Majority Element

Cloistered Monkey — 2018-07-05 16:01

Imports

With the exception of the defaultdict (https://docs.python.org/3.6/library/collections.html#collections.defaultdict), everything imported is for testing.

```python
# python standard library
from collections import defaultdict
from datetime import (
    datetime,
    timedelta,
    )
import random

# pypi
from expects import (
    equal,
    expect,
)
import numpy
```

The Majority Element Problem

Check whether a sequence contains an element that appears more than half of the time. Note that the problem doesn't ask if one of the elements appears more than all the other elements, but rather whether it appears more than half the time (it isn't a simple majority).

Input	A sequence of n integers
Output	1 if there is a majority element, 0 otherwise.
Constraints	$1 \leq n \leq 10^5; 0 \leq a_i \leq 10^9$ for all $1 \leq i \leq n$

Constants

```
class Constraints:
    min_length = 1
    max_length = 10**5
    min_value = 1
    max_value = 10**9
    max_time = timedelta(seconds=5)
```

```
class Outcome:
    has_majority = 1
    no_majority = 0
```

Samples

These are values to test against. The initial cases are for the naive-implementation and the last two are for an implementation that is more efficient.

```
class TestKeys:
    votes = 'input'
    expected = 'output'
```

```
SAMPLES = dict(
    one={
        TestKeys.votes: [2, 3, 9, 2, 2],
        TestKeys.expected: 1,
    },
    two={
        TestKeys.votes: [1, 2, 3, 1],
        TestKeys.expected: 0,
    },
    three={
        TestKeys.votes: [random.randint(1, Constraints.max_value),],
        TestKeys.expected: 1,
              }
)

vote = random.randint(1, Constraints.max_value)
SAMPLES["four"] = {
    TestKeys.votes: [vote, vote],
    TestKeys.expected: 1,
}
SAMPLES["five"] = {
    TestKeys.votes: [vote, vote + 1],
    TestKeys.expected: 0,
}
```

Now we're going to add two cases that have the maximum allowed number of values to make sure our solution can finish in a reasonable time.

```
half = Constraints.max_length//2

left_size = half + 10
right_size = half - 10

left = numpy.ones(left_size) * random.randrange(Constraints.max_value)
right = numpy.random.randint(Constraints.min_value,
                             Constraints.max_value + 1,
                             right_size)
three = numpy.concatenate((left, right))
SAMPLES["six"] = {
    TestKeys.votes: three,
    TestKeys.expected: 1,
}
```

This next case isn't necessarily guaranteed to be true (numpy might generate an array with one element that is in the majority), but I think that the chance of it failing is pretty close to zero.

```python
SAMPLES["seven"] = {
    TestKeys.votes: numpy.random.randint(Constraints.min_value,
                                         Constraints.max_value + 1,
                                         Constraints.max_length),
    TestKeys.expected: 0,

}
```

The Naive Solution

This is a translation of the pseudocode given with the problem. It has a runtime of $O(n^2)$

```python
def naive_majority(voters):
    """Decides if there is a majority element

   Args:
    voters: list of elements to check

   Returns:
    int: 1 if there is a majority element, 0 otherwise
   """
    half = len(voters)//2
    for index, voter in enumerate(voters):
        count = 0
        for other_voter in voters:
            if voter == other_voter:
                count += 1
        if count > half:
            return Outcome.has_majority
    return Outcome.no_majority
```

Now we can test if it is correct.

```
for sample in "one two three four five".split():
    values = SAMPLES[sample]
    actual = naive_majority(values[TestKeys.votes])
    expected = values[TestKeys.expected]
    print("{} actual: {} expected: {}".format(sample,
                                              actual,
                                              expected))

    expect(actual).to(equal(expected))
```

Although it looks correct the grader says it times out.

```
Failed case #11/21: time limit exceeded (Time used: 9.98/5.00, memory use
```

