Combinatorial Pattern Matching
Outline

- Hash Tables
- Repeat Finding
- Exact Pattern Matching
- Keyword Trees
- Suffix Trees
- Heuristic Similarity Search Algorithms
- Approximate String Matching
- Filtration
- Comparing a Sequence Against a Database
- Algorithm behind BLAST
- Statistics behind BLAST
- PatternHunter and BLAT
Genomic Repeats

- Example of repeats:
  - ATGGTCTAGGTCCTAGTGGTC

- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats
Genomic Repeats

- The problem is often more difficult:
  - ATGGTCTAGGACCTAGTGTTC
- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats
**ℓ-mer Repeats**

- Long repeats are difficult to find
- Short repeats are easy to find (e.g., hashing)

Simple approach to finding long repeats:

- Find exact repeats of short ℓ-mers (ℓ is usually 10 to 13)
- Use ℓ-mer repeats to potentially extend into longer, maximal repeats
\textbf{\ell-mer Repeats} (cont’d)

- There are typically many locations where an \ell-mer is repeated:

\texttt{GCTTTACAGATTCAGTCTTTACAGATGGT}

- The 4-mer \texttt{TTAC} starts at locations 3 and 17
Extending $\ell$-mer Repeats

GCTTTACAGATTCAGTCTTTACAGATGGT

- Extend these 4-mer matches:

GCTTTACAGATTCAGTCTTTACAGATGGT

- Maximal repeat: TTACAGAT
Maximal Repeats

• To find maximal repeats in this way, we need ALL start locations of all \( \ell \)-mers in the genome

• **Hashing** lets us find repeats quickly in this manner
Hashing: What is it?

• What does hashing do?
  • For different data, generate a unique integer
  • Store data in an array at the unique integer index generated from the data

• Hashing is a very efficient way to store and retrieve data
Hashing: Definitions

- **Hash table**: array used in hashing
- **Records**: data stored in a *hash table*
- **Keys**: identifies sets of *records*
- **Hash function**: uses a *key* to generate an index to insert at in *hash table*
- **Collision**: when more than one record is mapped to the same index in the hash table
Hashing: Example

- Where do the animals eat?
- Records: each animal
- Keys: where each animal eats

<table>
<thead>
<tr>
<th>Records</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Penguin</td>
<td>1</td>
</tr>
<tr>
<td>Octopus</td>
<td>4</td>
</tr>
<tr>
<td>Turtle</td>
<td>3</td>
</tr>
<tr>
<td>Mouse</td>
<td>2</td>
</tr>
<tr>
<td>Snake</td>
<td>3</td>
</tr>
<tr>
<td>Heron</td>
<td>1</td>
</tr>
<tr>
<td>Tiger</td>
<td>2</td>
</tr>
<tr>
<td>Iguana</td>
<td>3</td>
</tr>
<tr>
<td>Ape</td>
<td>2</td>
</tr>
<tr>
<td>Cricket</td>
<td>4</td>
</tr>
<tr>
<td>Sparrow</td>
<td>1</td>
</tr>
</tbody>
</table>
Hashing: Finding Repeats

- To find repeats in a genome:
  - For all $\ell$mers in the genome, note the start position and the sequence
  - Generate a hash table index for each unique $\ell$mer sequence
  - In each index of the hash table, store all genome start locations of the $\ell$mer which generated that index
Hashing: Collisions

- Dealing with collisions:
  - “Chain” all start locations of $\ell$-mers (linked list)
What if, instead of finding repeats in a genome, we want to find all sequences in a database that contain a given pattern?

This leads us to a different problem, the *Pattern Matching Problem*
Pattern Matching Problem

- **Goal:** Find all occurrences of a pattern in a text

- **Input:** Pattern $p = p_1 \ldots p_n$ and text $t = t_1 \ldots t_m$

- **Output:** All positions $1 \leq i \leq (m - n + 1)$ such that the $n$-letter substring of $t$ starting at $i$ matches $p$

- **Motivation:** Searching database for a known pattern
Exact Pattern Matching: A Brute-Force Algorithm

PatternMatching(p,t)

1 $n \leftarrow$ length of pattern $p$
2 $m \leftarrow$ length of text $t$
3 for $i \leftarrow 1$ to $(m - n + 1)$
4 if $t_{i} \ldots t_{i+n-1} = p$
5 output $i$
Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:

