Lecture 3:
Measuring Performance and Analysis Tools (plus Pseudocode Reference)

EECS 281: Data Structures & Algorithms

Complexity Recap

Notation and terminology

n = input size

f(n) = max number of steps when input has length *n*

O(f(n)) asymptotic upper bound

```
1  void f(int *out, const int *in, int size) {
2    int product = 1;
3    for (int i = 0; i < size; ++i)
4      product *= in[i];
5    for(int i = 0; i < size; ++i)
6      out[i] = product / in[i];
7  } // f()
```

$f(n) = 1 + 1 + 1 + 3n + 1 + 1 + 3n = 6n + 5 = O(n)$

Ways to measure complexity

- Analytically
  - Analysis of the code itself
  - Recognizing common algorithms/patterns
  - Based on a recurrence relation
- Empirically
  - Measure runtime programmatically
  - Measure runtime using external tools
  - Test on inputs of a variety of sizes

Measuring Runtime Programmatically

- "Programmatically" – i.e. in the code itself
- Varies greatly depending on the language
  - Many different ways to do it even just in C/C++!

Measuring Time In C++11+

```
1   #include <chrono>
2
3   class Timer {
4     std::chrono::time_point<std::chrono::system_clock> cur;
5     std::chrono::duration<double> elap;
6   public:
7     Timer() : cur(), elap(std::chrono::duration<double>::zero()) { }
8
9     void start() {
10      cur = std::chrono::system_clock::now();
11    }
12
13    void stop() {
14      elap += std::chrono::system_clock::now() - cur;
15    }
16
17    void reset() {
18      elap = std::chrono::duration<double>::zero();
19    }
20
21    double seconds() {
22      return elap.count();
23    }
24  };
```

Note: Checking time too often will slow down your program!

Example

```
int main() {

  Timer t;
  t.start();
  doStuff1();
  t.stop();
  cout << "1: " << t.seconds()
       << "s" << endl;

  t.reset();
  t.start();
  doStuff2();
  t.stop();
  cout << "2: " << t.seconds()
       << "s" << endl;
  return 0;
}
```

Credit: Amir Kamil

Let's try it!

From a web browser:

`goo.gl/gONvC0`

From a terminal:

```
wget goo.gl/gONvC0 -O search.cpp
```

After Downloading

- If you haven't already, add this to your **`.bash_profile`**:

```
module load gcc/6.2.0
```

- Compile:

```
g++ -std=c++1z -O3 search.cpp -o search
```

- Run a binary search, 1M items:

```
./search b 1000000
```

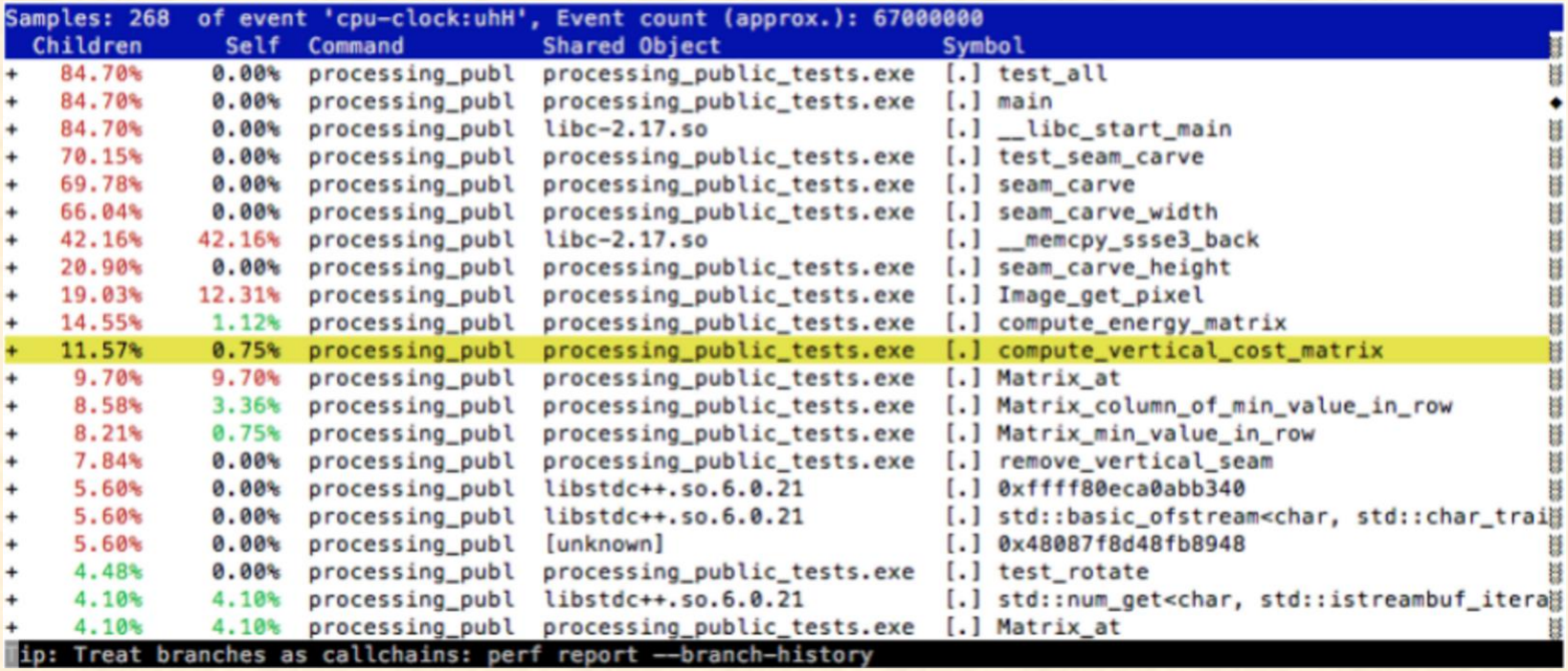
- Run a linear search, 1M items:

```
./search l 1000000
```

