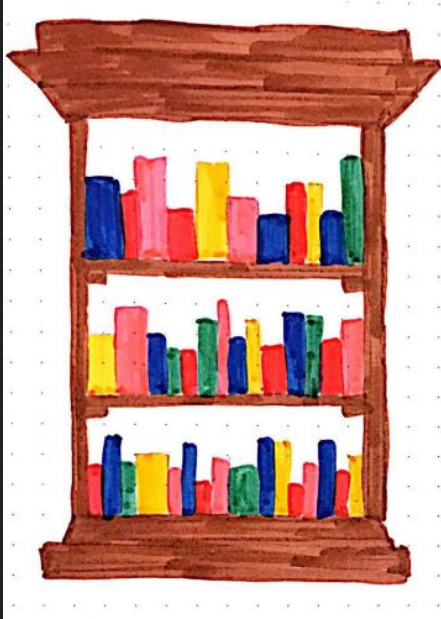


COMP90014

Algorithms for Bioinformatics

Week 1B - Indexing

Indexing



Pattern matching

Indexing

Hash Tables

Applying to Genomic Data

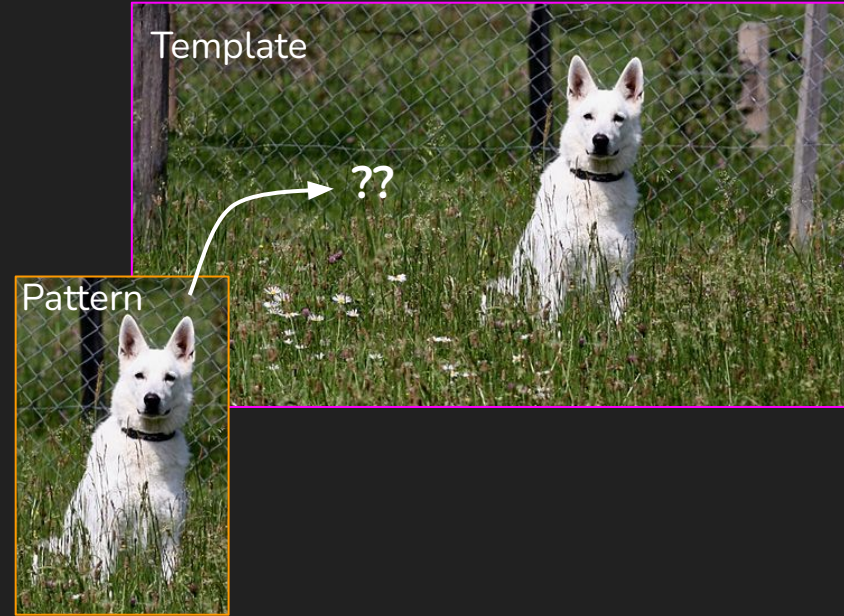
(K-mer hash tables)

Pattern Matching

Pattern Matching

Find where the pattern fits in the template

How did you find it?



Pattern Matching

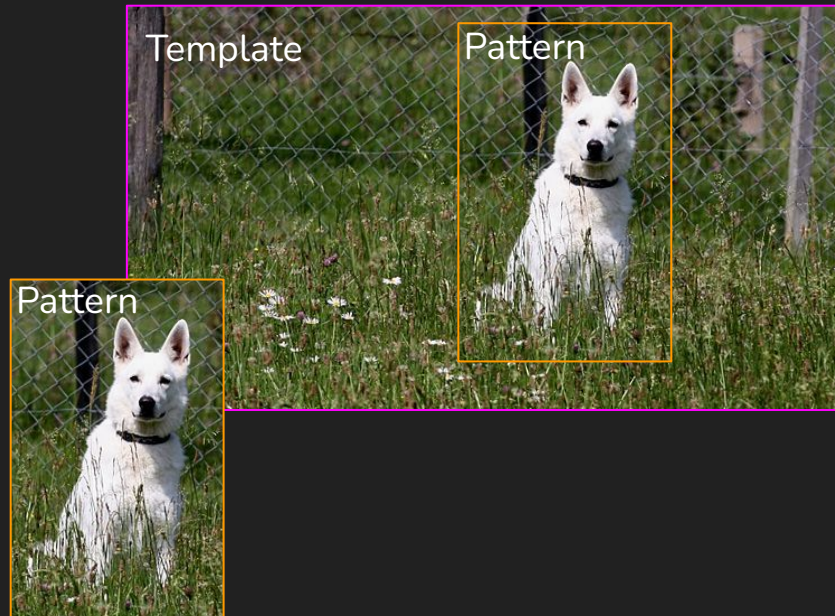
Find where the pattern fits in the template

How did you find it?

Humans are good at this image.

Advanced neural networks

Powerful pattern recognition *for familiar structures*



Pattern Matching

In bioinformatics, often want to...

Find where a **gene** is located in a genome

Find where a DNA sequencing **read** came from

Find if two DNA sequences **overlap**

Like finding the dog in the image, these involve pattern matching

gene 基因

genome 基因组

Genes are segments of your DNA,

Pattern
(DNA sequence)

A C T C

Template
(Genome)

T A C C G A G C

A C T T A T T A

G T A C T G C G

T A C T C G C G

Pattern Matching

Pattern matching for genomic data

Can't *"just do it"*

Let's write some rules

Series of steps describing our process

An *algorithm*, if you will.

Any Ideas?

Pattern

A C T C

Template

T A C C G A G C

A C T T A T T A

G T A C T G C G

T A C T C G C G

Pattern Matching

Pattern matching for genomic data

Can't *"just do it"*

Let's write some rules

Series of steps describing our process

An *algorithm*, if you will.

Any Ideas?

For each position in the template:

Match pattern & template

Add up score: +1 for each matching letter

Return the position with the highest score

Pattern

A C T C

Template

T A C C G A G C

A C T T A T T A

G T A C T G C G

T A C T C G C G

Pattern Matching

Pattern matching for genomic data

Can't *"just do it"*

Let's write some rules

Series of steps describing our process

An *algorithm*, if you will.

Any Ideas?

For each position in the template:

Match pattern & template

Add up score: +1 for each matching letter

Return the position with the highest score

Pattern



0...

N

Template



0...



...M

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location
best_score  $\leftarrow$  0;      # Tracking best score
N  $\leftarrow$  Len(pattern);    # Boundaries
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do
  i_score  $\leftarrow$  0;
  for j  $\leftarrow$  0...N do
    if pattern[j] == template[i + j] then
      | i_score += 1;
    end
  end
  if i_score > best_score then
    | best_score  $\leftarrow$  i_score;
    | best_loc  $\leftarrow$  i;
  end
end

return best_loc
```

Pattern

A C T C

0... N

best_loc: 0
best_score: 0

Template

T A C C G A G C

0...

A C T T A T T A

G T A C T G C G

T A C T C G C G

...M

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do  
  i_score  $\leftarrow$  0;  
  for j  $\leftarrow$  0...N do  
    if pattern[j] == template[i + j] then  
      | i_score += 1;  
    end  
  end  
  if i_score > best_score then  
    | best_score  $\leftarrow$  i_score;  
    | best_loc  $\leftarrow$  i;  
  end  
end
```

```
return best_loc
```

Pattern

A C T C

0...

N

best_loc: 0

best_score: 0

Template

T A C C G A G C

0...

A C T T A T T A

G T A C T G C G

T A C T C G C G

A C T

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do  
        if pattern[j] == template[i + j] then  
            i_score += 1;  
        end  
    end  
    if i_score > best_score then  
        best_score  $\leftarrow$  i_score;  
        best_loc  $\leftarrow$  i;  
    end  
end
```

```
return best_loc
```

Pattern

A C T C

0... N

best_loc: 0
best_score: 0

Template

T A C C G A G C

0...

A C T T A T T A

G T A C T G C G

T A C T C G C G

A C T C

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do  
        if pattern[j] == template[i + j] then  
            i_score += 1;  
        end  
    end  
    if i_score > best_score then  
        best_score  $\leftarrow$  i_score;  
        best_loc  $\leftarrow$  i;  
    end  
end
```

```
return best_loc
```

Pattern

A C T C

0... N

best_loc: 0
best_score: 0

Template

T A C C G A G C

0...

