Design and Analysis of Algorithms

Unit - I

Dr. R. Bhuvaneswari
Assistant Professor
Department of Computer Science
Periyar Govt. Arts College, Cuddalore.

Periyar Govt. Arts College
Cuddalore

Introduction to the Concept of Algorithms

Syllabus

UNIT-I

Algorithm Analysis – Time Space Tradeoff – Asymptotic Notations – Conditional asymptotic notation – Removing condition from the conditional asymptotic notation - Properties of big-Oh notation – Recurrence equations – Solving recurrence equations – Analysis of linear search.

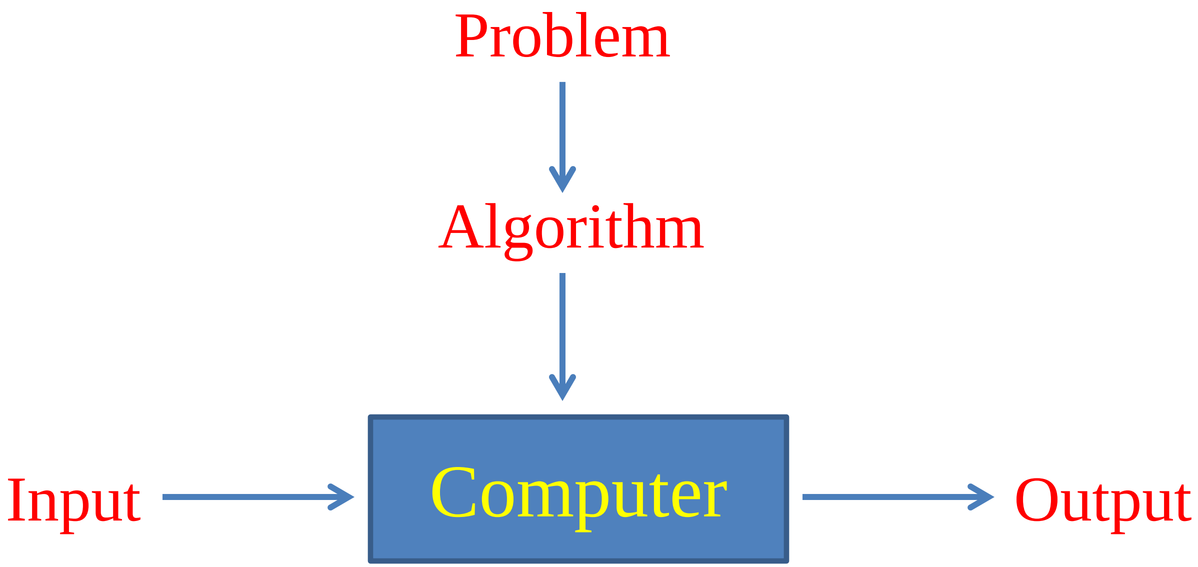
Text Book:

K.S. Easwarakumar, Object Oriented Data Structures using C++, Vikas Publishing House pvt. Ltd., 2000 (For Unit I)

Introduction to the Concept of Algorithms

- Algorithm
- Problem Solving
- Design of an Algorithm
- Analysis of an algorithm

Notion of an Algorithm



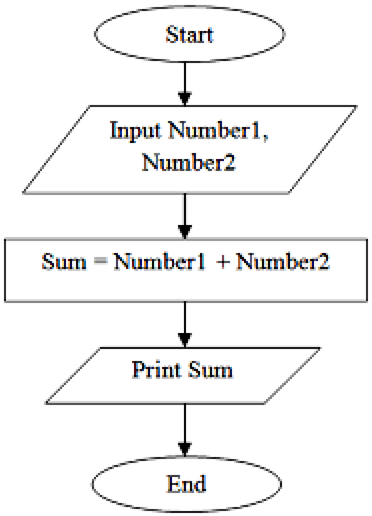
Algorithm

- An **algorithm** is a finite set of instructions that, if followed, accomplishes a particular task i.e., for obtaining a required output for any legitimate input in a finite amount of time.
- All algorithms must satisfy the following criteria:
  - **Definiteness.** Each instruction is clear and unambiguous.
  - **Effectiveness.** Every instruction must be very basic so that it can carried out, by a person using pencil and paper.
  - **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
  - **Input.** Zero or more quantities are externally supplied.
  - **Output.** At least one quantity is produced.

Algorithm Specification

- An **algorithm** can be described in three ways:
  - Natural language in English
  - Graphic representation called flowchart
  - **Pseudo-code method**
    - In this method we typically represent algorithms as program, which resembles C language

1. Input two numbers
2. Add the two numbers
3. Print the result



Pseudo-code Conventions

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces **{** and **}**.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Assignment of values to variables is done using the assignment statement.

