Sorting

Introduction

In many applications it is necessary to order give objects as per an attribute. For example, arranging a list of student information in increasing order of their roll numbers or arranging a set of words in alphabetical order. A **file** of size n is a sequence of n items $r[0], r[1] \dots \dots . r[n-1]$. Each item in the file is called a **record. Sorting** is the process of placing elements from a collection in some kind of order. The main purpose of sorting information is to optimize its usefulness for specific tasks. The attribute by which objects are arranged or sorted is called **key**. In the first example roll number is the key. A key is usually subfield of the entire record.

Classification

By Nature of Comparison

In this method, sorting algorithms are classified based on the number of comparisons. Comparison based sorting algorithms evaluate the elements of the list by key comparison operation. Most important methods in this class are Bubble, Insertion, shell, Selection, Quick, and Merge sort algorithms.

By Memory Usage

Some sorting algorithms are *"in place"* and they need $O(1)$ or $O(logn)$ memory to create auxiliary location for sorting the data temporarily. Other classifications under memory usage is

- Internal Sort
- External Sort

In **Internal sorting** all the data to sort is stored in memory at all times while sorting is in progress. In **External sorting** data is stored outside memory (like on hard disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into memory entirely.

In internal sorting you can access whatever array elements you want at whatever moment you want. You can't do that in external sorting - the array is not entirely in memory, so you can't just randomly access any element in memory and accessing it randomly on disk is usually extremely slow. The external sorting algorithm has to deal with loading and unloading chunks of data in optimal manner.

By Stability

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Or in other words a sorting technique is called stable if for all records I and j such that $k[i]$ equals $k[j]$, if $r[i]$ precedes $r[j]$ in the original file , $r[i]$ precedes $r[j]$ in the sorted file . Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

Sorting by address

Suppose that the amount of data stored in each of the records in the file is so large that the overhead involved in moving the actual data is prohibitive. In this case an auxiliary table of pointers may be used so that these pointers are moved instead of actual data as shown in figure

Sorting Actual Records

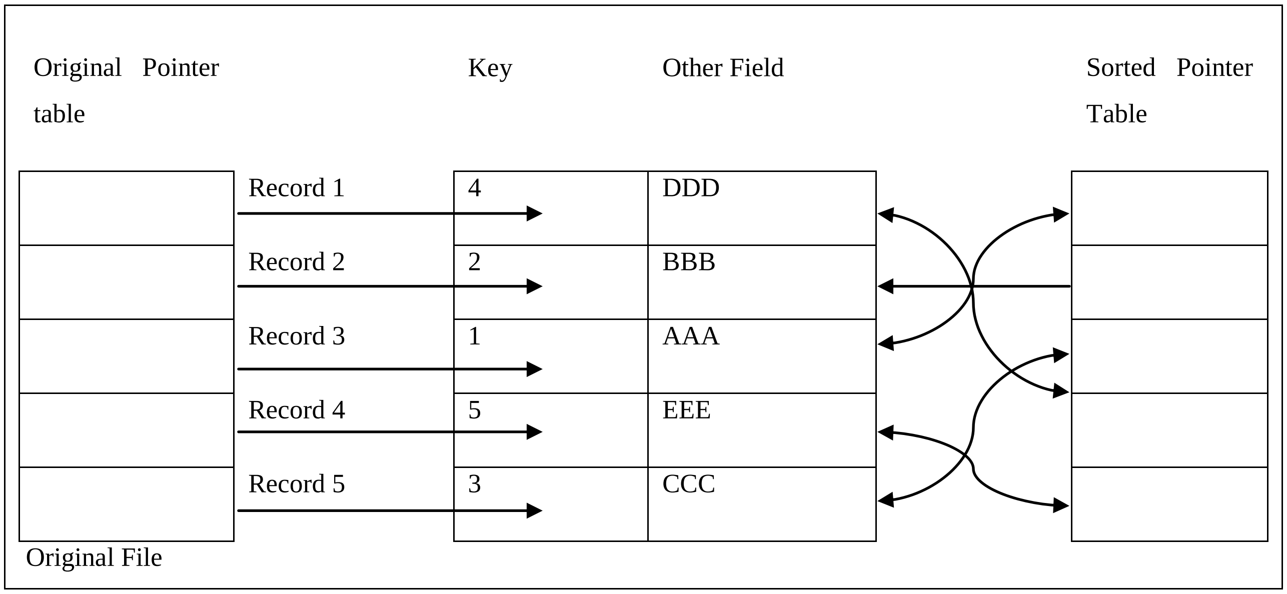
	Key	Other Field
Record 1	4	DDD
Record 2	2	BBB
Record 3	1	AAA
Record 4	5	EEE
Record 5	3	CCC

