ELEMENTARY SORTING TECHNIQUES

BUBBLE SORT

- Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
- The process is repeated until the list is sorted.

Steps of Bubble Sort:

- Start at the beginning of the list.
- Compare the first two elements. If the first is greater than the second, swap them.
- Move to the next pair of elements, and repeat the comparison and swap if necessary.
- Continue this process for each pair of adjacent elements until you reach the end of the list.
- After each pass through the list, the largest element will have "bubbled up" to its correct position.
- Repeat the process for the remaining unsorted portion of the list.
- Continue until no swaps are needed, indicating that the list is sorted.

Example:

Given an array: [5, 1, 4, 2, 8]

First Pass:

- Compare 5 and 1: Swap them → [1, 5, 4, 2, 8]
- Compare 5 and 4: Swap them → [1, 4, 5, 2, 8]
- Compare 5 and 2: Swap them → [1, 4, 2, 5, 8]
- Compare 5 and 8: No swap → [1, 4, 2, 5, 8]

Second Pass:

- Compare 1 and 4: No swap → [1, 4, 2, 5, 8]
- Compare 4 and 2: Swap them → [1, 2, 4, 5, 8]
- Compare 4 and 5: No swap → [1, 2, 4, 5, 8]
- Compare 5 and 8: No swap → [1, 2, 4, 5, 8]

Third Pass:

- Compare 1 and 2: No swap → [1, 2, 4, 5, 8]
- Compare 2 and 4: No swap → [1, 2, 4, 5, 8]
- Compare 4 and 5: No swap → [1, 2, 4, 5, 8]
- Compare 5 and 8: No swap → [1, 2, 4, 5, 8]

After these passes, the array is sorted: [1, 2, 4, 5, 8].

Code:

```
#include<iostream>
using namespace std;
   void print(int a[], int n) //function to print array elements
   {
   int i;
   for(i = 0; i < n; i++)
   {
     cout<<a[i]<<" ";
   }
   }
 void bubble(int a[], int n) // function to implement bubble sort
{
 int i, j, temp;
  for(i = 0; i < n; i++)
   {
    for(j = i+1; j < n; j++)
     {
        if(a[j] < a[i])
        {
           temp = a[i];
           a[i] = a[j];
```

```
            a[j] = temp;
          }
       }
    }

 }
int main()
{
   int i, j,temp;
   int a[5] = {4, 1, 2, 8, 5};
   int n = sizeof(a)/sizeof(a[0]);
   cout<<"Before sorting array elements are - \n";
   print(a, n);
   bubble(a, n);
   cout<<"\nAfter sorting array elements are - \n";
   print(a, n);
return 0;
}
```

Time Complexity:

Worst and Average Case: $O(n^2)$

SELECTION SORT

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on 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.

Steps of Selection Sort:

1. Start with the first element in the array.
2. Find the smallest element in the remaining unsorted portion of the array.
3. Swap the smallest element found with the first unsorted element.
4. Move to the next element and repeat steps 2-3 until the entire array is sorted.

Example:

Given an array: [64, 25, 12, 22, 11]

First Pass:

- Find the smallest element in [64, 25, 12, 22, 11], which is 11.
- Swap 11 with 64 → [11, 25, 12, 22, 64].

Second Pass:

- Find the smallest element in [25, 12, 22, 64], which is 12.
- Swap 12 with 25 → [11, 12, 25, 22, 64].

Third Pass:

- Find the smallest element in [25, 22, 64], which is 22.
- Swap 22 with 25 → [11, 12, 22, 25, 64].

Fourth Pass:

- Find the smallest element in [25, 64], which is 25.
- No swap needed → [11, 12, 22, 25, 64].

After these passes, the array is sorted: [11, 12, 22, 25, 64].

