DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING
RAMAIAH INSTITUTE OF TECHNOLOGY
(AUTONOMOUS INSTITUTE AFFILIATED TO VTU)
M. S. R. I. T. POST, BANGALORE – 560054

PROGRAMMING ASSIGNMENT REPORT

DESIGN AND ANALYSIS OF ALGORITHMS

SUBJECT CODE: IS45

Submitted By,

OWAIS IQBAL

USN: 1MS20IS081

Submitted to,

Dr. Prathima M N.
Professor
Dept. of ISE, RIT

Table of contents

PROBLEM STATEMENT

LEET CODE: 2279

Maximum Bags With Full Capacity of Rocks

Problem:

You have n bags numbered from 0 to n - 1. You are given two **0-indexed** integer arrays capacity and rocks. The i^{th} bag can hold a maximum of capacity[i] rocks and currently contains rocks[i] rocks. You are also given an integer additionalRocks, the number of additional rocks you can place in **any** of the bags.

Return *the **maximum** number of bags that could have full capacity after placing the additional rocks in some bags.*

Constraints:

- `n == capacity.length == rocks.length`
- $1 <= n <= 5 * 10^4$
- $1 <= capacity[i] <= 10^9$
- `0 <= rocks[i] <= capacity[i]`
- $1 <= additionalRocks <= 10^9$

IMPLEMENTATION (CODE)

```
#include<stdio.h>
#include<stdlib.h>

int maximumBags(int capacity[], int capacitySize, int rocks[], int rocksSize, int
additionalRocks){

    int sum=0;
    int cmpfunc (const void * a, const void * b) {
       return ( *(int*)a - *(int*)b );
    }

    for(int i=0;i<capacitySize;i++){
        capacity[i]=capacity[i]-rocks[i];
    }

    qsort(capacity, capacitySize , sizeof(int), cmpfunc);

    for(int i=0;i<capacitySize;i++){
        if(additionalRocks>=capacity[i]){
            additionalRocks-=capacity[i];
            sum+=1;
        }
        else if(additionalRocks<capacity[i])
             return sum;
    }

    return sum;
}

void main(){
    int capacity[]={2,3,4,5};
    int rocks[]={1,2,4,4};
    int additionalRocks=2;
    int capacitySize=4,rocksSize=4;
    int result=maximumBags(capacity, capacitySize, rocks, rocksSize, additionalRocks);
    printf("Result= %d", result);
}
```

TEST CASES

Example 1:

```
Input: capacity = [2,3,4,5], rocks = [1,2,4,4], additionalRocks = 2

Output: 3

Explanation:

Place 1 rock in bag 0 and 1 rock in bag 1.

The number of rocks in each bag are now [2,3,4,4].

Bags 0, 1, and 2 have full capacity.

There are 3 bags at full capacity, so we return 3.

It can be shown that it is not possible to have more than 3 bags at full capacity.

Note that there may be other ways of placing the rocks that result in an answer of 3.
```

Example 2:

```
Input: capacity = [10,2,2], rocks = [2,2,0], additionalRocks = 100

Output: 3

Explanation:

Place 8 rocks in bag 0 and 2 rocks in bag 2.

The number of rocks in each bag are now [10,2,2].

Bags 0, 1, and 2 have full capacity.

There are 3 bags at full capacity, so we return 3.

It can be shown that it is not possible to have more than 3 bags at full capacity.

Note that we did not use all of the additional rocks.
```

ADDITIONAL TEST CASES

```
[2,3,4,5]

[1,2,4,4]

2


[10,2,2]

[2,2,0]

100


[54,18,91,49,51,45,58,54,47,91,90,20,85,20,90,49,10,84,59,
29,40,9,100,1,64,71,30,46,91]

[14,13,16,44,8,20,51,15,46,76,51,20,77,13,14,35,6,34,34,13,
3,8,1,1,61,5,2,15,18]

77


[91,54,63,99,24,45,78]

[35,32,45,98,6,1,25]

17
```

RESULT

Accepted

Runtime: 0 ms

Your input

```
[2,3,4,5]
[1,2,4,4]
2
[10,2,2]
[2,2,0]
100
[54,18,91,49,51,45,58,54,47,91,90,20,85,20,90,49,10,84,59,29,40,9,100,1,64,71,
30,46,91]
[14,13,16,44,8,20,51,15,46,76,51,20,77,13,14,35,6,34,34,13,3,8,1,1,61,5,2,15,1
8]
77
[91,54,63,99,24,45,78]
[35,32,45,98,6,1,25]
17
```

Output

```
3
3
13
1
```

Expected

```
3
3
13
1
```

Success

Runtime: 202 ms, faster than 90.00% of C online submissions for Maximum Bags With Full Capacity of Rocks.

Memory Usage: 14 MB, less than 80.00% of C online submissions for Maximum Bags With Full Capacity of Rocks.

