**SORTING**

**EXPT NO: 11**  **DATE: 6/1/22**

**AIM**

WRAP to implement the following sorting techniques.

i. Bubble Sort

ii. Selection Sort

iii. Insertion Sort

iv. Heap Sort

v. Shell Sort

vi. Merge Sort

vii. Radix Sort

**THEORY**

Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.

**SORTING TECHNIQUES**

**1) BUBBLE SORT**

The bubble sort proceeds by scanning the list and exchanging the adjacent elements if they are out of order with respect to each other. It compares each element with its adjacent and swaps them if they are not in order i.e. arr[i] will be compared with arr[i+1] and if arr[i]>arr[i+1] then they will be swapped.

Bubble sort is not an efficient sorting technique but it is simple and easy to implement.

After first pass the largest element will be at its position in the array i.e. (n-1)th position, after second pass the second largest element will be at its position in the array i.e. (n-2)th position. Similarly after each pass the next larger elements will be moved to the end of the list and placed at their proper positions. If there are n elements, then only n-1 passes are required to sort the array.

In the second pass, comparisons will be done only up to (n-2)th position i.e. the last comparison is done between arr[n-3] and arr[n-2], because the largest element has already been placed at its proper position i.e. position (n-1). In the third pass, the last comparison is done between arr[n-4] and arr[n-3], because the second largest element has already been placed at its proper position i.e. position (n-2). N the last pass there is only one comparison which is between arr[0] and arr[1]. The remaining element arr[0] will definitely be the smallest element, so the whole list is stored after n-1 passes.

**a. Time Complexity**

* **Best Case** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).
* **Average Case** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is O(n2).
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is O(n2).

**b. Space Complexity**

* The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.

**Advantages of using bubble sort**

1. It is simple and easy to implement
2. Additional space requirement is only one temporary variable and it behaves O(n) for sorted array of elements.
3. Bubble sort is stable
4. It should be used for smaller lists

**Disadvantage of bubble sort**

1. Bubble sort should not be used for larger lists

**2) SELECTION SORT**

Suppose that you are given two numbers and asked to arrange them in ascending order. The most intuitive way to do this would be to find the smallest number and put in the place and then find the second smaller number and put it in the second place and so on. This is the simple technique on which selection sort is based. It is named so because in each pass it selects the smallest element and keeps it in its place.

Suppose we have n elements stored in an array arr. First we will search the smallest element from arr[0]………..arr[n-1] and exchange it with arr[0]. This will place the smallest element of list at 0th position of array. Now we will search smallest element from remaining elements arr[1]………….arr[n-1] and exchange it with arr[1]. This will place the second element of the list at 1st position of array. This process is continues till the whole array is sorted.

**a. Time Complexity**

* **Best Case –** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is O(n2)
* **Average Case** – It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is O(n2)
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is O(n2)

**b.** **Space Complexity**

* The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

**Advantages of using selection sort**

1. Selection sort is simple to implement
2. Requires only one temporary variable for swapping elements
3. Data movement is very less

**Disadvantage of selection sort**

1) Selection sort is not stable. It requires only one temporary variable so it is an in-place sort.

**3) INSERTION SORT**

The insertion sort proceeds by inserting each element at the proper place in a sorted list. This is the same technique used by card players for arranging cards. When they receive a card, they place it in the appropriate place among the cards that they have already arranged.

We will consider our list to be divided into two parts – sorted and unsorted. Initially the sorted part contains only the first element of the list and unsorted part contains the rest of the elements. In each pass the first element from the unsorted part is taken and inserted into the sorted part at appropriate place. If there are n elements in the list, the after n-1 passes the unsorted part disappears and our whole list becomes sorted.

**a. Time Complexity**

* **Best Case** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is O(n).
* **Average Case** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is O(n2).
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is O(n2).

