Answer Key for Iteration and Decomposition (Week 4)

Tutorial Questions

1. a) 1 for linear search, 4 for binary search
   b) 4 for linear search, 2 for binary search
   c) 18 for linear search, 5 for binary search

2. Modified linear search algorithms

   a) Number of occurrences:

```
def search(a, k):
  count = 0
  i = 0
  while i < len(a):
    if a[i] == k:
      count += 1  # <-- don't return here!
    i = i + 1
  return count
```

   b) Second occurrence:

```
def search(a, k):
  count = 0
  i = 0
  while i < len(a):
    if a[i] == k:
      count += 1

    if count == 2:
      return i  # <-- only if this is the 2nd occurrence
    i = i + 1
  return None
```

   c) Last occurrence:

```
def search(a, k):
  i = 0
  last_index = None
  while i < len(a):
    if a[i] == k:
      last_index = i  # <-- keep track of last index where k is found
    i = i + 1
  return last_index
```

   or:

```
# better algo!
def search(a, k):
  i = len(a)-1
  while i >= 0:  # <-- count backwards
    if a[i] == k:
      return i
    i = i - 1
```

```
        return None
```

3. a) `while (position > 0) and value > list[position-1]  # line 9`
   b) O(N) when originally already sorted in descending order
   c) O(N^2) when originally already sorted in ascending order
   See https://www.youtube.com/watch?v=ZkwWKB88TzY&list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&index=36&t=8s

4. a) x = [34, 80, 88, 96, 47, 52, 65, 73]
   b) x = [18, 34, 45, 48, 62, 64, 77, 88, 17, 82, 87]
   c) x = [27, 91, 92, 93, 22, 31, 35]
   See https://www.youtube.com/watch?v=W2qF43I79Jk&list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&index=36

5. a) Originally, before shifting, the largest element is at index (**n**-1). After shifting right by **k**, the new index of this largest element is now (**n**-1)+**k**. However, this is larger than **n**, so we circle back to the front, which is the same as taking the mod of **n**, i.e., ((**n**-1)+**k**)%**n**. Therefore, if the value of **k** is known, we should simply return the value with index ((**n**-1)+**k**)%**n** in the array. This operation is O(1).
   See a) https://www.youtube.com/watch?v=z6ZpcfCfXAg&list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&index=37

   Small note:
   ((n-1)+k)%n = (k-1 + n)%n
   = (k-1)%n
   You can use this simpler expression in your solution.

   b) We should proceed with "linear search"-like algorithm, inspecting the first element, the second element, and so on. We know we find the maximum value when we find an element that is larger than the number before it, and the number after it.
   See b) https://youtu.be/z6ZpcfCfXAg?list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&t=481

   c) We should proceed with "binary search"-like algorithm, inspecting the middle element. If the middle element is bigger than the element immediately to its right, then you have found your answer. If not, you can either discard the left or right half of the array. We should keep the left half if the middle element is smaller than the first element (at **lower**). We should keep the right half if the middle element is larger than the first element (at **lower**).
   See c) https://youtu.be/z6ZpcfCfXAg?list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&t=641

6. 1st algo: Brute force algorithm. Search for 1 in **B** (using linear search). Then search for 2 in **B** (using linear search), then search for 3 in **B** etc etc… Time complexity = *O(n^2)*
   See: https://www.youtube.com/watch?v=vDXhFwtMOz8&list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&index=38

   2nd algo: Sort the list first → O(n log n) if using merge sort. Then go down the list and look for the "gap" → O(n). Time complexity = O(n log n) + O(n) = *O(n log n)*.
   See: https://youtu.be/vDXhFwtMOz8?list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&t=236

   3rd algo: sum all numbers in list → *O(n)*. The difference between the expected sum (1 + 2 + 3 +… + (n+1)) and the actual sum indicates the missing number.
   See: https://youtu.be/vDXhFwtMOz8?list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&t=525

4th algo: use a corresponding "flag array" with n elements ([0, 0, 0, … 0]). Go down the list and set the corresponding value in the flag array → *O(n)*.
See: https://youtu.be/vDXhFwtMOz8?list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&t=704

Extra Practice Questions

7. a) insertion sort (insertion sort: only 7 comparisons, merge sort: 12 comparisons)
   b) merge sort (insertion sort: 20 comparisons, merge sort: only 17 comparisons)
   c) merge sort

8. a) O(N log N)
   b) O(N log N)
   c) Fewer comparisons required if array is already originally sorted.

9. a) `x = [2, 25, 45, 48, 51, 65, 67, 82, 36]`
   b) `x = [8, 12, 22, 25, 54, 55, 89, 95, 18, 29, 59, 75, 79]`
   c) `x = [15, 21, 42, 43, 69, 79, 80, 95, 11, 33, 52, 60, 67, 73, 81]`

10. No.
    For N-ary search on a list of **n** elements, as N becomes bigger and bigger, the number of levels decreases to $\log_N \mathbf{n}$, but the number of comparisons required at each level becomes bigger (N-1 comparisons in the worst case). So, when N = **n**, there is only 1 level, but the number of comparisons required at that level is **n**, thus making this a linear search (with complexity of O(n)).

?-ary search	no. of levels	max no. of "mid points" to compare at each level
2 (binary)	$\log_2 n$	1
3 (ternary)	$\log_3 n$	2
4	$\log_4 n$	3
5	$\log_5 n$	4
…	…	…
n	$\log_n n = 1$	n-1. This becomes linear search

11. To make sure that each pair of numbers sum as little as possible, we should first sort the numbers in increasing order, followed by pairing the smallest number with the largest number, the second smallest number with the second largest number, and so on.

    For example, given [3, 1, 5, 4, 2, 6], we should first sort it into [1, 2, 3, 4, 5, 6], then create the following pairs: (1, 6), (2, 5), (3, 4). This will result in the smallest maximum.

    The algorithm should then proceed in terms of sorting (which is $O(N^2)$ for insertion sort, or O(N log N) for mergesort), followed by pairing (which is O(N)).

12. This is a k-way merge problem.

    1st algo:
    1. create an output array of size n*k,
    2. copy all elements over → O(nk).
    3. Sort the output array using merge sort → O(nk log nk). Note: number of elements to be sorted is nk, and not n.

    Overall complexity = O(nk log nk) since step 3 is dominant

    2nd algo: think along the lines of merge sort. There are $\log_2 k$ levels. At each level there are n*k moves (appends). Total number of steps = nk * $\log_2 k$ → O(nk * log k)

    3rd algo:
    1. Set a pointer for each of the k arrays (we will need k pointers), & initialize them to point to the first element of each array.
    2. Compare the elements at all k pointers and select the smallest; shift the relevant pointer once the smallest is selected.

    There are at most (k-1) comparisons in order to select one element to be placed into the final sorted list (except for the last k elements, which will have fewer comparisons – for example, the last element to be inserted into the final sorted list doesn't need to be compared).
    Since there are kn elements in the final sorted list, the total number of comparisons will be (k-1)*kn in the worst case. Complexity = $O(nk^2)$.

    algo #2 is the best.

~End