

Note: NO partial score will be given for partially correct answer for subproblems in each problem (or for a single problem without subproblems). Unless otherwise noted, only the final solution is needed.

1. (10%) Let  $b_n$  be the number of different permutations obtainable by passing the numbers 1, 2, 3, ...,  $n$  through a stack and deleting in all possible ways. (a) Give the recursive formula for  $b_n$ . (b) Give the analytic formula for  $b_n$ .
2. (5%) Convert the expression " $a/(c*(b+d)))/(e-a)*c$ " into a postfix form.
3. (10%) Draw the final min-heap tree after the following operations: insert 7, insert 4, insert 3, insert 1, delete min, insert 9, insert 2, insert 5, delete min, delete min.
4. (8%) Describe how you can construct a heap of  $n$  keys with  $O(n)$  complexity. You can simply use  $n = 15$  as an example to explain your idea. (You don't need to prove the complexity.)
5. (20%) Quicksort is a fast and commonly used comparison-based sorting algorithm. Below you can find a version of quicksort algorithm in pseudo code format.

```
QUICKSORT( $A, p, r$ )
```

```
1  if  $p < r$ 
2       $q = \text{PARTITION}(A, p, r)$ 
3      QUICKSORT( $A, p, q - 1$ )
4      QUICKSORT( $A, q + 1, r$ )
```

```
PARTITION( $A, p, r$ )
```

```
1   $x = A[r]$ 
2   $i = p - 1$ 
3  for  $j = p$  to  $r - 1$ 
4      if  $A[j] \leq x$ 
5           $i = i + 1$ 
6          exchange  $A[i]$  with  $A[j]$ 
7  exchange  $A[i + 1]$  with  $A[r]$ 
8  return  $i + 1$ 
```

(a) (6%) True or false:

- (1) The output of QUICKSORT() above is a **non-increasing** sorted sequence. (2%)
- (2) The QUICKSORT() above is a **stable** sorting algorithm. (2%)
- (3) The QUICKSORT() above is an **in-place** sorting algorithm. (2%)

(b) (4%) The given version of QUICKSORT above cannot efficiently process the cases where in the input array there are a large number of repeated elements. Analyze the worst-case running time which occurs when all  $n$  keys in the input array  $A$  are identical. Please give it in big-O notation in terms of  $n$ .

(c) (10%) To address the previously mentioned problematic case, we will now develop a new partition algorithm PARTITIONTHREE() that separates the input keys into three groups, the ones that are smaller than, that are identical to, and that are larger than the pivot key  $x$ . The return values  $q1$  and  $q2$  represent the indices of the first and the last keys of the group that equals the pivot key, respectively. Fill the blanks (A), (B), (C) in PARTITIONTHREE() below to complete the implementation, such that QUICKSORTTHREE() runs in  $O(n)$ -time when the input array has all  $n$  elements identical.

```

QUICKSORTTHREE(A, p, r)
1  if p < r
2    (q1, q2) = PARTITIONTHREE(A, p, r)
3    QUICKSORTTHREE(A, p, q1 - 1)
4    QUICKSORTTHREE(A, q2 + 1, r)

```

```

PARTITIONTHREE(A, p, r)
1  x = A[r]
2  q1 = p
3  q2 = r
4  j = p
5  while j ≤ q2
6    if A[j] < x
7      exchange A[j] with A[q1]
8      // (A)
9      j = j + 1
10   elseif A[j] > x
11     exchange A[j] with A[q2]
12     // (B)
13   else
14     // (C)
15   return (q1, q2)

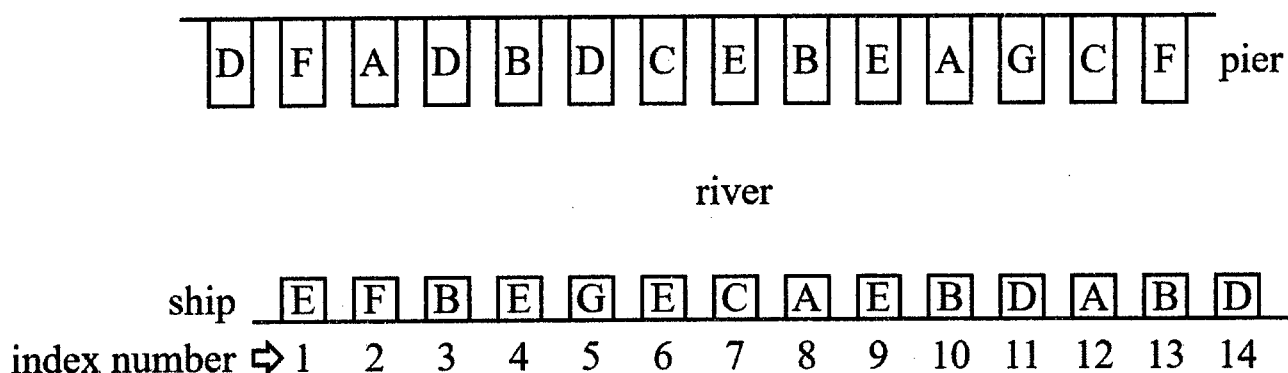
```

