We've now seen three *data structures*:

- Arrays, ArrayList, HashMap
- More such data structures to come in CS 261
  - Sets, trees, stacks, queues...

One of the key operations in all of CS is to find or *search* for things within collections of data:
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion

**Note:** These algorithms can be implemented over arrays or ArrayLists. We use Arrays for our lectures due to simplified syntax.
Given an array (or `ArrayList`) of elements and a search-key `key`,

- Returns the index of the 1st occurrence of the `key`, or `-1` if not found

Let's define the method called `linearSearch()`

Follow-up Question:

If there are $N$ items in my list, how many elements does `linearSearch()` need to compare?
Time Complexity: Linear Search

- Algorithm:

```java
public class Searcher {
    private int[] list;

    public int linearSearch(int key) {
        // loop through the list from beginning to end
        for (int i = 0; i < list.length; i++) {
            if (key == list[i]) {
                // found the key! return the index and stop searching!
                return i;
            }
        }
        // didn't find the key! return -1
        return -1;
    }
}
```

How many comparisons does `linearSearch()` take?

Let's test out some cases...
Yardstick for Comparing Algorithms

- How do we characterize the goodness of an algorithm?
  - Accuracy of results (*e.g.*, finding a square root with Newton's method)
  - The amount of space required to run algorithm (*space complexity*)
  - The time taken to produce the correct result (*time complexity*)
    - We'll focus mainly on time complexity in this course

- **Important:** The *time complexity* of an algorithm is the number of comparisons required in terms of the problem size $N$
Time Complexity Cases

- We're concerned with the following cases

  • Best Case
    - Scenario in which the algorithm requires the fewest number of comparisons to complete running an algorithm.

  • Worst Case
    - Scenario in which the algorithm requires the most number of comparisons to complete running an algorithm.

  • Average Case
    - The number of comparisons taken in the most likely scenario.

- For Linear Search:

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparisons</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Outline

- Motivation for Search and Sort
- Linear Search
- **Binary Search**
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion
There's a Smarter Way to Search

- Let's play the Secret Number game
  - Pick a number between 1 and 100...
Does List Order Matter?

- Given a *sorted list* of integers and a key, determine if key is in the list

```c
int[] list = {3, 4, 6, 7, 9, 10, 12, 13, 15};
```

- Is there a *better* way to search?
  - *Better* means more time-efficient (fewer comparisons in worst case)
    - Start from beginning? Start from back? Doesn't seem to do much
  - Think: Lots of things are sorted in life... how does that help you search?
The idea behind binary search:
- How do you make use of the ordering to help you find something faster?
- Why is it called binary?

Think phone book: Search for a name, Kate

Flip to the middle page, get the name from page.
1) Is it Kate?
   (a) If so, you're done (found Kate)!
2) If not, does Kate precede the name on the page?
   (a) Tear off last half of phonebook
3) Otherwise,
   (a) Tear off first half of phonebook
4) Repeat until you run find Kate, or run out of pages to tear
Binary Search Examples

- Remember, the list must already be in *ascending order* for binary search to work.

```c
int[] list = {3, 4, 6, 7, 9, 10, 12, 13, 15};
```
Step 1: Initialize the positions of the "book ends"

Search for key = 10

Comparisons: 0
Search for key = 10

- Step 2: Determine the position of the **mid-point** between book ends
  - Compare this element with the key!

```
list

<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>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>15</td>
</tr>
</tbody>
</table>

length: 9

left (0) = (left+right)/2  mid (4)  right (8)  

Comparisons: 1
```
Step 3: In this case, key > list[mid]

- If key is indeed in the list, it *must* be in the right half

Comparisons: 1
Step 3: In this case, key > list[mid]

- If key is indeed in the list, it **must** be in the right half
- Eliminate left half by moving left "book end" over

Comparisons: 1
Step 2 (repeated): Determine the position of the *mid-point* between book ends

- Compare this element with the key!
Search for key = 10

- Step 3 (repeated): In this case, $key < list[mid]$
  - If $key$ is indeed in the list, it *must* be in the left half

Comparisons: 2
Search for key = 10

- Step 3 (repeated): In this case, key < list[mid]
  - If key is indeed in the list, it *must* be in the left half
  - Eliminate right half by moving right "book end" over

