CS 61A Structure and Interpretation of Computer Programs
Spring 2023

Final 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. (10.0 points) What Would Python Display?

Assume the following code has been executed. The `Link` class appears on the midterm 2 study guide.

```
class Chain(Link):
    def add(self, v):
        self.rest = Chain(v, self.rest)
        return self

gold = Chain(3, Chain(5, Chain(7)))

silver = [1, [2], 3]
bronze = silver[1:]
silver.append(4)
silver[1].append(5)
```

For each expression below, choose the output displayed when the expression is evaluated. Assume the expressions are evaluated in order in the same interactive session, and so evaluating an earlier expression may affect the result of a later one.

(a) (3.0 pt) `print(gold.rest.add(1))`

- ○ `<3 1 5 7>`
- ○ `<3 <1> 5 7>`
- ○ `<3 5 1 7>`
- ○ `<3 5 <1> 7>`
- ● `<5 1 7>`
- ○ `<5 <1> 7>`
- ○ None of these

(b) (3.0 pt) `print(Chain(2, gold).add(4))`

```
<2 4 3 5 1 7>
```

(c) (2.0 pt) `print(bronze)`

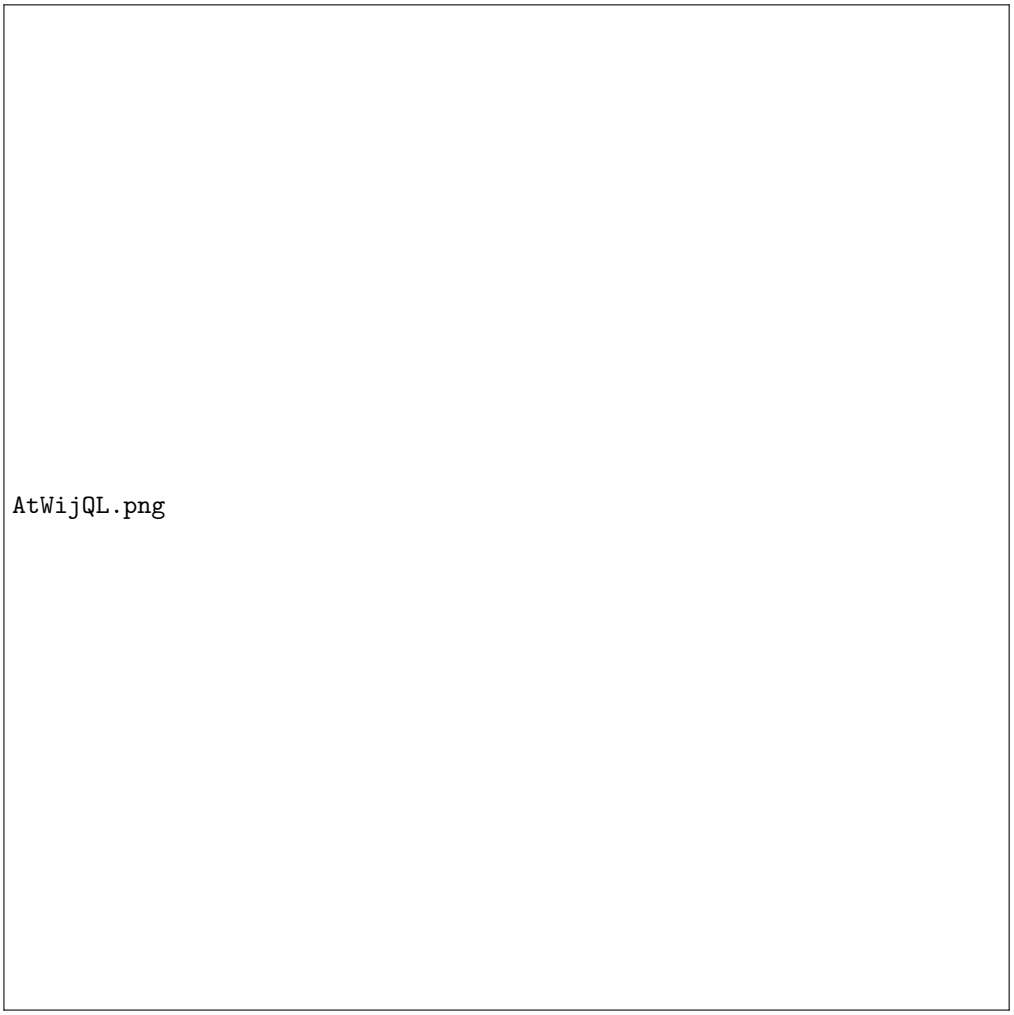
```
[[2, 5], 3]
```

(d) (2.0 pt) What is the order of growth of the time to run the `Chain.add` method with respect to the length of `self`?

- ● Constant
- ○ Logarithmic
- ○ Linear
- ○ Quadratic
- ○ Exponential

2. (5.0 points) Framed

Complete the environment diagram that results from running all of the code below and then answer the questions that follow. Blanks with no labels have no questions associated with them and are not scored.



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

- (●) 2
- ( ) 4
- ( ) 5
- ( ) None of these

(b) (1.0 pt) Blanks (b) and (d) have the same value; what is it?

- ( ) Two functions
- (●) A list containing two functions
- ( ) A list containing a list of two functions
- ( ) A list containing two lists that each contain one function

(c) (3.0 pt) What three numbers fill in blanks (c), (e), and (f) respectively?

- ● 4, 12, 15
- ○ 4, 14, 15
- ○ 4, 15, 15
- ○ 4, 25, 29
- ○ 4, 28, 29
- ○ 4, 36, 36
- ○ 5, 12, 15
- ○ 5, 15, 15
- ○ 5, 25, 29
- ○ 5, 28, 29
- ○ 5, 36, 36

3. (16.0 points) Trees Get Degrees

(a) (10.0 points)

