Properties of an algorithm

- It is written in simple English.
- An algorithm must have at least one input.
- An algorithm must have at least one output.
- Finiteness:

An algorithm has finite number of steps. That is, the algorithm must come to an end after a specific number of steps.

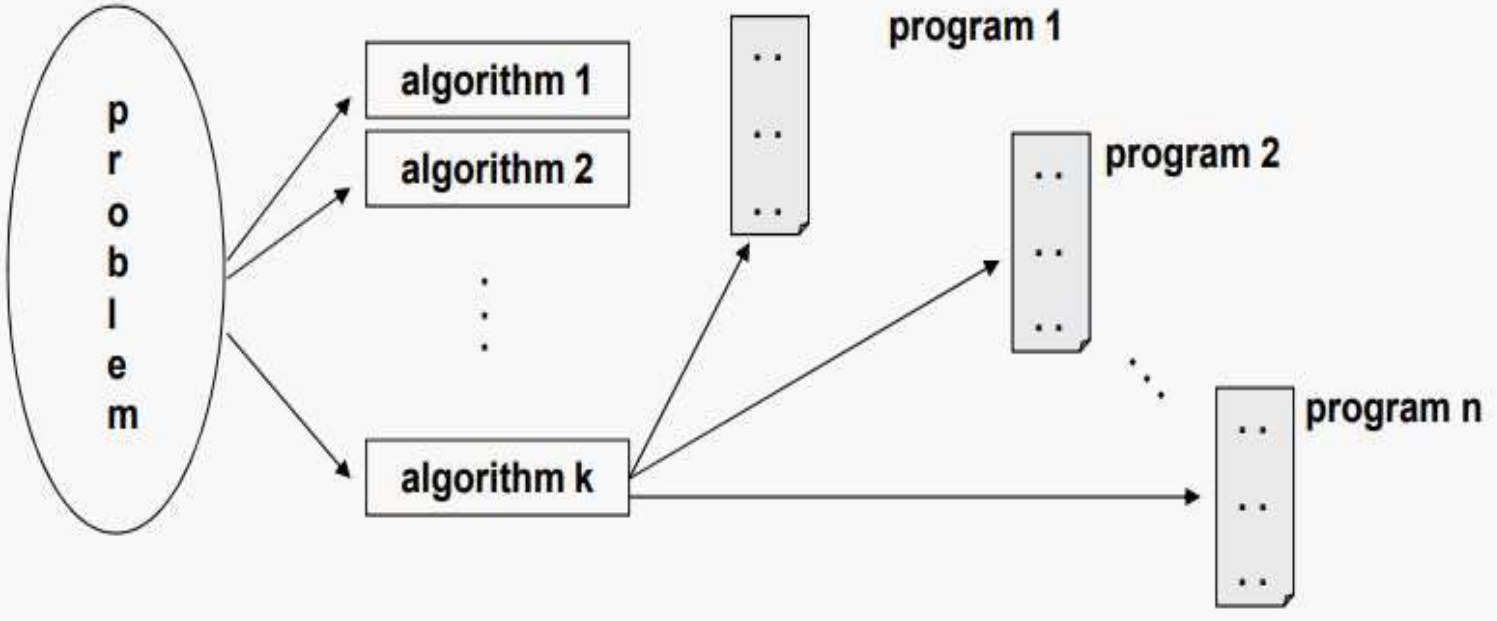
- Definiteness: - each instructions should be clear and unambiguous.

  This means that the action specified by the step cannot be interpreted in multiple ways & can be performed without any confusion.

Problems vs Algorithms vs Programs

For each problem or class of problems, there may be many different algorithms.

For each algorithm, there may be many different implementations (programs).



Types of algorithms:[Algorithm Design Technique]

- Simple recursive algorithms.
- Backtracking algorithms
- Divide and conquer algorithms.
- Dynamic programming algorithms.
- Greedy algorithms
- Branch and bound algorithms.
- Brute force algorithms.
- Randomized algorithms.

Analysis of Algorithm

Importance of Analyze Algorithm

-Need to understand limitations of various Algorithms for solving a problem.

-Need to understand relationship between problem size and running time.

-Need to learn how to analyze an Algorithm’s running time without coding it.

-Need to learn technique for writing more efficient code.

The overall goal of Analysis of Algorithm is on understanding of the complexity of Algorithm.

Complexity of Algorithm Is simply the amount of work the algorithm perform to complete its task.

or

A measure of the Analysis of Algorithms or performance of Algorithms.

There are mainly two types of complexity

1.Space complexity

2.Time complexity

Space complexity

of an algorithm is the amount of memory that it needs to run to completion.

The way in which the amount of storage space required by on algorithm varies with the size of the problem it is solving.

The amount of time required by on algorithm varies with the size of the problem.

as the complexity (space & time) grows, the size of the problem also grows.

Types of analysis of an algorithm

There are 2 types of analysiss of algorithm.

1.Priori analysis of algorithm

2.Posteriori analysis of algorithm

1.Priori analysis of algorithm

Priori analysis means doing an analysis of an algorithm(time and space) before running it on the system.

The analysis is done on the basis of different operations such as assignment, comparison, iterations, function call etc which are performed by the programming code.

2.Posteriori analysis of algorithm

In Posteriori analysis of algorithm, the analysis of an algorithm is done only after running it on the system. So this analysis depends highly on the system's performance and hence it changes from system to system.

The performance of the software may vary from system to system, so efficiency of an algorithm will not be same for all end users. Therefore, posteriori analysis is not very useful for software industry.

