String Matching

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\mathbf{p}, v) = f_\theta(s)$. The vector of move probabilities $\mathbf{p}$ represents the probability of selecting each move (including pass), $p_a = Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network [12] into a single architecture. The neural network consists of many residual blocks [4] of convolutional layers [16,17] with batch normalisation [18] and rectifier n… linea… s… ds).

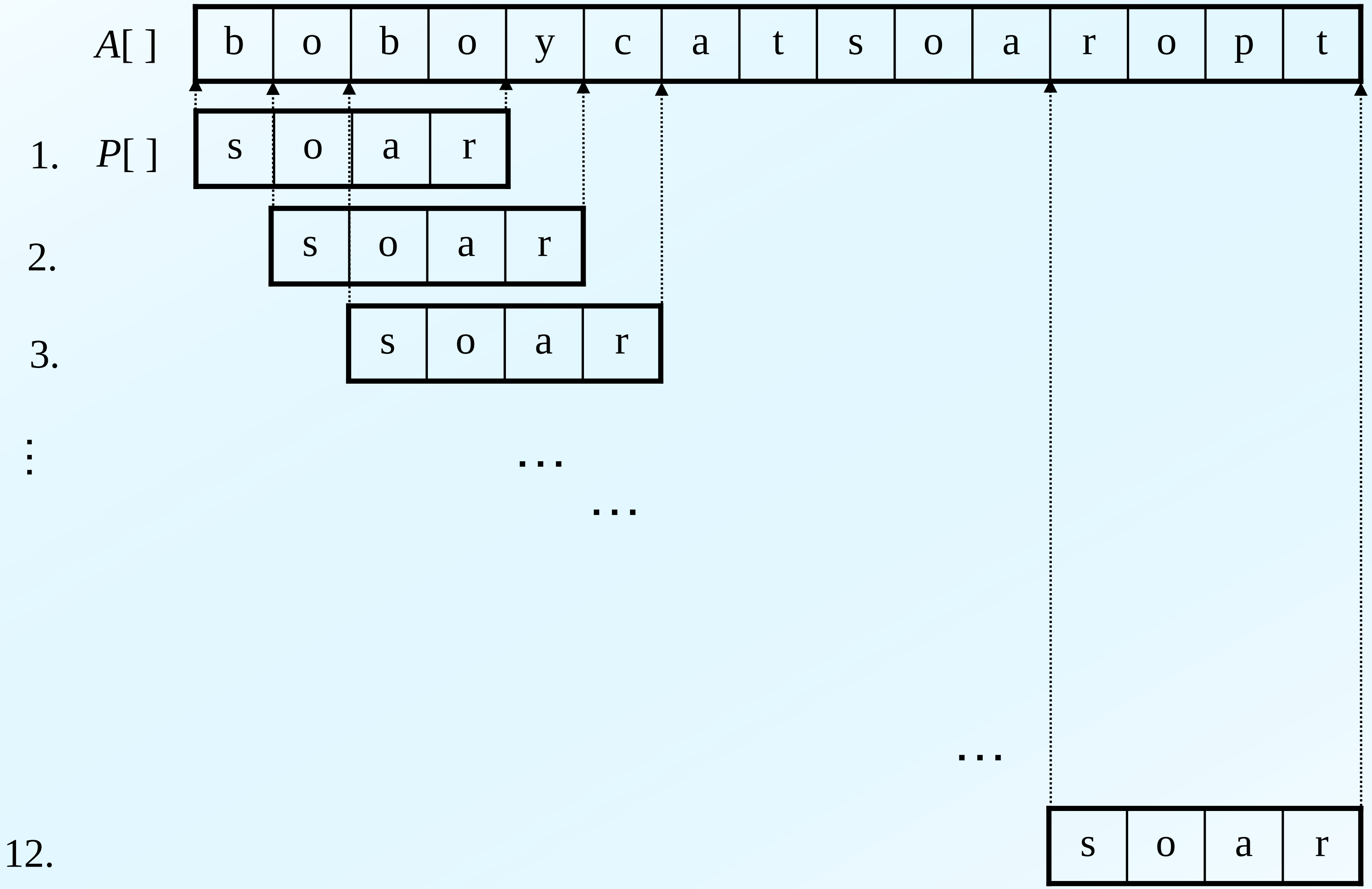
The neural network in *AlphaGo Z*… rained from games of self-play by a novel reinforcement learning algorithm. In each position s, an MCTS search is executed, guided by the neural network f_θ. The MCTS search outputs probabilities $\boldsymbol{\pi}$ of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities $\mathbf{p}$ of the neural network $f_\theta(s)$; MCTS may therefore be viewed as a powerful *policy improvement* operator [20,21]. Self-play with search – using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value – may be viewed as a powerful *policy evaluation* operator. The main idea of our reinforcement learning algorithm is to use these search operators repeatedly in a policy iteration procedure [22,23]: the neural network's parameters are updated to make the move probabilities and value $(\mathbf{p}, v) = f_\theta(s)$ more closely match the improved search probabilities and self-play winner $(\boldsymbol{\pi}, z)$; these new parameters are used in the next iteration of self-play to make the search even stronger. Figure 1 illustrates the self-play training pipeline.