Implement `big_paths`, which takes a `Tree` instance `t` with number labels. It returns the **number of paths** from the root of `t` to a leaf in which **all labels are greater than or equal to the root label**. The `Tree` class appears on the midterm 2 study guide.

```
def big_paths(t):
    """Return the number of paths from the root to a leaf in which all labels are
    greater than or equal to the root label of t.

    >>> t = Tree(5, [Tree(4, [Tree(6), Tree(7)]), Tree(8), Tree(7, [Tree(5), Tree(3)])])
    >>> big_paths(t) # 5-8 and 5-7-5
    2
    """
    def f(s):

        if _________:
              (a)

            return 0

        elif _________:
                (b)

            return 1

        else:

            return _________
                       (c)
    return _________
              (d)
```

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

- ○ `s.is_leaf()`
- ○ `t.is_leaf()`
- ○ `t.label >= s.label`
- ○ `s.label >= t.label`
- ● `t.label > s.label`
- ○ `s.label > t.label`

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

- ● `s.is_leaf()`
- ○ `t.is_leaf()`
- ○ `t.label >= s.label`
- ○ `s.label >= t.label`
- ○ `t.label > s.label`
- ○ `s.label > t.label`

iii. (4.0 pt) Fill in blank (c).

```
sum([f(b) for b in s.branches])
```

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

- ○ `f(s)`
- ● `f(t)`
- ○ `f(s, 0)`
- ○ `f(t, 0)`
- ○ `1 + f(s, 0)`
- ○ `1 + f(t, 0)`

(b) (6.0 points)

Implement `gen`, a generator function that takes a `Tree` instance `t` with number labels and a number `n`. It yields a two-element list for every pair of a parent label and child label that sum to `n`.

```
def gen(t, n):
    """Yield all pairs of a parent label and child label that sum to n in tree t.

    >>> t = Tree(5, [Tree(4, [Tree(6), Tree(7)]), Tree(8), Tree(7, [Tree(5), Tree(3)])])
    >>> sorted(gen(t, 10))  # Does not yield [5, 5]; one 5 is the grandparent of the other.
    [[4, 6], [7, 3]]
    """
    for b in _________:
                (e)
        if _________ == n:
              (f)

            _________
               (g)
        yield from _________
                      (h)
```

