Searching and Sorting
Part One
Apply to Section Lead!

Don’t just take my word for it...
Recap from Last Time
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.

- For example:
  - A square of side length $r$ has area $O(r^2)$.
  - A circle of radius $r$ has area $O(r^2)$.

Doubling $r$ increases area $4 \times$. Tripling $r$ increases area $9 \times$. 

Doubling $r$ increases area $4 \times$. Tripling $r$ increases area $9 \times$. 
Big-O Notation

- **Big-O notation** is a way of quantifying the rate at which some quantity grows.

- For example:
  - A square of side length $r$ has area $O(r^2)$.
  - A circle of radius $r$ has area $O(r^2)$.
  - A sphere of radius $r$ has surface area $O(r^2)$.
  - A cube of side length $r$ has surface area $O(r^2)$.
  - A cube of side length $r$ has volume $O(r^3)$.
  - A sphere of radius $r$ has volume $O(r^3)$.

Doubling $r$ increases area 4×.
Tripling $r$ increases area 9×.

This just says that these quantities grow at the same relative rates. It does not say that they’re equal!
Assume any individual statement takes one unit of time to execute. If the input Vector has \( n \) elements, how many time units will this code take to run?

```cpp
double averageOf(const Vector<int>& vec) {
    double total = 0.0;

    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }

    return total / vec.size();
}
```
Assume any individual statement takes one unit of time to execute. If the input Vector has $n$ elements, how many time units will this code take to run?
One possible answer: $3n + 4$. 

```cpp
double averageOf(const Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
```
double averageOf(const Vector<int>& vec) {
    double total = 0.0;
    for (int i = 0; i < vec.size(); i++) {
        total += vec[i];
    }
    return total / vec.size();
}
void printStars(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << '*' << endl;
        }
    }
}
void hmmThatssStrange(int n) {
    cout << "Mirth and Whimsy" << endl;
}

Work Done: $O(1)$. 
New Stuff!
Sorting Algorithms
What is sorting?
One style of “sorting,” but not the one we’re thinking about...
<table>
<thead>
<tr>
<th>Time</th>
<th>Auto</th>
<th>Athlete</th>
<th>Nationality</th>
<th>Date</th>
<th>Venue</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:36.8</td>
<td>Maria Gomers</td>
<td>Netherlands</td>
<td>14 June 1969[8]</td>
<td>Leicester</td>
<td></td>
</tr>
<tr>
<td>4:35.3</td>
<td>Ellen Tittel</td>
<td>West Germany</td>
<td>20 August 1971[8]</td>
<td>Sittard</td>
<td></td>
</tr>
<tr>
<td>4:29.5</td>
<td>Paola Pigni</td>
<td>Italy</td>
<td>8 August 1973[8]</td>
<td>Viareggio</td>
<td></td>
</tr>
<tr>
<td>4:17.44</td>
<td>Maricica Puică</td>
<td>Romania</td>
<td>9 September 1982[8]</td>
<td>Rieti</td>
<td></td>
</tr>
<tr>
<td>4:15.61</td>
<td>Paula Ivan</td>
<td>Romania</td>
<td>10 July 1989[8]</td>
<td>Nice</td>
<td></td>
</tr>
<tr>
<td>4:12.56</td>
<td>Svetlana Masterkova</td>
<td>Russia</td>
<td>14 August 1996[8]</td>
<td>Zürich</td>
<td></td>
</tr>
<tr>
<td>4:12.33</td>
<td>Sifan Hassan</td>
<td>Netherlands</td>
<td>12 July 2019</td>
<td>Monaco</td>
<td></td>
</tr>
</tbody>
</table>

**Problem:** Given a list of data points, sort those data points into ascending / descending order by some quantity.
Suppose we want to rearrange a sequence to put elements into ascending order. What are some strategies we could use? How do those strategies compare? Is there a “best” strategy?
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort

The smallest element should go in front.
An Initial Idea: **Selection Sort**
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort

This element is in the right place now.

The remaining elements are in no particular order.
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*
An Initial Idea: *Selection Sort*

The smallest element of the remaining elements goes at the front of the remaining elements.
An Initial Idea: *Selection Sort*
An Initial Idea: *Selection Sort*
An Initial Idea: *Selection Sort*

These elements are in the right place now.

The remaining elements are in no particular order.
An Initial Idea: Selection Sort
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*

The smallest of these remaining elements goes at the front of the remaining elements.
An Initial Idea: *Selection Sort*
An Initial Idea: *Selection Sort*

These elements are in the right place now.

The remaining elements are in no particular order.
An Initial Idea: \textit{Selection Sort}

\begin{center}
\begin{tikzpicture}
    \foreach \x in {1,2,4,7,6} {
        \node[rectangle, fill=blue!20, minimum width=2cm] (\x) at (\x,0) {}; \node[below] at (\x,0) {\x};
    }
\end{tikzpicture}
\end{center}
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*

The smallest of these elements needs to go at the front of this group of elements.
An Initial Idea: **Selection Sort**
An Initial Idea: Selection Sort
An Initial Idea: **Selection Sort**

These elements are in the right place now.

