Exact pattern matching (continued)

Boyer-Moore-Horspool and Knuth-Morris-Pratt

Exact pattern matching

Given string **x**=*abbacbbbababacabbbba* and pattern **p**=*bbba*
find all occurrences of **p** ind **x**

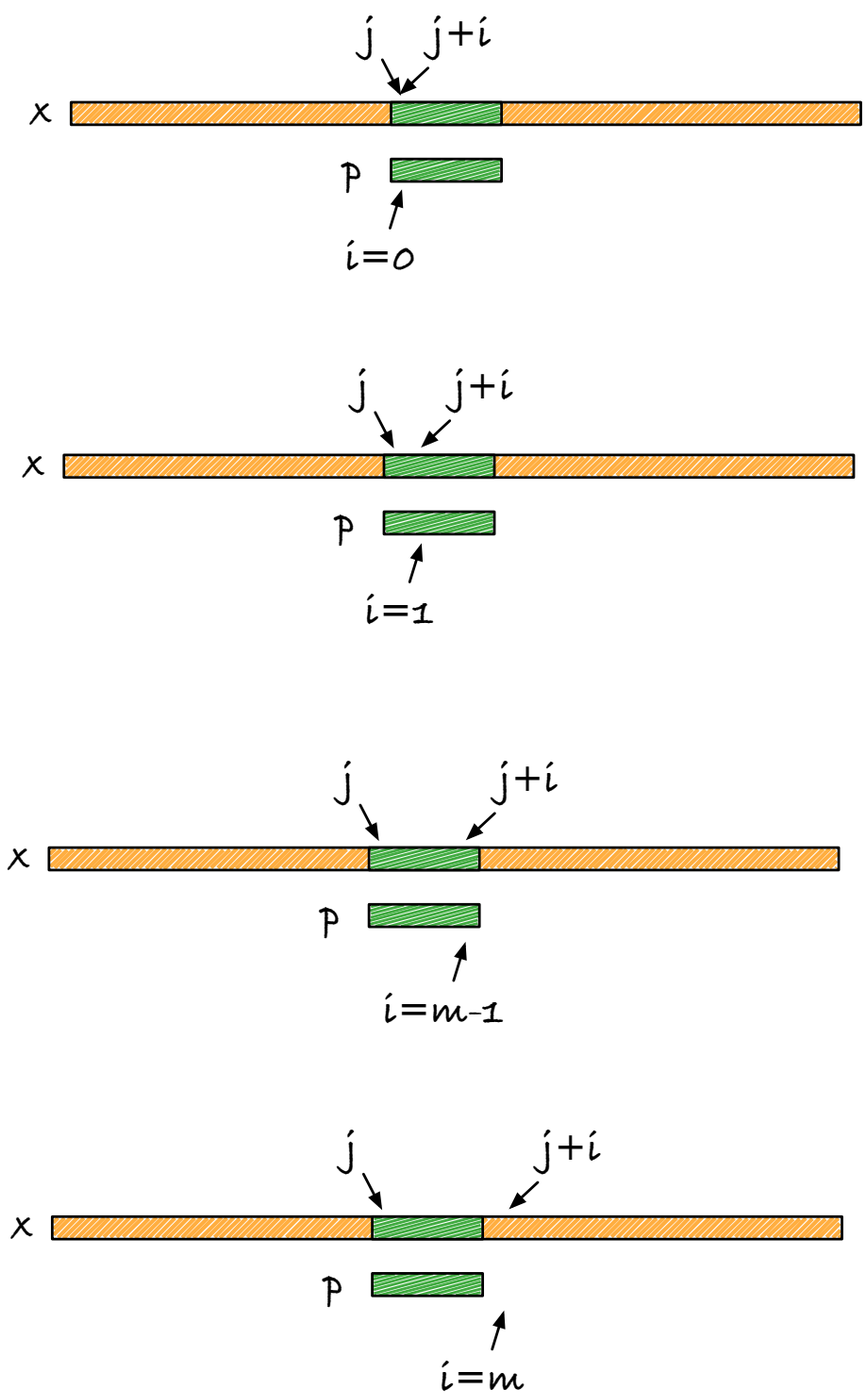
6 17

x=*abbacbbbababacabbbba*

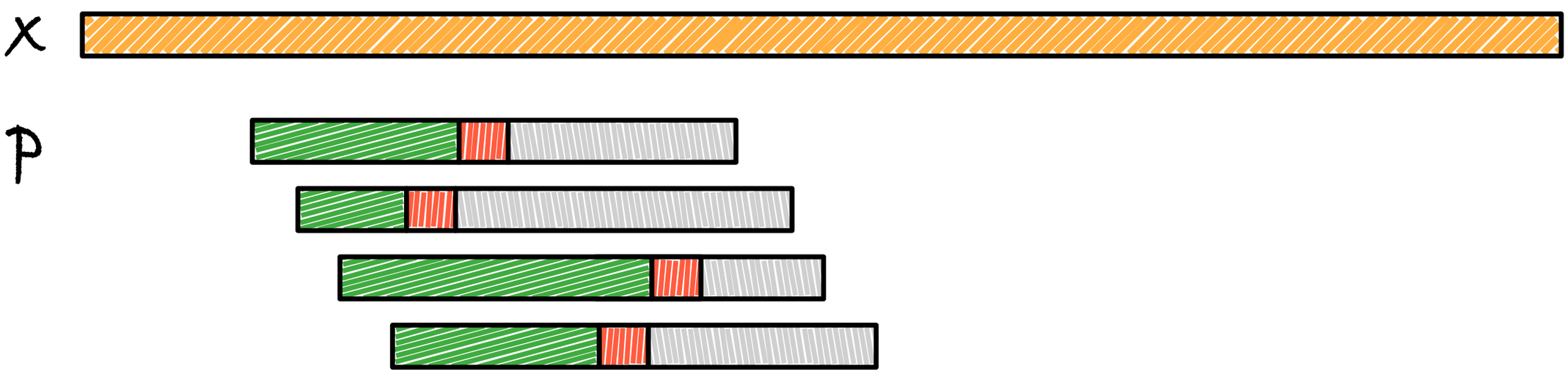
The naïve algorithm

```
for (int j = 0; j ≤ n - m; j++) {
    int i = 0;
    while (i < m && x[j+i] == p[i]) {
        i++;
    }
    if (i == m) {
        report(j);
    }
}
```

Forward search...

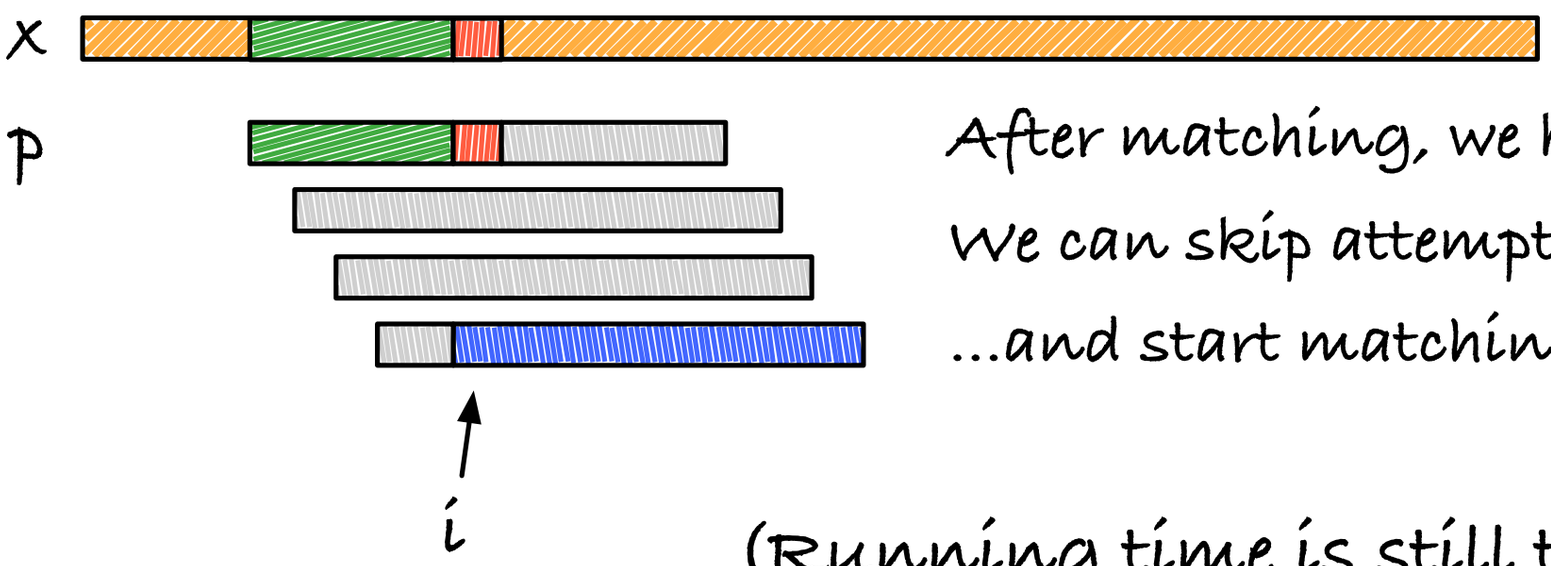


Matching forward



(Running time is total green + red)

