CSC-1S003-EP Introduction to Algorithms

TD1: Searching, Sorting, and In-Place Algorithms

February 6, 2025

In this problem sheet you will be asked to implement the algorithms discussed in Lecture 1 and you will also be introduced to some new material: binary search, three more sorting algorithms (*insertion sort* (Exercise 3), *counting sort* (Exercise 6) and *radix sort* (Exercise 7)) as well as the notion of *stability* (Exercise 8). There will also be a couple of exercises about in-place manipulation of lists.

Like all labs for this class, this lab is not assessed. You are not requested to submit your work. There is no time limit for the exercises, you may keep working on them during the rest of the week. The answers will be posted on Moodle after the last lab session on Thu evening. Feel free to ask questions during and after the lab (via email).

Let us stress that *this class is about learning and analyzing algorithms, not Python*. To make this more conspicuous, you are *forbidden* to use Python-specific shortcuts in your code. That is, *you are requested to use only the subset of Python which, essentially, is common to any other programming language*.

More specifically, for this problem sheet you will only use:

- function declarations (`def`), `if...elif...else` statements and `while` loops, as well as `for` loops with the `range(m,n,k)` construct restricted to the case in which `m`, `n` and `k` are *non-negative* integer variables or constants (not all three of them must be present, of course);
- basic datatypes like `int`, `bool` and `str`, basic arithmetic on integers (`+,-,*,//,%`), as well as increment/decrement assignments (`+=`, `-=`), standard comparison operators (`==`, `>`, `>=`, etc.), Boolean operations (`and`,`or`,`not`);
- lists are limited to the basic functionalities:
  - creation of a list of length `n` (either variable or constant): `l = [0] * n`;
  - accessing an element of a list `l[i]` where `i` is a *non-negative* integer variable or constant;
  - concatenation (`+`), the `append` method, and list length (`len`).

The only Python-specific shortcuts you are allowed to use are:

- one-line swap instructions `a,b = b,a`;
- declarations of default values for function arguments: `def f(a,b=0):`, so that calling `f(2)` is the same as calling `f(2,0)`.

You will also be allowed to use list comprehension to succinctly initialize a random list.

Nothing else is authorized. In particular, you **may not** use:

- list slicing or any list method other than `append` (`insert`, `remove`, `pop`...);
- any other complex type (sets, dictionaries...) or any built-in function (`sum`,`max`,`log`...).

Exercise 1: binary search

Binary search in a divide and conquer algorithm which takes as input an element a and an ascendingly sorted list l, and returns True if a is in l, or False otherwise, as follows:

- if l is empty, return False;
- otherwise, compare the "middle" element $l[m]$ of l with a;
- if they are equal, return True;
- otherwise:
  - if $l[m] > a$, recursively apply the algorithm to the list $l[0], \ldots, l[m-1]$;
  - if $l[m] < a$, recursively apply the algorithm to the list $l[m+1], \ldots, l[n-1]$, where n is the length of l.

Implement binary search *in place*, that is, you are not allowed to create *any* new list. Also, instead of `True/False`, it must return a position in l where a is found, or -1 if a is not in l.

Exercise 2: merge sort

Recall the MERGE problem:

MERGE:
input: two lists l_1, l_2 sorted ascendingly;
output: the list $l_1 + l_2$ (concatenation) sorted ascendingly.

We know that there is an easy efficient algorithm for MERGE: starting from the first elements of l_1 and l_2, progressively build the result by comparing the current elements of l_1, l_2 and selecting the smallest, until no element is left in one of the two lists, at which point we just append the rest of the other list to the result.

The merge sort algorithm takes as input a list l and returns the ascendingly-sorted version of l as follows:

- if l is empty or a singleton, return l;
- otherwise, let l'_1, l'_2 be the lists resulting from recursively applying the algorithm to the left and right half of l, respectively;
- return the list obtained by merging l'_1 and l'_2.