Order of Growth

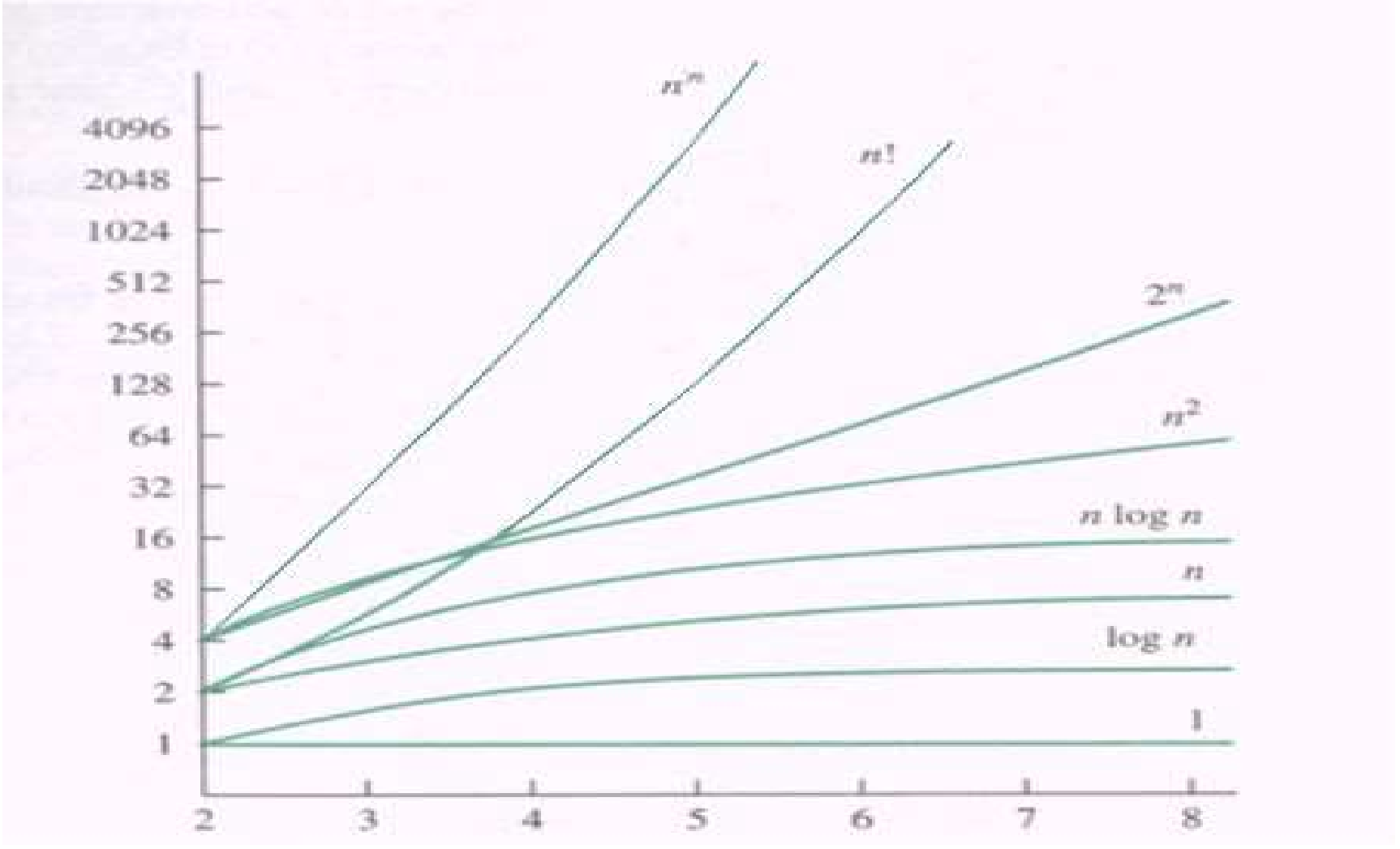
Time complexity of an algorithm is expressed as a function of the size of the input data of the problem.

Measuring the efficiency of an algorithm in relation with the input size of the problem is called order of growth.

If the running time of two algorithms have the same growth rate, then their performance is said to be the same .

If the running time of two algorithms have different growth rate, then their performance significantly changes as the problem's input size changes.

The order of growth for varying problem input sizes is shown below.



Asymptotic Notation

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

Big - Oh (O)

Big - Omega (Ω)

Big - Theta (Θ)

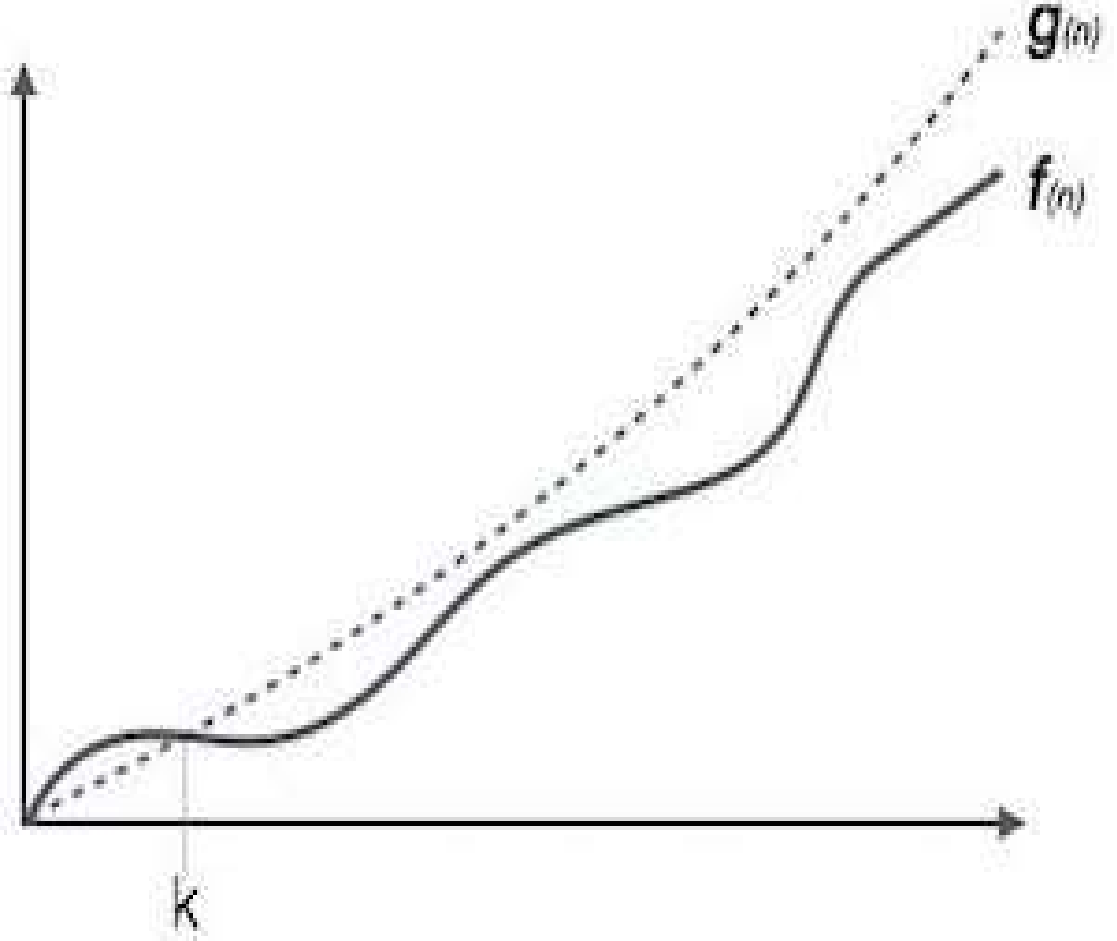
Big - Oh Notation (O)

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.

