Assignment 1: Theory Questions

● Graded

Student

Tyler NGUYEN

Total Points

79.5 / 80 pts

Question 1

Question 1 — **20** / 20 pts

1.1 **Q1 part (a)** — **2** / 2 pts

✔ **+ 2 pts** Correct

– 2 pts Failed to identify that the resultant array should be a rearrangement of the original array

1.2 **Q1 part (b)** — **8** / 8 pts

✔ **+ 8 pts** Correct

– 2 pts Stated loop invariant is not correct

– 2 pts Initialization step is not correct

– 2 pts Maintenance step is not correct

– 2 pts Termination step is not correct

1.3 **Q1 part (c)** — **6** / 6 pts

✔ **+ 6 pts** Correct

– 2 pts Presented loop invariant is wrong

– 1 pt Presented initalization step is not correct

– 2 pts Presented maintenance step is not correct

– 1 pt Presented termination step is not correct

1.4 **Q1 part (d)** — **4** / 4 pts

✔ **+ 4 pts** Correct

– 2 pts Runtime analysis of Bubble sort is not correct

– 2 pts The comparison of runtime for Bubble sort and Insertion sort is not accurate

Question 2

Question 2

19.5 / 20 pts

+ 20 pts Correct

− 15 pts Ordering of the functions is not correct (0.5 for each correct ordering)

− 5 pts Equivalence classes are not correct

− 0.5 pts Point adjustment

Question 3

Question 3

20 / 20 pts

3.1 **Q3 part (a)** — 5 / 5 pts

✔ + 5 pts Correct

− 5 pts Wrong

3.2 **Q3 part (b)** — 15 / 15 pts

✔ + 15 pts Correct

− 5 pts Wrong basis

− 5 pts Wrong inductive hypothesis

− 5 pts Wrong inductive steps calculation

Question 4

Question 5

20 / 20 pts

✔ + 20 pts Correct

− 8 pts Presented algorithm is incorrect

− 7 pts Proof of correctness is incorrect

− 5 pts Complexity analysis is incorrect

Questions assigned to the following page: 1.1 and 1.2

assignment 331

Author: Tyler Nguyen UCID: 301585

question 1: exercise 2-2 on pages 46 of CLRS:

a)

in order to show that BUBBLESORT actually sorts, we also need to prove that both arrays A’ and A contain the same elements. This is proved easily by the given pseudocode because the alteration that we are applying to array A is a swap/exchange between its elements, therefore the output array A’ contains the sorted sequence of array A’s elements.

b)

loop invariant:

- at the start of each iteration of step 2, the smallest element in A[i…n] is positioned within the subarray A[i…j].

in order to prove this loop invariant I will use the structure of proving that his loop invariant holds true for initialization, maintenance and termination.

- initialization:
  - at initialization of the loop, variable j is set to n which is the array length (A.length), therefore the smallest element in A[i…n] is still within A[i…j] since no major modification to the specific elements have occurred yet.
- maintenance
  - as step 2s inner loop continues to execute, it compares the adjacent elements A[j] and A[j-1]. Now if A[j] < A[j-1] then a swap/exchange will occur, this implies that the current element at position j is not in its correctly sorted position within A[i…j]. Therefore the smallest element moves closer and closer to the beginning of the subarray with each iteration. This still maintains the loop invariant because it still keeps the smallest element in A[i…n] within the subarray A[i…j] with each iteration either through swapping/exchanging elements or by not moving at all.
- termination
  - once reaching termination i has also reached n - 1, which means that step 1s outer loop has finished its iterations, ensuring all elements have been considered. Therefore the array should be sorted from the smallest to largest element. This then means that the smallest element in A[i…n] is in position i, which means that the smallest element is also still within A[i…j], therefore the loop invariant holds.

Questions assigned to the following page: 1.3 and 1.4

c)

loop invariant:

- at the start of each iteration of step 1, the elements in A[1…i-1] are arranged in ascending sorted order, where i - 1 are the smallest elements of A.

in order to prove this loop invariant I will use the structure of proving that this loop invariant holds true for initialization, maintenance and termination.

initialization:

- at initialization of the loop variable i is set to 1 which then can be visualized as A[1…1-1] which represents the subarray A[1…0]. Since A[1…0] means that it is empty therefore containing 0 smallest elements of A in sorted order. Therefore the invariant holds at initialization.

maintenance:

- as the algorithm continues, the loop invariant still holds through each iteration, by assuming that A[1…i-1] holds the i - 1 smallest elements of A in ascending order. Now execute the loop in lines 2 through 4. As established in part b when this loop terminates the smallest element from A[i..n] will be in position i. given that we already have arranged the i - 1 smallest elements of A within A[1…i-1] which then follows A[i] must represent the ith smallest element of A. Therefore the result of the subarray A[1…i] remains sorted in ascending order which preserves the loop invariant.

