Midterm Exam Solutions

- Don't Panic. Keep calm and take it a step at a time.
- Write your student id *clearly* at the bottom of this page and on the top right on every odd page (Do this now).
- The midterm exam contains 5 problems. You have 90 minutes to earn 90 points.
- The midterm exam is closed book. You may bring one double-sided sheet of A4 paper to the midterm exam. (You may not bring any magnification equipment!) You may **not** use a calculator, your mobile phone, or any other electronic device.
- Write your solutions in the space provided. If you need more space, please use the scratch paper at the end of the midterm. Do not put part of the answer to one problem on a page for another problem.
- When writing your solutions, you can use the algorithms and data structures we have discussed in class without describing them in detail. Unless you make a modification, you do not have to describe how the standard methods work.
- Read through the problems before starting. Do not spend too much time on any one problem.
- Show your work. Partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- If you finish early, drop off your midterm in the front, and leave quietly.
- You may not bring any part of this midterm (even the scratch pages) out of the room.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	True or False	15	
2	Herbert Goes to Space	10	
3	Algorithm Analysis	20	
4	Permutations	20	
5	Tree to Tape	25	
Total:		90	

Student id.: _____

IMPORTANT: Please ensure your student id is correct and legible.

Please circle your discussion group:

Si Jie 10am-12pm B110	Eldon 10am-12pm 0113	Shaowei 10am-12pm B111	Nicholas 12am-2pm B110	Advay 12am-2pm B111	Irham 2am-4pm 0120	David 4am-6pm B109	Enzio 4am-6pm B111	Jerrell 4am-6pm B108

Problem 1. True or False. [15 points] **Grading policy:** 1 point per correct answer.

Problem 1.a. Write (T)rue or (F)alse for the statements below about Mergesort.

☐ — **[False]** Mergesort is *not* an example of a divide-and-conquer algorithm.

☐ — **[True]** The merge step takes $O(n)$ time .

☐ — **[False]** Replacing the merge step with Hoare's partitioning algorithm without modifying the remainder of the mergesort algorithm will still produce a valid sort.

☐ — **[True]** The worst case computational complexity mergesort is $O(n^2)$.

☐ — **[False]** Mergesort is in-place but is not stable.

Problem 1.b. Write (T)rue or (F)alse for the statements below about a (binary) **min-heap**.

☐ — **[False]** The root is the largest element in the min-heap.

☐ — **[True]** For every node in min-heap that has two children, the value of the node must be smaller than or equal to the value of its children.

☐ — **[False]** When a heap is stored in an array, the array is sorted from smallest to largest.

☐ — **[False]** Given a min-heap, you can output a sorted list of all the values in *decreasing* order in $O(n)$ time.

☐ — **[True]** Given an unsorted array of values, you can build a heap in $O(n)$ time.

Problem 1.c. Write (T)rue or (F)alse for the statements below about AVL trees.

☐ — **[True]** If key u is in an AVL tree, you can find the successor of u (or report no successor) in $O(\log n)$ time.

☐ — **[False]** The left and right subtrees of an AVL tree are themselves height-balanced.

☐ — **[False]** The rightmost element in an AVL tree is the smallest element.

☐ — **[False]** Given an AVL tree, you can output a sorted list of all the keys in $O(n \log n)$ time using in-order traversal.

☐ — **[True]** If a node in an AVL tree has two children, its predecessor is the rightmost node in its left subtree.

Problem 2. Herbert Goes to Space. [10 points]

Our favorite robotic clown Herbert wants to buy a trip on the GalaticX spaceship (designed by the famous Elan Mosk) to one of n possible planets. Distances to the n planets are stored in an array D such that $D[i]$ gives the distance of planet i from from Earth. Herbert wants to travel as far as possible, but doesn't have much money (he's lazy, remember?). Using all his SmileBucks savings, he can only afford a trip to the k-th closest planet. For this problem, assume all distances to be unique.

