Array

(1) Find an efficient algorithm to search for an element in a 2-dimensional array of size $m \times n$. Assume each row and column in the given 2-D array are sorted.

eg: 3×4

5	8	10	15
6	9	12	17
14	16	19	23

$x = 14$: (3, 1)

$x = 28$: (-1, -1)

Solution 1 - Naive solution

time $\leq m * n$ (Comparisons)

↑ core/dominating

$\log n$ (2) → (decaying factor)

space → Constant

Solution 2 - Binary Search

(a) time complexity $\leq m * \log n$ (2) (comparisons)

space complexity → constant (inplace memory)

(b) $m < n$ or $m > n$ (comparison)

time → $\min(m \log n, n \log m)$ } improvement but not solution

Solution 3 - start from top right most / bottom leftmost element

Repeat until i or j reaches boundary

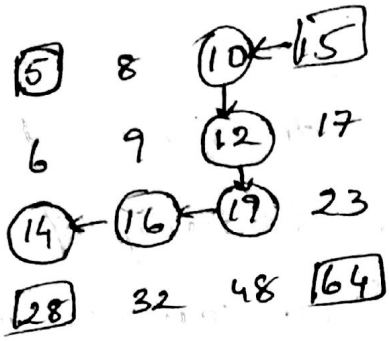
if $x <$ element at current (i, j)

--j;

else if $x >$ element at current (i, j)

++i;

else return (i, j)



time complexity → m+n comparisions
space → constant.

C-Program

```
typedef struct pair_s
  {
    int r1,
    int c1,
  } * pair;

pair search (int **a, int m, int n, int x)
  {
    pair p = malloc (sizeof (*pair));
    r=0, c=n-1;
    while (r<m && c>=0)
    {
      if (x < a[r][c])
          --c;
      else if (x > a[r][c])
          ++r;
      else {
        p->r1 := r;
        p->c1 = c;
        return p;
      }
    }
    p->r1 = -1;
    p->c1 = -1;
    return p;
  }
```

Problem Solving Flow

Tools

- Algorithm Patterns
  - Recursion
  - Divide & Conquer
  - Greedy
  - Dynamic Programming
  - Back Tracking
- Data Structures
  - List
  - Set = Tree set / Hash set
  - Map
  - Queue
  - Priority Queue
- Adhoc Solution

Problems

- Streaming Problems
- Graph Problem [Social n/w]
- String Problem [Biometrics]
- Regular coding problem

Google Summer Code

(P3) Find an efficient algorithm to compute the number of common elements between 2 arrays of size m & n respectively.

I/P:- 3 4

10	5	8

8	6	1	10

O/P:- 2

Solution 1 : Name Solution -

TC → ≤ mn Comparisions

SC → Constant

✓ Solution 2 :- Sort the bigger array.

m | 10 | 5 | 8 |

n | 1 | 6 | 8 | 10 |

TC → if m<n : m log n + n log n

if m>n : n log m + m log n

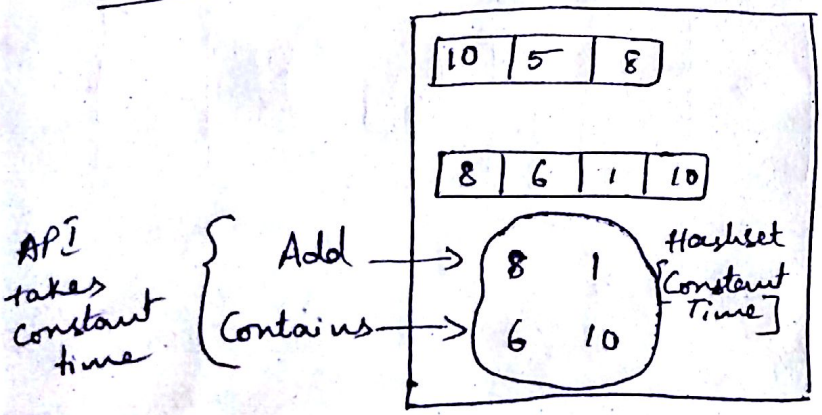
SC → Constant

Solution 3 :- Sort both arrays & scan through both of them

TC → m log m + n log n + m + n.

SC → constant.

✓ Solution 4 :- Use HashSet



TC → if n>m

n * Constant time + m * Constant Time = (m+n) * C

SC → n slot

Add
Contains
Remove
→ Hash Set

Ordering
[K^{th} smallest Element]
→ Tree Set

If List is used, time complexity is $\underbrace{n*n}_{\text{Contains}} + \underbrace{\text{Constant}}_{\text{Add}}$

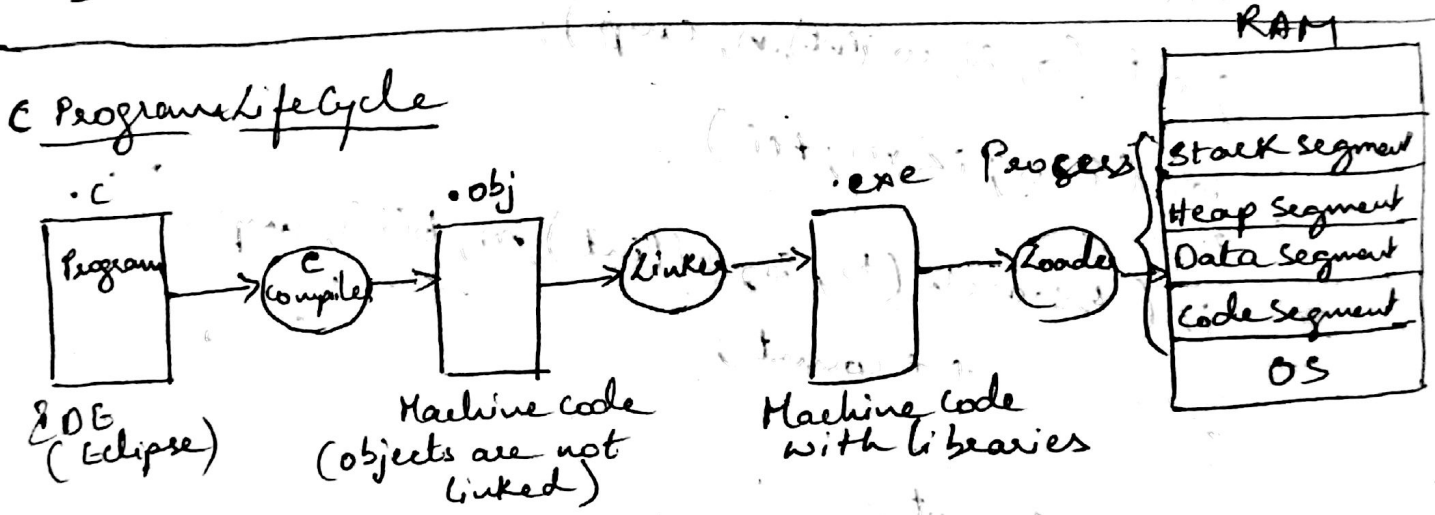
C-Program

Pre-condition: $n >> m$

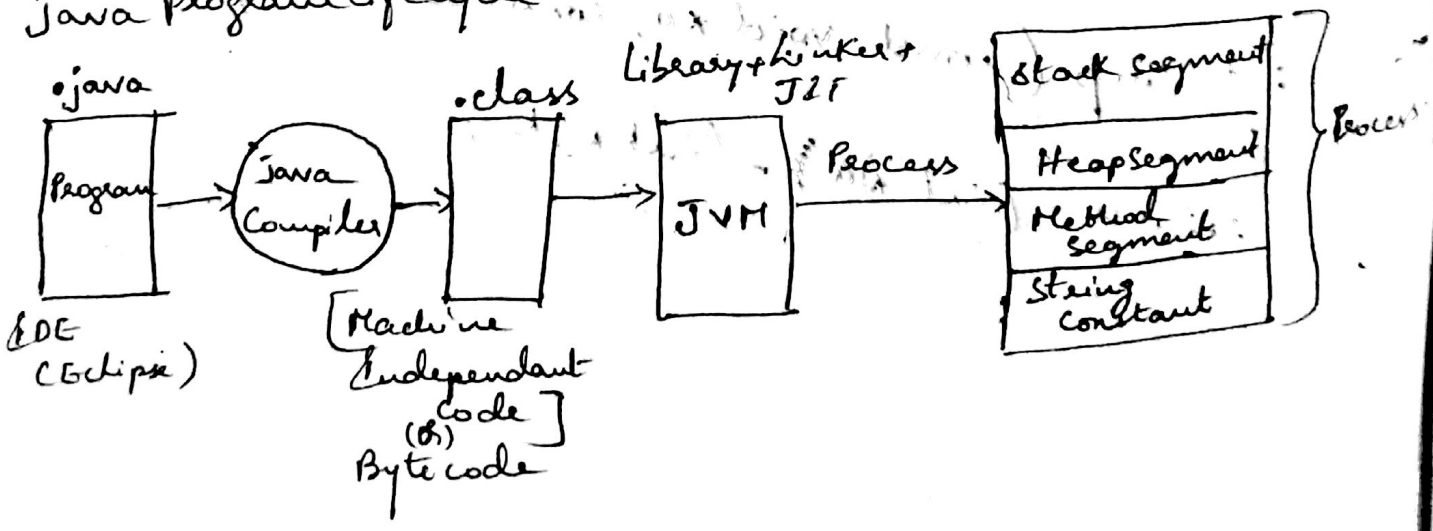
```
int findCommonElements(int [],int,int[],int)
{
  int count=0,i;
  qsort(a,sizeof(int),n,cmp);
  for(i=0;i<m;++i)
  {
   if(bsearch(b,sizeof(int),m,b[i],cmp))
          ++count;
  }
 return count;
}
int cmp(const void*a, const void* b)
{
 return (int*)a - (int*)b;
}
```

```
public class Test
{
Public x static int findCommonElements(int[]a, int[]b)
{
  Array.Sort(a); int count=0;
  for(int i=0; i<b.length; i++)
    if(Array.bsearch(a, b[i]) > 0)
      ++count;
  return count;
}
```

C Program Life Cycle



Java Program life cycle



In C

Data Segment → Static/Global Data } Initialized to 0.

Heap Segment → Dynamically requested Data Allocation
Eg:- malloc | realloc | calloc

Stack Segment → Local data

Code Segment → All the instructions across all functions
Eg:- Function pointer → pointer points to some code in code segment.
Eg:- const int a=10;

In java,

Stack Segment → Local data

Method segment → static data + All instructions across functions

Heap Segment → Object instance data

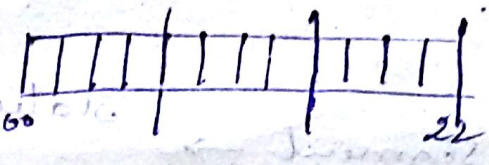
string constant → All constants not modifyable.

In any Language,

Activation Record → Record for function / Method call.
They goes into stack segment

return Value
Temporary data
Return Address
Local data

→ Return value of the function, to the caller function
→ f = e+d+10; } e+d+10 can't be done at same time so store it on temporary data
→ to the previous address
→ e=10; d=20;

int a[3][4]; ⟶ 

Static Array in C

```
int **a; r=3; c=4;
a=malloc(r * sizeof(int *));
for(i=0; i<r; i++)
    a[r] = malloc(c * sizeof(int));
```

In java,

```
int [][]a;
a = new int[r][c];
```

(or)

```
int [][]a;
a = new int[2][];
a[0] = new int[3];
a[1] = new int[10];
```

Until a new object is called in java, no memory is allocated

(P4) Find an efficient algorithm to compute Pow(x,n)

i/p pow(2,10) · Pow(3,4)

o/p 1024 3^4

Solution 1: Naive Solution

time complexity ≤ n-1 multiplication

Space complexity = constant

Solution 2: Use Recursion Pattern

time complexity ≤ $2*\log_2 n$

C-Program

```
long pow(int x, int n)
{
  if(n==0)
    return 1;
  if(n==1);
    return x;
  if(n%2==0)
  {
    tmp = pow(x, n/2);
    return tmp*tmp;
  }
  else
  {
    tmp = pow(x, n/2);
    return tmp*tmp*x;
  }
}
```

Activation Record →

Temp data
Return Address
Local data
Return Value

Space Complexity ≤ $c * \log_2 n$

Handling stack Overflow

(a) Reduce the depth of recursion.

(b) Reduce the size of AR.

(c) Increase stack size; if possible

(d) Convert Recursive Program → Non-recursive Program with the help of heap size.

(P5.) Towers of Hannoi.

Goal: Transfer all disks from A to C with minimum number of disks moves.

Rules: 1.) only one disk can be moved at a time.

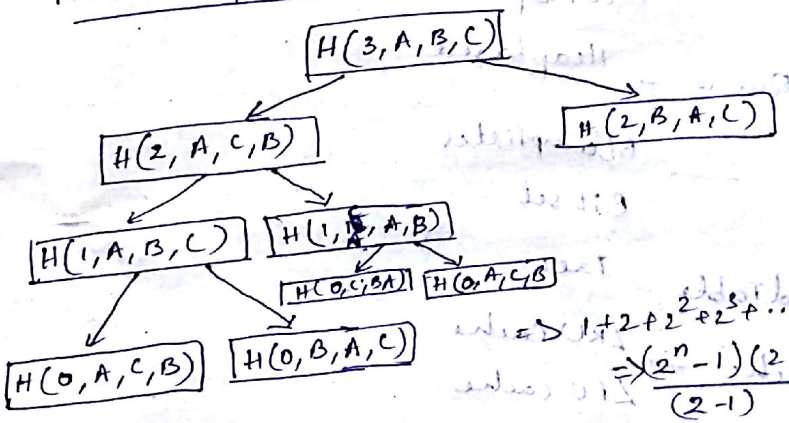
2.) The larger disk cannot be moved on to smaller disk

Solution 1:

Minimum number of moves from A to C for n disks = Minimum number of moves from A to B for n-1 disks + 1 + [B, C]

```
class Test
{
public static void hannoi(int n, char src, char aux, char target)
{   if(n==0) return; if(n==1) S.O.P(src + "→" + target);
    hannoi(n-1, src, target, aux);
    S.O.P(src + "→" + target);
    hannoi(n-1, aux, src, target);
}
}
```

Time Complexity. Number of nodes in the tree.



