Design and Analysis of Algorithms

String Matching

Finite-Automaton Matching • Knuth–Morris–Pratt

Ctrl+F hunts a 7-letter word through a million characters — and *never* re-reads one. Today we build that trick twice: first as a machine, then with no machine at all.

Dr. Golzari

University of Hormozgan

Fall 1405

Roadmap — Where We're Going

1. **The hunt** — naive matching: simple, and where it bleeds

Roadmap — Where We're Going

1. **The hunt** — naive matching: simple, and where it bleeds
2. **One fact** — read the text once; keep only *how much of the pattern still matches its tail*

Roadmap — Where We're Going

1. **The hunt** — naive matching: simple, and where it bleeds
2. **One fact** — read the text once; keep only *how much of the pattern still matches its tail*
3. **The machine** — the pattern becomes a finite automaton; we build it and run it on "ababaca"

Roadmap — Where We're Going

1. **The hunt** — naive matching: simple, and where it bleeds
2. **One fact** — read the text once; keep only *how much of the pattern still matches its tail*
3. **The machine** — the pattern becomes a finite automaton; we build it and run it on "ababaca"
4. **The lazy machine** — **KMP**: one small table π replaces the whole automaton

Roadmap — Where We're Going

1. **The hunt** — naive matching: simple, and where it bleeds
2. **One fact** — read the text once; keep only *how much of the pattern still matches its tail*
3. **The machine** — the pattern becomes a finite automaton; we build it and run it on "ababaca"
4. **The lazy machine** — **KMP**: one small table π replaces the whole automaton
5. **The receipt** — $O(m+n)$, and who survives on a stream

Roadmap — Where We're Going

1. **The hunt** — naive matching: simple, and where it bleeds
2. **One fact** — read the text once; keep only *how much of the pattern still matches its tail*
3. **The machine** — the pattern becomes a finite automaton; we build it and run it on "ababaca"
4. **The lazy machine** — **KMP**: one small table π replaces the whole automaton
5. **The receipt** — $O(m+n)$, and who survives on a stream

Today's promise

Run "ababaca" through the 24-edge machine *and* through KMP on the same text — and get the same state row both times.

The Hunt: Text, Pattern, Shifts

- **Given:** text $T[1..n]$ — the haystack; pattern $P[1..m]$ — the needle

The Hunt: Text, Pattern, Shifts

- **Given:** text $T[1..n]$ — the haystack; pattern $P[1..m]$ — the needle
- **Find:** every shift s with $T[s+1..s+m] = P[1..m]$

The Hunt: Text, Pattern, Shifts

- **Given:** text $T[1..n]$ — the haystack; pattern $P[1..m]$ — the needle
- **Find:** every shift s with $T[s+1..s+m] = P[1..m]$

Pop Quiz

$T =$ "abababacaba" $(n = 11)$, $P =$ "ababaca" $(m = 7)$.
Spot an occurrence by eye!

The Hunt: Text, Pattern, Shifts

- **Given:** text $T[1..n]$ — the haystack; pattern $P[1..m]$ — the needle
- **Find:** every shift s with $T[s+1..s+m] = P[1..m]$

Pop Quiz

$T =$ "abababacaba" $(n = 11)$, $P =$ "ababaca" $(m = 7)$.
Spot an occurrence by eye!

- It starts at character 3: $T[3..9] =$ "ababaca" — these two are our specimens for the whole session

Attempt #1: Check Every Shift

```
NaiveMatcher(T, n, P, m)
1: for s ← 0 to n − m do
2:     j ← 1
3:     while j ≤ m and T[s + j] = P[j] do
4:         j ← j + 1
5:     if j = m + 1 then
6:         print "occurrence at shift" s
```

Attempt #1: Check Every Shift

```
NaiveMatcher(T, n, P, m)
1: for s ← 0 to n − m do
2:     j ← 1
3:     while j ≤ m and T[s + j] = P[j] do
4:         j ← j + 1
5:     if j = m + 1 then
6:         print "occurrence at shift" s
```

Pop Quiz

Shift $s = 0$ collects 5 winning comparisons, then dies on the 6th. What happens to those 5?

Attempt #1: Check Every Shift

NaiveMatcher(T, n, P, m)

```
1: for s ← 0 to n − m do
2:     j ← 1
3:     while j ≤ m and T[s + j] = P[j] do
4:         j ← j + 1
5:     if j = m + 1 then
6:         print "occurrence at shift" s
```

Pop Quiz

Shift $s = 0$ collects 5 winning comparisons, then dies on the 6th. What happens to those 5?

- Thrown away. Shift $s = 1$ starts from scratch at $T[2]$ — all memory lived in j, and j just reset

Walkthrough — The Naive Matcher's Bad Day

shift	comparisons	verdict
$s = 0$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, then $b \neq c$	dies after 6

Walkthrough — The Naive Matcher's Bad Day

shift	comparisons	verdict
$s = 0$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, then $b \neq c$	dies after 6
$s = 1$	$b \neq a$	dies after 1

Walkthrough — The Naive Matcher's Bad Day

shift	comparisons	verdict
$s = 0$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, then $b \neq c$	dies after 6
$s = 1$	$b \neq a$	dies after 1
$s = 2$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, $c{=}c$, $a{=}a$	**occurrence!**

Walkthrough — The Naive Matcher's Bad Day

shift	comparisons	verdict
$s = 0$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, then $b \neq c$	dies after 6
$s = 1$	$b \neq a$	dies after 1
$s = 2$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, $c{=}c$, $a{=}a$	**occurrence!**

