Anagram

Let's break it down in very simple steps so it's easy to understand.

We want the largest rectangle of 1's in the matrix.
We do this row by row, treating each row like the base of a histogram (bars with different heights).

Step 0: Start

- At the beginning, heights = [0, 0, 0, 0, 0]
  (No bars yet)

Row 1: `1 0 1 0 0`

- If we see `1`, we add 1 to the height.
- If we see `0`, the height becomes 0.

So after Row 1:

```ini
heights = [1, 0, 1, 0, 0]
```

This looks like bars of height:

```yaml
|  |
1  1
```

The largest rectangle area here is just 1 (because only single bars of height 1).

Row 2: `1 0 1 1 1`

- Add 1 to columns with `1`, reset to 0 where `0`.

So after Row 2:

```ini
heights = [2, 0, 2, 1, 1]
```

Now it looks like:

```yaml
|     | | |
2     2 1 1
```

The biggest rectangle we can make:

- Take two bars of height 2 → area = 2
- Take three bars (2,1,1) → area = 3

So largest = 3

Row 3: `1 1 1 1 1`

- Add 1 again where we see `1`.

So after Row 3:

```ini
heights = [3, 1, 3, 2, 2]
```

Looks like:

```yaml
|   | | |
3 1 3 2 2
```

Biggest rectangle:

- First 3 bars `[3,1,3]` → area = 3
- Last 4 bars `[1,3,2,2]` → area = 4
- Middle 3 bars `[3,2,2]` → area = 6 (3 wide × height 2) ✅

So largest = 6

Row 4: `1 0 0 1 0`

- Add 1 where `1`, reset to 0 where `0`.

So after Row 4:

```ini
heights = [4, 0, 0, 3, 0]
```

Looks like:

```yaml
|
4   3
```

Biggest rectangle:

- A single tall bar `4` → area = 4
- A single bar `3` → area = 3

So largest = 4

Final Result

Across all rows:

- Row 1 max = 1
- Row 2 max = 3
- Row 3 max = 6 ✅
- Row 4 max = 4

Maximum rectangle area = 6

Super Simple Summary:

1. Each row builds up a histogram of heights of 1's.
2. For each row's histogram, find the largest rectangle area.
3. Keep track of the biggest one seen so far.

Would you like me to also draw a simple ASCII diagram of the histograms for each row to visualize better?

Let's go step by step on how the area is calculated for a histogram.

A rectangle area in a histogram =

```scss
width (how many bars you include) × height (smallest bar among them)
```

Why smallest?
Because if you take multiple bars, the rectangle can only be as tall as the shortest bar (otherwise it won't fit).

Example 1: First 3 bars → `[3, 1, 3]`

- You want one rectangle that covers all 3 bars.
- The shortest bar is `1` (middle one).
- Width = 3 bars.
- Height = 1.

✅ Area = width × height = 3 × 1 = 3

Example 2: Last 4 bars → `[1, 3, 2, 2]`

- The shortest bar here is `1` (first one).
- Width = 4 bars.
- Height = 1.

✅ Area = 4 × 1 = 4

(If you don't include the `1`, you can also pick `[3,2,2]`, see next)

Example 3: Middle 3 bars → `[3, 2, 2]`

- The shortest bar here is `2`.
- Width = 3 bars.
- Height = 2.

✅ Area = 3 × 2 = 6

So we look for the widest possible stretch of bars where the smallest bar is still tall enough to make a big rectangle.

That's why `[3,2,2]` gives the largest area = 6.

Want me to draw a tiny diagram showing these bars and the rectangles inside them?

Let's explain this very simply.

We want to find all places in `s = "cbaebabacd"` where the substring is an anagram of `p = "abc"`.

An anagram means:

- Same letters
- Same counts
- Any order

Step 1: Count letters in `p`

`p = "abc"`

So:

```
css

a  1
b  1
c  1
```

We'll compare every 3-letter window in `s` with this count.

Step 2: Look at every window of length 3 in `s`

Because `len(p) = 3`, we move a sliding window of size 3 over `s`.

Window 1: `"cba"` **(index 0-2)**

Count:

```
css

c  1
b  1
a  1
```

✅ Matches `p`'s count → add index 0

Window 2: `"bae"` **(index 1-3)**

Count:

```
css

b  1
a  1
```

```
e  1
```

❌ Not equal (has `e`, missing `c`)

Window 3: `"aeb"` **(index 2-4)**

Count:

```
css

a  1
e  1
b  1
```

❌ Not equal (has `e`, missing `c`)

Window 4: `"eba"` **(index 3-5)**

Count:

```
css

e  1
b  1
a  1
```

❌ Not equal

Window 5: `"bab"` **(index 4-6)**

Count:

```
css

b  2
a  1
```

❌ Not equal (too many `b`, missing `c`)

Window 6: `"aba"` **(index 5-7)**

Count:

```
css
```

```
a  2
b  1
```

❌ Not equal (too many `a`, missing `c`)

Window 7: `"bac"` **(index 6-8)**

Count:

```
css

b  1
a  1
c  1
```

✅ Matches `p` → add index 6

Window 8: `"acd"` **(index 7-9)**

Count:

```
css

a  1
c  1
d  1
```

❌ Not equal (has `d`, missing `b`)

Step 3: Collect results

The windows that matched were at index 0 (`"cba"`) and index 6 (`"bac"`).

✅ Result = [0, 6]

Super Simple View

We just:

1. Take a sliding window of size 3
2. Count letters
3. If the count matches `p` → record the index

Would you like me to show this as a table for better visualization?