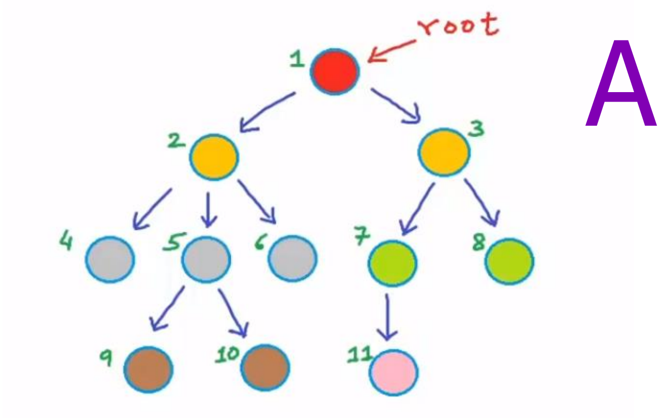
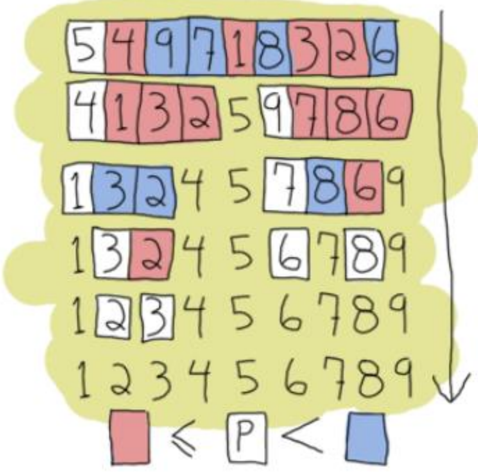
ECE 250 Data Structures & Algorithms



Algorithm Analysis



Ziqiang Patrick Huang

Electrical and Computer Engineering

University of Waterloo

WATERLOO | ENGINEERING

Admin

- Lab0 released.
  - Start early!
  - Remember: devise algorithms before translate them into code
  - Aim to at least have a solid plan (step 1-4) before coming to the lab session next week and ready to code (step 5)

- Reminders
  - Login your Gitlab account
  - Enroll Piazza
  - Do the exercise at the end of L2

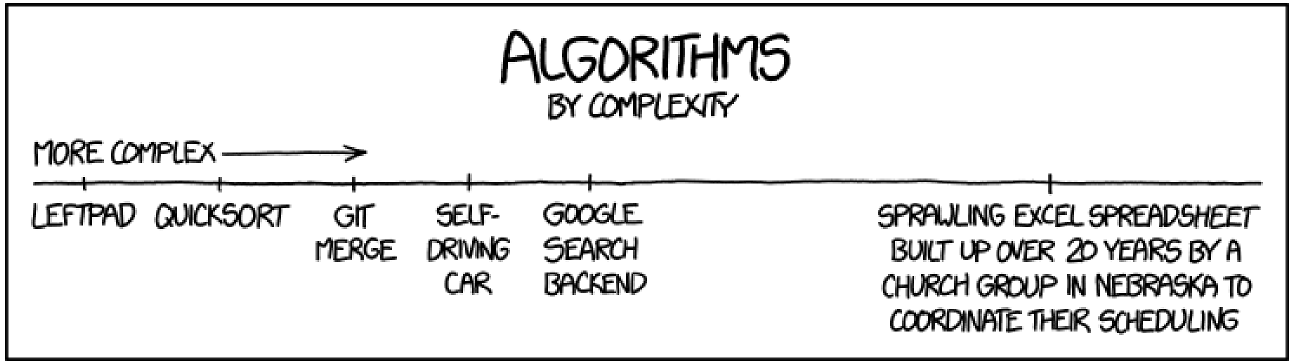
- Aside: if you need access to GPUs (for academic work!), contact Mike Cooper-Stachowsky (mstachow@uwaterloo.ca)

Motivation: How to Measure Efficiency?

- How fast is a piece of code?
  - It depends …
    - What are the inputs?
    - What compiler optimizations are done?
      - Different compilers produce different assembly
      - Most have different optimization settings: -O0, … -O3
    - What hardware is it running on?
- Given two algorithms, how to tell which is better(faster)?
  - Implement and run both?
    - Somewhat accurate but not very practical
  - What about something we could keep in our head as we develop?
    - Algorithm analysis

What is Algorithm Analysis?

