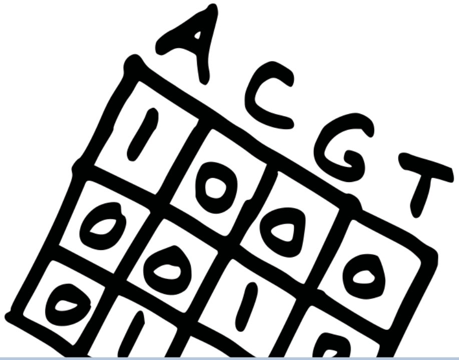
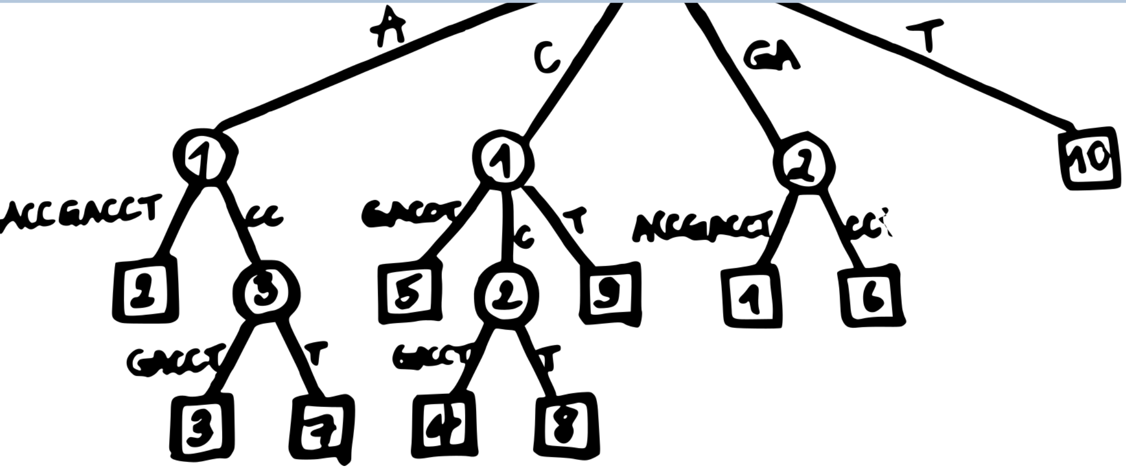
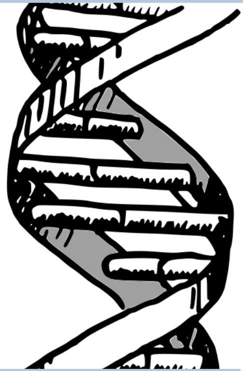


Sequence algorithms

DAG workshop, 2020
Cyril Matthey-Doret

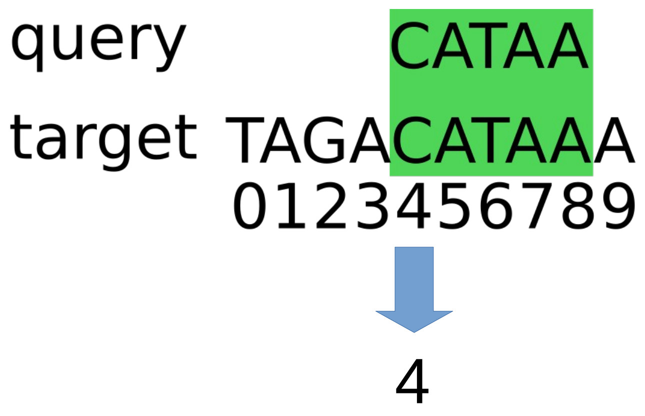



cyril.mattheydoret@gmail.com
cmdoret

Exercises from session 1

Exact sequence matching

Given two sequences (DNA, RNA, proteins, …)
At what position of target does query occur ?

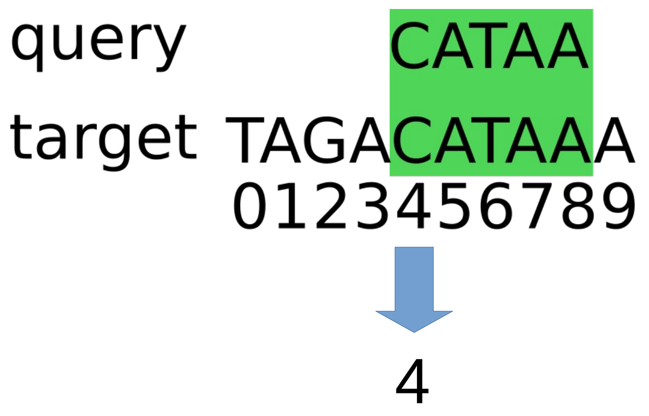


When is this useful ?

- RNA quantification
- Read overlapping in genome assembly
- Primer design

Exact sequence matching

Given two sequences (DNA, RNA, proteins, …)
At what position of target does query occur ?

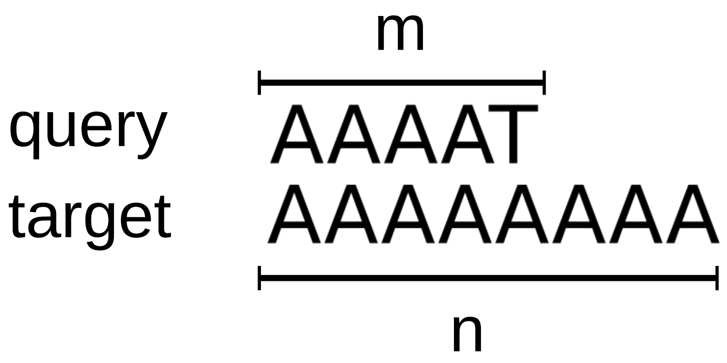


```
Pythonic way:
target = “TAGACATAAAAGA”
query = “CATAA”
target.find(query)
4
```

Can we come up with a naive algorithm ?

Naive solution

- How many possible alignments ?
- Number of comparisons ? (worst case)



Naive solution

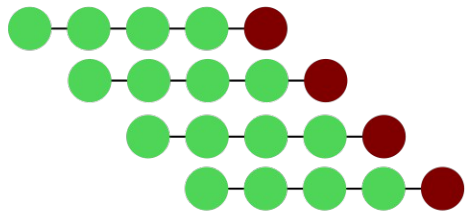
- How many possible alignments ? → n - m + 1
- Number of comparisons ? (worst case)

```
          AAAAT
         AAAAT
        AAAAT
query  AAAAT
target AAAAAAAA
```

Naive solution

- How many possible alignments ? → n - m + 1
- Number of comparisons ? (worst case) → m(n - m + 1)

```
         AAAAT
        AAAAT
       AAAAT
query AAAAT
target AAAAAAAA
```

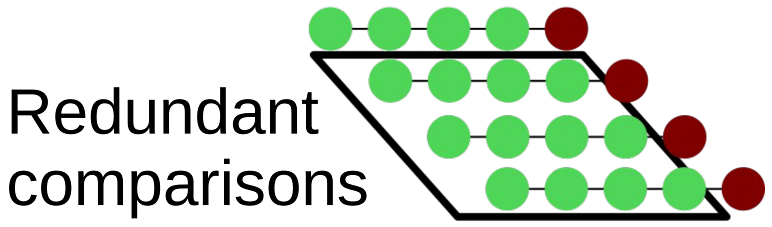


Big O notation ?

Naive solution

- How many possible alignments ? → n - m + 1
- Number of comparisons ? (worst case) → m(n - m + 1)

```
             AAAAT
           AAAAT
         AAAAT
query  AAAAT
target AAAAAAAA
```

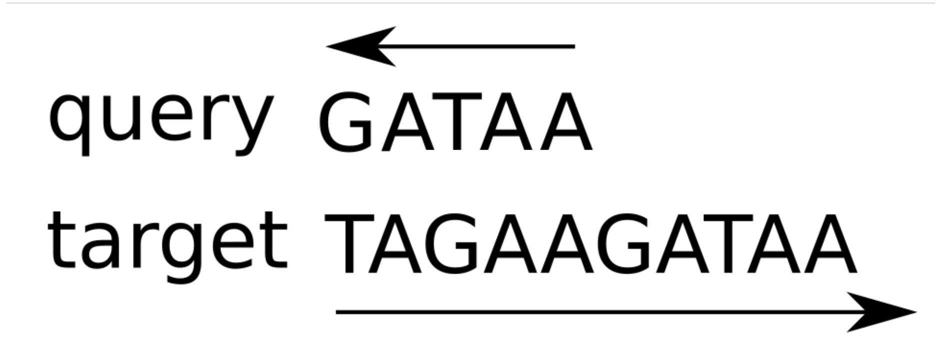


Big O notation ?
O(mn)

How to get faster ?

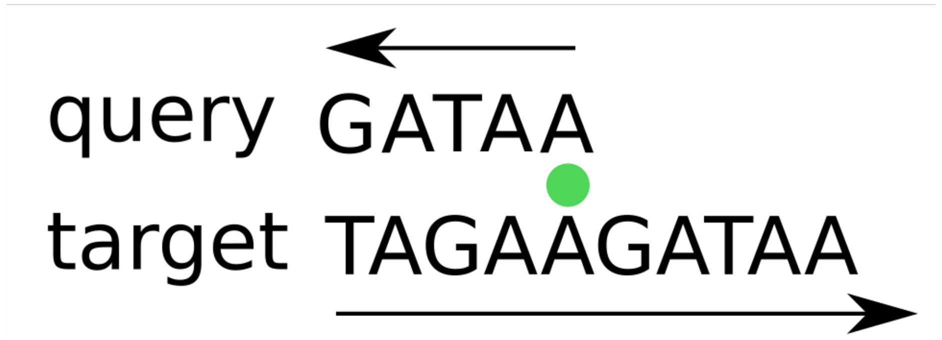
The bad character rule

- Skip multiple positions at once
- One of the optimisations used in the Boyer-Moore algorithm



