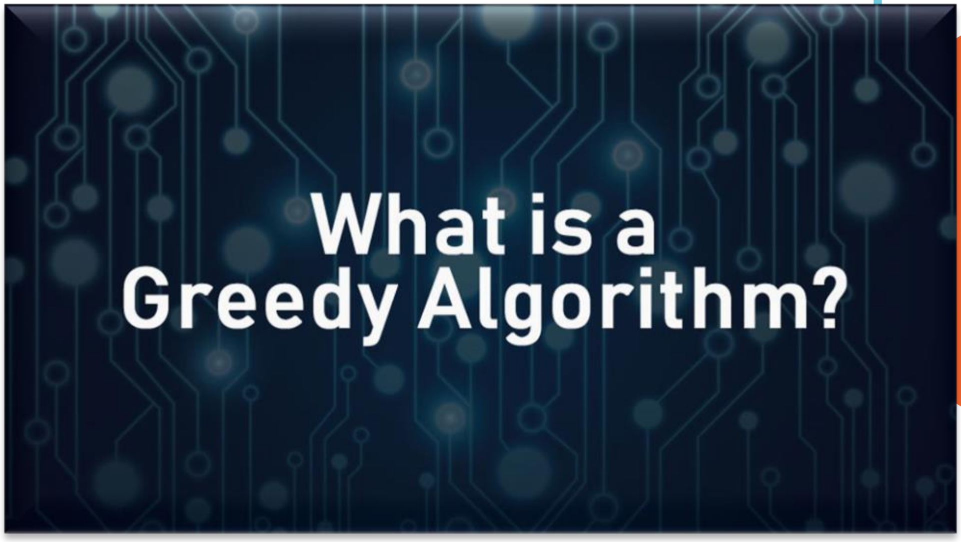


DSA Group Assignment

Greedy Algorithms to Solve Tasks

While the optimal solution to each smaller instance will provide an immediate output, the algorithm doesn't consider the larger problem as a whole. ... Greedy algorithms work by recursively constructing a set of objects from the smallest possible constituent parts.

Name	**W.P PALLEWATTA**	**A.U.S PATHINAYAKE**	**M.A.L PEIRIS**
INDEX	18001149	18001157	18001165

DSA Group Assignment

Greedy Algorithms to Solve Tasks

Table of Contents

Greedy Algorithms

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem. Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph.

However, in many problems, a greedy strategy does not produce an optimal solution.

Question No 1

Sandun is a wine lover. He had to be in his home for three months due to the Covid-19 situation. Sandun wants to buy some wine since the curfew has been lifted. He goes to his favorite wine yard with some empty bottles. Wine yard owner is pleased to see Sandun. Hence, he offers Sandun free wine on one condition. Sandun can fill only the bottles he carries in his bag. Wine barrels in the wine yard have a tag in each one of them which has two values containing the number of bottles in the barrel and the price of the whole barrel. You have to build a program to help Sandun to take maximum valued wine from this offer. Given the number of bottles that he brought and the information on the wine barrels, you have to find the maximum value of wines Sandun gets from this deal. Please note that he cannot fill any of the bottles with more than one type of wine. Assume that the bottle is a 750ml glass bottle.

Input Format

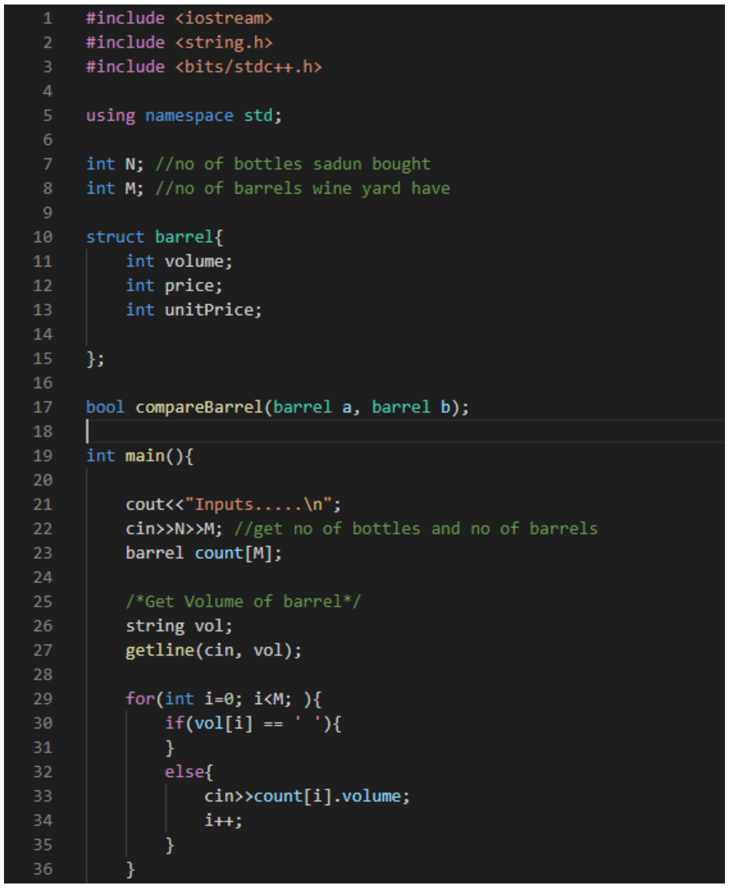
First Line contains two integers N and M. N is the number of bottles Sandun brought and M is the number of barrels in the wine yard. Second-line contains M space-separated integers contain the volume of barrels in bottles. The third line contains the M space-separated integers containing the price of each barrel.

