CS 61A Fall 2025

Structure and Interpretation of Computer Programs

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. (5.0 points) What Would Python Display?

Assume the following code has been executed.

```
square = lambda x: x * x

def times(a, b, c):
    x = a
    f = lambda y: x * y
    x = b
    g = (lambda q: lambda r: q * r)(x)
    x = c
    print(f(g(x)))

def delay(s):
    last = None
    for x in s:
        if last:
            print(x)
            return last
        last = x

w = [print, abs, square, print]
```

Write the output that would be displayed by evaluating each expression below. If an error occurs, write ERROR, but also write any output that is displayed before the error occurs.

Example: For `print(square(5))` you should answer 25.

(a) (2.0 pt) `times(2, 3, 10)`

```
300
```

(b) (3.0 pt) `print(delay(map(lambda f: f(-2), w)))`

```
-2
4
2
```

2. (5.0 points) Do You Copy?

Draw an environment diagram to answer the following questions. Only the questions will be scored.

```
 1: def copy(s):
 2:     return s[:1] + s[1:]
 3: inner = [5, [6]]
 4: first = copy(inner)
 5: second = inner[1]
 6: inner[0] = 7
 7: inner.append(7)
 8: inner[1].append(7)
 9: inner[1] = 8
10: print(first)
11: print(second)
```

(a) (2.0 pt) What is displayed by `print(first)` on line 10?

- ○ `[5, [6]]`
- ○ `[5, [6], 7]`
- ○ `[5, [6], 7, 7]`
- ● `[5, [6, 7]]`
- ○ `[5, [6, 7], 7]`
- ○ `[7, [6], 7]`
- ○ `[7, [6, 7]]`
- ○ `[7, [6, 7], 7]`
- ○ `[7, 8]`
- ○ `[7, 8, 7]`
- ○ `[[6]]`
- ○ `[[6, 7]]`
- ○ `[[6], 7]`
- ○ `[[6, 7], 7]`
- ○ `[8]`
- ○ `[8, 7]`

(b) (2.0 pt) What is displayed by `print(second)` on line 11?

- ○ `8`
- ○ `[6]`
- ○ `[[6]]`
- ● `[6, 7]`
- ○ `[[6, 7]]`
- ○ `[[6], 7]`
- ○ `[8]`
- ○ `[8, 7]`

(c) (1.0 pt) What is the order of growth of the time it takes to evaluate `[sum(s[:i]) for i in range(len(s))]` in terms of the length of `s` for a list of numbers `s`. The `sum` function takes linear time in the length of its input. Slicing a list takes linear time in the length of the slice. Creating a `range` and calling `len` on a list both take constant time (one step each).

- ○ logarithmic
- ○ linear
- ● quadratic
- ○ exponential

3. (21.0 points) Math Placement Exam

Definition. An *equation* is a list of numbers and operators. A *number* is either an integer or a `?`. An *operator* is either `+`, `-`, or `=`. Equations start and end with a number, alternate between numbers and operators, and contain exactly one `=`.

(a) (5.0 points)

Implement `is_equation`, which takes a list `s`. It returns whether `s` is an equation.

```
def is_equation(s):
    """Return whether the list s contains a well-formed equation.
    >>> is_equation([7, '+', 2, '=', '?', '+', 6])
    True
    >>> is_equation([-7, '+', '?', '=', '?', '-', 6])
    True
    >>> is_equation([-7, '+', 2, '=', '?', '-'])          # No number at the end
    False
    >>> is_equation(['-', 7, '+', 2, '=', '?', '-', 6])   # No number at the start
    False
    >>> is_equation([-7, '+', 2, '?', '+', 6, '=', 9])    # Two adjacent numbers (2 and ?)
    False
    >>> is_equation([-7, '+', 2, '+', '-', 6, '=', 9])    # Two adjacent operators (+ and -)
    False
    >>> is_equation([-7, '+', 2, '=', '?', '=', 6])       # More than one =
    False
    """
    number, equals = True, 0
    for x in s:
        ok = (number and (_______)) or (_______ ['+', '-', '='])
        if not ok:          (a)           (b)
            return False
        if x == '=':
            equals += 1
        number = not number
    return _______
             (c)
```

