Group 15 Activity 4: Sorting

CS 162, Fall 2016

Group Members who participated fully:
Peter Moldenhauer
Jessica Wade
Philip Sigillito
Mike Mann

Group members who participated some:
Michael Kozlowski
Joel Yoder

Group member who did not contribute:
Clifford Cho

You just finished collecting and putting in order of 200 applicants for a new job. Then 50 more applications come in and "have to be included". The final new larger stack of applications need to be sorted.

1. Describe an algorithm to sort the new come in applications and include them into the larger stack:

Before our group decided on a specific algorithm to use, we first discussed the criteria as to what the applications would be sorted by. As long as we could take two "objects" and say that one is bigger and one is smaller, only then could we start figuring out the algorithm to sort the new applications and then merge the sorted pile into the large (already sorted) pile. We need to base our sort of the smaller pile based on how the larger pile is sorted and then make sure we can compare the two piles to create one large sorted pile. For example, we could compare names to sort the smaller pile but what happens if the large pile is already sorted based on a ranking system of the most desirable candidate. We cannot compare names to this ranking system. Through the course of our group discussion, it was decided that we would sort the applications in alphabetical order of last name. We felt that this would reflect a real-world scenario the best.

After weighing the pro's and con's of two different potential algorithms to use for this problem, our team decided on the following algorithm (below). In this algorithm, the 50 new applications are first divided into two sub-groups and each of the two sub-groups are sorted. Once each of the two sub-groups from the 50 new applications are sorted, then they are sorted into the large stack of the 200 applications:

1. To sort the new applications:
    a. Divide the new applications into two piles by first letter of last name: A-M and N-Z.
    b. For each of the two piles of new applications, search through to find the lowest value last name and bring it to the first position.

    c. Repeat step 1b for each of the following positions in the pile with the remaining applications until the two piles are completely sorted.
    d. Place the A-M pile on top of the N-Z pile. Now, all of the new applications should be sorted.
2. To include the new applications in the larger stack:
    a. Compare the new application at the top of the new stack to the value of each application in the large stack until an application is found that should NOT come before the new application in question.
    b. Put the new application in front of it in the large stack.
    c. Move on to the next new application.
    d. Continuing from the spot where you left off in the large stack, repeat steps 2a-2c, until the new stack has all been added OR the last application in the large stack is reached.
    e. If any new applications have not been added when you reach the end of the large stack, add all remaining new applications to the end, keeping them in their order.

2. Describe an algorithm to bite the bullet and just put them in one at a time.

For this algorithm, our team again decided to base the sorting of the applications alphabetically by last name. This algorithm is somewhat similar to the approach in #1. The main differences are that it eliminated the sorting of the new pile prior to merging, and added a comparison to the most recently added application to see if it is necessary to start at the beginning of the stack.

1. Start at the top of the large stack and compare the new application's value to each of the old applications.
2. As soon as an application is found that should NOT come before the new application in question, put the new application in front of it.
3. If the end of the large stack is reached and the new application has not been inserted into it, add it to the back of the large stack.
4. Compare the next new application to the value of the new application that was just added to the large stack. If the value is greater than the one just added, then continue scanning the pile from application immediately following the last new application.
5. If the value is not greater, repeat steps 1 and 2.
6. Repeat steps 1-5 until all new applications have all been added.

3. Estimate the complexity of each. Which one is better? What if the number (200/50) changes?

Our group decided that while the first algorithm would be more complicated to code in a computer program, it would be more efficient than the second algorithm. The second algorithm is a lot simpler, but not nearly as efficient. Sorting with the first algorithm would be more efficient because you only have to start at the beginning of the large stack once, with the first new application, therefore reducing the total number of comparisons required. This algorithm is related to a recursive sort merge algorithm. This is introduced on slide 5 of the sort-merge lecture. We are sorting the stack of applications into smaller stacks and then merging all those small stacks into one. Slide 10 of the sort-merge lecture shows sort-merges have a complexity of O(n · log n). With the second algorithm, we are basically just inserting and sorting each application one at a time. This would have the complexity of O(n2), because this algorithm (where we insert one application at a time) most closely describes a bubble sort algorithm. Although the "semi-binary" step we added to the second algorithm made it more efficient than starting at the beginning each time, it still did not make it more efficient than the first algorithm.

If the numbers of the applications (200/50) changes, this would affect each of these two algorithms differently (due to their complexity). As the number of applicants increased, the first algorithm becomes increasingly more efficient. On the other hand, the second algorithm becomes increasingly less efficient as the number of applications increase.

Additionally, if new applications were to come in mid-merge, the second algorithm would allow more flexibility to include the newest new applications into the large stack. With the first algorithm, any applications that come in mid-sort would have to be put aside until the merging of the already sorted new applications is completed. Then, the newest new applications would be sorted and merged, requiring a restart from the beginning of the algorithm.

4. Compare the algorithms you develop to standard computer algorithms. Which computer algorithm is similar to your manual algorithm? Why? If you don't think it is similar to any computer algorithms, explain why not.

For the first algorithm, we determined that the method that we used to sort the new applications was closest to a selection sort algorithm, since we pulled out the lowest valued new application for the first position, and repeated this for each position that follows with the remaining applications. Therefore, we were performing a selection sort on the two subgroups before we stacked them on top of each other. Our group also decided that the merging approach taken in our first algorithm was closest to the merge sort algorithm. In this first algorithm we developed, the new stack was completely sorted in the same manner as the large stack, then they were merged together. This is, essentially, the Merge Sort.

The second algorithm we developed was closest to the bubble sort algorithm. With our algorithm we are essentially adding each new application into position 0 initially. Then, this new application gets swapped with the next element as many times as needed, as it moves deeper into the large stack of applications until it no longer requires a swap. Each new application would then get shuffled along in the same pair-swapping manner as bubble sort until they were all in order. Overall, as you go through and sort the applications using this algorithm, the new application essentially "bubbles" up to the location it is supposed to be and you are making multiple exchanges throughout the first pass until it reaches its proper position.

Unlike approach #1, where the next new application will always fit into the large stack somewhere after the previous new application, the next new application in approach #2 may belong before the previously added new application. Before starting back at the top of the stack, the second algorithm first compares the next new application value to the most recently added new application. The comparison step helps to determine whether the starting point should be at the beginning or after the most recently added application, thus making the approach slightly more efficient than starting at the beginning each time. This was inspired by the binary search algorithm, since this step could eliminate the need to compare to all of the large stack applications that come before it. This could potentially prevent the need to place the new application in position 0, but instead, if the application added before it ended in position n, the new application would start in position n + 1. We coined this a “semi-binary” approach, as it could reduce the number of comparisons needed to place the newest application because the earliest location that would result from the bubble sort would be position n + 1. Then, the new application would continue to follow the bubble sort algorithm until it reaches its correct spot in the large stack.