Midterm 2 Sample Solution

1 Writing a recurrence [8 marks]

Consider the following (strange) algorithm:

```
define vcan(A, first, n):             // our recurrence should be a function of n only
  sum = 0                             // O(1)
  if n > 0:                           // O(1); if n<=0, we're done: base case!
    sum = vcan(A, first + n//2, n//2) - // one recursive call T(n//2) plus O(1) work
          vcan(A, first, n//2)        // another recursive call T(n//2) plus O(1) work
    if n > 20:                        // another split in cases; see notes below
      sum = sum * vcan(A,first+10,n-20) // another recursive call T(n-20) plus O(1) work
  return sum                          // O(1)
```

Now, fill in the recurrence relation below that describes the worst-case running time of `vcan` as a function of `n`.

Notes:

- Write any base case(s) before any recursive case(s).
- You may ignore floors and ceilings in your recurrence.
- In the code above, $x//y$ means $\lfloor \frac{x}{y} \rfloor$.

Solution: See the comments above, this paragraph, plus the recurrence below. Looking at the comments above, we have three cases: when $n \leq 0$, we have a base case with $\Theta(1)$ work; when $0 < n \leq 20$, we have a recursive case with two recursive calls plus $\Theta(1)$ work; when $n > 20$, we have an **additional** third recursive call and $\Theta(1)$ work. (Side notes: First, remember that it doesn't matter whether we're adding, subtracting, multiplying, or doing something else to the result of a recursive call. We still **make** the recursive call and then take other operations. So, when assessing runtime, we add in the cost of that recursive call. Next, if it was **entirely** correct, we allowed an alternate formulation with just two cases overall based on recognizing that the "middle" case must take constant time for any value of n that would fall in the case. However, since this is a poorer fit to the expression of the code, incorrect answers are harder to judge for partial credit! Also, we include floors below, but you didn't have to.)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 0 \\ 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(1) & \text{if } 0 < n \leq 20 \\ 2T(\lfloor \frac{n}{2} \rfloor) + T(n-20) + \Theta(1) & \text{otherwise} \end{cases}$$

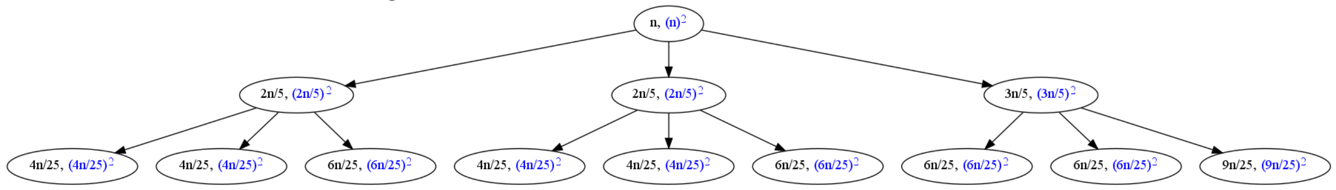
2 Solving a recurrence [6 marks]

An algorithm has a worst-case running time described by this recurrence:

$$T(n) = \begin{cases} 2T(2n/5) + T(3n/5) + n^2 & \text{if } n \geq 5 \\ 1 & \text{if } n \leq 4 \end{cases}$$

where n is the number of items passed as input to the algorithm. You'll answer the questions below based on $T(n)$. To do so, it may help to draw the first few levels of the recursion tree for $T(n)$ here:

SOLUTION: Here is our diagram of levels 0, 1, and 2 with problem size in black and work in blue.



We don't have the work per level, but that's the sum of the blue in a particular row. With this, we can mostly read off the pieces below. The big exception is a formula for the work per level. That we can get by observing that the work at level 1 is $\frac{4}{25} + \frac{4}{25} + \frac{9}{25} = \frac{4+4+9}{25} = \frac{17}{25}$ of the work at level 0. That same fraction applies to the work in the children of any node, which leads to our answer below about overall work per level.

In the recursion tree, we consider the root node to be at level 0. That node represents an invocation of the algorithm on n elements. (Feel free to leave multiplication, division, and exponentiation in your answers below.)

