MATH182 HOMEWORK #5
DUE July 29, 2020

Note: while you are encouraged to work together on these problems with your classmates, your final work should be written in your own words and not copied verbatim from somewhere else. You need to do at least five (5) of these problems. All problems will be graded, although the score for the homework will be capped at $N :=$ (point value of one problem) $\times 5$ and the homework will be counted out of N total points. Thus doing more problems can only help your homework score. For the programming exercise you should submit the final answer (a number) *and* your program source code.

Exercise 1. *Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights with two distinguished vertices s and t. Describe a dynamic programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?*

Exercise 2. *A **palindrome** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).*

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

Solution. The following is a bottom-up dynamic programming solution which also iteratively (non-recursively) determines the palindrome string:

```
def longest_palindrome(my_str):
    m=[]
    #m[i][j] will denote the longest palindrome contained in my_str[i:j+1]
    s=[]
    #s[i][j] will assist in reconstructing the longest palindrome according to the
     code:
    #s[i][j] = 0 means the longest palindrome of s[i:j+1] is contained in s[i+1:j
    +1]
    #s[i][j] = 1 means the longest palindrome of s[i:j+1] is contained in s[i:j]
    #s[i][j] = 2 means s[i] = s[j] and the longest palindrome of s[i:j+1] is of
    the form s[i]+(longest palindrome in s[i+1:j] + s[j])

    #make m and s the appropriate sizes
    for i in range(0,len(my_str)):
        m.append([])
        s.append([])
        for j in range(0,len(my_str)):
            m[-1].append(0)
            s[-1].append(0)

    #initialize the diagonals in both m and s
    for i in range(0,len(my_str)):
        m[i][i] = 1
        s[i][i] = 0

    for l in range(2,len(my_str)+1):
        for i in range(0,len(my_str)-l+1):
            j = i+l-1
            m[i][j]=m[i+1][j]
            if m[i][j]<m[i][j-1]:
                m[i][j]=m[i][j-1]
                s[i][j]=1
            if my_str[i]==my_str[j] and 2+m[i+1][j-1]>m[i][j]:
                m[i][j] = 2+m[i+1][j-1]
                s[i][j]=2

    length = m[0][len(my_str)-1]
    #pal_list is an array which will store the characters of the palindrome as we
   recover them from s
    pal_list = []
    #initialize pal_list
    for i in range(0,length):
        pal_list.append(0)

    #i points to the current left index in my_str
    i=0
    #pal_i points to the current left index in pal_list
    pal_i = 0
    #j points to the current right index in my_str
    j=len(my_str)-1
    #pal_j points to the current right index in pal_list
    pal_j = length-1
    while j-i>0:
        #if we are in case 0 then decrement j
        if s[i][j] == 0:
            i+=1
        #if we are in case 1 then increment i
        elif s[i][j] == 1:
            j-=1
        #if we are in case 2 then we need to write two more characters to pal_list
    and modify all four indices
        elif s[i][j] == 2:
            pal_list[pal_i] = my_str[i]
            pal_list[pal_j] = my_str[j]
            i += 1
            pal_i += 1
            j -= 1
            pal_j -= 1
    #if i=j, then our palindrome is of odd length with a unique middle character:
    if i==j:
        pal_list[pal_i] = my_str[i]

    return ''.join(pal_list)

```

The running time is $\Theta(n^2)$. □

Exercise 3. *Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.*

(1) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
(2) Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal optimal solution.
(3) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.
(4) Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Proof. (1) Below is the pseudocode for this greedy algorithm (also implementable using a switch statement)

```
MakeChange(quantity)
Create coin[1..4] = ⟨0, 0, 0, 0⟩
// array coin will hold number of quarters in coin[1],
// dimes in coin[2], nickels in coin[3], and pennies in coin[4]
while quantity > 0
    if quantity ≥ 0.25
        coin[1] = coin[1] + 1
        quantity = quantity − 0.25
    elseif quantity ≥ 0.1
        coin[2] = coin[2] + 1
        quantity = quantity − 0.1
    elseif quantity ≥ 0.05
        coin[3] = coin[3] + 1
        quantity = quantity − 0.05
    elseif quantity ≥ 0.01
        coin[4] = coin[4] + 1
        quantity = quantity − 0.01
return coin
```

(2)

(3) The set $D = \{1, 3, 4, 5\}$ does not work for the greedy algorithm For instance, with $n = 7$, the greedy algorithm yields the solution $7 = 5 + 2 \times 1$ (3 coins), whereas $7 = 4 + 3$ is the optimal solution.

