Midterm Examination Problem Sheet
TIME: 04/12/2022, 13:20–16:10

This is an open-book online exam. You can use any materials that you are able to find, but ***you cannot communicate with anyone other than the TAs and the instructors*** *during the exam. Any form of cheating, lying or plagiarism will not be tolerated. Students can get zero scores and/or get negative scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconduct.*

Both English and Chinese (if suited) are allowed for answering the questions. We do not accept any other languages.

Unless explicitly asked, you do ***not*** *need to justify why your algorithm runs in the desired time/space complexity, but you are welcomed to briefly explain the key ideas to help the TAs grade your solution.*

There are 16 problems. 8 of them are marked with * and are supposedly simple(r); 4 of them are marked with ** and are supposedly regular; 4 of them are marked with *** and are supposedly difficult. The simpler ones are worth 10 points, and others are worth 15 points. The full credit is 200 points.

We will allow you to use at most one A4 page in your PDF for each problem. For most problems, you do not need a full A4 page. So you can put multiple problems in one A4 page as well (to save your time in scanning). But all solution lines to one problem must be on the same page. In addition, you must associate your solution to the right problem by tagging on Gradescope. As long as there is a clear indication of which problem you are solving and the right tagging, you do not need to sequentially answer by the order of problems in the PDF.

You should keep your answers as *concise* as possible to facilitate grading. In particular, please use proper *pseudo code* and/or sufficiently-understandable words to describe your algorithm to the TAs clearly!

Array

1. (*) Consider an array `arr` of length n that contains different positive integers. Design an in-place and $O(n)$ time-complexity algorithm that moves the odd integers to the beginning of the array and the even integers to the end of the array. That is, the algorithm should change the original array to a modified array with exactly the same set of elements, while returning some k such that `arr`$[i]$ is odd for all $1 \leq i \leq k$ and `arr`$[i]$ is even for all $k < i \leq n$. The order within the odd elements and the order within the even ones can be arbitrary.

2. (*) Assume that someone forgets to store k after executing the algorithm in the previous problem. Design an algorithm with $O(\lg n)$ time complexity and $O(1)$ extra-space complexity to re-calculate k when given the modified `arr` of length n from the previous problem.

Linked List

3. (*) Given a singly-linked list L

$$\ell_1 \to \ell_2 \to \ell_3 \to \ell_4 \to \ldots \to \ell_{2n} \to \text{NIL}$$

with $L.head$ pointing to ℓ_1. Furthermore, assume that $L.middle$ points to ℓ_{n+1}. Design an $O(n)$-time and $O(1)$-extra-space algorithm that modifies L to the following new singly linked list

$$\ell_1 \to \ell_{n+1} \to \ell_2 \to \ell_{n+2} \to \ldots \to \ell_{2n} \to \text{NIL} \quad .$$

You are only allowed to modify the *next* field of any node.

4. (***) Given a singly-linked list L

$$\ell_1 \to \ell_2 \to \ell_3 \to \ell_4 \to \ldots \to \ell_{2n} \to \text{NIL}$$

with $L.head$ pointing to ℓ_1. Assume that each node consists of an integer field *value*. Design an $O(n)$-time and $O(1)$-extra-space algorithm that determines whether L is the same as its reverse or not. You are allowed to modify the content (*next* and *value*) of any node if needed.

If you really do not know how to solve this problem for a singly-linked list, you can assume a *doubly-linked list* when answering and receive a maximum of 5 points.

Stack and Queue

5. (*) Consider a stack with the usual `push`, `pop`, `peep` (without popping) operations on integers. Design a data structure that supports an additional operation, `remove_values`(v), that removes all elements that are of value v from the stack. For instance, if the stack (starting from the bottom) contains

$$5, 1, 2, 6, 6, 2, 1, 1, 7$$

Then `remove_values`(2) changes the stack to

$$5, 1, 6, 6, 1, 1, 7$$

Your operation should be $O(n)$-time and $O(n)$-extra-space complexity, where n is the size of the stack.

6. (**) Consider a deque with the usual `push_front`, `pop_front`, `push_back`, `pop_back` operations on integers. Design a data structure that supports an additional operation, `advance`(k), that advances the k-th element of the deque for any $k \geq 2$. The advancing operation swaps the k-th element and the $(k-1)$-th element of the deque. For instance, if the deque (starting from the front) contains

$$5, 1, 2, 6, 7$$

Then `advance`(3) changes the deque to

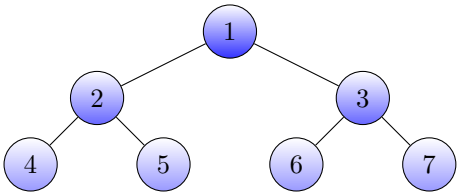
$$5, 2, 1, 6, 7$$

Your operation should be $O(k)$-time and $O(1)$-extra-space complexity, where k is parameter for `advance`.

