Computer Science 228
Project 2
Computing Distances Between Rankings

Important Notes

- This project consists of two parts:
  Part 1 (50 points): Due at 11:59 pm, Monday, February 18, 2013.
  Part 2 (250 points): Due at 11:59 pm, Friday, March 1, 2013.
- Please start the assignment as soon as possible and get your questions answered early.
- Read through this specification completely before you start.
- Some aspects of this specification are subject to change, in response to issues detected by students or the course staff. ***Check Blackboard regularly for updates and clarifications.***
- ***This assignment is to be done on your own.*** If you need help, see one of the instructors or the TAs. Please make sure you understand the "Academic Dishonesty Policy" section of the syllabus.

First posted: February 10

1 Introduction

All of us have seen rankings of some kind, be it of colleges, smart phones, or Superbowl commercials. Search engines routinely produce, on demand, ranked recommendations of hotels, restaurants, or electrical appliances according to a variety of criteria. Of course, rankings produced through different means or at different times can disagree. Table 1 illustrates this, showing two rankings of the world's most expensive cities[1]. This leads to a natural question: *How can we measure the degree of similarity (or difference) between two rankings for the same set of items?* It turns out that mathematicians and statisticians have studied this problem longer than the Internet has been around. The measures of distance between rankings that they developed are now used

[1]Adapted from:
`http://www.nbcnews.com/business/worlds-10-most-expensive-cities-live-eat-big-mac-buy-999380.`

City	**2011** σ_1	**2012** σ_2
Copenhagen	4	5
Geneva	3	4
Oslo	1	1
Tokyo	5	3
Zurich	2	2

Table 1: Five cities ranked based on cost of living for two consecutive years.

widely by search engines[2]. In this project, you will implement algorithms for computing two such measures. To define these measures precisely, we first need some terminology and notation.

Suppose we are given a set U of n items to be ranked —in Table 1, $U = \{$Copenhagen, Geneva, Oslo, Tokyo, Zurich$\}$ and $n = 5$. A ***ranking*** for U is a function σ that assigns to each item i in U a distinct integer $\sigma(i)$ between 1 and n. Function σ determines an ordering

$$\sigma(a_1) < \sigma(a_2) < \cdots < \sigma(a_n)$$

among the items in U, where lower σ-values imply higher ranking. Different rankings imply different orderings.

Example 1. The 2011 and 2012 rankings from Table 1 are given by functions σ_1 and σ_2, respectively, where

$$\sigma_1(\text{Oslo}) = 1 < \sigma_1(\text{Zurich}) = 2 < \sigma_1(\text{Geneva}) = 3 < \sigma_1(\text{Copenhagen}) = 4 < \sigma_1(\text{Tokyo}) = 5$$

and

$$\sigma_2(\text{Oslo}) = 1 < \sigma_2(\text{Zurich}) = 2 < \sigma_2(\text{Tokyo}) = 3 < \sigma_2(\text{Geneva}) = 4 < \sigma_2(\text{Copenhagen}) = 5.$$

We now define our distance measures.

Definition 1. The ***Spearman footrule distance*** (or the ***footrule distance*** for short) between two rankings σ_1 and σ_2 of a set of items U is given by

$$F(\sigma_1, \sigma_2) = \sum_{a \in U} |\sigma_1(a) - \sigma_2(a)|,$$

where "$|\cdot|$" denotes absolute value.

Thus, the footrule distance between σ_1 and σ_2 is the total displacement of the items between σ_1 and σ_2.

[2]C. Dwork, R. Kumar, M. Naor, D. Sivakumar. Rank Aggregation Methods for the Web. *WWW10*, May 2-5, 2001, Hong Kong. `http://www10.org/cdrom/papers/577/`

Example 2. For the rankings of Table 1, using the notation of Example 1, we have

$$\begin{aligned}F(\sigma_1,\sigma_2) =&|\sigma_1(\text{Copenhagen}) - \sigma_2(\text{Copenhagen})| + |\sigma_1(\text{Geneva}) - \sigma_2(\text{Geneva})| + \\ &|\sigma_1(\text{Oslo}) - \sigma_2(\text{Oslo})| + |\sigma_1(\text{Tokyo}) - \sigma_2(\text{Tokyo})| + |\sigma_1(\text{Zurich}) - \sigma_2(\text{Zurich})| \\ =&|4-5| + |3-4| + |1-1| + |5-3| + |2-2| \\ =&1+1+0+2+0 \\ =&4\end{aligned}$$

Definition 2. The ***Kemeny distance*** between two rankings σ_1 and σ_2 of a set of items U is the total number of distinct pairs (i, j) of items from U such that

$$\sigma_1(i) < \sigma_1(j) \quad \text{and} \quad \sigma_2(i) > \sigma_2(j).$$

Thus, the Kemeny distance between σ_1 and σ_2 is the number of distinct pairs of items whose *relative* rankings in σ_1 and σ_2 differ.

