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 $n$ (typically the number of input numbers/items in this class), i.e., $T(n)$
- apply asymptotic notation for these functions
- it ignores constants and only focuses on the highest-order term
  - why? machine independence, constants not important asymptotically
  - asymptotically = “in the long run or in the limit”
- see description and definitions in text (section 3.1, pp 43-52)
- $O$, $\Omega$, $\Theta$, $o$, $\omega$
Time spent at 1,000,000 operations per second:

<table>
<thead>
<tr>
<th>n</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>...</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$10^{-5}$ seconds</td>
<td>$2 \cdot 10^{-5}$ seconds</td>
<td>$3 \cdot 10^{-5}$ seconds</td>
<td>$4 \cdot 10^{-5}$ seconds</td>
<td>$5 \cdot 10^{-5}$ seconds</td>
<td>$6 \cdot 10^{-5}$ seconds</td>
<td>$10^{-4}$ seconds</td>
<td></td>
</tr>
<tr>
<td>$n^2$</td>
<td>$10^{-4}$ seconds</td>
<td>$4 \cdot 10^{-4}$ seconds</td>
<td>$9 \cdot 10^{-4}$ seconds</td>
<td>$1.6 \cdot 10^{-3}$ seconds</td>
<td>$2.5 \cdot 10^{-3}$ seconds</td>
<td>$3.6 \cdot 10^{-3}$ seconds</td>
<td>.01 second</td>
<td></td>
</tr>
<tr>
<td>$n^3$</td>
<td>$10^{-3}$ seconds</td>
<td>$8 \cdot 10^{-3}$ seconds</td>
<td>$2.7 \cdot 10^{-3}$ seconds</td>
<td>$6.4 \cdot 10^{-2}$ seconds</td>
<td>.125 second</td>
<td>.216 second</td>
<td>1 second</td>
<td></td>
</tr>
<tr>
<td>$n^{10}$</td>
<td>2.7 hours</td>
<td>118 days</td>
<td>18 years</td>
<td>333 years</td>
<td>3,103 years</td>
<td>19,213 years</td>
<td>31,775 centuries</td>
<td></td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^{-3}$ seconds</td>
<td>1 second</td>
<td>17 minutes</td>
<td>12 days</td>
<td>35.7 years</td>
<td>36,634 years</td>
<td>4 · $10^{14}$ centuries</td>
<td></td>
</tr>
<tr>
<td>$3^n$</td>
<td>.06 second</td>
<td>58 minutes</td>
<td>6.5 years</td>
<td>3863 centuries</td>
<td>2 · $10^8$ centuries</td>
<td>1.3 · $10^{13}$ centuries</td>
<td>1.6 · $10^{32}$ centuries</td>
<td></td>
</tr>
<tr>
<td>$n!$</td>
<td>3.6 seconds</td>
<td>773 centuries</td>
<td>$8 \cdot 10^{16}$ centuries</td>
<td>$2.6 \cdot 10^{32}$ centuries</td>
<td>$9.7 \cdot 10^{48}$ centuries</td>
<td>$2.6 \cdot 10^{66}$ centuries</td>
<td>$3 \cdot 10^{142}$ centuries</td>
<td></td>
</tr>
<tr>
<td>$2^{2^n}$</td>
<td>$&gt;10^{292}$ centuries</td>
<td>$&gt;10^{315637}$ centuries</td>
<td>ouch!</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

algorithm speed

input size
**big-Oh formally**

\[ f(n) = O(g(n)) \text{ if and only if (iff)} \]
\[ \exists c > 0 \\exists N \ \forall n \geq N \ 0 \leq f(n) \leq c \cdot g(n) \]

- \( c \) is the dropped constant
- \( N \) 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

\[ f(n) = \Omega(g(n)) \iff \exists c > 0 \exists N \forall n \geq N \ f(n) \geq c \cdot g(n) \geq 0 \]

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

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

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

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

$n_0$

$\begin{align*}
c_2g(n) \\
f(n) \\
c_1g(n)
\end{align*}$

$n_0$

$\begin{align*}
c_2g(n) \\
&\quad f(n) \\
c_1g(n)
\end{align*}$

$n_0$

$\begin{align*}
c_2g(n) \\
&\quad f(n) \\
c_1g(n)
\end{align*}$

$n_0$

$\begin{align*}
f(n) = O(g(n))
\end{align*}$

$n_0$

$\begin{align*}
f(n) = O(g(n))
\end{align*}$

$n_0$

$\begin{align*}
f(n) = O(g(n))
\end{align*}$

$n_0$

$\begin{align*}
f(n) = \Omega(g(n))
\end{align*}$

$n_0$

$\begin{align*}
f(n) = \Omega(g(n))
\end{align*}$

$n_0$

$\begin{align*}
f(n) = \Omega(g(n))
\end{align*}$
little-o and little-omega

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

or

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

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

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

or

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

the growth rate of \( f \) is strictly greater than that of \( g \)

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

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

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

- 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: \( n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right) \)

• 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 O(\log n) \text{? } O(n) \text{? } O(n^{0.0000001}) \text{? } \Omega(\log n) \text{? } O((\log n)^{0.5}) \text{? } \Omega((\log n)^{0.5})

2^{3^{2000}} \quad \text{is } O(1) \text{? } \Omega(1) \ ? \ 2^{3^{2000}} n \quad \text{is } O(n) \text{?}

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

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 } O(n) \text{? } O(n^{1.5}) \text{? } O(n^2) \text{? } \Omega(n) \text{? } \Omega(n^{1.5}) \text{? } \Omega(n^2)
reading for previous material

• chapter 3
• appendix A.1
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 $\alpha \wedge \gamma$ 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 $\neg \gamma \wedge \alpha$ 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

<table>
<thead>
<tr>
<th>input: integer ( n &gt; 0 )</th>
<th>output: integer ( k ), array ( b ) of ( k ) bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>--initialization</strong></td>
<td>int ( k = 0 )</td>
</tr>
<tr>
<td></td>
<td>int ( t = n )</td>
</tr>
<tr>
<td></td>
<td>array ( b = [] ) of bit</td>
</tr>
<tr>
<td><strong>--loop</strong></td>
<td>while ( t &gt; 0 ) do</td>
</tr>
<tr>
<td></td>
<td>( b[k] = (t \mod 2) )</td>
</tr>
<tr>
<td></td>
<td>( k = k + 1 )</td>
</tr>
<tr>
<td></td>
<td>( t = t \div 2 )</td>
</tr>
<tr>
<td><strong>--end</strong></td>
<td>return ( k, b )</td>
</tr>
</tbody>
</table>

\[
\gamma: t > 0
\]

\[
\alpha:
\begin{itemize}
  \item \( t \geq 0 \)
  \item Let \( m = \sum_{i=0}^{k-1} b[i] \cdot 2^i \) be the number represented by \( b \) in base 2. Then \( n = 2^k \cdot t + m \)
\end{itemize}

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

Compute the $n$-th Fibonacci number