Comparisons: 2
Step 2: Determine the position of the *mid-point* between book ends

- Compare this element with the key!

```
Length: 9
0 1 2 3 4 5 6 7 8
```

Comparisons: 3
Search for key = 10

- Step 3 (repeated): In this case, $\text{key} == \text{list}[\text{mid}]$
  - Return mid (5), the position in which the key was found
    - (Method terminates.)
Another Example

- What if the key wasn't in the list?
  - Do a trace for key = 5
public class Searcher {
    private int[] list;

    public int binarySearch(int key) {
        int left = 0;
        int right = list.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;  // compute midpoint
            if (key == list[mid]) {
                return mid;  // found the key! Return index
            }
            else if (key > list[mid]) {
                left = mid + 1;  // move 'left' edge
            }
            else {
                right = mid - 1;  // move 'right' edge
            }
        }

        // didn't find the key! return -1
        return -1;
    }
}
Let's try to analyze the best and worst case scenarios for binary search:

- **Best case is observed when...?**
  - Estimated number of comparisons?

- **Worst case is observed when...?**
  - Estimated number of comparisons?
  - Intuition: You have N elements in the list
  - The list size shrinks by half after each comparison
  - *Let's visualize the worst case on the board*

*Average case? (Next Lab)*
Binary Search Summary

Let's compare to Linear Search

<table>
<thead>
<tr>
<th>List size:</th>
<th>Linear Search: $n$</th>
<th>Binary Search: $\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>14</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>24</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>27</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000,000,000</td>
<td>30</td>
</tr>
</tbody>
</table>

Why would we ever use linear search?

- Think phone book again. What if I only had a phone number, but I wanted to find the name to whom it belongs?
Is This What Google Does?

Puget Sound - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Puget_Sound
Puget Sound /puːˈdʒɛt sɔːnd/ is a sound along the northwestern coast of the U.S. state of Washington, an inlet of the Pacific Ocean, and part of the Salish Sea.
Sound - Puget Sound region - Salish Sea - Puget Sound AVA

University of Puget Sound
www.pugetsound.edu
Admissions, student information, faculty and staff, alumni, news, events, athletics.
Located in Tacoma, WA.
Admission - Departments & Programs - Academics - Employment

Puget Sound Energy
https://www.pse.com
A regulated utility, providing electric and natural gas service to the Puget Sound region.
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Array Examples
- Conclusion

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
"Make the Common Case Fast!"

- Why do *you* sort things?
  - Your wardrobe...
  - Hand of playing cards...
  - Contacts in your phone...
  - Organizing books on your shelf...

- Core life (CS) tenet: "Make the common case fast!"
  - Search is one of the most frequently-used operations in life
    - And certainly in computing
Problem: Given an array of integers, put the list in *ascending* order, that is:

\[
\text{list[0] <= list[1] <= list[2] <= ... <= list[list.length-1]}
\]
Selection Sort

- Observe: at any point, a list is partitioned into:
  - A sorted sublist (blue) and an unsorted sublist (red)
    - Partitioned at index i (which points to the first item in the unsorted sublist)

- Algorithm: While i has not passed the end of the list:
  - Find the index of the smallest item in the unsorted sublist
  - Swap it with the first item in the unsorted sublist at index i
  - Increment i (increasing sorted sublist)
Selection Sort (Step-By-Step)

