Unit-1 Notes

Algorithm

1.1 Introduction

Definition

"Algorithmic is the backend concept of the program or it is just like the recipe of the program."

Understanding of Algorithm

An algorithm is a sequence of unambiguous instruction for solving a problem, for obtaining a required output for any legitimate input in a finite amount of time.

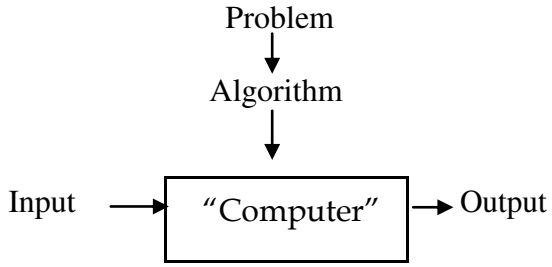


Fig 1.1

In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, algorithm and program distinguishes as:-

- Algorithm is a backend concept of program.
- A program does not necessarily satisfy the fourth condition.
- One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.
- We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Example:

PUZZLE(x)

```
while x != 1
   if x is even
    then  x = x / 2
    else x = 3x + 1
```

Input: x=2

Output: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

1.2 Fundamentals of the Analysis of Algorithm

- **How to express an Algorithm?**
  - English
  - Pseudo Code
- **Algorithm Design Goals**

The two basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space

- **Performance of a program:**

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

1.2.1 Analysis Frame work

Importance of Analyze Algorithm

- Need to recognize limitations of various algorithms for solving a problem.
- Need to understand relationship between problem size and running time.
  - When 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.

Why do we analyze about them?

- **Understand their behavior, and (Job -- Selection, performance, modify)**
- **Improve them. (Research)**

What do we analyze about them?

- Correctness
  - Does the input/output relation match algorithm requirement?
- Amount of work done
  - Basic operations to do task
- Amount of space used
  - Memory used
- Simplicity, clarity
  - Verification and implementation.

- Optimality
    - Is it impossible to do better?

There are two kinds of efficiency

Time efficiency - "indicates how fast an algorithm in question runs."

Space efficiency - "deals with the extra space the algorithm requires."

Complexity:-

"The complexity of an algorithm is simply the amount of work the algorithm performs to complete its task. "

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

1. **Instruction space:** Instruction space is the space needed to store the compiled version of the program instructions.
2. **Data space:** Data space is the space needed to store all constant and variable values.
   Data space has two components:
   - Space needed by constants and simple variables in program.
   - Space needed by dynamically allocated objects such as arrays and class instances.

Note: But there are some other factors more important than performance:

- Modularity
- Correctness
- Maintainability
- Functionality
- Robustness
- User-friendliness
- Programmer time
- Simplicity
- Extensibility
- Reliability

Note: Why study algorithms and performance?

- Algorithms help us to understand *scalability.*
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a *language for talking about program behavior.*

- Performance is the *currency of computing.*
- The lessons of program performance generalize to other computing resources.

1.3 Algorithm Designing Approaches

Most popular algorithm designing approaches:

Standard methods (for easy problems)

1. Incremental Approach
2. Divide-and-conquer
3. Greedy Strategy
4. Dynamic programming
5. Search

Advanced methods (for hard problems)

6. Probabilistic/randomized algorithms
7. Approximation algorithms

1.4 Classification of various Algorithms running time:

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

1. **log n:** When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction.
2. **n** :When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.
3. **nlog n:** This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions. When n doubles, the running time more than doubles.
4. $\mathbf{n^2}$: When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop).
5. **n3:** Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems.

1.5 Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

2. Asymptotic Notations (Rate of Growth):

The following notations are commonly use notations in performance analysis and used to characterize **2.1** The complexity of an algorithm:

1. Big–OH (O)
2. Big–OMEGA (Ω)
3. Big–THETA (Θ)

1. Big–OH O (Upper Bound)

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

It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm.

