Unit-3

Pradeep
IT- IIIrd sem (DS)

Linear Search:

A is a linear array with n elements. This algo finds the location (loc) of an item in A or set loc = 0 if search is unsuccessfull.

~~LINEAR (A, N, ITEM, LOC, I)~~

~~1. Set LOC = 0 and I = 0.~~
~~2. Repeat Set A[N] = ITEM.~~
~~3. Set LOC = 0~~
~~4. Repeat step 4 while~~
~~5. Set LOC = LOC + 1.~~
~~6. If LOC = N then~~
~~7. Exit.~~

LINEAR(A, N, ITEM, I, LOC)

1. Set I = 0 and LOC = -1
2. Repeat step 3 & 4 while I < N.
3. If [A[I] == ITEM] then set LOC = I and Exit.
4. I = I + 1
5. If LOC == -1 then write ITEM not found. Else write ITEM found at LOC.
6. Exit.

C-Program for linear search:

```
void main()
{
    int a[100], i, item, n, loc = -1;
    clrscr();
    printf("enter the numbers of elements in array");
    scanf("%d", &n);
    printf("enter the elements of array");
    for(i=0; i<n; i++)
    scanf("%d", &a[i]);
    printf("enter the number to be searched");
    scanf("%d", &item);
    ~~linear(a, n, item);~~
```

linear (cont)

```
for(i=0; i<n; i++)
{ if(a[i]==item)
  { loc=i;
    break;
  }
}
if(loc>=0)
printf("%d is found at %d", item, loc+1);
else
printf("item is not found");
}
```

Linear

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

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

suppose after k compare item will find.

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

Assume $n-k=1$ ⇒ $k = n-1$

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

$T(n) \in O(n)$

complexity:-

Best case = 1 [At first position]

Worst case = n [At last position]

Average case $= 1\cdot\frac{1}{n} + 2\cdot\frac{1}{n} + 3\cdot\frac{1}{n} + 4\cdot\frac{1}{n} + \cdots \frac{n}{n}$

$= [1+2+3+4+\cdots n)\cdot\frac{1}{n}$

$= \frac{n(n+1)}{2}\cdot\frac{1}{n}$

$= \frac{n+1}{2}$

Binary Search :- (A, UB, LB, beg, end, mid, item, LOC,

* Applied for only sorted Array.

① Set beg = L.B. and End = U.B., loc = NULL

and mid = int((beg+End)/2) if a[mid]=item then loc=mid & exit.

② Repeat steps ③ & ④ while beg <= end

~~(3) If item < A[mid] then~~
~~set end = mid - 1.~~
~~else~~
~~beg = mid + 1.~~

(3) Set mid = int ((BEG + END) / 2)

(4) If A[mid] = item then
set loc = mid
Else if ~~A[mid]~~ item > A[mid]
~~else set loc = NULL~~ beg = mid + 1;
else
end = mid - 1;

~~(6) Exit~~

(5) ~~If~~ loc = NULL then not find
Else ITEM find at loc.

(6) Exit

C - Program :-

```
Void main()
{ int a[100], i, n, item, loc, beg, end, mid;
  loc = -1;
  clrscr();
  printf("enter the size of array");
  scanf("%d", &n);
  printf("enter elements of array");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);
printf("enter the item to be search");
scanf("%d", &item);
  beg = 0;
  end = n-1;
~~do~~
~~mid = (beg + end)/2;~~ while(beg <= end)
{ mid = (beg + end)/2;
```

```
if (item == a[mid])
  { loc = mid;
    break;
  }
else if (a[mid] > item)
   end = mid-1;
 else
    beg = mid+1;
}
3.* while(beg<=end && a[mid] != item);
if (loc==-1)
printf("item is not in array");
else
printf("%d is at %d position", item, loc+1);
getch();
}
```

C-coding using Recursion:-

```
void main()
{ int a[100], i, beg, end, n, item, b;
  int binsearch(int a[], int, int, int);
  clrscr();
 printf("How many elements in array");
 scanf("%d", &n);
 printf("enter elements of array");
 for(i=0; i<n; i++)
  scanf("%d", &a[i]);
```

```
printf("Enter item to be search");
scanf("%d",&item);
beg=0;
end=n-1;
b=binsearch(a,beg,end,item);
if(b==-1)
printf("search is not successful");
else
printf("%d is at %d position",item,(b+1));
getch();
}
int binsearch(int a[],int beg,int end,int item)
{ int k,mid;
  mid=(beg+end)/2;
  if(beg<=end)
  { if(a[mid]==item)
      { return(mid);
      }
    else if(item>a[mid])
      beg=mid+1;
      else
      end=mid-1;
    k=binsearch(a,beg,end,item);
    return(k);
}
```

else
return (-1);
}

Complexity of binary search

Maximum no of compare

$f(n) = \lceil \log_2 n \rceil + 1$

$= O(\log_2 n)$

Exp:-

List is 4, 15, 20, 35, 45, 55, 65.

Search 45 in list. $n = 7$.

beg = 0, end = 7-1 = 6.

mid = (beg + end)/2

= (0+6)/2

mid = 3.

item > a[mid]

beg = mid + 1. = 3+1 = 4. , end = 6

mid = (4+6)/2 = 5.

item < a[mid]

end = mid - 1

= 5-1 = 4. beg = 4

mid = (beg + end)/2

= (4+4)/2

= 4

a[mid] == item so location = 4 in C.

Item found At 5th position.

First compare divide the list = $\frac{n}{2}$ element = $\frac{n}{2^1}$

Second ——— = $\frac{n}{4}$ elements = $\frac{n}{2^2}$

Suppose we need k compare to reduced list into list of 1 element.