Code:

```
#include <iostream>

using namespace std;

void selection(int arr[], int n)

{

   int i, j, small;
```

```
   for (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
   {
      small = i; //minimum element in unsorted array

      for (j = i+1; j < n; j++)
      if (arr[j] < arr[small])
         small = j;
// Swap the minimum element with the first element
   int temp = arr[small];
   arr[small] = arr[i];
   arr[i] = temp;
   }
}

void printArr(int a[], int n) /* function to print the array */
{
   int i;
   for (i = 0; i < n; i++)
      cout<< a[i] <<" ";
}

int main()
{
```

```
    int a[] = { 64, 25, 12, 22, 11 };

    int n = sizeof(a) / sizeof(a[0]);

    cout<< "Before sorting array elements are - "<<endl;

    printArr(a, n);

    selection(a, n);

    cout<< "\nAfter sorting array elements are - "<<endl;

    printArr(a, n);



    return 0;

}
```

Key Characteristics:

- **Time Complexity:** O(n^2) for all cases (worst, average, and best).

INSERTION SORT

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time. It works similarly to sorting playing cards in your hand. The algorithm repeatedly takes the next unsorted element, compares it to elements in the sorted part of the array, and inserts it into the correct position.

Steps of Insertion Sort:

1. Start with the second element (since the first element is trivially sorted).
2. Compare the current element with the elements in the sorted part of the array (to its left).
3. Shift elements of the sorted part to the right if they are greater than the current element.
4. Insert the current element into its correct position.
5. Repeat steps 2-4 for all elements in the array.

Example:

Given an array: [12, 11, 13, 5, 6]

First Pass:

- The element 11 is compared with 12.
- Since 11 < 12, move 12 to the right and insert 11 in the first position → [11, 12, 13, 5, 6].

Second Pass:

- The element 13 is compared with 12 and 11.
- Since 13 > 12, no shifting is required, and 13 stays in place → [11, 12, 13, 5, 6].

Third Pass:

- The element 5 is compared with 13, 12, and 11.
- Since 5 < 13, 5 < 12, and 5 < 11, move all of them to the right and insert 5 in the first position → [5, 11, 12, 13, 6].

Fourth Pass:

- The element 6 is compared with 13, 12, and 11.
- Since 6 < 13, 6 < 12, and 6 > 11, move 13 and 12 to the right and insert 6 in its correct position → [5, 6, 11, 12, 13].

After these passes, the array is sorted: [5, 6, 11, 12, 13].

Code:

```
#include <iostream>

using namespace std;


void insert(int a[], int n) /* function to sort an aay with insertion sort */

{

   int i, j, temp;

   for (i = 1; i < n; i++) {

      temp = a[i];

      j = i - 1;
```

```
    while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to one position ahea
d from their current position*/
    {
        a[j+1] = a[j];
        j = j-1;
    }
    a[j+1] = temp;
  }
}

void printArr(int a[], int n) /* function to print the array */
{
  int i;
  for (i = 0; i < n; i++)
    cout << a[i] <<" ";
}

int main()
{
  int a[] = { 12, 11, 13, 5, 6 };
  int n = sizeof(a) / sizeof(a[0]);
  cout<<"Before sorting array elements are - "<<endl;
  printArr(a, n);
  insert(a, n);
  cout<<"\nAfter sorting array elements are - "<<endl;
  printArr(a, n);
```

```
    return 0;
}
```

Key Characteristics:

- **Time Complexity:** O(n^2) for worst and average cases, O(n) for best case (when the array is already sorted).

RADIX SORT

Radix Sort is a non-comparative sorting algorithm that sorts numbers by processing individual digits. It sorts the elements by their least significant digit (LSD) or most significant digit (MSD), depending on the variant. The algorithm typically uses counting sort or bucket sort as a subroutine to sort digits at each significant place.

Steps of Radix Sor:

1. Determine the maximum number of digits in the largest number in the array.
2. Sort the array digit by digit, starting from the least significant digit (LSD) to the most significant digit (MSD).
3. Use a stable sorting algorithm (like Counting Sort) to sort elements by the current digit.

Example:

Given an array: [170, 45, 75, 90, 802, 24, 2, 66]

Step 1: Find the maximum number of digits. Here, the largest number is 802, which has 3 digits.

Step 2: Perform Counting Sort for each digit (units, tens, hundreds).

