COS 226	Algorithms and Data Structures	Fall 2025

Midterm

This exam has 8 questions worth a total of 60 points. You have 80 minutes.

Instructions. This exam is preprocessed by computer. Write neatly, legibly, and darkly. Put all answers (and nothing else) inside the designated answer spaces. *Fill in* bubbles and checkboxes completely: ● and ■. To change an answer, erase it completely and redo.

Resources. The exam is closed book, except that you are allowed to use a one-page reference sheet (8.5-by-11 paper, one side, in your own handwriting). No electronic devices are permitted.

Honor Code. This exam is governed by Princeton's Honor Code. Discussing the contents of this exam before the solutions are posted is a violation of the Honor Code.

Please complete the following information now.

Name:

NetID:

Exam room: ○ Friend 101 ○ Maeder 002 ○ McCosh 4 ○ Other

Precept:

P01	P01A	P02	P02A	P02B	P03A	P03B	P04
○	○	○	○	○	○	○	○

"I pledge my honor that I will not violate the Honor Code during this examination."

Signature

1. **Initialization. (1 point)**

   In the spaces provided on the front of the exam, write your name and NetID; fill in the bubbles for your exam room and the precept in which you are officially registered; write and sign the Honor Code pledge.

2. **Performance. (9 points)**

   (a) Consider a symbol table that is defined by the following Java implementation:

```
public class LinearProbingHashST {
    private int[] keys;     // the keys
    private String[] vals;  // the corresponding values
    private int n;          // number of key-value pairs

    ...
}
```

   In the *worst case*, how much memory (in bytes) does a `LinearProbingHashST` object use as a function of the number of key–value pairs n?

   Assume that `keys[]` and `vals[]` are *resizable arrays* whose lengths double when the array is $\geq \frac{1}{2}$ full and whose lengths halve when the array is $\leq \frac{1}{8}$ full.

   Count all memory (including object references) allocated by the `LinearProbingHashST`, but do not count the memory for the `String` objects themselves (which the client allocates). Analyze the memory using our 64-bit memory cost model; simplify your answer with *tilde notation.*

   Write your answers in the box below.

$\sim$	bytes

(b) Assume that a `MinPQ` is implemented using a binary heap with a *resizable array* (that doubles when full and halves when $\leq \frac{1}{4}$ full). What is the *worst-case running time* of the following code fragment as a function of n? *Fill in the best-matching bubble.*

```
MinPQ<Integer> pq = new MinPQ<Integer>();
for (int i = 0; i < n; i++)
    pq.insert(a[i]);
for (int i = 0; i < n; i++)
    a[i] = pq.delMin();
```

○ $\Theta(\log n)$　○ $\Theta(n)$　○ $\Theta(n \log n)$　○ $\Theta(n^2)$　○ $\Theta(n^2 \log n)$

(c) How many times is the function `op()` called as a function of n?
Fill in the best-matching bubble.

```
for (int i = 0; i < n*n; i++)
    for (int j = i+1; j < n*n; j++)
        for (int k = 1; k <= n*n; k = k*2)
            op();
```

○ $\sim \frac{1}{2} n^2 \log_2 n$　○ $\sim 2n^4$　○ $\sim \frac{1}{2} n^4 \log_2 n$　○ $\sim n^4 \log_2 n$　○ $\sim 2n^4 \log_2 n$

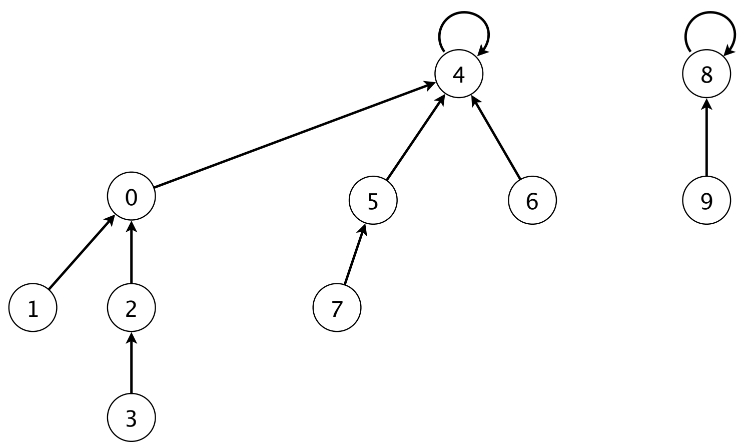
(d) Which expressions below correctly describe the function $f(n) = 2n^2 + 2n + 6 \log_2 n$?
Fill in all checkboxes that apply.

