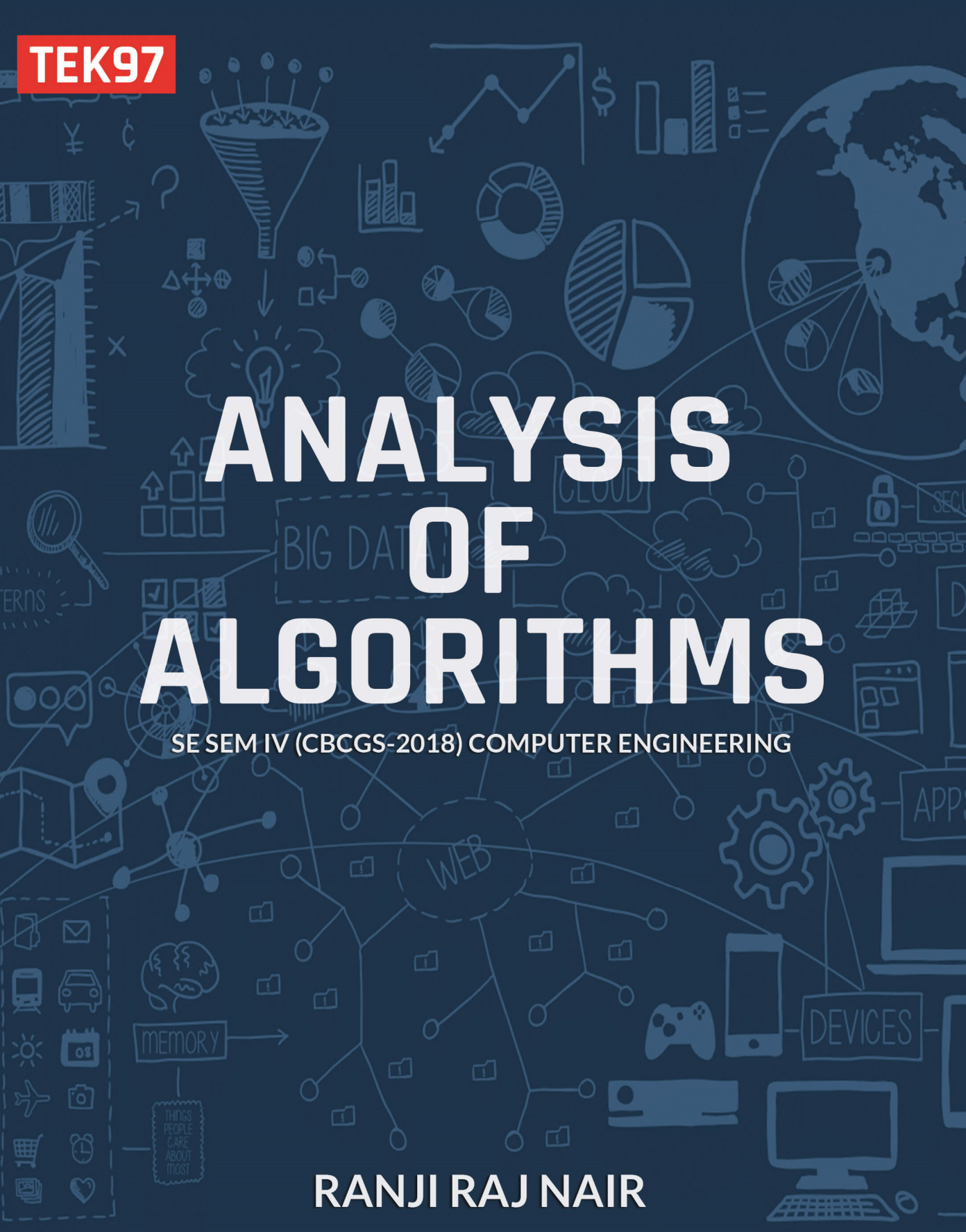
TEK97

ANALYSIS OF ALGORITHMS

SE SEM IV (CBCGS-2018) COMPUTER ENGINEERING



RANJI RAJ NAIR

TEK97

ANALYSIS OF ALGORITHMS

SE COMPUTER SCIENCE SEM IV

(New CBCGS Syllabus – Mumbai University)

