Today's topics

• Algorithms
  – Algorithm review
  – Orders of Growth

• Reading: Sections 2.1-3.3

Upcoming

• Integers

§2.1: Algorithms

- The foundation of computer programming.
- Most generally, an algorithm just means a definite procedure for performing some sort of task.
- A computer program is simply a description of an algorithm, in a language precise enough for a computer to understand, requiring only operations that the computer already knows how to do.
- Implementations of an algorithm. We say that a program implements (or "is an implementation of") an algorithm.

Our Pseudocode Language: §A2

Some important general features of algorithms:

- Input. Information or data that comes in.
- Output. Information or data that goes out.
- Definiteness. Algorithm is precisely defined.
- Correctness. Outputs correctly relate to inputs.
- Finiteness. Won’t take forever to describe or run.
- Effectiveness. Individual steps are all do-able.
- Generality. Works for many possible inputs.
- Efficiency. Takes little time & memory to run.

Algorithm Characteristics

- Efficiency
- Generality
- Effectiveness
- Finiteness
- Correctness
- Definiteness
- Optimal. Algorithm is precisely defined.
- Input. Information or data that comes in.
- Output. Information or data that goes out.
### Procedure

Declares that the following text defines a procedure named `procname` that takes inputs (arguments) named `arg` which are data objects of the type `type`.

**Example:**

```plaintext
procedure maximum (LL: list of integers)
  [statements defining maximum...]
```

### Assignment Statement

An assignment statement evaluates the expression, then reassigns the variable to the value that results.

**Example assignment statement:**

```plaintext
v := 3x + 7         (If x is 2, changes v to 13.)
```

**Informal statement**

Sometimes we may write a statement as an informal English imperative, if the meaning is still clear and precise: e.g., “swap `x` and `y`”

Keep in mind that real programming languages never allow this.

When we ask for an algorithm to do so-and-so, writing “Do so-and-so” isn’t enough! We need to break down algorithms into detailed steps.

Break down algorithms into detailed steps.

- Write detailed steps.
- Break down an informal imperative into pseudocode.
- When we ask for an algorithm to do so-and-so.
- Languages never allow this.

### Groups a sequence of statements together:

- Allows the sequence to be used just like a single statement.
- Might be used after a procedure declaration.
- In an if statement after then or else.
- In the body of a for or while loop.
- In the body of a for or while.

### Beginning Statements

Curly braces `{}` are used instead in many languages.
• Not executed (does nothing).
  Also called a remark in some real programming languages, e.g. BASIC.
  Example, might appear in a max program:
  \{ Note that \( v \) is the largest integer seen so far \}

• Evaluate the propositional (Boolean) expression \( \text{condition} \).
  If the resulting value is True, then execute \( \text{statement} \).
  Otherwise, just skip on ahead to the next statement after the \( \text{if} \) statement.

• Also equivalent to infinite nested if's, like so:
  \begin{align*}
  & \text{if} \quad \text{condition} \\
  & \text{begin} \\
  & \quad \text{statement} \\
  & \text{if} \quad \text{condition} \\
  & \text{begin} \\
  & \quad \text{statement} \\
  & \cdots \text{(continue infinite nested if's)} \\
  & \text{end} \\
  & \text{end}
  \end{align*}
for var := initial to final stmt

• Initial is an integer expression.
• Final is another integer expression.

Semantics: Repeatedly execute stmt first with variable var := initial
then with var := var + 1, then with var := var + 2,
until var > final evaluates to true.

Question: What happens if stmt changes the value of var?

Various real programming languages refer to procedures as functions (since the procedure call notation works similarly to function application f(x)), or as subroutines, subprograms, or methods.

Max procedure in pseudocode

procedure max(a_1, a_2, ..., a_n: integers): integers
  
  v := a_1          \{largest element so far\}

  for i := 2 to n    \{go thru rest of elements\}
    if a_i > v then
      v := a_i       \{found bigger?\}

  return v

\{at this point v's value is the same as the largest integer in the list\}
4.17 Inventing an Algorithm

• Requires a lot of creativity and intuition

– Like writing proofs.