- Shift $s = 1$ re-read $T[2]$ — a character shift $s = 0$ had read half a second ago

Walkthrough — The Naive Matcher's Bad Day

shift	comparisons	verdict
$s = 0$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, then $b \neq c$	dies after 6
$s = 1$	$b \neq a$	dies after 1
$s = 2$	$a{=}a$, $b{=}b$, $a{=}a$, $b{=}b$, $a{=}a$, $c{=}c$, $a{=}a$	**occurrence!**

- Shift $s = 1$ re-read $T[2]$ — a character shift $s = 0$ had read half a second ago
- Whole scan on our specimen: $6 + 1 + 7 + 1 + 4 = 19$ comparisons — for an *11*-character text

The Worst Case: All Pain, No Memory

- $T =$ “aaaaaaaaaa” $(n = 10)$, $P =$ “aaab”

The Worst Case: All Pain, No Memory

- $T =$ "aaaaaaaaaa" $(n = 10)$, $P =$ "aaab"
- Every shift matches *aaa*, then dies on *b*: $7 \times 4 = 28$ comparisons

The Worst Case: All Pain, No Memory

- $T =$ "aaaaaaaaaa" $(n = 10)$, $P =$ "aaab"
- Every shift matches *aaa*, then dies on *b*: $7 \times 4 = 28$ comparisons
- In general: $(n - m + 1) \cdot m$ comparisons $\Rightarrow$ $\Theta(nm)$ — quadratic

The Worst Case: All Pain, No Memory

- $T =$ "aaaaaaaaaa" $(n = 10)$, $P =$ "aaab"
- Every shift matches *aaa*, then dies on *b*: $7 \times 4 = 28$ comparisons
- In general: $(n - m + 1) \cdot m$ comparisons $\Rightarrow$ $\Theta(nm)$ — quadratic

Pop Quiz

After shift $s = 0$ fails, the matcher *knows* $T[1..3] =$ "aaa". What does it do with that knowledge?

The Worst Case: All Pain, No Memory

- $T =$ "aaaaaaaaaa" $(n = 10)$, $P =$ "aaab"
- Every shift matches *aaa*, then dies on *b*: $7 \times 4 = 28$ comparisons
- In general: $(n - m + 1) \cdot m$ comparisons $\Rightarrow$ $\Theta(nm)$ — quadratic

Pop Quiz

After shift $s = 0$ fails, the matcher *knows* $T[1..3] =$ "aaa". What does it do with that knowledge?

- Burns it: shift $s = 1$ re-reads $T[2]$, $T[3]$, $T[4]$ from zero

The Worst Case: All Pain, No Memory

- $T =$ "aaaaaaaaaa" $(n = 10)$, $P =$ "aaab"
- Every shift matches *aaa*, then dies on *b*: $7 \times 4 = 28$ comparisons
- In general: $(n - m + 1) \cdot m$ comparisons $\Rightarrow$ $\Theta(nm)$ — quadratic

Pop Quiz

After shift $s = 0$ fails, the matcher *knows* $T[1..3] =$ "aaa". What does it do with that knowledge?

- Burns it: shift $s = 1$ re-reads $T[2], T[3], T[4]$ from zero
- **The information was free. We threw it away.**

The One Fact Worth Keeping

- Read T once, left to right. After each character keep one number: how much of P currently matches the *tail* of what we have seen

The One Fact Worth Keeping

- Read T once, left to right. After each character keep one number: how much of P currently matches the *tail* of what we have seen
- Hunting "ababaca", we have seen "…ababa" $\Rightarrow$ the number is 5

The One Fact Worth Keeping

- Read T once, left to right. After each character keep one number: how much of P currently matches the *tail* of what we have seen
- Hunting "ababaca", we have seen "...ababa" $\Rightarrow$ the number is 5

Pop Quiz

The next character is b. How much survives?

The One Fact Worth Keeping

- Read T once, left to right. After each character keep one number: how much of P currently matches the *tail* of what we have seen
- Hunting "ababaca", we have seen "...ababa" $\Rightarrow$ the number is 5

Pop Quiz

The next character is *b*. How much survives?

- "...ababab" still ends in "abab" $= P[1..4] \Rightarrow$ **4** — *not* 0!

The One Fact Worth Keeping

- Read T once, left to right. After each character keep one number: how much of P currently matches the *tail* of what we have seen
- Hunting "ababaca", we have seen "...ababa" $\Rightarrow$ the number is 5

Pop Quiz

The next character is *b*. How much survives?

- "...ababab" still ends in "abab" $= P[1..4] \Rightarrow$ **4** — *not* 0!
- **Naive says 0. We say 4. That gap is the whole lecture.**

The Pattern Becomes a Machine

- State q = length of the *longest prefix of P that is a suffix* of the text read so far; states $0, 1, \ldots, m$, start 0, accept m

The Pattern Becomes a Machine

- State q = length of the *longest prefix of P that is a suffix* of the text read so far; states $0, 1, \ldots, m$, start 0, accept m
- One move: $\delta(q, a) = \sigma\big(P[1..q]\, a\big)$, where $\sigma(w)$ = the largest k with $P[1..k]$ a suffix of w

The Pattern Becomes a Machine

- State q = length of the *longest prefix of P that is a suffix* of the text read so far; states $0, 1, \ldots, m$, start 0, accept m
- One move: $\delta(q, a) = \sigma\big(P[1..q]\, a\big)$, where $\sigma(w)$ = the largest k with $P[1..k]$ a suffix of w

