Activity 1. Divide and Conquer by subtraction

In both Subtraction1.java and Subtraction2.java, we have a StackOverflowError for values over n=8000, because we exceed the size of the computer’s stack with a big amount of recursive calls.

In Subtraction3.java, we have a = 2, b = 1 and k = 0. As it’s subtraction, and a > 1, we can obtain the time complexity O(an/b), therefore the time complexity will be O(2n). Then, knowing that the time it takes to manage n­1 = 29 is t1 = 37076 ms, we can obtain the time that it will take for n2 = 80 with:

t2 = (2n2/2n1)\*t1 = 8.348772989 \* 1019 ms.

Converted to years:

8.348772989 \* 1019 ms -> 8.348772989 \* 1016 s -> 1.391462165 \* 1015 min -> 2.319103608 \* 1013 hours -> 9.6629317 \* 1011 days -> **2647378584** **years**

The Subtraction4.java algorithm was created using two nested loops and a recursive call that give us a= 1, b=1, k=2, so our O(nk+1) is O(n3).

We obtain the following time measurements:

|  |  |
| --- | --- |
| **n** | **nTimes** |
| 100 | 6.2 ms |
| 200 | 40.6 ms |
| 400 | 302 ms |
| 800 | 2339 ms |
| 1600 | 18544 ms |
| 3200 | Oot |

The Subtraction5.java algorithm was created using two nested loops and three recursive call that give us a= 3, b=2, k=0, so our O(an/b) is O(3n/2).

We obtain the following time measurements:

|  |  |
| --- | --- |
| **n** | **nTimes** |
| 30 | 953 ms |
| 32 | 2189 ms |
| 34 | 6867 ms |
| 36 | 19477 ms |
| 38 | Oot |

If we want to know how many years would it last for our algorithm for a problem size of 80, we can use:

t2 = (3n2/2/3n1/2)\*t1 = (340/318)\* 19477 = 6.11208898 \* 1014 ms.

Converted to years:

6.11208898 \* 1014 ms -> 6.11208898 \* 1011 s -> 1.018681497 \* 1010 min -> 169780249.4 hours -> 7074177.06 days -> **19381.3** **years**

Activity 2. Divide and conquer by division

As it can be seen, Division1 and Division3 have the same time complexity O(n), so they have similar time measurements. On the other hand, Division2 has a time complexity of O(nlogn), which is worse than O(n), therefore it has larger time measures.

Division4.java algorithm was created using two nested loops and a recursive call that give us a= 1, b=2, k=2, so our O(nk) is O(n2) and a < bk.

|  |  |
| --- | --- |
| **n** | **nTimes** |
| 1000 | 9.5 ms |
| 2000 | 37.4 ms |
| 4000 | 150 ms |
| 8000 | 591 ms |
| 16000 | 2328 ms |
| 32000 | 9359 ms |
| 64000 | 37632 ms |

Division5.java algorithm was created using two nested loops and a recursive call that give us a= 1, b=2, k=2, so our O(nk) is O(n2) and a < bk.

|  |  |
| --- | --- |
| **n** | **nTimes** |
| 1000 | 9.5 ms |
| 2000 | 37.4 ms |
| 4000 | 150 ms |
| 8000 | 591 ms |
| 16000 | 2328 ms |
| 32000 | 9359 ms |
| 64000 | 37632 ms |

Activity 3.Two basic examples

VectorSum2.java

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **Option1** | **Option2** | **Option3** |
| 3 | 0.000061 ms | 0.000126 ms | 0.000158 ms |
| 6 | 0.000122 ms | 0.0002 ms | 0.000233 ms |
| 12 | 0.000141 ms | 0.000343 ms | 0.0008 ms |
| 24 | 0.000252 ms | 0.000639 ms | 0.00164 ms |
| 48 | 0.000463 ms | 0.001264 ms | 0.00455 ms |
| 96 | 0.00092 ms | 0.002327 ms | 0.00487 ms |
| 192 | 0.00168 ms | 0.005434 ms | 0.0088 ms |
| 384 | 0.00330 ms | 0.011083 ms | 0.0183 ms |
| 768 | 0.0066 ms | 0.021856 ms | 0.0346 ms |
| 1536 | 0.0130 ms | 0.043443 ms | 0.071 ms |
| 3072 | 0.0262 ms | 0.0878 ms | 0.142 ms |
| 6144 | 0.065 ms | 0.2384 ms | 0.272 ms |
| 12288 | 0.107 ms | StackOverflow | 0.61 ms |
| 24576 | 0.210 ms | StackOverflow | 1.14 ms |
| 49152 | 0.412 ms | StackOverflow | 2.22 ms |
| 98304 | 0.95 ms | StackOverflow | 4.51 ms |

