Design and Analysis of Algorithms

UNIT-I - INTRODUCTION

Introduction, Algorithm, Pseudo code for expressing algorithms, Performance Analysis – Space Complexity, Time Complexity, Asymptotic Notation - Big Oh Notation, Omega notation, Theta Notation and Little oh notation, Probabilistic analysis, amortized analysis, Performance Measurement, Randomized Algorithms.

---------- * ---------- * ---------- * ----------

What is an Algorithm: An algorithm is any well defined computational procedure that takes some values as input and produces some values as output. It is thus a sequence of computational steps that transform input into output.

An algorithm is composed of finite steps each of which may require one or more operations. Each operation may be characterized as either simple or complex.
An algorithm is a step by step procedure for performing some tasks in a finite amount of time.
According to **D.E.Knuth** a pioneer in the computer science discipline an algorithm must have the following 5 basic properties

1. Input
2. Output
3. Finiteness
4. Definiteness
5. Effectiveness

Input: An algorithm has 0 or more inputs that are given to it initially before it begins (or) dynamically as it runs.

Outputs: An algorithm has 1 or more outputs that have a specified relation with the inputs. An algorithm produces at least one or more outputs.

Finiteness: An algorithm must always terminate after a finite number of steps.

Definiteness: Each operation specified in an algorithm must have definite meaning. Each step of the algorithm must be precisely defined. Instructions such as "compute x/0 " or "subtract 7 or 6 to x" are not permitted because it is not clear which of the two possibilities should be done or what the result is.

Effectiveness: Each operation of an algorithm should be effective. i.e. The operation must be able to be carried out in a finite amount of time. Tracing of each step should be possible.

Algorithm Vs Program: In computational theory, we distinguish between an algorithm and a program. Program does not have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs always terminate, we use "algorithm" and "program" interchangeably.

Pseudo Code Conventions

Algorithm specification: Pseudo code conventions:

1) Comments begin with // and continue until the end of line.
2) Blocks are indicated with matching braces: { }. A compound statement can be represented as a block.
3) All statements are separated or terminated by a semicolon(;).
4) An identifier begins with a letter followed by letters or digits or underscore.
5) Assignment of values to the variables is done using the assignment operator :=
6) There are two Boolean values true and false
7) Elements of single dimensional and multidimensional arrays.
8) Loops : while, do-while, repeat until, for

Ex1)
```
while (condition)
{
  statement1;
  statement2;
  -------
  -------
}
```

Ex2)
```
do
{
  statement1;
  statement2;
  -------
  -------
} while (condition);
```

Ex3)
```
Repeat
{
  -------
  --------
}until(condition);
```

Ex4)
```
for var:=valu1 to value2 do
{
  -------
  --------
}
```

9) Conditional statements : if then else are written in the following form

Ex1)
```
if (condition) then
        statement;
```

Ex2)
```
if (condition) then
        statement1
else
        statement2
```

10) Case statement

```
case
{
  condition1: statement1
  condition2: statement2
  --------
}
```

11) Input & output
12) Structure: algorithm is a procedure consisting of a heading and body.

```
Algorithm name_of_alg(parameters)
{
      ------
      ------
}
```

Ex1) Algorithm to find sum of elements of a single dimensional array

```
Algorithm addition(a,n)
// a is an array of size n
{
  sum:=0;
  for i:= 1 to n do
     sum:=sum+a[i];
  return sum;
}
```

Ex 2) Algorithm to find largest number from the elements of a single dimensional array

```
Algorithm find_large(a,n)
// a is an array of size n
{
  large:=a[1];
  for i:= 2 to n do
     if (a[i]>large) then large:=a[i];
  return large;
}
```

Ex 3) Algorithm to find smallest number from the elements of a single dimensional array

```
Algorithm find_small(a,n)
// a is an array of size n
{
  small := a[1];
  for i:= 2 to n do
     if (a[i]< small) then small := a[i];
  return small;
}
```

Ex 4) Algorithm to find Factorial of a given integer number

```
Algorithm factorial(n)
{
  fact:=1;
   for i:= 1 to n do
     fact:= fact*i;
  return fact;
}
```