termination:

- once reaching termination where loop variable i reaches n, the entire array A and its elements are sorted from least to greatest and all elements in A[1…n] are in their proper sorted positions.

d)

- by analyzing the number of swaps and comparisons we can determine the worst running time of BUBBLESORT
- comparisons and swaps:
    - number of comparisons are just the same as the amount of the if condition being evaluated, which is every iteration of the inner loop, therefore is n - i times for each iteration of the outer loop. This then makes it that the total number of comparisons is the sum of n - 1 + n - 2 +…+1, this arithmetic series can be seen as (n-1)*n/2 and by expansion we get (1/2)n^2 - (1/2)n, which can be translated into O(n^2)
    - for the swaps, this number depends on how many times the if condition evaluates to true, where in the worst case the array is in reverse sorted order, therefore making the if

Questions assigned to the following page: 2 and 1.4

statement true for each iteration. Now using what we know about the if conditional from above the number of swaps is also (n-1)*n/2 so we also get O(n^2)

    - this then means the bubble sort algorithm has a time complexity of O(n^2) in the worst case

- insertion sort comparison:
    - the worst running time of both insertion and bubble sort are the same which is O(n^2) where both arrays are reverse sorted
    - and the best running time for both are the same which is O(n) where both arrays are already sorted

question 2: exercise 3-3 part a on pages 71-72 of CLRS:

a) ranking the functions by order of growth

the rows with two functions represent the fact that they are in the same class

function simplifications:

- 2^(lg*n) = n from b^(log_b)*(M)=M
- 4^lgn = 2^2lgn = 2^lgn^2 = n^2 from b^(log_b)*(M)=M
- (√2)^lgn = n^1/2 from √2 = 2^(1/2) ∴ (2^(1/2))^lgn = 2^((1/2)*lgn) = 2^lgn^(1/2) = n^(1/2) from b^(log_b)*(M)=M
- n^(1/lgn) = n^log2 (base n) = 2 from b^(log_b)*(M)=M

function ranking
2^2^(n+1)
2^2^n
(n+1)!
n!
e^n
n*2^n
2^n
(3/2)^n
n^lglgn and (lg n)^lg n
(lgn)!
n^3
n^2 and 4^lgn

Questions assigned to the following page: 3.1, 2, and 3.2

n lg n and lg(n!)
n and 2^lg n
(√2)^lgn
2^√2lgn
lg^(2)n
ln n
√lgn
lnlnn
2^lg*n
lg*(lg n) and lg*n
lg(lg*n)
1 and n^(1/lgn)

question 3: exercise 4-1 on pages 119 of CLRS: T(n) = 2T(n/4)+√n

a) master method

- from the given recurrence relation: a = 2, b = 4, d =0.5, implies n^dlog(n) since a=b^d.
- therefore Θ(√nlg(n))

b) substitution method

- from the master method we identified that the solution to the recurrence is T(n) = Θ(√nlg(n)), but now we want to use the substitution method to prove that T(n) = O(g(n)). From the substitution method taught from lecture and tutorial T04 we first need to make a conjecture about the form of the solution. Since we already know the form of the solution let us make the conjecture to be form T(n) = T(n) = O(√nlg(n)), meaning that there exists n_0 > 0 and c > 0 such that for all n ≥ n_0 we will have T(n)≤c · √nlg(n).
    - recurrence relation to help us make a conjecture on the form of the solution: T(n) is constant for n≤2, say d, from the problem.

      T(n) = d if and only if n≤2

      T(n) = 2T([n/4])+√n if and only if n>2
- since, now we have a our conjecture on the form of the solution we need to verify it, which we will be doing so through establishing the basis and induction step. First since we know that we know that T(n) = d if n≤2 we can consider n = 4 as our base case which in turn makes n_0 = 4,
    - T(4) = 2T([4/4])+√4 ≤ 2d+2 ≤ q ≤ c · 2 · 2

Questions assigned to the following page: 3.2 and 4

- so long as there exists c ≥ q/4 where q is constant to simplify 2d+2 into one variable

- now for our inductive hypothesis: we can suppose for every m<n we have T(m)≤c · √m · lgm, then we can also assume T(n/4) ≤ c · √(n/4) · lg(n/4) from the established inductive hypothesis.
- finally our inductive step: we want to show that T(n) ≤ c · √nlg(n)
  - T(n) = 2T([n/4]) + √n
  - ≤ 2(c · √(n/4) · lg(n/4)) + √n
  - = 2(c · √n/2 · lg(n/4)) + √n
  - = 2(c · √n/2 · lg(n/4)) + √n
  - = 2(c · √n/2 · (lg(n) - lg4)) + √n (note: lg(n/4) = lg(n) - lg4 = lg(n) - 2)
  - = 2(c · √n/2 · (lg(n) - 2)) + √n
  - = c · √n · (lg(n) - 2) + √n
  - = c · √nlg(n) - 2c · √n + √n
  - ≤ c · √nlg(n) (if c≥1)
- lastly we can now conclude that our conjecture was correct for the solution form of T(n) = O(√nlg(n)) from our basis where when we have n_0 = 4 and c ≥ q/4 as well when our induction step works if and only if we have c≥1 therefore we can say that n_0 = 4 while c = max{1,q/4}, which proves our conjecture.

