Question 1: Complexity [2 points]

The following program takes a list of integers `x` of length d and checks if there are four elements (duplicates allowed) that sum to zero.

```
result = false
sums = new set
for a in x:
    for b in x:
        sum = a + b
        sums.add(sum)
        if sums contains (-sum):
            result = true
```

What is the asymptotic complexity in d of this program if the set `sums` is implemented:

(A) using an (unsorted) dynamic array,

(B) using an AVL tree?

Answer using O-notation. In each case, your answer should be best-possible (sharp) and as simple as possible; justify that the complexity of the program is of the stated order of growth.

We have two nested for-loops, each with d iterations. So the complexity is on the order of d^2 times the O-bound for the inner loop body. That O-bound is determined by the complexity of `add` and `contains`.

(A) `add` is amortized O(1). Since we execute it many times, we can regard it as O(1). `contains` is O(d^2) because `sums` grows to at most d^2 elements. So the total complexity is O(d^2 d^2) = O(d^4).

(B) sums grows to at most d^2 elements. So `add` and `contains` are O(log(d^2)) = O(log(d)). So the total complexity is O(d^2 log(d)).

Question 2: Sorting [2 points]

Part A: quicksort

This question is about the *partitioning* algorithm in quicksort. Consider the following array:

- [10, 17, 2, 19, 8, 5, 4, 9, 11]

Partition this array using the first element as pivot. **You do not need to sort the array**, you only need to perform the partitioning step of quicksort.

Your answer must state or show:

- the sequence of comparisons and swaps performed (for example, by writing "X < Y?" when the elements X and Y are compared, and "X ↔ Y" when they are swapped)
- the resulting array, with the two partitions indicated (for example, underlined)

Note: If you use a different partitioning algorithm than the course, state the algorithm you used (either its name or a description).

We initialize lo = 1, hi = 8. The pivot is 10.

- 17 < 10? No.
- 11 > 10? Yes, we advance hi.
- 9 > 10? No.
- We swap 17 and 9 and advance lo and hi.
- 2 < 10? Yes, we advance lo.
- 19 < 10? No.
- 4 > 10? No.
- We swap 19 and 4 and advance lo and hi.
- 8 < 10? Yes, we advance lo.
- 5 < 10? Yes, we advance lo.
- Finally, we swap the pivot 10 with 5.
-
-
-

The final partitioning is [5, 9, 2, 4, 8, 10, 19, 17, 11].

Part B: merge sort

This question is about the *merge* algorithm in merge sort. Consider these two sorted lists:

- [2, 7, 9, 17]
- [1, 3, 4, 12, 15]

State the sequence of comparisons the merge algorithm performs when called on this input.

You do not need to sort the two lists, you only need to perform the final merge step of merge sort.

- Comparing 2 and 1. Writing 1.
- Comparing 2 and 3. Writing 2.
- Comparing 7 and 3. Writing 3.
- Comparing 7 and 4. Writing 4.
- Comparing 7 and 12. Writing 7.
- Comparing 9 and 12. Writing 9.
- Comparing 17 and 12. Writing 12.
- Comparing 17 and 15. Writing 15.
- Writing 17.

The output (not required to state) is [1, 2, 3, 4, 7, 9, 12, 15, 17].

Question 3: Binary search trees [2 points]

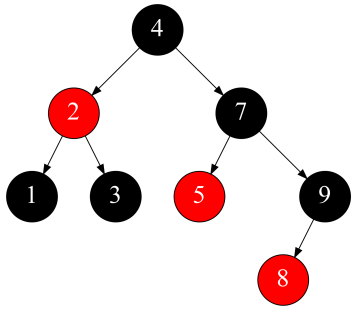
This question is about red-black trees. You may either use the version of red-black trees taught in the course, or the version taught in previous years.

If you use a different version of red-black trees than the one from the course in LP2, give a reference (e.g. a link) that explains the version you are using.

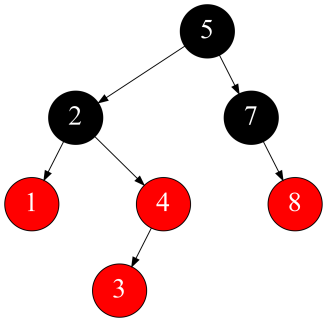
Part A

Exactly one of the following coloured trees is a red-black tree. Find it. For the other ones, say why they are not red-black trees by stating what is wrong.