The Monte-Carlo tree search uses the neural network f_θ to guide its simulations (see Figure 2). Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximise an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$ [12,24], until a leaf node s' is encountered. This leaf position is expanded and evaluated just

Given Condition

- Input
  - $A[1 \ldots n]$: text string
  - $P[1 \ldots m]$: pattern string
  - $m << n$
- Objective
  - Want to check to see whether $A[1 \ldots n]$ contains $P[1 \ldots m]$
  - Return all occurrences of $P[1 \ldots m]$ or just true/false

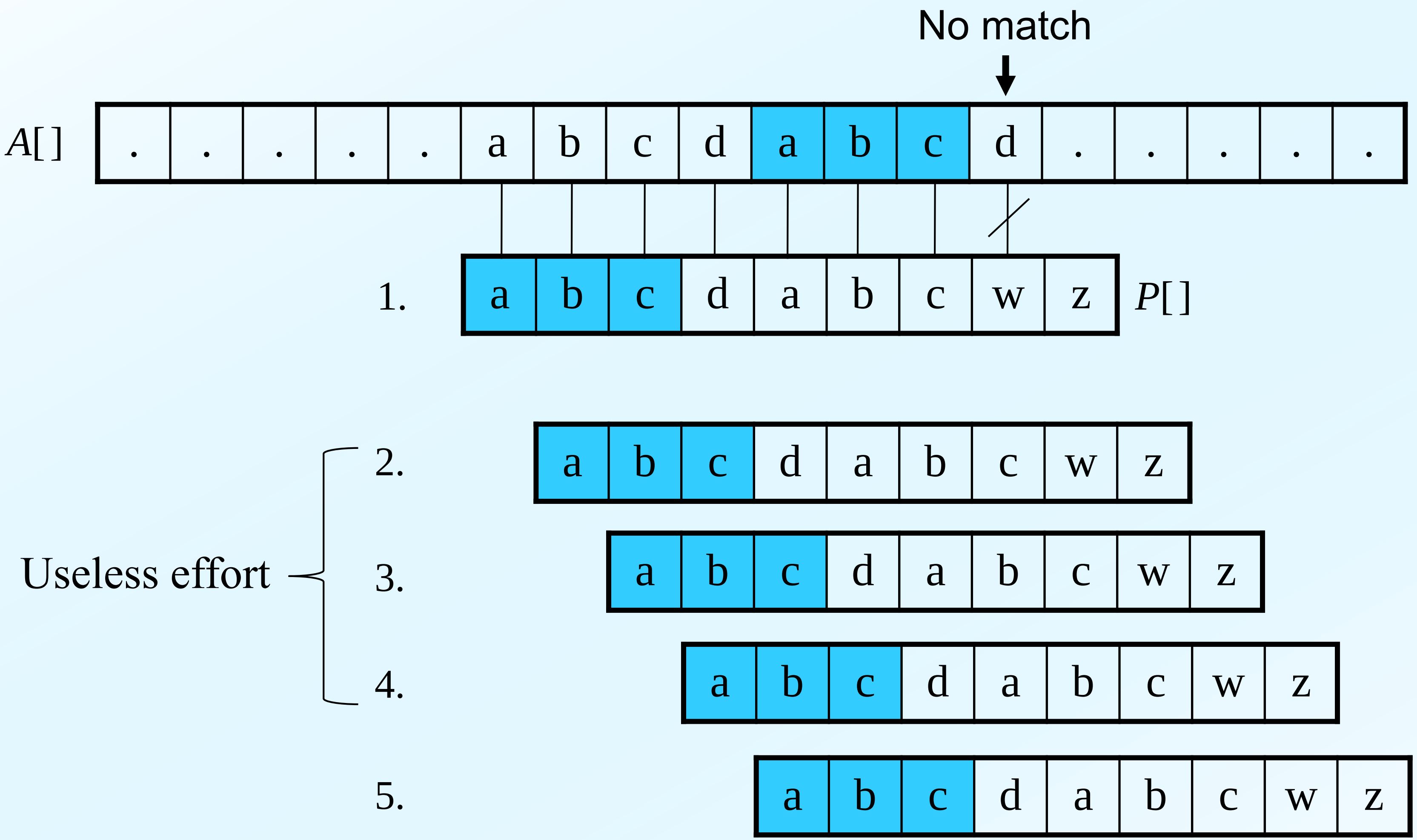
Naïve Matching



```
naiveMatching(A[ ], P[ ]):
    ▷ n: length of text array A[ ], m: length of pattern array P[ ]
    for i ← 1 to n-m+1
        if (P[1…m] = A[i…i+m-1])
            Report successful matching at A[i…]
```