Example 3. Continuing with Example 2, note that the only pairs of cities that have different relative positions in σ_1 and σ_2 are (Copenhagen, Tokyo) and (Geneva, Tokyo), since

$$\sigma_1(\text{Copenhagen}) = 4 < 5 = \sigma_1(\text{Tokyo}) \quad \text{and} \quad \sigma_2(\text{Copenhagen}) = 5 > 3 = \sigma_2(\text{Tokyo})$$

and

$$\sigma_1(\text{Geneva}) = 3 < 5 = \sigma_1(\text{Tokyo}) \quad \text{and} \quad \sigma_2(\text{Geneva}) = 4 > 3 = \sigma_2(\text{Tokyo}).$$

Thus, $K(\sigma_1, \sigma_2) = 2$.

The Kemeny distance and the footrule distance between any two rankings σ_1 and σ_2 are related by the following expression[3].

$$K(\sigma_1,\sigma_2) \le L(\sigma_1,\sigma_2) \le 2K(\sigma_1,\sigma_2).$$

It is easy to compute the footrule distance between two rankings of n items in $O(n)$ time —we leave it to you to figure out how. One of your tasks in this assignment is to implement such an algorithm. The naïve strategy to calculate the Kemeny distance is to enumerate all $n(n-1)/2$ distinct pairs of items, checking each to see if it is an inversion. Its time complexity is $O(n^2)$. In the next sections, we outline a $O(n \log n)$ algorithm, which is much faster than the naïve approach. Another one of your tasks in this project is to implement this $O(n \log n)$ algorithm.

2 Computing Kemeny Distance

Suppose we rearrange the items in U, ordering them according to one of the two given rankings, say σ_1. Table 2 shows the result of doing this for the rankings of Table 1. This has no effect on Kemeny distance (or the footrule distance) between the rankings, but it offers a benefit: it reduces computing Kemeny distance to a well-solved problem, defined next.

[3]P. Diaconis and R. L. Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society, Series B.* 39(2): 262–268 (1977).

City	**2011** σ_1	**2012** σ_2
Oslo	1	1
Zurich	2	2
Geneva	3	4
Copenhagen	4	5
Tokyo	5	3

Table 2: The cities from Table 1, rearranged according the 2011 ranking.

Definition 3. An ***inversion*** in a sequence $\tau = (\tau(1), \tau(2), \ldots, \tau(n))$ of distinct numbers is a pair of elements such that $\tau(i) > \tau(j)$, but $i < j$. In the ***inversion counting problem***, we are given a sequence τ as above, and we are asked to count the number of inversions in τ

Thus, an inversion in τ is pair of numbers that are out of order with respect to each other, and the number of inversions in τ measures how far it is from being sorted.

If you think about it for few moments, you will realize that the Kemeny distance between any two rankings σ_1 and σ_2 is exactly the same as the number of inversions in the sequence τ obtained by rearranging σ_2 according to the ordering of the elements implied by the *other* ranking, σ_1. That is, computing Kemeny distance reduces to counting inversions.

Example 4. Here and in what remains of this project description, it will be helpful to view the set of items U as a list of consecutive numbers, corresponding to the elements ordered by the first ranking. Thus, for the data in Table 2, we have Oslo $\equiv 1$, Zurich $\equiv 2$, Geneva $\equiv 3$, Copenhagen $\equiv 4$, Tokyo $\equiv 5$. Then,

$$\begin{aligned}\tau &= (\tau(1), \tau(2), \tau(3), \tau(4), \tau(5)) \\ &= (\sigma_2(1), \sigma_2(2), \sigma_2(3), \sigma_2(4), \sigma_2(5)) \\ &= (1, 2, 4, 5, 3).\end{aligned}$$

Sequence τ has two inversions, $4 \leftrightarrow 3$ and $5 \leftrightarrow 3$, which is precisely the Kemeny distance between σ_1 and σ_2.

