1. Exact string matching algorithms
2. Boyer Moore
3. Indexing structures

Lesson 06

Exact string matching algorithms

- Online exact matching algorithms (no preparation of the text)
    - Boyer-Moore
- Offline exact matching algorithms (prepare some kind of index of the text)
    - Multimap table
    - Hash table
    - Suffix array
    - Tries, suffix tries
    - Suffix trees
    - Suffix arrays
    - Burrows–Wheeler transformation, FM index

Slide adapted from Ben Langmead

String definitions

- String S is a finite sequence of characters
- Characters are drawn from alphabet Σ:
  Usually, Σ = { A, C, G, T }
- | S | = number of characters in S
- ε is “empty string” | ε | = 0

Exact matching

- At what offsets does pattern P occur within text T?
- What's a simple algorithm for exact matching?

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

  Pattern: word Answer: 40

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

Exact matching: Naïve algorithm

- Looking for places where a pattern P occurs as a substring of a Text
- 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:

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

P: word

Alignment 1: word

Alignment 2: word

Slide adapted from Ben Langmead

Exact matching

```python
def naive(p, t):
    occurrences = []
       for i in range(len(t) - len(p) + 1): # loop over alignments, L-to-R
        match = True
        for j in range(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

There would have been a time for such a word

--------word------------word----------------word

```
       --------->        --------->              --------->
```

Slide adapted from Ben Langmead

Exact matching: Naïve algorithm

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

m - n +1

- What is the lowest and greatest number of possible character comparisons?

m - n +1, n(m - n + 1)

- How many character comparisons in this example?

  There would have been a time for such a word

  --------word------------word---------------word

m - n mismatches, 6 matches

Exact matching: Naïve algorithm

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

Least:
$m - n + 1$

```
P: aaaa
T: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
   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 aaaa aaaa aaaa aaaa
```

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

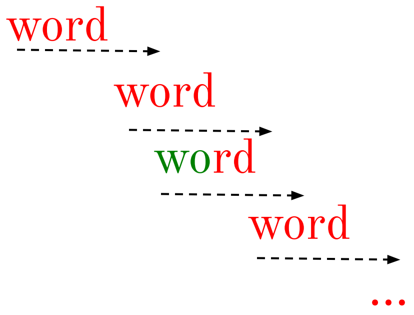
In the best case, we do only $\sim m$ character comparisons

Slide adapted from Ben Langmead

Exact matching: Can it be done better?

- Can we skip some alignments?
- Define some heuristic that could increase the shifts of pattern!
- Preprocess Text or Pattern and extract some information.

There would have been a time for such a word



Mismatch! Can we shift immediately by, let’s say, 4?

Online exact matching: Boyer Moore algorithm

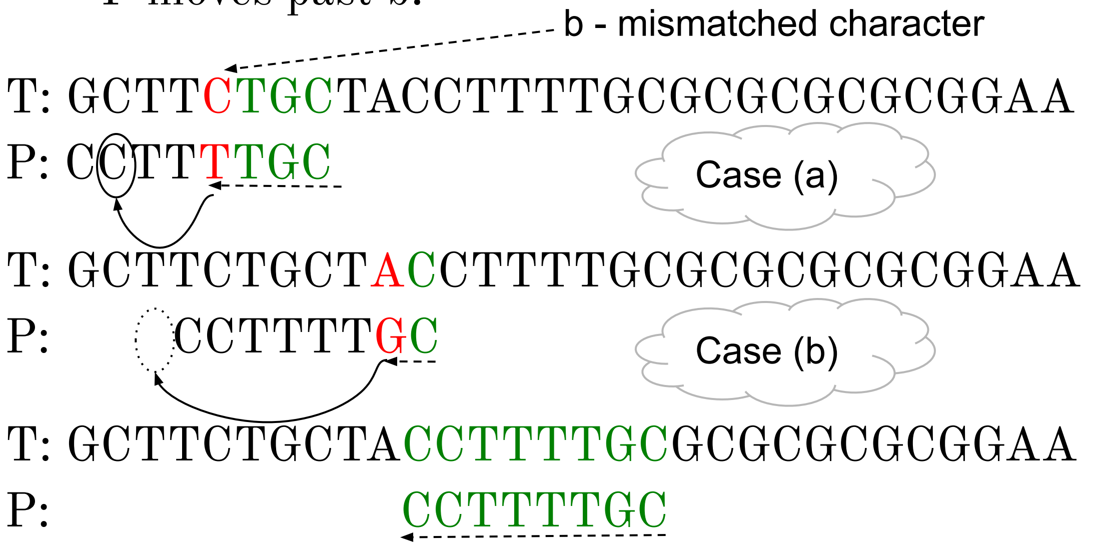
Use knowledge gained from character comparisons to skip future alignments that definitely won't match:

1. **Bad character rule**: If we mismatch, use knowledge of the mismatched text character to skip alignments
2. **Good suffix rule**: If we match some characters, use knowledge of the matched characters to skip alignments
3. **For longer skips**: If we match some characters, use knowledge of the matched characters to skip alignments

Boyer, RS and Moore, JS. "A fast string searching algorithm." *Communications of the ACM* 20.10 (1977): 762-772.

Boyer Moore: Bad character rule

- Upon mismatch, let b be the mismatched character in T.
  Skip alignments until (a) b matches its opposite in P, or (b)
  P moves past b.



