Assignment Lecture

21. September 2022

Assignment 2 Solution

Assignment 3 Walkthrough

Assignment 2 Solution

Task 1 a)

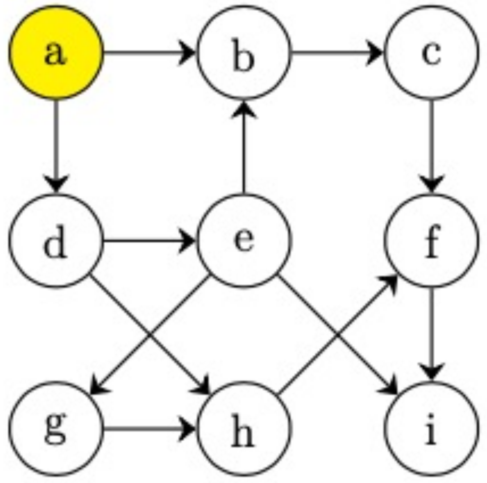


Figure 1: Graph Romeo

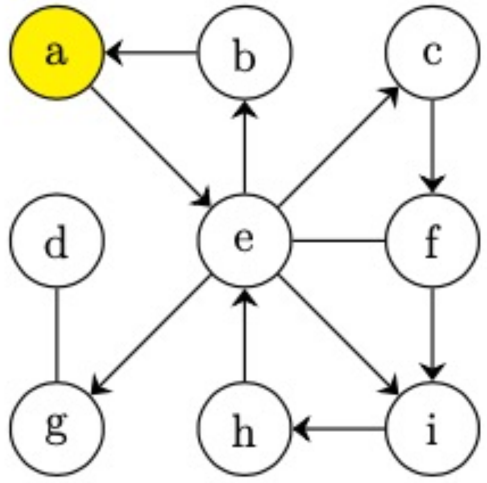


Figure 2: Graph Juliet

Romeo:
BFS:
discovered=abdcehfgi, visited=abdcehfgi

DFS (stack):
discovered=abdehfigc, visited=adhfiegbc

DFS (recursive):
discovered=abcfidegh, visited=abcfidegh

Juliet:
BFS:
discovered=aebcfgidh, visited=aebcfgidh

DFS (stack):
discovered=aebcfgihd, visited=aeihgdfcb

DFS (recursive):
discovered=aebcfgdih, visited=aebcfgdih

Solution:
Note that there are two possible solutions for DFS, one where a stack is used, and one where we imagine a recursive method (which essentially will instantly expand the node as it is discovered).

Note also that since DFS discovers the nodes in alphabetical order, they will be added to the stack in alphabetical order (and subsequently removed in the reverse order). This means that when using the stack from a, b is discovered first, then d, but then d is the first to be removed from the stack (and expanded/visited).

Task 1 b)

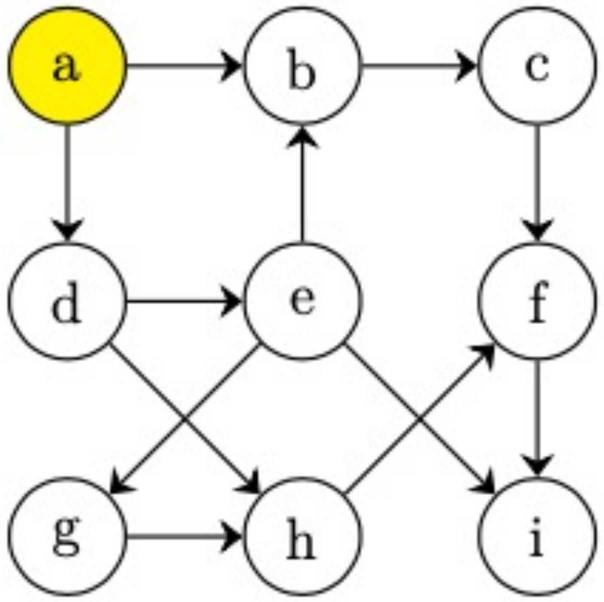


Figure 1: Graph Romeo

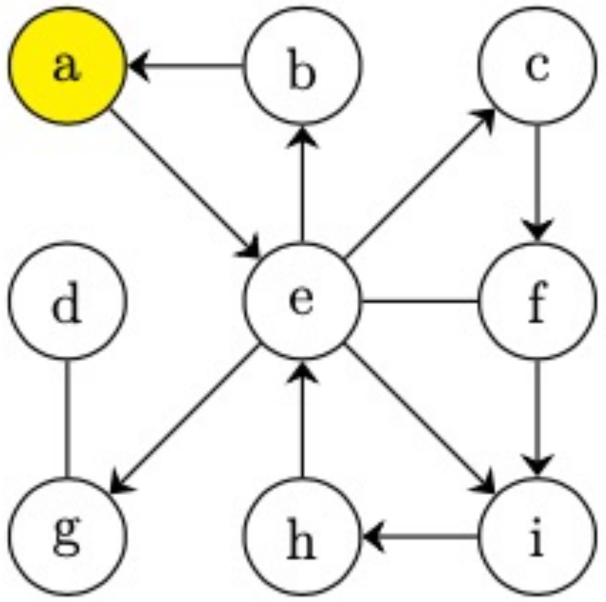


Figure 2: Graph Juliet

Give a (single) topological ordering of the two graphs, or explain why that isn't possible.

Solution:
Romeo: (6 valid orderings)
`adebcghfi`
`adebgchfi`
`adebghcfi`
`adegbchfi`
`adegbhcfi`
`adeghbcfi`

Juliet:
Cannot create a topological ordering because the graph contains cycles. For instance: `efe` or `aeba`

Task 1 c)

Will BFS or DFS be more space efficient? Justify your answer and any assumptions you make.

Usually

Solution:
DFS will be more space efficient.

In BFS nodes are added to a queue layer by layer, this means for instance in a binary tree (as an example, same principle works in all graphs) with a height of 10, once the BFS is going to start expanding nodes on the 10th layer, it will have 1024 nodes in its queue (all nodes from layer 10).

In DFS nodes are added to a stack, in a binary tree of height 10, the stack will at most contain 10 nodes. (as the search passes down the tree it adds a maximum of 2 nodes to its stack for each level, then removes one again to go the next layer down).