$\Rightarrow 1+2+2^2+2^3+\ldots+2^{n-1}$

$= \frac{(2^n-1)(2-1)}{(2-1)} = 2^n - 1$

Space Complexity → (n+1)*c

Height of the tree.

Data Structures → A logical container.

Abstract Datatype | interface | .h → Implementation / Representation

List — ArrayList, LinkedList.

Set — HashSet, TreeSet

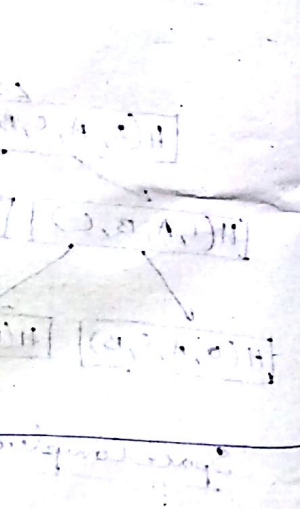
Map — HashMap, TreeMap.

PriorityQueue — Heapbased

Filter — BLoomfilter, Bitset

SortedTable — Tree

s/w Cache — LRU Cache, LFU cache.

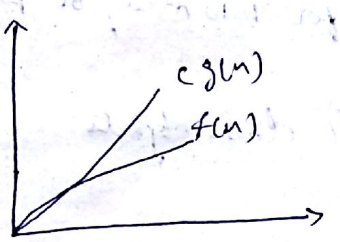


O (Big-oh) Upper Bound.

$f(n) = O(g(n))$ iff

$\exists$ no, c such that $\forall n \geq n_0$,

$f(n) \leq c \cdot g(n)$



Eg:- $n^2 + n\log n + 10 \simeq O(n^2)$

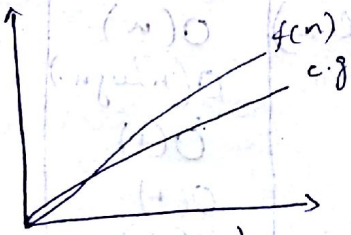
$\Rightarrow n^2 + n\log n + 10 \leq c \cdot n^2$.

Ω (Rem) Lower Bound

$f(n) = \Omega(g(n))$ iff

$\exists$ no, c such that $\forall n \geq n_0$,

$f(n) \geq c \cdot g(n)$.



$n^2 + 10 \simeq \Omega(n^2)$.

$n^2 + 10 \geq c \cdot n^2$

If $O(g(n)) = \Omega(g(n))$ then, it can be denoted with $\Theta(g(n))$.

Issues with Array

(a) Size must be known at run-time but in reality it is not useful.

(b) Insertion or Deletion is not effective.

(c) Contiguous Memory for 1-D Array or pointers pointing to 1-D Array.

(d) Arrays are not first class objects

List

A group of elements with relative order among themselves.

Interface / ADT / .h	← Implementation → ArrayList/Vector	LinkedList
add(e)	$O(1)_{amortized}$	$O(1)$
add(index, e)	$O(n)$	$O(n)$
contains(e)	$O(n)$	$O(n)$
remove(e)	$O(n)$	$O(n)$
remove(index)	$O(n)$	$O(n)$
sort()	$O(n \log n)$	$O(n \log n)$
size()	$O(1)$	$O(n)$
get(index)	$O(1)$	$O(n)$
iterator()	$O(n)$	$O(n)$

ADT → Abstract Data Type.

JAVA

```
interface List<T>
{
  void add(T e);
  boolean add(int ind, T e);
  T get(int ind);
  boolean remove(T e);
  int size();
  boolean contains(T, e);
}
```

Class → create/Implement user defined Types.

```
public class LinkedList<T> implements List<T>,
{ final
private Node<T> head;
  private int size;
  public LinkedList<T>()
  {
    head = new Node<T>();
  }
  public T get(int index)
  {
    Node current = head;
    for(int i=0; i<index && current.next!=NULL; i++)
      current = current.next;
    if(current.next==NULL)
      return NULL;
    return current.data;
  }
```

```
public void add(T e){
Node<T> current = head;
    while(current.next != NULL)
        current = current.next;

    current.next = new Node<T>(e);
    ++this.size;
}

class Node<T>
{
  T data;
  Node<T> next;
public Node<T>(){

  }
  private
  public Node<T>(T e){
   data = e;
   next = NULL;
  }
}
```

C - Program

list.h → header file

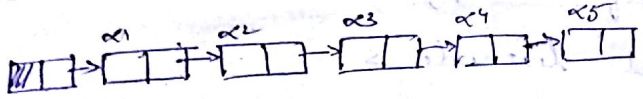
```
typedef ___ T;
struct listnode_s
{
  T data;
  struct listnode_s * Node next;
}; typedef listnode_s * Node;
struct linkedlist_s
{
   Node head;
   int size;
}
typedef linkedlist_s * LinkedList;
void add (LinkedList list, T e);
T get (LinkedList list, int index);
void init(LinkedList list);
```

list.c → c file

```
#include "list.h"
Void add(LinkedList list, T e)
{
    Node current = head; list->head;
    while (current->next != NULL)
      current = current->next;
    Node temp
    temp = malloc(sizeof(*Node));
      temp->data = e;
      temp->next = NULL /* use calloc to implicitly
                           set the pointers to
                           NULL reference */
    current->next = temp;
}
```

Singly linked list

(P1) Find an efficient algorithm that returns middle element of single linked list

i/p 

Solution1: Brute Force Solution

TC → n + n/2 link traversals

O(n)

SC → O(1)

Solution2 :- use 2-pointer

slow → one link at a time

fast → two links at a time

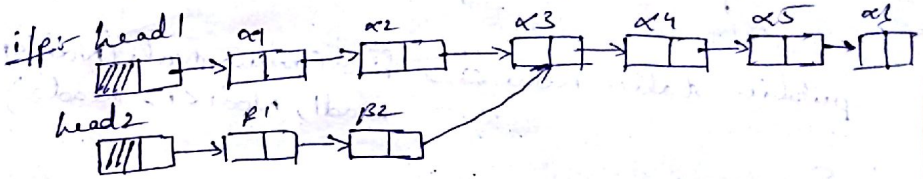
TC → n + n/2 = 3n/2 ~ O(n)

SC → O(1)

Program (Java)

```
public T findMiddle() {
 Node slow = fast = head;
    while(fast != NULL && fast.next != NULL)
    {
      slow = slow.next;
      fast = fast.next.next;
    }
 return slow.data;
}
```

(P2) Find an efficient algorithm to determine the first common node between two single linked lists.

i/p: 

Solution1 :- Naive Approach

Time Complexity ≤ mn → O(mn)

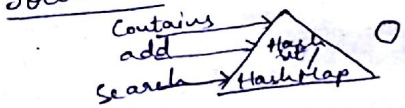
Space Complexity → O(1)

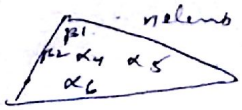
Solution 2 :- use two stacks.

Time complexity → m * O(1) + n * O(1)

space complexity → m+n

Solution 3 :- Use Hashset





Time Complexity → n * O(1) + m * O(1)

- Space Complexity → O(n)

Solution 4:- Use lengths of lists

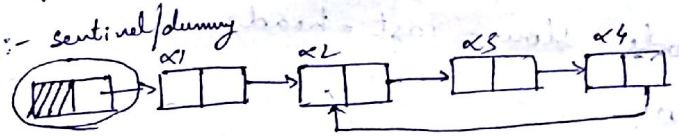
Time Complexity → $m+n+(\leq m+n)$ → $O(m+n)$

Space Complexity → $O(1)$

```
public static Node<T> findCommonNode(Node<T> head1, Node<T> head2)
{
    Set<Node<T>> set = new HashSet<Node<T>>();
    Node<T> current = null;
    for (current = head2; current != NULL; current = current.next)
        set.add(current);
    for (current = head1.next; current != NULL; current = current.next)
        if (set.contains(current))
            return current;
    return null;
}
```

(P3) Find an efficient algorithm to determine whether a given singly linked list has cycle or not.

i/p :- sentinel/dummy



o/p :- true

Solution 1 : use Hashset

Time Complexity → $n*2*O(1) = O(n)$

Space Complexity → $O(n)$.

Solution 2 :- Use two pointer approach:

Slow → one pointer a node at a time

fast → pointer to a node moving twice speed

Approach 1 :- No cycle.

Fast pointer (n) + slow pointer $(n/2)$

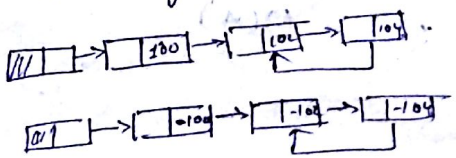
Approach 2 :- If cycle exists

Fast pointer $(2n)$ + slow pointer (n).

∴ Time complexity → $O(n)$.

Space Complexity → $O(1)$.

Solution 3 :- use negation

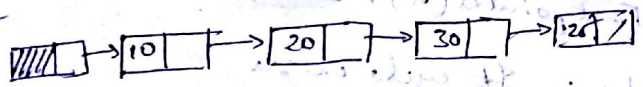


Java Program:

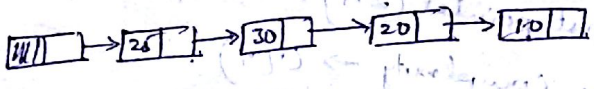
```
public boolean hasCycle()
{
    Node slow = fast = head;
    do
    {
        slow = slow.next;
        if(fast == NULL || fast.next == NULL)
        return false;
        fast = fast.next.next;
    }while(slow != fast)
    return true;
}
```

(P4) Find an efficient algorithm to reverse nodes of singly linked list.

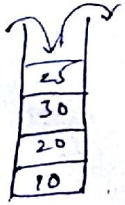
I/P:-



O/P:-



Solution1 :- Use stack



TC → 2n + O(1) = O(n)

SC → O(n)

Solution2 :- Adhoc strategy

```
public void reverse()
{
Node prev = NULL, current = head.next;
    temp = NULL;
    while(current != NULL)
    {
        temp = current.next;
        current.next = prev;
        prev = current; current = temp;
    }
    head.next = prev;
}
```

(P5) Algorithm to remove loop in a single linked list

Solution1 :- Use Hashset

TC → O(n)

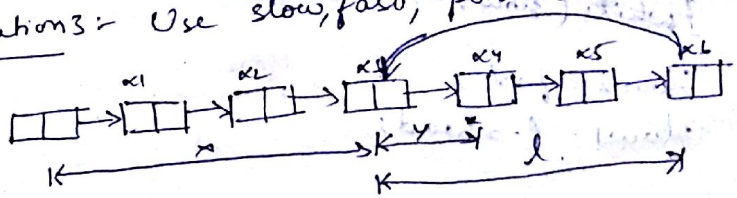
SC → O(n)

Solution2 :- Use Negation Approach. The first Node address to be changed from -ve to +ve.

TC → O(n)

SC → O(1)

Solution3 :- Use slow, fast, pointer:



Total distance travelled by fast pointer = 2 * Total distance travelled by slow pointer

$x + y + K \times l = 2 * (x + y)$

$\therefore x + y = Kl$

~~$x + y = Kl$~~

$x = Kl - y$

Time Complexity → $3n + (n + n')$

→ $5n \simeq O(n)$

Space Complexity → $O(1)$

C-Program:

```
boolean removeloop(LinkedList l)
{
    Node slow, fast, prev, slow1, slow2;
    slow = fast = l->head;
    do
    {
        slow = slow -> next;
        if (fast == NULL || fast -> next != NULL)
        {
            fast = fast -> next -> next;
            return false;
        }
    } while (slow != fast)
    slow2 = fast;
    slow1 = l->head;
    while (slow1 != slow2)
    {
        slow1 = slow1 -> next;
        prev = slow2;
        slow2 = slow2 -> next;
    }
    prev -> next != NULL;
    return true;
}
```

Solution1:-

```
boolean removeloop (LinkedList l)
{
    Hashset hs = create Hashset();
    hs.init(); // Constructs method
    for (current = l->head; current->next != NULL; current = current->next)
        if (! contains (hs, current -> next))
            add (hs, current -> next); // Data structure is passed as an argument
        else {
            current -> next = NULL;
            return true;
        }
    return false;
}
```

Issues with SLL

(1) If a pointer to any Node is given, it takes $O(n)$ to remove from SLL.

(2) Add a node before pointer, it takes $O(n)$

Circular DLL

(a) Boundary cases need not be handled separately

(b) No need to maintain a Node pointer to last element.

Issues with list

Array → Continuous Memory.
→ Size must be known at runtime
→ Operations are not native

List → An ordered (relative) collection of elements.

A TreeList doesn't exist since there we can't perform operations like adding after 2nd element.

∴ List is a Linear Data Structure.

It always maintain relative order that's why requires more time for operations like

add(e)
remove(e)
Contains(e)

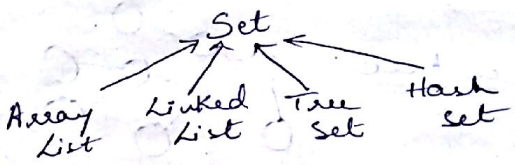
Set & Map

Set → A group of distinct unordered elements

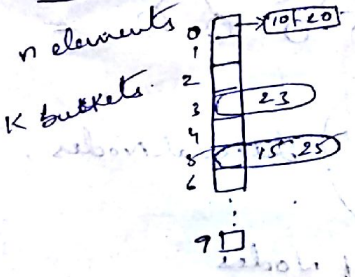
Map → A group of key, value pairs where keys are distinct unordered elements.