Output Format

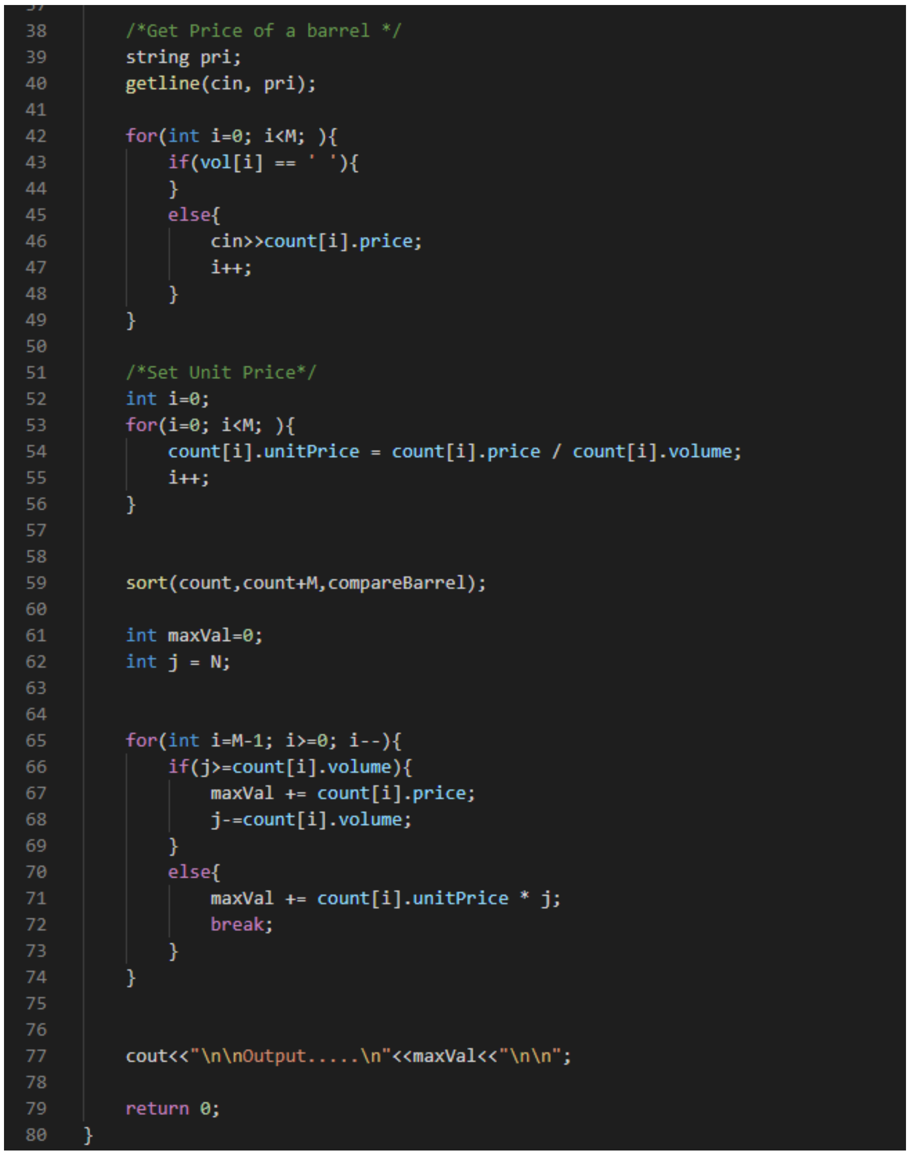
One integer containing the maximum value of wines Sandun gets from the offer.

Solution

Code Implementation



```cpp
#include <iostream>
#include <string.h>
#include <bits/stdc++.h>

using namespace std;

int N; //no of bottles sadun bought
int M; //no of barrels wine yard have

struct barrel{
    int volume;
    int price;
    int unitPrice;

};

bool compareBarrel(barrel a, barrel b);

int main(){

    cout<<"Inputs.....\n";
    cin>>N>>M; //get no of bottles and no of barrels
    barrel count[M];

    /*Get Volume of barrel*/
    string vol;
    getline(cin, vol);

    for(int i=0; i<M; ){
        if(vol[i] == ' '){
        }
        else{
            cin>>count[i].volume;
            i++;
        }
    }
```



```
    /*Get Price of a barrel */
    string pri;
    getline(cin, pri);

    for(int i=0; i<M; ){
        if(vol[i] == ' '){
        }
        else{
            cin>>count[i].price;
            i++;
        }
    }

    /*Set Unit Price*/
    int i=0;
    for(i=0; i<M; ){
        count[i].unitPrice = count[i].price / count[i].volume;
        i++;
    }


    sort(count,count+M,compareBarrel);

    int maxVal=0;
    int j = N;


    for(int i=M-1; i>=0; i--){
        if(j>=count[i].volume){
            maxVal += count[i].price;
            j-=count[i].volume;
        }
        else{
            maxVal += count[i].unitPrice * j;
            break;
        }
    }


    cout<<"\n\nOutput.....\n"<<maxVal<<"\n\n";

    return 0;
}
```

