CS 61B Spring 2021

Comparison Sorts

Topical Review Session 6: April 18, 2021

1 Sorting - Step by step

Show the steps taken by each sort on the following unordered list of integers (where duplicates are indicated with letters):

`2, 1, 8, 4A, 6, 7, 9, 4B`

(a) Selection Sort 1 | 2 8 4A 6 7 9 4B
1 2 | 8 4A 6 7 9 4B
1 2 4A | 8 6 7 9 4B
1 2 4A 4B | 6 7 9 8
1 2 4A 4B 6 | 7 9 8
1 2 4A 4B 6 7 | 9 8
1 2 4A 4B 6 7 8 | 9
1 2 4A 4B 6 7 8 9 |

(b) Insertion Sort

2 | 1 8 4A 6 7 9 4B
1 2 | 8 4A 6 7 9 4B
1 2 8 | 4A 6 7 9 4B
1 2 4A 8 | 6 7 9 4B
1 2 4A 6 8 | 7 9 4B
1 2 4A 6 7 8 | 9 4B
1 2 4A 6 7 8 9 | 4B
1 2 4A 4B 6 7 8 9 |

(c) Merge Sort

[2 1 8 4A 6 7 9 4B]
[2 1 8 4A] [6 7 9 4B]
[2 1] [8 4A] [6 7] [9 4B]
[2] [1] [8] [4A] [6] [6] [7] [9] [4B]
[1 2] [4A 8] [6 7] [4B 9]
[1 2 4A 8] [4B 6 7 9]
[1 2 4A 4B 6 7 8 9]

(d) Heap Sort *Note: if both children are equal, sink to the left.*

9 6 8 4A 1 7 2 4B ← heapified!
8 6 7 4A 1 4B 2 | 9
7 6 4B 4A 1 2 | 8 9
6 4A 4B 2 1 | 7 8 9
4A 2 4B 1 | 6 7 8 9

4B 2 1 | 4A 6 7 8 9
2 1 | 4B 4A 6 7 8 9
1 | 2 4B 4A 6 7 8 9
| 1 2 4B 4A 6 7 8 9

2 Sorting Runtime

Fill out the best-case and worst case runtimes for these sorts as well as whether they are stable or not in the table below.

	Best Case	Worst Case	Stable
Insertion Sort	$\Theta(N)$	$\Theta(N^2)$	Yes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No
Merge Sort	$\Theta(NlogN)$	$\Theta(NlogN)$	Yes
Heap Sort	$\Theta(N)$	$\Theta(NlogN)$	No
Quick Sort	$\Theta(NlogN)$	$\Theta(N^2)$	Depends

Notes:

- Insertion Sort is good for small and nearly sorted arrays

- Heapsort's best case is achieved when all the items are duplicates (so heapification takes constant time)

- In practice, quicksort is the fastest sort

3 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: 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

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

1429, 3291, 7683, 192, 1337, 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

(b) Quicksort. First item was chosen as pivot, so the first pivot is 1429, meaning the first iteration should break up the array into something like $| < 1429 | = 1429 | > 1429$

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, 9001, 4392, 7683

(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

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

(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.

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

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

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

4 Sorting Hat

The Sorting Hat has asked the 4 Houses to provide one new sorting algorithm each, which are modified versions of the sorts you have learned in 61B so far. For each of the algorithms, give the best and worst case runtimes using $\Theta(\cdot)$ notation with respect to the number of the elements in our input list, N. (Note that Slytherin chose not to participate as they were not selected as winners for the previous Sorting Hat competition).

(a) **LionSort:** Gryffindor has modified selection sort to additionally swap the maximum element of our unsorted list to the back at each iteration. Assume finding this maximum element takes $\Theta(N)$ time.

Selection sort has a runtime of $\sum_{i=1}^{N} i \in \Theta(N^2)$. After Gryffindor's modification, the sort does half the passes, but does more work on each of these passes because we have to find both the maximum and the minimum element in the pass. The runtime then, is $\sum_{i=1}^{N} 2i \in \Theta(N^2)$.

(b) **EagleSort:** Ravenclaw has decided to insert elements into a BST and then perform an in-order traversal on the tree to find their sorted order. In addition to performance, what sort is this most like?

First consider the runtime of inserting into a BST. For a balanced BST, the runtime for inserting N items is $\Theta(N^2)$ for spindly trees and $\Theta(N \log N)$ for bushy trees. The in-order traversal of the tree is the runtime of DFS, which is $\Theta(N)$ as we do not care about edges (our runtime is only with respect to the input list). This leaves us with a worst case runtime of $\Theta(N^2)$ and a best case runtime of $\Theta(N\ logN)$. EagleSort is most similar to quicksort as we can think of each BST element as a pivot for inserting new items.

(c) **BadgerSort:** Hufflepuff decide to modify merge sort by changing how sorted runs are merged. Instead of merging two sorted runs by iteratively choosing the minimum, Hufflepuff insertion sort the concatenation of the runs at each level using selection sort.

Hufflepuff did not make use of the fact that each of the merged halves are already sorted. Selection sort on each merge will take $\theta(n^2)$ where n is the number of items concatenated together. This sums as:

$$N^2 + 2 * (N/2)^2 + ... + \log N * (N/\log N)^2$$

Which ends up becoming asymptotically N^2. Hufflepuff made merge sort worse.

5 Choose A Sort

For each of the following scenarios, choose the best sort to use. Explain your reasoning.

(a) The list you have to sort was created by taking a sorted list and swapping N pairs of adjacent elements.

Insertion sort. A list created in such a manner will have at most N inversions. Recall insertion sort runs in $\Theta(N + K)$ time, where K is the number of inversions, so our overall runtime would be $\Theta(N)$.

(b) You have to sort a list on a machine where swapping two elements is much more costly than comparing two elements and you want to do the sort in place.

Selection sort. In its most common implementation, selection sort performs N swaps in the worst case, whereas all other common sorts perform $\Omega(NlogN)$ swaps.

(c) Your list is so large that not all of the data will fit into your computer at once. As is, at any given time most of the list must be stored in some external device (an HDD), where accessing it is extremely slow.

Merge sort. The divide-and-conquer strategy works well with the restriction on only being able to hold a partition of the list at any given time. Sorted runs of the list can be merged on your computer and flushed to the HDD one block at a time, minimizing HDD accesses. Note that Quicksort, with three way partitioning, is also valid. However, quicksort would more likely end up with a bad distribution, where as merge sort always leaves equal sized partitions.

(d) Given a list of emails ordered by send time, sort the list such the emails are ordered by the sender's name first while secondarily maintaining the time-ordering.

Merge sort. The usual implementation of quicksort is unstable and can change the time-ordering when sorting.

(e) You have a randomly shuffled list, where each number is unbounded in size, and want to sort the elements.

Quicksort. It is empirically faster than merge sort, and stability is not an issue.