- Try with larger numbers!

Using a Profiling Tool

- This won't tell you the complexity of an algorithm, but it tells you where you program spends its time.
- Many different tools exist – you'll learn to use `perf` in lab.



```
Samples: 268  of event 'cpu-clock:uhH', Event count (approx.): 67000000
  Children      Self  Command          Shared Object                  Symbol
+   84.70%     0.00%  processing_publ  processing_public_tests.exe    [.] test_all
+   84.70%     0.00%  processing_publ  processing_public_tests.exe    [.] main
+   84.70%     0.00%  processing_publ  libc-2.17.so                   [.] __libc_start_main
+   70.15%     0.00%  processing_publ  processing_public_tests.exe    [.] test_seam_carve
+   69.78%     0.00%  processing_publ  processing_public_tests.exe    [.] seam_carve
+   66.04%     0.00%  processing_publ  processing_public_tests.exe    [.] seam_carve_width
+   42.16%    42.16%  processing_publ  libc-2.17.so                   [.] __memcpy_ssse3_back
+   20.90%     0.00%  processing_publ  processing_public_tests.exe    [.] seam_carve_height
+   19.03%    12.31%  processing_publ  processing_public_tests.exe    [.] Image_get_pixel
+   14.55%     1.12%  processing_publ  processing_public_tests.exe    [.] compute_energy_matrix
+   11.57%     0.75%  processing_publ  processing_public_tests.exe    [.] compute_vertical_cost_matrix
+    9.70%     9.70%  processing_publ  processing_public_tests.exe    [.] Matrix_at
+    8.58%     3.36%  processing_publ  processing_public_tests.exe    [.] Matrix_column_of_min_value_in_row
+    8.21%     0.75%  processing_publ  processing_public_tests.exe    [.] Matrix_min_value_in_row
+    7.84%     0.00%  processing_publ  processing_public_tests.exe    [.] remove_vertical_seam
+    5.60%     0.00%  processing_publ  libstdc++.so.6.0.21            [.] 0xffff80eca0abb340
+    5.60%     0.00%  processing_publ  libstdc++.so.6.0.21            [.] std::basic_ofstream<char, std::char_trai
+    5.60%     0.00%  processing_publ  [unknown]                      [.] 0x48087f8d48fb8948
+    4.48%     0.00%  processing_publ  processing_public_tests.exe    [.] test_rotate
+    4.10%     4.10%  processing_publ  libstdc++.so.6.0.21            [.] std::num_get<char, std::istreambuf_itera
+    4.10%     4.10%  processing_publ  processing_public_tests.exe    [.] Matrix_at
ip: Treat branches as callchains: perf report --branch-history
```

A snapshot of a `perf` report generated for EECS 280 Project 2.
Image credit: Alexandra Brown

Measuring Runtime on Linux

If you are launching a program using command

```
% progName -options args
```

Then

```
% /usr/bin/time progName -options args
```

will produce a runtime report

```
0.84user 0.00system 0:00.85elapsed 99%CPU
```

If you're timing a program in the current folder, use `./`

```
% /usr/bin/time ./progName -options args
```

Often, you can just type `time` rather than `/usr/bin/time`.

