**Assignment 1**

**Draghici Bogdan**

**The Faculty of ACE, Craiova**

**First Year, Calculatoare Romana**

**CR 1.2.A**

**Problem 2**

**I. Problem statement**

Take a sequence of 2n real numbers as input. Design an O(n log n) algorithm that partitions the numbers into n pairs, with the property that the partition minimizes the maximum sum of a pair.

For example, say we are given the numbers (1,6,3,7), the possible partitions are ((1,6), (3,7)), ((1,3), (6,7)) and ((1,7), (3,6)). The pair sums for these partitions are (4,13), (7,10), and (8,9). Thus, the third partition has 9 as its maximum sum, which is theminimum over the three partitions.

**II. Pseudocode description of all the algorithms used in the lab homework**

The main algorithm used to solve the problem is contained in the next function:

1. *write “(‘vector[i]’,’vector[no\_pairs-i+1]’),”*

In order for this algorithm to work, the array that contains the 2n real numbers must be sorted. So, after reading the input values, I called the merge sort algorithm that I implemented in one of the previous AD labs.

When we take in order from the pairs with the biggest difference in module, to the pairs with the smallest difference in module, we obtain the most appropriate values for the sums of the pairs. Therefore, the biggest minimum sum value from the sums of the n pairs, and also the smallest maximum sum value from the sums of the n pairs, will be one of the pairs with the biggest difference taken in module.

The easiest way to make the pairs from which we will find the smallest maximum sum, is to sort the elements, then pair the elements equidistant to the middle of the vector. After the sum of each pair is made, all that remains is to find which of them is the biggest one.

This function is using the array of elements vector, a variable sum that is used to make the sums of the no\_pairs pairs, max\_value in which we will memorize from sum the minimum for maximum sum, and an iterator i to create the pairs of numbers.

On line 2 I initialized max\_value = -1, to represent that there have not been calculated any sums and to return -1 in case there are no pairs.

On line 3 is a repetitive structure with step 1 starting with i from 1 to no\_pairs, used to create the no\_pairs sums of the pairs of numbers from which we will find the minimum for de maximum sum.

On line 4 we write the pairs resulted in order to obtain the minimum for the maximum sum, in an output file.

The sums of the pairs equidistant to the middle of the vector are made on line 5, by adding the element from the position i in vector with the element from the position no\_pairs-i+1 in vector.

Next is comparison on line 6 in order to see if the recently calculated sum is bigger than the maximum of the previous sums. In the affirmative, on line 7 max\_value takes the value of sum. After the repetitive structure ends, the algorithm proceeds to line 8, on which we write the answer.

Next will be presented the Merge-Sort algorithm.



This function will self-call itself, each time with half the range of elements, till the range of elements will become single element, then will rebuild the sorted vector with the use of the function merge\_left\_right, shown as follows:



On line 2 it is attribute to no\_elem\_left and no\_elem\_right, the number of elements from the left to the middle of vector, respectively from the middle+1 to the right of vector. On line 3 and 4 it is allocated space for no\_elem\_left elements in vector\_left, respectively no\_elem\_right elements in vector\_right.

On lines 5 and 6 it is copied in vector\_left the elements from vector from position left to position left+no\_elem\_left. On lines 7 and 8 it is copied in vector\_right the elements from vector from position right to position right+no\_elem\_right.

On line 9, it is attributed to variables vector\_left and vector\_right, the strating position 0, and to it\_merge the starting position left.

The repetitive structure from line 10 will continue to run till one from vector\_left and vector\_right got all the values copied in vector.

The condition from line 11 is for the vector in order to take the smallest value from both other vectors, then increment the iterator that had the smallest one, and also it\_merge, the iterator of vector.

After the first while ends, there will still be elements in one of the two vectors that haven not been uncopied in vector. For this are the two repetitive structures remained, one in case vector\_left has elements that are bigger than all the other elements from vector\_right, that need to be copied in vector, and one in case vector\_right is the one that has elements that need to be copied in vector. Only one of this two structures will be executed, copying the remaining values and incrementing the specific vectors’ iterators.

