Exam score analysis: Find average, median and mode

The array of scores:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4 | 2 | 2 | 8 | 3 | 3 | 1 |  |

Object tuple model for this array :

|  |
| --- |
| A: 14361224 |
| Obj\_T: int[7] |
| N: score |
| S: 7 |
| V: 0x00546F1D |
| V\_T: int\* |
| Range Address: 14361224 - 14361252 |
| Range of Index: 0 - 6 |

Object tuple model of each element :

For idx 0 :

|  |
| --- |
| A: 14361224 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 4 |
| V\_T: int |
| Range Address: 14361224 - 14361228 |
| Range of Index: 0 - 0 |

For idx 1 :

|  |
| --- |
| A: 14361228 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 2 |
| V\_T: int |
| Range Address: 14361228 - 14361232 |
| Range of Index: 1 - 1 |

For idx 2 :

|  |
| --- |
| A: 14361232 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 2 |
| V\_T: int |
| Range Address: 14361232 - 14361236 |
| Range of Index: 2 - 2 |

For idx 3 :

|  |
| --- |
| A: 14361236 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 8 |
| V\_T: int |
| Range Address: 14361236 - 14361240 |
| Range of Index: 3 - 3 |

For idx 4 :

|  |
| --- |
| A: 14361240 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 3 |
| V\_T: int |
| Range Address: 14361240 - 14361244 |
| Range of Index: 4 - 4 |

For idx 5 :

|  |
| --- |
| A: 14361244 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 3 |
| V\_T: int |
| Range Address: 14361244 - 14361248 |
| Range of Index: 5 - 5 |

For idx 6 :

|  |
| --- |
| A: 14361248 |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 1 |
| V\_T: int |
| Range Address: 14361248 - 14361252 |
| Range of Index: 6 - 6 |

Now we are going to think about some critical point:

float calculateAverage(int \*score, int n)

{

    float sum = 0;

    for (int i = 0; i <= n; i++) // i goes to n

    {

        sum += \*(score + i);

    }

    return sum / n;

}

Suppose the index of the element we are thinking about is n .

Then it’s address will be = Adress from object tuple model + size of int \* idx

= 14361224 + 4 \* n

= 14361224 + 4 \* 7

= 14361252

Object tuple model of the array was

|  |
| --- |
| A: 14361224 |
| Obj\_T: int[7] |
| N: score |
| S: 7 |
| V: 0x00546F1D |
| V\_T: int\* |
| Range Address: 14361224 - 14361252 |
| Range of Index: 0 - 6 |

The object tuple model of this element is

|  |
| --- |
| A: 143612252 (Beyond scope!) |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 4 |
| V\_T: int |
| Range Address: 14361252 - 14361256 |
| Range of Index: 7-7 |

float calculateMedian(int \*score, int n)

{

    counting\_sort(score, n); // to sort the array of scores

    if (n % 2 == 0)

    {

        return (\*(score + n / 2) + \*(score + n / 2 - 1)) / 2;

    }

    else

    {

        return \*(score + n / 2);

    }

}

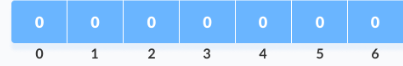
The score array:



This is how counting sort works:

Counting sort needs an array named count. Every index is considered as a bucket for that particular value. Then we traverse and the count array and finally store the values in score array where it belongs.

If any beginner defines the count array with size of score array then the count array will be



void counting\_sort(int \*array, int size)

{

. . . memory allocation . . .

    for (int i = 0; i < max + 1; i++)

    {

        \*(count + i) = 0;

    }

    for (int i = 0; i < size; i++)

    {

        // increments the count array at \*(array+i) position

        \*(count + \*(array + i)) = \*(count + \*(array + i)) + 1;

}

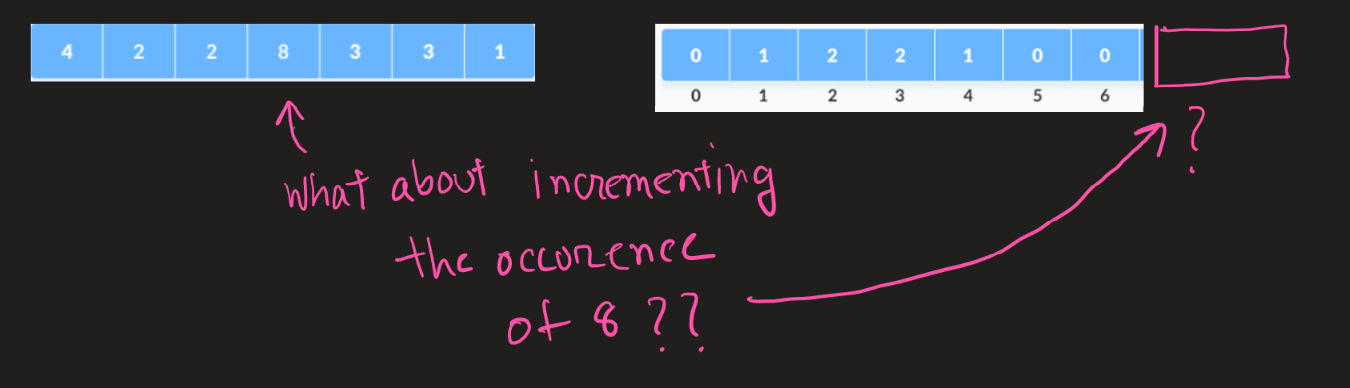
. . . other codes . . .