Measuring Runtime on Linux

- Example: this command just wastes time by copying zeros to the "bit bucket”

```
% time dd if=/dev/zero of=/dev/null
```

kill it with control-C

```
3151764+0 records in
3151764+0 records out
1613703168 bytes (1.6 GB) copied, 0.925958 s, 1.7 GB/s
Command terminated by signal 2
0.26user 0.65system 0:00.92elapsed 99%CPU
(0avgtext+0avgdata 3712maxresident)k
0inputs+0outputs (0major+285minor)pagefaults 0swaps
```

Measuring Runtime on Linux

```
0.26user 0.65system 0:00.92elapsed 99%CPU
```

- `user` time is spent by your program
- `system` time is spent by the OS on behalf of your program
- `elapsed` is wall clock time - time from start to finish of the call, including any time slices used by other processes
- `%CPU` Percentage of the CPU that this job got. This is just (`user` + `system`) / `elapsed`

- `man time` for more information

Using `valgrind`

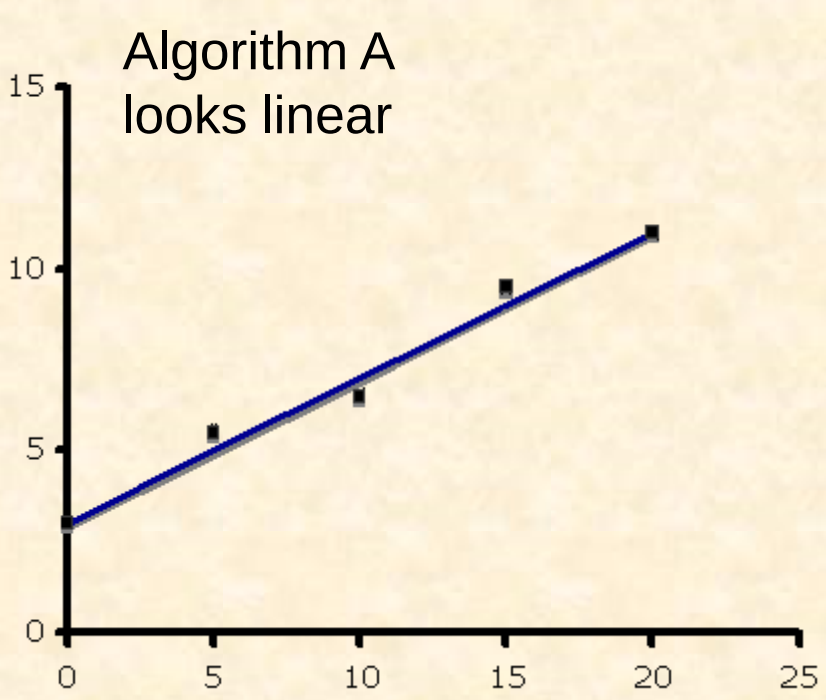
- Suppose we want to check for memory leaks:

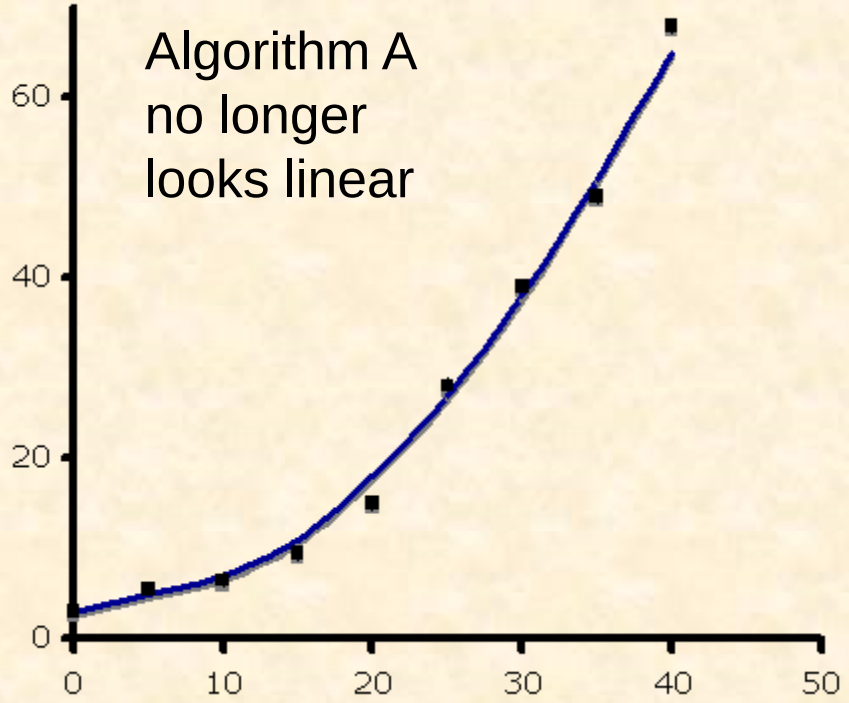
```
valgrind ./search b 1000000
```

- Force a leak!
  - Replace `return 0` with `exit(0)`, run valgrind using flags `--leak-check=full --show-leak-kinds=all`
- Who leaked that memory?
  - The memory address isn't very useful, we just know that `main()` called `operator new`
  - Recompile with `-g3` instead of `-O3` and run valgrind one more time

Empirical Results

- Plot actual run time of algorithm on varying input sizes
- Include a large range to accurately display trend





- Caveat: for small inputs, asymptotics may not play out yet

Prediction versus Experiment

- What if experimental results are *worse* than predictions?
  - Example: results are exponential when analysis is linear
  - Error in complexity analysis
  - Error in coding (check for extra loops, unintended operations, etc.)
- What if experimental results are *better* than predictions?
  - Example: results are linear when analysis is exponential
  - Experiment may not have fit worst case scenario
  - Error in complexity analysis
  - Error in analytical measurements
  - Incomplete algorithm implementation
  - Algorithm implemented is better than the one analyzed
- What if experimental data match asymptotic prediction but runs are too slow?
  - Performance bug?
  - Did you remember to compile with `-O3`??
  - Look for optimizations to improve the constant factor.

Recall: Job Interview Question

- Implement this function

```
// returns x^n
int power(int x, unsigned int n);
```

- The obvious solution uses n - 1 multiplications
  - $2^8 = 2*2* \ldots *2$
- Less obvious: O(log **n**) multiplications
  - Hint: $2^8 = ((2^2)^2)^2$
  - Does it work for 2^7 ?
- Write two solutions

Computing x^n

```
1   int power(int x, int n) {
2     if (n == 0) {
3       return 1;
4     }
5
6     int result = x;
7     for (int i = 1; i < n; ++i) {
8       result = result * x;
9     }
10
11    return result;
12  } // power()
```

Revisiting Recursion

- Recall your EECS 280:
- *Iterative* functions use loops
- A function is *recursive* if it calls itself

Revisiting Recursion

Iterative

```
1  int power(int x, int n) {
2    int result = 1;
3    for (int i = 0; i < n; ++i) {
4      result = result * x;
5    }
6
7    return result;
8  }
```

$\Theta(n)$
It's just a regular loop.

Recursive

```
9  int power(int x, int n) {
10   if (n == 0) {
11     return 1;
12   }
13   return x * power(x, n - 1);
14 }
```

???
We need another tool to analyze this.

What is the time complexity of each function?

Recurrence Relations

- A *recurrence relation* describes the way a problem depends on a subproblem.
  - We can write recurrences for a computation:

$$x^n = \begin{cases} 1 & n == 0 \\ x \cdot x^{n-1} & n > 0 \end{cases}$$

  - We can also write recurrences for the time taken!

