CSci 242: Algorithms and Data Structures **Spring, 2022**

Instructor: Dr. M. E. Kim Date: March 15, 2022

Due: by the end of the day, March 26 (Sat.), 2022.

Read the submission instructions and comply with them; otherwise, a penalty will be applied.

**Home Assignment 6: 70 pts. + 70 (implementation) + 30 (optional)**

\*Done Q1. [10] **Quick Sort**

Suppose the quick-sort algorithm is modified so that the pivot is always chosen at index ⎣*n*/2⎦, i.e. an element in the middle of the sequence. What is the running time of this modified quicksort on a sequence that is *already sorted*? Explain your answer.

* *Because the work done on each level of the quick sort halving methods is 2^i \* (n/2^i -1) each split costs log(n) time and because there are potentially many layers nlog(n) is the running time of the quick sort algorithm that has its pivot point set to the median of the sorted sequence.*

\*Done Q2. [10] **Ordering**

Suppose we’re given a sequence S of *n* elements, each of which is colored red, green, or blue.

Assume S is represented as an array S[1 .. *n*]. Give an in-place algorithm, **RGB(S, *n*)**, in a pseudo code for ordering S so that ***all the red elements are listed before all the green ones while all the blue ones are listed before red elements***.

Algorithm **partition**(S , low, high)

i ← low – 1

pivot ← S[high]

**for** j in range(low to high)

**if** S[j] <= pivot

i increased by 1

S[j] ↔ S[i] \*values at position i and position j swap

S[i + 1] ↔ S[High] \*high is assigned the value of i +1 of S

**Return** i + 1

Algorithm **RGB**(S, low ← 0, high ← NULL)

**if** high = NULL

high ← length(S) – 1

**if** low < high

partition index ← Algorithm **partition**(S, low, high)

Algorithm **RGB**(S, low , partition index - 1)

Algorithm **RGB**(S, partition index + 1, high)

\*Done Q2B. [10, optional] Implement **RGB(S, *n*)** algorithm in Python for the sequence S[1:15] = [G, B, R, G, R, B, G, B, R, B, G, R, G, R, B], and print the final sequence S.

\*Done Q3. [10] **Inversion**

Let S be an array of *n* elements on which a total order relation is defined.

An inversion in S is a pair of indices *i* and *j* such that *i* < *j* but S[*i*] > S[*j*].

Write a **Count-Inversion(S)** algorithm in a pseudo code that runs in **O(*n* log *n* )** time for determining the number of inversions in S[1: *n*].

Hint: Try to modify the merge-sort algorithm to design an algorithm.

*\*assuming that array of n elements is divided by floor[n/2] and each side is sorted*

Algorithm merge\_sort(S, temp\_array, left, right)

Inversions ← 0

if left < right

mid ← (left + right) / 2

inversions ← inversions + merge\_sort(S, temp\_array, left, mid)

inversions ← inversions + merge\_sort(S, temp\_array, mid + 1, right)

inversions ← inversions + merge(S, temp\_array, left, mid, right)

return inversions

Algorithm merge(S, temp\_array, left, mid, right)

i, k ← left

j ← mid + 1

inversions ← 0

while i is <= mid and j <= right

if S[i] (S at position i) <= S[j] (S at position j)

temp\_array[k] ← S[i]

k increased by 1

i increased by 1

else

temp\_array[k] ← S[j]

inversions ← inversions + (mid – i + 1)

k increased by 1

j increased by 1

while i <= mid

temp\_array ← S[i]

k increased by 1

i increased by 1

while j <= right

temp\_array[k] ← S[j]

k increased by 1

j increased by 1

for l in range(left to right + 1)

array[l] ← temp\_array[l]

return inversions

\*Done Q3B. [10, optional] Implement **Count-Inversion(S)** in Python for S[1:10]=[95, 80, 60, 40, 20, 10, 90, 70, 50, 30], and print the output.

\*Done Q4. [10] **In-Place** **Quick-Selection**

Give the ***in-place quick-select(A)*** algorithm that selects the *kt*h ***largest*** element in the array A of *n* elements in a pseudo code. i.e., no use of the external space for L, E, and G but every operation is performed in the array A only.

Algorithm partition(S, l, r)

x ← S[r]

i ← l

for j in range(l to r)

if S[j] >= x

S[i] ↔ S[j]

i increased by 1

S[i] ↔S[r]

Return i

Algorithm kLargest(S, l, r, k)

if k > 0 and k <= (r-1) + 1

index ← partition(S, l, r)

if (index - 1 = k – 1)

return[index value]

if (index -1 > k – 1)

return kLargest(S, 1, index -1, k)

return kLargest(S, index + 1, r, (k – index + 1) -1)

\*Done Q4B. [20] Implement an ***In-Place Quick Selection(A)*** in Python for A[1:15] = [90, 70, 60, 40, 20, 50, 60, 90, 20, 30, 90, 70, 50, 30, 90]. Print the ***median*** element, the **1st, 3rd, 5th, 10th, 13th** and the**15th *largest*** elements.

\*Done Q5. [10] **Mode**

A mode is an element that occurs most frequently in an array. Given an array, A of *n* elements, A[1:*n*], write an algorithm, **Mode(A)**, in a pseudo code that runs in **O(*n*)** time for finding the ***mode***.

Algorithm mode(A)

mxCount ← (occurrences ← 0, number ← 0)

for number in A

occurrences ← array(count occurrences or each number in A)

if occurrences > mxCount[0] \*the value of occurrences in mxCount

mxCount ← (occurrences, number)

return mxCount

\*Done Q5B. [10, optional] Implement **Mode(A)** in Python in A[1:15]=[90, 70, 60, 40, 20, 50, 60, 90, 20, 30, 90, 70, 50, 30, 90]

\*Done Q6. [10] What does the weighted median algorithm return if the weights of all the elements are equal? Explain your answer.

*If all values have the same weight, then a weighted median algorithm eventually reduces down to the median of the data set provided.*

\*Skip Q7. [10] Suppose you are given an integer ***c*** and an array, A, indexed from 1 to *n*, of *n* integers in the range from 0 to 3*n* (possibly with duplicates). i.e. 0 ≤ A[*i* ] ≤ 3*n ∀ I = {1, .., n}.*

Write an efficient algorithm that runs in **O(*n*)** time in a pseudo-code for determining if there are two integers, A[*i*] and A[*j*], in A whose sum is ***c***, i.e. ***c*** = A[*i*] + A[*j*], for 1 ≤ *i* < *j* ≤ *n*. Your algorithm has to return a *set of any pair of those indices (i , j).* If there were no such integers, return (0, 0).

**Q8 – Q9: Implementation in Python**

In the array A[1, .., 15] = [35, 65,85, 25,55, 15, 90, 40, 10, 60, 80, 70, 50, 20, 30],

\*Done Q8. [20] Sort the array A in descending order by ***Merge Sort*** and print the final array A.

\*Done Q9. [30] Sort the array A in ascending order by ***Quick Sort*** and print the final array A. To choose a pivot, find the median of the elements by a ***deterministic Selection***. Then, use the median as a pivot for Quick Sort.