1. Algorithms

An algorithm is a type of effective method in which a definite list of well-defined instructions for completing a task; that given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state. The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers.

Algorithms are named for the 9th century Persian mathematician Al-Khowarizmi. He wrote a treatise in Arabic in 825 AD, *On Calculation with Hindu Numerals*. It was translated into Latin in the 12th century as *Algoritmi de numero Indorum*, which title was likely intended to mean "[Book by] Algoritmus on the numbers of the Indians", where "Algoritmi" was the translator's rendition of the author's name in the genitive case; but people misunderstanding the title treated *Algoritmi* as a Latin plural and this led to the word "algorithm" (Latin *algorismus*) coming to mean "calculation method".

Algorithm Specification

The criteria for any set of instruction for an algorithm is as follows:

- Input : Zero of more quantities that are externally applied
- Output : At least one quantity is produced
- Definiteness : Each instruction should be clear and unambiguous
- Finiteness : Algorithm terminates after finite number of steps for all test cases.
- Effectiveness : Each instruction is basic enough for a person to carried out using a pen and paper. That means ensure not only definite but also check whether feasible or not.

Space Complexity

Space complexity of an algorithm is the amount to memory needed by the program for its completion. Space needed by a program has the following components:

1. **Instruction Space**

Space needed to store the compiled version of program. It depends on

i. Compiler used

ii. Options specified at the time of compilation

e.g., whether optimization specified, Is there any overlay option etc.

iii. Target computer

e.g., For performing floating point arithmetic, if hardware present or not.

2. **Data Space**

Space needed to store constant and variable values. It has two components:

i. Space for constants:

e.g., value '3' in program 1.1

Space for simple variables:

e.g., variables a,b,c in program 1.1

Program 1.1

```
int add (int a, int b, int c)
{
return (a+b+c)/3;
}
```

ii. Space for component variables like arrays, structures, dynamically allocated memory.

e.g., variables a in program 1.2

Program 1.2

```
    int Radd (int a[], int n)
1   {
2       If (n>0)
3           return Radd (a, n-1) + a[n-1];
4           else
5           return 0;
6   }
```

3. **Environment stack space**

Environment stack is used to store information to resume execution of partially completed functions. When a function is invoked, following data are stored in Environment stack.

i. Return address.
ii. Value of local and formal variables.
iii. Binding of all reference and constant reference parameters.

Space needed by the program can be divided into two parts.

i. Fixed part independent of instance characteristics. E.g., code space, simple variables, fixed size component variables etc.
ii. Variable part. Space for component variables with space depends on particular instance. Value of local and formal variables.

Hence we can write the space complexity as

$S(P) = c + S_p$ (instance characteristics)

Example 1.1

Refer Program 1.1

One word for variables a,b,c. No instance characteristics. Hence $S_p(TC) = 0$

Example 1.2

Program 1.3

```
int Aadd (int *a, int n)
1   {
2         int s=0;
3         for (i=0; i<n; i++)
4            s+ = a[i];
5         return s;
6   }
```

One word for variables n and i. Space for a[] is address of a[0]. Hence it requires one word. No instance characteristics. Hence $S_p(TC) = 0$

Example 1.3

Refer Program 1.2

Instance characteristics depend on values of n. Recursive stack space includes space for formal parameters, local variables and return address. So one

word each for a[],n, return address and return variables. Hence for each pass it needs 4 words. Total recursive stack space needed is 4(n).
Hence $S_p(TC) = 4(n)$.

Time Complexity

Time complexity of an algorithm is the amount of time needed by the program for its completion. Time taken is the sum of the compile time and the execution time. Compile time does not depend on instantaneous characteristics. Hence we can ignore it.

Program step: A program step is syntactically or semantically meaningful segment of a program whose execution time is independent of instantaneous characteristics. We can calculate complexity in terms of

1. Comments:
   No executables, hence step count = 0

2. Declarative Statements:
   Define or characterize variables and constants like (*int* , *long*, *enum*, …)
   Statement enabling data types (*class, struct, union, template*)
   Determine access statements (*public, private, protected, friend*)
   Character functions (*void, virtual*)
   All the above are non executables, hence step count = 0

3. Expressions and Assignment Statements:
   Simple expressions : Step count = 1. But if expressions contain function call, step count is the cost of the invoking functions. This will be large if parameters are passed as call by value, because value of the actual parameters must assigned to formal parameters.

   Assignment statements : General form is <variable> = <expr>. Step count = expr, unless size of <variable> is a function of instance characteristics. eg., a = b, where a and b are structures. In that case, Step count = size of <variable> + size of < expr >

4. Iterative Statements:

While <expr> **do**
Do .. While <expr**>**
Step count = Number of step count assignable to <expr>

For (<init-stmt>; <expr1>; <expr2>)
Step count = 1, unless the <init-stmt>, <expr1>,<expr2> are function of instance characteristics. If so, first execution of control part has step count as sum of count of <init-stmt> and <expr1>. For remaining executions, control part has step count as sum of count of <expr1> and <expr2>.