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n-1) + c_1 & n > 0 \end{cases}$$

Solving Recurrences

- Recursion tree method [CLRS]
- Substitution method
  1. Write out $T(n) = \ldots T(n - 1) + \ldots T(n - 2) \ldots$
  2. Substitute $T(n - 1)$, $T(n - 2)$ with equation for $T(n)$
  3. Look for a pattern
  4. Use a summation formula

Solving Recurrences Example

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n-1) + c_1 & n > 0 \end{cases}$$

```
1  int power(int x, int n) {
2    if (n == 0) {
3      return 1;
4    }
5    return x * power(x, n - 1);
6  }
```

Recall: Job Interview Question

- Implement this function

```
//returns x^n
int power(int x, unsigned int n);
```

- The obvious solution uses n - 1 multiplications
  - $2^8 = 2*2* \ldots *2$
- Less obvious: O(log **n**) multiplications
  - Hint: $2^8 = ((2^2)^2)^2$
  - Does it work for 2^7 ?
- Write two solutions

Revisiting Recursion

Recursive #1

```
1  int power(int x, int n) {
2    if (n == 0) {
3      return 1;
4    }
5    return x * power(x, n - 1);
6  }
```

Recurrence: $T(n) = T(n - 1) + c$
Complexity: $\Theta(n)$

Recursive #2

```
1  int power(int x, int n) {
2    if (n == 0) {
3      return 1;
4    }
5    int result = power(x, n / 2);
6    result *= result;
7    if (n % 2 != 0) { // x is odd
8      result *= x;
9    }
10   return result;
11 }
```

Recurrence: $T(n) = T(n / 2) + c$
Complexity: $\Theta(\log n)$

Solving Recurrences Example

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n/2) + c_1 & n > 0 \end{cases}$$

```
1   int power(int x, int n) {
2     if (n == 0) {
3       return 1;
4     }
5     int result = power(x, n / 2);
6     result *= result;
7     if (n % 2 != 0) { // x is odd
8       result *= x;
9     }
10    return result;
11  }
```

Binary Search Complexity

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n/2) + c_1 & n > 0 \end{cases} \quad \rightarrow \quad \Theta(\log n)$$

- Fits this same recurrence, because we cut the search space in half each time, and recurse into only one half.

- What if we had a different problem where we cut the space in three pieces? Or what if we cut in two pieces, but had to recursively process both?
  - Next time, we'll introduce the "Master Theorem", which serves as a shortcut for solving these recurrences.

Revisiting Tail Recursion

- Recall your EECS 280:
- When a function is called, it gets a *stack frame*, which stores the local variables
- A simply recursive function generates a stack frame for each recursive call
- A function is *tail recursive* if there is no pending computation at each recursive step
  - "Reuse" the stack frame rather than create a new one
- Tail recursion and iteration are equivalent

http://xkcd.com/1270/

Recursion vs. Tail Recursion

```
1  int factorial(int n) {
2    if (n == 0)
3      return 1;
4    return n * factorial(n - 1);
5  } // factorial()
```

Recursive

```
6   int factorial(int n, int result = 1) {
7     if (n == 0)
8       return result;
9     else
10      return factorial(n - 1, result * n);
11  } // factorial()
```

Tail recursive

Revisiting Recursion

Recursive

```
1 int factorial(int n) {
2   if (n == 0)
3     return 1;
4   return n * factorial(n - 1);
5 } // factorial()
```

O(n) time complexity, but uses *n* stack frames

Revisiting Recursion

Tail Recursive

```
1 int factorial(int n, int result = 1) {
2   if (n == 0)
3     return result;
4   else
5     return factorial(n - 1, result * n);
6 } // factorial()
```

O(n) time complexity, uses only one stack frame

Binomial Coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Binomial Coefficient – "*n* choose *k*"
- Write this function with pen and paper
- Compile and test what you've written
- Options
  - Iterative
  - Recursive
  - Tail recursive
- Analyze

Pseudocode

- The rest of these slides are provided as a reference
  - We won't be going over them in lecture
- We'll be using pseudocode for some algorithms throughout the course
- We won't ask you to write any on exams, but we might ask you to read some, and possibly analyze it for complexity

Pseudocode Introduction

“If you still think that the hard part of programming is the syntax, you are not even on the first rung of the ladder to enlightenment. What's actually hard is abstract thinking, designing algorithms, understanding a problem so well that you can turn it into instructions simple enough for a machine to follow.

All that weird syntax is not there to confuse the uninitiated. It's there in programming languages because it actually makes it easier for us to do the genuine hard work.”

