Welcome to Data C88C!

Lecture 19: Efficiency
Monday, July 28th, 2025
Week 6
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Rev 01: 2025-07-29 4:10 PM PST

Announcements

- "Clarification of due dates for Project01, Project02": [link]
  - **Project01 ("Maps")**: due Friday July 25th, 11:59 PM PST
    - Late due date (for 75% credit [link]): Tuesday July 29th, 11:59 PM PST
- Mid-semester survey feedback: [link]
  - If 75% of the class completes this form by Monday July 28th at 11:59 PM, everyone will receive 1 point of extra credit! If this goal is not met, nobody will receive the extra point.
  - As of today (3pm PST): ~50% of the class has completed the survey
- Midterm regrades: due this Friday
- August 1st: Change Grade Option deadline

Lecture Overview

- Efficiency
  - Orders of growth
  - "Big-O" notation
- (For fun) P vs NP

Linked List Practice

Spring 2023 Midterm 2 Question 3(b)

Definition. A *prefix sum* of a sequence of numbers is the sum of the first n elements for some positive length n.

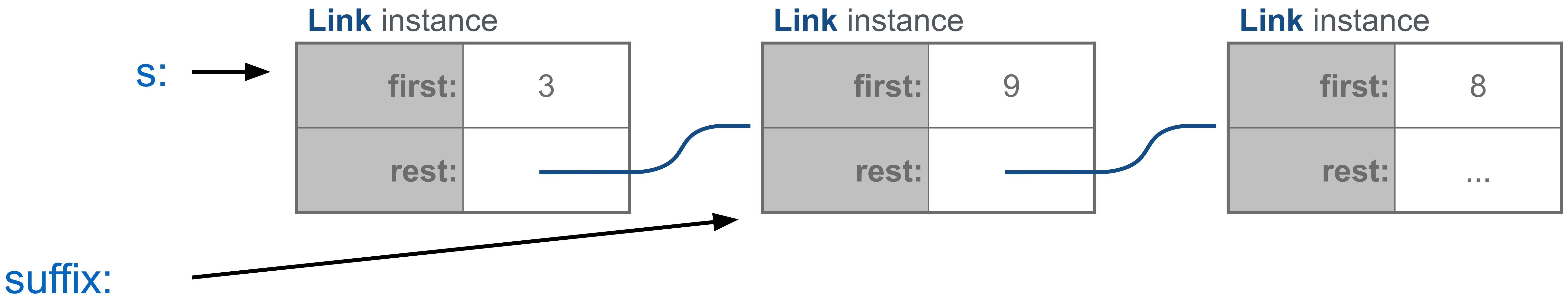
Implement tens, which takes a non-empty linked list of numbers s represented as a Link instance. It prints all of the prefix sums of s that are multiples of 10 in increasing order of the length of the prefix.

```
def tens(s):
    """Print all prefix sums of Link s that are multiples of ten.
    >>> tens(Link(3, Link(9, Link(8, Link(10, Link(0, Link(14, Link(6)))))))
    20
    30
    30
    50
    """
    def f(suffix, total):
        if total % 10 == 0:
            print(total)

        if __suffix is not Link.empty____________________:

            __f(suffix.rest, total + suffix.first)_____________

    _f(s.rest, s.first)___________________________________
```



