Module 2 - Analysis of Basic Algorithms

* Some common Sorting algorithms

① Insertion Sort

This is a common sorting algorithm that works by iterating through an array and inserts each element at its correct position.

→ Algorithm for Insertion Sort.

a) Start with the second element in array

b) Compare the second element with the first element, if second element is smaller, swap it.

c) Use this tactic for all elements in the array, if the coming element is greater than the existing element, places it to the right, else check the previous element

d) Keep shifting to the right till the correct position for the element is reached.

→ Pseudocode for Insertion Sort

```
insertionSort(array A)

  for i=1 to i = length(A)-1
      key = A[i]
      j= i-1
      while j>0 and A[j] > key
          A[j+1] = A[j]
          j--
      end while
      A[j+1] = key
  end for
end function
```

→ Program for Insertion Sort (In C++)

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for(i=0; i<n-1; i++)
    {
        key = arr[i];
        j = i-1;
        while(j>=0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

→ Analysis of Insertion sort

Total cost = No. of comparisons
+ No. of swaps.

a) Best case - The best case would be a sorted array.

Eg - [1, 2, 3, 4, 5]

No. of comparisons = O(n)
No. of swaps = O(1)

∴ Time complexity = O(n) {In best case}

b) Worst case - The worst case would be a reversed array.

Eg - [5, 4, 3, 2, 1]

∴ Cost = No. of comparisons
+ No. of swaps

No. of comparisons = O(n)

No. of swaps = O(n*(n-1))
= O(n²) {in worst case}

c) Average case - The average case runs swap $n/2$ times

$\therefore O(n^2)$ {for avg. case}

---

② Selection Sort

It is a simple sorting algorithm that works by repeatedly finding the minimum element in an unsorted array and placing it at the beginning of the array.

→ Algorithm of Selection Sort.

① Set the 1st element as the minimum value

② Iterate through the array, starting from the second element

③ Compare each element with the current minimum value, If element is smaller than the current value then update the current minimum value.

④ After iterating through the array, swap the minimum value with the first element of the array.

⑤ Repeat process for multiple cycles.

→ & Pseudocode for selection sort.

```
selection sort (array A)
    for {i=0 &  to  i = length(A) - 1
      min Index = i
      for j = i-1  to  j = length(A)
          if  A[j] < A[minIndex]
              minIndex = j
          endif
          Swap  A[i]  and  A[minIndex]
      end for
  end function.
```

→ Program Implementation (C++)

```
void  selectionsort (int arr[] , int n)
{
   for (int i=0 ; i<n-1; i++)
   {
      int min = i ;
      for (int j = i-1 ; j<n ; j++)
       {
           if (arr[j] < arr [min])
            {
                min = j ;
            }
       }
      swap ( A[i] , A[min])
   }
}
```

→ Analysis of selection sort

a) Best case :- The Best case for selection sort would be a sorted array.

Let A = [1, 2, 3, 4, 5].

No. of comparisons → $O(n^2)$
No. of swaps → $O(1)$

So Time complexity is $O(n^2)$

b) Worst case - The worst case for selection sort would be a reverse array.

Let A = [5, 4, 3, 2, 1]

No. of comparisons = $O(n^2)$
No. of swaps = $O(n^2)$

∴ Worst case T.C = $O(n^2)$

c) Avg. case - The avg. case would be.

Each iteration will have $(n-1)/2$ comparisons.

∴ Avg. case T.C = $O(n^2)$

* Sorting In Linear Time

→ Usually sorting algos have worst-case as Quadratic running time {$O(n^2)$} so to solve in linear time we can use additional space.

① Counting Sort {For a given Range}

→ It is one of the sorting algorithms which sorts an array in linear time.

→ The basic idea of count sort is to create a temporary array called the counting array or frequency array which stores how many times a no. has occurred.

→ Algorithm for count sort.

① Find minimum and maximum of the array.

