CSCI 132 Data Structures—Lab #7

Introduction

Today's lab is about algorithmic efficiency. Get ready, as usual:
`cp -ri ~csci132/labs/lab7 labs/ && cd ~labs/lab7/`

Reminder about the big O notations

Remember from class, big-O is an upper bound. But for the problems today, we will treat it kind of as a tight upper bound. So, if we ask you to find a $O(n^2)$ algorithm, that means, find an algorithm that takes roughly a multiple of n^2 steps. We'll provide a few "crude" examples below. Consult them for all of the Big-O questions. Ask a lot of questions to Prof Mohsin and the TAs.

Today, most of the tasks are broadly divided into three parts. 1) writing code, 2) Big-O analysis, 3) Timing code run-time. You can focus on parts 1 and 3 first.

Examples

Constant amount of work is $O(1)$.

```
// assigning a value
int a = 100;
// popping from a stack
stk.pop()
// enqueing to a queue
q.enqueue(1)
//-------------------------
// The following is NOT O(1). Why?
// checking if an item belongs to an ArrayBag
bool b = A.contains(10)
```

Simple loops with n iterations where the task inside the loop takes constant number of steps, the task can be $O(n)$

```
\\The following is O(n)
for(int i = 0; i < n; i++)
{
    c = a + b;
    a = b;
    b = c;
}
\\The following is also O(n). Why?
for(int i = 0; i < 10*n; i++)
{
    // append an item to an ArrayList
    list.append(i);
}
```

Binary search is an algorithm that takes $O(log(n))$ time.

Nested for loops can be interesting and challenging to think about.

```
\\ This is an example of code that has about n^2 steps
\\ So this is O(n^2)
for(int i = 0; i < n; i++){
    for(int j = 0; j < n; j++){
        // do some O(1) operation using i,j,n etc
        ...
    }
}
```

Find the Closest Pair in $O(n^2)$ time

Consider the problem of finding a closest pair of integers in a list of n numbers, i.e., a pair of integers `x` and `y` such that `|x-y|` is the minimum with respect to all pairs. For example, given the list `1, 5, 10, 4, 8`, the pair `5` and `4` is the closest, with distance 1.

The program `Pairs.cpp` reads in a sequence of n integers into an `ArrayList<int>` and calls a function `findClosestPair`, which you are to implement.

- Implement an $O(n^2)$ algorithm to find a closest pair of integers in a given list of n integers, print out the pair, and return the distance. If there are ties, print the first of the closest pairs.
- Compile using the commands in `Makefile`, or use the shortcut `make pairs`, then run using: `./pairs`
- The program can also generate n random numbers for you, by running: `./pairs n`

Question 1: In a separate plain-text `lab7.txt` file, give a justification for why your approach has $O(n^2)$ running time, and why this is the most-precise statement we can make about the function in terms of Big-O.

Reminder: some operations for `ArrayList` execute in $O(1)$ time, but other methods can be much slower, requiring $O(n)$ time to execute a single list method, where n is the size of the list being operated on.

Sample Output

Example 1:

```
Enter a number of values greater than two: 5
Enter a space delimited sequence of 5 integers: 1 5 10 4 8
A closest pair is: 5 and 4
Closest distance is: 1
```

Example 2:

```
Enter a number of values greater than two: 6
Enter a space delimited sequence of 6 integers: -1 3 6 -4 8 -8
A closest pair is: 6 and 8
Closest distance is: 2
```

Example 3:

```
Enter a number of values greater than two: 7
Enter a space delimited sequence of 6 integers: 21 13 8 5 3 2 1
A closest pair is: 3 and 2
Closest distance is: 1
```

Computing Unique, Common Items (Slowly)

In an earlier lab we computed the *intersection* of two sets, that is, the items two sets have in common. Let's we revisit that operation, but this time using sorted lists instead of sets. Suppose we have two lists of approximately equal length, which are already sorted into ascending order (smallest to largest). There may be duplicate values in the lists. The goal is to create a list of *unique* values the two original lists have in common. In other words, an item x should only appear in the result list if it was in both of the two input lists. Further, the result list should be sorted in ascending order, with no duplicates.

In `ComputeCommonSlow.cpp`, we have provided an implementation following nearly the identical algorithm as was used for set intersection previously. It iterates through each item in the first list. If that item is found to be in the second list, but not yet in the result, then it is appended to the end of the result list.

- Compile the program using the commands in `Makefile` or using `make slowcommon`, and run the program using `./slowcommon`. The program will ask for a length to use for the lists, and then compute the intersection of two lists of random numbers of that given length. The program prints out, along with the lists if they're not too long, a rough measurement of how long the `computeCommon` function actually took to execute on radius for the given list size.

