CS 61A Spring 2020

Structure and Interpretation of Computer Programs

Final Exam

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 `cs61a@berkeley.edu`. 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) Who is your TA? (See cs61a.org/staff.html for pictures.)

1. (8.0 points) What Does This Function Do?

Complete the description of each function so that it correctly describes the function's behavior.

(a) (4.0 points)

```
def count(n, t, k):
    if n == 0:
        return int(t <= 0)
    elif k > n:
        return 0
    else:
        a = count(n, t, k + 1)
        b = count(n-k, t-1, k)
        return a + b
```

Hint: `int(False)` evaluates to `0` and `int(True)` evaluates to `1`.

As in `count_partitions` from the Midterm 2 Study Guide, a "way of summing to `n` using parts" is a sum of zero or more positive integers (the parts) that appear in non-decreasing order and total `n`. For example, `1 + 2` is a way of summing to `3` using 2 parts, but `2 + 1` is not.

Complete this description: `count(n, t, k)` counts the ways of summing to `n` using ...

i. (2.0 pt)

◯ ... at least `t` parts ...

◯ ... at most `t` parts ...

◯ ... at least `k` parts ...

◯ ... at most `k` parts ...

ii. (2.0 pt)

◯ ... that are all less than or equal to `t`.

◯ ... that are all greater than or equal to `t`.

◯ ... that are all less than or equal to `k`.

◯ ... that are all greater than or equal to `k`.

(b) (4.0 points)

```
def prime(n):
    """Return the smallest prime number larger than n.

    >>> prime(2)
    3
    >>> prime(8)
    11
    >>> prime(prime(8))
    13
    >>> prime(prime(prime(8)))
    17
    """
    <implementation omitted>

def again():
    f, g = prime, prime
    def h(x):
        nonlocal g
        g, h = (lambda h: lambda y: h(f(y)))(g), g(x)
        return h
    return h
```

Assume that `prime` is implemented correctly and behaves as its docstring describes.

Below, *applying `prime` to `x` repeatedly 3 times* means evaluating `prime(prime(prime(x)))`.

Complete this description: `again()` returns a function `h` that takes a number `x` and returns the result of applying `prime` to `x` repeatedly ...

i. (2.0 pt)

- ◯ ... `k-1` times ...
- ◯ ... `k` times ...
- ◯ ... `k+1` times ...
- ◯ ... `2 ** k` times ...

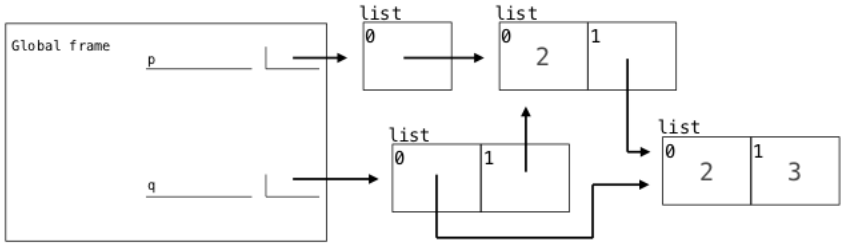
ii. (2.0 pt)

- ◯ ... where `k` is the number of times that this `h` function has been called.
- ◯ ... where `k` is the number of times that `prime` has been called during a call to this `h` function.
- ◯ ... where `k` is the total number of times that all `h` functions returned by calling `again()` (perhaps multiple times) have been called throughout the whole program.
- ◯ ... where `k` is the total number of times that `prime` has been called (perhaps in other ways than by this `h` function) throughout the whole program.

2. (12.0 points) Mind Your P's and Q's

(a) (6.0 points)

Fill in each blank in the code example below so that its environment diagram is the following:



https://i.imgur.com/xPJsDGg.png

```
p = [[2], [2, 2]]

p[0]._________(_________)
       (a)       (b)

q = [_________, _________]
       (c)       (d)

p_________ = 3
    (e)
```

i. (1.0 pt) Which of the following names could complete blank (a)?

- ◯ `add`
- ◯ `pop`
- ◯ `append`
- ◯ `extend`

ii. (1.0 pt) Which of the following expressions could complete blank (b)?

- ◯ `p`
- ◯ `p[0]`
- ◯ `p[1]`
- ◯ `p[:]`

iii. (1.0 pt) Which of the following expressions could complete blank (c)?

