Chapter-1
Introduction

1. 1 what is Algorithm?

It is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. It is thus a sequence of computational steps that transform the input into the output. It is a tool for solving a well - specified computational problem.

Algorithm must have the following criteria:

Input: Zero or more quantities is supplied

Output: At least one quantity is produced.

Definiteness: Each instruction is clear and unambiguous.

Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

Effectiveness: Every instruction must be basic so that it can be carried out.

1.2 What is program? Why DAA?

A program is the expression of an algorithm in a programming language. A set of instructions which the computer will follow to solve a problem.
It is learning general approaches to algorithm design.

Divide and conquer
Greedy method
Dynamic Programming
Basic Search and Traversal Technique
Graph Theory
Branch and Bound
NP Problems

1.3 Why do Analyze Algorithms?

To examine methods of analyzing algorithm
Correctness and efficiency
- **-Recursion equations**
- **-Lower bound techniques**
- **-O,Omega and Theta notations for best/worst/average case analysis**

Decide whether some problems have no solution in reasonable time
- **-List all permutations of n objects (takes n! steps)**
- **-Travelling salesman problem**

Investigate memory usage as a different measure of efficiency.

1.4 Importance of Analyzing Algorithms

Need to recognize limitations of various algorithms for solving a problem. Need to understand relationship between problem size and running timeWhen is a running program not good enough? Need to learn how to analyze an algorithm's running time without coding it. Need to learn techniques for writing more efficient code. Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize.

1.4.1 The Selection Problem

Problem: given a group of n numbers, determine the k^{th} largest

Algorithm 1

Store numbers in an array
Sort the array in descending order
Return the number in position k

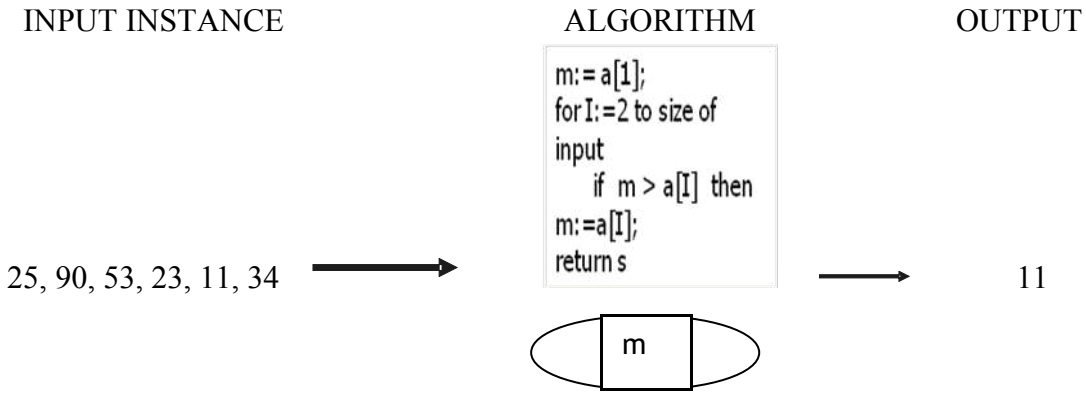
Algorithm 2

Store first k numbers in an array
Sort the array in descending order
For each remaining number, if the number is larger than the k^{th} number, insert the number in the correct position of the array
Return the number in position k

Example

Input is a sequence of integers stored in an array.

Output the minimum.



Problem: Description of Input-Output relationship.

Algorithm: A sequence of computational step that transform the input into the output.

Data Structure: An organized method of storing and retrieving data.

Our task: Given a problem, design a *correct* and *good* algorithm that solves it.

Example Algorithm A

Problem: The input is a sequence of integers stored in array.
Output the minimum.

Algorithm:

$m \leftarrow a[1]$;
For $i \leftarrow 2$ to size of input;
    if $m > a[i]$ then $m \leftarrow a[i]$;
output m.

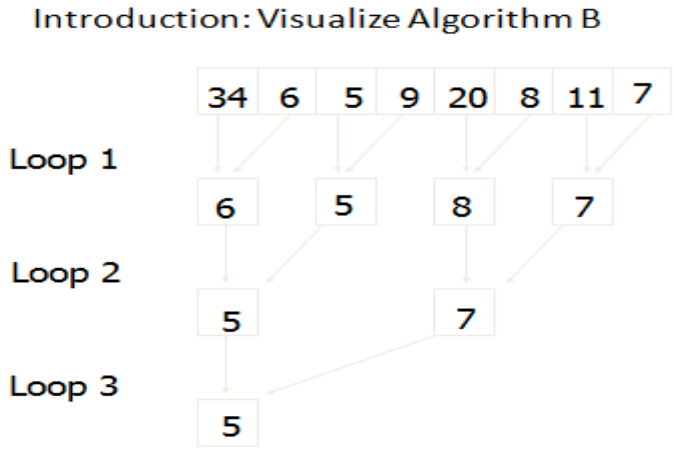
Example Algorithm B

This algorithm uses two temporary arrays.

