*Slides by [Kevin Miao](#) ([kevinmiao@berkeley.edu](mailto:kevinmiao@berkeley.edu))*

# CS61BL – Tutoring Section 12

## Sorting

- Review
- Quiz Review (Optional)
- Worksheet (20 min)
- Questions (5 min)

**Friendly reminder: Please participate in order to get marked as ‘present’**

## Resources:

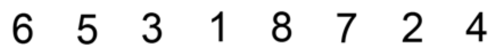
- www.cs61bl.org/su20/resources

## Comparison Based Sorts

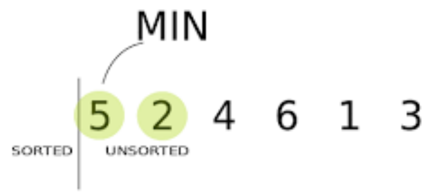
- **Idea:** Comparing items with each other to determine the sorted order.
- **Stability:** What if two items are `equal()` to each other?
  - **Example:** [‘ex’, ‘in’]
  - **Equal()** returns whether the strings are equal in length (in this example only)
  - **A stable sort will always return** [‘ex’, ‘in’]
  - **Unstable sort could return either** [‘ex’, ‘in’] or [‘in’, ‘ex’]

# Comparison Based Sorts

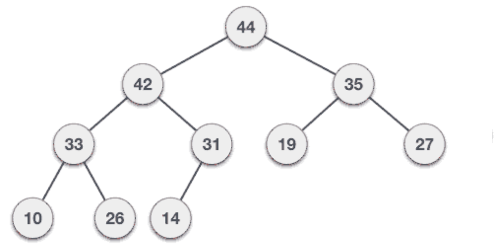
- **Insertion Sort**
  - *Algorithm:* Loop over all items in your list. For each item, **insert** item in the right position in the sorted list
  - **Runtime:**
    - Best Case (Sorted Array): $\Theta(n)$
    - Worst Case (Reversely Sorted Array): $\Theta(n^2)$
  - **Stable:** Yes
  - *Possible adjustments:*
    - ***Tree Sort***
      - *Use a binary search tree to sort the sorted array*
      - *Runtime will become O(nlog(n)) (Unbalanced Tree* $\rightarrow \Theta(n^2)$)

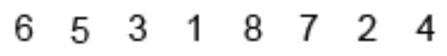


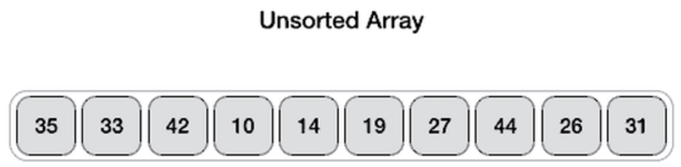
- **Selection Sort**
  - *Algorithm:* For each position, pick the minimum value on the right side.
  - **Runtime** (Best/Worst)**:** $\Theta(n^2)$
  - **Stable:** No



