Carla Fernández González. UO244965

Divide And Conquer

FAKE COINS

In order to solve this problem, my approach was the following:

- If we are down to 3 coins:
  - Balance 1st coin with 2nd coin.
  - Balance 2nd coin with 3rd coin.
  - If both sides are unbalanced, the fake is in the middle.
  - If the left side is unbalanced, the fake coin is the 1st.
  - If the right side is unbalanced, the fake coin is the 3rd.
- If we are down to 4 coins:
  - Balance 1st coin with 2nd coin.
  - Balance 3rd coin with 4th coin.
  - If the left side is unbalanced, use getFake3() with 1st, 2nd and 3rd coins.
  - If the right side is unbalanced, use getFake3() with 2nd, 3rd and 4th coins.
- If we are down to 5 coins:
  - Use getFake4() for 1st to 4th coins.
  - If the result is different from -1, we found the fake coin, return it.
  - Otherwise, the fake coin is the 5th coin.
- If we are down to 6 coins:
  - Balance the left side using getFake3() from 1st to 3rd.
  - Balance the right side using getFake3() from 4th to 6th.
  - Whichever result is different from -1 will be the fake coin.
- Else:
  - General approach: divide each part into halves and take balanceLeft and balanceRight on each half.
  - Problem: how to choose elements in order to always have the same amount of coins on each part of the balance?
  - Solution:
    - Distance = end – start
    - Center = distance / 2
    - For both variables, if the number is odd you do not repeat the central element when weighing. If the number is even, you repeat that element.
    - Odd distance, odd center → do not repeat central element on either half.

- Odd distance, even center → repeat central element only on each half.
- Even distance → repeat general central element.
- Even distance, even center → repeat the central element on each half.

Complexity:

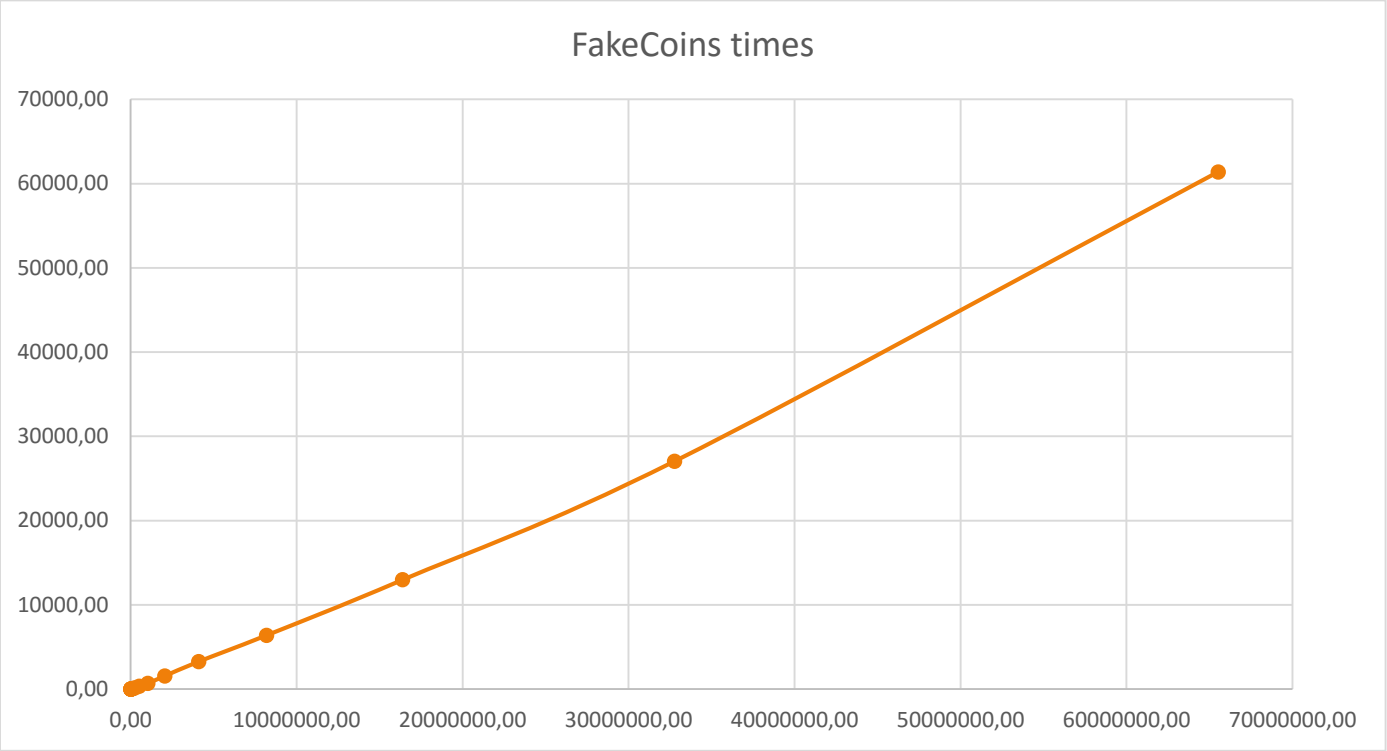
There will only be one recursive call in the whole algorithm, which means that a = 1. In each call, the vector size will be divided in half, which means that we have a Divide & Conquer by Division scheme with b = 2. There are no loops in the algorithm, only the one used by the balance to weigh the coins, a method which is O (n), so k = 1. Thus, the final complexity can be calculated using the formula:

$a < b^k \rightarrow O(n^k) = O(n)$

Times table

Size	Total time	time(micros)
1000,00	860,00	0,86
2000,00	1364,00	1,36
4000,00	2611,00	2,61
8000,00	559,00	5,59
16000,00	1064,00	10,64
32000,00	2113,00	21,13
64000,00	4173,00	41,73
128000,00	841,00	84,10
256000,00	1659,00	165,90
512000,00	3379,00	337,90
1024000,00	686,00	686,00
2048000,00	1564,00	1564,00
4096000,00	3275,00	3275,00
8192000,00	640,00	6400,00
16384000,00	1298,00	12980,00
32768000,00	2705,00	27050,00
65536000,00	614,00	61400,00

Times graph



The graph confirms what we knew: the algorithm is O (n) and thus solves the problem efficiently.

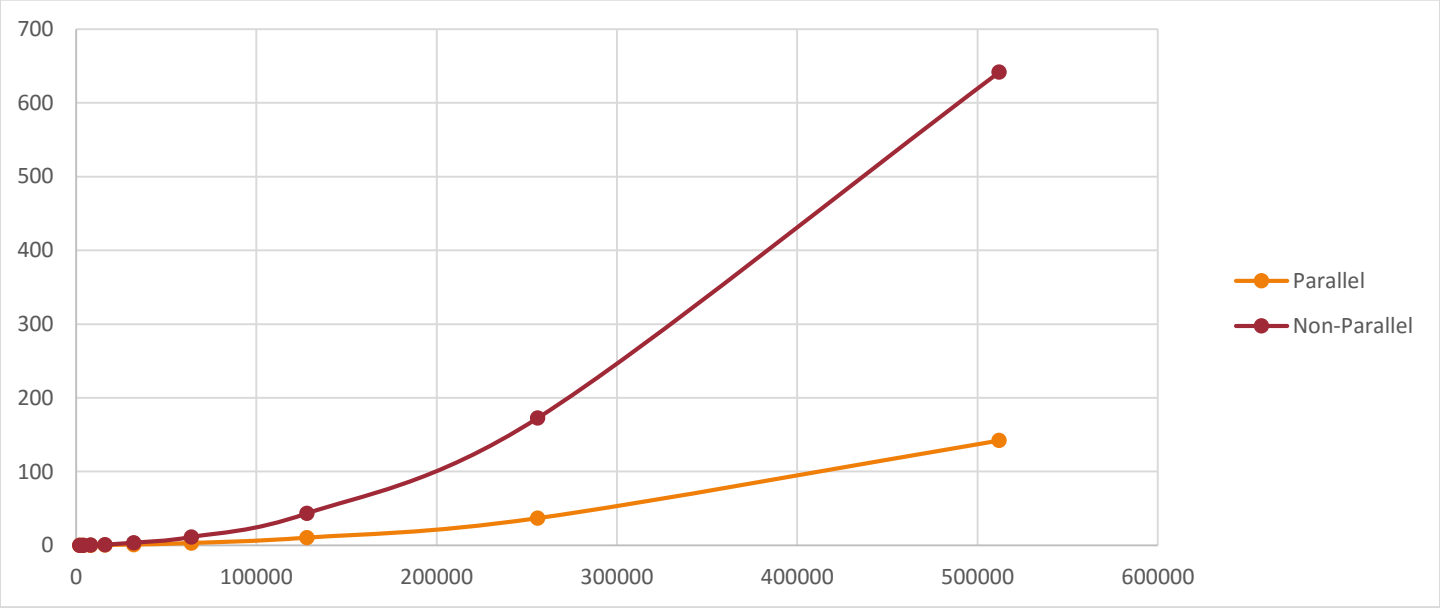
PARALLEL QUICKSORT

The parallel version of quicksort has the same complexity as the non-parallel version, O (n*log (n)) for the average case. However, it is much faster when values of n start getting big. This is due to the usage of more processor power (more than one processor vs. only one). The following is a table showing the empirical measures I took. They were taken using a randomly sorted vector in both cases:

Sorting	Size	Time (ms)
Parallel Random	2000	0,05148
Parallel Random	4000	0,10697
Parallel Random	8000	0,18
Parallel Random	16000	0,395
Parallel Random	32000	1,042
Parallel Random	64000	3,052
Parallel Random	128000	10,3
Parallel Random	256000	36,8
Parallel Random	512000	142,2

Sorting	Size	Time (ms)
Random	2000	0,03512
Random	4000	0,1
Random	8000	0,317
Random	16000	1,051
Random	32000	3,478
Random	64000	11,2
Random	128000	43,2
Random	256000	172,5
Random	512000	641,6

In the graph we can see the difference clearly:



Thus, parallelizing the quicksort algorithm provides a much better performance even with values of n as low as 8000.