Iterative Version

Although this is the divide-and conquer section, the more intuitive way for me is to just count and sort the items to see if the item with the most votes is the majority. The counting of the votes is $O(n)$ and the sort adds $O(n \log n)$.

```
def iterative_majority(votes):
    """Decides if there is a majority among the votes

   Args:
    votes (list): collection to check

   Returns:
    int: 1 if there is a majority, 0 otherwise
   """
    half = len(votes)//2
    counts = defaultdict(lambda: 0)
    for vote in votes:
        counts[vote] += 1

    sorted_counts = sorted((count for count in counts.values()), reverse=True)
    return (Outcome.has_majority if sorted_counts[0] > half
            else Outcome.no_majority)
```

```python
def test_implementation(implementation):
    """runs the implementation against the samples

    Args:
     implementation: callable to test

    Raises:
     AssertionError: answer wasn't the expected
    """
    for sample, values in SAMPLES.items():
        start = datetime.now()
        actual = implementation(values[TestKeys.votes])
        expected = values[TestKeys.expected]
        elapsed = datetime.now() - start
        print("({}) elapsed: {} actual: {} expected: {}".format(
            sample,
            elapsed,
            actual,
            expected))
        expect(actual).to(equal(expected))
        assert elapsed < Constraints.max_time
```

```python
test_implementation(iterative_majority)
```

This version passes the grader.

```
Good job! (Max time used: 0.14/5.00, max memory used: 22638592/536870912.
```

Iterative Two

We can get rid of the sort since only have to check the count. This reduces the runtime to $O(n)$, although since the for-loop is now pure python it might not actually speed things up much.

```python
def iterative_majority_two(votes):
    """Decides if there is a majority among the votes

    Args:
     votes (list): collection to check

    Returns:
     int: 1 if there is a majority, 0 otherwise
    """
    half = len(votes)//2
    counts = defaultdict(lambda: 0)
    for vote in votes:
        counts[vote] += 1

    for count in counts.values():
        if count > half:
            return Outcome.has_majority
    return Outcome.no_majority
```

```python
test_implementation(iterative_majority_two)
```

This one also passes the grader.

```
Good job! (Max time used: 0.14/5.00, max memory used: 22626304/536870912.
```

It took exactly the same amount of time in the grader (although that might be because the time difference is less than their rounding), but used up a little less memory.

Binary Search

Cloistered Monkey — 2018-07-04 17:28

Binary Search

Binary search takes a sorted array and repeatedly divides it in half, checking whether an item searched for is at the mid-point. If you were to just traverse the array, you would have a runtime of $O(n)$. But because you are using divide and conquer (repeatedly halving to reduce the search space), you have a runtime of $O(\log n)$.

Imports

```python
# python standard library
from math import log

# pypi
from expects import (
    equal,
    expect,
)
```

Constants

```python
NOT_FOUND = -1
```

Linear Search

The naive way is to just traverse the list.

```python
def linear_search(a, x, verbose=True):
    """Brute-force search

    Args:
     a (list): source to search
     x : Item to search for
     verbose (bool): Emit number of loops

    Returns:
     int: index of x in a or -1 if not found
    """
    counter = 0
    for i in range(len(a)):
        if verbose:
            counter += 1
            print("Loop {}".format(counter))
        if a[i] == x:
            return i
    return NOT_FOUND
```

Interestingly, if you submit this it will time out.

```
Failed case #32/36: time limit exceeded (Time used: 9.98/5.00, memory use
```

The Algorithm

This is given as part of the problem statement.

```python
def binary_search(K, q, verbose=False):
    """Finds the index of an item in the list

    Args:
     K (list): A sorted list of integers
     q (int): the item to search for
     verbose (bool): if true, emit the number of loops done

    Returns:
     int: index of q in K or -1 if not found
    """
    min_index, max_index = 0, len(K) - 1
    counter = 0
    while max_index >= min_index:
        if verbose:
            counter += 1
            print("In loop {} - {}:{}".format(counter, min_index, max_index))

        mid_index = (min_index + max_index)//2

        if K[mid_index] == q:
            return mid_index
        elif K[mid_index] < q:
            min_index = mid_index + 1
        else:
            max_index = mid_index - 1
    return NOT_FOUND
```

Testing

```
q = 9
K = [1, 3, 7, 8, 9, 12, 15]
expected = 4
actual = binary_search(K, q, True)
expect(actual).to(equal(expected))
```

Looking at the output you can see that after finding that the middle element didn't match q it searched the upper half of the list (by raising the Minimum Index to 4) and then found q at the point where the Minimum and Maximum Index were equal.

We can compare this to the linear search.

```
actual = linear_search(K, q)
expect(actual).to(equal(expected))
```

It works but it takes a little longer. You can see the theoretical (Big-O) runtimes are different as well.

```
print("O(n): {}".format(len(K)))
print("O(log n): {:.2f}".format(log(len(K), 2)))
```

Sorted Array Multiple Search Problem

Because you need at least a linear runtime to read in the inputs, the grader can't tell that our search took less time than it took to read in the list. Because of this they set up a slightly harder problem to solve which can be graded.

