What Makes a Good Algorithm?

- Suppose you have two possible algorithms or data structures that basically do the same thing; which is better?
  - Faster?
  - Less space?
  - Easier to code?
  - Easier to maintain?
  - Required for homework?
- How do we measure the first two?

Sample Problem: Searching

- Determine if a sorted array of integers contains a given integer
  - 1st solution: Linear Search (check each element)

```java
static boolean find (int[] a, int item) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == item) return true;
    }
    return false;
}
```

- 2nd solution: Binary Search

```java
int low = 0;
int high = a.length - 1;
while (low <= high) {
    int mid = (low + high) / 2;
    if (a[mid] < item)
        low = mid + 1;
    else if (item < a[mid])
        high = mid - 1;
    else return true;
}
return false;
```

Linear Search vs. Binary Search

- Which one is better?
  - Linear Search is easier to program
  - But Binary Search is faster... isn’t it?
- How do we measure to show that one is faster than the other?
  - Experiment?
  - Proof?
  - But which inputs do we use?

Simplifying assumption #1: Use the size of the input rather than the input itself
- For our sample search problem, the input size is n where n-1 is the array size
- Simplifying assumption #2: Count the number of “basic steps” rather than computing exact times

One Basic Step = One Time Unit

- Basic step:
  - Input or output of a scalar value
  - Accessing the value of a scalar variable, array element, or field of an object
  - Assignment to a variable, array element, or field of an object
  - A single arithmetic or logical operation
  - Method invocation (not counting argument evaluation and execution of the method body)
- For a conditional, we count number of basic steps on the branch that is executed
- For a loop, we count number of basic steps in loop body times the number of iterations
- For a method, we count number of basic steps in method body (including steps needed to prepare stack-frame)

Runtime vs. Number of Basic Steps

- But isn’t this cheating?
  - The runtime is not the same as the number of basic steps
  - Time per basic step varies depending on computer, on compiler, on details of code...
- Well... yes, it is cheating in a way
  - But the number of basic steps is proportional to the actual runtime
- Which is better?
  - n or n^2 time?
  - 100 n or n^2 time?
  - 10,000 n or n^2 time?
- As n gets large, multiplicative constants become less important
- Simplifying assumption #3: Multiplicative constants aren’t important
Using Big-O to Hide Constants

- Roughly, \( f(n) = O(g(n)) \) means that \( f(n) \) grows like \( g(n) \) or slower
- **Definition:** \( O(g(n)) \) is a set; \( f(n) \) is a member of this set if and only if there exist constants \( c \) and \( N \) such that
  \[
  0 \leq f(n) \leq c g(n),
  \]
  for all \( n \geq N \)
- We should write
  \( f(n) \in O(g(n)) \)
- But by convention, we write
  \( f(n) = O(g(n)) \)

**Claim:** \( n^2 + n = O(n^2) \)

We know \( n \leq n^2 \) for \( n \geq 1 \)

So \( n^2 + n \leq 2n^2 \) for \( n \geq 1 \)

So by definition,
\[
  n^2 + n = O(n^2)
\]
for \( c=2 \) and \( N=1 \)

**Big-O Examples**

- **Claim:** \( 100n + \log n = O(n) \)
  
  We know \( \log n \leq n \) for \( n \geq 1 \)
  
  So \( 100n + \log n \leq 101n \) for \( n \geq 1 \)
  
  So by definition,
  \[
  100n + \log n = O(n)
  \]
  for \( c=101 \) and \( N=1 \)

- **Claim:** \( \log_B n = O(\log n) \)
  
  Let \( k = \log n \)
  
  Then \( n = 2^k \)
  
  (the subscripts are too messy; switch to board)

**Question:** Which grows faster: \( \sqrt{n} \) or \( \log n \)?

Simple Big-O Examples

- Let \( f(n) = 3n^2 + 6n - 7 \)
  
  - Claim \( f(n) = O(n^2) \)
  - Claim \( f(n) = O(n^3) \)
  - Claim \( f(n) = O(n^4) \)
  - ... \[
  g(n) = 4n \log n + 34n - 89
  \]
  
