

# CS221: Algorithms and Data Structures

## Asymptotic Analysis

Alan J. Hu

(Borrowing slides from Steve Wolfman)

# Learning Goals

By the end of this unit, you will be able to...

- Define which program operations we measure in an algorithm in order to approximate its efficiency.
- Define “input size” and determine the effect (in terms of performance) that input size has on an algorithm.
- Give examples of common practical limits of problem size for each complexity class.
- Give examples of tractable, intractable, and undecidable problems.
- Given code, write a formula which measures the number of steps executed as a function of the size of the input ( $N$ ).

*Continued<sup>2</sup>...*

# Learning Goals

By the end of this unit, you will be able to...

- Compute the worst-case asymptotic complexity of an algorithm (e.g., the worst possible running time based on the size of the input ( $N$ )).
- Categorize an algorithm into one of the common complexity classes.
- Explain the differences between best-, worst-, and average-case analysis.
- Describe why best-case analysis is rarely relevant and how worst-case analysis may never be encountered in practice.
- Given two or more algorithms, rank them in terms of their time and space complexity.

# Today's Learning Goals/Outline

- Why and on what criteria you might want to compare algorithms
- Performance (time, space) is a function of the inputs.
  - We usually simplify that to be a function of the **size** of the input.
  - What are worst-case, average-case, common case, and best case analysis?
- What is and why do asymptotic analysis?
- Examples of asymptotic behavior to build intuition.

# Comparing Algorithms

- Why?
- What do you judge them on?

# Comparing Algorithms

- Why?
- What do you judge them on?

Many possibilities...

- Time (How long does it take to run?)
- Space (How much memory does it take?)
- Other attributes?
  - Expensive operations, e.g. I/O
  - Elegance, Cleverness
  - Energy, Power
  - Ease of programming, legal issues, etc.

# Analyzing Runtime

Iterative Fibonacci:

```
old2 = 1
```

```
old1 = 1
```

```
for (i=3; i<n; i++) {  
    result = old2+old1  
    old1 = old2  
    old2 = result  
}
```

How long does this take?

A second? A minute?

# Analyzing Runtime

Iterative Fibonacci:

```
old2 = 1
```

```
old1 = 1
```

```
for (i=3; i<n; i++) {  
    result = old2+old1  
    old1 = old2  
    old2 = result  
}
```

How long does this take?

A second? A minute?

Runtime depends on  $n$  !

Therefore, we will write it as  
a function of  $n$ .

More generally, it will be a  
function of the input.



# Analyzing Runtime

Iterative Fibonacci:

```
old2 = 1
```

```
old1 = 1
```

```
for (i=3; i<n; i++) {  
    result = old2+old1  
    old1 = old2  
    old2 = result  
}
```

What machine do you run on?

What language?

What compiler?

How was it programmed?

# Analyzing Runtime

Iterative Fibonacci:

```
old2 = 1
```

```
old1 = 1
```

```
for (i=3; i<n; i++) {  
    result = old2+old1  
    old1 = old2  
    old2 = result  
}
```

What machine do you run on?

What language?

What compiler?

How was it programmed?

We want to analyze **algorithm**,  
ignore these details!

Therefore, just count “basic  
operations”, like arithmetic,  
memory access, etc.

# Analyzing Runtime

Iterative Fibonacci:

```
old2 = 1
```

```
old1 = 1
```

```
for (i=3; i<n; i++) {  
    result = old2+old1  
    old1 = old2  
    old2 = result  
}
```

How many operations does this  
take?

# Analyzing Runtime

Iterative Fibonacci:

```
old2 = 1
```

```
old1 = 1
```

```
for (i=3; i<n; i++) {  
    result = old2+old1  
    old1 = old2  
    old2 = result  
}
```

How many operations does this take?

If we're ignoring details, does it make sense to be so precise?

We'll see later how to do this much simpler!

# Run Time as a Function of Input

- Run time of iterative Fibonacci is (depending on details of how we count and our implementation):  
 $3+(n-3)(6)+1$ , simplified to  $6n-14$

# Run Time as a Function of Input

- Run time of iterative Fibonacci is (depending on details of how we count and our implementation):

$$3+(n-3)(6)+1, \text{ simplified to } 6n-14$$

- Since we've abstracted away exactly how long different operations take, and on what computer we're running, does it make sense to say "6n-14" instead "6n-10" or "5n-20" or "3.14n-6.02"???

# Run Time as a Function of Input

- Run time of iterative Fibonacci is (depending on details of how we count and our implementation):

$$3+(n-3)(6)+1, \text{ simplified to } 6n-14$$

- Since we've abstracted away exactly how long different operations take, and on what computer we're running, does it make sense to say “ $6n-14$ ” instead “ $6n-10$ ” or “ $5n-20$ ” or “ $3.14n-6.02$ ”???

What matters is its linear in  $n$ .

(We will formalize this soon.)

