

CPSC 320 2016W1: Assignment 2

September 25, 2016

Submit this assignment via handin (see the syllabus for more information) to the target `assn2` by the deadline **Thursday 6 Oct at 10PM** (note the change in time, thanks to your TAs' suggestions). For credit, your group must make a **single** submission via one group member's account. Your group's submission **must**: **must** be a clearly legible

- Be on time.
- Consist of a single, clearly legible PDF file named `solution.pdf` with clearly indicated solutions to the problems. (Directly produced via L^AT_EX, 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 Olympic Scheduling

You are in charge of a live-streaming YouTube channel for the Olympics that promises never to interrupt an event. (So, once you start playing an event, you must play only that event from the time it starts to the time it finishes.) You have a list of the events, where each event includes its: **start time, finish time (which must be after its start time), and expected audience value**. Your goal is to **make a schedule to broadcast the most valuable complete events. The best schedule is the one with the highest-valued event; in case of ties, compare second-highest valued events, and so on.** (So, for example, you obviously **will** include the single highest-valued event in the Olympics—presumably the hockey gold medal game—no matter what else it blocks you from showing.)

(Times when you're not broadcasting events will be filled with "human interest stories" that have zero value; so, they're irrelevant.)

ASSUME: all event values are distinct and all event times are distinct. I.e., for any two values v_i and v_j with $i \neq j$, $v_i \neq v_j$. The same holds for start and end times (e.g., for any two start times s_i and s_j with $i \neq j$, $s_i \neq s_j$). Further, for any two start and finish times s_i and f_j , whether $i = j$ or not, $s_i \neq f_j$.

1.1 Naïve Algorithm

Consider the following algorithm. Assume that deleting an event from a list of events takes constant time.

```
Naive(E):
  result = new empty list of events
  while E is not empty:
    bestEvent = E[0]
    for each e in E:
      if value(e) > value(bestEvent):
        bestEvent = e
    delete bestEvent from E
    for each e in E:
      if start(e) < finish(bestEvent) and finish(e) > start(bestEvent):
        delete e from E
    add bestEvent to result
  return result
```

1.1.1 Finiteness

Briefly sketch a proof that the **while** loop in the algorithm above terminates. You need not give a formal proof, but you should include all key insights in the proof.

1.1.2 Efficiency

Give and briefly justify a good asymptotic bound on the runtime of the algorithm.

1.1.3 Correctness

Briefly sketch a proof that the algorithm is correct. You need not give a formal proof, but you should include all key insights in the proof.

1.2 Reduction on Simplified Problem

To make the Olympic Broadcasting problem simpler, we completely remove start time and finish times from the problem. So, now events only have values (not times), and a “schedule” is just a set of selected events. To make it slightly harder again, you are not allowed to select two events i and j if their values are within 10 units of each other: $|v_i - v_j| \leq 10$.

Give a correct reduction from this simplified Olympic Broadcasting problem to the sorting problem (where you provide both a list of items and a function to compare two items). Your reduction should take $O(n \lg n)$ time.

NOTE: You will likely find that (a) you can solve this with a single call to the sorting problem’s solution algorithm and (b) producing the sorting instance is the easier part and transforming the solution to sorting into a solution to this simplified Olympic Broadcasting problem is the harder part. Don’t forget to do both!

1.3 Olympic Reduction, BONUS ONLY

This was significantly harder than we intended it to be! So, we removed it from the quiz/assignment. It’s a bonus problem worth two CPSC 320 bonus points for extremely clear, correct, and efficient responses. (Extremely clear reductions that take $O(n)$ time—not counting an $O(1)$ number of calls to a sorting algorithm—may receive 3 bonus points, but we don’t know if such reductions are possible.)

Give a correct and efficient reduction from the Olympic broadcasting problem to the sorting problem (where you provide both a list of items and a function to compare two items). Your reduction—combined with an $O(n \lg n)$ sorting algorithm—should be asymptotically more efficient than the naïve algorithm above.