FAMatcher(T, n, δ, m)

```
1: q ← 0
2: for i ← 1 to n do
3:     q ← δ(q, T[i])
4:     if q = m then
5:         print "shift" i − m
```

The Pattern Becomes a Machine

- State q = length of the *longest prefix of P that is a suffix* of the text read so far; states $0, 1, \ldots, m$, start 0, accept m
- One move: $\delta(q, a) = \sigma\big(P[1..q]\, a\big)$, where $\sigma(w)$ = the largest k with $P[1..k]$ a suffix of w

FAMatcher(T, n, δ, m)

```
1: q ← 0
2: for i ← 1 to n do
3:     q ← δ(q, T[i])
4:     if q = m then
5:         print "shift" i − m
```

- **Pop quiz:** work per text character? One lookup $\Rightarrow$ the whole scan is $O(n)$

The Pattern Becomes a Machine

- State q = length of the *longest prefix of P that is a suffix* of the text read so far; states $0, 1, \ldots, m$, start 0, accept m
- One move: $\delta(q, a) = \sigma\big(P[1..q]\, a\big)$, where $\sigma(w)$ = the largest k with $P[1..k]$ a suffix of w

FAMatcher(T, n, δ, m)

```
1: q ← 0
2: for i ← 1 to n do
3:     q ← δ(q, T[i])
4:     if q = m then
5:         print "shift" i − m
```

- **Pop quiz:** work per text character? One lookup $\Rightarrow$ the whole scan is $O(n)$
- **So the real bill:** building δ — $(m{+}1)\,|\Sigma|$ cells

Hand-Building One Transition: $\delta(5,\ b)$

- In state 5 the text so far ends in “ababa”. Next character: b

Hand-Building One Transition: $\delta(5,\ b)$

- In state 5 the text so far ends in "ababa". Next character: b
- $\delta(5, b) = \sigma(\text{“ababa”} + \text{“b”}) = \sigma(\text{“ababab”})$ — the longest prefix of P that is a *suffix* of "ababab":

Hand-Building One Transition: $\delta(5,\ b)$

- In state 5 the text so far ends in "ababa". Next character: b
- $\delta(5, b) = \sigma(\text{"ababa"} + \text{"b"}) = \sigma(\text{"ababab"})$ — the longest prefix of P that is a *suffix* of "ababab":

k	candidate	verdict
6	"ababab" vs $P[1..6] =$ "ababac"	$\times$ (the c!)

Hand-Building One Transition: $\delta(5,\ b)$

- In state 5 the text so far ends in "ababa". Next character: b
- $\delta(5, b) = \sigma(\text{“ababa”} + \text{“b”}) = \sigma(\text{“ababab”})$ — the longest prefix of P that is a *suffix* of "ababab":

k	candidate	verdict
6	"ababab" vs $P[1..6] =$ "ababac"	$\times$ (the c!)
5	"babab" vs "ababa"	$\times$

Hand-Building One Transition: $\delta(5,\ b)$

- In state 5 the text so far ends in "ababa". Next character: b
- $\delta(5, b) = \sigma(\text{"ababa"} + \text{"b"}) = \sigma(\text{"ababab"})$ — the longest prefix of P that is a *suffix* of "ababab":

k	candidate	verdict
6	"ababab" vs $P[1..6] =$ "ababac"	$\times$ (the c!)
5	"babab" vs "ababa"	$\times$
4	"abab" vs "abab"	✓

Hand-Building One Transition: $\delta(5,\ b)$

- In state 5 the text so far ends in "ababa". Next character: b
- $\delta(5, b) = \sigma(\text{"ababa"} + \text{"b"}) = \sigma(\text{"ababab"})$ — the longest prefix of P that is a *suffix* of "ababab":

k	candidate	verdict
6	"ababab" vs $P[1..6] =$ "ababac"	$\times$ (the c!)
5	"babab" vs "ababa"	$\times$
4	"abab" vs "abab"	✓

Result

$\delta(5, b) = 4$: we fall from 5 to 4 and keep 4 characters of progress. Naive falls to 0.

The Full Automaton for “ababaca”

state q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, c)$
0	1	0	0
1	1	2	0
2	3	0	0

The Full Automaton for “ababaca”

state q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, c)$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	**4**	**6**

The Full Automaton for “ababaca”

state q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, c)$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	**4**	**6**
6	**7**	0	0
7	1	2	0

The Full Automaton for “ababaca”

state q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, c)$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	**4**	**6**
6	**7**	0	0
7	1	2	0

- Row q: “I hold q matched chars — next char x: how many now?”

The Full Automaton for “ababaca”

state q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, c)$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	**4**	**6**
6	**7**	0	0
7	1	2	0

- Row q: “I hold q matched chars — next char x: how many now?”

Pop Quiz

The text serves a letter d never seen in P — what is $\delta(q, d)$?

The Full Automaton for “ababaca”

state q	$\delta(q, a)$	$\delta(q, b)$	$\delta(q, c)$
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	**4**	**6**
6	**7**	0	0
7	1	2	0

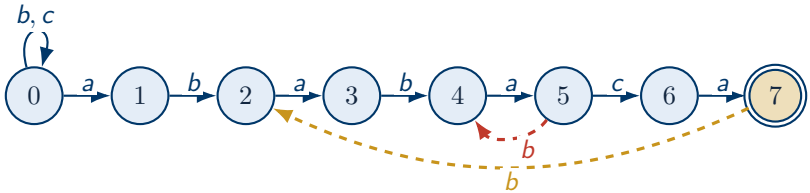
- Row q: “I hold q matched chars — next char x: how many now?”