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

- [x] `t.branches`
- [ ] `gen(t.branches, n)`
- [ ] `map(lambda x: gen(x, n), t.branches)`

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

- [ ] `b.label`
- [ ] `t.label`
- [x] `t.label + b.label`
- [ ] `sum([b.label for b in t.branches])`

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

```
yield [t.label, b.label]
```

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

- [ ] `t.branches`
- [ ] `b.branches`
- [ ] `[[t, b] for b in t.branches]`
- [ ] `[[t.label, b.label] for b in t.branches]`
- [ ] `gen(t, n)`
- [x] `gen(b, n)`

4. (6.0 points) Repark the Car

Definition. A *car* takes up 2 adjacent parking spots. A string of length `n` can represent `n` adjacent parking spots using `<>` for a car and `.` for an empty spot. For example `'.<>.<><>'` represents an empty spot, then two spots containing a car, then another empty spot, then four spots containing two cars.

Implement `cars`, which takes non-negative integers `k` and `n`. It returns the number of ways to park `k` cars in `n` adjacent parking spots. Treat all cars the same; it does not matter which car is parked in which spot.

```
def cars(k, n):
    """Return the number of ways to park k cars in n adjacent parking spaces where
    each car takes up two spaces.

    >>> cars(2, 5)  # '<><>.', '<>.<>', '.<><>'
    3
    >>> cars(3, 6)  # '<><><>'
    1
    >>> cars(1, 7)  # '<>.....', '.<>....', etc.
    6
    """
    if k == 0:

        return _________
                  (a)

    if _________:
          (b)

        return 0

    return _________
              (c)
```

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

- ○ `0`
- ● `1`
- ○ `n`
- ○ `n + 1`

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

- ○ `k > 0`
- ○ `k < 0`
- ○ `n == 0`
- ○ `n < 1`
- ● `n < 2`

(c) (3.0 pt) Fill in blank (c).

```
cars(k, n-1) + cars(k-1, n-2)
```

5. (16.0 points) Pairings

Definition. The *pairing* of a sequence `s` is a sequence of pairs containing all of the elements of s in order. A pair is a two-element list. E.g., the pairing of `[1, 3, 5, 6, 4, 2]` is `[[1, 3], [5, 6], [4, 2]]`.

(a) (6.0 points)

Implement `pair`, a function that takes a list `s` with an even length. It returns the pairing of `s`.

```
def pair(s):
    """Return the pairing of an even-length list s as a list of two-element lists.

    >>> pair([3, 4, 5, 8, 7, 6])
    [[3, 4], [5, 8], [7, 6]]
    """
    assert len(s) % 2 == 0
    return [_________ for i in _________]
              (a)                (b)
```

i. (3.0 pt) Fill in blank (a).

`[s[2 * i], s[2 * i + 1]]` **OR** `s[2*i:2*i+1]`

ii. (1.0 pt) Fill in blank (b). The built-in `map` function appears on the midterm 2 study guide.

- ○ `map(lambda x: x // 2, s)`
- ○ `map(lambda x: s // 2, s)`
- ○ `range(len(s // 2))`
- ● `range(len(s) // 2)`

iii. (2.0 pt) Fill in blank (c) in the implementation of `gen_pairs`, a generator function that takes an infinite iterator `t`. It yields the elements of the pairing of the sequence represented by `t`.

```
def naturals(k):
    yield k
    yield from naturals(k + 1)

def gen_pairs(t):
    """Yield elements of the pairing of an infinite iterable s.

    >>> t = gen_pairs(naturals(3))  # naturals(3) contains 3, 4, 5, 6, ...
    >>> next(t)
    [3, 4]
    >>> next(t)
    [5, 6]
    """
    while True:
        yield _________
                  (c)
```

`[next(t), next(t)]`

(b) (10.0 points)