In the next section, we describe a $O(n \log n)$ algorithm for counting inversions. One of your tasks will be to implement this algorithm as part of a `Ranking` class. You will also have to adapt this algorithm to compute the Kemeny distance between two rankings in $O(n \log n)$ time.

3 Counting Inversions

Figure 1 shows SORTANDCOUNT, an adaptation of mergesort that sorts and counts inversions in a sequence in $O(n \log n)$ time[4]. Like mergesort, SORTANDCOUNT is based on the principle of ***divide and conquer***, that is:

[4]The pseudocode in this section roughly follows that in Chapter 5, Section 3, of: J. Kleinberg and É. Tardos. *Algorithm Design*, Addison-Wesley, 2006.

```
SORTANDCOUNT
Input: A list $\tau = (\tau(1), \tau(2), \ldots, \tau(n))$ of distinct numbers.
Output: The number of inversions in $\tau$, along with a sorted version of $\tau$.
Step 1. If $n = 1$, return $0$ and $\tau$.
Step 2. Construct two lists
            $\tau_{\text{left}} = (\tau(1), \ldots, \tau(m))$ and $\tau_{\text{right}} = (\tau(m+1), \ldots, \tau(n))$,
        where $m$ is the integer part of $n/2$.
Step 3. Call SORTANDCOUNT recursively to obtain $\mathtt{inv}_{\text{left}}$, the number of inversions in $\tau_{\text{left}}$,
        and $\tau'_{\text{left}}$, the sorted version of $\tau_{\text{left}}$.
Step 4. Call SORTANDCOUNT recursively to obtain $\mathtt{inv}_{\text{right}}$, the number of inversions in $\tau_{\text{right}}$,
        and $\tau'_{\text{right}}$, the sorted version of $\tau_{\text{right}}$.
Step 5. Call MERGEANDCOUNT to count $\mathtt{inv}_{\text{cross}}$, the number of inversions involving an ele-
        ment in $\tau_{\text{left}}$ and an element in $\tau_{\text{right}}$, along with $\tau'$, the sorted combination of $\tau'_{\text{left}}$ and
        $\tau'_{\text{right}}$.
Step 6. Return $(\mathtt{inv}_{\text{left}} + \mathtt{inv}_{\text{right}} + \mathtt{inv}_{\text{cross}})$ and $\tau'$.
```

Figure 1: A mergesort-based algorithm for counting inversions.

The number of inversions in τ equals the number of inversions in the left half of τ, plus the number of inversions in the right half of τ, plus the number of inversions across the two halves.

SORTANDCOUNT relies on algorithm MERGEANDCOUNT, shown in Figure 2, which combines two sorted lists into another sorted list in time linear in the total number of elements, while also counting the inversions across the two lists. Study these algorithms carefully, paying special attention to MERGEANDCOUNT, and make sure that you understand why they are correct and why their running times are what we claim them to be. You might find it useful to watch the video on counting inversions by Prof. Dan Gusfield, of UC Davis[5]. To further help you along, here is a worked example.

Example 5. Suppose $\tau = (1, 5, 3, 8, 4, 2, 7, 6)$. The inversions in τ are

$$5 \leftrightarrow 3, 5 \leftrightarrow 4, 5 \leftrightarrow 2, 3 \leftrightarrow 2, 8 \leftrightarrow 4, 8 \leftrightarrow 2, 8 \leftrightarrow 7, 8 \leftrightarrow 6, 4 \leftrightarrow 2, \text{ and } 7 \leftrightarrow 6,$$

so the total number of inversions is 10. SORTANDCOUNT counts these inversions as follows:

1. Split τ into two sublists: $\tau_{\text{left}} = (1, 5, 3, 8)$ and $\tau_{\text{right}} = (4, 2, 7, 6)$.
2. Recursively invoke SORTANDCOUNT on τ_{left}, to obtain $\mathtt{inv}_{\text{left}} = 1$ (corresponding to the inversion $5 \leftrightarrow 3$) and the sorted list $\tau'_{\text{left}} = (1, 3, 5, 8)$.