We can represent it graphically

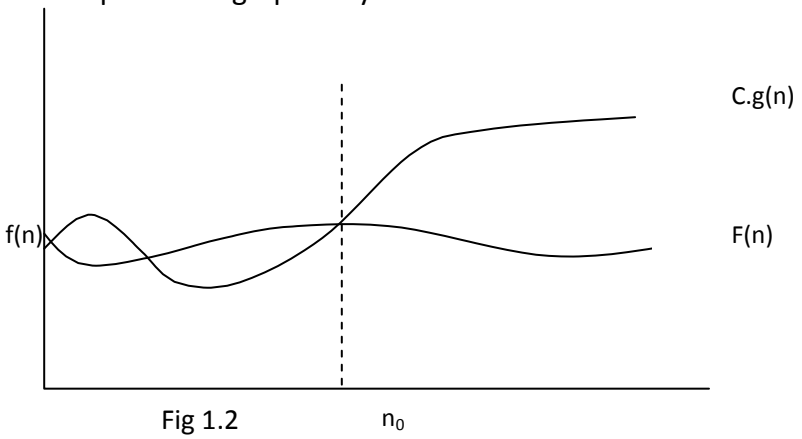


Fig 1.2

Example:

$f(n) = 3n + 2$

General form is $f(n) \le cg(n)$

When $n \ge 2$, $3n + 2 \le 3n + n = 4n$

Hence $f(n) = O(n)$

here $c = 4$ and $n_0 = 2$

When $n \ge 1$, $3n + 2 \le 3n + 2n = 5n$

Hence $f(n) = O(n)$, here $c = 5$ and $n_0 = 1$

Hence we can have different c, n_0 pairs satisfying for a given function.

2. Big–OMEGA Ω (Lower Bound)

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

It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm.

We can represent it graphically as

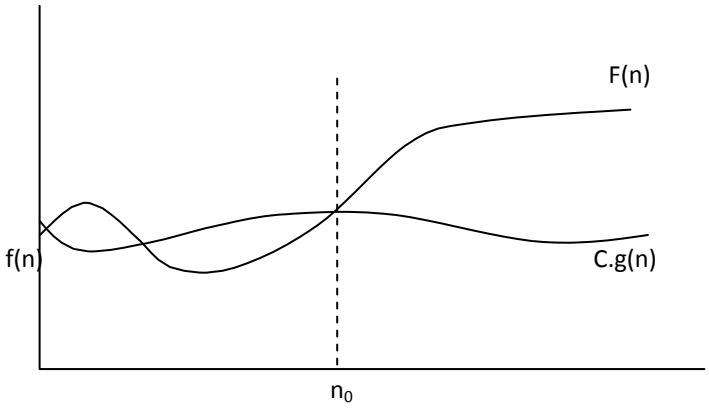


Fig 1.3

Example:

f(n) = 3n + 2

3n + 2 > 3n for all n. Hence f(n) = Ω(n)

3. Big–THETA Θ (Same order)

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).

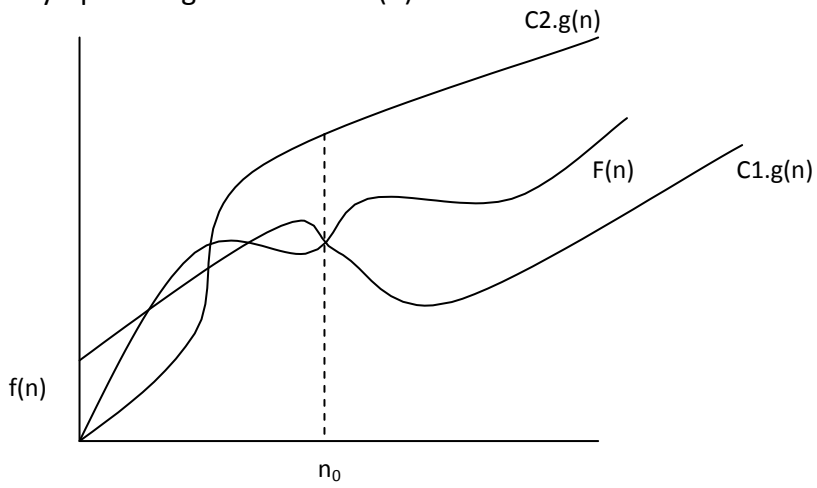


Fig 1.4

Example:

f(n) = 3n + 2

f(n) = Θ(n) because f(n) = O(n) , n ≥ 2.