- **minIdx**
  - 19 13 12 7
- **i** 7 13 12 19
  - j

- **minIdx**
  - 19 13 12 7
- **i** 7 13 12 19
  - j

- **minIdx**
  - 19 13 12 7
- **i** 7 13 12 19
  - j

- **minIdx**
  - 7 13 12 19
- **i** 7 12 13 19
  - j

- **minIdx**
  - 7 13 12 19
- **i** 7 12 13 19
  - j

- **minIdx**
  - 7 13 12 19
- **i** 7 12 13 19
  - j

- **minIdx**
  - 7 13 12 19
- **i** 7 12 13 19
  - j

- **minIdx**
  - 7 13 12 19
- **i** 7 12 13 19
  - j

**Done!**

- **Sorted**
- **Unsorted**

- **i** Index of the first element in unsorted sublist
- **minIdx** Index of smallest item in unsorted sublist
- **The item currently being compared**
Selection Sort (Cont.)

```java
public class Sorter {
    private int[] list;

    public void selectionSort() {
        //i is the index of the first item of the unsorted sublist
        for (int i = 0; i < list.length; i++) {
            int minIdx = i; //index of minimum item found so far in unsorted sublist
            for (int j = i+1; j < list.length; j++) {
                if (list[j] < list[minIdx]) {
                    //found a smaller item at list[j], so update minIdx to j
                    minIdx = j;
                }
            }
            //swap the two items at index i and minIdx
            int temp = list[i];
            list[i] = list[minIdx];
            list[minIdx] = temp;
        }
    }
}
```

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
(What if Data Was Sorted Already?)

- **minIdx**
  - Index of smallest item so far in unsorted sublist

- **Sorted sublist**
  - Blue markers

- **Unsorted sublist**
  - Red markers

- **Index of the first element in unsorted sublist**
  - i

- **The item currently being compared**
  - j

```
minIdx
7 12 13 19
i

minIdx
7 12 13 19
i j

minIdx
7 12 13 19
i j

minIdx
7 12 13 19
i j

minIdx
7 12 13 19
i

Done!
```
Selection Sort Complexity

- Selection sort
  - Does it matter if the list was already sorted or unsorted?
    - Nope. This is in contrast to an algorithm like `linearSearch()`
  - Always takes the same number of comparisons no matter what data is stored inside the list.
  - Complexity: \( \frac{n(n - 1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \)

- Selection sort has a `consistent` time-complexity
  - *i.e.*, Best/worst/average-case complexities are the same
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
  - Optimizations
- Bubble Sort
- More Examples
- Conclusion

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
A list has: unsorted sublist (red) and a sorted sublist (blue)

- Counter $i$ tracks the number of elements that are in sorted sublist
  - The sorted sublist is empty initially (so, initially $i = 0$)
  - Compare each pair of adjacent items (at $j$ and $j-1$) in unsorted sublist
    - Swap if they're out of order
- Continue until $i$ increments to the size of the list
Bubble Sort Algorithm

- Algorithm: Assume list is defined as a field

```java
public void bubbleSort() {
    // counts the number of elements that are in the sorted sublist
    for (int i = 0; i < list.length; i++) {
        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) {
                // swap the two adjacent items if out of order (bubble up)
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
            }
        }
    }
}
```
public void bubbleSort() {
    // counts the number of elements that are in the sorted sublist
    for (int i = 0; i < list.length; i++) {
        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) {
                // swap the two adjacent items if out of order (bubble up)
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
            }
        }
    }
}

Stop when j falls off the unsorted sublist.
(Everything to the right of j is already sorted!)
Bubble Sort Example (Step-by-Step)

### i = 0

```
19 12 13 7
```

### i = 1

```
12 13 7 19
```

### i = 2

```
12 7 13 19
```

### i = 3

```
7 12 13 19
```

### i = 4

```
7 12 13 19
```

A comparison made between these items

**Complexity Analysis:**
- Best: $n(n-1)/2$
- Worst: $n(n-1)/2$
- Average: $n(n-1)/2$
Another Example

<table>
<thead>
<tr>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = 3</th>
<th>i = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 7 12 13</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
</tr>
<tr>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
</tr>
<tr>
<td>7 19 12 13</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
</tr>
<tr>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
</tr>
<tr>
<td>7 12 19 13</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
</tr>
<tr>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
</tr>
<tr>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
<td>7 12 13 19</td>
</tr>
<tr>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
<td>j-1 j</td>
</tr>
</tbody>
</table>

A comparison made between these items

**Complexity Analysis:**
Best: \( n(n-1)/2 \)
Worst: \( n(n-1)/2 \)
Average: \( n(n-1)/2 \)
Couldn't We Have Stopped Sooner?

Could end the algorithm now!  
(How do we know the list was already sorted?)

