Facebook should actually be Tinder too

“The idea behind Meetups is smart, but the execution is a mess. Because Meetups ambiguously shows multiple people at once, sends aggressive notifications to participate and encompasses all kinds of relationships, the results are meaningless.

“Facebook’s made it easier than ever to “feel connected,” endlessly scrolling through friends’ photos, while actually allowing us to isolate ourselves.”

• [https://techcrunch.com/2018/02/14/facebook-matchmaker/](https://techcrunch.com/2018/02/14/facebook-matchmaker/)
LCM Report Wrap-Up
Administrivia

- Assignments:
  - Innovation Exploration post (2/27)
  - Project Update in lab on Thursday (3/1)
  - Innovation Exploration comments (3/2)

- “Big Ideas” lecture: Proofs and Computation
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 → inconsistent, inaccurate, not precise
  - Stopwatch → inconsistent, still inaccurate
  - Within your program → much more accurate
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. Transistors
  - 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 and tasks take many instructions
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(); // stores start time in variable
    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 (true/false)

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

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

- On average, which algorithm should take less time to complete a search?
  - Vote at http://PollEv.com/justinh

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

but how do you “prove” this?
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! (keeps showing 0 milliseconds)
  - How can we fix this?
    - try longer arrays
    - try different n
Which Case Do We Care About?

- 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 may happen often
  - 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? C/D
  - What is a worst case argument for `num` for Binary Search? A/C/D

A. 1 (beginning)
B. 500000 (middle)
C. 1000000 (end)
D. 1000001 (not in array)
E. Something else (else)
Timing Experiments

- For array length 100,000,000 and \( \text{intArr}[i] = i; \) searching for 100,000,001

- Let’s try running Linear Search on a worst case argument value
  - Results: 47, 47, 47, 31, 47 ms

- Now let’s run Binary Search on a worst case argument value
  - Results: 0, 0, 0, 0 ms
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: better “athlete”: speed skater or bobsledder?
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>

The graph shows that the runtime R(N) is approximately linear with respect to N, forming a straight line.
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>
<th>( \text{Algorithm Illustration} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 items</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>2 items</td>
<td>2</td>
<td>( 2 )</td>
</tr>
<tr>
<td>4 items</td>
<td>3</td>
<td>( 3 )</td>
</tr>
<tr>
<td>8 items</td>
<td>4</td>
<td>( 4 )</td>
</tr>
<tr>
<td>16 items</td>
<td>5</td>
<td>( 5 )</td>
</tr>
<tr>
<td>32 items</td>
<td>6</td>
<td>( 6 )</td>
</tr>
<tr>
<td>64 items</td>
<td>7</td>
<td>( 7 )</td>
</tr>
</tbody>
</table>

\[ R(N) = \log_2(N) \]

logarithmic!
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

  $\text{Graph of order of growth curves on log-log plot}$
Peer Instruction Question

- Algorithm for: do any pairs in array sum to zero?
- Which function does $R(N)$ look like?
  - 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>
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.