   ‹variable› := ‹expression›;

5. There are two Boolean values **true** and **false**.
   - Logical operators: AND, OR, NOT
   - Relational operators: $<, \leq, =, \neq, >, \geq$

Pseudo-code Conventions

6. The following looping statements are used: **while**, **for** and **repeat-until**

while loop:

```
while ‹condition› do
{
      ‹statement 1›
           .
           .
      ‹statement n›
}
```

for loop:

```
for variable:= value1 to  value2
           step step-value do
{
           ‹statement 1›
                .
                .
           ‹statement n›
}
```

repeat-until:

```
repeat
           ‹statement 1›
                .
                .
           ‹statement n›
until ‹condition›
```

Pseudo-code Conventions

7. A conditional statement has the following forms:

    if ‹condition› then ‹statement›

    if ‹condition› then ‹statement 1› **else** ‹statement 2›

case statement:

case

{

:‹condition 1›: ‹statement 1›

.

.

:‹condition n›: ‹statement n›

:**else**: ‹statement n+1›

}

Pseudo-code Conventions

8. Input and output are done using the instructions read and write.
9. There is only one type of procedure: Algorithm.

    Algorithm contains

    - **Heading**
    - **Body**

    The heading takes the form

```
Algorithm Name (‹parameter list›) ——→ heading
{
        ……
        ……      body
}
```

Pseudo-code Conventions

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
7. Result := A[i];
8. return Result;
9. }

n = 5, result = 10

A[1] = 10

A[2] = 87 result = 87

A[3] = 45

A[4] = 66

A[5] = 99 result = 99

Algorithm Analysis

Study of algorithm involves three major parts:

- Designing the algorithm
- Proving the correctness of the algorithm
- **Analysing the algorithm**

Analysing the algorithm deals with

1. Space Complexity
2. Time Complexity

Practically, time and space complexity can be reduced only to certain levels, as later on reduction of time increases the space and vice-versa → **time-space trade-off**.

Dr. R. Bhuvaneswari

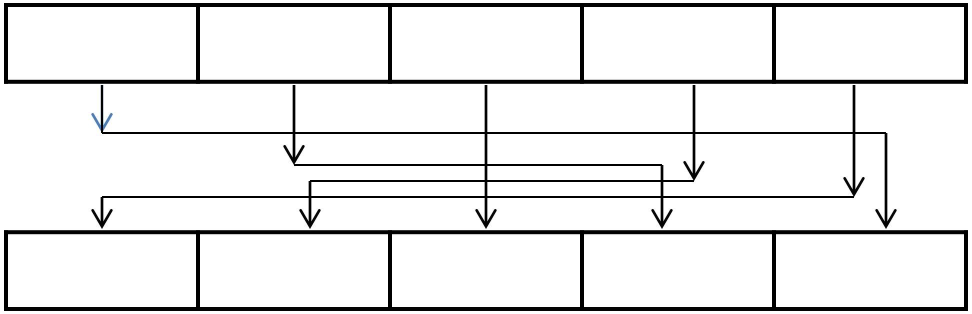
Periyar Govt. Arts College
Cuddalore

Algorithm Analysis

Method - 1

```
int ary1[n];
int ary2[n];
for (int i=0; i<n; i++)
      ary2[i] = ary1[(n-1)-i];
```

- An extra array of size n is used
- So total space required is 2n
- n assignments are made and the time complexity is n units of time.

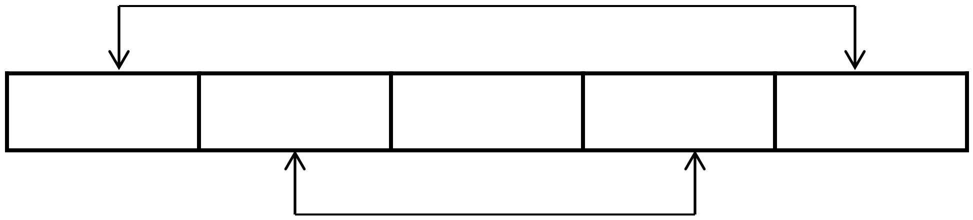


Algorithm Analysis

```
int  ary1[n];
int k = floor(n/2);
for (int i=0;i<k;i++)
   swap(&ary1[i],&ary1[(n-1)-i];

swap(int *a, int *b)
{
int temp = *a;
*a = *b;
*b = *temp;
}
```

- One array of size n and a temporary variable temp is used.
- So space occupied is n+1
- Swapping - 3 assignments are required.
- Number of times performed is half the size of the array.
- So the time complexity is 3n/2.



