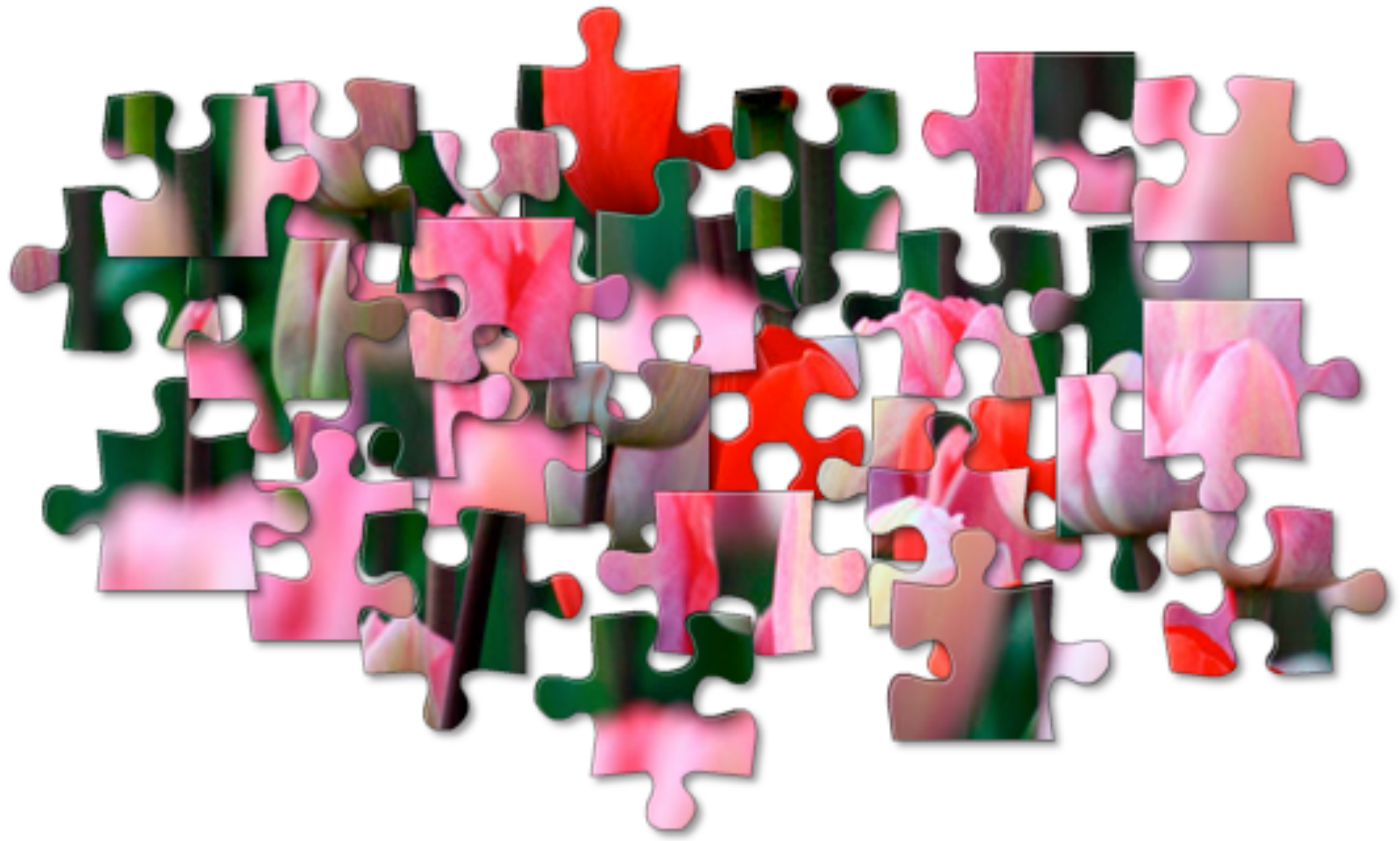
String Matching

Reads are strings

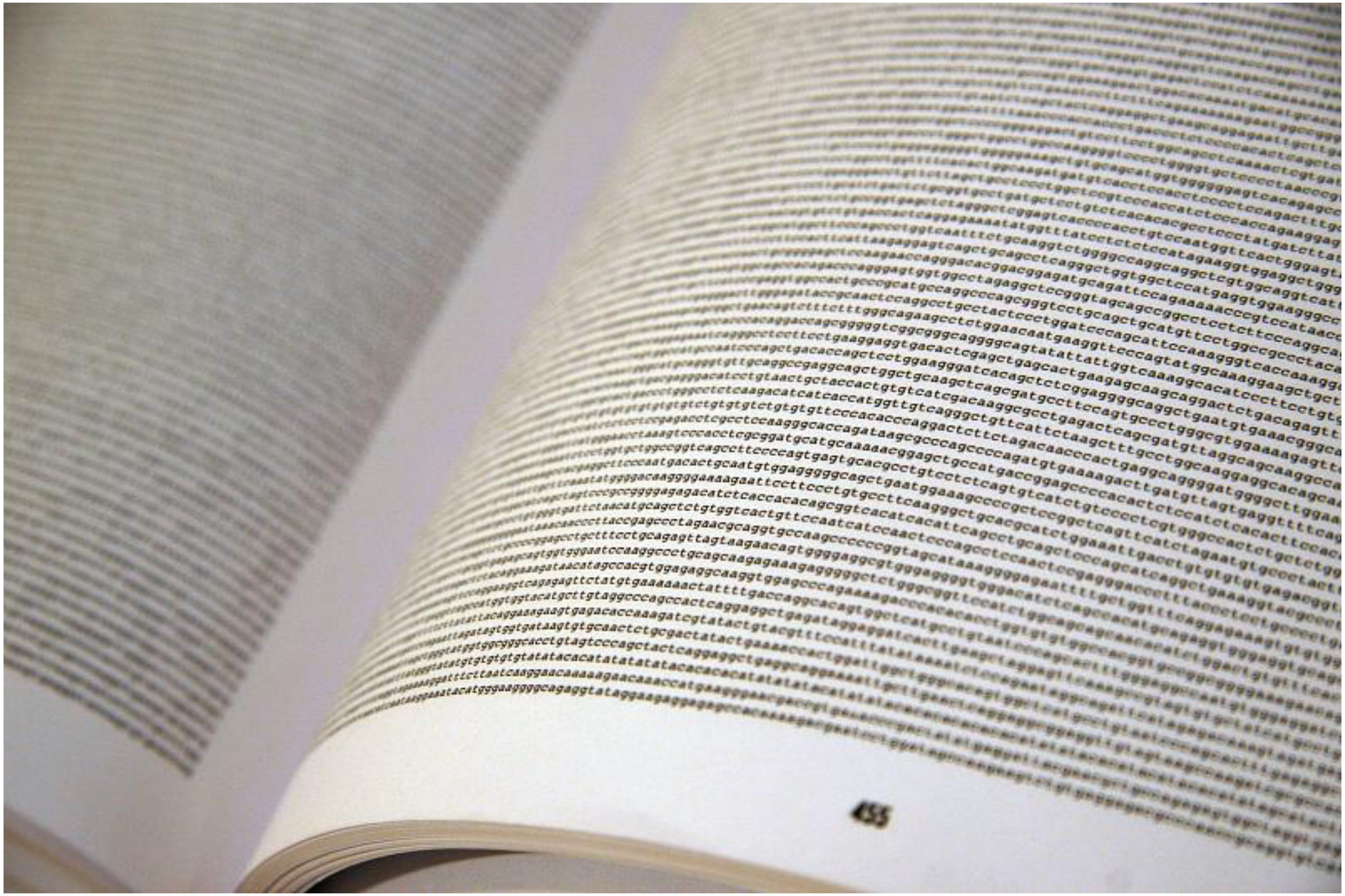
GTATGCACGCGATAG	TATGTCGCAGTATCT	CACCCTATGTCGCAG
TAGCATTGCGAGACG	GGTATGCACGCGATA	TGGAGCCGGAGCACC
TGTCTTTGATTCCTG	CGCGATAGCATTGCG	GCATTGCGAGACGCT
GACGCTGGAGCCGGA	GCACCCTATGTCGCA	GTATCTGTCTTTGAT
TATCGCACCTACGTT	CAATATTCGATCATG	GATCACAGGTCTATC
CACGGGAGCTCTCCA	TGCATTTGGTATTTT	CGTCTGGGGGGTATG
GTATGCACGCGATAG	ACCTACGTTCAATAT	TATTTATCGCACCTA
GCGAGACGCTGGAGC	CTATCACCCTATTAA	CTGTCTTTGATTCCT
CCTACGTTCAATATT	GCACCTACGTTCAAT	GTCTGGGGGGTATGC
GACGCTGGAGCCGGA	GCACCCTATGTCGCA	GTATCTGTCTTTGAT
TATCGCACCTACGTT	CAATATTCGATCATG	GATCACAGGTCTATC
CACGGGAGCTCTCCA	TGCATTTGGTATTTT	CGTCTGGGGGGTATG