```
T: GCTTCTGCTACCTTTTGCGCGCGCGGAA
P: CCTTTTGC

T: GCTTCTGCTACCTTTTGCGCGCGCGGAA
P:    CCTTTTGC

T: GCTTCTGCTACCTTTTGCGCGCGCGGAA
P:           CCTTTTGC
```

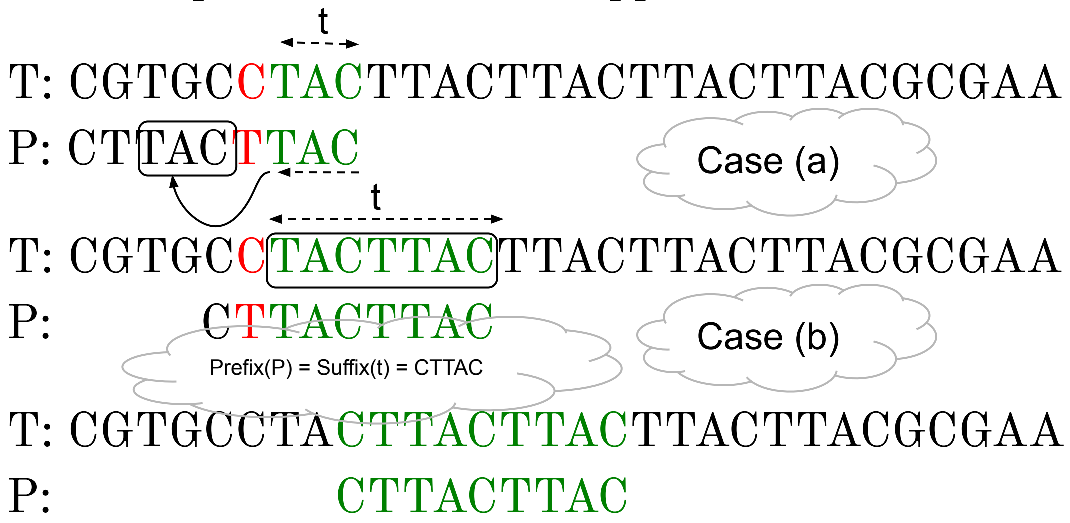
Compare characters from right to left!

We skipped 2 + 6 alignments!

Slide adapted from Ben Langmead

Boyer Moore: Good suffix rule (weak)

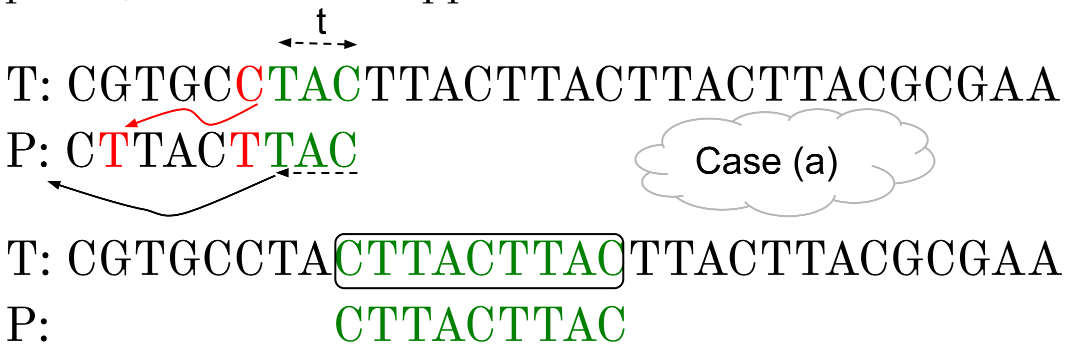
Let t be the substring of T that matched a suffix of P. Skip alignments until (a) t matches opposite characters in P, or (b) a prefix of P matches a suffix of t, or (c) P moves past t, whichever happens first.


t
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
Case (a)
t
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
Case (b)
Prefix(P) = Suffix(t) = CTTAC
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC

Slide adapted from Ben Langmead

Boyer Moore: Good suffix rule (strong)

Let t be the substring of T that matched a suffix of P. Skip alignments until (a) t matches opposite characters in P and character to the left of t is the same as it opposite character in P, or (b) a prefix of P matches a suffix of t, or (c) P moves past t, whichever happens first.



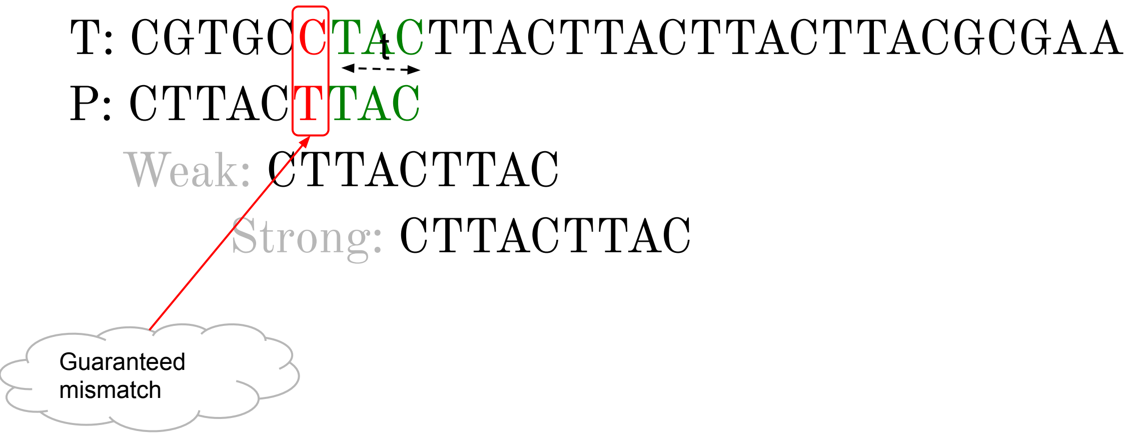
```
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC

T: CGTGCCTACTTACTTACTTACTTACGCGAA
P:         CTTACTTAC
```

Slide adapted from Ben Langmead

Boyer Moore: Good suffix rule

Like with the bad character rule, the number of skips possible using the good suffix rule can be precalculated into a few tables (Gusfield 2.2.4 and 2.2.5)

Strong good suffix rule (Gusfield 2.2.3)



```
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
    Weak: CTTACTTAC
           Strong: CTTACTTAC
```

Boyer Moore: Putting it all together

After each alignment, use bad character or good suffix rule, whichever skips more.

Bad character rule:
Upon mismatch, let b be the mismatched character in T. Skip alignments until (a) b matches its opposite in P, or (b) P moves past b.

Good suffix rule:
Let t be the substring of T that matched a suffix of P. Skip alignments until (a) t matches opposite characters in P, or (b) a prefix of P matches a suffix of t, or (c) P moves past t, whichever happens first.

```
T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P: GTAGCGGCG                        bc: 6, gs: 0 (Part (a) of bad character rule)

T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P:        GTAGCGGCG                 bc: 0, gs: 2 (Part (b) of good suffix rule)

T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P:           GTAGCGGCG              bc: 2, gs: 7 (Part (b) of good suffix rule)

