Write your anonymous code (*not* your name):

TDA417 2023 only: Check this box if you have passed the extra-exam 2023-10-12 ☐

Then you can skip questions 1–8. If you don't know what this is about you can safely ignore it.

Basic question 1: Complexity

The below function takes a list of numbers and finds all the products of two different numbers in the list together with their multiplicity (the number of ways of getting that product).

```
function count(xs : LinkedList<integer>) -> Map<integer, integer>
    res = new map
    for x in xs:                                  O(n) iterations
        for y in xs:                              O(n) iterations
            if x != y:                            Worst case always true
                z = x * y
                if not res.containsKey(z)         O(n²) or O(log n)  (x4)
                    res.put(z, 0)
                res.put(z, res.get(z)+1)
    return res
```

What is the asymptotic complexity in terms of the length n of `xs`? Assume that the map is implemented using:

A. a (simple) binary search tree: $O(n^4)$

B. an AVL tree: $O(n^2 \log n)$ or $O(n^2 \log n^2)$

Justify your answer briefly, or (better) by annotating the statements in the code.

See annotations in code.

Grading note: The word "not" was missing in the original code, but that has no impact on complexity. If someone answers O(1) because it crashes at null+1 or $O(n^2)$ because res only ever contains a single element (whatever the result of null+1 is), I suppose we will grudgingly give a point for that.

Basic question 2: Sorting

Once, in an alternate universe, some archaeologists were excavating an ancient tomb with strange scribblings on a wall. Fortunately, you were one of the archaeologists. Since you had taken a course in data structures, you quickly realised that it was an implementation of a well-known sorting algorithm.

However, the tomb was several thousand years old, so parts of the implementation had eroded:

```
def sort(A):
    n = length of A
    for i = 0 to n:
        j = i
        for k = j+1 to n:
            if A[k] < A[j]:
                j = k
        swap A[j] and A[i]
```

What sorting algorithm is this? Selection sort

Help the other archaeologists by filling in the eroded parts of the implementation above.

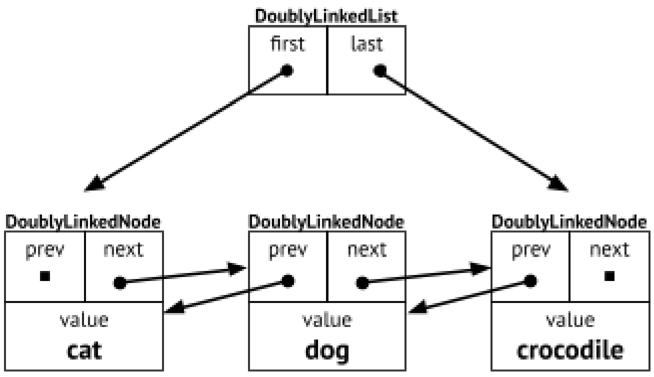
Note: it's important that you get everything right, otherwise the sorting algorithm won't work and the ancient gods will be upset, causing the tomb to collapse. (This is perfectly plausible in the alternate universe.)

Grading note: Swapping the order of A[j] and A[i] is fine. Omitting A in the swap operation is OK, but not in the comparison.

Basic question 3: Basic data structures

A doubly-linked list is a linked list where every list node has a pointer not only to the following element, but also to the previous one. The advantage is that you can iterate over the elements both forwards and backwards. Here's an example implementation with string values:

```
class DoublyLinkedList
    first : DoublyLinkedNode
    last : DoublyLinkedNode

class DoublyLinkedNode
    next : DoublyLinkedNode
    prev : DoublyLinkedNode
    value : String
```

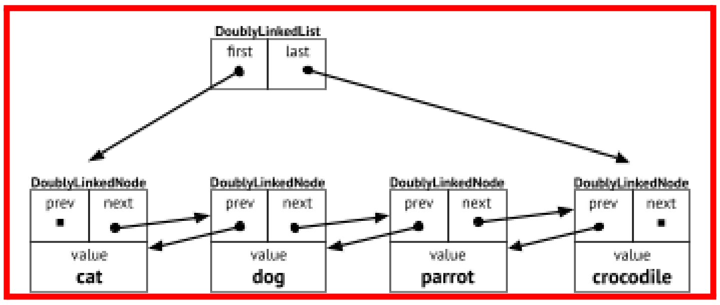


Note that the list points to both the first and last element of the list, to facilitate both forwards and backwards iteration. To the right is a picture of how the list [*"cat", "dog", "crocodile"*] would look.

