# Rubrik - DSA/LLD Coding

## Equivalent Lists

You are given two lists of numbers. Write a function which returns true if both contain same elements i.e. if number ‘x’ is in list 1 then its also in list 2 and vice versa.

Example:

- A = [1, 4, 2, 2, 1, 3, 4,]
- B = [1, 2, 3, 4]
- C  = [1, 2, 3]

Here A and B both are equivalent to each other.

**Solution**

The solution is to create a set from each list (to avoid duplication). Then iterate through each element and see if that element is present in another set or not.

Usually a good follow up question to ask

- How would you convert a list to a set ?
- What is the time complexity of it ?

As they are discussing it, spend time to figure out what all needs to be done to create a set from a list.

It is important to set up a point here that creating a set from a list is an expensive operation even if its O(N) complexity

# Reachable Nodes with IP

Given a starting node in a directed graph of immutable nodes, find a set of nodes that either
(a) have IP address
(b) or can reach the node having IP Address.

```
A(f)  -> B(f) ----------------------------> V1(f)
 |       |                              ^
 |       +-----> V2(f)                   |
 |                                      |
 +-----> C(f) -> D (f) ----------------------+
 |       |                              |
 |       +-----------------------------------+
 |                                      |
 +------------------------------------------+
```

(t) -> Node has IP address
(f) -> Node doesn't have an IP address

Node in the graph is represented as:
Node {
  label -> String // unique in the graph
   connections -> List [Node]
   hasIpAddress -> Boolean
}

In all 'independent' examples below, 'A' is the starting node:
1. V2 has IP address -> Solution(V2, B, A)
2. B has IP address   -> Solution(B, A)
3. C has IP address   -> Solution(C, A)
4. V1 has IP address  -> Solution(V1, B, A, D, C)
5. D has an IP Address -> Solution(?)

Clarifications (when candidates ask)

(a) Graph can have cycles

(b) Nodes in the graph are unique. Multiple nodes simply indicate that they have multiple parents

(c) No need to worry about 'undiscoverable' nodes

(d) Order of nodes in the solution is not important. (In fact, there can't be any)

(e) Graph and all its nodes are IMMUTABLE.

Phase II: Ask time and space complexity of code (not just algorithm)

Note: Please pay special attention to linear time complexity of adding an entire set to another set.

## Calendar

Implement a Calendar with `addEvent` and asynchronous notifications

Users can register an event with a timestamp, list of emails to send the notification, message and it could be recurring.
Calendar should generate an asynchronous notification using `sendEmail()` API if there is an event at the current timestamp. The calendar should send an event the next day at the same time if the event is recurring.

You need to implement the following API:

```
def addEvent(time, [email], message, isRecurring)
```

Assume below API is available (you don't need to implement):

```
def sendEmail(email, message) -> None
```

e.g:

def addEvent("2024-05-05 13:00:00", ['abc@xyz.com', 'def@qwe.com'], "Hi", True) <--- Send notification "Hi" at 1:00pm and it is recurring
def addEvent("2024-05-06 14:31:00", ['abc@xyz.com', 'uvw@qwe.com'], "Hello", False) <--- Send notification "Hello" at 2:31pm

**How do you proceed asking this question?**

We can tell the candidate that we want to implement a simple version of Google Calendar. The user should be able to add events to the calendar, and the calendar should have the functionality to send notifications to corresponding email addresses when an event starts (e.g. when the current time is the start time of an event).

We may also need to point out that the whole app will be an “in-memory“ app, which means that nothing is related with external storage or any DB stuff. And the `sendEmail` API is already available, which is something like an AWS SDK that we can just call and an email will be sent.

If the candidate asks if there's a standard datetime library available that can parse date time string to a dateTime object, we can tell them it’s available.