T: GTTATAGCTGATCGCGGCGTAGCGGCGAA
P:                   GTAGCGGCG
```

*15 alignments skipped, 11 text characters never examined

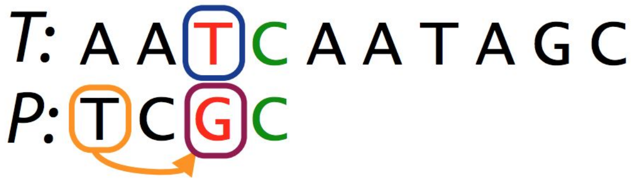
Slide adapted from Ben Langmead

Boyer Moore: Preprocessing

- Pre-calculate skips. For bad character rule, P = TCGC:

P

Σ	T	C	G	C
A	0	1	2	3
C	0	-	0	-
G	0	1	-	0
T	-	0	1	2



Boyer-Moore exercise in Python

Slide adapted from Ben Langmead

Boyer Moore: Worst and best cases

Boyer-Moore (or a slight variant) is O(m) worst-case time

What’s the best case?

Every character comparison is a mismatch, and bad character rule always slides P fully past the mismatch

How many character comparisons?

floor(|T| / |P|)

Slide adapted from Ben Langmead

Boyer Moore: Performance comparison

	Naïve matching		Boyer-Moore	
	# character comparisons	wall clock time	# character comparisons	wall clock time
P: "tomorrow" **T**: Shakespeare's complete works	5,906,125	2.90 s	785,855	1.54 s
P: 50 nt string from Alu repeat* **T**: Human reference (hg19) chromosome 1	307,013,905	137 s	32,495,111	55 s

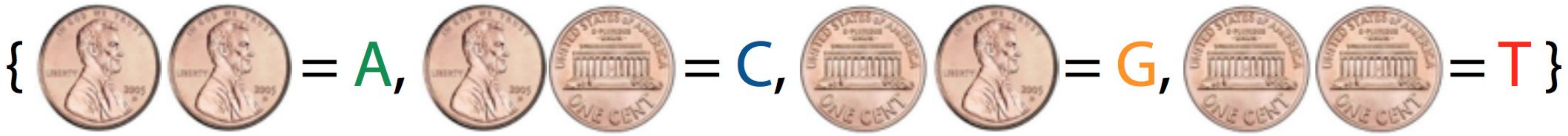
17 matches
| *T* | = 5.59 M

336 matches
| *T* | = 249 M

*
GCGCGGTGGCTCACGCCTGTAATCCCAGCACTTTGGGAGGCCGAGGCGGG

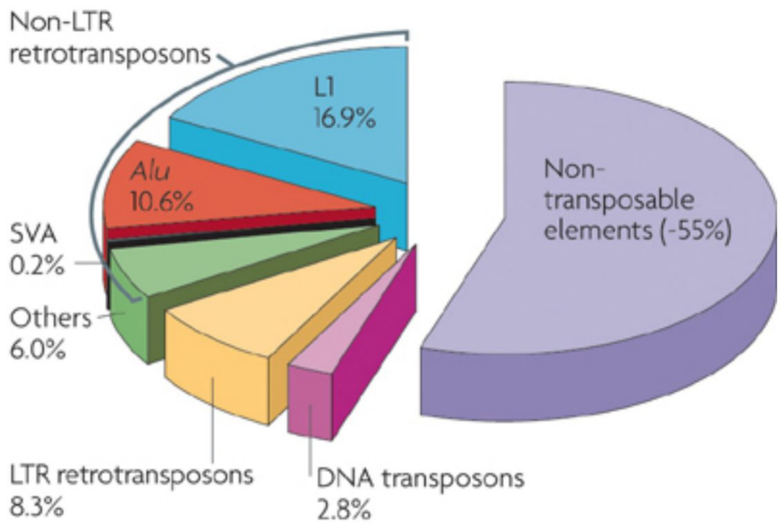
Small digression

- Real genomes are not like “random” genomes



Slide adapted from Ben Langmead

Repetitive sequences

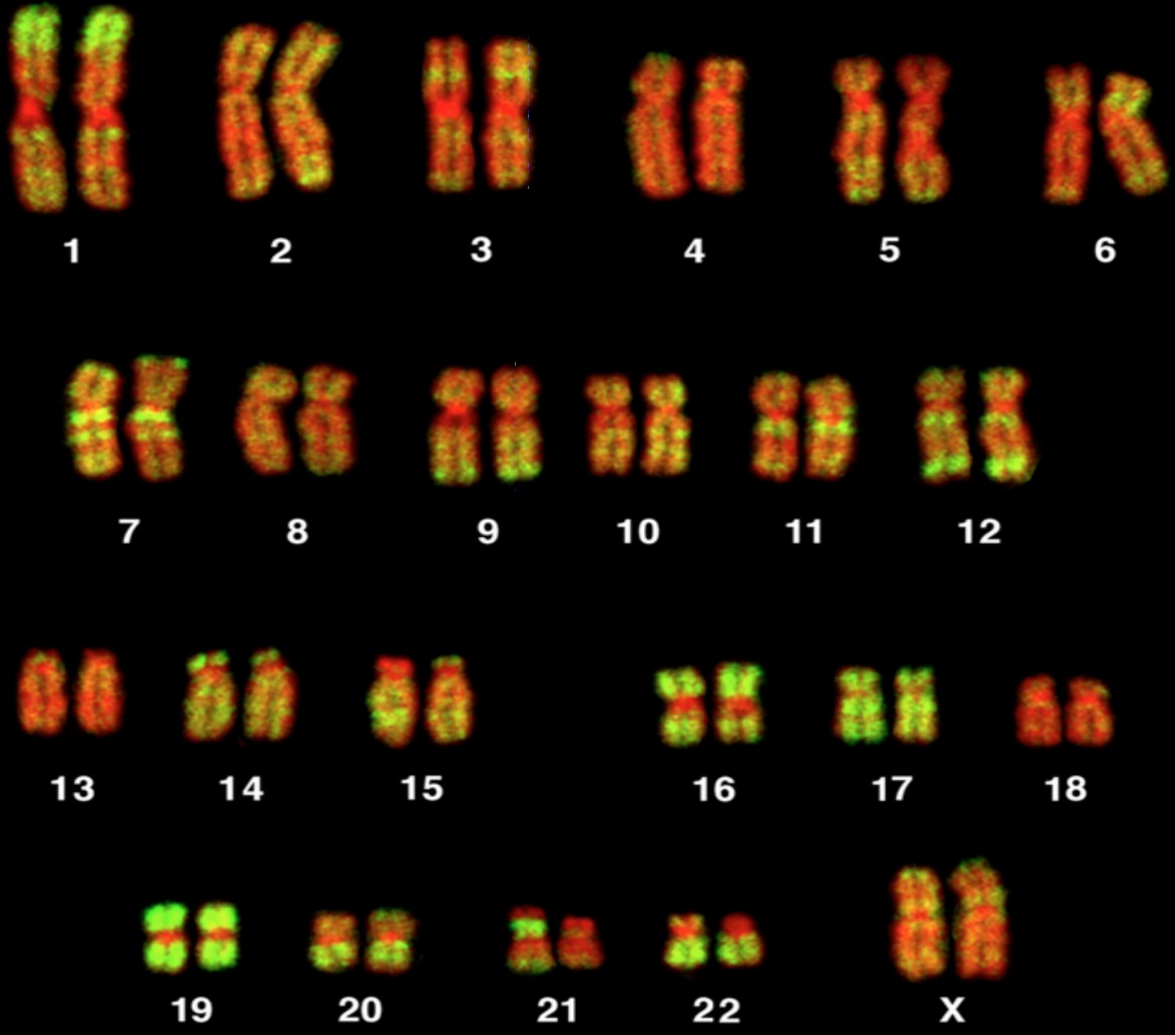


[human genetic identification based on short tandem repeats]

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

Alu repeats cover 11% of the human genome

Image: Alus glowing green in human chromosomes



Slide adapted from Ben Langmead

Offline exact matching: Create sorted index - Multimap

- T: $\underline{\text{CGTGC}}\text{GTGCTT}$

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

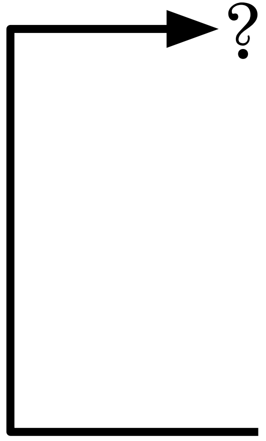
5-mer index

Slide adapted from Ben Langmead

Preprocessing: Use index

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

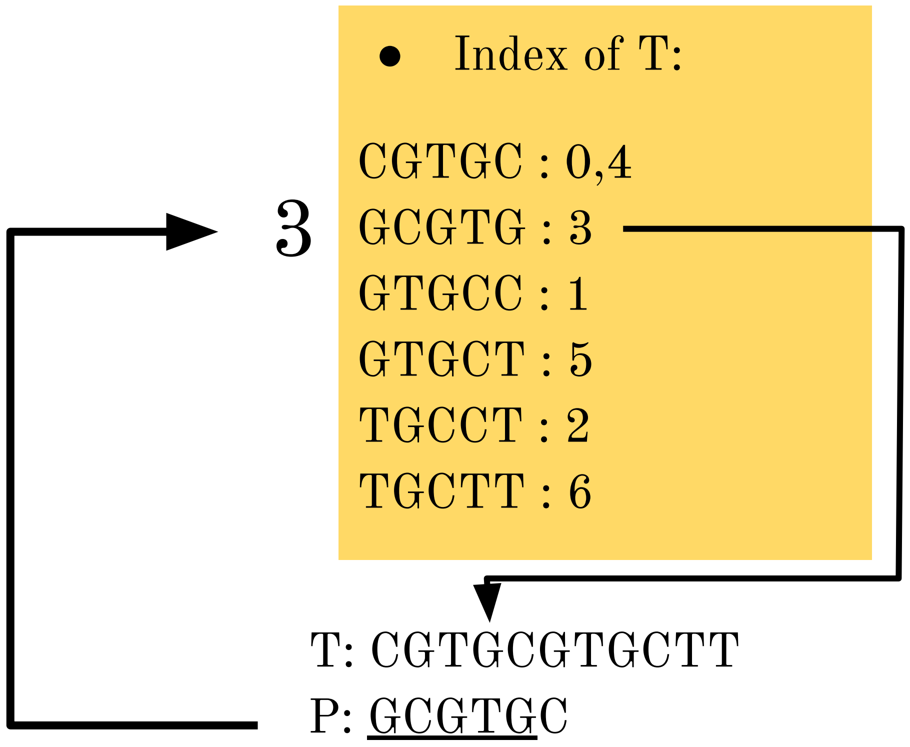


T: CGTGCGTGCTT

P: $\underline{\text{GCGTG}}\text{C}$

Slide adapted from Ben Langmead

Preprocessing: Use index



- Index of T:

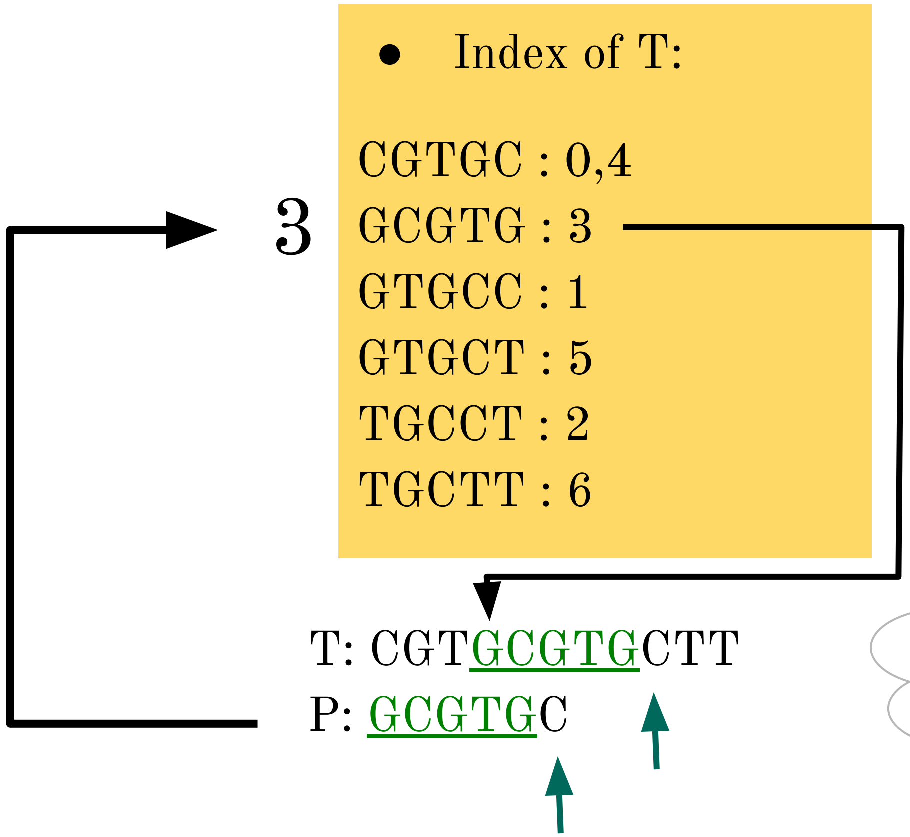
CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

T: CGTGCGTGCTT

P: GCGTGC

Slide adapted from Ben Langmead

Preprocessing: Use index



- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

3

T: CGT<u>GCGTG</u>CTT

P: <u>GCGTG</u>C

What about the rest?

Preprocessing: Use index - different 5-mer

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

0,4

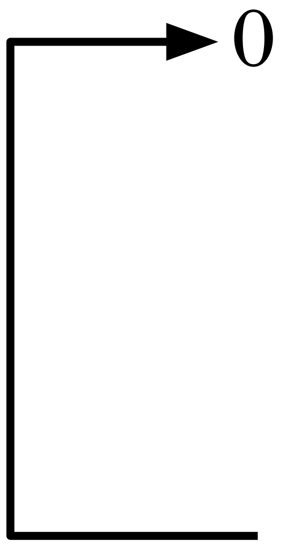
T: CGTGCGTGCTT

P: G$\underline{\text{CGTGC}}$

Preprocessing: Use index - different 5-mer

- Index of T:

CGTGC	: 0,4
GCGTG	: 3
GTGCC	: 1
GTGCT	: 5
TGCCT	: 2
TGCTT	: 6



T: CGTGCGTGCTT

P: GCGTGC

Preprocessing: Use index - different 5-mer

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

4

T: CGTGCGTGCTT

P: GCGTGC

Slide adapted from Ben Langmead

Preprocessing: Use index - different pattern

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6

3

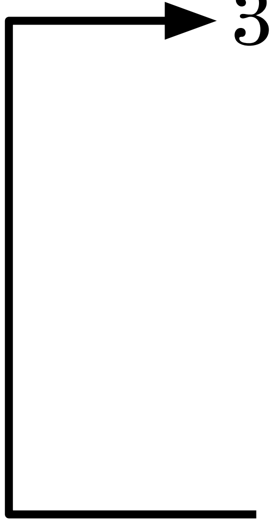
T: CGTGCGTGCTT

P: $\underline{\text{GCGTG}}$A

Preprocessing: Use index - different pattern