Problem	*Search multiple keys in a sorted sequence of keys*
Input	A sorted array $K = [k_0, \ldots, k_{n-1}]$ of integers and $Q = [q_0, \ldots, q_{n-1}]$
Output	For each q_i, its index in K or -1 if it isn't in K
Constraints	$1 \le n, m \le 10^4, 1 \le k_i \le 10^9$ for all $0 \le i < n; 1 \le q_j \le 10^9$ for all \(0 \le j

Implementation

```python
def multiple_search(source, keys):
        """Searches the source for the keys

    Args:
     source (list): sorted list of search items
     keys (list): items to search for in the source

    Returns:
     list: indices of keys in source
    """
        return [binary_search(source, key) for key in keys]
```

I wrote this based on the problem statement, but if you look at the sample code they actually do the iteration themselves so you only need to implement `binary_search`.

Sample

Inputs

```
1 5 8 12 13
8 1 23 1 11
```

Outputs

```
2 0 -1 0 -1
```

```python
class TestKeys:
    source = 'source'
    search_terms = 'search-terms'
    expected = 'outputs'

TEST_CASES = dict(
    one={
        TestKeys.source: [1, 3, 7, 8, 9, 12, 15],
        TestKeys.search_terms: [9, 56, 3, 55, 1],
        TestKeys.expected: [4, -1, 1, -1, 0],
    },
    two={
        TestKeys.source: [1, 5, 8, 12, 13],
        TestKeys.search_terms: [8, 1, 23, 1, 11],
        TestKeys.expected: [2, 0, -1, 0, -1],
    }
)
```

Testing

```python
for example, case in TEST_CASES.items():
    print(example)
    expected = case[TestKeys.expected]
    actual = multiple_search(case[TestKeys.source],
                             case[TestKeys.search_terms])
    expect(actual).to(equal(expected))
```

Grading

The binary search improves quite a bit over the linear search, passing the grader.

```
Good job! (Max time used: 0.69/5.00, max memory used: 40230912/536870912.
```

Collecting Signatures

Cloistered Monkey — 2018-07-03 17:32

General Problem

Find the minimum number of points needed to cover all given segments on a line.

Input	A sequence of n segments $[a_1, b_1], \ldots [a_n, b_n]$ on a line
Output	A set of points of minimum size such that each segment contains a point

The Concrete Problem

You have to collect signatures from the tenants in the building. You know the times each tenant will be in the building (represented by the *segments* in the problem) and you want to minimize the number of visits and time spent at the building. Assume that the actual visit with the tenant will take no time.

In other words, we have a bunch of line segments that may or may not overlap. We want to minimize the number of segments

Input	n, the number of segments, each following line is made of two points tha define a line segment a_i, b_i
Output	The minimum number m of points needed, followed by the integer values for each of the points
Constraints	$1 \leq n \leq 100; 0 \leq a_i \leq b_i \leq 10^9$ for all i

Sample Inputs

Sample One

Input:

```
3
1 3
2 5
3 6
```

Output:

```
1
3
```

Note that the way the code is setup, the first input value isn't relevant to our solver.

Sample Two

Input:

```
4
4 7
1 3
2 5
5 6
```

Output:

```
2
3 6
```

Implementation

Imports

```
# from pypi
from expects import (
    equal,
    expect,
)
```

Overlapping

First, what does it mean to say that two segments overlap? Let's say we have two segments. They won't overlap if:

- the first segment's rightmost point is to the left of the other segment's leftmost point
- the second segment's rightmost point is to the left of the other segment's leftmost point

So they won't overlap if:

$$R_0 < L_1 \lor L_0 > R_1$$

Where R means the rightmost point for that segment and L means the leftmost point of that segment (and the first segment is 0 and the second one is 1). To find where they *do* overlap we can negate the inequality.

$$\neg(R_0 < L_1 \lor L_0 > R_1) = R_0 \leq L_1 \land L_0 \leq R_1$$

Python functions are expensive, but to make it clearer I'll create a function to test for overlapping and if the final solution is too small I won't use it.