- ◯ `p.pop()`
- ◯ `p[1]`
- ◯ `p[0]`
- ◯ `p[:1]`

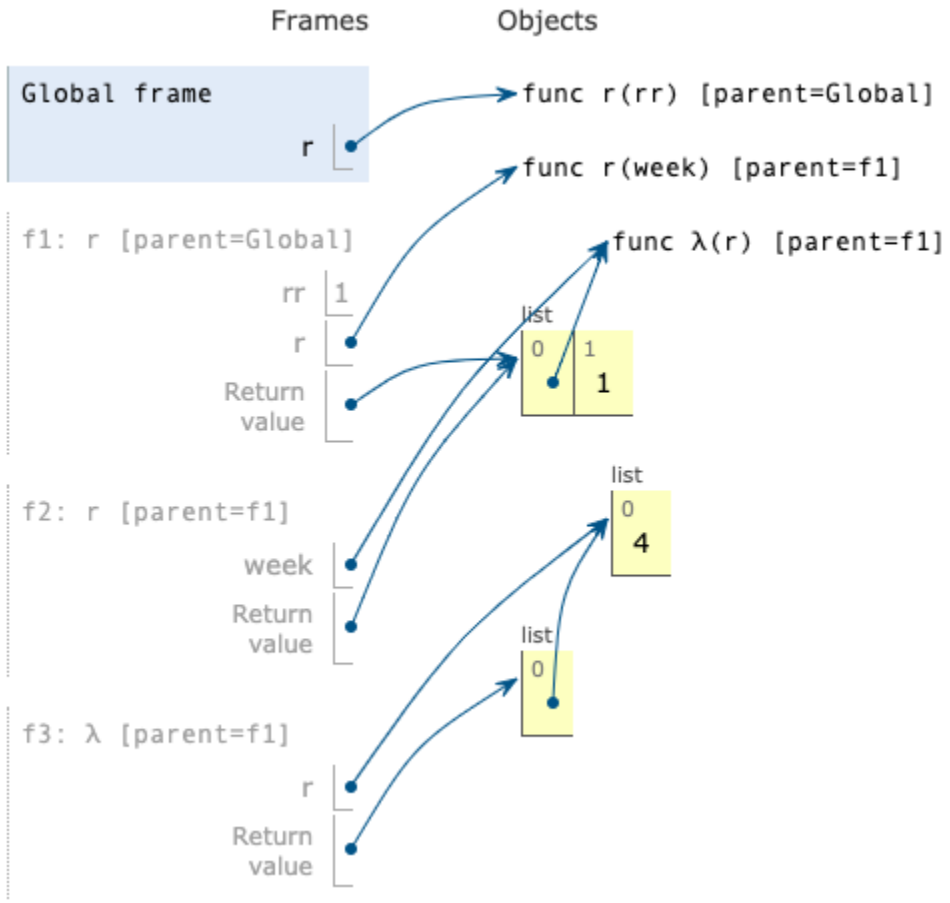
iv. (1.0 pt) Which of the following expressions could complete blank (d)?

- ◯ `p.pop()`
- ◯ `p[1]`
- ◯ `p[0]`
- ◯ `p[:1]`

v. (2.0 pt) Write code that could complete blank (e).

(b) (6.0 points)

Fill in each blank in the code example below so that its environment diagram is the following:



https://i.imgur.com/qk83xRw.png

```
def r(rr):

    if rr:

        def r(week):

            return [_________, rr]
                       (a)

    rr = _________
            (b)

    return r(_________)
                (c)

r(5)_________
       (d)
```

Note: Line numbers for lambda functions have been omitted intentionally.

i. (1.0 pt) Write an expression that could complete blank (a).

ii. (1.0 pt) Write an expression that could complete blank (b).

iii. (2.0 pt) Write an expression that could complete blank (c).

iv. (2.0 pt) Which of these could complete blank (d)?

- ◯ `.pop()([4])`
- ◯ `.pop()(4)`
- ◯ `[0](4)`
- ◯ `[0]([4])`

3. (12.0 points) Bounds

Definitions: A *bound* is a two-element tuple of numbers in which element 0 is smaller than element 1. A number `t` is *contained in* bound `b` if `b[0] < t and t < b[1]`. How *tight* a bound `b` is around a number `t` describes the largest absolute distance between `t` and one of the numbers in `b`. For example, the tightness of bound `(1, 7)` around `6` is 5 because the absolute difference between 6 and 1 is 5.

