CADT IDT វិទ្យាស្ថានបច្ចេកវិទ្យាឌីជីថល Institute of Digital Technology

Course T1Y2: Advanced Algorithms

Lecturer: Bou Channa

Student's name: Chea Ilong

ID: 100022

Group: 1 SE Gen10

Sorting Algorithms

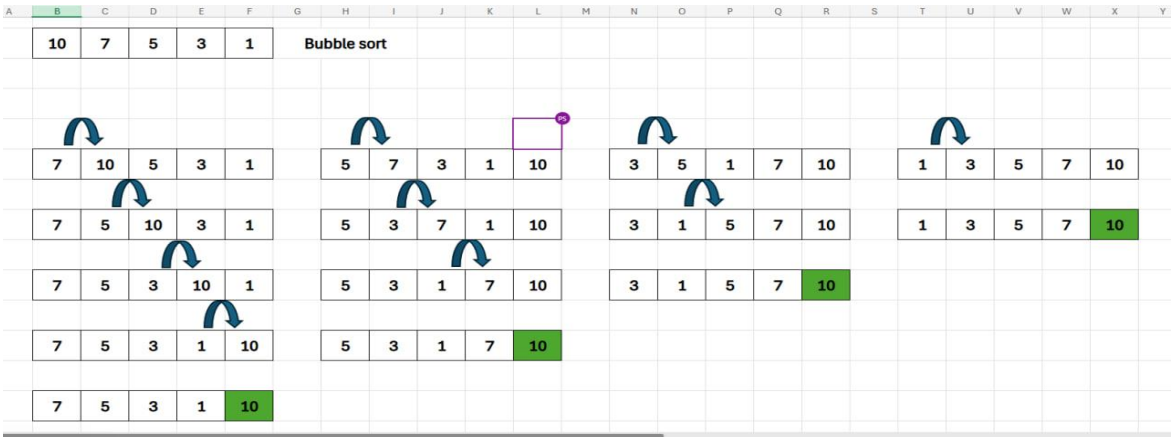
Sorting algorithms is a method that is use to arrange an array or a set of data in a specific order. There are various algorithms that have difference approach, efficiencies and use case like bubble sort, selection sort, insertion sort and so on.

1. Bubble Sort:

Bubble sort is a simple algorithm that is very common and easy to implement. It is use for sorting array of a set of data to arrange them into an order that is ascending or descending order. It works by repeatedly comparing the adjacent elements and swapping them if they are in the wrong arrange order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high $\boldsymbol{o(n^2)}$.

How it works:

1. Starting from the first element, compare the current element with the next element.
2. If the current element is greater than the next element, swap them.
3. Move to the next element and repeat the comparison and swapping until the end of the list is reached.
4. After the first pass, the largest element will have "bubbled up" to the last position in the list.
5. Repeat the process for the remaining elements (ignoring the last sorted elements) until the entire list is sorted

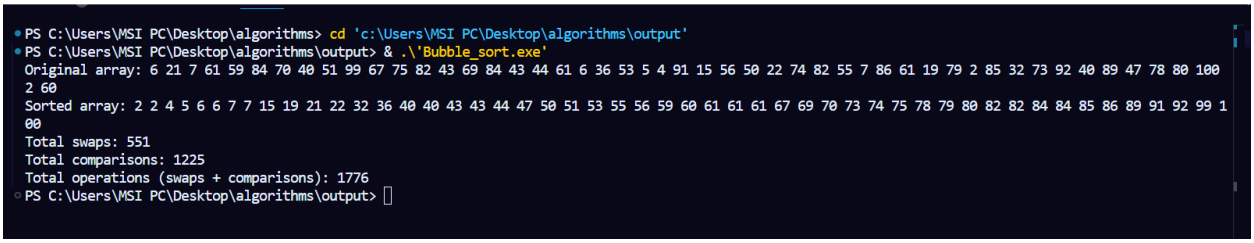


Source Code:

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

int main() {
    int n = 50;
    int array[n] = {
        6, 21, 7, 61, 59, 84, 70, 40, 51, 99, 67, 75, 82, 43, 69, 84, 43, 44, 61, 6,
        36, 53, 5, 4, 91, 15, 56, 50, 22, 74, 82, 55, 7, 86, 61, 19, 79, 2, 85, 32,
        73, 92, 40, 89, 47, 78, 80, 100, 2, 60
    };

    cout << "Before arrange: ";
    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                swap(array[j], array[j + 1]);
            }
        }
    }

    cout << "After arrange: ";
    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output: 50 elements test: As you can see from the output, we put we tested with 50 elements of number that are generated randomly and from the output statement we can see that the swap only happened 551 times and the comparison counted at 1225 times this mean that the algorithm only swap if the first element is greater than the second elements in the comparison(ascending order) and as a result the total number of operation is 1776times.

.



Time complexity:

Worst-case: $o(n^2)$: The worst-case scenario for bubble sort occurs when the input array is sorted in reverse order.

Average-case: $o(n^2)$: The average-case scenario occurs with a randomly arranged input array

Best-case: $o(n)$: In the best-case scenario for bubble sort, the input array is already sorted.

Advantages of Bubble Sort:

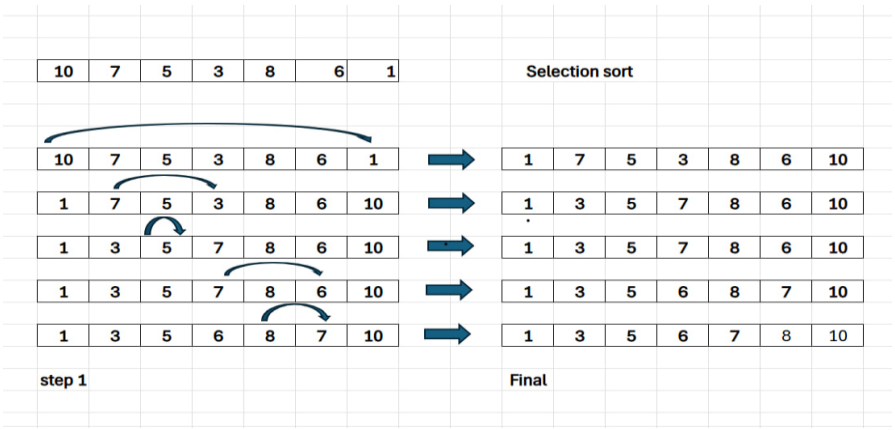
1. Bubble sort is straightforward to comprehend and implement.
2. It does not necessitate any extra memory space.
3. It is a stable sorting algorithm, which means that elements with identical key values retain their relative order in the sorted output.