5. Switch Statements:
   Switch (<expr>) {
   Case cond1 : <statement1>
   Case cond2 : <statement2>
   .
   .
   Default : <statement>
   }

   Switch (<expr>) has step count = cost of <expr>
   Cost of **Cond** statements is its cost plus cost of all preceding statements.

6. If-else Statements:
   If (<expr>) <statement1>;
   Else <statement2>;
   Step count of **If** and **Else** is the cost of <expr>.

7. Function invocation:
   All function invocation has Step count = 1, unless it has parameters passed as called by value which depend s on instance characteristics. If so, Step count is the sum of the size of these values.
   If function being invoked is recursive, consider the local variables also.

8. Memory management Statements:
   new object, **delete** object, **sizeof**(object), Step count =1.

9. Function Statements:
   Step count = 0, cost is already assigned to invoking statements.

10. Jump Statements:
   continue, break, goto has Step count =1
   return <expr>: Step count =1, if no *expr* which is a function of instance characteristics. If there is, consider its cost also.

Example 1.4

Refer Program 1.2

Introducing a counter for each executable line we can rewrite the program as

```
int Radd (int a[], int n)
{
    count++ // if
If (n>0)
{
        count++ // return
return Radd (a, n-1) + a[n-1];
    }
     else
     {
       count++ // return
       return 0;
      }
  }
```

Case 1: n=0

$t_{Radd} = 2$

Case 2: n>0

$2 + t_{Radd}(n-1)$

$= 2 + 2 + t_{Radd}(n-2)$

$= 2 * 2 + t_{Radd}(n-2)$

.

.

$= 2n + t_{Radd}(0)$

$= 2n + 2$

Example 1.5

Program 1.4

```
        int Madd (int a[][], int b[][], int c[][], int n)
   1    {
   2        For (int i=0; i<m; i++)
   3            For (int j=0; j<n; j++)
4                       c[i][j] = a[i][j] + b[i][j];
   5    }
```

Introducing a counter for each executable line we can rewrite the program as

```
        int Madd (int a[][], int b[][], int c[][], int n)
        {
            For (int i=0; i<m; i++)
             {
               count++ //for i
               For (int j=0; j<n; j++)
               {
                       count++ //for j
                       c[i][j] = a[i][j] + b[i][j];
                       count++ //for assignment
               }
               count++ //for last j
            }
            count++ //for last i
        }
```

Step count is 2mn + 2m +1.

Step count does not reflect the complexity of statement. It is reflected in step per execution (s/e).

Refer Program 1.2

Line	**s/e**	**Frequency**		**Total Steps**	
		n=0	**n>0**	**n=0**	**n>0**
1	0	1	1	0	0
2	1	1	1	1	1
3	$1 + t_{Radd}$ (n-1)	0	1	0	$1 + t_{Radd}$ (n-1)
4	0	1	0	0	0
5	1	1	0	1	0
Total no. of steps				**2**	**$2 + t_{Radd}$ (n-1)**

Refer Program 1.3

Line	**s/e**	**Frequency**	**Total Steps**
1	0	1	0
2	1	1	1
3	1	n+1	n+1
4	1	n	N
5	1	1	1
6	0	1	0
Total no. of steps			**2n + 3**

Refer Program 1.4

Line	**s/e**	**Frequency**	**Total Steps**
1	0	1	0
2	1	m+1	m+1
3	1	m(n+1)	m(n+1)
4	1	mn	Mn
5	0	1	0
Total no. of steps			**2mn + 2m + 1**

Asymptotic Notations

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Since

the values are not exact quantities we need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$.

For example, consider two programs with complexities $c_1n^2 + c_2n$ and c_3n respectively. For small values of n, complexity depend upon values of c_1, c_2 and c_3. But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$.This value of n is called break-even point. If this point is zero, c_3n is always faster (or at least as fast). Common asymptotic functions are given below.