Ex 5) Algorithm to find ncr

```
Algorithm find_ncr(n,r)
{
   fn:=1;
   for i:= 1 to n do
     fn:= fn*i;
   fr:=1;
   for i:= 1 to r do
     fr:= fr*i;
   k := n-r;
   fk:=1;
   for i:= 1 to k do
     fk:= fk*i;
   ncr:= fn / ( fr * fk);
  return ncr;
}
```

Ex 6) Algorithm to search for a target element using linear search method

```
Algorithm linear_search(a,n,tar)
// a is an array of size n
{
      pos:=0;
      for i:= 1 to n do
         if (a[i]=tar) then pos:=i;
      return pos;
}
```

Ex 7) Algorithm to search for a target element using Binary search method

```
Algorithm binary_search(a,n,tar)
// a is an array of size n
{
      low := 1;
      high := n;
      while ( low<=high)
      {
           mid:=(low+high)/2;
           if (tar = a[mid]) then pos:=mid;
           else
           if (tar>a[mid]) then low:= mid+1;
           else high := mid – 1;
       }
      return pos;
}
```

Ex 8) Algorithm to sort elements of a single dimensional array

```
Algorithm buble_sort(a,n)
// a is an array of size n
{
     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;
             }}
```

Ex 9) Algorithm to sort elements of a single dimensional array using selection sort technique.

```
Algorithm selection_sort(a,n)
// a is an array of size n
{
     for I := n downto 2 do
       {
            k:= 1;
            for j:= 2 to i do
               if (a[j]>a[k]) then k:=j;
            temp:=a[i];
            a[i]:= a[k];
            a[k]:=temp;
        }
}
```

RECURSION

Ex 10) Recursive algorithm to add elements of an array.

```
Algorithm add(a,n)
// a is an array of size n
{
    if (n=0) return 0;
    else
    return a[n]+add(a,n-1);
}
```

Ex 11) Recursive algorithm to find factorial of a given number.

```
Algorithm fact(n)
{
   if (n<=1) then return 1;
   else
   return n*fact(n-1);
}
```

Ex 12) Recursive algorithm to find ncr

```
Algorithm ncr(n,r)
{
   if (r=0) then return 1;
   else
     if (r=1) then return n;
     else
         if (r=n) then return 1;
         else
            return ( ncr(n-1,r)+ncr(n-1,r-1));

}
```

Ex 13) Algorithm to find nth Fibonacci number.

```
Algorthm fib(n)
{
   if (n=1) then return 0;
   else
      if (n=2) then return 1;
       else
          return (fib(n-1)+fib(n-2));
}
```

Ex 14) Algorithm to solve Towers of Hanoi problem.

There are three towers named A,B and C. N disks were stacked on one tower(A) in decreasing order of size from bottom to top. The problem is to move disks from tower A to tower b using tower C as intermediate storage. Only one disk can be moved at a time. At any time bigger disk cannot be placed on a smaller one.

```
Algorithm towersofhanoi(n,x,y,z)
// move the top n disks form tower x to tower y
{
    if (n>=1) then
      {
          towersofhanoi(n-1,x,z,y);
          write("move disk ", n , "from tower", x , "to tower",y);
          towersofhanoi(n-1,z,y,x);
      }
}
```

PROCESS FOR DESIGN AND ANALYSIS OF ALGORITHMS

1) Understand the Problem

The first thing you need to do before designing an algorithm is to understand problem completely.

2) Decide on Exact Vs Approximate Solving

Solve the problem exactly if possible, otherwise use approximation methods. Even though some problems are solvable by exact method, but they are not faster when compared to approximation method. So in that situation, we will use approximation method.

3) **Algorithm design techniques**. For a given problem, there are many ways to design algorithms for it.

i) Divide & Conquer (D&C)
ii) Greedy Method
iii) Dynamic programming
iv) Backtracking
v) Branch and bound
vi) Brute-force
vii) Decrease and conquer
viii) Transform and conquer
ix) Space and time Trade offs

Depending upon the problem, we will use suitable design method.

4) Prove Correctness

Once an algorithm has been specified, next we have to prove its correctness.
Usually testing is used for proving correctness.

5) Analyse Algorithm

Analysing the algorithm means studying the algorithm behavior, i.e. calculating the time complexity and space complexity. If the time complexity is more, then we will use one more designing technique such that time complexity should be minimum.

