Lecture 3
Measuring Runtime, and Pseudocode

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

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

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

Ways to measure complexity

- We now know how to model complexity mathematically
- Today, we will measure it empirically

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`

Measuring Runtime on Linux

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

```
% /usr/bin/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

Measuring Time In C

```
struct timeval{
    //--- seconds
    unsigned int tv_sec;

    //--- microseconds
    unsigned int tv_usec;
};
```

```
struct rusage{
    //--- user time used
    struct timeval ru_utime;

    //--- system time used
    struct timeval ru_stime;
    …
};
```

```
#include <iostream>
#include <sys/resource.h>
#include <sys/time.h>

void main(){
    struct rusage startu;
    struct rusage endu;

    getrusage(RUSAGE_SELF, &startu);
    //---- Do computations here
    getrusage(RUSAGE_SELF, &endu);

    double start_sec = startu.ru_utime.tv_sec + startu.ru_utime.tv_usec/1000000.0;
    double end_sec = endu.ru_utime.tv_sec + endu.ru_utime.tv_usec/1000000.0;
    double duration = end_sec - start_sec;
}
```

Initialize rusage variables

Set data of rusage variables at start/end of computation

Duration = End - Start

Measuring Time In C++

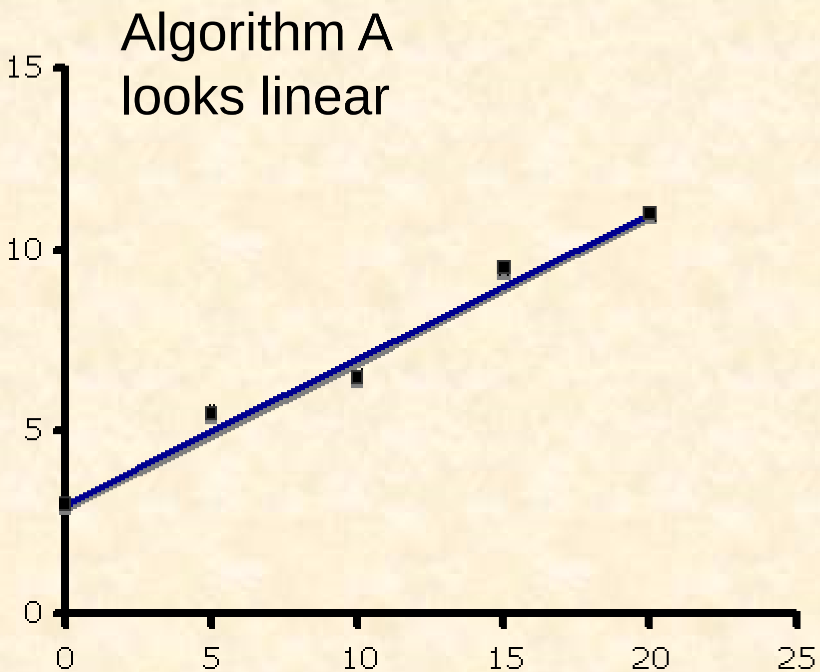
```
#include <iostream>
#include <sys/resource.h>
#include <sys/time.h>

class Timer {
private:
  struct rusage startu;
  struct rusage endu;
  double duration;
public:
  Timer() { getrusage(RUSAGE_SELF, &startu); }

  void recordTime() {
    getrusage(RUSAGE_SELF, &endu);
    double start_sec = startu.ru_utime.tv_sec + startu.ru_utime.tv_usec/1e7;
    double end_sec  = endu.ru_utime.tv_sec +endu.ru_utime.tv_usec/1e7;
    duration = end_sec - start_sec;
  }

  double getTime() { return duration; }
};
```

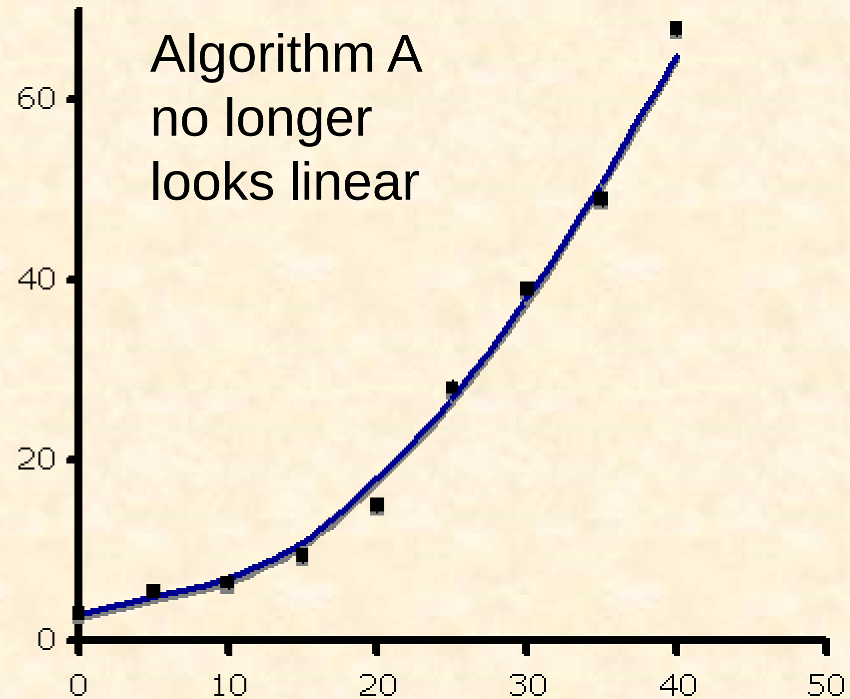
When you create an instance of the class, it begins timing. When you call `recordTime()`, it saves the amount of time that has passed between the instantiation of the object and the method call `getTime()` returns the last saved amount of time

Checking time too often will slow down your program

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
- Caveat: the Earth has only 7B people, 1T Web pages

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)
- What if experimental results are *better* than predictions?
  - Example: results are linear when analysis is exponential
  - Experiment may not have fit worst case scenario (no mistake)
  - 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?

Additional Uses of the Timer Class

1. Find out which line is faster

