Unit-4

Sorting : Sorting Techniques- Sorting by Insertion: Straight Insertion sort- List insertion sort- Binary insertion sort- Sorting by selection: Straight selection sort- Heap Sort- Sorting by Exchange- Bubble Sort- Shell Sort-Quick Sort-External Sorts: Merging Order Files-Merging Unorder Files- Sorting Process.

Sorting in general refers to various methods of arranging or ordering things based on criterias (numerical, chronological, alphabetical, hierarchical etc.). There are many approaches to sorting data and each has its own merits and demerits.

Bubble Sort:

Bubble Sort is probably one of the oldest, easiest, straight-forward, inefficient sorting algorithms. Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists. Bubble sort is not a stable sort which means that if two same elements are there in the list, they may not get their same order with respect to each other.

Bubble Sort Algorithm:

```
Step 1: Repeat Steps 2 and 3 for i=1 to 10
Step 2: Set j=1
Step 3: Repeat while j<=n
         (A) if a[i] < a[j]
              Then interchange a[i] and a[j]
              [End of if]
         (B) Set j = j+1
        [End of Inner Loop]
    [End of Step 1 Outer Loop]
Step 4: Exit
```

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int a[100],n,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
bubblesort(a,n);
printf("\n values after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
return 0;
}

bubblesort(int a[],int n)
{
int i,j,temp;
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc bs.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out
enter size
4
enter values
40
20
30
10
values before sorting

40      20      30      10

 values after sorting
```

10
20
30
40

Time Complexity:

Worst Case Performance	$O(N^2)$
Best Case Performance	$O(N^2)$
Average Case Performance	$O(N^2)$

Selection Sort:

The algorithm divides the input list into two parts: the sub-list of items already sorted, which is built up from left to right at the front (left) of the list, and the sub-list of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sub-list is empty and the unsorted sub-list is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sub-list, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sub-list boundaries one element to the right.

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n^2) time complexity, making it inefficient on large lists.

Selection Sort Algorithm and Pseudo Code:

```
/* a[0] to a[n-1] is the array to sort */
int i,j;
int iMin;
 /* advance the position through the entire array */
/*   (could do j < n-1 because single element is also min element) */
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */
     /* assume the min is the first element */
    iMin = j;
    /* test against elements after j to find the smallest */
    for ( i = j+1; i < n; i++) {
        /* if this element is less, then it is the new minimum */
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;





           }
       }

       /* iMin is the index of the minimum element. Swap it with the current
   position */
       if ( iMin != j ) {
           swap(a[j], a[iMin]);
       }
   }
```

PROGRAM:

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
int a[100],n,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
selectionsort(a,n);
printf("\n values after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}

return 0;
}

selectionsort(int a[],int n)
{
int i,j,temp,min;
for(i=0;i<n;i++)
{
min=i;
for(j=i+1;j<n;j++)
{
if(a[min]>a[j])
{
min=j;
}
}
temp=a[i];
a[i]=a[min];
a[min]=temp;
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-Veriton-Series:~$ gedit ss.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ gcc ss.c
gpcetadmin@gpcetadmin-Veriton-Series:~$ ./a.out
enter size
4
enter values
40
20
30
10
values before sorting

40      20      30      10
 values after sorting
10
20
30
40
```

Time Complexity:

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes n – 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n – 1 elements and so on, for $(n-1)+(n-2)+\ldots+2+1 = n(n-1)/2 \in O(n^2)$ comparisons. Each of these scans requires one swap for n – 1 elements (the final element is already in place).

Worst Case Performance	**$O(N^2)$**
Best Case Performance	**$O(N^2)$**
Average Case Performance	**$O(N^2)$**

Insertion Sort:

An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted). Insertion sort is an example of an **incremental** algorithm. It builds the sorted sequence one number at a time. This is a suitable sorting technique in playing card games. Insertion sort provides several advantages:

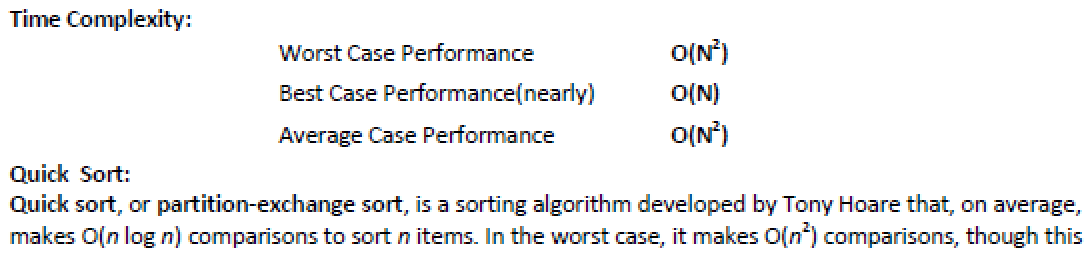
- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is $O(n + d)$, where d is the number of inversions
- More efficient in practice than most other simple quadratic (i.e., $O(n^2)$) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is $O(n)$
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

