CSE101

DUE APR. 28

23:59 KST

ASSIGNMENT III

PROFESSOR

FRANCOIS

RAMEAU

SEARCH

General instructions

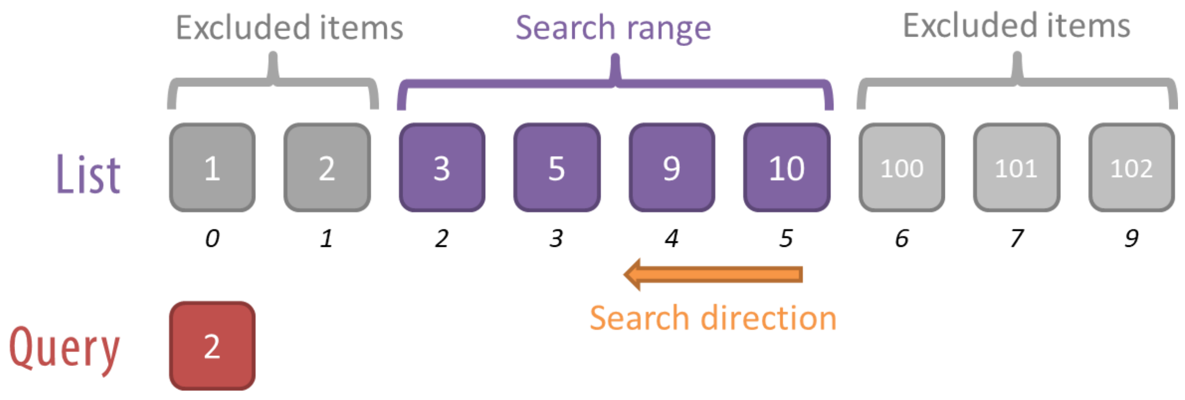
- Please add your name and email in the dedicated location at the top of your code
- Do not use any external library, except when explicitly requested
- Try to follow the naming convention proposed by PEP-8 seen in class
- Use ONLY what we have already seen in class. If not, you will get ZERO points
- Use meaningful names for your variables and functions
- If you face problem submitting your code via GitHub please contact the professor and the TA by email
- Note that the received code will be tested on a classifier to detect potential usage of Large Language Model. We will also pay particular attention to plagiarism
- Leave comments in your code to explain your code and describe the difficulties you faced

INVITATION LINK

https://classroom.github.com/a/fkawF0D5

Exercise 1: Backward linear search (3 points)

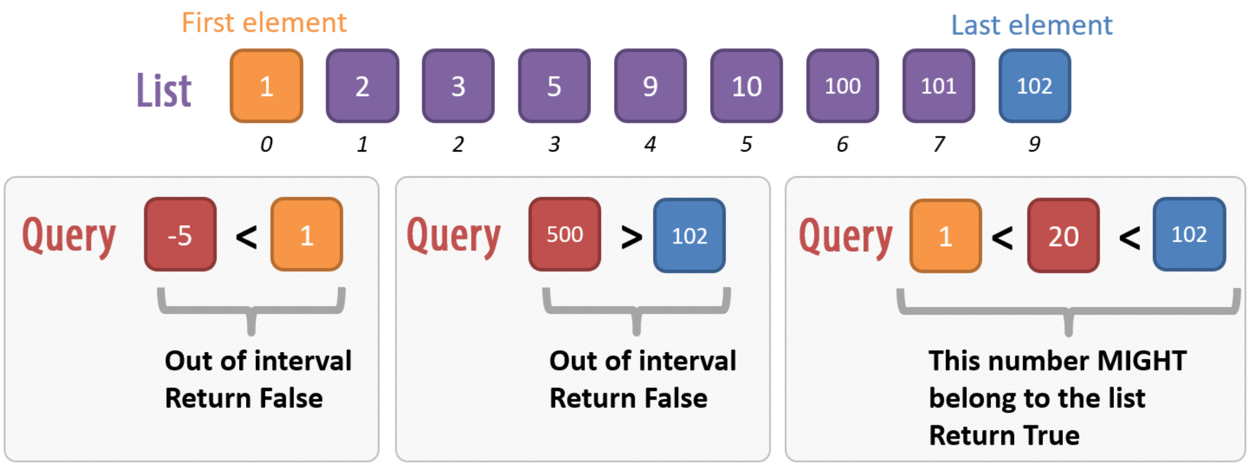
In this exercise, you will write a linear search function, similar to what you have seen in class. However, instead of initiating the search from the beginning of the list and scanning through all the elements, you will start from the end of the list and examine each element moving backward until you reach the beginning. This function, named `backwardLinearSearch()`, will also accept two additional arguments called `max_index and min_index`, which define the range within which the search will be performed. For example, if `max_index` is set to 5 and `min_index to 2`, only the elements in this range will be considered in the search. Refer to the provided illustration for further clarification.
If the element is not in the list, return None. You will use a FOR loop to implement this function.



