CS 61A Fall 2021

Structure and Interpretation of Computer Programs

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? A regex restricts inputs to numerical responses only.

1. (7.0 points) Hawkeye

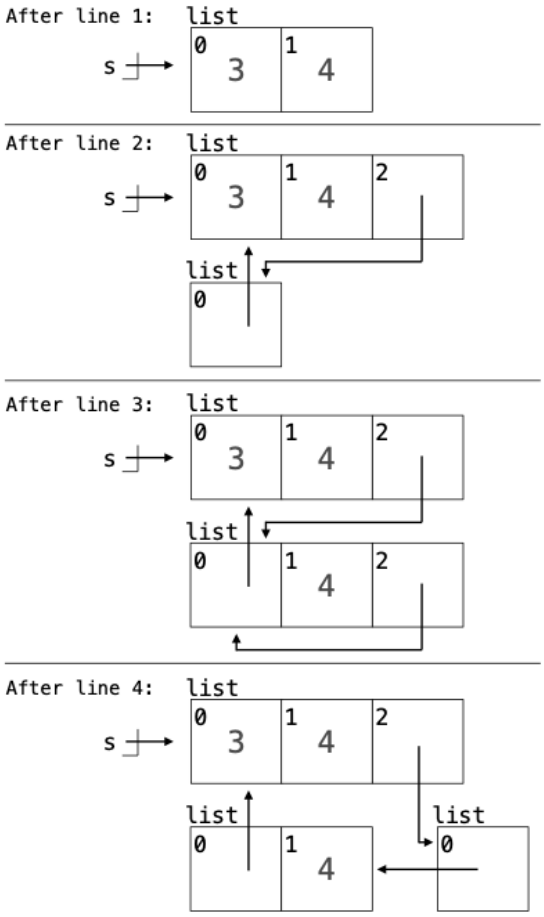
The box-and-pointer diagram for the value of `s` is shown after each line of the program below. Fill in the blanks to match the diagrams.

```
s = [3, 4]

s.append(_________)
            (a)

_________.________(_________)
   (b)       (c)       (d)

s[2] = _________
           (e)
```



(a) (2.0 pt) Which of these could fill in blank (a)? Check all that apply.

- ☐ `s`
- ☐ `s[:]`
- ☐ `s[0]`
- ■ `[s]`
- ☐ `[s[:]]`
- ☐ `[s[0]]`
- ☐ `list(s)`
- ☐ `list(s[:])`
- ☐ `list(s[0])`

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

```
s[2]
```

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

```
extend
```

(d) (1.0 pt) Which of these could fill in blank (d)? (Select one.)

- ○ `s[1]`
- ● `s[1:]`
- ○ `s[1:2]`
- ○ `[s[1]]`
- ○ `[s[1:]]`
- ○ `[s[1:2]]`
- ○ `[s[1], s]`
- ○ `[s[1]] + s`

(e) (2.0 pt) Fill in blank (e).

```
[s[2].pop()]
```

2. (11.0 points) Doctor Change

(a) (6.0 points)

Implement `change`, a function that takes an integer `n` and a list of positive integers `coins`. It returns whether there is a subset of the values in `coins` that sums to `n`. As a side effect, `change` modifies `coins`.

```
def change(n, coins):
    """Return whether a subset of coins adds up to n.

    >>> change(10, [2, 7, 1, 8, 2])  # e.g., 2 + 8
    True
    >>> change(20, [2, 7, 1, 8, 2])  # e.g., 2 + 7 + 1 + 8 + 2
    True
    >>> change(6, [2, 7, 1, 8, 2])   # Impossible; only two 2's in coins
    False
    """
    if n == 0:
        return True
    elif _________:
            (a)
        return False
    coin = coins.pop()  # remove the end of coins and name it "coin"
    return change(n, _________) _________ change(_________, _________)
                        (b)        (c)              (d)        (e)
```

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

```
n < 0 or not coins or just not coins
```

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

```
list(coins)
```
At least one of blank (b) and blank (e) must copy the `coins` list.

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

```
or
```

iv. (1.0 pt) Which of these could fill in blank (d)? (Select one.)

- (●) `n - coin`
- ( ) `n - coins[0]`
- ( ) `n - coins[1]`
- ( ) `n - coins.pop()`

v. (1.0 pt) Fill in blank (e).

```
list(coins)
```

(b) (5.0 points)

Implement `amounts`, which takes a list of positive integers `coins`. It returns a sorted list of all unique non-negative integers `n` for which `change(n, coins)` returns `True`. You **may not** call `change`.

```
def amounts(coins):
    """List all unique n such that change(n, coins) returns True (in sorted order).

    >>> amounts([2, 5, 3])
    [0, 2, 3, 5, 7, 8, 10]
    >>> amounts([2, 7, 1, 8, 2])
    [0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20]
    """
    if not coins:
        return _________
                  (a)
    coin = coins[0]
    rest = amounts(_________)
                      (b)
    return sorted(_________ + [k + coin for _________])
                    (c)                        (d)
```

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

```
[0]
```

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

```
coins[1:]
```

iii. (1.0 pt) Which of these could fill in blank (c)? (Select one.)

