COMP3506/7505 Week 4 Tutorial

Sorting Algorithms, Lists, and Amortization

Questions marked with a star ($\star$) are strongly recommended. Try to complete these before your tutorial. These questions will be covered during the tutorial class. While your tutor will attempt to cover all these questions, not all may be covered depending on time constraints. The questions without a star will *not* be explicitly covered during the tutorial class but are provided as additional resources for extension and further practice. You are welcome to ask your tutor about these problems if you wish. Otherwise, you are also welcome to post any questions you have on Ed Discussion.

1. ($\star$) The following questions ask you to use specific sorting algorithms to sort this list of numbers:

$$[6,\ 10,\ 8,\ 3,\ 2,\ 12,\ 5,\ 1].$$

   (a) Sort the list using merge sort. Show your progress using the recursion tree representation used in the lectures.

   (b) Sort the list using quicksort. Use the value at index position $\lceil n/2 \rceil$ as the pivot, where n is the size of the list to be sorted. Show your progress using the recursion tree representation used in the lectures.

   (c) Sort the list using a base-2 radix sort. Convert each number to a 4-digit binary number before performing the radix sort. Show your progress using the sequence representation used in the lectures.

   (d) How many operations were required to sort the list of numbers using each sorting algorithm?

   (e) Discuss the performance of each of these sorts based on the number of operations required. What is the asymptotic performance of each sort?

   (f) How do the observed number of operations relate to the theoretical asymptotic performance of each sort?

   (g) What would you expect to happen as the size of the list increased? (Say, from 8 to 800 to 800,000 numbers.)

2. ($\star$) Let's sort a deck of playing cards. Suppose we want the suits in the order of $\clubsuit, \diamondsuit, \heartsuit, \spadesuit$. Within each suit, cards should be sorted in numerical order. The final result will be something like A$\clubsuit$, 2$\clubsuit$, …, K$\clubsuit$, A$\diamondsuit$, …

   To do this, I group the cards by suit. Then for each suit, I first look for the Ace and move it to the front, then 2 after that, then 3, and so on.

   (a) What sorting algorithms have I used to sort these cards?

   (b) Suppose a deck has s suits and k ranks. What is the worst-case running time of this sorting process in terms of s and k?

   (c) Assuming a deck has exactly sk cards with none missing or repeated, how could we sort the deck into this order in $\Theta(sk)$ time?

3. ($\star$) Start with an empty singly linked list where the nodes store single characters. Assume that you only have a reference to the start of the list.

   Draw a boxes and arrows diagram of inserting the characters 't', 'h', 'e' (in that order). Pay careful attention to the cursor reference that accesses the nodes, and start at the beginning of the list when inserting each character.

   Then, insert the characters 'e', 'r' (in that order) so that when you iterate from the start of the list, the characters spell the word "there".

4. ($\star$) Suppose we want to maintain a sorted list of integers. Initially, we will store this inside an array.

   (a) Given a new element x, how would you (efficiently) find where to insert it?

   (b) What is the running time of inserting an element, in big-O notation?

   (c) Could we use a linked list to improve this running time? If so, explain how, otherwise explain why not.

5. You are given the first node of a singly linked list. Each list node has a *next* field which points to the next node, or is null if this is the last node.

   (a) Write pseudocode to reverse this singly linked list, i.e. reverse all the *next* pointers. Your algorithm should take a single argument, the original list's head, and return the head of the reversed list.

   Try to do this in $\Theta(n)$ time and $O(1)$ space.

   (b) Draw a boxes and arrows diagram to show how your algorithm reverses the list $[1, 2, 3, 4]$.

6. Consider an extensible list implementation which doubles its capacity if an `add()` would exceed its current capacity. Prove that with this resizing strategy, `add()` runs in *amortised* constant time.

7. Suppose we want to sort an array of strings. A string is made up of characters and each character is a byte within $0, \ldots, 255$. Therefore, we can (in theory) use radix sort.

   Discuss the advantages and disadvantages of using radix sort in this situation.

8. An adaptive sorting algorithm is one which can take advantage of its input being partially sorted. Such algorithms often run in $O(n)$ time for sorted or almost sorted input.

   For each of the following algorithms, determine whether it is adaptive or not and explain why:

   insertion sort, selection sort, merge sort, quicksort, bucket sort.

9. Quicksort requires a subroutine which partitions array values by whether they are less than or greater than some pivot.

   Specifically, the partition algorithm takes an n-length array A and some pivot value p. After executing *partition*, A should be organised into all the values $\leq p$, then p, then all values $> p$. The function should return the (new) index of the middle p.

   Write pseudocode for a simplified $\mathit{partition}(A, n)$ where the pivot is always chosen as $A[0]$. Your *partition* should run in $O(n)$ time and use $O(1)$ additional space.

   Here, you can assume elements of A are distinct. You can also assume that $\mathit{swap}(A[i], A[j])$ swaps the i-th and j-th elements of A.