Asymptotic Notations

- Three standard notations
  - **Big-oh (O)** : asymptotic "less than"
    - F(n) = O(g(n)) implies: f(n) "≤" g(n)
  - **Big omega (Ω)** : asymptotic "greater than"
    - F(n) = Ω(g(n)) implies: f(n) "≥" g(n)
  - **Theta (θ)** : asymptotic "equality"
    - F(n) = θ(g(n)) implies: f(n) "=" g(n)
- Time complexity of a function may be one of the following

$$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$$

Asymptotic Notations

Big-Oh

The function f(n) = O(g(n)) if and only if there exists positive constant c and n_0 such that $f(n) \leq c*g(n)$ for every $n \geq n_0$

Example:

$$f(n) = 2n+3$$

1. $2n+3 \leq 10n$ for every $n \geq 1$
   $f(n) = O(n)$

2. $2n+3 \leq 2n^2 + 3n^2$
   $2n+3 \leq 5n^2$
   $f(n) = O(n^2)$

Omega

The function $f(n) = \Omega(g(n))$ if and only if there exists positive constant c and n_0 such that $f(n) \geq c*g(n)$ for every $n \geq n_0$

Example:

$$f(n) = 2n+3$$

1. $2n+3 \geq 1*n$ for every $n \geq 1$
   $f(n) = \Omega(n)$

2. $2n+3 \geq 1*\log n$
   $f(n) = \Omega(\log n)$

Asymptotic Notations

Theta

The function $f(n) = \theta(g(n))$ if and only if there exists positive constant c_1, c_2 and n_0 such that

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for every } n \geq n_0$$

Example:

$$f(n) = 2n+3$$

$$1*n \leq 2n+3 \leq 5*n$$

$$f(n) = \theta(n)$$

Properties of big oh(O) notation

1. $O(f(n)) + O(g(n)) = O(\max\{f(n),g(n)\})$
2. $F(n) = O(g(n))$ and $g(n) \leq h(n)$ implies $f(n) = O(h(n))$
3. Any function can be said as an order of itself. That is, $f(n) = O(f(n))$

   $f(n) = 1*f(n)$
4. Any constant value is equivalent to $O(1)$. That is, $c = O(1)$
5. If $\lim_{n\to\infty}\{f(n)/g(n)\} \in R > 0$ then $f(n) \in \theta(g(n))$

   $R \rightarrow$ set of non negative real numbers
6. If $\lim_{n\to\infty}\{f(n)/g(n)\} = 0$ then $f(n) \in O(g(n))$ but $f(n) \notin \theta(g(n)$
7. If $\lim_{n\to\infty}\{f(n)/g(n)\} = \infty$ then $f(n) \in \Omega(g(n))$ but $f(n) \notin \theta(g(n)$

Recurrence Equations

Recurrence equations can be classified into

- **Homogeneous recurrence equations**
- **Inhomogeneous recurrence equations**

Suppose T(n) is the time complexity of an algorithm for the input size n.

Assume that T(n) is recursively defined as

$$T(n) = b_1T(n-1) + b_2T(n-2) + \ldots\ldots + b_kT(n-k)$$

$$\Rightarrow a_0T(n) + a_1T(n-1) + \ldots\ldots\ldots + a_kT(n-k) = 0$$

Let us denote T(i) as x^i

$$a_0x^n + a_1x^{n-1} + \ldots\ldots\ldots + a_kx^{n-k} = 0$$

which is a **homogeneous recurrence** equation.

$$a_0x^k + a_1x^{k-1} + \ldots\ldots\ldots + a_k = 0, \qquad n=k$$

will have k roots. Let the roots be $r_1, r_2, \ldots\ldots r_k$.

They may or may not be same.

Dr. R. Bhuvaneswari

Homogeneous Recurrence Equations

Solving homogeneous recurrence equation

Case (i): All roots are distinct

eg. $x^2-5x+6 = 0$

$(x-3)(x-2) = 0 \quad \Rightarrow x = 3$ and 2

General solution is $T(n) = c_1 3^n + c_2 2^n$

$$T(n) = \sum_{i=1}^{k} C_i r_i^n$$

Case (ii): Suppose some of p roots are equal and the remaining are distinct.

eg. $(x-2)^3(x-3) = 0 \quad \Rightarrow x = 2,2,2,3$

General solution is $T(n) = C_1 2^n + C_2 n2^n + C_3 n^2 2^n + C_4 3^n$

$$T(n) = \sum_{i=1}^{p} C_i\, n^{i-1} r_i^n + \sum_{i=p+1}^{k} C_i r_i^n$$

Inhomogeneous Recurrence Equations

A **linear non-homogenous recurrence relation** with constant coefficients is a recurrence relation of the form

$$a_n = c_1a_{n-1} + c_2a_{n-2} + \ldots + c_ka_{n-k} + f(n)$$

where c1, c2, …, ck are real numbers, and f(n) is a function depending only on n.