1. Fill in this box with the **smallest number of elements passed to** any invocation of the algorithm represented by one of **the nodes on level 2**: $\frac{4n}{25}$

   Fill in this box with the amount of work done by that invocation of the algorithm (not including work done in its children, as usual): $(\frac{4n}{25})^2$

2. Fill in this box with the **largest number of elements passed to** any invocation of the algorithm represented by one of **the nodes on level 2**: $\frac{9n}{25}$

3. Fill in this box with the total amount of work done at level j of the recursion tree:

   $$\boxed{(17/25)^j n^2}$$

4. Fill in the box with a tight asymptotic upper-bound on the value of $T(n)$.

   $$T(n) \in \boxed{\mathrm{O}(\mathrm{n}^2)}$$

3 QuickTrue or QuickFalse [6 marks]

Fill in the circle next to the **best** answer of "True" or "False" for each statement below.

1. ○ **True** ○ **False** For a version of QuickSelect that chooses the first element as pivot, it is possible to select inputs (A and k) over all possible array sizes n (a "best case") that guarantee QuickSelect runs in $o(n)$ time (i.e., little-o of n or **faster** than linear time).

   SOLUTION: Regardless of how we pick the pivot, QuickSelect performs a partition step even if it returns on its first recursive call. So, it takes $\Omega(n)$ time to run, and this is **False**.

2. ○ **True** ○ **False** For a version of QuickSelect that chooses its pivot randomly, it is possible to select inputs (A and k) over all possible array sizes n (a "best case") that guarantee QuickSelect runs in $o(n)$ time (i.e., little-o of n or **faster** than linear time).

   SOLUTION: As above, this is **False**.

3. ○ **True** ○ **False** For a version of QuickSelect that chooses the first element as pivot, it is possible to select inputs (A and k) over all possible array sizes n (a "worst case") that guarantee QuickSelect runs in $\omega(n \lg n)$ time (i.e., little-Ω of $n \lg n$ or **slower** than $n \lg n$ time).

   SOLUTION: True. For instance, if we put the elements in increasing sorted order but ask for the largest element, this version of QuickSelect runs in $\Omega(n^2) \subseteq \omega(n \lg n)$ time.

4. ○ **True** ○ **False** For a version of QuickSelect that chooses its pivot randomly, it is possible to select inputs (A and k) over all possible array sizes n (a "worst case") that guarantee QuickSelect runs in $\omega(n \lg n)$ time (i.e., little-Ω of $n \lg n$ or **slower** than $n \lg n$ time).

   SOLUTION: False. No matter what inputs we select it's the randomizer that selects the pivot. Thus, the overall runtime could be anywhere between linear (as happens in both the expected case and the case where we "get lucky" and pick the target element as pivot) and quadratic (as happens in the vanishingly improbable case where we repeatedly select worst-case pivots).

5. For this sentence and the next one, recall the "Essay, Essay" problem from Quiz #4. Briefly: n writers give positive integer valuations to n essays and we try to produce a perfect matching of writers to essays to give "good" valuations.

   ○ **True** ○ **False** If in the perfect matching no two writers would both (strictly) prefer to swap with each other, then it is also the case that no subset of the writers of any size would all (strictly) prefer to rearrange their assignments with each other.

   SOLUTION: False. Consider the case where w_1 gives values 3, 2, 1 for essays 1, 2, and 3; w_2 gives 1, 3, 2, and w_3 gives 2, 1, 3. We assign everyone their second-favorite essay (w_1 gets e_2, w_2 gets e_3, and w_3 gets e_1). They'd rather rearrange to get their top favorite, but no pair would prefer to swap with each other (because one person always "gets worse").

6. ○ **True** ○ **False** If in the perfect matching we maximize the total of all writers' valuations of their assigned essays and some writer has the single highest overall valuation of any essay, then that writer's valuation of their assigned essay will be higher than any other writer's valuation of that other writer's assigned essay.

   SOLUTION: False. Consider the case where w_1 gives values 5, 3 and w_2 gives values 4, 1. The best solution gives e_1 to w_2 and e_2 to w_1, with a total value of 7. The alternate solution where w_1 gets a higher valuation than anyone else gives e_1 to w_1 and e_2 to w_2 for a total value of 6.

4 Majorly Majority [14 marks]

We say that an array A of $n > 0$ numbers contains a *majority* when more than $n/2$ of its elements all have the same value. We then call this element the *majority element* of A. In this section, you design algorithms that **find and return a majority element if one exists** and **otherwise return an arbitrary element**.

