Anonymous code: ______________________________

	1	2	3	4	5	6	7	8	Grade
Results:									

Basic question 1: Complexity

What is the worst-case asymptotic time complexity (in the size *n* of `arr`) of the following function that searches for an element `x` in a sorted array `arr`?

```
// arr: sorted array of integers of length n
// x: an integer
bool contains(int[] arr, int x):
    int i = 1
    while i*2 < n and arr[i*2] < x:  // O(log n) because i is doubled in each step
        i = i*2
    while i < n and arr[i] < x:      // O(n) because in the worst case i loops
        i = i+1                      // from n/2 to n, which is O(n) steps
    return arr[i] == x
```

Write your answer in O-notation, and be as exact and simple as possible.

Answer: O(n)

Brief explanation:

(Alternatively, you can add comments directly to the code above if you want.)

There are two loops:

- the first iterates from 1 to something between n/2 and n, but since it doubles i in each step it will only use O(log n) steps
- the second iterates through every remaining value until n, and in the worst case this means all values from n/2 to n, which is O(n)

The total complexity is the sum of the two loops, O(log n) + O(n) = O(n).

Basic question 2: Sorting

As you know, the worst-case time complexity of *quicksort* is quadratic. Assuming a strategy of selecting the middle element (index rounded down), how can the elements 1,2,3,4,5 be arranged in an array to give the worst-case performance? To be more precise: Worst case means that one of the two recursive calls should always get an empty range of values.

Recall: The partitioning algorithm starts by swapping the pivot with the first element in the current sorting range.

Check the boxes ☑ for alternatives that are correct (may be more than one):

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

Brief explanation:

For each of the correct answers above, state the order in which the elements are used as pivots (one line per box you checked, e.g. [5,3,4,2,1] would mean first 5 is used, then 3, then 4...).

B) The pivots are selected in this order: 1, 2, 3, 4, 5

D) The pivots are selected in this order: 1, 2, 3, 5, 4

Basic question 3: Lists, stacks, queues

Assume you have the following circular-array-based implementation of a queue:

0	1	2	3	4	5	6	7
A	X	C			S	Q	M

Dequeue one element, and then enqueue the same element. How does the array look like now?

0	1	2	3	4	5	6	7
A	**X**	**C**	**S**			**Q**	**M**

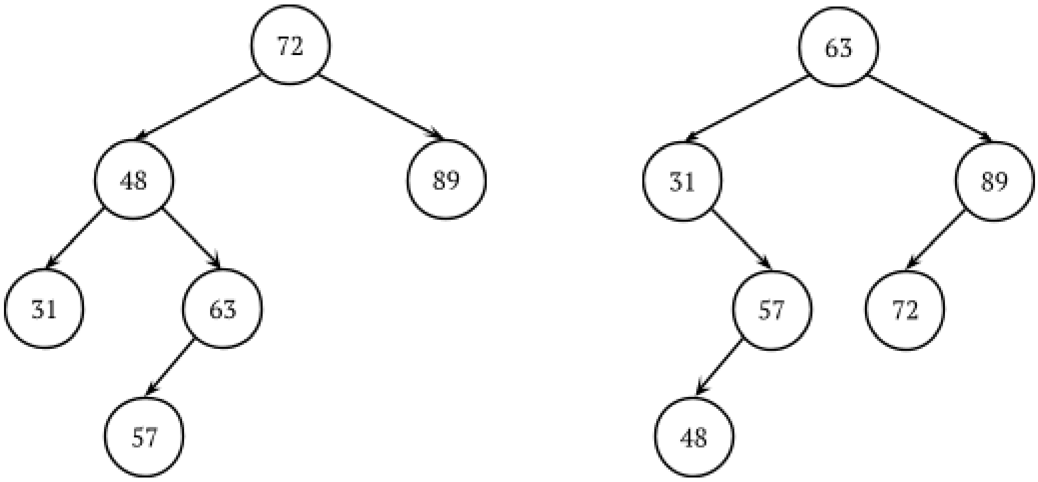
Now enlarge the array by 50% (after the operation above), preserving the order of values in the queue. What does the queue look like after this?

0	1	2	3	4	5	6	7	8	9	10	11
Q	**M**	**A**	**X**	**C**	**S**						

Note: there are some alternative solutions, but this is what the algorithm form the course book and the lectures.

