# CSE 5311 Homework Assignment 3 (Fall 2019)

Due date: 9/18 (Wednesday) (type, print, and hand-in in class)

[Submit source codes of your four programs required in the assignment via Canvas.]

**1. [20 points] Problem 4-2 on Page 107**

***4-2 Parameter-passing costs***

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an $N$-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time $= \Theta(1)$.
2. An array is passed by copying. Time $= \Theta(N)$, where $N$ is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p\,..\,q]$ is passed.

***a.*** Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let $N$ be the size of the original problem and $n$ be the size of a subproblem.

***b.*** Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

**Answer:**

**Binary search**

1. $T(n)=T(\frac{n}{2})+c=\Theta(\lg n)$ (master method)
2. $T(n) = cN+T(\frac{n}{2}) = 2cN+T(\frac{n}{4}) = 3cN+T(\frac{n}{8}) = \sum_{i=0}^{\lg n-1}(\frac{2^i cN}{2^i}) = cN\lg n = \Theta(n\lg n)$
3. $T(n)=T(\frac{n}{2})+cn=\Theta(n)$ (master method)

**Merge sort**

MERGE-SORT A[1 . . n]
1. If n = 1, done.
2. Recursively sort A[ 1 . . n/2 ] and A[ n/2 +1 . . n ] .
3. "Merge" the 2 sorted lists.

1. $T(n) = 2T(\frac{n}{2})+cn = \Theta(n\lg n)$ (master method)
2. $T(n) = 2N+cn+2T(\frac{n}{2}) = (2+4)N+(cn+cn)+4T(\frac{n}{4}) = (2+4+8)N+(cn+cn+cn)+8T(\frac{n}{8}) = \ldots$

$= (2N -2)N + cn*\log N +N*T(1) = \Theta(n^2)$

3. $T(n) = 2T(\frac{n}{2})+cn+2n/2 = 2T(\frac{n}{2})+(c+1)n = \Theta(n\lg n)$ (master method)

**2. [40 points] Problem 7-2 (a) (b) and (c) on Page 186**

***7-2 Quicksort with equal element values***

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

***a.*** Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

***b.*** The PARTITION procedure returns an index $q$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1..r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p..r]$ and returns two indices $q$ and $t$, where $p \le q \le t \le r$, such that

- all elements of $A[q..t]$ are equal,
- each element of $A[p..q-1]$ is less than $A[q]$, and
- each element of $A[t+1..r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

***c.*** Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION$'$, and name the new procedure RANDOMIZED-QUICKSORT$'$. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT$'(p, r)$ that calls RANDOMIZED-PARTITION$'$ and recurses only on partitions of elements not known to be equal to each other.

And (d) write two quicksort programs, one for RANDOMIZED_QUICKSORT(A, p, r) (see Page 189) and one for RANDOMIZED_QUICKSORT'(A, p, r). Run the programs on the array of numbers [5 6 8 10 11 13 8 8 3 5 2 11 8] and report number of recursive calls to RANDOMIZED_QUICKSORT() or RANDOMIZED_QUICKSORT'().

**Answer:**

**a.**

If all elements are equal, then when PARTITION returns, q = r and all elements in A[p…q-1] are equal. We get the recurrence $T(n) = T(n-1) + \Theta(n)$ for the running time, and so $T(n) = \Theta(n^2)$.

**b.**

**Algorithm 1** PARTITION'(A,p,r)

```
1:  x = A[r]
2:  exchange A[r] with A[p]
3:  i = p − 1
4:  k = p
5:  for j = p + 1 to r − 1 do
6:      if A[j] < x then
7:          i = i + 1
8:          k = i + 2
9:          exchange A[i] with A[j]
10:         exchange A[k] with A[j]
11:     end if
12:     if A[j] = x then
13:         k = k + 1
14:         exchange A[k] with A[j]
15:     end if
16: end for
17: exchange A[i + 1] with A[r]
18: return i + 1 and k + 1
```