✓ Running time: $O(mn)$

Inefficiency of Naïve Matching



Matching with Automata

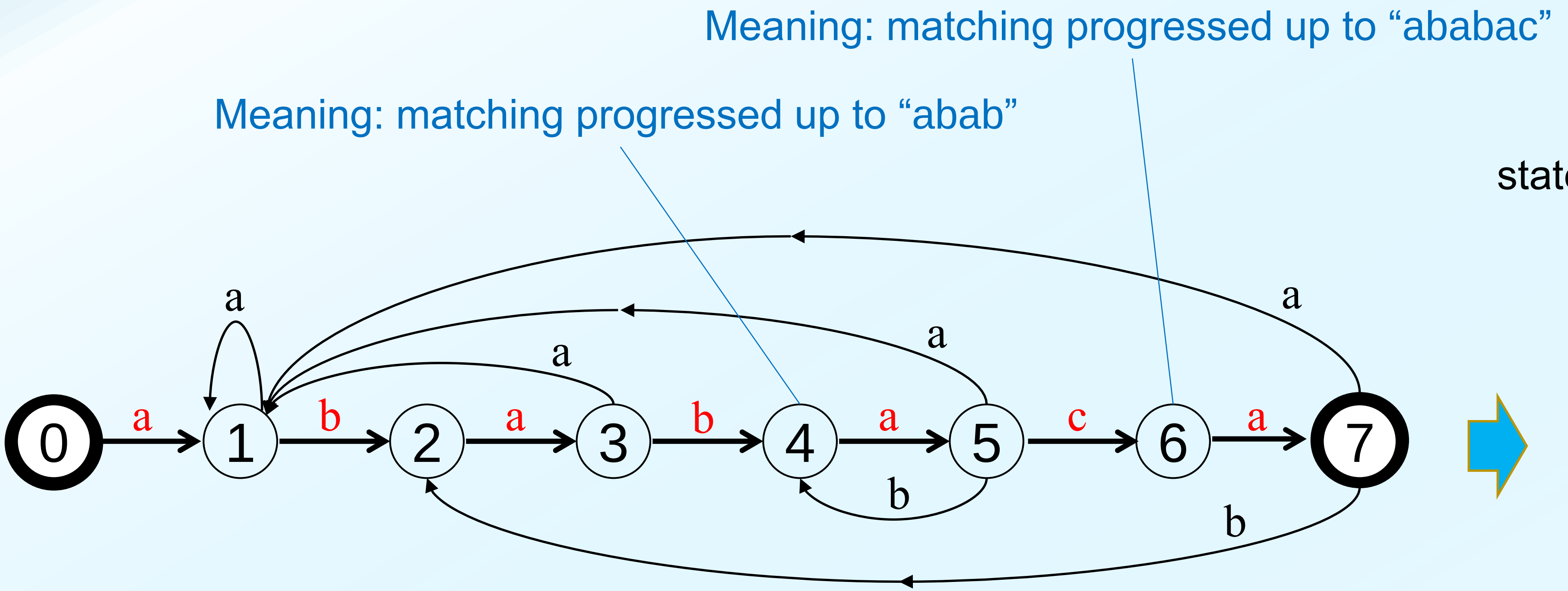
- Automata
  - Represents the process of problem solving by state transitions
  - An automaton: (Q, q_0, F, $\sum$, δ)
    - Q : set of states
    - q_0 : starting state
    - F : set of target states(one or more states)
    - $\sum$: set of input alphabets
    - δ : state-transition function
- Intuitive representation
  - A node: a progression state
  - An edge: progression by an input alphabet

Originally, automaton : automata

single plural

But, used interchangeably

An Automata Checking “ababaca”



S: dactababaababacabababacaagbk…

states \ Input alphabets	a	b	c	All others
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

Transition Function Table

Generating Automata

FA-Generater(P[ ], $\sum$):

▷ $P[1\ldots m]$: pattern

    for $q \leftarrow 0$ **to** m

        for each $a \in \sum$

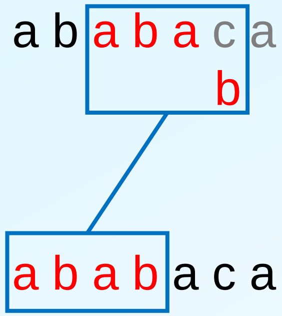
            $k \leftarrow \min(m+1, q+2)$

            repeat

                k--

            until ($P[1\ldots k]$ is a suffix of $P[1\ldots q]\cdot a$) ▷ $x\cdot a = xa$

            $\delta(q, a) \leftarrow k$



- ✓ A naïve implementation takes $\Theta(|\sum|m^3)$
- ✓ With some clever idea: $\Theta(|\sum|m)$ (related to KMP algorithm later)

