Exact pattern matching

A crude approach to read mapping

Reminder

- The goal of read mapping is to find approximate matches of billions of short reads against a string that is billions of characters long…

A crude approach

- Can we do efficient *exact* pattern matching?
- If so, can we solve the problem by generating all strings "close" to a read?

Exact pattern matching

- Constructing all strings close to a read is an exercise…
- Today we focus on the problem of *exact pattern matching*

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

6

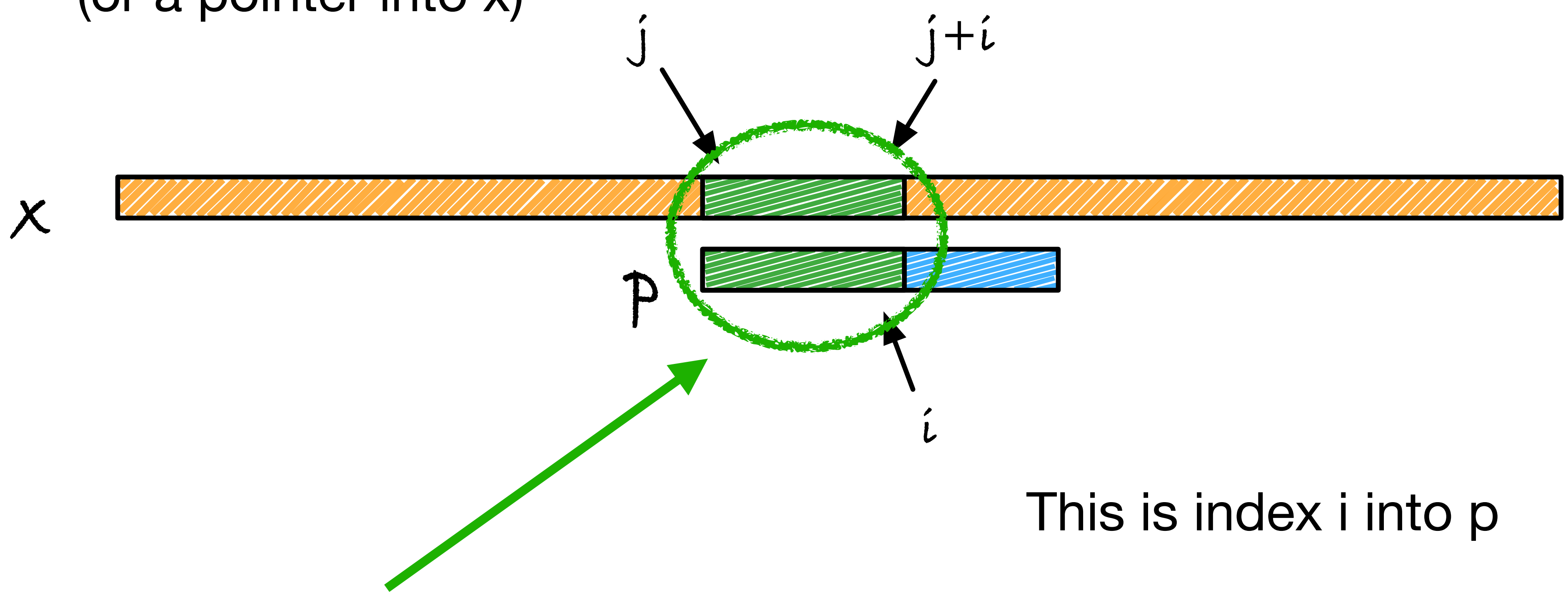
17

x=*abbacbbbababacabbbba*

Notation

This is index j in x
(or a pointer into x)

This is index j+i into x



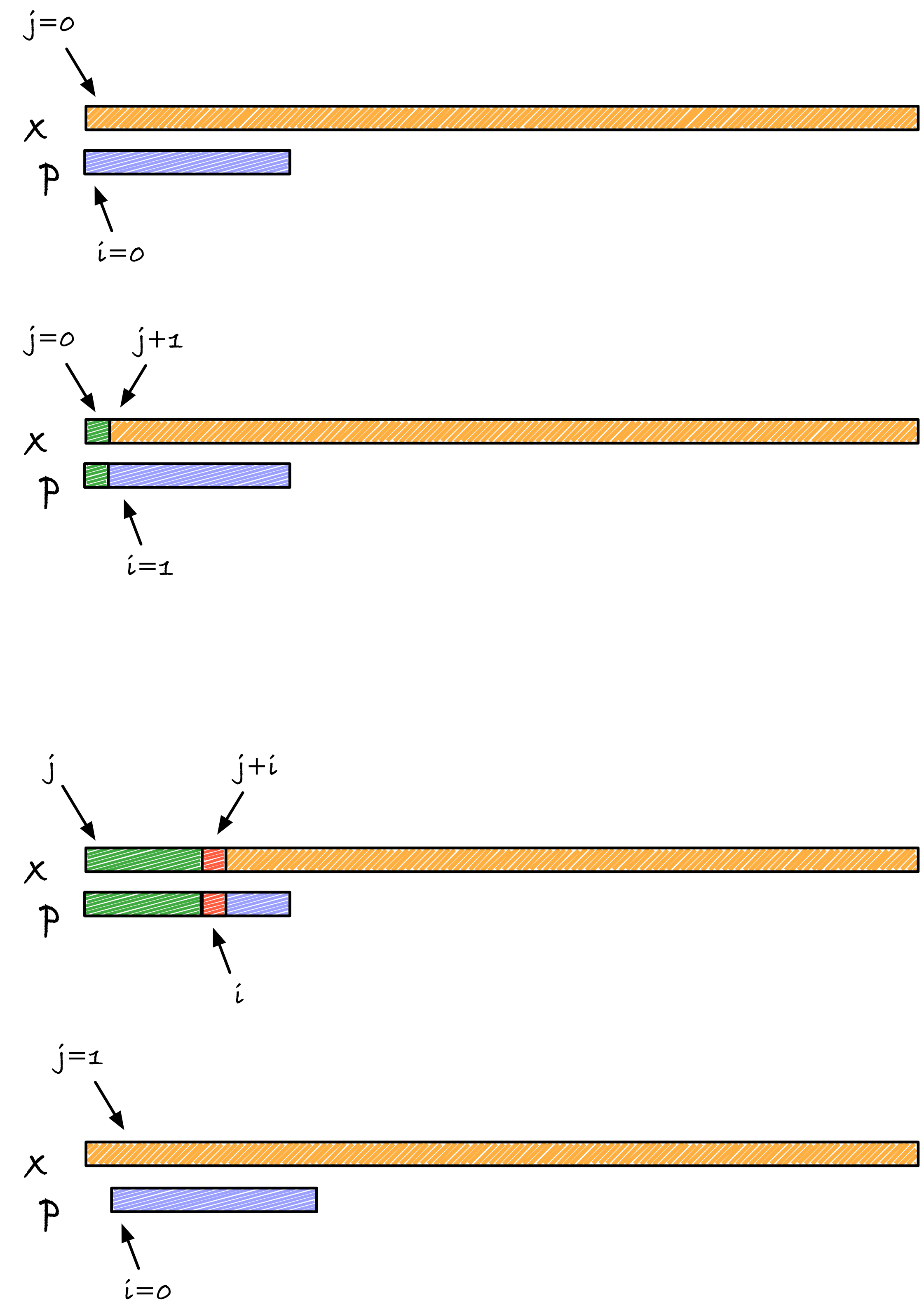
This is index i into p

When we draw a box like this, read it
as these parts of the strings are equal
(x[j:j+i] == p[0:i])