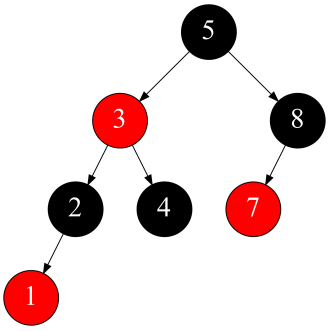
(1)



(2)



(3)



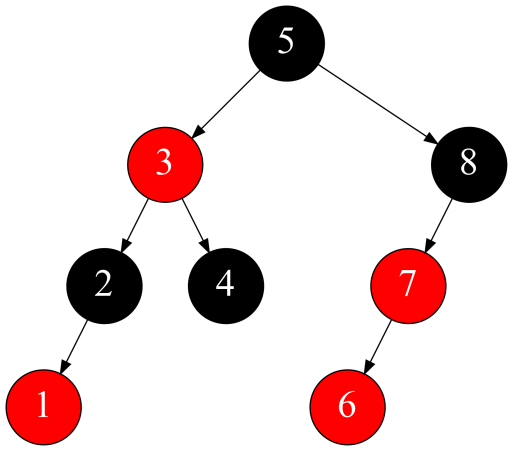
(1) The black balance invariant is violated: The path 4-7-9-8 has three black nodes, but the path 4-2-1 has only two.

(2) The invariant is violated: red node 3 is a child of the red node 4.

(3) This is a red-black tree.

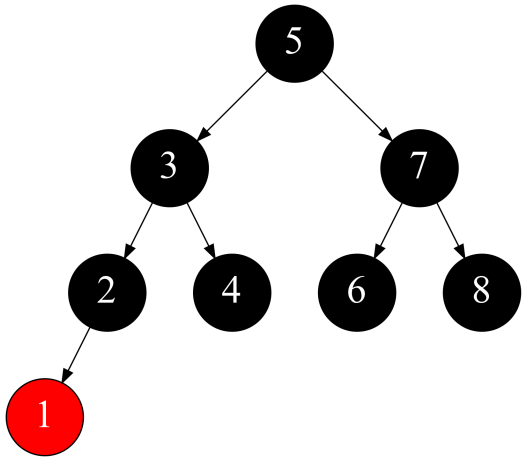
Part B

We now insert 6 into the red-black tree from Part A. We do it in two steps. First insert it using the standard BST insertion algorithm and color it red. Draw the resulting tree:



Now rebalance to obtain a valid red-black tree. State how you do it and draw the final red-black tree.

These are the rebalancing steps (terminology from the relevant lecture):
- apply “split” to 8→7→6,
- apply “skew” to 5→7,
- apply “split” to 8→5→3,
- color root 5 black.



Note: If you cannot colour the nodes, draw red nodes as squares and black nodes as circles.

Question 4: Priority queues [2 points]

This question is about **binary min-heaps** represented as arrays.
In the rest of the question, the word “heap” refers to a binary min-heap stored as an array.

Part A

Exactly one of the following arrays represents a heap:

a. [4, 6, 9, 10, 6, 18, 10, 16, 15, 12]
b. [2, 3, 7, 15, 11, 12, 13, 10, 16, 19]
c. [19, 18, 17, 15, 12, 11, 10, 7, 3, 0]

Say which one is a heap. For the other ones, explain why they are not heaps. Your explanation should point out a specific problem with the array (e.g. an element which is in the wrong place and why).

a. This is a heap.
b. This is not a heap: the element 10 is less than its parent 15.
c. This is not a heap: the root 19 is not the minimum. (Note that it is a max-heap instead.)

Part B

Remove the minimum from the heap you have identified in Part A, using the standard “remove minimum” algorithm for heaps. Your answer must state:

- the sequence of swaps performed,
- how the heap looks as an array afterwards.

We swap the root 4 with the last element 12, delete the new last element 4, and then sink down the root 12:

- We swap 12 with its left child 6.
- We swap 12 with its right child 6.
-

Final state: [6, 6, 9, 10, 12, 18, 10, 16, 15]

Question 5: Hash tables [2 points]

This question is about open addressing hash tables with **modular hashing** and **linear probing**. Every integer is its own hash code.

Part A

Consider the following hash table of size 9:

0	1	2	3	4	5	6	7	8
0	26		48	12		42		17

- Identify the clusters.
- For each cluster, state all possible orders its elements could have been inserted in. Assume no deletions happened.

The clusters are [48, 12], [42], and [17, 0, 26]. The possible insertion orders in each cluster are:

- for [48, 12]: only possibility "48 then 12",
- for [42]: only possibility "42",
- for [17, 0, 26]: "17 then 0 then 26" or "0 then 17 then 26".

Part B

Insert 19 and 53 into the hash table, in this order.