1. copy the input *a* to array *t1;*
   assign *n* ←size of input;
2. While *n* > 1
   For *i* ←1 to *n* /2
   t2[*i*] ← min (*t1* [2**i*], *t1*[2**i* + 1]);
   copy array *t2* to *t1*;
   n ←*n/2;*
3. Output *t2*[1];

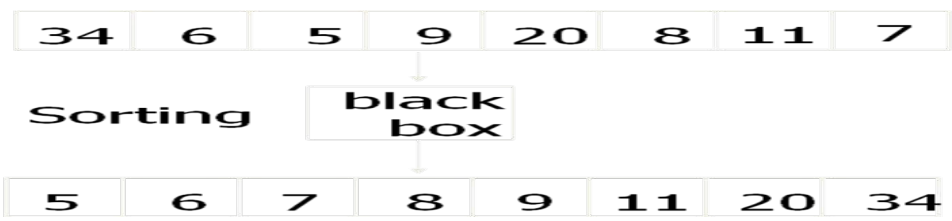
Visualize Algorithm B

34	6	5	9	20	8	11	7



Example Algorithm C

Sort the input in increasing order. Return the first element of the sorted data.

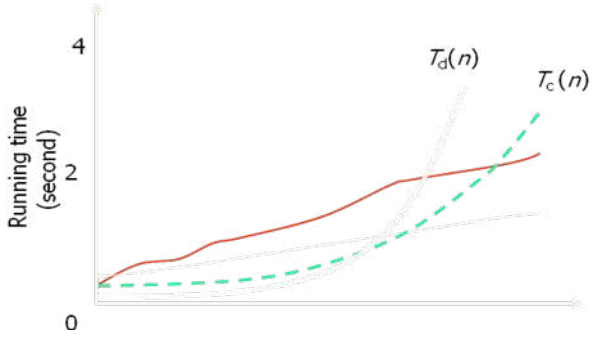


Introduction: Example Algorithm D
For each element, test whether it is the minimum.

```
1.  i ← 0;
    flag ← true;
2.  While flag
        i ← i + 1;
        min ← a[i];
        flag ← false;
        for j ← 1 to size of input
            if min > a[j] then flag ← true;
3.  Output min.
```

1.5 Time vs. Size of Input

Measurement parameterized by the size of the input. The algorihtms A,B,C are *implemented* and run in a PC. Algorithms D is implemented and run in a supercomputer. Let $T_k(n)$ be the amount of time taken by the Algorithm.



1.5.1 Methods of Proof

(a) Proof by Contradiction

Assume a theorem is false; show that this assumption implies a property known to be true is false --therefore original hypothesis must be true

(b) Proof by Counter example

Use a concrete example to show an inequality cannot hold. Mathematical Induction. Prove a trivial base case, assume true for k, and then show hypothesis is true for k+. Used to prove recursive algorithms

(c) Proof by Induction

Claim: S (n) is true for all n >= k

Basis: Show formula is true when n = k

Inductive hypothesis: Assume formula is true for an arbitrary n

Step: Show that formula is then true for n+1

Examples

Gaussian Closed Form

Prove $1 + 2 + 3 + \ldots + n = n (n+1) / 2$
Basis: If n = 0, then $0 = 0 (0+1) / 2$
Inductive hypothesis: Assume $1 + 2 + 3 + \ldots + n = n (n+1) / 2$
Step (show true for n+1): $1 + 2 + \ldots + n + n+1 = (1 + 2 + \ldots + n) + (n+1)$
$= n (n+1)/2 + n+1 = [n (n+1) + 2(n+1)]/2$
$= (n+1) (n+2)/2 = (n+1) (n+1 + 1) / 2$