So $\frac{n}{2^k} = 1$

$2^k = n$

$k = \log_2 n$

At last we need one compare to check whether a item is found or not

So max compare = $\lceil \log_2 n \rceil + 1$

$f(n) = O(\log_2 n)$

Sorting

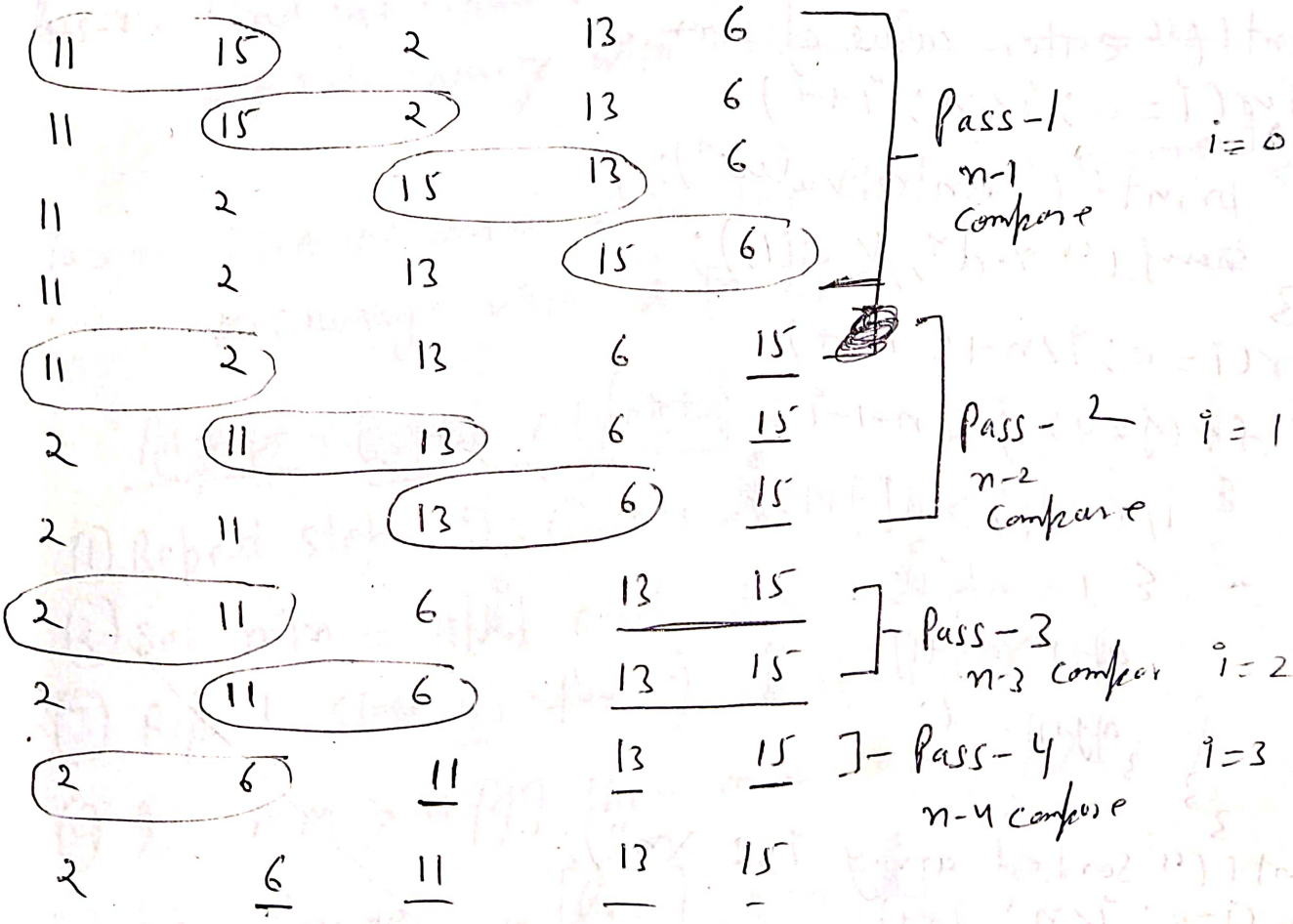
Internal Sorting: If list is stored in primary memory then sorting is called internal sorting.

External Sorting: If list is stored in secondary memory then sorting is called external sorting.

There are many methods of sorting:

(i) By selection
(ii) By insertion
(iii) By exchange
(iv) By merging

Bubble sort:



Total no. Pass = $n-1$, Total compare for each pass = $n-1-i$

Algo:-

(1) Repeat step 2 for I = 0 to I < (n-1).

(2) Repeat step 3 for J = 0 to J < n-1-i

(3) If A[j] > A[j+1] then interchange A[J] & A[J+1] else no change.

(4) ~~Read. Read~~ Exit.

C- coding:-

```
Void main()
{
  int a[50], i, n, j, t;
  clrscr();
 printf("enter size of array");
 scanf("%d", &n);
 ~~printf("enter value of arr~~
  for(i=0; i<n; i++)
  { printf("enter value");
    scanf("%d", &a[i]);
  }
for(i=0; i<n-1; i++)
  { for(j=0; j<n-1-i; j++)
    { if(a[j] > a[j+1])
      { t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
    }
  }
printf("sorted array is: \n");
 for(i=0; i<n; i++)
  printf("%d \n", a[i]);
}
```

Complexity of Bubble Sort:

Pass-1 requires = $n-1$ compare

Pass-2 ——— = $n-2$ ———

:

Pass $(n-1)$ requires = $n-(n-1) = 1$ compare.

So total compare = $1+2+3+\cdots+(n-1)$

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

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

Selection Sort: Procedure:-

Pass-1. Find the smallest in the list & ~~replace~~ interchange 1st element with a[0].

Pass-2. Find the smallest in the list of $n-1$ elements and ~~rep~~ interchange with a[1].

