Design and Analysis of Computer Algorithms

Unit I

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

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. Performing arithmetic on integers is an example of an effective e operation, but arithmetic with real numbers is not, since some values may be expressible only by infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property.

Algorithm Specification

Pseudocode Conventions

We can describe an algorithm in many ways such as use natural language like English, flowcharts etc. Most of our algorithms are using a pseudocode that resembles C and Pascal.

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces: { and }. A compound sttatement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;.
3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. We assume simple data types such as integer, float, char, Boolean, and so on. Compound data types can be formed with records. Here is an example:

```
node = record
        {
        Datatype_1 data-l;
                ..
        Datatype_n data_n;
```

```
node      *link;
}
```

In this example, link is a pointer to the record type node. Individual data items of a record can be accessed with --> and period.

4. Assignment of values to variables is done using the assignment statement

   <variable> := <expression>

5. There are two boolean values true and false. In order to produce these values, the logical operators and, or, and not and the relational operators <,≤,=, ≠, ≥ and > are provided.
6. Elements of multidimensional arrays are accessed using [and]. For example, the (i,j)th element of the array is denoted as A[i, j]. Array indices start at zero.
7. The following looping statements are employed: for, while, and repeat-until. The while loop takes the following form:

```
while <condition> do
{
    <statement 1>
        ……..
    <statement n>
 }
```

The general form of a for loop is

```
for variable := value1 to value2 step step do
{
<statement 1>
……..
<statement n>
}
```

Here valuel, value2, and step are arithmetic expressions. A variable of type integer or real or a numerical constant is a simple form of an arithmetic expression. The clause "step step" is optional and taken as +1 if it does not occur, step could either be positive or negative. variable is tested for termination at the start of each iteration.

A repeat-until statement is constructed as follows:

```
repeat
{
<statement 1>
……
```

<statement n>

until <condition>

The statements are executed as long as (condition) is false. The value of <condition> is computed after executing the statements.

The instruction break; can be used within any of the above looping instructions to force exit. In case of nested loops, break, results in the exit of the innermost loop that it is a part of. A return statement within any of the above also will result in exiting the loops. A return statement results in the exit of the function itself.

A conditional statement has the following forms:

if <condition> then <statement>

if <condition> then <statement 1> else <statement 2>

We also employ the following case statement:

```
case
{
        : <condition 1): <statement 1>
                ……
        : <condition 1): <statement 1>
        :else:<statement n + 1>
}
```

Here <statement 1>, <statement 2>, etc. could be either simple statements or compound statements. A case statement is interpreted as follows. If <condition 1> is true, <statement 1> gets executed and the statement is exited. If <statement 1> is false, <condition 2> is evaluated. If <condition 2> is true, <statement 2> gets executed and the case statement exited, and so on. If none of the conditions <condition 1>, ... , {condition n) are true, <statement n+1> is executed and the case statement is exited. The else clause is optional.

Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.

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})

where Name is the name of the procedure and ({parameter list)) is a listing of the procedure parameters. The body has one or more (simple or compound) statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are

passed by value. Arrays and records are passed by reference. An array name or a record name is treated as a pointer to the respective data type:

As an example, the following algorithm finds and returns the maximum of n given numbers:

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

In this algorithm (named Max), A and n are procedure parameters. Result and i are local variables.

Algorithm Design Techniques

For a given problem, there are many ways to solve them. The different methods are listed below.

1. Divide and Conquer.
2. Greedy Algorithm.
3. Dynamic Programming.
4. Branch and Bound.
5. Backtracking Algorithms.
6. Randomized Algorithm.

1. Divide and Conquer: Divide and conquer method consists of three steps.
   a. Divide the original problem into a set of sub-problems.
   b. Solve every sub-problem individually, recursively.
   c. Combine the solutions of the sub-problems into a solution of the whole original problem.
2. Greedy Approach:

Greedy algorithms seek to optimize a function by making choices which are the best locally but do not look at the global problem. The result is a good solution but not necessarily the best one. Greedy Algorithm does not always guarantee the optimal solution however it generally produces solutions that are very close in value to the optimal solution.

3. Dynamic Programming.

Dynamic Programming is a technique for efficient solution. It is a method of solving problems exhibiting the properties of overlapping sub problems and optimal sub-structure that takes much less time than other methods.