Multiset/Bag → A group of elements

MultiMap → A group of key, value paired elements.



Hash Arrangement



$h(Key) = \_\_ \% 10$

loadFactor = bucketSize

$\geq n/K$

Set	HashSet	TreeSet (Balanced BST)
add(e)	O(1)	O(logn)
remove(e)	O(1)	O(logn)
Contains(e)	O(1)	O(logn)
size()	O(1)	O(1)
iterator()	O(n)	O(n)
SortedSet		O(logn)
findMin()	O(n)	O(logn)
findMax()	O(n)	O(logn)
Select(K)	O( )	O(logn)

hashCode ("abc") → polynomial expression with a degree of 2.

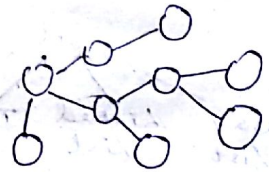
Meesenne's number → 31:

$\therefore a(31)^2 + b(31) + c$

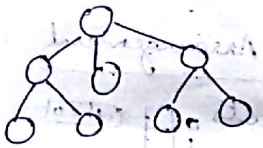
The hash code function need to be identified based on the pattern.

Tree Arrangement :-

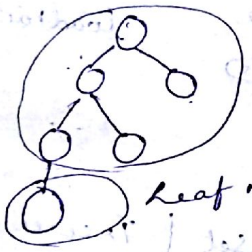
Spanning / Free tree :



Ordered / Rooted Tree :-



Binary Tree :-



Binary Search Tree (BST) :-

For every node x, all left subtrees are less than x & all right subtrees are greater than x.

All nodes are balanced → Balanced BST.

Map :- A group of ∞ (Key, Value) paired elements with distinct elements.

Map	HashMap	TreeMap
put(K,V)	O(1)	O(logn)
remove(K)	O(1)	O(logn)
containsKey(K)	O(1)	O(logn)
containsValue(V)	O(n)	O(n)
V get(K)	O(1)	O(logn)
size()	O(1)	O(1)
display()	O(n)	O(n)
SortedMAP		
findMin()	O(n)	O(logn)
findMax()	O(n)	O(logn)
select(K)	O( )	O(logn)

(1) Find an efficient algorithm to determine whether the two given array elements are identical or not.

i/p :- n=4 10 15 2 3
2 10 15 3

o/p :- 3

Solution 1:- Naive Approach. comparing one element with all other elements in another array

TC → $O(n^2)$

SC → $O(1)$

Solution 2:- Sorting / Sorting both arrays and compare each element

TC → $O(n \log n)$

SC → $O(1)$

Solution 3:- Hashset. Add one array to Hash set and remove elements using 2nd array.

TC → $O(n)$

SC → $O(n)$

If duplicates exists use HashMap.

(P2) Find the paragraph that contains all the words of a text file and clear contents.

Solution 1:- Naive Solution

Pass 1:- No: of distinct elements in the entire file.

Pass 2:- For each paragraph use a Hashset and compare the size.

Solution 2:- Use HashMap | Word | Para Number |

LRU CACHE Design

Cache is a buffer in memory which occupies fixed size.

add(K,V) → Replacement Policy ← LFU [Least Frequently Used], FIFO [First In First Out], LRU [Least Recently Used]

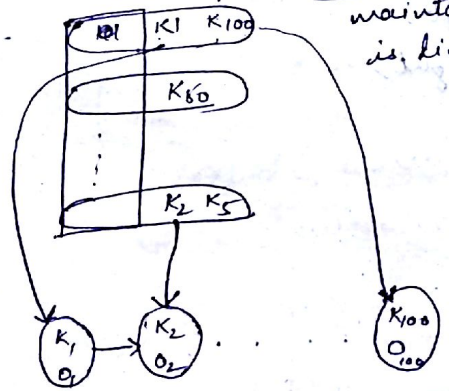
V get(K)

clear

0 1 2 3 . . 100

Cache(L)	HashMap with timestamp	Doubly LinkedList	LinkedList + HashMap
add(K,V)	O(n)	O(n)	O(1)
V get(K)	O(1)	O(n)	O(1)
clear()			

HashMap → In Memory data with indexing is maintained while actual data is LinkedList with Key.



LRU → LinkedHashMap.

① Design SortedTable DS

Application : UI Table

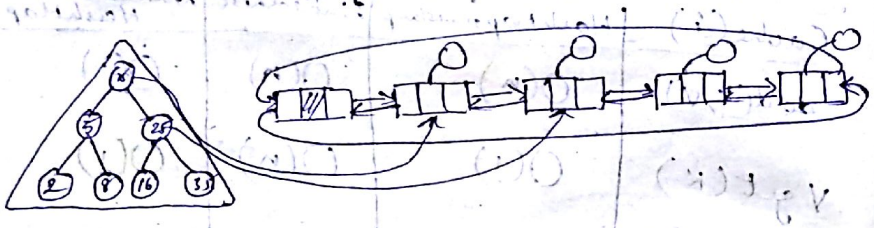
Solution1 : Use Linked List

foreach click on UI

TC → O(nlogn)

Solution2 :- Linked List + TreeMap for indexing

TC → O(n)



```
class TreeNode<T>
{
  TreeNode<T> left;
  TreeNode<T> right;
  T data;
```

```
class BST <T extends Comparable<T>>
{
  private TreeNode<T> root;

  public int size()
  {
    return auxsize(root);
  }
  private int auxsize(TreeNode current)
  {
    if(current == NULL) return 0;
    if(current.left == NULL && current.right == NULL)
      return 1;
    int ls = auxsize(current.left);
    int rs = auxsize(current.right);
    return ls + rs + 1;
  }
  public boolean contains(T key)
  {
    TreeNode current = root;
    while(current != NULL)
    {
      int res = key.compareTo(current.data);
      if(res < 0)
        current = current.left;
      else if(res > 0)
        current = current.right;
      else
        return true;
    }
    return false;
  }
```

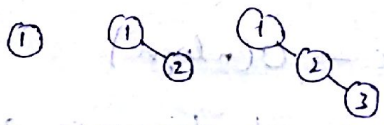
```
public int auxHeight()
{
  return auxHeight(root);
}

public int auxHeight(current)
{
  if (current == NULL)
      return 0;
  if (current.left == NULL && current.right == NULL)
    return 1;
  return Math.max(auxHeight(current.left),
      auxHeight(current.right)) + 1;
}

public boolean add(T key)
{
  current = root; int res;
  while (current != NULL)
  {
    int res = key.compareTo(current.data);
    if(res<0)
    {
      parent = current;
      current = current.left;
    }
    if (res>0)
    {
      parent = current;
      current = current.right;
    }
    if(res==0) return false;
    TreeNode temp = new TreeNode<T>(key);
    if (parent != NULL)
    if (res < 0)
      parent.left = tmp;
    else
      parent.right = tmp;
    else
      root = tmp;
    return true;
  }
}
```

Divided Conquer:- Special form of recursion where 'n' is reduced by a fraction.

(P1) Given nodes 1-n construct a BST

1 2 3 4 5 6 7

Solution 1: Incremental construction of BST



Solution 2: Use Batch Insertion technique:
- Divide and Conquer pattern.

```
public TreeNode buildBST(int l, int r)
{
    if(l==r) return new TreeNode(l);
    if(l>r) return null; //when not perfectly balanced
    int m = (l+r)/2;
TreeNode temp = new TreeNode(m);
    temp.left = buildBST(l, m-1);
    temp.right = buildBST(m+1, r);
    return temp;
}
```

TC → O(n)

SC → O(logn)

① ② ③ ④

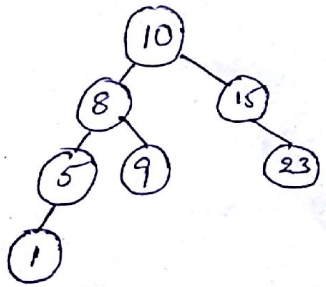
$\log_2^1 + \log_2^2 + \log_2^3 + \cdots + \log_2^{(n-1)} \simeq \log_2(n-1)!$

$\simeq \log_2 n^n$

$= n \log_2 n$

$\therefore TC \rightarrow O(n \log n)$

(P5.) Find an algorithm to traverse a BST of size 'n' in pre-order.



preorder :- 10 8 5 1 9 15 23

postorder :- 1 5 9 8 23 15 10

Inorder :- 1 5 8 9 10 15 23

Solution1 :- Use Recursion.

```
public void preOrder()
{
    preorder(root);
}
private void preorder(TreeNode current)
{
    if(current==NULL) return;
    S.O.P(current.data);
    preorder(current.left);
    preorder(current.right);
}
```

TC → O(n) ; SC → O(h)

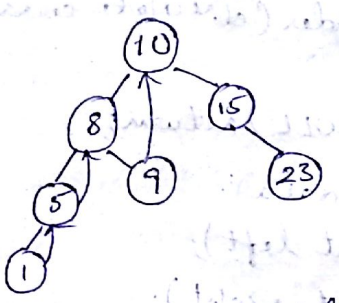
Solution2 :- Use non-recursive with explicit stack

```
public void preOrder()
{
Stack<TreeNode> st = new LinkedStack<TreeNode>();
    current = root;
    while(true)
    {
        while(current != NULL)
        {
            S.O.P(current.data);
            st.push(current);
            current = current.next;
        }
        if(st.empty()) break;
        current = st.pop().right;
    }
```

TC → O(n)

SC → O(h)

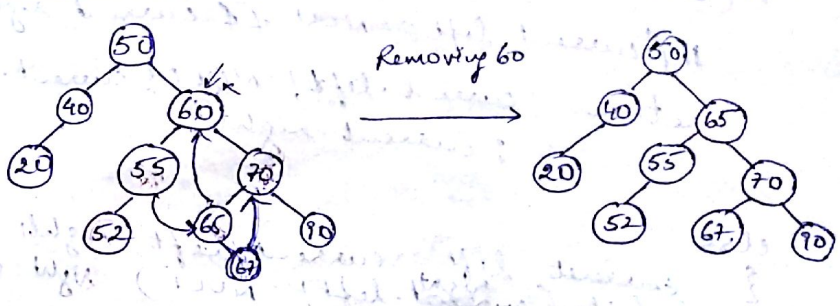
Solution3 :- Use Inplace traversal Idea.



Always from the node, traverse the left node and from there to the extreme right node leaf node. The right pointer of the leaf node is saved with the current node of traversal. Now on reaching the left most subtree which is a leaf node, traverse back by following through the right pointer to the root node.

To share a data across the recursive calls, the function can have an extra parameter which points to the heap space and on every function call, a local variable is stored in the AR which points to the heap space and modifies the code.

BST Removal :-




```
public boolean remove(T Key)
{
    return auxRemove(root, Key);
}
private TreeNode<T> auxRemove(TreeNode current, T Key)
{
    if(current==NULL)
        return NULL;
    int res = Key.compareTo(current.data);
    if(res<0)
current.left = auxRemove(current.left, Key);
    else if(res>0)
current.right = auxRemove(current.right, Key);
    else
    {
        if(current.left==NULL && current.right==NULL)
            current = NULL;
            return current;
        return NULL;
```

```
if (current.left != NULL && current.right == NULL
    || (current.left == NULL && current.right != NULL)
    return current.left != NULL ? current.left
                                : current.right;

else
{   current.left = current.left; right = current.right
    while (current.left != NULL)
    {
        right = right.left;
    }
    if (right.right != NULL)
        right.left = right.right;

    right.left = left;
    return right;
}
return current;
}
```

Order-based Operations

select(K)
FindMin() | FindMax() | findElements(K1, K2)

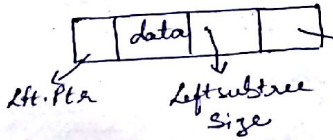
(P1) Find the K^{th} smallest element of a tree Node based data structure.

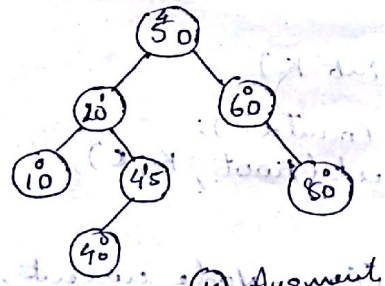
Solution1:- Go through the in-order process and then find the K^{th} smallest element.

TC → O(N)
SC → O(h).

Solution 2:- Rank based algorithm.

Augmenting base tree.





K = 3 → 45
K = 6 → 60

(1.) Augment the base tree with a new v.& pointer to point the size of the LST.

(2.) While moving to the left side of subtree, maintain the rank variable

(3.) While moving to the right side of subtree, decrement rank pointer by. (1)

(4) Rank is the position element given to the array.

```
public T select(int K)
{
  current = root;
  while (current != NULL)
  {
    if (K == current.ls_size+1)
      return current.data;
    if (K < current.ls_size+1)
      current = current.left;
    else
    {
      K = K - current.ls_size - 1;
      current = current.right;
    }
  }
  return null;
}
```

Solution 1 :-

```
public T select(int K)
{ counter c = new Counter();
  return auxSelect(root, K, c);
}
private T auxSelect(TreeNode current,
  int K; Counter c)
{
  if (current = NULL)
    return NULL;
  T left = auxSelect(current.left, K, c);
  if (left != NULL)
    return left;
  c.inc();
  if (c.get() == K)
    return current.data;
  T right = auxSelect(current.right, K, c);
  if (right != NULL) return right;
}
```

Safe URL feature in Browsing.

