CSE103 Introduction to Algorithms

TD1: Warm Up Using Lists

February 25, 2021

Exercise 1: square root

Consider the following problem:

SQUARE ROOT:

input:	a non-negative integer n
output:	the greatest integer k such that $k \leq \sqrt{n}$

Think of an algorithm for solving SQUARE ROOT by having only multiplication at your disposal (plus incrementing/decrementing counters). Write it informally however you want (English, pseudocode) and then implement it in Python.

Solution. Go brute force: starting from 0, try all non-negative integers until we find the first k such that $k^2 \geq n$. If the inequality is strict, we return $k-1$; otherwise, n is a perfect square and we return k.

Here's a possible implementation in Python:

```python
def NaiveSqrt(n):
    k = 0  # start with 0
    while k*k < n:  # check if k^2 goes above or is equal to n
        k += 1  # if not, try next integer
    if k*k > n:  # check if n is perfect square
        k -= 1  # if not, substract 1
    return k
```

Notice that we do not test whether `n` is non-negative. This is acceptable because:

1. the specification says nothing about what should happen in case the input is negative;
2. our function does not generate runtime errors when `n` is negative. (Can you tell what happens?)

In general, it is always good practice to make sure that our programs do not generate errors or run forever. In case the problem specification does not prescribe any particular behavior, we can return an arbitrary value (as we do here), or print an error message, or whatever we find suitable, as long as execution is guaranteed to terminate without errors, for all possible inputs, not just those covered by the specification.

Exercise 2: square root using Newton's method

If we're also allowed integer division (implemented by the operator // in Python), then we may resort to a more clever algorithm, which is a special case of *Newton's method* for finding zeros of real functions. Newton's method is an algorithm for solving the following problem:

FIND APX ZERO IN $[a, b]$:

input:	a sufficiently nice function $f : \mathbb{R} \to \mathbb{R}$ with a zero in $[a, b]$ and a positive real number ε
output:	a number $z \in [a, b]$ such that $\lvert f(z) \rvert < \varepsilon$

In the problem specification, $a < b$ are arbitrary real numbers and $[a, b]$ is the closed interval between them, and "APX" stands for "approximate": we are not asked to find an actual zero, only a zero up to ε. We will not detail what we mean by "sufficiently nice"; let us say that f must at least have a continuous second derivative in $[a, b]$.

Newton's method starts by guessing a zero, say $z_0 \in [a,b]$. If $|f(z_0)| < \varepsilon$, we are done. Otherwise, we start computing a sequence of values $z_1, z_2, z_3 \dots$ as follows:

$$z_{n+1} := z_n - \frac{f(z_n)}{f'(z_n)} \qquad n \in \mathbb{N}$$

where f' denotes the derivative of f. Each time, we check whether $|f(z_n)| < \varepsilon$ and we stop if this is the case. Under the "sufficiently nice" assumption for f, this is guaranteed to happen after a finite number of iterations.

Now, finding the square root of n within error $\varepsilon > 0$ is tantamount to solving Find apx zero in $[0,n]$ with input the function

$$f(x) := x^2 - n$$

and ε. (Do you see why?). It may be shown that, in this particular case, Newton's method may start with initial guess $z_0 := n$ and that the sequence $(z_i)_{i\in\mathbb{N}}$ defined by

$$z_{i+1} := z_i - \frac{f(z_i)}{f'(z_i)} = z_i - \frac{z_i^2 - n}{2z_i} = \frac{1}{2}\left(z_i + \frac{n}{z_i}\right)$$

is strictly decreasing. If we take all divisions above to be integer divisions, so that all z_i are integers, the sequence is initially strictly decreasing and then it either stabilizes on $\lfloor\sqrt{n}\rfloor$, which is precisely the number we wish to compute, or starts oscillating between $\lfloor\sqrt{n}\rfloor$ and $\lfloor\sqrt{n}\rfloor + 1$.

This suggests the following algorithm for solving Square root with input n: starting with $z_0 := n$, successively compute the integers z_i as above, stopping as soon as $z_i^2 \le n$.

