**EXPERIMENT NO. 08**

|  |  |
| --- | --- |
| **DATE OF PERFORMANCE:** | **GRADE:** |
| **DATE OF ASSESSMENT:** | **SIGNATURE OF LECTURER/ TTA:** |
|  |  |

**AIM: IMPLEMENTATION OF SORTING ALGORITHMS.**

**THEORY:**

**SORTING:** Sorting is the process of placing elements from a collection in some kind of order. Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted.

**SORTING EFFICIENCY**

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

**TYPES OF SORTING TECHNIQUES**

There are many types of Sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort

**BUBBLE SORT:** Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order. This algorithm is not suitable for large data sets

Insertion Sort: This is a in-place comparison based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. A element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and insert it there. Hence the name insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array).

**ALGORITHM:**

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**INSERTION SORT:** This is a in-place comparison based sorting algorithm. Here, a sub-list is maintained which is always sorted. The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array).

**ALGORITHM:**

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

**SELECTION SORT:** Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.

**ALGORITHM:**

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

**QUICK SORT:** Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than specified value say pivot based on which the partition is made and another array holds values greater than pivot value.

The quick sort partitions an array and then calls itself recursively twice to sort the resulting two subarray.

**QUICKSORT PIVOT ALGORITHM**

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

**QUICKSORT ALGORITHM**

Using pivot algorithm recursively we end-up with smaller possible partitions. Each partition then processed for quick sort. We define recursive algorithm for quick sort as below −

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 – quick sort left partition recursively

Step 4 – quick sort right partition recursively

**MERGE SORT:** Merge sort is a sorting technique based on divide and conquer technique. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

**ALGORITHM:**

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then merge sort combines smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