- **First Pass (Units place):**
  - Sort based on the last digit.
  - Result: [170, 90, 802, 2, 24, 45, 75, 66]
- **Second Pass (Tens place):**
  - Sort based on the second last digit.
  - Result: [802, 2, 24, 45, 66, 170, 75, 90]
- **Third Pass (Hundreds place):**
  - Sort based on the hundreds digit.
  - Result: [2, 24, 45, 66, 75, 90, 170, 802]

After these passes, the array is sorted: [2, 24, 45, 66, 75, 90, 170, 802].

Code:

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

int getMax(int a[], int n) {
  int max = a[0];
  for(int i = 1; i<n; i++) {
    if(a[i] > max)
      max = a[i];
  }
  return max; //maximum element from the array
}

void countingSort(int a[], int n, int place) // function to implement counting sort
{
 int output[n + 1];
 int count[10] = {0};

 // Calculate count of elements
 for (int i = 0; i < n; i++)
  count[(a[i] / place) % 10]++;

 // Calculate cumulative frequency
 for (int i = 1; i < 10; i++)
  count[i] += count[i - 1];

 // Place the elements in sorted order
```

```
  for (int i = n - 1; i >= 0; i--) {
    output[count[(a[i] / place) % 10] - 1] = a[i];
    count[(a[i] / place) % 10]--;
  }

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

// function to implement radix sort
void radixsort(int a[], int n) {

  // get maximum element from array
  int max = getMax(a, n);

  // Apply counting sort to sort elements based on place value
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(a, n, place);
}

// function to print array elements
void printArray(int a[], int n) {
  for (int i = 0; i < n; ++i)
    cout<<a[i]<<" ";
}

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

```
 cout<<"Before sorting array elements are - \n";
 printArray(a,n);
 radixsort(a, n);
 cout<<"\n\nAfter applying Radix sort, the array elements are - \n";
 printArray(a, n);
 return 0;
}
```

Key Characteristics:

- **Time Complexity:** O(d * (n + k)), where d is the number of digits, n is the number of elements, and k is the range of the digit values (for example, 0-9 for decimal numbers).

SHELL SORT

Shell Sort is an in-place comparison-based sorting algorithm that generalizes Insertion Sort. It allows the exchange of far-apart elements, which can move elements closer to their correct position more quickly than a simple Insertion Sort. The idea is to arrange the list in such a way that, starting from a large gap between elements, the gap is reduced until it becomes 1, at which point the array is sorted using a regular Insertion Sort.

Steps of Shell Sort:

1. **Start with a large gap size.** The gap is usually initialized to half the length of the array.
2. **Perform a "gapped" insertion sort** for each element, comparing and swapping elements that are gap positions apart.
3. **Reduce the gap** and repeat the process until the gap size is 1.
4. Finally, the array will be sorted when a regular insertion sort is applied with a gap of 1.

Example:

Given an array: [12, 34, 54, 2, 3]

Initial Gap = 5/2 = 2:

- Compare and sort elements that are 2 positions apart:
  - [12, 34, 54, 2, 3] → [12, 34, 3, 2, 54] (54 and 3 swapped)
  - [12, 34, 3, 2, 54] → [3, 2, 12, 34, 54] (12 and 3 swapped, 34 and 2 swapped)

Next Gap = 2/2 = 1:

- Perform a final insertion sort on the entire array:
  - [3, 2, 12, 34, 54] → [2, 3, 12, 34, 54] (2 and 3 swapped)

After these steps, the array is sorted: [2, 3, 12, 34, 54].