**b. Space Complexity**

* The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

**Advantages using insertion sort**

1) It is very efficient when number of elements to be sorted are very less

2) I it helps to find minimum number and greatest number Iinsertion sort is also efficient for lists that are almost sorted

**Disadvantage of using insertion sort**

1. The number of movements:- the elements of the sorted part also have to move to place for another element When the data items to be sorted are large then these movements can prove costly

**4) SHELL SORT**

Shell sort is an improved version of insertion sort. It works by first comparing elements that are far apart and then it compares closer elements, the distance between the elements to be compared reduces with ever pass until the last pass where adjacent elements are compared.

Now let us see the procedure of this sorting technique. In each pass we take a number ‘h’ called the increment. This number decreases in subsequent passes and finally in the last pass it is always 1. In each pass we divide the list into h sublists. Each sublist is created by taking elements that are at a distance of h from each other.

These sublists are separately sorted among themselves by insertion sort and at the end of the pass these sorted sublists are combined. The list that we get after each pass is partially sorted.

Initially the elements which are far apart are compared and then the elements which are closer and so on. The distance between elements being compared decreases with each pass and finally in the last pass adjacent elements are compared. We know that if the list is almost sorted order then insertion sort proves to be very efficient. Another feature of insertion sort is that it is very efficient on small lists. These two aspects of insertion sort are basis of shell sort.

**a. Time Complexity**

* **Best Case** - It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of Shell sort is O(n\*logn).
* **Average Case** -It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Shell sort is O(n\*logn).
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Shell sort is O(n2).

**b. Space Complexity**

* The space complexity of Shell sort is O(1).

**Advantages of shell sort**

1. Shell sort algorithm is only efficient for finite number of elements in an array
2. Shell sort algorithm is faster than bubble sort algorithm

**Disadvantages of shell sort**

1) Shell sort is not stable sort

2) Shell Sort. Shell sort algorithm is complex in structure and bit more difficult to understand

3) Shell sort algorithm is significantly slower than the merge sort, quick sort and heap sort algorithms

**5) MERGE SORT**

Merge sort has O(n log n) performance in both worst and average case. The main task in merge sort is merging two sorted lists into a single sorted list, so let's first examine the process of merging two sorted lists.

If there are two sorted arrays, then process of combining these sorted arrays into another sorted array is called merging. We will take one element from each array, compare them and then take the smaller one in third array. This process will continue until the elements of one array are finished. Then the remaining elements of unfinished array are put in the third array.

Initially i, j and k point to the beginning of the three arrays so i=0, j=0, k=0. The elements arr1[i] and arr2[j] are compared, and the smaller one is copied to the third array arr3 at position k. the variable k is incremented and one of the variable i or j is incremented.

**a. Time Complexity**

* **Best Case** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is O(n\*logn)**.**
* **Average Case** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is O(n\*logn)**.**
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is O(n\*logn)**.**

**b. Space Complexity**

* The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

**Advantage of merge sort**

1. It is quicker for larger lists because unlike insertion and bubble sort it doesn’t go through the whole list several times

**Disadvantages of merge sort**

1. Slower comparative to the other sort algorithms for smaller tasks.

**6) HEAP SORT**

**Heap sort is performed in two phases:-**

**Phase1** – Build a max heap from the given element

**Phase2** – Keep on deleting the root till there is only one element in the tree

The root of the heap always has the largest element so by successively deleting the root, we get the elements in descending order i.e. first the largest element of the list will be deleted then second largest and so on. We can store the deleted elements in a separate array or move them to the end of the same array that represents the heap.

If we have n elements that are to be sorted, first we build a heap of size n. the elements arr[1], arr[2],………arr[n] from the heap. Then the root is deleted and we get a heap of size n-1. So now the elements arr[1], arr[2],……….arr[n-1] form the heap but arr[n] is not part of the heap. The element deleted from the root is the largest element, we can store it in arr[n] because arr[n] is not part of heap now

Now the root from the heap of size n-1 is deleted and we get a heap of size n-2. The element deleted from the root can be stored in arr[n-1]

**a. Time Complexity**

* **Best Case** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is O(n logn).
* **Average Case -**It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is  O(n log n).
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is O(n log n).

**b. Space Complexity**

* The space complexity of Heap sort is O(1).

Heapsort is not a stable sort. It requires only one temporary variable so it is an in-place sort.