(a) (2.0 points)

Implement `minimum`, which takes a list `s` and a one-argument function `key`. It returns the value in `s` for which `key` produces the smallest return value. If `s` is empty, `minimum` returns `None`. If more than one value in `s` produces a `key` value at least as small as all others, then `minimum` returns the first.

```
def minimum(s, key):
    """Return the first value v in s for which key(v) is less than or equal to
    key(w) for all values w in s. Return None if s is empty.

    >>> minimum([5, 4, 3, 2, 1], lambda x: abs(x - 3.1)) # Closest to 3.1
    3
    >>> a = [3]
    >>> minimum([[5], [4], a, [3], [2], [1]], lambda x: abs(x[0] - 3.1)) is a
    True
    """
    if not s:

        return None

    m = s[0]

    for v in s[1:]:

        if _________:
              (a)

            m = v

    return m
```

i. (2.0 pt) What expression completes blank (a)?

Important: You may not call the built-in `min` or `max` functions for this blank.

(b) (4.0 points)

Implement `tightest`, which takes a list of `bounds` and a number `t`. It returns the first bound in `bounds` that both contains `t` and is the most tight around `t`. If no bound in `bounds` contains `t`, `tightest` returns `None`.

Assume `minimum` is implemented correctly.

```
def tightest(bounds, t):
    """Return the tightest bound around t in bounds.

    >>> bounds = [(2, 6), (3, 4), (1, 5), (1, 6), (0, 4)]
    >>> tightest(bounds, 3)
    (1, 5)
    >>> tightest(bounds, 3.1)
    (3, 4)
    >>> tightest(bounds, 5)
    (2, 6)
    >>> tightest(bounds, 2)
    (0, 4)
    >>> print(tightest(bounds, 6))
    None
    """
    return minimum([b for b in bounds if _________],
                                          (a)

                   lambda b: _________)
                                (b)
```

i. (2.0 pt) What expression completes blank (a)?

Important: You may not call the built-in `min` or `max` functions for this blank.

ii. (2.0 pt) What expression completes blank (b)?

- ◯ `max(t - b[0], b[1] - t)`
- ◯ `abs(t - max(b))`
- ◯ `[abs(t - x) for x in b][0]`
- ◯ `abs(max([t - x for x in b]))`

(c) (6.0 points)

Implement `nest`, which takes a list of `bounds`. It returns the largest number of bounds in the list that all overlap with each other.

```
def overlap(a, b):
    """Return whether there is some number t contained in both a and b.

    >>> overlap([2, 4], [1, 3])  # 2.5 is contained in both bounds.
    True
    >>> overlap([1, 3], [2, 4])  # 2.5 is contained in both bounds.
    True
    >>> overlap([2, 4], [1, 2])  # No number is contained in both bounds.
    False
    """
    return a[0] < b[1] and b[0] < a[1]

def nest(bounds):
    """Return the largest number of bounds that all contain the same number.

    >>> bounds = [(2, 6), (3, 4), (1, 5), (1, 6), (0, 4), (0, 3)]
    >>> nest(bounds)  # All but the last contain 3.1, so these 5 all overlap with each other.
    5
    >>> bounds = [(1, 5), (5, 7), (7, 9), (1, 9)]
    >>> nest(bounds)  # Any of the first three overlaps with the last, but not with each other.
    2
    >>> bounds = [(1, 9), (1, 5), (5, 7), (7, 9)]
    >>> nest(bounds)  # The first overlaps with any of the last three.
    2
    >>> bounds = [(2, 4), (1, 3), (1, 2)]
    >>> nest(bounds)  # Any two consecutive bounds overlap, but the first & last do not overlap.
    2
    """
    if not bounds:

        return 0

    rest = [b for b in bounds[1:] if overlap(b, _________)]
                                                    (a)

    return max(nest(_________), 1 + _________)
                      (b)             (c)
```

i. (2.0 pt) What expression completes blank (a)?

ii. (2.0 pt) What expression completes blank (b)?

- ◯ `bounds`
- ◯ `bounds[1:]`
- ◯ `rest`
- ◯ `bounds[0] + rest`

iii. (2.0 pt) What expression completes blank (c)?

4. (12.0 points) What Does This Function or Method Do?

Complete the description of each function or method so that its behavior is described correctly.