Matching Algorithm with Automata

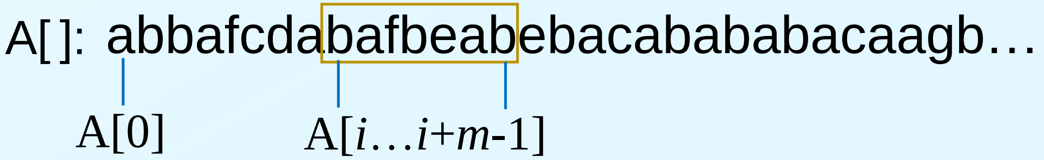
```
FA-Matcher(A, δ , f ):
▷ f : target state
▷ n: length of text array A[ ]
    q ← 0
    for i ← 1 to n
        q ← δ(q, A[i])
        if (q = f )
            Report successful matching at A[i-m+1…]
```

- ✓ Running time: $\Theta(n)$
- ✓ Total running time including generation: $\Theta(n + |\sum|m)$

Rabin-Karp Algorithm

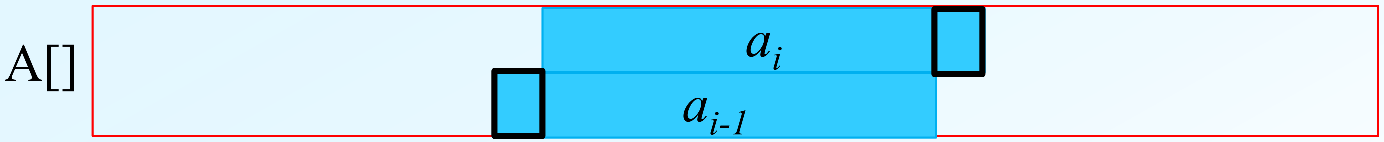
- Transforms each string into an integer (digitization)
  - string comparison → integer comparison
- Transformation
  - A string in k-ary alphabets is converted to a base-k integer ← $|\Sigma| = k$
  - e.g., $\Sigma = \{a, b, c, d, e\}$
    - $k = |\Sigma| = 5$
    - a, b, c, d, and e each corresponds to 0, 1, 2, 3, and 4, respectively
    - "cad" → $2*5^2+0*5^1+3*5^0 = 28$

Potential Problem in Digitization



- Digitizing $A[i \ldots i+m-1]$
  - $a_i = A[i+m-1] + d(A[i+m-2] + d(A[i+m-3] + d(\ldots + d(A[i]))\ldots)$
  - takes $\Theta(m)$
  - Thus, comparisons with all substrings in $A[1 \ldots n]$ takes $\Theta(mn)$
  - No better than the naïve matching
- Fortunately,

  constant-time digitization is possible independent of m
  - $a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$
  - d^{m-1} is used repeatedly; need only one-time computation in advance
  - Enough with just two multiplications and two additions



An Example of Digitization

$P[\ ]$	e	e	a	a	b

$p = 4*5^4 + 4*5^3 + 0*5^2 + 0*5^1 + 1 = 3001$

$A[\ ]$	a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_1=0*5^4+2*5^3+4*5^2+1*5^1+1 = 356$

a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_2= 5(a_1-0*5^4)+2 = 1782$

a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_3=5(a_2-2*5^4)+4 = 2664$

…

a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_7=5(a_6-2*5^4)+1 = 3001$

…

Matching Algorithm by Digitization

basicRabinKarp(A[ ], P[ ], d, q):
▷ n: length of text array A[ ], m: length of pattern array P[ ]
    $p \leftarrow 0$; $a_1 \leftarrow 0$
    for $i \leftarrow 1$ **to** m ▷ Compute a_1
        $p \leftarrow dp + P[i]$
        $a_1 \leftarrow da_1 + A[i]$
    for $i \leftarrow 1$ **to** n-m+1
        if ($i \neq 1$)
            $a_i \leftarrow d(a_{i-1} - d^{m-1}A[i\text{-}1]) + A[i+m\text{-}1]$
        if ($p = a_i$)
            Report successful matching at $A[i\ldots]$

✓ Running time: $\Theta(n)$

Potential Problem

- a_i can be too large depending on $|\Sigma|$ and m
  - may overflow in a computer word
- Resolution
  - Restrict a_i using modulo operator(%)
  - Instead of $a_i = d(a_{i-1} - d^{m-1}A[i-1]) + A[i+m-1]$,

    use $b_i = (d(b_{i-1} - (d^{m-1} \% q) A[i-1]) + A[i+m-1]) \% q$
  - Set q to a large enough prime number so that dq does not overflow in a word(register)