- In each case, state the sequence of cells that are tried for insertion.
- Show the final state of the hash table.

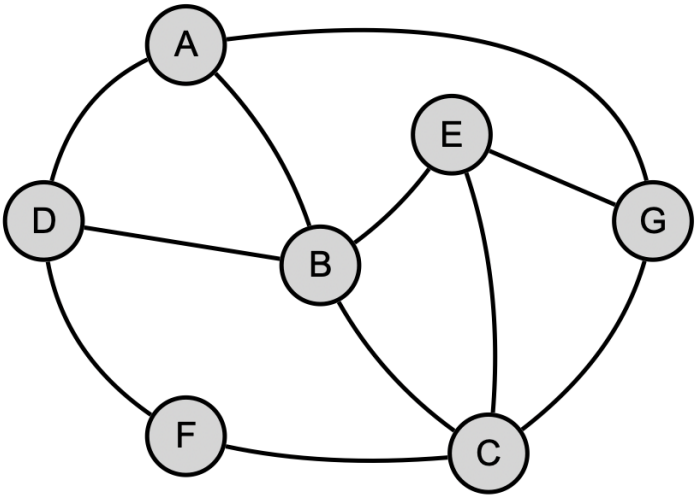
- The hash value of 19 is 1. Trying cells: 1 occupied, 2 free (inserting).
- The hash value of 53 is 8. Trying cells: 8 occupied, 0 occupied, 1 occupied, 2 occupied, 3 occupied, 4 occupied, 5 free (inserting).

0	1	2	3	4	5	6	7	8
0	26	19	48	12	53	42		17

Question 6: Graphs [2 points]

Part A

Consider the following graph:



Answer the following questions:

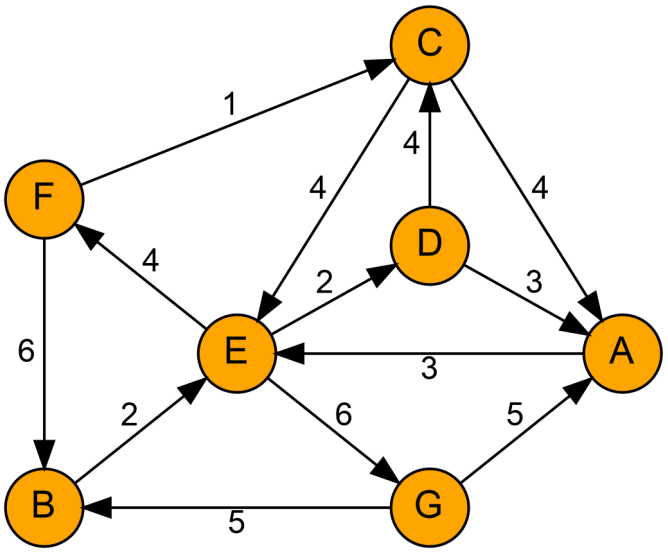
- What is the sum of the degrees?
- Draw (or list the edges of) a spanning tree.

The sum of degrees is 22 (shortcut: there are 11 edges).

Any collection of 6 edges that doesn't form a cycle will form a spanning tree. For example, we can take the path graph A-B-D-F-C-E-G.

Part B

Consider the following weighted directed graph:



Run UCS (Dijkstra's algorithm) with starting node A. Fill in the below table with:

- the order in which the nodes are visited,
- the cost to reach them,
- the content of the priority queue (nodes with cost) after the node has been processed, omitting all entries that lead back to already visited nodes.

Note: If you use a different version of UCS than the course, give a reference.

Node	Cost	Priority queue
A	0	E:3
E	3	D:5, F:7, G:9
D	5	F:7, G:9, C:9
F	7	C:8, G:9, C:9, B:13
C	8	G:9, B:13
G	9	B:13, B:14
B	13	

Question 7 (advanced): Weighted voting algorithm [3 advanced points]

In this question, you will design an efficient algorithm for ranking candidates in an election where each voter can split their vote over multiple candidates.

- A *candidate* is represented as a string. All string operations take constant time.
- A *vote* is a weighted selection of at most 5 candidates. It is represented as a map from the selected candidates to *weights*, numbers between 0 and 1. Because every vote should carry equal weight, the weights in each vote must sum to 1.
- Given a list of votes, the *score* of a candidate is the sum over all votes that select this candidate of the weight assigned to it.

Your task is to design a function

```
List<String> rankCandidates(List<Map<String,Number>> votes)
```

that takes a list of votes and returns the list of (unique) candidates appearing in the votes, sorted in descending order by their score.