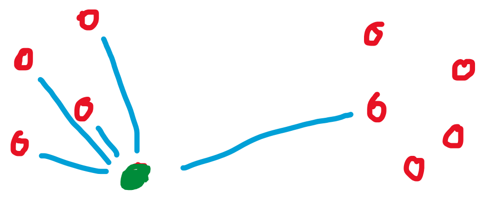
Task 2 a)

- At all times, each device i is within 200 meters of at least $n/2$ of the other devices. (You can assume n is an even number.)

Does this property guarantee that the network will remain connected? Explain your answer.

Solution:
Yes,

Lets say there are 10 phones, and my phone is one of them, the rule states that my phone must be connected to $10/2 = 5$ other devices at all times. This means that in my network there is at least 6 devices (5+my phone). That leaves only 4 devices "outside" of this network. In other words, for the rule to apply to these phones as well, they all have to be connected to at least one phone from "my" network, establishing a network between all the phones.

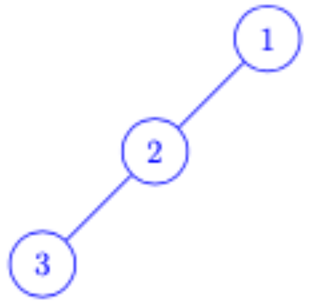


Task 3 a)

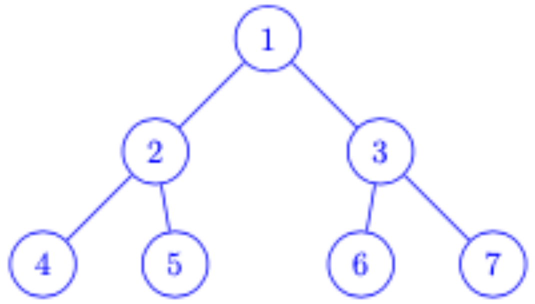
In any binary tree with a number of leaves (l), what can you say about the number of nodes (N) that has two children? Explain your answer.

Solution:

The number of nodes with two children is exactly one less than the number of leaves. ($N = l - 1$)



1 leaf = 0 nodes with 2 children



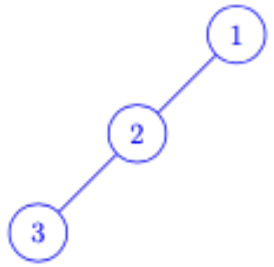
4 leaves = 3 nodes with 2 children

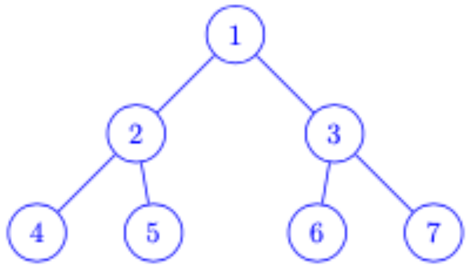
Task 3 b) i.

The height of a given binary tree is h.

What can you say about the total number of nodes n in the tree (min/max)?

Solution:
below are two examples of binary trees, both with a height of 2. The left tree has minimum number of nodes, and right has maximum number of nodes. Note that the value in the nodes only show how many nodes are in total after they are added level by level.





The simplest case will be where the tree is either completely left or right skewed (see left tree above). This tree will contain $h + 1$ nodes.

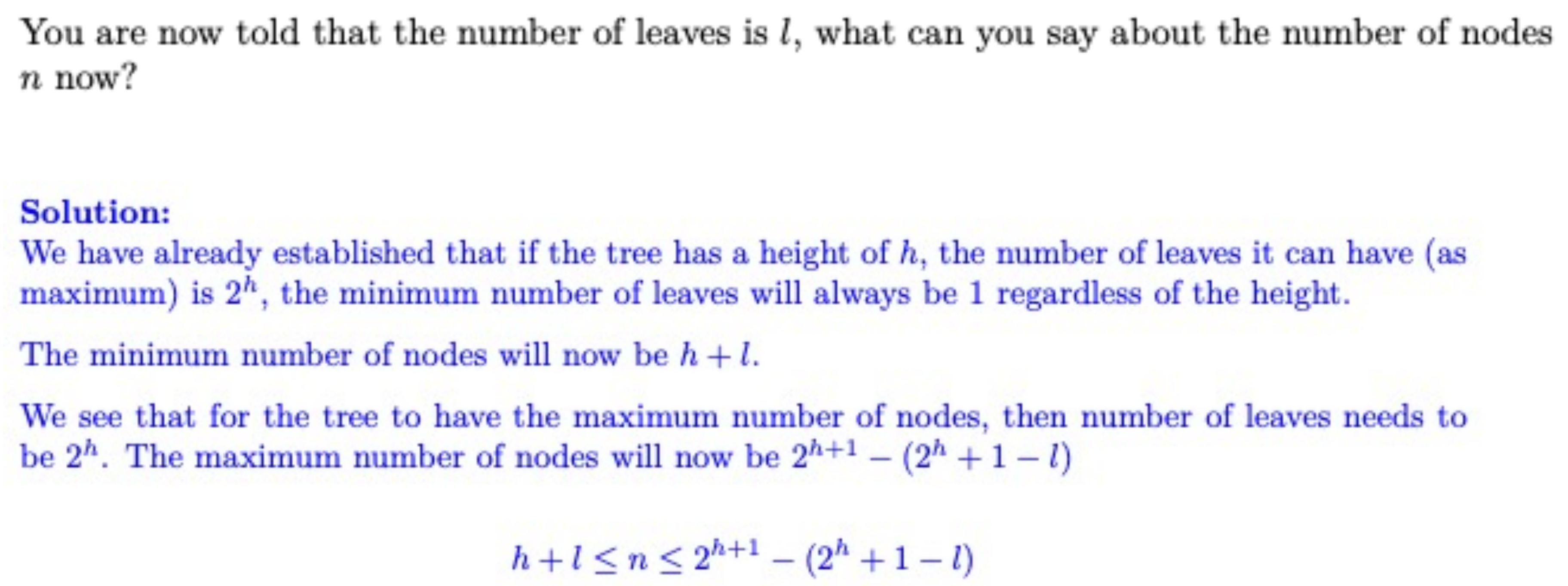
In the other end we have the maximum number of nodes (see right tree above). Lets start with a height of 0, this tree only contains the root ($2^0 = 1$ node). For each level we add below the root (height increases by 1), the number of new nodes added are equal to 2^h, so for the first level $2^1 = 2$ new nodes are added, and when increasing it to a height of 2 $2^2 = 4$ new nodes are added. In general we get that the maximum number of nodes in a binary tree with height h is:

$$\sum_{k=0}^{h} 2^k = 2^{h+1} - 1$$

We find that:

$$h + 1 \leq n \leq 2^{h+1} - 1$$

Task 3 b) ii.