Pop Quiz

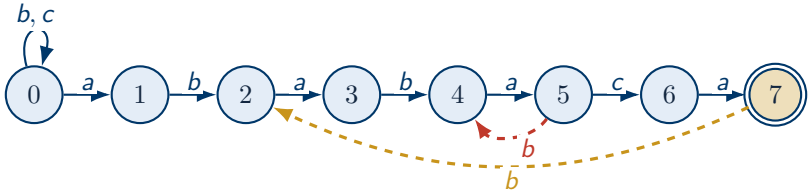
The text serves a letter d never seen in P — what is $\delta(q, d)$?

- 0 for *every* q — no prefix of P ends in d: one dead column, free

The Shape of the Machine

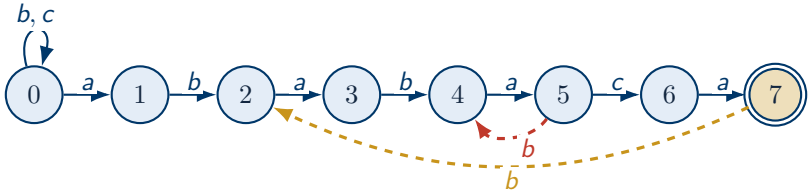


The Shape of the Machine



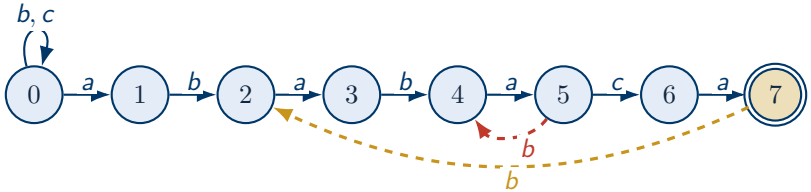
- Solid path: the success ladder $a\,b\,a\,b\,a\,c\,a$; dashed edges are the rescues: $\delta(5, b) = 4$, $\delta(7, b) = 2$; loops: "stay put" moves

The Shape of the Machine



- Solid path: the success ladder *a b a b a c a*; dashed edges are the rescues: $\delta(5, b) = 4$, $\delta(7, b) = 2$; loops: "stay put" moves
- The full machine has 24 edges — we drew the interesting ones

The Shape of the Machine

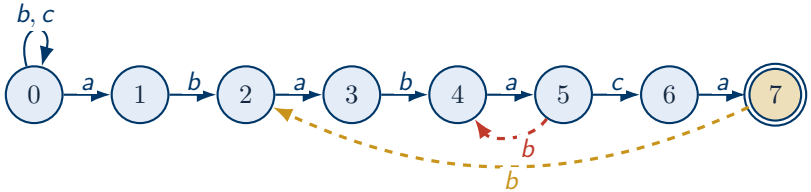


- Solid path: the success ladder *a b a b a c a*; dashed edges are the rescues: $\delta(5, b) = 4$, $\delta(7, b) = 2$; loops: "stay put" moves
- The full machine has 24 edges — we drew the interesting ones

Pop Quiz

State 7: a match *just* happened! Next character is *b*. Restart from 0?

The Shape of the Machine



- Solid path: the success ladder $a\,b\,a\,b\,a\,c\,a$; dashed edges are the rescues: $\delta(5, b) = 4$, $\delta(7, b) = 2$; loops: "stay put" moves
- The full machine has 24 edges — we drew the interesting ones

Pop Quiz

State 7: a match *just* happened! Next character is b. Restart from 0?

- Never: "…ababaca b" ends in "ab" $\Rightarrow \delta(7, b) = 2$ — the machine is *already hunting the next occurrence*

Walkthrough — The Automaton at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a

Walkthrough — The Automaton at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	2	3

Walkthrough — The Automaton at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	2	3

- $i = 6$: state 5 reads $b \Rightarrow$ 4, *not* 0 — four characters of progress survive

Walkthrough — The Automaton at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	2	3

- $i = 6$: state 5 reads $b \Rightarrow 4$, *not* 0 — four characters of progress survive
- $i = 9$: $q = 7 = m \Rightarrow$ shift $9 - 7 = 2$; check $T[3..9] =$ "ababaca" ✓

Walkthrough — The Automaton at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	2	3

- $i = 6$: state 5 reads $b \Rightarrow 4$, *not* 0 — four characters of progress survive
- $i = 9$: $q = 7 = m \Rightarrow$ shift $9 - 7 = 2$; check $T[3..9] =$ "ababaca" ✓
- $i = 10$: the gold rescue — $\delta(7, b) = 2$, and the hunt continues with no restart

The Bill: Cheap Scan, Pricey Factory

- **Scan:** n lookups $\times\, O(1) = O(n)$

The Bill: Cheap Scan, Pricey Factory

- **Scan:** n lookups $\times\, O(1) = O(n)$
- **Build:** $(m{+}1)\,|\Sigma|$ cells; each σ tries up to m candidate lengths, each check costs $O(m) \;\Rightarrow\; O(m^3|\Sigma|)$

The Bill: Cheap Scan, Pricey Factory

- **Scan:** n lookups $\times\, O(1) = O(n)$
- **Build:** $(m{+}1)\,|\Sigma|$ cells; each σ tries up to m candidate lengths, each check costs $O(m) \;\Rightarrow\; O(m^3|\Sigma|)$
- Real units: $m = 1000$, $|\Sigma| = 50 \Rightarrow 5 \times 10^{10}$ steps to build — KMP will do the same job in $m + n \approx 10^6$: about $50{,}000\times$ less work

The Bill: Cheap Scan, Pricey Factory

- **Scan:** n lookups $\times\, O(1) = O(n)$
- **Build:** $(m{+}1)\,|\Sigma|$ cells; each σ tries up to m candidate lengths, each check costs $O(m) \;\Rightarrow\; O(m^3|\Sigma|)$
- Real units: $m = 1000$, $|\Sigma| = 50 \Rightarrow 5 \times 10^{10}$ steps to build — KMP will do the same job in $m + n \approx 10^6$: about $50{,}000\times$ less work

Pop Quiz

Which factor explodes when the alphabet is rich (Unicode, DNA with quality codes)?

