CSM 61B Sorting Crib Sheet

Spring 2018 May 5, 2018

1 Overview

There exist many sorting algorithms - algorithms that turn something like [1,5,3,2,4] into [1,2,3,4,5] - each with different runtimes, uses of memory, data structures, and key high-level ideas. Each has pros and cons, of course. In this guide, we will explore the main ideas of each and go through brief examples.

2 Definitions

`Inversions`: The number of pairs of elements in a sequence that are out of order. An array with no inversions is ordered.

Example: How many inversions are in $[1, 5, 3, 3, 4]$? Answer: Three $(5, 3), (5, 3), (5, 4)$.

`Stability`: If a sorting algorithm is stable, then if we sort an array with two or more equal elements, the relative ordering of these equal elements will not change in the sorted array. As we will see, the stability of an algorithm depends on the implementation, though some some algorithms are always stable.

Example: Let us sort $[1, 5, 5, 3, 4]$. Since we have two fives, we can label them as [1, 5A, 5B, 3, 4]. If our sorting algorithm is stable, a sorted version of the array would be [1, 3, 4, 5A, 5B]. Note that 5A and 5B stayed in the same order they did in the original array!

3 "Assorted" Algorithms

1. `Selection Sort`
   - Iterate through list and find the smallest element. Swap it with the current first element (index 0). Iterate through array again and find next smallest element. Swap it with the second element (index 1). Repeat until sorted (so N iterations through the list)
   - Runtime: $\Theta(N^2)$. We iterate through our list with N elements N times.
   - Space complexity: $O(1)$. No extra space is used if done in place.
   - Notes: Simple but inefficient since we go through the array N times.
2. `Heap Sort`
   - Add all elements in the list to a heap (a min or max heap will work, though max is better for reasons described a few sentences below). Keep popping the max (or min) element from the heap and adding to an array.
   - Runtime: $\Omega(N), O(N \log N)$. In the best case, all the elements in the list are equal, so there will be no bubbling or sinking. In the best case, adding and removing from the heap will take constant time, and we do this for N elements. In the worst case, operations take $\Theta(\log N)$ time and we do this N times. So the overall runtime is $O(N \log N)$.
   - Space complexity: $\Theta(N)$. A new data structure of size N is created.

3. `In-place Heap Sort`
    - The same as before, except we do everything without creating the heap structure, so we use no additional memory! The first step is to apply bottom-up heapification to our array which converts the original array into a heap version of it.
    - Runtime: $O(N \log N)$.
    - Space complexity: $\Theta(1)$
4. `Merge Sort`
    - Divide array into two halves. Recursively mergesort each half. Finally, merge the two sorted halves.
    - Runtime: $\Theta(N \log N)$. The array is "split" $\log N$ times and N work is done at each level when we merge.
    - Space complexity: $\Theta(N)$
5. `Insertion Sort`
    - Starting at the beginning of the array, continuously swap with element to the left until it is no longer less than the left element. Repeat for all elements in the array.
    - Runtime: $\Omega(N), O(N^2)$. In the best case, the array is already sorted, and no swapping is needed. In the worst case, the array is fully unsorted. Runtime can also be denoted by $O(N + k)$ where k is the number of inversions.
    - Space complexity: $\Theta(N)$
    - Notes: Fast for arrays with a low number of inversions and for small arrays ($N < 15$).
6. `Quick Sort`
    - The key idea here is partitioning. We start by picking a pivot, which is an element in our list. Then we group together elements smaller than the pivot, equal to the pivot and greater than the pivot. After this partitioning is done, elements equal to the pivot are in the right place. Then, quicksort the less than and greater than subarrays.
    - Runtime: $\Omega(N), O(N^2)$ but $\Theta(NlogN)$ on average. We do $O(N)$ work for each recursive call.
    - Notes: In the worst case, the array is already sorted and we pick the leftmost element as the pivot always, resulting in a runtime of $O(N^2)$. To remedy this, we should pick pivots randomly (or shuffle the array before we sort it).

4 Additional Resources

7. CSM sorting worksheet. Solutions. On the solutions, you will find some more in-depth explanations of the algorithms succinctly mentioned above. It also has a problem on Radix Sort, not included here. Even if you already did the worksheet, it is pretty helpful to try again!
8. Josh Hug's Guides (on the course website). Basics. Quicksort. Even more on quicksort!. Sorting Bounds. Radix Sort.