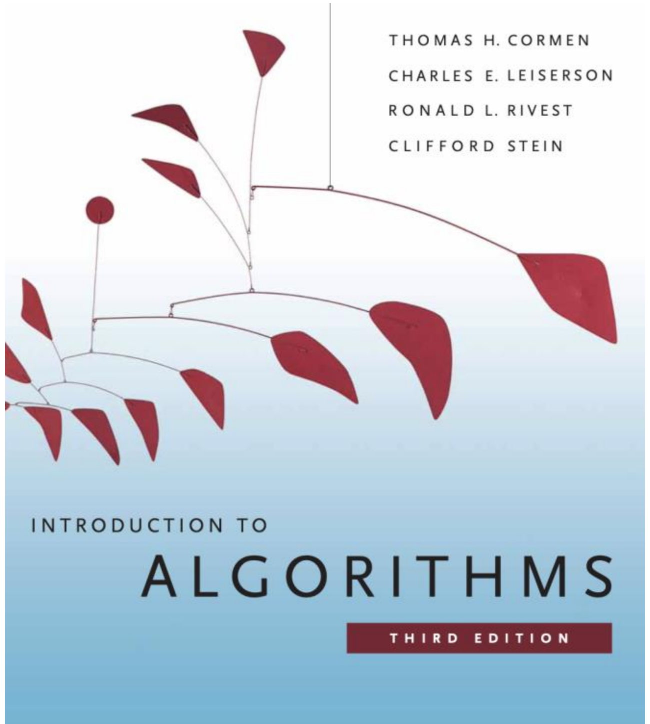
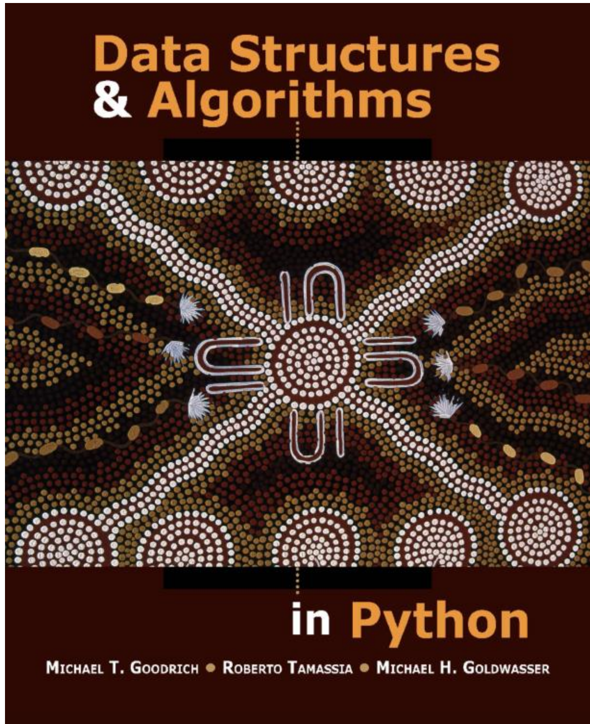
ALGORITHM ANALYSIS

Will

PREVIOUSLY ON DS&A

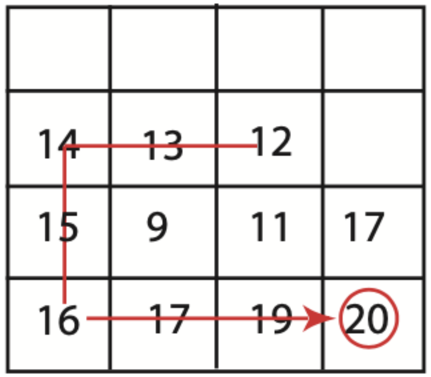
- Discussed what this course was about
- Review on Python
- Brief discussion on peak finding
    - Algorithm on sequnces