6)Coding an Algorithm

After completion of all phases successfully, then we will code an algorithm. Coding should not depend on any programming language. We use general notation (pseudo code) and English language statements.

Testing of Algorithms

Testing is a stage of implementation which is aimed at ensuring that the program works accurately and efficiently before the live operations starts.

Testing a program consists of two phases, debugging and profiling (performance measurement).

Debugging is the process of eliminating errors.

Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute results.

PERFORMANCE ANALYSIS

1. Space Complexity
2. Time Complexity

Algorithm analysis refers to the task of determining the computing time and storage requirements of an algorithm. It is also known as performance analysis which enables us to select an efficient algorithm.

We can analyze an algorithm by two ways.

1)By checking the correctness of an algorithm.

2)By measuring the time and space complexity of an algorithm.

To Compute the analysis of algorithm, two phases are required. They are

1)Priori analysis

2)Posteriori analysis

1) Priori analysis: This is one of the creative analysis of algorithms. In this we obtain a function which bounds the algorithms computing time.

Suppose there is a statement in the middle of a program. We wish to determine the total time that the statement will spend for the execution. This requires

i) The statements frequency count i.e. the number of times the statement will be executed

ii)The time taken for one execution

The product of these two numbers is the total time.

The priori analysis is mainly concerned with the order of determining the magnitude.

The notation used in priori analysis are Big-oh(O), Omega(Ω), theta(Θ) and small-oh(o). Priori analysis of computing time ignores all of the factors, which are machine or programming language dependent and only concentrates on determining the order of the magnitude of the frequency of execution of the statements.

2) Posteriori Analysis: in this we will collect the actual statistics about the algorithm, in conjunction of the time and space while executing. Once the algorithm is written it has to be tested. Testing a program consists of two major phases.

i) Debugging : It is the process of executing programs on sample data sets that determine whether we get proper results. If faulty results occurs it has to be corrected.

ii) Profiling : it is the process of executing a correct program on a actual data sets and measuring the time and space it takes to compute the results during execution. The actual time taken by the algorithm to process the data is called profiling.

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

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

Space Complexity: The space needed by the algorithms is seen to be the sum of the following components:

1) A fixed part that is independent of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables, space for constants etc..

2) 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 reference variables 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).

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

In this algorithm the problem instance is characterized by the specific values of a,b and c. making the assumption that one word is adequate to store the values of each of a,b and c, and the result, we see that space needed by abc is independent of the instance characteristics,
S_p (instance characteristics)=0
Space requirement s[p] = 3+0 =0
One space for each a,b,c
Space complexity is O(1)

```
Algorithm addition(a,n)
// a is an array of size n
{
  sum:=0.0; ------------------------------------------------- 1
  for i:= 1 to n do                                 ---------1              1
     sum:=sum+a[i]; ---------------------------------------n
  return sum;
}
```

Space needed by n is one word; space needed by a is n words; space needed by i and sum one word each.
S(addition)=3+n
Space complexity O(n)

```
Algorithm add(a,n)
// a is an array of size n
{
    if (n=0) return 0.0;
    else
    return a[n]+add(a,n-1);
}
```

The instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to add requires at least three words(n, return address, pointer to a[]). Since the depth of recursion is n+1, the recursion stack space needed is >=(3(n+1).
Recursion –stack space = 3(n+1) = 3n + 3 = O(n)

Time Complexity: The time taken by a program P is the sum of the compile time and the run time. The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will run several times without recompilation. Consequently we concern ourselves with just the run time of a program. This runtime is denoted by tp(instance characteristics).

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example : Let us consider the statement in the following algorithm.

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

The entire statement could be regarded as a step since its execution time is independent of the instance characteristics. The number of steps any program statement is assigned depends on the kind of statement. For example comments count as 0 steps. Assignment statement which does not involve any calls to other algorithms is counted as one step. In an iterative statement such as the for, while and repeat-until statements, we consider the step counts only for the control part of the statement.

Statement	s/e	frequency	Total Steps
Algorithm addition(a,n)	0	-	0
// a is an array of size n	0	-	0
{	0	-	0
sum:=0.0;	1	1	1
for i:= 1 to n do	1	n+1	n+1
sum:=sum+a[i];	1	n	n
return sum;	1	1	1
}	0	-	0
Total			2n+3

The number of steps per execution and the frequency of each of the statements in addition algorithm have been listed. The total number of steps required by the algorithm is determined to be 2n+3. It is important to note that the frequency of for statement is n+1 and not n. this is so because i has to be incremented to n+1 before the for loop can terminate.