- Pattern `GCAT`
- Text `CGCATC`
Exact Pattern Matching: Running Time

- *PatternMatching* runtime: $O(nm)$
- Probability-wise, it’s more like $O(m)$
  - Rarely will there be close to $n$ comparisons in line 4
- Better solution: *suffix trees*
  - Can solve problem in $O(m)$ time
  - Conceptually related to *keyword trees*
Multiple Pattern Matching Problem

• **Goal**: Given a set of patterns and a text, find all occurrences of any of patterns in text

• **Input**: $k$ patterns $p^1, \ldots, p^k$, and text $t = t_1 \ldots t_m$

• **Output**: Positions $1 \leq i \leq m$ where substring of $t$ starting at $i$ matches $p_j$ for $1 \leq j \leq k$

• **Motivation**: Searching database for known multiple patterns
Multiple Pattern Matching: Straightforward Approach

- Can solve as $k$ “Pattern Matching Problems”
- Runtime:
  
  $O(kmn)$

  using the PatternMatching algorithm $k$ times

- $m$ - length of the text
- $n$ - average length of the pattern
Keyword Trees: Example

- **Keyword tree:**
  - Apple
Keyword Trees: Example (cont’d)

- **Keyword tree:**
  - Apple
  - Apropos
Keyword Trees: Example (cont’d)

- **Keyword tree:**
  - Apple
  - Apropos
  - Banana
Keyword Trees: Example (cont’d)

- **Keyword tree:**
  - Apple
  - Apropos
  - Banana
  - Bandana
Keyword Trees: Example (cont’d)

- **Keyword tree:**
  - Apple
  - Apropos
  - Banana
  - Bandana
  - Orange
Keyword Trees: Threading (cont’d)

• Thread “appeal”
• appeal
Keyword Trees: Threading (cont’d)

- Thread “appeal”
- appeal
Keyword Trees: Threading (cont’d)

- Thread “appeal”
- appeal
Keyword Trees: Threading (cont’d)

- Thread “appeal”
- appeal
Keyword Trees: Threading (cont’d)

- Thread “apple”
- apple
Keyword Trees: Threading (cont’d)

- Thread “apple”
- apple
Keyword Trees: Threading (cont’d)

- Thread “apple”
- apple
Keyword Trees: Threading (cont’ d)

• Thread “apple”
• apple
Keyword Trees: Threading (cont’d)

• Thread “apple”
• apple
Keyword Trees: Threading

• To match patterns in a text using a keyword tree:
  • Build keyword tree of patterns
  • “Thread” the text through the keyword tree

$t = \text{"mr and mrs dursley of number 4 Privet drive were proud to say that they were perfectly normal thank you very much"}$
Keyword Trees: Threading (cont’d)

- Threading is “complete” when we reach a leaf in the keyword tree.

- When threading is “complete,” we’ve found a pattern in the text.

\[ t = "m and mrs dursley of number 4 privet drive were proud to say that they were perfectly normal thank you very much" \]
Suffix Trees

- Similar to keyword trees, except edges that form paths are collapsed
- Each edge is labeled with a **substring** of a text
- All internal edges have at least two outgoing edges
- Leaves labeled by the index of the pattern.
Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes
Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

How much time does it take?
Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes.

Time is linear in the total size of all suffixes, i.e., it is quadratic in the length of the text.
Suffix Trees: Advantages

- Suffix trees of a text are constructed for all its suffixes.
- Suffix trees build faster than keyword trees.

ATCATG
TCATG
CATG
ATG
TG
G

quadratic

Keyword Tree

linear (Weiner suffix tree algorithm)

Suffix Tree
Use of Suffix Trees

- Suffix trees hold all suffixes of a text
  - i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
- Builds in $O(m)$ time for text of length $m$
- To find any pattern of length $n$ in a text:
  - Build suffix tree for text
  - Thread the pattern through the suffix tree
  - Can find pattern in text in $O(n)$ time!
Suffix Trees: Example

Figure 9.6 Threading the pattern ATG through the suffix tree for the text ATGCATA-CATGG. The suffixes ATGCATACATGG and ATGG both match, as noted by the gray vertices in the tree (the p-matching leaves). Each p-matching leaf corresponds to a position in the text where p occurs.
Multiple Pattern Matching: Summary

- Keyword and suffix trees are used to find patterns in a text

- **Keyword trees:**
  - Build keyword tree of patterns, and *thread text* through it

- **Suffix trees:**
  - Build suffix tree of text, and *thread patterns* through it
Approximate vs. Exact Pattern Matching

- So far all we’ve seen exact pattern matching algorithms
- Usually, because of mutations, it makes much more biological sense to find approximate pattern matches
- Biologists often use fast heuristic approaches (rather than local alignment) to find approximate matches
Heuristic Similarity Searches

- Genomes are huge: Smith-Waterman quadratic alignment algorithms are too slow
- Alignment of two sequences usually has short identical or highly similar fragments
- Many heuristic methods (i.e., FASTA) are based on the same idea of *filtration*
  - Find short exact matches, and use them as seeds for potential match extension
  - “Filter” out positions with no extendable matches
Dot Matrices

- Dot matrices show similarities between two sequences.
- FASTA makes an implicit dot matrix from short exact matches, and tries to find long diagonals (allowing for some mismatches).

