Announcements

- The TA staff is (as of today) full assembled. Expect an OH schedule by end of this week.
- Programming Project 1 is on-going, I recommend starting if you haven't!
- Quiz 1 and HW 1 have been graded and released. Remember to submit any regrade requests before next Tuesday (as per syllabus)
- On that note…

Topic 2: Algorithm Analysis

By: Professor Lynam

What are some algorithms you know already?

Desirable Algorithm Characteristics

- A good algorithm…
- Produces the correct answer for all legal inputs
- Solves the problem efficiently (time & storage)
- Has reproducible output (it’s not random)
- Doesn’t enter infinite loops!
- Etc.

Problem Size

- Definition: *Instance Characteristic*
- *The instance characteristic(s) of a problem is/are the size(s) of the problem*

- For example:
- What would the instance characteristic be for sorting a list?
  - *n (the number of elements in the list)*
- How about for an in-order traversal of a binary tree?
  - The number of nodes in the tree
- Squaring a 2D matrix?
  - The number of rows *and* the number of columns! (2 instance characteristics)

Algorithm Speed

- Why bother investigating the speed of an algorithm?
  - To compare its speed to that of other algorithms
- There are two primary approaches to measuring algorithm speed:

1. Execute and time the algorithm
   - Problems with this approach?
   - Have to code, test, and debug the algorithm before it's ready to be timed!
   - Also, we're measuring wall-clock time, but the algorithm is running on CPU time, which is also running the OS, other programs, etc…
2. Count the operations the algorithm performs
   - Tedious, and possibly not useful (assumes each operation takes the same amount of time)… But can help us find a more general answer for algorithm efficiency

Step-Counting/Operation Counting

1. Augment the algorithm with operation counts
2. Estimate executions of selections & iterations
3. Remove/Ignore the algorithm’s statements
4. Produce a step-count expression in terms of the algorithm’s instance characteristics

- What do we end up with? A course estimate of the algorithm’s efficiency

Step-Counting Example 1

```
double sum = 0;
for (int i=0; i<n; i++) {
        sum = sum + list[i];
}
mean = sum / n;
```

- Before we start, we need to know something important: what is this algorithm’s instance characteristic? Is there more than one?
  - Just one, the amount of data in the array, *n*

Step-Counting a For Loop

- Our example has a for loop in it. So we need to know how to step-count one!
- First, imagine we initialize an operation counter (o = 0;).
- Second, imagine we augment the code with o++; statements to count the loop's operations.

```
o = 0;  # initialize the operation counter
o++;   # initialize the for loop
for (initialization; condition; increment) {
            o++; # this is for the conditional evaluation if it evaluates to true
            loop body # we'll worry this later
            o++; # this is for the increment of the for loop
}
o++ # this is for the conditional evaluation if it evaluates to false
```

Step-Counting a For Loop

- So, for a for loop, we need to count:
  - The initialization expression before the loop
  - True evaluations of the loop condition *inside* the loop body
  - False evaluations of the loop condition *after* the loop
  - The increment expression at the end of the loop body
- What about while or do-while loops? Same theory as for loops, but take out the steps that don’t make sense!

Step-Counting Example 1

```
double sum = 0;
for (int i=0; i<n; i++) {
        sum = sum + list[i];
}
mean = sum / n;
```

- Note: these are steps 1 and 2 of the process:

1. Augment the algorithm with operation counts
2. Estimate executions of selections & iterations

```
o = 0;
double sum = 0;
o++; # assignment of variable
o++; # loop initialization
for (int i=0; i<n; i++) {
        o++; # true loop condition evaluation
        sum = sum + list[i];
        o++; o++; o++; o++; # 1 for assignment, 2 for +, 3 and 4 for array
location calculation
        o++; # loop increment
}
o++; # false loop condition evaluation
mean = sum / n;
o++; o++; # 1 for assignment, 2 for division
```

Step-Counting Example 1

- Step 3: Remove/Ignore the algorithm's statements (but retain loop information!

```
o = 0;
double sum = 0;
o++; # assignment of variable
o++; # loop initialization
for (int i=0; i<n; i++) {
        o++; # true loop condition evaluation
        sum = sum + list[i];
        o++; o++; o++; o++; # 1 for assignment, 2 for +,
3 and 4 for array location calculation
        o++; # loop increment
}
o++; # false loop condition evaluation
mean = sum / n;
o++; o++; # 1 for assignment, 2 for division
```

```
o = 0;
o++; # assignment of variable
o++; # loop initialization
Iterate n times:
        o++; # true loop condition evaluation
        o++; o++; o++; o++; # 1 for assignment, 2 for +, 3 and
4 for array location calculation
        o++; # loop increment
o++; # false loop condition evaluation
o++; o++; # 1 for assignment, 2 for division
```

Step-Counting Example 1

- Step 4: Produce a step-count expression in terms of the algorithm’s instance characteristic(s)
- What it be for this step count example?
- Answer: 6n + 5

```
o = 0;
o++; # assignment of variable
o++; # loop initialization
Iterate n times:
        o++; # true loop condition evaluation
        o++; o++; o++; o++; # 1 for assignment, 2 for +, 3 and 4 for array location calculation
        o++; # loop increment
o++; # false loop condition evaluation
o++; o++; # 1 for assignment, 2 for division
```

How to Step-Count an If Statement

- Three possible approaches:

1. Optimistic: the body of the if statement is *never* executed!
2. Pessimistic: the body of the if statement is *always* executed!
3. Estimation: estimate what percent of the time the if body is executed, and make your step-count expression based on that