**Advantage of heap sort**

1. It helps to find minimum number and greatest number

**Disadvantage of heap sort**

1. It takes so much time to compare and then execute

**7) RADIX SORT**

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

**a. Time Complexity**

* **Best Case** -It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is Ω(n+k)**.**
* **Average Case** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is θ(nk)**.**
* **Worst Case** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is O(nk)**.**

Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).

**b. Space Complexity**

* The space complexity of Radix sort is O(n + k).

Radix sort is a stable sort. It is not an in-place sort.

**Advantage of radix sort**

1. Fast when the keys are short i.e., when the range of the array elements is less

**Disadvantages of radix sort**

1. Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. Hence, for every different type of data it needs to be rewritten
2. It takes more space compared to Quicksort which is in-place sorting

**CODES**

**1)**

#include<stdio.h>

#include<stdlib.h>

void bubblesort(int arr[], int n)

{

int flag;

for(int i=0;i<n-1;i++)

{

flag=1;

for(int j=0;j<n-1-i;j++)

{

if(arr[j]>arr[j+1])

{

int temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

flag=0;

}

}

if(flag)

return;

}

}

int main()

{

int \*arr, n;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(int i=0;i<n;i++)

scanf("%d,",&arr[i]);

bubblesort(arr, n);

printf("\nARRAY AFTER SORTING: ");

for(int i=0;i<n;i++)

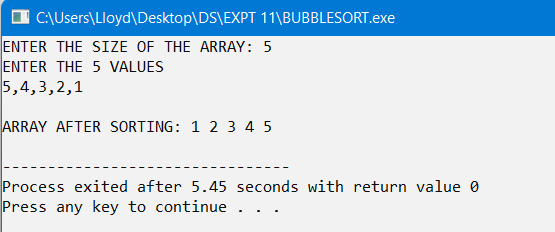
printf("%d ",arr[i]);

printf("\n");

return 0;

}

**OUTPUT**



**2)**

#include<stdio.h>

#include<stdlib.h>

void selectionsort(int arr[], int n)

{

int pos, temp;

for(int i=0;i<n-1;i++)

{

pos=i;

for(int j=i+1;j<n;j++)

{

if(arr[j]<arr[pos])

pos=j;

}

if(i!=pos)

{

temp=arr[i];

arr[i]=arr[pos];

arr[pos]=temp;

}

}

}

int main()

{

int \*arr, n;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(int i=0;i<n;i++)

scanf("%d,",&arr[i]);

selectionsort(arr, n);

printf("\nARRAY AFTER SORTING: ");

for(int i=0;i<n;i++)

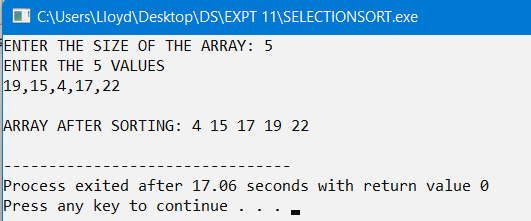
printf("%d ",arr[i]);

printf("\n");

return 0;

}

**OUTPUT**

****

**3)**

#include<stdio.h>

#include<stdlib.h>

void insertionsort(int arr[], int n)

{

int j, temp;

for(int i=1;i<n;i++)

{

j=i-1;

temp=arr[i];

while(arr[j]>temp && j>=0)

{

arr[j+1]=arr[j];

j--;

}

arr[j+1]=temp;

}

}

int main()

{

int n, \*arr;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(int i=0;i<n;i++)

scanf("%d,",&arr[i]);

insertionsort(arr, n);

printf("\nARRAY AFTER SORTING: ");

for(int i=0;i<n;i++)

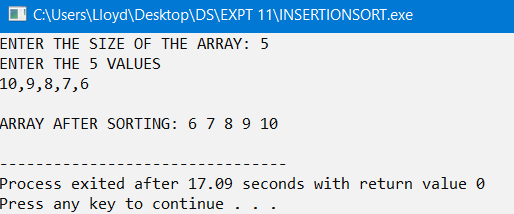
printf("%d ",arr[i]);

printf("\n");

return 0;

