CS 473: Algorithms, Fall 2018
Midterm I: October 2, 2018, 7pm-9pm, DCL 1320

Version: **1.1**

1 BEWARE OF ORACLES COLORING GRAPHS. (25 PTS.)

[A question from the example midterm with a small twist]

You are given an oracle $f(\mathsf{H})$, which in constant time can tell you whether or not the input graph H is 7-colorable. Describe an algorithm, using this oracle, as efficient as possible, that outputs for a given graph G with n vertices and m edges, the 3-coloring (note the 3 in the desired coloring vs. the 7 in the oracle) of the graph if it exists. Namely, it either outputs that G can not be colored by three colors, or it output a valid 3-coloring for the vertices of G. What is the running time of your algorithm?

Solution:

Algorithm. Assume $\mathsf{V}(\mathsf{G}) = \{v_1, \ldots, v_n\}$. Add a set of four vertices $U = \{v_{n+1}, v_{n+2}, v_{n+3}, v_{n+4}\}$ to $\mathsf{V}(\mathsf{G})$, and connect each one of the new vertices to all the other vertices in the graph (including the new vertices). Let H be the resulting graph.

Observe that H is 7-colorable $\iff$ G is 3-colorable. As such, call $f(\mathsf{H})$ to verify that it is 7-colorable. If not, output that the graph is not colorable, and return.

Let $\mathsf{G}_0 = \mathsf{G}$. Pick an arbitrary vertex $r \in \mathsf{G}$. In the ith iteration, let G'_i be the result of adding the edge rv_i to G_{i-1} (if $r = v_i$ or the edge already exists in the graph then we do not add an edge). Call $f(\mathsf{G}'_i)$, and if it is true, then set $\mathsf{G}_i = \mathsf{G}'_i$. Otherwise, set G_i to be G_{i-1}. Do this for $i = 1, \ldots, n$ (i.e., all the original vertices).

Next, pick an arbitrary vertex g that is connected to r in G_n. Repeat the above addition process and let G_{2n} be the resulting graph which is 7-colorable by construction. We can now "read" the coloring from this graph. The color of r is red, and the color of g is green. A vertex that is connected to both r and g, is colored by blue, a vertex connected to r (but not to g), is green. Finally, a vertex connected to g (but not to r) is red.

The algorithm outputs this coloring.

Running time. Assuming adjacency matrix representation in the input, this algorithm takes $O(n)$ time, and requires at most $2n - 2$ calls to the oracle f. Using adjacency list representation, and the running time becomes worse – say $O(nm)$ if we do not try to be clever. In any case, this is polynomial.

Correctness. By construction, and by the repeated use of the oracle G_{2n} is 7-colorable $\iff$ G is 3-colorable. Furthermore, since $\mathsf{EG} \subseteq \mathsf{E}(\mathsf{G}_{2n})$, any valid 7-coloring of G_{2n} is a valid 3-coloring of the original G – since the vertices of U get all unique colors, and they connected to all the other vertices in G. So, we remain with the task of arguing that the coloring of G_{2n} is unique. First, observe that as r and g are connected in G_{2n}, it follows that they must have distinct colors, in a valid coloring, and we can declare this colors be whatever we want (i.e., red and green).

Now, consider any other vertex in G_{2n}. By construction, it must be connected to at least one of the vertex out of r and g. If it is connected to both, then it must be colored by the third color – blue. Otherwise, the color assignment done by the algorithm is the only one possible.

Solution:

Alternative solution. For a graph G, and two vertices $u, v \in \mathsf{V}(\mathsf{G})$, such that $uv \notin \mathsf{E}(\mathsf{G})$, let G/uv be the graph resulting from merging the two vertices u and v into a single vertex, where the new vertex $\{u, v\}$ is connected to all the vertices that either u or v were connected to by an edge.

As before, we add a clique U of size 4 and its vertices to the graph. Let G be the resulting graph. If G is 3-colorable, and G/uv is 7-colorable, then we call the pair u, v a ***collapsible*** pair. We can check if a pair u, v is collapsible by performing two calls to the oracle. As long as there is a collapsible pair, the algorithm collapse it, till there is no collapsible pair. The vertices of U do not participate in this (crying) game of collapsing. If the graph is a clique of size 7, then the original graph is 3-colorable, and the vertices that collapse into each of the three vertices (that are not in U), are the three color classes (the same applies if the graph is a subgraph of a triangle). Clearly, after $O(n^2)$ oracle checks, either the algorithm found a collapsible edge, or it had failed. As such, overall, this algorithm, implemented naively would try to collapse $O(n^3)$ pairs, and trying each such pair takes $O(n^2)$ time (implemented naively), as such the overall running time is $O(n^5)$. The running time here can be improved but this is besides the point.

