Student Name: Emily Yeh
Check one:
[ ] I completed this assignment without assistance or external resources.
[ ] I completed this assignment with assistance from ___
and/or using these external resources: ___

Question 1:
Part A:

1. ∃x. ∀y. loves(x, y): Since this means “there exists one or more elements (x) that love all the other elements (column y),” I think Table 1 is NOT a model because there is no element in column x (a, b, c, or d) that loves all the elements in column y (a, b, c, and d).
2. ∃y. ∀x. loves(x, y): Since this means “there is some element (y) that is loved by all the other elements (column x),” I think Table 1 is a model because there is one element, c, that is loved by all the other elements in column x (a, b, c, and d).
3. ∀x. ∃y. loves(x, y): Since this means “all the elements (column x) love some element (y)”, I think that Table 1 is a model because every element (a, b, c, and d) loves at least one element (a, b, c, or d).
4. ∀y. ∃x. loves(x, y): Since this means “all elements (column y) are loved by some element (x),” I think that Table 1 is a model because every element (a, b, c, and d) is loved by at least one element (a, b, c, or d).
5. ∃x. ∃y. loves(x, y): Since this means “all elements (column x) love all elements (column y),” I think that Table 1 is NOT a model because not every element (a, b, c, and d) love all elements (a, b, c, and d).
6. ∃x. ∀y. ¬loves(x, y): Since this means “there exists some element (x) that does NOT love all elements (column y),” I think that Table 1 is a model because there definitely does exist several elements (a, b, c, or d) that do not love all the elements in column y (a, b, c, and d).
7. ∃x. ¬∀y. loves(x, y): Since this means “there exists some element (x) that loves NOT all elements (column y),” I think that Table 1 is a model because there definitely are some elements (a, b, c, or d) that do not love all elements (a, b, c, and d).

Part B:

Statement	Table 1	x >= y	x = y
∀x, x. loves(x, x)	False	True	True
∀x, y. loves(x, y) → loves(y, x)	False	False	True?
∀x, y, z. loves(x, y) ∧ loves(y, z) → loves(x, z)	False	True	True?

Question 2:

1. Prove: ∃x. sum(x, x, s(s(0)))

1. ∀x. sum(x, 0, x)	Axiom 1
2. sum(s(0), 0, s(0))	1, x→s(0)
3. ∀x. sum(x, y, z) → sum(x, S(y), S(z))	Axiom 2
4. sum(s(0), 0, s(0)) → sum(s(0), s(0), s(s(0)))	2, 3, y→0, z→s(0)
5. sum(s(0), s(0), s(s(0)))	4, 2, Modus Ponens
6. ∃x. sum(x, x, s(s(0)))	Existential Instantiation, 5, s(0)→x (I don't know how valid this logic actually is… I wasn't sure how else to reintroduce x as s(0) or how to add the ∃x part.)

2. Prove: ∀x. sum(0, x, x)

1. ∀x. sum(x, 0, x)	Axiom 1
2. sum(0, 0, 0)	1, x→0
3. ∀x. sum(x, y, z) → sum(x, S(y), S(z))	Axiom 2
4. sum(0, s(-1), s(-1)) → sum(0, 0, 0)	3, y→s(-1), z→s(-1)
5. sum(0, s(-1), s(-1))	4, 2, Modus Ponens
6. ∀x. sum(0, x, x)	Universal Instantiation, 5, s(-1)→x (again, I don't know how valid this logic actually is… partially because I don't remember how induction works. I've already spent about an hour on this problem, though, and I'm not getting anywhere, so I'll probably ask an instructor for some help with this topic after class on Monday.