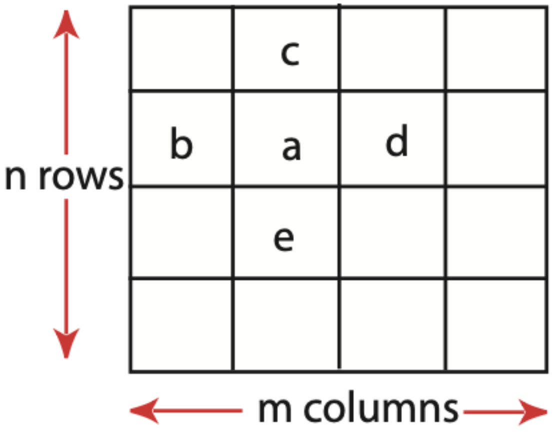




PEAK FINDING

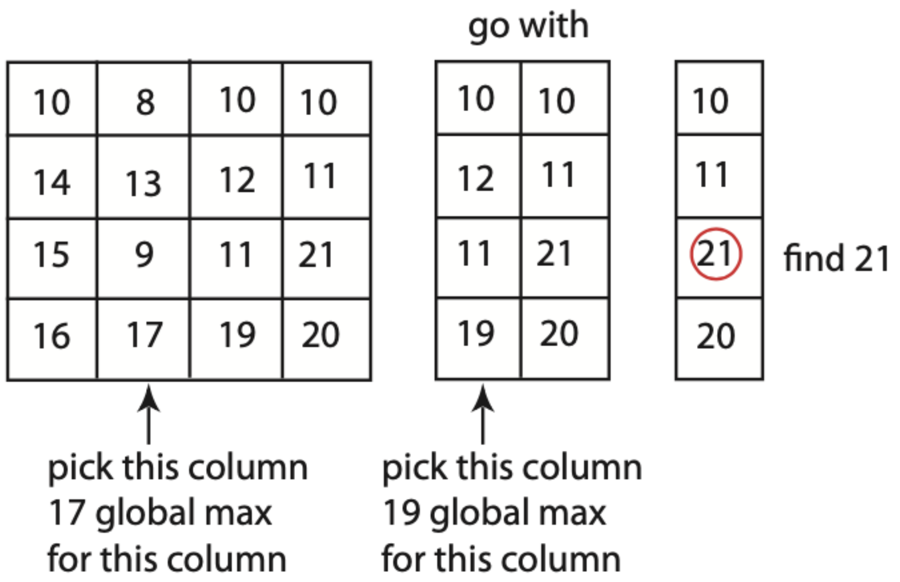
- Two-dimensional version
  - A is a 2D peak iff a>=b, a>=d, a>=c and a>=e
  - 20 is a peak
  - Greedy ascent algorithm
    - Going to the direction where the element is bigger
    - Complexity: $O(n^2)$ if m = n





PEAK FINDING

- Attempt #2
  - Pick middle column j = m/2
  - Find global maximum on column j at (i, j)
  - Compare (i, j-1), (i, j), (i, j+1)
  - Pick left columns if (i, j-1) > (i, j)
  - Similarly for the right
  - (i, j) is a 2D peak if neither condition holds -> why?
  - Solve the new problem with half the number of columns
  - When you have a single column
    - find global maximum and you are done



PEAK FINDING

- Attempt #2
  - Complexity?
  - If T(n, m) denotes work required to solve problems

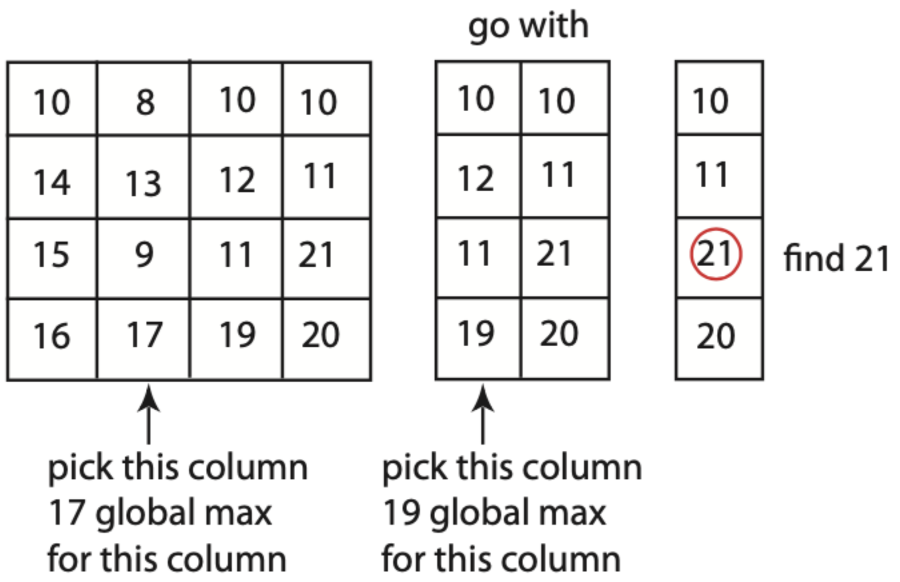
  With n rows and m columns

  T(n,m) = T(n, m/2) + O(n) – global maximum on m

  T(n,m) = O(n) + O(n) … log m times

  = O(n log m)

