Experiment No. 1

Problem Definition: Implementation of Selection sort and Insertion sort.

Theory:

1. **Selection Sort :**
   - **Idea:**
     1. Find the smallest element in the array
     2. Exchange it with the element in the first position
     3. Find the second smallest element and exchange it with the element in the second position
     4. Continue until the array is sorted

   - Disadvantage:
     1. Running time depends only slightly on the amount of order in the file.

   - **Algorithm:**

   Alg.: SELECTION-SORT*(A)*

   n ← length[A]

   for j ← 1 to n - 1

   do smallest ← j

   for i ← j + 1 to n

   do if A[i] < A[smallest]

   then smallest ← i

   exchange A[j] ↔ A[smallest]

- **Example :**

8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8
1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9

- **Complexity analysis:**

Alg.: SELECTION-SORT(*A*)	cost	times
$n \leftarrow$ length[A]	c_1	1
for $j \leftarrow 1$ to $n - 1$	c_2	n
do smallest $\leftarrow j$	c_3	$n-1$
for $i \leftarrow j + 1$ to n	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
do if A[i] < A[smallest]	c_5	$\sum_{j=1}^{n-1}(n-j)$
then smallest $\leftarrow i$	c_6	$\sum_{j=1}^{n-1}(n-j)$
exchange A[j] $\leftrightarrow$ A[smallest]	c_7	$n-1$

$\approx n^2/2$ comparisons

$\approx n$ exchanges

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

2. Insertion Sort :

- **Idea:** like sorting a hand of playing cards

    i. Start with an empty left hand and the cards facing down on the table.

    ii. Remove one card at a time from the table, and insert it into the correct position in the left hand

    1. compare it with each of the cards already in the hand, from right to left

    iii. The cards held in the left hand are sorted

    1. These cards were originally the top cards of the pile on the table.

3. Algorithm :

Alg.: INSERTION-SORT *(A)*

for j ← 2 **to** n

do key ← A[j]

Insert A[j] into the sorted sequence A[1 . . j -1]

i ← j - 1

while i > 0 and A[i] > key

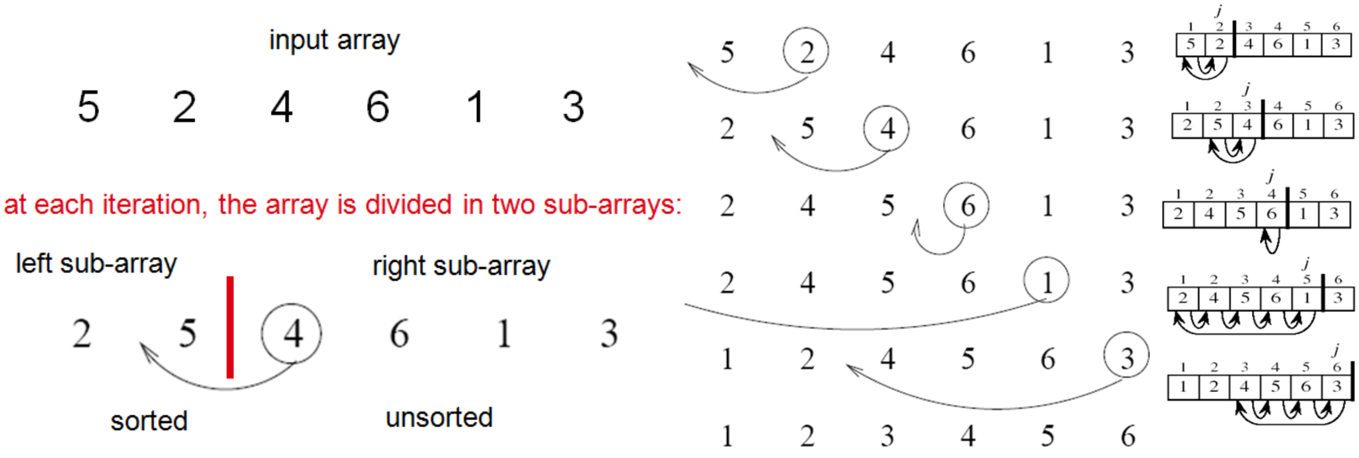
do A[i + 1] ← A[i]

i ← i – 1

A[i + 1] ← key

Insertion sort – sorts the elements in place

- **Example:**



- **Complexity Analysis :**

INSERTION-SORT*(A)*		cost	times
for j ← 2 **to** n		c_1	n
do key ← A[j]		c_2	n-1
Insert A[j] into the sorted sequence A[1 . . j -1]		0	n-1
i ← j - 1	≈ $n^2/2$ comparisons	c_4	n-1
while i > 0 and A[i] > key		c_5	$\sum_{j=2}^{n} t_j$
do A[i + 1] ← A[i]		c_6	$\sum_{j=2}^{n} (t_j - 1)$
i ← i – 1	≈ $n^2/2$ exchanges	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[i + 1] ← key		c_8	n-1

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

- Best Case analysis :
  - The array is already sorted **“while** i > 0 and A[i] > key”
    - A[i] ≤ key upon the first time the **while** loop test is run (when $i = j - 1$)
    - $t_j = 1$
  - $T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$

