Algorithms and Complexity

Unit 3

David Emanuel Craciunescu

Exercise 1

For a non-empty vector of non-repeating pseudo-randomly-generated integer values ordered in increasing order, design a divide-and-conquer algorithm that finds if the vector is coincident. A coincident vector is that in which at least one value is contained in an index equal to that value. An example would be: [-14, -2, 2, 4, 7, 9]. The value 2 can be found in index [2], therefore the vector is coincident.

The complexity of the provided solution should not exceed O(logn). Consider as inputs the vector and its size.

Solution

We've tried to come up with a different solution to this exercise than the provided one, so we've mixed up gradient descent with binary search. The method iterates over the provided array and decreases the jump with each iteration. We've basically implemented a "harmonic series search" that helps us find the solution. It doesn't comply with the complexity restriction of O(logn) time, but it's able to solve it in upper-bound O(√n) time, since the harmonic series does seem to compute to the square root as an upper bound.

Implementation

```
def coincident(vector: List[int]) -> bool:
    length = len(vector)
    pos = math.ceil(length/2)

    if length % 2 != 0: length -= 1

    iteration = 1
    while 0 <= pos < len(vector):
        if vector[pos] == pos: return True

        jump = math.ceil(length // 2^(iteration+1))

        if vector[pos] > pos: pos -= jump
        else: pos += jump

        iteration += 1

    return False
```

Tests

```
def test_coincident01(self):
    provided = coincident([5, 8, 4, 1, 7, 9])
    self.assertFalse(provided)

def test_coincident02(self):
    provided = coincident([0, 1, 2, 3, 4])
    self.assertTrue(provided)

def test_coincident03(self):
    provided = coincident([])
```

```
    self.assertFalse(provided)
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U3/test/test_e1.py
----------------------------------------------------------------------
Ran 4 tests in 0.000s | OK
```

Exercise 2

We are given a vector of unordered positive integers that we cannot alter or copy in any way, shape, or form. Yet, we'd like to obtain information from this vector as if it were ordered. One idea we have is creating an index vector Ind[1…N]. This type of vector would store the position in which each element of the original vector would be if the original vector were ordered.

This would be an example of a vector right next to his index vector. As you can see, the index vector simply stores the position in which every element of the original vector would be if this were ordered.

- Vector[1…N] => [50, 98, 10, 63, 31, 25, 63, 74]
- Index[1…N] => [3, 6, 5, 1, 4, 7, 8, 2]

Modify the standard MergeSort algorithm in order to obtain the vector indices of a vector V[1…N] given as a parameter.

Exercise 3

Consider a function f(x) which:

- Is known to have a unique local minimum called x0 at a point in the interval [p1, p2], that **can** be p1 or p2.
- Is strictly decreasing between [p1, x0]
- Is strictly increasing between [x0, p2]

Your task is to search, as efficiently as possible, for all points between p1 and p2 and find that unique local minimum. Formally, if f(x) = k, search for x ∈ [p1, p2]. To simplify the process, instead of the exact value of each x, a range of values [α, β] can be indicated, with a β - α < ε, where x is found. The algorithm data will be the interval [p1, p2], the value k that is being searched for, and the value ε for the approximation.

Exercise 4

We'd like to program a robot to match cork stoppers to glass bottles at a recyling factory. The factory has N bottles and has one (and only one) cork stopper to match each bottle. The process would seem simple but there are a number of details we need to take into consideration first:

- The bottles are all different from one another, just like the corks. Each bottle can only be closed with its correspondent cork and each cork only serves for a specific bottle.
- The robot only know how to close bottles. It has the sensors needed to know if a cork is too big or too small for a bottle but is not programmed to compare corks or bottles between themselves. So the robot would not be able to compare between corks or bottles and sort them by thickness.
- The robot has the space available and mechanical arms to place bottles and corks at will, for example in different positions of a table, if needed.

Design an algorithm that follows the Divide and Conquer scheme to plug N bottles optimally. Analyze the complexity of the provided solution and explain the reasoning behind it.

Solution

This exercise has been extremely fun and also challenging to implement. First of all, the solution is obviously quicksort, which has a complexity of O(nlogn). We cannot compare elements between themselves so forcing us to use elements from the other list to order the first one was a pretty clever approach.

Below I've attached the implementations of the quicksort pivot method, the main quicksort method used to sort the data, and a binary sort that I used to obtain the pivots faster (although this is not the whole program). Describing what quicksort does in a short memory is extremely hard, so I've made the code as straightforward and clear as possible.

