|  |
| --- |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **CS 201–DATA STRUCTURES LAB**  **Lab Session 08** |
| **Instructors:** Mr. Irfan , Ms. Mubashra |

**Objective**: In this lab session we will do basic sorting algorithm

This term is used to describe the process of organizing data in a particular order. For example you have an array of 5 numbers [2,4,7,3,1] after sorting it will become [1,2,3,4,7]. We will explore different techniques of sorting, in this lab session we will mainly focus on 3 basic sorting techniques which are as follows:

* Bubble sort
* Selection sort
* Insertion sort

**Bubble sort:**

The bubble sort algorithm is one of the simplest sorting methods to implement. The way it works is by repeatedly going through the array to be sorted, comparing (and swapping, if in the wrong order) two adjacent elements at a time.

**How Bubble Sort Works:**

* **I have an array of 5 numbers:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **1** | **12** | **-5** | **16** |

* **First iteration:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **1** | **12** | **-5** | **16** |

**5>1 , swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **5** | **12** | **-5** | **16** |

**5<12, no action**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **5** | **12** | **-5** | **16** |

**12>-5, swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **5** | **-5** | **12** | **16** |

**12<16, no action**

* **Second iteration:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **5** | **-5** | **12** | **16** |

**1<5, no action**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **5** | **-5** | **12** | **16** |

**5>-5, swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **-5** | **5** | **12** | **16** |

**5<12, no action**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **-5** | **5** | **12** | **16** |

**12<16, no action**

* **Third iteration:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **1** | **-5** | **5** | **12** | **16** |

**1>-5, swap**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **-5** | **1** | **5** | **12** | **16** |

**1<5, no action**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **-5** | **1** | **5** | **12** | **16** |

**5<12, no action**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **-5** | **1** | **5** | **12** | **16** |

**12<16, no action**

* **Sorted Array:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **-5** | **1** | **5** | **12** | **16** |

**Algorithm:**

Void bubblesort (int array[], int n)

{

for i from 1 to n

for j from 0 to n - i

if a[j] > a[j + 1]

swap( a[j], a[j + 1] )

}

**Selection Sort:**

It is also one of simplest sorting algorithm. We repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the smallest element and moving it to the first position. We can do this by swapping the smallest element at the first index of array. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

**How selection sort works:**

* Consider an array of size 10

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 23 | 4 | 5 | 0 | 1 | 2 | 7 | 8 | 9 |

* At 0 location we put smallest element by traversing whole array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 23 | 4 | 5 | 12 | 1 | 2 | 7 | 8 | 9 |

* At 1st location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 4 | 5 | 12 | 23 | 2 | 7 | 8 | 9 |

* At 2nd location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 5 | 12 | 23 | 4 | 7 | 8 | 9 |

* At 3rd location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 4 | 12 | 23 | 5 | 7 | 8 | 9 |

* At 4th location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 4 | 5 | 23 | 12 | 7 | 8 | 9 |

* At 5th location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 4 | 5 | 7 | 12 | 23 | 8 | 9 |

* At 6th location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 4 | 5 | 7 | 8 | 23 | 12 | 9 |

* At 7th location we put another smallest element by traversing sub-array

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 4 | 5 | 7 | 8 | 9 | 12 | 23 |

Now the array is sorted

**Algorithm of selection sort:**

for out = 1:n,

min = out

for in = out+1:n

if a[min] > a[in]

min = in

swap a[min,out]

end

**Insertion sort:**

Insertion sort is a simple [sorting algorithm](http://www.wikipedia.org/wiki/Sorting_algorithm), a [comparison sort](http://www.wikipedia.org/wiki/Comparison_sort) in which the sorted array (or list) is built one entry at a time. An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.

**How it works:**

* Consider an array of 5 elements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 9 | 0 | 6 |

* On 1st iteration

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 4 | 9 | 0 | 6 |

* On 2nd iteration

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 2 | 4 | 9 | 6 |

* On third iteration

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 2 | 4 | 6 | 9 |

**Algorithm of insertion sort:**

I

|  |
| --- |
| INSERTION\_SORT (*A*)  1.     **FOR** j ← 2 **TO** length[*A*]  2.             **DO**  key ← *A*[*j*]     3.                   {Put *A*[*j*] into the sorted sequence *A*[1 . . *j* − 1]}    4.                    *i* ← *j* − 1     5.                    **WHILE** *i* > 0 and *A*[*i*] > key 6.                                 **DO** *A*[*i* +1] ← *A*[*i*]             7.                                         *i* ← *i* − 1      8.                     *A*[*i* + 1] ← key |

