**Apartment Hunting**:

You are looking to move into a new apartment, and you are given a list of blocks where each block contains an apartment that you could move into. In order to pick your apartment, you want to optimize its location. You also have a list of requirements: a list of buildings that are important to you.

For instance, you might value having a school and a gym near your apartment.

The list of blocks that you have contains information at every block about all of the buildings that are present and absent at the block in question. For instance, for every block, you might know whether a school, a pool, an office, and a gym are present.

In order to optimize your life, you want to minimize the farthest distance you would have to walk from your apartment to reach any of your required buildings.

Write a function that takes in a list of blocks and a list of your required buildings and that returns the location(the index) of the block that is most optimal for you.

If there are multiple most optimal blocks, your function can return index of any one of them.

It has got a canonical less optimal and more optimal solution.

We are given a lot of information in the prompt, some of which is actually not at all useful for the problem.

This is often done in interviews because the interviewee is tested on their ability to be able to parse the prompt and identify what information is actually going to be useful and what information is not going to be useful.

List of hash tables. List represents blocks in a city.

Each index represents different block of the city and every hash table in that list represents the buildings that are either present or absent at a given block.

All of those buildings that map to false, we can almost ignore them entirely.

Answer: Index 3

At index 2, gym and school are super close. But we will have to walk 2 blocks to get to the store.

We have to find a block that has an apartment that is going to have to walk the least amount of distance to reach all of the important buildings.

Gym, school and store matter to us.

The first way to approach this is to iterate through these blocks and calculate distances.

At every block, we go through all of our required buildings.

For gym, we will see how much minimum distance it takes to get to the gym from all indices in the list.

How far is the closest gym from index 0 ?

How far is the closest school from index 0, our current index ?

It’s right there, 0 away.

For block 0, we iterate through all of our 3 requirements.

We have all the distances of the closest of each building that is important to us.

How far away is the farthest one of these buildings?

For index 0,

G, Sc, St

1 0 4

We take the maximum of these 3.

Our final decision for choosing a particular block as our apartment will depend on the maximum distance of a particular required place.

We have to take the minimum of these maximum values for each index.

For index 1,

G, Sc, St

0 1 3

This means that, if we take our apartment at index 1, we have to walk atleast 3 blocks to reach all of the 3 required buildings.

We can see here that index 1 is better than index 0.

What we are doing is, we are iterating through the entire array of blocks, then at each block we are iterating through the entire array of required buildings and **for each of those**, we are re-iterating the entire array of blocks to find the closest distances.

3 nested for loops.

Time Complexity: O(B^2\*R)

B represents the number of blocks that we have in our list and R represents the number of requirements that we have in our list.

Constant time operations (Comparisons) for each iteration.

Space complexity: O(B)

Creating an extra to keep track of maximum values.

We can avoid creating a list. Can keep track of a variable that says ‘smallest distance’ and keep track of index at which it will be overwritten.

**Can we do better from a time point of view? Yes**

It seems like we will have to iterate through all of the blocks and requirements once.

Thing that bothers is B^2 term and we can eliminate the B^2 term.

In the 3rd for loop, where we re-iterate through all of the blocks, so that for each requirement we can find the nearest building of that requirement and find that closest distance.

Is there some way in which we can pre compute that such that our iterations are then fewer? Yes

Precomputing technique.

This technique could be applicable in a lot of algorithm problems.

Let’s say we have a list and we have a bunch of for loops to compute values, ask yourself, maybe there is a way to precompute values and then optimize like that.

For all of our requirements, imagine that we already knew all of the closest distances from each index.

Imagine for the gym, we knew that at index 0, the closest gym was one block away. At index 1, the closest gym was 0 blocks away and so on.

G: [ 1, 0, 0, 1, 2 ]

Similarly imagine that we knew this for schools and store as well.

School: [ 0, 1, 0, 0, 0 ]

Store: [ 4, 3, 2, 1, 0 ]

We were calculating these values in the 3rd for loop in previous solution.

If we have these values, then we could literally just iterate through all of the blocks and for index 0, take the maximum of the 3 values at all the respective 0 indices and so on.

Store these maximum values and then take their minimum.

Imagine we are given these 3 arrays, the solution will then just be O(BR) time complexity.

We just do one iteration at each of the blocks and for each of the blocks, we calculate the maximum values for each of the r distances (r rows, number of requirements).

We can compute these values in a time efficient manner in O(BR) or less.

If we can compute the distances row for any of the requirement in O(B), then we can do all of these computations ‘r’ times.

For every requirement, we can do 1 pass from left to right.