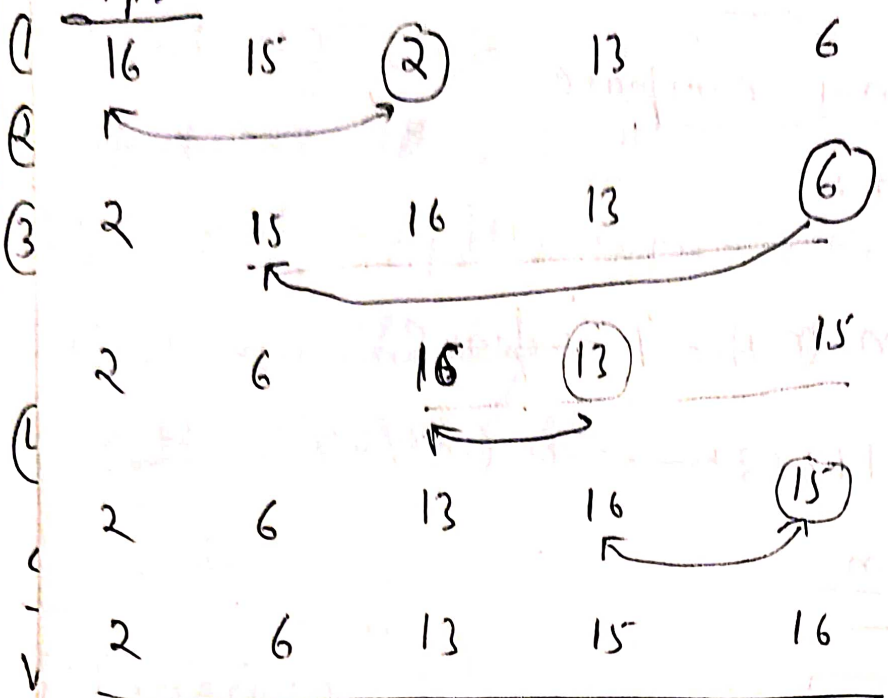
:

Pass $n-1$. Find the smallest in the list of 2 elements & interchange with a[n-2].

Algo:- ~~Algo~~ Algo:-

(1) Repeat steps (2) & (3) (5) for k = 0 to n-2.
(2) set min = a[k] and loc = k.
(3) Repeat step (4) for j = k+1 to n-1.
(4) If min > a[j] then min = a[j] & loc = j.
(5) Interchange a[loc] & a[k] by
temp = a[k], a[k] = a[loc], a[loc] = temp.
(6) Exit

Exp:-



Ans.

C-coding :-

```
for (k=0; k<n-1; k++)
{ loc = k;
  min = a[k];
  for( j= k+1; j<n; j++)
    {
        if (min > a[j])
          { min = a[j];
            loc = j;
          }
    }
  temp = a[k];
  a[k] = a[loc];
  a[loc]= temp;
}
```

Complexity of Selection sort:

Pass-1 needs $n-1$ compares

Pass-2 needs $n-2$ compare

Pass-3 needs $n-3$ compare

⋮

Pass $n-1$ ——— 1 compare

So total compare needed $= (n-1)+(n-2)+\cdots+2+1$

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

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

$$\boxed{f(n) = O(n^2)}$$

Insertion Sort:— Procedure:—

Pass-1 : a[0] by itself is sorted.

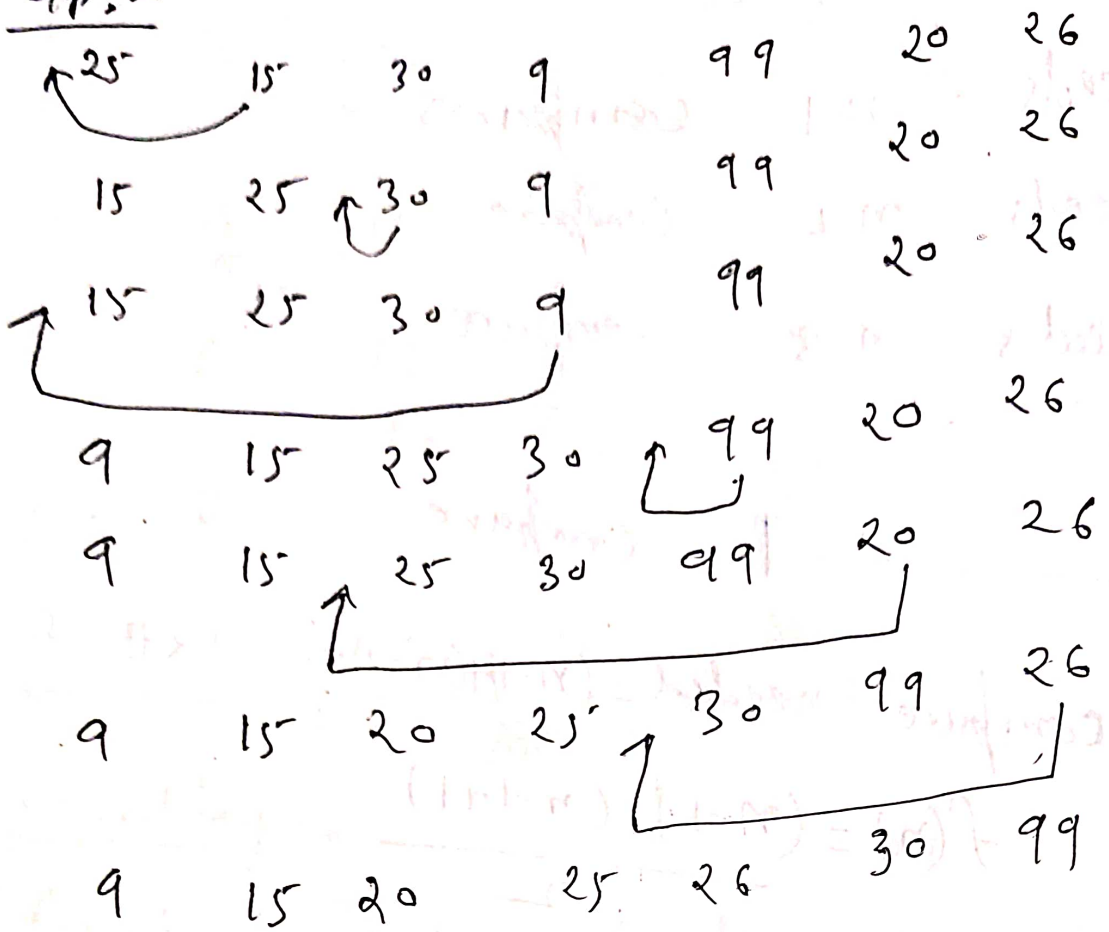
Pass-2 : a[1] is inserted before or after a[0]. now a[0] & a[1] are sorted.