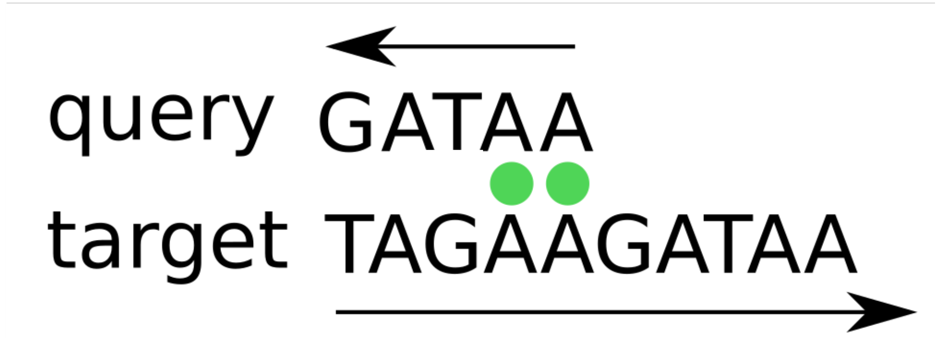
The bad character rule

- Skip multiple positions at once
- One of the optimisations used in the Boyer-Moore algorithm



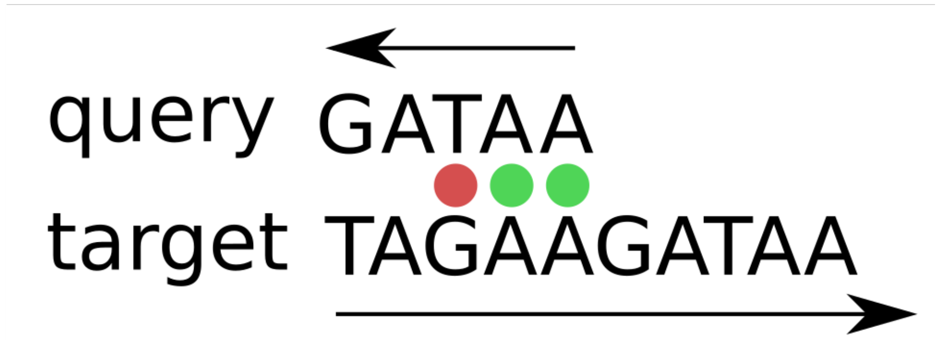
The bad character rule

- Skip multiple positions at once
- One of the optimisations used in the Boyer-Moore algorithm



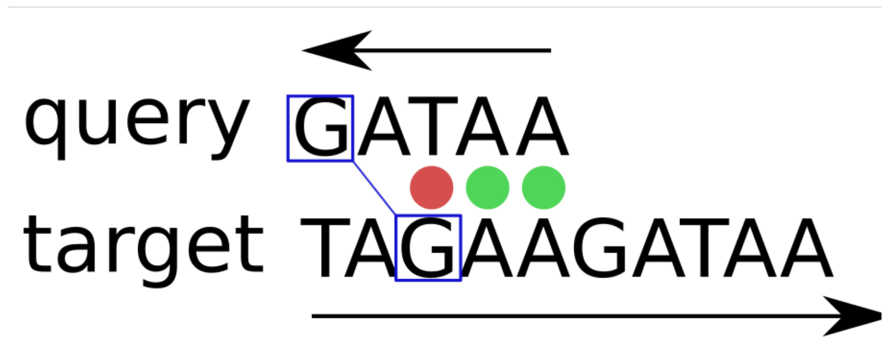
The bad character rule

- Skip multiple positions at once
- One of the optimisations used in the Boyer-Moore algorithm



The bad character rule

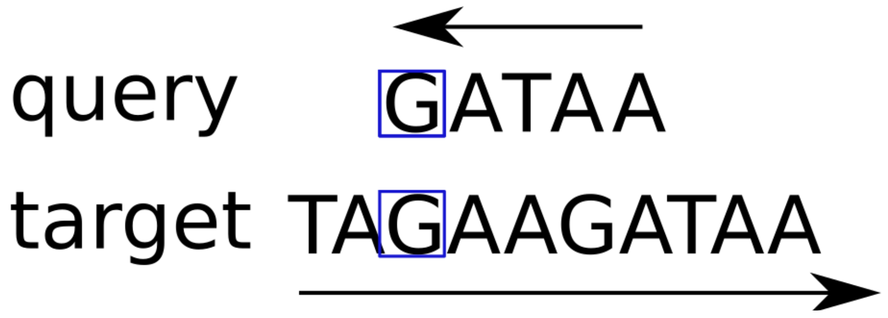
- Skip multiple positions at once
- One of the optimisations used in the Boyer-Moore algorithm



Upon mismatch, skip until next matching character

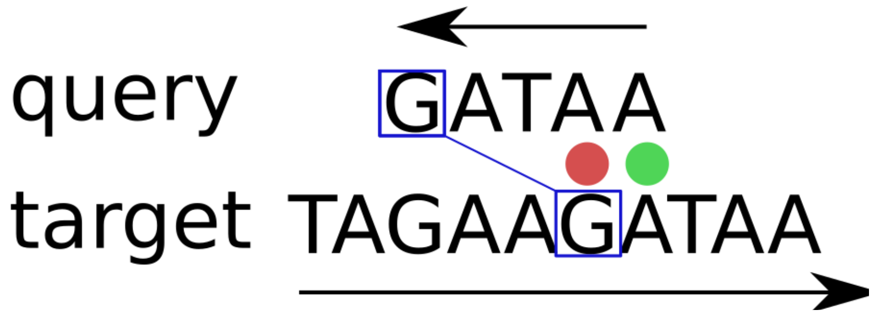
The bad character rule

- Skip multiple positions at once
- One of the optimisations used in the Boyer-Moore algorithm



Preprocessing jumps

- Pre-compute jump sizes using a hash table !
- Scanning the query for the next match would take O(m)
- Looking up [“A-G”] in the hash table takes O(1)

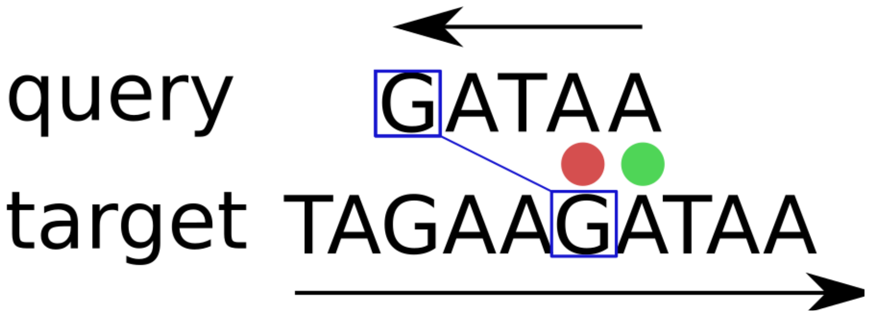


Query

Alphabet	G	A	T	A	A
A	0	-	0	-	-
C	0	1	2	3	4
T	0	1	-	0	1
G	-	0	1	$\boxed{3}$	3

Preprocessing jumps

- Pre-compute jump sizes using a hash table !
- Scanning the query for the next match would take O(m)
- Looking up [“A-G”] in the hash table takes O(1)



Query

Alphabet

	G	A	T	A	A
A	0	-	0	-	-
C	0	1	2	3	4
T	0	1	-	0	1
G	-	0	1	3	3

This and other optimisations, lead to the Boyer-Moore algorithm, which is O(m+n)

Indexing

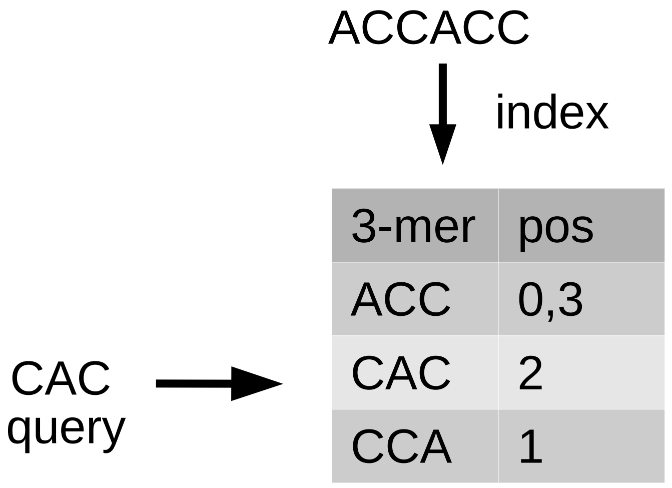
- Pre-processing the input sequence allows faster queries
- Different data structures can serve as an index
  - Hash table
  - suffix array
  - suffix tree
  - ...

When is this useful ?

- Read alignment
- Random access of large files

Indexing

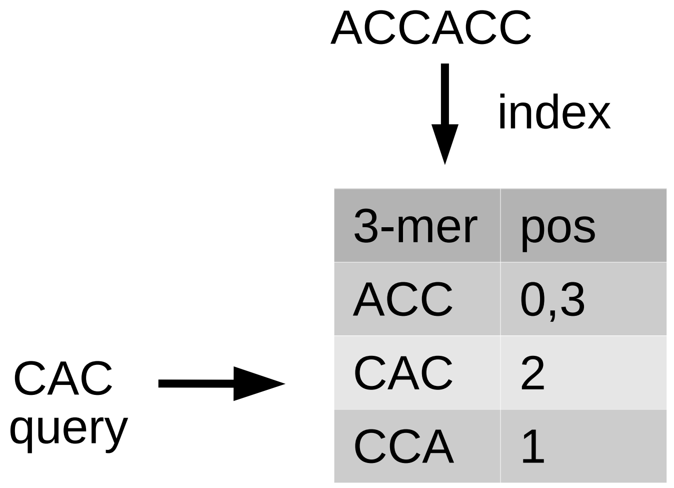
- Pre-processing the input sequence allows faster queries
- Different data structures can serve as an index
  - Hash table
  - suffix array
  - suffix tree
  - ...



3-mer	pos
ACC	0,3
CAC	2
CCA	1

Indexing

- With a fix query size, we can use a k-mer based dictionary
- But what if the query size varies ?



3-mer	pos
ACC	0,3
CAC	2
CCA	1

Indexing

- With a fix query size, we can use a k-mer based dictionary
- But what if the query size varies ?

