Problem Set 1—Algorithmic Strategies

📝 Important notes:

- Watch this video recorded by Joram Erbarth (M23) with advice on preparing for CS110 assignments. Most suggestions will also apply to other CS courses, so please make sure to bookmark this video for future reference. The video refers to submitting primary and secondary resources, but for this specific assignment, you'll answer questions directly in the notebook.
- Make sure to include your work whenever you see the labels `YOUR DOCSTRING HERE` or `###YOUR CODE HERE`. You can use several code cells per question throughout the notebook, but you need not use them all. They are just there if you need them and for us to test your work once you submit it.
- Please use the function names and positional arguments provided in the code cells. If you need extra parameters or functions, you can just define auxiliary ones called by the primary functions whose skeleton code we provide in the workbook.
- Each problem is intentionally divided into several questions that assist you in building up your work. Not all questions require the same time commitment, so please read the assignment in its entirety before starting and plan your time accordingly. Some questions need you to reflect on your work, so keep a record of your thoughts and processes as you work through this problem set.
- Please refer to the CS110 course guide on how to submit your assignment materials.

If you have any questions, please don't hesitate to reach out to the TAs through the proper channels or come to OHs. Good luck!

Question 1 of 10

🏁 Setting up:

Start by stating your name and identifying your collaborators. Please ellaborate on the nature of the collaboration (for example, if you briefly discussed the strategy to solve problem 1, say so, and explicitly point out what you consulted). Indicate how you might have, specifically, used AI, and acknowledge that this work is an accurate description of your own learning and working knowledge of the course LOs. Remember that while you may use collaborative discussions to help clarify concepts, your final submission should accurately represent your individual understanding and problem-solving skills.

Here is a simplified example:

```
Name: Ahmed Souza
Collaborators: Lily Shakespeare and Anitha Holmes
Details: I discussed the iterative strategy of problem 1 with Lily, specifically why the coding
solution that I tried with AI didn't pass the first code test. After Anitha joined our discussion,
it turns out that it was because of ... I acknowledge that all of the work included in this workbook
is my own, and I have not shared (or received) any working solutions with (or from) anyone, not even
an AI engine or tool.
```

Review the AI policy for this course in the CS110 course guide.

Normal

I acknowledge that all of the work included in this workbook is my own, and I have not shared (or received) any working solutions with (or from) anyone. I only used Grammarly AI to avoid grammatical mistakes and typos.

Question 2 of 10

Reflect on your learning trajectory so far. As Prof Gordon Stobart puts it:

> "Feedback is information that closes the gap between where somebody is and where they need to get to"

Check all the formative feedback you have received so far, specifically on #cs110-AlgoStratDataStruct, #cs110-ComplexityAnalysis, and #cs110-ComputationalCritique.

- Pick one of these, copy and paste it below for reference, between quotes, and with reference to the session date.
- Explain how you plan to use that feedback in this first assignment to support your learning further.

Normal

> "Correct! The recursive implementation was indeed O(n^2). To go further in your response, please try to provide explanation/justification for your response. Why is the recursive implementation O(n^2)?"

#cs110-ComplexityAnalysis - CS110 Session 7 - [4.1] Dividing and conquering the maximum subarray problem

This feedback was particularly useful to make me acknowledge with how much depth I should be explaining how I derived a particular asymptotic notation. In this assignment, I plan to use this feedback to calculate time complexities in far more detail in terms of its theoretical basis, with the support of concepts from Cormen et al.'s 'Algorithms'.

Problem 1

You are sitting at SFO to head to your next rotation city, and the boarding gate at the airport is connected to the main floor through a flight of stairs. The boarding process starts two hours from now, and you decide to make the most of your time by playing with algorithms while you wait—because why not?!

Your journey involves descending n steps to reach the boarding gate. With each step, you can take a single step or a confident (yet safe) leap of two steps. Your task is to determine how many ways you can make it to the boarding gate.

Clearly, there are a few limit cases:

```
Input: n = 0
Output: 0
Explanation: There is no staircase to descend!

Input: n = 1
Output: 1
Explanation: There is one way to descend.
1. 1 step

Input: n = 2
Output: 2
Explanation: There are two ways to descend.
1. 1 step + 1 step
2. 2 steps (in one go!)
```

Question 3 of 10

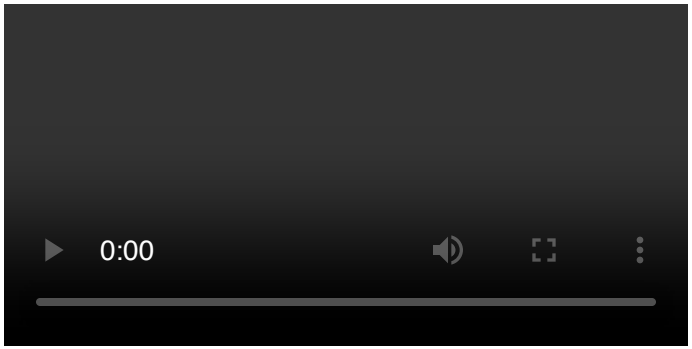
A. Upload a single video where you clearly explain how you would design two approaches (an iterative and a recursive one) to solve this computational problem.

- Avoid using technical jargon (such as "indices" and while/for loops), and focus on explaining in as simple terms as possible how the algorithm works and how it is guaranteed, upon termination, to return the correct answer.
- Your explanation should clarify how your solutions specifically follow the iterative and recursive paradigms.
- Show and Tell Tip: try following the algorithm steps in real life to find how many possible solutions exist in a flight of *at least* three stairs (whichever number you can find in the residence hall or anywhere else; we strongly advise you to find a relatively short staircase since you're just using it to deconstruct the problem). Be careful and have fun!

Notes:

- As a reference, shorter explanations have a higher potential to be more targeted and, hence, better.
- Use any resources you deem helpful for your explanation—you may wish to record, for example, a whiteboard where you write scribblings or a flowchart where you make annotations; you may also prefer to use your "show and tell" run to explain key concepts. Your face must be visible at all times during the recording.
- The recording cannot last for more than 4 minutes. Any content beyond this threshold will be ignored. Use this time constraint to supercharge your explanation—focus on **how** the algorithms work and **why** they are guaranteed to work.
- We recommend you follow the instructions on MyMinerva to register your Minerva account on Loom; use this software to record your video. You can use alternative software if you wish, as long as you comply with the requirements listed here.

recursive_iterative_explanations.mp4 (11.2 MB) ×



Question 4 of 10

B. Provide below both recursive and iterative Python implementations to the algorithms you have described above, using the function

Remember to add at least three test cases to demonstrate that your function is correctly implemented in Python. Use the text cell provided below to explain why your tests are representative and provide strong reassurance that the implementation is indeed correct (i.e., that the tests are sufficient).

Normal

My test cases, elaborated under the unittest library, are sufficient because they encompass edge cases in terms of 0, 1, and 2 steps, which indicate an important distinction from the solution to this problem and the original Fibonacci sequence. Furthermore, I also included a randomized test against the original Fibonacci equation (with the input shifted by n+1 as expected) within a reasonable range (1-30) which doesn't reach the recursion depth of the notebook. The latter test is also especially important to support the robustness of my algorithm in handling different sizes of inputs correctly.

Code Cell 1 of 19

In [29]

```python
def descending_stairs_iterative(n):
    '''
    This function calculates the number of ways to descend a staircase of n steps towards the boarding gate.

    Args:
    ------------
    n (int): The number of steps towards the boarding gate. Must be non-negative.

    Returns:
    ------------
    number_of_ways (int): The number of ways to descend a staircase of n steps.

    '''
    if n < 0:
        raise ValueError("The number of stairs must be positive")

    if n == 0:
        return 0
```

```
19
20      stairs = [0] * (n + 1)
21      stairs[0] = 1
22      stairs[1] = 1
23
24      for i in range(2, n+1):
25          stairs[i] = stairs[i - 1] + stairs[i - 2] # store the number of combinations cumulatively in a list
26
27      number_of_ways = stairs[n]
28      return number_of_ways
29
30  # version 1, without memoization
31  def descending_stairs_recursive(n):
32      '''
33      This function calculates the number of ways to descend a staircase of n steps towards the boarding gate, through
    a recursive approach.
34
35      Args:
36      ------------
37      n (int): The number of steps towards the boarding gate. Must be non-negative.
38
39      Returns:
40      ------------
41      number_of_ways (int): The number of ways to descend a staircase of n steps
42
43      '''
44
45      if n == 2:
46          return 2
47      elif n == 1:
48          # adding the last step before finishing the staircase
49          return 1
50      elif n <= 0:
51          return 0
```

```
52
53      return descending_stairs_recursive(n - 1) + descending_stairs_recursive(n - 2)
54
55  # version 2, without memoization
56  def descending_stairs_recursive_memoized(n, memo=None):
57      '''
58      This function calculates the number of ways to descend a staircase of n steps towards the boarding gate through
    a recursive, memoized approach.
59
60      Args:
61      ------------
62      n (int): The number of steps towards the boarding gate. Must be non-negative.
63      memo (dict, optional): A memoization dictionary to store computed results. Defaults to None.
64
65      Returns:
66      ------------
67      int: The number of distinct ways to descend the staircase.
68
69      '''
70
71      if memo is None:
72          memo = {} # introducing the cache to store return values
73
74      if n in memo:
75          return memo[n] # returning the cache value if 'n' is already stored
76
77      if n == 2:
78          return 2
79      elif n == 1:
80          # adding the last step before finishing the staircase
81          return 1
82      elif n <= 0:
83          return 0
```

```
    memo[n] = descending_stairs_recursive_memoized(n - 1, memo) + descending_stairs_recursive_memoized(n - 2, memo)
# storing the cache value

    # uncomment the line below to visualize the use of the cache
    # print(memo)

    return memo[n]
```