Code Explanation

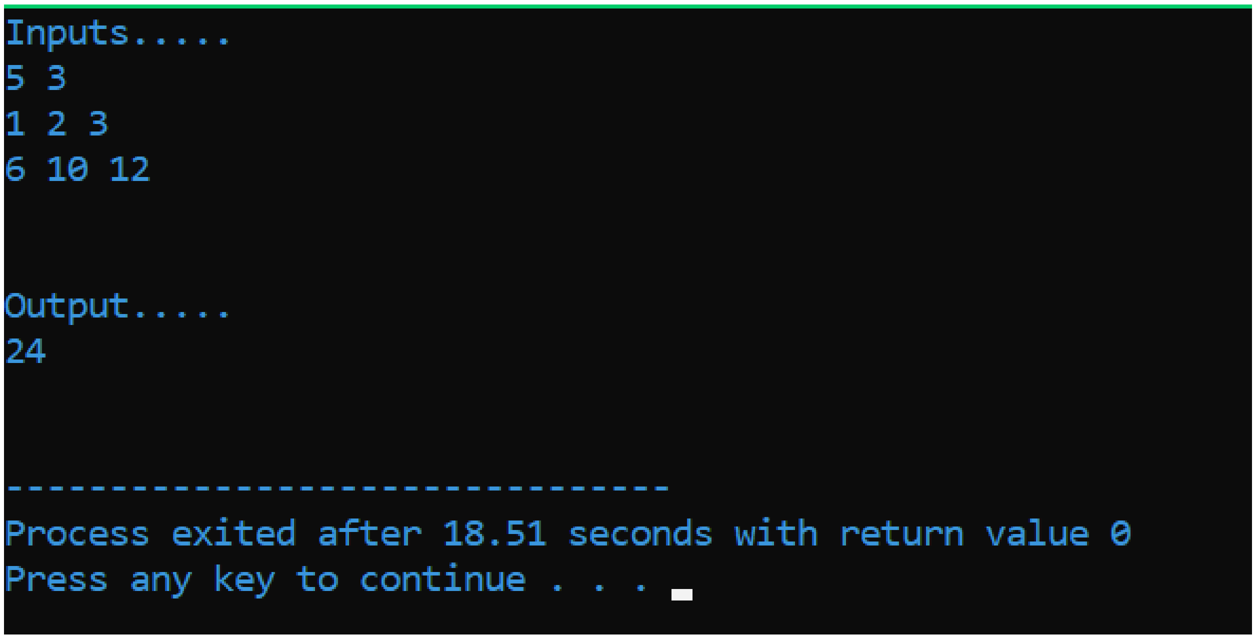
I used array of structure as data structure. First, I get N and M as user inputs. Then create an Array of structure then get volume of the barrels and price of the barrels as user inputs. Then I calculated unit price of the wine bottle. Then I sort the array of structure according to the unit price. Finally, I calculate maximum value of wines Sandun gets from this deal.

Greedy components

- **Candidate Set:** Array of number of bottles in each barrel
- **Objective Function:** Total number of bottles taken from the wine yard such that the value is maximized.
- **Feasibility Function:** The bottles taken cannot exceed the number of bottles in Sandun's bag.
- **Selection Function:** Selects the maximum number of bottles to be taken.

• **Solution Function:** Indicates the final solution.

Output with an Example.



```
Inputs.....
5 3
1 2 3
6 10 12


Output.....
24



--------------------------------
Process exited after 18.51 seconds with return value 0
Press any key to continue . . .
```

1) **Greedy Choice Property**

According to the problem, each volume of the barrel and price will be considered and cost per wine bottle being calculated. The intention is to get the height value for the vine.
the best choice at the moment without taking the remaining sub problems into consideration, reducing the problem instance to a smaller one.
Therefore, the problem has the greedy choice property.

2) **Optimal Substructure Property**

A problem exhibits optimal substructure property if the optimal solution to the sub problem yields an optimal solution to the original problem.

So, first the cost per unit in the barrel is being calculated my algorithm, then to maximize the profit from each barrel what will profit the owner is being reconsider. Which optimally substructure my problem by each stem to give as solution

Question No 2

Manel is a schoolteacher. She wants to give some face masks to the students in her class. All the students sit in a line and each of them has a score according to his or her performance in the class. Students can’t change their seating order. Manel wants to give at least 1 face mask to each student. If two students sit next to each other, then the one with the higher score must get more face masks. Note that when two children have equal score, they can have different number of face masks. Manel wants to minimize the total number of face masks she must buy.

For example, assume her students' scores are [4, 6, 4, 5, 6, 2]. She gives the students face mask in the following minimal amounts: [1, 2, 1, 2, 3, 1]. She must buy a minimum of 10 face masks.
You have to develop a program to help her find the minimum number of face masks she should buy.

Input Format

The first line contains an integer, **n**, the number of students to give face masks.
Each of the next **n** lines contains an integer **score[i]** indicating the score of the student at position **i**.

Constraints

1 ≤ n ≤ 105
1 ≤ score[i] ≤ 105

Output Format

Output a single line containing the minimum number of face masks Manel must buy.

Greedy Component in the Problem

The key ingredients for a greedy algorithm to solve an optimization problem are:

3) Greedy Choice Property

According to the problem, each student is allocated a minimum number of masks based only on the scores of the students seated immediately next to them. Therefore, in each instance, the number of masks allocated to a student is the minimum number that can be allocated at the given moment, without taking the scores of the remaining students (Students who are not seated immediately to the left or right) into consideration. i.e., the best choice at the moment without taking the remaining sub problems into consideration, reducing the problem instance to a smaller one.
Therefore, the problem has the greedy choice property.

