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

Solution:

0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |

(b) Selection sort

Solution:

0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |

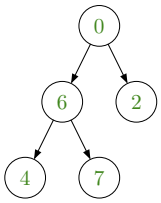
(c) Merge sort

Solution:

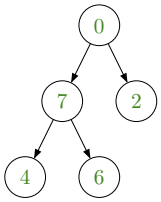
[0 4 2 7 6 1 3 5]
[0 4 2 7] [6 1 3 5]
[0 4] [2 7] [6 1] [3 5]
[0] [4] [2] [7] [6] [1] [3] [5]
[0 4] [2 7] [1 6] [3 5]
[0 2 4 7] [1 3 5 6]
[0 1 2 3 4 5 6 7]

(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`

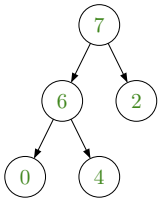
Solution: First, we need to heapify our array. We convert the current array to a max heap:



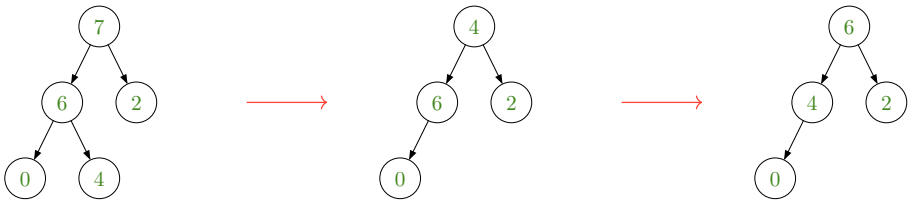
Recall that to heapify our array, we bubble down in reverse level order (bottom to top, right to left). This means we start by bubbling down 4, which in this case gives us the same heap structure. Bubbling down 7, and then 2, leaves the heap unchanged as well. Bubbling down 6 (swapping 6 and 7) then gives us the following:



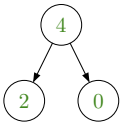
Bubbling down 0 gives us our final heap:



Note that as we heapify, we also modify the underlying array representation as well. This means that our final array looks like [**7, 6, 2, 0, 4**]. We then begin popping off the max value from the heap, placing it at the back of the array. Note that our underlying array representation doesn't consider the popped value as part of the heap any more. We start by popping off 7 and bubbling down:



Our array now looks like this: [**6, 4, 2, 0, 7**], where the bolded section is considered sorted and not part of the heap. We the continue by popping off 6:



and the array looks like **[4, 0, 2, 6, 7]**. We then pop off 4:

and our array looks like **[2, 0, 4, 6, 7]**. In a similar fashion, we pop off 2 and 0 from our heap, resulting in **[0, 2, 4, 6, 7]** and finally our sorted array: **[0, 2, 4, 6, 7]**

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`

Solution:

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

(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.

Solution: One surprisingly clean answer works for every even N: **odd numbers in descending order, followed by even numbers in ascending order.** For $N = 8$:

`7, 5, 3, 1, 2, 4, 6, 8`

The middle item is `1` (the smallest), so the partition is `[] | 1 | [7, 5, 3, 2, 4, 6, 8]`. The middle of that 7-item sub-array is `2`, again the smallest — and the pattern continues: at every level the pivot is the minimum, the partition is fully unbalanced, and the total work is $\Theta(N) + \Theta(N-1) + ... + \Theta(1) = \Theta(N^2)$.

The general construction (which also works for odd N) is: place the smallest value at index $\lfloor \frac{N-1}{2} \rfloor$, then — treating the remaining positions as a list in order — place the next-smallest at **that** list's middle, and so on. For $N = 7$ that yields `6, 4, 2, 1, 3, 5, 7`.

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

Solution:

1. Randomly choose pivots.
2. Shuffle the list before running Quicksort.

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: $\Theta($ $)$

Best Case: $\Theta(N)$, **Worst Case:** $\Theta(N^2)$

Once we have 100 runs of size $\frac{N}{100}$, insertion sort will take best case $\Theta(N)$ and worst case $\Theta(N^2)$ time. Note that the number of merging operations is actually constant (in particular, it takes about 7 splits and merges to get to an array of size $\frac{N}{2^7} = \frac{N}{128}$).

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

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

Solution:

Best Case: Θ(Nlog(N)), **Worst Case:** Θ(Nlog(N))

Doing an extra N work each iteration of quicksort doesn't asymptotically change the best case runtime, since we have to do N work to partition the array. However, it improves the worst case runtime, since we avoid the "bad" case where the pivot is on the extreme end(s) of the partition.

(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: $\Theta($ $)$

Solution:

Best Case: Θ(Nlog(N)), **Worst Case:** Θ(Nlog(N))

While a max-heap is better, we can make do with a min-heap by placing the smallest element at the right end of the list until the list is sorted in **descending order**. Once the list is in descending order, it can be sorted in ascending order with a simple linear time pass.

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

- There are at most N inversions.

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

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

  Recall that insertion sort takes $\Theta(N + K)$ time, where K is the number of inversions. Thus, the optimal sorting algorithm would be insertion sort. If $K < N$, then insertion sort has the best and worst case runtime of $\Theta(N)$.

- There is exactly 1 inversion.

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

Best Case: $\Theta(1)$, **Worst Case:** $\Theta(N)$

First, we notice that if there is only 1 inversion, it could only involve 2 adjacent elements. Intuitively, if two elements that are apart form an inversion, than some element between these two would also form an inversion with one of the elements.

(Optional) A formal argument is as follows: suppose the only inversion involves 2 elements that are not adjacent. Let's call their indices i and j, where $i < j$, and $a[i] > a[j]$ (definition of inversion). Because they are not adjacent, there exist some index k, such that $i < k < j$. In case 1, assume $a[k] > a[i]$. Then it follows that $a[k] > a[i] > a[j]$. Because $k < j$ but $a[k] > a[j]$, (k, j) forms an inversion, so we have a contradiction (we assumed only 1 inversion). In case 2, assume $a[k] < a[i]$, but then we have $k > i$, so (k, i) also form an inversion, which is also a contradiction.

Using this, we can just compare neighboring elements to find that exact inversion, and swap the 2 elements. If the inversion involves the first two elements, constant time is needed. If the inversion involves elements at the end, N time is needed.

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

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

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

  If a list has $\frac{N(N-1)}{2}$ inversions, it means it is sorted in descending order. This is because every possible pair is an inversion (The total number of unordered pairs from N elements is $\frac{N(N-1)}{2}$, which is the number of inversions in a reverse-sorted list. So, it can be sorted in ascending order with a simple linear time pass. We know that reversing any array is a linear time operation, so the optimal runtime of any sorting algorithm is $\Theta(N)$.

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

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.