RANJI RAJ NAIR

2018 Edition

Syllabus

Module	Detailed Content
1	**Introduction to analysis of algorithm** Performance analysis , space and time complexity Growth of function – Big –Oh ,Omega , Theta notation Mathematical background for algorithm analysis, Analysis of selection sort , insertion sort. **Recurrences:** -The substitution method -Recursion tree method -Master method **Divide and Conquer Approach:** General method Analysis of Merge sort, Analysis of Quick sort, Analysis of Binary search, Finding minimum and maximum algorithm and analysis, Strassen‟s matrix multiplication
2	**Dynamic Programming Approach:** General Method Multistage graphs single source shortest path all pair shortest path Assembly-line scheduling 0/1 knapsack Travelling salesman problem Longest common subsequence
3	**Greedy Method Approach:** General Method Single source shortest path Knapsack problem Job sequencing with deadlines Minimum cost spanning trees-Kruskal and prim‟s algorithm Optimal storage on tapes
4	**Backtracking and Branch-and-bound:** General Method 8 queen problem(N-queen problem)

	Sum of subsets Graph coloring 15 puzzle problem, Travelling salesman problem.
5	**String Matching Algorithms:** The naïve string matching Algorithms The Rabin Karp algorithm String matching with finite automata The knuth-Morris-Pratt algorithm
6	**Non-deterministic polynomial algorithms:** Polynomial time, Polynomial time verification NP Completeness and reducibility NP Completeness proofs Vertex Cover Problems Clique Problems

INDEX

Module 01: Introduction to Analysis of Algorithms

Q) Explain Space Complexity and Time Complexity in detail **10MKS**

1.1 Performance Analysis

Performance analysis of an algorithm depends upon two factors i.e. **amount of memory** used and **amount of compute** time consumed on any CPU. Formally they are notified as complexities in terms of:

1. **Space Complexity**
    - Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination.
    - Space need by any algorithm is the sum of following components:

- Fixed Component: This is independent of the characteristics of the inputs and outputs. This part includes: **Instruction Space, Space of simple variables, fixed size component variables, and constants variables**.
- Variable Component: This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like **Linked list, heap, trees, graphs** etc.

Therefore the total space requirement of any algorithm 'A' can be provided as

Space (A) = Fixed Components (A) + Variable Components (A)

- Among both fixed and variable component the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'.
- To identify the space complexity of any algorithm following steps can be followed:
  - Determine the variables which are instantiated by some default values.
  - Determine which instance characteristics should be used to measure the space requirement and this is will be problem specific.
  - Generally the choices are limited to quantities related to the number and magnitudes of the inputs to and outputs from the algorithms.

- Sometimes more complex measures of the interrelationships among the data items can used.

```
Algorithm Sum(number,size)\\ procedure will produce sum of all numbers provided in 'number' list
{
result=0.0;
for count = 1 to size do                \\will repeat from 1,2,3,4,....size times
 result= result + number[count];
return result;
```

- In above example, when calculating the space complexity we will be looking for both fixed and variable components. here we have

1. **Fixed components** as **'result'**, **'count'** and **'size'** variable there for total space required is three (3) words.
2. Variable components is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n'). Because this will decide the size of 'number' list and will also drive the 'for' loop. Therefore if the space used by size is one word then the total space required by 'number' variable will be 'n' (value stored in variable 'size').

Therefore the space complexity can be written as **Space (Sum) = 3 + n;**

2. <u>Time Complexity</u>

- Time Complexity an algorithm (basically when converted to program) is the amount of computer time it needs to run to completion.
- The time taken by a program is the sum of the compile time and the run/execution time.

- The compile time is independent of the instance (problem specific) characteristics. following factors affect the time complexity:

a. Characteristics of compiler used to compile the program.
b. Computer Machine on which the program is executed and physically clocked.
c. Multiuser execution system.
d. Number of program steps.

- Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 & 4), so for any algorithm 'A' it is provided as:

Time (A) = Fixed Time (A) + Instance Time (A)

- Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like
    - comments count as zero steps,
    - an assignment statement which does not involve any calls to other algorithm is counted as one step,
    - For iterative statements we consider the steps count only for the control part of the statement etc.

