**Knuth Morris Pratt Algorithm**:

Write a function that takes in 2 strings and checks if the first string contains the 2nd one using the Knuth Morris Pratt algorithm. The function should return a boolean.

Problem of string matching.

**Naïve approach**:

Compare every character one at a time and whenever we get a failure/mis-match, we restart.

When we find a mismatch, we move forward in the bigger string.

Time complexity for this will be O(nm)

KMP algorithm allows us to solve this problem of string matching in O(n+m) time.

We can turn this multiplication into an addition.

This algorithm takes advantage of the fact that there are often patterns in strings and specially in strings that have repeated letters.

Thinking about strands of DNA, we can imagine that there would be a bunch of repeated letters in string representations of these kinds of things.

If we have got billions of them, then we can do some clever pattern matching to do our string matching.

In naïve solution, when we were doing the naïve check of just checking every letter and we got to a point where we found a mismatch, we go all the back to the next letter of the bigger string.

Keep track of letters that we have already visited that have kind of formed patterns where we can see that they are already repeating. And use that to our advantage to come up with an optimal algorithm.

The first thing in kmp algorithm that we are gonna do is we are going to disregard the bigger string for a second and we are going to traverse the substring we are looking for.

We are going to build a table, a 1d array/list that is going to keep track of patterns that we identify in the substring.

We will declare two indices that we are going to use to traverse this substring. And these 2 indices are going to be i and j.

Initialize i=1 and j=0

The 1d array will be initialized to bunch of -1s.

First things we do is compare the 2 letters in the string we are looking for.

Compare letters at index i and index j

‘a’ and ‘e’ are unequal, increment i.

-1 means that at the give index, we don’t have any patterns.

We have not identified that the letters ‘a’ or ‘e’, any patterns within them appeared before in the substring.

For the first index 0, it is always going to be -1 as the ‘a’ there does not belong anywhere. ‘a’ is the first character in the string, so it can’t have appeared before, there can’t have been a pattern before it.

[ -1, -1, -1 ]

Now i reaches index 3.

Now index 0(j) matches with 3(i).

What that means is in the substring so far, “aefa” we have identified a pattern.

This pattern consists of one letter so far, the letter ‘a’

In some sense we can say that we have got a prefix in this substring that is also a suffix.

The letter ‘a’ is not only a prefix, but also a suffix.

When we are at index i=3, we can say where is the last time that this sort of pattern appeared ?

It is at index, which is 0.

Store 0 at index 3 of the 1d array.

Now increment both i and j.

Now ‘e’ is equal to ‘e’. There is a pattern in this substring that we have recognized.

And it is saying that at index i, where the letter ‘e’ is, the pattern matches whatever pattern we have ending at index j. Index j is 1.

Whenever we get a match, in the 1d array we store index j at index i of the array.

It is saying that there is a prefix here, that ends at index j that is also a suffix in this substring.

Again, we increment both i and j as they were equal to each other.

Now j is 2 and i is 5 and the 2 letters at these indices are not equal to each other.

Now j is greater than 0, what we do now is check the letter right before where index j is, which is letter ‘e’.

Let’s see if this ‘e’ had a pattern that it had stored that we could use to compare against.

So, we look in our table at index j-1, 1 and it has -1 in the 2d array.

So, we just reset the j=0

What this means is, the letter ‘d’ at index 5 or the substring “aefaed”, there is no pattern that we have identified in this substring that ends with ‘d’ and that has repeated characters at the beginning.

There is no prefix in the substring, “aefaed” that is also a suffix.

In the 1d array, we put -1 at index 5.

Increment i.

Letters at index 0 and index 6 are equal. For the substring “aefaeda” there is a prefix that is also a suffix, which happens to be just the letter ‘a’

The prefix ending at index 0 is also a suffix and it is pattern that we have identified.

Now we increment both j and i.

Now j is 5 and i is 11.

‘d’ is not equal to ‘f’ but there are clear patterns in the substring “aefaedaefaef”

There are clear patterns in this substring.

Look at the letter immediately before letter at index j and see if we have got a pattern here.

We failed the match of index 5 and 11.

What if we just went to the last previous matching pattern of “ae” and look at the letter that comes right after it, letter ‘f’ and check it against our current letter at index i.

Instead of bringing j all the way back to the beginning, we can say, right before the ‘d’ at index j where were we at.

We were at the ‘e’ at index 4.

We know that this ‘e’ matches the pattern at index 1.

So everything that came before index 1 matched a pattern here, leading up to this ‘e’.

We could also compare the letter at index 2, which comes after something that has matching “ae” and compare it with the letter at index 11.

The variable j is going to be equal to whatever is stored in our pattern, one index before j, index 4, letter ‘e’

Lets see what pattern this matches.

‘e’ matches the patter at index 1.

