Q1 Excel with Integrity Pledge

0 Points

I will complete this exam in a fair, honest, respectful, responsible and trustworthy manner. This means that I will complete the exam as if the professor was

watching my every action. I will act according to the professor's instructions, and I will neither give nor receive any aid or assistance other than what is authorized. I know that the integrity of this exam and this class is up to me, and I pledge to not take any action that would break the trust of my classmates or professor, or undermine the fairness of this class.

I promise to complete this exam in keeping with the Excel with Integrity Pledge.

Fill in your Name and today's Date below:

Your name, today's date

Q2 Instructions

0 Points

You have 90 minutes to complete the exam. Work to maximize points. If you get stuck, work through other problems and come back to it.

In general, if you think you've spotted a typo in the exam, do your best to answer in the spirit of the question. Keep in mind that some questions have interesting code examples with intentional bugs for you to find as part of the question. **Questions asked during the exam will not be answered, so do your best to interpret the questions.**

In general, assume that any necessary libraries (JUnit, ArrayList, List, and so on) have been imported.

You can use your notes and a compiler. However, the test is designed as if you were taking it during lecture without a compiler and it's highly suggested you not rely on your compiler and use pen & paper to figure out your answers.

Stay calm – you can do this!

Q3 Partition

6 Points

Consider this specification for partition:

```
int partition(int[] numbers)
```

Partition method description:
Chooses a pivot value from the array. Then, changes the array so that all elements smaller than or equal to the pivot appear in indices less than the index of the pivot value, and all elements larger than the pivot appear in indices greater than the index of the pivot value. Returns the final index of the pivot value, which may be different than its starting index. All elements in the input should appear at some index in the output.

Q3.1 A
2 Points

Consider this *input* array:

```
{ 61, 33, 11, 87, 48, 72 }
```

Consider the following array *after* a call to **partition()**, for the input array above. What are **all** the indices that could have been **returned** from **partition()** for this to be a valid partition result? Select **all** possible indices.

```
{ 33, 11, 48, 87, 61, 72 }
```

- [ ] 0
- [ ] 1
- [x] 2
- [ ] 3
- [ ] 4
- [ ] 5

Q3.2 B

2 Points

Consider this *input* array:

```
{ 61, 33, 11, 87, 48, 72 }
```

Consider the following array *after* a call to **partition()**, for the input array above. What are **all** the indices that could have been **returned** from **partition()** for this to be a valid partition result? Select **all** possible indices.

```
{ 11, 48, 33, 72, 61, 87 }
```

- [x] 0
- [ ] 1
- [ ] 2
- [ ] 3
- [ ] 4
- [x] 5

Q3.3 C

2 Points

Consider this *input* array:

```
{ 61, 33, 11, 87, 48, 72 }
```

Consider the following array *after* a call to **partition()**, for the input array above. What are **all** the indices that could have been **returned** from **partition()** for this to be a valid partition result? Select **all** possible indices.

```
{ 33, 11, 48, 61, 87, 72 }
```

- [ ] 0
- [ ] 1
- [x] 2
- [x] 3
- [ ] 4
- [ ] 5

Q4 Run-time

7 Points

Recall that we say *f is O(g)* if there exist *n0* and *C* such that for all *n > n0*, *f(n) < C * g(n)*

Recall that Θ (big-Theta) represents a *tight* bound, O (big-O) an *upper* bound, and Ω (big-Omega) a *lower* bound. (We intentionally do not give definitions for Θ and Ω).

Q4.1 1

1 Point

Give a Θ bound for the number of steps the following program takes in terms of n, assuming **doSomeWork()** does a constant amount of work.

```
for(int i = 0; i < n / 2; i += 1) {
  for(int j = 1; j < n; j = j * 2) {
    doSomeWork();
  }
}
```

- ○ Θ(1)
- ○ Θ(log(n))
- ○ Θ(n)
- ◉ Θ(n*log(n))
- ○ Θ(n^2)
- ○ Θ(n^3)
- ○ Θ(2^n)
- ○ Θ(n!)

