pattern matching

crucial for deciphering genomes

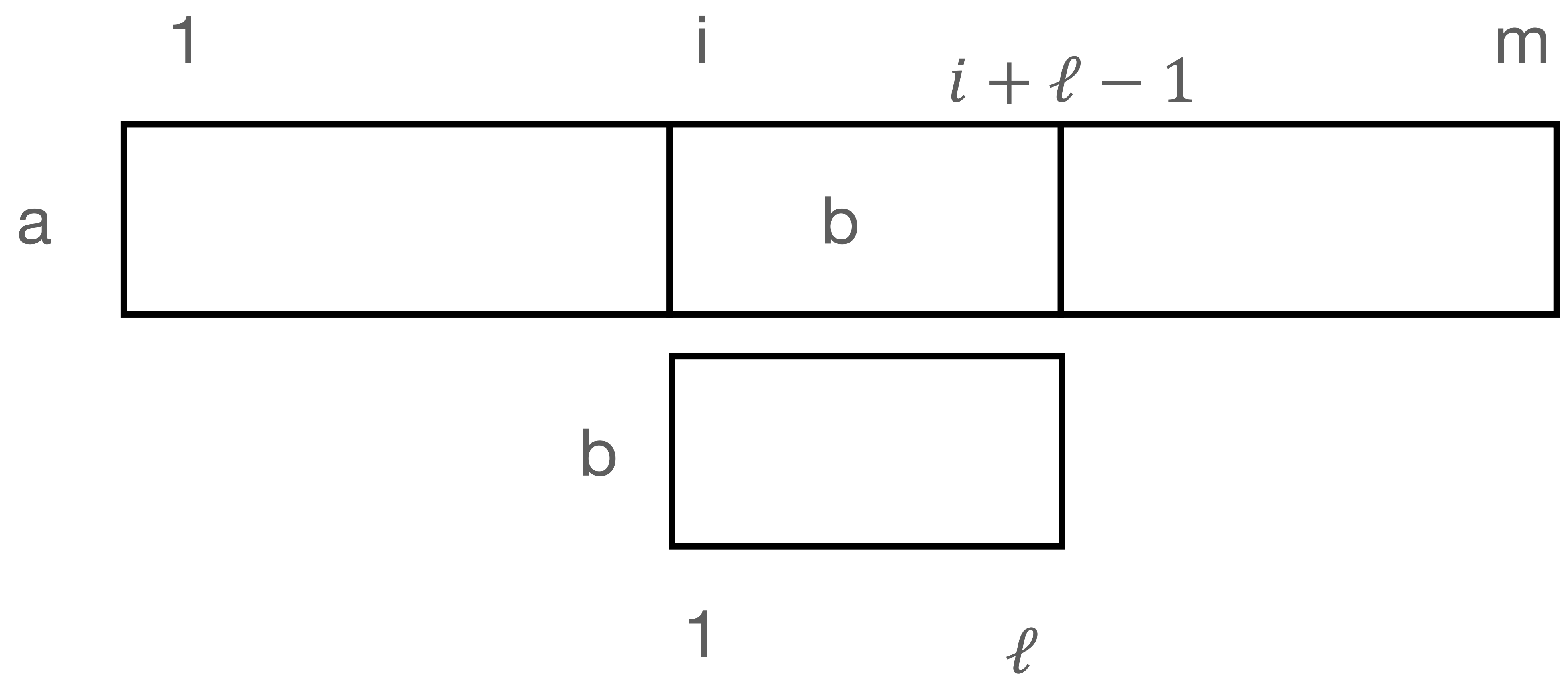
Morris & Pratt 1970

substring recognition: spec

spec:

- inputs
  - $b = b[1:\ell] \in \Sigma^*$: pattern
  - $a = a[1:m] \in \Sigma^*$: string
- output: position i of first occurrence of b in a

$$out = \begin{cases} \min\{i \mid b[1:\ell] = a[i:i+\ell-1]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

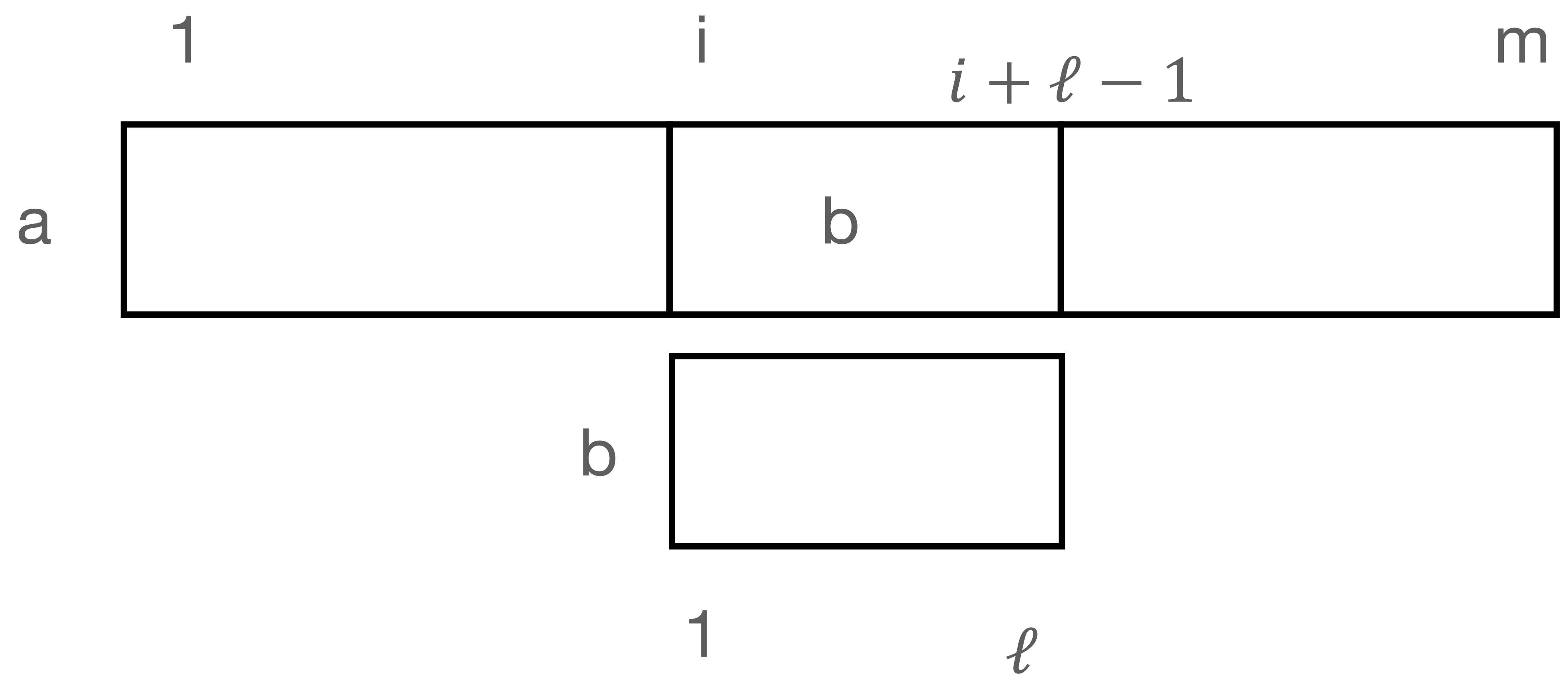


substring recognition: spec

spec:

- inputs
  - $b = b[1:\ell] \in \Sigma^*$: pattern
  - $a = a[1:m] \in \Sigma^*$: string
- output: position i of first occurrence of b in a

$$out = \begin{cases} \min\{i \mid b[1:\ell] = a[i:i+\ell-1]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

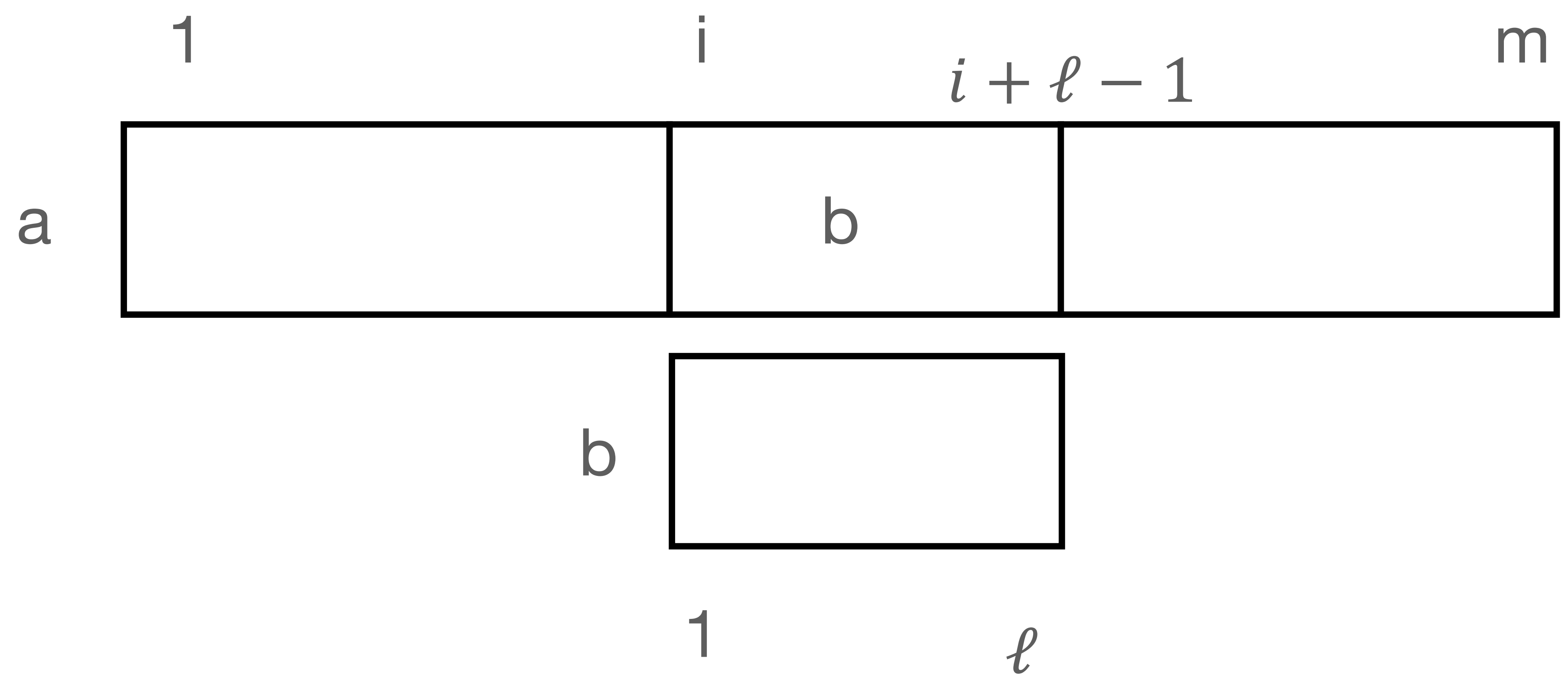


cost:

number of:

- instructions of compiled code on MIPS
- C-instructions (if compiled into $O(1)$ MIPS instructions)
- lines of executed C-code (if compiled into $O(1)$ MIPS instructions)
- comparisons between symbols $\in \Sigma$ (if they amount to a constant fraction of the C instructions)

MIPS and C models of computation with data of unbounded size



spec:

- inputs
  - $b = b[1:\ell] \in \Sigma^*$: pattern
  - $a = a[1:m] \in \Sigma^*$: string
- output: position i of first occurrence of b in a

$$out = \begin{cases} \min\{i \mid b[1:\ell] = a[i:i+\ell-1]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

cost:

number of:

- instructions of compiled code on MIPS
- C-instructions (if compiled into $O(1)$ MIPS instructions)
- lines of executed C-code (if compiled into $O(1)$ MIPS instructions)
- comparisons between symbols $\in \Sigma$ (if they amount to a constant fraction of the C instructions)

a word of caution about C and MIPS:

- MIPS registers, MIPS memory and C data types are finite
- problem sizes $n, m, \ell, \ldots$ and time t unbounded and we study asymptotic browth, i.e. $n, t \to \infty$
- we allow MIPS register size and MIPS data types to grow with $O(\log t)$.
- this also makes addressable memory grow with $2^{O(\log t)}$

naive solution

spec:

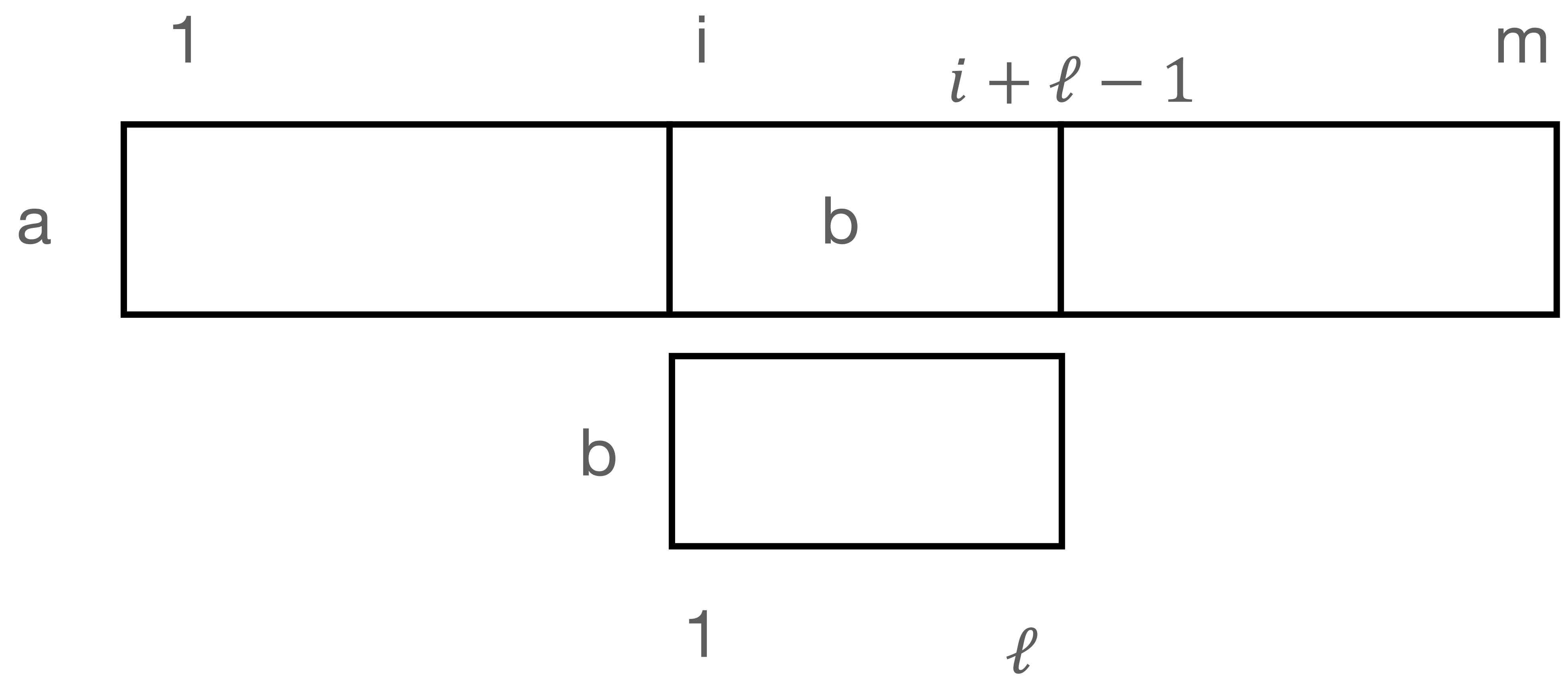
- inputs
  - $b = b[1:\ell] \in \Sigma^*$: pattern
  - $a = a[1:m] \in \Sigma^*$: string
- output: position i of first occurrence of b in a

$$out = \begin{cases} \min\{i \mid b[1:\ell] = a[i:i+\ell-1]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

cost:

number of:

- instructions of compiled code on MIPS
- C-instructions (if compiled into $O(1)$ MIPS instructions)
- lines of executed C-code (if compiled into $O(1)$ MIPS instructions)
- comparisons between symbols $\in \Sigma$ (if they amount to a constant fraction of the C instructions)



Naïve solution

```
test(i,b,a):
for k=1 to l
 if b[k] != a[i+k-1]
   break & return 0;
 return 1.
```

Cost ℓ

naive solution

spec:

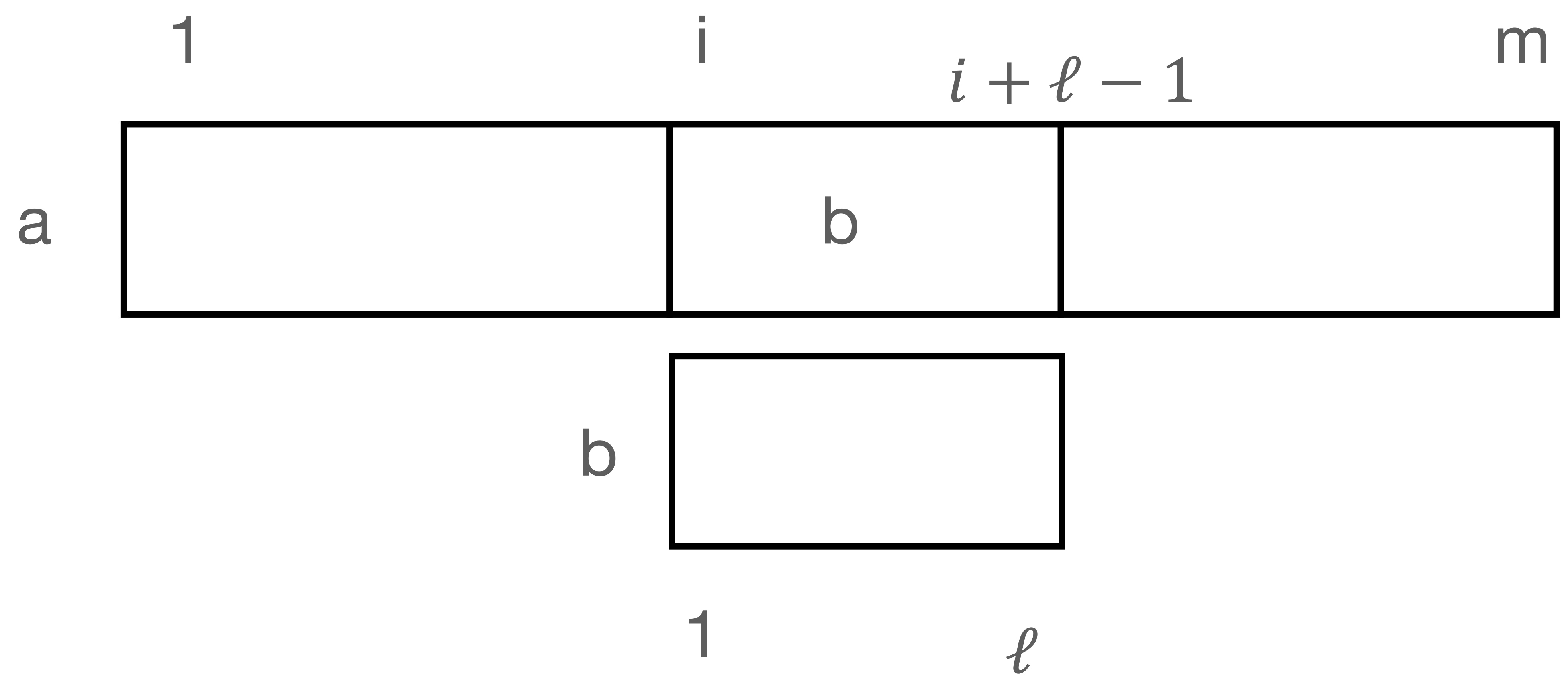
- inputs
  - $b = b[1:\ell] \in \Sigma^*$: pattern
  - $a = a[1:m] \in \Sigma^*$: string
- output: position i of first occurrence of b in a

$$out = \begin{cases} \min\{i \mid b[1:\ell] = a[i:i+\ell-1]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

cost:

number of:

- instructions of compiled code on MIPS
- C-instructions (if compiled into $O(1)$ MIPS instructions)
- lines of executed C-code (if compiled into $O(1)$ MIPS instructions)
- comparisons between symbols $\in \Sigma$ (if they amount to a constant fraction of the C instructions)



Naïve solution

```
test(i,b,a):
for k=1 to l
 if b[k] != a[i+k-1]
   break & return 0;
 return 1.
```

Cost ℓ

```
find(b,a):
i=1;
while ! test(i)  & i <= m-l+1
{i = i+1};
return (i <=m-l+1? i:0)
```

$\boxed{\text{cost } O(\ell \cdot m)}$

naive solution

spec:

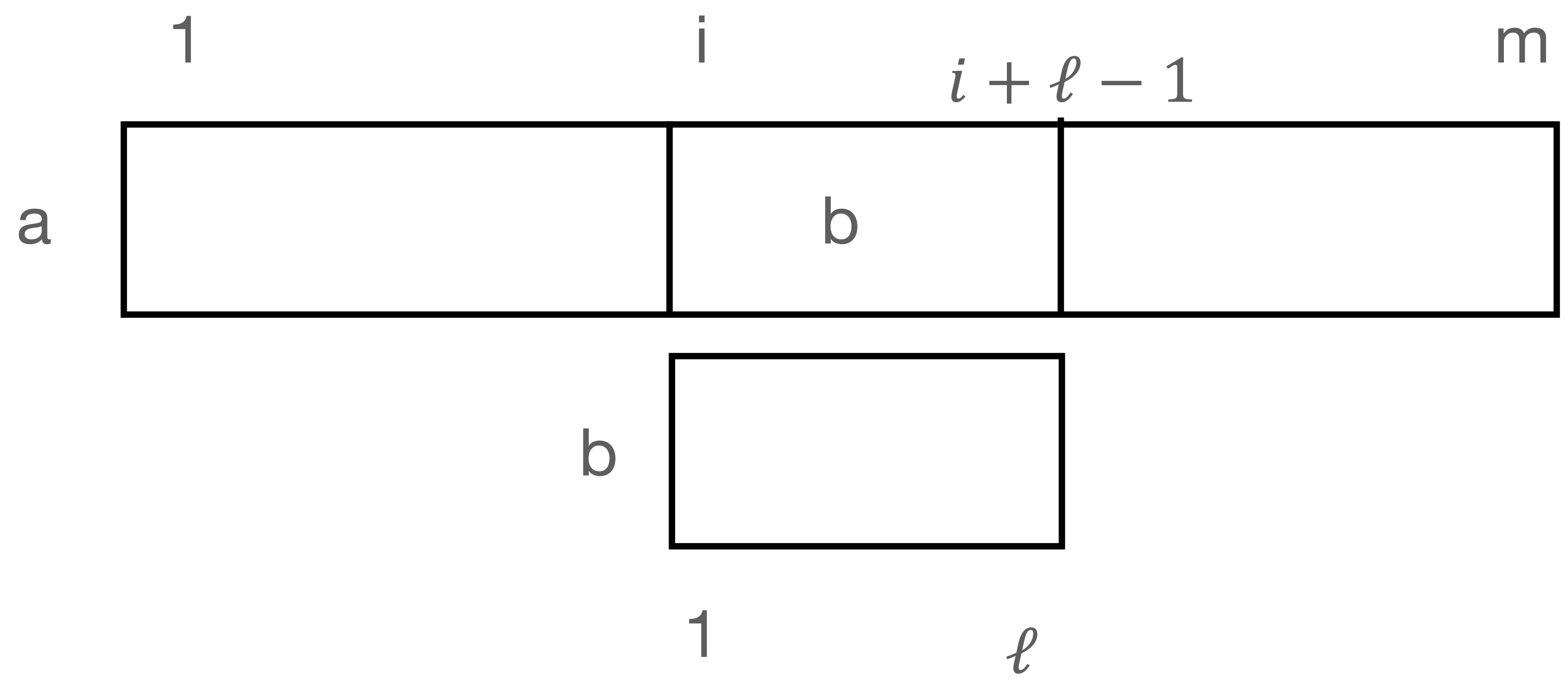
- inputs
  - $b = b[1:\ell] \in \Sigma^*$: pattern
  - $a = a[1:m] \in \Sigma^*$: string
- output: position i of first occurrence of b in a

$$out = \begin{cases} \min\{i \mid b[1:\ell] = a[i:i+\ell-1]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

cost:

number of:

- instructions of compiled code on MIPS
- C-instructions (if compiled into $O(1)$ MIPS instructions)
- lines of executed C-code (if compiled into $O(1)$ MIPS instructions)
- comparisons between symbols $\in \Sigma$ (if they amount to a constant fraction of the C instructions)



Naïve solution

```
test(i,b,a):
for k=1 to l
 if b[k] != a[i+k-1]
  break & return 0;
 return 1.
```

Cost ℓ

```
find(b,a):
i=1;
while ! test(i)  & i <= m-l+1
{i = i+1};
return (i <=m-l+1? i:0)
```

cost $O(\ell \cdot m)$ goal: cost $O(\ell + m)$

def: failure function

$$f(i)) = \begin{cases} \max\{s < i \mid b[1:s] = b[i-s+1:i]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

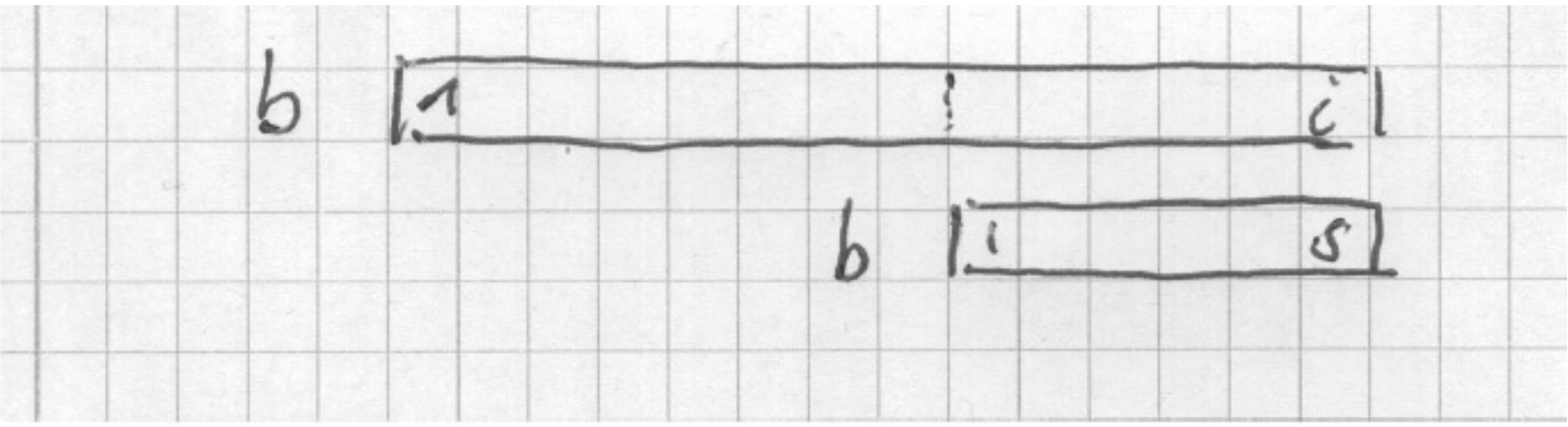
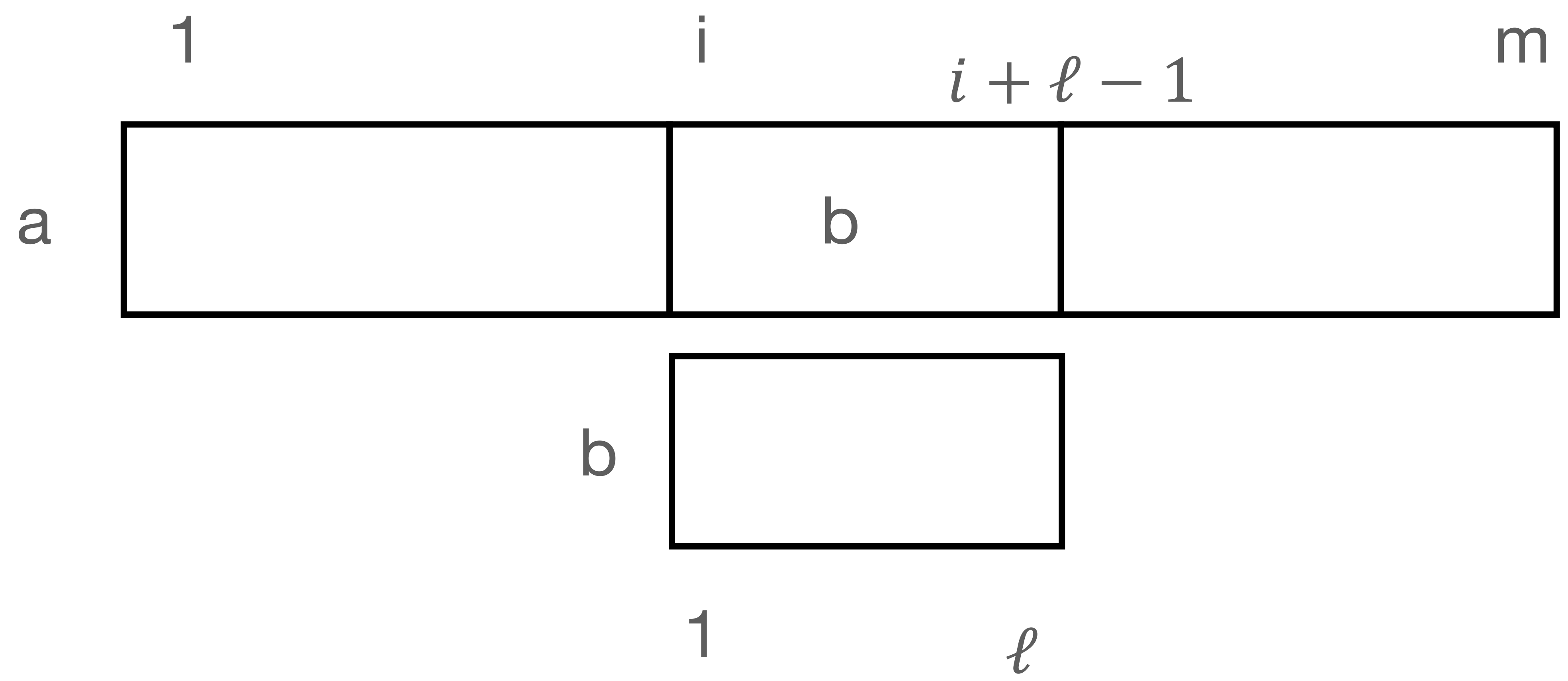


Figure 1: prefix $b[1:s]$ of b is suffix of $b[1:i]$

$b[1:s]$ are *suffixes* of $b[1:i]$

failure function



def: failure function

$$f(i)) = \begin{cases} \max\{s < i \mid b[1:s] = b[i-s+1:i]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

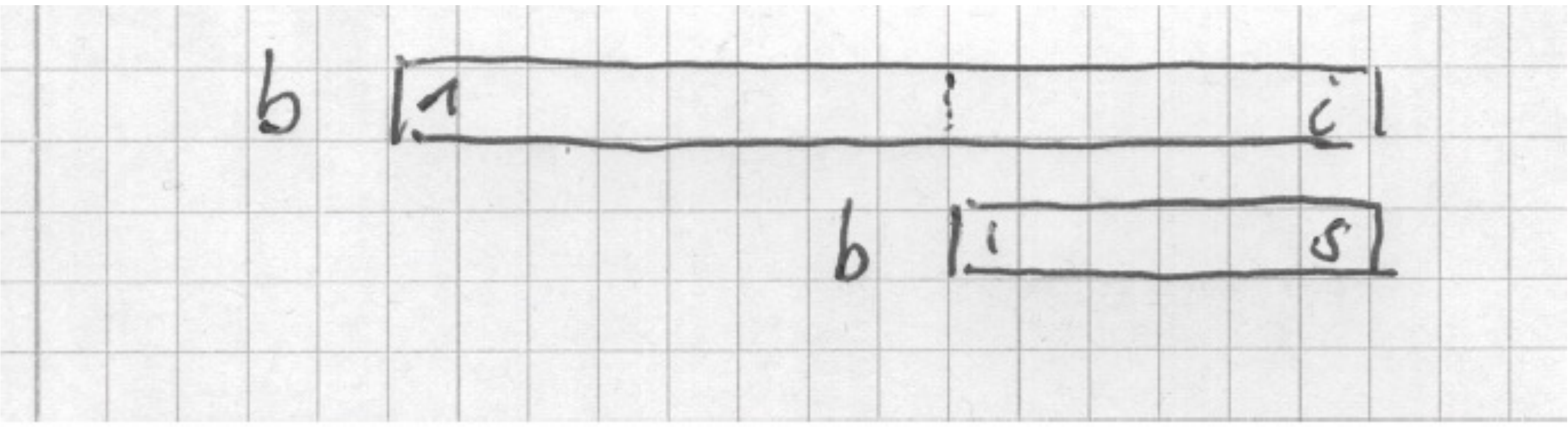


Figure 1: prefix $b[1:s]$ of b is suffix of $b[1:i]$

$b[1:s]$ are *suffixes* of $b[1:i]$

idea:

- assume match at positions i of b and j of a

$$b[1:i] = a[j-i+1:j]$$

  and mismatch at positions $i+1$ resp. $j+1$

$$b[i+1] \neq a[j+1]$$

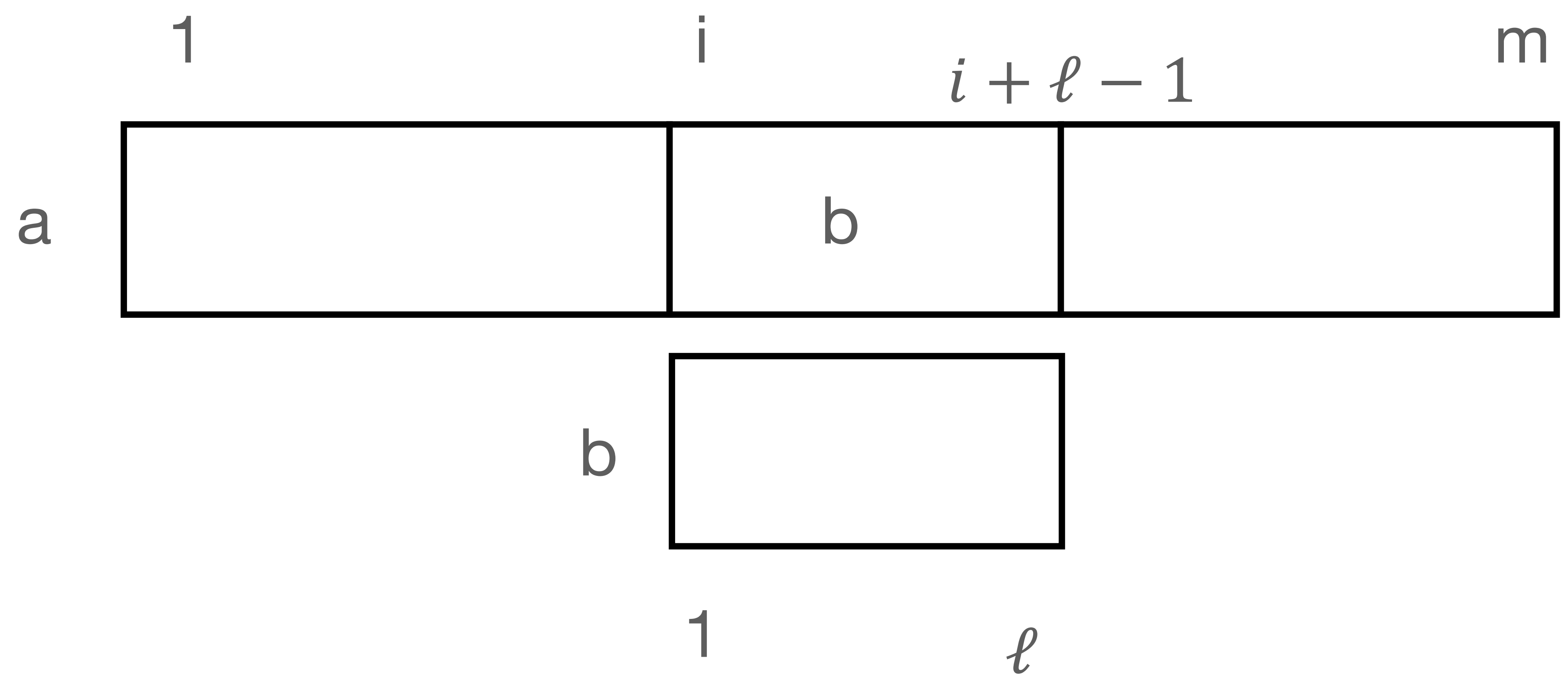
- then advance b such that

$$b[f(i)] = a[j]$$

  next test

$$b[f(i)+1] = a[j+1]$$

failure function



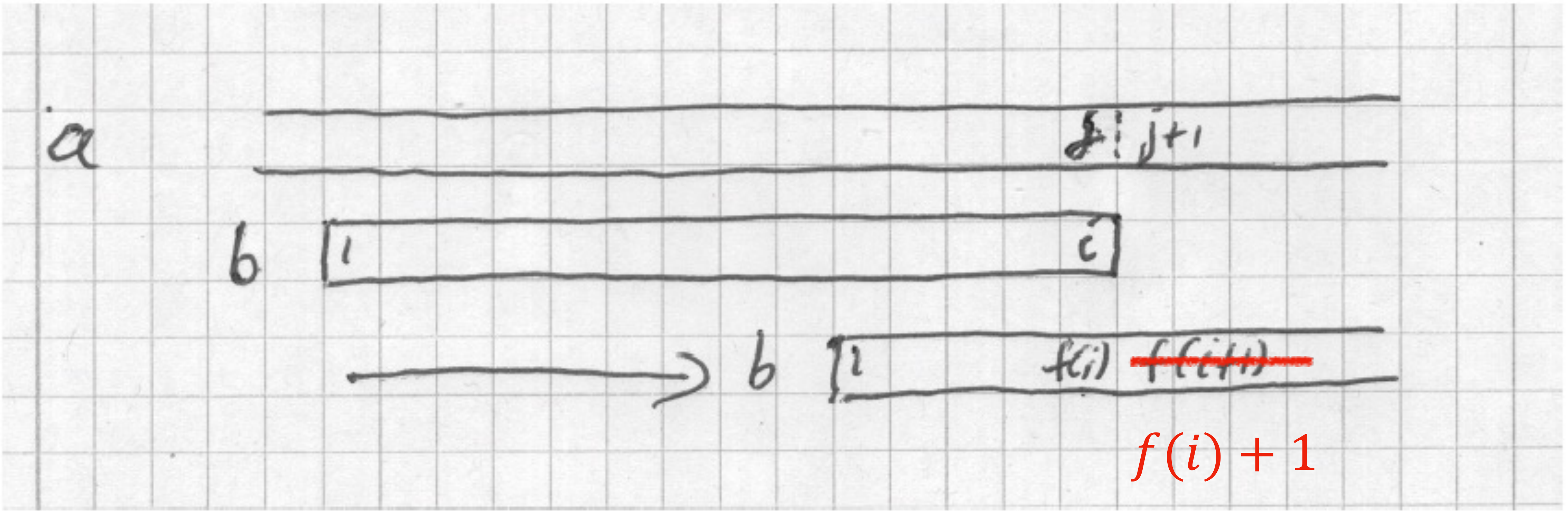


Figure 2: if test $b[i+1] = a[j+1]$ fails, pattern b is shifted right such that $b[f(i)]$ is below $a[j]$

def: failure function

$$f(i)) = \begin{cases} \max\{s < i \mid b[1:s] = b[i-s+1:i]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

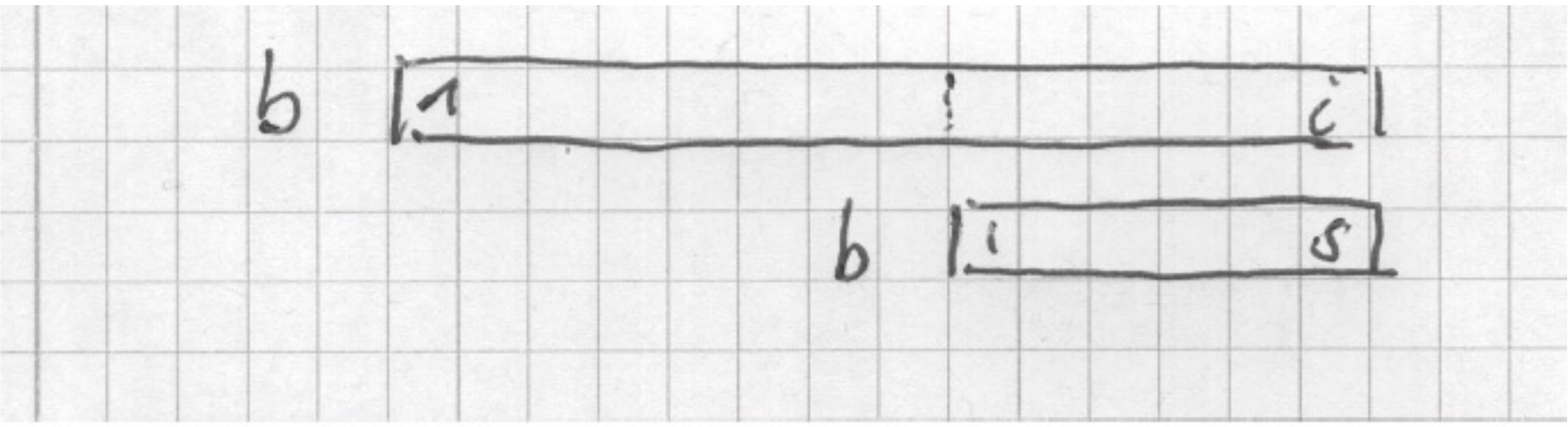


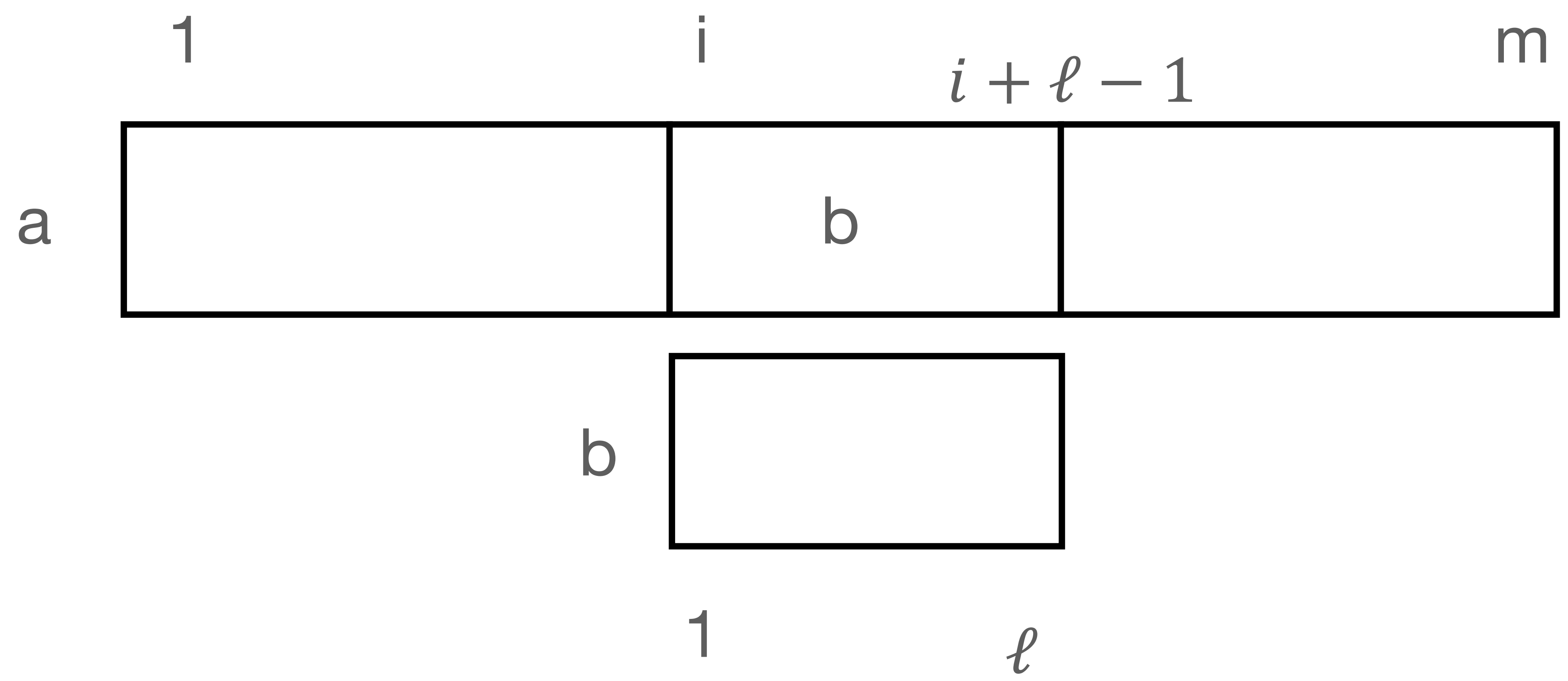
Figure 1: prefix $b[1:s]$ of b is suffix of $b[1:i]$

$b[1:s]$ are *suffixes* of $b[1:i]$

def: iterated failure function

$$\begin{aligned} f^0(i) &= i \\ f^{n+1}(i) &= f(f^n(i)) \end{aligned}$$

failure function



def: failure function

$$f(i)) = \begin{cases} \max\{s < i \mid b[1:s] = b[i-s+1:i]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

