CS 1110, LAB 13: SEQUENCE ALGORITHMS

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs/lab13/

First Name: ____________________ **Last Name**: ____________________ **NetID**: ______________

This final lab of the course helps you practice with more complicated invariants than you had in the last lab. In particular, it covers the sequence algorithms covered in Lecture 25. If you are having difficulty with this part of the lab, you might want to review the additional reading for that lecture.

Most of you have completed the requisite labs and so do not need to check off this lab for credit. However, we still recommend that you use this lab as a study guide for the final, regardless of whether or not you choose to complete it.

Lab Materials. We have created several Python files for this lab. You can download all of them from the Labs section of the course web page.

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs

When you are done, you should have the following two files.

- `lab13.py` (the primary module for the lab)
- `test13.py` (a unit test for `lab11.py`)

Create a *new* directory on your hard drive and download all of the files into that directory. You can also get the files bundled in a single ZIP file called `lab13.zip` from the course web page.

Note that many of the functions in `lab13.py` are optional. If you choose to skip a function, then you should comment its tests out of `test13.py` as well.

Getting Credit for the Lab. Once last time, you have a choice between getting credit through the online system or your instructor. The online lab is available at the web page

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs/lab13/

As with the last lab, you should be modifying the file `lab13.py` directly, and not typing all of your answers into the online system directly. The online lab is particularly picky about how it grades answers, so it may be best to check off your lab with a person instead.

If you absolutely need this lab for credit, then you have **until the day of the final exam** to finish this lab. You should always do your best to finish during lab hours. Remember that labs are graded on effort, not correctness.

To complete this lab you need to do **two of the four exercises in this lab**. In other words, you need to complete one of 1A or 1B and one of 2A or 2B.

Course authors: D. Gries, L. Lee, S. Marschner, W. White

Exercise 1A: Finding the Minimum of a List

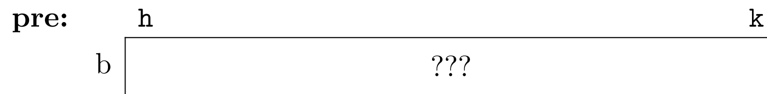
If you complete this section, you do *not* have to complete Exercise 1B.

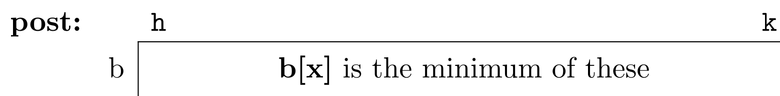
The following assertions are for an algorithm to find the minimum element of a list **b[h..k]**:

Precondition: `h <= k < len(b)`

Postcondition: **b**[**x**] is the minimum of **b**[**h..k**]

We represent each of these assertions as the pictures shown below.





There any many, many different invariants we could construct that would be compatible with these assertions. Below are two such invariants. For each invariant, draw the picture representation, and then implement the algorithm in `lab13.py`.

Invariant P1. Written in text form, this invariant is as follows.

P1: `b[x]` is the minimum of `b[w+1..k]`

Using the notation we showed in class, represent this invariant as a picture below.

The function `minpos1` in `lab13.py` uses this invariant. Implement this function accordingly.

Invariant P2. Written in text form, this invariant is as follows.

P2: `b[x]` is the minimum of `b[h..r-1]`

Using the notation we showed in class, represent this invariant as a picture below.

The function `minpos2` in `lab13.py` uses this invariant. Implement this function accordingly. To get credit for this part of the lab, show both your pictures above and your code in `lab13.py` to a consultant.

Exercise 1B: Partitioning on a Fixed Value

If you complete this section, you do *not* have to complete Exercise 1A.

The algorithms below swap the values of list b and store a value in k to make the postcondition true. List b is not sorted initially. The precondition and postcondition are as follows:

Precondition: `b[0..]` $=$? (i.e. nothing is known about the values in b)

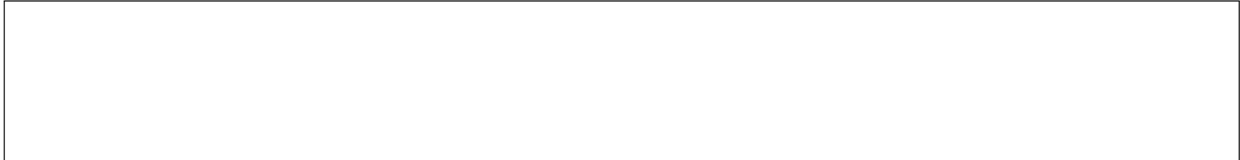
Postcondition: `b[0..k]` ≤ 6 and `b[k+1..]` > 6

Below are two different invariants. For each invariant, draw the picture representation, and then implement the algorithm in `lab13.py`.

