Module 1

Introduction to Algorithms

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

Properties of a good Algorithm

1. Input. Zero or more quantities are externally supplied.

2. Output. At least one quantity is produced.

3. Definiteness. Each instruction is clear and unambiguous.

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

5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion3; it also must be feasible.

Development of an Algorithm

The development of an algorithm is a key step in solving a problem. Once we have an algorithm, we can translate it into a computer program in some programming language. Algorithm development process consists of five major steps.

Step 1: Obtain a description of the problem.

Step 2: Analyse the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

Pseudocode Conventions

1. Comments begin with // and continue until the end of line

2. Blocks are indicated with matching braces: { and }

3. Statements are delimited by ;

4. An identifier begins with a letter. The data types of variables are not explicitly declared

5.Assignment of values to variables is done using the assignment statement
eg:(variable)=(expression

6. the logical operators **and, or** and **not** and the **relational operators**

$<, \leq, =, \neq, \geq,$ and $>$ are used.

7. Elements of arrays are accessed using[and].

8. The following looping statements are employed:
for ,while and **repeat- until**

repeat

(statement 1)

(statement n)

Until (condition)

9. A conditional statement has the following forms

If,

if else,

if elseif else

10. Input and output are done using the instructions **read** and **write**

11. There is only one type of procedure: **Algorithm**

An algorithm consists of a heading and a body. The heading takes the form
Algorithm Name (parameter list)

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4      Result := A[1];
5      for i := 2 to n do
6          if A[i] > Result then Result := A[i];
7      return Result;
8  }
```

Recursive Algorithms

A recursive function is a function that is defined in terms of itself. Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.

An algorithm that calls itself is ***direct recursive***. Algorithm A is said to be ***indirect recursive*** if it calls another algorithm which in turn calls A.

To build a recursive algorithm, break the given problem statement into two parts. The first one is the base case, and the second one is the recursive step.

Base Case: It is nothing more than the simplest instance of a problem, consisting of a condition that terminates the recursive function. This base case evaluates the result when a given condition is met.

Recursive Step: It computes the result by making recursive calls to the same function, but with the inputs decreased in size or complexity.

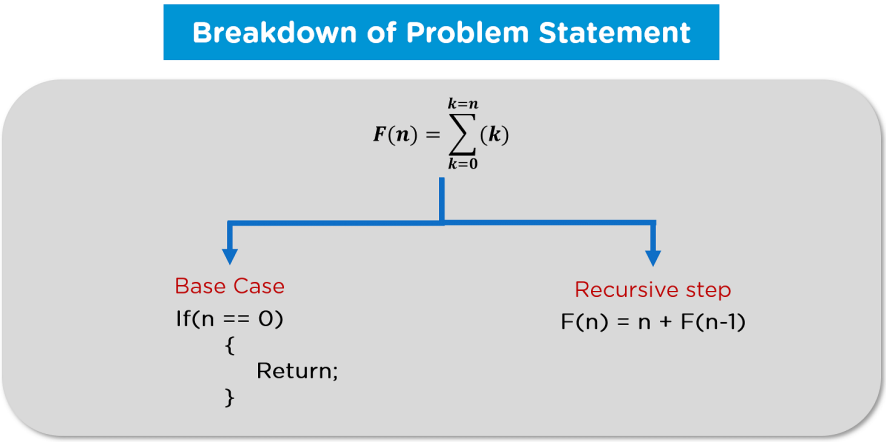
For example, consider this problem statement: Print sum of n natural numbers using recursion. This statement clarifies that we need to formulate a function that will calculate the summation of all natural numbers in the range 1 to n. Hence, mathematically you can represent the function as:

F(n) = 1 + 2 + 3 + 4 + ……..+ (n-2) + (n-1) + n

It can further be simplified as:

$$F(n) = \sum_{k=1}^{k=n} (k)$$

You can breakdown this function into two parts as follows:



Performance Analysis

Space and Time Complexity: The **space complexity** of an algorithm is the amount of memory it needs to run to completion.

The **time complexity** of an algorithm is the amount of computer time it needs to run to completion.

Running Time Comparison-Worst, Best and Average Case

Best Case Efficiency - is the minimum number of steps that an algorithm can take any collection of data values. Smaller Comparisons. In Big O(1) is considered as best case efficiency.

Average Case Efficiency - average comparisons between minimum no. of comparisons and maximum no. of comparisons.

Minimum number of comparisons = 1

Maximum number of comparisons = n

Therefore, average number of comparisons = (n + 1)/2

(n + 1)/2 is a linear function of n. Therefore, the average case efficiency will be expressed as O(n).

Worst Case Efficiency - is the maximum number of steps that an algorithm can take for any collection of data values. Maximum number of comparisons

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.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

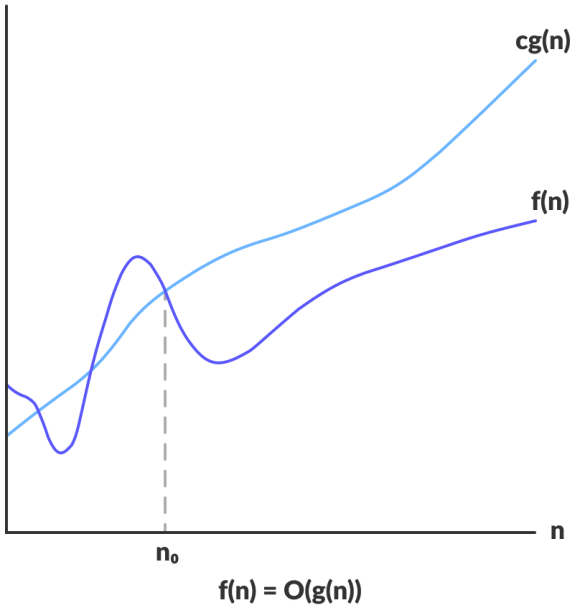
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Big-O gives the upper bound of a function

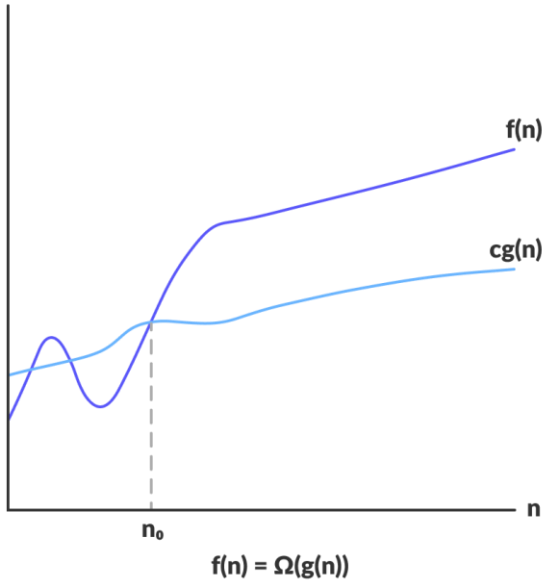
```
O(g(n)) = { f(n): there exist positive constants c and n₀
            such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n₀ }
```

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.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



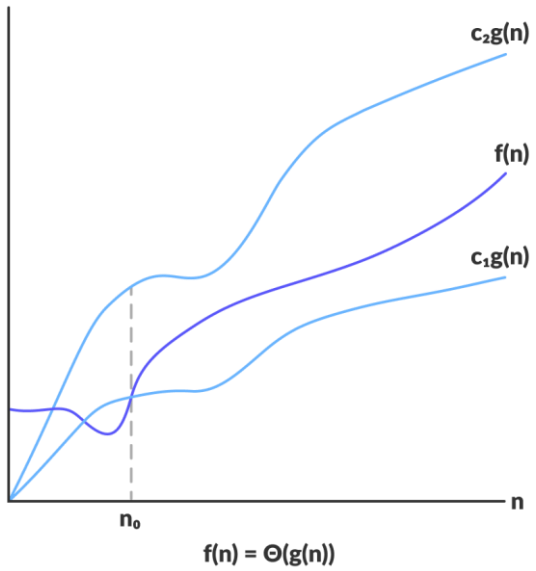
Omega gives the lower bound of a function