A C T T A T T A

G T A C T G C G

T A C T C G C G

...M

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location
best_score  $\leftarrow$  0;      # Tracking best score
N  $\leftarrow$  Len(pattern);   # Boundaries
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do           # Template position
```

```
    i_score  $\leftarrow$  0;
```

```
    for j  $\leftarrow$  0...N do
```

```
        if pattern[j] == template[i + j] then
```

```
            i_score += 1;
```

```
        end
```

```
    end
```

```
    if i_score > best_score then
```

```
        best_score  $\leftarrow$  i_score;
```

```
        best_loc  $\leftarrow$  i;
```

```
    end
```

```
end
```

```
return best_loc
```

Pattern

A	C	T	C
---	---	---	---

0... N

best_loc: 0
best_score: 0

Template

i=0

T	A	C	C	G	A	G	C
---	---	---	---	---	---	---	---

A	C	T	C
---	---	---	---

A	C	T	T	A	T	T	A
---	---	---	---	---	---	---	---

G	T	A	C	T	G	C	G
---	---	---	---	---	---	---	---

T	A	C	T	C	G	C	G
---	---	---	---	---	---	---	---

...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do           # Template position  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do        # Calculate score  
        if pattern[j] == template[i + j] then  
            | i_score += 1;  
        end  
    end  
    if i_score > best_score then  
        | best_score  $\leftarrow$  i_score;  
        | best_loc  $\leftarrow$  i;  
    end  
end  
return best_loc
```

Pattern

A C T C

0... N

best_loc: 0
best_score: 0

Template

i=0

T A C C G A G C

A C T C

j=0

A C T T A T T A

G T A C T G C G

T A C T C G C G

...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do      # Template position  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do    # Calculate score  
        if pattern[j] == template[i + j] then  
            i_score += 1;  
        end  
    end  
    if i_score > best_score then  
        best_score  $\leftarrow$  i_score;  
        best_loc  $\leftarrow$  i;  
    end  
end  
return best_loc
```

Pattern



0... *N*

best_loc: 0
best_score: 0

Template

i=0



j=0



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location
best_score  $\leftarrow$  0;      # Tracking best score
N  $\leftarrow$  Len(pattern);    # Boundaries
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do           # Template position
    i_score  $\leftarrow$  0;
    for j  $\leftarrow$  0...N do       # Calculate score
        if pattern[j] == template[i + j] then
            i_score += 1;
        end
    end
    if i_score > best_score then
        best_score  $\leftarrow$  i_score;
        best_loc  $\leftarrow$  i;
    end
end

return best_loc
```

Pattern



0... *N*

best_loc: 0
best_score: 0

Template

i=0



j=1



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location
best_score  $\leftarrow$  0;      # Tracking best score
N  $\leftarrow$  Len(pattern);    # Boundaries
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do           # Template position
    i_score  $\leftarrow$  0;
    for j  $\leftarrow$  0...N do        # Calculate score
        if pattern[j] == template[i + j] then
            i_score += 1;
        end
    end
    if i_score > best_score then
        best_score  $\leftarrow$  i_score;
        best_loc  $\leftarrow$  i;
    end
end

return best_loc
```

Pattern



0... *N*

best_loc: 0
best_score: 0

Template

i=0



j=2



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do           # Template position  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do        # Calculate score  
        if pattern[j] == template[i + j] then  
            i_score += 1;  
        end  
    end  
    if i_score > best_score then  
        best_score  $\leftarrow$  i_score;  
        best_loc  $\leftarrow$  i;  
    end  
end  
return best_loc
```

Pattern

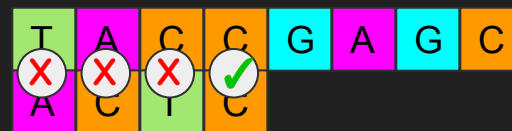


0... *N*

best_loc: 0
best_score: 0

Template

i=0



j=3



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location
best_score  $\leftarrow$  0;      # Tracking best score
N  $\leftarrow$  Len(pattern);    # Boundaries
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do      # Template position
    i_score  $\leftarrow$  0;
    for j  $\leftarrow$  0...N do    # Calculate score
        if pattern[j] == template[i + j] then
            i_score += 1;
        end
    end
    if i_score > best_score then
        best_score  $\leftarrow$  i_score;
        best_loc  $\leftarrow$  i;
    end
end

return best_loc
```

Pattern



0... *N*

best_loc: 0
best_score: 0

Template

i=0



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location
best_score  $\leftarrow$  0;      # Tracking best score
N  $\leftarrow$  Len(pattern);    # Boundaries
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do      # Template position
    i_score  $\leftarrow$  0;
    for j  $\leftarrow$  0...N do    # Calculate score
        if pattern[j] == template[i + j] then
            i_score += 1;
        end
    end
    if i_score > best_score then # Update
        best_score  $\leftarrow$  i_score;
        best_loc  $\leftarrow$  i;
    end
end

return best_loc
```

Pattern



0... *N*

best_loc: 0
best_score: 1

Template

i=0



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do      # Template position  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do    # Calculate score  
        if pattern[j] == template[i + j] then  
            i_score += 1;  
        end  
    end  
    if i_score > best_score then # Update  
        best_score  $\leftarrow$  i_score;  
        best_loc  $\leftarrow$  i;  
    end  
end  
return best_loc
```

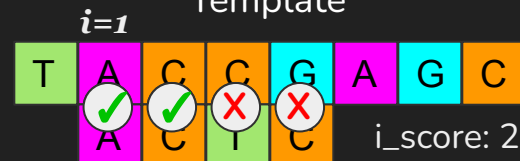
Pattern



0... *N*

best_loc: 1
best_score: 2

Template



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern*, *template*

Result: *best location*

```
best_loc  $\leftarrow$  0;           # Tracking best location  
best_score  $\leftarrow$  0;      # Tracking best score  
N  $\leftarrow$  Len(pattern);    # Boundaries  
M  $\leftarrow$  Len(template) - N;
```