Implement the above algorithm for MERGE and then, using it, implement merge sort. For efficiency reasons, it is better if you do *not* use recursion in the implementation of the MERGE algorithm; just use a loop instead. (You will use recursion for merge sort itself, of course)

Exercise 3: insertion sort

Merge sort is a divide and conquer algorithm: it is based on the idea that a list l of length n may be sorted by merging the sorted version of the two halves of l, of length $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. If, instead of splitting l in half, we systematically split it in the singleton containing the first element of l and the rest of the list (of length $n-1$), we obtain a recursive, non-divide-and-conquer algorithm known as *insertion sort*:

```
def insertionSort(l):
    n = len(l)
    if n <= 1:  # if empty or singleton, already sorted
        return l
    else:
        # l1 = l without first element
        l1 = [0] * (n-1)
        for i in range(1,n):
            l1[i-1] = l[i]
        # insert first element into sorted version of l1
        return insert(l[0],insertionSort(l1))
```

where `insert(a,l)` is a function taking an element `a` and an ascendingly sorted list `l`, and returning the ascendingly sorted list obtained by inserting `a` in `l` at the "right" position. That is, `insert(a,l)` has the same result as `merge([a],l)`, where `merge` implements MERGE as in Exercise 2. For example, `insert(3,[0,0,2,4,5])` returns `[0,0,2,3,4,5]`.

The interest of insertion sort is that, unlike merge sort, it is easily implementable in place and without recursion, just with loops:[1]

- let n be the length of the input list l;
- if $n \leq 1$, exit (there is nothing to do);
- otherwise, for i going from 1 to $n-1$, insert each element $l[i]$ in the sublist $l[0], \ldots, l[i-1]$. This is done in place: progressively swap the element $a := l[i]$ with the elements of $l[0], \ldots, l[i-1]$, starting from the right, until the "right" place for a is found.

[1] Another reason why insertion sort is useful is that it is employable as an *online algorithm*: the input l does not need to be completely known in advance, the list may be sorted while its elements arrive one after the other, asynchronously.

Implement the above algorithm for in-place, non-recursive insertion sort (remember that you do *not* need to `return` the sorted list; the modifications will be done directly to the input `l` and such modifications will be visible after the execution the function).

Exercise 4: quicksort with Lomuto's partitioning scheme

Lomuto's partitioning scheme is an in-place algorithm which takes a list l of length at least 2 and two positions $b < e$ within l, and returns a position $b \leq p \leq e$ while modifying l so that, at the end of the execution of the algorithm, it is partitioned as follows:

- for all $b \leq i < p$, $l[i] \leq l[p]$;
- for all $p < j \leq e$, $l[p] \leq l[j]$.

Here is the algorithm:

1. choose a pivot, place it in $l[e]$ and set $p = b$;
2. scan l from b to $e - 1$; if an element strictly smaller that the pivot is found, swap it with $l[p]$ and increment p;
3. swap $l[p]$ and $l[e]$ and return p.

Implement it and then use it to implement quicksort in place. For simplicity, take the pivot to be $l[e]$ (so we avoid the initial swap at point 1).

The recursive function implementing quicksort will be of the form `qsortRec(l,b,e)`, where `l` is the list to sort and `b` and `e` delimit the portion where we are currently working. If `p` is the value returned by the partitioning function, it will contain a recursive call to `qsortRec(l,b,p-1)` and to `qsortRec(l,p+1,e)`. The actual quicksort function will then simply be

```
def quicksort(l):
    qsortRec(l, 0, len(l)-1)
```

Exercise 5: quicksort with Hoare's partitioning scheme

Hoare's partitioning scheme is an in-place algorithm which takes a list l of length at least 2 and two positions $b < e$ within l, and returns a position $b \leq p < e$ (notice the difference with respect to Lomuto's scheme) while modifying l so that, at the end of the execution of the algorithm, it is partitioned as follows (again notice the difference with respect to Lomuto's scheme):

- for all $b \leq i \leq p$ and $p < j \leq e$, $l[i] \leq l[j]$.

Here is the algorithm:

1. choose a pivot;
2. set $i = b$, $j = e$;
3. increment i until $l[i] \geq$ pivot;
4. decrement j until $l[j] \leq$ pivot;
5. if $i \geq j$, go to 7;
6. otherwise, swap $l[i]$ and $lj]$, increment i, decrement j and go to 3;
7. set $p = j$ or $p = e - 1$ in case $j = e$, and return p.

Implement it and then use it to implement quicksort in place. You may take the pivot to be the value of the middle element, that is, `pivot = l[b + (e-b+1)//2]`.