And then we say, let’s go to the letter immediately to the right.

Now j=2.

Compare letter at index 2 with letter at index 11. And here we get a match. And we can mark index 2 here under the i in the 1d array.

Which means that in the substring “aefaedaefaef”, we have got a prefix ending at index 2 which is also a suffix.

Increment j and i. And we get a match.

What do the numbers in the 1d array represent ?

They represent the indices at which we found the matching patterns. And we store those indices such that we don’t have to go back all the way to the beginning of our substring.

Whenever there is a failure in a match, we can just go back to the last sort of matching pattern.

**Matching against the real string**:

We are going to employ a very similar technique that we used when we were doing the pattern building.

We are going to put index i in our actual string and we are going to put the index j in our substring.

Start comparing the letter in the big string, at index i to the letter in the substring, at index j.

We check if they are equal to each other.

As long as j is not at the very end of our substring, clearly we have not found our entire substring.

If j reaches the end of the substring, then that means that we found the substring in the main string.

We reach a point when i=5 and j=5 and we get a mismatch.

In the main string, there is a pattern “aef” that also presents itself in the substring.

How can we sort of programmatically make use of it ?

The letters at the two indices i and j don’t match. But j is greater than 0 and letter at index j is not the first letter in the substring.

So we might have a pattern here that we can exploit.

Go to the previous letter, right before the index j, letter ‘e’

We then go to index 1. And then go the index right of it.

“ae” matches “ae” and we have failed because of letter ‘d’.

Now we check whether the letter ‘f’ matches against the letter at index i.

We go to the left because the pattern fails at this index j.

If we have a pattern at the left of j, we jump to its prefix.

This allowed us to not to have to move the index i at all and only have to move the j index a little bit and to keep finding a match.

And now we can keep going on with our match because we exploited this pattern.

If we again get the same situation of a mismatch, then we do the same thing again.

What did we accomplish?

Our variable i did not have to move back to the beginning to the 2nd, 3rd or 4th character at all. And j did not have to move back much because of the pattern.

Before the failure everything was good, so lets go to the characer right before it.

While bringing j backwards and checking j+1, if there is a mismatch of new j and i, then we go back again, decrement the j.

And if there is no pattern at previous j, then we bring j all the way back to 0.

Some implementations fill the 1d array with 0s rather than -1s.

O(m) space where m is the length of the input substring.

O(n+m) time, n is the length of the main string.

First part of algorithm where we build the 1d array is going to take O(m) time.

The second part of the algorithm, the actual searching in the main string is going to take O(n) time.

When we build the 2d array, we atleast do 1 pass through the substring.

The index i that we used moved forward and index j was moving forward and backward according to the logic.

Anytime, the index j moves forward, i also moves forward.

Everytime j moves forward is when letters at indices i and j match.

When j moves back, then i does not necessarily move back.

If j were to move back everytime, it will eventually not be able to move back or have to move forward again, which would mean that we would have just double the weight for i to move forward.

For main string match, the index i never moves back.

Only time it didn’t move was when there was a mismatch.

We can only move j back so much before it has to start moving again with i.

If we make 5 steps forward with i and j, and 5 steps backward with j, we cannot make any more steps back.

We can make atmost 5 steps back.

**Important Note**:

The values in the pattern[ ] array represent the longest ending pattern that the character at a particular index is part of.

Subproblems.

When we fill value for a particular index, the substring till that index is our subproblem and we are only interested in that substring.

We are solving the smaller subproblems first in order to build the current subproblem.

**Rectangle Mania**:

Write a function that takes in a list of Cartesian coordinates (i.e., (x, y) coordinates) and returns the number of rectangles formed by these coordinates.

A rectangle must have its four corners amongst the coordinates in order to be counted, and we care about rectangles with sides parallel to the x and y axes (i.e., with rectangles horizontal and vertical sides).

We can also assume that no coordinate will be farther than 100 units from the origin.

It is the kind of question where the more inputs we have, the more impossible this question becomes to solve.

If we were to add 10 coordinates in the plane, it would be difficult for a human being to come up with an answer at a glance.

The key to solve this question is to realize that in order to have a rectangle, we need to have 4 corners.

And if we use that as our sort of guiding principle, things starts to become clearer and easier.

Naïve approach:

Explore all cases/possibilities of 4 corners.

The 4 corners should satisfy the properties of a rectangle.

The two corners on the left hand side of the rectangle need to be directly on the same line. In other words, they need to have the same x-axis or X coordinate rather.

But they need to have a different Y coordinate.

The 2 right hand corners also need to be on the same X coordinate.

The two top and bottom coordinates need to be on the same y coordinate.

If we find ourself at any given corner, we know which points near us could be another corner of a rectangle that we might be in by just checking if these other points are on the same vertical or horizontal line as us.

