6.101 Exam 1

Fall 2023

Name: **Answers**

Kerberos/Athena Username:

4 questions

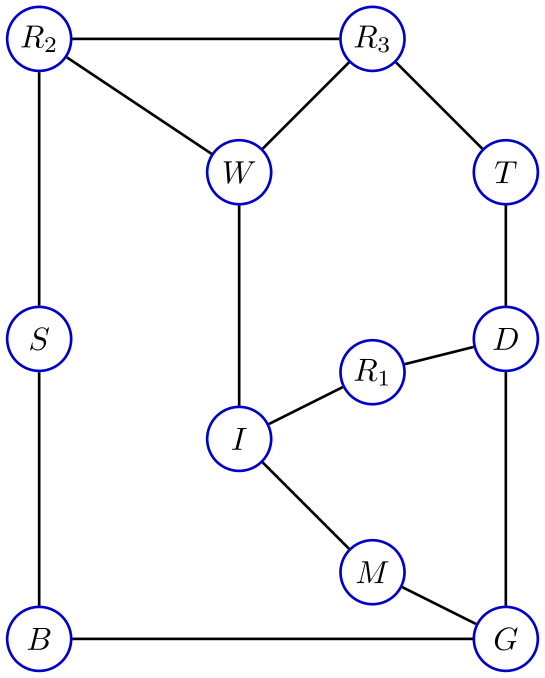
1 hour and 50 minutes

- Please **WAIT** until we tell you to begin.
- Write your name and kerberos **ONLY** on the front page.
- This exam is closed-book, and you may **NOT** use any electronic devices (including computers, calculators, phones, etc.).
- If you have questions, please **come to us at the front** to ask them.
- Enter all answers in the boxes provided. Work on other pages with QR codes may be taken into account when assigning partial credit. **Please do not write on the QR codes.**
- If you finish the exam more than 10 minutes before the end time, please quietly bring your exam to us at the front of the room. If you finish within 10 minutes of the end time, please remain seated so as not to disturb those who are still finishing their exams.
- You may not discuss the details of the exam with anyone other than course staff until final exam grades have been assigned and released.

1 Amusement Park

You are at an amusement park with your friends. On your way in to the park, you were given a map showing the locations of the various attractions and the walkways connecting them. The park has three **R**oller coasters, a **T**heater, a **D**ance Hall, a **W**ax Museum, a **S**lide, an **I**ce Cream Shop, a **M**ini-Golf Course, public **B**athrooms, and a **G**ift Shop.

From the map, you are able to construct the following graph, with states representing the different attractions at the park:



After having lunch at the Ice Cream Shop (I), you are overcome with a desire to go to the Gift Shop (G) to buy a plush Python to give to your 6.101 instructors. You decide to plan your route to the Gift Shop using the search algorithms discussed in 6.101 (specifically, the `find_path` function included on the last page of this exam, which you may remove). For each of the searches, assume that the neighbors function returns neighbors in alphabetical order (including subscripts, so R_1 before R_2 before R_3).

1.1 Searches

For DFS and BFS, respectively (using the `find_path` code from page 21), specify the sequence of states that are visited (added to the `visited` set) **in the order in which they are added** during that search. Also specify the final path found from I to G, as a sequence of states.

DFS States Visited: $I, M, R_1, W, R_2, R_3, T, D$

DFS Path found: I, W, R_3, T, D, G

BFS States Visited: I, M, R_1, W

BFS Path found: I, M, G

1.2 Celebrity

Your friend decides that, rather than going straight to the gift shop, you should stop by the wax museum first so that they can take a selfie next to the Justin Bieber sculpture. However, you notice that the path returned by your search algorithms might not go by the wax museum. As such, you would like to modify the state used in your search so that `find_path` always returns a sequence of states representing a path that includes the wax museum (W), regardless of the specific layout of the park, the starting state, or the goal state. Consider the following possible state representations:

(a) A string representing the current location

(b) The entire path so far (a tuple of locations), including the current location

(c) A boolean indicating whether the path contains ”W”

