Yardsticks for Comparing Algorithms

- How do we characterize the "goodness" of an algorithm?
  - Accuracy of results
  - Space required to run algorithm *(space complexity)*
  - Time required to run algorithm *(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*
Outline

- Motivation
- Search algorithms
  - Linear Search
  - Binary Search
- Sorting Algorithms
  - 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.
Searching within Collections

- We've now seen three data structures:
  - Arrays, ArrayList, HashMap
  - More such data structures to come in CS 261
    - Sets, trees, stacks, queues, heaps...

- One of the key operations is to find or search for things within collections of data:
Linear Search

- 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 linearSearch():
  - Visits each item in order and checks against the key
    - Might return part way in the loop if key is found!
```
Linear search algorithm:

```java
public static int linearSearch(int[] list, int key) {
    // Need to return the index of the key, so we
    // don't want to use a for-each loop (index-less)
    for (int i = 0; i < list.length; i++) {
        // found the key! return the index and return early!
        if (key == list[i]) {
            return i;
        }
    }
    // didn't find the key! return -1
    return -1;
}
```

Suppose `list` contains [90, 33, 64, 23, 9, 34, 100, 56]

- Length of array is 8
- How many key-comparisons do we need to find 23, 12, and then 90?
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 **likeliest** scenario.

### Time Complexity of Linear Search for list of size \( N \)?

<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
Does List Order Help?

- Is there a *smarter* way to search if the list is sorted in ascending order?
  - Think: Lots of things are sorted in life... how does that help you search?
Binary Search

- The idea: Exploit the sorted ordering
  - Skip to the middle item in the list, and compare with key...
  - If not found, then which half might the key be?
Binary Search

- The idea: Exploit the sorted ordering
  - Skip to the middle item in the list, and compare with key...
  - If not found, then which half might the key be?

Think contact list: Search for the name Kate

Scroll to the name in the middle of the list.

1) Is it Kate?
   (a) If so, you're done!
2) If not, does Kate precede the name on the page?
   Repeat on first half of the contact list
3) Otherwise,
   Repeat on second half of the contact list
4) Repeat until you run find Kate, or run out of names
Remember, the list must already be in **ascending order** for binary search to work.
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>
<th>9</th>
<th>10</th>
<th>12</th>
<th>13</th>
<th>15</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

length: 9
```

Comparisons: 1

left (0)
mid (4)
From: (0+8)/2
right (8)
Step 3: In this case, \( \text{key} > \text{list[mid]} \)

- If \( \text{key} \) 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
Search for key = 10

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

Comparisons: 2
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
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
Search for key = 10

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

```
3 4 6 7 9 10 12 13 15
```

**length: 9**

**Comparisons: 3**
Search for key = 10

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

Comparisons: 3
Another Example

- What if the key wasn't in the list?
  - Do a trace for key = 5
public static int binarySearch(int[] list, int key) {
    int left = 0;
    int right = list.length - 1;
    while (left <= right) {
        // compute midpoint
        int mid = (left + right) / 2;
        if (key == list[mid]) {
            return mid;  // found the key! Return early
        }
        else if (key > list[mid]) {
            left = mid + 1;  // shift 'left' edge to the right
        }
        else {
            right = mid - 1;  // shift 'right' edge to the left
        }
    }
    // 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: $n$</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 Wikipedia
Puget Sound /ˈpuːtəɡˈsʌnd/ 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/ University of Puget Sound
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/ Puget Sound Energy
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
http://cs.pugetsound.edu/~dchiu/cs161/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 CS (tenet) tenet: "Make the common case fast!"
  - Search is one of the most frequently-used operations in life
    - And certainly in computing
Sorting

- Problem: Given an array of integers, put the list in *ascending* order, that is:

  
  \[
  \text{list}[0] \leq \text{list}[1] \leq \text{list}[2] \leq \ldots \leq \text{list}[\text{list.length}-1]
  \]
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**
  - **i**: Index of the first element in unsorted sublist
  - **minIdx**: Index of smallest item in unsorted sublist
  - **Sorted**: Blue
  - **Unsorted**: Red
  - **The item currently being compared**: Green

**Done!**
public static void selectionSort(int[] list) {
    //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**
  - 7 12 13 19
  - i

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

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

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

---

The item currently being compared

Sorted sublist

Index of the first element in unsorted sublist

Index of smallest item so far in unsorted sublist

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 \textit{consistent} time-complexity
  - \textit{i.e.}, Best/worst/average-case complexities are the same
Outline

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

Check out this link for animation
http://cs.pugetsound.edu/~dchiu/cs161/sorters/
Bubble Sort

