Sorting (DSA)

Sorting in computer terms means arranging elements in a specific order - ascending or descending order.

Bubble sort

The basic idea of bubble sort is to compare two adjoining values and exchange them if they are not in proper order.

eg. 33, 44, 22, 11, 66, 55

Pass 1

33 ↕	33	33	33	33	33
44 ↕	44 ↕ swap	22	22	22	22
22	22	44 ↕ swap	11	11	11
11	11	11	44 ↕	44	44
66	66	66	66	66 ↕ swap	55
55	55	55	55	55	66

Pass 2

33 ↕ swap	22	22	22	22
22	33 ↕	11	11	11
11	11	33 ↕	33	33
44	44	44 ↕	44 ↕	44
55	55	55	55	55
66	66	66	66	66

Pass 3 ... Continue...

Implementation

```
for (i = 1; i<n; i++)
{ for (j = n-1; j >= i; j--)
  { if (a[j] < a[j-1])
      swap a[j] and a[j-1].
```

Algorithm

1. Repeat step 2 and 3 for $i=1$ to n
2. Set $j=1$.
3. Repeat for $j<=n$
   A. if $a[j] <> a[j+1]$
      Then interchange $a[j]$ and $a[j+1]$
   B. Set $j=j+1$
4. Exit.

Program pseduo code

```
for (i=0 ; i<n; i++)
{
  for (j=0; j<n-i-1 ; j++)
  {
    if (arr[j] > arr[j+1])
    {
      temp = arr[j];
      arr[j] = arr[j+1];
      arr[j+1] = temp;
    }
  }
}
```

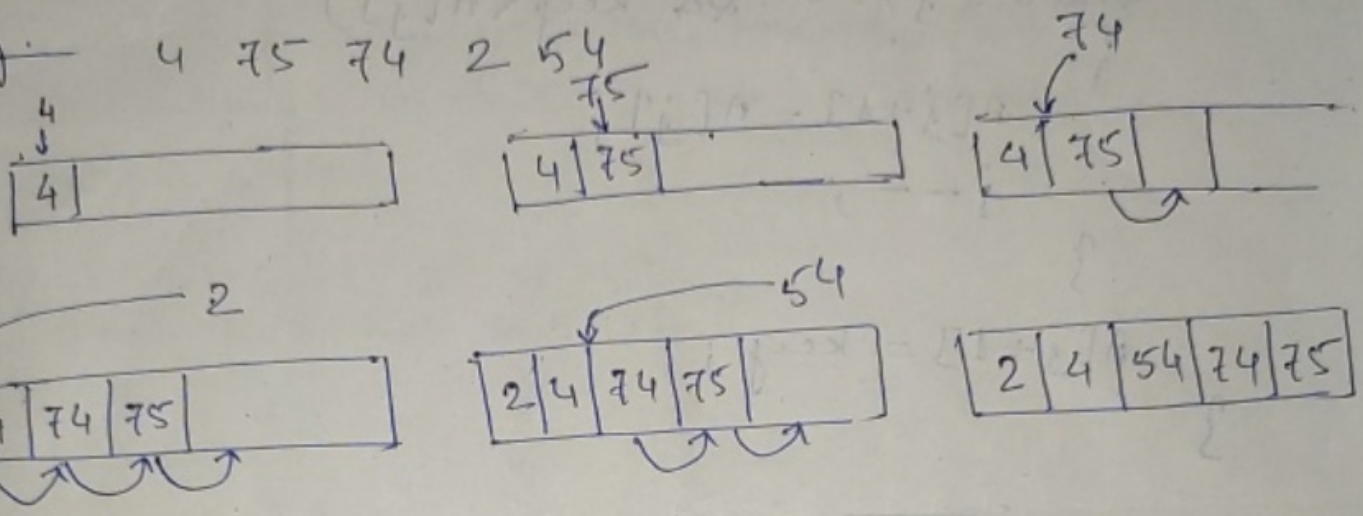
Time Complexity.