- What would be the safest choice?
- Pessimistic

How to Step-Count an If-Else Statement

- We can’t logically do both the if *and* the else bodies, we have to choose one. The question is, which one should we choose?
- Being pessimistic, we should choose the statement body that has the most steps! It’s the same for switch statements, and other mutually exclusive conditional statements

Step-Counting Example 2

```
double min, max;
min = max = list[0];
for (int i=1; i<n; i++) {
        if (list[i] < min):
                min = list[i];
        if (list[i] > max):
                max = list[i];
}
```

```
o = 0;
double min, max;
min = max = list[0];
o++ * 4; # two assignments, one array calculation
o++; # for loop initialization
for (int i=1; i<n; i++) {
        o++; # true eval of loop condition
        if (list[i] < min):
                min = list[i];
                 o++ * 4; # 1 assignment, one comparison, 1 array calc
                # Note, we only do an array calc once per loop body
        if (list[i] > max):
                max = list[i];
                 o++ * 2; # 1 assignment, one comparison
         o++; # for loop increment
}
o++;  # false eval of loop condition
```

Step-Counting Example 2

```
o = 0;
double min, max;
min = max = list[0];
o++ * 4; # two assignments, one array calculation
o++; # for loop initialization
for (int i=1; i<n; i++) {
        o++; # true eval of loop condition
        if (list[i] < min):
                min = list[i];
                 o++ * 4; # 1 assignment, one comparison, 1
array calc
        # Note, we only do an array calc once per loop body
        if (list[i] > max):
                max = list[i];
                 o++ * 2; # 1 assignment, one comparison
         o++; # for loop increment
}
o++;  # false eval of loop condition
```

```
o = 0;
 o++ * 4; # two assignments, one array calculation
o++; # for loop initialization
Iterate n – 1 times:
        o++; # true eval of loop condition
        o++ * 4; # 1 assignment, one comparison, 1
array calc
        o++ * 2; # 1 assignment, one comparison
         o++; # for loop increment
}
o++;  # false eval of loop condition
```

What step-counting expression will this produce?

$8(n-1) + 6 = 8n - 2$

Step-Counting Example 3

```
int low = 0, high = n;
int i;
for (i=0; i<n-1; i +=2) {
        if (list[i] < list[i+1]) {
                candidates[low++] = list[i];
                candidates[high--] = list[i+1];
        } else {
                candidates[low++] = list[i+1];
                candidates[high--] = list[i];
        }
}
if (i == n-1) {
        candidates[low++] = list[i];
        candidates[high--] = list[i];
}
min = candidates[0];
for (int j = 1; j < low; j++) {
                if (candidates[j] < min):
                        min = candidates[j];
}
```

```
max = candidates[high+1];
for (int k = high +2; k < n+1; k++) {
        if (candidates[k] > max):
                max = candidates[k];
}
```

Try to step count this own on your own later for practice!
A few quick questions: which of the if else statements in the 1st for loop would we count?
What is the first for loop doing?
What is the second and third for loops doing?
What's the point of the if statement between the two?
The answer for the step counting expression is: 15.5n +36.5

Step-Counting Example 4

```
int min = max = list[0];
for (int i=1; i<n-1; i +=2) {
        if (list[i] < list[i+1]) {
                if (list[i] < min):
                        min = list[i];
                if (list[i+1] > max):
                        max = list[i+1];
        } else {
                if (list[i] < min):
                        min = list[i+1];
                if (list[i+1] > max):
                        max = list[i];
        }
}
if (i == n-1) {
        if (list[i] < min):
                min = list[i];
        if (list[i] > max):
                max = list[i];
}
```

Let's try this one out!
What's the count for everything before the loop?
Answer: 5
What's the count for everything after the loop?
Answer: 9
What's the count for the for loop?
Answer: 14(n-2)/2
The answer for the step counting expression is:
7n

Key Comparisons

- Definition: *Key Comparison*
- *A key is a value that identifies an element in a data structure.*
- *A key comparison is any condition involving one or more keys.*
- For example: list[i] < min is a key comparison, i < n-1 is not a key comparison

Key Comparisons

- If we just used key comparisons for our earlier three examples:
- Version 1 had 8n – 2 operations, and 2n key comparisons
- Version 2 had 15.5n + 36.5 operations, and ~1.5n key comparisons
- Version 3 had 7n+7 operations, and ~1.5n key comparisons
- We can see that key comparison approximations can be pretty different from overall step-counts

Code Profiling

- Definition: *Code Profiling*
- *Code profiling is the augmentation of programs to aid the complexity analysis of execution (time) requirements and memory (space) usage*
- For Java, a free code profiler is the JDK Flight Recorder (jfr)
- jfr's sampling frequency isn't very high, meaning the results are coarse
- Commercial code profilers are more sophisticated (can do statement-level profiling)
- Instrumenting code does slow it down, and can mess with the results
- A disadvantage of any of these code profilers… they require code!

Execution Timing

- Another option for checking efficiency is by timing the code execution
- Some disadvantages to this:
    - Wall-clock time vs CPU time
    - Precision versus resolution
    - Some programs take very long to run
    - Code profilers usually have extra features with more detail than just time
    - And of course, just like code profilers… you still have to write the code!

```
start = System.nanoTime();
// code to time
seconds = (System.nanoTime() – start) / 1000000000;
```

Questions about an Algorithm

- What do we want to know about an algorithm’s efficiency before we adopt it?
    - Will it find correct answers in a reasonable amount of time?
    - Is it better than any other algorithm we’re considering? (Better under what circumstances?)
    - How much RAM (or disk space!) does this algorithm need?
    - How much slower will it be if the problem size doubles?