- Taken from: http://www.quora.com/Computer-Programming/What-are-some-of-the-most-common-misconceptions-myths-about-programming

Issue: Clearly Describe an Algorithm

```
if input[i] = '(' or '[' or '{'…

if (input[i] == '(' || '[' || '{')…

if ((input[i] == '(') ||
    (input[i] == '[') ||
    (input[i] == '{'))…
```

The Tower of Babel
by Pieter Bruegel the Elder (1563)

Solution: Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

- Key property:
  - no side-effects

```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output max element of A

    currentMax = A[0]
    for i = 1 to n – 1
        if A[i] > currentMax
            currentMax = A[i]
    return currentMax
```

Language-Agnostic Approach

- In theory
  - Efficiency is independent of implementation language
  - Efficiency is a property of the *algorithm* and not the language

  $\Rightarrow$ Use a **descriptive language** that translates into C++/C#/Java, but **does not distract from ideas**
- In practice
  - Some languages incur a significant slowdown
  - This affects some algorithms & DS more than others
  - ***Asymptotic complexity* is language-independent**
  - Well-defined pseudocode in our textbook
  - Different, well-defined pseudocode from other sources

Pseudocode Syntax

Assignment

- =
- Can write `i = j = e`
- In other sources may be ←

Test for equality

- ==
- In other sources may be =

Pseudocode Syntax

Method declaration

- **Algorithm *method*(*parm1, parm2*)**
  - **Input** …
  - **Output** …

Method/function call

- ***method*(*arg1, arg2*)**

Pseudocode Syntax

Return value

- **`return <expression>`**
- Assume that simple parameters (int, char, …) are passed by value
- Assume that complex parameters (containers) are passed by reference

In pseudocode, can return multiple items

Pseudocode Syntax

Control flow

- **`if`** … **`[else`** …**`]`**
- **`while`** …
- **`do`** … **`while`** …
- **`for`** …
  - `for i = 1 to n`
  - `for j = n downto 1`
  - `for k = 1 to n by x`

Indentation replaces braces

Pseudocode Syntax

Array indexing

- **`A[i]`** is i^{th} cell in array **`A`**
- In lecture slides **`A[0]`**… **`A[n-1]`** unless otherwise specified
- In text (CLRS) **`A[1]`**… **`A[n]`**

Pseudocode Syntax

Comments

- `// comments go here,`
- `/* just like in C/C++ */`

Expressions

- n^2 superscripts and other math formatting allowed

Actual C++ Code vs. Pseudocode

```
1  int arrayMax(int A[], int n) {
2    int currentMax = A[0];
3
4    for (int i = 1; i < n; i++)
5      if (currentMax < A[i])
6        currentMax = A[i];
7
8    return currentMax;
9  } // arrayMax()
```

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output max element of A

  currentMax = A[0]
  for i = 1 to n − 1
    if A[i] > currentMax
      currentMax = A[i]
  return currentMax
```

Pseudocode in Practice

- Written for a human, not computer
- Pros
  - Can be used with many programming languages
  - Communicates high-level ideas, not low-level implementation details
- Cons
  - Omit important steps, can be *mis*interpreted
  - Does not run
  - No tools to check correctness $\Rightarrow$ often *"pseudocorrect"*
  - Requires significant effort to use in practice

When (not) to Use Pseudocode

- One textbook uses pseudocode
- The other teaches C++ STL usage
- Wikipedia uses pseudocode and often C++, Java, Python
- Our projects use C++
- Job interviews
  - Usually require writing real code (C++, C#, Java)
  - Companies selling a product usually use a specific language
  - Other companies give you a choice or require both C++ and Java (e.g., “compare C++ and Java implementations of this algorithm”)

We want you to practice writing C++ code, compiling it and checking if it is correct

- Large-scale algorithm design uses pseudocode
- Being fluent in OO programming, you can use real C++ or Java to express high-level ideas

Pseudocode Summary

- Pseudocode expresses algorithmic ideas
  - Good for learning & looking up algorithms
- Pseudocode avoids details
  - Good: can be read quickly
  - Good: usable with C++, C#, Java, etc
  - Good: implementable in many ways
  - Bad: details need to be added
  - Bad: can harbor subtle mistakes
  - Bad: translation mistakes
- Pitfalls
  - Hidden complexity
  - Neglect for advanced language features
  - Professional programmers often bypass pseudocode