

Computer Science 1001.py, Lecture 19

String Matching

Instructors: Daniel Deutch, Amiram Yehudai

Teaching Assistants: Michal Kleinbort, Amir Rubinstein

School of Computer Science

Tel-Aviv University

Fall Semester, 2014-15

<http://tau-cs1001-py.wikidot.com>

String Matching

When we edit a text document we continuously modify a string. In addition to typing, the next most common operation while editing is probably **search**, *i.e.* looking for a (relatively short) **pattern** within a (typically much longer) **text**. This problem is known as **string matching**.

In this lecture we explore four very different algorithmic approaches to string matching.

Exact String Matching: Definition

Input: Two strings over a fixed alphabet, Σ .

A **text** of length n , $T = T[0..n - 1]$, and
a shorter **pattern** of length m , $P = P[0..m - 1]$.

Goal: Find all indices, or shifts, s , in the text ($0 \leq s \leq n - m$)
such that $T[s..s + m - 1] = P[0..m - 1]$.

That is, all shifts where the pattern matches the text **exactly**.

This is a fundamental problem in **stringology**. We will examine **four**
approaches to solving it, bearing the associated complexity in mind.

1. The **naïve**, exhaustive, approach.
2. An approach based on **hashing**.
3. A **randomized** (coin flipping) algorithm, due to Karp and Rabin.
4. A DFA-based solution (**Only briefly**)

The Naïve Algorithm

Goal: Find all indices, or shifts, s , in the text ($0 \leq s \leq n - m$) such that $T[s..s + m - 1] = P[0..m - 1]$.

Method: For every **shift**, s , $0 \leq s \leq n - m$, compare the two strings $T[s..s + m - 1]$ and $P[0..m - 1]$, character by character. If they are equal, report s (and continue).

Time complexity: For each shift, s , we compare two blocks of m characters. This takes m operations. The number of shifts is $n - m + 1$.

So the overall cost is $m \cdot (n - m + 1) = O(m \cdot n)$.

An Algorithm Based on Hashing

Goal: Find all indices, or shifts, s , in the text ($0 \leq s \leq n - m$) such that $T[s..s + m - 1] = P[0..m - 1]$.

Initial Step: Compute $\text{hash}(P[0..m-1])$. Denote the outcome by t .

Method: For every shift, s , $0 \leq s \leq n - m$, compute $\text{hash}(T[s..s+m-1])$. If this equals t , report s (and continue).

Time complexity: For each shift, s , we compute and compare the hash of an m long substring. This takes $O(m)$ operations. The number of shifts is $n - m + 1$. So the overall cost is $m \cdot (n - m + 1) = O(m \cdot n)$. This is **no better** than the naïve algorithm!

An Algorithm Based on Hashing: Observations

Initial Step: Compute $\text{hash}(P[0..m-1])$. Denote the outcome by t .

Method: For every **shift**, s , $0 \leq s \leq n - m$, compute $\text{hash}(T[s..s+m-1])$. If this equals t , report s (and continue).

Note that we use just the **hash function** here, and not **hash tables**.

A variant: Suppose our task is not just finding matches for one pattern, but of **many patterns** with the **same length**, m . Then a sensible preprocessing would be to first compute all hash values of text windows with length m . Then hash each pattern, one by one, and look up all the windows with the same hash value as the current pattern.

Representation of Characters: Unicode (reminder)

Unicode is a **variable length encoding**: Different characters can be encoded using a different number of bits. For example, ASCII characters are encoded using one byte per character, and are **compatible** with the 7 bits ASCII encoding (a leading zero is added). Hebrew letters' encodings, for example, are in the range 1488 (hex 05D0, \aleph) to 1514 (hex 05EA, tav , unfortunately unknown to \LaTeX), reflecting the $22+5=27$ letters in the Hebrew alphabet.

Python employs Unicode. The built in function `ord` returns the Unicode encoding of a (single) character (in decimal). The `chr` of an encoding returns the corresponding character.

```
>>> ord(" ")      # space
32
>>> ord("a")
97
>>> hex(ord("a"))
0x61
>>> chr(97)
a
```

Karp–Rabin's Algorithm: A Bird's Eye View

Goal: Find all indices, or shifts, s , in the text ($0 \leq s \leq n - m$) such that $T[s..s + m - 1] = P[0..m - 1]$.

Initial Step: Compute $\text{finger_print}(P[0..m-1])$. Denote the outcome by t .

Method: For every shift, s , $0 \leq s \leq n - m$, compute $\text{finger_print}(T[s..s+m-1])$. If this equals t , report s (and continue).

Key Idea: Given the finger print of one window, $\text{finger_print}(T[s..s+m-1])$, it is possible to compute the finger print of the next one, $\text{finger_print}(T[s+1..s+m])$ in a **constant** number of operations.

Karp–Rabin's Algorithm: A Bird's Eye View, cont.

Key Idea: Given the finger print of one window, $\text{finger_print}(T[s..s+m-1])$, it is possible to compute the finger print of the next one, $\text{finger_print}(T[s+1..s+m])$ in a **constant** number of operations.

Time complexity: $O(m)$ operations to compute the two finger prints $\text{finger_print}(P[0..m-1])$ and $\text{finger_print}(T[0..m-1])$.

$O(m)$ operations for both, Additional $n - m - 1$ finger prints to compute, each taking $O(1)$ operations. Total cost is $O(n) + O(m) = O(n + m)$.

Caveat: The algorithm is **randomized**. Computations are done modulo a **random prime** of moderate size (say 30 to 32 bits long). There is a **small** but **non negligible** probability of **false positive errors** (announcing a match where there isn't one).

False negative errors never occur: If there is a match, it will always be announced.

Karp–Rabin, Step 1: From Strings to Integers

Let $P[0..m-1]$ be an m character long string (the pattern).

Let p_i be the **unicode representation** of the character $P[i]$.

Recall that 16 bits suffice to represent all reasonable characters.

So for all i ($0 \leq i \leq m-1$), $0 \leq p_i < B$, where $B = 2^{16}$.

We transform $P[0..m-1]$ into a base B number, denoted by N_P :

$$N_P = p_0 B^{m-1} + p_1 B^{m-2} + \dots + p_{m-3} B^2 + p_{m-2} B + p_{m-1}.$$

This sum can be computed efficiently, using Hörner's rule,

$$N_P = (((((\dots ((p_0 \cdot B) + p_1) \cdot B) + \dots + p_{m-3}) \cdot B) + p_{m-2}) \cdot B) + p_{m-1}$$

Arithmetization

$$N_P = (((((\dots ((p_0 \cdot B) + p_1) \cdot B) + \dots + p_{m-3}) \cdot B) + p_{m-2}) \cdot B) + p_{m-1} \cdot$$

This requires m additions and m multiplications.

The outcome is of size $\approx B^m$.

The mapping from the pattern, $P[0..m-1]$, into N_P , is one to one. So N_P uniquely determines the pattern.

We can think of the transformation from the pattern P (a string) to N_P (a number) as an **arithmetization** of the string.

From Strings to Integers: Karp–Rabin, Step 2

Let $T[0..n-1]$ be an n character long string (the text).

Let t_i be the **unicode representation** of the character $T[i]$.

Like before, for every **shift**, s , $0 \leq s \leq n - m$, we transform the m character long “shift” $T[s..s+m-1]$ into base B number, denoted by T_s :

$$T_s = t_s B^{m-1} + t_{s+1} B^{m-2} + \dots + t_{s+m-3} B^2 + t_{s+m-2} B + t_{s+m-1}.$$

Once we got N_P and all of T_0, T_1, \dots, T_{n-m} , the string matching problem reduces to the question of finding all indices i for which $N_P = T_i$.

From Strings to Integers: Python Code

The following function computes the arithmetization of a single text:

```
def arithmetize(text, basis=2**16):
    """ convert substring to number using basis powers
        employs Horner rule """
    partial_sum=0
    for ch in text:
        partial_sum = partial_sum*basis+ord(ch) # Horner
    return partial_sum

>>> arithmetize("b")
98
>>> arithmetize("be")
6422629
>>> arithmetize("ben")
420913414254
>>> ((ord("b")*2**16)+ord("e"))*2**16+ord("n") # sanity check
420913414254
```

From Strings to Integers 2: Python Code

The following function computes the arithmetization of all length `m` contiguous substrings in the `text`:

```
def arithmetize_text(text,m,basis=2**16):
    """ computes arithmization of all m long substrings
        of text, using basis powers """
    t=[]
    #will store list of numbers representing consecutive substrings
    for s in range(0,len(text)-m+1):
        t=t+[arithmetize(text[s:s+m],basis)]
        # append the next number to existing t
    return t
```

From Strings to Integers 2: Running the Code

```
>>> arithmetize_text("speak softly, and carry a big stick",1)
[115, 112, 101, 97, 107, 32, 115, 111, 102, 116, 108, 121, 44, 32,
97, 110, 100, 32, 99, 97, 114, 114, 121, 32, 97, 32, 98, 105, 103,
32, 115, 116, 105, 99, 107]
```

```
>>> arithmetize_text("speak softly, and carry a big stick",2)
[7536752, 7340133, 6619233, 6357099, 7012384, 2097267, 7536751,
7274598, 6684788, 7602284, 7078009, 7929900, 2883616, 2097249,
6357102, 7209060, 6553632, 2097251, 6488161, 6357106, 7471218,
7471225, 7929888, 2097249, 6357024, 2097250, 6422633, 6881383,
6750240, 2097267, 7536756, 7602281, 6881379, 6488171]
```

We now [slightly change the basis](#)

```
>>> arithmetize_text("speak softly, and carry a big stick",2,
    basis=2**16-1)
[7536637, 7340021, 6619132, 6357002, 7012277, 2097235, 7536636,
7274487, 6684686, 7602168, 7077901, 7929779, 2883572, 2097217,
6357005, 7208950, 6553532, 2097219, 6488062, 6357009, 7471104,
7471111, 7929767, 2097217, 6356927, 2097218, 6422535, 6881278,
6750137, 2097235, 7536641, 7602165, 6881274, 6488072]
```

From Strings to Integers 2: Efficiency Considerations

Once we got N_P and all of T_0, T_1, \dots, T_{n-m} , the string matching problem reduces to the question of finding all indices i for which $N_P = T_i$.

But how many operations are needed to compute T_0, T_1, \dots, T_{n-m} ?

Every T_i takes $O(m)$ multiplications/additions to compute. So all by all this takes $O(m \cdot (n - m + 1))$ operations.

No better than the naïve algorithm!

(in fact even worse, as the naïve algorithm requires just single characters' comparisons, while here we got fairly involved arithmetic.)

From Strings to Integers Efficiently

How many operations are **really** needed to compute

$$T_0, T_1, \dots, T_{n-m}?$$

Key observation: Computing each T_i **from scratch** is rather **wasteful**.

Suppose we have already computed

$$T_s = t_s B^{m-1} + t_{s+1} B^{m-2} + \dots + t_{s+m-3} B^2 + t_{s+m-2} B + t_{s+m-1} \quad .$$

$$\begin{aligned} T_{s+1} &= t_{s+1} B^{m-1} + t_{s+2} B^{m-2} + \dots + t_{s+m-2} B^2 + t_{s+m-1} B + t_{s+m} \\ &= (T_s - t_s B^{m-1}) B + t_{s+m}. \end{aligned}$$

Once T_s is given, we can thus compute T_{s+1} by using just **one** subtraction, **one** addition, and **two** multiplications – $O(1)$ operations overall. (We will precompute B^{m-1} once, and reuse it.)

From Strings to Integers Efficiently – Python Code

```
def arithmetize_text2(text,m,basis=2**16):
    """ efficiently computes arithmetization of all m long
        substrings of text, using basis powers """
    b_power=basis**(m-1)
    t=[arithmetize(text[0:m],basis)]
    # t[0] equals first word arithmetization
    for s in range(1,len(text)-m+1):
        new_number=(t[s-1]-ord(text[s-1])*b_power)*basis
            +ord(text[s+m-1])
        t=t+[new_number] # append new_number to existing t
    return(t)
# t stores list of numbers representing m long words of text
```

From Strings to Integers Efficiently – Running the Code

```
>>> arithmetize_text2("speak softly, and carry a big stick",1)
[115, 112, 101, 97, 107, 32, 115, 111, 102, 116, 108, 121, 44, 32,
97, 110, 100, 32, 99, 97, 114, 114, 121, 32, 97, 32, 98, 105, 103,
32, 115, 116, 105, 99, 107]
```

```
>>> arithmetize_text2("speak softly, and carry a big stick",2)
[7536752, 7340133, 6619233, 6357099, 7012384, 2097267, 7536751,
7274598, 6684788, 7602284, 7078009, 7929900, 2883616, 2097249,
6357102, 7209060, 6553632, 2097251, 6488161, 6357106, 7471218,
7471225, 7929888, 2097249, 6357024, 2097250, 6422633, 6881383,
6750240, 2097267, 7536756, 7602281, 6881379, 6488171]
```

Same same, but **more efficient**.

From Strings to Integers Efficiently – Python Code 2

This code first `arithmetize`: Converts the pattern to a natural number, and converts every substring of the text whose length equals the pattern's length to a natural number. It then determines the shifts where the `numbers are the same`, which are the shifts where the substrings equal the pattern.

```
def find_matches(pattern, text, basis=2**16):
    """ find all occurrences of pattern in text
        using efficient arithmetization of text """
    assert(len(pattern) <= len(text))
    p=arithmetize(pattern, basis)
    t=arithmetize_text2(text, len(pattern), basis)
    matches=[]
    for s in range(len(t)):
        if p==t[s]: matches=matches+[s]
    return(matches)

>>> find_matches("bar", "bennyXbirburbirbarYraniZbarbarossa")
[15, 24, 27]
```

Same same, but `more efficient`.

Are We Really Being Efficient Now?

Computing N_P, T_0 and B^m initially takes $O(m)$ operations.
All other T_s will take $O(n - m + 1)$ operations (together) to compute.
And finally, we have $n - m + 1$ comparisons: N_P vs. all T_s .
Overall $O(n + m)$ operations, which is **optimal**.

Caveat: N_P and the various T_s are **not** constant size numbers. They probably won't fit in a constant number of computer words either. So operations over them are, in practice, costly (though we usually do not count this as part of our complexity analysis..) To remedy this, we utilize two important notions in an improved algorithm:

1. Probabilistic (coin flipping) algorithms (**saw this notion earlier**).
2. Finger printing (**new, yet closely related to hashing**).

A Probabilistic (Coin Flipping) Algorithm

Method:

1. Compute the number $p = \text{fingerprint}(N_P)$.
2. For every shift, s , $0 \leq s \leq n - m$, compute the number $f_s = \text{fingerprint}(T_s)$.
3. For every shift, s , $0 \leq s \leq n - m$, compare the number f_s with p .
4. If $p = f_s$, declare the shift, s , as a possible match (and continue).

What the #*&% is going on here? What are these fingerprints?

Fingerprints

What the #*&% is going on here? What are these **fingerprints**?

In the “real world”, fingerprints are used for identifying (human) individuals. Kind of a “**compact representation**” of people.

Fingerprints

What the #*&% is going on here? What are these fingerprints?

Inspired by the real world, a fingerprint F_r in our string matching context is a mapping from strings to fixed size numbers in the range $[0..k]$ (say 32 bits, namely $2^{31} \leq k < 2^{32}$).

If $S_1 \neq S_2$ are two different strings, we require that

$$\Pr(F_r(S_1) = F_r(S_2)) < 2^{-30}.$$

I.e., the probability that the fingerprint maps two different strings to the same number is small (smaller than twice the inverse of the target space size).

Wait a Minute

$F_r(\cdot)$ maps patterns, namely strings over Σ^m , to numbers in $[0..k]$. But the domain (space of possible patterns) is typically **much larger** than the target (numbers in $[0..k]$). So $F_r(\cdot)$ surely is **not one to one**. There must surely be $S_1 \neq S_2$ such that

$$F_r(S_1) = F_r(S_2) .$$

So how can we require

$$\Pr(F_r(S_1) = F_r(S_2)) < 2/k ?$$

You were probably told you should always **read the small print**.

In our case, the **small print** is the subscript r above.

What is the subscript r in the mapping F_r ?

Indeed, any single function $F(\cdot)$ would have collisions, namely $S_1 \neq S_2$ such that $F(S_1) = F(S_2)$.

For this reason, we do not use a **single** function $F(\cdot)$.

Instead, we have a **collection** of many functions, $F_r(\cdot)$, indexed by the parameter, r .

When we require

$$\Pr(F_r(S_1) = F_r(S_2)) < 2^{-30}$$

The **probability** is over **the choice of r** .

If $S_1 \neq S_2$, then for **most** choices of r , $F_r(S_1) \neq F_r(S_2)$, while for a few choices, we could have equality.

Karp-Rabin Fingerprinting

Pick a **random prime**, r , in the range $2^{31} < r < 2^{32}$ (a 32-bit random prime).

The corresponding fingerprint function is $F_r(m) = m \bmod r$.

Details:

1. Compute the number $p = \text{fingerprint}(N_P)$.
2. For every **shift**, s , $0 \leq s \leq n - m$, **compute** the number $f_s = \text{fingerprint}(T_s)$.
3. For every **shift**, s , $0 \leq s \leq n - m$, **compare** the number f_s with p .
4. If $p = f_s$, declare the shift, s , as a **possible match** (and continue).

Karp-Rabin Fingerprints: Details

- ▶ Had we computed N_P, B^m and T_0, \dots, T_{n-m} first, and then do the **modular reduction, mod r** , we would have to deal with large numbers (unbounded sizes).
- ▶ Instead, we employ the **mod r** computations initially: In computing N_P, B^m, T_0 , and when going from T_s to T_{s+1} . The intermediate numbers involved are never larger than $r^2 < 2^{64}$.
- ▶ One can show that if $T_s \neq N_P$, then indeed $\text{Prob}_r(T_s \bmod r = N_P \bmod r) < 2^{-30}$.
Proof (omitted) uses density of primes, the so called **Chinese remainder theorem**, and some simple arithmetic.

Karp-Rabin Fingerprints: False vs. True Matches

- ▶ If $p = f_s$, the algorithm declares the shift, s , as a **possible match**.
- ▶ Is this a **true match**, namely $P[0..m - 1] = T[s..s + m - 1]$,
- ▶ Or a **false match**, namely $P[0..m - 1] \neq T[s..s + m - 1]$?

We face a **choice** here. We could try and verify all matches, at the cost of additional operations.

Or we could “trust” the announced matches without exploring further, risking **errors**.

For “random strings” (whatever this means has to be clarified, and will typically **not** apply to the text you are reading or writing), matches will be rare and thus verifying them, even exhaustively, will not cause an unacceptable overhead.

Karp-Rabin Fingerprints: A Dilemma

For “random strings” (whatever this means has to be clarified, and will typically **not** apply to the text you are reading or writing), matches will be rare and thus verifying them, even exhaustively, will not cause an unacceptable overhead.

But suppose the pattern and the text consist of just **A**s. Then **every shift** will yield a **true match**. But verifying all matches will cost $O(m(n - m + 1))$ operations, an expression we have learnt to **hate**.

So in our case, randomization brought with it not only an improved efficiency, but also a **dilemma**, of **cost vs. accuracy**.

Karp-Rabin Fingerprints: Python Code

We will use here the parameters $B = 2^{16}$ (the basis for the arithmetization) and $r = 2^{32} - 3 = 4294967293$ (the prime number used for remainder computations). Recall that r should be picked at random **every time we employ Karp-Rabin**.

```
def fingerprint(text, basis=2**16, r=2**32-3):  
    """ used to compute karp-rabin fingerprint of the pattern  
    employs Horner method (modulo r) """  
    partial_sum=0  
    for ch in text:  
        partial_sum=(partial_sum*basis+ord(ch)) % r  
    return partial_sum
```

Karp-Rabin Fingerprints: Python Code (cont.)

```
def text_fingerprint(text,m,basis=2**16,r=2**32-3):
    """ used to computes karp-rabin fingerprint of the text """
    f=[]
    b_power=pow(basis,m-1,r)
    list.append(f, fingerprint(text[0:m],basis,r))
    # f[0] equals first text fingerprint
    for s in range(1,len(text)-m+1):
        new_fingerprint=((f[s-1]-ord(text[s-1])*b_power)*basis
                        +ord(text[s+m-1])) % r
        # compute f[s], based on f[s-1]
    list.append(f,new_fingerprint)# append f[s] to existing f
    return f
```

Karp-Rabin Fingerprints: Python Code (cont. 2)

```
def find_matches_KR(pattern, text, basis=2**16, r=2**32-3):
    """ find all occurrences of pattern in text
        using coin flipping Karp-Rabin algorithm """
    if len(pattern) > len(text):
        return []
    p=fingerprint(pattern, basis, r)
    f=text_fingerprint(text, len(pattern), basis, r)
    matches = [s for s, f_s in enumerate(f) if f_s == p]
    # list comprehension
    return matches
```

Note: `enumerate(lst)`, returns a list of the same length, where the *i*-th element is the pair `(i, lst[i])`. For example

```
>>> lst=["a","b","c","d","e"]
>>> enumerate(lst)
<enumerate object at 0x170d1c0>
>>> list(enumerate(lst))
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

Running the Code – Three Samples

```
>>> pattern="aaaaaa"
>>> text="aaaaaaaaaaaaaaaaaaaaaaaaaaaa"
>>> find_matches_KR(pattern,text)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
 12, 13, 14, 15, 16, 17, 18, 19]

>>> text="aaaaaaaaaaaaabbbaaaaaaaaaa"
>>> find_matches_KR(pattern,text)
[0, 1, 2, 3, 4, 5, 6, 7, 16, 17, 18, 19]

>>> find_matches_KR(text , pattern)
[]
```

Running the Code – a Fourth Sample

```
>>> text="""Twas brillig, and the slithy toves Did gyre and gimble
in the wabe: All mimsy were the borogoves, And the mome
raths outgrabe. Beware the Jabberwock, my son! The jaws that bite,
the claws that catch! Beware the Jubjub bird, and shun The frumious
Bandersnatch! He took his vorpal sword in hand: Long time the
manxome foe he sought      So rested he by the Tumtum tree, And
stood awhile in thought. And, as in uffish thought he stood, The
Jabberwock, with eyes of flame, Came whiffing through the tulgey
wood, And burbled as it came! One, two! One, two! And through and
through The vorpal blade went snicker-snack! He left it dead, and
with its head He went galumphing back. And, has thou slain the
Jabberwock? Come to my arms, my beamish boy! O frabjous day!
Callooh! Callay! He chortled in his joy. Twas brillig, and the
slithy toves Did gyre and gimble in the wabe; All mimsy were the
borogoves, And the mome raths outgrabe."""
# quotation marks removed from original
>>> len(text)
923
>>> pattern="gyre and gimble"
>>> find_matches_KR(pattern,text)
[39, 836]
```

Running the Code – Changing the Moduli Wrecklessly

```
>>> text = "Twas brillig, and the slithy toves Did gyre and gimble  
in the wabe: All mimsy were the borogoves, And the mome raths  
outgrabe. Beware the Jabberwock, my son! The jaws that bite,  
the claws that catch! Beware the Jubjub bird, and shun The frumious  
Bandersnatch! He took his vorpal sword in hand: Long time the  
manxome foe he sought      So rested he by the Tumtum tree, And  
stood awhile in thought. And, as in uffish thought he stood, The  
Jabberwock, with eyes of flame, Came whiffling through the tulgey  
wood, And burbled as it came! One, two! One, two! And through and  
through The vorpal blade went snicker-snack! He left it dead, and  
with its head He went galumphing back. And, has thou slain the  
Jabberwock? Come to my arms, my beamish boy! O frabjous day!  
Callooh! Callay! He chortled in his joy. Twas brillig, and the  
slithy toves Did gyre and gimble in the wabe; All mimsy were the  
borogoves, And the mome raths outgrabe."
```

```
# quotation marks removed from original
```

```
>>> pattern="gyre and gimble"
```

```
>>> find_matches_KR(pattern,text)
```

```
[39, 836]
```

```
>>> find_matches_KR(pattern,text,r=97) # small prime moduli
```

```
[6, 39, 435, 567, 644, 654, 666, 785, 803, 836]
```

```
>>> find_matches_KR(pattern,text,r=2**32)
```

```
# large composite moduli w/ small factors
```

```
[39, 366, 501, 601, 768, 836]
```

Deterministic Finite Automata (DFA)

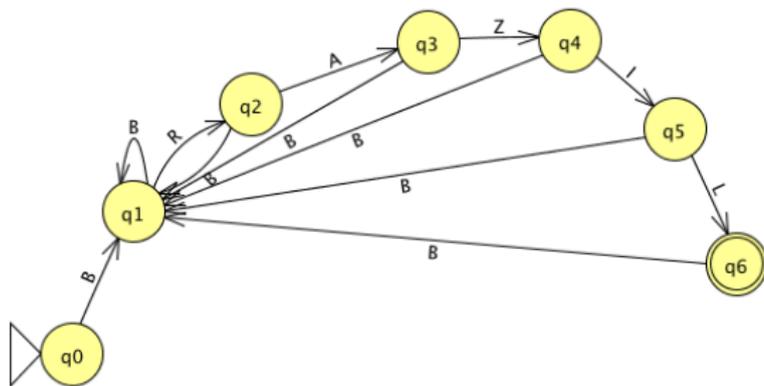
- ▶ A **DFA** is one of the simplest models of computation.
- ▶ It consists of a finite number, m , of **states**, usually denoted q_0, \dots, q_{m-1} .
- ▶ States are viewed as nodes (denoted by circles) in a **directed graph**.
- ▶ **Directed edges** in the graph are labeled by single **letters** from the alphabet Σ .
- ▶ The DFA has distinguished **start state** (denoted by an incoming triangle) and **accept state(s)** (denoted by a double circle).

Deterministic Finite Automata (cont.)

- ▶ The DFA has distinguished **start state** (denoted by an incoming triangle) and **accept state(s)** (denoted by a double circle).
- ▶ On input that is a string of letters over Σ , the DFA starts at the start node.
- ▶ It “reads” one letter at a time, proceeding according to the arrows.
- ▶ **Deterministic** means every new character corresponds to **just one** outgoing arrow.
- ▶ There are also **non deterministic** automata, where a new character could correspond to **more than one** outgoing arrow.
- ▶ **Non deterministic** automata are an important computational model, but not within our scope. They will be discussed in, ahm, the **computational model course**.

The BRAZIL Pattern DFA

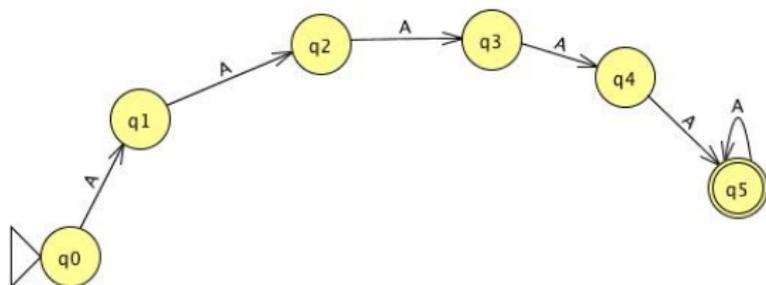
- ▶ We describe specific DFAs, each corresponding to a different **pattern**.



- ▶ States with no outgoing edges for some letter implicitly imply an outgoing edge **to the initial state, q_0** , with that label.
- ▶ Each such DFA will then “go over” a **text** string, and tell us what it “thinks” of the text. **Red** letter indicates accept state reached.
- ▶ First text: **BRIBROBRABRAZILTURKEY**.
- ▶ Second text: **BRAZILUZIPHERBRAZILYOO**

The AAAAA Pattern DFA

- ▶ We describe specific DFAs, each corresponding to a different



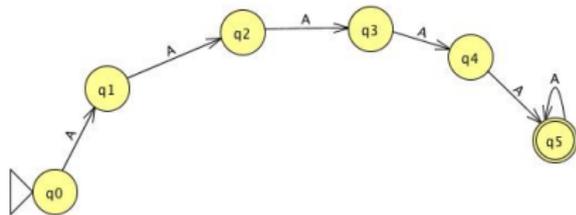
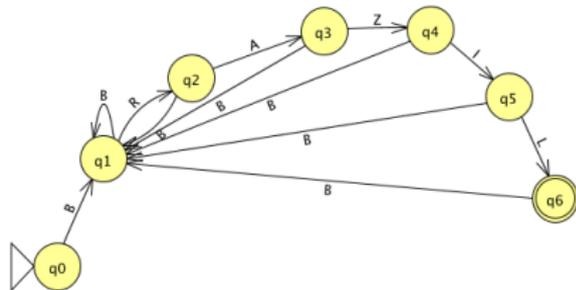
- ▶ Each such DFA will then “go over” a **text** string,
- ▶ and tell us what it “thinks” of the text.
- ▶ First text: **AAAAA**ABRAZILAAAA**AA**B.
- ▶ Second text: BRAZILZIPHERBRAZILYOO.

The Two Pattern DFAs

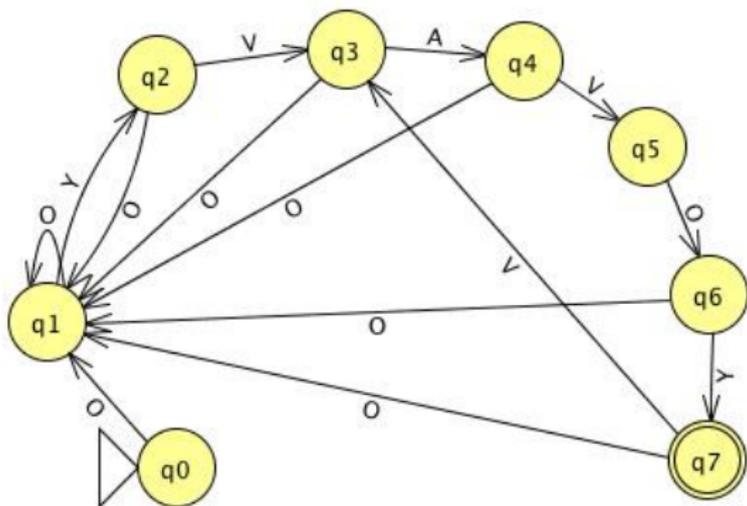
The pattern **BRAZIL** has no “inner repetitions”.

The pattern **AAAAA** is highly repetitive.

What, if any, effect do these facts have on the resulting pattern DFAs?



The OYVAVOY Pattern DFA



The Pattern DFA Approach

1. Given the **pattern**, construct the corresponding DFA.
 - ▶ Can you design an algorithm to do this, based on the examples given above?
2. Given the **text** and the pattern DFA, run the text through the DFA. Each time we hit the **accept state**, we report the **location** in the **text**.
3. To be rigorously treated in the algorithms course (e.g. the KMP algorithm).