4. Branch and Bound Algorithm.

In Branch and Bound Algorithm a given Algorithm which cannot be bounded has to be divided into at least two new restricted sub-problems. Branch and Bound Algorithm can be slow, however in the worst case they require efforts that grows exponentially with problem size. But in some cases the methods converge with much less effort. Branch and Bound Algorithms are methods for global optimization in non-convex problems.

5. Backtracking Algorithm.

They try each possibility until they find the right one. It is a depth first search of a set of possible solution. During the search if an alternative doesn't work, the search backtrack to the choice point, the place which presented different alternatives and tries the next alternative. If there are no more choice points the search fails.

6. Randomized Algorithm.

A Randomized Algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased, random bits, and it is then allowed to use these random bits to influence its computation.

Performance evaluation can be loosely divided into two major phases:

(1) a priori estimates and (2) a posteriori testing. We refer to these as performance analysis and performance measurement respectively.

Performance Analysis

There are two criteria for judging algorithms that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

The Space/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.

Space Complexity

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

1. A fixed part that is independent of the characteristics (eg. Number, size) of the inputs and outputs. This part typically includes the instruction space (space for code), space

for simple variable and fixed-size component variables (aggregate), space for constants and so on.

2. A variable part that consists of the 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 this depends on instance characteristics) and the recursion stack space (depends on the instance characteristics)

The space requirement S(P) of any algorithm P may therefore be written S (P) = c+ Sp (instance characteristics), where c is a constant. When analyzing the space complexity of an algorithm, we concentrate solely on estimating Sp (instanced characteristics). For any given problem, we need first to determine which instance characteristics to use to measure the space requirements.
Algorithm abc computes a+b+b*c+(a+b-c)/(a+ b)+4.0

```
1. Algorithm abc(a, b, c)
2. {
3.       Return a+b+b*c+(a+b-c)/(a+ b)+4.0;
4. }
```

Algorithm Sum computes $\sum_{i=1}^{n}$ a[i] iteratively, where the a[i]'s are real numbers; and RSum is a recursive algorithm that computes $\sum$ a[i].

Algorithm Sum

```
1. Algorithm Sum(a, n)
2. {
3.     s:=0.0;
4.     for i:= 1 to n do
5.         s:= s+ a [i];
6.     return s;
7. }
```

In the case of the algorithm abc, the problem instance is characterized by the specific values of a, b, and c. One word is adequate to store the values of each of a, b, c so space needed by abc is independent of the instance characteristics. Consequently, Sp(instance characteristics)= 0.
The problem instances for algorithm Sum 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. Then, Ssum(n) ≥ (n+3) (n for a[], one each for n, I, and s).

Time Complexity

The time T (P) taken by a program P is the sum of the compile time and the run (execution) time. The compile time does not depend on the instance characteristics. This run time is denoted by t*p* (instance characteristics). If we know the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on, that would be made by the code for P is used.

So we could obtain an expression for t*p* (n) of the form

$t_p(n) = c_a\ ADD\ (n) + c_s\ SUB(n) + c_m\ MUL\ (n) + c_d\ DIV\ (n) + \ldots$

Where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, and so on, respectively, denote the time needed for an addition, subtraction, multiplication, division and so on, and *ADD*, *SUB*, *MUL*, *DIV* and son, are functions whose values are the numbers of additions, subtractions, multiplications, divisions, and so on, that are performed when the code for P is used on an instance with characteristic n.

The runtime of an execution depends on the parameters such as,

- Number of comparisons
- Number of data exchanges.
- Presence of recursive executions.
- The Runtime is proportional to the size of the program.
- The execution time increases as the Input data set increases in size.

To illustrate how to find Time complexity. It is done by two methods as,

1. Step count method
2. Step per statement execution or frequency method

Step count method

In order to calculate the time of execution it is necessary to obtain a count for the total number of operations. It is done by count the number of program steps in a program. 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, the entire statement

return a+b+b*c+(a+b-c)/(a+ b)+4.0; of algorithm abc could be regarded as a step.

The number of steps any program statement is assigned depended on the kind of statement. For example, comments count as zero steps; an 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. The control parts of for and while statements have the following forms:

for i:= <expr> to <expr1> do

while (<expr>) do

Each execution of the control part of a while statement is given a step count equal to the number of step counts assignable to <expr>.