2 Locating burger joints. (25 pts.)

[Similar to homework problem.]

The McYucky chain wants to open several restaurants along Red street in Shampoo-Banana. The possible locations are at $L_1, L_2, \ldots, L_n$ where L_i is at distance m_i meters from the start of Red street. Assume that the street is a straight line and the locations are in increasing order of distance from the starting point (thus $0 \leq m_1 < m_2 < \ldots < m_n$). McYucky has collected some data indicating that opening a restaurant at location L_i will yield a profit of p_i independent of where the other restaurants are located. However, the city of Shampoo-Banana has a zoning law which requires that any two McYucky locations should be D or more meters apart. Moreover, McYucky has realized that its budget constraints imply that it can open at most $k < n$ restaurants. Describe an algorithm that McYucky can use to figure out the locations for its restaurants so as to maximize its profit while satisfying its budget constraint and the city's zoning law.

Solution:

Let $F(i, \ell)$ denote the maximum profit for opening ℓ restaurants while satisfying the distance constraints using locations $L_i, \cdots, L_n$.

$$F(i,l) = \begin{cases} 0 & \ell = 0 \\ 0 & i > n \\ \max\left\{p_i + F(next(i), \ell - 1), \quad F(i+1, \ell)\right\} & \text{otherwise} \end{cases}$$

where $next(i) = \min\{j : m_j \geq m_i + D\}$ is the next available location that is at least D distance away from the current location at L_i, if $next(i)$ is not defined, then set it to $n+1$. All $next(i), \forall i = 1, \cdots, n$ can be calculated in $O(n)$ time.

The initial call to the recurrence will be $F(1, k)$.

The naive way to memoize is to store into a two-dimensional array $F[1, \cdots, n+1; 0, \cdots, k]$. Space for the array is $O(nk)$.

There are $O(nk)$ distinct subproblems. If *next* array is computed in $O(n)$ time then each call to $next(i)$ takes $O(1)$ time. This requires an extra $O(n)$ space to hold the array. Thus, the total runtime is $O(nk)$ and total space is $O(nk)$.

Without the preprocessing *next*, finding $next(i)$ will incur a time cost of $O(\log n)$ (for binary search) or $O(n)$ (for a linear scan of the array) for each subproblem. The total running time will then be $O(nk \log n)$ or $O(n^2 k)$.

Alternatively, notice that a subproblem in the column indexed by l only depends on subproblems in the same column or the previous column indexed by $l-1$. Therefore, only two columns instead of k columns need to be memoized. Such an approach uses $O(n)$ space. We memoize this in an array $F[i]]$ in decreasing values i (from $n+1$ to 1).

Rubric: Standard DP rubric. 25 points total. Furthermore:

- **0 points** for solving the problem without using constraint k.
- **13 points max** for no recurrence/pseudo code but correct description and idea presented.
- **22 points max** for a correct but slower solution(scaled accordingly).

3 A point above all given lines. (25 pts.)

[A combination of two algorithms seen in class.]

Let L be a set of n lines in the plane. Assume the lines are in general position (no line is horizontal, no line is vertical, no three lines meet in a point). Let $\ell^-, \ell^+ \in L$ be the two lines with the maximum and minimum slope in L, respectively. Furthermore, assume that ℓ^+ has a positive slope, and ℓ^- has a negative slope. The task at hand is to compute the lowest point that is above all the lines of L.

Consider the algorithm that randomly permute the lines of L, and let $\ell_1, \ldots, \ell_n$ be the resulting ordering. Let p_i be the lowest point that is above all the lines of $L_i = \{\ell^+, \ell^-\} \cup \{\ell_1, \ldots, \ell_i\}$. The algorithm works as follows. It computes $p_0 = \ell^- \cap \ell^+$. In the ith iteration, for $i = 1, \ldots, n$, it does the following:

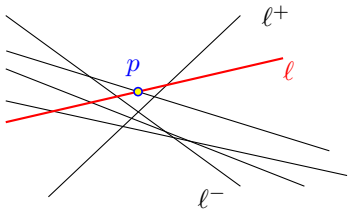
(I) Check if p_{i-1} is above ℓ_i. If so, it sets $p_i \leftarrow p_{i-1}$ and continue to the next iteration. (That is, p_{i-1} is still the lowest point above all the lines of L_i.)

(II) Otherwise, p_i must be on ℓ_i. It computes in $O(i)$ time the lowest point on ℓ_i that is above all the lines of L_{i-1}. It sets p_i to be this point.

The purpose of this question is to describe and analyze this algorithm in more detail.

3.A. (5 pts.) Given a set T of i lines (in general position), such that $\ell^+, \ell^- \in T$, and a line ℓ (which has, say, a positive slope), describe **shortly** how to compute in $O(i)$ time, the lowest point p (i.e., point with minimum value of the y axis) that lies on ℓ, and is above all the lines of T (you can safely assume that such a point exists). (This shows how to implement step (I) above.)

