Chapter 1 - Introduction

Algorithm Specification

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. There are zero or more quantities that 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 basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

Recursive Algorithms

Recursion is a technique whereby functions can call themselves. Recursive functions can call themselves either *directly* or *indirectly*:

> **Direct** - *FunctionA* calls *FunctionA*
> **Indirect** - *FunctionA* calls *FunctionB*, *FunctionB* calls *FunctionA* or *FunctionA* calls *FunctionB*, *FunctionB* calls *FunctionC*, *FunctionC* calls *FunctionA*.

Example *[Binary search]:*

To transform an iterative function into a recursive one, we must

1. Establish boundary condi-tions that terminate the recursive calls, and
2. Implement the recursive calls so that each call brings one step closer to a solution.

There are two ways to terminate the search: one signaling a success (list[middle] = searchnum), the other signaling a failure (the left and right indices cross i.e., left > right). Creating recursive calls that move closer to a solution is also simple since it requires only passing the new left or right index as a parameter in the next recursive call. Following algorithm implements the recursive binary search.

```
int compare(int x, int y) {

/* compare x and y, return -1 for less than, 0 for equal, 1 for greater */

if (x < y) return -1;

else if (x == y) return 0;

else return 1;

)

int binsearch(int list[], int searchnum, int left, int right)

{

/* search list[[0] <= list[l] <= • • • <= list[n-l] for searchnum. Return its position if found,
Otherwise return -1 */

int middle;

if (left = right) {

middle = (left + right)/2;

switch (COMPARE(list[middle], searchnum)) {

case -1: return binsearch(list, searchnum, middle+1, right);

case 0: return middle;

case 1 : return binsearch(list, searchnum, left, middle-1);

}

}

return -1;

}
```

Data Abstraction

Data abstraction is one of the most essential and important feature of object oriented programming. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation, i.e., to represent the needed information in a program without presenting the details.

Data Type

Data Type A data type is a collection of objects and a set of operations that act on those objects.

Abstract Data Type

An abstract data type(ADT) is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

The specification of operations consists of the names of every function, the type of its arguments, and the type of its result. Also, description of what the function does, but without appealing to internal representation or implementation details. An abstract data type is implementation-independent.

ADT operations can be classified into following categories:

- Creator/constructor: These functions create a new instance of the designated type.
- Transformers: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between construc-tors and transformers will become more clear with some examples.
- Observers/reporters: These functions provide information about an instance of the type, but they do not change the instance.

Performance Analysis:

Complexity:

- Complexity refers to the rate at which the storage,time grows as a function of the problem size
- The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.
- Complexity of an algorithm is analyzed in two perspectives: Time and Space. Time Complexity - It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take. Space Complexity - It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this. Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Space Complexity:

Space Complexity Example:

```
Algorithm abc(a,b,c)
{
        return a+b++*c+(a+b-c)/(a+b) +4.0;
}
```

The Space needed by each of these algorithms is seen to be the sum of the following components:

- A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.
    - The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.
- A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space

needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

The space requirement s(p) of any algorithm p may therefore be written as, S(P) = c+ Sp(Instance characteristics), Where 'c' is a constant.

Example 2:

```
Algorithm sum(a,n) {
s=0.0;
for I=1 to n do
        s= s+a[I];
return s;
}
```

- The problem instances for this algorithm are characterized by n, the number of elements to be summed. The space needed by 'n' is one word, since it is of type integer.
- The space needed by 'a' is the space needed by variables of type array of floating point numbers.
- This is atleast 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So,we obtain Ssum(n)>=(n+s) [n for a[], one each for n,I, a & s]

Time Complexity:

- The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)
- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).
- The number of steps any problem statement is assigned depends on the kind of statement. For example,
- comments 0 steps.
- Assignment statements 1 steps. [Which does not involve any calls to other algorithms]
- Interactive statement such as for, while & repeat-until - Control part of the statement.

