



Department of Computer Science

COMP2421-Data Structures and Algorithms

(First Semester–Fall 2023/2024)

Project#4

Due Date: 2 February 2024 @ 11:00 PM

Instructor: Dr. Ahmad Abusnaina

Abstract:

This report explores sorting algorithms and dynamic programming. In the sorting section, three new sorting algorithms are studied, each explained with an example and corresponding code for the functions. The time and space complexities are also stated for various scenarios such as random data generation, ascending order, and descending order.

In the dynamic programming section, the concept of dynamic programming is defined. Three problems that can be solved using this technique are mentioned. Additionally, an example code is provided to solve a problem both with and without using dynamic programming.

Contents

Introduction	5
1) Sorting	5
2) Dynamic Programming	5
Theory	5
1)sorting	5
1.1Gnome Sort	5
1.1.1How gnome sort works?	6
1.1.2Algorithm Steps:	6
1.1.3 Complexity	7
1.1.4 Some of my notes.....	7
1.1.5 Implementations in c language & it's running.....	8
1.1.5.1 ascending sort by gnome sort	8
1.1.5.2 descending sort by gnome sort	10
1.1.6 code with choose data random of gnome sort & run it.....	13
1.2 Comb Sort Algorithm	15
1.2.1How comb sort works?	15
The vanilla comb-sort algorithm uses increments that form a geometric progression. Let N be the number of elements in the array and let r be the shrinkage factor. Then, mathematically the gap size takes values from the following sequence:	15
1.2.3 Complexity of comb sort	15
1.2.4code with choose data random of comb sort & run it.....	15
1.3 Counting Sort	18
1.3.1What is Counting Sort?	18
1.3.2 How does Counting Sort Algorithm work?	19
1.3.3 summary steps Counting Sort Algorithm:	29
1.3.4Complexity Analysis of Counting Sort:	29
1.3.5Advantage of Counting Sort:	30
1.3.6Disadvantage of Counting Sort:.....	30
1.3.7 Implementation counting sort by c language with random values	30
2)Dynamic Programming	33
2.1 What is Dynamic Programming?	33
Simply put, dynamic programming is an optimization method for recursive algorithms, most of which are used to solve computing or mathematical problems.....	33

Dynamic programming is a problem-solving technique for resolving complex problems by recursively breaking them up into sub-problems, which are then each solved individually. Dynamic programming optimizes recursive programming and saves us the time of re-computing inputs later. dynamic programming allows us to simply store the results of each step the first time and reuse it each subsequent time. so that we do not have to re-compute them when needed later.	33
Dynamically programmed solutions have a polynomial complexity which assures a much faster running time than other techniques like recursion or backtracking. In most cases, dynamic programming reduces time complexities, also known as big-O, from exponential to polynomial.	33
2.2 Mention three problems that can be solved using Dynamic Programming.	33
2.2.1)Knuth’s Optimization in Dynamic Programming	33
2.2.2) Coin Change Problem	33
2.2.3. Fibonacci	35
Figure: Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers	36
example from Dynamic Programming: Examples, Common Problems, and Solutions (makeuseof.com).....	36
2.3 Show an example of a code that solves a problem using DynamicProgramming and without using DynamicProgramming.	37
2.3.1Techniques to solve Dynamic Programming Problems:.....	37
2.3.1. Top-Down(Memoization):.....	37
2.3.2 Bottom-Up(Dynamic Programming):.....	37
1)Recognize the DP problem	38
2)Identify problem variables	38
3) Express the recurrence relation	38
4) Identify the base case	38
5) Decide the iterative or recursive approach	38
6) Add memoization	38
2.3.2 Nth Fibonacci Number	39
2.3.2.1 Iterative Approach to Find and Print Nth Fibonacci Numbers:.....	39
2.3.2.2 Recursion Approach to Find and Print Nth Fibonacci Numbers:.....	40
2.3.2.3 Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers:	41
3) Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers:	43
2.2.4Advantages & Disadvantages	43
2.2.5When should DP be used to solve a problem?	43
4)References:	45
1)link references of Gnome Sort.....	45

Introduction

1) Sorting

Sorting is the process of arranging elements in a specific order, typically in ascending or descending order, based on certain criteria. It is a fundamental task in computer science and plays a crucial role in a wide range of applications. Sorting allows us to organize and retrieve data efficiently, making it easier to search, analyze, and manipulate information.

Sorting is used for several reasons. Firstly, it helps in improving the efficiency of searching algorithms. When data is sorted, it becomes easier to locate specific elements through techniques like binary search, which significantly reduces the search time. Sorting also facilitates data analysis by enabling algorithms to perform operations like finding the largest or smallest element or calculating medians. Sorting is applicable in various scenarios and domains. It is commonly employed in databases, where data often needs to be presented in a well-organized manner. Sorting is also used in data analysis, computational geometry, and optimization problems. In fields such as finance, e-commerce, and logistics, sorting is vital for tasks like inventory management, order processing, and financial analysis.

2) Dynamic Programming

Dynamic programming is a powerful technique used in computer science and mathematics to solve optimization problems by breaking them down into smaller and overlapping subproblems. It is a method for efficiently solving complex problems by dividing them into smaller, manageable subproblems and storing the solutions to these subproblems to avoid repetitive calculations. Dynamic programming offers a systematic approach to problem-solving, allowing for optimal solutions to be found in a time-effective manner. It can be applied to various problem domains and has been successfully used in fields such as computer science, operations research, economics, and biology. It provides a framework for solving problems that would otherwise be computationally expensive or impractical to solve using brute-force methods.

Theory

1) sorting

1.1 Gnome Sort

- Gnome sort is used everywhere where a stable sort is not needed.
- Gnome Sort is an in-place sort that does not require any extra storage.
- The gnome sort is a sorting algorithm which is similar to insertion sort in that it works with one item at a time but gets the item to the proper place by a series of swaps, similar to a bubble sort.

- It is conceptually simple, requiring no nested loops. The average running time is $O(n^2)$ but tends towards $O(n)$ if the list is initially almost sorted
- Stability: gnome sort is stable, meaning it maintains the relative order of elements with equal values. If two elements have the same value, they will not be swapped during the sorting process.

1.1.1 How gnome sort works?

Lets consider an example: $arr[] = \{34, 2, 10, -9\}$

- Underlined elements are the pair under consideration.
- "Red" colored are the pair which needs to be swapped.
- Result of the swapping is colored as "blue"

```

34 2 10 -9
2 34 10 -9
2 34 10 -9
2 10 34 -9
2 10 34 -9
2 10 34 -9
2 10 34 -9
2 10 34 -9
2 10 -9 34
2 10 -9 34
2 -9 10 34
2 -9 10 34
-9 2 10 34
-9 2 10 34
-9 2 10 34
-9 2 10 34(Sorted output)

```

1.1.2 Algorithm Steps:

- 1) If you are at the start of the array then go to the right element (from $arr[0]$ to $arr[1]$).
- 2) If the current array element is larger or equal to the previous array element then go one step right

```

if (arr[i] >= arr[i-1])
    i++;

```

- 3) If the current array element is smaller than the previous array element then swap these two elements and go one step backwards

```

    if (arr[i] < arr[i-1])
    {
        swap(arr[i], arr[i-1]);
        i--;
    }

```

4) Repeat steps 2) and 3) till 'i' reaches the end of the array (i.e- 'n-1')

5) If the end of the array is reached then stop and the array is sorted.

1.1.3 Complexity

The variable – 'index' in our program doesn't always get incremented, it gets decremented too, so the time complexity is $O(N^2)$. As there are no nested loops (only one while) it may seem that this is a linear $O(N)$ time algorithm, so the time complexity is $O(N^2)$.

Space Complexity: This is an in-place algorithm. So $O(1)$ auxiliary space is needed.

- Worst case time complexity: $\Theta(N^2)$
- Average case time complexity: $\Theta(N^2)$
- Best case time complexity: $\Theta(N)$
- Space complexity: $\Theta(1)$ auxiliary

1.1.4 Some of my notes

In general, Gnome Sort and Bubble Sort are similar in their basic technique of repeatedly swapping adjacent elements. However, Gnome Sort is a more adaptive algorithm that can take advantage of partially sorted data and maintains stability, while Bubble Sort is a bit more straightforward but lacks adaptability. Both algorithms have a quadratic time complexity $\{O(n^2)\}$, which makes them less efficient for large data sets compared to more advanced sorting algorithms such as Merge Sort or Quick Sort.