}

**OUTPUT**

****

**4)**

#include<stdio.h>

#include<stdlib.h>

void restoredown(int arr[], int i, int n)

{

int left=2\*i, right=left+1, num=arr[i];

while(right<=n)

{

if(num>=arr[left] && num>=arr[right])

{

arr[i]=num;

return;

}

else if(arr[left]>arr[right])

{

arr[i]=arr[left];

i=left;

}

else

{

arr[i]=arr[right];

i=right;

}

left=2\*i;

right=left+1;

}

if(left==n && num<arr[left])

{

arr[i]=arr[left];

i=left;

}

arr[i]=num;

}

void buildheap(int arr[], int n)

{

int i;

for(i=n/2;i>=1;i--)

restoredown(arr, i, n);

}

int del(int arr[], int \*n)

{

int max=arr[1];

arr[1]=arr[\*n];

(\*n)--;

restoredown(arr,1,\*n);

return max;

}

void heapsort(int arr[], int n)

{

int max;

buildheap(arr, n);

// printf("\nHEAP: ");

// for(int i=1;i<=n;i++)

// printf("%d ",arr[i]);

// printf("\n");

while(n>1)

{

max=del(arr, &n);

arr[n+1]=max;

}

}

int main()

{

int \*arr, n,i;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(i=1;i<=n;i++)

scanf("%d,",&arr[i]);

heapsort(arr, n);

printf("\nARRAY AFTER SORTING: ");

for(i=1;i<=n;i++)

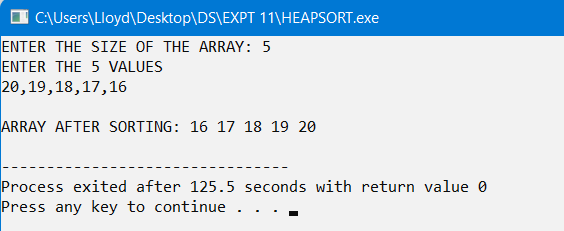
printf("%d ",arr[i]);

printf("\n");

return 0;

}

**OUTPUT**

****

**5)**

#include<stdio.h>

#include<stdlib.h>

void shellsort(int arr[], int incre, int n)

{

int j,k;

while(incre>=1)

{

for(int i=incre;i<n;i++)

{

k=arr[i];

for(j=i-incre;j>=0 && arr[j]>k;j=j-incre)

arr[j+incre]=arr[j];

arr[j+incre]=k;

}

incre-=2;

}

}

int main()

{

int \*arr, n,incre;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(int i=0;i<n;i++)

scanf("%d,",&arr[i]);

printf("ENTER MAXIMUM INCREMENT(ODD VALUE): ");

scanf("%d",&incre);

shellsort(arr, incre, n);

printf("\nARRAY AFTER SORTING: ");

for(int i=0;i<n;i++)

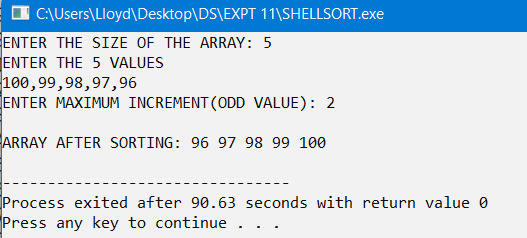
printf("%d ",arr[i]);

printf("\n");

return 0;

}

**OUTPUT**

****

**6)**

#include<stdio.h>

#include<stdlib.h>

void merge(int arr[], int temp[], int l1, int h1, int l2, int h2)

{

int k=l1;

while(l1<=h1 && l2<=h2)

{

if(arr[l1]<=arr[l2])

temp[k++]=arr[l1++];

else

temp[k++]=arr[l2++];

}

while(l1<=h1)

temp[k++]=arr[l1++];

while(l2<=h2)

temp[k++]=arr[l2++];

}

void copy(int arr[], int temp[], int low, int high)

{

for(int i=low;i<=high;i++)

arr[i]=temp[i];

}

void mergesort(int arr[], int low, int high)

{

if(low<high)