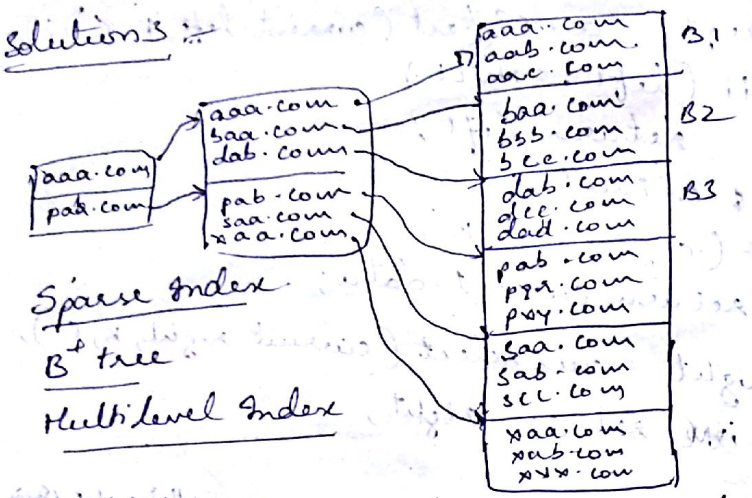
Solution 1 :- Naive Solution
→ Search entire file.
→ O(N/B) I/O
B → Block size in Bytes
N → File size in Bytes

Solution 2 :- Arrange all urls in a HashSet
preprocessing time : O(N/B).
Query time : O(1)
Space : O(n).

Solution 3 :- Summary data structures.
↳ Bit Set
↳ Bloom Set Filter

mathematics for ML
Khan's academy
(1) Data Mining.
(2) Queue sorting
(3) Bit set → character string
(4) Hash function more than one value for key

Solution 3 :-



Sparse Index

B^+ tree.

Multilevel Index

The indexing mechanism is useful when the file is sorted. In C, we use fseek and in Java we use RandomAccessFile to get the file location.

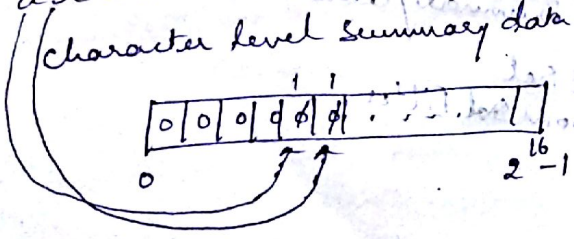
If the file is unsorted, then again load all the contents of the file into RAM and for every record construct a B^+tree and then apply the searching algorithm.

→ B^+tree, Dense Index, Multilevel Index.

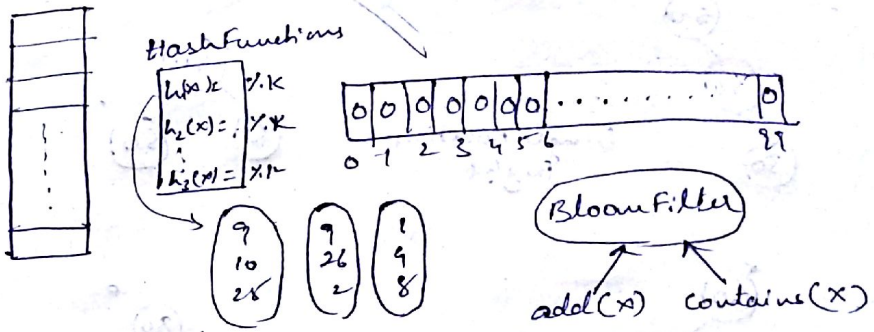
Solution 4 :- Use Summarize data structures.

↳ Bitset

↳ BloomFilter.



Word level Summary data



Sometimes it can result in FalsePositives. In that case we need to consider FPR.

preprocessing time → $O(N/B)$

Query time → $O(1)$

Space complexity → $O(1) \leq c \cdot 1$

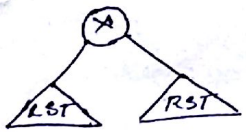
Balanced BST

BST → $O(h)$ → $h \to n \to \log_2 n$

Balanced BST → $O(\log_2 n)$

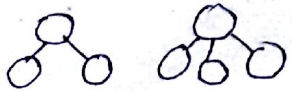
Height balancing

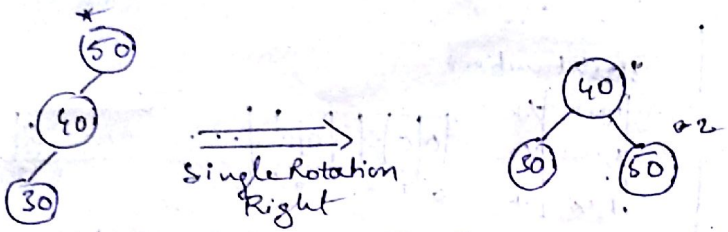
Condition : At every node x,

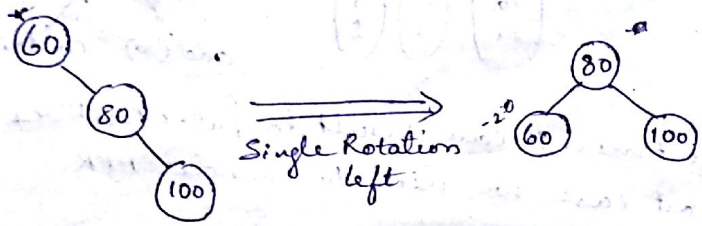


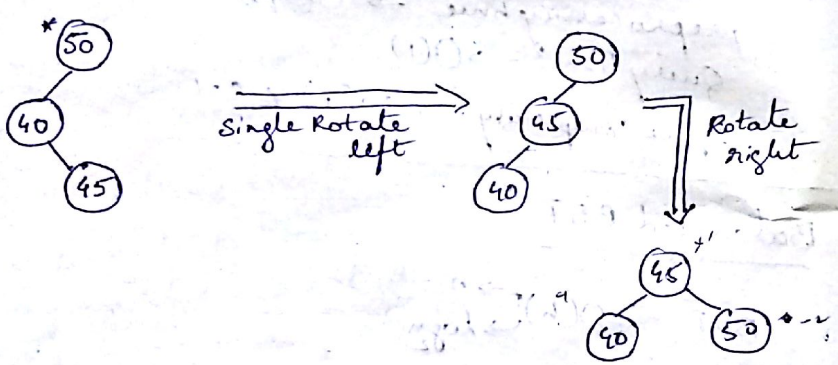
AVL Tree, BST + (Height(LST) − Height(RST)) ≤ 1

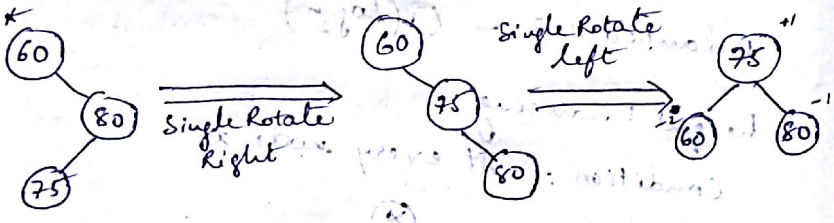
Red Black Tree, Based on 2-3 tree



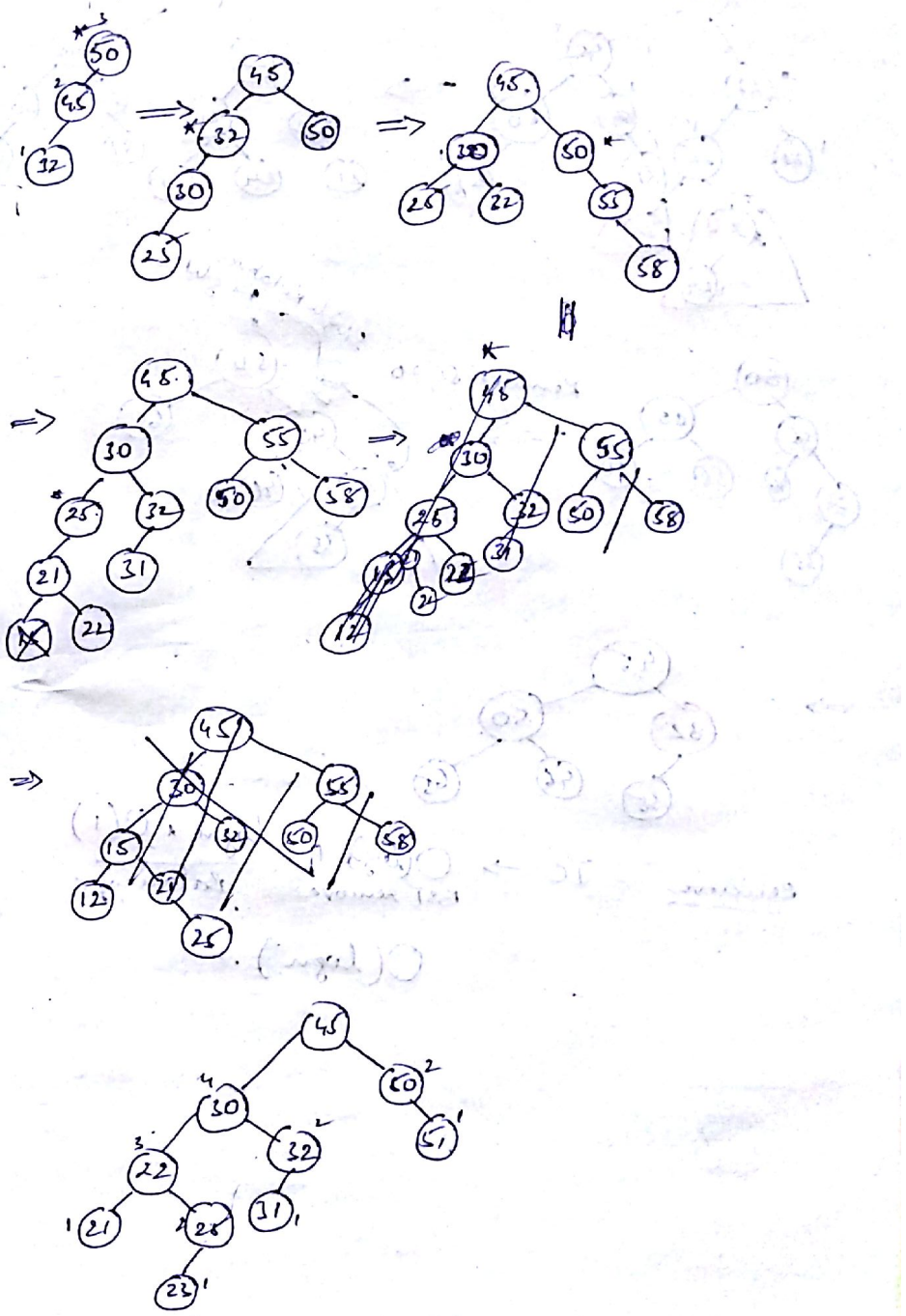


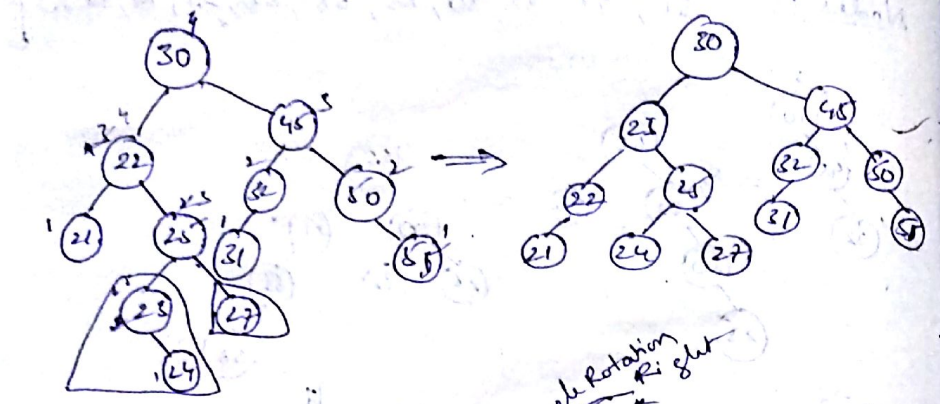


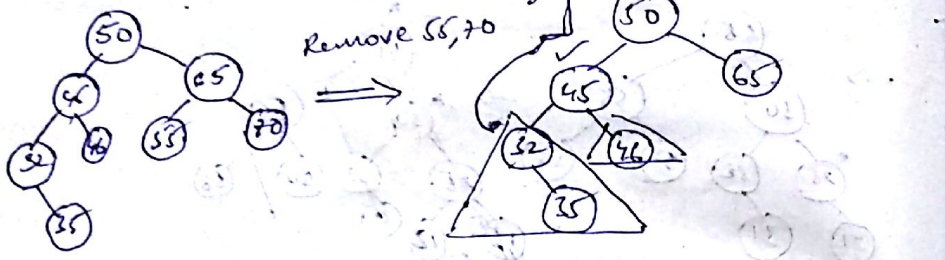


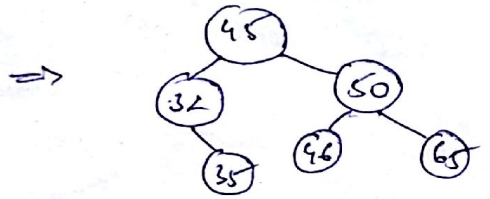


Nodes :- 50, 45, 32, 30, 25, 55, 58, 21, 5, 43, 27, 2









Remove :- TC → O(logn) + log n * O(1)
BST remove — Balancing

O(log n).

* Bitwise Operators

& (AND)
| (OR)
^ (XOR)
<< (Left shift)
>> (Right shift)
>>> (Logical Right Shift)
~ (One's complement)

① 2's complement
② shift operation on -ve number.

(1) &: 5 & 7 = 5

(a) To check whether a bit of any number is 0/1

(b) To clear a bit of any number.

(a) Verify the K^{th} bit of n ⇒ $n \& 2^{K-1} != 0 \; ? \; 1 : 0$

⇒ $(n \& 1 << (K-1)) != 0 \; ? \; 1 : 0$.

(b) clear the K^{th} bit of n ⇒ $(n \& \sim 1 << (K-1))$

(2) <<: 10 << 1 = 20.