  = O(n log n) if m = n



THIS LECTURE

- Object-Oriented Design
  - Goals
  - Principles
  - Patterns
- Asymptotic Analysis

OBJECT-ORIENTED DESIGN

- Why object-oriented?
- The complexity of software systems grow
- Maintaining code becomes difficult
- Solution?
- Raise the level of abstraction
- In design and implementation
- Assembly -> procedure-oriented -> object-oriented
- And of course model-oriented (or sometimes model-driven)

OBJECT-ORIENTED DESIGN

- Goals for software
  - Robustness: the ability to handle unexpected inputs
    - Recover gracefully from errors
    - Life-critical applications
  - Adaptability: the ability to evolve over time to changing conditions
    - How easy is it to make changes to your software?
    - Can your software be ported to another platform?
      - i.e. using a different programming language?
  - Reusabiility: the ability to reuse some parts of your software
    - How modular is your software?
    - Good software is often re-used
      - Good: efficient, long MTTE, safe, robust

OBJECT-ORIENTED DESIGN

- Object-Oriented Principles
  - Modularity: how organized are the components of your software
    - “modules” in Python: closely related classes (with their functions)
    - Robustness: it is easier to test and debug separate components
    - Adaptability: isolation of concerns
    - Reusability: modules can be reused when related need arises in other contexts
  - Abstraction: most fundamental concepts of a complicated system
    - Abstract Data Types (ADT): mathematical model of a data structure
      - **What** each operations do, not **how** they do it
      - Public **interface**
  - Encapsulation: internal details of implementations not revealed
    - Only public interfaces
    - Robustness and adaptability: easy to test, easy to change

DESIGN PATTERNS

- Algorithm
  - Recursion
  - Amortisation
  - Divide-and-conquer
  - Prune and search (decrease-and-conquer)
  - Brute force
  - Dynamic programming
  - Greedy methods
- Software engineering
  - Iterator
  - Adapter
  - Position
  - Composition
  - Template methods
  - Factory

MEASURING THE QUALITY OF ALGORITHMS

- Data structure: systematic way of organising and accessing data
- Algorithm: Al-Khwarzmi /al-kha-raz-mi/
  - ”father of algebra” with his book “The compendious Book on Calculation by Completion & Balancing”
  - Linear & quadratic equation solving: some of the first algorithms
- This course: design of “good” data structures and algorithms
- What’s “good”? how do we measure?

EXPERIMENTAL STUDIES

- Observe running time of an algorithm
  - Executing on various test inputs
  - Record the time spent during each execution
  - Doing it in Python
  - Problems?

```
from time import time
start_time = time( )                   # record the starting time
run algorithm
end_time = time( )                     # record the ending time
elapsed = end_time − start_time        # compute the elapsed time
```