That means, Big - Oh notation always indicates the maximum time required by an algorithm for all input values.

That means Big - Oh notation describes the worst case of an algorithm time complexity.

Definition 1.4 [Big "oh"] The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq c * g(n)$ for all n, $n \geq n_0$.



Consider f(n)=2n + 5

g(n)=n

f(n)<= c.g(n)

that is:

2n + 5 <=c.n

Assume c=3

2n + 5 <=3.n

For n=1 ==>, 7<=3 F

For n=2 ==>,9<=6 F

For n=3 ==>, 11<=9 F

For n=4 ==>, 13<=12 F

For n=5 ==>,15<=15 T

Therefore

f(n)<=O(g(n), for all n>=5 and c=3.

Big - Omege Notation (Ω)

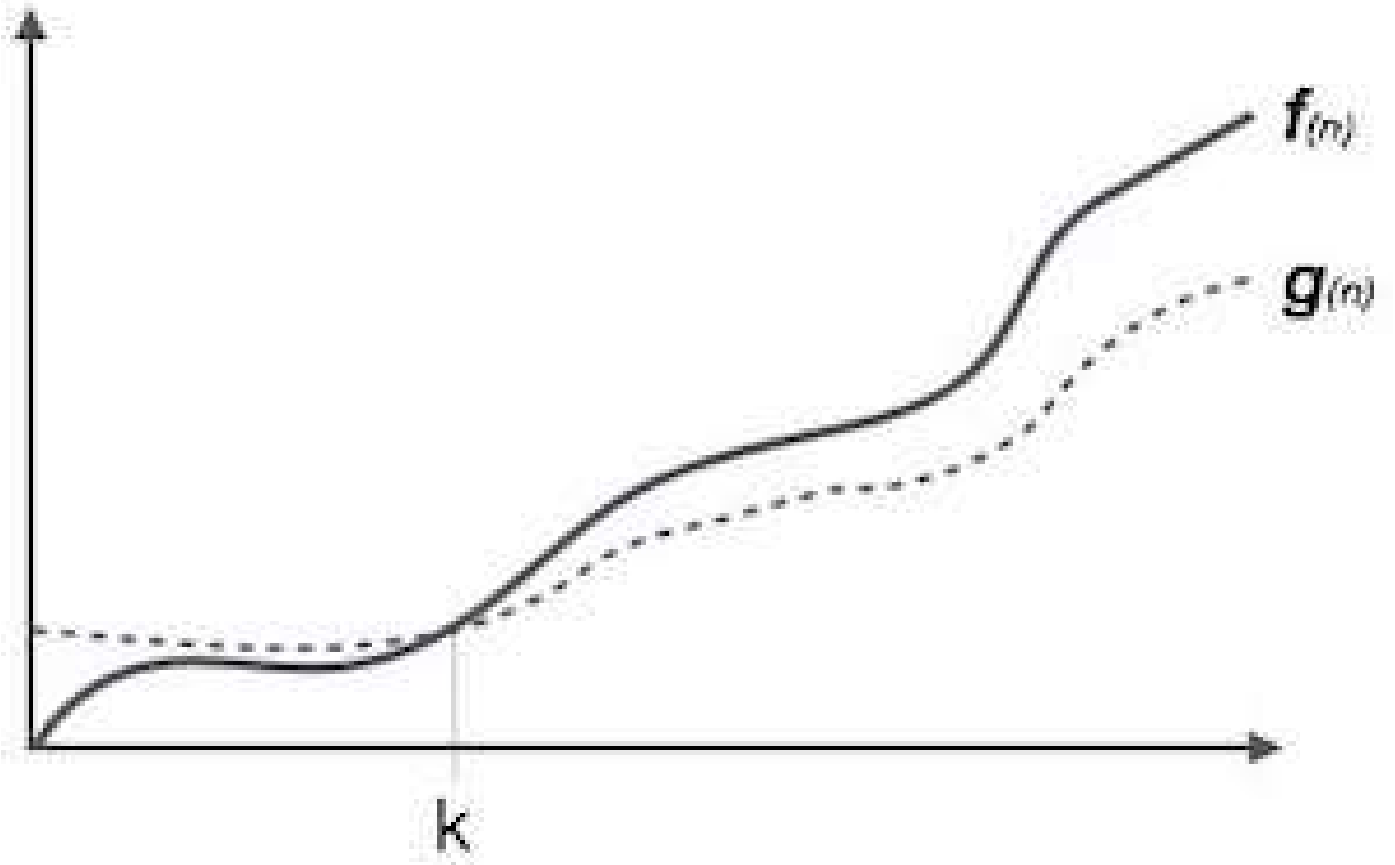
Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.

That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values.

That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Definition 1.5 [Omega] The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq c * g(n)$ for all n, $n \geq n_0$.



Consider $f(n)= n^2$

$g(n)=n$

$f(n)>= c.g(n)$

that is:

$n^2 >=c.n$

Assume c=3

$n^2 >=3.n$

For n=1 ==>, 1>=3 F

For n=2 ==>,4>=6 F

For n=3 ==>, 9>=9 T

Therefore

f(n)>=O(g(n), for all n>=3 and c=3

Big - Theta Notation (Θ)

