CPSC 320 2017W2: Tutorial quiz 4

1 The elements go up and down

Let A be an integer array with n elements in total, consisting of two sections: first one with numbers strictly increasing followed by one with numbers strictly decreasing. For instance, the array

$$(3, 8, 14, 17, 26, 27, 31, 35, 28, 22, 6, 1)$$

satisfies this property. We want to find the smallest and largest elements of A efficiently.

1. Suppose that you know the values of $A[j]$ and $A[2j]$, where $j = \lfloor n/3 \rfloor$. Suppose moreover that $A[j] \leq A[2j]$. Where could the **largest** element of A be? Fill in the box next to **ALL** that apply:

   ☐ in $A[0] \ldots A[j]$ ☐ in $A[j] \ldots A[2j]$ ☐ in $A[2j] \ldots A[n-1]$

2. **This problem is for the group quiz only (ungraded on the individual quiz).** Design an efficient divide-and-conquer algorithm to find the **largest** value in A based on this idea of investigating $A[j]$ and $A[2j]$.

3. Write down a recurrence relation that describes the worst case running time $T(n)$ of an efficient divide-and-conquer algorithm based on the idea of investigating $A[j]$ and $A[2j]$, where n is the length of the array or subarray under consideration. You may ignore floors and ceilings.

$$T(n) = \begin{cases} \boxed{} & \text{if } n \leq 3 \\ \boxed{} & \text{otherwise} \end{cases}$$

Give a good bound on the worst-case runtime of this algorithm: $T(n) \in O(\boxed{})$

4. Suppose that you know the values of $A[j]$ and $A[2j]$, where $j = \lfloor n/3 \rfloor$. Suppose moreover that $A[j] \leq A[2j]$. In how many positions of the array could the **smallest** element of A be? Fill in the circle next to the **best** answer.

○ only one ○ exactly two ○ exactly three ○ four or more

Suppose instead that $A[j] \geq A[2j]$. Where could the **largest** element of A be? Fill in the box next to **ALL** that apply:

☐ in $A[0] \dots A[j]$ ☐ in $A[j] \dots A[2j]$ ☐ in $A[2j] \dots A[n-1]$

5. Let us now consider the case where A is an integer array with n elements in total, consisting of **three** sections: first one with numbers strictly increasing, followed by one with numbers strictly decreasing, followed by another section with numbers strictly increasing. For instance, the array

$$(3, 8, 14, 17, 26, 27, 31, 35, 28, 22, 6, 1, 4, 7, 8, 13, 21, 34)$$

satisfies this property. Once again we want to find the smallest and largest elements of A efficiently.

Suppose that you know the values of $A[j]$, $A[2j]$ and $A[3j]$ where $j = \lfloor n/4 \rfloor$. Suppose moreover that $A[j] \leq A[2j] \leq A[3j]$. Where could the **largest** element of A be? Fill in the box next to **ALL** that apply:

☐ in $A[0] \dots A[j]$ ☐ in $A[j] \dots A[2j]$ ☐ in $A[2j] \dots A[3j]$ ☐ in $A[3j] \dots A[n-1]$

2 Essay, Essay Again

2.1 Essay, Essay

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 positive valuation (number) for each essay, representing how much they would like to write 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, and a valuation of 9 is as much better than 6 as 6 is better than 3. However, **essays** have no valuation or preference over writers.

You want to find the valid assignment of essays to writers (a perfect matching) of highest quality. **For our purposes**, we define such an optimal assignment to be any perfect matching of writers and essays in which no two writers would both (strictly) prefer to switch essays with each other than to complete the essays assigned to them.

Start the group stage with the following two (ungraded) tasks:

1. Write out and draw trivial and small instances of the problem and their solutions.
2. Design a greedy algorithm to solve the problem.

