String Algorithms

David Kauchak
cs302
Spring 2012

Where did "dynamic programming" come from?

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities" (p. 159).

Richard Bellman On the Birth of Dynamic Programming

Stuart Dreyfus

http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf

Strings

- Let Σ be an alphabet, e.g. Σ = (, a, b, c, …, z)
- A string is any member of Σ*, i.e. any sequence of 0 or more members of Σ
  - 'this is a string' ∈ Σ*
  - 'this is also a string' ∈ Σ*
  - '1234' ∉ Σ*

String operations

- Given strings s_1 of length n and s_2 of length m
- Equality: is $s_1 = s_2$? (case sensitive or insensitive)

  'this is a string' = 'this is a string'

  'this is a string' ≠ 'this is another string'

  'this is a string' =? 'THIS IS A STRING'

- Running time
  - O(n) where n is length of shortest string

String operations

- Concatenate (append): create string s_1s_2

  'this is a' . ' string' → 'this is a string'

- Running time
  (assuming we generate a new string)
  - Θ(n+m)

String operations

- Substitute: Exchange all occurrences of a particular character with another character

  Substitute('this is a string' , 'i' , 'x')
  → 'thxs xs a strxng'

  Substitute('banana' , 'a' , 'o') → 'bonono'

- Running time
  - Θ(n)

String operations

- Length: return the number of characters/ symbols in the string

  Length('this is a string') → 16

  Length('this is another string') → 24

- Running time
  - O(1) or Θ(n) depending on implementation

String operations

- Prefix: Get the first j characters in the string

  Prefix('this is a string' , 4) → 'this'

- Running time
  - Θ(j)
- Suffix: Get the last j characters in the string

  Suffix('this is a string' , 6) → 'string'

- Running time
  - Θ(j)

String operations

- Substring – Get the characters between i and j inclusive

  Substring('this is a string' , 4, 8) → 's is '

- Running time
  - Θ(j – i + 1)
- Prefix: Prefix(S, i) = Substring(S, 1, i)
- Suffix: Suffix(S, i) = Substring(S, i+1, length(n))

Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Insertion:

ABACED → ABACCED → DABACCED

Insert 'C'   Insert 'D'

Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Deletion:

ABACED

Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Deletion:

ABACED → BACED

Delete 'A'

Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Deletion:

ABACED → BACED → BACE

Delete 'A'   Delete 'D'

Edit distance (aka Levenshtein distance)

Edit distance between two strings is the minimum number of insertions, deletions and substitutions required to transform string s_1 into string s_2

Substitution:

ABACED → ABADED → ABADES

Sub 'D' for 'C'   Sub 'S' for 'D'

Edit distance examples

Edit(Kitten, Mitten) = 1

Operations:

Sub 'M' for 'K'   Mitten

Edit distance examples

Edit(Happy, Hilly) = 3

Operations:

Sub 'a' for 'i'   Hippy
Sub 'l' for 'p'   Hilpy
Sub 'l' for 'p'   Hilly

Edit distance examples

Edit(Banana, Car) = 5

Operations:

Delete 'B'	anana
Delete 'a'	nana
Delete 'n'	naa
Sub 'C' for 'n'	Caa
Sub 'a' for 'r'	Car

Edit distance examples

Edit(Simple, Apple) = 3

Operations:

Delete 'S'	imple
Sub 'A' for 'i'	Ample
Sub 'm' for 'p'	Apple

Edit distance

Why might this be useful?

Is edit distance symmetric?

- that is, is Edit(s_1, s_2) = Edit(s_2, s_1)?

  Edit(Simple, Apple) =? Edit(Apple, Simple)

- Why?
  - sub 'i' for 'j' → sub 'j' for 'i'
  - delete 'i' → insert 'i'
  - insert 'i' → delete 'i'

Calculating edit distance

X = A B C B D A B

Y = B D C A B A

Ideas?

Calculating edit distance

X = A B C B D A ?

Y = B D C A B ?

After all of the operations, X needs to equal Y

Calculating edit distance

X = A B C B D A ?

Y = B D C A B ?

Operations: Insert
Delete
Substitute

Insert

X = A B C B D A ?

Y = B D C A B ?

Insert

X = $\boxed{\text{A B C B D A ?}}$

Edit

Y = $\boxed{\text{B D C A B}}$?

$$Edit(X,Y) = 1 + Edit(X_{1\ldots n}, Y_{1\ldots m-1})$$

Delete

X = A B C B D A ?

Y = B D C A B ?

Delete

X = $\boxed{\text{A B C B D A}}$?

Edit

Y = $\boxed{\text{B D C A B ?}}$

$$Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m})$$

Substition

X = A B C B D A ?

Y = B D C A B ?

Substition

X = $\boxed{\text{A B C B D A}}$?

Edit

Y = $\boxed{\text{B D C A B}}$?

$$Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$$

Anything else?

X = A B C B D A ?

Y = B D C A B ?

Equal

X = A B C B D A ?

Y = B D C A B ?

Equal

X = $\boxed{\text{A B C B D A}}$?

Edit

Y = $\boxed{\text{B D C A B}}$?

$$Edit(X,Y) = Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$$

Combining results

Insert: $$Edit(X,Y) = 1 + Edit(X_{1\ldots n}, Y_{1\ldots m-1})$$

Delete: $$Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m})$$

Substitute: $$Edit(X,Y) = 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$$

Equal: $$Edit(X,Y) = Edit(X_{1\ldots n-1}, Y_{1\ldots m-1})$$

Combining results

$$Edit(X,Y) = \min \begin{cases} 1 + Edit(X_{1\ldots n}, Y_{1\ldots m-1}) & \text{insertion} \\ 1 + Edit(X_{1\ldots n-1}, Y_{1\ldots m}) & \text{deletion} \\ Diff(x_n, y_m) + Edit(X_{1\ldots n-1}, Y_{1\ldots m-1}) & \text{equal/substitution} \end{cases}$$

```
EDIT(X, Y)
m ← length[X]
n ← length[Y]
for i ← 0 to m
        d[i, 0] ← i
for j ← 0 to n
        d[0, j] ← j
for i ← 1 to m
        for j ← 1 to n
            d[i, j] = min(1 + d[i − 1, j],
                          1 + d[i, j − 1],
                          DIFF(x_i, y_j) + d[i − 1, j − 1])
return d[m, n]
```

Running time

```
EDIT(X, Y)
m ← length[X]
n ← length[Y]
for i ← 0 to m
        d[i, 0] ← i
for j ← 0 to n
        d[0, j] ← j
for i ← 1 to m
        for j ← 1 to n
            d[i, j] = min(1 + d[i − 1, j],
                          1 + d[i, j − 1],
                          DIFF(x_i, y_j) + d[i − 1, j − 1])
return d[m, n]
```

Θ(nm)

Variants

- Only include insertions and deletions
  - What does this do to substitutions?
- Include swaps, i.e. swapping two adjacent characters counts as one edit
- Weight insertion, deletion and substitution differently
- Weight **specific** character insertion, deletion and substitutions differently
- Length normalize the edit distance

String matching

Given a pattern string P of length m and a string S of length n, find **all** locations where P occurs in S

P = ABA

S = DCABABBABABA

String matching

Given a pattern string P of length m and a string S of length n, find **all** locations where P occurs in S

P = ABA

S = DCABABBABABA

Uses

- grep/egrep
- search
- find
- java.lang.String.contains()

Naive implementation

```
NAIVE-STRING-MATCHER(S, P)
  n ← length[S]
  m ← length[P]
  for s ← 0 to n − m
        if S[1...m] = T[s + 1...s + m]
              print "Pattern at s"
```

Is it correct?

```
NAIVE-STRING-MATCHER(S, P)
n ← length[S]
m ← length[P]
for s ← 0 to n − m
        if S[1...m] = T[s + 1...s + m]
                print "Pattern at s"
```

Running time?

```
NAIVE-STRING-MATCHER(S, P)
n ← length[S]
m ← length[P]
for s ← 0 to n − m
        if S[1...m] = T[s + 1...s + m]
                print "Pattern at s"
```

- What is the cost of the equality check?
  - Best case: O(1)
  - Worst case: O(m)

Running time?

```
NAIVE-STRING-MATCHER(S, P)
n ← length[S]
m ← length[P]
for s ← 0 to n − m
        if S[1...m] = T[s + 1...s + m]
                print "Pattern at s"
```

- Best case
  - Θ(n) – when the first character of the pattern does **not** occur in the string
- Worst case
  - O((n-m+1)m)

Worst case

P = AAAA

S = AAAAAAAAAAAAA

Worst case

P = AAAA

S = AAAAAAAAAAAAA

Worst case

P = AAAA

S = AAAAAAAAAAAAA

Worst case

P = AAAA

S = AAAAAAAAAAAAA

repeated work!

Worst case

P = AAAA

S = AAAAAAAAAAAAA

Ideally, after the first match, we'd know to just check the next character to see if it is an 'A'

Patterns

Which of these patterns will have that problem?

P = ABAB

P = ABDC

P = BAA

P = ABBCDDCAABB

Patterns

Which of these patterns will have that problem?

P = ABAB

P = ABDC

P = BAA

P = ABBCDDCAABB

If the pattern has a suffix that is also a prefix then we will have this problem

Finite State Automata (FSA)

- An FSA is defined by 5 components
  - Q is the set of states

q_0 q_1 q_2 . . . q_n

Finite State Automata (FSA)

- An FSA is defined by 5 components
  - Q is the set of states

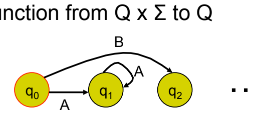
q_0 q_1 q_2 . . . q_n

  - q_0 is the start state
  - $A \subseteq Q$, is the set of accepting states where $|A| > 0$

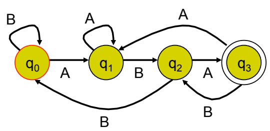
q_7

  - Σ is the alphabet (e.g. {A, B})
  - δ is the transition function from $Q \times \Sigma$ to Q

Q	Σ	Q
q_0	A	q_1
q_0	B	q_2
q_1	A	q_1
. . .		



FSA operation



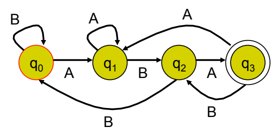
An FSA starts at state q_0 and reads the characters of the input string one at a time.

If the automaton is in state q and reads character a, then it transitions to state $\delta(q,a)$.

If the FSA reaches an accepting state ($q \in A$), then the FSA has found a match.

FSA operation

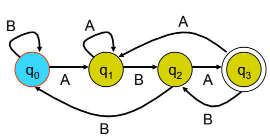
P = ABA



What pattern does this represent?

FSA operation

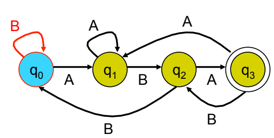
P = ABA



S = BABABBABABA

FSA operation

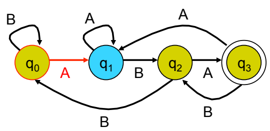
P = ABA



S = BABABBABABA

FSA operation

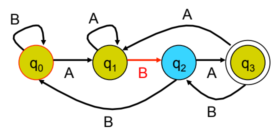
P = ABA



S = BABABBABABA

FSA operation

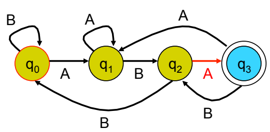
P = ABA



S = BABABBABABA

FSA operation

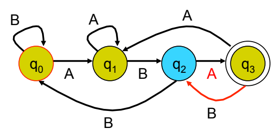
P = ABA



S = BABABBABABA

FSA operation

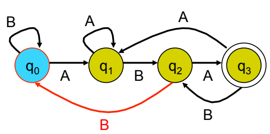
P = ABA



S = BABABBABABA

FSA operation

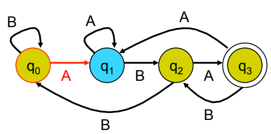
P = ABA



S = BABABBABABA

FSA operation

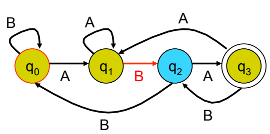
P = ABA



S = BABABBABABA

FSA operation

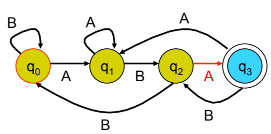
P = ABA



S = BABABBABABA

FSA operation

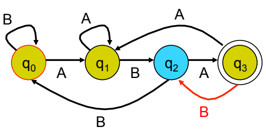
P = ABA



S = BABABBABABA

FSA operation

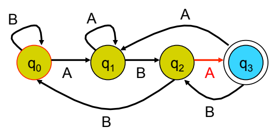
P = ABA



S = BABABBABABA

FSA operation

P = ABA



S = BABABBABABA

Suffix function

- The *suffix function* σ(x,y) is the length of the longest suffix of x that is a prefix of y

$$\sigma(x,y) = \max_i (x_{m-i+1\ldots m} = y_{1\ldots i})$$

σ(abcdab, ababcd) = ?

Suffix function

- The *suffix function* σ(x,y) is the length of the longest suffix of x that is a prefix of y

$$\sigma(x,y) = \max_i (x_{m-i+1\ldots m} = y_{1\ldots i})$$

σ(abcdab, ababcd) = 2

Suffix function

- The *suffix function* σ(x,y) is the index of the longest suffix of x that is a prefix of y

$$\sigma(x,y) = \max_i (x_{m-i+1\ldots m} = y_{1\ldots i})$$

σ(daabac, abacac) = ?

Suffix function

- The *suffix function* σ(x,y) is the length of the longest suffix of x that is a prefix of y

$$\sigma(x,y) = \max_i (x_{m-i+1\ldots m} = y_{1\ldots i})$$

σ(daabac, abacac) = 4

Suffix function

- The *suffix function* σ(x,y) is the length of the longest suffix of x that is a prefix of y

$$\sigma(x,y) = \max_i (x_{m-i+1\ldots m} = y_{1\ldots i})$$

σ(dabb, abacd) = ?

Suffix function

- The *suffix function* σ(x,y) is the length of the longest suffix of x that is a prefix of y

$$\sigma(x,y) = \max_i (x_{m-i+1\ldots m} = y_{1\ldots i})$$

σ(dabb, abacd) = 0

Building a string matching automata

- Given a pattern $P = p_1, p_2, \ldots, p_m$, we'd like to build an FSA that recognizes P in strings

P = ababaca

Ideas?

Building a string matching automata

P = ababaca

- $Q = q_1, q_2, \ldots, q_m$ corresponding to each symbol, plus a q_0 starting state
- the set of accepting states, $A = \{q_m\}$
- vocab Σ all symbols in P, plus one more representing all symbols not in P
- The transition function for $q \in Q$ and $a \in \Sigma$ is defined as:
  - $\delta(q, a) = \sigma(p_{1\ldots q}a, P)$

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1\ldots q}a, P)$

state	a	b	c	P
q_0	?			a
q_1				b
q_2				a
q_3				b
q_4				a
q_5				c
q_6				a
q_7				

σ(a, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1\ldots q}a, P)$

state	a	b	c	P
q_0	1	?		a
q_1				b
q_2				a
q_3				b
q_4				a
q_5				c
q_6				a
q_7				

σ(b, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	?	a
q_1				b
q_2				a
q_3				b
q_4				a
q_5				c
q_6				a
q_7				

σ(b, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1				b
q_2				a
q_3				b
q_4				a
q_5				c
q_6				a
q_7				

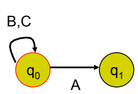
σ(b, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1				b
q_2				a
q_3				b
q_4				a
q_5				c
q_6				a
q_7				



Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	?			b
q_4				a
q_5				c
q_6				a
q_7				

We've seen 'aba' so far

σ(abaa, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	1			b
q_4				a
q_5				c
q_6				a
q_7				

We've seen 'aba' so far

σ(abaa, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	1	4	0	b
q_4	5	0	0	a
q_5	1	?		c
q_6				a
q_7				

We've seen 'ababa' so far

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	1	4	0	b
q_4	5	0	0	a
q_5	1	?		c
q_6				a
q_7				

We've seen 'ababa' so far

σ(ababab, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	1	4	0	b
q_4	5	0	0	a
q_5	1	4		c
q_6				a
q_7				

We've seen 'ababa' so far

σ(ababab, ababaca)

Transition function

P = ababaca

- $\delta(q, a) = \sigma(p_{1...q}a, P)$

state	a	b	c	P
q_0	1	0	0	a
q_1	1	2	0	b
q_2	3	0	0	a
q_3	1	4	0	b
q_4	5	0	0	a
q_5	1	4	6	c
q_6	7	0	0	a
q_7	1	2	0	

Matching runtime

- Once we've built the FSA, what is the runtime?
  - $\Theta(n)$ - Each symbol causes a state transition and we only visit each character once
- What is the cost to build the FSA?
  - How many entries in the table?
    - $\Omega(m|\Sigma|)$
  - How long does it take to calculate the suffix function at each entry?
    - Naïve: $O(m)$
  - Overall naïve: $O(m^2|\Sigma|)$
  - Overall fast implementation $O(m|\Sigma|)$

Rabin-Karp algorithm

- Use a function T to that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = ABA

S = BABABBABABA

Rabin-Karp algorithm

- Use a function T to that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = ABA Hash P

T(P)

S = BABABBABABA

Rabin-Karp algorithm

- Use a function T to that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = ABA

S = BABABBABABA

Hash m symbol sequences and compare

T(BAB)
=
T(P)

Rabin-Karp algorithm

- Use a function T to that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = ABA

match

S = BABABBABABA

Hash m symbol sequences and compare

T(ABA)
=
T(P)

Rabin-Karp algorithm

- Use a function T to that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = ABA

S = BABABBABABA

Hash m symbol sequences and compare

T(BAB)
=
T(P)

Rabin-Karp algorithm

- Use a function T to that computes a numerical representation of P
- Calculate T for all m symbol sequences of S and compare

P = ABA

S = BABABBABABA

Hash m symbol sequences and compare

…

T(BAB)
=
T(P)

Rabin-Karp algorithm

P = ABA

For this to be useful/ efficient, what needs to be true about T?

S = BABABBABABA

$\underbrace{\text{BAB}}$ …

T(BAB)
=
T(P)

Rabin-Karp algorithm

P = ABA

For this to be useful/ efficient, what needs to be true about T?

Given $T(s_{i...i+m-1})$ we must be able to efficiently calculate $T(s_{i+1...i+m})$

S = BABABBABABA

$\underbrace{\text{BAB}}$ …

T(BAB)
=
T(P)

Calculating the hash function

- For simplicity, assume Σ = (0, 1, 2, …, 9). (in general we can use a base larger than 10).
- A string can then be viewed as a decimal number
- How do we efficiently calculate the numerical representation of a string?

T('9847261') = ?

Horner's rule

$$T(p_{1...m}) = p_m + 10(p_{m-1} + 10(p_{m-2} + ... + 10(p_2 + 10p_1)))$$

9847261

9 * 10 = 90

(90 + 8)*10 = 980

(980 + 4)*10 = 9840

(9840 + 7)*10 = 98470

… = 9847621

Horner's rule

$$T(p_{1...m}) = p_m + 10(p_{m-1} + 10(p_{m-2} + ... + 10(p_2 + 10p_1)))$$

9847261

9 * 10 = 90

(90 + 8)*10 = 980

(980 + 4)*10 = 9840

(9840 + 7)*10 = 98470

… = 9847621

Running time?

Θ(m)

Calculating the hash on the string

- Given $T(s_{i...i+m-1})$ how can we efficiently calculate $T(s_{i+1...i+m})$?

m = 4

$\underbrace{9638}_{T(s_{i...i+m-1})}01572348267$

$$T(s_{i+1...i+m}) = 10(T(s_{i...i+m-1}) - 10^{m-1}s_i) + s_{i+m}$$

Calculating the hash on the string

- Given $T(s_{i...i+m-1})$ how can we efficiently calculate $T(s_{i+1...i+m})$?

m = 4

801

$\underbrace{9638}_{T(s_{i...i+m-1})}01572348267$

subtract highest order digit

$$T(s_{i+1...i+m}) = 10(T(s_{i...i+m-1})\boxed{-10^{m-1}s_i}) + s_{i+m}$$

Calculating the hash on the string

- Given $T(s_{i...i+m-1})$ how can we efficiently calculate $T(s_{i+1...i+m})$?

m = 4

8010

$\underbrace{9638}_{T(s_{i...i+m-1})}01572348267$

shift digits up

$$T(s_{i+1...i+m}) = \boxed{10}(T(s_{i...i+m-1}) - 10^{m-1}s_i) + s_{i+m}$$

Calculating the hash on the string

- Given $T(s_{i...i+m-1})$ how can we efficiently calculate $T(s_{i+1...i+m})$?

m = 4

8015

963801572348267

$\underbrace{9638}_{T(s_{i...i+m-1})}$

add in the lowest digit

$$T(s_{i+1...i+m}) = 10(T(s_{i...i+m-1}) - 10^{m-1}s_i) + \boxed{s_{i+m}}$$

Calculating the hash on the string

- Given $T(s_{i...i+m-1})$ how can we efficiently calculate $T(s_{i+1...i+m})$?

m = 4

963801572348267

$\underbrace{9638}_{T(s_{i...i+m-1})}$

Running time?

- $\Theta(m)$ for $s_{1...m}$
- $O(1)$ for the rest

$$T(s_{i+1...i+m}) = 10(T(s_{i...i+m-1}) - 10^{m-1}s_i) + s_{i+m}$$

Algorithm so far…

- Is it correct?
  - Each string has a unique numerical value and we compare that with each value in the string
- Running time
  - Preprocessing:
    - $\Theta(m)$
  - Matching
    - $\Theta(n-m+1)$

Is there any problem with this analysis?

Algorithm so far…

- Is it correct?
  - Each string has a unique numerical value and we compare that with each value in the string
- Running time
  - Preprocessing:
    - $\Theta(m)$
  - Matching
    - $\Theta(n-m+1)$

How long does the check $T(P) = T(s_{i...i+m-1})$ take?

Modular arithmetics

- The run time assumptions we made were assuming arithmetic operations were constant time, which is not true for large numbers
- To keep the numbers small, we'll use modular arithmetics, i.e. all operations are performed mod q
  - a+b = (a+b) mod q
  - a*b = (a*b) mod q
  - …

Modular arithmetics

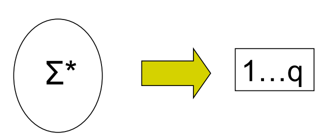
- If T(A) = T(B), then T(A) mod q = T(B) mod q
  - In general, we can apply mods as many times as we want and we will not effect the result
- What is the downside to this modular approach?
  - Spurious hits: if T(A) mod q = T(B) mod q that does **not** necessarily mean that T(A) = T(B)
  - If we find a hit, we must check that the actual string matches the pattern

Runtime

- Preprocessing
  - $\Theta(m)$
- Running time
  - Best case:
    - $\Theta(n-m+1)$ – No matches and no spurious hits
  - Worst case
    - $\Theta((n-m+1)m)$

Average case running time

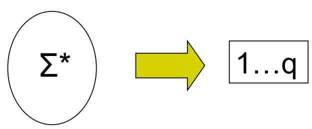
- Assume v valid matches in the string
- What is the probability of a spurious hit?
  - As with hashing, assume a uniform mapping onto values of q:



  - What is the probability under this assumption?

Average case running time

- Assume v valid matches in the string
- What is the probability of a spurious hit?
  - As with hashing, assume a uniform mapping onto values of q:



  - What is the probability under this assumption? 1/q

Average case running time

- How many spurious hits?
  - n/q
- Average case running time:

$$\underbrace{O(n-m+1)}_{\text{iterate over the positions}} + \underbrace{O(m(v+n/q))}_{\text{checking matches and spurious hits}}$$

Matching running times

Algorithm	Preprocessing time	Matching time
Naïve	0	O((n-m+1)m)
FSA	Θ(m|Σ|)	Θ(n)
Rabin-Karp	Θ(m)	O(n)+O(m(v+n/q))
Knuth-Morris-Pratt	Θ(m)	Θ(n)