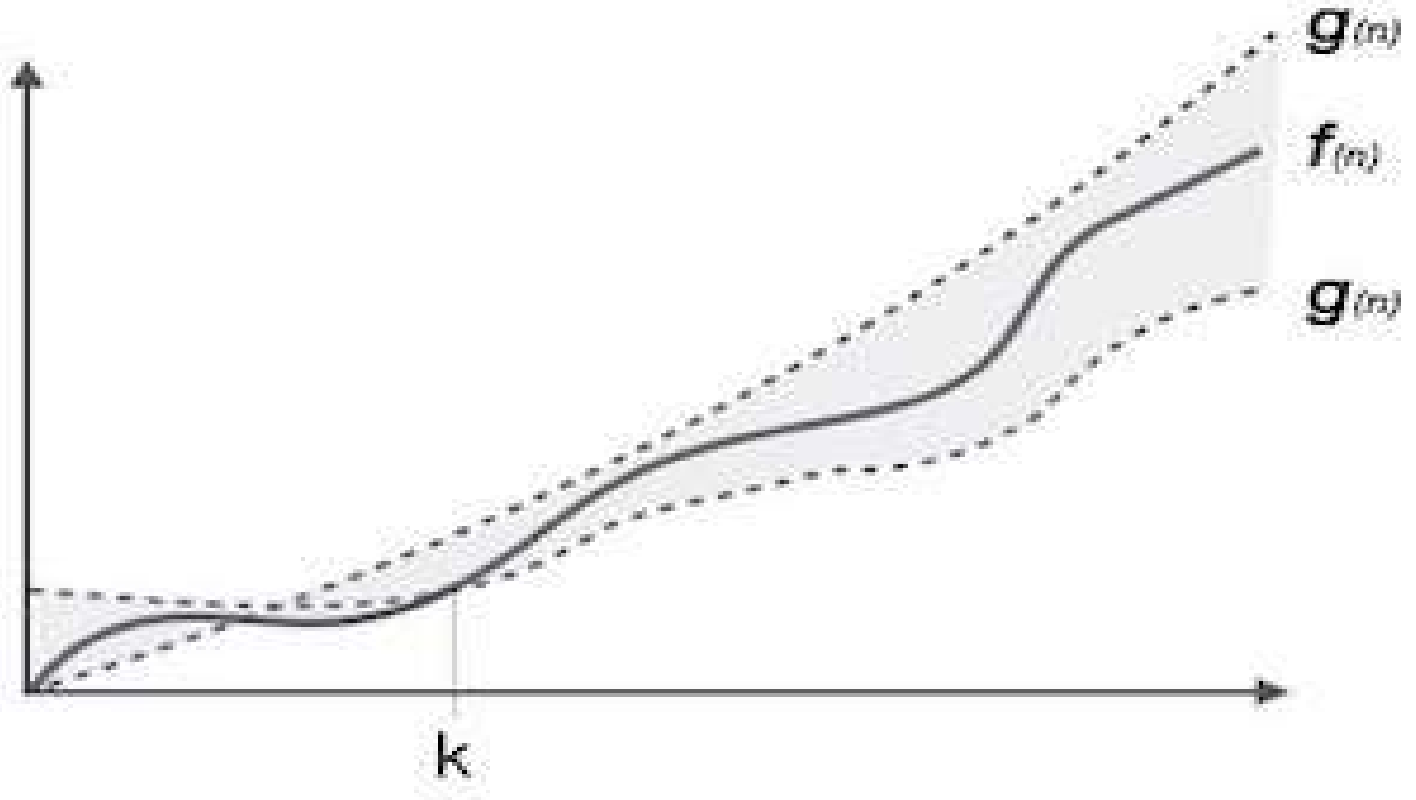
Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values.

That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Definition 1.6 [Theta] The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff 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 $n, n \geq n_0$.



$f(n) = 2n+5$

$g(n) = n$

$c_1 = 2, \quad c_2 = 3$

$\therefore c_1 g(n) \leq f(n) \leq c_2 g(n)$

i.e. $2n \leq 2n+5 \leq 3n$

for $n=1 \Rightarrow 2 \leq 7 \leq 3$ F

$n=2 \Rightarrow 4 \leq 9 \leq 6$ F

$n=3 \Rightarrow 6 \leq 11 \leq 9$ F

$n=4 \Rightarrow 8 \leq 13 \leq 12$ F

$n=5 \Rightarrow 10 \leq 15 \leq 15$ T

$\therefore f(n) = \Theta g(n)$

$\boxed{2n+5 = \Theta(n)}$ for all $n \geq 5$, $c_1=2, c_2=3$

Three time complexity functions

(or 3 different cases of time complexity)

1) Worst Case
2) Average Case
3) Best Case

```
int linearsearch (int a [ ], int first, int last, int key)
{
        for (int i = first; i < = last; i ++)
        {
                if (key = = a [i])
                {
                return i;          // successfully found the
                }         // key and return location
        }
        return – 1;                  // failed to find key element
}
```

Worst Case

The worse case time complexity of the algorithm is the function defined by the maximum number of operations performed, taken across all instances of size n. The worst-case time complexity is an upper bound on the running time for any input.

The worst case occurs when the search term is in the last slot in the array, or is not in the array.

The number of comparisons in worst case = size of the array = n.

Time complexity = O(n)

Avarage Case

- On average, the search term will be somewhere in the middle of the array.
- The average Case takes - **(*n*+1)/2** comparisons.
- Time complexity = O(n)

Best Case

The best case time complexity of the algorithm is the function defined by the minimum number of operations performed, taken across all instances of size n.

- The best case occurs when the search term is in the first slot in the array.
- Number of comparisons in best case = 1.
- Time complexity = O(1).

Growth function – Big-Oh (Big-O) notation

As the size grows, the growth rate becomes the most important aspect of the complexity of Algorithm. For this reason, a notation for expressing growth rate is used called Big-O notation.

Big-O notation indicates the growth rate

. Big-O function has a parameter n, where n is usually the size of the input to the algorithm.

. Big-O stands for “order of”

Definitions:-

T(n) = O(f(n)) is read as T of N is equal to Big-O of ‘f’ of n. It means that, growth rate of T(n) is less than or equal to growth rate of f(n).

where

T(n) is a non-negative valued function n be the size of the input.

Rules for Big-O notation

1. Ignore added constants.

   Because they become insignificant as the data increases

   eg: n+50 --> O(n)

   eg: a+n+c --> O(n)

2. Ignore constant multiplies

   (same reason)

   eg: 10*n --> O(n)

   50*n --> O(n)

3. When adding terms together, variable having highest power is selected. Use the larger of two
eg: n+n2 --> n2 O(n2)
    4n+20n4+10-->O(n4)

4. Write multiplication more compactly
    eg: n*n-->n2 O(n2)

5. If there are no loops or function call that can’t form a loop, the complexity is O(1).

6. For a sequence of S1,S2,S3,...Sk statements, running time is maximum of running time of individual statements.

Example
find the T(n) of 3n2+10n+10
3n2+10n+10 becomes==========>3n2+10n
(using rule 1)
3n2+10n becomes=======>3n2 (using rule 3)
3n2 becomes ========>n2
(using rule 2)
there fore T(n)= n2

The Most important Notations are:

O(1):-

- An algorithm with this running time is known as constant running time. That is O(1).
- Simple program statement take a constant amount of time.
- This means the algorithm always take the same amount of time regardless of the size of the input.

- For example, an algorithm which performs 7 multiplications has a constant running time. An algorithm which finishes in under a year has a constant running time.
- constant time is the best running time of an algorithm
- some examples are: Inserting an element into stack, array, linked list. or deleting .....

