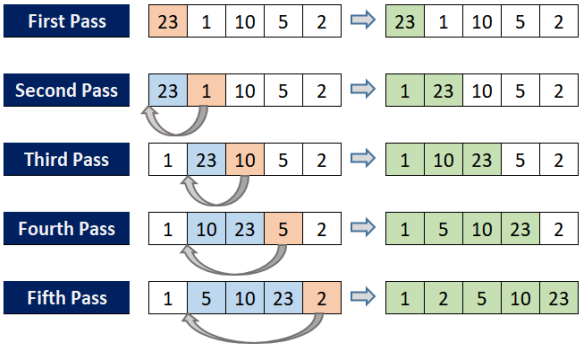
Name:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Analysis of Algorithm Laboratory
Course Code:	DJ19CEL404
Experiment No.:	01

AIM: **IMPLEMENT AND ANALYSE INSERTION AND SELECTION SORT**

THEORY:



- **Insertion sort**
  - Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
  - The array is virtually split into a sorted and an unsorted part.
  - Values from the unsorted part are picked and placed at the correct position in the sorted part.
  - Basically, Insertion sort is efficient for small data value.
  - Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.
  - Pseudocode:

```
procedure insertionSort(A: list of sortable items)
   n = length(A)
   for i = 1 to n - 1 do
       j = i
       while j > 0 and A[j-1] > A[j] do
           swap(A[j], A[j-1])
           j = j - 1
       end while
   end for
end procedure
```

  - Time Complexity:
    - The **worst-case (and average-case) complexity of the insertion sort algorithm is O(n^2).** Meaning that, in the worst case, the time taken to sort a list is proportional to the square of the number of elements in the list.
    - **The best-case time complexity of insertion sort algorithm is O(n)** time complexity. Meaning that the time taken to sort a list is proportional to the number of elements in the list; this is the case when the list is already in the correct order. There's only one iteration in this case since the inner loop operation is trivial when the list is already in order.
  - Space Complexity
    - The insertion sort encompasses a **space complexity of O(1)** due to the usage of an extra variable key.

Code:

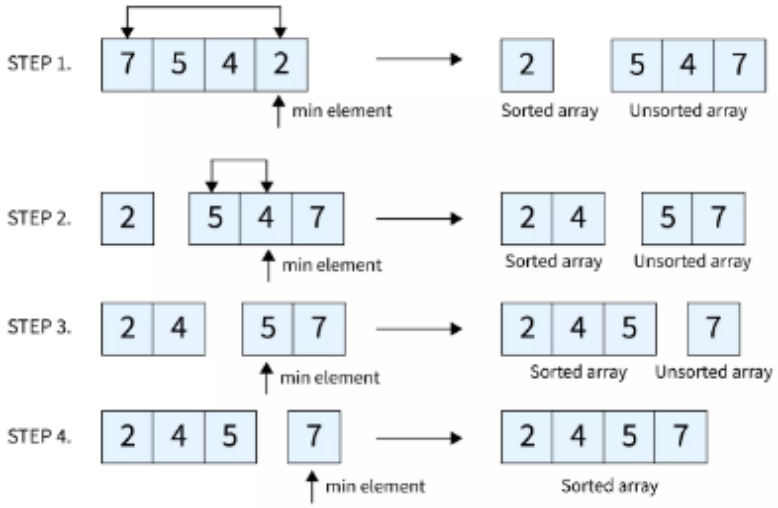
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int arr[10000];

void main()
{
   clock_t start, stop;
   clock_t start_b, stop_b;
   clock_t start_w, stop_w;

   for (int i = 0; i < 10000; i++)
   {
      arr[i] = rand();
   }

   int key, j, n = 10000;
   start = clock();

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

   stop = clock();
   float res = stop - start;
   printf("\nAvg case CPU time =%f units", res);

   start_b = clock();