- Analyzing an algorithm to understand how long it will run and/or how much memory it will take
  - Also known as complexity analysis: time complexity and space complexity
  - Other “complexities” to keep in mind:
    - Programmability: how easy to implement?
    - Scalability: how easy to scale? (often related to hardware, e.g., multicores)
    - Security: how difficult to be hacked?



Algorithm Analysis: What do We Want?

- Measure both time and space complexity
- Platform/hardware independent
- High level description of algorithm
- Determine how efficient the algorithm is in terms of input size
  - Result can be expressed as a function of input size

High Level Description

- Primitive Operations (algorithmic steps)
  - Assign Value
  - Arithmetic operation
  - Comparing two entities
  - Function call and returns (not the function body)
  - Access an element or follow a pointer
- Primitive operations execute in constant time
  - Treating all primitive operations the same
- Count the operations

Measuring Complexity as a Function

- $f(n)$ represents the function of primitive operations on an input of n

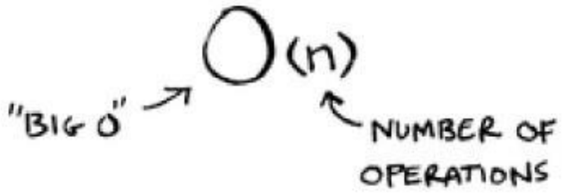
- Three cases worth considering:
  - Worst-Case: the algorithm running with the worst set of data, worst performance
  - Best-Case: the algorithm running with the best designed set of data, fastest performance
  - Average-Case: somewhere between

- For algorithm analysis: we want the most accurate worst-case analysis
  - Why?
  - "Never forget the six-foot-tall man who drowned crossing the river that was five feet deep on average"
  - Also applies to many other things in life (e.g., investing)

Big-O Notation

- We need to formalize our notion of efficiency of an algorithm

- Big-O Notation
  - Denoted $O(f(n))$ where n is the size of our input(s)
  - Typically represents an upper bound, but for this class we want the most accurate upper bound
    - E.g., most of things in this course is $O(n^4)$, but this is not a very helpful description of performance
  - Other Measures: $o(n), \Omega(n), \Theta(n)$, etc



Big-O: Asymptotic Behavior

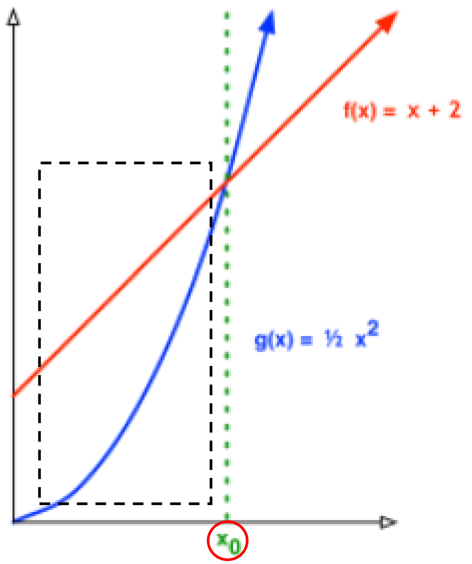
- Observation: small things generally don't matter
  - Computers: pretty fast
  - Difference between 20 on 50 instructions: you won't notice
- Instead, think about behavior on large inputs
  - How does the execution time scale as input size grows?
  - Use Big-O to approximate with some conventions/simplifications
    - Ignore constant factors, drop lower order terms, etc.

Big-O: Formal Definition

- Big-O notation is a mathematical formalism

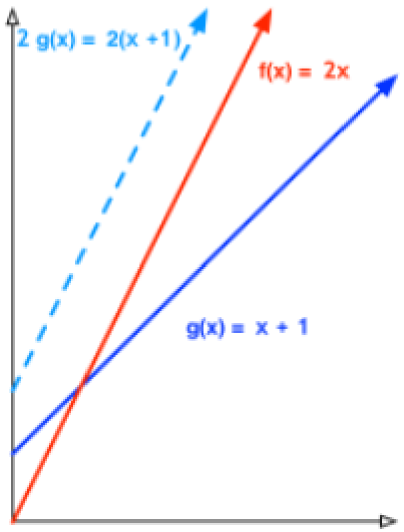
$$f(x) \text{ is } O(g(x)) \iff \exists x_0, c.\forall x > x_0.\ f(x) \leq cg(x)$$

f(x) > g(x)
for small x



f(x) ≤ cg(x)
for x > x_0
(here c = 1)

f(x) is O(g(x)), but g(x) is not O(f(x))



cannot pick an x_0 such that
For x > x_0, 2x ≤ x + 1

c = 2, and x_0 = 0,
For x > 0, 2x ≤ 2(x + 1)

f(x) is O(g(x)), and g(x) is O(f(x))