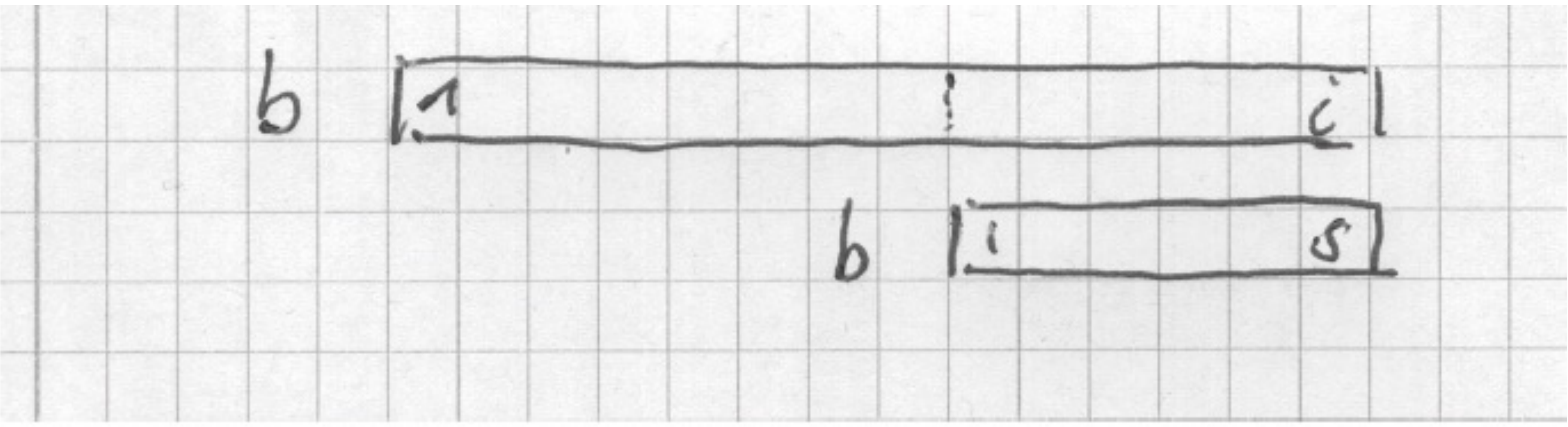


Figure 1: prefix $b[1:s]$ of b is suffix of $b[1:i]$

$b[1:s]$ are *suffixes* of $b[1:i]$

def: iterated failure function

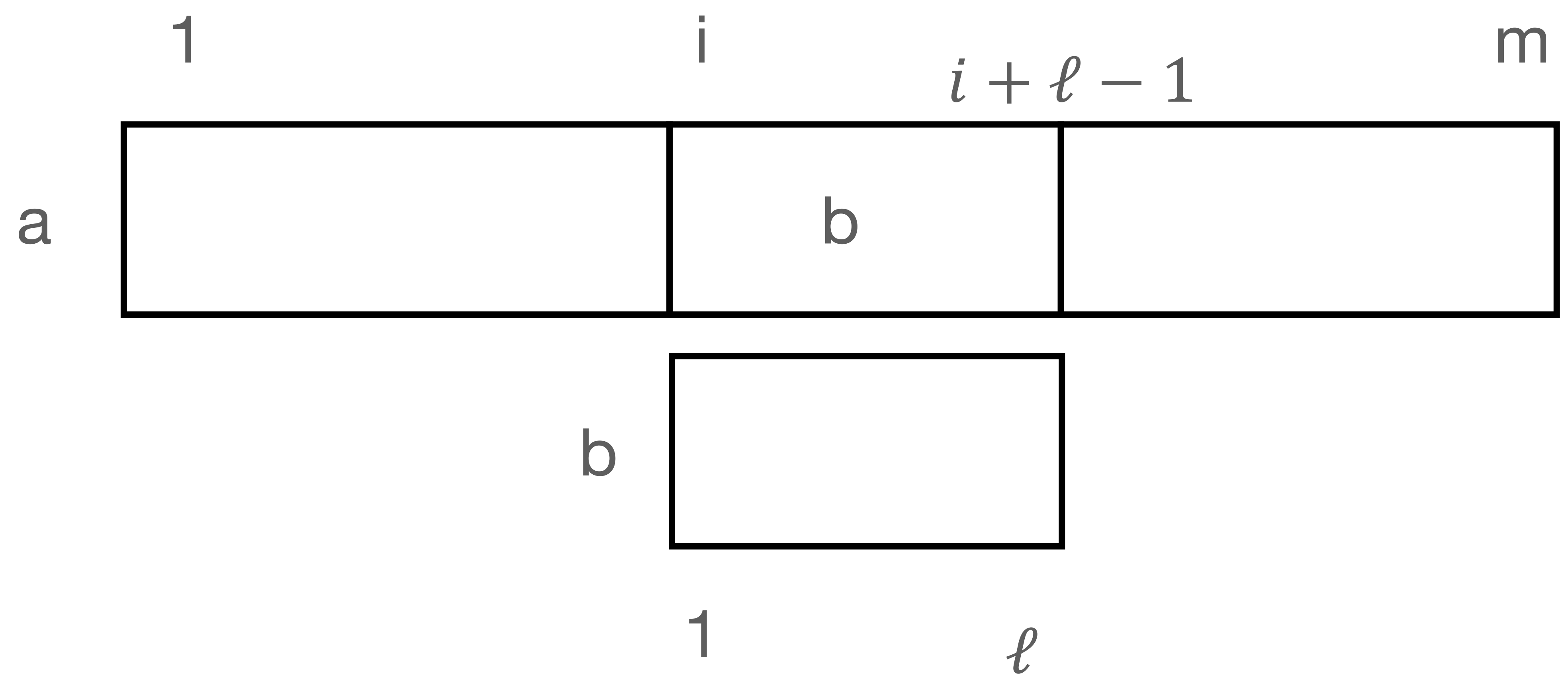
$$\begin{aligned} f^0(i) &= i \\ f^{n+1}(i) &= f(f^n(i)) \end{aligned}$$

All suffixes $b[1:s]$ of $b[1:i]$ can be found by the iterated failure function

Lemma 1.

$$\{b[1:s] \mid b[1:s] \textit{ suffix of } b[1:i]\} = \{b[1:f^n(i)] \mid n \geq 0\}$$

failure function



failure function

def: failure function

$$f(i)) = \begin{cases} \max\{s < i \mid b[1:s] = b[i-s+1:i]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

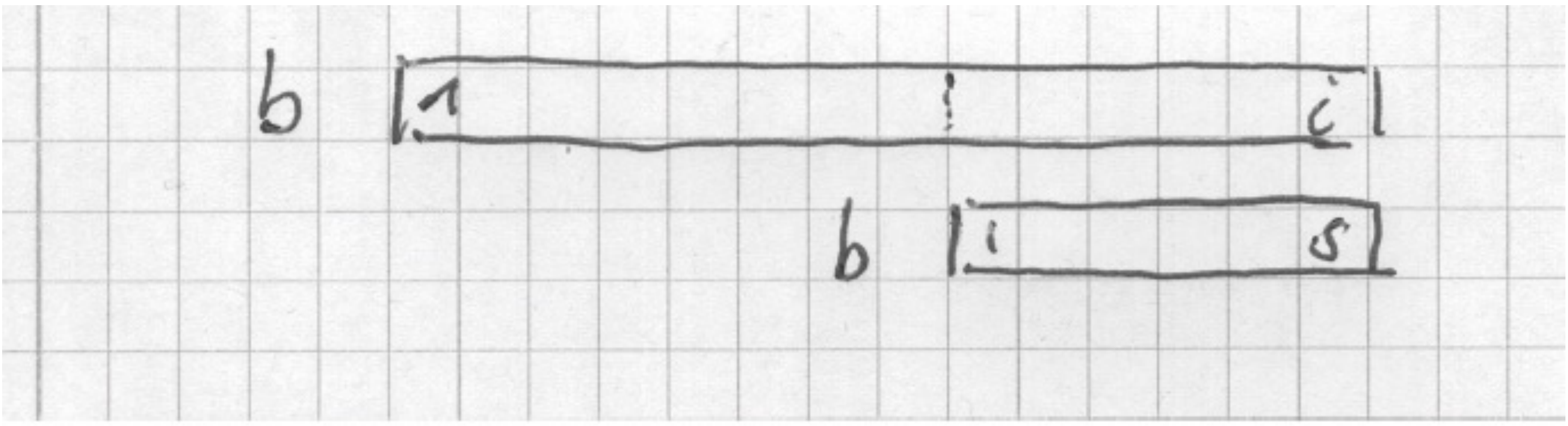


Figure 1: prefix $b[1:s]$ of b is suffix of $b[1:i]$

$b[1:s]$ are *suffixes* of $b[1:i]$

def: iterated failure function

$$\begin{aligned} f^0(i) &= i \\ f^{n+1}(i) &= f(f^n(i)) \end{aligned}$$

All suffixes $b[1:s]$ of $b[1:i]$ can be found by the iterated failure function

Lemma 1.

$$\{b[1:s] \mid b[1:s] \textit{ suffix of } b[1:i]\} = \{b[1:f^n(i)] \mid n \geq 0\}$$

- $\supseteq$: ' is suffix of' is transitive

failure function

def: failure function

$$f(i)) = \begin{cases} \max\{s < i \mid b[1:s] = b[i-s+1:i]\} & \text{if it exists} \\ 0 & \text{otherwise} \end{cases}$$

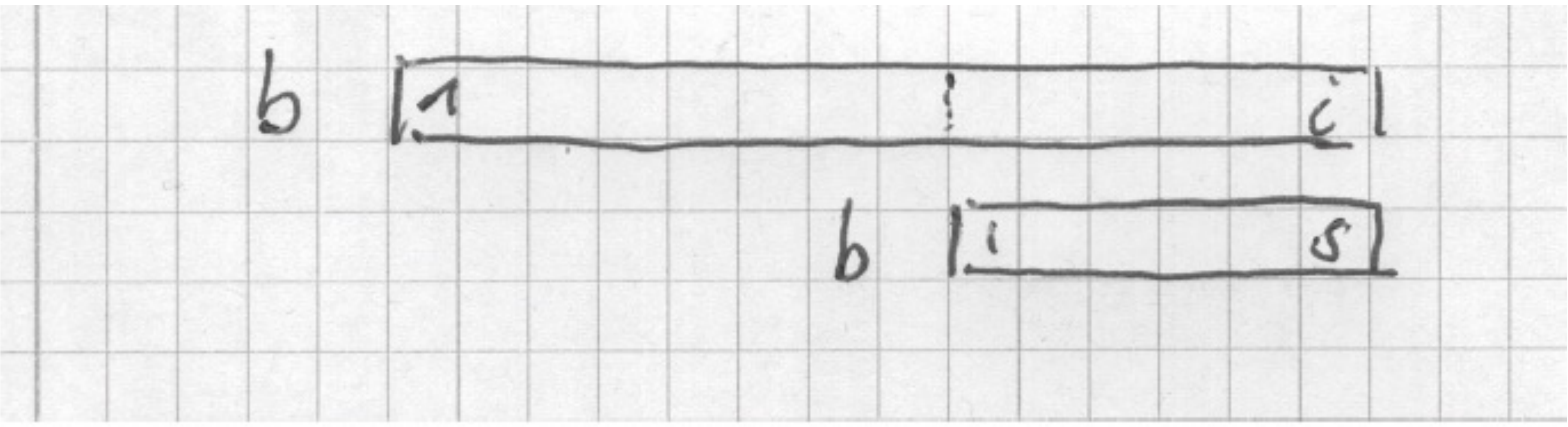


Figure 1: prefix $b[1:s]$ of b is suffix of $b[1:i]$

$b[1:s]$ are *suffixes* of $b[1:i]$

def: iterated failure function

$$\begin{aligned} f^0(i) &= i \\ f^{n+1}(i) &= f(f^n(i)) \end{aligned}$$

All suffixes $b[1:s]$ of $b[1:i]$ can be found by the iterated failure function

Lemma 1.

$$\{b[1:s] \mid b[1:s] \textit{ suffix of } b[1:i]\} = \{b[1:f^n(i)] \mid n \geq 0\}$$

- $\supseteq$: ' is suffix of' is transitive
- $\subseteq$:

$$b[1:s] \text{ suffix of } b[1:i]\ ,\ f^n(i) < s \leq f^{n-1}(i)$$

$$s = f^{n-1}(i) \text{ done}$$

$$s < f^{n-1}(i) \rightarrow b[1:f^n(i)] \text{ not longest suffix of } b[1:f^{n-1}(i)] \ldots$$