Question 3:

Part A:

Look to your right → → →

Part B:

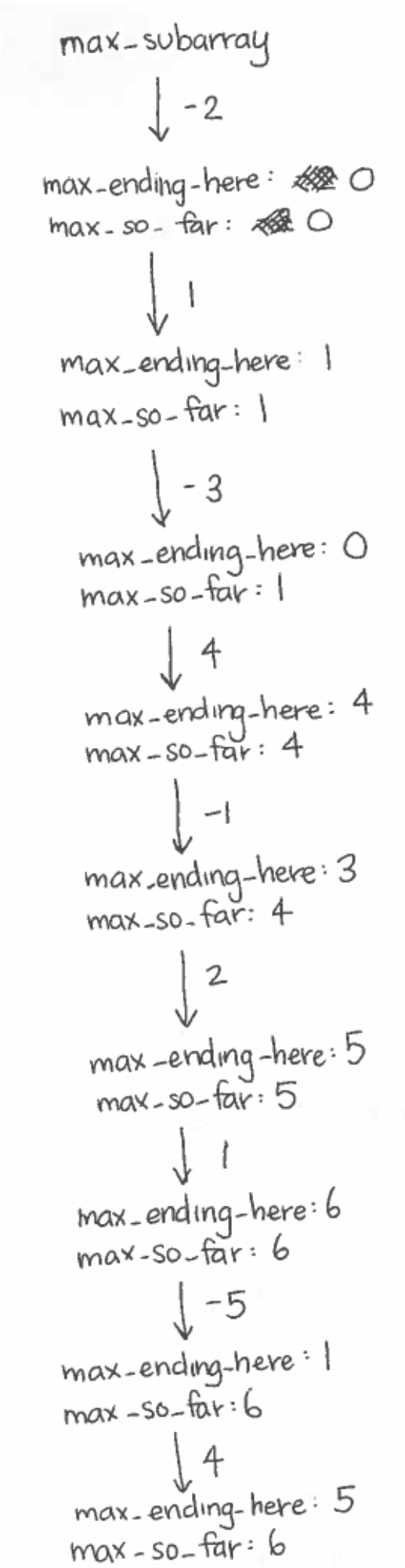
I think that max_subarray is not an example of divide and conquer because it doesn't really divide the problem into multiple smaller problems, and it doesn't return a result of any kind of "dividing" and "conquering." A divide and conquer algorithm for this problem would probably divide the array into three parts, because the maximum sum subarray would either be contained entirely in the first half, the last half, or in some overlap of both halves. I could be completely wrong, though. I just thought that the divide and conquer algorithm requires that the problem be divided into various subproblems, then those subproblems should be solved through recursion (or possibly other methods), and the results of these subproblems should be combined to form a solution for the original problem.