Therefore to calculate total number program of program steps we use following procedure.

For this we build a table in which we list the total number of steps contributed by each statement.

This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed. This procedure is explained using an example.

Statement	Steps per execution	Frequency	Total Steps
Algorithm Sum(number, size)	0	-	0
{	0	-	0
result=0.0;	1	1	1
for count = 1 to size do	1	size+1	size+1
result= result + number[count];	1	size	size
return result;	1	1	1
}	0	-	0
Total			2size + 3

1.2 Growth of Function – Big 'O', Omega and Theta Notation

Q) Explain the Asymptotic notations **10MKS**

Q) Explain Big – Oh, Omega and Theta notations with the help of example. How do we analyze and measure time and space complexity of algorithms?

- To choose the best algorithm, we need to check efficiency of each algorithm.
- The efficiency can be measured by computing time complexity of each algorithm.
- Asymptotic notation is a shorthand way to represent the time complexity.
- There are three asymptotic notations which are used to give the running time of the algorithm.

1. **Big 'Oh' Notation (O)**
   - O (g(n)) = {f(n): there exists positive constant c and n_0 such that 0 <=f (n) <= cg(n) for all n >= n_0}
   - It is an **upper bound** of any function.
   - Hence, it denotes the **worst case complexity** of any algorithm.
   - We can graphically represents as,

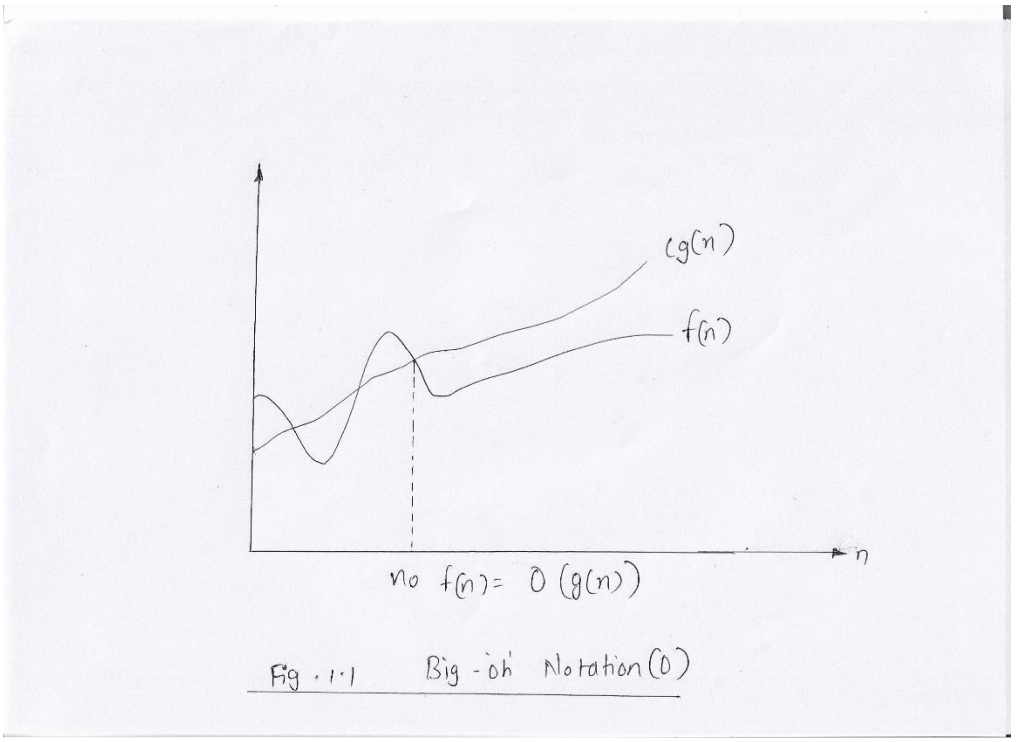


Fig . 1.1 Big - oh Notation (O)