1.1.5 Implementations in c language & it's running

1.1.5.1 ascending sort by gnome sort

- 1) Enter the elements number will store it in array & enter type of input: Enter 1 for integer input or 2 for character input

```
#include <stdio.h>
#include <stdlib.h>
// levan burait
//1211439
//#1

int main()
{
    // need int variable l & temp ,
    // int n to limit the number of element in array then will initial size of array will change
    int l, temp, n;
    // ascending order sort or 2 for descending
    int sortType ;
    // type of input integer or characters
    int type ;
    printf("\nEnter the elements number will store it in array :");
    scanf("%d", &n);
    int arrayint[n];
    char arraychar[n];

    printf("\nEnter 1 for integer input or 2 for character input: ");
    scanf("%d", &type);
    if (type== 1 ){
        printf("\nEnter elements :\n");
        for (l = 0; l < n; l++){
            scanf("%d", &arrayint[l]);
        }
    }
```

Figure: Enter the elements number will store it in array & enter type of input: Enter 1 for integer input or 2 for character input

- 2) user if choose 1 ascending order sort or 2 for descending order sort

```
// user if choose 1 ascending order sort or 2 for descending order sort
printf("\nEnter 1 for ascending order sort or 2 for descending order sort: ");
scanf("%d", &sortType);
if (sortType ==1 ){
    l = 0;
    while (l < n)
    {
        if (l == 0 || arrayint[l - 1] <= arrayint[l])
            l++;
        else
        {
            temp = arrayint[l-1];
            arrayint[l - 1] = arrayint[l];
            arrayint[l] = temp;
            l = l- 1;
        }
    }
}
```


Figure: ascending order of integer type
With the for loop to print array

```

    for (l = 0; l < n; l++){
        printf("%d\t\t", arrayint[l]);
    }
}

```

If run above then the result

```

Enter the elements number will store it in array :8
Enter 1 for integer input or 2 for character input: 1
Enter elements :
1
34
567
890
543
876
432
16
Enter 1 for ascending order sort or 2 for descending order sort: 1
1        16        34        432        543        567        876        890
Process returned 0 (0x0)   execution time : 33.209 s
Press any key to continue.

```

Figure: run integer type inputs then sort it ascending by gnome sort

The ascending order of character type as same above but use the array of characteristics size of it n : will enter by user then if choose type characters & choose ascending order

```

    else if (type == 2){
        printf("\nEnter the character elements :\n");
        for (l = 0; l < n; l++)
            scanf(" %c", &arraychar[l]);
        // user if choose 1 ascending order sort or 2 for descending order sort
        printf("\nEnter 1 for ascending order sort or 2 for descending order sort: ");
        scanf("%d", &sortType);
        if (sortType == 1){
            l = 0;
            while (l < n)
            {
                if (l == 0 || arraychar[l - 1] <= arraychar[l])
                    l++;
                else
                {
                    temp = arraychar[l-1];
                    arraychar[l - 1] = arraychar[l];
                    arraychar[l] = temp;
                    l = l - 1;
                }
            }
        }
    }
}

```

Figure: ascending order of character type
With the for loop to print array of characters

```

    for (l = 0; l < n; l++) {
        printf("%c \t\t", arraychar[l]);
    }

```

If run above then the result

```

Enter 1 for integer input or 2 for character input: 2
Enter the character elements :
f
e
d
c
b
a

Enter 1 for ascending order sort or 2 for descending order sort: 1


a b c d e f
Process returned 0 (0x0)   execution time : 34.328 s
Press any key to continue.

```

Figure: run character type inputs then sort it ascending by gnome sort

1.1.5.2 descending sort by gnome sort

- 1) Enter the elements number will store it in array & enter type of input: Enter 1 for integer input or 2 for character input
- 2) user if choose 1 ascending order sort or 2 for descending order sort

```

        // 2 descending
        else if (sortType ==2 ){
            l = 0;
            while (l < n)
            {
                if (l == 0 || arrayint[l - 1] >= arrayint[l])
                    l++;
                else
                {
                    temp = arrayint[l-1];
                    arrayint[l - 1] = arrayint[l];
                    arrayint[l] = temp;
                    l = l - 1;
                }
            }
        }

        else {
            printf("\nInvalid sort type entered.");
            return 0;
        }
    }
}

```

Figure: descending order of integer type & else if not choose 1 or 2 & with for loop to print array integer

```

        for (l = 0; l < n; l++){
            printf("%d\t\t", arrayint[l]);
        }
    }
}

```

Then the result will as fellow picture

```

Enter the elements number will store it in array :10
Enter 1 for integer input or 2 for character input: 1
Enter elements :
101
12
55
89
40
700
1000
402
66
679

Enter 1 for ascending order sort or 2 for descending order sort: 2
1000      700      679      402      101      89      66      55      40
12
Process returned 0 (0x0)   execution time : 75.426 s
Press any key to continue.
|

```

Figure: run integer type inputs then sort it descending by gnome sorting

The descending order of character type as same above but use the array of characteristics size of it n : will enter by user then if choose type characters & choose descending order

```

        // 2 descending
        else if (sortType ==2 ){
            l = 0;
            while (l < n)
            {
                if (l == 0 || arraychar[l - 1] >= arraychar[l])
                    l++;
                else
                {
                    temp = arraychar[l-1];
                    arraychar[l - 1] = arraychar[l];
                    arraychar[l] = temp;
                    l = l - 1;
                }
            }
        }

        else {
            printf("\nInvalid sort type entered.");
            return 0;
        }

        for (l = 0; l < n; l++){
            printf("%c ", arraychar[l]);
        }
    }
}

```

Figure: descending order of character type & else if not choose 1 or 2 & with for loop to print array character

Then the result will as fellow picture

```

Enter 1 for integer input or 2 for character input: 2

Enter the character elements :
u
t
l
k
z
e

Enter 1 for ascending order sort or 2 for descending order sort: 2
z u t l k e

Process returned 0 (0x0)   execution time : 23.110 s
Press any key to continue.
|

```

Figure: run characters type inputs then sort it descending by gnome sorting

1.1.6 code with choose data random of gnome sort & run it

```
printf("\nEnter 1 for integer input or 2 for character input: ");
scanf("%d", &type);
if (type== 1 ){

// printf("\nEnter elements :\n");
// random data
printf("\nGenerating random integer elements...\n");
srand(time(0)); // Initialize random seed

    for (l = 0; l < n; l++){
        arrayint[l] = rand() % 100;

        printf("\nGenerated integer elements: ");
    for (l = 0; l < n; l++){
        printf("%d\t", arrayInt[l]);

/ user if choose 1 ascending order sort or 2 for descending order sort
printf("\nEnter 1 for ascending order sort or 2 for descending order sort: ");
scanf("%d", &sortType);
if (sortType ==1 ){
l = 0;
while (l < n)
{
    if (l == 0 || arrayint[l - 1] <= arrayint[l])
        l++;
    else
        l = l - 1;
}
```

Figure: generation random integer element

Library: #include <time.h>

will generate random integer elements between 0 and 99

Result of ascending by random integer element

```
Enter the elements number will store it in array :12
Enter 1 for integer input or 2 for character input: 1
Generating random integer elements...
Generated integer elements: 12 72 12 88 37 6 49 56 65 2 38 77
Enter 1 for ascending order sort or 2 for descending order sort: 1
2 6 12 12 37 38 49 56 65 72 77 88
Process returned 0 (0x0) execution time : 6.602 s
Press any key to continue.
```

Figure: Result of ascending by random integer element

```
Enter the elements number will store it in array :12
Enter 1 for integer input or 2 for character input: 1
Generating random integer elements...
Generated integer elements: 25 23 4 39 25 0 82 92 74 94 24 40
Enter 1 for ascending order sort or 2 for descending order sort: 2
94 92 82 74 40 39 25 25 24 23 4 0
Process returned 0 (0x0) execution time : 6.128 s
Press any key to continue.
```

Figure: Result of descending by random integer element

```

    }
}

else if (type == 2){
    // printf("\nEnter the character elements :\n");
    printf("\nGenerating random character elements...\n");
    srand(time(0)); // Initialize random seed

    for (l = 0; l < n; l++){
        arraychar[l] = 'A' + rand() % 26;

        printf("\nGenerated character elements: ");
        for (l = 0; l < n; l++){
            printf("%c", arraychar[l]);
        }
    }
}

user if choose 1 ascending order sort or 2 for descending order sort
printf("\nEnter 1 for ascending order sort or 2 for descending order sort: ");
scanf("%d", &sortType);
if (sortType == 1){
    l = 0;
    while (l < n)
    {
        if (l == 0 || arraychar[l - 1] <= arraychar[l])
            l++;
        else
        {
            temp = arraychar[l-1];
            arraychar[l - 1] = arraychar[l];
            arraychar[l] = temp;
            l++;
        }
    }
}

```

Figure: generation random integer element

random uppercase character elements from 'A' to 'Z'

```

Enter the elements number will store it in array :12

Enter 1 for integer input or 2 for character input: 2

Generating random character elements...

Generated character elements: L D      W      X      X      O      R      A      K      B      D      L
Enter 1 for ascending order sort or 2 for descending order sort: 1
A B D D K L L O R W X X
Process returned 0 (0x0)   execution time : 10.108 s
Press any key to continue.
|

```

Figure: Result of ascending by random character element

```

120 | while (l < n)
C:\Users\hnp\Desktop\gnome |
Enter the elements number will store it in array :12

Enter 1 for integer input or 2 for character input: 2

Generating random character elements...

Generated character elements: T U      H      X      P      Z      I      C      Z      L      Z      G
Enter 1 for ascending order sort or 2 for descending order sort: 2
Z      Z      Z      X      U      T      P      L      I      H      G      C
Process returned 0 (0x0)   execution time : 7.100 s
Press any key to continue.

```

Figure: Result of descending by random character element

1.2 Comb Sort Algorithm

Comb sorting is essentially an improvement over bubble sorting. Bubble sort always compares adjacent values. Comb Sort improves on Bubble Sort by using a gap with a size larger than 1. The gap starts with a large value and shrinks by a factor of 1.3 in each iteration until it reaches the value 1. Thus Comb performs better than Bubble Sort.

The shrinkage factor was empirically found to be 1.3 (by Comb sort testing on over 200,000 random lists) [Source: [Wiki](#)]

Although it works better than Bubble Sort on average, worst-case remains $O(n^2)$.

