Quiz 1 Solutions

- Don't Panic.
- Write your name on every page.
- The quiz contains seven problems. You have 100 minutes to earn 100 points.
- The quiz contains 16 pages, including this one and 2 pages of scratch paper.
- The quiz is closed book. You may bring one double-sided sheet of A4 paper to the quiz. (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 quiz. Do not put part of the answer to one problem on a page for another problem.
- 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.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	Sorting Jumble	10	
2	Algorithm Analysis	16	
3	Barley Mow	10	
4	Buggy Code I	16	
5	Matrix Math	14	
6	Buggy Code II	14	
7	Mind the Gap	20	
Total:		100	

Name: _____ Matric. Num.: _____

Please circle your discussion group:

Pallav Tues. 2pm	Davin Tues. 2pm	Mingyu Tues. 4pm	Sharon Tues. 4pm	Shi-Jie Thurs. 10am	Yik Jiun Thurs. 10am	Xuanyi Thurs. 12pm	Chuyu Thurs. 4pm

Problem 1. Sorting Jumble. [10 points]
Your goal is to identify which sorting algorithm is being used in which case. The first column in the table below contains an unsorted list of telephones. The last column contains a sorted list of telephones. Each intermediate column contains a partially sorted list of telephones.

Each intermediate column was constructed by beginning with the unsorted list at the left and running one of the sorting algorithms that we learned about in class, stopping at some point before it finishes. Identify, below, which column was (partially) sorted with which algorithm.

Hint: Do not just execute each sorting algorithm, step-by-step, until it matches one of the columns. Instead, think about the invariants that are true at every step of the sorting algorithm.

Unsorted	**InsertionS**	**MergeS**	**QuickS**	**SelectionS**	**BubbleS**	**Sorted**
OnePlus	Apple	Apple	Motorola	Apple	Apple	Apple
Vodafone	Ericsson	Ericsson	LG	Asus	Ericsson	Asus
Samsung	HTC	HTC	Blackberry	Blackberry	HTC	Blackberry
Yotaphone	Nokia	Nokia	Huawei	Ericsson	Asus	Ericsson
Nokia	OnePlus	OnePlus	HTC	HTC	Kyocera	HTC
HTC	Samsung	Samsung	Apple	Nokia	Huawei	Huawei
Apple	Vodafone	Vodafone	Ericsson	OnePlus	Blackberry	Kyocera
Ericsson	Yotaphone	Yotaphone	Kyocera	Yotaphone	LG	LG
Kyocera	Kyocera	Asus	Asus	Kyocera	Nokia	Motorola
Asus	Asus	Kyocera	Nokia	Vodafone	Motorola	Nokia
Sony	Sony	Huawei	OnePlus	Sony	OnePlus	OnePlus
Huawei	Huawei	Sony	Sony	Huawei	Samsung	Samsung
Blackberry	Blackberry	Blackberry	Yotaphone	Samsung	Sony	Sony
LG	LG	LG	Samsung	LG	Vodafone	Vodafone
Xiaomi	Xiaomi	Xiaomi	Xiaomi	Xiaomi	Xiaomi	Xiaomi
Motorola	Motorola	Motorola	Vodafone	Motorola	Yotaphone	Yotaphone
Unsorted	**A**	**B**	**C**	**D**	**E**	**Sorted**

A ___ 1. BubbleSort
B ___ 2. SelectionSort
C ___ 3. InsertionSort
D ___ 4. MergeSort (top down)
E ___ 5. QuickSort (with first element pivot)

Problem 2. Algorithm Analysis [16 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. $\Theta(1)$	B. $\Theta(\log n)$	C. $\Theta(n)$	D. $\Theta(n \log n)$
E. $\Theta(n^2)$	F. $\Theta(n^2)$	G. $\Theta(2^n)$	H. None of the above.