Statement	s/e	Frequency		Total Steps	
		n=0	n>0	n=0	n>0
Algorithm add(a,n)	0	-	-	0	0
// a is an array of size n	0	-	-	0	0
{	0	-	-	0	0
if (n=0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else	0	-	-	0	0
return a[n]+add(a,n-1);	1+x	0	1	0	1+x
}	0	-	-	0	0
Total				2	2+x

$x=t_{add(a,n-1)}$

Notice that under the s/e column, the else clause has been given a count of $1 + t_{add(a,n-1)}$ it includes all steps that get executed as a result of the invocation of add() from the else clause. The frequeny and total steps columns have been split into two parts: one for the case n=0 and the other for the case n>0.

Statement	s/e	frequency	Total Steps
Algorithm addmat(a,b,c,m,n)	0	-	0
// a,b,c are two dimensional arrays	0	-	0
// m,n are no of rows and cols	0	-	0
{	0	-	1
for i:= 1 to m do	1	m+1	m+1
for j:= 1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+b[i,j]	1	mn	mn
}	0	-	0
Total			2mn+2m+1

The frequency of first for loop is m+1. Similarly the frequency of frequency for the second for loop is m(n+1).

When you have obtained sufficient experience in computing step counts, you can avoid constructing the frequency table and obtain the step count as in the following example.

Example: The Fibonacci sequence of numbers starts as 0,1,1,2,3,5,8,13,21,34,55…..

Each new term is obtatined by taking the sum of the two previous terms.

```
Algorithm Fibonacci(n)    // compute the nth fibonacci number
{
   if ( n<= 1) then write (n);
      else
          {
                first=0;
                second=1;
                for i:= 2 to n do
                  {
                     next = first+second;
                     first=second;
                     second=next;
                  }
             write(next);
           }
 }
```

To analyse the time complexity of this algorithm, we need to consider the two cases (1) n=0 or 1 and (2) n is > 1. The total steps for the case n>1 is 4n+1.

The order of growth:

The size of n is either increased or decreased. The time analysis may not be always linearly proportional, because it depends upon what kind of basic operations we have in the algorithm. Therefore, while analyzing an algorithm's time efficiency the order of growth of n is also important. We expect the algorithms must execute faster, but for as n varies the execution time varies. To understand the meaning of the order of growth, let us list the most common computing time functions.

s.no	Function	Name
1	1	Constant
2	log n	Logarithmic
3	n	Linear
4	n log n	n log n
5	n^2	Quadratic
6	n^3	Cubic
7	2^n	Exponential
8	n!	Factorial

The functions shown are quite common in the analysis of algorithms. These functions have strong significance in terms of their relative values. The growth of these common functions typical values for n can be considered

n	**log n**	**n log n**	**n^2**	**n^3**	**2^n**	**n!**
1	0	0	1	1	2	1
2	1	2	4	8	4	2
4	2	8	16	64	16	24
8	3	24	64	512	256	40320
16	4	64	256	4096	65536	20922789888000
32	5	160	1024	32768	$4.2x10^9$	$2.6x10^{35}$
10^5	17	$1.7x10^6$	10^{10}	10^{15}	-	
10^6	20	$2.0x10^7$	10^{12}	10^{18}	-	

The function log n grows very slowly compared to any other function with respect to n. But 2^n and n! grows very fast.

ASYMPTOTIC NOTATION (O, Ω, Θ)

Asymptotic means study of function of parameter 'n' and 'n' becomes larger and larger without bound. Here we are concerned about how the running time of an algorithm increases with the size of the input. Generally an algorithm that is asyptotically more efficent will be the best choice.

Big-Oh Notation (O): Big-oh notation gives an upper bound and a function f(n). The upper bound of f(n) indicates that the function f(n) will be the worst case that it doesnot consume more than this computing time.

Def: The function f(n) = O(g(n)) (read as f of n is big-oh of g of n) if and only if There exists positive constants c and n_0 such that f(n) <= c* g(n) for all n, n>= n_0.

