INTRODUCTION TO SORTING

Sorting techniques can be classified into two types **Internal sorting techniques and External sorting techniques**.

- Any sort algorithm that uses main memory exclusively during the sorting is called as internal sort algorithms. Internal sorting is faster than external sorting. Some example internal sorting algorithms are **Insertion Sort, Bubble Sort, Selection Sort, Heap Sort, Shell Sort, Bucket Sort, Quick Sort, Radix Sort.**
- Any sort algorithm that uses external memory, such as tape or disk, during the sorting is called as external sort algorithms**. Merge Sort** is one of the external sort algorithms.

INSERTION SORT:

Insertion sort is an application of the decrease by one algorithm design technique to sort an array A[0...n1].It is a very simple and efficient sorting algorithm for sorting a small number of elements in which the sorted array is built one element at a time. The main idea behind insertion sort is that it inserts each element into its proper location in the sorted array.

Let us take there are n elements the array arr. Then process of inserting each element in proper place is as-
Pass 1- arr[0] is already sorted because of only one
element. Pass 2-arr[1] is inserted before or after arr[0].
So arr[0] and arr[1] are sorted.
Pass 3- arr[2] is inserted before arr[0] or in between arr[0] and arr[1]
or after arr[1]. So arr[0], arr[1] and arr[2] are sorted
Pass 4- arr[3] is inserted into its proper place in array arr[0],
arr[1], arr[2] So, arr[0] arr[1] arr[2] and arr[3] are sorted.
.....
.....
.....
Pass N - arr[n-1] is inserted into its proper place in array. arr[0], arr[1],
arr[2],..... arr[n-2].
So, arr[0] arr[1],..... arr[n-1] are sorted.

Example:
Sort below elements in increasing order using insertion sort:

5 2 1 3 6 4

Compare first and second elements i.e., 5 and 2, arrange in increasing order. **2 5** 1 3 6 4

Next, sort the first three elements in increasing order.

1 2 5 3 6 4

Next, sort the first four elements in increasing order.

1 2 3 5 6 4

Next, sort the first five elements in increasing order.

1 2 3 5 6 4

Next, sort the first six elements in increasing order.

1 2 3 4 5 6

Algorithm:

Insertion(int a[ ],int n)

1. Start
2. set i=1
3. repeat the steps 4,5,8 and 9 while(i<n)
4. set key=a[i] and j=i-1
5. repeat the steps 6 and 7 while j>=0 && a[j]>key
6. set a[j+1] =a[j]
7. j=j-1
8. set a[j+1]=key
9. set i=i+1
10. stop

SELECTION SORT

As the name suggests selection sort is the selection of an element and keeping it in sorted order. If we have a list of elements in unsorted order and we want to make a list of elements in sorted order then first we will take the smallest element and keep in the new list, after that second smallest element and so on until the largest element of list.

Let us take an array a[0],a[1],……a[n-1] of elements. First we will search the position of the smallest element form a[0]…….a[n-1]. Then we will interchange that smallest element with a[0]. Now we will search position of smallest element(second smallest element because a[0] is the first smallest element) from a[1]……..a[n-1],then interchange that smallest element with a[1]. Similarly the process will be for a[2]……a[n-1].

Example:

1	2	3	4	5	6	7	8	9	10
42	23	74	11	65	58	94	36	99	87

Find the minimum element and if it is not the first element, then interchange first element and the minimum element. Then the list becomes

11 23 74 42 65 58 94 36 99 87—pass 1

Find the second minimum element and if it is not the second element, then interchange second element and the second minimum element. Then the list becomes

11 **23** 74 42 65 58 94 36 99 87---pass 2

Continue till pass 9

11 23 **36** 42 65 58 94 **74** 99 87---pass 3

11 23 36 42 65 58 94 74 99 94---pass 4

11 23 36 42 **58** **65** 94 74 99 87---pass 5

11 23 36 42 **58** 65 94 74 99 87---pass 6

11 23 36 42 58 65 **74** **94** 99 87---pass 7

11 23 36 42 58 65 74 **87** 99 **94**---pass 8

11 23 36 42 58 65 74 87 **94** 99---pass 9

Algorithm:

Selection(int a[ ],int n)

1. start
2. set i=0
3. repeat steps 4,5 and 7 while(i<n-1)
4. set min=i and set j=i+1
5. repeat the steps 6 while(j<n)
6. if(a[j]<a[min]) set min=j
7. temp=a[i]

a[i]=a[min]
a[min]=temp

8. stop

BUBBLE SORT

If n elements are given in memory then for sorting we do following steps:

1. First compare the 1st and 2nd element of array if 1st <2nd then compare the 2nd with 3rd.
2. If 2nd >3rd Then interchange the value of 2nd and 3rd.
3. Now compare the value of 3rd (which has the value of 2nd) with 4th.
4. Similarly compare until the (n-1)th element is compared with nth element.
5. Now the highest value element is reached at the nth place.
6. Now elements will be compared until n-1 elements.

Example:

42	23	74	11	65	58	94	36	99	87
23	42	11	65	58	74	36	94	87	**99--**pass1
23	11	42	58	65	36	74	87	**94**	**99--p**ass2
11	23	42	58	36	65	74	**87**	**94**	**99**—pass3
11	23	42	36	58	65	**74**	**87**	**94**	**99**—pass4
11	23	36	42	58	**65**	**74**	**87**	**94**	**99**—pass5
11	23	36	42	**58**	**65**	**74**	**87**	**94**	**99**—pass6
11	23	36	42	**58**	**65**	**74**	**87**	**94**	**99**—pass7
11	23	36	42	**58**	**65**	**74**	**87**	**94**	**99**—pass8
11	23	36	42	**58**	**65**	**74**	**87**	**94**	**99**—pass9

Algorithm

Bubblesort(int a[ ],int n)

1. start
2. set i=0

3. repeat steps 4,5 and 8 while(i<n-1)
4. j=i+1
5. repeat steps 6,7 while(j<n)
6. if a[j]<a[i]
   temp=a[i] //exchange a[i] and a[j]
   a[i]=a[j]
   a[j]=temp
7. set j=j+1
8. set i=i+1
9. stop

Disadvantage of bubble sort:

It is acceptable for sorting a table which contains a small number of records, but it becomes difficult for large sized tables.

Divide-and-Conquer Algorithms

The ***divide and conquer*** strategy solves a problem by:

1. Breaking into ***sub problems*** that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub-problems.
3. Appropriately combining their answers.

Two types of sorting algorithms which are based on this divide and conquer algorithm:

1. **Quick sort:** Quick sort also uses few comparisons (somewhat more than the other two). Like heap sort it can sort "in place" by moving data in an array.
2. **Merge sort:** Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

MERGE SORT

If there are two sorted lists of array then process of combining these sorted lists into sorted order is called merging.

Take one element of each array, compare them and then take the smaller one in third array. Repeat this process until the elements of any array are finished. Then take the remaining elements of unfinished array in third array.

To sort $A[p \,..\, r]$:

1. **Divide Step**

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p \,..\, r]$ into two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$, each containing about half of the elements of $A[p \,..\, r]$. That is, q is the halfway point of $A[p \,..\, r]$.

2. **Conquer Step**

Conquer by recursively sorting the two subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$.