(Hint: First consider the case that ℓ has a positive slope.)



Solution:

Assume ℓ has a positive slope. The negative slope case is handled similarly. By assumption, we need to consider only the intersections of lines with ℓ, where these lines have smaller slope that ℓ. Compute all these intersections, and take the one that has the maximum x coordinate value. Clearly, this is the desired point, and it can be computed in $O(i)$ time.

3.B. (5 PTS.) Give an upper bound to the probability that $p_i \neq p_{i-1}$.

Solution:

The point p_i is different than p_{i-1} if and only if, the ith line inserted by the algorithm is one of the two lines defining p_i. By backward analysis, the probability for this is at most $2/i$.

3.C. (10 PTS.) Let R_i be the running time of the ith iteration. Prove that $\mathbb{E}[R_i] = O(1)$.

Solution:

The time spend in the ith iteration is $O(1)$ if $p_i = p_{i-1}$, and $O(i)$ if $p_i \neq p_{i-1}$. As such, the expected running time is

$$\mathbb{E}[R_i] = O(1) + \mathbb{P}[p_i \neq p_{i-1}]O(i) = O(1) + \frac{2}{i}O(i) = O(1).$$

3.D. (5 PTS.) Provide and prove an upper bound on the expected running time of the algorithm.

Solution:

By linearity of expectations, we have:

$$\mathbb{E}[\text{running time}] = \sum_{i=1}^{n} \mathbb{E}[R_i] = \sum_{i=1}^{n} O(1) = O(n).$$

4 2SUM, 3SUM, FAST SUM. (25 PTS.)

[Similar to homework problem.]

4.A. (5 PTS.) Describe a *simple* algorithm that determines whether a given set S of n distinct integers contains two elements whose sum is zero, in $O(n \log n)$ time.

Solution:

Sort the numbers in S, and let L be the resulting sorted list. Let $-L$ be the result of copying L, reversing its order, and replacing each number by its negation. Merge the two sorted lists. Clearly, there are two numbers in S as desired, if in the merged list there are two consecutive copies of the same number.

4.B. (10 PTS.) Describe an algorithm that determines whether a given set S of n distinct integers contains three (distinct) elements whose sum is zero, in $O(n^2)$ time. For simplicity, you can assume that S does not contain the number 0.

[A correct full solution with running time $O(n^2 \log n)$ would get 12 points. A solution with $O(n^3)$ time would get as many points as IDK.]

Solution:

Sort S, and let $L = s_1, \ldots, s_n$ be the numbers in the sorted order. Let

$$L_i = s_1 + s_i, s_2 + s_i, \ldots, s_{i-1} + s_i, \cancel{s_i + s_i}, s_{i+1} + s_i, \ldots, s_n + s_i.$$

Clearly, the lists $L_1, \ldots, L_n$ are sorted, and can be computed in $O(n^2)$ time. We can decide in $O(n)$ time if L_i contains a number that is in $-L$ (the reverse and negated sorted list) by merging the two lists. IF any of these checks succeed, then we found the desired solution. The running time is clearly $O(n^2)$.

Slower solution: Compute all $\binom{n}{2}$ sums of pairs of numbers in S. Next, sort these numbers, and let U be the sorted set. Finally, for every number $x \in S$, using binary search decide if $-x \in U$. If so, we found the desired sum. If this fail for all tests, then there is no such triple. The running time is $O(n^2 \log n)$.

4.C. (10 PTS.) Now suppose the input set X contains only integers between 1 and $1,000n$. Describe an algorithm, as fast as possible (way faster than (B)), that determines whether X contains three (distinct) elements $x, y, z \in S$, such that $x + y = z$. What is the running time of your algorithm?

Solution:

Sort S, and let $s_1, \ldots, s_n$ be the sorted numbers. Let $p(x) = \sum_{i=1}^{n} x^{s_i}$. Consider the polynomial $q(x) = p(x)p(x)$. The polynomial $q(x)$ has a monomial x^i that appears also in $p(x) \iff$ we have the desired triple. Using FFT, $q(x)$ be computed in $O(n \log n)$ time, and finding the common monomial can be done in linear time.

There is however a minor subtlety (which is not that important) – the polynomial $q(x)$ might contain the monomial x^i with a coefficient 1. This happens only if i is even, and $x^{i/2}$ is a monomial in $p(x)$. In particular, in the above check, we need to ignore these fake news monomials in $q(x)$ (i.e., monomials with coefficients 1) since they represent sums of two numbers that are the same number repeated twice.

Rubric: Remove one point if the subtlety was ignored.