Basic question 4: Red-black search trees

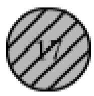
Exactly one of the following two trees can be painted as a valid red-black search tree:



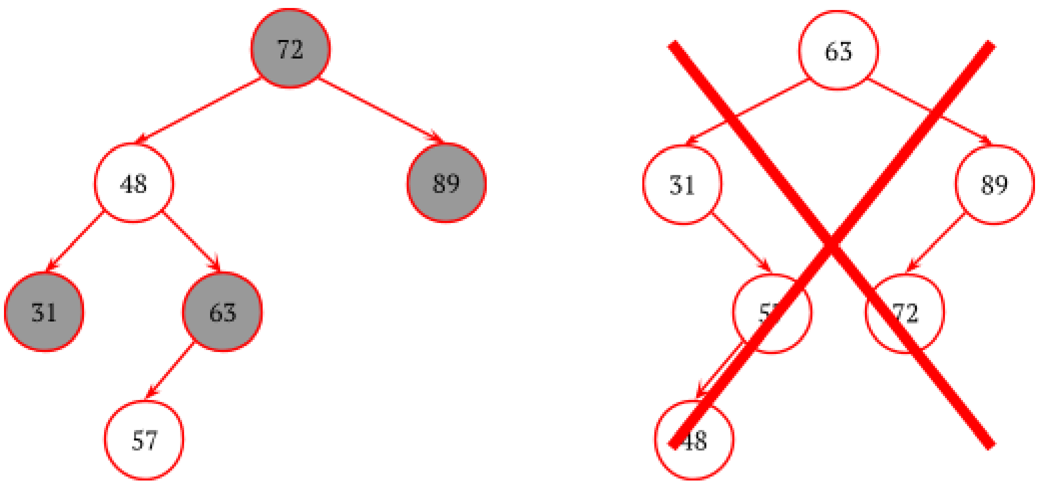
- Cross out the tree that is not valid.
- Paint the black nodes in the valid tree.

Note: You are ***not*** allowed to use a red pen, so instead you should paint the black nodes.
Leave the red nodes as they are, like this:

Red node: 

Black node: 

Alternatively, draw the valid tree below, with the black nodes painted and the red nodes hollow.



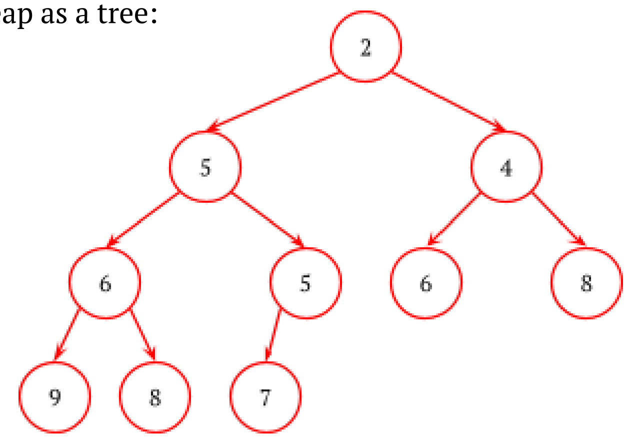
Basic question 5: Priority queues

Exactly one of the following three integer arrays is a valid representation of a binary min-heap:

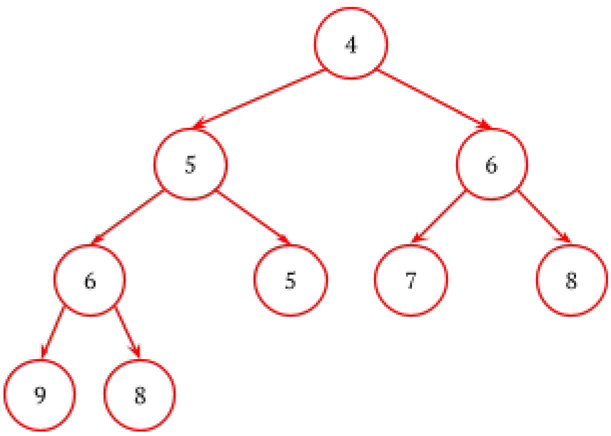
	0	1	2	3	4	5	6	7	8	9	10	11
A:	2	5	4	6	5	6	8	9	8	7		
B:	3	6	4	7	5	5	8	8	6	9		
C:	1	2	3	4	7	7	3	4	5	6		

Which one is a binary min-heap? **A**

Draw the heap as a tree:



Remove the minimal value from the heap and show the resulting tree:

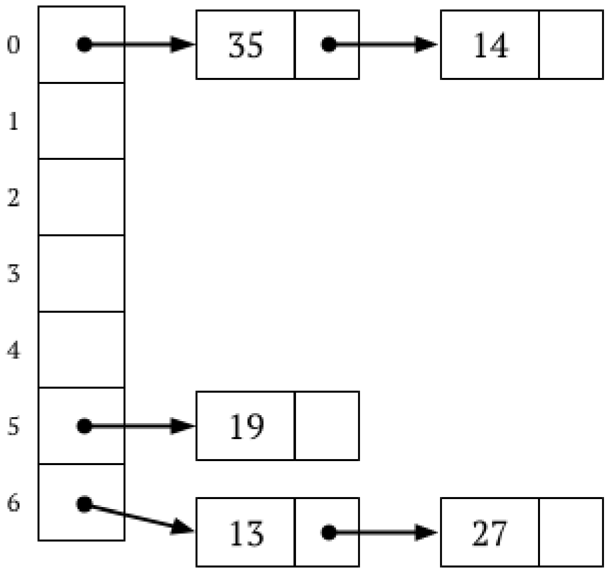


And finally, show how the final heap is represented as an array:

