ADA – 2016 – 5. semester
Frederik Mazur Andersen – fande14@student.sdu.dk
Robtek14

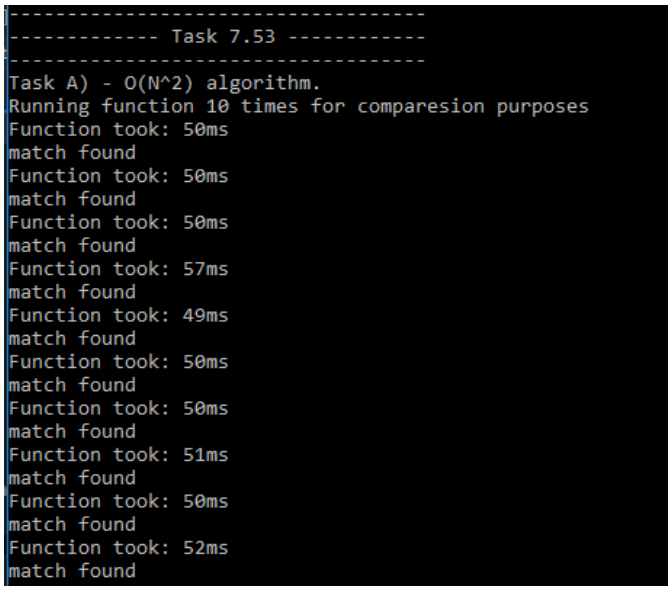
Sorting – Mandatory

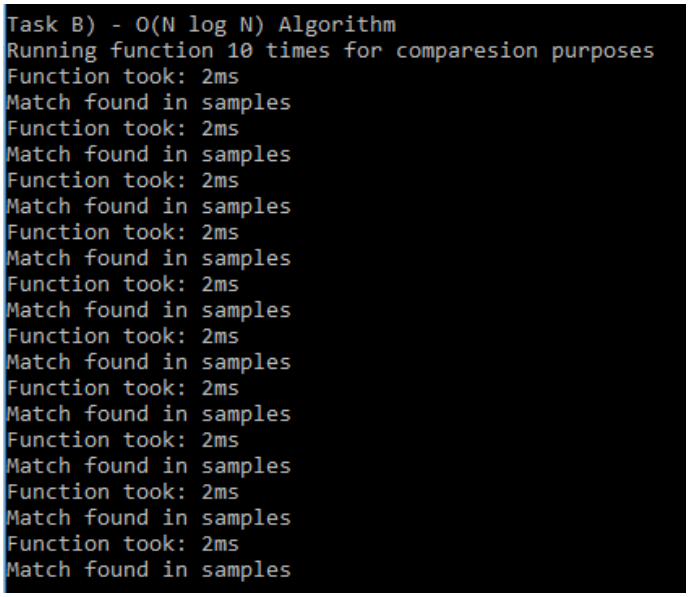
7.53

First algorithm is the O(N^2) which takes substantial more time than the algorithm on the sorted list. The O(N^2) algorithm have a nested for-loop so for each point you check all other points, and see if you have a match.

The O(N log N) algorithm on the other hand is way faster as it utilizes the list is sorted. I sort the list by using HeapSort. When the list is sorted you can check for sums, by having a pointer in each end of the list. (front and back) As you know the highest item is most-front in the list, you can use the information to quickly step though the list. If the sum is lower than our target, you let the back-pointer take a step towards the front. This makes back-pointer item larger or equal. On the other hand, if sum is larger than our target, then the front-pointer takes a step towards the back.

In that way we will always move one step though the list, until we hit same spot or a match is found. So the for-loop will only need to be repeated N times.





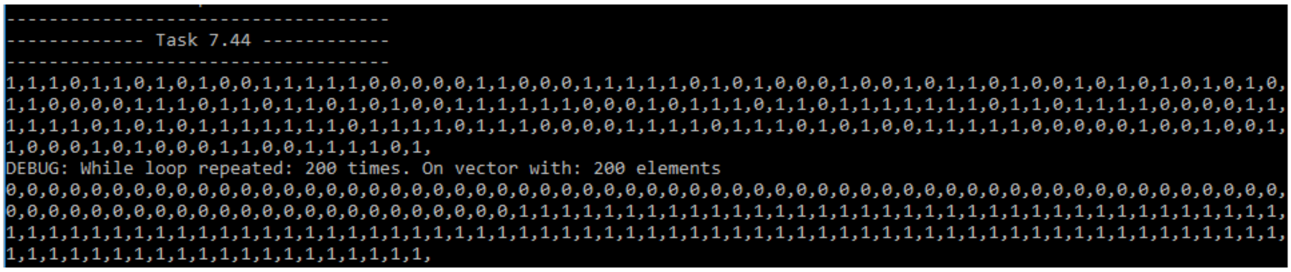
As the above screenshots show the O(N log N) algorithm is much faster. Here with 10.000 random samples, it takes the O(N^2) algorithm ~50ms to find a match. While the other algorithm only uses ~2ms.