Matching forward



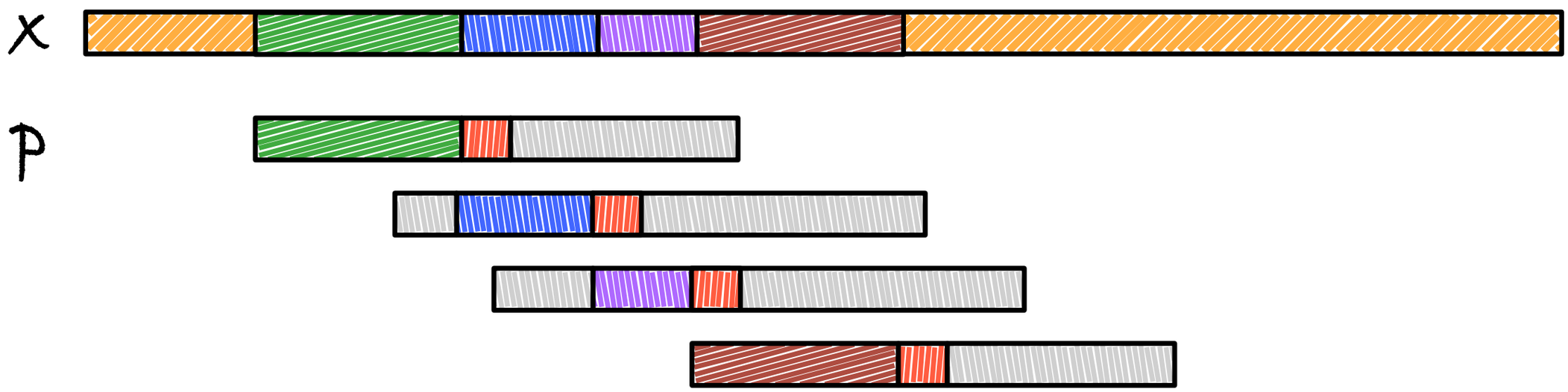
After matching, we know something about x

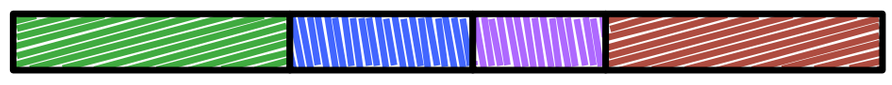
We can skip attempts we know won't match

...and start matching after the green substring

(Running time is still total green + red)

Matching forward

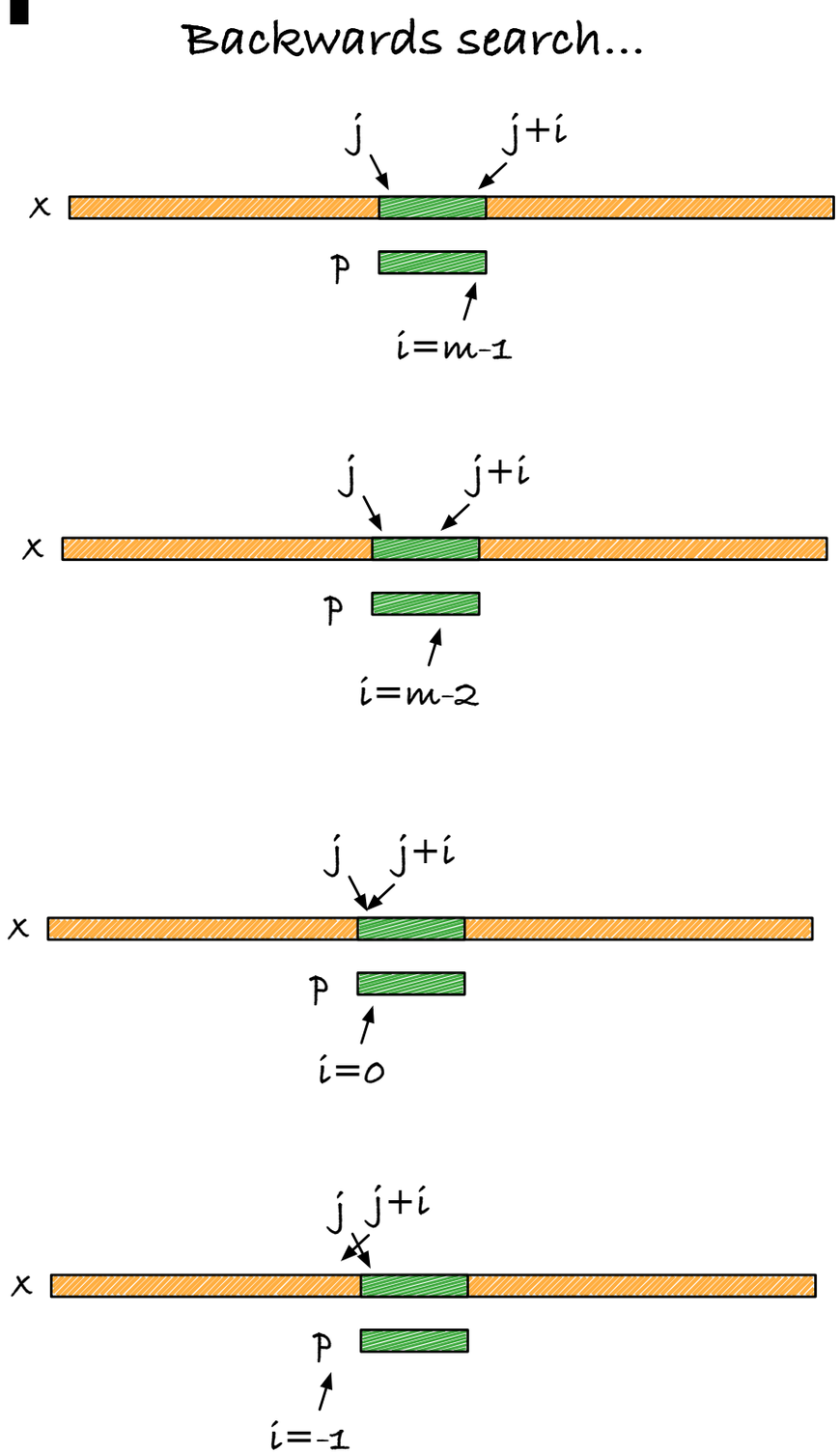


Running time: 

KMP is an algorithm in this category that runs in O(n+m)

Variation on the naïve algorithm

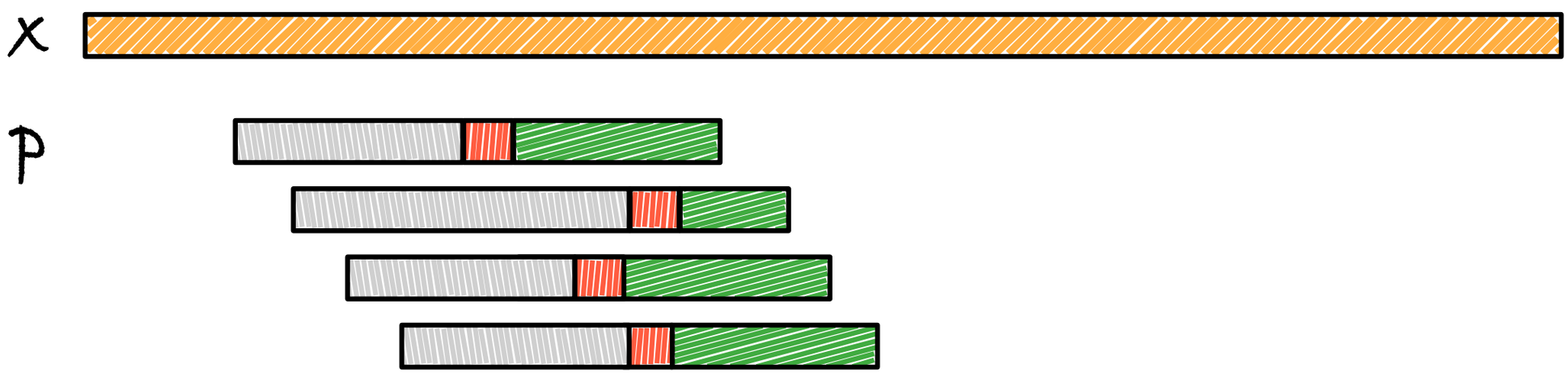
```
for (int j = 0; j < n - m + 1; j++) {
    int i = m - 1;
    while (i ≥ 0 && p[i] == x[j + i]) {
        i--;
    }
    if (i == -1) {
        report(j);
    }
}
```



But why?

