The University of Michigan
Electrical Engineering & Computer Science
EECS 281: Data Structures and Algorithms
Winter 2024

PRACTICE MIDTERM EXAM
KEY 1 – ANSWERS

Tuesday, March 5, 2024
7:00PM – 9:00PM

Uniqname: _____ **Student ID:** _____

Name: _____

Uniqname of person to your left: _____

Uniqname of person to your right: _____

Honor Pledge:
"I have neither given nor received unauthorized aid on this examination,
nor have I concealed any violations of the Honor Code."

Signature: _____

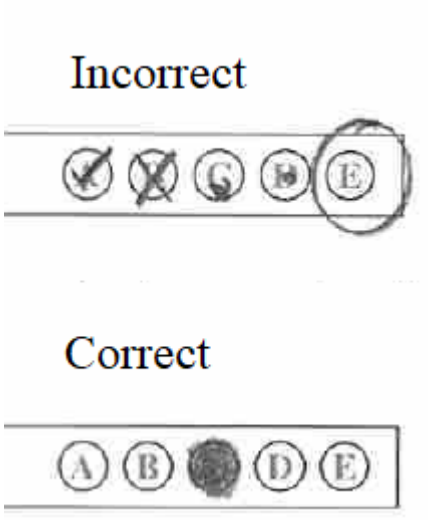
INSTRUCTIONS:

- The exam is closed book and notes except for one 8.5"x11" sheet of handwritten notes (both sides). All electronic device use is prohibited during the exam (calculators, phones, laptops, etc.).
- Print your name, student ID number, and uniqname **LEGIBLY**. Sign the Honor Pledge; this pledge applies to both the written and multiple choice sections. Your exam will not be graded without your signature.
- Record the **UNIQNAME** of the students to both your left and right. Write down `nullptr` if you are at the end of a row and there is no one on a given side.
- Record your **UNIQNAME** at the top of each odd-numbered page in the space provided. This will allow us to identify your exam if pages become separated during grading.
- This exam booklet contains **BOTH** the multiple choice and written portions of the exam. Instructions for each portion are provided at the beginning of their respective sections.
- **DO NOT** detach any pages from this booklet.
- There should be 20 pages in this book; if you are missing any pages, please let an instructor know.

Multiple Choice Portion [60 points]

MULTIPLE CHOICE INSTRUCTIONS:

- Select only **ONE** answer per multiple choice question.
- There are 24 multiple choice questions worth 2.5 points each.
- Record your choices for all multiple choice questions using the circles in the boxed area next to each question. Use a number 2 pencil, not a pen (so that you may change your answer). Fill the circle completely. There is no partial credit for multiple-choice questions. Make sure you bubble in the correct letter. See the example below for how to select your answer.



- There is no penalty for wrong answers: it is to your benefit to record an answer to each multiple-choice question, even if you're not sure what the correct answer is.
- The questions vary in difficulty — try not to spend too much time on any one question. Use your time wisely.
- When asked about memory complexity, consider all possible sources (stack and heap).
- The term "heap" as a data structure refers to a binary heap, unless explicitly stated otherwise.
- The term "binary search tree" refers to a standard implementation that does not self-balance, unless explicitly stated otherwise.

Record your answers in the bubbles next to each question.

1. Master Theorem

(A) (B) (C) (D) (E)

Which of the following recurrence relations can one solve by applying the Master Theorem?

A) $T(n) = \frac{1}{2}T(\frac{n}{4}) + \Theta(n)$

B) $T(n) = T(\frac{n}{2}) + \Theta(n^{\log_2 4})$

C) $T(n) = 4T(n-1) + \Theta(1)$

D) $T(n) = 4T(\frac{n}{2}) + 2T(\frac{n}{4}) + \Theta(n^2)$

E) None of the above.

