



DATA STRUCTURES AND ALGORITHMS
COMP2421
Project-4-

Name : Mariam Turk

ID : 1211115

Section : 1

Introduction : Dr . Ahmed Abusnaina

Abstract :

in this project I will talk about 3 types of Sort Algorithm , ``Quick Sort , Counting Sort and Comb Sort`` ,and I will explain there Algorithms and sodocode , an example for Each one ,put the code of them , and calculate the time and space complexity (if data in random or ascending or descending sort) .

Then Know what is Dynamic Programming , when we use it , advantages and disadvantages of Dynamic Programming , show three problem that it can solve , and talk about the Fibonacci Revised , how it before Dynamic Programming and after Dynamic Programming .

Contents

Abstract :	2
1. Sorting :	4
1.1 Quick Sort :	4
1.2 Counting Sort :	22
1.3 Comb Sort :	34
2. Dynamic Programming	42
2.1 What is Dynamic Programming?	42
2.2 three problems that can be solved using DynamicProgramming:	43
2.3 example of a code that solves a problem using DynamicProgramming and without using DynamicProgramming	43
3. References	48

Figures

Figure 1 : How Quick Sort works	4
Figure 2 :code of Partition function	17
Figure 3:code of QuickSort function	18
Figure 4 : the output of QuickSort code	18
Figure 5 : worst case time complexity in q_sort	21
Figure 6 : Counting Sort	22
Figure 7 : function of Counting Sort	30
Figure 8 : the out put Counting Sort code	31
Figure 9 : Comb Sort code	40
Figure 10 : Dynamic Programming	42
Figure 11 : Fibonacci Revised	44
Figure 12 : Fibonacci Revised for 5	45
Figure 13 : Fibonacci Revised for 5 after DB	47

1. Sorting :

1.1 Quick Sort :

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 . [1]

It partition the array in a way such that all the elements less than the pivot (key) will be to the left of the pivot and all elements greater than pivot will be to the right side , if the element = pivot it can be to the left or to the right according to the logic of the Quick Sort Code . [2]

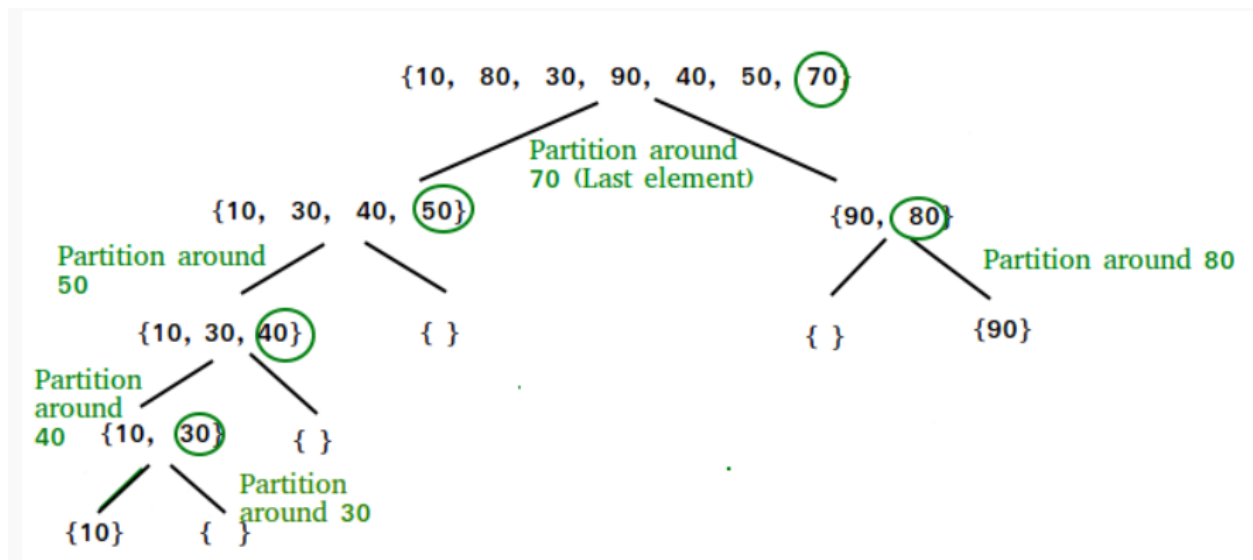


Figure 1 : How Quick Sort works

Advantages of Quick Sort:

- 1- It is a divide-and-conquer algorithm that makes it easier to solve problems. [1]
- 2- It is efficient on large data sets. [1]
- 3- It has a low overhead, as it only requires a small amount of memory to function. [1]

Disadvantages of Quick Sort:

- 1- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
- 2- It is not a good choice for small data sets.
- 3- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Quick Sort algorithm :

- **Step 1** - Consider the first element of array as **pivot** (Element at first position in the list).
- **Step 2** - Define two variables i and j. Set i (start index in arr) and j (end index in arr) .
- **Step 3** - Increment i until $arr[i] > pivot$ then stop.
- **Step 4** - Decrement j until $arr[j] < pivot$ then stop.
- **Step 5** - If $i < j$ then swap $arr[i]$ and $arr[j]$.
- **Step 6** - Repeat steps 3,4 & 5 until $i > j$.
- **Step 7** - swap the pivot element with $arr[j]$ element. [3]

```
partition( Array arr , first_i , last_i ) {  
  
    pivot = arr[first_i]  
  
    start = first_i  
  
    end = last_i  
  
    while (start < end )  
  
        while ( arr[start] <= pivot && start < last_i)  
  
            start++  
  
        while ( arr[end ] > pivot)  
  
            end --
```

```

if ( start < end)

swap(arr[start] , arr[end])

if(i>j)

swap(arr[first_i] , arr[end])

return end ;

}

```

```

QuickSort( Array arr , first_i , last_i ) {

If ( first_i < last_i )

Loc = partition(arr , first_i , last_i )

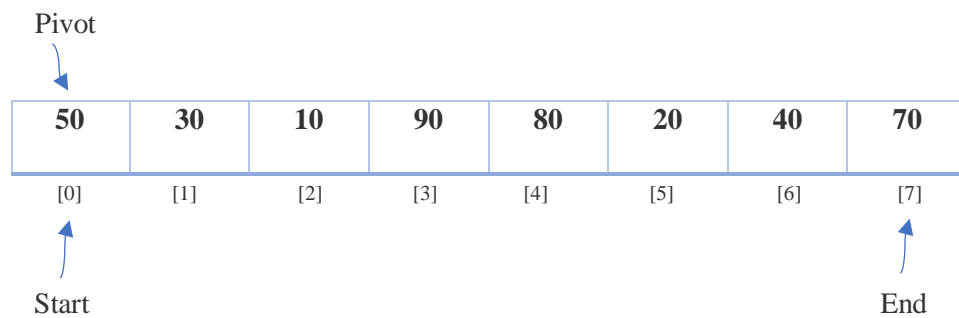
QuickSort(arr , first_i , loc-1)

QuickSort(arr , loc+1 , last_i )

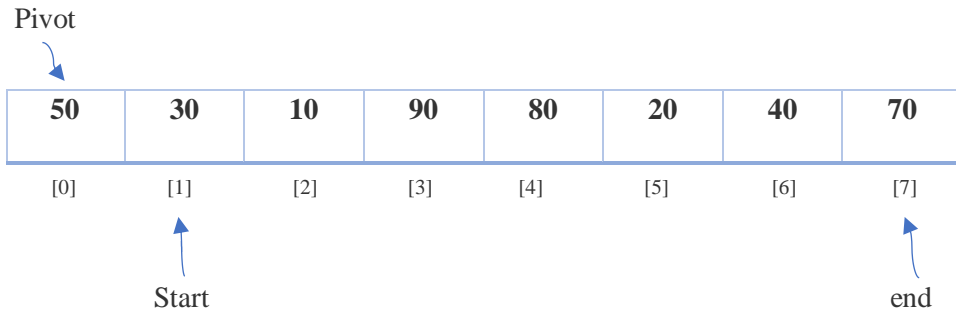
}

```

Example of Quick Sort :

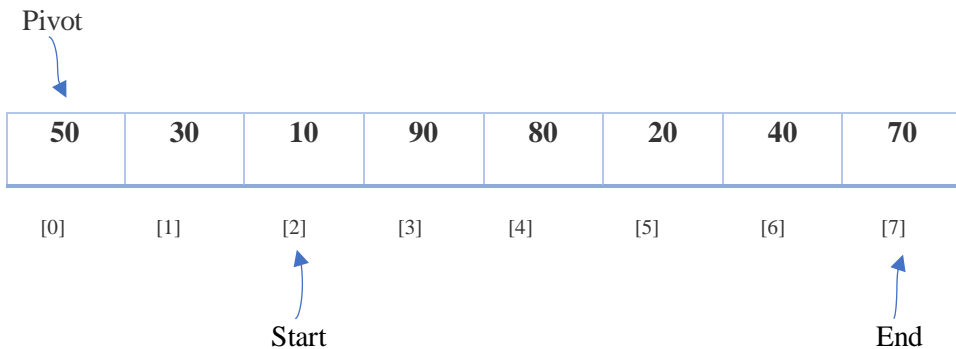


Here the $A[\text{Start}] = \text{Pivot}$ ($50 = 50$) So by the Quick Sort algorithm the Start will increase , $\text{Start} = 0$, $\text{Start}++ \rightarrow 0+1 = 1$, it will go to index 1 .



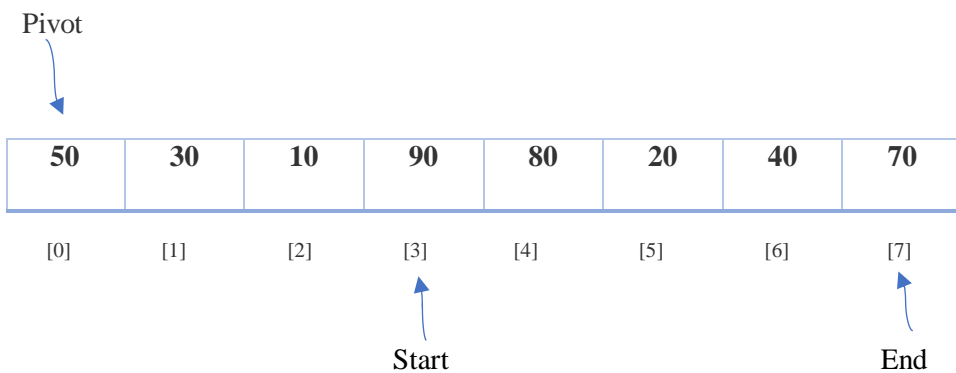
After increase Start we will see if $A[\text{Start}] < \text{Pivot}$, $30 < 50$ this is true so we will increase Start again

Start $\rightarrow 1+1=2$, it will go to index 2.