Ex 1) Given f(n) = 3n+2, then prove that f(n) = O(n).

f(n) = 3n+2

= 3n+2 <= 5n for all n>=1

The above inequality can be satisfied using the definition of Big-oh-notation by setting c=5, g(n)=n and n_0 = 1.

Therefore f(n)= O(n).

Ex 2) Given f(n) = $20n^3$-3 then prove that f(n) = $O(n^3)$

f(n) = $20n^3$-3

$20n^3 – 3 <= 20n^3$ for all n>=1

The above inequality can be satisfied using the definition of Big-oh notation by setting c=20 and g(n)=n^3, n_0=1

Therefore f(n) = $O(n^3)$

Ex 3) Given f(n) = $10n^2$ + 4n +3 then prove that f(n) = $O(n^2)$.

f(n) = $10n^2$ + 4n +3

$10n^2$ + 4n +3 <= $10n^2$ + 5n for all n>= 3

5n<=n^2 for all n>= 5

$10n^2$ + 4n +3 <= $10n^2$ + n^2 for all n>= 5

$10n^2$ + 4n +3 <= $11n^2$ for all n>= 5

The above inequality can be satisfied using the definition of Big-oh-notation by setting c=11 g(n)=n^2 and n_0=5

Therefore f(n)=$O(n^2)$.

We write O(1) to mean a computing time that is a constant

O(n) is called linear

$O(n^2)$ is called quadratic

$O(n^3)$ is called cubic

$O(2^n)$ is called exponential

O(log n) is called logarithmic

O(n log n) is called n log n

The relation among these computing time is

O(1) < O(log n) < O(n) < O(n log n) < $O(n^2)$ < $O(n^3)$ < $O(2^n)$

Ex 4) f(n) = n^3 log n+ n^2 log n + n log n

Time complexity is O (n^3 log n).

Ex 5) f(n) = $n^2 + n^3 + 4^n = O(4^n)$

The time complexity is $O(4^n)$

Ex 6) for =1 to n div 2 do

for j=1 to n*n do

x=x+1

Time complexity = (n/2) n^2 = $n^3/2$ = $O(n^3)$

Omega Notation Ω : This notation is used to find the lower bound behavior of f(n). The lower bound implies that below this time the algorithm cannot perform better. It is represented mathematically by notation Ω.

Def: f(n)=Ω(g(n)) iff there exists two positive constants C and n_0 .
f(n)>=c.g(n) for all n>=n_0

Ex 1) Given f(n) = 3n + 2 then prove that f(n) = Ω(n).
3n+2 >= 3n for all n>=1
The above inequality is satisfied according to Ω definition by setting
C=3 g(n)=n n_0=1
Therefore f(n)=Ω(n)

Ex 2) Given f(n) = $50n^2$+10n -5 then prove that f(n) = Ω(n^2).
$50n^2$+10n -5 >= $50n^2$ For all n>=1
The above inequality is satisfied according to Ω definition by setting
C=50 g(n)=n^2 and n_0=1 Therefore f(n) = Ω(n^2)

Theta Notation θ

For some functions the lower bound and the upper bound may be same. i.e. big Oh and omega will have the same function. For example to find minimum or maximum of an array of elements the computing time is O(n) and Ω(n).

There exists a special notation to denote for functions having the same time complexity for lower and upper bounds and this notation is called Theta θ Notation.
Def: f(n)=θ (g(n)) iff there exists three positive constants c_1 c_2 and n_0 with the constraint that
c_1 g(n) <= f(n) <= c_2 (g(n) for all n >= n_0
Ex 1) f(n)=$^1/_2n^2$-3n θ(n^2)
Ex2 f(n)=3n+2 θ(n)
Ex3 f(n)=10n2+4n+2 θ(n^2)
Ex 4 f(n)=10xlogn+4 θ(log n)

Little-Oh Notation (o) : The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2=O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We use o-notation to denote an upper bound that is not asymptotically tight.

Definition : f(n) = o(g(n), iff f(n) < c.g(n) for any constants c > 0, n_0>0 and n>n_0.
Or

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Examples
Ex1) Given f(n)=3n+2 The time complexity = o(n^2)
Ex2) Given f(n)=2^n, then prove that f(n)=o(n!).

$$f(n) = \lim_{n\to\infty} \frac{2^n}{n!}$$

=0
Therefore f(n)=o(n!)

