Ivy Truong

3/2/2020

CSCI 3412

Homework 3

1.

a) We are given that there are sublists and that the lists are length of k. The worst case for insertion sort of size n is O(n2) when n is the input size (in terms of k as the length because we are sorting sublists it would be O(k2)). To find the worst case for insertion sort of sublists, we get O(k2 \* ) = O(nk).

b) We are given that there are sublists and the sublists are length k. To merge all of the sublists we

use the conquering method. We know that sorting each subarray for a normal merge sort in each level takes O(logk) because you are sorting every element in each level of subarrays/sorting k sublists since merge sort divides until k = 1. For sorted sublists and sorting all the lists together, it takes log() to sort the lists because there are lists that need to be sorted. Instead of sorting every element, we are sorting every sublist. After sorting, we would merge the lists. The time it takes to merge is O(k \* ) = O(n), because there are sublists and k elements, it would take the number of sublists multiplied by the total number of elements in each sublist to merge everything into a single array. The total worse case time for merging sublists together becomes O(n) \* O(log) = O(nlog).

c) If the modified algorithm runs O(nk + nlog()), the largest asymptotic value of k as a function of n for it to be same as the merge sort is log(n). To solve, we have O(nk + nlog()) = O(nlogn) to find out the largest asymptotic value that equals the running time of merge sort, which is O(nlogn). There are two possibilities for solving k, either O(nk) = O(nlogn) or O(nlog()) = O(nlogn). When solving for k by doing nk = nlog(n), then k equals log(n). When solving for nlog() = nlog(n), k = 20 = 1. Out of these two equations, the largest asymptotic value out of both of them is k = log(n), which is the largest k where the modified algorithm will have the same running time as merge sort.

2.

a) The loop invariant for the outer loop and the inner loop in the selection sort algorithm is finding the smallest value in the array. In the inner loop, it checks each element in the array and finds the index of the smallest value in the array. In the outer loop, it will swap the smallest value in the array and moves it to the leftmost position in the array, then finds the next smallest value and continues the process.

b) In bubble sort, the neighboring values in the array are compared to see which out of the two is the smallest value and swaps so that the smallest value is on the left side. It is different from selection sort because bubble sort checks two neighboring values at a time (so it only checks two values), while selection sort checks one value with all the other values in the array one at a time (checks the whole array in respect to current value to be sorted).

c) In insertion sort, the rightmost value in the unsorted array is compared to each value in the array and shifts other values in the array to the right until it finds the right position. It is different from selection sort because insertion sort involves shifting values in the array one position to the right if it is bigger than the current value that is being placed, while selection sort keeps track of the index of the smallest value and swaps it in the right position. Insertion sort takes longer since it has to move up to n values in the array while selection sort just has to swap 2 values in the array when sorting.

d) To show that bubble sort actually sorts, we need to prove that the neighboring elements are sorted. For example, we would need to show that A’[1] <= A’[2], A’[2] <= A’[3], up to A’[n-1]. If i = 0 and is incremented by 1 after each comparison until it reaches the last element (n-1), we would need to show that A[i] <= A[i+1], and if not, swap the neighboring values.

e) The loop invariant for the selection sort in lines 3-5 is to find the index of the smallest value in the array from A[j….(n-1)].

* Initialization
  + Prior to the start of the loop, we have that min\_index = i, where i = 0. We see that the element in A[0] is the first element we know so far/current element we are checking, and see this is currently the smallest value in the array that we know. So currently the index of the smallest element before the start of the loop is min\_index = 0, which satisfies the loop invariant.
* Maintenance
  + For the next iteration(s), we check the current value in the min\_index to the other values in the subarray. If A[min\_index] > A[j], then the min\_index = j. This will store the new minimum index value into min\_index when the minimum value is greater than the current value in the array we are comparing. To preserve the loop invariant, j is incremented by 1 so that the next value in the array is compared to the minimum value based on the min\_index stored.
* Termination
  + The loop terminates when it has reached the last element in the subarray, where j = len(A) – 1. This shows that each value in the array has been compared to obtain the smallest value in the array. The loop will end with the min\_index of the smallest value in the subarray from A[j….len(A) – 1)], which shows that the algorithm is correct.

f) The loop invariant for the selection sort in lines 1-6 is to move the smallest value in the array to the leftmost position in the array until all values are sorted in ascending order, where A[1] <= A[2] <=A[3] … <= A[n-1] and n is the length of the array.