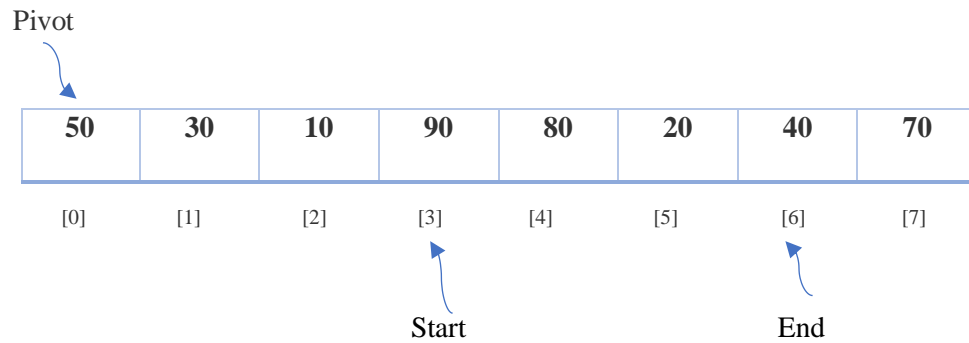


After increase Start we will see if $A[\text{Start}] < \text{Pivot}$, $10 < 50$ this is true so we will increase Start again

Start $\rightarrow 2+1=3$, it will go to index 3.

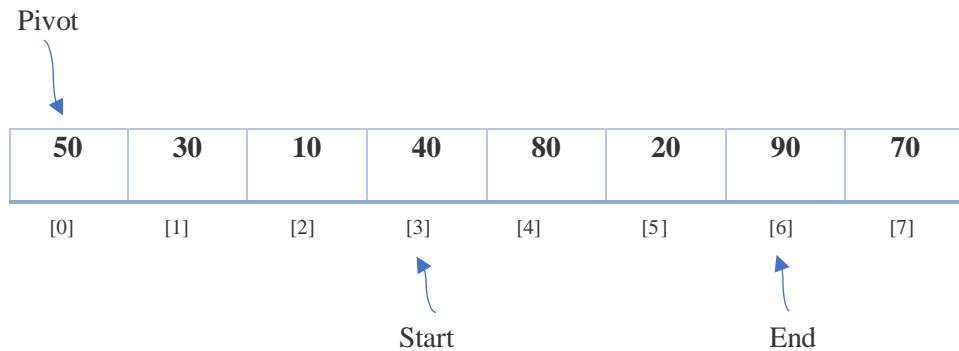


Now $A[\text{Start}]$ is not less than Pivot, $90 > 50$ so the increase of Start will stop, and then check End, if $A[\text{End}] > \text{pivot}$, End --, $A[\text{End}] = 70$ and $\text{Pivot} = 50$, $70 > 50$, so we will decrease $\text{End} = 7 \rightarrow 7 - 1 = 6$, it will go to index 6.

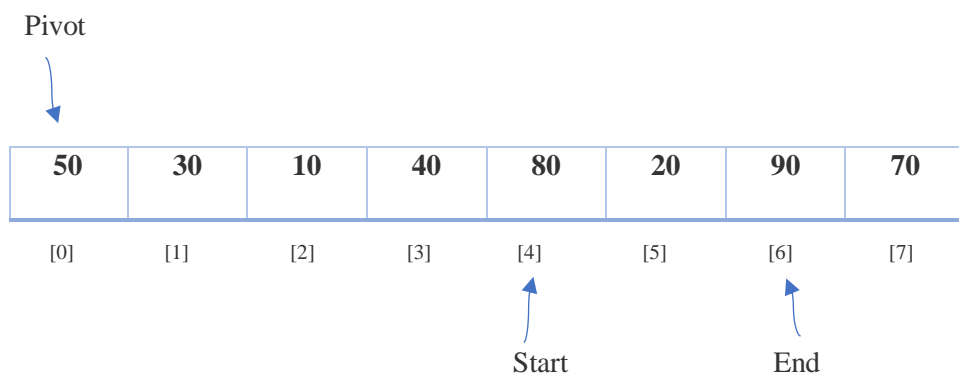


Now we will check again if $A[\text{End}] > \text{Pivot}$, $A[\text{End}] = 40$, but $40 < 50$ so this is false, now check if $\text{Start} < \text{End} \rightarrow \text{Start} = 3$, $\text{End} = 6$, $3 < 6$, so swap ($A[\text{Start}]$, $A[\text{End}]$).

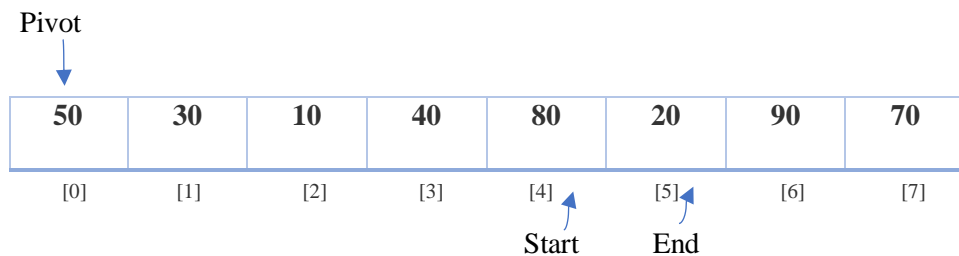
40 will be in index = 3, 90 will be in index = 6,



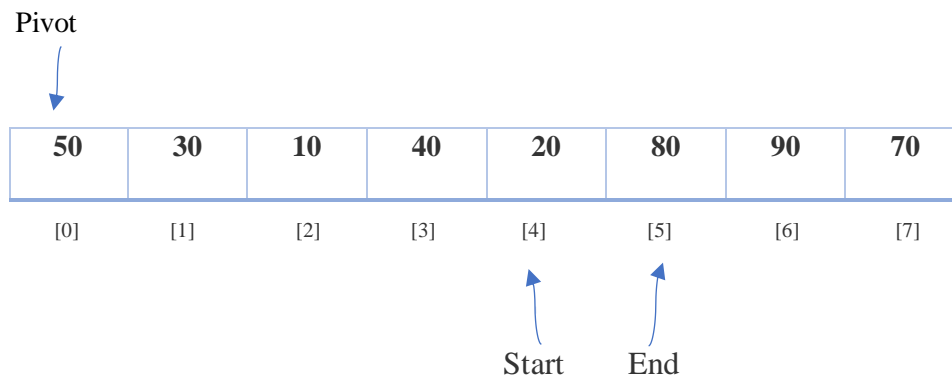
After swap we will check again the A [Start] if it less then Pivot , so A [Start] = 40 , Pivot = 50 , $40 < 50$, Start = 3 we will increase it $\rightarrow 3 + 1 = 4$, t well go to index 4 .



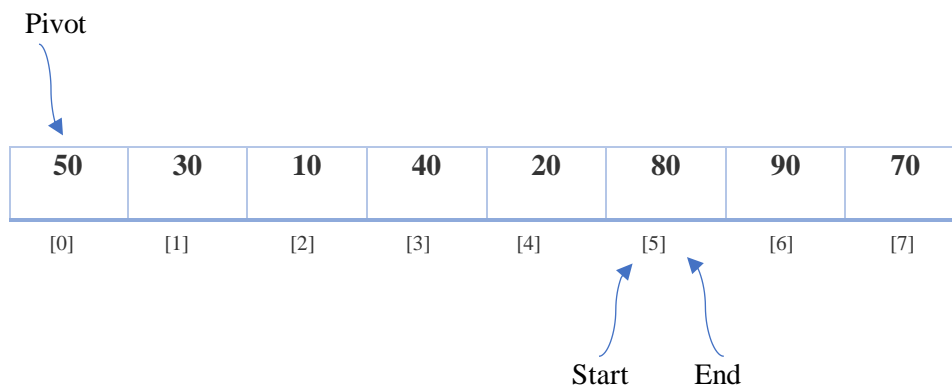
Check the A [Start] if it less than Pivot , A [Start] = 80 , and $80 > 50$ so stop increasing and check the A [End] if it greater than Pivot . $90 > 80$ so we will decrease End , End = 6 $\rightarrow 6 - 1 = 5$, End will go to index 5 .



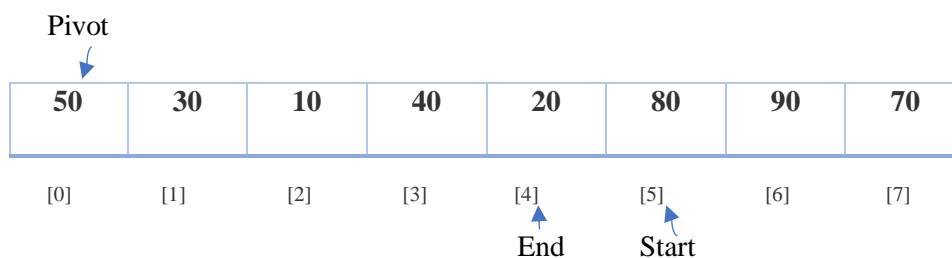
Check A [End] again , A [End] = 20 , and Pivot = 50 , but A [End] not greater than Pivot , $20 < 50$, so stop here and check if $Start < End$, $Start = 4$, $End = 5$, $4 < 5$ so it is true , swap (80 , 20) .