For example, in $[2, 7, 3, 3, 1, 2, 3, 3]$ no element occurs **more** than $8/2 = 4$ times, and so **any of** 2, 7, 3, or 1 is a correct solution. However, in $[2, 3, 7, 3, 3, 1, 2, 3, 3]$, 3 is the majority element and the only correct solution because it occurs 5 times, which is more than $9/2$.

1. Fill in the circle next to the **most specific, accurate completion** of the following sentence. If we arbitrarily distribute (partition) the elements of an array A that contains a majority into k subarrays, then the majority element of A is also the majority element of... **[1 mark]**

   ○ All k subarrays
   ○ A majority of the k subarrays (more than $k/2$)
   ○ At least two subarrays
   [✓ ● At least one subarray]
   ○ None of the above

   SOLUTION: This answer is related to the political practice of gerrymandering. For an extreme example, imagine we have at least $k - 1$ elements not equal to the majority element. We divide into k subarrays, putting all of the majority element in one subarray and one each of the other elements in the remaining subarrays. For a more gerrymandering-like solution, we imagine a slim majority and create k (roughly) equal-sized subarrays. We pack all majority elements into one subarray and then give (possibly-slim) majorities to the non-majority elements in all other subarrays. Politics! What fun :)

2. Complete the following to design a sensible divide-and-conquer algorithm that computes a majority element (if one exists) by first dividing the array in half and then performing other work. Your algorithm's worst case runtime must match a recurrence with a constant base case for sufficiently small problems and the recursive case $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, ignoring floors and ceilings. **[6 marks]**

   SOLUTION: In place below. The central idea is that the majority element must be the majority element in at least one subarray. So, we find the majority elements in the subarrays and simply count their occurrences. Whichever is most frequent is the majority element (if there is one). The sensible base case is for an array of length 1, but just for fun, we made it for an array of length 2. (If you return either element from an array of length 2, it's either the majority element or there is no majority element because the two elements are different.)

```
Majority(A):

    if length(A) <= __2__:

        return ___A[0]____

    else:

        let Left = the first floor(n/2) elements of A
        let Right = the last ceiling(n/2) elements of A

        // Fill in the remainder of your algorithm here:
        let LM = Majority(Left)
let LCount = 0
for each element x of A:  // or just say "count # of occurrences of LM"
    if x = LM: increment LCount
if LCount > length(A) / 2:
    return LM
else:
    return Majority(Right)

// Or: just make both recursive calls, count frequency
// of both results in A, and return the more frequent.
```

3. What is the worst case running time of your algorithm? (Note that you know a recurrence relation for the algorithm even without filling in the blanks above.) **[1 mark]**

   Worst-case runtime is $\Theta(\boxed{\text{n } \lg n})$

   SOLUTION: The recurrence exactly matches MergeSort's recurrence.

4. Recall the "Playing the Blame Game" problem (hereafter called "BLAME"), repeated here (with some irrelevant detail removed) from Assignment #3:

   > A distributed computing system composed of n nodes is responsible for ensuring its own integrity against attempts to subvert the network. To accomplish this, nodes can assess each others' integrity, which they always do in pairs. A node in such a pair with its integrity intact will correctly assess the node it is paired with to report either "intact" or "subverted". However, a node that has been subverted may freely report "intact" or "subverted" regardless of the other node's status.

   We consider a complete, correct solution to the BLAME problem to be: the set of **ALL intact nodes** (not just one) if the majority were intact initially and otherwise any arbitrary, non-empty set of nodes. **First**, complete the following reduction from the majority problem to BLAME. Your reduction's algorithm for each node to report "intact" or "subverted" must require $O(1)$ time. **[6 marks]**

   > **Converting an instance of majority to BLAME:** Given a list A of n numbers, create n nodes numbered $0, 1, 2, \ldots, n-1$ in BLAME. When node i is asked to report on node j, it should report "intact" when $\boxed{\text{A[i] = A[j]}}$, and otherwise report "subverted".
   >
   > **Converting a solution to the BLAME instance to a solution to the majority instance:** Given the (non-empty) solution set of nodes $\{x, \ldots\}$, produce $\boxed{\text{A[x]}}$.

