Sorting
Sorting

- One of the most fundamental problems in CS
  - Still many questions open!
- Given a list of objects we want to put **in order**
  - Alphabetical, word length, by score in the exam, sickness level,…
  - We assume we have a comparison
- How fast can we do it?
Speed is not the only concern

- How much extra memory do we use?
- Can we handle repeated numbers?
- Is information destroyed?
- Easy to implement?
- What computation model?
- ...
Algorithm 1: selection sort

- Most intuitive algorithm
- Look for smallest value
  - Place it in first position
- Look for second smallest value
  - Place it second
- Etc
**Example**

Look for smallest value
Example

- Look for smallest value
Example

Place first place, now look for second smallest
Example

- Already in second place we do nothing
Example

- Third one to third position
Example

<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>8</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>14</th>
<th>10</th>
<th>22</th>
<th>43</th>
<th>15</th>
<th>72</th>
<th>31</th>
<th>15</th>
<th>42</th>
<th>16</th>
</tr>
</thead>
</table>

- And so on...
Runtime

- Two nested loops
  - Quadratic runtime!
- Let’s prove it!
  - Outer loop n times
  - Inner loop n-j
- Constant number of operations inside

TOTAL?
Big O bound

- Two nested loops
  - Quadratic runtime!
- Let’s prove it!
  - Outer loop at most \( n \) times
  - Inner loop \( n-j \) (at most \( n \) times)
- Constant number of operations inside \( O(N^2) \)
Algorithm 2: insertionSort

- Assume the first i elements have been sorted
- Insert the i+1 in its proper place
- Start with i=1 and stop when i=n
**Algorithm 2: insertionSort**

- The first position is always sorted with itself (progress!)

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>12</td>
<td>11</td>
<td>14</td>
<td>2</td>
<td>22</td>
<td>43</td>
<td>15</td>
<td>72</td>
<td>31</td>
<td>15</td>
<td>42</td>
<td>16</td>
</tr>
</tbody>
</table>
Algorithm 2: insertionSort

- Second position is smaller: we swap
Algorithm 2: insertionSort

- Second position is smaller: we swap
Algorithm 2: insertionSort

- Third position is largest: nothing to do
Algorithm 2: insertionSort

<table>
<thead>
<tr>
<th>5</th>
<th>9</th>
<th>10</th>
<th>8</th>
<th>12</th>
<th>11</th>
<th>14</th>
<th>2</th>
<th>22</th>
<th>43</th>
<th>15</th>
<th>72</th>
<th>31</th>
<th>15</th>
<th>42</th>
<th>16</th>
</tr>
</thead>
</table>

- Fourth position is smaller than third: we swap with previous position
Algorithm 2: insertionSort

- 8 is still too big, we need another swap
Algorithm 2: insertionSort

8 is still finally in place. Let’s look for next number.
Algorithm 2: insertionSort

- Already sorted, nothing to do
Algorithm 2: insertionSort

Next position almost sorted
Algorithm 2: insertionSort

Swap with previous number
Algorithm 2: insertionSort

Done! Is it always this easy?
Algorithm 2: insertionSort

No! The j-th position may travel j-1 positions in worst case
VOID INSERTIONSORT(INT ARR[], INT N) {
    INT I, KEY, J;
    FOR (I = 1; I < N; I++) {
        KEY = ARR[I];
        J = I - 1;

        /* MOVE ELEMENTS OF ARR[0..I-1], THAT ARE GREATER THAN KEY, TO ONE POSITION AHEAD OF THEIR CURRENT POSITION */
        WHILE (J >= 0 && ARR[J] > KEY) {
            ARR[J + 1] = ARR[J];
            J = J - 1;
        }
        ARR[J + 1] = KEY;
    }
}
Introducing Mergesort

- Partition the array of \( n \) elements into two arrays of size \( n/2 \)
  - Recursively sort them
  - Merge the two solutions into one
Combining two sorted lists

- Keep 3 pointers
- Current positions in the three arrays
- Start at rightmost positions
Combining two sorted lists

- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest
Combining two sorted lists

While arrays have not been fully explored
- Add smallest to Result.
- Advance Result and the array containing smallest
Combining two sorted lists

- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest
Combining two sorted lists

- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest
Combining two sorted lists

- While arrays have not been fully explored
- Add smallest to Result.
- Advance Result and the array containing smallest
Combining two sorted lists

- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest
Combining two sorted lists

