Algorithm

There are some following definitions of algorithms:-

- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudo code.

Time-Space Trade-Off in Algorithms

The time-space trade-off refers to a situation where saving time requires using more memory (space), and saving memory requires more processing time.

Types of Trade-Offs

Time vs Space: Faster algorithms may use more memory.
Space vs Time: Algorithms that use less memory may take longer to execute.

Examples of Time-Space Trade-Off:
Time Efficient: Searching for an element using a hash table is fast (O(1)), but requires additional memory to store the table.
Space Efficient: Linear search uses less memory but takes more time (O(n)).

Dynamic Programming:

Time Efficient: Storing solutions to subproblems reduces computation time (e.g., Fibonacci with memoization runs in O(n) time).
Space Inefficient: It requires extra space to store results of subproblems.

Merge Sort vs Insertion Sort:

Merge Sort: Faster (O(n log n)) but requires extra space for merging.
Insertion Sort: Slower ($O(n^2)$) but sorts in-place with no extra memory required.

Trade-Off in Compression Algorithms:
Compressed Data: Saves space but requires extra time to decompress when accessed.
Uncompressed Data: Faster access but uses more space.

Trade-Off in Recursive Algorithms:
Time Efficient: Some recursive algorithms can solve problems quickly, but they use more memory in the form of a call stack.
Space Efficient: Iterative solutions may use less space but might involve longer loops or calculations.

Calculating Running Time of an Algorithm

Calculation Depends on:

1. **Basic Operation:** Constant-time operations like arithmetic, Boolean, and comparison operations.
2. **Input Size:** Number of inputs processed by the algorithm (e.g., for sorting, it is the number of records).
3. **Growth Rate:** Describes how the algorithm's running time increases with input size.

Complexity of an Algorithm

Space Complexity: Memory required by the algorithm.
Fixed Part: Space for input, output, and program size.
Variable Part: Space for temporary variables, dynamic allocation, recursion, etc.

Time Complexity: Time required by the algorithm.
Constant Time Part: Operations executed once (input/output).
Variable Time Part: Operations executed multiple times (loops, recursion).
Formula: T(P) = C + T(I)
Example (Linear Search): T(n) = 5 + n

Best, Average, and Worst-Case Complexity

1. **Worst-Case:** Maximum time for any input (upper bound).
2. **Average-Case:** Expected time for random input.
3. **Best-Case:** Minimum time for the optimal input.

Asymptotic Analysis

Asymptotic analysis measures the performance of algorithms by analyzing their behavior as input size increases.
Example: Linear Search (O(n)) vs. Binary Search (O(log n)) — for large input sizes, Binary Search is faster, regardless of machine speed.

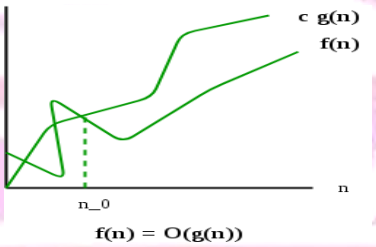
Cases of Algorithm Analysis

1. **Worst Case:** Guarantees upper bound;
2. **Average Case:** Requires knowledge of input distribution;
3. **Best Case:** Guarantees lower bound;

Asymptotic Notations

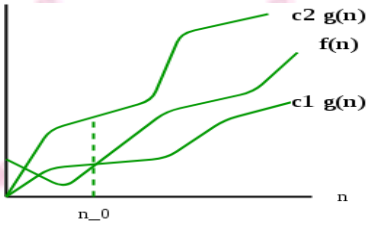
Used to express the running time complexity concerning input size:

1. **Big-O Notation (O):**
   Defines the upper bound (worst-case scenario).
   Example: Insertion Sort's time complexity is $O(n^2)$.