(a) (4.0 points) Game

```
def time(hour, minute, second):
    """Create a time value using data abstraction."""
    <implementation omitted>

def format(t):
    """Return a string that formats a time such that the hours, minutes,
    and seconds all have two digits.

    >>> format(time(3, 30, 5))
    '03:30:05'
    """
    <implementation omitted>

class Game:
    time = None
    def __init__(self, time):
        Game.time = time
    def __str__(self):
        return format(self.time)
```

Assume that `time` takes numbers that can represent the hours, minutes, and seconds of a valid time and returns a value that can be passed to `format`. Assume that `format` behaves as described in its docstring. A `Game` instance is constructed from the return value of a call to `time`.

Complete this description:

`print(Game(time(2, 10, 0)), Game(time(3, 0, 0)))` will display ...

i. (4.0 pt)

◯ ... `02:10:00 03:00:00`

◯ ... `03:00:00 03:00:00`

◯ ... `'02:10:00' '03:00:00'`

◯ ... `'03:00:00' '03:00:00'`

◯ ... `Game(time(2, 10, 0)) Game(time(3, 0, 0))`

◯ ... `Game(time(3, 0, 0)) Game(time(3, 0, 0))`

◯ ... `'Game(time(2, 10, 0))' 'Game(time(3, 0, 0))'`

◯ ... `'Game(time(3, 0, 0))' 'Game(time(3, 0, 0))'`

(b) (4.0 points) Mystery

```
def mystery(t):
    def e(r, y):
        assert type(r.label) == int
        myst = [e(b, max(y, r.label)) for b in r.branches]
        if r.label > y:
            myst.append(r.label)
        return sum(myst)
    return e(t, 0)
```

Assume that `mystery` is called on a `Tree` instance with integer labels. Assume that the sum of an empty set of labels is 0. The `Tree` class is defined on the Midterm 2 Study Guide.

An *ancestor* is a parent, or parent's parent, or parent's parent's parent, etc.

A *descendant* is a child, or child's child, or childs' child's child, etc.

Complete this description: `mystery(t)` returns the sum of ...

i. (2.0 pt)

◯ ... all leaf labels in `t` ...

◯ ... all labels in `t` ...

◯ ... all positive leaf labels in `t` ...

◯ ... all positive labels in `t` ...

ii. (2.0 pt)

◯ ... that are either the root label or larger than their parent label.

◯ ... that are either the root label or larger than all their ancestor labels.

◯ ... that are either leaf labels or larger than all their child labels.

◯ ... that are either leaf labels or larger than all their descendant labels.

(c) (4.0 points) Add

```
class Add:
    s = 2
    def __init__(self, s):
        assert isinstance(s, Link) or s is Link.empty
        self.t = s
        s = self.s + 1
    def this(self, v):
        def f(t):
            if t is Link.empty or t.first >= v:
                return Link(v, t)
            else:
                return Link(t.first, f(t.rest))
        for i in range(self.s):
            self.t = f(self.t)
```

Assume `Add` is called on `Link.empty` or a `Link` instance containing numbers, and the `this` method is called on a number. The `Link` class is defined on the Midterm 2 Study Guide.

Complete this description: For an instance `a = Add(s)`, the expression `a.this(v)` inserts `v` into `a.t` ...

i. (2.0 pt)

◯ ... once ...

◯ ... twice ...

◯ ... three times ...

◯ ... a number of times equal to two plus the number of Add instances ever constructed ...

ii. (2.0 pt)

◯ ... at the latest position within `a.t` where `v` is larger than all previous elements in `a.t`.

◯ ... at the earliest position within `a.t` where `v` is smaller than all subsequent elements in `a.t`.

◯ ... at the latest position within `a.t` where `v` is smaller than all previous elements in `a.t`.

◯ ... at the earliest position within `a.t` where `v` is larger than all subsequent elements in `a.t`.

5. (6.0 points) Multiples

Implement `multiples`, a generator function that takes positive integers `k` and `n`. It yields all positive multiples of `k` that are smaller than `n` in decreasing order.

```
def multiples(k, n):
    """Yield all positive multiples of k less than n in decreasing order.

    >>> list(multiples(10, 50))
    [40, 30, 20, 10]
    >>> list(multiples(3, 25))
    [24, 21, 18, 15, 12, 9, 6, 3]
    >>> list(multiples(3, 3))
    []
    """
    if _________:
          (a)

        for why in _________:
                      (b)

            yield _________
                      (c)

        yield k
```

(a) (2.0 pt) Which expression completes blank (a)?

