CS 61A Structure and Interpretation of Computer Programs
Spring 2024

Midterm 2 Solutions

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

◯ You must choose either this option

◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

☐ You could select this choice.

☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

You can complete and submit these questions before the exam starts.

(a) What is your full name?

(b) What is your student ID number?

(c) What is your @berkeley.edu email address?

(d) Sign (or type) your name to confirm that all work on this exam will be your own. The penalty for academic misconduct on an exam is an F in the course.

1. (7.0 points) What Would Python Display?

Assume the following code has been executed. The `Link` class appears on the midterm 2 study guide (page 2, left side).

```
def shake(it):
    if it is not Link.empty and it.rest is not Link.empty:
        if it.first + 1 < it.rest.first:
            it.rest = Link(it.rest.first-1, it.rest)
            shake(it)
        else:
            shake(it.rest)
it = Link(2, Link(5, Link(7)))
off = Link(1, it.rest)
shake(it)

def cruel(summer):
    while summer is not Link.empty:
        yield summer.first
        summer = summer.rest
        if summer is not Link.empty:
            summer = summer.rest
summer = Link(1, Link(2, Link(3, Link(4))))
```

Write the output printed for each expression below or *Error* if an error occurs.

(a) (2.0 pt) `print(it)`

- ○ `<2 5 7>`
- ○ `<2 4 5 7>`
- ○ `<2 4 5 6 7>`
- ○ `<2 3 4 5 7>`
- ○ `<2 4 3 5 7>`
- [✓ `<2 3 4 5 6 7>`]
- ○ `<2 4 3 5 6 7>`

(b) (2.0 pt) `print(off)`

```
<1 5 6 7>
```

(c) (2.0 pt) `print([x*x for x in cruel(summer)])`

```
[1, 9]
```

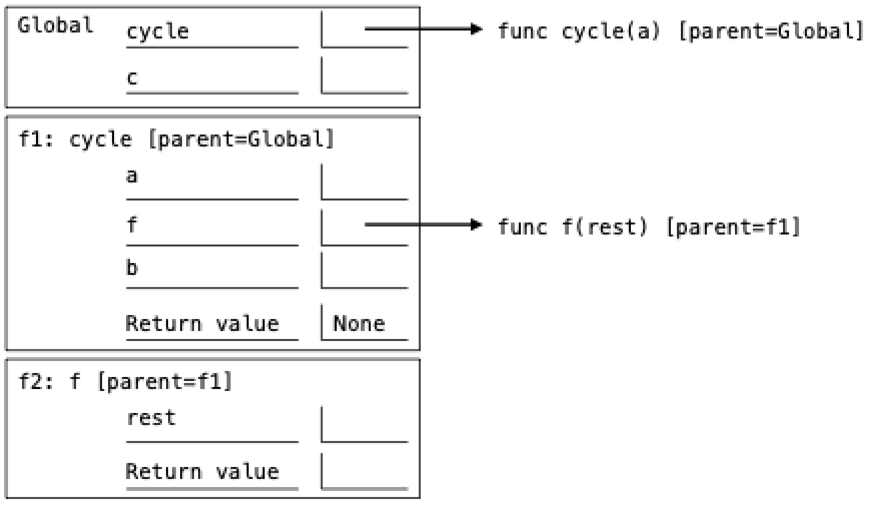
(d) **(1.0 pt)** What is the order of growth of the time it takes to evaluate `shake(Link(1, Link(n)))` in terms of `n`?

- ○ exponential
- ○ quadratic
- [✓ linear]
- ○ constant

2. (6.0 points) Spin Cycle

Complete the environment diagram below and then answer the questions that follow. Do not add frames for calls to built-in functions (such as `print`).

```
 1: def cycle(a):
 2:     def f(rest):
 3:         rest.append(a.pop())
 4:         return [rest.append(a)]
 5:     b = []
 6:     a = f(b)
 7:     b = b + [5]
 8:     print(a)
 9:     print(b)
10:
11: c = [2, 3, 4]
12: cycle(c)
13: print(c)
```



(a) (2.0 pt) What would be printed by the expression `print(a)` on line 8?

`[None]` **https://i.postimg.cc/Zn1D5MZM/cycle-sol.png**

(b) (2.0 pt) What would be printed by the expression `print(b)` on line 9?