- While arrays have not been fully explored
  - Add smallest to Result.
  - Advance Result and the array containing smallest
| 99 | 6  | 86 | 15 | 58 | 35 | 86 | 4  | 0  |

Merge Sort Full Example
### Merge Sort Full Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Merge Sort Full Example

<table>
<thead>
<tr>
<th>99</th>
<th>6</th>
<th>86</th>
<th>15</th>
<th>58</th>
<th>35</th>
<th>86</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td></td>
<td></td>
<td>58</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>6</td>
<td>86</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>86</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>86</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>
Merge Sort Full Example

99  6  86  15  58  35  86  4  0

99  6  86  15
58  35  86  4  0

99  6  86  15
58  35
86  4  0

99  6  86  15  58  35  86  4  0

4  0
4  0
Building pieces upwards
Building pieces upwards
Building pieces upwards
Building pieces upwards
Building pieces upwards

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>15</td>
<td>35</td>
<td>58</td>
<td>86</td>
<td>86</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>86</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>6</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td>6</td>
<td>86</td>
<td>15</td>
<td>58</td>
<td>35</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Runtime?

Mergesort time for $n$ elements:
Quicksort

- Very similar in than MergeSort
  - **Divide and Conquer** strategy
  - Worse from a theoretical standpoint
  - Faster in practice
  - Does not need extra space
Quicksort

- Pick an element of the array (the pivot)
- Split array into smaller and larger than pivot

NAIVE CHOICE: $A[0]$
Quicksort

- Pick an element of the array (the **pivot**)
- Split array into smaller and larger than pivot
- Recursively sort both arrays

**NAIVE CHOICE: A[0]**

```
6 5 9 12 3 4
```

```
5 3 4 6 9 1 2
```
Execution example

- Pick an element of the array (the **pivot**)
- Split array into smaller and larger than pivot
- Recursively sort both arrays
Execution example

- Pick an element of the array (the **pivot**)
- Split array into smaller and larger than pivot
- Recursively sort both arrays
Runtime?

- Worst case?
  - Already sorted input!
  - $O(n^2)$ runtime
- Easy solution?
  - Pick a pivot \textbf{at random}
  - Still $O(n^2)$ worst case runtime
Part 2: Sorting in Linear Time
CountingSort

- Let’s spice things up
- Can we do it in a different way?  
  **Not** based on usual comparison
- Assume input has limited range  
  $0 < A[i] < k$ for all values of $i$
- Can you make an algorithm that uses this property?
Countingsort: phase 1

- Scan array, find largest value $k$
- Make array of size $k+1$, all entries zero
- Scan array again, each time increasing count

<table>
<thead>
<tr>
<th>9</th>
<th>5</th>
<th>10</th>
<th>8</th>
<th>3</th>
<th>6</th>
<th>5</th>
<th>2</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>2</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
</table>

0 0 0 0 0 0 0 0 0 0 0

10 IN THIS EXAMPLE

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
### Countingsort: phase 1

- Scan array, find largest value $k$
- Make array of size $k+1$, all entries zero
- Scan array again, each time increasing count

---

**Example:**

```
9  5 10  8  3  6  5  2  5  2  1  3  5  2  6  5
```

```
```

```
0  1  3  2  0  5  2  0  1  1  1
```

```
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10]
```

10 IN THIS EXAMPLE
Countingsort: phase 2

- Scan *multiplicity* array
- For each index I add $A[i]$ many copies into the solution

NOTHING TO DO
Countingsort: phase 2

- Scan **multiplicity** array
- For each index I add \( A[i] \) many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index $i$ add $A[i]$ many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index I add A[i] many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index I add A[i] many copies into the solution
Countingsort: phase 2

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
</tr>
</tbody>
</table>

For each index I add A[i] many copies into the solution.

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index I add A[i] many copies into the solution
Countingsort: phase 2

- Scan **multiplicity** array
- For each index I add A[i] many copies into the solution
Countingsort: phase 2

- Scan **multiplicity** array
- For each index I add \( A[i] \) many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index I add $A[i]$ many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index I add A[i] many copies into the solution
Countingsort: phase 2

- Scan multiplicity array
- For each index I add $A[i]$ many copies into the solution
Runtime?

Phase 1:

- Scan input array, find max $O(N)$
- Create array of size $k$ $O(1)$
- Make all entries zero $O(K)$
- Scan input array, increase count at each step $O(N)$
### Runtime?