Run Code

Code Cell 2 of 19

In [30]

```
import unittest
import random
import math

class TestProblem1(unittest.TestCase):

    def test_randomized_input(self):
        # Compare the outputs of the staircase function against the results of the fibonacci sequence formula,
shifted by n+1

        n = 5 #random.randint(1, 30) # pick a random number between 1-30
        print(f'Selected Input: {n}')

        # Actual Fibonacci formula
        phi = (1 + math.sqrt(5)) / 2
        psi = (1 - math.sqrt(5)) / 2

        expected = round((phi**(n+1) - psi**(n+1)) / math.sqrt(5))

```

```
        print(f'Recursive approach: {descending_stairs_recursive(n)}')
        print(f'Recursive (optimized) approach: {descending_stairs_recursive_memoized(n)}\n')

        self.assertEqual(descending_stairs_iterative(n), expected)
        self.assertEqual(descending_stairs_recursive(n), expected)
        self.assertEqual(descending_stairs_recursive_memoized(n), expected)


    def test_no_steps(self): # There are no steps to take
        n = 0
        expected = 0

        print(f'Iterative approach: {descending_stairs_iterative(n)}')
        print(f'Recursive approach: {descending_stairs_recursive(n)}')
        print(f'Recursive (optimized) approach: {descending_stairs_recursive_memoized(n)}\n')

        self.assertEqual(descending_stairs_iterative(n), expected)
        self.assertEqual(descending_stairs_recursive(n), expected)
        self.assertEqual(descending_stairs_recursive_memoized(n), expected)

    def test_one_step(self): # There is only one step left
        n = 1
        expected = 1

        print(f'Iterative approach: {descending_stairs_iterative(n)}')
        print(f'Recursive approach: {descending_stairs_recursive(n)}')
        print(f'Recursive (optimized) approach: {descending_stairs_recursive_memoized(n)}\n')

        self.assertEqual(descending_stairs_iterative(n), expected)
        self.assertEqual(descending_stairs_recursive(n), expected)
        self.assertEqual(descending_stairs_recursive_memoized(n), expected)

    def test_two_steps(self): # There are two steps left; especially important test because if we consider the
```

```
        n = 2
        expected = 2

        print(f'Iterative approach: {descending_stairs_iterative(n)}')
        print(f'Recursive approach: {descending_stairs_recursive(n)}')
        print(f'Recursive (optimized) approach: {descending_stairs_recursive_memoized(n)}\n')

        self.assertEqual(descending_stairs_iterative(n), expected)
        self.assertEqual(descending_stairs_recursive(n), expected)
        self.assertEqual(descending_stairs_recursive_memoized(n), expected)

if __name__ == '__main__':
    unittest.main(argv=[''], defaultTest = 'TestProblem1', exit=False)
```

Run Code

Out [30]

```
....
Iterative approach: 0
Recursive approach: 0
Recursive (optimized) approach: 0

Iterative approach: 1
Recursive approach: 1
Recursive (optimized) approach: 1

Selected Input: 5
Iterative approach: 8
Recursive approach: 8
Recursive (optimized) approach: 8

Iterative approach: 2
Recursive approach: 2
Recursive (optimized) approach: 2


----------------------------------------------------------------------
```

```
OK
```

Code Cell 3 of 19

```
In [30]  1
```

Run Code

Code Cell 4 of 19

```
In [30]  1
```

Run Code

Please run the following code cell to check if your code passes some corner cases.

Code Cell 5 of 19 - Hidden Code

Run Code

```
Out [31]    Looking good!
            Looking good!
            Looking good!
            Looking good!
```

```
Looking good!
Looking good!
```

Question 5 of 10

C. Discuss in detail the trade-off (if any) between the recursive and iterative approaches to this problem.

- Analyse their scalability using two different metrics, carefully deriving the theoretical results with an appropriate explanation.
- Complement and check your theoretical analysis with appropriate plots. The range of input should be appropriate.
- Explain which approach you believe is better and in which circumstances one should use one instead of the other. You will need to provide appropriate evidence for your arguments.

Normal

Iterative Approach

```
def descending_stairs_iterative(n):
    '''
                                                cost | times

    '''
    if n == 0:                                   c1     1
        return 0                                 c2     1

    stairs = [0] * (n + 1)                       c3     n + 1
    stairs[0] = 1                                c4     1
    stairs[1] = 1                                c5     1

    for i in range(2, n+1):                      c6     n - 1
        stairs[i] = stairs[i - 1] + stairs[i - 2]    c7     n - 1 # does not iterate through the first two elements in
the list

    return stairs[n]                             c8     1
```

T(n) = c1*1 + c2*1 + c3*(n+1) + c4*1 + c5*1 + c6*(n - 1) + c7*(n-1) + c8*1

T(n) = c1 + c2 + c3*n + c3 + c4 + c5 + c6*n - c6 + c7*n - c7 + c8

T(n) = c3*n + c6*n + c7*n + c1 + c2 + c3 + c4 + c5 - c6 - c7 + c8

We can express the running time as a linear function in the an + b format for constants a and b that depend on the statement costs ck. Because the function's main coefficient is n, the time complexity is a linear function of n. The upper bound of the algorithm's running time as the input n scales is linear since we are populating the list of combinations of steps through only one passage. O(n)

Recursive approach

```
def descending_stairs_recursive(n, steps=0):
    '''
                                                                                costs
(constant) | times
    '''
    if n == 2:                                                                   c1
1
        return 2                                                                 c2
1
    elif n == 1:                                                                 c3
1
        return 1                                                                 c4
1
    elif n <= 0:                                                                 c5
1
        return 0                                                                 c6
1

    return descending_stairs_recursive(n - 2, steps + 1) + descending_stairs_recursive(n - 1, steps + 1)   c7
1
```

The recursive function call for descending_stairs_recursive is this:

- T(n) = T(n-1) + T(n-2) + 7

- To calculate the upper bound, we'll approximate T(n-2) ≈ T(n-1). In reality, T(n-2) will be slightly less costly than T(n-1), because it considers a smaller input space.
    - T(0) = T(1) = 1 and 7 = c (constant)
    - T(n) = 2T(n-1) + c
    - T(n) = 4T(n-2) + 3c
    - T(n) = 8T(n-3) + 7c
    - T(n) = 16*T(n-4) + 15c
    - Generalizing the expression for an arbitrary k

- T(n) = 2^k T(n - 2k) + (2^k - 1)*c
- We can simplify T(n - k) to T(0) when n - k = 0, such that k = n
- T(n) = 2^k/2*T(0) + (2^k/2 - 1)*c
- Simplifying the expression we find that:
- T(n) = (1 + c)*2^n - c
- Main coefficient: 2^n, Complexity: O(2^n)

In this case, the upper bound of the algorithm's running time as the input n scales is exponential (O(2^n)) since our function makes two recursive calls for each step.

We can compare the performance of both algorithms in terms of steps and asymptotic notation. Using the steps is straightforward and captures how the algorithms differ in terms of iteration and recursion. In the iterative approach, most of our steps are concentrated on looping through the list while updating the number of ways to descend the steps. On the other hand, in the recursive approach, the number of steps is updated in terms of the function calls, as we break down the input size.

A raw number of steps is an oversimplified metric to compare the efficiency of algorithms. The number of steps does not directly translate into better execution/running time. We might develop several different algorithms for the same problem (e.g. sorting), whose steps take a variable amount of time to be executed. An algorithm with a larger number of steps can take less time to be executed if during running time it provides a lower number of instructions to the computer.

Thus, if we compare algorithms solely by the number of steps, we would be overlooking aspects such as the actual time taken for each operation, differences in the input as it scales, hardware differences between computers, and even the selection of the programming language. Instead, we can make use of more robust metrics such as asymptotic notation, which provides a metric of how long it would take to run an algorithm as the input scales to infinity, which removes the dependency on machine-specific constraints and the implementation details of the algorithms in the runtime. Instead, we focus on the fundamental growth rate of the algorithm's time complexity as the input scales. We can represent this time complexity in terms of a lower (big-omega), upper (big-O), and tight bound (big-theta). In this case, we'll be applying the big-O notation, as the complexity of the algorithm would not achieve a higher running time for any input size of n.

By the time complexity metric, the closer the algorithms are to constant time O(1), the more efficient they are. In this example, if we compare both approaches, we notice that O(n) is far closer to O(1) compared to O(2^n), as we can visualize in the graphics below.

This indicates our iterative approach would be more efficient in addressing the problem, even for a very large input. However, there is an important difference between the theoretical analysis and the practical application of the algorithm. Realistically, we can estimate the number of steps in the stairs of the airport to be 100 or less. In this case, it can be sufficient to compare the algorithms directly by the

number of steps, and for very small inputs (e.g. N <= 10), we even have the same number of steps or a very small difference between the recursive and iterative approaches, so we could use either of the algorithms interchangeably.

Furthermore, we can also optimize the recursive approach through memoization, which stems from dynamic programming. Through memoization, we store the results of expensive function calls and only reuse them when necessary. This way, we don't need to recompute them; our recursive tree is thus reduced from an exponential to a linear size. Ultimately, the memoized recursive algorithm would have an upper-bound complexity of O(N), achieving the same time complexity as the iterative approach.

The theoretical metric comparison of the algorithms can also be supported by the experimental metric of time elapsed in the runtime (Figures 1 and 2). Considering the results below we notice that both the iterative and the optimized recursive algorithms achieve similar average runtimes as the input scales and that the traditional recursive algorithm exposes an exponential behavior.