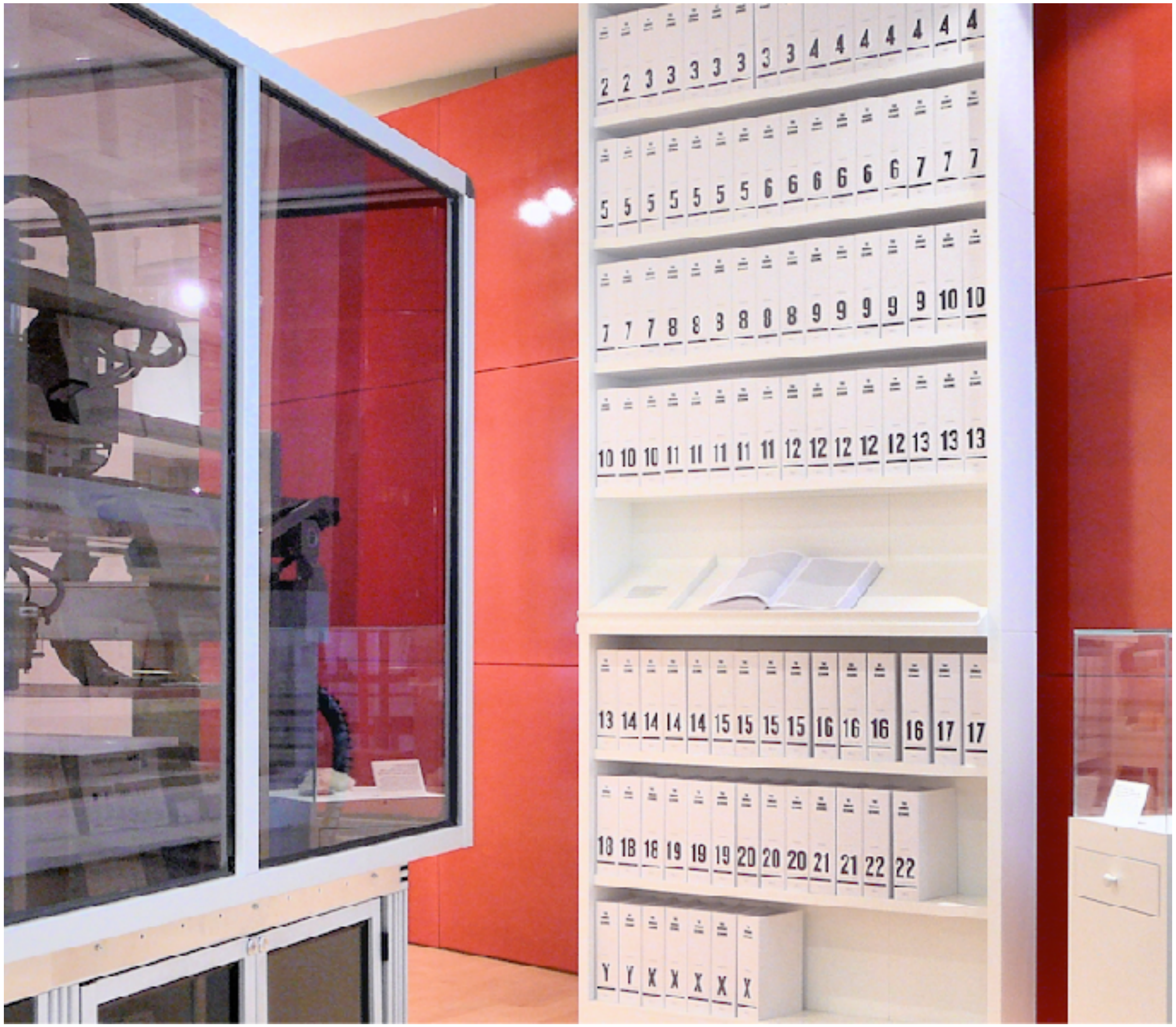
Sequencing reads are *strings;* sequences of characters

The strings are the only hints we get about *where the reads came from from* with respect to the longer DNA molecules...