We never move p around to align it under x,
but we do it conceptually all the time, and in different ways.
It is helpful to think about p sliding along x, but it is always
indices that change that give us that effect.

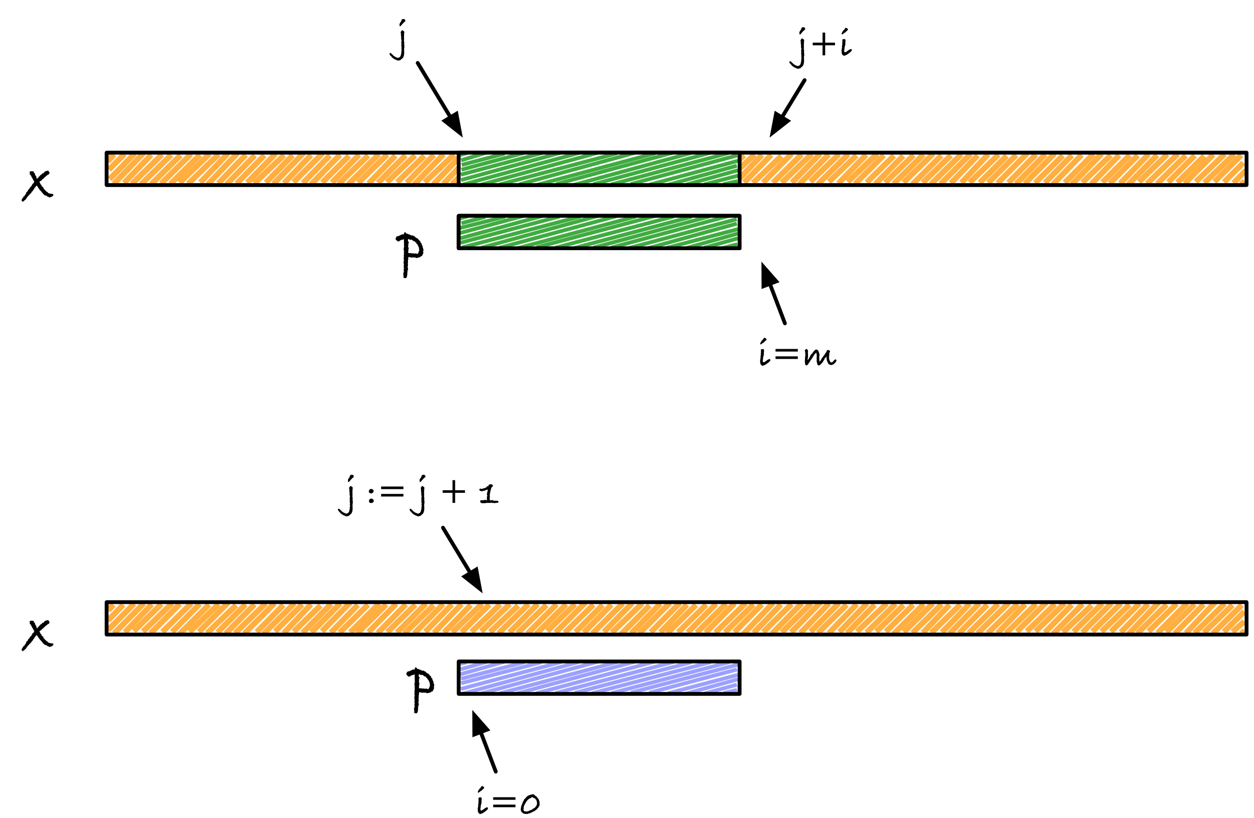
The naïve algorithm

- Point j at first character in x, i at first character in p, then match forward, x[j+i] vs p[i]…
    - As long as there is a match, x[j+i] == p[i] increment i
    - If there is a mismatch, abandon hope, increment j and set i to zero.



The naïve algorithm

- If we reach i = m, the end of p, we have a match that we can report.
- After that, move j one forward.



```
> gsa show exact x p naive
```

```
> pip3 install git+https://github.com/birc-gsa/gsa#egg=gsa
```

or

```
> git clone https://github.com/birc-gsa/gsa.git
```

Requires Python 3.10

The naïve algorithm

What is the worst case time complexity of this algorithm?
A worst case example?

The naïve algorithm

What is the worst case time complexity of this algorithm?
A worst case example?

What is the best case running time?
A best case example?

Linear time algorithms?

- If x has length n and p has length m the worst case is $O(nm)$
- An early algorithmic goal was to improve on this and there are many algorithms with worst case running time $O(n+m)$
- Today, we see a particularly simple one based on *borders*

What is a border?

A border of a string ***x*** is any *proper* prefix of ***x*** that equals a suffix of ***x***.

```
abaabbbbabaab
ab         ab
abaab   abaab
```

A *prefix* of string *x* is a substring x[0:j]
A *suffix* of string *x* is a substring *x*[i:n]
They are *proper* if they are not the full string

Computing borders

```
abaabbbbabaab (empty border)
abaabbbbabaab
abaabbbbabaab
abaabbbbabaab
abaabbbbabaab
abaabbbbabaab
abaabbbbabaab
abaabbb         (also consider overlapping pre/suffixes)
      bbabaab
abaabbbb
     bbbabaab
abaabbbba
    bbbbabaab
abaabbbbab
   abbbbabaab
abaabbbbaba
  aabbbbabaab
abaabbbbabaa
 baabbbbabaab
```