**PROGRAM 1: IMPLEMENTATION OF BUBBLE SORT.**

**#include<stdio.h>**

**#include<conio.h>**

**void main()**

**{**

**int a[100],i,j,s,p;**

**clrscr();**

**printf("\n Enter size of array:\n");**

**scanf("%d",&p);**

**printf("\n Enter elements:\n");**

**for(i=0;i<p;i++)**

**{**

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

**}**

**for(i=0;i<p-1;i++)**

**{**

**for(j=0;j<p-i-1;j++)**

**{**

**if(a[j]>a[j+1])**

**{**

**s=a[j];**

**a[j]=a[j+1];**

**a[j+1]=s;**

**}**

**}**

**}**

**printf("\n Sorted list:\n");**

**for(i=0;i<p;i++)**

**{**

**printf("%d\n",a[i]);**

**}**

**getch();**

**}**

**OUTPUT:**

**PROGRAM 2: IMPLEMENTATION OF INSERTION SORT.**

**#include<stdio.h>**

**#include<conio.h>**

**void main()**

**{**

**int n,a[100],c,d,t;**

**clrscr();**

**printf("\n Enter number of elements:\n");**

**scanf("%d",&n);**

**printf("\n Enter elements:\n");**

**for(c=0;c<n;c++)**

**{**

**scanf("%d",&a[c]);**

**}**

**for(c=1;c<=n-1;c++)**

**{**

**d=c;**

**while(d>0 && a[d]<a[d-1])**

**{**

**t=a[d];**

**a[d]=a[d-1];**

**a[d-1]=t;**

**d- -;**

**}**

**}**

**printf("\n Sorted list:\n");**

**for(c=0;c<n;c++)**

**{**

**printf("%d\n",a[c]);**

**}**

**getch();**

**}**

**OUTPUT:**

**PROGRAM 3: IMPLEMENTATION OF SELECTION SORT.**

**#include<stdio.h>**

**#include<conio.h>**

**void main()**

**{**

**int a[100],i,j,s,p,pos;**

**clrscr();**

**printf("\n Enter size of array:\n");**

**scanf("%d",&p);**

**printf("\n Enter elements:\n");**

**for(i=0;i<p;i++)**

**{**

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

**}**

**for(i=0;i<p-1;i++)**

**{**

**pos=i;**

**for(j=i+1;j<p;j++)**

**{**

**if(a[pos]>a[j])**

**{**

**pos=j;**

**}**

**}**

**if(pos!=i)**

**{**

**s=a[i];**

**a[i]=a[pos];**

**a[pos]=s;**

**}**

**}**

**printf("\n Sorted list:\n");**

**for(i=0;i<p;i++)**

**{**

**printf("%d\n",a[i]);**

**}**

**getch();**

**}**

**OUTPUT:**

**PROGRAM 4: IMPLEMENTATION OF QUICK SORT.**

**#include<stdio.h>**

**void swap (int a[], int left, int right)**

**{**

**int temp;**

**temp=a[left];**

**a[left]=a[right];**

**a[right]=temp;**

**}//end swap**

**void quicksort( int a[], int low, int high )**

**{**

**int pivot;**

**// Termination condition!**

**if ( high > low )**

**{**

**pivot = partition( a, low, high );**

**quicksort( a, low, pivot-1 );**

**quicksort( a, pivot+1, high );**

**}**

**} //end quicksort**

**int partition( int a[], int low, int high )**

**{**

**int left, right;**

**int pivot\_item;**

**int pivot = left = low;**

**pivot\_item = a[low];**

**right = high;**

**while ( left < right )**

**{**

**// Move left while item < pivot**

**while( a[left] <= pivot\_item )**

**left++;**

**// Move right while item > pivot**

**while( a[right] > pivot\_item )**

**right--;**

**if ( left < right )**

**swap(a,left,right);**

**}**

**// right is final position for the pivot**

**a[low] = a[right];**

**a[right] = pivot\_item;**

**return right;**

**}//end partition**

**// void quicksort(int a[], int, int);**

**void printarray(int a[], int);**

**int main()**

**{**

**int a[50], i, n;**

**printf("\nEnter no. of elements: ");**

**scanf("%d", &n);**

**printf("\nEnter the elements: \n");**

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

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

**printf("\nUnsorted elements: \n");**

**printarray(a,n);**

**quicksort(a,0,n-1);**

**printf("\nSorted elements: \n");**

**printarray(a,n);**

**}//end main**

**void printarray(int a[], int n)**

**{**

**int i;**

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

**printf(" %d ", a[i]);**

**printf("\n");**

**}**

**OUTPUT:**

**PROGRAM 5: IMPLEMENTATION OF MERGE SORT.**

**#include<stdio.h>**

**#include<conio.h>**

**int a[50];**

**void merge(int,int,int);**

**void merge\_sort(int low,int high)**

**{**

**int mid;**

**if (low<high)**

**{**

**mid=(low+high)/2;**

**merge\_sort(low,mid);**

**merge\_sort(mid+1,high);**

**merge(low,mid,high);**

**}**

**}**

**void merge(int low, int mid, int high)**

**{**

**int h,i,j,b[50],k;**

**h=low;**

**i=low;**

**j=mid+1;**

**while((h<=mid)&&(j<=high))**

**{**

**if (a[h]<=a[j])**

**{**

**b[i]=a[h];**

**h++;**

**}**

**else**

**{**

**b[i]=a[j];**

**j++;**

**}**

**i++;**

**}**

**if (h>mid)**

**{**

**for(k=j;k<=high;k++)**

**{**

**b[i]=a[k];**

**i++;**

**}**

**}**

**else**

**{**

**for(k=h;k<=mid;k++)**

**{**

**b[i]=a[k];**

**i++;**

**}**

**}**

**for(k=low;k<=high;k++)**

**a[k]=b[k];**

**}**

**int main()**

**{**

**int num,i;**

**printf("\nEnter the total numbers: ");**

**scanf("%d",&num);**

**printf("\nEnter %d numbers: \n",num);**

**for(i=1;i<=num;i++)**

**{**

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

**}**

**merge\_sort(1,num);**

**printf("\nSORTED ORDER: \n");**

**for(i=1;i<=num;i++)**

**printf("\t%d",a[i]);**

**getch();**

**}**

**OUTPUT:**