2. Selection Sort:

Selection sort Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

How it works:

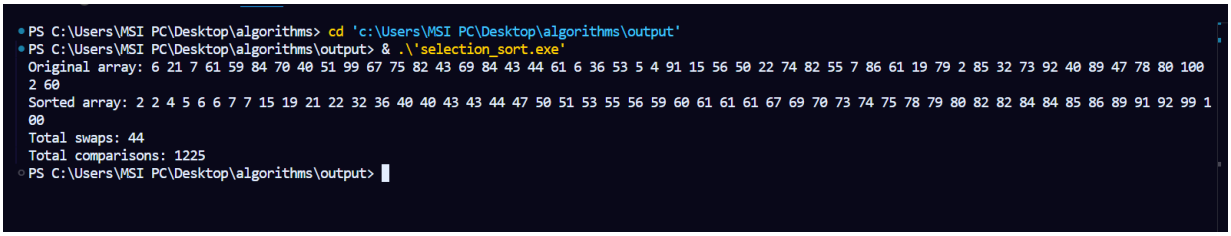
1. First, we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
3. Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.



Source code:

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

int main() {
    int n = 50;
   int array[n] = {
        6, 21, 7, 61, 59, 84, 70, 40, 51, 99, 67, 75, 82, 43, 69, 84, 43, 44, 61, 6,
        36, 53, 5, 4, 91, 15, 56, 50, 22, 74, 82, 55, 7, 86, 61, 19, 79, 2, 85, 32,
        73, 92, 40, 89, 47, 78, 80, 100, 2, 60
    };

    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < n; i++) {
        int min_index = i;

        for (int j = i + 1; j < n; j++) {
            if (array[min_index] > array[j]) {
                min_index = j;
            }
        }

        swap(array[min_index], array[i]);
    }

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << array[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Output: 50 elements test: As you can see from the output, we put we tested with 50 elements of number that are generated randomly and from the output statement we can see that the swap only happened 44 times and the comparison counted still at 1225 times. This indicates that the algorithm examines each element in the array, identifies the smallest element in each pass, and swaps it into the correct position. This approach minimizes the number of swaps by only making one per pass, placing the smallest element in the correct index for each iteration. Same elements won't be swap that is why there're so few.



Time complexity:

Worst-case: $o(n^2)$

Average-case: $o(n^2)$

Best-case: $o(n^2)$

In Selection Sort, regardless of the order of the input data, the algorithm goes through the entire list to find the smallest element in each pass and places it at the correct position. This means the time complexity is quadratic, which makes Selection Sort inefficient for large datasets.

Advantages of Selection Sort

1. Easy to understand and implement, making it ideal for teaching basic sorting concepts.
2. Requires only a constant O(1) extra memory space.
3. It requires a smaller number of swaps (or memory writes) compared to many other standard algorithms. Only cycle sort beats it in terms of memory writes. Therefore, it can be simple algorithm choice when memory writes are costly.

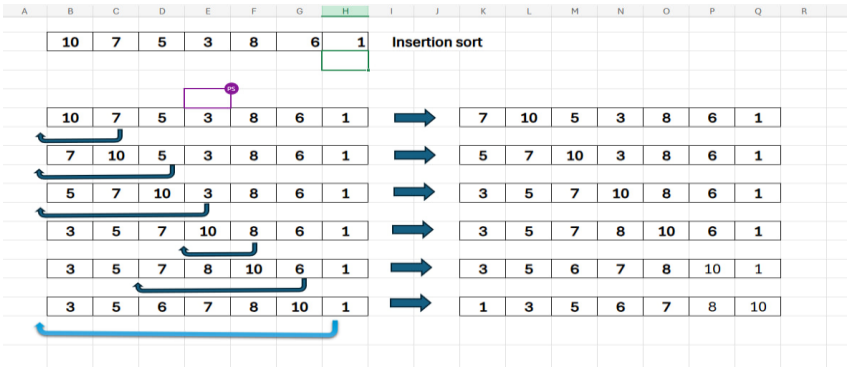
3.Insertion Sort:

Insertion Sort is a straightforward sorting algorithm that arranges elements by repeatedly placing each item from an unsorted portion into its correct position within an already sorted section of the list. It works much like organizing playing cards in your hand: you separate the cards into two groups — one sorted and one unsorted. Then, you pick a card from the unsorted group and place it into its correct spot within the sorted group, continuing this process until all cards are sorted.

How it works:

1. We start with second element of the array as first element in the array is assumed to be sorted.
2. Compare second element with the first element and check if the second element is smaller than swap them.

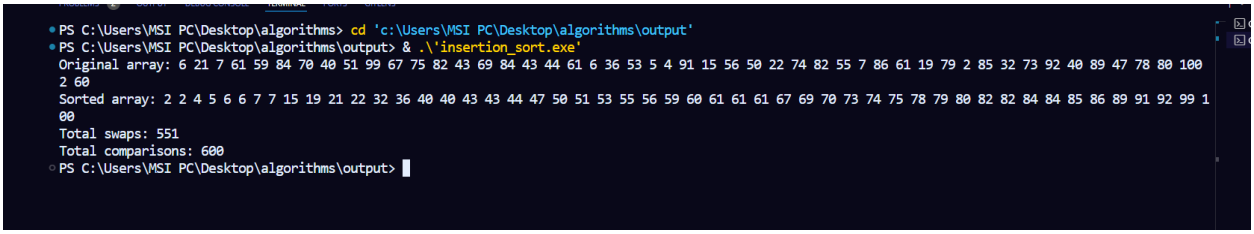
3. Move to the third element and compare it with the first two elements and put at its correct position
4. Repeat until the entire array is sorted.



Source code:

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

int main()
{
    int array[] = {
        6, 21, 7, 61, 59, 84, 70, 40, 51, 99, 67, 75, 82, 43, 69, 84, 43, 44, 61, 6,
        36, 53, 5, 4, 91, 15, 56, 50, 22, 74, 82, 55, 7, 86, 61, 19, 79, 2, 85, 32,
        73, 92, 40, 89, 47, 78, 80, 100, 2, 60
    };
    int n = sizeof(array) / sizeof(array[0]);

    cout << "Original array: ";
    for (int i = 0; i < n; ++i)
        cout << array[i] << " ";
    cout << endl;

    for (int i = 1; i < n; ++i)
    {
        int key = array[i];
        int j = i - 1;

        while (j >= 0 && array[j] > key)
        {
            array[j + 1] = array[j];
            j = j - 1;
        }
        array[j + 1] = key;
    }

    cout << "Sorted array: ";
    for (int i = 0; i < n; ++i)
        cout << array[i] << " ";
    cout << endl;