Example 1:-

```
 i=1;
printf(“%d”,i);
printf(“%d”,n);
printf(“%d”,i*n);
printf(“%d”,i*i);
printf(“%d”,n);
printf(“%d”,i*n);
printf(“%d”,i*i);
```

T(n) for each line is 1.

T(n) for the entire line algorithm is

1+1+1+1+1+1+1+1.

Using rule 1, the complexity is O(1).

so T(n) for entire algorithm is O(1).

- If there are 5 statements, we don’t say O(5). we always use O(1) to represent constant time.

- The amount of time spent executing the code will be the same, if there were 0 items or 10000 items.

O(n):

- An algorithm with this running time is known as “linear running time”.
- This is basically means that the amount of time to run the algorithm is proportional to the size of the input.
- Execution time increases linearly with the number of items you have.

- O(n) means that, if the size of the problem doubles, then running time and space are also doubles.
- Some examples are: Searching through an unordered list, increasing every element of an array.

Example

For(i=0;i<n;i++)

Printf(“%d”,i)

Here,

The amount of time required to execute the first line=n that is O(n).

The amount of time required to execute the second line=1 that is O(1).

T(n) for the entire 2 line algorithm is 1 x n------->n

So the algorithm is O(n)

T(n)=O(log n)

- An algorithm with this running time is called logarithmic running time.
- This means that, as the size of the input increases by a factor of n, the running time increases by a factor of the logarithm of n.
- For a binary search algorithm, we devide the list in half, in quarters, in eights, etc until we find the key.

- For example , there are 100 elements and it would take no more than 7 executions. Why 7? The answer is that $26<100<27$. Written another way, $6<\log 100< 7$, where log is in base 2

Therefor T(n) for binary search is O(logn)

- The running time is better than O(n), but not as good as O(1).

- Example code:

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

- An algorithm with this running time is known as quadratic running time.
- This means that whenever you increases the size of the input by a factor of n, the running time increases by a factor of $n^{2.}$
- execution time increases with the squire of the number of items you have.
- $O(n^2)$ means if the size of the problem doubles then four time as much storage or time will be needed.
- Example:

O(nlogn)

- The running time is better than O(n2), but not as good as O(n)
- The fastest sorting algorithms including merge sort, and quick sort have O(nlogn) running time.

Example:

O($\mathbf{2^n}$)

- An algorithm with this running time is known as exponential.
- This means that, its running time will doubles every time you add another element to the input.
- An algorithm which takes an input with 30 elements need to perform as many as 1 billion steps. If the input has 40 elements, then 1 trillion steps may be necessary.
- Ex: factoring large numbers expressed in binary.

$O(n^n)$

- This is known as polynomial growth
- Algorithm which takes 10 elements of input may need to perform 10 billion steps.

O(n!)

- An algorithm with this running time is known as factorial.
- If the input size is n, then the total time will be proportional to nx(n-1)x(n-2)x….2x1.
- For example if n=8. The no of steps will be proportional to 8x7x6x5x4x3x2x1=40520
- If the input size reaches 15, the no of steps may exceed 1 trillion.

Algorithm	Complexity
Bubble Sort Insertion Sort	**Worst complexity:** n^2 **Average complexity:** n^2 **Best complexity:** n **Space complexity:** 1
Selection sort	Worst complexity: n^2 Average complexity: n^2 Best complexity: n^2 Space complexity: 1
Merge sort	Worst complexity: n*log(n) Average complexity: n*log(n) Best complexity: n*log(n) Space complexity: n

Algorithm	Complexity
Quick sort	Worst complexity: n^2 Average complexity: n*log(n) Best complexity: n*log(n)
Bucket sort	Worst complexity: $O(n^2)$. Average complexity : $O(n + k)$.
Heap sort	Worst complexity: n*log(n) Average complexity: n*log(n) Best complexity: n*log(n) Space complexity: 1
Count sort	Worst complexity: n+r Average complexity: n+r Space complexity: n+r

Algorithm	Complexity
Linear search	Worst complexity: O(n) Average complexity: O(n) Worst-case space complexity: O(1)
Binary search	Worst complexity: O(log n) Average complexity: O(log n) Best complexity: O(1)

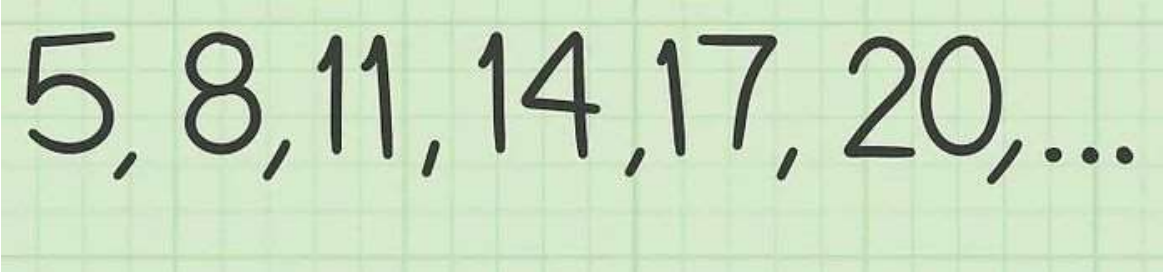
Recursion

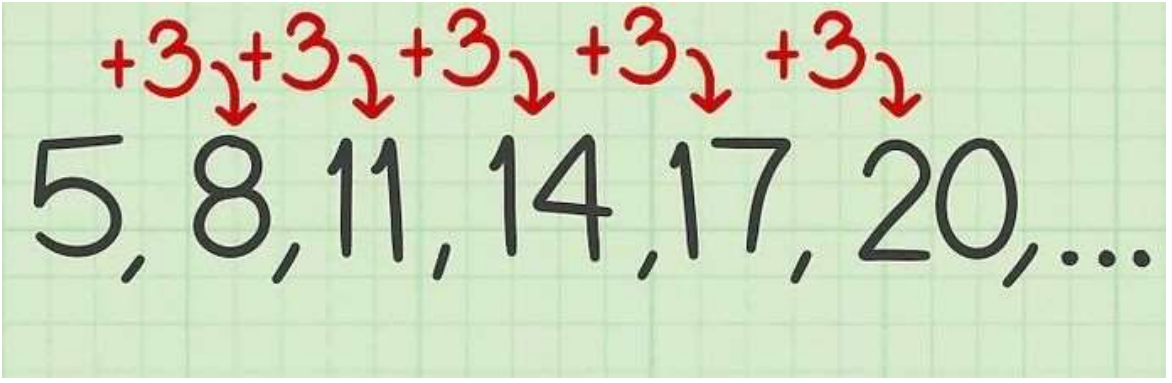
Recurence Relation

An equation that expresses 'a_n' in terms of one or more of the previous terms of the sequence, namely a_0, a_1, a_2, …a_{n-1}, for all integers $n>=n_0$, where n_0 is non negative integer is called a recurrence relation for the sequence $\{a_n\}$ or a difference equation.

