MATH182 HOMEWORK #4
DUE July 19, 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 seven (7) of these problems. All problems will be graded, although the score for the homework will be capped at $N :=$ (point value of one problem) $\times 7$ 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. *This problem is about heaps:*

(1) What are the minimum and maximum numbers of elements in a heap of height h?
(2) Show that an n-element heap has height $\lfloor \lg n \rfloor$.
(3) Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Solution. (1) We will prove the following claim:

Claim. *Suppose $h \geq 0$ and A is a heap. The following are equivalent:*

(1) The height of A is h,
(2) $2^h \leq A.\mathit{heap\text{-}size} \leq 2^{h+1} - 1$.

Proof of claim. We will prove this by strong induction on $h \geq 0$:

$$P(h): \quad \text{“The height of } A \text{ is } h \text{ iff } 2^h \leq A.\mathit{heap\text{-}size} \leq 2^{h+1} - 1\text{”}$$

(Base case) Suppose $h = 0$ and A is a heap. Then A has height 0 iff the root has height 0 iff the root does not have any children iff A has size 1.

(Inductive step) Assume for some $h \geq 0$ we know that $P(h)$ holds. Suppose A is a heap with height $h+1$. Let $n = A.\mathit{heap\text{-}size}$. Then there is a path of length $h+1$ from the root to leaf indexed n and so there is a path of length h from the root to $\textsc{Parent}(n) = \lfloor n/2 \rfloor$. Since $A[1\,..\,\textsc{Parent}(n)]$ is a heap of size h, we know by $P(h)$ that

$$2^h \;\leq\; \textsc{Parent}(n) \;=\; \lfloor n/2 \rfloor \;\leq\; 2^{h+1} - 1 \;<\; 2^{h+1}.$$

I.e.,

$$2^h \;\leq\; \lfloor n/2 \rfloor \;<\; 2^{h+1}.$$

Thus

$$2^h \;\leq\; n/2 \;<\; 2^{h+1},$$

since 2^h and 2^{h+1} are integers. Multiplying by 2 yields

$$2^{h+1} \;\leq\; n \;<\; 2^{h+2},$$

and since n is an integer,

$$2^{h+1} \;\leq\; n \;\leq\; 2^{h+2} - 1.$$

Reversing these steps show the converse direction. Thus the inductive step, and hence the claim, is proved. □

(2) Suppose A is an n-element heap. The height of A is the unique $h \in \mathbb{N}$ such that $2^h \leq n \leq 2^{h+1} - 1$, equivalently, the unique $h \in \mathbb{N}$ such that $2^h \leq n < 2^{h+1}$. Taking lg yields

$$h \ \leq \ \lg n \ < \ h + 1,$$

and thus $h = \lfloor \lg n \rfloor$.

(3) The first leaf will be one more than the parent of the last leaf (since the parent of the last leaf is the greatest node of height 1, then next node must be height 0). Thus the first leaf is $\text{Parent}(n) + 1 = \lfloor n/2 \rfloor + 1$. □

Exercise 2. *The code for* MAX-HEAPIFY *is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient* MAX-HEAPIFY *that uses an iterative control construct (a loop) instead of recursion.*

Solution. In the following implementation of MAX-HEAPIFY we use a **while** loop that continues to run as long as i has at least a left child.

```
MAX-HEAPIFY($A, i$)
1   $l = \text{Left}(i)$
2   $r = \text{Right}(i)$
    // The while loop will continue as long as $i$ has a left child, unless we break out of it
3   while $l \leq A.\textit{heap-size}$
4       $\textit{largest} = i$
5       if $A[l] > A[i]$
6           $\textit{largest} = l$
7       if $r \leq A.\textit{heap-size}$ and $A[r] > A[\textit{largest}]$
8           $\textit{largest} = r$
9       if $\textit{largest} == i$
10          Break                // Exit loop, we are done.
11      else exchange $A[i]$ with $A[\textit{largest}]$
12          $i = \textit{largest}$
13          $l = \text{Left}(i)$
14          $r = \text{Right}(i)$
```

MAX-HEAPIFY works as follows:

(1) In lines 1-2 we set the indices of the left and right child of i.
(2) The **while** loop in lines 3-14 will run as long as i has at least a valid left child, or until we break out of the loop if we are finished.
(3) In line 4 we initialize the largest index to i.
(4) In lines 5-6 we check if the left child is larger than i, if it is, then we set largest to l.
(5) In lines 7-8 we check first if i has a right child, and if it does, is it the largest of the three. If it is, we set the largest to r.
(6) In lines 9-10, we check if the largest is i itself. If it is, then we are done with max-heapifying at i, so we break out of the loop and the algorithm exits.
(7) Otherwise, in lines 11-14 we exchange $A[i]$ with $A[\textit{largest}]$ (guaranteed to be one of the children). Then we redefine i to be the largest child, and redefine the left and right children accordingly.