Tree Class

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

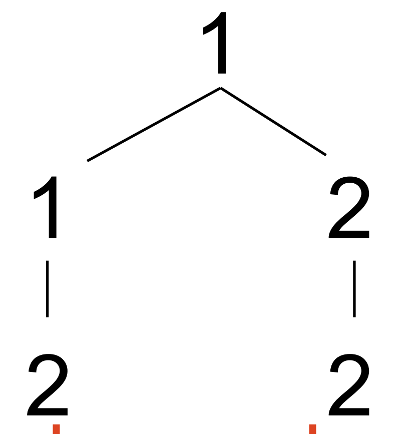
Tree Practice

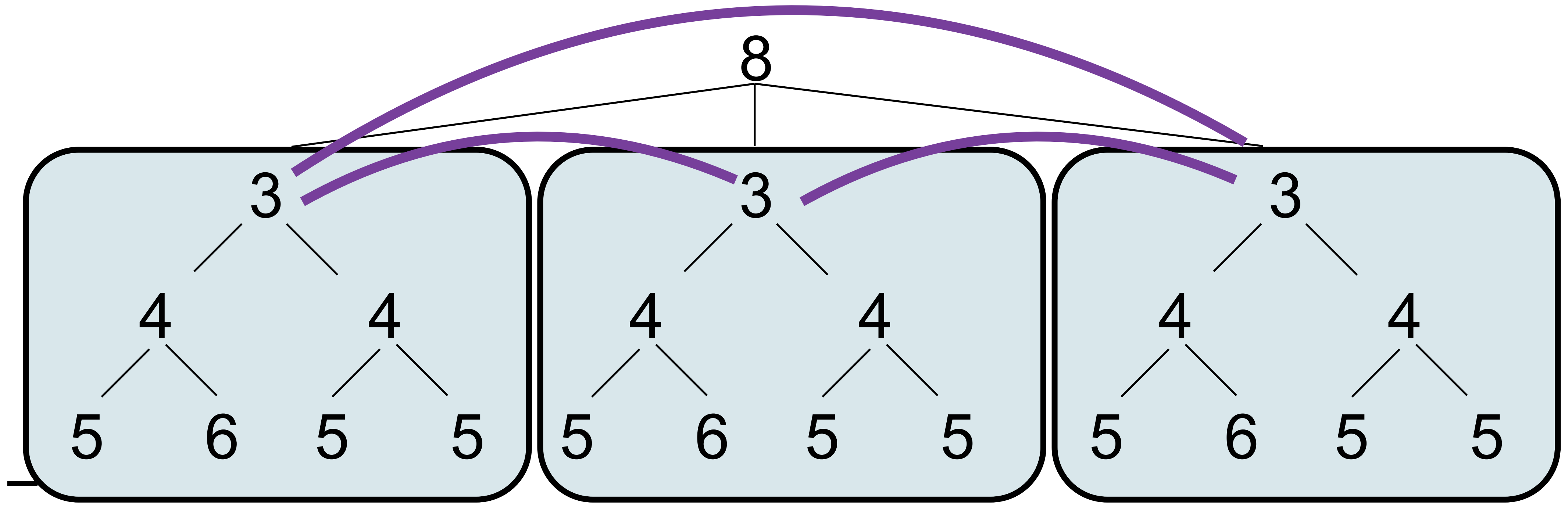
Example: Count Twins

Implement twins, which takes a Tree t. It return the number of pairs of sibling nodes whose labels are equal.

```
def twins(t):
    """Count the pairs of sibling nodes with equal labels.

    >>> t1 = Tree(3, [Tree(4, [Tree(5), Tree(6)]), Tree(4, [Tree(5), Tree(5)])])
    >>> twins(t1)   # 4 and 5
    2
    >>> twins(Tree(1, [Tree(1, [Tree(2)]), Tree(2, [Tree(2)])]))
    0
    >>> twins(Tree(8, [t1, t1, t1]))  # 3 pairs of twins at the top, plus 2 in each branch
    9
    """
    count = 0
    n = len(t.branches)
    for i in range(n-1):
        for j in range(i+1, n):
            if t.branches[i].label == t.branches[j].label
                count += 1
    return count + sum([twins(b) for b in t.branches])
```

