TRIBHUVAN UNIVERSITY

Paschimanchal Campus

Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On:

Lab No: 7

Title:

Submitted By

Name: Anish Karki

Roll No: PAS080BCT007

Submitted To: HPB Sir

Date of submission: 07/08/2025

PRACTICAL-1
SELECTION SORT

THEORY

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on the sorting order) element from the unsorted portion of the array and swapping it with the first unsorted element. This process continues until the entire array is sorted.

The algorithm divides the list into two parts: a **sorted sublist** which is built up from left to right at the front (left) of the list, and the **unsorted sublist** which occupies the rest of the list. On each pass, the smallest element from the unsorted sublist is selected and moved to the sorted sublist.

Time Complexity:

- **Best Case:** $O(n^2)$
- **Average Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$

  Selection sort always makes the same number of comparisons, regardless of the input.

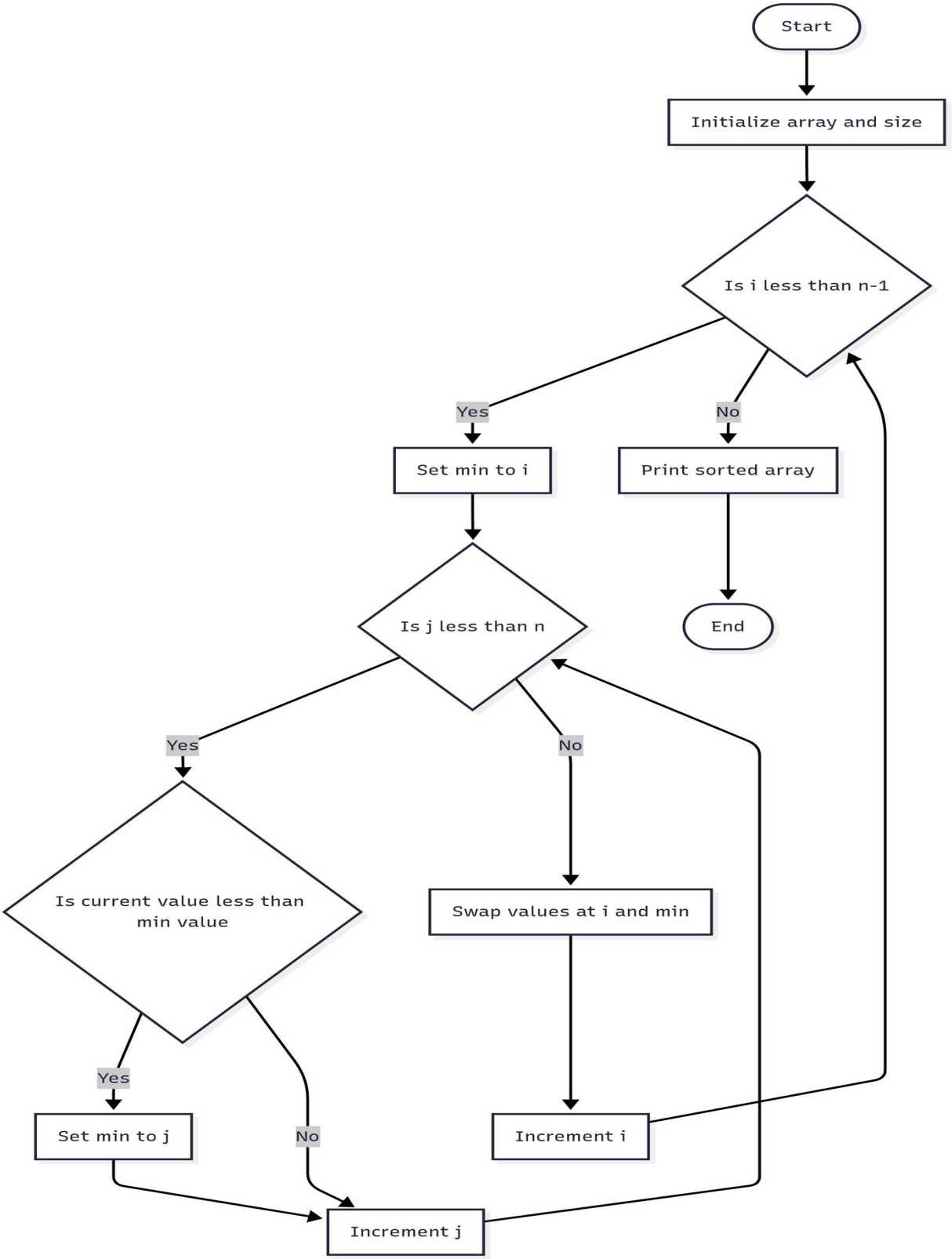
Characteristics:

