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

```
Define Vector class
    Define init(x, y)
        Assign x to field x
        Assign y to field y
    Define add(other)
        If other is instance of class Vector
            then return the summed Vector
        else
            raise TypeError

# Main code
v1 = (1, 2)
v2 = (3, 4)
v3 = v1 + v2
```

Code

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        else:
            raise TypeError(
            "Unsupported operand type for +: "+str((type(other))
            )

# Main code
v1 = Vector(1, 2)
v2 = Vector(3, 4)
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.

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

```
Define Order class
    Define init(id, selection time, shipping time)
        Assign id to field id
        Assign selection time to field selection time
        Assign shipping time to field shipping time
    Define eq(other)
        Return true if this object and other object have same fields value
            else false
    Define gt(other):
        Return true if this object fields sum is greater than other object
            fields sum else false

Define bubbleSort(array)
    for each item in the array with i being the current index
        for each element but the last i
            swap current element with the following if the following is
                smaller than the current

Define sort(data)
    for each triplet in data
        append order initialized with the triplet to the array
    bubbleSort(array)

    return list of [id of each order in array]
```

Code

```
class Order:
    def __init__(self, id, selection_time, shipping_time):
        self.id: int = id
        self.selection_time: int = selection_time
        self.shipping_time: int = shipping_time

    def __eq__(self, other):
        return (self.selection_time + self.shipping_time) == (other.
            selection_time + other.shipping_time)

    def __gt__(self, other):
        return (self.selection_time + self.shipping_time) > (other.
            selection_time + other.shipping_time)


def bubbleSort(array: List[Order]):
    n = len(array)

    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.

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
```

```
        Assign the value after left to left

    While there are elements remaining in right
        Append right to result
        Assign the value after right to right

    Return the pointer to the start of the list
```

Code

```python
def merge_sort(head: Union[Order,None]) -> Union[Order,None]:
    if not head:
        return None
    arr: List[Union[Order,None]] = [None for _ in range(32)]
    result: Union[Order,None] = None
    next: Union[Order,None] = None
    result = head

    while result:
        next = result.next
        result.next = None
        i = 0
        for i in range(32):
            if arr[i] is None:
                break
            result = merge(arr[i], result)
            arr[i] = None

        arr[i] = result
        result = next

    result = None
    for i in range(32):
        result = merge(arr[i], result)

    return result


def merge(left: Union[Order,None], right: Union[Order,None]) -> Union[Order,
    None]:
    result: Union[Order,None] = None
    ref: Union[Order,None] = None

    while left and right:
        if left <= right:
            if result:
                result.add_next(left)
                result = left
            else:
                ref = left
                result = left

            left = left.next

        else:
```

```
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.

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

```
Define swap(arr, i, j)
    Swap objects between the indexes i and j of the arr list

Define quicksort(arr, lo, hi)
    If lo and hi are greater or equal to 0 and if lo is less than hi
        Partition the array
        Call the quicksort on the right partition
        Call the quicksort on the left partition

Define partition(arr, lo, hi)
    Pivot object is the object in the middle of the list

    Set one counter (i) to the index before the lower index
    Set one counter (j) to the index after the upper index

    Loop forever
        Update i by decreasing it by one
        While the value in arr at index i is less than the pivot
            Increase i (so that we are consider the next element in arr)
        Update j by decreasing it by one
        While the value in the arr at index j is greater that the pivot
            Increase j (so that we are consider the previous element in arr)

        If the indexes i and j have crossed (i>=j) then return j

        Swap the element at index i with the one at index j

Define sort(data)
    Create an empty support list

    Fill the support list with the objects in data, converting them into an
        Order object

    Call quicksort on the list

    Return the id of every Order in the list
```

Code

```
def swap(arr: List[Order], i: int, j: int):
    arr[i], arr[j] = arr[j], arr[i]

def quicksort(arr: List[Order], lo: int, hi: int):
    if lo >= 0 and hi >= 0 and lo < hi:
        p = partition(arr, lo, hi)
        quicksort(arr, p + 1, hi)
        quicksort(arr, 0, p)

```

```
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.

