CPSC 320 2017W2: Assignment 4

BE SURE TO READ THE UNUSUAL, INDENTED DUE DATE NOTES BELOW!

Please submit this assignment via GradeScope at `https://gradescope.com`. Be sure to identify everyone in your group if you're making a group submission. (Reminder: groups can include a maximum of three students; we strongly encourage groups of two.)

Remember that this assignment is based on the collected quizzes and quiz solutions. You will likely want to refer to those where you need more details on assignment questions.

> Submit by the **UNUSUAL** deadline **Thursday 22 Mar at 10PM**. **If your last submission** is no later than Tuesday 20 Mar at 10PM, each member of your group will also receive one bonus point and a tiny, secret smile from a member of the course staff.

For credit, your group must make a **single** submission via one group member's account, marking all other group members and the pages associated with each problem in that submission **using GradeScope's interface**. Your group's submission **must**:

- Be on time.
- Consist of a single, clearly legible file uploadable to GradeScope with clearly indicated solutions to the problems. (PDFs produced via LaTeX, Word, Google Docs, or other editing software work well. Scanned documents will likely work well. **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 quizzes). Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the **ugrad.cs.ubc.ca e-mail addresses** of each member of your team. (Please do **NOT** include your name on the assignment, however.[1])
- 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 (and GradeScope information) 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 your outside-group collaborators' ugrad.cs.ubc.ca IDs, but **not** their names. (Be sure to get those IDs when you collaborate!)

1 The elements go up and down

Let A be an integer array with n elements in total. In the tutorial, you designed an $O(\log n)$ time algorithm to find the largest element of A in the case where A consists of two sections: first one with numbers strictly

[1] If you don't mind private information being stored outside Canada and want an extra double-check on your identity, include your student number rather than your name.

increasing followed by one with numbers strictly decreasing. This algorithm divided the array in thirds, and called itself recursively on two-thirds of the original array.

You also saw that in the case where A consists of three sections: first one with numbers strictly increasing followed by one with numbers strictly decreasing, followed by another section with numbers strictly increasing, then dividing the array into quarters did not help.

Now prove an $\Omega(n)$ lower bound on the worst-case time complexity of any correct algorithm to finds the largest element of A in the case where it consists of three sections as described above.

Hint: show that the task requires looking at every single element of A by filling in a proof structure like this one (where you may will need to fill in the large gaps below):

> We prove by contradiction that any correct algorithm for this problem **must** look at every element in the array. Suppose not...
>
> For each element $A[i]$ the algorithm accesses, we produce the value ... but we do not yet commit to other values. ...
>
> When the algorithm returns its result, we invalidate its answer by selecting the remaining, unaccessed values as follows ...

2 Essay Assignment, Essay Assignment Again

Your company manages a large group of freelance writers. Given a large group of essays (e.g., newspaper articles), you want to assign each writer exactly one essay to write. (We assume the number of writers and essays is equal.)

Each writer gives a non-negative valuation (number) for each essay, essentially how much they like that essay. We assume that valuations are directly comparable and additive; so, e.g., a valuation of 6 for one writer is exactly twice as good as a valuation of 3 for another a valuation of 9 is as much better than 6 as 6 is than 3. However, **essays** have no valuation or preference over writers.

You want to find a valid assignment of essays to writers (a perfect matching) of highest quality.

1. Consider **Algorithm 2** from the quizzes: Repeatedly pick an arbitrary remaining (unassigned) writer. Assign the remaining essay that they value highest to that writer. Repeat until all essays are assigned. (Break ties arbitrarily.)

   (a) Give a very small but non-trivial instance of the problem and the solution produced by this algorithm.

   (b) Sketch the key points in a proof that the following property holds in any solution produced by this algorithm: no two writers would (strictly) prefer to switch essays with each other than complete the essays assigned to them.

   (c) Give and briefly explain a small counterexample to the optimality of the algorithm according to this metric: an optimal assignment maximizes the total for each writer w of w's valuation for the essay assigned to w. **For this part only**, assume that rather than picking an arbitrary writer, the algorithm picks the "first" remaining writer, in the order the writers were initially supplied as input. (I.e., you—perhaps fiendishly—pick a single, consistent order that the algorithm must use as it selects writers to assign.)

2. The "weighted maximum matching" problem is an adaptation of the maximum matching problem to weighted graphs. An instance of the problem is an undirected, weighted graph[2] $G = (V, E)$ where $w(e)$ is an integer weight for edge e. A valid solution is a set of edges $E' \subseteq E$ such that no vertex

[2]With no self-loops and at most a single edge between two vertices. I.e., a normal undirected, weighted graph.

is incident on two different edges in E'. An optimal solution is a valid solution of maximum total weight $\sum_{e \in E'} w(e)$.

Give a good reduction from our essay assignment problem—with the metric that an optimal assignment maximizes the total of each writer's valuation of their assigned essay—to weighted maximum matching.

3. Consider **Algorithm 1** from the quiz: Repeatedly pick the remaining (unassigned) essay with the highest valuation from any one remaining writer. Assign that essay to that writer. Repeat until all essays are assigned. (Break ties arbitrarily.)

   Somewhat surprisingly, this can be considered an implementation of **algorithm 2** (described above). Briefly but clearly justify this statement.