   Next, clearly and concisely complete the following proof in two cases of your reduction's correctness:

   Case 1: A did not contain a majority. By assumption, BLAME always produces some non-empty set of nodes. Thus, the reduction correctly produces some element of A.

   Case 2: A did contain a majority. First, we justify considering that each $\boxed{\text{majority element}}$ corresponds to an intact node in BLAME. Our intact nodes correctly report "intact" for other intact nodes because their values $\boxed{\text{are equal}}$; they report "subverted" for subverted nodes because the subverted nodes' values $\boxed{\text{are unequal to the majority element}}$. Of course, subverted nodes report correctly because $\boxed{\text{they may freely report either intact or subverted}}$. Thus, our definition of intact nodes is correct in BLAME.

   Since $\boxed{\text{most nodes are intact}}$, BLAME must produce a complete set of intact nodes. Therefore, the reduction correctly produces a majority element.

   In either case, the reduction is correct. QED

5 WestGrid (and North/East/SouthGrid) [6 marks]

Recall the WestGrid problem repeated below (with some irrelevant detail removed) from Quiz #4:

> In this problem, we imagine a $n \times n$ grid of 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).

We can draw a WestGrid instance as a table of numbers, like:

1	8	15	9
2	4	22	14
25	7	19	6
25	12	3	31

Unlike in the quiz, we want to find the longest non-decreasing simple path ("non-decreasing" meaning staying the same or increasing and "simple" meaning without cycles) that starts in the upper-left corner and moves by single steps in one of the four directions N, E, S, or W. So, in this example, that's $1, 2, 4, 7, 12, 25, 25$, which is one number longer than the next best path $1, 2, 4, 8, 15, 22$.

1. Consider the following greedy algorithm for this problem:

   > Start with node $(1, 1)$ as the current node, mark it as visited, and initialize the set of neighbours to contain its neighbours that have at least as high congestion as it does (if any). Then repeat until the set of neighbours is empty: (1) Assign as the current node: the node in the set of neighbours with lowest congestion (breaking ties arbitrarily). (2) Mark the current node as visited. (3) Assign as the set of neighbors: the current node's *unvisited* neighbours with congestion at least as high as the current node.

   Now, **fill in the remainder of the table below** to create a counterexample to this algorithm's optimality. Your instance **must** be composed of distinct integers (i.e., have no duplicate congestions). **[4 marks]**

   SOLUTION: There are many correct solutions, but perhaps the simplest is to put values smaller than 5 next to 5 so it "dead-ends" but let 7 go at least one more step.

1	**5**	*2*
7	*3*	*6*
8	*4*	*9*

   Give the **optimal** solution to your instance: 1, 7, 8

   Give the **algorithm's** solution to your instance: 1, 5

2. A friend proposes that we could remove the part of the algorithm in the previous problem that marks nodes as visited, and the algorithm would still be **correct** (produce valid solutions), if **not necessarily optimal**. Fill in the **best** circle in each column below. **[2 marks]**

   SOLUTION: Note that we can think of an instance without duplicates as a DAG with edges from values to higher neighbouring values. Our algorithm finds a path in that DAG and so clearly terminates with a valid (but not necessarily optimal) solution. However, duplicate values introduce cycles in that graph, and our algorithm may end up stuck in such a cycle, never producing a result, which is what would happen in the example given above.

The new algorithm is correct for [✓ all] some no instances with no duplicate congestion values and correct for all [✓ some] no instances **with** duplicate congestion values.

6 BONUS: From the Cutting Room Floor [1 BONUS marks]

Bonus marks add to your exam and course bonus mark total but are **not** required. **WARNING**: These questions are too hard for their point values. We are free to mark these questions harshly. Finish the rest of the exam before attempting these questions. Do not **taunt** these questions.

Just one bonus problem this time:

1. Revisiting the **Majorly Majority** problem: We can also solve this problem in expected linear time by relying on `QuickSelect`. Give a **very concise**, **clear**, **reasonable** algorithm that solves this problem in expected $O(n)$ time. You **must** briefly and clearly justify both the correctness and runtime of your algorithm (but you should assume `QuickSelect` has its usual properties).

   SOLUTION: It's a bonus; so, we leave it to you with one hint: it's a single, simple line of code.