... like pictures on puzzle pieces



What if I told you to find all the places where the string GATACCA occurs in here?



What if I told you to find all the places where the string GATACCA occurs in here?

Strings

Read

CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTAGGCCTTC x billions

Reference

```
GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTT
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTC
GCAGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACTTACTAAAGTGTGTTAATTAATTAATGCTTGTAGGACATAATAATA
ACAATTGAATGTCTGCACAGCCACTTTCCACACAGACATCATAACAAAAAATTTCCACCA
AACCCCCCCTCCCCCGCTTCTGGCCACAGCA[illegible]AAACACA[illegible]CTCTGCCAAACCCCAAAA
ACAAAGAACCCTAACACCAGCCTAACCA[illegible]ATTTCAAATTTTATC[illegible]TTTGGCGGTATGCAC
TTTTAACAGTCACCCCCCAACTAACA[illegible]ATTATTTTCCCCTCCCAC[illegible]CATACTACTAAT
CTCATCAATACAACCCCCGCCCATC[illegible]TACCCAGCACACACACACCG[illegible]CTAACCCCATA
CCCCGAACCAACCAAACCCCAAAG[illegible]CACCCCCCACAGTTTATGTAGC[illegible]ACCTCCTCAAA
GCAATACACTGACCCGCTCAAAC[illegible]CCTGGATTTTGGATCCACCCAGCGC[illegible]TTGGCCTAAA
CTAGCCTTTCTATTAGCTCTTAG[illegible]AAGATTACACATGCAAGCATCCCCG[illegible]TCCAGTGAGT
TCACCCTCTAAATCACCACGATC[illegible]AAAGGAACAAGCATCAAGCACGCAG[illegible]AATGCAGCTC
AAAACGCTTAGCCTAGCCACACCC[illegible]CCACGGGAAACAGCAGTGATTAACC[illegible]TTAGCAATAA
ACGAAAGTTTAACTAAGCTATACT[illegible]ACCCCAGGGTTGGTCAATTTCGT[illegible]CCAGCCACCGC
GGTCACACGATTAACCCAAGTCAAT[illegible]GAAGCCGGCGTAAAGAGTGTT[illegible]TAGATCACCCCC
TCCCCAATAAAGCTAAAACTCACCTGA[illegible]TTGTAAAAAACTCCAGTT[illegible]ACAAAATAGAC
TACGAAAGTGGCTTTAACATATCTGAACA[illegible]ACAATAGCTAAGA[illegible]TGGGATTAGA
TACCCCACTATGCTTAGCCCTAAACCTCAACAG[illegible]AACAA[illegible]CGCCAGAA
CACTACGAGCCACAGCTTAAAACTCAAAGGACCTGGCGGTGCTTCAT[illegible]TAGAGG
AGCCTGTTCTGTAATCGATAAACCCCGATCAACCTCACCACCTCTTGCT[illegible]TATA
CCGCCATCTTCAGCAAACCCTGATGAAGGCTACAAAGTAAGCGCAAGTACC[illegible]AG
ACGTTAGGTCAAGGTGTAGCCCATGAGGTGGCAAGAAATGGGCTACATTTTC[illegible]
AAAACTACGATAGCCCTTATGAAACTTAAGGGTCGAAGGTGGATTTAGCAGTAA[illegible]
AGTAGAGTGCTTAGTTGAACAGGGCCCTGAAGCGCGTACACACCGCCCGTCACCCT[illegible]
AAGTATACTTCAAAGGACATTTAACTAAAACCCCTACGCATTTATATAGAGGAGACAA[illegible]
CGTAACCTCAAACTCCTGCCTTTGGTGATCCACCCGCCTTGGCCTACCTGCATAATGAAG
AAGCACCCAACTTACACTTAGGAGATTTCAACTTAACTTGACCGCTCTGAGCTAAACCTA
GCCCCAAACCCACTCCACCTTACTACCAGACAACCTTAGCCAAACCATTTACCCAAATAA
AGTATAGGCGATAGAAATTGAAACCTGGCGCAATAGATATAGTACCGCAAGGGAAAGATG
AAAAATTATAACCAAGCATAATATAGCAAGGACTAACCCCTATACCTTCTGCATAATGAA
TTAACTAGAAATAACTTTGCAAGGAGAGCCAAAGCTAAGACCCCCGAAACCAGACGAGCT
ACCTAAGAACAGCTAAAAGAGCACACCCGTCTATGTAGCAAAATAGTGGGAAGATTTATA
GGTAGAGGCGACAAACCTACCGAGCCTGGTGATAGCTGGTTGTCCAAGATAGAATCTTAG
TTCAACTTTAAATTTGCCCACAGAACCCTCTAAATCCCCTTGTAAATTTAACTGTTAGTC
CAAAGAGGAACAGCTCTTTGGACACTAGGAAAAAACCTTGTAGAGAGAGTAAAAAATTTA
```