Implement `pairs`, a Scheme procedure that takes a list `s` with an even length. It returns the pairing of `s`. For example, `(pairs '(3 4 5 8 7 6))` evaluates to `((3 4) (5 8) (7 6))`.

```
(define (pairs s) (if (null? s) nil ( _________ ( list (car s) _________ ) _________ )))
                                        (d)                    (e)         (f)
```

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

- ○ `append`
- ○ `car`
- ○ `cdr`
- ● `cons`
- ○ `if`
- ○ `list`

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

```
(car (cdr s))
```

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

```
(pairs (cdr (cdr s)))
```

iv. (2.0 pt) What is the value of the expression `(map pairs (pairs '(3 4 5 6)))`? The built-in `map` procedure appears on the final study guide. Assume `pairs` is implemented correctly.

- ○ `(3 4 5 6)`
- ○ `((3 4) (5 6))`
- ○ `(((3 4) (5 6)))`
- ○ `((3 (4)) (5 (6)))`
- ● `(((3 4)) ((5 6)))`

v. (3.0 pt) Which expressions are passed to `scheme_eval` when evaluating `((lambda (x) 1) 2)`? Check all that apply.

- ■ `((lambda (x) 1) 2)`
- ■ `(lambda (x) 1)`
- ☐ `lambda`
- ☐ `(x)`
- ☐ `x`
- ■ `1`
- ■ `2`

6. (14.0 points) Log and Count

(a) (6.0 points)

The `Log` class is constructed with a one-argument function `f` and has a method `call` that takes `n` and returns `f(n)`. The `args` attribute of a `Log` instance for function `f` is a list containing all `n` passed to the `call` method of **any** `Log` instance with the same `f`.

```
class Log:
    """Store the arguments passed to a one-argument function in a list called args.
    If two logs are created for the same function f, they share a list of args.

    >>> f = Log(lambda x: x * x)
    >>> f.call(f.call(3))
    81
    >>> f.args
    [3, 9]
    >>> g, h = Log(abs), Log(abs)
    >>> g.call(h.call(-3))
    3
    >>> g.args
    [-3, 3]
    """
    logged = {}
    def __init__(self, f):
        if f not in self.logged:
            _________ = []
               (a)
        self.f = _________
                    (b)
        self.args = _________
                       (c)
    def call(self, n):
        self.args.append(n)
        return _________
                  (d)
```

i. (3.0 pt) One expression can fill **both blanks (a) and (c)**; what is it?

```
self.logged[f] or Log.logged[f]
```

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

```
f
```

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

- ○ `Log.call(self, n)`
- ○ `self.call(n)`
- ● `self.f(n)`
- ○ `self.f.call(n)`

(b) (8.0 points)

The `Counter` class has a method `observe` that takes an iterable and a method `count` that takes a value `v` and returns the number of times `v` appears among the elements of all the iterables ever passed to `observe`. Finally, the `forget` method takes no arguments and omits the values of the most recent (not yet forgotten) call to `observe` from subsequent calls to `count`. Assume `forget` is never called more times than `observe`.

Hint: The built-in `count` method of a list returns the count of a value: `[3, 4, 3].count(3)` is 2.

```
class Counter:
    """Counts the number of times a value was observed.

    >>> c = Counter()
    >>> c.observe(map(abs, range(-3, 3)))  # observe  3,  2,  1, 0, 1, 2
    >>> c.observe(range(-3, 3))            # observe -3, -2, -1, 0, 1, 2
    >>> {i: c.count(i) for i in range(-3, 3)}
    {-3: 1, -2: 1, -1: 1, 0: 2, 1: 3, 2: 3}
    >>> c.forget()  # forget range(-3, 3)
    >>> {i: c.count(i) for i in range(-3, 3)}
    {-3: 0, -2: 0, -1: 0, 0: 1, 1: 2, 2: 2}
    >>> c.forget()  # forget map(abs, range(-3, 3))
    >>> c.count(1)
    0
    """
    def __init__(self):
        self.obs = []
    def observe(self, vs):
        _________
           (e)
    def count(self, v):
        return __________
                  (f)
    def forget(self):
        _________
           (g)
```

i. (3.0 pt) Fill in blank (e).

```
self.obs.append(list(vs))
```

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

```
sum([s.count(v) for s in self.obs])
```