  - Claim \( g(n) = O(n \log n) \)
  - Claim \( g(n) = O(n^2) \)
  - Claim \( h(n) = O(2^n) \)
  - Claim \( a(n) = 34 \)
  - Claim \( a(n) = O(1) \)

  - Only the leading term (the term that grows most rapidly) matters

**Big-O Examples**

- **Claim:** \( 100n + \log n = O(n) \)
  
  We know \( \log n \leq n \) for \( n \geq 1 \)
  
  So \( 100n + \log n \leq 101n \) for \( n \geq 1 \)
  
  So by definition,
  \[
  100n + \log n = O(n)
  \]
  for \( c=101 \) and \( N=1 \)

- **Claim:** \( \log_k n = O(\log n) \)
  
  Let \( k = \log n \)
  
  Then \( n = 2^k \)
  
  (the subscripts are too messy; switch to board)

**Question:** Which grows faster: \( \sqrt{n} \) or \( \log n \)?

Problem-Size Examples

- Suppose we have a computing device that can execute 1000 operations per second; how large a problem can we solve?

<table>
<thead>
<tr>
<th>Complexity</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>31</td>
<td>244</td>
<td>1897</td>
</tr>
<tr>
<td>( 3n^2 )</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

Commonly Seen Time Bounds

<table>
<thead>
<tr>
<th>( O(1) )</th>
<th>constant</th>
<th>excellent</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(\log n) )</td>
<td>logarithmic</td>
<td>excellent</td>
</tr>
<tr>
<td>( O(n) )</td>
<td>linear</td>
<td>good</td>
</tr>
<tr>
<td>( O(n \log n) )</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>quadratic</td>
<td>OK</td>
</tr>
<tr>
<td>( O(n^3) )</td>
<td>cubic</td>
<td>maybe OK</td>
</tr>
<tr>
<td>( O(2^n) )</td>
<td>exponential</td>
<td>too slow</td>
</tr>
</tbody>
</table>

Related Notations

**Big-Omega**

**Definition:** \( f(n) \) is a member of the set \( \Omega(g(n)) \) if and only if there exists constants \( c \) and \( N \) such that
\[
  0 \leq c g(n) \leq f(n),
\]
for all \( n \in \mathbb{N} \)

**Big-Theta**

**Definition:** \( f(n) \) is a member of the set \( \Theta(g(n)) \) if and only if
\[
  f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))
\]
Worst-Case/Expected-Case Bounds

- We can’t determine time bounds for all possible inputs of size n.
- Simplifying assumption #4: Determine number of steps for either
  - worst-case or
  - expected-case
- Worst-case
  - Determine how much time is needed for the worst possible input of size n
- Expected-case
  - Determine how much time is needed on average for all inputs of size n

Our Simplifying Assumptions

1. Use the size of the input rather than the input itself
2. Count the number of “basic steps” rather than computing exact times
3. Multiplicative constants aren’t important
4. Determine number of steps for either
   - worst-case or
   - expected-case

Worst-Case Analysis of Searching

- Linear Search (check each element)
  ```java
  static boolean find(int[] a, int item) {
    for (int i = 0; i < a.length; i++) {
      if (a[i] == item) return true;
    }
    return false;
  }
  ```
  For Linear Search, worst-case time is $O(n)$
  For Binary Search, worst-case time is $O(\log n)$

- Binary Search
  ```java
  static boolean find(int[] a, int item) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      if (a[mid] < item)
        low = mid + 1;
      else if (item < a[mid])
        high = mid - 1;
      else return true;
    }
    return false;
  }
  ```
  Analysis of Matrix Multiplication

- Code for multiplying $n$-by-$n$ matrices A and B:
  ```java
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
  ```
  By convention, matrix problems are measured in terms of $n$, the number of rows and columns
  - Note that the input size is $2n^2$
  - Worst-case time is $O(n^3)$
  - Expected-case time is also $O(n^3)$