Midterm #1

CIS 121—Fall 2016

In-class exam: Thursday, September 29th, 2016.
Exam Starts: 10:30AM
Exam Ends: 11:50AM

This is a closed book exam. No computers or internet-connected devices are allowed during the exam. You are permitted to use one 8.5”x11” page with handwritten notes (these can be on both sides of the paper). If you need scratch paper, please get some from the front of the classroom. There are 10 questions on this exam, plus 2 extra credit questions, for a total of 12 questions. May the odds be ever in your favor.

Your name:

Your PennKey (i.e. ccb):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(circle one)</td>
<td>Mon 12-1</td>
<td>Mon 2-3</td>
<td>Mon 4-5</td>
<td>Mon 5-6</td>
<td>Tues 12-1</td>
<td>Tues 2-3</td>
<td>Tues 3-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tues 4-5</td>
<td>Tues 5-6</td>
<td>Mon 1-2</td>
<td>Mon 3-4</td>
<td>Mon 6-7</td>
<td>Tues 6-7</td>
<td>Mon 11-12</td>
<td>Tues 1-2</td>
</tr>
</tbody>
</table>

The box below will be used during grading.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Question</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Total w/o EC</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Total w/EC</td>
<td></td>
</tr>
</tbody>
</table>
1 Analysis of Algorithms
10 points

For each of the statements below, please say whether it is true or false, and give a sentence explanation of your answer.

1. Worst case analysis provides a running time bound that holds for every input of length $N$.

2. Worst case analysis is usually easier to establish than average case analysis.

3. We retain lower order terms in asymptotic analysis, since we are concerned with getting a very accurate estimate of running time.

4. Constant factors can depend on system architecture, choice of compiler or programming language.

5. To establish the bounds on the class of algorithms that solve a problem, we typically implement an algorithm to establish the lower bound, and rely on a proof to establish the upper bound.

6. Big Oh provides a good estimate of the average running time for an algorithm.

7. Asymptotic analysis is concerned with large values of $N$ and can be inaccurate for small $N$.

8. If an algorithm has a running time of $\Theta(N \log N)$ then: (say True/False for each of the items below).
   (a) It is $O(N \log N)$.
   (b) It is $\Omega(N \log N)$.
   (c) It is optimal.

9. If the lower bound on the class of algorithms that solve a problem is an algorithm $\Omega(N^2)$, and an algorithm in that class is $O(N^2)$ then:
   (a) The algorithm is $\Theta(N^2)$.
   (b) The algorithm is $\Omega(N^2)$.
   (c) The algorithm is optimal.

10. If two algorithms have the same running time in terms of Big Oh, then they will have equivalent running times in Tilde
Solution

1. Worst case analysis provides a running time bound that holds for every input of length $N$. TRUE! Since worse case analysis looks at the input that is the most difficult for the algorithm, it stands to reason that any inputs that are easier than that difficult input should take no more time.

2. Worst case analysis is usually easier to establish than average case analysis. TRUE! The proofs for average case analysis require a good model of what average input looks like, which can be difficult, or can rely on a probabilistic guarantee (like in quicksort) which can be difficult to work through.

3. We retain lower order terms in asymptotic analysis. FALSE! We ignore lower order terms, since as $N$ grows very large, they have little impact when trying to estimate the running time.

4. Constant factors can depend on system architecture, choice of compiler or programming language. TRUE! Our justification for dropping constant factors in asymptotic analysis is that the often depend on things that are external to the algorithm, like system architecture.

5. To establish the bounds on the class of algorithms that solve a problem, we typically implement an algorithm to establish the lower bound, and rely on a proof to establish the upper bound. FALSE! We can establish an upper bound on a problem/class of algorithms by implementing an algorithm, but for lower bounds we have to show that all imaginable algorithms must do at least that much work. The lower bound requires a proof rather than in implementation.

6. Big Oh provides a good estimate of the average running time for an algorithm. FALSE! This is a common misinterpretation of Big Oh notation. A counter example is quicksort which is $O(N^2)$ but which has a much better average running time.

7. Asymptotic analysis is concerned with large values of $N$ and can be inaccurate for small $N$. TRUE! Asymptotic analysis discards lower order terms, which are more dominant for small values of $N$ than with large values. Typically we establish a threshold $N_0$ after which our asymptotic analysis holds, but that would be a large value.

