*Homework #1*

*By Rasa Rasiulyte*

*Phone 425.218.9263*

*9/6/2020*

1. Analyze the following algorithms for performing searches
   1. **LINEAR-SEARCH(x, A)**

|  |  |  |
| --- | --- | --- |
| *Code to execute* | *Cost* | *Time* |
| i = 1 | C1 | 1, the run time constant here since this line only gets executed once |
| **while** i <= n and x != A[i] | C2 | O(n), the number of comparisons here will depend on input n here at worst. |
| i = i + 1 | C3 | O(n), because it is the body of the loop and this line will get executed until x is found (in the worst case until i becomes larger than n) |
| **If** i <= n | C4 | 1 (constant) outside the loop |
| location = i | C5 | 1 (constant runtime here) |
| **else** location = 0 | C6 | 1 (constant) |
| **return** location | C7 | 1 (constant) |

The running time of the algorithm is the sum of running times for each statement executed, so to compute T(n) we will do following:

T(n) = C1\*1 + C2\*n + C3\*n + C4\*1 + C5\*1 + C6\*1 + C7\*1

T(n) = C2n + C3\*n , to get upper time complexity of T(n), we will drop lower order terms and constants and will be left with T(n) = O(n).

* 1. **BINARY-SEARCH(x, A)**

|  |  |  |
| --- | --- | --- |
| *Code to execute* | *Cost* | *Time* |
| i = 1 | C1 | 1 (constant) |
| j = n | C2 | 1 (constant) |
| **while** i < j | C3 | log n, if loop here will iterate over the entire array, so running time will be equal to O(n). But because we are reducing amount of information by half on lines below, the running time is O(log n) |
| *m = [ (i + j) / 2 ]* | *C4* | *Ignore this line according to directions, so we will assume this run some constant time.* |
| **if** x > A[m] | C5 | if x is greater than array A at the m’th element/index. Using m here just divided our data in half. Due to this, every time our loop executes, we reduce the amount information by half. |
| i = m + 1 | C6 | 1 (constant) |
| **else** j = m | C7 | if else, the j is going to be moved back to the half way point. |
| **if** x = A[i] | C8 | 1 (constant) |
| location = i | C9 | 1(constant) |
| **else** location = 0 | C10 | 1(constant) |
| **return** location | C11 | 1(constant) |
|  |  |  |

The running time of the algorithm is the sum of running times for each statement executed, so to compute T(n) we will do following:

T(n) = C1\*1 + C2\*1 + C3\*log n + C4+ C5\*1 + C6\*1 + C7\*1 then to get upper time complexity of T(n), we will drop lower order terms and constants and will be left with **T(n) = O(log n).**

1. [Followed provided hint]

T(n) = T() +1

1. T(n) = 2T(n/3 + 1) + n, Master method does not apply directly here, but since any constant is degree-0 polynomial, we can express 1 as (n^0), and because it is lower order form than that of n^1, it can be ignored. And subsequently, the guess for upper bound is T(n) = This guess can be checked by substitution or master method.
   1. We know that worst case time for insert sort (of length k) is k^2, then for the sublist n/k, worst case time is (n/k)\*(n^2), which results to nk. So that proves the upper bound for insertion sort .
   2. We know we have n/k sublist. Merging 2 of the n/k lists at a time until all elements are merged gives us log(n/k) complexity. And then we will need to multiply n for each merge step to account for worst case comparison. So overall complexity results to
   3. Provided that standard running time for merge sort is , it is reasonable to find a value of k that doesn’t grow faster than log(n), otherwise the nk term would result in a worse complexity than standard merge sort. So when k = log n, time complexity for modified algorithm will be
   4. In practice, I would not want to break the datasets into super small pieces, which could result in wasteful read and write data, unless k is too large (to get exact numeric value for “too large” I would need to consider system that this algorithm runs on and benchmark n and k, to get the values of k where the largest sublist length makes insertions merge sort faster than regular merge sort).