`[4, [2, 3], 5]`

(c) (2.0 pt) What would be printed by the expression `print(c)` on line 13?

`[2, 3]`

3. (8.0 points) Fearless

If you sing lyrics into a mic, every connected speaker repeats them.

A `Mic` instance has a dictionary `speakers` containing `Speaker` instances as values, each with its location (`str`) as its key. Its `sing` method takes a string `lyrics` and invokes the `repeat` method of each `Speaker` instance connected to it.

A `Speaker` takes a `transform` function that takes and returns a string. To `connect` a `Speaker` instance to `m` (`Mic`) in a `location` (`str`), add that instance to the `speakers` dictionary of `m` in that `location`. To `repeat` a signal `s` (`str`), return the result of calling the speaker's `transform` function on `s`.

Every `Mic` starts connected to a `Speaker` in the `Front` location that repeats the exact same signal it receives.

Implement the `Mic` and `Speaker` classes to match the doctests. The `str.lower` and `str.upper` functions return lowercased and uppercased versions of a string, respectively.

```
class Mic:
    """A microphone connected to speakers.

    >>> m = Mic()  # Front is connected automatically
    >>> m.sing('Is this thing on?')
    Front - Is this thing on?
    >>> Speaker(str.lower).connect(m, 'Left Side')
    >>> Speaker(str.upper).connect(m, 'Right Side')
    >>> m.sing("You belong with me.")
    Front - You belong with me.
    Left Side - you belong with me.
    Right Side - YOU BELONG WITH ME.
    """
    def __init__(self):
        self.speakers = _______
                          (a)

    def sing(self, lyrics):
        for k in self.speakers.keys():  # iterate over the keys of a dictionary
            print(k, '-', _______.repeat(lyrics))
                            (b)

class Speaker:
    def __init__(self, transform):
        self.transform = transform

    def connect(self, m, location):
        _______
          (c)

    def repeat(self, s):
        return _______
                  (d)
```

(a) (3.0 pt)

```
{'Front': Speaker(lambda x: x)}
```

(b) (2.0 pt)

- ○ `self`
- ○ `self.speaker`
- ○ `self.speakers`
- [✓ `self.speakers[k]`]
- ○ `self.speakers[k].self`
- ○ `speaker`
- ○ `speakers`
- ○ `speakers[k]`
- ○ `speakers[k].self`

(c) (2.0 pt)

```
m.speakers[location] = self
```

(d) (1.0 pt)

- ○ `s`
- ○ `lyrics`
- ○ `transform`
- ○ `transform(s)`
- ○ `transform(lyrics)`
- ○ `self.transform`
- [✓ `self.transform(s)`]
- ○ `self.transform(lyrics)`

4. (29.0 points) Who's counting?

Definition. A *strip* is a list of integers in which each integer is one more than the last. For example, `[3, 4, 5, 6]` is a strip. Empty and one-element lists are strips.

(a) (4.0 points)

Implement `is_strip`, which takes a list of integers `s` and returns whether it is a strip.

```
def is_strip(s):
    """Return whether list s is a strip.

    >>> is_strip([3, 4, 5, 6])
    True
    >>> is_strip([3, 3, 3])        # 3 after 3
    False
    >>> is_strip([3, 4, 5, 4, 6])  # 4 after 5
    False
    >>> is_strip([3, 4, 5, 5, 6])  # 5 after 5
    False
    >>> is_strip([3, 4, 5, 6, 8])  # 8 after 6
    False
    >>> is_strip([5])
    True
    >>> is_strip([])
    True
    """
    assert type(s) == list
    return _______ or s == list ( range ( _______ , _______ ))
              (a)                            (b)       (c)
```

i. (1.0 pt) Fill in blank (a).

- [✓ `len(s) == 0`]
- ○ `s[0] + 1 == s[1]`
- ○ `s[0] + 1 in s`
- ○ `s[0] + 1 in s[1:]`

ii. (1.0 pt) Fill in blank (b).

```
s[0]
```

iii. (2.0 pt) Fill in blank (c).