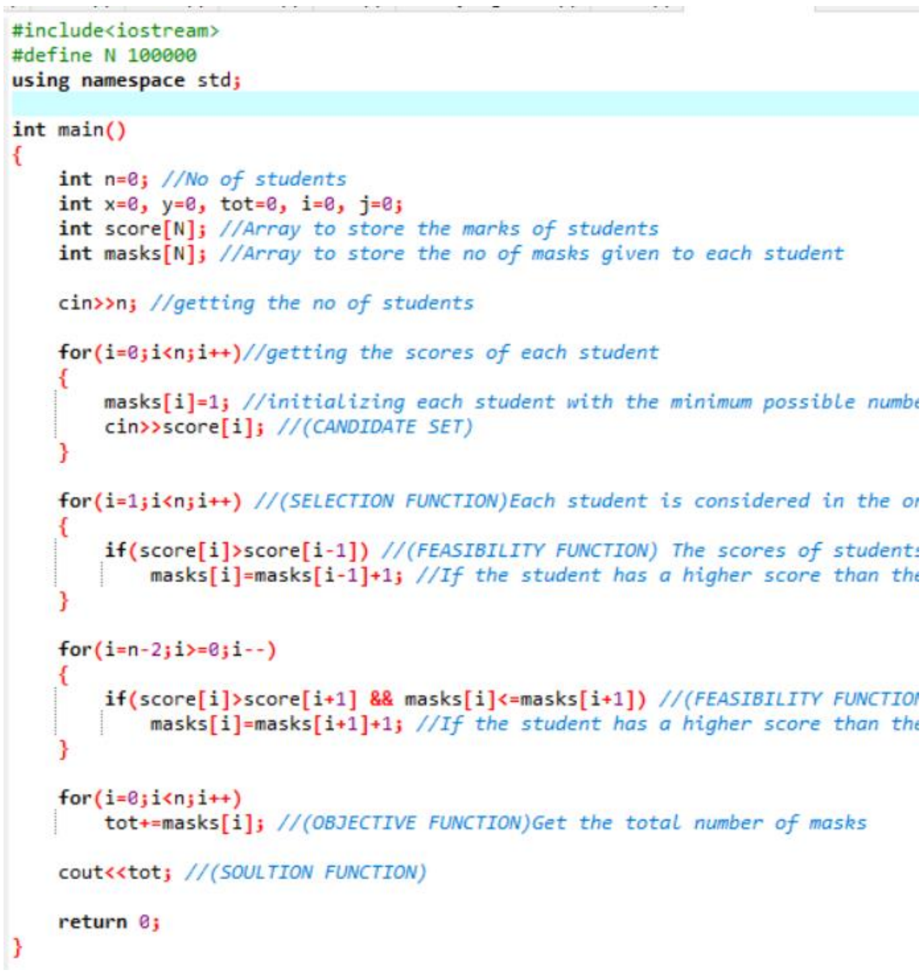
4) Optimal Substructure Property

A problem exhibits optimal substructure property if the optimal solution to the sub problem yields an optimal solution to the original problem.

At each step, the student considered will be allocated the minimum possible number of masks based on the score and the number of masks allocated to the students on either side. Thus, the number of masks given to each student is minimized, and as a result, the total number of masks that should be bought by the teacher will be minimized.
Therefore, the problem has the optimal substructure property.

Solution

Code Implementation



```
#include<iostream>
#define N 100000
using namespace std;

int main()
{
    int n=0; //No of students
    int x=0, y=0, tot=0, i=0, j=0;
    int score[N]; //Array to store the marks of students
    int masks[N]; //Array to store the no of masks given to each student

    cin>>n; //getting the no of students

    for(i=0;i<n;i++)//getting the scores of each student
    {
        masks[i]=1; //initializing each student with the minimum possible numbe
        cin>>score[i]; //(CANDIDATE SET)
    }

    for(i=1;i<n;i++) //(SELECTION FUNCTION)Each student is considered in the or
    {
        if(score[i]>score[i-1]) //(FEASIBILITY FUNCTION) The scores of student:
            masks[i]=masks[i-1]+1; //If the student has a higher score than the
    }

    for(i=n-2;i>=0;i--)
    {
        if(score[i]>score[i+1] && masks[i]<=masks[i+1]) //(FEASIBILITY FUNCTIOI
            masks[i]=masks[i+1]+1; //If the student has a higher score than the
    }

    for(i=0;i<n;i++)
        tot+=masks[i]; //(OBJECTIVE FUNCTION)Get the total number of masks

    cout<<tot; //(SOULTION FUNCTION)

    return 0;
}
```

Explanation

The greedy strategy is to consider each student and allocate the minimum possible number of masks, based on the comparative scores of the students on either side.

- Allocate all students with the minimum possible number of masks, i.e., 1.

- Iterate through the set of scores from beginning to end.
    - If the student’s score is higher than the score of the student on the left (if any) the number of masks allocated to that student is one more than the number of masks allocated to the student on the left.
- Iterate through the set of scores A from end to beginning.

- If the student's score is higher than the score of the student on the right (if any) and if the number of masks given to that student is less than or equal to the masks given to the student on the right, the number of masks allocated to the student is one more than the number of masks allocated to the student on the right.

(A student is allocated only one mask more than that of the student on the left/right to ensure that the number of masks allocated to each student is minimum)

- Iterate through the set of masks and get the total number of masks

- Return the total as the minimum number of masks that should be bought

The time complexity is O(n)

Components of the greedy algorithm

1) **Candidate Set:** The array *marks* is used to store the scores of each student.

2) **Feasibility Function:** The score of each student is compared with the score of the student seated immediately to the left and right to determine whether that student should be allocated a higher number of masks.

3) **Selection Function:**
For each student, if his/her score is greater than the score of the student seated immediately to the left, then the student is allocated a higher number of masks (one more) than the student on the left.

- Then, the score of each student is compared with the score of the student seated immediately to the right. If the student's score is higher, then the student is allocated a higher number of masks (one more) than the student on the right.