- **Stable sort** (preserves the order of equal elements).
- Efficient for small datasets or nearly sorted data.
- Easy to implement.

ALGORITHM

1.Start
2.Input the array and its size.
3.For i from 0 to size - 1:
    1.Set min = i.
    2.For j from i to size :
        •If array[j] < array[min], set min = j.
    3.Swap array[i] with array[min].
4.Repeat steps 3.1 to 3.3 until the array is sorted.
5.Output the sorted array.
6.Stop

FLOWCHART


Start
Initialize array and size
Is i less than n-1
Yes
No
Set min to i
Print sorted array
End
Is j less than n
Yes
No
Is current value less than min value
Swap values at i and min
Yes
Set min to j
No
Increment i
Increment j

CODE:

```
#include <iostream>
using namespace std;

void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void selectionSort(int array[], int size) {
  for (int i = 0; i < size - 1; i++) {
    int min = i;
    for (int j = i; j < size; j++) {
      if (array[j] < array[min]) {
        min = j;
      }
    }
    swap(&array[i], &array[min]);
  }
}

int main() {
  int data[] = {-10, 100, 12, 0, 12, 2, 1};
  int size = sizeof(data) / sizeof(data[0]);
  selectionSort(data, size);
  cout << "Sorted array in ascending order: " << endl;
  printArray(data, size);
}
```

PRACTICAL-2
INSERTION SORT

THEORY

Insertion Sort is a straightforward and comparison-based sorting algorithm that builds the final sorted array one element at a time. It mimics the way humans often sort items, like arranging playing cards in order.

In this method, the array is conceptually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed into the correct position in the sorted part by shifting the larger elements.

Insertion Sort is efficient for small datasets and performs well when the data is already partially sorted. It is also a **stable** and **in-place** sorting algorithm, meaning it does not require extra space and maintains the relative order of equal elements.

Time Complexity:

- **Best Case:** $O(n)$
- **Average Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$