8. If an algorithm has a running time of $\Theta(N \log N)$ then:
   (a) It is $O(N \log N)$. TRUE. The definition of $\Theta$ is $O = \Omega$.
   (b) It is $\Omega(N \log N)$. TRUE. The definition of $\Theta$ is $O = \Omega$.
   (c) It is optimal. FALSE. Just because we have established a tight bound on the running time of an algorithm does not mean it is optimal. For example, selection sort is $O(N^2) = \Omega(N^2) = \Theta(N^2)$ but it is clearly not an optimal sorting algorithm because others like merge sort do better than it does.

9. If the lower bound on the class of algorithms that solve a problem is an algorithm $\Omega(N^2)$, and an algorithm in that class is $O(N^2)$ then:
   (a) The algorithm is $\Theta(N^2)$. TRUE. It must have an $\Omega(N^2)$ (as described below) so therefore it is $\Theta(N^2)$.
   (b) The algorithm is $\Omega(N^2)$. TRUE. If there is a lower bound on the class of algorithms and this algorithm is in that class, then it $\Omega$ is the same as for the class.
   (c) The algorithm is optimal. TRUE. Its worse case time is equal to the lower bound for the class of algorithms, so that is the best that is possible, and it is optimal.

10. If two algorithms are equivalent in terms of Big Oh, then they will be equivalent in Tilde notation. FALSE. Tilde notation keeps the leading constants, so we can differentiate between algorithms with the same BigOh. For example, two algorithm might be $O(N)$ but one could be $1000N$ and the other could be $2N$. 

4
Rubric

Each question is worth 1 point - half a point for the correct true/false answer, and half a point for a good explanation.
2  Common Orders of Growth
   10 points

The nature of software means that certain orders of growth are common in our algorithms. Enumerate the orders of growth that are typically encountered in computer science. Give their name, their expression in terms of $N$ and an example of how they arise in code. E.g.

- linear, $N$, a for loop from 0 to $N - 1$.

(Hint: you should think of at least 5 in addition to linear. You’ll get a little extra credit if you list a 6th that we haven’t talked about yet in class.)

Solution

1. constant, 1, a variable assignment like $i=0$;
2. logarithmic, $\log_2 N$, searching an already sorted list with binary sort
3. linear, $N$, a for loop from 0 to $N - 1$
4. linearithmic, $N\log_2 N$, sorting a list of $N$ items with e.g. merge sort
5. quadratic, $N^2$, two nested for loops
6. cubic, $N^3$, three nested for loops
7. exponential $2^N$, searching for all possible subsets of $N$ items (we haven’t covered exponential runtimes yet, so this one might be considered extra credit).

Rubric

RUBRIC
3 Making sorting a priority
15 points

We would like to modify a simple sorting algorithm using one of the data structures that we learned in class, and see how it changes its performance.

1. What is the running time of the selection sort code given below? No need for a formal proof, just explain your answer in a sentence or two.

2. Re-implement the search for the smallest item with a priority queue. You can write in pseudocode for your answer, and you can assume you’re already given a priority queue implementation.

3. What is the running time of your new method? State any assumptions that you make about the priority queue operations.

For this problem, running time should be in Big Oh, not Tilde.

```java
public class Selection {
    public static void sort(int[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++)
            int min = i;
            for (int j = i+1; j < N; j++)
                if (a[j] < a[min])
                    min = j;
        exchange(a, i, min);
    }
}
```

**Solution**

1. Selection sort has a running time of $\Theta(N^2)$. It loops over all $N$ items in the outer loops, and then over the items from $i$ to $N$ in the inner loop. This means that it makes $\sim (1/2N^2)$ calls to the comparison function.

2. The code is given below.

3. We can implement a priority queue with a heap that supports insertion and deletion in $O(\log_2 N)$ time. If our minPQ uses that implementation then it makes $N$ insertions and $N$ deletions, so it would be $O(N \log_2 N)$

**Rubric**

```java
public class PQSelection {
    public void sort(int[] a) {
        int N = a.length;
        MinPQ pq = new MinPQ();
        for (int i = 0; i < N; i++)
            pq.insert(a[i]);
        for (int i = 0; < N; i++)
            a[i] = pq.delMin();
    }
}
```
4 Picking the Right Sort
10 points

Insertion sort is $O(N^2)$ and merge sort is $O(N \log_2 N)$. Should we always prefer merge sort over insertion sort? Explain your answer. Give two reasons.