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

- ○ `obs.pop()`
- ● `self.obs.pop()`
- ○ `[obs.pop() for i in range(len(vs))]`
- ○ `[self.obs.pop() for i in range(len(vs))]`

7. (8.0 points) Room Service

The `finals` table has columns `hall` (strings) and `course` (strings), and has rows for each lecture `hall` in which a `course` is holding its final exam. Each `course` and each `hall` value may appear multiple times because a course may use multiple lecture halls at the same time, and a lecture hall may hold finals for multiple courses at different times.

The `sizes` table has columns `room` (strings) and `seats` (numbers), and has one row per unique `room` on campus containing the number of `seats` in that room. Each `room` appears once. All lecture halls are rooms.

Create a table with two columns, `course` (string) and `seats` (number), and with one row containing the name of the `course` and the total number of `seats` in final rooms for that course. Only include a row for **each course that uses at least two rooms for its final**.


```
SELECT course, SUM(_________) AS seats FROM _________ GROUP BY _________;
                       (a)                      (b)               (c)
```

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

- ◯ `course`
- ◯ `room`
- ● `seats`
- ◯ `sizes`

(b) (4.0 pt) Fill in blank (b). You may include a `WHERE` or `HAVING` clause.

```
finals, sizes WHERE hall=room
```

(c) (3.0 pt) Fill in blank (c).

- ◯ `hall WHERE seats > 1`
- ◯ `hall HAVING seats > 1`
- ◯ `hall WHERE sizes > 1`
- ◯ `hall HAVING sizes > 1`
- ◯ `hall WHERE COUNT(*) > 1`
- ◯ `hall HAVING COUNT(*) > 1`
- ◯ `course WHERE seats > 1`
- ◯ `course HAVING seats > 1`
- ◯ `course WHERE sizes > 1`
- ◯ `course HAVING sizes > 1`
- ◯ `course WHERE COUNT(*) > 1`
- ● `course HAVING COUNT(*) > 1`

8. Group Work

These A+ questions are not worth any points. They can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first!

Definition. The *n-grouping* of a list `s` for positive integer `n` is a list of `n`-length lists containing all of the elements of s in order. The last list may contain fewer than `n` elements.

E.g., the *3-grouping* of `[4, 3, 2, 1, 5, 6, 7, 8]` is `[[4, 3, 2], [1, 5, 6], [7, 8]]`.

(a) Fill in the blank to implement `group`, which takes positive integer `n` and list `s`. It returns the the `n`-grouping of `s`.

```
def group(n, s):
    """Return the n-grouping of a list s as a list of n-element lists.

    >>> group(2, [3, 4, 5, 6, 7, 8])
    [[3, 4], [5, 6], [7, 8]]
    >>> group(3, [3, 4, 5, 6, 7, 8])
    [[3, 4, 5], [6, 7, 8]]
    >>> group(4, [3, 4, 5, 6, 7, 8])
    [[3, 4, 5, 6], [7, 8]]
    """
    if len(s) <= n:
        return [s]
    else:
        return _________
```

```
[s[:n]] + group(n, s[n:])
```

(b) Fill in the blank to implement `group`, which takes positive integer `n` and list `s`. It returns the the `n`-grouping of `s`.

```
; (group 3 '(3 4 5 6 7 8)) evaluates to ((3 4 5) (6 7 8))
; (group 4 '(3 4 5 6 7 8)) evaluates to ((3 4 5 6) (7 8))

(define (group n s)
  (define (first n s f)
    (cond ((null? s) nil)
          ((= n 0) (cons nil (f s)))
          ((null? (cdr s)) (list (list (car s))))
          (else (let ((t (first (- n 1) (cdr s) f)))
                  (cons (cons (car s) (car t)) (cdr t))))))
  _________ )
```

```
(first n s (lambda (s) (group n s)))
```

No more questions.