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

- ○ `isinstance(x, int) or x == ?`
- ○ `isinstance(x, int) and x == ?`
- ● `isinstance(x, int) or x == '?'`
- ○ `isinstance(x, int) and x == '?'`

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

```
not number and x in
```

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

```
not number and equals == 1
```

(b) (5.0 points)

Definition. A list of integers `answers` *satisfies* an equation if there is one integer per placeholder `?` in the equation, and after replacing each `?` with the corresponding integer from `answers`, the expression on the left of `=` has an equal value to the expression on the right of `=`.

Implement `tally` to complete the implementation of `evaluate`. The `evaluate` function takes an `equation` and a list of integers called `answers` that contains one value for each `?` in the `equation`. It returns `True` if `answers` satisfies the `equation` and `False` otherwise.

The `count` method of a list takes a value and returns the number of items in the list that are equal to the value.

```
def evaluate(equation, answers):
    """Return whether the equation is true once each ? is filled by the corresponding answer.

    >>> evaluate([7, '-', 2, '=', '?', '+', 4], [1])       # 7 - 2 == 1 + 4
    True
    >>> evaluate([7, '-', 2, '=', '?', '-', 4], [3])       # 7 - 2 != 3 - 4
    False
    >>> evaluate([7, '-', 2, '=', 1, '+', 4], [])          # 7 - 2 == 1 + 4
    True
    >>> evaluate([7, '-', '?', '=', '?', '-', 4], [2, 9])  # 7 - 2 == 9 - 4
    True
    """
    assert is_equation(equation) and len(answers) == equation.count('?')
    eq_iter, ans_iter = iter(equation), iter(answers)
    return tally(eq_iter, ans_iter) == tally(eq_iter, ans_iter)

def tally(eq, ans):
    """A helper function for evaluate that computes the value of one side of an equation,
       stopping either at an = sign or the end of the equation."""
    number, total = True, 0
    multiplier = 1  # should always be either 1 or -1
    for x in eq:

        if _______:
             (d)
            if x == '?':

                -------
                  (e)
                total += multiplier * x
            else:
                if x == '=':

                    -------
                      (f)
                else:

                    multiplier = _______[x]  # Hint: use a dictionary
                                    (g)
            number = not number

    return total
```

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

- (●) `number`
- ( ) `not number`
- ( ) `isinstance(x, int)`
- ( ) `not isinstance(x, int)`

ii. **(1.0 pt)** Fill in blank (e).

```
x = next(ans)
```

iii. **(1.0 pt)** Fill in blank (f).

- (●) `return total`
- ( ) `return sum(ans)`
- ( ) `return sum(eq)`
- ( ) `total += sum(ans)`
- ( ) `total += sum(eq)`
- ( ) `total += tally(eq, ans)`
- ( ) `total -= tally(eq, ans)`

iv. **(2.0 pt)** Fill in blank (g). **Hint:** Use a dictionary.

```
{'+': 1, '-': -1}
```

(c) (6.0 points)

Implement `feasible`, which takes an `equation` and a range of integers called `allowed`. It returns `True` if it's possible to satisfy the `equation` using numbers from `allowed` and `False` otherwise. In other words, it returns whether it is possible to replace each `?` in the equation with some number from `allowed` such that the expressions to the left and right of `=` are indeed equal. The same number can replace more than one `?`. Assume `evaluate` is implemented correctly. You may call `evaluate`.

```
def feasible(eq, allowed=range(0, 10)):
    """Return whether it's possible to satisfy the equation eq using numbers from allowed.

    >>> feasible([7, '+', 2, '=', '?', '+', 6])    # the ? could be 3
    True
    >>> feasible(['?', '+', 2, '=', '?', '+', 6])  # the first ? could be 7 and the second 3
    True
    >>> feasible([-7, '+', 2, '=', '?', '+', 6])
    False
    >>> feasible(['?', '+', 5, '=', '?', '-', 6])
    False
    >>> feasible(['?', '+', 5, '=', '?', '-', 6], range(-10, 10))  # -5 and 6, for example
    True
    """

    if '?' not in eq:

        return _______
                 (h)
    n = min([i for i in range(len(eq)) if _______])  # index of the first ?
                                            (i)
    return any([_______ for x in allowed])
                  (j)
```