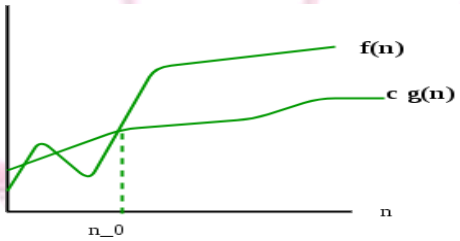
f(n) = O(g(n))

2. **Theta Notation (Θ):**
   Defines both upper and lower bounds (average-case).
   Example: $3n^3 + 6n^2 = \Theta(n^3)$.



f(n) = theta(g(n))

3. **Omega Notation (Ω):**
   Defines the lower bound (best-case).
   Example: Insertion Sort can be written as Ω(n).



f(n) = Omega(g(n))

Little-o Notation (o):
Provides a strict upper bound.
Example: $7n + 8 \in o(n^2)$.
Little-Omega Notation (ω):
Provides a strict lower bound.
Example: $4n + 6 \in \omega(1)$.

Common Time Complexities

O(1): Constant time, no loops or recursion.
Example: swap() function.

O(n): Loop where the variable is incremented/decremented linearly.
Example:

```
for (int i = 1; i <= n; i++) { // O(1) operation }
```

$O(n^2)$: Nested loops.
Example:
```
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j++) { // O(1) operation }
}
```

O(log n): Loop where the variable is divided/multiplied by a constant factor.
Example:
```
for (int i = 1; i <= n; i *= 2) { // O(1) operation }
```

O(log log n): Exponential growth/decrease in loop variable.
```
for (int i = 2; i <= n; i = pow(i, c)) { // O(1) operation }
```

Summation of Loops: For consecutive loops, sum their complexities.
```
for (int i = 1; i <= m; i++) { // O(m) operation }
for (int i = 1; i <= n; i++) { // O(n) operation }
```
Total complexity = O(m + n)

Design Techniques of algorithm

An Algorithm is a procedure to solve a particular problem in a finite number of steps for a finite-sized input. It takes a set of input(s) and produces the desired output.

Qualities of a Good Algorithm:-

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

The algorithms can be classified in various ways. There are some following classifications:

a) Implementation Method
b) Design Method
c) Design Techniques
d) Other Classifications

Design techniques of algorithm

- It is important to learn algorithm design techniques in data structures in order to be able to build scalable systems.
- Selecting a proper design technique for algorithms is a complex but important task. There are many standard approaches to design an algorithm based on different strategies.

Following are some of the main algorithm design techniques:

1. **Brute-force or exhaustive search**
2. **Divide and Conquer**
3. **Greedy Algorithms**
4. **Dynamic Programming**
5. **Branch and Bound Algorithm**
6. **Backtracking**

1. Brute force technique

- It is the general and the simplest way to design an algorithm. It simply involves a direct logic-based structure to design an
- algorithm.
- This is also called an 'Exhaustive search algorithm'.
- A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found.
- The time complexity of a brute force algorithm is often proportional to the input size.
- Brute force algorithms are simple and consistent, but very slow.

Advantages of a brute-force algorithm

- This algorithm finds all the possible solutions, and it also guarantees that it finds the correct solution to a problem.
- This type of algorithm is applicable to a wide range of domains.
- It is mainly used for solving simpler and small problems.
- It can be considered a comparison benchmark to solve a simple problem and does not require any particular domain knowledge.

Disadvantages of a brute-force algorithm

- It is an inefficient algorithm as it requires solving each and every state. For real-time problems, algorithm analysis often goes above the O(N!) order of growth.
- It is a very slow algorithm to find the correct solution as it solves each state without considering whether the solution is feasible or not.
- The brute force algorithm is neither constructive nor creative as compared to other algorithms.

2. Divide and conquer technique

- This approach is based on dividing the problem into smaller sub-problems and solving those sub-problems. Once the sub-problems are solved, we combine the solutions of those sub-problems to get the final solution.
- In other words, Divide and conquer is a recursive problem-solving approach in data structure and algorithms that divide the problem into smaller sub-problems, recursively solve each sub-problem, and combine the solutions to the sub-problems to get the solution of the original problem.

There are some standard algorithms given below that follow Divide and Conquer algorithm:

1. **Binary Search**
2. **Quick sort**
3. **Merge Sort**
4. **Closest Pair of Points**
5. **Strassen's Algorithm**
6. **DAC solution of the convex hull**
7. **DAC solution of merging k sorted list**
8. **DAC solution of finding min and max**

Recurrence Relation for DAC algorithm

$$T(n)=2T(n/2)+\Theta(n)$$

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle.
- It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.

- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

3. Greedy Algorithm Techniques

- This method is used to solve optimization problems in which set of input values are given, that are required either to be increased or decreased according to the objective.
- An optimization problem is a problem that demands either maximum or minimum results.
- Greedy Algorithm always chooses the option, which appears to be the best at the moment. It doesn't worry whether the current best result will bring the overall optimal result. That is why it is known as greedy algorithm.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm makes optimal choices at every iteration to get the best possible solution.
- It may not always give the optimized solution.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Advantages of the Greedy Approach

- The greedy approach is easy to implement.
- Typically have less time complexity.
- Greedy algorithms can be used for optimization purposes or finding close to optimization in case of Hard problems.

Disadvantages of the Greedy Approach

- The local optimal solution may not always be globally optimal.
- The problematic part for a greedy algorithm is analyzing its accuracy. Even with the proper solution, it is difficult to demonstrate why it is accurate.
- Optimization problems (Dijkstra's Algorithm) with negative graph edges cannot be solved using a greedy algorithm.

Applications of Greedy Algorithm

- Kruskal's Minimum Spanning Tree
- Prim's Minimum Spanning Tree
- Dijkastra's Shortest Path Algorithm
- Huffman Coding
- Job Sequencing Problem
- Fractional knapsack problem
- Optimal Reliability Allocation
- activity selection problem
- Graph - Map Coloring
- Graph - Vertex Cover

4. Dynamic programming

- Like greedy method dynamic programming also provides optimal solution. The drawback of greedy method is, we will make one decision at a time. This can be overcome in dynamic programming.
- Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.
- Dynamic programming is applicable when the sub-problems are not independent, that is when sub-problems share sub-sub-problems.
- A dynamic programming algorithm solves every sub-sub-problem just once and then saves its answer in a table, there by avoiding the work of re-computing the answer every time the sub-problem is encountered.
- Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again.
- The sub-problems are optimized to optimize the overall solution is known as optimal substructure property.
- The main use of dynamic programming is to solve optimization problems.

- Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem.
- The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

Properties of dynamic programming

There are two main properties of a problem that suggests that the given problem can be solved using Dynamic programming:-

a) Overlapping Subproblems
b) Optimal Substructure

Applications Of Dynamic Programming

1. Bellman ford shortest path algorithm
2. Floyd Warshall all pair shortest path algorithm
3. 0/1 Knapsack Problem
4. Multi stage Graph (Resource-allocation problems)
5. Matrix Chain Multiplication
6. Reliability Design

5. Backtracking

- A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.
- The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- Backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.
- Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.
- In any backtracking problems, the algorithm tries to find a path to the feasible solution which has some intermediary checkpoints.
- The backtracking algorithms are generally exponential in nature with regards to both time and space.
- A backtracking algorithm uses the depth-first search method. When the algorithm begins to explore the solutions, the abounding function is applied so that the algorithm can determine whether the proposed solution satisfies the constraints. If it does, it will keep looking. If it does not, the branch is removed, and the algorithm returns to the previous level.

Types of Backtracking Algorithm

Backtracking algorithms are classified into two types:

- Algorithm for recursive backtracking
- Non-recursive backtracking algorithm

Applications of Backtracking

1. N-queen problem
2. Sum of subset problem
3. Graph coloring
4. Hamiliton cycle
5. Generate k-ary Strings
6. Rat in a maze problem
7. The Knight's Tour Problem

6. Branch and bound

- It is an optimization algorithm used to solve combinatory, discrete and general mathematical problems.
- It involves dividing the problem, obtaining sub-solutions for them and then finding the most optimal solution.
- It is similar to the backtracking since it also uses the state space tree.
- It is used for solving the optimization problems and minimization problems.
- If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

Types of Branch and Bound

There are multiple types of the Branch and Bound method, based on the order in which the state space tree is to be searched. We will be using the variable solution method to denote the solutions in these methods.