Function	**Name**
1	Constant
log n	Logarithmic
N	Linear
n log n	n log n
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

1.5.1 Big 'Oh' Notation (O)

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$
It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

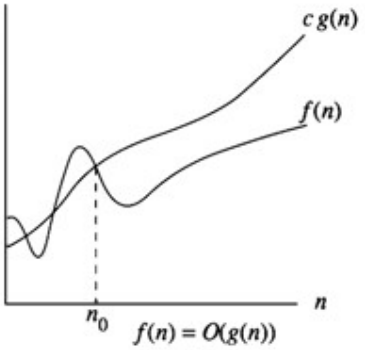


Fig 1.1

Omega Notation (Ω)

Ω (g(n)) = { f(n) : there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }

It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

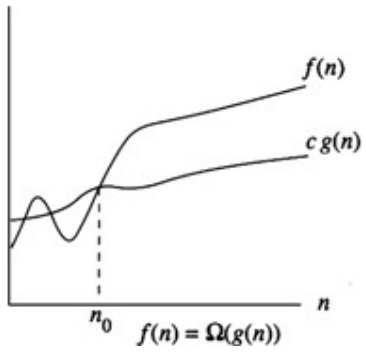


Fig 1.2

Theta Notation (Θ)

Θ(g(n)) = {f(n) : there exist 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 $n \geq n_0$ }

If f(n) = Θ(g(n)), all values of n right to n_0 f(n) lies on or above $c_1 g(n)$ and on or below $c_2 g(n)$. Hence it is asymptotic tight bound for f(n).

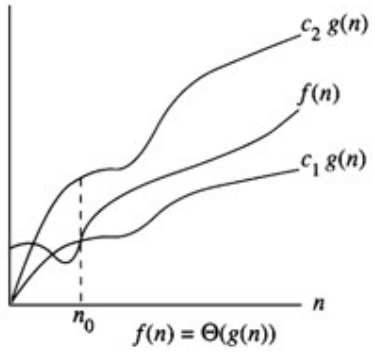


Fig 1.3

Little 'Oh' Notation (o)

o(g(n)) = { f(n) : for any positive constants c > 0, there exists n_0>0, such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$ }

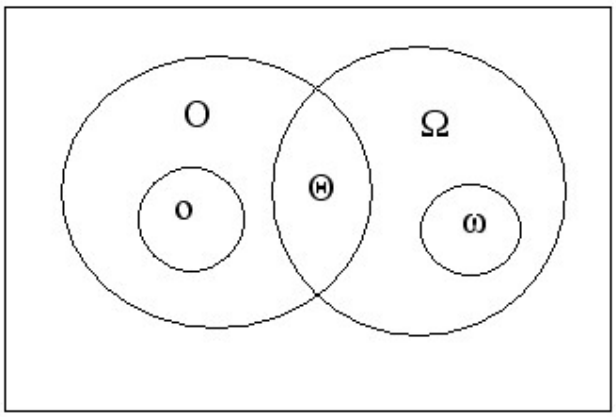
It defines the asymptotic *tight* upper bound. Main difference with Big Oh is that Big Oh defines for some constants c by Little Oh defines for all constants.

Little Omega (ω)

ω(g(n)) = { f(n) : for any positive constants c>0 and n_0>0 such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$ }

It defines the asymptotic *tight* lower bound. Main difference with Ω is that, ω defines for some constants c by ω defines for all constants.

1.1



Recurrence Relations

Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs, and one or more base cases
e.g., recurrence for Merge-Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if n} = 1 \\ 2T(n/2) + \Theta(n) & \text{if n} > 1 \end{cases}$$

- Useful for analyzing recurrent algorithms
- Make it easier to compare the complexity of two algorithms
- Methods for solving recurrences
  - Substitution method
  - Recursion tree method
  - Master method
  - Iteration method

Substitution Method

- Use mathematical induction to derive an answer
- Derive a function of n (or other variables used to express the size of the problem) that is not a recurrence so we can establish an upper and/or lower bound on the recurrence
- May get an exact solution or may just get upper or lower bounds on the solution

2. Divide and Conquer

Divide and conquer (D&C) is an important algorithm design paradigm. It works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. A divide and conquer algorithm is closely tied to a type of recurrence relation between functions of the data in question; data is "divided" into smaller portions and the result calculated thence.

Steps

- Splits the input with size n into k distinct sub problems $1 < k \leq n$
- Solve sub problems
- Combine sub problem solutions to get solution of the whole problem. Often sub problems will be of same type as main problem. Hence solutions can be expressed as recursive algorithms

Control Abstraction

Control abstraction is a procedure whose flow of control is clear but primary operations are specified by other functions whose precise meaning is undefined

```
DAndC (P)
{
    if Small(P) return S(P);
    else
    {
            divide P into smaller instances P1, P2,…,Pk k>1;
            apply DandC to each of these sub problems;
            return Combine(DandC(P1), DandC(P2),…,DandC(Pk));
    }
}
```

The recurrence relation for the algorithm is given by

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n_1) + T(n_2) + .. + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

g(n) – time to computer answer directly from small inputs.
f(n) – time for dividing P and combining solution to sub problems