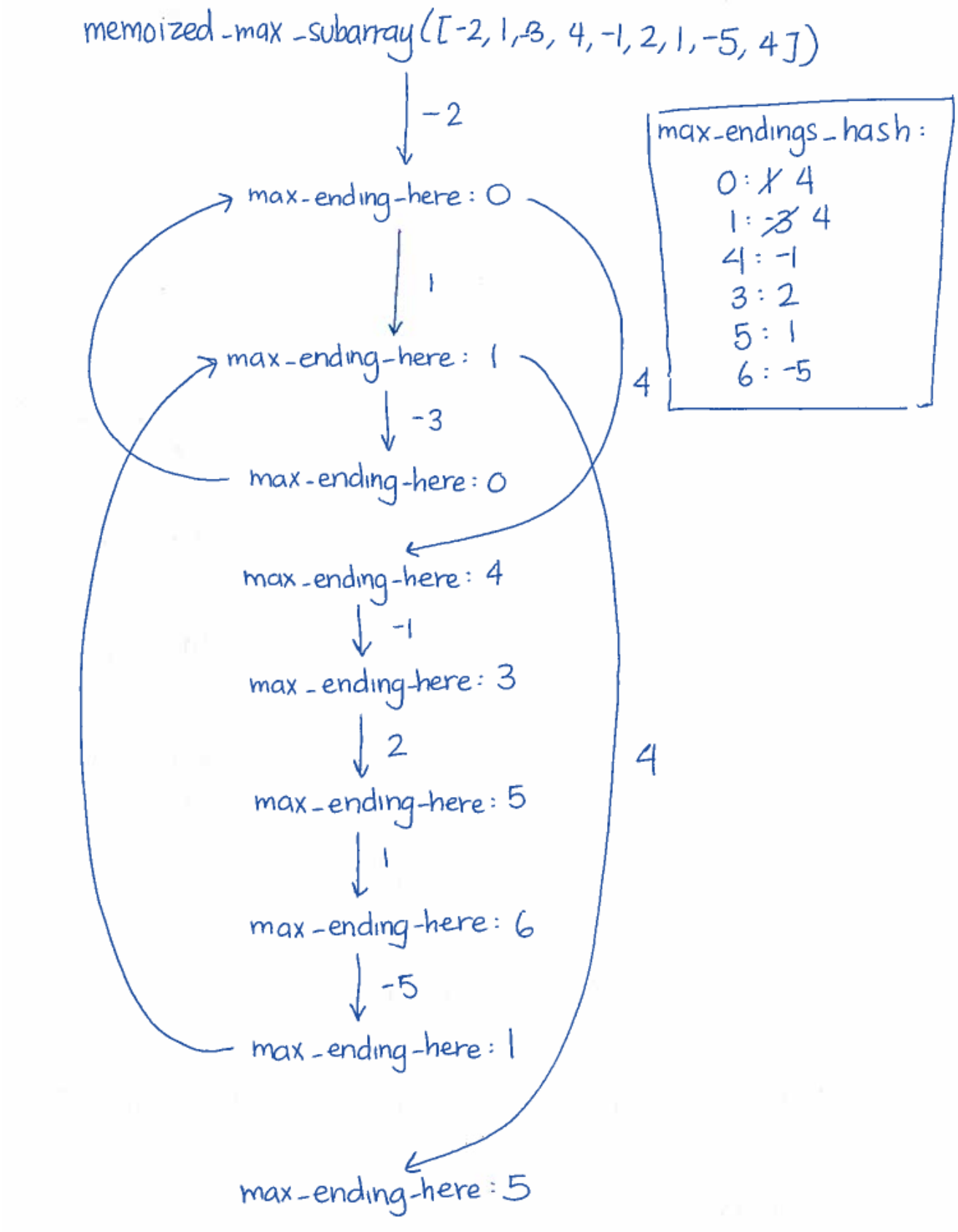
Part C:

I think that max_subarray is an example of dynamic programming. According to Wikipedia, dynamic programming "is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions – ideally, using a memory-based data structure. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution." This is certainly true in this case; max_subarray keeps track of the max sum so far in one variable, and the max ending at any given array element in another variable, looking these values up every time it checks a new array element.