2.2 Growth of Function:

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O(log n), O(n), O(nlogn), $O(n^2)$, $O(n^3)$, $O(2^n)$, n! and n^n.

The execution time for six of the typical functions is given below:

N	logn	n*logn	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

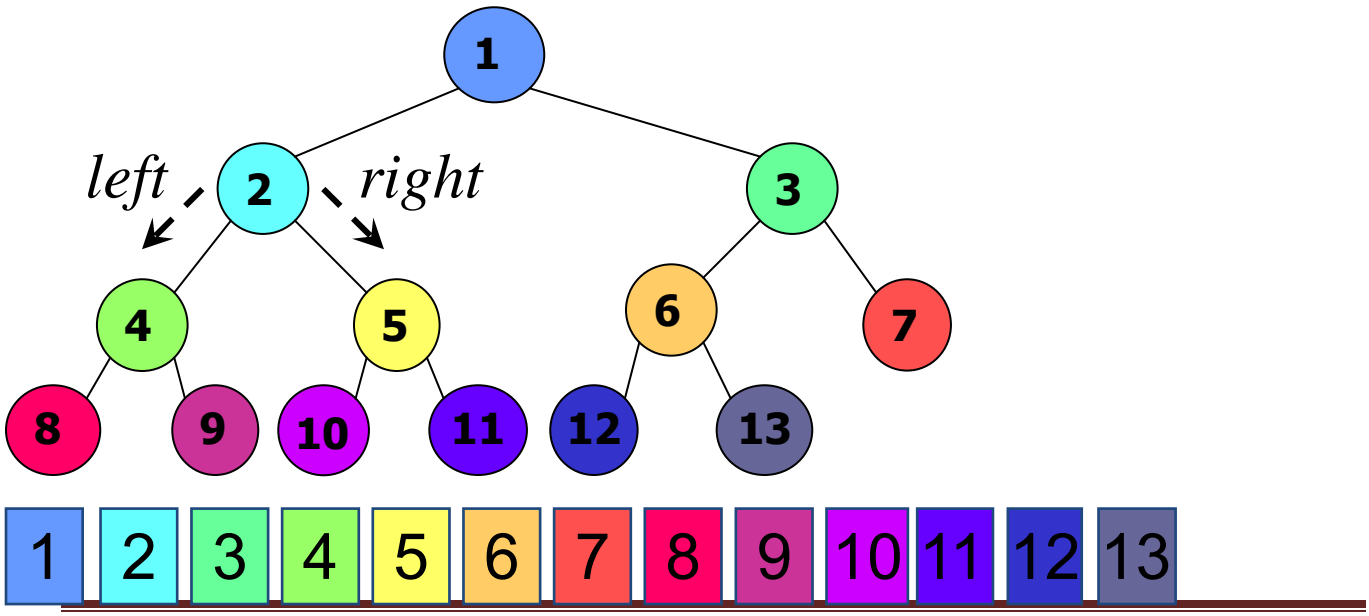
Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.
Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billion years.

3. Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

3.1 Binary Heap:

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.



3.2 Heap Sort Algorithm for sorting in increasing order:

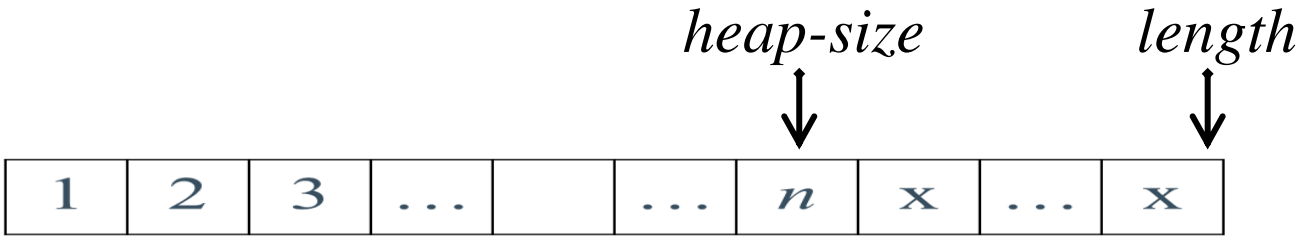
1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.