3 Marvelous Medians

Suppose that we are given an unsorted array A with n elements, and another *sorted* array *Positions* with k distinct elements chosen from the set $\{1, 2, \ldots, n\}$. In the tutorial, you gave an $O(n \log k)$ time algorithm to find the Positions[1], Positions[2], ..., Positions[k] smallest elements of A. For instance, if $A = (16, 3, 19, 12, 16, 21, 18, 10)$ and Positions $= (3, 5, 8)$, then the solution is the array $(12, 16, 21)$ because 12 is the third smallest element of A, 16 is the fifth smallest element of A, and 21 is the eigth smallest element of A.

Suppose now that instead of receiving the positions all at once, you get a sequence of k queries of the form "what is the i^{th} smallest element of A", and that you must respond to each query when it is received.

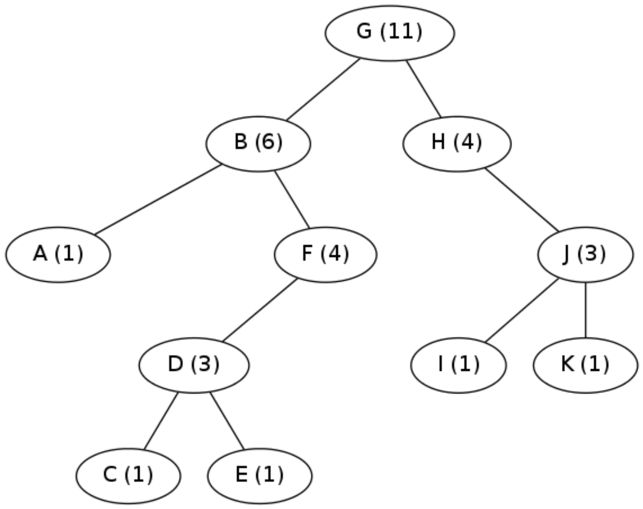
1. Asymptotically, for which values of k (the number of queries) is the best solution to sort the array A and then answer each query in $\Theta(1)$ time?

   When $k \in$ $\boxed{}$

2. Now, instead of only queries for the i^{th} smallest element of A, you receive a sequence of requests where each request is one of

   - insert a new element x into A
   - delete an element x from A
   - query for the i^{th} smallest element of A

   Keeping the elements in a sorted array A is no longer an option. One alternative is to store the elements in a balanced binary search tree T. We will keep in each node the size of the subtree rooted at that node (the sizes are in parentheses in the following example). This size information can be maintained when insertions and deletions are performed on the tree without affecting the running times of these operations.



Write an algorithm that takes a binary search tree T (with the size information) and an integer i, and returns the i^{th} smallest element of the tree T.

3. Finally, write an algorithm that takes a binary search tree T (with the size information) and a key x, and returns the rank of x in T. The smallest element of T has rank 1, the second smallest element of T has rank 2, etc. You may assume that x is present in the tree.

4 Mastering recurrences

Consider case 3 of the Master theorem: $f(n) \in \Omega(n^{\log_b a+\varepsilon})$ for some $\varepsilon > 0$, and $af(n/b) < \delta f(n)$ for some $\delta < 1$ and all sufficiently large values of n. If the regularity condition holds, then we can prove by induction that $a^j f(n/b^j) \leq \delta^j f(n)$. **Use** (i.e., assume) this fact to prove that, if the conditions for case 3 of the Master theorem hold, then $T(n) \in \Theta(f(n))$.

5 WestGrid (and North/East/SouthGrid)

As transistor densities continue to increase but processor speed and the complexity (in transistors) of processors does not, chip manufacturers turn more and more to on-chip multi-core solutions to provide additional performance. The 2-D nature of VLSI chips thus makes communication on a 2-D grid important.

In this problem, we imagine a $n \times n$ grid of processors, also called nodes. Each node is described by its (x, y) coordinate pair, where the upper-leftmost node is $(1, 1)$ and the lower-rightmost node is (n, n), and by a single positive integer $grid[x, y]$ describing its congestion level (how busy it is).

The quiz solution included algorithms for finding the maximum congestion of any node at or northwest of a specific node and the maximum congestion overall of any node that does not share the same x or y coordinates with a given node:

Congestion at or to the northwest of a node:

$$NWC(x,y) = \begin{cases} 0 & \text{if } x < 1 \text{ or } y < 1 \\ \max(NWC(x-1,y), NWC(x,y-1), grid[x,y]) & \text{otherwise} \end{cases}$$

Maximum congestion outside of a nodes row/column: `MaxC(x, y)`:

```
return max(NWC(x - 1, y - 1), NEC(x + 1, y - 1), SEC(x + 1, y + 1), SWC(x - 1, y + 1))
```

Now, solve these problems:

1. Give a complete (not just northwest!), dynamic programming approach to efficiently prepare to answer repeated calls to `MaxC`. You may set aside memory to be used and reused on calls to `MaxC` up to at most big O of the amount of memory the initial input grid uses, but you should make clear what memory you're using and what values in that memory mean.

   Specify a pre-processing function `PrepareMaxC` to be called once before all calls to `MaxC`. `PrepareMaxC` can assume access to *grid*. The query function `MaxC` can also assume access to *grid* and to any data structures you indicate are shared with `PrepareMaxC`. Make clear how `PrepareMaxC` should be called.

   Your priority is to make `MaxC` as efficient in terms of asymptotic runtime as possible. Within that constraint, you should make `PrepareMaxC` as efficient in terms of asymptotic runtime as possible and ensure both use no more than the space restrictions laid out above.

2. Give and briefly justify the runtime and memory usage of `PrepareMaxC` and `MaxC` in terms of n, the grid dimension.