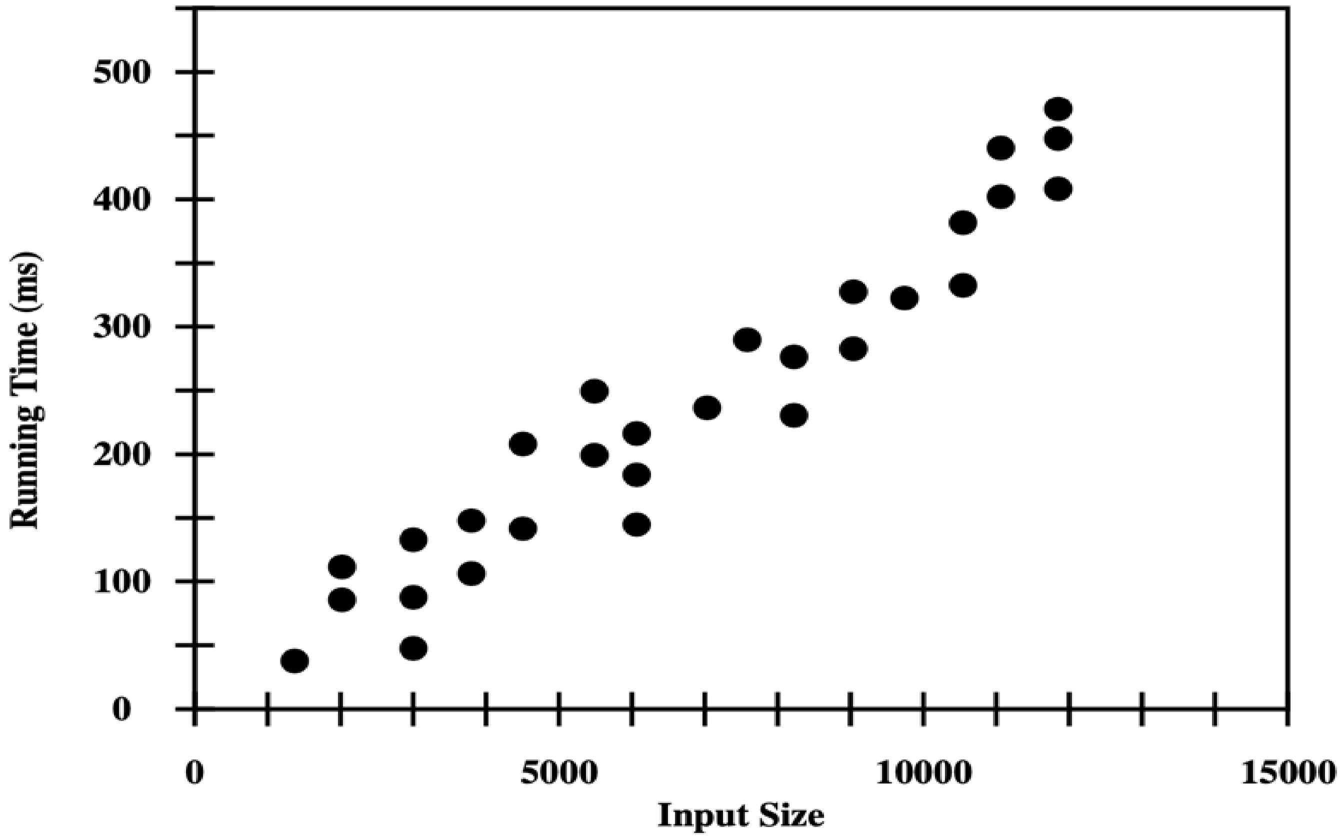
EXPERIMENTAL STUDIES

- Observe running time of an algorithm
  - Doing it in Python
  - Problems?
    - Input may not be the worst case
    - Algorithm may not scale
    - Time() may be affected by other processes sharing the CPU
    - What about clock()?

```
from time import time
start_time = time( )                        # record the starting time
run algorithm
end_time = time( )                          # record the ending time
elapsed = end_time − start_time             # compute the elapsed time
```