Exercise 2: Check interval validity (2 points)

Assuming you are working with a sorted list, we aim to implement a quick check with O(1) time complexity to verify if a number can belong to the list of interest.

Since the list is sorted, if the query value is smaller than the first element, it cannot be present within the list. Likewise, if the query is larger than the last value of the list, it is also impossible for the query number to be a part of the list. Create a function called `checkIntervalValidity()` that takes two arguments: 1) the list you want to examine, and 2) the query value. The function should return True or False based on whether the query value could potentially be present in the sorted list.



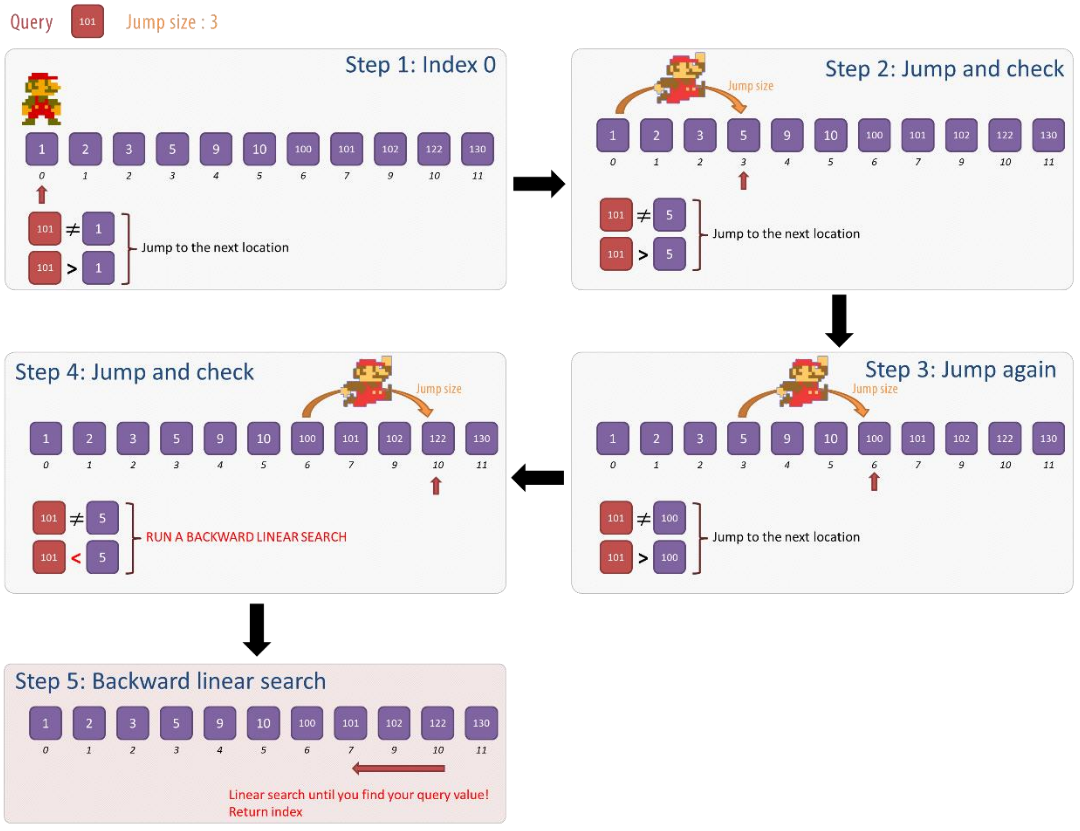
Exercise 3: Jump Search (5 points)