Besides, the recursive function typically results in higher memory usage compared to the iterative approach because each recursive call adds a new frame to the function call stack, a data structure that follows the "Last In, First Out" procedure, meaning the last item added is the last item to be removed. In each call, we store local variables and function arguments until the call is finished, which leads to greater memory allocation (we can visualize this under a very small difference in the last plot, considerering the memory constraints of the notebook itself for a larger input).

Code Cell 6 of 19

In [32]

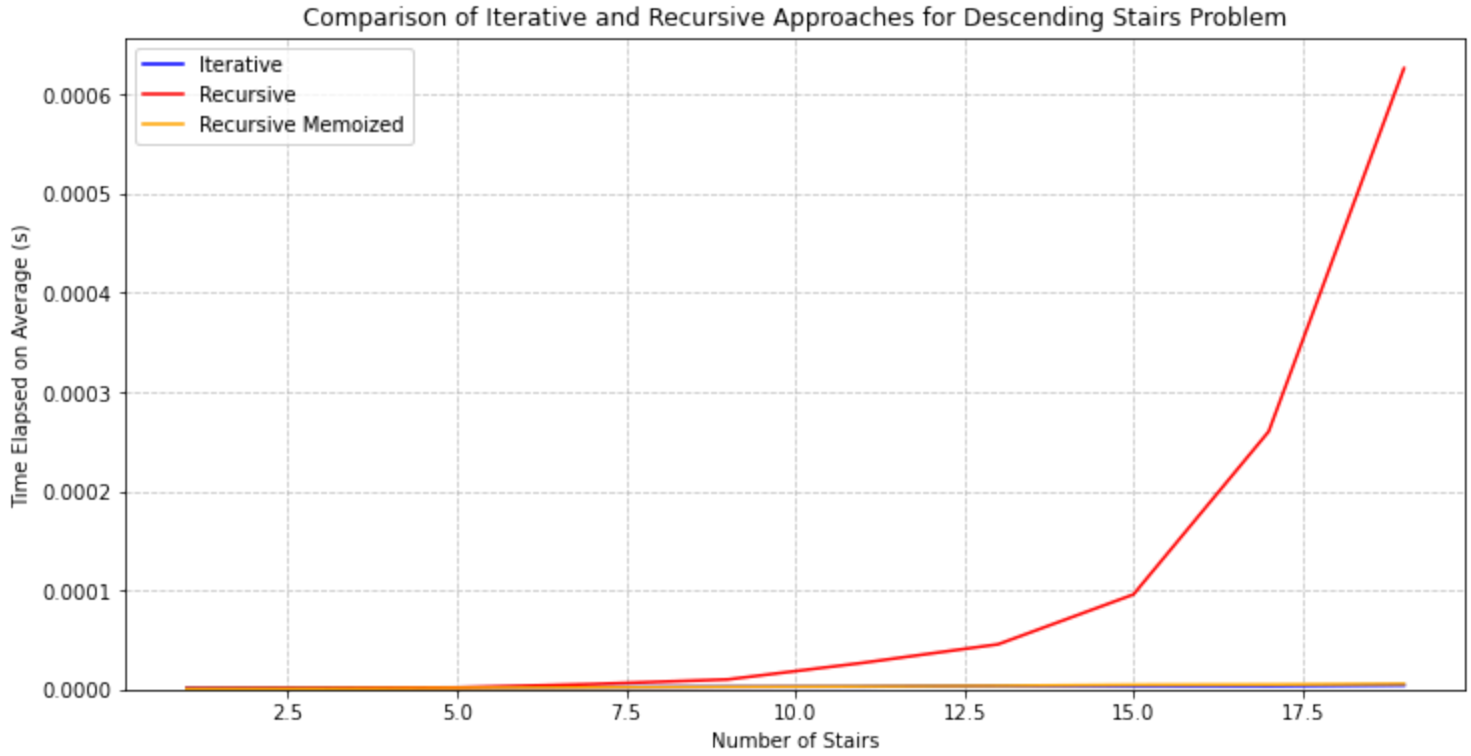
```
import time
import matplotlib.pyplot as plt

input_sizes = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

def run_experiments_avg_running_time(inputs, staircase_function, trials=30):
    '''
    Perform the simulation of the experimental results (in seconds) of different staircase functions according to
the input.

    Args
    ----------
    inputs (list): different sizes of inputs sorted in increasing order
    staircase_function (function): function used for calculating the number of ways of descending a staircase
```

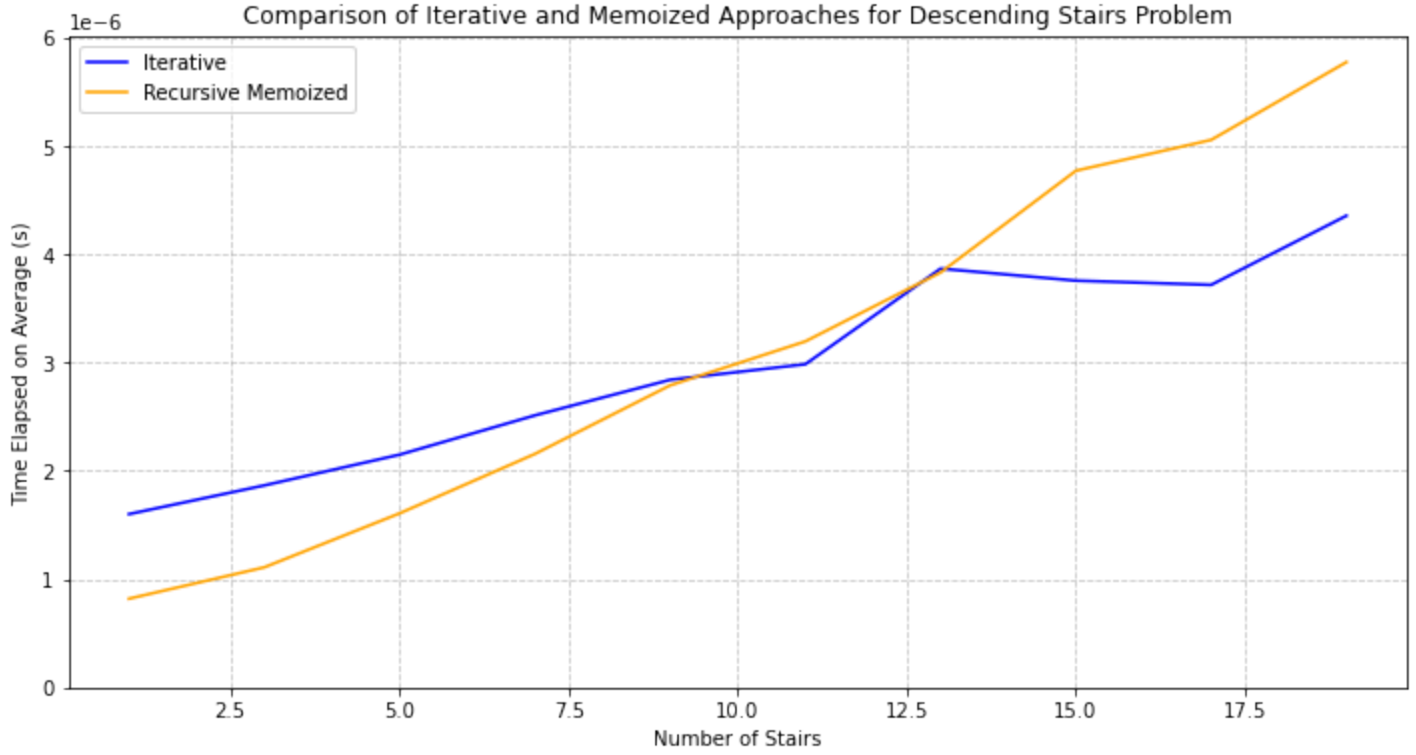
```
    Results
    -------------
    experimental_results (list): experimental results calculated and averaged accross different trials
    '''
    experimental_results = []
    average = 0

    for size in input_sizes:
        for i in range(trials):
            start_time = time.process_time()

            staircase_function(size)

            end_time = time.process_time()

            elapsed = end_time - start_time

            average += elapsed

        average /= trials # averaging the runtime across different trials

        experimental_results.append(average)

    return experimental_results

# Storing results for each function
iterative_results = run_experiments_avg_running_time(input_sizes, descending_stairs_iterative)
recursive_results = run_experiments_avg_running_time(input_sizes, descending_stairs_recursive)
recursive_results_memoized = run_experiments_avg_running_time(input_sizes, descending_stairs_recursive_memoized)

# Figure 1: Comparison of the three approaches together
plt.figure(figsize=(12, 6))
```

```python
plt.plot(input_sizes, recursive_results, label='Recursive', color='red')
plt.plot(input_sizes, recursive_results_memoized, label='Recursive Memoized', color='orange')

plt.title('Comparison of Iterative and Recursive Approaches for Descending Stairs Problem')
plt.xlabel('Number of Stairs')
plt.ylabel('Time Elapsed on Average (s)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylim(bottom=0)

# Figure 2: Comparison of the Iterative and Memoized approaches
plt.figure(figsize=(12, 6))
plt.plot(input_sizes, iterative_results, label='Iterative', color='blue')
plt.plot(input_sizes, recursive_results_memoized, label='Recursive Memoized', color='orange')

plt.title('Comparison of Iterative and Memoized Approaches for Descending Stairs Problem')
plt.xlabel('Number of Stairs')
plt.ylabel('Time Elapsed on Average (s)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.ylim(bottom=0)

plt.show()

```

Run Code

Out [32]