Geometric Closed Form

Prove $a_0 + a_1 + \ldots + a_n = (a_n+1-1) / (a -1)$ for all a 1
Basis: show that $a_0 = (a_0+1-1) / (a -1)$
$a_0 = 1 = (a1-1) / (a -1)$
Inductive hypothesis: Assume $a_0 + a_1 + \ldots + an = (an+1-1) / (a -1)$
Step (show true for n+1): $a_0 + a_1 + \ldots + a_n+1 = a_0 + a_1 + \ldots + a_n + a_n+1$
$= (a_n+1-1) / (a -1) + a_n+1 = (a_n+1+1-1) / (a -1)$

Strong induction also holds

Basis: show S (0)

Hypothesis: assume S (k) holds for arbitrary k <= n

Step: Show S (n+1) follows

Another variation

Basis: show S (0), S (1)

Hypothesis: assume S (n) and S (n+1) are true

Step: show S (n+2) follows.

1.6 Basic Recursion

Base case: value for which function can be evaluated without recursion
Two fundamental rules:-
1. Must always have a base case
2. Each recursive call must be to a case that eventually leads toward a base case

Problem: Write an algorithm that will strip digits from an integer and print them out one by one

```
void print_out(int n)
if(n < print_digit(n); /*outputs single-digit to terminal*/
else
print_out (n/); /*print the quotient*/
print_digit (n % ); /*print the remainder*/
```

Prove by induction that the recursive printing program works:
Basis: If n has one digit, then program is correct.

hypothesis: Print_out works for all numbers of k or fewer digits

case k+: k+ digits can be written as the first k digits followed by the least significant digit

The number expressed by the first k digits is exactly floor (n /)? Which by hypothesis prints correctly; the last digit is n%; so the (k+)-digit is printed correctly. By induction, all numbers are correctly printed.

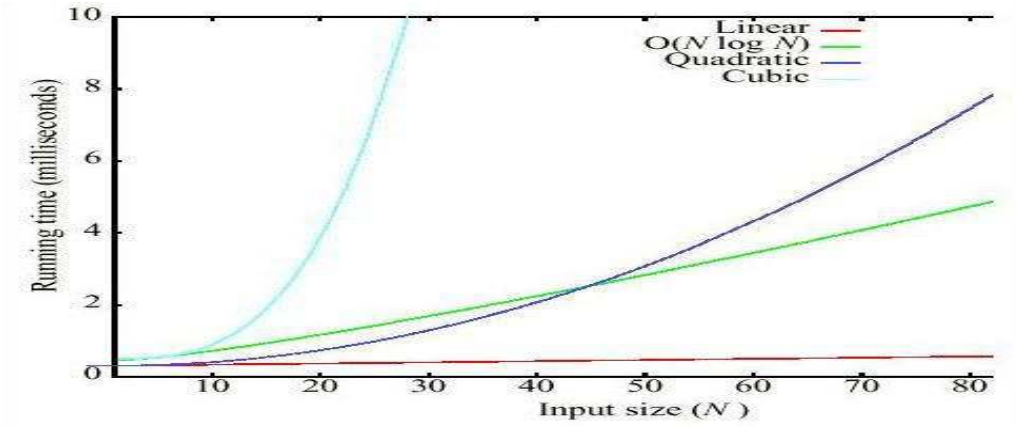
Recursion is expensive in terms of space requirement; avoid recursion if simple loop will do
Last two rules
Assume all recursive calls work
Do not duplicate work by solving identical problem in separated recursive calls
Evaluate fib () --use a recursion tree

Fib (n) = fib (n-1) + fib (n-2)

1.7 Algorithm Analysis and Running Time

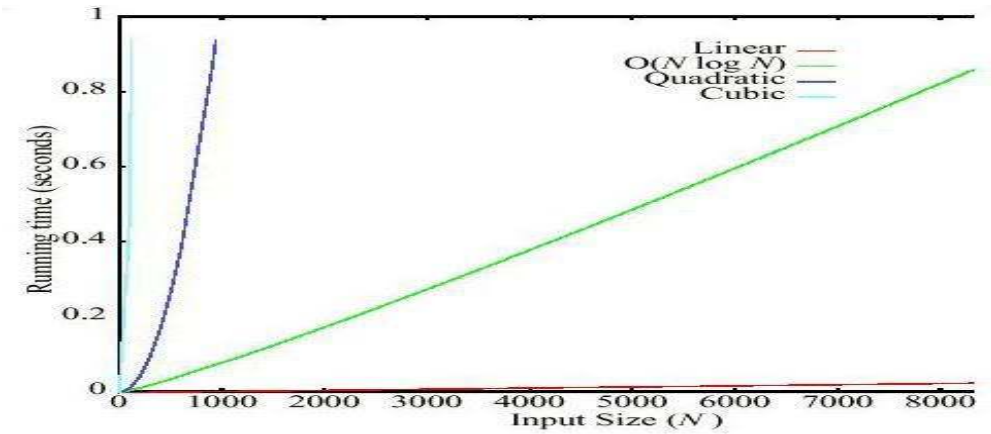
How to estimate the time required for an algorithm. Techniques that drastically reduce the running time of an algorithm. A mathematical framework that more rigorously describes the running time of an algorithm.