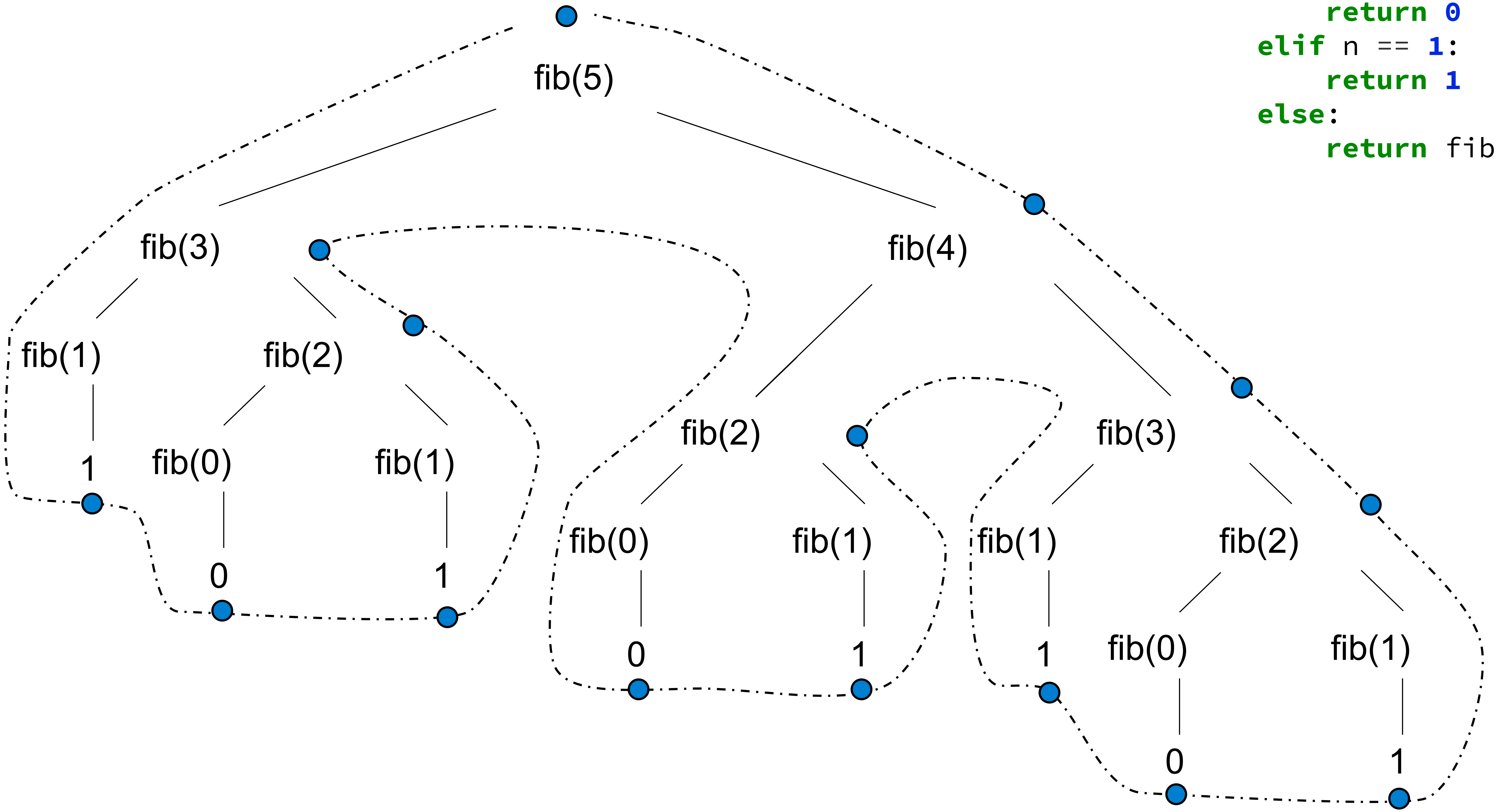




Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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



Memoization

Memoization

Idea: Remember the results that have been computed before

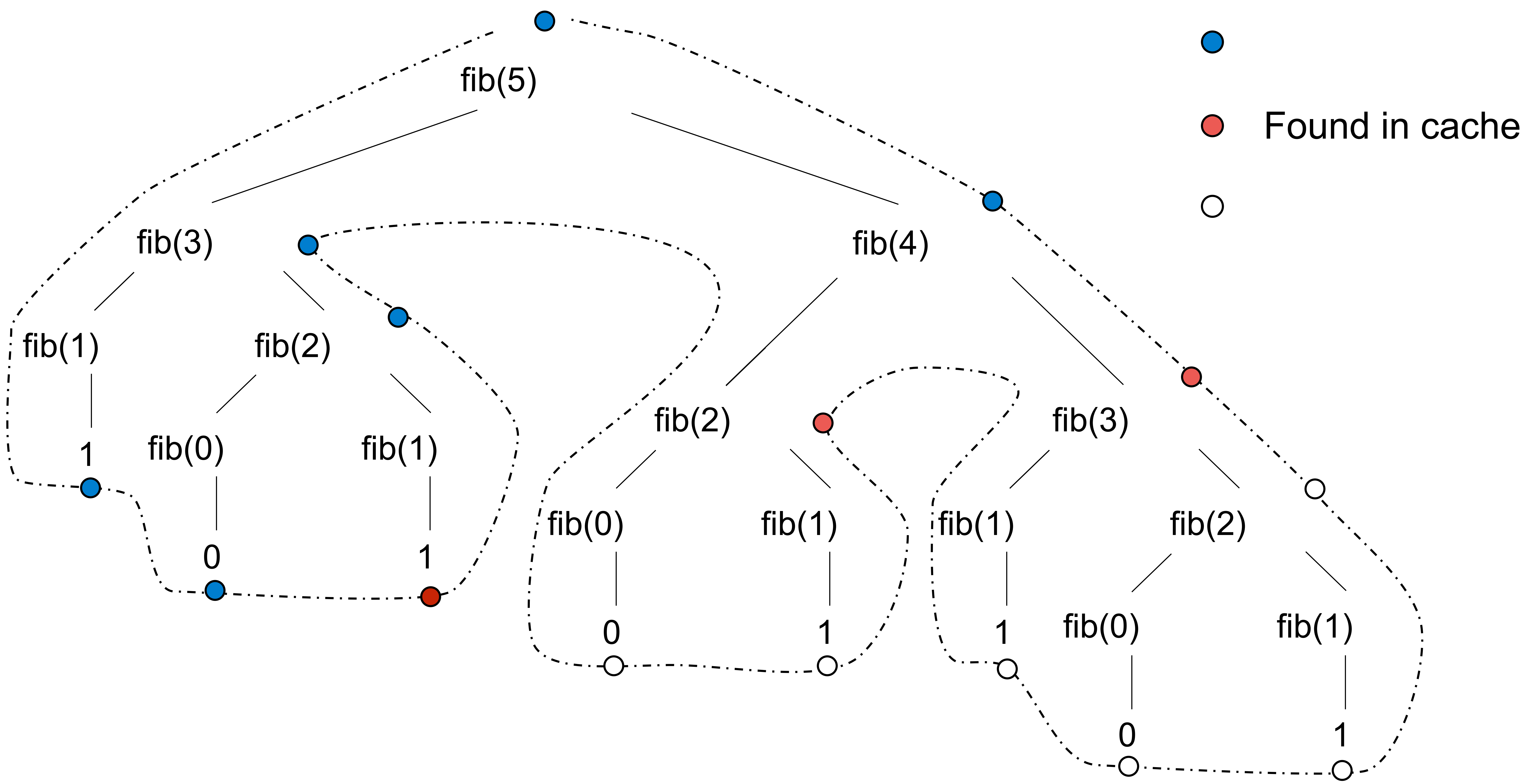
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