```
for i  $\leftarrow$  0...M do      # Template position  
    i_score  $\leftarrow$  0;  
    for j  $\leftarrow$  0...N do    # Calculate score  
        if pattern[j] == template[i + j] then  
            i_score += 1;  
        end  
    end  
    if i_score > best_score then # Update  
        best_score  $\leftarrow$  i_score;  
        best_loc  $\leftarrow$  i;  
    end  
end  
return best_loc
```

Pattern



0... *N*

best_loc: 1
best_score: 2

Template
i=2



...*M*

Pattern Matching

Algorithm 1: Finding best location for pattern in template

Data: *pattern, template*

Result: *best location*

best_loc \leftarrow 0; # Tracking best location

best_score \leftarrow 0; # Tracking best score

N \leftarrow *Len(pattern)*; # Boundaries

M \leftarrow *Len(template)* - *N*;

for *i* \leftarrow 0...*M* do # Template position

i_score \leftarrow 0;

 for *j* \leftarrow 0...*N* do # Calculate score

 if *pattern*[*j*] == *template*[*i* + *j*] then

i_score += 1;

 end

 end

 if *i_score* > *best_score* then # Update

best_score \leftarrow *i_score*;

best_loc \leftarrow *i*;

 end

end

return *best_loc*

Pattern



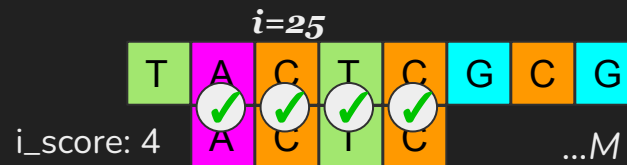
0... *N*

best_loc: 25
best_score: 4

Template



0...



Pattern Matching

Does our algorithm scale?

$O(N \times M)$



Small input size: ok

Pattern

A C T C

0...

N

Template

T A C C G A G C

0...

A C T T A T T A

G T A C T G C G

T A C T C G C G

...M

Pattern Matching

Does our algorithm scale?

$O(N \times M)$

Small input size: ok

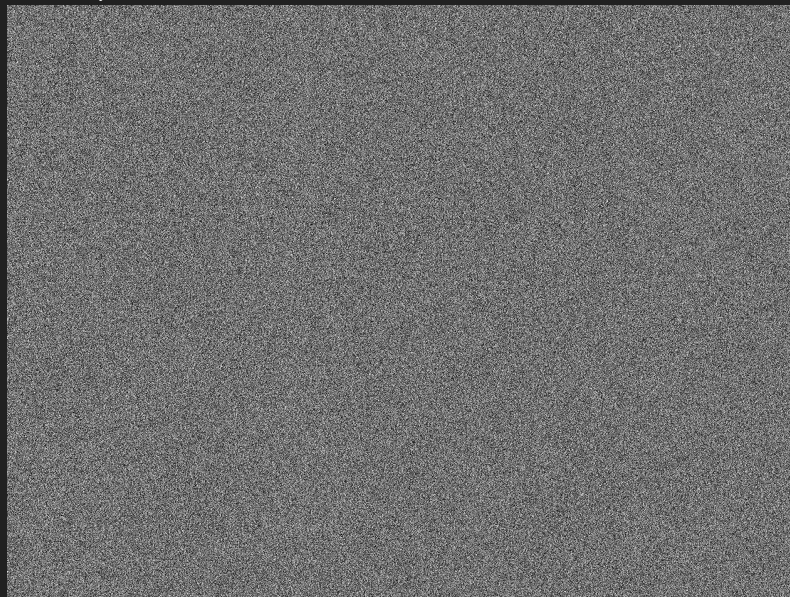
Data in bioinformatics is **BIG**

Pattern



40k pixels
(single long read)

Template



7.6M pixels
(bacterial genome)

Pattern Matching

Does our algorithm scale?

$O(N \times M)$

Small input size: ok

Data in bioinformatics is **BIG**

Template of 7.6M pixels?

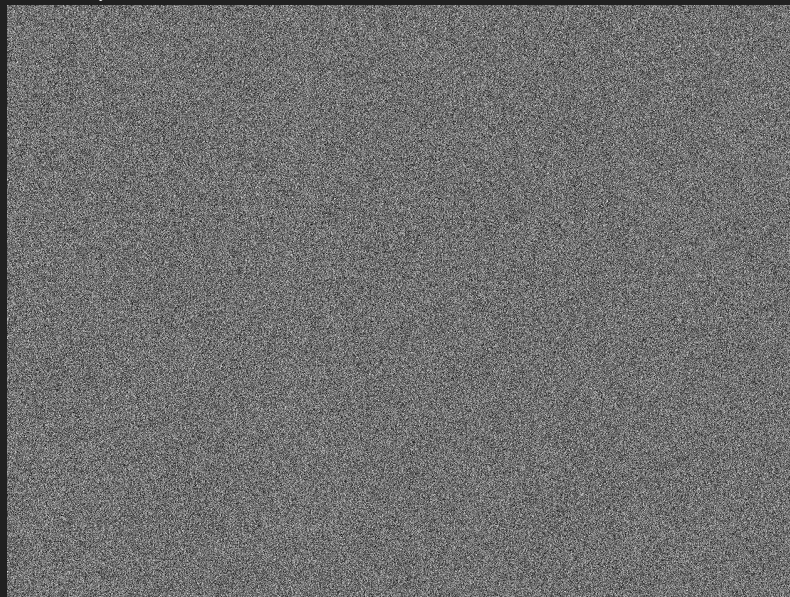
Child's play. **304 billion** operations.

Pattern



40k pixels
(single long read)

Template



7.6M pixels
(bacterial genome)

Pattern Matching

Does our algorithm scale?

$$O(N \times M)$$

Small input size: ok

Data in bioinformatics is **BIG**

Template of 7.6M pixels?

Child's play. **304 billion** operations.

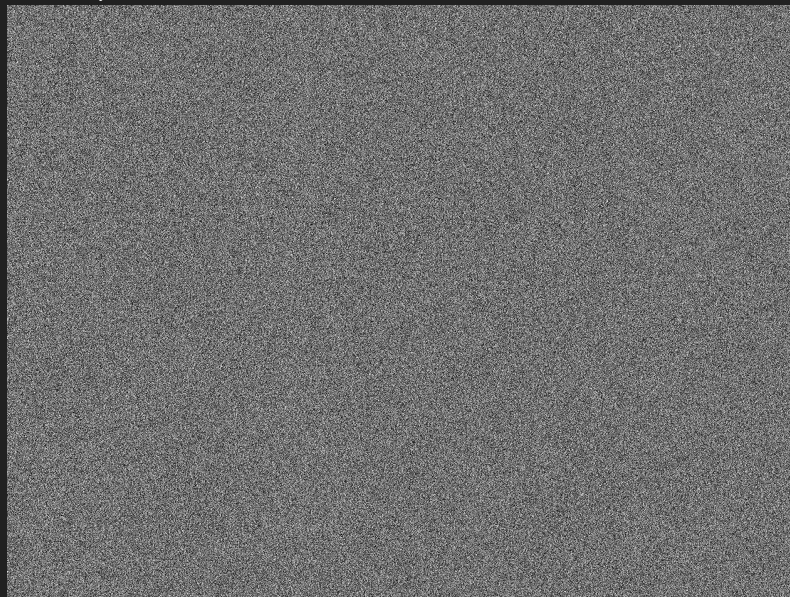
Try ~3B pixels :) (human genome)

Pattern



40k pixels
(single long read)

Template



7.6M pixels
(bacterial genome)

Pattern Matching

Does our algorithm scale?

$O(N \times M)$

Small input size: ok


Data in bioinformatics is **BIG**

Template of 7.6M pixels?

Child's play. **304 billion** operations.

Try ~3B pixels :) (human genome)

One hundred trillion operations.

Pattern

40k pixels
(single long read)

Template

7.6M pixels
(bacterial genome)

