CSCI 470 Fundamentals of Algorithms
Howard University
Instructor: Vijay Chaudhary — Practice Exam 1

Practice Exam 1

Date: September 23, 2023

DIRECTIONS:

- Write your answers on the exam paper.
- If you need extra space, please use the back of a page.
- You are allowed one cheat sheet.
- You have 80 minutes to complete the exam.
- Please do not turn the exam over until you are instructed to do so.
- Good Luck!

1	/25
2	/15
3	/25
4	/10
5	/25
Total	/100

1. (25 points, 5 each) For each of the following problems, answer **True** or **False** and BRIEFLY JUSTIFY your answer.

   (a) If $T(n) = n^2$, $T(n) = O(n \lg n)$.
   Solution: False.
   $T(n) = O(n \lg n)$ implies $T(n) \leq cn \lg n$ for some $c > 0$. However, $T(n) = n^2$, and, $n^2 > cn \lg n$, because $n > \lg n$ for all positive values of n. Therefore, $T(n) > cn \lg n$ for any value of $c > 0$.

   (b) Here is a pseudocode to calculate a factorial of n, where $n \in \mathbb{Z}^+$ (positive integers).

```
FACTORIAL(n)
x = 1
for i = 1 to n
    x = x * i
return x
```

   A loop invariant for this algorithm is:

   Before i-th iteration, $k = i - 1$, $x = 1 * 2 * \ldots * k = k!$, and $0! = 1$.

   Solution: True.
   In order to find a loop invariant, we look for a pattern with each iteration.
   Before $i = 1$ iteration, $x = 1$; $(x = (1 - 1)!)$
   Before $i = 2$ iteration, $x = 1 * 1$; $(x = (2 - 1)!)$
   Before $i = 3$ iteration, $x = 1 * 2$; $(x = (3 - 1)!)$
   Before $i = 4$ iteration, $x = 1 * 2 * 3 = 6$; $(x = (4 - 1)!)$
   Before $i = 5$ iteration, $x = 1 * 2 * 3 * 4 = 24$; $(x = (5 - 1)!)$
   Therefore, before i-th iteration, $(i - 1)!$ is a loop invariant.

   The following note is not something you are required to write in the exam. I am just trying to explain more.
   Note that we look for a loop invariant, that stays the same **before each subsequent iteration**. This essentially helps us with the termination step, and base case/initialization.

   For a base case, we consider before $i = 1$ iteration starts. Here, the base case should align with the loop invariant. In case of FACTORIAL(n), it will be $(1 - 1)! = 0! = 1$, which is true.

   For a loop running its body till n, it terminates at $i = n+1$, which keeps the loop invariant unchanged at $n + 1 - 1 = n$, giving us a useful result.

   In case of FACTORIAL(n), at $i = n + 1$, loop invariant $(n + 1 - 1)! = n!$, which is the final result, the procedure is expected to return.

(c) Given a n-element heap, $\lfloor n/2 \rfloor$ is a leaf in the binary heap.

Solution: False.

For any node to be a leaf, it should have no children. In case of binary heap, LEFT(i) and RIGHT(i) give us the index to the left child and right child for a node at index i. Since the heap starts filling up from left to right, we first check with LEFT(i).

LEFT(i)

return $2i$

LEFT$(\lfloor n/2 \rfloor) = 2(\lfloor n/2 \rfloor) \leq 2 * n/2 = n$

Therefore, LEFT$(\lfloor n/2 \rfloor) \leq n$ which implies that the left child of $\lfloor n/2 \rfloor$ is contained within the binary heap as it's a n-element heap.

Note that this result also implies that any node greater than $\lfloor n/2 \rfloor$ will be a leaf.

(d) Here is a pseudocode for bubblesort:

BUBBLESORT(A)

for $i = 1$ **to** $A.length - 1$
    for $j = A.length$ **downto** $i + 1$
        if $A[j] < A[j-1]$
            exchange $A[j]$ with $A[j-1]$

In the worst-case, the runtime of this algorithm is $T(n) = \Theta(n^2)$.

