Problem Set 6, Part I

Problem 1: Counting unique values

1-1) The worst case occurs when all the elements in arr are unique to each other, which means that each element has to compare with every following element until the end of arr to ensure no duplicates.

1-2) In the worst case, assume arr has a length of n. The first element has to compare with every element from the second to the last, so the number of comparisons will be (n-1). The second element has to compare with every element from the third to the last, so the number of comparisons will be (n-2). The same logic applies to the leftover elements so that we can derive a formula for the total number of comparisons: ((n-1)+1)(n-1)/2 = **(n^2-n)/2**

1-3) We focus on the fastest-growing term in (n^2-n)/2, which is (n^2)/2. We also ignore the coefficient, so the big-O notation of the worst case is **O(n^2)**.

1-4) The best case occurs when there is only one unique element in arr, and the rest are duplicates of it.

1-5) In the best case, assume arr has a length of n. The first element only needs to compare with the second element (since the second element is the first duplicate) to break out of the inner loop so that the comparison will be executed only once. For the second element, similarly, it only needs to compare with the third element (since the third element is the second duplicate). The same logic applies to the leftover array, so we can deduce that the total number of comparisons is consistent with the array's length (i.e., Each element in the loop only needs to perform one comparison). Therefore, the big-O notation for the best case is **O(n)**.

Problem 2: Improving the efficiency of an algorithm

2-1)

```
public static int numUnique(int[] arr) {
      Sort.mergeSort(arr);
      int count = 0;

      for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] != arr[i+1]) {
                  count++;
            }
      }
      return count + 1;

}
```

2-2) The worst-case time efficiency for the new algorithm is the big-O notation of mergeSort's worst-case and of the loop. The worst-case time efficiency of mergeSort is O(nlogn). Since the comparison arr[i] != arr[i+1] is executed once for each pass of the loop, the worst case will have a time efficiency of O(n). If we only focus on the fastest-growing term, O(nlogn) + O(n) = O(nlogn). Therefore, the worst-case time efficiency for the new algorithm is O(nlogn).

2-3) The best-case time efficiency for the new algorithm is O(nlogn), since the mergeSort becomes the most inefficient portion of the algorithm, and it has the same big-O notation for all possible cases. Compared with the best-case time efficiency of the old algorithm, O(nlogn) is nlogn time, whereas O(n) is linear time and more efficient. Therefore, although the new algorithm has a better worst-case performance, the old algorithm is more efficient with the best possible case.

Problem 3: Practice with references

3-1)

Expression	**Address**	**Value**
n	0x128	0x800
n.ch	0x800	‘e’
n.next	0x802	0x240
n.prev.next	0x182	0x800
n.next.prev	0x246	0x800
n.next.prev.prev	0x806	0x180

3-2)

```
n.next.prev = m;
m.next = n.next;
n.next = m;
m.prev = n;
```

3-3)

```
public static void addNexts(DNode last) {
      last.next = null;
      DNode trav = last;
      while (trav.prev != null) {
            trav.prev.next = trav;
            trav = trav.prev;
      }
}
```

Problem 4: Printing the odd values in a list of integers

4-1)

```
public static void printOddsRecur(IntNode first) {
	/*
	 * printOddsRecur() uses recursion to print the odd values
	 * in the list (if any), with each value printed on a separate line.
	 * If there are no odd values,
	 * the method should not do any printing.
	 */
	if (first == null) {
		return;
	} else if (first.next == null) {
		if (first.val % 2 == 1) {
			System.out.println(first.val);
		}
	} else if (first.val % 2 != 1) {
		System.out.println(first.val);		// note the position of this line
		printOddRest(first.next);
	} else {
		printOddRest(first.next);
	}


}
```

4-2)

```
public static void printOddsIter(IntNode first) {
	if (first == null) {
		return;
	}
	IntNode trav = first;
	while (trav != null) {
		if (trav.val % 2 == 1) {
			System.out.println(trav.val);
		}
		trav = trav.next;
	}

}
```