Strings and exact matching

Ben Langmead

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

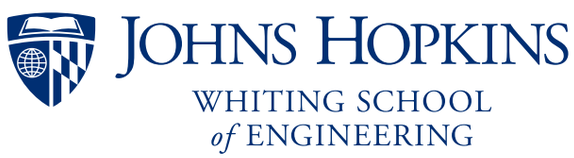
Department of Computer Science

Resources

Gusfield, Dan. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

iPython notebooks:
https://github.com/BenLangmead/comp-genomics-class

Including notebooks on strings, exact matching, and Z algorithm



Strings: a useful abstraction

Lots of data is string-like: books, web pages, files on your hard drive, sensor data, medical records, chess games, ...

Algorithms for one kind of string are often applicable to others:

Regular expression matching is used to search files on your filesystem (grep), and to find "bad" network packets (snort)

Methods for indexing books and web pages (inverted indexing) can also be used to index DNA sequences

Methods for understanding speech (HMMs) can also be used to understand handwriting or identify genes in genomes

Strings come from somewhere

Processes that give rise to real-world strings are complicated. It helps to understand these processes.

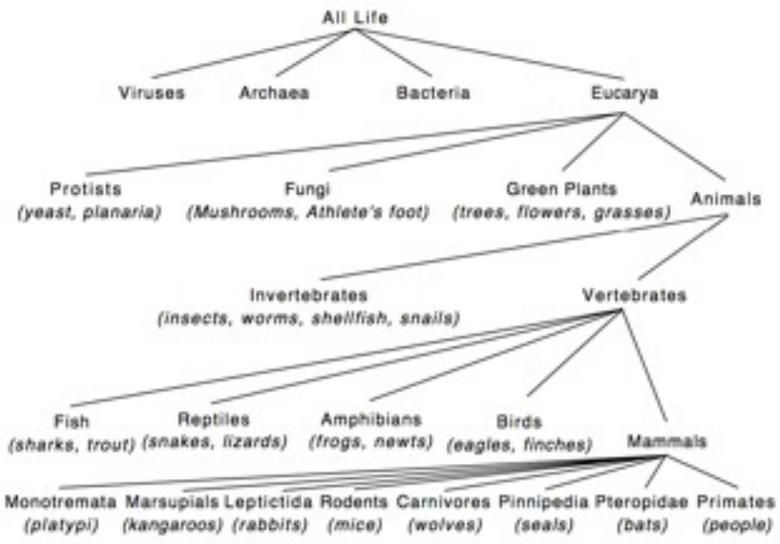
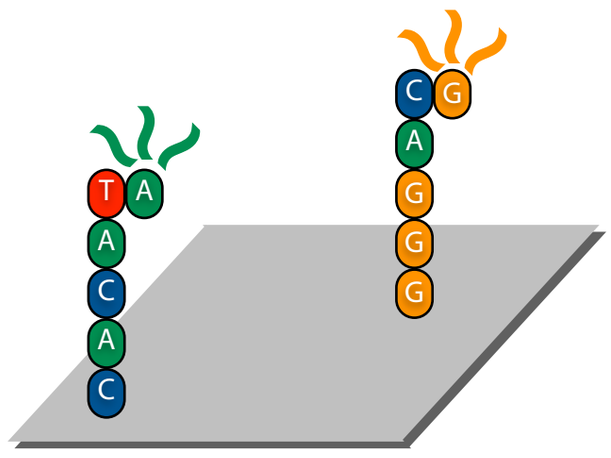


Figure from: Hunter, Lawrence. "Molecular biology for computer scientists." *Artificial intelligence and molecular biology* (1993): 1-46.

1. Evolution:
   - Mutation
   - Recombination
   - (Retro)transposition

2. Lab procedures:
   - PCR
   - Cell line passages



3. Sequencing:
   - Fragmentation bias
   - Miscalled bases

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Strings have structure

One way to model a string-generating process is with coin flips:

{  = A,  = C,  = G,  = T }

But such strings lack internal patterns ("structure") exhibited by real strings

More than 40% of human genome is covered by *transposable elements*, which copy-and-paste themselves across the genome and mutate

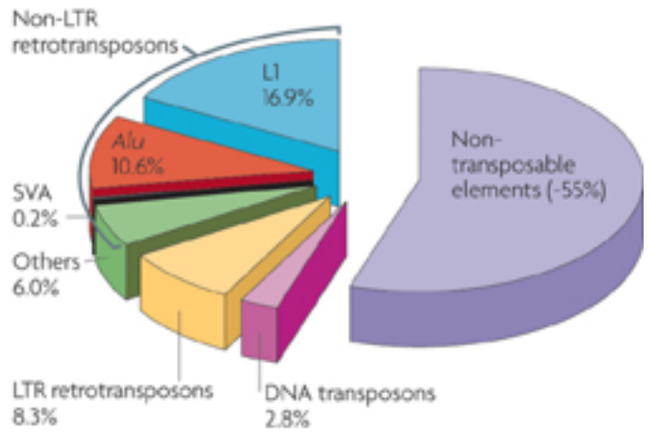


Image from: Cordaux R, Batzer MA. The impact of retrotransposons on human genome evolution. Nat Rev Genet. 2009 Oct;10(10):691-703

Slipped strand mispairing during DNA replication results in expansion or retraction of simple (*tandem*) repeats

••• ATATATATATATAT •••

↕

••• ATATATATATATATATAT •••

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

String definitions

A *string S* is a finite ordered list of characters

Characters are drawn from an alphabet Σ. We often assume Σ has $O(1)$ elements *.

