DSA: A1, A2 and A3 Desk Checking Lab Exercises

Leonardo Dominici, Alexia Muresan, Katja Tuma

October 16, 2024

Abstract

The purpose of this documents is to provide you concrete preparation material for the examination of the practicum.

It contains pseudocode and code of the tasks from the assignment A1, A2 and A3. Remember that the practicum examination will also include A4 (due this week) and **will not include** pseudocode.

While the pseudocode is correct, the code may contain conceptual or minor syntax errors. Every task contains a series of questions regarding the previous code. At the end of every assignment you will also find multiple technical questions about the relevant topics.

Contents

1 Assignment 1

1.1 Task 0: Operator overloading

This code creates a Vector class. Every Vector object is made of an x and a y value, they represent the coordinates of a vector in a 2D space. The class overloads the "+" operator and realizes a vector sum.

Pseudocode

```
1   Define Vector class
2       Define init(x, y)
3           Assign x to field x
4           Assign y to field y
5       Define add(other)
6           If other is instance of class Vector
7               then return the summed Vector
8           else
9               raise TypeError
10
11  # Main code
12  v1 = (1, 2)
13  v2 = (3, 4)
14  v3 = v1 + v2
```

Code

```python
1   class Vector:
2       def __init__(self, x, y):
3           self.x = x
4           self.y = y
5
6       def __add__(self, other):
7           if isinstance(other, Vector):
8               return Vector(self.x + other.x, self.y + other.y)
9           else:
10              raise TypeError(
11              "Unsupported operand type for +: "+str((type(other))
12              )
13
14  # Main code
15  v1 = Vector(1, 2)
16  v2 = Vector(3, 4)
17  v3 = v1 + v2
```

Which of the following statement(s) about the implementation ('code') above are **true**:

☐ The __add__ is wrongly named. In order to overload the plus operator, the function should be called __+__ instead.

☐ The overload of the + operator has been correctly implemented, the result will be a vector with coordinates (4, 6).

☐ The overload of the + operator has been correctly implemented, the result will be a vector with coordinates (3, 8).

☐ The implementation is wrong since the other variable used in the __add__ function is defined outside the function scope, therefore being not visible inside the function.

Solution

☐ The `__add__` is wrongly named. In order to overload the plus operator, the function should be called `__+__` instead.

☑ The overload of the + operator has been correctly implemented, the result will be a vector with coordinates (4, 6).

☐ The overload of the + operator has been correctly implemented, the result will be a vector with coordinates (3, 8).

☐ The implementation is wrong since the `other` variable used in the `__add__` function is defined outside the function scope, therefore being not visible inside the function.

- The correct naming, as described in the python documentation is indeed `__add__` and not `__+__`

- The code correctly overrides the `+` operator for objects belonging to the `Vector` class

- As can be seen in line 8 the returned vector has coordinates that are the sum of the X coordinates and the sum of the Y coordinates. According to the Vectors defined at line 15 and 16, the sum of those values are 4 for the Xs (1+3) and 6 for the Ys (2+4).

- The `other` variable in this case refers to the formal parameter `code` (the one in the function signature) which is represented by the actual parameter `v2` (the one used in the function call). The `other` variable refers to the correct Vector.

1.2 Task 1: Bubble sort

The code implements the Order class. An Order will contain 3 fields: id, selection_time, shipping_time . The $=, <, >$ operators have been overloaded. A bubble sort algorithm has been implemented, the bubble sort algorithm is wrapped in the sort method that parses the data and creates Order objects to pass to the bubble sort core method.

Pseudocode

```
1  Define Order class
2      Define init(id, selection time, shipping time)
3          Assign id to field id
4          Assign selection time to field selection time
5          Assign shipping time to field shipping time
6      Define eq(other)
7          Return true if this object and other object have same fields value
                else false
8      Define gt(other):
9          Return true if this object fields sum is greater than other object
                fields sum else false
10
11 Define bubbleSort(array)
12     for each item in the array with i being the current index
13         for each element but the last i
14             swap current element with the following if the following is
                    smaller than the current
15
16 Define sort(data)
17     for each triplet in data
18         append order initialized with the triplet to the array
19     bubbleSort(array)
20
21     return list of [id of each order in array]
```

Code

```
1  class Order:
2      def __init__(self, id, selection_time, shipping_time):
3          self.id: int = id
4          self.selection_time: int = selection_time
5          self.shipping_time: int = shipping_time
6
7      def __eq__(self, other):
8          return (self.selection_time + self.shipping_time) == (other.
               selection_time + other.shipping_time)
9
10     def __gt__(self, other):
11         return (self.selection_time + self.shipping_time) > (other.
               selection_time + other.shipping_time)
12
13
14 def bubbleSort(array: List[Order]):
15     n = len(array)
16
17     for i in range(n-1):
```

```
18
19            for j in range(1, n—i—1):
20
21                if array[j] > array[j+1]:
22                    array[j], array[j+1] = array[j+1], array[j]
23
24 def sort(data: List[Tuple[int,int,int]]) —> List[int]:
25     arr: List[Order] = []
26
27     for id, selection_t, shipping_t in data:
28         arr.append(Order(id, selection_t, shipping_t))
29
30     bubbleSort(arr)
31
32     return [a.id for a in arr]
```

Which of the following statement(s) about the implementation ('code') above are **true**:

☐ On line 17, `range(n-1)` prevents the i index to traverse all the elements leaving the last element untouched.

☐ The implementation is correct and contains no bug.

☐ The j index is misused since it avoids comparing the first item of the list with the second.

☐ On line 22, a support variable should be used to swap the array values, otherwise the result could be non deterministic.

Solution

☑ On line 17, `range(n-1)` prevents the `i` index to traverse all the elements leaving the last element untouched.

☐ The implementation is correct and contains no bug.

☑ The `j` index is misused since it avoids comparing the first item of the list with the second.

☐ On line 22, a support variable should be used to swap the array values, otherwise the result could be non deterministic.