    return 0;
}
```

Output: 50 elements test: As you can see from the output, we put we tested with 50 elements of number that are generated randomly and from the output statement we can see that the swap only happened 551 times and the comparison counted still at 600 times. This indicate that elements were shifted multiple times, moving step-by-step until they reached their proper positions.



Time complexity:

Worst-case: $o(n^2)$: The worst-case scenario for insertion sort occurs when the input array is sorted in reverse order.

Average-case: $o(n^2)$**:** In the average case, the elements are in a random order.

Best-case: $o(n)$**:** The best-case scenario occurs when the array is already sorted.

Advantages of Insertion Sort:

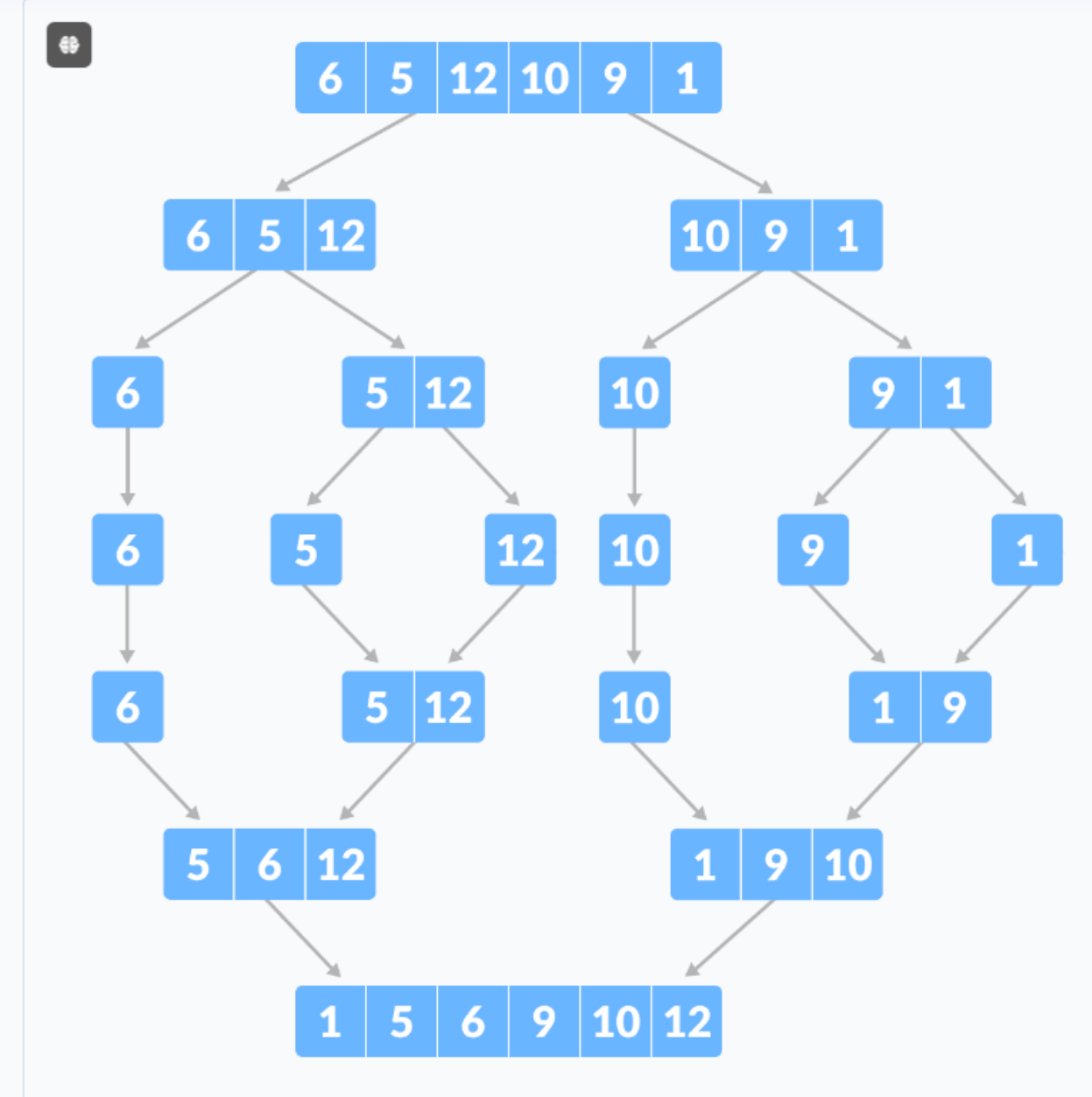
1. Simple and easy to implement.
2. Stable sorting algorithm.
3. Efficient for small lists and nearly sorted lists.
4. Space-efficient as it is an in-place algorithm.
5. Adoptive. the number of inversions is directly proportional to number of swaps. For example, no swapping happens for a sorted array and it takes O(n) time only.

4.Merge Sort:

Merge sort Merge sort is a sorting algorithm based on the divide-and-conquer strategy. It operates by recursively splitting the input array into smaller subarrays, sorting each one individually, and then merging them back together to form a fully sorted array. Simply put, merge sort works by dividing the array into two halves, sorting each half, and then merging the sorted halves. This process continues until the entire array is sorted.

How it works:

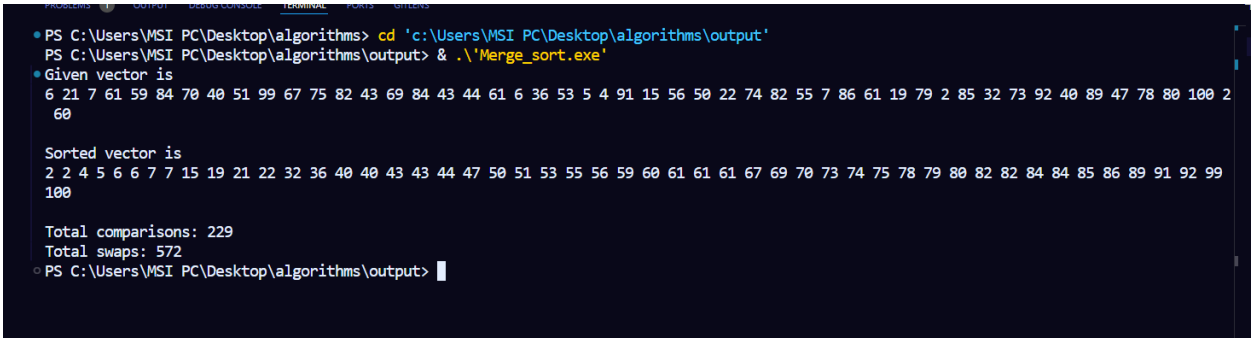
1. Divide: Divide the list or array recursively into two halves until it can no more be divided.
2. Conquer: Each subarray is sorted individually using the merge sort algorithm.
3. Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.


6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

Source code:

```cpp
#include <bits/stdc++.h>
using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

void printVector(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    vector<int> arr = { 12, 11, 13, 5, 6, 7 };
    int n = arr.size();

    cout << "Given vector is \n";
    printVector(arr);

    mergeSort(arr, 0, n - 1);

    cout << "\nSorted vector is \n";
    printVector(arr);
    return 0;
}
```

Output: Output: 50 elements test: As you can see from the output, we put we tested with 50 elements of number that are generated randomly and from the output statement we can see that the swap only happened 572 times and the comparison counted still at 229 times. As you can see from the output, we tested with 50 randomly generated elements. The results show that there were 572 swaps and 229 comparisons during the merge sort process. This indicates that merge sort efficiently organizes the elements by dividing the array into smaller subarrays, sorting them, and then merging them back together. The results highlight that swaps occur more frequently than comparisons, as each element may need to be shifted multiple times during the merging process, reflecting the algorithm's behavior.



Time complexity:

Best Case: $o(n\,log\,n)$: This occurs when the array is already sorted or nearly sorted. Even in this case, merge sort still divides and merges the subarrays.

Average Case $o(n\,log\,n)$: This is the expected time complexity for most scenarios since merge sort consistently divides the array into halves.

Worst Case: $o(n\,log\,n)$: In the worst case, the performance remains the same due to the nature of the divide-and-conquer approach.

Advantages of Merge Sort

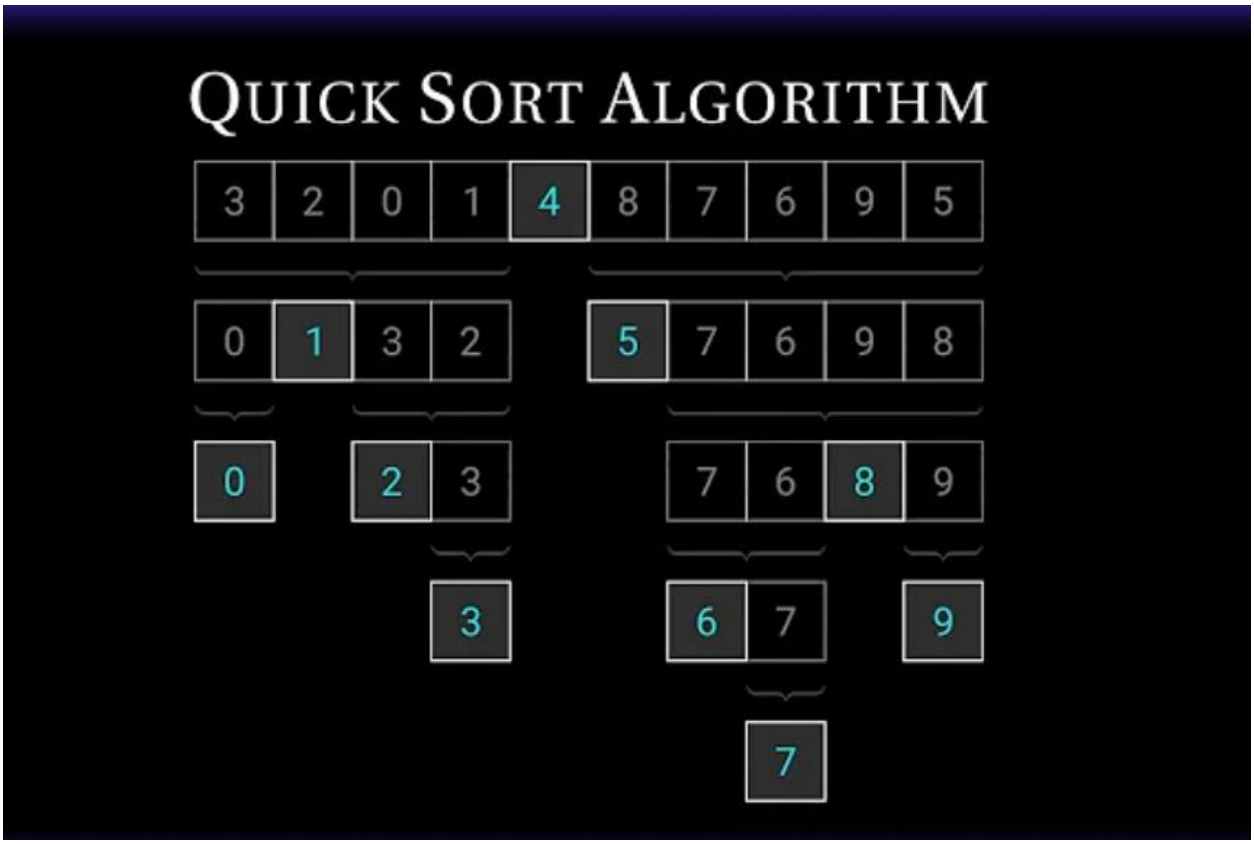
1. Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
2. Merge sort has a worst-case time complexity of $o(n\,log\,n)$:, which means it performs well even on large datasets.
3. The divide-and-conquer approach is straightforward.
4. We independently merge subarrays that makes it suitable for parallel processing.

5.Quick Sort:

Quick sort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How it works:

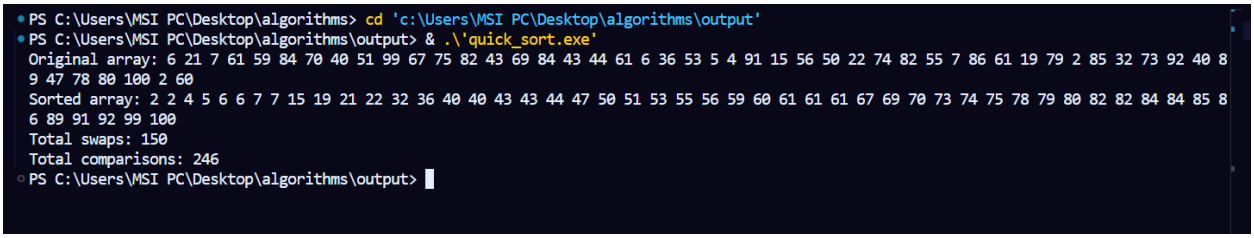
1. Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.



Source code:

```cpp
#include <bits/stdc++.h>
using namespace std;

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    int n = arr.size();
    quickSort(arr, 0, n - 1);