(4) □

Exercise 4. *Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose the residents of these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.*

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Solution. We can denote the houses as $h_1, \ldots, h_n \in \mathbb{R}$, where $h_1 < h_2 < \cdots < h_n$. Furthermore, we will denote the towers as $t_1, \ldots, t_k \in \mathbb{R}$, where $t_1 < t_2 < \cdots < t_k$. For each $j = 1, \ldots, n, n+1$, we will denote by H_j all houses starting at h_j and going to the east:

$$H_j := \{h_\ell : \ell \geq j\}$$

(so $H_{n+1} = \emptyset$). These will be our subproblems, and H_1 is the original problem. The following claim shows we can always make the greedy choice of choosing the next tower to be four miles to the east of the western most uncovered house:

Claim. *Consider any nonempty subproblem H_j (so $j \leq n$). Consider the first house h_j of this subproblem and the tower $t := h_j + 4$ (the eastern most point which we can put a tower which covers h_j). Then t is included in some optimal solution to the subproblem H_j.*

Proof of claim. Let $\{t'_\ell, \ldots, t'_k\}$ be an arbitrary optimal solution to the subproblem H_j, with $t'_\ell < \cdots < t'_k$. We will show that exchanging t'_ℓ with $t = h_j + 4$ is also an optimal solution. Let $h_j, h_{j+1}, \ldots, h_m$ be those houses with the property $t'_\ell - 4 \leq h \leq t'_\ell + 4$ (i.e., those houses covered by t'_ℓ). Since h_j is the first house and t'_ℓ is the first tower, h_j must be included in this list. Thus $t'_\ell - 4 \leq h_j = t - 4$. Thus $t'_\ell \leq t$. In particular

$$t - 4 \; = \; h_j \; < \; h_{j+1} \; < \; \cdots \; < \; h_m \; \leq \; t'_\ell + 4 \; \leq \; t + 4.$$

Thus the tower t covers at least the same houses that t'_ℓ covers. Thus $\{t, t'_{\ell+1}, \ldots, t'_k\}$ is a solution also, and of the same size, so it is an optimal solution. □

Here is a pseudocode implementation of this greedy algorithm:

```
TOWERS(Houses)
1  FirstHouse = 1 // Index of first uncovered house
2  Initialize empty array Towers
3  while FirstHouse ≤ Houses.length
4      Towers.append(Houses[FirstHouse] + 4)
5      while FirstHouse ≤ Houses.length and Houses[FirstHouse] ≤ Towers[Towers.length]
6          FirstHouse = FirstHouse + 1
7  return Towers
```

TOWERS works as follows:

(1) We take as input an array *Houses* which contains $\langle h_1, h_2, \ldots, h_n \rangle$ (assumed to be in increasing sorted order).
(2) In line 1 we initialize the variable *FirstHouse* to be 1. This variable corresponds to the western-most uncovered house. It also represent which subproblem $H_{FirstHouse}$ we are currently solving.
(3) In line 2 we initialize an empty array *Towers* of length 0. We will add towers to this list as we determine them.
(4) While $FirstHouse \leq Houses.length$, we still have a nonempty subproblem, so we still have towers to add. The **while** loop adds towers using the greedy choice until we have finished covering all houses.
(5) In line 4 we add a new tower to the list of towers in accordance with the greedy choice.
(6) In the **while** loop of lines 5-6 we find the next subproblem we need to address.
(7) In line 7 we return the list *Towers*.

This algorithm runs in $\Theta(n)$ time. □

Exercise 5. *Given a list of n natural numbers $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G = (V, E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (That is, if $V = \{v_1, \ldots, v_n\}$, then the degree of v_i should be exactly d_i.) G should not contain multiple edges between the same pair of nodes, or "loop" edges with both endpoints equal to the same node.*

Exercise 6. *A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.*

(1) Prove that in a breadth-first search of an undirected graph, the following properties hold:
  (a) There are no back edges and no forward edges.
  (b) For each tree edge (u, v), we have $v.d = u.d + 1$.
  (c) For each cross edge (u, v), we have $v.d = u.d$ or $v.d = u.d + 1$.

(2) Prove that in a breadth-first search of a directed graph, the following properties hold:
  (a) There are no forward edges.
  (b) For each tree edge (u, v), we have $v.d = u.d + 1$.
  (c) For each cross edge (u, v), we have $v.d \leq u.d + 1$.
  (d) For each back edge (u, v), we have $0 \leq v.d \leq u.d$.

Proof. (1)(a)
(b)
(c)
(2)(a)
(b)
(c)
(d) □

Exercise 7. *An* ***Euler tour*** *of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.*

(1) Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) *for each vertex $v \in V$.*
(2) Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists.