x million

We're going to *need* the right algorithms...

Strings are well studied

Many kinds of data are string-like: books, web pages, files on your hard drive, medical records, chess games, ...

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

Regular expression matching can find files on your filesystem (grep), or bad network packets (snort)

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

Methods for understanding speech (HMMs) can 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 them.

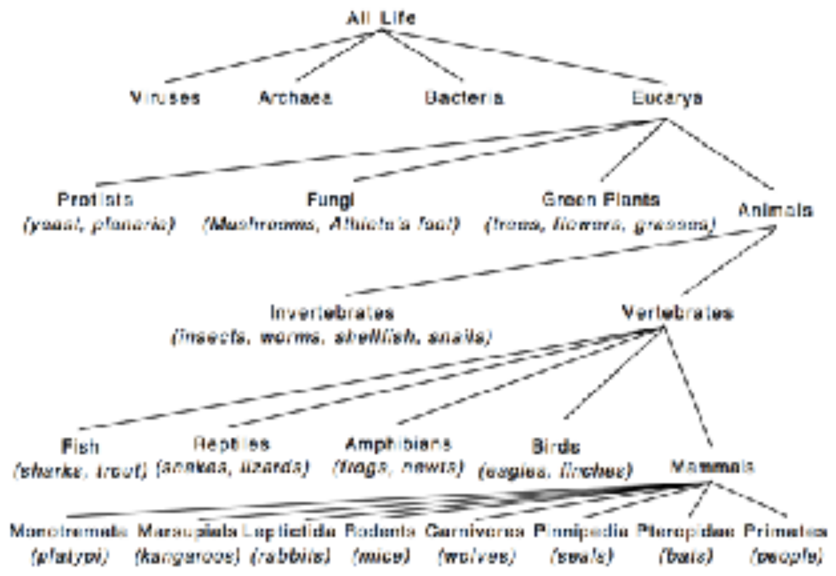
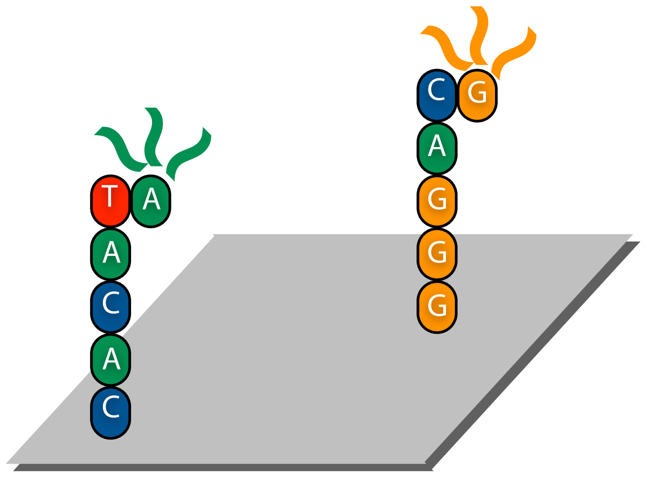


Figure from: Hunter, Lawrence. "Molecular biology for computer scientists." *Artificial intelligence and molecular biology* (1993): 1-46.