## Comparison Based Sorts



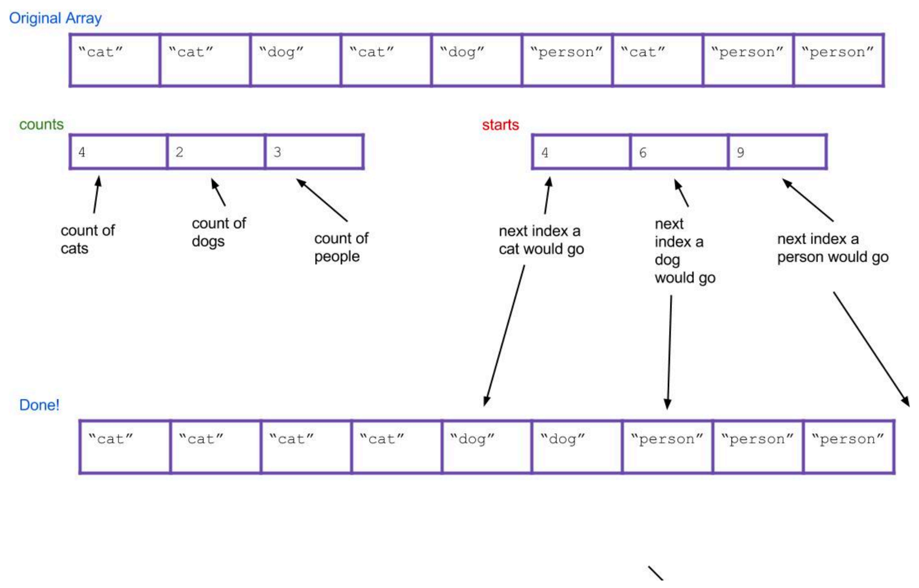




- **Heap sort**
  - *Algorithm:* Basically throw all elements in a heap to be sorted.
  - **Runtime (Best/Worst): Θ(nlog(n))**
  - **Stable:** No
- **Merge Sort**
  - *Algorithm:* Divide by half recursively and then build up again but sorted
  - **Runtime (Best/Worst): Θ(nlog(n))**
  - **Stable:** Yes
- **Quick Sort**
  - *Algorithm: Pick a pivot, sort elements smaller to the left and larger to the right. Recurse down in both halves.*
  - **Runtime**
    - **Best Case (Good pivot):** Θ(log(n))
    - **Worst Case (Bad pivot):** Θ($n^2$)
  - **Stable:** Depends (Three way: stable)

## Counting Based Sorting

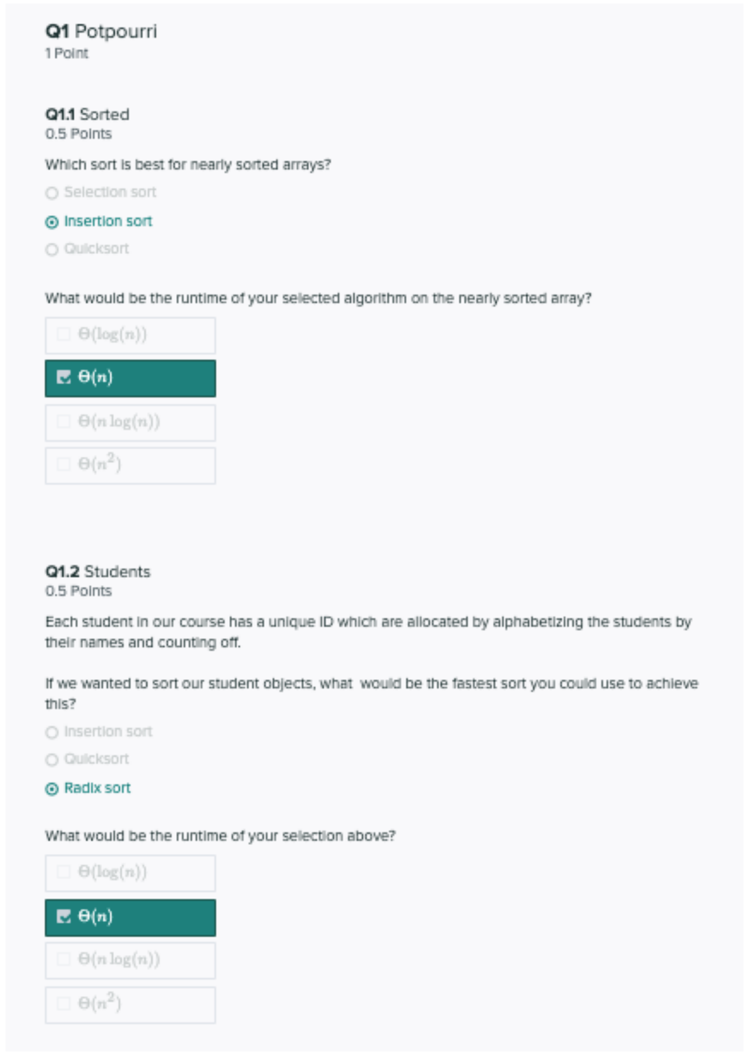
- **The fastest we can do with Comparisons is** O(nLog(n))
- **Counting Sort**
  - Iterate over collection and count occurences
  - Then initialize new array with indices based on the counts
  - **Runtime:** O(N+R)
    - **N =** number of items
    - **R =** number of distinct groups



