DAA 1

Mayank Rohilla

August 2023

1 Introduction

- Sorting in Data Structures and Algorithm Analysis (often abbreviated as DAA) refers to arranging a set of data in a particular order, either in increasing (ascending) or decreasing (descending) order.

1.1 Types of Sorting

- Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort

2 Bubble Sort Algorithm

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order

2.1 Algorithm

- For i from 0 to n-2 (where n is the number of elements)
  For j from 0 to n-i-2
  If element[j] ¿ element[j+1]
  Swap element[j] with element[j+1]

2.2 Time Complexity

- **Best-case:O(n)**
- This is when the Average-case:O(n^2) but the algorithm still takes a pass through the list to make sure no swaps are needed.

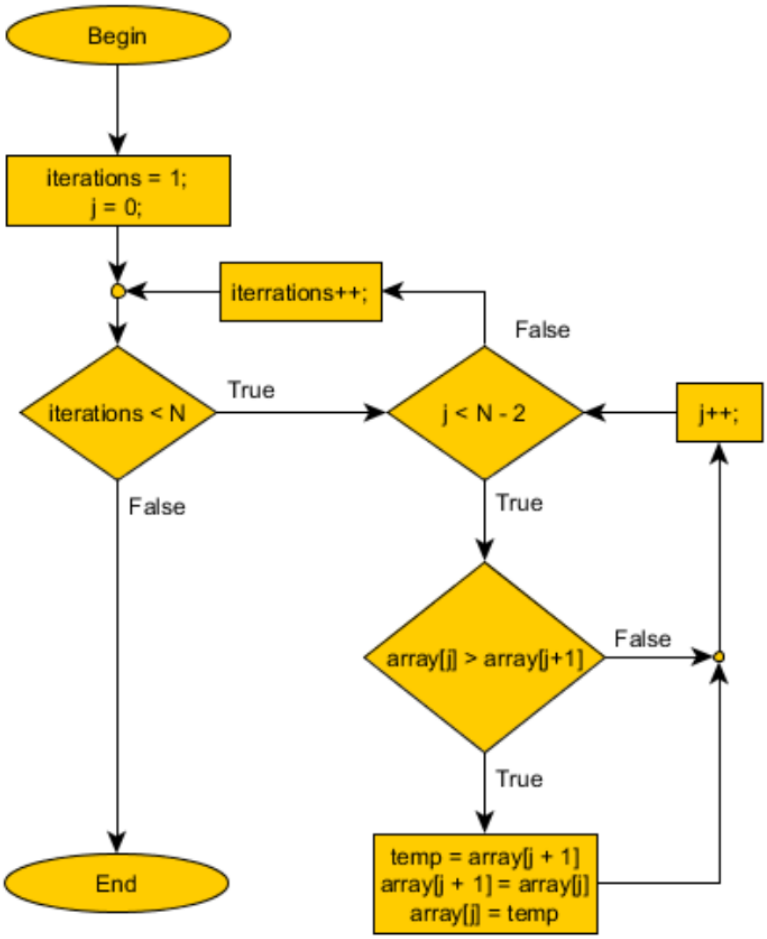
- **Average-case:O(n^2)**
- On average, the algorithm will need to make a number of comparisons and swaps that are proportional to the square of the number of elements.

- **Worst-case:O(n^2)**
- This is when the list is reversed. The algorithm will need to do the maximum number of swaps for every element.

2.3 Space Complexity

- **Space Complexity:O(1)**
- Bubble sort is an in-place sorting algorithm, which means it doesn't require any additional storage (apart from a temporary variable for swapping) regardless of the size of the input.

2.4 Flowchart



2.5 Code

```
#include <stdio.h>
void bubble_sort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubble_sort(arr, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;

}
```

2.6 Result

```
PS E:\WEB DEV> cd "e:\WEB DEV\" ; if ($?) { gcc new.c -o new } ; i
Sorted array: 11 12 22 25 34 64 90
PS E:\WEB DEV>
```

3 Selection Sort Algorithm

- Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list

3.1 Algorithm

- SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
END OF LOOP
Step 4: EXIT

SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL ¿ arr[j])
SET SMALL = arr[j]
SET pos = j
END OF if
END OF LOOP
Step 4: RETURN pos