Give an implementation of the method append, which takes a `DoublyLinkedNode` and a string and inserts the string directly after the given node. For simplicity, you can assume that the given node is ***not the first or the last*** in the list.

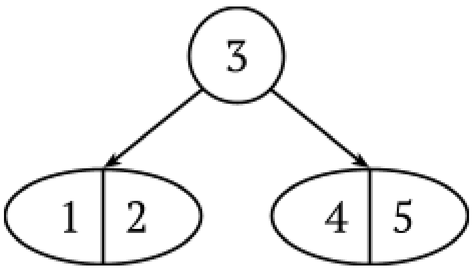
```
def append(nodeBefore : DoublyLinkedNode, str : String):
    newNode = new DoublyLinkedNode(value = str)

    newNode.next=nodeBefore.next

    newNode.prev=nodeBefore

    nodeBefore.next=newNode

    newNode.next.prev=newNode

    ..........................................
```

Show how the example list looks after appending *"parrot"* after the node containing *"dog"*:



Basic question 4: Search trees

Assume the 2-3 tree to the right.



In which order could the elements have been added to the tree? Assume that no elements have been deleted yet.

Check ***all*** the orders that are possible:

- ☐ **A**: 2, 1, 3, 4, 5
- [✓] **B**: 2, 3, 4, 1, 5
- ☐ **C**: 3, 1, 2, 4, 5
- [✓] **D**: 3, 1, 5, 4, 2
- [✓] **E**: 4, 3, 1, 2, 5

Basic question 5: Abstract Data Types

Below is part of a program interacting with a collection data type (initially empty).

```
add(9)
add(7)
x = remove()
add(6)
add(3)
add(5)
y = remove()
z = remove()
```

What are the values of x y and z assuming that:

a) add/remove are push/pop for a stack?

x = 7, y =5, z = 3

b) add/remove are enqueue/dequeue for a queue?

x = 9, y =7, z = 6

c) add/remove are add/removeMin for a priority queue?

x = 7, y = 3, z = 5

Basic question 6: Hash tables

Consider this (partial) implementation of an open addressing hash table using linear probing, with an error in the `resize` function:

```
class HashTable<A> :
    size       : Int = 0
    loadFactor : Double = 0.75
    table      : Array of A with initial size 4

    add(item : A) -> Void
    …

    resize() -> Void :
        if size >= loadFactor * table.length :
            newTable = new Array of A with size table.length * 2

            for i in 0 to table.length :    // up to not including
                newTable[i] = table[i]

            table = newTable
```

The hash table is implemented with a fixed-size array and keeps track of the number of items in the table with the `size` variable. The `loadFactor` is a constant that determines when we need to resize the array.

Give a short text explanation of why this `resize` method does not work (you choose a suitable type A, and hash function for it). The explanation should also include a concrete example (showing the contents of the table) that illustrates the problem. Include both what happens when using the faulty `resize` method above, and what should happen using a correct method.

It does not re-hash the elements.

Example: _ are empty table cells (null). elements are integers hashed by their value % size.

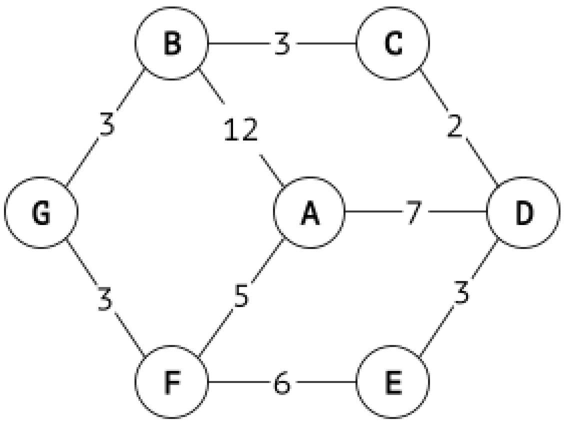
[0,5,2,_] will be resized to [0,5,2,_,_,_,_,_], but the correct table for those elements would be [0,_,2,_,_,5,_,_] since hash(5)%8=5, not =1.