    $= (c_1 + c_2 + c_4 + c_5 + c_8)n + (c_2 + c_4 + c_5 + c_8)$

    $= an + b = \Theta(n)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

- Worst Case Analysis :

- The array is in reverse sorted order **“while** i > 0 and A[i] > key”
  - Always A[i] > **key** in **while** loop test
  - Have to compare **key** with all elements to the left of the **j**-th position ⇒ compare with **j-1** elements ⇒ $t_j = j$

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \;=>\; \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \;=>\; \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1)$$

a quadratic function of n

- $T(n) = \Theta(n^2)$ order of growth in n^2

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j - 1) + c_7\sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Experiment No.2

Problem Definition: Implementation of Quick sort and Merge sort.

Theory:

1. **Quick Sort:**
    - Divide-and-conquer algorithm:
    - *Divide*: *A* [*p*…*r*] is partitioned (rearranged) into two nonempty sub arrays *A* [*p*…*q*-1] and *A*[*q*+1…*r*] s.t. each element of *A*[*p*…*q*-1] is less than or equal to each element of *A*[*q*+1…*r*]. Index *q* is computed here, called **pivot**.
    - *Conquer*: two sub arrays are sorted by recursive calls to quick sort.
    - *Combine*: unlike merge sort, no work needed since the sub arrays are sorted in place already.

➢ **Algorithm:**

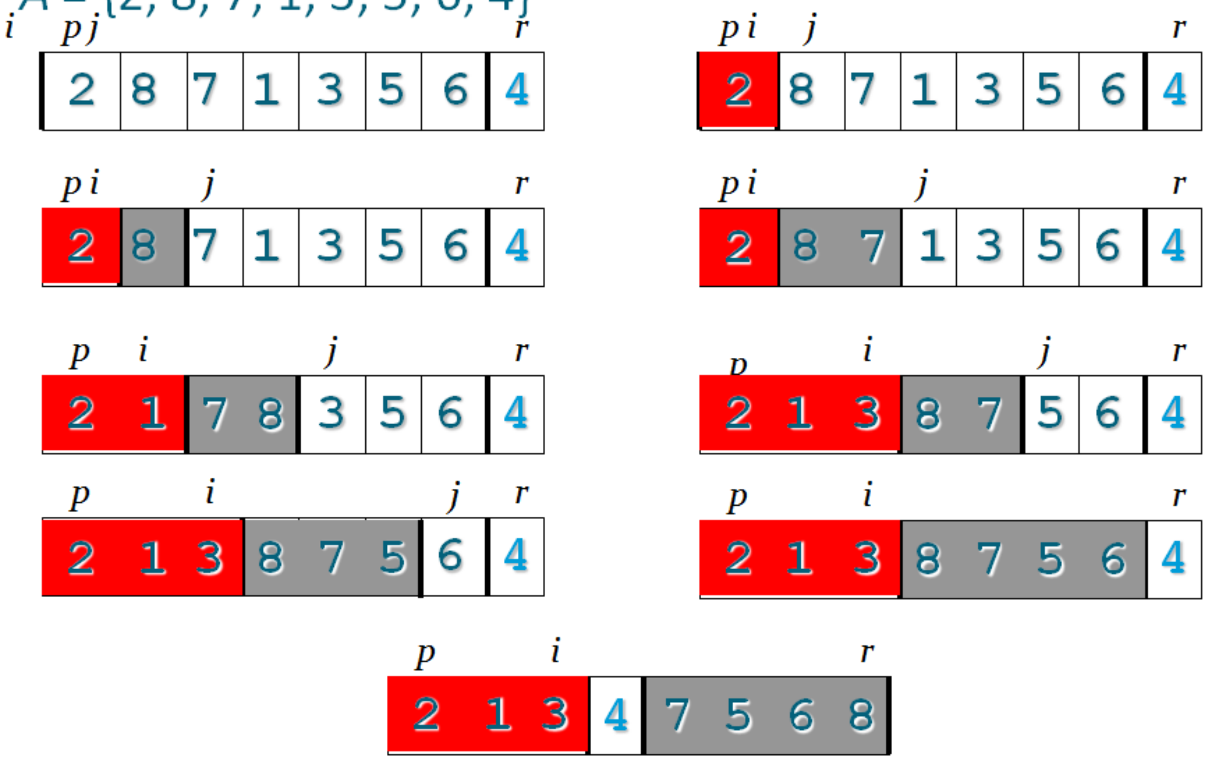
- Pseudo-Code

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

- **Example:**

$A = \{2, 8, 7, 1, 3, 5, 6, 4\}$



- **Complexity Analysis:**

- We are dividing the array into two parts based on pivot.
- T(0)=T(1)=c
- T(n) =T(left)+T(right)+n

Time required sorting left sub-array

Time required sorting right sub-array

Time required for partitioning n elements

- **Worst-Case** Performance (unbalanced):