Solution

There are a number of reasons why you might want to pick insertion sort over mergesort in practice, despite mergesorts better asymptotic runtime. First, insertion sort is in-place and mergesort is not, so in an odd case where space constraints are present it might be a better idea to pick insertion sort. Second, if the array to be sorted has any invariants, insertion sort might be a better choice. For example, if we know that each element in the unsorted array is at most $k$ places away from its place in the final array, where $k$ is some small constant, then the runtime of insertion sort will be $O(kN) = O(N)$, which is better than mergesorts runtime of $O(N \log_2 N)$.

Rubric

RUBRIC

5 Impossible Running Time?
15 points

In class we discussed several implementations of a max priority queue that could sort Java Objects. We determined that the running time for the different implementations was the following:

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>delete max</th>
<th>find max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>$\log_2 N$</td>
<td>$2 \log_2 N$</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>$\log_d N$</td>
<td>$d \log_d N$</td>
<td>1</td>
</tr>
</tbody>
</table>

We also alluded to more complex data structures which had better running times without discussing their details.

Prove or disprove: it is possible to have a max priority queue with an underlying data structure that supports constant time operations for insert, delete max, and find max. Please say whether it is possible or impossible, and give a justification for your answer. (Hint: think about the lower bound on comparison based sorting).
Solution

This is impossible. We can prove it with reference to the lower bound on comparison based sorting, which we proved in lecture to be $N \log_2 N$. The proof used a decision tree model. The worst case is dictated by the height $h$ of the decision tree can support no more than $2^h$ leaves. The minimum number of leaves need for $N$ distinct possible combinations of keys is $N!$. By Sterlings approximation we know $\log_2 N!$ is proportional to $N \log_2 N$. (The students don’t have to give all of the details of the proof. It’s sufficient to say that we proved a $N \log_2 N$ lower bound for compare based sorts).

If the max priority queue had constant runtime, then we would be able to do sorting with fewer than $N \log_2 N$ compares via heapsort. Since we know that this is impossible, at least one of the operations must take more than constant time.

Rubric

RUBRIC
6 Symbol Tables and BSTs
5 points

For each of the statements below, please say whether it is true or false, and give a 1 sentence explanation of your answer.

1. Height of a binary search tree with \( N \) elements will always be the floor of \( \log_2 N \).
   FALSE. It must be a balanced binary tree for this to be the case.

2. There is a single unique binary search tree for any input sequence.
   FALSE. there are many ways of constructing a binary search tree from the same input array.

3. A heap and a BST are both binary trees, but we implemented the heap using an array and a BST using a Java object representing a tree.
   TRUE. The heap we represented as a complete binary branching tree but instead of explicitly representing the tree as an Object we used an array indexed from 1 that allowed us to store children of a node at \( k \) at \( 2k \) and \( 2k+1 \). For BSTs we built a java object with the left and right children as Nodes.

4. Just like a heap, a BST is a complete binary tree.
   FALSE. A BST does not have to be balanced, and that’s why we compute the worst time analysis of its operations in terms of the height of the tree instead of \( \log_2 N \).

5. Unlike a max priority queue, the ordered symbol table API (like the one we implemented with a BST) supports functions to get the minimum and get the maximum element.
   TRUE. We showed how the symbol table could support a wide variety of ordered operations (min, max, floor, ceiling, rank, num items in a range). These are made efficient with the binary search tree data structure that we use to implement the symbol table API, whereas the PQ’s heap only efficiently supports either min or max.

Solution

Rubric

RUBRIC
7 Sorting in Java
15 points

We want to find good TAs for 121 next year, so we started writing an Object that stores information about a student (given below). Our goal is to sort the students by their CIS 121 scores, and email the top 20% of the students to ask them to apply to be TAs.

1. If we want to use one of Java’s built in sorting methods, what Java Interface must we implement? Write a short bit of code that implements the method from that interface, so that instances of the Student class will be sorted in descending order of cis121score. This means that students with the highest scores are ranked first.

2. Uh oh, CCB and Rajiv disagree about whether to sort by the 160 or the 121 scores. Luckily, Java overloads its sorting functions so that in addition to passing in an array to be sorted, you may also pass in another Object that allows a different sort order. What is that other Object? How would it simplify your code to satisfy CCB’s and Rajiv’s conflicting demands?

