Readings, Homework

- Textbook: Chapter 4
- Homework:
  - Recall our discussion of linked lists from two weeks ago.
  - What is the worst case complexity for appending N items on a linked list? For testing to see if the list contains X? What would be the best case complexity for these operations?
  - If we were going to talk about O() complexity for a list, which of these makes more sense: worst, average or best-case complexity? Why?

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 sorted array b contains integer v
- First solution: Linear Search (check each element)

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

Doesn't make use of fact that b is sorted.

```java
static boolean find(int[] b, int v) {
    for (int x : b) {
        if (x == v) return true;
    }
    return false;
}
```

Second solution: Binary Search

Keep true: all occurrences of v are in b[low..high]

```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;
}
```
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 where n is array size

- Simplifying assumption #2:
  - Count number of “basic steps” rather than computing exact times

- One Basic Step = One Time Unit

  - 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

- 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 ≥ N, f(n) ≤ c · g(n)

A Graphical View

- To prove that f(n) is O(g(n)):
  - Find N and c such that f(n) ≤ c · g(n) for all n > N
  - Pair (c, N) is a witness pair for proving that f(n) is O(g(n))

Big-O Examples

- Claim: 100 n + log n is O(n)
  - We know log n ≤ n for n ≥ 1
  - So 100 n + log n ≤ 101 n for n ≥ 1
  - So by definition, 100 n + log n is O(n) for c = 101 and N = 1

- Claim: log_b n is O(log_a n)
  - since log_b n = (log_b A) log_a n

- 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 \( O(n^2) \)
- \( f(n) \) is \( O(n^2) \)

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

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>( n )</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>Constant</td>
</tr>
<tr>
<td>( O(n) )</td>
<td>Linear</td>
</tr>
<tr>
<td>( O(n\log n) )</td>
<td>( n \log n )</td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>Quadratic</td>
</tr>
<tr>
<td>( O(n^3) )</td>
<td>Cubic</td>
</tr>
<tr>
<td>( O(2^n) )</td>
<td>Exponential</td>
</tr>
</tbody>
</table>

Worst-Case/Expected-Case Bounds

May be difficult to determine time bounds for all imaginable inputs of size \( n \)
- **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 \( n \)

### Simplifying Assumptions

- Use the size of the input rather than the input itself – \( n \)
- Count the number of "basic steps" rather than computing exact time
- Ignore multiplicative constants and small inputs (order-of, big-O)
- Determine number of steps for either
  - worst-case
  - expected-case

These assumptions allow us to analyze algorithms effectively

### Simplifying assumption #4

Determine number of steps for either
- worst-case or
- expected-case or
- average case

### Worst-Case Analysis of Searching

#### Linear Search

```java
// return true if v is in b
static bool find (int[] b, int v) {
    for (int x : b) {
        if (x == v) return true;
    }
    return false;
}
```

**worst-case time:** \( O(n) \)

#### Binary Search

```java
// return h that satisfies
// b[0..h] <= v < b[h+1..]
static bool bsearch(int[] b, int v) {
    int h = -1; int t = b.length;
    while ( h <= t - 1 ) {
        int e = (h+t)/2;
        if (b[e] <= v) h = e;
        else t = e;
    }
}
```

**Always takes \(~(log n+1)\) iterations.**

Worst-case and expected times:
\( O(\log n) \)
Comparison of linear and binary search

![Linear vs. Binary Search](image)

Analysis of Matrix Multiplication

Multiply n-by-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?

One difficulty:
- Determining runtime for recursive programs
  - Depends on the depth of recursion

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 $n^5/10$ msec
- $B$ runs in $10n \log n$ msec

Problem of size $n=10^3$
- $A$: $10^5$ sec $\approx$ 17 minutes
- $A'$: $10^5$ sec $\approx$ 1.7 hours
- $B$: $10^6$ sec $\approx$ 1.7 minutes

Problem of size $n=10^6$
- $A$: $10^9$ sec $\approx$ 30 years
- $A'$: $10^9$ sec $\approx$ 3 years
- $B$: $10^9$ sec $\approx$ 2 days

1 day $= 86,400$ sec $= 10^5$ sec
1,000 days $= 3$ years

Algorithms for the Human Genome

Human genome $= 3.5$ billion nucleotides $\sim 1$ Gb

@1 base-pair instruction/msec
- $n^2$ $\approx$ 368,445 years
- $n \log n$ $\approx$ 30,824 hours
- $n$ $\approx$ 1 hour
### Limitations of Runtime Analysis

<table>
<thead>
<tr>
<th>Limitation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Big-O can hide a very large constant</td>
<td>Example: selection</td>
</tr>
<tr>
<td></td>
<td>Example: small problems</td>
</tr>
<tr>
<td>The specific problem you want to solve may not be the worst case</td>
<td>Example: Simplex method for linear programming</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Your program may not be run often enough to make analysis worthwhile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example: one-shot vs. every day</td>
</tr>
<tr>
<td></td>
<td>You may be analyzing and improving the wrong part of the program</td>
</tr>
<tr>
<td></td>
<td>Very common situation</td>
</tr>
<tr>
<td></td>
<td>Should use profiling tools</td>
</tr>
</tbody>
</table>

### 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: $n = \text{number-of-rows} = \text{number-of-columns}$
  - 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