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}

```

Output: Output: 50 elements test: As you can see from the output, we put we tested with 50 elements of number that are generated randomly and from the output statement we can see that the swap only happened 150 times and the comparison counted still at 246 times. As you can see from the output, we tested with 50 randomly generated elements. The results show that there were 150 swaps and 246 comparisons during the quick sort process. The higher number of comparisons compared to swaps suggests that while many elements were assessed to determine their relative positions, the actual movement of elements (swaps) was less frequent.



Time complexity:

Best Case: $o(n\,log\,n)$: This occurs when the pivot divides the array into two nearly equal halves at each recursive step.

Average Case $o(n\,log\,n)$: On average, Quick Sort also performs well with a time complexity the same reasons as the best case.

Worst Case: $\boldsymbol{o(n^2)}$: The worst-case scenario occurs when the pivot is always the smallest or largest element, resulting in unbalanced partitions.

Advantages of Quick Sort

1. It is a divide-and-conquer algorithm that makes it easier to solve problems.
2. It is efficient on large data sets.
3. It has a low overhead, as it only requires a small amount of memory to function.
4. It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
5. Fastest general-purpose algorithm for large data when stability is not required.

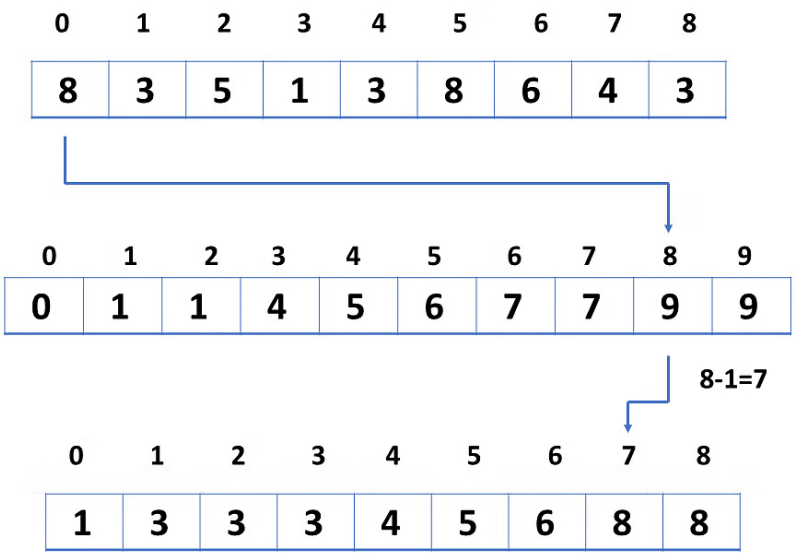
6. Counting Sort:

Counting Sort is a non-comparison-based sorting algorithm. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions.

How it works:

1. Find the Range: Determine the minimum and maximum values in the input array to find the range of the elements.
2. Initialize the Count Array: Create a count array (of size equal to the range of input values) to store the count of each unique element from the input array.
3. Count the Elements: Iterate through the input array and populate the count array. For each element in the input array, increment the corresponding index in the count array.
4. Cumulative Count: Modify the count array by adding the count of the previous index to the current index. This gives the position of each element in the sorted output.
5. Build the Output Array: Create an output array and fill it by placing elements from the input array into their correct positions based on the cumulative counts.
6. Copy Back to Original Array: Copy the sorted elements from the output array back to the original array to reflect the sorted order.

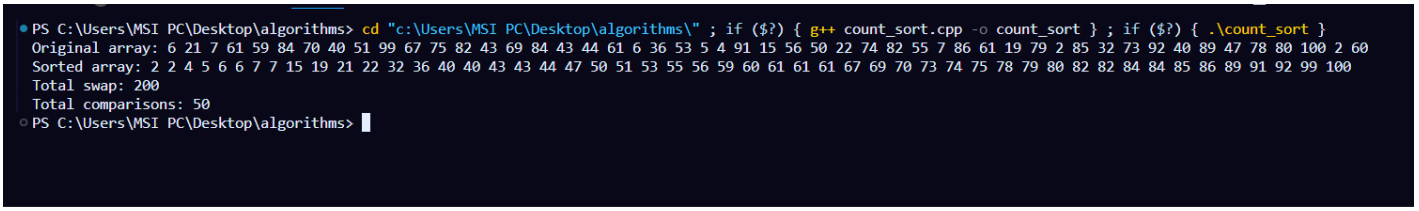
Counting sort



Source Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> countSort(vector<int> &inputArray)
{

    int N = inputArray.size();

    int M = 0;

    for (int i = 0; i < N; i++)
        M = max(M, inputArray[i]);

    vector<int> countArray(M + 1, 0);

    for (int i = 0; i < N; i++)
        countArray[inputArray[i]]++;

    for (int i = 1; i <= M; i++)
        countArray[i] += countArray[i - 1];

    vector<int> outputArray(N);

    for (int i = N - 1; i >= 0; i--)

    {
        outputArray[countArray[inputArray[i]] - 1] = inputArray[i];

        countArray[inputArray[i]]--;
    }

    return outputArray;
}

int main()

{

    vector<int> inputArray = {4, 3, 12, 1, 5, 5, 3, 9};

    vector<int> outputArray = countSort(inputArray);

    for (int i = 0; i < inputArray.size(); i++)
        cout << outputArray[i] << " ";

    return 0;
}

```

Output: Output: 50 elements test: As you can see from the output, we tested with 50 randomly generated elements. The results show that there were 150 assignments (placements) and 246 comparisons during the counting sort process. The relatively high count of comparisons indicates that counting sort requires multiple steps to build the count and cumulative arrays, while the actual movement of elements (assignments) is controlled and occurs only during the final placement in the output array.



Time complexity:

Best Case: $o(n + K)$: When the range of elements (k) is small relative to the number of elements (n)

Average Case $o(n + K)$: Generally, in typical scenarios

Worst Case:$o(n + K)$: : Counting sort becomes inefficient if the range of values (k) is very large relative to n

Where:

1. n is the number of elements in the input array.
2. k is the range of the input (i.e., the maximum value in the array).

Advantages of Counting Sort

1. Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort and quicksort, if the range of input is of the order of the number of inputs.
2. Counting sort is easy to code
3. Counting sort is a stable algorithm.

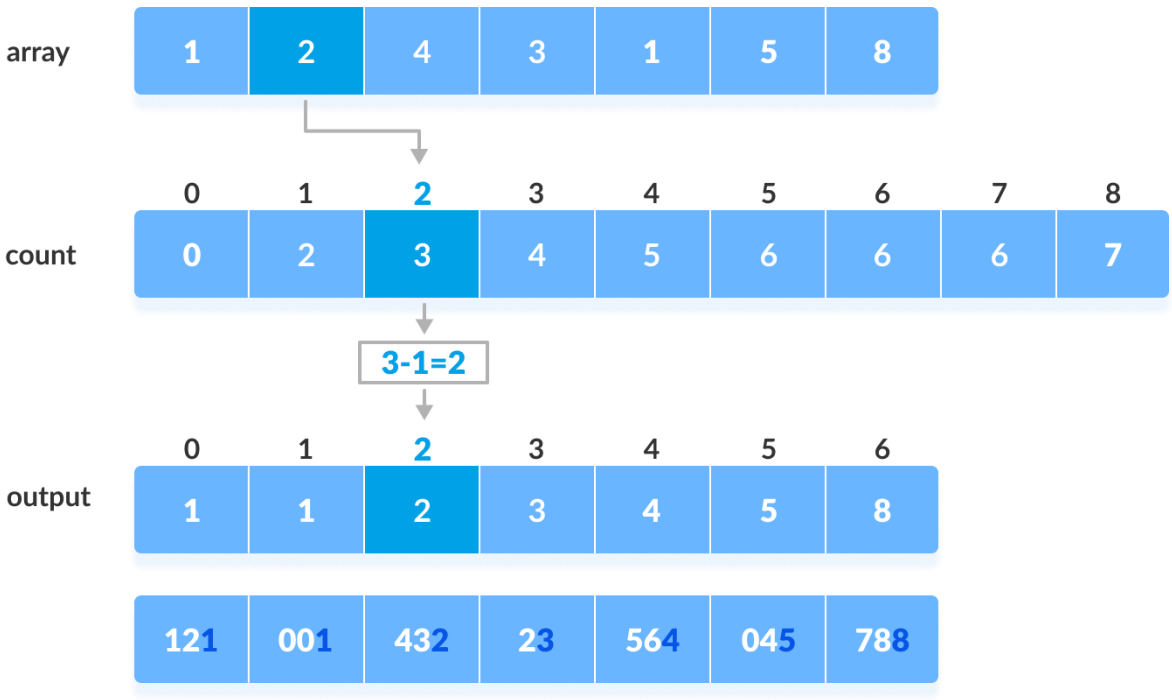
7.Radix Sort:

Radix Sort is a linear sorting algorithm that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

How it works:

1. Find Maximum Digits: Determine the largest number in the array to know how many digits it has. This tells you how many times you need to sort the array.
2. Sort by Each Digit: Start with the least significant digit (rightmost) and sort the array using a stable sorting method, like counting sort.
3. Count the Digits: For the current digit, count how many times each digit (0-9) appears in the array.
4. Cumulative Count: Modify the count array so that each position holds the cumulative count. This helps to determine where each digit should go in the output.
5. Build Output Array: Use the cumulative counts to place the elements in their correct positions in a new output array.
6. Copy Back: Copy the sorted output back to the original array.
7. Repeat: Move to the next more significant digit (tens, hundreds, etc.) and repeat steps 2-6 until all digit places have been sorted.

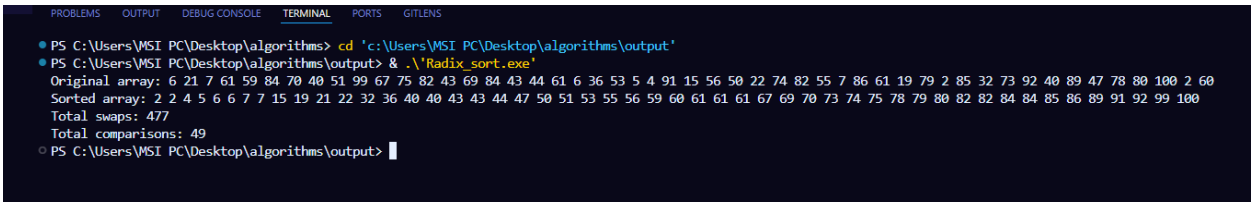
Radix Sort



Source code:

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

int getMax(int arr[], int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

void countSort(int arr[], int n, int exp) {
    int output[n];
    int i, count[10] = { 0 };

    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixsort(int arr[], int n) {
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

void print(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

int main() {
    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
    int n = sizeof(arr) / sizeof(arr[0]);

    radixsort(arr, n);
    print(arr, n);
    return 0;
}
```

Output: Output: 50 elements test: For the test with 50 randomly generated elements, the results demonstrate that there was a total of 477 and 49 during the radix sort process. The significant number of comparisons reflects the multiple iterations required for processing each digit of the elements, while the assignment count indicates how elements are organized into the output array.



Time complexity:

Best Case: $o(nK)$: This occurs when the input consists of n elements and k is the number of digits in the maximum number.

Average Case $o(nK)$: Similar to the best case, the average case also involves processing each of the k digits for all n elements.

Worst Case:$o(nK)$: The worst-case scenario: regardless of the input, radix sort processes all n elements for each of the k digits.

Where:

1. n is the total count of elements being sorted.
2. k is related to the maximum number in the dataset, specifically how many digits that number has.

Advantages of Radix Sort:

1. Preserves the relative order of equal elements.
2. Can achieve O(n) performance under certain conditions, especially with fixed-length keys.
3. Does not rely on element comparisons, making it efficient for large datasets.
4. Particularly suited for sorting numbers or strings of fixed length.

8.Bucket Sort:

Bucket sort is a sorting technique that involves dividing elements into various groups, or buckets. These buckets are formed by uniformly distributing the elements. Once the elements are divided into buckets, they can be sorted using any other sorting algorithm. Finally, the sorted elements are gathered together in an ordered fashion.

How it works:

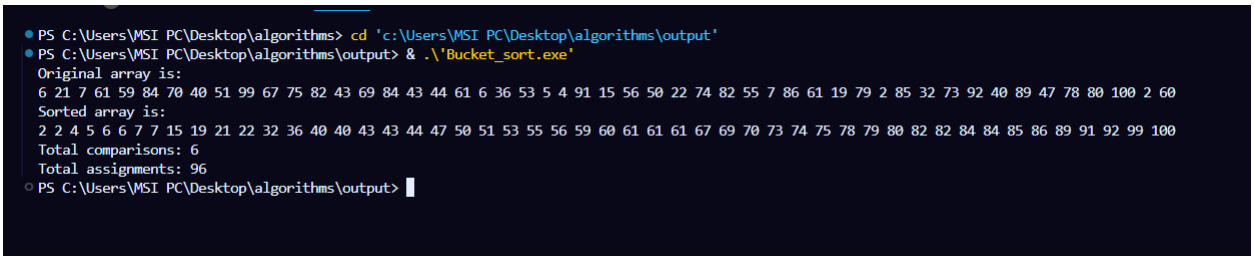
1. Create Buckets: Divide the input range into a fixed number of equally spaced intervals or "buckets." Each bucket will hold a portion of the input elements.
2. Distribute Elements: Iterate through the input array and distribute the elements into their corresponding buckets based on their value.
3. Sort Individual Buckets: Sort each bucket individually using a stable sorting algorithm, such as insertion sort or quicksort.
4. Merge Buckets: Concatenate the sorted elements from each bucket back into a single output array.
5. Return the Sorted Array: The output array is now sorted, and you can return it as the result of the bucket sort.

Bucket Sort

Source Code:

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

void insertionSort(vector<float>& bucket, int& comparisonCount, int& assignmentCount) {
    for (int i = 1; i < bucket.size(); ++i) {
        float key = bucket[i];
        assignmentCount++;
        int j = i - 1;
        while (j >= 0 && bucket[j] > key) {
            comparisonCount++;
            bucket[j + 1] = bucket[j];
            assignmentCount++;
            j--;
        }
        bucket[j + 1] = key;
        assignmentCount++;
    }
}

void bucketSort(float arr[], int n, int& comparisonCount, int& assignmentCount) {
    vector<float> b[n];

    for (int i = 0; i < n; i++) {
        int bi = n * arr[i];
        b[bi].push_back(arr[i]);
    }

    for (int i = 0; i < n; i++) {
        insertionSort(b[i], comparisonCount, assignmentCount);
    }

    int index = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < b[i].size(); j++) {
            arr[index++] = b[i][j];
            assignmentCount++;
        }
    }
}

int main() {
    float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original array is: \n";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    int comparisonCount = 0;
    int assignmentCount = 0;

    bucketSort(arr, n, comparisonCount, assignmentCount);

    cout << "Sorted array is: \n";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    cout << "Total comparisons: " << comparisonCount << endl;
    cout << "Total assignments: " << assignmentCount << endl;

    return 0;
}

```

Output: For the test with 50: randomly generated elements, the results demonstrate that there was a total of 6 comparisons and 96 assignments during the bucket sort process. The low number of comparisons suggests that the insertion sort was efficient for sorting the small buckets. The higher assignment count indicates how the elements were organized into their respective buckets and then rearranged into the final sorted output.