- ◯ `k < n`
- ◯ `k > 0`
- ◯ `n > 0`
- ◯ `k > 0 and n > 0`

(b) (2.0 pt) Which expression completes blank (b)?

- ◯ `multiples(k, n // 10)`
- ◯ `multiples(k, n - 1)`
- ◯ `multiples(k, n - k)`
- ◯ `multiples(k, n / k)`

(c) (2.0 pt) What expression completes blank (c)?

6. (16.0 points) Meeting in Venue

Implement the methods of the `User` and `Meeting` classes as follows:

- When a `User` decides to `attend` a `Meeting` for the first time, if they are the `host` of the `Meeting`, they will be added to the end of the `approved` list; otherwise they will be added to the end of the `waiting` list.
- When a `User` attempts to `attend` a meeting again, they are not added to any list. A string is returned stating that the `User` is already attending.
- A `Meeting`'s `admit` method takes a function `f` that takes a `User` and returns whether they should be admitted. The `admit` method moves all `waiting` `Users` for which `f` returns a true value from the `waiting` list to the end of the `approved` list.

```
class User:
    """A User can attend a Meeting.

    >>> john = User('denero@berkeley')
    >>> oski = User('oski@berkeley')
    >>> jack = User('jack@junioruniversity')
    >>> section = Meeting(john)
    >>> for x in [john, oski, jack]:
    ...     x.attend(section)
    >>> section.approved
    [User('denero@berkeley')]
    >>> section.waiting
    [User('oski@berkeley'), User('jack@junioruniversity')]

    >>> oski.attend(section)
    oski@berkeley is already attending

    >>> section.admit(lambda x: 'berkeley' in x.identifier)
    >>> section.approved
    [User('denero@berkeley'), User('oski@berkeley')]
    >>> section.waiting
    [User('jack@junioruniversity')]

    >>> oski.attend(section)
    oski@berkeley is already attending
    >>> User('denero@berkeley').attend(section) # A different user with the same identifier can attend
    >>> section.waiting
    [User('jack@junioruniversity'), User('denero@berkeley')]
    """
    def __init__(self, identifier):

        self.identifier = identifier

    def attend(self, meeting):

        if _________ in _________:
              (a)          (b)

            print(self.identifier, 'is already attending')

        else:

            users = _________
                       (c)
```

```
            if _________:
                  (d)

                users = _________
                            (e)

            users.append(_________)
                           (f)

    def __repr__(self):

        return 'User(' + repr(self.identifier) + ')'

class Meeting:
    """A Meeting can admit waiting Users."""
    def __init__(self, host):
        self.waiting = []
        self.approved = []
        self.host = host

    def admit(self, f):

        for x in self.waiting:

            if _________:
                  (g)

                self.approved.append(x)

        self.waiting = _________
                           (h)
```

(a) (2.0 pt) What expression completes blank (a)?

(b) (2.0 pt) Which expression completes blank (b)?

- ◯ `meeting.pending + meeting.joined`
- ◯ `[meeting.pending, meeting.joined]`
- ◯ `meeting.pending.extend(meeting.joined)`
- ◯ `meeting.pending.append(meeting.joined)`

(c) (2.0 pt) What expression completes blank (c)?

(d) **(2.0 pt)** Which expression completes blank (d)?

- ◯ `self.identifier in meeting`
- ◯ `self in meeting`
- ◯ `self.identifier in meeting.host`
- ◯ `self in meeting.host`
- ◯ `self.identifier in meeting.joined`
- ◯ `self in meeting.joined`
- ◯ `self.identifier == meeting.host`
- ◯ `self is meeting.host`

(e) **(2.0 pt)** What expression completes blank (e)?

(f) **(2.0 pt)** What expression completes blank (f)?

(g) **(2.0 pt)** What expression completes blank (g)?

(h) **(2.0 pt)** What expression completes blank (h)?

7. (14.0 points) Apply Here

(a) (4.0 points) max

Implement `max`, which takes a non-empty list of integers and returns the largest.

For example, `(max '(1 4 3 5 2))` evaluates to 5.