Comparison of Iterative and Recursive Approaches for Descending Stairs Problem
Iterative
Recursive
Recursive Memoized
Time Elapsed on Average (s)
0.0000
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Number of Stairs


Comparison of Iterative and Memoized Approaches for Descending Stairs Problem
1e−6
Iterative
Recursive Memoized
Time Elapsed on Average (s)
0
1
2
3
4
5
6
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Number of Stairs

Code Cell 7 of 19

In [33]
```
# Please run the cell below to install the Memory Profiler LIibrary
!pip install -U memory_profiler
```

Run Code

Out [33]
```
Requirement already satisfied: memory_profiler in /opt/conda/lib/python3.8/site-packages (0.61.0)
Requirement already satisfied: psutil in /opt/conda/lib/python3.8/site-packages (from memory_profiler) (5.7.2)

[notice] A new release of pip available: 22.1.2 -> 24.2
[notice] To update, run: pip install --upgrade pip
```

Code Cell 8 of 19

In [34]
```
from memory_profiler import memory_usage # importing the external library
import matplotlib.pyplot as plt
from statistics import mean



input_sizes = [1, 3, 5, 7, 9, 11, 13, 15]

# Obs: requires running a few times to visualize differences, but the notebook does not support much larger inputs
def run_experiments_avg_memory_usage(inputs, staircase_function, trials=30):
    '''
    Perform the simulation of the experimental results on memory usage (in MiB) of different staircase functions
according to the input.
```

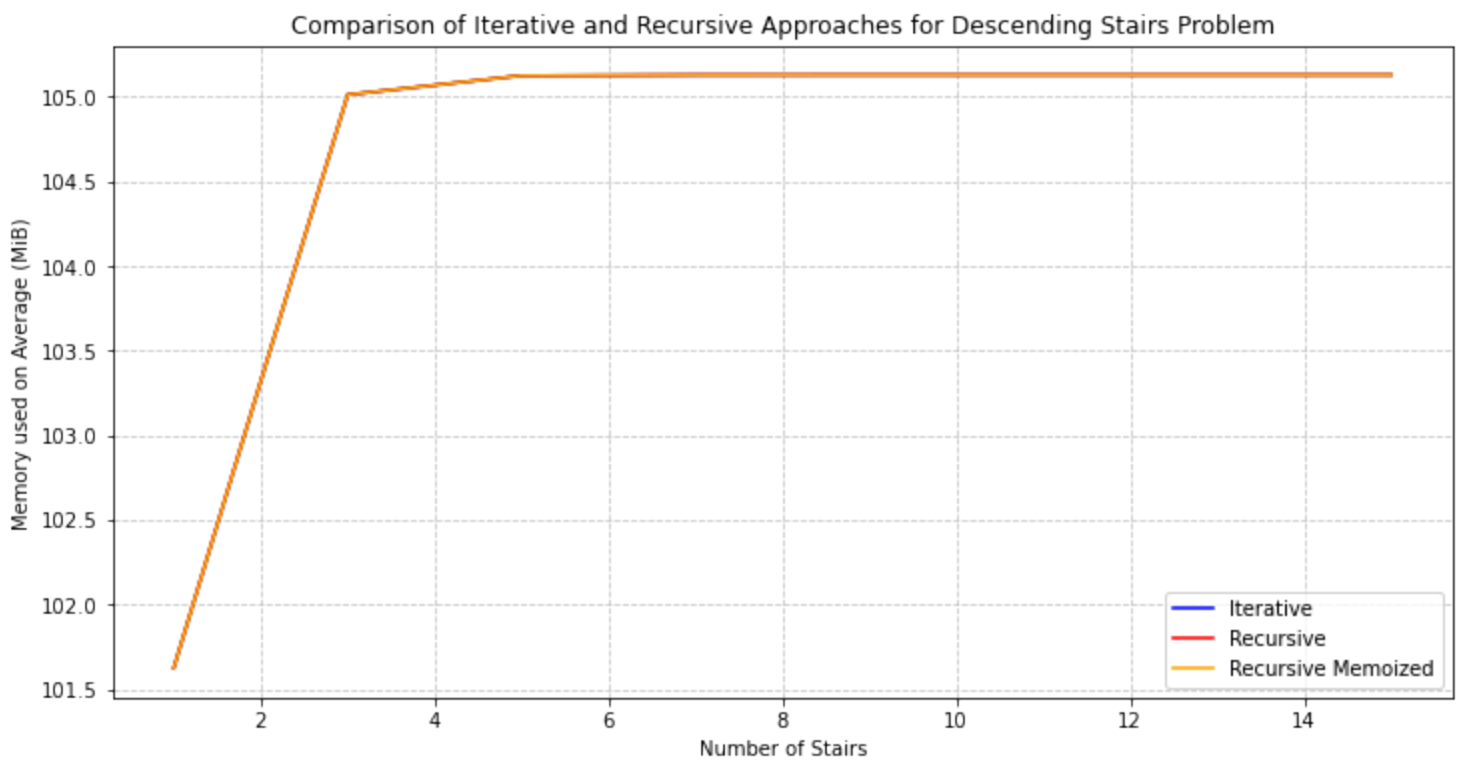
```
    Args
    ----------
    inputs (list): different sizes of inputs sorted in increasing order
    staircase_function (function): function used for calculating the number of ways of descending a staircase
    trials (int): number of repeated experiments

    Results
    -------------
    experimental_results (list): experimental results calculated and averaged accross different trials
    '''
    experimental_results = []
    average = 0

    for size in input_sizes:
        for i in range(trials):

            memory_used = max(memory_usage((staircase_function, (size,)))) # gathering the maximum memory usage measured mebibyte (MiB) at a particular moment

            average += memory_used

        average /= trials

        experimental_results.append(average)

    print(f'Mean of experimental results ({staircase_function.__name__}): {mean(experimental_results):.6f}')
    return experimental_results

# Storing results for each function
iterative_results = run_experiments_avg_memory_usage(input_sizes, descending_stairs_iterative)
recursive_results = run_experiments_avg_memory_usage(input_sizes, descending_stairs_recursive)
recursive_results_memoized = run_experiments_avg_memory_usage(input_sizes, descending_stairs_recursive_memoized)

# Figure 3: Comparison of the three approaches together
```

```
plt.figure(figsize=(12, 6))
plt.plot(input_sizes, iterative_results, label='Iterative', color='blue')
plt.plot(input_sizes, recursive_results, label='Recursive', color='red')
plt.plot(input_sizes, recursive_results_memoized, label='Recursive Memoized', color='orange')

plt.title('Comparison of Iterative and Recursive Approaches for Descending Stairs Problem')
plt.xlabel('Number of Stairs')
plt.ylabel('Memory used on Average (MiB)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)

plt.show()
```

Run Code

Out [34]

```
Mean of experimental results (descending_stairs_iterative): 104.676167
Mean of experimental results (descending_stairs_recursive): 104.676167
Mean of experimental results (descending_stairs_recursive_memoized): 104.676167
```



Problem 2

Our CS110 interns are working on a project to track students' assessment data on a specific learning objective (LO) throughout the semester. We currently have two lists of grades, each sorted in descending order, representing the latest assessment results for our students. Each grade appears as a pair (technically, a tuple) of values `(ID, grade)`, where `ID` represents the student's unique identifier, and `grade` is their corresponding assessment grade.

IDs are unique within each list (since each student is unique), but a student's ID may appear in both lists with different grades. In the latter cases, we want to find the maximum of the two grades and put it correctly within the sorted final list (in descending order). Here are a few other requirements:

- If a student's ID appears in only one list, use that grade as is.
- If two students end up having the same final grade, to present the output, these IDs should be sorted by descending order.

```
Eg.
Input:
    list1 = [ (1, 5), (5, 3), (2, 3), (4, 2) ]
    list2 = [ (3, 5), (4, 3), (1, 2) ]

Output:
    [ (3, 5), (1, 5), (5, 3), (4, 3), (2, 3) ]
```

Note how students 2 and 4 share the same final grade (in this case, 3), and the output returns first **student 4** and then **student 2**.

A. A **possible** brute-force approach to this problem would start by concatenating the two lists, arranging them by `id`, retaining the maximum grade for each, and then using a sorting algorithm to get the desired result, like:

```
Concatenate step : [ (1, 5), (5, 3), (2, 3), (4, 2) , (3, 5), (4, 3), (1, 2) ] →

Group step: [ (1, 5), (1, 2), (2, 3), (3, 5), (4, 2), (4, 3), (5, 3) ] →

Calculation step: [ (1, 5), (2, 3), (3, 5), (4, 3), (5, 3) ] →

Sort step: [ (3, 5), (1, 5), (5, 3), (4, 3), (2, 3) ]
```

Note that in the group step, the only requirement is for the duplicate IDs to be side-by-side (there is no sorting requirement at all in this specific step).

In the code cell provided below, implement this algorithmic strategy using the function name provided. In the sorting step, described above, you make a call to selection sort (which you will need to implement on your own). Demonstrate that it works with at least three test cases. Explain why your test cases are appropriate or sufficient in the companion text cell below.