□

Exercise 3. *Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.*

Exercise 4. *Argue the correctness of* HEAPSORT *using the following loop invariant:*

> *(Loop Invariant) At the start of each iteration of the* **for** *loop of lines 2-5, subarray $A[1\,..\,i]$ is a max-heap containing the i smallest elements of $A[1\,..\,n]$, and the subarray $A[i+1\,..\,n]$ contains the $n-i$ largest elements of $A[1\,..\,n]$, sorted.*

Exercise 5. *What is the running time of* HEAPSORT *on an array A of length n that is already sorted in increasing order? What about decreasing order?*

Exercise 6. *Argue the correctness of* HEAP-INCREASE-KEY *using the following loop invariant:*

> *(Loop Invariant) At the start of each iteration of the* **while** *loop of lines 4-6, $A[\text{PARENT}(i)] \geq A[\text{LEFT}(i)]$ and $A[\text{PARENT}(i)] \geq A[\text{RIGHT}(i)]$, if these nodes exist, and the subarray $A[1\,..\,A.\mathit{heap\text{-}size}]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.*

You may assume that the the subarray $A[1\,..\,A.\mathit{heap\text{-}size}]$ satisfies the max-heap property at the time HEAP-INCREASE-KEY *is called.*

Exercise 7. *A* ***d-ary heap*** *is just like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.*

(1) How would you represent a d-ary heap in an array?
(2) What is the height of a d-ary heap of n elements in terms of n and d?
(3) Give an efficient implementation of EXTRACT-MAX *in a d-ary max-heap. Analyze its running time in terms of d and n.*
(4) Give an efficient implementation of INSERT *in a d-ary max-heap. Analyze its running time in terms of d and n.*
(5) Give an efficient implementation of INCREASE-KEY(A, i, k), *which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.*

Exercise 8. *An $m \times n$* ***Young tableau*** *is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus a Young tableau can be used to hold $r \leq mn$ finite numbers.*

(1) Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.
(2) Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Arge that Y is full (contains mn elements) if $Y[m,n] < \infty$.
(3) Give an algorithm to implement EXTRACT-MIN *on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. (Hint: Think about* MAX-HEAPIFY.*) Define $T(p)$, where $p = m + n$, to be the maximum running time of* EXTRACT-MIN *on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m+n)$ time bound.*
(4) Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.
(5) Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau in to sort n^2 numbers in $O(n^3)$ time.
(6) Given an $O(m+n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

Exercise 9. *Explain how to implement two stacks in one array $A[1\,..\,n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The* PUSH *and* POP *operations should run in $O(1)$ time.*

Exercise 10. *Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the* *sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.*

Exercise 11. *Modify* Memoized-Cut-Rod *to return not only the value but the actual solution, too.*

Exercise 12 (Programming Exercise)**.** *Let $d(n)$ be defined as the sum of proper divisors of n (numbers less than n which divide evenly into n). If $d(a) = b$ and $d(b) = a$, where $a \neq b$, then a and b are an* ***amicable pair*** *and each of a and b are called* ***amicable numbers****.*

For example, the proper divisors of 220 are $1, 2, 4, 5, 10, 11, 20, 22, 44, 55$ and 110; therefore $d(220) = 284$. The proper divisors of 284 are $1, 2, 4, 71$ and 142; so $d(284) = 220$.