Problem 2.a.

$$T(n) = \frac{n^6 - 4n^2 + 2n - 17}{2n^4 + 2n + 2020}$$

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

Problem 2.b.

$$T(n) = \sum_{i=1}^{n} \frac{n}{i}$$

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

Problem 2.c.

$$\begin{aligned} T(n) &= 6T(n/6) + 6n \\ T(1) &= 1 \end{aligned}$$

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

Problem 2.d. The running time of the following code, as a function of n:

```
public static int loopy(int n){
    int j = 0;
    while (n > 0) {
        j++;
        n /= 2;
    }
    return j;
}
```

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

Problem 3. Jeff Erickson's Barley Mow [10 points]

Jeff Erickson is a professor at UIUC in Illinois. One of Jeff Erickson's favorite songs is "The Barley Mow," which he loves to sing to his algorithms classes. The "Barley Mow" is a traditional Devonian/Cornish drinking song that can be constructed according to the following algorithm, as adapted by Jeff Erickson. Assume that `container[i]` is the name of a container[1], and that it holds 2^i ounces of your favorite beverage.

```
BarleyMow(n)

  Here's a health to the barley-mow, my brave boys,
  Here's a health to the barley-mow!
  We'll drink it out of the jolly brown bowl,
  Here's a health to the barley-mow!

  For i = 1 to n do
    We'll drink it out of the container[i], boys,
    Here's a health to the barley-mow!
    For j = i downto 1 do
        The  container[j],
    And the jolly brown bowl!
    Here's a health to the barley-mow, my brave boys,
    Here's a health to the barley-mow!
```

Problem 3.a. Suppose each container name `container[i]` is a single word, and you can sing four words a second. How long would it take you to sing `BarleyMow(n)`? Give a tight asymptotic bound.

Solution: $\Theta(n^2)$
(The cost is dominated by the inner loop, which is equivalent to $\sum_{i=1^n} \sum_{j=1}^{i} O(1) = \Theta(n^2)$. Everything outside the loops is $O(1)$, and everything in the outer loop adds up to $O(n)$.)

[1] One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon,half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

Problem 3.b. Suppose each time you mention the name of a container, you drink the corresponding amount of your favorite beverage: one ounce for the jolly brown bowl, and 2^i ounces for each container[i]. Exactly how many ounces would you drink if you sang BarleyMow(n)? (Give an exact answer, not just an asymptotic bound.)

Solution: $3 \cdot 2^{n+1} - n - 5$
The inner for-loop (including the "brown bowl" that follows it) has volume $\sum_{j=0}^{i} 2^j = 2^{i+1} - 1$. Summing this as i goes from 1 to $n4(i.e.,$over the outer for-loop), this yields $2^{n+2} - n - 4$. The additional continers in the outer for-loop and the brown bowl outside the for loop add another $2^{n+1} - 1$, yielding a total of: $3 \cdot 2^{n+1} - n - 5$.

Problem 3.c. *Extra credit: 1 point*
If you want to sing this song for $n > 20$, you will have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases[2]. Suppose `container[n]` has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing `BarleyMow(n)`? Give a tight asymptotic bound. *Hint: Recall that* $\log(n!) = \Theta(n \log n)$.

Solution: $\Theta(n^2 \log n)$
The problem reduces to $\sum_{i=1}^{n} i \log i$, where $\Theta(i \log i)$ is the cost of the inner for-loop. This yields: $\Theta(n^2 \log n)$

[2]We'll drink it out of the hemisemidemiyottapint, boys!

Problem 4. Buggy Code: AddEmUp. [16 points]

As part of your work for HalfruntCorp, you have been asked to debug the following code written by your predecessor Jeltz who was recently fired. Strangely, it does not seem to work.

```
1:  public class AddEmUp
2:  {
3:      public static void sumThings(int[] values)
4:      {
5:           int n = values.length - 1;
6:           int sum = 0;
7:           while (n >= 0) {
8:                int tempValue = values(n);
9:                if (tempValue < 0);
10:                   tempValue = -1*tempValue;
11:               sum += tempValue;
12:          }
13:          return sum;
14:     }
15:
16: }
```

Problem 4.a. There are several bugs that will *prevent this program from compiling.* Only list bugs that prevent compiling (not warnings, bad coding style, or bugs that yield the wrong answer.) Identify (in fifteen words or less, each) two such problems with this code. Specify the line number.