Questions about an Algorithm

- Given those questions, we know what we’re trying to determine about an algorithm:
  - First, can we answer those questions *without* having to code the algorithm and test it on sample data?
  - How can we communicate the answers to those questions to other people?
  - Can we do this communication clearly, using math?

Asymptotic Analysis

- Definition: *Asymptotic Analysis*
- *The determination of limitations of growth rates of functions that describe the amount of work performed (or memory required) by algorithms.*
- Step-counting was a pain! But it did result in something very useful: a function that uses our instance characteristic(s)!

“Big-O” Notation

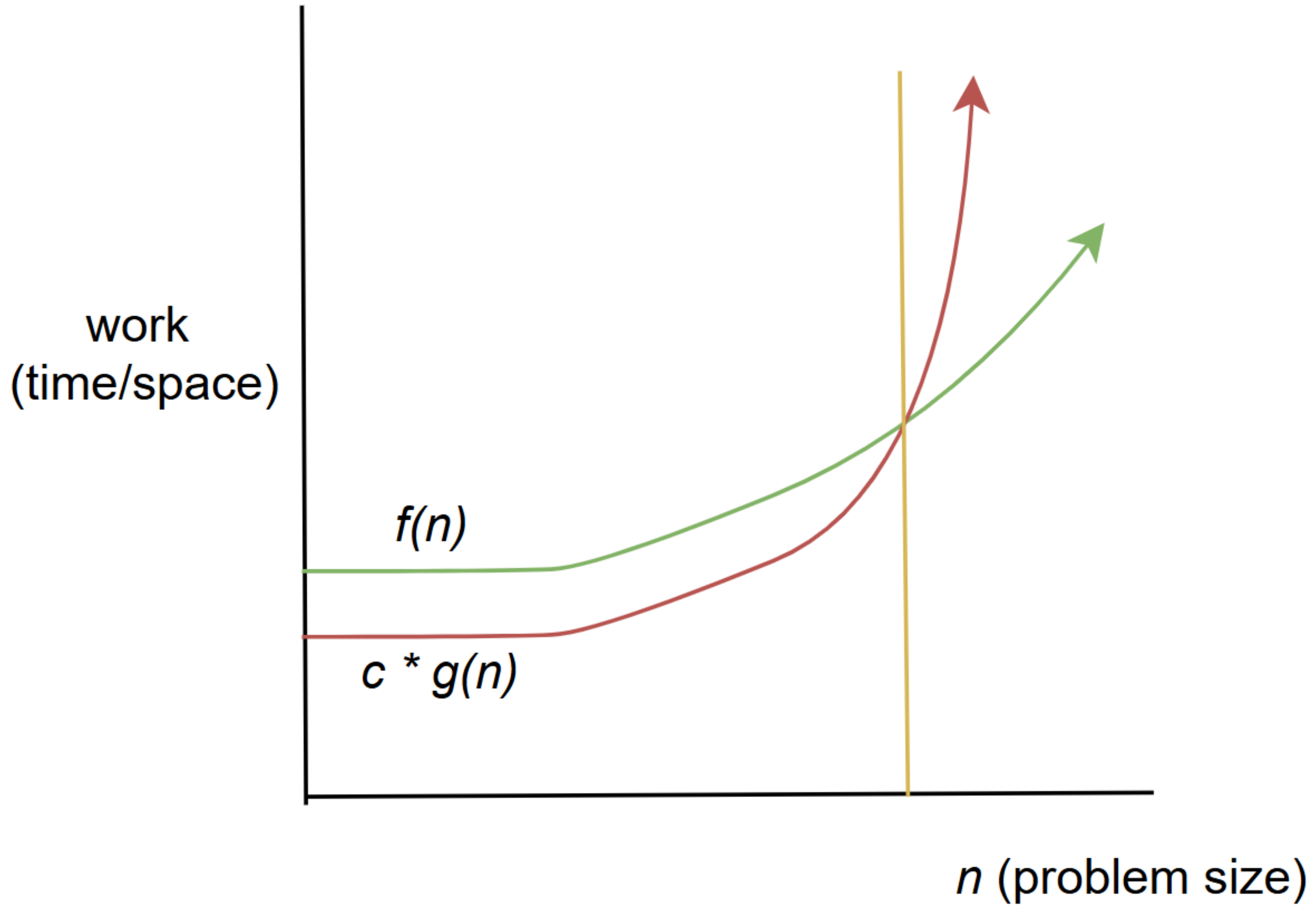
- The idea: Have an approximate function growth rate notation
- The “O” stands for Ordnung (German for “Order”)
- For example, instead of saying “f(x) is approximately $\frac{n}{2}$,” we would say, “f(x) is on the order of $\frac{n}{2}$.”
- When writing about this, the notation is: $f(x) \in O(\frac{n}{2})$ or $f(x) \text{ is } O\left(\frac{n}{2}\right)$, but never $f(x) = O(\frac{n}{2})$

“Big-O” Notation

- Definition: *“Big-O” Notation*
- *Let f(n) and g(n) be functions that map positive integers to positive reals. We say that f(n) is O(g(n)) if there exists a real constant c > 0 and there exists an integer constant* $n_0 \geq 1$ *such that* $f(n) \leq c * g(n)$ *for every integer* $n \geq n_0$ *.*

“Big-O” Notation

- What does the orange line represent here?
- Answer: n_0



“Big-O” Notation Example

- Example: Show that 7n + 8 is O(n).
- We need constants $c > 0$ and $n_0 \geq 1$ such that $\mathrm{f(n)} \leq c \ * g(n)$ for every integer $\mathrm{n} \geq n_0$
- $f(n) = 7n + 8$ and $\mathrm{g}(n) = n$
- We need to force c*g(n) to grow faster than $7n + 8$
- Let’s choose c = 8
- As for n_0, $7n + 8 \leq 8n$, or $n \geq 8$, so let’s choose $n_0 = 8$
- Great, all done!
- Just kidding; how do you know that these values actually work? How can we *prove* it to someone who doesn’t believe us?