Submission Detail

79 / 79 test cases passed.

Status:

Accepted

Runtime: 189 ms

Memory Usage: 14.1 MB

ANALYSIS

1) Input size is 'n' (capacitySize) which is the number of bags (the size of capacity and rocks arrays).
2) Basic operation here is comparison.
3) The efficiency of the algorithm depends on the size of the input.
4) Time complexity:

$$C(n)= \sum_{i=0}^{n-1}(1)$$

- where (1) is the times the basic operation occurs.
- So, the time complexity is

$$C(n)= \theta(n)$$

- We have three cases in it.

i. **Best case:**

This occurs when `additionalRocks == 0`.

So,

$$C(n)= \theta(1).$$

ii. **Worst case:**

This occurs when,

$$\texttt{additonalRocks} >= \sum_{i=0}^{n-1}(\texttt{capacity[i]-rocks[i]})$$

So,

$$C(n)= \theta(n).$$

iii. Average case:

This occurs when,

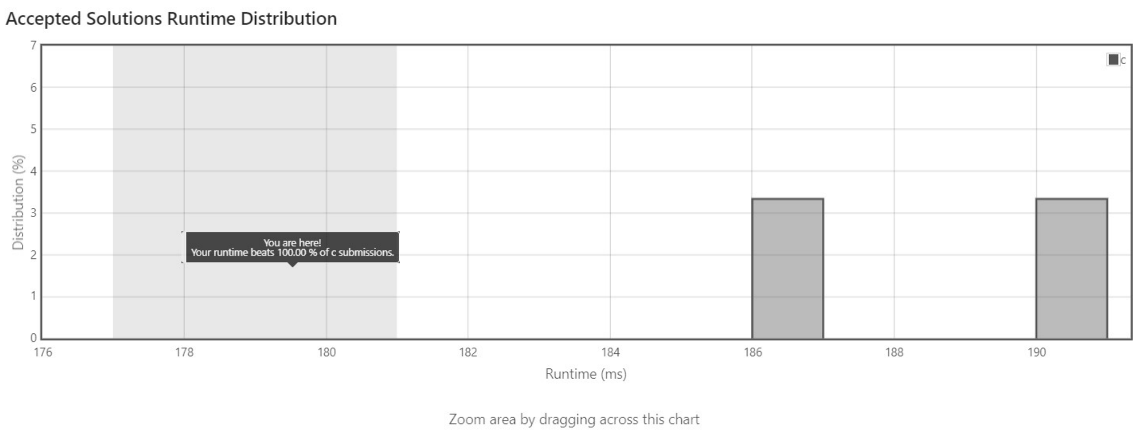
$$\texttt{additonalRocks} < \sum_{i=0}^{n-1}(\texttt{capacity[i]-rocks[i]})$$

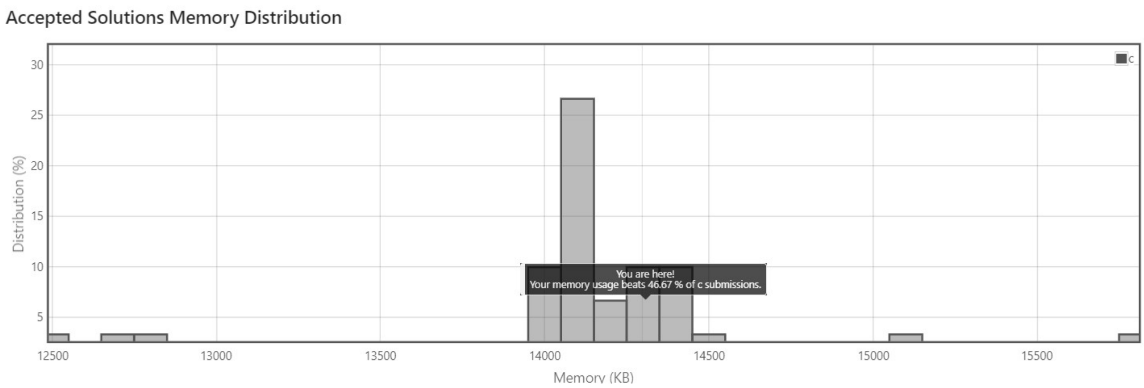
So,

$$C(n) = \theta(n).$$

5) Space complexity

- The space complexity is also θ(capacitySize).





Conclusion

The given code for **Maximum Bags With Full Capacity of Rocks** has the **best time complexity as θ(1) and the worst and average being θ(n)**. This cannot be improved further as it is necessary to go through each and every element in a sequential fashion inorder to update the number of full capacity bags. Therefore, this stands to be the best design of the solution and could not be improved any further.

I would like to thank my Design and Analysis of Algorithms Professor. **Dr. Prathima M N** for giving me the opportunity to work on the prescribed problem also my friends who were supportive enough to guide me through the hurdles that came along.