COSC 302, Practice Exam #1
February 20, 2018

Honor Code

I agree to comply with the spirit and the rule of the Colgate University Academic Honor Code during this exam. I will not discuss the contents of this exam with other students until all students have finished the exam. I further affirm that I have neither given nor received inappropriate aid on this exam.

Signature: ______________________________

Printed Name: ______________________________

Write and sign your name to accept the honor pledge. Do not open the exam until instructed to do so.

You have 75 minutes to complete this exam.

There are 5 questions and a total of 65 points available for this exam. Don't spend too much time on any one question.

If you want partial credit, show as much of your work and thought process as possible.

Question	Points	Score
1	8	
2	12	
3	15	
4	15	
5	15	
Total:	65	

1. (8 points) Answer the following True/False questions by clearly circling your answer. No justification is required.

   (a) **True or False**: $\sin n = O(1)$

   Solution: False; while $\sin n$ bounded above by 1, is not asymptotically positive, and therefore is not bounded below by 0 for any choice of n_0.

   (b) **True or False**: $2^n = O(2^{n/2})$

   Solution: False; assume it's true, then $2^n \le c2^{n/2}$ for some positive constant c and all $n \ge n_0 > 0$. Solving for c, we get that $c \ge \frac{2^n}{2^{n/2}} = 2^{n/2}$, which grows as n increases and therefore cannot be a constant.

   (c) **True or False**: The recurrence $T(n) = T(\sqrt{n}) + 1$ solves to $T(n) = \Theta(\lg \lg n)$.

   Solution: True; solve with change of variables. Let $m = \lg m$, then $S(m) = S(m/2) + 1$, which solves to $\Theta(\lg m) = \Theta(\lg \lg n)$.

   (d) **True or False**: Let $Q(n)$ be the worst-case running time of QUICKSORT. Then $Q(n) = \Omega(n \lg n)$.

   Solution: True; even in the best case, quicksort runs no faster than $\Omega(n \lg n)$, therefore, the same applies to the worst case

   (e) **True or False**: Let $f(n)$ be an asymptotically positive function and further suppose that $f(n) = O(n)$. Then the solution to the recurrence $T(n) = T(n/2) + f(n)$ is $T(n) = \Theta(n)$.

   Solution: False; this one is subtle: $f(n)$ can be any function upper bounded by cn, therefore it could also be a constant, or logarithmic, or any number of other functions. We therefore don't have enough information to compute a *tight* asymptotic bound $T(n)$.

   (f) **True or False**:

$$\sum_{i=1}^{n} \sum_{k=0}^{\lfloor \lg n \rfloor} 2^k = \Theta(n^2)$$

   Solution: True; $\sum_{i=1}^{n} \sum_{k=0}^{\lfloor \lg n \rfloor} 2^k = \sum_{i=1}^{n} \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2-1} = \sum_{i=1}^{n} \Theta(n) = \Theta(n^2)$.

(g) **True or False**: The recurrence $T(n) = 4T(n/2) + n^3$ solves to $T(n) = \Theta(n^3)$.

Solution: True; by case 3 of the master theorem, $f(n) = n^3 = \Omega(n^{\log_2 4+\epsilon})$ for $\epsilon = 1 > 0$, and the regularity condition $af(n/b) = 4(\frac{n}{2})^3 \leq \frac{1}{2}n^3$ holds for $c = 1/2 < 1$.

(h) **True or False**: The recurrence $T(n) = 999T(n/998) + n \lg^{1000000000000} n$ solves to $T(n) = \Theta(n^{\log_{998} 999})$.

Solution: True; by case 1 of the master theorem, $f(n) = n \lg^{1000000000000} n = O(n^{\log_{998} 999-\epsilon})$ is true for any ϵ such that $0 < \epsilon < n^{\log_{998} 999}$. Note that $\log_{998} 999 > 1$, and therefore $n^{\log_{998} 999}$ grows polynomially faster than n^1 times any polylogarithmic factor.

2. For the following problems, you must show your work. Unjustified answers will receive 0 points.

   (a) (6 points) Let $f(n)$ and $g(n)$ be asymptotically positive functions, such that $f(n) = O(g(n))$. Formally prove that there exists some positive constant n_0, such that $0 \le f(n) \le n\,g(n)$ for all $n \ge n_0$.

Solution:

Proof. By the definition of Big-Oh, we already know that for some positive constants c, n_1 that

$$0 \le f(n) \le cg(n) \text{ for all } n \ge n_1.$$

Thus,

$$\begin{aligned} 0 &\le f(n) && \text{for } n \ge n_1 \\ &\le cg(n) && \text{for } n \ge n_1 \\ &\le ng(n) && \text{for } n \ge \max\{c, n_1\}. \end{aligned}$$

We therefore conclude that for $n \ge n_0 = \max\{c, n_1\}$ the claim holds. □

(b) (6 points) Consider the following pseudocode. Let $D(n)$ be the running time of Do-Something for a given input n. Give a recurrence for $D(n)$ and solve it using your favorite method. Your solution should be asymptotically tight. You may assume that $\sqrt[3]{n}$ is an integer.