Solution: True

For each iteration, the inner loop runs for

- $(n-1)$ times
- $(n-2)$ times
- $(n-3)$ times
- ...
- 1 times

$T(n) = n - 1 + n - 2 + ... + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$

(e) Given $f(n) = n$ and $g(n) = 2n^2 + 3n + 5$, where $n \in \mathbb{Z}$ (integers), $f(n) \geq g(n)$ for all $n \geq 4$.
Solution: False
$f(4) = 4$ and $g(4) = 2 * 4^2 + 3 * 4 + 5 > 4$ and $f(n)$ grows slower than $g(n)$, thus it's false.

2. (15 points)PALINDROME procedure iterates over a string to check if a string is a palindrome or not. A string is considered palindrome if the string reads the same backward as forward. For instance, "madam" is a palindrome.

Input: S is a string with indices $S[1, 2, ...n]$
Output: True if S is a palindrome, otherwise False.

PALINDROME(S)
1 $i = \lfloor (S.length + 1)/2 \rfloor$
2 $j = \lceil (S.length + 1)/2 \rceil$
3 **while** $i \geq 1$ and $j \leq S.length$
4   **if** $S[i] \neq S[j]$
5   **return** False
6   **else**
7         $i = i - 1$
8         $j = j + 1$
9 **return** True

Here is a loop invariant for PALINDROME procedure:

Before j-th iteration, $S[i + 1 \mathinner{..} j - 1]$ is a palindrome.

Use the loop invariant to prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

[Hint: The size of a string will either be even or odd. Therefore, you should argue for each case, especially for initialization/base case and maintenance/inductive step.]
Solution:

Initialization: Before the loop starts, $j-1 < i+1$, which makes the substring $S[i+1 \mathinner{..} j-1]$ empty, making it trivially true.

Maintenance: We assume that before j-th iteration, $S[i + 1 \mathinner{..} j - 1]$ is a palindrome. As the body of the while loop runs with j-th iteration, there are two possible cases: (i) either line 4 holds true, and the procedure returns False, terminating the loop, (ii) or line 7 and 8 run, increasing the value of j by 1 and decreasing the value of i by 1. In case (i), as the loop terminates the loop invariant stays the same and the procedure ends. In case (ii), line 4 infers that $S[i] = S[j]$, making $S[i, i+1, \mathinner{..} j-1, j]$ a palindrome, which is the loop invariant before $(j + 1)$-th iteration starts, preserving the loop invariant for the succeeding iteration.

Termination: The loop terminates, when
case 1: $j = n + 1$, at the same time $i = 0$. Before $(n + 1)$-th iteration, $S[i + 1 \mathinner{..} j - 1] = S[1, \mathinner{..}, n + 1 - 1] = S[1, \mathinner{..}, n]$, which shows that the entire string is a palindrome, which is also the desired result in this case.
case 2: the loop terminates prematurely during j-th iteration, when line 4 holds true, preserving the loop invariant $S[i + 1 \mathinner{..} j - 1]$.

(Please use this page to write your answer for question (2) if the previous page is not enough.)

3. (25 points) Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 2T(n/2) + n^2$, $T(1) = 1$

   (a) Draw a recursion tree for $T(n) = 2T(n/2)+n^2$ to determine an asymptotic **upper bound**.
   Solution:

   Each node will have two branching until the branching reaches the leaves. *Note that* $T(n) = T(n/2) + T(n/2) + n^2$*, where* $T(n)$ *depends on two recursive calls,* $2T(n/2)$*, and the depth of the tree depends on how the input is getting divided. In this case,* n *is divided by* 2 *with each recursive call.* Therefore, the recursion tree will have $\lg n + 1$ depth, and $n^{\log_2 2} = n$ leaves.

   More importantly, in this problem, we should focus on the pattern we get while adding up the cost ocurring at each level.

   - at level 0, $n^2 = \frac{n^2}{2^0}$.
   - at level 1, $(n/2)^2 + (n/2)^2 = \frac{n^2}{2} = \frac{n^2}{2^1}$.
   - at level 2, $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = \frac{n^2}{2^2}$
   - ...
   - at level i, $\frac{n^2}{2^i}$
   - ...

   This pattern continues till the bottom of the tree, and we know the depth of the tree is $\lg n + 1$.
   Therefore, $T(n) = \sum_{i=0}^{\lg n} \frac{n^2}{2^i}$.
   You must note that, since we are enumerating from $i = 0$*, we go up to* $\lg n$*, as the depth of the tree is* $\lg n + 1$.