Solution: line 8: `values(i)` should be `values[i]`, since it is an array.

Solution: line 3: Signature specifies return `void`, but method returns `int`.

Problem 4.b. Once you have fixed those compilation errors, there are *still* several bugs that keep the code from working as intended. The program is supposed to calculate the sum of the absolute values of the integers in the array. Identify (in fifteen words or less, each) two such problems with this code.

Solution: line 9: There is an extra semicolon after the `if` clause. The next line is always executed.

Solution: The while loop is infinite, as `n` never decreases.

Problem 5. Matrix Math. [14 points]

Assume you are given an $n \times n$ matrix of integers M as input:

`int[][] M` =

17	2	11	12
12	9	12	9
9	14	7	12
4	17	12	9

Your job is to find the maximum value that appears in every row. For example, in the matrix above, the answer is 12: the value 12 appears in every row, and no larger value appears in every row.

Describe the most time-efficient algorithm you can think of to solve this problem: (Be precise and detailed, but pseudocode or Java is not necessary unless it helps you to explain. You can assume you already have access to all the algorithms we have discussed in class. You do not need to restate how they work.)

Solution:
Step 1. Sort each row. This takes $O(n \log n)$ time per row, and hence $O(n^2 \log n)$ time total.
Step 2. For each item x in row 1, from largest to smallest: perform a binary search for x in every other row; if x appears in every other row, then return x. For each item, the binary search takes $O(\log n)$ time per row. Since there are n rows, this yields $O(n \log n)$ time. Since you may have to check all n items, this takes $O(n^2 \log n$ time.

Solution: $O(n^2 \log n)$

Problem 6. Buggy Code: Urban Farming. [14 points]

Problem 6.a. Consider the following code:

```
1.   public class Vegetables extends Food implements IGreen, IPurple  {
2.           private static int numVegetables = 0;
3.           private int vegetableIndex = -1;
4.
5.           Vegetables(){
6.                   super(numVegetables);
7.           }
8.
9.           public void checkVeggies(){
10.                 if (numVegetables > 10)
11.                         System.out.println("Too many vegetables!");
12.         }
13.
14.         public static int countVeggies(){
15.           if (vegetableIndex > 0)
16.             return numVegetables;
17.           else return 0;
18.         }
19.
20.         @Override // for IGreen
21.         public int getGreen(){
22.                 return vegetableIndex;
23.         }
24.
25.         @Override // for IPurple
26.         public int getPurple(){
27.                 return -1;
28.         }
29.  }
```

Briefly describe **one** bug that will **prevent this from compiling** and explain **why** it is a problem. (Assume that class Food, interface IGreen, and interface IPurple are properly defined. Note that the `@Override` usage is correct.)

Solution: The static method countVeggies cannot access the non-static member variable vegetableIndex because it is a member variable associated with a specific instance while the static method is associated with the class.

Problem 6.b. Consider the following code:

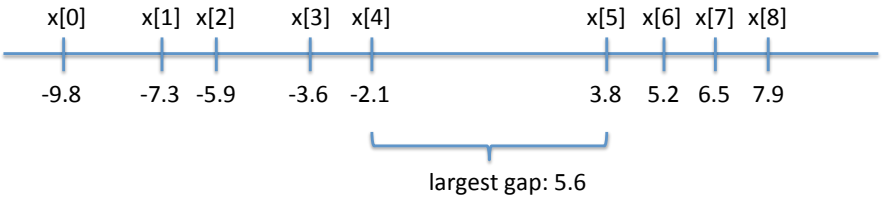
```
1.   public class Building {
2.        public void washFruit() {
3.             System.out.println("Washing fruit and vegetables.");
4.        }
5.   }
6.
7.   public class GreenHouse extends Building {
8.        public void cleanVegetables() {
9.             System.out.println("Cleaning the vegetables.");
10.       }
11.  }
12.
13.  public class UrbanFarm {
14.       public static void doChores(GreenHouse a, GreenHouse b)
15.       {
16.            a.washFruit();
17.            b.cleanVegetables();
18.       }
19.
20.       public static void main(String[] args) {
21.            Building b = new GreenHouse();
22.            GreenHouse h = new GreenHouse();
23.            doChores(b, h);
24.       }
25.  }
```