1.2.1 How comb sort works?

The vanilla comb-sort algorithm uses increments that form a geometric progression. Let N be the number of elements in the array and let r be the shrinkage factor. Then, mathematically the gap size takes values from the following sequence:

$$\left\{ \frac{N}{r}, \frac{N}{r^2}, \frac{N}{r^3}, \dots \right\}$$

1.2.3 Complexity of comb sort

1) Average case time complexity of the algorithm is $(N^2/2^p)$, where p is the number of increments.

2) The worst-case complexity of this algorithm is $O(n^2)$

3) the Best Case complexity is $O(n \log n)$.

4) Space complexity : $O(1)$.

1.2.4 code with choose data random of comb sort & run it

```
// levan burait
//1211439
//#1
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// global array int with size =100
int newArrayInt[100];
//global array character with size =100
char newArrayChar[100];
// swap the integer in integer array
void swapInt(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
// function to swap the character in char array
void swapChar(char *x, char *y) {
    char temp = *x;
    *x = *y;
    *y = temp;
}
```

Figure: function to swap integer type & the other of characters type

```

// function comb sort of integer ascending
void combSortInt(int arr[], int n) {
    // number of element in array = gap
    int gap = n;
    // shrink factor : r = 1.3 by WIKI search in report
    float shrink = 1.3;
    // limit if array sort
    int sorted = 0;
    // update n = gap = number of element by divide it by 1.3 : r : shrink factor
    while (gap > 1 || sorted) {
        //\[(gap == 1) \land \text{no swaps occurred in the last pass}\]
        if (gap > 1) {
            gap = gap / shrink;
        }

        sorted = 0;
        // will add to last update of gap (gap/r) to i start 0 then iteration until i + gap < number of element:n
        for (int i = 0; i + gap < n; i++) {
            // swap element if arr[i] > arr[i + gap] this sort ascending
            // in index i if element larger than element in index i+gap then swap it

            if (arr[i] > arr[i + gap]) {
                swapInt(&arr[i], &arr[i + gap]);
                // update sorted to 1
                sorted = 1;
            }
        }
    }
}

```

Figure: function comb sort of integer ascending

```

// function comb sort of character ascending
void combSortChar(char arr[], int n) {
    // number of element in array = gap

    int gap = n;
    // shrink factor : r = 1.3 by WIKI search in report
    float shrink = 1.3;
    int sorted = 0;

    while (gap > 1 || sorted) {
        if (gap > 1) {
            gap = gap / shrink;
        }

        sorted = 0;
        // will add to last update of gap (gap/r) to i start 0 then iteration until i + gap < number of element:n
        // this sort ascending
        for (int i = 0; i + gap < n; i++) {
            // in index i if element larger than element in index i+gap then swap it
            if (arr[i] > arr[i + gap]) {
                swapChar(&arr[i], &arr[i + gap]);
                // update sorted to 1

                sorted = 1;
            }
        }
    }
}

```

Figure: function comb sort of character ascending


```

// function to sort integer descending of integer type
void combSortIntDesc(int arr[], int n) {
    // number of element in array = gap

    int gap = n;
    // shrink factor : r = 1.3 by WIKA search in report

    float shrink = 1.3;
    int sorted = 0;

    while (gap > 1 || sorted) {
        if (gap > 1) {
            gap = gap / shrink;
        }

        sorted = 0;

        for (int i = 0; i + gap < n; i++) {
            // in index i if element less than element in index i+gap then swap it

            if (arr[i] < arr[i + gap]) {
                swapInt(&arr[i], &arr[i + gap]);
                sorted = 1;
            }
        }
    }
}

```

Figure: function comb sort integer descending of integer type

```

// function to sort integer descending of character type
void combSortCharDesc(char arr[], int n) {
    // number of element in array = gap

    int gap = n;
    // shrink factor : r = 1.3 by WIKA search in report

    float shrink = 1.3;
    int sorted = 0;
    // iteration until the value of gap ~1
    while (gap > 1 || sorted) {
        if (gap > 1) {
            gap = gap / shrink;
        }

        sorted = 0;

        for (int i = 0; i + gap < n; i++) {
            // in index i if element less than element in index i+gap then swap it
            if (arr[i] < arr[i + gap]) {
                swapChar(&arr[i], &arr[i + gap]);
                sorted = 1;
            }
        }
    }
}

```

Figure: function comb sort integer descending of character type

In the int main() {} as same logic in gnome sort code but I use the switch case instead of (if , else if) then the result as this

```

Enter the number of elements to store in array: 12
Enter 1 for integer input or 2 for character input: 1
Generating random integer elements...
Generated integer elements: 94 34 55 19 5 78 17 93 12 11 13 76
Sorting using Comb Sort in ascending order...
Sorted array in ascending order:
5 11 12 13 17 19 34 55 76 78 93 94
Sorting using Comb Sort in descending order...
Sorted array in descending order:
94 93 78 76 55 34 19 17 13 12 11 5
Process returned 0 (0x0) execution time : 4.440 s
Press any key to continue.

```

figure: run of integer type then sort ascending & descending by comb sort

```

"C:\Users\hnp\Desktop\sort alk" X + v
Enter the number of elements to store in array: 12
Enter 1 for integer input or 2 for character input: 2
Generating random character elements...
Generated character elements: C O E C R X R L G Y H F
Sorting using Comb Sort in ascending order...
Sorted array in ascending order:
C C E F G H L O R R X Y
Sorting using Comb Sort in descending order...
Sorted array in descending order:
Y X R R O L H G F E C C
Process returned 0 (0x0) execution time : 5.503 s
Press any key to continue.

```

figure: run of character type then sort ascending & descending by comb sort

1.3 Counting Sort

1.3.1 What is 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.

1.3.2 How does Counting Sort Algorithm work?

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

1) Find out the **maximum** element from the given array.

As follow input array the max of element is 5

Step 1:

	0	1	2	3	4	5	6	7	max
inputArray	2	5	3	0	2	3	0	3	5

Counting Sort



Figure: from [Counting Sort - Data Structures and Algorithms Tutorials - GeeksforGeeks](#)

2) Initialize a countArray[]

max element in input array = k as above input array k=5 then

Length of count array = max element in input array +1 = $k+1 = 5+1 = 6$

Then puts all elements as 0 in counter array .

This array will be used for storing the occurrences of the elements of the input array.

Step 2:

	0	1	2	3	4	5
countArray	0	0	0	0	0	0

Figure: Initialize a **countArray[]** of length **$k+1 = 6$** with all elements as **0**

3)

- In the **countArray[]**, store the count of each unique element of the input array at their respective indices.
- **Value of element in each index in counting array:**
- The count of element **0** in the input array is **2**. So, store **2** at index **0** in the **countArray[]**.
- The count of element **1** in the input array is **0**. So, store **0** at index **1** in the **countArray[]**.
- The count of element **2** in the input array is **2**. So, store **2** at index **2** in the **countArray[]**.
- The count of element **3** in the input array is **3**. So, store **3** at index **3** in the **countArray[]**.
- The count of element **4** in the input array is **0**. So, store **0** at index **4** in the **countArray[]**.
- The count of element **5** in the input array is **1**. So, store **1** at index **5** in the **countArray[]**.

Then the counting array will become as the fellow figure



Figure: value element of counting sort

Step 4:

- Store the **sum** of the elements of the **countArray[]** by doing **countArray[i] = countArray[i - 1] + countArray[i]**.

	0	1	2	3	4	5
countArray	2	0	2	3	0	1

Will update value of element

	0	1	2	3	4	5
countArray	2	2	4	7	7	8

After update

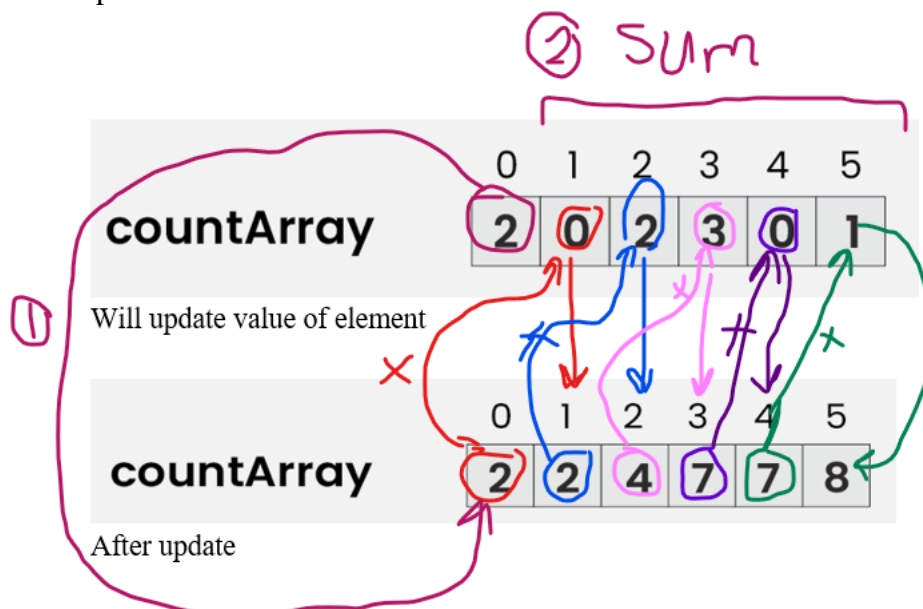


Figure: Update the value

1) the index 0 after update will have the same value of it before just move it as shown above in 1