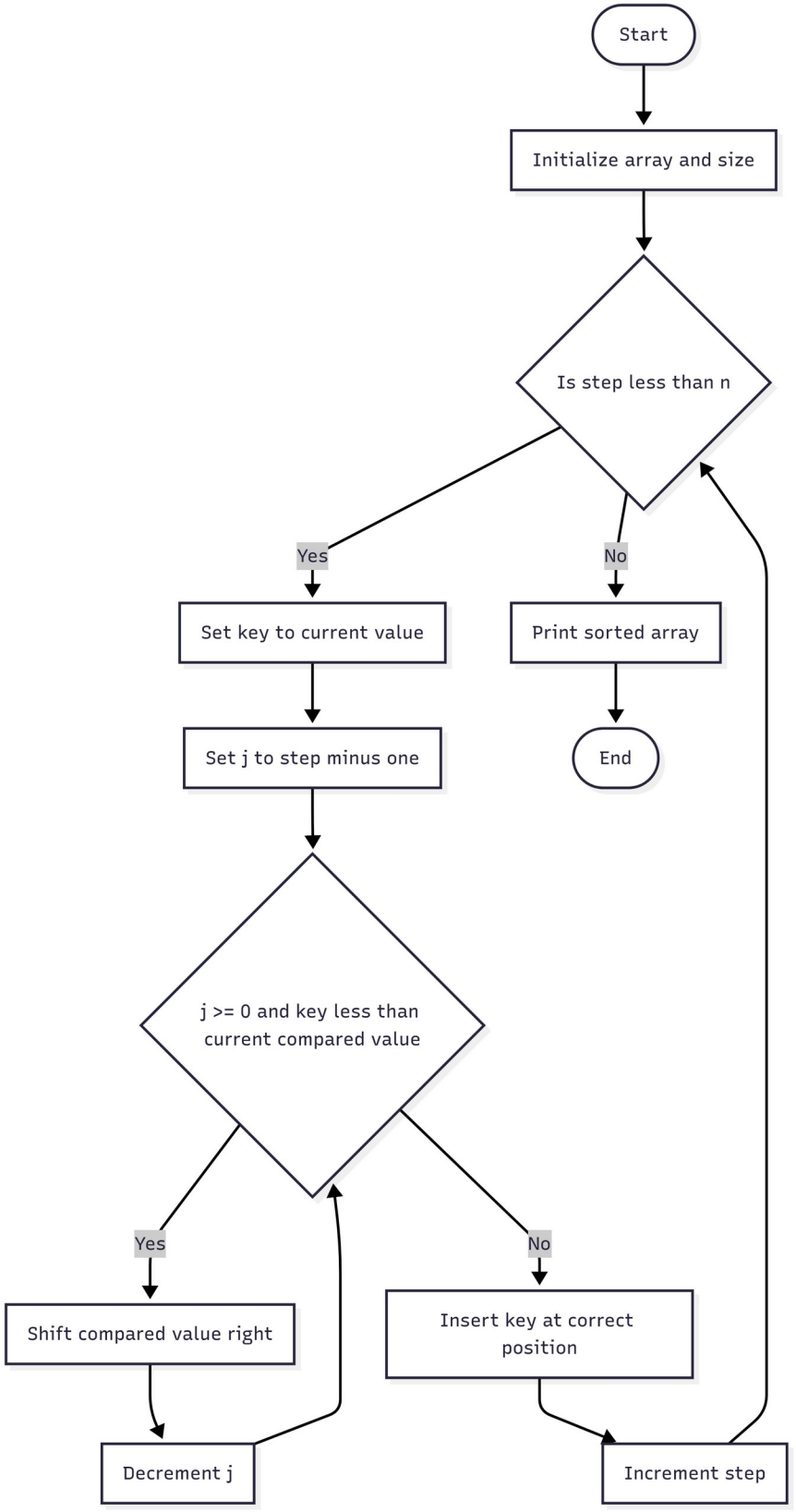
Characteristics:

- Stable sort
- In-place algorithm
- Performs well for nearly sorted data or small input sizes

ALGORITHM

1.Start
2.Input the array and its size.
3.For step from 1 to size - 1:
    1.Set key = array[step].
    2.Set j = step - 1.
    3.While j >= 0 and key < array[j]:
        •Set array[j + 1] = array[j].
        •Decrement j by 1.
    4.Set array[j + 1] = key.
4.Repeat steps 3.1 to 3.4 until the array is sorted.
5.Output the sorted array.
6.Stop

FLOWCHART


Start
Initialize array and size
Is step less than n
Yes
No
Set key to current value
Print sorted array
Set j to step minus one
End
j >= 0 and key less than current compared value
Yes
No
Shift compared value right
Insert key at correct position
Decrement j
Increment step

CODE:

```
#include <iostream>
using namespace std;

void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;
    while (j >= 0 && key < array[j]) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

int main() {
  int data[] = {-10, 100, 12, 0, 12, 2, 1};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  cout << "Sorted array in ascending order: " << endl;
  printArray(data, size);
}
```

PRACTICAL-3
QUICK SORT

THEORY

Quick Sort is a highly efficient, divide-and-conquer sorting algorithm. It works by selecting a **pivot** element from the array and partitioning the other elements into two sub-arrays — one containing elements less than the pivot and the other containing elements greater than the pivot. The process is then recursively applied to the sub-arrays.