```
printf("I am taking %d\n", courseNum);
cout << "I am taking " << courseNum << "\n";
cout << "I am taking " << courseNum << endl;
```

   – Embed each line in a for-loop and time them
   – May take numerous repetitions to get non-0 time

2. A large program can print a “self-profile” after each execution, assigning a % of total runtime to each meaningful part of the program

Part 2: Pseudocode

“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 older slides may be ←

Test for equality

- ==
- In older slides 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

- $\mathbf{n^2}$ superscripts and other math formatting allowed

Actual C++ Code vs. Pseudocode

```
int arrayMax(int A[], int n) {
  int currentMax = A[0];

  for (int i = 1; i < n; i++)
    if (currentMax < A[i])
      currentMax = A[i];

  return currentMax;
}
```

```
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
  - <u>No tools to check correctness</u> $\Rightarrow$ 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

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

Exercise: C++ to Pseudocode

```
int power(int x, unsigned y, int result = 1) {
  if (y == 0)
    return result;
  else if (y % 2)
    return power(x * x, y / 2, result * x);
  else
    return power(x * x, y / 2, result);
}
```

Revisiting Recursion

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

Revisiting Recursion

Iterative

```
int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    n--;
  }
  return result;
}
```

Recursive

```
int factorial(int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

What is the complexity of each function?

Revisiting Recursion

Iterative

```
int factorial(int n) {
  int result = 1;
  while (n > 0) {
    result *= n;
    n--;
  }
  return result;
}
```

$O(n)$ complexity

Recursive

```
int factorial(int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

We need another tool to solve this

Recurrence Equations

- A *recurrence equation* describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs. [CLRS]

Recurrence Equation Example

```
int factorial (int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

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

- $T(n)$ is the running time of `factorial()` with input size n
- $T(n)$ is expressed in terms of the running time of `factorial()` with input size n - 1
- c_0 and c_1 are constants

Solving Recurrences

- Recursion tree method [CLRS]
- AKA Telescoping method

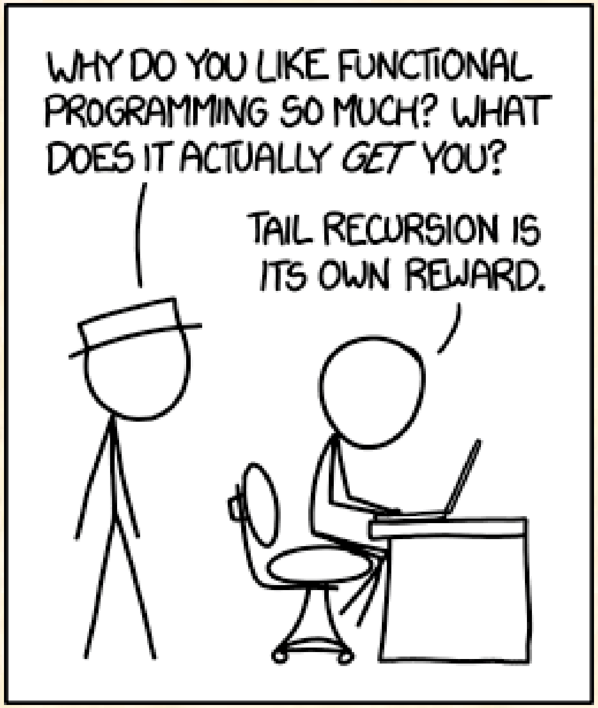
1. Write out $T(n)$, $T(n - 1)$, $T(n - 2)$
2. Substitute $T(n - 1)$ and $T(n - 2)$ into $T(n)$
3. Look for a pattern
4. Use a summation formula

Solving Recurrences Example

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

Revisiting Tail Recursion

- Recall your EECS 280:
- When a function is called, it gets a *stack frame*, which stores the local variables
- A 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
  - "Re-use" the stack frame rather than create a new one
- Tail recursion and iteration are equivalent



http://xkcd.com/1270/

Revisiting Recursion

```
int factorial(int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

Recursive

```
int factorial(int n, int result = 1) {
  if (n == 0)
    return result;
  else
    return factorial(n - 1, result * n);
}
```

Tail recursive

Exercise

- Write the recurrence equation
- Solve the recurrence equation
- Express your answer in Big-O notation

```
int factorial(int n, int result = 1) {
  if (n == 0)
    return result;
  else
    return factorial(n - 1, result * n);
}
```

Revisiting Recursion

Recursive

```
int factorial(int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

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

Revisiting Recursion

Tail Recursive

```
int factorial(int n, int result = 1) {
  if (n == 0)
    return result;
  else
    return factorial(n - 1, result * n);
}
```

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