}

and the object tuple model of count array will be

|  |
| --- |
| A: 26485616 |
| Obj\_T: int[7] |
| N: count |
| S: 7 |
| V: 26485616 |
| V\_T: int\* |
| Range Address: 26485616 - 26485644 |
| Range of Index: 0 - 6 |

When we will increment for array value 8



When the beginner will go to increment the occurrence of 8 in count array,

He will try to access the address = 26485616 + 8\*4 = 26485648

The object tuple model for idx 8 is

|  |
| --- |
| A: 26485648 (Beyond scope!) |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 4 |
| V\_T: int |
| Range Address: 26485648 - 26485652 |
| Range of Index: 8-8 |

So The thing that is wrong in this situation is the size of count array, The last index of the count array should be the max value of score array.

So we modify the count sort function like this

void counting\_sort(int \*array, int size)

{

int max = array[0];

    for (int i = 1; i < size; i++)

    {

        if (array[i] > max)

            max = array[i];

    }

int \*count = (int \*)malloc((max + 1) \* sizeof(int));

. . . other part of code . . .

}

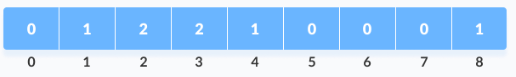
Now object tuple model of the count array is

|  |
| --- |
| A: 26485616 |
| Obj\_T: int[9] |
| N: count |
| S: 7 |
| V: 26485616 |
| V\_T: int\* |
| Range Address: 26485616 - 26485652 |
| Range of Index: 0 - 8 |

Object tuple model for index 8

|  |
| --- |
| A: 26485648 (In scope) |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 4 |
| V\_T: int |
| Range Address: 26485648 - 26485652 |
| Range of Index: 8-8 |

The count array:



Reverse loop mistakes:

void counting\_sort(int \*array, int sz)

{

int \*output = (int \*)malloc(size \* sizeof(int));

. . . other code . . .

    for (int i = sz ; i >= 0; i--) // <--- error here

    {

        \*(output + \*(count + \*(array + i)) - 1) = \*(array + i);

        \*(count + \*(array + i)) = \*(count + \*(array + i)) - 1;

}

. . . other code . . .

}

Reverse loops are often mistaken by beginners as arrays do not start from 1 , rather 0. So, the last index of the array is not equal to size of the array.

Object tuple model for score array

|  |
| --- |
| A: 14361224 |
| Obj\_T: int[7] |
| N: score |
| S: 7 |
| V: 14361224 |
| V\_T: int\* |
| Range Address: 14361224 - 14361252 |
| Range of Index: 0 - 6 |

Object tuple model for index sz

|  |
| --- |
| A: 14361252 (beyond scope!) |
| Obj\_T: int |
| N: N/A |
| S: 4 |
| V: 14361224 |
| V\_T: int |
| Range Address: 14361252 - 14361256 |
| Range of Index: 7 - 7 |

Rotate an array by k steps clockwise

Here k can be any positive integer

A possible solution can be

void rotate(int\* nums, int numsSize, int k) {

    int\* temp = (int\*)malloc(numsSize \* sizeof(int));

    for(int i = 0; i < k; i++) {

        \*(temp + i) = \*(nums + numsSize - k + i);

    }

    for(int i = 0; i < numsSize - k; i++) {

        \*(temp + i + k) = \*(nums + i);

    }

    memcpy(nums, temp, numsSize \* sizeof(int));

    free(temp);

}

Now if k (suppose 11) is greater than the size of nums (suppose 7), then the above solution does not work. The reason can be shown using object tuple model.

For segment

    for(int i = 0; i < k; i++) {

        \*(temp + i) = \*(nums + numsSize - k + i);

    }

Object tuple for temp array object tuple model for temp array element when i=10

|  |
| --- |
| A: 26485616 |
| Obj\_T: int[7] |
| N: temp |
| S: 7 |
| V: 26485616 |
| V\_T: int\* |
| Range Address: 26485616 - 26485652 |
| Range of Index: 0 - 6 |

|  |
| --- |
| A: 26485656 (out of index) |
| Obj\_T: int[7] |
| N: N/A |
| S: 7 |
| V: 26485616 |
| V\_T: int\* |
| Range Address: 26485656 - 26485660 |
| Range of Index: 9 - 9 |

Beginner can address this problems using object tuple model

Now when we rotate the array by 11 steps and and we rotate the array 4 steps they are basically the same.

So we mod k with size of array.

void rotate(int\* nums, int numsSize, int k) {

    int\* temp = (int\*)malloc(numsSize \* sizeof(int));

    k = k % numsSize; //<-- solve

    for(int i = 0; i < k; i++) {

        \*(temp + i) = \*(nums + numsSize - k + i);

    }

    for(int i = 0; i < numsSize - k; i++) {

        \*(temp + i + k) = \*(nums + i);

    }

    memcpy(nums, temp, numsSize \* sizeof(int));

    free(temp);

}