Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and
Communication Technology

Introduction and Analysis of Algorithms
Chapter 1

Mrs. Bhumi Shah
Assistant Professor
Computer Science and Engineering

Content

What is Algorithm?

- The word algorithm comes from the name of a Persian mathematician Abu Ja’far Mohammed ibn-i Musa al Khowarizmi.
- Algorithm is a finite set of instructions used to accomplish particular task.

An algorithm takes some value, or set of values, as input and produces some value, or set of values, as output.

What is Algorithm?

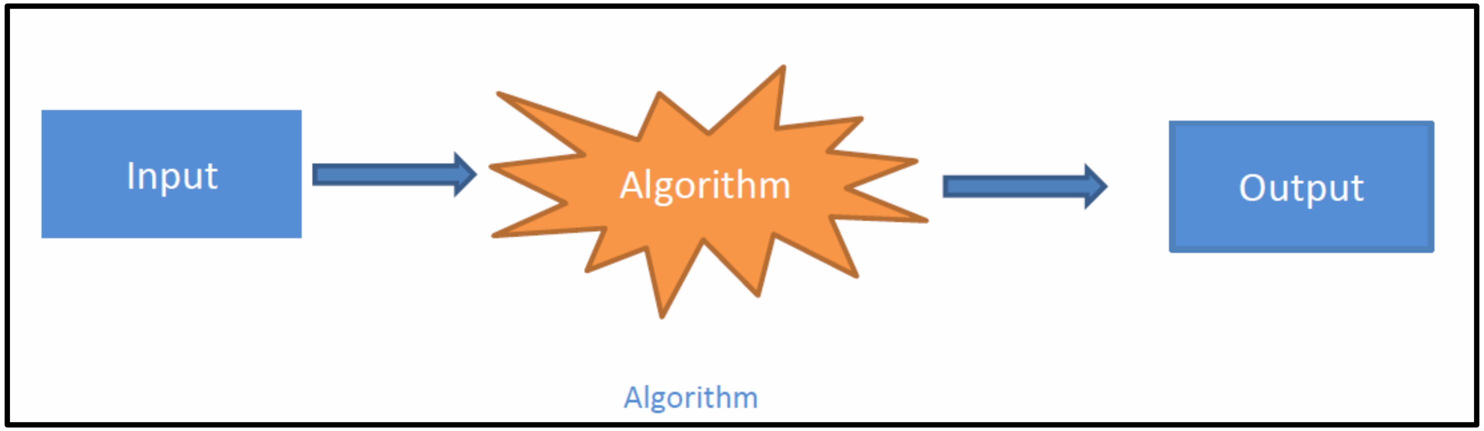


Fig 1.1 Algorithm

Properties of Algorithm

- Clear and Unambiguous: Each step should be clear and lead to only one interpretation .
- Well-Defined Inputs: If an algorithm takes inputs, they should be well-defined. An algorithm may or may not require inputs .
- Well-Defined Outputs: The algorithm must clearly define the output and produce at least one output .
- Finiteness: The algorithm must terminate after a finite number of steps .
- Feasible: It must be simple, practical, and executable with available resources, without relying on future technology .

Properties of Algorithm

- Language Independent: The algorithm should be independent of any specific programming language .
- Deterministic: For the same input, the algorithm should produce the same output .
- Effective: Every step in the algorithm must perform some work

Types of Algorithm

- Sorting Algorithms: These algorithms rearrange a given array or list of elements in a specific order . Examples include merge sort and quicksort
- Searching Algorithms: Used to find specific items within a collection of data . Common examples include linear search and binary search .
- Greedy Algorithms: These make locally optimal choices at each step in hopes of finding a global optimum solution .
- Dynamic Programming: An algorithmic technique that optimizes recursion by storing results of repeated calls for the same inputs .
- Backtracking Algorithms: These explore different options to find the best solution by trying different paths and backtracking when a path doesn't work .
- Divide and Conquer Algorithms: This strategy involves dividing a problem into smaller subproblems, solving them, and combining the solutions .

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Introduction and Analysis of Algorithms
Chapter 1

Mrs. Bhumi Shah
Assistant Professor
Computer Science and Engineering

Content

What is Algorithm Analysis?

- Definition: Evaluating an algorithm's performance in terms of time and space .
- Goal: To estimate the resources (time, memory) that an algorithm consumes .
- **Importance:**
- Choosing the most efficient algorithm for a task .
- Understanding scalability with increasing input size

Parameters for Algorithm Analysis

Time Complexity

- *Definition:* The amount of time required by an algorithm to complete its execution.
- *Notation:* Big O, Big Omega, Big Theta.
- *Common Time Complexities:*
    - O(1) - Constant Time
    - O(n) - Linear Time
    - O(n log n) - Linearithmic Time
    - O(n^2) - Quadratic Time
    - O(log n) - Logarithmic Time
    - O(2^n) - Exponential Time

Parameters for Algorithm Analysis

Space Complexity

•Definition: The amount of memory space required by an algorithm.

•Components: Instruction space, data space, environment stack space.

•Importance: Essential for algorithms processing large datasets.

Algorithm Design Techniques

Divide and Conquer

•*Concept:* Break down a problem into smaller subproblems, solve them recursively, and combine their solutions.

•*Examples:* Merge Sort, Quick Sort, Binary Search.

Dynamic Programming

•*Concept:* Solve overlapping subproblems by storing their solutions to avoid recomputation.

•*Examples:* Fibonacci sequence, Knapsack problem etc.

•**Greedy Algorithms**

•*Concept:* Make locally optimal choices at each step with the hope of finding a global optimum.

•*Examples:* Dijkstra's algorithm, Prim's algorithm, Huffman coding.

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Introduction and Analysis of Algorithms
Chapter 1

Mrs. Bhumi Shah
Assistant Professor
Computer Science and Engineering

Content

Why algorithm analysis?

- For one problem one or more solutions are available.
- Which one is better ? How can we choose?
- The analysis of an algorithm can help us in understanding of solution in better way.
- Time & Space analysis Image

Order of Growth

- Any algorithm is expected to work fast for any input size.
- Smaller input size algorithm will work fine but for higher input size execution time is much higher.
- So how the behavior of algorithm changes with the no. of inputs will give the analysis of the algorithm and is called the Order of Growth.