Problem 2.a. (4 points) **Describe the most time efficient algorithm you can think of to help Herbert find the k-th closest planet.** Be precise, but pseudocode or Java is not necessary unless it helps you to explain. You can assume you already have access to all the algorithms and data structures we have discussed in class. Unless you make a modification, you do not have to describe how the standard methods work. Write the time and space complexity of your method below:

Running time: [ ] Space: [ ]

Your algorithm with explanation:

Solution: This is our usual top-k question. There are multiple solutions. Award 3 points for the solution and 1 points for the time and space complexity (0.5 points each). Mark the time and space complexity separately from the solution, i.e., if they get the space/time complexity correct, give them the points.
Wrong solution but an attempt (0.5 point):
Correct but very inefficient (1 points): $O(n^2)$ sort (bubble, not quick), or $O(kn)$ solutions.
Correct but not very efficient (2 points): Sort and get the k-th element. Time $O(n \log n)$ if they use a fast sort.
Correct and more efficient methods (3 points): Quickselect or Heap based solutions qualify. For heap, it would be $O(n + k \log n)$. They could also use a k-sized heap $O(n \log k)$. Either one would suffice here.

The GalacticX space program has just increased the number of planets accessible to $n > 10^{12}$. This is much too large to fit into Herbert's spare memory, which can only store up to $O(k)$ elements. Instead, Herbert has access to the following functions:

- `long getNumPlanets()`: returns total number of planets accessible via GalacticX.
- `long getDistance(long i)`: returns the distance of the planet i from Earth.

Problem 2.b. (6 points) **Describe the most time-efficient algorithm you can think of to find the k-the closest planet, given that you can only use $O(k)$ space.** Be precise, but pseudocode or Java is not necessary unless it helps you to explain. You can assume you already have access to all the algorithms and data structures we have discussed in class. Unless you make a modification, you do not have to describe how the standard methods work. Write the time and space complexity of your method below:

Running time: Space:

Your algorithm with explanation:

Solution: 4 points for the solution and 2 points for the analysis (1 point time, 1 point space). Again, grade the
Wrong solution but attempt made (0.5 points).
Violations of the constraint (1 point). The solution works, but it violates the restriction to store only $O(k)$ elements. Examples including sorting all n elements or a heap with n elements.
Inefficient solution (2 points); not using a heap and keeping storing the k minimum points $O(nk)$.
4 points for a k-sized heap, which takes $O(n \log k)$ time, and $O(k)$ time. Alternative correct solutions that also take $O(n \log k)$ (or better?) also work.

Problem 3. Algorithm Analysis [20 points]
For each of the following, choose the best (tightest) asymptotic function T from among the given options. Some of the following may appear more than once, and some may appear not at all.

A. $O(1)$	B. $O(\log n)$	C. $O(n)$	D. $O(n \log n)$
E. $O(n^2)$	F. $O(n^3)$	G. $O(2^n)$	H. None of the above.

Problem 3.a.

$$T(n) = \frac{3n^7 - 4n - 17}{2n^5}$$

$T(n) =$ **Solution:** E : $O(n^2)$

Problem 3.b.

$$T(n) = 3n^2 + \sum_{i=1}^{n!} \frac{n}{5^i}$$

$T(n) =$ **Solution:** E : $O(n^2)$

Problem 3.c. The running time of the following code, as a function of n:

```
public static int calcj(int n){
    int j = 0;
    while (n > 0) {
        for (int i=0; i<n-1; i++) {
            j++;
        }
        n--;
    }
    return j;
}
```

$T(n) =$ **Solution:** E : $O(n^2)$

Problem 3.d.

$$\begin{aligned} T(n) &= T\left(\frac{2n}{10}\right) + T\left(\frac{3n}{10}\right) + T\left(\frac{5n}{10}\right) + 2n \\ T(1) &= 1 \end{aligned}$$

$T(n) =$ **Solution:** D : $O(n \log n)$

Problem 4. Permutations [20 points]
For this problem, you are given an array $A = [0, 1, 2, 3, \ldots, n-1]$ of size n. Your goal is to generate all possible permutations of A. For example, given $A = [0, 1, 2]$ ($n = 3$), your program should output:

```
[0,1,2]
[0,2,1]
[1,0,2]
[1,2,0]
[2,0,1]
[2,1,0]
```

Your algorithm need **not** follow the exact sequence of permutations above, but it should print out each of the different permutations exactly once.