Pattern Matching

Does our algorithm scale?

$O(N \times M)$

Small input size: ok

Data in bioinformatics is **BIG**


Template of 7.6M pixels?

Child's play. **304 billion** operations.

Try ~3B pixels :) (human genome)

One hundred trillion operations.

Our approach doesn't scale to large input sizes.

Pattern

40k pixels
(single long read)

Template

7.6M pixels
(bacterial genome)

Pattern Matching

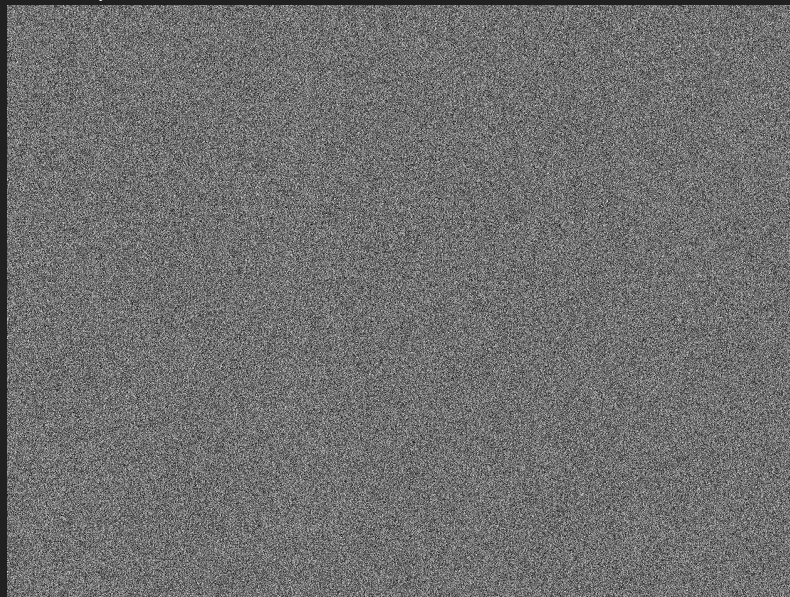
What if I told you, long read aligners...

Pattern



40k pixels
(single long read)

Template



7.6M pixels
(bacterial genome)

Pattern Matching

What if I told you, long read aligners...

Allow for shifts in the pattern & template
(indels)

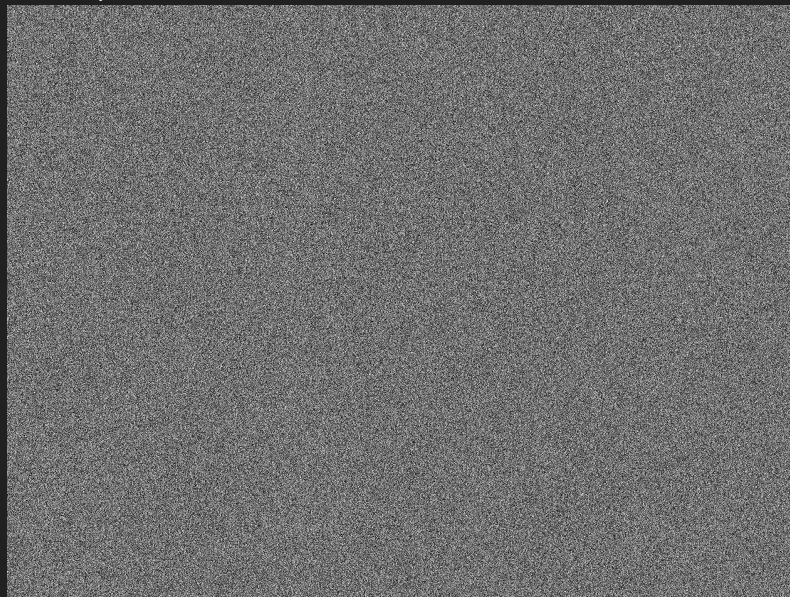
(orders of magnitude harder)

Pattern



40k pixels
(single long read)

Template



7.6M pixels
(bacterial genome)

Pattern Matching

What if I told you, long read aligners...

Allow for shifts in the pattern & template
(indels)

(orders of magnitude harder)

Allow for the pattern to be chopped up in the
template

(structural variants)

(orders of magnitude harder)

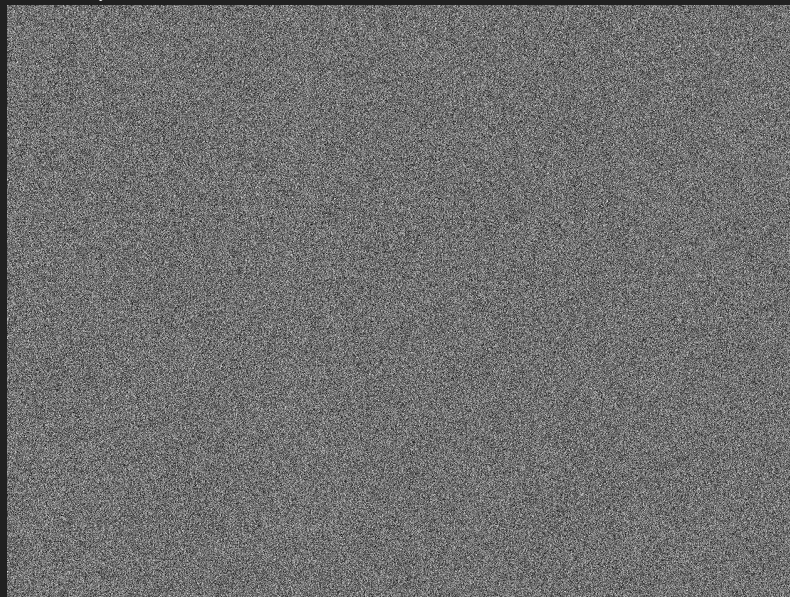
Pattern



40k pixels

(single long read)

Template



7.6M pixels

(bacterial genome)

Pattern Matching

What if I told you, long read aligners...

Allow for shifts in the pattern & template
(indels)

(orders of magnitude harder)

Allow for the pattern to be chopped up in the
template

(structural variants)

(orders of magnitude harder)

...Can do this in literally 1 second.

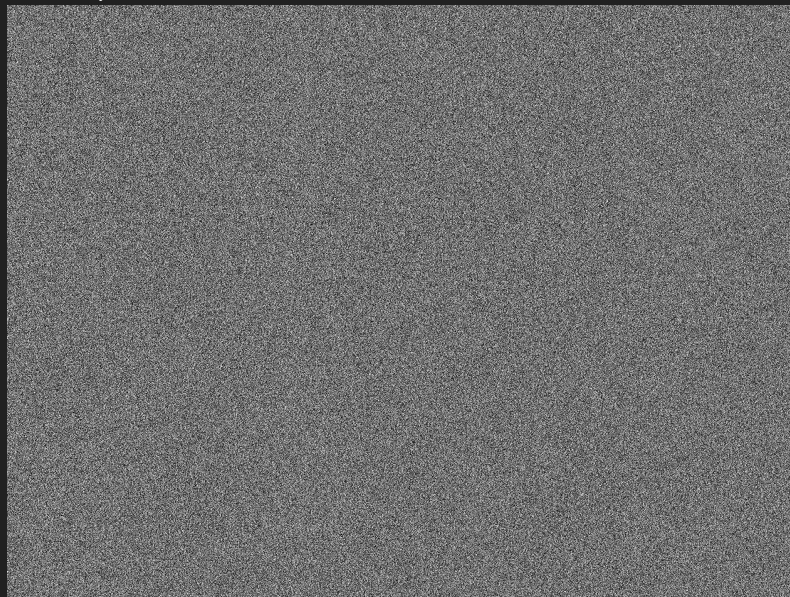
Pattern



40k pixels

(single long read)

Template



7.6M pixels

(bacterial genome)

Pattern Matching

What if I told you, long read aligners...

Allow for shifts in the pattern & template
(indels)

(orders of magnitude harder)

Allow for the pattern to be chopped up in the
template

(structural variants)

(orders of magnitude harder)

...Can do this in literally 1 second.

How?

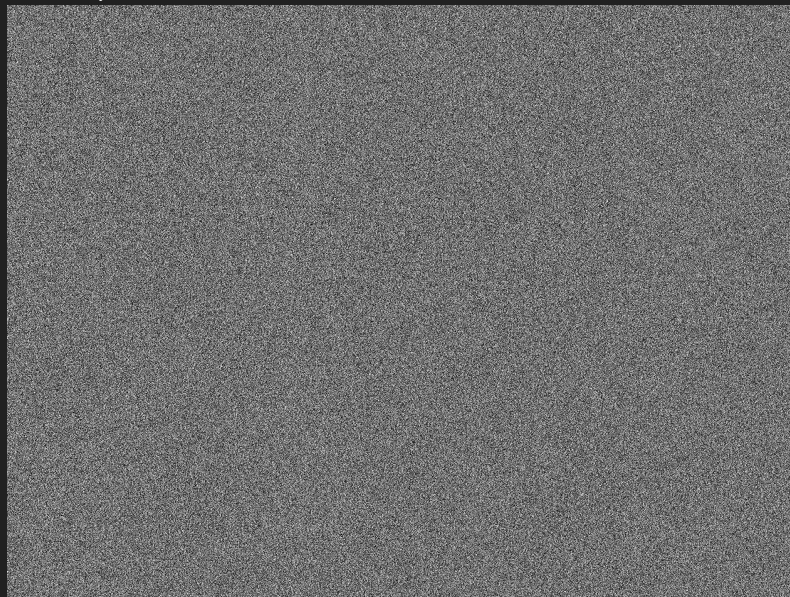
Pattern



40k pixels

(single long read)

Template



7.6M pixels

(bacterial genome)

Pattern Matching

What if I told you, long read aligners...

Allow for shifts in the pattern & template
(indels)

(orders of magnitude harder)

Allow for the pattern to be chopped up in the
template

(structural variants)

(orders of magnitude harder)

...Can do this in literally 1 second.

How?

Let's talk about a trick they use: [Indexing](#).

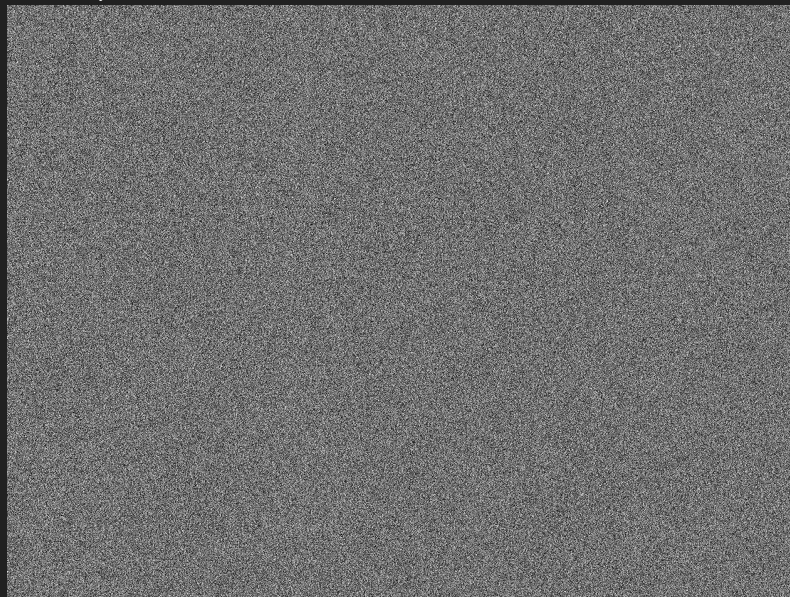
Pattern



40k pixels

(single long read)

Template



7.6M pixels

(bacterial genome)

允许模式和模板中的位移 (Allow for shifts in the pattern & template) :

在DNA序列对齐中，需要考虑插入 (insertions) 和删除 (deletions)，统称为INDELs。这些改变在序列中造成“位移”，对算法来说是一大挑战，因为需要处理的复杂性大大增加。