Basic question 7: Graphs

You are given the undirected weighted graph to the right.

Perform uniform-cost search (also known as Dijkstra's algorithm) starting from node A (in the middle of the picture).

In which order does the algorithm visit the nodes, and what is the computed distance to each of them?



	first visited						*last visited*	
node	A	F	D	G	C	E	B	
distance	0	5	7	8	9	10	11	

The last column should be empty, but if you put something there you will still get a point if all the other cells are correct.

Basic question 8: Mystery Data Structure

Below is a simple class implementing a common data structure, and a function that operates on it:

```
class What:
    value : String
    unknown : List<What>

function mystery(x : What) -> Int:
    if x == null
        return 0
    res = 1
    for y in x.unknown:
        res += mystery(y)
    return res
```

Here is an example of a little program using the data structure (assume `unknown` is automatically initialised to an empty list of sufficient capacity for each `What`):

```
x = new What("Animals")
x.unknown[0] = new What("Frogs")
x.unknown[1] = new What("Mammals")
x.unknown[1].unknown[0] = new What("Humans")
x.unknown[1].unknown[1] = new What("Squirrels")
x.unknown[1].unknown[2] = new What("Dogs")
```

What data structure is this? Tree (Directed Acyclic graph would be OK, just directed graph is dubious since then the size function will only work in the absence of cycles). "Binary tree" is incorrect.

What is a better name for the instance variable '`unknown`'? Children

Which property of the data structure does the function '`mystery`' compute?

Size (number of nodes)

Advanced question 9: Complexity

We are given:

- an array (of numbers) of size *n*
- A collection of *k* numbers that we wish to find in the array.

For this, we consider two approaches:

A. Perform a linear search for each of the *k* numbers.

B. Sort the array using merge sort. Then use binary search for each of the *k* numbers.

What is the asymptotic complexity of each approach in terms of *n* and *k*?

A. $O(nk)$

B. $O(n \log n + k \log n) = O((n+k) \log n)$

For which values of *k* is the asymptotic complexity of approach A at least as good as that of approach B? Answer in *O*-notation in terms of *n*.

$k \in O(\log n)$

Explain your reasoning:

Grading note: since it's a bit unclear if this refers to the whole question, just explaining the latter part is sufficient.

For $k \in O(\log n)$ the complexity of A becomes $O(n \log n)$, and the complexity of B becomes $O(n \log n + (\log n)^2) = O(n \log n)$

Advanced question 10: Amortised balanced trees?

A fellow data structures student is fascinated with amortised complexity as seen in dynamic arrays, and wants to apply it to binary search trees. The student makes these observations:

- We can flatten a binary search tree of size n into a sorted array in O(n) time.
- We can turn a sorted array of length n into a well-balanced binary search tree in O(n) time by taking the middle element as the root and recursively "treeifying" the left and right sides.

The data structure the student proposes is this implementation of a set:

```
class BalancingTreeSet:
    bst = a regular (non-balanced) binary search tree
    capacity = 1

    add(key):
        bst.add(key)
        if bst.size() >= capacity:
            bst = fromSortedList(bst.toSortedList())
            capacity = capacity * 2

    contains(key):
        return bst.contains(key)
```

The student claims that the argument we apply to dynamic arrays applies here as well, and the operations `add` and `contains` will be amortised as efficient as AVL or other self-balancing trees.

Is the student correct (is `BalancingTreeSet` as efficient as an AVL Tree set)? (check a box)

Yes, it's as efficient ☐ [✓ No, it's less efficient]

If you answered yes: provide a more convincing argument.

If you answered no: disprove the claim and demonstrate a worst-case that exhibits suboptimal performance.

The worst case, after $n=2^k$ insertions the tree will have a balanced part of $2^{k-1}=n/2$ elements and the remaining n/2 elements unbalanced, in the worst case a so-called *pathological* tree which is essentially a linked list. So adding those last n/2 elements will have taken quadratic time. Furthermore, if we stop inserting at that point and run an arbitrarily large number of contains operations, each of those will be O(n) time worst case.