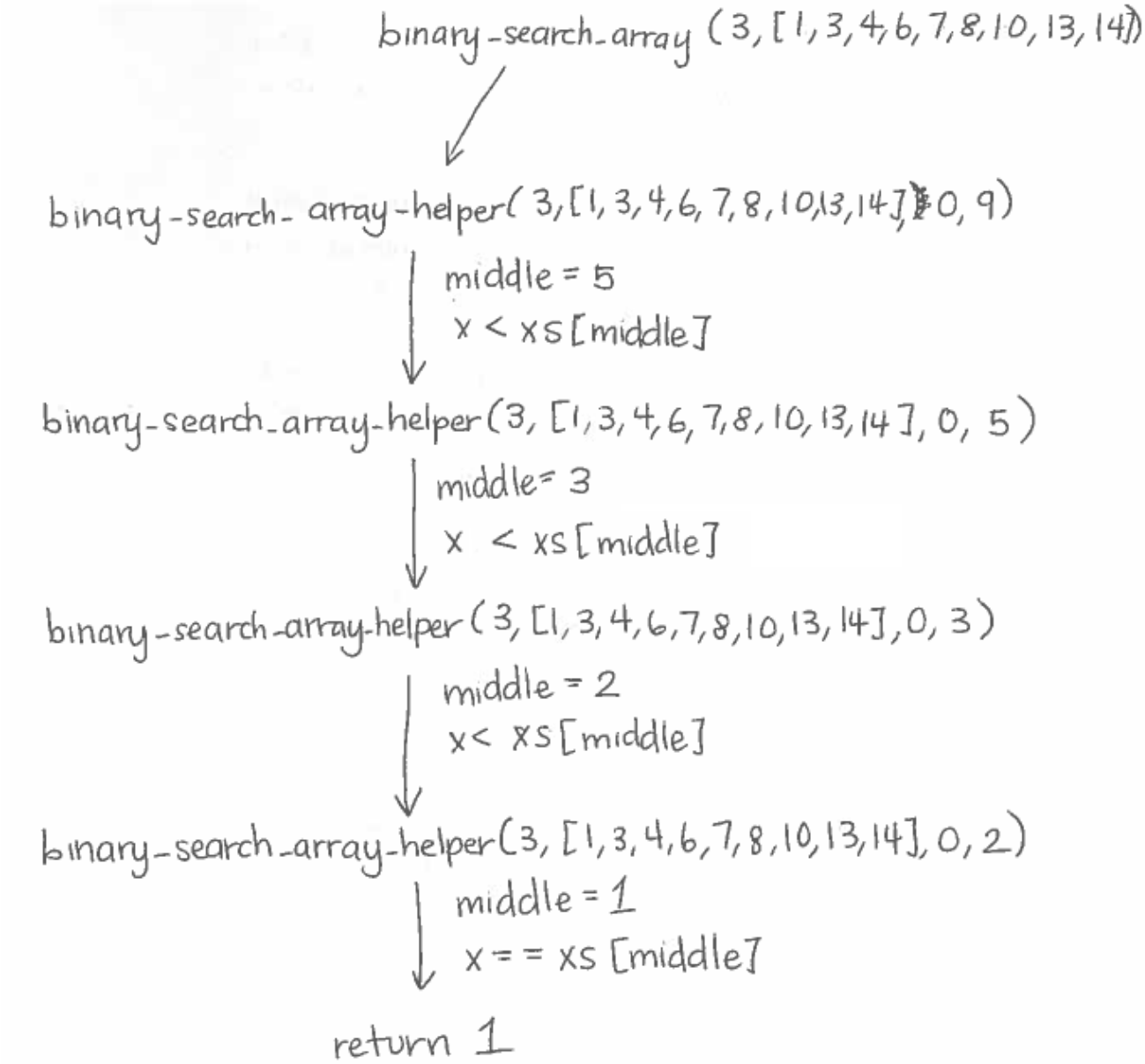
Part D:

I'm not actually sure what a memoized version of max_subarray would look like. What results would I store in the hash table? I think I should keep track of the max_ending_here at every point as keys. Then, the values would be the next input. The function would then only proceed to calculate the next max_ending_here if the next input value was greater than the previous input value in the hash table; it wouldn't bother calculating the max_ending_here if the next input value was lower. (Actually, I can think of several things that might go wrong with this system, but I can't really think of any other way to memoize this function…)