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

```
evaluate(eq, [])
```

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

- ○ `i == '?'`
- ○ `i != '?'`
- ● `eq[i] == '?'`
- ○ `eq[i] != '?'`

iii. (3.0 pt) Fill in blank (j). You may **not** write `for` or `in` or `map` or `range`.

```
feasible(eq[:n] + [x] + eq[n+1:], allowed)
```

(d) (5.0 points)

Implement `solve`, which takes an `equation` and a range of integers called `allowed`. It returns an iterator over lists of integers, and each of these lists *satisfies* the equation. You may call `evaluate` and `feasible`.

```
def print_all(t):
    for x in t:
        print(x)

def solve(equation, allowed=range(0, 10)):
    """Return an iterator over all answer lists that satisfy an equation.

    >>> print_all(solve([7, '+', 2, '=', '?', '+', 6]))
    [3]
    >>> print_all(solve(['?', '-', 2, '=', '?', '+', 6]))
    [8, 0]
    [9, 1]
    >>> print_all(solve(['?', '=', '?', '+', '?'], range(1, 4)))
    [2, 1, 1]
    [3, 1, 2]
    [3, 2, 1]
    """
    def candidates(n):
        """Yield all possible answer lists with n values."""

        if n == 0:

            yield []

        else:

            for first in allowed:

                for rest in _______:
                                (k)
                    yield _______
                            (l)
    blanks = equation.count('?')  # the number of ? in equation

    return _______(_______ , candidates(blanks))
             (m)     (n)
```

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

- ◯ `allowed`
- ◯ `equation`
- ◯ `solve(equation[1:], allowed)`
- ◯ `solve(equation[2:], allowed)`
- ● `candidates(n-1)`
- ◯ `candidates(n-first)`

ii. **(1.0 pt)** Fill in blank (l).

- ○ `first + rest`
- ● `[first] + rest`
- ○ `first + [rest]`
- ○ `[first, rest]`
- ○ `[[first], rest]`
- ○ `[first, [rest]]`

iii. **(1.0 pt)** Fill in blank (m).

- ○ `map`
- ● `filter`
- ○ `any`
- ○ `all`
- ○ `zip`

iv. **(2.0 pt)** Fill in blank (n).

```
lambda a: evaluate(equation, a)
```

4. (10.0 points) Trim the Tree

(a) (5.0 points)

Implement `sums`, which takes a `Tree` of integers `t`. It returns a dictionary in which each node of `t` is a key and the corresponding value is the sum of the labels of the tree rooted at that node.

The `Tree` class appears on the left side of Page 2 of the Midterm 2 study guide.

```
def sums(t):
    """Return a dictionary from nodes to their label sums.

    >>> t = Tree(2, [Tree(4, [Tree(5)]), Tree(3), Tree(5, [Tree(6), Tree(7)])])
    >>> d = sums(t)
    >>> nodes = [t, t.branches[0], t.branches[0].branches[0], t.branches[2]]
    >>> [d[n] for n in nodes]
    [32, 9, 5, 18]
    """

    result = {}

    def insert(t):

        for b in t.branches:

            _______
              (a)
        result[t] = _______
                      (b)
    insert(t)

    return result
```

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

- (●) `insert(b)`
- ( ) `sums(b)`
- ( ) `result += sums(b)`
- ( ) `result[b] = insert(b)`
- ( ) `result[b] += insert(b)`
- ( ) `result[t] += insert(b)`
- ( ) `result[t] += result[b]`
- ( ) `result[b] = sums(t)`
- ( ) `result[b] += sums(t)`

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

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

(b) (5.0 points)