3.3 Representation of Binary Heaps:

- An array *A* that represents a heap is an object with two attributes:
  - *length*[*A*], which is the number of elements in the array
  - *heap-size*[*A*], the number of elements in the heap stored within array *A*.

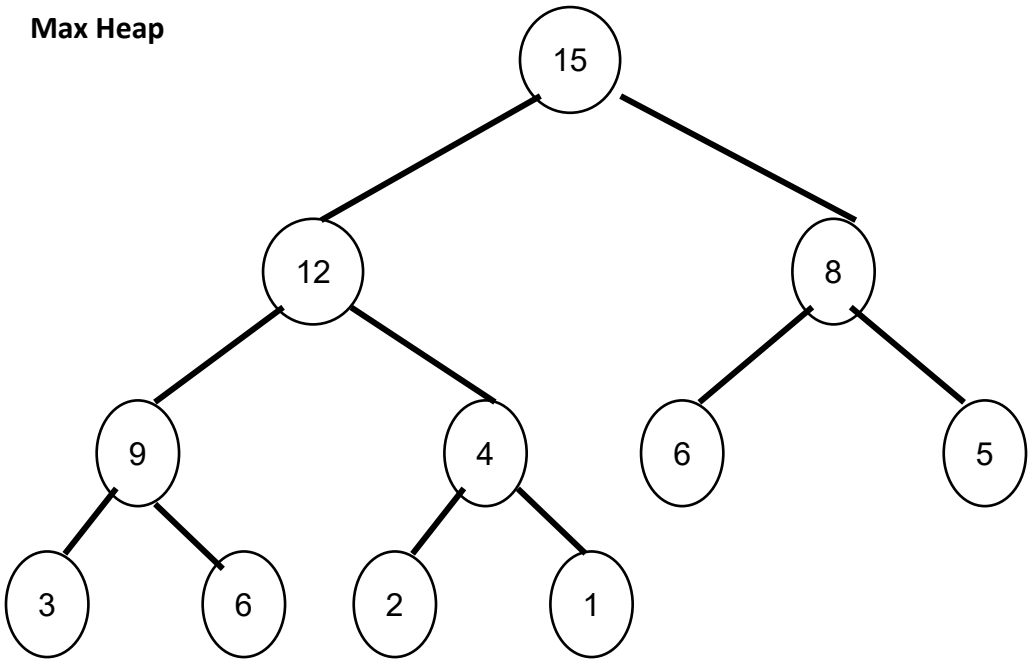


3.4 Properties of Binary Heaps

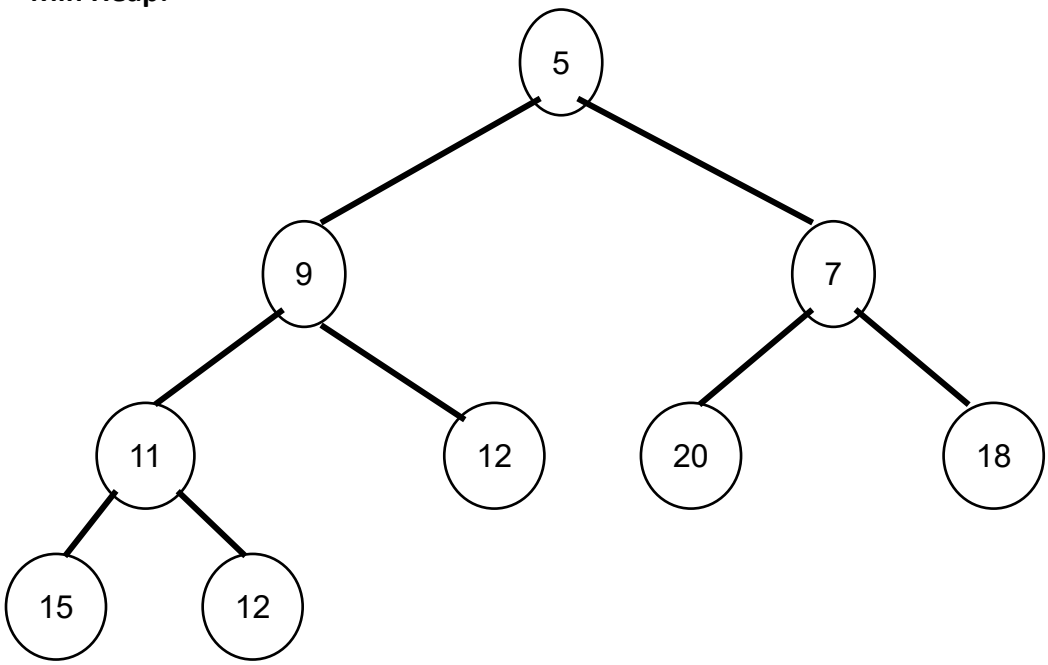
- If a heap contains *n* elements, its height is $\lg_2 n$.
- In a max-heaps
  For every non-root node *i*, *A*[PARENT(*i*)] ≥ *A*[*i*]
- In a min-heaps
  For every non-root node *i*, *A*[PARENT(*i*)] ≤ *A*[*i*]

