CSCI 161 Introduction to Computer Science

Review Guide 3 (selected sol)

Topics

- Exam I and II stuff, plus...
- List-Based Algorithms
  - Specific algorithms you may be tested on include: linear search, binary search, selection sort, and bubble sort. Understand their mechanisms and complexities.
  - Know how to write various list manipulation algorithms. Here are some examples: reversing a list, palindrome, printing unique pairs of elements, *etc.*
- I/O
  - Know how to read and write from files using `Scanner` and `PrintWriter` classes, respectively.
  - Understand how to accept and manipulate user input via keyboard (`Scanner`).
- Complexity Analysis
  - Understand how the time complexity for given algorithms is derived.
  - Understand and be able to give examples of algorithms belonging to various complexity classes: constant, logarithmic, linear, quadratic, and exponential.
  - Be able to analyze the best/worst/average-case complexity of *any* given algorithm as a function of the problem size.
  - You do **not** need to know how to analyze recursive algorithms.
- Recursion
  - Know how to read (and trace) recursive methods.
  - Be able to fix bugs in recursive methods.
  - With some guidance, be able to solve certain problems recursively instead of iteratively (with loops).
- Misc.
  - Information hiding: purpose of visibility modifiers.
  - The purpose and effect of the `static` keyword.
  - Significance of the `main()` method.
- Remember to bring your APIs to the exam! The following are acceptable: Scanner/PrintWriter, ArrayList, HashMap, String.

Practice Problems

1. **[On your own]** Go collect some data from Twitter. Save each tweet as a line to a file. Use your `TweetParser` homework assignment (or my solution) to process the statistics for all tweets in the file. You can compare your solution to mine (see: File I/O TweetParser).

   Now write a class with a `main()` method that allows users to input tweets repeatedly as strings. Process each tweet until the user decides to stop, and print out the stats collected on the entered tweets.

2. **[Code Management]** What is the significance of declaring a method to be `static`? When might you consider creating a static method?

3. **[Code Management]** What is an `enum` class used for?

4. **[Search]** When using binary search, and the array is not sorted, what can we expect to happen? (a) the program will crash, (b) you will never find the value you are looking for even if it does exist in the array, or (c) sometimes the search will be successful, but sometimes it will fail even if the value does exist in the array.

   Solution: C. You can expect it to work sometimes and not others, and it's only by chance. Its answer is not reliable.

5. **[Sorting]** The bubble sort algorithm is given below:

```
1  public void bubbleSort() {
2      boolean swapped = true;
3      for (int i=0; swapped && i < A.length; i++) {
4          swapped = false;
5          for (int j=1; j < A.length - i; j++) {
6              if (A[j-1] > A[j]) {
7                  int temp = A[j-1];
8                  A[j-1] = A[j];
9                  A[j] = temp;
10                 swapped = true;
11             }
12         }
13     }
14 }
```

   (a) What are the correct contents in `A` after each pass, if `A` is input as: 15,20,18,10? (4pts)

   i. $15, 10, 20, 18 \rightarrow 15, 10, 20, 18 \rightarrow 15, 10, 18, 20 \rightarrow 10, 15, 18, 20$

   ii. $15, 18, 10, 20 \rightarrow 15, 10, 18, 20 \rightarrow 10, 15, 18, 20$

   iii. $15, 20, 10, 18 \rightarrow 10, 15, 20, 18 \rightarrow 10, 15, 20, 18 \rightarrow 10, 15, 18, 20$

   iv. $15, 18, 10, 20 \rightarrow 10, 15, 18, 20$

   Solution: The pattern observed in (ii) is correct. The largest item always percolates up to the right location.

(b) With regards to time-complexity, the best case for bubbleSort() is observed when the list `A` is already sorted in ascending order. Explain why.

Solution: When the list is already sorted, bubbleSort only costs $N-1$ comparisons. This is done by checking to see if an element was ever swapped. If no swap happens after all elements have been visited, then everything must have been in order. So we can break out of the outer loop after just one pass, which requires $N-1$ comparisons inside the inner loop.

6. **[Complexity]** What does it mean when we say that an algorithm's time complexity is *consistent*? Why is that significant?

Solution: A consistent algorithm will perform the same number of comparisons regardless of input data or other factors. That is, it has the same best/worst/average cases. It is significant because its performance is predictable.

7. **[Complexity]** Consider the following algorithm that searches an array of Strings for duplicate elements.

```
1  public static boolean hasDuplicates(String[] list)
2  {
3      for (int i = 0; i < list.length; i++) {
4          for (int j = i + 1; j < list.length; j++) {
5              //found one!
6              if (list[i].equals(list[j])) {
7                  return true;
8              }
9          }
10     }
11     return false;
12 }
```

Assuming that n, the list size, is arbitrarily large, analyze the best and worst case complexities for this algorithm. Explain the conditions under which each case would be observed, and give the complexity as a mathematical expression in terms of n. You can ignore the average case.

To do a complexity analysis on any algorithm, it's important you understand mechanism under which it operates. The outer loop (i) iterates through each element of the list, and the inner loop (j) iterates through each element succeeding position i. The comparison is done once per inner-loop iteration between elements at i and j, but it can terminate early if the two elements are the same (that is, a duplicate has been found).

The **best case** occurs when a duplicate number is found at positions [0] and [1]. In this case, only 1 comparison was made before termination. In the **worst case** there are no duplicates in the list, and you only know that by having compared every unique pair of integers, breaking out of both loops and returning false. In the first iteration of the outer loop, $n-1$ comparisons are made. In the second iteration of the outer loop, $n-2$ comparisons are made, and so on. This results in the sum, $1+2+...+(n-2)+(n-1)$, which we know to be $n(n-1)/2$.

8. **[Complexity]** In class, we saw that the time it takes to execute the `binarySearch` method depends on the number of elements in the input array. Suppose that it takes 3 seconds on average to complete a `binarySearch` of an array with 1024 elements. Approximately how long would it take to search 16,777,216 elements, and why? In explaining your answer, make specific reference to the mathematical formula(s) we studied in class. Recall that $\log_a b = \frac{\ln b}{\ln a}$.

   Solution: We know that binary search requires $\log_2 n$ comparisons in the worst/average case. Here, $n = 1024$ so it must've done $\log_2 1024 = 10$ comparisons in 3 seconds. That gives us an average of 0.3 sec per comparison. Binary search would make 24 comparisons over $n = 16,777,216$, resulting in an estimated runtime of 7.2 seconds.

9. **[Complexity]** Write a `static` method `boolean exists(int[] list, int k, int m)` that inputs a 1D array of integers, a search key K, and a positive integer $M (M \geq 0)$. This method should determine if there are at least M copies of K in the list. For instance:

```
int[] list = {5,1,2,3,4,2,3};
System.out.println(exists(list, 2, 1));
> true

System.out.println(exists(list, 2, 2));
> true

System.out.println(exists(list, 10, 0));
> true

System.out.println(exists(list, 5, 2));
> false
```

   Assuming that N, the list size, is arbitrarily large, analyze the best, worst, and average-case complexities for this algorithm. Explain the conditions under which each case would be observed, and give the complexity as a mathematical expression in terms of M and N. Furthermore, does list order matter, and would it affect the best and worst cases? (You can ignore the average case)

   Solution: Best case is triggered when the first M items are equal to the search key, in which case, M comparisons are made. Worst case is when the entire list must be traversed to find the M copies, in which case N comparisons are made.