**Recurrence relation types:**

Recurrences are classified by the way in which terms are combined, the nature of the coefficients involved, and the number and nature of previous terms used.

**First-Order Recurrences**

**First order Recurrence relation :-**A recurrence relation of the form : **an = can-1 + f(n)** for n>=1  
where c is a constant and f(n) is a known function is called linear recurrence relation of first order with constant coefficient. If f(n) = 0, the relation is homogeneous otherwise non-homogeneous.

**Example :-**xn = 2xn-1 – 1, an = nan-1 + 1, etc.

**Second order linear homogeneous Recurrence relation :-**A recurrence relation of the form  
**cnan + cn-1an-1 + cn-2an-2 = 0** ——> (1)  
for n>=2 where cn, cn-1 and cn-2 are real constants with cn != 0 is called a second order linear homogeneous recurrence relation with constant coefficients.

**Divide and conquer recurrence relations**

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

T(n) = 2T(n/2) + √n

**Performance Analysis of Algorithm**:  
Algorithm analysis is the determining of algorithm efficiency and then  
enhancing it. There is two criterions directly relates to algorithm performance  
that are:  
**Space Complexity**:  
The space complexity of a program is the amount of memory it needs to  
run to completion. The space needed by each program is seen to be the sum  
of the following components:  
• A Fixed Part that is independent of the characteristics (e.g., number,  
size) of the inputs and outputs. This part typically includes the instruction  
space (i.e., space for the code), space for simple variables and fixed  
size component variables (also called aggregate), space for constant,  
etc.  
• A Variable Part that consists of the space needed by component  
variables whose size is dependent on the particular problem instance  
being solved, space needed by referenced variables (to the extent that  
this depends on instance characteristics), and the recursion stack space  
( in so far as this space depends on the instance characteristics).  
The space requirement S(p) of any program p may therefore be written  
as:  
S(p) = c + Sp ( Instance Characteristics) where c is constant.  
**Time Complexity**:  
The time complexity of a program is the amount of computer time it  
needs to run to completion. The time, T(p), taken by a program p, is the sum  
of the compile time and the run (or 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.  
Algorithm Design And Analysis Asst. Prof. Ali Kadhum Idrees  
Department of Computer Science College 4 of Science for Women  
Consequently, we shall concern ourselves with just the run time of the  
program. The run time is defined by tp (Instance Characteristics). The time  
requirement T(p) of any program p may therefore be written as:  
T(p) = c + tp (Instance Characteristics) where c is the compilation time.  
Two manageable approaches to estimating run time are (1) identify one  
or more key operations and determine the number of times these are  
performed and (2) determine the total number of steps executed by the  
program.

Different methods of measuring running time of algorithm

**Step count method:**

Suppose we have one algorithm to perform sequential search. Suppose each instruction will take c1, c2, …. amount of time to execute, then we will try to find out the time complexity of this algorithm

| **Algorithm** | **Number of times** | **Cost** |
| --- | --- | --- |
| seqSearch(arr, n, key) i := 0 while i < n, do    if arr[i] = key, then       break    end if done return i | 1 n+1 n 0/1    1 | c1 c2 c3 c4    c5 |

Now if we add the cost by multiplying the number of times it is executed, (considering the worst case situation), we will get

*Cost=c1+(n+1)c 2+nc3+c 4+c 5*

*Cost=c1+nc 2+c2+nc 3+c 4+c5*

*Cost=n(c 2+c3)+c 1+c 4+c5*

*Cost=n(c 2+c3)+C*

Considering the c1 + c4 + c5 is C, so the final equation is like straight line y = mx + b. So we can say that the function is linear. The complexity will be O(n).

**Operations Count Method:**  
One way to estimate the time complexity of a program or function is to  
select one or more operations, such as add, multiply, and compare, and to  
determine how many of each is done. The success of this method depends on  
our ability to identify the operations that contribute most to the time complexity.  
Several examples of this method as follow.  
Example (1): Write a function that return the position of the largest element in  
the array a[1..n], and then estimate its time complexity using operations count  
method.  
**Example 1:**

Solution:  
Function Max(var a:ElemList; n:integer): integer;  
Var pos:integer;  
Begin  
Pos:=1;  
For i:=2 to n do  
If ( a[pos] < a[i] ) then  
Pos:= i;  
Max:=pos;  
End  
We can estimate its time complexity by the number of comparisons  
made between elements of the array a. each iteration of the for loop makes  
one such comparisons, so the total number of element comparisons **is n-1.**

**Example 2:**

**Insertion into a Sorted Array:**

gives an algorithm to insert an element x into a sorted array a[0:n-1].

We wish to determine the number of comparisons made between x and the elements of a. For the problem size, we use the number n of elements initially in a. Assume that *n ≥*1.

The best or minimum number of comparisons is 1, which happens when the new element x

void insert(int [] a, int n, int x)

{

// find proper place for x

int i;

for (i = n – 1; i >= 0 && x < a[i]; i–)

a[i+1] = a[i];

a[i+1] = x; // insert x

}

is to be inserted at the right end. The maximum number of comparisons is n, which happens when x is to be inserted at the left end. For the average assume that x has an equal chance of being inserted into any of the possible n+1 positions. If x is eventually inserted into position i+1 of a, i *≥*0, then the number of comparisons is n-i. If x is inserted into a[0], the number of comparisons is n. So the average count is

**The recurrence relation of the code of recursive insertion sort is T(n) = T(n-1) + n. It can be solved by the method of substitution and is found to be equal to n2T**