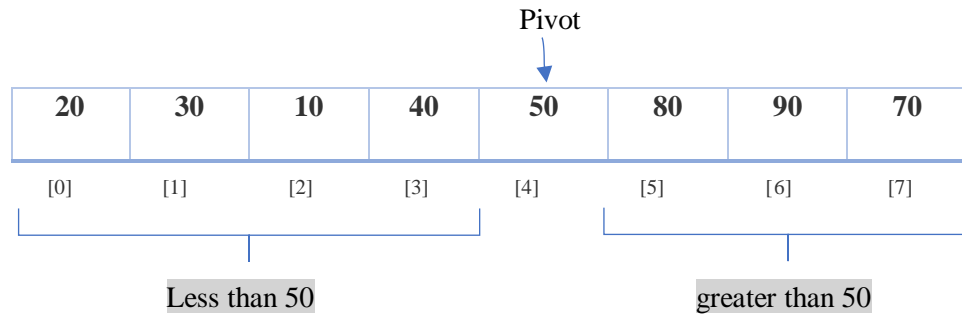
Check if A [Start] less than Pivot , A [Start] = 20 , and Pivot = 50 , $20 < 50$, so increase State , $Start++ \rightarrow 4 + 1 = 5$, it well go to index 5 ,



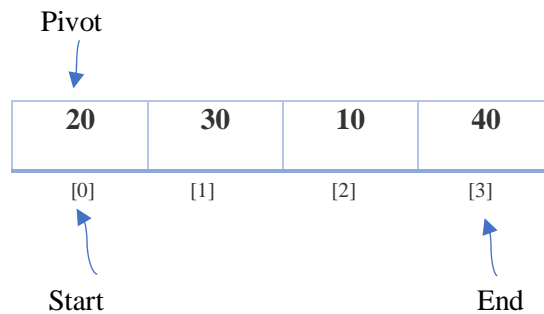
Check if A [Start] less than Pivot , A [Start] = 80 , and Pivot = 50 , $80 > 50$ so it is false stop , check the A [End] if it grater than Pivot , A [End] = 80 , $80 > 50$, so decrease End , $End -- \rightarrow 5 - 1 = 4$, it well go to index 4 .



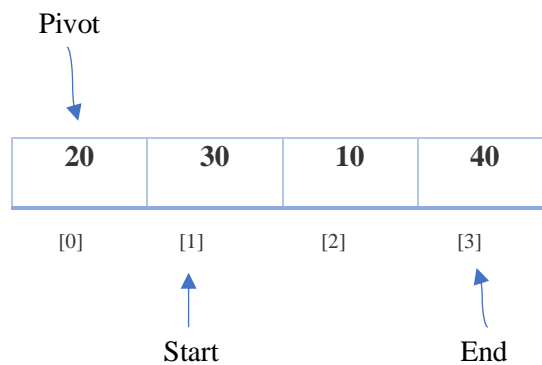
A [End] = 20 , and Pivot = 40 , $20 < 40$ so A [End] is not greater than Pivot , check if Start < End , Start = 5 , End = 4 . $5 > 4$ is false , so swap (Pivot , A [End]) .



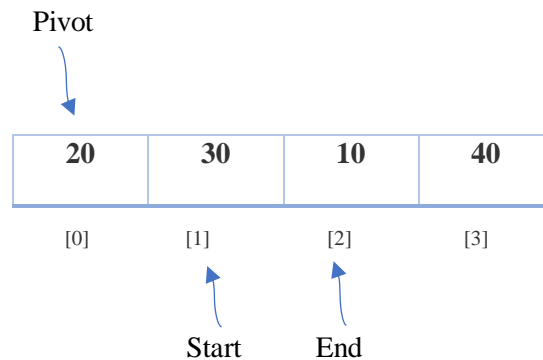
Now make sort in the sub array , first I will start with the less than 50 :



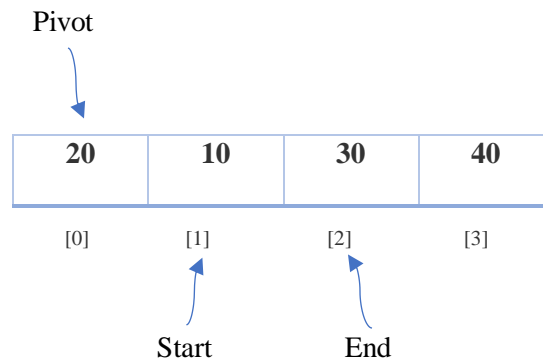
A [Start] = Pivot so increase Start . Start ++ → $0 + 1 = 1$, it will go to index 1



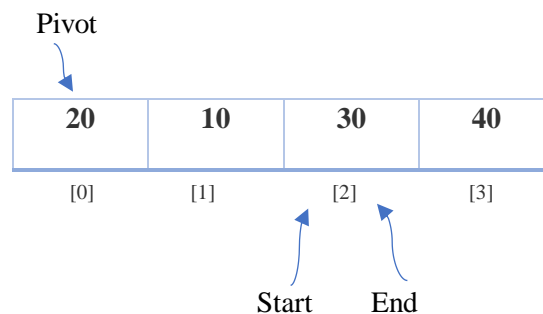
Check if A [Start] is less than Pivot , A [Start] = 30 , and Pivot = 20 , $30 > 20$ so it is false , now check if A [End] is greater than Pivot , A [End] = 40 , and Pivot = 20 , $40 > 20$ so it is true , decrease End , End - $\rightarrow 3 - 1 = 2$, it will go to index 2 .



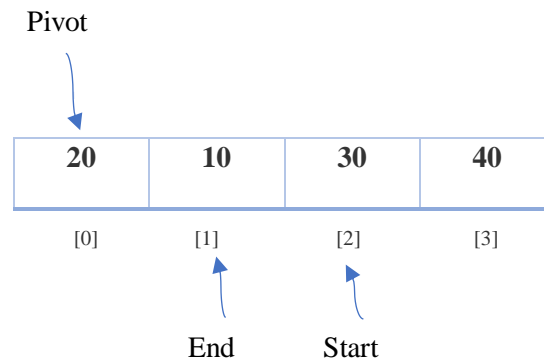
Check if A [Start] is less than Pivot , A [Start] = 10 , and Pivot = 20 , $10 < 20$, so it is false , now check if Start < End , Start = 1 , End = 2 , $1 < 2$ it is true , swap (30 , 10) .



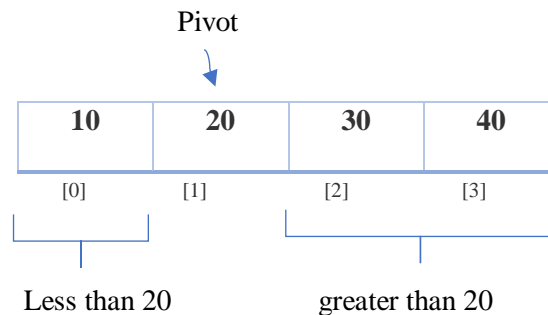
Check if A [Start] is less than Pivot , A [Start] = 10 , and , Pivot = 20 , $10 < 20$ so it is true , so increase Start , Start ++ $\rightarrow 1 + 1 = 2$, so it will go to index 2 .



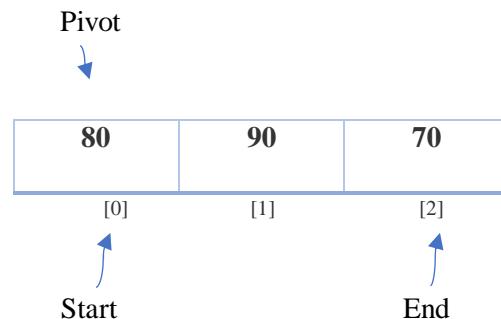
Check if A [Start] is less than Pivot , A [Start] = 30 , and , Pivot = 20 , $30 > 20$, so it is false , go to check the A [End] if it greater than Pivot , A [End] = 30 , and Pivot = 20 , $30 > 20$ so it is true , decrease End , End - - $\rightarrow 2 - 1 = 1$, so it well goes to index 1 .



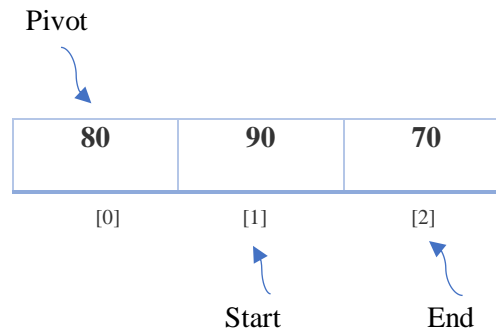
Check if A [End] is greater than Pivot , A [End] = 10 , and Pivot = 20 , $10 < 20$, so it is false , go to check if the Start < End , Start = 2 , End = 1 , $2 > 1$ so it is false , so swap (Pivot , A [End]) .



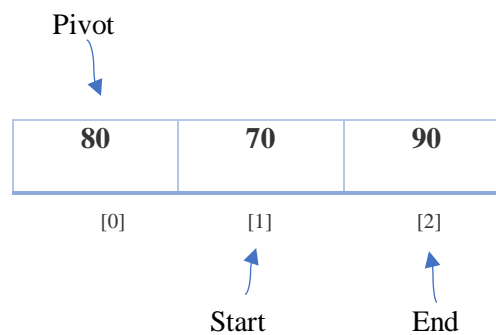
End of the first sup array , now I well make the sort in the second sup array :



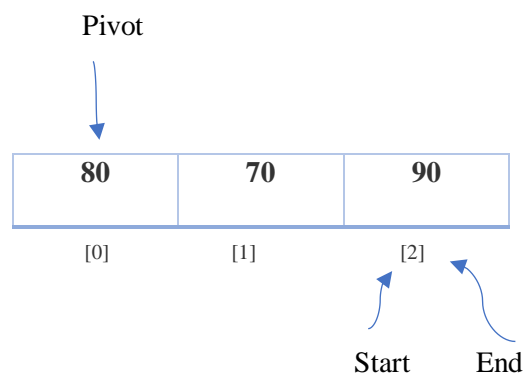
A [Start] = Pivot , (80 = 80) , so increase Start , Start ++ $\rightarrow 0 + 1 = 1$, it well goes to the index 1 .



