CPSC 320 2016W1: Assignment 1

September 12, 2016

Submit this assignment via handin (see the syllabus for more information) to the target `assn1` by the deadline **Thursday 22 Sep at 5PM**. For credit, your submission **must** be a clearly legible PDF file with clearly indicated solutions to the problems.

For credit, your group must make a **single** submission via one group member's account. Your submission must:

- Be on time.
- Consist of a single, clearly legible PDF file. (Directly produced via LaTeX, Word, Google Docs, or other editing software is best. Scanned is fine. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout (not the individual quiz postings). Put these **in order**, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the names and ugrad.cs.ubc.ca account IDs of each member of your team.
- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff" (Go read those guidelines!)
- Include at the start of the document acknowledgment of collaborators and references (with the exceptions listed in the conduct guidelines).

1 Finding Some Middle Ground

You are designing a median-finding data structure that has two operations: $\textsc{Insert}(M, x)$ inserts the number x into median data structure M, and $\textsc{FindMedian}(M)$ produces the median of the numbers inserted so far into median data structure M (with a precondition that M is non-empty).

We describe each operation's performance in terms of the size of M (i.e., the number of `insert` operations so far), which we call n. The median of an odd number of elements is well-defined. For an even number of elements, we take the smaller of the two middle elements to be the median.

Consider the following approach using binary heaps (min- and max-heaps, both stored in resizable arrays). Heaps support four operations: Size, FindMin (or FindMax as appropriate), DeleteMin (or DeleteMax as appropriate), and Insert (distinct from the median data structure's Insert).

IMPLEMENTATION APPROACH: The median data structure implementation contains two fields Left and Right. Left is initialized to an empty binary **max**-heap and contains the left (smaller) half of the elements seen so far. Right is initialized to an empty binary **min**-heap and contains the right (larger) half of the elements seen so far. The data structure supports two procedures INSERT and FIND-MEDIAN:

```
procedure INSERT(medianDS, $x$)
    if SIZE(medianDS.Left) $= 0$ or $x \leq$ FINDMAX(medianDS.Left) then
        INSERT(medianDS.Left, x)
    else
        INSERT(medianDS.Right, x)
    end if
    if SIZE(medianDS.Left) $>$ SIZE(medianDS.Right) then
        $t \leftarrow$ FINDMAX(medianDS.Left)
        DELETEMAX(medianDS.Left)
        INSERT(medianDS.Right, $t$)
    else if SIZE(medianDS.Right) $>$ SIZE(medianDS.Left) then
        $t \leftarrow$ FINDMIN(medianDS.Right)
        DELETEMIN(medianDS.Right)
        INSERT(medianDS.Left, $t$)
    end if
end procedure

procedure FINDMEDIAN(medianDS, $x$)
    return FINDMAX(medianDS.Left)
end procedure
```

1.1 Bug in the Implementation

The data structure's implementation has a small bug.

1. Give the *shortest possible* sequence of INSERT and/or FINDMEDIAN commands that illustrates the bug. (Substantial partial credit is available for "almost-shortest" answers.)

2. Indicate what the implementation above does and also what **should** happen for your commands.

3. Fix the bug in the code above.

1.2 Asymptotic Bounds

Give and **briefly** justify asymptotic bounds on the worst-case runtime performance of a single call to each of the data structure's two operations in terms of n for the corrected code (or, if you don't see the bug, for the existing code, assuming it runs fine).

1. FINDMEDIAN:

2. INSERT:

1.3 An Alternate Approach

Give an alternate approach that does not use heaps and is correct. It may use any common data structures you like and be as (in)efficient as you like but should be clear and correct. Use comments to highlight the key insights and invariants in your data structure that show why it is correct. (1 Bonus Point for a good approach, 2 Bonus Points for the best one; purely subjective marker opinion!)

2 Two's a Crowd, Three's Company

Logs from a web server include one line per access to the system (ordered by time of access) with a user ID (a string) on each line (plus other fields we don't care about). Unusual accesses may suggest security concerns. In this problem we are identifying the first user ID that only ever accessed the system once.

We will use n—the total number of entries in the log—to describe problem size. Note: **assume** that comparing two user IDs (strings) for equality or order takes constant time.

2.1 Brute Force

Consider this algorithm that attempts to solve the problem by brute force:

```
for each user ID $s_i$ in order by indexes $i$ do
    found ← false
    for each later user ID $s_j$ in order do
        if $s_i = s_j$ then
            found ← true
        end if
    end for
    if found is false then
        return $i$
    end if
end for
return None
```

Now, answer the following questions:

1. Give and **briefly** justify (perhaps including annotating the algorithm) a good asymptotic bound on the algorithm's worst-case runtime in terms of n.
2. If the algorithm is **correct**, briefly sketch (only the key elements/insights in) a proof of its correctness.

