Computer Science CSC263H
St. George Campus

February 14, 2019
University of Toronto

Homework Assignment #4

Due: February 28, 2019, by 5:30 pm

- You must submit your assignment as a PDF file, named **a4.pdf**, of a typed (**not** handwritten) document through the MarkUs system by logging in with your CDF account at:

  `https://markus.teach.cs.toronto.edu/csc263-2019-01`

  To work with one or two partners, you and your partner(s) must form a group on MarkUs.

- The **a4.pdf** PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTeX typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can use other typesetting systems if you prefer, but handwritten documents are not accepted.

- If this assignment is submitted by a group of two or three students, the **a4.pdf** PDF file that you submit should contain for each assignment question:

  1. The name(s) of the student(s) who *wrote* the solution to this question, and
  2. The name(s) of the student(s) who *read* this solution to verify its clarity and correctness.

- By virtue of submitting this assignment you (and your partners, if you have any) acknowledge that you are aware of the homework collaboration policy that is stated in the csc263 course web page: `http://www.cs.toronto.edu/~sam/teaching/263/#HomeworkCollaboration`.

- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.

- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.

- Your **a4.pdf** submission should be no more than 4 pages long in a 10pt font.

Question 1. (1 marks)

Given an array $A[1..n]$ of integers and an integer x to search for, consider the following "lucky search" algorithm, where r is an arbitrary integer such that $r \geq 1$:

```
I := {1, 2, ..., n}                    // set of indices
repeat r times
    pick i uniformly at random from I
    if x = A[i] then return TRUE       // x was found
end repeat
return FALSE                           // x was not found
```

Note that there are circumstances under which this algorithm may return FALSE even if x is in $A[1..n]$.

In the following, suppose there are ***exactly*** k ***copies*** of integer x in $A[1..n]$.

a. What is the probability that this algorithm returns TRUE in the ***first*** iteration of the repeat loop? Justify your answer.

b. What is the probability that this algorithm returns TRUE? Justify your answer.

c. We now modify the algorithm as follows:

```
I := {1, 2, ..., n}                    // set of indices
repeat forever
    pick i uniformly at random from I
    if x = A[i] then return TRUE       // x was found
end repeat
```

What is the *expected* number of loop iterations of this algorithm? Justify your answer.

Question 2. (1 marks)

You are given a list of m *constraints* over n distinct variables $x_1, x_2, ..., x_n$. Each constraint is of one of the following two types.

1. An *equality* constraint of the form $x_i = x_j$ for some i, j where $1 \leq i \neq j \leq n$.
2. An *inequality* constraint of the form $x_i \neq x_j$ for some i, j where $1 \leq i \neq j \leq n$.

For such a list of constraints, it may be possible to assign an integer to each variable such that this assignment does not violate any of the constraints in this list, or such assignment may not exist.

For example, the assignment of $(1, 2, 1, 1, 2)$ to the variables x_1, x_2, x_3, x_4, x_5, satisfies all the constraints in the list:

$$x_1 = x_4, x_1 = x_3, x_1 \neq x_2, x_2 \neq x_3, x_2 = x_5$$

However, ***no*** possible assignment of integers to the variables $x_1, x_2, x_3, x_4, x_5, x_6, x_7$, can satisfy all the constraints in the list:

$$x_5 = x_1, x_1 = x_6, x_4 = x_5, x_1 = x_3, x_2 \neq x_6, x_2 = x_7, x_3 \neq x_4$$

Design an efficient algorithm, which takes as input a list of m constraints over n variables, and outputs an assignment of integers to variables that satisfies all the constraints in this list, and if no such assignment exists the algorithm outputs NIL. Describe your algorithm in *clear and concise* English and analyze its worst-case time complexity. ***The worst-case running time of your algorithm must be asymptotically better than*** $O(mn)$***.***

HINT: Use a data structure that we learned in class.