$\log_2 4 = 2$

2. Measuring Runtime

(A) (B) (C) (D) (E)

Consider the following function:

```
1   int func(double m, double n) {
2       int c = 0;
3       while (m > n) {
4           m = m/2;
5           for (int i = 1; i < n; i*=3;)
6               ++c;
7       } // while
8
9       return c;
10  } // func
```

What best characterizes the running time of a call to this function?

A) $O(n \log m)$

B) $O(\frac{m}{n} \log n)$

C) $O((\log \frac{m}{n}) \cdot (\log n))$

D) $O(mn \log(mn))$

E) $O(mn \cdot (\log m) \cdot (\log n))$

Look at how increasing m increases complexity, increasing n decreases complexity. Note that $(\log m) - (\log n) = \log \frac{m}{n}$

Record your answers in the bubbles next to each question.

3. Math Foundations

(A) (B) (C) (D) **(E)**

Which of the following is **NOT** true regarding logarithms?

A) $\log_2(n) = O(\log_3(n))$

B) $\log_3(n) = O(\log_2(n))$

C) $\log_2(n!) = O(n \log_2(n))$

D) $\log_2(n) = \Theta(\log_3(n))$

E) All of the above are true

(A), (B) and (D) are from lecture (dividing billiard balls into 2 piles or 3). (C) relies on the fact that $\log n^r = r \log n$, $O(\log n^n) = O(n \log n)$, and since $n^n > n!$, $O(n \log n)$ is an upper bound on $\log(n!)$.

4. Complexity Analysis

(A) (B) **(C)** (D) (E)

If $f(n) = O(g(n))$, and $f(n) = \Omega(h(n))$ and $f(n)$, $g(n)$, and $h(n)$ are strictly increasing functions, which of the following MUST be true?

A) $g(n) = O(h(n))$

B) $g(n) = \Theta(h(n))$

C) $g(n) = \Omega(h(n))$

D) $g(n) = \Theta(f(n))$

E) None of the above

We know that $g(n)$ is an upper bound on $f(n)$ and that $h(n)$ is a lower bound on $f(n)$. Therefore $h(n)$ must be a lower bound on $g(n)$.

Record your answers in the bubbles next to each question.

5. Recursion

(A) (B) (C) (D) **(E)**

Consider the following recurrence relation:

$$T(n) = \begin{cases} c_0, & n = 1 \\ nT(n-1) + c, & n > 1. \end{cases}$$

What is the complexity of $T(n)$?

A) $\Theta(n \log n)$

B) $\Theta(n^2)$

C) $\Theta(n^3)$

D) $\Theta(c^n)$

E) $\Theta(n!)$

Start the expansion, substituting in: $T(n) = nT(n-1) + c = n(n-1)T(n-2) + nc + c = n(n-1)(n-2)T(n-3) + n(n-1)c + nc + c$ and notice that it is looking like the formula for a factorial (plus some polynomial, which is less important).

6. Queues

(A) (B) (C) (D)

What is the best possible space complexity for a function printing out the contents of a `std::queue` of size n in order, such that the contents and order of the `queue` are the same before and after the function is called and executed? This means the auxiliary, or additional, space required to make this occur, not the space taken up by the data that was in the `queue` to begin with. You can assume that the `queue` is passed by reference.

A) $O(1)$

B) $O(n)$

C) $O(n^2)$

D) $O(n^3)$

Loop n times: print the front, push a copy of the front onto the back, pop the front.

Record your answers in the bubbles next to each question.

7. Code Analysis

(A) (B) (C) (D) (E)

NOTE: Same code as next page, different question

Consider an array `data` of size n which contains 0s and 1s. Each element of this array containing 0 appears before all those elements containing 1. For example, [0, 0, 0, 0, 1] is such an array.

```
1   int myFunction(int data[], int length) {
2       int mid = 0, left = 0; int right = length - 1;
3       if (data[0] == 1)
4           return 0;
5       else if (data[length - 1] == 0)
6           return length;
7       else {
8           bool isFound = false;
9           while (!isFound) {
10              mid = left + (right - left) / 2;
11              if (data[mid] == 0)
12                  left = mid + 1;
13              else {
14                  if (data[mid - 1] == 0)
15                      isFound = true;
16                  else
17                      right = mid - 1;
18              } // else
19          } // while
20          return mid;
21      } // else
22  } // myFunction()
```

In general, what does the function return when it is given such an array?