If the user is not familiar with Threading stuff and gets stuck, we can provide them some mock APIs similar to `Java`'s `wait()` and `notify()`, or `JavaScript`'s `setTimeOut` and `clearTimeOut`:

```
sleepWithTimeOut(ms); // will block the function untill wakeUp()
is called or timeouts

wakeUp()              // to manually wake up a sleep function
```

**Possible Solutions:**

The optimal solution of the question is to use a `heap`/ `priorityQueue` to store the events, and use timestamp as the key. The key idea is to easily get the next event’s start time, and calculate the time difference between the next event and current time, then sleep till that time and wake up to send the emails. If the event is recurring, add the time by 1 day, and add it back to the heap.

A **critical edge case** is that, while we sleep till the next event, a new event is added, and the new event’s start time is earlier than the current “pending“ event. In this case, the calendar should “cancel“ the waiting, and start a new one.

Other edge cases such as empty heap should also be taken care of.

A simple pseudo-python example:

## Common pitfalls

- A big portion of candidates may start with using a map to store the events, where the key is the timestamp, or just use an array to store events. And have a busy while loop to frequently "poll" the or to check every X seconds, and go through the map/array and to see if we need to send any emails. This naive method doesn't actually work, since we want to send the email at the *exact* time of an event start time, and we don't want any delay.

## Common hints

- If the candidate can not change their mind from "polling" every X seconds to calculating the time difference between events, we may provide them a static set of events, and ask them to schedule sending emails just on these static events(that is the user won't add new events any more). This might help them to realize how to calculate time diffs rather than using a fixed time interval to sleep.
- If the candidate doesn't find out the critical edge case, we may also need to provide some examples that run into that edge case.

```
import heapq
import time
import threading

class Event:
    def __init__(self, time, emails, message, is_recurring):
        self.time = time
        self.emails = emails
        self.message = message
        self.is_recurring = is_recurring

    def __lt__(self, other):
        return self.time < other.time

class Calendar:
    def __init__(self, height, width):
```

```python
        self.events = []  # min-heap
        self.next_event_time = None
        self.timeout_handler = None
        self.lock = threading.Lock()

    def add_event(self, event_time, emails, message, is_recurring):
        event = Event(event_time, emails, message, is_recurring)

        with self.lock:
            if self.next_event_time is None:
                heapq.heappush(self.events, event)
                self.schedule_next()
            elif event_time < self.next_event_time and self.timeout_handler
is not None:
                heapq.heappush(self.events, event)
                self.timeout_handler.cancel()
                self.schedule_next()
            else:
                heapq.heappush(self.events, event)

    def schedule_next(self):
        if not self.events:
            return

        next_event = self.events[0]  # peek
        current_time = time.time()
        time_diff = max(0, next_event.time - current_time)

        self.next_event_time = next_event.time

        self.timeout_handler = threading.Timer(time_diff, self.consume)
        self.timeout_handler.start()

    def consume(self):
        with self.lock:
            if not self.events:
                return

            event = heapq.heappop(self.events)
            self.send_email(event)

            if event.is_recurring:
                event.time += 86400  # 1 day in seconds
                heappq.heappush(self.events, event)

            if not self.events:
                self.next_event_time = None
                self.timeout_handler = None
            else:
                self.schedule_next()

    def send_email(self, event):
        print(f"Sending email to {event.emails} with message:
'{event.message}' at {time.ctime(event.time)}")
```

## Quadruplet Sum

**NOTE:**

1. Even if the candidate has done Pair Sum and Triplet Sum, this question entails handling lots of edge cases especially with duplicates.
2. If you have a hunch or if the candidate has already solved Quadruplet sum, after coding it up, he should solve/answer most or ALL of the bonus questions.

Given an array of unsorted numbers, find all unique quadruplets in it that add up to given value, two quadruplets Q1 and Q2 are different if their multiset difference is non empty.