A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

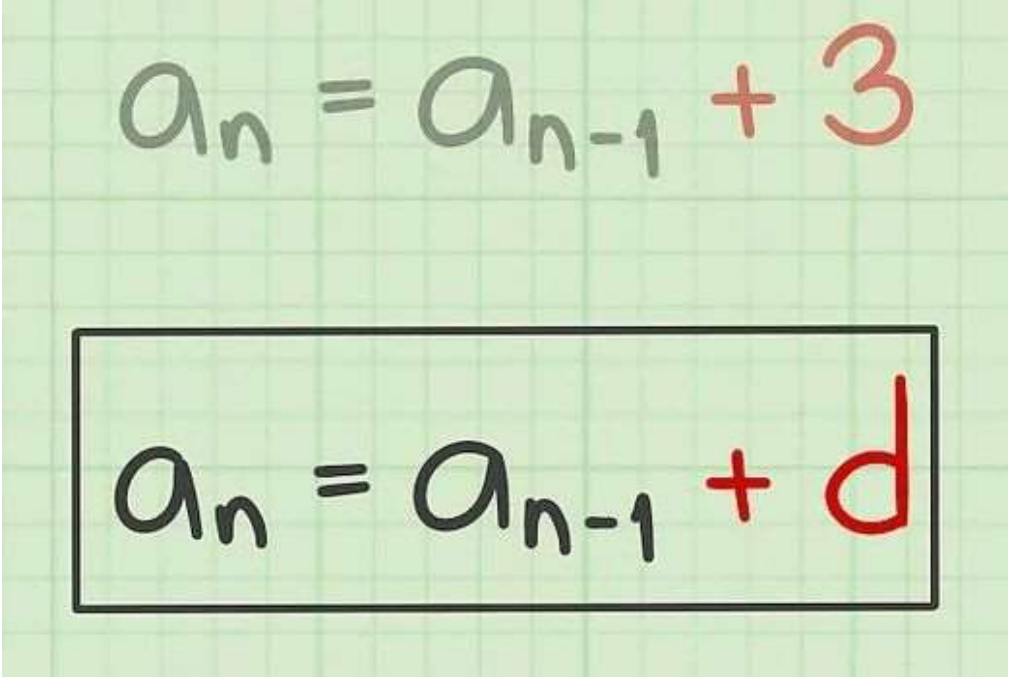
The simplest form of a recurrence relation is the case where the next term depends only on the immediately previous term.
Ex:





$$a_0 = 5$$

$$a_n = a_{n-1} + 3$$


$a_n = a_{n-1} + 3$
$a_n = a_{n-1} + d$

Like all recursive functions, it has one or more recursive cases and one or more base cases.

T(n) = time to solve problem of size n

- Recursive Case

T(0) = time to solve problem of size 0

- Base Case

Example:

Recurence relation for fibonacci sequence

$$F_n = F_{n-1} + F_{n-2} \qquad n>=2$$

$$F_0=1 \quad F_1=1$$

N=2 $F_2 = F_{2-1} + F_{2-2}$

$F_2 = F_1 + F_0$

$F_2=1+1=2$

N=3 $F_3 = F_{3-1} + F_{3-2}$

$F_3 = F_2 + F_1$

$F_3 = 2+1 = 3$

N=4 $F_4 = F_{4-1} + F_{4-2}$

$F_4 = F_3 + F_2$

$F_4 = 3+2 = 5$

N=5 $F_5 = F_{5-1} + F_{5-2}$

$F_5 = F_4 + F_3$

$F_5 = 5+3 = 8$

The sequence {1,1,2,3,5,8,……} is called fobonacci sequence.

The recurrence relation for Fibonacci sequence is :

$$F_n = F_{n-1} + F_{n-2} \qquad n>=2$$

$$F_0=1 \quad F_1=1$$

Find the recurrence relation for the sequence $\{ 1, 3, 3^2, 3^3, 3^4, 3^5, \ldots\ldots \}$?

$a_n = \{ 1, 3, 3^2, 3^3, 3^4, 3^5, \ldots\ldots \}$

$a_0 = 1$ $a_0 = 1$

$a_1 = 3$ $a_1 = 3x1$ $a_1 = 3xa_0$

$a_2 = 3^2$ $a_2 = 3x3$ $a_2 = 3xa_1$

$a_3 = 3^3$ $a_3 = 3x\ 3^2$ $a_3 = 3xa_2$

$a_4 = 3^4$ $a_4 = 3x\ 3^3$ $a_4 = 3xa_3$

Recurrence relation is :

$a_n = 3a_{n-1}$

Initial condition:

$a_0 = 1$

```
int fun(n)
{
if(n>0)
Printf(“mace”);
fun (n-1);
}
```

$$T(n)=\begin{cases} T(n-1)+1 & n>=0 \\ 1 & n=0 \end{cases}$$

```
int fact(n)
{
if(n==0)
return(1);
else
{
n=n*fact(n-1);
return n;
}
}
```

$$T(n)=\begin{cases} n \times T(n-1)+1 & n>=0 \\ 1 & n=0 \end{cases}$$

Solution to the recurrence relation.

There are four methods for solving Recurrence:

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Method

Substitution Method

The substitution method is a technique used to solve recurrence relations by making a guess about the form of the solution and then proving that the guess is correct using mathematical induction.

Solve the following recurrence relation using substitution method.

$$T(n) = \begin{cases} T(n-1)+1 & n>=0 \\ 1 & n=0 \end{cases}$$

$T(n) = T(n-1)+1$ (1)

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

$T(n-2) = T(n-2-1)+1 = T(n-3)+1$ (3)

$T(n-3) = T(n-3-1)+1 = T(n-4)+1$ (4)

By substitute (1) with (2):

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

$= T(n-3)+1+2 = T(n-3)+3$

$= T(n-4)+4$

$= T(n-5)+5$

……

For Step k times

$= T(n-k)+k$

Assume, $n-k=0$

Then $n=k$

That is $T(n)=T(n-n)+n$

$T(n)=T(0)+n$

$T(n)=1+n$

$=O(n)$

Iteration method

The iteration method, also known as the unfolding or expansion method, is another technique used to solve recurrence relations.

This method involves expanding the recurrence relation step-by-step until a pattern emerges. Once the pattern is identified, it can be generalized to find the closed-form solution.

