Basic question 1: Mergesort

Consider the following 8-element array:

0	1	2	3	4	5	6	7
12	7	14	9	10	11	6	2

We use merge sort to sort the elements in increasing order.

Show every merge step in the order that they are performed (the two input arrays and the merged output) and the final sorted array. Use the following table and use as many rows as you need for all the merge steps.

Array 1	**Array 2**	**Merged output**
[12]	[7]	[7,12]
[14]	[9]	[9,14]
[7,12]	[9,14]	[7,9,12,14]
[10]	[11]	[10,11]
[6]	[2]	[2,6]
[10,11]	[2,6]	[2,6,10,11]
[7,9,12,14]	[2,6,10,11]	[2,6,7,9,10,11,12,14]

Note that other orders are also possible: e.g., you can perform all the small merges first, followed by the two mid-size merges – this will then be bottom-up merge sort.

Basic question 2: Hash tables

The hash table to the right uses linear probing with lazy deletion. The hash table uses modular compression (hash(x) % 10), but the hash function is unknown.

0	<deleted>
1	A
2	
3	
4	C
5	
6	E
7	B
8	F
9	D

a) Which one of these is a possible hash function used to build the table?
Note that this is **not** the order values were added, just the hash function used.
(select only one option)

- ☐ hash(A)=29, hash(C)=4, hash(E)=16, hash(B)=7, hash(D)=21
- ☐ hash(A)=39, hash(C)=4, hash(E)=27, hash(B)=6, hash(D)=11
- ☑ hash(A)=49, hash(C)=4, hash(E)=36, hash(B)=7, hash(D)=19
- ☐ hash(A)=89, hash(C)=4, hash(E)=47, hash(B)=6, hash(D)=39
- ☐ hash(A)=71, hash(C)=4, hash(E)=56, hash(B)=7, hash(D)=11
- ☐ hash(A)=21, hash(C)=4, hash(E)=66, hash(B)=6, hash(D)=81
- ☑ hash(A)=11, hash(C)=4, hash(E)=76, hash(B)=7, hash(D)=99
- ☑ hash(A)=11, hash(C)=4, hash(E)=86, hash(B)=6, hash(D)=9

Note: All three answers above are correct, and it is fine if you answer any of them.

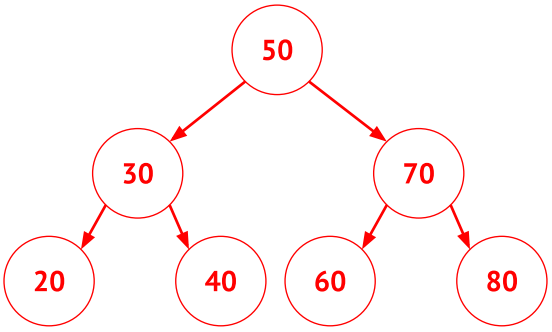
b) Add F with hash(F)=46 to the table (without resizing). You can add the value directly to the table above.

It gets added at index 8 (because both 6 and 7 are occupied), see the table.

Basic question 3: BSTs

We start from an empty binary search tree (BST). We add values to it using the standard insertion procedure, in the following order: 50, 30, 70, 20, 40, 60, 80.

a) Draw the resulting **binary search tree**.



b) List the **nodes** in the order they are processed in a (recursive) pre-order and post-order traversal.

Hint: Pre-order traversal processes a node before making the recursive calls on its left and right children, while post-order traversal processes it after the recursive calls.

Pre-order: _____

Post-order: _____

Pre-order: 50, 30, 20, 40, 70, 60, 80

Post-order: 20, 40, 30, 60, 80, 70, 50

Basic question 4: Data structure complexity

Below is a table (partially filled in) of the asymptotic complexity of some operations in efficient implementations of some data structures. The asymptotic complexity is expressed in terms of the number n of elements of the data structure.