1. **FIFO Branch and Bound**
2. **LIFO Branch and Bound**
3. **Least Cost-Branch and Bound**

Applications of Branch and Bound

1. **Job Sequencing Problem:-** It can be solved by Dynamic programming and also by branch and bound.
2. **0/1 Knapsack problem:-** This problem can also be solved using the backtracking, brute force and the dynamic programming approach.
3. **Traveling Salesman Problem:-** Here, we are given a set of N cities, and the cost of traveling between all pairs of cities. The problem is to find a path such that one starts from a given node, visits all cities exactly once, and returns back to the starting city. This problem can also be solved using the backtracking, and the dynamic programming approach.

P/ NP/ NP Hard/NP Complete

P (Polynomial Time): Problems that can be solved by a deterministic algorithm in polynomial time.

- Time complexity to solve polynomial time problems is n^k, (where k is a constant).
- It is solvable efficiently (feasible) by computers.
- **Examples:** searching and Sorting algorithms like bubble, Merge Sort (O(n log n)), finding the shortest path in a graph (Dijkstra's Algorithm) etc.

NP (Nondeterministic Polynomial Time): Problems that can be verified in polynomial time by a deterministic algorithm, but not necessarily solved in polynomial time.

- Time complexity to solve Nondeterministic polynomial time problems is either exponential or non deterministic.
- **Examples:** Traveling Salesman problem, 0/1knapsack problem, n queen problem, Subset Sum, Sudoku, Hamiltonian Path Problem.
- Every problem in P is also in NP because if you can solve a problem in polynomial time, you can verify the solution in polynomial time.

Difference between Polynomial and Non-deterministic Polynomial

Aspect	P (Polynomial Time)	NP (Nondeterministic Polynomial Time)
Definition	Problems that can be solved in polynomial time.	Problems whose solutions can be verified in polynomial time, but may not be solvable in polynomial time.
Time Complexity	Solvable in polynomial time by a deterministic algorithm (O(n^k), where k is a constant).	Solutions may take non-polynomial time to find but can be verified in polynomial time. (Ex: $O(2^n)$)
Verifiability	Both solving and verifying a solution can be done in polynomial time.	Solution can be verified in polynomial time if one is provided.

Examples	Searching and Sorting algorithms (linear, Binary search, Merge Sort, Quick Sort), shortest path algorithms (Dijkstra's Algorithm). etc	0/1 Knapsack Problem, Traveling salesman problem, N queen problem, Sudoku, Subset Sum, Hamiltonian Path Problem etc.
Relation	All P problems are in NP.	Not all NP problems are in P. It's still an open question whether P = NP.
Determinism	Solved using a deterministic algorithm (one that behaves predictably and follows specific rules).	Requires a nondeterministic algorithm, which in theory guesses a solution and verifies it quickly.
Ease of Solution	These problems are considered "easy" or "tractable" because they can be solved efficiently.	These problems are "harder" to solve, as we don't know an efficient algorithm to solve them (yet).

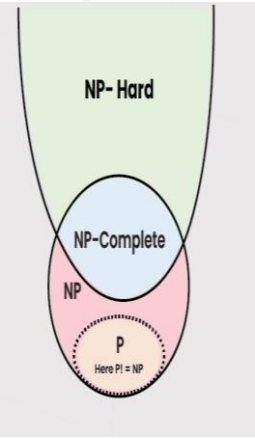
NP-Hard (Nondeterministic Polynomial Hard):

- Problems that are at least as hard as the hardest problems in NP, but not necessarily in NP themselves.
- If we can solve an NP-Hard problem in polynomial time, then we can solve all NP problems in polynomial time.
- These problems may not be decision problems (they can be optimization problems too).
- **Examples:** Traveling Salesman Problem (TSP), Halting Problem.
- NP-Hard problems are not necessarily in NP (meaning they may not have solutions that can be verified in polynomial time).
- NP-Hard problems are as hard or harder than NP problems, but they are not required to have a polynomial-time verification process.