Invariant P1. Written in text form, this invariant is as follows.

P1: `b[0..k]` ≤ 6 and `b[t..]` > 6

Using the notation we showed in class, represent this invariant as a picture below.

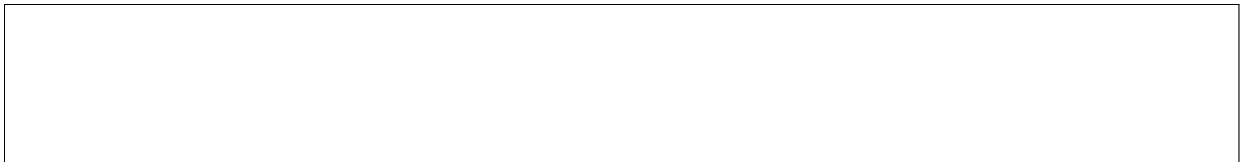


The function `fixedpartition1` in `lab13.py` uses this invariant. Implement this function accordingly, making sure to satisfy the invariant.

Invariant P2. Written in text form, this invariant is as follows.

P2: `b[0..s-1]` ≤ 6 and `b[k+1..]` > 6

Using the notation we showed in class, represent this invariant as a picture below.



The function `fixedpartition2` in `lab13.py` uses this invariant. Implement this function accordingly. To get credit for this part of the lab, show both your pictures above and your code in `lab13.py` to a consultant.

Exercise 2A: General Partitioning Algorithm

If you complete this section, you do *not* have to complete Exercise 2B.

Assume that list `b` is not necessarily sorted initially. The partition algorithm (which uses only swap operations) is designed to meet the assertions below.

Precondition: `b[h] = x` for some `x` and `h` $\leq$ `k` $<$ `len(b)`
(this is so we can talk about b[h]; `x` is not a program variable.)

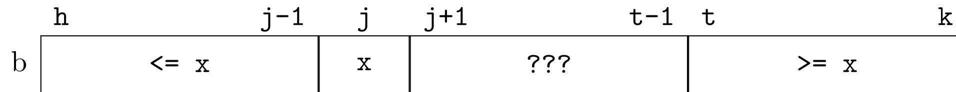
Postcondition: `b[h..j-1]` $\leq x =$ `b[j]` $\leq$ `b[j+1..k]`

Below are three different invariants. We provide the first one as an example. You are to complete the invariant and the algorithm code for the other two.

Invariant P1. Written in text form, this invariant is as follows.

P1: `b[h..j-1]` $\leq x =$ `b[j]` $\leq$ `b[t..k]`

Using the notation we showed in class, we represent this invariant as the picture below.



In addition, the code satisfying this invariant is shown below.

```
# Make invariant true at start
j = h
t = k+1

# inv:  b[h..j-1] <= x = b[j] <= b[t..k]
while j < t-1:
    if b[j+1] <= b[j]:
        swap(b,j,j+1)
        j = j + 1

    else:
        swap(b,j+1,t-1)
        t = t - 1

# post:  b[h..j-1] <= x = b[j] <= b[j+1..k]
```

Invariant P2. Written in text form, this invariant is as follows.

P2: `b[h..j-1]` $\leq x =$ `b[j]` $\leq$ `b[q+1..k]`

Using the notation we showed in class, represent this invariant as a picture below.

The function `partition2` in `lab13.py` uses this invariant. Implement this function accordingly, making sure to satisfy the invariant. You may find it easier to modify the code from invariant P1 to meet the new invariant.

Invariant P3. Written in text form, this invariant is as follows.

> **P3**: `b[h..j-1]` $\leq x =$ `b[j]` $\leq$ `b[j+1..n-1]`

Using the notation we showed in class, represent this invariant as a picture below.