Check out this link for animation: http://cs.pugetsound.edu/~aasmith/sorters/
Can We Do Better? (Cont.)

\[ i = 0 \]

\[
\begin{bmatrix}
7 & 12 & 13 & 19 \\
7 & 12 & 13 & 19 \\
7 & 12 & 13 & 19 \\
7 & 12 & 13 & 19 \\
\end{bmatrix}
\]

\[ i = 1 \]

\[
\begin{bmatrix}
7 & 12 & 13 & 19 \\
\end{bmatrix}
\]

THIS IS THE BEST CASE!

(Only one pass of \( i \) is needed when list is already sorted)
Bubble Sort (Optimize It!)

- Optimization: If a pass did not require swaps, the list is already sorted.
  - How do I know if a swap has been made? How do I stop the algorithm if a swap was not made in a pass?
  - Here's the old code:

```java
public void bubbleSort()
{
    for (int i = 0; i < list.length; i++) {
        for (int j = 1; j < list.length - i; j++) {

            // Need to bubble list[j-1] up
            if (list[j-1] > list[j]) {
                // Swap the two adjacent items if out of order
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
            }
        }
    }
}
```
Bubble Sort Algorithm (Optimized)

- If a pass did not require swaps, the list is already sorted.
  - Optimizations made in red (below)

```java
public void bubbleSort()
{
    boolean swapOccurred = true; // Why initialize to true?
    for (int i = 0; swapOccurred && i < list.length; i++) {
        swapOccurred = false; // Assume no swaps will happen

        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) {
                // Need to bubble list[j-1] up
                // Swap the two adjacent items if out of order
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
                swapOccurred = true; //we swapped elements, more passes needed
            }
        }
    }
}
```

Best case time complexity NOW?
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
    - Shaker Sort
- More Examples
- Conclusion

Check out this link for animation (note animation builds list backwards):
http://cs.pugetsound.edu/~aasmith/sorters/
(Cocktail) Shaker Sort

- **Shaker Sort** is a further optimization on Bubble Sort

- Recall: One badly placed item screws up Bubble Sort's optimization
  - (e.g., a small item toward the end of the list)

- Shaker sort fixes this by alternating bubble sort left and right each pass
Shaker Sort Example

$i = 0$

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 5 & 7 & 0 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j & j+1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j & j+1 \\
\end{array}
\]

\[
\begin{array}{cccc}
1 & 0 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j & j+1 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cc}
j-1 & j \\
\end{array}
\]

(No swaps! Done)
Outline

roman

Motivation for Search and Sort

Linear Search

Binary Search

Selection Sort

Bubble Sort

• Optimizations

More Examples

Conclusion
Write a method `reverseList(int[] list)` that reverses the contents of an integer array, and returns it.

- And analyze its complexity
More Practice...

- Write the following methods and determine their time complexity.
  - Write a method `inAscendingOrder(int[] list)` that determines whether an integer array, named `list`, is in increasing order.
  - Write a method `printPairs(ArrayList<String> names)` that prints every pair of names exactly once in an ArrayList of Strings. For instance:

```
ArrayList<String> names
Aidan, Halle
Aidan, Marisa
Aidan, Troy
Halle, Marisa
Halle, Troy
Marisa, Troy
```

Output
More Practice: Prime Numbers

- Write the following methods:

\[
\text{public boolean isPrime(int n)}
\]

\[
\text{public ArrayList<Integer> findPrimes(ArrayList<Integer> list)}
\]

- Given an array list of integers, return an array list of prime numbers from that list.
Complexity of `isPrime()`?

- **Best case?**
  - What would cause the algorithm to terminate quickly?

- **Worst case?**
  - What would cause algorithm to incur the most iterations?

- **Average case?**
  - Is the common number prime or not?

```java
/** Tests whether the given number is prime. 
 * @param N a positive number 
 * @return true if N is prime, and false otherwise 
 */
public boolean isPrime(int N) {
    int divisor = 2;
    while (N % divisor != 0) {
        // try each divisor...
        divisor++;
    }
    return divisor == N;
}
```
Two Dimensional (2D) Arrays

- Sometimes, an array (or ArrayList) isn't sufficient to model our data...

- Example: An airline tracking distances between two cities:
  - Find the total mileage for: Chicago -> Boston -> Atlanta
  - How to store all the distances in an array or ArrayList?
Tables Are Sometimes Necessary

- Solution: Store the distance data in a table
  - Observe: A table is just a array of arrays...

