CSci 242: Algorithms and Data Structures  **Spring, 2020**

Instructor: Dr. M. E. Kim Date: March 13, 2020

Due: by the end of day, March 23rd (Mon.), 2020.

**Home Assignment 5: 110/60 points + 90 (implementation)**

Q1. [10/10] **Quick Sort**

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

Runtime is O(N log N)

This is because the first level divide the input into 2 parts and each part into halves again thus meaning O(n) (2, 4, 8, etc). At each level there is at most N comparison. If the pivot yields 2 equal parts, then there will be N levels thus N \* Log N comparisons.

Q2. [3/10] **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 ~~green~~ **red** elements.

**Algorithm** Ordering

Input: Array S, n elements

pos\_red = N-1

i = 0

while (i < pos\_red) do

if( color.A[i] = ‘Red’) then

swap(A[i], A[pos\_red])

pos\_red = pos\_red - 1

else

i = i+1

endif

endwhile

Have to separate blue and green as well, a correct solution is given below:

**Algorithm Blue-Red-Green(S, *n*)  
Input:** a sequence **S** of *n* elements, each colored green, red, or blue, held in an array indexed from 1 to *n*  
**Output:** the sequence **S**, ordered so that all blue elements are listed before red and all red elements are listed before green

r ← 1 //tracks first red element

g ← *n* //tracks first green element

p ← 1 //tracks last red element

**while** p < g

**if** S[p] **=** red **then** p ← p + 1

**else if** S[p] =blue **then**

swap S[p] and S[r]

r ← r + 1

p ← p + 1

**else** //S[p] = green

swap S[p] and S[g]

g ← g - 1

(return S)

Q3. [3/10] **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 an Count-inversion(S) algorithm that runs in O(*n* log *n* ) time for determining the number of inversions in S. The beginning index in S is 1, i.e. S[1 .. *n*].

Hint: Try to modify the merge-sort algorithm to solve it.

**Algorithm** countInversion

Input: Array S

Int swap = 0;

mergedSize = k – i + 1

mergePos = 0

j= 0

i= 0

Int [ ] = leftPos

Int [ ] = rightPos

mergeArray = new int[mergedSize]

while (i<leftPos.length and j<=rightPos.length) do

if (leftPos[i] <= right[j]) then

mergeArray[k++] = leftPos[i++]

swap ++

Else

mergeArray[k++] = rightPos[j++]

swap ++

Endif

Endwhile

While (i<leftPos.length) do

mergeArray[k++] = leftPos[i++]

endwhile

While (i<rightPos.length) do

mergeArray[k++] = rightPos[j++]

endwhile

return swap

-need to partition the array for recursive calls

-swap count only needs to update when leftPos[i] > right[j]

-the update amount is not 1

Correct algorithm shown below:

**CountInversion(S)** // Count # of inversion during MergeSort.

If S.size() ≤ 1 return (S, 0)

(S1, S2) 🡨 partition(S, *n*/2)

(S1, L) ← **CountInversion**(S1) // S1 is sorted array and L is the # of inversion in S1 during sorting.

(S2, R) ← **CountInversion**(S2) // S2 is sorted array and R is the # of inversion in S2 during sorting.

(S, Total) ← **MergeAndCountI**(S1, S2, S) // S is sorted array through Merge and

// Total is the # of inversion between S1 and S2, being counted during Merge.

Return S, L + R + Total

**MergeAndCountI (S1, S2, S)**

Total ← 0 // # of inversion

*i* 🡨 1; *j* 🡨 1;

*n1* 🡨 S1.size(); *n2* 🡨 S2.size();

while*i* ≤ *n1* and *j* ≤ *n2* do {

If S1[*i*] > S2[*j*] then S[*i*+*j*-1] 🡨 S2[*j*]

*j* 🡨 *j*+1,

**Total ← Total + (*n1*- *i* + 1)** }

// all S1[*i .. n1*] including the current S1[*i*] are > S2[*j*].

else S[*i*+*j-*1] 🡨 S1[*i*]

*i* 🡨 *i* + 1 }

while *i* ≤ *n1* { // where *j* = *n2*+1

S[*i*+*j*-1] 🡨 S1[*i*]

*i* 🡨 *i* + 1 }

while *j* ≤ *n2*  { // where *i* = *n1*+1

S[*i*+*j*-1] 🡨 S2[*j*]

*j* 🡨 *j* + 1 }

return S, Total // S is sorted in the ascending order and Total is the # of inversion between S1 and S2.

Q4. [1/10] **In-Place** **Quick-Selection**

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

**Algorithm** largestElement

Input: Array A, k

Output: kth largest element

For(i=0; i< A.length; i++) do

If (A[i] > A[k]) then

Largest = i

Endif

endfor

Return A[largest]

This would return the largest element of the array not k-th largest. Solution given below:

// For the partition, any algorithm of Blue-Red-Green in Q2B may be used.

// i.e. Blue < pivot, Red = pivot, Green > pivot.

**Algorithm In-Place-Quick-Selection(A, *a, b, k*): // A[*a .. b*]**

**Input:** An array A[*a* .. *b*] of *n* comparable elements, *k* ∈[1, *n*]

**Output:** The *k*th largest element of A.

**If** *b=a* **then return** A[*a*] // # of element = 1

(pivot ← A[*b*] // or choose a pivot randomly, then swap it with the last element in A[*b*].