An Example of Digitization by Modulo Operator

P[ ]					
	e	e	a	a	b

$p = (4*5^4 + 4*5^3 + 0*5^2 + 0*5^1 + 1)\ \%\ 113 = 63$

A[ ]																
	a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_1 = (0*5^4+2*5^3+4*5^2+1*5^1+1)\ \%\ 113 = 17$

a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_2 = (5(a_1-0*(60)+2)\ \%\ 113 = 87$ ← $5^4\ \%\ 113 = 60$ ← 뒷 페이지의 h

a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_3 = (5(a_2-2*(60))+4)\ \%\ 113 = 65$

…

a	c	e	b	b	c	e	e	a	a	b	c	e	e	d	b

$a_7 = (5(a_6-2*(60))+1)\ \%\ 113 = 63$

…

Rabin-Karp Algorithm

```
RabinKarp(A[ ], P[ ], d, q):
▷ n: length of text array A[ ], m: length of pattern array P[ ]
    p ← 0; b_1 ← 0
    for i ← 1 to m                                        ▷ Compute b_1
        p ← (dp+ P[i]) % q
        b_1 ← (db_1 + A[i]) % q
    h ← d^{m-1} % q
    for i ← 1 to n-m+1
        if (i ≠ 1)
            b_i ← (d(b_{i-1} – hA[i-1]) + A[i+m-1]) % q
        if (p = b_i)
            if (P[1…m] = A[i…i+m-1])
                Report successful matching at A[i…]
```

Probability of accidental $p = b_i$: $1/q$

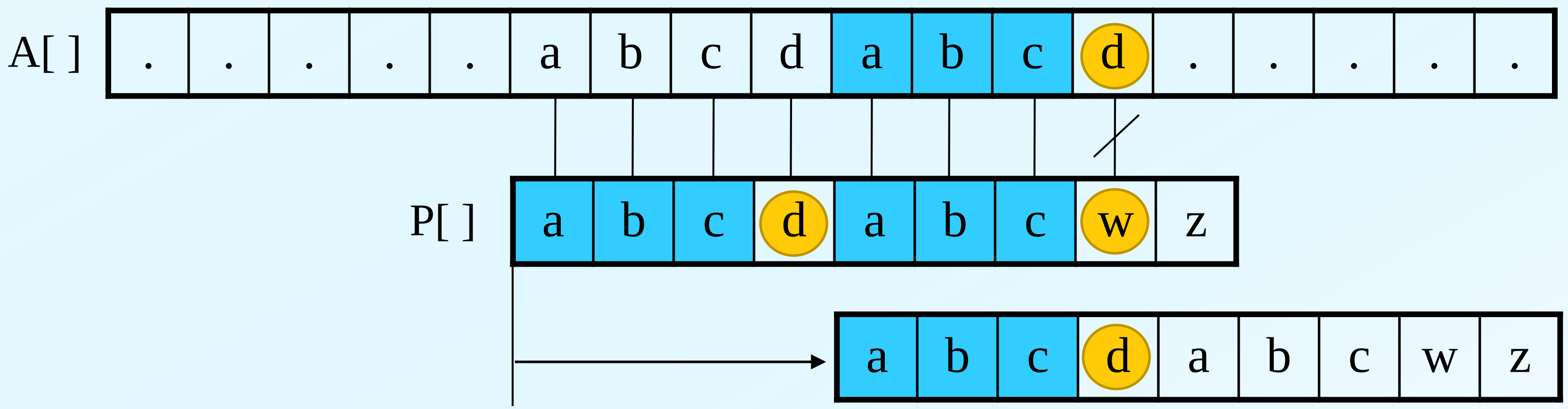
Expected # of accidental $p = b_i$: n/q (negligible)

Usually $n << q$

✓ Average running time: $\Theta(n)$

KMP Knuth-Morris-Pratt Algorithm

- Similar motivation to the matching by automata
- Common part with matching by automata
  - Prepares the returning position after matching failed
  - Simpler preparation than automata matching



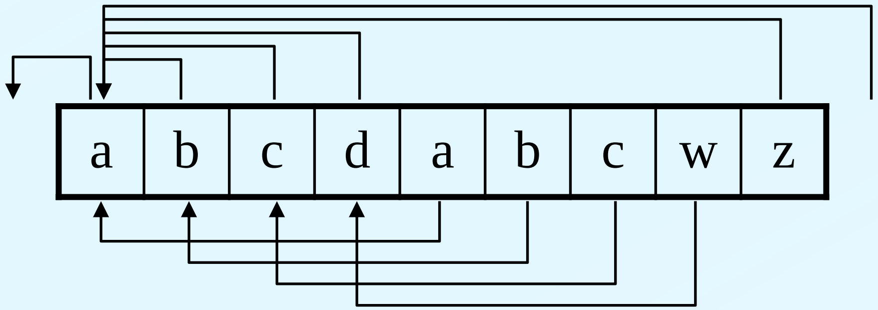
Preparing Returning Position after Fail

	1	2	3	4	5	6	7	8	9
P[ ]	a	b	c	d	a	b	c	w	z

$\pi[8] = 4$

Situation: matched up to the pattern substring “abcdabc” and failed at the pattern symbol ‘w’
Observation: the prefix “abc” and the postfix “abc” right before the failed symbol ‘w’ are the same
→ Compare the text symbol at the failed position with P[4]

