CS 61B Spring 2026

Sorting II

Exam-Level Discussion 14: April 27, 2026

1 Summary (Fill in Names of the Sorts)

Sorting. A sorting algorithm rearranges an array into non-decreasing order. This requires a **total order** on the keys: for any `a` and `b`, exactly one of `a < b`, `a = b`, `a > b` holds (trichotomy), and `<` is transitive.

A sorting algorithm is considered **stable** if order of equivalent items is preserved.

Example: Two items, A and B, are equal. B comes before A in the unsorted array. A stable sort would ensure that B comes before A in the sorted array. An unstable sort would not gurantee this.

Quicksort. Pick a pivot, **partition** the array so that every item left of the pivot is $\leq$ pivot and every item right is $\geq$ pivot (the pivot ends up in its final sorted position), then recurse on the left and right sides.

- 3-Scan: scan three times into a new array. $\Theta(N)$ time but $\Theta(N)$ extra space.
- Hoare: walk one pointer in from the left (stopping on items $\geq$ pivot) and another in from the right (stopping on items $\leq$ pivot), swapping whenever both have stopped. In-place, $\Theta(N)$ per partition.

Runtime: best and average $\Theta(N \log N)$; worst case $\Theta(N^2)$

Stability: Depends on partitioning scheme. 3-Scan paritioning is stable, Hoare partitioning is not.

Avoiding the $\Theta(N^2)$ worst case. Four philosophies:

- **Randomness:** shuffle the input before sorting, or pick random pivots.
- **Smart pivot selection:** take the median, or an approximation such as median-of-three.
- **Introspection:** track recursion depth and switch to mergesort if recursion goes too deep.
- **Preprocessing:** try to detect pathological inputs in advance (limited effectiveness).

Counting Sort. Count the number of occurrences of each item, and store the respective counts in an array. Iterate through the unsorted array, using the count array to place each item in the sorted array.

Runtime: With an alphabet of size R, runs in $\Theta(N + R)$

Stability: Stable

LSD Radix Sort. Sort each digit independently from rightmost to leftmost using a stable sort as a subroutine (such as Counting Sort). Terminate once the leftmost digit has been sorted.

Runtime: With an alphabet of size R and longest key of width W, runs in $\Theta(WN + WR)$

Stability: Stable

MSD Radix Sort. Sort each digit independently from leftmost to rightmost using a stable sort as a subroutine (such as Counting Sort). Terminate when either the rightmost digit has been sorted or if each recursive "bucket" only contains one item.

Runtime: With an alphabet of size R and longest key of width W, runs in $\Theta(N + R)$ in the best case and $\Theta(WN + WR)$

Stability: Stable

2 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 AirBears-n-Beds 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 `i`th `Bear` is the same size as the `i`th `Bed`
- **Constraints:**
  - `Bears` can only be compared to `Beds` 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

Solution:

Our solution will modify quicksort. Let's begin by choosing a pivot from the Bears list. To avoid quicksort's worst case behavior on a sorted array, we will choose a random Bear as the pivot. Next we will partition the Beds into three groups — those less than, equal to, and greater than the pivot Bear. Next, we will select a pivot from the Beds list. This is very important — our pivot Bed will be the Bed that is equal to the pivot Bear. Given that the Beds and Bears have unique sizes, we know that **exactly** one Bed will be equal to the pivot Bear. Next we will partition the Bears into three groups — those less than, equal to, and greater than the pivot Bed.

Next, we will "match" the pivot Bear with the pivot Bed by adding them to the Bears and Beds lists at the same index, which is as easy as just adding to the end. Finally, in the same fashion as quicksort, we will have two recursive calls. The first recursive call will contain the Beds and Bears that are **less** than their respective pivots. The second recursive call will contain the Beds and Bears that are **greater** than their respective pivots.

Here is a video walkthrough of the solutions as well.

3 MSD Radix Sort

Recursively implement the method **msd** below, which runs MSD radix sort on a **List** of **Strings** and returns a sorted **List** of **Strings**. For simplicity, assume that each string is of the same length, and all characters are lowercase alphabets. You may not need all of the lines below.

In lecture, recall that we used counting sort as the subroutine for MSD radix sort, but any stable sort works! For the subroutine here, you may use the **stableSort** method from the previous question, which sorts the given list of strings in place, comparing two strings by the given index. Finally, you may find following methods of the **List** class helpful:

(a) **List<E> subList(int fromIndex, int toIndex)**. Returns the portion of this list between the specified **fromIndex**, inclusive, and **toIndex**, exclusive.

(b) **addAll(Collection<? extends E> c)**. Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

```
public static List<String> msd(List<String> items) {
    return msd(items, 0);
}

private static List<String> msd(List<String> items, int index) {
    if (items.size() <= 1 || index >= items.get(0).length()) {
        return items;
    }

    List<String> answer = new ArrayList<>();
    int start = 0;


    stableSort(items, index);
    for (int end = 1; end <= items.size(); end += 1) {
        if (end == items.size() ||
items.get(start).charAt(index) != items.get(end).charAt(index)) {

            List<String> subList = items.subList(start, end);

            answer.addAll(msd(subList, index + 1));

            start = end;
        }
    }
    return answer;
}

/* Sorts the strings in `items` by their character at the `index` index alphabetically. */
private static void stableSort(List<String> items, int index) {
    // Implementation not shown
}
```

4 Zero, One, Two-Step

(a) Given an array that only contains 0's, 1's and 2's, write an algorithm to sort it in linear time without creating a new array. You may want to use the provided helper method, swap. Hint: Consider how Hoare partitioning rearranges elements in an array.

```
public static void specialSort(int[] arr) {
  int front = 0;
  int back = arr.length - 1;
  int curr = 0;

  while (curr <= back) {
    if (arr[curr] < 1) {
      swap(arr, curr, front);
      front += 1;
      curr += 1;
    } else if (arr[curr] > 1) {
      swap(arr, curr, back);
      back -= 1;
    } else {
      curr += 1;
    }
  }
}

private static void swap(int[] arr, int i, int j) {
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}
```

(b) We just wrote a linear time sort, how cool! Why can't we always use this sort, even though it has better runtime than Mergesort or Quicksort?

Solution: While our algorithm is super cool, we were only able to write it because we knew there were exactly 3 possible values that could be in our array! For the general case, when the collections we're sorting have much more variety, we can't make these kinds of guarantees.

(c) The sort we wrote above is also "in place". What does it mean to sort "in place", and why would we want this?

Solution: In general, we consider a sorting operation to be done in place if it does not require significant extra space. "Significant extra space" is often defined as linear or greater, with respect to the number of elements we are sorting. For example, if our sorting algorithm requires making a whole new array and copying elements over to it, then it is NOT in place because we had to allocate significant Sorting 7 space for this new array. Some algorithms that can be implemented in place are selection sort, insertion sort, and heap sort. Mergesort technically can be implemented in place, but it's rather complex. Doing operations in place is beneficial because we typically want to use as little memory/computer resources as possible. Though in this class, we focus on time efficiency, space efficiency is important too in the real world!