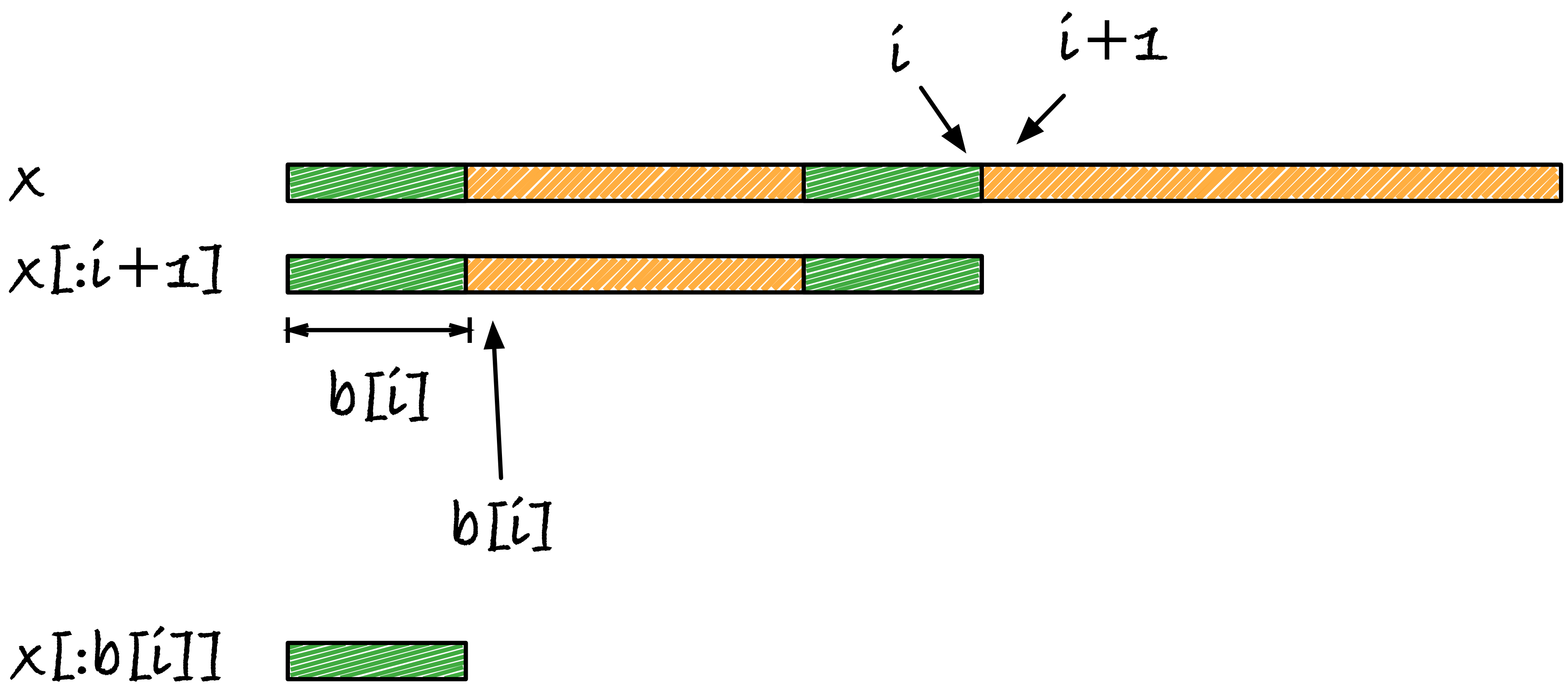
How long does it take to check all prefixes versus suffixes?

Border arrays

- We are actually not so much interested in borders as *border arrays*
  - These also give us a faster way to compute the longest border of a string…

What is a border array?

The **border array** of string x is an integer array b, where
b[i] is the *length* of the *longest border* of x[0:i+1]



When we consider *i* here we include the letter x[i], unlike other places.
It is inconsistent, but it is easier some places, including in the calculation.

Computing border arrays

How would you compute the border array in the simplest way?

What would the running time be?

Induction

(sweet child has many names)

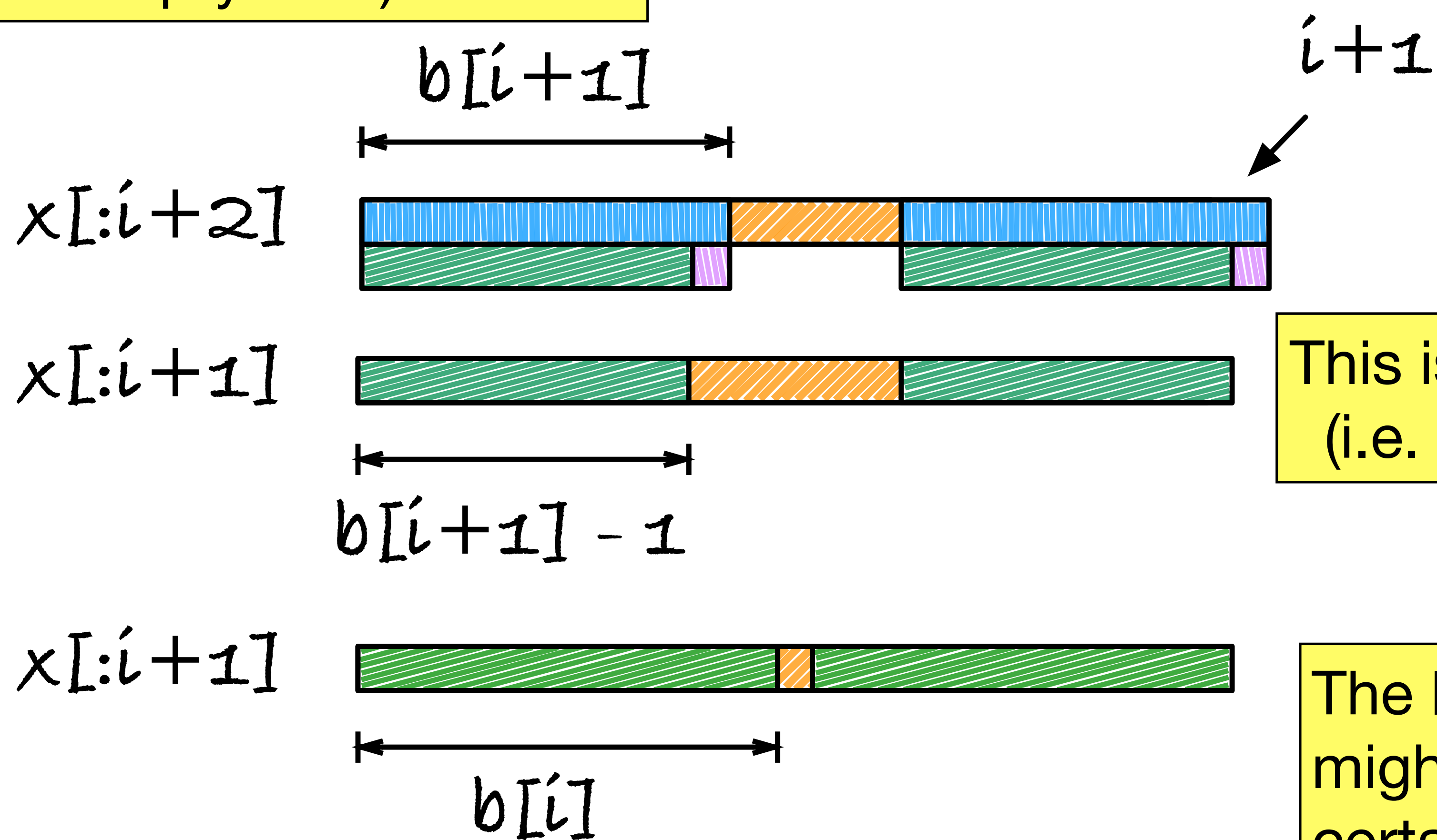
- Handle base case
- Handle $i + 1$ assuming solution to all $j \leqq i$