- The `range` in python includes the first item (0 by default), but excludes the last item. Since the last item of a list has index = the length of the list-1, we have to pass the length of the list to range since it will not reach it, but we'll stop at the length - 1, which is our last index. Using `n-1` will therefore prevent the last element to be ever visited.

- The implementation is not correct due to the errors highlighted in the previous and in the following points.

- being `j` in the `range(1, n-i-1)` it will start from 1 instead of 0, causing the first element of the list to never be compared with the second.

- In python we are allowed to use this kind of object swap without relying on support variables. Due to the order the expression is evaluated, in the assignment cases python always evaluates the right-hand side of the expression first.

1.3 Task 2: Merge sort

The code implements the Order class just like in the previous example. We did not copy the Order class implementation, you can assume it's correct. The difference from the previous Task Order implementation is that there is a `Order.next` field in this class which represent the following Order (a pointer to the next element, just like in linked lists). The code implements a bottom up merge sort. The final `sort` method used in the previous Task code has been omitted, focus on the core merge sort algorithm instead.

Pseudocode

```
1   # Assume that the objects have a .next field that represent the following
        element, None if there's none
2
3   Define merge_sort(head)
4       if head is empty return None
5
6       arr = list of 32 Nones
7       result, next = None
8
9       result = head
10
11      # Merge nodes into array
12      while result is not null:
13          Save the reference to the result next node into next
14          Reset the result to None
15          For each value of arr
16              If current value is None then break
17              result = merge(current value, result)
18              Assign null to current value
19          Assign result to the second last value of arr
20          result = next
21
22      # Merge array into single list
23      result = null
24      For each item in arr
25          result = merge(current value, result)
26
27      return result
28
29  Define merge(left, right)
30      result = None
31      ref = None
32
33      While left is not null and right is not null:
34          If right greater or equal to left
35              If result is not null then append left to the result
36              else assign left to the result
37              Assign the value after left to left
38          else
39              If result is not null then append right to the result
40              else assign right to the result
41              Assign the value after right to right
42
43      While there are elements remaining in left
44          Append left to result
```

```
45          Assign the value after left to left
46
47      While there are elements remaining in right
48          Append right to result
49          Assign the value after right to right
50
51      Return the pointer to the start of the list
```

Code

```python
1  def merge_sort(head: Union[Order,None]) -> Union[Order,None]:
2      if not head:
3          return None
4      arr: List[Union[Order,None]] = [None for _ in range(32)]
5      result: Union[Order,None] = None
6      next: Union[Order,None] = None
7      result = head
8
9      while result:
10         next = result.next
11         result.next = None
12         i = 0
13         for i in range(32):
14             if arr[i] is None:
15                 break
16             result = merge(arr[i], result)
17             arr[i] = None
18
19         arr[i] = result
20         result = next
21
22     result = None
23     for i in range(32):
24         result = merge(arr[i], result)
25
26     return result
27
28
29 def merge(left: Union[Order,None], right: Union[Order,None]) -> Union[Order,
       None]:
30     result: Union[Order,None] = None
31     ref: Union[Order,None] = None
32
33     while left and right:
34         if left <= right:
35             if result:
36                 result.add_next(left)
37                 result = left
38             else:
39                 ref = left
40                 result = left
41
42             left = left.next
43
44         else:
```

8

```
45              if result:
46                  result.add_next(right)
47                  result = right
48              else:
49                  ref = right
50                  result = right
51
52              right = right.next
53
54      while left:
55          if result:
56              result.add_next(left)
57              result = left
58          else:
59              ref = left
60              result = left
61
62          left = left.next
63
64      while right:
65          if result:
66              result.add_next(right)
67              result = right
68          else:
69              ref = right
70              result = right
71
72          right = right.next
73
74      return ref
```

Which of the following statement(s) about the implementation ('code') above are **true**:

☐ At line 15, we should not `break` but instead `return result`, otherwise the code will never return once we are at the end.

☐ The implementation is correct and contains no bug.

☐ The code will only run for array of length divisible by 32.

☐ At line 35 we cannot use `while left and right:` since they are lists and will never have `True` or `False` as a value.

Solution

☐ At line 15, we should not `break` but instead `return result`, otherwise the code will never return once we are at the end.

☑ The implementation is correct and contains no bug.

☐ The code will only run for array of length divisible by 32.

☐ At line 35 we cannot use `while left and right:` since they are lists and will never have `True` or `False` as a value.

- We need the break because we need to move on since we reached the end of the array. Returning at the point in time would not guarantee that the list got sorted. Break allows us to exit the for loop.

- This represents an indeed correct implementation of merge sort.

- The code simply partitions the list in sublist of max 32 elements, the code will work even if the last sublist has more than 32 elements, making this code effective also for lists with a length not divisible by 32.

- All collection objects in python (lists, dictionaries, sets...) have a boolean value which is determined by having or not elements in it. Their boolean value will be true if they have at least one element in them. In this case the loop goes on as long as both `left` and `right` have at least one element in each of them.

1.4 Task 3: Quick sort

The code implements the Order class just like in the previous examples. We did not copy the Order class implementation, you can assume it's correct. The code implements a quick sort.