0	1	2	3	4	5	6	7	8	9	10	11
4	5	6	6	5	7	8	9	8			

Basic question 6: Hash tables

Here is a separate-chaining hash table of integers, where the containers are simple linked lists:



In which order can the elements have been inserted into the hash table?

Check the box ☑ with the correct alternative (there is only one):

- ☐ 35, 27, 19, 14, 13
- ☐ 19, 27, 13, 35, 14
- ☐ 13, 14, 19, 27, 35
- ☑ 27, 19, 14, 13, 35

Note: assume that elements are always added at the front of the linked list.

Now, insert the elements in the same order into the following linear probing hash table:

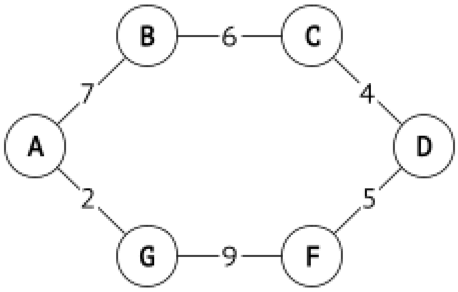
0	1	2	3	4	5	6
14	13	35			19	27

Note: this hash table has the same internal array size as the initial table.

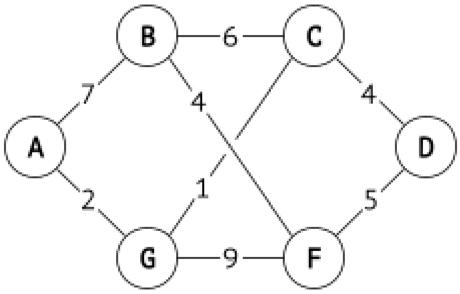
Basic question 7: Graphs

Here are two different weighted undirected graphs.

Graph 1:

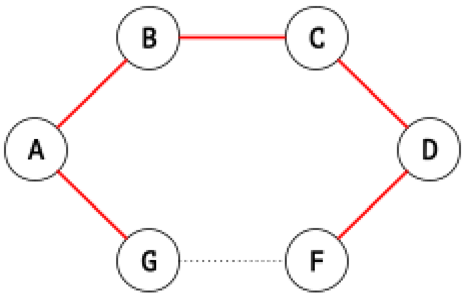


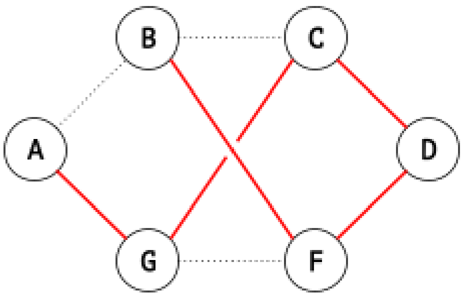
Graph 2:



Perform **Kruskal's algorithm** to construct a minimum spanning tree (MST) for each of the two graphs. Recall that Kruskal is the algorithm that doesn't have a starting vertex.

Draw the MSTs of both graphs by filling the dotted edges:





List the edges of the MSTs *in the order they are produced* by Kruskal's algorithm.

- Write the edges in the form AC, DF, ...
- Note that not all rows have to be used

Graph 1

1	**AG**
2	**CD**
3	**DF**
4	**BC**
5	**AB**
6	

Graph 2

1	**GC**
2	**AG**
3	**BF or CD**
4	**CD or BF**
5	**DF**
6	

Basic question 8: Mystery data structure

The following code implements a common data structure.

```
class Mysterious:
    a : Mysterious
    b : Mysterious
    c : String

def f(x, y):
    if y == null: return new Mysterious(null, null, x)
    else if x < y.c: y.b = f(x, y.b)
    else if x > y.c: y.a = f(x, y.a)
    return y

def g(x, y):
    if y == null: return false
    else if x < y.c: return g(x, y.b)
    else if x > y.c: return g(x, y.a)
    else: return true
```

Which data structure is implemented?

BST = binary search tree

(a set implemented as a BST, to be more exact)

(a node in a BST set, to be even more more exact)

Give the methods `f` and `g` names that are more descriptive (with respect to the data structure):

`f` = **`add`** (or **`put`** or **`set`** or **`add_this_string_to_the_set`**)

`g` = **`contains`** (or **`exists`** or **`has`** or **`is_this_string_in_the_set`**)