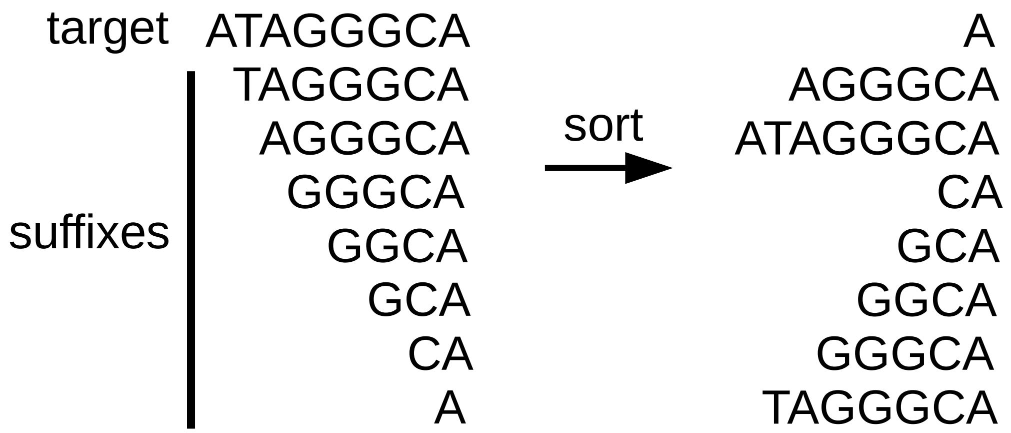
  → Index all suffixes instead of k-mers

```
target    ATAGGGCA
        |  TAGGGCA
        |   AGGGCA
        |    GGGCA
suffixes|     GGCA
        |      GCA
        |       CA
        |        A
```

Indexing

- With a fix query size, we can use a k-mer based dictionary
- But what if the query size varies ?

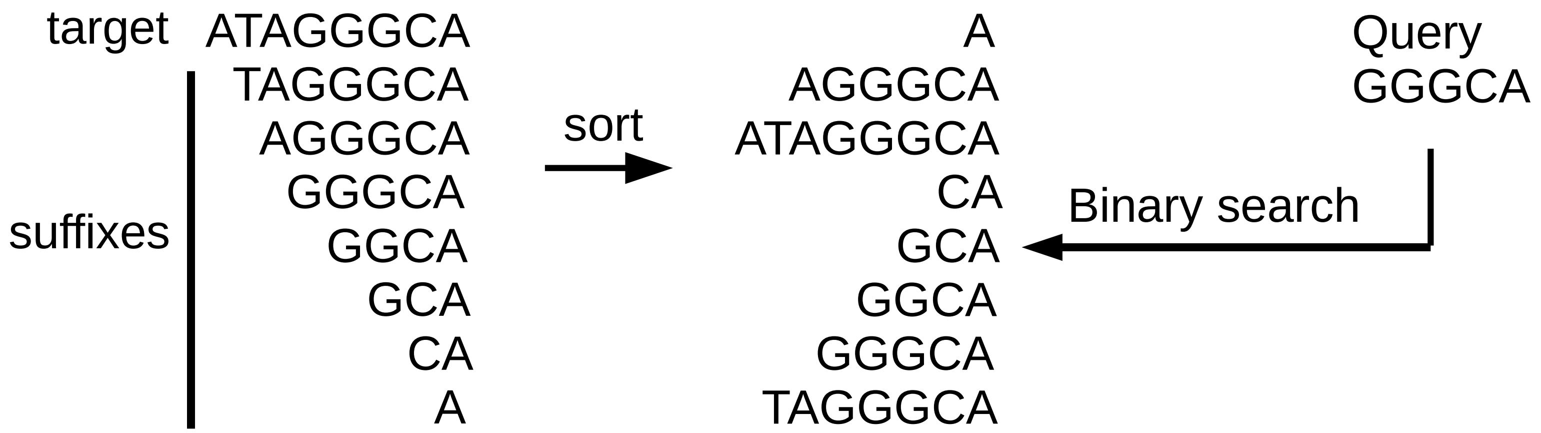
  → Index all suffixes instead of k-mers



Indexing

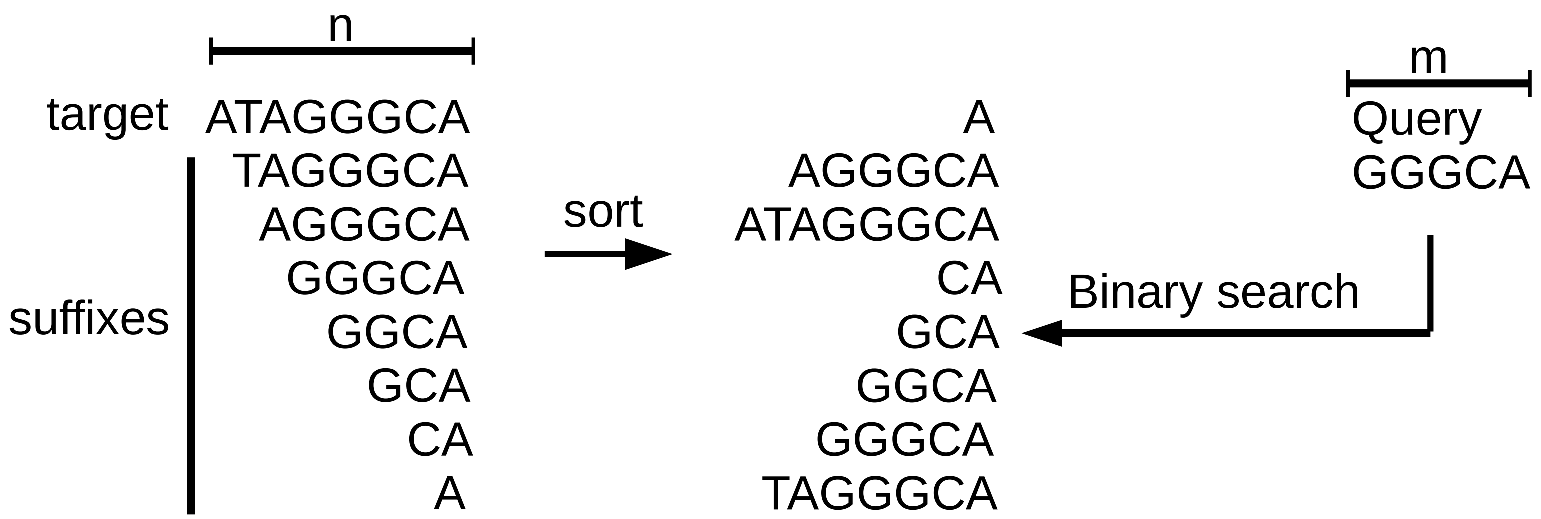
- With a fix query size, we can use a k-mer based dictionary
- But what if the query size varies ?

  → Index all suffixes instead of k-mers



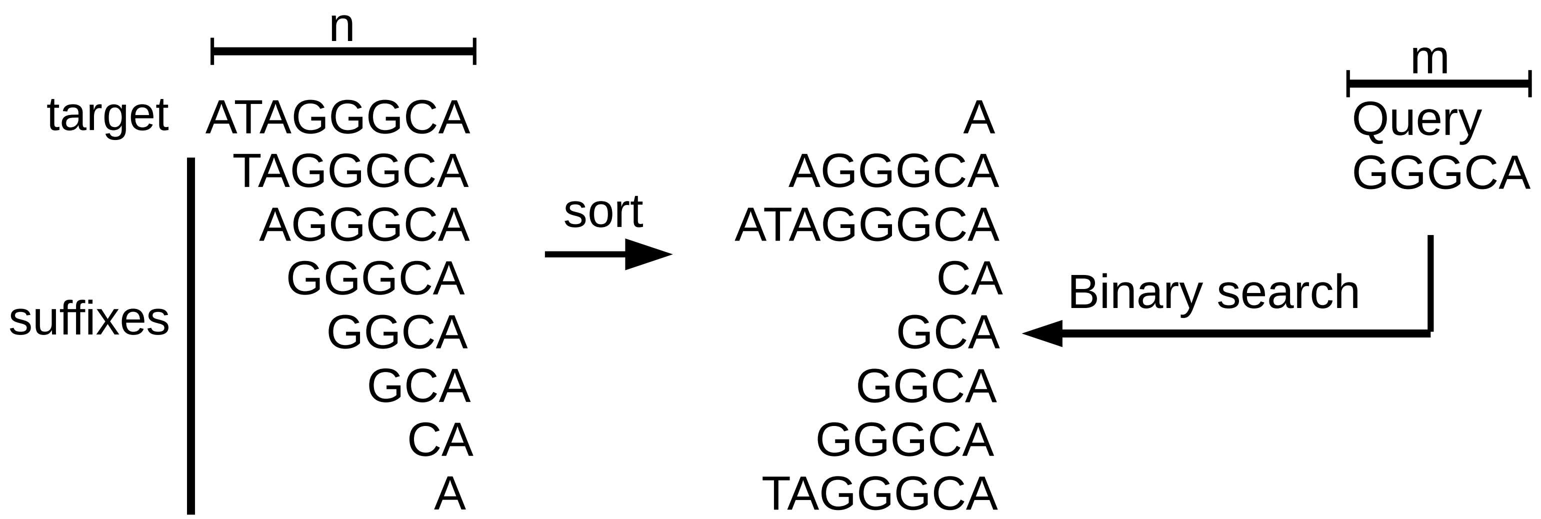
Space analysis

- This is fast ! What is the query time complexity ?



Space analysis

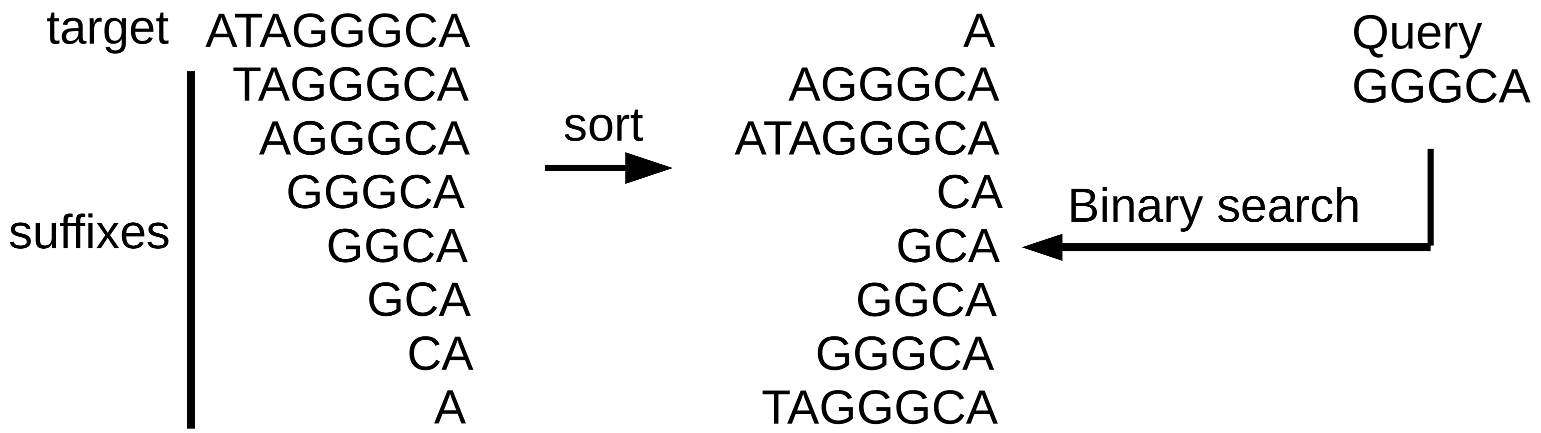
- This is fast ! What is the query time complexity ? O(m log n)



Space analysis

- This is fast ! What is the query time complexity ? O(m log n)
- But… It's huge ! How to estimate memory requirement ?

