Solution Review: Rearrange Sorted List in Max/Min Form

This lesson gives a solution to the challenge in the previous lesson.

We'll cover the following

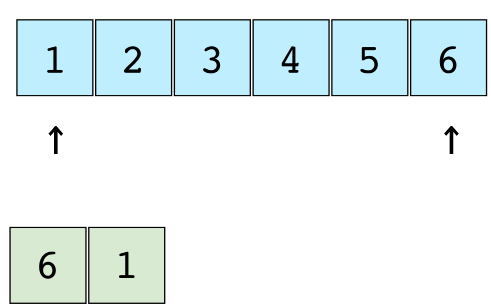
Solution #1: Creating a new list#

In this solution, we first create a new empty list that we will append the appropriate elements to and return. We then iterate through the list starting from the $0th$ index till the middle of the list indexed as `lst[length(list)/2]`. So if the length of the given list is 10, the iterator variable `i` on line 4 in our solution would start from 0 and end at $10/2 = 5$. Note that the starting index `0` in the example is inclusive, and the ending index `5` is exclusive. At each iteration, we first append the largest unappended element and then the smallest. So in the first iteration, `i = 0` and `lst[-(0+1)] = lst[-1]` corresponds to the last element of the list, which is also the largest. So the largest element in the list is appended to `result` first, and then the current or element indexed by `i` is appended.

Next, the second largest and the second smallest are appended and until the end of the list.

```python
def max_min(lst):
    result = []
    # iterate half list
    for i in range(len(lst)//2):
        # Append corresponding last element
        result.append(lst[-(i+1)])
        # append current element
        result.append(lst[i])
    if len(lst) % 2 == 1:
        # if middle value then append
        result.append(lst[len(lst)//2])
    return result


print(max_min([1, 2, 3, 4, 5, 6]))
```



1 of 3

Time Complexity#

The time complexity of this problem is $O(n)$ as the list is iterated over once.

Solution #2: Using $O(1)$ Extra Space#

```
def max_min(lst):
    # Return empty list for empty list
    if (len(lst) is 0):
        return []

    maxIdx = len(lst) - 1  # max index
    minIdx = 0  # first index
    maxElem = lst[-1] + 1  # Max element
    # traverse the list
    for i in range(len(lst)):
        # even number means max element to append
        if i % 2 == 0:
            lst[i] += (lst[maxIdx] % maxElem) * maxElem
            maxIdx -= 1
        # odd number means min number
        else:
            lst[i] += (lst[minIdx] % maxElem) * maxElem
            minIdx += 1

    for i in range(len(lst)):
        lst[i] = lst[i] // maxElem
    return lst


print(max_min([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

This solution exploits the properties of the **modulus** operator to store **two** elements at **one** index. Let's take an example list `[1, 2, 3, 4, 5, 6]`. The maximum element is 6, and if we increment it by 1, we get 7. If we were to apply the modulus 7 to, let's say, the element at index 0, we would get the same element back, i.e., 1.

The other important characteristic is that we can add multip 7 e elements, and we will still get back the original values by applying the **modulus** operator.

Let's consider `lst[0]` as an example. In the max/min ordering, we need to store 6 at index 0 in the list, since that is the maximum value in the list. Multiply 6 with 7 and add `lst[0]` to it, we get $7 * 6 + 1 = 43$. For our last trick, when we apply 43 $modulo$ 7, we get back the original 1. At the same time, if we $divide$ 43 by 7, we get back 6.

We achieve this behavior with the following line of code,

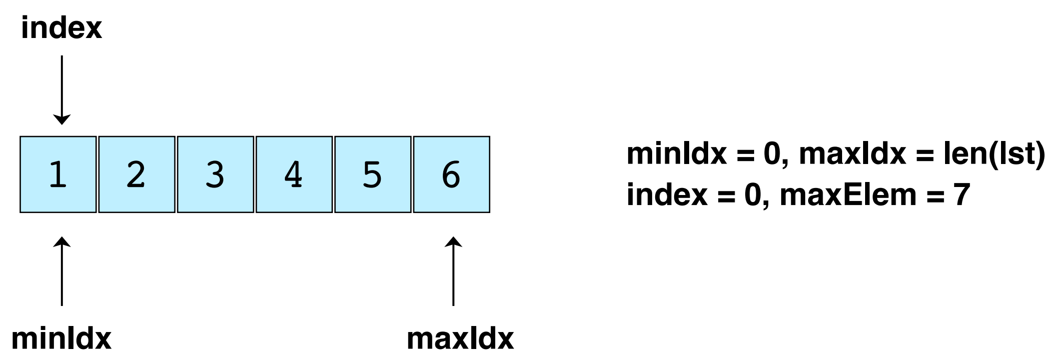
```
lst[i] += (lst[maxIdx] % maxElem) * maxElem
```

`lst[maxIdx]` is stored as a multiplier and `lst[i]` is stored as remainder. Taking the same example from above, in the list, `[1, 2, 3, 4, 5, 6]`, the `maxElem` is $6 + 1 = 7$ and 43 is stored at index 0. Once we have 43, we can get the new element 6 using $43/7$. Also, we can go back to the original element, 1, using the expression $43\%7$.

Similarly, we use the following line of code for **odd** indexes,

```
lst[i] += (lst[minIdx] % maxElem) * maxElem
```

Review the rest of the iterations below,



Initial setup

1 of 15

This allows us to swap the numbers in place without using any extra space. To get the final list, we simply divide each element by `maxElem` as done in the last for loop.

> Note: This approach only works for non-negative numbers!

Time Complexity#

The time complexity of this solution is in $O(n)$. The space complexity is constant.

Back

Challenge 10: Rearrange Sorted List i...

Next

Challenge 11: Maximum Sum Sublist

Completed

Report an Issue