The remaining elements are in no particular order.
An Initial Idea: **Selection Sort**

![Diagram of Selection Sort](image)
An Initial Idea: *Selection Sort*
An Initial Idea: Selection Sort

The smallest element from this group needs to go at the front of the group.
An Initial Idea: Selection Sort
An Initial Idea: *Selection Sort*

These elements are in the right place now.
Selection Sort

- Find the smallest element and move it to the first position.
- Find the smallest element of what’s left and move it to the second position.
- Find the smallest element of what’s left and move it to the third position.
- Find the smallest element of what’s left and move it to the fourth position.
- (etc.)
/**
 * Sorts the specified vector using the selection sort algorithm.
 * @param elems The elements to sort.
 */
void selectionSort(Vector<int>& elems) {
    for (int index = 0; index < elems.size(); index++) {
        int smallestIndex = indexOfSmallest(elems, index);
        swap(elems[index], elems[smallestIndex]);
    }
}

/**
 * Given a vector and a starting point, returns the index of the smallest
 * element in that vector at or after the starting point
 * @param elems The elements in question.
 * @param startPoint The starting index in the vector.
 * @return The index of the smallest element at or after that point
 */
int indexOfSmallest(const Vector<int>& elems, int startPoint) {
    int smallestIndex = startPoint;
    for (int i = startPoint + 1; i < elems.size(); i++) {
        if (elems[i] < elems[smallestIndex]) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
Analyzing Selection Sort

• How much work do we do for selection sort?

• To find the smallest value, we need to look at all $n$ array elements.

• To find the second-smallest value, we need to look at $n - 1$ array elements.

• To find the third-smallest value, we need to look at $n - 2$ array elements.

• Work is $n + (n - 1) + (n - 2) + \ldots + 1$. 
\[ n + (n-1) + \ldots + 2 + 1 = \frac{n(n+1)}{2} \]
The Complexity of Selection Sort

\[ O(n (n + 1) / 2) \]
\[ = O(n (n + 1)) \]
\[ = O(n^2 + n) \]
\[ = O(n^2) \]

So selection sort runs in time \( O(n^2) \).
The area of a 2D figure that fits snugly in an $n \times n$ box is $O(n^2)$. 

$$n + (n-1) + \ldots + 2 + 1$$
Our theory predicts that the runtime of selection sort is $O(n^2)$.

Does that match what we see in practice?

What should we expect to see when we look at a runtime plot?
Another Sorting Algorithm
Our Next Idea: **Insertion Sort**
Our Next Idea: **Insertion Sort**
Our Next Idea: *Insertion Sort*

This sequence in blue, taken in isolation, is in sorted order.

This sequence in gray is in no particular order.
Our Next Idea: *Insertion Sort*

```
7

4

2

1

6
```

*Insert* this element into the blue sequence, making the blue sequence one element longer.
Our Next Idea: Insertion Sort
Our Next Idea: *Insertion Sort*
Our Next Idea: *Insertion Sort*

This sequence in blue, taken in isolation, is in sorted order.

This sequence in gray is in no particular order.
Our Next Idea: **Insertion Sort**

Insert this element into the blue sequence, making the blue sequence one element longer.
Our Next Idea: **Insertion Sort**

4 7 2 1 6
Our Next Idea: **Insertion Sort**
Our Next Idea: *Insertion Sort*
Our Next Idea: *Insertion Sort*

\[ \begin{array}{cccccc}
2 & 4 & 7 & 1 & 6 \\
\end{array} \]
Our Next Idea: **Insertion Sort**

This sequence in blue, taken in isolation, is in sorted order.

This sequence in gray is in no particular order.
Our Next Idea: *Insertion Sort*

Insert this element into the blue sequence, making the blue sequence one element longer.
Our Next Idea: **Insertion Sort**
Our Next Idea: **Insertion Sort**
Our Next Idea: Insertion Sort
Our Next Idea: *Insertion Sort*
Our Next Idea: **Insertion Sort**
Our Next Idea: *Insertion Sort*
Our Next Idea: **Insertion Sort**

1 2 4 7

This sequence in blue, taken in isolation, is in sorted order.

6

This sequence in gray is in no particular order.
Our Next Idea: Insertion Sort

Insert this element into the blue sequence, making the blue sequence one element longer.
Our Next Idea: **Insertion Sort**

1 2 4 7 6
Our Next Idea: *Insertion Sort*
Our Next Idea: *Insertion Sort*

This sequence in blue, taken in isolation, is in sorted order.

There are no more gray elements, so the sequence is sorted!
Insertion Sort

• Repeatedly *insert* an element into a sorted sequence at the front of the array.