1. Implement the above algorithm in Python.
2. Try and compare the running time of the function you wrote for Exercise 1 to that of the function you wrote for point 1 above, when they are applied to bigger and bigger powers of 10. When do you start noticing a difference?

Solution.

1. Here's a possible implementation:

```
def NewtonSqrt(n):
    k = n  # start with initial guess n
    while k*k > n:  # check if k^2 >= n
        k = (k + n // k) // 2  # if not, compute next element
    return k
```

2. On an average computer, applying `NaiveSqrt` from Exercise 1 to 10^{16} gives the answer after several seconds (well over 5), whereas `NewtonSqrt` answers instantaneously for numbers up to at least 10^{2048}. Apparently, this fast algorithm for computing the square root was already known to the Babylonians more than three thousand years ago.

Exercise 3: counting prime numbers

Recall that a positive integer is *prime* if it is strictly greater than 1 and if it is divisible only by 1 and itself. Also recall our naive algorithm for determining whether a number n is prime:

```
def isPrime(n):
    if n < 2:
        return False
    foundDivisor = False
    k = 2
    bound = NewtonSqrt(n)
    while (not foundDivisor) and (k <= bound):
        if n % k == 0:
            foundDivisor = True
        k += 1
    return not foundDivisor
```

where we are using the function `NewtonSqrt` defined in Exercise 2. Finally, recall the function $\pi(n)$ defined to be the number of prime numbers less than or equal to n.

1. Write a Python function `piBrute(n)` which computes $\pi(n)$ using the brute force algorithm, which tests the primality of each number between 2 and n (invoke the function `isPrime` defined above for this).

2. Recall the pseudocode description of the sieve of Eratosthenes:

```
def piEratosthenes(n):
    p = 0
    # initialize a row of n+1 lamp posts
    # all of which are lighted except the first two
    l = [False,False,True,True,True,...,True]
    for k in range(2,n+1):
        if l[k]:  # if lamp post k is lighted
            p = p + 1  # increment counter
            # switch off all lamp posts strict multiples of k
            for all m>k multiple of k:
                l[m] = False
    return p
```

   Turn this into actual Python code.

3. Compare the running time of `piBrute` and `piEratosthenes` when they are applied to larger and larger powers of 10 (you may use the table on

   `https://www.wikipedia.org/wiki/Prime-counting_function`

   to check that you are getting the right numbers). When do you start noticing a difference? How far can you go with respect to the Wikipedia table?

Solution.

1. Here is a possible implementation:

```
def piBrute(n):
    p = 0
    for k in range(2,n+1):
        if isPrime(k):
            p += 1
    return p
```

2. Here is a possible solution:

```
def piEratosthenes(n):
    p = 0
    # initialize a Boolean array of length n
    # containing True everywhere except for the first two positions
    l = [True] * (n+1)
    l[0] = False
    l[1] = False
    for k in range(2,n+1):
        if l[k]:  # if lamp post k is on
            p += 1  # increment counter
            # turn off all lamp posts strict multiples of k
            for m in range(2*k,n+1,k):
                l[m] = False
    return p
```

Notice that setting `l[0]` and `l[1]` to `False` before entering the loop is not necessary: the first two elements of `l` are never used, because the loop goes from the third element onward. Nevertheless, setting the first two elements to `False` has the effect that, at the end of the execution, `l[k]` is `True` iff `k` is prime. If `l` were not discarded at the end, this could be useful.

3. On an average computer, both functions return nearly instantaneously for arguments up to 10^4. From there, the running time of `piBrute` quickly worsens and it takes about 5 minutes for it to compute $\pi(10^7)$. By contrast, `piEratosthenes` returns $\pi(10^7)$ in a few seconds, and $\pi(10^8)$ in about half a minute. By allowing more time (say, up to days), one may probably use `piEratosthenes` for going about halfway down Wikipedia's table (say, up to 10^{13}). Beyond that, more sophisticated algorithms are necessary.