(In dynamic arrays, access time is not affected by how close you are to resize!)

Advanced question 11: Union of BST sets

Assume the following standard BST implementation of sets of integers:

```
class IntSet:
    left : IntSet
    mid : Integer
    right : IntSet
```

Here is an idea of a divide-and-conquer algorithm for calculating the union $A \cup B$ of two sets:

- Let x be the root value (mid) of A, and A_L and A_R the left and right children of A
- Split the other tree (B) into two new trees:
    - $B_<$ contains all values less than x
    - $B_>$ contains all values greater than x
- Calculate (recursively) the union of the subtrees:
    - $L = A_L \cup B_<$
    - $R = A_R \cup B_>$
- Now the union $A \cup B$ can be constructed by: `IntSet`(L, x, R)

First implement the helper functions for splitting B.

Writing one of these two (very similar) functions is enough:

```
def splitLesser(s : IntSet, x : Integer) -> IntSet
def splitGreater(s : IntSet, x : Integer) -> IntSet
```

Then implement union on IntSets as the following function:

```
def union(a : IntSet, b : IntSet) -> IntSet
```

The complexity of `splitLesser Lesser` and `splitGreater` must be O(h) where h is the height of the tree.

Note 1: Answer on a separate sheet of paper.

Note 2: Don't forget to handle empty sets (i.e., where the `IntSet` is null/None).

Note 3: You may treat `IntSet` as immutable (instances cannot be changed after creation). This allows you to safely reuse subtrees of the input in the output.

Note 4: If you want, you can implement this in Haskell too. Then you can assume that `IntSet` is the following type: `data IntSet = Empty | Node IntSet Integer IntSet`

Grading note: There was a typo in the exam where it said splitLeft/splitRight in the complexity requirement. See next page for solution.

```
def splitLesser(s : IntSet, x : Integer) -> IntSet
    if s==null:
        return null // If s is empty it has no elements less than x
    if s.mid >= x:
        return splitLesser(s.left, x) // all the values are in s.left
    // Now we know s.mid < x
    // The tree should include s.mid and everything from s.left,
    //   and the parts of s.right that are < x
    return IntSet(s.left, s.mid, splitLesser(s.right, x))

def union(a : IntSet, b : IntSet) -> IntSet
    if a==null:
        return b
    x = a.mid
    lft = union(a.left, splitLesser(b, x))
    rgt = union(a.right, splitGreater(b, x))
    Return IntSet(lft, x, rgt)
```

Advanced question 12: Graph algorithm

The class `Graph` implements directed graphs using an adjacency matrix (where the vertices are numbered 0, 1, 2, ...). For a graph `g`:

- `g.size()` gives the number of vertices.
- `g.connected(x,y)` is true if there is an edge from vertex number x to vertex number y in the graph (and false for all other inputs).

Both operations take O(1) time. Your task is to determine what `fun` does and how fast it is.

```
class Graph:
    size() -> int
    connected(from : int, to : int) -> bool

fun(g : Graph, s : int) -> Bool:
    visited = new Set
    queue = new Queue
    queue.enqueue(s)

    while not queue.isEmpty():
        here = queue.dequeue()
        if not visited.contains(here):
            visited.add(here)
            if g.connected(here, s):
                return true
            for i from 0 to g.size():
                if g.connected(here, i):
                    queue.enqueue(i)
    return false
```

Describe briefly what the function does:

Determines if s is part of a cycle (if there is a path from s back to s)

What is the worst-case asymptotic complexity of the function in the numbers *V* of vertices and *E* of edges of g? Make reasonable assumptions about the complexity of data structure operations. Justify your answer.

O(V^2) assuming enqueue is O(1) and dominates the complexity. not visited.contains is true at most V times, and the for-loop clearly has V iterations.

One could argue for O(E log V + V^2) if visited.contains is O(log n), but using an AVL tree set or such here does not make much sense since a simple array of booleans will give you O(1) contains.

Grading notes: A few other answers can probably be acceptable if they have sound arguments.