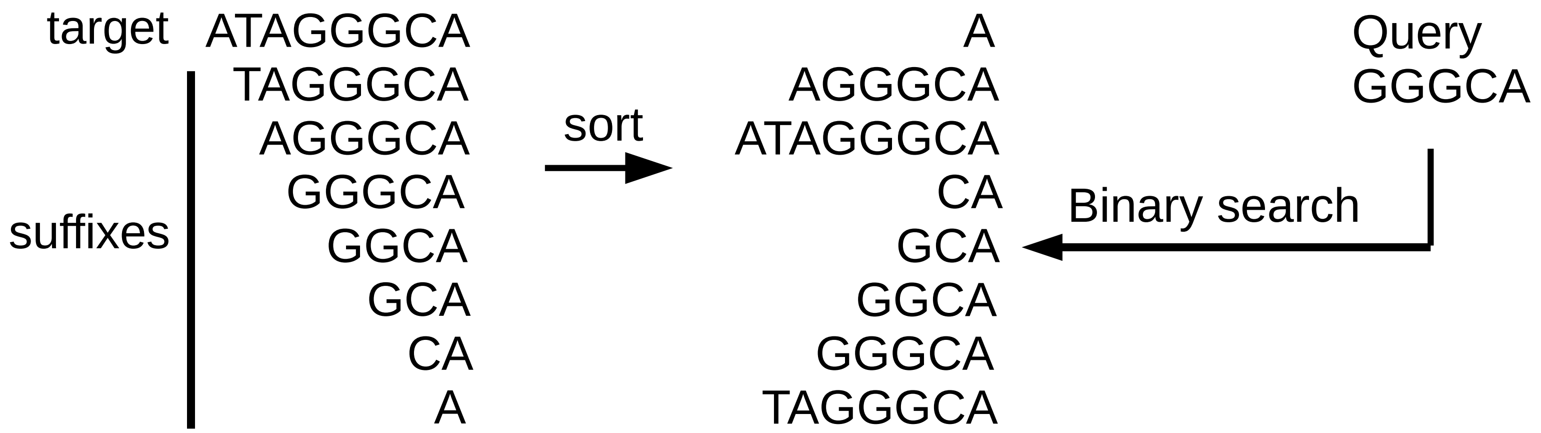
  → Big O notation



Space analysis

- This is fast ! What is the query time complexity ? O(m log n)
- But… It's huge ! How to estimate memory requirement ?

→ Big O notation: $n + (n - 1) + (n - 2) \dots + 1 = \frac{n(n+1)}{2} = O(n^2)$



Space analysis

Any ideas how to remove the redundant parts ?

```
      A
 AGGGCA
ATAGGGCA
     CA
    GCA
   GGCA
  GGGCA
 TAGGGCA
```

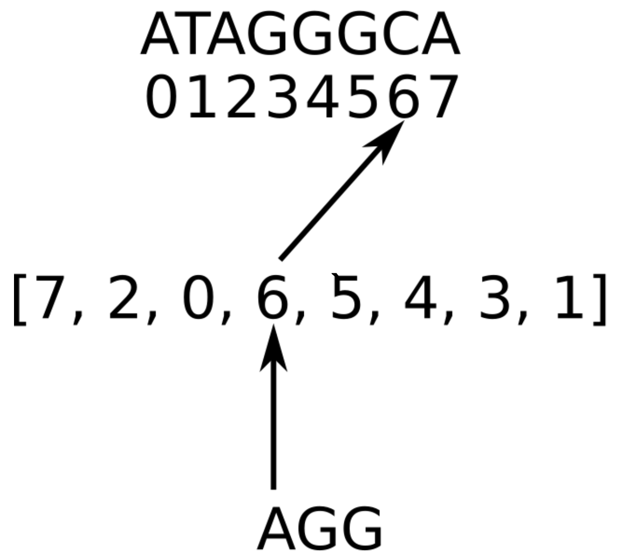
Space analysis

Any ideas how to remove the redundant parts ?
→ Only store the offsets

7	A
2	AGGGCA
0	ATAGGGCA
6	CA
5	GCA
4	GGCA
3	GGGCA
1	TAGGGCA

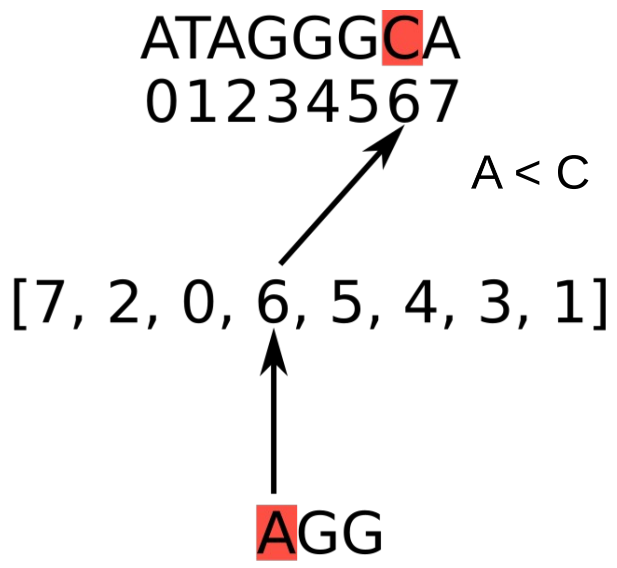
Suffix array

- We can use binary search to look up the suffix (offset) array



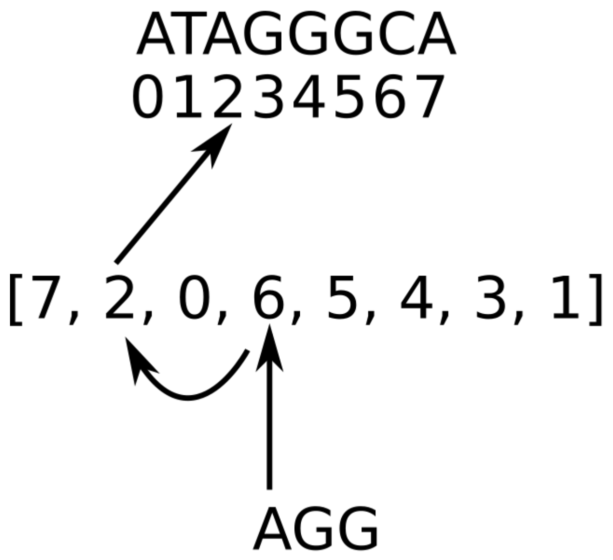
Suffix array

- We can use binary search to look up the suffix (offset) array



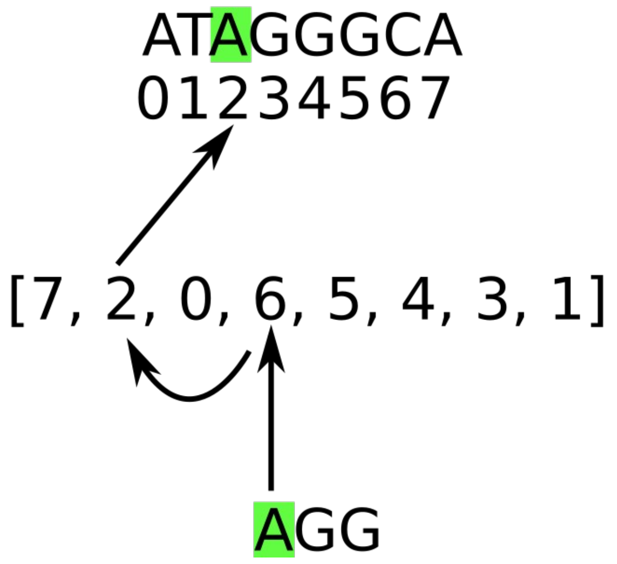
Suffix array

- We can use binary search to look up the suffix (offset) array



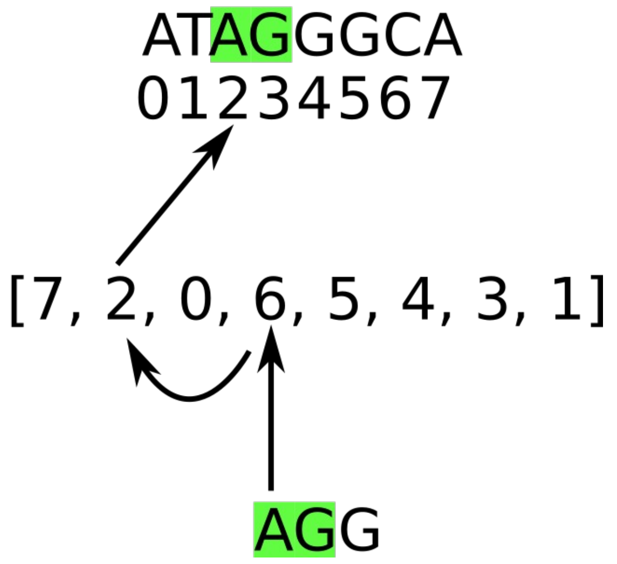
Suffix array

- We can use binary search to look up the suffix (offset) array



Suffix array

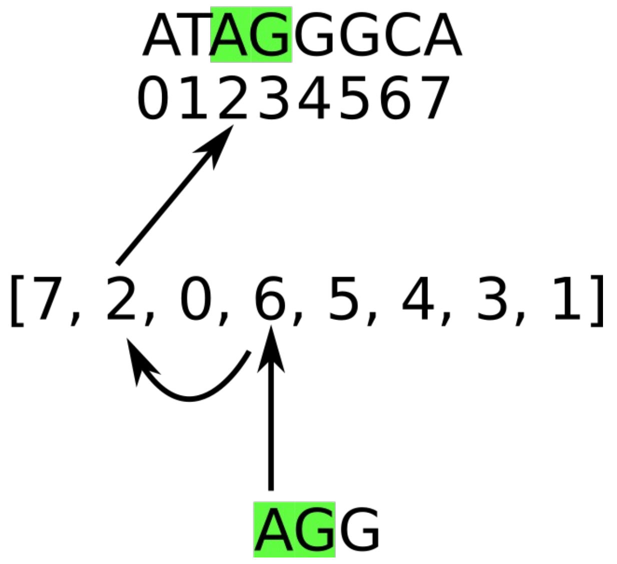
- We can use binary search to look up the suffix (offset) array



What is the search time for a pattern of length m ?