(d) The current location, and a boolean indicating whether the path contains ”W”

(e) The entire path so far, and a boolean indicating whether the path contains ”W”

(f) Two booleans: one indicating whether the path contains ”W”, and one indicating whether the current location is the goal

(g) A tuple containing a boolean for each location, representing whether that location is in the path

Which of these state representations would allow us to solve the problem, given enough time?
Enter some combination of the letters a-g in the box below:

b, d, e

As the size of the theme park grows, which of the options that allow us to solve the problem will make effective use of the `visited` set to find the goal efficiently?

d

2 Counting Frames

For each of the short programs below, what will be the value of the `result` variable after the program has run? Also specify how many total frames would be created when running that program (not including the global frame). If the code would produce an error, write ERROR in both boxes.

2.1 Program 1

```
def double(x):
    return double(x * x)

first = double

def double(y):
    return y + y

result = first(3)
```

Value of `result`:	18	Number of Frames:	2

2.2 Program 2

```
def apply_to_each(func):
    def _helper(inp):
        return [func(elt) for elt in inp]
    return _helper

test_input = [[9, 8], [7, 6], [5, 4]]
fn = apply_to_each(lambda x: x[1])
result = fn(test_input)
```

Value of `result`:	`[8, 6, 4]`	Number of Frames:	5

2.3 Program 3

```
funcs = {}
for person in ('Rob', 'Max', 'Hope', 'Karen', 'Adam'):
    funcs[person] = lambda phrase: (person + " says: " + phrase)

m = funcs['Max']
result = m('Eat your veggies!')
```

Value of `result`: "Adam says: Eat your veggies!" Number of Frames: 1

2.4 Program 4

```
def f1(x):
    return x + "1"
def f2(x):
    return x + "2"

def func_cascade(functions):
    def fc(x):
        for func in functions:
            x = func(x)
        return x
    return fc

func_list = [f1, f2]
fc = func_cascade(func_list)
def f3(x):
    return x + "3"
func_list.append(f3)
f1 = f2
result = fc("0")
```

Value of `result`: "0123" Number of Frames: 5

3 Polynomials

We will start this problem by implementing two functions to perform operations on polynomials: `poly_add(p1, p2)` and `poly_mul(p1, p2)`. For both functions, the inputs `p1` and `p2` each represent polynomials, but we are not told the details of the actual representation of `p1` and `p2`; rather, we are given descriptions of various helper functions that operate on polynomials (this list is also reproduced on page 19 of the exam, which you may remove):

- `zero_poly()` returns a new polynomial representing 0.
- `get_order(poly)` returns the order of the polynomial `poly`. For example, if the input represented $8 + 7x + 4x^3$, this function would return `3`. Assume that the order of the zero polynomial is -1.
- `get_coeff(poly, i)` returns the coefficient associated with the x^i term in the given polynomial. For example, if we called `get_coeff(p, 3)` where `p` represented $8 + 7x + 4x^3$, this function would return `4`; and calling `get_coeff(p, 2)` would return 0.
- `set_coeff(poly, i, val)` *mutates* the given polynomial such that the coefficient associated with x^i is replaced with `val`. For example, calling `set_coeff(p, 2, 9)` where `p` represented $8 + 7x + 4x^3$ would result in the input polynomial being mutated to represent $8 + 7x + 9x^2 + 4x^3$.
- `shifted(poly, n)` returns a *new* polynomial representing the given `poly` multiplied by x^n. For example, calling `shifted(p, 4)` where `p` represented $8+7x+4x^3$ would result in a new polynomial $8x^4+7x^5+x^7$.
- `scaled(poly, n)` returns a *new* polynomial representing the given `poly` with call coefficients scaled by `n`. For example, calling `scaled(p, 4)` where `p` represented $8 + 7x + 4x^3$ would result in a new polynomial $32 + 28x + 16x^3$.