EXPERIMENTAL STUDIES

- Observe running time of an algorithm

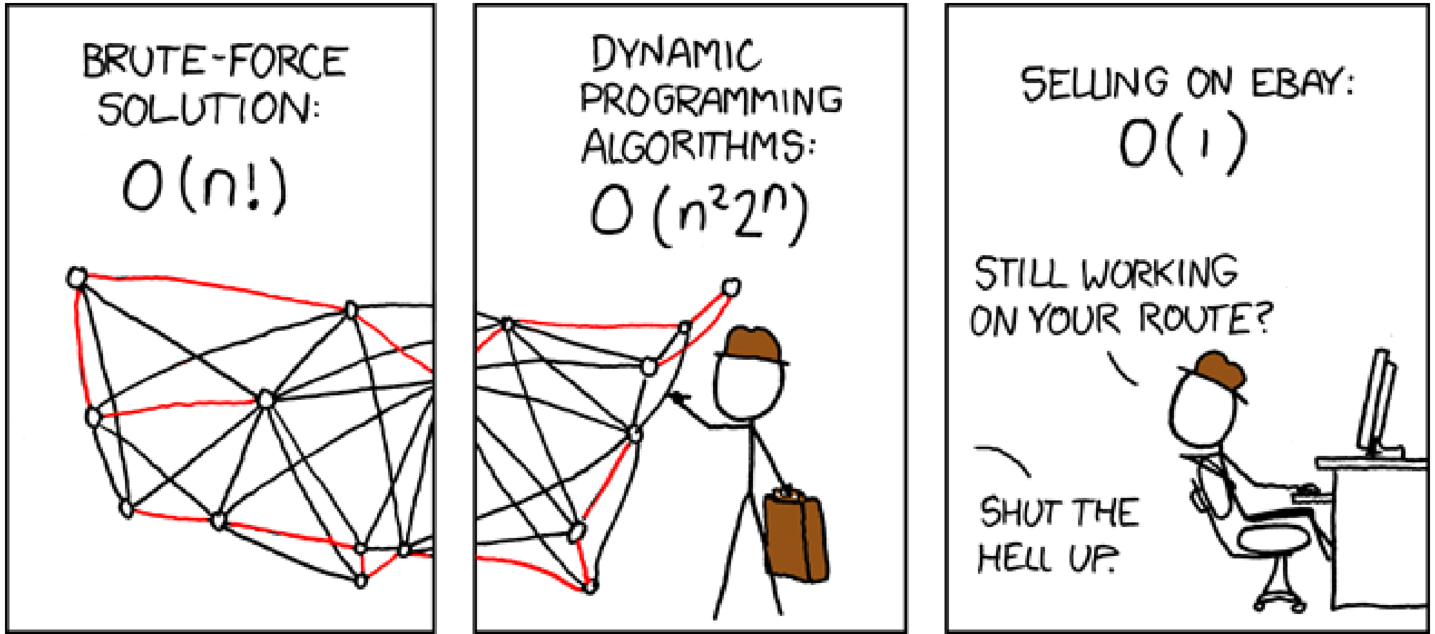


CHALLENGES OF EXPERIMENTAL STUDIES

- Directly comparing two algorithms are difficult
  - Same hardware and software
  - Same CPU activities
- Only on a limited set of test inputs
  - Is this input the worst case?
    - Easy to determine for an input of size 8
    - What about the input of size 1,000,000?
  - Does the algorithm scale when the input is very large?
    - E.g. 1,000,000,000?
- An algorithm must be fully implemented in order to execute
  - Time consuming
    - What if our design in inefficient?
  - C vs. Python?

MOVING BEYOND EXPERIMENTAL ANALYSIS

- Approach to analyse the efficiency of algorithms
  - Allows us to evaluate the relative efficiency of any two algorithms
  - Is performed by studying a high-level description of the algorithm without implementing it
  - Takes into account all possible inputs