Pseudocode

```
1   Define swap(arr, i, j)
2       Swap objects between the indexes i and j of the arr list
3
4   Define quicksort(arr, lo, hi)
5       If lo and hi are greater or equal to 0 and if lo is less than hi
6           Partition the array
7           Call the quicksort on the right partition
8           Call the quicksort on the left partition
9
10  Define partition(arr, lo, hi)
11      Pivot object is the object in the middle of the list
12
13      Set one counter (i) to the index before the lower index
14      Set one counter (j) to the index after the upper index
15
16      Loop forever
17          Update i by decreasing it by one
18          While the value in arr at index i is less than the pivot
19              Increase i (so that we are consider the next element in arr)
20          Update j by decreasing it by one
21          While the value in the arr at index j is greater that the pivot
22              Increase j (so that we are consider the previous element in arr)
23
24          If the indexes i and j have crossed (i>=j) then return j
25
26          Swap the element at index i with the one at index j
27
28  Define sort(data)
29      Create an empty support list
30
31      Fill the support list with the objects in data, converting them into an
            Order object
32
33      Call quicksort on the list
34
35      Return the id of every Order in the list
```

Code

```python
1   def swap(arr: List[Order], i: int, j: int):
2       arr[i], arr[j] = arr[j], arr[i]
3
4   def quicksort(arr: List[Order], lo: int, hi: int):
5       if lo >= 0 and hi >= 0 and lo < hi:
6           p = partition(arr, lo, hi)
7           quicksort(arr, p + 1, hi)
8           quicksort(arr, 0, p)
9
```

```
10  def partition(arr: List[Order], lo: int, hi: int):
11      pivot = arr[(hi+lo)//2]
12
13      i = lo
14      j = hi
15
16      while(True):
17
18          i = i + 1
19          while arr[i] < pivot:
20              i = i + 1
21
22          j = j - 1
23          while arr[j] > pivot:
24              j = j - 1
25
26          if i >= j:
27              return j
28
29          swap(arr, i, j)
30
31
32  def sort(data: List[Tuple[int,int,int]])->List[int]:
33      arr: List[Order] = []
34
35      for id, selection_t, shipping_t in data:
36          arr.append(Order(id, selection_t, shipping_t))
37
38      quicksort(arr, 0, len(arr)-1)
39
40      return [a.id for a in arr]
```

Which of the following statement(s) about the implementation ('code') above are **true**:

☐ The implementation is correct and contains no bug.

☐ At line 2, the value swap between two variables cannot be done in that manner. It is necessary to rely on a temporary variable.

☐ At lines 12 and 13 the i and j indexes should be initialized as i = lo-1 and j = hi+1

☐ At lines 12 and 13 the i and j indexes should be initialized as i = lo+1 and j = hi-1

☐ At line 10, the pivot should be set as pivot = arr[lo], else the algorithm will not be able to sort correctly.

Solution

☐ The implementation is correct and contains no bug.

☐ At line 2, the value swap between two variables cannot be done in that manner. It is necessary to rely on a temporary variable.

☑ At lines 13 and 14 the `i` and `j` indexes should be initialized as `i = lo-1` and `j = hi+1`

☐ At lines 13 and 14 the `i` and `j` indexes should be initialized as `i = lo+1` and `j = hi-1`

☐ At line 11, the pivot should be set as `pivot = arr[lo]`, else the algorithm will not be able to sort correctly.

- The implementation is not correct due to the point that will be explained later.

- As previously mentioned: in python we are allowed to use this kind of object swap without relying on support variables. Due to the order the expression is evaluated, in the assignment cases python always evaluates the right-hand side of the expression first.

- The `i` and `j` variables need to be set to `i = lo-1` and `j = hi+1` since once in the loop they will be respectively increased and decrease at every step to move towards the center. Since we want a single loop to run instead of one loop for the first step and a second loop for all the other steps, we need to set the variable to that "correct" values. It is a simple coding trick that allows us to have a single loop, it is not really something specific about quicksort. If they are set to `i = lo` and `j = hi` then the first and last value will not be checked since in the while loop those values are then increased and decreased.

- For the previous point, we know that the lower index must be decreased before starting the loop and the higher index must be increased, because in the while loop they'll be then increased and decreased again.

- The pivot of the array can be any element in the array. The best pivot point would be the one that has items that are less than itself on the left, and greater than itself items on the right. The middle point of the array is often used as pivot since reduces the cases in which all the elements needs to be moved (if you pick the first item as pivot you'll have to move many elements unless the pivot you picked is the smallest number). But any index can be picked as pivot, non affecting the correctness of the final solution.

1.5 Task 4: Quick sort

The code implements the Order class just like in the previous example. We did not copy the Order class implementation, you can assume it's correct. The code implements a quick sort with out-of-place sorting since smaller and greater elements than the pivot are moved into new separate lists and not swapped within the starting list. In-place sorting would consist in moving the elements within the same original list, like we've done in the previous task. This process makes the code easier to understand but less memory efficient.

Pseudocode

```
1   Define swap(arr, i, j)
2       Swap values of arr at index i and j
3
4   Define quicksort(ar)
5       If there is one or less elements return ar
6
7       Pick the middle element as pivot
8       Create two empty lists, one for objects smaller and one for objects
            greater than the pivot
9       If you are analyzing the pivot then skip to the next object in list
10      Put elements greater or same as the pivot in the list for the greater
            objects
11      Put elements smaller than the pivot in the list fot the smaller objects
12
13      Set as the left list the quicksorted list with objects smaller than the
            pivot
14      Set as the right list the quicksorted list with objects greater than the
            pivot
15      Return the left list, appending the pivot, appending the right list
```

Code

```
1   def swap(arr: List[Order], i: int, j: int):
2       tmp = arr[i]
3       arr[i] = arr[j]
4       arr[j] = tmp
5
6   def quicksort(ar: List[Order]):
7
8       if len(ar) <= 1:
9           return ar
10
11      mid = len(ar)//2
12      pivot = ar[mid]
13
14      smaller,greater = [],[]
15
16      for i, val in enumerate(ar):
17          if i == mid:
18              continue
19          if val < pivot:
20              smaller.append(val)
21          elif val > pivot:
22              greater.append(val)
```

```
23
24      left  = quicksort(smaller)
25      right = quicksort(greater)
26      return left+[pivot]+right
```

Which of the following statement(s) about the implementation ('code') above are **true**:

☐ The implementation is correct and contains no bug.

☐ The case of `val == pivot` is not specified, for this reason all the values equal to the pivot will be lost at every iteration.

☐ The `pivot` is not included in the left sub-list nor the right sub-list, therefore its value will be lost during the execution.