2) update value from 1 to k : first index 1 to index =5 by sum use this

$\text{countArray}[i] = \text{countArray}[i - 1](\text{after update}) + \text{countArray}[i].$ (found in step2)

*element value of index 1

$\text{countArray}[1] = \text{countArray}[1 - 1] + \text{countArray}[1].$

$\text{countArray}[1] = 2 + 0 = 2$

$\text{countArray}[1] = 2$

*element value of index 2

$\text{countArray}[2] = \text{countArray}[2 - 1] + \text{countArray}[2].$

$\text{countArray}[2] = 2 + 2 = 4$

$\text{countArray}[2] = 4$

*element value of index 3

countArray[3] = countArray[3 - 1] + countArray[3].

countArray[3] = 4+3 = 7

countArray[3] = 7

*element value of index 4

countArray[4] = countArray[4- 1] + countArray[4].

countArray[4] = 7+0 = 7

countArray[4] = 7

*element value of index 4

countArray[5] = countArray[5- 1] + countArray[5].

countArray[5] = 7+1 = 8

countArray[5] = 8

then the counter array as follow

Step 4 :

	0	1	2	3	4	5
countArray	2	2	4	7	7	8

Counting Sort

Figure: counter array after update

In c language implementation this part by 2 for loop as follow:

```
for(i=0; i < n; i++)  
{  
    ++Count[a[i]];  
}  
for(i=1; i <= k; i++)  
{  
    Count[i] = Count[i] + Count[i-1];  
}
```

Figure: ascending sort from video https://youtu.be/pEJiGC-ObQE?si=wSOK46p2e-Nu2H_T

```

        count[i] = count[i] - 1;
    }
    // update element value in count from 0 to new value as i expliane it in report
    //count[i] += count[i + 1]
} else if (order == 2) { // Descending order

    for (int i = max_value - 1; i >= 0; i--) {
        count[i] += count[i + 1];
    }
}

```

Figure: descending sort by counting sort

5)built output array size of it = size of input array

- Iterate from end of the input array and because traversing input array from end preserves the order of equal elements, which eventually makes this sorting algorithm **stable**.

Index of out array := $\text{countArray}[\text{inputArray}[i]] - 1$

Element value of above index of out array = $\text{Inputtearray}[i]$

For $i = 6$

1) $\text{Inputtearray}[6] = 0$

0: this will store in outarray but where this index???

,so go to $\text{countarray}[0] = 2$

then decrease value of element 2 by 1 as : $2 - 1 = 1$

this **1** is the index of out array & the element value is **0**

for $i = 6$ then as follows:

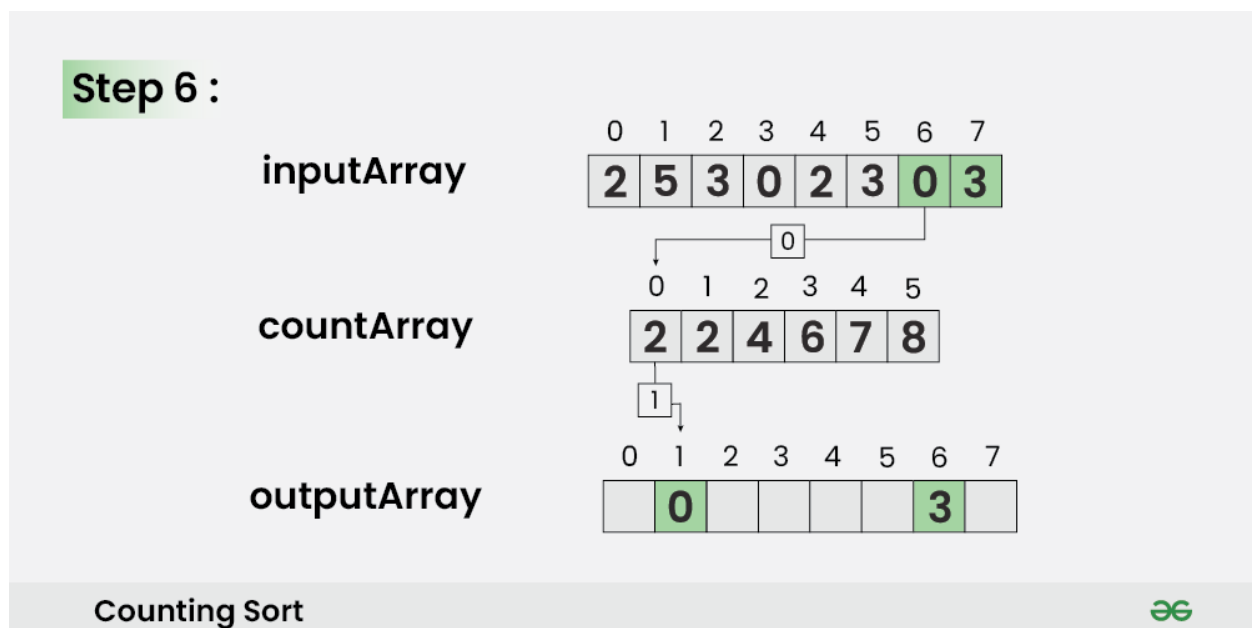


Figure: For $i = 6$

Index of out array : = $\text{countArray}[\text{inputArray}[i]]$ - -

Element value of above index of out array = $\text{Inputearray}[i]$

For $i = 5$

1) $\text{Inputearray}[5] = 3$

2) 3: this will store in outarray but where this index???

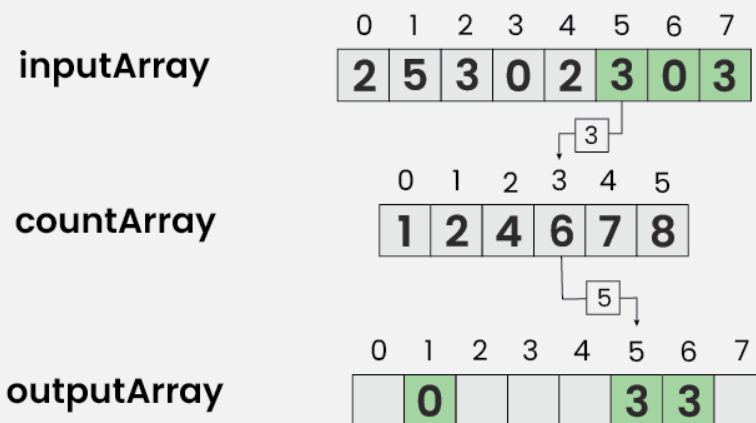
,so go to $\text{countarray}[3] = 6$

then decrease value of element 6 by 1 as : $6-1 = 5$

this 5 is the index of out array & the element value is 3

for $i = 6$ then as follows:

Step 7:



Counting Sort



Figure: For $i=5$

For $i = 4$,

Index of out array : = $\text{countArray}[\text{inputArray}[i]]$ - -

Element value of above index of out array = $\text{Inputearray}[i]$

For $i = 4$

1) $\text{Inputearray}[4] = 2$

2) 2 : this will store in outarray but where this index???

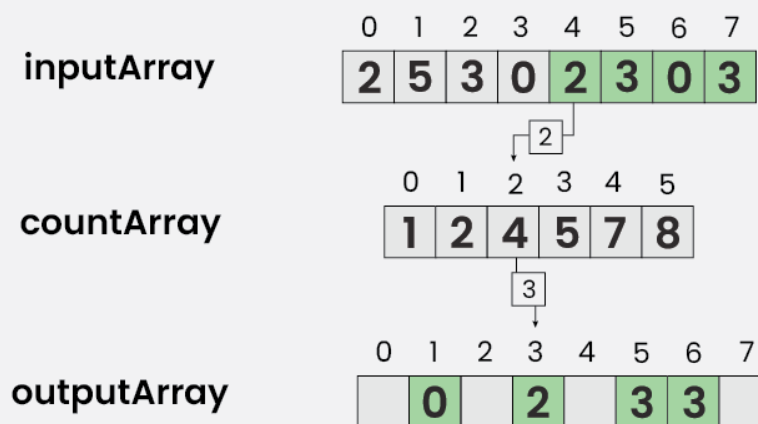
,so go to countarray[2] = 4

then decrease value of element 4 by 1 as : $4-1 = 3$

this 3 is the index of out array & the element value is 2

for $i = 4$ then as follows:

Step 8 :



Counting Sort



For $i = 3$,

Update $outputArray[countArray[inputArray[3]] - 1] = inputArray[3]$

Also, update $countArray[inputArray[3]] = countArray[inputArray[3]] - 1$

Index of out array : = $countArray[inputArray[i]] - 1$

Element value of above index of out array = $inputArray[i]$

1) $inputArray[3] = 0$

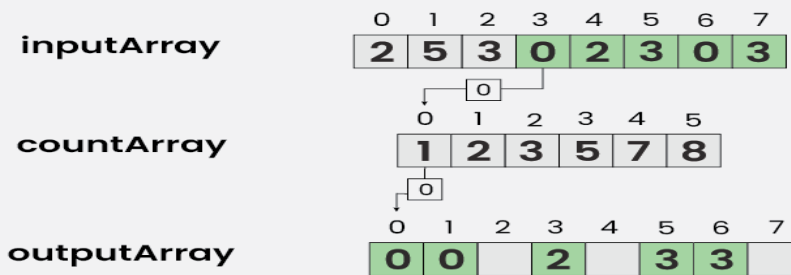
2) 0: this will store in outarray but where this index???