[***The questions below will not be corrected/graded. They are given here as interesting problems that use material that you learned in class.***]

Question 3. (0 marks) Consider the sorting algorithm given by the pseudocode below. It takes an array $A[1\,..\,n]$ of size n, and outputs A with its elements in sorted (non-decreasing) order.

```
1  for i = 2 to n
2      j = i − 1
3      while A[j + 1] < A[j]  &  j ≥ 1
4          swap A[j] and A[j + 1]
5          j = j − 1
```

In the following subquestions, assume that the array A contains a uniformly chosen random permutations of the integers $1, \ldots, n$.

a. Let S_i be the number of swaps performed by the algorithm in the i-th iteration of the for-loop. What is the ***exact expected value*** of S_i as a function of n and i? Justify your answer.

b. Let $S = S_1 + \ldots + S_{n-1}$ be the total number of swaps performed by the algorithm. What is ***exact expected value*** of S as a function of n? Justify your answer.

Question 4. (0 marks) Assume you have a biased coin, which, when flipped, falls on Heads with probability p, where $0 < p < 1$, and on Tails with probability $1 - p$. However, you do not know p. How can you use the coin to simulate an unbiased coin? Formally, you have access to a procedure FlipBiasedCoin(), which returns either 1 or 0 at random. FlipBiasedCoin() returns 1 with probability p, and 0 with probability $1 - p$, but you do not know p. Design an algorithm that, without making any other random choices except calling FlipBiasedCoin(), returns 1 with probability 1/2 and 0 with probability 1/2. Your algorithm should not use p. You can assume that each time you call FlipBiasedCoin(), the value it returns is independent of all other calls to it.

a. Describe the algorithm in clear and concise English, and prove that it outputs 0 with probability 1/2 and 1 with probability 1/2.

b. Analyze the expected running time of your algorithm. Note that while the algorithm itself does not use p, the expected running time should be expressed in terms of p.

Question 5. (0 marks) Consider the following "Graph Dismantling Problem". Let $G = (V, E)$ be a connected undirected graph with $n \geq 4$ nodes $V = \{1, 2, \ldots, n\}$ and m edges. Let $e_1, e_2, \ldots, e_m$ be all the edges of G listed in some specific order. Suppose that we ***remove*** the edges from G one at a time, ***in this order***. Initially the graph is connected, and at the end of this process the graph is disconnected. Therefore, there is an edge e_i, $1 \leq i \leq m$, such that just before removing e_i the graph has at least one connected component with more than $\lfloor n/4 \rfloor$ nodes, but after removing e_i every connected component of the graph has at most $\lfloor n/4 \rfloor$ nodes. Give an efficient algorithm that determines this edge e_i. Assume that G is given to the algorithm as a (plain) linked list of the edges appearing in the order $e_1, e_2, \ldots, e_m$.

The worst-case running time of your algorithm ***must be asymptotically better than*** $O(mn)$***.***

Hint: See "An application of disjoint-set data structures" in pages 562-564 of CLRS (Chapter 21).

Question 6. (0 marks) Consider the forest implementation of the disjoint-sets abstract data type, with an initial forest of n distinct elements (each one in a one-node tree). Let σ be any sequence of k UNIONs followed by k' FINDs; so *all* UNIONs occur before the FINDs. Prove that the algorithm using Path Compression only (it does *not* use the Weighted-Union rule) executes σ in $O(k + k')$ time, i.e., in time proportional to the length of σ, in the worst-case.

Do *not* make assumptions on k or k' (for example, do not assume that $k = n - 1$ or that $k' \leq k$). As we did in class, assume that the parameters of each UNION are two set representatives, i.e., two tree roots (so there are *no* FINDs "inside" each UNION).

HINT: Note that if a vertex becomes a child of a root during the execution of one of the FINDs (because of Path Compression), then it remains a child of this root during all the subsequent FINDs. Use this to compute the "cost" of executing all the k' FINDs.