• Unfortunately, we can’t give you an algorithm for inventing algorithms.

– Just look at lots of examples.

– And practice (preferably, on a computer).

– And look at more examples.

– And practice some more. Etc., etc.

0.47

4.18 Algorithm-Inventing Example

• Suppose we ask you to write an algorithm to compute the predicate:

\[ \text{IsPrime}(n) \]

• Computes whether a given natural number is a prime number.

First, start with a correct predicate-logic definition of the desired function:

\[ n \colon \text{IsPrime}(n) \equiv \neg \exists d \mid 1 < d < n \text{ such that } d | n \]

This works because of this theorem:

\[ \text{If } n \text{ has any (integer) divisors, it must have one less than } \frac{n}{2} \]

Proof:

Suppose \( n \)’s smallest divisor >1 is \( a \), and let \( b = n/a \). Then \( n = ab \), and since \( a < n/2 \), \( b > n/2 \).

This universal can then be translated directly into a corresponding for loop:

for \( d := 2 \text{ to } \sqrt{n} \)

if \( d | n \) then return \( F \)

return \( T \)

This works because of this theorem:

\[ \text{If } n \text{ has any (integer) divisors, it must have one less than } \frac{n}{2} \]

Further optimizations are possible:

Optimizing IsPrime

• The IsPrime algorithm can be further optimized:

for \( d := 2 \text{ to } \sqrt{n} \)

if \( d | n \) then return \( F \)

return \( T \)

This works because of this theorem:

\[ \text{If } n \text{ has any (integer) divisors, it must have one less than } \frac{n}{2} \]

Proof:

Suppose \( n \)’s smallest divisor >1 is \( a \), and let \( b = n/a \). Then \( n = ab \), but if \( a > n/2 \), then \( b < n/2 \). Since \( a < n/2 \), \( b > n/2 \).

Further optimizations are possible:

– E.g., only try divisors that are primes less than \( n^{1/2} \).

– And look at lots of examples.

– And practice some more. Etc., etc.

– Like writing proofs.

– Requires a lot of creativity and intuition.

Algorithm-Inventing Example

Inventing an Algorithm
Another example task

- Problem of searching an ordered list.
  - Given a list $L$ of $n$ elements that are sorted into a definite order (e.g., numeric, alphabetical),
  - And given a particular element $x$,
  - Determine whether $x$ appears in the list,
  - and if so, return its index (position) in the list.
- Problem occurs often in many contexts.
- Let’s find an efficient algorithm!

Search alg. #1: Linear Search

procedure linear search
($x$: integer, $a_1$, $a_2$, ..., $a_n$: distinct integers)
$i := 1$  {start at beginning of list}
while ($i \leq n \land x \neq a_i$) {not done, not found}
  $i := i + 1$  {go to the next position}
if $i \leq n$ then location := $i$  {it was found}
else location := 0  {it wasn’t found}
return location  {index or 0 if not found}

Search alg. #2: Binary Search

- Basic idea: On each step, look at the middle element of the remaining list to eliminate half of it, and quickly zero in on the desired element.

procedure binary search
($x$:integer, $a_1$, $a_2$, ..., $a_n$: distinct integers)
$i := 1$  {left endpoint of search interval}
$j := n$  {right endpoint of search interval}
while $i < j$ begin  {while interval has >1 item}
  $m := \lfloor (i+j)/2 \rfloor$  {midpoint}
  if $x > a_m$ then $i := m + 1$ else $j := m$
end
if $x = a_i$ then location := $i$ else location := 0
return location  {index or 0 if not found}
4.25 Practice exercises

2.1.3: Devise an algorithm that finds the sum of all the integers in a list. [2 min]

```
procedure sum(a_1, a_2, …, a_n: integers): integers
ss := 0 {sum of elements so far}
for i := 1 to n {go through all elements}
s := s + a_i {add current item}
{at this point s is the sum of all items}
return s {end this code}
```

Orders of Growth - Motivation

Which program do you choose, knowing you'll want to support millions of users? 