// can be executed in Partition.

e, g ← Partition(A, 1, n) // A[*a* .. e-1] < pivot, A[e .. g-1] = pivot, A[g .. *b*] > pivot

*k* = *n*-*k*+1 // kth largest element = (n-k+1)th smallest element

sizeL = e-1

sizeE = g – e

sizeG = *n* – g +1

**if** *k* ≤ sizeG **then** In-Place-Quick-Selection(A, *g, b, k*) // select it in in A[*g .. b*] ( i.e. in G)

**else if** *k* ≤ sizeG + sizeE **then return** pivot // pivot = kth largest element

**else** In-Place-Quick-Selection(A, *a*, *e-1*, *k* – sizeG – sizeE) // select it in A[*a .. e-1*] ( i.e. in L)

**Algorithm Partition(A, *a, b*)**

// For the partition, any algorithm of Blue-Red-Green in Q2B may be used.

// A[*a* .. e-1] < pivot, A[e .. g-1] = pivot, A[g .. *b*] > pivot.

**Input:** an array A of *n* elements and an integer k ∈ [1.. *n*]  
**Output:** the indices, e and g, s.t. an array A that is partitioned in 3 sections in the order

s.t. A[*a* .. e-1] < pivot, A[e .. g-1]= pivot, A[g .. b] > pivot.

pivot ← A[b] (or choose a pivot randomly, then swap it with A[b]). // it could be done in the main.

e1 ← *a* //tracks first element = pivot, so A[e1-1] = the last element < pivot

g ← *b* //tracks first element > pivot

e2 ← *a* //tracks last element = pivot,

**while** e2 < g

**if S[e2] = pivot then** e2 ← e2 + 1

**else if S[e2] < pivot**  **then**

swap S[e2] and S[e1]

e1 ← e1 + 1

e2 ← e2 + 1

**else** //S[e2] > pivot

swap S[e2] and S[g]

g ← g – 1

return e1, g

Q5. [10/10] **Weighted Median**

In finding the weighted median element, what does the weighted median algorithm return if the weights of all the elements are equal?

When all the weights of the elements are equal, the weighted median will return to the “regular” median.

Q6. [10/10] **Mode**

Given an array A of *n* numbers in the range from 1 to *n*, write an algorithm that runs in O(*n*) time for finding the ***mode***, i.e. the number that occurs most frequently in A.

**Algorithm** mode

Input: Array A[n]

Int maxKey, maxCount = 0;

Int [ ] counts = new int[A.length]

For (i=0; i< A.length; i++) do

Counts[A[i]]++

If (maxCount < counts[A[i]] ) then

maxCount = counts[A[i]]

maxKey = A[i]

endif

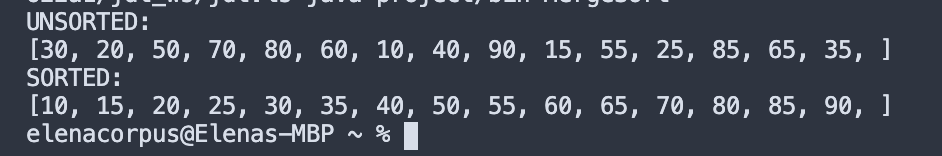
endfor

return maxKey

**Q7 – Q10: Implementation in Python/Java**

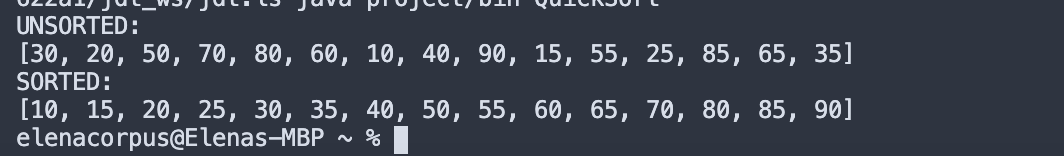
In the array A[35, 65,85, 25,55, 15, 90, 40, 10, 60, 80, 70, 50, 20, 30],

Q7. [13/25] Sort the array A in the descending order by Merge Sort and print the final array A.

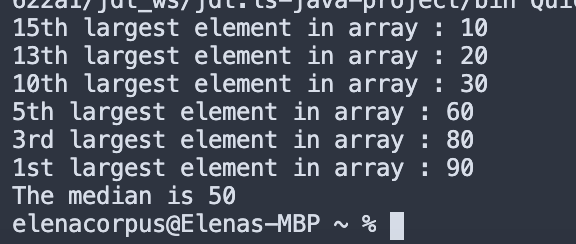


# the question asks for decreasing order.

Q8. [25/25] Sort the array A in the ascending order by Quick Sort and print the final array A.



Q9. [17/20] Find the *k*th largest element in the array A by an ***In-Place Quick Selection*** and print the median element, the 1st, 3rd, 5th, 10th, 13th and the15th largest elements.

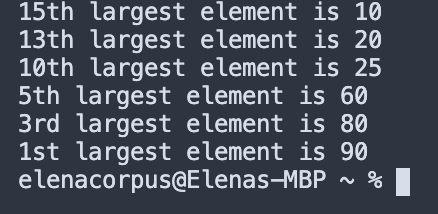


#10th largest element-35

5th largest-65

you missed to print the median which is 50.

Q10. [18/20] Find the *k*th largest element in the array A by a ***deterministic Selection*** and print the 1st, 3rd, 5th, 10th, 13th, 15th largest elements.



# 10th largest- 35

5th largest- 65