[5]`http://www.youtube.com/watch?v=Ly-ht7PN9UM`.

MERGEANDCOUNT

Input: Two sorted lists $\alpha = (\alpha(1), \alpha(2), \ldots, \alpha(p))$ and $\beta = (\beta(1), \beta(2), \ldots, \beta(q))$, each consisting of distinct integers, with no numbers in common.

Output: The number of pairs (i, j) such that $\alpha(i) > \beta(j)$, together with the sorted combination of α and β.

```
1. Set $i = 1$, $j = 1$, count $= 0$, and $\gamma$ equal to the empty list
2. While $i \leq p$ and $j \leq q$ do the following:
    (a) If $\alpha(i) < \beta(j)$
        i. Append $\alpha(i)$ to $\gamma$.
        ii. $i$++
    (b) If $\beta(j) < \alpha(i)$        // Inversion found: $\alpha(i), \ldots, \alpha(p)$ are all greater than $\beta(j)$.
        i. Append $\beta(j)$ to $\gamma$.
        ii. count $=$ count $+ (p - i + 1)$
        iii. $j$++
3. If $i > p$
    (a) Append $\beta(j), \ldots, \beta(q)$ to $\gamma$.
4. If $j > q$
    (a) Append $\alpha(i), \ldots, \alpha(p)$ to $\gamma$.
5. Return count and $\gamma$.
```

Figure 2: Merging and counting inversions across two sequences.

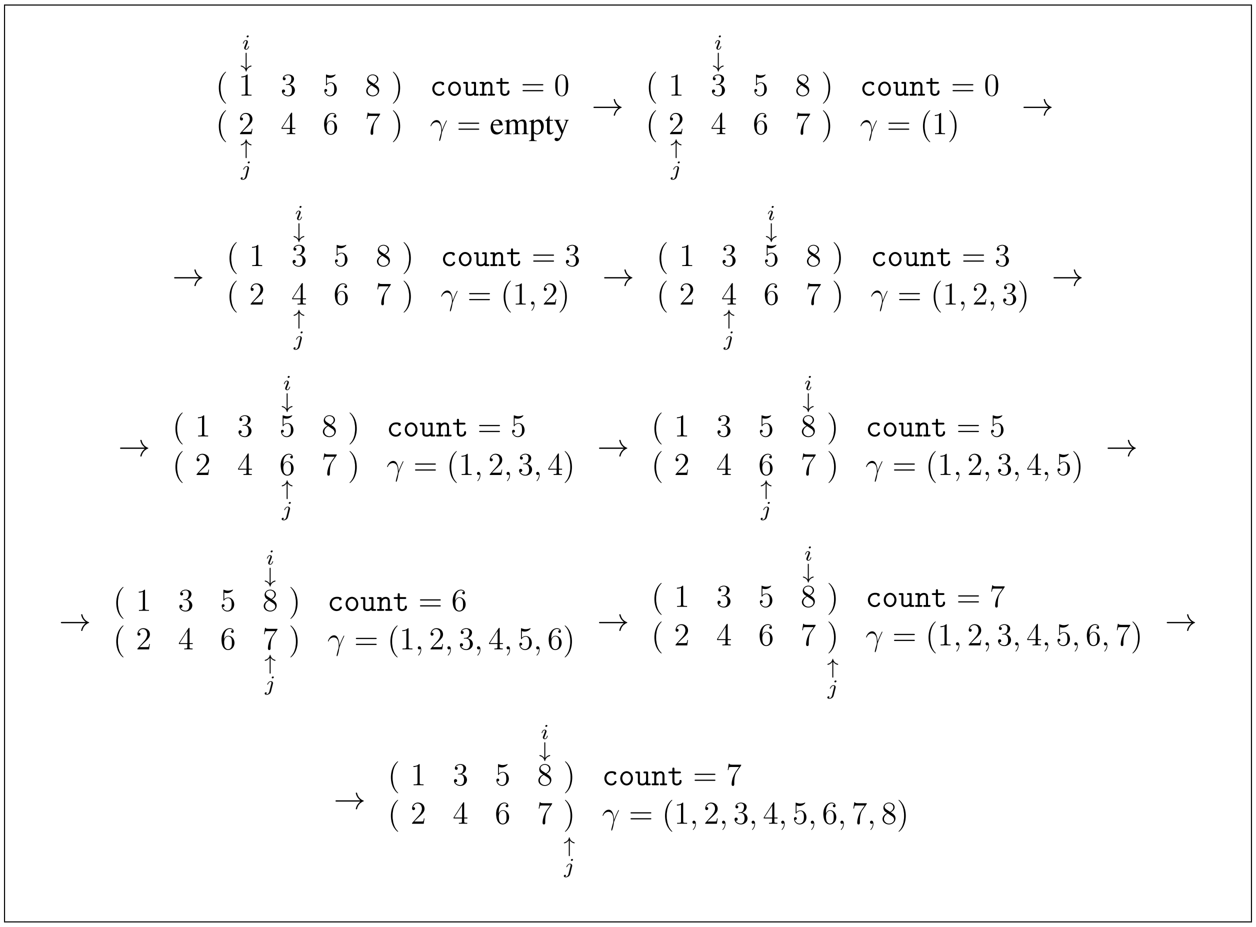
count = 0, γ = empty
count = 0, γ = (1)
count = 3, γ = (1,2)
count = 3, γ = (1,2,3)
count = 5, γ = (1,2,3,4)
count = 5, γ = (1,2,3,4,5)
count = 6, γ = (1,2,3,4,5,6)
count = 7, γ = (1,2,3,4,5,6,7)
count = 7, γ = (1,2,3,4,5,6,7,8)