Exercise 4: reversing a list

Consider the following problem:

Reverse list:

input:	a list l
output:	a list containing the same elements of l, in reverse order

For example, reversing the list `[1,1,3,2,7]` gives `[7,2,3,1,1]`.

Two possible, "symmetric" algorithms for Reverse list are as follows:

- initialize an empty list r; scan l from left to right (that is, forwards), successively adding each element of l to the left (that is, at the beginning) of r; return r.
- Initialize an empty list r; scan l from right to left (that is, backwards), successively adding each element of l to the right (that is, at the end) of r; return r.

In both cases, r is built "backwards" with respect to l, so we get what we want.

1. Implement both algorithms in Python, as a function `reverseLeft` and `reverseRight`, respectively. For the first one, adding an element `a` at the beginning of a list `l` may be achieved by the instruction `l = [a] + l`. For the second one, we may use Python's list method `append`: adding an element `a` at the end of a list `l` is achieved by the instruction `l.append(a)`.

2. Initialize a list `l` containing 10^5 elements, for example using the instruction

   ```
   l = list(range(100000))
   ```

   Now run `reverseLeft(l)` and `reverseRight(l)`. Do you see any difference?

Solution.

1. Here is a possible answer, following the given indications:

```
def reverseLeft(l):
    r = []
    for i in range(len(l)):
        r = [l[i]] + r
    return r

def reverseRight(l):
    r = []
    for i in range(len(l)-1,-1,-1):
        r.append(l[i])
    return r
```

2. `reverseLeft` is much slower than `reverseRight`. This is a Python-specific phenomenon and is due to how lists are implemented in Python: adding an element at the end of a list is efficient, and the method `append` is provided for that. On the other hand, adding an element at the beginning of a list may only be done using the concatenation operator + and is *not* efficient.

   Of course, there is a simple alternative making both algorithms run equally fast in Python: rather than initialing `r` to the empty list, one may initialize it to an arbitrary list of the same length as the input `l` and then run two indices going in opposite directions, copying the elements of `l` into `r`. At this point, scanning `l` forwards or backwards is essentially the same thing, so we really have just two variants of the same algorithm. Here is an implementation of the first variant:

```
def reverseLeftAlt(l):
    n = len(l)
    r = [0] * n
    for i in range(n):
        r[n - i - 1] = l[i]
    return r
```

Exercise 5: reversing a list "in place"

The algorithms of Exercise 4 all have the effect of creating a new list, whose length is equal to that of the input list (which is left unchanged). In some circumstances, we may not want to do this: for example, because we are working under severe memory constraints (think of a microchip on a credit card), or because we are manipulating huge lists (think of a raw, uncompressed video stream). In these situations, one prefers so-called *in-place* algorithms, which act directly on the input list using a constant amount of extra space (that is, independent of the length of the input list).

In the case of reversal, we want our function to modify the input `l` so that, at the end of its execution, `l` itself is the reversed version of its original form (in that case, there will be no need for a `return` instruction at the end of the function). Moreover, we want our function not to use any list internally, otherwise we wouldn't really be working in place. For exemple, something like

```
def reverseNotReallyInPlace(l)
    l = reverseLeft(l)
```

(where `reverseLeft` is the function from Exercise 4) does *not* count as an in-place reversal: the call to `reverseLeft` creates a new list, which is copied into `l` and then discarded, but we do need memory to store it, at least temporarily.

Think of an algorithm for reversing a list in place, and implement it in Python. Once you have done that, modify it so that it solves, still in place, this slight generalization of Reverse list:

Reverse sublist:
input: A list l of length m, an integer $0 \leq s < m$ and an integer $n \geq 0$ such that $s + n < m$
output: The list l such that the sublist with indices from s to $s + n - 1$ has been reversed

In other words, we wish to reverse only a sublist of a list l, starting at the position specified by s and of length n.

Solution. The usual in-place algorithm works as follows: on input a list `l`, scan `l` up to half-length and, while scanning, swap each element with the element in "mirror position" with respect to the middle of the list.

Adapting this to solve Reverse sublist is straightforward: just add the starting position to every index. (It is also good practice to check that the starting position and the length of the substring are consistent with the input list, in order to avoid runtime errors).