Algorithm falls under three types

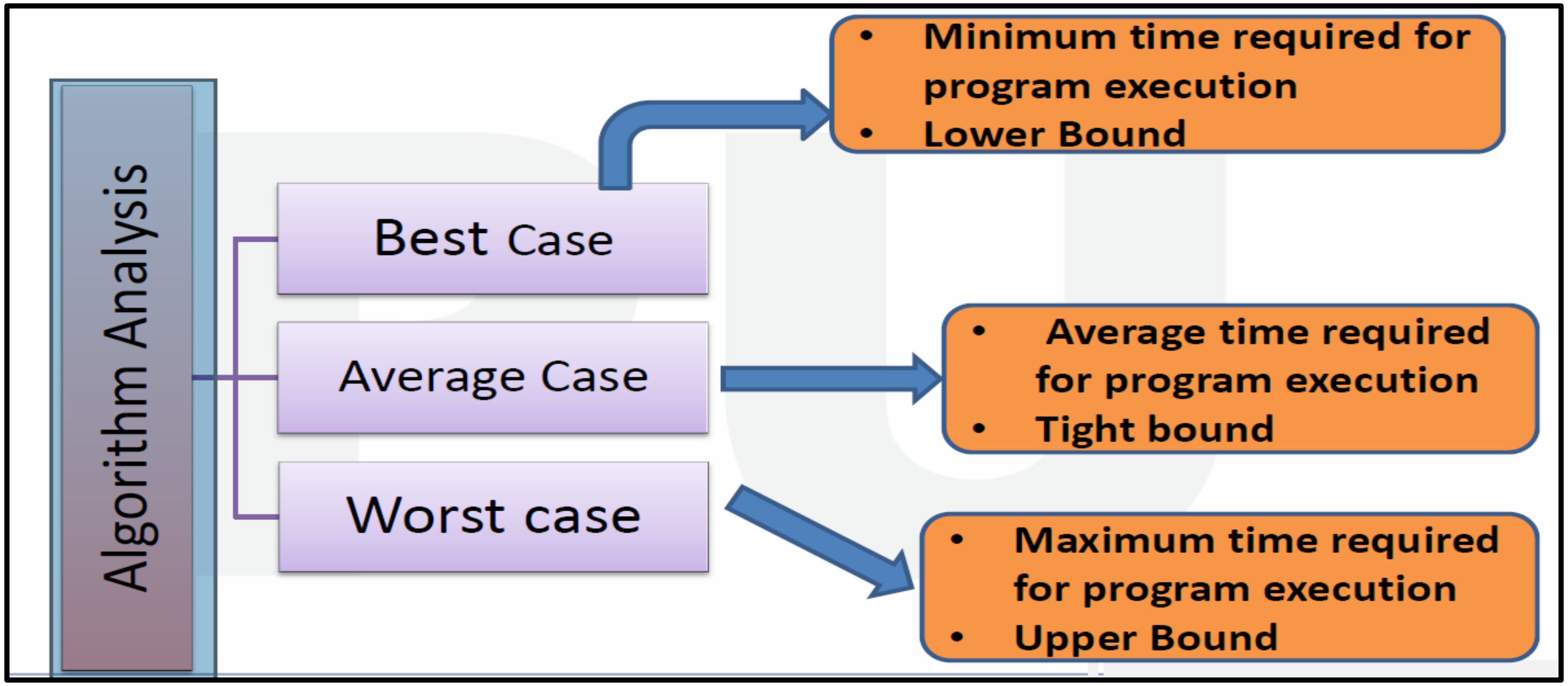
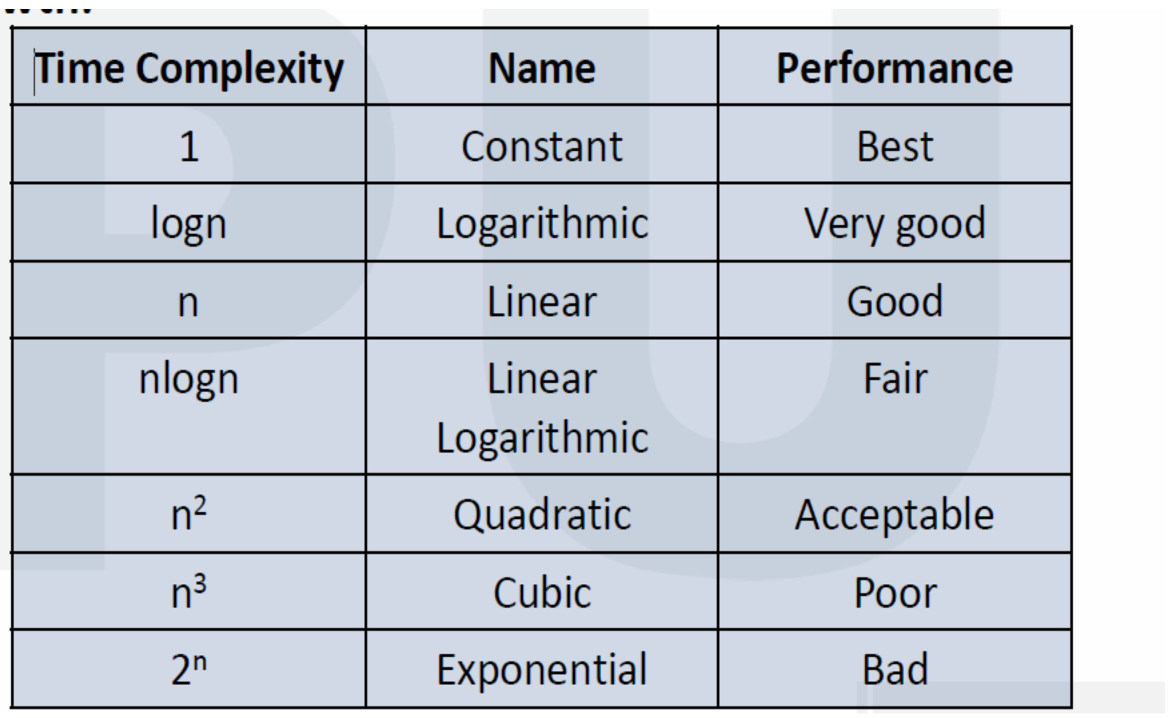

Algorithm Analysis
Best Case
Average Case
Worst case
• Minimum time required for program execution
• Lower Bound
• Average time required for program execution
• Tight bound
• Maximum time required for program execution
• Upper Bound

Fig 1.1 :Algorithm falls under three types

Rate of Growth

Rate at which the running time increases as a function of input is called rate of growth.