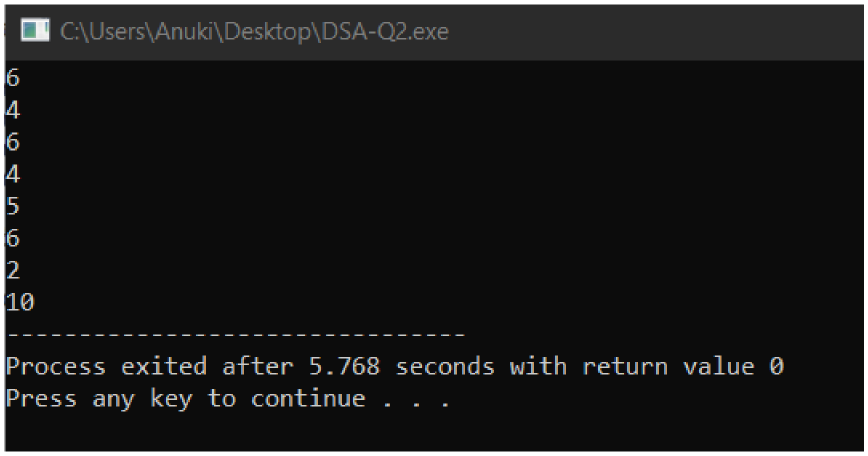
4) **Objective Function:** After allocating masks for each student is complete. We iterate through the *masks* array and add the number of masks allocated to each student to obtain the sum.

5) **Solution Function:** Finally, the sum of masks is returned as the minimum number of masks to be bought

Output with an Example.

1) **The number of students to give face masks** – 6

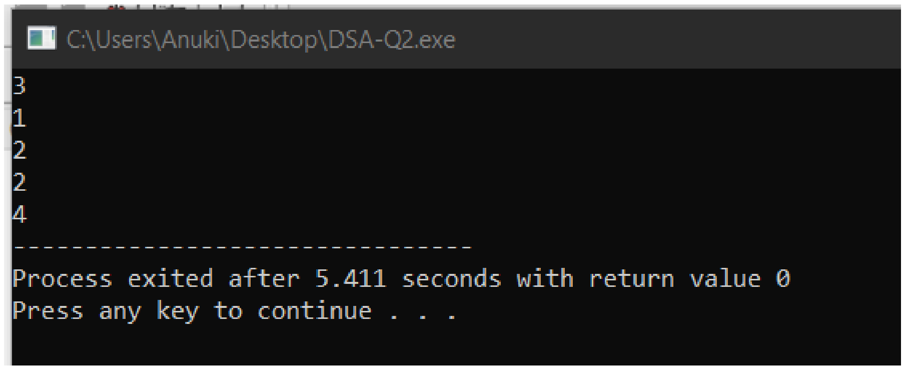
   Score of each student – [4, 6, 4, 5, 6, 2]



```
C:\Users\Anuki\Desktop\DSA-Q2.exe
6
4
6
4
5
6
2
10
--------------------------------
Process exited after 5.768 seconds with return value 0
Press any key to continue . . .
```

2) **The number of students to give face masks** – 3

   Score of each student – [1, 2, 2]



```
C:\Users\Anuki\Desktop\DSA-Q2.exe
3
1
2
2
4
--------------------------------
Process exited after 5.411 seconds with return value 0
Press any key to continue . . .
```

Question No 3

Seetha works for an online store that uses air mail to send ordered products. But due to the covid-19 situation they have to ship the products using containers. Her task is to the determine the lowest cost way to combine her orders for shipping. She has a list of product weights. The shipping company has a requirement that all products loaded in a container must weigh less than or equal to 4 units plus the weight of the minimum weight product. Otherwise they will not take responsibility for addresses getting defaced. All products meeting that requirement will be shipped in one container.

You have to develop a program to find minimum number of containers required.

For example, there are products with weights **w=[1,2,3,4,5,10,11,12,13]**. This can be broken into two containers: **[1,2,3,4,5]** and **[10,11,12,13]**. Each container will contain products weighing within 4 units of the minimum weight product.

Input Format

The first row includes an integer n, the number of shipping products.

The next line includes the weight sequence of products in n space-separated integers.

