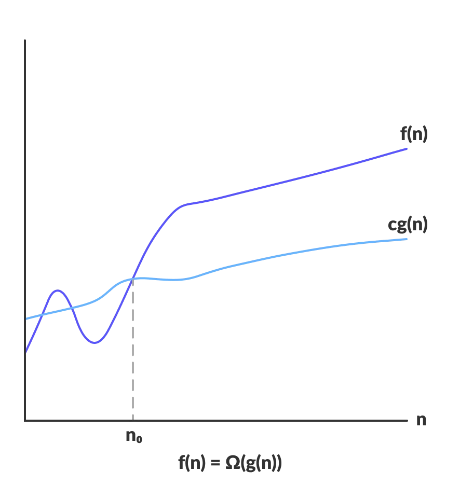
If an algorithm has a time complexity of O(n^2), it means that for sufficiently large input sizes, the algorithm will run no worse than quadratically. It might run better, but it won't run worse.

**Omega Notation (Ω):**

Omega notation is used to describe the lower bound or best-case scenario of an algorithm's running time. It answers the question: "How good can it get?"

Characteristics:

1. Lower Bound: Omega provides a lower limit on the growth rate of a function.
2. Best-Case Scenario: It focuses on the minimum time an algorithm takes to run.
3. Essential Complexity: Omega notation helps in understanding the inherent complexity of an algorithm.
4. Determines Algorithm's Efficiency: It helps in determining the algorithm's efficiency when the input size is large.
5. Not Necessarily Tight: Just like Big O, Omega may not always provide a tight lower bound.



*Ω(g(n)) = { f(n): there exist positive constants c and n0 such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }*

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

Example:

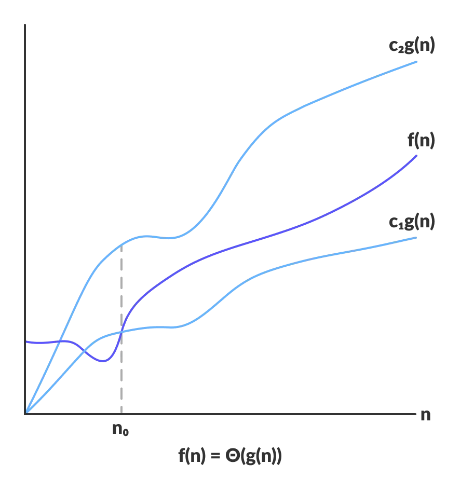
If an algorithm has a time complexity of Ω(n), it means that for sufficiently large sizes, the algorithm will run no better than linearly. It might run worse, but it won't run better.

**Theta Notation (θ):**

Theta notation provides both upper and lower bounds or tight bounds on the growth rate of a function. It essentially answers the question: "How does the algorithm behave exactly?"

Characteristics:

1. Tight Bound: Theta notation gives a tight range within which the algorithm's running time falls.
2. Average Case or Exact Growth Rate: It represents the exact growth rate of a function.
3. Balanced View: It offers a balanced view of the algorithm's performance across different input sizes.
4. Useful for Analysis: Theta notation is particularly useful for precise analysis of algorithms.
5. Exact Performance: It indicates that the algorithm's performance matches the growth rate described by θ.



*Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }*

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

Example:

If an algorithm has a time complexity of θ(n^2), it means that for sufficiently large input sizes, the algorithm's running time grows exactly quadratically. It neither runs faster nor slower than quadratic for large inputs.