Code Cell 9 of 19

In [35]
```
1 def selection_sort(A, key=0, reverse = False):
2     '''
3     Performs selection sort on a list of tuples. It can sort based on different keys and in ascending or descending
order.
```

```
    Args:
    -----------
    A (list): list of tuples to be sorted.
    key (int or tuple):
      - Specifies which element(s) of the tuple to use as the sorting key.
      - 0 (default) sorts based on the first element of each tuple.
      - (1, 0) sorts primarily on the second element, then on the first element.
    reverse (bool): If False (default), sort in ascending order. If True, sort in descending order, as similar to the Python sorted() function.

    Returns:
    ------------
    list: The sorted list of tuples.

    Behavior:
    ------------
    - When key=0 and reverse=False:
      Sorts the list based on the first element of each tuple in ascending order.
    - When key=(1, 0) and reverse=True:
      Sorts the list primarily based on the second element of each tuple in descending order.
      For tuples with equal second elements, it sorts based on the first element in descending order.
    '''
    n = len(A)

    if key == 0 and reverse == False: # sort based on first element of the tuple
        for i in range(n):
            min_ind = i # working in ascending order, setting a minimum index first
            for j in range(i + 1, n):
                if A[j][0] < A[min_ind][0]:
                    min_ind = j
            A[i], A[min_ind] = A[min_ind], A[i]
        return A

    elif key == (1, 0) and reverse == True: # sort based on second element of the tuple
```

```
        for i in range(n):
            max_ind = i # working in descending order, setting a maximum index first
            for j in range(i + 1, n):
                if A[j][1] == A[max_ind][1]: # having the same grade
                    if A[j][0] > A[max_ind][0]:
                        max_ind = j
                else:
                    if A[j][1] > A[max_ind][1]: # if grades are different, maintain sorting priority of the grades
                        max_ind = j
            A[i], A[max_ind] = A[max_ind], A[i]
        return A

def sort_grades_brute_force_selection(list1, list2):
    '''
    Merges and sorts two lists of student grades using a brute force selection sort approach.

    Takes two lists of student grades, where each grade is represented as a tuple
    (student_id, grade). Merges these lists, removes duplicate entries for the same student
    (keeping the higher grade), and sorts the result in descending order of grades.

    Args:
    -------------
    list1 (list of tuples): A list of (student_id, grade) tuples.
    list2 (list of tuples): A list of (student_id, grade) tuples.

    Returns:
    ------------
    list of tuples: A sorted list of (student_id, grade) tuples, where:
        - Each student_id appears only once
        - Grades are sorted in descending order
        - For equal grades, student_ids are sorted in descending order
    '''
    # Concatenation step
```

```
72
73      # Grouping step
74      grouped_list = selection_sort(concat_list)
75
76      # Calculation step
77      i = 0
78      while i < len(grouped_list) - 1:
79          x, y = grouped_list[i], grouped_list[i+1]
80          if x[0] == y[0]:  # For the same student ID
81              if x[1] < y[1]:
82                  grouped_list.pop(i)  # Remove the lower grade at the current tuple (x)
83              else:
84                  grouped_list.pop(i+1)  # Remove the lower grade at the next tuple (y)
85          else:
86              i += 1
87
88      # Sorting step
89      sorted_list = selection_sort(grouped_list, key=(1,0), reverse=True)
90
91      return sorted_list
92
93
94  # Basic test case
95  list1 = [ (1, 5), (5, 3), (2, 3), (4, 2) ]
96  list2 = [ (3, 5), (4, 3), (1, 2) ]
97
98  sort_grades_brute_force_selection(list1, list2)
```

Run Code

Out [35]

```
[(3, 5), (1, 5), (5, 3), (4, 3), (2, 3)]
```

Code Cell 10 of 19

In [36]

```
'''
No text-cell provided. Explanation for tests provided below:

My current test cases are sufficient because they experiment with edge cases of the algorithm, including having one of the lists empty, exploring reverse orders, keeping grades from the same student and several equal grades from different students. Besides, I also included a more complex test case at the end which involves pairs of grades of the same student across different lists, both with equal and different grades (with all the grades listed in descending order; sorted by the expectations of the algorithm).
'''
import unittest

class TestProblem2(unittest.TestCase):

    def test_sorted_empty_list(self): # we have one empty list as a part of the input
        list1 = []
        list2 = [(1, 5), (2, 4), (3, 3)]
        expected = [(1, 5), (2, 4), (3, 3)]

        result = sort_grades_brute_force_selection(list1, list2)
        print(f'Final List: {result}')

        self.assertEqual(result, expected)

    def test_reversed_one_student_same(self): # one student has a record of all grades, listed in reversed order (reversed from the expected descending order from the problem)
        list1 = [(1, 3)]
        list2 = [(1, 1)]
        expected = [(1, 3)]

        result = sort_grades_brute_force_selection(list1, list2)
```

```
27
28          self.assertEqual(result, expected)
29
30      def test_reversed_all_grades_same(self): # all students, listed in a reversed order (reversed from the expected
    descending order from the problem), for both lists have the same grade
31          list1 = [(5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (10, 3)]
32          list2 = [(1, 3), (2, 3), (3, 3), (4, 3)]
33          expected = [(10, 3), (9, 3), (8, 3), (7, 3), (6, 3), (5, 3), (4, 3), (3, 3), (2, 3), (1, 3)]
34
35          result = sort_grades_brute_force_selection(list1, list2)
36          print(f'Final List: {result}')
37
38          self.assertEqual(result, expected)
39
40      def test_sorted_grade_all_pairs(self): # each student has grades listed in both lists (which can be the same or
    different grades), all listed in sorted order of grades (descending)
41          list1 = [(4, 3), (2, 2), (3, 2), (1, 1), (5, 1)]
42          list2 = [(1, 5), (3, 4), (4, 4), (2, 2), (5, 1)]
43          expected = [(1, 5), (4, 4), (3, 4), (2, 2), (5, 1)]
44
45          result = sort_grades_brute_force_selection(list1, list2)
46          print(f'Final List: {result}')
47
48          self.assertEqual(result, expected)
49
50  if __name__ == '__main__':
51      unittest.main(argv=[''], defaultTest = 'TestProblem2', exit=False)
```

Run Code

Out [36]

```
....
Final List: [(10, 3), (9, 3), (8, 3), (7, 3), (6, 3), (5, 3), (4, 3), (3, 3), (2, 3), (1, 3)]
```

```
Final List: [(1, 5), (2, 4), (3, 3)]
Final List: [(1, 5), (4, 4), (3, 4), (2, 2), (5, 1)]

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK
```

Code Cell 11 of 19

```
In [36]  1
```

Run Code

Code Cell 12 of 19

```
In [36]  1
```

Run Code

Question 6 of 10

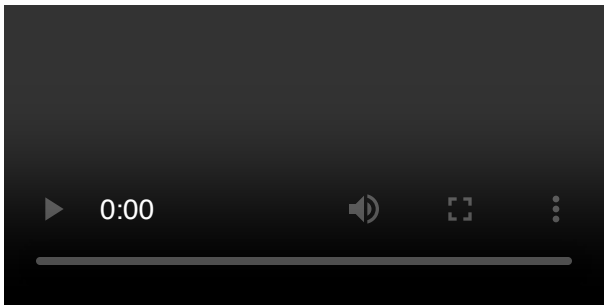
B. It's sensible to conjecture that an improved approach should take advantage of the pre-sorting of each list in your solution. This approach acts as a shortcut, bypassing the need to sort the combined array. Take your time, and don't jump into coding the algorithm—that is not the goal of this question.

Record a single video where you explain how such an approach would work. As you work through this question:

- consider the following potential situations and how your strategy can handle these:

    - what happens when there is a student who has only been assessed once?
    - what happens when a student has been assessed twice?
    - how would you adapt your algorithmic strategy if there were more than two lists as input?
- your face should be visible at all times, and your explanation should be clear enough for anyone in this course to follow it and go ahead to the next (implementation) question without contacting you for any clarification. Like in problem 1, you are encouraged to use any resources you deem relevant to explain your algorithm (flowcharts, whiteboards, etc).
- your video cannot last for more than 4 minutes. Any content beyond this threshold will be ignored. Use this time constraint to supercharge your explanation—focus on **how** the algorithms work and **why** they are guaranteed to work.

pre_sorting_explanation.mp4 (5.74 MB)



C. Produce a Python implementation of your algorithm using the function name provided and demonstrate that it works as intended when **two** lists are provided as input. You can't use the built-in sorting function in any step of this algorithm; instead, use your favourite sorting algorithm (where needed) and explain why it is a suitable choice (in the text box provided).

Show that this implementation works on the three test cases you have provided above for the brute-force method.

Code Cell 13 of 19

In [37]
```
'''
No text-box provided; explanation for Merge Sort provided below:


Merge sort is an adequate algorithm for tackling sorting of large, unsorted arrays. It leverages the divide and
conquer approach to recursively break down a larger array into smaller subarrays until we end up with a single
element array. We then sort and combine each subpart until we achieve the final sorted array with all elements.