Exercise 8. *Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.*

We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be theset of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a ***second-best minimum spanning tree*** *is a spanning tree T such that $w(T) = \min_{T'' \in \mathcal{T} \setminus \{T'\}}\{w(T'')\}$.*

(1) Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.
(2) Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T \setminus \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.
(3) Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$-time algorithm that, given T, computes $\max[u, v]$ for all $u, v \in V$.
(4) Give an efficient algorithm to compute the second-best minimum spanning tree of G.

Exercise 9 (Programming exercise)**.** *All square roots are periodic when written as continued fractions and can be written in the form:*

$$\sqrt{N} \;=\; a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \cdots}}}$$

For example, let us consider $\sqrt{23}$:

$$\sqrt{23} \;=\; 4 + \sqrt{23} - 4 \;=\; 4 + \frac{1}{\frac{1}{\sqrt{23}-4}} \;=\; 4 + \frac{1}{1 + \frac{\sqrt{23}-3}{7}}$$

If we continue we would get the following expansion:

$$\sqrt{23} = 4 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{8 + \cdots}}}}$$

The process can be summarized as follows:

$$\begin{aligned}
a_0 &= 4, & \frac{1}{\sqrt{23}-4} &= \frac{\sqrt{23}+4}{7} &= 1 + \frac{\sqrt{23}-3}{7} \\
a_1 &= 1, & \frac{7}{\sqrt{23}-3} &= \frac{7(\sqrt{23}+3)}{14} &= 3 + \frac{\sqrt{23}-3}{2} \\
a_2 &= 3, & \frac{2}{\sqrt{23}-3} &= \frac{2(\sqrt{23}+3)}{14} &= 1 + \frac{\sqrt{23}-4}{7} \\
a_3 &= 1, & \frac{7}{\sqrt{23}-4} &= \frac{7(\sqrt{23}+4)}{7} &= 8 + \sqrt{23}-4 \\
a_4 &= 8, & \frac{1}{\sqrt{23}-4} &= \frac{\sqrt{23}+4}{7} &= 1 + \frac{\sqrt{23}-3}{7} \\
a_5 &= 1, & \frac{7}{\sqrt{23}-3} &= \frac{7(\sqrt{23}+3)}{14} &= 3 + \frac{\sqrt{23}-3}{2} \\
a_6 &= 3, & \frac{2}{\sqrt{23}-3} &= \frac{2(\sqrt{23}+3)}{14} &= 1 + \frac{\sqrt{23}-4}{7} \\
a_7 &= 1, & \frac{7}{\sqrt{23}-4} &= \frac{7(\sqrt{23}+4)}{7} &= 8 + \sqrt{23}-4
\end{aligned}$$

It can be seen that the sequence is repeating. For conciseness, we use the notation $\sqrt{23} = [4; (1,3,1,8)]$, *to indicate that the block* $(1,3,1,8)$ *repeats indefinitely.*

The first ten continued fraction representations of (irrational) square roots are:

$$\begin{aligned}
\sqrt{2} &= [1;(2)] \quad \textit{period=1} \\
\sqrt{3} &= [1;(1,2)] \quad \textit{period=2} \\
\sqrt{5} &= [2;(4)] \quad \textit{period=1} \\
\sqrt{6} &= [2;(2,4)] \quad \textit{period=2} \\
\sqrt{7} &= [2;(1,1,1,4)] \quad \textit{period=4} \\
\sqrt{8} &= [2;(1,4)] \quad \textit{period=2} \\
\sqrt{10} &= [3;(6)] \quad \textit{period=1} \\
\sqrt{11} &= [3;(3,6)] \quad \textit{period=2} \\
\sqrt{12} &= [3;(2,6)] \quad \textit{period=2} \\
\sqrt{13} &= [3;(1,1,1,1,6)] \quad \textit{period=5}
\end{aligned}$$

Exactly four continued fractions, for $N \leq 13$, *have an odd period.*

How many continued fractions for $N \leq 10000$ *have an odd period?*

Exercise 10 (Programming exercise)**.** *Consider quadratic Diophantine equations of the form:*

$$x^2 - Dy^2 = 1$$

For example, when $D = 13$, the minimal solution in x is $649^2 - 13 \times 180^2 = 1$.

It can be assumed that there are no solutions i positive integers when D is square.

By finding minimal solutions in x for $D = \{2, 3, 5, 6, 7\}$, we obtain the following:

$$\begin{aligned} 3^2 - 2 \times 2^2 &= 1 \\ 2^2 - 3 \times 1^2 &= 1 \\ 9^2 - 5 \times 4^2 &= 1 \\ 5^2 - 6 \times 2^2 &= 1 \\ 8^2 - 7 \times 3^2 &= 1 \end{aligned}$$

Hence, by considering minimal solutions in x for $D \leq 7$, the largest x is obtained when $D = 5$.