☐ $O(\log n)$　☐ $O(n)$　☐ $O(n^2)$　☐ $O(n^3)$

☐ $\Omega(\log n)$　☐ $\Omega(n)$　☐ $\Omega(n^2)$　☐ $\Omega(n^3)$

3. **Data structures. (10 points)**

(a) Consider the following *parent-link* representation of a *weighted quick union* (*link-by-size*) data structure:



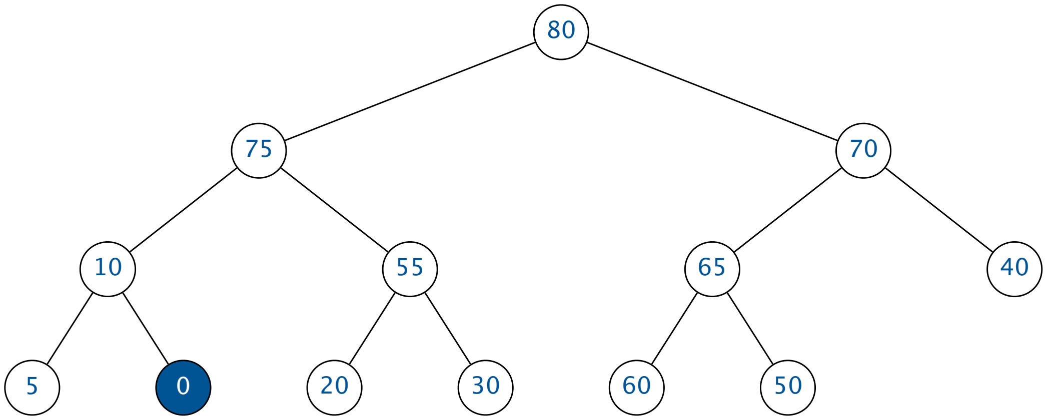
Which of the following could have been the pair (p, q) in the last call to `union(p, q)`? Assume that just prior to the call, p and q were in different sets.

Recall: when calling `union(p, q)` *with two trees of equal size, our tiebreaking convention is to make the root of tree containing q point to the root of the tree containing p.*

Fill in all checkboxes that apply.

☐ $(0, 4)$	☐ $(2, 3)$	☐ $(4, 0)$	☐ $(4, 5)$
☐ $(5, 0)$	☐ $(5, 7)$	☐ $(6, 4)$	☐ $(8, 9)$

(b) Consider the following binary tree representation of a *max-oriented binary heap*:



Suppose that the key in the highlighted node is changed from 0 to 77 and then a `swim()` operation is applied to restore heap order. Which pairs of keys are *compared*?

Fill in all checkboxes that apply.

☐ 0 *and* 5

☐ 0 *and* 77

☐ 5 *and* 77

☐ 10 *and* 55

☐ 10 *and* 77

☐ 70 *and* 75

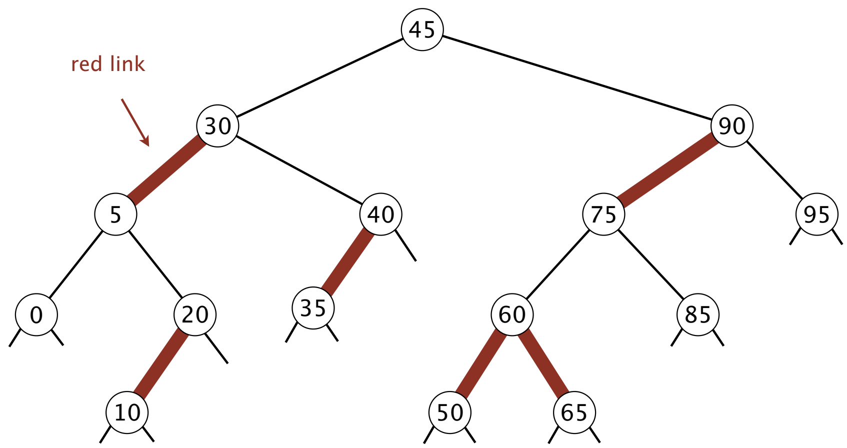
☐ 70 *and* 77

☐ 40 *and* 77

☐ 75 *and* 77

☐ 77 *and* 80

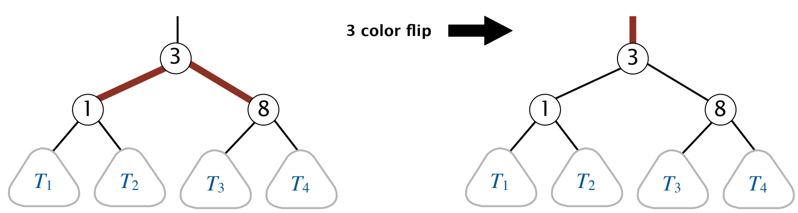
(c) The following BST satisfies perfect black balance, but violates the color invariants:

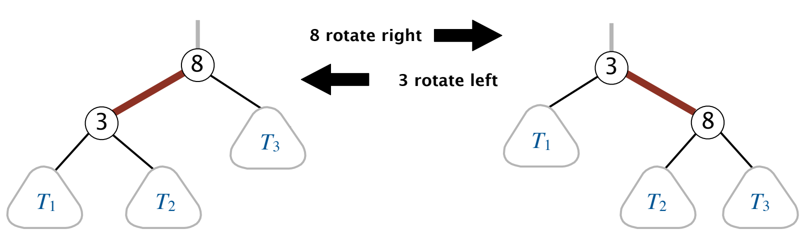


Give a sequence of 4 elementary operations that restores the color invariants.

	operation 1	*operation* 2	*operation* 3	*operation* 4
key				
color flip	○	○	○	○
rotate left	○	○	○	○
rotate right	○	○	○	○

Examples of elementary operations (for reference):





4. **Five sorting algorithms. (5 points)**

The leftmost column contains an array of 24 integers to be sorted; the rightmost column contains the integers in sorted order; the other columns are the contents of the array at some intermediate step during one of the five sorting algorithms listed below.

Match each algorithm by writing its letter in the box under the corresponding column. Use each letter exactly once.

37	11	11	95	35	11	11
11	17	20	94	11	16	16
39	20	23	83	16	17	17
86	23	35	88	17	20	20
54	35	37	92	20	23	23
44	37	39	68	23	35	35
35	39	44	37	37	37	37
23	44	54	75	44	39	39
94	54	67	86	94	40	40
20	67	68	56	54	42	42
67	68	86	67	67	44	44
68	83	94	63	68	54	54
95	86	17	44	95	95	56
17	88	75	17	86	68	63
83	94	77	35	83	83	67
88	95	83	23	88	88	68
75	75	88	11	75	75	75
77	77	95	77	77	77	77
40	40	16	40	40	94	83
42	42	40	42	42	86	86
56	56	42	20	56	56	88
92	92	56	54	92	92	92
16	16	63	16	39	67	94
63	63	92	39	63	63	95
A						G

A. Original array

B. Selection sort

C. Insertion sort

D. Mergesort (*top-down*)

E. Quicksort (*standard, no shuffle*)

F. Heapsort

G. Sorted array

5. **Analysis of a sorting algorithm. (10 points)**

Consider the following sorting algorithm:

```
// Rearrange elements of a[] to put in ascending order
public static void mysterySort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        boolean exchanged = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (less(a[j+1], a[j])) {
                exch(a, j+1, j);
                exchanged = true;
            }
        }
        if (!exchanged) break;  // no exchanges, already sorted
    }
}
```

(a) Which invariants hold at the beginning of each iteration of the i loop (i.e., for each value of i)? *Fill in all checkboxes that apply.*

- ☐ The first i array elements are in sorted order.
- ☐ The last i array elements are in sorted order.
- ☐ The first i array elements contain the i smallest elements in the array.
- ☐ The last i array elements contain the i largest elements in the array.

(b) Is the sorting algorithm *in-place*? *Fill in the corresponding bubble.*

- ○ *Yes.* It is in-place.
- ○ *No.* It is not in-place.

(c) Is the sorting algorithm *stable*? *Fill in the corresponding bubble.*

- ○ *Yes.* It is stable.
- ○ *No.* It is not stable.

(d) Consider running `mysterySort()` on an input array `a[]` of length n. For each quantity on the left, write the letter of the best matching expression on the right.

Write one uppercase letter in each box. You may use each letter once, more than once, or not at all.

☐	Number of calls to `exch()` to sort an integer array of the form $[2, 3, 4, ..., n-1, n, 1]$.	**A.** 0
		B. $\sim \frac{1}{2}n$
☐	Number of calls to `less()` in the *best case*.	**C.** $\sim n$
		D. $\sim 2n$
☐	Number of calls to `less()` in the *worst case*.	**E.** $\sim \frac{1}{2}n \log_2 n$
		F. $\sim n \log_2 n$
		G. $\sim \frac{1}{4}n^2$
		H. $\sim \frac{1}{2}n^2$
		I. $\sim n^2$

6. **Properties of algorithms and data structures. (9 points)**

Identify each statement as *true* or *false* by filling in the appropriate bubble.