```

```
Merge sort is also a stable sorting algortihm, which maintains the relative order of the elements with the same
value when sorting. However, it is important to mention that merge sort could be costly in terms of its running time
for smaller, sorted arrays, since it is not an adaptive sorting algorithm (its time complexity remains the same for
any order of the array). Therefore, for the examples mentioned in this assignment, which are very small, merge sort
would not be the most efficient sorting algorithm.

Nevertheless, if we consider a real world application of this grade-sorting software, it is preferrable to have a
solution which is able to handle larger, unsorted arrays more efficiently as the number of grades stored per student
(or the number of students) increases over time. Using an algorithm which performs well on smaller arrays such as
insertion sort could be beneficial on the short term, but could provide a performance bottleneck for the scalability
of the solution in the longer-term.

My selection of sorting algorithm for this assignment will keep be centered in a real-world application perspective,
which is why I would select merge sort as my sorting algorithm. Alternatively, we could optimize merge sort for
smaller array applications if we create a hybrid algorithm which starts with merge sort but at some point in the
recursion tree applies insertion sort to sort smaller arrays (equivalent to Tim Sort).
'''

# Global merge function
def merge(left, right, key=0, reverse=False):
    '''
    Merges two sorted lists of tuples into a single sorted list.

    Args:
    ---------
    left (list): The left sorted list of tuples.
    right (list): The right sorted list of tuples.
    key (int or tuple): Specifies which element(s) of the tuple to use as the sorting key.
        - 0 (default) uses the first element.
        - (1, 0) uses the second element, then the first for ties.
    reverse (bool): If False (default), sort in ascending order. If True, sort in descending order.

    Returns:
    ----------
    list: A new sorted list containing all elements from both input lists.

    Observations:
```

```
        - This function assumes that both input lists are already sorted according to the specified key and order.
        - The function preserves the stability of the sort.
    '''
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        left_item, right_item = left[i], right[j]

        # altering results depending on the sorting key
        if key == 0:
            left_value, right_value = left_item[0], right_item[0]
        elif key == (1, 0):
            left_value = (left_item[1], left_item[0])
            right_value = (right_item[1], right_item[0])

        # altering results depending on the reversed parameter
        if reverse:
            if left_value > right_value:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        else:
            if left_value < right_value:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

    result.extend(left[i:])
    result.extend(right[j:])
```

```python

# Merge Sort Function
def merge_sort(A, key=0, reverse=False):
    """
    Sorts a list of tuples using the merge sort algorithm.

    Args:
    ---------
    A (list): The list of tuples to be sorted.
    key (int or tuple): Specifies which element(s) of the tuple to use as the sorting key.
    - 0 (default) uses the first element.
    - (1, 0) uses the second element, then the first for ties.

    reverse (bool): If False (default), sort in ascending order. If True, sort in descending order.

    Returns:
    ------------
    list: A new sorted list containing all elements from the input list.

    Note:
    ------------
        - The sort is stable, meaning that the relative order of equal elements is preserved.
    """
    # Base case: list with one element
    if len(A) <= 1:
        return A

    mid = len(A) // 2

    left_half = merge_sort(A[:mid], key=key, reverse=reverse)
    right_half = merge_sort(A[mid:], key=key, reverse=reverse)

    # Merge the sorted halves using the merge function
```

```python

# sort_grades_shortcut function using merge_sort
def sort_grades_shortcut(list1, list2):
    '''
    Merges and sorts two lists of student grades, removing duplicates and keeping the highest grade for each student.

    Args:
    -----------
    list1 (list): A list of tuples, where each tuple is (student_id, grade).
    list2 (list): A list of tuples, where each tuple is (student_id, grade).

    Returns:
    -----------
    list: A sorted list of tuples (student_id, grade), where:
        - Each student_id appears only once
        - The grade for each student is the highest among their grades in both input lists
        - The list is sorted in descending order of grades, then by student_id for ties

    General Process:
    ------------------
    1. Sorts both input lists using merge sort
    2. Merges the sorted lists
    3. Uses a dictionary to track repeated student grades and only keep the highest grade for each student
    4. Returns the values from the dictionary as a list
    '''
    list1 = merge_sort(list1, key=(1, 0), reverse=True)
    list2 = merge_sort(list2, key=(1, 0), reverse=True)

    merged_list = merge(list1, list2, key=(1, 0), reverse=True)

    grouped_dict = {}
    for elem in merged_list:
        if elem[0] not in grouped_dict or elem[1] > grouped_dict[elem[0]][1]: # If the ID is already in the
```

```
dictionary, only keep the tuple with the higher grade
            grouped_dict[elem[0]] = elem

    return list(grouped_dict.values())

# Basic test case
list1 = [ (1, 5), (5, 3), (2, 3), (4, 2) ]
list2 = [ (3, 5), (4, 3), (1, 2) ]
result = sort_grades_shortcut(list1, list2)
```

Run Code

Code Cell 14 of 19

In [38]

```
import unittest

class TestProblem2(unittest.TestCase):

    def test_sorted_empty_list(self):
        list1 = []
        list2 = [(1, 5), (2, 4), (3, 3)]
        expected = [(1, 5), (2, 4), (3, 3)]

        result = sort_grades_shortcut(list1, list2)
        print(f'Final List: {result}')

        self.assertEqual(result, expected)

    def test_reversed_all_students_same(self): # one student has a record of all grades, listed in reversed order
(reversed from the expected descending order from the problem)
        list1 = [(1, 3)]
```

```
        list2 = [(1, 1)]
        expected = [(1, 3)]

        result = sort_grades_shortcut(list1, list2)
        print(f'Final List: {result}')

        self.assertEqual(result, expected)

    def test_reversed_all_grades_same(self): # all students, listed in a reversed order (reversed from the expected descending order from the problem), for both lists have the same grade
        list1 = [(5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (10, 3)]
        list2 = [(1, 3), (2, 3), (3, 3), (4, 3)]
        expected = [(10, 3), (9, 3), (8, 3), (7, 3), (6, 3), (5, 3), (4, 3), (3, 3), (2, 3), (1, 3)]

        result = sort_grades_shortcut(list1, list2)
        print(f'Final List: {result}')

        self.assertEqual(result, expected)

    def test_reversed_grade_all_pairs(self): # each student has grades listed in both lists (which can be the same or different grades), all listed in reversed order of grades (ascending)
        list1 = [(1, 1), (3, 2), (2, 2), (4, 3), (5, 1)]
        list2 = [(1, 5), (3, 4), (2, 2), (4, 4), (5, 1)]
        expected = [(1, 5), (4, 4), (3, 4), (2, 2), (5, 1)]

        result = sort_grades_shortcut(list1, list2)
        print(f'Final List: {result}')

        self.assertEqual(result, expected)

if __name__ == '__main__':
    unittest.main(argv=[''],  defaultTest = 'TestProblem2', exit=False)
```

Run Code

```
Out [38]    ....
            Final List: [(10, 3), (9, 3), (8, 3), (7, 3), (6, 3), (5, 3), (4, 3), (3, 3), (2, 3), (1, 3)]
            Final List: [(1, 3)]
            Final List: [(1, 5), (4, 4), (3, 4), (2, 2), (5, 1)]
            Final List: [(1, 5), (2, 4), (3, 3)]

            ----------------------------------------------------------------------
            Ran 4 tests in 0.004s

            OK
```

Code Cell 15 of 19

```
In [38]  1
```

Run Code

Code Cell 16 of 19

```
In [38]  1
```

Run Code

Question 7 of 10

D. Discuss in detail the trade-off (if any) between the two approaches (brute-force and shortcut) to this problem.

- Analyse their runtime scalability, carefully deriving the theoretical results with an appropriate explanation.
- Complement and check your theoretical analysis with appropriate plots. (If there is a disparity, explain the divergence.)
- Explain which approach you believe is better and in which circumstances one should use one instead of the other. You will need to provide appropriate evidence for your arguments.

Conclude by detailing your preference for one of the algorithms and elucidating the scenarios in which this choice would be most appropriate. Justify your assessment in detail. For instance, is the "shortcut" algorithm true to its name?

Normal

Time Complexity of Brute Force Approach

```
def sort_grades_brute_force_selection(list1, list2):
                                                                    Cost
    concat_list = list1 + list2                                     O(1)
    grouped_list = selection_sort(concat_list)                      O(N^2)
    i = 0                                                           O(1)

    while i < len(grouped_list) - 1:                                O(N)
        x, y = grouped_list[i], grouped_list[i+1]                   O(1)
        if x[0] == y[0]:                                            O(1)
            if x[1] < y[1]:                                         O(1)
                grouped_list.pop(i)                                 O(N) # popping at specific index
            else:                                                   O(1)
                grouped_list.pop(i+1)                               O(N) # popping at specific index
        else:                                                       O(1)
            i += 1                                                  O(1)

    sorted_list = selection_sort(grouped_list, key=(1,0), reverse=True)  O(N^2)
    return sorted_list                                              O(1)
```

The while loop .pop() operations could still be considered as O(N^2) in a worst-case scenario where we need to pop the lowest from the beginning of the list each time, causing the most significant index shift. The overall complexity of the algorithm is obtained by selecting the dominant term. Considering the above-listed upper-bound complexities, the overall time complexity of the algorithm is O(N^2).

Time Complexity of the Shortcut Approach

```
def sort_grades_shortcut(list1, list2):                             Cost
```

```
    list1, steps1 = merge_sort(list1, key=(1, 0), reverse=True)                    O(N*logN)
    list2, steps2 = merge_sort(list2, key=(1, 0), reverse=True)                    O(N*logN)

    merged_list = merge(list1, list2, key=(1, 0), reverse=True)                    O(N)

    grouped_dict = {}                                                              O(1)
    for elem in merged_list:                                                       O(N)
        if elem[0] not in grouped_dict or elem[1] > grouped_dict[elem[0]][1]:      O(1)
            grouped_dict[elem[0]] = elem                                           O(1)

    result = list(grouped_dict.values())                                           O(N)
    total_steps += len(result)                                                     O(1)

    return result                                                                  O(1)
```

Complexity of merge()

- The merge function's time complexity is O(n) since it iterates through both lists once and performs constant-time operations on each element.

Complexity of merge_sort()

- The time complexity of merge_sort() is O(n log n) where n in the function is the number of elements in the input list.

Overall complexity of the algorithm

- Considering the dominant term among all the other complexities listed above, we discover that our overall complexity is O(N*log N)

Comparing both algorithms by their upper bound complexity, we discover the shortcut approach provides a more efficient solution to grade sorting. In practical terms, the shortcut algorithm makes the sorting and merging operations flexible. The function merge was selected to be incorporated outside the main merge_sort() function so we could merge the two pre-sorted lists while making sure the elements are ordered in the final list. In sequence, the process of calculating the maximum is also optimized to remove the .pop() operations from the brute-force approach. Using dictionaries to keep count of the duplicates prevents us from altering indexes in the grouped list of grades, an operation that could take O(N) in a scenario where we are changing an index at the beginning of the list.

