CS 61A SU26

Lecture 17: Efficiency

July 21, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Announcements
- Computing in the News

Announcements

- Reminder: Due to the Soda → Gateway building move, some discussion sections will be in McLaughlin starting this week
  - **UPDATE: Disc sections moving to Grimes will actually be there starting the week of 8/3, NOT this week**
- Reminders:
  - Project problems marked optional or "EC" or "Extra Challenge" are NOT extra credit; they are not worth any points
  - Do not discuss the contents of the quiz outside of when you're going over your quiz with a staff member in OH. **Violations are considered academic misconduct.**

Computing in the News

- Scalable Detection of Adversarial Synthetic Slop and Coordinated Media Abuse: A LoRA-Enabled Multimodal Defense System (Google Research)
  - "Operational data spanning a **six-month period** demonstrates the system's significant impact, resulting in the **successful termination of 50K clusters comprising 130K channels of synthetic spam generators**. Furthermore, the LLM-driven automation significantly improves operational efficiency, **saving approximately 83 human review hours** to **cut down human reviews by 50%**."
  - Interested in social media algorithms? Take CS 194/294-273 (Designing Algorithmic Media, taught by Prof. Jonathan Stray, last offered SP26)

Review

10 min

- Link to Diagram
- WWPD

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Which of the following diagrams corresponds to this Link object? Link(1, Link(Link(2, Link(3)), Link(Link(4), Link(5))))

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Linked Lists: What would Python display?

Efficiency

30 min

- Motivation
- Exponentiation
- Fibonacci
- Orders of Growth
- Time Complexity
- Space Complexity

Motivation

- There are often many ways to solve the same problem
- How do we judge whether one algorithm is "better" than another?
- One possible way: Algorithm A is better than Algorithm B if Algorithm A **runs faster** (A is more **efficient** than B).
    - Ex: Linked list vs. Python list insertion at the front demo from the previous lecture
- To create effective software at scale, you **must** consider efficiency
    - A poorly designed system might have long wait times (latency) → users won't like it!

Exponentiation (1 of 3)

Exponentiation is the mathematical operation of taking some number b and multiplying it by itself n times. b is the base, n is the exponent.

You can also define this recursively:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b * b^{n-1} & \text{if } n \geq 1 \end{cases}$$

```
def exp(b: int, n: int) -> int:
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

n	Calls to exp
0	1
1	2
2	3
…	…
n	n + 1

Linear time!

Exponentiation (2 of 3)

Here's a different recursive definition:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ \left(b^{\frac{1}{2}n}\right)^2 & \text{if } n \text{ is even} \\ b * (b^{n-1}) & \text{if } n \text{ is odd} \end{cases}$$

```
def exp(b: int, n: int) -> int:
    if n == 0:
        return 1
    elif n % 2 == 0:
        temp = exp(b, n // 2)
        return temp * temp
    else:
        return b * exp(b, n - 1)
```

Exponentiation (3 of 3)

n	Calls to exp
0	1
1	2 (1 → 0)
2	3 (2 → 1 → 0)
8	5 (8 → 4 → 2 → 1 → 0)
64	8 (64 → 32 → 16 → 8 → 4 → 2 → 1 → 0)

```
def exp(b: int, n: int) -> int:
    if n == 0:
        return 1
    elif n % 2 == 0:
        temp = exp(b, n // 2)
        return temp * temp
    else:
        return b * exp(b, n - 1)
```

Logarithmic time!

Fibonacci: Recursive (1 of 3)

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Try `virfib(5)`: https://www.recursionvisualizer.com/

How many function calls are made relative to the input size `n`?

Exponential time!

Demo: `fib(100)`

Fibonacci: Memoization (2 of 3)

- We make a lot of redundant calls to `fib`, but the behavior and return value don't change between calls (`fib` is a pure function). How can we fix this?
- **Memoization** is a technique where we cache (e.g. save) values we have already computed, so that we don't have to compute them again.
  - Note: Decorators are not in scope for this class, but the idea of memoization is
- **Demo:** `fib(100)`

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized


@memo
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)
```

Fibonacci: Iterative (3 of 3)

```
def fib(n):
    curr, nxt = 0, 1
    k = 0
    while k < n:
        curr, nxt = nxt, curr + nxt
        k += 1
    return curr
```

How many operations are there relative to the input size n?

Linear time!