Original File

Key	Other Field
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Sorted File

Sorting by using Auxiliary table of pointers



The table in the center is the file and the table at the left is the initial table of pointer. The entry in position j in the table of pointers points to record *j*. During the sorting process, the entries in the pointer table are adjusted so that the final table is as showing at the right.

Bubble Sort

Bubble sort, sometimes referred to as sinking 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. Among various other sorting algorithm, bubble sort algorithm is one of the popular and frequently used algorithm to sort elements either in ascending or descending order.

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, last and second last elements are not compared because; the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

Pseudo code

```
procedure bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
     swapped = false
     for i = 1 to n-1 inclusive do
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

Example 1

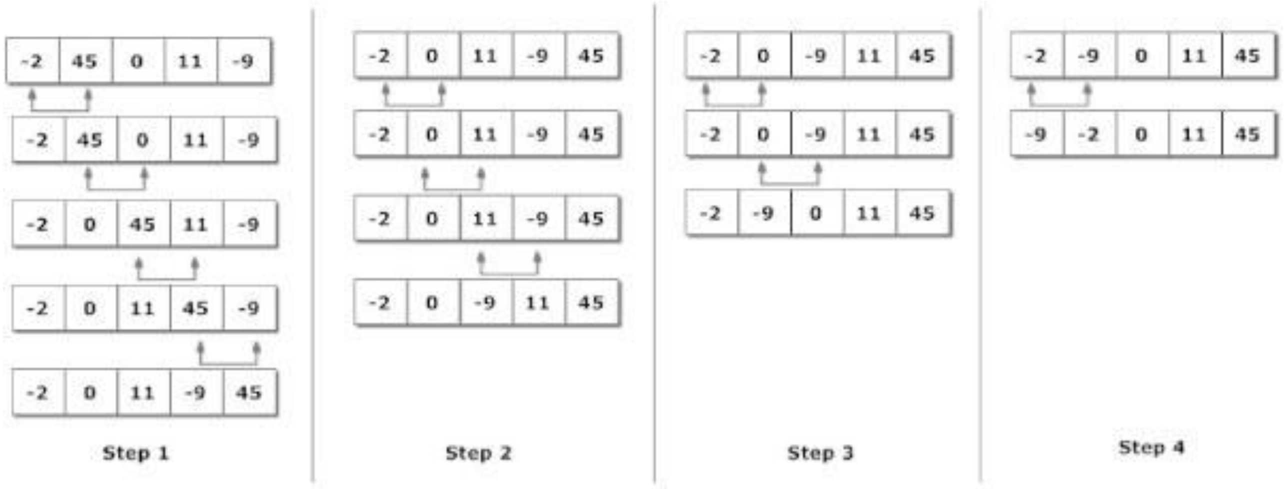
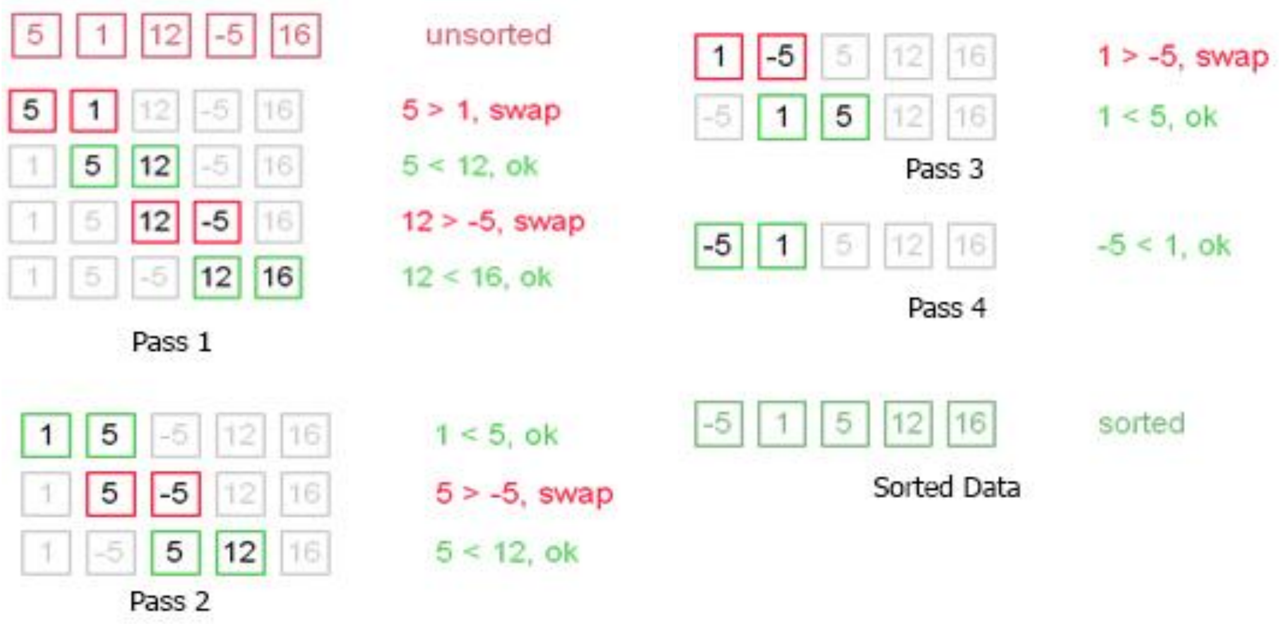
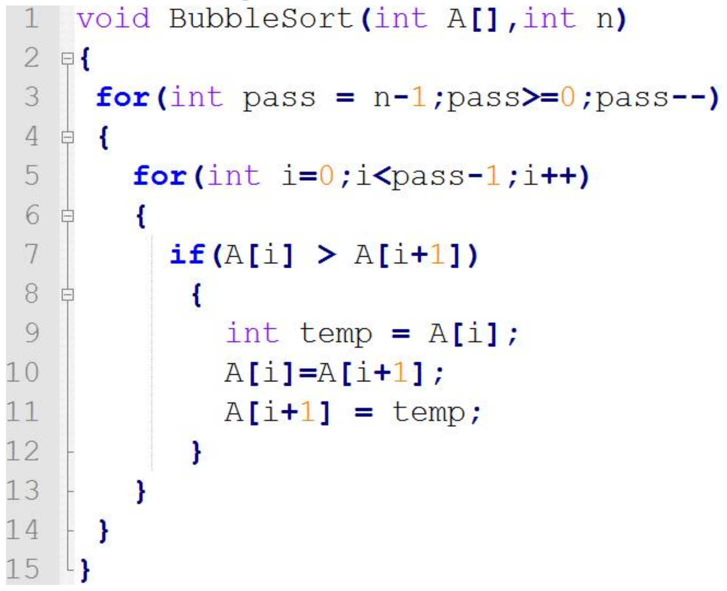


Figure: Working of Bubble sort algorithm

Example 2



Bubble Sort C Program