允许模式在模板中被切割 (Allow for the pattern to be chopped up in the template) :

这指的可能是结构变异 (structural variants)，比如DNA序列中的重复、删除、倒置等。这些也大大增加了对齐的难度。可以在1秒内完成：

这个声明表明，有些高效的算法或系统可以非常快地完成这些复杂的对齐任务。

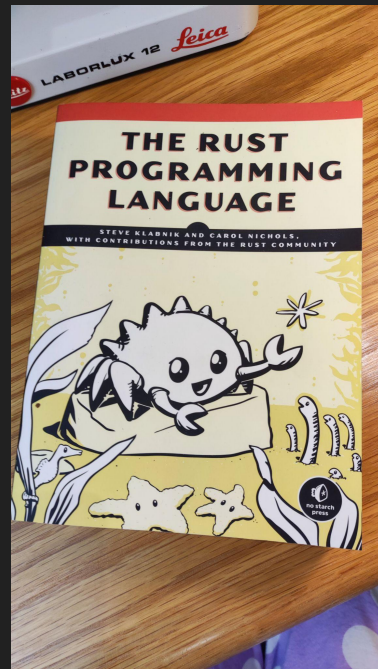
怎么做？索引 (Indexing) :

索引是一种常用于快速数据检索的技术。在生物信息学中，通过建立基因组的索引，可以快速地将读取序列与参考序列进行比对。这个过程大大加快了模式匹配的速度。

Indexing

Indexing

Let's say you bought a book about Rust.

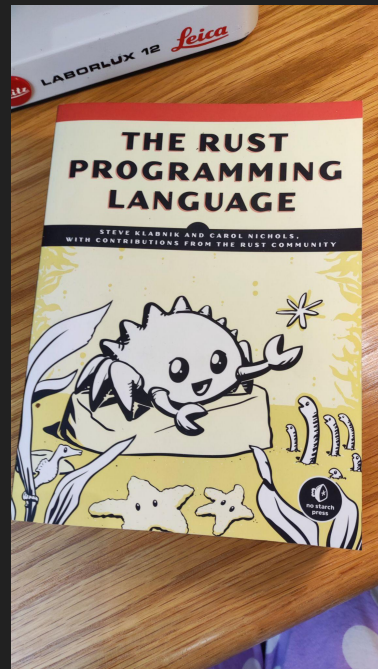


Indexing

Let's say you bought a book about Rust.

Want to know how to do “addition” in rust.

How do you find where “addition” is mentioned in the book?



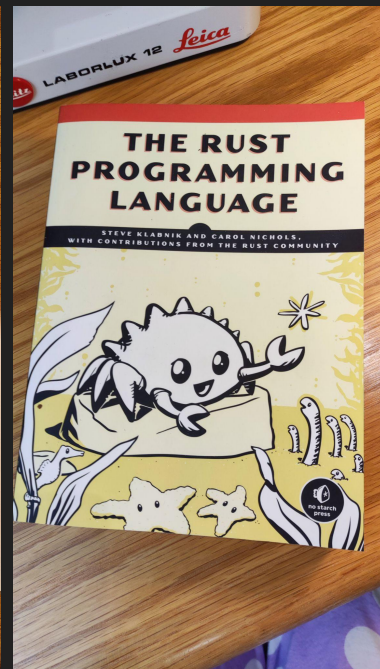
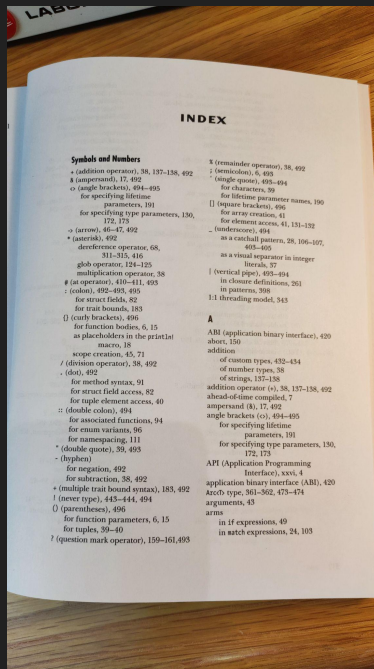
Indexing

Let's say you bought a book about Rust.

Want to know how to do “addition” in rust.

How do you find where “addition” is mentioned in the book?

Check the [index](#)



Indexing

Let's say you bought a book about Rust.

Want to know how to do “addition” in rust.

How do you find where “addition” is mentioned in the book?

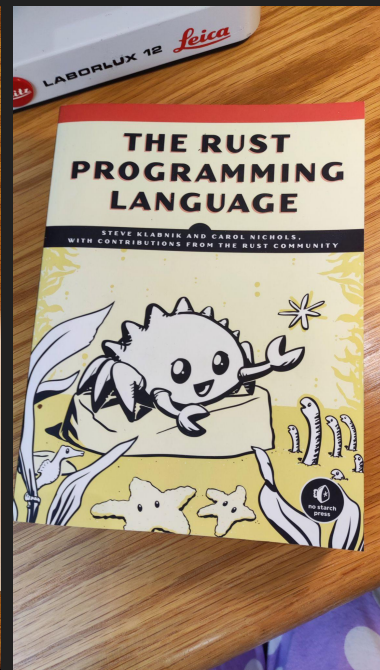
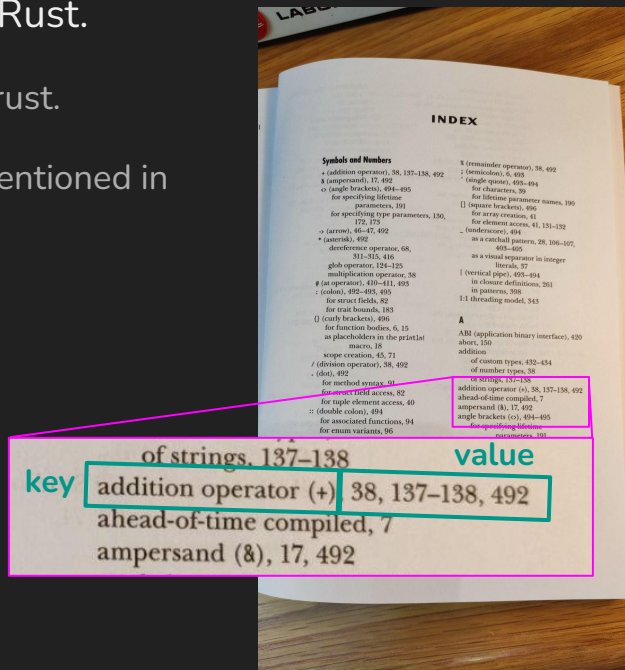
Check the [index](#)

Book Index

Stores phrases & their occurrences

[Key](#): [Value](#) pairs

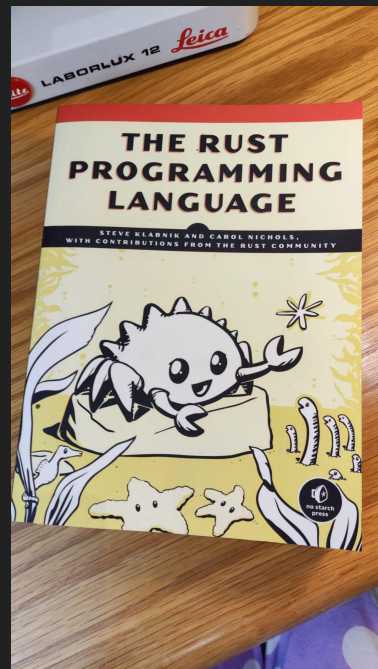
Shortlist of places to look.



A mechanism to speed up access to data

Attribute or set of attributes used to look up records in a file.
e.g. DOI for papers

Records (data) indexed by search keys.
e.g. paper abstracts



Indexing

Structure of our book index (index file)

Keys are words or phrases

Values are list of locations

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
→ "empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
→ "enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for

Complexity?

Best case: $O(1)$ (at the start)

Average case: $O(\frac{1}{2}n) = O(n)$ (in the middle)

Worst case: $O(n)$ (at the end)

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
→ "enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

How do we search the index?

Index is organised alphabetically.

Let's start at the beginning, and look at each key until we find the one we're looking for

Complexity?

Best case: $O(1)$ (at the start)

Average case: $O(\frac{1}{2}n) = O(n)$ (in the middle)

Worst case: $O(n)$ (at the end)

What if there are **thousands** of keys in the index?

What if there are **millions** of keys in the index?

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
→ "enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
→ "encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
→ "encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
→ "encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half