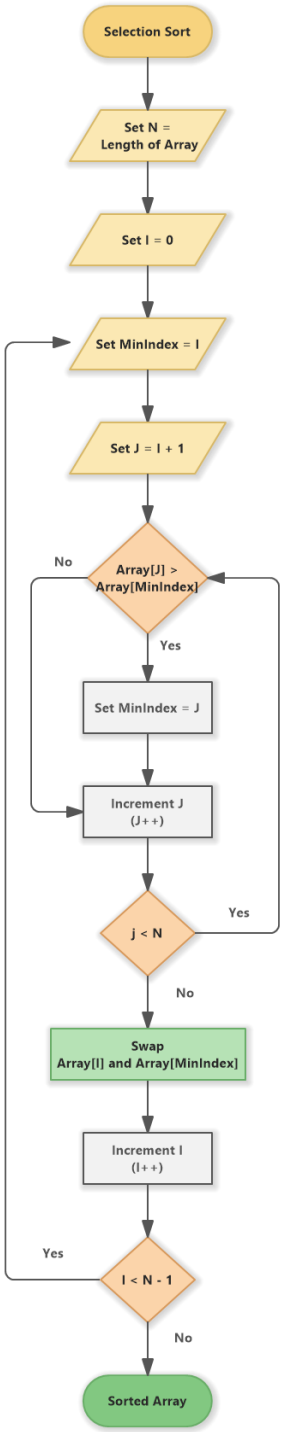
3.2 Time Complexity

- **Best Case: O(n^2)**
- Even if the list is sorted, it goes through the entire list for each element.
- **Average Case: O(n^2)**
- It generally requires checking through the list for each element.
- **Worst Case: O(n^2)**
- Even if the list is in reverse order, the checks remain similar.

3.3 Space Complexity

- Worst Case:O(1) - This means that the space (or memory) used does not increase with the size of the input list. Selection Sort is an in-place sorting algorithm, so it only uses a constant amount of extra space (e.g., for temporary variables)

3.4 Flowchart



3.5 Code

```
#include <stdio.h>
int main() {
   int arr[10]={6,12,0,18,11,99,55,45,34,2};
   int n=10;
   int i, j, position, swap;
   for (i = 0; i < (n - 1); i++) {
      position = i;
      for (j = i + 1; j < n; j++) {
         if (arr[position] > arr[j])
            position = j;
      }
      if (position != i) {
         swap = arr[i];
         arr[i] = arr[position];
         arr[position] = swap;
      }
   }
   for (i = 0; i < n; i++)
      printf("%d\t", arr[i]);
   return 0;
}
```

3.6 Result

```
PS E:\WEB DEV> cd "e:\WEB DEV\" ; if ($?) { gcc new.c
0       2       6       11      12      18      34
```

4 Insertion Sort Algorithm

- Insertion sort is an algorithm used to sort a collection of elements in ascending or descending order. The basic idea behind the algorithm is to divide the list into two parts: a sorted part and an unsorted part

4.1 Algorithm

```
insertionSort(array)
mark first element as sorted
for each unsorted element X
'extract' the element X
for j <- lastSortedIndex down to 0
if current element j > X
move sorted element to the right by 1
break loop and insert X here
end insertionSort
```

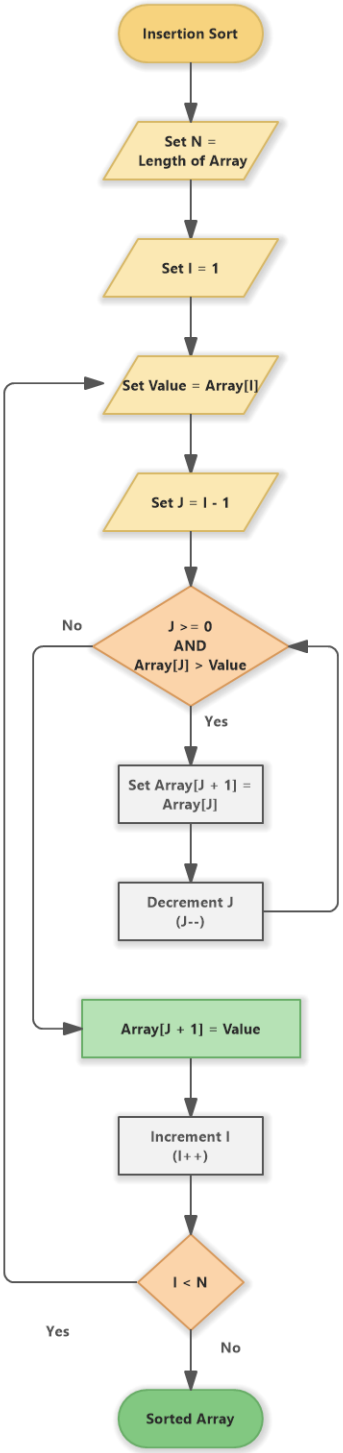
4.2 Time Complexity

- **Best Case: O(n)**
- When the input list is already sorted
- **Average case: O(n^2)**
- Each element is compared with all the other elements before it in the list, leading to quadratic time.
- **Worst Case:O(n^2)**
- Every element will be compared with all the elements before it in the list.