You are now told that the number of leaves is l, what can you say about the number of nodes n now?

Solution:
We have already established that if the tree has a height of h, the number of leaves it can have (as maximum) is 2^h, the minimum number of leaves will always be 1 regardless of the height.

The minimum number of nodes will now be $h + l$.

We see that for the tree to have the maximum number of nodes, then number of leaves needs to be 2^h. The maximum number of nodes will now be $2^{h+1} - (2^h + 1 - l)$

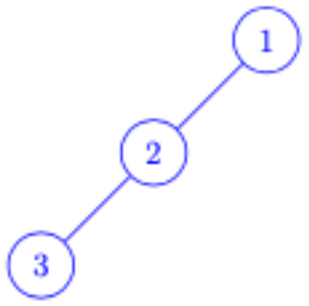
$$h + l \leq n \leq 2^{h+1} - (2^h + 1 - l)$$

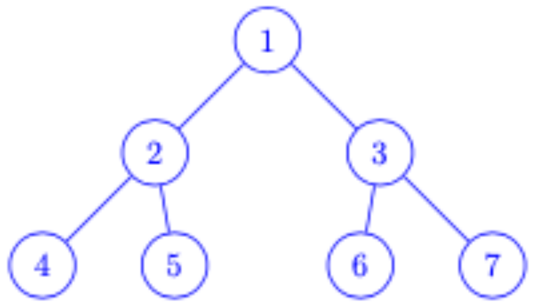
Task 3 c)

The efficiency of a search through a binary tree is dependant upon the height of the tree. The higher the tree is the more comparisons has to be done to reach the furthest leaves. How should the root of a binary tree be selected to decrease its height?

Solution:

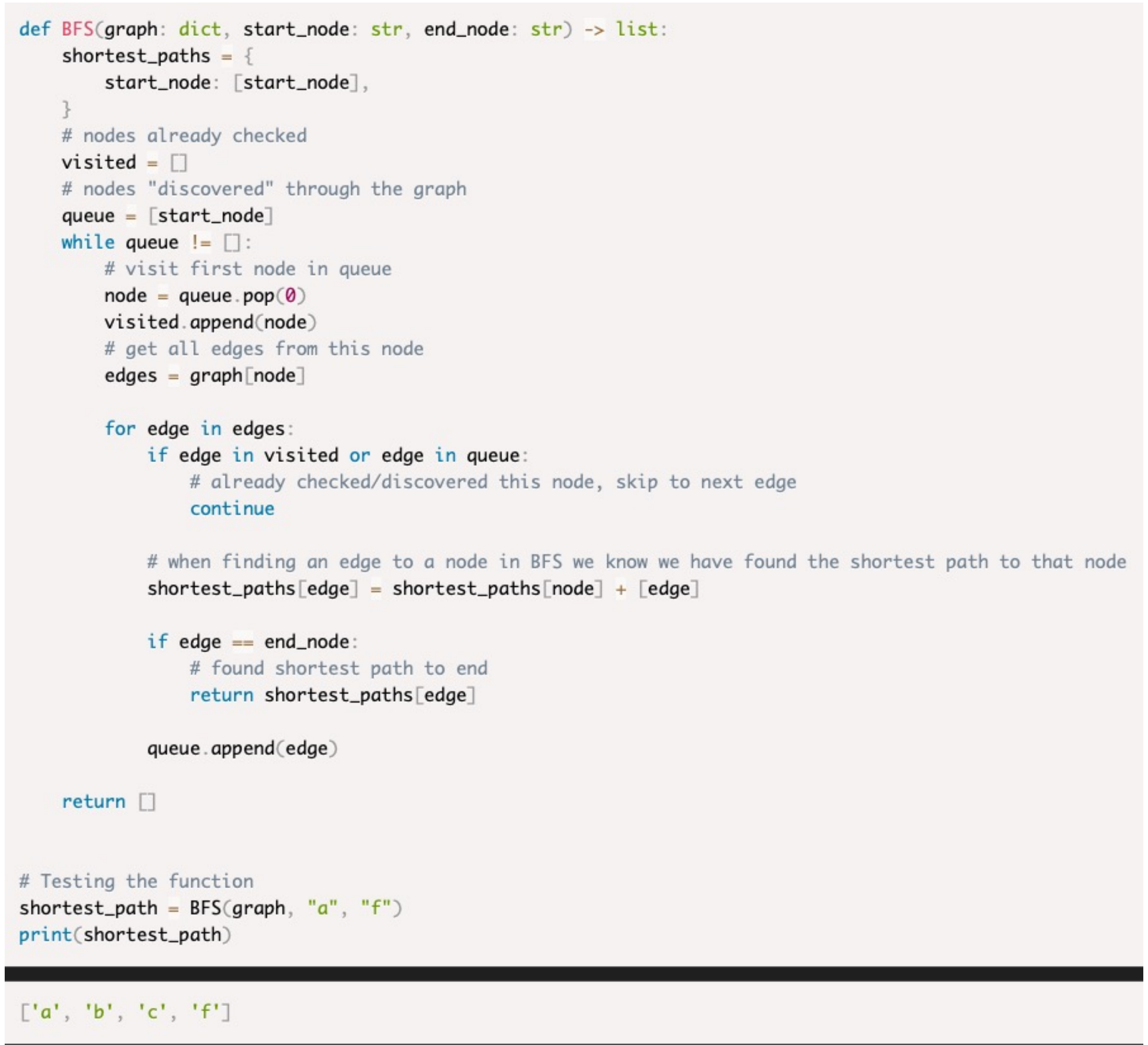
The root should be selected such that the tree is as balanced as possible. For example, in a binary tree containing numbers, we want the root to be the median of all the numbers, that way the amount of numbers smaller than the root(and to the left in the tree), will be roughly equal to the amount of numbers larger than the root (to the right in the tree).





