Samip Shah 1001709873
samipjanish.shah@mavs.uta.edu CSE5311

1. 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 = $\Theta(p - q + 1)$ if the subarray $A[p \ldots q]$ 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.

Solution:
Binary search works on sorted array. It starts with pointing the middle element of the list and ends if the element is found or the size of sub list becomes one.
The worst-case scenario is that the element is not present in the list. The first sub list searched in binary search is list itself with N elements and after every recursion, the list is divided into half. Thus, to complete this is expressed as T(n/2).

1. When array is passed by pointer
$T(n) = T(n/2) + c$

So by using master method for this, we have a=1, b=2, f(n)=c.
Thus it becomes $=n^{\log_2 1} = n^0 = 1$
As $f(n) = \Theta(n^{\log_b a}) = \Theta(1) = c$
$T(n) = \Theta(f(n) \lg n) = \Theta \lg(n)$
So the worst case is O(lg n)

2. When array is passed by copying
$T(n) = T(n/2) + \Theta(N)$

For every recurrence copied in binary search there are N elements copied.
Time involved = $\Theta(N)$
$T(n) = T(n/2) + \Theta(N)$
$= (n/2) + (n/4) + 2N$
$= (n/2) + (n/4) + (n/8) + 3N$
$= (n/2) + (n/4) + (n/8) + (n/16) + 4N$
$= (n/2) + (n/4) + (n/8) + (n/16) + (n/32) + 5N$
.
.
For lg n times
$= \sum_{i=1}^{\lg n} (n/2^i) + \lg n\ (N) = \Theta\ (N \lg n)$
Worst case is O(N lg n)

Reference for question 2 and 3 had been taken from various sources on internet and even some of the sentences and language has been copied but after understanding what it says. Programs are been submitted on canvas.

3. When array is passed by copying only subrange

$T(n)= T(n/2) + \Theta(n/2) = T(n/2) + \Theta(n)$

Because in next recurrence there will be half elements from current recurrence.

Applying master method a=1, b=2, $f(n)= \Theta(n/2)$

$n^{lg1} = n^0 = 1$

$[c(n/2)]/2 = [cn/4]$ and

$[cn/4] =< [cn/2]$

$T(n) = \Theta(n/2)= \Theta(n)$

Worst case is O(n)

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

1. When array is passed by pointer

   $T(n) = 2T(n/2) + cn$

   Solving using Master's method we get $T(N) = \Theta(N \lg N)$.

2. When array is passed by copying

   $T(n) = 2T(n/2) + cn + 2\Theta(N)$

   Solving the reccurence we get $T(N) = \Theta(N \lg N) + \Theta(N2) = \Theta(N2.$

3. When array is passed by copying only subrange

   $T(n)= 2T(n/2) + cn + 2c^i\ n/2 = T(n/2) + \Theta(n)$

   Solving using Master's method we get $T(N) = \Theta(N \ln N)$.

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

Solution.

a. Since all elements are the same, the initial random choice of index and swap change nothing. Thus, randomized quicksort's running time will be the same as that of quicksort. Since all elements are equal, PARTITION(A, P, r) will always return r − 1. This is worst-case partitioning, so the runtime is $\Theta(n2)$.

b. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

c

/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)

Reference for question 2 and 3 had been taken from various sources on internet and even some of the sentences and language has been copied but after understanding what it says. Programs are been submitted on canvas.

```
{
if (low < high)
{
/* pi is partitioning index, arr[pi] is now
at right place */
pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);  // Before pi
quickSort(arr, pi + 1, high); // After pi
}
}
```

d. Already Uploaded the Solution.

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

Solution:

a. We'll proceed by induction. For the base case, if A contains 1 element then
p = r so the algorithm terminates immediately, leave a single sorted element.
Now suppose that for $1 \leq k \leq n - 1$, TAIL-RECURSIVE-QUICKSORT correctly sorts an array A containing k elements. Let A have size n. We set
q equal to the pivot and by the induction hypothesis, TAIL-RECURSIVEQUICKSORT correctly sorts the left subarray which is of strictly smaller
size. Next, p is updated to q + 1 and the exact same sequence of steps follows
as if we had originally called TAIL-RECURSIVE-QUICKSORT(A,q+1,n).
Again, this array is of strictly smaller size, so by the induction hypothesis it
correctly sorts A[q + 1...n] as desired.

b. The stack depth will be $\Theta(n)$ if the input array is already sorted. The right subarray will always have size 0 so there will be $n - 1$ recursive calls before the while-condition $p < r$ is violated.

c. :

```
while p < r do
q =PARTITION(A, p, r)
if q < b(r − p)/2c then
```

Reference for question 2 and 3 had been taken from various sources on internet and even some of the sentences and language has been copied but after understanding what it says. Programs are been submitted on canvas.

MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
p = q + 1
else
MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, q + 1, r)
r = q − 1
end if
end while

d.

Reference for question 2 and 3 had been taken from various sources on internet and even some of the sentences and language has been copied but after understanding what it says. Programs are been submitted on canvas.