The Schedule

```python
def schedule(schedules):
    """Finds the times to visit

    Args:
     schedules (list): list of times people are available

    Returns:
     list: times to visit
    """
    return
```

Testing

```python
SAMPLES = dict(
    one=dict(
        inputs=[(1,3), (2, 5), (3, 6)],
        outputs=[3],
    ),
    two=dict(
        inputs=[(4, 7), (1, 3), (2, 5), (5, 6)],
        outputs=[3, 6],
    )
)
```

```python
class SampleKeys:
    inputs = "inputs"
    expected = "outputs"
```

```python
for sample, values in SAMPLES.items():
    actual = schedule(values[SampleKeys.inputs])
    expect(actual).to(equal(values[SampleKeys.expected]))
```

Maximum Advertisement Revenue

Cloistered Monkey — 2018-07-03 16:46

The Maximum Product of Two Sequences Problem

This is the more general problem statement.

Problem	*Find the maximum dot product of two sequences of numbers.*
Inputs	Two sequences of n positive integers.
Output	The maximum sum of pair-wise multiplications of the values.

The Revenue Optimization Problem

We have n advertising slots that we want to sell to advertisers. Each slot gets a different number of clicks and each advertiser is willing to pay a different amount. How do you pair the advertiser with the slot to maximize you click-revenue?

Input	Sequence of integer prices $price_1, price_2, \ldots, price_n$ and a sequence of click-counts $count_1, count_2, \ldots, count_n$.
Output	The maximum value achievable by matching prices with click counts
Constraints	$1 \leq n \leq 10^3; 0 \leq price_i, clicks_i \leq 10^5$ for all $1 \leq i \leq n$

Samples

n	prices	clicks	output
1	23	39	897
3	2 3 9	7 4 2	79

Testing

```
# from pypi
from expects import (
    equal,
    expect,
)
```

```
SAMPLES = dict(one=
               dict(prices=[23,],
                    clicks=[39,],
                    output=897),
               two=
               dict(prices=[2, 3, 9],
                    clicks=[7, 4, 2],
                    output=79)
)
```

```python
class Keys:
    prices = "prices"
    clicks = "clicks"
    expected = "output"
```

Implementation

This might be cheating, but I'm going to use python's generator functions again to sort things.

```python
def optimal_advertising(prices, clicks):
    """Finds the optimal dot product

    Args:
     prices (list): prices we can charge advertisers
     clicks (list): expected clicks per slot

    Returns:
     float: the maximum we can get from the prices-clicks
    """
    clicks = sorted(clicks, reverse=True)
    prices = sorted(prices, reverse=True)
    clicks_and_prices = zip(clicks, prices)
    return sum(click * price for click, price in clicks_and_prices)
```

Testing

```python
for label, sample in SAMPLES.items():
    expected = sample[Keys.expected]
    actual = optimal_advertising(sample[Keys.prices], sample[Keys.clicks])
    expect(actual).to(equal(expected))
```

Grader Output

```
Good job! (Max time used: 0.03/5.00, max memory used: 9887744/536870912.)
```

Maximum Value of the Loot

Cloistered Monkey — 2018-07-02 19:09

Introduction

A thief breaks into a spice shop and finds spices with varying values per pound. She needs to be able to maximize the amount she steals by stuffing spices into her backpack.

	Description
First Input	n, the number of compounds and W, the capacity of the backpack.
Remaining Inputs	n lines of price-per-pound and weight of each compound
Output	Maximum price of spices stuffed into the backpack.
Constraints	$1 \leq n \leq 10^3, 0 \leq W \leq 2 \cdot 10^6, 0 \leq p_i \leq 2 \cdot 10^6, 0 \leq w_i \leq 2 \cdot 10^6$ for $1 \leq i \leq n$

Although the inputs will always be integers, the outputs might be real numbers. To match the grader output at least four digits to the right of the decimal point.

Samples

Sample One

Input:

```
3 50
60 20
100 50
120 30
```

Output:

```
180.0000
```

The output tells us that the thief's maximum haul is worth $180, which if you look at the inputs means taking 20 pounds of the first spice (worth $60) and 30 pounds of the last spice.

Sample Two

```
1 10
500 30
```

Output.

```
166.6667
```

The input tells us that the thief can only carry 10 pounds of the only available spice, so her haul is $\frac{500}{3} \approx 166.6667$.