Briefly describe **one** bug that will **prevent this from compiling** and explain **why** it is a problem.

Solution: The `doChores` method is specified to take two `GreenHouse` variables. But Building `b` is not a `GreenHouse`. Even though the object was created as a GreenHouse, it is stored in a Building variable and hence can only be used as a Building. (Notice that it is perfectly legal to store the GreenHouse in a Building variable due to subclass substitutivity.)

Problem 7. Mind the Gap [20 points]

Assume you are given a set of n points on a line. The points are given to you as an array of real numbers $x[0\,..\,n-1]$ where $x[0]$ is the leftmost point on the line, $x[1]$ is the next point, $x[2]$ is the next point, etc. The goal, in the first part of the problem, is to find the pair of consecutive points with the largest gap, i.e., to find an i such that $|x[i+1]-x[i]|$ is largest. Then, the second part asks about finding a gap that is at least as large as the *average* gap. Here is an example input:



Problem 7.a. Give an efficient algorithm to find the largest gap. Your algorithm should return a value i that maximizes $|x[i+1]-x[i]|$.

In less than four words, what is your basic approach: Linear search

What is the running time of your approach? $O(n)$

Describe your algorithm briefly, in 2-3 sentences. (Use pseudocode here only if necessary.)

Solution: In this case, you cannot do better than linear search. The solution, then, is to iterate through all the values of i from 0 to $n-2$ and return the value with the maximum gap.

Problem 7.b. Give an efficient algorithm to find a gap that is at least as big as the *average* gap. The average gap is of size $A = \frac{1}{n-1}\sum_{i=0}^{n-2}|x[i+1]-x[i]|$. Your algorithm should return a value i such that $|x[i+1]-x[i]| \geq A$.

In less than four words, what is your basic approach: Binary search

What is the running time of your approach? $O(\log n)$

Describe your algorithm briefly, in 2-3 sentences, and then give pseudocode (or Java).

Solution: In this case, we can use binary search. Initially, we are searching the range of points from 0 to $n-1$, and the average gap $A = (X[n-1]-X[0])/n$. We examine the middle point $m = \lfloor n/2 \rfloor$, and calculate the average of the left half $A_L = (X[m]-X[0])/(m+1)$ and the average of the right half $A_R = (X[n-1]-X[m])/(n-m)$. If A_L is larger, then we recurse to the left, otherwise we recurse to the right. We terminate when the range of points contains only two points.

Pseudocode follows:

```
AveragePointSearch(float[] X, int begin, int end)
    // Base case: only two points left
    if (end <= begin+1) return begin;

    // Calculate midpoint and averages
    A = (X[end] - X[begin]) / (end-begin+1);

    // Note: (end - begin) >= 2, so mid > begin and mid < end.
    // Beware the case where there are only 3 points.
    mid = begin + floor((end - begin)/2);
    Aleft = (X[mid] - X[begin]) / (mid-begin+1);
    Aright = (X[end] - X[mid]) / (end-mid+1);

    // Recurse on the left or right half
    if (Aleft > Aright) return AveragePointSearch(X, begin, mid);
    else return AveragePointSearch(X, mid, end);
```

