# CS 61A Spring 2025

# Structure and Interpretation of Computer Programs

MIDTERM 2

**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. (6.0 points) What Would Python Display?**

Assume the following code has been executed. The `pop` method of a list removes and returns its last element.

```
s = [2, [5, 8], 11]
t = [1, 3, 5, 7, 9]

def add_t(x):
    if isinstance(x, list):
        return [add_t(y) for y in x]
    else:
        return x + t.pop()

m = map(add_t, s)
```

Write the value of each expression below or *Error* if an error occurs. Assume that each expression is evaluated in order and sequentially, so evaluating the first could affect the value of the second. Write *Map* for a map object and *Function* for a function.

**(a) (2.0 pt)** `[x[1] for x in s if isinstance(x, list)]`

**(b) (1.0 pt)** `next(m)`

- ◯ 3
- ◯ 5
- ◯ 7
- ◯ 9
- ◯ 11

**(c) (1.0 pt)** `t.pop()`

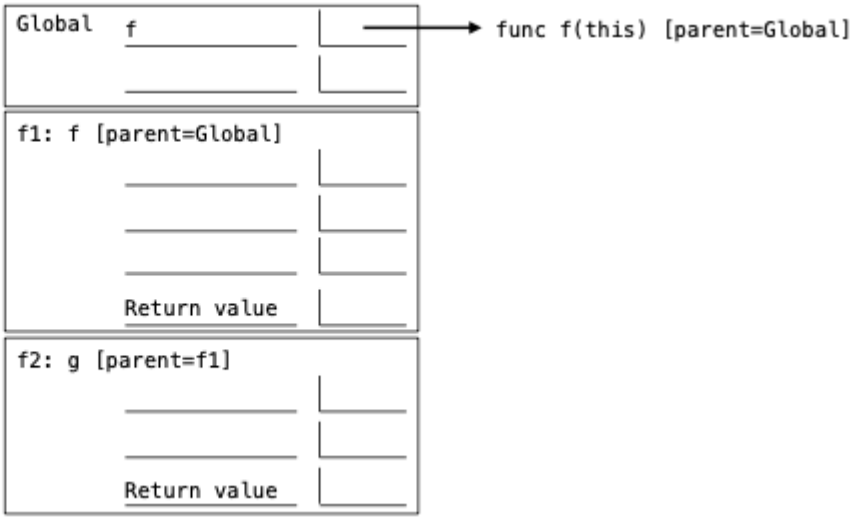
- ◯ 1
- ◯ 3
- ◯ 5
- ◯ 7
- ◯ 9

**(d) (2.0 pt)** `[next(m) for z in range(2)]`

**2. (7.0 points) This or That**

Answer the questions about the code below. Draw an environment diagram, but the diagram itself will not be scored.

```
 1: def f(this):
 2:     that = this[1]
 3:     def g(copy):
 4:         that[0] = 4
 5:         copy[0] = 5
 6:         copy[1].append(6)
 7:         print(copy)
 8:     g(list(this))
 9:     this.append(that)
10:     that.append(7)
11:
12: t = [1, [2], 3]
13: f(t)
14: print(t)
```




**(a) (3.0 pt)** What is displayed by the call to `print` on line 7?

**(b) (3.0 pt)** What is displayed by the call to `print` on line 14?

**(c) (1.0 pt)** What is the order of growth of the time it takes to execute `result = nest(n)` in terms of positive integer `n`? Assume that `append` always takes one step (constant time).

```
def nest(n):
    """Create a nested list.
    >>> nest(3)
    [[[[], []], [[], []]], [[[], []], [[], []]]]
    """
    s = []
    first = s
    for x in range(n):
        next_s = []
        s.append(next_s)
        s.append(next_s)
        s = next_s
    return first
```

◯ exponential

◯ quadratic

◯ linear

◯ logarithmic

◯ constant

**3. (10.0 points) Berkeley Time**

**(a) (5.0 points)**

An `Event` instance has `start` and `end` attributes that are positive integers representing the number of minutes after midnight that the `Event` begins and concludes.