If it were not clear enough, please contact me and I'll attach a detailed explanation of the quicksort algorithm.

Implementation

```
def pivot(val, data, start, end):
    left, right = start, end

    # Iterate until unordered element
    while left < right:
        while data[left] < val: left += 1
        while data[right] > val: right -= 1

        # Swap if out of place
        if left <= right:
            data[left], data[right] = data[right], data[left]
            left += 1
            right -= 1

    return (data, left, right)
```

```
def quick_sorted(data: List[Number], compare: List[Number], start: int, end: int) -> List[Number]:
    if start == end: return data

    # Obtain a pivot and sort sides.
    pivot_idx = search_binary(data[start], compare)
    pivot_elem = compare[pivot_idx]
    data, left, right = pivot(pivot_elem, data, start, end)

    # If not yet fully sorted.
    if start < right: quick_sorted(data, compare, start, right)
    if end > left: quick_sorted(data, compare, left, end)

    return data
```

```python
def search_binary(val: Number, data: List[Number]) -> Optional[int]:
    start = 0
    end = len(data)-1

    def search_binary_aux(val: Number, data: List[Number], start: int, end: int) -> Optional[int]:
        mid_idx = (start + end) // 2
        mid_val = data[mid_idx]

        # Cases
        if mid_val == val: return mid_idx
        if mid_idx == end: return None
        if mid_val < val:  return search_binary_aux(val, data, mid_idx+1, end)
        else:              return search_binary_aux(val, data, start, mid_idx)

    return search_binary_aux(val, data, start, end)
```

Tests

The tests have also been quite interesting. What we do in order to generate datasets is to generate two ordered lists of integer values which zipped will generate our “expect” dataset. Then we shuffle them and check that our algorithm returns them in order.

```python
@staticmethod
def data_generator(limit):
    length = randint(1, limit)

    provided1 = list(range(length))
    provided2 = provided1[:]
    expect = list(zip(provided1, provided2))

    shuffle(provided1)
    shuffle(provided2)

    return (provided1, provided2, expect)

def test_match01(self):
    set1, set2, expected = self.data_generator(100)
    provided = match(set1, set2)
    self.assertEqual(provided , expected)

def test_match02(self):
    set1, set2, expected = self.data_generator(1000)
    provided = match(set1, set2)
    self.assertEqual(provided, expected)
```

And here’s evidence that the tests run correctly.

```
┌dave💻ec2 🗀 algo 🌿master ❯ python3 -m unittest U3/test/test_e4.py
----------------------------------------------------------------------
Ran 4 tests in 6.382s | OK
```

Exercise 5

In Aceland, the very celebrated national sport is tennis. The country keeps a ranking in which each player is assigned a number of points according to their quality as a player. In other words, the best player in the country is the one with the highest number of points.

Every year a couple from Aceland competes in an international doubles tennis tournament, but the way the Acelandics select their best couple is a little peculiar, though. Each player's name is placed on a list in a random position, and each player can only compete with the players that are adjacent to them in the list. In other words, the ones in front of them and behind them on the list.

Obviously, the first player on the list can only pair up with the second, and the last player can only pair up with the one before them. The rest do have two possible options to form a pair of doubles, the one before or the one after them. With this seemingly nonsensical restriction and methodology, the Aceland team chooses its best pair of players. What they do is count the sum ofthe points of each pair and select the pair with the greatest amount.

Design a divide-and-conquer algorithm that shall help the Acelandics find their next pair of tennis champions, and analyze the complexity of the provided solution.

Exercise 6

After passing through the Tile Room and stealing the Craddle of Life, Indiana Croft faces a new challenge before leaving the Cursed Temple! The Temple itself is located on a bridge under which there is a deep darkness. Fortunately, this place also appears in the diary. The bridge crosses the so-called Valley of Shadows, which begins with a descent slope (not necessarily constant), so that after reaching the lowest point he must start to climb to the other end of the bridge.

Just at the bottom of the valley, one can find a river, but the diary does not give any specific information about its whereabouts, so Indiana Croft only knows the river can be found "at the bottom of the valley" and nothing else. On the slopes, there are sharp rocks.

If Indiana Croft had time, he could easily find the point where to get off the bridge to get exactly to the river, given that he has a laser pointer that he can measure heights with and tells him how many meters there are from the bridge to the ground at a certain point. Unfortunately, the priests of the Temple have already found him and they are chasing him down. If he doesn't jump off the bridge they'll catch him before he gets off the bridge. Our adventurer must quickly find the position of the river to get off and flee safely.