* Initialization
  + Prior to the start of the loop, i = 0, therefore the current smallest value known so far is A[0]. Since A[0] is the current value we know, it is the smallest value in this subarray and is technically sorted, satisfying the loop invariant.
* Maintenance
  + For the next iteration(s) of the algorithm, the values are checked and swapped if needed to get the smallest value in the leftmost position in the array. Since the inner loop value j depends on i, the inner loop will find the smallest value in each subarray of the whole array. For example, to find the smallest value in the array, it starts with i = 0 and j =1, so the inner loop will compare the value in A[0] to the values in A[1……len(A) – 1] and record the index of the smallest value. After retrieving the index of the smallest value in the subarray, it will swap the value in the leftmost position in the original array (in this case A[0]) with the smallest value, which is in A[min\_index]. To preserve the loop invariant, i is incremented by 1 so it finds the next smallest value, so the algorithm will compare the value in A[i] to the values in A[i+1…..len(A) – 1] to find the next smallest value and move it to the next leftmost position.
* Termination
  + The algorithm ends when i = len(A), which means that it has traversed through the whole array. This also means that the smallest value from the original unsorted array is in the leftmost position of the new sorted array, and so on where the largest value from the unsorted array is placed in the rightmost position in the new sorted array. The algorithm will end with the original array sorted in ascending order, which shows that the algorithm is correct.

3.

a) To choose k in practice, k should be the largest list length in which the slowest sort out of the two is faster than the fastest sort of the two. For example, between selection sort and merge sort, selection sort (slower than merge sort with worst case O(n2)) should handle the smaller sized arrays up to length k while the merge sort (faster than selection sort with worst case O(nlogn)) should handle sorting the larger sized arrays, or anything larger than k.

b) Yes

c) Code and output on separate file

d) For the selection sort and merge sort algorithm combo, I figured out k to be log(n) through these steps. (k is the length of the sublists, n is the length of the original array that has not been split into the sublists, and is the number of sublists)

* Selection sort’s worst case is O(k2), and with sublists the worst case for sorting each sublist is O(nk).
* Merge sort’s worst case is O(klogk), and with sublists the worst case is O(nlog()).
* Finding the largest k for selection and merge sort combined to have the same run time as merge sort alone is similar to problem 1c in this homework, where the largest k is log(n).

Performance data with a combination of selection and merge sort with array size of 1,000,000

|  |  |
| --- | --- |
| k (length of sublist) | Time taken to execute algorithm (seconds) |
| k - 10 | 382.000 |
| k - 5 | 365.000 |
| k | 335.859 |
| k + 5 | 352.844 |
| k + 10 | 360.656 |

* Value of k found for this algorithm is 19.

4.

a) One type of recursion is the substitution method. This method lets us guess a bound for the solution and prove it using mathematical induction. Usually with this method, the recursion tree method is used to make a good guess so it can be used as the base case in mathematical induction.

Another type of recursion is the recursion-tree method. This method provides the sum of the nodes at each level that represents the total work done and where each node represents a single subproblem in the set of recursion functions called. In the end of this method, a recurrence relation can be solved by finding a pattern from the costs in each of the levels, which can help us determine a guess of the running time of the recurrence relation.

The last type of recursion is the master method. This method provides bounds for the recurrence relations using the form T(n) = aT(n/b) + f(n), where a ≥ 1 and b > 1. The constants a represents the number of recursive calls made, b represents the factor in which the work is reduced by each call, and f(n) is a function. This method contains three cases that a recurrence can fall under using the form above:

* Case 1: If f(n) = O(nlogba-e) where e > 0, then T(n) = O(nlogba)
* Case 2: If f(n) = Θ(nlogba), then T(n) = O(nlogba \* log(n))
* Case 3: If f(n) = Ω((nlogba+e) where e > 0 and a\*f() ≤ c\*f(n) for some c > 1 and large n, then T(n) = O(f(n))