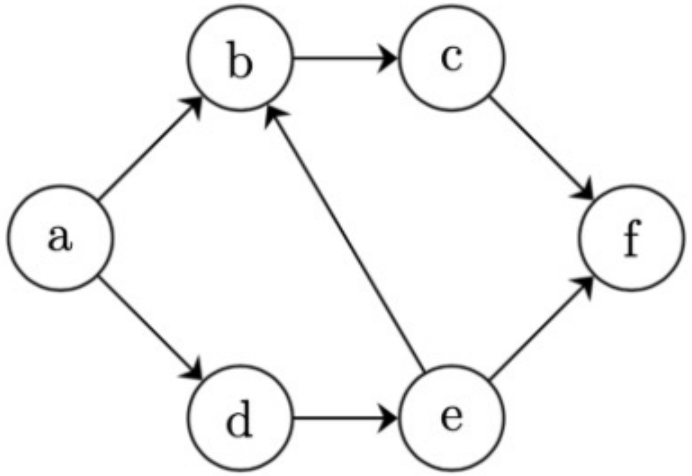
Programming – Task 1

BFS



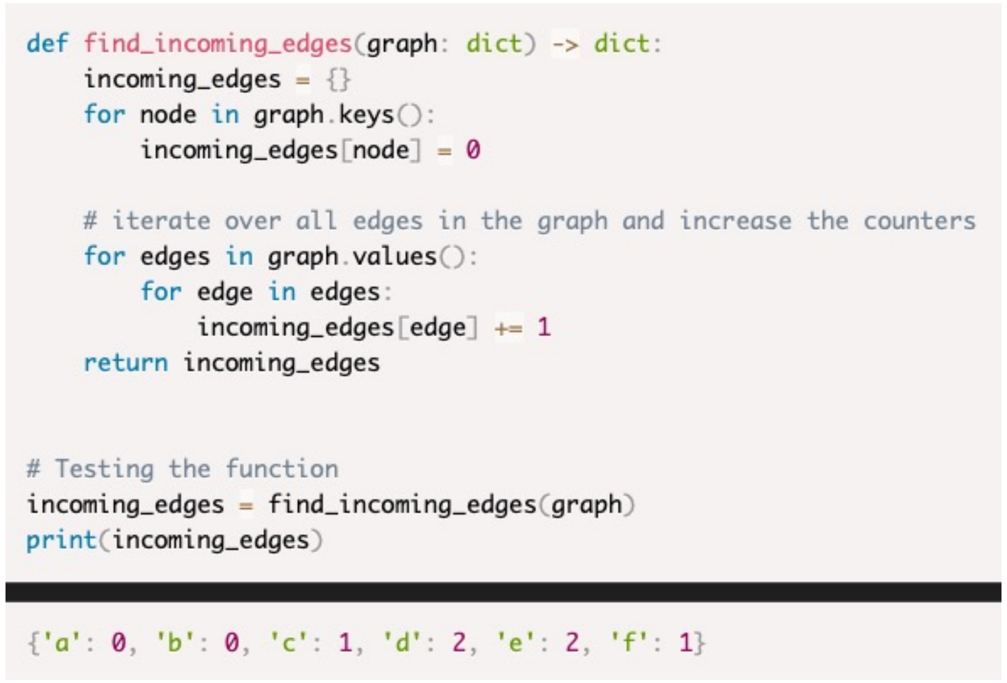
```python
def BFS(graph: dict, start_node: str, end_node: str) -> list:
    shortest_paths = {
        start_node: [start_node],
    }
    # nodes already checked
    visited = []
    # nodes "discovered" through the graph
    queue = [start_node]
    while queue != []:
        # visit first node in queue
        node = queue.pop(0)
        visited.append(node)
        # get all edges from this node
        edges = graph[node]

        for edge in edges:
            if edge in visited or edge in queue:
                # already checked/discovered this node, skip to next edge
                continue

            # when finding an edge to a node in BFS we know we have found the shortest path to that node
            shortest_paths[edge] = shortest_paths[node] + [edge]

            if edge == end_node:
                # found shortest path to end
                return shortest_paths[edge]

            queue.append(edge)

    return []


# Testing the function
shortest_path = BFS(graph, "a", "f")
print(shortest_path)
```

```
['a', 'b', 'c', 'f']
```



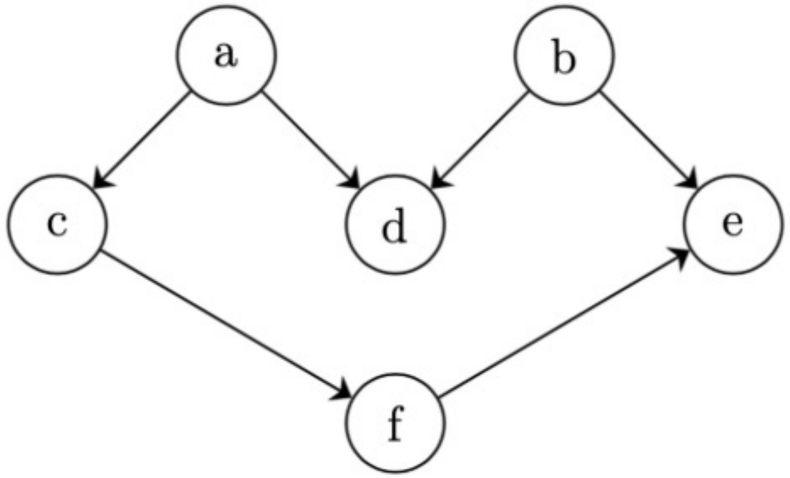
$$\begin{tikzcd} & b \arrow[r] & c \arrow[dr] & \\ a \arrow[ur] \arrow[dr] & & & f \\ & d \arrow[r] & e \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Programming – Task 2
Find all topological orderings
finding incoming edges