# Counting Based Sorting

- **Sort based on the most/least significant digit**
- **LSD Radix**
  - Sort starting on the last digit to the most significant digit
  - **Runtime: O(D(N+K))**
    - **D = max number of digits**
    - **N = number of items**
    - **K = number of possible digits**
- **MSD Radix**
  - Sort starting on the last digit to the most significant digit
  - **Runtime Best Case: O(N+K)**
    - 100, 999, 200, 320 (Radix 100 (3 passes) vs Radix 1000 (1 pass))
  - **Runtime Worst Case: O(D(N+K))**
- **O(D(N+K)) ~ O(N) (for some b)**
  - **D = O(b/log(n))**
  - **K = O(N)**

# Quiz Q1



**Q1** Potpourri
1 Point

**Q1.1** Sorted
0.5 Points

Which sort is best for nearly sorted arrays?

- Selection sort
- [✓ Insertion sort]
- Quicksort

What would be the runtime of your selected algorithm on the nearly sorted array?

- $\Theta(\log(n))$
- [✓ $\Theta(n)$]
- $\Theta(n\log(n))$
- $\Theta(n^2)$

**Q1.2** Students
0.5 Points

Each student in our course has a unique ID which are allocated by alphabetizing the students by their names and counting off.

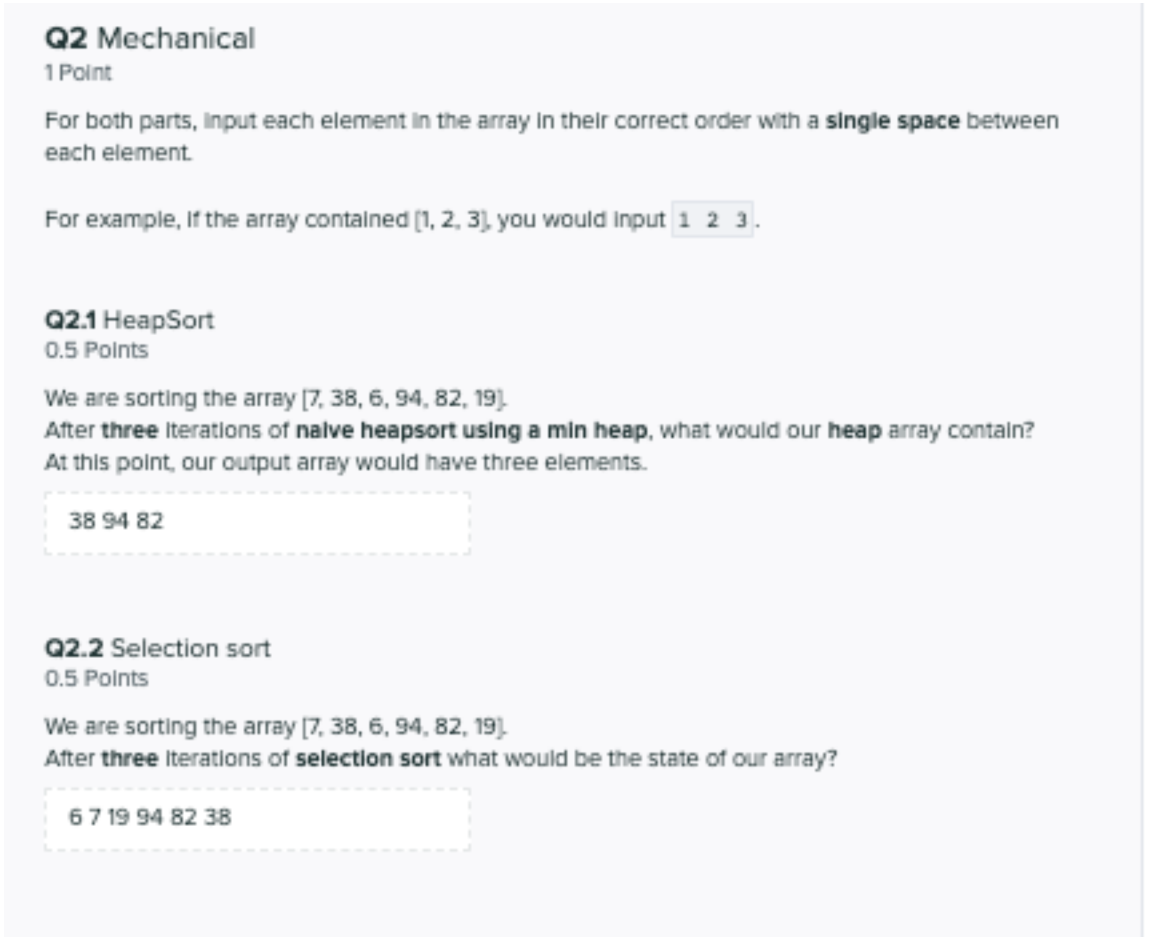
If we wanted to sort our student objects, what would be the fastest sort you could use to achieve this?