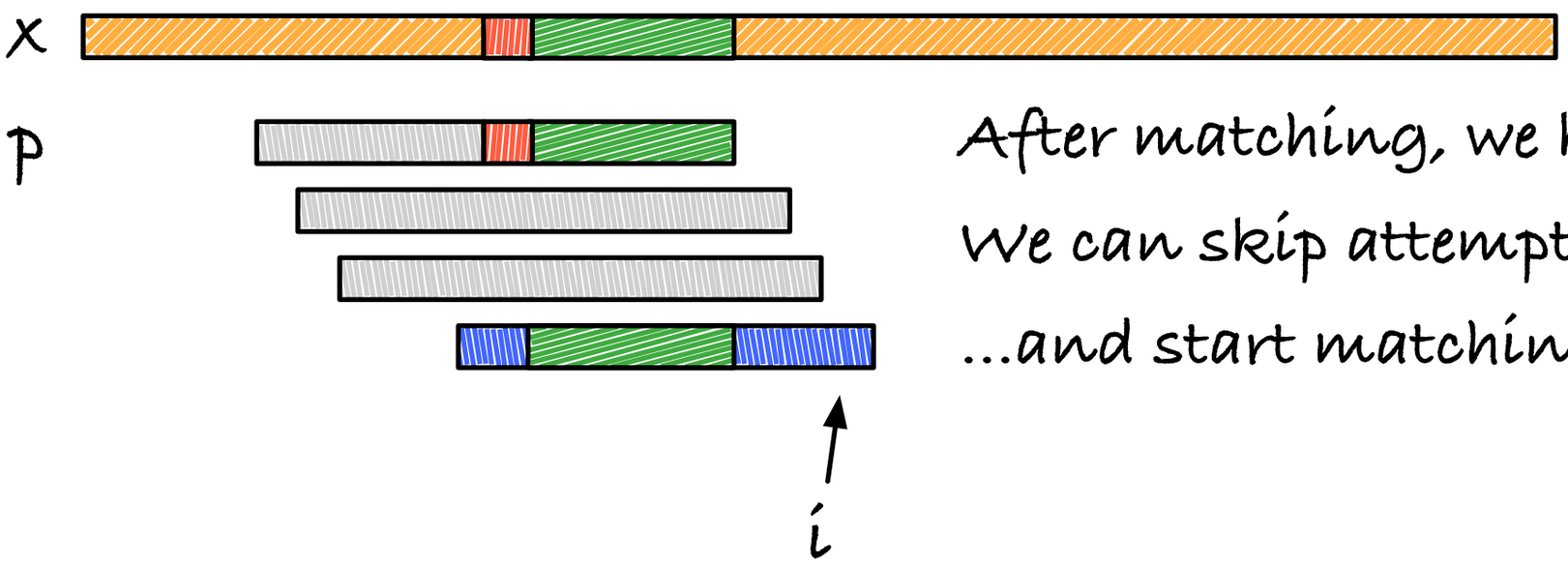
- It has exactly the same complexity matching right-to-left of course
- But by looking ahead, we might be able to avoid some work here and now!

Matching backwards



(Running time is total green + red)

Matching backwards



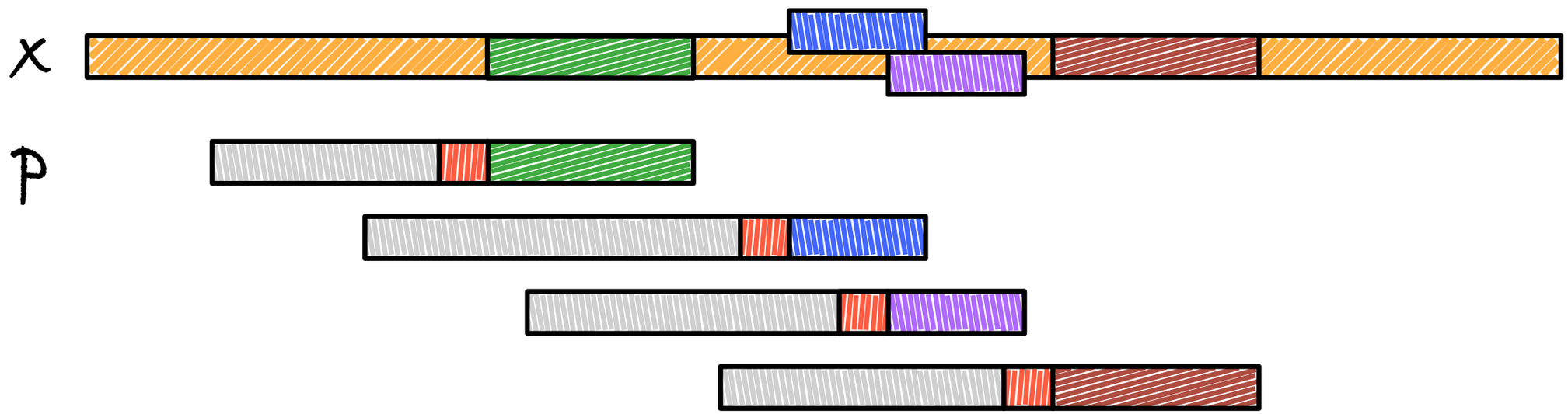
After matching, we know something about x

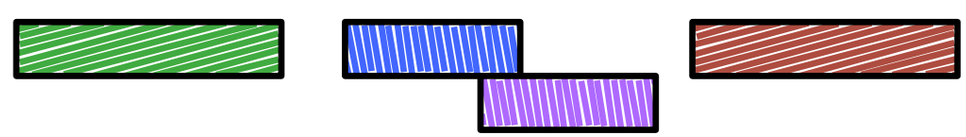
We can skip attempts we know won't match

...and start matching at a new position

(Running time is still total green + red)

Matching backwards



Running time: 

Can be $O(n^2)$ but also $o(n)$ (sub-linear)

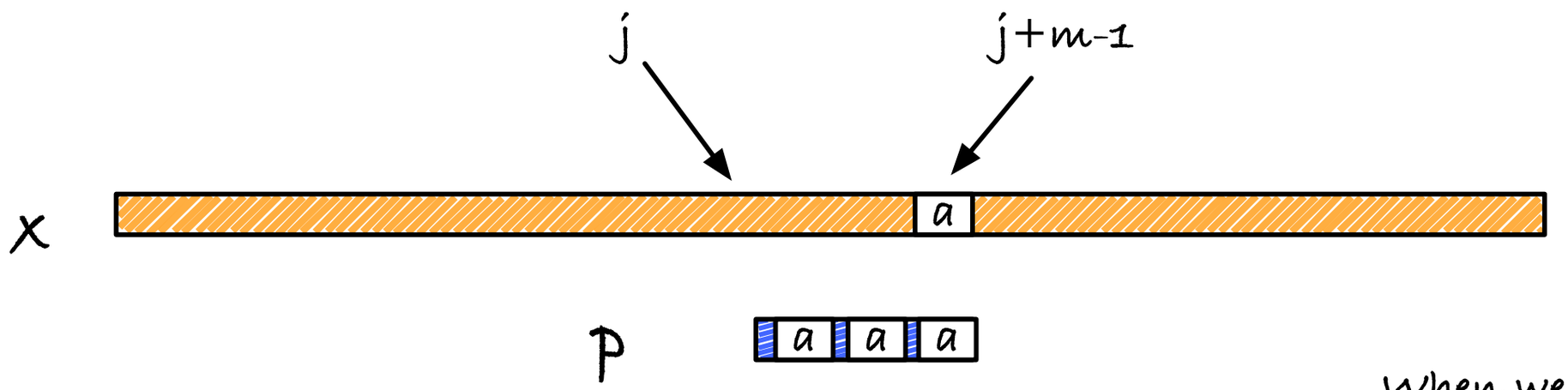
BMH is in this category, with a worst case O(nm) and best case O(n/m) running time.

There are algorithms with worst-case O(n+m) in this category as well, but they are more complicated, and we won't see them in class.

