CS 61B Exam Prep 11: Sorting Spring 2020

1 Identifying Sorts

Below you will find intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm (that is, many steps are missing), but they are in the correct sequence. For each of them, select the algorithm it illustrates from among the following choices: insertion sort, selection sort, mergesort, quicksort (first element of sequence as pivot), and heapsort.

Input list:1430, 3292, 7684, 1338, 193, 595, 4243, 9002, 4393, 130, 1001

(a) Mergesort. One identifying feature of mergesort is that the left and right halves do not interact with each other until the very end.

1430, 3292, 7684, 193, 1338, 595, 4243, 9002, 4393, 130, 1001

1430, 3292, 193, 1338, 7684, 595, 4243, 9002, 130, 1001, 4393

193, 1338, 1430, 3292, 7684, 130, 595, 1001, 4243, 4393, 9002

(b) Quicksort. First item was chosen as pivot, so the first pivot is 1430, meaning the first iteration should break up the array into something like $| < 1430 | = 1430 | > 1430$

1338, 193, 595, 130, 1001, 1430, 3292, 7684, 4243, 9002, 4393

193, 595, 130, 1001, 1338, 1430, 3292, 7684, 4243, 9002, 4393

130, 193, 595, 1001, 1338, 1430, 3292, 4243, 9002, 4393, 7684

(c) Insertion Sort. Insertion sort starts at the front, and for each item, move to the front as far as possible. These are the first few iterations of insertion sort so the right side is left unchanged

1338, 1430, 3292, 7684, 193, 595, 4243, 9002, 4393, 130, 1001

193, 1338, 1430, 3292, 7684, 595, 4243, 9002, 4393, 130, 1001

193, 595, 1338, 1430, 3292, 7684, 4243, 9002, 4393, 130, 1001

(d) Heapsort. This one's a bit more tricky. Basically what's happening is that the first line is in the middle of heapifying this list into a maxheap. Then we continually remove the max and place it at the end.

1430, 3292, 7684, 9002, 1001, 595, 4243, 1338, 4393, 130, 193

7684, 4393, 4243, 3292, 1001, 595, 193, 1338, 1430, 130, 9002

130, 4393, 4243, 3292, 1001, 595, 193, 1338, 1430, 7684, 9002

2 Conceptual Sorts

Answer the following questions regarding various sorting algorithms that we've discussed in class. If the question is T/F and the statement is true, provide an explanation. If the statement is false, provide a counterexample.

(a) (T/F) Quicksort has a worst case runtime of Θ(NlogN), whereN is the number of elements in the list that we're sorting.

False, quicksort has a worst case runtime of Θ(N2), if the array is partitioned very unevenly at each iteration.

(b) We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

The input is small or the array is nearly sorted. Note that insertion sort has a best case runtime of Θ(N), which is when the array is already sorted.

(c) Give a 5 integer array such that it elicits the worst case running time for insertion sort.

A simple example is: 5 4 3 2 1. Any 5 integer array in descending order would work.

(d) (T/F) Heapsort is stable.

False. Stability for sorting algorithms mean that if two elements in the list are defined to be equal, then they will retain their relative ordering after the sort is complete. Heap operations may mess up the relative ordering of equal items and thus is not stable. As a concrete example(taken from Stack Overflow), consider the max heap: 21 20a 20b 12 11 8 7

(e) Give some reasons as to why someone would use mergesort over quicksort

Some possible answers: mergesort has $\Theta(NlogN)$ worst case runtime versus quicksort's $\Theta(N2)$. Mergesort is stable, whereas quicksort typically isn't. Mergesort can be highly parallelized because as we saw in the first problem the left and right sides don't interact until the end. Mergesort is also preferred for sorting a linked list.

(f) You will be given an answer bank, each item of which may be used multiple times. You may not need to use every answer, and each statement may have more than one answer.