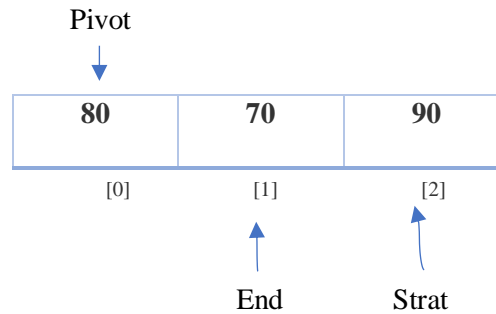
Check if A [Start] is less than Pivot , A [Start] = 90 . and Pivot = 80 , $90 > 80$ so it is false , go to check if A [End] is greater than Pivot , A [End] = 70 , Pivot = 80 , $70 < 80$ so it also false , go to check if Start < End , Start = 1 , End = 2 , $1 < 2$ so it is true , swap (90 ,70) .



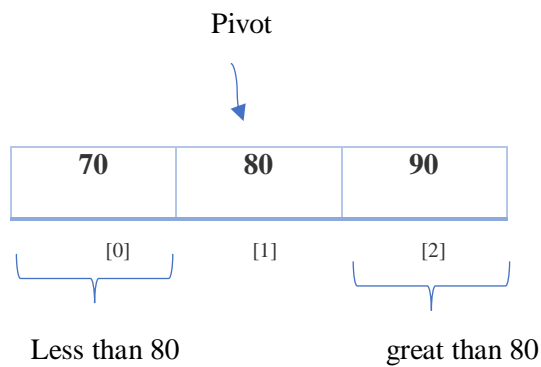
Check if A [Start] is less than Pivot , A [Start] = 70 , and Pivot = 80 , $70 < 80$ so it is true , increase Start , Start ++ $\rightarrow 1 + 1 = 2$, it well goes to index 2 .



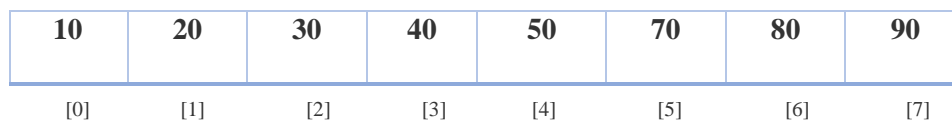
Check if A [Start] is less than Pivot , A [Start] = 90 , and Pivot = 80 , $90 > 80$, so it is false go to check A [End] if it greater than Pivot , A [End] = 90 , Pivot = 80 , $90 > 80$ so it is true , decrease End , End - - $\rightarrow 2 - 1 = 1$, it well goes to index 1 .



$70 < 80$, is false , check if Start < End , Start = 2 , End = 1 , $2 > 1$ so it is false , swap (Pivot , A [End]).



Finally array after Quick Sort :



Code for Quick Sort in Code Blocks :

Partition function :

// function to make partition in the array we want to sort (divide the array based on the less and the great of the pivot)

```
int partition (int arr[] , int first_i , int last_i ){  
    // the key  
    int pivot = arr[first_i];  
    // pointer to the index 0 in the array  
    int start = first_i;  
    // pointer to the last index in the array  
    int end = last_i;  
    //to make swap  
    int temp;  
    //if the start > end it well not enter the loop  
    while ( start < end ){  
        // while the data in index start < = pivot increase start  
        while ( arr[start] <= pivot && start<last_i){  
            start ++;  
        }  
        //while the data in index end in greater than pivot decreases the end  
        while (arr[end] > pivot){  
            end--;  
        }  
        //if tow while loop dont enter check if start < end and swap there data  
        if (start<end){  
            temp = arr[start];  
            arr[start] = arr[end];  
            arr[end] =temp ;  
        }  
    }  
}
```



```

    }
}

// if start > end swap between pivot and data in the index end
temp = arr[first_i];
arr[first_i] = arr[end];
arr[end] = temp ;

return end ;
}

```

```

// function to make partition in the array we want to sort (divide the array based on the less and the great of the pivot)
int partition (int arr[] , int first_i , int last_i ){
    // the key
    int pivot = arr[first_i];
    // pointer to the index 0 in the array
    int start = first_i;
    // pointer to the last index in the array
    int end = last_i;
    //to make swap
    int temp;
    //if the start > end it well not enter the loop
    while ( start < end ){
        // while the data in index start < = pivot increase start
        while ( arr[start] <= pivot && start<last_i){
            start ++;
        }
        //while the data in index end in greater than pivot decreases the end
        while (arr[end] > pivot){
            end--;
        }
        //if tow while loop dont enter check if start < end and swap there data
        if (start<end){
            temp = arr[start];
            arr[start] = arr[end];
            arr[end] =temp ;
        }
    }
    // if start > end swap between pivot and data in the index end
    temp = arr[first_i];
    arr[first_i] = arr[end];
    arr[end] =temp ;

    return end ;
}

```

Figure 2 :code of Partition function

Quick sort function :

```

//function of Quick sort
void QuickSort (int arr[] ,int first_i , int last_i){
    if ( first_i < last_i) {
        int loc = partition(arr,first_i,last_i);
    }
}

```

```

//for the side that less than pivot
QuickSort(arr,first_i,loc-1);

//for the side that greater than pivot
QuickSort(arr,loc+1,last_i);
}

```

```

//function of Quick sort
void QuickSort (int arr[] ,int first_i , int last_i){
    if ( first_i < last_i) {
        int loc = partition(arr,first_i,last_i);
        //for the side that less than pivot
        QuickSort(arr,first_i,loc-1);
        //for the side that greater than pivot
        QuickSort(arr,loc+1,last_i);
    }
}

```

Figure 3:code of QuickSort function

```

"C:\Users\user\Desktop\DATA STRUCTER 2\new_Quick\bin\Debug\new_Quick.exe"
Enter size of the list: 7
Enter 7 integer values: 70 50 40 90 30 80 10
List after sorting is: 10 30 40 50 70 80 90
Process returned 0 (0x0)   execution time : 25.532 s
Press any key to continue.

```

Figure 4 : the output of QuickSort code

Explanation of the code :

Partition function :

it has 3 prameters , **arr** is the array that has the numbers not sorted , **first_i** it means the index 0 this to get the start point and the Pivot number , **last_i** it means the last index in the array this to get the end point .

Pivot = arr[first_i] ; this line to get the first number in the array to make the sort based to it.

Start = first_i ; to make Start as pointer , Initially point to the first number in the array

End = last_i ; to make end as pointer , Initially point to the last number in the array

While (start < end) { } this while will continue to work if the start less than end , is start is greater than end the code will not enter this while loop , because in case start < end I want it to increase start or decrease end or make swap , in case start > end the code will do something else .

while (arr[start] <= pivot && start<last_i){ start ++ ; } this while loop will continue to work if arr[start] <= pivot && start<last_i and if the condition is true increase start with 1.

while (arr[end] > pivot){ end -- ; } this while loop will continue to work if arr[end] > pivot and if the condition is true decrease end with 1 .

if (start<end){ temp = arr[start]; arr[start] = arr[end]; arr[end] =temp ; } if the 2 while not work this means that start stop increasing and end stop decreasing , check if start < end , make swap between arr[start] and arr[end] .

temp = arr[first_i]; arr[first_i] = arr[end]; arr[end] =temp ; in case start > end it will not enter the main loop , so it will swap between pivot and arr[end] .

return end ; to now the position of Pivot

QuickSort function :

int loc = partition(arr,first_i,last_i); to get the position of pivot

QuickSort(arr,first_i,loc-1); to make a Quick sort for the sub array that less than Pivot

QuickSort(arr,loc+1,last_i); to make a Quick sort for the sub array that greater than Pivot

Time and space complexity :

The Quick Sort is a Recursion function so I will calculate the time complexity by Recurrence relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + n \rightarrow 2T\left(\frac{n}{2^1}\right) + 1 \cdot n$$

$$T\left(\frac{n}{2}\right) = 2\left[2T\left(\frac{n/2}{2}\right) + \frac{n}{2}\right] + n \rightarrow 4T\left(\frac{n}{4}\right) + 2n \rightarrow 2T\left(\frac{n}{2^2}\right) + 2 \cdot n$$

$$T\left(\frac{n}{4}\right) = 4\left[2T\left(\frac{n/4}{4}\right) + \frac{n}{4}\right] + 2n \rightarrow 8T\left(\frac{n}{8}\right) + 3n \rightarrow 3T\left(\frac{n}{2^3}\right) + 3 \cdot n$$

↓
...

$$\text{SO, } T(n) = 2^K T\left(\frac{n}{2^K}\right) + K \cdot n$$

Now, let $2^K = n$

$$\log 2^k = \log n$$

$$k \log 2 = \log n$$

$$k = \log n$$

$$T(n) = n T\left(\frac{n}{n}\right) + \log n \cdot n$$

$$= n \cdot T(1) + n \log n$$

$$= n \log n$$

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

I but $T(n) = 2T\left(\frac{n}{2}\right) + n$ like this because, $2T\left(\frac{n}{2}\right)$ I for two recursive call function for the 2 sub array so this equation divide the array into two arrays to make the Quick sort, and $+n$ I for the time complexity of Partition function.

And this time complexity $O(n) = n \log n$ is for Best and Average cases (random sort numbers in the array) .