The recurrence relation

$$a_n = c_1a_{n-1} + c_2a_{n-2} + \ldots + c_ka_{n-k},$$

is called the **associated homogeneous recurrence relation**.

This recurrence includes k initial conditions.

$a_0 = C_0$, $a_1 = C_1 \ldots a_k = C_k$

Dr. R. Bhuvaneswari

Periyar Govt. Arts College
Cuddalore

Inhomogeneous Recurrence Equations

Case (i): Solve the recurrence equation

$$T(n) - 2T(n-1) = 1 \quad \text{subject to } T(0) = 0$$

Proof: The characteristic equation is $(x-2)(x-1)=0$. Therefore, the roots are 2 and 1. Now, the general solution is

$$T(n) = c_1 1^n + c_2 2^n$$

Since $T(0) = 0$, from the given equation $T(1)$ will be 1.

Thus, from the general solution we get $c_1 = -1$ and $c_2 = 1$.

So,

$$T(n) = 2^n - 1 = \theta(2^n)$$

$n = 0,\ T(0) = c_1 + c_2$
ie., $c_1 + c_2 = 0$ ------- 1
$n = 1,\ T(1) = c_1 + 2c_2$
ie., $c_1 + 2c_2 = 1$ ------- 2
from 1, $c_1 = -c_2$
substituting in 2,
$-c_2 + 2c_2 = 1 \rightarrow c_2 = 1$
from 1, $c_2 = -c_1$
substituting in 2,
$c_1 + 2(-c_1) = 1 \rightarrow c_1 = -1$

Inhomogeneous Recurrence Equations

Case (ii): Solve the recurrence equation

$$T(n) = 2T(n-1) + n2^n + n^2$$

Proof: The characteristic equation is $(x-2)(x-2)^2(x-1)^3 = 0$. That is $(x-2)^3(x-1)^3 = 0$. Therefore, the roots are 2, 2, 2, 1, 1 and 1. Now, the general solution is

$$T(n) = c_1 2^n + c_2 n2^n + c_3 n^2 2^n + c_4 1^n + c_5 n1^n + c_6 n^2 1^n$$

Hence, $T(n) = O(n^2 2^n)$

Analysis of linear search

- Algorithms are analyzed to get best-case, worst-case and average-case.
- Each problem is defined on a certain domain
  - eg. Algorithm to multiply two integers. In this case, the domain of the problem is a set of integers.
- From the domain, we can derive an instance of the problem.
  - Any two integers may be an instance to the above problem.
- So, when an algorithm is analyzed, it is necessary that the analyzed value is satisfiable for all instances of the domain.
- Let D_n be the domain of a problem, where n be the size of the input.
- Let $I \in D_n$ be an instance of the problem taken from the domain Dn.
- T(I) be the computation time of the algorithm for the instance $I \in D_n$

Analysis of linear search

Best-case analysis:

This gives the minimum computed time of the algorithm with respect to all instances from the respective domain.

$$B(n) = \min\{T(I) \mid I \in D_n\}$$

Worst-case analysis:

This gives the maximum computation time of the algorithm with respect to all instances from the respective domain.

$$W(n) = \max\{T(I) \mid I \in D_n\}$$

Average-case analysis:

$$A(n) = \sum_{I \in Dn} p(I)T(I)$$

where p(I) is the average probability with respect to the instance I.

Analysis of linear search

```
int linearsearch(char A[], int size, char ch)
{
 for (int i=0; i<size; i++)
 {
  if (A[i] == ch)
        return(i);
 }
 return(-1);
}
```

Location of the element	Number of comparisons required
0	1
1	2
2	3
. .	. .
n-1	n
not in the array	n

Dr. R. Bhuvaneswari

Periyar Govt. Arts College
Cuddalore

Analysis of linear search

$B(n) = \min\{1,2,\ldots\ldots, n\} = 1$

$= \mathbf{O(1)}$

$W(n) = \max\{1, 2, \ldots\ldots., n\} = n$

$= \mathbf{O(n)}$

Let k be the probability of x being in the array.

Successful search = $1+2+3+\ldots.+n = n(n+1)/2$

$$\text{Average} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}$$

Probability of unsuccessful search = $1 - k$

$A(n) = k * (n+1)/2 + (1-k) * n$, where $n \rightarrow$ number of unsuccessful search

Suppose x is in the array, then $k = 1$. Therefore,

$A(n) = (n+1)/2 = \mathbf{O(n)}$