Computational Structures in Data Science

Efficiency
& Run Time Analysis

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org | © CC BY-NC-SA

Announcements

- **Reminder to practice using pen & paper, notebooks, etc.**
- Use the extensions form, please don't email for extensions
  - https://go.c88c.org/extensions
  - Post on ed first, please!
    - Way more staff on ed than on email.
- Review and Exam Prep sections starting this week (tomorrow!)
  - Check the CS88 Calendar
- Reminder:
  - MT Survey
  - Regrade requests close tomorrow.

Computational Structures in Data Science

Efficiency
& Run Time Analysis

Berkeley
UNIVERSITY OF CALIFORNIA

© CC BY-NC-SA

Learning Objectives

- Runtime Analysis:
  - How long will my program take to run?
  - Why can’t we just use a clock?
  - How can we simplify understanding computation in an algorithm
- Enjoy this stuff? Take 61B!
- Find it challenging? Don’t worry! It’s a different way of thinking.

Efficiency is all about trade-offs

- Running Code: Takes Time, Requires Memory
  - More efficient code takes less time or uses less memory
- Any computation we do, requires both time and "space" on our computer.
- Writing efficient code is not obvious
  - Sometimes it is even convoluted!
- But!
- We need a framework before we can optimize code
- Today, we're going to focus on the time component.

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
- Slow code takes up battery power

Beware!

"Premature Optimization is the root of all evil"

- Donald Knuth, Stanford CS Professor

There is **no use** in fast code if it is wrong!

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
    - 1 + 2 is one step
    - lst[5] is one step
  - *When we say "runtime", we'll mean # of steps, not time!*

Runtime: input size & efficiency

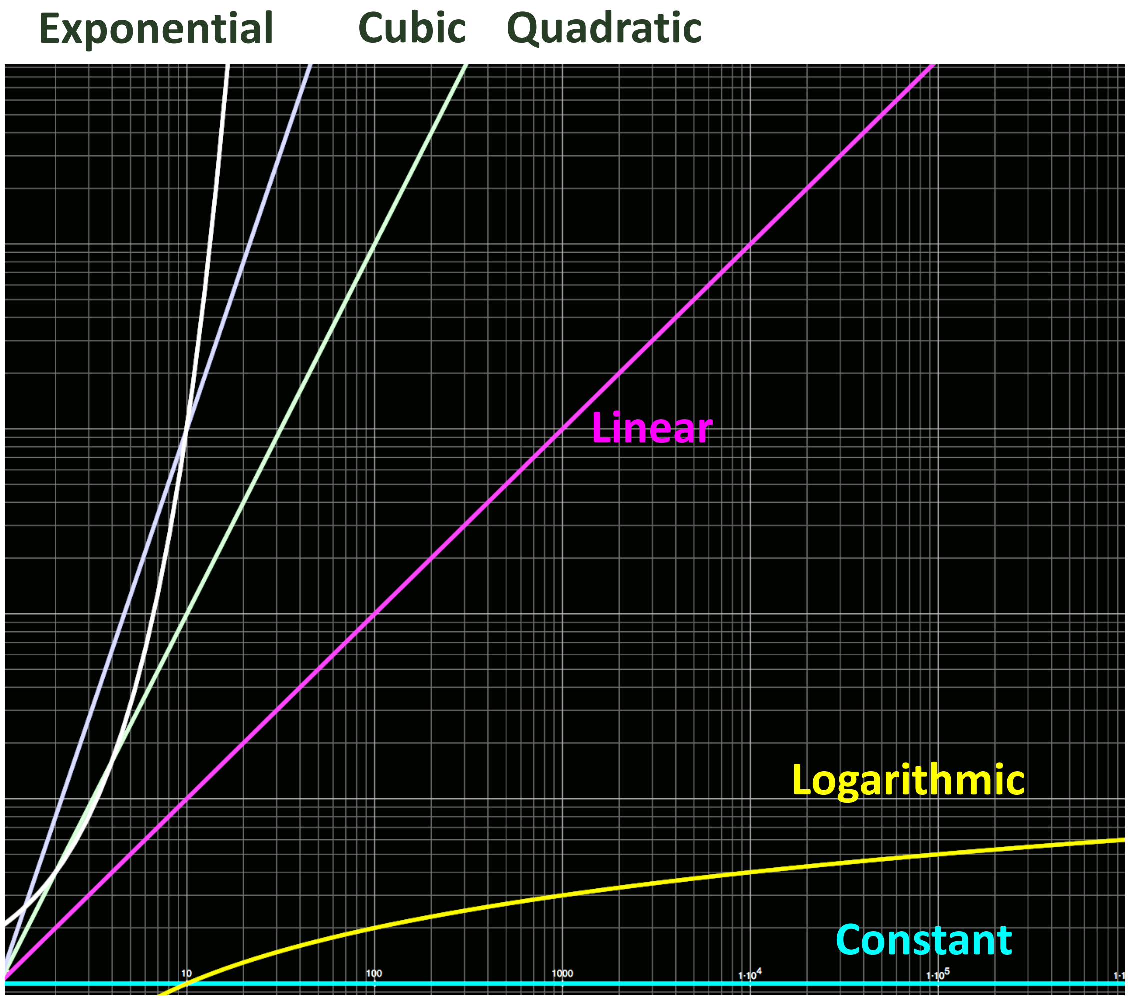
- Definition:
  - **Input size**: the # of things in the input.
  - e.g. length of a list, the number of iterations in a loop.
  - Running time as a function of input size
  - Measures **efficiency**
- Important!
  - In CS88 <u>we won't care</u> about the efficiency of your solutions!
  - ...in CS61B we will