Problem 4.a. (8 points) **Describe a time-efficient algorithm for enumerating the all the permutations of A.** Be brief but precise. In addition to your explanation, **provide pseudocode or java code**. You can assume you already have access to all the algorithms and data structures we have discussed in class. Unless you make a modification, you do not have to describe how the standard methods work. State the time and space complexity of your method.

Running time: [ ] Space: [ ]

Explanation with Pseudocode/Java code:

Solution: 6 points for the solution, 2 points for time running time and space complexity (1 point each).
We can do this is $O(n!n)$ since it takes $O(n)$ time to print a string and there are $n!$ permutations. A simple straightforward solution is a recursive algorithm, but you can implement an iterative version as well. Example code is available online at `https://www.programcreek.com/2013/02/leetcode-permutations-java/`). 6 points for a correct solution.
If they make minor errors, take away 0.5-1.5 points. Be generous as long as they are on the right track.
Completely wrong solutions (but an attempt) get 0.5 points. I'm unsure what correct solutions there are with horrible efficiency (worse than $O(n!n)$) but those would get at most 3 points.

Problem 4.b. (12 points) We can impose a total order on permutations using dictionary ordering. A permutation P is less than a permutation Q, denoted as $P < Q$, if in the first place i (starting from index 0) that P and Q differ, $P[i] < Q[i]$. As an example, `[0,1,2] < [0,2,1]` since $P[1] < Q[1]$. The smallest permutation under dictionary ordering is `[0,1,2]` and the largest is `[2,1,0]`.

Given a permutation P, find the *next largest* permutation under dictionary ordering. For example, given `[1,2,0]`, your program should output `[2,0,1]`. Describe the most time efficient algorithm you can think of. Be precise, but pseudocode or Java is not necessary unless it helps you to explain. What is the time and space efficiency of your algorithm?

Running time: Space:

Your algorithm with explanation:

Solution: 10 points for the solution, 2 points for time running time and space complexity (1 point each). This part is harder and can be difficult to figure out in a short time span. The algorithm itself is pretty well-known; there's even a Wikipedia article on it `https://en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order`. From the article:
The following algorithm generates the next permutation lexicographically after a given permutation. It changes the given permutation in-place.

a. Find the largest index k such that $a[k] < a[k+1]$. If no such index exists, the permutation is the last permutation.

b. Find the largest index l greater than k such that $a[k] < a[l]$.

c. Swap the value of $a[k]$ with that of $a[l]$.

d. Reverse the sequence from $a[k+1]$ up to and including the final element $a[n]$.

This takes $O(n)$ time and $O(1)$ space. This solution gets a full 10 points. If they make a small error, e.g., sorting in the final step, they get a point knocked off.
A completely wrong solution (but an attempt) gets 1 point.
A correct but less efficient solution gets a maximum of 7 points. There could be a range here so, we'll discuss.

Problem 5. Tree to Tape [25 points]

Naruto was exploring the office and found a big box of tape storage! Before the advent of hard disks and solid-state drives (SSDs), data used to be stored on magnetic tape. It might surprise you to learn that tape is still being used today; compared to other data storage technologies, tape is cheap and can last decades.

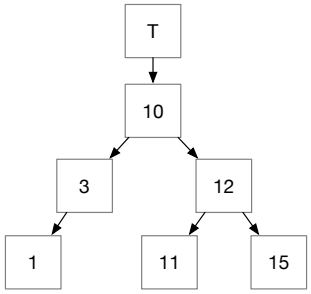
Figure 1: A Magnetic Tape Drive. (Image Credit: Wikipedia)

For this problem, we want to save binary search trees (BSTs) to tape for long-term storage. But one limitation of tape drives is their *sequential access* nature; sequential data transfer is fast, but random access is very slow. So, we want to convert a given BST to a *sorted doubly linked list*, which is more easily written to (and read from) the tape drive. Assume you have a standard BST with the following node structure:

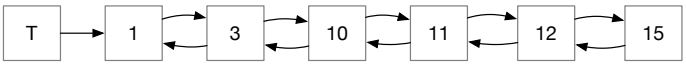
```
class Node {
    Node left
    Node right

    int key
    Object value
}
```

As a specific example, given a tree T:



Your method should create the following sorted doubly linked list:



You can reuse the nodes in the BST by using `left` as the `prev` reference and `right` as the `next` reference in the doubly linked list. Recall that in a doubly linked list, `prev` points to the previous node, and `next` points to the next node.

