Review for Midterm 2

NEW CS 473: Theory II, Fall 2015 Version: **1.2**

The following is a collection of problems that you should solve in preparation for the second midterm. Some of the following questions might appear in the midterm, as some questions from homeworks, stuff from lectures, etc. We will note provide solutions for this problems, since they are provided so you can improve your ability to solve problems, not your reading ability.

---

1. (25 PTS.) Check the matching.
Given a matching M in a bipartite graph with n vertices and m edges. Provide an algorithm, as fast as possible, for checking if M is a maximum cardinality matching. What is the running time of your algorithm?

2. (25 PTS.) The matching disaster.
You are given a maximum matching M in a bipartite graph G with n vertices and m edges. Due to eddies in the time-space continuum, k edges, $e_1, \ldots, e_k$, got deleted from $\mathsf{G} = (L \cup R, \mathsf{E})$ (some of these edges are in M). Describe an algorithm, as fast as possible (as a function of n, k, m), that computes a maximum matching in the graph $\mathsf{G}' = (L \cup R, \mathsf{E} \setminus \{e_1, \ldots, e_k\})$.

3. (25 PTS.) Prove something.
Prove the following theorem:

Theorem 0.1. *Let M be a matching of maximum weight among matchings of size $|M|$, in a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with weights on the edges. Let π be an augmenting path for M of maximum weight, specifically, it is the augmenting path maximizing*

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \tag{1}$$

Furthermore, let T be the matching formed by augmenting M using π. Then T is of maximum weight among matchings of size $|M| + 1$.

4. (25 PTS.) Cover with cycles.
Given a bipartite graph G with n edges and m edges, describe an algorithm, as efficient as possible, that computes a collection of vertex disjoint cycles that cover all the vertices of G exactly once, if such a collection of cycles exists. What is the running time of your algorithm? Prove the correctness of your algorithm.
[For credit, your solution have to be self contained – references to "as seen in class" or "as done in homework" are not acceptable (but only for this problem).

5. (25 PTS.) Extract heavy matching.
Let G be a graph with n vertices and m edges, which is definitely not bipartite. Describe an algorithm, as efficient as possible, that output, in expectation, a matching of weight $\geq w/2$, where w is the weight of the maximum weight matching in G. What is the running time of your algorithm?

6. (25 PTS.) Even more on much matching over nothing.
Let G be a graph with n vertices and m edges. Imagine, that you are given a black box **aug**, such that given a matching M, returns you the shortest augmenting path for M. In particular, if the augmenting path is of length t, the running time of the black box is $O(t)$.
Describe an algorithm, as efficient as possible, that computes the maximum matching in G. Prove the bound on the running time of your algorithm. (Hint: This algorithm is much faster than you think.)

7. (20 PTS.) Linear time Union-Find,

(a) (2 PTS.) With path compression and union by rank, during the lifetime of a Union-Find data-structure, how many elements would have rank equal to $\lfloor \lg n - 5 \rfloor$, where there are n elements stored in the data-structure?

(b) (2 PTS.) Same question, for rank $\lfloor (\lg n)/2 \rfloor$.

(c) (4 PTS.) Prove that in a set of n elements, a sequence of n consecutive FIND operations take $O(n)$ time in total.

(d) (2 PTS.)
Write a non-recursive version of FIND with path compression.

(e) (6 PTS.) Show that any sequence of m MAKESET, FIND, and UNION operations, where all the UNION operations appear before any of the FIND operations, takes only $O(m)$ time if both path compression and union by rank are used.

(f) (4 PTS.) What happens in the same situation if only the path compression is used?

8. (20 PTS.) Off-line Minimum

The *off-line minimum problem* asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \ldots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \ldots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array $extracted[1 \ldots m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ is the key returned by the ith EXTRACT-MIN call. The problem is "off-line" in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.

(a) (4 PTS.)
In the following instance of the off-line minimum problem, each INSERT is represented by a number and each EXTRACT-MIN is represented by the letter E:

$$4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.$$

Fill in the correct values in the *extracted* array.

(b) (8 PTS.)
To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by

$I_1, E, I_2, E, I_3, \ldots, I_m, E, I_{m+1}$,

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j, we initially place the keys inserted by these operations into a set K_j, which is empty if I_j is empty. We then do the following.

```
OFF-LINE-MINIMUM(m,n)
1  for i ← 1 to n
2      do determine j such that i ∈ K_j
3         if j ≠ m + 1
4            then extracted[j] ← i
5                 let l be the smallest value greater than j for which set K_l exists
6                 K_l ← K_j ∪ K_l, destroying K_j
7  return extracted
```

Argue that the array *extracted* returned by OFF-LINE-MINIMUM is correct.