Iterating for gym:

For each block, Keep track of the **closest/nearest** gym that came before me.

At index 0, there is just school. No gym is before it.

At index 1, there is 1 gym and it is 0 blocks away.

For gym,

[ X, 0, 0, 1, 2 ]

Now, we do another iteration from right to left.

For index 4, we check whether there is a gym, no there is not and the closest gym is 2 blocks away.

For index 1, there is a gym to the right of it which is 1 block away.

But at index 1 there is already a value less than 1, we have to take the minimum value.

At index 0, we update the minimum distance from the gym to be equal to 1.

We do this in O(B) time. We do ‘r’ computations in O(B) time. So, that is O(BR) time.

Now the total time complexity is O(BR) time.

Space complexity will be a bit worse.

**Space**: O(B+BR) = O(BR)

We are storing ‘r’ arrays and each array has ‘B’ length.

We got rid of the squared term.

**IMPORTANT POINT TO NOTE**:

The buildings given as part of the required buildings array have to be present at one of the blocks.

So we don’t need to check whether a building won’t be present in any of the blocks or not.

**Calendar Matching**:

Imagine that you want to schedule a meeting of a certain duration with a co-worker. You have access to your calendar and your co-worker’s calendar (both of which contain your respective meetings for the day, in form of

[startTime, endTime], as well as both of your daily bounds (i.e., the earliest and latest times at which you are available for meetings every day, in the form of [earliestTime, latestTime].

Write a function that takes in your calendar, your daily bounds, your co-worker’s calendar, you co-worker’s daily bounds, and the duration of the meeting that you want to schedule, and returns a list of all the time blocks (in the form of [startTime, endTime]) during which you could schedule the meeting, ordered from earliest time block to latest.

Note that times will be given and should be returned in military time. For example: 8:30, 9:01, 23:56.

It has got a real world application.

Coding part is difficult in comparison to algorithm part.

Algorithm part is logical.

We are given the duration of the meeting and 2 calendars, our and our co-worker’s.

We are also given our and co-worker’s daily bounds, that is in-time and out-time.

What does it mean for a time range to be valid?

We both don’t have to be busy. There should not be any other meeting in that range and the range should fall in the in-time, out-time range.

Find the availabilities in between the given blocks of time.

Common availabilities.

We are given the calendar of meetings, that is the times we are unavailable.

First we could convert the given 4 inputs, the 2 calendars and 2 daily bounds into one calendar of all of the **unavailabilities** where atleast one of the 2 co-workers in unavailable.

First thing we should do is update both the calendars to take into account the daily bounds.

We can create 2 additional meetings for each person from 2 bounds.

0:00-9:00 (will go to the far left of the 1st calendar),

20:00-23:59 (will go to the far right of the 1st calendar)

Updated calendars that tell about when both the individuals are busy.

Now, we merge these calendars.

We longer want to deal with 2 separate calendars.

Lets just deal with 1 merged calendar that basically tells us what the unavailabilities of these 2 individuals are.

If we are trying to find a meeting that matches for both individuals, then we need the availabilities or the blocks of time where not just 1, but both the individuals are available.

Opposite of this is a calendar with all the unavailabilities where atleast one person is unavailable. It’s basically merging the 2 calendars.

The way we are going to merge these 2 calendars is we are going to merge them in a sort of merge-sort fashion.

We want the calendar to be ordered from beginning of the day all the way to the end of the day because that is going to make things a lot easier for us when we finally have to find the availabilities.

The question gives us the calendars in a sorted order. Good question to ask to the interviewer.

If we want to merge both of them in a sorted order, then we can do so in a merge-sort fashion where we iterate through both calendars looking at the first value in the calendar at any given time and seeing which one is smaller than the other.

We will compare the starting time of a given meeting.

We will put pointers at the beginning of both calendars.

Pick 2 times and compare which range has smaller starting time.

If they both start at the time, we can pick either.

Move the pointers.

We merged the 2 calendars in ascending order based on the start time of every meeting.

These represent all of the times where atleast one of the co-worker is unavailable or busy.

Now we can look in-between blocks of time. These in-between times will be where both the co-workers will be available.

If one block of time is bigger or atleast equal to the meeting duration then it would be one of the valid blocks of time.

Our merged calendar is crazy looking in terms of end time.

We can flatten or merge some of the times into one.

Flattening the array or ranges into fewer ranges that can capture multiple ranges at once.

Flatten these so that they no longer overlap.

Check the gaps.

**Time**: O(c1+c2), merge step

**Space**: O(c1+c2), will be creating a merged calendar.

Will convert String to number.

Total **5 steps**.