In terms of time complexity, therefore, the preference would be selecting the shortcut, instead of the brute-force approach.

Another way we can compare both algorithms is in terms of their space complexity:

- Concatenated list1 + list2: O(N)
- Selection Sort Space complexity: O(1)
- Grouped List: O(N)
- Variables in the loop: single variables of space complexity O(1) each
- Final sorted list: O(N)

Considering the dominant term, the overall space complexity of the algorithm is O(N).

```
def sort_grades_shortcut(list1, list2):
```

- Concatenated list1 + list2: O(N)
- Merge Sort Space Complexity: O(N)
- Merged list: O(N)
- Dictionary: O(N)
- Result list: O(N)

Considering the dominant term, the overall space complexity of the algorithm is O(N).

Although the space complexity of both algorithms overall is equivalent, if we have memory-constrained environments, selection_sort will be a more space-efficient choice compared to merge sort, which uses a recursive approach that requires additional space to merge several smaller sorted subarrays.

Besides, in realistic terms, considering the applications of this scenario to CS110 interns, it can be estimated that:

- There are approximately 10 different CS110 sessions, each with roughly 20 students
    - 200 students
- On an assignment, a student is graded on 6 LOs, meaning:
    - 200 * 60 = 1200 grades
- There are 4 assignments in the year
    - 4 * 1200 = 4800 grades

The upper bound scale of the input would be less than half of 10^3, which is small considering high-scale computing, but it can still show a significant difference in complexity. Considering the experimental results below (Figure 4), in situations of larger input (e.g. N > 100), our shortcut approach which uses merge sort and pre-sorting would still perform with significantly less running time than our brute-force approach. However, if we are executing this algorithm on a small pool of grades (e.g. N < 50), using the brute force approach could be sufficient to fulfill the purposes of the task without many significant differences.

Alternatives for optimization include:

1. Optimize the brute-force algorithm by incorporating merge sort as a substitute for selection sort, and changing the logic of the algorithm so that instead of popping elements off the grouped list, we create a new list with the maximum of each element.

Optimize the shortcut approach by incorporating a hybrid version of Merge Sort and Insertion Sort (Tim Sort) which would work flexibly on both smaller, sorted arrays, as well as larger, unsorted arrays.

Code Cell 17 of 19

In [39]:

```
import matplotlib.pyplot as plt
import random

def generate_grade_lists(num_students, max_id, grade_range=(0, 5), duplicate_chance=0.2):
    '''
    """
    Generates two lists of student grades for testing purposes.

    Args:
    -------------
    num_students (int): the number of students to generate grades for.
    max_id (int): the maximum student ID. IDs will be randomly chosen between 1 and this number.
    grade_range (tuple): A tuple of (min_grade, max_grade). Grades will be randomly chosen within this range (inclusive). Default is (0, 5).
    duplicate_chance (float): The probability (0 to 1) that a student ID in list1 will also appear in list2. Default is 0.2 (20% chance).

    Returns:
    -------------
    tuple: A tuple containing two lists (list1, list2), where each list contains tuples of (student_id, grade).
    '''
    list1 = []
    list2 = []

    for _ in range(num_students):
        student_id = random.randint(1, max_id) # pick random student ID
```

```
        grade2 = random.randint(*grade_range) # pick random grade

        list1.append((student_id, grade1))


        if random.random() < duplicate_chance:
            list2.append((student_id, grade2)) # add a chance for duplicate IDs with different grades
        else:
            new_id = random.randint(1, max_id)
            while new_id == student_id:
                new_id = random.randint(1, max_id)
            list2.append((new_id, grade2))

    return list1, list2


num_students = [x for x in range(10, 500, 10)]  # Number of students in each list
max_id = 100 # Maximum student ID, from our estimate above

def run_experiments_avg_running_time(input_sizes, generate_input_function, max_id, grade_sorting_function,
trials=30):
    '''
    Perform experiments to calculate the average running time (in seconds) for grade sorting functions with various
input sizes.
    Run multiple trials of a grade sorting function for different input sizes, measuring the average execution time
for each input size.

    Args:
    ----------
    input_sizes (list): list of different input sizes to test, sorted in increasing order.
    generate_input_function (function): function selected to generate input data
    max_id (int): maximum student ID to use in input generation.
    grade_sorting_function (function): the grade sorting function to be tested.
    trials (int): number of trials to run for each input size. Default is 30.
```

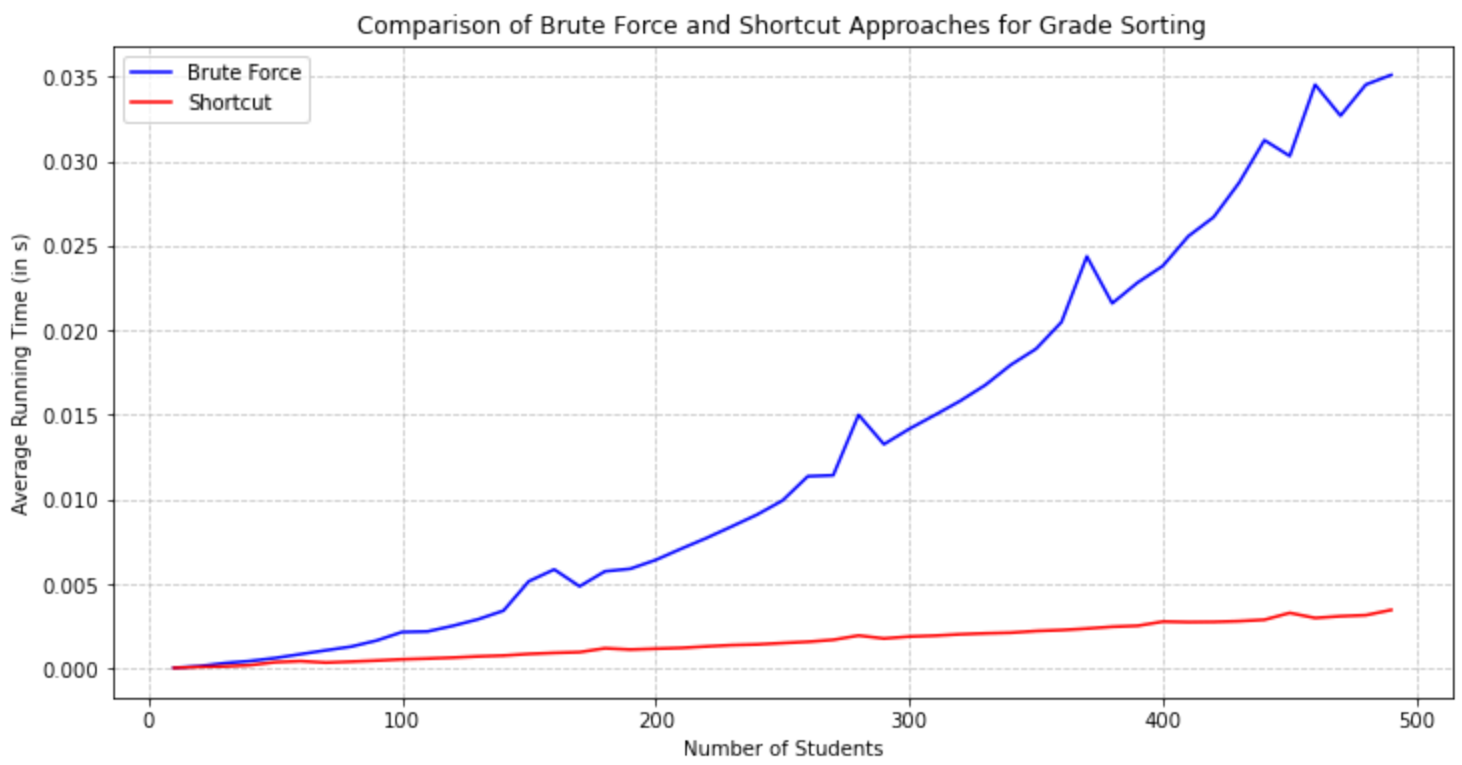
```
    Returns:
    -------------
    experimental_results (list): experimental results calculated and averaged accross different trials
    '''
    experimental_results = []
    average = 0

    for size in input_sizes:
        for i in range(trials):
            list1, list2 = generate_grade_lists(size, max_id) # not considering the generation of input for the time

            start_time = time.process_time()

            grade_sorting_function(list1, list2)

            end_time = time.process_time()

            elapsed = end_time - start_time

            average += elapsed

        average /= trials

        experimental_results.append(average)

    print(f'Mean of experimental results ({grade_sorting_function.__name__}): {mean(experimental_results):.6f}')
    return experimental_results

brute_force_results = run_experiments_avg_running_time(num_students, generate_grade_lists, max_id,
sort_grades_brute_force_selection)
shortcut_results = run_experiments_avg_running_time(num_students, generate_grade_lists, max_id,
sort_grades_shortcut)

# Figure 4
```

```
plt.figure(figsize=(12, 6))
plt.plot(num_students, brute_force_results, label='Brute Force', color='blue')
plt.plot(num_students, shortcut_results, label='Shortcut', color='red')

plt.title('Comparison of Brute Force and Shortcut Approaches for Grade Sorting')
plt.xlabel('Number of Students')
plt.ylabel('Average Running Time (in s)')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)

plt.show()
```

Run Code

Out [39]

```
Mean of experimental results (sort_grades_brute_force_selection): 0.013020
Mean of experimental results (sort_grades_shortcut): 0.001590
```