☐ The variable `mid` defined at line 11 may be a decimal number when `len(ar)` is an odd number. Thus causing the code to fail, since we cannot use a decimal value as array index.

1.6 Solution

☐ The implementation is correct and contains no bug.

☑ The case of `val == pivot` is not specified, for this reason all the values equal to the pivot will be lost at every iteration.

☐ The `pivot` is not included in the left sub-list nor the right sub-list, therefore its value will be lost during the execution.

☐ The variable `mid` defined at line 11 may be a decimal number when `len(ar)` is an odd number. Thus causing the code to fail, since we cannot use a decimal value as array index.

- The implementation is not correct due to the following reason.

- The behaviour when `val == pivot` is not specified in the if. This poses no problem for the pivot itself since it is added in the return statement at the end of the function. The problem concerns other objects that have the same value as the pivot but a different position in the index. No condition is defined in this case therefore all objects equal to the pivot (but not the pivot) will be lost at every iteration. (This is a very nasty error, I must admit it. Kudos to the ones who spotted it.)

- The code `mid = len(ar)//2` will never return a decimal number since the `//` operand is a division with flooring operand. This means that after dividing the number the operand rounds the result by flooring it (removing the decimal part and keeping the integer only).

1.7 Task 5: Quick sort

The code implements the Job class just like in Assignment 1 Task 3. In the Job class, `p` stands for the processing time, `w` represents the importance. When a Job finishes at time `t` then the total cost is `t*w`. The goal is to sequence the jobs in order to minimize the total cost of all the jobs. It uses Smith's rule, that is, schedule the jobs in the order of their ratio of processing time over weight. This greedy rule turns out to be optimal. The code implements a quick sort. This time, differently from the Assignment, we don't care if the order of equal jobs is the same both in the input and the output, if the jobs are equal according to `==` then they can be in whatever order.

Pseudocode

```
 1  Define the Job class
 2      Define the Job constructor(self, id, p, w)
 3
 4      Override the greater or equal operator(self, other)
 5          Return the result of self.p/self.w >= other.p/self.w
 6
 7      Override the less than operator(self, other)
 8          Return the result of self.p/self.w < other.p/self.w
 9
10      Override the greater than operator(self, other)
11          Return the result of self.p/self.w > other.p/self.w
12
13  Define quicksort(ar)
14      If there is one or less elements return ar
15
16      Pick the middle element as pivot
17      Create two empty lists, one for objects smaller and one for objects
            greater than the pivot
18      If you are analyzing the pivot then skip to the next object in list
19      Put elements greater than the pivot in the list for the greater objects
20      Put elements smaller than the pivot in the list for the smaller objects
21      Put elements equal to the pivot in the greater or smaller list based on
            their index
22
23
24      Set as the left list the quicksorted list with objects smaller than the
            pivot
25      Set as the right list the quicksorted list with objects greater than the
            pivot
26      Return the left list, appending the pivot, appending the right list
27
28  Define sort(data)
29      Define an empty array of jobs
30
31      For every value in data
32          Create a Job object starting from the three individual values (id, p
                for processing time, w for importance) and add it to the list of
                jobs
33
34      Run quicksort on the array
35
36      Return every id of the job in the sorted array
```

Code

```python
1   class Job:
2       def __init__(self, id, p, w):
3           self.id: int = id
4           self.p: int = p
5           self.w: int = w
6
7       def __ge__(self, other):
8           return ((self.p / self.w) >= (other.p / other.w))
9
10      def __lt__(self, other):
11          return ((self.p / self.w) < (other.p / other.w))
12
13      def __gt__(self, other):
14          return ((self.p / self.w) > (other.p / other.w))
15
16
17  def quicksort(ar: List[Job]):
18
19      if len(ar) <= 1:
20          return ar
21
22      mid = len(ar)//2
23      pivot = ar[mid]
24
25      smaller,greater = [],[]
26
27      for i, val in enumerate(ar):
28          if i == mid:
29              smaller.append(val)
30          if val < pivot:
31              smaller.append(val)
32          elif val > pivot:
33              greater.append(val)
34
35          else:
36              if i < mid:
37                  greater.append(val)
38              else:
39                  smaller.append(val)
40
41      left  = quicksort([smaller])
42      right = quicksort([greater])
43      return left+[pivot]+right
44
45
46  def sort(data:List[Tuple[int,int,int]])->List[int]:
47      arr: List[Job] = []
48
49      for id, p, w in data:
50          arr.append(Job(id, p, w))
51
52      arr = quicksort(arr)
53
54      return [a.id for a in arr]
```

Which of the following statement(s) about the implementation ('code') above are **true**:

☐ The if statement at line 36, appends jobs equal to the pivot into the wrong list, preventing the correct sorting of the list.

☐ The implementation is correct and contains no bug.

☐ The code will duplicate the pivot value of the array passed to the `quicksort` method.

☐ The `greater` and `smaller` lists passed to `quicksort` will not be correctly unpacked, preventing the correct execution.

1.8 Solution

☐ The if statement at line 36, appends jobs equal to the pivot into the wrong list, preventing the correct sorting of the list.

☐ The implementation is correct and contains no bug.

☑ The code will duplicate the pivot value of the array passed to the `quicksort` method.

☑ The `greater` and `smaller` lists passed to `quicksort` will not be correctly unpacked, preventing the correct execution.

- Appending the pivot to one of the two sides isn't an error itself, it is an error adding it to a list and also returning it in the return statement, but doing only one of those is not an error itself. Also there is not a wrong list, the pivot can be appended to any list.

- The implementation is not correct for the reasons explained in the next points.

- The code will indeed duplicate the pivot object since it will be both placed in the `smaller` list and also returned in the return statement. In this way every time the function returns, it will return a duplicated pivot.

- The lists passed as `left = quicksort([smaller])` and its right counterpart will not be correctly unpacked since they're surrounded by brackets. This will make the list be the only element of another list, and when the quicksort method will run it will not be able to find multiple elements, the only found element will be a list of objects instead of an object.