Constraints

1 ≤ n ≤ 105

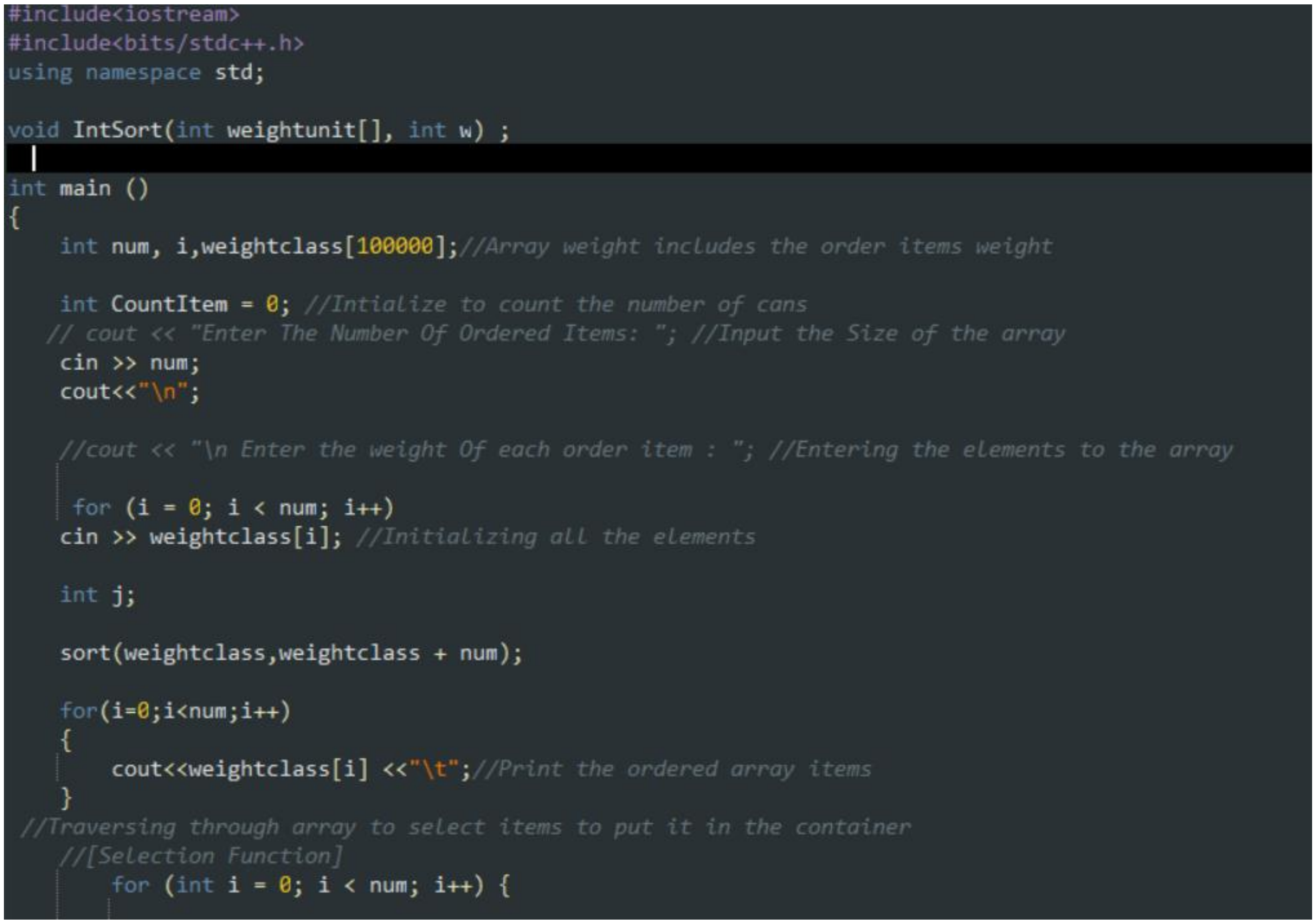
0 ≤ weight ≤ 104

Output Format

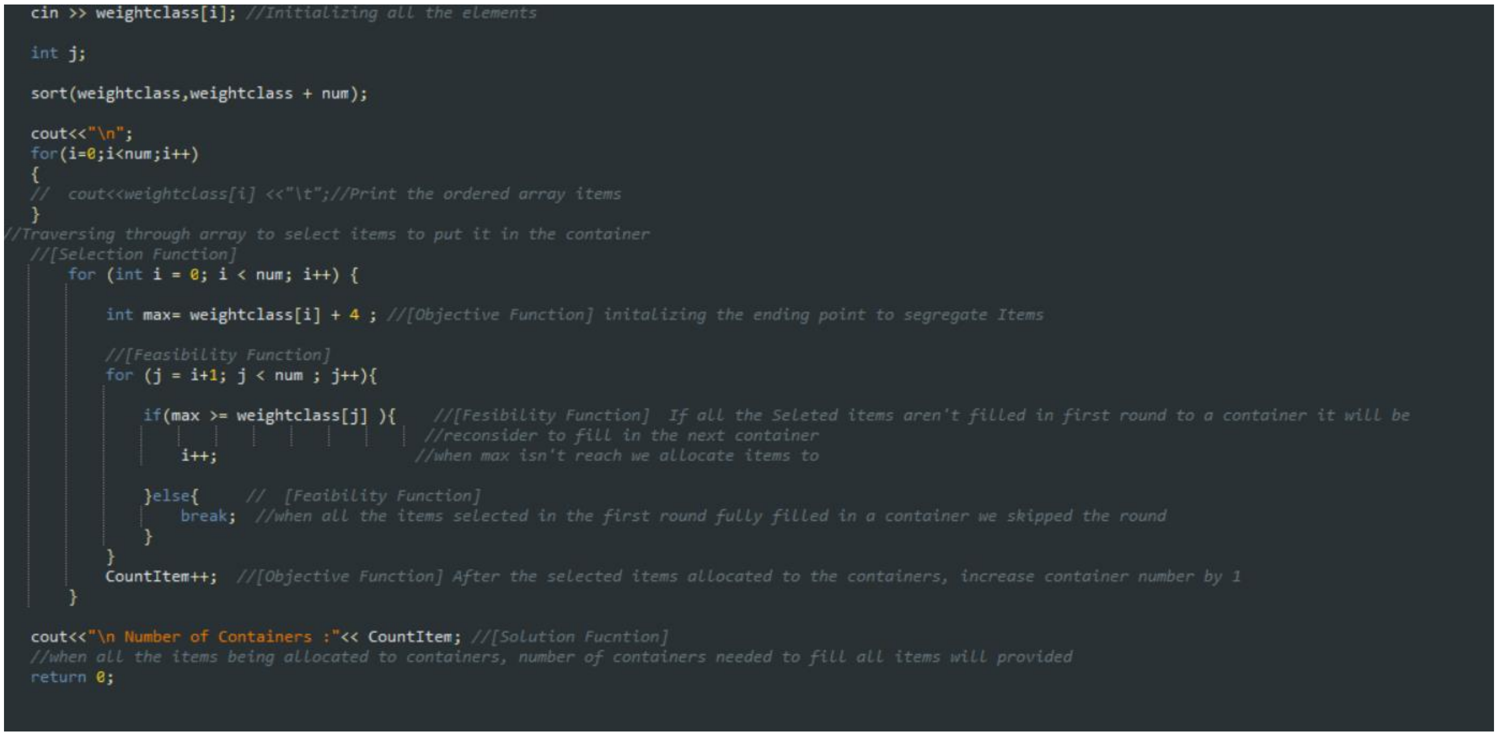
Return single integer containing minimum number of containers needed.