- (●) `rest`
- ( ) `rest[1:]`
- ( ) `amounts(rest)`
- ( ) `amounts(rest[1:])`
- ( ) `[k + coin for k in rest]`
- ( ) `[k + coin for k in rest[1:]]`

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

```
k in rest if k + coin not in rest
```

3. (13.0 points) Shang-Chi

(a) (6.0 points)

Implement the `Valet` and `Garage` classes.

A `Valet` instance has two instance attributes: their total `tips` and the `garage` where they work (a `Garage` instance).

- The `park` method takes a string `car` and parks it in the `Garage` where they work.
- The `wash` method takes a string `car` that has been parked in their garage and a number `tip`. The `Valet` who washed the car and the `Valet` who **most recently** parked the car split the tip.

The `Garage` constructor takes a list of the `Valet`s who work in that `Garage`. Assume that `park` and `wash` are only invoked on a `Valet` that already has a `Garage`.

```
class Valet:
    """A valet is tipped after they wash a car,
    or after one of their parked cars is washed.

    >>> shaun = Valet()
    >>> katy = Valet()
    >>> g = Garage([shaun, katy])
    >>> shaun.park('Benz')
    >>> katy.park('BMW')
    >>> shaun.wash('Benz', 1)   # $1.0 to Shaun
    >>> katy.wash('Benz', 2)    # $1.0 to Katy, $1.0 to Shaun

    >>> shaun.park('Rolls')
    >>> katy.park('Rolls')
    >>> katy.wash('BMW', 2)     # $2.0 to Katy
    >>> shaun.wash('Rolls', 2)  # $1.0 to Shaun, $1.0 to Katy
    >>> [shaun.tips, katy.tips]
    [3.0, 4.0]
    """
    def __init__(self):
        self.tips = 0
        self.garage = None

    def park(self, car):
        _________
           (a)

    def wash(self, car, tip):
        self.tips += tip / 2
        _________ += tip / 2
           (b)

class Garage:
    """A garage holds cars parked by the valets who work there."""
    def __init__(self, valets):
        self.cars = {}
        for valet in valets:
            _________ = _________
               (c)         (d)
```

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

```
self.garage.cars[car] = self
```

ii. (2.0 pt) Fill in blank (b).

```
self.garage.cars[car].tips
```

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

```
valet.garage
```

iv. (1.0 pt) Which of these could fill in blank (d)? (Select one.)

- ○ `Garage`
- ○ `valet`
- ○ `valets`
- ○ `garage`
- ○ `self.valet`
- ○ `self.valets`
- ○ `self.garage`
- ● `self`

(b) (2.0 points)

Definition. An *infinite* iterator `t` is one for which `next(t)` can be called any number of times and always returns a value.

Implement `ring`, a generator function that takes a non-empty list `s`. It returns an infinite generator that repeatedly yields the values of `s` in the order they appear in `s`.

```
def ring(s):
    """Yield all values of non-empty s in order, repeatedly.

    >>> t = ring([2, 5, 3])
    >>> [next(t), next(t), next(t), next(t), next(t), next(t), next(t)]
    [2, 5, 3, 2, 5, 3, 2]
    """
    _________:
       (a)

        _________
           (b)
```

i. (1.0 pt) Which of these could fill in blank (a)? (Select one.)

- ● `while True`
- ○ `while s`
- ○ `for x in s`
- ○ `for x in ring(s)`

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

```
yield from s
```

(c) (5.0 points)

Implement `fork`, a function that takes an infinite iterator `t`. It returns two infinite iterators that each iterate over the contents of `t`.

```
def fork(t):
    """Return two iterators with the same contents as infinite iterator t.

    >>> a, b = fork(ring([1, 2, 3]))
    >>> [next(a), [next(b), next(b), next(b)], next(a), [next(b), next(b), next(b)], next(a)]
    [1, [1, 2, 3], 2, [1, 2, 3], 3]
    """
    s = []
    def copy():
        i = 0
        while True:
            if _________:
                  (a)
               s.append(_________)
                            (b)
            yield _________
                      (c)
            i += 1
    return _________
              (d)
```

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

```
len(s) == i
```

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

```
next(t)
```

iii. (1.0 pt) Which of these could fill in blank (c)? (Select one.)

- ○ `from s`
- ○ `from t`
- ○ `s[0]`
- ● `s[i]`
- ○ `next(t)`

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

```
copy(), copy()
```

4. (10.0 points) Thanos

Hint: you may call built-in sequence functions: `sum`, `max`, `min`, `all`, `any`, `map`, `filter`, `zip`, and `reversed`.

(a) (4.0 points)

Implement `snap`, which takes a one-argument function `f`, a one-argument function `g`, and a sequence `s`. It returns a list of `(x, f(x))` pairs (two-element tuples) for all `x` in `s` for which `g(f(x))` is a true value. The implementation of `snap` only calls `f` once per element of `s`; never twice.