Each of the following presents an algorithm for solving this problem. For each one fill in the circle next to the **best** answer among the following:

1. The algorithm is **OPTIMAL**, i.e., produces a valid and optimal solution for *every* valid instance of this problem.
2. The algorithm is **CORRECT** (but not optimal), i.e., produces a valid solution for *every* valid instance of this problem but sometimes produces a suboptimal one.
3. The algorithm is **INCORRECT**, i.e., does not produce a valid solution for at least one valid instance of this problem.

Here are the algorithms:

1. **Algorithm 1**: Repeatedly pick the remaining (i.e., unassigned) essay with the highest valuation from any one remaining writer. Assign that essay to that writer. Repeat until no essays remain. (Break ties arbitrarily.)

   This algorithm is:

   ○ **OPTIMAL** ○ **CORRECT** ○ **INCORRECT**

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

   This algorithm is:

   ◯ **OPTIMAL** ◯ **CORRECT** ◯ **INCORRECT**

3. **Algorithm 3**: Repeatedly pick an arbitrary remaining writer. If there is only one essay remaining on that writer's list of valuations, assign the essay to them. Otherwise, eliminate from their list of valuations the essay with lowest value. Repeat until all essays are assigned.

   This algorithm is:

   ◯ **OPTIMAL** ◯ **CORRECT** ◯ **INCORRECT**

4. Rather than giving **algorithm 4**, we only give a description of the type of solution it produces. Algorithm 4 produces a matching that maximizes the total for each writer w of w's valuation of the essay assigned to w.

   This time, you should fill in the "highest" blank the algorithm is **guaranteed** to achieve. That is, if all such algorithms are optimal, fill in **OPTIMAL**. If all such algorithms are correct but not all are optimal, fill in **CORRECT**. Otherwise, fill in **INCORRECT**.

   The best answer for such algorithms is:

   ◯ **OPTIMAL** ◯ **CORRECT** ◯ **INCORRECT**

2.2 Essay, Essay 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 positive valuation (number) for each essay, representing how much they would like to write 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, and a valuation of 9 is as much better than 6 as 6 is better than 3. However, **essays** have no valuation or preference over writers.

You want to find the valid assignment of essays to writers (a perfect matching) of highest quality. **For our purposes**, we define such an optimal assignment to be a perfect matching of writers and essays that maximizes the total for each writer w of w's valuation for the essay assigned to w.

Start the group stage with the following two (ungraded) tasks:

1. Write out and draw trivial and small instances of the problem and their solutions.
2. Design and critique a greedy algorithm for the problem.

Each of the following presents an algorithm for solving this problem. For each one fill in the circle next to the **best** answer among the following:

1. The algorithm is **OPTIMAL**, i.e., produces a valid and optimal solution for *every* valid instance of this problem.
2. The algorithm is **CORRECT** (but not optimal), i.e., produces a valid solution for *every* valid instance of this problem but sometimes produces a suboptimal one.
3. The algorithm is **INCORRECT**, i.e., does not produce a valid solution for at least one valid instance of this problem.

Here are the algorithms:

1. **Algorithm 1**: Repeatedly pick the remaining (i.e., unassigned) essay with the highest valuation from any one remaining writer. Assign that essay to that writer. Repeat until no essays remain. (Break ties arbitrarily.)

   Fill in the blank next to the **best** answer. This algorithm is:

   ◯ **OPTIMAL** ◯ **CORRECT** ◯ **INCORRECT**

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

   Fill in the blank next to the **best** answer. This algorithm is:

   ○ **OPTIMAL** ○ **CORRECT** ○ **INCORRECT**

3. **Algorithm 3**: Repeatedly pick an arbitrary remaining writer. If there is only one essay remaining on that writer's list of valuations, assign the essay to them. Otherwise, eliminate from their list of valuations the essay with lowest value. Repeat until all essays are assigned.

   Fill in the blank next to the **best** answer. This algorithm is:

   ○ **OPTIMAL** ○ **CORRECT** ○ **INCORRECT**

4. Rather than giving **algorithm 4**, we only give a description of the type of solution it produces. Algorithm 4 produces a perfect matching in which no two writers would both (strictly) prefer to swap essays with each other over completing the essays they were assigned themselves.