Implement the `Event` class. The `split` method takes an integer `k` that **evenly divides** the difference between the `end` and `start` times. It is a generator function that yields `k` sequential events of equal length that together span the same time interval as the original event. Calling `split` **does not** change the event.

```
class Event:
    """An Event has a start and end time and can be split into equal-length shorter events.
    >>> calapalooza = Event(980, 1180)
    >>> list(calapalooza.split(4))  # Split it into 4 smaller events that go from 980 to 1180.
    [Event(980, 1030), Event(1030, 1080), Event(1080, 1130), Event(1130, 1180)]
    """
    def __init__(self, start, end):
        self.start, self.end = start, end
    def __repr__(self):
        return f'Event({self.start}, {self.end})'
    def split(self, k):
        assert (self.end - self.start) % k == 0  # k evenly divides the duration of the event.
        length = (self.end - self.start) // k

        start = _______
                  (a)
        for i in range(k):

            yield _______
                    (b)

            _______
              (c)
```

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

- ◯ `0`
- ◯ `None`
- ◯ `self`
- ◯ `start`
- ◯ `self.start`

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

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

- ◯ `start = start + length`
- ◯ `start = self.start + length`
- ◯ `self.start = start + length`
- ◯ `self.start = self.start + length`

**(b) (5.0 points)**

A `BerkeleyEvent` is an `Event` with a modified `split` method. The first `Event` yielded by `split` starts at the same time as the original event, but all events after the first one are 10 minutes shorter and start 10 minutes after the previous event ends. Again, assume `k` evenly divides the difference between the `end` and `start` attributes.

```
class BerkeleyEvent(Event):
    """When a BerkeleyEvent is split, there is time between events.
    >>> list(BerkeleyEvent(840, 930).split(3))
    [Event(840, 870), Event(880, 900), Event(910, 930)]
    """
    time = 10  # How much time between events
    def split(self, k):
        first = True
        for e in _______:
                    (d)
            if first:
                first = False
            else:
                _______.start = _______ + _______
                  (e)             (f)       (g)
            yield e
```

**i. (2.0 pt)** Fill in blank (d).

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

- ◯ `e`
- ◯ `self`
- ◯ `self.e`

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

- ◯ `self`
- ◯ `self.start`
- ◯ `self.end`
- ◯ `e`
- ◯ `e.start`
- ◯ `e.end`

**iv. (1.0 pt)** Fill in blank (g). **Select all that apply.**

- ☐ `time`
- ☐ `self.time`
- ☐ `self.BerkeleyEvent.time`
- ☐ `BerkeleyEvent.time`
- ☐ `Event.time`

**4. (11.0 points) DeepSeek**

**Definition.** The *depth* of the root node of a tree is 0. The depth of each other node is one plus the depth of its parent.

**(a) (5.0 points)**

Implement `deepest`, which takes a `Tree` instance `t` and a value `x`. It returns the largest depth of a node labeled `x` in `t`. If there are no nodes labeled `x` in `t`, then `deepest(t, x)` returns `None`.

The `Tree` class appears on the Midterm 2 study guide (page 2, left column).

```
def deepest(t, x):
    """Return the maximum depth of a node labeled x in Tree t.
    Return None if x is not a label in t.

    >>> example = Tree(30, [Tree(20), Tree(20, [Tree(30, [Tree(20)])])])
    >>> deepest(example, 30)
    2
    >>> deepest(example, 20)
    3
    >>> print(deepest(example, 10))
    None
    """
    below = [deepest(b, x) for b in t.branches]
    depths = [ _______ ]
                 (a)
    if _______:
         (b)
        depths._______
                 (c)
    if depths:
        return max(depths)
    else:
        return None
```

**i. (3.0 pt)** Fill in blank (a). Include `for` and `in` and `if` in your answer.

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

- ◯ `x in depths`
- ◯ `depths`
- ◯ `not depths`
- ◯ `t == x`
- ◯ `t.label == x`
- ◯ `t.is_leaf()`

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

**(b) (6.0 points)**

Implement `deep`, which takes a `Tree` instance `t`. It returns a dictionary containing each unique node label of `t` as a key. The value for each key `k` is the largest depth of any node in `t` labeled `k`.