“Big-O” Notation Example

- Conjecture: $7n + 8 \leq 8n, \forall n \geq 8$
- Proof (by weak induction):
- Basis step: Let n = 8. 7(8)+8 = 8 * 8, $64 \leq 64$, so the basis step holds true.
- Inductive step: If $7n + 8 \leq 8n$, then we must show that $7(n + 1) + 8 \leq 8(n + 1)$
- $7(n + 1) + 8 = 7n + 7 + 8$ (by distributivity of multiplication)
- $7n + 7 + 8 = 7n + 8 + 7$ (by commutativity of addition)
- $7n + 8 + 7 \leq 8n + 7$ (by the inductive hypothesis)
- $8n + 7 \leq 8n + 8$ and $8n + 8 = 8(n + 1)$
- This shows that $7(n + 1) + 8 \leq 8(n + 1)$.
- Therefore, $7n + 8 \leq 8n, \forall n \geq 8$ Q.E.D.

“Big-O” Notation Example (Round 2)

- Conjecture: $7n + 8 \leq 8n, \forall n \geq 8$
- Proof (by weak induction):
- Basis step: Let n = 8. 7(8)+8 = 8 * 8, $64 \leq 64$, so the basis step holds true.
- Inductive step: If $7n + 8 \leq 8n$, then we must show that $7(n+1) + 8 \leq 8(n+1)$
- $7(n) + 8 = 8n$ (by the inductive hypothesis)
- $7n + 8 + 7 \leq 8n + 7$
- $7(n+1) + 8 \leq 8n + 7$
- $7(n+1) + 8 \leq 8n + 8$
- $7(n+1) + 8 \leq 8(n+1)$
- Therefore, $7n + 8 \leq 8n, \forall n \geq 8$ Q.E.D.

“Big-O” Notation

- “But wait!” you say, disgusted by this fallacy of logic. “n is not an upper-bound to 7n, what do you mean 7n is O(n)?!”
- Remember, the definition of Big-O means that we were allowed to pick 8n as the upper bound—not just 1*n
- Plus, only when n was greater than 8 (or some other large number)
- The big picture here is that 7n+8 and n are both *linear* functions, so they belong to the same order of functions

“Big-O” Notation

- “Well, okay…” you say reluctantly, searching for a way to trip your professor up. “But what about O(n^2)? By the definition, isn’t 7n+8 O(n^2) too?”
- Well… yes. That’s true—and it’s not helpful information, since the f(n) and g(n) would be different classes of functions
- However! In practice, people use Big-O as shorthand for a tight upper-bound, not as a loose upper-bound.

“Big-O” Notation

Function	Common Name	Example Algorithm
1	Constant	Array Reference
$log_2 n$	Logarithmic	Binary Search
n	Linear	Linear Search
$n * log_2 n$	Log-Linear	Mergesort
n^2	Quadratic	Bubblesort
n^3	Cubic	Matrix Multiplication
n^k	Polynomial	Essentially all of the above
2^n	Exponential	Traveling Salesperson
$n!$	Factorial	Sub-graph isomorphism

Notes: Polynomial-time algorithms are generally “efficient” (but the exponent does matter)
Why log base 2, not log base 10? Log base 2 grows faster

A Useful “Big-O” Theorem

- Conjecture: $\mathrm{f}(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$, then f(n) is O($n^m$).
- Proof (direct):
- First, restate f(n)
- $f(n) = \sum_{i=0}^{m} a_i n^i$
- Now, let’s force the a_i values to be positive: $f(n) \le \sum_{i=0}^{m} |a_i| n^i$
- Dividing through by n^m exposes it: $f(n) \le n^m (\sum_{i=0}^{m} |a_i \Big| n^{i-m}\Big)$
- Because $\frac{|a_i|}{n^k} \le |a_i|$ when k is positive:
- $f(n) \le n^m\ (\sum_{i=0}^{m} |a_i|)\ , for\ n \ge 1$

A Useful “Big-O” Theorem

- Conjecture: $\mathrm{f}(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$, then f(n) is O($n^m$).
- Proof (direct, continued):
- Thus $\mathrm{f}(n) \leq\ n^m\,(\sum_{i=0}^{m} |a_i|)$. This has the form of $\mathrm{f}(n) \leq c * g(n)$.
- By the definition of Big-O, $\mathrm{f}(n)$ is O(n^m). QED

