introduction Programming assignment 01's purpose is to reinforce algorithm assessment techniques learned in the classroom. The goal is to implement some sorting algorithms and perform runtime analysis, correctness proofs, and experiment with the given algorithms. In order to properly proof the correctness of the algorithms, invariant proofs will be used.

Bubble sort pseudo code and invariant proof Invariant: At the beginning of the Jth iteration, A[A.length-J...A.length - 1] has the largest J elements in sorted order and N = number of swaps is not zero

Initialization: At J=1, A[A.length -J...A.length-1] is a single, trivially sorted element while N is zero.

Maintenance: During the Jth iteration, either:
$\forall$ A[j...A.length - 1] is sorted
or
$\exists$ i: A[0...i] $\leq$ A[J] $\leq$ A[i+2...A.length-1] and N $\neq 0$

Termination: When N is ≥ 0, A[0...A.length - 1] is sorted and the loop ends

```
BubbleSort(A):
// let A be an array [0...n] of n elements
N = A.length
while N != 0:
        newn = 0
        {I}
        for i = 1 to A.length - 1:
                {I ^ i < A.length - 1}
                if A[i - 1] > A[i]:
                        swap(A[i], A[i-1])
                        newn = i
                {I ^ i = A.length}
        N = newn
        {Q}
        return A
```

Quick sort pseudo code and invariant proof Invariant: At the beginning of the Jth iteration, there exists a P, I, K, R, and a pivot X such that A[P...I] $\leq$ X and A[I+1...K-1] $>$ X and A[K...R] is unrestricted.
Initialization: At J = 1, I = P - 1 and K = P such that A[P...I] is empty and A[I+1...K-1] is empty, and A[K...R] is the entire array, therefor the invariant is

true.
Maintenance: During the Jth iteration: either
$\exists$ X > A[K], in this case no swapping is required and the iteration ends
$\exists$ K such that A[K] $\leq$ X, in this case swapping occurs and A[I] $\leq$ X while A[K - 1] > X
Termination: When J=R, $\forall$ A[0...A.length - 1] is in one of the sorting partitions, and the array is sorted

```
QuickSort(A, p, r):
        // let A be an array [0...n] of n elements
        // let p be an integer greater than or equal to 0
        // let r be an integer less than n
        if p < r:
                q = self.partition(A, p, r)
                self.quicksort(A, p, q - 1)
                self.quicksort(A, q+1, r)
        return A

partition(A, p, r):
        x = A[r]
        i = p - 1
        {I}
        for j = p, j < r, j++:
                if A[j] <= x:
                { I ^ x > A[j]}
                        i = i + 1
                        swap(A[j], A[i])
        swap(A[i+1], A[r])
        { I ^ A[j] > x}
        return i + 1
        {Q}
```

Radix sort pseudo code and invariant proof Invariant: At the beginning of the Jth iteration, there exists a K such that K = J - 1 and K ¡ length(max(A)), and the last K elements in the array have been sorted.
Initialization: At J = 1, K = 0 and the last K digits of the array A have been trivially sorted.
Maintenance: During the Jth iteration: $\forall$ A[0...N] digits [K - J...K - 1] have been sorted. And either:
K == length(max(A)), therefore no further sorts are required
K ¡ length(max(A)), therefor a sort occurs on the K - 1 digit
Termination: When K == length(max(A)) the array A has been sorted

```
RadixSort(A):
        // let A be an array [0...n] of n elements
        N = 10
```

```
    max_length = number of digits in the largest value of A
    {I}
    for i= 0, i < max_length, i++:
            {I ^ i < max_length}
            bins = list of lists with N columns
            for item in A:
                    append item to the bin number with the value of the ith
            A = []
            for bin in bins:
                    append each item in bin into A
    {I ^ i == max_length}
    return A
    {Q}
```

Bucket sort pseudo code and invariant proof Invariant: At the beginning of the Jth iteration, there exists an N such that N = number of buckets, M such that M = A.length / N and A[0...J-1] have been sorted into one N buckets.
Initialization: At J = 1, A[0...0] items are in the buckets and the buckets are trivially sorted.
Maintenance: During the Jth iteration: A[J - 1] items have been sorted to buckets and either:
J == A.length: this case is the end case, ∀ A[0..A.length - 1] have been sorted into N buckets
J ¡ A.length: ∃ K such that K = J and A[k] is not in one of N buckets, A[K] is added to bucket (A[k] - A.min / M) and A[0...J] have been sorted into buckets
Termination: When J = A.length, all items have been sorted into buckets and the buckets are combined into one array

```
BucketSort(A, N):
    // let A be an array [0...n] of n elements
    // let N be the number of buckets
    buckets = list of lists with N columns
    bucket_size = A.length / N
    min_value = A.min
    {I}
    for i = 0, i < A.length, i++:
            {I ^ i < A.length}
            pos = floor((A[i] - min_value) / bucket_size)
            append A[i] to the bucket with positon pos
    {I ^ i == A.length}
    result = []
    for bucket in buckets:
            append the sorted bucket to result
    {Q}
    return result
```

Testing Plan My testing plan is to provide each of the sorting algorithms an unsorted random array of a large size to determine when they start to exhibit asymptotic growth while confirming the sorting algorithms correctness using smaller empty and reverse sorted arrays. Testing of the large data sets will also coincide with changing the bucket size from that equal to a tenth of the smallest data set all the way up to the size of the smallest data set.