National Technical University of Athens

School of Electrical and Computer Engineering

Foundations of Computer Science, 2024-25

2nd set of exercises

(algorithmic techniques – graph algorithms – arithmetic algorithms)

Exercise 1. (Minimum number of stops)

Given a directed graph with n vertices (the cities of a country) and m edges (roads) connecting cities. Not all cities are necessarily connected by a road. There may be roads between two cities in both directions, but it is not necessary. The weights of the edges represent the distances between two cities (positive distances). A car starts from city s and wants to reach city t. The car has a known maximum travel distance, assuming it starts with a full tank, of k kilometers; this distance is provided as part of the input. In each city there is a gas station where the car can stop for refueling. Since the delay for refueling is long, we wish to find the minimum number of refueling stops that the car needs to make in order to reach its destination, and the corresponding route.

1. Give the most efficient algorithm possible for this problem. Explain the correctness and calculate the complexity of the algorithm you propose.
2. Note that we do not want to find the route with the minimum total distance, but the one with the fewest possible stops. Give an example in which these two routes differ.
3. Suppose that we also want to find the route with the least total distance among the routes with the fewest stops. Give as efficient an algorithm as possible for this problem, explain its correctness and find its complexity.

Exercise 2. (Distance reduction)

Let $G(V, E)$ be a graph representing the road network of a country with V cities (the vertices of the graph) and E roads (the edges of the graph). Each road e_i has length w_i. We want to add another road $e^{'}$ of length $w^{'}$ between two cities. The candidate pairs of cities for adding the new road are included in a set $E^{'}$, which is given as input. The road to be added must be the one that achieves the maximum possible distance reduction between two given cities v_i and v_j. Describe an efficient algorithm to determine the the best road in this respect. Explain the correctness and give the complexity of your algorithm.

Exercise 3. (2nd-MST) Consider an undirected connected graph $G(V, E, w)$ with positive weights on the edges, and assume that $|E| \geq |V|$ and that all weights of the edges of G are distinct. Let T be the set of all spanning trees of G and let $t \in T$ be a minimum spanning tree of G. The Second Minimum Spanning Tree (2nd-MST) of G is a spanning tree $t' \in T$ such that $w(t') = min_{t'' \in T \setminus \{t\}}\{w(t'')\}$. In other words, the second minimum spanning tree t' is a spanning tree of G that has a weight greater than or equal to the weight of the MST t and less than or equal to the weight of every other spanning tree.

1. Show that the MST of G is unique, and that this is not necessarily true for the 2nd-MST of G.

2. Let t be the MST of G and $t^{'}$ the 2nd-MST; show that t and $t^{'}$ differ by only one edge, i.e., that there exist edges $e \in t$ and $e^{'} \notin t$ such that $t^{'} = t \cup \{e^{'}\} \setminus \{e\}$

3. Let t be a spanning tree of G and, for any pair of vertices $u, v \in V$, let e_{uv}^{max} be the edge of maximum weight on the unique $u \to v$ path in t. Design an $O(|V|^2)$ time algorithm that takes t as input and determines the edge e_{uv}^{max} for all pairs of vertices $u, v \in V$.

4. Design an efficient algorithm that computes the 2nd-MST of G. Explain the correctness and find the complexity of the algorithm you propose.

Exercise 4. (Cycle removal) Let be an undirected graph $G(V, E, w)$ with positive weights w on its edges. We want to remove a set of edges, with the minimum possible total weight, so that the remaining graph is acyclic.

Suggest an efficient algorithm that computes the edges to be removed. Explain the correctness and give the complexity of your algorithm.

Exercise 5. (Find GCD) Consider the following algorithm for finding GCD known as Binary GCD.

bgcd(a, b): (* assume $a, b > 0$)

- If $a = b$ return a
- if a, b are even, return $2 \cdot$ bgcd$(a/2, b/2)$
- if a is even and b odd return bgcd$(a/2, b)$, and respectively if b is even and a odd
- if a, b are odd, return bgcd$(\min(a, b), |a - b|/2)$

(a) Prove the correctness of Binary GCD.

(b) What is its complexity and why?

(c) Implement it and compare its efficiency with that of the Euclidean algorithm. Test the two algorithms with at least 10 pairs of very large numbers.

