Lab 2

Exercise 1: In my high school, students were given letter grades based on their numerical scores. Here is the table of which numerical scores correspond to which letter grades:

Score	Grade
99-100	A+
96-98	A
93-95	A-
90-92	B+
87-89	B
84-86	B-
81-83	C+
78-80	C
75-77	C-
70-74	D
0-69	F

Write a function `convertScoreToGrade`(n) which takes an `int` numerical score n and returns a string corresponding to the letter grade in my high school.

Example solution:

```
def convertScoreToGrade(n):
    if n>=99:
        return 'A+'
    elif n>=96:
        return 'A'
    elif n>=93:
        return 'A-'
    elif n>=90:
        return 'B+'
    elif n>=87:
        return 'B'
    elif n>=84:
        return 'B-'
    elif n>=81:
        return 'C+'
    elif n>=78:
        return 'C'
    elif n>=75:
        return 'C-'
    elif n>=70:
        return 'D'
```

```
    else:
        return 'F'
```

Exercise 2: Write a function `listOfWords`(s) which takes as input a sentence s and outputs a list containing all the words. We'll assume that the sentence s is just a list of words each separated by a single space. So, for example, `listOfWords`('How are you today') should return the list ['How', 'are', 'you', 'today'].

Example solution:

```
def listOfWords(s):
    cur = ''
    ret = []
    for c in s:
        if c==' ':
            ret += [cur]
            cur = ''
        else:
            cur += c
    ret += [cur]
    return ret
```

Exercise 3: Recall, an integer is prime if it is larger than 1 and it has no divisors other than 1 and itself. Write a function `isPrime`(n) which returns `True` if n is prime, and returns `False` otherwise. Then, write a function `listOfPrimes`(n) which returns a list of all primes, in order, between 2 and n — you might want to call the function `isPrime` from within `listOfPrimes`. Notice that if you try to run `listOfPrimes` on a large number, it will take a long time. Is there a way to write `isPrime` so that `listOfPrimes`(n) takes only a few seconds to return its result when n is 1000000?

Example solution:

```
def isPrime(n):
    if n<2:
        return False
    for x in xrange(2,n):
        if x*x > n:
            break
        if n%x==0:
            return False
    return True

def listOfPrimes(n):
    ret = []
    for x in xrange(2, n+1):
        if isPrime(x):
```

```
            ret += [x]
    return ret
```

Notice that we can break in `isPrime` if `x*x>n`, since if a number n has a divisor other than 1 or itself, then one of its divisors must be less than $\sqrt{n}$. By breaking early, the loop only runs $\sqrt{n}$ times instead of n times, so it runs faster and we can handle larger numbers.

Exercise 4: Let's call an integer *well-spaced* if when you write down its prime factorization, you don't need to use two adjacent primes. For example, 10 is well-spaced since $10 = 2 \cdot 5$, and $2, 5$ are not adjacent primes since the prime 3 is in between them. However, 6 is not well-spaced, since $6 = 2 \cdot 3$, and $2, 3$ are adjacent primes, and neither is $154 = 2 \cdot 7 \cdot 11$, since $7, 11$ are adjacent primes. Any prime number itself is of course also well-spaced. Write a function `wellSpaced`(n) which returns `True` if n is well-spaced, and returns `False` otherwise. Is 147525307 well-spaced? How about 147914243?

Example solution:

```
def wellSpaced(n):
    primeList = listOfPrimes(n+1)
    for i in xrange(len(primeList)-1):
        if n%primeList[i]==0 and n%primeList[i+1]==0:
            return False
    return True
```

The above solution is correct, but it takes way too long on the numbers 147525307 and 147914243 given above. The problem is that our loop to check whether n is prime takes about $\sqrt{n}$ steps, so `listOfPrimes`(n) takes roughly $n \cdot \sqrt{n}$ steps. When $n = 147525307$, $n \cdot \sqrt{n}$ is a lot of steps so it takes our computer a long time.

We can derive a faster solution by noticing that whenever we have $n = a \cdot b$, one of a, b must be at most $\sqrt{n}$. Thus, if $a < b$ are adjacent primes that multiply to n, then $a \leq \sqrt{n}$, and either $b \leq \sqrt{n}$ or it is the smallest prime larger than $\sqrt{n}$. This observation leads to a much faster implementation.

Faster example solution:

```
def wellSpaced(n):
    # First calculate the floor of the square root of n
    squareRoot = 0
    while squareRoot*squareRoot <= n:
        squareRoot += 1
    squareRoot -= 1

    # Get the list of primes up to the square root
    primeList = listOfPrimes(squareRoot)

    # Add on the very next prime after the square root
    lastPrime = primeList[len(primeList) - 1] + 1
```

```
    while not isPrime(lastPrime):
        lastPrime += 1
    primeList += [lastPrime]

    # Now check for adjacent prime divisors
    for i in xrange(len(primeList)-1):
        if n%primeList[i]==0 and n%primeList[i+1]==0:
            return False
    return True
```

The above code is now much faster, and we find that neither of the given numbers is well-spaced.

Exercise 5: An integer is said to be a *palindrome* if its digits are the same forward and backwards (not including leading zeroes). For example, 12321 is a palindrome, as is 5. 1231 on the other hand is not a palindrome, and neither is 50 (remember we are not including leading zeroes). Write a function `isPalindrome`(n) which returns `True` if n is a palindrome and `False` otherwise.

Example solution:

```
def isPalindrome(n):
    s = str(n)
    for i in xrange(len(s)):
        if s[i] != s[len(s)-i-1]:
            return False
    return True
```

Exercise 6: Write a function `nextPalindrome`(n) which returns the smallest integer m larger than n such that m is a palindrome. For example, `nextPalindrome`(9) should return 11, and `nextPalindrome`(12) should return 22.

Example solution:

```
def nextPalindrome(n):
    x = n+1
    while not isPalindrome(x):
        x += 1
    return x
```

Faster example solution: Let us first define a function to reverse strings.

```
def reverse(s):
    t = ''
    for i in xrange(len(s)):
        t += s[len(s) - i - 1]
    return t
```

The following implementation of `nextPalindrome` is much faster than the last, since in the previous implementation we might have to iterate in the `while` loop a long time before reaching the next palindrome. The implementation below, rather than iterating to find the next palindrome, just figures out what the next palindrome has to look like then creates it.

```
def nextPalindrome(n):
    if n<9:
        return n+1
    elif n==9:
        return 11
    s = str(n)
    t = s[0:len(s)/2]
    trev = reverse(t)
    if int(t + trev) > n:
        return int(t + trev)
    if len(s)%2==1 and int(t + s[len(s)/2] + trev)>n:
        return int(t + s[len(s)/2] + trev)
    if len(s)%2==1 and s[len(s)/2]!='9':
        return int(t + str(int(s[len(s)/2]) + 1) + trev)
    else:
        tInc = str(int(t)+1)
        tIncRev = reverse(tInc)
        if len(tInc)==len(t):
            if len(s)%2==0:
                return int(tInc + tIncRev)
            else:
                return int(tInc + '0' + tIncRev)
        else:
            ret = '1'
            for i in xrange(len(s)-1):
                ret += '0'
            ret += '1'
            return int(ret)
```