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisions is $(n-1)*(n-1) = n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

Insertion Sort

Insertion sort is a sorting algorithm that builds a sorted list one element at a time from the ~~list~~ unsorted list by inserting the element at its correct position in sorted list.

eg. 4 75 74 2 54



Algorithm. (Sort the array A with N elements)

1. Set A[0] = $-\infty$
2. Repeat step 3 to 5 for k = 2 to n.
3. Set key = A[k]. And j = k-1
4. Repeat while key < A[j]
   A. Set A[j+1] = A[j]
   B. j = j-1.
5. Set A[j+1] = key

Code

```
for (i = 1 ; i < n ; i++)
{
    key = A[i] ;
    j = i - 1 ;
    while (j >= 0 . && key < A[j])
    {
        A[j+1] = A[j] ;
        j -- ;
    }
    A[j+1] = key ;
}
```

Complexity

The no. f(n) of comparisions in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K-1 of comparisons. Hence

$$F(n) = 1 + 2 + 3 + \dots + (n-1) = n(n-1)/2 = O(n^2)$$

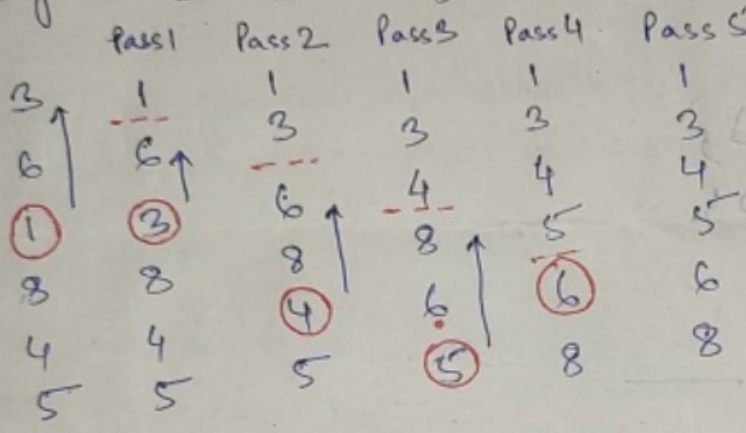
On the average, there will be approximately (K-1)/2 comparison in the inner loop. Accordingly, for the average case,

$$F(n) = O(n^2).$$

Selection Sort

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

eg. 3 6 1 8 4 5



Algorithm:

1. Repeat For J = 0 to N-1
2. Set MIN = j
3. Repeat FOR K = J+1 to N
4. If (A[K] < A[MIN]) Then
5. Set MIN = K
   [end of If]
   [end of step 3 for loop]
6. Interchange A[j] and A[MIN]
   [End of step 1 for loop]
7. Exit.

Code

```
for(i=0; i<n-1; i++)
{
    min = i;
    for(j = i+1; j<n; j++)
    {
        if(a[j] < a[min])
        {
            min = j;
        }
    }
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
}
```

Complexity

The no. of comparison in the selection sort algorithm is independent of the original order of the element. That is there are $n-1$ comparison during Pass 1 to find the smallest element, there are $n-2$ comparisons during Pass 2 to find the second smallest element, and so on.

$\therefore F(n) = (n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$

$= O(n^2)$.

Quick Sort

The quick sort was invented by Prof. C.A.R Hoare in the early 1960's. It was one of the most efficient sorting algorithms. Quick sort is not stable search, but it is very less additional space. It is based on the rule of Divide and Conquer. This algorithm divides the list into three main parts

1. Pivot element
2. Element less than pivot element
3. Element greater than the pivot element.

The key to the algorithm is the Partition procedure which rearranges the subarray A[p r] in place.

```
Partition (A, p, r)
1. x = A[r]
2. i = p-1
3. for j ← p to r-1.
4.     if A[j] ≤ x
5.         then i = i+1,
6.              exchange A[i] with A[j]
7. Exchange A[i+1] with A[r]
8. return i+1.
```

eg

13 19 9 5 12 8 7 4 21 2 6 11

Quicksort (A, p, r)

1. if $p < r$
2. then q = Partition (A, p, r)
3. Quicksort (A, p, q-1)
4. Quicksort (A, q+1, r)

Complexity

The worst case occurs when the list is sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have n-1 elements. Accordingly the second element require n-1 comparisons to recognize that it remains in the second position ~~so~~ and so on.

$$f(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$
$$= n(n+1)/2 = O(n^2)$$

Average case $O(n \log n)$

Best Case $O(n \log n)$

Mergesort

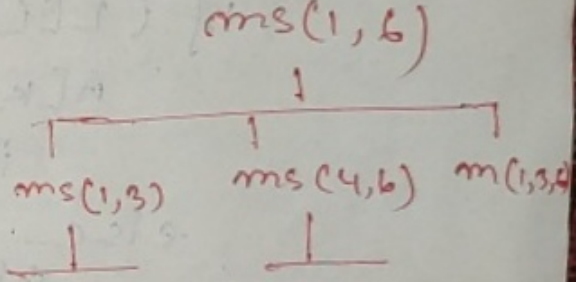
The algorithm based on splitting the array of item into two sub-array. This simply splits the array at each stage into ~~the~~ its first and last half, without any reordering of the items in it. However, that will obviously not result in a set of sorted sub-arrays that we can just append to each other at the end. So merge sort needs another procedure 'merge' that merges two sorted sub-arrays into another sorted array.