**IMPORTANT: You may not call `deepest` in your implementation.**

**Hint:** For a dictionary `d`, `d.get(1, 2)` returns the value for the key `1` if that key appears in `d` and 2 otherwise.

```
def deep(t):
    """Return a dictionary containing each unique label of t as a key
    and the maximum depth of a node labeled with that key as its value.

    >>> deep(Tree(30, [Tree(20), Tree(20, [Tree(30, [Tree(20)])])]))
    {30: 2, 20: 3}
    """
    result = {}
    def visit(node, depth):
        _______ = max(_______, result.get(_______, 0))
          (d)           (e)                 (f)
        for b in node.branches:
            -------
              (g)
    visit(t, 0)
    return result
```

**i. (2.0 pt)** Fill in blank (d).

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

- ◯ 0
- ◯ 1
- ◯ `depth`
- ◯ `t.label`
- ◯ `node.label`

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

- ◯ `depth`
- ◯ `t`
- ◯ `t.label`
- ◯ `node`
- ◯ `node.label`

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

**5. (16.0 points) Just Add and Multiply**

**Definition.** A plus-times-expression for a list (or linked list) of numbers inserts either `+` or `*` between each adjacent pair of numbers. For example, `2*3+4+5` is one possible plus-times-expression for `[2, 3, 4, 5]`.

**(a) (5.0 points)**

Implement `muladd`, which takes a **linked list** of numbers `s`. It returns the value of the plus-times-expression that starts with `*` and then alternates between `+` and `*`. If `s` has only one element, `muladd` returns that element.

The `Link` class is defined on the left column of page 2 of the Midterm 2 Study Guide.

```
def muladd(s):
    """Combine the numbers in linked list s by alternately multiplying and adding.

    >>> example = Link(9, Link(4, Link(7, Link(2, Link(0)))))  # <9 4 7 2 0>
    >>> muladd(example)           # 9 * 4 + 7 * 2 + 0, not 9 * (4 + 7 * (2 + 0))
    50
    >>> muladd(Link(2, example))  # 2 * 9 + 4 * 7 + 2 * 0
    46
    >>> muladd(Link(2))
    2
    """
    if s is Link.empty:

        return 0

    elif s.rest is Link.empty:

        return _______
                 (a)
    else:

        return _______
                 (b)
```

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

- ◯ `0`
- ◯ `1`
- ◯ `s.first`
- ◯ `s.rest`
- ◯ `s.rest.first`

**ii. (4.0 pt)** Fill in blank (b). If your answer is too long, you can continue on another line.

**(b) (5.0 points)**

Implement `ways`, which takes positive integers `k` and `n`. It returns the number of possible plus-times-expressions for a list of length `n` that have at most `k` `*`-symbols in a row. For example, `2+3*4*5*6+7*8*9+10+11*12*13*14+15*16` has 3 in a row `3*4*5*6`, then 2 in a row `7*8*9`, then 3 in a row `11*12*13*14`, then 1 in a row `15*16`.

```
def ways(k, n):
    """Return the number of plus-times-expressions with at most k consecutive *'s for n numbers.

    >>> ways(1, 4) # For [2, 3, 4, 5], these 5 ways: 2+3+4+5, 2+3+4*5, 2+3*4+5, 2*3+4+5, 2*3+4*5
    5
    >>> ways(2, 4) # For [2, 3, 4, 5], the 7 ways would also include 2+3*4*5 and 2*3*4+5
    7
    >>> ways(2, 5) # Some examples for [2, 3, 4, 5, 6]: 2+3*4+5*6 & 2*3*4+5*6 (but not 2+3*4*5*6)
    13
    """
    def f(left, n):

        if _______:
             (c)

            return 1

        result = f( _______ , n - 1)
                      (d)
        if left:

            result += _______
                        (e)
        return result

    return f(k, n)
```