**c.**

```
RANDOMIZED-PARTITION'(A,p,r)
    i = RANDOM(p,r)
    exchange A[r] with A[j]
    return PARTITION'(A,p,r)

QUICKSORT'(A,p,r)
    if p<r
        q,t = RANDOMIZED-PARTITION'(A,p,r)
        QUICKSORT'(A,p,q-1)
        QUICKSORT'(A,t+1,r)
```

**3. [40 points] Problem 7-4 on Page 188**

***7-4 Stack depth for quicksort***

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called ***tail recursion***, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      // Partition and sort left subarray.
3      q = PARTITION(A, p, r)
4      TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5      p = q + 1
```

***a.*** Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array $A$.

Compilers usually execute recursive procedures by using a ***stack*** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is ***pushed*** onto the stack; when it terminates, its information is ***popped***. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The ***stack depth*** is the maximum amount of stack space used at any time during a computation.

***b.*** Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an $n$-element input array.

***c.*** Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

(Hint: For subproblem (c) you can choose the smaller subarray to apply recursion so that the O(lgn) worst-case stack depth can be achieved. You don't have to explain how to "Maintain the O(nlg n) expected running time of the algorithm.".

And (d) write two quicksort programs, one for TAIL-RECURSIVE-QUICKSORT (A, p, r) and one for Optimized_TAIL-RECURSIVE-QUICKSORT (A, p, r) designed in subproblem (c). Assume each function call increases the stack depth by one. And each exiting of a function decreases the stack depth by one. Record the stack depth each time it changes. Run the programs on the array of numbers [5 6 8 10 11 13 8 8 3 5 2 11 8] and plot the changing stack depths for each program's run as a curve. And draw the two curves in one figure.

Answer:

**a.** TAIL-RECURSIVE-QUICKSORT does exactly what QUICKSORT does; hence it sorts correctly. QUICKSORT and TAIL-RECURSIVE-QUICKSORT do the same partitioning, and then each calls itself with arguments A, p, q − 1. QUICKSORT then calls itself again, with arguments A, q + 1, r. TAIL-RECURSIVE-QUICKSORT instead sets p ← q + 1 and performs another iteration of its **while** loop. This executes the same operations as calling itself with A, q + 1, r, because in both cases, the first and third arguments (A and r) have the same values as before, and p has the old value of q + 1.

**b.** The stack depth of TAIL-RECURSIVE-QUICKSORT will be Θ(n) on an n-element input array if there are Θ (n) recursive calls to TAIL-RECURSIVE-QUICKSORT. This happens if every call to PARTITION(A, p, r) returns q = r. The sequence of recursive calls in this scenario is

QUICKSORT′(A,1,n) ,
QUICKSORT′(A,1,n − 1) ,
QUICKSORT′(A,1,n − 2) ,
…
QUICKSORT′(A,1,1) .

Any array that is already sorted in increasing order will cause TAIL-RECURSIVE-QUICKSORT to behave this way.

**c.** The problem demonstrated by the scenario in part (b) is that each invocation of TAIL-RECURSIVE-QUICKSORT calls TAIL-RECURSIVE-QUICKSORT again with almost the same range. To avoid such behavior, we must change TAIL-RECURSIVE-QUICKSORT so that the recursive call is on a smaller interval of the array. The following variation of TAIL-RECURSIVE-QUICKSORT checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is Θ(lg n) in the worst case. Note that this method works no matter how partitioning is performed (as long as the PARTITION procedure has the same functionality as the procedure given in Section 7.1).

**Algorithm 4** MODIFIED-TAIL-RECURSIVE-QUICKSORT(A,p,r)

```
1: while p < r do
2:     q =PARTITION(A, p, r)
3:     if q < ⌊(r − p)/2⌋ then
4:         MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5:         p = q + 1
6:     else
7:         MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, q + 1, r)
8:         r = q − 1
9:     end if
10: end while
```

The expected running time is not affected, because exactly the same work is done as before: the same partitions are produced, and the same subarrays are sorted.