The recursive function implementing quicksort will be of the form `qsortRec(l,b,e)`, where `l` is the list to sort and `b` and `e` delimit the portion where we are currently working. If `p` is the value returned by the partitioning function, it will contain a recursive call to `qsortRec(l,b,p)` and to `qsortRec(l,p+1,e)` (notice the difference with respect to Lomuto's scheme). The actual quicksort function will be just like for Lomuto's scheme:

```
def quicksort(l):
    qsortRec(l, 0, len(l)-1)
```

Exercise 6: counting sort

All sorting algorithms seen so far are *comparison-based*: they sort by making comparisons between elements of the list they are sorting. There are sorting algorithms that work without making any comparison. One such algorithm is *counting sort*. It works on input lists l whose values are assumed to range between 0 and $k-1$, where k is some (usually small) positive constant. Here is a description:

- compute a list c of length k such that, for all $0 \leq i < k$, $c[i]$ is the number of occurrences of i in l;
- the sorted list is now the list starting with $c[0]$ occurrences of 0, then $c[1]$ occurrences of 1, then $c[2]$ occurrences of 2 and so on until $c[k-1]$ occurrences of $k-1$.

1. Implement the above algorithm in place, that is, write the resulting list directly in l. For this implementation, fix $k = 100$.

2. Albeit it only works for lists containing integers whose size is bounded a priori (by k), counting sort has the advantage of being extremely fast. It may be useful even in the case of lists containing complex objects, of a priori unbounded size, as long as such objects have some sorting key of small size. For example, imagine having a list of records of the form

   (FirstName, LastName, Weight, Height).

   We may imagine this to be a list l whose elements are themselves lists of length 4, so that, for example, $l[i][2]$ is the weight of the i-th record in the list. Now, we may like to sort l according to *any* of the four fields: by last name, by first name, by weight or by height. Since the value of the latter two fields is bounded a priori,[2] it may be a good idea to apply counting sort for those.

   In general, suppose that we are given a list of lists l, such that the j-th position in each list within l contains the sorting key, which is an integer between 0 and $k-1$. We want to sort l ascendingly with respect to the j-th field, that is, if r is the sorted version of l, then for all positions $p \leq q$, we must have $r[p][j] \leq r[q][j]$, whereas there is no constraint on the fields other than j.

   If we want to apply counting sort to this more complex situation, we cannot simply build the sorted list r out of the counting list c. Instead, for each element $l[i]$ of the input list, we need to compute the index in r where such an element will be located. This may be done by observing that, if $l[i][j] = h$, then certainly $l[i]$ will appear in r at a position greater or equal to $c[0] + c[1] + \cdots + c[h-1]$, because all positions before that will be occupied by elements of l whose j-th field contains a value strictly smaller than h. This suggests the following algorithm, which uses two further lists of length k, besides c:[3]

   - a list b (the list of *base values*) such that $b[h]$ is the smallest position at which an element with key h may appear in r, computed as the above sum;
   - a list o (the list of *offsets*) such that $o[h]$ acts like a counter, so that whenever we meet an element of l whose key is h, we will store it in $r[b[h] + o[h]]$, and increment $o[h]$.

   Here is a description of the algorithm, which takes as input a list l containing n lists, a position j (the key field) and an integer k (the bound to the key values):

   - compute a list c of length k such that, for all $0 \leq h < k$, $c[h]$ is the number of elements $l[i]$ of l such that $l[i][j] = h$;
   - compute a list b of length k such that, for all $0 \leq h < k$, $b[h] = \sum_{i=0}^{h-1} c[i]$;
   - initialize a list o containing k zeros;
   - initialize the output list r, of length n;
   - for each $0 \leq i < n$, let $h = l[i][j]$, copy $l[i]$ in $r[b[h] + o[h]]$ and increment $o[h]$;
   - return r.

[2] Out of curiosity: the Guinness World Records apparently has it that the heaviest human being ever known weighed 442 kg, whereas the tallest human being ever known measured 272 cm...

[3] Actually, by cleverly reusing c, one may get away with only one list. But, for the purpose of understanding the algorithm, keeping three lists is conceptually simpler.

Implement the above algorithm as a function `csort(l,j,k)`. Notice that this will *not* be in place. NB: for building the list b, you are *not* allowed to use Python's list slicing and the built-in `sum` function. In fact, there's a simple way of doing this which does not need a sum function at all.