Code:

```
#include <iostream>

using namespace std;

/* function to implement shellSort */

int shell(int a[], int n)

{

  /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */

  for (int interval = n/2; interval > 0; interval /= 2)

  {

    for (int i = interval; i < n; i += 1)

    {

      /* store a[i] to the variable temp and make the ith position empty */

      int temp = a[i];

      int j;

      for (j = i; j >= interval && a[j - interval] > temp; j -= interval)

        a[j] = a[j - interval];



      // put temp (the original a[i]) in its correct position

      a[j] = temp;
```

```
      }
   }
   return 0;
}
void printArr(int a[], int n) /* function to print the array elements */
{
   int i;
   for (i = 0; i < n; i++)
      cout<<a[i]<<" ";
}
int main()
{
   int a[] = { 32, 30, 39, 7, 11, 16, 24, 41 };
   int n = sizeof(a) / sizeof(a[0]);
   cout<<"Before sorting array elements are - \n";
   printArr(a, n);
   shell(a, n);
   cout<<"\nAfter applying shell sort, the array elements are - \n";
   printArr(a, n);
   return 0;
}
```

Key Characteristics:

- **Time Complexity:** Best case: O(n log n), Average/Worst case: between O(n^3/2) and O(n^2) depending on the gap sequence.

COMB SORT

Combo Sort is a hybrid sorting algorithm that combines the principles of Bubble Sort and Shell Sort. It improves on Bubble Sort by using a gap sequence (similar to Shell Sort) to reduce the total number of comparisons and swaps. The key idea is to eliminate turtles (small values near the end of the list) more effectively than Bubble Sort does.

How Combo Sort Works:

1. **Initialize the gap**: Start with a gap equal to the length of the array divided by a shrink factor (typically 1.3).
2. **Compare and swap**: Compare elements that are gap positions apart. If they are out of order, swap them.
3. **Reduce the gap**: After each pass, reduce the gap by dividing it by the shrink factor.
4. **Continue until gap is 1**: Once the gap is reduced to 1, continue with a final pass (similar to Bubble Sort) to ensure the array is fully sorted.

Example:

Given an array: [8, 4, 6, 3, 7, 5, 2, 1]

Initial Gap = 8/1.3 ≈ 6:

- Compare and swap elements 6 positions apart:
  - Compare 8 and 2 → [2, 4, 6, 3, 7, 5, 8, 1]
  - Compare 4 and 1 → [2, 1, 6, 3, 7, 5, 8, 4]

Next Gap = 6/1.3 ≈ 4:

- Compare and swap elements 4 positions apart:
  - Compare 2 and 7 → [2, 1, 6, 3, 7, 5, 8, 4]
  - Compare 1 and 5 → [2, 1, 6, 3, 7, 5, 8, 4]
  - Compare 6 and 8 → [2, 1, 6, 3, 7, 5, 8, 4]

Next Gap = 4/1.3 ≈ 3:

- Continue reducing the gap until the gap becomes 1 and perform the final pass.

The array is fully sorted after the final pass: [1, 2, 3, 4, 5, 6, 7, 8].

Code:

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

int updatedGap(int gap)
{
   // Shrink gap by Shrink factor
   gap = floor(gap/1.3);

   if (gap < 1)
      return 1;
   return gap;
}

// Function to sort array elements using Comb Sort
void combSort(int a[], int n)
{
   int gap = n; /* Initialize gap size equal to the size of array */

   int swapped = 1;

   while (gap != 1 || swapped == 1)
   {
      gap = updatedGap(gap); // find updated gap
      // Initialize swapped as false so that we can
      // check if swap happened or not
```

```
      swapped = 0;

      for (int i = 0; i < n-gap; i++) /* Compare all elements with current gap */
      {
         if (a[i] > a[i+gap]) //swap a[i] with a[i+gap]
         {
            int temp = a[i];
            a[i] = a[i+gap];
            a[i+gap] = temp;
            swapped = 1;
         }
      }
   }
}
void printArr(int a[], int n) /* function to print array elements */
{
   for (int i=0; i<n; i++)
      cout<<a[i]<<" ";
}

int main()
{
   int a[] = {8, 4, 6, 3, 7, 5, 2, 1};
   int n = sizeof(a)/sizeof(a[0]);
   cout<<"Before sorting array elements are - \n";
   printArr(a, n);
   combSort(a, n);
```

```
    cout<<"\nAfter sorting array elements are - \n";
    printArr(a, n);
    return 0;
}
```

Key Characteristics:

- **Time Complexity:** Best case: O(n log n), Average/Worst case: O(n^2).