Pass-3 : a[2] is inserted into it's proper place in a[0], a[1] i.e. before a[0], bet^n a[0] and a[1] or after a[1]. Now a[0], a[1], a[2] are sorted.

⋮

Pass n : a[n-1] is inserted into it's proper place in a[0], a[1], — a[n-2]. So now a[0], a[1], --- a[n-1] are sorted.

Exp:-



Algo:-

① set k=1.

② Repeat steps 3, 4 & 6 for k=1 to n-1.

③ set temp = a[k] and j = k-1.

④ Repeat steps ⑤ till (temp < a[j] and j>=0)

⑤ Set a[j+1] = a[j] and j = j-1

⑥ a[j+1] = temp.

⑦ Exit

C-Program:-

```
for(k=1; k<n; k++)
{
  temp = a[k];
  j = k-1;
while(temp < a[j] && j>=0)
{
  a[j+1] = a[j];
  j = j-1;
}
a[j+1] = temp;
}
```

Complexity of Insertion Sort:-

Pass - 1 : For a[0] requires = 0 compare max
Pass - 2 : for a[1] ——— = 1 compare max
Pass - 3 : For a[2] ——— = 2 compare max
Pass - N : For a[N-1] ——— = n-1 compare max.

So total compare required $f(n) = 1+2+3+\cdots+(n-1)$

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

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

Ans

Radix sort:- To sort decimal number, where radix or base is 10 we need 10 pockets. These packets are numbered 0,1,2,3,4,5,6,7,8,9. Numbers are sorted from right digit to left digit i.e. The numbers are sorted first according to unit digit. On the second pass the numbers are sorted according to the tens digit. On the third pass numbers are sorted according to the hundreds digit & so on.

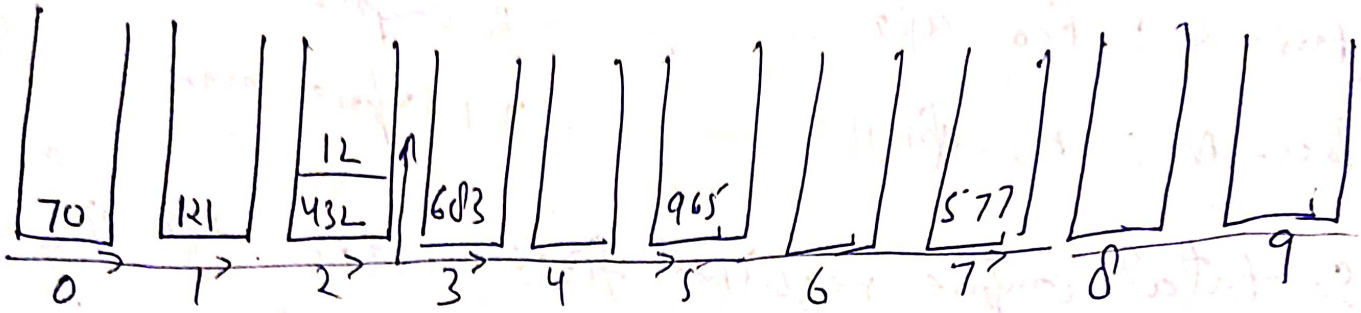
Number of Required Passes = no of digits in largest number.

For sorting names we need 26 pockets.

Exp:- Sort 121, 70, 965, 432, 12, 577, 683

largest no = 965

no of passes required = 3 (no. of digits in largest no.)

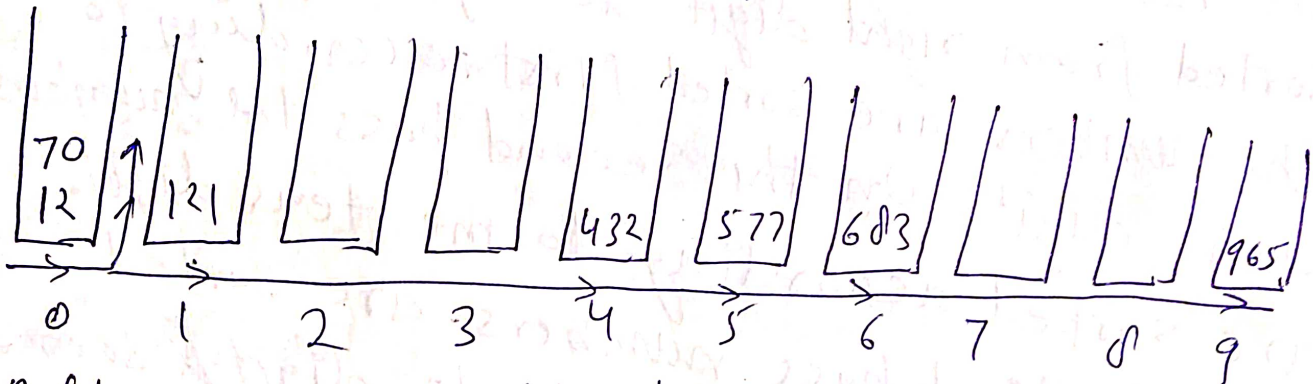


After First pass resulted list is:

70, 121, 432, 12, 683, 965, 577

After second pass resulted list is:

012, 121, 432, 965, 070, 577, 683



After Third or last pass resulted list is:

12, 70, 121, 432, 577, 683, 965.