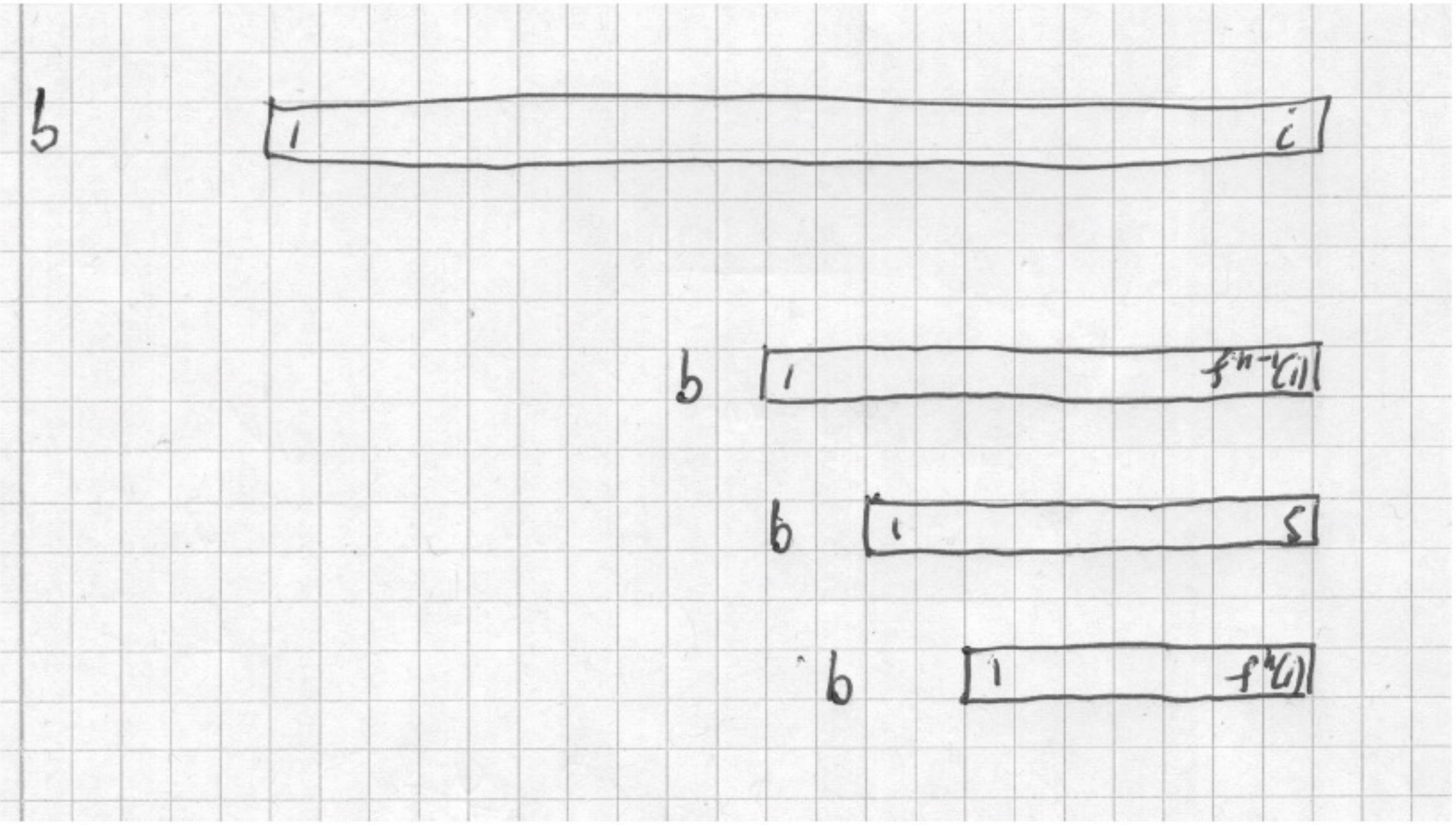


Figure 3: $s > f(f^{n-1}(i))$ contradicts the definition of failure function f

computing the failure function

```
f(1) = 0;
for j=2 to l do
  {i=f(j-1);
  while b[j] != b[i+1] & i>0 {i = f(i)};
  if b[j] != b[i+1] & i=0 {f(j)=0}
  else {f(j) = i+1}
  }
```

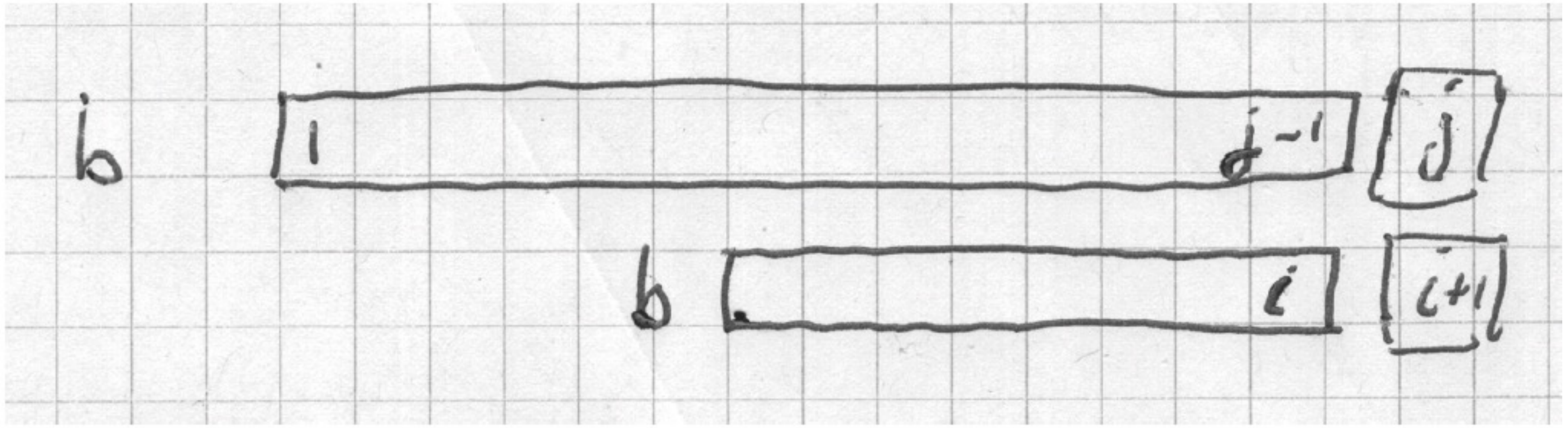


Figure 4: b is shifted to $i = f^n(j-1)$ for some n. Tests are between $b[i+1]$ and $b[j]$.

computing the failure function

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.   while b[j] != b[i+1] & i>0 {i = f(i)};
5.   if b[j] != b[i+1] & i=0 {f(j)=0}
6.   else {f(j) = i+1}
    }
```

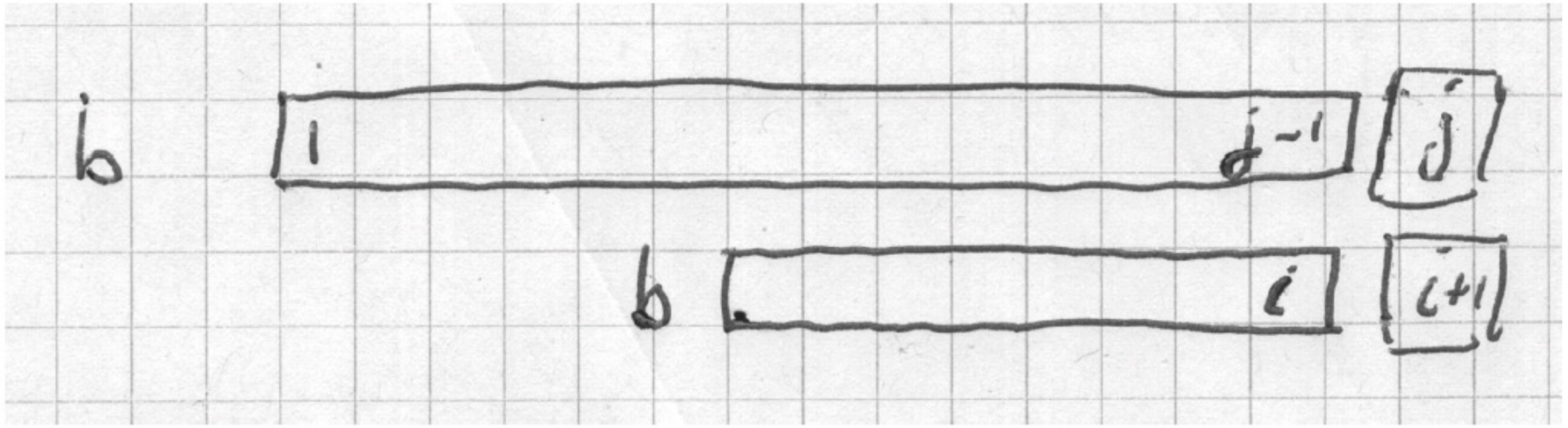


Figure 4: b is shifted to $i = f^n(j-1)$ for some n. Tests are between $b[i+1]$ and $b[j]$.

correctness: line 4 shifts b repeatedly by iterated error function. By lemma 1 this generates all relevant suffixes. The longest is found first.

computing the failure function

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

run time

Lines 1, 2, 3, 5, 6: $O(1)$

ℓ passes of loop: $O(\ell)$

while loop: each pass decrements i

computing the failure function

```
1. f(1)  = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

i initialised with $i = f(2-1) = f(1) = 0$

using primed notation (without: before, with after) incrementing i only once per while loop by

```
6. f(j)=i+1
2. for... (j'=j+1)
3. i' = f(j'-1)= f(j) = i+1
```

run time

Lines 1, 2, 3, 5, 6: $O(1)$

ℓ passes of loop: $O(\ell)$

while loop: each pass decrements i

computing the failure function

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

i initialised with $i = f(2-1) = f(1) = 0$

using primed notation (without: before, with after) incrementing i only once per while loop by

```
6. f(j)=i+1
2. for... (j'=j+1)
3. i' = f(j'-1)= f(j) = i+1
```

run time

Lines 1, 2, 3, 5, 6: $O(1)$

ℓ passes of loop: $O(\ell)$

while loop: each pass decrements i

computing the failure function

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

i initialised with $i = f(2-1) = f(1) = 0$

using primed notation (without: before, with after) incrementing i only once per while loop by

```
6. f(j)=i+1
2. for... (j'=j+1)
3. i' = f(j'-1)= f(j) = i+1
```

run time

Lines 1, 2, 3, 5, 6: $O(1)$

ℓ passes of loop: $O(\ell)$

while loop: each pass decrements i

computing the failure function

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

i initialised with $i = f(2-1) = f(1) = 0$

using primed notation (without: before, with after) incrementing i only once per while loop by

```
6. f(j)=i+1
2. for... (j'=j+1)
3. i' = f(j'-1)= f(j) = i+1
```

run time

Lines 1, 2, 3, 5, 6: $O(1)$

ℓ passes of loop: $O(\ell)$

while loop: each pass decrements i

$O(\ell)$ passes of while loop

$\boxed{\text{run time} = O(\ell)}$

Since i is incremented once every loop run (in total ℓ-times) so while loop can not decrement it more then ℓ-times in total (since ℓ must stay positive)

alternative estimate of run time using potential method

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

Example:

$c_1 \rightarrow line(1) = 1.$

$c_2 \rightarrow line(2) = 2.$

$c_3 \quad line(3) = 3.$

entered while loop.

$line(4) = 4. \leftarrow c_4$

$line(5) = 4. \leftarrow c_5$

$line(6) = 4. \leftarrow c_6$

$\vdots$

op_5 = exec. 4.