Exercise 7: radix sort

As mentioned above, counting sort is very fast, but it only works for sorting lists of small integers. Nevertheless, it may be used as the building block of another non-comparison-based algorithm, called *radix sort*. Radix sort is fully general, that is, it may be applied to lists containing integers whose value is not bounded a priori. It is also very efficient: its worst-case asymptotic complexity is the same as that of merge sort, but usually it is faster and in fact its performance may rival that of quicksort. Moreover, when the size of the integers contained in the input list l is much smaller than the length of l, radix sort performs better than any comparison-based algorithm.

Let us first consider radix sort in its general form, which is for sorting lists according to the *lexicographic order*. Two lists l_1, l_2 of comparable items and of equal length are *lexicographically ordered* as $l_1 \leq l_2$ if they are either equal, or they first differ at a position h such that $l_1[h] < l_2[h]$. The alphabetical order is an example of lexicographic order: bounded $<$ bourbon.

Radix sort uses counting sort to solve the following problem:

LEXICOGRAPHIC SORT:
input: a list l of lists of fixed length m, containing integers between 0 and $k-1$;
output: the list l lexicographically ordered, ascendingly.

The algorithm is extremely simple: repeatedly apply counting sort to l, starting with $m-1$ as the key (the "least significant field"), then $m-2$ and so on until 0 (the "most significant field"). For example, if the input list is

$$[[2,1,2],[1,1,0],[2,2,0],[0,1,1],[1,1,1]],$$

then sorting with 2 (the last field) as the key yields

$$[[1,1,0],[2,2,0],[0,1,1],[1,1,1],[2,1,2]],$$

from which, sorting with 1 (the middle field) as the key, we get

$$[[1,1,0],[0,1,1],[1,1,1],[2,1,2],[2,2,0]]$$

and then, sorting with 0 (the first field) as the key, we finally obtain

$$[[0,1,1],[1,1,0],[1,1,1],[2,1,2],[2,2,0]],$$

which is the input list sorted lexicographically. Notice that, if we interpret the elements of the inner lists as digits, and the lists as numbers (for example, $[1,1,0]$ as the number 110), and if we ignore leading zeroes, then the final list corresponds to the numerically sorted list $[11, 110, 111, 212, 220]$. This is because, after adding leading zeroes, the usual numerical order coincides with the lexicographic order on digit expansions (with respect to any base).

Implement the above algorithm via a function `lexsort(l,m,k)`, using the function `csort(l,j,k)` you wrote at the end of Exercise 6. Then, write the following functions:

- `maximum(l)`, which takes a list of integers and returns its maximum value. NB: this already exists in Python as the built-in function `max`, which you are of course *not* allowed to use; you are asked to implement it yourself.
- `declen(n)`, which takes a non-negative integer n and returns the number of digits in the decimal expansion of n (for example, `declen(107)` returns 3). NB: you are *not* allowed to use Python's built-in logarithm function for this!

- `decexp(n,m)`, which takes two non-negative integers n and m and returns a list l of length m such that $l[i]$ is the $(m-i)$-th decimal digit of n, from the right, with leading zeros if the decimal expansion of n has less than m digits (for example, `decexp(1203,3)` returns `[2,0,3]`, whereas `decexp(54,3)` returns `[0,5,4]`. NB: you are *not* allowed to use Python's integer-to-string conversion and then play with list slicing for this!
- `deccmp(l)`, which takes a list of integers between 0 and 9 and returns the number such that l is its decimal expansion (for example, `deccmp([0,1,2])` returns `12`).

(To test your implementations, you may check that, for any list of digits `l`, `decexp(deccmp(l),len(l))` returns `l` itself, and for any non-negative integer `n`, `deccmp(decexp(n,declen(n)))` returns `n` itself). Using the above functions, implement radix sort on non-negative integer lists as follows:

- find the maximum value `mx` of the input list `l`, and let `m = declen(mx);`
- using `decexp`, convert `l` into a list of lists of length `m`;
- apply `lexsort` to sort this list lexicographically, obtaining a list of lists `r`;
- convert `r` into a list of non-negative integers by applying `deccmp`;
- return `r`.