A. QuickSort (nonrandom, inplace using Hoare partitioning, and choose the leftmost item as the pivot)

B. MergeSort

C. Selection Sort

D. Insertion Sort

E. HeapSort

N. (None of the above)

List all letters that apply. List them in alphabetical order, or if the answer is none of them, use N indicating none of the above. All answers refer to the entire sorting process, not a single step of the sorting process. For each of the problems below, assume that N indicates the number of elements being sorted.

A, B, C Bounded by Ω(NlogN)lower bound.

B, E Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

C In the worst case, performs Θ(N) pairwise swaps of elements.

A, B, D Never compares the same two elements twice.

N Runs in best case Θ(logN)time for certain inputs

3 Counting Inversions

Given an array of size N, find the number of inversions in O(NlogN) time. Hint: Use merge sort.

```
public static int countInversions (int[] arr, int left, int right) {
    int count = 0;
    int middle = (left + right)/2;

    if (left<right) {
        count+=countInversions(arr, left, middle);
        count+=countInversions(arr, middle+1, right);
        count+=mergeAndCount(arr, left, middle, right);
    }
    return count;
}

public static int mergeAndCount(int[] arr, int left, int middle, int
   right) {
    int count = 0;
    int[] leftArr = new int[middle - left + 1];
    int[] rightArr = new int[right - middle];

    System.arraycopy(arr, left, leftArr, 0, middle-left+1);
    System.arraycopy(arr, middle + 1, rightArr, 0, right-middle);

    int leftPointer = 0;
    int rightPointer = 0;
    int mergePointer = left;
    while (leftPointer < leftArr.length && rightPointer <
       rightArr.length) {
        if (leftArr[leftPointer] > rightArr[rightPointer]) {
            count += leftArr.length - leftPointer;
            arr[mergePointer] = rightArr[rightPointer];
            mergePointer++;
            rightPointer++;
        } else {
            mergePointer++;
            leftPointer++;
        }
    }

    //if uneven split
    while (leftPointer < leftArr.length) {
        arr[mergePointer] = leftArr[leftPointer];
        mergePointer++;
        leftPointer++;
    }
    while (rightPointer < rightArr.length) {
        arr[mergePointer] = rightArr[rightPointer];
        mergePointer++;
        rightPointer++;
    }
    return count;
}
```

4 Sorted Runtimes

We want to sort an array of N distinct numbers in ascending order. Determine the best case and worst case runtimes of the following sorts -

(a) Once the runs in merge sort are of *size* $<= N/100$, we perform bubble sort on them.

Best Case: N, Worst Case: N^2. Once we have 100 runs of size N/100, bubble sort will take best case N and worst case N^2 time. The constant number of linear time merging operations don't add to the runtime.

(b) We can only swap adjacent elements in selection sort.

Best Case: N^2, Worst Case: N^2. The best case and worst case don't change since swapping at most doubles the work each iteration, which produces the same asymptotic runtime as normal selection sort.

(c) We use a linear time median finding algorithm to select the pivot in quicksort.

Best Case: $NlogN$, Worst Case: $NlogN$. Doing an extra n work each iteration of quicksort doesn't asymptotically change the best case runtime, but it improves the worst case runtime.

(d) We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must have constant space complexity.

Best Case: $NlogN$, Worst Case: $NlogN$. While a max-heap is better, we can make do with a min-heap by placing the smallest element at the right end of the list, until the list is sorted in **descending order**. Once the list is in descending order it can be sorted in ascending order with a simple linear time pass.

(e) We run an optimal sorting algorithm of our choosing knowing:

- There are at most N inversions

  Best Case: N, Worst Case: N. Insertion sort takes N + number of inversions time
- There is exactly 1 inversion

  Best Case: 1, Worst Case: N. The inversion may be the first two elements, in which case constant time is needed. Or, it may involve elements at the end, in which case N time is needed.
- There are exactly $(N^2 - N)/2$ inversions

  Best Case: N, Worst Case: N. If a list has $N(N-1)/2$ inversions, it means it is sorted in descending order! So, it can be sorted in ascending order with a simple linear time pass.