Objective:

In this exercise, you will implement another type of searching algorithm that works on sorted lists called **jump search** (also known as block search). It is an improvement over the linear search, where the algorithm checks every element one by one. Jump Search, as the name suggests, "jumps" through the list to find the desired element more quickly. To implement this strategy, we will use the functions we have developed in exercise 1 and 2. You will implement this algorithm in the function `jumpSearch()` which returns the index of the retrieved item or None if the item is not in the list.

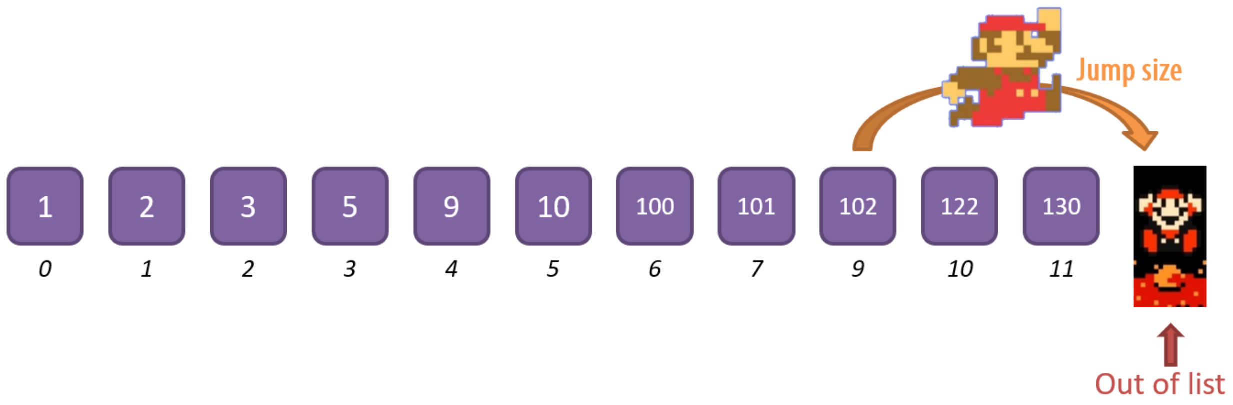
Let’s look at each stage of jump search:

1. **Check if your number is potentially in the list (1 point):** Use the function `checkIntervalValidity()` you have developed earlier to check if your query can potentially be in the sorted list. If it is impossible for your number to be in the list, then return None
2. **Determine the jump size (1 point):** You will complete the helper function called `computeJumpSize()` to determine the jump size (step size) that you will use in jump search. This step size is easy to calculate as it is only the square root of the number of elements in the list: $jump\_size = \sqrt{N}$, where N is the number of elements in your list. Your function `computeJumpSize()` will take the list as a single input and return the jump size as an integer. Make sure this value is indeed an integer as we will use it to iterate through a list!
3. **Start at the beginning:** Start at the first element of the list (index 0).
4. **Jump forward (2 points):** Jump forward by the jump size calculated in **step 2**. You will use a while loop to perform this jump. Compare the element at this index with the query value.
    a. If the element is equal to the query value, the search is successful, and the index is returned.
    b. If the element is less than the query value, repeat **step 4**.
    c. If the element is greater than the target value or the end of the list is reached, proceed to **step 5**.
5. **Perform a linear search (1 point):** Perform a linear search in the previous block (between the current index and the index jump size positions back) to find the target value. For this purpose you will use the function you have already created before: `backwardLinearSearch()`.
    a. If the target value is found, return the index.
    b. If the target value is not found, return None, indicating that the value is not present in the list.



!!! Warning !!!

Considering the previous illustration, imagine that we were looking for the value **122? You will jump outside the list and it will create an error. One trick could be the following, if the index is outside of the list then force it to the last index**


Jump size
1
2
3
5
9
10
100
101
102
122
130
0
1
2
3
4
5
6
7
9
10
11
Out of list