,so go to countarray[0] = 1

then decrease value of element 1 by 1 as : $1-1 = 0$

this 0 is the index of out array & the element value is 0

Step 9 :



Counting Sort



For $i = 2$,

1) $\text{Inputarray}[2] = 3$

2) 3: this will store in outarray but where this index???

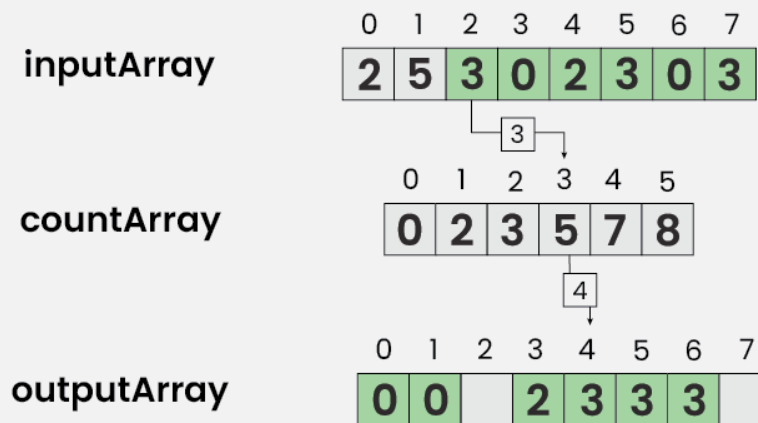
,so go to $\text{countarray}[3] = 5$

then decrease value of element 5 by 1 as : $5-1 = 4$

this 4 is the index of out array & the element value is 3

as following :

Step 10 :



Counting Sort



For $i=2$

For $i = 1$

1) $\text{Inputarray}[1] = 5$

2)5: this will store in outarray but where this index???

,so go to countarray[5] = 8

then decrease value of element 3 by 1 as : $8-1 = 7$

this **7** is the index of out array & the element value is **5**

as following:

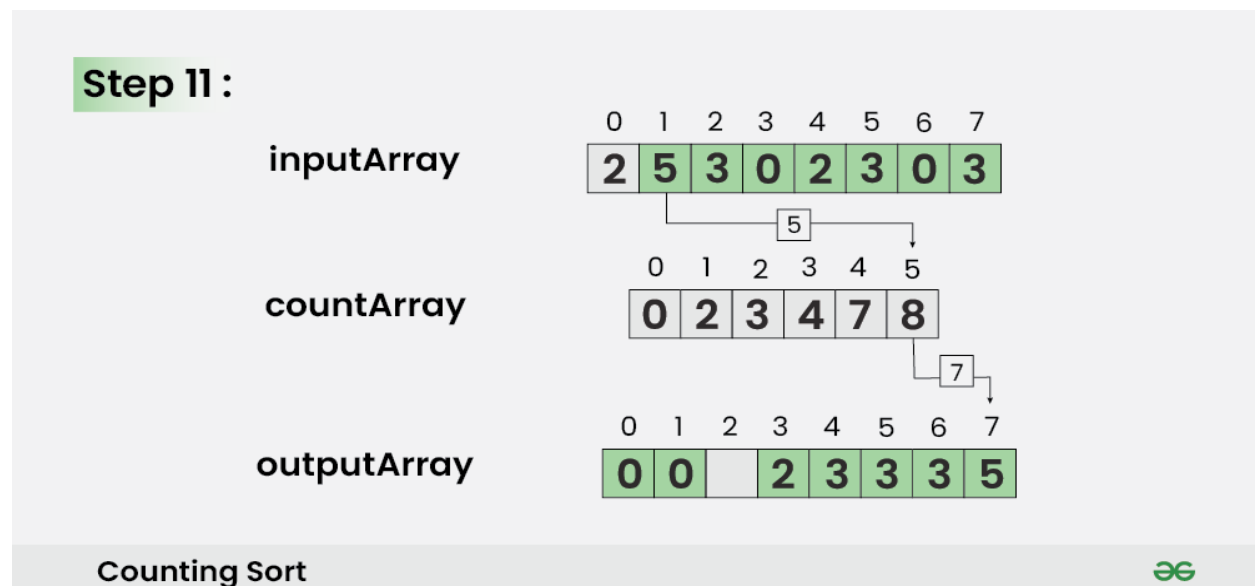


Figure: For i=1

For i = 0,

1)Inputearray[0] = **2**

2)2: this will store in outarray but where this index???

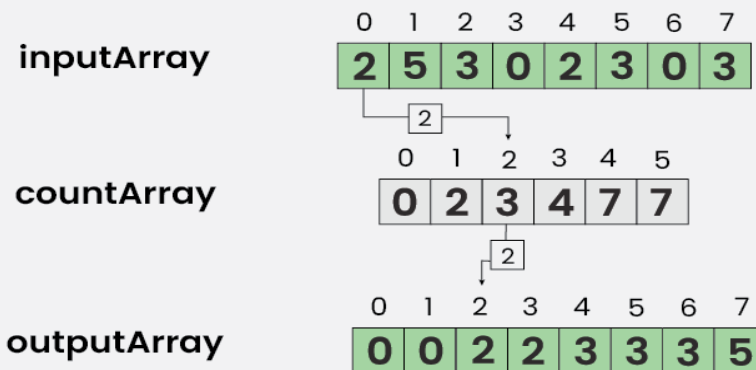
,so go to countarray[2] = 3

then decrease value of element 3 by 1 as : $3-1 = 2$

this **2** is the index of out array & the element value is **2**

as following:

Step 12 :



Counting Sort



Figure: for $i=0$

Another example by the above steps

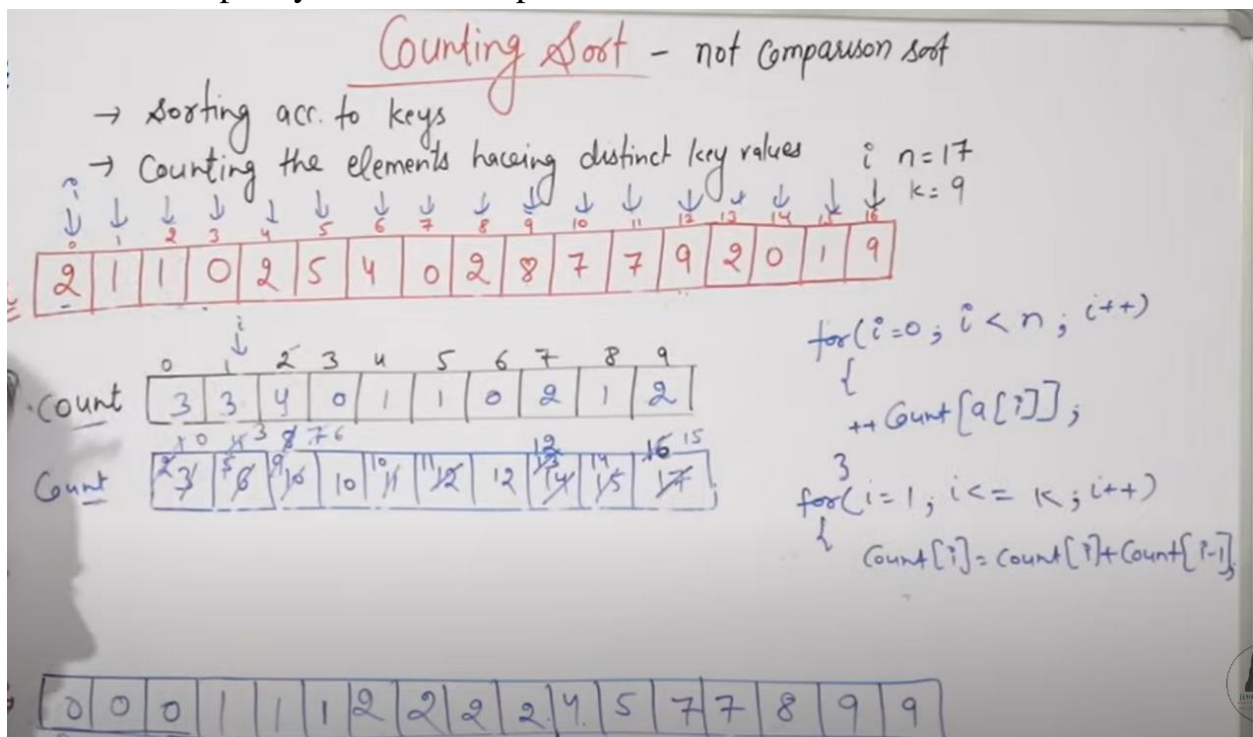


Figure: example from: <https://youtu.be/pEJiGC-ObQE?si=JAyv4NwvcAoTipxe>

Note:

1) All picture of this steps from:

[Counting Sort - Data Structures and Algorithms Tutorials - GeeksforGeeks](https://www.geeksforgeeks.org/counting-sort/)

2) This link of video help me to write all above steps

<https://youtu.be/pEJiGC-ObQE?si=JAyv4NwvcAoTipxe>

1.3.3 summary steps Counting Sort Algorithm:

- Declare an array **countArray[]** of size **max(inputArray[])+1** and initialize it with **0**.
- Traverse array **inputArray[]** and map each element of **inputArray[]** as an index of **countArray[]** array, i.e., execute **countArray[inputArray[i]]++** for **0 ≤ i < N**.
- Calculate the sum at every index of array **inputArray[]**.
- Create an array **outputArray[]** of size **N**. as size of input array
- Traverse array **inputArray[]** from end and update **outputArray[countArray[inputArray[i]] - 1] = inputArray[i]**. Also, update **countArray[inputArray[i]] = countArray[inputArray[i]] - 1**.

