Problem Statement:

You're looking to move into a new apartment on specific street, and you're given a list of contiguous blocks on that street where each block contains an apartment that you could move into.

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 pick an apartment block such that you minimize the farthest distance you'd have to walk from your apartment to reach any of your required buildings.

Write a function that takes in a list of contiguous blocks on a specific street and a list of your required buildings and that returns the location (the index) of the block that's most optimal for you.

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

Input :

$R = \{r_0, r_1, r_2, ..., r_k\}$ A finite set of requirements.
$B = (b_0 = (r_0, t_0), b_1 = (r_1, t_1), b_2 = (r_2, t_2), ... b_n = (r_n, t_n))$ A vector of contiguous blocks (r_i, t_i) where $r_i \in R$ and t_i is a boolean indicating wheather the requirement is satisfied by the block. [0 <= i <= n]

Output:

Apartment block $b_i \in B$ which is at the optimal position.

Naive Solution:

A naive way to solve this would be to compute the shortest distance to each required building for every apartment block,
and choose the best option.

```
Pseudocode:
    minTotalDistance = ∞
    optimalApartment = NULL
    for each b_i in B:                                          //
O(n^2k)
        totalDistance = 0
        for each r_i in R:                                      // O(kn)
```

```
            totalDistance += minimum_distance_between r_i and b_i    // O(n)
        if totalDistance < minTotalDistance:
            minTotalDistance = totalDistance
            optimalApartment = b_i
    return optimalApartment
```

Finding the minimum distance between ri and bi will require O(n) time in the worst case [|B| = n].
We have to do this for every reuirement in R so the total run time is O(kn) [|R| = k].
We repeat this computation for every b_i in B. Therefore, the overall time complexity is $O(n^2k)$

Time Complexty = $O(n^2k)$

Space Complecity = O(1)

Optimal Solution:

Intuition: Initially we will start with a region (or window) of size 0. We will keep expanding this region until all the required buildings exist in this region.
After we have found such a region, the next step is to keep shrinking the region as long as the requirements are satisfied.
The block at the center of this minimal region is at the optimal position. This block may be the most optimal solution.
We repeat this process of expanding and shrinking until the entire list has been explored. We choose the smallest region satisfying the requirements and return the block at the center of this region.

A region is simply a subarray of B. This problem boils down to finding the minimal subarray which satisfies all the requirements.
An easy way to check if a region satisfies all the requirements is to keep track of the number of required buildings that exists in the current region. We update this variable as we expand or shrink the region.

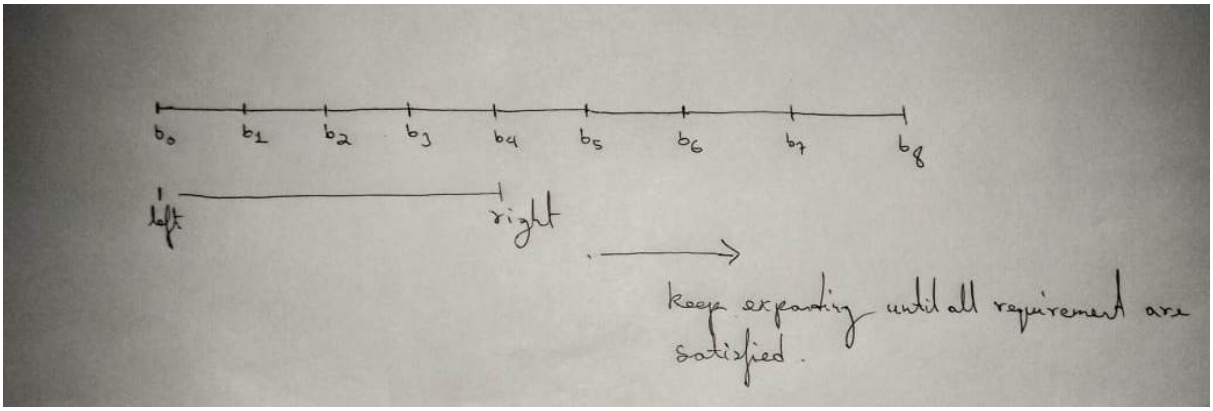
```
Pseudocode:
solution_exists = False            // It is possible that there exists no
region which satisfies all the requirements.
left = 0                           // Beginning of the region.
no_of_requirements_satisfied = 0   // Keeps track of the number of
requirements satisfied by the current region.
Map = {}                           // Initialize a map to keep track of the
frequencies of the required buildings.
optimal_length = ∞                 // Keeps track of the length of the
smallest region seen so far.
optimal_block = NULL


// Keep expanding the window until all the requirements are satisfied.
for right = 0 upto B.length:                  // O(n)
```