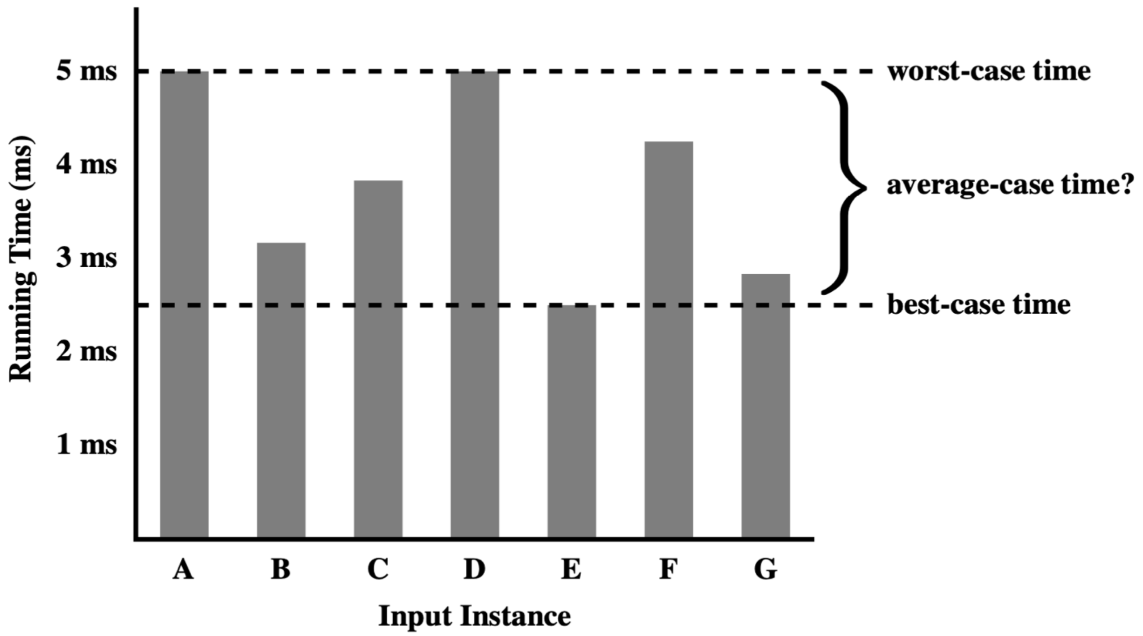


COUNTING PRIMITIVE OPERATIONS

- Often correspond to low-level instructions with an execution time that is constant
  - Assignment
  - Determine the object with an identifier
  - Arichmetic operation: +,-,*,/, etc.
  - Comparing two numbers
  - Accessing an element of a python list by index
  - Calling a function
    - Excluding what's inside the function
  - Returning from a function

FOCUSING ON THE WORST CASE INPUT

- An algorithm runs in different time for different input
- We are interested in the worst case
  - Why?
    - Some applications are time sensitive
    - Upper bound
    - Easier than analysing average



SEVEN FUNCTIONS WE NEED FOR ALGORITHM ANALYSIS

- Constant functions
  - $f(n) = c$
- Logarithm functions
  - $f(n) = \log_b n$ for some constant b > 1
  - x = $\log_b n$ if and only if $b^x = n$
  - most common base for the logarithm function: 2
- Linear functions
  - $f(n) = n$
- The N-log-N functions
  - $f(n) = \mathrm{n} \log n$

SEVEN FUNCTIONS WE NEED FOR ALGORITHM ANALYSIS

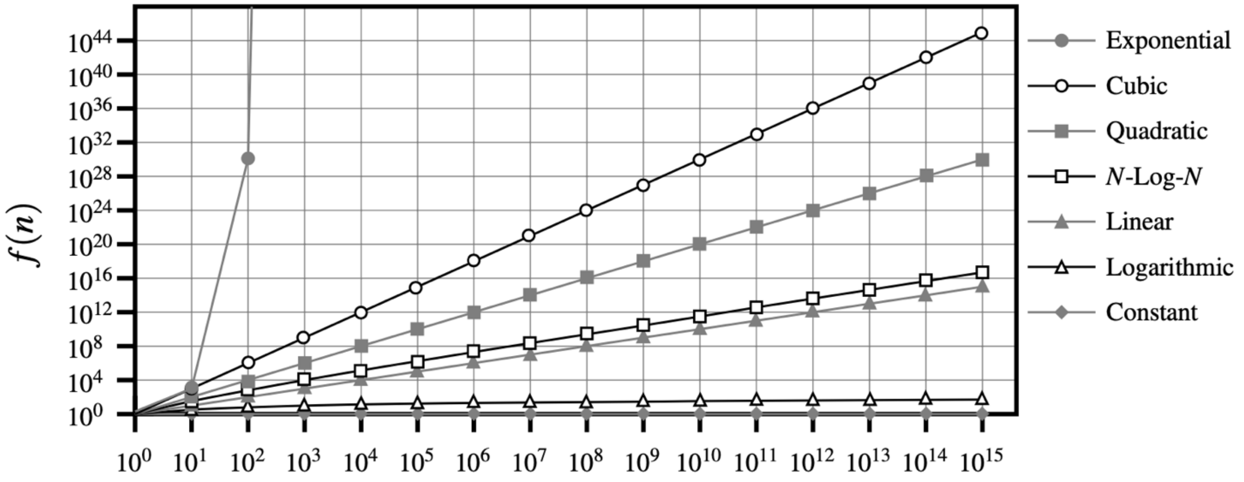
- Quadratic functions
  - $f(n) = n^2$
  - Typically happens for nested loops
- Cubic functions
  - $f(n) = n^3$
  - Typically happens for nested loops
- Exponential functions
  - $f(n) = b^n$ for some constant b > 0
  - Geometric summation

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1$$

GROWTH RATES OF FUNCTIONS

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



ASYMPTOTIC ANALYSIS

- Growth rate of the running time as a function of the input size n
  - Grows proportionally to n
- Mathematical notation that disregard constant factors
  - Size of input n -> values correspond to the main factor that determines the growth rate in terms of n
- Finding the largest element of a Python list
  - Complexity: c*n for some constant c (c=?)