As an example the following algorithm shows the count variable of algorithm sum.

```
1.  Algorithm Sum(a, n)
2.  {
3.      s:=0.0;
4.      count:= count + 1; //count is global; it is initially zero.
5.      for i:= 1 to n do
6.      {
7.      count:= count + 1;  //For for
8.          s:= s+ a [i]; count:= count + 1; // For assignment
9.      }
10.     count:= count + 1; // For last time of for
11.     count:= count + 1; // For the return
12.     return s;
13. }
```

Following algorithm shows the simplified form.

```
1.  Algorithm Sum (a,n)
2.  {
3.  for i:= 1 to n do count := count + 2;
4.  count := count + 3;
5.  }
```

It is easy to see that in the **for** loop, the value of *count* will increase by a total of 2n. If *count* is zero to start with, then it will be 2n + 3 on termination. So each invocation of Sum executes a total of 2n + 3 steps.

Input size: One of the instance characteristics that is frequently used in the literature is the input size. The input size of any instance of a problem is defined to be the number of words (number of elements) needed to describe that instance. The input size for the problem of summing an array with n elements n + 1, n for listing the n elements and 1 for the value of n. The input size is normally taken to be the number of bits needed to specify that element.

Step per statement execution or frequency method

The second 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. For this first determining the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed. The s/e of a statement is the amount by which the count changes as a result of the execution of that statement. By combining these two quantities, the total contribution of each statement is obtained. By adding the contributions of all statements, the step count for the entire algorithm is obtained.

Following table shows the s/e of algorithm sum

Statement	s/e	frequency	total steps
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			$2n+3$

In the case of Sum algorithm, the total number of steps required by the algorithm is determined to be 2n+ 3. It is important to note that the frequency of the 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.

There are three kinds of step counts: **best case, worst case, and average**. The **best case step count** is the minimum number of steps that can be executed for the given parameters. The **worst case step count** is the maximum number of steps that can be executed for the given parameters. The **average step count** is the average number of steps executed on instances with the given parameters.

Eg. searching

Best case: If the algorithm finds the element at the first search itself, it is referred as a best case algorithm.

Worst case : If the algorithm finds the element at the end of the search or if the searching of the element fails, the algorithm is in the worst case or it requires maximum number of steps that can be executed for the given parameters.

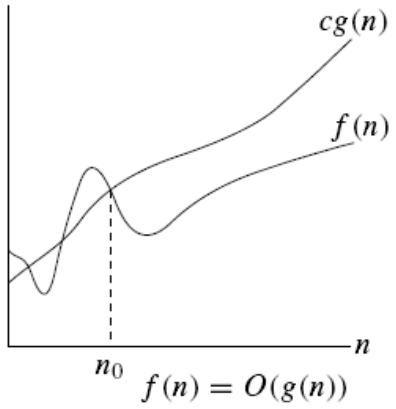
Average case: The analysis of average case behavior is complex than best case and worst case analysis, and is taken by the probability of Input data. As the volume of Input data increases, the average case algorithm behaves like worst case algorithm. In an average case behavior; the searched element is located in between a position of the first and last element.

Asymptotic Notation (O, Ω, Θ)

Big O-notation(O -Big "oh") (Asymptotic upper bound)

The O-notation (pronounced as Big “Oh”) is used to measure the performance of an algorithm which depends on the volume of Input data. The O-notation is used to define the order of growth of an algorithm, as the input size increases, the performance varies.

We write $f(n) = O(g(n))$if there exist constants $c> 0$, $n0> 0$such that $0 \leq f(n) \leq cg(n)$for all $n \geq n0$.



Eg. Suppose we have a program having count 3n+2. We can write it as f (n) =3n+2. We say that

3n+2=O (n), that is of the order of n, because f (n) <=4n or 3n + 2 <= 4n, for all n>=2.

Normally suppose a program has step count equals 5, and then we say that it has an order O (constant) or O (1).

If f (n) = 3n+5 then O (n) or O (n2) or O (n3) or ---

But we cannot express its time complexity as O (1).

If f (n) = 5n2+8n+2 then O (n2) or O (n3) or ---

But we cannot express its time complexity as O (n) or O (1).

If it is 6n3+3n+2 then O (n3) or O (n4) or ----

But we cannot express its time complexity as O (n2) or O (n) or O (1).

If it is n log n+6n+9 then O (n log n) (to the base 2)

If it is log n+6 then O (log n)

The seven various O-notations used are:

- O (1) – We write O (1) to mean a computing time that is constant. Where the data item is searched in the first search itself.
- (n) – O (n) is called linear, where all the elements of the list are traversed and searched and the element is located at the nth location.