PROGRAM:

```
#include<stdio.h>
int main()
{
int a[100],n,i;
printf("enter n\n");
scanf("%d",&n);
printf("enter values \n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
is(a,n);
printf("after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
return 0;
}


is(int a[],int n)
{
int i,j,index;
for(i=1;i<n;i++)
{
index=a[i];
j=i;
while((j>0) && a[j-1]>index)
{
a[j]=a[j-1];
j=j-1;
}
a[j]=index;
}
}
```

INPUT/OUTPUT:

```
gpcet@gpcet-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$ gcc ins.c
gpcet@gpcet-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$ ./a.out
enter n
5
enter values
50
30
20
10
40
before sorting
50      30      20      10      40
after sorting
10      20      30      40      50
```

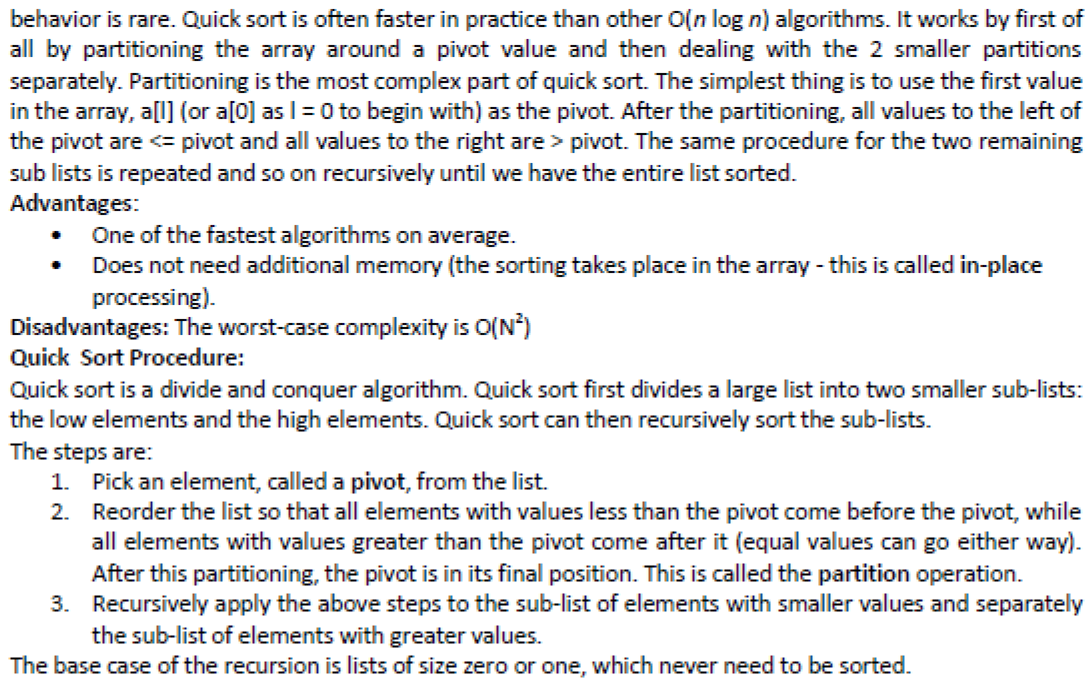


Time Complexity:

Worst Case Performance	$O(N^2)$
Best Case Performance(nearly)	$O(N)$
Average Case Performance	$O(N^2)$

Quick Sort:

Quick sort, or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare that, on average, makes O(*n* log *n*) comparisons to sort *n* items. In the worst case, it makes $O(n^2)$ comparisons, though this



behavior is rare. Quick sort is often faster in practice than other O(*n* log *n*) algorithms. It works by first of all by partitioning the array around a pivot value and then dealing with the 2 smaller partitions separately. Partitioning is the most complex part of quick sort. The simplest thing is to use the first value in the array, a[l] (or a[0] as l = 0 to begin with) as the pivot. After the partitioning, all values to the left of the pivot are <= pivot and all values to the right are > pivot. The same procedure for the two remaining sub lists is repeated and so on recursively until we have the entire list sorted.