A Useful “Big-O” Theorem Example

- Conjecture: $\mathrm{f}(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0$, then f(n) is O($n^m$).
- Example: What is the Big-O of $\mathrm{f}(n) = 3n^2 - 2n + 6$?
- First, we want to get rid of that annoying negative. Fortunately…
- $3n^2 - 2n + 6 \leq 3n^2 + 2n + 6$
- $3n^2 + 2n + 6 = n^2\left(3 + \frac{2}{n} + \frac{6}{n^2}\right)$
- $n^2\left(3 + \frac{2}{n} + \frac{6}{n^2}\right) \leq (3 + 2 + 6) * n^2 \text{ if n} \geq 1$
- This has the form of $\mathrm{f}(n) \leq c * g(n)$.
- By the definition of Big-O, $\mathrm{f}(n)$ is O(n^2). QED

Flashback to Step-Counting

- Remember the min/max-finding algorithms we step-counted?
- Version 1: $\mathrm{f}(n) = 8n - 2$
- Version 2: $\mathrm{f}(n) = 15.5n - 36.5$
- Version 3: $\mathrm{f}(n) = 7n + 8$
- What can we say about all these versions, in terms of Big-O?
  - Answer: they are all linear, or O(n)
- So, if you just need a category of efficiency, Big-O is useful

Beyond Big-O

- There are some issues with Big-O: first, recall that the upper bound is loose, not tight, which can lead to correct (but not useful/precise) claims to be made about an algorithm
- To prevent that issue, and get a more precise categorization, we need a tight upper bound… *and* a tight lower bound!
- Without a tight lower bound, we don’t know the limitations of the algorithm, or how efficient it could *possibly* be

Little-o

- Definition: *Little-o (o())*
- *Let f(n) and g(n) be functions that map positive integers to positive reals. We say that f(n) is o(g(n)) if for any real constant c > 0, there exists an integer constant* $n_0 \geq 1$ *such that* $f(n) < c * g(n)$ *for every integer* $n \geq n_0$ *.*

- What are the differences between this definition and the Big-O definition?
  - Answer: instead of “there exists” a c, it’s “for any” c. And instead of $\leq$, it’s $<$

Little-o example

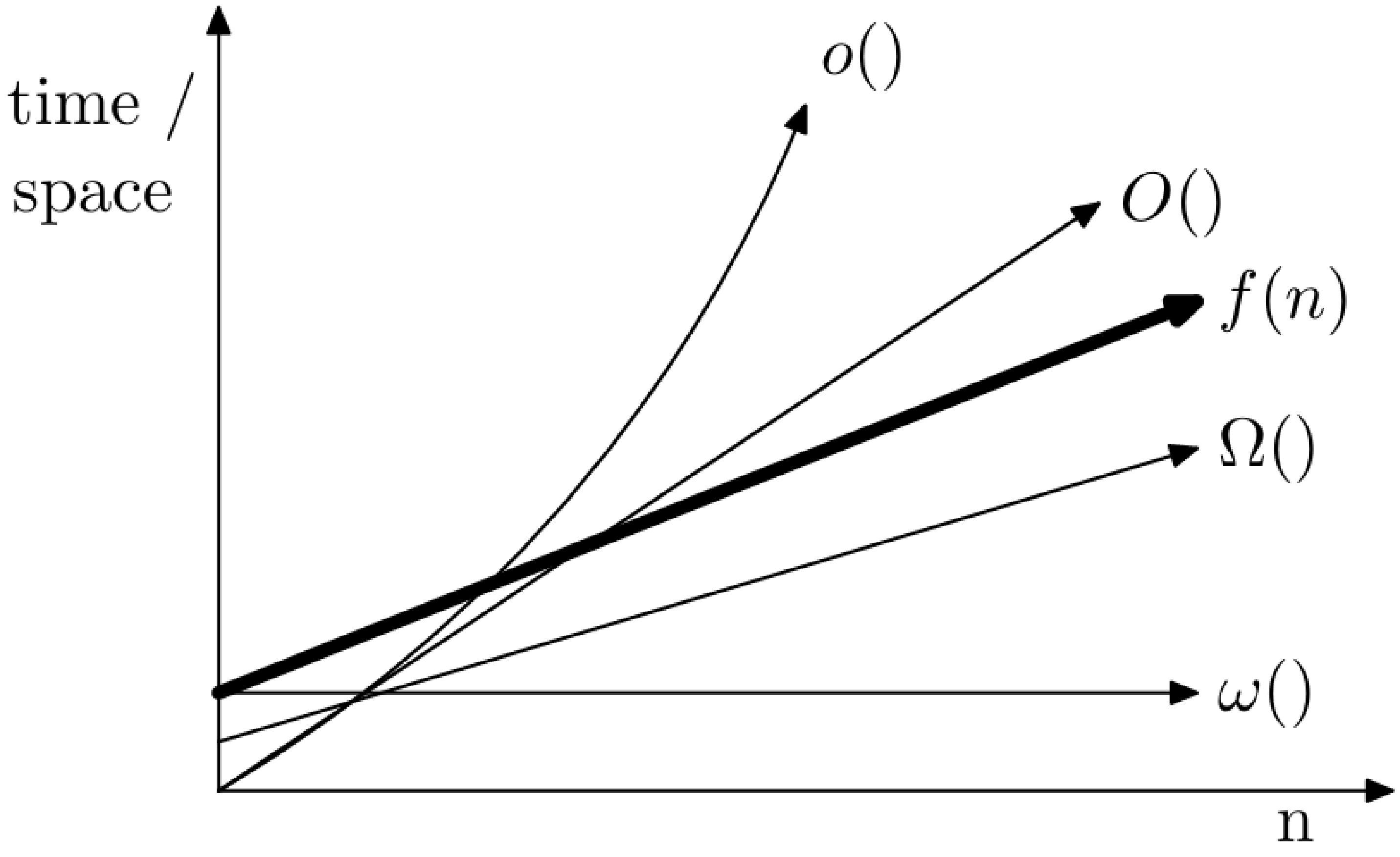
- Is : $7n + 8 \in o(n)$?
- By the definition, any c>0 *must* make f(n) < c * g(n)
- Let's try some c's!
- c = 100?
- 7n+8 < 100n for $n \geq 1$ seems to work!
- c = 1?
- 7n+8 < n for $n \geq 1$ doesn't seem to work…
- c = 1/100? Obviously doesn't work