Trivial base case…

b`[0] = 0`

Observation

Assume here that the border isn't empty…
(we'll handle empty later)

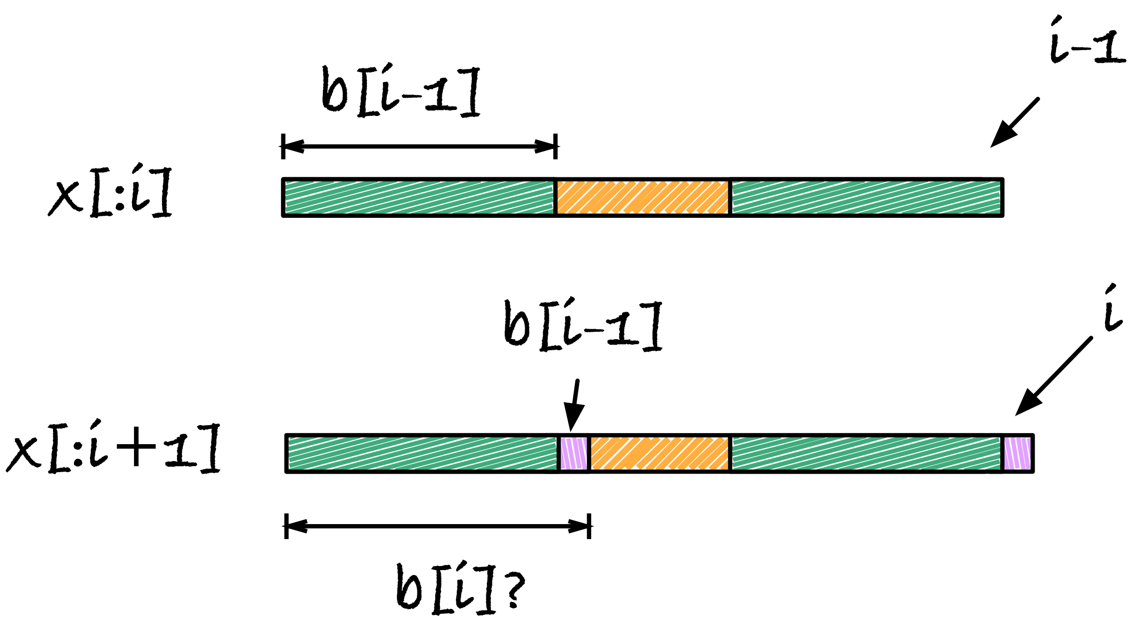


This is a border of x[:i+1]
(i.e. one that ends at i)