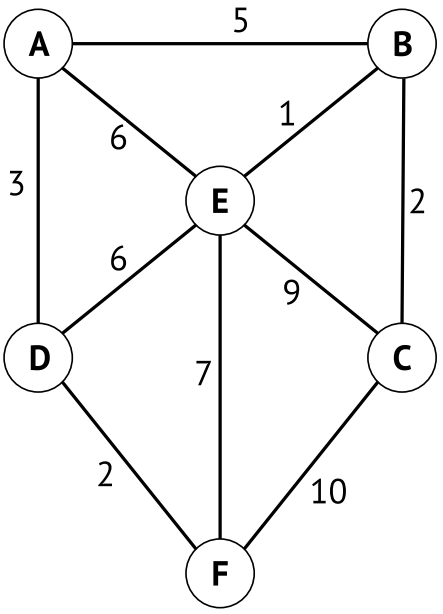
Fill in the rest of the table.

Data structure	**getMin() (Finding smallest element, without modifications)**	**contains(x) (Searching for a specified element)**	**add(x) (Adding a new element)**
Sorted linked list	$O(1)$	$O(n)$	$O(n)$
Binary min-heap	$O(1)$	$O(n)$	amortized $O(\log(n))$
Dynamic array (not sorted)	$O(n)$	$O(n)$	amortized $O(1)$ (non-amortized $O(n)$)
AVL-tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Basic question 5: Graphs

We run this familiar piece of code on the graph to the right:

```
visited = new set of nodes
result = new list of node pairs
agenda = new min-priority queue of edges ordered by their weights
agenda.add( (null,0,E) )    // add a dummy edge for start node E
while agenda is not empty:
        (v, weight, w) = agenda.removeMin()
        if not visited.contains(w):
                visited.add(w)
                for (from, weight, to) in outgoingEdges(w):
                        if not visited.contains(to)
                                agenda.add( (from, weight, to) )
                result.append( (v, w) )
return result
```



a. What is this algorithm computing?

This is Prim's algorithm that computes a minimum spanning tree of the graph.

b. What are the contents of the set ***visited*** and the list ***result***, initially and after each of the first four iterations of the while loop? (Some values are pre-filled for explanatory purposes.)

Note: you must report all of the contents of ***visited*** and ***result*** *in each iteration*.

Iteration	***visited***	***result***
before the loop	—	—
after iteration 1	E	(null, E)
after iteration 2	E, B	(null, E), (E, B)
after iteration 3	E, B, C	(null, E), (E, B), (B,C)
after iteration 4	E, B, C, A	(null, E), (E, B), (B,C), (B,A)

Basic question 6: Complexity

Below are two (mostly nonsense) functions, each taking an integer array as a parameter.

State the asymptotic time complexity of:

- Algorithm 1 in terms of the **parameter k**.
- Algorithm 2 in terms of the **length *N* of the array**.

In each case, **briefly state** how you concluded this (you may do this by annotating the programs).

***Notes*:**

- Assume that all arithmetic operations and array indexing take O(1) time.
- Answer in O-notation, and as tight and simple as possible.

Algorithm 1

```
function f(x, arr, offs, k):
    if k ⩽ 0:
        return offs
    half = k // 2      (note: // is integer division)
    m = offs + half
    if arr[m] < x:
        offs = offs + half
    return f(x, arr, offs, half)
```

Complexity: $O(\log k)$

Justification:

The argument k is halved in the recursive call (rounding down), so there will be $O(\log(k))$ function calls in total. The remaining statements in each call take O(1) time.

So the complexity is $O\log(k) \cdot 1) = O(\log(k))$.

Note: The question originally said “n” instead of k in the if-statement

Algorithm 2

```
function rolling(arr):
    res = new dynamic array
    for i from 0 to arr.length−7:
        sum = 0
        for offset from 0 to 6:
            sum = sum + arr[i+offset]
        res.add(sum/7)
    return res
```