Algorithm 1 An algorithm to do something.

```
proc Do-Something($n$)
1: if $n \leq 2$ then
2:     return 1
3: $k \leftarrow 1$
4: for $i \leftarrow 1$ to 27
5:     $k \leftarrow k$ + Do-Something($\sqrt[3]{n}$)
6: return $k$
```

Solution: This algorithm makes 27 recursive calls, each on a subproblem of size $\sqrt[3]{n}$, and then spends $\Theta(1)$ time in the current recursive call. Including the base case ($\Theta(1)$ when $n \leq 2$), we get the recurrence

$$D(n) = \begin{cases} \Theta(1) & n \leq 2 \\ 27D(\sqrt[3]{n}) + \Theta(1) & n > 2. \end{cases}$$

We solve this recurrence with change of variables *and* the master method. First, let $n = 2^m$, then we get a new recurrence

$$D(2^m) = 27D(2^{m/3}) + \Theta(1).$$

Renaming $D(2^m)$ to $S(m)$, we get that

$$S(m) = 27S(m/3) + \Theta(1).$$

which solves to $\Theta(m^{\log_3 27}) = \Theta(m^3)$ by case 1 of the master theorem,

$$\text{since } f(m) = c = O(m^{3-\epsilon}) \text{ for } \epsilon = 3 > 0.$$

Substituting back with $m = \lg n$, we get that $D(n) = \Theta(\lg^3 n)$.

3. (15 points) The algorithm below computes $n2^n$ when $\sqrt{n}$ is an integer. Give a loop invariant, and use it to prove that N-EXPONENTIAL correctly computes $n2^n$.

Algorithm 2 An algorithm to compute $n2^n$.

```
proc N-EXPONENTIAL($n$)
1: $k \leftarrow 2^{\sqrt{n}}\sqrt{n}$
2: for $i \leftarrow 1$ to $\sqrt{n} - 1$
3:     $k \leftarrow 2^{\sqrt{n}}k \cdot \frac{i+1}{i}$
4: return $k$
```

Solution:

Invariant: Before the i-th iteration of the **for** loop on line 2, we have that

$$k = i\sqrt{n}2^{i\sqrt{n}}.$$

To show correctness, we show that the invariant holds at initialization, during maintenance, and that at termination the algorithm computes the correct value $n2n^2$

Initialization: Before iteration $i = 1$ of the **for** loop, we have that

$$k = \sqrt{n}2^{\sqrt{n}} = (1)\sqrt{n}2^{(1)\sqrt{n}} = i\sqrt{n}2^{i\sqrt{n}}.$$

Maintenance: Suppose that before the i-th loop, we have that $k = i\sqrt{n}2^{i\sqrt{n}}$, we show that at the beginning of the $(i+1)$-th loop, $k = (i+1)\sqrt{n}2^{(i+1)\sqrt{n}}$.

In the i-th loop we compute $k = 2^{\sqrt{n}}k\frac{i+1}{i}$. Just before this computation, we have that $k = i\sqrt{n}2^{i\sqrt{n}}$, and therefore after computing, we get that

$$k = 2^{\sqrt{n}}\left(i\sqrt{n}2^{i\sqrt{n}}\right)\frac{i+1}{i} = (i+1)\sqrt{n}2^{(i+1)\sqrt{n}}.$$

Termination: Just before the last loop (where the body of the **for** loop does not execute) we have that $i = \sqrt{n}$. Thus we have that

$$k = i\sqrt{n}2^{i\sqrt{n}} = \sqrt{n}\sqrt{n}2^{\sqrt{n}\sqrt{n}} = n2^n,$$

which is then returned. Therefore, the algorithm correctly computes $n2^n$.

4. (15 points) Given an array $A[1..n]$, recall that the routine SqrtSort takes an integer k between 0 and $n - \sqrt{n}$ (inclusive) and sorts a subarray $A[k+1..k+\sqrt{n}]$. (To simplify the problem, assume that $\sqrt{n}$ is an integer.)

   (a) What is the maximum number of inversions that a call to SqrtSort(k) removes? Your answer should be exact and not asymptotic.