② Create a temporary array of size {min-max} and initialize all elements to 0.

③ Traverse the input array and update the counting array based on how many times an element occurred.

④ Modify the counting array by adding prev. count to the next count to determine the position of each element in the array.

⑤ Traverse input array in reverse and use counting array to place each element in the correct position.

⑥ output array is sorted.

→ Pseudocode for count sort.

```
countingSort(arr, n, min, max)
    range = max - min + 1
    count[range] = {0}

    for i=0 to n-1
    count[arr[i] - min] += 1

    for i=1 to range-1
        count[i] += count[i-1]

    SortedArray = [0] * len(array)
    for element in array:
        sortedarray[count[element] - 1] = element
        coun[element] -= 1

    return sortedArray.
```

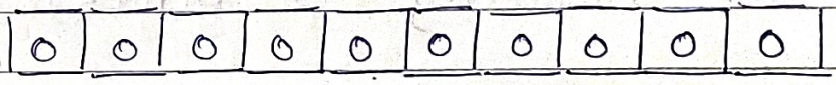
→ Program Implementation (C++)

```
void countingSort (int arr[], int n, int max)
{
    int count[max+1] = {0};

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

    for (int i=1; i<=max; i++)
    {
        count[i] = count[i] + count[i-1];
    }

    int sortedarr[n];

    for (int i=n-1; i>=0; i--)
    {
        sortedarr[count[arr[i]-1]] = arr[i];
        count[arr[i]] --;
    }
}
```

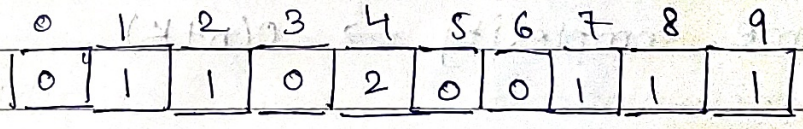
Example - [2, 9, 7, 4, 1, 8, 4]

① Find max of the array {9}

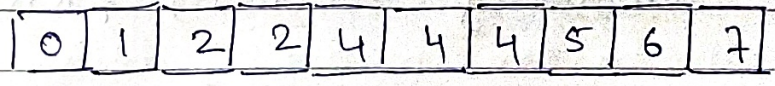
② Make a new array of size max+1 {10}



③ Update the new array with freq. of each element.

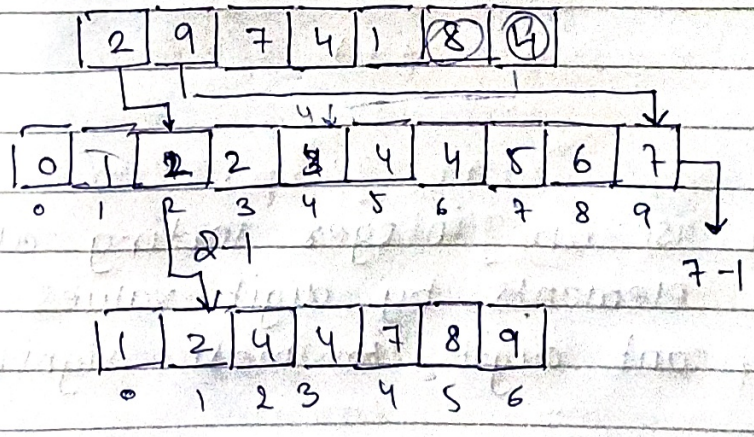


④ Store cumulative sum of count array elements.



{New count array}

⑤ Place the elements in the sorted array



* Analysis of Count sort.

① Best case - Occurs when no sorting is required and array is already sorted.

Time complexity → O(n+K)

② Worst case - occurs when the array is in reverse order

Time complexity → O(n+K)

③ Avg. case - occurs when array is in jumbled order.

Time complexity → O(n+K) ∀ K is the range

One disadvantage of Count sort is that there are lots of unused spaces in the array causing inefficiency of space usage.