Find the value of $D \leq 1000$ in minimal solutions of x for which the largest value of x is obtained.

(This link might be helpful: `https://en.wikipedia.org/wiki/Pell%27s_equation`*)*

Exercise 11 (Programming exercise)**.** *Consider the fraction $n/3$, where n and d are positive integers. If $n < d$ and $\gcd(n, d) = 1$, it is called a* ***reduced proper fraction****.*

If we list the set of reduced proper fractions for $d \leq 8$ in ascending order of size, we get:

$$1/8, 1/7, 1/6, 1/5, 1/4, 2/7, 1/3, 3/8, 2/5, 3/7, 1/2, 4/7, 3/5, 5/8, 2/3, 5/7, 3/4, 4/5, 5/6, 6/7, 7/8$$

It can be seen that $2/5$ is the fraction immediately to the left of $3/7$.

By listing the set of reduced proper fractions for $d \leq 1000000$ in ascending order of size, find the numerator of the fraction immediately to the left of $3/7$.

Solution. First we have algorithms for computing GCD (Euclidean Algorithm) and for comparing two fractions in term of their numerators and denominators:

```python
def gcd(a,b):
    if b==0:
        return a
    return gcd(b,a%b)

#checks if a/b<c/d
def frac_less_than(a,b,c,d):
    return a*d<c*b
```

Next we run the following script. It initializes the best fraction so far at $1/3$. Then it checks all denominators from $d = 4$ to 1000000. For each d it computes the best possible numerator. Then it only considers the fraction n/d if $\gcd(n, d) = 1$ (otherwise we've already seen this fraction before for a lower d):

```python
best_num = 1
best_den = 3

for d in range(4,1000001):
    if d%7==0:
        n = int(3*d/7)-1
    else:
        n = int(3*d/7)
    #only check if we haven't seen this fraction before
    if gcd(n,d)==1:
        if frac_less_than(best_num,best_den,n,d):
            best_num=n
            best_den=d
            #print(str(best_num)+"/"+str(best_den))
```

```python

print(best_num)
```

This prints the following answer:

```
428570
```

□

Exercise 12 (Programming exercise)**.** *It is possible to write ten as the sum of primes in exactly five different ways:*

$$7+3$$
$$5+5$$
$$5+3+2$$
$$3+3+2+2$$
$$2+2+2+2+2$$

What is the first value which can be written as the sum of primes in over five thousand different ways?

Proof. First we have some algorithms which grows and maintains a list of primes:

```python
#this should always contain consecutive primes
primes = [2,3,5]

def next_prime():
    #This will grow primes by one more prime
    value = primes[-1]
    orig_length = len(primes)

    while len(primes) == orig_length:
        value += 1
        if is_prime(value):
            primes.append(value)

def is_prime(N):
    #assumes that primes is big enough to detect primality of N

    for p in primes:
        if N%p == 0:
            return False
    return True
```

Next we have some algorithms which grows and maintains a two-dimensional array $ways_table$:

```python
ways_table = [[1],[0],[1],[0,1],[1,1],[0,1,2]]
#ways_table[n,k] = number of ways to write n as a sum of primes using primes p_0
    ,...,p_k

#this finds the greatest i such that primes[i]<=n, this is the largest prime which
     might be used to sum to n
def greatest_prime_index(n):
    while primes[-1]<=n:
        next_prime()
    index=len(primes)-1
    while primes[index]>n:
        index-=1
    return index

#If we attempt to lookup ways_table[n][k] with k to large, this will return ways_table[n][k]. This saves us from making the ways_table bigger than it needs to be
def ways_table_lookup(n,k):
    if len(ways_table[n])<=k:
        return ways_table[n][-1]
    return ways_table[n][k]

#Fills in the next row of ways_table, for n it fills in ways_table[n][0] through ways_table[n][greatest_prime_index(n)]
def next_ways_row():
    n = len(ways_table)
    ways_table.append([])
    max_index = greatest_prime_index(n)
    ways_table[-1].append(ways_table_lookup(n-2,0))
    for i in range(1,max_index+1):
        ways_table[-1].append(ways_table_lookup(n-primes[i],i)+ways_table_lookup(n,i-1))

#This returns the # of ways we can write n as a sum of primes
def ways(n):
    while len(ways_table)<=n:
        next_ways_row()
    return ways_table[n][-1]
```

Finally we run the following script:

```python
n=0
while ways(n)<=5000:
    n+=1
print(n)
```

which prints the answer:

```
71
```

□

Exercise 13 (Programming exercise)**.** *Do Project Euler Problem 83: Path sum: four ways* `https://projecteuler.net/problem=83`*. As a warmup, you might want to do problems 81 and 82 first.*