Homework 1 Solutions

(1) **(15 pts)** Write a recursive method to determine if a character is in a list of characters in O(logN) time. Mathematically prove *(as we did in class)* that T(N) = O(logN). You can assume that this list is sorted lexicographically.

```
def contains(l, x):
    if len(l) == 1:
        return x == l[0]
    else:
        midpoint = len(l)/2
        if l[midpoint] > x:
            return contains(l[midpoint:], x)
        elif l[midpoint] < x:
            return contains(l[:midpoint], x)
        else:
            return True
```

The algorithm comes from a simple binary search we discussed earlier in class. Proof using Masters Theorem: T(n) = T(n/2) + 1 is our recurrence relation. T(n/2) is for the recursive call and 1 is for the base case. So let's solve it.
T(n) = $T(n/2^2) + 2$
T(n) = $T(n/2^3) + 3$
Here we can see a pattern so let's generalize T(n) = $T(n/2^k) + k$ So here we can solve for $n/2^k = 1$ since it will eventually hit the base case. So $n/2^k = 1 -> n = 2^k$ using log rules we can rewrite this as $logn = k$. Therefore plugging it back into the recurrence relation $T(n) = 1 + logn$ which means the running time of this algorithm is $O(logn)$

(2) **(20 pts)** Write a function that determines if a string has the same number of 0's and 1's using a stack. The function must run in O(N) time. *You can assume there already exists a stack class and can just use it*

```
def same_one_zero(s):
    stack = Stack()
    for c in s:
        if c == '1':
            if stack.peek() == '0':
                stack.pop()
            else:
                stack.push(c)
        else:
            if stack.peek() == '1':
                stack.pop()
            else:
                stack.push(c)

    if stack.peek():
        return False
    else:
        return True
```

Here the idea is everytime we see a 1 we check if we already have seen a 0, if so pop it off, otherwise push a 1. The same idea goes for a 0. This way no matter what order they occur if they are even the stack will always be empty if the number of 0's and 1's are equal

(3) **(30 pts)** Write a method to determine if a positive integer, N, is prime in $O(\sqrt{N})$.

```
import math
def isPrime(x):
    sqrt = math.floor(math.sqrt(x))
    prime = False
    for i in range(2, sqrt):
        if x % i == 0:
            prime = True

    return prime
```

(4) **(15 pts)** Given a list of numbers from [1-100] with one number missing, determine which number is missing in O(N) time using basic arithmetic and a max of two new variables *The original list does not count as one of the two variables.*

```
def missing(l):
    sum = 0
    for i in range(len(l)):
        sum += l[i]
    return (100*101)/2 - sum
```

This one is where we exploit some math principles. That if you have a list in a range from $x_1, ..., x_n$ and there is a number missing. We can simply take the sum of the list and subtract it from the summation of number from x_1 to x_n. This will give us the missing number. Since the list started at 1 we can can used the closed form solution of the summation.

(5) **(20 pts)** Write a method to determine if a string has matching parenthesis for the set of all parenthesis {},(),[] using only one stack.

```
def paren_match(s):
    stack = Stack()
    for c in s:
        if c in ['(', '{', '[']:
            s.push(c)
        elif c == ']' and s.peek() == '[':
            s.pop()
        elif c == '}' and s.peek() == '{':
            s.pop()
        elif c == ')' and s.peek() == '(':
            s.pop()
        else:
            return False
    if s.peek():
        return False
    else:
        return True
```

This is similar to the one we did in class with just basic parenthesis. However in this case we need to make sure when we see a closing parenthesis that opening parenthesis at the top of the stack is the same type as the closing. If they are we can simply pop it off the stack otherwise the parenthesis are no longer balanced as the original opening parenthesis has no matching correct close.