Time Complexity	Name	Performance
1	Constant	Best
logn	Logarithmic	Very good
n	Linear	Good
nlogn	Linear Logarithmic	Fair
n^2	Quadratic	Acceptable
n^3	Cubic	Poor
2^n	Exponential	Bad

Asymptotic Complexity

- Refers to defining the mathematical bound of its run-time performance.
- Running time of an algorithm as a function of input size n for large n.
- Asymptotic means approaching a value or curve arbitrarily.

Asymptotic Notations

- Refers to defining the mathematical bound of its run-time performance.
- Running time of an algorithm as a function of input size n for large n.
- Asymptotic means approaching a value or curve arbitrarily.

O Notation

•express the tight upper bound of an algorithm

•f(n)=O(g(n)) implies: O(g(n)) = { f(n) : there exists positive constants c>0 and n0 such that f(n) ≤ c.g(n) for all n > n0. }

Asymptotic Notations

Ω Notation

•express the lower bound of an algorithm

•f(n)= ⍰ (g(n)) implies: ⍰ (g(n)) = { f(n) : there exists positive constants c>0 and n0 such that f(n) ≥ g(n) for all n > n0. }

θ Notation

•express both the lower bound and the upper bound (tight bound)”

•f(n)= ⍰ (g(n)) implies: ⍰ (g(n)) = { f(n) : there exists positive constants c1>0, c2>0 and n0 such that c1.g(n) ≤ f(n) ≤ c2.g(n) for all n > n0. }

OR, if & only if f(n)=O(g(n)) and f(n)= ⍰ (g(n)) for all n > n0

Asymptotic Notations

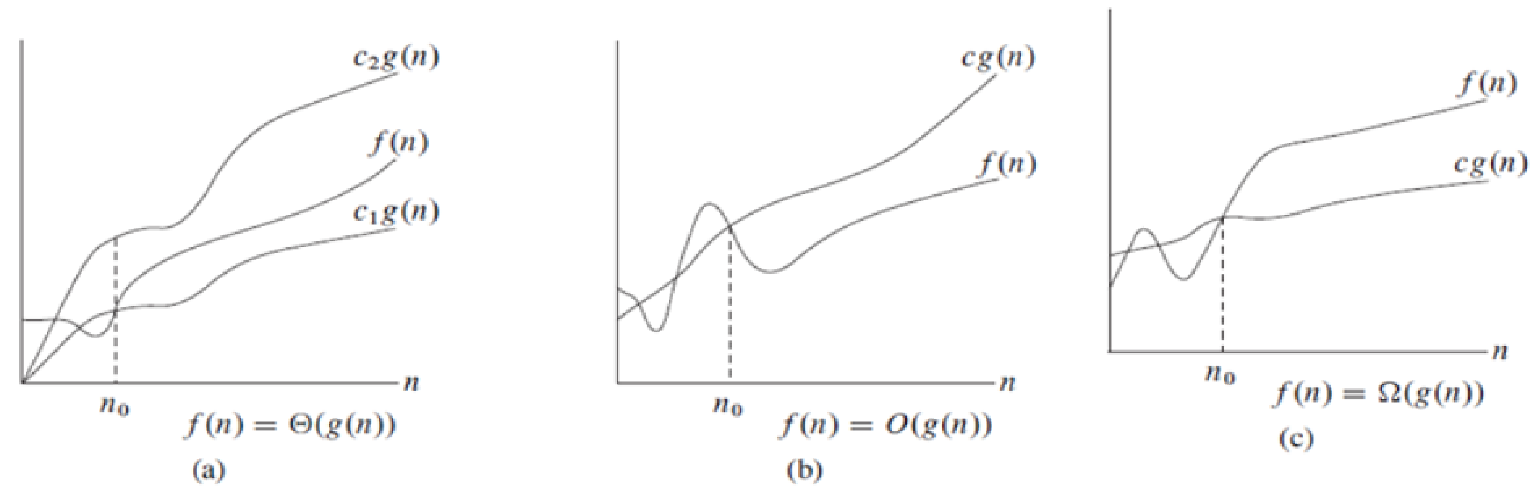


Fig: Asymptotic average bound (a), upper bound (b)and lower bound(c)
Image source:http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap02.htm

Asymptotic Notations

Examples on Big-O Notation(Upper bound/Worst case)

1)find upper bound for f(n)=2n+2

solution:2n+2<=4n, for all n>=1 so 2n+2=O(n) with c=4 and n0 =1

2)find upper bound for f(n)=2n2+1

solution:2n2+1<=3n2, for all n>=1 so 2n2+1=O(n2) with c=3 and n0 =1

3)Find upper bound for f(n)=5n4+4n+1

Solution : 5n4+4n+1 <=7n4, for all n>=2 so 5n4+4n+1 =O(n4) with c=7 and n0 =2

3)Find upper bound for f(n)=5n4+4n+1

Solution : 5n4+4n+1 <=7n4, for all n>=2 so 5n4+4n+1 =O(n4) with c=7 and n0 =2

Now Try to solve below Questions

I. Find upper bound for f(n)=200

II. Find upper bound for f(n)=n3+n2

III. Show that 20n3=O(n4) for appropriate c and n0.

Asymptotic Notations

Examples on Big- Ω Notation(Lower bound/ Best case)

