C200 Programming Assignment №6
Recursion, Optimization, and Numbers

Dr. M.M. Dalkilic
Computer Science
School of Informatics, Computing, and Engineering

Indiana University, Bloomington, IN, USA

March 11, 2025

Introduction

Due Date: 5:00 PM, Friday, March 14, 2025, but **only** for this assignment we will allow a late submission up until **March 16, 2025 at 5:00 PM, EST**. Please note that the assignment will need to be submitted to Gradescope **, and no help will be available after Friday, March 14, 2025 at 5:00 PM. We will not accept submissions after the late due date**. Any form of cheating or plagiarism will be dealt with seriously. If this results in a grade change, we are compelled by the University to send a formal notification to the Dean, please review the policy on academic misconduct in the syllabus.

As always, all the work should be with you and your partner; but *both* of you should contribute. Please remember:

- You will be submitting on Gradescope
- Your highest score of **10 submissions** will be used for your final score.
- Any function that is defined requires a docstring, please refer to lab 03 for what a docstring should have.
- Finally, manual grading will be done, so if you receive a 50 out of 100 on the autograder, it means the lowest grade you can receive is 50. The rest will come from manual grading of the problems.

If your timestamp is 5:01PM or later, the homework will not be graded. So **do not wait until 4:59 PM** to submit, make sure to also plan accordingly. If you have questions or problems with Gradescope, please visit office hours or make a post on Inscribe ahead of time. Since you are working in pairs, your paired partner is shown on canvas. Esure that you are using only features that we have discussed in lectures and labs.

Problem 1: Recursion and Tail Recursion

In class for the second lecture we practiced recursion on this fuction:

$$h(n) \quad = \quad \begin{cases} 1 & n \leq 0 \\ 2n + h(n-1) + h(n-2) & o.w. \end{cases} \tag{1}$$

If we tried our first technique to implement tail recursion, we'd have this:

$$htr(n, acc0, acc1) \quad = \quad \begin{cases} [acc0, acc1][n] & n \leq 2 \\ htr(n-1, acc1, 2n + acc0 + acc1) & o.w. \end{cases} \tag{2}$$

The first ten values are:

```
1  h(0) = 1; htr(0) = 1
2  h(1) = 4; htr(1) = 4
3  h(2) = 9; htr(2) = 9
4  h(3) = 19; htr(3) = 19
5  h(4) = 36; htr(4) = 40
6  h(5) = 65; htr(5) = 79
7  h(6) = 113; htr(6) = 153
8  h(7) = 192; htr(7) = 288
9  h(8) = 321; htr(8) = 533
10 h(9) = 531; htr(9) = 971
```

Observe that for arguments 4,5,... the values differ. In class I said an alternative approach is using a while loop where we can manually build the stack frames. In some cases, the only way is to build bottom-up as we did with memoization. To give a simple example, suppose we want to implement tail recursion for factorial bottom-up. We add an additional variable for the state as shown here.

```
def f(n):
    if n:
        return n * f(n-1)
    else:
        return 1

def fw(n):
    if n in [0,1]:
        return 1
    else:
        acc = 1
        i = 2
        while i < n + 1:
            acc *= i
            i += 1
    return acc

def ftr(n,i=2,acc=1):
    if n in [0,1]:
        return 1
    elif i < n + 1:
        return ftr(n,i+1,acc*i)
    else:
        return acc

for n in range(10):
    print(n,f(n),fw(n),ftr(n))
```

produces:

```
0 1 1 1
1 1 1 1
2 2 2 2
3 6 6 6
4 24 24 24
5 120 120 120
6 720 720 720
7 5040 5040 5040
8 40320 40320 40320
9 362880 362880 362880
```

the while loop is emulated in the tail recursion. We added an additional variable i and counted up. In fact, n is never changed. This allowed for a bottom-up tail recursion. Take this into account when solving the recursions.

In this problem, you'll implement the following recursive functions.

$$
\begin{aligned}
s(0) &= 0 && (3)\\
s(n) &= s(n-1)+n && (4)
\end{aligned}
$$

$$
\begin{aligned}
p(0) &= 10000 && (5)\\
p(n) &= p(n-1)+0.02p(n-1) && (6)
\end{aligned}
$$

$$
\begin{aligned}
c(1) &= 9 && (7)\\
c(n) &= 9c(n-1)+10^{n-1}-c(n-1) && (8)\\
& && (9)
\end{aligned}
$$

