Algorithmic Complexity I
CSE 120 Spring 2017

Instructor:  Teaching Assistants:
Justin Hsia         Anupam Gupta, Braydon Hall, Eugene Oh, Savanna Yee
Administrivia

- **Assignments:**
  - Binary Practice (4/21)
  - Creativity Assignment (4/24)

- **Midterm in class on Wednesday, 4/26**
  - 1 sheet of notes (2-sided, letter, handwritten)
  - Fill-in-the-blank(s), short answer questions, maybe simple drawing
    - Questions will cover lectures, assignments, and readings
  - Midterm Review sheet will be released tonight (4/19), will be covered in lab next week (4/25)
Outline

- Algorithm Analysis: The Basics
- Comparing Algorithms
- Orders of Growth
Algorithm Correctness

- An algorithm is considered **correct** if for every input, it reports the correct output **and** doesn’t run forever or cause an error.

- Incorrect algorithms may run forever, crash, or not return the correct answer:
  - But they could still be useful!
  - *e.g.* an approximation algorithm

- Showing correctness:
  - Mathematical proofs for algorithms
  - Empirical verification of implementations
Algorithm Analysis

- One commonly used criterion for analyzing algorithms is **computation time**
  - How long does the algorithm take to run and finish its task?
  - Can be used to compare different algorithms for the same computational problem

- How to measure this time?
  - Counting in my head
  - Stopwatch
  - Within your program
Aside: Computation Time

- Computers take time to complete their tasks
  - Under the hood, it’s sort of like a bunch of buckets of water filling up – you have to wait for water to reach the top of a bucket for a single computation to complete
  - Buckets take about a billionth of a second to fill (~1 nanosecond)
    - There are billions of them on a single chip!

- A CPU can generally only execute one instruction at a time
Timing in Processing

- The function `millis()` returns the number of milliseconds since starting your program (as an `int`)
  - To start timing, call and store the value in a variable
  - Call again after your computation and subtract the values

```java
void draw() {
    int time = millis();
    someComputation();
    println(" Took " + (millis()-time) + " milliseconds to compute.");
    noLoop();
}
```
Outline

- Algorithm Analysis: The Basics
- Comparing Algorithms
- Orders of Growth
Algorithm: Searching A Sorted Array

- **Input:** Numbers in a sorted array, desired number
- **Output:** If desired number is in the array \((\text{true/false})\)

**Algorithm 1:**
- Check each index starting from 0 for desired number
  - If equal, then report \text{true}
  - If not equal, then move to next index
  - If at end of array, then report \text{false}
- Called Linear Search (also works for unsorted array)
Algorithm: Searching A Sorted Array

- **Input:** Numbers in a sorted array, desired number
- **Output:** If desired number is in the array (true/false)

**Algorithm 2:**
- Check “middle” index for desired number
  - If equal, then report true
  - If less than desired number, check halfway forwards next
  - If greater than desired number, check halfway backwards next
- If no halfway point left, then report false
- Called Binary Search
Peer Instruction Question

- On average, which algorithm would take less time to complete a search?

A. Algorithm 1 (Linear Search)
B. Algorithm 2 (Binary Search)
C. They’d take about the same amount of time
Measuring Linear Search

- Let’s time Linear Search:

```java
void draw() {
    int n = 3;
    println("Is " + n + " in intArr?");
    int time = millis();
    println(linearSearch(n));
    println("Took " + (millis()-time) + " milliseconds to compute.");
    noLoop();
}
```

- One issue: our algorithm seems to be too fast to measure!
  - How can we fix this?
Best Case vs. Worst Case vs. Average Case

- We were measuring close to the best case!
  - Didn’t matter how long our array was

- Could measure average case instead
  - Run many times on random numbers and average results

- Instead, we’ll do worst case analysis. Why?
  - Nice to know the most time we’d ever spend
  - Worst case happens often
  - Average case is often similar to worst case
Example of Worst Case in Action

- Many web servers out there run something called “The Apache HTTP Server” (or just Apache for short)
  - When a user enters a particular URL, Apache delivers the correct files from the server to the person on the internet
  - An old version of Apache had a bug where if you entered a URL with tons of consecutive slashes, it could take *hours* to complete the request
    - *e.g.* [http://someurl.com//////////](http://someurl.com//////////)

- Bottom line: an algorithm is often judged by its worst case behavior
What is the Worst Case?

Discuss with your neighbor (no voting):

- Assume `intArr.length` is `1000000` and `intArr[i] = i`;
- What is a worst case argument for `num` for Linear Search?
- What is a worst case argument for `num` for Binary Search?

A. 1
B. 500000
C. 1000000
D. 1000001
E. Something else

```java
boolean linearSearch(int num) {
    for(int i = 0; i < intArr.length; i = i + 1) {
        if(intArr[i] == num) {
            return true;
        }
    }
    return false;
}
```
Timing Experiments

- Let’s try running Linear Search on a worst case argument value
  - Results:

- Now let’s run Binary Search on a worst case argument value
  - Results:
Runtime Intuition

❖ Does it seem reasonable that the runtimes were inconsistent?