1.3.4 Complexity Analysis of Counting Sort:

K: size of counter array n : size of input & out array

- The first loop takes $O(k)$ time.
- The second loop takes $O(n)$ time.
- The third loop takes $O(k)$ time.
- The fourth loop takes $O(n)$ time.

Figure: time complexity from [Counting Sort Algorithm \(enjoyalgorithms.com\)](https://enjoyalgorithms.com)

Analysis of Time Complexity:

- **Time Complexity:** $O(N+k)$, where **N** and **k** are the size of **inputArray[]** and **countArray[]** respectively.
 - Worst-case: $O(N+k)$.
 - Average-case: $O(N+k)$.
 - Best-case: $O(N+k)$.
- **complexity Space:** $O(N+k)$, where **N** and **k** are the space taken by **outputArray[]** and **countArray[]** respectively.
- **More explain**

The overall time complexity of the counting sort = $O(k) + O(n) + O(k) + O(n) = O(k + n)$. If $k = O(n)$, the time complexity of the counting sort algorithm becomes $O(n)$.

The space complexity of the counting sort = Size of the output array $B[]$ + Size of the count array $C[] = O(n) + O(k) = O(n + k)$. If $k = O(n)$, the space complexity of the counting sort becomes $O(n)$.

Figure: time complexity from [Counting Sort Algorithm \(enjoyalgorithms.com\)](https://enjoyalgorithms.com)

1.3.5 Advantage of Counting Sort:

- 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.
- Counting sort is easy to code
- Counting sort is a **stable algorithm**.

1.3.6 Disadvantage of Counting Sort:

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

1.3.7 Implementation counting sort by c language with random values

```
1 // levan burait
2 // 12111439
3 // #1
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <time.h>
8
9
10 // Function to print the array of integer type
11 void printArray(int arr[], int n) {
12     for (int i = 0; i < n; i++) {
13         printf("%d\t", arr[i]);
14     }
15     printf("\n");
16 }
17
18 ~
```

Figure: function to print array before or after sorting by counting sort

Now I build function of counting sort : I use the above step in 1.3.2 & 1.3.3 to do it

```
// Function to sort the array using Counting Sort
// int order 1 to ascending 2 th descending
void countingSort(int arr[], int n, int max_value, int order) {
    // Create a count array to store the count of each element
    int count[max_value + 1];
    // first loop to
    // set all elements in the count array to zero at the beginning
    for (int i = 0; i <= max_value; i++) {
        count[i] = 0;
    }

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

    // 1 descending sort
    // Modify the count array to store the cumulative count
    if (order == 1) {
        // Ascending order
        // update element value in count from 0 to new value as i explain it in report
        // count[i] += count[i - 1]
        for (int i = 1; i <= max_value; i++) {
            // add the count of each element to the preceding element count
            count[i] += count[i - 1];
        }
        // // Update the count array to hold the count
    }
}
```

Figure: initialize count array & then all 0 then update it element above more explain & the ascending sort by counting sort

Then the descending

```
//2: Descending order
// Starting from the second last index (max_value - 1), subtract the count of each element
// from the succeeding element's count
else if (order == 2) { // Descending order

    for (int i = max_value - 1; i >= 0; i--) {
        // subtract the count of each element from the succeeding element's count
        count[i] += count[i + 1];
    }
}

// Create a temporary array to store the sorted output
int output[n];
for (int i = 0; i < n; i++) {
    output[i] = 0;
}

// Build the sorted array : out array
for (int i = n - 1; i >= 0; i--) {
    output[count[arr[i]] - 1] = arr[i];
    count[arr[i]]--;
}

// Copy the sorted output array to the original array : input array
for (int i = 0; i < n; i++) {
    arr[i] = output[i];
}
}
```

Figure: descending , build out array to store value after sort

- Traverse array **inputArray[]** from end and update **outputArray[countArray[inputArray[i]] - 1] = inputArray[i]**. Also, update **countArray[inputArray[i]] = countArray[inputArray[i]] - 1**.

```

83     int n;
84     // of character
85     int m;
86     printf("Enter the number of elements: ");
87     scanf("%d", &n);
88
89     // Generate random values for the array of integer
90     int arr[n];
91     int max_value = 100; // Maximum value allowed in the array
92     for (int i = 0; i < n; i++) {
93         arr[i] = rand() % (max_value + 1);
94     }
95
96
97
98     printf("Original Array: \n");
99     printArray(arr, n);
100
101     // Sort the array in ascending order
102     countingSort(arr, n, max_value, 1);
103     printf("Array sorted in ascending order: \n");
104     printArray(arr, n);
105
106     // Sort the array in descending order
107     countingSort(arr, n, max_value, 2);
108     printf("Array sorted in descending order: \n");
109     printArray(arr, n);
110
111
112

```

Figure: main function & take random value of array

```

Enter the number of elements: 12
Original Array:
41  85  72  38  80  69  65  68  96  22  49  67
Array sorted in ascending order:
22  38  41  49  65  67  68  69  72  80  85  96
Array sorted in descending order:
96  85  80  72  69  68  67  65  49  41  38  22

Process returned 0 (0x0)   execution time : 3.793 s
Press any key to continue.

```

Figure: run code small element 12

Run it by 50 random element

```

Enter the number of elements: 50
Original Array:
41  85  72  38  80  69  65  68  96  22  49  67  51  61  63  87  66  24  80  83  71  60  64  52  90
60  49  31  23  99  94  11  25  24  51  15  13  39  67  97  19  76  12  33  99  18  92  35  74
0
Array sorted in ascending order:
0  11  12  13  15  18  19  22  23  24  24  25  31  33  35  38  39  41  49  49  51  51  52  60  60
61  63  64  65  66  67  67  68  69  71  72  74  76  80  80  83  85  87  90  92  94  96  97  99
99
Array sorted in descending order:
99  99  97  96  94  92  90  87  85  83  80  80  76  74  72  71  69  68  67  67  66  65  64  63  61
60  60  52  51  51  49  49  41  39  38  35  33  31  25  24  24  23  22  19  18  15  13  12  11
0

Process returned 0 (0x0)   execution time : 3.322 s
Press any key to continue.

```

Figure: run code with 50 random element then sort ascending then descending sort

Conclusions

In summary, dynamic programming is a valuable problem-solving technique that allows for the efficient and optimal solution of complex problems by breaking them down into smaller, overlapping subproblems. It is a versatile method applicable to various domains, providing significant improvements in efficiency over brute-force methods. Understanding dynamic programming concepts and techniques is essential for tackling challenging optimization problems in computer science and other related fields.

2)Dynamic Programming

2.1 What is Dynamic Programming?

Simply put, dynamic programming is an optimization method for recursive algorithms, most of which are used to solve computing or mathematical problems.

Dynamic programming is a problem-solving technique for resolving complex problems by recursively breaking them up into sub-problems, which are then each solved individually. Dynamic programming optimizes recursive programming and saves us the time of re-computing inputs later. dynamic programming allows us to simply store the results of each step the first time and reuse it each subsequent time. so that we do not have to re-compute them when needed later.

Dynamically programmed solutions have a polynomial complexity which assures a much faster running time than other techniques like recursion or backtracking. In most cases, dynamic programming reduces time complexities, also known as big-O, from exponential to polynomial.

2.2 Mention three problems that can be solved using Dynamic Programming.

2.2.1)Knuth's Optimization in Dynamic Programming

Knuth's optimization is a very powerful tool in dynamic programming, that can be used to reduce the time complexity of the solutions primarily from $O(N^3)$ to $O(N^2)$. Normally, it is used for problems that can be solved using range DP, assuming certain conditions are satisfied.

Goal of use it : is to find the maximum or minimum value of a certain quantity

Time_complexity: $O(N^3)$

Auxiliary Space: $O(N^2)$

2.2.2) Coin Change Problem

Problem Statement

Suppose you're given an array of numbers that represent the values of each coin. Given a specific amount, find the minimum number of coins that are needed to make that amount.



Algorithm

1. Create an array of size $n+1$, where n is the target amount.
2. Set the initial value of each index in the array to be equal to the target amount. This represents the maximum number of coins needed to make up that amount, assuming we use coins of denomination 1.
3. Set the base case where $\text{array}[0] = 0$, since we don't need any coins to make 0 amount.

4. Iterate through each index starting from 1. For each index i , compare the current value in the array (initialized as the target amount) with the value at index $i-k + 1$, where k is any denomination that is less than i .
5. This comparison checks the entire array up to index $i-1$ to find the minimum number of coins required. If the value at index $i-k + 1$ is smaller than the current value at index i , update the value at index i with the new minimum value.
6. Repeat this process until you reach the target amount at the last index of the array.
7. The final value at $\text{array}[n]$ will give you the minimum number of coins needed to make up the target amount using the given denominations.

This algorithm uses the concept of dynamic programming to store and reuse the solutions to subproblems, gradually building up to the solution of the main problem in an efficient manner.

2.2.3. Fibonacci

Problem Statement

The Fibonacci Series is a sequence of integers where the next integer in the series is the sum of the previous two.