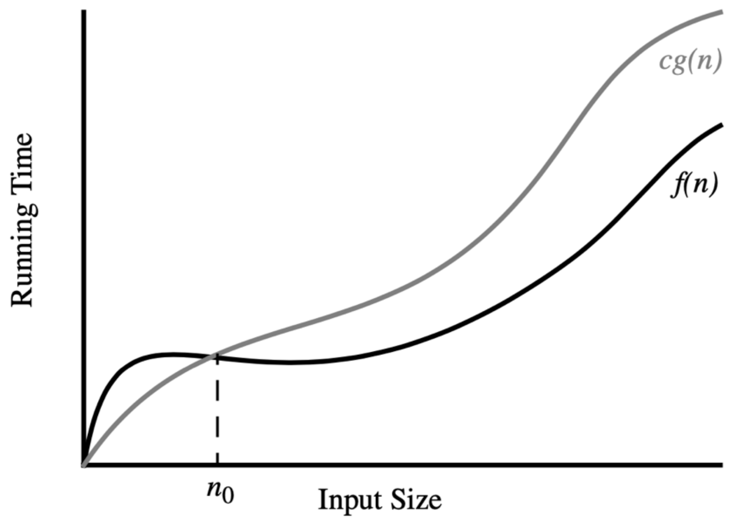
```
def find_max(data):
  """Return the maximum element from a nonempty Python list."""
  biggest = data[0]                 # The initial value to beat
  for val in data:                  # For each value:
    if val > biggest                # if it is greater than the best so far,
      biggest = val                 # we have found a new best (so far)
  return biggest                    # When loop ends, biggest is the max
```

THE "BIG-O" NOTATION

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant c>0 and an integer constant n_0 >= 1 such that

$$f(n) \le c\, g(n), \text{ for } n \ge n_0$$

- y=x vs. y=x^2



THE "BIG-O" NOTATION

- Function: 8n + 5 is $O(n)$
- Find a constant c > 0 and an integer constant $n_0 \geq 1$
  - $8n+5 \leq cn$ for every $n \geq n_0$
  - c = 9 and $n_0 = 5$
- Big-O: f(x) is "less than or equal to" another function g(n) up to a constant factor and in the asymptotic sense as n grows towards infinity
- $f(n) = O(g(n))$
- "$f(n)$ is $O(g(n))$ "

THE "BIG-O" NOTATION

- The algorithm find_max() for computing the maximum element of a list of n numbers, runs in $O(n)$ time

```
def find_max(data):
    """Return the maximum element from a nonempty Python list."""
    biggest = data[0]                  # The initial value to beat
    for val in data:                   # For each value:
        if val > biggest               # if it is greater than the best so far,
            biggest = val              # we have found a new best (so far)
    return biggest                     # When loop ends, biggest is the max
```

THE "BIG-O" NOTATION

- We can ignore constant factors and lower-order terms when talking about asymptotic complexity
- $5n^4 + 3n^3 + 2n^2 + 4n^1$ is $O(n^4)$
  - $5n^4 + 3n^3 + 2n^2 + 4n^1 \leq (5 + 3 + 2 + 4)\mathrm{n}^4 = \mathrm{cn}^4$
- If $f(n)$ is a polynomial degree d
  - $f(n) = a0 + a_1 n + \cdots + a_n n^d$
  - $a_d > 0$, then $f(n)$ is $O(n^d)$
- Characterising functions in simplest terms
  - $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or $O(n^4)$
  - It is more accurate to say $f\ (\mathrm{n}) = O(n^3)$

BIG-OMEGA AND BIG-THETA

- Big-Omega
  - Big Omega is "greater than or equal to"
    - Normally referred to as the "upper bound"
  - $f(n)$ is $\Omega(g(n))$ if there is a real constant c>0 and an integer constant n_0 >= 1

$$f(n) \geq c\, g(n), \text{ for } n \geq n_0$$

- Big-Theta
  - Two function grow at the same rate
  - $f(n)$ is $\theta(g(n))$, if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
  - If there are real constants c'>0 and c''>0, and an integer constant n_0 >= 1

$$c'\, g(n) \leq f(n) \leq c''\, g(n), \text{ for } n \geq n_0$$

COMPARATIVE ANALYSIS

- Two algorithms
  - A: $O(n)$
  - B: $O(n^2)$
- Which one is better?
- Does it matter if A is $10000n$ and B is n^2?