- ○ `s`
- ○ `s[0]`
- ○ `s[1]`
- ○ `s[0] + 1`
- [✓ `s[0] + len(s)`]
- ○ `s[len(s) - 1]`
- ○ `s[1] - s[0]`

(b) (7.0 points)

Implement `drip`, which takes two non-empty lists of integers `s` and `t`. It returns `True` if there is a strip containing all and only the elements of `s` and `t` **starting with** `s[0]` in which the elements of `s` appear in order and the elements of `t` appear in order. It returns `False` otherwise.

```
def drip(s, t):
    """Return whether there is a strip made out of interleaving s and t.

    >>> drip([1, 3, 5], [2, 4, 6])    # 1 2 3 4 5 6
    True
    >>> drip([1, 4, 5], [2, 3, 6])    # 1 2 3 4 5 6
    True
    >>> drip([1, 2, 3], [4, 5, 6])    # 1 2 3 4 5 6
    True
    >>> drip([2, 4, 5], [1, 3, 6])    # No strip starting with 2 can contain 1
    False
    >>> drip([1, 2, 4, 5], [1, 3, 6]) # No strip can contain 1 and 1
    False
    >>> drip([1, 4, 5], [2, 3, 7])    # No strip can contain 5 and 7 but no 6
    False
    >>> drip([1, 5, 4], [2, 3, 6])    # No strip can contain 5 before 4
    False
    >>> drip([2], [3, 4, 5])          # 2 3 4 5
    True
    >>> drip([1], [2, 3, 5])          # No strip can contain 3 and 5 but no 4
    False
    """
    while s and t:
        if s[0] + 1 == t[0]:
            s, t = _______  # The next element of the strip is in t
                     (a)
        elif _______:
                (b)
            s = s[1:]       # The next element of the strip is in s
        else:
            return _______
                     (c)
    return _______
             (d)
```

i. (1.0 pt) Fill in blank (a).

○ `t, s`

● `t, s[1:]`

○ `t[1:], s`

ii. (3.0 pt) Fill in blank (b).

`len(s) > 1 and s[0] + 1 == s[1]`

iii. (1.0 pt) Fill in blank (c).

- [✓ `False`]
- ○ `s[0] != t[0]`
- ○ `is_strip(s) or is_strip(t)`
- ○ `not (is_strip(s) or is_strip(t))`
- ○ `is_strip(s) and is_strip(t)`
- ○ `not (is_strip(s) and is_strip(t))`

iv. (2.0 pt) Fill in blank (d).

- ○ `True`
- ○ `is_strip(s)`
- ○ `is_strip(t)`
- ○ `is_strip(s) or is_strip(t)`
- ○ `not (is_strip(s) or is_strip(t))`
- [✓ `is_strip(s) and is_strip(t)`]
- ○ `not (is_strip(s) and is_strip(t))`

(c) (6.0 points)

Implement `longest`, which takes a list of integers `s`. It returns the longest strip whose elements appear in s in order. If two such strips exist, return the one that starts earlier in `s`. **Hint:** `s[-1]` is the last element in list `s`.

```
def longest(s):
    """Return the longest strip whose elements appear in s in order.
    >>> longest([4, 2, 3, 5, 6, 4, 6, 5])  # 2 3 4 5 is the longest strip in s
    [2, 3, 4, 5]
    >>> longest([4, 2, 3, 5, 6, 4])  # 4 5 6 is as long as 2 3 4 and is earlier
    [4, 5, 6]
    >>> longest([2, 4, 6])
    [2]
    """
    if len(s) == 0:
        return []
    return max([longest_with_s0(s), _______ ], key= _______ )
                                      (a)              (b)
def longest_with_s0(s):
    """Return the longest strip in s that starts with s[0]."""
    result = _______
                (c)
    for k in s[1:]:
        if _______:
             (d)
            result.append(k)
    return result
```

i. (2.0 pt) Fill in blank (a).

```
longest(s[1:])
```

ii. (1.0 pt) Fill in blank (b).

[✓ `len`]

`len(s)`

`s[0]`

`lambda i: s[i]`

iii. (1.0 pt) Fill in blank (c).