Advantages:

- One of the fastest algorithms on average.
- Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing).

Disadvantages: The worst-case complexity is $O(N^2)$

Quick Sort Procedure:

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a **pivot**, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion is lists of size zero or one, which never need to be sorted.

PROGRAM: (Recursive)

```
#include<stdio.h>
int main()
```

```
{
int a[50],n,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}

printf("values before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
qs(a,n);
printf("values after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
return 0;
}



qs(int a[],int n)
{
sort(a,0,n-1);
}

sort(int a[],int left,int right)
{
int pivot,l,r;
pivot=a[left];
l=left;
r=right;
while(left<right)
{
while((a[right]>=pivot) && (left<right))
right--;
if(left!=right)
{
a[left]=a[right];
left++;
}
while((a[left]<=pivot) && (left<right))
left++;
if(left!=right)
{
a[right]=a[left];
right--;
}
}
a[left]=pivot;
pivot=left;
left=l;
right=r;
if(left<pivot)
sort(a,left,pivot-1);
if(right>pivot)
sort(a,pivot+1,right);
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gcc qsr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out

enter size
5
enter values
30
50
20
40
10
values before sorting
30      50      20      40      10
values after sorting
10      20      30      40      50
```

PROGRAM:(Non-Recursive)

```
#include <stdio.h>
// A utility function to swap two elements
void swap ( int* a, int* b )
{
        int t = *a;
        *a = *b;
        *b = t;
}


/* This function is same in both iterative and recursive*/
int partition (int arr[], int l, int h)
{
        int x = arr[h];
        int i = (l - 1);
        int j;

        for (j = l; j <= h- 1; j++)
        {
                if (arr[j] <= x)
                {
                        i++;
                        swap (&arr[i], &arr[j]);
                }
        }

        swap (&arr[i + 1], &arr[h]);

return (i + 1);
}


/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */

void quickSortIterative (int arr[], int l, int h)
{
// Create an auxiliary stack

int stack[ h - l + 1 ];



        // initialize top of stack

int top = -1;

// push initial values of l and h to stack

stack[ ++top ] = l;

stack[ ++top ] = h;
```

```
        // Keep popping from stack while is not empty

        while ( top >= 0 )
        {
                // Pop h and l
                h = stack[ top-- ];
        l = stack[ top-- ];


                // Set pivot element at its correct position in sorted array
                int p = partition( arr, l, h );

                // If there are elements on left side of pivot, then push left
                // side to stack
                if ( p-1 > l )
                {
                        stack[ ++top ] = l;
                stack[ ++top ] = p - 1;
                }
                // If there are elements on right side of pivot, then push right
        // side to stack
                if ( p+1 < h )
        {
                stack[ ++top ] = p + 1;
                     stack[ ++top ] = h;
                }
        }
}


void printArr( int arr[], int n )
{
        int i;
        for ( i = 0; i < n; ++i )
                printf( "%d ", arr[i] );



int main()
{
        int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};
        int n = sizeof( arr ) / sizeof( *arr );
        quickSortIterative( arr, 0, n - 1 );
        printArr( arr, n );
        return 0;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$ gcc
nrqs.c
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$
./a.out
1 2 2 3 3 3 4 5
```

PROGRAM :(Randomized Element as pivot)

```
#include<stdlib.h>
#include<stdio.h>
#include <time.h>


int partition(int *a,int p,int r)
{
srand(time(0));
int pivot = rand()%(r-p);
pivot += p;
swap(&a[r],&a[pivot]);
pivot = r;
int temp = a[r];
int i = p-1;
int j;
for (j = p; j < r ;j++)
 {
if ( a[j] <  temp )
 {
i++;
swap (&a[i],&a[j]);
}
}
i++;
swap (&a[r],&a[i]);
return i;
}



swap(int *r,int *pivot)
{
```

```
int temp;
temp=*r;
*r=*pivot;
*pivot=temp;
}




void quicksort(int *a,int p,int r)
{
if (p < r)
{
int q = partition(a,p,r);
quicksort(a,p,q);
quicksort(a,q+1,r);
}
}



int main() {
// your code goes here

int i;
int a[] ={2,5,6,2,1,9,8};
quicksort (a,0,6);
for (i=0;i<7;i++)
printf("%d\t",a[i]);
return 0;
}
```

INPUT/OUTPUT:

```
  gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$
gedit sariyu.c
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$ gcc
sariyu.c
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$
./a.out

1	2	2	5	6	8	9
```

Merge Sort:

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The
two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

Merge Sort Procedure:

This is a divide and conquer algorithm.
This works as follows :

1. Divide the input which we have to sort into two parts in the middle. Call it the left part and right part.
Example: Say the input is -10 32 45 -78 91 1 0 -16 then the left part will be -10 32 45 -78 and the right part will be 91 1 0 6.
2. Sort each of them separately. Note that here sort does not mean to sort it using some other method. We use the same function recursively.
3. Then merge the two sorted parts.