Determine Big-O Relationship

- How to determine Big-O between two functions?
  - Taking $\lim_{n\to\infty}\left|\frac{f(x)}{g(x)}\right|$

$\lim_{x\to\infty}\left\|\frac{f(x)}{g(x)}\right\|$	$f(x)$ is $O(g(x))$?	$g(x)$ is $O(f(x))$
0	Yes	No
non-zero, finite	Yes	Yes
∞	No	Yes
undefined	Unknown	

Big-O: Not so much Formalism Here

- Advanced algorithm classes (e.g., ECE 406): prove things
  - Use formal definition, prove functions O(other functions)
  - Prove (formally) the code executes in O(something) time
  - Clearly differentiate between O, o, Ω, Θ

- We don't do these things
  - But we will talk about Big-O to understand the efficiency of our algorithms

Common Complexities

- O(1): constant time
  - Runtime doesn't vary with problem size
  - May require setting up the data structures
  - Example: find k^{th}smallest element in a sorted array

- O(log(N)): logarithmic time
  - Runtime scales logarithmically with problem size
  - Such algorithms typically involves splitting input in half at each step
  - Example: binary search a word in the dictionary
  - Note: The base doesn't matter due to change of base (m constant)
    - $\log_m(n) = \frac{\log_2(n)}{\log_2(m)} = \log_m(2)\log_2(n) = C\log_2(n) \rightarrow O(\log(n))$

Common Complexities Continued

- O(N): linear time
  - Double problem size, double runtime
  - Example: find the maximum element of an unsorted array
  - Difference between O(N) and log(N) can be huge, e.g., log(1billion) ~= 30

- O(N^2): quadratic time
  - Double problem size, quadruple runtime
  - Usually involve examine all parings of the input data
  - Example: many sorting algorithms, e.g., bubble sort
  - General: O(N^c) → polynomial time → tractable

Complexities You Do Not Want

- O(2^N): exponential time
  - Increase problem size by 1, double runtime
  - N = 60, 2^{60} = 1 quintillion (same as O(N^2), N = 1 billion)
  - NP-complete problems
    - Best known algorithm → Exponential time
    - Solve one in polynomial time → Solve all of them (million-dollar question)
    - E.g., traveling salesman, graph coloring

- O(N!): Factorial time
  - $20! \approx 2^{61}, 22! \approx 2^{70}, 24! \approx 2^{80}, 26! \approx 2^{88}, 28! \approx 2^{98}, 30! \approx 2^{108}$
  - Example: list all permutations of an array

Big-O Conventions

- Drop Constant Factors:

$O(4n) \rightarrow O(n)$

$O(0.5n^2) \rightarrow O(n^2)$

- Drop Lower Order Terms:

$O(n^2 + 2000n - 5) \rightarrow O(n^2)$

$O\left(3n + 2\log(n) + nlog(n)\right) \rightarrow O(nlog(n))$

Big-O example: Count Duplicates

```
int countDuplicates (int * array, int n) {
    int dupCount = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (array[i] == array[j]) {
                dupCount++;
            }
        }
    }
    return dupCount;
}
```

O(1)

This takes O(N) time.
Maybe N-1, or N-5, etc..
But we call those O(N)

We repeat this O(N) work O(N) times.
O(N) * O(N) is O(N^2)

