Collections, Part Two
Today

- Short Review From Last Week
- Vector
- Grid
- Vector Performance
- Containers: Common mistakes
From Last Week...
A **recursive solution** is a solution that is defined in terms of itself.
Recursion: Fibonacci Numbers

• Fibonacci Numbers
  • 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
  • Defined recursively:

\[
\text{fib}(n) = \begin{cases} 
  n & \text{if } n = 0 \text{ or } 1 \\
  \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise}
\end{cases}
\]
Another View of Factorials

\[ n! = \begin{cases} 
1 & \text{if } n = 0 \\
 n \times (n - 1)! & \text{otherwise} 
\end{cases} \]

```cpp
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```
The `TokenScanner` class can be used to break apart a string into smaller pieces.

Construct a `TokenScanner` to piece apart a string as follows:

```java
TokenScanner scanner = new TokenScanner(str);
```

Configure options (ignore comments, ignore spaces, add operators, etc.)

Use the following loop to read tokens one at a time:

```java
while (scanner.hasMoreTokens()) {
    string token = scanner.nextToken();
    /* ... process token ... */
}
```

Check the documentation for more details; there are some really cool tricks you can do with the `TokenScanner`!
A **Stack** is a data structure representing a stack of things.

Objects can be **pushed** on top of the stack or **popped** from the top of the stack.

Only the top of the stack can be accessed; no other objects in the stack are visible.

Example: Function calls
Vector
Vector

- The **Vector** is a collection class representing a list of things.
  - Similar to Java's **ArrayList** type.
- Probably the single most commonly used collection type in all programming.
Example: Cell Tower Purchasing
Buying Cell Towers

137
42
95
272
52
Buying Cell Towers

137 42 95 272 52
Buying Cell Towers

14  22  13  25  30  11  9
Buying Cell Towers

14  22  13  25  30  11  9
Buying Cell Towers

- Given the populations of each city, what is the largest number of people you can provide service to given that no two cell towers are adjacent?

- Proposed Algorithm: Iteratively pick the "largest population" cell towers from the set of remaining towers we can select
  - Problems with this algorithm?
Proposed Algorithm: Problem

99 100 99
Proposed Algorithm: Problem

99 100 99
Buying Cell Towers

- Our proposed algorithm won't always give us the correct answer!
- Correct algorithm is best explained pictorially...
Maximize what's left in here.
Maximize what's left in here.
Maximize what's left in here.
Maximize what's left in here.
Cell-towers Pseudocode
(On Board)
cell-towers.cpp
(On Computer)
How the Recursion Works
How the Recursion Works

14  22  13

22  13
How the Recursion Works
How the Recursion Works

Best is 13
How the Recursion Works

Best is 13
How the Recursion Works

Best is 13

Best is 0
How the Recursion Works

Best is 22

Best is 13

Best is 0
How the Recursion Works

Best is 13

Best is 22

Best is 14

Best is 0
How the Recursion Works

Best is 13

13

Best is 22

22

Best is 13

13

Best is 0

0

14

22

13

+14

+22

Best is 13

13

Best is 22

22

+14
How the Recursion Works

Best is 27

Best is 22

Best is 13

Best is 22

Best is 13

Best is 0

Best is 13

Best is 13

Best is 13

Best is 13
How the Recursion Works
Pass-by-Reference and Objects

- Recall: In C++, *all* parameters are passed by value unless specified otherwise.
- Passing by value makes a copy of the parameter.
- When using container types (*Stack*, *Vector*, etc.) it is often useful to use pass-by-reference for efficiency reasons.
  - Takes a *long* time to make a copy of a large collection!
  - Let's see what happens when we do this for `cell-towers.cpp`!
Vector or Stack?

- Any Stack can be replaced with a Vector with which we only add and remove from the back.
  - So why should we ever use a Stack?
  - Hint: It's not for performance reasons
Vector or Stack?

- **Reason 1:** It makes your code easier to read
  - Someone reading your code knows that you are only going to read and add to the top of the Stack.

- **Reason 2:** It protects you from making mistakes
  - If you use a Vector, you might accidentally add/read/remove from the middle instead of the end.

- **Summary:** Use Stack when the algorithm lets you, otherwise use Vector.
Grid
Two-Dimensional Data

- The **Grid** type can be used to store two-dimensional data.
  - e.g. matrices, scrabble boards, etc.
- Can construct a grid of a certain size by writing
  
  ```
  Grid<Type> g(numRows, numCols);
  ```
- Can access individual elements by writing
  
  ```
  g[rows][cols]
  ```
Stanford is not as safe as it seems...
Velociraptors Spotted on Campus!

- Everyone knows how dangerous velociraptors are, but not everyone knows how to survive an attack.

IF MONEY DOESN'T GROW ON TREES
WHY DO BANKS HAVE BRANCHES?
Good News

- Luckily, velociraptors are constrained to exist on cells of a Grid!
Good News

- Also, velociraptors can only move in the 8 cardinal and ordinal directions.
A natural question arises – given a grid of locations of velociraptors, is there a position on the grid that is safe?
Good News

- A natural question arises – given a grid of locations of velociraptors, is there a position on the grid that is safe?
- Represent the grid with...a `Grid<bool>` where `true` indicates that a velociraptor is there.
A natural question arises – given a grid of locations of velociraptors, is there a position on the grid that is safe?

Represent the grid with... a `Grid<bool>` where `true` indicates that a velociraptor is there.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

Good News
raptor-defense.cpp

(Computer)
**Grid or Vector<Vector>?**

- Any **Grid** can be replaced with a **Vector<Vector>** in which we make the length of the “inner vectors” equal.
  - So why should we ever use a **Grid**?
- For reasons similar to the “**Vector or Stack**” decision:
  - Easier to read.
  - Less likely to make a mistake.
Vector Performance

- Where you add/remove from a Vector can have a huge performance impact
Vector Performance?

```cpp
Vector<int> myVector;
for (int i = 0; i < 1000; i++)
    myVector[i] = 0;

vs

Vector<int> myVector;
for (int i = 0; i < 1000; i++)
    myVector.insert(0,i);
```
Vector Performance

• Why was this?
  • When you remove (or insert) at the beginning of a vector, all the other elements in the vector must be shifted over
  • This can have big performance consequences
    - We will learn about other data structures that solve this

• It turns out, reading from a vector takes the same amount of time no matter where you read from
  • We'll learn why later in the quarter
Collections: Common Pitfall 1

Vector numbers;
Collections: Common Pitfall 1

`Vector<int> numbers;`
Collections: Common Pitfall 2

```
Vector<Vector<int>> numbers;
```
Collections: Common Pitfall 2

Vector<Vector<int> > numbers;
void myFunction(Grid<bool> bigGrid);
void myFunction(Grid<bool> &bigGrid);
Next Time

- **Map**
  - A collection for storing associations between elements.

- **Set**
  - A collection for storing an unordered group of elements.

- **Lexicon**
  - A special kind of **Set**.