Important: For full credit, your implementation may only call `f` **once** on each element of `s`.

```
def snap(f, g, s):
    """Return a list of (x, f(x)) pairs for each x in s such that g(f(x)) is a true value.

    >>> snap(lambda x: x * x, lambda x: x < 10, range(5))
    [(0, 0), (1, 1), (2, 4), (3, 9)]
    >>> snap(lambda x: x * x, lambda x: x > 10, range(5))
    [(4, 16)]
    >>> snap(lambda x: x * x, lambda x: x and x - 9, range(5))
    [(1, 1), (2, 4), (4, 16)]
    """
    return [(x, _________) for _________ in _________ if _________]
                  (a)            (b)          (c)          (d)
```

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

```
y
```

ii. (1.0 pt) Which of these could fill in blank (b)? (Select one.)

- ○ `x`
- ● `x, y`
- ○ `y`
- ○ `y, z`

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

```
zip(s, map(f, s)) or [(z, f(z)) for z in s]
```

iv. (1.0 pt) Fill in blank (d).

```
g(y)
```

(b) (4.0 points)

Implement `max_diff`, which takes a non-empty sequence `s` and a one-argument function `f`. It returns a pair of elements `(v, w)` in `s` for which `f(v) - f(w)` is largest. `v` and `w` may be the same or different elements of `s`. Also, describe the order of growth of the run time of `max_diff`.

```
def max_diff(s, f):
    """Return two elements (v, w) of s for which f(v) - f(w) is largest.

    >>> max_diff(range(-7, 4), lambda x: x * x)  # (-7 * -7) - (0 * 0) = 49
    (-7, 0)
    >>> max_diff(['what', 'a', 'great', 'film'], len)  # len('great') - len('a')
    ('great', 'a')
    """
    assert s, 's cannot be empty'
    v, w = None, None
    for x in s:
        for y in s:
            if v is None or _________:
                              (a)

                ---------
                   (b)
    return v, w
```

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

```
f(x) - f(y) > f(v) - f(w)
```

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

```
v, w = x, y
```

iii. (2.0 pt) What is the order of growth of the time required to evaluate `max_diff(s, f)` for a sequence `s` of length n and a function `f` that requires constant time to evaluate.

- ○ Exponential, $\Theta(b^n)$
- ● Quadratic, $\Theta(n^2)$
- ○ Linear, $\Theta(n)$
- ○ Logarithmic, $\Theta(\log_2 n)$
- ○ Constant, $\Theta(1)$

(c) (2.0 points)

Implement `max_diff_fast`, which has the same signature and behavior as `max_diff`, but has a faster order of growth of its run time.

Important. You may **not** use a list comprehension in your solution.

```
def max_diff_fast(s, f):

    return _________ , _________
              (a)          (b)
```

i. (1.0 pt) Fill in blank (a), but do not use a comprehension.

```
max(s, key=f)
```

ii. (1.0 pt) Fill in blank (b), but do not use a comprehension.

```
min(s, key=f)
```

5. (9.0 points) Groot

Definition. A *twig* is a tree that is not a leaf but whose branches are all leaves.

The `Tree` and `Link` classes appear on your midterm 2 study guide. Assume they are defined.

(a) (4.0 points)

Implement `twig`, which takes a `Tree` instance t. It returns `True` if `t` is a twig and `False` otherwise.

```
def twig(t):
    """Return True if Tree t is a twig and False otherwise.

    >>> twig(Tree(1))
    False
    >>> twig(Tree(1, [Tree(2), Tree(3)]))
    True
    >>> twig(Tree(1, [Tree(2), Tree(3, [Tree(4)])]))
    False
    """
    return _________ and _________
              (a)           (b)
```

i. (2.0 pt) Which of these could fill in blank (a)? Check all that apply. *Hint*: The `bool` function returns `True` for a true value and `False` for a false value.

- ☐ `t`
- ☐ `bool(t)`
- ☐ `t.is_leaf()`
- ■ `not t.is_leaf()`
- ☐ `t.branches`
- ■ `bool(t.branches)`
- ☐ `len(t.branches)`
- ■ `len(t.branches) > 0`
- ☐ `len(t.branches) >= 0`

ii. (2.0 pt) Fill in blank (b).

```
all([b.is_leaf() for b in t.branches])
```

(b) (5.0 points)

Implement `twigs`, which takes a `Tree` instance `t`. It returns a linked list (either a `Link` instance or `Link.empty`) containing all of the labels of the twigs in `t`. Labels should be in the same order as they appear in `repr(t)`.

```
def twigs(t):
    """Return a linked list of the labels of the twigs in t.

    >>> t = Tree(1, [Tree(2), Tree(3, [Tree(4)]), Tree(5, [Tree(6, [Tree(7), Tree(8)])])])
    >>> print(twigs(t))
    <3 6>
    >>> print(twigs(Tree(0, [t, t, t])))
    <3 6 3 6 3 6>
    >>> twigs(Tree(0)) is Link.empty
    True
    """

    def add_twigs(t, rest):

        if twig(t):

            return _________
                     (a)

        for b in reversed(t.branches):

            rest = _________
                     (b)

        return rest

    return add_twigs(t, _________)
                           (c)
```

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

```
Link(t.label, rest)
```

ii. (2.0 pt) Fill in blank (b).

```
add_twigs(b, rest)
```

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

```
Link.empty
```

No more questions.