Implement `one_cut`, which takes a `Tree` of integers `t` and a number `n`. It returns a new tree that has the same labels and structure as `t` but with **one** sub-tree excluded. In other words, the returned tree looks like `t` with a non-root node and its descendents removed. Exclude the non-root node that makes the sum of the remaining labels of the tree as close to `n` as possible (in absolute value). Do not mutate `t`. Assume `t` has at least 2 nodes.

```
def one_cut(t, n):
    """Return a tree like t but omit a node so that the sum of remaining labels is close to n.

    >>> t = Tree(2, [Tree(4, [Tree(5)]), Tree(3), Tree(5, [Tree(6), Tree(7)])])  # sums to 32
    >>> one_cut(t, 30)  # cuts just the 3 to get 29
    Tree(2, [Tree(4, [Tree(5)]), Tree(5, [Tree(6), Tree(7)])])
    >>> one_cut(t, 26)  # cuts just the 6 to get 26
    Tree(2, [Tree(4, [Tree(5)]), Tree(3), Tree(5, [Tree(7)])])
    >>> one_cut(t, 15)  # cuts the 5 node with 6 and 7 below it to get 14
    Tree(2, [Tree(4, [Tree(5)]), Tree(3)])
    >>> one_cut(t, 1)   # 14 is the smallest possible sum after one cut
    Tree(2, [Tree(4, [Tree(5)]), Tree(3)])
    """
    d = sums(t)
    total = d.pop(t) # remove the root node from the dictionary; total is the sum of its labels
    target = min(d.keys(), key=lambda node: _______)
                                             (c)
    def cut(t):
        _______
          (d)
    _______
      (e)
```

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

```
abs(n - (total - d[node]))
```

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

- ○ `t.branches.remove(target)`
- ○ `t.branches = [b for b in t.branches if b is not target]`
- ○ `t.branches = [cut(b) for b in t.branches if b is not target]`
- ○ `t = Tree(t.label, [b for b in t.branches if b is not target])`
- ○ `t = Tree(t.label, [cut(b) for b in t.branches if b is not target])`
- ○ `return Tree(t.label, [b for b in t.branches if b is not target])`
- ● `return Tree(t.label, [cut(b) for b in t.branches if b is not target])`

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

- ◯ `return t`
- ● `return cut(t)`
- ◯ `return Tree(t.label, list(map(cut, t.branches)))`
- ◯ `return [cut(t), t][1]`

5. (4.0 points) Downloads

The coroutine function `download` fetches a file with the given name. Its implementation is not shown.

```
async def download(filename: str):
    ...

class Downloader():
    def __init__(self, filenames):
        self.filenames = filenames
        self.current_index = 0

    async def download_files(self):
        while self.current_index < len(self.filenames):
            filename = self.filenames[self.current_index]
            self.current_index += 1
            await download(filename)

    async def download_all(self):
        await asyncio.gather(self.download_files(), self.download_files())

d = Downloader(["a", "b", "c", "d"])
asyncio.run(d.download_all())
```

(a) (3.0 pt) The code above initializes a `Downloader` with a list of 4 files `["a", "b", "c", "d"]`. Which of the following statements are true? Select all that apply.

- [x] The function `download` will be called exactly 4 times.
- [ ] The function `download` may be called more than 4 times.
- [ ] The function `download` may be called fewer than 4 times.
- [x] The function `download` will be called on each file exactly once.
- [ ] The function `download` may not be called on all of the files.
- [ ] The function `download` may be called more than once on the same file.
- [ ] The code above may error because the index `self.current_index` is out of range.

(b) (1.0 pt) Assume that the function `download` takes one second to run, and spends all of that time waiting for the file to be downloaded. During that wait time, it gives up control so that other coroutines can run. How many seconds does the code above take to run?

2 seconds

6. (15.0 points) Scheme VS Python

(a) (5.0 points)

Implement `match`, which takes a linked list of numbers `s` (a `Link` instance or `Link.empty`) and a positive integer `k`. It returns the number of pairs of equal elements that are `k` positions apart in `s`. **Hint: You may use multiple assignment in your solution:** `___ , ___ = ___ , ___`