It's defined by the following recursive relation: $F(0) = 0$, $F(n) = F(n-1) + F(n-2)$, where $F(n)$ is the n th term. In this problem, we have to generate all the numbers in a Fibonacci sequence up till a given n th term.

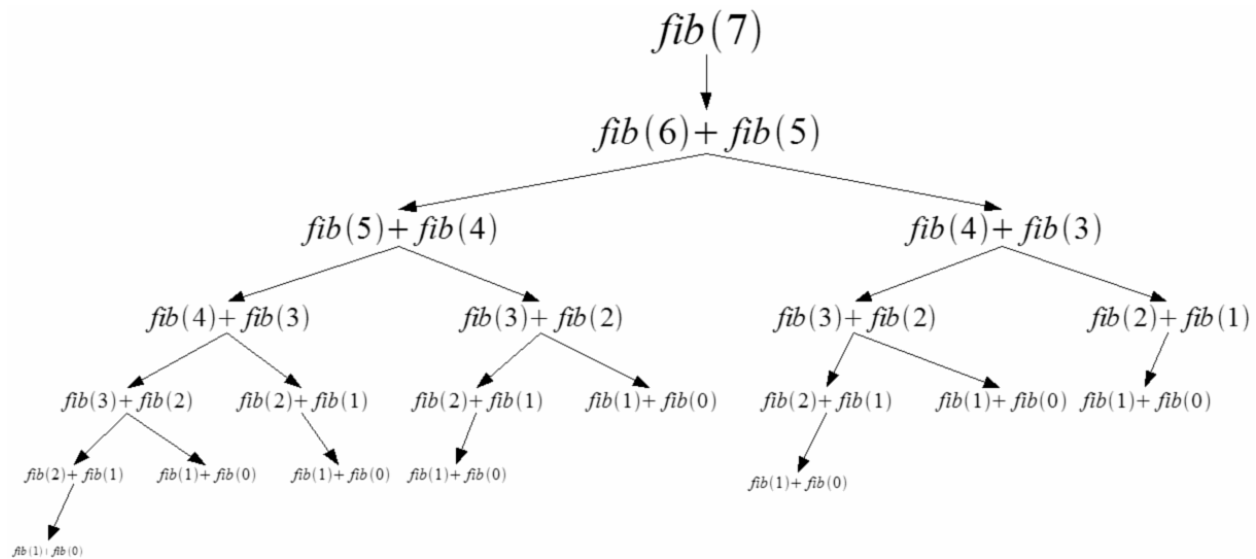


Figure: Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers

example from [Dynamic Programming: Examples, Common Problems, and Solutions \(makeuseof.com\)](https://www.makeuseof.com/dynamic-programming-examples-common-problems-and-solutions/)

algorithm:

1. Start by implementing the recursive approach for the given problem. This involves breaking down the problem into smaller subproblems using a recurrence relation. For example, in the Fibonacci sequence problem, the recurrence relation is $F(n) = F(n-1) + F(n-2)$. Implement this recursive function that returns the nth Fibonacci number.
2. Now, introduce the technique of memoization. Create an array of size $n+1$ to store the results of function calls. Initially, set all values in the array to -1, which denotes that the result for that index hasn't been calculated yet.
3. Modify the recursive function to check if the result for the given index has already been calculated.
4. If the result for the given index is already present in the array, retrieve it directly. This avoids unnecessary recalculations.
5. If the result for the given index is not present, make the recursive calls and store the result in the array before returning it.
6. Repeat steps 3-5 until the base cases are reached.
7. Finally, the result for the nth Fibonacci number will be stored in the array at position n .

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

2.3.1 Techniques to solve Dynamic Programming Problems:

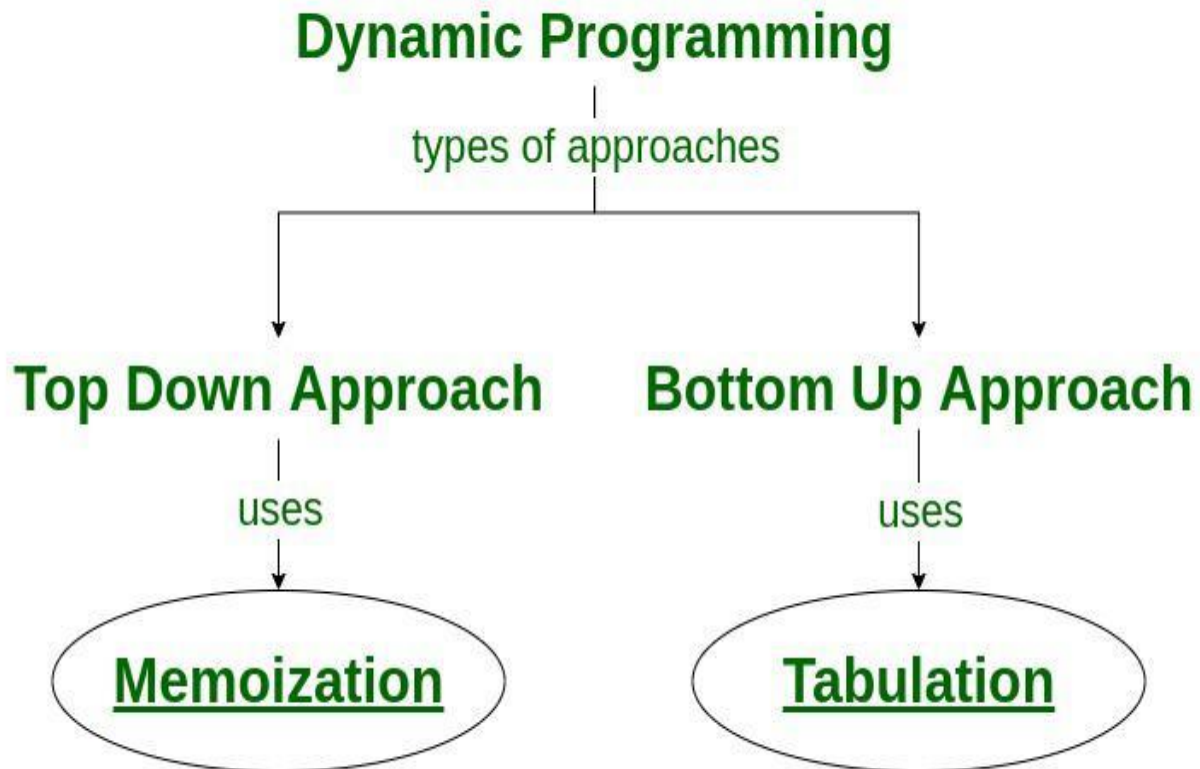


FIGURE: [Dynamic Programming \(DP\) Tutorial with Problems - GeeksforGeeks](#)

2.3.1. Top-Down(Memoization):

Break down the given problem in order to begin solving it. If you see that the problem has already been solved, return the saved answer. If it hasn't been solved, solve it and save it. This is usually easy to think of and very intuitive, This is referred to as [Memoization](#).

2.3.2 Bottom-Up(Dynamic Programming):

Analyze the problem and see in what order the subproblems are solved, and work your way up from the trivial subproblem to the given problem. This process ensures that the subproblems are solved before the main problem. This is referred to as [Dynamic Programming](#).

How to Solve Dynamic Programming Problems?



Figure: [Dynamic Programming in Python: Top 10 Problems \(with code\) \(favtutor.com\)](#)

In my code to solve DP problems:

- 1) Recognize the DP problem
- 2) Identify problem variables
- 3) Express the recurrence relation
- 4) Identify the base case
- 5) Decide the iterative or recursive approach
- 6) Add memoization

Memoization is the process of storing the result of the subproblem and calling them again when a similar subproblem is to be solved. This will reduce the time complexity of the problem. If we do not use memorization, similar subproblems are repeatedly solved which can lead to exponential time complexities.

2.3.2 Nth Fibonacci Number

The Fibonacci numbers are the numbers in the following integer sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

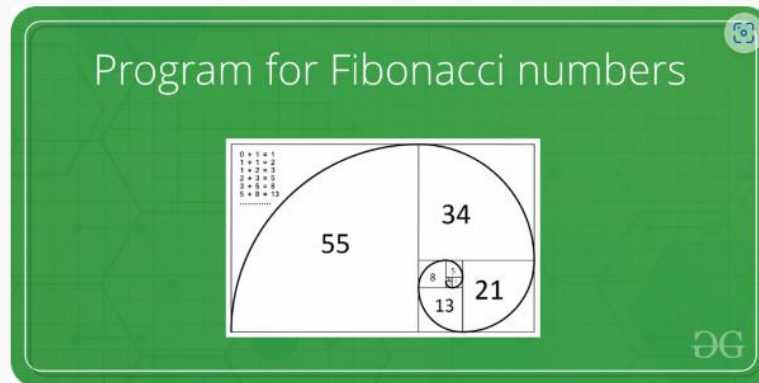


FIGURE: [Nth Fibonacci Number - GeeksforGeeks](#)

2.3.2.1 Iterative Approach to Find and Print Nth Fibonacci Numbers:

Time

Complexity: $O(n)$

Auxiliary Space: $O(1)$

```
//LEYAN BURAIT
// 1211439
// #1
#include <stdio.h>
#include <stdlib.h>

// Fibonacci Series using Space Optimized Method
int fib(int n)
{
    int a = 0, b = 1, c, i;
    if (n == 0)
        return a;
    for (i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main()
{
    int n;
    printf("enter the number of n ");
    scanf("%d", &n);
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

```
enter the number of n 12
144
Process returned 0 (0x0)   execution time : 6.046 s
Press any key to continue.
```

2.3.2.2 Recursion Approach to Find and Print Nth Fibonacci Numbers:

Time Complexity: Exponential, as every function calls two other functions.

Auxiliary space complexity: $O(n)$, as the maximum depth of the recursion tree is n .