The Bill: Cheap Scan, Pricey Factory

- **Scan:** n lookups $\times\, O(1) = O(n)$
- **Build:** $(m{+}1)\,|\Sigma|$ cells; each σ tries up to m candidate lengths, each check costs $O(m) \;\Rightarrow\; O(m^3|\Sigma|)$
- Real units: $m = 1000$, $|\Sigma| = 50 \Rightarrow 5 \times 10^{10}$ steps to build — KMP will do the same job in $m + n \approx 10^6$: about $50{,}000\times$ less work

Pop Quiz

Which factor explodes when the alphabet is rich (Unicode, DNA with quality codes)?

- $|\Sigma|$: the table has a full column for *every* symbol

The Bill: Cheap Scan, Pricey Factory

- **Scan:** n lookups $\times\, O(1) = O(n)$
- **Build:** $(m{+}1)\,|\Sigma|$ cells; each σ tries up to m candidate lengths, each check costs $O(m) \;\Rightarrow\; O(m^3|\Sigma|)$
- Real units: $m = 1000$, $|\Sigma| = 50 \Rightarrow 5 \times 10^{10}$ steps to build — KMP will do the same job in $m + n \approx 10^6$: about $50{,}000\times$ less work

Pop Quiz

Which factor explodes when the alphabet is rich (Unicode, DNA with quality codes)?

- $|\Sigma|$: the table has a full column for *every* symbol
- **New question:** can each transition be computed *just in time* — from the pattern alone, no table?

The Failure Function π

- $\pi[q]$ = length of the longest *proper* prefix of $P[1..q]$ that is also a *suffix* of $P[1..q]$

The Failure Function π

- $\pi[q]$ = length of the longest *proper* prefix of $P[1..q]$ that is also a *suffix* of $P[1..q]$
- "Proper" = strictly shorter — a genuine fallback, never the whole thing

The Failure Function π

- $\pi[q]$ = length of the longest *proper* prefix of $P[1..q]$ that is also a *suffix* of $P[1..q]$
- "Proper" = strictly shorter — a genuine fallback, never the whole thing
- $P[1..5]$ = "ababa": prefix "aba" = suffix "aba" $\Rightarrow \pi[5] = 3$

The Failure Function π

- $\pi[q]$ = length of the longest *proper* prefix of $P[1..q]$ that is also a *suffix* of $P[1..q]$
- "Proper" = strictly shorter — a genuine fallback, never the whole thing
- $P[1..5]$ = "ababa": prefix "aba" = suffix "aba" $\Rightarrow \pi[5] = 3$

Pop Quiz

$P[1..6]$ = "ababac" — guess $\pi[6]$.

The Failure Function π

- $\pi[q]$ = length of the longest *proper* prefix of $P[1..q]$ that is also a *suffix* of $P[1..q]$
- "Proper" = strictly shorter — a genuine fallback, never the whole thing
- $P[1..5]$ = "ababa": prefix "aba" = suffix "aba" $\Rightarrow \pi[5] = 3$

Pop Quiz

$P[1..6]$ = "ababac" — guess $\pi[6]$.

- **0**: the only prefix of P ending in c is $P[1..6]$ itself — and π must be *proper*

The Failure Function π

- $\pi[q]$ = length of the longest *proper* prefix of $P[1..q]$ that is also a *suffix* of $P[1..q]$
- "Proper" = strictly shorter — a genuine fallback, never the whole thing
- $P[1..5]$ = "ababa": prefix "aba" = suffix "aba" $\Rightarrow \pi[5] = 3$

Pop Quiz

$P[1..6]$ = "ababac" — guess $\pi[6]$.

- **0**: the only prefix of P ending in c is $P[1..6]$ itself — and π must be *proper*
- **Key sentence:** from state q on a mismatch, the best rescue lands at $\pi[q]$ — then we re-compare the *same* text character

Computing π for "ababaca"

q	1	2	3	4	5	6	7
$P[1..q]$	a	ab	aba	abab	ababa	ababac	ababaca

Computing π for "ababaca"

q	1	2	3	4	5	6	7
$P[1..q]$	a	ab	aba	abab	ababa	ababac	ababaca
$\pi[q]$	0	0	1	2	3	**0**	1

Computing π for "ababaca"

q	1	2	3	4	5	6	7
$P[1..q]$	a	ab	aba	abab	ababa	ababac	ababaca
$\pi[q]$	0	0	1	2	3	**0**	1

- The chain behind $\pi[6]$: start $k = \pi[5] = 3$; $P[4] =$ "b" $\neq$ "c" $\Rightarrow$ $k = \pi[3] = 1$; $P[2] =$ "b" $\neq$ "c" $\Rightarrow$ $k = \pi[1] = 0$; $P[1] =$ "a" $\neq$ "c" $\Rightarrow$ $\pi[6] = 0$