The method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Asymptotic notation

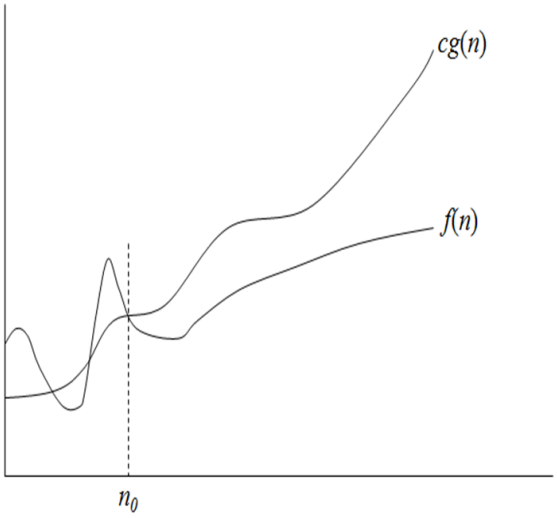
- Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω).
- An algorithm with asymptotically low running time (for example, one that is $O(n^2)$ is beaten in the long run by an algorithm with an asymptotically faster running time (for example, one that is O(n log n)), even if the constant factor for the faster algorithm is worse. It describes how the running time of an algorithm increases with the size of the input in the limit.
- Asymptotically more efficient algorithms are best for all but small inputs

Big O Notation

Definition: $f(n) = O(g(n))$ if there are positive constants n_0 and c such that $|f(n)| \leq c\,|g(n)|$ *for all* $n \geq n_0$

As n increases, f(n) grows no faster than g(n).

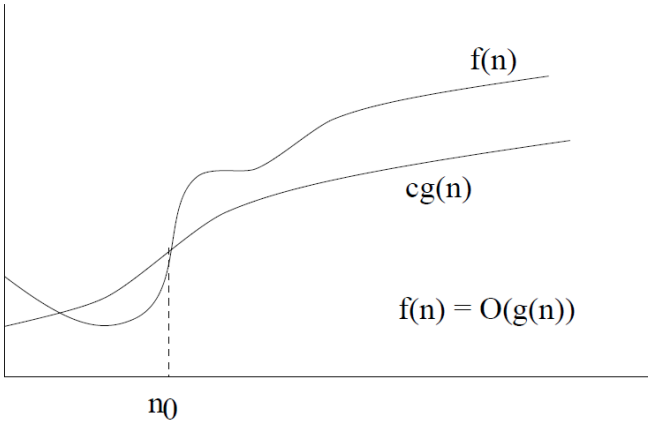
In other words, g(n) is an asymptotic upper bound on f(n).



Big Ω- notation

Definition: f(n) = Ω(g(n)) iff there are two positive constants c and n_0 such that $|f(n)| \geq c\,|g(n)|$ *for all* $n \geq n_0$

As n increases, f(n) grows faster than g(n). In other words, g(n) is an asymptotic lower bound on f(n).

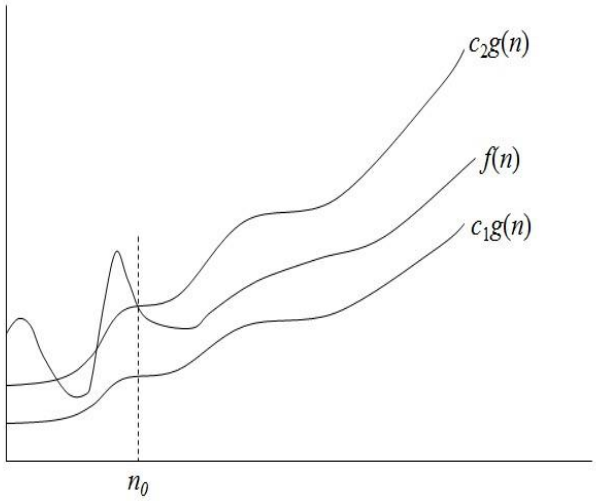


Big Θ – notation

Definition : f(n) = Θ (g(n)) iff there are three positive constants c_1, c_2 and n_0 such that:

$c_1\,|g(n)| \leq |\,f(n)\,| \leq c_2\,|g(n)|$ *for all* $n \geq n_0$

As n increases, f(n) grows at the same rate as g(n). In other words, g(n) is an asymptotically tight bound on f(n).


$c_2g(n)$
$f(n)$
$c_1g(n)$
n_0