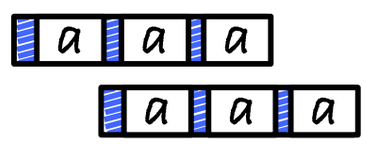
First a backwards,
then a forwards…

Boyer-Moore-Horspool

Observation

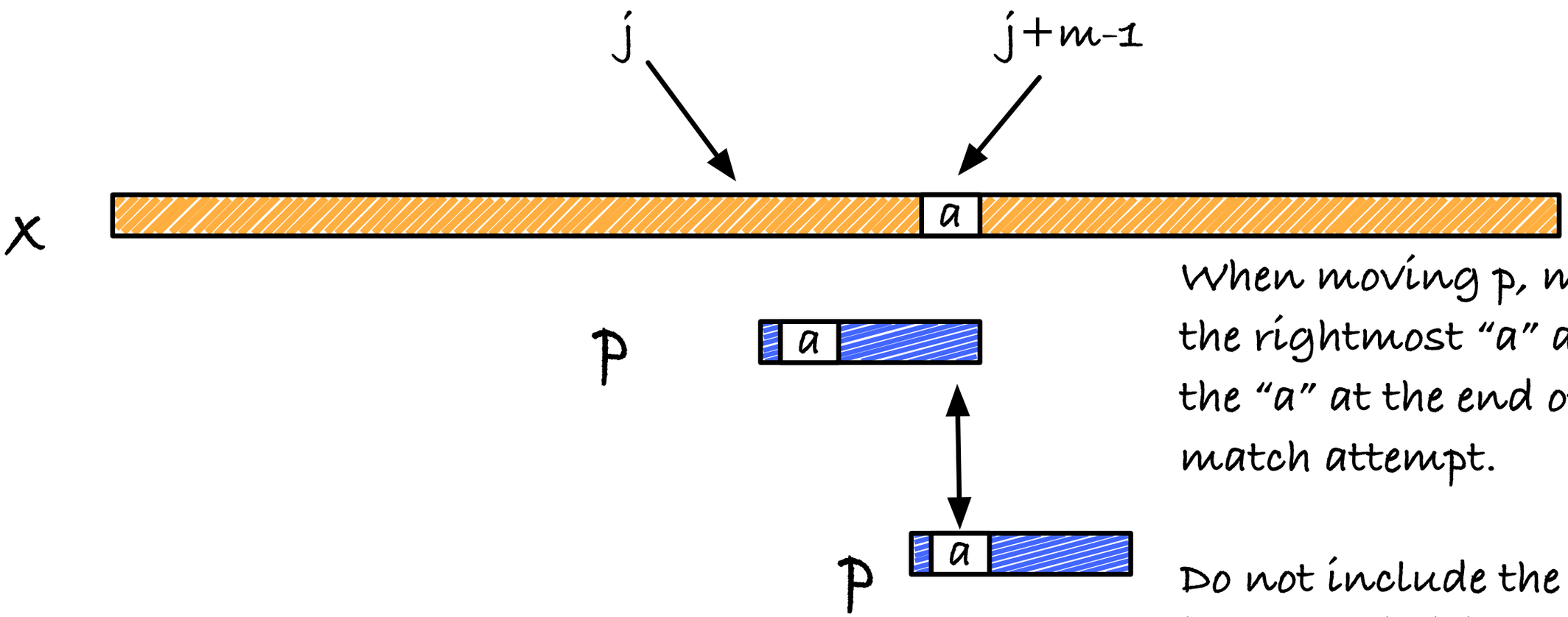


Only potential matches:



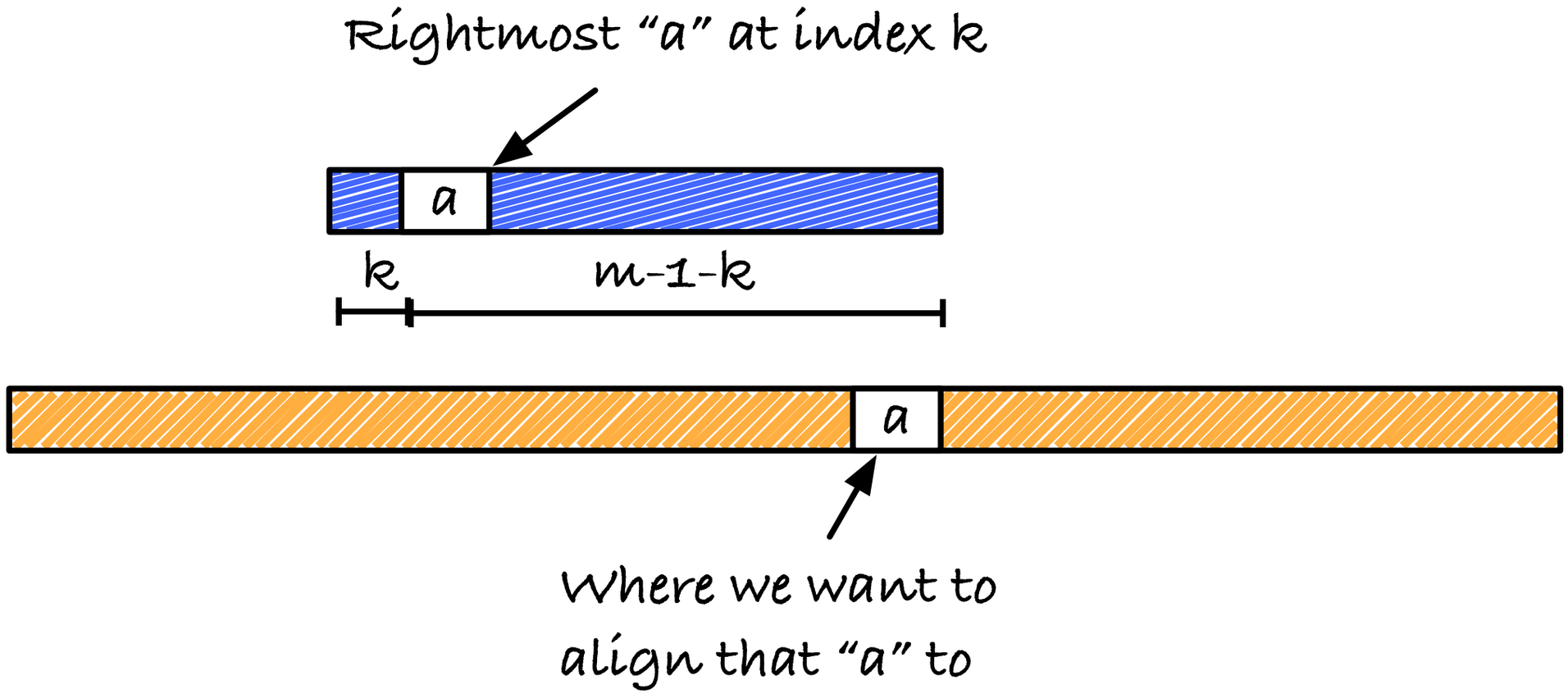
When we move p forward, we can only match if we put an "a" under the "a" in x. All other positions won't give us a match.

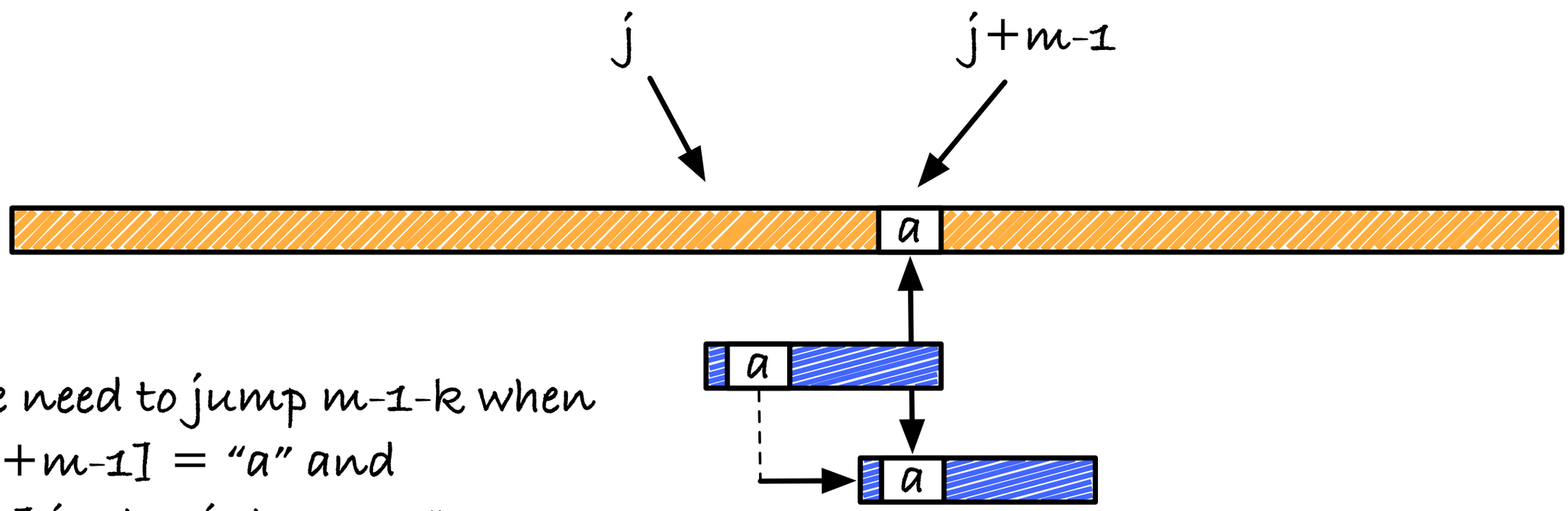
The minimal move matches against the rightmost "a" (that isn't the last character).



When moving p, move p so the rightmost "a" aligns with the "a" at the end of the current match attempt.

Do not include the last character in the definition of "rightmost"!





We need to jump m-1-k when
x[j+m-1] = "a" and
p[k] is the rightmost "a".