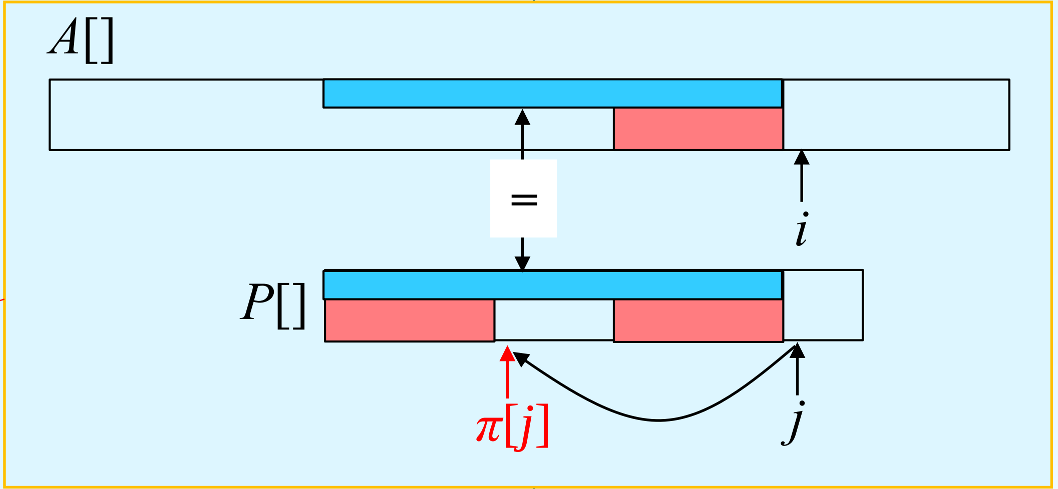
	1	2	3	4	5	6	7	8	9	10
π[ ]	0	1	1	1	1	2	3	4	1	1



For each position of the pattern,
prepare the returning position after fail

KMP Algorithm

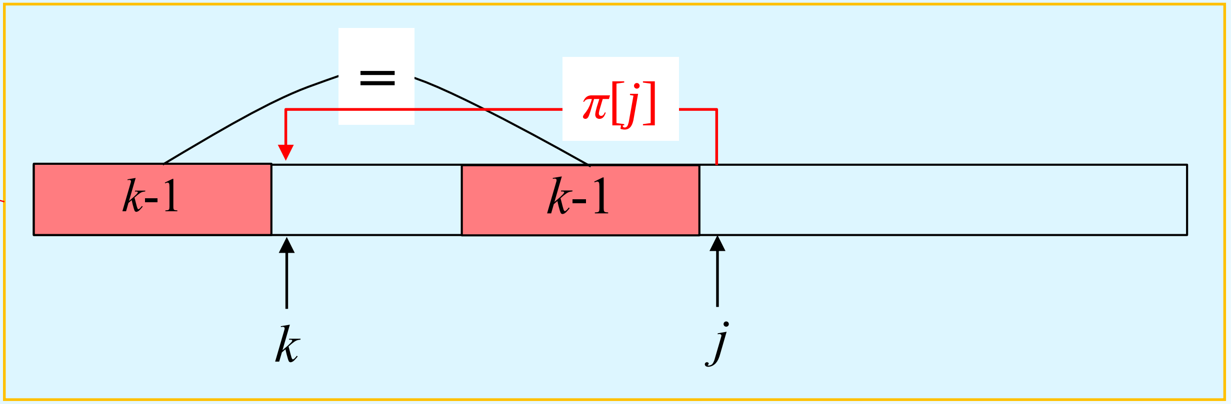
```
KMP(A[ ], P[ ], n, m):
▷ n: length of text array A[ ], m: length of pattern array P[ ]
    preprocessing(P)
    i ← 1   ▷ finger in text string
    j ← 1   ▷ finger in pattern string
    while (i ≤ n)
        if (j = 0 or A[i] = P[j])
            i++;  j++
        else
            j ← π [j]
        if (j = m+1)
            Report successful matching at A[i-m…]
            j ← π [j]
```



✓ Running time: $\Theta(n)$

Preparation

```
preprocessing(P[ ], m):
▷ m: length of pattern array P[ ]
    j ← 1
    k ← 0  ▷ prefix finger
    π[1] ← 0
    while (j ≤ m)
        if (k = 0 or P[j] = P[k])
            j++;  k++; π[j] ← k
        else
            k ← π[k]
```



✓ Running time: $\Theta(m)$

Running-Time Analysis of KMP

Every time we go thru the loop, the algorithm advances
in the text (by i++) or shift the pattern(by $j \leftarrow \pi[j]$).