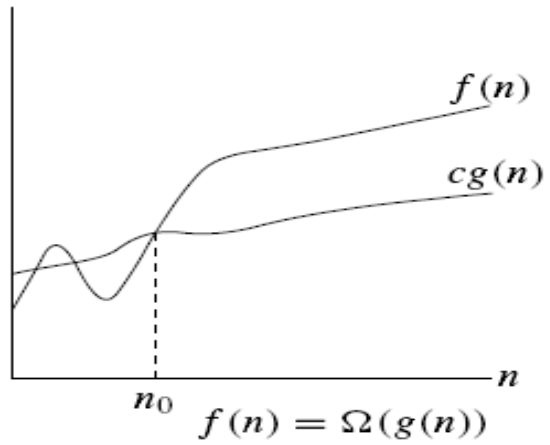
- (n2) – It is called quadratic, where all elements of the list are traversed for each element.

  E.g.: worst case of bubble sort.
- (log n) –The O(log n) is sufficiently faster, for a large value of n, when the searching is done by dividing a list of items into 2 half's and each time a half is traversed based on middle element. E.g.: binary searching, binary tree traversal.
- (nlogn) – The O (nlogn) is better than O(n2),but not good as O(n),when the list is divided into 2-halfs and a half is traversed each time. E.g.: Quick sort.

The other notations are O (n2), O (n3), O (2n) etc.

Big-Omega notation (Ω)(Asymptotic lower bound)

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$



Suppose we have a program having count 3n+2. we can write it as f(n)=3n+2. And we say that 3n+2=W(n), that is of the order of n, because f(n)>=3n or 3n+2>=3n, for all n>=1. Normally suppose a program has step count equals 5, and then we say that it has an order W(constant) or W(1).

If f(n) = 3n+5 then W(n) or W(1)

But we cannot express its time complexity as O(n2).

If f(n) = 5n2+8n+2 then W(n2) or W(n) or W(1)

But we cannot express its time complexity as O(n3) or O(n4).

If f(n) = 6n3+3n+2 then W(n3) or W(n2) or W(n) or W(1)

But we cannot express its time complexity as O(n4)

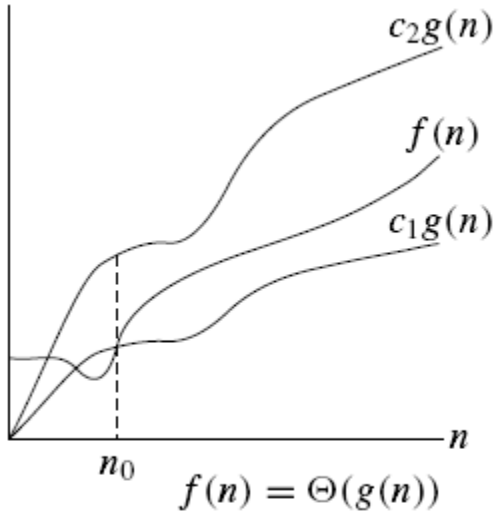
If f(n) = n log n+6n+9 then W(n log n) (to the base 2)

If f(n) = log n+6 then W(log n)

Big-theta notation

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.[1]

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$



(a)

Suppose we have a program having count 3n+2. we can write it as f(n)=3n+2. We say that 3n+2=q (n),that is of the order of n, because c1. n >= 3n + 2 <= c2. n .after a particular value of n.

Normally suppose a program has step count equals 5, then we say that it has an order q (constant) or q (1).

If f(n) = 3n+5 then q (n)

But we cannot express its time complexity as q(n2) or q(1).

If f(n) = 5n2+8n+2 then q (n2).

But we cannot express its time complexity as q(n3) or q (n4) or q(n) or q(1).

If f(n) = 6n3+3n+2 then q(n3)

But we cannot express its time complexity as O(n4) or q(n2) or q(n) or q(1).