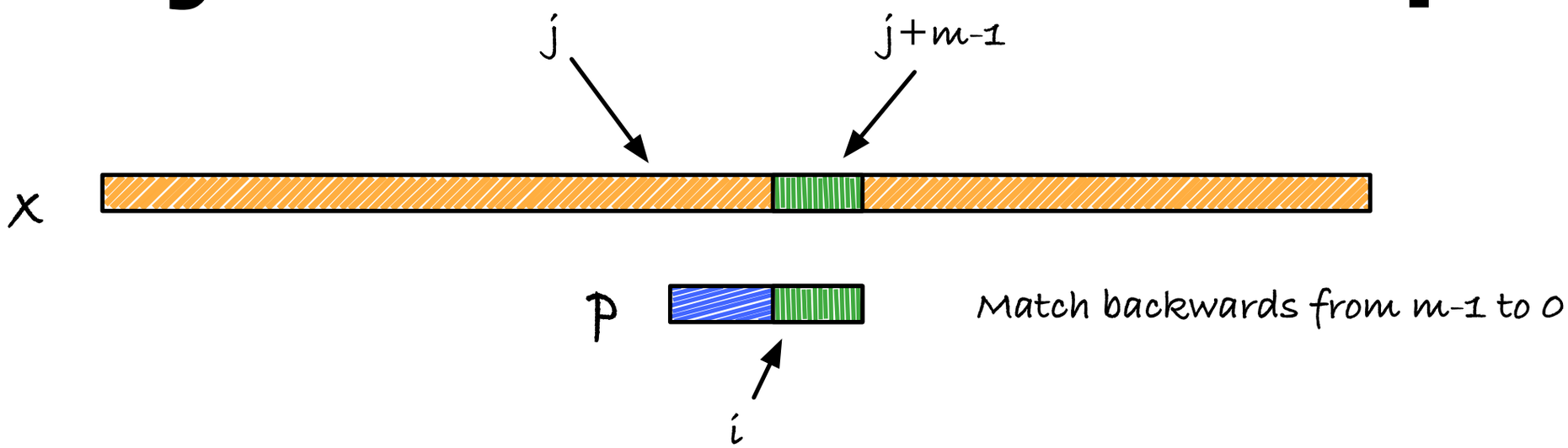
Jump table

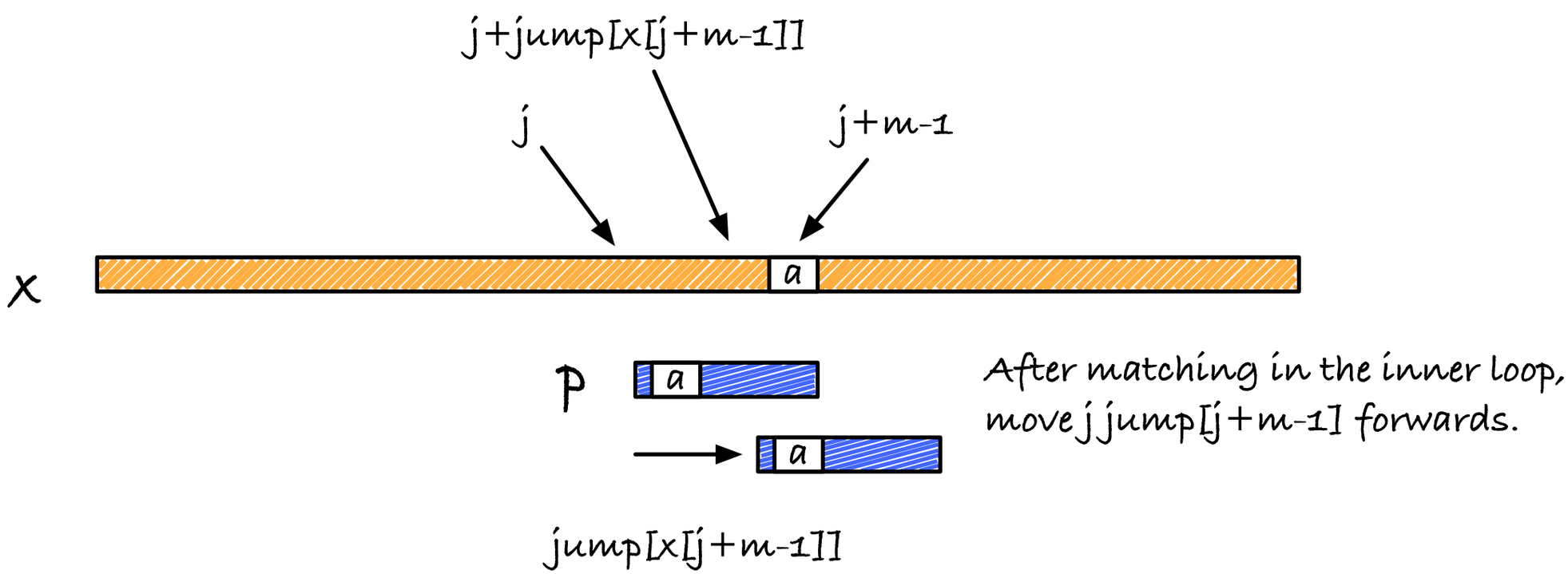
```
int jump_table[ALPHABET_SIZE];
for (int k = 0; k < ALPHABET_SIZE; k++) {
    jump_table[k] = m;
}
for (int k = 0; k < m - 1; k++) {
    jump_table[p[k]] = m - k - 1;
}
```

It is important not to include the last character!

This makes sure the jump table always gives us at least 1 to move j forward with.

Boyer-Moore-Horspool





```
> gsa show exact x p bmh
```

Backwards to BMH

```
for (int j = 0; j < n - m + 1; j++) {
    int i = m - 1;
    while (i ≥ 0 && p[i] == x[j + i]) {
        i--;
    }
    if (i == -1) {
        report(j);
    }
}
```

```
for (int j = 0; j < n - m + 1; j += jump[x[j+m-1]]) {
    int i = m - 1;
    while (i ≥ 0 && p[i] == x[j + i]) {
        i--;
    }
    if (i == -1) {
        report(j);
    }
}
```

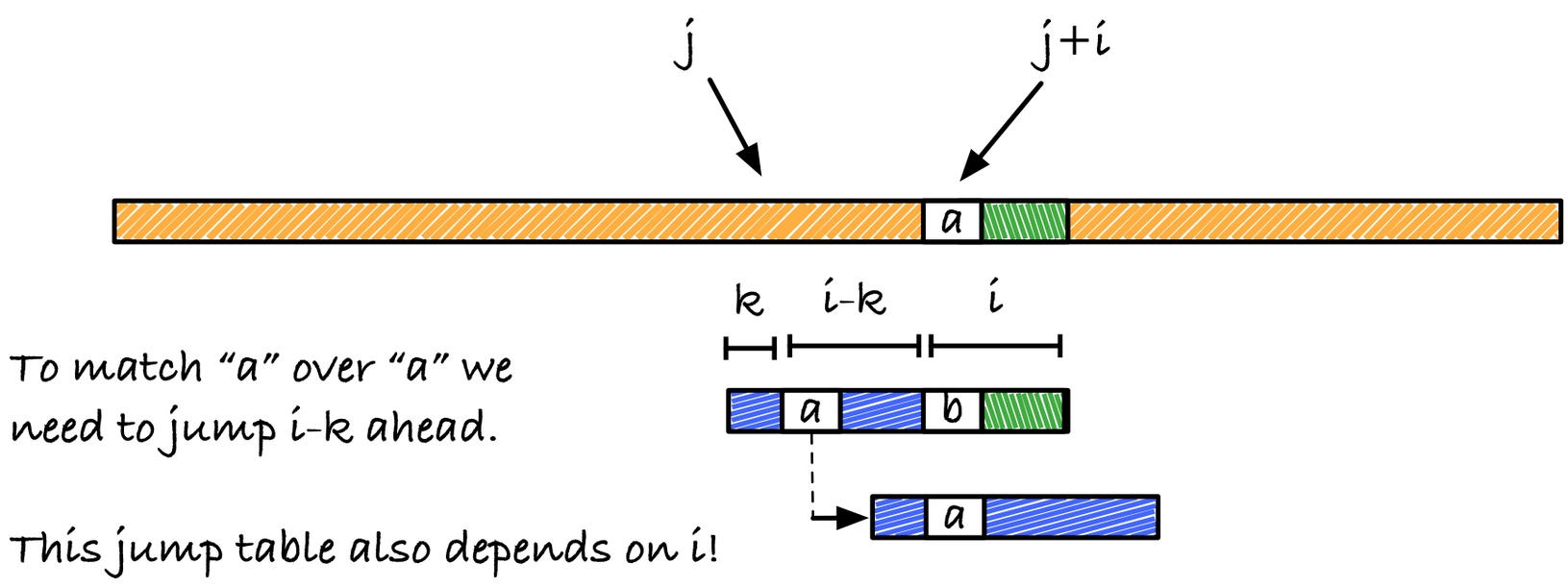
Running time?

