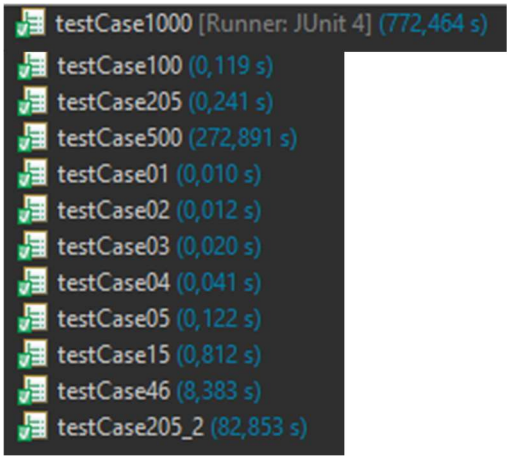
Algorithmics	Student information	Date	Number of session
	UO: 271407	13/04/2020	6
	Surname: Llera		
	Name: Elías		

Escuela de Ingeniería Informática
Universidad de Oviedo

Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Activity 1. Test cases



Algorithmics	Student information	Date	Number of session
	UO: 271407	13/04/2020	6
	Surname: Llera		
	Name: Elías		

Activity 2. Times for different executions.

n	Time
1	0 ms
2	19 ms
3	32 ms
4	49 ms
5	26 ms
6	60 ms
7	103 ms
8	58 ms
9	232 ms
10	226 ms
11	392 ms
12	407 ms
13	547 ms
14	511 ms
15	609 ms
16	1157 ms
17	1294 ms
18	1110 ms
19	1140 ms
20	1498 ms
21	1843 ms
22	1700 ms
23	2279 ms
24	2598 ms
25	3382 ms
26	2492 ms
27	3238 ms
28	2450 ms
29	3587 ms
30	3188 ms
31	3970 ms
32	4498 ms
33	3640 ms
34	4441 ms
35	4351 ms

36	4589 ms
37	5139 ms
38	4833 ms
39	5212 ms
40	5755 ms
41	5692 ms
42	5825 ms
43	6893 ms
44	6080 ms
45	7408 ms
46	6877 ms
47	6929 ms
48	7531 ms
49	7349 ms
50	8940 ms
51	7407 ms
52	8286 ms
53	10066 ms
54	10521 ms
55	10004 ms
56	10464 ms
57	10584 ms
58	10535 ms
59	11229 ms
60	10926 ms

Calculating the complexity for an algorithm like this is not easy. It has a lot of recursive calls to itself and invocations to other methods. However, we can estimate it. We can start by discarding polynomial complexities, since backtracking algorithm almost never have it. Also, if we have a look at the times obtained with the tests, we can see that, after increasing the size of the dictionary, incrementing the size of the table has a big impact on the executing time. Also, we have to take into account that we have to execute the algorithm for every square in the board (that is already square complexity), but in every call we are creating a lot of recursive calls and generating a very big tree in which every potentially every node has the same number of children than its parent node. Taking al this into account, we can conclude that the complexity of the algorithm would be close to exponential or factorial. What really improves the times here is the way we prune the nodes and the way we iterate the dictionary.

Algorithmics	Student information	Date	Number of session
	UO: 271407	13/04/2020	6
	Surname: Llera		
	Name: Elías		

Activity 3. Code improvements.

As we mentioned in the previous activity, what really improves the times of this algorithm is both pruning the tree and improving the dictionary search method. In this implementation, both this methods are done in the search() method. This method uses the dictionary, but not directly. It first accesses an index table (created when the dictionary is imported) that returns the position of the first appearance of a word starting with a given letter. This could also be done with different approaches, such as a table that returns smaller dictionaries containing only the words starting with that given letter or incrementing the number of keys so that we can also access the words starting with a given prefix and not only a letter. This is a very good measure if the dictionary we are using is always going to be the same, since we could implement it so that this tables are only created once and then the different games could reuse it.

Going back to the search() method, we already have a sub-list in which the word will be if it exists. We iterate over it and, if we find it, we return the value FOUND, which means that we have to save the word as a solution and continue developing that branch. If we don't find it, we have two possibilities: the first is that a potential worth starting with the characters that we already have may exist, so we cannot prune the node. To know if this happens, whenever we find a word that returns true for the startsWith(word) invocation, we return NOT_FOUND and continue executing. However, if we finish iterating and the we didn't return any value, we can return IMPOSSIBLE_WORD, which stops the development of the current branch and adds the word to a HashSet containing prefixes that we know don't lead to anywhere. This HashSet is checked at the beginning of the execution of this algorithm to avoid iterating the dictionary if we already know it is impossible.

Several more improvements could be made. Probably using another HashSet for the filtered dictionary (instead of sub-lists) could improve the performance, but we would have to convert it to a list if a contains() call returns false, since we would have to check if we can prune it or not. Another option would be to try to use another algorithm to iterate over the sub-list of the dictionary, and not a plain foreach. Maybe something similar to a binary

search could be useful. We could also try to generate the tree in a whole different structure (like a tree or similar, if we have enough memory).

Finally, the more obvious improvement would be to make this application a multithreaded one, by dividing the board into smaller ones for example. However, several things should be checked when doing so. The more straight forward one would be that solutions cannot be repeated so we would have to lock that part of the code. However, this would not be a big limitation to the improvement, since with a HashSet the contains() method has a constant time and adding the points to a global variable is also constant, so the heavy time-consuming part of the algorithm (the search in the dictionary) could still be done in parallel with no problem (because it is only reading, threads should be able to access it at the same time).

CPU: Intel® Core™ i5-3470 CPU @ 3.20GHz

RAM: 8GB