  $T(n) = T(1) + T(n\text{-}1) + \Theta(n)$

  partitioning takes $\Theta(n)$

  $= [2 + 3 + 4 + \ldots + n\text{-}1 + n\ ] + n$

  $= [\sum_{k = 2 \text{ to } n} k\ ] + n = \boldsymbol{\Theta(n^2)}$

  This occurs when

  ✓ the input is **completely sorted**

  or when

  ✓ the pivot is always the **smallest** (**largest**) element

- **Best Case:**

  When the partitioning procedure produces two regions of size ***n*/2**, we get the **balanced** partition with **best case** performance:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + n & \text{if } n \geq 2 \end{cases}$$

- We iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/2^2)) + (n/2)) + n \\ &= 2^2 T(n/2^2) + 2n \\ &= 2^3 T(n/2^3) + 3n \\ &= 2^4 T(n/2^4) + 4n \\ &= \ldots \\ &= 2^i T(n/2^i) + in \end{aligned}$$

- Note that base, T(n)=c, case occurs when 2^i=n. That is, i = log n.

- So,

$$T(n) = cn + n \log n$$

- Thus, T(n) is **O(n log n).**

2. Merge Sort :

Sort a sequence of *n* elements into non-decreasing order.

- *Divide*: Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- *Conquer:* Sort the two subsequences recursively using merge sort.
- *Combine*: Merge the two sorted subsequences to produce the sorted answer.

➢ **Algorithm:**

- Pseudo-Code

Merge (*A, p, q, r*)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

for $i \leftarrow 1$ **to** n_1

  do $L[i] \leftarrow A[p + i - 1]$

for $j \leftarrow 1$ **to** n_2

  do $R[j] \leftarrow A[q + j]$

$L[n_1+1] \leftarrow \infty$

$R[n_2+1] \leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow p$ **to** r

do if $L[i] \le R[j]$

then $A[k] \leftarrow L[i]$

$i \leftarrow i + 1$

else $A[k] \leftarrow R[j]$

$j \leftarrow j + 1$

***MergeSort* (*A*, *p*, *r*)   // sort *A*[*p*..*r*] by divide & conquer**

if $p < r$

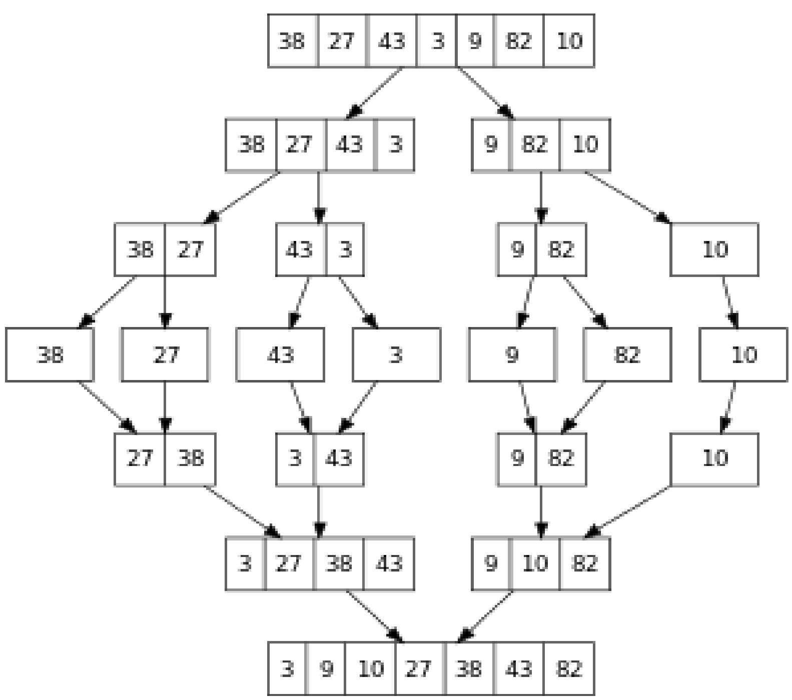
then $q \leftarrow \lfloor (p+r)/2 \rfloor$

MergeSort (*A*, *p*, *q*)

MergeSort (*A*, *q*+1, *r*)

Merge (*A*, *p*, *q*, *r*) // merges *A*[*p*..*q*] with *A*[*q*+*1*..*r*]

- **Example :**



- **Complexity Analysis:**
  - The conquer step of merge-sort consists of merging two sorted sequences, each with *n*/2 elements and takes at most *cn* steps, for some constant *c*.
  - Likewise, the basis case ($n < 2$) will take at *c* most steps.
  - Therefore, if we let *T*(*n*) denote the running time of merge-sort:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{if } n \geq 2 \end{cases}$$

  - We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.
    - That is, a solution that has *T* (*n*) only on the left-hand side.

  - In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/2^2)) + c(n/2)) + cn \\ &= 2^2 T(n/2^2) + 2cn \\ &= 2^3 T(n/2^3) + 3cn \\ &= 2^4 T(n/2^4) + 4cn \\ &= \ldots \\ &= 2^i T(n/2^i) + icn \end{aligned}$$

  - Note that base, T(n)=c, case occurs when 2^i=n. That is, i = log n.
  - So,

$$T(n) = cn + cn \log n$$

  - Thus, T(n) is **O(n log n).**