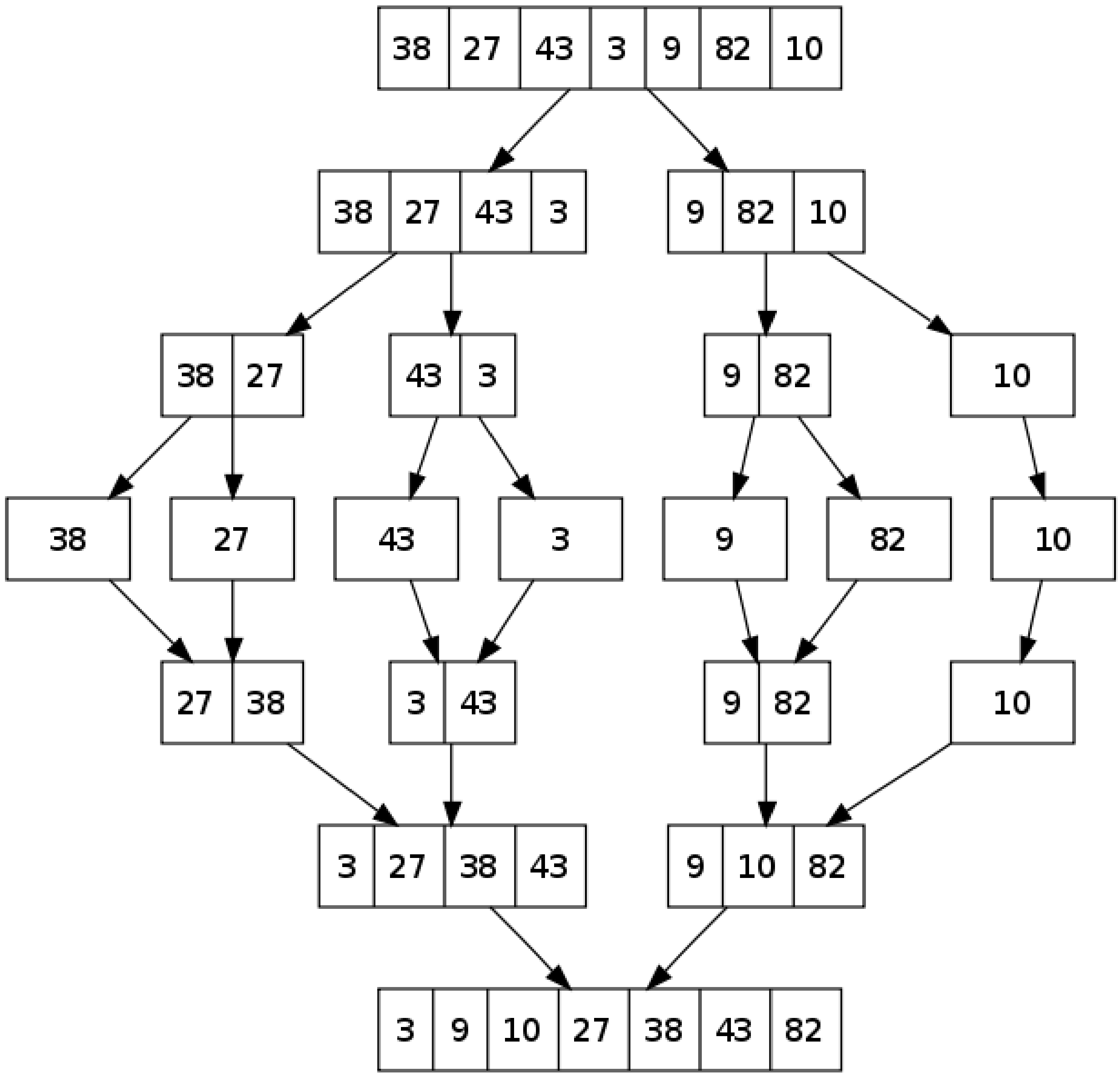
Input the total number of elements that are there in an array (number_of_elements). Input the array (array[number_of_elements]). Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in the range [left,right] i.e. from index left to index right inclusive. Merge() function merges the two sorted parts. Sorted parts will be from [left, mid] and [mid+1, right]. After merging output the sorted array.

