2C03 - Assignment 1

Nathan Jervis

1.

`1.3.45`

Determining whether the stack underflows can be performed by removing the data storage and retrieval operations from the `push` and `pop` and simply tracking the `N` value. If at any point while consuming the input stream `N` becomes negative, then the input stream would cause an underflow.

To determine if a given permutation can occur, perform the following:

1. Compare the next number to appear to the number on top of the stack
2. If it's the same number, pop it off and return to step 1
3. Otherwise push the next number on the stack and return to step 1.

When all of the `N` have been consumed, the output is either matched, or the given permutation can not occur.

`1.3.46`

Using the partial solution from the book as a base

> Suppose that there is a forbidden triple (a, b, c). Item c is popped before a and b, but a and b are pushed before c. Thus, when c is pushed, both a and b are on the stack. Therefore, a cannot be popped before b.

(Algorithms 4th edition, pg 171)

If there are no forbidden triples then for each element `x` in the output, when `x` is required for the output, there may be no number greater than `x` on top. If a number greater than `x` is on top (let's call it `y`), then `y` must have pushed after `x`. If `x` was pushed first, then it could have been popped before `y` was pushed unless a number `z`, `z>y` was required first. If that was the case then `c=z, y=b, x=a` and there exists a forbidden triple. Since numbers are pushed in ascending order, there can't exist a number less than `x` that was pushed later, and since numbers are unique `x` must be at the top of the stack and therefore available to be popped.

`1.3.48`

Implementing two stacks with a deque is easy, each stack simply takes either the front or the back of the queue to perform it's operations on. It's important that each stack keeps track of how many are on it's stack so it doesn't underflow and pop items from the bottom of the other side. Java implementation below:

```
public class DoubleStack<T> {
  Deque<T> deque = new ArrayDeque<T>();

  int length1 = 0;
  int length2 = 0;
  void push1(T value){
      length1++;
      deque.addFirst(value);
  }
  void push2(T value){
      length2++;
      deque.addLast(value);
  }
  T pop1(){
      if (length1==0)
          throw new IndexOutOfBoundsException();
      length1--;
      return deque.removeFirst();
  }
```

```
    T pop2(){
        if (length2==0)
            throw new IndexOutOfBoundsException();
        length2--;
        return deque.removeLast();
    }
}
```

2.

1.4.5

a) N
b) 1
c) 1
d) 2N^3
e) 1
f) 2
g) 0

1.4.6

a) N^2
b) N x lg(N)
c) N x lg(N)

1.4.9

1.4.12

```
    public static void PrintSameNums(int[] nums1, int[] nums2){
        int p1 = 0;
        int p2 = 0;
        while(p1<nums1.length&&p2<nums2.length){
            if (nums1[p1]==nums2[p2])
                System.out.println(nums1[p1]);
            if (nums1[p1]<nums2[p2])
                p1++;
            else
                p2++;
        }
    }
```

1.4.15

The basic idea of the algorithm is to start i and k at opposite sides. If the absolute sum of the values at i and j is still less than k, then decrease k, otherwise move j towards k. When it reaches k, increase i and start again. `i` and `k` will never overlap, so together they perform the operation with j once for each element. The operation with j touches each element once at the worst case. So together it's quadratic in the worst case.

```
    public static int ThreeSumFaster(int[] nums){
        int total = 0;
        int i = 0;
        int j = 1;
        int k = nums.length - 1;
        while(i!=j&&nums[i]<=0&&nums[k]>=0){
            if (nums[i]+nums[j]+nums[k]==0){
                total++;
            }
```

```
            if (Math.abs(nums[i]+nums[j])>nums[k]){
                j++;
                if (j==k){
                    i++;
                    j = i+1;
                    if (j==k)
                        break;
                }
            }
            else{
                k--;
            }
        }
        return total;
    }
```

1.4.17

The farthest pair is simply the minimum and the maximum value, so the following works as a simple linear algorithm:

```
    public static void FarthestPair(double[] nums){
        double minimum = Double.MAX_VALUE;
        double maximum = Double.MIN_VALUE;
        for(double value:nums){
            if (value<minimum)
                minimum = value;
            if (value>maximum)
                maximum = value;
        }
        System.out.format("%.3f, %.3f, distance = %.3f",minimum, maximum, maximum - minimum);
    }
```

3.

1.4.20

Simply find the maximum point (`bitronicMax`, O(lg n)) and then do a binary search on either half (`binaryFind`, O(lg n)).

```
    public static int bitronicFind(int[] nums, int target){
        int low = 0;
        int high = nums.length-1;
        int middle = bitronicMax(nums,low,high);
        return binaryFind(nums,low,middle,target) || binaryFind(nums,middle,high,target);
    }
    public static int bitronicMax(int[] nums, int low, int high){
        if (low==high)
            return high;
        int middle = low + (high - low)/2;
        if (nums[middle] < nums[middle+1])
            return max(nums,middle+1,high);
        if (nums[middle]>nums[middle+1])
            return max(nums,low,middle);
        return middle;
    }
    public static boolean binaryFind(int[] nums, int low, int high, int target){
        if (high<=low)
            return nums[low]==target;
        int middle = low + (high - low)/2;
        if (nums[middle]==target)
            return true;
        if (nums[middle]<target)
            return binaryFind(nums,middle+1,high,target);
        return binaryFind(nums,low,middle-1);
```

```
}
```