UC Berkeley EECS
Lecturer Michael Ball

Computational Structures in Data Science

Lecture #10: Efficiency & Data Structures

Nov 12, 2019 http://inst.eecs.berkeley.edu/~cs88

1

Why?

- **Runtime Analysis:**
  - How long will my program take to run?
  - Why can't we just use a clock?
- **Data Structures**
  - OOP helps us organize our *programs*
  - Data Structures help us organize our data!
  - You already know lists and dictionaries!
  - We'll see two new ones today
- **Enjoy this stuff? Take 61B!**
- **Find it challenging? Don't worry! It's a different way of thinking.**

11/12/19 UCB CS88 Fa19 L10 2

2

Efficiency

How long is this code going to take to run?

11/12/19 UCB CS88 Fa19 L10 3

3

Is this code fast?

- **Most code doesn't *really* need to be fast! Computers, even your phones are already amazingly fast!**
- **Sometimes…it does matter!**
  - Lots of data
  - Small hardware
  - Complex processes
- **We can't just use a clock**
  - Every computer is different? What's the benchmark?

2/22/16 UCB CS88 Sp16 L4 4

4

Runtime analysis problem & solution

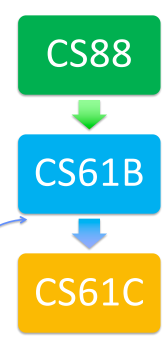
- **Time w/stopwatch, but…**
  - Different computers may have different runtimes. ☹
  - Same computer may have different runtime on the same input. ☹
  - Need to implement the algorithm first to run it. ☹
- ***Solution*: Count the number of "steps" involved, not time!**
  - Each operation = 1 step
  - *If we say "running time", we'll mean # of steps, not time!*

2/22/16 UCB CS88 Sp16 L4 5

5

Runtime: input size & efficiency

- **Definition**
  - Input size: the # of things in the input.
  - E.g., # of things in a list
  - Running time as a function of input size
  - Measures efficiency
- **Important!**
  - In CS88 we won't care about the efficiency of your solutions!
  - …in CS61B we will



6

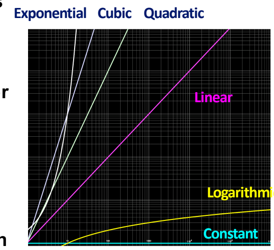
Runtime analysis : worst or avg case?

- Could use avg case
  - Average running time over a vast # of inputs
- Instead: use worst case
  - Consider running time as input grows
- Why?
  - Nice to know most time we'd <u>ever</u> spend
  - Worst case happens often
  - Avg is often ~ worst
- Often called "Big O"
  - We use "Omega" denote runtime

7

Runtime analysis: Final abstraction

- Instead of an exact number of operations we'll use abstraction
  - Want order of growth, or dominant term
- In CS88 we'll consider
  - Constant
  - Logarithmic
  - Linear
  - Quadratic
  - Exponential
- E.g. $10\ n^2 + 4 \log n + n$
  - …is quadratic



Graph of order of growth curves on log-log plot

8

Example: Finding a student (by ID)

- Input
  - <u>Unsorted</u> list of students L
  - Find student S
- Output
  - True if S is in L, else False
- Pseudocode Algorithm
  - Go through one by one, checking for match.
  - If match, true
  - If exhausted L and didn't find S, false

- Worst-case running time as function of the size of L?
  1. Constant
  2. Logarithmic
  3. Linear
  4. Quadratic
  5. Exponential

9

Example: Finding a student (by ID)

- Input
  - <u>Sorted</u> list of students L
  - Find student S
- Output : same
- Pseudocode Algorithm
  - Start in middle
  - If match, report true
  - If exhausted, throw away half of L and check again in the middle of remaining part of L
  - If nobody left, report false

- Worst-case running time as function of the size of L?
  1. Constant
  2. Logarithmic
  3. Linear
  4. Quadratic
  5. Exponential

10

Computational Patterns

- If the number of steps to solve a problem is always the same → Constant time: O(1)
- If the number of steps increases similarly for each larger input → Linear Time: O(n)
  - Most commonly: `for each item`
- If the number of steps increases by some a factor of the input → Quadradic Time: $O(n^2)$
  - Most commonly: Nested for Loops
- Two harder cases:
  - Logarithmic Time: O(log n)
    - » We can double our input with only one more level of work
    - » Dividing data in "half" (or thirds, etc)
  - Exponential Time: $O(2^n)$
    - » For each bigger input we have 2x the amount of work!
    - » Certain forms of Tree Recursion

11

11

Comparing Fibonacci

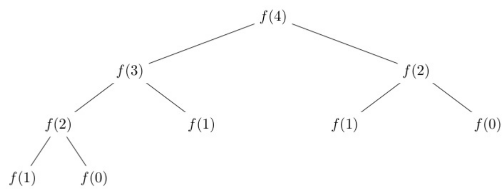
```
def iter_fib(n):
    x, y = 0, 1
    for _ in range(n):
       x, y = y, x+y
    return x


def fib(n): # Recursive
    if n < 2:
       return n
    return fib(n - 1) + fib(n - 2)
```

12

12

Tree Recursion

- Fib(4) → 9 Calls
- Fib(5) → 16 Calls
- Fib(6) → 26 Calls
- Fib(7) → 43 Calls
- Fib(20) →



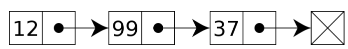
What next?

- Understanding ***algorithmic complexity*** **helps us know whether something is possible to solve.**
- **Gives us a formal reason for understanding why a program might be slow**
- **This is only the beginning:**
  - We've only talked about time complexity, but there is *space complexity.*
  - In other words: How much memory does my program require?
  - Often times you can trade time for space and vice-versa
  - Tools like "caching" and "memorization" do this.
- **If you think this is cool take CS61B!**

Linked Lists

Linked Lists

- **A series of items with two pieces:**
  - A value
  - A "pointer" to the next item in the list.



- **We'll use a very small Python class "Link" to model this.**