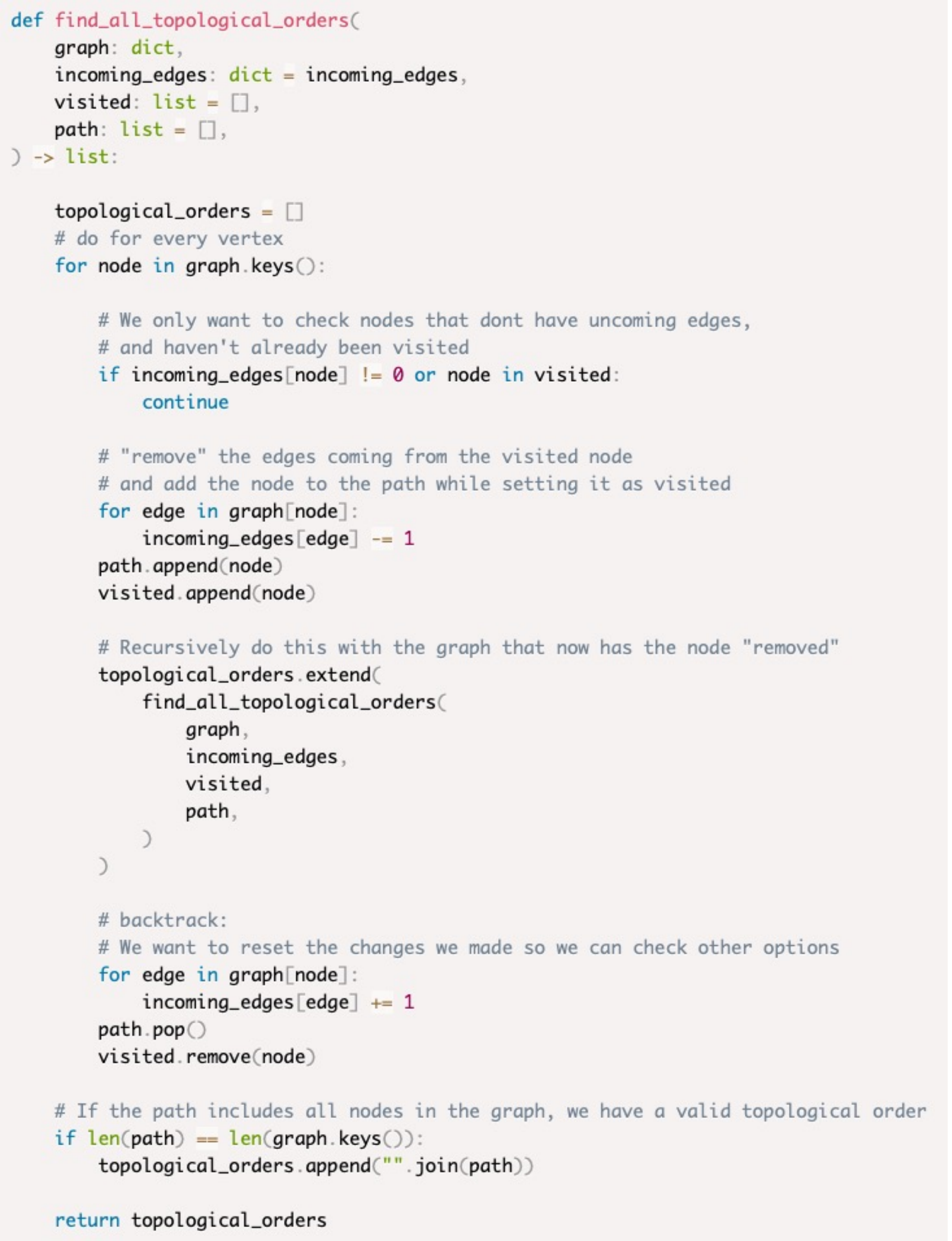


```python
def find_incoming_edges(graph: dict) -> dict:
    incoming_edges = {}
    for node in graph.keys():
        incoming_edges[node] = 0

    # iterate over all edges in the graph and increase the counters
    for edges in graph.values():
        for edge in edges:
            incoming_edges[edge] += 1
    return incoming_edges


# Testing the function
incoming_edges = find_incoming_edges(graph)
print(incoming_edges)
```

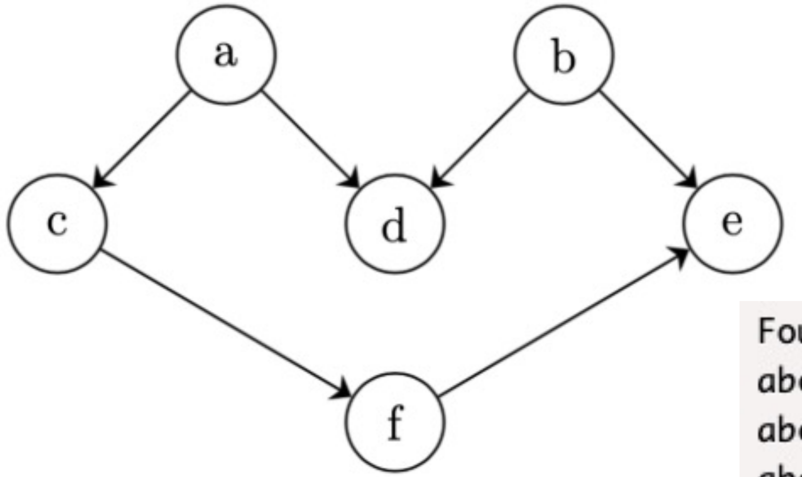
```
{'a': 0, 'b': 0, 'c': 1, 'd': 2, 'e': 2, 'f': 1}
```