Here are possible implementations:

```python
# Reverse in place
def reverse(l):
    n = len(l)
    for i in range(n // 2):
        tmp = l[i]
        l[i] = l[n-i-1]
        l[n-i-1] = tmp

# Reverse sublist in place
def revBlock(l,start,n):
    if start >= 0 and start + n//2 < len(l):
        for i in range(n//2):
            tmp = l[start+i]
            l[start+i] = l[start+n-i-1]
            l[start+n-i-1] = tmp
```

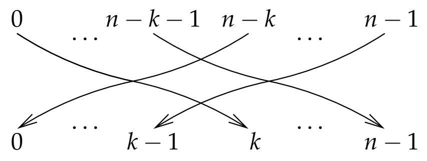
Notice how we use the built-in integer division operator `//`, which returns k in case $n = 2k + 1$ (that is, if `l` is of odd length), so we automatically avoid swapping the middle element of the list with itself.

Exercise 6: rotating a list

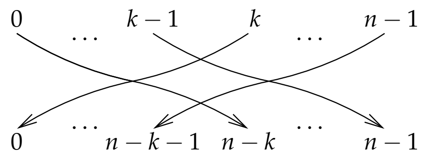
Consider the following problem:

Rotate list:
input: a list l and an integer $k \in \mathbb{Z}$
output: the list obtained from l by shifting its elements k places to the right if $k > 0$ (resp. $|k|$ places to the left if $k < 0$), considering that an element shifted out the end (resp. the beginning) of the list must reappear at the beginning (resp. the end).

We may assume that $|k| < n$, where n is the length of l (if not, just replace k with $k \bmod n$). When $k > 0$, we say that we are rotating the list *to the right* (by k), whereas when $k < 0$ we say that we are rotating it *to the left* (by $|k|$). Graphically, if $k \geq 0$, it looks like this:



Rotate right by k.



Rotate left by k.

That is, when you rotate right by k, the element at position 0 is shifted to position k, whereas when you rotate left by k, it is the element at position k which is shifted to position 0. For example, rotating the list `[0,1,2,3,4]` to the right by 2 results in the list `[3,4,0,1,2]`, whereas rotating the same list to the left by 2 results in `[2,3,4,0,1]`.

Think of an algorithm for solving Rotate list and implement it in Python. Do no worry about doing it in place (it will be the subject of the next exercise).

Solution. Here's a possible solution:

```python
def rotate(l,k):
    n = len(l)
    r = [0] * n
    for i in range(n):
        r[i] = l[(i-k) % n]
    return r
```

Notice that we use modular arithmetic in order to succinctly express that elements "falling off" one end of the list should reappear at the other end: `a % n` is the Python syntax for `a` mod `n`, the remainder of the division of `a` by `n`, which is necessarily a number between `0` and `n-1`, that is, a valid position in the input list, given that `n` is its length. Indeed, the above function works even if the absolute value of `k` is greater than the length of `l`. In that case, the result will be equivalent to rotating by `k % n`, which is what we expect.

Exercise 7: rotating a list in place

Let us think about rotating a list "in place", that is, changing directly the input l without creating a whole new list. If $k = -1$, there is a simple algorithm:

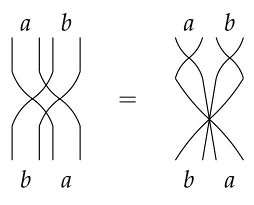
- store the first value of l in a temporary variable;
- scan l from left to right, starting from the second position, and shift every element one position to the left;
- copy the value stored in the temporary variable into the last position of the list.

1. Implement the above algorithm as a Python function `rotLInPlace` (be careful with the empty list!). Then, using a suitable modification of the algorithm, write a function `rotRInPlace` rotating a list in place one position to the right. Finally, using the above two functions, write a Python function `rotateInPlace` solving the general case of Rotate list in place.