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6



T: CGTGCGTGCTT

P: GCGTGA

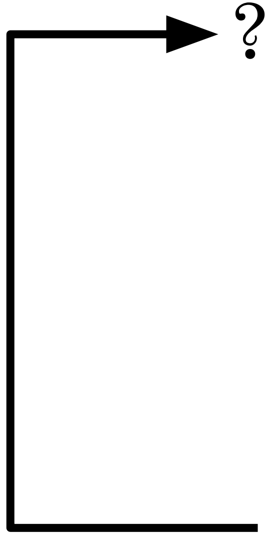
We had index hit but pattern mismatch!

Slide adapted from Ben Langmead

Preprocessing: Use index - different pattern

- Index of T:

CGTGC	0,4
GCGTG	3
GTGCC	1
GTGCT	5
TGCCT	2
TGCTT	6



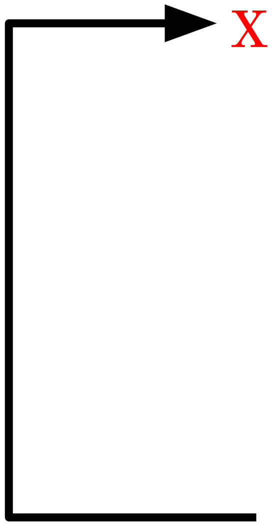
T: CGTGCGTGCTT

P: $\underline{\text{GCGTA}}$C

Preprocessing: Use index - different pattern

- Index of T:

CGTGC : 0,4
GCGTG : 3
GTGCC : 1
GTGCT : 5
TGCCT : 2
TGCTT : 6



We had index miss!

T: CGTGCGTGCTT
P: GCGTAC

Slide adapted from Ben Langmead

Multimap

GTG	0
TGC	1
GCG	2
CGT	3
GTG	4
TGT	5
GTG	6
TGG	7
GGG	8
GTG	9
GGG	10

- T: GTGCGTGTGGGGG

Slide adapted from Ben Langmead

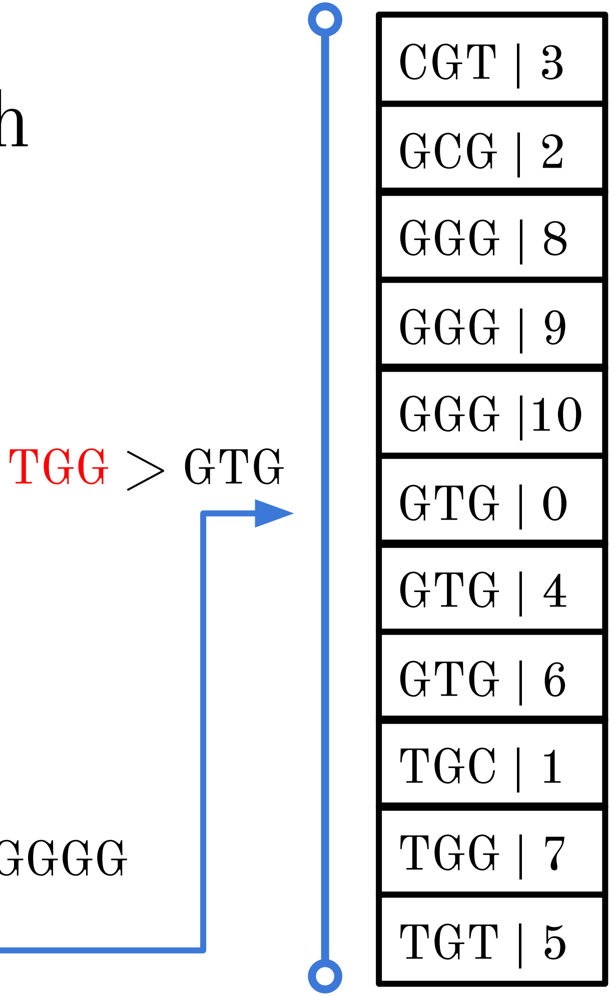
Multimap

Alphabetical by k-mer

CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

- T: GTGCGTGTGGGG

Binary search

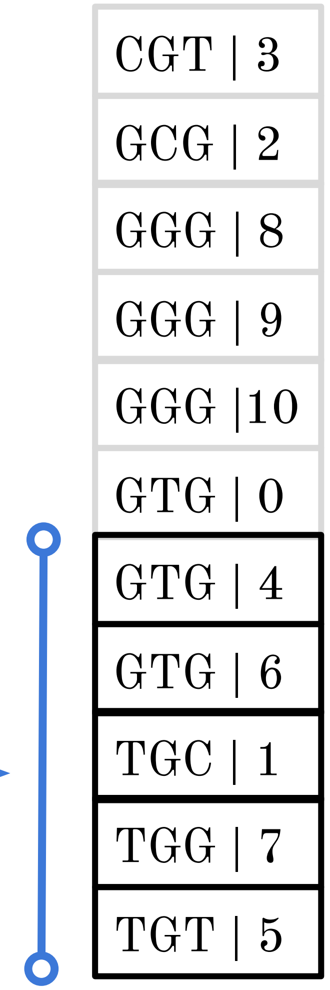


CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

- T: GTGCGTGTGGGG
- P: GCG<u>TGG</u>

Binary search

After 1st bisection



TGG > TGC

- T: GTGCGTGTGGGG
- P: GCGTGG

Slide adapted from Ben Langmead

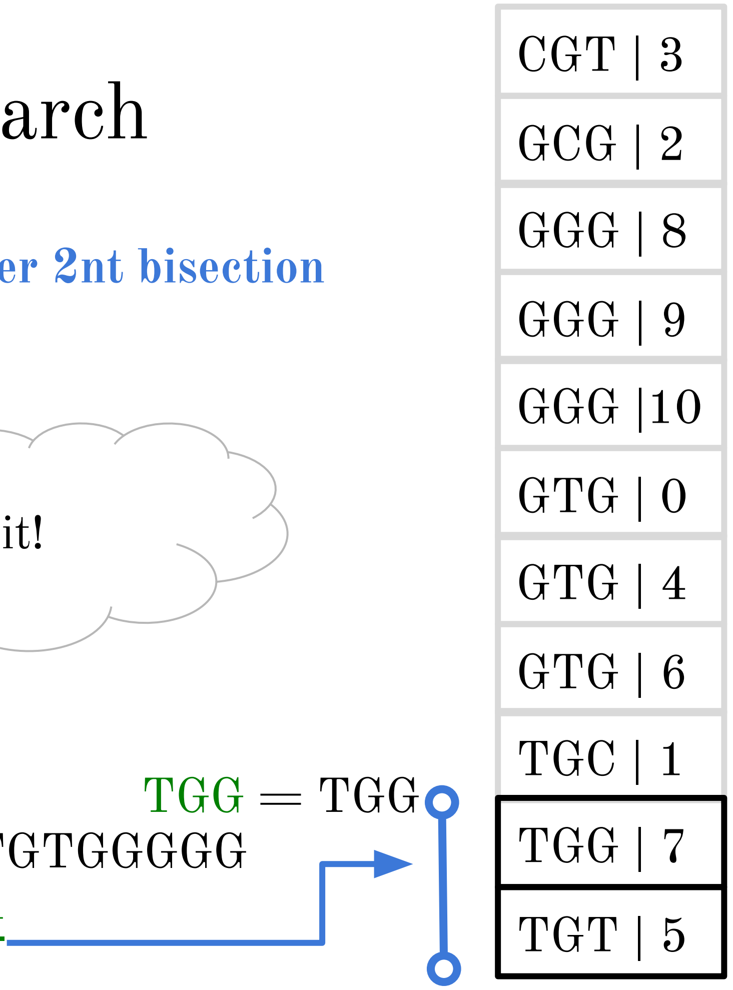
Binary search

After 2nt bisection

Hit!

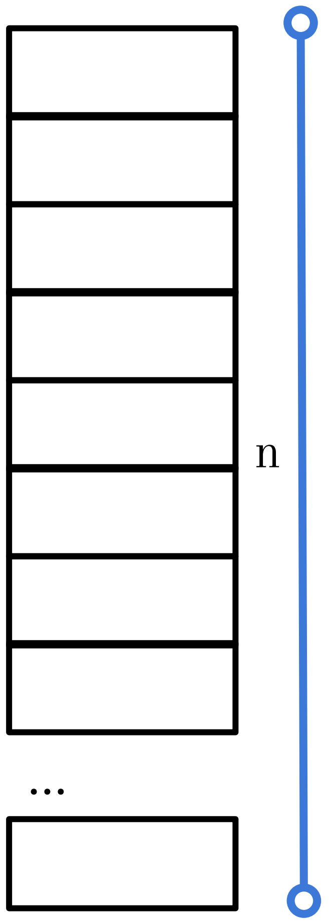
TGG = TGG

- T: GTGCGTGTGGGG
- P: GCG<u>TGG</u>



CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

Binary search



How many bisections per query?

$\log_2(n)$

Slide adapted from Ben Langmead

Binary search - python

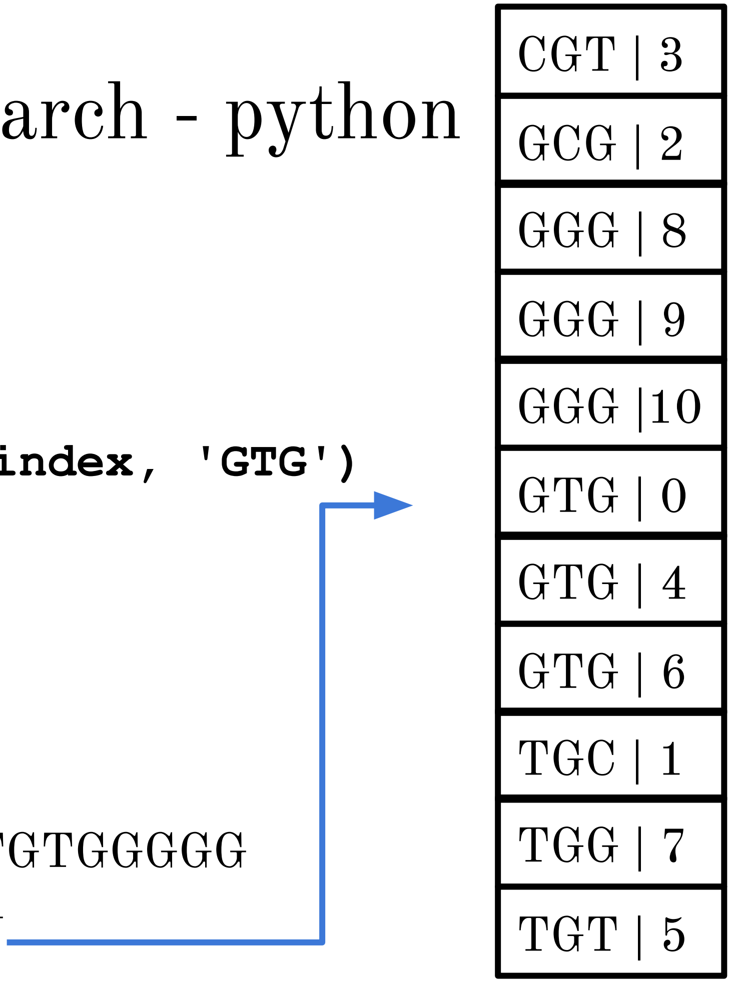
`bisect.bisect_left(a, x)`: Leftmost offset where x can be inserted into a to maintain order