MergeSort() function:

It takes the array, left-most and right-most index of the array to be sorted as arguments. Middle index (mid) of the array is calculated as (left + right)/2. Check if (left<right) cause we have to sort only when left<right because when left=right it is anyhow sorted. Sort the left part by calling MergeSort() function again over the left part MergeSort(array,left,mid) and the right part by recursive call of MergeSort function as MergeSort(array,mid + 1, right). Lastly merge the two arrays using the Merge function.

Merge() function:

It takes the array, left-most , middle and right-most index of the array to be merged as arguments.
Finally copy back the sorted array to the original array.



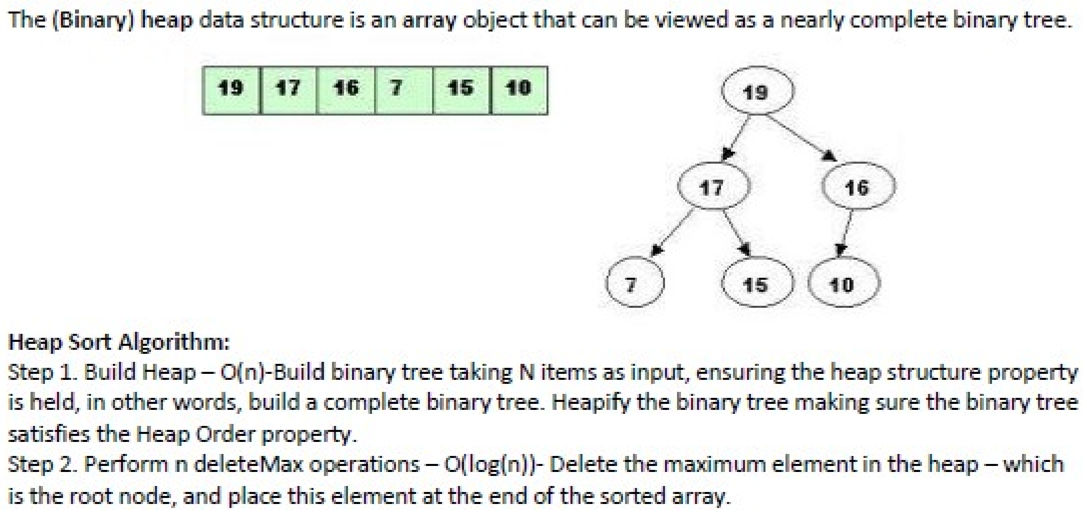
```
main()
{
int a[100],n,i;
printf("enter n\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
mergesort(a,0,n-1);
printf("values after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
}

int 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,mid,high);
}
}

int merge(int a[ ],int l,int m,int h)
{
int i,j,k,b[50];
i=l;
j=m+1;
k=l;
while(i<=m && j<=h)
{
if(a[i]<=a[j])
b[k++]=a[i++];
else
b[k++]=a[j++];
}
while(i<=m)
b[k++]=a[i++];
while(j<=h)
b[k++]=a[j++];
for(k=l;k<=h;k++)
{
a[k]=b[k];
}
return 0;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$ gcc
mer.c
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$
./a.out
enter n
3
enter values
30
0
-1
values before sorting
30        0         -1
          values after sorting
-1
0
30
```

Heap Sort:

The heap sort algorithm can be divided into two parts. In the first step, a heap is built out of the data. In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one. Heap sort can be performed in place. The array can be split into two parts, the sorted array and the heap.


The (Binary) heap data structure is an array object that can be viewed as a nearly complete binary tree.

Heap Sort Algorithm:

Step 1. Build Heap – O(n)-Build binary tree taking N items as input, ensuring the heap structure property is held, in other words, build a complete binary tree. Heapify the binary tree making sure the binary tree satisfies the Heap Order property.