# Run Time as a Function of Input

- What if we have lots of inputs?
  - E.g., what is the run time for linear search in a list?



# Run Time as a Function of Input

- What if we have lots of inputs?
  - E.g., what is the run time for linear search in a list?

We could compute some complicated function

$$f(\text{key}, \text{list}) = \dots$$

but that will be too complicated to compare.

# Run Time as a Function of **Size of** Input

- What if we have lots of inputs?
  - E.g., what is the run time for linear search in a list?

Instead, we usually simplify to take the run time only in terms of the “size of” the input.

- Intuitively, this is e.g., the length of a list, etc.
- Formally, it’s the number of bits of input

This keeps our analysis simpler...

# Run Time as a Function of **Size of** Input

- But, **which** input?
  - Different inputs of same size have different run times

E.g., what is run time of linear search in a list?

- If the item is the first in the list?
- If it's the last one?
- If it's not in the list at all?

What should we report?

# Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- etc.

# Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- etc.

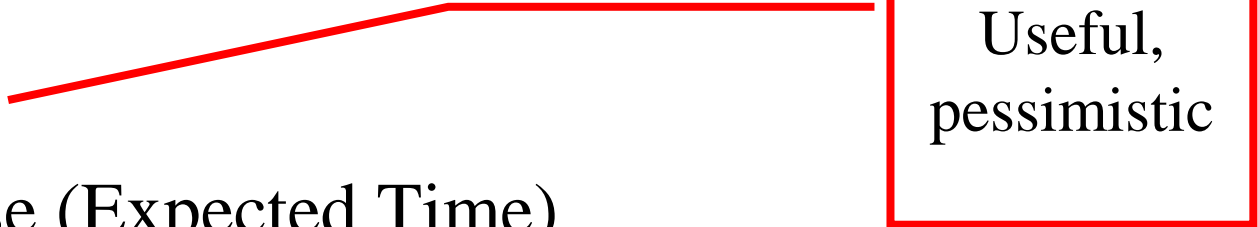


Mostly  
useless

# Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- etc.

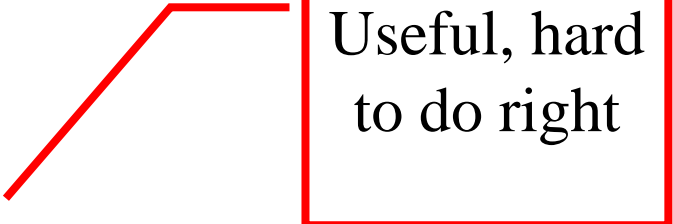


Useful,  
pessimistic

# Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- etc.




Useful, hard  
to do right

# Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- etc.




Very useful,  
but ill-defined



# Which Run Time?

There are different kinds of analysis, e.g.,

- Best Case
- Worst Case
- Average Case (Expected Time)
- Common Case
- Amortized
- etc.



Useful, you'll see  
this in more  
advanced courses

# Multiple Inputs (or Sizes of Inputs)

- Sometime, it's handy to have the function be in terms of multiple inputs
  - E.g., run time of counting how many times string A appears in string B

It would make sense to write the result as a function of both A.length and B.length

# Which BigFib is faster?

- We saw an exponential time, simple recursive Fibonacci, and a log time, more complex Fibonacci.

# Which BigFib is faster?

- We saw an exponential time, simple recursive Fibonacci, and a log time, more complex Fibonacci.
- At  $n=5$ , simple version is faster.
- At  $n=35$ , complex version is faster.

What's more important?

# Scalability!

- Computer science is about solving problems people couldn't solve before.
- Therefore, the emphasis is almost always on solving the big versions of problems.
- (In computer systems, they always talk about “scalability”, which is the ability of a solution to work when things get really big.)

# Asymptotic Analysis

- Asymptotic analysis is analyzing what happens to the run time (or other performance metric) as the input size  $n$  goes to infinity.
  - The word comes from “asymptotes”, which is where you look at the limiting behavior of a function as something goes to infinity.
- This gives a solid mathematical way to capture the intuition of emphasizing scalable performance.
- It also makes the analysis a lot simpler!



# Interpreters, Compilers, Linkers

- Steve tells me that 221 students often find linker errors to be mysterious.
- So, what's a linker?



# Separate Compilation

- A compiler translates a program in a high-level language into machine language.
- A big program can be many millions of lines of code. (e.g., Windows Vista was 50MLoC)
- Compiling something that big takes hours or days.
- The source code is in many files, and most changes affect only a few files.
- Therefore, we compile each file separately!

# Symbol Tables

- How can you compile an incomplete program?
  - Header files tell you the types of the missing functions
    - These are the .h file in C and C++ programs
  - The object code includes a list of missing functions, and where they are called.
  - The object code also includes a list of all public functions declared in it.
  - These lists are called the “symbol table”.

# Linking

- The linker puts all these files together into a single executable file, using the symbol tables to hook up missing functions with their definitions.
  - In C and C++, the executable starts with a function called “main”, like in Java.