The built-in `length` procedure returns the length of a list.

```
(define (max vals) (if (= (length vals) 1) _________
                                            (a)

                       (_________ ((_________ _________))
                           (b)          (c)       (d)

                         (if (> (car vals) enormous) (car vals) enormous) )))
```

i. (1.0 pt) What expression completes blank (a)?

ii. (1.0 pt) Which completes blank (b)?

- ◯ `define`
- ◯ `let`
- ◯ `if`
- ◯ `cond`
- ◯ `begin`

iii. (1.0 pt) What expression completes blank (c)?

iv. (1.0 pt) What expression completes blank (d)?

(b) (6.0 points) partial

Implement `partial`, which takes a procedure `action` and a list `args`. It returns a procedure that takes one argument `final-arg` and returns the result of calling `action` on the values in `args` as well as `final-arg`, in that order.

For example `((partial - '(10 2)) 7)` evaluates to 1, just like `(- 10 2 7)`.

Assume that `action` can be called on `k` arguments, where `k` is one more than the length of `args`.

Hint: The built-in `apply` procedure takes two arguments: a procedure and a list of arguments. It returns the result of calling the procedure on the arguments. For example, `(apply - '(10 2 7))` evaluates to 1, just like `(- 10 2 7)`.

```
(define (partial action args)

  (_________ _________ (apply action (_________ args _________))))
      (a)       (b)                      (c)            (d)
```

i. (1.0 pt) What completes blank (a)?

ii. (1.0 pt) Which completes blank (b)?

- ◯ `final-arg`
- ◯ `(final-arg)`
- ◯ `()`
- ◯ `(args final-arg)`

iii. (1.0 pt) Which symbol completes blank (c)?

Hint: All of these built-in procedures are demonstrated on top of Page 1 of the final study guide.

- ◯ `cons`
- ◯ `list`
- ◯ `append`
- ◯ `car`
- ◯ `cdr`

iv. (3.0 pt) What expression completes blank (d)?

(c) (4.0 points) max again

Suppose there were a built-in `largest` procedure that took one or more numbers as arguments and returned the largest argument. For example, `(largest 1 4 3 5 2)` would evaluate to 5.

Again, define `max`, which takes a list of numbers and returns its largest element. This time, **you may use only** the symbols `largest`, `partial`, `apply`, and `list`, along with parentheses, in your implementation. **You may not use any special forms** (such as quotation, lambda, etc.).

Important: `largest` takes multiple arguments that are numbers, while `max` takes a single argument that is a list of numbers.

For example, `(max '(1 4 3 5 2))` evaluates to 5, just like `(largest 1 4 3 5 2)`.

```
(define (max args) _________)
                       (a)
```

i. (4.0 pt) You may use only `largest`, `partial`, `apply`, `list`, and parentheses.

What expression completes blank (a)?

8. (13.0 points) Contact Tracing

Each row of the `visits` table describes a visit to some establishment by an individual.

- The `who` column indicates the individual's name. Assume everyone has a unique name.
- The `venue` column indicates which establishment they visited.
- The `arrive` column indicates the hour they arrived and the `depart` column indicates the hour they departed. Assume all visits begin and end on the hour. All hours are from 1pm to 11pm. Assume `depart` is greater than `arrive`.

```
CREATE TABLE visits AS
    SELECT "Oski" AS who, "Bar" AS venue, 4 AS arrive, 6 AS depart UNION
    SELECT "Oski"       , "Grocery"     , 6          , 8           UNION
    SELECT "Oski"       , "Bar"         , 8          , 11          UNION
    SELECT "Jane"       , "Grocery"     , 5          , 7           UNION
    SELECT "Jane"       , "Restaurant"  , 7          , 8           UNION
    SELECT "Jane"       , "Bar"         , 8          , 10          UNION
    SELECT "Jack"       , "Restaurant"  , 7          , 9           UNION
    SELECT "Jack"       , "Bar"         , 9          , 10;
```

(a) (8.0 points) contacts

Create the `contacts` table. Each row describes a period of time in which another individual was in the same establishment as Oski. If one individual departs just as another arrives, they do not make contact; they must overlap in time to be included in the `contacts` table.

- The `other` column indicates the name of the individual (not Oski) who came in contact with Oski.
- The `location` column indicates the establishment in which they made contact.
- The `start` column indicates the hour that the contact period began.
- The `stop` column indicates the hour that the contact period ended.