2. Omega Notation (Ω)

- Ω (g (n)) = {f (n): there exists positive constant c and n_0 such that 0 <= cg (n) <=f (n) for all n >= n_0}.
- It is the **lower bound** of any function.
- Hence, it denotes the **best case complexity** of any algorithm.
- We can graphically represents as,

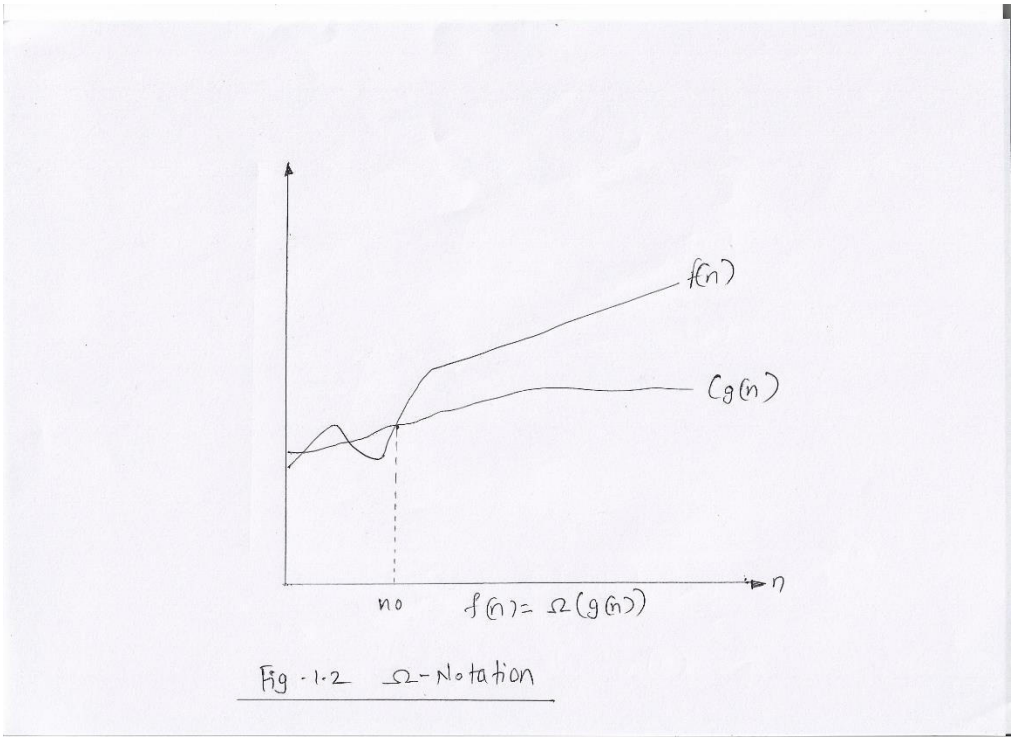


Fig - 1.2 Ω- Notation

3. Theta Notation (Θ)

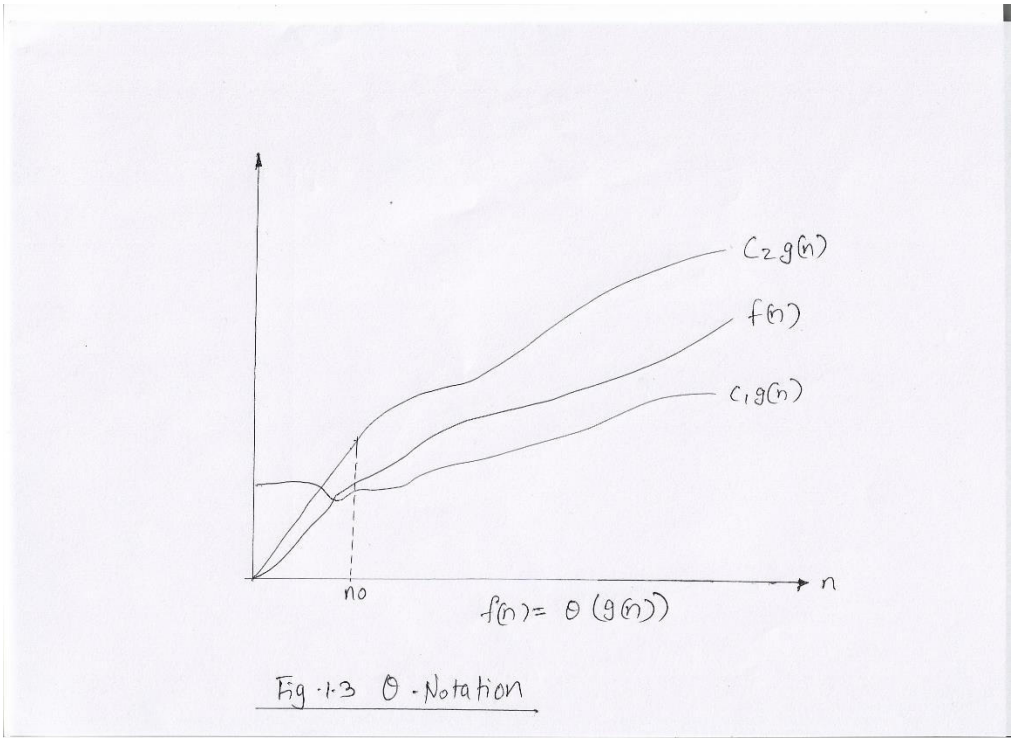


Fig 1.3 Θ - Notation

- Θ (g (n)) = {f (n): there exists positive constant c_1, c_2 and n_0 such that c_1 g (n) <=f (n) <= c_2 g (n) for all n >= n_0}.
- If f (n) = Θ (g (n)), all values of n right to n_0 f (n) lies on or above c_1 g (n) and on or below c_2 g (n).
- Hence, it is asymptotic **tight bound** for f (n).

1.3 Mathematical background for algorithm analysis

- In order to deal with the mathematical aspects of algorithm analysis, we need to be sure we have a clear grasp of some notational conventions, and that we understand a few basic principles and formulas.

Interval Notation

- Interval notation is a convenient means of indicating a certain kind of range of values.
- First let us assume that we have an underlying set of values of some kind (integer, real, even something exotic like C++ STL iterators or Java iterators). The notations shown below assume that a < b (whatever that means in the given context) and should be interpreted to mean, "All values strictly between a and b, and if a square bracket is adjacent to a value, that value is also included, but if a round bracket is next to a value, that value is omitted." That is,
    - [a, b] means all values between a and b, as well as a and b themselves
    - (a, b) means all values between a and b (neither a nor b is included)
    - [a, b) means all values between a and b, with a included, but b not included
    - (a, b] means all values between a and b, with b included, but a not included

Ranges and Mid-Points