If we are on a coordinate, we check whether we have a point or multiple points directly above us on the same X coordinate as ours.

We go in a clockwise manner and follow same pattern for every coordinate.

Let’s treat every single pair of coordinates in our input list as the potential lower left corner of a rectangle.

If a coordinate is a valid bottom left coordinate of a rectangle then we should be able to go directly up, find a top left corner, then for every potential top left corner, go to the right, find a top right corner for every top right corner, go to the bottom and so on and so forth.

Let’s grab all of our coordinates here and for every coordinate, let’s store the other coordinates or the other that are directly above us, directly to the right, directly below us and directly to the left.

And this is going to help us to easily do the clockwise traversal.

For (0, 0) point, the origin, we know that directly above us, we have got a (0, 1).

And directly to the right of us, there are (1, 0) and

(2, 0).

There is nothing below and to the left of (0, 0).

The first thing we are gonna do is we are going to iterate through all the points.

And for every point, we are then gonna iterate through all the other points and bucket them as directly up, directly to the right, directly down or directly to the left of us.

We will have a hash table of every coordinate or every point mapping to all of the points, in all 4 directions of that point.

And this is going to take O(n^2) time where n is the length of the coordinates because we are going to iterate through all the coordinates once and for each coordinate, we are gonna iterate through all the other coordinates and bucket them in all these directions.

This will take O(n^2) space.

We can imagine that maybe every point is gonna be on the exact same line.

If we have this situation, then for the lowermost point, we would have to store every other point above it, meaning every other point in the list.

For bottom most point: n-1

n-2, n-3, …. for points above

(n-1) + (n-2) + (n-3) + …..

It is the equivalent of 2 nested for-loops where one index is sort of in front of the other.

This leads to O(n^2) space.

We have the hash table where every coordinate maps to all of the coordinates directly above, directly to the right, directly down, directly to the left of them.

Then we just do this clockwise pattern, where we re-iterate through all the coordinates, but for each of them, treat it as a potential bottom left corner, and then in a sort of depth first search kind of way.

This is why it is sort of resembles a graph.

Backtracking

dfs way

The time complexity to bucket all of the coordinates into these directions of directly above, directly to the right, etc, that was gonna be probably two for-loops.

That is gonna be O(n^2) time.

For the clockwise pattern,

Some of the coordinate are gonna have points directly above, some of them are gonna have points directly to the right, etc.

How do we calculate this ?

In the worst case, we will have all the points on one line. This is also like a double for-loop bounded by O(n^2).

The time complexity for pattern traversal thing is going to be atmost O(n^2).

Recursion call stack space would never exceed O(n^2).

Recursive calls would never exceed O(n).

We will atmost have 4 recursive calls as we will be doing a clockwise traveral and then backtracking.

We will have atmost 4 recursive calls on the function call stack at any given time.

Can we do better ? Yes

We can optimize on the space complexity.

Do we actually need to store every coordinate for every other coordinate directly above, directly below and so on.

What constitutes a point directly above or directly to the right or directly below or to the left ?

All we have to do is compare coordinates.

If we are at (0, 0) and we want to check all the points that are directly above us, we said, we are going to store all of these points in a hash table and we will know them.

We will know that we will be able to access them easily from there.

Another way to do it would just be to say, when we are at this point, iterate through all the other coordinates in general, iterate through all the points.

And if we find one point, that has got a coordinate of the same x-value but of y-value that is higher, then we know that point is above.

We can redo this exact algorithm with the same logic of clockwise pattern and going up, then right, bottom, then left except, instead of storing all the points and positioning them as up, right, down, left for every other coordinate, we can just immediately just start iterating through the coordinates and say,

Whenever I am at a coordinate, if I know that this has a potential bottom left and I want to go directly up, all I am going to do is maybe store all of the coordinates indexed by x coordinate or x-value, and iterate through all the coordinates that have an x equal to my and a y greater than my y.

When I find those points, and maybe now I am at the top left corner, and I am looking for points directly to the right, we are gonna say,

iterate through all the coordinates that have the same y-value as me.

So, maybe we stored again, all the coordinates indexed at their y-value, and only pick the ones that have an x-value that is greater than me.

For this solution, instead of having a hash table of every single coordinate mapping to all of the other coordinates that are directly above, below, etc,

we are just gonna have a hash table that has probably like 2 properties.

One is gonna be called x and one is gonna be called y.

And in these 2 properties, we are just gonna store all the coordinates of a given x value, all the coordinates of a given y value.

That way, we will be able easily access all of the coordinates of a given value. And when we access it, we will just iterate through all those coordinates and say, what is the condition that needs to be met such that I find a point that is directly above, right, below, etc.

We will just do that computation in real time which is a O(1) time operation.

And we will find the correct points.