The longest border, b[i] might be longer, but it certainly cannot be shorter…

$$b[i] \geq b[i+1]-1 \quad \text{thus} \quad b[i+1] \leq b[i]+1$$

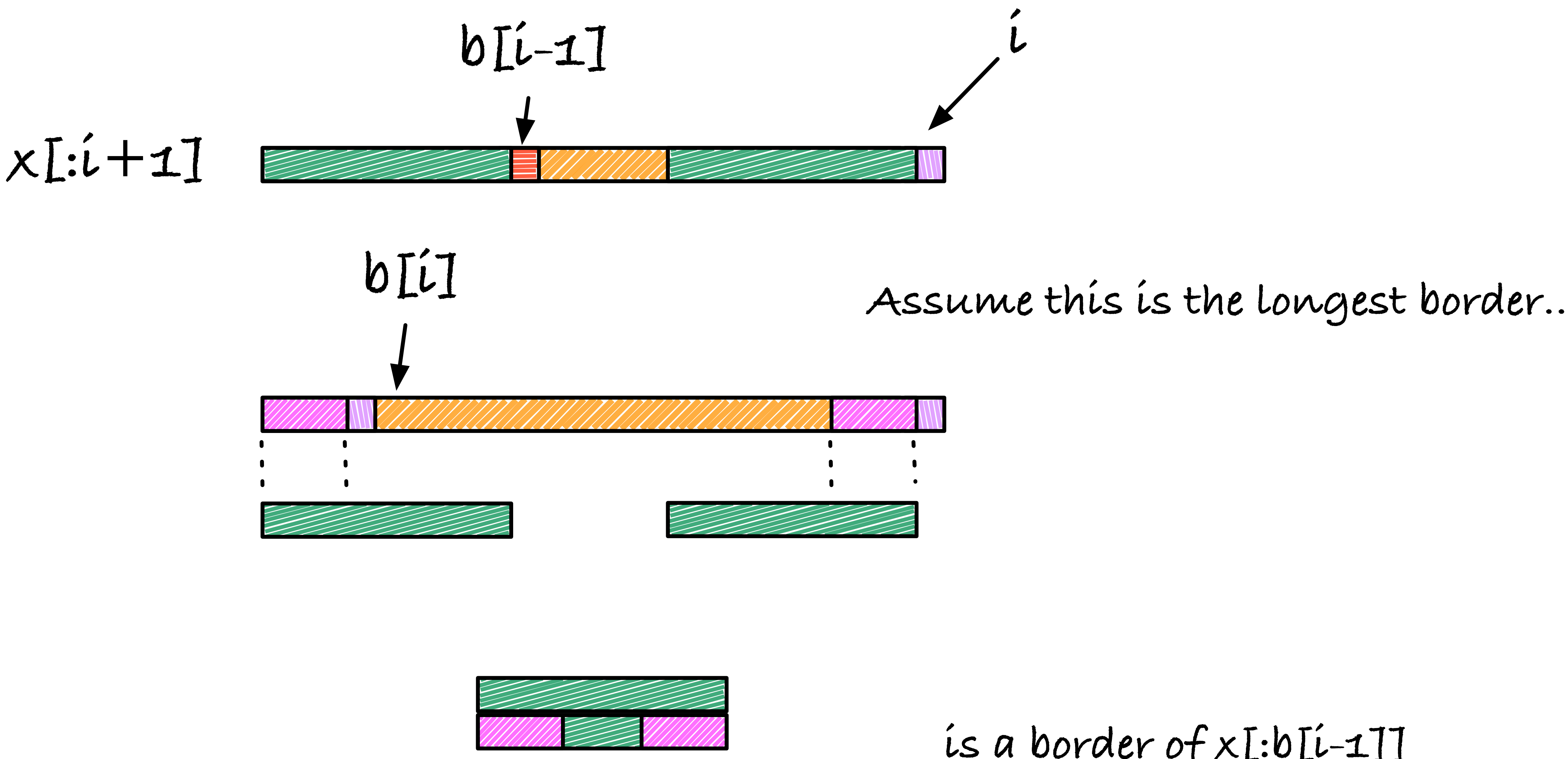
Observation



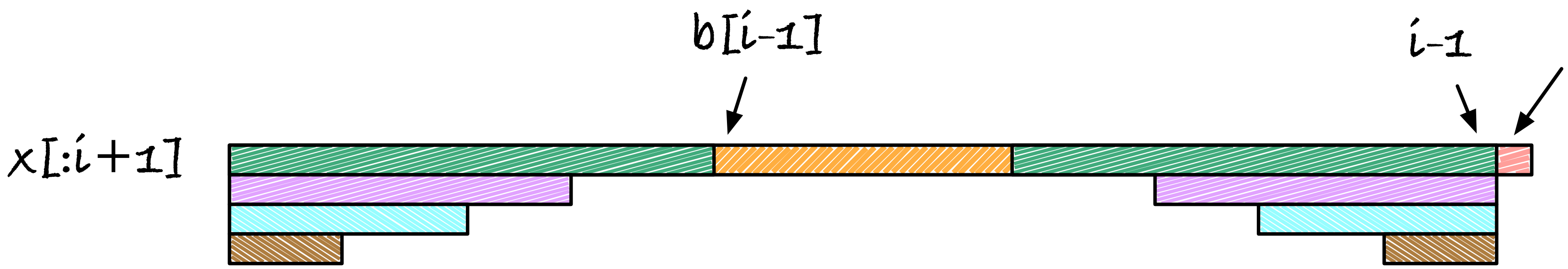
$b[0] = 0$

$b[i] = b[i\text{-}1] + 1 \text{ if } x[b[i\text{-}1]] = x[i]$

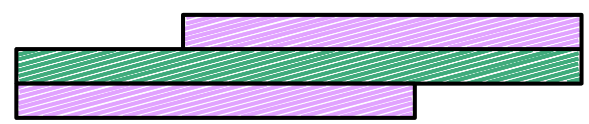
Observation



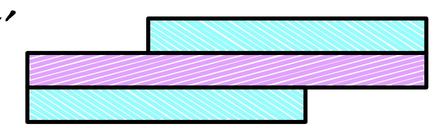
Observation



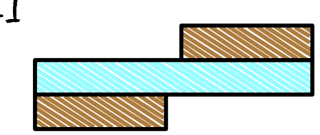
The second-longest border of x[:i]
is the longest border of x[:b[i-1]].
That is b[b[i-1]-1]. Call it b2.



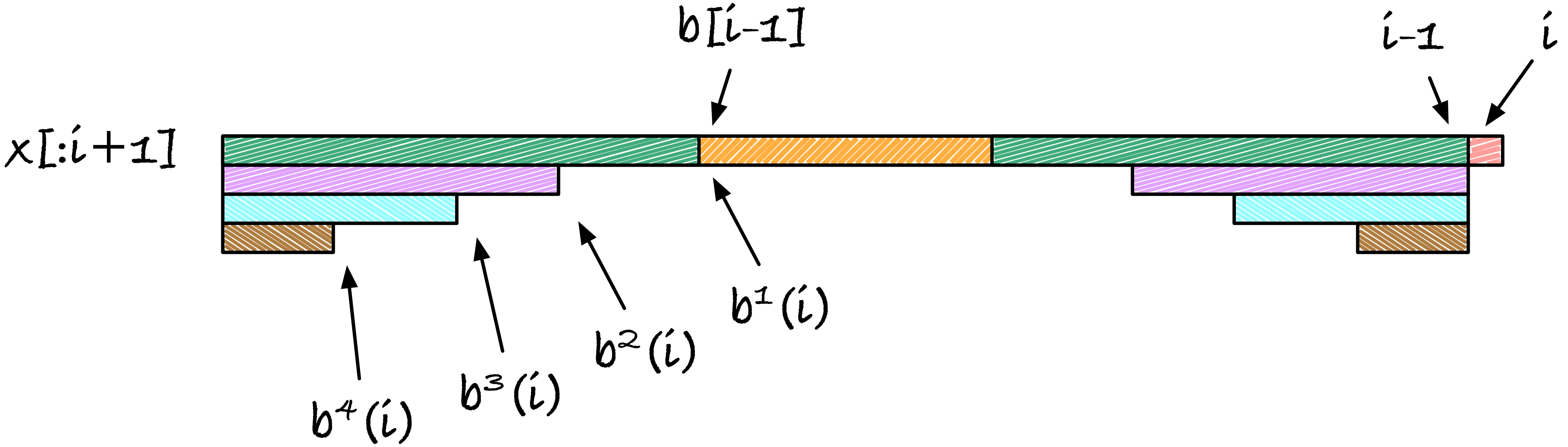
The third-longest border of x[:i]
is the longest border of x[:b2],
i.e. b[b2-1], call it b3.



The fourth-longest border of x[:i+1]
is the longest border of x[:b3],
call it b4.



Observation



Define $b^k(i)$ as a function:

$b^0(i) = i$

$b^k(i) = b[b^{k-1}(i)-1]$

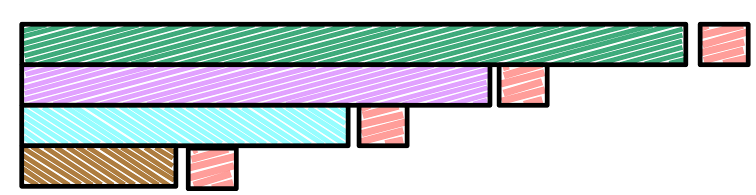
(move one left and jump to border)