- What is the worst case running time?
  - Give an example hitting it
- What is the best case running time?
  - Give an example hitting it

(show code)

Comments

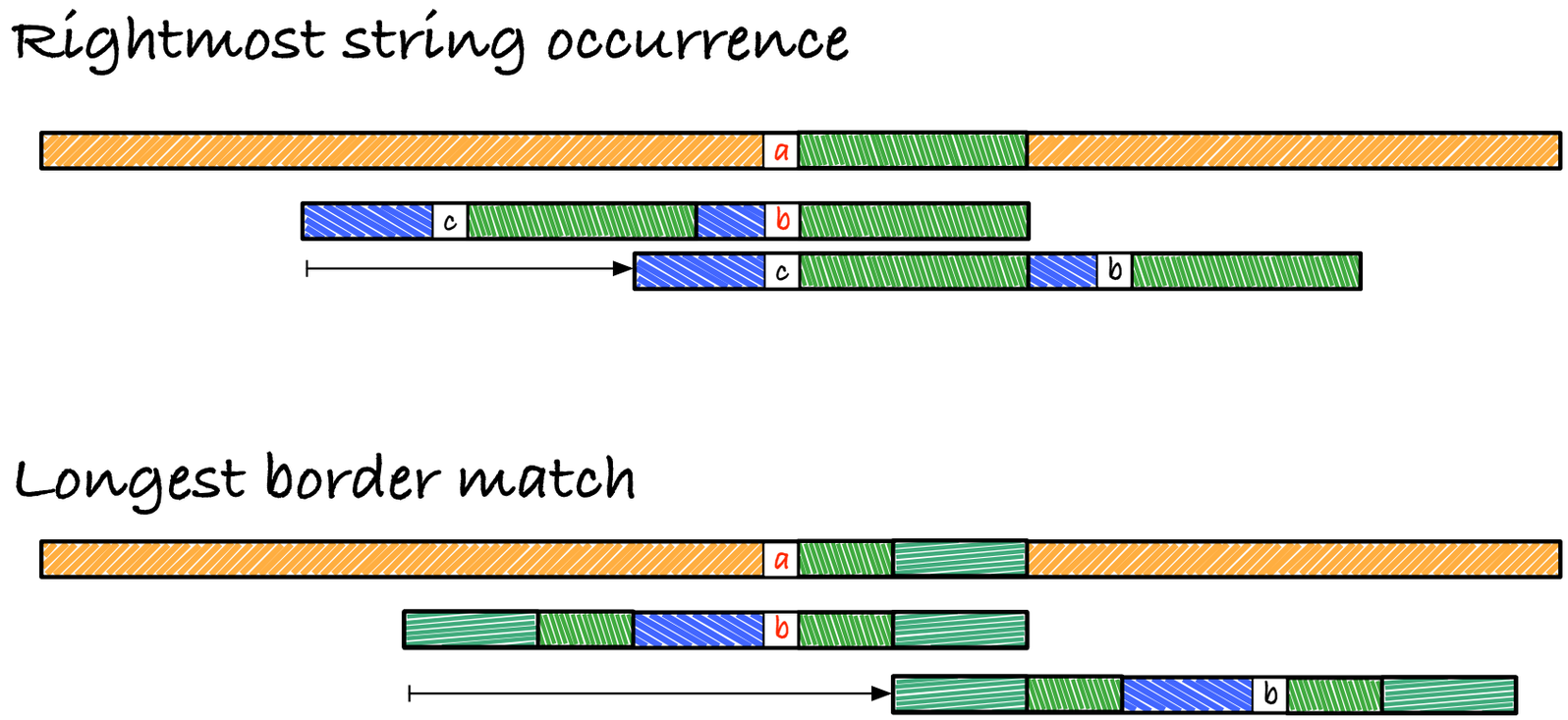
- We can improve on the jump rules, e.g. exploiting that we know the mismatching character



- This is a little more work, to handle a jump table that depends on the index in p.

Comments

- We can also exploit that we know the part we matched in the inner loop

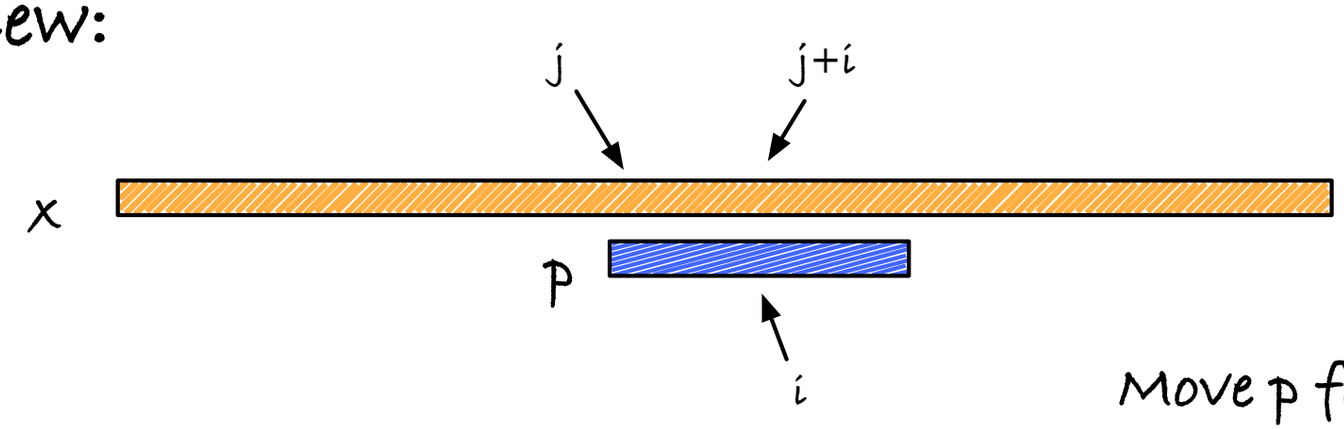


- This is a lot more work, but gives you the Boyer-Moore algorithm. It has the same complexity as the simpler BMH

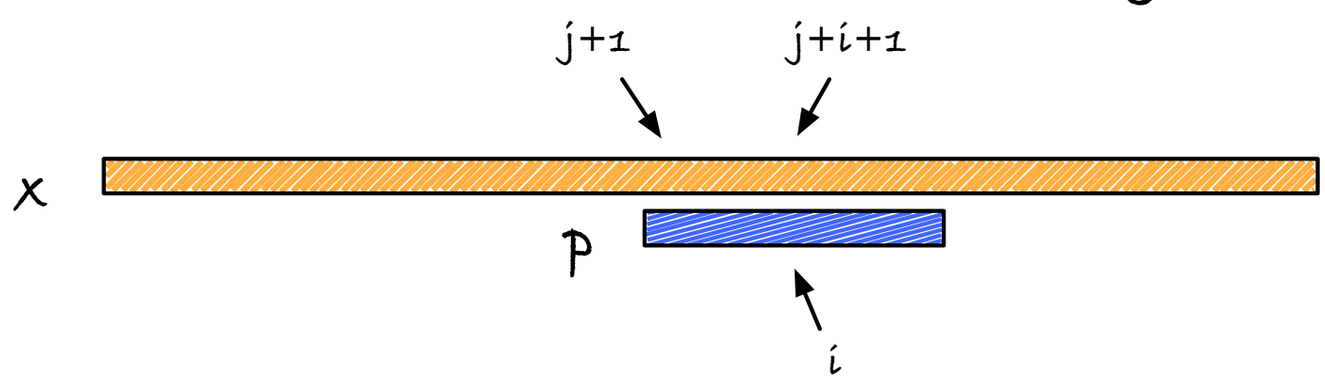
- (You can add some improvements to get worst case O(n+m) to it, though)

Ways to “align” p

x[j] = p[0] view:

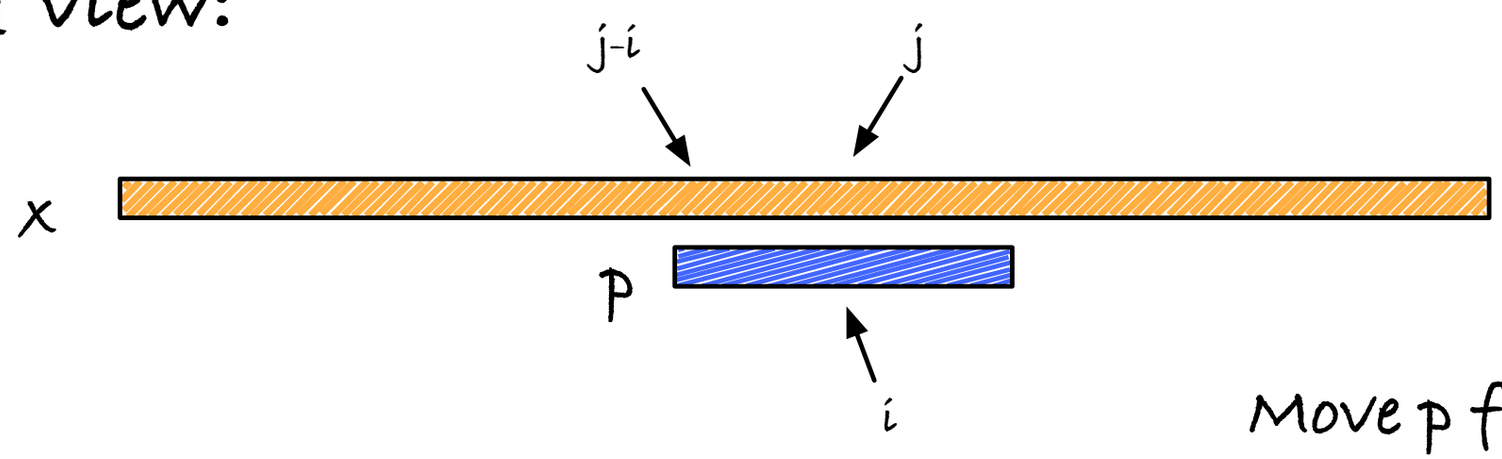


Move p forward by increasing j.

