Logistics

- Lecture: Here!
- Labs: Monday and Tuesday, led by Nithin Gowda
- Textbook: Introduction to Algorithms (aka, CLRS)
- Office Hours:
  - Daniel Lowd: Monday 1-2pm, Wednesday 2-3pm (262 Deschutes)
  - Nithin Gowda: Thursday 12-2pm, Friday 10-11:50am
- Web page
  - Syllabus, tentative schedule
  - Links to materials
  - Will be posted later today!
- Piazza
  - Announcements, updates, supplemental info
  - Ask and answer questions
  - Link on the web page! Please sign yourself up.
Evaluation

• Written assignments: 15% (~4 total)
• Programming assignments: 15% (~4 total)
• Gradescope
  • Submit written and programming homeworks
  • Get somewhat detailed rubric feedback on written assignments
  • Get instant (partial) feedback from autograder
  • Link on webpage

• Late policy
  • If you need a short extension, you must ask. (Email both Daniel and Nithin.)
  • Earlier is better.
  • I will typically say yes
  • If you abuse this policy, I may start saying no
More Evaluation

• Midterm exam: 20%
  • Friday, November 8

• Final exam: 40%
  • Tuesday, December 10

• You are allowed one sheet of handwritten notes (8.5x11, two-sided)

• Quizzes: 10%
  • Beginning of most lectures, starting next week (10/6)
  • Review material from last lecture
  • 3 questions, 2 minutes per question
  • Use iClicker to answer
  • Drop lowest two quiz scores
Programs = Algorithms + Data Structures
(by Niklaus Wirth)

• From the book
  • **Algorithm**: any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
  • **Data structure**: a way to store and organize data in order to facilitate access and modifications.
Themes

• Computational complexity, start to measure it
• Simple data structures (mostly review)
• Tree based structures
  • binary trees
  • binary heaps, binomial heaps
  • self adjusting trees: AVL, Red-Black
  • (2,4) trees, B-trees
• Sorting, order statistics, voting
First algorithm

find the maximum number in an array

Input: a sequence of numbers $a_1, a_2, ..., a_n$
Output: the maximum number in the input sequence

Algorithm:

$\text{max} = a_1$

for $i = 2$ to $n$:

if $a_i > \text{max}$:

$\text{max} = a_i$

return $\text{max}$

How long does this take?
Maybe: $n$ variable assignments, $n-1$ comparisons, $n-2$ increments, one return?
How do we talk about algorithm speed?

• Use functions of the size of the input $n$, e.g., $T(n)$
  (typically $n$ is the number of input numbers/items in this class)
• Apply asymptotic notation for these functions ($O$, $\Omega$, $\Theta$, $o$, $\omega$)
• See description and definitions in text (section 3.1, pp 43-52)
<table>
<thead>
<tr>
<th>algorithm speed</th>
<th>input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>10</td>
</tr>
<tr>
<td>$10^{-5}$ seconds</td>
<td>$2 \cdot 10^{-5}$ seconds</td>
</tr>
<tr>
<td>$n^2$</td>
<td>10^{-4} seconds</td>
</tr>
<tr>
<td>$n^3$</td>
<td>10^{-3} seconds</td>
</tr>
<tr>
<td>$n^{10}$</td>
<td>2.7 hours</td>
</tr>
<tr>
<td>$2^n$</td>
<td>10^{-3} seconds</td>
</tr>
<tr>
<td>$3^n$</td>
<td>.06 second</td>
</tr>
<tr>
<td>$n!$</td>
<td>3.6 seconds</td>
</tr>
<tr>
<td>$2^{2^n}$</td>
<td>&gt;10^{292} centuries</td>
</tr>
</tbody>
</table>
big-Oh formally

\[ f(n) = O(g(n)) \text{ if and only if (iff)} \]
\[ \text{there are positive constants } c, n_0 \text{ such that} \]
\[ 0 \leq f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0 \]