1st bisection



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half

1st bisection



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half

1st bisection



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half

1st bisection



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

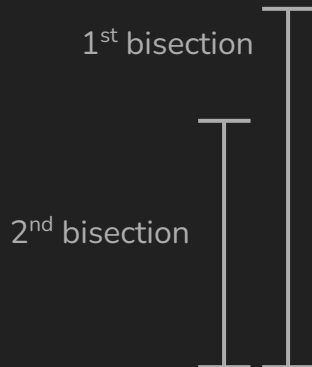
Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

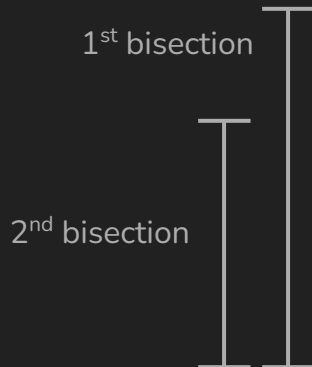
Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Searching more efficiently

Binary search

Go to the middle

If we've gone too far, repeat for first half

If we're not there yet, repeat for second half

Complexity?



$\log_2(n)$ bisections

Best case: $O(1)$ (in the middle)

Average case: $O(\log n)$

Worst case: $O(\log n)$ (final bisection)

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
→ "enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Doing good!

Linear search: $O(n)$

Binary search: $O(\log n)$

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Doing good!

Linear search: $O(n)$

Binary search: $O(\log n)$

Genomic data is really big.

Can we beat $O(\log n)$?

Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

Indexing

Doing good!

Linear search: $O(n)$

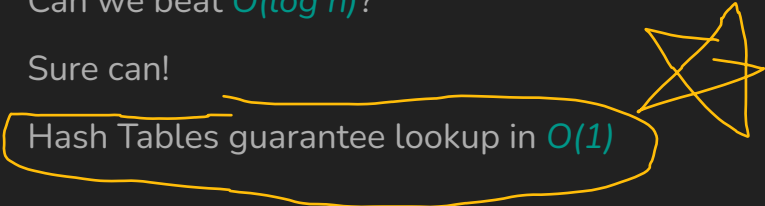
Binary search: $O(\log n)$

Genomic data is really big.

Can we beat $O(\log n)$?

Sure can!

Hash Tables guarantee lookup in $O(1)$



Key	Value
...	...
"else if expression"	[50, 51]
"else keyword"	[49]
"empty type"	[443, 444, 494]
"encapsulation"	[366, 367, 368]
"entry method"	[145, 146, 147]
"entry type"	[145, 146, 147]
"enumerate method"	[74]
"enums"	[95, 96, 97, 98, ...]
...	...

数学上，如果你有 n 个元素，并且每一次迭代你都把搜索范围减少到之前的一半，那么在 k 次迭代之后，搜索范围就会减少到：

$$n/2^k$$

我们想知道需要多少次迭代 k ，才能达到只有一个元素的搜索范围，即 $n/2^k = 1$ 。要解这个方程，可以两边同时取对数。首先我们先解方程：

$$n = 2^k$$

接下来我们两边同时取以2为底的对数：

$$\log_2(n) = \log_2(2^k)$$

使用对数的幂的性质， $\log_b(a^c) = c \log_b(a)$ ，我们可以将右边简化：

$$\log_2(n) = k \cdot \log_2(2)$$

由于 $\log_2(2) = 1$ ，因此这个表达式进一步简化为：

$$\log_2(n) = k$$

Hash Tables

Hash Tables

Hash Tables

Instead of linear / binary search, want a direct mapping between key and location data is stored.

Data structure for indexing

Allow us to instantly* go to the right location

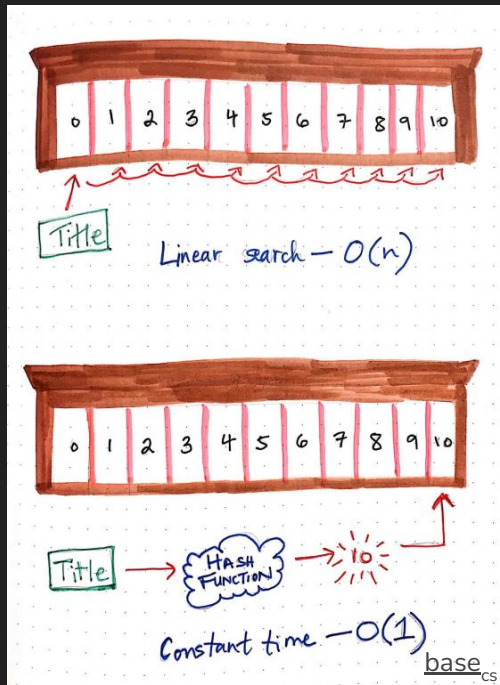
Composed of two parts

1. Array / Table

Where the data is stored
(our book index in previous examples)

2. Mapping (hash) function

Maps input data (key) to a specific location in the table



Hash Tables

Hash Function

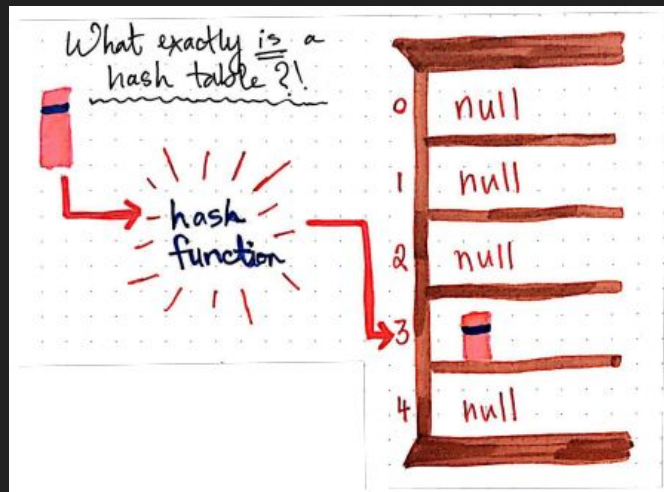
Instead of organising items in index file alphabetically...

Perform some mathematical function on the key to determine its place.