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago [0]</td>
<td>0</td>
<td>983.5</td>
<td>787.8</td>
<td>714.1</td>
</tr>
<tr>
<td>Boston [1]</td>
<td>983.5</td>
<td>0</td>
<td>214</td>
<td>1102</td>
</tr>
<tr>
<td>NYC [2]</td>
<td>787.8</td>
<td>214.0</td>
<td>0</td>
<td>888.2</td>
</tr>
<tr>
<td>Atlanta [3]</td>
<td>714.1</td>
<td>1102</td>
<td>888.2</td>
<td>0</td>
</tr>
</tbody>
</table>
Tables Are Sometimes Necessary

- Adding a route...

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago [0]</td>
<td>0</td>
<td>983.5</td>
<td>787.8</td>
<td>714.1</td>
<td>2000</td>
</tr>
<tr>
<td>Boston [1]</td>
<td>983.5</td>
<td>0</td>
<td>214</td>
<td>1102</td>
<td>3500</td>
</tr>
<tr>
<td>NYC [2]</td>
<td>787.8</td>
<td>214.0</td>
<td>0</td>
<td>888.2</td>
<td>3200</td>
</tr>
<tr>
<td>Atlanta [3]</td>
<td>714.1</td>
<td>1102</td>
<td>888.2</td>
<td>0</td>
<td>4000</td>
</tr>
<tr>
<td>Seattle [4]</td>
<td>2000</td>
<td>3500</td>
<td>3200</td>
<td>4000</td>
<td>0</td>
</tr>
</tbody>
</table>
Making a 2D Array

- Declaration: `dataType[][] arrayName;`

- Instantiation: `arrayName = new dataType[numRows][numColumns];`

- Example:

```java
// make a table of scores (18 holes) for 4 golfers
int[][] golf_scores;
golf_scores = new int[4][18];

// make 3 class rosters, with room for 25 names
String[][] classRoster = new String[3][25];
```
Default Values within a 2D Array

- Recall that array elements are *initialized* to their data type's default values, or `null` if storing objects

  - Example:

    ```java
    int[][] golf_scores;
    golf_scores = new int[4][18];
    ```

What does a 2D array object look like?
If elements' values are known in advance, you can specify them during declaration:

```java
String[][] reportCard = {
    {"A", "B", "A"}, // 1st semester
    {"C-", "C", "B+"}, // 2nd semester
};
```
Accessing 2D Arrays

- Recall to access a 1D array element, we use: `arrayName[index]`
- In contrast, each 2D element has two indices: `arrayName[rowIndex][colIndex]`

Example:

```java
int[][] table = new int[3][4];
table[0][0] = 90;
table[0][1] = 43;
table[0][2] = 10;
table[0][3] = 21;
table[1][1] = 20;
table[3][3] = 100; // runtime error

int Y = table[0][2] + table[1][1];
System.out.println(Y);
```

```java
90 43 10 21
0 1 2 3
90 43 10 21
0 1 2 3
90 43 10 21
0 1 2 3
```
Getting 2D Dimensions

- We know that every 1D array has a `.length` field that stores its size.
- Given a 2D array, how do we get the following?
  - How many rows does table have?
  - How many columns?
- Important to remember that a 2D array is an array of arrays.
- Knowing this: print out all elements in a 2D array.
Some 2D Array Exercises

public class Example2DArray
{
    private int[][] A;

    //assume a constructor instantiates and populates A
}

- Write a method public void printTable() that prints A.

- Write a method public double getAvg() that finds and returns the average value of all elements in A.

- Write a method public int lessThanAvg() that returns the number of elements less than the average of the elements in A.
  
  • *Something's unsettling about this... how fast does it run?*
Open up BlueJ...

- Write a method `public double getDistance(int cityID1, int cityID2)` that returns the distance between any two cities.
- Write a method `public int getNearestCityID(int cityID)` that returns the ID of the city nearest to the city with the given ID.