Exercise 8: stability

Consider again a list of records of the form

(FirstName, LastName, Weight, Height).

As we said in Exercise 6, we may be interested in sorting this list according to any field. Our current implementations of comparison-based algorithms (merge sort, insertion sort, quicksort) do not allow this, because they consider the records as a whole (in fact, in this situation they would sort according to lexicographic order, see point 4 of Exercise 9).

Modify each of these algorithms (merge sort, insertion sort, quicksort with both partition schemes) so that they take as input a list of lists l and an integer k, the *sorting key*, and they sort according to the value of the k-th field of the inner lists. For example, if

$$l = [[1,2,5],[0,3,6],[5,8,2],[2,1,9],[8,8,7]],$$

then on input l and with $k = 1$ the sorted list could be

$$[[2,1,9],[1,2,5],[0,3,6],[5,8,2],[8,8,7]]$$

(we are using the second field as key, and we are sorting ascendingly with respect to it). (*NB: this modification should be trivial! It should simply take copy-and-pasting your code, adding an argument* `k` *to your functions and appending* `[k]` *in a few places here and there. If you are taking too long to do this, something's not right...*).

A sorting algorithm is said to be *stable* if it preserves the original order of elements with the same sorting key. In the above example, where the sorting key is $k = 1$, the lists $[5,8,2]$ and $[8,8,7]$ have the same sorting key (the second field is 8 in both), so the list

$$[[2,1,9],[1,2,5],[0,3,6],[8,8,7],[5,8,2]]$$

is also sorted according to the second field and is a perfectly acceptable result (that is why we said "the sorted list could be" and not "should be"). However, a sorting algorithm returning the above list is *not* stable, because, in the original list, $[8,8,7]$ came *after* $[5,8,2]$, whereas now their relative order is reversed.

Stability is interesting when one wants to order relatively to a main key and a secondary key. For instance, imagine that we want to sort the above records alphabetically by last name, and then, if more people have the same last name, we want them to appear in alphabetical order with respect to their first names. We may achieve this with two sorting passes, as long as we use a stable sorting algorithm for the second pass: the first one sorts alphabetically by first name, the second by last name.

If the algorithm used for the second pass is not stable, there is no guarantee that the first-name-based ordering between records with the same last name will be respected.

Take the modified version "with key" (that you just wrote for this exercise) of merge sort, insertion sort, quicksort (both variants), as well as the function `csort` implementing counting sort "with key" that you wrote at the end of Exercise 6, and test it on a list of the form

```
l = [[random.randint(0,9),i] for i in range(20)]
```

using as sorting key $k = 0$ (the first field, which has small random values). Since the second field coincides with the position of the element in the original list, you will be able to quickly assess the stability of the result. Which algorithm seems to be stable and which one does not? Would radix sort work if `csort` were not stable?

Exercise 9: experiment!

Now that you have programmed lots of different sorting algorithms, you may test their speed! For instance, try the following:

1. use `import random` to enable invoking Python's random number generator, then generate a random list of 5000 integers in the range 0–999 with the instruction

   ```
   l = [random.randint(0,999) for _ in range(5000)]
   ```

   Apply all the different sorting algorithms to random lists like this (except counting sort, which would not work because the elements go beyond 99). You should see that all algorithms work instantaneously, except one... which one?

2. In order to test the difference between the "fast" algorithms, you'll need to time their execution. Use `import time` to enable access to the internal clock. Then, you may time the execution of a program as follows:

   ```
   start = time.time()
   # execute something
   t = time.time() - start
   print("The␣execution␣took␣%.6f␣seconds" %t)
   ```

   The last instruction will print the time up to microseconds. Now try and time the "fast" algorithms (merge sort, the two versions of quicksort, radix sort), maybe using even longer lists, but with smaller integers so we may also test counting sort:

   ```
   l = [random.randint(0,99) for _ in range(10000)]
   ```

   or even

   ```
   l = [random.randint(0,9) for _ in range(10000)]
   ```

   Who's the winner of the "contest"? Do you see any difference between the two versions of quicksort?

3. Now, instead of testing the algorithms on random lists, let's test them on "special" lists. For example, a constant list of length 2000:

   ```
   l = [5] * 2000
   ```

   or an already sorted list of the same length:

   ```
   l = [i for i in range(2000)]
   ```

   How does quicksort with Lomuto's partitioning scheme behave on such lists? Is its performance closer to merge sort or to insertion sort?