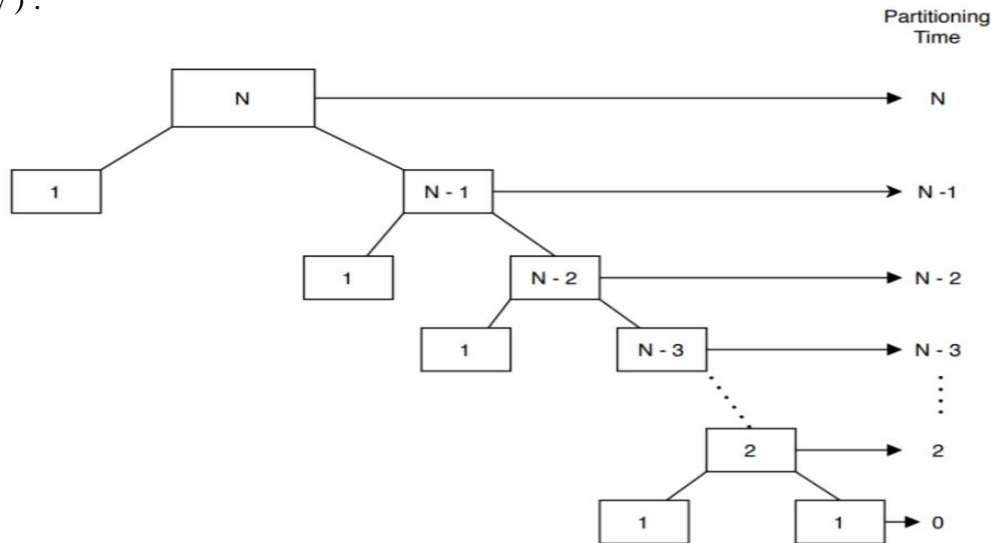


Figure 5 : worst case time complexity in q_sort

If we sum $\rightarrow N + (N-1) + (N-2) + (N-3) + \dots = \frac{N(N+1)}{2} = O(N^2)$

So when the numbers in array sort ascending or descending , the Quick sort will be in the worst case . [5]

- **Best Case: $\Omega(N \log(N))$**
The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.
In this case, the algorithm will make balanced partitions, leading to efficient Sorting.[1]
- **Average Case: $\theta(N \log(N))$**
Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm. [1]
- **Worst Case: $O(N^2)$**
The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting. [1]

- Auxiliary Space: $O(1)$, if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make $O(N)$. [1]

1.2 Counting Sort :

Counting Sort is a **non-comparison-based** sorting algorithm that works well when there is limited range of input values. 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. [6]

Advantage 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 input.
- 2- Counting sort is easy to code
- 3- Counting sort is a **stable algorithm**.

Disadvantage of Counting Sort:

- 1- Counting sort doesn't work on decimal values.
- 2- Counting sort is inefficient if the range of values to be sorted is very large.
- 3- Counting sort is not an **In-place sorting** algorithm, It uses extra space for sorting the array elements.

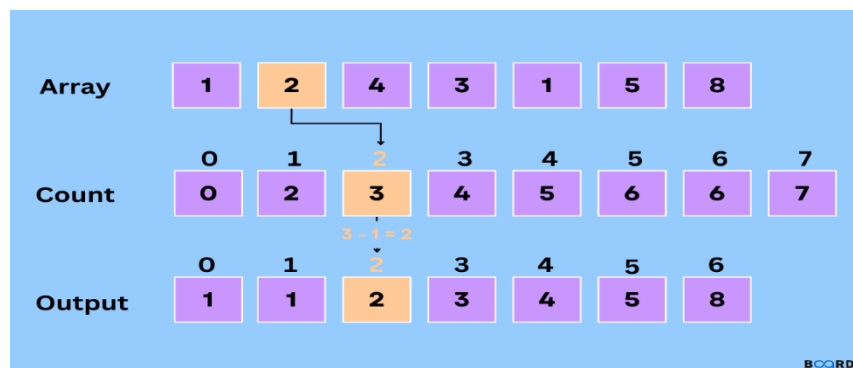


Figure 6 : Counting Sort

Counting Sort algorithm :

- **Step 1** – Find the maximum number in the array.
- **Step 2** - Define new array (count) and make the size of count = max number + 1 and initial count to 0 .
- **Step 3** - discover the total number of occurrences of each element and save the count in the count array at the index .
- **Step 4** - Add the count(i) and count(i-1) counts to get the cumulative sum, which you may save in the count array.
- **Step 5** - Decrement the count of each element copied by one in array (output) .
- **Step 6** - but the sorted array (output) in the array . [7]

```
CountingSort ( Array array , size ) {
```

```
    Array outoput
```

```
    Max = array [0]
```

```
    For ( i=0 ; i < size ; i++ )
```

```
        If array [i] > max
```

```
            Max = array [i]
```

```
    Array count
```

```
    For ( i=0 ; i<= max ; i++)
```

```
        Count [i] = 0
```

```
    For (i=0 ; i<=size ;i++)
```

```
        Count[array [i] ]++
```

```
    For (i=1 ; i<=max ; i++)
```

```
        Count [i] = count [i] + count[i-1]
```

For (i = size-1 ; i >= 0 ; i --)

Output[-- count [array [i]]] = array [i]

Example of Counting Sort :

Array :

i ↓	2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Array have 17 number so the size of it = 16 , then the max number in the array = 9 , next I will create new array (count) , the size of it = max + 1 → 9 + 1 = 10 (the size of count)

Count :

0	0	0	0	0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Then start to check every number in the Array , go to the index in the Count that equal to the number in the Array and increase the number in the Counter by 1 . so I will make for loop to take all numbers in the Array by increase i (i++) , first Array [0] = 2 so go to index 2 in Count then Count [2] ++ , increase i = 0 + 1 = 1 .

Array:

i ↓	2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Count :

0	0	1	0	0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Now i point to index 1 , Array [1] so go to index 1 in Count and Count[1] ++ , then increase I by one i ++ . $i = 1 + 1 = 2$.

Array:

<div>i ↓</div>																
2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Count :

0	1	1	0	0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Now i point to index 2 , Array [2] so go to index 1 in Count and Count[0] ++ , then increase I by one i ++ . $i = 2 + 1 = 3$.

Array :

<div>i ↓</div>																
2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Count

1	1	1	0	0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

We will repeat this until $i = 16$, and the Count will be like this :

Count :

3	3	4	0	1	1	0	2	1	2
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

And if you check the Array you will see that 0 repeat 3 times , 1 repeat 3 times , 2 repeat 4 times , 4 repeat 1 time , 5 repeat 1 time , 7 repeat 2 times , 8 repeat 1 time , 9 repeat 2 times .

Now we want to know the new index for each number in the Array to be in the true position , so we will update Count , Count [0] do not change Count = 3 , then we will start to sum the number in the index we stop on it and the number in index - 1 (Count [i] += Count [i - 1]) .

Count [1] = Count [1] + Count [0] → $3 + 3 = 6$ → Count [1] = 6

Count [2] = Count [2] + Count [1] → $4 + 6 = 10$ → Count [2] = 10

Count [3] = Count [3] + Count [2] → $0 + 10 = 10$ → Count [3] = 10

Count [4] = Count [4] + Count [3] → $1 + 10 = 11$ → Count [4] = 11

Count [5] = Count [5] + Count [4] → $1 + 11 = 12$ → Count [5] = 12

Count [6] = Count [6] + Count [5] → $0 + 12 = 12$ → Count [6] = 12

Count [7] = Count [7] + Count [6] → $2 + 12 = 14$ → Count [7] = 14

Count [8] = Count [8] + Count [7] → $1 + 14 = 15$ → Count [8] = 15

Count [9] = Count [9] + Count [8] → $2 + 15 = 17$ → Count [9] = 17

Count

3	6	10	10	11	12	12	14	15	17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Now start to sort the number in Array in the right position in a new array (output) , start to check the number in Array from the last index in the Array , Array [16] = 9 so go to index 9 in

Count , Count [9] = 17 then decrease Count [9] -- , $17 - 1 = 16$, then go to output [16] and insert 9 in this position and we will repeat this proses .

Array :

																i
2	1	1	0	2	5	4	0	2	8	7	7	9	2	0	1	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Count :

3	6	10	10	11	12	12	14	15	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Output :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Array [15] = 1 , go to Count [1] , Count [1] = 6 , $6 - 1 = 5$, go to output [5] and put 1 in it .

Array [14] = 0 , go to Count [0] , Count [0] = 3 , $3 - 1 = 2$, go to output [2] and put 0 in it .

Array [13] = 2 , go to Count [2] , Count [2] = 10 , $10 - 1 = 9$, go to output [9] and put 2 in it .

Array [12] = 9 , go to Count [9] , Count [9] = 16 , $16 - 1 = 15$, go to output [15] and put 9 in it .

Array [11] = 7 , go to Count [7] , Count [7] = 14 , $14 - 1 = 13$, go to output [13] and put 7 in it .

Array [10] = 7 , go to Count [7] , Count [7] = 13 , $13 - 1 = 12$, go to output [12] and put 7 in it .

Array [9] = 8 , go to Count [8] , Count [8] = 15 , $15 - 1 = 14$, go to output [14] and put 8 in it .

Array [8] = 2 , go to Count [2] , Count [2] = 9 , $9 - 1 = 8$, go to output [8] and put 2 in it .

Array [7] = 0 , go to Count [0] , Count [0] = 2 , $2 - 1 = 1$, go to output [1] and put 0 in it .

Array [6] = 4 , go to Count [4] , Count [4] = 11 , $11 - 1 = 10$, go to output [10] and put 4 in it .

Array [5] = 5 , go to Count [5] , Count [5] = 6 , $12 - 1 = 11$, go to output [11] and put 5 in it .

Array [4] = 2 , go to Count [2] , Count [2] = 8 , $8 - 1 = 7$, go to output [7] and put 2 in it .
 Array [3] = 0 , go to Count [0] , Count [0] = 1 , $1 - 1 = 0$, go to output [0] and put 0 in it .
 Array [2] = 1 , go to Count [1] , Count [1] = 5 , $5 - 1 = 4$, go to output [4] and put 1 in it .
 Array [1] = 1 , go to Count [1] , Count [1] = 4 , $4 - 1 = 3$, go to output [3] and put 1 in it .
 Array [0] = 2 , go to Count [2] , Count [2] = 7 , $7 - 1 = 6$, go to output [6] and put 2 in it .