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

- ◯ `left == 0`
- ◯ `left == 1`
- ◯ `n == 0`
- ◯ `n == 1`

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

- ◯ `left`
- ◯ `left - 1`
- ◯ `k`
- ◯ `k - 1`

**iii. (3.0 pt)** Fill in blank (e).

**(c) (6.0 points)**

Implement `close`, which takes a list of numbers `s` and a number `x`. It returns the number closest (in absolute value) to `x` that is the value of some plus-times-expression for `s`.

**Hint:** A slice that starts past the end of a list is empty. For example, `[2, 3, 4][3:]` evalautes to `[]`.

The `product` function used to implement `close` is defined below.

```
def close(s, x):
    """Return the value of a plus-times-expression for s that is closest to x.

    >>> print(close([1, 2, 3, 4], 10))  # 1 + 2 + 3 + 4 = 10
    10
    >>> print(close([1, 2, 3, 4], 20))  # 1 * 2 * 3 * 4 = 24 (4 away from 20)
    24
    >>> print(close([1, 2, 3, 4], 30))  # 1 + 2 * 3 * 4 = 25 (5 away from 30)
    25
    >>> print(close([3, 7, 5, 4, 2, 6], 66))  # 3 * 7 + 5 * 4 * 2 + 6 = 67 (1 away from 66)
    67
    """
    if not s:
        return 0
    choices = [product(s[:i]) + _______ for i in range(1, len(s) + 1)]
                                  (f)

    return min(choices, key=_______)
                             (g)

def product(s):
    """Return the result of multiplying together the elements of a non-empty list of numbers s."""
    if len(s) == 1:
        return s[0]
    return s[0] * product(s[1:])
```

**i. (4.0 pt)** Fill in blank (f).

**ii. (2.0 pt)** Fill in blank (g). **Select all that apply.** Assume `sub` has been imported from the `operator` module. The `sub` function takes two arguments and subtracts the second from the first.

- ☐ `abs`
- ☐ `sub`
- ☐ `abs(sub)`
- ☐ `sub(abs)`
- ☐ `lambda y: abs(x - y)`
- ☐ `lambda x, y: abs(x - y)`

**(d) (0.0 points)**

**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 `close_exp`, which takes a list of numbers `s` and a number `x`. It returns a string containing a plus-times-expression for `s` that evaluates to a number that is as close (in absolute value) as possible to `x`. (If there is more than one such expression, return any of them.)

The built-in `eval` function takes a string containing an expression and returns its value. For example, `eval('2+2')` evaluates to 4.

```
def close_exp(s, x):
    """Return the plus-times-expression of s that has a value closest to x.

    >>> print(close_exp([1, 2, 3, 4], 10))  # 1 + 2 + 3 + 4 = 10
    1+2+3+4
    >>> print(close_exp([1, 2, 3, 4], 20))  # 1 * 2 * 3 * 4 = 24 (4 away from 20)
    1*2*3*4
    >>> print(close_exp([1, 2, 3, 4], 30))  # 1 + 2 * 3 * 4 = 25 (5 away from 30)
    1+2*3*4
    >>> print(close_exp([3, 7, 5, 4, 2, 6], 66))  # 3 * 7 + 5 * 4 * 2 + 6 = 67 (1 away from 66)
    3*7+5*4*2+6
    """
    goal = close(s, x)
    term = ''
    for i in range(len(s)):
        if term:
            term += '*'
        term += str(s[i])
        candidate = term
        if i < len(s) - 1:
            candidate += ____________________________
        if eval(candidate) == goal:
            return candidate
```

**i. (0.0 pt)** Fill in the blank. If your answer is too long, you can continue on another line.

**No more questions.**