To start, implement `poly_add` below. This function should take two polynomials as input, and it should return a new polynomial representing the sum of the two input polynomials, without mutating either of the inputs. For example, if `p1` represents $3x + 7$ and `p2` represents $2x^2 + 9x$, then `poly_add(p1, p2)` should return a new polynomial representing $2x^2 + 12x + 7$. You should not make any assumptions about the exact representations of `p1` or `p2`, but you may assume working implementations of all of the helper functions described above.

```
def poly_add(p1, p2):
    out = zero_poly()
    for power in range(max(get_order(p1), get_order(p2))+1):
        set_coeff(out, power, get_coeff(p1, power) + get_coeff(p2, power))
    return out
```

Next, we'll implement poly_mul. This function should take two polynomials as input, and it should return a new polynomial representing the *product* of the two input polynomials, without mutating either of the inputs. For example, if p1 represents $3x + 7$ and p2 represents $2x^2 + 9x$, then poly_mul(p1, p2) should return a new polynomial representing $6x^3 + 41x^2 + 63x$. Write your code for poly_mul in the box below. You may assume working implementations of all of the helper functions from the previous page, as well as a working implementation of poly_add, but you should not define any additional helper functions here. As before, your code should not make assumptions about the exact representations of p1 or p2.

```
def poly_mul(p1, p2):
    out = zero_poly()
    for ix1 in range(get_order(p1)+1):
        new = shifted(scaled(p2, get_coeff(p1, ix1)), ix1)
        out = poly_add(out, new)
    return out
```

As we have seen throughout 6.101, the choice of representation is an important one! Careful choice of internal representation can often allow us to write more concise, and clearer code; and this problem is no exception! While your code for poly_add and poly_mul should work for *any* internal representation, we'll now try to fill in the details.

Here the choice of internal representation is up to you. You are welcome to use any combination of Python int, float, str, bool, list, tuple, set, frozenset, and/or dict objects; but you are not welcome to use classes (which have not yet been covered in 6.101). Your grade for this problem will depend on the correctness of your implementation. Your code will not be graded for efficiency in terms of time or memory, but note that some representations may be harder to implement correctly.

In the box below, briefly describe your choice of representation:

There are many possibilities here, but these solutions will use a dictionary mapping powers to their associated coefficients. For example, $5x^4 + 6x^3 + 27$ will be represented as {4: 5, 3: 6, 0: 27}.

Then, in the box below and the box on the facing page, implement the zero_poly, get_order, get_coeff, set_coeff, shifted, and scaled helper functions using that representation.

```
# your polynomial helper functions here
def zero_poly():
    return {}

def get_order(poly):
    return max((k for k in poly if poly[k] != 0), default=-1) # make sure to ignore 0s!

def get_coeff(poly, i):
    return poly.get(i, 0)

def set_coeff(poly, i, val):
    poly[i] = val

def shifted(poly, n):
    return {k+n: v for k, v in poly.items()}

def scaled(poly, n):
    return {k: v*n for k,  v in poly.items()}
```

```
# your polynomial helper functions here
# or, a different representation involving lists
# here, p[i] is the coefficient associated with the x**i term
# we'll assume that trailing zeros NEVER exist in the representation
# (and we'll set up the code to make sure that's always the case)

def zero_poly():
    return []

def get_order(poly):
    # again, we need to be careful not to count explicit zeros toward
    # our order calculation; but in this case, we handle that in set_coeff
    # instead
    return len(poly) - 1

def get_coeff(poly, i):
    return poly[i] if 0 <= i < len(poly) else 0

def set_coeff(poly, i, val):
    while get_order(poly) < i:
        poly.append(0)
    poly[i] = val
    while poly and poly[-1] == 0:
        poly.pop()

def shifted(poly, n):
    return [0]*n + poly

def scaled(poly, n):
    return [coeff * n for coeff in poly]
```

Worksheet (intentionally blank)