1.9   Questions

Which of the following statement(s) are true:

☐ There is no point in using quicksort over mergesort since quicksort worst case scenario is worse than the mergesort one ($O(n^2)$ versus $O(n \log(n))$).

☐ Recursive functions use more memory that their respective iterative counterpart.

☐ There's no difference between the way we choose the pivot in quicksort.

Solution

☐ There is no point in using quicksort over mergesort since quicksort worst case scenario is worse than the mergesort one ($O(n^2)$ versus $O(n \log(n))$).

☑ Recursive functions use more memory that their respective iterative counterpart.

☐ There's no difference between the way we choose the pivot in quicksort.

- One of the most common disadvantages of mergesort is its memory usage. Since we need a second array as bog as the one we need to sort in order to accomplish the sorting. For this and other reasons quicksort may be chosen instead of mergesort.

- Recursive functions call the same function multiple times, every time the function is called it ends up in a function stack until the function returns. Recursive function recursively call themselves, staying all on the stack until the first function return statement is resolved (when the base case is reached). This causes multiple functions to lay on the stack for a while significantly increasing the memory consumption compared to iterative functions. Iterative functions consist in one function that loop through items, but doesn't call other function, therefore not creating other clutter on the function call stack.

- The way we choose the pivot in quicksort is extremely important and statistically speaking choosing a random pivot proves to be the best choice overall. Choosing a pivot that requires the least swaps to have all greater and smaller items in the right places is the ideal thing.

2 Assignment 2

2.1 Task 1

In this exercise we will implement a linked list and and its remove method, used to remove the first occurrence of an object.

Pseudocode

```
1   Define class Node
2       Define init(value)
3           Set field value to value
4           Set field next to None
5
6   Define class LinkedList
7       Define init
8           Set head field to None
9
10      Define append(value)
11          Create a new node starting from the value parameter
12          If there is no value in the list, then set the new node to the head
                field
13          else
14              Starting from the head
15              Reach the last node in the list
16              Attach the new node, referencing it in the next field of the
                    current last value
17
18      Define remove(value)
19          If there is no node, terminate
20          If the first element is the one to remove, then set the head to the
                second node
21          Set the head as current node
22          While there is a next value in the next field of the current node
23              If the value of the next item is the one to remove then
24                  Set the next field to reference the node after the next node
25                  Terminate
26              Set the current node to the next node
```

Code

```python
1   class Node:
2       def __init__(self, value):
3           self.value = value
4           self.next = None
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9
10      def append(self, value):
11          new_node = Node(value)
12          if self.head is None:
13              self.head = new_node
14          else:
```

```
15              current = self.head
16              while current.next:
17                  current = current.next
18              current.next = new_node
19              new_node.next = current
20
21      def remove(self, value):
22          if self.head is None:
23              return
24          if self.head.value == value:
25              self.head = self.head.next
26              return
27          current = self.head
28          while current.next:
29              if current.next.value == value:
30                  current.next = current.next.next.next
31                  return
32              current = current.next
```
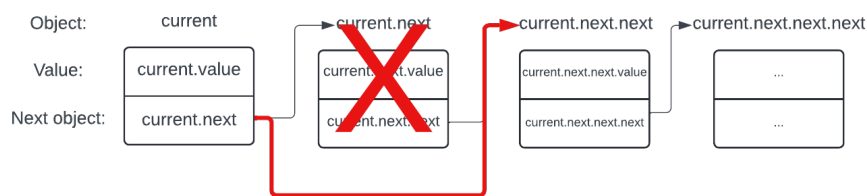
Which of the following statement(s) about the implementation above are **true**:

☐ This is a correct implementation of a linked list and its append and remove methods

☐ On line 30, assigning `current.next.next.next` would cause the method to remove the correct value but also the value after it.

☐ The `append` method is bugged, since after adding a new item to a non empty list we set the last item's `next` field to the second last item, creating a loop among the two last nodes.

☐ The `__init__` method in the `LinkedList` class is bugged, since we cannot have an empty list because then we have no previous item to attach new nodes to.

Solution

☐ This is a correct implementation of a linked list and its append and remove methods

☑ On line 30, assigning `current.next.next.next` would cause the method to remove the correct value but also the value after it.

☑ The `append` method is bugged, since after adding a new item to a non empty list we set the last item's `next` field to the second last item, creating a loop among the two last nodes.

☐ The `__init__` method in the `LinkedList` class is bugged, since we cannot have an empty list because then we have no previous item to attach new nodes to.

- This is not a correct implementation for the errors mentioned in the following points.

- Using `current.next = current.next.next.next` means that we are redirecting the next object point not to the following object (`current.next`) (that we want to delete), not to the second following (`current.next.next`), but to the third following (`current.next.next.next`) which is too many deleted objects, we only wanted one to be deleted. This is what we should have done:

- After reaching the end of the list we assign the new node to the next field of the last object at line 18. After doing so, at line 19, we assign the previously last node to the next field of the new last item (the one we've just appended). This is clearly wrong because the last node's next value should be set to now. Therefore, the last item is pointing to the second last node. Instead, the last node's next field should be set to None since there is no node after the last node.

- Empty lists are often used and when we do not have any node, we simply use None. When a new node is pushed into the list, then we assign it to the node head, with no previous node.

2.2 Task 2

This tasks implements a binary search algorithm in a recursive fashion, we can assume that the arr array is sorted.