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n} \frac{n^2}{2^i} \\
&\leq \sum_{i=0}^{\infty} \frac{n^2}{2^i} \\
&= n^2 \left(\frac{1}{1 - 1/2} \right) \\
&= 2n^2
\end{aligned}
$$

   The calculation shows $T(n) \leq 2n^2$, which implies that the upper bound of the runtime should be $T(n) = O(n^2)$.

(b) Use the substitution method to verify your upper bound.

Solution: To show that $T(n) = O(n^2)$, we need to show that $T(n) \leq cn^2$ for some $c > 0$.

To show this, our inductive hypothesis will be $T(m) \leq cm^2$, for $m < n$. Therefore, $T(n/2) \leq c(n/2)^2$.

Now, we substitute $T(n/2) \leq c(n/2)^2$ to evaluate $T(n)$, yielding,

$$\begin{aligned} T(n) &\leq c(n/2)^2 + n^2 \\ &= (c/4 + 1)n^2 \\ &\leq cn^2 \quad (\text{c} \geq 4/3) \end{aligned}$$

To make this conclusion, we need to decide the value of c, such that $(c/4 + 1)n^2 \leq cn^2$, this leads to $(c/4 + 1) \leq c$. Solving $(c/4 + 1) \leq c$, we find $c \geq 4/3$. For any value of $c \geq 4/3$, if we choose a c, we can always show that $T(n) \leq cn^2$.

One can notice that we are not arguing about n_0 here to find the upper bound. An upper bound (for $T(n)$ above) is determined based on two constants, c, and n_0, such that $0 \leq T(n) \leq cn^2$, and $n \geq n_0$. With the substitution method above, we inductively showed that for any $c \geq 4/3$, $T(n)$ will always result in $T(n) \leq cn^2$. Therefore, we merely need to decide on n_0 such that $T(n_0) \leq c(n_0)^2$, which can always be decided for a given base case. However, in some cases, the base case do not hold for any choice of c, in that case, we simply shift the base case to higher values of n such that for a larger value of c, those base cases satisfy. ***The conclusion is, as long as we can inductively show that*** $T(n) \leq cn^2$ ***for some*** $c > 0$***, we can conclude that*** $T(n) = O(n^2)$*, which is what we do with the substitution method.*

4. (10 points) Using the definition, show that $T(n) = \frac{1}{2}n^2 - 3n$ is $T(n) = \Theta(n^2)$.

   Solution: This is straight out of the book, pg 46, CRLS 3rd Edition. As long as you use the definition of $\Theta(g(n))$ (where $g(n) = n^2$ in this case), and find the constants c_1, c_2, and n_0, you can show that $T(n) = \Theta(n^2)$.

5. (25 points) Heaps

   (a) (10 points) Write pseudocode for BUILD-MAX-HEAP(A) procedure. You may assume that MAX-HEAPIFY(A, i) has already been implemented, where i is an index of any node in the binary heap.

   Solution: pg. 157 CLRS 3rd Edition. You will get full points as long as your pseudocode is correct.

   BUILD-MAX-HEAP(A)

   $A.heapsize = A.length$
   for $i = \lfloor A.length/2 \rfloor$ **downto** 1
       MAX-HEAPIFY(A, i)

   (b) (5 points) State an upper bound for the runtime of BUILD-MAX-HEAP(A) procedure. Justify your answer.

   Solution: A simpler upper bound for BUILD-MAX-HEAP(A) is $O(n \lg n)$. We know that MAX-HEAPIFY(A, i) takes $O(\lg n)$, and this procedure is called $O(n)$ times, therefore, making the runtime $O(n \lg n)$.

(c) (10 points) Illustrate BUILD-MAX-HEAP(A) on array $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$.

Solution: While building a binary heap using an array view, we simply write the elements from left to right, filling out all the nodes in a binary heap.

The solution to this question is in pg. 158 CLRS 3rd Edition.

The iteration starts at $i = \lfloor A.length/2 \rfloor$, and decreases down to 2. With each iteration, we call MAX-HEAPIFY(A, i), eventually, building max-heaps upward, by fixing all max-heap violation starting from $\lfloor A.length/2 \rfloor$ to 2.