Implementation

Because this takes a greedy approach, it will have a $O(n)$ run-time. Since I'm sorting the values first there's actually a $O(\log n) + O(n)$, but especially since I'm using the built-in python generators, the sort is negligible compared to the main loop.

```
# this package
from algorithmic_toolbox.helpers import assert_close
```

```python
def maximize_loot(capacity, weights, values):
    """Figure out the maximum value the thief can haul off

   Args:
    capacity (int): number of pounds backpack can hold
    weights (list): how many pounds of each item there is
    values (list): how much each item is worth per pound

   Raises:
    AssertionError: weights and values are different lengths

   Returns:
    float: max-value the backpack can hold
   """
    weight_count = len(weights)
    assert weight_count == len(values), \
        "Weights and Values not same shape: weights={} values={}".format(
            weight_count, len(values))
    values_per_pound = ((values[index]/weights[index], index)
                        for index in range(weight_count))

    # we have to reverse-sort it (otherwise sorting puts the smallest
    # number first)
    per_poundage = sorted(values_per_pound, reverse=True)

    # loot is the value of what we've taken so far
    loot = 0

    # precondition: per_poundage is the value-per-pound in descending
    # order for each item along with the index of the original weight/value
    for value, index in per_poundage:
        # invariant: value is the largest price-per-pound available
        if capacity < weights[index]:
            # we don't have enough strength to take all of this item
            # so just take as much as we can and quit
            loot += value * capacity
            break
        # otherwise take all of this item
        loot += values[index]
        # reducing our capacity by its total weight
        capacity -= weights[index]
        if capacity == 0:
```

```
                # we're out of capacity, quit
                break
    return loot
```

Test One

```
n = 3
capacity = 50
prices = [60, 100, 120]
weights = [20, 50, 30]
expected = 180.0000
actual = maximize_loot(capacity, weights, prices)
assert_close(expected, actual, "Test One")
```

Test Two

```
capacity = 10
prices = [500]
weights = [30]
expected = 166.6667
actual = maximize_loot(capacity, weights, prices)
assert_close(expected, actual, "Test Two")
```

Grader Output

```
Good job! (Max time used: 0.03/5.00, max memory used: 9752576/671088640.)
```

Money Change

Cloistered Monkey — 2018-07-02 16:40

Problem Description

Task	Find the minimum number of coins to change the input to coins with denominations 1, 5, 10
Input	A single integer m.
Constraints	$1 ≤ m ≤ 10^3&
Output	Minimum number of coins with denominations 1, 5, or 10 that changes m.

Samples

Input	Output	Coins
2	2	1 + 1
28	6	10 + 10 + 5 + 1 + 1 + 1

Solution

While m is greater than 0, keep taking a coin with the largest denomination that isn't greater that m, subtracting its value from m.

```
DENOMINATIONS = 10, 5, 1
```

```
def change_money(money):
    """Make change

    Args:
     money (int): amount to break

    Returns:
     int: minimum number of coins that money breaks into
    """
    coins = 0
    while money > 0:
        for denomination in DENOMINATIONS:
            if money >= denomination:
                money -= denomination
                coins += 1
                break
    return coins
```

```
expected = 2
actual = change_money(2)
assert expected == actual
expected = 6
actual = change_money(28)
assert expected == actual
```

Although this is really a brute-force approach, it is good enough.

```
Good job! (Max time used: 0.03/5.00, max memory used: 9596928/536870912.)
```

If you look at it, even in the worst case where you only give out pennies, the maximum run time is the value of $money$, that is, if $money = 1.50$ then the maximum theoretical run time is 150, regardless of the denominations, so this solution is $O(n)$, even though it looks brute-force-ish.

Least Common Multiple

Cloistered Monkey — 2018-06-27 11:08

Introduction

The least common multiple (https://en.wikipedia.org/wiki/Least_common_multiple) of two positive integers, *a* and *b* is the least positive integer *m* that is divisible by both *a* and *b*.

Problem Description

	Description
Task	Given two integers a and b find their least common multiple.
Input	The integers a and b on the same line separated by whitespace.
Constraints	$1 \leq a, b \leq 2 \cdot 10^9$
Output	The least common multiple of a and b.

Samples

Input	Output
6 8	24
28851538 1183019	1933050546

```
SAMPLES = {(6, 8): 24,
           (28851538, 1183019): 1933053046,
}
MAX_INPUT = 2 * 10**9
```

Imports

```
from algorithmic_toolbox.helpers import time_two_inputs
from algorithmic_toolbox.implementations import greatest_common_divisor
```

Naive

```python
def lcm_naive(a, b):
    """Computes the Least Common multiple of a and b

    Args:
     a, b: non-negative integers

    Returns:
     int: the least common multiples of a and b
    """
    for l in range(1, a*b + 1):
        if l % a == 0 and l % b == 0:
            return l

    return a*b
```

```python
time_two_inputs(lcm_naive, "Naive", (6, 8), 24, MAX_INPUT)
```