```
>>> a  =  [1, 3,  3,  6,  8,  8,  9,  10]
>>> import bisect
>>> bisect.bisect_left(a,  2)
1
>>> bisect.bisect_left(a,  4)
3
>>> bisect.bisect_left(a,  8)
4
```

Binary search - python

```
bisect_left(index, 'GTG')
```

- T: GTGCGTGTGGGG
- P: GCGTGG



CGT	3
GCG	2
GGG	8
GGG	9
GGG	10
GTG	0
GTG	4
GTG	6
TGC	1
TGG	7
TGT	5

Index exercise in Python

Slide adapted from Ben Langmead

Indexing subsequences

- Subsequence of S: string of characters also occurring in S in the same order
- Substrings are also subsequences, subsequences are not necessarily substrings

```
>>> seq  = 'AACCGGTT'
>>>  seq[0] + seq[1] + seq[5] + seq[7]
'AAGT' # subsequence
>>>  seq.find('AAGT')
-1 #not  a substring
```

Slide adapted from Ben Langmead

Indexing subsequences

- Index of T:

CGGGT : 0
CGGTT : 4
GCTCT : 3
GTCTG : 1
TGGC : 2

Using subsequences improves specificity! Why?

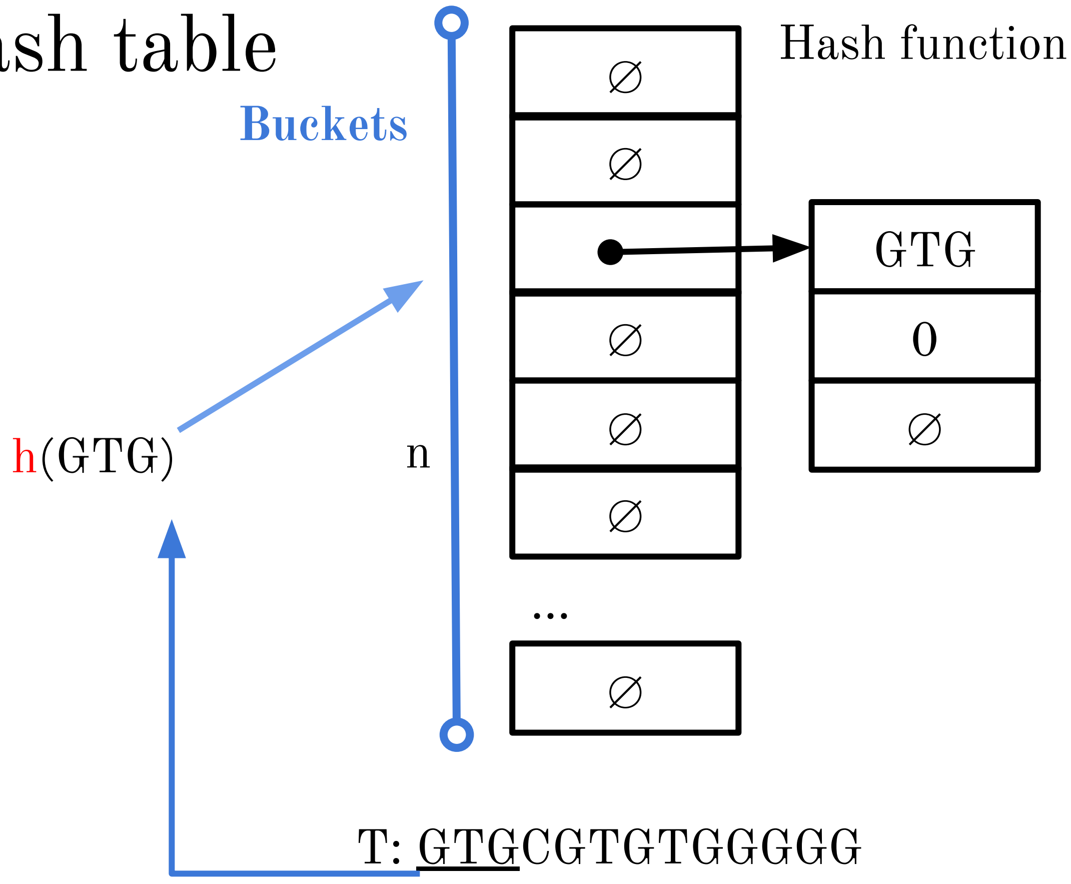
Find CGGGT

T: $\underline{\text{CG}}\text{T}\underline{\text{G}}\text{C}\underline{\text{GT}}\text{GCTT}$

Slide adapted from Ben Langmead

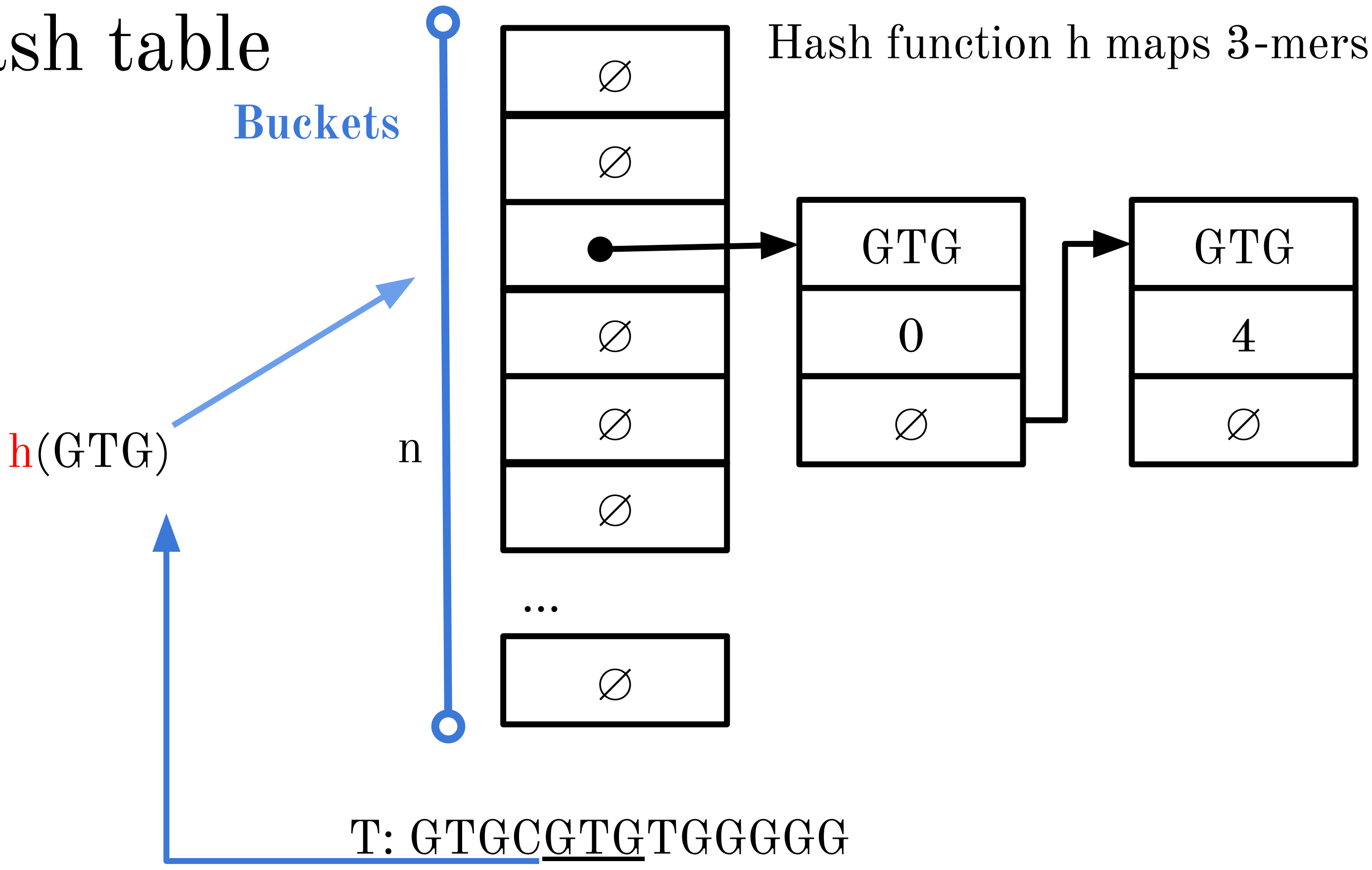
Hash table

Hash function h maps 3-mers to buckets



Hash table

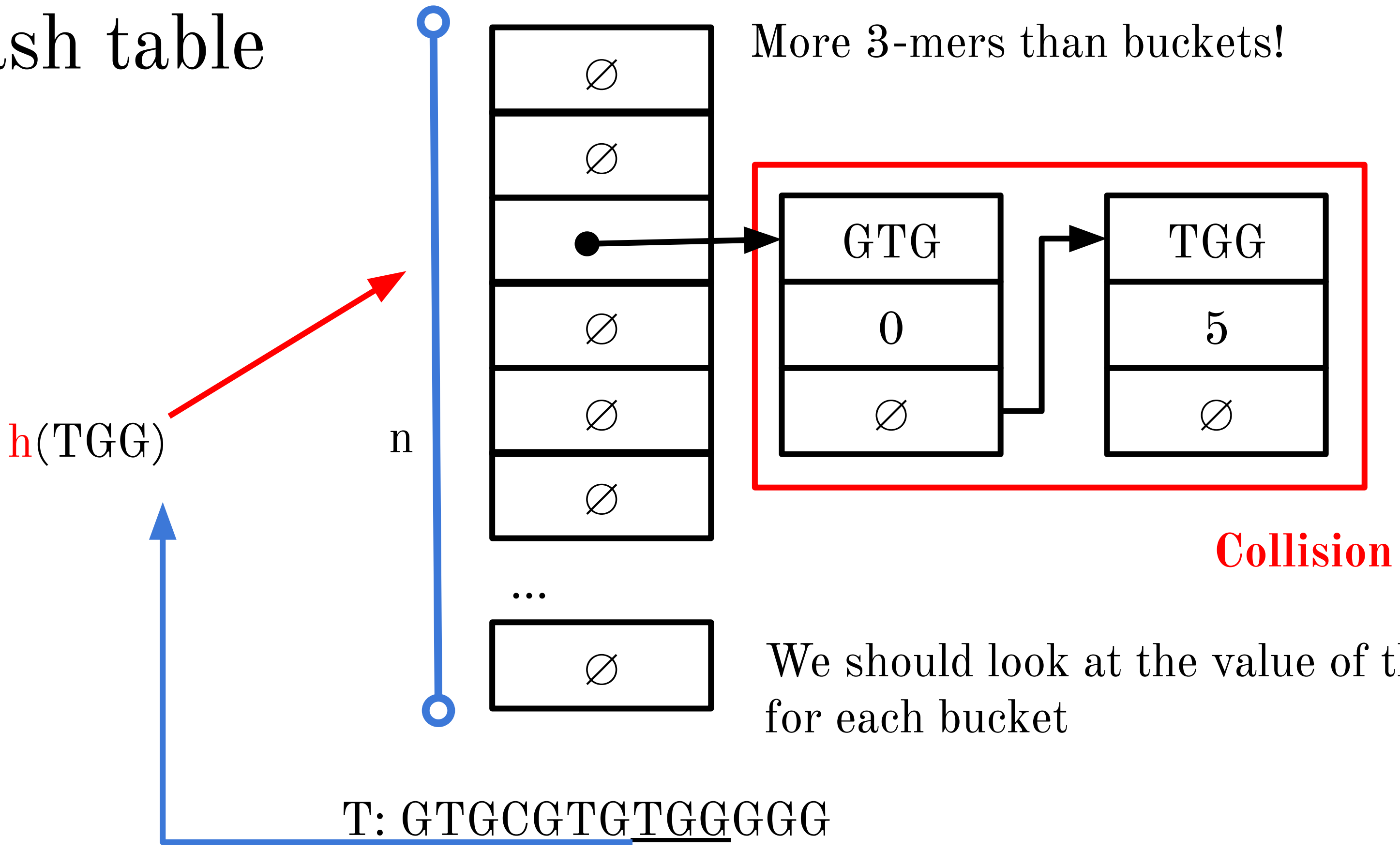
Hash function h maps 3-mers to buckets



Slide adapted from Ben Langmead

Hash table

More 3-mers than buckets!



We should look at the value of the key for each bucket

T: GTGCGTGTGGGG

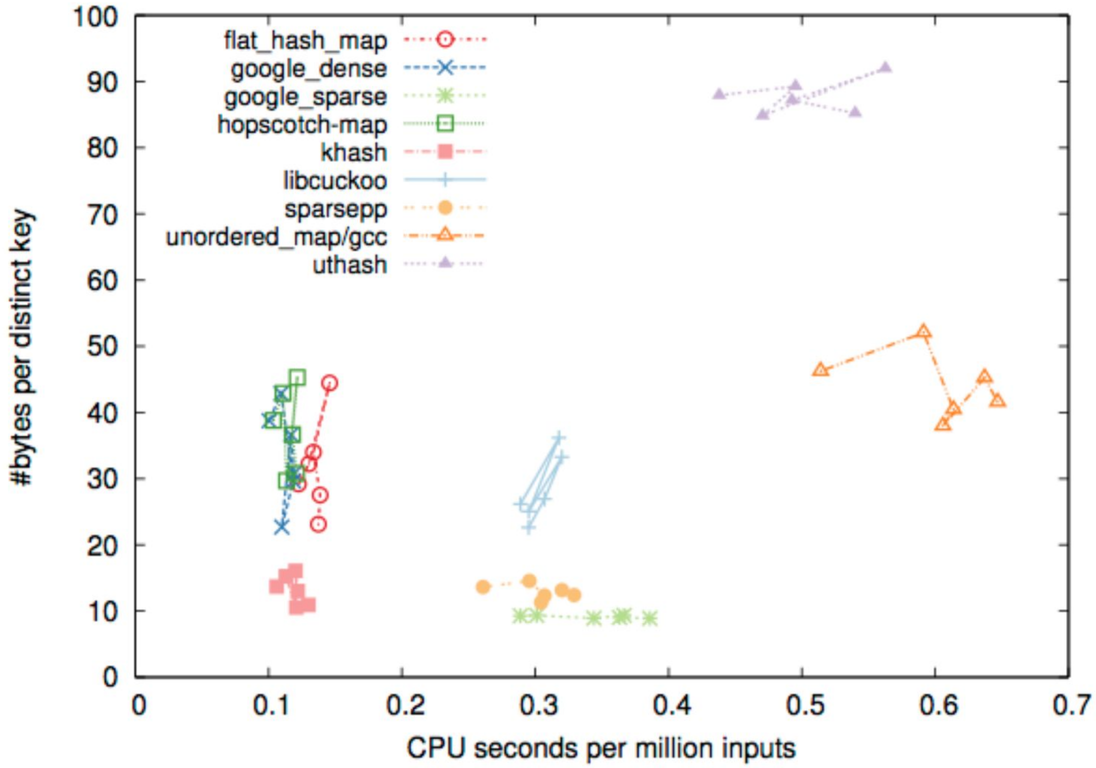
Dictionary in Python