Little-o example

- Is : $7n + 8 \in o(n^2)$?
- By the definition, any c>0 *must* make f(n) < c * g(n)
- Let's try some c's!
- c = 100?
- 7n+8 < $100n^2$ for $n \geq 1$ seems to work!
- c = 1?
- 7n+8 < n^2 for $n \geq 1$ doesn't seem to work… but n doesn't have to be 1. Maybe we pick n_0 to be 15 for this c: that would make this work.
- c = 1/100?
- What if we pick n_0 to be 1000? Or 1000000?
- This is (obviously) not a great proof… we'll get into that in a bit

What about lower bounds?

- We have our upper bounds, o() and O() already. The lower bound equivalents are big Omega Ω() and little Omega ω(), and they're essentially mirror images of the upper bounds



Big-Omega

- Definition: Big-Omega (Ω())
- *Let f(n) and g(n) be functions that map positive integers to positive reals. We say that f(n) is Ω(g(n)) if there exists a real constant c > 0 and there exists an integer constant* $n_0 \geq 1$ *such that* $f(n) \geq c * g(n)$ *for every integer* $n \geq n_0$.

- What are the differences between this definition and the Big-O definition?
  - Answer: instead of ≤, it's ≥

Little-Omega

- Definition: *Little-Omega (ω())*
- *Let f(n) and g(n) be functions that map positive integers to positive reals. We say that f(n) is ω(g(n)) if for any real constant c > 0, there exists an integer constant* $n_0 \geq 1$ *such that* $f(n) > c * g(n)$ *for every integer* $n \geq n_0$ *.*

- What are the differences between this definition and the Big-O definition?
  - Answer: instead of “there exists” a c, it’s “for any” c. And instead of $\leq$, it’s $>$

Big-Theta (Θ): The Big Squeeze

- Definition: *Big-Theta (Θ())*
- *Let f(n) and g(n) be functions that map positive integers to positive reals. We say that f(n) is Θ(g(n)) if and only if* $f(n) \in O(g(n))$ *and* $f(n) \in \Omega(g(n))$.

- This lets us *know* that f(n) is a particular class of function, since both the upper-bound and lower-bound are tight
- Which is the upper-bound, and which is the lower bound?
  - Answer: Big-Omega is the lower bound, Big-O is the upper bound

Wait, why are we using Big-O at all then?

- Two main reasons:
  - We use Big-O as a tight upper-bound, so its g() is the same as Big-Theta’s g()
  - Big-O is challenging enough to explain, and to get to Big-Theta you *also* need to understand the concept of big Omega
- Also (my opinion) Big-O sounds more interesting than big Theta

Pop Quiz!

- Definition: *Size-x (?())*
- *Let f(n) and g(n) be functions that map positive integers to positive reals. We say that f(n) is ?(g(n)) if ____ real constant c > 0, there exists an integer constant* $n_0 \geq 1$ *such that* $f(n)\_\_c * g(n)$ *for every integer* $n \geq n_0$.

- We have big-O, little-o, Big-Omega, little-Omega, and big-Theta. Replace the blanks in the generic definition above for each.
- Little-O: “for any” and $<$
- Big-O: “there exists” and $\leq$
- Big-Omega: “there exists” and $\geq$
- Little-Omega: “for any” and $>$
- Big Theta… doesn’t fit the generic definition above.

Properties of Asymptoticity

- **Symmetry of Θ**: $f(n) \in \Theta\ (g(n))$ iff g$(n) \in \Theta\ (f(n))$
- **Transpose Symmetry**:$f(n) \in \mathrm{O}\ (g(n))$ iff g$(n) \in \Omega\ (f(n))$ **and** $f(n) \in \mathrm{o}(g(n))$ iff g$(n) \in \omega(f(n))$
- **Reflexivity**: for X=O, Ω, and Θ, f(n) is big-X(f(n))
- Why doesn't reflexivity work for little-o or little-ω?
- **Transitivity**: for X=O, o, Ω, ω, and Θ, if f$(n) \in \mathrm{X}(g(n))$ and g$(n) \in \mathrm{X}(h(n))$, then f$(n) \in \mathrm{X}(h(n))$
- For example, $5\mathrm{n} + 4 \in \mathrm{o}(n(\log(n))$ and $\mathrm{nlog(n)} \in \mathrm{o}(n^2)$, so $5\mathrm{n} + 4 \in \mathrm{o}(n^2)$

Analyzing Subdivided Algorithms

- What if the algorithm we're looking at is divided into different parts? We can analyze each part, but how do we combine the results?
- **Theorem:** If $f_1(n)\ is\ O\left(g(n)\right)\ and f_2(n)\ is\ O\left(h(n)\right)$, then $f_1(n) + f_2(n)$ is $\max(O\left(g(n)\right), O\left(h(n)\right))$, and $f_1(n) * f_2(n)$ is $O(g(n) * h(n))$.
- Proof: Try it for yourself! Not a very difficult proof.
- In terms of what the '+' and '*' mean in this context, + is for *sequential execution*, * is for *iteration*