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>C</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>T</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>C</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>T</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>T</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>A</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>G</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

The diagram shows a dot matrix with matches indicated by asterisks. The sequences are aligned vertically and horizontally, and matches are marked along the diagonals.
Dot Matrices (cont’d)

- Identify diagonals above a threshold length
- Diagonals in the dot matrix indicate exact substring matching
Diagonals in Dot Matrices

• Extend diagonals and try to link them together, allowing for minimal mismatches/indels
• Linking diagonals reveals approximate matches over longer substrings
Approximate Pattern Matching Problem

Goal: Find all approximate occurrences of a pattern in a text

Input: A pattern $p = p_1 \ldots p_n$, text $t = t_1 \ldots t_m$, and $k$, the maximum number of mismatches

Output: All positions $1 \leq i \leq (m - n + 1)$ such that $t_i \ldots t_{i+n-1}$ and $p_1 \ldots p_n$ have at most $k$ mismatches (i.e., Hamming distance between $t_i \ldots t_{i+n-1}$ and $p \leq k$)
Approximate Pattern Matching: A Brute-Force Algorithm

```
ApproximatePatternMatching(p, t, k)
1  n ← length of pattern p
2  m ← length of text t
3  for i ← 1 to m - n + 1
4      dist ← 0
5      for j ← 1 to n
6          if t_{i+j-1} ! = p_j
7              dist ← dist + 1
8          if dist ≤ k
9              output i
```
Approximate Pattern Matching: Running Time

- We can generalize the “Approximate Pattern Matching Problem” into the “Query Matching Problem”:
  - We want to match substrings in a query to substrings in a text with at most $k$ mismatches
  - **Motivation**: we want to see similarities to some gene, but we may not know which parts of the gene to look for
Query Matching Problem

- **Goal**: Find all substrings of the query that approximately match the text
- **Input**: Query \( q = q_1 \ldots q_w \),
  text \( t = t_1 \ldots t_m \),
  \( n \) (length of matching substrings),
  \( k \) (maximum number of mismatches)
- **Output**: All pairs of positions \((i, j)\) such that the
  \( n \)-letter substring of \( q \) starting at \( i \) approximately matches the
  \( n \)-letter substring of \( t \) starting at \( j \),
  with at most \( k \) mismatches
Approximate Pattern Matching vs Query Matching

(a) Approximate Pattern Matching

(b) Query Matching
Query Matching: Main Idea

- Approximately matching strings share some perfectly matching substrings.
- Instead of searching for approximately matching strings (difficult) search for perfectly matching substrings (easy).
Filtration: Match Detection

- If \( x_1 \ldots x_n \) and \( y_1 \ldots y_n \) match with at most \( k \) mismatches, they must share an \( \mathcal{L} \)-mer that is perfectly matched, with \( \mathcal{L} = \left\lceil \frac{n}{k + 1} \right\rceil \) (see the textbook for the proof).

- Break string of length \( n \) into \( k+1 \) parts, each of length \( \left\lfloor \frac{n}{k + 1} \right\rfloor \)
  - \( k \) mismatches can affect at most \( k \) of these \( k+1 \) parts
  - At least one of these \( k+1 \) parts is perfectly matched
Filtration for Query Matching is a Two Stage Process

- **Potential match detection (stage 1):** find all matches of \( \ell \)-mers in query and text

- **Potential match verification (stage 2):** Verify each potential match by extending it to the left and right until the first \( k+1 \) mismatches are found
Filtration: More Details

- Potential match detection in this algorithm can be implemented either by hashing or using a suffix tree.

- For most practical values of n and k, the number of potential matches between the text and the query is small, yielding a fast algorithm.
Local alignment is to slow...

- Quadratic local alignment is too slow while looking for similarities between long strings (e.g. the entire GenBank database)
Local alignment is too slow... 