- When dealing with algorithms, we are often processing data that is indexed by a range of values, and when doing so we need to be careful not to commit the insidious off-by-one error.
- In other words, we need to be sure at all times exactly how many values are in the range we are working with.

- So, for example, we need to know that the number of values in the closed interval [a, b] with integer end points is b-a+1, while the number of values in the half-open interval [a, b) with integer end points is b-a.
- These seem to be the two most frequently encountered intervals.
- Another operation we perform from time to time is finding the mid-point of an interval with integer end points, and we need to be clear on this procedure.
- Usually we are doing this because we want the "**middle index**" between the first and last values in a range of index values.
- The "middle" value is just the average of the two values, i.e. middle = (first + last)/2, but we need to look closely at what this calculation gives us.
- First, note that only if there is an odd number of values in the range do we have a "true" middle value.
- Fortunately, the calculation middle = (first + last)/2 gives us the correct value of middle in both cases (i.e., whether the number of values is odd or even), as you can easily convince yourself by looking at a few examples:
    - middle = (1+10)/2 = 5
    - middle = (0+9)/2 = 4
    - middle = (1+11)/2 = 6
    - middle = (0+10)/2 = 5

Sum and Product Notation

- The Σ symbol to indicate summations (used frequently)
- The Π symbol to indicate products (used less frequently)

1.4 Analysis of Selection Sort

- Selection Sort works by **repeatedly sorting** elements.
- It works as follows: **first** find the smallest in the array and exchange it with the element in the first position, then find the **second** smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Algorithm: Selection-Sort (A)

fori ← 1 to n-1 do

min j ← i;

min x ← A[i]

for j ← i + 1 to n do

if A[j] < min x then

min j ← j

min x ← A[j]