4 Farmer, Goat, Wolf, Cabbage

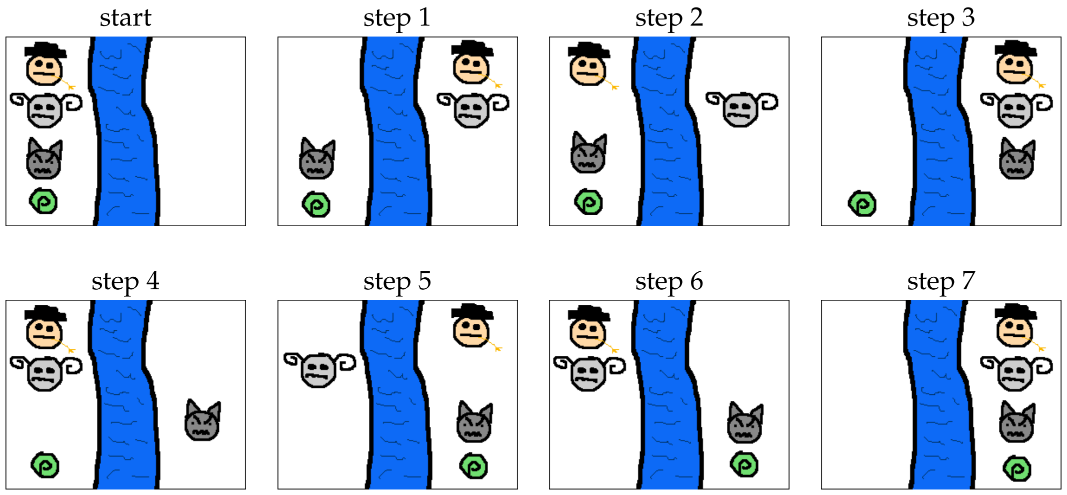
One version of a standard puzzle called the *Farmer, Goat, Wolf, Cabbage* puzzle goes as follows:

- A farmer has a goat, a wolf, and a cabbage.
- The farmer and all the animals (and the cabbage!) come to a river, and they all need to get to the the bank on the right side of the river.
- The farmer can row a boat across the river with at most one other passenger (it's a big head of cabbage). The boat cannot cross the river without the farmer on board.
- If the farmer leaves the goat and cabbage on the same side of the river, when the farmer is not present the goat will eat the cabbage.
- Similarly, if the farmer leaves the goat and the wolf on the same side of the river, when the farmer is not present...well, that's not allowed, either.

So, the problem is to find a sequence of actions that go from an arbitrary initial state, to the final state when they are all on the right bank. On any given step, the farmer can travel alone across the river, or can take one of the goat, wolf, or cabbage with him.

For example, one solution to the puzzle, where everyone starts on the left bank (start) is as follows:

- Take the goat across the river (step 1), then return alone (step 2).
- Take the wolf across the river (step 3), then return with the goat (step 4).
- Take the cabbage across the river (step 5), then return alone (step 6).
- Take the goat across (step 7), and the puzzle has been solved!



For this problem, we will use the `find_path` code from the 6.101 readings (reproduced on page 21 for convenience) to solve this puzzle. Your code should actually make use of `find_path` to solve the problem, rather than hard-coding answers to the puzzle you've solved by hand.

On the following four pages, briefly describe your chosen state representation and fill in the necessary code so that `result` (as defined at the bottom of page 15) will be a list of actions necessary to cross successfully without any of the farmer's pets or vegetables eating each other. `'g'` means take the goat across, `'w'` means take the wolf, `'c'` means take the cabbage, and `None` means go across alone; so, for example, the solution above corresponds to `['g', None, 'w', 'g', 'c', None, 'g']`.

Firstly, please briefly describe your chosen representation for the states in this search:

As in the last problem, there are many different possible representations here. The solutions below as a tuple of Boolean values, each one representing a character and indicating whether they are on the left bank (`False`) or the right bank (`True`). For example, `(False, True, False, False)` corresponds to the goat being on the right bank and everyone else being on the left bank.