Running time for small inputs



Running times for small inputs

Running time for moderate inputs



Running time for moderate inputs

Algorithm Analysis

Measures the efficiency of an algorithm or its implementation as a program as the input size becomes very large. We evaluate a new algorithm by comparing its performance with that of previous approaches. Comparisons are asymtotic analyses of classes of algorithms. We usually analyze the time required for an algorithm and the space required for a data structure.

Many criteria affect the running time of an algorithm, including

-speed of CPU, bus and peripheral hardware
-design think time, programming time and debugging time
-language used and coding efficiency of the programmer
-quality of input (good, bad or average)
-Machine independent
-Language independent
-Environment independent (load on the system)
-Amenable to mathematical study
-Realistic

In lieu of some standard benchmark conditions under which two programs can be run, we estimate the algorithm's performance based on the number of key and basic operations it requires to process an input of a given size. For a given input size n we express the time T to run the algorithm as a function T (n). Concept of growth rate allows us to compare running time of two algorithms without writing two programs and running them on the same compute. Formally, let T (A, L, M) be total run time for algorithm A if it were implemented with language L on machine M. Then the complexity class of algorithm A is O (T(A, L, M) U O(T(A, L, M)) U O(T(A, L, M)) .

Call the complexity class V; then the complexity of A is said to be f if V = O (f).The class of algorithms to which A belongs is said to be of at most linear/quadratic/ etc. The growth in best case if the function TA best(n) is such (the same also for average and worst case).

1.8 Asymptotic Performance

Asymptotic performance means it always concerns with how does the algorithm behave as the problem size gets very large? Running time, Memory/storage requirements, and Band width/power requirements/logic gates/etc.

Asymptotic Notation

By now you should have an intuitive feel for asymptotic (big-O) notation:
What does O (n) running time mean? O (n2)? O (n log n).?
How does asymptotic running time relate to asymptotic memory usage?.
Our first task is to define this notation more formally and completely.

Analysis of Algorithms

Analysis is performed with respect to a computational model we will usually use a generic uni processor random-access machine (RAM).

All memory equally expensive to access.

No concurrent operations.

All reasonable instructions take unit time.

Ex: Except, of course, function calls

Constant word size

Ex: Unless we are explicitly manipulating bits

Input Size

Time and space complexity. This is generally a function of the input size.
E.g., sorting, multiplication
How we characterize input size depends:
Sorting: number of input items

Multiplication: total number of bits

Graph algorithms: number of nodes & edges

Running Time

Number of primitive steps that are executed. Except for time of executing a function call most statements roughly require the same amount of time.

y = m * x + b

c = 5 / 9 * (t -32)

z = f(x) + g(y)

Analysis

Worst case provides an upper bound on running time. An absolute guarantee
Average case provides the expected running time random (equally likely) inputs.

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

1.9 Space Complexity (S (P)=C+SP(I))

Fixed Space Requirements (C) Independent of the characteristics of the inputs and outputs instruction space. Space for simple variables, fixed-size structured variable, constants. Variable Space Requirements (SP(I))depend on the instance characteristic I–number, size, values of inputs and outputs associated with recursive stack space, formal parameters, local variables, return address.

Program: Simple arithmetic function

```
float abc (float a, float b, float c)
{
return a + b + b * c + (a + b -c) / (a + b) + 4.00;
}
```

Program: Iterative function for summing a list of numbers.

```
float sum(float list[ ], int n)
{
float tempsum = 0;
int i;
for (i = 0; i<n; i++)
tempsum += list [i];return tempsum;
}
```

Sabc(I) = 0
Ssum(I) = 0

Recall: pass the address of the first element of the array & pass by value.