Nucleic acid alphabet: { A, C, G, T }

Amino acid alphabet: { A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V }

Length of S, $|S|$, is the number of characters in S

ϵ is the empty string. $|\epsilon| = 0$

* sometimes we'll consider $|\Sigma|$ explicitly

String definitions

For strings *S* and *T* over *Σ*, their *concatenation* consists of the characters of *S* followed by the characters of *T*, denoted *ST*

S is a *substring* of *T* if there exist (possibly empty) strings *u* and *v* such that $T = uSv$

S is a *prefix* of *T* if there exists a string *u* such that $T = Su$.
If neither *S* nor *u* are ϵ, *S* is a *proper prefix* of *T*.

Definitions of *suffix* and *proper suffix* are similar

Python demo: http://nbviewer.ipython.org/6512698

String definitions

We defined *substring*. *Subsequence* is similar except the characters need not be consecutive.

"cat" is a substring and a subsequence of "con**cat**enate"

"cant" is a subsequence of "con**ca**te**n**a**t**e", but not a substring

Exact matching

Looking for places where a *pattern P* occurrs as a substring of a *text T*. Each such place is an *occurrence* or *match*.

Let $n = |P|$, and let $m = |T|$, and assume $n \leq m$

An *alignment* is a way of putting *P's* characters opposite *T's* characters. It may or may not correspond to an occurrence.

```
P: word
T: There would have been a time for such a word
```

```
Alignment 1:  word
```

```
Alignment 2:                                word
```

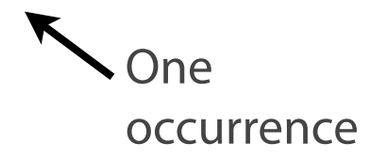
Exact matching

What's a simple algorithm for exact matching?

P: `word`

T: `There would have been a time for such a word`

```
word word word word word word word word word word
 word word word word word word word word word word
  word word word word word word word word word word
   word word word word word word word word word word
```



Try all possible alignments. For each, check whether it's an occurrence. "Naïve algorithm."

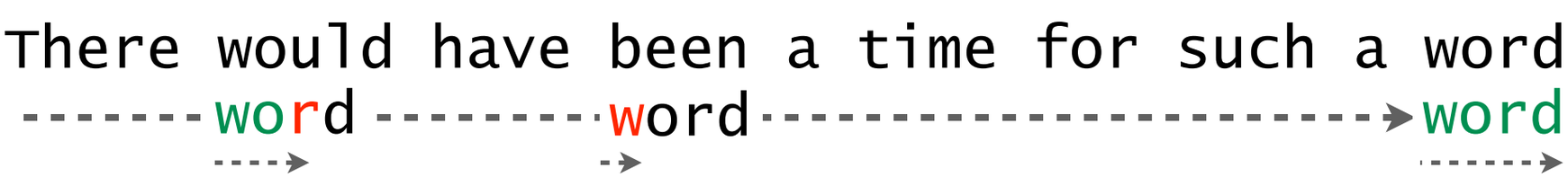
Exact matching: naïve algorithm

```
def naive(p, t):
    occurrences = []
    for i in xrange(len(t) - len(p) + 1): # loop over alignments, L-to-R
        match = True
        for j in xrange(len(p)):          # loop over characters, L-to-R
            if t[i+j] != p[j]:            # character compare
                match = False             # mismatch; reject alignment
                break
        if match:
          occurrences.append(i)           # all chars matched; record
    return occurrences
```

Python demo: http://nbviewer.ipython.org/6513059

P: word

T: There would have been a time for such a word



Exact matching: naïve algorithm

How many alignments are possible given n and m ($|P|$ and $|T|$)?

$m - n + 1$

What is the greatest number of character comparisons possible?

$n(m - n + 1)$

the *least* possible?

$m - n + 1$

How many character comparisons in this example?

P: `word`

T: `There would have been a time for such a word`

$m - n$ mismatches, 6 matches

Exact matching: naïve algorithm

Greatest # character comparisons	Least:
$n(m - n + 1)$	$m - n + 1$

Worst-case time bound of naïve algorithm is $O(nm)$

In the best case, we do only ~ m character comparisons