The function `partition3` in `lab13.py` uses this invariant. Implement this function accordingly. You may find it easier to modify the code from invariant P1 to meet the new invariant. To get credit for this part of the lab, show both your pictures above and your code in `lab13.py` to a consultant.

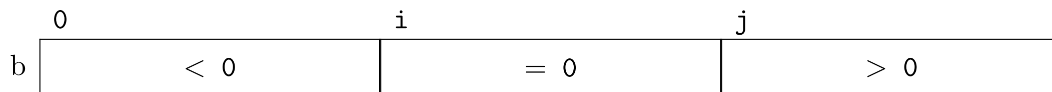
Exercise 2B: Dutch National Flag

If you complete this section, you do *not* have to complete Exercise 2B.

The last algorithm is like partition in that it groups numbers into general buckets without actually sorting them. While partition groups the numbers into two buckets (those groups ¡= and ¿= about the pivot `x`), Dutch National Flag organizes them into three buckets. Stated as text, the postcondition is

> **Postcondition**: The elements of `b[0..i-1]` are negative, the elements of `b[j..len(b)-1]` are positive, and the elements of `b[i..j-1]` are all zeroes.

The pictorial representation of this postcondition is as follows.

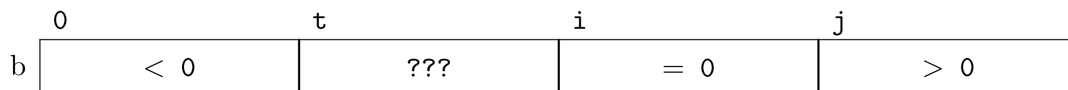


Below are three different invariants. We provide the first one as an example. You are to complete the invariant and the algorithm code for the other two.

Invariant P1. Written in text form, this invariant is as follows.

> **P1**: The elements of `b[0..t-1]` are negative, the elements of `b[j..len(b)-1]` are positive, and the elements of `b[i..j-1]` are all zeroes.

Using the notation we showed in class, we represent this invariant as the picture below.



In addition, the code satisfying this invariant is shown below.

```
# Make invariant true at start
t = 0; j = len(b); i = len(b)

# inv:  b[0..t-1] < 0, b[t..i-1] unknown, b[i..j-1] = 0, and b[j..]  > 0
while t < i:
    if b[i-1] < 0:
        swap(b,i-1,t)
        t= t+1
    elif b[i-1] == 0:
        i= i-1
    else:
        swap(b,i-1,j-1)
        i= i-1; j= j-1

# post:  b[0..i-1] < 0, b[i..j-1] = 0, and b[j..]  > 0
```

Invariant P2. Written in text form, this invariant is as follows.

> **P2**: The elements of `b[0..i-1]` are negative, the elements of `b[j..len(b)-1]` are positive, and the elements of `b[i..t-1]` are all zeroes.

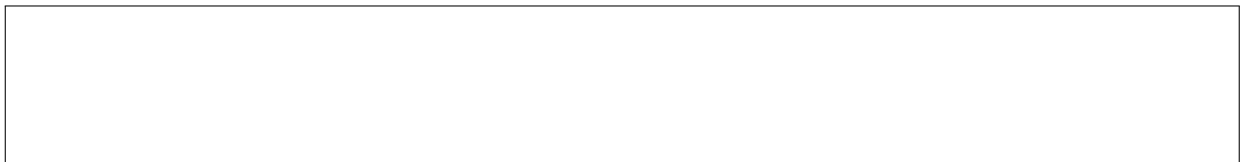
Using the notation we showed in class, represent this invariant as a picture below.

The function `dnf2` in `lab13.py` uses this invariant. Implement this function accordingly, making sure to satisfy the invariant. You may find it easier to modify the code from invariant P1 to meet the new invariant.

Invariant P3. Written in text form, this invariant is as follows.

> **P3**: The elements of `b[0..s]` are negative, the elements of `b[j..len(b)-1]` are positive, and the elements of `b[i..j-1]` are all zeroes.

Using the notation we showed in class, represent this invariant as a picture below.



The function `dnf3` in `lab13.py` uses this invariant. Implement this function accordingly. You may find it easier to modify the code from invariant P1 to meet the new invariant. To get credit for this part of the lab, show both your pictures above and your code in `lab13.py` to a consultant.