NP-Complete

- Problems that are both in NP and NP-Hard.
- They are the "hardest" problems in NP.
- If any NP-Complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time.
- **Examples:** Boolean Satisfiability Problem (SAT), 3-SAT, Clique Problem.
- All NP-Complete problems can be reduced to each other using polynomial-time reductions.
- These are decision problems where a solution can be verified in polynomial time and, at the same time, are at least as hard as every problem in NP.

Some Important Point



- **P ⊆ NP:** Every problem that is solvable in polynomial time can also be verified in polynomial time.
- **NP-Complete ⊆ NP**: NP-Complete problems are a subset of NP problems.
- **NP-Hard ⊇ NP-Complete**: NP-Hard problems include NP-Complete problems but also other problems that might not even be in NP.
- **Unresolved Question (P vs NP):** It is still unknown whether P = NP. If P = NP, then every problem in NP, including all NP-Complete problems, could be solved in polynomial time.
- P ⊆ NP ⊆ NP-Hard.
- NP-Complete is a subset of NP-Hard and NP.

Examples:
P: Binary Search, Minimum Spanning Tree, Matrix Multiplication.
NP: Hamiltonian Cycle, Subset Sum (solutions can be verified in polynomial time).
NP-Hard: TSP (optimization version), Halting problem.
NP-Complete: Boolean Satisfiability Problem (SAT)., 3-SAT, Graph Coloring 3SAT (logical satisfaction problem), knapsack problem, Hamiltonian path.

Hashing

Hashing is a popular technique for storing and retrieving data as fast as possible. The main reason behind using hashing is that it gives optimal results as it performs optimal searches.

- In other words hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements.
- Hashing is also used in data encryption. Passwords can be stored in the form of their hashes.
- A simple hashing approach would be to take the modular of the key (assuming it's numerical) against the table's size:
  index=key MOD tableSize (k mod m)

- Time Complexity: Average: O(1) and Worst Case: O(n)

Hash Function

A function which process the given data and generate a fixed range value to store in a hash table. A hash table uses the hash function to efficiently map data such that it can be retrieved and updated quickly.

- Hash function takes the data item as an input and returns a small integer value as an output called hash value.
- Hash value of the data item is then used as an index for storing it into the hash table.
- The hash function is used to compute the index of the array (hash table).

Properties of a Good Hash Function

- Efficiently computable.
- Uniform key distribution.
- Minimal collisions.
- Low load factor (number of items / table size).
- Perfect Hash Function: Maps each key to a unique slot (no collisions).

Hash Table

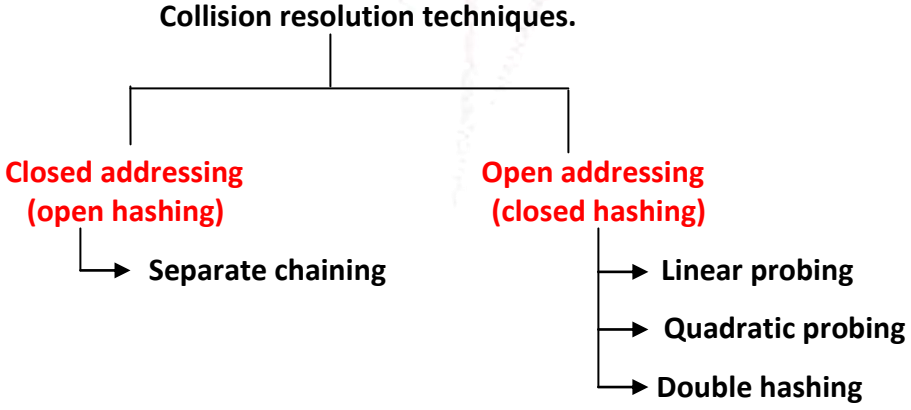
An array-like data structure that stores data based on the hash key generated by a hash function.
Hash Key Value A unique index generated using the hash function for efficient data retrieval.

Hashing Collision

When two or more keys have the same hash value, resulting in a collision.
Example: If two keys (e.g., 7 and 17) using hash function h(k) = h(k) mod 10. both map to index 7, a collision occurs.