Pseudocode

```
1  Define binary_search(arr, x)
2      If the array is empty
3          Return −1
4      Set mid var to half of the array length
5      If the middle value of the array is the searched value, then return the
           middle index
6      Else if the middle value is smaller than the searched value
7          then set the result to binary_search(subarray starting from mid+1
               onward, x)
8          Return the result+index of the middle index+1, if result is −1 return
               −1
9      else return the result of binary_search(subarray from index 0 to mid−1, x)
```

Code

```python
1  def binary_search(arr, x):
2      if not arr:
3          return −1
4
5      mid = len(arr) // 2
6
7      if arr[mid] == x:
8          return mid
9      elif arr[mid] < x:
10         result = binary_search(arr[mid+1:], x)
11         return result + mid + 1 if result != −1 else −1
12     else:
13         return binary_search(arr[:mid], x)
```

Which of the following statement(s) about the implementation above are **true**:

☐ The algorithm is sub-optimal, since on line 13, the array passed to binary_search also includes `arr[mid]` which we've already proven not equal to `x`.

☐ The code is correct but using a linked list would reduce the execution time.

☐ The code is correct and wouldn't be faster with a linked list.

☐ Line 11 is not correct Python syntax.

☐ Being a recursive algorithm, if no correct value is found, the algorithm would keep running in a loop.

Solution

☐ The algorithm is sub-optimal, since on line 13, the array passed to binary_search also includes `arr[mid]` which we've already proven not equal to `x`.

☐ The code is correct but using a linked list would reduce the execution time.

☑ The code is correct and wouldn't be faster with a linked list.

☐ Line 11 is not correct Python syntax.

☐ Being a recursive algorithm, if no correct value is found, the algorithm would keep running in a loop.

- The slicing notation in Python allows to quickly generate a sublist. `arr[:mid]` means that the array sublist has to range from the first index up to mid, excluding the last index (mid). The slice notation includes the starting index but excludes the last. Therefore we are not considering `arr[mid]` by returning the sublist `arr[:mid]`.

- Using a linked list would not reduce the execution time. Getting the mid element would take O(n) for a list while it would take only O(1) to an array. Also, creating a new sub array requires O(k) where k is the number of elements to copy, while we would need to traverse the whole list (O(n)) in order to create a sublist from the second half of the list.

- For the aforementioned reasons this code is correct and wouldn't be improved by using a linked list.

- Line 11 is correct and contains a ternary operator. Ternary operators allow to quickly write simple if-else statements in a single line. They follow the syntax: A if CONDITION else B, and it returns A if CONDITION is true, otherwise it returns B. It's often used in return statements and variable assignment.

- Correct recursive algorithms contain a termination condition which in this case is the first if on lines 2 and 3. If the item hasn't been found then it will return -1. In the worst case the algorithm will terminate in $O(\log(n))$.

2.3 Task 3

Heaps are commonly implemented with array, the positioning in the array determines the positioning in the tree, here's an example: [A, B, C, D, E, F] A has 2 leaves (B and C), B has 2 leaves (D and E), C has 1 leaf (F). Considering the lexicographical order this would be a min-heap since the children are always greater than the parent. Here's an implementation of min-heap and some of its main methods.

Pseudocode

```
1  Define class MinHeap
2      Define init()
3          Initialize heap field to an empty list
4
5      Define parent(i)
6          Return the index of the parent of the node at the i—th position
7
8      Define left_child(i)
9          Return the index of the left child of the node at the i—th position
10
11     Define right_child(i)
12         Return the index of the right child of the node at the i—th position
13
14     Define insert(value)
15         Append the new value at the end of the heap
16         Starting from the last element of the heap
17         While the element index is greater than 0 and the current element is
                less than its parent
18             Swap the values of the current element and its parent
19
20     Define get_min()
21         If the heap has elements, then return the element of the heap field at
                index 0
22         else return None
```

Code

```python
1  class MinHeap:
2      def __init__(self):
3          self.heap = []
4
5      def parent(self, i):
6          return (i — 1) / 2
7
8      def left_child(self, i):
9          return 2 * i + 1
10
11     def right_child(self, i):
12         return 2 * i + 2
13
14     def insert(self, value):
15         self.heap.append(value)
16         i = len(self.heap)
17         while i > 0 and self.heap[i] < self.heap[self.parent(i)]:
```

```
18              self.heap[i], self.heap[self.parent(i)] = self.heap[self.parent(i)
                    ], self.heap[i]
19              i = self.parent(i)
20
21      def get_min(self):
22          if len(self.heap) > 0:
23              return self.heap[0]
24          else:
25              return None
```

Which of the following statement(s) about the implementation above are **true**:

☐ The `parent` method is wrong, since it may return a non integer value.

☐ The while at line 18 stops as soon as the parent is greater than the child, but it should instead continue to verify that it is true throughout the whole hierarchy.

☐ The `get_min` method doesn't take into consideration lists with a negative number of values, it could break because of this.

☐ The `insert` method will break raising `IndexError:index out of range`.

Solution

☑ The `parent` method is wrong, since it may return a non integer value.

☐ The while at line 18 stops as soon as the parent is greater than the child, but it should instead continue to verify that it is true throughout the whole hierarchy.

☐ The `get_min` method doesn't take into consideration lists with a negative number of values, it could break because of this.

☑ The `insert` method will break raising `IndexError:index out of range`.

- The `parent` function could return a float value, which is definitely wrong since it needs to return the index of the parent of the i-th item, and indexes must be integer values. The correct operator is `//` which would truncate decimal division results, leading to the correct index being returned.

- There is no need for the while at line 18 to continue looping after having found that the parent is greater than the child. Since we are in a min-heap we know that before adding a new element the min-heap is already correctly ordered. For this reason, once we have that the new item becomes greater than its parent, we are assured that the parent, the grand parent and so on will be already correctly ordered.

- Lists cannot contain a negative number of values; we cannot have less than 0 items in a list.

- Setting `i` to the length of the list after having appended the new element will cause us to subsequently look for an item that doesn't exist at line 17 when we use `self.heap[i]`. List indexes are zero-based, therefore the last index will be equal to the list length minus one. In order to fix the issue we should have done `i = len(self.heap)-1` or swapped line 15 with line 16.

2.4   Questions

Which of the following statement(s) are true:

☐ In a system where the free memory cells are sparse and not contiguous it's better to use a linked list rather than an array.

