Homework 1
Design and Analysis of Algorithms
Sections BCS-4C, BCS-6A, Spring 2021

Due Wednesday, March 31, 2021 at 1159pm

Instructions

In the following, wherever you are asked to design/write an algorithm, two things are required: a bullet list of algorithm steps written in English, a clean and precise Pseudocode.

Problem 1

a. Write an in-place version of the merge function that works in $O(n^2)$.

b. Now Merge Sort uses your $O(n^2)$ function from part a, what is its new recurrence? Solve this recurrence using the recursion tree method and find a big-O bound.

c. Suppose you're choosing between the following three algorithms.

   (i) Algorithm A solves problems by dividing them into seven subproblems of quarter the size, recursively solving each problem, and combining the solutions in linear time.

   (ii) Algorithm B solves problems of size n by recursively solving two subproblems of size $n - 1$ each and then combining the solutions in constant time.

   (iii) Algorithm C solves problems by dividing them into nine subproblems of size $n/3$ each, recursively solving each problem, and combining the solutions in $O(n^2)$.

   What are the running times of each of these algorithms (in big-O terms), and which one would you choose?

Problem 2

a. Figure 1 shows an algorithm called Insertion Sort. Read the algorithm carefully and dry run it on the array: $A = [5, 9, 0, 2, -1, 11, 6, -2, 5, 8]$. Show the contents of A after each execution of line no. 8. Note: A.length = 10, and indexing begins from 1.

b. Perform step count analysis on the code in Figure 1 and show that the asymptotic running time of Insertion Sort is $T(n) = O(n^2)$.

c. It is claimed that Insertion sort is faster than any of its competing $O(n^2)$ sorts. This claim becomes important for small values of n. For large values of n, $O(n^2)$ is simply too slow and the differences of small constant multiples inside the big-O don't make a difference.

In this exercise, your job is to test the claim for small and moderate values of n; specifically, for n=10, n=25, n=100, n=1000 and n=10000. You will practically compare the execution times of Insertion Sort, Selection Sort and Bubble sort.

You must follow these steps:

(1) Write C++ functions for InsertionSort, BubbleSort and SelectionSort.

(2) For n=10, 25, 100, 1000 and 10000:

   i Generate a random array, A, of n integers.

   ii Make three identical copies of A: A1, A2 and A3.

   iii Execute InsertionSort on A1, BubbleSort on A2 and SelectionSort on A3, and compute their execution times T1, T2 and T3.

   iv Present your results in a tabular format, with rows for values of n and columns for each algorithm. Each entry should show the running time as a suitable fraction of a second.

(3) Now implement Merge Sort and compare it with Insertion Sort for an array of size 8, and size 100,000. What time difference do you see in each case?

Visit here to see how you can measure the elapsed time of a C++ program using the chrono library.

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Figure 1: Insertion Sort

Problem 3

Given an unsorted array of integers, A, its size n, and two numbers x and y both elements of A, write an algorithm that returns the *distance* between x and y. The distance between two numbers of an array is the number of elements that lie between them in the sorted order. Achieve the asymptotically fastest time for this problem.

Problem 4

a. We saw in class that dividing a Merge Sort problem into 3 subproblems may not be advantageous. Now we explore in more depth the effect of dividing into multiple subproblems. Describe a recurrence $T(n, m)$ of a Merge Sort variant which divides a problem of size n into m subproblems of size n/m each and merges their outputs. Give an expression for $T(n, m)$ in terms of n and m. Pay attention to how much time will be needed to merge the solutions of the m subproblems of size n/m each. Will it still be cn?

b. Consider the following: In Merge Sort we divide the array as usual for the first $k - 1$ levels of the recursion tree (where k is a parameter to the Merge Sort function). After that, we use Bubble Sort on the kth level to sort the subarrays at that level and go back up the tree merging as usual. What is the running time (in big-O terms) for this strange algorithm? The expression for $T(n, k)$ should involve both n and k.

Problem 5

a. In Data Structures, you studied binary heaps. Binary heaps support the insert and extractMin functions in $O(lgn)$, and getMin in $O(1)$. Moreover, you can build a heap of n elements in just $O(n)$. Refresh your knowledge of heaps from chapter no. 6 of your algorithms text book.

   Now implement Merge Sort, Heap Sort, and Quick Sort in C++ and perform the following experiment:

   1. Generate an Array A of 10^7 random numbers. Make its copies B and C. Sort A using Merge Sort, B using Heap Sort, and C using Quick Sort.
   2. During the sorting process, count the total number of comparisons between array elements made by each algorithm. You may do this by using a global less-than-or-equal-to function to compare numbers, which increments a count variable each time it is called.
   3. Repeat this process 5 times to compute the average number of comparisons made by each algorithm.
   4. Present these average counts in a table. These counts give you an indication of how the different algorithms compare asymptotically (in big-O terms) for a large value of n.

b. Now compare the same algorithms in terms of practical time, i.e. the actual running time. Simply, repeat the previous example but use the chrono library to compute the actual times taken by each algorithm, and report the average value of the time for each algorithm.

c. The sorting algorithm available in the C++ STL is called IntroSort. Here is what Wikipedia says about it: *"Introsort or introspective sort is a hybrid sorting algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance. It begins with quicksort, it switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted and it switches to insertion sort when the number of elements is below some threshold."* In light of your experiments above, explain why IntroSort does what it does. You may take a look at its pseudo-code here.

Problem 6

A singly linked list contains $n-1$ strings that are binary representations of numbers from the set $\{0, 1, \ldots n-1\}$ where n is an exact power of 2. However, the string corresponding to one of the numbers is missing. For example, if $n = 4$, the list will contain any three strings from 00, 01, 10 and 11. Note that the strings in the list may not appear in any specific order. Also note that the length of each string is lgn, hence the time to compare two strings in $O(lgn)$. Write an algorithm that generates the missing string in $O(n)$.

Problem 7

Take the Merge Sort algorithm and modify it to solve the following problem:
Input: An array of integers, A, its size n.
Output: The number of *inversions* in A.
A pair of elements $A[i]$ and $A[j]$ is counted as an inversion if $i < j$ and $A[j] < A[i]$.

Problem 8

Input: A array of unique integers, A, its size n.
Output: Two arrays sl and sr each of size n, containing the 'left scores' and 'right scores' of the elements of A respectively. To be precise: $sl[i]$ will contain the left score and $sr[i]$ the right score of the element $A[i]$ respectively.

The scores are defined as below:
The left score of the element $A[i]$ is defined as: $sl[i] = A[j]$, where $j < i$, and $A[j] > A[i]$ and j is the nearest such index to i. Similarly, the right score of the element $A[i]$ is defined as: $sr[i] = A[k]$, where $k > i$, and $A[k] > A[i]$ and k is the nearest such index to i.

If no j exists for an i that satisfies the above condition, then $sl[i] = 0$. Similarly, if no k exists for an i that satisfies the above condition, then $sr[i] = 0$.

It should be easy to work out the $O(n^2)$ algorithm. Design an $O(nlgn)$ algorithm for this problem.

END