Explaining Big-O Using Limits

- Derivatives of Polynomials:
- If $f(n) = c * n^r$, then $f'(n) = c * r * n^{r-1}$
- For example, if $f(n) = 2n^2 - 6n + 3$, then $f'(n) = 4n - 6$
- L'Hôpital's Rule: If the limits of $f(n)$ and $\mathrm{g}(n)$ are $\in \{-\infty, 0, \infty\}$, and $\lim_{n->\infty} \frac{f(n)}{g(n)}$ exists, then $\lim_{n->\infty} \frac{f(n)}{g(n)} = \lim_{n->\infty} \frac{f\prime(n)}{g\prime(n)}$
- Defining Big-O with limits: If $\lim_{n->\infty} \frac{f(n)}{g(n)}$ = c, where $0 \leq \mathrm{c} < \infty$, then $f(n)$∈O(g(n))
- For example, for $f(n) = 7n + 8, \mathrm{g}(n) = n$
- $\lim_{n->\infty} \frac{f(n)}{g(n)} = \lim_{n->\infty} \frac{7n+8}{n} = \lim_{n->\infty} \frac{7}{1} = 7$

Explaining Big-O Using Limits

- There are similar definitions for the rest:
- O(): If $\lim_{n->\infty} \frac{f(n)}{g(n)}$ = c, where $0 \leq c < \infty$, then $f(n)$∈O(g(n))
- o(): If $\lim_{n->\infty} \frac{f(n)}{g(n)}$ = 0, then $f(n)$∈o(g(n))
- Ω(): If $\lim_{n->\infty} \frac{f(n)}{g(n)}$ = c, where $0 < c \leq \infty$, then $f(n)$∈Ω(g(n))
- ω(): If $\lim_{n->\infty} \frac{f(n)}{g(n)}$ = ∞, then $f(n)$∈ω(g(n))
- Θ(): If $\lim_{n->\infty} \frac{f(n)}{g(n)}$ = c, where $0 < c < \infty$, then $f(n)$∈O(g(n))

Analysis of Recursive Algorithms

- Step-Counting doesn't work well with recursion… but recurrence relations do!
- Computing the Fibonacci sequence recursively:
- $f_n = f_{n-1} + f_{n-2}$, where $f_0 = 0$ and $f_1 = 1$
- This is a: Linear Homogeneous Recurrence Relation with Constant Coefficients of Degree 2
- Recurrence relations describe the work performed by recursive algorithms (as opposed to polynomials, which describe the work performed by iterative algorithms)
- We can solve recurrence relations to produce equivalent closed-form expressions… and here's how!

Solving RRs using “Find the Pattern”

```
public long factorial (long n) {
        if (n == 0) return 1;
        else        return (n * factorial(n - 1));
}
```

- The recurrence describing the work performed is:
- F(0) = c   <- initial condition
- F(n) = F(n-1) + k   <- recurrence
- Why bother with k? Why not re-use c?
  - Answer: because it represents a different constant amount of work

Solving RRs using “Find the Pattern”

```
public long factorial (long n) {
        if (n == 0) return 1;
        else      return (n * factorial(n - 1));
}
```

- F(0) = c <- initial condition
- F(n) = F(n-1) + k <- recurrence
- Time to find the pattern:
- F(n-1) = F(n-2) + k
- F(n) = F(n-2) + k + k
- F(n-2) = F(n-3) + k
- F(n) = F(n-3) + k + k + k

Solving RRs using “Find the Pattern”

```
public long factorial (long n) {
        if (n == 0) return 1;
        else      return (n * factorial(n - 1));
}
```

- Now, we need to generalize the expression:
- F(n) = F(n-1) + k
- F(n) = F(n-2) + 2k
- F(n) = F(n-3) + 3k
- What does F(n) equal in general?
- F(n) = F(n-a) + a*k

Solving RRs using “Find the Pattern”

```
public long factorial (long n) {
        if (n == 0) return 1;
        else        return (n * factorial(n - 1));
}
```

- Recall: F(0) = c
- Question: In F(n) = F(n-a) +a*k, how large can a become? Asked another way: how small can n-a become?
- n-a = 0, so n=a
- F(n) = F(n-n) + n*k
- F(n-n) + n*k = F(0) + n*k = c + n*k
- F(n) = c + n*k   Which is Big-O of what?
- Answer: O(n)

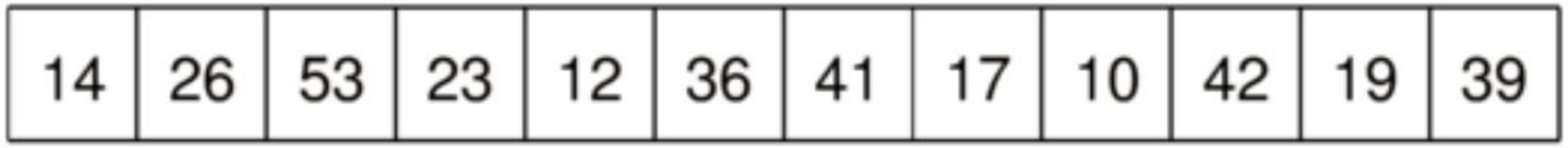
But Were Our Assumptions Correct?