As expected, this fails the grader.

```
Failed case #2/42: time limit exceeded Input: 14159572 63967072 Your outp
```

Using The Greatest Common Divisor

If you multiply both numbers together you will get their greatest common multiple. If you then divide that by their greatest common divisor, you will be left with their least common multiple.

```python
def lcm_gcd(a, b):
    """Finds the least common multiple of two integers

    Args:
     a, b: integers greater than or equal to 1
    """
    return a * b//greatest_common_divisor(a, b)
```

```python
for a_and_b, answer in SAMPLES.items():
    time_two_inputs(lcm_gcd, "GCD", a_and_b, answer, MAX_INPUT)
```

The Grader output:

```
Good job! (Max time used: 0.27/5.00, max memory used: 9924608/536870912.)
```

Greatest Common Divisor

Cloistered Monkey — 2018-06-26 14:51

Introduction

The greatest common divisor $GCD(a, b)$ of two non-negative integers (a and b) which are not both equal to 0 is the greatest integer d that divides both a and b. The goal here is to implement the Euclidean Algorithm (https://en.wikipedia.org/wiki/Euclidean_algorithm) for computing the GCD.

Problem Description

	Description
Task	Given two integers a and b, find their greatest common divisor.
Input	The two integers a and b are given on the same line separated by a space.
Constraints	$1 \leq a, b \leq 2 \cdot 10^9$
Output	GCD(a,b)

Imports

```
# python standard library
from datetime import (
    datetime,
    timedelta,
)
```

Samples

Input	Output
18 35	1
28851538 1183019	17657

```
SAMPLES = {(18, 35): 1,
           (28851538, 1183019): 17657}
MAX_TIME = timedelta(5)
MAX_INPUT = 2 * 10**9
```

```
def time_two_inputs(implementation, tag, a_and_b, expected, max_time=MAX_TIME, max_input=MA
    """Time the implementation

    Args:
     implementation: callable to time
     tag (str): name for the output
     a_and_b (tuple): inputs for the implementation
     expected (int): the expected output of the implementation

    Raises:
     AssertionError: output was wrong or it took too long
    """
    a, b = a_and_b
    assert a <= max_input, "a too large: {}".format(a)
    assert b <= max_input, "b too large: {}".format(b)
    print("Starting {}".format(tag))
    start = datetime.now()
    actual = implementation(a, b)
    elapsed = datetime.now() - start
    print("Elapsed time: {}".format(elapsed))
    assert actual == expected, "Expected: {} Actual: {}".format(expected, actual)
    assert elapsed <= MAX_TIME, "Took too long: {}".format(elapsed)
    return
```

Naive GCD

```
def gcd_naive(a, b):
    """Naive implementation of GCD

    Args:
     a, b: non-negative integers
    """
    current_gcd = 1
    for d in range(2, min(a, b) + 1):
        if a % d == 0 and b % d == 0:
            if d > current_gcd:
                current_gcd = d
    return current_gcd
```

```python
for inputs, answer in SAMPLES.items():
    time_it(gcd_naive, "Naive", inputs, answer)
```

This fails the grader.

```
Failed case #10/22: time limit exceeded Input: 100000000 100000000 Your c
```

Modulus Version

This is a variation on Euclid's Algorithim where you repeatedly use the remainder of \frac{a, b} to replace b until there is no remainder ($b = 0$).

```python
def gcd_modulus(a, b):
    """finds the GCD of a and b

    Args:
     a, b: non-negative integers

    Returns:
     int: the GCD of a and b
    """
    while b != 0:
        a, b = b, a % b
    return a
```

```python
for inputs, answer in SAMPLES.items():
    time_it(gcd_modulus, "Modulus", inputs, answer)
```

```python
a_b = 100000000, 100000000
start = datetime.now()
expected = gcd_naive(*a_b)
print("Elapsed: {}".format(datetime.now() - start))
```

My computer appears to be faster than the grader, but it still fails.

```python
time_it(gcd_modulus, "Modulus", a_b, expected)
```

Last Digit of a Large Fibonacci Number

Cloistered Monkey — 2018-06-25 16:28

Introduction

The goal is to find the last digit of the n-th Fibonacci number. The problem is that Fibonacci numbers grow exponetially fast. For instance

$$F_{200} = 280571172992510140037611932413038677189525$$

So even our iterative version will prove too slow. Also, it may produce numbers that are too large to fit in memory. So instead we are going to only save the last digit of each number.