The `Link` class appears on the left side of Page 2 of the Midterm 2 study guide.

```
def match(s, k):
    """Return how many pairs of values in linked list s that are k positions apart are equal.
    >>> nums = Link(3, Link(1, Link(4, Link(1, Link(5, Link(2, Link(5, Link(3, Link(5)))))))))
    >>> match(nums, 2)  # 1 and 1; 5 and 5; 5 and 5
    3
    >>> {k: match(nums, k) for k in range(1, 8)}  # k=4: 5 and 5; k=7: 3 and 3
    {1: 0, 2: 3, 3: 0, 4: 1, 5: 0, 6: 0, 7: 1}
    """
    t, count = s, 0
    _______:
      (a)
        if t is not Link.empty:
            _______
              (b)
    while t is not Link.empty:
        if _______:
             (c)
            count += 1

        _______
          (d)
    return count
```

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

- ○ `def f(t)`
- ○ `if s is not Link.empty`
- ○ `while s is not Link.empty`
- ● `for x in range(k)`

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

```
t = t.rest
```

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

```
s.first == t.first
```

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

```
s, t = s.rest, t.rest
```

(b) (5.0 points)

Implement `zip-map`, a Scheme procedure that takes a two-argument procedure `f` and lists `s` and `t`. It returns a list containing the results of calling `f` on pairs of values from `s` and `t` that appear in the same position (index) within `s` and `t`. For example, the first value in the returned list is the return value of calling `f` on the first value in `s` and the first value in `t`. The length of the returned list is the minimum of the length of `s` and the length of `t`.

```
;;; Return a list of results from calling f on pairs of co-indexed values from s and t
;;;
;;; scm> (zip-map + '(10 30 20 40) '(5 6 7))
;;; (15 36 27)
;;; scm> (zip-map list '(10 30 20 40) '(9 8 7 6 5 4))
;;; ((10 9) (30 8) (20 7) (40 6))
(define (zip-map f s t)

  (if (or (null? s) (null? t)) nil

    (_______ _______ _______)))
       (e)     (f)     (g)
```

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

- (●) `cons`
- ( ) `list`
- ( ) `append`
- ( ) `zip-map`

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

- ( ) `(car s)`
- ( ) `(car t)`
- ( ) `(cons (car s) (car t))`
- ( ) `(list (car s) (car t))`
- ( ) `(f (cons (car s) (car t)))`
- ( ) `(f (list (car s) (car t)))`
- (●) `(f (car s) (car t))`
- ( ) `(apply f (car s) (car t))`

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

```
(zip-map f (cdr s) (cdr t))
```

(c) (5.0 points)

Implement `match`, a Scheme procedure that takes a list of numbers `s` and a positive integer `k`. It returns the number of pairs of equal elements that are `k` positions apart in `s`. You may call `advance` and `zip-map`.

```
;;; Return the elements of list s starting at index k
(define (advance s k) (if (or (zero? k) (null? s)) s (advance (cdr s) (- k 1))))

;;; Return how many pairs of values in list s which are k positions apart are equal
;;; scm> (define nums '(3 1 4 1 5 2 5 3 5))
;;; nums
;;; scm> (match nums 2)
;;; 3
;;; scm> (map (lambda (k) (list k ': (match nums k))) '(1 2 3 4 5 6 7))
;;; ((1 : 0) (2 : 3) (3 : 0) (4 : 1) (5 : 0) (6 : 0) (7 : 1))
(define (match s k)
  (_______ (_______ (lambda (a b) _______ ) s _______ )))
     (h)      (i)                  (j)         (k)
```

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

- ○ `+`
- ○ `sum`
- ○ `len`
- ○ `cons '+`
- ● `apply +`

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

- ○ `advance`
- ● `zip-map`
- ○ `map`
- ○ `filter`
- ○ `apply`

iii. (2.0 pt) Fill in blank (j). **Hint:** Write an expression that evaluates to a number.

```
(if (= a b) 1 0)
```