- Conjecture: The solution to the recurrence F(n)=F(n-1) + k, with the initial condition F(0)=c, is F(n)=c+k*n.
- Proof (weak induction):
- Basis: n = 0. Initial condition F(0) = c. Conjecture: F(0) = c + k*0 = c.
- Inductive Step: If F(n) = c+k*n (IH), then F(n+1) = c+k*(n+1).
- F(n+1) = F(n) + k
- = c + k*n + k [by the IH]
- F(n+1) = c + k*(n+1)
- QED

Summary of the “Find the Pattern” Process

1. Determine the work required for the base and general cases of the given recursive algorithm.
2. Generate a few more equivalent recurrence for the work required in the general case.
3. Find the pattern within those expressions.
4. Create an equivalent closed-form (non-recurrence) expression in terms of the instance characteristic(s).
5. Prove that your closed-form expression is correct.
6. Determine the order of the algorithm.

Recursively Find the Max Value in a List



14	26	53	23	12	36	41	17	10	42	19	39

- Given an array or list of values, what is the largest value?
- Linear Search finds 53 in O(n) time. Can we do better?
- Let’s try the following algorithm:

1. Divide the list in half
2. Recursively find the largest # in each half
3. Compare these to find the largest overall

- The algorithm works, but how much work does it perform?

Recursively Find the Max Value in a List

- Step 1: Determine the base and general cases.
- The smallest useful list contains one value. Work required:
- T(1) = c
- For a list of n items, the work required is:
- T(n) = T($\frac{n}{2}$) + T($\frac{n}{2}$) + k = 2*T($\frac{n}{2}$) + k
- Well, if we're being accurate, when the list size is odd, $\frac{n}{2}$ isn't an integer! Which makes our recurrence relation a mess.
- However, we can grow the list by 1 item to make n even. This won't change our recurrence relation!

Recursively Find the Max Value in a List

- Step 2: Generate more equivalent recurrences.
- T(1) = c
- T(n) = 2*T($\frac{n}{2}$) + k
- T($\frac{n}{2}$) = 2*T($\frac{n}{4}$) + k
- T(n) = 2*(2*T($\frac{n}{4}$) + k) + k = T(n) = 4*T($\frac{n}{4}$) + 3k
- T(n) = 8*T($\frac{n}{8}$) + 7k
- What would be the next piece of the “pattern?”
- T(n) = 16*T($\frac{n}{16}$) + 15k

Recursively Find the Max Value in a List

- Step 2: Find the pattern. Our recurrences are:
- T(n) = 2*T($\frac{n}{2}$) + k
- T(n) = 4*T($\frac{n}{4}$) + 3k
- T(n) = 8*T($\frac{n}{8}$) + 7k
- T(n) = 16*T($\frac{n}{16}$) + 15k
- What would be the general case be?
- T(n) = 2^i *T($\frac{n}{2^i}$) + (2^i - 1)k, where i is a positive integer

Recursively Find the Max Value in a List

- Step 4: Create an equivalent closed-form expression
- T(n) = 2^i *T($\frac{n}{2^i}$) + (2^i - 1)k, where i is a positive integer
- What must the relationship be between n and i to reach T(1)?
- $\frac{n}{2^i}$ will reach 1 when n = 2^i
- T(n) = n*T($\frac{n}{n}$) + (n - 1)k
- = n * T(1) + (n – 1)k
- T(n) = c *n + k (n – 1)
- Step 5: Prove that the closed-form expression is correct… good practice for you!
- Step 6: Determine the order of the algorithm: O(n)

The Master Theorem

- Given a recurrence of the form T(n) = a * T($\frac{n}{b}$) + c * n^d, where:
- T(n) is an increasing function
- n= b^{k}, where k ∈ ℤ and k > 0
- a is a real number and ≥ 1
- b is an integer and > 1
- c is a real number and > 0
- d is a real number and ≥ 0 … Then:
- T(n) is O(n^{d}) if a $< b^d$
- T(n) is O($\mathrm{n}^{\mathrm{d}} * \log_2 n$) if a $= b^d$
- T(n) is O($\mathrm{n}^{\log_b a}$) if a $> b^d$

The Master Theorem

- Given a recurrence of the form T(n) = a * T($\frac{n}{b}$) + c * n^d,
- T(n) = O(n^{d}) if a $< b^d$
- T(n) = O($\mathrm{n}^{\mathrm{d}} * \log_2 n$) if a $= b^d$
- T(n) = O($\mathrm{n}^{\log_b a}$) if a $> b^d$
- Note that the Master Theorem does not *solve* the recurrence, it just tells you what the Big-O for the recurrence is
- Consider the recurrence T(n)=2T($\frac{n}{2}$) + n. Does M.T. apply here? If so, what are a, b, c, and d?
- Yes! a=2, b=2, c=1, d=1. So, what is the Big-O for this recurrence?
- T(n) = O($\mathrm{n}^{\mathrm{d}} * \log_2 n$) = O(n$* \log_2 n$)

The Master Theorem

- Given a recurrence of the form T(n) = a * T($\frac{n}{b}$) + c * n^d,
- T(n) = O(n^{d}) if a $< b^d$
- T(n) = O($\mathrm{n}^{\mathrm{d}} * \log_2 n$) if a $= b^d$
- T(n) = O($\mathrm{n}^{\log_b a}$) if a $> b^d$
- Recall our ‘max value’ recurrence? T(n)=2T($\frac{n}{2}$) + k
- Does this fit the form for M.T? If so, what are a, b, c, and d?
- Yes! a=2, b=2, c=k, d=0. What would our big-O be?
- O($\mathrm{n}^{\log_b a}$) -> O($\mathrm{n}^{\log_2 2}$)-> O(n)