- Let's compare the performance of database programs A and B:
  - Program A takes \( f_A(n) = 30n + 8 \) microseconds to process all records.
  - Program B takes \( f_B(n) = n^2 + 1 \) microseconds to process all records.

For large numbers of user records, \( f_B(n) \) will always take more time.

Concept of order of growth

We say \( f_A(n) = 30n + 8 \) is \( \text{order } n \), or \( O(n) \).

- For large numbers of user records, the exactly order \( f_B(n) = n^2 + 1 \) will always take more time.

Visualization of Orders of Growth

- On a graph, as you go to the right, the faster-growing function always becomes the larger one.

Orders of Growth - Exercise

2.1.3: Develop an algorithm that finds the sum of all the integers in a list. [2 min]
4.29
Definition: O(g)

Let be any function .

Define 

\[ f(n) \in O(g(n)) \text{ if and only if } \exists c > 0, k \in \mathbb{R} \text{ such that } f(n) \leq cg(n) \text{ for all } n \geq k. \]

Beyond some point beyond some point \( n = k \), function \( f \) is at most proportional to \( g \)."

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

\[ \rightarrow \]

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

4.30
Points about the definition

Note that \( f \in O(g) \) so long as any values of \( c \) and \( k \) exist that satisfy the definition.

However, you should prove that the values you choose do work.

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

\[ \rightarrow \]

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

\[ \text{Big-O example, graphically} \]

4.31
\[ f(n) = O(g(n)) \]

\[ \rightarrow \]

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

\[ \text{Big-O Proof Examples} \]

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

\[ \rightarrow \]

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

\[ \text{Definition: } O(9) \text{ at most order 9} \]
Useful Facts about Big O

- Big O, as a relation, is transitive: $f \in \Theta(g) \iff \Theta(f) \subseteq \Theta(g)$

- O with constant multiples, roots, and logs...

\begin{align*}
\{x \in \mathbb{R} \mid f(x) = O(g(x))\} &= \Theta(f(x) + g(x))
\end{align*}

Sums of functions:

- $f \in \Theta(g)$ and $h \in \Theta(h)$, then $f + h \in \Theta(f + h)$

More Big-O Facts

Orders of Growth (§1.8) - So Far

- Formally, you can think of any such function that is $O(f)$.

- Often, one deals only with positive functions and can ignore absolute value symbols.

- The latter form is an instance of a more general convention...

- Sums of functions...
Order of Growth Equations

• Suppose \( f(x) \) and \( g(x) \) are order-of-growth expressions corresponding to the sets of functions \( S \) and \( T \), respectively.

Then the equation \( f(x) = O(g(x)) \) really means

\[ f(x) = \Theta(g(x)) \quad \text{or simply} \quad f(x) \in \Theta(g(x)). \]

Example:

\[ (x)O = x + x \cdot O, \quad (x)O = O + f, \quad (x)f = O + f. \]

Rules for \( \Theta \)

Mostly like rules for \( O() \), except:

\[ f, g \in O() \text{ with } b > 0, \quad \text{and } |f(x)| \leq b|g(x)| \quad \text{for all } x \geq x_0. \]

Also, if \( f(x) = x \cdot O \) then we say, "\( f \) is (exactly) order \( O(x) \)" and write \( f(x) \in O(x) \).