```
// Recursion Approach to Find and Print Nth Fibonacci Numbers:
int fib_Recursion(int n)
{
    if (n <= 1)
        return n;
    return fib_Recursion(n - 1) + fib_Recursion(n - 2);
}

int main()
{
    int n ;
    printf("enter the number of n ");
    scanf("%d", &n);
    printf("%d", fib(n));
    getchar();
    printf("%dth Fibonacci Number: %d", n, fib_Recursion(n));
    return 0;
}
```

```
enter the number of n
12
Fibonacci Series using Space Optimized Method
144
Recursion Approach to Find and Print Nth Fibonacci Numbers
12th Fibonacci Number: 144
Process returned 0 (0x0)   execution time : 1.627 s
Press any key to continue.
```

Figure: green run of space optimized , yellow run of recursion

2.3.2.3 Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers:

Time complexity: $O(n)$ for given n

Auxiliary space: $O(n)$

```
26 //
27 int fib_DP(int n)
28 {
29     /* Declare an array to store Fibonacci numbers. */
30     int f[n + 2]; // 1 extra to handle case, n = 0
31     int i;
32
33     /* 0th and 1st number of the series are 0 and 1*/
34     f[0] = 0;
35     f[1] = 1;
36
37     for (i = 2; i <= n; i++) {
38         /* Add the previous 2 numbers in the series
39            and store it */
40         f[i] = f[i - 1] + f[i - 2];
41     }
42
43     return f[n];
44 }
45
46 int main()
47 {
```

Figure: Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers

```
45 //
46 int main()
47 {
48     int n ;
49     printf("enter the number of n \n");
50     scanf("%d", &n);
51     printf("Fibonacci Series using Space Optimized Method\n\n") ;
52     printf("%d\n", fib(n));
53     getchar();
54     printf("Recursion Approach to Find and Print Nth Fibonacci Numbers\n\n") ;
55     printf("nth Fibonacci Number: %d\n", n, fib_Recursion(n));
56     printf("Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers") ;
57     printf("%d", fib_DP(n));
58     getchar();
59
60     return 0;
61 }
62
```

Figure: main function of all : recursion ,dynamic & space optimized

```

enter the number of n
12
Fibonacci Series using Space Optimized Method

144
Recursion Approach to Find and Print Nth Fibonacci Numbers

12th Fibonacci Number: 144
Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers
144

Process returned 0 (0x0)   execution time : 8.861 s
Press any key to continue.
|

```

Figure: Run of all : recursion ,dynamic & space optimized & blue is the DP

Now if I run each of one alone then the execution time will be?

1 Iterative Approach to Find and Print Nth Fibonacci Numbers

```

enter the number of n 12
144
Process returned 0 (0x0)   execution time : 6.046 s
Press any key to continue.

```

Execution time = 6.046 s

2) Recursion Approach to Find and Print Nth Fibonacci Numbers

```

.c
enter the number of n
12
Recursion Approach to Find and Print Nth Fibonacci Numbers

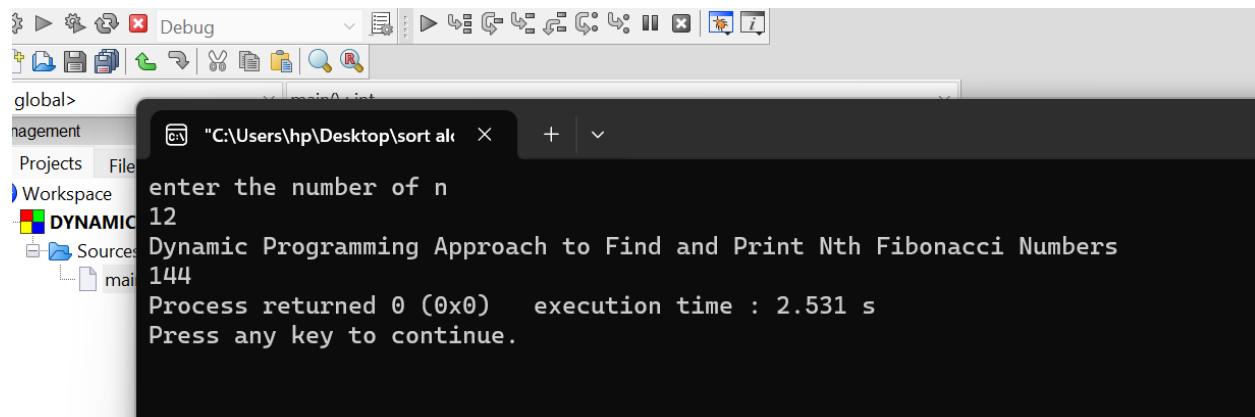
12th Fibonacci Number: 144

Process returned 0 (0x0)   execution time : 3.215 s
Press any key to continue.
|

```

Execution time = 3.215 s

3) Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers:



```
global>
nagement
Projects File
Workspace
DYNAMIC
Sources
mail
"C:\Users\hnp\Desktop\sort alk" x + v
enter the number of n
12
Dynamic Programming Approach to Find and Print Nth Fibonacci Numbers
144
Process returned 0 (0x0)   execution time : 2.531 s
Press any key to continue.
```

Execution time = 2.531 s

After running each alone then the Execution time of dynamic program = 2.531s less than other one then become the **Recursion Approach to Find and Print Nth Fibonacci Numbers = 3.215 s** then the last one is space optimize = 6.046 s: **Iterative Approach to Find and Print Nth Fibonacci Numbers**

2.2.4 Advantages & Disadvantages

The biggest advantages of Dynamic Programming is that you can obtain the both local and total optimal solution, it reduces the lines of the code, & it is applicable to both linear and non-linear problem.

However, DP makes unnecessary memory utilization. As you make use of the table while solving the problem, it needs a lot of memory space for storage purposes

2.2.5 When should DP be used to solve a problem?

The powerful method of dynamic programming can be applied to a variety of issues. However, there are several requirements that must be satisfied before choosing to utilize dynamic programming to solve an issue, therefore it isn't always the best option.

Overlapping Subproblems: The existence of overlapping subproblems in the original problem is one of the essential conditions for dynamic programming. In other words, the issue can be divided into more manageable subproblems that are used repeatedly in the resolution.

Optimal substructure: The problem's optimal substructure is another prerequisite for dynamic programming. In other words, the problem's optimal solution can be created from the answers to its individual subproblems.

Memory work can enhance performance: By saving the outcomes of pricey function calls in memory, dynamic programming methods can run better. Memorization can drastically lower the number of function calls necessary if the same subproblem appears numerous times in the solution.

Complexity in polynomial time: Algorithms for dynamic programming should have polynomial time complexity. That is, the amount of time needed to solve the issue should be inversely correlated with some polynomial function of the size of the input.

All my main.c I work it I will store it in this link in google drive

https://drive.google.com/file/d/1a-TLegRriTlh2NfwGH4SZvAH156EgcfJ/view?usp=drive_link

Name: Leyan Burait

ID:1211439

DR.Ahmed Abusnaina

#1

4)References:

1)link references of Gnome Sort

*how gnome work

<https://youtu.be/43noNfrbEnQ?si=BQmzwGSBLTtIZ7-v>

[Gnome Sort - GeeksforGeeks](#)

*complexity

[Gnome Sort \(opengenius.org\)](#)

2) link references of comb Sort

*this video help me to be able write the code in c

https://youtu.be/pUClrUZhL6I?si=HN3_cGn0AzKmQCaN

*how comb sorting work

[Comb Sort Explained | Baeldung on Computer Science](#)

*complexity of comb sort: [Comb Sort - GeeksforGeeks](#)

3)link references of counting Sort

*video helps me to understand the counting sort

<https://youtu.be/pEJiGC-ObQE?si=dmAkt6BuS8E8yhkD>

*What is Counting Sort? & how work?

[Counting Sort - Data Structures and Algorithms Tutorials - GeeksforGeeks](#)

*Image link of steep[1,12] of counting sort

1: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182425/1.png>

2: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182436/2.png>

3: <https://media.geeksforgeeks.org/wp-content/uploads/20230922132754/3.png>

4: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182646/4.png>

5: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182656/5.png>

6: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182724/6.png>

7: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182741/7.png>

8: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182752/8.png>

9: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182807/9.png>

10: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182827/10.png>

11: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182855/11.png>

12: <https://media.geeksforgeeks.org/wp-content/uploads/20230920182910/12.png>

4)reference of dynamic programming

[Knuth's Optimization in Dynamic Programming - GeeksforGeeks](#)

[Dynamic Programming: Examples, Common Problems, and Solutions \(makeuseof.com\)](#)

[The 13 Best Browser IDEs Every Programmer Should Know About \(makeuseof.com\)](#)

Example code: https://youtu.be/Hdr64lKQ3e4?si=-ekhDltfysca_-Vu

[Dynamic Programming in Python: Top 10 Problems \(with code\) \(favtutor.com\)](#)

[Nth Fibonacci Number - GeeksforGeeks](#)

<https://www.geeksforgeeks.org/program-for-nth-fibonacci-number>