CSCI 262 Data Structures

Spring 2019

Lab 4: Analysis

(Back to Labs)

Goals

- Practice analyzing algorithms
- Validate your analysis with experimental data

Overview

This lab will walk you through some exercises concerning analysis. As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

Instructions

Step 1: analyze

For this lab you will try to find the "big-O" complexity of three common algorithms: selection sort, insertion sort, and the Fisher-Yates shuffling algorithm. (Yes, you can look these up on Wikipedia, but don't until you've at least tried to analyze them yourself, first.)

To start with, look at the pseudo-code given below for each algorithm. If you find the pseudo-code too confusing, you can try the actual code implementing these functions in lab4.cpp, but pseudo-code is how algorithms are typically expressed. We already looked at selection sort in class, and worked through an analysis of it. Now try to work out the "big O" complexity of the other two algorithms, expressing the "big O" in terms of the size of the input. Don't worry too much if you are struggling with the analysis of these algorithms at this point in your training as a computer scientist, but try your best! Tell us your analysis in the lab 4 quiz in Canvas.

Selection sort

```
selection_sort(A) : A is a vector
    for left = 0 to A.size - 2 do
        let right = left
        for j = left + 1 to A.size - 1 do
            if A[j] < A[right] then let right = j
            swap values A[left] and A[right]
```

In words, selection sort repeatedly looks for the minimum element in the unsorted part of the vector (initially the whole vector is considered unsorted). The minimum element is swapped with the leftmost index of the unsorted portion, at which point the leftmost index is added in to the sorted portion.

Insertion sort

```
insertion_sort(A) : A is a vector
    for i = 1 to A.size - 1 do
        let x = A[i]
        let j = i - 1
        while j >= 0 and A[j] > x do
            A[j + 1] = A[j]
            let j = j - 1
        A[j + 1] = x
```

In words, insertion sort starts by assuming that the first value (e.g., the sub-vector of length 1 starting at index 0) is "sorted". Now, starting with the second element and moving up, each element in turn is shifted left (into the sorted part of the vector) until it finds its proper place in the sorted part (which is when the element is no longer less than the element to its left).

Fisher-Yates shuffle

```
shuffle(A) : A is a vector
    let i = A.size - 1
    while i > 0 do
        let index = a random integer between 0 and i, inclusive
        swap values A[i] and A[index]
        i--
```

In words, Fisher-Yates starts at the end of the vector and swaps the last value with the value in a position chosen at random from all possible positions in the vector. The last position is then fixed, and the remaining values are shuffled, i.e., the second-to-last value is swapped with a value chosen at random from all but the last position, etc. When only one position is left, there is no reason to continue and the algorithm is finished. Given a true random value function, this algorithm produces a "perfect" shuffle (i.e., all permutations are equally probable).

Step 2: experiment

How well do these algorithms perform in practice? You will do some experiments to find out. Download lab4.cpp and put it into a new project. The code should compile and run as is. You can do anything you want to with this code - it is your microscope for this lab, but you won't be turning it in. You can use the code as is for the rest of the lab, or you can customize it if you want to streamline your data gathering.

As written, the lab code prompts you for an algorithm to test, whether to use random or sorted vectors, a size of vector to test with and a number of trials to run. You want to vary the size of the vector to see how the cost of the algorithm varies with size - we are trying to estimate some function of the size of the input. For some algorithms, it can be difficult to accurately measure timing with just one run, particularly if the algorithm runs very fast (or the input size is small). So we typically measure a lot of trials of the same algorithm and average over them. The code does all of the work to run the trials, creating appropriate vectors each time and executing and timing the algorithm on the vectors. At the end, it reports the average time to execute the algorithm in seconds.

Now, for each algorithm, for both random and sorted data, run timing experiments for at least 6 sizes of input - you will need quite a few data points to get a good experimental result. Collect the timing data and (optionally) plot it in Excel (or the graphing tool of your choice). Your x-axis should show the size of the input, while your y-axis should indicate timing information. Label your axes! You will want to experiment with and plot each experiment separately, as it is difficult to get them all on one plot in a useful form - three algorithms x 2 types of vector = 6 experiments in total. If you choose not to plot your data, please present it in Canvas in a well-labeled tabular form, along with some text explaining what you see in the data.

In your results, you may notice that one algorithm (we won't spoil it for you and tell you which one) actually gives different results depending on whether the input data is sorted. For this algorithm, we actually have two "big O" measurements. One is what we call the "worst case" complexity - meaning that, for any type of input, we can't do worse than this complexity. The other big O measurement in this case is the "best case" complexity - we can't do any better than this with this algorithm, and we can only achieve the best case with certain types of input. See if you can figure out why this algorithm does so much better with one type of input than the other. In addition to "best" and "worst" case complexities, computer scientists are often interested in "average" case complexities, which can be different from the others, or the same as one or both of the others. You'll learn more about these complexities in CSCI 406.

Based on your experimental data, revisit your "big-O" analyses. Provide your experimental data and your analyses in the lab 4 quiz in Canvas.