Iteration method

Solve $T(n)=8T(n/2)+n^2$ $T(1)=1$

$T(n) = \mathbf{n^2+8T(n/2)}$ $T(n) = n^2+8T(n/2)$

$= n^2+8(8T(n/2^2)+(n/2)^2$ $T(n/2) = (n/2)^2+8T(n/2/2)$

$= n^2+8^2T(n/2^2)+8(n^2/4)$ $=$

$\mathbf{n^2+ 2n^2+8^2T(n/2^2)}$

$= n^2+ 2n^2+8^2[(n/4)^2+8T(n/4/2)]$ $T(n/4)=(n/4)^2+8T(n/4/2)$

$= n^2+ 2n^2+8^2[(n/4)^2+8T(n/8)]$

$= n^2+ 2n^2+8^2(n/4)^2+8^3T(n/2^3)]$

$= n^2+ 2n^2+4n^2+8^3T(n/2^3)$

$= \mathbf{n^2+ 2n^2+2^2n^2+8^3T(n/2^3)}$

$= \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$

ith time
$n/2^i = 1$
$n = 2^i$
$i = \log n$

It execute i times. That is

$T(n) = n^2 + 2n^2 + 2^2n^2 + \ldots\ldots + 2^{i-1}n^2 + 8^iT(n/2^i)$

$= n^2 + 2n^2 + 2^2n^2 + \ldots\ldots + 2^{i-1}n^2 + 8^iT(n/2^i)$

$= n^2 + 2n^2 + 2^2n^2 + \ldots\ldots + 2^{\log n - 1}n^2 + 8^{\log n}T(n/2^{\log n})$

$= n^2 \sum_{k=0}^{\log n - 1} 2^k + (2^3)^{\log n}$

$\sum_{k=0}^{\log n - 1} 2^k$ is a geometric series

$= \ldots\ldots\ldots\ldots$

$= n^2\theta(2^{\log n - 1}) + n^3$

$= \theta(n)$

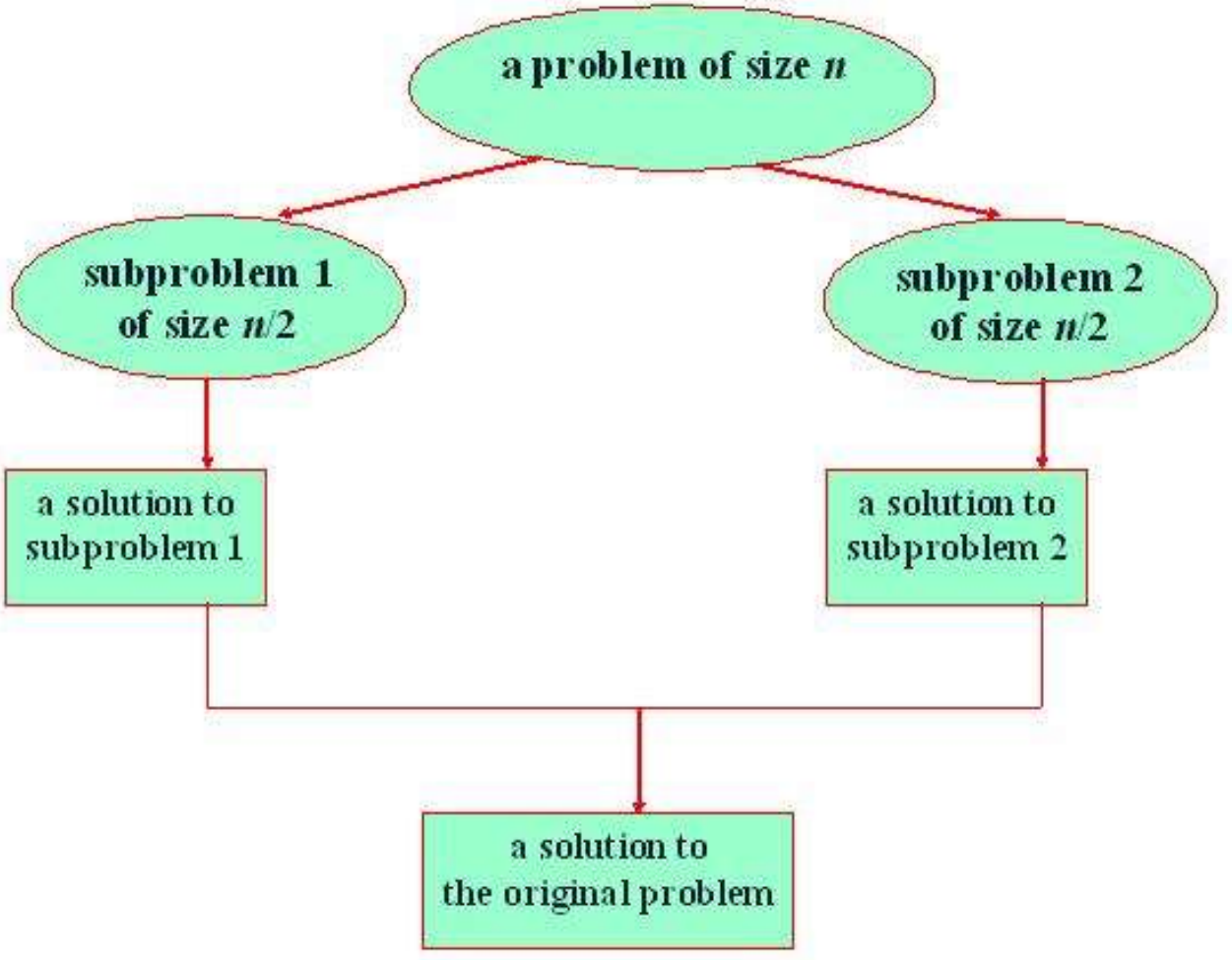
Divide and Conquer Method

A divide and conquer algorithm 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.