☐ Popping an item which is not the last element added to a stack is not the proper way to interact with a stack. But we can often do it since stacks are Abstract Data Types.

☐ The graph representations (edges list, adjacency list, adjacency matrix) have no effect on the efficiency of the algorithm we then implement on the graph. They are just a way for the developer to see the graph in a different manner.

2.5 Solution

☑ In a system where the free memory cells are sparse and not contiguous it's better to use a linked list rather than an array.

☑ Popping an item which is not the last element added to a stack is not the proper way to interact with a stack. But we can often do it since stacks are Abstract Data Types.

☐ The graph representations (edges list, adjacency list, adjacency matrix) have no effect on the efficiency of the algorithm we then implement on the graph. They are just a way for the developer to see the graph in a different manner.

- It is indeed better to use a linked list in cases where the free cells are not contiguous, this is due to the fact that a linked list can be made up of elements that are stored in any space in memory, the elements may never be one after the other. Therefore a linked list is better at fully exploiting the sparse free space in the system.

- Stacks allow only the last added item to be popped (they are LIFO structures), in our Python code. For instance, we can call `pop(x)` to pop the xth element since the stack is actually implemented via a list, where we can access any element in any order we want.

- Graph representation can have a big impact on memory consumption and computing times. An adjacency matrix costs a fixed memory space ($nodes^2$) and it is ideal for dense graphs. This is because in the adjacency matrix, for every couple of nodes we have to define whether there's an edge between the two or not. Therefore, the memory consumption is independent from the number of edges, so we will always have ($nodes^2$) number of values (0 for no edge, 1 for edge). The edge list representation is faster for traversing the graph, since we can simply explore all the nodes that share an edge. We just begin from a starting node and cycle through the edge list of the starting node and we do the same for every node we encounter.

3 Assignment 3

3.1 Task 1

In the following code the path from `start` to `end` is going to be searched with a depth-first algorithm. The `graph` passed to the function is an edge list, represented by a dictionary where the key is the vertex and its value is a list of the vertices it is connected to.

Pseudocode

```
1  Define dfs(graph, start, end)
2      Create the stack to track nodes and their paths
3      Put the start node in the stack with itself as the first node of the path
4      While there are values in the stack
5          Pop the stack, returning the starting node and the path
6          If the most recently extracted node is the end node, then return the
               path that was just returned
7          For every neighbour of the most recently returned node
8              If the neighbour is not in the path that was just returned, then
9                  append the neighbour node to the current path, and append the
                      node and the updated path to the top of the stack
10     Return None
```

Code

```
1  def dfs(graph, start, end):
2      stack = [(start, [start])]
3      while stack:
4          node, path = stack[-1]
5          del stack[-1]
6          if node == end:
7              return path
8          for neighbor in graph[node]:
9              if neighbor not in path:
10                 new_index = len(stack)
11                 stack[new_index] = (neighbor, path + [neighbor])
12     return None
```

Which of the following statement(s) about the implementation above are **true**:

☐ The pop at line 4 is not correctly implemented, we must use the `pop()` method instead. That code will prevent the correct popping.

☐ On line 4 and 5 we cannot return and delete the index `-1` since it will raise an `IndexError` not having the index -1 in the list.

☐ On line 11 we cannot append to the list by using the index, we should use the `append()` method instead.

☐ This is not a depth-first search but a breadth-first search. In order to make it a DFS we should start from the bottom, which in this case is the `end` node.

Solution

☐ The pop at line 4 is not correctly implemented, we must use the `pop()` method instead. That code will prevent the correct popping.

☐ On line 4 and 5 we cannot return and delete the index `-1` since it will raise an `IndexError` not having the index -1 in the list.

☑ On line 11 we cannot append to the list by using the index, we should use the `append()` method instead.

☐ This is not a depth-first search but a breadth-first search. In order to make it a DFS we should start from the bottom, which in this case is the `end` node.

- The pop consists in returning the last item of a list and deleting it from the list itself. The `pop()` does it all in one method, in this case it's simply been re-implemented.

- You can indeed get negative indices out of an array, a negative index means that the index count should start from the end of the list. `[-1]` means the last item, `[-2]` means the second last, and so on.

- We cannot refer a position in the list that has never been initialized, since the stack didn't have the `new_index` position assigned (because it was `new_index` but indexing starts from 0, meaning the last index is `new_index-1`). Therefore we need to use the `append()` method to add an item at the end of the list.

- This is indeed a DFS algorithm since we always look at the current node's children first. We do not need to "start from the bottom" in order to have a DFS.

3.2   Task 2

The following code implements a Breadth-First Search, it simply looks whether it is possible to reach the destination node starting from the source node.

Pseudocode

```
1  Define bfs(graph, start, target)
2      Create a set for the visited nodes
3      Create a queue with the start node
4      While there are nodes in the queue
5          Set the current node to the first node in the queue
6          If the current node is the target then return true
7          Add the current node to the visited set
8          For every neighbour of the current node
9              If the neighbour is not in the visited set add it to the queue
10     Return false
```

Code

```python
1  def bfs(graph, start, target):
2      visited = set()
3      queue = [start]
4
5      while queue:
6          current_node = queue.pop(0)
7          if current_node == target:
8              return True
9
10         visited.add(current_node)
11
12         for neighbor in graph[current_node]:
13             if neighbor not in visited:
14                 queue.append(neighbor)
15
16     return False
```

Which of the following statement(s) about the implementation above are **true**:

☐ The code will not work in case the graph is cyclical.

☐ The pop method does not have a parameter, it can just pop the last element.

☐ The method correctly implements a Breadth-First Search.

☐ The method would work only for directed graphs and not for undirected graphs.

Solution

☐ The code will not work in case the graph is cyclical.

☐ The pop method does not have a parameter, it can just pop the last element.

☑ The method correctly implements a Breadth-First Search.

☐ The method would work only for directed graphs and not for undirected graphs.