```
>>> t = 'GTGCGTGTGGGG'
>>> table = {'GTG':[0, 4, 6], 'TGC':[1],
             'GCG':[2], 'CGT':[3], 'TGT':[5],
             'TGG':[7], 'GGG':[8, 9, 10]}
>>> table['GGG']
[8, 9,  10]
>>> table['CGT']
[3]
```

Hash table comparison

Each line - 6 dots, corresponding to N=10,18,26,34,42,50 million inputs

10 years ago google_danse was fastest



Revisiting hash table performance

Suffix index

T = GTTATAGCTGATCGCGGCGTAGCGG$

```
GTTATAGCTGATCGCGGCGTAGCGG$
 TTATAGCTGATCGCGGCGTAGCGG$
  TATAGCTGATCGCGGCGTAGCGG$
   ATAGCTGATCGCGGCGTAGCGG$
    TAGCTGATCGCGGCGTAGCGG$
     AGCTGATCGCGGCGTAGCGG$
      GCTGATCGCGGCGTAGCGG$
       CTGATCGCGGCGTAGCGG$
        TGATCGCGGCGTAGCGG$
         GATCGCGGCGTAGCGG$
          ATCGCGGCGTAGCGG$
           TCGCGGCGTAGCGG$
            CGCGGCGTAGCGG$
             GCGGCGTAGCGG$
              CGGCGTAGCGG$
               GGCGTAGCGG$
                GCGTAGCGG$
                 CGTAGCGG$
                  GTAGCGG$
                   TAGCGG$
                    AGCGG$
                     GCGG$
                      CGG$
                       GG$
                        G$
                         $
```

Slide adapted from Ben Langmead

Suffix Array

T = abaaba

abaaba
baaba
aaba
aba
ba
a

Alphabetical order

$$\xrightarrow{\hspace{3cm}}$$

a
aaba
aba
abaaba
ba
baaba

P = ab

- Querying uses binary search

Suffix index

T = GTTATAGCTGATCGCGGCGTAGCGG$

```
GTTATAGCTGATCGCGGCGTAGCGG$
 TTATAGCTGATCGCGGCGTAGCGG$
  TATAGCTGATCGCGGCGTAGCGG$
   ATAGCTGATCGCGGCGTAGCGG$
    TAGCTGATCGCGGCGTAGCGG$
     AGCTGATCGCGGCGTAGCGG$
      GCTGATCGCGGCGTAGCGG$
       CTGATCGCGGCGTAGCGG$
        TGATCGCGGCGTAGCGG$
         GATCGCGGCGTAGCGG$
          ATCGCGGCGTAGCGG$
           TCGCGGCGTAGCGG$
            CGCGGCGTAGCGG$
             GCGGCGTAGCGG$
              CGGCGTAGCGG$
               GGCGTAGCGG$
                GCGTAGCGG$
                 CGTAGCGG$
                  GTAGCGG$
                   TAGCGG$
                    AGCGG$
                     GCGG$
                      CGG$
                       GG$
                        G$
                         $
```

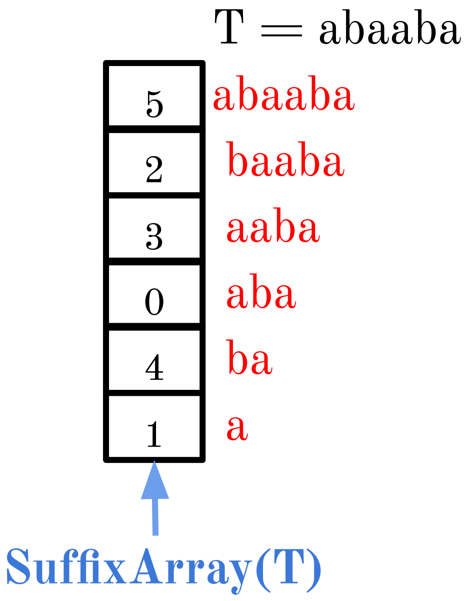
$n(n+1)/2$ chars $\approx (n2)/2$

Modern genomics algorithms still use Suffix index? How come?

- Imagine suffix index of 3 billion nucleotides long human reference genome

Slide adapted from Ben Langmead

Suffix array



Suffix array is |T| integers long

- Save in index only positions of suffixes in T

References

- Dan Gusfield: **Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology,** Cambridge University Press
- Pavel Pevzner, Neils Jones: **An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)**, MIT Press
- R. Durbin, S. Eddy, A. Krogh, G. Mitchinson: **Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids** , Cambridge University Press
- Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, Alexandru I. Tomescu: **Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing**, Cambridge University press