The array can have duplicates. A quadruplet can have duplicate elements, the below examples make it clear.

```
Input: [10, 2, 3, 4, 5, 9, 7, 8], target = 23
Output: [[2, 3, 8, 10],[2, 4, 7, 10],[3, 4, 7, 9],[2, 4, 8, 9],[2, 5, 7,
9],[3, 5, 7, 8]]
Explanation: There are six unique quadruplet whose sum is 23

Input: [2,2,2,2,4,4,1,1,1,1], target = 8
Output: [[1, 1, 2, 4],[2, 2, 2, 2]]
Explanation: There are two unique quadruplets whose sum is 8

Input: [2,2,2,6], target = 8
Output: []
Explanation: There are no quadruplets whose sum is 8
```

Lets say n1, n2, n3, n4 add up to target, the following approaches are possible

1. **Time: O(n^4), Space O(1)** Naive brute force solution has time complexity
2. **Time: O(n^3), Space O(1)** Sort the array using quicksort. Iterate over for n1, n2 and use two pointers one form the start and other at the end of the remaining array.
3. **Time: NOT O(n^2 logn) Please see notes, Ω(n^3), O(n^4), Space O(n^2)** following the following steps
    a. Iterate through the array and store all possible pair sums with their respective indices in a simple struct/class
    b. Sort the pair sum array(quicksort)
    c. Use two pointer approach on the pair sum array
    d. Avoid pair sums which use the same elements by comparing their indices
4. **Time: NOT O(n^2) Please see notes, Ω(n^3), O(n^4), Space O(n^2)**
    a. Store all pair sums with their respective indices in hash map
    b. iterate through the map and if X is the current sum, find if target - X is found in the map
    c. same as 3d

**Notes:**

1. The lower bound time complexity of a problem when all elements are distinct is n^3, so this problem's lower bound is n^3: https://leetcode.com/problems/4sum/discuss/8565/Lower-bound-n3. The worst case scenario in methods 3 and 4 is actually O(n^4) when all elements are the same. We could use something smarter
2. We aim for a workable solution which compiles and runs correctly at the end of the interview. If candidate is not able to come up with Approach 3 or 4 after hints, let them implement their best thought of solution.

## Ping Logs

Sure! Here's a well-formatted version of your problem statement that you can directly paste into a Google Doc:

---

# Node Ping Query in a Distributed Cluster

### Description

In a distributed system, multiple nodes work together to process data and maintain system reliability. To ensure the health of the system, nodes are periodically pinged to check their availability and responsiveness. These pings are logged with timestamps and can be used to analyze node behavior and system stability.

In this problem, we are interested in determining how many nodes were **not successfully pinged** in a given time interval. By analyzing the ping logs, we can derive insights into node outages or communication issues within the system.

---

## Problem Statement

You are given:

- A multi-node cluster with `N` nodes.
- A list of ping logs: each entry is a tuple `<nodeId, timestamp>`, indicating that the node was pinged at a specific time.
- An integer `k` that defines the size of the time window.
- A list of queries, where each query is a timestamp `t`.

For each query, compute the number of nodes that were **not pinged at least once** in the time window `[t - k, t]`.

---

## Input

- `N`: Integer — the number of nodes in the cluster
- `ping_logs`: List of tuples — each tuple is in the form `[nodeId, timestamp]`
- `k`: Integer — the length of the time window
- `queries`: List of integers — each query is a timestamp

---

## Output

- A list of integers where each element represents the number of nodes that were not pinged in the time window `[queries[i] - k, queries[i]]`.

---

## Sample Scenario

**Cluster Size:** `N = 3`
**Ping Logs:** `[[1, 3], [2, 6], [1, 5]]`
**k:** `5`
**Queries:** `[10, 11]`

**Step-by-step Explanation:**

**Query 0:**
Window = `[10 - 5, 10]` → `[5, 10]`
Ping logs in this window:

- Node 1 was pinged at time 5
- Node 2 was pinged at time 6
- Node 3 was not pinged

**Result:** Only Node 3 was not pinged → Output: `1`

---

**Query 1:**
Window = `[11 - 5, 11]` → `[6, 11]`
Ping logs in this window:

- Node 2 was pinged at time 6
- Node 1 and Node 3 were not pinged

**Result:** Two nodes (1 and 3) were not pinged → Output: `2`

---

## Final Output

[1, 2]

---

Let me know if you want this styled with bold sections, inline code formatting, or in table format for easier reading!

## Problem

Let’s say we have apps which can be installed on a system. Each app requires d bytes of space while downloading and takes up s bytes of space after the download is complete. Notice that, to install an app, we need at least max(d, s) bytes of space available. All three cases are possible: d < s, d = s, and d > s.

