What Makes a Good Algorithm?

- Suppose you have two possible algorithms or data structures that basically do the same thing; which is better?
- Well... what do we mean by better?
  - Faster?
  - Less space?
  - Easier to code?
  - Easier to maintain?
  - Required for homework?
- How do we measure time and space for an algorithm?

Sample Problem: Searching

- Determine if a sorted array of integers contains a given integer
- First 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;
}
```

- Second solution: 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 (a[mid] > item) 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
- Which inputs do we use?

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, count number of basic steps on the branch that is executed
- For a loop, count number of basic steps in loop body times the number of iterations
- For a method, count number of basic steps in method body (including steps needed to prepare stack-frame)
Runtime vs Number of Basic Steps

- But is 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, in a way
  - But the number of basic steps is proportional to the actual runtime

Using Big-O to Hide Constants

- We say \( f(n) \) is order of \( g(n) \) if \( f(n) \) is bounded by a constant times \( g(n) \)
- Notation: \( f(n) = O(g(n)) \)
- Roughly, \( f(n) = O(g(n)) \) means that \( f(n) \) grows like \( g(n) \) or slower, to within a constant factor
- "Constant" means fixed and independent of \( n \)

Formal definition:

\[
\text{if } f(n) = O(g(n)) \text{ then there exist constants } c \text{ and } N \text{ such that for all } n \geq N, f(n) \leq c \cdot g(n)
\]

A Graphical View

- To prove that \( f(n) = O(g(n)) \):
  - Find an \( N \) and \( c \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq N \)
  - We call the pair \( (c, N) \) a witness pair for proving that \( f(n) = O(g(n)) \)

Big-O Examples

- Let \( f(n) = 3n^2 + 6n - 7 \)
  - \( f(n) \) is \( O(n^2) \)
  - \( f(n) \) is \( O(n^3) \)
  - \( f(n) \) is \( O(n^4) \)
  - ...
- \( g(n) = 4n \log n + 34n - 89 \)
  - \( g(n) \) is \( O(n \log n) \)
  - \( g(n) \) is \( O(n^2) \)
- \( h(n) = 20 \cdot 2^n + 40n \)
  - \( h(n) \) is \( O(2^n) \)
- \( a(n) = 34 \)
  - \( a(n) \) is \( O(1) \)

\( \Rightarrow \) Only the leading term (the term that grows most rapidly) matters

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></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>Time Bound</th>
<th>Complexity</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>constant</td>
<td>excellent</td>
</tr>
<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>n log n</td>
<td>pretty good</td>
</tr>
<tr>
<td>O(n^2)</td>
<td>quadratic</td>
<td>often 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>

**Worst-Case/Expected-Case Bounds**

- We can’t possibly determine time bounds for all possible inputs of size n
- Simplifying assumption #4: Determine number of steps for either
  - 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**

- Use the size of the input rather than the input itself – n
- Count the number of “basic steps” rather than computing exact times
- Multiplicative constants and small inputs ignored (order-of, Big-O)
- Determine number of steps for either
  - worst-case
  - expected-case
  - These assumptions allow us to analyze algorithms effectively and easily

**Worst-Case Analysis of Searching**

**Linear Search**

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

worst-case time = O(n)

**Binary Search**

```java
public 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 (a[mid] > item)
            high = mid - 1;
        else return true;
    }
    return false;
}
```

worst-case time = O(log n)

**Comparison of Algorithms**

- Linear Search vs. Binary Search
  - Linear Search
    - Number of comparisons increases linearly with the number of items in the array.
  - Binary Search
    - Number of comparisons increases logarithmically with the number of items in the array.

**Comparison of Algorithms**

- Linear Search vs. Binary Search
  - Linear Search
    - Number of comparisons increases linearly with the number of items in the array.
  - Binary Search
    - Number of comparisons increases logarithmically with the number of items in the array.
Analysis of Matrix Multiplication

• Code for multiplying n-by-n matrices A and B:
  – By convention, matrix problems are measured in terms of n, the number of rows and columns
    • Note that the input size is really \( 2n^2 \), not \( n \)
  – Worst-case time is \( O(n^3) \)
  – Expected-case time is also \( O(n^3) \)

```c
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    for (k = 0; k < n; k++)
      C[i][j] += A[i][k]*B[k][j];
```

Remarks

• Once you get the hang of this, you can quickly zero in on what is relevant for determining asymptotic complexity
  – For example, you can usually ignore everything that is not in the innermost loop. Why?

• Main difficulty:
  – Determining runtime for recursive programs

Why Bother with Runtime Analysis?

• Computers are so fast these days that we can do whatever we want using just simple algorithms and data structures, right?
  • Well... not really – data-structure/algorithm improvements can be a very big win

  • Scenario:
    – \( A \) runs in \( n^2 \) msec
    – \( A' \) runs in \( n^2/10 \) msec
    – B runs in \( 10 \log n \) msec
    – Problem of size \( n=10^3 \)
      – \( A: 10^4 \) sec = 17 minutes
      – \( A': 10^3 \) sec = 1.7 minutes
      – B: \( 10^5 \) sec = 17 minutes
    – Problem of size \( n=10^5 \)
      – \( A: 10^7 \) sec = 30 years
      – \( A': 10^6 \) sec = 3 years
      – B: \( 2 \times 10^5 \) sec = 2 days
    – 1 day = 86,400 sec = 105 sec
    – 1,000 days = 3 years

Algorithms for the Human Genome

• Human genome
  – \( 3.5 \) billion nucleotides
  – \( \sim 1 \) Gb

  • @1 base-pair instructions/\( \text{sec} \)
    – \( n^2 \approx 3 \times 10^{14} \) years
    – \( n \log n \approx 3 \times 824 \) hours
    – \( n \approx 1 \) hour

Limitations of Runtime Analysis

• Big-O can hide a very large constant
  – Example: selection
  – Example: small problems

• The specific problem you want to solve may not be the worst case
  – Example: Simplex method for linear programming

• Your program may not be run often enough to make analysis worthwhile
  – Example: one-shot vs. every day
  – You may be analyzing and improving the wrong part of the program

• Should also use profiling tools

Summary

• Asymptotic complexity
  – Used to measure of time (or space) required by an algorithm
  – Measure of the algorithm, not the problem

• Searching a sorted array
  – Linear search: \( O(n) \) worst-case time
  – Binary search: \( O(\log n) \) worst-case time

• Matrix operations:
  – Note: \( n = \) number-of-rows = number-of-columns
  – Matrix-vector product: \( O(n^2) \) worst-case time
  – Matrix-matrix multiplication: \( O(n^3) \) worst-case time

• More later with sorting and graph algorithms