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

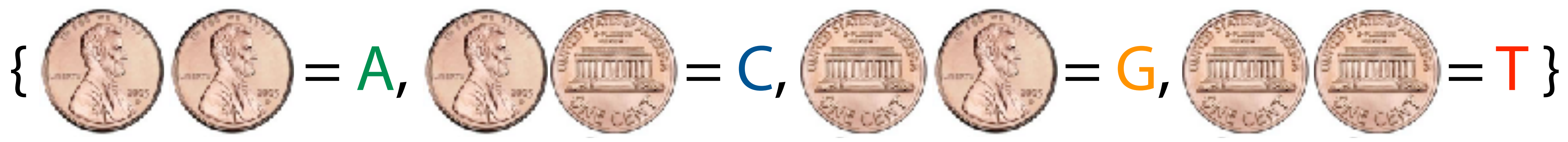
2. Lab procedures: PCR, Cell line passages



3. Sequencing: Fragmentation bias, Miscalled bases

Strings have structure

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



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

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

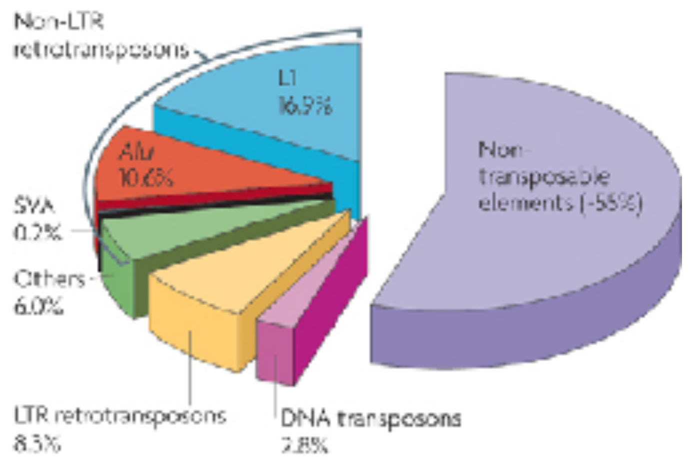


Image from: Cordaux R, Batzer MA. The impact of retrotransposons on human genome evolution. Nat Rev Genet. 2009 Oct;10(10):691-703

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

••• ATATATATATATAT •••

↕

••• ATATATATATATATATAT •••

String definitions

String S is a finite sequence of characters

Characters are drawn from alphabet Σ

Usually, $\Sigma = \{$ A, C, G, T $\}$

$|S|$ = number of characters in S

```
>>> s = 'ACGT'
>>> len(s)
4
```

ε is "empty string" $|\varepsilon| = 0$

```
>>> len('')
0
```

String definitions

Positions within a string *S* are referred to with *offsets*

```
>>> s = 'ACGT'
>>> s[0]
'A'
>>> s[2]
'G'
```

Leftmost offset = 0 in Python and most other languages

String definitions

Concatenation of *S* and *T* , *ST* = characters of *S* followed by characters of *T*

```
>>> s = 'AACC'
>>> t = 'GGTT'
>>> s + t
'AACCGGTT'
```

String definitions

Substring of *S* is a string occurring inside *S*

```
>>> s = 'AACCGGTT'
>>> s[2:6]
'CCGG' # substring of seq
```

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

String definitions

Prefix of *S* is a substring starting at the beginning of *S*

```
>>> s = 'AACCGGTT'
>>> s[0:6]
'AACCGG' # prefix
>>> s[:6] # same as above
'AACCGG'
```

S is a *prefix* of *T* if there exists a string *u* such that…

String definitions

Prefix of S is a substring starting at the beginning of S

```
>>> s = 'AACCGGTT'
>>> s[0:6]
'AACCGG' # prefix
>>> s[:6] # same as above
'AACCGG'
```

S is a *prefix* of T if there exists a string u such that $T = Su$

String definitions

Suffix is substring ending at end of *S*

```
>>> s = 'AACCGGTT'
>>> s[4:8]
'GGTT' # suffix
>>> s[4:] # like s[4:len(s)]
'GGTT'
>>> s[-4:] # like s[len(s)-4:len(s)]
'GGTT'
```