Step 2. Perform n deleteMax operations – O(log(n))- Delete the maximum element in the heap – which is the root node, and place this element at the end of the sorted array.

```
#include<stdio.h>
 int p(int);
 int left(int);
 int right(int);
void heapify(int[],int,int);
 void buildheap(int[],int);
 void heapsort(int[],int);
 void main()
{
int x[20],n,i;
printf("enter the no. of elements to b sorted");
 scanf("%d",&n);
printf("enter the elements ");
for(i=0;i<n;i++)
scanf("%d",&x[i]);
printf("values before sorting\n");
for(i=0;i<n;i++)
printf("%d\t ",x[i]);
heapsort(x,n);
 printf("sorted array is\n");
 for(i=0;i<n;i++)
printf("%d\t ",x[i]);
 }



int p(int i)
{
return i/2;
}



int left(int i)
{
 return 2*i+1;
 }



 int right(int i)

 {
 return 2*i+2;
 }



void heapify(int a[],int i,int n)
 {
 int l,r,large,t;
 l=left(i);
 r=right(i);
 if((l<=n-1)&&(a[l]>a[i]))
large=l;
 else large=i;
 if((r<=n-1)&&(a[r]>a[large]))
 large=r;
if(large!=i)
 {
t=a[i];
 a[i]=a[large];
 a[large]=t;
 heapify(a,large,n);
 }
 }


void buildheap(int a[],int n)
{
 int i;
 for(i=(n-1)/2;i>=0;i--)
 heapify(a,i,n);
 }


void heapsort(int a[],int n)
{
 int i,m,t;
 buildheap(a,n);
 m=n ;
for(i=n-1;i>=1;i--)
{
 t=a[0];
 a[0]=a[i];

 a[i]=t;
 m=m-1;
 heapify(a,0,m);
 }
 }
```

INPUT/OUPUT:

```
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$ gcc
hs.c
gpcetadmin@gpcetadmin-HCL-Infosystems-Limited-BIOS-For-GA-G31M-S2L:~$
./a.out
```

```
enter the no. of elements to b sorted5
enter the elements 50
30
20
10
40
values before sorting
50       30      20      10      40
 sorted array is
10          20         30         40         50
```

Time Complexity:

Worst Case Performance	**O(N $\log_2$ N)**
Best Case Performance(nearly)	**O(N $\log_2$ N)**
Average Case Performance	**O(N $\log_2$ N)**

Radix/Bucket Sort:

Bucket sort, or **bin sort**, is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. It is a distribution sort, and is a cousin of radix sort in the most to least significant digit flavor.

Bucket sort works as follows:

1. Set up an array of initially empty "buckets".
2. **Scatter**: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. **Gather**: Visit the buckets in order and put all elements back into the original array.

Radix/Bucket Procedure:

1. Reading the list of elements. Calling the Radix sort function.
2. Checking the biggest number (big) in the list and number of digits in the biggest number (nd).
3. Inserting the numbers in to the buckets based on the one's digits and collecting the numbers

   and again inserting in to buckets based on the ten's digits and soon...
4. Inserting and collecting is continued 'nd' times. The elements get sorted.
5. Displaying the elements in the list after sorting.

Source code:

```
#include<conio.h>
#include<stdio.h>
void radix_sort(int a[20],int n)
{
        int bucket[10][10],ctr[10]={0};
        int big,nd,i,j,k,l,div=1,b_no;
        big=a[0];
        i=1;
        while(i<n)
        {
                if(a[i]>big)
                        big=a[i];
                        i++;
        }                   /*checking the biggest number*/
        nd=0;
        while(big>0)
        {
```