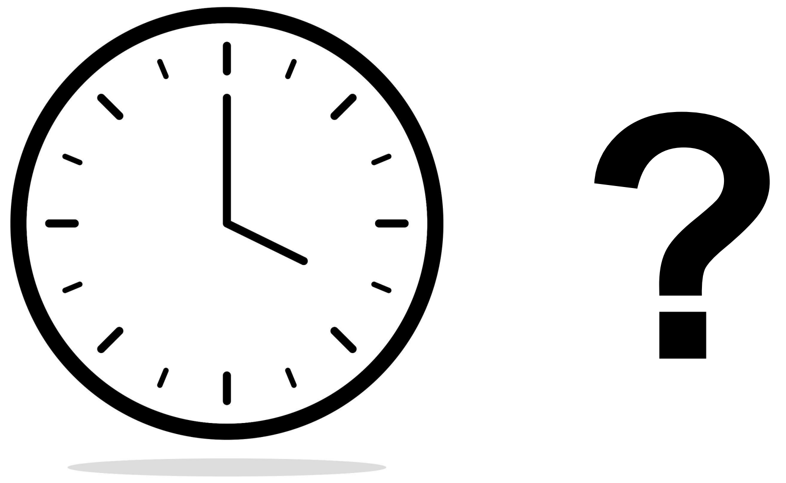
Memoized Tree Recursion



Measuring Efficiency

How to measure efficiency?

- **Idea**: use seconds ("wall clock time") to quantify how "fast" a code/function runs
  - **Downside**: time-based measurements will change based on which machine I run the benchmark on.
- **Idea**: instead, let's pursue a generic, hardware-agnostic way of measuring how "fast (or slow)" a program is: by counting simple operations*
-



https://www.freepik.com/free-vector/clock-basic-style_198044500.htm#fromView=keyword&page=1&position=21&uuid=f2e5c101-bca2-44c3-b7ff-4c055a8d5d0e&query=Clock+Clipart

Python: counting operations

- In Python, the following operations are considered a "single operation":
  - creating a new primitive variable
  - reading/writing to a variable
  - integer/float arithmetic**
  - accessing an attribute
- Functions/methods: the runtime is the total number of operations in the function body
- Common list methods that are considered a "single operation" [link]: creating a new list, appending to a list
- Tip: it's not enough to "count lines" to estimate how much work a function does, as one line can be more expensive than other lines.

** Fun fact: in Python, integers are implemented as "bignum" that allow them to increase in value arbitrarily large (bounded by your computer's available CPU memory), but at the expense of mathematical operations (+, *, etc) taking longer if your integer grows larger. But, for the purposes of this class, let's assume integer operations are a single operation.

Floats (eg 3.14), however, do not have infinite range: they're bounded by the limits as dictated by the IEEE floating point format [link]

Example: counting operations in Python code

Let `lst_nums` be a list of integers with length N

def f1(lst_nums):	**Number of operations**
`x = 0`	1 (create variable x and assign it the value 0)
`x = x + 2`	3 (read x, add 2, write to x)
`tmp_nums = []`	2 (create tmp_nums and assign it to a new list instance)
`tmp_nums.append(42)`	2: read tmp_nums, call the append method (which is itself a single operation)
`total = 0`	1: create variable total and assign it the value 0
`for num in lst_nums:`	
`total = total + num`	4 (**per iter**): read total, read num, add total + num, write to total
`return total`	1: read and return total to caller

Repeat N times => Total operations: 4 * N

Total operations for `f1()`: 10 + (4 * N) → O(N)

Tip: O(N) notation lets us not worry about this tedious bookkeeping, and instead let us think about the "big picture" of performance

Orders of Growth

Notation: Ω(N) vs O(N) vs Θ(N)

Let R(N) be a function that outputs the number of operations of a function f, in terms of the input problem size N.

Ω(R(N)): a lower-bound on growth

O(R(N)): an upper-bound on growth

Θ(R(N)): a "tight" bound on growth: the growth of a function is Θ(R(N)) if R(N) provides both a lower-bound AND upper-bound on the growth.

Note: Ω() and O() can be loose bounds. Ex: Ω(1) and O(infinity) are technically valid bounds for all functions, though not very useful bounds. In this class, for assignments/exams we'll only accept tight bounds for Ω() and O().