A[min j] ← A [i]

A[i] ← min x

- Selection sort is among the simplest of sorting techniques and it works very well for **small files**.
- It has a quite important application as each item is actually moved **at the most once**.
- Section sort is a method of choice for sorting files with **very large** objects (records) and small keys.
- The worst case occurs if the array is already sorted in a descending order and we want to sort them in an **ascending order**.
- Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if A[j] < min x is executed exactly the same number of times in every case.
- Selection sort spends most of its time trying to find the minimum element in the unsorted part of the array.
- It clearly shows the similarity between Selection sort and Bubble sort.
    - Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
    - Selection sort is **quadratic** in both the worst and the average case, and requires **no extra memory.**
- For each i from 1 to n - 1, there is one exchange and n - i comparisons, so there is a total of n - 1 exchanges and
- (n − 1) + (n − 2) + ...+ 2 + 1 = n (n − 1)/2 comparisons.
- These observations hold, no matter what the input data is.
- In the worst case, this could be quadratic, but in the average case, this quantity is O (n log n). It implies that the running time of Selection sort is quite insensitive to the input.

Implementation

```
Void Selection-Sort(int numbers[], int array_size) {
int i, j;
int min, temp;
for (i = 0; I < array_size-1; i++) {
min = i;
for (j = i+1; j < array_size; j++)
if (numbers[j] < numbers[min])
min = j;
temp = numbers[i];
numbers[i] = numbers[min];
numbers[min] = temp;
}
}
```

Example

Unsorted list:

5 2 1 4 3

1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

Swap 5 and 1

1 2 5 4 3

2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

No Swap

1 2 5 4 3

3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3

1 2 3 4 5

4th iteration:

Smallest = 4

4 < 5, smallest = 4

No Swap

1 2 3 4 5

Finally,

The sorted list is

1 2 3 4 5

1.5 Analysis of Insertion Sort

- Insertion sort is a very simple method to sort numbers in an ascending or descending order.
- This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.
- The numbers, which are needed to be sorted, are known as keys. Here is the algorithm of the insertion sort method.

