Introduction to
Computer Science Theory
How complex is a program (or algorithm)?

- **Algorithms** is the study of
  1. What algorithms can be used to solve common problems?
  2. How fast/slow are those algorithms?

- For example:
  - Binary search on an array of size $m$ takes about $\log_2(m)$ steps.
  - Linear search on an array of size $m$ takes about $m$ steps.

- Also, the sorting problem:
  - BubbleSort on an array of size $m$ takes about $m^2$ steps.
  - MergeSort on an array of size $m$ takes about $m \log_2(m)$ steps.
How complex is a problem?

- Computational Complexity is the study of how hard a (computational) problem is.
  - In other words, how fast/slow is the fastest algorithm that solves the problem?

- For example, the searching problem:
  - Searching an unsorted array: the best known algorithm is linear search, which takes about $m$ steps.
  - Searching a sorted array: the best known algorithm is binary search, which takes about $\log_2 m$ steps.

- Also, the sorting problem:
  - The best known algorithms are MergeSort, HeapSort, and others, which all take about $m \log_2(m)$ steps.
What do you mean, “about”?

Computer Scientists study how fast algorithms are *approximately*, in the *worst case*.

For example, for an array of size $m$:
- Algorithm 1 takes $2m$ steps in the worst case
- Algorithm 2 takes $3m + 40$ steps in the worst case

Then we say Algorithm 1 and 2 are both *about* the same!
Tangent: Moore’s Law

- Manufacturers double the speed of computers roughly every 1.5 years.
  - This has held true since the first computers (~1950)

- So if you’re trying to decide on an algorithm, and the constant factor for one seems high, wait 1.5 years and it will be cut in half.
Moore’s Law
The Fifth Paradigm

![Logarithmic Plot](image)

Calculations per Second per $1,000$

<table>
<thead>
<tr>
<th>Year</th>
<th>Electromechanical</th>
<th>Relay</th>
<th>Vacuum Tube</th>
<th>Transistor</th>
<th>Integrated Circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1900</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1910</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1920</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1930</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1940</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1950</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1960</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1970</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1980</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1990</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Defining “approximately”

- We say \( f(m) = O(g(m)) \) if:
  - There are constants \( a \) and \( M \) such that
    \[
    a \cdot g(m) \geq f(m), \text{ for all } m \geq M
    \]

- This is called “Big-O” notation
  - e.g., \( 3m + 40 = O(m) \)
    (read: “Big-O of \( m \)” or “O of \( m \)”)
  - Likewise, \( 2m = O(m) \)
  - So, \( 3m + 40 \) and \( 2m \) are basically the same
  - \( 3m^2 + 7m + 2 \) is not \( O(m) \), but \( 3m^2 + 7m + 2 = O(m^2) \).
Recall: Linear Search

# Input: Array D, integer key
# Output: first index of key in D, 
# or -1 if not found

For i := 0 to end of D:
   if D[i] equals key:
      return i
return -1
Recall: Binary Search Algorithm

# Input: Sorted Array D, integer key
# Output: first index of key in D, or -1 if not found

left = 0, right = index of last element
while left <= right:
    middle = index halfway between left, right
    if D[middle] matches key:
        return middle
    else if key comes before D[middle]:  // D is sorted
        right = middle - 1
    else:
        left = middle + 1
return -1
How much faster is binary search?

- Way, way faster
  - Assuming the array is already sorted

- But precisely how much?

<table>
<thead>
<tr>
<th>For an array of size:</th>
<th>Linear search might visit:</th>
<th>Binary search might visit:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^4 = 16$</td>
<td>16 elements</td>
<td>$4 + 1 = \log_2(16) + 1$</td>
</tr>
<tr>
<td>$2^8 = 256$</td>
<td>256 elements</td>
<td>$8 + 1 = \log_2(256) + 1$</td>
</tr>
<tr>
<td>$2^{12} = 4096$</td>
<td>4096 elements</td>
<td>$12 + 1 = \log_2(4096) + 1$</td>
</tr>
<tr>
<td>$2^n = m$ elements</td>
<td>$m$ elements</td>
<td>$n + 1 = \log_2(m) + 1$</td>
</tr>
</tbody>
</table>
Recall: BubbleSort