In this class: we'll generally only ask questions about tight bounds on O().
In classes like cs170 ("Algorithms"), you will study this topic in much greater detail

Aside: in practice, many people use "O(R(N))" when they actually mean "Θ(R(N))". Be mindful about the distinction, as there is a subtle difference

Common Orders of Growth

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

Quadratic growth.

Incrementing *n* increases *time* by *n* times a constant

Linear growth.

Incrementing *n* increases *time* by a constant

Logarithmic growth.

Doubling *n* only increments *time* by a constant

Constant growth. Increasing *n* doesn't affect time

(reference) Examples

<table>
<tr><th>Order of growth</th><th>Example function</th></tr>
<tr><td>O(1)</td><td><pre>def f1():
    return 4 * 2</pre></td></tr>
<tr><td>O(N)</td><td><pre>def f2(nums):
    total = 0
    for n in nums:
        total += n
    return total</pre></td></tr>
<tr><td>O(N^2)</td><td><pre>def f3(nums):
    total = 0
    for n1 in nums:
        for n2 in nums:
            total += n1 * n2
    return total</pre></td></tr>
<tr><td>O(N^3)</td><td><pre>def f4(nums):
    total = 0
    for n1 in nums:
        for n2 in nums:
            total += f2(nums)
    return total</pre></td></tr>
</table>

(reference) Examples

<table>
<tr><th>Order of growth</th><th>Example function</th></tr>
<tr><td>O(log(N))</td><td><pre>def f5(n):
    n_cur, out = n, 0
    while n_cur > 1:
        out += n_cur
        n_cur = n_cur // 2
    return out</pre></td></tr>
<tr><td>O(2^N)</td><td><pre>def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)</pre></td></tr>
</table>

O(1) < O(log(N)) < O(N) < O(N^2) < O(2 ^ N)

Spring 2023 Midterm 2 Question 3(a) Part (iii)

Definition. A *prefix sum* of a sequence of numbers is the sum of the first n elements for some positive length n.

(1 pt) What is the order of growth of the time to run prefix(s) in terms of the length of s? Assume append and + take one step (constant time) for any arguments.

```
def prefix(s):
    "Return a list of all prefix sums of list s."
    t = 0
    result = []
    for x in s:
        t = t + x
        result.append(t)
    return result
```

Answer: O(len(s))

Follow-up Question: what is the order of growth for this alternate implementation?

```
def prefix_alt(s):
    "Return a list of all prefix sums of list s."
    t = 0
    result = []
    for i in range(len(s)):
        result.append(sum(s[:i]))
    return result
```

Answer: O(len(s)^2)

Recall: Python slice creates a copy, eg is an O(N) operation, where N is the number of elements to copy
And:
1 + 2 + 3 + … + N = N * (N + 1) / 2