1)find lower bound for f(n)=2n+2

solution: 2n<=2n+2, for all n>=1 so 2n+2= Ω(n) with c=2 and n0 =1

2)find lower bound for f(n)=2n2+1

solution: n2<=2n2+1, for all n>=1 so 2n2+1= Ω(n2) with c=1 and n0 =1

3)Find lower bound for f(n)=5n4+4n+1

Solution : 4n4 <=5n4+4n+1, for all n>=1 so 5n4+4n+1 = Ω(n4) with c=4 and n0 =1

Now Try to solve below Questions

I. For $\sqrt{n}+54 = \Omega(\lg n)$. Choose c and n_0

II. Any linear *function* $an + b$ is in $\Omega(n)$. **How?**

Asymptotic Notations

Examples on Big- θ Notation(Tight bound/ Average case)

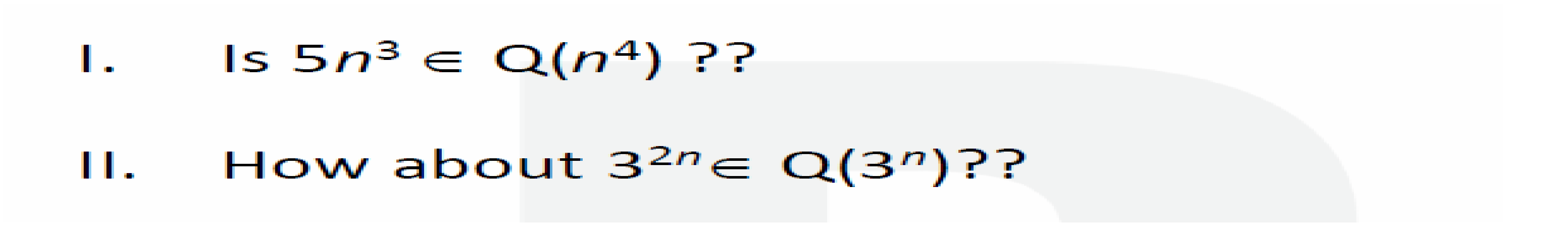
1)find tight bound for f(n)=2n+2

solution: 2n<=2n+2<=4n, for all n>=1 so 2n+2= θ(n) with c1 =2, c2 =4 and n0 =1

2)find lower bound for f(n)=2n2+1

solution: n2<=2n2+1<=3n2, for all n>=1 so 2n2+1= θ(n2) with c1 =1, c2 =3 and n0 =1

Now Try to solve below Questions



I. Is $5n^3 \in Q(n^4)$??

II. How about $3^{2n} \in Q(3^n)$??

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Introduction and Analysis of Algorithms
Chapter 1

Mrs. Bhumi Shah
Assistant Professor
Computer Science and Engineering

Content

1. Algorithm
2. Writing an Algorithm Analysis
3. Asymptotic Analysis
4. Analyzing control statement, Loop invariant and the correctness of the algorithm
5. Recurrences
6. Sorting Techniques with analysis

Analysing control statement

Control Statements are:

1. Sequencing
2. For Loop
3. While and Repeat loop

1. Sequencing

Sequencing means putting one instruction after another

Example

int i=10; ……………. 1

printf("%d",&i);………..1

i=i+5;……………………..1

Here each instruction execute only once then

frequency count=1+1+1=3

2. For Loops

```
Main()
{
X=y+z;------------------------------O(1)
For(i=1;i<n;i++)---------------------------------O(n)
{
A=b+c;---------------------------O(1)
}
}
```

Total Frequency Count=O(1)+O(n)=O(1+n)=O(n)

3. While and Repeat Loops

```
1) While(n>0)-----------O(n)
{
n=n-1;----O(n)
}
 total Frequency count= O(n)

2)
While(n>0)
{
n=n/2;…………….O(log n)
}
Total frequency count= O(log n)
```

What is a Loop Invariant?

A loop invariant is a condition or property that:

- Holds true before the loop starts,
- Remains true after every iteration of the loop,
- And helps in proving the correctness of the algorithm.

Why is it important?

Loop invariants are used to **prove that a loop works correctly** and contributes to the overall correctness of an algorithm.

Steps to Use a Loop Invariant

1. **Initialization**: True before the first iteration.
2. **Maintenance**: Remains true during each iteration.
3. **Termination**: Helps prove the final result is correct when the loop ends.

Example: Finding Minimum Value

```
int min = arr[0];
for (int i = 1; i < n; i++) {
    if (arr[i] < min)
        min = arr[i];
}
```

Loop Invariant:At the start of each iteration, min holds the minimum of elements arr[0] to arr[i-1].

Correctness of the Algorithm

An algorithm is **correct** if:

- It **produces the correct output** for all valid inputs.
- It **terminates** (does not go into infinite loop).

Correctness = Partial Correctness + Termination

- ✅ **Partial correctness**: Loop invariants help prove that the logic is sound.
- ✅ **Termination**: Prove the loop stops after a finite number of steps.

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Parul® University
Vadodara, Gujarat
NAAC GRADE A++

Information and
Communication Technology

Introduction and Analysis of Algorithms
Chapter 1

Mrs. Bhumi Shah
Assistant Professor
Computer Science and Engineering

Content

Methods For Solving Recurrence Relations

1. Substitution method
2. Recursion tree method
3. Master method.

1. Substitution method

Substitution Method is very famous method for solving any recurrences. There are two types of substitution methods-

1. Forward Substitution
2. Backward Substitution

Forward Substitution

It is called Forward Substitution because here we substitute recurrence of any term into next terms. It uses following steps to find Time using recurrences-

- Pick Recurrence Relation and the given initial Condition
- Put the value from previous recurrence into the next recurrence
- Observe and Guess the pattern and the time
- Prove that the guessed result is correct using mathematical Induction.