- \( c \) is the dropped constant
- \( n_0 \) is the crossover point so that ...
- ... if \( n \) is big enough \( f \) is bounded above by \( c \cdot g \)
- the growth rate of \( g \) bounds the growth rate of \( f \) from above

**example:** let \( f(n) = 3n^3 + 5n^2 + n + 17 \)

**some true statements:**
- \( f(n) = O(n^3) \)
- \( f(n) = O(n^4) \)
- \( f(n) = O(17 \ n^3) \)
- \( f(n) = 3n^3 + O(n^2) \)
Big Omega and Theta

\[ f(n) = \Omega(g(n)) \quad \text{iff} \quad \text{there are positive constants } c, n_0 \text{ such that} \]
\[ 0 \leq c \cdot g(n) \leq f(n) \quad \text{for all } n \geq n_0 \]

thus, the growth rate of \( g \) is less than or equal to the growth rate of \( f \) (ignoring the constant)

\[ f(n) = \Theta(g(n)) \quad \text{iff} \quad f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]

- here \( f \) and \( g \) have the \underline{same} growth rate
- sort of like saying \( A \leq B \) and \( A \geq B \) implies that \( A = B \)

now we can say \( f(n) = 3n^3 + 5n^2 + n + 17 \)
- \( f(n) = \Omega(n^3) \)
- \( f(n) = \Omega(n^2) \)
- \( f(n) = \Theta(n^3) \)
- \( f(n) = 3 \cdot n^3 + \Theta(n^2) \)
\[ f(n) = \Theta(g(n)) \]

\[ f(n) = O(g(n)) \]

\[ f(n) = \Omega(g(n)) \]
little-oh and little-omega

\[ f(n) = o(g(n)) \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \]

or

\[ \forall c > 0 \exists n_0 \forall n \geq n_0 \ 0 \leq f(n) \leq c \cdot g(n) \]

in other words, the growth rate of f is strictly less than that of g

\[ f(n) = \omega(g(n)) \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \]

or

\[ \forall c > 0 \exists n_0 \forall n \geq n_0 \ f(n) \geq c \cdot g(n) \geq 0 \]

the growth rate of f is strictly greater than that of g

examples (f(n) = 3n^3 + 5n^2 + n + 17):
• \( f(n) = o(n^4) \)
• \( f(n) = \omega(n^2) \)
• \( f(n) = 3 \cdot n^3 + o(n^3) \)
• \( \frac{1}{n} = o(1) \)
some properties

- Transitivity:
  \[ f(n) = \alpha(g(n)) \text{ and } g(n) = \alpha(h(n)) \text{ imply } f(n) = \alpha(h(n)) \ (\alpha \in \{O, \Omega, \Theta, o, \omega}\} \]

- Reflexivity:
  \[ f(n) = \alpha(f(n)) \ (\alpha \in \{O, \Omega, \Theta}\} \]

- Symmetry:
  \[ f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n)) \]

- Transpose Symmetry:
  \[ f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \]
  \[ f(n) = o(g(n)) \iff g(n) = \omega(f(n)) \]
common functions

- \( n^k \), where \( k \) is a constant (polynomial)
- \( 2^n \), \( 3^n \), \( c^n \) (exponential)
- \( \log_2 n \), \( \log_c n \), \( \ln n \) (logarithmic – usually \( \log n \) implies base 2)
  - fact: \( \log_2 n = O(\log_c n) \) (why?)
- \( O(n \log n) \) (also poly, but very common)
- \( n! \) (factorial)
- \( 2^{(\log n)^2} \) (super-poly, sub-exponential) (ok, not so common)
other functions

• factorials: \( n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1 \)

• Stirling’s Approximation: \( n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot (1 + \Theta \left(\frac{1}{n}\right)) \)

• importantly \( \log n! = \Theta(n \cdot \log n) \)

• binomial coefficients

• Fibonacci sequence: \( F_0 = 0, F_1 = 1, F_{k+2} = F_{k+1} + F_k \) (Fibonacci used for AVL trees)
more examples

10 \log n + \log \log n \quad \text{is} \quad \Theta(\log n)? \quad O(n)? \quad O(n^{0.00000001})? \quad \Omega(\log n)? \quad O((\log n)^{0.5})? \quad \Omega((\log n)^{0.5})?