As it can be seen , the algorithm with the best performance is the one corresponding to the Option1. This is not a recursive algorithm, and uses only 1 loop that iterates over an array one time, performing the addition of its elements. It has complexity O(n)

On the other hand, the algorithm corresponding to the Option2 uses the recursive method recSust that uses recursion by subtraction with a=1, b=1, k=0, so therefore it has complexity O(n)

Finally, the algorithm corresponding to the Option3 implements also a recursive solution for our problem, but now using the recursion by division approach with a=2, b=2, k=0, so therefore it has complexity O(n) (And it does not cause StackOverflow at 12288)

Fibonacci2.java

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Order** | **Option1** | **Option2** | **Option3** | **Option4** |
| 10 | 0.00013 ms | 0.000187 ms | 0.000293 ms | 0.00371 ms |
| 15 | 0.000178 ms | 0.000251 ms | 0.000395 ms | 0.0624 ms |
| 20 | 0.000221 ms | 0.000323 ms | 0.000508 ms | 0.449 ms |
| 25 | 0.000268 ms | 0.000420 ms | 0.000639 ms | 5.059 ms |
| 30 | 0.000314 ms | 0.000479 ms | 0.000781 ms | 99 ms |
| 35 | 0.000358 ms | 0.000549 ms | 0.000896 ms | 616 ms |
| 40 | 0.000408 ms | 0.000668 ms | 0.001144 ms | 7737 ms |
| 45 | 0.000572 ms | 0.000711 ms | 0.001117 ms | Oot |
| 50 | 0.000631 ms | 0.000774 ms | 0.001267 ms | Oot |
| 55 | 0.000702 ms | 0.000845 ms | 0.001343 ms | Oot |

The overflow problem was solved by changing the functions to return long and all the in variables to long.

The Fibonacci2.java class uses 4 different algorithms in Fibonacci1.java with different approaches to compute the Fibonacci sequence until a given point. Analyzing this algorithms:

1. fib1: It’s the fastest one. It’s an iterative solution for our problem that computes the values in a for loop, overriding the previous values in each iteration. It has complexity O(n)
2. fib2: It’s also an iterative solution, but now instead of using variables to store the values and override then, it fills an array and locates the values in the array via indexes in order to compute the next value until the target one. It has complexity O(n)
3. fib3: It’s a recursive solution for the Fibonacci problem. It has complexity O(n)
4. fib4: It’s also a recursive solution for the Fibonacci problem, but, in this case, it has the worst complexity of all the previous solutions as it’s exponential -> O(1.6n). Therefore, the algorithm gets Oot pretty fast as we increase the size of the problem, so the time measurements increase (a lot)

Activity 4. Another task

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **t ordered** | **t reverse** | **t random** |
| 31250 | LoR | LoR | LoR |
| 62500 | LoR | LoR | 56 ms |
| 125000 | 101 ms | 101 ms | 119 ms |
| 250000 | 202 ms | 192 ms | 231 ms |
| 500000 | 376 ms | 372 ms | 440 ms |
| 1000000 | 811 ms | 857 ms | 842 ms |
| 2000000 | 1777 ms | 1640 ms | 1944 ms |
| 4000000 | 3323 ms | 3161 ms | 3878 ms |
| 8000000 | 7004 ms | 6830 ms | 8043 ms |
| 16000000 | 14097 ms | 13235 ms | 16101 ms |
| 32000000 | 28552 ms | 27040 ms | 33666 ms |
| 64000000 | 60332 ms | 57278 ms | Oot |
| 128000000 | Oot | Oot | Oot |
| 256000000 | Oot | Oot | Oot |

As the measurements for the quicksort in the previous lab ware done in the school’s computer and the mergesort ones in my own computer, I’ll repeat them to obtain a more accurate result.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **tMergesort(t1)** | **tQuicksort(t2)** | **t1/t2** |
| 250000 | 234 ms | 149 ms | 1.570469799 |
| 500000 | 439 ms | 313 ms | 1.402555911 |
| 1000000 | 891 ms | 664 ms | 1.341867469 |
| 2000000 | 1793 ms | 1349 ms | 1.329132691 |
| 4000000 | 3983 ms | 3288 ms | 1.211374695 |
| 8000000 | 8516 ms | 7503 ms | 1.135012662 |
| 16000000 | 17270 ms | 20303 ms | 0.850613209 |
| 32000000 | 35524 ms | Oot | ---------------------- |
| 64000000 | Oot | Oot | ---------------------- |

As it can be seen, the comparison constant tends to decrease as the size of the problem increases.