4.3 Space Complexity

- **Space Complexity: O(1)**
- Only a constant amount of extra space (like temporary variables) is used, regardless of the input list size

4.4 Flow Chart



4.5 Code

```c
    // Insertion sort in C

#include <stdio.h>

// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", array[i]);
  }
  printf("\n");
}

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

    // Compare key with each element on the left of it until an element smaller
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

// Driver code
int main() {
  int data[] = {9, 5, 1, 4, 3};
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  printf("Sorted array in ascending order:\n");
  printArray(data, size);
}
```

4.6 Result

```
PS E:\WEB DEV> cd "e:\WEB DEV\" ; if ($?) { gcc new.c -o new } ; i
Sorted array in ascending order:
1 3 4 5 9
```

5 Merge Sort Algorithm

- In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

5.1 Algorithm

```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

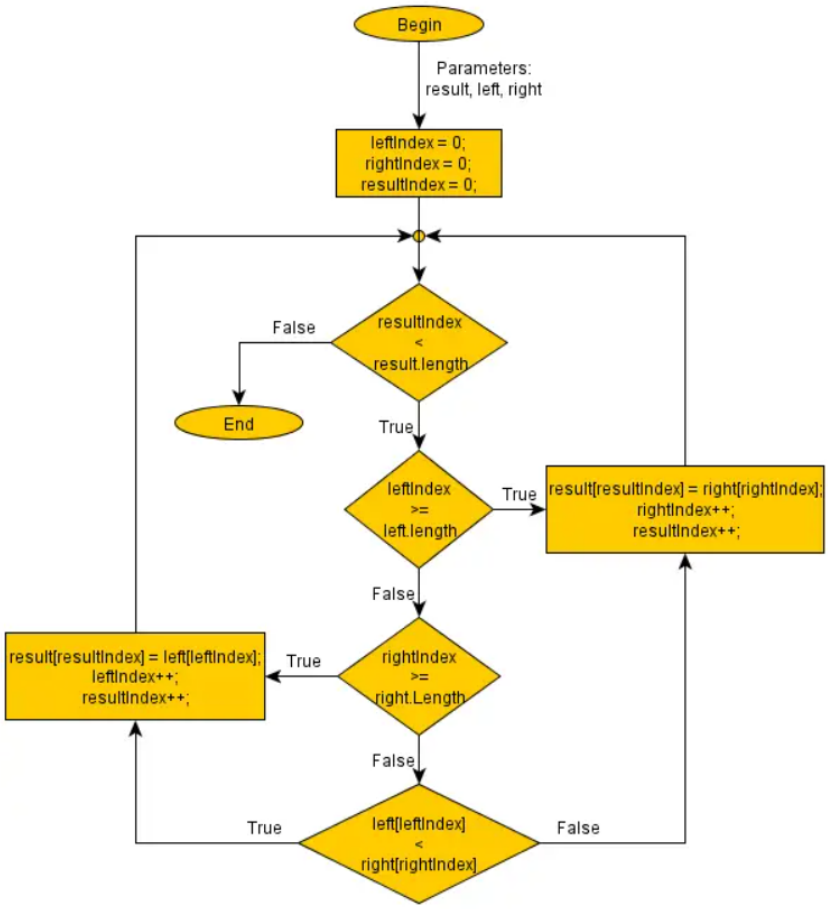
5.2 Time Complexity

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n)

5.3 Space Complexity

- Worst Case:O(n)

5.4 Flow Chart



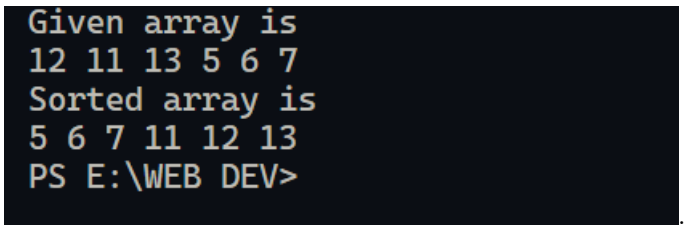
5.5 Code

```
    #include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // create temp arrays
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
```

```
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main() {
```

```
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");

    mergeSort(arr, 0, arr_size - 1);

    printf("Sorted array is \n");
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

5.6 Result



6 Quick Sort Algorithm

- Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm 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.