$n << K \rightarrow n * 2^K$.

$n * 7 \rightarrow n << 3 - n$

(P1) Find an efficient algorithm to determine whether a given number is in powers of 2 (8) not.

Solution1 :- Count of number of 1's.

Solution2 :- $n \& (n-1) = 0$ ⇒ then it is in powers of 2.

1010

6

(P2.) Find an efficient algorithm to find number of 1's in a given integer.

Solution1 :- Naive Approach.

TC → O(K)

Solution 2 :- Use preprocessing array.

```
int [] ones = {0, 1, 2, 3}
              {0, 1, 1, 2}

int [] ones = {0, 1, 2, 3, 4, 5, 6, 7}
              {0, 1, 1, 2, 1, 2, 2, 3}

int [] ones = {0, 1, 2, ......, 2^7-1}
              {0, 1, 1,  . . .  7}

int countNumber(n)
{ int count = 0;
  while (n!=0)
  {
    count += ones[n & 0xFF];

    n=n>>>8
  }
  return count;

}
```

Solution3

Any number with '&' operation to the number minus '1' will remove the 1's till its right most.

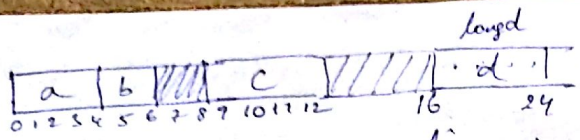
```
  1111010
  1111001
① -------
  1111000
  1110111
② -------
  1110000
  1101111
③ -------
  1100000
  1011100
④ -------
  1000000
  0111111
⑤ -------
  0000000
```

```
int countNumber(n)
{ int count=0;
  while (n!=0)
  {
     n = n & (n-1);
     ++count;
  }
  return count;
}
```

(P3.) Find an efficient algorithm to determine next ~~highest~~ lowest multiple of 8 in a given number.

Memory Banking :- on a hardware bus, we are taken only in 8 bytes.

```
class B
{
  int a; (4)
  char b; (2)      => 12 bytes in reality not 10
  int c; (4)
  long d; (8)
}
```



Boundary level optimization → extra memory is taken to maintain multiple of 2's

12 → 8
16 → 16
35 → 32
23 → 16

The maximum difference is 7 → 111 to have multiple of 8's

n => n & (~7)

=> next lowest

next highest = (n+7) & (~7)

=>

OR operator

(a) To set a bit for a given n.

=> n | (1<<(K-1))

XOR operator

(b) To toggle any bit for a given n.

=> n ^ (1<<(K-1))

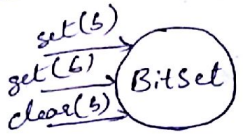
Arithmetic Right Shift (>>)

→ Signed is preserved.

Logical Right shift (>>>)

→ Sign is not preserved.

n>>K = n/2^K.




```
class BitSet
{
    byte[] bs;
    public Bitset()
    {
        bs = new byte[n/8];
    }
    public void set(int K)
    {
        bs[K/8] |= (1<<(K%8-1));
    }
    public boolean get(int K)
    {
    if (bs[K/8] &= (1<<(K%8-1)))
        return true;
    else
        return false;
    }
    public void clear(int K)
    {
        bs[K/8] &= ~(1<<(K%8-1));
    }
}
```

BloomFilter:-

False Positive Rate $= 10^{-6}$

```
public class BloomFilter<T>
{
    float fpr;
    BitSet bs;
    int max, size;
    Hash<T> h;
public BloomFilter(int keys, float fp)
{
    bs = new BitSet(
    h = new Hash();
}
public void add(T key)
{
int[] index = h.getIndices(key);
for(int i=0; i<index.length; i++)
{
    bs.set(index[i]);
}
++size;
}
public boolean contains(T key)
{
int[] index = h.getIndices(key);
for(int i=0; i<index.length; i++)
{
if(bs.get(index[i])
    return false;
}
return true;
}
```

hashfunctions

→ MURMER

→ JENKINS.

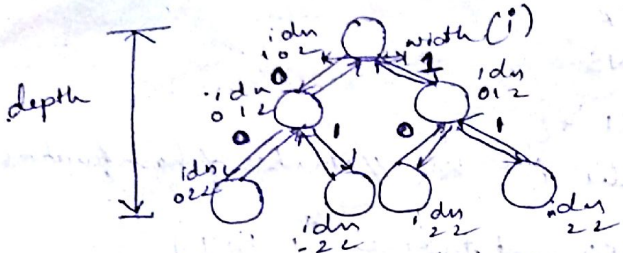
```
class Hash<T>{
private static int K=3; // Number of hashfunctions
public int[] getIndices(T key)
{
    return murmer(key, K);
}
}
```

(P1) Find an efficient algorithm to display all binary sequences of length n:

```
void generateBinary(int d, int n, int[] out)
{
if(d==n){
print(out);
return;
}
for(i=0; i<=1; i++){
    out[d] = i;
generateBinary(d+1, n, out);
}
}
```

d → depth of the for loop

n → Number of for loops.



(P2) Find an efficient algorithm to generate all ternary sequences of length 'n'.

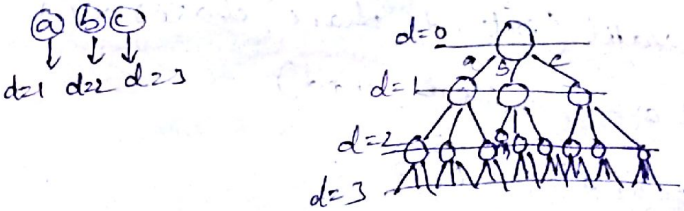
Soln:- just change i=2 in the previous program.

(P3) Find an efficient algo to generate all possible n length sequences of given character array.

i/p:- n=2 [a|d]

a a
a d
d d
d a

```
void generateAllSeq(int d, int n, char[] in,
                    char[] out)
{
  if(d==n)
  {
    print(out);
    return;
  }
  for(i=0; i<n; i++)
  { out[d] = in[i];
    generateAllSeq(d+1, n, in, out);
  }
}
```



(P4) find an efficient algorithm to generate all combinations of length n' for a given character array.

Recursion + Prune logic
= Backtracking

Soln:-

```
void generatePerm(int d, int n, char[] in,
                  char[] out)
{
  if(d==n) {
    print(out, n);
    return;
  }
  for(i=0; i<n; ++i)
  { if(valid(d, in[i], out))
    { out[d] = in[i];
      generatePerm(d+1, n, in, out);
    }
  }
}
```

```
boolean valid(int cd, char c, char[] out).
{
  for(int d=0; d<cd; ++d)
  {
    if(out[d] == c)
      return false;
  }
  return true;
}
```

Heuristic → Elimination of branches.

(P5) n×n chessboard.
Find an efficient algorithm to determine the valid positions for N queens on an n×n chessboard.
No two queens should attack each other.

i/p:- 4 o/p:- [1,3,0,2]
[2,0,3,1]

Soln:-

```
public static void nqueens(int n)
{
  int[] out = new int[n];
  auxqueens(0,n,out);
}
private static void auxqueens(int d, int n, int[] out)
{
  if(d==n){
    print(out);
    return;
  }
  for(int c=0; c<n; ++c)
  {
    if(valid(d,c,out))
    {
      out[d] = c;
      auxqueens(d+1,n,out);
    }
  }
}
private boolean valid(int cq, int cc, int[] out)
{
  int q;
  for(q=0; q<cq; q++)
  {
    if(cc==out[q] || (cq-q)==Math.abs(cc-out[q]))
    {
      return false;
    }
  }
  return true;
}
```

(current queen) (previous queens positions) (current column)

(P6) Sukodu Problem
Fill the unoccupied cells such that
(a) Each row must have 1-9 without repetition
(b) Each column must " "
(c) Each non-overlaping 3×3 grid " "

```
public static void sudokusolver(int[][] a)
{
    auxsolver(0,0,a);
}
public static void auxsolver(int r, int c, int[][] a)
{
    if(r==9){
    print(a);
    return;
    }
    if(a[r][c]!=0){
    if(c==8) r=r+1;
    c=(c+1)%9;
    auxsolver(r,c,a);}
    else{
    for(d=1;d<9;d++)
    {
    if(valid(r,c,d,a))
    {a[r][c]=d; oldr=r; oldc=c;
    if(c==8) r=r+1;
    c=(c+1)%9;
    auxsolver(r,c,a);
    }
    }
    a[oldr][oldc]=0;
}
```

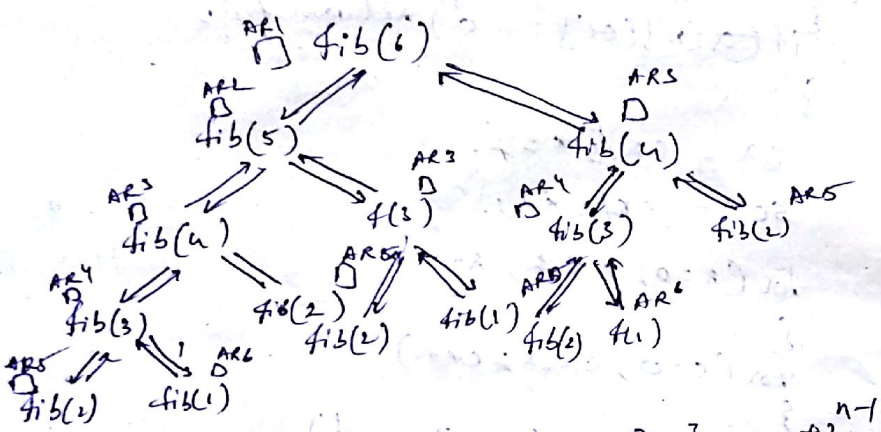
```
private static boolean valid(int cr, int cc, int d,
                             int[][] a)
{   int r, c, sr, sc;
    for(c=0;c<9;c++)
    {if(a[cr][c]==d) return false;
    }
    for(r=0;r<9;r++)
    {if(a[r][cc]==d) return false;
    }
    sr=cr/3*3;
    sc=cc/3*3;
    for(r=0;r<3;r++)
    {
    for(c=0;c<3;c++)
    {if(a[sr+r][sc+c]==d)
    return false;
    }
    }
    return true;
}
```

For DP,

(P1) Find an efficient algorithm to compute n^{th} fibonacci number

Solution1:- use Recursion Reduction

$$f(n) = \begin{cases} f(n-1)+f(n-2) & n>2 \\ 1 & n<=2 \end{cases}$$



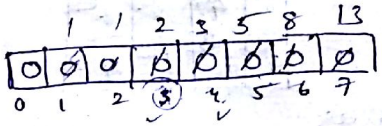
Number of Nodes: $1+2+2^2+2^3+\ldots+2^{n-1}$

$$= \frac{1(2^n-1)}{2-1} = 2^n-1$$

Solution 2:- Recursion + Memory = Memoization

```
public static int fib(int n)
{
    int m[] = new int[n+1];
    return auxfib(n,m);
    return m[n];
}
```

```
private static int auxfib(int n, int[] m)
{
    if(n<=2) return 1;
    if(m[n] != 0) return m[n];
    m[n] = auxfib(n-1) + auxfib(n-2);
    return m[n]; // returns at base case first
}
```



TC → O(n)

SC → O(n+n) = O(n)

Recursion + memory

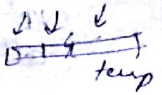
This is also called as top-down memoization

Solution3:- Non-Recursive + Memory.

```
public static int fib(int n)
{
    int[] m = new int[n+1];
    m[1] = m[2] = 1;
    for(i=3; i<=n; i++)
    {
        m[i] = m[i-1] + m[i-2];
    }
    return m[n];
}
```

TC → O(n);

SC → O(n);

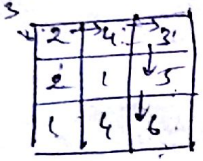


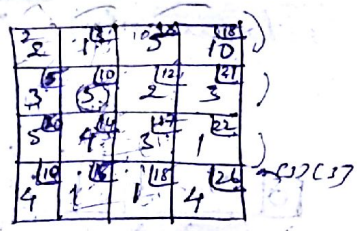
8

This is called as Dynamic Programming.

→ Optimized DP → SC → O(n)
TC → O(n)

(P2) Puzzle





Start at top left corner.

move → right / down | → 1 cell at a time

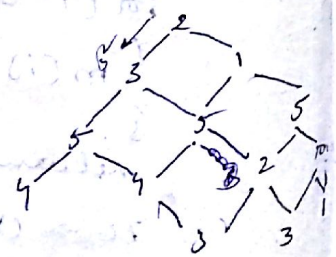
How many (maximum) apples can be collected?

optimization

$$M(i,j) = \max\left[M(i-1,j), M(i,j-1)\right] + a[i-1][j-1]$$

Maximum no. of apple can be collected at cell(i,j)

If $i = 0$ or $j = 0$
M returns 0.



Dynamic Programming

Solution1 : Use Recursion.

```
public static int findMaxApples(int[][] a)
{
    return auxMaxApples(a, a.length, a.length);
}
private static int auxMaxApples(int[][] a, int i, int j)
{
    int l = auxMaxApples(a, i-1, j);
    int r = auxMaxApples(a, i, j-1);
    return Math.max(l, r) + a[i-1][j-1];
}
```

TC → $\leq 1 + 2 + 2^2 + \cdots + 2^{2n} \rightarrow O(2^n)$.
SC → O(n)

Solution2 Use Recursion with Memory (Memoization)

```
public static int findMaxApples(int[][] a)
{
    int[][] mem = new int[a.length+1][a.length+1];
    return auxMaxApples(a, a.length, a.length, mem);
}
private static int auxMaxApples(int[][] a, int i, int j, int[][] mem)
{
    if (i==0 || j==0)
        return 0;
    if (mem[i][j] != 0) return mem[i][j];
```

```
int l = auxMaxApples(a, i-1, j, mem);
int r = auxMaxApples(a, i, j-1, mem);
mem[i][j] = Math.max(l, r) + a[i-1][j-1];
return m[i][j];
}
```

TC → $(n+1)*(n+1) \simeq O(n^2)$

SC → $O(n^2+n) \simeq O(n^2)$

Solution3: Use DP → Non-Recursive + Memoly.