BEST CASE, WORST CASE AND AVERAGE CASE EFFICIENCIES

1) Best Case : It is the minimum number of steps that can be executed for a given parameter.
2) Worst case : It is the maximum number of steps that can be executed for a given parameter.
3) Average case: It is the average number of steps executed for a given parameter.

Ex 1) Linear search (Sequential search)
Best Case:

2	4	5	6	8	9	10	11	13
1	2	3	4	5	6	7	8	9

If we want to search an element 2, whether it is present in the array or not, first A[1] is compared with 2. Match occurs. So the number of comparisons is only one. It is observed that search takes minimum number of comparisons, so it come under best case.
Time complexity is O(1).

Average Case: If we want to search an element 8, whether it is present in the array or not . first A[1] is compared with 8, no match occurs. Compare A[2],A[3] and A[4] with 8, no match occurs. Up to now 4 comparisons taken place. Now compare A[5] and 8 so match occurs. The number of comparisons are 5. It is observed that search takes average number of comparisons. So it comes under average case. If there are n elements, then we require n/2 comparisons. Time complexity is O(n/2) which is O(n). we can neglect constant.

Worst Case : If we want to search an element 13, whether it is present in the array or not. First A[1] is compared with 13. No match occurs. Continue this process until element is found or the list exhausted. The element is found at 9^{th} comparison. So number of comparisons are 9. It is observed that search takes maximum number of comparisons. So it comes under worst case.
Time complexity is O(n)

Note : If the element is not found in the list then we have to search entire list, so it comes under worst case.

Ex 2) Binary Search: The array elements are in ascending/descending order. The target is compared with middle element. If it is equal we stop our search otherwise we repeat the same process either in the first half or second half of the array depending on the value of target.
Best case time complexity O(1) : If the target found in the first comparison then it is called best case
Average case time complexity O(log n) :Average number of comparisons
Worst case time complexity O(log n) : maximum number of comparisons
Assume that the number of elements is considered as 2^m as every time the list is divided into two halfs.
$n = 2^m$
$\log (n) = \log(2^m)$
$\log (n) = m \log(2)$
$m = \log(n)/\log(2)$
$m = \log_2 n$ i.e. log n base 2
$m = \log (n)$
Maximum number of comparisons will be **m** in a binary search.

PROBABILISTIC ANALYSIS AND AMORTIZED ANALYSIS

Amortized analysis means finding the average running time per operation over a worst case sequence of operations. While giving the average case complexity probability is involved. On the other hand amortized analysis guarantees the time per operation over the worst case performance.

Amortized analysis assumes worst case input and typically does not allow random choices.

The average case analysis and amortized analysis are different. In average case analysis, we are averaging over all possible inputs whereas in amortized analysis we are averaging over a sequence of operations.

Amortized analysis does not allow random selection of input.

There are several techniques used in amortized analysis.

1) Aggregate analysis. : In this type of analysis the upper bound T(n) on the total cost of a sequence of n operations is decided, then the average cost is calculated as T(n)/n.
2) Accounting Method: In this method the individual cost of each operation is determined, by combining immediate execution time and its influence on the running time of future operations.
3) Potential method : it is like the accounting method, but overcharges operations early to compensate for undercharges later.

   A randomized algorithm or probabilistic algorithm is an algorithm which employs a degree of randomness as part of its logic. Probabilistic analysis is based on assuming something about the set of all possible inputs. This assumption is then used to design an efficient algorithm. If the characteristics of the input to an algorithm is unknown, Example, we do not know the distribution of the inputs, probabilistic analysis can not be used.

   Amortised analysis differs from average performance, in that probability is not involved. Actually amortised analysis is an analysis technique that provides tight bounds on the worst case running time of an algorithm.

PERFORMANCE MEASUREMENT

A good algorithm is correct, but a great algorithm is both correct and **efficient**. The most efficient algorithm is one that takes the least amount of execution time and memory usage possible while still yielding a correct answer.

Counting the operations

One way to measure the efficiency of an algorithm is to count how many operations it needs in order to find the answer across different input sizes.

Let's start by measuring the **linear search** algorithm, which finds a value in a list. The algorithm looks through each item in the list, checking each one to see if it equals the target value. If it finds the value, it immediately returns the index. If it never finds the value after checking every list item, it returns -1.

Here's what that algorithm looks like in pseudocode:

```
PROCEDURE searchList(numbers, targetNumber) {
  index ← 1
  REPEAT UNTIL (index > LENGTH(numbers)) {
    IF (numbers[index] = targetNumber) {
      RETURN index
    }
    index ← index + 1
  }
  RETURN -1
}
```

Note that this pseudocode assumes a 1-based index for lists.

Let's step through the algorithm for this sample input:

```
searchList([3, 37, 45, 57, 93, 120], 45)
```

The algorithm starts off by initializing the variable index to 1. It then begins to loop.

Iteration #1:

- It checks if `index` is greater than `LENGTH(numbers)`. Since 1 is *not* greater than 6, it executes the code inside the loop.
- It compares `numbers[index]` to `targetNumber`. Since 3 is *not* equal to 45, it does *not* execute the code inside the conditional.
- It increments `index` by 1, so it now stores 2.

Iteration #2:

- It checks if `index` is greater than `LENGTH(numbers)`. Since 2 is *not* greater than 6, it executes the code inside the loop.
- It compares `numbers[index]` to `targetNumber`. Since 37 is *not* equal to 45, it does *not* execute the code inside the conditional.
- It increments `index` by 1, so it now stores 3.

Iteration #3:

- It checks if `index` is greater than `LENGTH(numbers)`. Since 3 is *not* greater than 6, it executes the code inside the loop.
- It compares `numbers[index]` to `targetNumber`. Since 45 *is* equal to 45, it executes the code inside the conditional.
- It returns the current `index`, 3.

Now let's count the number of operations that the linear search algorithm needed to find that value, by counting how often each type of operation was called.

operation	# of times
Initialize index to 1	1
Check if `index` is greater than `LENGTH(numbers)`	3
Check if `numbers[index]` equals `targetNumber`	3
Return `index`	1
Increment `index` by 1	2

That's a total of 10 operations to find the targetNumber at the index of 3. Notice the connection to the number 3? The loop repeated 3 times, and each time, it executed 3 operations.

- The **best case** for an algorithm is the situation which requires the least number of operations. According to that table, the best case for linear search is when targetNumber is the very first item in the list.

What about the **worst case**?

- According to that table, the worst case is when targetNumber is the very last item in the list, since that requires repeating the 3 loop operations for every item in the list.
- There is actually a slightly worse worst case: the situation where targetNumber is not in the list at all. That'll require a couple extra operations, to check the loop condition and return -1.
- It doesn't require an entire extra loop repetition, however, so it's not that much worse. Depending on our use case for linear search, the worst case may come up very often or almost never at all.

The **average case** is when targetNumber is in the middle of the list. That's the example we started with, and that required 3 repetitions of the loop for the list of 6 items.

Let's describe the efficiency in more general terms, for a list of n items. The average case requires 1 operation for variable initialization, then n/2, 2 loop repetitions, with 3 operations per loop. That's 1+3(n/2).

This table shows the number of operations for increasing list sizes:

list size	# of operations
6	10
60	91
600	901

Empirical measurements

- ✓ The number of operations does *not* tell us the amount of time a computer will take to actually run an algorithm. The running time depends on implementation details like the speed of the computer, the programming language, and the translation of the language into machine code. That's why we typically describe efficiency in terms of number of operations.
- ✓ However, there are still times when a programmer finds it helpful to measure the run time of an implemented algorithm. Perhaps a particular operation takes a surprisingly long amount of time, or maybe there's some aspect of the programming language that's slowing an algorithm down.
- ✓ To measure the run time, we must make the algorithm runnable; we must implement it in an actual programming language.