Another equivalent definition:

\[ f(x) = \Theta(g(x)) \quad \text{if and only if} \quad f(x) \in O(g(x)) \quad \text{and} \quad g(x) \in O(f(x)). \]

Definition: \( \Theta(g) \), exactly order \( g \)

Everywhere beyond some point \( x \), \( f(x) \) lies between two multiples of \( g(x) \).

\[ \{ |f(x)| \leq c_1 |g(x)| \quad \text{for all } x \geq x_0, \quad \text{with } c_1 > 0 \}. \]

Example:

\[ (x)O = x + x \cdot O, \quad (x)O = O + f, \quad (x)f = O + f. \]

Useful Facts about Big \( O \)

- \( (x)f = \Theta(g) \) and \( g \in O(f) \), then we say, "\( f \) and \( g \) are order-of-growth.

Order of Growth Equations
A function that is $O(x)$, but neither $o(x)$ nor 
\[ (x) \in \Theta. \]

Why $o(f) \subset (\Theta - (x))$?

\[
\begin{align*}
\{ (x) \} &> \{ (x) \in \Theta \} : \forall x A \forall E 0 < x < A | f \} = (\Theta) \in \Theta, \quad \text{then } S \subset \Theta \in \Theta \\
\{ (x) \in (\Theta - (x)) \} &> \{ (x) \} : \forall x A \forall E 0 < x < A | f \} = (x) \in \Theta, \quad \text{then } S \subset \Theta \in \Theta \\
\{ (f) \in (\Theta ((x))) \} &> \{ (f) \} : \forall x A \forall E 0 < x < A | f \} = (\Theta) \in \Theta, \quad \text{then } S \subset \Theta \in \Theta
\end{align*}
\]

Other Order-of-Growth Relations

Subset relations between order-of-growth sets.

Relations Between the Relations

\[
\begin{align*}
(f) \in O &\Rightarrow (f) \in \Theta \\
(f) \in O &\Rightarrow (f) \in \Theta \\
(f) \in O &\Rightarrow (f) \in \Theta \\
(f) \in O &\Rightarrow (f) \in \Theta
\end{align*}
\]

Why $o(f) \subset (\Theta - (x))$?

\[
\begin{align*}
\{ (x) \} &> \{ (x) \in \Theta \} : \forall x A \forall E 0 < x < A | f \} = (\Theta) \in \Theta, \quad \text{then } S \subset \Theta \in \Theta \\
\{ (x) \in (\Theta - (x)) \} &> \{ (x) \} : \forall x A \forall E 0 < x < A | f \} = (x) \in \Theta, \quad \text{then } S \subset \Theta \in \Theta \\
\{ (f) \in (\Theta ((x))) \} &> \{ (f) \} : \forall x A \forall E 0 < x < A | f \} = (\Theta) \in \Theta, \quad \text{then } S \subset \Theta \in \Theta
\end{align*}
\]

Determine whether:

- \[ (\frac{1}{u}) \Theta = \]
- \[ (\frac{1}{u}) \Theta = \]
- \[ (1 - u) = \left( \sum_{i} \frac{1}{u} \right) \]
- \[ (\frac{1}{u}) \Theta = \left( \sum_{i} \right) \]

Quick solution:

\[ (\frac{1}{u}) \Theta = \]

Example
Strict Ordering of Functions

• Temporarily let’s write $s$ to mean $f(o(g))$, and $f$ to mean $(g(o(f)))$.

Note that:

Let $k > 1$. Then the following are true:

1. $p \log n \sim \log n$
2. $p \log n \sim \log n$
3. $p \log n \sim \log n$
4. $p \log n \sim \log n$

Let $k > 1$. The following are false:

1. $\frac{\log n}{x} \sim \log n$
2. $\lim_{x \to \infty} \frac{\log n}{x} = 0$
3. $\lim_{x \to \infty} \frac{\log n}{x} = 0$

Note that:

• Temporarily let’s write $f \sim g$ to mean $f \in \Theta(g)$.

Definitions of order-of-growth sets:

- $O(f)$
- $\Omega(f)$
- $\Theta(f)$
- $o(f)$
- $\omega(f)$

Review: Orders of Growth (§1.8)