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

```java
/** return true iff v is in a */
static boolean find(int[] a, int v) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == v)
            return true;
    }
    return false;
}
```

Sample Problem: Searching

```java
/**
 * Determine if sorted array a contains integer v
 */
static boolean find(int[] a, int v) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == v)
            return true;
        if (a[mid] < v)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return false;
}
```

Linear Search vs Binary Search

- Which one is better?
  - Linear: easier to program
  - Binary: faster... isn’t it?
- How do we measure speed?
  - Experiment?
  - Proof?
  - What inputs do we use?
- Simplifying assumption #1: Use size of input rather than input itself
- For sample search problem, input size is n+1 where n is array size
- Simplifying assumption #2: Count number of “basic steps” rather than computing exact times
- For conditional: number of basic steps on branch that is executed
- For loop: (number of basic steps in loop body) * (number of iterations)
- For method: number of basic steps in method body (include steps needed to prepare stack-frame)
Runtime vs Number of Basic Steps

- Is this cheating?
  - The runtime is not the same as number of basic steps
  - Time per basic step varies depending on computer, compiler, details of code...
- Well... yes, in a way
  - But the number of basic steps is proportional to the actual runtime
- As n gets large, multiplicative constants become less important
- Simplifying assumption #3: Ignore multiplicative constants

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: $f(n)$ is $O(g(n))$
- Roughly, $f(n)$ is $O(g(n))$ means that $f(n)$ grows like $g(n)$ or slower, to within a constant factor
- "Constant" means fixed and independent of $n$

Formal definition: $f(n)$ is $O(g(n))$ if there exist constants $c$ and $N$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

A Graphical View

To prove that $f(n)$ is $O(g(n))$:
- Find $N$ and $c$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$
- Pair $(c, N)$ is a witness pair for proving that $f(n)$ is $O(g(n))$

Big-O Examples

- Claim: $100n + \log n$ is $O(n)$
  - We know $\log n \leq n$ for $n \geq 1$
  - So $100n + \log n \leq 101n$ for $n \geq 1$
  - By definition, $100n + \log n$ is $O(n)$ for $c = 101$ and $N = 1$

- Claim: $\log_B n$ is $O(\log_A n)$
  - Since $\log_B n = \frac{\log_A n}{\log_A B}$
  - Question: Which grows faster: $n$ or $\log n$?

Big-O Examples

Let $f(n) = 3n^2 + 6n - 7$
- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^2)$
- $f(n)$ is $O(n^4)$
- $f(n)$ is $O(n^2)$

$g(n) = 4n \log n + 34n - 89$
- $g(n)$ 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 $O(1)$

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

Problem-Size Examples

- Consider 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>Type</th>
<th>Evaluation</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>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 or
• expected-case

Worst-Case Analysis of Searching

**Linear Search**
```java
/** return true iff v is in a */
static boolean find(int[] a, int v) {
    for (int x : a) {
        if (x == v) return true;
    }
    return false;
}
worst-case time: O(n)
```

**Binary Search**
```java
static boolean find(int[] a, int v) {
    int low = 0;
    int high = a.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == v) return true;
        if (a[mid] < v) 
            low = mid + 1;
        else   high = mid - 1;
    }
    return false;
}
worst-case time: O(log n)
```

Comparison of Algorithms

Linear vs. Binary Search

<table>
<thead>
<tr>
<th>Number of Items in Array</th>
<th>Max Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>1000</td>
<td>50</td>
</tr>
<tr>
<td>2000</td>
<td>100</td>
</tr>
<tr>
<td>4000</td>
<td>200</td>
</tr>
</tbody>
</table>

Comparison of Algorithms

Linear vs. Binary Search

- Linear Search
- Binary Search

<table>
<thead>
<tr>
<th>Number of Items in Array</th>
<th>Max Number of Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>800</td>
<td>800</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>
Comparison of Algorithms

Analysis of Matrix Multiplication

Multiply \( n \times n \) matrices \( A \) and \( B \):

Convention, matrix problems measured in terms of \( n \), the number of rows, columns

- Input size is really \( 2n^2 \), not \( n \)
- Worst-case time: \( O(n^3) \)
- Expected-case time: \( O(n^3) \)

```c
for (i = 0; i < n; i++)
for (j = 0; j < n; j++) {
    c[i][j] = 0;
    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

- 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 so fast that we can do whatever we want using simple algorithms and data structures, right?

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 \( 10 n \log n \) msec

Problem of size \( n=10^3 \)

- \( A \): \( 10^5 \) sec \( \approx \) 17 minutes
- \( A' \): \( 10^4 \) sec \( \approx \) 1.7 minutes
- \( B \): \( 10^3 \) sec \( \approx \) 17 minutes

Problem of size \( n=10^6 \)

- \( A \): \( 10^8 \) sec \( \approx \) 30 years
- \( A' \): \( 10^7 \) sec \( \approx \) 3 years
- \( B \): \( 2 \times 10^6 \) sec \( \approx \) 2 days

1 day = 86,400 sec \( \approx \) \( \frac{10^5}{3600} \) sec
1,000 days \( \approx \) 3 years

Algorithms for the Human Genome

- Human genome = 3.5 billion nucleotides \( \sim 1 \) Gb
  - \( @1 \) base-pair instruction/\( \sec \)
  - \( n^2 \to 388445 \) years
  - \( n \log n \to 30.824 \) hours
  - \( n \to 1 \) 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
- Very common situation
- Should 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: $n \equiv$ number-of-rows $\equiv$ 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.