❖ Some reasons:
  ▪ Your computer isn’t just running Processing – there’s a lot of other stuff running (e.g. operating system, web browser)
  ▪ The computer hardware does lots of fancy stuff to avoid slowdown due to physical limitations
    • These may not work as well each execution based on other stuff going on in your computer at the time
Empirical Analysis Conclusion

- We’ve shown that Binary Search is seemingly much faster than Linear Search
  - Similar to having two sprinters race each other

- Limitations:
  - Different computers may have different runtimes
  - Same computer may have different runtime on same input
  - Need to implement the algorithm in order to run it

- Goal: come up with a “universal algorithmic classifier”
  - Analogous to coming up with a metric to compare all athletes (or fighters)
Outline

- Algorithm Analysis: The Basics
- Comparing Algorithms
- Orders of Growth
Characterizing Algorithms

The method computer scientists use is roughly:

1) Measure the algorithm’s runtime on many different input sizes $N$ (e.g. arrays of length 100, 200, 400, 800, …)
   • To avoid runtime issues, can also count number of “steps” involved

2) Make a plot of the runtime as a function of $N$, which we’ll call $R(N)$

3) Determine the general *shape* of $R(N)$
   • Does $R(N)$ look like $N$ (linear), $N^2$ (quadratic), $N^3$ (cubic), $\log N$ (logarithmic), etc.
Linear Search

- As the name implies, Linear Search is linear
  - If you double N, then R(N) should roughly double

<table>
<thead>
<tr>
<th>N (input size)</th>
<th>R(N) (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 items</td>
<td>1.4 sec</td>
</tr>
<tr>
<td>500 items</td>
<td>2.8 sec</td>
</tr>
<tr>
<td>671 items</td>
<td>3.8 sec</td>
</tr>
<tr>
<td>1000 items</td>
<td>5.7 sec</td>
</tr>
</tbody>
</table>
Peer Instruction Question

- Algorithm for: do any pairs in array sum to zero?
- Which function does \( R(N) \) look like?
  - Vote at http://PollEv.com/justinh
  - A. \( \sqrt{N} \)
  - B. \( \log(N) \)
  - C. \( N \)
  - D. \( N^2 \)
  - E. \( 2^N \)

<table>
<thead>
<tr>
<th>N (input size)</th>
<th>R(N) (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 items</td>
<td>1.1 seconds</td>
</tr>
<tr>
<td>200 items</td>
<td>4.3 seconds</td>
</tr>
<tr>
<td>300 items</td>
<td>9.6 seconds</td>
</tr>
<tr>
<td>400 items</td>
<td>17.0 seconds</td>
</tr>
</tbody>
</table>
Orders of Growth

- The order of growth of \( R(N) \) is its general shape:
  - Constant \( 1 \)
  - Logarithmic \( \log N \)
  - Linear \( N \)
  - Quadratic \( N^2 \)
  - Cubic \( N^3 \)
  - Exponential \( 2^N \)
  - Factorial \( N! \)
Orders of Growth

- The order of growth of R(N) is its general shape:
  - Use *dominant* term
  - *e.g.* $10N^2 + 4 \log N$ is quadratic
Binary Search

What order of growth is Binary Search?
- Analyze using number of “steps” in worst case

<table>
<thead>
<tr>
<th>N (input size)</th>
<th>Indices to Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 items</td>
<td></td>
</tr>
<tr>
<td>2 items</td>
<td></td>
</tr>
<tr>
<td>4 items</td>
<td></td>
</tr>
<tr>
<td>8 items</td>
<td></td>
</tr>
<tr>
<td>16 items</td>
<td></td>
</tr>
</tbody>
</table>
Which is Faster?

- Suppose we have two algorithms: one is linear in \( N \) and the other is quadratic in \( N \)
  - No voting

- Which is faster?
  A. Linear Algorithm
  B. Quadratic Algorithm
  C. It depends
The Reason Order of Growth Matters

- Roughly speaking, we care about really big N in real world applications
  - e.g. For Facebook, N (users) is ~ 1 billion
    - Want to generate list of suggested friends? Better be a fast algorithm as a function of N

- Order of growth is just a rough rule of thumb
  - There are limited cases where an algorithm with a worse order of growth can actually be faster
  - In almost all cases, order of growth works very well as a representation of an algorithm’s speed
Orders of Growth Comparison

- The numbers below are rough estimates for a “typical” algorithm on a “typical” computer – provides a qualitative difference between the orders of growth.

<table>
<thead>
<tr>
<th></th>
<th>Linear</th>
<th>Linearithmic</th>
<th>Quadratic</th>
<th>Cubic</th>
<th>Exponential</th>
<th>Exponential</th>
<th>Factorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n$</td>
<td>$n \log_2 n$</td>
<td>$n^2$</td>
<td>$n^3$</td>
<td>$1.5^n$</td>
<td>$2^n$</td>
<td>$n!$</td>
</tr>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.