Ans

C-Prog for Radix sort:-

```
void main()
{ int packet[10][10], buck[10], a[20];
 int i, j, k, l, num, div, large, pass, n;
   div=1;
   num=0;
   printf("enter How many number u want to enter");
   scanf("%d", &n);
   for(i=0; i<n; i++)
   { printf("enter value");
     scanf("%d", &a[i]);
   }
   large = a[0];
  for(i=1; i<n; i++)
   { if(a[i] > large)
      large = a[i];
   }
  while(large>0)
  { num++;
    large = large/10;
  }
for(pass=0; pass<num; pass++)
 { for(k=0; k<10; k++)
    buck[k]=0;
  for(i=0; i<n; i++)
  { l=(a[i]/div)%10;
    packet[l][buck[l]] = a[i];
    buck[l] = buck[l]+1;
  }
```

```
    i=0;
    for(k=0; k<10; k++)
    {for(j=0; j<bucket[k]; j++)
      {a[i] = packet[k][j];
       i=i+1;
      }
    }
    div = div*10;
}
printf("Sorted array is:\n");
for(i=0; i<n; i++)
printf("%d\t", a[i]);
getch();
}
```

Algo:-

① Find the largest element of the array.
② Find the total number of digits in largest number.
③ Repeat steps 4, 5, 6, 7, 8 for pass=1 to n.
④ Repeat step 4.1 for k=0 to 9
  4.1 bucket[k] = 0;
⑤ Repeat step 5.1 for i=0 to n-1.
  5.1 l = (a[i]/div)%10, packet[l] bucket[l] = a[i], and bucket[k] = bucket[l]+1.
⑥ i=0.
⑦ Repeat step 7.1 for k=0 to 9
  7.1 j=0 and Repeat step 7.2 while j < bucket[k].
  7.2 a[i] = packet[k][j], i=i+1, j=j+1.
⑧ div = div*10
⑨ Exit

Complexity:-

$f(n) = n * s * d$

where n = number of element
s = no of digit in largest no. or number of pass
d = base or Radix

Worst case:-

$s = n$ (Assume)

$f(n) = d \times n \times n$

$= d n^2$

$= O(n^2)$

Best case:-

$s = \log_d n$

$f(n) = d \times \log_d n \times n$

$f(n) = O(n \log n)$

Quick Sort:- In Quick sort we devide the original list into two sublists.
We choose the item from list called key from which all the left side elements are smaller and all the right side elements are greater than key element.

Process:-

① Take the first element of list as key.
② Place key at proper place in list. So one element (key) will be it's proper place.
③ Create two sublists left and right side of key.
④ Repeat the same process untill all elements of lists are at proper position in list.

Process for placing key at proper place :-

① Compare the key element one by one from right to left for getting the element which has value less than key.
② Interchange the element with key element.
③ Now the comparison will start from the interchange element position from left to right for getting the element which has higher value than key.
④ Repeat the same process untill key is at proper place.

Eg:-

48, 29, 8, 59, 72, 88, 42, 65, 95, 19, 82, 68

key = 48

compare from right to left & we get 19 which is smaller than 48 we interchange 19, 48.

19, 29, 8, 59, 72, 88, 42, 65, 95, 48, 82, 68

compare key from left to right & we get 59 which is greater than 48 we interchange 48 & 59.

19, 29, 8, 48, 72, 88, 42, 65, 95, 59, 82, 68

compare from right to left & we get 42 which is smaller than 48 we interchange 48, 42.

19, 29, 8, 42, 72, 88, 48, 65, 95, 59, 82, 68

compare from left to right & we get 72 which is greater than 48, interchange 48, 72.

19, 29, 8, 42, 48, 88, 72, 65, 95, 59, 82, 68

Sublist 1 — Apply Quick sort

Sublist 2 — Apply Quick sort

Quick (A, low, up):-

① Set left = low, right = up, key = left

② If low >= up then return else goto 3.

③ Repeat steps ④ & ⑤.

④ (a) Repeat while A[key] ≤ A[right] and key ≠ right
right = right - 1

(b) If key = right then goto ⑥

(c) If A[key] > A[right] then interchange A[key] and A[right] by using
temp = A[key], A[key] = A[right], A[right] = temp.
and set key = right.

⑤ (a) Repeat while A[key] ≥ A[left] and key ≠ left.
left = left + 1.

(b) If key = left then goto ⑥

(c) If A[key] < A[left] then interchange A[key] and A[left] by using
temp = A[left];
A[left] = A[key] and a[key] = temp. and
set key = left.

⑥ call Quick (A, low, key - 1).
call Quick (A, key + 1, up).

⑦ Exit

C-coding for Quick-sort:

```
void main()
{
  int a[20], n, i;
  int Quick(int [], int, int);
  clrscr();
  printf("enter size of array");
  scanf("%d", &n);
  for(i=0; i<n; i++)
  {
    printf("enter value");
    scanf("%d", &a[i]);
  }
  Quick(a, 0, n-1);
  printf("sorted array is:");
  for(i=0; i<n; i++)
    printf("%d", a[i]);
  getch();
}
int Quick(int a[], low, up)
{
  int key, left, right, temp;
  left = low;
  right = up;
  key = left;
  if(low >= up)
    return;
  while(1)
  {
    while(a[key] < a[right] && key != right)
      right = right-1;
    if(key == right)
      break;
```

Worst case complexity:-

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

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

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

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

$\vdots$

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

$= n + (n-1) + (n-2) + \cdots + 1$

$= n \cdot (n+1)/2$

$= O(n^2)$

Same Bubble selection & Insertion sort for Recurrence Relation

```
if ( a[key] > a[right] )
  {
    temp = a[right];
    a[right] = a[key];
    a[key] = temp;
    key = right;
  }
while ( a[key] > a[left] && key != left )
  left = left + 1;
if (key == left)
  break;
if (a[key] < a[left])
  {
    temp = a[key];
    a[key] = a[left];
    a[left] = temp;
    key = left;
  }
}
Quick (a, low, key - 1);
Quick (a, key + 1, up);
}
```

Complexity of Quick sort:-

Worst case complexity:-

If list is already sorted then worst case occurs. Then first element will require n comparisons to recognize that it remains in the first position. Farther more, the first sublist will be empty, but the second sublist will have n-1 elements. Accordingly, the second element will require n-1 comparisons to recognize that it remains in the second position. And so on..

