1A Lent Algorithms

Supervision 4: Data Structures, Graphs

Hayk Saribekyan

May 3, 2018

When you are asked to describe an algorithm, also analyse its time and space complexity. Mention how you would implement the solution if there are subtle details e.g. order of computation in a tabulation method of DP.

This supervision will have two parts. We will finish data structures and move to the next chapter of the course: graph algorithms.

This example sheet is a hybrid of lecturers' problems and mine. Because of that it is slightly longer than I expected but all the problems are quite important to understand. I do not expect you to solve and write down precise solutions to everything, but make sure to read all the problems and write ideas if you have any. You can go back to them during revision or when you have time later.

Focus on the concepts that you have most trouble with rather than writing down solutions to problems that are easy for you.

1 Refresher (Data Structures)

1. Given an arithmetic sequence as a string that is composed of positive integers, basic operations $+, -, *$ and parentheses. Describe an algorithm that converts the expression to post-fix notation. For example, For example `(2 + 3) * 4 + 5` becomes `2 3 + 4 * 5 +`. Notice that in this notation there is no need to think about operator priority or parentheses! For simplicity, you can assume that the sequence is valid and that the numbers are single digit only.

   Hint: The parentheses problem from supervision 3 is somewhat related to this.

   Note: You cannot use the `eval` method of Python or its equivalent in another language. The goal is to *implement* the `eval`.

2. Show how to compute the number of valid sequences of parentheses of length n? For example, if $n = 4$ there are only two such sequences: `()()` and `(())`.

3. Given a one directional linked list. Using only *constant* memory print the stored values in reverse order.

4. How would you do an in-order traversal of a BST using *constant* memory? Notice, that the recursive implementation we have seen uses $\Omega(height)$ memory (why?).

2 Data Structures Continued

Some problems are from the lecturer's notes[1] (Robert Harle). The ones marked with * are out of the curriculum and are for you to grasp the concepts better and see how they can be used in real life.

1. Example 7.1
2. Given m sorted lists, each of length n. Explain how to compute their merge efficiently.
3. Example 7.3. Write down a hash function you could use in this case.
4. * In the previous problem we use a hash table to determine correctly spelled words. Tries[2] are a beautiful data structure that could be used in such problems. Explain when you would prefer tries over hash tables and vice-versa.
5. * Given two strings a and b, determine if a is a contiguous substring of b. This problem is actually quite relevant in bioinformatics (a and b are gene sequences). There are beautiful and efficient algorithms to do this[3], but one can do it using hashes as well.

   Let the length of a be n and the length of b be m. If $hash(a) = hash(b[i..i+n))$ then we can be quite confident that a appears in b. This means, we have to compute $h_i = hash(b[i..i+n))$ for all $i = 0..m-n$. Computing h_i by itself clearly takes $\Theta(n)$ time simple computation of $h_1, h_2, \dots, h_{m-n}$ could take $\Omega(nm)$ time.

   Can you come up with a function $hash$ for which, computing a single instance of $hash(a)$ still takes $\Theta(n)$ time, but computing the set of all $h_1, \dots, h_{m-n}$ can be done in $O(m+n)$ time?

   Hint: You have to use the fact that the substrings of b corresponding to h_i and h_{i+1} are almost the same (differ by a character at each end).

3 Graphs

Some problems are from the lecturer's notes[4] (Damon Wischik). The ones marked with * are out of the curriculum and are for you to grasp the concepts better and see how they can be used in real life.

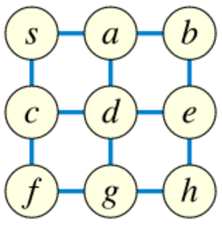
1. Example 1.
2. **This is perhaps the most important question in the whole example sheet.**

   Suppose we run a DFS from source s on a connected graph G (every vertex is reachable from every other one). Each vertex $v \neq s$ is visited from its "parent" p_v. The subgraph of G composed from edges (v, p_v) forms a tree, which we will call a DFS-tree. s is the root of the tree.

   The lecture notes implement `dfs`: an iterative version of DFS using a stack by just taking the BFS implementation and replacing the queue with a stack. There is also an implementation `dfs_recurse`.