$b^0(i) = i$

$b^1(i) = b[i-1]$

$b^2(i) = b[b^1(i)-1] = b[b[i-1]-1]$

$b^3(i) = b[b^2(i)-1] = b[b[b^1(i)-1]-1] = b[b[b[i-1]-1]-1]$

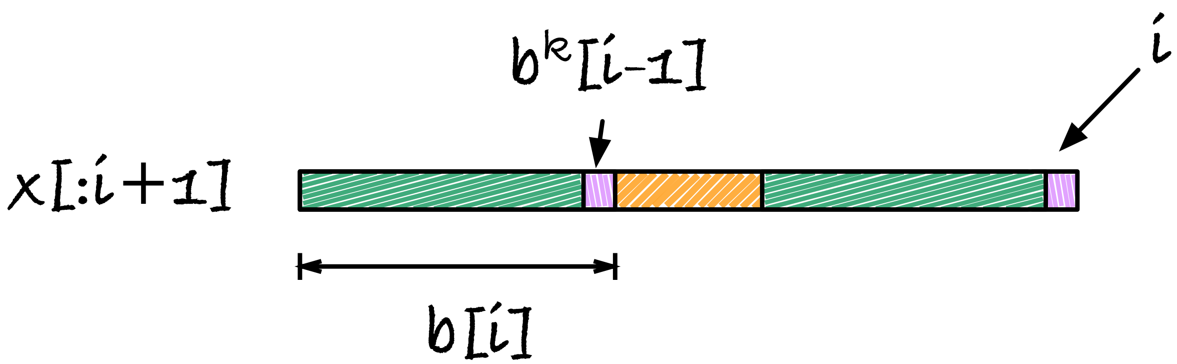


```
j = 1
while x[:b^j(i)] ≠ "":
    try to expand border…
```

Algorithm

Start by setting i = 0 and b[0] = 0

Then for i up to n-1, run through the borders of x[:i-1] from longest (b[i-1]) to shortest (b[0]) and try to extend them (you can do it if $x[b^k[i-1]]=x[i]$).



$b[i] = b^k[i-1]+1$

If you cannot find a border to extend, the longest border is the empty string, so then b[i] = 0.

Running time…

- For index i from 0 to n…
    - For k from 1 to (?) check $x[b^k[i-1]] == x[i]$

Don't compute $b^k(i)$ each time.
Use a variable, v, initialised as v := b[i-1] and updated as v := b[v-1].
That is a constant time update to go to the next border you want to try.

Running time…

- For index i from 0 to n…
  - For k from 1 to (?) check $x[b^k[i-1]] == x[i]$

Naïve analysis: $O(n^2)$ because we run i from 0 to n and k is bounded by n.

More detailed: $O(n)$ + how many times we go $b^k \rightarrow b^{k+1}$ in total.

Amortised analysis

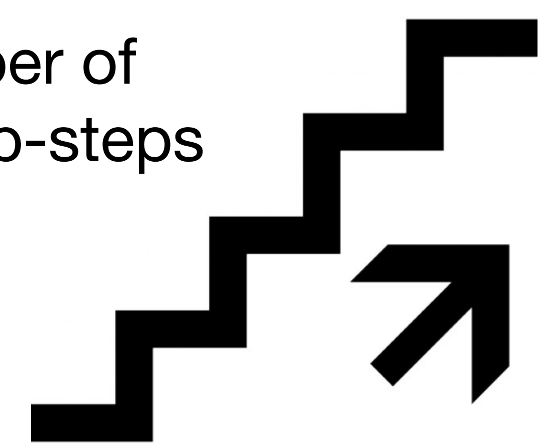
Bankers' method (save up computations)

E.g. binary odometer (put one coin on each new 1)

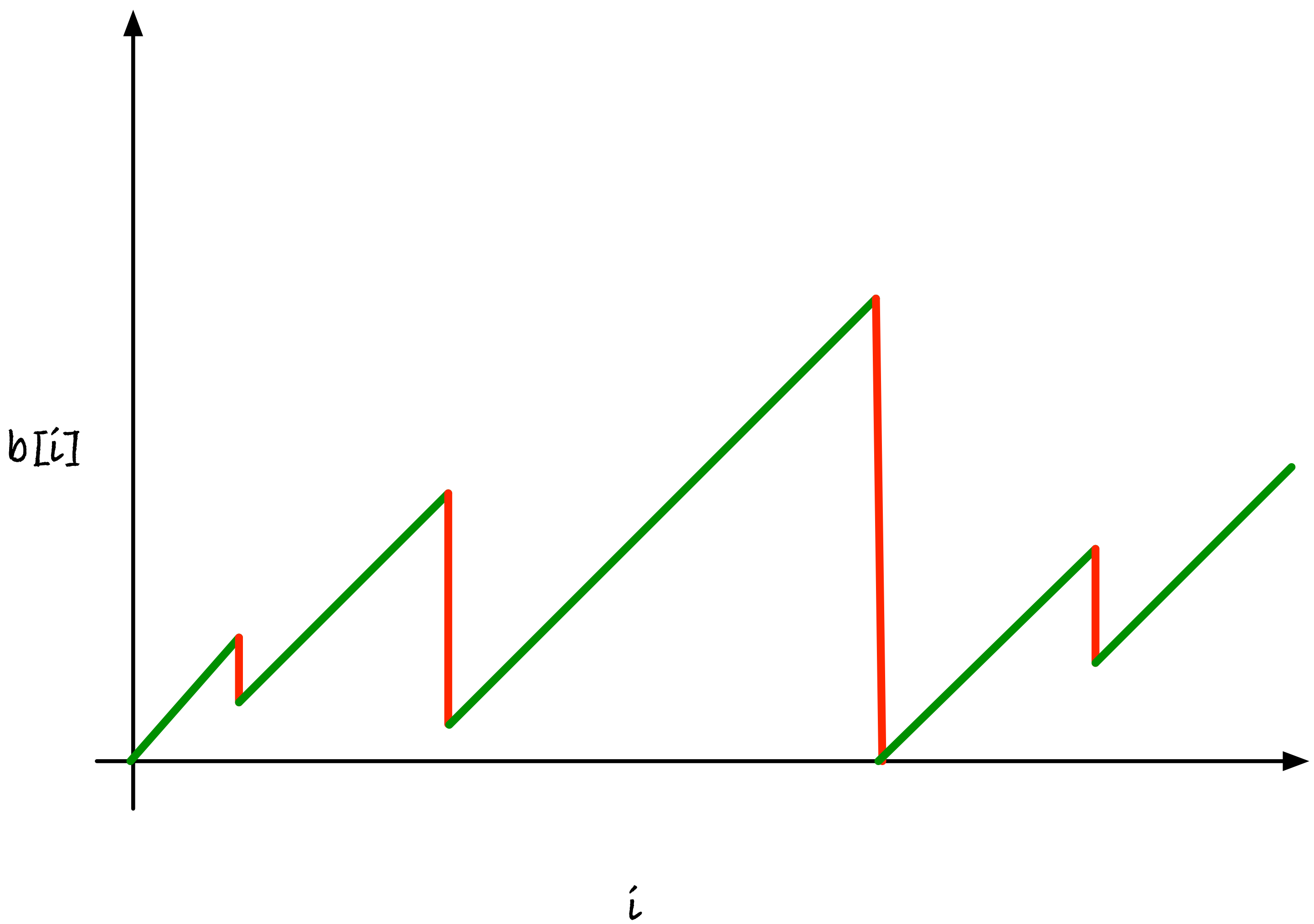
```
*  ****     → * *
1001111 + 1 → 1010000
(cost: flip four 1s, but you have four coins)
```

Potential method (increasing and decreasing potential)

Think e.g. stairs. You cannot go below the lowest floor, so the number of down-steps are bounded by the floor you start on and how many up-steps you take. We need this version now.