**Merge Sort:**

**#include <iostream>**

**using namespace std;**

**class Sorting**

**{**

**public:**

**void merge(int a[], const int low, const int mid, const int high)**

**{**

**// Variables declaration.**

**int \* b = new int[high+1-low];**

**int h,i,j,k;**

**h=low;**

**i=0;**

**j=mid+1;**

**// Merges the two array's into b[] until the first one is finish**

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

**{**

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

**{**

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

**h++;**

**}**

**else**

**{**

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

**j++;**

**}**

**i++;**

**}**

**// Completes the array filling in it the missing values**

**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++;**

**}**

**}**

**// Prints into the original array**

**for(k=0;k<=high-low;k++)**

**{**

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

**}**

**delete[] b;**

**}**

**void Print(int a[],int b,int c)**

**{**

**for(int k=b;k<=c;k++)**

**{**

**cout<<a[k]<<endl;**

**}**

**}**

**void merge\_sort(int a[], const int low, const int high)// Recursive**

**{**

**int mid;**

**if(low<high)**

**{**

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

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

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

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

**}**

**}**

**};**

**int main()**

**{**

**int arraySize=11;**

**int a[]={2,11,3,43,21,211,3,22,12,44,122};**

**Sorting s;**

**s.merge\_sort(a, 0, (arraySize-1) );**

**s.Print(a,0,(arraySize-1));**

**system("pause");**

**return 0;**

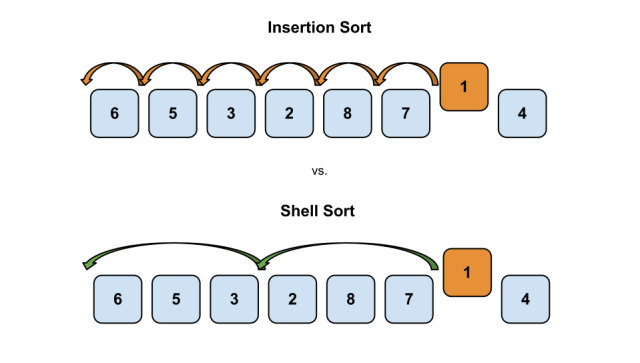
**}**

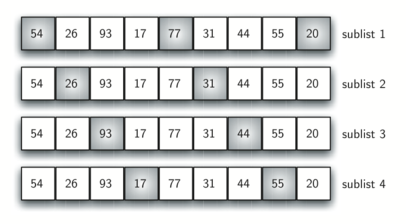
**Task1: Before running the above complete code, run debug on merg\_sort function to check the recursive calls behavior to partition array into one element (sorted already);**

**Shell Sort:**

Insertion sort is a great algorithm, because it’s very intuitive and it is easy to implement, but the problem is that it makes many exchanges for each “light” element in order to put it on the right place. Thus “light” elements at the end of the list may slow down the performance of insertion sort a lot. That is why in 1959 Donald Shell proposed an algorithm that tries to overcome this problem by comparing items of the list that lie far apart.

In the other hand it is obvious that by comparing items that lie apart the list can’t be sorted in one pass as insertion sort. That is why on each pass we should use a fixed gap between the items, then decrease the value on every consecutive iteration.





|  |
| --- |
| **#include<iostream>**  **using namespace std;**  **class shellsort**  **{**  **private:**  **int no\_of\_elements;**  **int elements[10];**  **public:**  **void getarray();**  **void sortit(int [],int);**  **int return\_no\_elements();**  **void print();**  **};**  **void shellsort::getarray()**  **{**  **cout<<"how many elements: ?";**  **cin>>no\_of\_elements;**  **cout<<"insert elemet to sort:"<<endl;**  **for(int i=0;i<no\_of\_elements;i++)**  **{**  **cin>>elements[i];**  **}**  **}**  **int shellsort:: return\_no\_elements()**  **{**  **return no\_of\_elements;**  **}**  **void shellsort::sortit(int inc[],int incnum)**  **{**  **int incr,j,k,span, y;**  **for(incr=0;incr<incnum;incr++)**  **{**  **span=inc[incr];**  **for(j=span;j<no\_of\_elements;j++)**  **{**  **y=elements[j];**  **for(k=j-span;k>=0&& y<elements[k]; k-=span)**  **{**  **elements[k+span]=elements[k];**  **}**  **elements[k+span]=y;**  **}**  **cout<<"iteration= "<<incr+1<<"span = "<<span<<" : ";**  **print();**  **if(span==1)**  **break;**  **}**  **}**  **void shellsort:: print()**  **{**  **for(int i=0;i<no\_of\_elements;i++)**  **{**  **cout<<elements[i]<<" ";**  **}**  **cout<<endl;**  **}**  **int main()**  **{**  **shellsort ss;**  **int n,i,j;**  **ss.getarray();**  **n=ss.return\_no\_elements();**  **int incrm[n];**  **for(i=n,j=0;i>0;i=i/2,j++)**  **{**  **incrm[j]=i;**  **}**  **ss.sortit(incrm,j+1);**  **//system("pause");**  **}** |

**Quick Sort:**

|  |
| --- |
| **#include <iostream>**  **#include <conio.h>**  **using namespace std;**  **static const int size =10;**  **//int arr1[size] = {12,24,17,34,6,10,6,3,8,9,1};**  **//int arr4[size] = {1,2,3,4,5,6,7,8,9,10,11};**  **//int arr5[size] = {8,5,4,7,6,1,6,3,8,12,10};**  **int arr6[size] = {60,24,1,33,99,45,78,2,11,23};**  **class Sorting**  **{**  **public:**  **void PrintArray(int input[])**  **{**  **for ( int i = 0; i < size; i++ )**  **cout << input[i] << " ";**  **cout << endl;**  **}**  **void Swap(int\* i, int\* j)**  **{**  **int tmp;**  **tmp = \*i;**  **\*i = \*j;**  **\*j = tmp;**  **}**  **};//end of class Sorting**  **int main()**  **{**  **Sorting sort;**  **getch();**  **return 0;**  **}** |