**Phase 2:**

For all entries of count array **K ITERATIONS**

\[
\begin{array}{cccccccccccccc}
1 & 2 & 2 & 2 & 3 & 3 & 5 & 5 & 5 & 5 & 5 & 6 & 6 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 1 & 3 & 2 & 0 & 5 & 2 & 0 & 1 & 1 & 1 \\
\end{array}
\]

**TOTAL TIME O(N+K)**
n: number of elements

n = 6
RADIX SORT

\( n \): number of elements

\( l \): (max) length of each element

\( r \): radix (number of symbols available at each digit) e.g., binary, decimal, hex

\[ r = 9 \quad (0 \ldots 8) \]

\[ n = 6 \]

\[ 1073 \quad 284 \quad 5 \quad 8261 \quad 2714 \quad 382 \]

\[ 1073 \quad 0284 \quad 0005 \quad 8261 \quad 2714 \quad 0382 \]
RADIX SORT

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5

n = 7
l = 3
r = 10
RADIX SORT uses the least significant digit. Usually:

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
RADIx SORT uses the least significant digit.

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5
7 2 0
3 5 5
8 3 9
3 5 5
RADIX SORT uses the least significant digit.

don't cross the streams

3 2 9 7 2 0
4 5 7 3 5 5
6 5 7 4 3 6
8 3 9 4 5 7
4 3 6 6 5 7
7 2 0 3 2 9
3 5 5 8 3 9
RADIX SORT uses the least significant digit.

use stable counting sort

\[ \Theta(n+r) \]
WAIT A SECOND: WAS THIS STABLE?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
<th>[10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
RADIX SORT uses the least significant digit.

iteration 2

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5

7 2 0
3 5 5
4 3 6
4 5 7
8 3 9
6 5 7
3 5 5
8 3 9
6 5 7
Radix sort uses the least significant digit.

<table>
<thead>
<tr>
<th>Radix Sort</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 2 9</td>
<td>7 2 0</td>
</tr>
<tr>
<td>4 5 7</td>
<td>3 5 5</td>
</tr>
<tr>
<td>6 5 7</td>
<td>4 3 6</td>
</tr>
<tr>
<td>8 3 9</td>
<td>4 5 7</td>
</tr>
<tr>
<td>4 3 6</td>
<td>6 5 7</td>
</tr>
<tr>
<td>7 2 0</td>
<td>3 2 9</td>
</tr>
<tr>
<td>3 5 5</td>
<td>8 3 9</td>
</tr>
<tr>
<td></td>
<td>7 2 0</td>
</tr>
<tr>
<td></td>
<td>3 2 9</td>
</tr>
<tr>
<td></td>
<td>3 5 5</td>
</tr>
<tr>
<td></td>
<td>4 3 6</td>
</tr>
<tr>
<td></td>
<td>4 5 7</td>
</tr>
<tr>
<td></td>
<td>7 2 0</td>
</tr>
<tr>
<td></td>
<td>8 3 9</td>
</tr>
<tr>
<td></td>
<td>6 5 7</td>
</tr>
<tr>
<td>RADIX SORT</td>
<td>uses the <strong>least</strong> significant digit.</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td><strong>Time = ?</strong></td>
<td></td>
</tr>
<tr>
<td>3 2 9</td>
<td>7 2 0</td>
</tr>
<tr>
<td>4 5 7</td>
<td>3 5 5</td>
</tr>
<tr>
<td>6 5 7</td>
<td>4 3 6</td>
</tr>
<tr>
<td>8 3 9</td>
<td>4 5 7</td>
</tr>
<tr>
<td>4 3 6</td>
<td>6 5 7</td>
</tr>
<tr>
<td>7 2 0</td>
<td>3 2 9</td>
</tr>
<tr>
<td>3 5 5</td>
<td>8 3 9</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**RADIX SORT**

\[ \Theta(l \cdot (n+r)) \]

| 3 2 9 | 7 2 0 | 7 2 0 | 3 2 9 |
| 4 5 7 | 3 5 5 | 3 2 9 | 3 5 5 |
| 6 5 7 | 4 3 6 | 4 3 6 | 4 3 6 |
| 8 3 9 | 4 5 7 | 8 3 9 | 4 5 7 |
| 4 3 6 | 6 5 7 | 3 5 5 | 6 5 7 |
| 7 2 0 | 3 2 9 | 4 5 7 | 7 2 0 |
| 3 5 5 | 8 3 9 | 6 5 7 | 8 3 9 |

**uses the least significant digit.**