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
key = A[j]
i = j – 1
while i > 0 and A[i] > key
A[i + 1] = A[i]
i = i -1
A[i + 1] = key
```

Analysis

- Run time of this algorithm is very much dependent on the given input.
- If the given numbers are sorted, this algorithm runs in O (n) time. If the given numbers are in reverse order, the algorithm runs in O (n2) time.

Example

Unsorted list:

2 13 5 18 14

1st iteration:

Key = a[2] = 13

a[1] = 2 < 13

Swap, no swap

2 13 5 18 14

2nd iteration:

Key = a[3] = 5

a[2] = 13 > 5

Swap 5 and 13

2 5 13 18 14

Next, a[1] = 2 < 13

Swap, no swap

2 5 13 18 14

3rd iteration:

Key = a[4] = 18

a[3] = 13 < 18,

a[2] = 5 < 18,

a[1] = 2 < 18

Swap, no swap

2 5 13 18 14

4th iteration:

Key = a[5] = 14

a[4] = 18 > 14

Swap 18 and 14

2 5 13 14 18

Next, a[3] = 13 < 14,

a[2] = 5 < 14,

a[1] = 2 < 14

So, no swap

2 5 13 14 18

Finally,

the sorted list is

2 5 13 14 18

1.6 Recurrences:

- Many algorithms are **recursive** in nature.
- When we analyze them, we get a **recurrence relation** for time complexity.
- We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes.
- For example in Merge Sort, to sort a given array, we divide it in **two halves** and recursively repeat the process for the two halves.
- Finally we merge the results. Time complexity of Merge Sort can be written as **T (n) = 2T (n/2) + cn**.
- There are many other algorithms like **Binary Search, Tower of Hanoi**, etc.

1) Substitution Method:

- We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction

to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true

for values smaller than n.

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

<= cn/2Log(n/2) + n

= cnLogn - cnLog2 + n

= cnLogn - cn + n

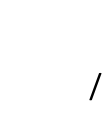
<= cnLogn

2) <u>Recurrence Tree Method:</u>

- In this method, we draw a recurrence tree and calculate the time taken by every **level of tree**.
- Finally, we **sum the work** done at all levels.
- To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels.
- The pattern is typically an **arithmetic** or **geometric** series.

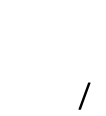
For example consider the recurrence relation

T(n) = T(n/4) + T(n/2) + cn2

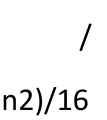


If we further break down the expression T(n/4) and T(n/2),

we get following recursion tree.



Breaking down further gives us following



VIVA QUESTIONS

Q) What is the time complexity of binary search?

Binary search is a logarithmic algorithm and executes in O (log N) time.

Q) What is OBST?

Optimal **B**inary **S**earch **T**ree is used for fast searching a very good example is google search engine.

Q) Advantages of divide and conquer

1. Solving difficult problem

2. Algorithm efficiency

3. Parallelism

4. Memory access

5. Round - off control

Q) What is space complexity?

It is defined as amount of space and memory required to solve problem

Q) What is O-notation (upper bound)?

- This notation gives an upper bound for a function to within a constant factor.
- We write f (n) =O (g (n)) if there are positive constant n0, the value of f (n) always lies on or below cg (n).

Q) How to derive the complexity of minimum and maximum method using master method?

T (n) =2T (n/2) + C

By master method,

a=2; b=2; k=0;

$b^k = 1$;

a>b;

Therefore T (n) = O (n)

Since if a>b, T (n) =O (n)

Q) Define Big-Oh, Omega and Theta Notations

Theta Notation (Same order):

- This notation bounds a function to within constant factors.
- We say (f (n) = theta (big(g(n)big)) if there exist positive constants n0, c1, c2 such that to the right of n0 the value of f(n) always lies between c1g(n) and c2g(n) inclusive.

Big - Oh Notation (Upper Bound):

- This notation gives an upper bound for a function to within a constant factor.
- We say (f(n) = O(big(g(n)big)) if there are positive constants n0 and c such that to the right of n0 , the value of f(n) lies on or below cg(n).

Omega Notation:

- This notation gives a lower bound for a function to within a constant factor.
- We say (f(n)=Omega(big(g(n)big)) if there are positive constants n0 and c such that to the right of n0, the value of f(n) always lies on or above cg(n).

Q) Define an algorithm

• An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into the output.

Q) Define the properties of algorithms

Properties of Algorithms

Input: an algorithm accepts zero or more inputs

Output: it produces at least one output.

Finiteness: If we trace out the instructions of an algorithm, then for all cases, it terminates after a finite number of steps.

Definiteness: Each step in algorithm is unambiguous. This means that the action specified by the step cannot be in multiple ways & can be performed without any confusion.

Effectiveness: It consists of basic instructions that are realizable.

The operations are do-able.

Q) What is Asymptotic Notations?

- Asymptotic notations are mathematical tools use to represent complexity of algorithm for asymptotic analysis.
- Three Asymptotic Notations are
    - Big- O- Notation.
    - Theta notation.
    - Omega Notation.

Q) What is Time and Space Complexity?

- Time Complexity is defined as the number of steps required to solve the entire problem using some efficient algorithm.

- Space Complexity of a problem is defined as the amount of space and memory required by an algorithm to solve the problem.
- It is measured in terms of Big-O- Notations.

Q) What is Randomized Algorithm?

- A randomized algorithm are also called as probabilistic Algorithms
- A randomized Algorithm is an algorithm which employs a degree of randomness as part of its logic.
- Randomized Algorithms are approximated using a pseudo random number generator in place of true source of random bits.

Q) What are different approached for designing an algorithm?

Greedy algorithm

Divide and conquer

Dynamic programming

Backtracking

Q) Explain divide and conquer.

- In divide and conquer, the given problem is divided into smaller sub problems.
- This sub problems are solved independently.
- Then combining all the solutions of sub problems into a solution of the whole.
- If the problem are large enough then divide and conquer is reapplied.

Q) Explain importance of sorting?

- Sorting is supposed to be the basic algorithm on which many algorithms can be generated.
- By using the sorting techniques we can solve a lot of problems.