Binary (Search) Tree

7. (*) For keys $1, 2, \ldots, 12$, show (draw) one of the shortest binary search trees for those keys; then show one of the tallest binary search trees for those keys.

8. (***) Consider the post-order traversal on a complete binary tree. For instance, for a complete binary tree with 7 nodes,



the post-order traversal visits the nodes in the order of

$$4, 5, 2, 6, 7, 3, 1$$

The traversal distance is defined as the location difference between two nodes in the traversal. For instance, in the traversal above, the traversal distance between node 6 and its parent (node 3) is 2. Now, assume that the root is of height 0, the complete binary tree is of height h, and the tree contains exactly $2^{h+1}-1$ nodes. What is the maximum traversal distance between a node at height d and its parent? Prove the correctness of your answer.

Heap

9. (*) Consider a binary max-heap, design an additional operation, `get_third`, that returns the third largest element, without modifying the heap. Your algorithm can use any internal structure (i.e. left/right/parent links) of the heap, and should run with $O(1)$-time and $O(1)$-extra-space complexity.

10. (***) Prove or disprove the following claim: the smallest element in a binary max-heap must be within its leaf.

Sort

11. (*) Consider the following INSERTION-SORT algorithm.

```
INSERTION-SORT(A)
1  for m = 2 to A.length
2      key = A[m]
3      i = m − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

Consider n numbers $a_1 < a_2 < a_3 < \ldots < a_n$. Assume that we apply INSERTION-SORT on a nearly-ordered array that contains

$$[a_1, a_2, \ldots, a_{k+1}, a_k, a_{k-1}, a_{k+2}, \ldots, a_n]$$

for some $1 < k < n$. How many times is line 5 executed in INSERTION-SORT?

12. (**) Consider the following recursive version of (top-down) MERGE-SORT.

```
MERGE-SORT(array A)
1  n = A.length
2  if n ≥ 2
3      m = floor((1 + n)/2)
4      MERGE-SORT(A[1 ... m])
5      MERGE-SORT(A[(m + 1) ... n])
6      MERGE(A[1 ... m], A[(m + 1) ... n])
7  PRINT(A[1 ... n])
```

The MERGE operation merges two sorted sub-arrays into one sorted array (with the help of some auxiliary space). For instance, MERGE-SORT($[1, 3, 4, 2]$) calls MERGE-SORT($[1, 3]$), which in term calls MERGE-SORT($[1]$) and MERGE-SORT($[3]$). We can examine the status of MERGE-SORT by printing out $A[1 \dots n]$ in line 7. Then, in the order of finishing each MERGE-SORT call,

result of PRINT	the corresponding function call
`[1]`	MERGE-SORT([1])
`[3]`	MERGE-SORT([3])
`[1, 3]`	MERGE-SORT([1, 3])
`[4]`	MERGE-SORT([4])
`[2]`	MERGE-SORT([2])
`[2, 4]`	MERGE-SORT([4, 2])
`[1, 2, 3, 4]`	MERGE-SORT([1, 3, 4, 2])

Write down the results of PRINT when calling MERGE-SORT($[3, 1, 4, 2, 6, 5]$) with a table similar to the one above.

Complexity!

For the next two sections, please prove the results using the following definition of the asymptotic notations (which is exactly the same as the notation you used in Homework 1). Suppose that f and g are asymptotically non-negative functions defined on $\mathbb{N}$. We call $f(n) = O(g(n))$ if and only if there exist positive numbers c and n_0 such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0.$$

For your proof, you can use

- any results that have been proved in our class
- any results in related sections of your textbook, regardless of whether they have been taught
- any results in fundamental Calculus or Discrete Match classes

Please properly list where those results are from if you choose to use them.

13. (*) Prove or disprove that $\lg(\lg k^n) = O(\lg n)$ for any $k > 0$.
14. (**) Prove or disprove that $(\lg n)^k = O(\lg n)$ for any $k > 0$.

Oh No, More Complexity!!

15. (**) Prove or disprove the following statement: If $|f(n) - g(n)| = O(1)$, then $2^{f(n)} = O(2^{g(n)})$.
16. (***) Assuming that both $f(n)$ and $g(n)$ are positive functions. Prove or disprove the following statement: If $|\lg f(n) - \lg g(n)| = O(1)$, then $2^{f(n)} = O(2^{g(n)})$.