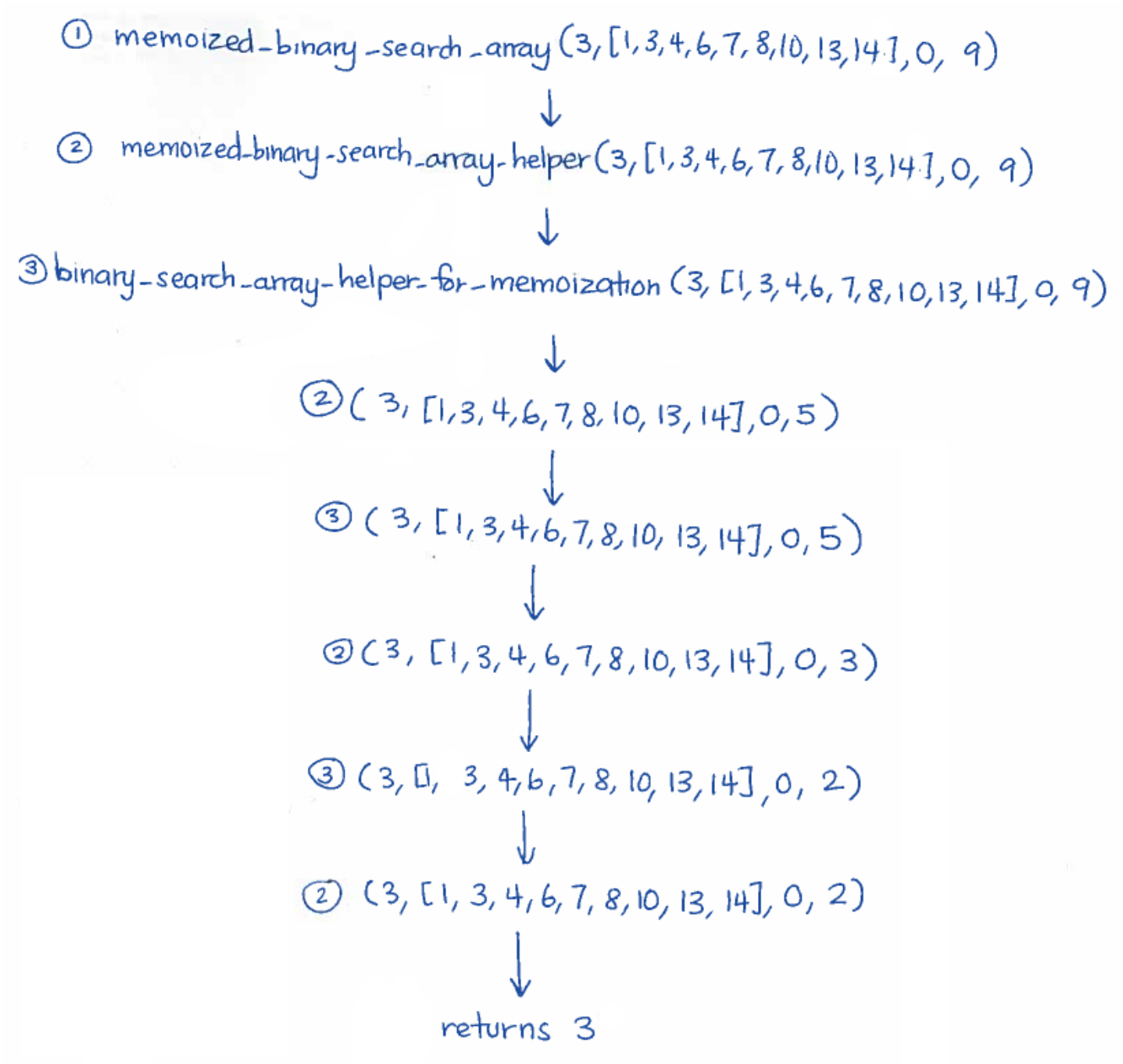



memoized_max_subarray([-2, 1, -3, 4, -1, 2, 1, -5, 4])
-2
max_ending_here: 0
1
max_ending_here: 1
-3
max_ending_here: 0
max_ending_here: 4
-1
max_ending_here: 3
2
max_ending_here: 5
1
max_ending_here: 6
-5
max_ending_here: 1
max_ending_here: 5
max_endings_hash:
0: ✗ 4
1: ~~-3~~ 4
4: -1
3: 2
5: 1
6: -5
4
4

Question 4:
Part A:



Part B:
I think that binary_search_array is an example of divide and conquer because it does split the problem into smaller and more easily solved subproblems; it continuously breaks the given array into halves until it finds the element it is searching for (or returns None if it can't find it).
Part C:
I think that binary_search_array is not an example of dynamic programming because while it does split the problem into smaller parts, there are no calculations being performed on each of those parts, and therefore, there are no values being stored and there are no values to look up for subsequent calculations.
Part D:
(Note: I refer to memoized_binary_search_array as function (1), its helper as (2), and that helper's helper as (3) because I really didn't want to write out their full names so many times.)



You know, I don't believe memoizing the binary search array function actually did anything for the given input. Interesting.

Part E:

At first, I thought that there might not actually be any benefits to memoizing binary searching. However, it might be beneficial in the special case of when the input array has repeating elements, or where the input array has multiples of the same element.

Part F:

If the input array is not pre-sorted, won't the binary_search_array function break? For example, finding the 3 in a small unsorted array like [1, 3, 5, 2, 4] will probably not work. This bug can easily be fixed by sorting the array, of course. This won't even be an expensive process, if the array is small. (I feel like this answer was too simple, though, so it must be wrong. Where is the bug and what input will cause the function to break?)