Complexity: $O(N)$

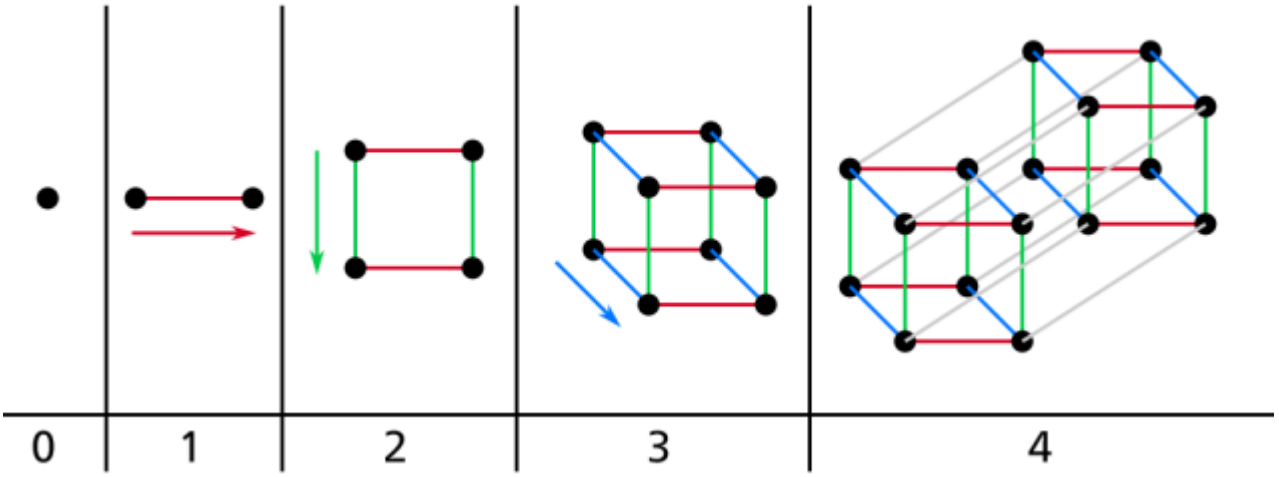
Justification:

The for-loop does $O(N)$ iterations. Amortized over all loop iterations, adding an element to the dynamic array res takes O(1) time. The rest of the body also takes O(1) time.

So the complexity is $O(N \cdot 1) = O(N)$.

Advanced question 7: Hypercubes

An N-dimensional hypercube can be considered as a graph. Here is an illustration for $0 \leqslant N \leqslant 4$:



Suppose you have an N-dimensional hypercube with undirected weighted edges, and you want to run an $O(E \log(E))$ implementation of Kruskal's algorithm on it (where E is the number of edges). What is the asymptotic complexity of doing so in terms of N, the number of dimensions of the hypercube? You should answer in a simplified O-notation, and explain your reasoning.

Hint: Notice how an N-dimensional cube is two copies of an $(N-1)$-dimensional cube, with exactly one additional edge to/from each node. This means all nodes in an N-dimensional cube will have a certain property in common that will help you calculate the number of edges.

Hint: This is going to be a slow complexity class.

a) Answer these questions about the graph for the N-dimensional hypercube:

Number V of nodes (in terms of N): _____

Number E of edges (in terms of N): _____

b) What is the asymptotic complexity of running Kruskal's algorithm on an N-dimensional hypercube? (answer in O-notation, and remember to simplify as much as possible)

_____

Explanation:

The number of nodes for dimension N is 2^N, and the degree of each node is N. Since the graph is undirected this gives us $E = 2^N \times N/2 = 2^{N-1} \times N$, so $O(E)=O(2^N \times N)$. That makes Kruskal's: $O(2^N \times N \times \log(2^N \times N)) = O(2^N \times N \times (N+\log(N))) = O(2^N \times N^2)$.

Advanced question 8: LinkedList Splitting

You are given a singly linked list that contains *n* elements. The class for LinkedList is as follows:

```
class LinkedList<Item>:
    Node head          // points to the head of the list
    class Node:
        Node next      // points to the next list node
        Item value     // the value of this node
```

Design an algorithm to split the list into two halves so that the following requirements are met:

- If n is even, both halves have *n*/2 nodes.
- If n is odd, the first half has one more node than the second.
- You may only use a single loop with at most *n* loop iterations (and no recursion).
- Your loop can use multiple pointers to traverse the list, but you may only use O(1) extra space (so no arrays or secondary lists are allowed).