[1] `http://www.cl.cam.ac.uk/teaching/1718/Algorithms/2018-examples-rkh.pdf`
[2] `https://en.wikipedia.org/wiki/Trie`
[3] Knuth-Morris-Pratt, look up in Wikipedia.
[4] `http://www.cl.cam.ac.uk/teaching/1718/Algorithms/ex5.pdf`

- Draw the two DFS-trees of the graph below that result from running both versions of DFS.

  Start from a tree containing just a single node s (root). Every time a vertex u becomes visited, add u to the the tree connected it to the vertex v it is visited from.

- Compare these two trees. In particular, consider the edges that are not included *outside* in the DFS-trees.

  The tree $T_{recurse}$ that results from `dfs_recurse` should have the following property: for every edge (u, v) that is not in $T_{recurse}$, either u is an ancestor of v or vice-versa. This is a crucial property of a DFS-tree!

  Does this property hold for the tree T_{stack} of the function `dfs`?

- Fix the `dfs` function to make sure that T_{stack} satisfies the property of a DFS-tree. This is not an implementation detail. This is a major and embarrassing mistake in the `dfs` code that is unfortunately made in the lecture notes, in Wikipedia and in some renowned textbooks.

Note: Example 5 from the sheet is asking you to do the same thing as this problem without realising that by doing so you fix an incorrect implementation.

Note 2: Of course, it's a convention what we call a DFS-tree. The `dfs` from the lecture note does indeed traverse the graph, but this is not the only reason why we might want to use the DFS algorithm. The DFS-tree is used in many problems[5] hence it is "correct" and is the "convention". I do not know an example of a problem where the tree from the `dfs` is useful.

3. Explain why Dijkstra's algorithm is just a generalisation of BFS.

4. Consider a graph G, where each edge has a weight of either 1 or 2. Modify the BFS algorithm to find the shortest paths from a source vertex s to all other vertices.

   Note: Plain BFS does not work in this case and Dijkstra's algorithm is too general. Your algorithm has to trade the generality of Dijkstra with speed.

5. * Example 4.

6. State the invariant that holds in Dijkstra's algorithm and prove the correctness of the algorithm.

7. Suppose a graph G has edges with negative weight. Let the minimum of those weights be $w_0 < 0$. Consider the modified graph G', that consists of the same edges but the weights are

[5] e.g. `https://en.wikipedia.org/wiki/Lowest_common_ancestor`

different: for an edge e, $w'(e) = w(e) - w_0 \geq 0$, where w' and w are the weights of edges of G' and G respectively. Since G' has edges with non-negative weights, can we use Dijkstra's algorithm in G' and find the shortest paths in G?

8. We can implement Dijkstra's algorithm in $O(|E| \log |V|)$ time using priority queues (e.g. heaps). Using Fibonacci heaps we can improve it to $O(|V| \log |V|$, however that is quite complicated.

   Modify (actually simplify) Dijkstra's algorithm so that it runs in $O(|V|^2)$ time. Notice that if $|E|$ is close to $|V|^2$ this algorithm is faster.

4 Implementation

Implementations are critical to sharpen your understanding of graph algorithms. I would suggest that you work on these during the break if you do not have time now.

1. Solve this problem using tries (see problem 2.4 above)

   `http://www.spoj.com/problems/PHONELST/`.

2. The following problem is quite challenging but shows a nice combination of two things we have learnt so far: hashing and binary search. It is related to problem 2.5 above.

   `http://www.spoj.com/problems/LPS/`

   Hint: Suppose you have a function `pal_exists(k)` that returns `true` iff the input string contains a palindrome of length k. Suppose it runs in $O(n)$ time where n is the length of the string. Then, you can do a binary search on the length of the palindrome, because if `pal_exists(k) == true` then `pal_exists(k - 2) == true` as well. Thus, your algorithm will run in $O(n \log n)$ time. Notice that you need to consider palindromes of even and odd lengths separately.

   The question now is, how to implement such a function. You can uses rolling hashes for that[6].

3. Sharpen your graph algorithms skills

   `http://www.spoj.com/problems/EZDIJKST/`

[6]`https://en.wikipedia.org/wiki/Rolling_hash`