<table>
<thead>
<tr>
<th>Distance</th>
<th>Chicago (0)</th>
<th>Boston (1)</th>
<th>NYC (2)</th>
<th>Atlanta (3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago (0)</td>
<td>0</td>
<td>983.5</td>
<td>787.8</td>
<td>714.1</td>
</tr>
<tr>
<td>Boston (1)</td>
<td>983.5</td>
<td>0</td>
<td>214</td>
<td>1102</td>
</tr>
<tr>
<td>NYC (2)</td>
<td>787.8</td>
<td>214.0</td>
<td>0</td>
<td>888.2</td>
</tr>
<tr>
<td>Atlanta (3)</td>
<td>714.1</td>
<td>1102</td>
<td>888.2</td>
<td>0</td>
</tr>
</tbody>
</table>
On Your Own: Make a Multiplication Table

<table>
<thead>
<tr>
<th>x</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>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>50</td>
<td>55</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>66</td>
<td>72</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
<td>70</td>
<td>77</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
<td>80</td>
<td>88</td>
<td>96</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
<td>90</td>
<td>99</td>
<td>108</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>100</td>
<td>110</td>
<td>120</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>55</td>
<td>66</td>
<td>77</td>
<td>88</td>
<td>99</td>
<td>110</td>
<td>121</td>
<td>132</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>24</td>
<td>36</td>
<td>48</td>
<td>60</td>
<td>72</td>
<td>84</td>
<td>96</td>
<td>108</td>
<td>120</td>
<td>132</td>
<td>144</td>
</tr>
</tbody>
</table>
Make a Multiplication Table

- How should this class behave? (Code pad)

```
MultTable myTable = new MultTable(5,5);  //creates a 5 x 5 multiplication table
myTable.printTable();
1  2  3  4  5
2  4  6  8 10
3  6  9 12 15
4  8 12 16 20
5 10 15 20 25

MultTable myTable = new MultTable(4,8);  //creates a 4 x 8 multiplication table
myTable.printTable();
1  2  3  4  5  6  7  8
2  4  6  8 10 12 14 16
3  6  9 12 15 18 21 24
4  8 12 16 20 24 28 32
```
Outline

- Motivation for Search and Sort
- Linear Search
- Binary Search
- Selection Sort
- Bubble Sort
  - Optimizations
- More Examples
- Conclusion
Summary of Time Complexity

- **Constant Time:** Number of steps/comparisons is independent of N
  - Example: 1 (best case in linear search)

- **Logarithmic Time:** Strive for this when possible
  - Example: \( \log_2(N) \) (binary search, finding logarithms)

- **Linear Time:** Very good!
  - Example: \( N \)

- **Quadratic Time:** Not bad.... depends on exponent of leading term.
  - Examples: \( N^3 \) (matrix-matrix mult.), \( N^2 \) (bubble and selection sort)

- **Exponential Time:** Avoid like plague
  - Examples: \( 2^N \) (traveling salesman, finding power sets)

- **Unbounded:** No guarantee algorithm will ever stop running!
  - Examples: BogoSort, roll die until 1 shows up
In Conclusion...

- Saw several classic CS algorithms...
- Searching a large collection is one of the most commonly-used operations
  - The most important thing:
    - Making the common case fast
    - Therefore, we sort to make search fast!

- Time complexity:
  - Time taken by an algorithm is quantified as number of comparisons required for a given problem size.
  - Not the only "yardstick"

- More sophisticated object behaviors *(Next time)*
Exam 2 graded:

- Avg = 80.95
  - A: 13
  - B: 11
  - C: 6
  - D: 3
  - F: 6

Good: "Which Data structure to use?", ArrayLists, Strings

Bad: rotateLeft
Solutions posted:
- Lab 8: Life
- Lab 9: Smarter Chatbot
- Lab 10 tonight

Due dates:
- Hwk 7 Pump due tonight!
- Hwk 8 proposal due tonight! (Soft deadline)

Last time... complexity, searching
Hi CS 161 students!

As the semester comes to a close, I wanted to share with you all some tutoring opportunities for reading period and finals week.

Thursday, 5/9 - 161 review session from 3-5 pm
Friday, 5/10 - drop-in hours from 2-4 pm

These hours will all be held at the CWLT. See ya there!

Best,
Lia the CWLT Computer Science Tutor