$$F_i \leftarrow (F_{i-1} + F_{i-2}) \mod 10$$

Problem Description

	Description
Task	Given an integer n, find the last digit of the /n/th Fibonacci number F_n ($F_n \mod 10$)
Input	A single integer n.
Constraints	$0 \leq n \leq 10^7$
Output	The last digit of F_n

Samples

Input	Output
3	2
331	9
327305	5

Constants

```
MAX_INPUT = 10**7
MAX_TIME = 5
```

Imports

```
from datetime import (
    datetime,
    timedelta,
    )
# this project
from algorithmic_toolbox.helpers import time_it
```

Naive Implementation

By taking the modulo of 10 for the final number you reduce it to the final digit because it's the remainder of some number times 10. For example, 112 is 110 + 2, so $112 \mod 10$ is $112 - (11 \times 10) = 2$.

```
def get_fibonacci_last_digit_naive(n):
    if n <= 1:
        return n

    previous = 0
    current  = 1

    for _ in range(n - 1):
        previous, current = current, previous + current
    return current % 10
```

```
time_it(get_fibonacci_last_digit_naive, "Naive", 3, 2, max_input=MAX_INPUT)
```

```
time_it(get_fibonacci_last_digit_naive, "Naive", 331, 9, max_input=MAX_INPUT)
```

Modulo Version

Each number in the sequence is the sum of the previous two numbers. The last digit is always the sum of the last digits of the previous two numbers. So to calculate the last digit you only need to keep track of the last digit of each number. By taking the modulus of 10, you are always.

```
first = (0, 1)
def get_fibonacci_last_digit_modulo(n):
    if n in first:
        return n

    previous, current = first

    for _ in range(n - 1):
        previous, current = current, (previous + current) % 10
        print("Current: {}".format(current))
    return current
```

```
time_it(get_fibonacci_last_digit_modulo, "Modulo", 3, 2, max_input=MAX_INPUT)
```

```
time_it(get_fibonacci_last_digit_modulo, "Modulo", 331, 9, max_input=MAX_INPUT)
```

```
time_it(get_fibonacci_last_digit_modulo, "Modulo", 327305, 5, max_input=MAX_INPUT)
```

```
time_it(get_fibonacci_last_digit_modulo, "Modulo", 200, 5, max_input=MAX_INPUT)
```

This is the grader output.

```
Good job! (Max time used: 0.12/5.00, max memory used: 9580544/536870912.)
```

Fibonacci Number

Cloistered Monkey — 2018-06-24 20:49

Imports

```python
# python standard library
from datetime import (
    datetime,
    timedelta,
    )
import random

# third party
from joblib import Memory
```

The Fibonacci Problem

THe Fibonacci Sequence is defined as

$$F_0 = 0, F_1 = 1, \ldots, F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2$$

	Description
Task	Given an integer n, find the /n/th Fibonacci number F_n.
Input	A single integer n.
Constraints	$0 \leq n \leq 45$
Output	F_n

Sample Values

Input	Output	Meaning
10	55	$F_10 = 55$

Constants

These are translations from the problem description

```
MAX_TIME = timedelta(seconds=5)
MAX_INPUT = 45
```

Helpers

```
def time_it(implementation, tag, input_value, expected, max_time=MAX_TIME):
        """Times the implementation

    Args:
     implementation: callable to pass input_value to
     tag (str): identifier to add to the output
     input_value (int): the number to pass to the implementation
     expected (int): the expected value
     max_time (float): number of seconds allowed

    Raises:
       AssertionError: wrong value or took too long
    """
        assert input_value <= MAX_INPUT, "n too large: {}, max allowed: {}".format(input_value,

        start = datetime.now()
        print("Starting {}".format(tag))
        actual = implementation(input_value)
        assert actual == expected, "Actual: {} Expected: {}".format(
                    actual, expected)
        elapsed = datetime.now() - start
        print("({}) Okay Elapsed time: {}".format(tag, elapsed))
        assert elapsed <= max_time, "Time Greater than {}".format(max_time)
        return
```

Naive Implementation

The 'naive' implementation uses recursion to calculate a fibonacci number.

```python
def calculate_fibonacci_recursive(n):
    """Calculates the nth fibonacci number

    Args:
     n (int): the fibonacci number to get (e.g. 3 means third)

    Returns:
     int: nth fibonacci number
    """
    if (n <= 1):
        return n
    return (calculate_fibonacci_recursive(n - 1)
            + calculate_fibonacci_recursive(n - 2))
```

```python
time_it(calculate_fibonacci_recursive, "Recursive", 10, 55)
```