**Step 2: Quick Sort**

|  |
| --- |
| **void quick\_sort(int data[], int low, int high)**  **{**  **int i = low;**  **int j = high;**  **int pivot = data[low + ((high - low)/2)];**    **while (i <= j)**  **{**  **while (data[i] < pivot) i++;**  **while (data[j] > pivot) j--;**  **if (i <= j)**  **{**  **Swap(&data[i], &data[j]);**  **i++;**  **j--;**  **}**  **}**  **if (low < j)**  **quick\_sort(data, low, j);**  **if (i < high)**  **quick\_sort(data, i, high);**  **}** |
| ***Function call from main***  **sort.quick\_sort(arr6,0,size-1);** |

**Radix Sort:**

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines.

Basic Steps to Be Performed:

Each key is first figuratively dropped into one level of buckets corresponding to the value of the rightmost digit. Each bucket preserves the original order of the keys as the keys are dropped into the bucket. There is a one-to-one correspondence between the number of buckets and the number of values that can be represented by a digit. Then, the process repeats with the next neighboring digit until there are no more digits to process[.](http://adf.ly/KiLQs) In other words:

1. Take the least significant digit of each key.
2. Group the keys based on that digit, but otherwise keep the original order of keys.
3. Repeat the grouping process with each more significant digit.

The sort in step 2 is usually done using bucket sort or counting sort, which are efficient in this case since there are usually only a small number of digits.

|  |
| --- |
| **/\*Safia Baloch: FAST-NUCES\*/**  **#include <iostream>**  **#include <conio.h>**  **#define MAX 10**  **using namespace std;**  **class radixsort{**  **int arr[MAX],n;**  **public:**  **void getdata()**  **{**  **cout<<"How many elements you require : ";**  **cin>>n;**  **for(int i=0;i<n;i++)**  **cin>>arr[i];**  **}**  **void showdata()**  **{**  **cout<<"\n--Display--\n";**  **for(int i=0;i<n;i++)**  **cout<<arr[i]<<" ";**  **}**  **void sortLogic()**  **{**  **//for base 10int temp;**  **int bucket[10][20], buck\_count[10], b[10];**  **int i,j,k,r,no\_of\_passes=0,divisor=1,largest,pass\_no;**  **largest=arr[0];**  **for(i=1;i<n;i++) //Find the largest Number**  **{**  **if(arr[i] > largest)**  **largest=arr[i];**  **}**  **while(largest > 0) //Find number of digits in largest number**  **{**  **no\_of\_passes++;**  **largest /= 10;**  **}**  **for(pass\_no=0; pass\_no < no\_of\_passes; pass\_no++)**  **{**  **for(k=0; k<10; k++)**  **{**  **buck\_count[k]=0; //Initialize bucket countfor(i=0;i<n;i++){**    **}**  **for (i=0 ; i<n ; i++)**  **{**  **r=(arr[i]/divisor) % 10;**  **bucket[r][buck\_count[r]++]=arr[i];**  **}**  **i=0; //collect elements from bucketfor(k=0; k<10; k++){**  **for (k=0;k<10;k++)**  **for(j=0; j<buck\_count[k]; j++)**  **arr[i++] = bucket[k][j];**    **divisor \*= 10;**  **}**  **}**  **};**  **int main(){**    **cout<<"\n\*\*\*\*\*Radix Sort\*\*\*\*\*\n";**  **radixsort obj;**  **obj.getdata();**  **obj.sortLogic();**  **obj.showdata();**    **}** |

**Task:**

1. Run the above radix code and find is it LSD or MSD, re-implement radix with your algorithm.
2. Take file of unsorted elements, put all above sorting functions in single program and calculate the time.
3. Identify the total number of comparisons in Quick sort, for sorted array(best case), unsorted array (reverse order (worst case) and average case.
4. Run the above code, give a print of different output.
5. Given array, 23,3,7,13,89,7,66,2,6,44,18,90,98,57. Show the content after it has gone through a one-increment pass of the shell sort. The increment factor is k=3
6. create a class having function which take an array as argument and sort the array using bubble sort algorithm? Also print each pass or iteration.
7. The function bubblesort( ) is inefficient because it continues execution after an array is sorted by performing unnecessary comparisons. Therefore, the number of comparisons in the best and worst case is the same.
8. The implementation can be improved by making a provision for the case when the array is already sorted. Modify bubble sort() by adding a flag (Boolean) indicating whether or not it is necessary to make the next pass.
9. you can also do sorting in characters i-e A is smaller than B, and B is smaller than C, So do sorting for character array by using bubble sort algorithm and also show its output.