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

- ○ `s`
- ○ `k`
- ○ `(cdr s)`
- ● `(advance s k)`

7. (9.0 points) Students

A student has a `name` (a `str`) and a discussion `group` (a `DiscGroup` instance). Each student is placed into a discussion group based on a `keyword` (a `str`) supplied by the student. When a `Student` instance is created, that student is placed into an existing discussion group that has fewer than 6 students and that has the same `keyword` that the student supplied. If the student cannot be placed into any existing group, the student will be placed into a new discussion group. Each discussion group (a `DiscGroup` instance) has a unique integer `id`, starting at 0 for the first discussion group created and counting up, as well as a list of `Student` instances called `students` and a `keyword` (a `str`).

```
class Student():
    """A student has a name and a keyword that is used to place them in a discussion group.
    >>> chris = Student("Chris", "kickstart")
    >>> pranav = Student("Pranav", "tortilla")
    >>> sebastian = Student("Sebastian", "tortilla")
    >>> print(chris)
    Student Chris is in Disc 0: kickstart (['Chris'])
    >>> print(pranav)
    Student Pranav is in Disc 1: tortilla (['Pranav', 'Sebastian'])
    >>> print(sebastian)
    Student Sebastian is in Disc 1: tortilla (['Pranav', 'Sebastian'])
    """
    def __init__(self, name, keyword):
        """Initialize the student and place them in a discussion group."""
        self.name = name
        self.group = _______
                       (a)
        for group in _______:
                       (b)
            if group.keyword == keyword and len(group.students) < 6:
                self.group = _______
                               (c)
        if not self.group:
            self.group = _______
                           (d)

        _______
          (e)

    def __str__(self):
        return f"Student {self.name} is in {self.group}"

class DiscGroup():
    groups = []  # All of the DiscGroup instances

    def __init__(self, keyword):
        self.students = []  # All of the students in this DiscGroup
        self.keyword = keyword
        self.id = _______
                    (f)

        _______
          (g)

    def __str__(self):
        return f"Disc {self.id}: {self.keyword} ({[s.name for s in self.students]})"
```

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

- ● `None`
- ○ `group`
- ○ `DiscGroup(keyword)`
- ○ `DiscGroup(self.keyword)`
- ○ `DiscGroup(self, keyword)`

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

```
DiscGroup.groups
```

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

- ○ `None`
- ● `group`
- ○ `DiscGroup(keyword)`
- ○ `DiscGroup(self.keyword)`
- ○ `DiscGroup(self, keyword)`

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

- ○ `None`
- ○ `group`
- ● `DiscGroup(keyword)`
- ○ `DiscGroup(self.keyword)`
- ○ `DiscGroup(self, keyword)`

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

```
self.group.students.append(self)
```

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

```
len(DiscGroup.groups) or len(self.groups)
```

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

```
DiscGroup.groups.append(self) or self.groups.append(self)
```

8. (6.0 points) Students in SQL

The `students` table has one row per student and columns for their `name` (`string`), the `discussion_id` (`int`) of their discussion, and a `keyword` (`string`) chosen by the student. The keyword was used to place the student in a discussion with other students who selected the same keyword.

The `discussions` table has one row per discussion and columns for the discussion's unique `id` (`int`), the `ta` (`string`) who leads the discussion, and the `room` (`string`) where the discussion is located.

The first few rows in each table are shown below. **Not all rows are shown.**

students:

name	discussion_id	keyword
Chris	2	kickstart
Pranav	0	tortilla
Sebastian	0	tortilla
Lavanya	2	kickstart
Mia	1	unit2
Mahanth	0	tortilla

...

discussions:

id	ta	room
1	Rhys	VLSB203
2	Amr	Evans2
0	Janet	Soda310

...

Example result, part (a)

keyword	total
tortilla	3

Example result, part (b)

ta	id
Rhys	1
Caleb	12

(a) (3.0 points)

Select the most frequently used `keyword`, and the total number of students with that `keyword`. Assume there is one keyword used more than any other (no ties).

```
SELECT keyword, _______ AS total FROM students _______;
                  (a)                            (b)
```

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

- ○ `1`
- ○ `name`
- ● `COUNT(*)`
- ○ `MAX(keyword)`
- ○ `MAX(name)`
- ○ `MAX(COUNT(*))`