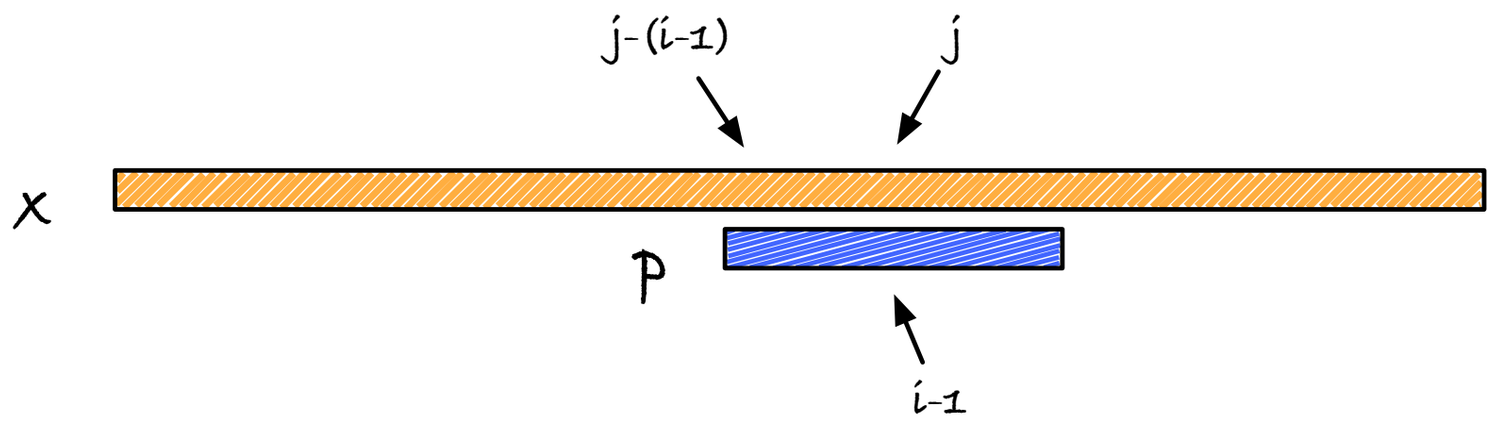


Ways to “align” p

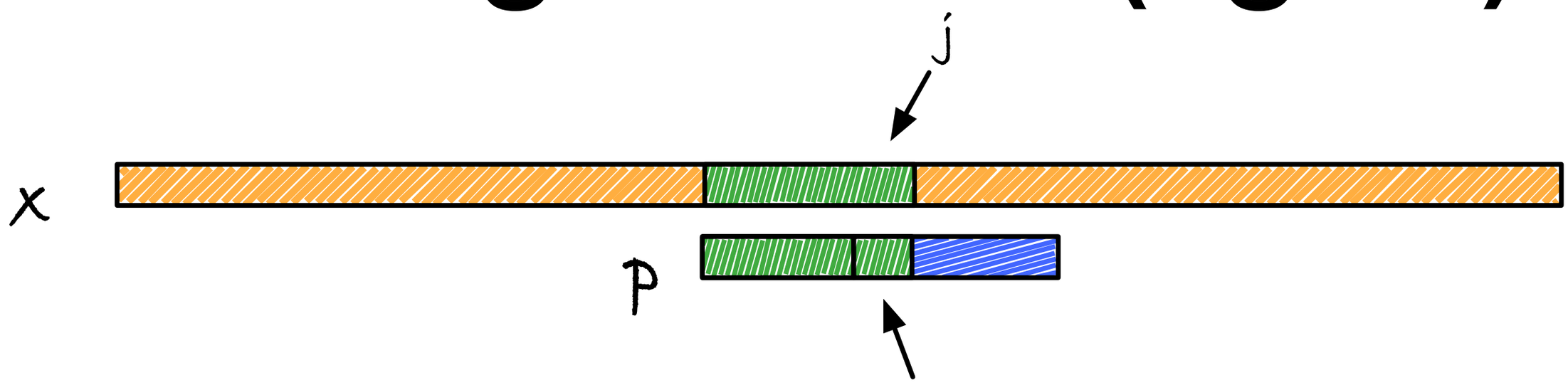
x[j] = p[i] view:



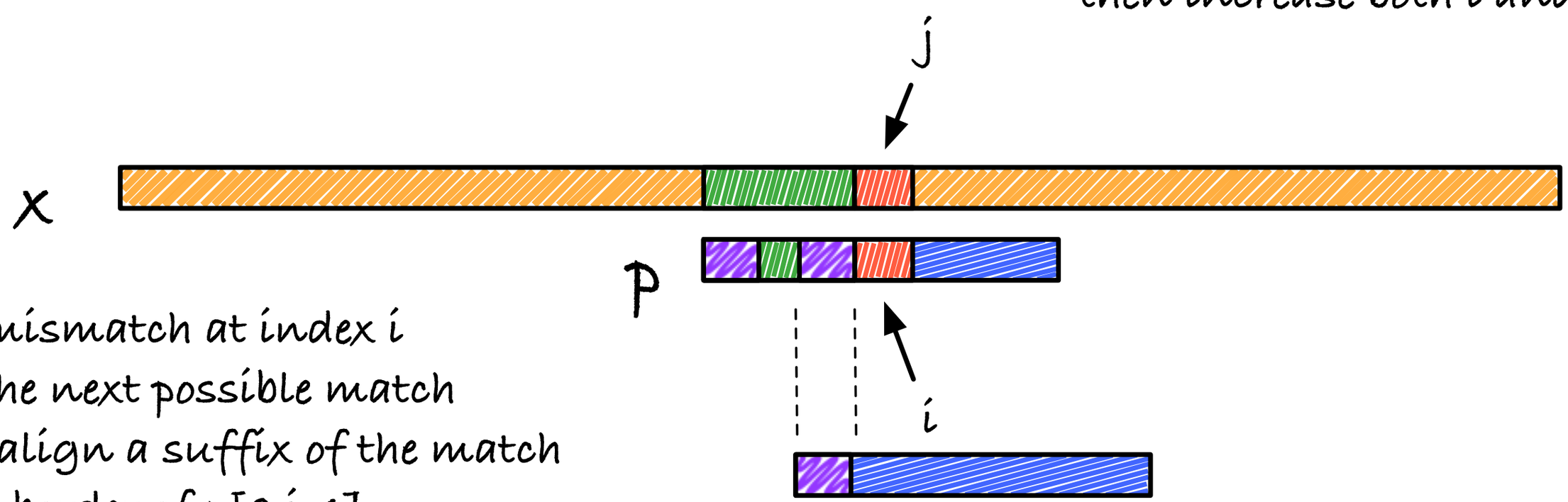
Move p forward by decreasing i.



Matching forward (again)



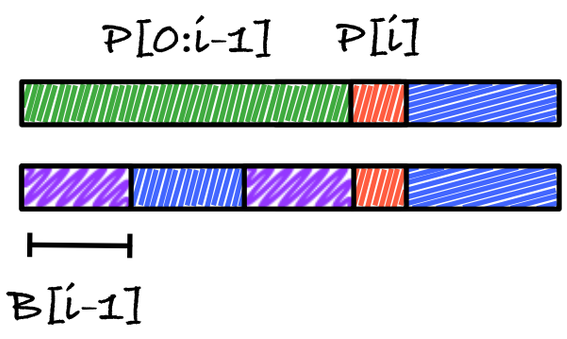
If we match at index i
then increase both i and j.