Program: Recursive function for summing a list of numbers.

```
 float rsum(float list[ ], int n)
 {
 if (n) return rsum(list, n-1) + list[n-1];
return 0;
 }
```

Ssum(I)=Ssum(n)=6n

Assumptions

Space needed for one recursive call of

Type	Name	Number of bytes
parameter: float parameter: integer return address:(used internally)	List[n] N	2 2 2
TOTAL per recursive call		6

1.10 Time Complexity

- Compile time (C) : Independent of instance characteristics.
- Run (execution) time TP

Definition: TP (n) = caADD(n) + csSUB(n) + clLDA(n) + cstSTA(n)
A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

abc = a + b + b * c + (a + b -c) / (a + b) + 4.0

abc = a + b + c

Methods to compute the step count

- Introduce variable count into programs
- Tabular method

Determine the total number of steps contributed by each statement step per execution × frequency

add up the contribution of all statements

Program: Iterative summing of a list of numbers:

```
float sum(float list[ ], int n)
{
float tempsum = 0;
count++; /* for assignment */
int i;
for (i = 0; i < n; i++)
{
count++; /*for the for loop */
tempsum += list[i];
count++; /* for assignment */}
count++; /* last execution of for */
return tempsum;
count++; /* for return */
}
```

2n + 3 steps

Program: Simplified version of Program

```
float sum(float list[ ], int n)
{
```

```
float tempsum = 0;
int i;
for (i = 0; i < n; i++)
count += 2;
count += 3;
return 0;
}
```

2n + 3 steps

Program : Recursive summing of a list of numbers

```
float rsum(float list[ ], int n)
{
count++; /*for if conditional */
if (n)
{
count++; /* for return and rsum invocation */
return rsum(list, n-1) + list[n-1];
}
count++;
return list[0];
}
```

2n+2 times

Program : Matrix addition

```
void add( int a[ ] [MAX_SIZE], int b[ ] [MAX_SIZE],int c [ ] [MAX_SIZE], int rows, int
cols)
{
int i, j;
for (i = 0; i < rows; i++)
for (j= 0; j < cols; j++)
c[i][j] = a[i][j] +b[i][j];
}
```

Matrix addition with count statements:

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],int c[ ][MAX_SIZE], int row, int cols )
{
int i, j;
for (i = 0; i < rows; i++)                    (2rows * cols + 2 rows + 1)
{
count++; /* for i for loop */
for (j = 0; j < cols; j++)
 {
count++; /* for j for loop */
c[i][j] = a[i][j] + b[i][j];
count++; /* for assignment statement */}
count++; /* last time of j for loop */
```

```
}
count++; /* last time of i for loop */
}
```

Program: Simplification of Program