Suffix array

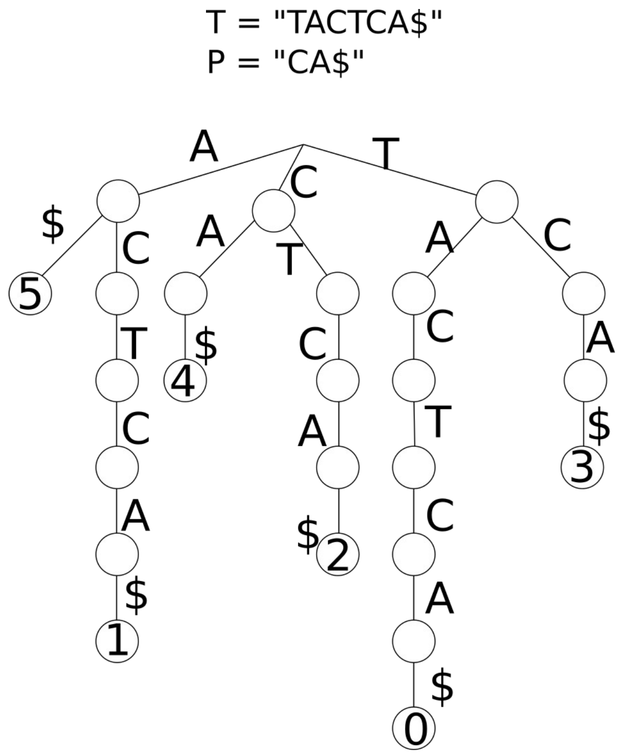
- We can use binary search to look up the suffix (offset) array



What is the search time for a pattern of length m ? O(m log n)

Suffix trees

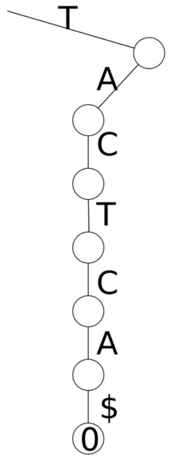
- We can search in O(m) using a suffix tree
- Construction time is non-trivial (O(n))



Suffix trees

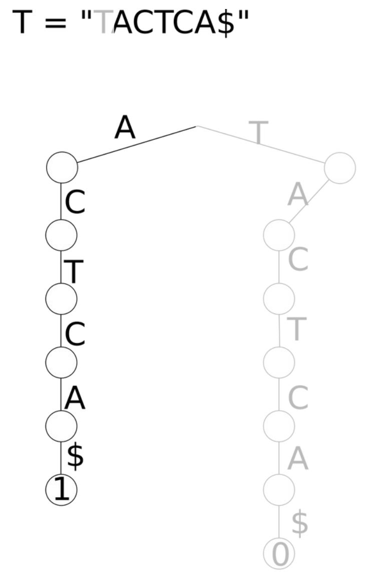
- We can search in O(m) using a suffix tree
- Construction time is non-trivial (O(n))

T = "TACTCA$"



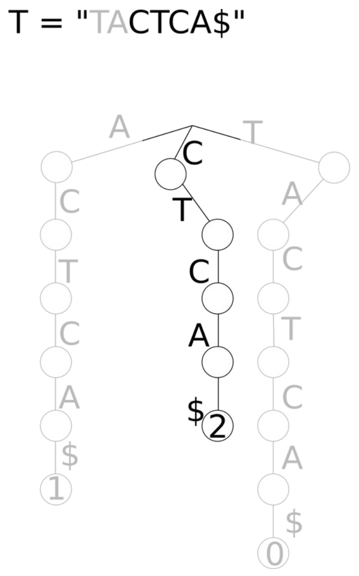
Suffix trees

- We can search in O(m) using a suffix tree
- Construction time is non-trivial (O(n))



Suffix trees

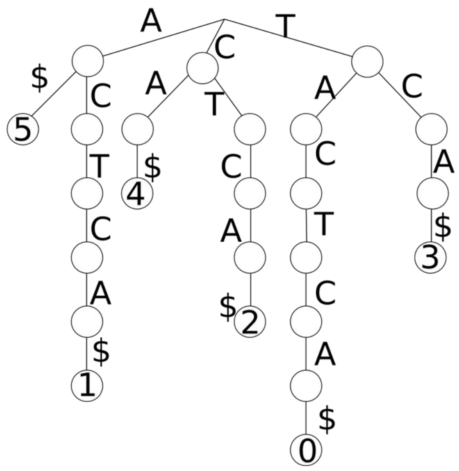
- We can search in O(m) using a suffix tree
- Construction time is non-trivial (O(n))



Suffix trees

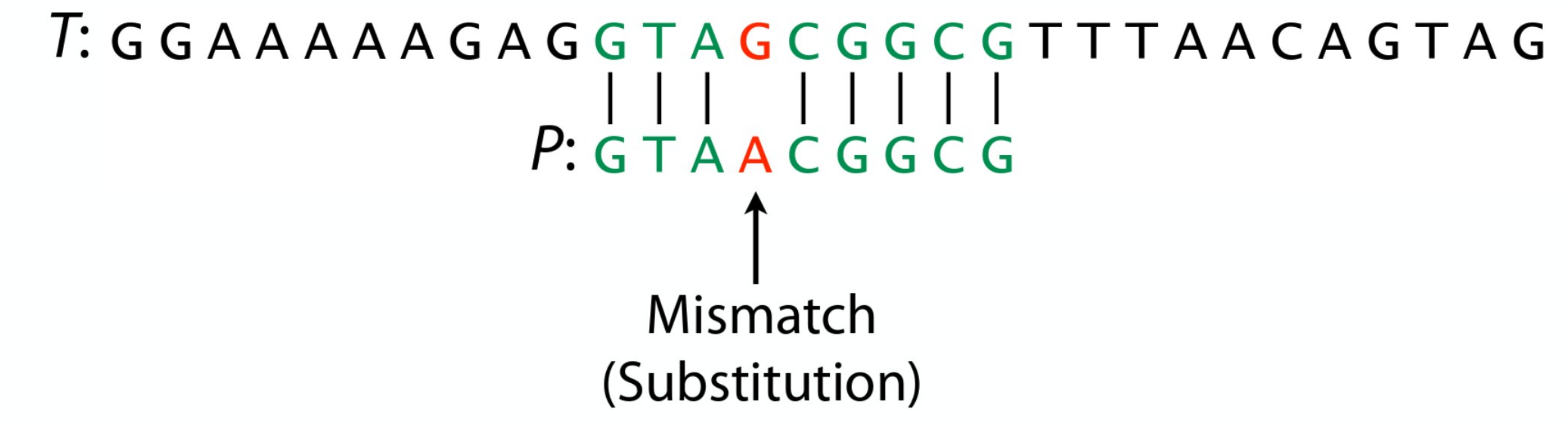
- We can search in O(m) using a suffix tree
- Construction time is non-trivial (O(n))

T = "TACTCA$"



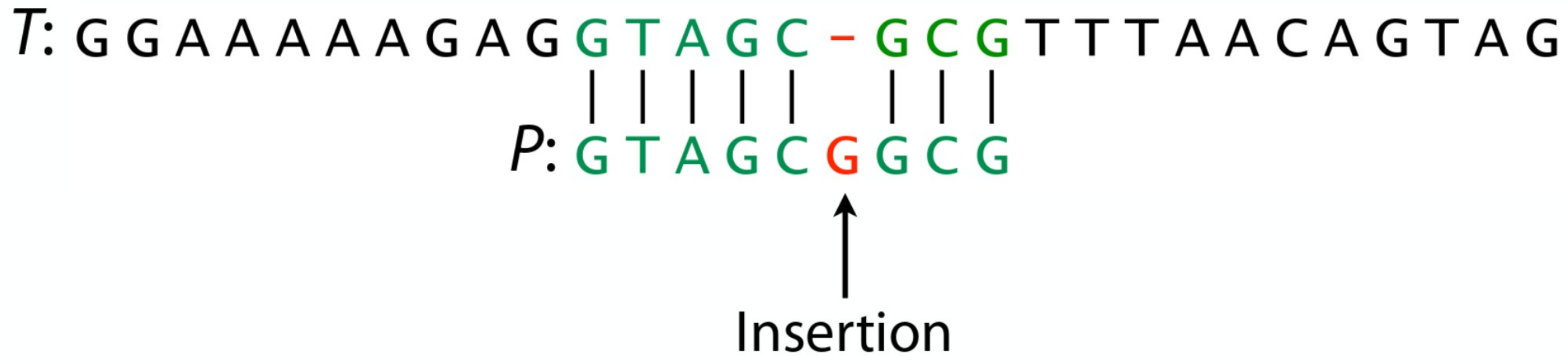
Inexact alignments

- We often want to allow *some* errors in alignments !
  - Sequencing errors in reads
  - Measure strain / species divergence
  - Find homologous genes


T: G G A A A A A G A G G T A G C G G C G T T T A A C A G T A G
P: G T A A C G G C G
Mismatch
(Substitution)