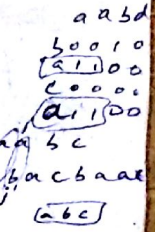
```
public static int findMaxApples(int[][]a)
{
int[][] mem = new int[a.length+1][a.length+1];
for(i=0; i<mem.length; ++i)
{
mem[0][i] = 0;
mem[i][0] = 0;
}
for(i=0; i<a.length; ++i)
{
for(j=1; j<a.length; ++j)
{
mem[i][j] = Math.max(mem[i-1][j],
                     mem[i][j-1]) + a[i-1][j-1];
}
}
return mem[a.length-1][a.length];
}
```

(P3) Find an efficient algorithm to compute the length of the longest common subsequence of given strings.

Solution1:-
Subsequence = A sequence of characters with relative order.

Substring = A sequence of continuous characters.

if:- ① a b a c b a a c
② a a b c



Solution1:- All possible subsequences.

TC → $2^n \times 2^m \times t$ (common path ans function).

Solution2:- Explore all subsequences of smaller string where $m<n$.

TC → $O(2^m \times n)$

Solution3:- Explore all possible alignments.

$$L(i,j) = \begin{cases} 0 & \text{if } i==0 \text{ || } j==0 \\ 1+L(i-1,j-1) & \text{if } s[i-1]==s2[j-1] \\ \max\left(L(i-1,j), L(i,j-1)\right) & \text{if } s[i-1]+s2[j-1] \end{cases}$$

L(i,j) = length of LCS of s[1...i] & s2[1...j]

use DP

```
public static int findLCS(String s1, String s2)
{
    int n = s1.length, m = s2.length;
    int [][] mem = new int[n+1][m+1];

    for (int j=0; j<=m; ++j)
        mem[0][j] = 0;
    for (int i=0; i<n; ++i)
        mem[i][0] = 0;
    for (int i=0; i<n; i++)
        for (int j=1; j<m; j++)
            if(s1.charAt(i-1) == s2.charAt(j-1))
            {
                mem[i][j] = 1 + mem[i-1][j-1];
            }
            else
            {
                mem[i][j] = Math.max(mem[i-1][j],
                                     mem[i][j-1]);
            }
    return mem[n][m];
}
```

0,0
1,1
2,2
3,3

TC → O(nm)

SC → O(nm)

- → Linear Algebra
- → Stochastic Theory.
- → Optimization Theory.
- → Information Theory.
- → Calculus.

Mathinsight.org.
Khan's academy ✓

(P4) Find an efficient algorithm to compute the length of longest palindrome subsequence in a given string.

S = a b a c b a

a a
a b a
a a a
a b a b a

O/p :- 5

Solution1 :- Find all subsequences & find if the subsequence is palindrome or not.

Solution2 :- Find LCS(S, S^R)

TC → $O(n^2)$

SC → $O(n^2)$.

Solution3 :- Use Recursion

$$P(i,j) = \begin{cases} 2 + P(i+1, j-1) & \text{if } s[i-1] == s[j-1] \quad (i == j) \\ \text{Max}(P(i+1,j), P(i,j-1)) & \text{if } s[i-1] \neq s[j-1] \end{cases}$$

(P5) How to design Address book. (in email)/mobile

Trie (Retrieval) → Data Structures

↳ Routing s/w
↳ Searching.

TrieSet / TrieMap API

- add(Key)
- remove(Key)
- contains(Key)
- FindAllKeysWithPrefix(prefix)
- display() / Iterator()
- size()
- FindAllWildCard(prefix)

Trie Implementation

- → HashSet/TreeSet
- → R-Way Trie Implementation (Space ↑)
- → TST (Ternary search Tree)
- → List

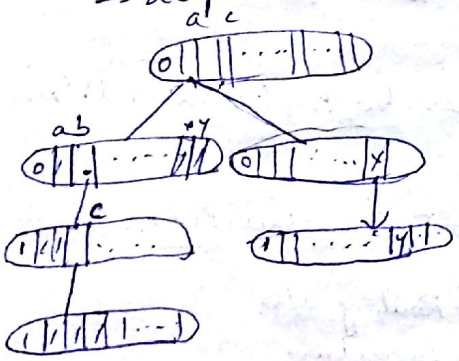
→ No. of children at each Node

R-Way Implementation

→ Good for binary keys (0/1)

→ 26 pointers

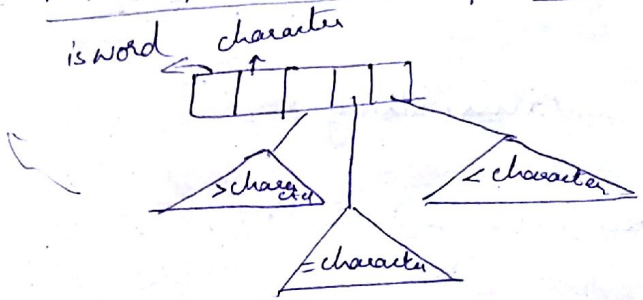
ab, abc, axy, pqr, cab, xyz, zxy.

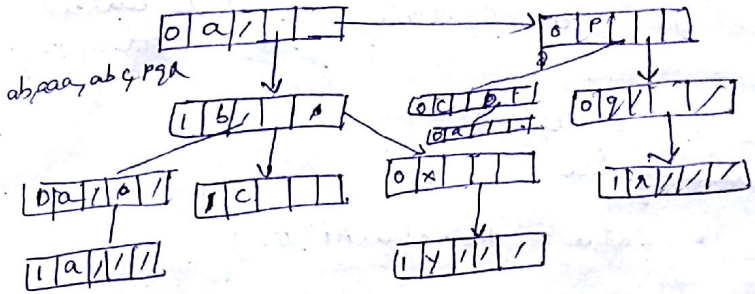


→ Storage Issue

→ Time O(1)

TST Implementation | Radix Tree





TC → $O(26 * K) \approx O(K)$

K → length of key

```
public class TST
{
    class TSTNode
    {
        char data;
        boolean isWord;
        TSTNode left;
        TSTNode middle;
        TSTNode right;

        TSTNode(char char)
        {this.data = char;
        }
    }
    private TSTNode root;
    private int size;
```

```
public TST()
{
}
public boolean contains(string key)
{
    return auxContains(root, key);
}
private boolean auxContains(TSTNode current,
                            string key)
{
  if(current == NULL)
      return false;
  if(current.data == key.charAt(0))
  {
    if(key.length() == 1)
      return current.isUsed == 1;
    else
      return auxContains(current.middle,
                         key.substring(1));
  }
  else if(current.data > key.charAt(0))
    return auxContains(current.left, key);
  else
    return auxContains(current.right, key);
}
public void add(string key)
{
root = auxAdd(root, key);
}
```

```
private TSTNode auxAdd(TSTNode current, string key)
{
  if(current == NULL)
  {
    current = new TSTNode(key.charAt(0));
  }
  if(current.data == key.charAt(0))
  {
    if(key.length == 1)
    {
      current.isWord = 1;
      return current;
    }
    else
    {
      current.middle = auxAdd(current.middle,
                              key.substring(1));
    }
  }
  else if(key.charAt(0) < current.data)
  {
    current.left = auxAdd(current.left, key);
  }
  else
  {
    current.right = auxAdd(current.right, key);
  }
  return current;
}
```

```
public List<String> findAllKeysWithPrefix(String pref)
{
    char[] out = new char[100];
    List<String> keys = new LinkedList<String>();
    current = find the correct prefix(root, prefix);
    collectAllKeysforPrefix(current.middle,
                            out, prefix, keys);
    return keys;
}
private void collectAllKeysForPrefix(TSTNode current, int d, char[] out, String
        prefix, List<String> key)
{
    if(current==NULL) return;
    collectAllKeysForPrefix(current.left, d, out, prefix, keys);
    out[d] = current.data;
    if(current.isWord==1)
    { str = prepareString(prefix, out[d]);
      keys.add(str);
    }
    collectAllKeysforPrefix(current.middle, d+1, out, prefix, key);
    collectAllKeysforPrefix(current.right, d, out, prefix, key);
}
```

Sorting: → In memory
→ File

Arranging elements in ascending/descending order.

ip:- 10 8 15 12 11 7 5 2

op:- 2 5 7 8 10 11 12 15

Solution 1:- Use Balanced BST, InOrder Traversal.

TC → O(n log n)
SC → O(n).

$\log 1 + \log 2 + \log 3 + \dots \log(n-1)$
$= \log(1 \cdot 2 \cdot 3 \dots (n-1))$
$= \log(n-1)! \simeq n \log n.$

Solution 2:- Use Heap DS

TC → O(n log n)
SC → O(1) ← In Place Order.

Solution 3:- At i^{th} step find the i^{th} rank element & swap it with the i^{th} position element.

(P1) 2 8 15 12 11 7 5 10 (n-1)

(P2) 2 5 15 12 11 7 8 10 (n-2)

(P3) 2 5 7 12 11 15 8 10

(P4) 2 5 7 8 11 15 12 10

(P5) 2 5 7 8 10 15 12 11

(P6) 2 5 7 8 10 11 12 15

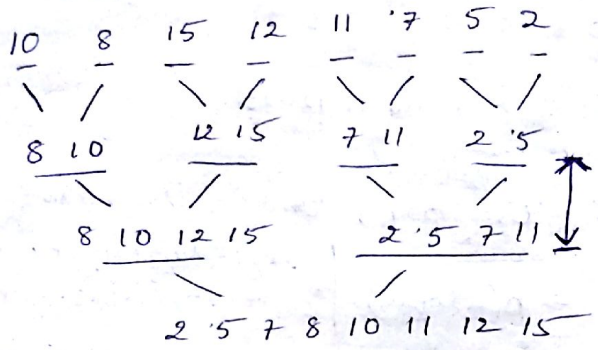
(P7) 2 5 7 8 10 11 12 15

In-place sorting algorithm

TC → $O(n^2)$

SC → $O(1)$

Solution 4: Use Merge Principle



Dividing → $\log n$.

Comparisions → $n/2 + n + n \simeq n$

∴ TC → $O(n \log n)$

But while merging, in place sorting is possible by using an extra array & in place. So in-place merging is possible only by rotation so the order is preserved.

Subproblem :- Given an array of size n. Find an efficient algorithm to rotate the array to right by 'K' times.

i/p :- 10 5 8 1 4 2. K=2

o/p :- 2 4 4 2 | 10 5 8 1.

Idea-I Naive Approach

TC → $O(K * n)$

SC → $O(1)$

Idea-II Reverse the array, 2 levels. then split it at K levels

2 4 | 1 8 5 10

split 4 2 | 10 5 8 1 K=2

TC → $O(n)$ | 2 4 4 2

SC → $O(1)$

Idea-III :- Reverse the array by K levels every time

2 4 4 2
10 5 8 1 4 2 } K=1

→ 4 2 | 8 1 | 10 5 } K=2 ✓

→ 4 2 | 10 5 8 1

TC → $O(n)$

SC → $O(1)$.

∴ For final problem,

TC → $O(n^2 \log n)$ | If space can be taken TC → $O(n \log n)$

SC → $O(1)$ | SC → $O(n)$

QuickSort

- Element → Find the correct position.
  (Opposite of selection element)

[while swaping to its correct position, move all the elements which are lesser to its left and greater number should be to right]

If the size of the problem is divided into half then,

① —— $n-1 \simeq n$

② —— $n/2 + n/2 \simeq n$

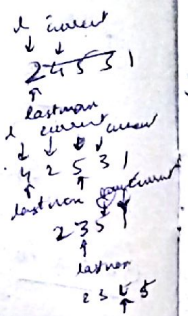
④ —— $n/4 * 4 \simeq n$

⑧ —— $n/8 * 8 \simeq n$

⋮

n —— $1 * n \simeq n$

$\therefore n * \log n$.



Selection of the right element is very important

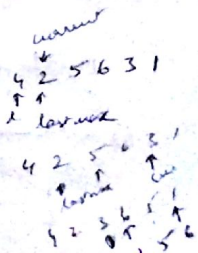
pivot selection ← First/last element — $O(n^2)$
- Random element
- Median of three (P)
- Median of Median (T).

Median — Middle element of a sorted array.

TC — $O(n \log n)$ / $O(n^2)$

SC — $O(\log n)$

```
public static void qsort(int[] a)
{
        auxSort(a, 0, a.length-1);
}
private static void auxSort(int[]a, int l,
                            int r)
{
    if(l<r){
  int p = partition(a, l, r);
    auxSort(a, l, p-1);
    auxSort(a, p+1, r);
    }
}
private static int partition(int[]a, int l, int r)
{
    int pi = getPivotIndex(a, l, r);
     swap(a, l, pi);
   for(int lastmax=l, current=l+1; current<=r; ++current)
   {
     if(a[current]<a[l])
       swap(a, ++lastmax, current);
   }
   swap(a, l, lastmax);
   return lastmax;
}
```



Random

Bayes's Algorithm – Spam Reduction.

i/p

0	1	2	3	4
10	50	30	40	70

1/5 – rand(0,4) = 1
1/4 – rand(1,4) = 3
1/3 – rand(2,4) = 4
1/2 – rand(3,4) = 3
1/1 – rand(4,4) = 4

```
public static void shuffle(int[] a)
{
  Random r = new Random();
  for (i=0; i<a.length; ++i) {
  int ri = r.getRandom(i, a.length-1);
     swap(a,i,ri);
  }
}
```

(0 - 1) ⟶ (a - b)

Rand(0-1) * (a-b) + a

∴ Math.random(0,1) * (Max-Min) + Min.

