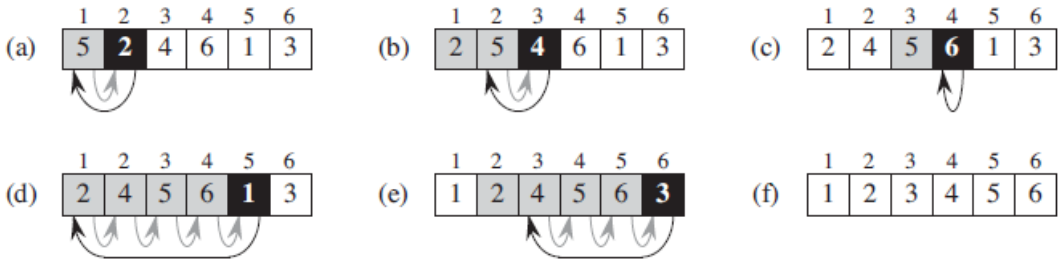
AA LAB ASSIGNMENT – 01 | INSERTION SORT

TITLE: Insertion Sort

MECHANISM

Insertion sort is a simple sorting algorithm that builds the final sorted list one item at a time. The basic idea of insertion sort is to start with an empty "sorted" list, and repeatedly insert new elements into the correct position within the sorted list. The process can be visualized as building a deck of cards, where each card represents an element in the list.

Example: For a sequence [5, 2, 4, 6, 1, 3], here is what the algorithm looks like with each pass –



ALGORITHM / PSEUDOCODE

Input: A sequence of n numbers (a_1, a_2, a_3, ……. a_n).
Output: A permutation (reordering) (a'_1, a'_2, a'_3, ……. a'_n) of the input sequence such that $a'_1 <= a'_2 <= a'_3 <= \ldots\ldots <= a'_n$

```
1.  for j=2 to A. length
2.      key =A[j]
3.      // Insert A[j] into the sorted sequence A[1…j-1].
4.      i = j -1
5.      while i > 0 and A[j] > key
6.              A[i+1] = A[i]
7.              i = i – 1
8.      A[i+1] = key
```

IMPLEMENTATION (ITERATIVE AND RECURSIVE)

```
#include<iostream>
#include<conio.h>

using namespace std;
void display ( int a[], int n) {
    for (int i = 0; i < n; i++)
    {
        cout<< a[i]<< " ";
    }
    cout << endl;
}

void insertion_sort(int a[], int n) {

    int key = a[0];
    for ( int i = 1; i < n; ++i )
    {
        display(a, n);
        int j = i-1;

        int key = a[i];
        while ( j >= 0 && a[j] > key ) {
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
            j-=1;
        }
    }
}

void insertion_sort_rec(int arr[], int n)
{
    if (n <= 1)
        return;

    insertion_sort_rec( arr, n-1 );

    display(arr, n);

    int last = arr[n-1];
    int j = n-2;

    while (j >= 0 && arr[j] > last)
    {
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = last;
}
```

```cpp
int main() {

    int a[] = {9, 8, 7, 6, 5, 4};
    int n = 6;
    int ch;

    cout << "Array: ";
    display(a, n);

    cout << "\n1. Recursive Insertion Sort.";
    cout << "\n2. Iterattive Insertion Sort.";
    cout << "\n\nChose an option: ";
    cin >> ch;

    switch (ch) {

        case 1: {

            cout << "\nPasses:\n";
            insertion_sort_rec(a, n);
            cout << "\nFinal: ";
            display(a, n);
            break;
        }
        case 2: {
            cout << "\nPasses:\n";
            insertion_sort(a, n);
            cout << "\nFinal: ";
            display(a, n);
            break;
        }
        default: {
            break;
        }
    }
    return 0;
}
```

```
PS F:\#3 SIT\6. SEM 6\Advance Algo> cd
Array: 9 8 7 6 5 4

1. Recursive Insertion Sort.
2. Iterattive Insertion Sort.

Chose an option: 2

Passes:
9 8 7 6 5 4
8 9 7 6 5 4
7 8 9 6 5 4
6 7 8 9 5 4
5 6 7 8 9 4

Final: 4 5 6 7 8 9
PS F:\#3 SIT\6. SEM 6\Advance Algo>
```

```
PS F:\#3 SIT\6. SEM 6\Advance Algo> cd "f
Array: 9 8 7 6 5 4

1. Recursive Insertion Sort.
2. Iterattive Insertion Sort.

Chose an option: 1

Passes:
9 8
8 9 7
7 8 9 6
6 7 8 9 5
5 6 7 8 9 4

Final: 4 5 6 7 8 9
PS F:\#3 SIT\6. SEM 6\Advance Algo>
```