`[]`

`[s]`

`s[0]`

[✓ `s[:1]`]

iv. (2.0 pt) Fill in blank (d).

```
k == result[-1] + 1
```

(d) (5.0 points)

Implement `has_strip`, which takes a `Tree` of integers `t`. It returns whether there is a strip containing the labels along some path from the root to a leaf of `t`. The `Tree` class is on the midterm 2 study guide (page 2 left side).

```
def has_strip(t):
    """Return whether the elements of some root-to-leaf path form a strip.

    >>> has_strip(Tree(1, [Tree(3, [Tree(4)]), Tree(2, [Tree(2), Tree(3)])])) # 1, 2, 3
    True
    >>> has_strip(Tree(1, [Tree(3, [Tree(4)]), Tree(2, [Tree(2, [Tree(3)])])]))
    False
    >>> has_strip(Tree(1, [Tree(3, [Tree(4)]), Tree(2, [Tree(2)])]))
    False
    >>> has_strip(Tree(1, [Tree(2, [Tree(4)]), Tree(3, [Tree(3)])]))
    False
    """
    if t.is_leaf():
        return _______
                 (a)
    for b in t.branches:
        if _______
             (b)
            return True
    return _______
             (c)
```

i. (1.0 pt) Fill in blank (a).

- [✓ `True`]
- `False`
- `t`
- `t.label`
- `t.label == t + 1`
- `t + 1 == t.label`

ii. (3.0 pt) Fill in blank (b).

```
b.label == t.label + 1 and has_strip(b)
```

iii. (1.0 pt) Fill in blank (c).

- `True`
- [✓ `False`]
- `t`
- `t.label`
- `t.label == t + 1`
- `t + 1 == t.label`

(e) (7.0 points)

Implement `strips`, a generator function that takes a `Tree` of integers `t`. It yields all strips that contain the labels along a path from the root to a leaf of `t`.

```
def strips(t):
    """Yield the paths from the root to a leaf of Tree t that form strips.

    >>> list(strips(Tree(1, [Tree(3, [Tree(4)]), Tree(2, [Tree(2), Tree(3)])])))
    [[1, 2, 3]]
    >>> list(strips(Tree(1, [Tree(2), Tree(2, [Tree(2), Tree(3, [Tree(4)]), Tree(3)])])))
    [[1, 2], [1, 2, 3, 4], [1, 2, 3]]
    """
    if t.is_leaf():
        yield _______
                (a)

    for b in t.branches:
        if _______:
             (b)

            for s in _______:
                       (c)

                yield _______
                        (d)
```

i. (2.0 pt) Fill in blank (a).

○ `None`

○ `t`

○ `t.label`

○ `[t]`

[✓ `[t.label]`]

ii. (1.0 pt) Fill in blank (b).

○ `t.label == b.label + 1`

[✓ `b.label == t.label + 1`]

○ `t.label == b.label + 1 and is_strip(b)`

○ `b.label == t.label + 1 and is_strip(b)`

iii. (2.0 pt) Fill in blank (c).

○ `strips(t)`

[✓ `strips(b)`]

○ `t.branches`

○ `b.branches`

○ `range(len(t.branches))`

○ `range(len(b.branches))`

iv. (2.0 pt) Fill in blank (d).

- ○ `s`
- ○ `t`
- ○ `t + s`
- ○ `t.label + s`
- ○ `[t] + s`
- [✓ `[t.label] + s`]
- ○ `[t] + [s]`
- ○ `[t.label] + [s]`

v. (0.0 pt) This A+ question is not worth any points. It can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

Fill in the blank of `only_strips`, which takes a Tree of integers `t`. It removes all nodes that are **not** on a *strip path*. A *strip path* is a path from the root to a leaf whose labels form a strip. You may use functions implemented earlier in this question.

```
def test_only_strips(t):
    """Call only_strips(t) and then return t. Assume has_strip(t) is True.

    >>> test_only_strips(Tree(1, [Tree(2), Tree(2, [Tree(2), Tree(3, [Tree(4)]), Tree(3)])]))
    Tree(1, [Tree(2), Tree(2, [Tree(3, [Tree(4)]), Tree(3)])])
    >>> test_only_strips(Tree(5, [Tree(6), Tree(6, [Tree(7, [Tree(9)])]), Tree(7)]))
    Tree(5, [Tree(6)])
    """
    only_strips(t)
    return t

def only_strips(t):
    """Remove all nodes of Tree t that are not on a path from the root to a leaf
    whose labels form a strip. Assume has_strip(t) is True.
    """
    t.branches = [b for b in t.branches if _______]
```

```
has_strip(b) and (only_strips(b) or t.label + 1 == b.label)
```

No more questions.