(a-b) – (0-1)

(a-b) ⟷ (c-d) ↳ Rand(a-b) - a / b-a

↳ (0-1) ⟶

Random Sampling [1, 2, 3, 4, 5, 6, 7] ↳ [1, 2, 3]

Streaming data

(1) Find a random number in stream of data [upper bound not given] ⟶ 1/K for each element

(2) Find a random sample of size K in the stream.

Random

pick a Random element in the stream data.

Solution: Pick the K^{th} element with probability 1/K.

```
public static int randomInStream(int K, int elem,
int oldelem)
{
  if (K == 1) return elem;
  Random r = new Random();
  if (r.nextInt(K) == K) return elem;
  return oldelem;
}
```

10, 20, 30

10 — 1 | 1/2

20 — 1/2 * 2/3 = 1/3

30 — 1/2

Sorting (Insertion) - Arranging cards.

10 5 8 7 6 4

TC → $O(n^2)$

SC – $O(1)$

PASS1 5 10 | 8 7 6 4

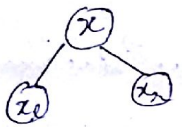
PASS2 5 8 10 | 7 6 4

PASS3 5 7 8 10 | 6 4

PASS4 5 6 7 8 10 | 4

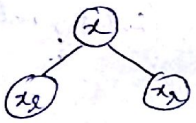
PASS5 4 5 6 7 8 10

Heap – A complete Binary Tree with order property



At every node,

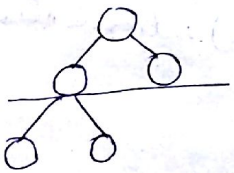
$x < x_l$ & $x < x_r$ — Min Heap



At every node,

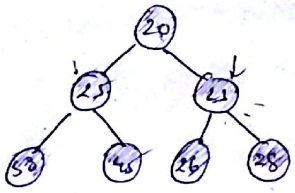
$x > x_l$ & $x > x_r$ — Max Heap

Complete Binary Tree



} All nodes to left & right are filled

} The last level of nodes should be filled from left to right



⇒

20	25	23	50	45	26	28
0	1	2	3	4	5	6

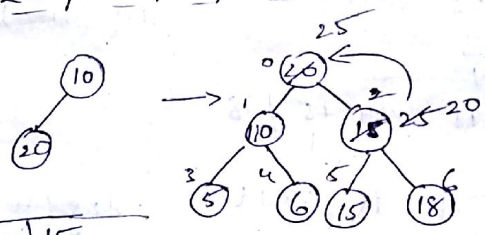
At every i^{th} node

left → $2*i+1$

right → $2*i+2$

parent → $\left(\frac{i-1}{2}\right)$

10, 20, 15, 5, 6, 18, ~~18~~ 25



20	10	15	5	6	18	25
0	1	2	3	4	5	6

Addition of an element → $O(\log_2 n)$

While Removing the node [at root level]

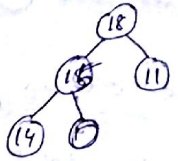
Copy the last child element to its root & then maintain order.

25	18	17	14	16	17
0	1	2	3	4	5

Maxheap → Ascending order.

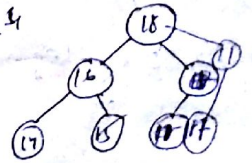
18	15	11	14	16	17
0	1	2	3	4	5

← pass1



18	16	11	14	15	17
0	1	2	3	4	5

← pass 4



18	16	17	14	15	11

⇒ $\log 1 + \log 2 + \cdots + \log(n-1)$

$\simeq n \log n$

Step 2:- Swap last element to with root node and remove the rootnode (actual).

step1 11 16 17 14 15 | 18 — $\log(n-1)$

step2 ↓↓

17 16 11 14 15 | 18

step1 15 16 11 14 | 17 18 — $\log(n-2)$

step2 ↓↓

16 15 11 14 | 17 18

step1 14 15 11 | 16 17 18

step2 ↓↓

15 14 11 | 16 17 18

14 11

11 14 | 15 16 17 18 — $\log 2$

Final 11 14 15 16 17 18

$\therefore \log(n-1)! \simeq n \log n$

$\Rightarrow$ TC $\rightarrow O(n \log n)$

SC $\rightarrow O(1)$

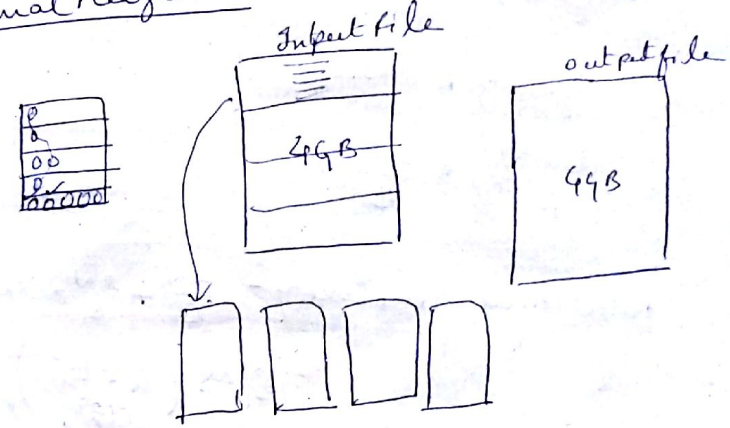
Quicksort is preferable than Heapsort because quicksort takes $n \log n$ but heapsort takes $2*(n \log n)$.

For linked list sorting $\rightarrow$ MergeSort

File Sorting Select * from R order by R.a.

External MergeSort.

Cache



Analysis:-

I/O $\rightarrow$ Block.

N $\rightarrow$ File size

B $\rightarrow$ Block size

step 1:- Sorting

$\frac{N}{B}$ $\leftarrow$ In operations $\therefore \frac{2N}{B}$ Block I/O's.

$\frac{N}{B}$ $\leftarrow$ Out operations

step 2:- Merge sort M $\rightarrow$ Memory size.

$\rightarrow \frac{N}{M}$ File Temps are generated

Only 1 step of merging is required

then $\frac{2*N}{B}$.

Case 1 :-

step 2 :-

Merge step 1 = $2N/B$ $\quad \frac{N}{M} * \left(\frac{2N}{B}\right)$

Merge step 2 = $2N/B$

⋮

Merge step n = $2N/B$

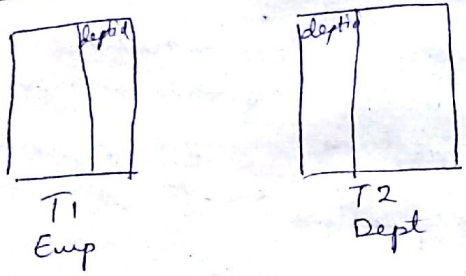
$\therefore \log_{\left(\frac{M}{B}\right)}\left(\frac{N}{M}\right) * \frac{2N}{B}$

$\therefore$ Total time complexity = $\frac{2N}{B} + \frac{2N}{B} * \log_{\left(\frac{M}{B}\right)}\frac{N}{M}$

$= \Theta\left(\frac{2N}{B}\left(\log_{\left(\frac{M}{B}\right)}\left(\frac{N}{M}\right)\right)\right)$

Join Problem :-

Case



|Emp| >> |Dept|

Ideal :- ~~Replicated~~ In Memory buffered join.

Since Dept table is small in size, store them into a Hashmap and then store each block from employee check the hashmap and make a new record. Finally print it onto the console.

Ideal :- Sort + Merge Join.

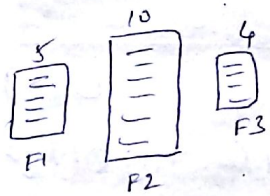
Two files are comparative in size.

(1) Sorted by join key.

(P1) Find an efficient algorithm that determines minimum no: of record movements required to merge n sorted files of size $S_1, S_2, \ldots, S_n$ respectively.

Note :- Can only merge two files at a time.

Ip :-



$(F_1F_2)F_3 = 15+19 = 34$

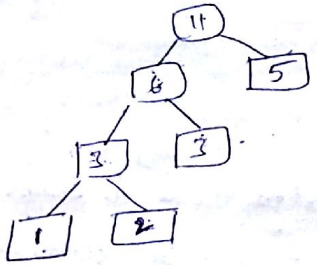
$F_1(F_2F_3) = 14+19 = 33$

$(F_1F_3)F_2 = 9+19 = 28$ ✓

Op :- 28

Ideal :- Sort the files in ascending order and take the smallest of the two files and merge them

1 2 3 5



$11 + 6 + 3 = 20$

$= (1*3) + (2*3) + (3*2) + (5*1)$

$= 3 + 6 + 6 + 5$

$= 20.$

Weighted path length.

Proof by exchange argument

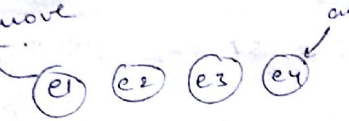
If 2 is moved to 3, 3 moved to 5

Try to move values in weighted path length

and prove that they cannot beat optimal solution.

Priority Queue: add(x), removeMin(), findMin/Max

A queue in which elements have priority.



Priority based on insertion.

Implementations of Priority Queue:

	findMin	removeMin	add
Sorted list	O(1)	O(1)	O(n)
TreeSet	O(logn)	O(logn)	O(logn)
Heap	O(1)	O(logn)	O(logn)

→ Remove 2 elems & add merged add

TC → add n + (3logn + 3log n-1 + 3log n-2 + ... + 3log 2)

→ n logn + ( )

Incremental

∴ O(n logn)

SC → O(1)

(P2) Find an efficient algorithm to find maximum profit that can be obtained using following items by filling a given bag of size 'W'.

	i1	i2	i3	
✓ qty	10	5	7	→ Greedy ✗
profit	5	10	15	→ Dynamic Programming ✓

→ 0/1 Knapsack problem.

(P3) Find an efficient algorithm that returns minimum number of coins required to get sum 'S'.

Note:- No:- of coins ~~required~~ in each denomination 'oo'

if:- two 1, 5, 10

S = 17

$S[i] = \min \begin{cases} S[i] \\ 1 + \end{cases}$

$K[i,S] = \min \begin{cases} K[i,S] \\ 1 + K[i-1, S-\text{in}[i]] \end{cases}$

Greedy Pattern:-

	i1	i2	i3	...	in
qty	q1	q2	q3	...	qn
Profit	p1	p2	p3	...	pn

Recursion: Explore all possibilities < Pick / Don't Pick

The max profit

$$K(i,J) = \begin{cases} \max\{K(i-1,J), K(i-1,J-q[i-1]) + p[i-1]; & \\ K(i-1,J) & \text{if } q[i-1] > J \\ 0 & \text{if } i=0 \ \& \ J=0 \end{cases}$$

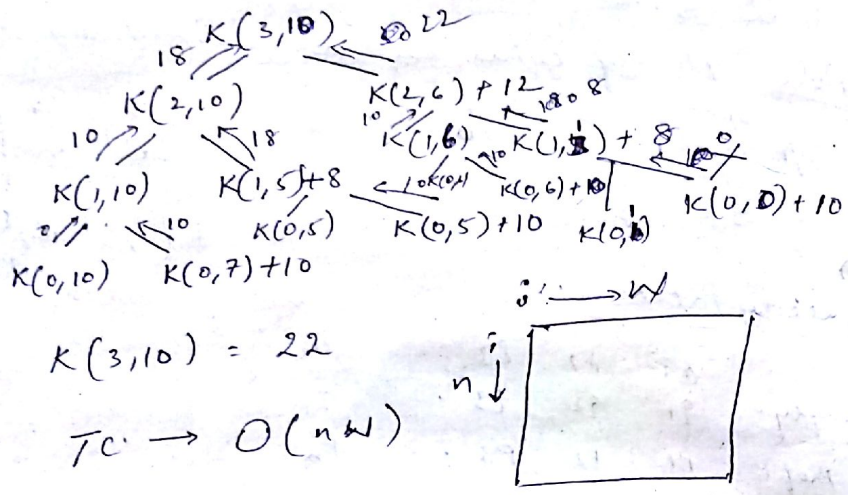
base case

Goal: K(n, W)

$$K[i,S] = \begin{cases} \min\{K[i+1,S], \\ \quad 1 + K[i-1, S-\text{in}[i-1]] \\ 0 & \text{oo}, S=0 \\ 10000 & i=0, S \neq 0 \end{cases}$$

$K(n, W)$

	d_1	d_2	d_3
g_i	3	5	4
	10	8	12



$K(3,10) = 22$

$TC \rightarrow O(nW)$

(P4) Find an efficient algorithm that determines minimum number of conference rooms required to arrange 'n' meetings.

meeting $\rightarrow$ (starttime, endtime)

n = 7

i/p :-

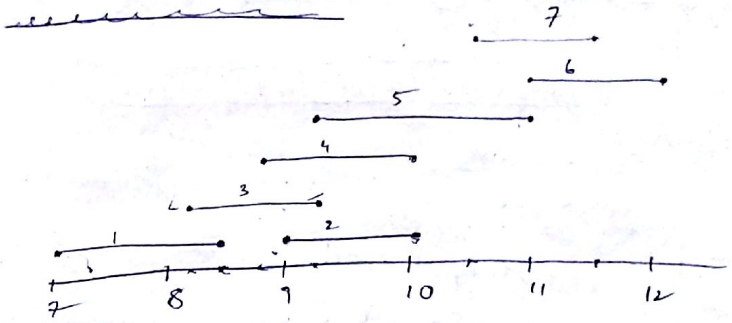
1. 7:00 AM – 8:30 AM
2. 9 – 10:00 AM
3. 8:15 – 9:15 AM
4. 8:45 – 10 AM
5. 9:15 – 11:00 AM
6. 11:00 AM – 12:00 PM
7. 10:30 AM – 11:30 AM

o/p :-

R1 – 1, 2, 7

R2 – 3, 5, 6

R3 – 4



R1 – 1, 2, 6
1, 2, 7
1, 4, 6
3, 5, 6

R2 – 3, 5, 6

R3 – 4

Solution1 :- Try to check every overlapping time schedules & the maximum is the answer

$TC \leq 2n * n \simeq O(n^2)$

$SC \rightarrow O(1)$

Solution 2 :- Greedy choice. At every end point, if end point represents start time, allocate new room, iff all rooms are occupied.