* Radix Sort

→ Radix sort is an integer sorting algorithm that sorts elements by digit values from least significant digit to most significant digit.

→ Algorithm for Radix Sort

① Find max value in the array to determine the number of digits in the largest element.

② Initialize exponent to 1 for L.S.D.

③ Repeat the following process from L.S.D to M.S.D.

a) Create a count array.
b) Count the number of occurence of each digit
c) Modify count array to store position of digit
d) Create the new sorted array
e) Multiply the exponent by 10 to the next digit.

④ Return the sorted array.

→ Pseudocode for Radix Sort

```
radixSort (array)
    max = max (array)
    exponent = 1
    output = [0] * len (array)

    While max/exponent > 0:

        count = [0] * 10.
        for element in array
            digit = (element/exponent) % 10.
            count[digit] += 1.
```

```
for i in range (1,10)
    count[i] += count[i-1]

for i in range (len(array)-1, -1)
    digit = (array[i]/exponent) % 10
    output[count[digit]-1] = array[i]
    count[digit] -= 1

for i in range len(array)
    array[i] = output[i]

exponent *= 10.
```

→ Program Implementation (C++)

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

    for (int i=0; i<n; i++)
    {
        int digit = (arr[i]/exp) % 10;
        count[digit]++;
    }
    for (int i=1; i<10; i++)
    {
        count[i] += count[i-1];
    }
```

```
    for (int i = 9; i >= 0; i--)
    {
        int digit = (arr[i]/exp) % 10;
        output[count[digit]-1] = arr[i];
        count[digit]--;
    } }.
Void radixSort (int arr[], int n)
{
    int max = arr[0];

    for (int i=1; i<n; i++)
    {
        if (arr[i] > max)
            max = arr[i]
    }
    for (int exp = 1; max/exp > 0; exp *= 10)
    {
        countSort (arr, n, exp);

    }.

}.
```

→ Analysis of Radix Sort

Counting sort within Radix sort is O(n)

∴ Best case → O(n+K) {K is the range}
Worst case → O(nK)
Avg. case → O(nK)

Best case - When the array is already sorted.

Avg. case - When the array is jumbled

Worst case - When elements are in descending order.

* Divide and Conquer Algorithm.

→ Divide and Conquer is a problem-solving method used for dividing the problems into smaller sub-problems, solving each problem seperately and then merging results.

Some examples are.

① Quick Sort
② Merge Sort.

* Quick Sort.

→ Quick Sort is a divide and conquer algorithm used to sort an array by dividing the array into smaller parts based on the pivot element. and continues division of array into sub-arrays.

→ Algorithm of Quick Sort

① Select a pivot element from the array

② Partition the array around the pivot by rearranging the array such that elements smaller than pivot element are on left and others on the right side.

③ Recursively sort the sub-arrays on each side of the pivot element.

④ Combine the sorted sub-arrays into one array.

→ Pseudocode for Quick Sort

```
quickSort ( arr, low , high)
    if low < high
      pivotIndex = partition (arr, low , high)
      quickSort (arr , low , pivotIndex - 1)
      quickSort (arr, pivotIndex+1, high)

partition (arr, low, high)
    pivot = arr [high]
    i = low - 1
    for j = low to high - 1
        if arr[j] <= pivot
            i = i+1
            swap (arr [i], arr[j]
    swap [arr[i+1], arr[high])
    return i+1.
```

* Program Implementation (C++)