Note: the number of nodes in the list is not known in advance. The class definition above is complete, there are no extra methods or variables you can use.

a) One attempt at a solution is to first count the number of nodes, and then split at the middle node. Why does this not meet the requirements?

b) Propose an algorithm that satisfies all conditions above. Use pseudocode or a detailed text description. Your algorithm should take a Node object as input and return two Nodes (the heads of the two halves).

c) Show that your algorithm runs in O(n) time and O(1) extra space.

d) Demonstrate your algorithm using the following example list:

   1 → 4 → 7 → 9 → 12 → 15 → null

   Show the steps your algorithm takes to split the above list in halves.

Write your solution on a separate sheet of paper.

Solution suggestion:

a) To count first and then split, you need to traverse the list twice, which violates the 3rd condition.

b) Here is one possible solution:

```
splitLeft(head : Node) –> (leftHead : Node, rightHead : Node):
    if head == null or head.next == null:
        return head, null   // leftHead = all, rightHead = empty
    slow = head
    fast = head.next          // key choice for equal halves on even n
    while fast != null and fast.next != null:
        slow = slow.next
        fast = fast.next.next
    rightHead = slow.next   // capture start of right half
    slow.next = null          // cut after slow
    return head, rightHead
```

c) **Time:** The loop advances based on fast. Each iteration fast advances by two nodes, so the loop body executes O(n/2) times. All other operations take constant time, so we have O(n) total time.

Space: we only need four extra pointers to realize this algorithm: slow, fast, leftHead, rightHead. That is O(1) extra space.

d) Here's one way to show the steps of the algorithm:

Step	slow	fast
0	1	4
1	4	9
2	7	15 (stop because fast.next=null)

Advanced question 9: Updateable priority queue

Here is a draft implementation of a priority queue of unique keys with updatable priorities. We use a heap together with an AVL-map (positions) to quickly find where keys are stored in the heap. In addition to the heap invariant, that means we have the following invariant:

If (and only if) *heap[i]=(prio,key)* then *positions.get(key) = i*

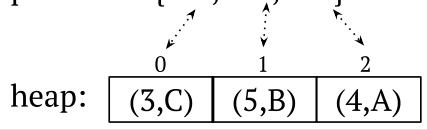
```
class MinPrioMap<Key, Priority>:
    positions: AVL-map from Key to integers       // maps keys to positions in the heap
    heap: Dynamic Array of (Priority, Key)-pairs  // binary heap ordered by priorities

    function setPriority(Key key, Priority prio):
        if positions.containsKey(key):          // key is already present, update priority
            i = positions.get(key)
            (oldPrio, key) = heap[i]
            if prio < oldPrio:                  // we only update priority if the new priority is lower
                heap[i] = (prio, key)
                swim(i)
        else:       // this is a new key, extend the heap
            heap.add( (prio, key) )
            swim(heap.length-1)
```

The illustration to the right shows the connection between the positions-map and the heap when the invariant holds.

Example after running:

```
setPriority(A,4)
setPriority(B,5)
setPriority(C,3)
```

positions: {C:0, B:1, A:2}



a) Implement the method **swim**, such that **setPriority** preserves the invariants.

b) What is the (amortized) complexity of **setPriority** using your **swim** function, in terms of the number of values in the queue n?

Here is a recursive solution that doesn't do any needless updates of the map:

```
swim(index):
    parent = (index-1)/2
    if index > 0 and heap[parent] > heap[index]:
        swap heap[parent] and heap[index]
        swim(parent)
    (prio, key) = heap[index]
    positions.put(key, index)
```

And an iterative solution:

```
swim(index):
    while (index >= 0
            and heap[parent] > heap[index]):
        parent = (index-1)/2
        swap heap[parent] and heap[index]
        (prio, key) = heap[index]
        positions.put(key, index)
        index=parent
    (prio, key) = heap[index]
    positions.put(key, index)
```

The (amortized) complexity is $O(\log(n)^2)$ as it calls put on the AVL map $O(\log(n))$ times.
Note: the call heap.add() is amortized $O(\log(n)$, but would be $O(n)$ without amortization.

Note (unrelated to grading): This is not a good enough complexity for something like Dijkstra's. Using a hash map would be better, especially if keys have a perfect hash function.