WATERLOO | ENGINEERING

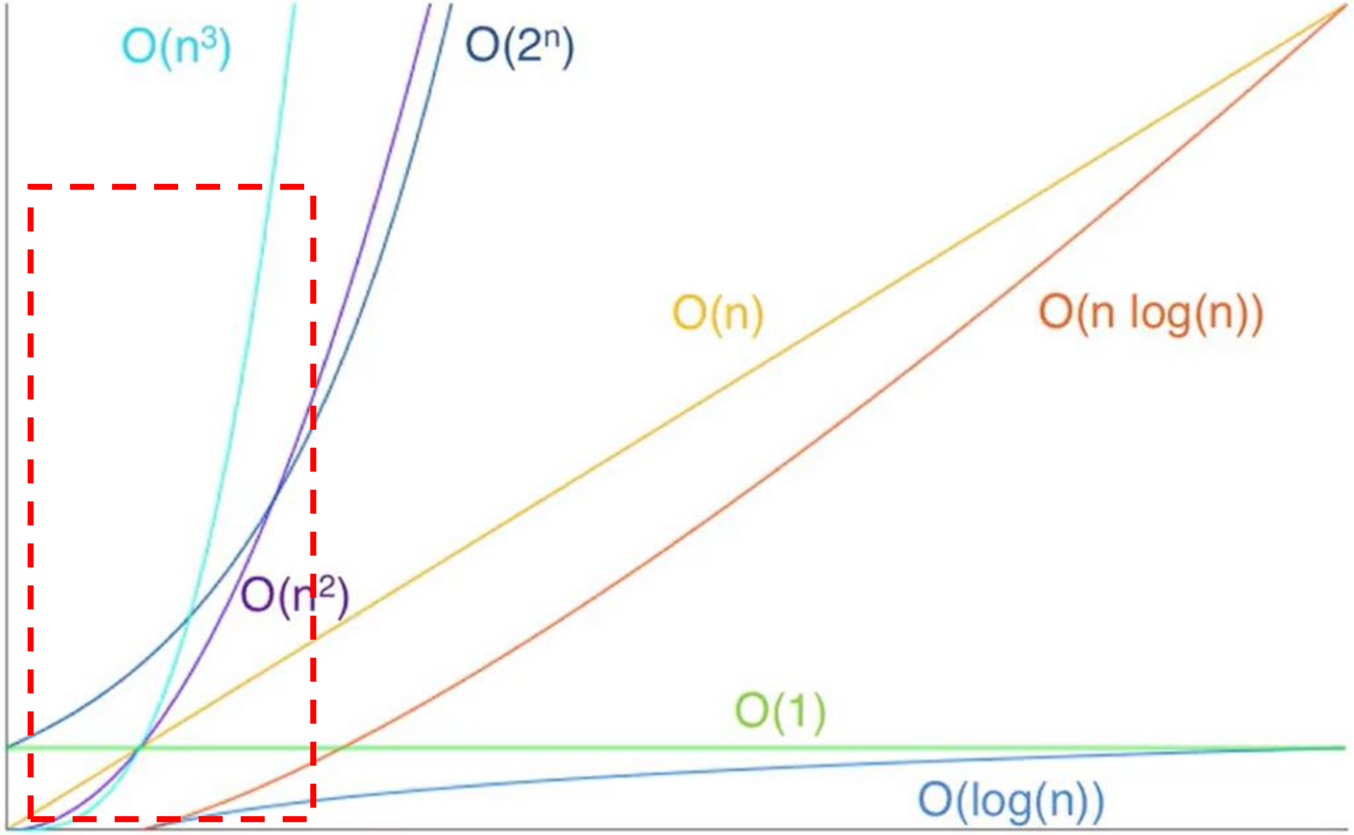
Big-O Exercise

What is the time complexity of the following function using Big-O?

```
int someFuntion (int * array, int n) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            for (int k = j+1, k < 100; k++) {
                if (array[i] + array[j] == array[k])
                    count++;
            }
        }
    }
    return count;
}
```

Common Complexities (Closer Look)