Figure 3: Execution of MERGEANDCOUNT.

3. Recursively invoke SORTANDCOUNT on τ_{right}, to obtain $\texttt{inv}_{\text{right}} = 2$ (corresponding to the inversions $4 \leftrightarrow 2, 7 \leftrightarrow 6$) and the sorted list $\tau'_{\text{right}} = (2, 4, 6, 7)$.
4. Invoke MERGEANDCOUNT with arguments τ'_{left} and τ'_{right}, to obtain $\texttt{inv}_{\text{cross}} = 7$ (corresponding to the inversions $3 \leftrightarrow 2, 5 \leftrightarrow 2, 5 \leftrightarrow 4, 8 \leftrightarrow 2, 8 \leftrightarrow 4, 8 \leftrightarrow 6, 8 \leftrightarrow 7$) and $\tau' = (1, 2, 3, 4, 5, 6, 7, 8)$, the merge of τ'_{left} and τ'_{right}. Figure 3 illustrates the execution of MERGEANDCOUNT on lists τ'_{left} and τ'_{right}.
5. Return $\texttt{inv}_{\text{left}} + \texttt{inv}_{\text{right}} + \texttt{inv}_{\text{cross}} = 10$, along with τ'.

4 Tasks

Your job is to implement `public class Ranking`. A `Ranking` object represents a ranking of a set of items numbered 1 through n, for some n. The `Ranking` class contains various methods to construct rankings. Once a ranking is constructed, you can determine how many items it ranks and obtain the rank of some specific item. You can also compute distances between a ranking and other rankings. `Ranking` objects are ***immutable***; that is, once created, they cannot be modified.

Required Methods

We now specify the methods that you must implement. Since a considerable part of the testing will be automated, you ***must*** follow these specifications ***exactly***. This means, among other things, that the names and types of classes and methods must be kept ***as is***. You can, of course, define additional helper methods of your own.

`public Ranking(int n)`

Constructs a random ranking of the numbers `1` through `n`. Throws an `IllegalArgumentException` if `n` < 1. Must run in $O(n \log n)$ time.

Note. For random number generation, use the `RandomSingleton` class from Project 1; this generator might be modified slightly for use in this project. To generate a random permutation of `1` through `n`, use the "shuffle" algorithm, described in `http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_modern_algorithm`

`public Ranking(int[] rank)`

Constructs a ranking σ of the set $U = \{1, \ldots, \texttt{rank.length}\}$, where $\sigma(i) = \texttt{rank}[i-1]$. Throws a `NullPointerException` if `rank` is `null`. Throws an `IllegalArgumentException` if `rank` does not consist of distinct elements between 1 and `rank.length`. Must run in $O(n \log n)$ time, where $n =$ `rank.length`.

