Sorting

Exam Prep 12

Announcements

- Week 12 Survey due Monday 11/06
- Lab 12 due Friday 11/10
- Project 3 released
    - Project 3A due 11/13
    - Project 3B&C due 11/27

Content Review

Insertion Sort

Insertion sort iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

Runtime: $O(N^2)$

Selection Sort

Selection sort finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

Runtime: $\Theta(N^2)$

Merge Sort

Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

3 5 1 2 4

Runtime: $\Theta(N \log N)$

Quicksort

Quicksort picks a pivot (ie. first element) and uses Hoare partitioning to divide the list so that everything greater than the pivot is on its right and everything less than the pivot is on its left.

3 5 1 2 4

Runtime: Average case O(NlogN), slowest case $O(N^2)$ (dependent on pivot selection)

Heap Sort

Heapsort heapifies the array into a max heap and pops the largest element off and appends it to the end until there are no elements left in the heap. You can heapify by sinking nodes in reverse level order.

3 5 1 2 4

Runtime: O(NlogN)

Summary for comparison sorts

Stability: a sort is stable if duplicate values remain in the same relative order after sorting as they were initially. In other words, is 2a guaranteed to be before 2b after sorting the list [2a, 2b, 1]?

	Worst Case	Best Case	Stable?
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No
Insertion Sort	$\Theta(N^2)$	$\Theta(N)$	Yes
Merge Sort	$\Theta(N\log N)$	$\Theta(N\log N)$	Yes
Quicksort	$\Theta(N^2)$	$\Theta(N\log N)$	No*
Heapsort	$\Theta(N\log N)$	$\Theta(N)$	No

Try reasoning out or coming up with examples for these best and worst case runtimes!

*with hoare partitioning

Worksheet

CS 61B Sorting

Fall 2023 Exam-Level 12: November 6, 2023

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. When we split an odd length array in half in mergesort, assume the larger half is on the right.

Input list: 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

(a) 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 192, 1337, 7683, 594, 4242, 9001, 129, 1000, 4392

192, 1337, 1429, 3291, 7683, 129, 594, 1000, 4242, 4392, 9001

Mergesort

Splits each time

(b) 1337, 192, 594, 129, 1000, 1429, 3291, 7683, 4242, 9001, 4392

192, 594, 129, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392

129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

Quicksort

Notice the pivots being chosen and kept for future iterations

(c) 1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000

192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000

192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000

sorted section grows →

Insertion Sort

(d) 1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192

7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001

129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

Heap structure with maximum elements at the front

Heapsort

In all these cases, the final step of the algorithm will be this:

129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

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) 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?

Nearly sorted - hence why this is often used for database insertion

(b) Give a 5 integer array that elicits the worst case runtime for insertion sort.

Desending order (Opposite of a)

5 4 3 2 1

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

False - the heapification shuffles thing around, somewhat arbitrarily

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

Worst case runtime is better - $N \log N$ vs N^2

Stability - merge is, quick isn't

(e) 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 (in-place 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 $\Omega(N \log N)$lower bound. Insertion: Sorted takes N
Heapsort: Identical items takes N (no heapification needed)

__B, E__ Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.
Needs NlogN worst case

__A, B, D__ Never compares the same two elements twice.
Insertion - shuffles new item backwards
Quick - pivot gets set and never used
Merge - only does pairwise once in each stage

__N (None)__ Runs in best case $\Theta(\log N)$time for certain inputs
↓
$\Omega(N)$ required to at least check all items

3 Bears and Beds

In this problem, we will see how we can sort "pairs" of things without sorting out each individual entry. The hot new Cal startup AirBearsnBeds has hired you to create an algorithm to help them place their bear customers in the best possible beds to improve their experience. Now, a little known fact about bears is that they are very, very picky about their bed sizes: they do not like their beds too big or too little - they like them just right. Bears are also sensitive creatures who don't like being compared to other bears, but they are perfectly fine with trying out beds.

The Problem:

- **Inputs:**
  - A list of `Bears` with unique but unknown sizes
  - A list of `Beds` with unique but unknown sizes
  - *Note: these two lists are not necessarily in the same order*
- **Output:** a list of `Bears` and a list of `Beds` such that the `ith Bear` is the same size as the `ith Bed`
- **Constraints:**
  - `Bears` can only be compared to `Beds` and we can get feedback on if the `Bed` is too large, too small, or just right.
  - `Beds` can only be compared to `Bears` and we can get feedback on if the `Bear` is too large, too small, or just right for it.
  - Your algorithm should run in $O(N \log N)$ time on average. → hint for quicksort! mergesort wouldn't work

In order to do the cross comparison, use quicksort
↳ try with other comparison sorts between two lists

Big idea: Use the pivot from one list to sort and find equivalent pivot in other list
↓
Use this to sort the first list