Note that $\forall j, \pi[j] < j$, so $j \leftarrow \pi[j]$ decreases j.

Thus, each time we go thru the loop, $\boxed{i+(i\text{-}j)}$ will be increased by at least 1.

$i+(i\text{-}j) \le 2i \le 2(n+1)$, i.e., we go thru the loop at most $2n+1$ times.

Since each **while** loop takes $\Theta(1)$, the running time is $O(n)$.

Since $\Omega(n)$, finally $\Theta(n)$.

```
i ← 1; j ← 1
while (i ≤ n)
    if (j = 0 or A[i] = P[j])
        i++;  j++
    else
        j ← π [j]
    if (j = m+1)
        Report successful matching at A[i-m…]
        j ← π [j]
```

Boyer-Moore Algorithm

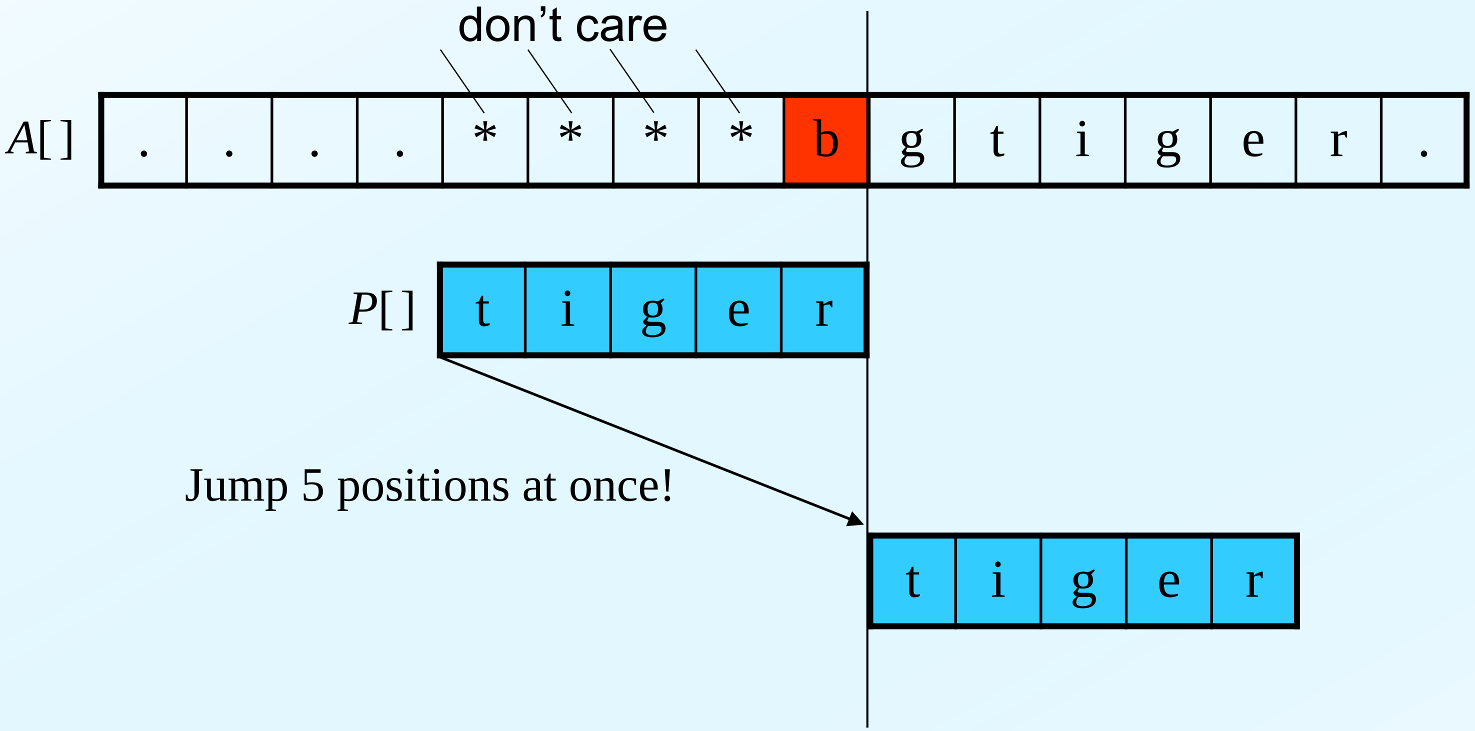
- Common in the algorithms so far
  - Looks at every position in the text at least once
  - So, $\Omega(n)$ even in the best case
- Boyer-Moore algorithm does not have to look at every position in the text
  - Switching the way of thinking: start comparison