1.3.1 Correctness

Briefly sketch a proof that your algorithm is correct. You need not give a formal proof, but you should include all key insights in the proof.

1.3.2 Efficiency

Give and briefly justify a good asymptotic bound on the runtime of **just** your reduction, **not** including the call to the sorting algorithm. So, for the purposes of this asymptotic bound, you can imagine that we somehow solve sorting in constant time. (Note: it's possible to give a reduction that takes $O(n)$ time.)

2 Exhausted of Marriage

We modify SMP with the very reasonable change that not every woman need list every man in her preferences. She prefers to be unmarried to marrying unlisted men. Note that she clearly prefers any man on her preference list to any man not on her preference list. Men can similarly truncate their lists of women.

Here is the Gale-Shapley algorithm:

```
1: procedure STABLE-MARRIAGE( $M, W$ )
2:   initialize all men in  $M$  and women in  $W$  to unengaged
3:   while an unengaged man with at least one woman on his preference list remains do
4:     choose such a man  $m \in M$ 
5:     propose to the next woman  $w \in W$  on his preference list
6:     if  $w$  is unengaged then
7:       engage  $m$  to  $w$ 
8:     else if  $w$  prefers  $m$  to her fiancé  $m'$  then
9:       break engagement of  $m'$  to  $w$ 
10:      engage  $m$  to  $w$ 
11:     end if
12:     cross  $w$  off  $m$ 's preference list
13:   end while
14:   report the set of engaged pairs as the final matching
15: end procedure
```

With one small change, we can apply this algorithm and ensure that the (not necessarily perfect) matching produced never marries a person to someone they left off of their preference list.

1. Make the small change necessary **to the algorithm above**.
2. Briefly sketch the key elements of a proof that the algorithm terminates.
3. We need a new definition of instability now that some people may end up unmarried. Here is one new type of instability that we call an *elopement instability*: m_i and w_j are both unmarried but list each other on their preference lists (in which case they have incentive to break the imposed matching and marry each other).

Describe another new type of instability involving an unmarried woman. (Note: an analogous instability exists involving an unmarried man.)

4. Briefly sketch the key elements of a proof that your modified G-S algorithm cannot generate an elopement instability.

2.1 Even More Exhausted

Briefly sketch the key elements of a proof that your modified G-S algorithm cannot generate any of the other three types of instability (the classic SMP instability, the instability you defined above, and the analogous instability with the roles of men and women swapped).

3 Footblog

The massive social network Footblog tracks relationships based on whether two people have “enemied” each other. (“Enemyship” is a mutual agreement, meaning that a person is not allowed to “enemy” another person unless the other person agrees to “enemy” them back. No one can “enemy” themselves.)

3.1 Isolationism

We investigate whether Footblog’s network is a single connected component.

Footblog’s founder created the first Footblog account, and that account has no “sponsor” (and cannot be assigned one). Every other account must have a single, designated “sponsor” who they have “enemied”. If a sponsors b , we call b the *sponsee* of a .

There are then four major actions to consider on Footblog, some of which involve others as steps:

Joining When a new Footblog member joins, they must do so by choosing as sponsor (and “enemying”) someone already in the network who agrees to be their sponsor (and their enemy). After members join, they’re free to “enemy” and “unenemy” anyone except their sponsor and their sponsees.

Enemying Already described above. Remember that when one person “enemies” another, the other must agree to “enemy” that person back.

Un-Enemying Unlike making an “enemy” link, one person alone can “unenemy” another person, in which case neither “enemies” the other any more.

Change Sponsor If a person wishes to change their sponsor, they must “unenemy” their sponsor and simultaneously “enemy” a new sponsor. The new sponsor must agree to act as sponsor and enemy and **must be a new enemy** (i.e., must not already be the person’s enemy). Note that while a sponsee can choose to change their sponsor, a sponsor cannot choose to change their sponsee.