• Quadratic local alignment is too slow while looking for similarities between long strings (e.g. the entire GenBank database)

$$s_{i,j} = \max \begin{cases} 
0 \\ 
S_{i-1,j} + \delta(v_i, -) \\ 
S_{i,j-1} + \delta(-, w_j) \\ 
S_{i-1,j-1} + \delta(v_i, w_j) 
\end{cases}$$
BLAST

• A tool for comparing a sequence against a database
  • similarity can reveal evolutionary relationships, potential genes, and protein functions.
• Minimizes search space instead of exploring entire search space between two sequences
• Finds short exact matches ("seeds"), only explores locally around these "hits"
BLAST algorithm

• **Keyword search** of all words of length $w$ from the query of length $n$ in database of length $m$ with score above threshold
  • $w = 11$ for DNA queries, $w = 3$ for proteins
• **Local alignment extension** for each found keyword
  • Extend result until longest match above threshold is achieved
• Running time $O(nm)$
**BLAST algorithm (cont’d)**

Query: KRHRKVLRDNIQGITKPAIRRLARRGGVKRISGLIYEETRGVLKIFLENVIRD

Keyword:
- GVK 18
- GAK 16
- GIK 16
- GGK 14
- GLK 13
- GNK 12
- GRK 11
- GEK 11
- GDK 11

Neighborhood words (T = 13):
- GVN
- GAN
- GIN
- GGN
- GGN
- GRG
- GEG
- GDK

Extension:
- Query: 22 VLRDNIQGITKPAIRRLARRGGVKRISGLIYEETRGVLK 60
  +++DN +G + IR L G+K I+ L+ E+ RG++K
- Sbjct: 226 IKDNGRGFSGKQIRNLNYGIGLKVIA+DLV-EKHRIIK 263

High-scoring Pair (HSP)
Original BLAST

- **Dictionary**
  - All words of length $w$

- **Alignment**
  - *Ungapped* extensions until score falls below some statistical threshold

- **Output**
  - All local alignments with score $> \text{threshold}$
Original BLAST: Example

- $w = 4$
- Exact keyword match of **GGTC**
- Extend diagonals with mismatches until score is under 50%
- Output result:
  - GTAAGGTCC
  - GTTAGGTCC

From lectures by Serafim Batzoglou (Stanford)
Gapped BLAST: Example

- Original BLAST exact keyword search, THEN:
- Extend with gaps around ends of exact match until score < threshold
- Output result GTAAGGTCCAGT GTTAGGTC-AGT

From lectures by Serafim Batzoglou (Stanford)
Assessing sequence similarity

- Need to know how strong an alignment can be expected from chance alone
- “Chance” relates to comparison of sequences that are generated randomly based upon a certain sequence model
- Sequence models may take into account:
  - G+C content
  - Poly-A tails
  - “Junk” DNA
  - Codon bias
  - Etc.
BLAST: Segment Score

- BLAST uses scoring matrices ($\delta$) to improve on efficiency of match detection
  - Some proteins may have very different amino acid sequences, but are still similar
- For any two $\ell$-mers $x_1 \ldots x_\ell$ and $y_1 \ldots y_\ell$:
  - **Segment pair**: pair of $\ell$-mers, one from each sequence
  - **Segment score**: $\sum_{i=1}^{\ell} \delta(x_i, y_i)$
BLAST: Locally Maximal Segment Pairs

- A segment pair is **maximal** if it has the best score over all segment pairs.
- A segment pair is **locally maximal** if its score can’t be improved by extending or shortening.
- Statistically significant **locally maximal** segment pairs are of biological interest.
- BLAST finds all locally maximal segment pairs with scores above some threshold.
  - A significantly high threshold will filter out some statistically insignificant matches.
BLAST: Statistics

• Threshold: Altschul-Dembo-Karlin statistics
  • Identifies smallest segment score that is unlikely to happen by chance
• # matches above $\theta$ has mean $E(\theta) = Kmne^{-\lambda \theta}$; $K$ is a constant, $m$ and $n$ are the lengths of the two compared sequences
• Parameter $\lambda$ is positive root of:
  $$\sum_{x,y \text{ in } A} p_x p_y e^{\delta(x,y)} = 1,$$
  where $p_x$ and $p_y$ are frequencies of amino acids $x$ and $y$, and $A$ is the twenty letter amino acid alphabet
P-values

- The probability of finding $b$ HSPs with a score $\geq S$ is given by:
  - $(e^{-E}E^b)/b!$
- For $b = 0$, that chance is:
  - $e^{-E}$
- Thus the probability of finding at least one HSP with a score $\geq S$ is:
  - $P = 1 - e^{-E}$
Sample BLAST output