4. As mentioned in Exercise 7, the original application of radix sort is for sorting lexicographically. It happens that Python already has lexicographical ordering built in: if `l1` and `l2` are two integer lists, writing for example `l1 < l2` will test whether `l1` is strictly smaller than `l2` in the lexicographic order. Any other comparison operator may also be applied (`>`, `>=`, `<=`). This means that your Python implementation of comparison-based algorithms (merge sort, quicksort, insertion sort) automatically works for lists of integer lists, and sorts them ascendingly according to the lexicographic order. Therefore, we may compare the performance of these functions with the function `lexsort` you wrote in Exercise 7. Try executing merge sort, quicksort (with Hoare's scheme, Lomuto's will do poorly here) and `lexsort(1,3,10)` when `l` is a random list of triples

   ```
   l = [[random.randint(0,9),random.randint(0,9),random.randint(0,9)]
        for _ in range(n)]
   ```

   with `n` progressively growing from 10^3 to 10^6. What do you notice?

Exercise 10: 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. The following is a naive algorithm for determining whether a number n is prime:

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

where `isqrt` is a Python function computing the integer square root of a number (because a prime factor of n can never be bigger than $\lfloor\sqrt{n}\rfloor$).

1. The function $\pi(n)$ is defined to be the number of prime numbers less than or equal to n. Write a Python function `piBrute(n)` which computes $\pi(n)$ using a brute force algorithm, which tests the primality of each number between 2 and n (invoke the function `isPrime` defined above for this).

2. Imagine you're at the beginning of an infinite road, lit up by lamp posts numbered with the integers $2, 3, 4, \ldots$ Next to each lamp post k there's a switch; flipping it turns off all lamp posts numbered with a multiple of k, except k itself. You start walking along the road, and each time you pass by a lighted lamp post, you flip the corresponding switch. As you go along, the lighted lamp posts you'll leave behind are exactly those corresponding to the prime numbers.

   The above algorithm for constructing the list of prime numbers is called *the sieve of Eratosthenes*. Write a Python function `piEratosthenes(n)` computing $\pi(n)$ using the sieve of Eratosthenes.

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?

Exercise 11: reversing a list in place

Consider the following problem:

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

For example, if $l =$`[1,2,3,4,5,6,7]`, $s = 1$ and $n = 3$, then l must be transformed into `[1,4,3,2,6,7]`.

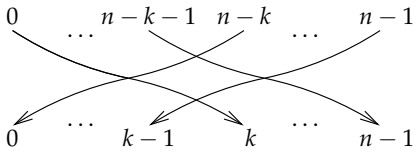
Implement in Python an algorithm for solving the above problem. Notice that we must operate "in place", that is, it is the original list l which must be modified (so it will not be necessary to have `return` at the end fo the function) and, if we use extra variables, the space they occupy must not depend on the length of l.

Exercise 12: rotating a list in place

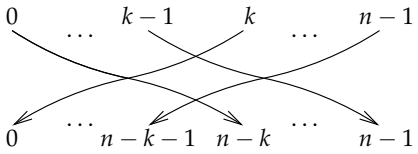
Consider the following problem:

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 mod 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]`.

We want to solve ROTATE LIST in place. If $k = -1$, there is a simple algorithm:

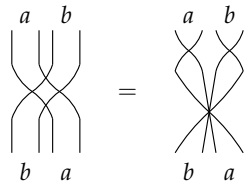
- store the first value of l in a temporary variable;
- scan l from left to right, starting from the second position, and shift every element one position to the left;
- copy the value stored in the temporary variable into the last position of the list.

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)`? (If you used Python swap instructions of the form `a,b = b,a` in your implementation, you may count them as 3 assignments).

3. Regardless of the exact number you found in answering the previous question, it will depend on both *n and k*. There is an in-place algorithm solving ROTATE LIST and performing a number of assignments which is linear in n but *independent* of k. If you wish to try and find it on your own, stop reading here, the solution is on the next page (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 12);
- 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 11). Using this, implement the algorithm for Rotate list suggested by the above diagram, making sure that it performs the number of assignments mentioned above.