- The code works in both cyclical and acyclical graphs, nodes in cycles will not be visited twice since they are stored in the visited set and filtered at line 13.

- Even though the pop concept is derived from stacks where only the last element can be popped, in Python lists the pop method can take a parameter, which is the index of the element to remove and then delete. If no parameter is passed, the last item will be deleted just like in the classic pop concept.

- It does implement a BFS, it checks all nodes at the same depth before moving onto the deeper layer.

- Being directed or non directed is irrelevant since the edges would be represented in the edge list. In non directed graphs there will be an item in the list of two vertices instead of just one.

3.3 Task 3

The code computes the shortest distance between the `start_id` node and any other node in the graph. The algorithm used is Dijkstra's algorithm.

Pseudocode

```
1   Define dijkstra(graph, start_id)
2       Create a distances dictionary, the values are the distance between the
            start_id node and the key (a node id)
3       Create a dictionary to store the parent of vertices found along the
            shortest path
4       Create a queue for vertices
5
6       For every vertex in the graph
7           Set its parent to -1
8           Add the node to the queue
9           Set the distance to infinity for all nodes, start_id distance set to 0
10
11
12      While there are values in the queue
13          Set the smallest known distance to infinity
14          Set the node the distance refers to, to the next smallest node in the
                queue (later called the current node)
15
16          The following for loop looks for the node with the shortest distance
                in the queue
17          For every node in the queue
18              If the distance between the start_id and the node is less than the
                    smallest known distance, then set the current node to this
                    node, together with its distance
19          Remove the current node from the queue
20
21          Now update the distances of the neihboring vertices passing by the
                current node
22          For every neighbour of the current node that is still in the queue
23              The distance from the neighbour to start_id passing by the current
                    node is the distance of the current node from start_id + the
                    distance of the current node and the neighbour
24              If this new distance is lower than the previous distance of the
                    neighbour from start_id
25                  Then update the distance in the distances dictionary
26                  In the parents dictionary, set the current node as the current
                        neighbor parent
27
28      Return the distances and parent dictionaries
```

Code

```
1   def dijkstra(graph, start_id: int) -> Tuple[dict, dict]:
2       dist = {}
3       parent = {}
4       queue = []
5       for v in graph:
6           parent[v] = -1
```

```
7              queue.append(v)
8              dist[v] = float('inf') if v != start_id else 0
9
10     while queue:
11             u_val: float = float('inf')
12             u: Node = queue[0]
13             for q in queue:
14                 u_val, u = (dist[q.id], q) if dist[q.id] > u_val else (u_val, u)
15             queue.remove(u)
16
17             children = [i for i in u.children if i in queue]
18             for v in children:
19                 alt = v.weight
20                 if alt < dist[v.id]:
21                     dist[v.id] = alt
22                     parent[v.id] = u.id
23     return dist, parent
```

Which of the following statement(s) about the implementation above are **true**:

☐ The ifs on line 8 and 14 are not a correct Python syntax.

☐ On line 14 a less than (<) should be used, since we want to get the node which distance is the shortest.

☐ On line 19, the new distance should be `dist[u.id] + v.weight` instead of only the last node weight (which is what we often call distance in this scenarios).

☐ This is a correct implementation of Dijkstra's algorithm.

Solution

☐ The ifs on line 8 and 14 are not a correct Python syntax.

☑ On line 14 a less than (`<`) should be used, since we want to get the node which distance is the shortest.

☑ On line 19, the new distance should be `dist[u.id] + v.weight` instead of only the last node weight (which is what we often call distance in this scenarios).

☐ This is a correct implementation of Dijkstra's algorithm.

- The syntax is indeed admitted, it is often called a ternary operator. It is just a shorthand for an if-else statement in python, just like in many other languages.

- It is true, if we need to find the closest node to `start_id`, we need to update `u` and `u_val` with a value that is less that the current distance, not greater.

- The distances in the `dist` dictionary represent the distances from the `start_id`, therefore we need to get the distance to the previous point (`u`) plus the distance between `u` and `v`, in order to get the whole path from `start_id` to `v`.

- Due to the 2 previous statements this is not a correct Dijkstra's algorithm implementation, by solving those two issues it would become one.

3.4 Questions

Which of the following statement(s) are true:

☐ When looking for an element in a tree we must perform both a BFS and a DFS since the BFS will look for the items at the same level of the starting point and the DFS will look for the items on the branch starting from the starting point. Using only one of them may exclude some items from the search.

☐ Dijkstra's algorithm can find the shortest path only for acyclical graphs, if cycles are in the graph then we need to use A* that will instead follow its heuristic without being tricked into the cycles.

☐ Dijkstra's algorithm runs on an unweighted graph (every edge has the same weight), works just like a classic BFS search.

3.5 Solution

☐ When looking for an element in a tree we must perform both a BFS and a DFS since the BFS will look for the items at the same level of the starting point and the DFS will look for the items on the branch starting from the starting point. Using only one of them may exclude some items from the search.

☐ Dijkstra's algorithm can find the shortest path only for acyclical graphs, if cycles are in the graph then we need to use A* that will instead follow its heuristic without being tricked into the cycles.

☑ Dijkstra's algorithm runs on an unweighted graph (every edge has the same weight), works just like a classic BFS search.

- BFS and DFS look for items within a tree. The way they operate is similar but they simply follow different paths, DFS explores the tree vertically first while BFS explores it breadth first. Regardless of the order they'll both check out every element, so using one of them is totally fine, we do not need to use both at the same time.

- Dijkstra's algorithm works also in cyclical graphs, since it will not revisit the exact same path twice. But it could visit the same node multiple times to try find a shorter path to it.

- Dijkstra's algorithm works just like BFS when no difference in weights between the edges is found. This is because Dijkstra is going to take the next node with the shortest distance from the source. Dijkstra is essentially a BFS with respect to the edge weights rather than the topology (BFS is guided by topology, not edge weights).