**III. Algorithms' memory and computational complexity**

The algorithm used to solve the problem has the complexity , where

.

Complexity of Merge Sort function:

Complexity of function max\_sum:

So, the complexity of the algorithm is:

The memory used is .

**Best case scenario**

In this case, the max is the first pair of numbers, the pair formed from the smallest and biggest number in the vector.

For the function max\_sum, the elemental time is:

* 1 for the first attribution of max and also 1 for the first attribution of i, and with the condition and the incrementation of i from the repetitive structure, results
* 4 for each writing and each attribution for sum, and 1 for each verification of the condition, those operations being made n times, where we add the only attribution of max, resulting
* The writing of max has elemental time 1

In the end, we have elemental time for the function max\_sum in the best case scenario.

**Worst case scenario**

In this case, the pairs have the sum as being orderly ascending, resulting in attribution of max each time we enter in the repetitive structure.

For the function max\_sum, the elemental time is:

* 1 for the first attribution of max and also 1 for the first attribution of i, and with the condition and the incrementation of i from the repetitive structure, results
* 4 for each writing and each attribution for sum, and 2 for each verification of the condition and attribution to max, those operations being made n times, where we add the only attribution of max, resulting
* The writing of max has elemental time 1

In the end, we have elemental time for the function max\_sum in the worst case scenario.

**IV. Experimental data**

For the generation of the input data sets I used the rand() function from the library math.h, and also an function implemented by myself, that generates a random number from a given range.

1. *function random(min, max)*

RAND\_MAX is a constant which represents the biggest value that can be generated using the function rand(), and it’s value is 32767.

On line 2 I initialized counter with the length of the range and the variable in which the number will be created with 0.

On line 3 is a repetitive structure that will be executed for times, in order to obtain a number in the range.

On line 4, the number is multiplied by RAND\_MAX and also added a new random value generated by the rand() function. On line 5 the counter is divided by the maximum value that can be taken by a number generated in rand().

So, when the number is multiplied by RAND\_MAX, the counter is divided by RAND\_MAX, in order to get to the given range as quickly as possible.

After the repetitive structure ends, the number created might exceed the range length, so we must do the rest of the division with the range length+1, then transform the number intro a real one, than divide it by RAND\_MAX and multiply it by a random value to obtain the decimal part of the number.

Finally, I add the first value of the range in order to obtain a number between min and max.

**V. Results and Conclusion**

The algorithm has a fast time on a small data input, but this changes with a bigger data input. Next is shown the graphic for execution time for different input data sizes.

Next will be presented the average time for average data input.

Here we can see the ratio Time/Data slightly decreasing on average data input, but it is increasing on a large input data.

|  |  |  |
| --- | --- | --- |
| **Average Data** | **Average time** | **Ratio Time/Data** |
| 8882031.2 | 5.5744 | 0.62760419 |
| 18917650.8 | 12.493 | 0.660388551 |
| 24986467.2 | 13.1004 | 0.52429981 |
| 30796920.4 | 16.2392 | 0.527299476 |
| 37077883.2 | 20.9374 | 0.564687037 |
| 41454936.4 | 22.605 | 0.54529091 |
| 48507834 | 26.8348 | 0.553205488 |
| 61277067.6 | 39.4726 | 0.644165942 |
| 75311527.6 | 47.7588 | 0.634149931 |
| 90233199.2 | 56.4284 | 0.625361846 |

1. **Summary of achievements**

One of my achievements is that I figured out a way to calculate the smallest maximum sum from a set of elements without the need to try all the permutations. I also believe that by solving this problem I learned a way to pair two units in such a way that all pairs have a similar value and are close to the average, to be in balance.

1. **Brief description of the most challenging/ interesting achievements**

Most challenging achievement was finding a way to generate all the permutations, but eventually giving up this variant in favor of a more efficient algorithm in my opinion.

Most interesting achievement is that I found out that I can also write in, not only read from input files.

1. **Future directions for extending the lab homework**

I am looking forward to find a faster algorithm that solves the problem, one without the need of sorting the vector.