S is a *suffix* of *T* if there exists a string *u* such that…

String definitions

Suffix is substring ending at end of *S*

```
>>> s = 'AACCGGTT'
>>> s[4:8]
'GGTT' # suffix
>>> s[4:] # like s[4:len(s)]
'GGTT'
>>> s[-4:] # like s[len(s)-4:len(s)]
'GGTT'
```

S is a *suffix* of *T* if there exists a string *u* such that $T = uS$

String definitions

Usually assume alphabet Σ is finite, with $O(1)$ 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 }

Occasionally we'll consider what happens as $|\Sigma|$ grows

Exact matching

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

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

Alignment: a way of putting *P's* characters opposite *T's*.
May or may not correspond to an match.

```
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   <- One occurrence
    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
```

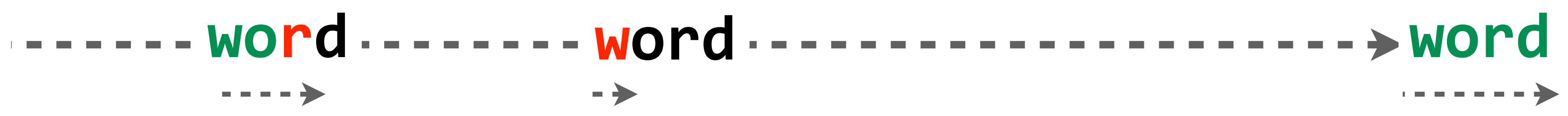
Try all possible alignments. For each, check if it matches.
This is the *naïve algorithm*.

Exact matching: naïve algorithm

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

P: **word**

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

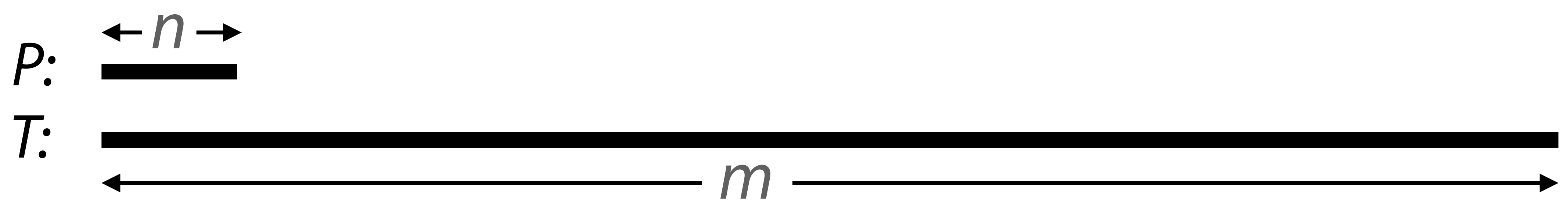


Exact matching: naïve algorithm

$n = |P| \quad m = |T|$

How many alignments are possible?

$m - n + 1$



Exact matching: naïve algorithm

$n = |P| \quad m = |T|$

Greatest # character comparisons possible?

$$n(m - n + 1)$$

```
P: aaaa
T: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
    aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
     aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
      aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
       aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
```

Exact matching: naïve algorithm

$n = |P| \quad m = |T|$

Least # character comparisons possible?

$m - n + 1$

```
P: abbb
T: bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
   abbb abbb abbb abbb abbb abbb abbb abbb abbb
    abbb abbb abbb abbb abbb abbb abbb abbb
     abbb abbb abbb abbb abbb abbb abbb abbb
      abbb abbb abbb abbb abbb abbb abbb abbb
       abbb abbb abbb abbb abbb abbb abbb abbb
```

Exact matching: naïve algorithm

How many character comparisons in this example?

```
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 word
```

Hint: there are 41 possible alignments

Exact matching: naïve algorithm

How many character comparisons in this example?

```
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 word
```

40 mismatches + 6 matches = 46 character comparisons

Closer to the minimum (41) than the maximum (164)

Exact matching: naïve algorithm

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

Even more naïve:
remove **break**

Exact matching: better algorithms?

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

u doesn't occur in *P*, so skip next two alignments

```
P: word
T: There would have been a time for such a word
    -------word ------------------------------------->
            word    skip!
             word     skip!
              word
```

We'll take such ideas further when we discuss Boyer-Moore