• To *insert* an element, swap it backwards until either
  • (1) it’s bigger than the element before it, or
  • (2) it’s at the front of the array.
/**
 * Sorts the specified vector using insertion sort.
 * @param v The vector to sort.
 */

void insertionSort(Vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        /* Scan backwards until either (1) there is no
         * preceding element or the preceding element is
         * no bigger than us.
         */
        for (int j = i - 1; j >= 0; j--) {
            if (v[j] <= v[j + 1]) break;
            /* Swap this element back one step. */
            swap(v[j], v[j + 1]);
        }
    }
}
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

1 2 4 6 7
How Fast is Insertion Sort?

Work done: $O(n)$
How Fast is Insertion Sort?

7  6  4  2  1
How Fast is Insertion Sort?

1. 7
2. 6
3. 4
4. 2
5. 1
How Fast is Insertion Sort?

7 6 4 2 1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

6

7

4

2

1
How Fast is Insertion Sort?

6

7

4

2

1
How Fast is Insertion Sort?

6
7
4
2
1
How Fast is Insertion Sort?

6
7
4
2
1
How Fast is Insertion Sort?

6
4
7
2
1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>7</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>
How Fast is Insertion Sort?

4  6  7  2  1
How Fast is Insertion Sort?

4 6 7 2 1
How Fast is Insertion Sort?

4 6 7 2 1
How Fast is Insertion Sort?

4 6 2 7 1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

4  2  6  7  1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2 4 6 7 1
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2 4 6 7 1
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

2  4  6  1  7
How Fast is Insertion Sort?

2 4 1 6 7
How Fast is Insertion Sort?

2 4 1 6 7
How Fast is Insertion Sort?

2 1 4 6 7
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?
How Fast is Insertion Sort?

Work Done: \[ 1 + 2 + 3 + \ldots + n \]

\[ = O(n^2) \]
Three Analyses

- **Worst-Case Analysis**
  - What's the *worst* possible runtime for the algorithm?
  - Useful for “sleeping well at night.”

- **Best-Case Analysis**
  - What's the *best* possible runtime for the algorithm?
  - Useful to see if the algorithm performs well in some cases.

- **Average-Case Analysis**
  - What's the *average* runtime for the algorithm?
  - Far beyond the scope of this class; take CS109, CS161, or CS265 for more information!
The Complexity of Insertion Sort

- In the best case (the array is sorted), insertion takes time $O(n)$.
- In the worst case (the array is reverse-sorted), insertion sort takes time $O(n^2)$.
- **Fun fact:** Insertion sorting an array of random values takes, on average, $O(n^2)$ time.
  - Curious why? Come talk to me after class!
How do selection sort and insertion sort compare against one another?
Building a Better Sorting Algorithm
Thinking About $\mathcal{O}(n^2)$

![Diagram showing the relationship between T(n), T(\frac{1}{2}n), and elements in a sequence.](image-url)
Thinking About $O(n^2)$
Thinking About $O(n^2)$

\[
2 \cdot \frac{1}{4} T(n) = \frac{1}{2} T(n)
\]
With an $O(n^2)$-time sorting algorithm, it takes twice as long to sort the whole array as it does to split the array in half and sort each half.

Can we exploit this?
The Key Insight: *Merge*
The Key Insight: *Merge*
The Key Insight: **Merge**
The Key Insight: Merge
The Key Insight: Merge
The Key Insight: Merge
The Key Insight: **Merge**
The Key Insight: \textit{Merge}
The Key Insight: *Merge*
The Key Insight: Merge
The Key Insight: Merge
The Key Insight: Merge
The Key Insight: **Merge**
The Key Insight: **Merge**
The Key Insight: *Merge*

Each step makes a single comparison and reduces the number of elements by one.

If there are $n$ total elements, this algorithm runs in time $O(n)$. 

1 2 3 4 5 6 7 8 9 10
The Key Insight: *Merge*

- The *merge* algorithm takes in two sorted lists and combines them into a single sorted list.
- While both lists are nonempty, compare their first elements. Remove the smaller element and append it to the output.
- Once one list is empty, add all elements from the other list to the output.
- *Note:* Removing the first element of a Vector takes time $O(n)$, so you can’t implement this algorithm by repeatedly removing the first element of one of the Vectors. See if you can find another approach for doing this.
1. Split the input in half.

“Split Sort”
“Split Sort”

1. Split the input in half.
2. Insertion sort each half.
1. Split the input in half.
2. Insertion sort each half.
3. Merge the halves back together.

“Split Sort”
"Split Sort"

```cpp
void splitSort(Vector<int>& v) {
    /* Split the vector in half */
    int half = v.size() / 2;
    Vector<int> left  = v.subList(0, half);
    Vector<int> right = v.subList(half);

    /* Sort each half. */
    insertionSort(left);
    insertionSort(right);

    /* Merge them back together. */
    merge(left, right, v);
}
```

Prediction: This should still take time $O(n^2)$, but be about twice as fast as insertion sort.
Next Time

- **Mergesort**
  - A beautiful, elegant sorting algorithm.
- **Analyzing Mergesort**
  - An unusual runtime analysis.
- **Hybrid Sorting Algorithms**
  - Improving on mergesort.
- **Binary Search**
  - Finding things fast!