Exercise 6. (Repeated squaring – primality tests)

(a) Write a program in a language of your choice (should support operations with 100-digit numbers) that checks whether a number is prime by using Fermat's test:

If n is prime then for every a s.t. $1 < a < n - 1$, it holds that

$$a^{n-1} \bmod n = 1$$

So, if for a given n, a number a is found such that the above equality does not hold, then number n is definitely composite. If the equality holds for the given a, then the test must be repeated with a new a, since it is possible that the number is composite and yet the equality holds for some values of a. An interesting property says that if n is composite, the probability that the equality holds is $\leq 1/2$ (this is true for all n except for some cases, called Carmichael numbers, see question (b) below). Thus, we can significantly increase the probability of success (i.e., of confirming that number n is composite) by repeating the test a few times (typically 30 times) with a different a. If all times the above equality is found to hold, then we say that n "passes the test" and we declare n to be prime number; if the test fails even once, then we are sure that the number is composite.

Your program should work correctly for numbers of thousands of digits. Try it with the numbers:

$$67280421310721, 170141183460469231731687303715884105721, 2^{2281} - 1$$

Note: $a^{2^{2281}-2}$ has an "astronomically" large number of digits (it doesn't even fit in the whole universe!), while $a^{2^{2281}-2} \bmod (2^{2281} - 1)$ is relatively "small" (it has some hundreds of decimal digits *only* ☺) so it is possible to compute it (with some careful implementation).

(b) There are (few) composite numbers that have the property of passing the Fermat test for any a that is relatively prime to n, so for them the test will will fail no matter how many tests are performed (unless by chance we get a that is not relatively prime with n, which is quite unlikely for large enough n). These numbers are called *Carmichael* – see also `http://en.wikipedia.org/wiki/Carmichael_number`. Check your function with *sufficiently large* Carmichael numbers, which you can find e.g. at `http://de.wikibooks.org/wiki/Pseudoprimzahlen:_Tabelle_Carmichael-Zahlen`. What do you observe?

(c) Design and implement the Miller-Rabin test which is an improvement of Fermat's test and gives a correct answer with probability at least 1/2 for *any* natural number (so with 30 iterations we have negligible probability of error for each input number). Test it with various Carmichael numbers. Do you see any strange results? If so, how would you explain them?

(d) Write a program that finds all prime Mersenne numbers, i.e. of the form $n = 2^x - 1$ with $1 < x < 200$ (note that if x is not prime, neither is $2^x - 1$ prime – can you prove it?). Compare your results with what is stated at `https://www.mersenne.org/primes/`.

Exercise 7. (Fibonacci Numbers)

(a) Implement and compare the following algorithms for computing the n-th Fibonacci number: recursive with memorization, iterative, and the algorithm using 2×2 matrices.

Implement the algorithms in a language that supports very large integers (100s of digits), e.g., in Python. Use the integer multiplication provided by the language. What do you conclude?

(b) Try to solve the above problem by exponentiation, using the relation of F_n to ϕ (golden ratio). What do you observe?

(c) Implement a function, as efficient as possible, that takes as input two positive integers n, k and computes the k least significant digits of the n-th Fibonacci number.

Assuming $n = 10^i, k = 17$ find the largest i for which your function gives a correct result within 1 sec.

(d) Search for and consider the Fast Doubling method to solve question (c). Compare it with the method of using matrices 2×2 theoretically and computationally.

Exercise 8. (Divide and Conquer Algorithms)
[exercise from the book *Algorithms*, by Dasgupta, Papadimitriou, Vazirani]

A sequence of elements $A[1], \ldots, A[n]$ is said to have a *majority element* if more than half of the elements in the sequence are the same. Given a sequence, the goal is to design an efficient algorithm that determines whether the sequence has a majority element, and if so, to find this element. The elements of the sequence do not necessarily come from some ordered field of values such as integers, and so there cannot be comparisons of the form "is $A[i] > A[j]$?". (For example, you can consider the

sequence elements to be GIF files.) However, you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

1. Show how to solve this problem in time $O(n \log n)$. (Hint: Divide the sequence A into two sequences A_1 and A_2 of half the size. Does learning the majority elements of A_1 and A_2 help you find the majority element of A; (If so, you can use a 'divide and conquer' approach.)

2. Can you find a linear-time algorithm?

   (Hint: consider another 'divide and conquer' approach:

   - Combine the elements of A in an arbitrary way to create $n/2$ pairs.
   - Examine each pair: if the two elements are different, discard them both. If they are the same, keep only one of them.

     Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element in case the sequence A has such an element.)

Deadline for submission and instructions. Your deadline for submission is 15/12/2024, exclusively through Helios (make sure the final file is <5MB in total).

It is strongly recommended that you dedicate enough time to solving the exercises on your own before seeking assistance from external sources (such as the internet, literature, or discussions with peers). In any case, the solutions should be strictly *individual*.

To be graded, you will need to briefly present your solutions on a date and time that will be announced later.

For questions/clarifications: send an email to `focs@corelab.ntua.gr`.