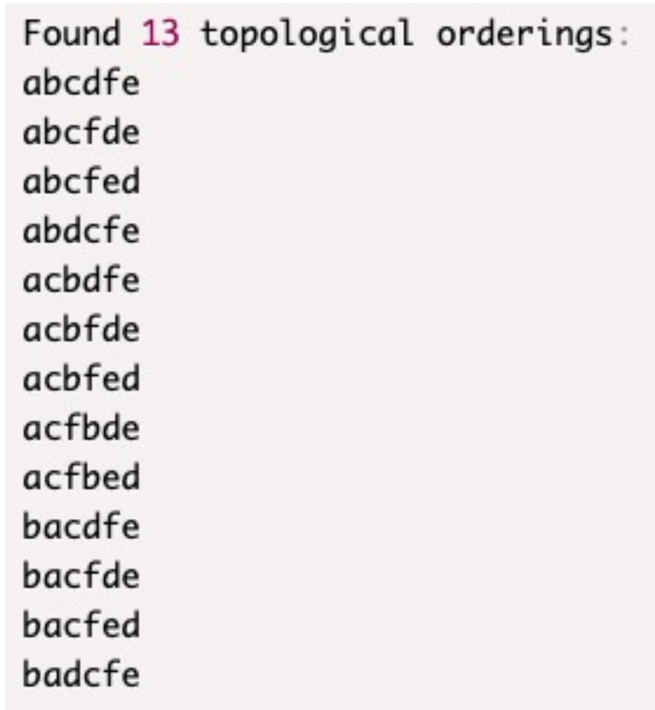

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & & b \arrow[dl] \arrow[dr] & \\
c \arrow[drr] & & d & & e \\
 & & f \arrow[urr] & &
\end{tikzcd}$$

Programming – Task 2
All topological orderings



```python
def find_all_topological_orders(
    graph: dict,
    incoming_edges: dict = incoming_edges,
    visited: list = [],
    path: list = [],
) -> list:

    topological_orders = []
    # do for every vertex
    for node in graph.keys():

        # We only want to check nodes that dont have uncoming edges,
        # and haven't already been visited
        if incoming_edges[node] != 0 or node in visited:
            continue

        # "remove" the edges coming from the visited node
        # and add the node to the path while setting it as visited
        for edge in graph[node]:
            incoming_edges[edge] -= 1
        path.append(node)
        visited.append(node)

        # Recursively do this with the graph that now has the node "removed"
        topological_orders.extend(
            find_all_topological_orders(
                graph,
                incoming_edges,
                visited,
                path,
            )
        )

        # backtrack:
        # We want to reset the changes we made so we can check other options
        for edge in graph[node]:
            incoming_edges[edge] += 1
        path.pop()
        visited.remove(node)

    # If the path includes all nodes in the graph, we have a valid topological order
    if len(path) == len(graph.keys()):
        topological_orders.append("".join(path))

    return topological_orders
```





```
Found 13 topological orderings:
abcdfe
abcfde
abcfed
abdcfe
acbdfe
acbfde
acbfed
acfbde
acfbed
bacdfe
bacfde
bacfed
badcfe
```

Questions?

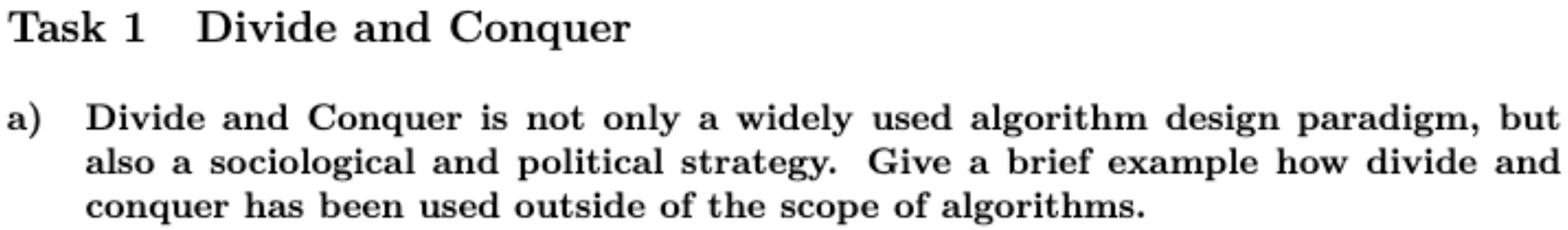
For assignment 2 solution

Assignment 3 Walkthrough

Available now!

Deadline is **4. October** (in 2 weeks)

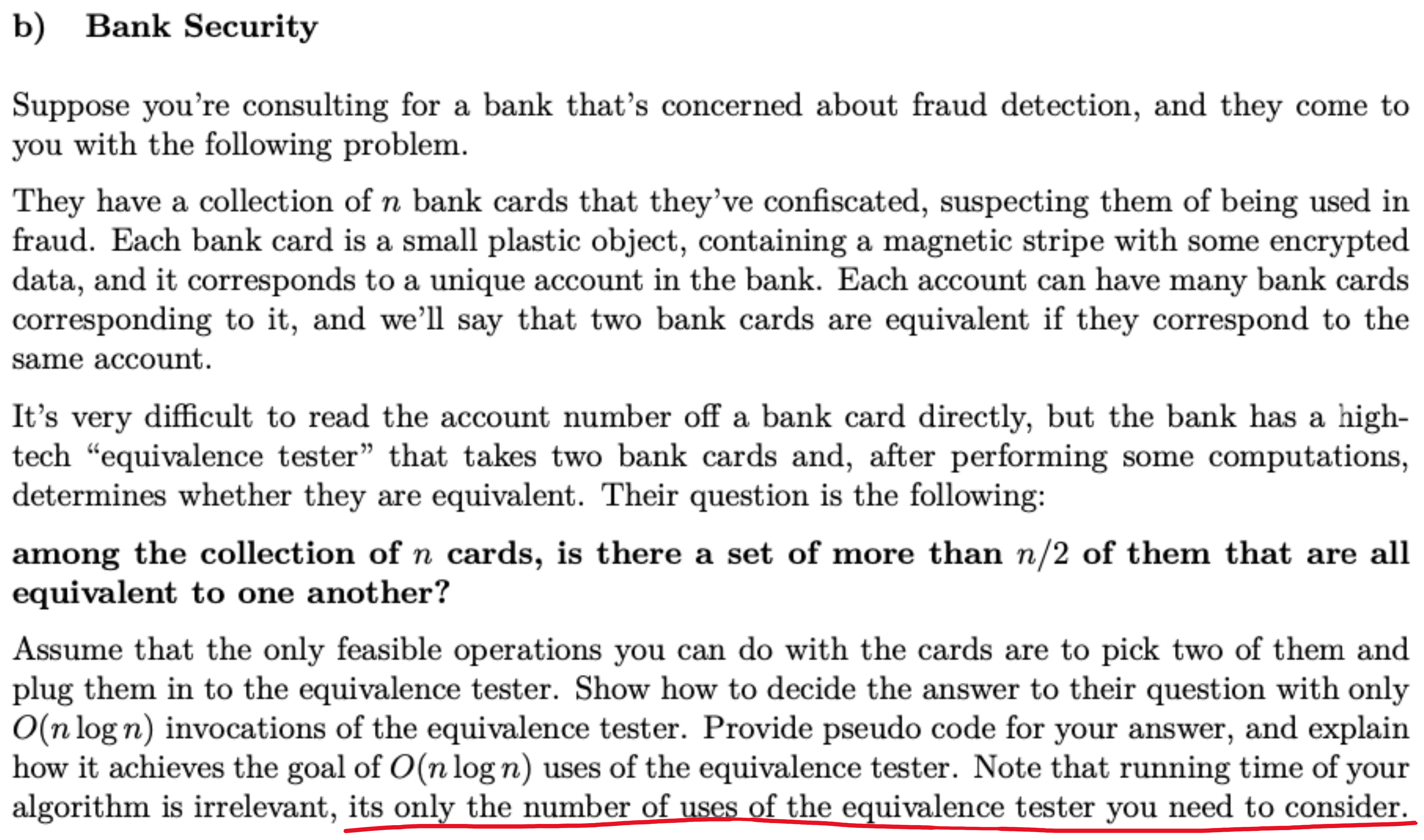
Task 1 a)