Orders of Growth (1 of 4)

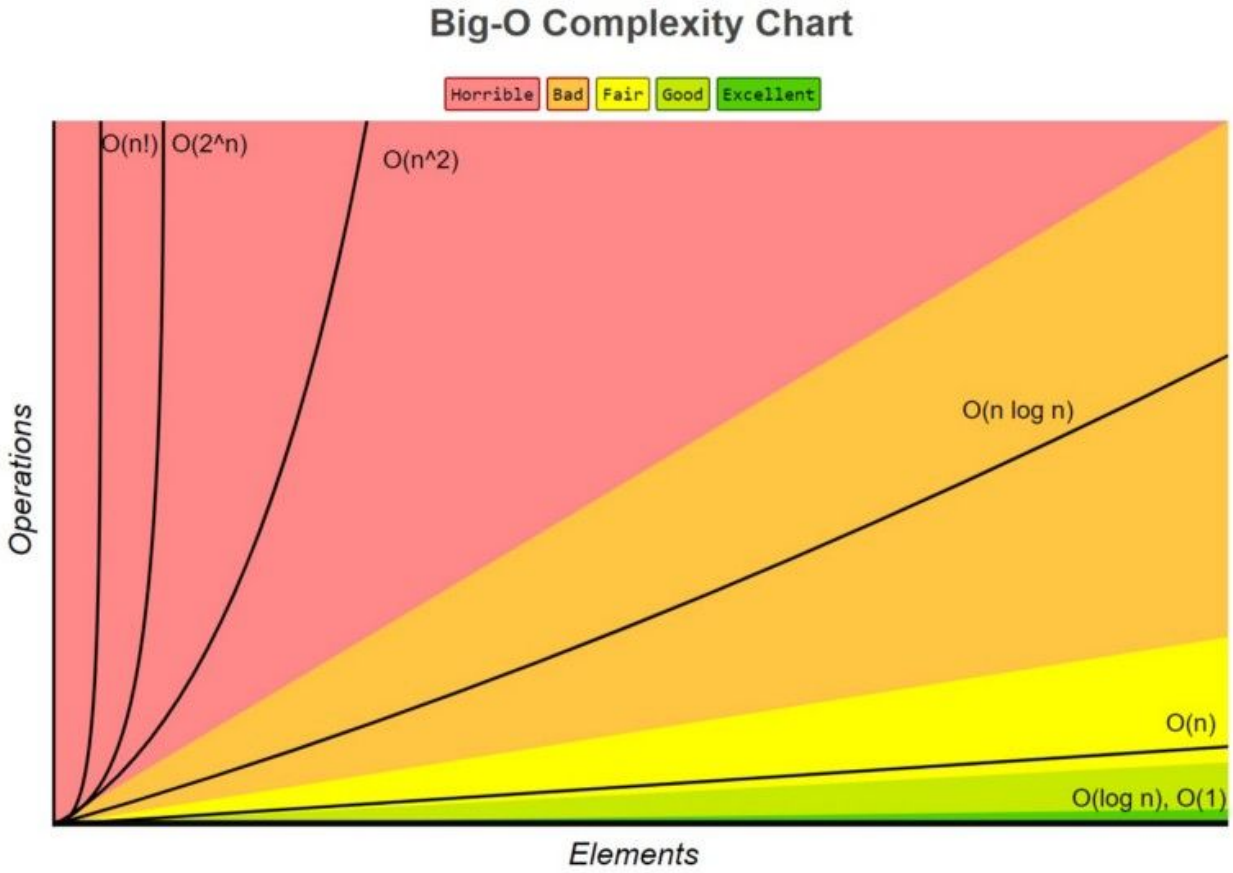
- Notice that in the previous examples, we described the runtime of an algorithm using a mathematical function (linear, logarithmic, exponential)
- Why don't we just use...
  - Time
    - Differences between hardware means that some code might run for a totally different amount of time between 2 machines
    - Even within the same machine, there can be slight variations
  - Lines of code
    - There can be varying levels of operations/complexity in a single line of code
    - Programming languages differ in syntax, even if the algorithm is the same
  - # of function calls
    - Each function has a different number of operations
- Also notice that we define the runtime relative to some input size (usually n, but you could have multiple inputs)

Orders of Growth (2 of 4)

Here are common orders of growth you should know for this class, from fastest to slowest:

Order of Growth	Big O Notation	Description
Constant	O(1)	Increasing n doesn't affect runtime
Logarithmic	O(log n)	Doubling n only increments runtime by a constant (technically this is for $O(\log_2 n)$)
Linear	O(n)	Incrementing n only increments runtime by a constant
Quadratic	$O(n^2)$	Incrementing n increases runtime by n times a constant
Exponential	$O(2^n)$	Incrementing n multiplies runtime by a constant (base doesn't have to always be 2)

Orders of Growth (3 of 4)



Orders of Growth (4 of 4)

- Note: **Big O notation** technically refers to the **upper bound** of the runtime of an algorithm.
  - Technically, you could say that iterative Fibonacci is $O(2^n)$ since it runs in at most exponential time
- For our class, generally when we ask you what the runtime is, we're asking for the **exact worst case runtime** (not the best case or average case) unless otherwise stated
  - e.g. we'd expect you on an exam to say "iterative Fibonacci is O(n) time"

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Algorithm #1: What is the runtime of this function with respect to n?

Time Complexity (1 of 4)

The formal process of analyzing an algorithm's runtime is called **time complexity analysis** (aka **runtime analysis** or **asymptotic analysis**)

IMPORTANT: There is no magic formula to determine the runtime of a program but these are some starting points.

1. Read through the code line by line and determine how much time each line takes
2. If there are loops, it can be helpful to multiply however much time the inside of the loop takes by the number of iterations there are
3. If there is recursion, it can be helpful to draw out the tree of recursive calls and add up how much “work” is done at each node in the tree
4. Drop constant factors and lower-order terms

Time Complexity (2 of 4)

Big O Notation	Name	Examples
O(1)	Constant	• Arithmetic • Variable (re)assignment • Appending to the end of a list • Adding/retrieving/updating a key-value pair in a dictionary
O(log n)	Logarithmic	• Input is being divided by some constant factor for each loop or recursive call, e.g. `exp`
O(n)	Linear	• Iterating through each element in a sequence or dictionary
$O(n^2)$	Quadratic (more generally n^k is called polynomial)	• Nested loop
$O(2^n)$	Exponential (more generally k^n)	• Recursive Fibonacci • Tree recursion

Time Complexity (3 of 4)

```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

Assignment → O(1)

Comparison in condition → O(1)

Arithmetic and assignment → O(2)

Arithmetic and assignment → O(2)

Return evaluated expression → O(1)

Loop runs ~$\log_2 n$ times

Total runtime = $O(1) + \log_2 n * (O(1) + O(2) + O(2)) + O(1)$

= $O(5 * \log_2 n) + O(2)$

= $O(5 * \log_2 n)$ *drop lower order terms*

= **O(log n)** *drop constant factors*

Time Complexity (4 of 4)

- If you're curious, this archived wiki page contains the time complexity of common Python data structures
- Fun fact: A bottleneck in the transformer architecture used by LLMs is the fact that matrix multiplication is a polynomial algorithm
  - Efficiency has real-world implications for software usage, adoption, and environmental impact!
  - Take CS 189, CS 182, or DATA 188 to learn more
- If you take CS 61B or CS 170, you will do more formal and advanced time complexity analysis

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Algorithm #2: What is the runtime of this function with respect to n?

Space Complexity

- You can do a similar analysis for the **space complexity** of an algorithm (e.g. how much computer memory it uses)
- **Ex: Recursive and memoized Fibonacci have a higher space complexity than iterative Fibonacci** (e.g. recursive functions require a bunch of frames to be created, with local variables in each of those frames)
- Often, there is a tradeoff between time and space complexity
- Often in technical interviews, you will not only be expected to come up with an algorithm in <1 hour, but also analyze its time and space complexity
- Space complexity is not emphasized in this course

5 min break

Practice

(reminder to self to use high contrast theme)

30 min

- Search
- Is Prime
- Bar

Search

- Determine the runtime of `search_list1` and `search_list2` with respect to `n`, the length of the `nums` list
- Download starter code `17.py` from the course website under Lecture 17
- 5 min: Try analyzing it yourself
- 5 min: Discuss with someone next to you and compare your analyses

Is Prime

- Determine the runtime of `is_prime1` and `is_prime2` with respect to `n`
- Download starter code `17.py` from the course website under Lecture 17
- 5 min: Try analyzing it yourself
- 5 min: Discuss with someone next to you and compare your analyses

Bar

- Determine the runtime of `bar` with respect to `n`
- Download starter code `17.py` from the course website under Lecture 17
- 5 min: Try analyzing it yourself
- 5 min: Discuss with someone next to you and compare your analyses