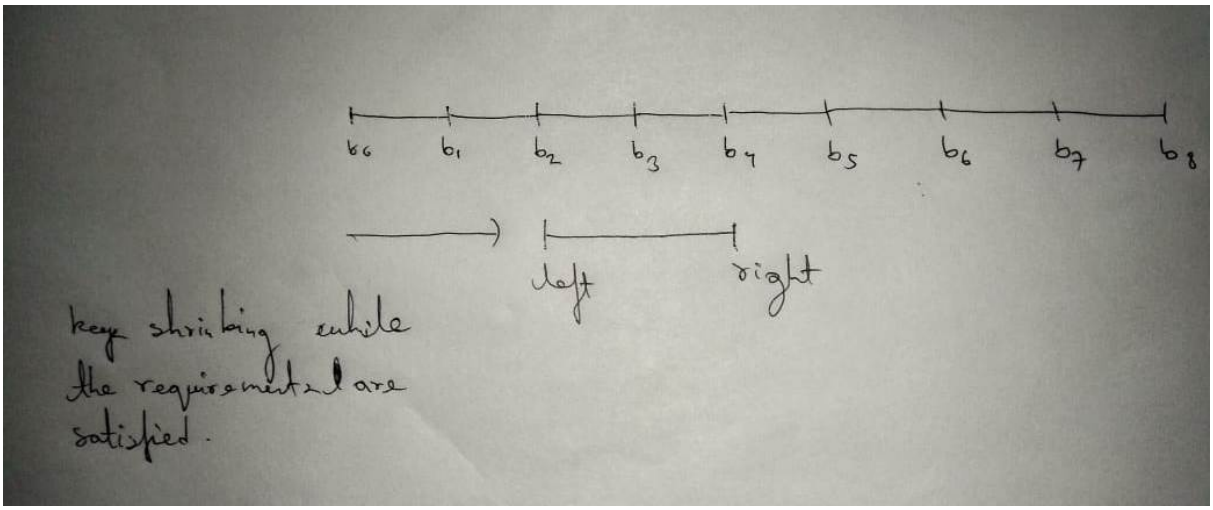
```
// Update the frequencies
for r_i in R:                           // O(k)

    if r_i exists in B[right]:
    // If frequency of r_i is 0, then a new requirement has been satisfied.
        if Map[r_i] == 0:
            no_of_requirements_satisfied += 1
        Map[r_i] += 1

    length = ∞                              // Size of the current region.

    // Keep shrinking the region while all the requirements are satisfied.
    while no_of_requirements_satisfied == R.length:
        solution_exists = True              // Now we can conclusively say
that a region satisfying all the requirements exists.
        length = right - left               // Update the length of the
current region.

        for r_i in R:             // O(k)
            if r_i exists in B[left]:
                Map[r_i] -= 1
                // If frequency of r_i is 0, then a requirement which was
previously satisfied is not longer valid.
                if Map[r_i] == 0:
                    no_of_requirements_satisfied -= 1
        left += 1                            // Shrink the window.

    // If we find a smaller region.
    if length < optimal_length:
        optimal_block = (left + right) / 2   // Optimal block will be at
the center of the block.
        optimal_length = length

if solution_exists == False:
    return NULL
return optimal_block
</code>
```