A) Number of 0's in `data[]`

B) Number of 1's in `data[]`

C) (Number of 0's in `data[]`) + 1

D) (Number of 1's in `data[]`) + 1

E) (Number of 1's in `data[]`) - 1

Binary searches for the first 1 that's preceded by a 0.

Record your answers in the bubbles next to each question.

8. Complexity of Code (A) (B) **(C)** (D) (E)

NOTE: Same code as previous page, different question

Consider an array `data` of size n which contains 0s and 1s. Each element of this array containing 0 appears before all those elements containing 1. For example, [0, 0, 0, 0, 1] is such an array.

```
1  int myFunction(int data[], int length) {
2      int mid = 0, left = 0; int right = length - 1;
3      if (data[0] == 1)
4          return 0;
5      else if (data[length - 1] == 0)
6          return length;
7      else {
8          bool isFound = false;
9          while (!isFound) {
10             mid = left + (right - left) / 2;
11             if (data[mid] == 0)
12                 left = mid + 1;
13             else {
14                 if (data[mid - 1] == 0)
15                     isFound = true;
16                 else
17                     right = mid - 1;
18             } // else
19         } // while
20         return mid;
21     } // else
22 } // myFunction()
```

What is the complexity of the above algorithm?

A) $\Theta(1)$

B) $\Theta(\log(\log n))$

C) $\Theta(\log n)$

D) $\Theta(n)$

E) $\Theta(n \log n)$

Record your answers in the bubbles next to each question.

9. Heaps and Heapsort

(A) (B) (C) (D) **(E)**

Which of the following is **TRUE** about binary heaps and heapsort?

A) Finding the maximum element in a min heap tree has a complexity of $\Theta(\log n)$

B) Turning a random set of elements into a heap has a complexity of $\Theta(\log n)$

C) If the priority of one element in a heap changes, fixing the heap requires a minimum of $\Theta(n)$ operations

D) The worst-case time complexity of heapsort is $\Theta(n^2)$

E) Heapsort can be implemented with $O(1)$ additional space

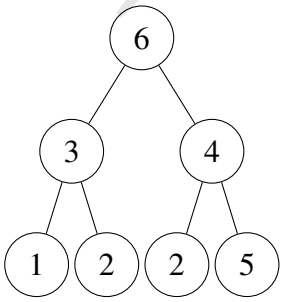
Given an array/vector of values, you can perform the heapify and repeated swap/fixdown in place.

10. Binary Heaps

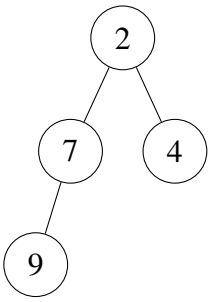
(A) (B) **(C)** (D)

Which of the following represents a valid binary heap?

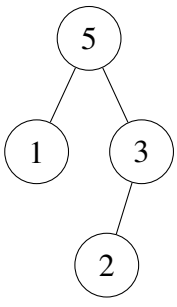
A)



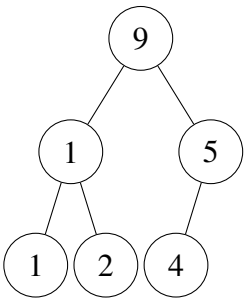
C)



B)



D)



(C) is a valid min-heap. (A) is almost a valid max-heap, but 5 is below 4, (B) is not complete, and (D) has 2 below 1.

Record your answers in the bubbles next to each question.

11. Heaps

(A) (B) (C) (D) **(E)**

Suppose that we wanted to add a function `popRandom()` to our priority queues in Project 2. When implementing it with the binary heap, we could generate a random index, swap values between that index and the last element of the data vector, and call the vector's `pop_back()` function. What else would we have to do to preserve the binary heap property? Choose the answer with the **smallest worst-case** complexity.

A) Call `update_priorities()`.

B) Call `fixDown()` followed by `fixUp()`, both on the random index.

C) Call `fixDown()`, starting at the root of the heap.

D) Call `fixUp()`, starting at the last element of the heap.

E) Call exactly one of `fixUp()` or `fixDown()`, depending on the values swapped.

To pop a value we replace it with the back(), pop_back(), then we have to either fix up or down, depending on whether the value went up or down in priority.

12. Reversing an Array

(A) **(B)** (C) (D)

You are asked to reverse the order of an arbitrary array of length n in constant memory. Your code skeleton looks like:

```
1  for (...) {
2      std::swap(array[i], array[n - i - 1]);
3  }
```

What belongs in the `for` loop header?

A) `for (size_t i = 0; i < n; ++i)`

B) `for (size_t i = 0; i < (n / 2); ++i)`

C) `for (size_t i = n; i >= 0; --i)`

D) None of the above

Swap values, stopping before the middle.

Record your answers in the bubbles next to each question.

13. Linked Lists 1

(A) (B) (C) **(D)**

Which of the following is a reason that you would use a doubly linked list over a vector?

A) You need random access

B) You need perform binary search on a data set

C) You need to sort a large number of characters

D) You're given a pointer to an element and need to insert an element right in front of it

Slightly outdated question from when we had a lecture on linked lists. You need to modify the "previous" node to point to this node's "next" node, which is fastest if each node has a previous pointer.

14. Linked Lists 2

(A) (B) **(C)** (D)

For which operation does a doubly linked list outperform a singly linked list? Assume the worst case, and that neither version of the linked list contains a tail pointer.

A) Accessing element at random index

B) Appending an element to the end

C) Deleting an element given a pointer to the node that contains the element

D) Swapping two elements that are not next to each other, given pointers to the two nodes

(A) cannot be done efficiently in either version of a linked list. (B) needs a tail pointer to be efficient, not doubly-linked (unless it's BOTH doubly- and circularly linked). (C) without the previous pointer of a doubly-linked list, you must search from the beginning to find which node's next pointer points to the node you want to delete. In (D) you can just swap(a->elt, b->elt).

Record your answers in the bubbles next to each question.

15. Array-based Containers

(A) (B) (C) (D)

Suppose we implement a vector with an underlying dynamic array and a special `push_back()` method. This `push_back()` method works normally when the array is not full. When the array fills up, the `push_back()` method creates a new dynamic array with space for 100 more elements than the old array, copies the elements from the old array to the new array, and then deletes the old array. What is the amortized complexity of this special `push_back()` operation?

A) $O(1)$

B) $O(\log n)$

C) $O(\sqrt{n})$

D) $O(n)$

Growing the vector takes time n, then we get 100 pushes at a cost of $O(1)$ each. So the amortized cost is $\frac{n}{100} = O(n)$.

16. Container Data Structures

(A) (B) (C) (D)

Which data structure is best for implementing a sorted container that supports binary search?

A) Stack

B) Vector

C) Linked List

D) Binary Heap

Record your answers in the bubbles next to each question.

17. Contiguous Containers

(A) (B) (C) (D)

Which of the following statements about native (C-style) array and `std::vector` is **TRUE**?

A) A `vector` can store references while a native array cannot.

B) It is valid for a function to return a `vector` declared on its own stack, but invalid to return an array declared on its own stack

C) Since a `vector` stores data on the heap, you must always explicitly destroy them using the `delete` operator

D) A `vector` may be multi-dimensional, an array cannot

In (B), the vector allocates memory on the heap and has its own member functions to make sure that when the function ends, you get a copy of the object that still points to the same place on the heap. When a function ends, a local array is on the stack, and the stack frame disappears, invalidating a pointer to that block of memory.

18. The STL

(A) (B) (C) (D)

Which of the following statements is **TRUE**?

A) If you define a class with a `std::vector` member variable, you must define the Big Three (copy constructor, assignment operator, destructor) for the class because vectors use dynamic memory

B) STL container classes (such as `std::vector`) may only contain objects, not primitive types

C) To use a `std::priority_queue<T>`, type `T` must support `operator<()`

D) None of the above

If all we specify is the type T and nothing else, the stored type must either be a primitive type on which < can be performed, or be a class that overloads operator<(). Either way it supports operator<().

Record your answers in the bubbles next to each question.

19. Ordered Arrays and Related Algorithms

(A) (B) (C) (D)

Given a C-string, what is the best space complexity of an algorithm to determine whether the string contains duplicate characters?

A) $O(1)$

B) $O(n)$

C) $O(n \log n)$

D) $O(n^2)$

Doubly-nested loop is $O(n^2)$ but just need 2 loop variables and a count in memory.

20. Ordered Arrays

(A) (B) (C) (D) (E)

Given two sorted arrays, each with n numbers, you are asked to find the median of all of the elements from the two sorted arrays. What is the time complexity of the best solution?

A) $O(1)$

B) $O(\log n)$

C) $O(n)$

D) $O(n \log n)$

E) $O(n^2)$

Think of this as performing a binary search on two sorted arrays at once. Each comparison can tell you about a portion of one of the arrays that you can ignore. See the second version at `http://www.geeksforgeeks.org/median-of-two-sorted-arrays/`

Record your answers in the bubbles next to each question.

21. Sorting

(A) (B) **(C)** (D)

On a nearly-sorted singly linked list, which of the following sorting algorithms is asymptotically fastest on large input?

A) Heap sort

B) Insertion sort

C) Merge sort

D) All of these would be linear

Not the best question, since Insertion sort should do well when the contents are nearly sorted, but cannot approach $n \log n$ in the worst case. We were concentrating too much at the "linked list" part. We would accept either (B) or (C).

22. Quicksort

(A) (B) (C) **(D)**

Suppose we run quicksort on the following list [4, 12, 9, 1, 2, 5, 3, 8, 7]. We choose the last element, 7, as the pivot. After partitioning as described in lecture (but **BEFORE** we run the recursive function calls) what is the resulting list?

A) [4, 3, 5, 1, 2, 7, 12, 9, 8]

B) [4, 3, 5, 1, 2, 9, 12, 8, 7]

C) [4, 3, 1, 5, 2, 7, 12, 8, 9]

D) [4, 3, 5, 1, 2, 7, 12, 8, 9]

Run through the algorithm as presented in class.

Record your answers in the bubbles next to each question.

23. Sorting Students

(A) (B) (C) (D) (E)

Given the following code:

```
1   enum class Standing_e : int { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };
2
3   struct Student {
4       int age;
5       Standing_e standing;
6       double gpa;
7   };
8
9   bool operator<(const Student& s1, const Student& s2) const {
10      return static_cast<int>(s1.standing) < static_cast<int>(s2.standing);
11  }
```

Given an array of `Student` objects of size n (where n is very large), what would be the best possible worst-case time complexity to sort this array by class standing?

A) $\Theta(n)$

B) $\Theta(n \log n)$

C) $\Theta(n^2)$

D) $\Theta(n^3)$

E) None of the above

This is what Counting Sort is great at: sorting by a key that has a very limited number of possible values, which results in $O(n)$ sorting time.

24. Elementary Sorts

(A) (B)

Executing a selection sort algorithm on a singly linked list has an asymptotically worse average-case time complexity than executing a selection sort on an array with random access.

A) True

B) False

It's going to be $O(n^2)$ either way.

Written Portion [40 points]

WRITTEN PORTION INSTRUCTIONS:

- There are **two** questions in this section, with the following point distribution:

Question 25	20 points
Question 26	20 points

- Please write your solution legibly in the space provided. Solutions that are difficult to read may receive fewer points.
- Use the back of the question page to work out solutions for each problem and then rewrite your final answer in the space provided.
- The directions for each problem will specify which STL algorithms/functions may be used in your solution.
- Solutions that exceed the allowed line count will lose one (1) point per line over the limit.
- Partial credit will be awarded for partial solutions.
- Errors in program logic or syntax may result in a reduced score.
- Be sure to consider all possible sources of memory (stack and heap) and time complexity.

---

25. Programming: Linear-time Algorithms and the STL [20 Points]

Implement STL's `unique_copy()` function according to its official interface and description.

```
template <class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                           OutputIterator result);
```

Quoting from `http://www.sgi.com/tech/stl/unique_copy.html`:

> `unique_copy()` copies elements from the range `[first, last)` to a range beginning with `result`, except that in a consecutive group of duplicate elements only the first one is copied. The return value is the end of the range to which the elements are copied.

Complexity: Linear. For elements in the ranges, exactly `last - first` applications of `operator==()` and at most `last - first` assignments.

For example, if you executed this code:

```
int data[6] = {1, 3, 3, 1, 1, 0}, output[6];
unique_copy(data, data + 6, output);
```

You would have this in array `output`:

1	3	1	0		

Implementation: Use the back of this page as a working area, then rewrite **neatly** on the front. Limit: 15 lines of code (points deducted if longer). You may **NOT** use other STL algorithms/functions.

```
template <class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                           OutputIterator result) {

    if (first == last)
        return result;

    *result++ = *first; // establish a starting point
    ForwardIterator prev = first++;

    while (first != last) {
        if (!(*first == *prev)) // *first != *prev
            *result++ = *first;
        prev = first++;
    } // while
    return result;
}
```

This page is intentionally left blank.
You may use this page as working space.

26. Programming: Finding Max Elements [20 Points]

Suppose you are given an array of `int`, size `n` and a number `k`. Return the k largest elements Output does not need to be sorted. You can assume that $k < n$.

Requirements: Your solution **must be faster than** $O(n \log n)$ time. You may use up to $O(n)$ auxiliary space.

Implementation: Use the back of this page as a working area, then rewrite **neatly** on the front. Limit: 15 lines of code (points deducted if longer). You may use any C, C++, or STL function/algorithm that you wish.

```
vector<int> findKMax(int arr[], size_t n, size_t k) {
    // O(n + k log n) solution: Create a max-PQ of everything,
    // which is O(n).  Then pop off the k largerst items:
    // k operations, each of which is O(log n)
    priority_queue<int> myPQ(arr, arr + n);
    vector<int> output;
    output.reserve(k);
    for (size_t i = 0; i < k; ++i) {
        output.push_back(myPQ.top());
        myPQ.pop();
    } // for
    return output;
} // findKMax()

// O(k + (n-k) log k) solution: create a min-PQ of the first k items, in O(k).
// Then n-k times, push one item and pop off the SMALLEST item, leaving
// the k largest remaining: (n-k) * O(log k), since the PQ is of size k.
vector<int> findKMax2(int arr[], size_t n, size_t k) {
    // If it was k smallest, would be simpler declaration
    priority_queue<int, vector<int>, std::greater<int>> myPQ(arr, arr + k);
    vector<int> output;
    output.reserve(k);
    for (size_t i = k; i < n; ++i) {
        myPQ.push(arr[i]);
        myPQ.pop();
    } // for
    while (!myPQ.empty()) {
        output.push_back(myPQ.top());
        myPQ.pop();
    } // while
    return output;
} // findKMax2()
```

This page is intentionally left blank.
You may use this page as working space.