2. An instruction of the form `a = <expression>` is called an *assignment*. If `l` is a list of length n, how many assignments are performed during the execution of `rotLInPlace(l)`? How many in total during the execution of `rotateInPlace(l)`? How does that compare with the execution of `rotate(l)`, where we denote by `rotate` the not-in-place function you wrote for Exercise 6?

3. There is an in-place algorithm solving Rotate list and performing only $3n + 2$ assignments for a list of length n, independently of the value of k. By "in place" we mean, as usual, that it does not create any auxiliary list, but only uses a constant number (that is, independent of n and k) of auxiliary variables. If you wish to try and find it on your own, stop reading here (but be warned: in spite of its striking simplicity, it's not easy at all to come up with this algorithm!).

   The idea is wonderfully simple, and is based on the following diagram:



A bit more formally: the permutation swapping two blocks a and b of a list, while preserving the order of the elements within each block, is equal to

- the permutation reversing the position of all elements within block a and within block b,
- followed by the permutation reversing the position of all elements of the list.

This is related to our current endeavor because:

- the permutation on the left hand side of the equality is exactly the one we apply when we solve ROTATE LIST, for suitable values of a and b (see Exercise 6);
- the permutations on the right hand side which reverse a given block are exactly those that we apply when we solve REVERSE SUBLIST.

Therefore, we may find an in-place algorithm for ROTATE LIST by composing instances of an in-place algorithm for REVERSE SUBLIST. But we already have such an algorithm! (Exercise 5). Using this, implement the algorithm for ROTATE LIST suggested by the above diagram, making sure that it performs the number of assignments mentioned above.

Solution.

1.
```
def rotLInPlace(l):
    n = len(l)
    if n > 0:
        tmp = l[0]
        for i in range(1,n):
            l[i-1] = l[i]
        l[n-1] = tmp

def rotRInPlace(l):
    n = len(l)
    if n > 0:
        tmp = l[n-1]
        for i in range(n-1,0,-1):
            l[i] = l[i-1]
        l[0] = tmp

def rotateInPlace(l,k):
    if k >= 0:
        for i in range(k):
            rotRInPlace(l)
    else:
        for i in range(-k):
            rotLInPlace(l)
```

2. When called on a list of length $n > 0$, the function `rotLInPlace` performs 2 assignments, then executes $n - 1$ times a `for` loop containing one assignment, then performs a final assignment. The total is therefore $n + 2$. The same holds for `rotRInPlace`. The execution of `rotateInPlace(l,k)` executes k times one of `rotLInPlace` or `rotRInPlace`; in both cases, the total number of assignments performed is $k(n + 2)$.

   By contrast, the `rotate` function given in the solution to Exercise 6 always performs 2 assignments and then enters a `for` loop containing one assignment, which is executed n times. Therefore, the total number of assignments is $n + 2$, which is independent of k. In the worst case, when k is close to n, the number of assignments performed by `rotateInPlace` grows quadratically with n, whereas it only grows linearly for `rotate`.

3. If we call `revBlock` the function solving REVERSE SUBLIST from Exercise 5, a possible solution is as follows:

```
def rotateInPlaceRev(l,k):
    n = len(l)
    if k >= 0:
        k = n - (k % n)
    else:
        k = -k % n
    revBlock(l,0,k)
    revBlock(l,k,n-k)
    revBlock(l,0,n)
```

Notice the `% n`, which takes care of the case in which the absolute value of `k` is greater than `n`.

Let `l` be of length n and let `k` be arbitrary. Let us count the number of assignments performed during the execution of `rotateInPlaceRev(l,k)`. First of all, `revBlock` performs $\frac{3}{2}m$ assignments, where m is the length of the sublist: indeed, the `for` loop it contains executes 3 assignments $\frac{m}{2}$ times. Now, independently of whether k is positive or negative, `rotateInPlaceRev(l,k)` starts by performing 2 assignments and then applies `revBlock` to a sublist of length k, a sublist of length $n - k$ (not necessarily in that order) and to the whole list. Therefore, the total number of assignments is

$$2 + \frac{3}{2}k + \frac{3}{2}(n-k) + \frac{3}{2}n = 3n + 2,$$

as desired.