```c
void BubbleSort(int A[],int n)
{
 for(int pass = n-1;pass>=0;pass--)
 {
   for(int i=0;i<pass-1;i++)
   {
     if(A[i] > A[i+1])
     {
       int temp = A[i];
       A[i]=A[i+1];
       A[i+1] = temp;
     }
   }
 }
}
```

Performance Analysis

In bubble sort $n-1$ comparisons will be done in 1st pass, $n-2$ comparisons will be done in 2nd pass, n – 3 comparisons will be done in 3rd pass and so on. Algorithm requires $n-1$ passes. In each pass it places one item in its correct position.

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

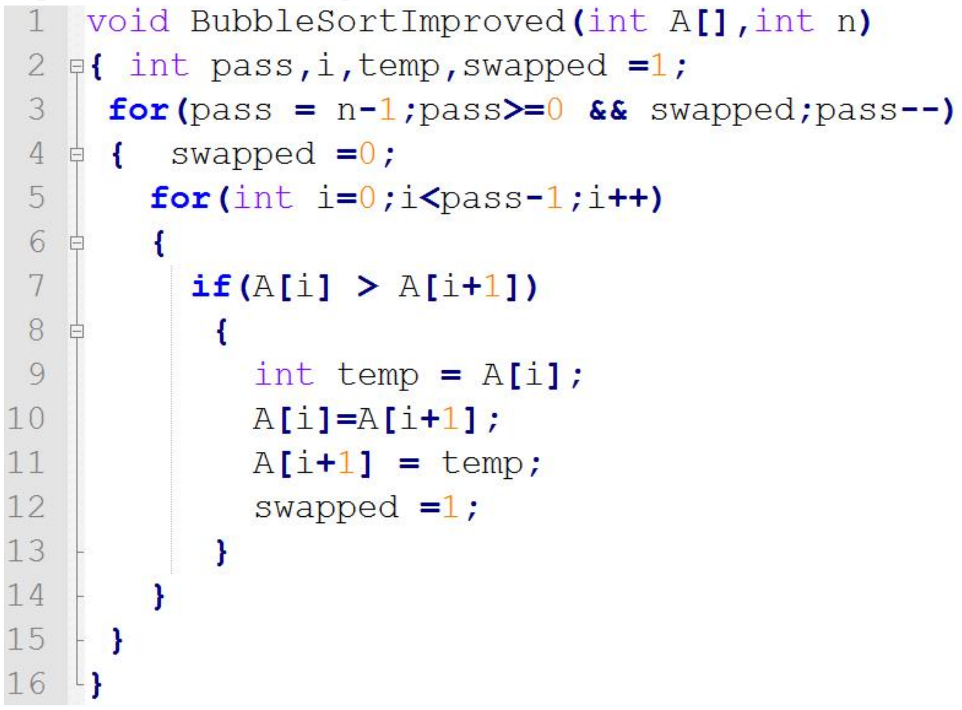
$$T(n) = \sum\_(i = 1)\hat{}(n-1)▒ i$$

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

$$T(n) = O(n\hat{}2)$$

This Algorithm takes $O(n^2)$ *(Even in best case)*. We can improve it by using one extra flag. When there are no more swaps, Indicates the completion of sorting. If the list is already sorted, by using swaps we can skip the remaining passes.

Improved Bubble Sort Implementation



```
void BubbleSortImproved(int A[],int n)
{ int pass,i,temp,swapped =1;
 for(pass = n-1;pass>=0 && swapped;pass--)
 {   swapped =0;
    for(int i=0;i<pass-1;i++)
    {
      if(A[i] > A[i+1])
       {
         int temp = A[i];
         A[i]=A[i+1];
         A[i+1] = temp;
         swapped =1;
       }
    }
 }
}
```

This Modified Version Improves the best case of bubble sort to (n) .

Performance

Worst Case Complexity	$O(n^2)$
Best Case Complexity (Improved Version)	$O(n)$
Average Case Complexity (Basic Version)	$O(n^2)$
Worst Case Space Complexity	$O(1)$ Auxiliary