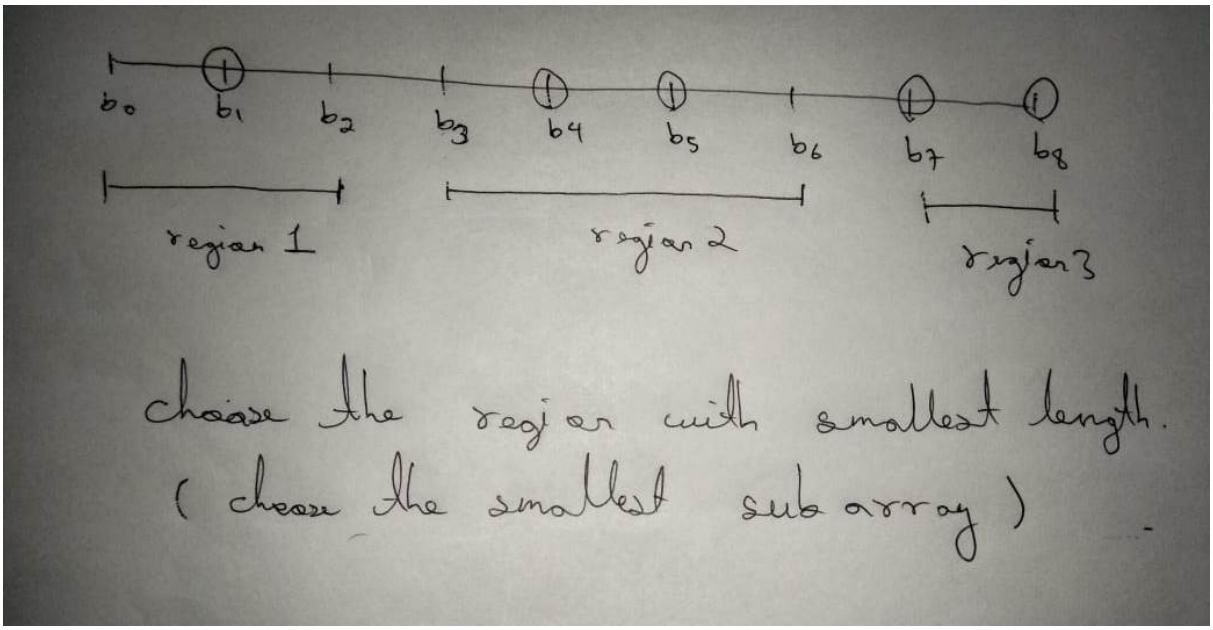
Time Complexity = O(nk)

Space Complexity = O(k)

Diagram illustrating the algorithm:







Python solution:

In [ ]:

```
from math import inf
from collections import defaultdict

def apartmentHunting(blocks, reqs):
    solution_exists = False

    # Beginning of the window.
```

```
    left = 0

    # Map to keep track of frequency of the required buildings seen so far.
    mp = defaultdict(int)

    # Keep track of the number of requirements satisfied so far.
    no_of_requirements_satisfied = 0

    optimal_apartment, optimal_distance = 0, inf

    # Keep expanding the window while the given requirements are not satisfied
    for right, block in enumerate(blocks):

        # Update the frequency.
        for req in reqs:
            # If the required building is present in the block update its frequency.
            if block[req]:
                if mp[req] == 0: no_of_requirements_satisfied += 1
                mp[req] += 1

        distance = inf

        # Keep shrinking the window until the given requirements are not satisfied.
        while no_of_requirements_satisfied == len(reqs):
            solution_exists = True
            distance = right - left
            for req in reqs:
                if blocks[left][req]:
                    mp[req] -= 1
                    if mp[req] == 0: no_of_requirements_satisfied -= 1
            # Shrink the window
            left += 1

        if distance < optimal_distance:
            optimal_apartment = (left + right) // 2
            optimal_distance = distance


    # If no solution exists return None.
    if not solution_exists:
        return None

    return optimal_apartment
```