   If the algorithm is **incorrect**, illustrate the fact by giving the smallest possible example (in terms of n) on which it fails and explaining what the algorithm does and what **should** happen.

   Justification:

2.2 Tracking Uniqueness and Indexes

The following algorithm solves the same problem using self-balancing BSTs:

```
User ← an empty self-balancing BST (that will map indexes to user IDs)
Index ← an empty self-balancing BST (that will map user IDs to indexes)
for each user ID $s_i$ in order by indexes $i$ do
    if Index does not contain $s_i$ then
        Index[$s_i$] ← $i$
        User[$i$] ← $s_i$
    else
        Delete Index[$s_i$] from User (if it is present)
    end if
end for
if User is empty then
    return None
else
    return the value (ID) of the minimum key (index) in User
end if
```

Now, answer the following questions:

1. Give and **briefly** justify (perhaps including annotating the algorithm) a good asymptotic bound on the algorithm's worst-case runtime in terms of n.
2. This algorithm **is correct**. **Briefly** sketch (only the key elements/insights in) a proof of its correctness. (E.g., you will want to justify that every ID that is in User after the loop ends appears exactly once in the log.)

3 Open-Faced Marriage

A group of n people are presented with a set of n sandwich options. We need to provide one sandwich to each person. Each person ranks the sandwich options in their own preference order.

3.1 A New Definition of Instability

The notion of an "instability" still applies well to this problem, but it's a different instability. **Assuming** everyone does indeed want **exactly one** sandwich, what kind of instability could cause problems in the allocation of sandwiches to people? Clearly and concisely define an instability.

3.2 The Dangers of Random Sandwich Preferences

A friend proposes that we generate random preference lists for the sandwiches. (So, each sandwich would have a preference order over all the people.) Then, we run Gale-Shapley with people as the proposers, with the intent of guaranteeing that the sandwich allocation primarily reflects the people's preferences.

This approach does not work in general. Complete the following example input by choosing preferences for p_3 and the sandwiches such that G-S with people proposing produces a matching that will cause problems in practice between p_1 and p_2.

```
p1: s1 s2 s3      s1:

p2: s2 s1 s3      s2:

p3:               s3:
```

Next, give the matching G-S with people proposing would produce on your example: **Finally**, explain what the problem with this matching is.

4 A Marriage of Utility

We sometimes use the notion of "utility" to reason about preferences, as in the stable marriage problem.

Rather than having all women **rank** all men and vice versa, we have each woman **rate** each man (and vice versa). To rate man m_j, woman w_i assigns him an integer from 0 to 100—which we designate $w_i(m_j)$. w_i prefers m_j to m_k if and only if she rates m_j higher than m_k; that is, $w_i(m_j) > w_i(m_k)$. Furthermore, we'll insist that a woman's ratings for any two men are distinct (i.e., for any woman w_i there are no two different men m_j and m_k such that $w_i(m_j) = w_i(m_k)$) and similarly for men rating women.

Finally, we assume that a difference of 1 unit of utility means the same thing for everyone at all points in their scales (so we can compare people's ratings to each other, meaningfully add and subtract ratings, and so forth).

4.1 Failure of Distinctness

Explain why we cannot possibly ensure our "distinctness" criterion holds as the instance's size scales up. (In subsequent parts, assume this problem has been fixed.)

4.2 Converting to SMP

Given a list L of one woman's ratings of all the men—where $L[1]$ is her rating for m_1, $L[2]$ is her rating for m_2, $\ldots$, and finally $L[n]$ is her rating for m_n—give an algorithm to convert that into a preference list. Again, assume all ratings are distinct.

4.3 Comparing Utilities and Preferences

In this part, you will show that utility **may** be a better measure than stability for the quality of an assignment. Be sure to read **all** the questions here before answering any.

1. Give a **small** instance of this utility-rating problem (that will satisfy all subsequent parts).
2. Give the corresponding preference lists for men and women.
3. Give a stable matching for this instance.
4. Give an **unstable** matching for the same instance that is **much better** in terms of utilities than the stable one. **Explain** why the unstable matching is so much better in terms of utilities!

4.4 Maximum Matching

For a weighted bipartite graph, it's possible to efficiently find a maximum matching: a matching with maximal total edge weight.[1]

1. Give a reduction from the utility-based marriage problem to maximum matching on a weighted, bipartite graph.
2. Give at least one measure of the "goodness" of a solution for which your reduction produces an optimal result. Briefly explain why your reduction produces an optimal result.
3. Give at least one measure of the "goodness" of a solution for which your reduction does **not** produce an optimal result. Use a small example to illustrate how the reduction fails.

[1] You probably do not know what some of these terms mean. If not, find out!