Time complexity:

Best Case: $o(n + K)$: Similar to the average case, as the elements are well distributed.

Average Case $o(n + K)$: This assumes that the input is uniformly distributed across the buckets, leading to fewer elements in each bucket.

Worst Case: $\boldsymbol{o(n^2)}$: This occurs when all elements are placed into a single bucket,

Where n is the number of elements in the input array and k is the number of buckets.

Advantages of Bucket Sort:

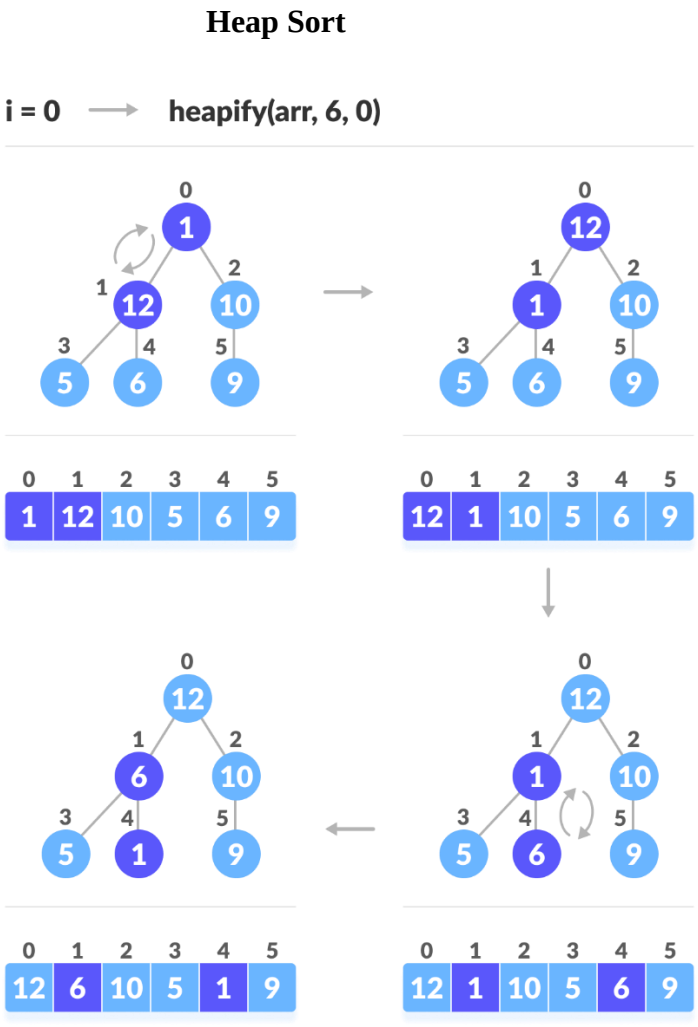
1. can achieve linear time complexity for uniformly distributed data.
2. It maintains the relative order of equal elements, making it useful when order matters.
3. The algorithm is easy to understand and implement.
4. It can use various sorting algorithms for individual buckets, optimizing performance.

9.Heap Sort:

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

How it works:

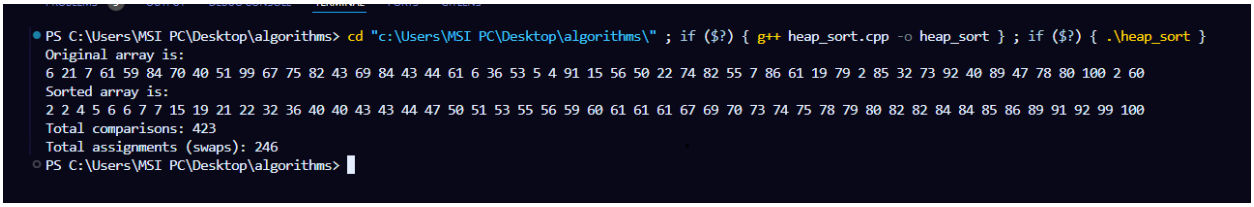
1. **Build the Max Heap:** Rearrange the array into a max heap structure, where the largest element is at the root of the heap. This ensures that every parent node is greater than its child nodes.
2. **Extract the Maximum:** Swap the root element (maximum) with the last element in the heap, effectively removing it from the heap.
3. **Reduce Heap Size:** Decrease the size of the heap by one, excluding the last element (which is now sorted).
4. **Heapify:** Restore the max heap property by heapifying the root node. This involves comparing the root with its children and swapping it with the larger child until the heap property is satisfied.
5. **Repeat:** Repeat steps 2-4 until all elements are sorted. Continue extracting the maximum and heapifying the root until the heap is empty.
6. **Final Sorted Array:** The array will be sorted in ascending order as a result of repeatedly extracting the maximum from the heap and placing it in the correct position.



Source code:

```cpp
#include <bits/stdc++.h>
using namespace std;

void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;

    if (r < n && arr[r] > arr[largest])
        largest = r;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

void printArray(vector<int>& arr) {
    for (int i = 0; i < arr.size(); ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

int main() {
    vector<int> arr = { 9, 4, 3, 8, 10, 2, 5 };

    heapSort(arr);

    cout << "Sorted array is \n";
    printArray(arr);
}
```

Output: 50 elements test: For the test with 50 randomly generated elements, the results demonstrate that there was a total of 423 comparisons and 246 assignments during the heap sort process. The relatively high number of comparisons reflects the numerous checks made to maintain the heap property during the sorting, while the assignment count indicates how elements were swapped in the array to achieve the final sorted order.



Time complexity:

Best Case: $o(n\,log\,n)$: The best-case scenario occurs when the input array is structured in such a way that the heap construction and the subsequent sorting phases are both efficient.

Average $o(n\,log\,n)$: The average case assumes that the elements are randomly distributed throughout the input array.

Worst Case:$o(n\,log\,n)$: The worst-case scenario for heap sort can occur when all elements are placed into a single bucket or, more generally, when the input data is structured in a way that leads to maximum swaps during heapification.