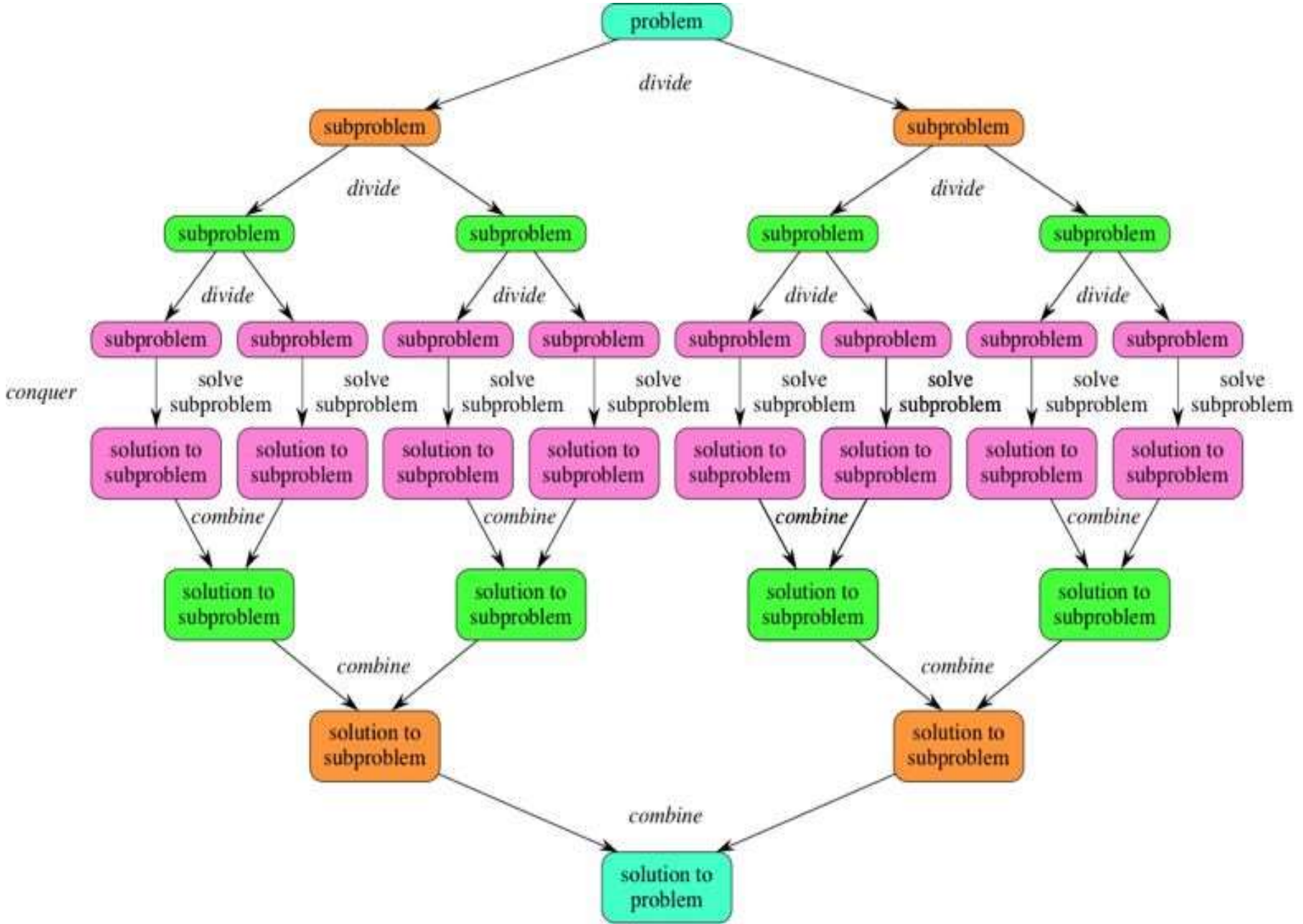
A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide:* Break the given problem into sub problems of same type.
2. *Conquer:* Recursively solve these sub problems
3. *Combine:* Appropriately combine the answers

Divide and conquer method



If we expand out two more recursive steps, it looks like this:



```
1   Algorithm DAndC(P)
2   {
3       if Small(P) then return S(P);
4       else
5       {
6           divide P into smaller instances P_1, P_2, ..., P_k, k ≥ 1;
7           Apply DAndC to each of these subproblems;
8           return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
9       }
10  }
```

Algorithm 3.1 Control abstraction for divide-and-conquer

Merge Sort

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

```
1   Algorithm Merge(low, mid, high)
2   // a[low : high] is a global array containing two sorted
3   // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4   // is to merge these two sets into a single set residing
5   // in a[low : high]. b[ ] is an auxiliary global array.
6   {
7       h := low; i := low; j := mid + 1;
8       while ((h ≤ mid) and (j ≤ high)) do
9       {
10          if (a[h] ≤ a[j]) then
11          {
12              b[i] := a[h];h := h + 1;
13          }
14          else
15          {
16              b[i] := a[j]; j := j + 1;
17          }
18          i := i + 1;
19      }
20      if (h > mid) then
21          for k := j to high do
22          {
23              b[i] := a[k]; i := i + 1;
24          }
25      else
26          for k := h to mid do
27          {
28              b[i] := a[k]; i := i + 1;
29          }
30      for k := low to high do a[k] := b[k];
31  }
```

Quick Sort

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

```
1    Algorithm Partition(a, m, p)
2    // Within a[m], a[m + 1], ..., a[p − 1] the elements are
3    // rearranged in such a manner that if initially t = a[m],
4    // then after completion a[q] = t for some q between m
5    // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6    // for q < k < p. q is returned. Set a[p] = ∞.
7    {
8        v := a[m]; i := m; j := p;
9        repeat
10       {
11           repeat
12               i := i + 1;
13           until (a[i] ≥ v);

14           repeat
15               j := j − 1;
16           until (a[j] ≤ v);

17           if (i < j) then Interchange(a, i, j);

18       } until (i ≥ j);

19       a[m] := a[j]; a[j] := v; return j;
20   }

1    Algorithm Interchange(a, i, j)
2    // Exchange a[i] with a[j].
3    {
4        p := a[i];
5        a[i] := a[j]; a[j] := p;
6    }
```

---

gorithm 3.12 Partition the array $a[m : p-1]$ about $a[m]$

Strassen’s Matrix multiplication

Strassen’s Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n.

If the matrix is large, then divide the problem into several subproblem until it is a 2x2 matrix. Then we can applay strassens matrix multiplication

The complexity of traditional matrix multiplication is o(n3), that is there are 8 multiplications and 4 additions.

Multiplication is more expensive than adition.

So Volker strassen discover a new method for matrx multiplication.

It require only 7 multiplication and 18 additions. The formula is given below:

Ex:

$$\begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \times \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix} = \begin{pmatrix} P+S-T+V & R+T \\ Q+S & P+R-Q+U \end{pmatrix}$$

Where,

P=(A11+A22) X (B11+B22)

Q=(A21+A22) X B11

R= A11 X (B12-B22)

S=A22 X (B21-B11)

T=(A11+A12) XB22

U=(A21-A11) X (B11 + B12)

V=(A12-A22) X (B21+B22)