Computing π for "ababaca"

q	1	2	3	4	5	6	7
$P[1..q]$	a	ab	aba	abab	ababa	ababac	ababaca
$\pi[q]$	0	0	1	2	3	**0**	1

- The chain behind $\pi[6]$: start $k = \pi[5] = 3$; $P[4] =$ "b" $\neq$ "c" $\Rightarrow$ $k = \pi[3] = 1$; $P[2] =$ "b" $\neq$ "c" $\Rightarrow$ $k = \pi[1] = 0$; $P[1] =$ "a" $\neq$ "c" $\Rightarrow$ $\pi[6] = 0$
- $\pi[7]$: $k = 0$, $P[1] =$ "a" $= P[7] \Rightarrow 1$ — the ladder restarts

KMP — The Pseudocode

ComputePi(P, m)

1: $\pi[1] \leftarrow 0; \quad k \leftarrow 0$
2: **for** $q \leftarrow 2$ **to** m **do**
3: $\quad$ **while** $k > 0$ **and** $P[k+1] \neq P[q]$ **do**
4: $\quad\quad k \leftarrow \pi[k]$
5: $\quad$ **if** $P[k+1] = P[q]$ **then**
6: $\quad\quad k \leftarrow k + 1$
7: $\quad \pi[q] \leftarrow k$
8: **return** π

KMPMatcher(T, n, P, m)

1: $\pi \leftarrow$ **ComputePi**(P, m)
2: $q \leftarrow 0$
3: **for** $i \leftarrow 1$ **to** n **do**
4: $\quad$ **while** $q > 0$ **and** $P[q+1] \neq T[i]$ **do**
5: $\quad\quad q \leftarrow \pi[q]$
6: $\quad$ **if** $P[q+1] = T[i]$ **then**
7: $\quad\quad q \leftarrow q + 1$
8: $\quad$ **if** $q = m$ **then**
9: $\quad\quad$ **print** "shift" $i - m; \quad q \leftarrow \pi[q]$

KMP — The Pseudocode

ComputePi(P, m)

```
1: π[1] ← 0;   k ← 0
2: for q ← 2 to m do
3:     while k > 0 and P[k+1] ≠ P[q] do
4:         k ← π[k]
5:     if P[k+1] = P[q] then
6:         k ← k+1
7:     π[q] ← k
8: return π
```

KMPMatcher(T, n, P, m)

```
1: π ← ComputePi(P, m)
2: q ← 0
3: for i ← 1 to n do
4:     while q > 0 and P[q+1] ≠ T[i] do
5:         q ← π[q]
6:     if P[q+1] = T[i] then
7:         q ← q+1
8:     if q = m then
9:         print "shift" i − m;   q ← π[q]
```

- Left: the rescue map in $O(m)$. Right: the **while + if** pair is exactly $\delta(q, T[i])$ computed on demand — each $T[i]$ is compared, then *gone*

KMP — The Pseudocode

ComputePi(P, m)

1: $\pi[1] \leftarrow 0; \quad k \leftarrow 0$
2: **for** $q \leftarrow 2$ **to** m **do**
3: **while** $k > 0$ **and** $P[k+1] \neq P[q]$ **do**
4: $k \leftarrow \pi[k]$
5: **if** $P[k+1] = P[q]$ **then**
6: $k \leftarrow k + 1$
7: $\pi[q] \leftarrow k$
8: **return** π

KMPMatcher(T, n, P, m)

1: $\pi \leftarrow$ **ComputePi**(P, m)
2: $q \leftarrow 0$
3: **for** $i \leftarrow 1$ **to** n **do**
4: **while** $q > 0$ **and** $P[q+1] \neq T[i]$ **do**
5: $q \leftarrow \pi[q]$
6: **if** $P[q+1] = T[i]$ **then**
7: $q \leftarrow q + 1$
8: **if** $q = m$ **then**
9: **print** "shift" $i - m; \quad q \leftarrow \pi[q]$

- Left: the rescue map in $O(m)$. Right: the **while + if** pair is exactly $\delta(q, T[i])$ computed on demand — each $T[i]$ is compared, then *gone*
- Why is $P[m+1]$ never touched? After a hit, q is cut to $\pi[m]$ at once — state m never survives into the next round

Walkthrough — KMP at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a

Walkthrough — KMP at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	**2**	3

Walkthrough — KMP at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	**2**	3

- $i = 6$: $q = 5$, $P[6] =$ “c” $\neq$ “b” $\Rightarrow q = \pi[5] = 3$; retry: $P[4] =$ “b” ✓ $\Rightarrow q = 4$

Walkthrough — KMP at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	**2**	3

- $i = 6$: $q = 5$, $P[6] =$ "c" $\neq$ "b" $\Rightarrow q = \pi[5] = 3$; retry: $P[4] =$ "b" ✓ $\Rightarrow q = 4$
- $i = 9$: $q = 7 = m \Rightarrow$ shift 2 printed; then $q \leftarrow \pi[7] = 1$

Walkthrough — KMP at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	**2**	3

- $i = 6$: $q = 5$, $P[6] =$ "c" $\neq$ "b" $\Rightarrow q = \pi[5] = 3$; retry: $P[4] =$ "b" ✓ $\Rightarrow q = 4$
- $i = 9$: $q = 7 = m \Rightarrow$ shift 2 printed; then $q \leftarrow \pi[7] = 1$
- $i = 10$: $q = 1$, $P[2] =$ "b" $=$ "b" $\Rightarrow q = 2$ — the gold rescue, recomputed with no table