```
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))`.

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function `g(n)`, `Θ(g(n))` is given by the relation:

```
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.

(More in pdf given by Anoop sir)

Recurrence Relations

Solving Recurrences using substitution

In the substitution method for solving recurrences :

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works.

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
    <= 2cn/2Log(n/2) + n
    = cnLogn - cnLog2 + n
    = cnLogn - cn + n
    <= cnLogn

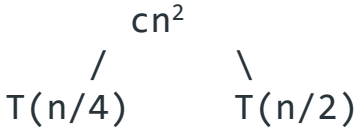
(More examples : check recursive solutions in pdf given by Anoop sir)

Solving Recurrences using recurrence trees

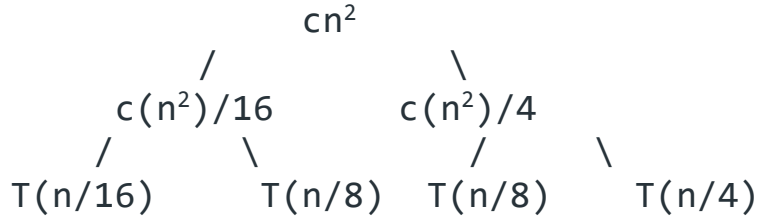
A recursion tree is a tree where each node represents the cost of a certain recursive subproblem. Then you can sum up the numbers in each node to get the cost of the entire algorithm.

For example, consider the recurrence relation

$T(n) = T(n/4) + T(n/2) + cn^2$



If we further break down the expression T(n/4) and T(n/2), we get the following recursion tree.



```
Breaking down further gives us following
                  cn^2
             /            \
       c(n^2)/16          c(n^2)/4
       /      \             /      \
c(n^2)/256   c(n^2)/64  c(n^2)/64    c(n^2)/16
 /    \       /    \     /    \       /    \

To know the value of T(n), we need to calculate the sum of tree
nodes level by level. If we sum the above tree level by level,
we get the following series
T(n)  = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ....
The above series is a geometrical progression with a ratio of 5/16.

To get an upper bound, we can sum the infinite series.
We get the sum as (n^2)/(1 - 5/16) which is O(n^2)
```

Amortized Analysis

Amortized analysis is a worst-case analysis of a sequence of operations — to obtain a tighter bound on the overall or average cost per operation in the sequence than is obtained by separately analyzing each operation in the sequence.

There are generally three methods for performing amortized analysis:

the aggregate method, the accounting method, and the potential method.

The aggregate method : where the total running time for a sequence of operations is analyzed.

The accounting (or banker's) method : where we impose an extra charge on inexpensive operations and use it to pay for expensive operations later on.

The potential (or physicist's) method : in which we derive a potential function characterizing the amount of extra work we can do in each step. This potential either increases or decreases with each successive operation, but cannot be negative.

More details: [Amortized Analysis]