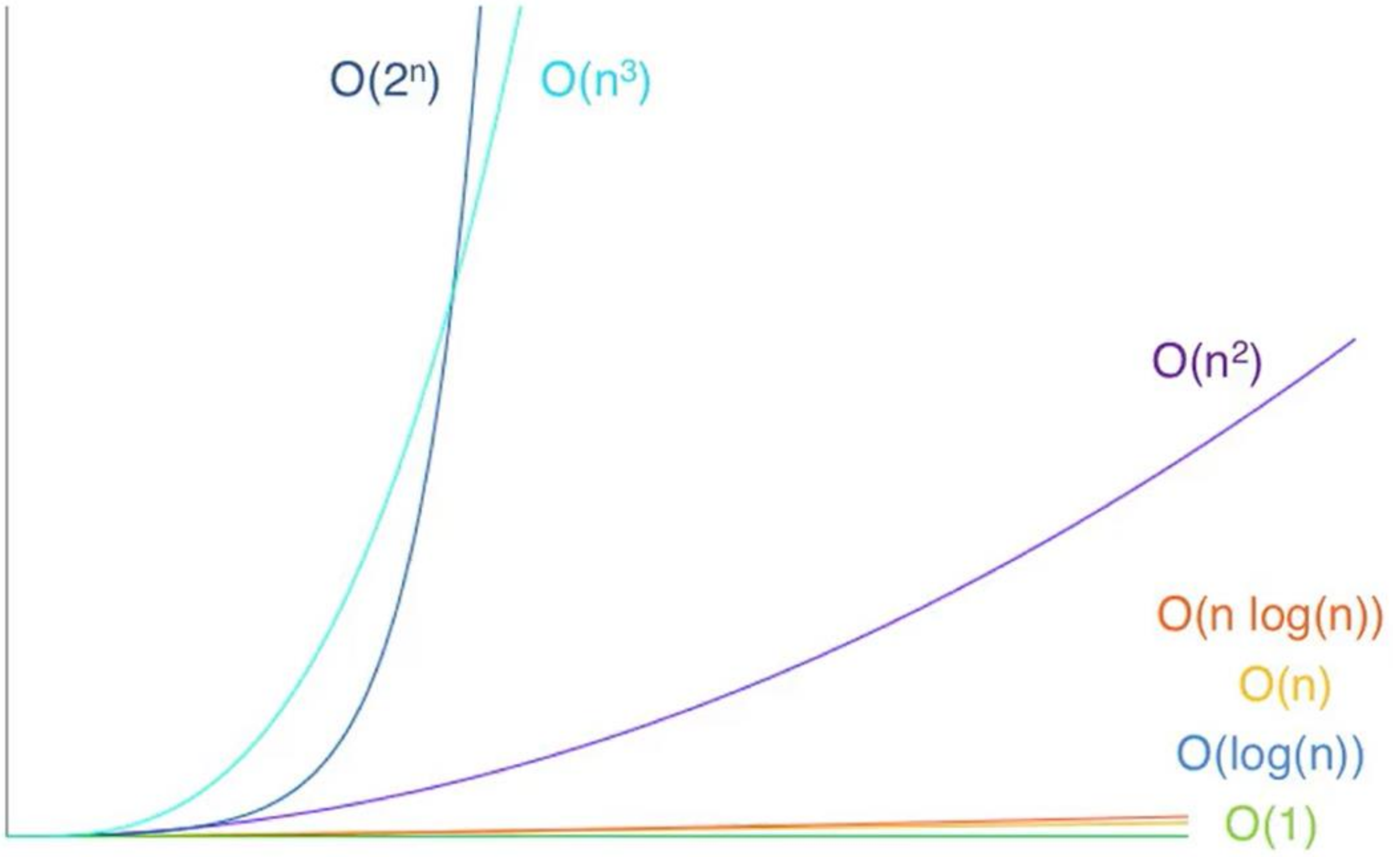
Complexity Name	Big-O Notation
Constant	$O(1)$
Logarithmic	$O(\log(n))$
Linear	$O(n)$
Log-Linear	$O(n\log(n))$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(a^n)$, a constant



For small n, hard to tell

Example: 100 Operations with 1000 Inputs

Algorithm Complexity	N = 1000
O(1)	constant
O(log(n))	0.09966
O(n)	10
O(nlog(n))	99.66
$O(n^2)$	10,000
$O(n^3)$	100,000,000
$O(2^n)$	1.07×10^{299}



Big Picture: High-Performance Programming

- Efficient Algorithm
  - Including choosing the proper data structures
  - This is what this course is about

- High-performance implementation
  - Choose proper programming languages (e.g., Python vs C++)
  - Understand the underlying hardware (e.g., Multicores, GPUs)
  - Compiler knowledge also helpful (e.g., understanding various optimizations)
  - Profile your code → Find where your code is spending time
  - Interested? Consider taking: ECE 320, ECE 351, ECE 459

Exercise For You: Shuffle the Deck

- Considering a deck of cards stored in an array, how would you devise an algorithm to shuffle a deck?

- What is the time complexity of the algorithm your algorithm?

- How could you use lots of space, but very little time, to quickly get a shuffled deck when you need one? (Don’t worry about how practical your solution would be; this is a “thought exercise”)

Wrap Up

- In this lecture we talked about:
  - Why do we need algorithm analysis?
  - What is Big-O notation?
  - How to determine Big-O relationships?
  - What are some of the common complexities?
  - What are some of the algorithms that exhibit those complexities?

- Next Up:
  - Abstract Data Types (ADTs)

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 2
- Introduction to Algorithms: Chapter 3.1

Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)
  - Mary Hudachek-Buswell (Gatech)