   Solution: In the worst-case the subarray of size $\sqrt{n}$ that is sorted by SqrtSort has the maximum number of inversions possible. That is all pairs of values are inverted. There are a maximum of $\binom{\sqrt{n}}{2} = \frac{\sqrt{n}(\sqrt{n}-1)}{2}$ such inversions. This can also be seen as follows:

   To avoid double counting inversions, count the later elements that each element is inverted with in the subarray of size $\sqrt{n}$. Then element one has $\sqrt{n}-1$ inversions with later elements, element 2 has $\sqrt{n}-2$ inversions with later elements, etc. This gives us a summation:

$$\sum_{i=0}^{\sqrt{n}-1} i = \frac{(\sqrt{n}-1)\sqrt{n}}{2}.$$

   (b) Now consider *any* algorithm that sorts by calling SqrtSort, and which is not allowed to directly compare, move, or copy array elements. Give a lower bound for the worst case, number of calls to SqrtSort than an algorithm must make to guarantee that $A[1..n]$ is sorted? Your answer should apply to *all* algorithms that solve the problem. Give your answer as an asymptotic lower bound. *Note:* To receive full credit, your lower bound should be as large as possible; a trivial lower bound such as $\Omega(1)$ will receive 0 points. Justify your answer.

   Solution: In the worst case, there can be $\frac{n(n-1)}{2}$ inversions in the array $A[1..n]$. Since SqrtSort removes at most $\frac{(\sqrt{n}-1)\sqrt{n}}{2}$ inversions, it must be called at least

$$\frac{\frac{n(n-1)}{2}}{\frac{(\sqrt{n}-1)\sqrt{n}}{2}} = \Omega(n) \text{ times.}$$

(c) An algorithm is said to be *worst-case optimal with running time* $\Theta(f(n))$ when its worst-case running time is $O(f(n))$ (an upper bound) and *any* algorithm *must* have running time $\Omega(f(n))$ to solve the problem in the worst case. Pick an algorithm that sorts A by making calls to SqrtSort (for example, your algorithm from homework 3). Could this algorithm be worst-case optimal? Justify your answer.

Solution:

#1: Darren presented a solution (in recitation) that sorts using $O(n)$ calls to SqrtSort. Since any algorithm must call SqrtSort $\Omega(n)$ times, the algorithm is worst-case optimal.

#2: My algorithm makes $O(n^2)$ calls to SqrtSort. However, it is possible that there exists a higher lower bound than $\Omega(n)$. Since there *could* be a higher lower bound, it is possible that my algorithm is worst-case optimal. However, it is impossible to say whether it actually is worst-case optimal at the moment, as we have not shown there is a lower bound of $\Omega(n^2)$.

5. (15 points) Suppose you are given an array $A[1..n]$ of distinct integers in $1..n$. We say that index p in A is *peak*, if $A[p] > A[p-1]$ and $A[p] > A[p+1]$. Similarly, we call an index v a *valley* if $A[v] < A[v-1]$ and $A[v] < A[v+1]$. Call A *bi-modal* if A has exactly two peaks and one valley.

   (a) Suppose A is bi-modal. Give pseudocode for a worst-case $O(n)$-time algorithm to find the peaks of A.

   Solution:

Algorithm 3 An algorithm to do something.

```
proc FindPeaks($A[1..n]$)
1:  $peak_1 = -1$
2:  $peak_2 = -1$
3:  $foundFirst$ = false
4:  for $i = 2$ to $n-1$ // Note that 1 and $n$ can't be peaks
5:      $isPeak = A[i] > A[i+1]$ and $A[i] > A[i-1]$
6:      if (not $foundFirst$) and $isPeak$ then
7:          $foundFirst$ = true
8:          $peak_1 = i$
9:      else if $foundFirst$ and $isPeak$ then
10:         $peak_2 = i$
11:         return ($peak_1$, $peak_2$)
```

   (b) Suppose you are told that the two peaks p_1 and p_2 of A are more than $n/2$ indices apart (e.g., $|p_2 - p_1| > n/2$). Describe a best-case $O(1)$ algorithm to find the peaks of A. What is the worst-case running time of your algorithm?

   Solution:
   First check to see if indices 2 and $n-1$ are peaks. If so, then we've found the peaks in $O(1)$ time (the best case!). If they aren't, perform the search as in (a) and find them in $\Theta(n)$ worst-case time.

   (c) Describe an algorithm that computes the two peaks of A in $O(\lg n)$ time if $n/2$ is the valley.

   Solution: We conceptually break $A[1..n]$ into two subproblems $A[1..n/2]$ and $A[n/2..n]$, then on each half we perform the search for the peak of a unimodal peak (as in recitation). We are guaranteed that the peaks are within $A[2..n/2-1]$ and $A[n/2+1..n-1]$ by the definition of a peak, therefore this meets the assumptions of that problem. Each such search takes $O(\lg n)$ time, and we perform 2 searches. Thus, the algorithm takes $O(\lg n)$ time overall.

(This page is intentionally blank. Label any work with the corresponding problem number.)

(This page is intentionally blank. Label any work with the corresponding problem number.)