```
void add(int a[ ][MAX_SIZE], int b [ ][MAX_SIZE],int c[ ][MAX_SIZE], int rows, int cols)
{
int i, j;for( i = 0; i < rows; i++)
{
for (j = 0; j < cols; j++)
count += 2;
count += 2;
 }
count++;
}
```

2rows cols + 2rows +1 times

Tabular Method

Step count table
Iterative function to sum a list of numbers

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
TOTAL			2n+3

Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
TOTAL			2n+3

Matrix Addition

Step count table for matrix addition

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE])	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows (cols+1)	rows cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows cols	rows cols
}	0	0	0
TOTAL		2 rows cols+2rows+1	

1.11 Asymptotic Notation (Q, *O*, W, *o*, w)

Defined for functions over the natural numbers.
Ex: $f(n) = Q(n_2)$.

Describes how $f(n)$ grows in comparison to $n2$. Define a *set* of functions; in practice used to compare two function sizes. The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

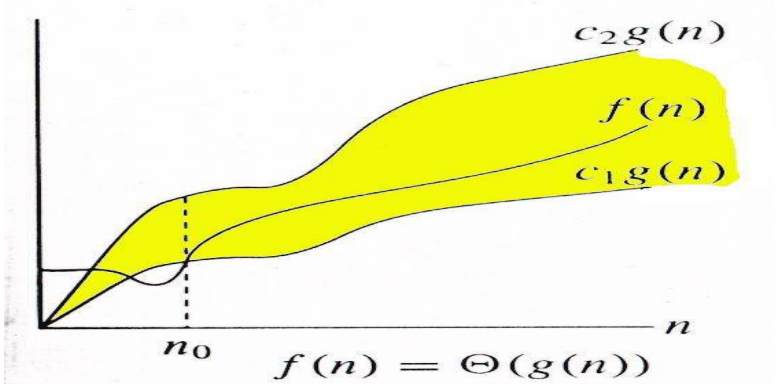
(a) Θ notation

For function $g(n)$, we define $(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{ f(n) : \exists$ positive constants c_1, c_2, and n_0,such that $\forall n >= n_0$, we have $0 <= c_1 g(n) <= f(n) <= c_2 g(n) \}$

Intuitively:

Set of all functions that have the same rate of growth as g (n). g (n) is an asymptotically tight bound for f(n).



For function $g(n)$, we define $(g(n))$, big-Theta of n, as the set:

$\Theta(g(n)) = \{f(n)$: positive constants c_1, c_2, and n_0,such that $\forall\ n >= n_0$,
we have $0 <= c_1 g(n) < f(n) <= c_2 g(n)\}$

$F(n)$ and $g(n)$ are nonnegative, for large n.

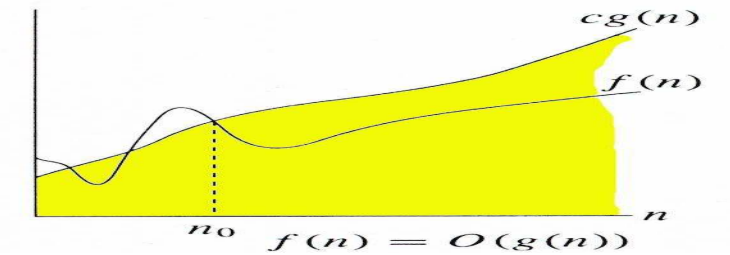
$3n+2 = \Theta(n)$ as $3n+2>=3n$ for all $n>=2$ and $3n+2<=4n$ for all $n>=2$
So $c_1=3$ and $c_2=4$and $n_0=2$. So, $3n+3= \Theta(n)$,
$10n^2+4n+2= \Theta(n^2)$, $6*2n+n^2= \Theta(2^n)$ and
$10*\log n+4= \Theta(\log n)$.$3n+2\# \Theta(1)$,
$3n+3\# \Theta(n2)$, $10n^2+4n+2\# \Theta(n)$, $10n^2+4n+2\# \Theta(1)$

***(b)O*-notation**

For function g (n), we define O (g(n)), big-O of n, as the set:

$O(g(n)) = \{f(n)$: positive constants c and n_0,such that $n>= n_0$, we have$0<= f(n)<=cg(n)\}$

Intuitively: Set of all functions whose rate of grow this same as or lower than that of g (n).
G (n) is an asymptotic upper bound for f(n).
$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n)).\Theta(g(n)) \subset O(g(n))$.



Example- 1

7n-2
7n-2 is O (n)
need c > 0 and n_{01} such that7n-2 c• n for n n0
this is true for c = 7 and n0= 1

Example - 2

$3n^3+ 20n^2+ 5$
$3n^3+ 20n^2+ 5$ is O (n^3)