- A list is partitioned into an unsorted sublist & a sorted sublist
  - Counter $i$ tracks the number of elements that are in sorted sublist
    - The sorted sublist starts out empty (so, initially $i = 0$)
    - Compare each pair of adjacent values (in $j$ and $j-1$) in the unsorted sublist
      - Swap if they're out of order
  - After each pass, increment $i$ by 1 and start over
  - Continue until $i$ increments to the size of the list
Bubble Sort Algorithm

- Algorithm: Assume list is defined as a field

```java
public static void bubbleSort(int[] list) {
    //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 static void bubbleSort(int[] list) {
    //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)

Complexity Analysis:
Best: $\frac{n(n-1)}{2}$
Worst: $\frac{n(n-1)}{2}$
Average: $\frac{n(n-1)}{2}$
Another Example

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

i = 0

19 7 12 13
j-1 j

7 19 12 13
j-1 j

7 12 19 13
j-1 j

7 12 19 13
j-1 j

7 12 13 19
j-1 j

7 12 13 19
j-1 j

19 7 12 13
j-1 j

i = 1

7 12 13 19
j-1 j

7 12 13 19
j-1 j

7 12 13 19
j-1 j

7 12 13 19
j-1 j

7 12 13 19
j-1 j

Check out this link for animation:
http://cs.pugetsound.edu/~aasmith/sorters/

Should end the algorithm now!

(How do we know the list was already sorted?)
Can We Do Better? (Cont.)

\[ i = 0 \]

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

\[ i = 1 \]

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

\[ \text{Done!} \]

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 static void bubbleSort(int[] list) {
    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;
            }
        }
    }
}
```
If a pass did not require swaps, the list is already sorted.

- Optimizations made in red (below)

```java
public static void bubbleSort(int[] list)
{
    boolean swapOccurred = true; // Why initialize to true?
    for (int i = 0; swapOccurred && i < list.length; i++) {
        swapOccurred = false; // No swaps yet
        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
http://cs.pugetsound.edu/~dchiu/cs161/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

1 5 7 0

j-1 j

1 5 7 0

j-1 j

1 5 0 7

j j+1

1 5 0 7

j j+1

1 5 0 7

j j+1

i = 1

1 5 0 7

j j+1

1 5 0 7

j j+1

1 0 5 7

j j+1

1 0 5 7

j j+1

1 0 5 7

j j+1

(No swaps! Done)

i = 2

0 1 5 7

j-1 j

0 1 5 7

j-1 j

0 1 5 7

j-1 j

(No swaps! Done)
public static void shakerSort(int[] list) {
    boolean swapped = true;
    for (int i = 0; swapped && i < list.length/2; i++) {
        swapped = false;
        for (int j = 1; j < list.length - i; j++) {
            if (list[j-1] > list[j]) {
                int temp = list[j-1];
                list[j-1] = list[j];
                list[j] = temp;
                swapped = true;
            }
        }
        // now bubble down (go left), if needed
        if (swapped) {
            for (int j = list.length - i - 1; j > i; j--) {
                if (list[j-1] > list[j]) {
                    int temp = list[j-1];
                    list[j-1] = list[j];
                    list[j] = temp;
                }
            }
        }
    }
}
Outline

- 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:

```java
public boolean isPrime(int n)

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 the 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;
}
```
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 comparisons is independent of the search space
  - 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: $N$

- **Polynomial Time**: Not bad --- All depends on exponent of leading term.
  - Examples: $N^3$ (matrix multiplication), $N^2$ (bubble and selection sort)

- **Exponential Time**: Just don't, unless $N$ is very small
  - Examples: $2^N$ (traveling salesman, finding power sets)

- **Factorial Time**: Just don't
  - Examples: BogoSort
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)*
Hwk 8
- Optional Pitch is due this Friday! (By email)

Tomorrow's Lab:
- Install IntelliJ https://www.jetbrains.com/idea/

Last time…
- Enum classes for streamlining constants
- Non-static vs. static
  - The static main method

Today:
- A brief tour of IntelliJ
- Start complexity analysis
Lab 11 Post-Mortem:
  • Thoughts on IntelliJ? Pros and cons?

Last time...
  • Time-Complexity Analysis
  • Linear Search

Today:
  • Binary Search Algorithm
Breathe: Last full week of classes!

Hwk 8 due Wed, May 6
  • Finish it early!
  • You have all the tools to finish it right now!

Last time...
  • Lab 11 post-mortem
  • Binary Search

Today...
  • Impact of logarithmic-time complexity
  • Selection sort
Hwk 8 due Wed, May 6

- Finish it early!
- You have all the tools to finish it right now!

Last time...

- Impact of logarithmic-time complexity
- Selection sort

Today:

- Review selection sort
- Bubble Sort
- With live demonstration!