You may assume these actions never happen in parallel, i.e., a defined sequence occurs of the operations: joining, enemying, un-enemying, and changing sponsors.

1. Based on these rules, sketch a brief proof that when a person changes their sponsor, their new sponsor cannot also be one of their sponsees.
2. Based on these rules, either **sketch the key points in a proof that Footblog’s enemy graph forms a single connected component** or **give a small sequence of actions that creates multiple components**.

Circle one:

SINGLE ONLY

MAY BE MULTIPLE

Provide your proof sketch or example:

3.2 Centrality

Footblog has defined a notion of “centrality” for its users: a user’s “centrality” is the minimum number of people they’d need to go through to get a message to the person farthest from them on the network, following “enemy” links. (The “farthest” person is exactly the one to whom there is the longest minimum-length path of enemies.)

For this problem, **assume that the Footblog network does indeed form a single connected component.**

Briefly describe an algorithm to compute the centrality of a user given a graph G represented as a number of users $n > 0$ (where the users themselves are vertices named $\{v_1, v_2, \dots, v_n\}$, a vertex number i (where $1 \leq i \leq n$) of the user whose centrality we wish to compute, and an adjacency list A of edges (i.e., an array of linked lists, where the entries in the list $A[j]$ are the vertex numbers of the users j has “enemied”). You may use any common data structures you need. **Your algorithm must run in linear (i.e., $O(n + m)$ for n nodes and m edges) time.**

```
Centrality(n, i, A):  
  // Fill in your algorithm here!
```

4 Heaps of Fun Might Be OK

You’re managing a major online tournament of the hot new game Flappy Squirrel. There are a huge number of users, each with a competitiveness rating (a floating point number). You need an algorithm that—given a desired number of competitors c and a list of these competitiveness ratings (an array A of length n)—returns a list of the c highest ratings. You’re guaranteed that $c \leq n$. (Note: we use 1-based indexing on arrays.)

4.1 Algorithm 1

Give and briefly justify a good asymptotic upper-bound (i.e., big- O bound) on the runtime of the following algorithm to solve this problem. (**Note:** the `buildMaxHeap` operation returns a max-heap built from the elements of a given array of length n in $O(n)$ time.)

```
TopC(A, c):  
  best <- empty list  
  h <- buildMaxHeap(A)  
  for i = 1 to c:  
    add findMax(h) to best  
    deleteMax(h)  
  return best
```

4.2 Algorithm 2

Give and briefly justify a good asymptotic upper-bound (i.e., big- O bound) on the runtime of the following algorithm to solve this problem. (Note: the notation `A[1..c]` produces a list of the elements `A[1]`, `A[2]`, `A[3]`, ..., `A[c]` in $O(c)$ time.)

```
TopC(A, c):  
  for i = 1 to c:  
    maxIndex = i  
    for j = i+1 to n:  
      if A[j] > A[maxIndex]:  
        maxIndex = j
```

```
max = A[maxIndex]
A[maxIndex] = A[i]
A[i] = max
return A[1..c]
```

4.3 Algorithm 3

Give and briefly justify a good asymptotic upper-bound (i.e., big- O bound) on the runtime of the following algorithm to solve this problem.

```
TopC(A, c):
  sort A using an efficient, comparison-based sorting algorithm
  return A[1..c]
```

4.4 Algorithm 4

Give and briefly justify a good asymptotic upper-bound (i.e., big- O bound) on the runtime of the following algorithm to solve this problem. (Note: `Elements(h)` produces all elements in the heap `h` in constant time, but `h` can no longer be used after that point.)

```
TopC(A, c):
  h <- empty min-heap
  for i = 1 to n:
    if Size(h) < c:
      Insert(h, A[i])
    else if A[i] > FindMin(h):
      DeleteMin(h)
      Insert(h, A[i])
  return Elements(h)
```