need c > 0 and n01 such that$3n^3+ 20n^2+ 5$ c•$n^3$for n n0
this is true for c = 4 and n0= 21

Example: 3
3 log n + log log n
3 log n + log log n is O (log n)
need c > 0 and n01 such that3 log n + log log n c• log n for n n0
this is true for c = 4 and n0= 2

Big-Oh and Growth Rate

The big-Oh notation gives an upper bound on the growth rate of a function. The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$. We can use the big-Oh notation to rank functions according to their growth rate.

	f(n)) is O (g(n))	g(n) is $O(f(n))$
g(n) grows more	Yes	No
f(n) grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

If is $f(n)$a polynomial of degree d, then $f(n)$is $O(nd)$, i.e.,
1. Drop lower-order terms
2. Drop constant factors

Use the smallest possible class of functions Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$". Use the simplest expression of the class. Say "$3n+5$ is $O(n)$" instead of "$3n+5$ is $O(3n)$"

Relatives of Big-Oh

Big-Omega
F (n) is Ω (g(n)) if there is a constant c > 0 and an integer constant $n_{0>=}1$ such that f(n)>= c• g(n) for n>= n0

Big-Theta
f(n) is Θ (g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant n_0>1 such that c'• g(n)<= f(n)<= c''• g(n) for n>= n_0

Little-oh
F (n) is o (g(n)) if, for any constant c > 0, there is an integer constant n_0> 0 such that f(n) < c• g(n) for n>= n_0

Little-omega
f(n) is ω (g(n)) if, for any constant c > 0, there is an integer constant n_0> 0 such that f(n) > c• g(n) for n>= n_0

Intuition for Asymptotic Notation Big-Oh

f (n) is O (g(n)) if f(n) is asymptotically less than or equal to g(n)

Big-Omega
f (n) is Ω (g (n)) if f(n) is asymptotically greater than or equal to g(n)

Big-Theta
f (n) is Θ (g (n)) if f(n) is asymptotically equal to g(n)

Little-oh
f (n) is o (g (n)) if f(n) is asymptotically strictly less than g(n)

Little-omega
f (n) is ω (g (n)) if is asymptotically strictly greater than g(n)

Examples

$5n^2\Omega$ is (n^2)
F (n) is Ω (g(n)) if there is a constant c > 0 and an integer constant n_0>=1 such that f(n)>= c• g(n) for n>= n_0 let c = 5 and n_0= 1

$5n^2\Omega$ is (n)
f(n) is Ω (g(n)) if there is a constant c > 0 and an integer constant n_0>=1 such that f(n) >=c• g(n) for n>= n_0 let c = 1 and n_0= 1

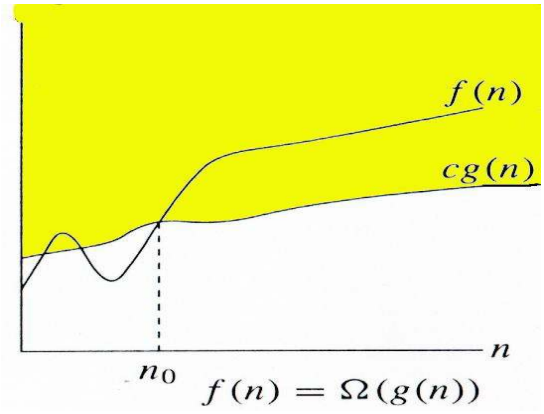
5 n^2 ω (n)
F (n) is ω (g(n)) if, for any constant c > 0, there is an integer constant n_0> 0 such that f(n) > c• g(n) for n >=n_0 need $5n_0^2$> c•n0given c, the $n_0$that satisfies this is n_0> c/5 > 0.

(c) Ω **Notation**

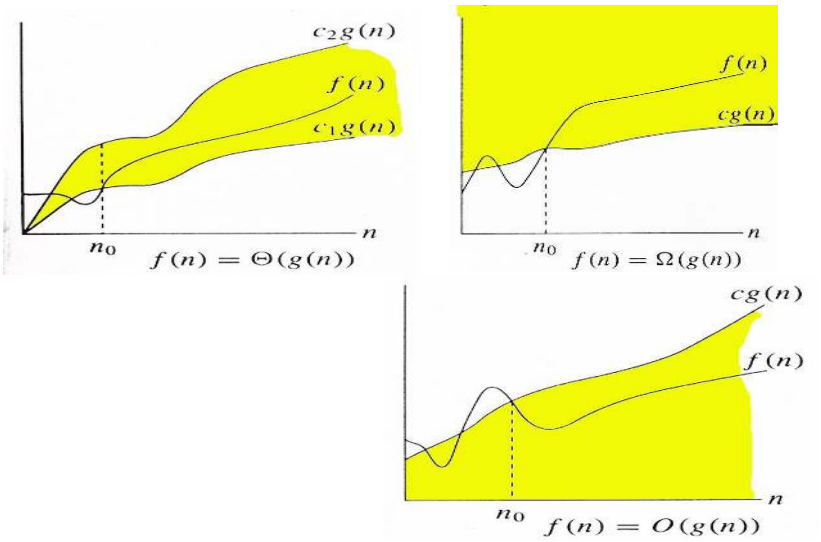
For function *g* (*n*), we define Ω (*g*(*n*)), big-Omega of *n*, as the set:
Ω (*g* (*n*)) ={ *f*(*n*) :∃ positive constants *c* and n_0,such that ∀*n* >=n_0,we have0 <=c*g*(*n*)<=*f*(*n*) }

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of *g*(*n*). *g*(*n*) is an *asymptotic lower bound* for *f*(*n*).

$$f(n) = \Theta(g(n)) \; f(n) \Rightarrow f(n) = \Omega(g(n)) \; \Theta(g(n)) \subset \Omega(g(n))$$



Relations Between Θ , *O*, Ω



Theorem : For any two functions $g(n)$ and $f(n)$, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

i.e., $\Theta(g(n)) = O(g(n))\ (g(n))$
Asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Running Times

"Running time is $O(f(n))$" Worst case is $O(f(n))$
$O(f(n))$ bound on the worst-case running time $\Rightarrow O(f(n))$ bound on the running time of every input.
$\Theta(f(n))$ bound on the worst-case running time $\Theta(f(n))$ bound on the running time of every input.
"Running time is $(f(n))$" Best case is $(f(n))$ Can still say "Worst-case running time is $\Omega(f(n))$". Means worst-case running time is given by some unspecified function $g(n) \in \Omega(f(n))$.

Asymptotic Notation in Equations

We can use asymptotic notation in equations to replace expressions containing lower-order terms.

Example: $4n_3 + 3n_2 + 2n + 1 = 4n_3 + 3n_2 + (n)$
$= 4n_3 + (n_2) = (n_3)$.
$\Theta(f(n))$ always stands for an *anonymous function* $g(n) \in \Theta(f(n))$

***Little o*-notation**
For a given function $g(n)$, the set little-o: $\forall o(g(n)) = \{f(n): \forall c > 0, \exists\ n_0 > 0$ such that $\forall n >= n_0$, we have $0 <= f(n) < cg(n)\}$.
$F(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:
$lim\ [f(n) / g(n)] = 0$
$n \to \alpha$
$g(n)$ is an *upper bound* for $f(n)$ that is not asymptotically tight.

Little ω –notation

For a given function $g(n)$, the set little-omega: $(g(n))= \{f(n): \forall c> 0, \exists\ n0> 0$such that $\forall n >= n_0$, we have $0 <= cg(n) < f(n) \}$.
$F(n)$ becomes arbitrarily large relative to $g(n)$as n approaches infinity:
$lim\ [f(n) / g(n)] = \alpha$.
$n-> \alpha$
$g(n)$ is a *lower bound* for $f(n)$that is not asymptotically tight.

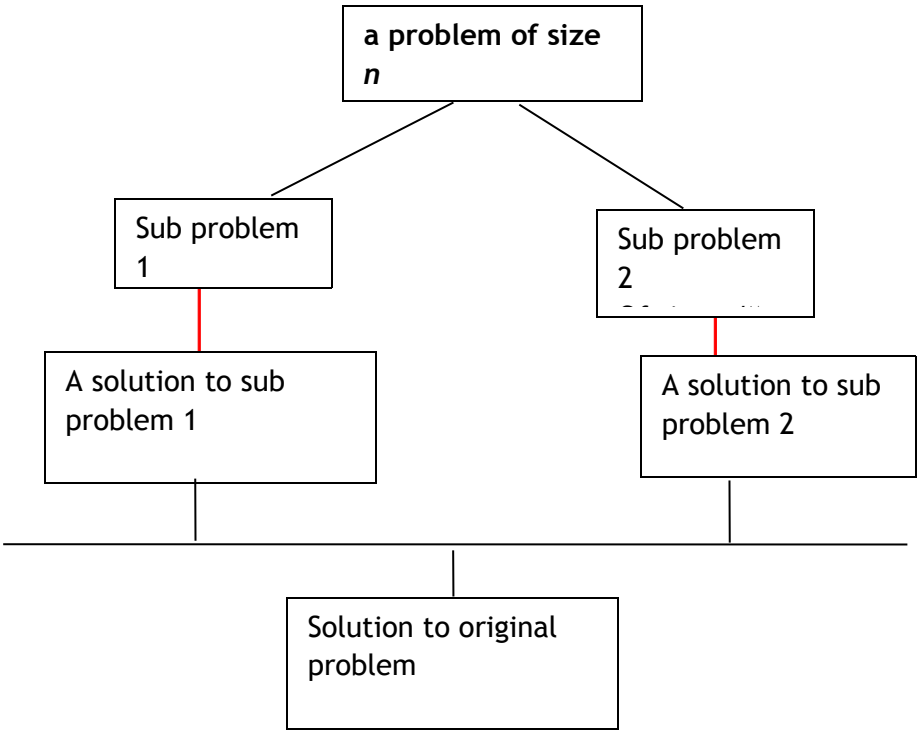
……..

Chapter-2
Divide and Conquer

2.1 General Method

Definition

Divide the problem into a number of sub problems; conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.



Divide-and conquer is a general algorithm design paradigm

Divide: divide the input data S in two or more disjoint subsets S_1, S_2,
Recursively: solve the sub problems recursively
Conquer: combine the solutions for S_1, $S_{2...}$ into a solution for S
The base case for the recursion is sub problems of constant size. Analysis can be done using recurrence equations