Code Cell 18 of 19

In [40]

```
1  import matplotlib.pyplot as plt
2  import random
3
4  # Obs: requires running a few times to visualize differences, but the notebook does not support larger inputs
5  def run_experiments_space_complexity(input_sizes, generate_input_function, max_id, grade_sorting_function, trials =
   30):
6      '''
7      Perform experiments to calculate the average memory used (in MiB) for grade sorting functions with various input
   sizes.
8      Run multiple trials of a grade sorting function for different input sizes, measuring the average execution time
   for each input size.
9
10     Args:
```

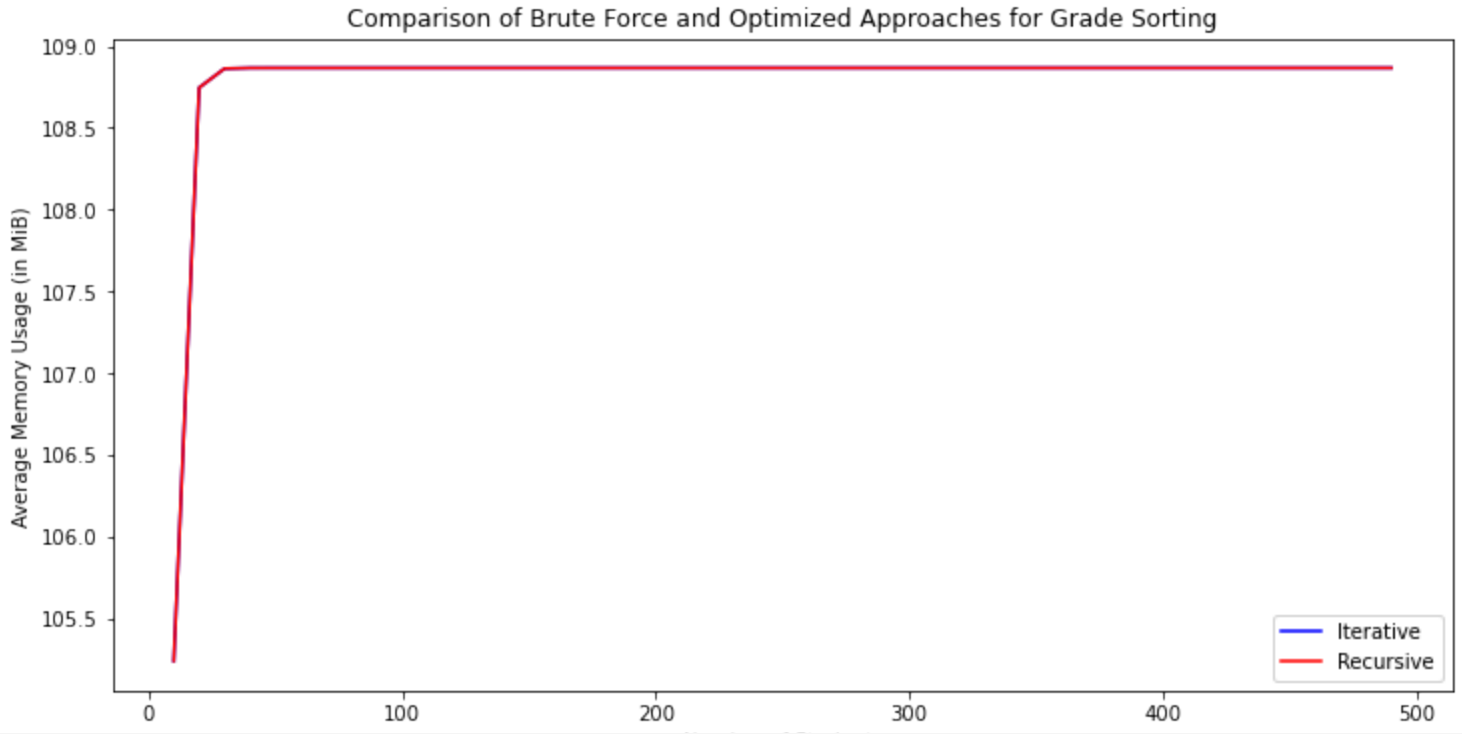
```
    input_sizes (list): list of different input sizes to test, sorted in increasing order.
    generate_input_function (function): function selected to generate input data
    max_id (int): maximum student ID to use in input generation.
    grade_sorting_function (function): the grade sorting function to be tested.
    trials (int): number of trials to run for each input size. Default is 30.

    Returns:
    -------------
    experimental_results (list): experimental results calculated and averaged accross different trials
    '''
    experimental_results = []
    average = 0

    for size in input_sizes:
        for i in range(trials):
            list1, list2 = generate_grade_lists(size, max_id)

            memory_used = max(memory_usage((grade_sorting_function, (list1, list2,)))) # gathering the maximum memory usage measured mebibyte (MiB) at a particular moment

            average += memory_used

        average /= trials

        experimental_results.append(average)

    print(f'Mean of experimental results ({grade_sorting_function.__name__}): {mean(experimental_results):.6f}')
    return experimental_results

brute_force_results = run_experiments_space_complexity(num_students, generate_grade_lists, max_id, sort_grades_brute_force_selection)
shortcut_results = run_experiments_space_complexity(num_students, generate_grade_lists, max_id, sort_grades_shortcut)

```

```
plt.plot(num_students, brute_force_results, label='Iterative', color='blue')
plt.plot(num_students, shortcut_results, label='Recursive', color='red')

plt.title('Comparison of Brute Force and Optimized Approaches for Grade Sorting')
plt.xlabel('Number of Students')
plt.ylabel('Average Memory Usage (in MiB)')
plt.legend()
plt.grid(True, which='minor', linestyle=':', alpha=0.4)


plt.show()
```

Run Code

Out [40]

```
Mean of experimental results (sort_grades_brute_force_selection): 108.790574
Mean of experimental results (sort_grades_shortcut): 108.790574
```



Code Cell 19 of 19

```
In [40]  1
```

Run Code

Question 8 of 10

LO applications

Help the grader grasp the depth of your LO applications. For each core course LO (there are 6), explain how you have applied them specifically in this problem set (30-word description each). Please include a word count at the end of each justification.

Normal

#professionalism: The assignment has included all parts, including videos, explanations, code, and visualizations. Included references organized in an alphabetized list in APA style; completed the AI statement.
#cs110-AlgoDataStruct: In problem 1, I explained the use of a list to compute the cumulative number of staircase combinations, as well as compared the difference between an iterative versus a recursive approach based on a function call stack. In problem 2, explained the divide and conquer approach of merge sort and highlighted the use of a dictionary to keep track of the grade duplicates in the shortcut approach.
#cs110-CodeReadability: Throughout the assignment, wrote code that uses appropriate function and variable naming, comments, and docstrings. Added inline comments to clarify specific details of the implementation. I also included test cases with the widely industry-used library unittest.
#cs110-ComplexityAnalysis: In problem 1, I derived the upper bound time complexity (big O) of the algorithms by hand, achieving functions in terms n and scaling it to several iterations. In problem 2, derived the big-O notation out of more complex functions with other nested algorithms within it. Furthermore, I also included thorough interpretations of the results, including suggestions and implementations for further optimization

#cs110-ComputationalCritique: In problems 1 and 2, I compared different algorithmic approaches (recursive vs iterative, brute-force vs shortcut), supporting which would be the most appropriate choice for a particular problem using factors such as real-world applications and increasing input size. Contrasted approaches both in terms of theoretical complexity analysis and experimental results.
#cs110-PythonProgramming: Ensured my code was functional, correct, and robust by including over three test cases, which explored basic and edge cases for each problem. With the simulations, I made sure that the visualizations were plotted for a broad range of inputs and included titles, axis descriptions, and legends.

Question 9 of 10

HC application

What was the single most relevant HC you applied while working through this problem set? In a 50-word description, examine how this HC has supercharged your approach and allowed you to improve your own work.
Please include a word count at the end of your justification.

Normal

#modeling: I applied this HC while simulating the runs of the algorithms approached in problems 1 and 2, and comparing metrics of time (average runtime) and space (average memory usage) visually through plots. Using this HC, combined with the theory derived from time and space complexity, I developed simulations of experiments performed across different input sizes with results averaged over the number of trials to avoid noise in the data.

Question 10 of 10

References

Please write here all the references you have used for your work. If you have used AI, include here the script of your interactions with AI.

Normal

Simulations and plot visualizations were inspired by the code on the Breakout Workbook for *Session 10 - [5.2] Randomized Quicksort.*
https://forum.minerva.edu/app/courses/3282/sections/11992/classes/86127/review?tab=workbooks

Cormen, T. H. (n.d.). *Introduction to algorithms (fourth edition)*. Ebooks World. https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

Computer science. StudySmarter UK. (n.d.). https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/fibonacci-algorithm/

GeeksforGeeks. (2022, August 1). *Adaptive and non-adaptive sorting algorithms*. https://www.geeksforgeeks.org/adaptive-and-non-adaptive-sorting-algorithms/

GeeksforGeeks. (2023, July 20). *Count ways to reach the nth stair using step 1, 2 or 3*. https://www.geeksforgeeks.org/count-ways-reach-nth-stair-using-step-1-2-3/

GeeksforGeeks. (2024, August 7). *Sorting algorithms*. https://www.geeksforgeeks.org/sorting-algorithms/

GeeksforGeeks. (2022b, September 7). *What does "space complexity" mean?* https://www.geeksforgeeks.org/g-fact-86/

Silveira, O. S. (2023, March 6). *A beginner's guide to unit tests in Python (2023)*. Dataquest. https://www.dataquest.io/blog/unit-tests-python/

🏁 You are all done! Congratulations on completing your first CS110 assignment! 🎉