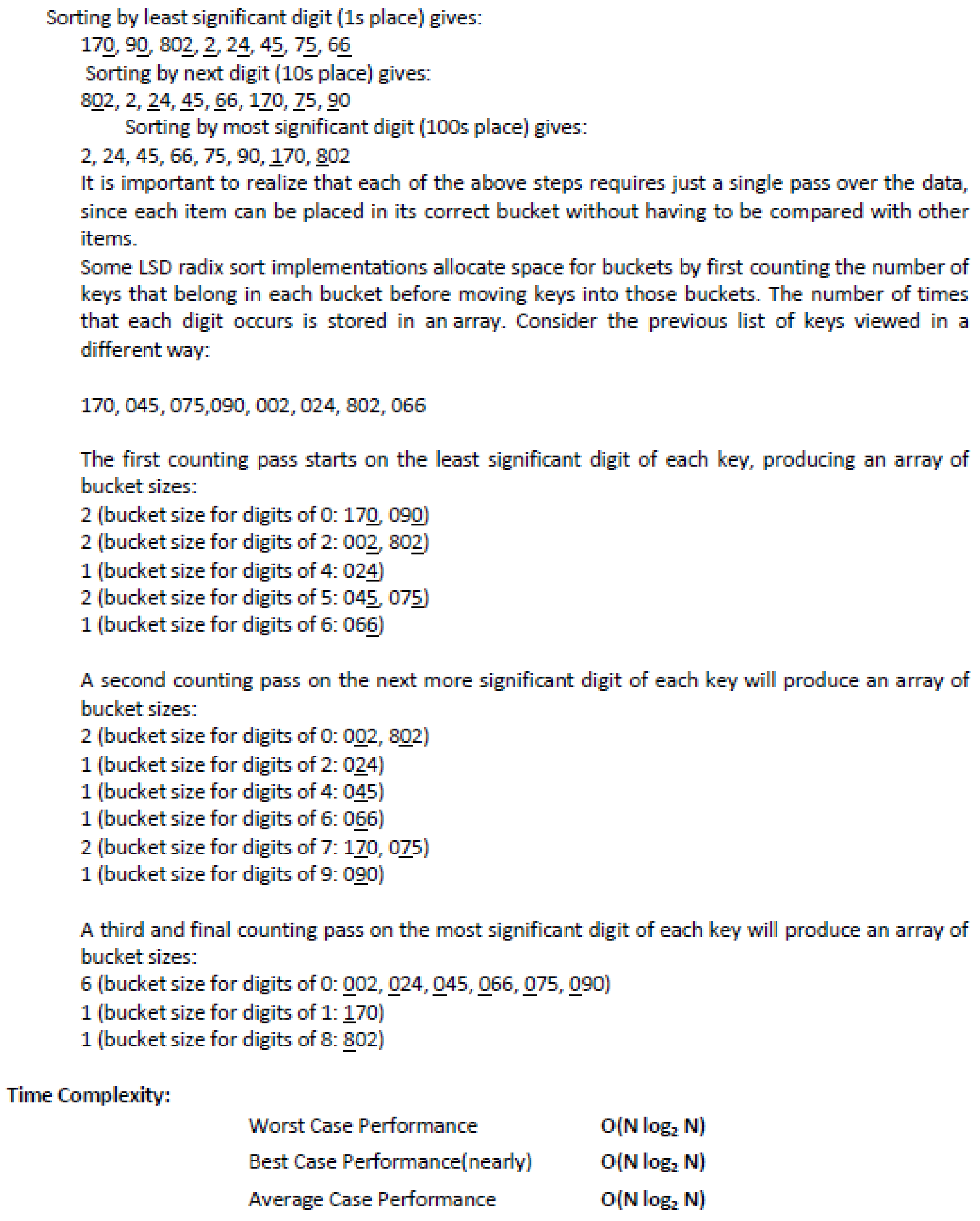
```
nd=nd+1;
big=big/10;
} /*checking number of digits in the biggest number*/
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
b_no=((a[j]/div)%10);
bucket[b_no][ctr[b_no]]=a[j];
ctr[b_no]=ctr[b_no]+1;
} /*Inserting the numbers in to the buckets*/
div=div*10;
j=0;
for(k=0;k<10;k++)
{
for(l=0;l<ctr[k];l++)
{
a[j]=bucket[k][l];
j++;
}
ctr[k]=0;
} /*collecting the elements from the buckets*/
}
}
void main()
{
int i,n,a[20];
printf("\nEnter n:");
scanf("%d",&n);
printf("\nEnter the elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\nElements before sorting:\n");
for(i=0;i<n;i++)
printf("%5d",a[i]);
radix_sort(a,n);
printf("\nElements after sorting:\n");
for(i=0;i<n;i++)
printf("%5d",a[i]);
getch();
}
```

Output:

```
 Enter n:5
Enter the elements:013 323 526 314
33
Elements before sorting:
13 323 526 314 33
Elements after sorting:
13 33 314 323 526
```

Step-by-step example:

Original, unsorted list:
170, 45, 75, 90, 802, 24, 2, 66



Sorting by least significant digit (1s place) gives:
170, 90, 802, 2, 24, 45, 75, 66
Sorting by next digit (10s place) gives:
802, 2, 24, 45, 66, 170, 75, 90
Sorting by most significant digit (100s place) gives:
2, 24, 45, 66, 75, 90, 170, 802
It is important to realize that each of the above steps requires just a single pass over the data, since each item can be placed in its correct bucket without having to be compared with other items.
Some LSD radix sort implementations allocate space for buckets by first counting the number of keys that belong in each bucket before moving keys into those buckets. The number of times that each digit occurs is stored in an array. Consider the previous list of keys viewed in a different way:

170, 045, 075,090, 002, 024, 802, 066

The first counting pass starts on the least significant digit of each key, producing an array of bucket sizes:
2 (bucket size for digits of 0: 170, 090)
2 (bucket size for digits of 2: 002, 802)
1 (bucket size for digits of 4: 024)
2 (bucket size for digits of 5: 045, 075)
1 (bucket size for digits of 6: 066)