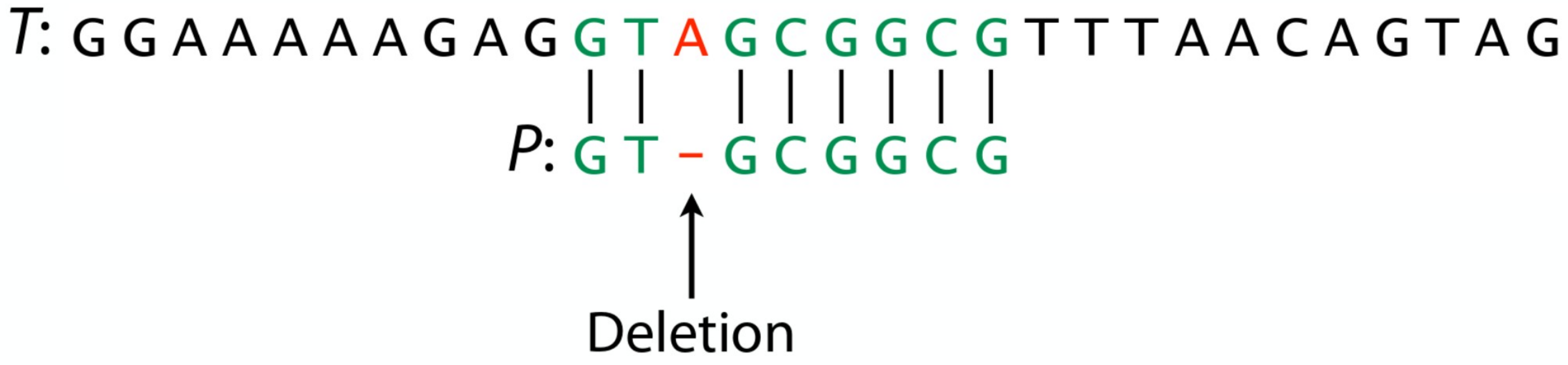
Inexact alignments

- We often want to allow *some* errors in alignments !
  - Sequencing errors in reads
  - Measure strain / species divergence
  - Find homologous genes


T: G G A A A A A A G A G G T A G C – G C G T T T A A C A G T A G
P: G T A G C G G C G
Insertion

Inexact alignments

- We often want to allow *some* errors in alignments !
  - Sequencing errors in reads
  - Measure strain / species divergence
  - Find homologous genes


T: G G A A A A A A G A G G T A G C G G C G T T T A A C A G T A G
P: G T – G C G G C G
Deletion

Quantifying errors

- Hamming distance: # of substitutions to turn P into T
- Edit distance: Same, but allows insertions / deletions

T = TTGCC
P = CTCGC

↓ Substitute C → T

TTGCC
TTCGC

↓ Substitute C → G

TTGCC
TTGGC

Hamming distance = 2

What if P ant T have different lengths ?

Inexact alignments

- Aligned sequence will almost always have different lengths
- This can be solved with 2 approaches

$$T = \overbrace{TTGCCC}^{n}$$
$$P = \underbrace{TGCG}_{m}$$

Global alignment

Naive: compute all possible alignments

```
TTGCCC      TTGCCC      TTGCCC
TGCG__      TGC_G_      TGC__G

TTGCCC      TTGCCC      ...
TC_C_G      _TGC_G
```

Local alignment

Search for substring of length m in T with smallest edit distance

```
TTGCCC
 TGCG
```

Inexact alignments

- Aligned sequence will almost always have different lengths
- This can be solved with 2 approaches

$$T = \overbrace{TTGCCC}^{n}$$
$$P = \underbrace{TGCG}_{m}$$

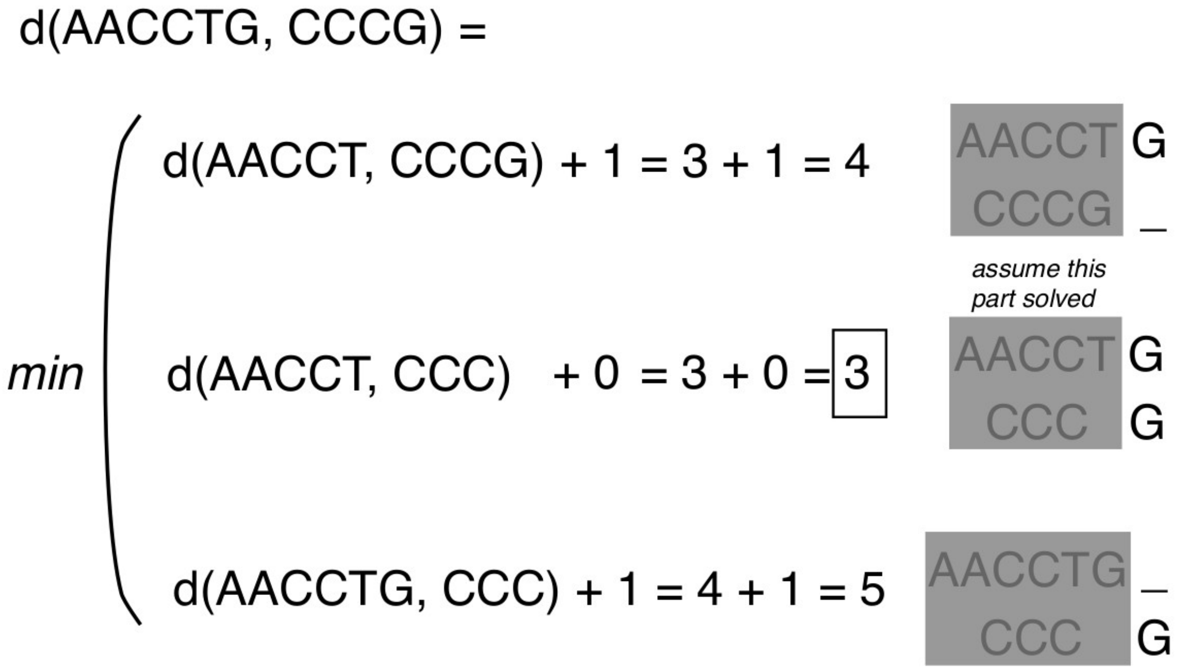
Global alignment

Local alignment

Edit distance: Too many possibilities !

Inexact alignments

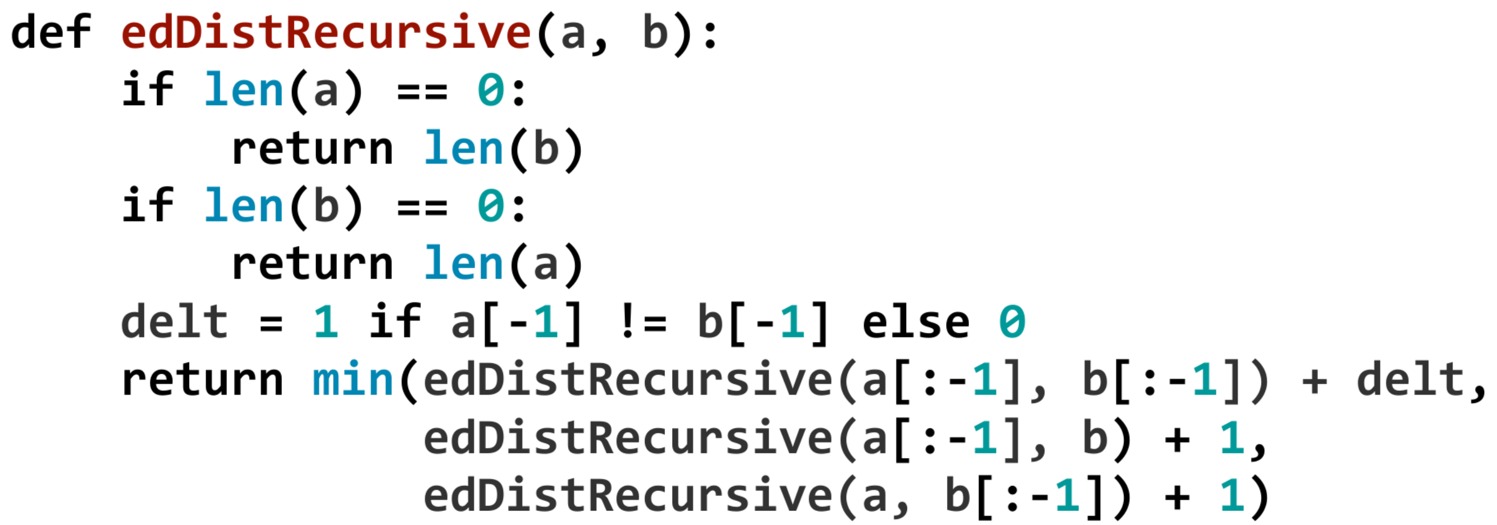
- Can be formulated as a recursive problem
- Large exploration space



$$d(\text{AACCTG, CCCG}) = \min \begin{cases} d(\text{AACCT, CCCG}) + 1 = 3 + 1 = 4 \\ d(\text{AACCT, CCC}) + 0 = 3 + 0 = \boxed{3} \\ d(\text{AACCTG, CCC}) + 1 = 4 + 1 = 5 \end{cases}$$

Inexact alignments

- Can be formulated as a recursive problem
- How to reduce the exploration space ?



```python
def edDistRecursive(a, b):
    if len(a) == 0:
        return len(b)
    if len(b) == 0:
        return len(a)
    delt = 1 if a[-1] != b[-1] else 0
    return min(edDistRecursive(a[:-1], b[:-1]) + delt,
               edDistRecursive(a[:-1], b) + 1,
               edDistRecursive(a, b[:-1]) + 1)
```

Image source: https://github.com/BenLangmead/ads1-slides/

Inexact alignment in feasible time

- Can be formulated as a recursive problem
- How to reduce the exploration space ?