Running time…



Expanding border
O(n) because we cannot increase more than 1 for each i.

Cost of exploring borders
≤ total expanding ≤ O(n)

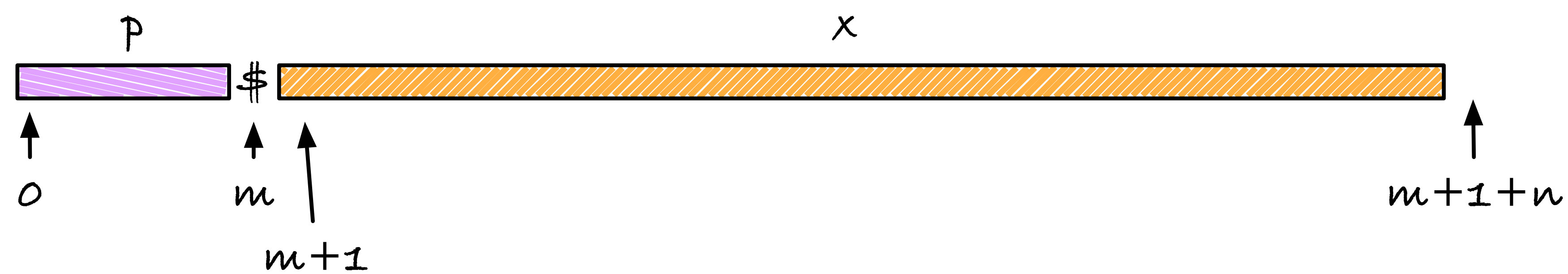
Border arrays and exact pattern matching

Problem: Given text $\boldsymbol{x}[0{:}n]$ and pattern $\boldsymbol{p}[0{:}m]$, find all occurrences of $\boldsymbol{p}$ in $\boldsymbol{x}$, i.e. all i where $\boldsymbol{x}[i{:}i+m] = \boldsymbol{p}[0{:}m]$.

Solution: Can be solved naively in time $O(nm)$. More efficient solutions (both in theory and practice) exists.

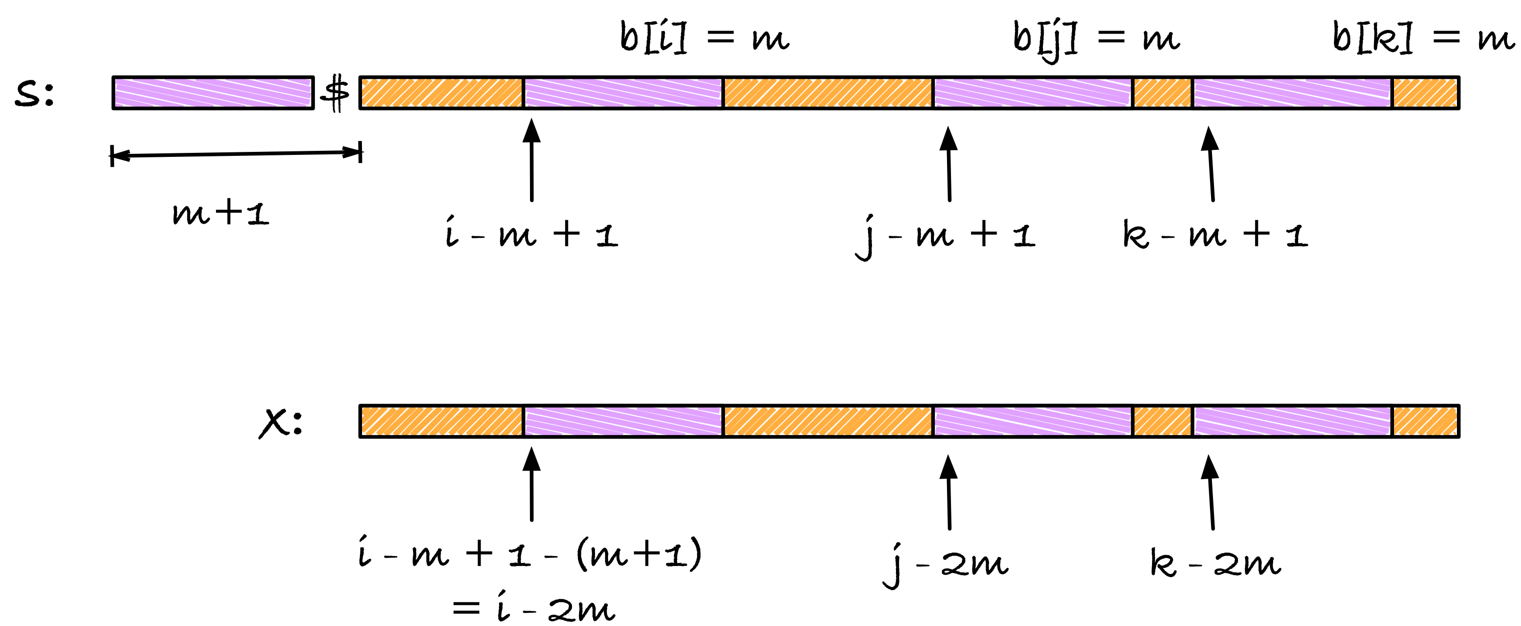
There is a (surprisingly) simple solution based on border arrays that has worst-case running time $O(n+m)$, i.e. the same running time as the classic KMP algorithm (which we will see next week).