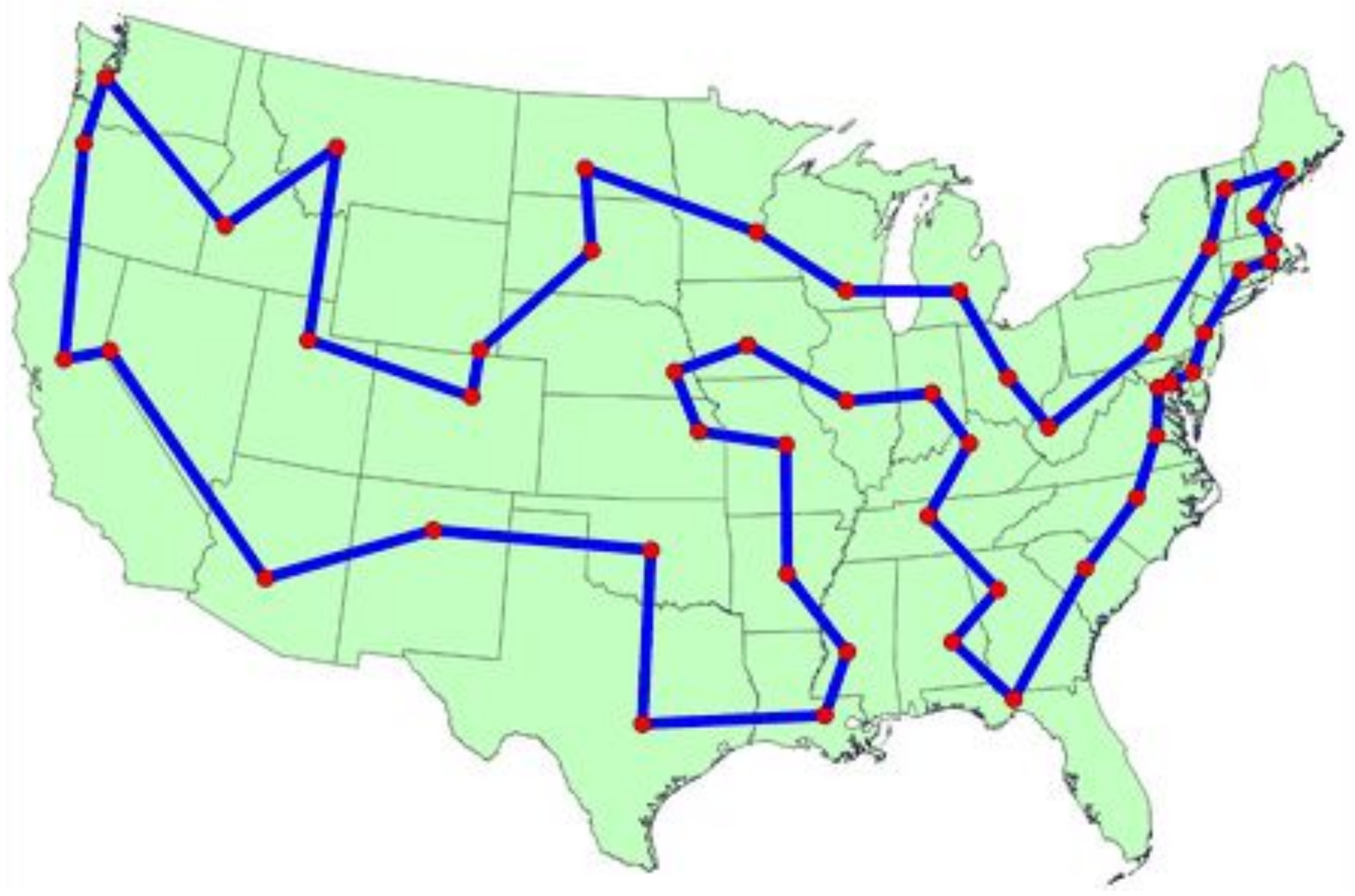
(Aside) P vs NP

- One of the central, unanswered questions in theoretical computer science involves the orders of growth of algorithms
- **Tractable orders of growth**: polynomial and smaller
  - ex: `O(1), O(log(N)), O(N), O(N^2), O(N^3), …`
  - These are algorithms that we (humanity) can reasonably solve for very large problem sizes
- **Intractable orders of growth**:
  - ex: `O(2^N), O(N^N), O(N!)`
  - These are algorithms that we can only solve for small/medium problem sizes

List contains: checking if an element is in a list (`elem in lst`) is O(N), a **tractable** order of growth.

Traveling Salesman Problem [link]: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city.

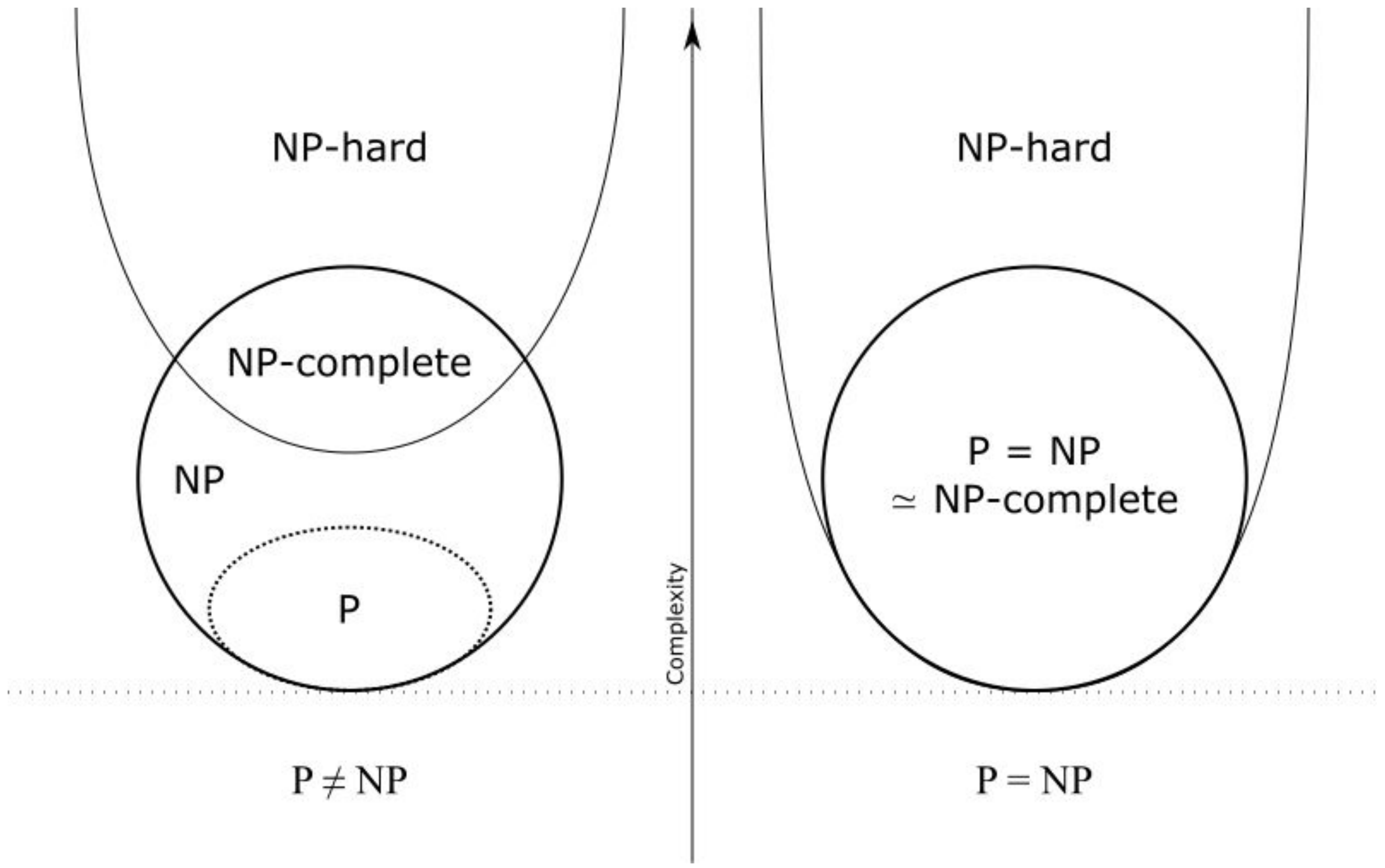
Held-Karp algorithm [link]: $O(n^2 2^n)$ where n is the number of cities