Example:

Max Heap

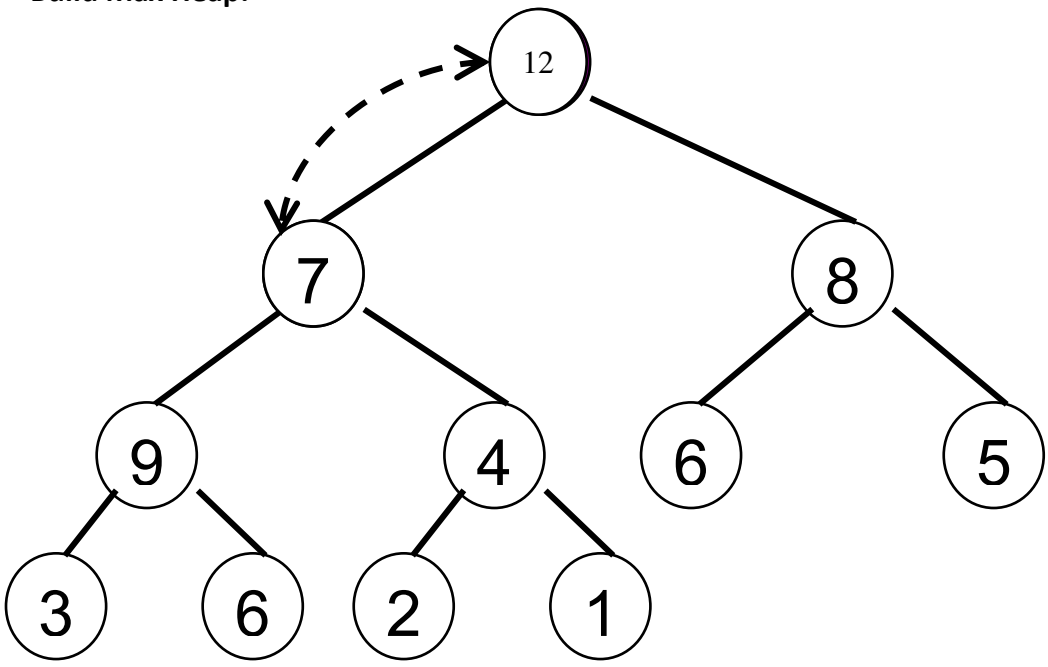


Min Heap:



Heapify:

Build Max Heap:



Complexity of Heap Sort :
Worst Case Time Complexity : O(n log n)
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)

Space Complexity : O(1)

Introduction to Divide and Conquer Technique

4. Binary Search:

The Binary search technique is a search technique which is based on Divide & Conquer strategy. The entered array must be sorted for the searching, then we calculate the location of mid element by using formula mid= (Beg + End)/2, here Beg and End represent the initial and last position of array. In this technique we compare the Key element to mid element. So there May be three cases:-

1. If array[mid] = = Key (Element found and Location is Mid)
2. If array[mid] > Key ,Then set End = mid-1.(continue the process)
3. If array [mid] < Key, Then set Beg=Mid+1. (Continue the process)

4.1 Binary Search Algorithm

1. [Initialize segment variable] set beg=LB,End=UB and Mid=int(beg+end)/2.
2. Repeat step 3 and 4 while beg<=end and Data[mid] != item.
3. If item< data[mid] then set end=mid-1
   Else if Item>data[mid] then set beg=mid+1[end of if structure]
4. Set mid= int(beg+end)/2.[End of step 2 loop]
5. If data[mid]=item then set Loc= Mid.
   Else set loc=null[end of if structure]
6. Exit.