So total compare required

$f(n) = n + n-1 + n-2 + \cdots + 1$

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

$= O(n^2)$

Average case complexity :-

On the average, each reduction step of the algorithm produce two sublist. Accordingly :-

step ① Reduce initial list place 1 element & produce two sublists. So total 1 element is placed. $2^1-1=1$

step ② Reduce two sublists place 2 elements & produce four sublists.

So total $1+2=3$ elements are placed. $2^2-1=3$

Step ③ - Reduce four sublists place 4 elements & produce 8 sublists.

So total $3+4=7$ elements are placed. $2^3-1=7$

suppose we need k-step for placing n elements then -

$$2^k - 1 = n$$

$$2^k = n+1$$

$$\log_2 2^k = \log_2 (n+1)$$

$$k = \log_2 (n+1)$$

For each step maximum compare needed $= n$.

So for k steps total compare needed

$$f(n) = n \times \log_2 (n+1) \qquad \because \log_2(n+1) \simeq \log_2 n$$

$$f(n) = n \log_2 n$$

$$\boxed{f(n) = O(n \log n)}$$

Merge Sort :-

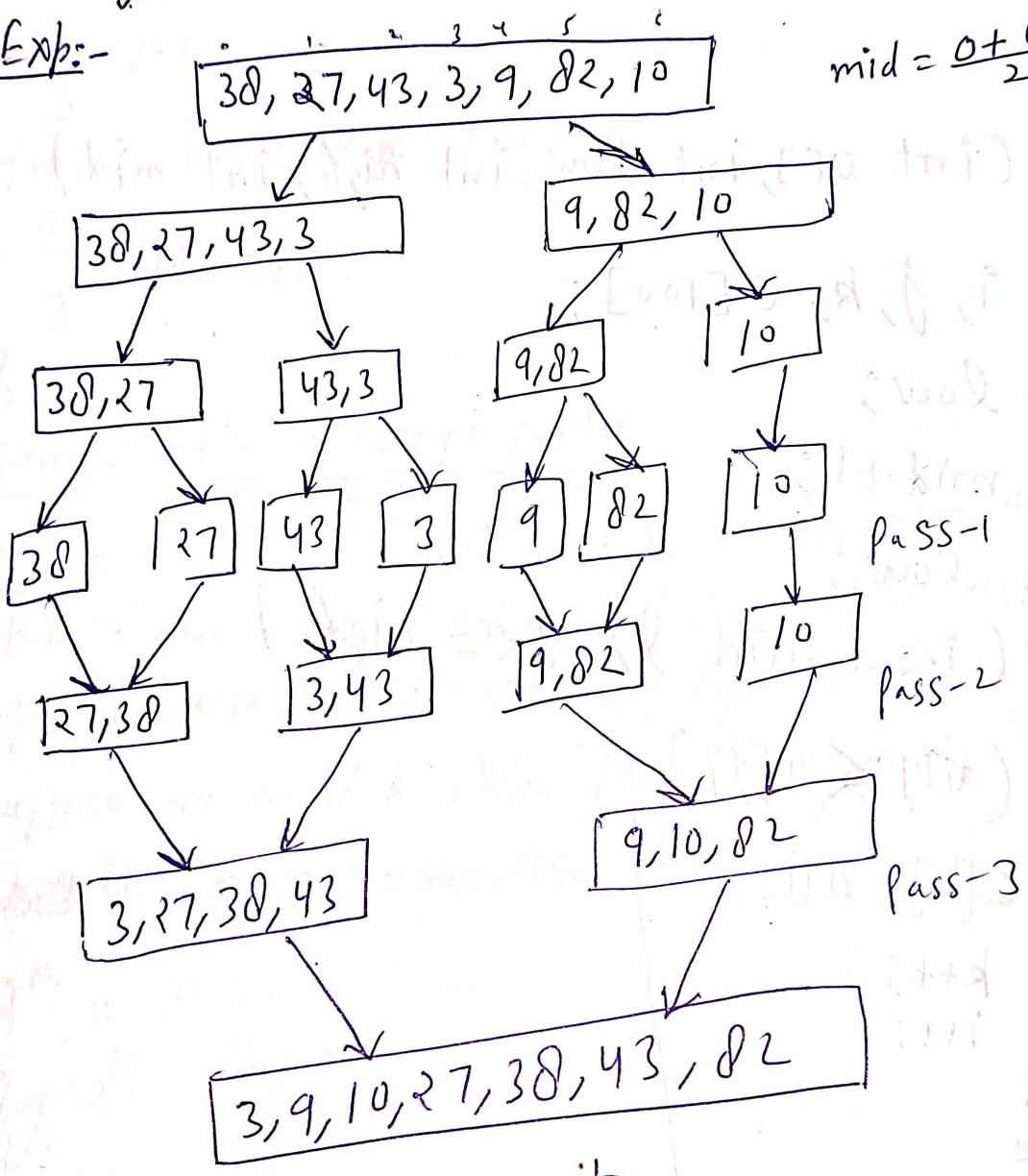
① If the list is of 0 or 1 element then it is already sorted, otherwise.

② Devide the unsorted list into two sublists of about half the size.

③ Sort each sublist recursively.

④ Merge the two sublists back into one sorted list.