The contents of `contacts` are below. **Any row order is fine.**

```
other | location | start | stop
======|==========|=======|======
Jane  | Grocery  | 6     | 7
Jane  | Bar      | 8     | 10
Jack  | Bar      | 9     | 10
```

Important: Your query should construct `contacts` correctly even if the rows of `visits` were different.

Hint: The `MIN` function computes the smaller of two values when called on two arguments. The `MAX` function computes the larger. These are not aggregate functions when called on two arguments.

```
CREATE TABLE contacts AS

    SELECT b.who                   AS other,
           b.venue                 AS location,
           MAX(a.arrive, b.arrive) AS start,
           MIN(a.depart, b.depart) AS stop

    FROM visits AS a, visits AS b

    WHERE a.who = _________ AND _________ AND _________ AND _________;
                    (a)           (b)           (c)           (d)
```

i. (2.0 pt) What completes blank (a)?

ii. (2.0 pt) Which expression completes blank (b)?

- ◯ `a.who = b.who`
- ◯ `a.who != b.who`
- ◯ `a.who < b.who`
- ◯ `a.who > b.who`

iii. (2.0 pt) Which expression completes blank (c)?

- ◯ `stop > start`
- ◯ `start > stop`
- ◯ `stop >= start`
- ◯ `start >= stop`
- ◯ `start = stop`
- ◯ `start != stop`

iv. (2.0 pt) What expression completes blank (d)?

(b) (5.0 points) time

Select one row per individual other than Oski. Each row should contain two columns: the `name` of the individual and the total `time` in hours that they spent in contact with Oski.

The result appears below. **Any row order is fine.**

```
name | time
=====|======
Jane | 3
Jack | 1
```

Important: Your query should construct this result correctly even if the rows of `contacts` were different. Assume `contacts` is constructed correctly.

```
SELECT _________ AS name, _________ AS time FROM contacts _________;
          (a)                (b)                              (c)
```

i. (1.0 pt) What expression completes blank (a)?

- ◯ `who`
- ◯ `name`
- ◯ `other`
- ◯ `"who"`
- ◯ `"name"`
- ◯ `"other"`

ii. (2.0 pt) What expression completes blank (b)?

iii. (2.0 pt) What clause completes blank (c)?

9. (7.0 points) Expression Tree

Definition: A *tree expression* for a `Tree` instance `t` is a string that starts with `t` and contains a Python expression that evaluates to a node label within `t` by using `branches` and `label` attributes and item selection.

For example, if `t = Tree(3, [Tree(4, [Tree(-1)]), Tree(-5)])`, then there are 4 tree expressions for `t`:

- `'t.label'`
- `'t.branches[0].label'`
- `'t.branches[1].label'`
- `'t.branches[0].branches[0].label'`.

The `Tree` class is defined on the Midterm 2 Study Guide.

(a) (4.0 points) labels

Implement `labels`, which takes a `Tree` instance `t` and returns a list of all tree expressions for `t` in any order.

```
def labels(t):
    """List all tree expressions for tree t.

    >>> t = Tree(3, [Tree(4, [Tree(-1)]), Tree(-5)])
    >>> for e in labels(t):
    ...     print(e)
    t.label
    t.branches[0].label
    t.branches[0].branches[0].label
    t.branches[1].label
    """
    def traverse(t, e):

        result.append(_________)
                        (a)

        for i in range(len(t.branches)):

            traverse(t.branches[i], _________)
                                        (b)
    result = []
    traverse(t, 't')
    return result
```

i. (2.0 pt) What expression completes blank (a)?

ii. (2.0 pt) What expression completes blank (b)?

(b) (3.0 points) smallest

Complete the expression below that evaluates to the node label within `t` that is closest to zero (i.e., has the smallest absolute value). For `t = Tree(3, [Tree(4, [Tree(-1)]), Tree(-5)])`, this would evaluate to `-1`.

Assume `labels` is implemented correctly and the name `t` is bound to a `Tree` instance in the current environment. The built-in `eval` function takes a string and returns the result of evaluating the expression contained in the string in the current environment.

Important: Complete each blank with a single name.

```
_________([eval(_________) for e in labels(t)], key=_________)
   (a)             (b)                                 (c)
```

i. (1.0 pt) What name completes blank (a)?

ii. (1.0 pt) What name completes blank (b)?

iii. (1.0 pt) What name completes blank (c)?

No more questions.