Output :

0	0	0	1	1	1	2	2	2	2	4	5	7	7	8	9	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Finally sort array output to the Array :

Array

0	0	0	1	1	1	2	2	2	2	4	5	7	7	8	9	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]

Code of Counting Sort in code blocks :

```
void countingSort(int array[], int size) {
    int output[50];

    // Find the largest element of the array
```

```

int max = array[0];
for (int i = 1; i < size; i++) {
    if (array[i] > max)
        max = array[i];
}
//array with size = max+1 , it will have the number of repet for each number in the array
int count[50];

// Initialize count array with all zeros.
for (int i = 0; i <= max; ++i) {
    count[i] = 0;
}

// Store the count of each element in array
for (int i = 0; i < size; i++) {
    count[array[i]]++;
}

// Store the actual position for each element in array
for (int i = 1; i <= max; i++) {
    count[i] += count[i - 1];
}

// Find the index of each element of the original array in count array, and
// place the elements in output array
for (int i = size - 1; i >= 0; i--) {
    output[-- count[ array [i]]] = array[i];
}

```

```

// Copy the sorted elements into original array
for (int i = 0; i < size; i++) {
    array[i] = output[i];
}
}

```

// Function to print an array

```

void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d ", array[i]);
    }
    printf("\n");
} [8]

```

```

void countingSort(int array[], int size) {
    int output[50];

    // Find the largest element of the array
    int max = array[0];
    for (int i = 1; i < size; i++) {
        if (array[i] > max)
            max = array[i];
    }
    //array with size = max+1 , it will have the number of repeat for each number in the array
    int count[50];

    // Initialize count array with all zeros.
    for (int i = 0; i <= max; ++i) {
        count[i] = 0;
    }

    // Store the count of each element in array
    for (int i = 0; i < size; i++) {
        count[array[i]]++;
    }

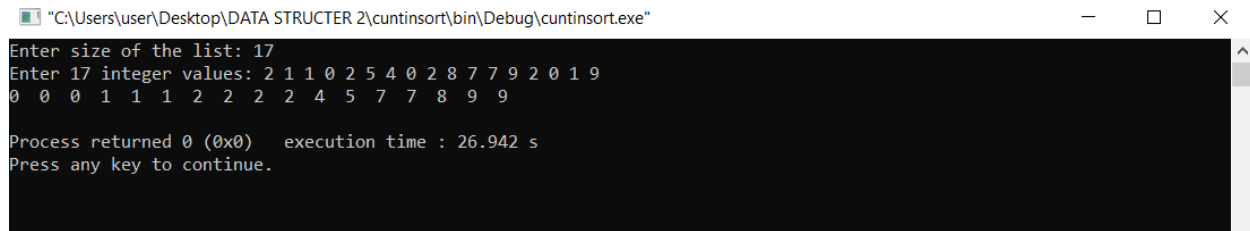
    // Store the actual position for each element in array
    for (int i = 1; i <= max; i++) {
        count[i] += count[i - 1];
    }

    // Find the index of each element of the original array in count array, and
    // place the elements in output array
    for (int i = size - 1; i >= 0; i--) {
        output[-- count[ array [i]]] = array[i];
    }

    // Copy the sorted elements into original array
    for (int i = 0; i < size; i++) {
        array[i] = output[i];
    }
}

```

Figure 7 : function of Counting Sort



```
"C:\Users\user\Desktop\DATA STRUCTER 2\cuntinsort\bin\Debug\cuntinsort.exe"
Enter size of the list: 17
Enter 17 integer values: 2 1 1 0 2 5 4 0 2 8 7 7 9 2 0 1 9
0 0 0 1 1 1 2 2 2 2 4 5 7 7 8 9 9
Process returned 0 (0x0)   execution time : 26.942 s
Press any key to continue.
```

Figure 8 : the out put Counting Sort code

Function Count Sort :

It has two parameter **int array[]** is the array that has the number not sorted , **int size** is the size of the array .

int max = array[0]; for (int i = 1; i < size; i++) { if (array[i] > max) max = array[i] } this for loop to know the maximum number in the array and save it in max variable .

int count[50]; this new array to know how many times the each number is repeated , and it size is equal to (max + 1) .

for (int i = 0; i <= max; ++i) { count[i] = 0; } this for loop to initial the count array with zero.

for (int i = 0; i < size; i++) { count[array[i]]++; } this for loop to put How many times each number in repeated , ex if Array [i] = 9 , it will go to Count [9] and increase it with one , it will repeat this until the I reach the size of the array .

for (int i = 1; i <= max; i++) { count[i] += count[i - 1]; } this for loop to update count array to have the actual position for numbers in array , this by start the loop form 1(because I want firs position not to chamg) to size of Count , this loop will sum the number that the I point on it count [i] with the number in the pointer - 1 , count [i - 1] .

for (int i = size - 1; i >= 0; i--) { output[-- count[array [i]]] = array[i]; } and this for loop to Arrange the numbers in array in right position in the new array it is called output , it will check the number which the pointer point on it (it will start from the end of the array) and then go to the Count [number] and decrease it and the result that we have by this drcreasing well by the new position of this number.

for (int i = 0; i < size; i++) { array[i] = output[i]; } and finally this for loop to copy the output array to the array .

Time and Space complexity :

for (int i = 1; i < size; i++) { **—————>** this for loop will run n(size of array) of times , so
if (array[i] > max) for loop = n
max = array[i];
}

for (int i = 0; i <= max; ++i) { **—————>** this for loop will num k time (k = max number
count[i] = 0; in array +1) , so for loop = k
}

for (int i = 0; i < size; i++) { **—————>** this for loop will run n times , for loop = n
count[array[i]]++;
}

for (int i = 1; i <= max; i++) { **—————>** this for loop will run k times , for loop = k
count[i] += count[i - 1];
}

for (int i = size - 1; i >= 0; i--) { **—————>** this for loop will run n times , for loop = n
output[-- count[array [i]]] = array[i];
}

for (int i = 0; i < size; i++) { **—————>** this for loop will run n times , for loop = n
array[i] = output[i];
}

Now calculate the time complexity $\rightarrow n + k + n + k + n + n = 4n + 2k = O(n + k)$

This is the time complexity for the Count sort

- Best case :

The Count sort will be in the best case when the numbers are sort ascending or descending and the number in the same rang that is when k is equal to 1. [9]

In this case, counting the occurrence of each element in the input range takes constant time and then finding the correct index value of each element in the sorted output array takes n time, thus the total time complexity reduces to $O(1 + n)$ **$O(n)$** which is **linear**. [9]

- Average case :

It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. [11]

in this case k computes to be $(k+1/2)$ and the average case will be $N+(K+1)/2$. But as K tends to infinity, K is the dominant factor.

Similarly, now if we vary N, we see that both N and K are equally dominant and hence, we have **$O(N+K)$** as average case.[9]

- worst case

the Count sort will be in the worst case if the number sorted randomly , and largest element is significantly large than other elements. This increases the range K . [9]

As the time complexity of algorithm is $O(n+k)$ then, for example, when k is of the order $O(n^2)$, it makes the time complexity $O(n+(n^2))$, which essentially reduces to $O(n^2)$ and if k is of the order $O(n^3)$, it makes the time complexity $O(n+(n^3))$, which essentially reduces to $O(n^3)$. Hence, in this case, the time complexity got worse making it **$O(k)$** for such larger values of k. And this is not the end. It can get even worse for further larger values of k. [9]

Thus the worst case time complexity of counting sort occurs when the range k of the elements is significantly larger than the other elements. [9]

- Space Complexity

In the above algorithm we have used an auxiliary array C of size k, where k is the max element of the given array. Therefore the space complexity of Counting Sort algorithm is $O(k)$. [9]

1.3 Comb Sort :

Comb Sort is mainly an improvement over Bubble Sort. Bubble sort always compares adjacent values. So all inversions are removed one by one. Comb Sort improves on Bubble Sort by using a gap of the size of more than 1. The gap starts with a large value and shrinks by a factor of 1.3 in every iteration until it reaches the value 1. Thus Comb Sort removes more than one inversion count with one swap and performs better than Bubble Sort.

The shrink factor has been empirically found to be 1.3 (by testing Combsort on over 200,000 random lists) . [12]

Advantages :

- Comb Sort is a simple and easy-to-implement sorting algorithm. [13]
- It is a variation of Bubble Sort, but with better average-case performance and lower worst-case time complexity. [13]
- The algorithm can be easily adapted to work on linked lists or other data structures with sequential access. [13]
- Comb Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the input array. [13]
- The algorithm can be tuned for different types of input data by adjusting the shrink factor and/or choosing the optimal initial gap value. [13]

Disadvantages :

- Although Comb Sort has a lower worst-case time complexity than Bubble Sort, it is still an $O(n^2)$ sorting algorithm. This means that it can be slow for large input sizes and is not as efficient as $O(n \log n)$ sorting algorithms such as Merge Sort or Quick Sort. [13]
- The performance of Comb Sort can be highly dependent on the choice of the shrink factor and the initial gap value. Choosing suboptimal values can lead to poor performance. [13]

- Unlike some other sorting algorithms, such as Insertion Sort or Selection Sort, Comb Sort does not have any early termination mechanism to detect when the input array is already sorted. This means that it may perform unnecessary iterations and comparisons even when the input is already sorted. [13]

Counting Sort algorithm :

- **Step 1** – start with an initial gap value , which is equal to the size of the array.
- **Step 2** – set shrink factor which is = 1.3 , for reducing the gap in each iteration .
- **Step 3** – compare elements that are gap position apart .
- **Step 4** – if the elements are out of order ($a[i] > a[i+gap]$) swap them .
- **Step 5** - reduce the gap by applying the shrink value .
- **Step 6** - continue comparing and swapping elements with new gap until the gap become

< 1 . [14]

Comb_sort (Array array [] , size)

Gap = size

Shrink = 1.3

While (gap \geq 1)

Gap = gap / shrink

i = 0

while ((i + gap) < size)

if (array [i] > array[i + gap])

temp = array [i]

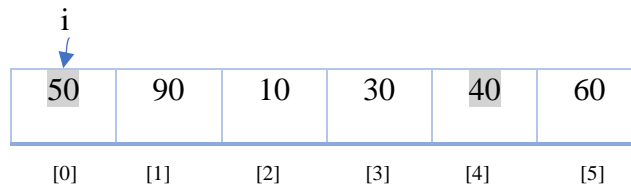
array[i] = array[i + gap]

array[i + gap] = temp

i++

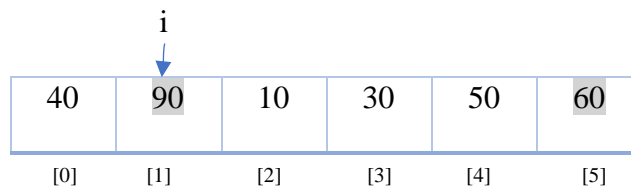
return array

Example of Counting Sort :

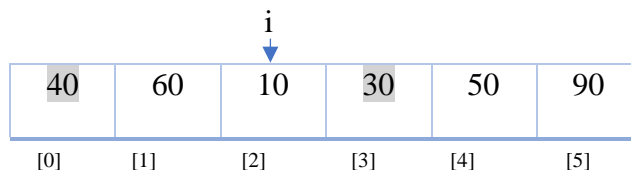


gap = size of the array , so gap = 6 , divide gap with shrink , $\text{gap} = 6 / 1.3 = 4$, now check if $(i + \text{gap}) < \text{size}$, $0 + 4 < 6$ true , so compare between array [i] and array [i+gap] , if array [i] > array [i+gap] , swap (array [i] , array [i+gap]). Note i = 0 here .