T(n) ANALYSIS WITH PROOF

Insertion-Sort(A)	*Cost*	*Times*
1. ***for*** *j=2* ***to*** *A. length*	*c1*	*n*
2. *key =A[j]*	*c2*	*n-1*
3. *// Insert A[j] into the sorted sequence A[1…j-1].*	*0*	*n-1*
4. *i = j -1*	*c4*	*n-1*
5. ***while*** *i > 0 and A[j] > key*	*c5*	$\Sigma_{j=2}^{n}$ *tj*
6. *A[i+1] = A[i]*	*c6*	$\Sigma_{j=2}^{n}$ *tj-1*
7. *i = i – 1*	*c7*	$\Sigma_{j=2}^{n}$ *tj-1*
8. *A[i+1] = key*	*c8*	*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 (ALREADY SORTED):

tj = 1

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

In the form of an + b, i.e., **linear time: Ω(n)**

WORST CASE (ALREADY SORTED):

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

In the form of $an^2 + bn + c$, i.e., **quadratic time: $O(n^2)$**

AVERAGE CASE:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

In the form of $an^2 + bn + c$, i.e., **quadratic time: Θ(n^2)**

SPACE COMPLEXITY ANALYSIS

In each iteration of the insertion sort algorithm, a constant amount of memory is used to store the current element, as well as temporary variables for swapping elements in the array. This memory usage does not increase with the size of the input array, n. Therefore, the total amount of memory used by the algorithm can be expressed as a constant value, c, multiplied by n, where c is a constant value representing the memory used for each iteration of the algorithm.
This can be written as: Memory usage = c * n
Since c is a constant value, the space complexity of insertion sort can be **expressed as O(1)** in terms of the size of the input.

ADVANTAGES / DISADVANTAGES

ADVANTAGE	DISADVANTAGES
Simple and easy to understand and implement.	Inefficient for large datasets
Efficient for small datasets	Not suitable for partially sorted arrays
Maintains the relative order of equal elements in the input array, making it a stable sorting algorithm	High overhead for swapping elements

REAL LIFE APPLICATIONS:

1. Sorting playing cards: Insertion sort can be used to sort a deck of playing cards by their rank or suit.
2. Organizing personal finances: Insertion sort can be used to sort financial transactions, such as bank statements or credit card statements, by date or amount.
3. Sorting small lists: Insertion sort can be used to sort small lists, such as lists of names, phone numbers, or addresses, in an address book or personal database.
4. Sorting library books: Insertion sort can be used to sort a library's collection of books by title, author, or publication date.
5. Sorting collections of small items: Insertion sort can be used to sort small collections, such as coins, stamps, or small toys, by value, date, or other criteria.

OPTIMIZATIONS AND ADVANCEMENTS:

1. Bidirectional Insertion Sort: A variation of the insertion sort algorithm that sorts the array from both the front and the back. This optimization can reduce the number of movements required to sort the array, leading to faster performance.
2. Shell Sort: A sorting algorithm that uses the principles of insertion sort, but with a more advanced approach to choosing the gap between elements to be compared. This optimization can result in a faster running time compared to traditional insertion sort.
3. Adaptive Insertion Sort: An optimization that uses the properties of partially sorted arrays to reduce the number of swaps required by the insertion sort algorithm.
4. In-Place Insertion Sort: An optimization that sorts the input array in place, without using any additional memory space to store intermediate results. This optimization can result in a more memory-efficient implementation of insertion sort.
5. Parallel Insertion Sort: An optimization that divides the input array into multiple parts and sorts each part in parallel, reducing the overall time required to sort the array.