3. If we just email the 20% of the students who have the highest scores, then we might email graduating seniors too. Instead, someone proposes that we first use quicksort to rank the students by score, and then use quicksort again so that it re-groups the students by year. Then we could email the first 20% of the juniors, sophomores and freshman in the list. When we implement this idea, people with bad scores are going to be emailed. What went wrong and how do you fix it?

```java
public class Student implements ??? { 
    protected String name;
    protected int graduationYear;
    protected double cis121score;
    protected double cis160score;
    ...
}
```

Solution

1. Note that Collections.sort sorts from low to high, but we want high to low. I apologize, can’t figure out how to get verbatim blocks inside the solution block.

```java
@Override
public int compareTo(Student s) {
    return s.cis121score - this.cis121score;
}
```

2. You have to pass in a Comparator. Rajiv and CCB would be able to implement their own custom comparators. Rajiv can compare using cis160score and CCB can compare using cis121score.

3. Quicksort is not a stable sort. Instead of using quicksort, we can use something that is stable like mergesort.

Rubric
RUBRIC
8 Dynamically Resizing Arrays
15 points

In our implementation of a stack using a dynamically resizing array, when we try to push to the stack but the underlying array is full, we double the array’s size.

1. Why do we double the size of the array each time instead of just increasing the size of the array by 1? Discuss the worst-case and amortized running time of push() in your answer.

2. When we try to pop from the stack, we halve the size of the array when it is one-quarter full. Why not when it is half full? Think about what the worst case scenario would be.

Solution

1. We double the size of the array each time instead of increasing the size of the array by 1 because in Java (and many other languages) resizing an array consists of copying every element over to a new array of the proper size. The worst case running time of push() for the doubling operation becomes $O(N)$ because in the worst case we need to double the size of the array and copy over the n elements to the new array. The amortized running time of push() is $O(1)$ because logically the number of times we have to resize the array are so infrequent that most of the time we simply add the value to the array. A conceptual way to demonstrate this is imagine each element in the array has some constant cost c but we assign them a cost of 2c. Every time the array resizes, each newly added element before the next resize use up one of its cost tokens (1 * c) to “pay” for one of the previous elements costs before them and then pay for themselves using their leftover token (another c). Note that right before the next resize, everyone is “paid” for. And following the next resize, everyone will be accounted for again. In this sense we have an amortized cost of 2c to every element, and since c was a constant, the amortized running time is just a constant, or $O(1)$.

2. If we halved the array when it was half full, there could be a worst case sequence of push and pop operations that cause us to resize the array for every operation. More specifically, say that the array currently has N elements. Consider the sequence pop(), push(x), pop(), push(x), and so on. This frequent resizing behavior is sometimes called “thrashing”. Each operation causes the array to resize, which makes the running time (both worst case and amortized) of push and pop $O(N)$ instead of $O(1)$. If we halved the size of the array when it is one-quarter full then a subsequent push(x) operation will not cause it to immediately grow again.

Rubric

RUBRIC
9 Extra Credit: Middlest Priority Queue
10 points

You have a stream of \( N \) integers coming in, and you can read one integer at a time. Design an \( O(N \lg N) \) algorithm to output a length \( N \) array, \( A \), such that \( A[i] \) is the median of the first \( i \) elements of the stream. For the sake of this problem, \( 1 \leq i \leq N \). The median is defined as the element at index \( \lceil N/2 \rceil \) in a sorted ordering. Hint: use two heaps.

For example: Given \([5, 8, 1, 4, 3, 9]\), you should output \([5, 5, 5, 4, 4, 4]\)

Solution

We maintain two heaps, a max-heap and a min-heap, and an effective median. Every element in the max-heap is lesser than the effective median, and every element in the min-heap is greater than the effective median.

Each time we process an element, we place it in the appropriate heap. If the element is greater than the effective median, it is placed in the min-heap, and if it is lesser than the effective median, it is placed in the max-heap.

Further, if the size difference between the two heaps is more than 1, we remove an element from the larger heap and place it in the smaller heap.

During this process, the number of elements in each heap can differ by almost 1. When both heaps have the same number of elements, the effective median is the average of the root element of both heaps. When the heaps do not have the same number of elements, the effective median is the root of the max-heap.

We use a pointer to keep track of the indices in the new array that stores all the medians, and every time we see a new element, we add the effective median to the array and increment the pointer by one.

Running time: Finding the effective median and inserting it into the new array takes \( O(1) \) time. Insertion into each heap takes \( O(\lg N) \) time per element, even if we need to readjust the size of the heaps. Since we have \( N \) elements, the total running time is \( O(N \lg N) \).

Rubric

RUBRIC