- Blast of human beta globin protein against zebra fish

Sequences producing significant alignments:

<table>
<thead>
<tr>
<th>Seq ID</th>
<th>Description</th>
<th>Score (bits)</th>
<th>Expect</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gi</td>
<td>18858329</td>
<td>ref</td>
<td>NP_571095.1</td>
<td>ba1 globin [Danio rerio]</td>
</tr>
<tr>
<td>gi</td>
<td>18858331</td>
<td>ref</td>
<td>NP_571096.1</td>
<td>ba2 globin; SI:dZ118J2.3 [Danio rerio]</td>
</tr>
<tr>
<td>gi</td>
<td>37606100</td>
<td>emb</td>
<td>CAE48992.1</td>
<td>SI:bY187G17.6 (novel beta globin) [Danio rerio]</td>
</tr>
<tr>
<td>gi</td>
<td>31419195</td>
<td>gb</td>
<td>AAH53176.1</td>
<td>Bal protein [Danio rerio]</td>
</tr>
</tbody>
</table>

ALIGNMENTS

>gi|18858329|ref|NP_571095.1| ba1 globin [Danio rerio]

Length = 148

Score = 171 bits (434), Expect = 3e-44
Identities = 76/148 (51%), Positives = 106/148 (71%), Gaps = 1/148 (0%)

Query: 1

MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPK 60
MV T E++A+ LWGK+N+DE+G +AL R L+VYPWTQR+F +FG+LS+P A+MGNPK

Sbjct: 1

MVEWTDAERTAILGLWGTKNIDEIGPQALSRCGLIVYPWTQRYFATFGNLSPPAAIIMGNK 60

Query: 61

VKAHGKKVLGAFSDLGLHNLKGTATLSELHCDKLHVDPENFRLLGNVLVCVLAHF 120
V AHG+ V+G + ++DN+K T+A LS +H +KLHVDP+NFRLL + + A FG

Sbjct: 61

VAAHGRTVMGGGLERAIAKNMDNVKNTYAAALSVMHSEKLVDPDNFRLLLADICTVCAAMKFG 120

Query: 121

KE-FTPPVQAAYQKVAVANALAHKYH 147
+ F VQ A+QK +A V +AL +YH

Sbjct: 121 QAGFNADVQEAQKFLAVVVSALCRQYH 148
Sample BLAST output (cont’d)

- Blast of human beta globin DNA against human DNA

Sequences producing significant alignments:

<table>
<thead>
<tr>
<th>Accession</th>
<th>Description</th>
<th>Score</th>
<th>E Value (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gi</td>
<td>19849266</td>
<td>gb</td>
<td>AF487523.1</td>
</tr>
<tr>
<td>gi</td>
<td>183868</td>
<td>gb</td>
<td>M11427.1</td>
</tr>
<tr>
<td>gi</td>
<td>44887617</td>
<td>gb</td>
<td>AY534688.1</td>
</tr>
<tr>
<td>gi</td>
<td>31726</td>
<td>emb</td>
<td>V00512.1</td>
</tr>
<tr>
<td>gi</td>
<td>38683401</td>
<td>ref</td>
<td>NR_001589.1</td>
</tr>
<tr>
<td>gi</td>
<td>18462073</td>
<td>gb</td>
<td>AF339400.1</td>
</tr>
</tbody>
</table>

ALIGNMENTS

>gi|28380636|ref|NG_000007.3| Homo sapiens beta globin region (HBB@) on chromosome 11
Length = 81706
Score = 149 bits (75), Expect = 3e-33
Identities = 183/219 (83%)
Strand = Plus / Plus

Query: 267   ttggagatgccacaaagcacctggatgatctcaagggcacctttgcccagctgagtgaa 326
Sbjct: 54409 ttcggaaaagctgttatgctcacggatgacctcaaaggcacctttgctacactgagtgac 54468

Query: 327   ctgcactgtgacaagctgcacgtggaccctgagaacttc 365
Sbjct: 54469 ctgcactgtaacaagctgcacgtggaccctgagaacttc 54507
Timeline

1970: Needleman-Wunsch global alignment algorithm
1981: Smith-Waterman local alignment algorithm
1985: FASTA
1990: BLAST (basic local alignment search tool)
2000s: BLAST has become too slow in “genome vs. genome” comparisons - new faster algorithms evolve!
   • Pattern Hunter
   • BLAT