question 4: on gradescope

question 5:

in this question I will design my algorithm through the basis of the observations that were taught in tutorial section T04, which also proves and explains my algorithm. It will presented after the pseudocode. The observations are:

1. point out key observations that you rely on in your design of the algorithm
2. give an overview of the algorithm design
3. give a specification of the array
4. explain the individual steps
5. point out key observations about the algorithms run time behaviour
6. show the correctness of the algorithm
7. show time/space complexity analysis

Question assigned to the following page: 4

pseudocode:

```
/*
Algorithm: KLargestElements(H, k) Pseudocode
Input: max-heap H, integer k
Output: array A with the k largest elements in H
*/
A = {} //initialize empty array that holds our output of k largest elements
PriorityQueue B = {} //initialize a priority queue

B.push((H[1],1)) //insert the root of H with its index into B
for j from 1 to k do
  current_value and i = B.pop() //get the current lagest value and its index from B
  A[j] = current_value
  left_child_index = 2*i
  right_child_index = 2*i+1
  if left_child_index <= size of H do
    B.push((H[left_child_index], left_child_index)) //insert the left childs value and index into B
  if right_child_index <= size of H do
    B.push((H[right_child_index], right_child_index)) //insert the right childs value and index into B
end
Return A //return the array A which contains the k largest elements
```

observation 1:

- the observations of the algorithm is that it uses a max-heap called H tod the k largest elements and to do so, it utilizes a priority queue named B in order to record the current largest elements and the indices. It also takes advantage of the max-heap property of how the maximum element is always at the root which is index 1 for the pseudocode

observation 2:

- firstly the algorithm will initialize two arrays A and B which represent the storage of the k largest elements and a priority queue respectively. After it will push the root of the max-heap, H, into B. Since we need a k amount of largest elements we enter a loop which starts from 1 to k, where for each k we need to extract the current largest value and its index from B to further store the current largest value in the result array A, then insert the left and right children of the current node back in B. Then we can return the array A which holds the k largest elements

observation 3:

- the output array A holds the k largest elements from the max-heap H

observation 4:

- the presented algorithm initializes an empty array A that stores the k largest elements from max-heap H and a priority queue B, which will be utilized so that H does not get modified through out the algorithm. Then the algorithm extracts the largest element from priority

Question assigned to the following page: 4

queue, B that also corresponds to current largest in max-heap, H, repeatedly as well as inserting the left and right children of the extracted node back in priority queue B, where these process will continue for k iterations find the largest element every time.

observation 5:

- this algorithm takes advantage of the max-heap structure and priority queue to determine the k largest elements.

observation 6:

- i will be proving the correctness of this algorithm through partial correctness by establishing a loop invariant then proving that it holds through initialization, maintenance and termination.
- loop invariant: At the beginning of each iteration of the for loop, array A will contain the j-1 largest elements of the max-heap in descending order
- initialization: before the first iteration where j = 1, A is empty hence it contains 0 of the largest elements from H. Therefore no elements were added to array A, which satisfies the loop invariant in where j - 1 = 1 - 1 = 0, 0 largest elements in descending order in the output array A.
- maintenance: Once we get past initialization pre iteration we want to see if the loop invariant still holds. So during this iteration where j > 1, so firstly we extract the current largest value from B and then store it into output array A, this element will be the jth largest element from the H. Because, we are storing elements in descending order, adding the jth largest element to A will establish that A will continue to contain the j largest elements of the H which is the max-heap in descending order. Then from the current node the left and right children will be inserted into the priority queue B which ensures that the k-j largest elements not yet added to array A are still there for future iterations if there are any, this makes it so that the loo invariant holds after the jth iterations.
- termination: At the end where the loop will terminate after the k iterations, array A will contain the k -1 largest elements from the max-heap H in descending order which was enforced by the loop invariant. Because the kth largest element was stored in the final iteration output array A finally contains the k largest elements in descending order therefore satisfying the loop invariant.

observation 7:

- since the main loop has to run k times that means the operations of popping and pushing from and into the priority queue are performed as well. popping is O(log k) and pushing twice from the priority queue is O(log k) for each single push. Therefore for the worst-case time complexity for this algorithm is O(klogk) since we also want it to be dependent on k and not the size n, of the max-heap H.

No questions assigned to the following page.