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Running Time (μs)	Maximum Problem Size (n) 1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

COMPARATIVE ANALYSIS

- But what about $10^{100}n$ vs $n\log n$?
  - We'd prefer $n\log n$
  - Why?
  - 10^{100} is believed to be the upper bound on the number of atoms in the observable universe
- What is a "efficient" algorithm?
  - $O(n\ log\ n)$
  - $O(n^2)$
- Exponential running times?
  - Bad
  - Very bad
  - You're likely to lose your job bad
  - The famous "grain on chess board" problem

EXAMPLES

```
def find_max(data):
  """Return the maximum element from a nonempty Python list."""
  biggest = data[0]              # The initial value to beat
  for val in data:               # For each value:
    if val > biggest             # if it is greater than the best so far,
      biggest = val              # we have found a new best (so far)
  return biggest                 # When loop ends, biggest is the max
```

- List: arr
- Constant time operations on list
  - len(arr): $O(1)$
  - Access an element with index arr[j] : $O(1)$
- find_max() again
  - How many times do we have to update the "biggest" value?
  - Worst case: n-1
  - Random: probability that the j^{th} element is the largest of the first j elements is 1/j.
    - Harmonic number.
  - For n elements, it is $\ln(n) + C$, C is Euler's constant, and therefore $O(\log n)$

PREFIX AVERAGE

- Given a sequence S consisting of n numbers, compute a sequence A
  - A[j] is the average of elements S[0] … S[j] for j = 0, …, n-1

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$$

  - A quadratic time algorithm

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                  # create new list of n zeros
  for j in range(n):
    total = 0                  # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)       # record the average
  return A
```

PREFIX AVERAGE

- A quadratic time algorithm
  - len(S): O(1)
  - A=[0]*n: O(n)
  - For loops
    - Outer loop: O(n)
    - Inner loop: 1, 2, 3, 4, … n times: $n(n+1)/2 = O(n^2)$

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  for j in range(n):
    total = 0                    # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)         # record the average
  return A
```

PREFIX AVERAGE

- A linear time algorithm
  - len(S): O(1)
  - A=[0]*n: O(n)
  - For loop: O(n)

```
def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  total = 0                      # compute prefix sum as S[0] + S[1] + ...
  for j in range(n):
    total += S[j]                # update prefix sum to include S[j]
    A[j] = total / (j+1)         # compute average based on current sum
  return A
```

THREE-WAY SET DISJOINTNESS

- 3 sequences of numbers: A, B and C
  - No individual sequence contains duplicate values
  - The intersection of the three sequences is empty
  - There is no element x such that $x \in A, x \in B, and\ x \in C$
  - Complexity?

```
def disjoint1(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          return False          # we found a common value
  return True                   # if we reach this, sets are disjoint
```

THREE-WAY SET DISJOINTNESS

- 3 sequences of numbers: A, B and C
  - No individual sequence contains duplicate values
  - The intersection of the three sequences is empty
  - There is no element x such that $x \in A, x \in B, and\ x \in C$
  - Improved version: O(n^2)

```
def disjoint2(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      if a == b:              # only check C if we found match from A and B
        for c in C:
          if a == c           # (and thus a == b == c)
            return False      # we found a common value
  return True                 # if we reach this, sets are disjoint
```

ELEMENT UNIQUENESS

- Find if there are duplicate elements in a sequence

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False              # found duplicate pair
  return True                     # if we reach this, elements were unique
```

ELEMENT UNIQUENESS

- Efficiency can be improved if the sequence is sorted

```
def unique2(S):
  """Return True if there are no duplicate elements in sequence S."""
  temp = sorted(S)                    # create a sorted copy of S
  for j in range(1, len(temp)):
    if S[j−1] == S[j]:
      return False                    # found duplicate pair
  return True                         # if we reach this, elements were unique
```

SMALL QUIZ FOR THIS WEEK:

- Problem settings:
  - 2 players
  - 1 table
  - Infinite amount of coins
- Rule:
  - Each turn: one player places a coin on the table
    - Anywhere
  - The player that places the last coin (i.e. table if full of coins) wins
  - Coins cannot overlay with each other
- If you are to win, would you choose to:
  - Take turns first
  - Take turns second
- What's the winning strategy?

THANKS

See you in the next session!