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

```
Define swap(arr, i, j)
    Swap values of arr at index i and j

Define quicksort(ar)
    If there is one or less elements return ar

    Pick the middle element as pivot
    Create two empty lists, one for objects smaller and one for objects
        greater than the pivot
    If you are analyzing the pivot then skip to the next object in list
    Put elements greater or same as the pivot in the list for the greater
        objects
    Put elements smaller than the pivot in the list fot the smaller objects

    Set as the left list the quicksorted list with objects smaller than the
        pivot
    Set as the right list the quicksorted list with objects greater than the
        pivot
    Return the left list, appending the pivot, appending the right list
```

Code

```
def swap(arr: List[Order], i: int, j: int):
    tmp = arr[i]
    arr[i] = arr[j]
    arr[j] = tmp

def quicksort(ar: List[Order]):

    if len(ar) <= 1:
        return ar

    mid = len(ar)//2
    pivot = ar[mid]

    smaller,greater = [],[]

    for i, val in enumerate(ar):
        if i == mid:
            continue
        if val < pivot:
            smaller.append(val)
        elif val > pivot:
            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 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

```
class Job:
    def __init__(self, id, p, w):
        self.id: int = id
        self.p: int = p
        self.w: int = w

    def __ge__(self, other):
        return ((self.p / self.w) >= (other.p / other.w))

    def __lt__(self, other):
        return ((self.p / self.w) < (other.p / other.w))

    def __gt__(self, other):
        return ((self.p / self.w) > (other.p / other.w))


def quicksort(ar: List[Job]):

    if len(ar) <= 1:
        return ar

    mid = len(ar)//2
    pivot = ar[mid]

    smaller,greater = [],[]

    for i, val in enumerate(ar):
        if i == mid:
            smaller.append(val)
        if val < pivot:
            smaller.append(val)
        elif val > pivot:
            greater.append(val)

        else:
            if i < mid:
                greater.append(val)
            else:
                smaller.append(val)

    left  = quicksort([smaller])
    right = quicksort([greater])
    return left+[pivot]+right


def sort(data:List[Tuple[int,int,int]])->List[int]:
    arr: List[Job] = []

    for id, p, w in data:
        arr.append(Job(id, p, w))

    arr = quicksort(arr)

    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.

1.7 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.

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

```
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.

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

```
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.

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

```
Define class MinHeap
    Define init()
        Initialize heap field to an empty list

    Define parent(i)
        Return the index of the parent of the node at the i—th position

    Define left_child(i)
        Return the index of the left child of the node at the i—th position

    Define right_child(i)
        Return the index of the right child of the node at the i—th position

    Define insert(value)
        Append the new value at the end of the heap
        Starting from the last element of the heap
        While the element index is greater than 0 and the current element is
            less than its parent
            Swap the values of the current element and its parent

    Define get_min()
        If the heap has elements, then return the element of the heap field at
             index 0
        else return None
```

Code

```
class MinHeap:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i — 1) / 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def insert(self, value):
        self.heap.append(value)
        i = len(self.heap)
        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`.

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.

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.

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

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

Code

```
def bfs(graph, start, target):
    visited = set()
    queue = [start]

    while queue:
        current_node = queue.pop(0)
        if current_node == target:
            return True

        visited.add(current_node)

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                queue.append(neighbor)

    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.

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

```
Define dijkstra(graph, start_id)
    Create a distances dictionary, the values are the distance between the
        start_id node and the key (a node id)
    Create a dictionary to store the parent of vertices found along the
        shortest path
    Create a queue for vertices

    For every vertex in the graph
        Set its parent to -1
        Add the node to the queue
        Set the distance to infinity for all nodes, start_id distance set to 0


    While there are values in the queue
        Set the smallest known distance to infinity
        Set the node the distance refers to, to the next smallest node in the
            queue (later called the current node)

        The following for loop looks for the node with the shortest distance
            in the queue
        For every node in the queue
            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
        Remove the current node from the queue

        Now update the distances of the neihboring vertices passing by the
            current node
        For every neighbour of the current node that is still in the queue
            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
            If this new distance is lower than the previous distance of the
                neighbour from start_id
                Then update the distance in the distances dictionary
                In the parents dictionary, set the current node as the current
                     neighbor parent

    Return the distances and parent dictionaries
```

Code

```
def dijkstra(graph, start_id: int) -> Tuple[dict, dict]:
    dist = {}
    parent = {}
    queue = []
    for v in graph:
        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.

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.