7.44

This task I used a similar approach as in 7.53. The Algorithm have an back- and front-pointer and you move them though the array. As you want all the zeroes(false) to be first, you have 4 cases you want to check for.

1. If (front = 1 And back = 0) → you want to swap their value. Then move front one down towards back.
2. If (front = 1 And back = 1) → back is correct so can't swap with front. Move back one step closer to front
3. If (front = 0 And back = 0) → front is correct so can't swap with back. Move front one stop closer back
4. If (front = 0 And back = 1) → Both is correct. Move front one step closer back

Ideally you can see that in case 1 and 4, you could potentially move both pointers, as they both have correct values, and such save a step. Unfortunately, this will cause issues if they happen to both move when they are on the last position. As instead of detecting each other and stop the loop, they would surpass each other and mess up the sorted list + never stop the loop. By thought experiment I think it's possible to add some if-sentences that makes sure they check 2 steps ahead for the other pointer, and if they are closer than 2, only take one step. But I didn't try and implement it.



The above screenshot shows the unsorted list, then the result from sorting the list, and at last the sorted list. And we can see that it only took N repeats to sort an vector with N length.

7.38

In this task I implemented an algorithm that find lines based on the slope of a selected point. So you have a for-loop that goes though all points. Inside that one, you have a second for-loop that calculates every points slope relative to the point selected in the first for-loop.

When calculating the slopes, I check if the x-value from both points used to calculate is the same value. Because if that's the case you get a divide-by-zero exception as it is a vertical line which has a slope of infinity. I choose to express infinity/vertical lines by giving them the slope of 1000. I choose 1000 as it is high enough that in my defined coordinate-system of (200,200) it would be impossible to get that high slope. In case you want to use it on not-known sized coordinate-systems I would need to change my class to handle vertical lines differently. Could do that by adding a Boolean to my Line class which would indicated if it was vertical.

When all the slopes are calculated I sort the list of slopes with heapsort, and then go though and see if the same slope has 4+ entries. If it does, it is a line and gets saved to my line-list. The good thing with this is

that after you have calculated and sorted the slopes, the check if a line exists with 4+ points, is a constant time N. As you only need to visit each element in the list once.

My function outputs a vector<Line> with all the found lines. The result of a 200x200 grid with 300 random points is shown below:

```
------------------------------------
------------ Task 7.38 ------------
------------------------------------

Creating sample plane. (200,200) grid with 300 random points
found 52 lines with 4 or more points
```

Example view of the returned vector<Line> lines:

lines	{ size=52 }
[capacity]	63
[allocator]	allocator
[0]	{points={ size=4 } }
[1]	{points={ size=4 } }
[2]	{points={ size=4 } }
[3]	{points={ size=4 } }
[4]	{points={ size=4 } }
[5]	{points={ size=4 } }
[6]	{points={ size=4 } }
points	{ size=4 }
[capacity]	4
[allocator]	allocator
[0]	{x=80 y=175 slope=1.0000000000000000 }
[1]	{x=81 y=176 slope=1.0000000000000000 }
[2]	{x=13 y=108 slope=1.0000000000000000 }
[3]	{x=61 y=156 slope=1.0000000000000000 }
[Raw View]	{...}
[7]	{points={ size=4 } }
[8]	{points={ size=4 } }
[9]	{points={ size=4 } }
points	{ size=4 }
[capacity]	4
[allocator]	allocator
[0]	{x=68 y=171 slope=-0.00000000000000000 }
[1]	{x=114 y=171 slope=0.00000000000000000 }
[2]	{x=151 y=171 slope=0.00000000000000000 }
[3]	{x=179 y=171 slope=0.00000000000000000 }
[Raw View]	{...}
[10]	{points={ size=4 } }
[11]	{points={ size=4 } }
[12]	{points={ size=4 } }
points	{ size=4 }
[capacity]	4
[allocator]	allocator
[0]	{x=121 y=172 slope=2.0000000000000000 }
[1]	{x=131 y=192 slope=2.0000000000000000 }
[2]	{x=85 y=100 slope=2.0000000000000000 }
[3]	{x=126 y=182 slope=2.0000000000000000 }
[Raw View]	{...}