Task 1 Divide and Conquer

a) Divide and Conquer is not only a widely used algorithm design paradigm, but also a sociological and political strategy. Give a brief example how divide and conquer has been used outside of the scope of algorithms.

Task 1 b)



b) Bank Security

Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem.

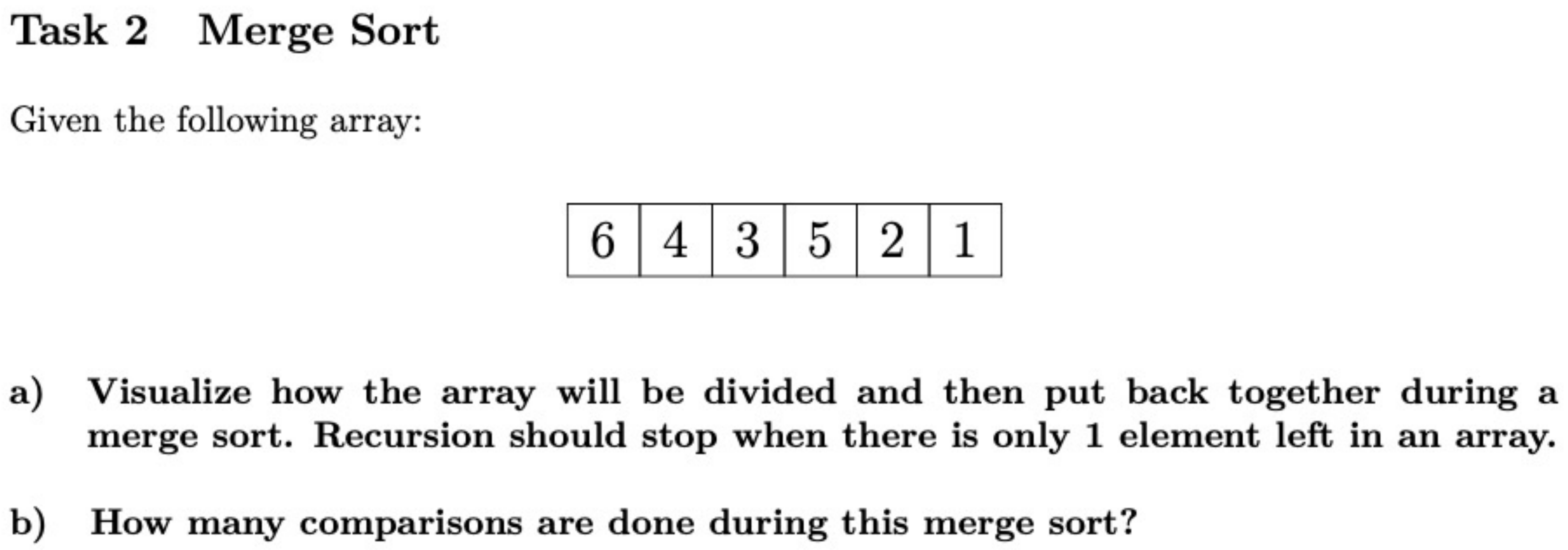
They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account.

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent. Their question is the following:

among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another?

Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester. Provide pseudo code for your answer, and explain how it achieves the goal of $O(n \log n)$ uses of the equivalence tester. Note that running time of your algorithm is irrelevant, its only the number of uses of the equivalence tester you need to consider.

Task 2



Task 2 Merge Sort

Given the following array:

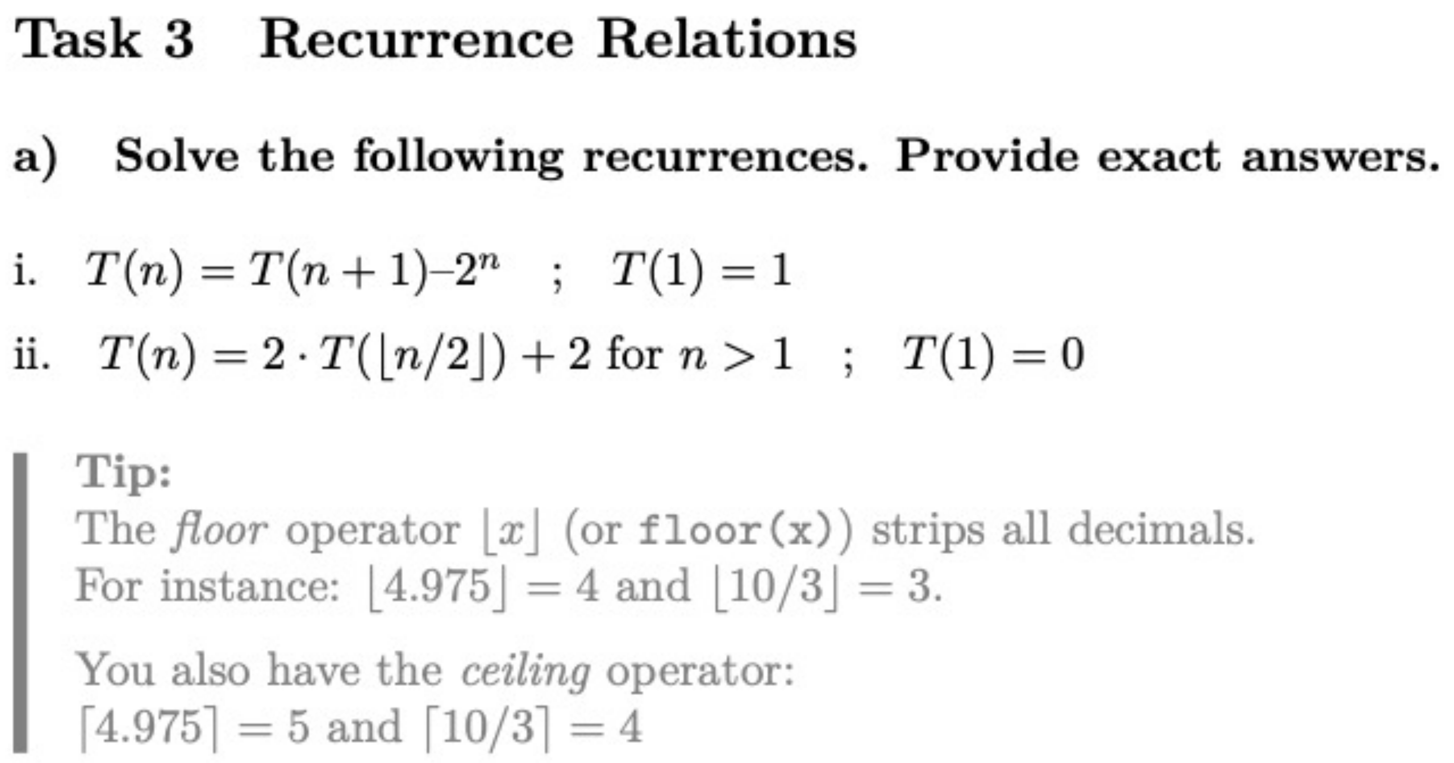
6	4	3	5	2	1

a) Visualize how the array will be divided and then put back together during a merge sort. Recursion should stop when there is only 1 element left in an array.

b) How many comparisons are done during this merge sort?

Task 3

See lecture for examples of how to solve these



Task 3 Recurrence Relations

a) Solve the following recurrences. Provide exact answers.

i. $T(n) = T(n+1) – 2^n \quad ; \quad T(1) = 1$

ii. $T(n) = 2 \cdot T(\lfloor n/2 \rfloor) + 2$ for $n > 1 \quad ; \quad T(1) = 0$

Tip:
The *floor* operator $\lfloor x \rfloor$ (or `floor(x)`) strips all decimals.
For instance: $\lfloor 4.975 \rfloor = 4$ and $\lfloor 10/3 \rfloor = 3$.

You also have the *ceiling* operator:
$\lceil 4.975 \rceil = 5$ and $\lceil 10/3 \rceil = 4$

Task 4

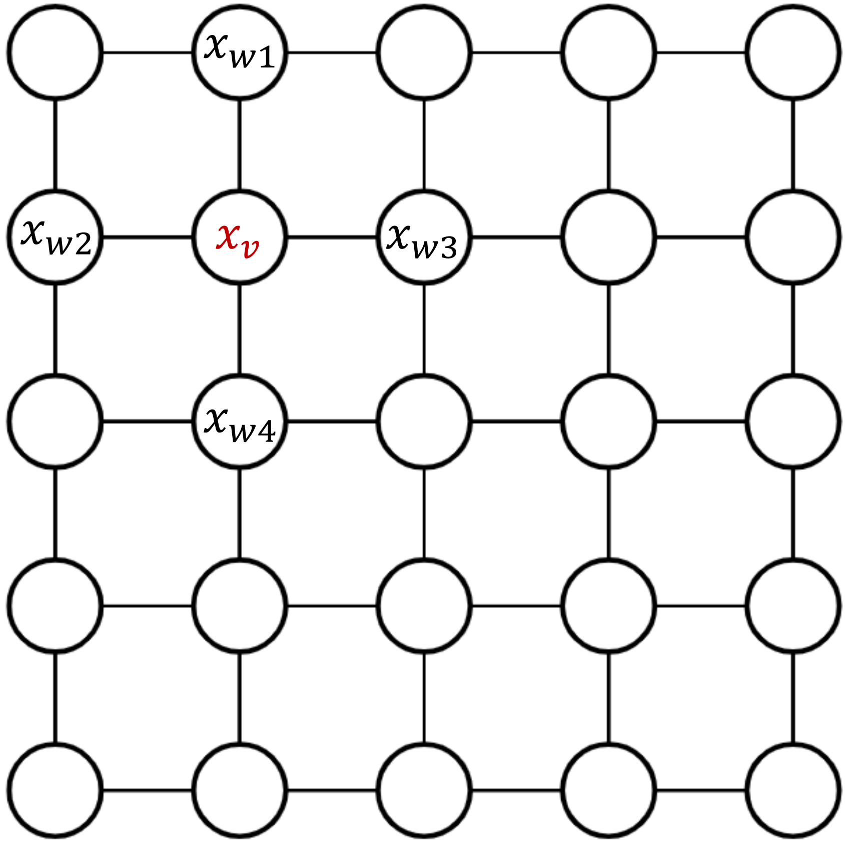
Task 4 Local Minimum of a Grid Graph

Suppose that you are given an $n \times n$ grid graph G.

> An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \le i \le n$ and $1 \le j \le n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$.

Each node v is labeled by a real number x_v; you may assume that all these labels are distinct. A node v of G is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. The labeling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v.

Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 total nodes.)



Programming

- Implementing Merge Sort!
    - Numbers *and* letters

Questions?

For assignment 3