If end point represents end time, take out room.

→ Find maximum number of rooms required, throughout the process.

$TC \rightarrow O(n \log n)$

$SC \rightarrow O(1)$.

(P1) Find an efficient algorithm to determine min & max of given array of size 'n'.

i/p :- | 50 | 40 | 60 | 80 | 10 | 15 | 35 | 25 | 140 |

o/p :- min = 10
max = 140

Solution1 Naive Approach

$\leq 1 + 2(n-2) = 2n-3$ comparisons.

TC → O(n)
SC → O(1)

Solution2 :- Pairwise Comparisons → 3 pair.

$1 + \left(\frac{n-2}{2}\right)3 = \frac{3n-4}{2} \approx 1.5n - 2$

TC → O(n)
SC → O(1)

Solution3 :- Triplet comparisons → 5'

$1 + \frac{n-2}{3} * 5 \simeq 1.67n - 7$

(P2) Find an efficient algorithm to determine second smallest/largest element in a given array.

i/p :- | 10 | 5 | 8 | 16 | 15 | 12 | 11 | 20 |

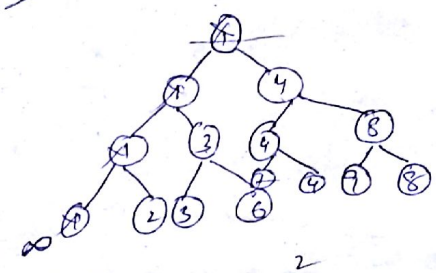
o/p :- 8

Naive :- | 5 | 8 | 8
1st 2nd

$\leq 1 + 2(n-2)$

$\simeq 2n - 3$

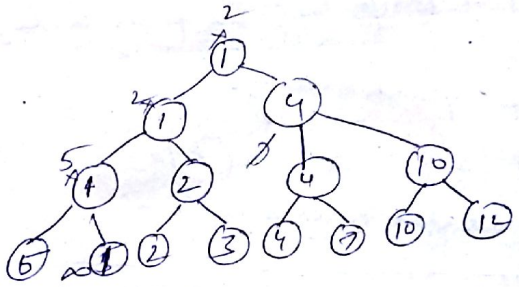
Solution2 :- Use Tournament Approach.



$(n-1) + (\log_2 n - 1)$
For smallest For 2nd smallest

Code it

Eg :-



1	1	4	1	2	4	10	5	1	2	3	4	7	10	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

(P3) Find an efficient algorithm to determine K^{th} smallest element in a given array.

Select(K).

Solution1 :- Sort the array & pick $(K-1)^{th}$ index element

TC → O(nlogn)
SC → O(1)

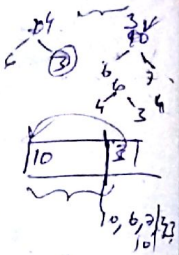
Solution2 :- Use tournament tree

TC → (n-1) + Klogn
SC → O(n)

Solution3 :- Use TreeSet

nlogn + log4 ≃ O(nlogn)
SC → O(n)

Solution 4 :- Use ~~min~~ Max heap · Min Heap



→ batch

$n + K \log n$

$O(n + K \log n)$

$SC \rightarrow O(1)$

Solution 5 :- Use Max Heap · → construct K elements to heap.

$K + (n-K) \log K$

Solution 6 :- Use partitioning.

$TC \rightarrow O(n)$, $SC \rightarrow O(1)$

↳ Random Pivot selection

Graph Problem

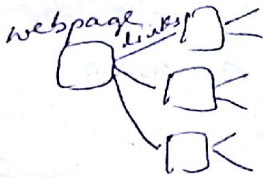
Graphs in real world ?

1. Web Graph — Crawlers (Traversing Graph)
2. Social n/ws — Friends n/w.
3. Route Planning — Flight
4. Project Planning — Critical Path Finding.
5. Build-tools (Ant / Maven / Make)

Ex Google search, gather entire web data, relevant results, Indexes

↳ Machine learning

↳ Information Retrieval.



Graph Databases

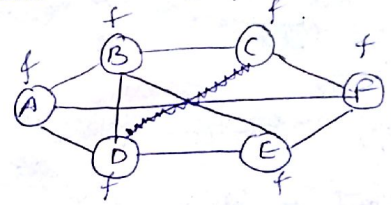
↳ Neo4S

↳ Titan.

P1. Find an efficient algorithm to traverse a given graph $G(V, E)$

Directed Graph } Weight / Non-weight

Undirected Graph }



O/P :-

O/P :- B C F E D A

3 Issues

(a) No start Node

(b) Cycles ~ (Loops)

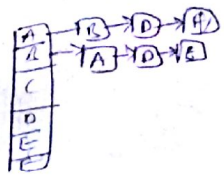
(c) Disconnected

Solution 1 :- Recursion Pattern - DFS

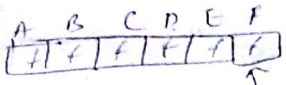
Representation of Graphs (In Memory)

Adjacency Matrix *

Adjacency List *



	A	B	C	D	E	F
A	0	1	0	1	0	1
B	1	0	1	1	1	0
C						
D						
E						
F						



Adjacency Matrix

Matrix

TC → $O(V + V^2) \simeq O(V^2)$

SC → $O(V + V) \simeq O(V)$

↑ Depth of Recursion

Adjacency List

TC → $O(V + E)$

SC → $O(V + V) \simeq O(V)$

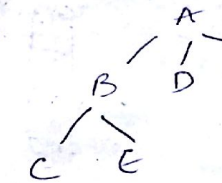
Sparse Graph / Dense Graph

If 'V' vertices are there then maximum number of Edges are $\leq V^2$

If Edges are close to V^2 — Dense Graph.

Graphs are stored as list in a file/database

Solution 2:- Adhoc Strategy - Level Strategy - BFS



→ A B D F C E

Queue for storing BFS

	Adj Matrix	Adj List
TC →	$O(V+V^2)$ $\simeq O(V^2)$	$V+E$ $\simeq O(V+E)$
SC →	$O(V+V)$ $O(V)$	$O(V+V)$ $\simeq O(V)$

Apache Nutch → Free software for crawling graphs/ Graphs / Ranking / Crawlers

Graph

```
public class Graph
{
  private int[][] adj;
  private int V;
  private int E;
  public Graph(int V)
  {
    adj = new int[V][V];
    this.V = V;
    this.E = 0;
  }
  public void traverse()
  {
    boolean visit = new boolean[V];
    for (u=0; u<V; ++u)
    if (visit[u] == false)
    dfs(u, visit);
  }
  private void dfs(int u, boolean[] visit)
  {
    visit[u] = true;
    S.o.p(u);
    for (w=0; w<V; ++w)
    {
      if (adj[u][w] == 1) && visit[w] == false)
      {
        dfs(w, visit);
      }
    }
  }
}
```

```
public void addEdge (int v, int u)
{
    adj [v][v] = 1;
    adj [v][u] = 1;
}
```
} undirected graphs

(P2) Find an efficient algorithm to determine whether a given graph is Bi-Partite (or) not

(No odd length cycles)

Solution:- Initially Assign '0' when it is visited for 1st time then assign either '1' or '2'

If i is the parent then assign 2 otherwise vice versa.

(P3) Find an efficient algorithm whether given graph has cycles or not.

Solution:- Modify either DFS or BFS

```
boolean                                  int p
private boolean dfs(int u, boolean[] visit, int p)
{
    visit[u] = true;
```

```
public boolean hasCycle()
{
    boolean[] visit = new boolean[v];
    for (u=0; u<v; ++u)
        if (visit[u] == false)
            if(dfs(u,u,visit)) return true;
            return false;
}
private boolean dfs(int u, int p, boolean[] visit)
{
    visit[v] = true;
    for (w=0; w<v; ++w)
    {
        if (adj[u][w] == 1) {
            if (visit[w] == false)
            {
                if (dfs(w,u,visit)) return true;
            }
            else if (p != w)
                return true;
        }
    }
    return false;
}
```

```
public boolean isBipartite()
{
  int[] visit = new int[V];
  for (u=0; u<V; u++)
  {
    if (visit[u]==0)
      if (!dfs(u, 1, visit))
        return false;
  }
  return true;
}
private boolean dfs(int u, int p, boolean[] visit)
{
  visit[u] = p;
  for (int w=0; w<V; ++w)
  {
    if (adj[u][w]==1)
      if (visit[w]==false)
      {
        if(!dfs(w, p==1?2:1, visit)) return false;
      }
      else if (visit[w]==visit[u])
        return false;
  }
  return true;
}
```

TC → $O(V^2)$

SC → $O(V)$

→ Thread vs Process

→ Why do we need Multi-threading?

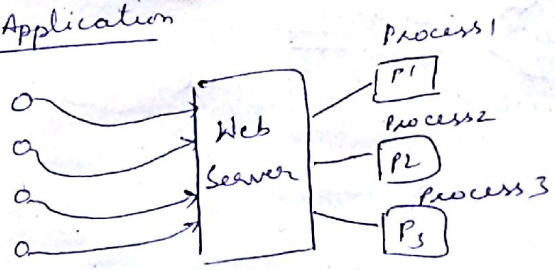
→ Issues with multi-threading.

→ Approaches to solve MT Issues.

Divided & Conquer Application
n/c division on each processor.

Multicore processor.

Web Application



Solution:- Create a light weight component inside a process, which is also called as Thread pool.

(a) Thread creation component → ThreadPool

(b) Thread switching is negligible.

MP

Chrome

Firefox

MT

Strength of MP

(a) Failure of any process doesn't affect other

(b) There will not be any saturation threshold like Multithreading.

Thread have limit:

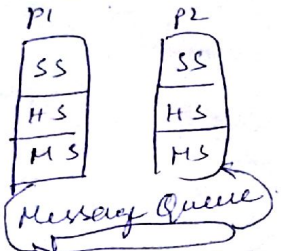
Goal:

(a) Parallelism in an application
(b) Faster response time / Low latency
(c) Improving resource utilization

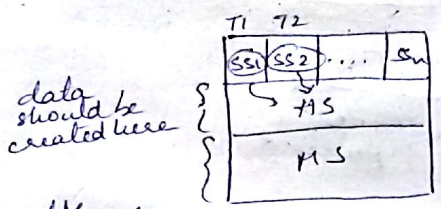
NodeJS — Single Thread Server
       — Asynchronous callback

Thread → Any piece of code that has independent context of execution.

Process Image



Multi-Threaded Process Image



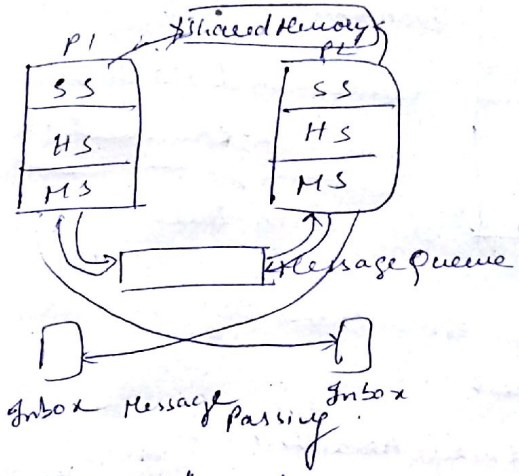

```
class Test extends Thread
{
    public void run()
    {
    }
}
class TestMain
{
  main(String args[])
  {
    Thread t = new Test();
    t.start;
  }
}
```

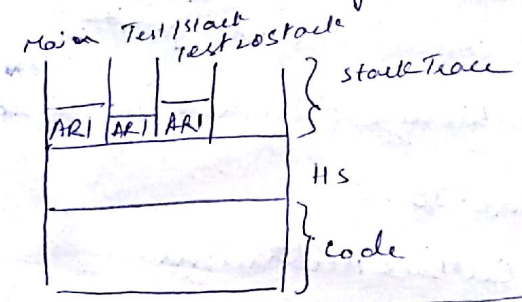
Overriding

→ polymorphism assignment.

→ Dynamic binding.

Inter Process Communication





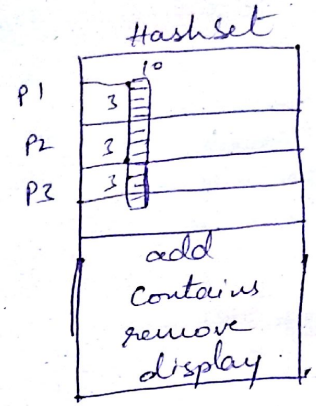
Issues with Multi-Threading

(a) Race condition :- difficult to debug

Thread Safe class

Solution1 Use synchronized method,
Add transactional methods.

Solution2 :- Re-Design class.
Performance ↑ + safety ↑
→ Explicit lock of API.



Concurrent Hash set

java.util.Concurrent.*;

Multi-Threading.

6 Solutions:- Use Immutable objects.

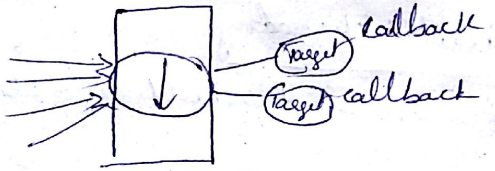
Reads ↑ writes ↓.

Eg:- CopyandWriteArrayList in java.util.concurrent.* package.

Collections.synchronize(Hashset);

Solution 4:- Node.JS

Asynchronous Callback Mechanism.



(2.) Thread Communication Issue

Solutions (a.) ~~Notify~~ wait

(b.) Notify.

(c.) NotifyAll

Solutions2:- Conditional variable.