**Initial problem (no coding)**

If we need to install two apps (d_i, s_i) and (d_j, s_j), which order should we install them in to minimize the space required?

Hint: Assuming the system can handle the storage space of both apps, we should try to minimize the temporary space needed while the second app is downloading. If i is installed first, how much space is needed while j is downloading? What if j is installed first?

Solution:

We should first install whichever app takes the least storage space compared to its download space. In other words, we should install in ascending order of (s - d).

Formal proof of solution: if i is installed first, the space needed while downloading is s_i + d_j. If j is installed first, we need s_j + d_i. So we should install i before j if (s_i + d_j) < (s_j + d_i) => (s_i - d_i) <  (s_j - d_j). If the candidate figures out the first inequality, show them how this transforms into the second inequality, since it may not be readily clear to them why that transformation is helpful.

If the candidate intuitively knows we should order by (s - d), no formal proof is necessary. Walking through the steps of the formal proof can be helpful though for candidates who need to figure out the ordering.

We now know that, when installing some collection of apps, installing them in order of (s - d) will minimize the space required.

However, this does not mean that apps with lower (s - d) values should always be installed; for example, consider: A(d = 6, s = 5) and B(d = 3, s = 4). If we install both apps, A → B is the optimal installation order. However, if there is only 8 space available (s.t. we cannot install both apps), skipping A and just installing B uses the least amount of space. There's no easy metric to determine whether an app *should* be installed or not. But the selection of apps which are best to install should be installed in relative order of (s - d). Make sure to emphasize this fact to the candidate, because it is important going forward.

**Advanced problem**

Initially there are no apps, x bytes of free space, and a collection of N apps we can attempt to download. Find the maximum number of apps we can install together.

Note: Emphasize to the candidate that time complexity is not a major concern for us, and a brute-force solution is completely appropriate.

Solution:

Sort all apps by (s - d), and for each app, evaluate the maximum subsequent apps installable in the case where i) the current app is installed, and ii) the current app is not installed. DP can be used to optimize this calculation, but it is not required for the candidate to pass or even to get a strong yes. If a candidate is considering the DP route but seems to not be confident about it, you are encouraged to let them know that it is not required.

<u>Bonus (worth +.2 points)</u>: return the optimal collection of apps which should be installed.

**Problem**

Let’s say we want to write a text editor or notepad-like application. We want to be able to store a potentially large body of text which supports the following operations:

- insert(offset, text) - inserts the chunk of text starting at the given offset
- delete(offset, length) - deletes all the characters in the range [offset, offset + length]

**Naive solution**

Store the text as one long string. This isn’t great since we have to reconstruct the entire string after each update (every operation is O(length of string)). This solution should be improved upon before coding begins. Ask the candidate: What happens if the string is MB/GB long?

**Intermediate solution** - Store the inserted chunks in an ordered linked-list. Important edge case: notice that insert and delete might possibly split chunks. Here operations are O(# of chunks) because we might need to run down the entire list of chunks to find the offset we are looking for.

<u>Follow-up problem</u>: How would you modify this solution to support multiple levels of undo/redo?

Solution: Keep a stack recording past operations and a stack recording “redoable” operations. When undoing an action, perform the complement of that action.

**Advanced solution** - Is there anything you can think of which could make the offset lookup faster? Answer: store the chunks in a binary tree or skip-list. Here insertion would be O(log(# of chunks)), and in the case of deletion an additional  O(# of chunks being deleted)).

<u>There is a big complication here</u>: How can we tell at any particular node what offset we are at? The solution it to have each parent node record the total length of the strings in its left-subtree, so that we can do a modified binary-search to find a particular offset (

Rope (data structure) ). When creating a node or updating a node’s length, we need to update the lengths of the parent/ancestor nodes to keep this tracking consistent.

This solution is complex and has many parts. If the candidate gets here, emphasize that we are not expecting them to complete this in code; this is best treated first as a high-level exercise. Allow the candidate to complete this in pseudocode, and tell the candidate to not worry about balancing any tree-based solution.