- Insertion sort
- Quicksort
- [✓ Radix sort]

What would be the runtime of your selection above?

- $\Theta(\log(n))$
- [✓ $\Theta(n)$]
- $\Theta(n\log(n))$
- $\Theta(n^2)$

## Quiz Q2



**Q2** Mechanical
1 Point

For both parts, input each element in the array in their correct order with a **single space** between each element.

For example, if the array contained [1, 2, 3], you would input `1 2 3`.

**Q2.1** HeapSort
0.5 Points

We are sorting the array [7, 38, 6, 94, 82, 19].
After **three** iterations of **naive heapsort using a min heap**, what would our **heap** array contain?
At this point, our output array would have three elements.

38 94 82

**Q2.2** Selection sort
0.5 Points

We are sorting the array [7, 38, 6, 94, 82, 19].
After **three** iterations of **selection sort** what would be the state of our array?

6 7 19 94 82 38

## 2 Sorting Runtimes

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 | Stability |
|---|---|---|---|
| Selection Sort | $\Theta\left(N^2\right)$ | $\Theta\left(N^2\right)$ | No |
| Insertion Sort | $\Theta\left(N\right)$ | $\Theta\left(N^2\right)$ | Yes |
| Heapsort | $\Theta\left(N\right)$ | $\Theta\left(NlogN\right)$ | No |
| Mergesort | $\Theta\left(NlogN\right)$ | $\Theta\left(NlogN\right)$ | Yes |
| Quicksort | $\Theta\left(NlogN\right)$ | $\Theta\left(N^2\right)$ | Depends |
| Counting Sort | $\Theta\left(N+R\right)$ | $\Theta\left(N+R\right)$ | No |
| LSD Radix Sort | $\Theta\left(L(N+R)\right)$ | $\Theta\left(L(N+R)\right)$ | Yes |
| MSD Radix Sort | $\Theta\left(N+R\right)$ | $\Theta\left(L(N+R)\right)$ | Yes |

## 4 Name That Sort

Below you will find some intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm, but they are in the correct sequence. Identify the algorithm for each problem:
**Input list:** 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1.
```
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
```

2.
```
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
```

3.
```
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
```

4.
```
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
```

5.
```
12, 32, 14, 11, 17, 38, 23, 34
12, 14, 11, 17, 23, 32, 38, 34
```

## 3 You Choose

1. We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

2. Give a 5 element array such that it elicits the worst case runtime for insertion sort.

3. Give some reasons why someone would use merge sort over quicksort.

4. Which sorts never compare the same two elements twice?

5. When might you decide to use radix sort over a comparison sort, and vice versa?

## 1 Step by Step Sorts

Show the steps taken by each sort on the following unordered list of integers (duplicate items are denoted with letters):

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

1. Insertion Sort

2. Selection Sort

3. Merge Sort

4. Heapsort *Note: if both children are equal, sink to the left.*

FAILING DATA STRUCTURES AND ALGORITHMS
ME
ONE MORE MEME ON PROGRAMMERHUMOR
I'm looking forward to next semester

## ALMOST DONE

***Thank you.***

***(If you see me in person, say hi! Let's not act like strangers!)***