  Try a few small sizes, no larger than 30, to make sure you understand the result being computing and the output.

Question 2: Examine the code for `computeCommon` and, in your `lab7.txt` file, give an estimate for the worst-case Big-O run-time of this function. Briefly justify your answer. Be careful: there are *three* lists involved in the loop–`first`, `second`, and `result`–and the size of one of those lists is changing as the algorithm runs. The code is calling methods on all those lists, so you need to think about what those methods do, how big those lists are (in the worst case), and why this still adds up to no more than your Big-O estimate. Hint: this implementation of `computeCommon` is definitely *worse* than $O(n)$.

Note: To speed up your code, you can consider commenting out the `assert(verifyCommon(...))` line. That line is mostly there for debugging purposes.

Question 3:

Run the program with much larger list sizes. Find a size that takes about 2 seconds to execute. Find a size that takes about 4 seconds, or 8 seconds. Report your results in your `lab7.txt` file. List each of there list sizes and their corresponding execution times in your `lab7.txt file.`

Computing Unique, Common Items (Faster)

We can improve the algorithm significantly: because the input lists and result list will be in sorted order, we can use an efficient binary search rather than `ArrayList::contains`. Make a new copy of the program, named `ComputeCommonFaster`. (Note: You can just use `cp ComputeCommonSlow.cpp ComputeCommonFaster.cpp` to create a copy.) Write a new function that takes an `ArrayList` and a target item as a parameter, and uses binary search to check if the list contains the target item. The function should return a boolean to indicate if the target was found. Change the code for `computeCommon` so it calls your binary-search function (twice), rather than `ArrayList::contains`.

As a reminder, the binary search algorithm operates as follows:

- Declare two variables (e.g. `low` and `high`), to delineate the range of possible indices that need to be searched. Start with `low = 1` and `high =` n, where n is the size of the list being searched. Remember: `ArrayList` is 1-indexed.

- Calculate the index of the (approximate) midpoint betwen `low` and `high`. Examine that item.

- If the middle item equals the target, the search can stop. Otherwise, depending on whether the middle item was larger or smaller than the target, the `low` or `high` variables should be adjusted to reduce the range of possible indices to either just the left, or just the right half of the list.
- Continue in this fashion until either the target item is found (return `true` in this case), or the range of possible indices left to search has been reduced to nothing (return `false` in this case).
- You can use recursion. Or, you can use a simple loop.

Compile the program using the commands in `Makefile` or using `make fastercommon`, and run the program using `./fastercommon`. Test your program with small list sizes until you are confident it is correct

Question 4: Repeat your Big-O analysis of this program.

- In `lab7.txt`, give a Big-O estimate of the worst case run-time, and briefly justify it. Your estimate should be strictly better than the previous "slow algorithm" Big-O execution time.
- Find list sizes that cause your algorithm to take about 2 seconds to execute, or 4 or 8 seconds and record the sizes and their corresponding runtimes in your `lab7.txt` file.

Computing Unique, Common Items (Fastest)

We can improve efficiency yet further using a completely new $O(n)$ algorithm, inspired by a 2-finger approach we might use in the physical world. Given two sorted lists of items, you can place your left index finger on the first item of one list, and your right index finger on the first item of the other list. As items are compared and placed in the result list, you slide one finger or the other down the list, until one list or the other is exhausted. Make a third copy of the program, named `ComputeCommonFastest`, to implement this new efficient $O(n)$ algorithm. You won't need binary search at all, or the `ArrayList::contains` method. Instead, your code should:

- Declare two integers to serve as the list indices, one for the first list, one for the second list. Initialize both variables to 1.
- Examine the list items at those positions, one item from each list.
- If one item is less than the other, advance that index forward by one.
- If both items are equal, then add that item to the result list and advance *both* indices forward. Be careful about duplicates: either advance both indices enough to reach a different item (without falling off the end), or check the result list (quickly!) as items are added to avoid duplicates. Hint: We can quickly retrieve the most-recently added entry in the result list.
- Repeat so long as both indices are still valid.

Compile the program using the commands in `Makefile` or using `make fastestcommon`, and run the program using `./fastestcommon`. Test your program with small list sizes until you are confident it is correct

Question 5:

- Again, run a few tests by hand to find sizes that take about 2, 4, or 8 seconds to execute and record these times in your `lab7.txt` file.

What to Turn In

Submit a soft copy of your source code and responses using the following command:

```
~csci132/bin/submit
```

Be sure you add your name and date to the prologue for each file you modify.