6. (13%) When implementing a hash table, one of the commonly used collision resolution methods is open addressing. Let  $k$  be the key, and  $m$  be the size of the hash table. Assume each table entry can hold one key.
- (a) (3%) The simplest open addressing method is called **linear probing**, where if a slot is already occupied, the following slots are "probed" sequentially. Assume an auxiliary hash function  $h'(k) = k \bmod m$ . Please give the hash function  $h(k, i)$  of linear probing with the given auxiliary function, where  $i$  is the number of times the hash table has been probed with this key (and thus starting from 0).
- (b) (5%) Linear probing is easy to implement but suffers from the problem of primary clustering. A more complex open addressing method which can mitigate this problem is **double hashing**, using a hash function in the form of  $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$ , with two auxiliary hash functions  $h_1(k)$  and  $h_2(k)$ .  $i$  is the number of times the hash table has been probed with key  $k$ . Assume two auxiliary hash functions  $h_1(k) = k \bmod 16$ ,  $h_2(k) = 1 + (k \bmod 15)$ , and  $m = 16$ . Show the final hash table content after inserting all of the following keys to an initially empty hash table in the given order: {16, 3, 35, 67, 51, 1, 15, 31, 19, 17}.
- (c) (5%) Prof. Alpha proposes a different double hashing method, using two auxiliary hash functions  $h_1(k) = k \bmod 16$ ,  $h_2(k) = 2(k \bmod 8)$ , respectively. Assume the table size  $m = 16$ . Use an example to briefly explain how Prof. Alpha's method is problematic.
7. (6%) True or false:
- (a) (2%) The Dijkstra's algorithm (single run) solves the single-source shortest path problem with positive, zero, or negative edge weights, if there is no negative cycle.
- (b) (2%) The Bellman-Ford algorithm (single run) solves the all-pairs shortest path problem with non-negative edge weights.
- (c) (2%) The Floyd-Warshall algorithm (single run) solves the all-pairs shortest path with positive, zero, or negative edge weights, even if there are negative cycles.
8. (8%) Given a directed graph  $G$  where vertex  $u$  (in  $G$ ) has at least one path to any other vertex (in  $G$ ), the following pseudocode is designed to output an acyclic graph. However, in Lines 8, 10, 11, and 14, each of A, B, C, and D should be 0, 1, or 2. Write down the correct values of A, B, C, and D so that the pseudocode can successfully output an acyclic graph.

```

1 Cycle-Removal(G,u)
2   Initialize S as a stack
3   Label all vertices in G as Type-0
4   Label u as Type-1
5   S.push(u)
6   while S is not empty
7     v = the top vertex of S // Comment: not S.pop() here
8     if v has a Type-A neighbor w in G
9       Remove (v,w) from G
10    else if v a Type-B neighbor w in G
11      Label w as Type-C
12      S.push(w)
13    else
14      Label v as Type-D
15      S.pop()
16    end if
17  end while
18  output G
    
```

9. (10%) At 9am, 14 ships are leaving for the other side as shown below, where the English alphabets mean the types of those ships. For safety reasons, each ship can only stop at a pier having the same alphabet, and, if the routes of two ships cross each other, one ship needs to wait 15 minutes before leaving.



(a) (4%) Write down the maximum number of ships which can leave at 9am.







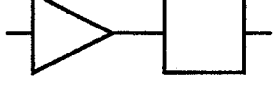



(b) (6%) Following (a), write down the index numbers (between 1 and 14) of the ships which can leave at 9am.

見背面

10. (10%) The goal of this problem is to transform the following sequence of shapes into another sequence of shapes so that the total cost is minimized.



The rules of transformation are shown below. For example, if you do not replace anything, the total cost is  $6 + 3 + 2 + 6 + 2 + 6 + 2 + 6 + 3 + 2 = 38$ . If you replace the last two shapes (circle and triangle) by the rule #3, the total cost is  $6 + 3 + 2 + 6 + 2 + 6 + 2 + 6 + 4 = 37$ . Note that you can choose not to replace a shape if the total cost can be minimized. Also, a rule of transformation can be applied more than once.

shape	cost	cost before transformation	rule of transformation	cost after transformation
	6	9 (6+3)	#1 → 	7
	3	8 (6+2)	#2 → 	7
	2	5 (3+2)	#3 → 	4
		8 (2+6)	#4 → 	6
		11 (6+3+2)	#5 → 	8
		14 (6+2+6)	#6 → 	12
		10 (2+6+2)	#7 → 	7

(a) (4%) Write down the minimum total cost after transforming the given sequence.

(b) (6%) Following (a), write down the applied rules of transformation from left of the sequence to right of the sequence.

試題隨卷繳回