Exp:-



Coding for merge sort:

```
mergesort(int a[], int low, int high)
{ int mid;
```

```
if (low < high)
{
  mid = (low+high)/2;
  mergesort (a, low, mid);
  mergesort (a, mid+1, high);
  merge (a, low, high, mid);
}
}

merge (int a[], int low, int high, int mid)
{
  int i, j, k, c[100];
  i = low;
  j = mid+1;
  k = low;
  while (i<= mid && j<= high)
  {
    if (a[i] < a[j])
    {
      c[k] = a[i];
      k++;
      i++;
    }
    else
    {
      c[k] = a[j];
      j++;
      k++;
    }
  }
```

```
while (i<=mid)
{ c[k] = a[i];
  k++;
  i++;
}
while (j<=high)
{ c[k] = a[j];
  k++;
  j++;
}
for (i= low; i<k; i++)
{ a[i] = c[i];
}
}
```

Complexity of Merge Sort:

In merging:-

Step-1:- Produce sorted sublists of Two elements. 2^1

Step-2: Produce sorted sublists of four elements. 2^2

Step-3: Produce sorted sublists of eight elements. 2^3

Suppose we need k steps for ~~placing~~ producing sorted list of n elements. Then

$$2^k = n$$

$$\log_2 2^k = \log_2 n$$

$$\boxed{k = \log_2 n}$$

each step requires maximum compare $= n$

∴ Total compare for k-steps

$$f(n) = nk = n \log n$$

$$f(n) = O(n \log n)$$

Binary search complexity :-

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

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

$= 1 + 1 + T(n/4)$

$= 2 + T(n/4)$

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

$T(n) = 3 + T\left(\frac{n}{2^3}\right)$

suppose r steps list size is 1.

$T(n) = r + T\left(\frac{n}{2^r}\right)$

$\frac{n}{2^r} = 1$

$n = 2^r$

$r = \log n$

$$\boxed{T(n) = \log n + 1}$$

$T(n) = O(\log n)$

Quick sort :

Average case :

Same as merge sort.

Merge sort :

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

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

$= n + 2\left(\frac{n}{2} + 2T(n/4)\right)$

$= n + n + 4T(n/4)$

$= n + n + 4\left(\frac{n}{4} + 2T(n/8)\right)$

$= n + n + n + 2^3 T\left(\frac{n}{8}\right)$

Suppose k steps needed

$= n + n + n + 2^k T\left(\frac{n}{2^k}\right)$

$\frac{n}{2^k} = 1 \rightarrow k = \log_2 n$

$T(n) = kn + 2^k$ $O(n \log n)$

$= kn + n$

$= n(k+1)$

$= n(\log_2 n + 1)$

Unit-3

Hashing: In hashing the record for a key value(k) is directly referred by calculating the address from the key value using hash function.

Suppose there are n elements which are to be stored in hash table of size "M" where $M \geq n$.

An element with key value "k" will be put in slot "j" of hash table if $\boxed{j = h(k)}$

where h(k) is hash function.

Ideally hash function should result in a unique value when applied to any key. But this type of hash function not practically available.

It is quite possible that $h(k_1)$ may be same as $h(k_2)$. This is called collision & k_1 and k_2 are called synonyms.

Different Hash functions:

① **Division method:**

$\boxed{h(k) = k \bmod M}$ key should be distributed uniformly.

So M should be chosen carefully.

M should not be power of 10(10, 100, 1000) because if more odd keys then more odd addresses & less even address [Not - uniform].

M should be non-prime odd integer not divisible by less than 19. or M should be large prime number.

② The mid square method: In this method key is squared and mid portion of squared value is selected.

Exp:- Suppose the q digit address is generated from P digit keys.

P = 4, q = 3.

key = 3271

k^2 = 10699441

or

h(k) = 699 or 994

③ Truncation method:- Here we take only part of the key as address, it can be some rightmost digit or leftmost digit.

Let us take some 8 digit keys.

82394561, 87139465, 83567271, 85943228

Now we can take the number of rightmost digits or leftmost digits based on the size of hash table. Suppose table size is 100 then take the 2 rightmost digits for getting the hash table address.

So address will be 61, 65, 71, 28.

④ Folding method: suppose we have P digit keys. from which q digit address are to be generated. In this method the digits of a key are partitioned into group of q digit from the right. These group are added and the right most q digits of the

sum is selected as the address. In the case of character key we replace the character with their ASCII values.

Exp:- key = 39427829

q = 3.

k = 39 427 829 ← from right.

```
  39
 427
+829
 1295   Right most 3 digit
```

h(k) = 295

(ii) key = DOEACC

D → 68, O → 79, E → 69, A → 65, C → 67, C → 67

```
 687
 969
 656
+767
 3079
```

h(k) = 079

A → 65

a → 97

Hash function for floating point numbers:

1. Take the fractional part of key.
2. Multiply the fractional part with size of hash table.
3. Take the integer part of the multiplication result as a hash address of key.

Exp:- Let us take the keys as floating point number and table size 97.

123.4321, 19.463, 2.0298

0.4321 × 97 = 41.9137
0.463 × 97 = 44.911
0.0298 × 97 = 2.8906

h (123.431) = 41
h (19.463) = 44
h (2.0298) = 2

collision
Conflict Resolution techniques:

There are two method for conflict resolution:

(i) Open addressing
(ii) Chaining

In open addressing an empty position is found out within the hash table itself and the new key is inserted in that empty position.

In chaining method the synonyms are placed in linked list. Nodes of linked list may be allocated from some space outside the hash table.

Open Addressing:

Linear & Quadratic probing: In this method if

linear probing

new hash address is already occupied by another element, then hashed index are sequentially examined to locate the first empty position. The concerned element is stored in this index. In this hash table is considered as circular.

In linear probing

$$j = (j+i) \% M \quad \text{where } i = 1, 2, 3, \ldots\ldots$$

The main disadvantage of the linear probing technique is clustering problem. when half of the table is full then it is difficult to find empty position in hash table in case of collision. searching will also become slow because it will go for linear searching.

It will search $j, j+1, j+2, j+3, \ldots\ldots$

Quadratic probing: In Quadratic probing it search location $j = (j + i^2) \% M$, where $i = 1, 2, 3, \ldots$

So it will search the locations $j, j+1, j+4, j+9, \ldots$

So it will decrease the problem of clustering but this technique can not search all the locations where M is size of hashtable.