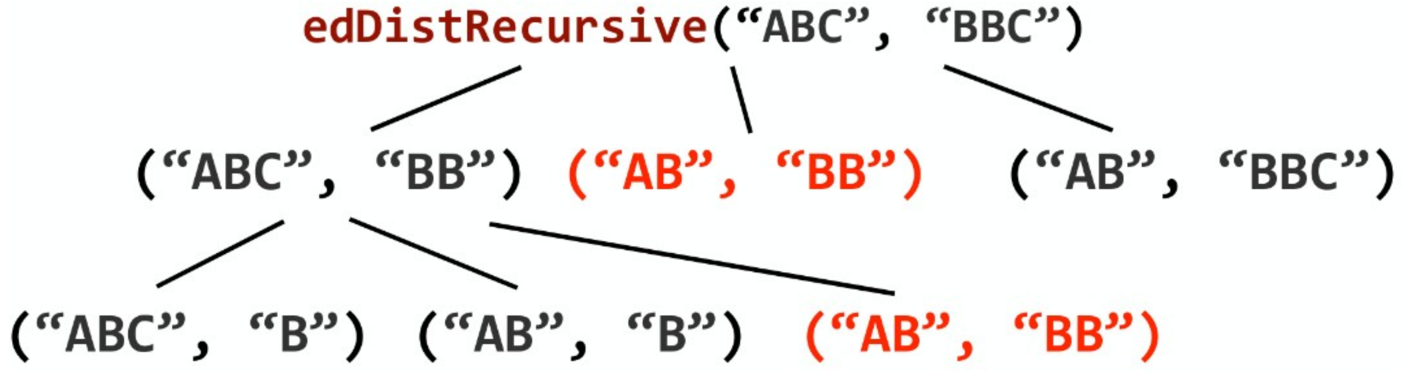
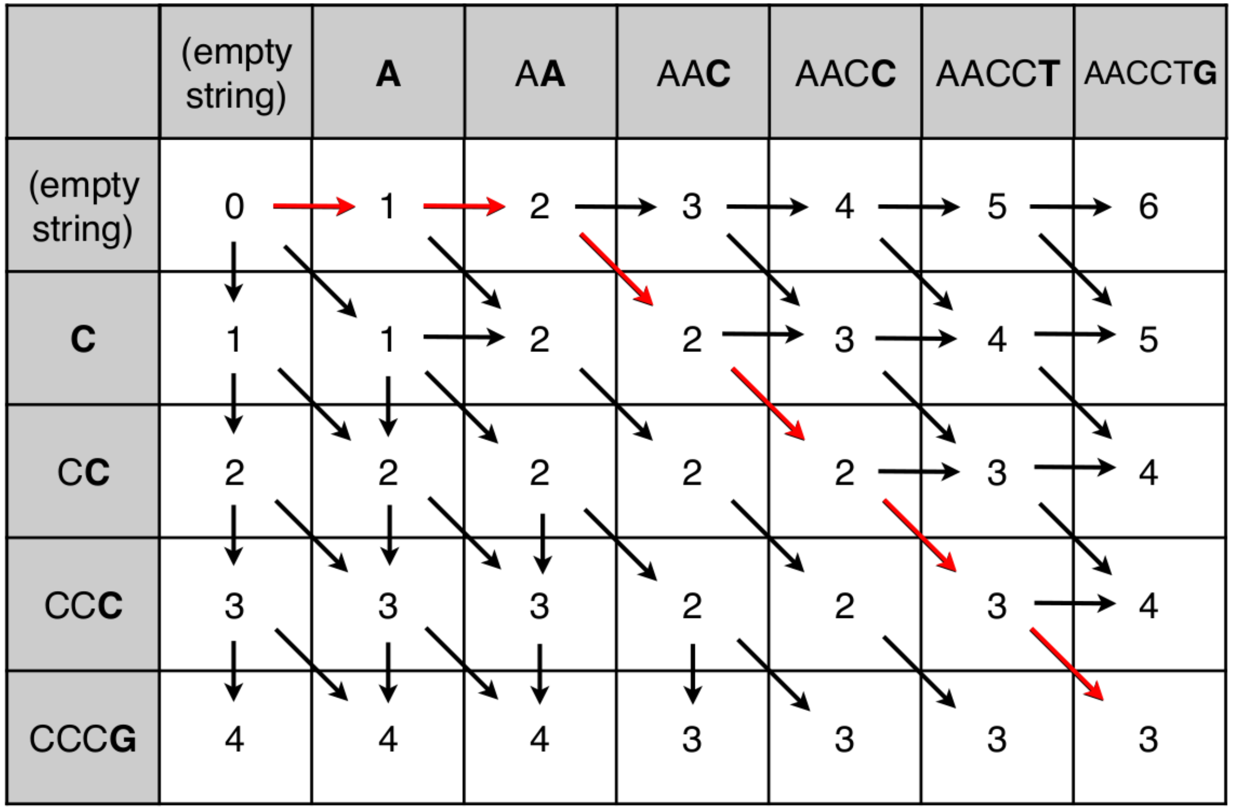


Image source: https://github.com/BenLangmead/ads1-slides/

Dynamic programming

- We perform “traceback to recover the alignment from scores



	(empty string)	**A**	A**A**	AA**C**	AAC**C**	AACC**T**	AACCT**G**
(empty string)	0	1	2	3	4	5	6
C	1	1	2	2	3	4	5
C**C**	2	2	2	2	2	3	4
CC**C**	3	3	3	2	2	3	4
CCC**G**	4	4	4	3	3	3	3

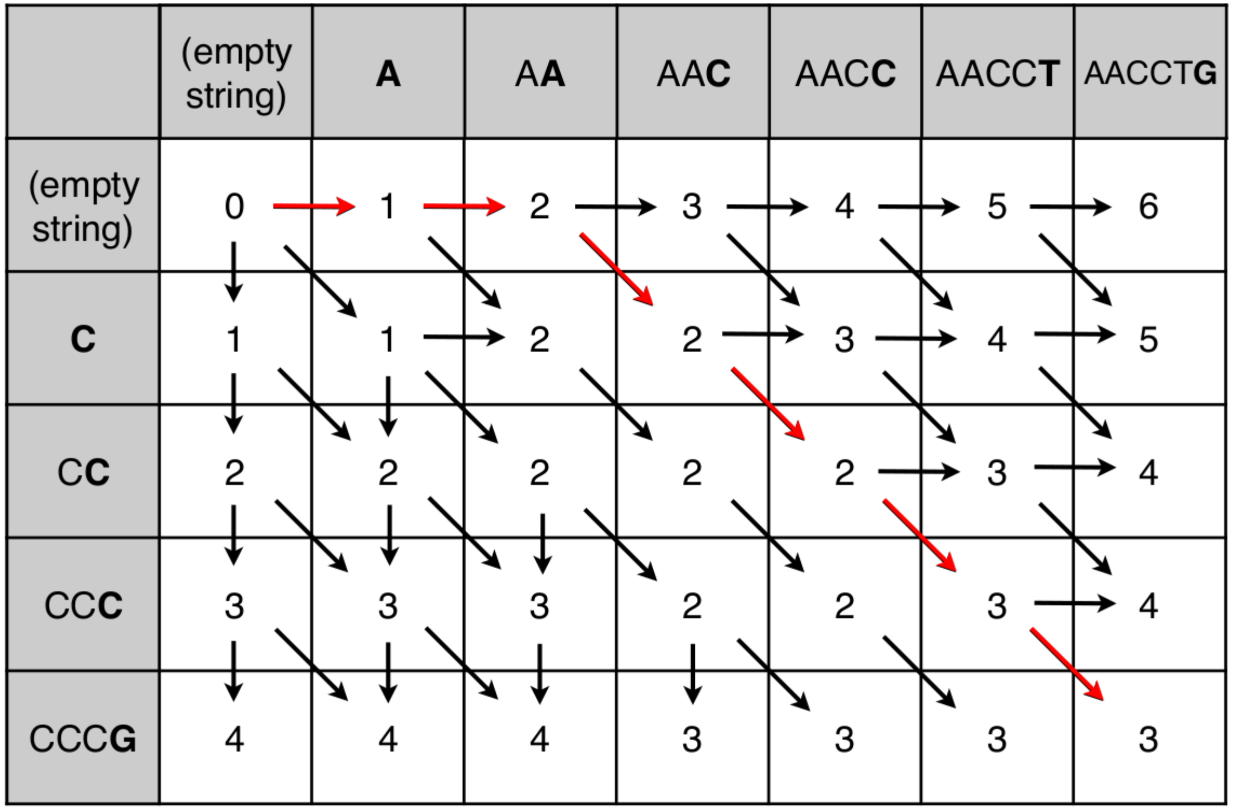
Recursion equation

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & + \quad -1 \quad a_i = b_j \\ D_{i-1,j-1} & + \quad 1 \quad a_i \neq b_j \\ D_{i-1,j} & + \quad 1 \quad b_j = - \\ D_{i,j-1} & + \quad 1 \quad a_i = - \end{cases}$$

AACCTG
__CCCG

Dynamic programming

- We perform “traceback to recover the alignment from scores



	(empty string)	A	AA	AAC	AACC	AACCT	AACCTG
(empty string)	0	1	2	3	4	5	6
C	1	1	2	2	3	4	5
CC	2	2	2	2	2	3	4
CCC	3	3	3	2	2	3	4
CCCG	4	4	4	3	3	3	3

Recursion equation

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & + & -1 & a_i = b_j \\ D_{i-1,j-1} & + & 1 & a_i \neq b_j \\ D_{i-1,j} & + & 1 & b_j = - \\ D_{i,j-1} & + & 1 & a_i = - \end{cases}$$

AACCTG
__CCCG

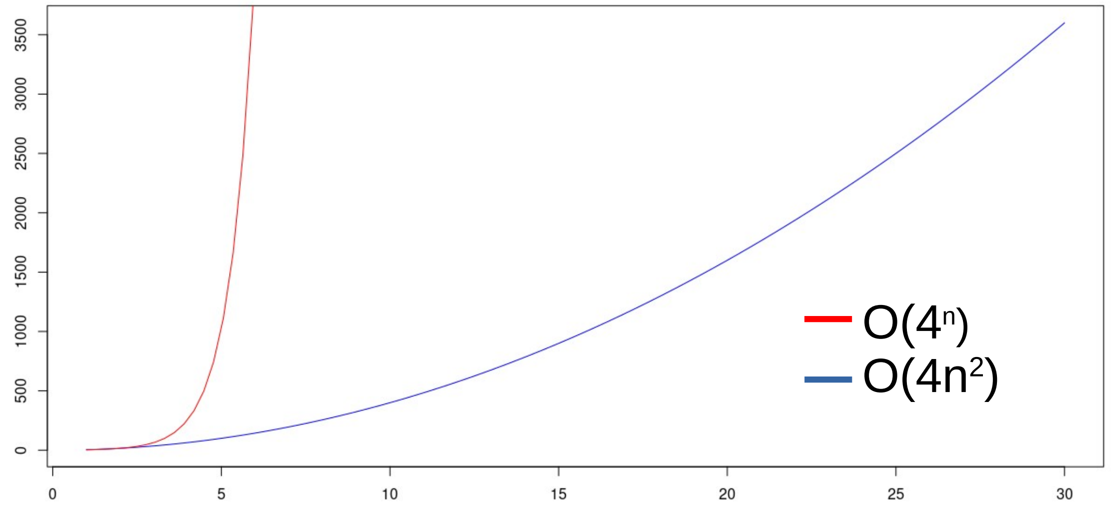
Live visualisation here !