Algorithm

```
mergesort (A, p, r)
{
    if (p < r)
    {
        q = (p + r)/2;
        mergesort (A, p, q)
        mergesort (A, q+1, r)
        merge (A, p, q, r)
    }
}
```

eg.

9 6 5 0 8 2



```
merge(A, p, q, r)
{
    n1 = q - p + 1
    n2 = r - q
    Let L[1 to n1+1] and R[1 to n2+1] be
    new array.
    for (i = 1 to n1)
        L[i] = A[p + i - 1]
    for (j = 1 to n2)
        R[j] = A[q + j]
    L[n1+1] = ∞
    R[n2+1] = ∞
    i = 1 , j = 1 .
    for (k = p to r)
        if (L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
}
```

p		q	q+1		r
1	5	7	8	2	4	6	9

L = | 1 | 5 | 7 | 8 | ∞ |

R = | 2 | 4 | 6 | 9 | ∞ |

1	2	4	5	6	7	8	9

Complexity.

The total number of comparisons needed at each recursion level of mergesort is the number of items needing merging which is $O(n)$, and the number of recursions needed to get to the single item level is $O(\log_2 n)$; so the total number of comparisons and its time complexity are $O(n \log_2 n)$. This holds for the worst case as well as the average case.

Radix sort

The idea is to consider the key one character at a time and to divide the entries, not into two sub lists, but into as many sub lists as there are possibilities for the given character from the key. If our keys, for example, are words or other alphabetic strings, then we divide the list into 26 sub lists at each stage. That is, we set up a table of 26 lists and distribute the entries into the lists according to one of the characters in the key.

eg — 82 901 100 12 150 77 55 33

Once place digit →

0	1	2	3	4	5	6	7	8	9
100	901	82	33		55		77		
150		12							

100 150 901 82 12 33 55 77

Tens place digit

0	1	2	3	4	5	6	7	8	9
100	12		33		150		77	82	
901					55				

100 901 12 33 150 55 77 82

Hundres placed digit

0	1	2	3	4	5	6	7	8	9
12	100								901
33	150								
55									
77									
82									

12 33 55 77 82 100 150 901

Merge Sort

```
#include <stdio.h>
void mergesort (int a[], int i, int j);
void merge (int a[], int i1, int j1, int i2, int j2);
int main()
{
    int a[30], n, i;
    pf ("Enter no of elements:");
    sf ("%d", &n);
    pf ("Enter array elements:");
    for (i=0; i<n; i++)
        sf ("%d", &a[i]);
    mergesort ("In sorted array is:", a, 0, n-1);
    for (i=0; i<n; i++)
        pf ("%d", a[i]);
    return 0;
}
void mergesort (int a[], int i, int j)
{
    int mid;
    if (i<j)
    {
```

```
mid = (i+j)/2;
mergesort(a, i, mid);
mergesort(a, mid+1, j);
merge(a, i, mid, mid+1, j)
}
}
```

p mid mid+1 r

```
void merge(int a[], int i1, int j1, int i2, int j2)
{
  int temp[150]   //array used for merging.
  int i, j, k;
  i = i1;   //beginning of the 1st list
  j = i2;   //  "   "   "  2nd  "
  k = 0;
  while(i<=j1 && j<=j2)
  {
    if(a[i] < a[j])
      temp[k++] = a[i++];
    else
      temp[k++] = a[j++];
  }
  while(i<=j1)   //copy remaining elements of the list 1.
    temp[k++] = a[i++];
  while(j<=j2)   //  "   "   "   "   "  list 2
    temp[k++] = a[j++];
```

```
// transfer elements from temp back to a[]
for(i=i1 , j=0; i<=j2; i++, j++)
    a[i] = temp[j];
}.
```