The strength of Quick Sort lies in its **average-case efficiency** and ability to handle large datasets. It is typically faster in practice than other $O(n^2)$ algorithms like Selection Sort or Insertion Sort, especially when implemented with good pivot selection strategies (like choosing the median or using randomized pivots).

Although its worst-case time complexity is $O(n^2)$, this happens rarely (e.g., when the smallest or largest element is always chosen as the pivot), and in most practical cases, it performs close to $O(n \log n)$.

Time Complexity:

- **Best Case:** $O(n \log n)$
- **Average Case:** $O(n \log n)$
- **Worst Case:** $O(n^2)$

Space Complexity:

- **O(log n)** (for recursive stack calls)

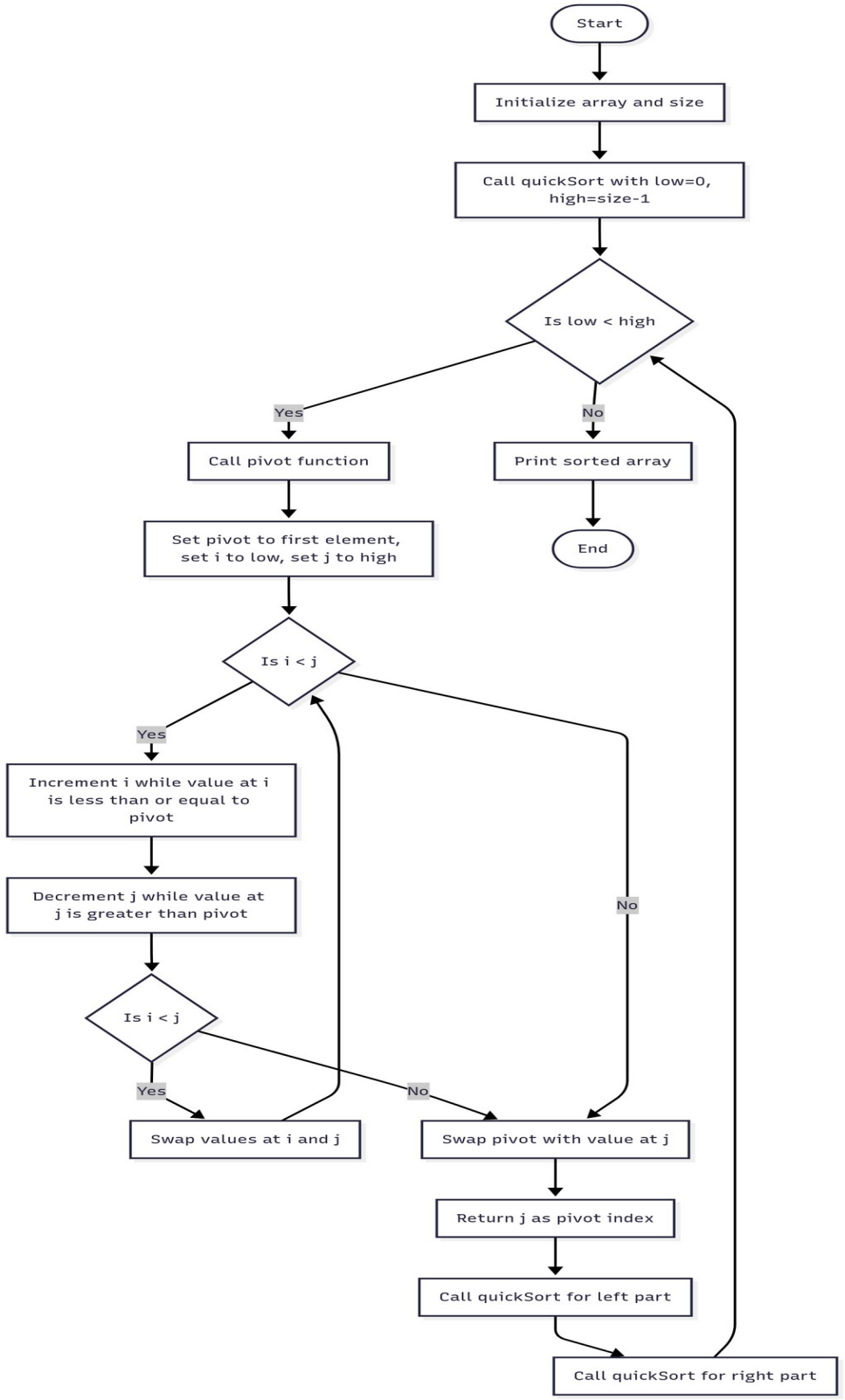
Characteristics:

- Not a stable sort
- In-place sorting algorithm
- Very efficient for large datasets
- Based on divide-and-conquer strategy

ALGORITHM

1.Start
2.Input the array and its size.
3.Call quickSort(array, low, high) with low = 0 and high = size - 1.
4.quickSort(array, low, high):
    1.If low < high:
        1.Call pivot(array, low, high) to partition the array:
            •Set pivotele = array[low], i = low, j = high.
            •While i < j:
                •Increment i while array[i] <= pivotele and i < high.
                •Decrement j while array[j] > pivotele and j > low.
                •If i < j, swap array[i] and array[j].
            •Swap array[low] and array[j].
            •Return j (pivot index).
        2.Recursively call quickSort(array, low, pivot_index - 1).
        3.Recursively call quickSort(array, pivot_index + 1, high).
5.Output the sorted array.
6.Stop

FLOWCHART


Start
Initialize array and size
Call quickSort with low=0, high=size-1
Is low < high
Yes
No
Call pivot function
Print sorted array
End
Set pivot to first element, set i to low, set j to high
Is i < j
Yes
No
Increment i while value at i is less than or equal to pivot
Decrement j while value at j is greater than pivot
Is i < j
Yes
No
Swap values at i and j
Swap pivot with value at j
Return j as pivot index
Call quickSort for left part
Call quickSort for right part

CODE:

```
#include <iostream>
using namespace std;

void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int pivot(int array[], int low, int high) {
  int pivotele = array[low];
  int i = low;
  int j = high;
  while (i < j) {
    while (i < high && array[i] <= pivotele) {
      i++;
    }
    while (j > low && array[j] > pivotele) {
      j--;
    }
    if (i < j) {
      swap(array[i], array[j]);
    }
  }
  swap(array[low], array[j]);
  return j;
}

void quickSort(int array[], int low, int high) {
  if (low < high) {
    int pe = pivot(array, low, high);
```

```
    quickSort(array, low, pe - 1);
    quickSort(array, pe + 1, high);
  }
}

int main() {
  int data[] = {-10, 100, 12, 0, 12, 2, 1};
  int size = sizeof(data) / sizeof(data[0]);
  quickSort(data, 0, size - 1);
  cout << "Sorted array in ascending order: " << endl;
  printArray(data, size);
}
```

CONCLUSION

In this lab, we studied and implemented various sorting algorithms, including Selection Sort, Insertion Sort, and Quick Sort. Each of these algorithms has its own advantages and trade-offs based on time complexity, space usage, and input size.

Selection Sort and Insertion Sort are simple to understand and implement, making them suitable for small datasets or educational purposes. However, they are inefficient for large datasets due to their $O(n^2)$ time complexity. On the other hand, Quick Sort, with its average-case time complexity of $O(n \log n)$, proves to be much faster and more efficient for large inputs, though it is not stable and may require careful pivot selection to avoid worst-case scenarios.

Understanding these algorithms helps build a strong foundation in algorithm design, time-space analysis, and problem-solving techniques essential for computer science and software development.