https://optimization.cbe.cornell.edu/index.php?title=Traveling_salesman_problem

(Aside) P vs NP

- Let's define the following sets of problems
- **P**: "easy" problems
  - Can **verify** in **polynomial time**
  - Can **solve** in **polynomial time**
- **NP**: set of problems that are easy to verify, but (currently) unknown if easy to solve
  - Can **verify** in **polynomial time**
  - Solving can be more expensive than polynomial time (eg exponential)
-



The Big Question: does the **set P equal the set NP**?

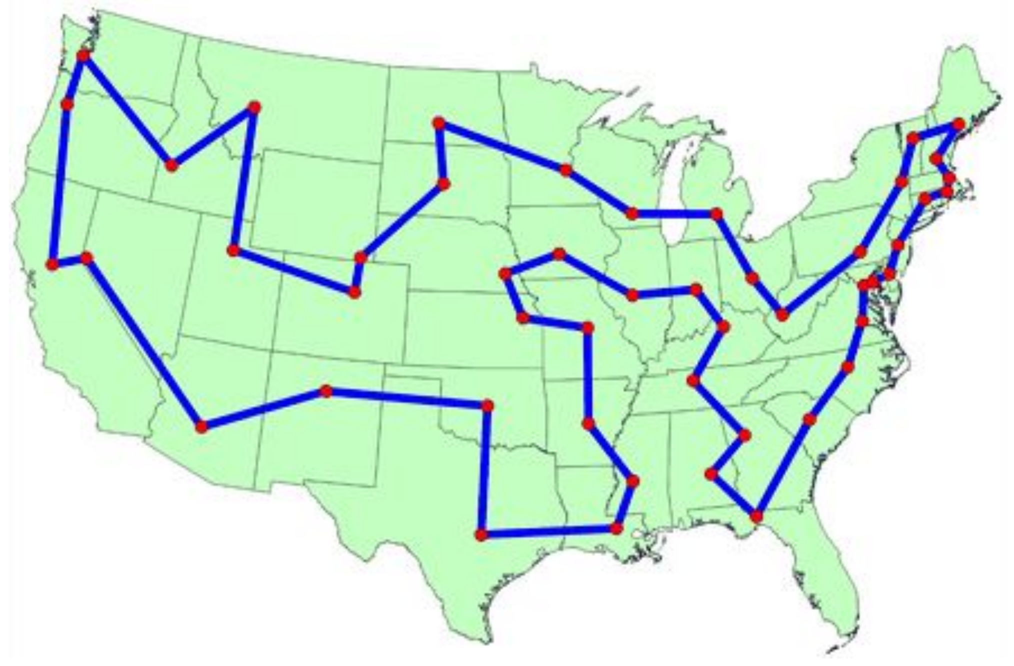
In other words: if a problem is easy to verify, does it also mean that it's easy to solve (**implies P = NP**)?

Or: is it possible that some problems are fundamentally difficult to solve (**implies P != NP**)?

One of the "Millenium Prize Problems" [link]. Winner gets $1M!

(Aside) NP-Complete

- Some very smart people have shown that
  - (1) There exists a class of problems, **NP-Complete**, that is verifiable in polynomial time (in NP), and
  - (2) All other problems in NP can be **converted to any NP-Complete problem** in polynomial time
- NP-Complete examples
  - Traveling Salesman Problem (TSP)
  - Knapsack problem [link]
  - ...
- Currently (as of 2025), no known efficient (polynomial) algorithm exists to solve any NP-Complete problem.