Dynamic programming

- For any pair of sequence prefixes, score is calculated once
- Time complexity ?
  - how many operations /cell ?
  - How many cells ?

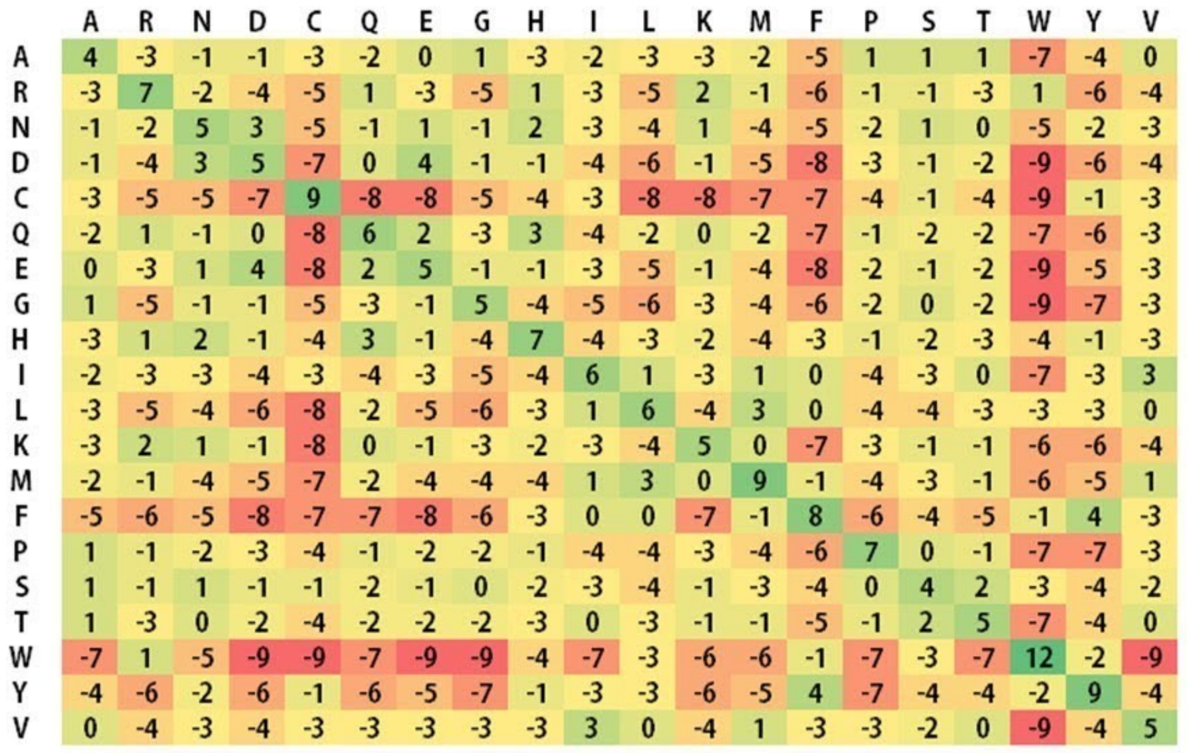
Dynamic programming

- For any pair of sequence prefixes, score is calculated once
- Time complexity ? O(mn)
  - how many operations /cell ? **3 addition and 1 minimum**
  - How many cells ? **(m + 1) (n + 1)**



Refining dynamic programming

- Add different penalties (e.g. match=1, gap=-2, mismatch=-1)
- Use a substitution matrix (measures amino-acid similarity)

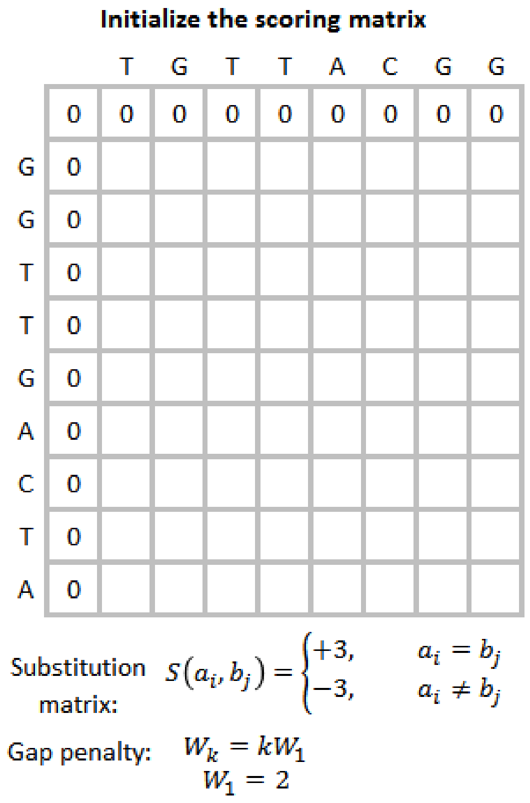


	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4	-3	-1	-1	-3	-2	0	1	-3	-2	-3	-3	-2	-5	1	1	1	-7	-4	0
R	-3	7	-2	-4	-5	1	-3	-5	1	-3	-5	2	-1	-6	-1	-1	-3	1	-6	-4
N	-1	-2	5	3	-5	-1	1	-1	2	-3	-4	1	-4	-5	-2	1	0	-5	-2	-3
D	-1	-4	3	5	-7	0	4	-1	-1	-4	-6	-1	-5	-8	-3	-1	-2	-9	-6	-4
C	-3	-5	-5	-7	9	-8	-8	-5	-4	-3	-8	-8	-7	-7	-4	-1	-4	-9	-1	-3
Q	-2	1	-1	0	-8	6	2	-3	3	-4	-2	0	-2	-7	-1	-2	-2	-7	-6	-3
E	0	-3	1	4	-8	2	5	-1	-1	-3	-5	-1	-4	-8	-2	-1	-2	-9	-5	-3
G	1	-5	-1	-1	-5	-3	-1	5	-4	-5	-6	-3	-4	-6	-2	0	-2	-9	-7	-3
H	-3	1	2	-1	-4	3	-1	-4	7	-4	-3	-2	-4	-3	-1	-2	-3	-4	-1	-3
I	-2	-3	-3	-4	-3	-4	-3	-5	-4	6	1	-3	1	0	-4	-3	0	-7	-3	3
L	-3	-5	-4	-6	-8	-2	-5	-6	-3	1	6	-4	3	0	-4	-4	-3	-3	-3	0
K	-3	2	1	-1	-8	0	-1	-3	-2	-3	-4	5	0	-7	-3	-1	-1	-6	-6	-4
M	-2	-1	-4	-5	-7	-2	-4	-4	-4	1	3	0	9	-1	-4	-3	-1	-6	-5	1
F	-5	-6	-5	-8	-7	-7	-8	-6	-3	0	0	-7	-1	8	-6	-4	-5	-1	4	-3
P	1	-1	-2	-3	-4	-1	-2	-2	-1	-4	-4	-3	-4	-6	7	0	-1	-7	-7	-3
S	1	-1	1	-1	-1	-2	-1	0	-2	-3	-4	-1	-3	-4	0	4	2	-3	-4	-2
T	1	-3	0	-2	-4	-2	-2	-2	-3	0	-3	-1	-1	-5	-1	2	5	-7	-4	0
W	-7	1	-5	-9	-9	-7	-9	-9	-4	-7	-3	-6	-6	-1	-7	-3	-7	12	-2	-9
Y	-4	-6	-2	-6	-1	-6	-5	-7	-1	-3	-3	-6	-5	4	-7	-4	-4	-2	9	-4
V	0	-4	-3	-4	-3	-3	-3	-3	-3	3	0	-4	1	-3	-3	-2	0	-9	-4	5

A→ R: -3
A → W: -7

Local alignment

- Smith-waterman uses Dynamic programming for local alignments
  - All negative scores set to 0
  - Stop extending when encountering zeros
  - Start backtracking from highest score



Initialize the scoring matrix

		T	G	T	T	A	C	G	G
	0	0	0	0	0	0	0	0	0
G	0								
G	0								
T	0								
T	0								
G	0								
A	0								
C	0								
T	0								
A	0								

Substitution matrix: $S(a_i, b_j) = \begin{cases} +3, & a_i = b_j \\ -3, & a_i \neq b_j \end{cases}$

Gap penalty: $W_k = kW_1$
$W_1 = 2$

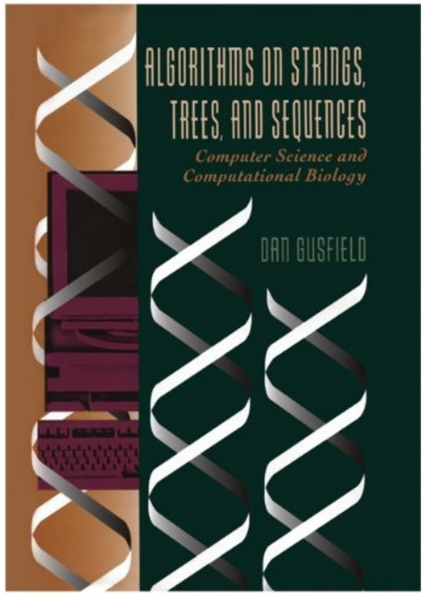
Gif: https://w.wiki/ZAL

Dynamic programming in real life

- Used in most alignment tools
- Often combined with a first step of exact matching for speed
- Example: BLAST
    1. Extract k-mers from the query
    2. Find their position in the database (exact match)
    3. Identify regions with several exact matches as “High scoring pairs”
    4. Extend HSPs in both directions using dynamic programming

Additional resources

- **Book on string-based algorithms**: Algorithms on Strings, Trees, and Sequences Computer Science and Computational Biology, Dan Gusfield


ALGORITHMS ON STRINGS, TREES, AND SEQUENCES
Computer Science and Computational Biology
DAN GUSFIELD