6.1 Algorithm

```
QUICKSORT (array A, start, end)
{
  if (start < end)
  {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
```

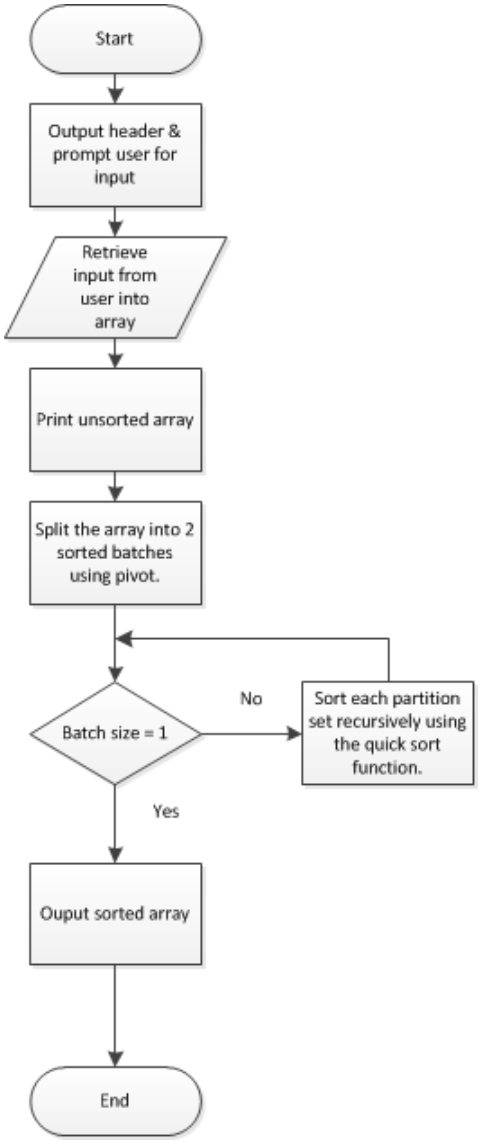
6.2 Time Complexity

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

6.3 Space Complexity

- Worst Case:O(n)
- Average Case:O(log n)
- Best Case:O(log n)

6.4 Flow Chart



6.5 Code

```
    #include <stdio.h>

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

int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // choosing the last element as the pivot
    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(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() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

6.6 Result

```
Sorted array: 1 5 7 8 9 10
PS E:\WEB DEV>
```

7 Heap Sort Algorithm

- Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

7.1 Algorithm

```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
End
```

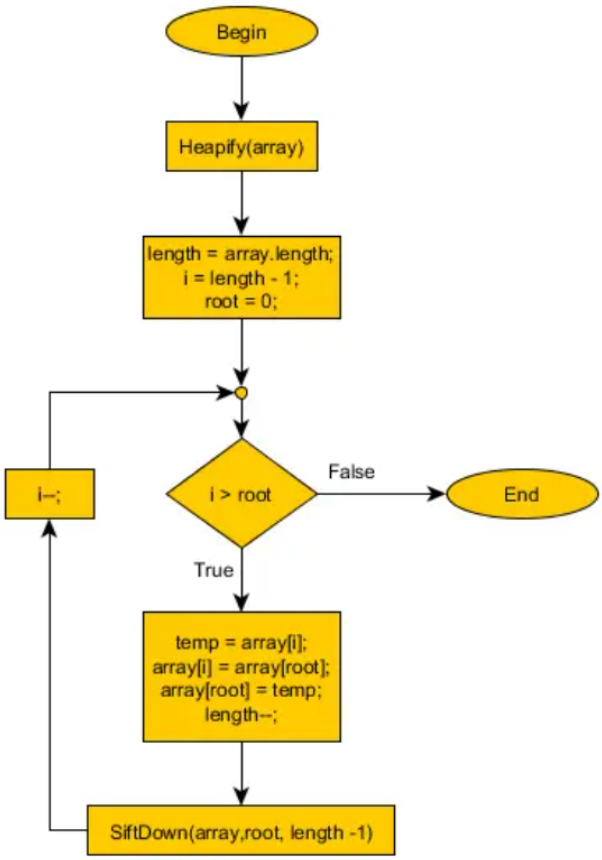
7.2 Time Complexity

- Best Case: O(n log n)
- Average Case: O(n log n)
- Worst Case: O(n log n)

7.3 Space Complexity

- Worst Case:O(1)

7.4 Flow Chart



7.5 Code

```
  #include <stdio.h>

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

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
```

```
    int right = 2 * i + 2;

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

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

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

void heapSort(int arr[], int n) {
    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);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    printf("Sorted array is: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

7.6 Result

```
Sorted array is:
5 6 7 11 12 13
PS E:\WEB DEV>
```