{

int mid=(low+high)/2, temp[50];

mergesort(arr, low, mid);

mergesort(arr, mid+1, high);

merge(arr, temp, low, mid, mid+1, high);

copy(arr, temp, low, high);

}

}

int main()

{

int \*arr, n;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(int i=0;i<n;i++)

scanf("%d,",&arr[i]);

mergesort(arr, 0, n-1);

printf("\nARRAY AFTER SORTING: ");

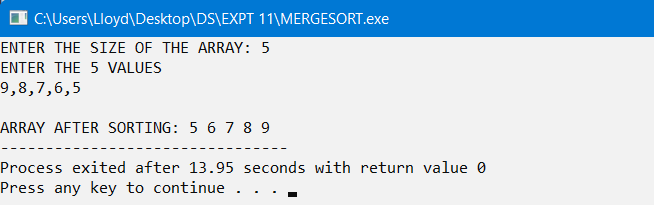
for(int i=0;i<n;i++)

printf("%d ",arr[i]);

return 0;

}

**OUTPUT**

****

**7)**

#include<stdio.h>

#include<stdlib.h>

struct node{

int data;

struct node \*link;

}\*start=NULL, \*front[10], \*rear[10];

int largedigit()

{

int large=0, n=0;

struct node \*p=start;

while(p!=NULL)

{

if(large<p->data)

large=p->data;

p=p->link;

}

while(large!=0)

{

large/=10;

n++;

}

return n;

}

int digit(int num, int k)

{

for(int i=1;i<k;i++)

num/=10;

return num%10;

}

void radixsort()

{

int least\_sig=1, most\_sig=largedigit(), dig;

struct node \*p;

for(int k=least\_sig;k<=most\_sig;k++)

{

for(int i=0;i<10;i++)

{

front[i]=NULL;

rear[i]=NULL;

}

for(p=start;p!=NULL;p=p->link)

{

dig=digit(p->data, k);

if(front[dig]==NULL)

front[dig]=p;

else

rear[dig]->link=p;

rear[dig]=p;

}

int i=0;

while(front[i]==NULL)

i++;

start=front[i];

while(i<9)

{

if(rear[i+1]!=NULL)

rear[i]->link=front[i+1];

else

rear[i+1]=rear[i];

i++;

}

rear[9]->link=NULL;

}

}

int main()

{

struct node \*t, \*p;

int \*arr, n;

printf("ENTER THE SIZE OF THE ARRAY: ");

scanf("%d",&n);

arr=(int \*)malloc(n\*sizeof(int));

printf("ENTER THE %d VALUES\n",n);

for(int i=0;i<n;i++)

{

t=(struct node\*)malloc(sizeof(struct node));

scanf("%d,",&t->data);

t->link=NULL;

if(start==NULL)

start=t;

else{

p=start;

while(p->link!=NULL)

p=p->link;

p->link=t;

}

}

radixsort();

printf("\nARRAY AFTER SORTING: ");

p=start;

while(p!=NULL)

{

printf("%d ",p->data);

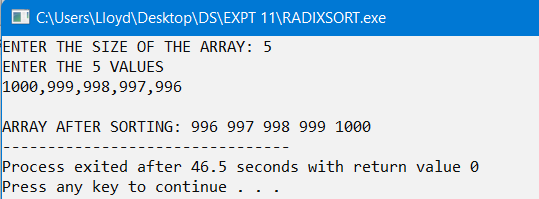
p=p->link;

}

return 0;

}

**OUTPUT**

****

**CONCLUSION**

The given problem statements were successfully compiled and executed.

**LEARNINGS AND FINDINGS**

1) Sorting Techniques.

2) Time and space complexities.

3) Implementation.

4) Advantages and disadvantages.

All the sorting techniques learnt through this experiment have their own advantages and shortcomings.

|  |  |
| --- | --- |
| **SR. NO.** | **COMPILATION TIME** |
| 1 | 0.19 s |
| 2 | 0.19 s |
| 3 | 0.20 s |
| 4 | 0.19 s |
| 5 | 0.19 s |
| 6 | 0.19 s |
| 7 | 0.18 s |