3. **Combine Step**

Combine the elements back in $A[p \,..\, r]$ by merging the two sorted subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Algorithm: Merge Sort

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

```
1.   IF p < r                                  // Check for base case
2.      THEN q = FLOOR[(p + r)/2]               // Divide step
3.           MERGE (A, p, q)                    // Conquer step.
4.           MERGE (A, q + 1, r)                // Conquer step.
5.           MERGE (A, p, q, r)                 // Conquer step.
```

The **pseudocode** of the MERGE procedure is as follow:

MERGE (A, p, q, r)

```
1.   n1 ← q − p + 1
2.   n2 ← r − q
3.   Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
4.   FOR i ← 1 TO n1
5.        DO L[i] ← A[p + i − 1]
6.   FOR j ← 1 TO n2
7.        DO R[j] ← A[q + j ]
8.   L[n1 + 1] ← ∞
```

9. $R[n_2 + 1] \leftarrow \infty$
10. $i \leftarrow 1$
11. $j \leftarrow 1$
12. **FOR** $k \leftarrow p$ **TO** r
13. **DO IF** $L[i] \leq R[j]$
14. **THEN** $A[k] \leftarrow L[i]$
15. $i \leftarrow i + 1$
16. **ELSE** A[k] ← R[j]
17. $j \leftarrow j + 1$

Example:

Sorted List1				Sorted List2				List3
1	13	24	26	**2**	15	27	38	
1	**13**	24	26	**2**	15	27	38	1
1	**13**	24	26	2	**15**	27	38	1 2
1	13	**24**	26	2	**15**	27	38	1 2 13
1	13	**24**	26	2	15	**27**	38	1 2 13 15
1	13	24	**26**	2	15	**27**	38	1 2 13 15 24
1	13	24	26	2	15	**27**	38	1 2 13 15 24 26
1	13	24	26	2	15	27	**38**	1 2 13 15 24 26 27 38

QUICK SORT

The basic version of quick sort algorithm was invented by C. A. R. Hoare in 1960 and formally introduced quick sort in 1962. It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is

not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution.

The idea behind this sorting is that sorting is much easier in two short lists rather than one long list. Divide and conquer means divide the big problem into two small problems and then those two small problems into two small ones and so on. As example, we hve a list of 100 names and we want to list them alphabetically then we will make two lists for names. A-L and M-Z from original list. Then we will divide list A-L into A-F and G-L and so on until the list could be easily sorted. Similar policy we will adopt for the list M-Z.

Quicksort( )

if (p<q)

1. j=partition(a,p,q+1);
2. quicksort(p, j-1);
3. quicksort(j+1, q);

Quick sort works by partitioning a given array a[*p* . . *q*] into two non-empty sub array a[*p* . . *j-1*] and a[*j*+1 . . *q*] such that every key in *a*[*p* . . *j-1*] is less than or equal to every key in *A*[*j*+1. . *q*]. Then the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index j is computed as a part of the partitioning procedure.

Note that to sort entire array, the initial call Quick Sort (*a*, 1, length[*a*])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partitioning the Array

Partitioning procedure rearranges the subarrays in-place.

partition(a, m, n)

1. int pivot = a[m];
2. int i=m;
3. int j=n;
4. do repeat steps 5-7 while (i < j)
5. while (a[i] < pivot) increment i;
6. while (a[j] > pivot) decrement j;
7. if (i < j) interchange a[i] and a[j]
8. a[m]=a[j];
9. a[j]=pivot;
10. return j;

Example:
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

j = 1 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping

arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Searching techniques

Linear Search/Sequential Search-sequential search on unsorted data or on sorted data

In this Searching Technique, Search element is compared Sequentially with each element in an Array and process of comparison is stopped when element is matched with array element. If not, element is not found.

Algorithm

Linear(int x[ ],int n)

1. start
2. key=element to be searched, set i=0
3. repeat steps 4,5 while(i<n)
4. if(x[i]==key), key element is found and goto 7
5. i=i+1
6. element not found
7. stop

Binary search/Divide and conquer scheme-Search on sorted data

- Here elements must be in Ascending/Descending order.

Algorithm

Binarysearch(int x[ ],int n)

1. start
2. Set low=0,high=n-1,key is the element to be searched

3. repeat steps 4,5,6 while(low<=high)
4. mid=(low+high)/2
5. if(x[mid]==key),element is found goto 9
6. if(x[mid]<key),low=mid+1,goto step 3
else if(x[mid]>key),high=mid-1,goto step 3 8. else element not found
9. stop