In order to save our hero, design the algorithm that Indiana Croft should use to find the minimum point of the valley under the conditions mentioned above. The algorithm must be efficient, for he cannot afford to waste a single second: at least in the best case it must have a logarithmic order.

You can consider the time that it takes for Indiana Croft to travel along the bridge as negligible and that the estimate of the point of the river where to drop off can have an approximation error of ε meters (ε is a given constant). Explain the reasoning behind the provided solution and analyze its efficiency and complexity.

Solution

The problem basically forces us to use Gradient Descent. Since we have to optimize each move and cannot afford to waste time on the absolute optimal of answers, we look at what happens to the slope of the function created by the heights of the bridge.

Even though recursive, the complexity of this algorithm is clearly O(logn) since each iteration, no matter what happens, the dataset is divided in half.

I also took extra time to make the implementation space-efficient as well. This means that no extra storage elements or auxiliary temporal variables are used when calculating the gradient descent, only a dataset, a start point, and an endpoint.

Lastly, I ignored the "(…) estimate of the point of the river where drop off can have an approximation error of ε meters" and chose to go directly with the lowest possible error there could be.

Implementation

```
def grad_descent(data: List[Number]) -> Number:
```

```
start = 0
end = len(data) - 1

def grad_descent_aux(data, start, end):
    # Basic cases.
    is_tuple = (end - start) <= 2
    is_increasing = data[start] < data[end]

    if is_tuple: return start if is_increasing else end

    # Not-so-basic cases.
    mid_idx = (start + end) // 2
    is_descending = data[mid_idx - 1] >= data[mid_idx]

    if is_descending: return grad_descent_aux(data, mid_idx, end)

    return grad_descent_aux(data, start, mid_idx)

    return grad_descent_aux(data, start, end)
```

Tests

In the tests we generate different heights in a range that would resemble a realistic mathematical function. In order to test the method works I order the values and then slice away those that are above the error_window. Then we check if the value we obtained as “the point to jump” is in the values that we deem acceptable.

```
@staticmethod
def generate_dataset(error_rate):
    test_case = [randint(500, 600)]

    for _ in range(randint(5, 10)): test_case.append(test_case[-1] - randint(10, 30))
    for _ in range(randint(5, 10)): test_case.append(test_case[-1] + randint(10, 30))

    # Allow for error.
    error_window = ceil(len(test_case) * error_rate)
    expected = sorted(test_case[:])
    expected = expected[:error_window]

    return test_case, expected

def test_grad_descent01(self):
    test_case, expected = self.generate_dataset(0.1)
    provided = test_case[grad_descent(test_case)]

    self.assertIn(provided, expected)
```

```
┌dave💻ec2 📁 algo 🌿master ❯ python3 -m unittest U3/test/test_e6.py
----------------------------------------------------------------------
Ran 4 tests in 0.000s | OK
```

Exercise 7

In Abeceland, a city famous for its beautiful N squares and that even you may know, they have a very curious system of roads. The way it

works is the following:

- Each square is directly connected to all other squares that begin with a letter that is in their name. For example, Ace Square has streets that go to any square whose name starts with A, C, or E. This way they would be connected to Cut Square, Coast Square or East Square, but not with Doom, Fall or Tiara Square.
- The streets are one way and they follow the rule of the letters. For example, from Ace Square you can go to Cut Square, but not the other way around, since it doesn’t comply with the rule.
- Places like Ace Square and Cat Square are connected in both directions, since they both comply with the rule of letters.

All these connections between the N squares are stored in a street map, represented by an adjacency matrix of size NxN. In this map, the value of Streets[p, q] indicates whether one can go from p to q.

April 26th, feast of San Isidoro of Seville, patron of letters and computer scientists, is coming up. The city of Abeceland has decided to celebrate it in a rather peculiar way. They play to reverse the direction of all streets that connect their city. What a wonderful idea, isn’t it? On that day one can’t move from Ace Square to Cut Square anymore, but they will be able to go the other way around, from Cut Square to Ace Square. Obviously, Ace Square and Cat Square will still remain connected to each other.

Our objective is to design a standard divide-and-conquer algorithm that, receiving the street map adjacency matrix as a parameter, returns the street map valid for the day of San Isiodoro de Seville. Analyze the complexity of the provided solution and indicate the data structures you have used.