The following code:

```
for i in range(4):
    print(f"n = {i}")
    print(f"s(n) = {s(i)} s_memo(n) = {s_memo(i)}")
    print(f"p(n) = {p(i)} p_tail(n) = {p_tail(i)}")

for i in range(1,5):
        print(f"n = {i}")
        print(f"c(n) = {c(i)} c_tail(n) = {c_tail(i)} c_while(n) = {↩
            c_while(i)}")
```

produces:

```
n = 0
s(n) = 0 s_memo(n) = 0
p(n) = 10000 p_tail(n) = 10000
n = 1
s(n) = 1 s_memo(n) = 1
p(n) = 10200.0 p_tail(n) = 10200.0
n = 2
s(n) = 3 s_memo(n) = 3
p(n) = 10404.0 p_tail(n) = 10404.0
n = 3
s(n) = 6 s_memo(n) = 6
p(n) = 10612.08 p_tail(n) = 10612.08

n = 1
c(n) = 9 c_tail(n) = 9 c_while(n) = 9
n = 2
c(n) = 82 c_tail(n) = 82 c_while(n) = 82
```

```
n = 3
c(n) = 756 c_tail(n) = 756 c_while(n) = 756
n = 4
c(n) = 7048 c_tail(n) = 7048 c_while(n) = 7048
```

Programming Problem 1: Recursion on Numbers

- Complete each of the functions.
- Implement s using recursion and memoization.
- Implement p using recursion and tail recursion.
- Implement c using recursion, tail recursion, and a while loop.
- Include a docstring for each function.

2: Pyramids of Egypt

This recursion is an old mathematical way to build a pyramid of numbers.

$$
\begin{aligned}
m([x]) &= [\,] && (10)\\
m([x,y]) &= [[x+y]] && (11)\\
m([x_0,x_1,x_2,\ldots,x_n] &= m([x_0+x_1,x_1+x_2,\cdots,x_{n-1}+x_n]) + && (12)\\
&\quad [[x_0+x_1,x_1+x_2,\cdots,x_{n-1}+x_n]] && (13)
\end{aligned}
$$

The code:

```
x = [[1,2,3,4,5],[1],[3,4],[5,10,22],[1,2,3,4,5,6]]
for i in x:
    print(m(i))
```

produces

```
[[48], [20, 28], [8, 12, 16], [3, 5, 7, 9]]
[]
[[7]]
[[47], [15, 32]]
[[112], [48, 64], [20, 28, 36], [8, 12, 16, 20], [3, 5, 7, 9, 11]]
```

Programming Problem 2: Pyramids of Egypt

- Complete the function m
- The function must be recursive
- Include a docstring

3: Recursion finding Factors

Here is a recursion for d for $x = 0, 1, 2, \ldots$ and $y = 1, 2, 3, \ldots$

$$d(1, y) = 1 \tag{14}$$

$$d(0, y) = y \tag{15}$$

$$d(x, y) = d(b \ \% \ a, a) \tag{16}$$

where $a = \min(x, y), b = \max(x, y)$ Here is a recursion e that relies on d:

$$e(x, y) = \begin{cases} xy & d(x, y) = 1 \\ ze(x//z, y//z) & d(x, y) = z \end{cases} \tag{17}$$

where run

```
data = [[15,25],[6,7],[1,1],[1,2],[0,4],[210,2310]]

for i in data:
    print(e(*i))
```

produces

```
75
42
1
2
0
2310
```

Programming Problem 3: Recursion finding Factors

- Complete the functions d_1() and e_1() using recursion
- Function e_1() must use d_1()
- You will need to use Python's integer division // and modulus %
- Include a docstring for each function

Problem 4: Coins

Sometimes problems are a lot simpler than you initially imagine. Write a function that takes an amount tot in dollars and returns a list:

$$[q, d, n, p]$$

where q is the number of quarters, d is the number of dimes, n is the number of nickels, and p is the number of pennies and

```
tot = q*.25 + d*.10 + n*.05 + p*.01
q+d+n+p is a minimum
```

For example, $2.24 = [8, 2, 0, 4] = [.25 \times 8 + .10 \times 2 + .05 \times 0 + .01 \times 4]$. Thus, your list should contain the fewest total coins possible that equals the dollar amount. You must only return non-negative integers!

Programming Problem 4: Coins

- Complete the function
- You must return non-negative integers.
- Include a docstring

Problem 5: Root Finding with Newton

Root finding is a ubiquitous need. A root is a value for which a given function equals zero. In the simplest case, for a function $f : \mathbb{R} \to \mathbb{R}$,

$$f(x*) = 0 \quad x* \in [a, b] \tag{18}$$

$x*$ is called a root. The Newton-Raphson is an algorithm to find roots that uses a function $f(n)$ and its derivative. The following algorithm finds successively better approximations to a root:

$$x_0 = estimate \tag{19}$$

$$x_{n+1} = x_n - \frac{f(x_n)}{(Df)(x_n)} \tag{20}$$

See Fig. 1 for a visualization. One weakness of this technique is that you cannot begin with an estimate that is less than the root. For the function $f(x) = x^2 - 2$ (which is simply $\sqrt{2}$, if you start with 1, you'll get an erroneous answer. On the other hand, if you begin with 100, you'll find the root. In this homework, all the estimates are greater than the root. We can employ the technique of convergence using a threshold τ to stop at an acceptable precision:

$$x_0 = estimate \tag{21}$$

$$x_{n+1} = \begin{cases} x_n - \frac{f(x_n)}{(Df)(x_n)} & f(x_n) > \tau \\ x_n & otherwise \end{cases} \tag{22}$$