- op_k: executes line $line(k)$ in the C code for $1 \leq k \leq t$
- $c_k = 1$: cost of executing a line (good enough; up to a constant factor)
- potential function

$$\Phi_k = \text{value of } i \text{ after execution of } op_k$$

$$\Phi_0 = 0\,, \Phi_k \geq 0 \quad \rightarrow \quad \sum_{k=1}^{t} c_k \leq \sum_{k=1}^{t} \hat{c}_k$$

alternative estimate of run time using potential method

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.   while b[j] != b[i+1] & i>0 {i = f(i)};
5.   if b[j] != b[i+1] & i=0 {f(j)=0}
6.   else {f(j) = i+1}
    }
```

- op_k: executes line $line(k)$ in the C code for $1 \leq k \leq t$
- $c_k = 1$: cost of executing a line (good enough; up to a constant factor)
- potential function

$$\Phi_k = \text{value of } i \text{ after execution of } op_k$$

$$\Phi_0 = 0\ , \Phi_k \geq 0 \quad \rightarrow \quad \sum_{k=1}^{t} c_k \leq \sum_{k=1}^{t} \hat{c}_k$$

potential difference

$$\Phi_k - \Phi_{k-1} = \begin{cases} 1 & line(k) = 3 \\ 0 & line(k) \notin \{3,4\} \end{cases}$$

$$\Phi_k - \Phi_{k-1} \leq -1 \text{ if } line(k) = 4 \quad \text{(while loop)}$$

i decreased if
line 4. executed

alternative estimate of run time using potential method

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

- op_k: executes line $line(k)$ in the C code for $1 \leq k \leq t$
- $c_k = 1$: cost of executing a line (good enough; up to a constant factor)
- potential function

$$\Phi_k = \text{value of } i \text{ after execution of } op_k$$

$$\Phi_0 = 0\,, \Phi_k \geq 0 \quad \rightarrow \quad \sum_{k=1}^{t} c_k \leq \sum_{k=1}^{t} \hat{c}_k$$

potential difference

$$\Phi_k - \Phi_{k-1} = \begin{cases} 1 & line(k) = 3 \\ 0 & line(k) \notin \{3,4\} \end{cases}$$

$$\Phi_k - \Phi_{k-1} \leq -1 \text{ if } line(k) = 4 \quad \text{(while loop)}$$

amortized cost

$$\hat{c}_k = \begin{cases} 2 & line(k) = 3 \\ 1 & line(k) \notin \{3,4\} \end{cases}$$

$$\hat{c}_k \leq 0 \text{ if } line(k) = 4 \quad \text{(while loop)}$$

alternative estimate of run time using potential method