Walkthrough — KMP at Work

i	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	a	b	a	b	a	b	a	c	a	b	a
q	1	2	3	4	5	**4**	5	6	**7**	**2**	3

- $i = 6$: $q = 5$, $P[6] =$ "c" $\neq$ "b" $\Rightarrow q = \pi[5] = 3$; retry: $P[4] =$ "b" ✓ $\Rightarrow q = 4$
- $i = 9$: $q = 7 = m \Rightarrow$ shift 2 printed; then $q \leftarrow \pi[7] = 1$
- $i = 10$: $q = 1$, $P[2] =$ "b" $=$ "b" $\Rightarrow q = 2$ — the gold rescue, recomputed with no table

Punchline

State row: $1, 2, 3, 4, 5, 4, 5, 6, 7, 2, 3$ — identical to the machine's. KMP rebuilt every δ value on the fly, from π alone.

Why $O(m+n)$ Is Honest

- In the scan, only one line raises q: $q \leftarrow q+1$ — at most once per character: $\leq n$ raises in total

Why $O(m+n)$ Is Honest

- In the scan, only one line raises q: $q \leftarrow q+1$ — at most once per character: $\leq n$ raises in total
- $\pi[q] < q$ *always* (proper prefix!) — so every while-iteration strictly *lowers* q

Why $O(m+n)$ Is Honest

- In the scan, only one line raises q: $q \leftarrow q+1$ — at most once per character: $\leq n$ raises in total
- $\pi[q] < q$ *always* (proper prefix!) — so every while-iteration strictly *lowers* q
- Total lowerings $\leq$ total raisings $\leq n \Rightarrow$ the while is amortized free: scan $O(n)$; same argument on k in **ComputePi** $\Rightarrow O(m)$

Why $O(m+n)$ Is Honest

- In the scan, only one line raises q: $q \leftarrow q+1$ — at most once per character: $\leq n$ raises in total
- $\pi[q] < q$ *always* (proper prefix!) — so every while-iteration strictly *lowers* q
- Total lowerings $\leq$ total raisings $\leq n \Rightarrow$ the while is amortized free: scan $O(n)$; same argument on k in **ComputePi** $\Rightarrow O(m)$

The receipt

$O(m)$ memory · $O(m+n)$ time · each character of T read exactly once

Why $O(m+n)$ Is Honest

- In the scan, only one line raises q: $q \leftarrow q+1$ — at most once per character: $\leq n$ raises in total
- $\pi[q] < q$ *always* (proper prefix!) — so every while-iteration strictly *lowers* q
- Total lowerings $\leq$ total raisings $\leq n \Rightarrow$ the while is amortized free: scan $O(n)$; same argument on k in **ComputePi** $\Rightarrow O(m)$

The receipt

$O(m)$ memory · $O(m+n)$ time · each character of T read exactly once

Pop Quiz

State 7 just matched; next char is b: $q \leftarrow \pi[7] = 1$, retry $P[2] =$ "b" ✓ $\Rightarrow q = 2$. Did KMP re-read the text?

Why $O(m+n)$ Is Honest

- In the scan, only one line raises q: $q \leftarrow q+1$ — at most once per character: $\leq n$ raises in total
- $\pi[q] < q$ *always* (proper prefix!) — so every while-iteration strictly *lowers* q
- Total lowerings $\leq$ total raisings $\leq n \Rightarrow$ the while is amortized free: scan $O(n)$; same argument on k in **ComputePi** $\Rightarrow O(m)$

The receipt

$O(m)$ memory · $O(m+n)$ time · each character of T read exactly once

Pop Quiz

State 7 just matched; next char is b: $q \leftarrow \pi[7] = 1$, retry $P[2] =$ "b" ✓ $\Rightarrow q = 2$. Did KMP re-read the text?

- No — same character, new state. That was the machine's gold rescue edge, rebuilt from π

The Big Picture

	preprocessing	scan	worst total	re-reads T?
Naive	—	restart-heavy	$O\big((n-m+1)m\big)$	yes

The Big Picture

	preprocessing	**scan**	**worst total**	**re-reads T?**
Naive	—	restart-heavy	$O\big((n-m+1)m\big)$	**yes**
Automaton	$O(m^3\|\Sigma\|)$	$O(n)$	$O(m^3\|\Sigma\| + n)$	**no**

The Big Picture

	preprocessing	scan	worst total	re-reads T?
Naive	—	restart-heavy	$O\big((n-m+1)m\big)$	**yes**
Automaton	$O(m^3\|\Sigma\|)$	$O(n)$	$O(m^3\|\Sigma\| + n)$	**no**
KMP	$O(m)$	$O(n)$	$O(m+n)$	**no**

The Big Picture

	preprocessing	scan	worst total	re-reads T?
Naive	—	restart-heavy	$O\big((n-m+1)m\big)$	**yes**
Automaton	$O(m^3\lvert\Sigma\rvert)$	$O(n)$	$O(m^3\lvert\Sigma\rvert + n)$	**no**
KMP	$O(m)$	$O(n)$	$O(m+n)$	**no**

One sentence to keep

KMP = the string-matching automaton, manufactured *lazily* from π — same answers, no factory.

Try It Yourself #1 — Build a Machine

Your turn

$P =$ "aab" over $\Sigma = \{a, b\}$. Build the full table $\delta(q, a)$, $\delta(q, b)$ for $q = 0, 1, 2, 3$ — then run the machine on $T =$ "aabaabaab".