Forward Substitution

Example

T(n) = T(n-1) + n, n>1

T(n) = 1, n=1

Solution:

1. Pick Recurrence and the given initial Condition:

T(n)=T(n-1)+n, n>1T(n)=1, n=1

2. Put the value from previous recurrence into the next recurrence:

T(1) = 1T(2) = T(1) + 2 = 1 + 2 = 3T(3) = T(2) + 3 = 1 + 2 + 3 = 6T(4)= T(3) + 4 = 1 + 2 + 3 + 4 = 10

3. Observe and Guess the pattern and the time:

So guessed pattern will be-T(n) = 1 + 2 + 3 .... + n = (n * (n+1))/2Time Complexity will be O(n2)

Forward Substitution

4. Prove that the guessed result is correct using mathematical Induction:

Prove T(1) is true:
T(1) = 1 * (1+1)/2 = 2/2 = 1 and from definition of recurrence we know T(1) = 1. Hence proved T(1) is true
Assume T(N-1) to be true:
Assume T(N-1) = ((N - 1) * (N-1+1))/2 = (N * (N-1))/2 to be true
Then prove T(N) will be true:T(N) = T(N-1) + N from recurrence definition
Now, T(N-1) = N * (N-1)/2So, T(N) = T(N-1) + N = (N * (N-1))/2 + N
= (N * (N-1) + 2N)/2 =N * (N+1)/2And from our guess also T(N)=N(N+1)/2Hence T(N) is true.Therefore our guess was correct and **time will be O(N2)**

Backward Substitution

It is called Backward Substitution because here we substitute recurrence of any term into previous terms. It uses following steps to find Time using recurrences-

- Take the main recurrence and try to write recurrences of previous terms
- Take just previous recurrence and substitute into main recurrence
- Again take one more previous recurrence and substitute into main recurrence
- Do this process until you reach to the initial condition
- After this substitute the the value from initial condition and get the solution

Backward Substitution

Example:
T(n) = T(n-1) + n, n>1T(n) = 1, n = 1
Solution:
1. Take the main recurrence and try to write recurrences of previous terms:
T(n) = T(n-1) + nT(n-1) = T(n-2) + n - 1T(n-2) = T(n-3) + n – 2
2. Take just previous recurrence and substitute into main recurrence
put T(n-1) into T(n)So, T(n)=T(n-2)+ n-1 + n
3. Again take one more previous recurrence and substitute into main recurrence
put T(n-2) into T(n)So, T(n)=T(n-3)+ n-2 + n-1 + n

Backward Substitution

4. Do this process until you reach to the initial condition
So similarly, we can find T(n-3), T(n-4)......and so on and can insert into T(n). Eventually we will get following: T(n)=T(1) + 2 + 3 + 4 +.........+ n-1 + n

5. After this substitute the the value from initial condition and get the solution

Put T(1)=1, T(n) = 1 +2 +3 + 4 +.............+ n-1 + n = n(n+1)/2. So Time will be O(N2)

Recursion tree method

In this method, a recurrence relation is converted into recursive trees. Each node represents the cost incurred at various levels of recursion. To find the total cost, costs of all levels are summed up.

Steps to solve recurrence relation using recursion tree method:

- Draw a recursive tree for given recurrence relation
- Calculate the cost at each level and count the total no of levels in the recursion tree.
- Count the total number of nodes in the last level and calculate the cost of the last level
- Sum up the cost of all the levels in the recursive tree

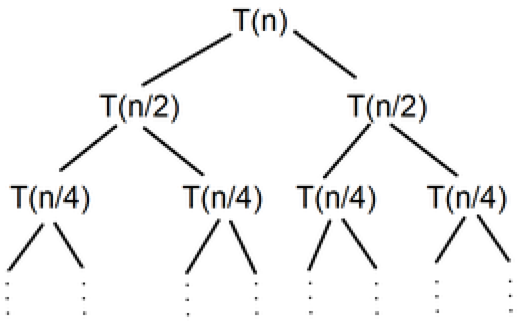
Recursion tree method

Question 1:

$T(n) = 2T(n/2) + c$

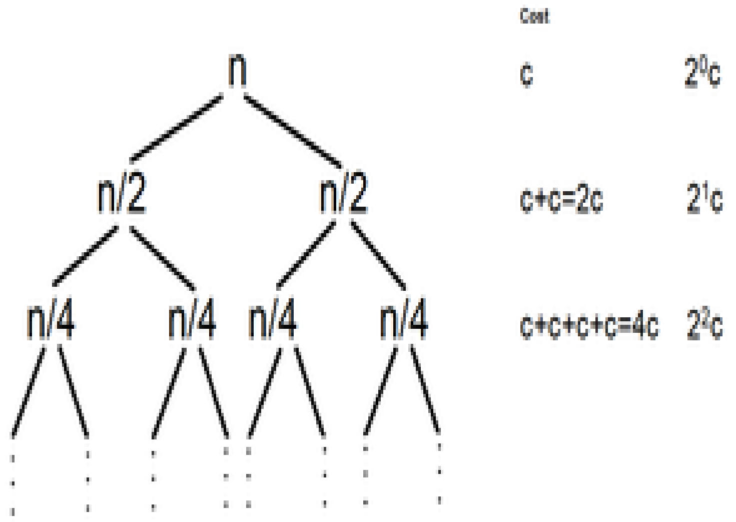
Solution:

Step 1: Draw a recursive tree



Recursion tree method

Step 2: Calculate the work done or cost at each level and count total no of levels in recursion tree



Recursion tree method

Count the total number of levels –

Choose the longest path from root node to leaf node

n/20 -→ n/21 -→ n/22 -→ ……… -→ n/2k

Size of problem at last level = n/2k

At last level size of problem becomes 1

n/2k = 1

2k = n

k = log2(n)

Total no of levels in recursive tree = k +1 = log2(n) + 1

Recursion tree method

• **Step 3:** Count total number of nodes in the last level and calculate cost of last level

No. of nodes at level 0 = $2^0 = 1$

No. of nodes at level 1 = $2^1 = 2$

………………………………………………………

No. of nodes at the (k+1)th level (or last level) = $2^k = 2^{\log_2(n)} = n^{\log_2(2)} = n$

Cost of sub problems at level last level = n x $\Theta(1) = \Theta(n)$

Recursion tree method

- **Step 4:** Sum up the cost all the levels in recursive tree

$T(n) = c + 2c + 4c +$ —- ((no. of levels-1) times) + last level cost

$= c + 2c + 4c +$ —- ($\log_2(n)$ times) + $\Theta(n)$

$= c(1 + 2 + 4 +$ —- ($\log_2(n)$ times)) + $\Theta(n)$

$1 + 2 + 4 +$ —– ($\log_2(n)$ times) –> $2^0 + 2^1 + 2^2 +$ —– (log2(n) times) –> Geometric Progression(G.P.)