```
int partition (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 (int arr[], int low, int high)
{
    if (low < high)
    {
        int pivotIndex = partition (arr, low, high);
        quickSort ( arr, low, pivotIndex - 1);
        quickSort ( arr, pivotIndex + 1, high);
    }
}
```

Example - Low

[24, 9, 29, 14, 19, 27]

↑ pivot ↑ High.

The partition function will arrange the numbers as follows-

Left Arr. Right Arr.

19, 9, 14, 24, 29, 27.

These two will be partitioned recursively.

→ Analysis of Quick Sort.

① Best case - When the pivot element is already in the center, that is the best case for quicksort.

Time complexity = $O(n \log n)$

Total cost = cost of recursion + cost of iteration.
$(n \log n)$ (n)

$\therefore O(n \log n)$

② Worst case - When the array is sorted in either ascending or descending order.

Time complexity = $O(n^2)$

The tree is skewed in a direction

so $\frac{n(n-1)}{2} \rightarrow O(n^2)$

③ Avg - case - When the array is in jumbled order.

Time Complexity = $O(n \log n)$

---

* Merge Sort.

→ Merge Sort is a divide and conquer algorithm for sorting. It breaks down the unsorted array into sub-arrays till each array has a single element left and merges after comparison.

→ Algorithm for Merge Sort

① Divide the unsorted array into n-subarrays each containing 1 element.

② Repeatedly merge adjacent sub-arrays to produce new sorted sub-arrays until there is only one sorted array left.

a) Compare the first element of adjacent sub-arrays and select the smaller one.

b) Place the smaller element in the sorted sub-array.

c) Move the pointer from the selected subarray to the next element.

d) Repeat steps recursively

→ Pseudocode for merge sort

```
mergesort (array A)

  if length(A) ≤ 1
     return A

  mid = length(A)/2
  left = first half
  right = second half.

  left = mergesort (left)
  right = mergesort (right)

  return Merge (left, right)

Merge ( left, right)
   result = []

  While length(left) > 0 and length(right) > 0
     if first element of left ≤ first element of
                                    right
        append first element of left to result
        remove first element from left.
```

else
append first element of right to result
remove first element from right.

Append the remaining elements

While length(left) > 0
append to result from left
remove appended element

While length(right) > 0
append to result from right
remove appended element.

return result

→ Program Implementation (C++)

```
void merge( int arr[], int left, int mid, int right
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    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+j+1]; }
```

```
int i=0;
int j=0;
int K= left;

While (i<n2 and 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(int arr[], int left, int right)
{
    if (left >= right)
    {
        return;
    }

    int mid = (left+right)/2

    mergeSort(arr, left, mid);
    mergeSort(arr, mid+1, right);
    merge(arr, left, mid, right);
}
```

→ Analysis of Merge sort.

The overall recurrence relation for Merge sort is.

$$T(n) = 2T(n/2) + n.$$

On solving we get $O(n \log n)$

This is for all cases of Merge sort.

* Strassen's Multiplication Matrix.

→ Strassen's algorithm is a divide and conqu algorithm for multiplying two matrices. It wa to lower the time complexity of the algorithm.

→ It recursively breaks down nigh order matrices into smaller submatrices and then combine the smaller submatrices into one matrix.

→ Algorithm of Strassen's Algorithm.

① Divide each input matrix A and B into four submatrices.

$$A = [[A_{11}], [A_{12}], [A_{13}], [A_{14}]] \; ; \; B = [[B_{11}], [B_{12}], [B$$

② Compute these 7 products.

$$P_1 = [A_{11} * (B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12}) * B_{22}$$
$$P_3 = B_{11} * (A_{21} + A_{22})$$
$$P_4 = A_{22} * (B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_2) * (B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21}) * (B_{11} + B_{12})$$

③ Combine the submatrices using the formula:

$C_{11} = P_5 + P_4 - P_2 + P_6$

$C_{12} = P_1 + P_2$

$C_{21} = P_3 + P_4$

$C_{22} = P_5 + P_1 - P_3 - P_7$.

④ Return final matrix.

→ Analysis of Strassen's Multiplication Algorithm.

Time complexity = $O(n^{\log_2 7})$

$T(n) = 7T(n/2) + n^2$

$\therefore O(n^{2.81})$

This is faster than the standard algorithm which requires 3 for loops and is of time complexity $O(n^3)$