Now fill in the following functions (noting how `result` is computed using these functions on page 15):

```
def fgwc_initial_state(farmer, goat, wolf, cabbage):
    # each of the arguments is a string, 'right' or 'left', describing one character's
    # initial position.  you may assume that the given configuration is valid,
    # in the sense that none of the animals or vegetables would be eaten in the
    # initial configuration.
```

```
    return tuple(loc == 'right' for loc in (farmer, goat, wolf, cabbage))
```

```
def fgwc_goal_test(state):
```

```
    return all(state)
```

```
# space for helper functions beyond those we've specified:
# (leave blank if you don't want/need any additional helpers)
```

```
def is_safe(state):
    f, g, w, c = state
    if g == c != f:
        return False
    if g == w != f:
        return False
    return True
```

```
def fgwc_successors(state):
```

```
f, g, w, c = state
out = []
for ix in range(len(state)):
    if f != state[ix]:
        continue  # farmer needs to be on the same side to move
    s = list(state)
    s[0] = not state[0]
    s[ix] = not state[ix]
    out.append(s)
return [tuple(i) for i in out if is_safe(i)]
```

```
def process_fgwc_path(path):
    if path is None:
        return None
    out = []
    for ix in range(len(path) - 1):
        out.append(action_between(path[ix], path[ix+1]))
    return out

def action_between(state1, state2):
```

```
    moved = [None, 'g', 'w', 'c']
    changes = [ix for ix, (o,n) in enumerate(zip(state1, state2)) if o != n]
    return moved[max(changes)]
```

```
initial = fgwc_initial_state('left', 'left', 'left', 'left')
result = process_fgwc_path(find_path(fgwc_successors, initial, fgwc_goal_test))
```

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Worksheet (intentionally blank)

Helper Functions for Polynomials

- `zero_poly()` returns a new polynomial representing 0.
- `get_order(poly)` returns the order of the polynomial `poly`. For example, if the input represented $8 + 7x + 4x^3$, this function would return `3`. Assume that the order of the zero polynomial is -1.
- `get_coeff(poly, i)` returns the coefficient associated with the $x^{\texttt{i}}$ term in the given polynomial. For example, if we called `get_coeff(p, 3)` where `p` represented $8 + 7x + 4x^3$, this function would return `4`; and calling `get_coeff(p, 2)` would return 0.
- `set_coeff(poly, i, val)` *mutates* the given polynomial such that the coefficient associated with $x^{\texttt{i}}$ is replaced with `val`. For example, calling `set_coeff(p, 2, 9)` where `p` represented $8 + 7x + 4x^3$ would result in the input polynomial being mutated to represent $8 + 7x + 9x^2 + 4x^3$.
- `shifted(poly, n)` returns a *new* polynomial representing the given `poly` multiplied by $x^{\texttt{n}}$. For example, calling `shifted(p, 4)` where `p` represented $8+7x+4x^3$ would result in a new polynomial $8x^4+7x^5+x^7$.

Worksheet (intentionally blank)

Path Finding Code

```
def find_path(neighbors_function, start_state, goal_test, bfs=True):
    """
    Return a path through a graph from a given starting state to any state that
    satisfies a given goal condition (or None if no such path exists).

    Parameters:
      * neighbors_function(state) is a function which returns a list of legal
        neighbor states
      * start_state is the starting state for the search
      * goal_test(state) is a function which returns True if the given state is
        a goal state for the search, and False otherwise.
      * bfs is a boolean (default True) that indicates whether we should run a
        bfs or dfs

    Returns:
        A path from start_state to a state satisfying goal_test(state) as a
        tuple of states, or None if no path exists.

    Note the state representation must be hashable in order for this function
    to work.
    """
    if goal_test(start_state):
        return (start_state,)

    agenda = [(start_state,)]
    visited = {start_state}

    while agenda:
        this_path = agenda.pop(0 if bfs else -1)
        terminal_state = this_path[-1]

        for neighbor_state in neighbors_function(terminal_state):
            if neighbor_state not in visited:
                new_path = this_path + (neighbor_state,)

                if goal_test(neighbor_state):
                    return new_path

                agenda.append(new_path)
                visited.add(neighbor_state)
```

Worksheet (intentionally blank)