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

```
    {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
  }
  stop_b = clock();
  float x = stop_b - start_b;
  printf("\nBest case CPU time =%f units", x);

  start_w = clock();

  for (int i = 1; i < n; i++)
  {
    key = arr[i];
    j = i - 1;
    while (j >= 0 && arr[j] < key)
    {
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = key;
  }
  stop_w = clock();
  x = stop_w - start_w;
  printf("\nworst case CPU time =%f units", x);
}
```

Output:

```
swljx2.5ad' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'

Avg case CPU time =158.000000 units
Best case CPU time =0.000000 units
worst case CPU time =330.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

Selection Sort



- Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted portion.
- This process is repeated for the remaining unsorted portion of the list until the entire list is sorted.
- Pseudocode:

```
function selection sort
   for i = 1 to size - 1
      minimum = i // set current element as minimum
      for j = i+1 to n // check the element to be minimum
         if array[j] < array[minimum] then
            minimum = j;
         end if
      end for
      if indexofMinimum != i then //swap the minimum element with the current
element
         swap array[minimum] and array[i]
      end if
   end for
end function
```

- Time Complexity:
  - Worst Case Complexity: O(n2)
    - ✓ If we want to sort in ascending order and the array is in descending order then, the worst case occurs.
  - Best Case Complexity: O(n2)
    - ✓ It occurs when the array is already sorted
  - Average Case Complexity: O(n2)
    - ✓ It occurs when the elements of the array are in jumbled order (neither ascending nor descending).
- Space Complexity:
  - Space complexity is O(1) because an extra variable is used.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    clock_t start, stop;
    clock_t start_b, stop_b;
    clock_t start_w, stop_w;

    int n = 10000, i, j, position, swap;
    int a[n];
    for (i = 0; i < n; i++)
    {
        a[i] = rand();
    }

    // sorts a jumbled array to give average case time complexity
    start = clock();

    for (i = 0; i < n - 1; i++)
    {
        position = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[position] > a[j])
                position = j;
        }
        if (position != i)
        {
            swap = a[i];
            a[i] = a[position];
            a[position] = swap;
        }
    }
    stop = clock();
    float res = stop - start;
    printf("\nAverage case CPU time =%f units", res);

    // sorts sorted array to give the best case time complexity
    start_b = clock();
    for (i = 0; i < n - 1; i++)
    {
        position = i;
```

```
        for (j = i + 1; j < n; j++)
        {
            if (a[position] > a[j])
                position = j;
        }
        if (position != i)
        {
            swap = a[i];
            a[i] = a[position];
            a[position] = swap;
        }
    }
    stop_b = clock();
    float x = stop_b - start_b;
    printf("\nBest case CPU time =%f units", x);
    // sorts the array in descending order to give worst case time complexity
    start_w = clock();
    for (i = 0; i < n - 1; i++)
    {
        position = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[position] < a[j])
                position = j;
        }
        if (position != i)
        {
            swap = a[i];
            a[i] = a[position];
            a[position] = swap;
        }
    }
    stop_w = clock();
    x = stop_w - start_w;
    printf("\nworst case CPU time =%f units", x);
    return 0;
}
```

OUTPUT:

```
n\gdb.exe' '--interpreter=mi'

Average case CPU time =239.000000 units
Best case CPU time =315.000000 units
worst case CPU time =389.000000 units
PS C:\Users\Jadhav\Desktop\BTech\4th sem\AOA\Prac\Code>
```

CONCLUSION:

Sr. No.	Insertion Sort	Selection Sort
1.	The number of comparison operations performed in this sorting algorithm is less than the swapping performed.	The number of comparison operations performed in this sorting algorithm is more than the swapping performed.
2.	It is more efficient than the Selection sort.	It is less efficient than the Insertion sort.
3.	The insertion sort is used when: • The array is has a small number of elements • There are only a few elements left to be sorted	The selection sort is used when • A small list is to be sorted • The cost of swapping does not matter • Checking of all the elements is compulsory • Cost of writing to memory matters like in flash memory (number of Swaps is O(n) as compared to O(n2) of bubble sort)
4.	The insertion sort is Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is **O(kn)** when each element in the input is no more than **k** places away from its sorted position	Selection sort is an in-place comparison sorting algorithm