2.1 Merge Sort

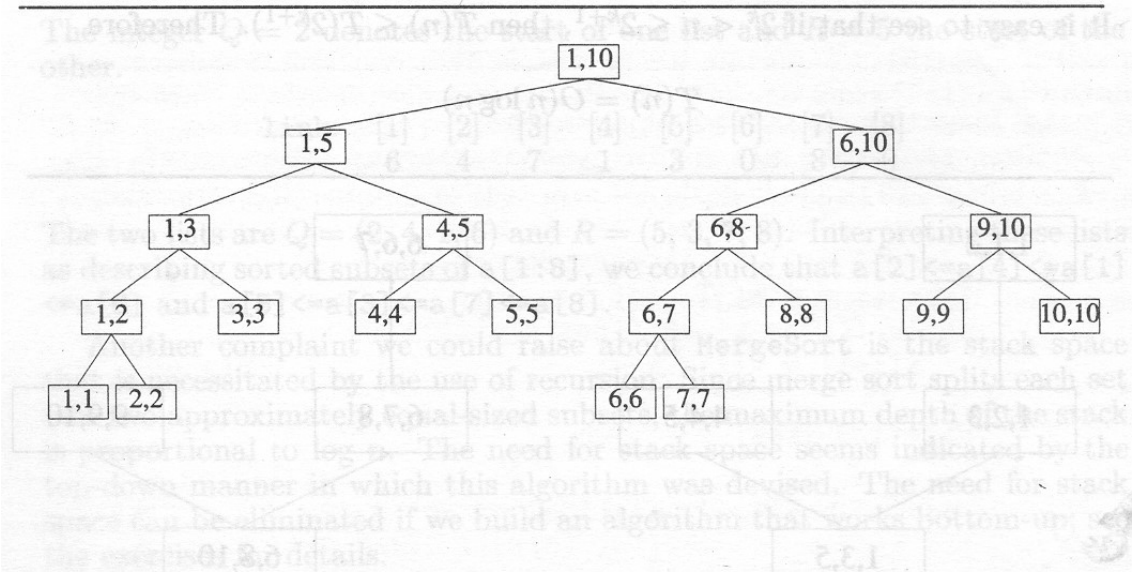
Given a sequence of n elements a[1],...,a[n].
Spilt into two sets $a[1],..,a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1],\ldots, a[n]$
Each set is individually sorted, resultant is merged to form sorted list of n elements

Program 2.3

```
Void MergeSort (int low, int high)
{
    if (low<high)                    //Small(P) is true if there is only one element, if so list is sorted
    {
            mid = (high+low)/2;
            MergeSort(low, mid);      } // Subproblems
            MergeSort(mid+1, high);   }
            Merge(low, mid, high);       // Combine
    }
}
Void Merge (int low, int mid, int high)
{
    int l=low, i = low, j = mid+1, k;
    while((l <= mid) && (j <= high))
    {
            if (a[l] <= a[j])
            {       b[i] = a[l];      h++;}
            else
            {       b[i] = a[j];      j++;}
    }
    if (l<mid)
            for(k=j; k<=high; k++)
            {       b[i] = a[k];      i++;}
    else
            for(k=h; k<=mid; k++)
            {       b[i] = a[k];      i++;}
    for(k=low; k<=high; k++)
            a[k] = b[k];
}
```

Example 2.2

Consider 10 elements 310, 285, 179, 652, 351, 423, 861, 254, 450 and 520. The recursive call can be represented using a tree as given below



T(n) = O(nlogn)

2.2 Quick Sort

Given a sequence of n elements a[1],...,a[n].
Divide the list into two sub arrays and sorted so that the sorted sub arrays need not be merged later. This is done by rearranging elements in the array such that a[i]<=a[j] for 1≤i≤m and m+1≤j≤n, 1≤m≤n. Then the elements a[1] to a[m] and a[m+1] to a[n] are independently sorted, Hence no merging is required.

Pick an element, t=a[s], reorder other elements so that all elements appearing before t is less than or equal to t, all elements after t is greater than or equal to t.

Program 2.4

```
Void QuickSort (int p, int q)
{
    if (p<q)          // more than one element, divide into sub problems
    {
            j = partition(a,p,q+1)
            QuickSort(p, j-1) ;
            QuickSort(j+1, q) ;   } Combine
    // No need of combining solutions
    }
}
int partition (int a[], int m, int p)
{
    int v=a[m], i=m, j=p ;
    do
    {
            do
                    i++;
            while(a[i]<v);
            do
                    j--;
            while(a[j]>v);
            if (i<j) interchange(a,i,j);
    }
    while(i<j);
    a[m] = a[j];     a[j] = v;          return(j);
}
```

While calculating complexity we are considering element comparison alone. Assume that n elements are distinct and the distribution is such that there is equal probability for any element to get selected for partitioning.