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

- ○ `GROUP BY keyword;`
- ○ `GROUP BY keyword HAVING MAX(COUNT(*));`
- ○ `GROUP BY keyword HAVING COUNT(*) = MAX(COUNT(*));`
- ○ `GROUP BY keyword LIMIT MAX(COUNT(*));`
- ○ `GROUP BY keyword ORDER BY keyword DESC LIMIT 1;`
- ● `GROUP BY keyword ORDER BY COUNT(*) DESC LIMIT 1;`
- ○ `AS s1 JOIN students AS s2 ON s1.keyword = s2.keyword LIMIT 1;`
- ○ `ORDER BY COUNT(*) DESC;`
- ○ `ORDER BY keyword DESC;`
- ○ `ORDER BY keyword DESC LIMIT MAX(COUNT(*));`
- ○ `ORDER BY keyword DESC LIMIT 1;`

(b) (3.0 points)

Create a new table `big` with one row for each discussion that has more than 20 students. The columns give the `ta` name and discussion `id` for each of these discussions.

An example `big` result is shown on the previous page. The first row indicates that the discussion with ID `1` taught by `Rhys` has more than 20 students in it (even though only 1 of these students is shown in the example rows of the `students` table).

```
CREATE TABLE big AS
    SELECT ta, id FROM students JOIN discussions _______;
                                                  (c)
```

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

ON discussion_id = id GROUP BY id HAVING COUNT(*) > 20

ii. (0.0 pt) Optional: Draw a picture of you experience as a student in CS 61A.

See the other side of this page for the A+ questions.

9. (0.0 points) Final Boss

These two 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! If your answers are long, write on multiple lines.

(a) (0.0 pt) Definition. An *infix expression* is a list that begins and ends with an integer and alternates between integers and operators, where each operator is either `+` or `-`. For example, `(7 + 2 - 1 + 3 - 6 + 8 - 1 + 4)`.

Fill in the blank to implement `preorder`, which takes an infix expression `expr` and returns a call to `+` on all of the integers from `expr`. Each subtracted number in `expr` should appear within a one-argument call to `-`.

```
;;; Convert an infix expression to a valid Scheme expression with the same value.
;;; scm> (preorder '(7 + 2 - 1 + 3 - 6 + 8 - 1 + 4))
;;; (+ 7 2 (- 1) 3 (- 6) 8 (- 1) 4)
;;; scm> (preorder '(-7 - -1 - 2 - -3))
;;; (+ -7 (- -1) (- 2) (- -3))
(define (preorder expr)
  (define (rest expr f)
    (define (helper s)
      (cond ((null? s) nil)
            ((eq? (car s) '+) (helper (cdr s)))
            ((eq? (car s) '-) (f helper s))
            (else             (cons (car s) (helper (cdr s))))))
    (helper expr))
  _______ )
```

```
(cons '+ (rest expr (lambda (h s) (cons (list '- (car (cdr s))) (h (cdr
(cdr s)))))))
```

(b) (0.0 pt) Implement `two_cut`, which takes a `Tree` of **positive** integers `t` with at least 3 nodes and an integer `n`. It returns a new tree like `t` but with **2** non-root nodes (and their descendents) removed. It's ok if one of the removed nodes is a descendent of the other. Remove the two non-root nodes that change the label sum of the tree to be as close to `n` as possible (in absolute value). Do not mutate `t`. You may call `one_cut` from 4(b) but not `sums`.

```
def g(t):
    return t.label + sum([g(b) for b in t.branches])
def two_cut(t, n):
    """Return a tree like t but omit 2 nodes so that the sum of remaining labels is close to n.
    >>> t = Tree(2, [Tree(4, [Tree(5)]), Tree(3, [Tree(1)]), Tree(5, [Tree(6), Tree(7)])])
    >>> two_cut(t, 30)  # cut 1 and then 3; result sums to 29
    Tree(2, [Tree(4, [Tree(5)]), Tree(5, [Tree(6), Tree(7)])])
    >>> two_cut(t, 12)  # cut 3 and then 5; result sums to 11
    Tree(2, [Tree(4, [Tree(5)])])
    >>> two_cut(t, 6)   # cut 4 and then 5; result sums to 6
    Tree(2, [Tree(3, [Tree(1)])])
    """
    return _______
```

min([one_cut(one_cut(t, k), n) for k in range(g(t))], key=lambda t: abs(n - g(t)))

No more questions.