# Input: an array of elements called D
# Output: D is sorted

performedSwap = true
while we performed at least one swap:
    performedSwap = false
    for i goes from 0 to the end of D, less 1:
        if D[i] > D[i+1]:
            swap D[i] and D[i+1]
            performedSwap = true
Recall: The MergeSort Algorithm

# input array D, output sorted D

mergeSize = 1
while mergeSize < length of D:
    i = 0
    while i < length of D:
        middle = i + mergeSize
        right = the smaller of
            i + 2*mergeSize, length of D
        merge(D, i, middle-1, middle, right-1)
        i = right
    mergeSize = mergeSize * 2
Recall: The Merge Algorithm

# input array `D`, with two sorted sub-arrays
# from `left1` to `right1`, from `left2` to `right2`

# output array `D`, with one sorted sub-array
# from `left1` to `right2`

```python
function merge(D, left1, right1, left2, right2):
    Temp = an array big enough to hold both subarrays
    i1 = left1, i2 = left2, iTemp = 0
    while i1 <= right1 and i2 <= right2:
        if D[i1] < D[i2]:
            Temp[iTemp] = D[i1]
            increment i1
        else:
            Temp[iTemp] = D[i2]
            increment i2
        increment iTemp
    copy Temp back to the right positions of D
```
How much faster is MergeSort?

- Way, way faster (on big arrays)
- But precisely how much?

<table>
<thead>
<tr>
<th>For an array of size:</th>
<th>BubbleSort comparisons:</th>
<th>MergeSort comparisons:</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^4 = 16$</td>
<td>$\sim 16^2 = 256$</td>
<td>$\sim 16 \times 4 = 64$</td>
<td>$\times 4$</td>
</tr>
<tr>
<td>$2^8 = 256$</td>
<td>$\sim 256^2 = 65,536$</td>
<td>$\sim 256 \times 8 = 4,096$</td>
<td>$\times 32$</td>
</tr>
<tr>
<td>$2^{12} = 4096$</td>
<td>$\sim 4,096^2 = 16,777,216$</td>
<td>$\sim 4096 \times 12 = 49,152$</td>
<td>$\times 341$</td>
</tr>
<tr>
<td>$2^n = m$ elements</td>
<td>$\sim m^2$</td>
<td>$\sim m \times \log m$</td>
<td>$\times (m / \log m)$</td>
</tr>
</tbody>
</table>
The Big-O notation helps us group algorithms into classes with similar speed.

For example, MergeSort and QuickSort both belong to the class with speed \(O(m \log_2(m))\).

**Common classes of algorithms:**

<table>
<thead>
<tr>
<th>Speed</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(\log_2(m)))</td>
<td>Binary search</td>
</tr>
<tr>
<td>(O(m))</td>
<td>Linear search</td>
</tr>
<tr>
<td>(O(m \log_2(m)))</td>
<td>MergeSort, QuickSort (expected case)</td>
</tr>
<tr>
<td>(O(m^2))</td>
<td>BubbleSort, QuickSort (worst case)</td>
</tr>
<tr>
<td>(O(m^3))</td>
<td>Naïve Matrix multiplication</td>
</tr>
<tr>
<td>(O(2^m))</td>
<td>Finding the shortest route to (m) different cities</td>
</tr>
</tbody>
</table>
Exercise

Let’s say you have implementations of linearSearch, binarySearch, and mergeSort with the following worst-case complexities:

- linearSearch: $m$
- binarySearch: $\log_2 m$
- mergeSort: $2m \log_2 m + 5$

What’s faster (worst case) on an array with 1000 elements:

- Running linearSearch once,
  or running mergeSort followed by binarySearch?
- Running linearSearch 20 times,
  or running mergeSort followed by binarySearch 20 times?
Remember: *Algorithmic Thinking* (aka, *Computational Thinking*)

- Programming is a skill you will use in all computer science courses
- **Programming != Computer Science**
- *Algorithmic Thinking* is a collection of different ways that computer scientists think about problems
- It’s what you mostly learn about in computer science classes, when you’re done learning core programming techniques
What you should know

- When comparing algorithms, computer scientists analyze approximate worst-case performance.

- Big-O notation groups algorithms into complexity classes (e.g., $O(m)$, $O(m^2)$).

- Remember the complexity classes for:
  - Linear search, binary search
  - BubbleSort, MergeSort