Solution

Code Implementation



```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

void IntSort(int weightunit[], int w) ;

int main ()
{
    int num, i,weightclass[100000];//Array weight includes the order items weight

    int CountItem = 0; //Intialize to count the number of cans
  // cout << "Enter The Number Of Ordered Items: "; //Input the Size of the array
    cin >> num;
    cout<<"\n";

    //cout << "\n Enter the weight Of each order item : "; //Entering the elements to the array

     for (i = 0; i < num; i++)
    cin >> weightclass[i]; //Initializing all the elements

    int j;

    sort(weightclass,weightclass + num);

    for(i=0;i<num;i++)
    {
        cout<<weightclass[i] <<"\t";//Print the ordered array items
    }
 //Traversing through array to select items to put it in the container
    //[Selection Function]
        for (int i = 0; i < num; i++) {
```



```
    cin >> weightclass[i]; //Initializing all the elements

    int j;

    sort(weightclass,weightclass + num);

    cout<<"\n";
    for(i=0;i<num;i++)
    {
    //  cout<<weightclass[i] <<"\t";//Print the ordered array items
    }
//Traversing through array to select items to put it in the container
    //[Selection Function]
        for (int i = 0; i < num; i++) {

            int max= weightclass[i] + 4 ; //[Objective Function] initalizing the ending point to segregate Items

            //[Feasibility Function]
            for (j = i+1; j < num ; j++){

                if(max >= weightclass[j] ){    //[Fesibility Function]  If all the Seleted items aren't filled in first round to a container it will be
                                              //reconsider to fill in the next container
                    i++;                     //when max isn't reach we allocate items to

                }else{     //  [Feaibility Function]
                    break;  //when all the items selected in the first round fully filled in a container we skipped the round
                }
            }
            CountItem++;  //[Objective Function] After the selected items allocated to the containers, increase container number by 1
        }

    cout<<"\n Number of Containers :"<< CountItem; //[Solution Fucntion]
    //when all the items being allocated to containers, number of containers needed to fill all items will provided
    return 0;
```

Code Explanation

In my Solution I have Used **Array** Data structure because only insertion function is wanted. So, the complexity is O (n).
Also used the predefined sort in CPP which is quicksort. Used to do the sorting.

<u>Greedy Properties: -</u>

1)Optimal Substructure:

In this Problems I have selected first item and add 4 weight units to find max. From each single iteration and find out with in the weight range. Then solution optimized to allocated to using minimum number of containers. The advantage is weights that are in same range will be allocated to same container and not be reconsider which give us optimal solution. (**Created sub Problem to Allocate containers** – Selecting first min weight item which is not allocated then using the max weight if the items can be founded in between that range allocate container)

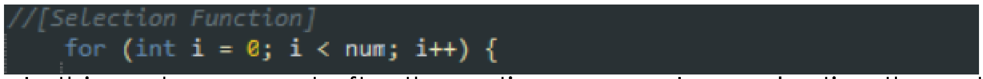
2)Greedy Choice Property

To store the weight of the items and selected the Containers to the range of the weight that is being search in each iteration. Ignored Allocated items and moved to the next minimum weighted item do the same process. When the items between 4-unit range being allocated it that items will not check and moved to next min. (Best solution to allocate same range weighted items to a container)
(Not checking the entire item weight array)

Mechanism (Components of Greedy Algorithm)

I have created array store the items weight and sorted before the greedy algorithm apply. Then according to greedy algorithm characteristics, I have introduced the functions to solve the main problem dividing into sub problems.

So, in the **1) Selection Function**

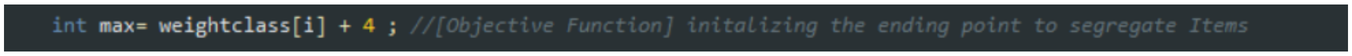


```
//[Selection Function]
    for (int i = 0; i < num; i++) {
```

In this code segment after the sorting process I am selecting the entire array that includes weights of each items.

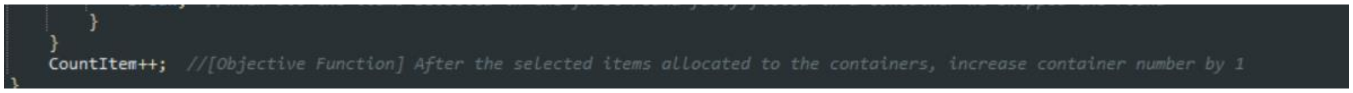
2)Objective Function

In this code segment my focus to select first non-allocated item and set the maximum weight to segregate the items within that range.



```
    int max= weightclass[i] + 4 ; //[Objective Function] initalizing the ending point to segregate Items
```

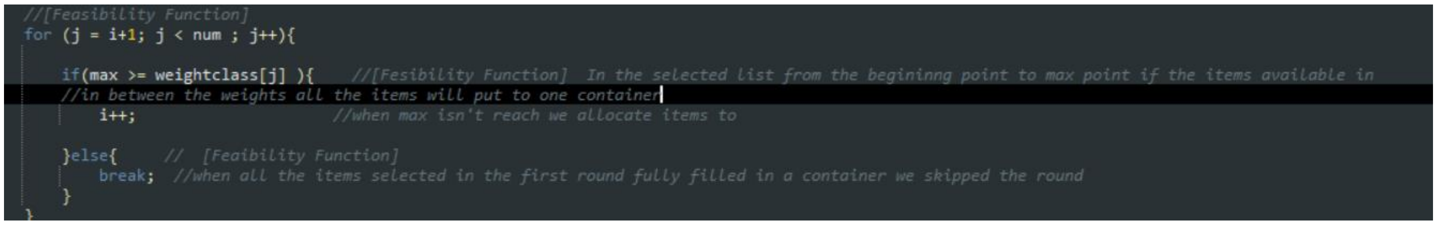
After the feasibility function is executed according to constrains it will allocate the container for the selected items.



```
        }
    }
    CountItem++;  //[Objective Function] After the selected items allocated to the containers, increase container number by 1
}
```

3)Feasibility Function

In this function the main problem being solved using a loop.