Advantages of Heap Sort:

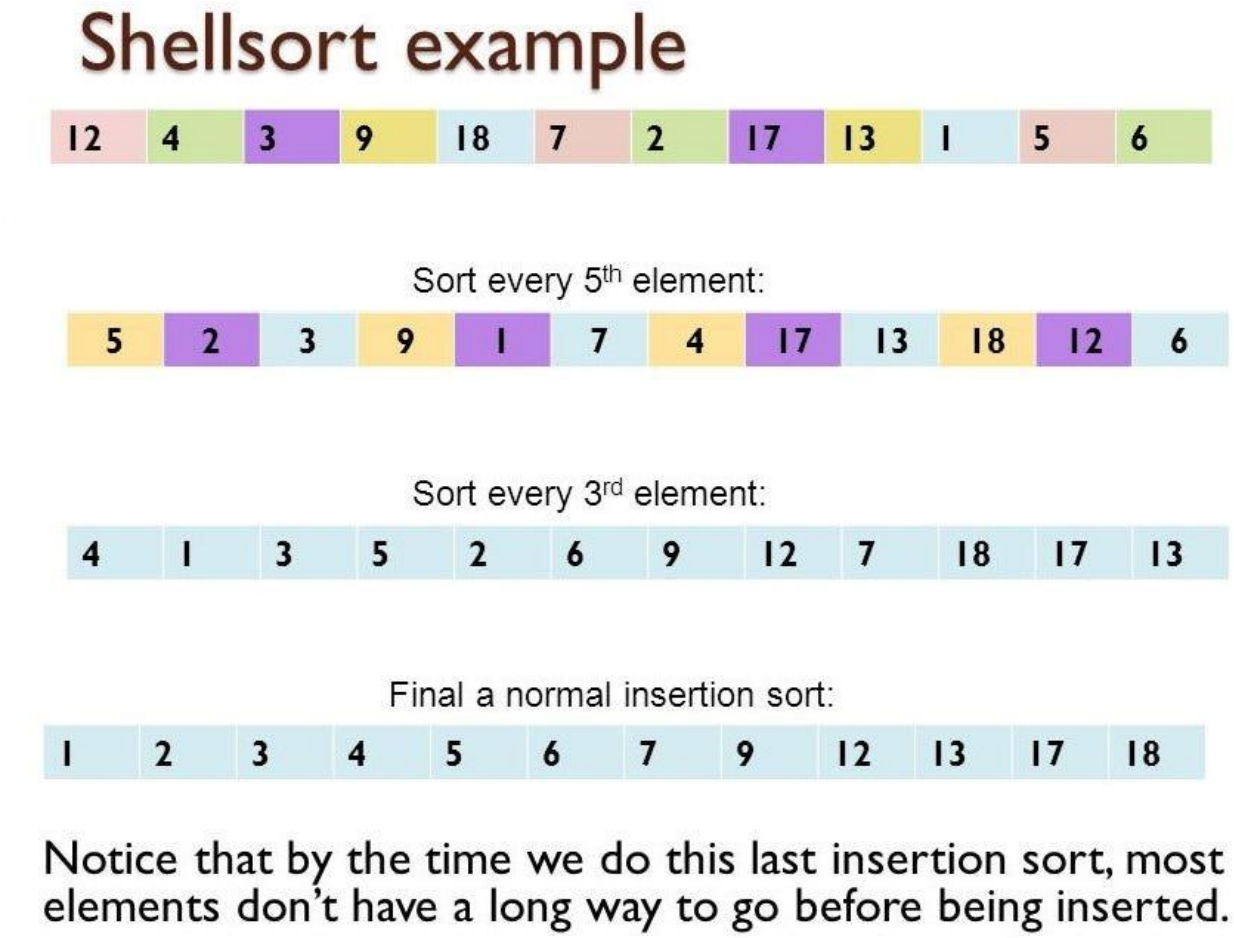
1. Efficient for large data sets with a time complexity of $o(n\,log\,n)$
2. In-place sorting algorithm, requiring minimal additional space.
3. Works well with data that cannot fit into memory due to its ability to sort in chunks.
4. Guarantees a consistent runtime regardless of the input distribution.

10.Shell Sort:

Shell sort is mainly a variation of Insertion Sort. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all subsists of every h element are sorted.

How it works:

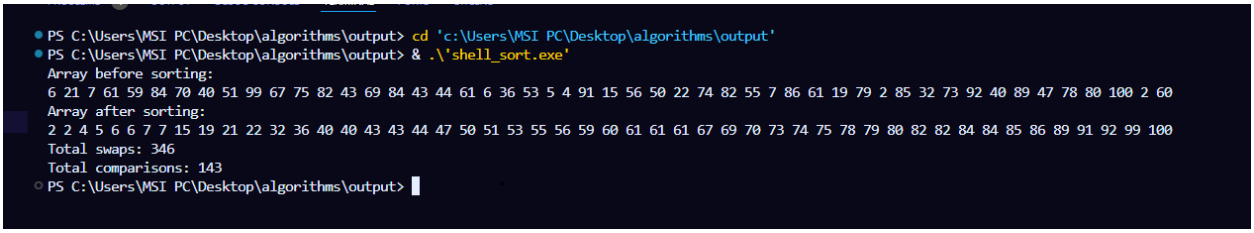
Step 1 − Start
Step 2 − Initialize the value of gap size, say h.
Step 3 − Divide the list into smaller sub-part. Each must have equal intervals to h.
Step 4 − Sort these sub-lists using insertion sort.
Step 5 – Repeat this step 2 until the list is sorted.
Step 6 – Print a sorted list.
Step 7 – Stop.



Source code:

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

int shellSort(int arr[], int n)
{
    for (int gap = n / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < n; i += 1)
        {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];
            arr[j] = temp;
        }
    }
    return 0;
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

int main()
{
    int arr[] = {12, 34, 54, 2, 3}, i;
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Array before sorting: \n";
    printArray(arr, n);

    shellSort(arr, n);

    cout << "\nArray after sorting: \n";
    printArray(arr, n);

    return 0;
}

```

Output: For the test with 50: randomly generated elements, the results demonstrate that there was a total of 143 comparisons and 346 assignments during the Shell Sort process. The number of comparisons reflects the multiple iterations and checks required to find the correct position for each element during the sorting process, while the assignment count indicates how many times elements were moved within the array to achieve the final sorted order.



Time complexity:

Best Case: $o(n\,log\,n)$: This scenario occurs when the input array is already partially sorted.

Average $o(n\,log\,n)$: In a typical scenario, Shell Sort performs at this complexity due to the gap reduction strategy, which allows for a relatively balanced distribution of elements during the sorting process.

Worst Case:$o(n^2)$:: This occurs when the array is sorted in reverse order or has a poor gap sequence.

Advantages of Shell Sort:

1. Shell Sort improves upon insertion sort's performance, especially on large lists.
2. The algorithm's performance can improve on partially sorted arrays.
3. It requires a small, constant amount of additional memory space.
4. Shell Sort is easy to implement and understand, making it accessible for beginners.