A second counting pass on the next more significant digit of each key will produce an array of bucket sizes:
2 (bucket size for digits of 0: 002, 802)
1 (bucket size for digits of 2: 024)
1 (bucket size for digits of 4: 045)
1 (bucket size for digits of 6: 066)
2 (bucket size for digits of 7: 170, 075)
1 (bucket size for digits of 9: 090)

A third and final counting pass on the most significant digit of each key will produce an array of bucket sizes:
6 (bucket size for digits of 0: 002, 024, 045, 066, 075, 090)
1 (bucket size for digits of 1: 170)
1 (bucket size for digits of 8: 802)

Time Complexity:

Worst Case Performance	**$O(N \log_2 N)$**
Best Case Performance(nearly)	**$O(N \log_2 N)$**
Average Case Performance	**$O(N \log_2 N)$**

Shellsort

- Improves on insertion sort.

 Starts by comparing elements far apart, then elements less far apart, and finally comparing adjacent elements (effectively an insertion sort). By this stage the elements are sufficiently sorted that the running time of the final stage is much closer to O(N) than $O(N_2)$.

Shellsort, is also known as the **diminishing increment sort**,
Shellsort is one of the oldest sorting algorithms, named after its inventor Donald. L. Shell (1959). It is fast, easy to understand and easy to implement. However, its complexity analysis is sophisticated. The idea of Shellsort is the following:

a) arrange the data sequence in a two-dimensional array
b) sort the columns of the array

The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column.
In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted. However, the number of sorting operations necessary in each step is limited, due to the presortedness of the sequence obtained in the preceding steps.

Example: Let `3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2` be the data sequence to be sorted. First, it is arranged in an array with 7 columns (left), then the columns are sorted (right):

```
3 7 9 0 5 1 6                        3 3 2 0 5 1 5
8 4 2 0 6 1 5                        7 4 4 0 6 1 6
7 3 4 9 8 2                          8 7 9 9 8 2
```

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is also still there. In the next step, the sequence is arranged in 3 columns, which are again sorted:

```
3 3 2                  0 0 1
0 5 1                  1 2 2
5 7 4                  3 3 4
4 0 6                  4 5 6
1 6 8                  5 6 8
7 9 9                  7 7 9
8 2                    8 9
```

Now the sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 6, an 8 and a 9 that have to be moved a little bit to their correct positions

Implementation Actually, the data sequence is not arranged in a two-dimensional array, but held in a one-dimensional array that is indexed appropriately.
The algorithm uses an increment sequence to determine how far apart elements to be sorted are:
$h_1, h_2, ..., h_t$ with $h_1 = 1$

At first elements at distance h_t are sorted, then elements at distance h_{t-1}, etc, until finally the array is sorted using insertion sort (distance $h_1 = 1$).

An array is said to be h_k-sorted if all elements spaced a distance h_k apart are sorted relative to each other.
Shellsort only works because an array that is h_k-sorted remains h_k-sorted when h_{k-1}-sorted. This means that subsequent sorts with a smaller increment do not undo the work done by previous phases

The **correctness of the algorithm** follows from the fact that in the last step (with *h* = 1) an ordinary Insertion Sort is performed on the whole array. But since data are presorted by the preceding steps (*h* = 3, 7, 21, ...) only few Insertion Sort steps are required.

```
int j, p, gap;
comparable tmp;
for (gap = N/2; gap > 0; gap = gap/2)
for ( p = gap; p < N ; p++)
{
tmp = a[p];
for (j = p; j >= gap && tmp < a[j- gap]; j = j - gap)
a[j] = a[j-gap];
a[j] = tmp;
```

What should the increment sequence be?
There are many choices for the increment sequence. Any sequence that begins at 1 and always increases will do, although some yield better performance than others:
1. Shell's original sequence: N/2 , N/4 , ..., 1 (repeatedly divide by 2);
2. Hibbard's increments: 1, 3, 7, ..., $2^k - 1$;
3. Knuth's increments: 1, 4, 13, ..., $(3^k - 1) / 2$;
4. Sedgewick's increments: 1, 5, 19, 41, 109, ... Each term is of the form either $9 \cdot 4^k - 9 \cdot 2^k + 1$ or $4^k - 3 \cdot 2^k + 1$.

Analysis
A Shellsort's worst-case performance using Hibbard's increments is $\Theta(n^{3/2})$.
The average performance is thought to be about $O(n^{5/4})$
The exact complexity of this algorithm is still being debated .
Experience shows that for mid-sized data (tens of thousands elements) the algorithm performs nearly as well if not better than the faster (*n log n*) sorts.