Q4.2 2

1 Point

Give a Θ bound for the number of steps the following program takes in terms of n, assuming **doSomeWork()** does a constant amount of work.

```
for(int i = 0; i <= n * n; i = i + 2) {
  for(int j = 0; j < n; j += 1) {
    doSomeWork();
  }
}
```

- ○ Θ(1)
- ○ Θ(log(n))
- ○ Θ(n)
- ○ Θ(n*log(n))
- ○ Θ(n^2)
- ◉ Θ(n^3)
- ○ Θ(2^n)
- ○ Θ(n!)

Q4.3 3a

1 Point

Consider the following pair of functions. Indicate whether *f is O(g)* and/or *f is Ω(g)*. Select zero, one, or both as appropriate.

$f(x) = x * log(x) + 4 * x$ $g(x) = 10 * x * log(x)$

- [x] *f is O(g)*
- [x] *f is Ω(g)*

Q4.4 3b

1 Point

Consider the following pair of functions. Indicate whether *f is O(g)* and/or *f is Ω(g)*. Select zero, one, or both as appropriate.

$f(x) = x^3 + x^2 + x$ $g(x) = x!$

- [x] *f is O(g)*
- [ ] *f is Ω(g)*

Q4.5 3c

1 Point

Consider the following pair of functions. Indicate whether *f is O(g)* and/or *f is Ω(g)*. Select zero, one, or both as appropriate.

$f(x) = x * log(x)$ $g(x) = 3 * log(x) * log(x)$

- [ ] *f is O(g)*
- [x] *f is Ω(g)*

Q4.6 3d

1 Point

Consider the following pair of functions. Indicate whether *f is O(g)* and/or *f is Ω(g)*. Select zero, one, or both as appropriate.

$$f(x) = \frac{x * (x-1)}{2} \qquad g(x) = 5 * x * log(x)$$

- [ ] *f is O(g)*
- [x] *f is Ω(g)*

Q4.7 3e

1 Point

Consider the following pair of functions. Indicate whether *f is O(g)* and/or *f is Ω(g)*. Select zero, one, or both as appropriate.

$$f(x) = \frac{x * (x-1)}{x} \qquad g(x) = 3 * x * log(x)$$

- [x] *f is O(g)*
- [ ] *f is Ω(g)*

Q5 HashTable

9 Points

Consider this implementation of a hash table based on one from the lecture notes, with some portions especially relevant for the question bolded.

interface Hasher { int hash(K key); }

```
class HashTable<K,V> {
  class Entry {
```

```
  K k; V v;
  public Entry(K k, V v) { this.k = k; this.v = v; }
}
List[] buckets; // An array of Lists of Entries
int size;
Hasher hasher;

public HashTable(Hasher h, int startCapacity) {
  this.size = 0;
  this.hasher = h;
  this.buckets = (List[])(new List[startCapacity]);
}

public double loadFactor() { return (double)(this.size) / this.buckets.length; }

public V get(K k) {
  int hashCode = this.hasher.hash(k);
  int index = hashCode % this.buckets.length;
  if(this.buckets[index] == null) {
    return null;
  }
  else {
    for(Entry e: this.buckets[index]) {
      if(e.k.equals(k)) { return e.v; }
    }
    return null;
  }
}

public void set(K k, V v) {
  int hashCode = this.hasher.hash(k);
  int index = hashCode % this.buckets.length;
  if(this.buckets[index] == null) {
    this.buckets[index] = new ArrayList();
    this.buckets[index].add(new Entry(k, v));
  }
  else {
    for(Entry e: this.buckets[index]) {
      if(e.k.equals(k)) { e.v = v; return; }
    }
```

```
      this.buckets[index].add(new Entry(k, v));
    }
    this.size += 1;
  }

}
```