Runtime analysis : worst or average 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
  - The "average" can be similar to the worst
- Often called "Big O" for "order"
  - O(1), O(n) …

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 $O(1)$
  - Logarithmic $O(\log n)$
  - Linear $O(n)$
  - Quadratic $O(n^2)$
  - Exponential $O(2^n)$
- E.g. $10n^2 + 4\log(n) + n$
  - …is quadratic



Graph of order of growth curves on log-log plot

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

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

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

Efficiency of Linked Lists vs Lists

- Linked Lists generally use less memory.
  - But this can make it slower to *compute* data.
- Linked Lists:
  - Once you've found an item, inserting / removing is easy, O(1)
  - Finding anything other than the first/last item is O(n)
- "Regular" Lists:
  - Inserting / Removing items, other than the last is O(n) – due to internal copying
  - Finding any random item is O(1).

- What if you need to iterate over all items in order?
  - O(n) in both cases

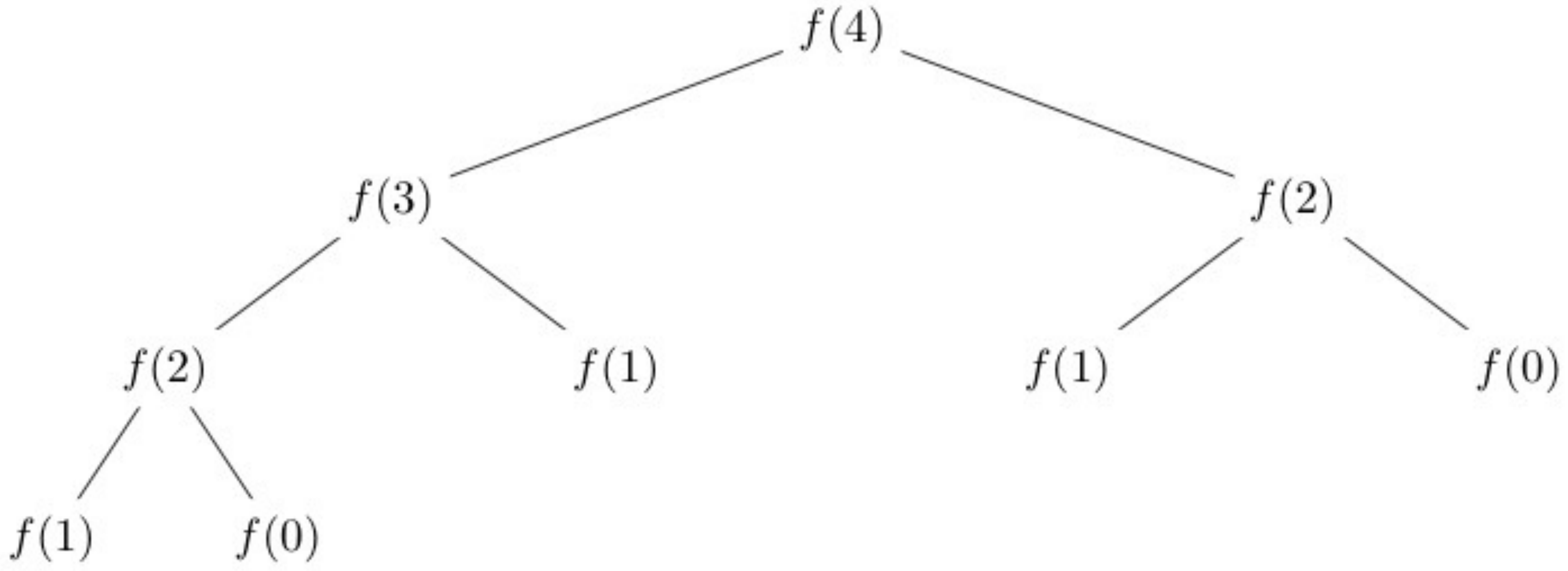
Comparing Fibonacci

```
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)
```

Tree Recursion

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



Why?

- Notice there was all this duplication in the tree?
- What is the exact order of growth?
  - It's exponential.
  - phi to the N (φ^{n}), where phi is the golden ratio.

N	Operations
1	1
2	3
3	5
4	9
7	41
8	67
20	21891

Computational Structures in Data Science

Improving Efficiency

Learning Objectives

- Learn how to cache the results to save time.
- "memoization" is a specific version to avoid repeated calculations

Example

- Use a dictionary to cache results.
- This is called *memoization*

```
fib_results = {}
def memo_fib(n): # Look up values in our dictionary.
    global fib_results
    if n in fib_results:
        print(f'found {n} -> {fib_results[n]}')
        return fib_results[n]
    if n < 2:
        fib_results[n] = n
        return n
    result = memo_fib(n - 1) + memo_fib(n - 2)
    fib_results[n] = result
    return result
```

A Better Approach

- Python's functools module has a `cache` function
- Uses a technique called decorators that we don't cover.
  - Decorators are really just a "shortcut" for higher order functions.
  - e.g. `cache_fib = cache(fib)` is a similar approach to the function below, but less commonly used.

```
from functools import cache


@cache
def cache_fib(n): # Recursive
    if n < 2:
        return n
    return cache_fib(n - 1) + cache_fib(n - 2)
```

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 you can trade time for space and vice-versa
  - Tools like "caching" and "memorization" do this.

- If you think this is cool take CS61B!