Example: Given the following list of `votes`:

```
[ {"Mark": 0.4, "Dora": 0.6},
  {"Alex": 1},
  {"Dora": 0.5, "Mark": 0.5} ]
```

Your algorithm should return the list `["Dora","Alex","Mark"]`, because Dora has score 0.6 + 0.5 = 1.1, Alex has score 1, and Mark has score 0.4 + 0.5 = 0.9.

Your algorithm must have complexity O(n log(n)) where n is the size of `votes`. You should justify this complexity.

Notes:

- You don't have to check that the input is valid (i.e., you don't have to check that the weights in a vote sum to 1).
- You can specify your algorithm in pseudocode.
- You can use any data structure or algorithm treated in the course as a building block, and you can use any reasonable names for operations on abstract data types (for example, the course API).

There are many ways. Here is an example solution:

```
scores = new red-black map from candidates to numbers
for each vote in votes:
    for each entry (candidate, weight) in vote:
        if candidate not in scores:
```

```
            scores[candidate] = 0
        scores[candidate] += weight

x = array of entries (candidate, score) of scores
merge sort x according to reverse order of second component
return array of first components of x
```

In the nested for-loops, the first one has O(n) and the second one has O(1) (at most 5) iterations. So `scores` will have O(n) entries. The body uses red-black tree operations O(log(n)). So this part of the program is O(n log(n)). In the second part, merge sort on an array of size O(n) is O(n log(n)). This justifies the complexity O(n log(n)).

Question 8 (advanced): Denial-of-service attack [3 advanced points]

This question is about an online stock exchange, where customers can send requests to buy or sell shares in companies. When a customer sends a request to buy a stock at a certain price, the request is kept on hold until another customer sends a request to sell at that price. Then the buy order is matched with the sell order and the share is transferred.

The stock exchange's job is to match buyers with sellers, and to remember the unmatched buy and sell orders. To maintain good service, the stock exchange needs to handle the requests efficiently.

In this question, you will break the online stock exchange. To do so, you will send a series of seemingly harmless requests that take longer and longer to process. Eventually, there will be no resources left to serve the other customers' requests.

Part A

You have bribed an insider to supply you with the source code to the stock exchange.

The stock exchange remembers the unmatched buy and sell orders for each stock (or company). To keep these unmatched orders sorted by price, the exchange stores them in an ordinary binary search tree. There is one BST per company for buy orders and one for sell orders.

For buy orders, the binary search tree stores objects of class `BuyOrder`, defined as follows:

```
class BuyOrder:
    int id        // order ID (random)
    double price  // buy price in kr (e.g. 12.3456789)
    int number    // number of stocks to buy (at least one)

    int compareTo(BuyOrder other):
        if price != other.price:
            return Double.compare(price, other.price)
        return Integer.compare(id, other.id)
```

You can request to place a buy order via this API:

```
void placeBuyOrder(String stock, double price, int number)
```

If the price is less than the lowest sell order, then the buy order is not matched and is stored in the BST instead. Otherwise, the buy order is matched to the sell order and the stock is transferred. The price can have any number of decimal places, e.g. you can place an order to buy a stock at 12.3456789 kr.

In your attack, you will buy shares in "Pyramid AB". The lowest sell order is currently 14.83 kr, so any buy order less than 14.83 kr will not match a sell order, and will instead be stored in the BST. The lowest buy order is currently 10.01 kr.

The stock exchange allows you to make some large number N (≈ 1 000 000) of requests.

Answer the following:

- Describe a sequence of N requests that would take an extremely long time to process.
- Explain in terms of asymptotic complexity the time the exchange takes to process all the requests.

The basic idea is to cause the BST that stores buy orders to degenerate (become highly unbalanced) and behave like a linked list, realizing the worst-case complexity.

We execute `placeBuyOrder("Pyramid AB", i / N * 10, 1)` for i from 1 to N. This places orders with increasing prices 1/N 10 kr, 2/N 10 kr, …, N/N 10 kr (we could also place orders with decreasing prices). When the exchange stores them in the BST, this will cause the same branch to continue growing to the right. So the BST behaves like a linked list, with insertion complexity Θ(n) where n is its current size. The total complexity of processing these N orders is Θ(N^2), which for large N would mean days of processing time.

Part B

You have convinced the exchange to hire you to prevent future attacks of this kind. Describe a change you could make to their codebase that fixes the above problem.

Note: You cannot restrict the number of orders per user, or limit the number of decimal places in prices. The high-frequency traders will get angry!

Change the BST to a self-balancing BST (e.g., red-black tree or AVL tree).