CS 61BL Summer 2026

Counting Sorts

Reciation 03: August 03, 2026

1 All Sorts of Sorts

Show the steps taken by each sort on the following unordered list:

`0, 4, 2, 7, 6, 1, 3, 5`

(a) Insertion sort

(b) Selection sort

(c) Merge sort

(d) Use heapsort to sort the following array (hint: draw out the heap).
Draw out the array at each step: `0, 6, 2, 7, 4`

2 Quicksort

(a) Sort the following unordered list using Quicksort. Assume that we always choose first element as the pivot and that we use the 3-way merge partitioning process described in lecture. Show the steps taken at each partitioning step.

```
18, 7, 22, 34, 99, 18, 11, 4
```

(b) Suppose we use 3-scan partitioning with the **middle item** (rounded down — i.e. the item at index $\lfloor \frac{N-1}{2} \rfloor$) as the pivot at every level of recursion. Describe an input that gives $\Theta(N^2)$ runtime, assuming we don't shuffle the array first.

(c) What are two techniques that can be used to reduce the probability of Quicksort taking its worst-case runtime?

3 Sorted Runtimes

We want to sort an array of N **unique** 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 $<= \frac{N}{100}$, we perform insertion sort on them.

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

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

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

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

Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

(d) We use any algorithm to sort the array knowing that:

- There are at most N inversions.

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

- There is exactly 1 inversion.

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

- There are exactly $\frac{N(N-1)}{2}$ inversions.

  Best Case: $\Theta($ $)$, Worst Case: $\Theta($ $)$

4 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