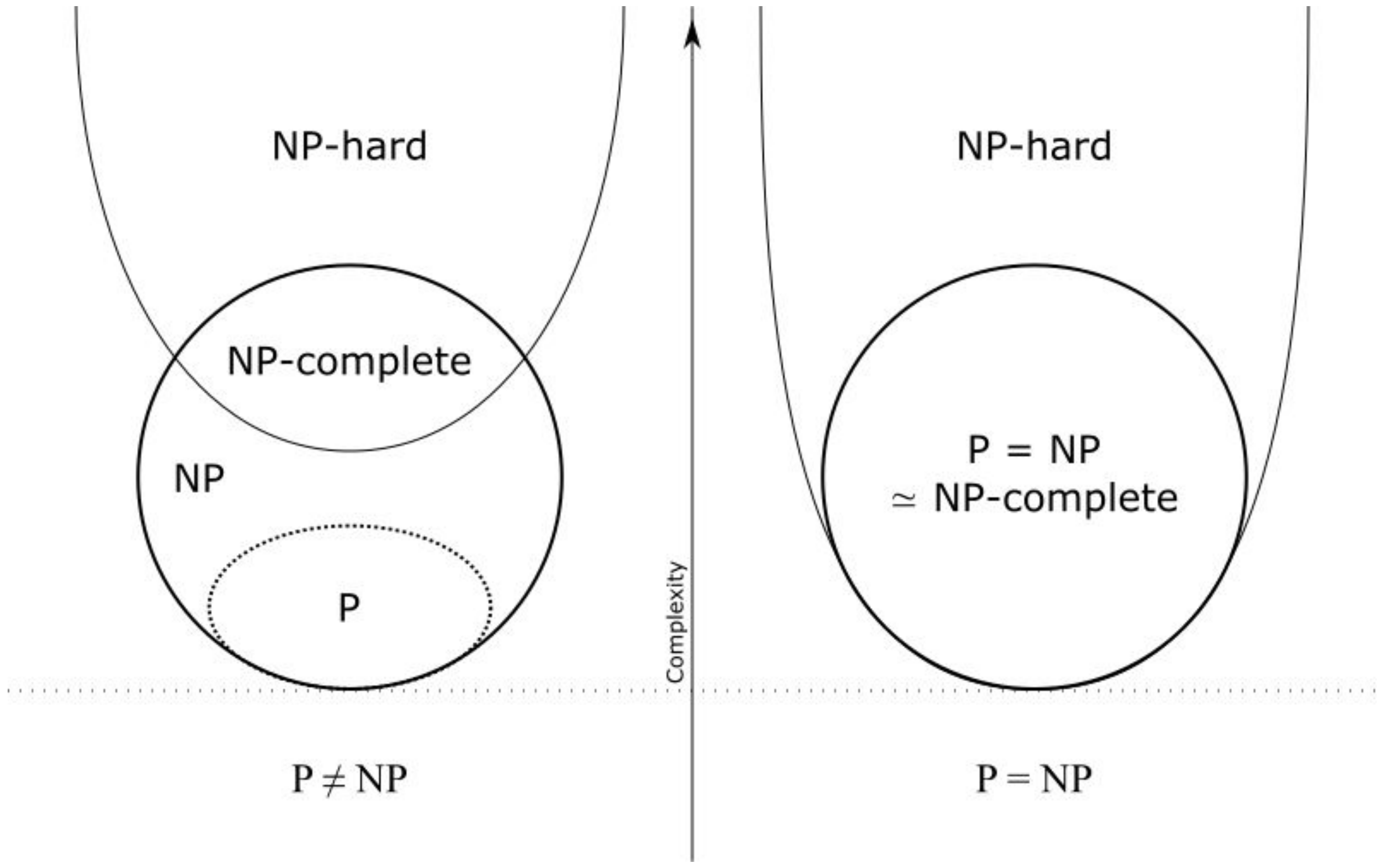


Crucially: if anyone finds an efficient (polynomial) algorithm to ANY NP-Complete problem, then we've found an efficient algorithm to ALL NP problems, which means we've discovered that: **P = NP**

https://optimization.cbe.cornell.edu/index.php?title=Traveling_salesman_problem

(Aside) Implications of P = NP

- Most modern cryptographic digital security becomes broken / insecure*
  - public-key cryptography
  - Cryptographic hashing, which powers blockchain technology!
- Automatic mathematical proof solvers would take a gigantic leap forward



* as always, "it depends". If the algorithm is something like O(n^100) or has a gigantic constant factor, then the algorithm may be impractical in practice

Image by OpenIcons from Pixabay

(Aside) P vs NP: What do experts think?

"Since 2002, William Gasarch has conducted three polls of researchers concerning this... Confidence that P ≠ NP has been increasing – in 2019, **88% believed P ≠ NP**, as opposed to **83% in 2012** and **61% in 2002**. When restricted to experts, the 2019 answers became **99% believed P ≠ NP**". [link_source]