Result is the location (bucket) we place the data for this key.

Works for adding items to the table, and retrieving data for items in the table.

If the hash function is efficient we can access data in constant time: $O(1)$



Hash Tables

Indexing book titles (example)

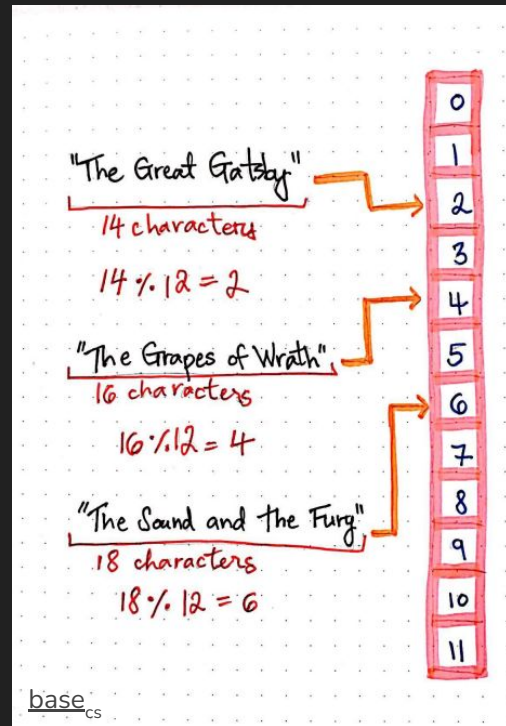
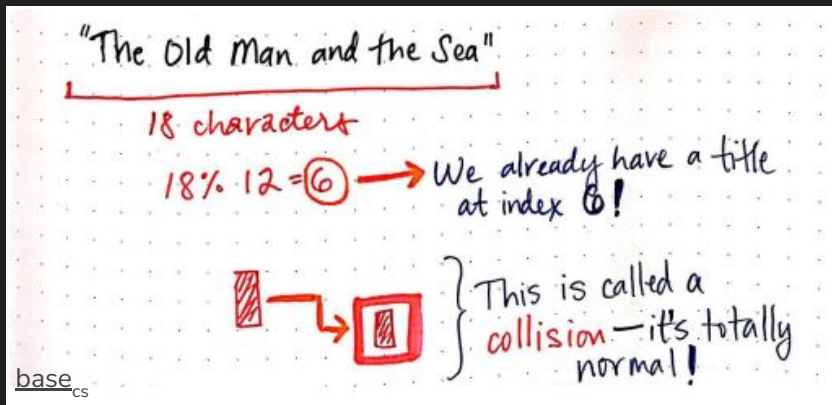
Hash table of size 12

Use modulo function as our hash function.

Take the number of characters in a string

Divide that by the size of the table

Get the remainder of the division as **bucket**



Hash Tables

A good hash function should

Be fast and easy to compute
(otherwise no advantage over sorted indexes)

Map to a finite output value ($\text{RAM} < \infty\text{GB}$)

Produce a good distribution of values over the table
(space efficiency)

Avoid too many collisions
(deteriorate performance)

Deterministic!
(otherwise can't reliably access/retrieve data)

Collisions

Occur when 2 or more keys produce same output from hash function.

Not the end of the world!

Most hash table implementations will guarantee that collisions are rare.

Resolving collisions: closed hashing

Closed hashing

Data stored in the table/array itself

Collision resolution

Linear probing

Search the table until you find the next available hash bucket/index

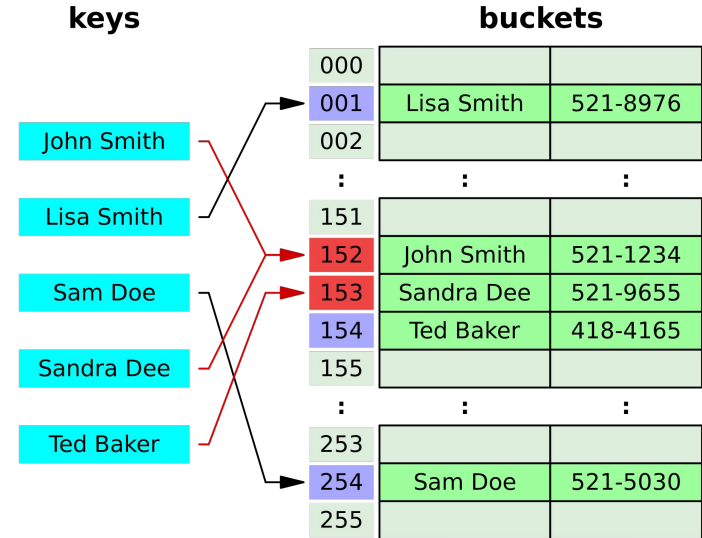
Drawback: tendency of clustering/poor distribution

Double hashing

Uses another hash function to treat collisions

Calculate incremental offsets

Linear probing



Resolving collisions: open hashing

Open hashing



Data stored in a linked list

Collision resolution

Separate chaining

Each index/bucket is a list

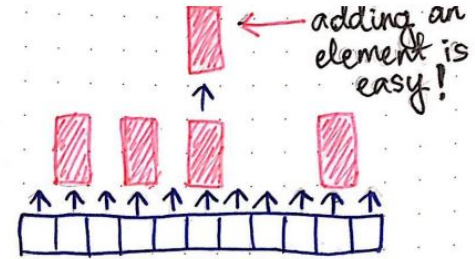
If there's a collision, add item to the end of the list

Drawback: more time to search if multiple items in a bucket

If we have a good hash function, collisions will be rare.

Remember: keys are stored with data, so know which item in the linked list to retrieve!

Separate Chaining



instead of storing
one item, we can store
an entire linked list.

Applying to Genomic Data

Applying to Genomic Data

所谓的“读取”指的是通过测序技术获得的DNA序列。把基因组想象成一个由许多这样的序列组成的长字符串。通过在哈希表中创建基因组的索引，每一个独特的序列或“读取”都可以与其在基因组中的位置相关联。因此，当研究人员想要找到特定序列的位置时，他们可以简单地使用哈希表立即查找到它，这就是所说的“ $O(1)$ 时间”。

Hash Tables can help us locate sequences

Instead of looking at **all possible locations** a DNA sequence might be in a genome...

Create an **index** of the genome

Then **lookup** the location of our sequence

Keep in mind

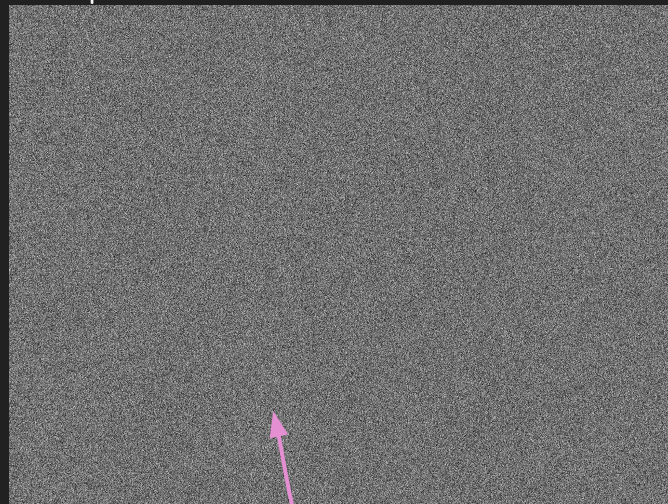
Genomes are massive

If we can look up the location of a read in $O(1)$ time, that's amazing!

Much better than trillions of operations.



Template