Step 1: Construct $\boldsymbol{s} = \boldsymbol{p}\$\boldsymbol{x}$ and its border array b



Observation 1: No border of $\boldsymbol{s}$ has length more than m because of $

Observation 2: $\boldsymbol{p}$ occurs in $\boldsymbol{x}$ at pos. i-m+1-(m+1) = i-2m iff $b[i]$=m



Step 2: Report an occurrence of $\boldsymbol{p}$ in $\boldsymbol{x}$ at position i-2m iff $b[i]$=m

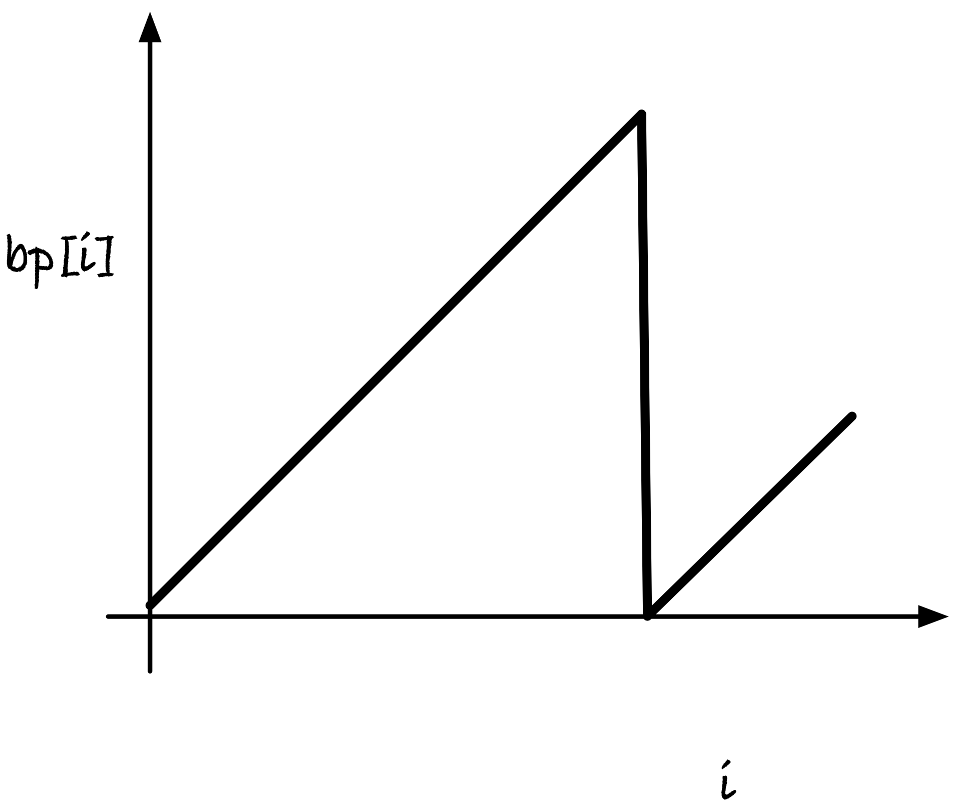
Time and space O(n+m)

Further observations…

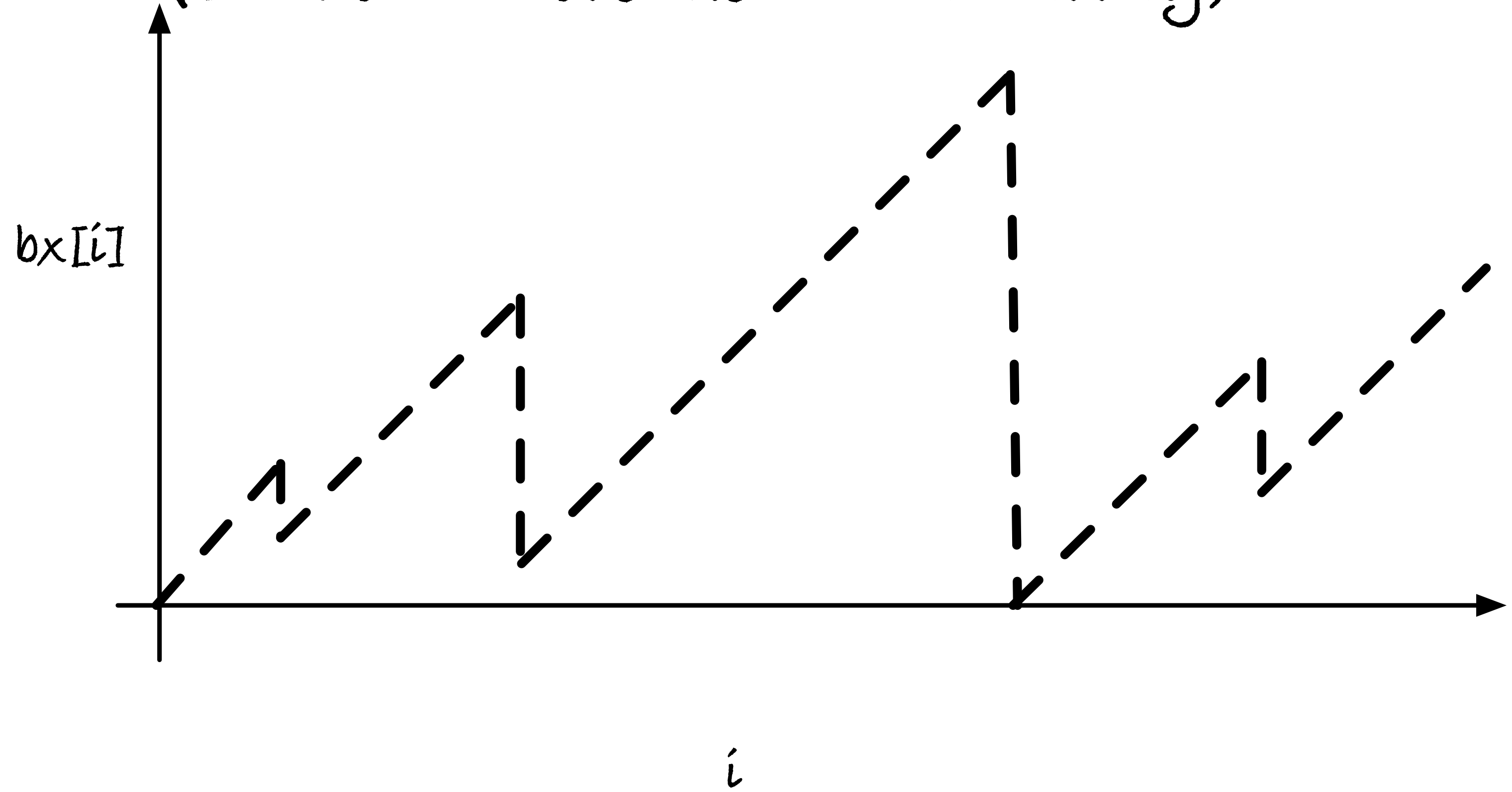
- You don’t need to construct the new string. You can always fake it from the two original strings.

- You only need to construct the border array for p. You never need to look up borders longer than m, since there are none. For x, you only need to know the length of the current border, and you can represent that with a single number.

Compute border array for p



For x, just keep track of border length
(but don't store them in an array)



```
> gsa show exact x p border
```

That's all Folks!