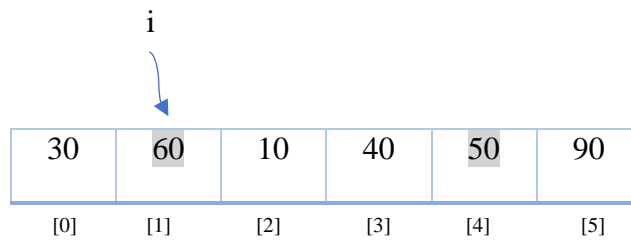
array [0] is greater than array [0 + 4] $\rightarrow 50 > 40$,true , swap (50 , 40) and then increase i , i++ , $0 + 1 = 1$, i = 1 .



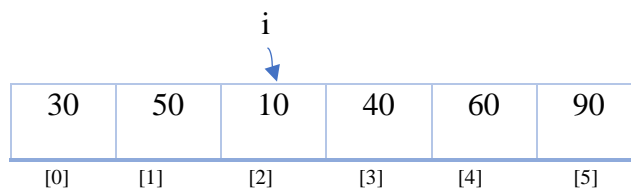
Check If $i + \text{gap} < \text{size}$, $1 + 4 = 5$, $5 < 6$ true , so compare between array [i] and array [i+gap] array[1] > array [1 + 4] $\rightarrow 90 > 60$ true , swap (90 , 60) and then increase i , i++ , $1 + 1 = 2$, i = 2 .



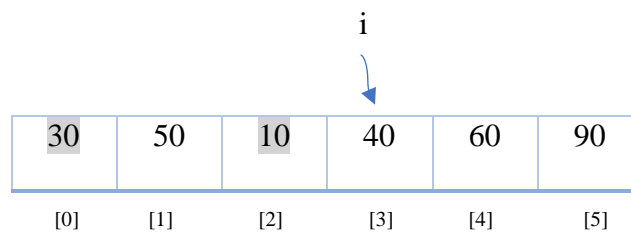
Check If $i + \text{gap} < \text{size}$, $2 + 4 = 6$, $6 < 6$ false , so gap now = 4 , divide gap by the shrink , $\text{gap} = 4 / 1.3 = 3$, reset i = 0 , now check if $(i + \text{gap}) < \text{size}$, $0 + 3 < 6$ true , so compare between array [i] and array [i+gap] , array [0] > array [0+3] , $40 > 30$ true , swap (40 , 30) , increase i by 1 , i++ , i = 0 + 1 = 1 .



Check If $i + \text{gap} < \text{size}$, $1 + 3 = 4$, $4 < 6$ true , so compare between array $[i]$ and array $[i+\text{gap}]$
 $\text{array}[1] > \text{array}[1 + 3]$ $\rightarrow 60 > 50$ true , swap (60 , 50) and then increase i , $i++$, $1 + 1 = 2$, $i = 2$.



Check If $i + \text{gap} < \text{size}$, $2 + 3 = 5$, $5 < 6$ true , so compare between array $[i]$ and array $[i+\text{gap}]$
 $\text{array}[2] > \text{array}[2 + 3]$ $\rightarrow 10 > 90$ false , do not swap , increase i , $i++$, $2+1=3$.



Check if $i + \text{gap} < \text{size}$, $3 + 3 < 6$ false , so reset i to zero , $\text{gap} = 3 / 1.3 = 2$, $0 + 2 < 6$ true ,
 compare between array $[i] > \text{array}[i+\text{gap}]$ $\rightarrow \text{array}[0] > \text{array}[0+2]$, $30 > 10$ true , swap (30 , 10) , increase i , $i++$, $0 + 1 = 1$.

	i				
	↓				
10	50	30	40	60	90
[0]	[1]	[2]	[3]	[4]	[5]

Check $i + \text{gap} < \text{size}$, $1 + 2 < 6$ true , array [1] > array [1 + 2] ➔ $50 > 40$ true , swap (50 , 40) , and increase i , $i++$, $1 + 1 = 2$.

		i			
		↓			
10	40	30	50	60	90
[0]	[1]	[2]	[3]	[4]	[5]

Check $i + \text{gap} < \text{size}$, $2 + 2 < 6$ true , array [2] > array [2 + 2] ➔ $30 > 60$ false , do not swap, and increase i , $i++$, $2 + 1 = 3$.

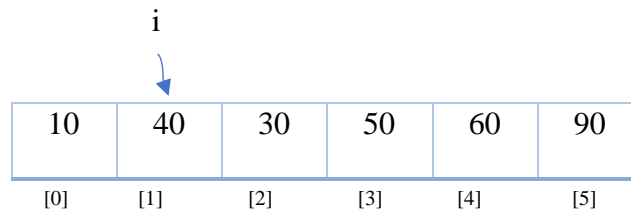
			i		
			↓		
10	40	30	50	60	90
[0]	[1]	[2]	[3]	[4]	[5]

Check $i + \text{gap} < \text{size}$, $3 + 2 < 6$ true , array [3] > array [3 + 2] ➔ $50 > 90$ false , do not swap, and increase i , $i++$, $3 + 1 = 4$.

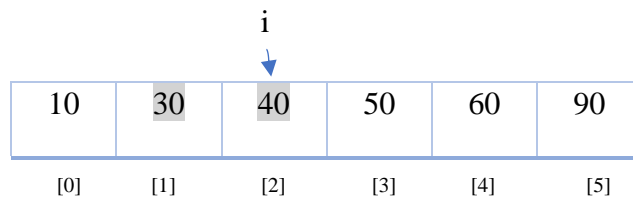
Then $4 + 2 < 6$ false , so reset $i=0$, $\text{gap} = 2/1.3 = 1$.

	i				
	↓				
10	40	30	50	60	90
[0]	[1]	[2]	[3]	[4]	[5]

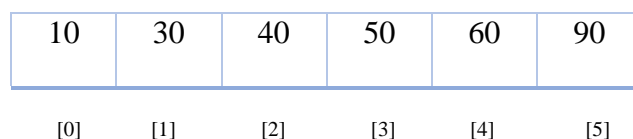
Check $i + \text{gap} < \text{size}$, $0 + 1 < 6$ true , array $[0] > \text{array}[0 + 1]$ ➔ $10 > 40$ false , do not swap, and increase i , $i++$, $0 + 1 = 1$.



Check $i + \text{gap} < \text{size}$, $1 + 1 < 6$ true , array $[1] > \text{array}[1 + 1]$ ➔ $40 > 30$ true , swap (40 , 30) , and increase i , $i++$, $1 + 1 = 2$.



We will continue to check if $i + \text{gap} < \text{size}$ and give true , and check if array $[i] > \text{array}[i + \text{gap}]$ and will give false because all numbers are sorted , and then $i + \text{gap} > \text{size}$, reset $i = 0$, $\text{gap} = 1 / 1.3 = 0$, that mean that all e=numbers are sorted .



Code of Counting Sort in code blocks :

```
int* comb_sort(int array_nums[], int size)
{
    int gap = size; // Initialize gap size
    int SHRINK = 1.3;
```

```

while (gap >= 1) // gap < 1 means that the array is sorted
{
    // Update the gap value for a next comb
    gap = gap / SHRINK;

    int i = 0;
    while ((i + gap) < size)
    { // similiar to the Shell Sort
        if (array_nums[i] > array_nums[i + gap])
        {
            int tmp = array_nums[i];
            array_nums[i] = array_nums[i + gap];
            array_nums[i + gap] = tmp;
        }
        i++;
    }
}

return array_nums;
}

```

```

int* comb_sort(int array_nums[], int size)
{
    int gap = size; // Initialize gap size
    int SHRINK = 1.3;
    while (gap >= 1) // gap < 1 means that the array is sorted
    {
        // Update the gap value for a next comb
        gap = gap / SHRINK;
        int i = 0;
        while ((i + gap) < size)
        { // similiar to the Shell Sort
            if (array_nums[i] > array_nums[i + gap])
            {
                int tmp = array_nums[i];
                array_nums[i] = array_nums[i + gap];
                array_nums[i + gap] = tmp;
            }
            i++;
        }
    }
    return array_nums;
}

```

Figure 9 : Comb Sort code

Time and Space complexity :

- **Best Case Complexity (all numbers sorted)** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of comb sort is $\theta(n \log n)$.
- **Average Case Complexity (number sort randomly)** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of comb sort is $\Omega(n^2/2^p)$, where p is a number of increments..
- **Worst Case Complexity(number stored ascending or descending)** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of comb sort is $O(n^2)$. [15]
- **Analysing the space**
Comb sort algorithm does not use any variable sized data structure or buffer nor make use of any recursive call, therefore, it takes $O(1)$ space in all the cases

2. Dynamic Programming

2.1 What is Dynamic Programming?

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to recompute them when needed later. This simple optimization reduces the complexities from exponential to polynomial. [16]

And it is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution. [17]

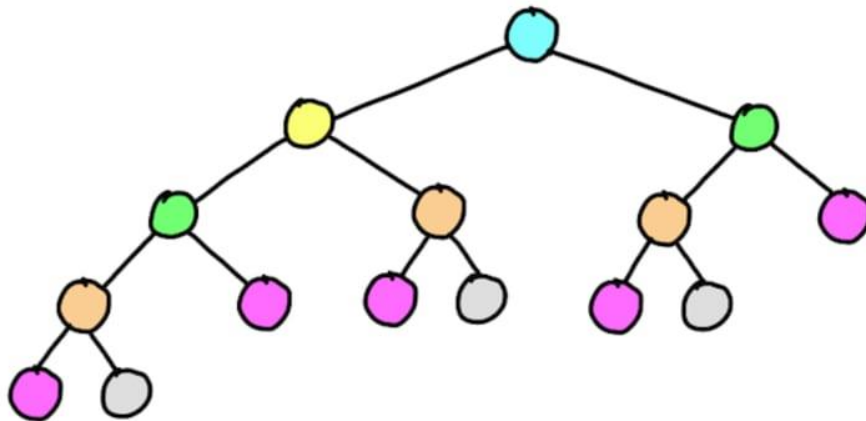


Figure 10 : Dynamic Programming

Where should dynamic programming be used?