7.6M pixels
(bacterial
genome)

Pattern



40k pixels
(single long read)



Applying to Genomic Data

For this example

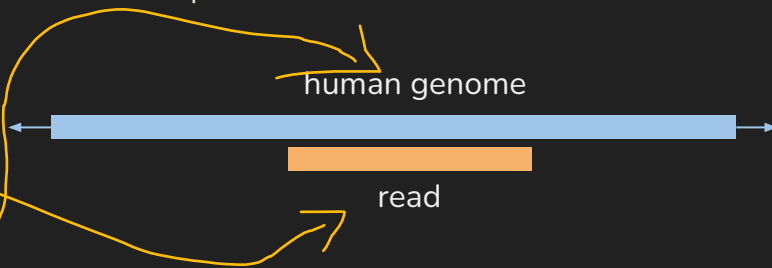
Pattern: DNA sequencing **reads** (~100bp)

base pair

Template: The **human reference genome**

"Pattern"通常指的是我们试图在基因组数据中找到的一个特定的DNA序列

"Template"在这里指的是整个基因组，它是参照物或背景，我们要在其中找到"pattern"的位置。



Applying to Genomic Data

For this example

Pattern: DNA sequencing **reads** (~100bp)

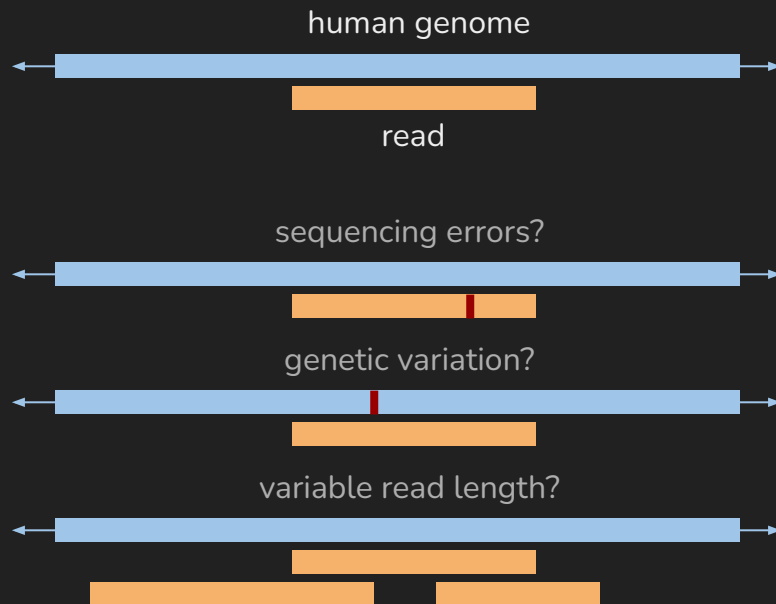
Template: The **human reference genome**

Considerations

Errors in DNA sequencing

Genetic variation between individual and reference human genome

Variable read length (± 10 bp)



k-mers

Applying to Genomic Data

Using k-mers

To avoid these issues, let's split up our reads & the genome into smaller pieces

K-mer: subsequence of length K

Choose a value for k : 3bp (3-mer)

Result

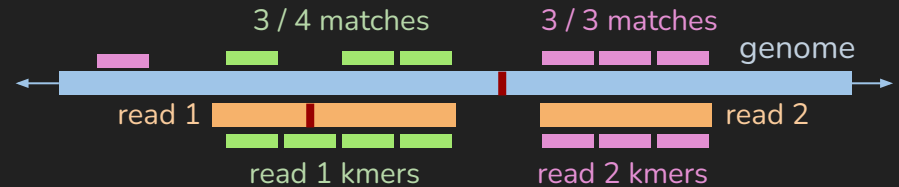
If there are disagreements between the genome and a read, most of the k-mers will still match

Caveat

Smaller pieces = Off-target k-mer matches

Correct location = Most k-mer matches

More on kmers in the following weeks



Applying to Genomic Data

Indexing the Genome

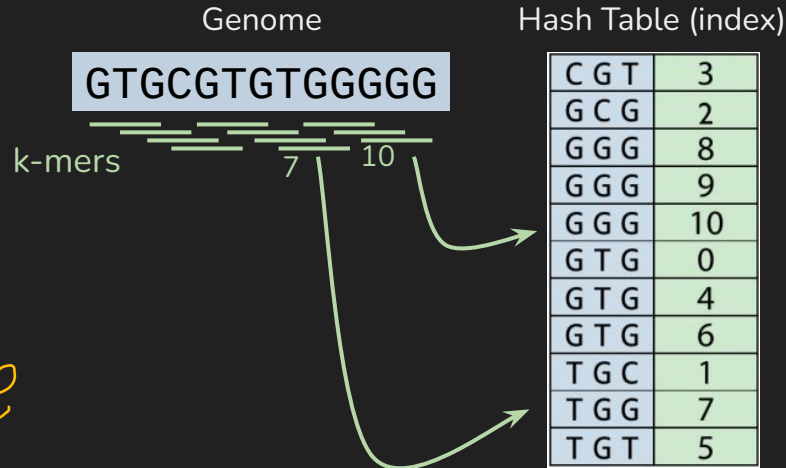
We have selected $k=3$

Scan across genome, **extract** each k-mer

Store k-mers as **keys**, locations as **values**

Only build the index once

in Genome



Applying to Genomic Data

Indexing the Genome

We have selected $k=3$

Scan across genome, **extract** each k-mer

Store k-mers as **keys**, locations as **values**

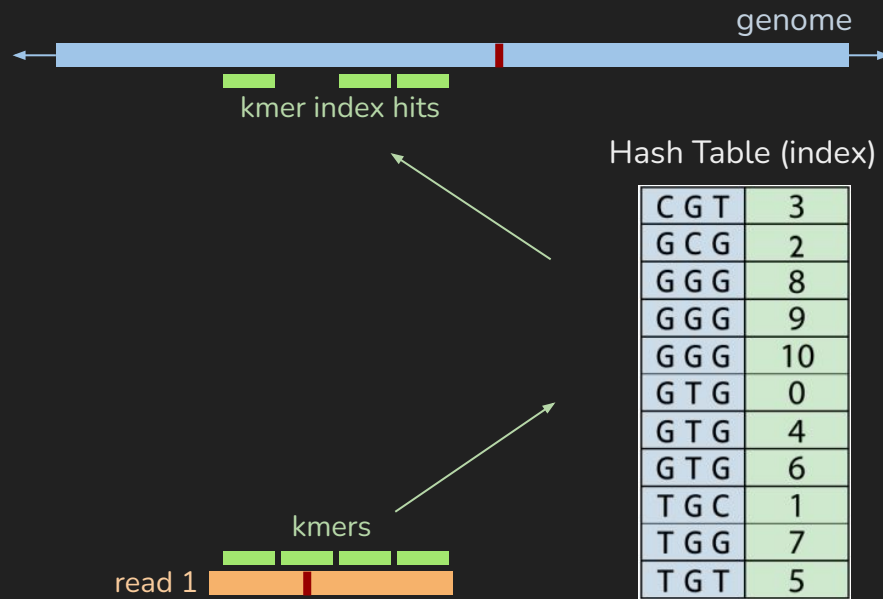
Only build the index once

Finding read locations

Extract k-mers from our read

Look up locations in the index

Use matches to shortlist locations to look



Applying to Genomic Data

Hash tables in Python

In Python, a hash table is also known as a dictionary or map

Multimap: when there are multiple values stored per key

A dict() has keys which look up values in $O(1)$ time

```
>>> t = 'GTGCGTGTGGGGG'
>>> 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]
```

Applying to Genomic Data



If we want to index k-mers for a genome of length M...

$O(M)$ time to build

$O(M)$ space requirements

$O(1)$ lookup time (typical case)

A good hash function will

Generate well distributed indexes
(avoid collisions)

Be quick & deterministic

Caveats

We can only look up exact keys of length k

What about lots of mismatches / variation?

What about off-target matches?

More on these in coming weeks.

Thank you!

Don't forget your signed academic integrity statement

Background survey on Poll Everywhere.

polllev.com/gracehall381

Today: Indexing

Next time: Sequencing Alignment