Solution: Java code:

```
/*
 * GapTester
 * Returns a gap that is at least as large as the average gap.
 */
public class GapTester {
  /*
   * avgGap
   * Calculates the average gap
   */
  public static double avgGap(double[] x, int a, int b){
    double width = x[b] - x[a];
    double number = b-a;
    return (width/number);
  }
  /*
   * gapTest
   * Implements a binary search for the largest average gap in x[begin..end]
   */
  public static int gapTest(double[] x, int begin, int end){
    if (end<=begin+1){
      return begin;
    }
    int mid = begin + (end-begin)/2;
    double A =avgGap(x, begin, end);
    double Alow = avgGap(x, begin, mid);
    double Ahigh = avgGap(x, mid, end);

    if (Alow > A){
      return gapTest(x, begin, mid);
    }
    else {
      return gapTest(x, mid, end);
    }
  }
  /*
   * gapTest
   * Implements a binary search for the largest average gap in x
   */
  public static int gapTest(double[] x){
    return gapTest(x, 0, x.length-1);
  }
  /*
   * Example main
   */
  public static void main(String[] args){
    double[] x = {-9.8, -7.3, -5.9, -3.6, -2.1, 0, 3.8, 5.2, 6.5, 7.9};
    System.out.println(gapTest(x));
  }
}
```

Problem 4. Cool cats. [20 points]

Cats are cool. Cats can also be fat. Your goal is to build a dynamic data structure that maintains a database of cats and supports the following type of query:

> Find me the coolest cat that weighs at most 4.3kg.

In more detail, a cat is defined by three things: its name (a string), its coolness (a non-negative real number), and its weight (a non-negative real number). There is no known upper bound on the coolness or weight of a cat. The data structure should support two operations:

- **insertCat(String name, double cool, double weight)**: Add a cat with the specified parameters to the database.
- **findCoolestCat(double w)**: Let C be the set of cats in the database with weights $\leq w$. You should return the coolest cat in the set C.

Your goal is to develop a data structure to efficiently implement these two operations. Your algorithm should be both time and space efficient.

Example.

```
insertCat("Poof", 17.2, 4.1);
insertCat("Spots", 42.0, 8);
findCoolestCat(5) → "Poof";
findCoolestCat(9.4) → "Spots";
insertCat("Dot", 9.0, 12);
insertCat("Puddles", 0.8, 2.2);
findCoolestCat(4) → "Puddles";
findCoolestCat(4.1) → "Poof";
```

Continued on next page.

Problem 4.a. Suppose that there are currently n cats in the database. How efficiently can we support these operations?

insertCat(...): **Solution:** $O(\log n)$

findCoolestCat(...): **Solution:** $O(\log n)$

Problem 4.b. Describe (briefly) the main idea behind your data structure for implementing these operations:

Solution: This problem can be solved with an augmented AVL tree which is sorted by coolness. Each node also contains the name and weight of the cat. Every node in the AVL tree is augmented to contain the minWeight of any cat in its subtree. Inserting a cat proceeds as a typical insert into an AVL tree, with the following modifications: on inserting a new cat, all the ancestors of the inserted node have their minWeight updated; and during any rotations, the minWeight is properly updated (which should be explained in more detail in a complete solution).

We can implement the findCoolestCat method as follows: We search the tree starting at the root. At every step, we proceed to either the left or the right child, until we find the coolest cat. Assume that the limiting weight is w. Assume we are at node x. We check the minWeight of the right child. If $x.right.minWeight \leq w$, then we go right. Otherwise, if the weight of x is $\leq w$, then we return $x.name$. Otherwise, we move to the left child.

To see that this works, assume that there is some cat in the tree that satisfies the weight bound, and consider the following invariant: if we are at node x, then the tree rooted at x always contains some cat that satisfies the weight bound. This invariant can be proved inductively, noting that at every step we recurse into a subtree that contains a satisfactory cat.

In the same way, we can show that we always find the coolest satisfactory cat, since if there is any satisfactory cat in the right tree, we go right; and if there is a satisfactory cat at the current node, we take it. We therefore always go to a subtree containing the coolest satisfactory cat.
The algorithm runs in $O(\log n)$ steps since each query or insertion involves a single AVL tree operation.