Problem 5.a. (10 points) **Describe an efficient method that converts a Binary Search Tree (BST) to a doubly-linked list.** Be brief but precise. For this problem, assume that the keys are *unique*. Explain the time and space complexity of your algorithm.

Hint: *It is possible to solve this problem in $O(n)$ time and use less than $O(n)$ space.*

Running time: [ ] Space: [ ]

Your algorithm with explanation:

Solution: points for the solution, 2 points for the time and space analysis (1 each).
Completely wrong solutions get 0.5 points (an attempt).
A (strange) inefficient solution might be to copy to an array, then perform a sort, then create a doubly linked list. This gets 3 points at most.
The inefficient solution is to simply create another doubly linked list by copying the elements as they are encountered during an inorder traversal. This takes $O(n)$ time and $O(n)$ space. Students proposing this get 5 points. (Hopefully, most students will see this solution).
The extra 3 points will go to students who modify the BST in place. This takes $O(\log n)$ space (on the stack). There are two ways to do this. The first relies on inorder traversal (e.g., `https://articles.leetcode.com/convert-binary-search-tree-bst-to/` but without the circular references).
The second approach is to use divide and conquer. See http://cslibrary.stanford.edu/109/TreeListRecursion.html

Now that we have managed to store our BST to tape, we need a way to convert the stored doubly linked list back to a BST. One trivial way might be just to use the linked list as a lop-sided BST. But this is inefficient since many operations will take $O(n)$ time. Instead, we want to convert the doubly linked list to a *height-balanced* BST.

Problem 5.b. (15 points) **What is a an efficient way to convert the data back from a doubly linked list to a *height-balanced* BST?** Provide a description for your algorithm. Be precise, but pseudocode or Java is not necessary unless it helps you to explain. What is the worst-case time and space complexity of your method in terms of the number of nodes n?

Hint: *It is possible to solve this problem in $O(n)$ time and use less than $O(n)$ space.*

Running time: Space:

Your algorithm with explanation:

Solution: 13 points for the solution, 2 points for the analysis (1 for time, 1 for space).
Completely wrong solutions, but an attempt (0.5 points).
A simple, but inefficient solution is to simply insert the elements one by one into an AVL tree. This takes $O(n \log n)$ time, and $O(n)$ space. This approach (and similar) gets 5 points.
Two solutions are given in: `https://www.geeksforgeeks.org/in-place-conversion-of-sorted-dll-to-balanced-bst/`
One approach is to construct the tree recursively from root to leaves. The approach is simple: make the middle element the root, then recursively perform the operation for the left subtree (left half of the linked list) and the right subtree (right half of the linked list). This takes $O(n \log n)$ time, but you can do this in place to use $O(\log n)$ space. This gets 8 points.
The second approach is to construct the tree "upwards" from the leaves, by recursing on the left and right portions. Code on next page. This takes $O(n)$ time and $O(\log n)$ space. Full 13 points for this one.

Solution: Example code:

```
/* This function counts the number of nodes in Linked List
   and then calls sortedListToBSTRecur() to construct BST */
Node sortedListToBST()
{
    /*Count the number of nodes in Linked List */
    int n = countNodes(head);
    /* Construct BST */
    return sortedListToBSTRecur(n);
}

/* The main function that constructs balanced BST and
   returns root of it.
   n  --> No. of nodes in the Doubly Linked List */
Node sortedListToBSTRecur(int n)
{
    /* Base Case */
    if (n <= 0)
        return null;
    /* Recursively construct the left subtree */
    Node left = sortedListToBSTRecur(n / 2);
    /* head_ref now refers to middle node,
       make middle node as root of BST*/
    Node root = head;

    // Set pointer to left subtree
    root.prev = left;

    /* Change head pointer of Linked List for parent
       recursive calls */
    head = head.next;

    /* Recursively construct the right subtree and link it
       with root. The number of nodes in right subtree  is
       total nodes - nodes in left subtree - 1 (for root) */
    root.next = sortedListToBSTRecur(n - n / 2 - 1);

    return root;
}
```

Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper