Computer Science 1001.py

Lecture 20: String Matching
Letter Frequencies in Real, Written Text

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Amir Gilad, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University, Spring Semester, 2017
http://tau-cs1001-py.wikidot.com

And Now for Something Completely Different:
Text Processing and String Matching

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 the rest of this lecture, and most of the next one, we explore four very different algorithmic approaches to string matching.

An Example (courtesy of Charles Lutwidge Dodgson)

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.

Pattern: gyre and gimble

First Match

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.

Pattern: gyre and gimble

And a Second One

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.

Pattern: gyre and gimble

Jabberwocky (Through the Looking-Glass, and What Alice Found There)

'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.

Lost in Translation?

Here is a partial translation, courtesy of Yuval Pinter

Twas brillig, and the slithy toves'	עֵת מַרְתֵּחַ הָיְתָה, הַלְּטוֹרוֹת
;Did gyre and gimble in the wabe	הַגְּמוּסוֹת
,All mimsy were the borogoves	סָבְּבוּ וְרָדְעוּ בְּחֵיק הַנַּרְחָק;
.And the mome raths outgrabe	כֹּה מְעוּדָּשׁוֹת הָיוּ הַבָּרְגּוֹסוֹת,
	וְהַזּוֹרָחִים גָּעֲמוּ בַּפַּרְסָתָק.
!Beware the Jabberwock, my son"	"הִשָּׁמֵר מֵהַלַּהֲגָרְן, בְּנִי יָקִיר!
The jaws that bite, the claws that	מֶלְתַּעַת נוֹגֶסֶת, צִפָּרֶן בּוֹצַעַת!
!catch	גּוֹרָה תָּגוּר מֵחוּבְחוּב, וְתָדִיר
Beware the Jubjub bird, and shun	אֶת מִלְהֶטֶת הַמַּעַף שֶׁהִיא
"!The frumious Bandersnatch	הַלַּפְתּוֹץ!"
:He took his vorpal sword in hand	נָטַל הוּא חַרְבּוֹ הַשְּׁנוֹטָה לְיָדוֹ:
Long time the manxome foe he	זְמָן רַב תָּר אָחַר הַצּוֹרֵר הַמַּצְלֵף –
—sought	אָחַר-כָּךְ הוּא נָח תַּחַת עֵץ הַגְּרִידוֹ,
,So rested he by the Tumtum tree	עָמַד בְּעוֹמְדוֹ לִזְמָן-מָה וְשָׂרְעֵף.
.And stood awhile in thought	

So, gibberish remains gibberish, even when translated to Hebrew.
(the complete translation can be found here.)

String Matching in Python

Python has a built-in string class and methods. Combined with the regular expression (re) package, these methods can be used to solve all our string matching needs neatly and efficiently.
(Iterator ahoy!)

```
>>> import re
>>> occurances=re.finditer("ab", "abcabcaabbc") # pattern is ab
# creates an iterator with all occurances of pattern in text
>>> type(occurances)
<class 'callable_iterator'>
>>> [m.start() for m in occurances] # starting points of matches
[0, 3, 7]
>>> [m.start() for m in occurances] # starting points of matches
[]      # iterator cannot be reused - must be re-created
>>> occurances=re.finditer("ab", "abcabcaabbc")
>>> [m.span() for m in occurances] # spanning intervals of matches
[(0, 2), (3, 5), (7, 9)]
```

Of course, these built-in regular expression methods for string matching are not good enough for us, as we want to know what is going on "under the hood".

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 (highlights only).

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 `hash(P[0..m-1])`. Denote the outcome by `t`.

Method: For every shift, s, $0 \leq s \leq n - m$, compute `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!

Note that this solution may introduce false positive errors (report matches where they do not exist, due to collisions in the hash function). This occurs with very small, yet positive probability.

An Algorithm Based on Hashing: Observations

Initial Step: Compute `hash(P[0..m-1])`. Denote the outcome by `t`.

Method: For every shift, s, $0 \leq s \leq n - m$, compute `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 finding matches for 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.

And Now for Something Somewhat Different: Karp–Rabin's Algorithm

אֲנִי מַאֲמִין / שאול טשרניחובסקי

שַׁחֲקִי, שַׁחֲקִי עַל הַחֲלוֹמוֹת,
זוּ אֲנִי הַחוֹלֵם שָׂח.
שַׁחֲקִי כִּי בָאָדָם אַאֲמִין,
כִּי עוֹדֶנִּי מַאֲמִין בָּךְ.

כִּי עוֹד נַפְשִׁי דְּרוֹר שׁוֹאֶפֶת
לֹא מְכַרְתִּיהָ לְעֶגֶל-פָּז,
כִּי עוֹד אַאֲמִין גַּם בָּאָדָם,
גַּם בְּרוּחוֹ, רוּחַ עָז.

רוּחוֹ יַשְׁלִיךְ בַּבְּלֵי-הֶבֶל,
יְרוֹמְמֶנּוּ בָּמֳתֵי-עָל;
לֹא בָרָעָב יָמוּת עֹבֵד,
דְּרוֹר – לַנֶּפֶשׁ, פַּת – לַדָּל.

שַׁחֲקִי כִּי גַּם בְּרֵעוּת אַאֲמִין,
אַאֲמִין, כִּי עוֹד אֶמְצָא לֵב,
לֵב תִּקְוֹתַי גַּם תִּקְוֹתָיו,
יָחוּשׁ אֹשֶׁר, יָבִין כְּאֵב.

אַאֲמִינָה גַּם בָּעָתִיד,
אַף אִם יִרְחַק זֶה הַיּוֹם,
אַךְ בּוֹא יָבוֹא – יִשְׂאוּ שָׁלוֹם
אָז וּבְרָכָה לְאֹם מִלְאֹם.

יָשׁוּב יִפְרַח אָז גַּם עַמִּי,
וּבָאָרֶץ יָקוּם דּוֹר,
בַּרְזֶל-כְּבָלָיו יוּסַר מֵנּוּ,
עֵין-בְּעֵין יִרְאֶה אוֹר.

יִחְיֶה, יֶאֱהַב, יִפְעַל, יָעַשׂ,
דּוֹר בָּאָרֶץ אָמְנָם חָי
לֹא בָּעָתִיד – בַּשָּׁמַיִם,
חַיֵּי-רוּחַ לוֹ אֵין דָּי.

אָז שִׁיר חָדָשׁ יָשִׁיר מְשׁוֹרֵר,
לְיֹפִי וְנִשְׂגָּב לִבּוֹ עֵר;
לוֹ, לַצָּעִיר, מֵעַל קִבְרִי
פְּרָחִים יִלְקְטוּ לַזֵּר.

אודיסה 1892

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 `finger_print(P[0..m-1])`. Denote the outcome by `t`.

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

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

Time complexity: $O(m)$ operations to compute the two finger prints `finger_print(P[0..m-1])` and `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:
$N_P = p_0 B^{m-1} + p_1 B^{m-2} + \ldots + p_{m-3} B^2 + p_{m-2} B + p_{m-1}$.

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

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

Arithmetization

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

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:

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

Once we got N_P and all of $T_0, T_1, \ldots, T_{n-m}$, 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 $T_0, T_1, \ldots, T_{n-m}$, 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 $T_0, T_1, \ldots, T_{n-m}$?

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

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

Suppose we have already computed
$T_s = t_s B^{m-1} + t_{s+1} B^{m-2} + \ldots + t_{s+m-3} B^2 + t_{s+m-2} B + t_{s+m-1}$.

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

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 occurances 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 $p = \texttt{fingerprint}(N_P)$.
2. For every shift, s, $0 \leq s \leq n - m$, compute the number $f_s = \texttt{fingerprint}(T_s)$.
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.

(http://www.dalycity.org/City_Hall/Departments/police_department/Services/fingerprinting.htm)

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

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

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

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

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 Algorithm: Intuition (mostly on the board)

1) hash does not allow "sliding window",
2) simple arithmetization (without modulo) needs to handle large numbers.
Fingerprint solves both.

Karp-Rabin Fingerprinting

Pick a random prime, r, in the range $2^{31} < r < 2^{32}$ (a 32-bit random prime; larger primes can be used "upon demand").
The corresponding fingerprint function is $F_r(m) = m \mod r$.

Details:

1. Compute the number $p = \texttt{fingerprint}(N_P)$.
2. For every shift, s, $0 \leq s \leq n - m$, compute the number $f_s = \texttt{fingerprint}(T_s)$.
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 $T_0, \ldots, T_{n-m}$ first, and then do the modular reduction, $\bmod r$, we would have to deal with large numbers (unbounded sizes).
- Instead, we employ the $\bmod r$ 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
```

Incidentally, if you want to use a 64 bit r, you may try $r = 2^{64} - 59 = 18446744073709551557$.

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 occurances 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="aaaaaaaaaaaaaaaaaaaaaaaaaa"
>>> 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="aaaaaaaaaaaaabbbaaaaaaaaa"
>>> 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 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
>>> 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]
```

Karp–Rabin Algorithm: Reflections

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 finger_print($P[0..m - 1]$). Denote the outcome by t.
Method: For every shift, s, ($0 \leq s \leq n - m$), compute finger_print($T[s..s + m - 1]$). If this equals t, report s (and continue).

Key Idea: Given the finger print of one window, finger_print($T[s..s + m - 1]$), we will can compute the finger print of the next one, finger_print($T[s + 1..s + m]$) in a constant number of operations.
How: Arithmetization + Horner technique + applying modular reduction in order to avoid large numbers.

String Matching Based on DFA: Deterministic Finite Automata (for reference only)

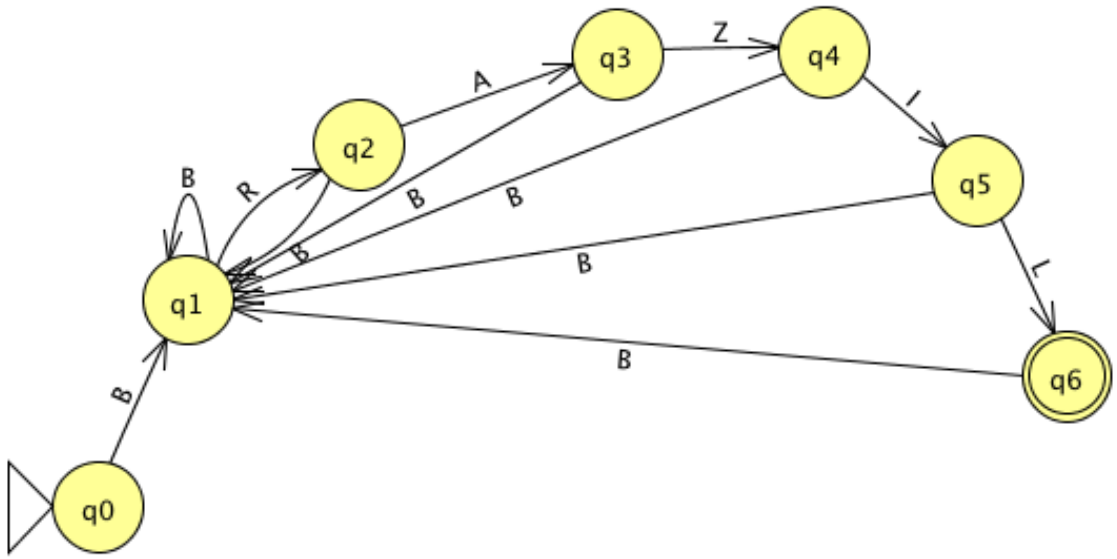
- A DFA is one of the simplest models of computation.
- It consists of a finite number, m, of states, usually denoted $q_0, \ldots, q_{m-1}$.
- States are viewed as nodes (denoted by circles) in a directed graph.
- Directed edges in the graph are labeled by single letters from the alphabet Σ.
- The DFA 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 corresponds to more than one outgoing arrow.
- Non deterministic automata are an important computational model, but not within our scope. They will be discussed in, ahhm, the computational model course.

The BRAZIL Pattern DFA

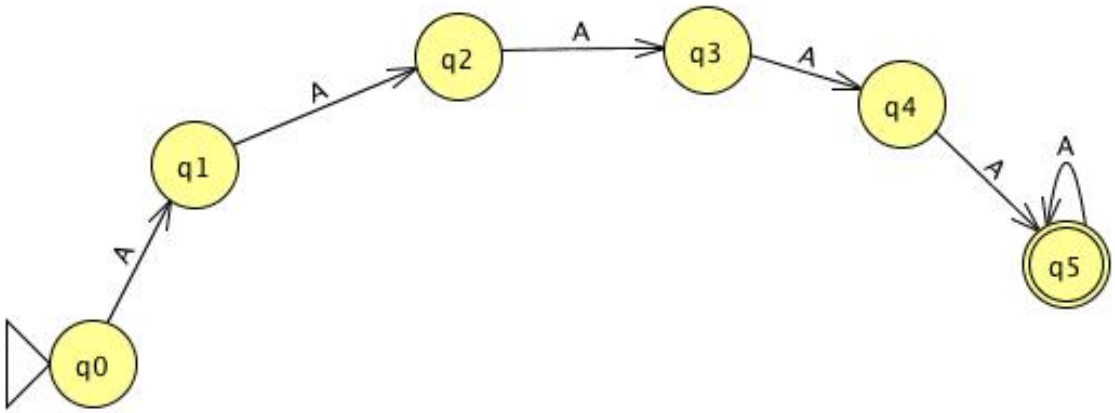
- We describe specific DFAs, each corresponding to a different pattern.



- 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

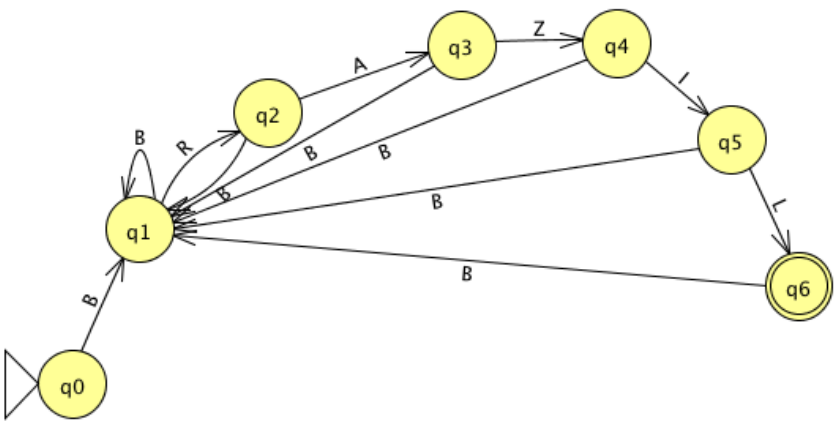
- We describe specific DFAs, each corresponding to a different

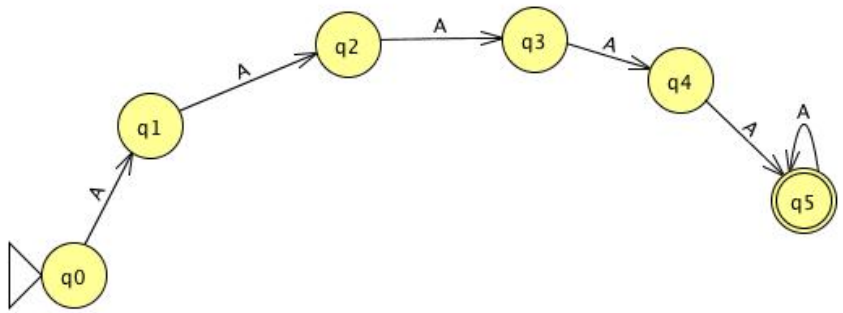


- Each such DFA will then "go over" a text string,
- and tell us what it "thinks" of the text.
- First text: AAAAAAABRAZILAAAAAAB.
- Second text: BRAZILZIPHERBRAZILYOO.

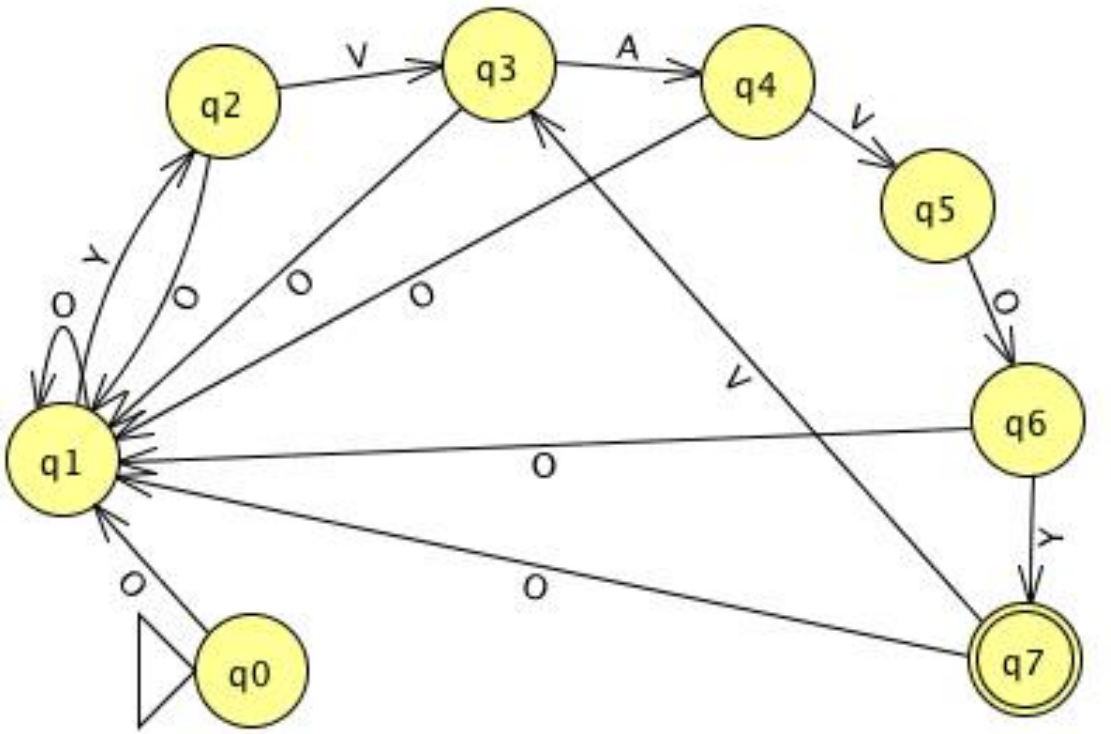
The Two Pattern DFAs

The pattern BRAZIL has no "inner repetitions".
The patern 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).

And Now for Something Completely Different:
Frequencies of Letters in Human Generated Texts

"I weep for you," the Walrus said:
"I deeply sympathize."
With sobs and tears he sorted out
Those of the largest size,
Holding his pocket-handkerchief
Before his streaming eyes.

"O Oysters," said the Carpenter,
"You've had a pleasant run!
Shall we be trotting home again?'
But answer came there none–
And this was scarcely odd, because
They'd eaten every one.

Through the Looking-Glass and What Alice Found There:
Lewis Carroll, 1871.

Why We Care About Frequencies of Letters

"Recent analyses show that letter frequencies, like word frequencies, tend to vary, both by writer and by subject. One cannot write an essay about x-rays without using frequent Xs, and the essay will have an especially strange letter frequency if the essay is about the frequent use of x-rays to treat zebras in Qatar. Different authors have habits which can be reflected in their use of letters. Hemingway's writing style, for example, is visibly different from Faulkner's. Letter, bigram, trigram, word frequencies, word length, and sentence length can be calculated for specific authors, and used to prove or disprove authorship of texts, even for authors whose styles aren't so divergent."

(Text taken from Wikipedia).

Letter Counts in Text

The following code first converts every character to lower case (this has no effect for non-letters). Filters out non-letters; counts the number of occurrences of each letter, using a (Python) dictionary called `d`; turns the dictionary into a list, and sorts it by the counts.

```
def char_count(text):
    """ counts the number of occurrences of ascii letters in a text
    Returns a list of pairs, where first elements in pairs
    are the letters, and second elements being the counts """

    d = {}  # initializing an empty dictionary
    for ch in text:
        ch = str.lower(ch) # convert letter lower case
        if ch < "a" or ch > "z":  # not a letter
            continue
        else:    # incrementing letter count by 1
            if ch in d:
                d[ch]=d[ch]+1
            else:
                d[ch]=1
    lst=list(d.items())  # turning dictionary into list
    return sorted(lst, key=lambda pair: -pair[1])
                            # sort high to low by counts
```

Letter Counts in Text: Sample Executions

We will try to understand what this code is doing by running it on a few synthetic examples.

```
>>> text="AaBbCDEFGhijklmNOPQRSTuvwxyz1234567890+-=/*a"
>>> char_count(text)
[('a', 3), ('b', 2), ('c', 1), ('e', 1), ('d', 1), ('g', 1),
('f', 1), ('i', 1), ('h', 1), ('k', 1), ('j', 1), ('m', 1),
('l', 1), ('o', 1), ('n', 1), ('q', 1), ('p', 1), ('s', 1),
('r', 1), ('u', 1), ('t', 1), ('w', 1), ('v', 1), ('y', 1),
('x', 1), ('z', 1)]
```

Letters Statistics in Text

For longer strings, it is the frequencies of letters rather than their counts we are interested in. We will express those as percentages.

```
def percent(frac):
    """ represents fraction as percentage, w/ 2 digit accuracy """
    return int(frac*10000)/100

def stats(text):
    """ letter frequencies (as percentage of total letters) """
    srt=char_count(text)
    total=sum(elem[1] for elem in srt)
    return [(elem[0],percent(elem[1]/total)) for elem in srt]
```

Real, Written Text

With the availability of all sorts of texts on the web, we can write code that downloads relevant pieces of text that will subsequently be analyzed. However, this requires some additional modules and interfaces, some websites block attempts by programs (aka robots) to access them, etc. etc.

For these reasons, we (read "your lecturer") have accessed 5 texts manually, cut parts of the text and pasted them into corresponding Python code.

- Chapter 1 of Genesis (we will not risk putting a date here :-).
- Book I of Homer's Iliad (800BC, translated by Samuel Butler).
- Act 3, Scene 1 of Shakespeare's Merchant of Venice (estimated 1596–1598).
- Chapter 1 of Lewis Carol's Alice in Wonderland (Charles Lutwidge Dodgson, 1865).
- The May 13, 1940 first speech of Winston Churchill as prime minister to house of commons ("Blood, Toil, Tears and Sweat").

Real, Written Text

We load the different text by using the import command. Then we examine portions of the text to see we got what we bargained for.

```
>>> from genesis1 import *  # genesis chapter1
>>> print(genesis1[0:257])

1  In the beginning God created the heavens and the earth.

2 And the earth was waste and void; and darkness was upon
the face of the deep: and the Spirit of God moved upon the
face of the waters.

3 And God said, Let there be light: and there was light.

>>> from iliad1 import *
>>> print(iliad1[115:217])

Sing, O goddess, the anger of Achilles son of Peleus, that brought
countless ills upon the Achaeans.

>>> from shylock import *
>>> from alice1 import *
>>> from churchill import *
```

Real, Written Text (cont.)

These texts greatly vary in their origin, time of writing, original language (the bible and the Iliad are translated from Hebrew and ancient Greek to English, correspondingly), and length. Lets examine their lengths first.

```
>>> len(genesis1)
4214
>>> len(iliad1)
30689
>>> len(shylock)
6441
>>> len(alice1)
11514
>>> len(churchill)
4773
```

Letter Frequencies in Real, Written Text (cont. cont.)

We will exhibit just the 6 most frequent letters in each text. We will also check if all 26 letters in the English alphabet appear in each text.

```
>>> stats(genesis1)[0:6]; len(stats(genesis1))
[('e',14.48),('t',10.3),('a',9.7),('h',7.57),('d',7.54),('n',7.35)]
23   # only 23 letters appear

>>> stats(iliad1)[0:6]; len(stats(iliad1))
[('e',12.42),('t',8.35),('o',8.32),('a',8.04),('h',7.8),('s',6.65)]
26   # all letters appear

>>> stats(shylock)[0:6]; len(stats(shylock))
[('e',11.66),('t',8.6),('a',8.56),('o',8.29),('h',6.91),('n',6.82)]
25  # one letter missing

>>> stats(alice1)[0:6]; len(stats(alice1))
[('e',12.47),('t',10.05),('o',8.03),('a',7.86),('h',6.85),
 ('i',6.58)]
26

>>> stats(churchill)[0:6]; len(stats(churchill))
[('e',12.35),('t',10.43),('o',8.18),('a',7.91),('i',7.64),
 ('n',6.79)]
25 # one letter missing
```

Letter Frequencies in Real, Written Text (cont.)

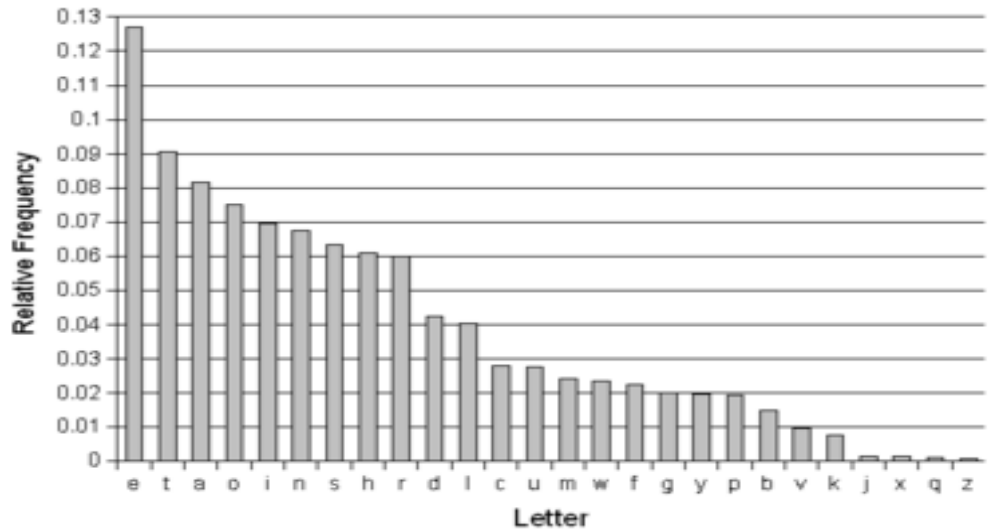
We see that even the 6 most popular letters are not the same in these 6 texts. In fact, even the top 3 letters are not the same. Remarkably enough, though, the top 2 letters are `e` and then `t` in all 6 texts.

Here is the complete list of letters' frequencies in chapter 1 of Alice in Wonderland.

```
>>> stats(alice1)
[('e',12.47),('t',10.05),('o',8.03),('a',7.86),('h',6.85),
('i',6.58),('n',6.5),('s',5.81),('r',5.24),('l',4.83),('d',4.49),
('w',2.91),('u',2.87),('g',2.35),('f',2.14),('c',2.03),
('y',1.88),('b',1.66),('m',1.66),('p',1.5),('k',1.13),('v',0.78),
('x',0.08),('j',0.06),('q',0.06),('z',0.04)]
```

Frequencies of Letters in Natural Languages

The distribution of single letters in any natural languages' texts is highly non uniform. According to the diagram below (taken from Wikipedia), the four most frequent letters are {e, t, a, o}. The letter e appears in 12.8% of written English, t's frequency is 9.05%, a's is 8.1%, and o's is 7.6%.



The frequencies in this diagram are based some "representative text", or corpus (for example, a dictionary).

More on Frequencies of Letters

Letter frequencies in different languages differ substantially. For example, while in many European languages the letter `e` had the highest frequency (like in English), the second most frequent letter is `t` in English, `n` in German, and `a` in Spanish. In fact, the most frequent letter in Portuguese is `a` (`e` is the second).
(data taken from Wikipedia.)

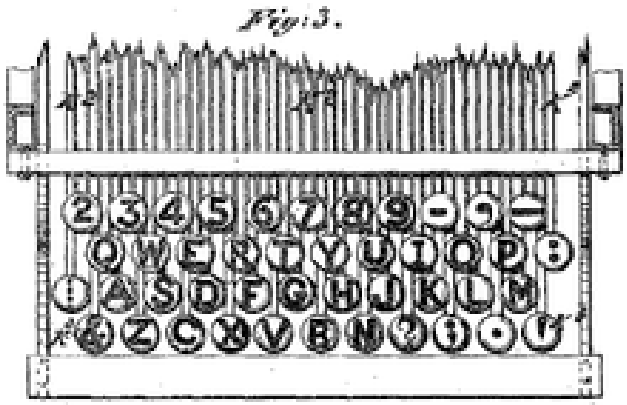
More on Frequencies of Letters, cont.

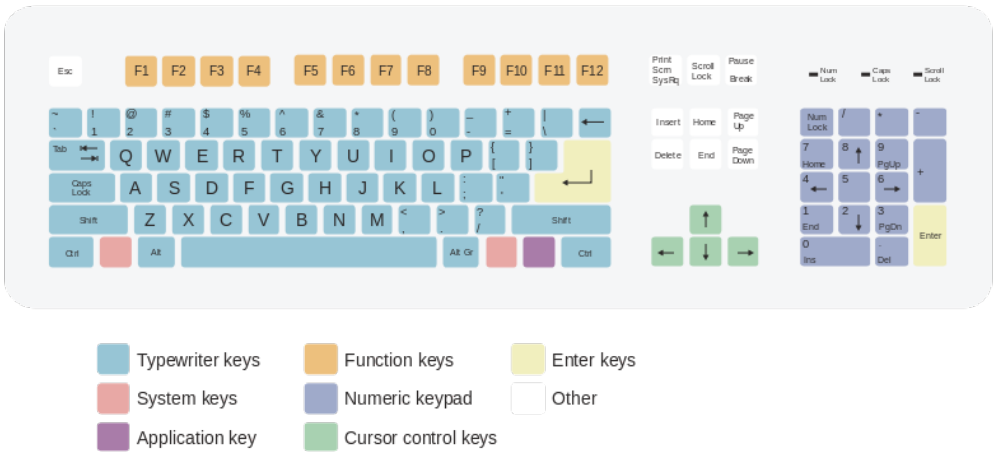
An important application of the non uniform frequency of letters is text compression. The Huffman method encodes frequent letters using short binary strings, and infrequent letters using longer binary strings. A typical text contains more frequent letters than infrequent letters. This enable the Huffman method to compress typical, human produced texts and save as much as 40% (compressed text vs. original text).

In addition, the non uniform distribution of letters in written text is the key to breaking substitution cypher encryption. Unfortunately, we will most probably not have time to cover this fun topic in the course.

An Unexpected Application of Frequencies of Letters

Layout of letters and other characters on keyboards were designed at the age of mechanical typewriters (not that long ago, mind you). A major goals was to minimize occurrences of jams (caused by two levers crossing each other). Frequently used pairs of letters (*e.g.* `th` or `st`) were placed far away from each other. The standard layout of keyboards, Qwerty, reflects this principle.



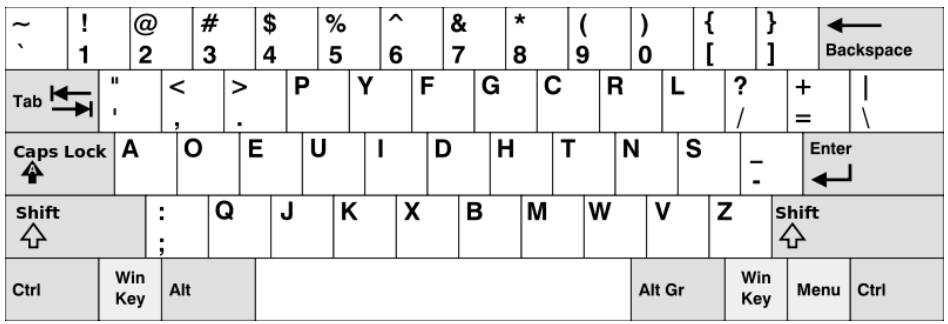


Such considerations became obsolete with the introduction of computers' keyboards, which eliminated the use of mechanical levers.

(Figures taken from Wikipedia).

An Unexpected Application of Frequencies of Letters, cont.

An alternative keyboard layout was proposed back in 1932 by August Dvorak, an educational psychologist and professor of education at the University of Washington in Seattle. The Dvorak layout takes letter frequencies into account. For example, the 8 most frequent English letters, `e, t, a, o, i, n, s, h`, are all placed on the home row of the keyboard, which is the easiest row to use.



The Dvorak system was not widely adopted, so most keyboard are still using the Qwerty layout. Operating systems, including Windows and MAC OSX, allow one to use Dvorak without physically changing the keyboard. Thus, the user has to remember the "virtual" layout. It is supposedly faster to type using Dvorak. Popular among geeks.

(Figures taken from Wikipedia).