   This time, you should fill in the "highest" blank the algorithm is **guaranteed** to achieve. That is, if all such algorithms are optimal, fill in **OPTIMAL**. If all such algorithms are correct but not all are optimal, fill in **CORRECT**. Otherwise, fill in **INCORRECT**.

   The best answer for such algorithms is:

   ○ **OPTIMAL** ○ **CORRECT** ○ **INCORRECT**

3 Marvelous Medians

Suppose that we are given an unsorted array A with n distinct elements, and another ***sorted*** array Positions with k distinct elements chosen from the set $\{1, 2, \ldots, n\}$.

In this question, we consider the problem of finding the Positions[1], Positions[2], ..., Positions[k] smallest elements of A. For instance, if $A = (15, 3, 19, 12, 16, 21, 18, 10)$ 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.

1. Describe a divide-and-conquer algorithm to compute the solution in $O(n \log k)$ average-case time.[1] You may assume that you can compute the sub-array $X[p \ldots r]$ of an array X in constant time if it makes your algorithm easier to understand. **Group part only; ungraded on individual.**

[1]You may ignore this, but: the average here is over a uniform distribution on the permutations of A and the possible subsets of size k of $\{1, \ldots, n\}$ for `Positions`.

2. Suppose instead that we called algorithm `QuickSelect` k times (once for each position). What would be the running time then? Fill in the circle next to the **best** answer.

   ○ $\log(kn)$

   ○ $k \log n$

   ○ $n \log k$

   ○ nk

FOR SUBSEQUENT PARTS: Here is a *rough outline* of an algorithm to solve this problem:

(a) If |`Positions`| = 1, use plain QuickSelect to solve the problem.

(b) Use QuickSelect to find the `Positions`$[\frac{k}{2}]^{th}$ smallest element. It is our new pivot.

(c) Partition A into `ALesser` and `AGreater` based on the pivot.

(d) Partition `Positions` into left and right halves (adjusting the values in the right half appropriately, given that we've discarded `ALesser` and the pivot in the recursion).

(e) Recursively solve the subproblems (the lesser and greater sides of `A` and `Positions` generated in the previous two steps).

(f) Concatenate the left recursive solution with the pivot with the right recursive solution.

3. Complete the recurrence relation below that describes the expected running time of this algorithm, assuming that the first pivot element you find is at rank p (i.e., the p^{th} smallest element). Assume that algorithm `QuickSelect` runs in expected $\Theta(n)$ time where n is the size of the array it receives as input.

$$T(n,k) = \begin{cases} T\left(\square, \square\right) + T\left(\square, \square\right) + \square & \text{if } k \geq \square \\ \square & \text{if } k \leq \square \end{cases}$$

4. Complete the following explanation of why this algorithm runs in expected $O(n \log k)$ time, assuming that algorithm `QuickSelect` runs in expected $O(n)$ time where n is the size of the array it receives as input. (The explanation imagines a standard recursion tree drawn for T above.)

   All leaves in the **entire** recursion tree together take $O(\square)$ time. There are (about) $\square$ levels of recursion before we reach the base case. Each level of the tree takes $O(\square)$ time total.

4 Mastering recurrences

In this tutorial you will consider recurrences of the following form, that frequently arise from divide and conquer algorithms:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{if } n \geq n_0 \\ \Theta(1) & \text{if } n < n_0 \end{cases}$$

where $a \geq 1$ is an integer, $b > 1$ is a positive real number, and $f(n)$ is a function from $\mathbf{N}$ into $\mathbf{R}^+$. We will moreover assume that $n = b^t$ for some positive integer t.

1. **This problem is for the group quiz only (ungraded on the individual quiz).** Prove that $a^t = n^{\log_b a}$.

2. **This problem is for the group quiz only (ungraded on the individual quiz).** Draw the first three levels and the last level of the recursion tree for this recurrence.