Using GP Sum Formula, we get the sum of the series as n

$= c(n) + \Theta(n)$

Master method.

- The time complexity of the algorithm is represented in the form of recurrence relation.
- When analyzing algorithms, recall that we only care about the asymptotic behavior
- Rather than solving exactly the recurrence relation associated with the cost of an algorithm, it is sufficient to give an asymptotic characterization
- The main tool for doing this is the master theorem

Master method.

Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where $a \geq 1$, $b \geq 2$, $c>0$. If f(n) is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master method.

Example 1

Let T(n) = T(n/2) + ½ n2 + n.

What are the parameters?

a = 1

b = 2

d = 2 $1 < 2^2$, case 1 applies

Therefore, which condition applies?

$$T(n) \in \Theta(n^d) = \Theta\ (n^2)$$

Master method.

Example 2

- Let T(n)= 2 T(n/4) + √n + 42. What are the parameters?

a =	2
b =	4
d =	1/2

Therefore, which condition applies?

$2 = 4^{1/2}$, case 2 applies

- We conclude that

$$T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$$

Master Method.

Example 3

- Let T(n)= 3 T(n/2) + 3/4n + 1. What are the parameters?

  a = 3

  b = 2

  d = 1

  Therefore, which condition applies?

  $3 > 2^1$, case 3 applies

- We conclude that

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

- Note that $\log_2 3 \approx 1.584\ldots$, can we say that $T(n) \in \Theta(n^{1.584})$

  No, because $\log_2 3 \approx 1.5849\ldots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/

Introduction and Analysis of Algorithms
Chapter 1

Mrs. Bhumi Shah
Assistant Professor
Computer Science and Engineering

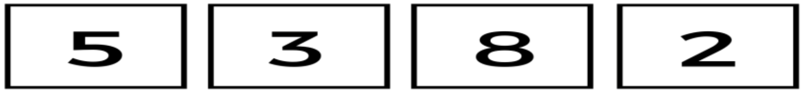
Content

Sorting Techniques with analysis

1. Bubble Sort
2. Selection Sort
3. Insertion sort

Bubble Sort

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.



Bubble Sort

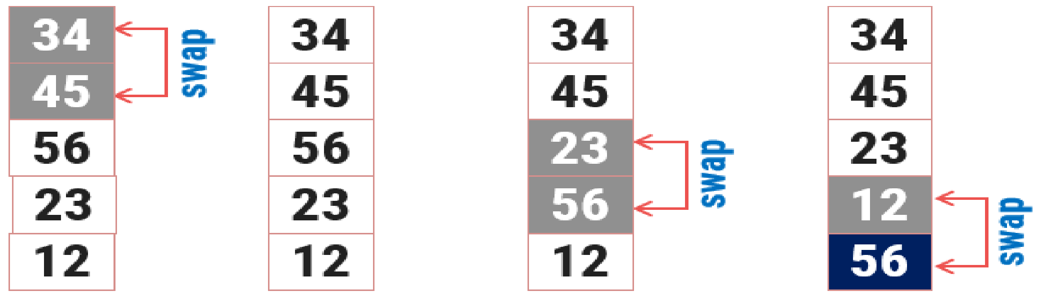
- In Bubble sort, each element is compared with its adjacent element. If the first element is smaller than the second one, then the positions of the elements are interchanged, otherwise it is not changed.
- Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array until we get a sorted array.

Bubble Sort Example

- Sort the following array in Ascending order

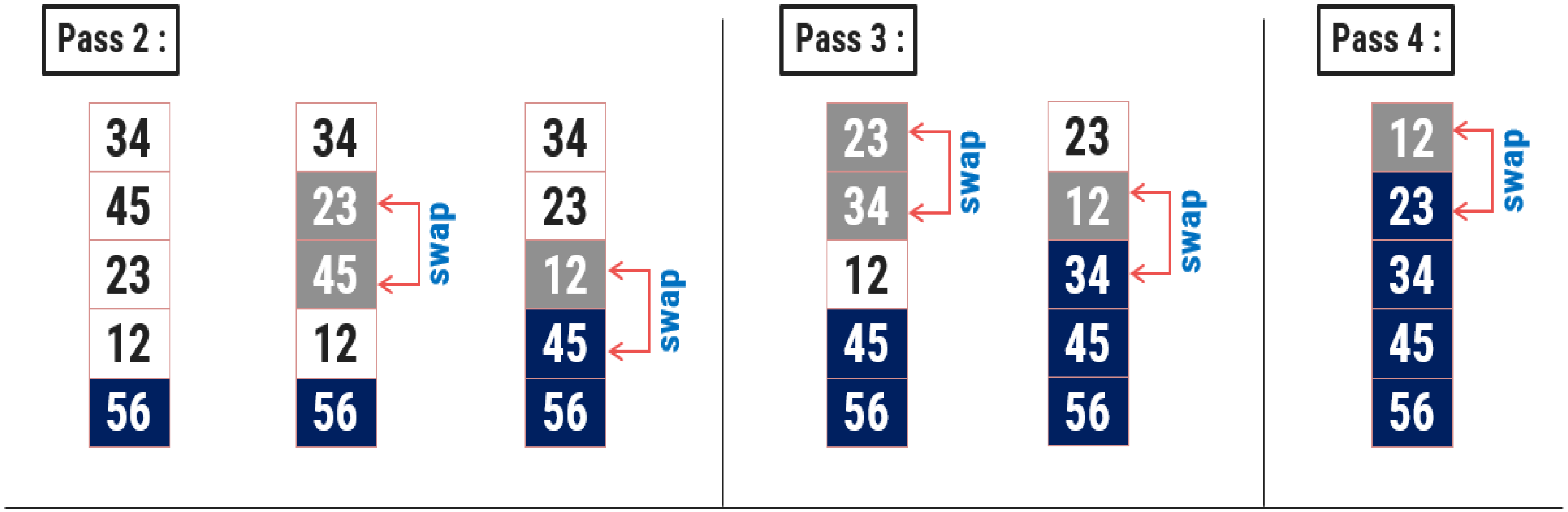
45	34	56	23	12

Pass 1 :



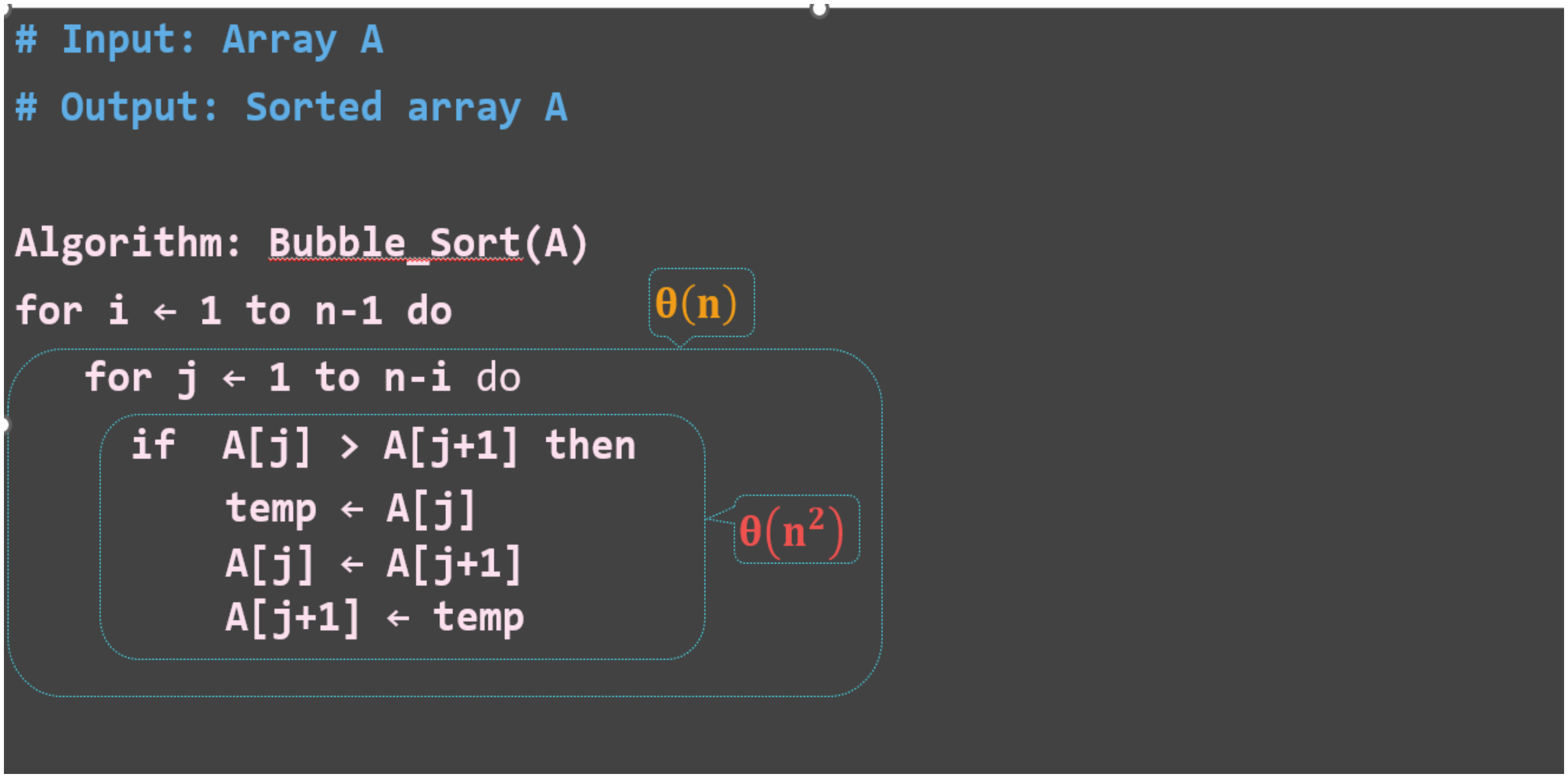
```
if (A[j] > A[j + 1])
    swap(A[j], A[j + 1])
```

Bubble Sort Example



$$\begin{aligned}&if\,(A[j] > A[j+1])\\&\quad swap(A[j], A[j+1])\end{aligned}$$

Bubble Sort Algorithm



```
# Input: Array A
# Output: Sorted array A

Algorithm: Bubble_Sort(A)
for i ← 1 to n-1 do
   for j ← 1 to n-i do
      if  A[j] > A[j+1] then
          temp ← A[j]
          A[j] ← A[j+1]
          A[j+1] ← temp
```

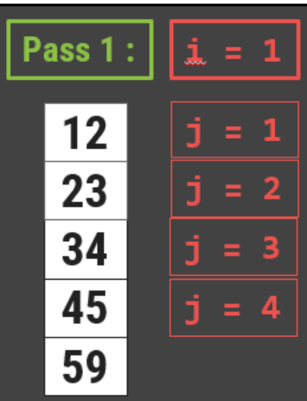
Bubble Sort

- It is a simple sorting algorithm that works by comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use.
- The time complexity of bubble sort is $\boldsymbol{\theta}(\boldsymbol{n}\text{^}\mathbf{2}\ )$

Bubble Sort Best case Analysis

```
# Input: Array A
# Output: Sorted array A
Algorithm: Bubble_Sort(A)
int flag=1;
for i ← 1 to n-1 do
     for j ← 1 to n-i do
          if A[j] > A[j+1] then
               flag = 0;
               swap(A[j],A[j+1])
     if(flag == 1)
          cout<<"already sorted"<<endl
          break;
```