Consider these implementations of the **Hasher** interface:

```
class HasherA implements Hasher<String> {
  int i;
  public HasherA() { this.i = 0; }
  public int hash(String s) { i += 1; return s.length() + i; }
}

class HasherB implements Hasher<String> {
  public int hash(String s) { return Character.codePointAt(s, 0); }
}
```

Recall that Character.codePointAt retrieves the ASCII code of the character at the given index. Consider this sequence of operations:

```
1   HashTable<String, Integer> ht = new HashTable<>(???, 5);
2   ht.set("blue", 500);
3   System.out.println(ht.size);
4   ht.set("green", 500);
5   System.out.println(ht.size);
6   System.out.println(ht.get("blue"));
7   ht.set("green", 200);
8   System.out.println(ht.size);
9   System.out.println(ht.get("green"));
10  ht.set("black", 300);
```

Consider using each of the two Hasher implementations to replace the **???** above by filling it in with either **new HasherA()** or **new HasherB()**.

ASCII Table:

a	97	f	102	k	107	p	112	u	117	z	122
b	98	g	103	l	108	q	113	v	118		
c	99	h	104	m	109	r	114	w	119		
d	100	i	105	n	110	s	115	x	120		
e	101	j	106	o	111	t	116	y	121		

Q5.1 HasherA - Output

3 Points

For HasherA, write the **output** below (5 lines of output):

Q5.2 HasherB - Output

3 Points

For HasherB, write the **output** below (5 lines of output):

Q5.3 HasherA - Length

1 Point

For HasherA, at the end of running all the statements, what would be the length of the longest List in the entries array? Write your answer as a number below:

1

Q5.4 HasherB - Length

1 Point

For HasherB, at the end of running all the statements, what would be the length of the longest List in the entries array? Write your answer as a number below:

3

Q5.5 Incorrect

1 Point

Which of the two produces *incorrect* results? Identify which one does, and **give a one-sentence answer why**:

Q6 Sorting

4 Points

Consider this helper method:

```
public static void dosth(int loc, int[] arr) {
     int j = loc;
     while (j > 0 && arr[j] < arr[j-1]){
       int temp = arr[j];
       arr[j] = arr[j-1];
       arr[j-1] = temp;
       j--;
     }
  }
```

Q6.1 1

1 Point

Consider this method skeleton:

```
void isort(int[] arr) {
  for(int i = 0; i < arr.length; i += 1) {

  }
}
```

Fill in a **single call to the helper method dosth()** that would complete the method and cause it to always change the input array to be in **increasing sorted order**. Write the body of the loop (a single call to dosth):

Q6.2 2

1 Point

Give a Θ bound (a tight bound) for the number of steps the **dosth()** method takes in terms of n in the **worst case** arrangement of the elements in arr, and assuming arr.length is at least as large as n.

- ○ Θ(1)
- ○ Θ(log(n))
- ◉ Θ(n)
- ○ Θ(n*log(n))
- ○ Θ(n^2)
- ○ Θ(n^3)
- ○ Θ(2^n)
- ○ Θ(n!)

Q6.3 3

1 Point

Give a Θ bound (a tight bound) for the number of steps the **dosth()** method takes in terms of n in the **best case** arrangement of the elements in arr, and assuming arr.length is at least as large as n.

- [x] Θ(1)
- [ ] Θ(log(n))
- [ ] Θ(n)
- [ ] Θ(n*log(n))
- [ ] Θ(n^2)
- [ ] Θ(n^3)
- [ ] Θ(2^n)
- [ ] Θ(n!)

Q6.4 4

1 Point

Give a Θ bound (a tight bound) for the number of steps the completed **isort()** method you wrote takes in terms of arr.length in the **worst case** arrangement of the elements in arr, for input arrays of arbitrary size.

- [ ] Θ(1)
- [ ] Θ(log(n))
- [ ] Θ(n)
- [ ] Θ(n*log(n))
- [x] Θ(n^2)
- [ ] Θ(n^3)
- [ ] Θ(2^n)
- [ ] Θ(n!)