Dynamic programming is used when one can break a problem into more minor issues that they can break down even further, into even more minor problems. Additionally, these subproblems have overlapped. That is, they require previously calculated values to be recomputed. With dynamic programming, the computed values are stored, thus reducing the need for repeated calculations and saving time and providing faster solutions. [17]

Advantages of Dynamic Programming:

- Efficiency gain: For addressing difficult problems, dynamic programming may significantly reduce time complexity compared to the naïve technique.
- Dynamic programming ensures that issues that adhere to the notion of optimality find optimal solutions.

Disadvantages of Dynamic Programming:

- High memory usage: When working with bigger input sizes, dynamic programming uses a lot of memory to hold answers to sub-problems.
- Finding the appropriate sub-problems can be difficult, and doing so frequently necessitates a deep understanding of the main issue at hand.

2.2 three problems that can be solved using DynamicProgramming:

- Fibonacci Revised [24]
- Coin Change Problem Number [23]
- Longest Common Subsequence [23]

2.3 example of a code that solves a problem using DynamicProgramming and without using DynamicProgramming

1.Fibonacci Revised :

The Fibonacci sequence is a set of integers (the Fibonacci numbers) that starts with a zero, followed by a one, then by another one, and then by a series of steadily increasing numbers. The sequence follows the rule that each number is equal to the sum of the preceding two numbers. [18]

The Fibonacci sequence begins with the following 14 integers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233 ...

Each number, starting with the third, adheres to the prescribed formula. For example, the seventh number, 8, is preceded by 3 and 5, which add up to 8. [18]

The sequence can theoretically continue to infinity, using the same formula for each new number. Some resources show the Fibonacci sequence starting with a one instead of a zero, but this is fairly uncommon. [18]

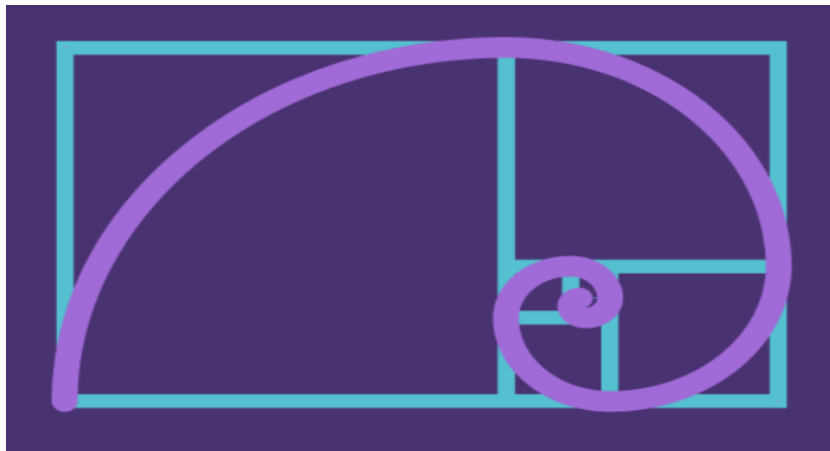


Figure 11 : Fibonacci Revised

```
int Fib ( int n ) {  `` this function without Dynamic Programming ``  
if ( n < 0 ) {  
printf("error"); }  
if ( n == 0 ) {  
return 0 ; }  
if ( n==1 ){  
return 1 ; }  
sum = Fib ( n-1 ) + Fib ( n-2 ) ;  
return sum ;  
}
```

[illegible]

As we see the call function repeat 15 time , until the number increase the calling of recursive function will increase also , let count = number of calling function :

• • • •

$n = 10 \rightarrow \text{count} = 177$

$n = 20 \rightarrow \text{count} = 21891$

as we see the number of calling function be very large and is increasing exponentially $O(2^n)$, so to solve this problem we use Dynamic Programming to sup the problem and calculate this sub problem and save the result in array once ,to use it without call the function many times . once the call od function is repeat it will just take the value from the array without solve it .

f : (initially it is null '-1')

-1	-1	-1	-1	-1	-1
----	----	----	----	----	----

```
static int fib(int n) `` update the function with the Dynamic Programming
{
    int i;

    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers in the series
        and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}
```

F : after the Dynamic

0	1	1	2	3	5
---	---	---	---	---	---

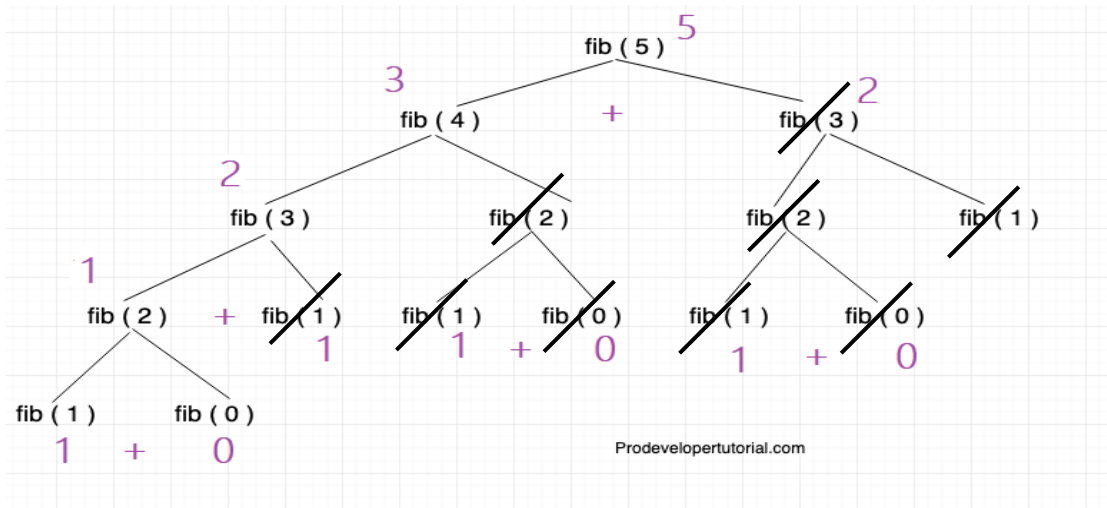


Figure 13 : Fibonacci Revised for 5 after DB

I clear this fib because they are save in the array , so we do not have to call the function , after this the time of calling = 6 , it means it decrease the time complexity from $O(2^n)$ to $O(n)$.

3. References

- [1] <https://www.geeksforgeeks.org/quick-sort/>
[Accessed on 29-1-2024, 11:35 Am]
- [2] <https://www.youtube.com/watch?v=QN9hnmAgmOc>
[Accessed on 29-1-2024, 12:05 pm]
- [3] http://www.btechsmartclass.com/data_structures/quick-sort.html
[Accessed on 29-1-2024, 4:13 pm]
- [4] <https://www.geeksforgeeks.org/counting-sort/>
[Accessed on 30-1-2024, 1:01 pm]
- [5] <https://www.baeldung.com/cs/quicksort-time-complexity-worst-case>
[Accessed on 30-1-2024, 1:44 pm]
- [6] <https://www.geeksforgeeks.org/counting-sort/>
[Accessed on 30-1-2024, 3:30 pm]
- [7] <https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm>
[Accessed on 30-1-2024, 4:16 pm]
- [8] <https://www.programiz.com/dsa/counting-sort>
[Accessed on 30-1-2024, 5:11 pm]
- [9] <https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/>
[Accessed on 31-1-2024, 11:19 Am]
- [10] <https://youcademy.org/counting-sort-algorithm/>
[Accessed on 31-1-2024, 11:39 Am]
- [11] <https://www.javatpoint.com/counting-sort>
[Accessed on 31-1-2024, 11:40 Am]
- [12] <https://www.geeksforgeeks.org/comb-sort/>
[Accessed on 31-1-2024, 12:43 Pm]
- [13] <https://www.algowalker.com/comb-sort.html>
[Accessed on 31-1-2024, 12:50 Pm]

- [14] <https://www.sarthaks.com/3570439/comb-sort-algorithm>
[Accessed on 31-1-2024, 1:12 Pm]
- [15] <https://www.javatpoint.com/comb-sort>
[Accessed on 31-1-2024, 9:24 Pm]
- [16] <https://www.geeksforgeeks.org/dynamic-programming/>
[Accessed on 31-1-2024, 10:24 Pm]
- [17] <https://www.spiceworks.com/tech/devops/articles/what-is-dynamic-programming/>
[Accessed on 31-1-2024, 10:37 Pm]
- [18] <https://www.techtarget.com/whatis/definition/Fibonacci-sequence>
[Accessed on 31-1-2024, 11:02 Pm]
- [19] <https://stackoverflow.com/questions/37873654/fibonacci-series-using-dynamic-programming>
[Accessed on 31-1-2024, 11:24 Pm]
- [20] <https://www.geeksforgeeks.org/understanding-the-coin-change-problem-with-dynamic-programming/>
[Accessed on 1-2-2024, 12:08 Am]
- [21] <https://stackoverflow.com/questions/37873654/fibonacci-series-using-dynamic-programming>
[Accessed on 1-2-2024, 12:45 Am]
- [22] <https://www.simplilearn.com/tutorials/data-structure-tutorial/what-is-dynamic-programming>
[Accessed on 1-2-2024, 1:45 Am]
- [23] <https://medium.com/@tigerkdp/examples-of-problems-that-can-be-solved-using-dynamic-programming-582ae71af52f>
[Accessed on 1-2-2024, 2:00 Am]
- [24] <https://www.waynewbishop.com/dynamic-programming-problems>
[Accessed on 1-2-2024, 2:10 Am]