Report the state row and every shift where the machine hits state 3.

Try It Yourself #1 — Build a Machine

Your turn

$P =$ "aab" over $\Sigma = \{a, b\}$. Build the full table $\delta(q, a)$, $\delta(q, b)$ for $q = 0, 1, 2, 3$ — then run the machine on $T =$ "aabaabaab".

Report the state row and every shift where the machine hits state 3.

Answer

δ: row 0 $\to (1, 0)$; row 1 $\to (2, 0)$; row 2 $\to (2, \mathbf{3})$; row 3 $\to (1, 0)$.
State row: $1, 2, 3 \mid 1, 2, 3 \mid 1, 2, 3$ — hits at shifts 0, 3, 6.

After each hit the machine drops to state 1 and is instantly ready for the next "aab".

Try It Yourself #2 — Failure Functions

Your turn

Compute $\pi[q]$ for $q = 1, \ldots, 8$ of $P =$ "abababab".

Try It Yourself #2 — Failure Functions

Your turn

Compute $\pi[q]$ for $q = 1, \ldots, 8$ of $P =$ "abababab".

Answer

$\pi = (0,\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6)$ — from $q = 3$ on, $\pi[q] = q - 2$: every step extends the match, the ladder never breaks.

This shape is why "abababab" is so fast to *find* — and think about what the same shape does to a naive matcher.

Try It Yourself #3 — Trace KMP

Your turn

$P =$ "aab" $(\pi = 0, 1, 0)$, $T =$ "aababaab".

Trace **KMPMatcher**: q after each character, every fallback, every printed shift.

Try It Yourself #3 — Trace KMP

Your turn

$P =$ "aab" $(\pi = 0, 1, 0)$, $T =$ "aababaab".

Trace **KMPMatcher**: q after each character, every fallback, every printed shift.

Answer

i	1	2	3	4	5	6	7	8
$T[i]$	a	a	b	a	b	a	a	b
q	1	2	**3**	1	**0**	1	2	**3**

Hits: $i = 3$ (shift 0) and $i = 8$ (shift 5); after each, $q \leftarrow \pi[3] = 0$.

$i = 5$: $q = 1$, "a" $\neq$ "b" $\Rightarrow$ one hop to $\pi[1] = 0$ — one fallback, zero re-reads.

Concept Corner

Q1

Why can the automaton answer in $O(1)$ per character — what does its state remember?

Concept Corner

Q1

Why can the automaton answer in $O(1)$ per character — what does its state remember?

The longest prefix of P matching the text's tail — everything outside it can never start a match: safe to forget.

Concept Corner

Q1

Why can the automaton answer in $O(1)$ per character — what does its state remember?

The longest prefix of P matching the text's tail — everything outside it can never start a match: safe to forget.

Q2

Where exactly does KMP compute $\delta(q, T[i])$?

Concept Corner

Q1

Why can the automaton answer in $O(1)$ per character — what does its state remember?

The longest prefix of P matching the text's tail — everything outside it can never start a match: safe to forget.

Q2

Where exactly does KMP compute $\delta(q, T[i])$?

The while + if pair: slide back along π until $P[q+1] = T[i]$ or $q = 0$, then maybe advance.

Concept Corner

Q1

Why can the automaton answer in $O(1)$ per character — what does its state remember?

The longest prefix of P matching the text's tail — everything outside it can never start a match: safe to forget.

Q2

Where exactly does KMP compute $\delta(q, T[i])$?

The while + if pair: slide back along π until $P[q+1] = T[i]$ or $q = 0$, then maybe advance.

Q3

The text arrives as a *stream* — no going back. Who survives?

Concept Corner

Q1

Why can the automaton answer in $O(1)$ per character — what does its state remember?

The longest prefix of P matching the text's tail — everything outside it can never start a match: safe to forget.

Q2

Where exactly does KMP compute $\delta(q, T[i])$?

The while + if pair: slide back along π until $P[q+1] = T[i]$ or $q = 0$, then maybe advance.

Q3

The text arrives as a *stream* — no going back. Who survives?

The automaton and KMP — each character read once, $O(m)$ memory. Naive re-reads; a stream does not rewind.

Summary — What You Own Now

The hunt

Find every shift s with $T[s+1..s+m] = P[1..m]$; naive pays $(n-m+1)\,m$ in the worst case because it forgets everything between shifts.

The machine

State = longest prefix of P matching the text's tail; $\delta(q,a) = \sigma(P[1..q]\,a)$. Scan $O(n)$, build $O(m^3|\Sigma|)$ — the factory is the bill.

The lazy machine

π = longest *proper* prefix-suffix; KMP: $O(m)$ build, $O(n)$ scan, never re-reads — and $\pi[q] < q$ is the entire proof.

Summary — What You Own Now

The hunt

Find every shift s with $T[s+1..s+m] = P[1..m]$; naive pays $(n-m+1)\,m$ in the worst case because it forgets everything between shifts.

The machine

State = longest prefix of P matching the text's tail; $\delta(q,a) = \sigma(P[1..q]\,a)$. Scan $O(n)$, build $O(m^3|\Sigma|)$ — the factory is the bill.

The lazy machine

π = longest *proper* prefix-suffix; KMP: $O(m)$ build, $O(n)$ scan, never re-reads — and $\pi[q] < q$ is the entire proof.

Before next session

Re-run Try-It #3 with the answer covered; then compute π for "aabaab".