(c) (8 PTS.)
Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

9. (20 PTS.) How many min-cuts are there anyway?

Prove, that in any undirected graph G, with n vertices, there are at most n^2 minimum cuts. (Hint: Consider the set of minimum cuts, and argue about the probability of a specific minimum-cut in this set to be output by the simple min-cut algorithm.)

10. (20 PTS.) Adapt min-cut

Consider adapting the min-cut algorithm to the problem of finding an *s-t min-cut* in an undirected graph. In this problem, we are given an undirected graph G together with two distinguished vertices s and t. An s-t min-cut is a set of edges whose removal disconnects s from t; we seek an edge set of minimum cardinality. As the algorithm proceeds, the vertex s may get amalgamated into a new vertex as the result of an edge being contracted; we call this vertex the s-vertex (initially s itself). Similarly, we have a t-vertex. As we run the contraction algorithm, we ensure that we never contract an edge between the s-vertex and the t-vertex.

(a) (10 PTS.) Show that there are graphs in which the probability that this algorithm finds an s-t min-cut is exponentially small.

(b) (10 PTS.) How large can the number of s-t min-cuts in an instance be?

11. (20 PTS.) Find kth smallest number.

This question asks you to design and analyze a *randomized incremental* algorithm to select the kth smallest element from a given set of n elements (from a universe with a linear order).

In an *incremental* algorithm, the input consists of a sequence of elements $x_1, x_2, \ldots, x_n$. After any prefix $x_1, \ldots, x_{i-1}$ has been considered, the algorithm has computed the kth smallest element in $x_1, \ldots, x_{i-1}$ (which is undefined if $i \leq k$), or if appropriate, some other invariant from which the kth smallest element could be determined. This invariant is updated as the next element x_i is considered.

Any incremental algorithm can be *randomized* by first randomly permuting the input sequence, with each permutation equally likely.

(a) (5 PTS.) Describe an incremental algorithm for computing the kth smallest element.

(b) (5 PTS.) How many comparisons does your algorithm perform in the worst case?

(c) (10 PTS.) What is the expected number (over all permutations) of comparisons performed by the randomized version of your algorithm? (Hint: When considering x_i, what is the probability that x_i is smaller than the kth smallest so far?) You should aim for a bound of at most $n + O(k \log(n/k))$. Revise (A) if necessary in order to achieve this.

12. (20 PTS.) Sorting Random Numbers

Suppose we pick a real number x_i at random (uniformly) from the unit interval, for $i = 1, \ldots, n$.

(a) (5 PTS.) Describe an algorithm with an expected linear running time that sorts $x_1, \ldots, x_n$.

To make this question more interesting, assume that we are going to use some standard sorting algorithm instead (say merge sort), which compares the numbers directly. The binary representation of each x_i can be generated as a potentially infinite series of bits that are the outcome of unbiased coin flips. The idea is to generate only as many bits in this sequence as is necessary for resolving comparisons between different numbers as we sort them. Suppose we have only generated some prefixes of the binary representations of the numbers. Now, when comparing two numbers x_i and x_j, if their current partial binary representation can resolve the comparison, then we are done. Otherwise, the have the same partial binary representations (upto the length of the shorter of the two) and we keep generating more bits for each until they first differ.

(b) (10 PTS.) Compute a tight upper bound on the expected number of coin flips or random bits needed for a single comparison.

(c) (5 PTS.) Generating bits one at a time like this is probably a bad idea in practice. Give a more practical scheme that generates the numbers in advance, using a small number of random bits, given

an upper bound n on the input size. Describe a scheme that works correctly with probability $\geq 1-n^{-c}$, where c is a prespecified constant.

13. (QuickSort is fast. pts.) Prove, that with high probability, **QuickSort** runs in $O(n \log n)$ time.

14. (From expectation to high probability. pts.)
Consider a randomized algorithm **alg**, that for an input of size n, has expected running time, say, $M = O(n^3)$. Give a new algorithm that runs in $O(n^3 \log n)$ time with high probability (say, with probability $\geq 1 - 1/n^{10}$). (Hint: Run **alg** as before - if it takes too much time, abort its execution, and starts from scratch. Argue that this restarting can not happen to many times, with high probability.)
Consider a randomized algorithm **alg**, that for an input of size n, has expected running time, say, $M = O(n^3)$. Give a new algorithm that runs in $O(n^3 \log n)$ time with high probability (say, with probability $\geq 1 - 1/n^{10}$). (Hint: Run **alg** as before - if it takes too much time, abort its execution, and starts from scratch. Argue that this restarting can not happen to many times, with high probability.)