Evaluate the sum of all the amicable numbers under 10000. (This page might be helpful: `https://en.wikipedia.org/wiki/Divisor_function`)

Solution. First we have some subroutines for growing an maintaining a list of consecutive prime numbers:

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

def nth_primes(N):
    if len(primes) >= N:
        print(primes[N-1])
    else:
        for i in range(1,N-len(primes)+1):
            next_prime()
    print(primes[N-1])

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 a subroutine which computes the number of divisors of N directly:

```python
#this function will compute the sum of divisors of N directly
def sigma(N):
    copy=N
    #first make sure we have enough primes
    while primes[-1]<N:
        next_prime()
    prime_factorization = {}
    while N>1:
        for p in primes:
            if p<=N:
                exp=0
                while N%p==0:
                    N/=p
                    exp+=1
                if exp>=1:
                    prime_factorization[p]=exp
    divisors=1
    for p in prime_factorization.keys():
        divisors*=int((p**(prime_factorization[p]+1)-1)/(p-1))
    return divisors-copy
```

Next we have a dynamic programming version which computes the number of divisors and stores the answer:

```
divisors = [0,0]

def dynamic_sigma(N):
    if len(divisors)>N:
        return divisors[N]
    while len(divisors)<=N:
        divisors.append(sigma(len(divisors)))
    return divisors[-1]
```

Next we have a subroutine which detects whether a number is amicable:

```
def is_amicable(a):
    b = dynamic_sigma(a)
    if dynamic_sigma(b)==a and a!=b:
        return True
    return False
```

Finally, we run the following script:

```
sum=0
for i in range(2,10000):
    if is_amicable(i):
        sum+=i
print(sum)
```

which returns the answer:

```
31626
```

□

Exercise 13 (Programming Exercise)**.** *A* ***perfect number*** *is a number for which the sum of its proper divisors is exactly equal to the number. For example, the sum of the proper divisors of* 28 *would be* $1+2+4+7+14=28$*, which means that* 28 *is a perfect number.*

A number n *is called* ***deficient*** *if the sum of its proper divisors is less than* n *and it is called* ***abundant*** *if this sum exceeds* n*.*

As 12 *is the smallest abundant number,* $1+2+3+4+6=16$*, the smallest number that can be written as the sum of two abundant numbers is* 24*. By mathematical analysis, it can be shown that all integers greater than* 28123 *can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of two abundant numbers is less than this limit.*

Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

Solution. This code refers to some of the subroutines introduced in the solution to the previous problem. We have an algorithm for checking if a number is abundant:

```
def is_abundant(N):
    if dynamic_sigma(N)>N:
        return True
    return False
```

Next we have an algorithm checking if a number can be written as the sum of two abundant numbers:

```
def is_sum_abundant(N):
    for i in range(1,N//2+1):
        if is_abundant(i) and is_abundant(N-i):
            return False
    return True
```

Finally, we run the following script:

```
sum=0
for i in range(0,28123):
    if is_sum_abundant(i):
        sum+=i
print(sum)
```

which outputs:

```
4179871
```

□

Exercise 14 (Programming Exercise)**.** *The number* 3797 *has an interesting property. Being prime itself, it is possible to continuously remove digits from left to right, and remain prime at each stage:* $3797, 797, 97,$ *and* 7*. Similarly we can work from right to left:* $3797, 379, 37,$ *and* 3*.*

Find the sum of the only eleven primes that are both truncatable from left to right and right to left.

Note: $2, 3, 5,$ *and* 7 *are not considered to be truncatable primes.*

Solution. This solution also uses previous prime subroutines. This is not the most efficient algorithm as it takes several minutes. First we have a dynamic algorithm which stores whether or not a number is prime:

```
#An array of booleans: prime_bool[n] is True iff n is prime
prime_bool=[False, False, True]
def is_prime_dynamic(n):
    while len(prime_bool)<=n:
        while primes[-1]<len(prime_bool):
            next_prime()
        prime_bool.append(len(prime_bool) in primes)
    return prime_bool[n]
```

Next we have a dynamic algorithm which keeps track of the left-truncatable, right-truncatable and truncatable primes. It stops once it has found 15 truncatable primes (the four primes 2,3,5,7 and the eleven true truncatable primes we are looking for).

```python
left_truncatable = {2,3,5,7}
right_truncatable = {2,3,5,7}
truncatable = {2,3,5,7}
n=10
while len(truncatable)<15:
    if is_prime_dynamic(n):
        if (int(str(n)[1:]) in left_truncatable):
            left_truncatable.add(n)
        if (int(str(n)[:-1]) in right_truncatable):
            right_truncatable.add(n)
        if (n in right_truncatable) and (n in left_truncatable):
            truncatable.add(n)
    n+=1
```