Condition never becomes true



Best case time complexity = $\theta(n)$

Analysis of Bubble Sort

Algorithm

```
Procedure bubble (T[1….n])                          cost     times
for i ← 1 to n do                                   C1       n+1
     for i ← 1 to n-i do                            C2       $\sum_{i=1}^{n}(n+1-i)$
          if T[i] > T[j]                            C3       $\sum_{i=1}^{n}(n-i)$
               T[i] ↔ T[j]                          C4       $\sum_{i=1}^{n}(n-i)$
     end
end
```

Analysis

$$T(n)=C1\,(n+1) + C2\ \sum_{i=1}^{n}(n+1-i) + C3 \sum_{i=1}^{n}(n-i) + C4 \sum_{i=1}^{n}(n-i)$$

Analysis of Bubble Sort

Best case:

Take i = 1

$$T(n) = C_1 n + C_1 + C_2 n + C_3 n - C_3 + C_4\, n - C_4$$
$$= (C_1+C_2+C_3+C_4)\, n - (C_2, C_3, C_4, C_7)$$
$$= an - b$$

Thus, $T(n) = \Theta(n)$

Worst Case:

$$= C_1 n + C_1 + C_2\, n + C_2 - C_2\left(\frac{n(n+1)}{2}\right) + C_3 n - C_3\left(\frac{n(n+1)}{2}\right) + C_4 n - C_4\left(\frac{n(n+1)}{2}\right)$$
$$= [-C_2/n^2 - C_3/n^2 - C_4/n^2] + [-\,C_2/n - C_3/n - C_4/n] + C_1 + C_2 + C_3 + C_4$$
$$= an^2 + bn + c$$
$$= \Theta\,(n^2)$$

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It divides the array into two parts:

- The sorted part (built from left to right)
- The unsorted part

On each pass:

- Find the minimum element from the unsorted part
- Swap it with the first unsorted element

Algorithm Steps

1. Start from the first element, assume it's the minimum.
2. Compare it with all other elements.
3. If any smaller element is found, update the minimum index.
4. Swap the minimum element with the first unsorted element.
5. Repeat for the remaining unsorted part.

Example of Selection Sort

Unsorted array:[64, 25, 12, 22, 11]
Pass 1:Minimum in [64, 25, 12, 22, 11] is 11
Swap 11 with 64 → [11, 25, 12, 22, 64]
Pass 2: Minimum in [25, 12, 22, 64] is 12
Swap 12 with 25 → [11, 12, 25, 22, 64]
Pass 3: Minimum in [25, 22, 64] is 22
Swap 22 with 25 → [11, 12, 22, 25, 64]
Pass 4: Minimum in [25, 64] is 25Already in place → no
swapSorted array: [11, 12, 22, 25, 64]

Selection Sort Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
minj ← i;
minx ← A[i];
for j ← i + 1 to n do
if A[j] < minx then
minj ← j;
minx ← A[j];
A[minj] ← A[i];
A[i] ← minx;
```

Selection Sort Analysis

Algorithm

```
Procedure select ( T [1….n] )                          Cost    times
        for i←1 to n-1 do                               C1      n
              minj ← i ; minx ← T[i]                    C2      n-1
        for j←i+1 to n do                               C3      $\sum_{i=1}^{n-1}(i+1)$
                 if T[j] < minx then minj ← j           C4      $\sum_{i=1}^{n-1} i$
                   minx ← T[j]                          C5      $\sum_{i=1}^{n-1} i$
        T[minj] ← T[i]                                  C6      n-1
        T[i] ← minx                                     C7      n-1
```

Analysis

$$T(n)=C_1n+ C_2(n-1)+ C_3\left(\sum_{i=1}^{n-1}(i+1)\right) + C_4\left(\sum_{i=1}^{n-1}(i)\right)+C_5\left(\sum_{i=1}^{n-1}(i)\right)+C_6(n-1)+ C_7(n-1)$$

$$= C_1n+ C_2n+ C_6n+ C_7n+ C_3\frac{n}{2}+C_4\frac{n}{2}+ C_5\frac{n}{2}+ C_3\frac{n^2}{2}+ C_4\frac{n^2}{2}+ C_5\frac{n^2}{2}-C_2-C_6-C_7$$

$$=n(C_1+ C_2+ C_6 +C_7 + C_3\frac{1}{2}+ C_4\frac{1}{2}+ C_5\frac{1}{2}) + n^2(C_3\frac{1}{2}+ C_4\frac{1}{2}+ C_5\frac{1}{2})-1(C_2+ C_6 +C_7)$$

$$= an^2+bn+c$$

Insertion Sort

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time, similar to sorting playing cards in your hand.

How It Works:

- Start from the second element.
- Compare the current element with all elements before it.
- Shift larger elements to the right.
- Insert the current element in its correct position.

Example of Insertion Sort

Initial array:[8, 4, 1, 5]

Pass 1 (i = 1):

Compare 4 with 8 → shift 8 → insert 4[4, 8, 1, 5]

Pass 2 (i = 2):Compare 1 with 8 → shift 8

Compare 1 with 4 → shift 4 → insert 1[1, 4, 8, 5]

Pass 3 (i = 3):

Compare 5 with 8 → shift 8

Compare 5 with 4 → insert 5[1, 4, 5, 8]

Final sorted array: [1, 4, 5, 8]

Insertion Sort Algorithm

Input: Array T
# Output: Sorted array T

```
Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
x ← T[i];
j ← i – 1;
while x < T[j] and j > 0 do
T[j+1] ← T[j];
j ← j – 1;
T[j+1] ← x;
```

Insertion Sort Analysis

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
        x ← T[i];
        j ← i – 1;
        while x < T[j] and j > 0 do
                T[j+1] ← T[j];
                j ← j – 1;
        T[j+1] ← x;
```

$\theta(n)$

Pass 1 :

12			
23	i=2	x=23	T[j]=12
34	i=3	x=34	T[j]=23
45	i=4	x=45	T[j]=34
59	i=5	x=59	T[j]=45

The best case time complexity of Insertion sort is $\theta(n)$
The average and worst case time complexity of Insertion sort is $\theta(n^2)$

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/