This fails the grader (as expected).

```
Failed case #37/46: time limit exceeded
Input: 36
Your output: 14930352
stderr: (Time used: 5.63/5.00, memory used: 9613312/536870912.)
```

The Tester

We have kind of a chicken and the egg problem here. We know that the recursive version is correct, but it is too slow. But in order to validate our newer versions, we need to run it to check for correctness. To solve this problem I'm going to use a cache.

```
memory = Memory(location="memories")

@memory.cache
def reference_implementation(n):
        """Calculates the nth fibonacci number

    Args:
     n (int): the fibonacci number to get (e.g. 3 means third)

    Returns:
     int: nth fibonacci number
    """
        if (n <= 1):
                return n
        return (calculate_fibonacci_recursive(n - 1)
                        + calculate_fibonacci_recursive(n - 2))
```

```
def time_once(implementation, n):
        """Runs the implementation once

    Args:
     implementation: callable to pass input
     n: value to pass to the implementation

    Returns:
     output of implementation
    """
        start = datetime.now()
        output = implementation(n)
        print("Elapsed: {}".format(datetime.now() - start))
        return output
```

```
def run_range(n):
    """run the reference implementation n times

   Args:
    n (int): number of times to run the reference implementation
   """
    start = datetime.now()
    for input_value in range(n):
        output = reference_implementation(input_value)
    return
```

```
for endpoint in range(0, MAX_INPUT + 1, 10):
    print("endpoint: {}".format(endpoint))
    for n in range(endpoint):
        run_range(n)
```

```
time_once(reference_implementation, 45)
```

In case I accidentally re-run that last call and it uses the cache I'll note here that the original run time was 8 minutes and 40 seconds.

```
class Tester:
    """Class to test the implementation

   Args:
    implementation: callable to pass input_value to
    tag (str): identifier to add to the output
    iterations (int): number of times to run the testing
    verbose (bool): if true, emit more text
    max_time (float): number of seconds allowed
   """
    def __init__(self, implementation, tag, iterations,
                 verbose=False,
                 max_time=MAX_TIME):
        self.implementation = implementation
        self.tag = tag
        self.max_time = max_time
        self.verbose = verbose
        self.iterations = iterations
        return

    def output(self, statement):
        """prints the statement if verbose is on"""
        if self.verbose:
            print(statement)

    def time_it(self, input_value):
        """Times the implementation

     .. warning:: This uses the ``reference_implementation`` to get the
       expected value. Make sure it's implemented and the values are cached

     Args:
      input_value (int): input for the implementation
     Raises:
      AssertionError: wrong value or took too long
     """
        start = datetime.now()
        self.output("Starting {}".format(self.tag))
        expected = reference_implementation(input_value)
        actual = self.implementation(input_value)
        assert actual == expected, "n: {} Actual: {} Expected: {}".format(
            input_value, actual, expected)
```

```python
        elapsed = datetime.now() - start
        self.output("({}) Okay Elapsed time: {}".format(self.tag, elapsed))
        assert elapsed <= self.max_time, "Time Greater than {}".format(self.max_time)
        return

    def __call__(self):
        """Generates random numbers and times it"""
        start = datetime.now()
        print("***** {} *****".format(self.tag))
        for iteration in range(self.iterations):
            n = random.randrange(MAX_INPUT + 1)
            self.output("n: {}".format(n))
            self.time_it(n)
        print("Total Elapsed: {}".format(datetime.now()))
        return
```

An Iterative Version

To try and speed things up I'm going to use an iterative version instead of a recursive one.

```python
def fibonacci_iterative(n):
    """Calculates the nth fibonacci number

    Args:
     n (int): the fibonacci number to get (e.g. 3 means third)

    Returns:
     int: nth fibonacci number
    """
    first = (0, 1)
    if n in first:
        return n
    previous, current = first
    for index in range(2, n + 1):
        previous, current = current, previous + current
    return current
```

```python
test = Tester(fibonacci_iterative, "Iterative", 1000)
test()
```

```
f_0 = fibonacci_iterative(45)
f_1 = reference_implementation(45)
print(f_0)
print(f_1)
assert f_0 == f_1
```

This passes the grader.

```
Good job! (Max time used: 0.03/5.00, max memory used: 9637888/536870912.)
```

Older posts (index-1.html)