`public Ranking(float[] scores)`

Constructs a ranking of the set $U = \{1, \ldots, \texttt{scores.length}\}$, where element i gets rank k if and only if `scores`$[i-1]$ is the kth largest element in the array `scores`. Throws a `NullPointerException` if `scores` is `null`. Throws an `IllegalArgumentException` if `scores` contains duplicate values. Must run in $O(n \log n)$ time, where $n =$ `scores.length`.

Example 6. Suppose `scores` $= (0.75, 0.36, 0.65, -1.5, 0.85)$. Then, the corresponding ranking is $\sigma = (2, 4, 3, 5, 1)$.

`public int getNumItems()`

Returns the number of items in the ranking. Must run in $O(1)$ time.

`public int getRank(int i)`

Returns the rank of item `i`. Throws an `IllegalArgumentException` if item `i` is not present in the ranking. Must run in $O(1)$ time.

`public static int footrule(Ranking r1, Ranking r2)`

Returns the footrule distance between `r1` and `r2`. Throws a `NullPointerException` if either `r1` or `r2` is `null`. Throws an `IllegalArgumentException` if `r1` and `r2` have different lengths. Must run in $O(n)$ time, where n is the number of elements in `r1` (or `r2`).

`public static int kemeny(Ranking r1, Ranking r2)`

Returns the Kemeny distance between `r1` and `r2`. Throws a `NullPointerException` if either `r1` or `r2` is `null`. Throws an `IllegalArgumentException` if `r1` and `r2` have different lengths. Must run in $O(n \log n)$ time, where n is the number of elements in `r1` (or `r2`).

`public int fDist(Ranking other)`

Returns the footrule distance between `this` and `other`. Throws a `NullPointerException` if `other` is `null`. Throws an `IllegalArgumentException` if `this` and `other` have different lengths. Must run in $O(n)$ time, where n is the number of elements in `this` (or `other`).

`public int kDist(Ranking other)`

Returns the Kemeny distance between `this` and `other`. Throws a `NullPointerException` if `other` is `null`. Throws an `IllegalArgumentException` if `this` and `other` have different lengths. Must run in $O(n \log n)$ time, where n is the number of elements in `this` (or `other`).

`public int invCount()`

Returns the number of inversions in `this` ranking. Should run in $O(n \log n)$ time, where n is the number of elements in `this`.

Note. Since `Ranking` objects are immutable, you could, in fact, compute the number of inversions in a ranking just once, at the time of creation, and store it for later access. With this implementation, `invCount` would take $O(1)$ time. You are free to implement this version or the one that computes inversions every time the method is called; your documentation should indicate clearly which approach your method uses.

The precise representation of `Ranking` objects is left up to you —you could, for instance, use arrays or `ArrayLists`. Your code ***must*** include implementations of algorithms SORTANDCOUNT and MERGEANDCOUNT and it ***must*** use these implementations to count inversions and compute Kemeny distance.

5 Submission

The submission of this project consist of two parts.

Part 1: Submit a file containing JUnit test classes containing multiple test cases for all the methods in the `Ranking` class. Your code must

(a) test that methods through exceptions under the appropriate circumstances (e.g., null arguments),

(b) verify that all three constructors produce valid rankings (i.e., permutations of 1 through n),

(c) test boundary cases —including proper handling of single-element rankings, inversion counting on ascending and descending sequences, determining distances between identical rankings and rankings that are reversals of each other, etc.—,

(d) perform any additional tests that you deem necessary.

Part 2: Turn in a zip file named `Firstname_Lastname_HW2.zip` containing all your source code for the `Ranking` class. The package name for the project must be `edu.iastate.cs228.hw2`. Include the Javadoc tag `@author` in each class source file. ***Do not turn in class files.***

All submissions must be done through Blackboard Learn. Please follow the guidelines posted under "Documents & Links".

6 Grading

Your grade for Part 1 will mainly be determined by thoroughness with which your JUnit tests check the required methods. We may also run your tests on some correct and incorrect versions of the various classes. The correct versions should pass your tests, and the incorrect versions should fail appropriately.

A significant portion of your grade for Part 2 will be based on running our unit tests on your classes. We will also inspect your code to verify that your methods satisfy the stipulated time complexity requirements. Documentation and style will count for roughly 15% of the total points.

Late Penalties

Asssignments may be submitted up to 24 hours after the the deadline with a 25% penalty (not counting weekends or holidays). No credit will be given for assignments submitted after that time.