If f(n) = n log n+6n+9 then q (n log n) (to the base 2)

If f(n) = log n+6 then q (log n)

From this we may conclude that q is the most precise notation. For the binary search program we will

get a time of the order log n. (to the base 2)

The asymptotic complexity can be determined easily without determining the exact step count. This is usually done by first determining the asymptotic complexity of each statement in the algorithm and then adding these complexities. Following tables show asymptotic complexity of Sum

Statement		s/e	frequency	total steps
1	**Algorithm** Sum(a, n)	0	–	$\Theta(0)$
2	{	0	–	$\Theta(0)$
3	$s := 0.0;$	1	1	$\Theta(1)$
4	**for** $i := 1$ **to** n **do**	1	$n+1$	$\Theta(n)$
5	$s := s + a[i];$	1	n	$\Theta(n)$
6	**return** $s;$	1	1	$\Theta(1)$
7	}	0	–	$\Theta(0)$
Total				$\Theta(n)$

Table 1.4 Asymptotic complexity of Sum (Algorithm 1.6)

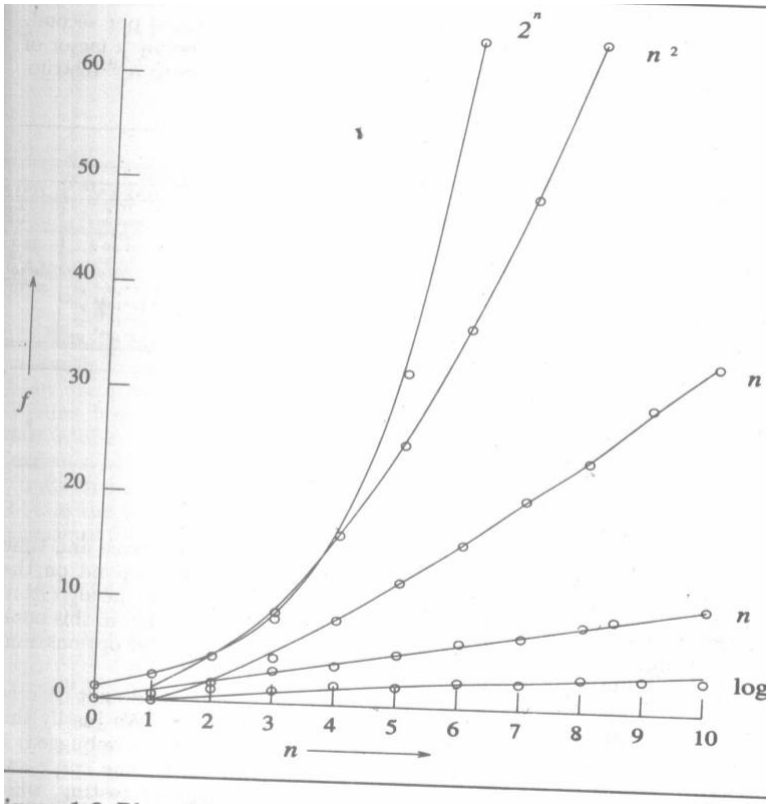
Practical complexities

We have seen that the time complexity of an algorithm is generally some function of the instance characteristics. The complexity function can also be used to compare two algorithms P and Q that perform the same task. Assume that algorithm P has complexity $\Theta(n)$ and algorithm Q has complexity $\Theta(n^2)$. We can assert that algorithm P is faster than algorithm Q for sufficiently large n. When deciding which of the two algorithms to use, you must know whether the n you are dealing with is, in fact, sufficiently large. If algorithm P runs in 10^6 n milliseconds, whereas algorithm Q runs in n^2 milliseconds, and if you always have $n \leq 10^6$, then, other factors being equal, algorithm Q is the one to use.

To get a feel for how the various functions grow with n, observe the following table and figure very closely.

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4,096	65,536
5	32	160	1,024	32,768	4,294,967,296

able 1.7 Function values



igure 1.3 Plot of function values

The function 2^n grows very rapidly with n. In fact, if an algorithm needs 2^n steps for execution, then when n = 40, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing one billion steps per second, this would require about 18.3 minutes. If n = 50, the same algorithm would run for about 13 days on this computer. So we can say that the utility of algorithms with exponential complexity is limited to small n (typically $n \leq 40$).

Algorithms that have a complexity that is a polynomial of high degree are also of limited utility. If the value of n is reasonably large (n > 100) small complexity such as n, nlogn, n^2, and n^3 are only feasible.

Polylog, polynomial, and exponential: These are the most common functions that arise in analyzing algorithms:

Polylogarithmic: Powers of log *n*, such as $(\log n)^7$. We will usually write this as $\log^7 n$.

Polynomial: Powers of *n*, such as n^4

Exponential: A constant (not 1) raised to the power *n*, such as 3^n.

Performance measurement

Performance measurement is concerned with obtaining the space and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as on the computer on which the algorithm is run. To obtain the computing or run time of a program,

we need a clocking procedure. We assume the existence of a program GetTime() that returns the current time in milliseconds.