CMPT 435 - Spring 2019 - Dr. Labouseur

Assignment Two – Sorting, Searching, & Hashing

Christopher Ravosa
christopher.ravosa1@marist.edu

March 15, 2019

1 Comparisons in Sorting Methods

Selection Sort	3459	$O(n^2)$
Insertion Sort	114,309	$O(n^2)$
Merge Sort	665	$O(n \cdot \log_2 n)$
Quick Sort	417	$O(n \cdot \log_2 n)$

2 Explanation - Running Times of Sorts

Here, the asymptotic running times of the above sorts will be explained. We'll start with selection sort. Selection sort has a run-time of $O(n^2)$ simply because it operates on a method which implements nested for-loops. For-loops have a run-time of n, being the number of times the loop runs. The inner-loop runs n-times per iteration of the outer-loop. Therefore, the inner-loop's number of iterations is multiplied by the number of times it runs. This means it is multiplied by the number of times the outer-loop runs. Therefore, selection sort has a run-time of $O(n \cdot n)$, or $O(n^2)$.

Moving on, insertion sort has a running time of $O(n^2)$ as well. The reason for this is similar to the reason for selection sort having the same run-time. Insertion sort also runs on the implementation of nested loops. The outer-loop is a for-loop which runs n-times. The inner-loop, however, is a while-loop. While this is a different kind of loop, the while loop still runs n-times throughout the course of the method. This is because insertion sort increases the size of the portion of the array it is sorting with each iteration of the higher-up, for-loop.

The while-loop doesn't run n-times with each iteration, but the number of times it runs changes with each run through the for-loop.

The last sorts are merge sort and quick sort. These sorting methods have a run-time of $O(n \cdot \log_2 n)$. Merge sort and quick sort are both divide-and-conquer methods which means they are recursive. This means they can be visualized as recursion trees. The height of their recursion trees 'h' is equal to $\log_2 n$. This value of height refers to the number of times the methods call themselves. For each level of recursion, the methods break the arrays down a little more. Because the arrays are divided with each recursive call, they become a fraction of the total length of the initial array value 'n'. Because of this, sorting through each sub-array requires n/x amount of effort. Therefore, after sorting through each sub-array, the functions perform 'n' amount of effort at each level of the total height of the tree. Since the height is equal to $\log_2 n$, and the functions perform 'n' effort at each level, the final asymptotic run-time for each function is $O(n \cdot \log_2 n)$.

3 Comparisons in Searching Methods

Linear Search	282	O(n)
Binary Search	8	$O(\log_2 n)$
Hashing	2	O(n)

4 Explanation - Running Times of Searches

Here, we will discuss the asymptotic run-times of the above searching methods. To begin, we'll look at linear search. Linear search is a simple search method that accepts a target value and iterates through an array to find it. Since this method runs on a single for-loop that runs from the beginning of the array until it finds the target value, it has a worst-case run-time of O(n). This is because the target value may be at the end of the list, or it may not be present at all. Therefore, the search will often run through the entire array of length 'n'. Therefore, linear search's asymptotic run-time is O(n).

Binary search is another recursive function. In contrast to the previous recursive functions discussed above, it has an asymptotic run-time of $O(\log_2 n)$. Why is binary search's run-time not multiplied by 'n'? This is simply because binary search does not do the extra 'n' work with each recursive call. This is the case because binary search does not iterate through the sub-arrays it creates. Instead, binary search simply ignores the sub-arrays until it finds the target value. This is because binary search is not a sorting method, which means it doesn't need to merge the arrays back together. Therefore, it can literally toss the values that are confirmed to not be the target value. This is why binary search's run-time is simply $O(\log_2 n)$.

The last search and final method whose run-time we will discuss is searching via hash table. This is technically not a searching method, but is more of a data structure. This implementation of a hash table is interesting in that the

hash table is created and then the target values are pulled out of it after it has been loaded. The asymptotic run-time of locating and retrieving values from the hash table we created (thanks to the help of Professor Labouseur's code) is O(n). This is because the first part of finding a value involves moving to the linked-list whose key corresponds to the value in the hash table that is equivalent to the target value's hash-code. This is a constant time operation that can be dropped when calculating the big-O run-time. If the linked-list here is only one node in length, we've found our guy. However, assuming collisions have occurred, we may encounter a linked-list with more than one node. In this case, we must implement a linear search to iterate through the linked-list until we find the target value in the linked-list that corresponds to the target value's hash-code. As stated above, linear search has a run-time of O(n). Therefore, this implementation of searching using a hash table is O(1+n) which is equal to O(n).