Finally we run the following script:

```python
sum=0
for n in truncatable:
    if n>10:
        sum+=n
print(sum)
```

which prints

```
748317
```

□

Exercise 15 (Programming Exercise)**.** *If p is the perimeter of a right angle triangle with integer length sides, $\{a, b, c\}$, there are exactly three solutions for $p = 120$. $\{20, 48, 52\}$, $\{24, 45, 51\}$, $\{30, 40, 50\}$.*

For which value of $p \leq 1000$, is the number of solutions maximised? (This page might be helpful: https://en.wikipedia.org/wiki/Pythagorean_triple)

Solution. First we have a routine for the Euclidean algorithm:

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

Next we have the following script:

```python
#Keeps track of number of solutions for a given integer
num_sols=[]
#Initializes array to all zeros
for i in range(0,1001):
    num_sols.append(0)

#The first two for loops iterate through all possible pairs (m,n) which can
   generate a primitive pythagorean triple with permeter <=1000
for n in range(1,16):
    for m in range(n+1,16):
        #Only consider (m,n) if m and n are relatively prime
        if gcd(m,n)==1:
            perim = 2*m**2+2*m*n
            k=1
            #Consider all nonprimitive multiples w perimeter <=1000
            while perim*k <= 1000:
                num_sols[perim*k]+=1
                k+=1
max_so_far = 0
max_index = 0
for n in range(0,len(num_sols)):
    if num_sols[n]>max_so_far:
        max_so_far = num_sols[n]
        max_index = n

print(max_index)
```

which prints the following answer:

```
840
```

□

Exercise 16 (Programming Exercise)**.** *It is possible to show that the square root of two can be expressed as an infinite continued fraction.*

$$\sqrt{2} = 1 + \cfrac{1}{2 + \frac{1}{2 + \frac{1}{2 + \cdots}}}$$

By expanding this for the first four iterations, we get:

$$\begin{aligned} 1 + \frac{1}{2} &= \frac{3}{2} \\ 1 + \frac{1}{2 + \frac{1}{2}} &= \frac{7}{5} \\ 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}} &= \frac{17}{12} \\ 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}} &= \frac{41}{29} \end{aligned}$$

The next tree expansions are 99/70, 239/169, *and* 577/408, *but the eighth expansion,* 1393/985, *is the first example where the number of digits in the numerator exceeds the number of digits in the denominator.*

In the first one-thousand expansions, how many fractions (when put in lowest terms) contain a numerator with more digits than the denominator? (This page might be helpful: `https://en.wikipedia.org/wiki/Continued_fraction`*)*

Solution. Using the theory of continued fractions (found in the above wikipedia article), the nth fraction is defined recursively (automatically in lowest terms!!) as h_n/k_n by the recurrence

$$\begin{aligned} h_n &= a_n h_{n-1} + h_{n-2}, \quad h_{-1} = 1 \quad h_{-2} = 0 \\ k_n &= a_n k_{n-1} + k_{n-2}, \quad k_{-1} = 0 \quad k_{-2} = 1 \end{aligned}$$

where the sequence a_n in this case is $1, 2, 2, 2, 2, 2, \ldots$ (these are the numbers occurring in the continued fraction expression for $\sqrt{2}$). To implement this, we first have a subroutine which returns the value of a_n (we can modify this subroutine in the future for other continued-fraction problems):

```
def a_term(n):
    return (n!=0)+1
```

Next we have some dynamic programming routines which computes the values h_n and k_n as needed. Notice that we need to shift the indices by 2 since the base cases are h_{-2}, h_{-1} and k_{-2}, k_{-1}:

```python
h_terms = [0,1]
k_terms = [1,0]

def next_h_term():
    n = len(h_terms)-2
    h_terms.append(a_term(n)*h_terms[-1]+h_terms[-2])

def h_term(n):
    while len(h_terms)<n+3:
        next_h_term()
    return h_terms[n+2]

def next_k_term():
    n = len(k_terms)-2
    k_terms.append(a_term(n)*k_terms[-1]+k_terms[-2])

def k_term(n):
    while len(k_terms)<n+3:
        next_k_term()
    return k_terms[n+2]
```

Finally we run the following script. To compare the number of digits, we convert the integers to strings and then compare the lengths of those strings:

```python
counter=0
for n in range(1,1001):
    if len(str(h_term(n)))>len(str(k_term(n))):
        counter+=1
print(counter)
```

This prints out the final answer:

```
153
```

□