true	*false*	
○	○	Mergesort makes $\sim 2n \log_2 n$ compares to sort any array of $2n$ comparable elements (both in the best case and worst case).
○	○	Any compare-based algorithm for *searching for a key in a sorted array* of length $3n$ must make $\Omega(\log n)$ compares in the worst case.
○	○	Consider the problem of sorting an array of n comparable elements in which there are only 4 distinct keys. It is possible to design an algorithm for the problem that makes $\leq 4n$ compares in the worst case.
○	○	Consider any *max-oriented binary heap* with at least 8 keys. Let x be a key in a node at depth 2 (grandchild of the root) and y be a key in a node at depth 3 (great-grandchild of the root). Then, we must have $x \geq y$.
○	○	Given any two *binary search trees*, each on n keys, it is possible to create a *binary search tree* on the $2n$ keys using at most $2n$ compares. Assume the $2n$ keys are distinct.
○	○	Consider inserting a key into a *left-leaning red–black BST*. During the insertion, there can be two (or more) consecutive *left rotations*, without an intervening *color flip* or *right rotation*.

7. **Binary search. (6 points)**

Design an algorithm to find the *index* of the first 1 in a *sorted array* of 0s and 1s. For simplicity, assume that the array contains at least one 0 and at least one 1. For example, in the follow array, the index of the first 1 is 3.

0	0	0	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9

To do so, complete the following partial implementation:

```
// precondition: a[] is sorted and contains only 0s and 1s,
//               with at least one 0 and at least one 1
public static int indexOfFirstOne(int[] a) {
    int lo = 0;
    int hi = a.length - 1;
    // invariants: a[lo] = 0 and a[hi] = 1
    while ( [1] ) {
        int mid = lo + (hi - lo) / 2;
        if (a[ [2] ] == [3] ) lo = [4] ;
        else                  hi = [5] ;
    }
    return [6] ;
}
```

A. `0`
B. `1`
C. `hi >= lo`
D. `hi > lo`
E. `hi > lo + 1`
F. `lo`
G. `lo + 1`
H. `mid - 1`
I. `mid`
J. `mid + 1`
K. `hi - 1`
L. `hi`

For each numbered oval above, write the uppercase letter of the corresponding expression on the right in the space provided. You may use each letter once, more than once, or not at all.

1	2	3	4	5	6

8. **Data structure design. (10 points)**

   Design a data type for a *uni-stack* of strings that supports the classic *push* and *pop* operations for a stack, but handles duplicate items in a different manner: if a string is pushed onto a uni-stack that already contains that string, the pre-existing string is removed from the stack.

   In particular, it should implement the following API:

`public class UniStack`

	`UniStack()`	*create an empty uni-stack of strings*
`void`	`push(String item)`	*remove any string on the uni-stack equal to* `item`*;* *then, add* `item` *to the uni-stack*
`String`	`pop()`	*remove and return the string on the uni-stack that was added most recently*
`int`	`size()`	*number of strings on the uni-stack*

Performance requirements.

- The amount of memory must be $O(n)$, where n is the number of items on the uni-stack.
- Each operation must take $O(\log n)$ time in the worst case.

Partial credit (for 50% credit).

- Same API as `UniStack` except that calling `push()` on a string already on the uni-stack has no effect (the string remains in place and is not duplicated).
- Same performance requirements as above.

Example. Here is a sample sequence of operations:

```
UniStack stack = new UniStack(); //  [ ]
stack.push("A");                 //  [ A ]
stack.push("B");                 //  [ B A ]
stack.push("C");                 //  [ C B A ]
stack.push("D");                 //  [ D C B A ]
stack.push("B");                 //  [ B D C A ]
stack.size();                    //  [ B D C A ]  returns 4 (not 5)
stack.pop();                     //  [ D C A ]    returns B
stack.pop();                     //  [ C A ]      returns D
stack.pop();                     //  [ A ]        returns C
stack.pop();                     //  [ ]          returns A (not B)
```

Note: the comments show the strings in the stack, with the most recently added item at left, (but the API does not require you to maintain the strings internally in any particular order).

Are you attempting the partial or full credit solution? You may attempt only one.

◯ *partial credit* ◯ *full credit*

(a) *Using Java code*, declare the instance variables (along with any supporting nested classes) that you would use to implement `UniStack`. You may use any of the data types that we have considered in this course (either `algs4.jar` or `java.util` versions). If you make any modifications to these data types, describe the modifications.

(b) Give a concise English description of your algorithm for implementing the method `push(String item)`. You may use code or pseudocode to improve clarity.

(c) Give a concise English description of your algorithm for implementing the method `pop()`. You may use code or pseudocode to improve clarity.

This page is intentionally blank. You may use this page for scratch work.