3. **This problem is for the group quiz only (ungraded on the individual quiz).** Using your tree from part 2, write an equation for $T(n)$. Separate the work done on the last level of the tree from the work done on the rest of the levels; the second term will be a summation.

4. Fill in the blanks in the following paragraph: a node on level j of the recursion tree for this recurrence (the root is at level 0) handles [ ] elements, and does [ ] work. There are [ ] nodes on level j, and the total amount of work done on level j is [ ].

5. Now consider a recurrence like

$$T(n) = \begin{cases} T(k_1 n) + T(k_2 n) + n^2 & \text{if } n \geq n_0 \\ 1 & \text{if } n < n_0 \end{cases}$$

We know that $0 < k_1 \leq k_2 < 1$. What single additional inequality do we need to know about k_1 and k_2 to conclude that $T(n) \in O(n^2)$? *Hint:* draw at least two levels of the recursion tree and think about the work per level and the cases of the Master Theorem.

The needed inequality is: [ ] $<$ [ ]

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).

1. For the first part of this problem, for a given node (provided by (x, y) coordinates) we want to know four quantities: the maximum congestion value of all nodes strictly to its North-West (lower x and lower y coordinates), the maximum congestion value for nodes strictly to its North-East (higher x, lower y), the maximum congestion for nodes strictly to its South-East, and the maximum for nodes strictly to its South-West.

   Complete the following recursive formulation of the North-West congestion (`NWC`) quantity. (Assume that *grid* is "global" to the recursion.)

$$NWC(x, y) = \begin{cases} 0 & \text{if } \boxed{\qquad\qquad} \\ \max(NWC(\boxed{\quad}, \boxed{\quad}), & \\ \quad NWC(\boxed{\quad}, \boxed{\quad}), & \text{otherwise} \\ \quad grid[\boxed{\quad}, \boxed{\quad}]) & \end{cases}$$

2. The overall goal is to find the maximum congestion value outside of the current cell's row and column. Assuming that functions for the four directions `NWC`, `NEC`, `SEC`, and `SWC` have all been correctly implemented (along the lines of `NWC` for the northwest direction above), use them to complete the following function to find the maximum congestion outside a cell's row and column, called `MaxC`:

```
MaxC(x, y):

    return ______________________________________________________________
```

3. Which of these best describes the runtime of a naive, recursive implementation of `NWC` run with the paramaters (n, n), i.e., `NWC(n,n)`? Fill in the circle next to the **best** answer.

   ◯ $O(1)$ ◯ $O(n)$ ◯ $O(n^2)$ ◯ $O(n^3)$ ◯ none of these

4. Now, imagine that given coordinates for a node (x, y), we want the maximum congestion over every node **except** the given node (including those due north, east, south, or west). However, we want to be able to compute this quantity very rapidly.

   Complete the following $O(n^2)$ time, $O(1)$ space pre-processing algorithm that sets up for a $O(1)$ time algorithm to respond to these queries:

```
Let preprocessData be a variable shared across preCong and queryCong

// Preprocesses the given nxn grid of processors for queryCong, storing
// results in preprocessData.
//
// Here and for queryCong, grid[x, y] is equal to the congestion of node (x, y)
// and can be found in constant time.
//
// Must be called on an nxn grid of processors once BEFORE calling queryCong
// on that same grid of processors, but any number of calls to queryCong may
// occur after the single call to preCong.
preCong(grid):
    Iterate over each node in the grid to find the


       ___________________________________ congestion value(s)


       and its (or their) coordinates.


    Store the value(s) and (x, y) coordinates in preprocessData.

// Given an nxn grid of processors on which preCong has already been called,
// and the (x, y) coordinates of a processor in that grid, returns the
// maximum of all congestion values BESIDES the one at grid[x, y].
queryCong(grid, x, y):

    If preprocessData contains the coordinates (x, y), then:


        return __________________________________________________

    Otherwise:



        return __________________________________________________
```