2^{32000} \quad \text{is} \quad O(1)? \quad \Omega(1)? \quad 2^{32000} \quad n \text{ is } O(n)?

2/n \quad \text{is} \quad O(1/n)? \quad O(1/\sqrt{n})? \quad O(1/n^{1.7})? \quad O(1)?

f(n) = \begin{cases} 
0.1 \ n \text{ if } n \text{ is odd} \\
3 \ n^2 \text{ if } n \text{ is even}
\end{cases}
\quad \text{is} \quad O(n)? \quad O(n^{1.5})? \quad O(n^2)? \quad \Omega(n)? \quad \Omega(n^{1.5})? \quad \Omega(n^2)?
reading for previous material

• chapter 3
• appendix A.1
Big-O Miscellany #1

(You can ignore this if it’s confusing...)

$f(n) = O(g(n))$ is an abuse of notation.

To be more precise, we can define big-O as a set:

$$O(g(n)) = \{ f : \exists c > 0, n_0 > 0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \forall n \geq n_0 \}$$

And then write:

$$f(n) \in O(g(n))$$
Big-O Miscellany #2

(You can ignore this if it’s confusing...)

Just like little-o and little-omega, big-O can be defined as a limit!

\[ f(n) = O(g(n)) \iff \exists c > 0 \text{ such that: } \lim_{n \to \infty} \frac{f(n)}{g(n)} < c \]

And you can do something similar for big Omega.
(This you should know...)  

- Exponentials with a smaller base grow slower than exponentials with a larger base: \[ 2^n = o(3^n) \]
- Polynomials grow slower than exponentials: \[ n^{100} = o(1.001^n) \]
- Polynomials of smaller degree (exponent) grow slower than polynomials of higher degree: \[ n^2 = o(n^{2.1}) \]
- Logs grow slower than polynomials: \[ (\log n)^{100} = o(n^{0.001}) \]
loop invariants

• “simple” method to prove correctness of a loop structure
• follows induction
• three phases: initialization (base case),
  invariance maintenance (induction), and
  termination

• look at pp 18-20 of text for more discussion
• while there, look at pp 20-22 for description of pseudo-code
general structure of argument

**code:**
<init>
while $\gamma$ 
do $\mathcal{L}$

**invariant:** $\alpha$
a true/false statement about the variables of the code

**Initialization:** show that $\alpha$ is true after the <init> phase of the code has been executed

**Maintenance:** show that if $\alpha \land \gamma$ is true, then $\alpha$ will be true after one execution of the loop body $\mathcal{L}$

**Termination:** the loop finishes when $\gamma$ is false, so argue that $\neg \gamma \land \alpha$ is the desired outcome
example

input: integer n>0
output: n(n+1)/2

--initialization
int s=0
int k=0

--loop
while k < n+1 do
  s = s+k
  k = k+1
--end
return s

\[ \gamma: k < n+1 \]

\[ \alpha: \]
• \( 0 \leq k \leq n + 1 \)
• \( s = k(k-1)/2 \)
example

input: integer n>0
output: integer k, array b of k bits

--initialization
int k=0
int t=n
array b=[] of bit

--loop
while t>0 do
  b[k] = (t mod 2)
  k = k+1
  t = t div 2

--end
return k, b

γ: t>0

α:
• t ≥ 0
• Let $m = \sum_{i=0}^{k-1} b[i] \cdot 2^i$ be the number represented by b in base 2. Then $n = 2^k \cdot t + m$

notice:
• initialization is easy
• termination also easy
• see handout (posted on class site) for full discussion
example

Compute the $n$-th Fibonacci number