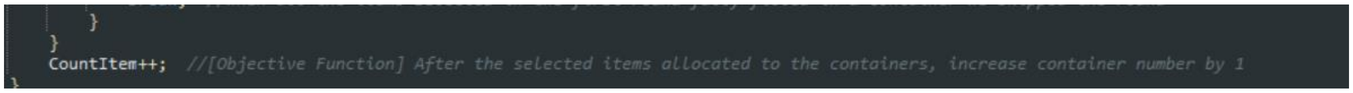


```
//[Feasibility Function]
for (j = i+1; j < num ; j++){

    if(max >= weightclass[j] ){    //[Fesibility Function]  In the selected list from the begininng point to max point if the items available in
    //in between the weights all the items will put to one container
        i++;                        //when max isn't reach we allocate items to

    }else{    //  [Feaibility Function]
        break;  //when all the items selected in the first round fully filled in a container we skipped the round
    }
}
```

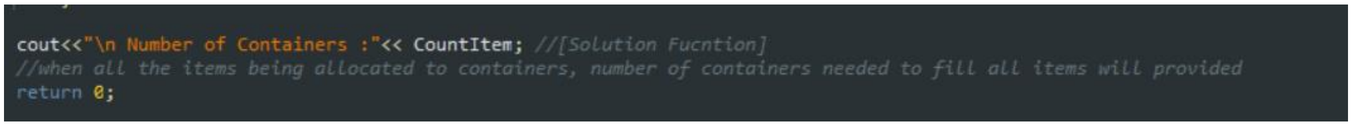
In this code segment I have initiated the loop to segregate the items that is in between the initiated min product and max (which is 4 units + min at the current Occurrence). If it is allocated the function points to the last non allocated minimum weighted product without checking other items. So, this will minimize the time complexity as well and guarantee to provide minimum containers to given order of products.

4)Solution Function

This code segment act as **objective** by allocating selected items to the container and also act as **solution function** by keep in tabs of number of containers in each iteration.



```
        }
    }
    CountItem++;  //[Objective Function] After the selected items allocated to the containers, increase container number by 1
}
```



```
cout<<"\n Number of Containers :"<< CountItem; //[Solution Fucntion]
//when all the items being allocated to containers, number of containers needed to fill all items will provided
return 0;
```

In this function when we executed the greedy algorithm after allocating every item to a container according to the constraint. This function returns the Final Solution.

5)Candidate Set: - I have used "weightclass array" to store items weight in the given Order.

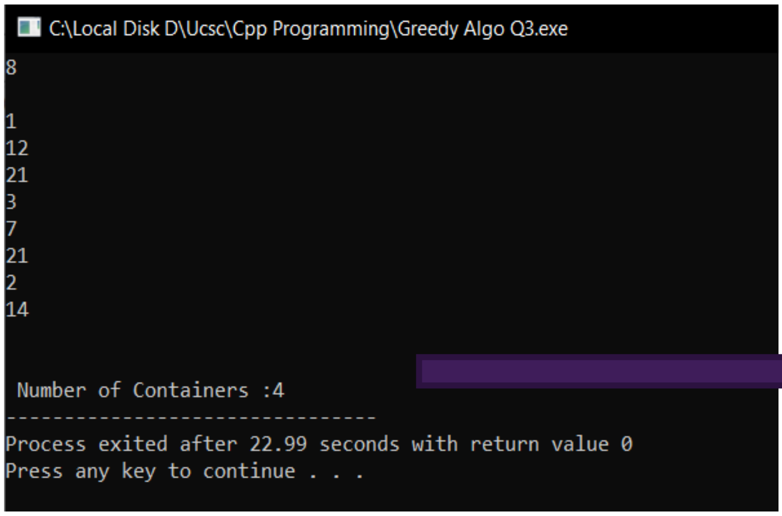
So, this is the method that I have Solved my problem using the Greedy Approach. When we consider the Time Complexity in **sorting worst case O (n^2)** and using **double loops** to check the constraints and allocated in **worst case close to O(n^2).**

So, the Overall Complexity Close to **O (n^2)**.

Output with an Example.

1) **Number of Items in the Order** – 8

   Weights in integer: - (1, 12, 21, 3, 7, 21, 2, 14)



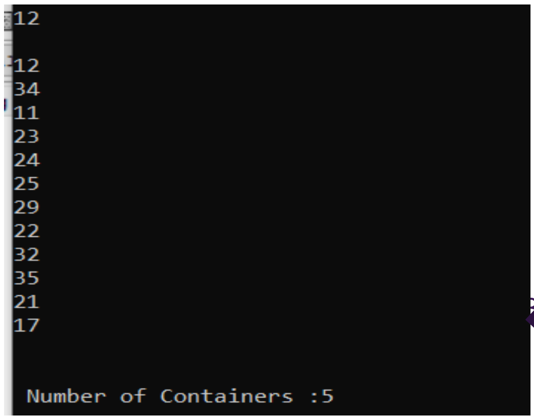
```
C:\Local Disk D\Ucsc\Cpp Programming\Greedy Algo Q3.exe
8

1
12
21
3
7
21
2
14


 Number of Containers :4
--------------------------------
Process exited after 22.99 seconds with return value 0
Press any key to continue . . .
```

Output:

Minimum No of Containers to Given Constraints

2) **Number of Items in the Order** – 12

   Weights in integer: - (12, 34, 11, 23, 24, 25, 29, 22, 32, 35, 21, 17)



```
12

12
34
11
23
24
25
29
22
32
35
21
17


 Number of Containers :5
```