The derivative makes a new function from f (in the starter code, the derivative calculation has to be implemented in the **D(f)** function).

$$(Df) = \lambda x : \frac{f(x+h) - f(x-h)}{2h} \quad h \text{ is tiny, positive} \tag{23}$$

$$\tag{24}$$

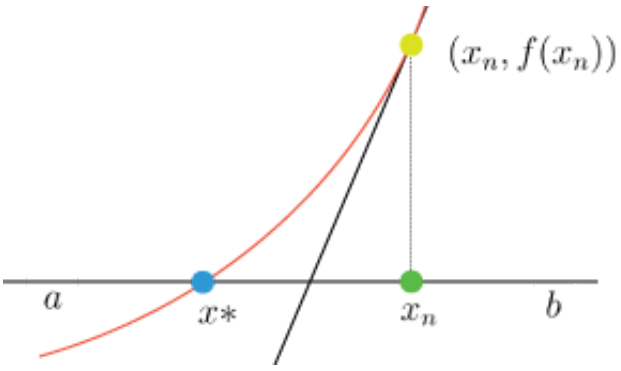

Figure 1: The root is $x*$. Our approximation x_n moves toward the root as long as we're larger than our threshold. Observe in the graphic that $f(b)$ is positive and $f(a)$ is negative insuring that there exists a root $x*$, $f(x*)$.

The following listing shows Newton-Raphson for f using $h = .00001$ and $\tau = 0.0001$. that will be used in the homework.

$$f(x) = x^6 - x - 1 \tag{25}$$

$$(Df) = \lambda x : \frac{f(x + .00001) - f(x - .00001)}{2(.00001)} \tag{26}$$

In Table 1 we start the the algorithm with an initial guess of 1.5 The next value to the right shows this is $f(1.5) \approx 8.890625$ which is significantly greater than our threshold. The algorithm stops on the last line (output colored in blue) which is very close to zero.

x	$f(x)$	$(Df)(x)$
1.5	8.890625	44.56250000703931
1.3004908836219782	2.537264144112494	21.319672162123382
1.1814804164432096	0.5384585848412315	12.812868825062827
1.1394555902943637	0.0492352512051355	10.524929241917391
1.1347776252388793	0.0005503238766089158	10.290289315828538
1.134724145316234	7.113601707686712e-08	

Table 1: Using Newton-Raphson to determine a root. The last row shows where x is sufficiently close to the root to stop.

The code:

```
1  def D(f):
2      pass
3
4  def newton(f,x,tau):
5      pass
6
7  p1 = [[lambda x:x**2 - 2, 100],[lambda x:x**6-x-1,1.5],
8        [lambda x:x**3-(100*(x**2))-x + 100,0]]
9  tau = 0.0001
10
11 for f,g in p1:
12     root = newton(f,g,tau)
```

yields

```
1  1.4142156862745259 6.007304928168367e-06
2  1.134724145316234 7.113601707686712e-08
3  100.0 0.0
```

Programming Problem 5: Root Finding with Newton

- Note that **D(f) must** return a lambda function, not a numeric value. While implementing **D(f)** use equation-23, and h=0.00001.
- Implement newton(f,x,tau) using the equation on line (22). You should be using D(f) in your implementation.
- You are free to implement the function recursively or with a while-loop only.
- Include a docstring.

6: Optimus Prime

Consider some $p \in \mathbb{Z}$ where $p > 1$. If the only factors are $p, 1$, then p is called a *prime number*. If p is not a prime number, then it is called a *composite number*. Remember, a factor is a number that integer divides another. Prime numbers are important to cryptography and have fascinated people for a long time. Let's leverage modulo to find a small list of primes. Using modulo, implement a Boolean function prime(p) that returns true if p is prime for any positive integer p > 1.

Here is code:

```
ps = []
for p in range(2,100):
    if prime(p):
        ps.append(p)

print(ps)
```

and its behavior:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, ↩
    67, 71, 73, 79, 83, 89, 97]
```

Programming Problem 6: Optimus Prime

- Complete the prime function.
- Given a number p, it's prime if no other number $p-1, p-2, \ldots, 2$ is a factor. Do a little mathematics and reduce the size of the numbers you have to check.
- You are only allowed to use modulo and integer division if at all.
- Hint: Using *math.ceil* might be helpful.
- Include a docstring.