4.2 Time complexity:

As we dispose of one part of the search case during every step of binary search, and perform the search operation on the other half, this results in a worst case time complexity of O (log2N).

5. Merge Sort

The merge () function is used for merging two halves. The merge (arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

5.1 Merge Sort Algorithm

```
procedure mergesort ( var a as array )
  if ( n == 1 ) return a
  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]
  l1 = mergesort( l1 )
  l2 = mergesort( l2 )
  return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
  var c as array
  while ( a and b have elements )
```

```
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while
  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while
  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while
  return c
end procedure
```

5.2 Time Complexity:

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \Theta(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(n\text{Log}n)$.

Time complexity of Merge Sort is $\Theta(n\text{Log}n)$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

6. Quick Sort:

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

```
/* low  --> Starting index,  high  --> Ending index */
```

6.1 Quick Sort Algorithm:

```
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[p] is now at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
/* This function takes last element as pivot, places  the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right
  of pivot */
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
   pivot = arr[high];
    i = (low - 1)  // Index of smaller element
   for (j = low; j <= high- 1; j++)
   {
      // If current element is smaller than or equal to pivot
      if (arr[j] <= pivot)
      {
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
      }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

6.2 Analysis of Quick Sort

Time taken by Quick Sort in general can be written as following.

$T(n) = T(k) + T(n-k-1) + \Theta(n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by Quick Sort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$T(n) = T(0) + T(n-1) + \Theta(n)$, which is equivalent to

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

The solution of above recurrence is $\Theta(n^2)$.

Best Case:

The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$T(n) = 2T(n/2) + \Theta(n)$

The solution of above recurrence is $\Theta(nLogn)$. It can be solved using case 2 of Master Theorem.

Average Case:

To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.

We can get an idea of average case by considering the case when partition puts O (n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$T(n) = T(n/9) + T(9n/10) + \Theta(n)$

Solution of above recurrence is also O (nlogn)

Although the worst case time complexity of Quick Sort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

7. Strassen's matrix Multiplication:

Suppose we want to multiply two matrices of size *N* x *N*: for example *A* x *B* = *C*.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$C_{11} = a_{11} b_{11} + a_{12} b_{21}$$

$$C_{12} = a_{11} b_{12} + a_{12} b_{22}$$

$$C_{21} = a_{21} b_{11} + a_{22} b_{21}$$

$$C_{22} = a_{21} b_{12} + a_{22} b_{22}$$

This type of matrix multiplication done by the conventional Divide & Conquer Technique, here the input matrix is divided into N/2 X N/2 matrix, so if N =2 then the size of each sub problem will be 1 X 1,then there are total 8 sub problems.

Hence in summarized, 2x2 matrix multiplication can be accomplished in 8 multiplication.

Mathmatically: **($2^{\log_2 8} = 2^3$)**

7.1 General Algorithm of Conventional Matrix Multiplication Technique:

```
void matrix_mult ()
{
  for (i = 1; i <= N; i++)
 {
      for (j = 1; j <= N; j++)
{
compute C_{i,j};
}
  }
}
```

Time Analysis:

$$C_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} c = cN^3 = O(N^3)$$

Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions.

($2^{\log_2 7} = 2^{2.807}$)

- This reduce can be done by Divide and Conquer Approach.

7.2 Strassens's Matrix Multiplication:

P1 = (A11+ A22)(B11+B22)
P2 = (A21 + A22) * B11
P3 = A11 * (B12 - B22)
P4 = A22 * (B21 - B11)
P5 = (A11 + A12) * B22
P6 = (A21 - A11) * (B11 + B12)
P7 = (A12 - A22) * (B21 + B22)
C11 = P1 + P4 - P5 + P7
C12 = P3 + P5
C21 = P2 + P4
C22 = P1 + P3 - P2 + P6
Here are no of multiplication is 7 so-
Complexity = T(n) = $\Theta(n^{\log_2(7)}) = \Theta(n^{2.8})$