```
1. f(1) = 0;
2. for j=2 to l do
    {3.  i=f(j-1);
4.  while b[j] != b[i+1] & i>0 {i = f(i)};
5.  if b[j] != b[i+1] & i=0 {f(j)=0}
6.  else {f(j) = i+1}
    }
```

- op_k: executes line $line(k)$ in the C code for $1 \leq k \leq t$
- $c_k = 1$: cost of executing a line (good enough; up to a constant factor)
- potential function

$$\Phi_k = \text{value of } i \text{ after execution of } op_k$$

$$\Phi_0 = 0\,, \Phi_k \geq 0 \quad \rightarrow \quad \sum_{k=1}^{t} c_k \leq \sum_{k=1}^{t} \hat{c}_k$$

potential difference

$$\Phi_k - \Phi_{k-1} = \begin{cases} 1 & line(k) = 3 \\ 0 & line(k) \notin \{3,4\} \end{cases}$$

$$\Phi_k - \Phi_{k-1} \leq -1 \text{ if } line(k) = 4 \quad \text{(while loop)}$$

amortized cost

$$\hat{c}_k = \begin{cases} 2 & line(k) = 3 \\ 1 & line(k) \notin \{3,4\} \end{cases}$$

$$\hat{c}_k \leq 0 \text{ if } line(k) = 4 \quad \text{(while loop)}$$

$$\begin{aligned} \sum_{k=1}^{t} \hat{c}_k &\leq \sum_{line(k)\neq 4} \hat{c}_k \\ &\leq \hat{c}_1 + \ell \cdot (\overbrace{\hat{c}_2 + \hat{c}_3}^{\text{these executed max } \ell \text{ times}} + \underbrace{\max\{\hat{c}_5, \hat{c}_6\}}_{\text{one of these executed in each of } \ell \text{ runs.}}) \\ &= 1 + \ell \cdot (1 + 2 + 1) \\ &= O(\ell) \end{aligned}$$

finding a substring

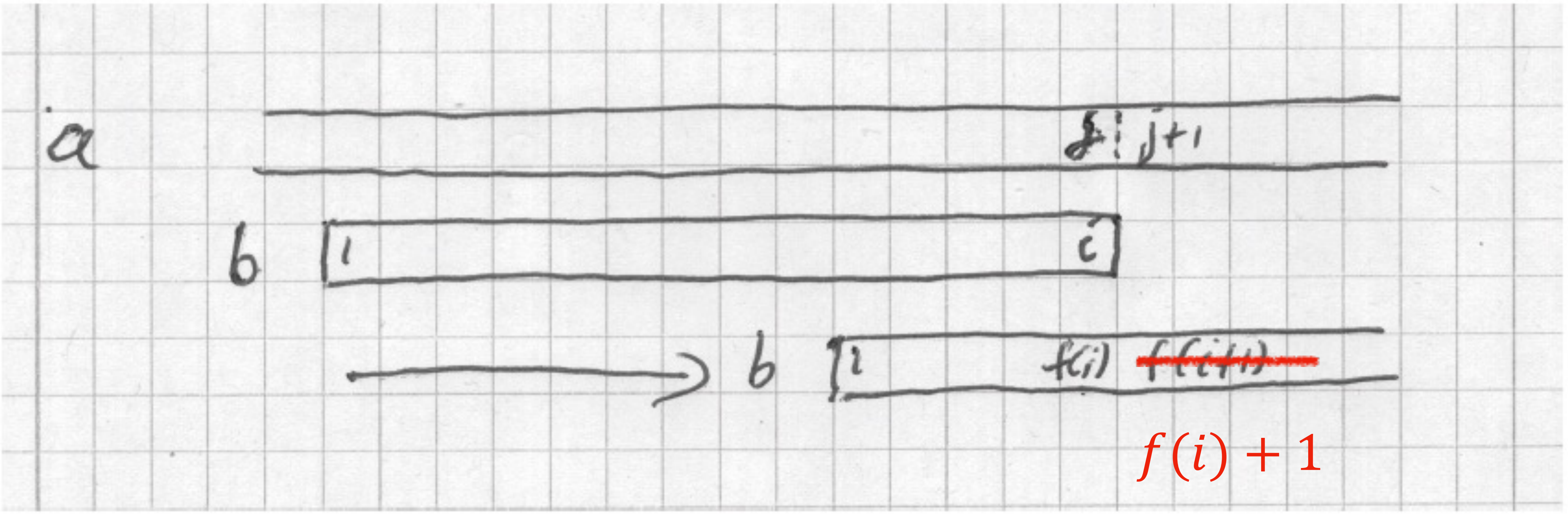


Figure 5: if test $b[i+1] = a[j+1]$ fails, pattern b is shifted right such that $b[f(i)]$ is below $a[j]$.

finding a substring

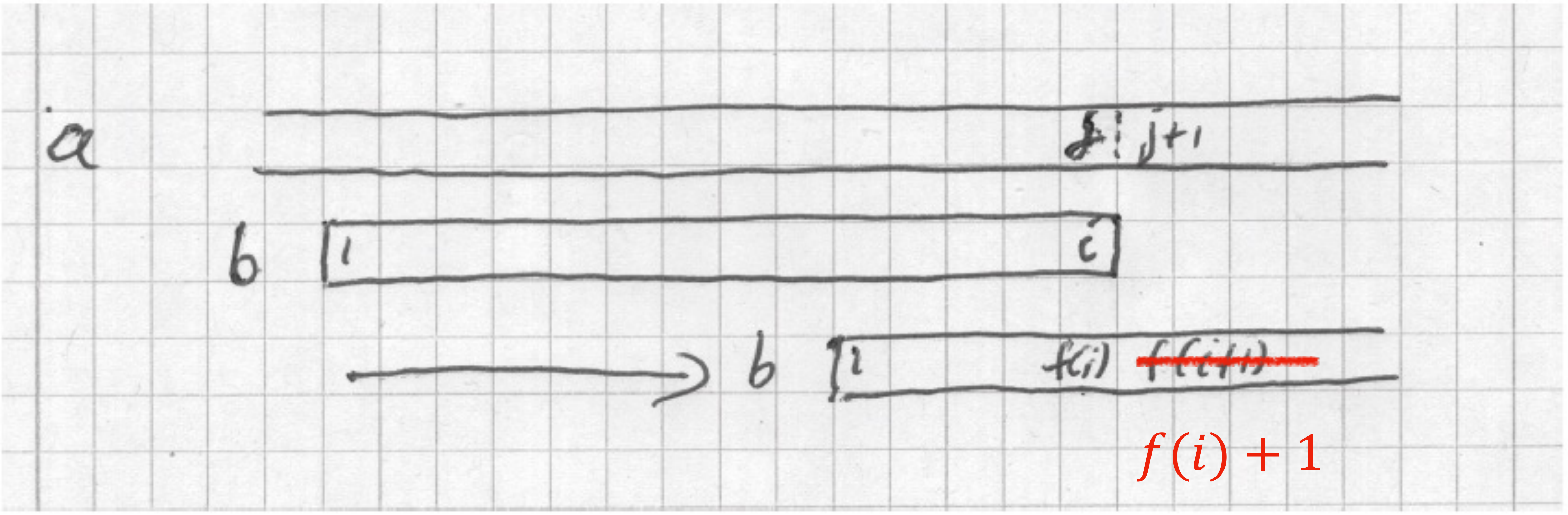


Figure 5: if test $b[i+1] = a[j+1]$ fails, pattern b is shifted right such that $b[f(i)]$ is below $a[j]$.

With function f known:

- implement pattern matching using idea of figure 6 : attention after $i = f(i)$ the next comparison is $a[j+1] = b[i+1]$?
- show that the run time is $O(m)$

exercise

spec of finite automata

$A = (Z, z_0, I, \delta)$

hardware lab slide set OS support-12

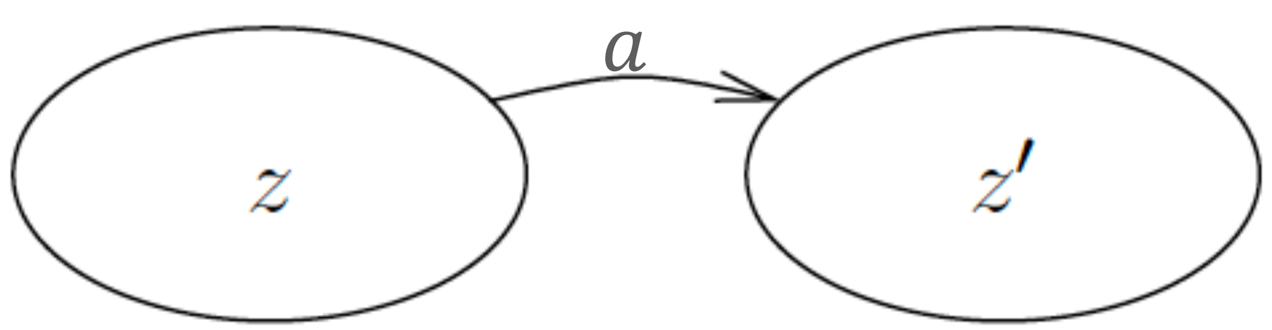
- Z finite set of states
- $z_0 \in Z$ initial state
- I input alphabet
- $\delta: Z \times I \to Z$ transition function

double cycle

$Z_A \subseteq Z$ set of accepting states.

$\delta(z, a) = z'$

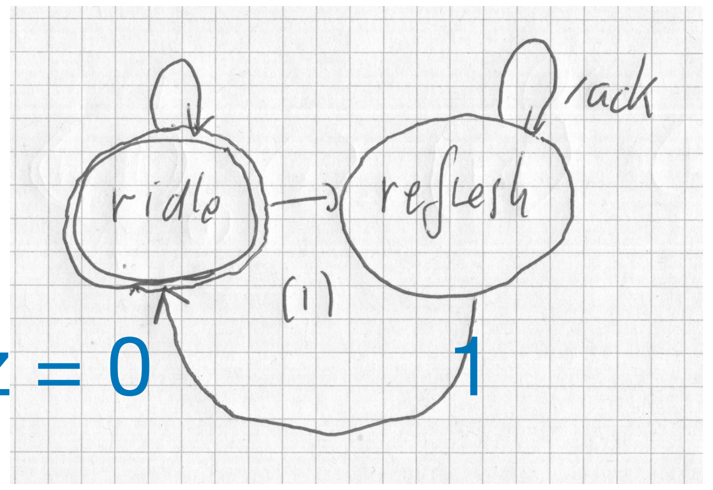
if A reads a in states z,
then it goes to state z'



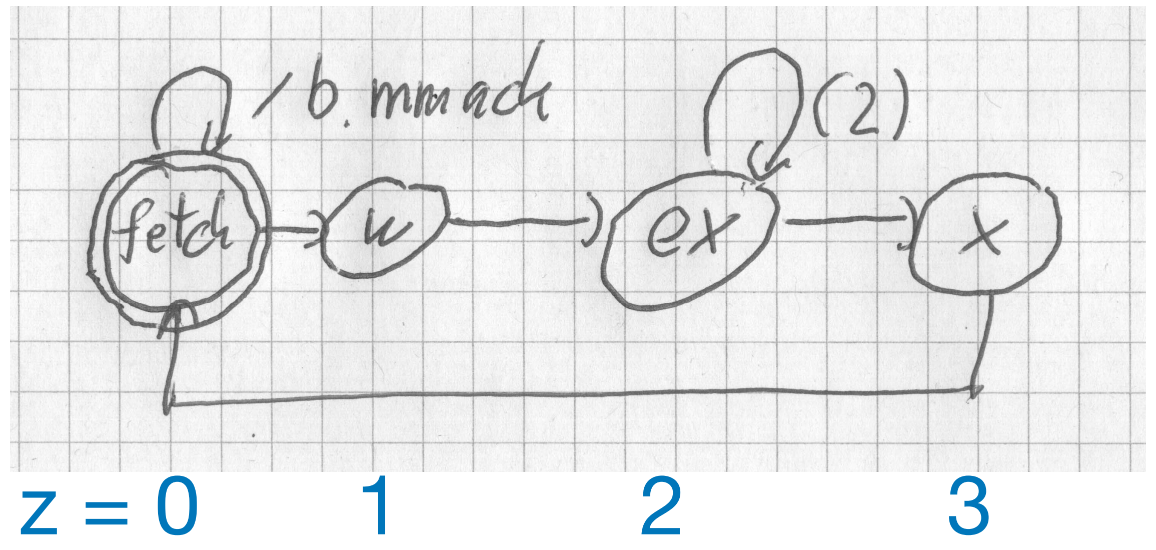
visualisation

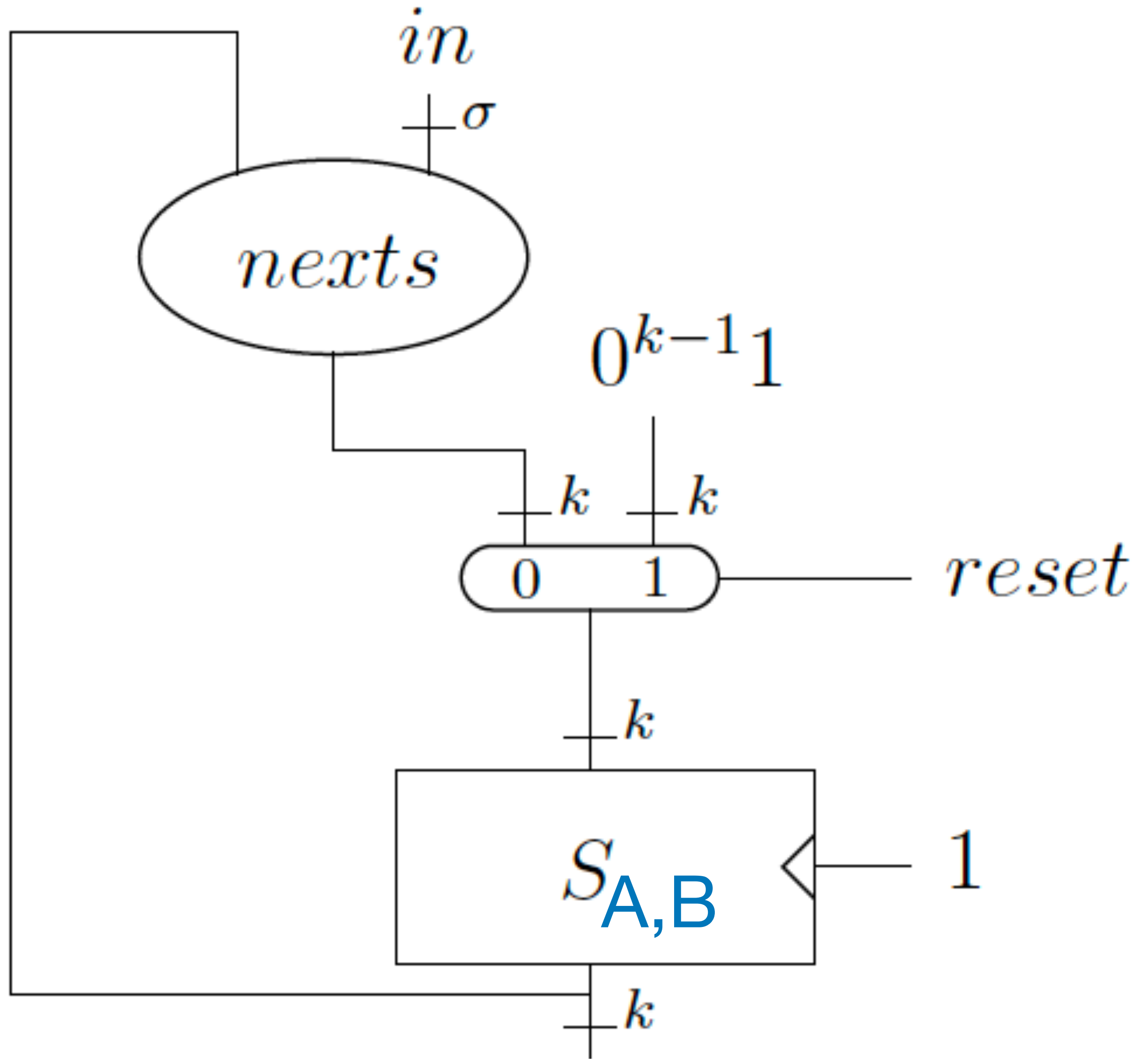
automata

A



B





$\delta_{z,z'}(in) = 1$ ↔ $\delta(z, in) = z'$

$$\boxed{nexts[z'](in) = \bigvee_{z} S[z] \wedge \delta_{z,z'}(in)}$$

failure automaton for pattern *b*

- states $S = 0, \dots, \ell + 1$
- input alphabet Σ
- start state 0
- transition function

Corresponding Algorithm is Different from slide 26

$$\delta(i,x) = \begin{cases} i+1 & x = b_x \\ f(i)+1 & \text{otherwise} \end{cases}$$

$f(b_1 \dots b_i x)^{(i+1)}$, for $x \in \Sigma$

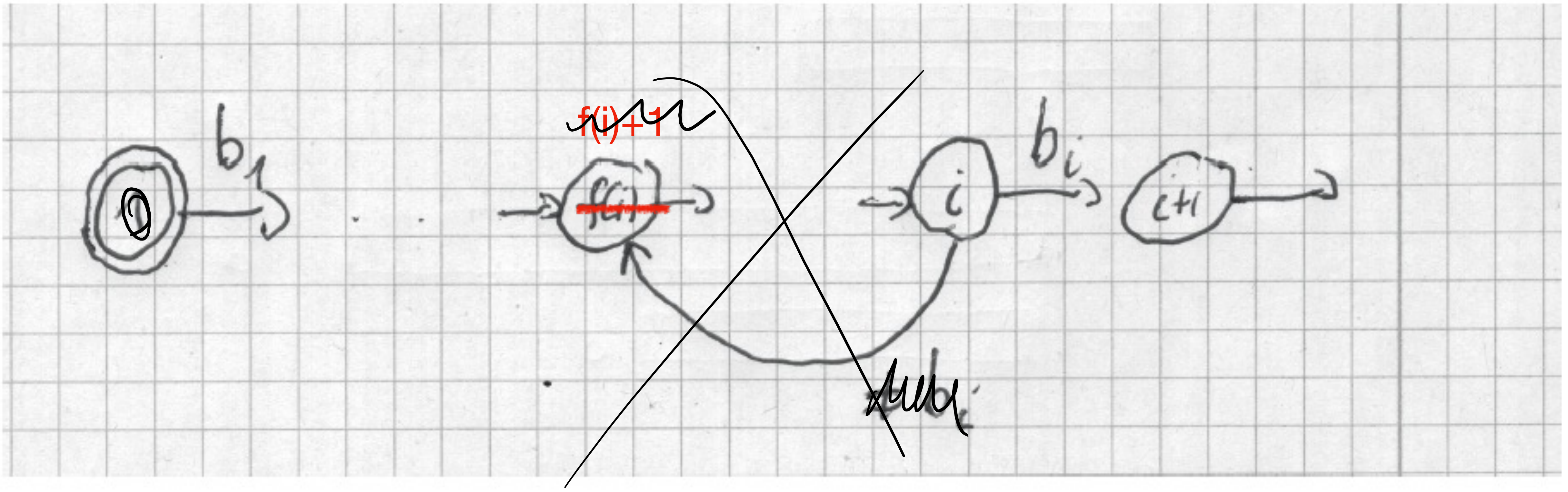
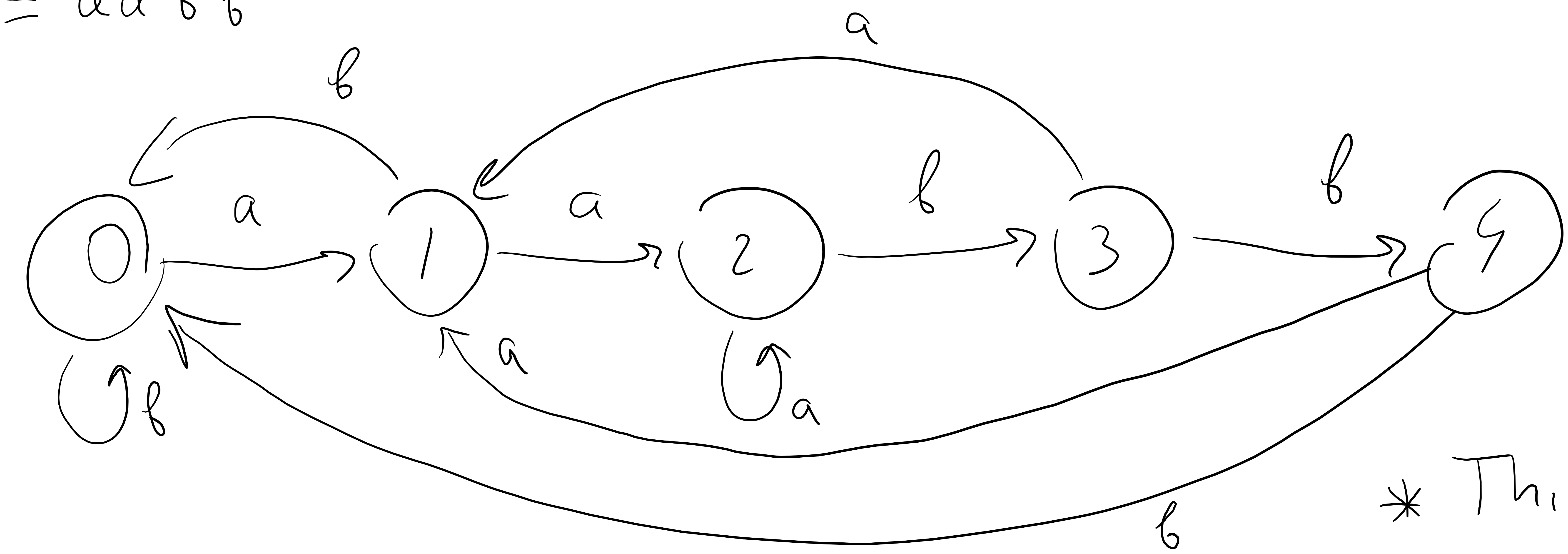


Figure 6: to test if b is a substring of a run this automaton with input a. Decide, that b is a substring of a, if state $\ell + 1$ is reached

string = aabb



	a	b
0	1	0
1	2	0
2	2	3
3	1	4
4	1	0

Ex: $f_{aaa}(3) = 2$

$f_{aaba}(4) = 1$

$f_{aabba}(5) = 1$

$f_{aabbb}(5) = 0$

* This DFA accepts all strings ending in aabb