Here's a translation from the pseudocode to JavaScript:

```
function searchList(numbers, targetNumber) {
    for (var index = 0; index < numbers.length; index++) {
        if (numbers[index] === targetNumber) {
            return index;
        }
    }
    return -1;
}
```

- ✓ The program below calls `searchList()` on the list of six numbers and reports the time taken in milliseconds.
- ✓ If your computer is as fast as mine, then you'll see 0 milliseconds reported. That's because this algorithm takes *way* less time than a millisecond. Our JavaScript environment can't measure time in units less than milliseconds, however.
- ✓ Here's what we'll try instead: we'll call `searchList()` 100,000 times, record how many milliseconds that takes, and divide to approximate the number of nanoseconds per call. That's what this program does:
- ✓ On my computer, that reports around 150 nanoseconds. There are 1,000,000,000 nanoseconds in a second, so that's really quite fast. In a faster language, environment, or computer, it could be even faster.
- ✓ That only tells us the average run-time for `searchList()` on a list of 6 numbers. We care more about how run time increases as the input size increases, however. Does it keep increasing linearly, like our above analysis predicted?
- ✓ The program below reports the run-time for lists of 6 numbers, 60 numbers, and 600 numbers.

Reference: https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/measuring-an-algorithms-efficiency

Randomized Algorithms

Randomized algorithm is a different design approach taken by the standard algorithms where few random bits are added to a part of their logic. They are different from deterministic algorithms; deterministic algorithms follow a definite procedure to get the same output every time an input is passed where randomized algorithms produce a different output every time they're executed. It is important to note that it is not the input that is randomized, but the logic of the standard algorithm.

- ✓ Unlike deterministic algorithms, randomized algorithms consider randomized bits of the logic along with the input that in turn contribute towards obtaining the output.
- ✓ However, the probability of randomized algorithms providing incorrect output cannot be ruled out either. Hence, the process called amplification is performed to reduce the likelihood of these erroneous outputs. Amplification is also an algorithm that is applied to execute some parts of the randomized algorithms multiple times to increase the probability of correctness. However, too much amplification can also exceed the time constraints making the algorithm ineffective.

Classification of Randomized Algorithms

Randomized algorithms are classified based on whether they have time constraints as the random variable or deterministic values. They are designed in their two common forms − Las Vegas and Monte Carlo.

- ***Las Vegas*** − The Las Vegas method of randomized algorithms never gives incorrect outputs, making the time constraint as the random variable. For example, in string matching algorithms, las vegas algorithms start from the beginning once they encounter an error. This increases the probability of correctness. Eg., Randomized Quick Sort Algorithm.
- ***Monte Carlo*** − The Monte Carlo method of randomized algorithms focuses on finishing the execution within the given time constraint. Therefore, the running time of this method is deterministic. For example, in string matching, if monte carlo encounters an error, it restarts the algorithm from the same point. Thus, saving time. Eg., Karger's Minimum Cut Algorithm

Need for Randomized Algorithms

This approach is usually adopted to reduce the time complexity and space complexity. But there might be some ambiguity about how adding randomness will decrease the runtime and memory stored, instead of increasing; we will understand that using the game theory.

The Game Theory and Randomized Algorithms

The basic idea of game theory actually provides with few models that help understand how decision-makers in a game interact with each other. These game theoretical models use assumptions to figure out the decision-making structure of the players in a game. The popular assumptions made by these theoretical models are that the players are both rational and take into account what the opponent player would decide to do in a particular situation of a game. We will apply this theory on randomized algorithms.

Zero-sum game

The zero-sum game is a mathematical representation of the game theory. It has two players where the result is a gain for one player while it is an equivalent loss to the other player. So, the net improvement is the sum of both players' status which sums up to zero.

Randomized algorithms are based on the zero-sum game of designing an algorithm that gives lowest time complexity for all inputs. There are two players in the game; one designs the algorithm and the opponent provides with inputs for the algorithm. The player two needs to give the input in such a way that it will yield the worst time complexity for them to win the game. Whereas, the player one needs to design an algorithm that takes minimum time to execute any input given.

For example, consider the quick sort algorithm where the main algorithm starts from selecting the pivot element. But, if the player in zero-sum game chooses the sorted list as an input, the standard algorithm provides the worst case time complexity. Therefore, randomizing the pivot selection would execute the algorithm faster than the worst time complexity. However, even if the algorithm chose the first element as pivot randomly and obtains the worst time complexity, executing it another time with the same input will solve the problem since it chooses another pivot this time.

On the other hand, for algorithms like merge sort the time complexity does not depend on the input; even if the algorithm is randomized the time complexity will always remain the same. Hence, **randomization is only applied on algorithms whose complexity depends on the input.**