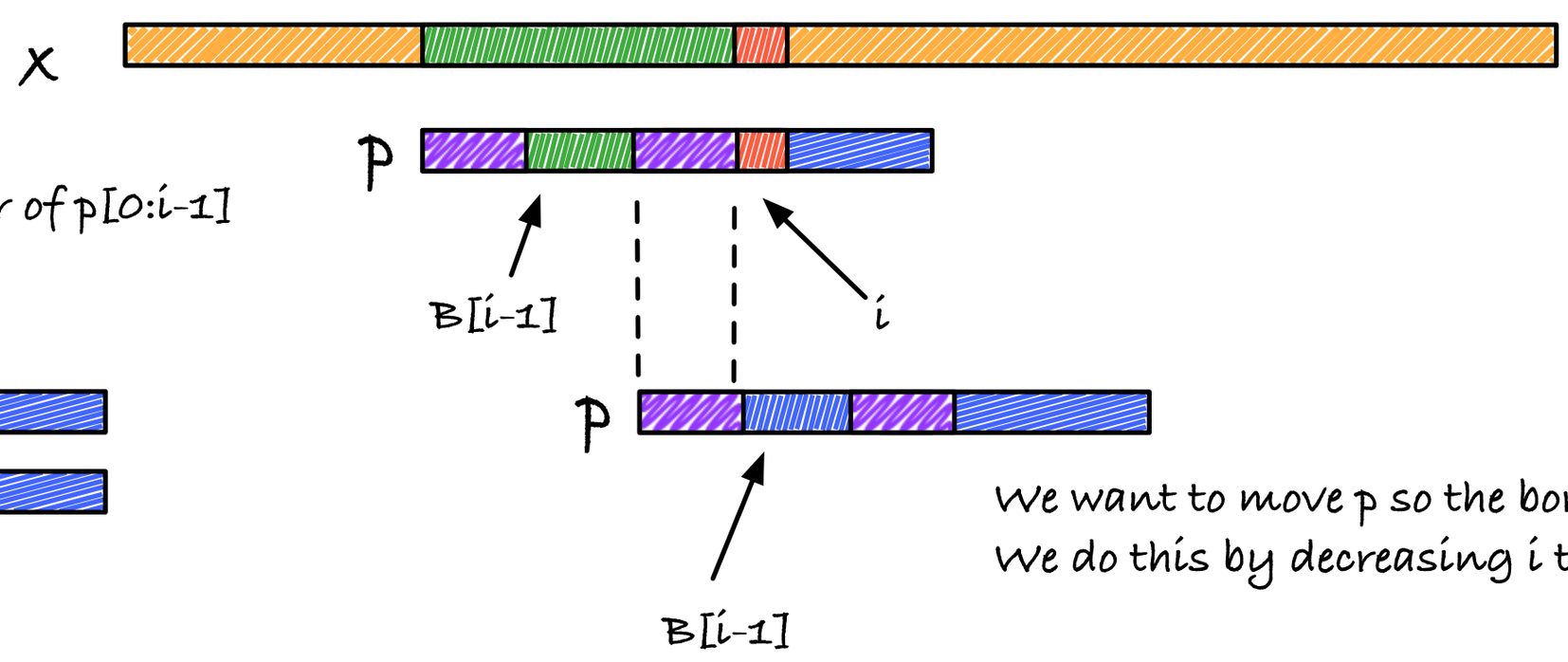


If we mismatch at index i
then the next possible match
must align a suffix of the match
with a border of p[0:i-1].

Matching borders...

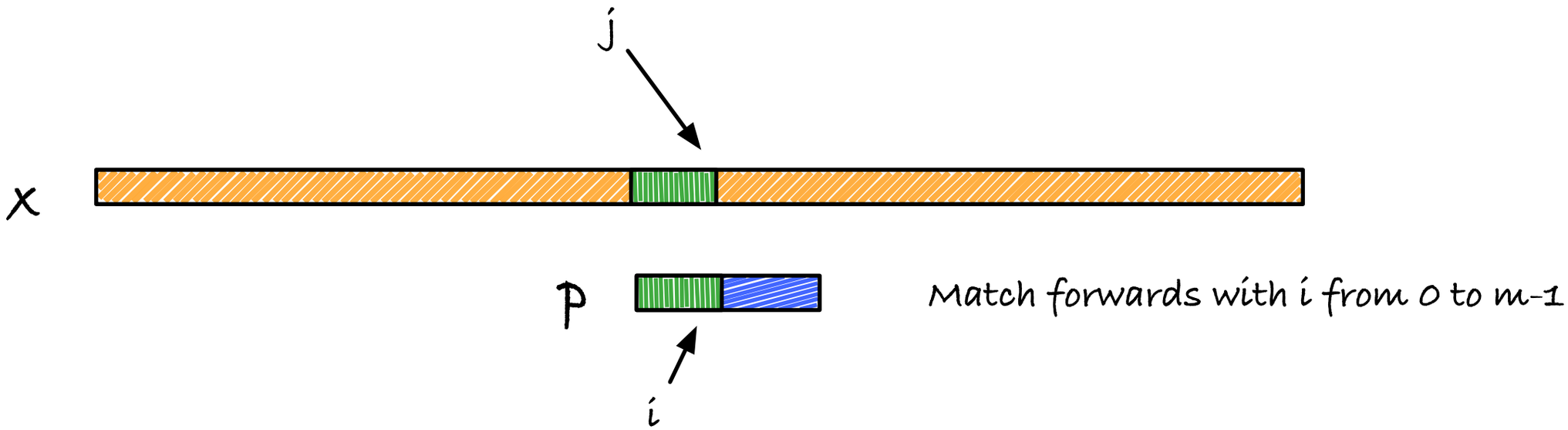
Consider the longest border of p[0:i-1]

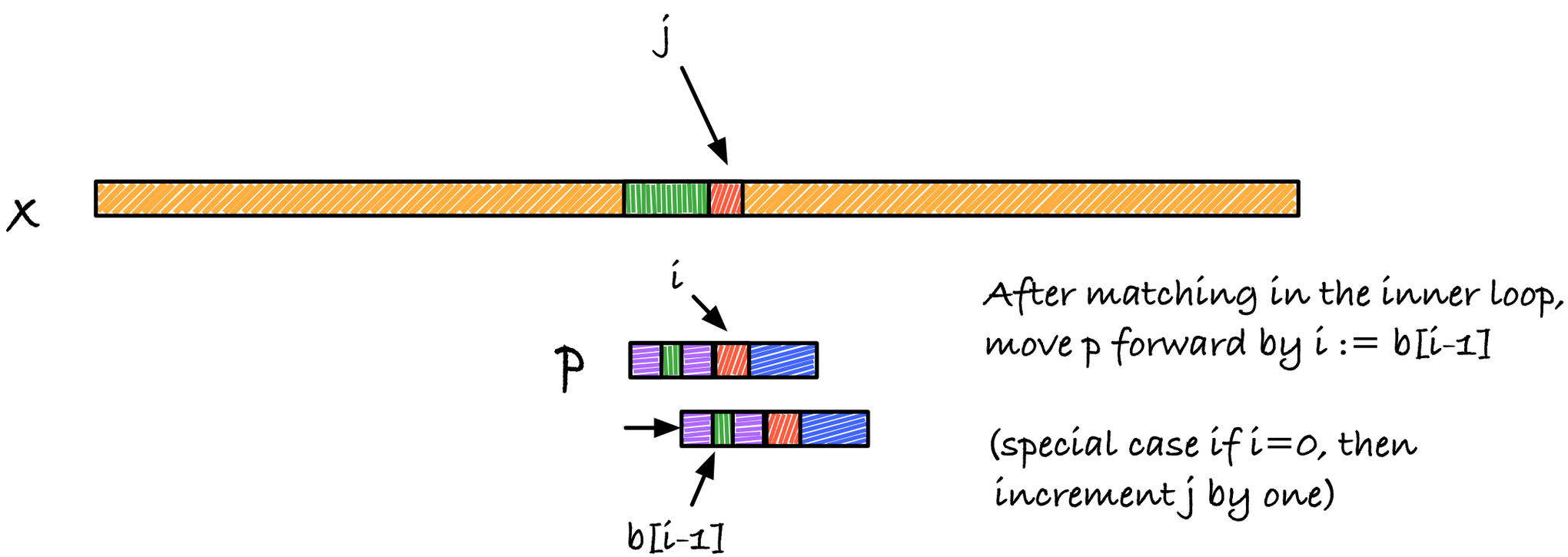




We want to move p so the borders line up.
We do this by decreasing i to B[i-1].

Knuth-Morris-Pratt





```
> gsa show exact x p kmp
```

Knuth-Morris-Pratt

```
for (int j = 0; j ≤ n - m; j++) {
    int i = 0;
    while (i < m && x[j+i] == p[i]) {
        i++;
    }

    if (i == m) {
        report(j);
    }
}
```

```
int i = 0, j = 0;
while (j < n) {
    while (i < m && j < n && x[j] == p[i]) {
        j++; i++;
    }

    if (i == m) {
        report(j);
    }

    if (i == 0) { j++; }
    else        { i = ba[i - 1]; }
}
```

(show code)

Running time

- Time bounded by the sum of matches and mismatches
  - Why?

Running time

- Time bounded by the sum of matches and mismatches
  - Each match increases *i*
  - Each mismatch either increases *i* or **p**'s position (conceptually)

Running time

- Time bounded by the sum of matches and mismatches
  - Each match increases i
  - Each mismatch increases **p**'s position (conceptually)
  - Matches bounded by n
  - Mismatches bounded by n+(n-m)
- Runtime in O(n)

That's all Folks!