$h(k) = j \% M$.

Let us take some elements & table size 11.

29, 18, 43, 10, 36, 25, 46

Linear Probing

0	10
1	
2	46
3	36
4	25
5	
6	
7	29
8	18
9	
10	43

$H(29) = 29 \% 11 = 7$
$H(18) = 18 \% 11 = 7$ collision
$H(18) = (18+1) \% 11 = 8$
$H(43) = 43 \% 11 = 10$
$H(10) = 10 \% 11 = 10$ collision
$H(10) = (10+1) \% 11 = 0$
$H(36) = 36 \% 11 = 3$
$H(25) = 25 \% 11 = 3$ collision
$H(25) = (25+1) \% 11 = 4$
$H(46) = 46 \% 11 = 2$

Quadratic Probing

29, 18, 43, 10, 46, 54

0	10
1	
2	46
3	54
4	
5	
6	
7	29
8	18
9	
10	43

$H(29) = 29 \% 11 = 7$
$H(18) = 18 \% 11 = 7$ collision
$H(18) = (18+1) \% 11 = 8$
$H(43) = (43 \% 11) = 10$
$H(10) = (10 \% 11) = 10$ collision
$H(10) = (10+1) \% 11 = 0$
$H(46) = 46 \% 11 = 2$
$H(54) = 54 \% 11 = 10$ collision
$H(54) = (54+1) \% 11 = 0$ collision
$H(54) = (54+4) \% 11 = 3$

② Double Hashing:- This method required two hash functions. The function $h_1(k)$ used as a primary function. If address generated by $h_1(k)$ is already occupied by a key then second hash function $h_2(k)$ is used to compute the increment to be added to the address obtained by $h_1(k)$.

$h_1(k) = h$, $h_2(k) = h'$ $\therefore$ $\boxed{h(k) = (h + h') \% m}$

Now we will search the location $h, h+h', h+2h', h+3h', \ldots \% m$

*:- Second hash function should not give zero value.

Eg:- h = key %13

h' = 11 - (key % 11)

Next prob will be $(h+h') \% 13$ where 13 is size of table.

Exp:- 8, 55, 48, 68

0	
1	
2	
3	55
4	
5	
6	
7	
8	8
9	48
10	
11	
12	68

$h(8) = (8 \% 13) = 8$

$h(55) = (55 \% 13) = 3$

$h(48) = (48 \% 13) = 9$

$h(68) = (68 \% 13) = 3$ collision

$h(68) = (68 \% 13 + (11 - 68 \% 11)) \% 13$

$= (3 + 9) \% 13$

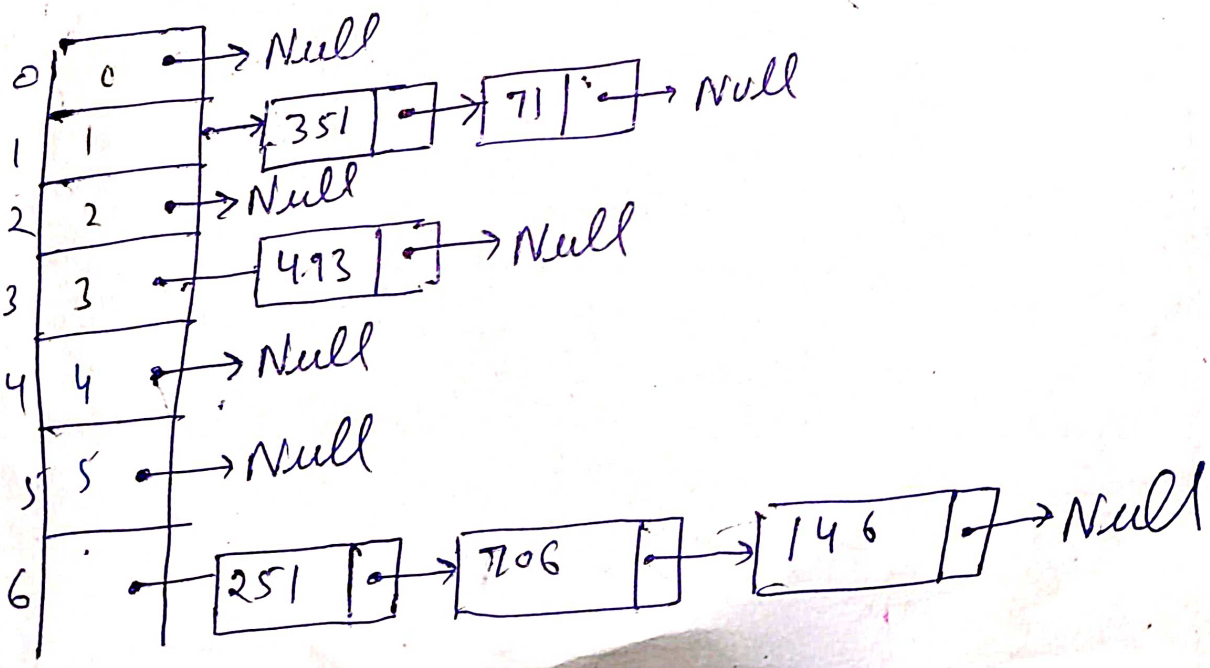
$= 12$

separate chaining :- This method maintains the chain of elements which have same hash address. We can take the hash table as an array of pointers. ~~Size of hash table can be number of records.~~ Here each pointer will point to one linked list and the elements which have same hash address will be maintained in the linked list. Nodes are stored outside the space of hash table so this is called separate chaining. elements will be represent by

```
struct node
{
    int key;
    struct node *next;
};
```

Exp:- Let 351, 493, 251, 71, 706, 146

$h(k) = k \bmod 7$, key 351, 493, 251, 71, 706, 146
h(k) 1, 3, 6, 1, 6, 6,



The main disadvantage is wastage of memory space. In case where records are very small or contains only key then link part for every element in linked list & pointer space in hash table will be wastage of memory space.