Collision resolution techniques

Collision resolution is a technique to resolve the collision occur in storing the data in the hash table by using hashing.
There are two types of collision resolution techniques.



1. **Closed Addressing (Open Hashing):** Each hash table index stores multiple keys using a linked list or other structures.
   Separate Chaining:
   - Each index in the hash table points to a linked list.
   - When a collision occurs, the new value is added to the linked list at that index.
   - **Pros:** Easy to implement; hash table can grow dynamically.
   - **Cons:** Increased space usage due to linked lists.

2. **Open Addressing (Closed Hashing):** When a collision occurs, the algorithm searches for another empty slot within the hash table.

a) Linear Probing: To resolve the collision we use function h(k, i) = [h(k) + i] mod m, where m = size of the hash table, h(k) = given hash function to find the location of the key value in the hash table, i = the probe number that varies from 0 to m–1.

- Therefore, for a given key k, the first location is generated by [h(k) + 0] mod m, the first time i=0.
- If the location is free, the value is stored at this location. If value successfully stores then probe count is 1 means location is founded on the first go.
- If location is not free then second probe generates the address of the location given by [h(k) + 1]mod m.
- Similarly, if the generated location is occupied, then subsequent probes generate the address as [h(k) + 2]mod m, [h(k) + 3]mod m, [h(k) + 4]mod m, [h(k) + 5]mod m, and so on, until a free location is found.
- Probes is a count to find the free location for each value to store in the hash table.
- **Pros:** Simple to implement.
- **Cons:** Can lead to clustering (consecutive filled slots) primary and secondary both.
- **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
- **Secondary Clustering:** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

Time Complexity-

Worst time to search an element in linear probing is O(table size), if n is the size of table then O(n). Deletion will also take linear time. First we search the nth element in O(n) time and then delete it in O(n) time.

b) Quadratic Probing: To resolve primary clustering we can use quadratic probing to find the location for the key when collision occur.

- In this technique, if a value is already stored at a location generated by hash function h(k), then the following hash function is used to resolve the collision:
  h(k, i) = (h(k) + i^2) mod m
  where m is the size of the hash table,
  h(k) =given hash function to find the location of the key value in the hash table,
  i is the probe number that varies from 0 to m–1,
  - else new locations will be generated using hash function [h(k) + i^2] mod m.
  - Value of i will change until free space is founded and probe count is increased until free space is founded.

Advantages of quadratic probing:-

- Quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table.

Disadvantages of quadratic probing:-

- it does not search all locations of the list.
- It also suffers from secondary cluster.

c) Double Hashing: To resolve secondary clustering we use double hashing function. In this technique, if the collision occurs we use two hash functions to calculate empty slot to store value.

- In the case of collision we take the second hash function $h_2(k)$ and look for i * $h_2(k)$ free slot in an ith iteration.
- Double hashing requires more computational time because two hash functions need to be computed.

- In double hashing, we use two hash functions rather than a single function. To resolve the collision the hash function in the case of double hashing can be given as:

$$H(k, i) = [h_1(k) + i*h_2(k)] \bmod m$$

where m is the size of the hash table,
$h_1(k)$ and $h_2(k)$ are two hash functions
$h_1(k)$ = given hash function to find the location of the key value in the hash table,
$h_2(k)$ = If collision occurs then calculate this function in double hashing collision resolution.
i is the probe number that varies from 0 to m–1

- When we have to insert a key k in the hash table, we first probe the location given by applying
- $[h_1(k) \bmod m]$ because during the first probe, i = 0. If the location is vacant, the key is inserted into it.
- And if the location is not vacant then increase value of i to calculate next location using
- $h(k,1) = [h_1(k) + 1*h_2(k)] \bmod m$.
- Otherwise
- $h(k,0) = [h_1(k) + 0*h_2(k)] \bmod m$ for next key

Advantages of double hashing:-

- Double hashing resolves primary and secondary clustering.

Disadvantages of double hashing:-

- it does not search all locations of the list.
- Double hashing has poor cache performance
- Double hashing requires more computation time as two hash functions need to be computed.