    not from the front of the pattern

    but from the rear of the pattern

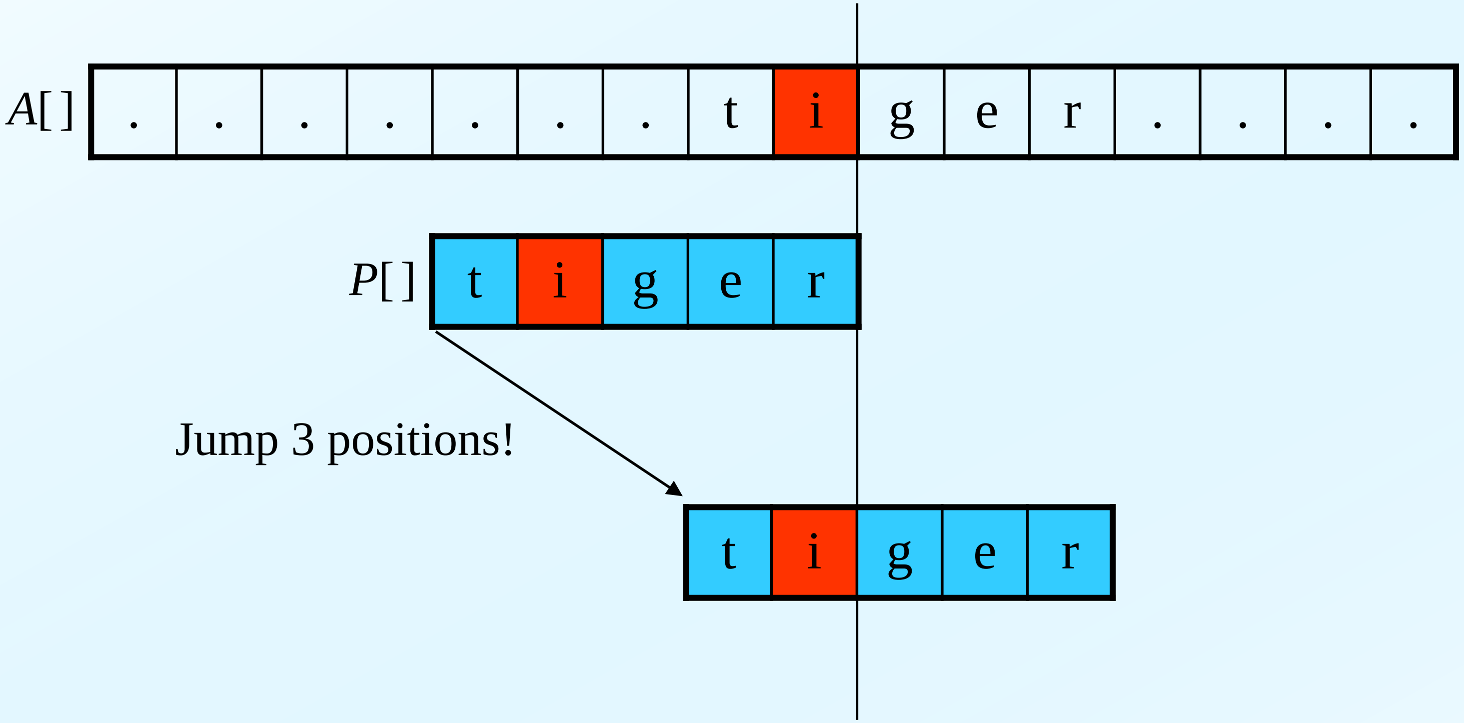
Motivation

Situation: failed by comparison of ‘b’ in the text and ‘r’ in the pattern



✓ Observation: since there is no symbol ‘b’ in the pattern, the pattern can skip over ‘b’ in the text

Situation: failed by comparison of ‘i’ in the text and ‘r’ in the pattern



✓ Observation: since symbol ‘i’ appears at the 3rd left position of ‘r’ in the pattern, the pattern can skip 3 positions

Preparation

Jumping information for "tiger"

Text symbol aligned with 'r'	t	i	g	e	r	others
jump	4	3	2	1	5	5

Jumping information for "rational"

Text symbol aligned with 'l'	r	a	t	i	o	n	a	l	others
jump	7	6	5	4	3	2	1	8	8

Text symbol aligned with 'l'	r	t	i	o	n	a	l	others
jump	7	5	4	3	2	1	8	8

Boyer-Moore-Horspool Algorithm

```
BoyerMooreHorspool(A[ ], P[ ]):
▷ n: length of text array A[ ], m: length of pattern array P[ ]
    computeSkip(P, jump)
    i ← 1
    while (i ≤ n − m+1)
        j ← m;  k ← i + m −1
        while ( j > 0 and P[j] = A[k])
                j--; k--
        if (j = 0)
                Report successful matching at A[i…]
        i ← i + jump[A[i + m − 1]]
```

- ✓ Worst case: $\Theta(mn)$
- ✓ Affected by input, but generally lighter than $\Theta(n)$
- ✓ Best case: $\Theta(\frac{n}{m})$