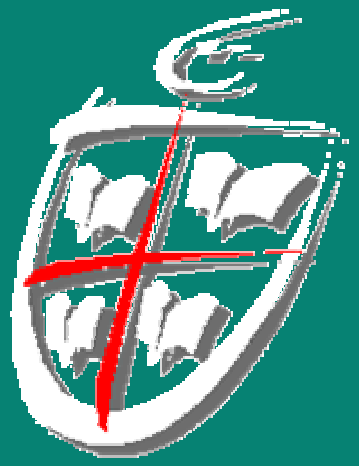


CS900-3a
Computer Science –
A Multifaceted Introduction

Lecture #14
String Matching

String Matching (1)

Problem Formulation

- Find the first occurrence of a string (**pattern**) as a substring in another string (**text**).
- Implement as a function with the following signature

```
int match(string text, string pattern);
```

  - In case that the pattern is found in the text, the returned integer will denote the beginning of the (first) pattern occurrence.
  - Otherwise the function will return the null pointer.

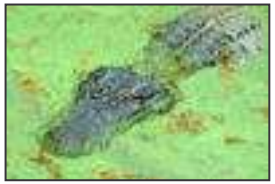
String Matching (2)

- Applications
  - Search functionality in **text editors & web browsers**
  - Search engines
  - **Molecular biology** (finding patterns in DNA / proteins)

String Matching (3)

- How can this be achieved? Suggestions for an algorithm?

- Basic idea: slide pattern window over text and check for equality at every possible position.
- Example: (Hemoglobin of the American Alligator)

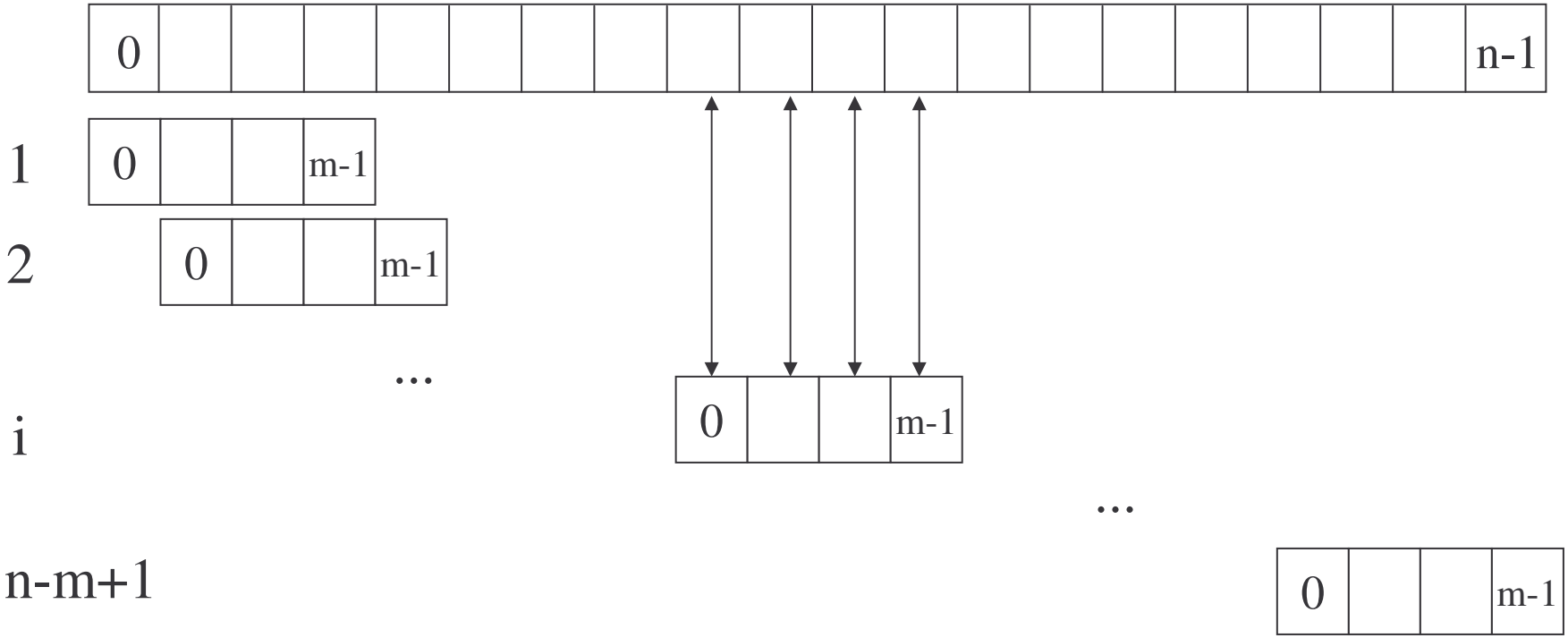


V	L	S	M	E	D	K	S	N	V	K	A	I	W	G	K	A	S	G	H	. . .
K	K	K	K	K	K	K	K	K	K	K	A	I								

- Worst case run time: $(n-m+1) \cdot m$, where n is the length of the text and m is the length of the pattern

String Matching (4)

- Worst case run time: $(n-m+1) \cdot m$, where n is the length of the text and m is the length of the patterm



String Matching, C++ Implementation (1)

- Implementation of the brute-force string matching algorithm in C++

```
int match(string text, string pattern)
{
   /* loop over all possible n-m+1 alignments     */
   /* compare the pattern with the current window */
   /* if all characters equal, return pointer     */
   /* otherwise, try next window                  */
   /* if no match found, return null pointer      */
}
```

String Matching, C++ Implementation (2)

- Implementation of the brute-force string matching algorithm in C++

```
int match(string text, string pattern)
{
   int i, j, n ,m ;
   n = text.length();
   m = pattern.length();

   /* loop over all possible n-m+1 alignments     */
   /* compare the pattern with the current window */
   /* if all characters equal, return pointer     */
   /* otherwise, try next window                  */
   /* if no match found, return null pointer      */
}
```

String Matching, C++ Implementation (3)

- Implementation of the brute-force string matching algorithm in C++

```
int match(string text, string pattern)
{
   int i, j, n ,m ;
   n = text.length();
   m = pattern.length();

   for (i = 0; i < (n-m+1); i++) {
       /* compare the pattern with the current window */
       /* if all characters equal, return pointer     */
       /* otherwise, try next window                  */
   }
   return -1;
}
```

String Matching, C Implementation (4)

- Implementation of the brute-force string matching algorithm in C++

```
int match(string text, string pattern)
{
   int i, j, n ,m ;
   n = text.length();
   m = pattern.length();

   for (i = 0; i < (n-m+1); i++) {
      for (j = 0; j < m ; j++)
      /* if all characters equal, return pointer     */
      /* otherwise, try next window                  */
   }
   return -1;
}
```

String Matching, C++ Implementation (5)

- Implementation of the brute-force string matching algorithm in C++

```
int match(string text, string pattern)
{
   int i, j, n ,m ;
   n = text.length();
   m = pattern.length();

   for (i = 0; i < (n-m+1); i++) {
      for (j = 0; j < m && text[i+j] == pattern[j]; j++);
      if (j == m)
         return i;
   }
   return -1;
}
```

Improved String Matching (1)

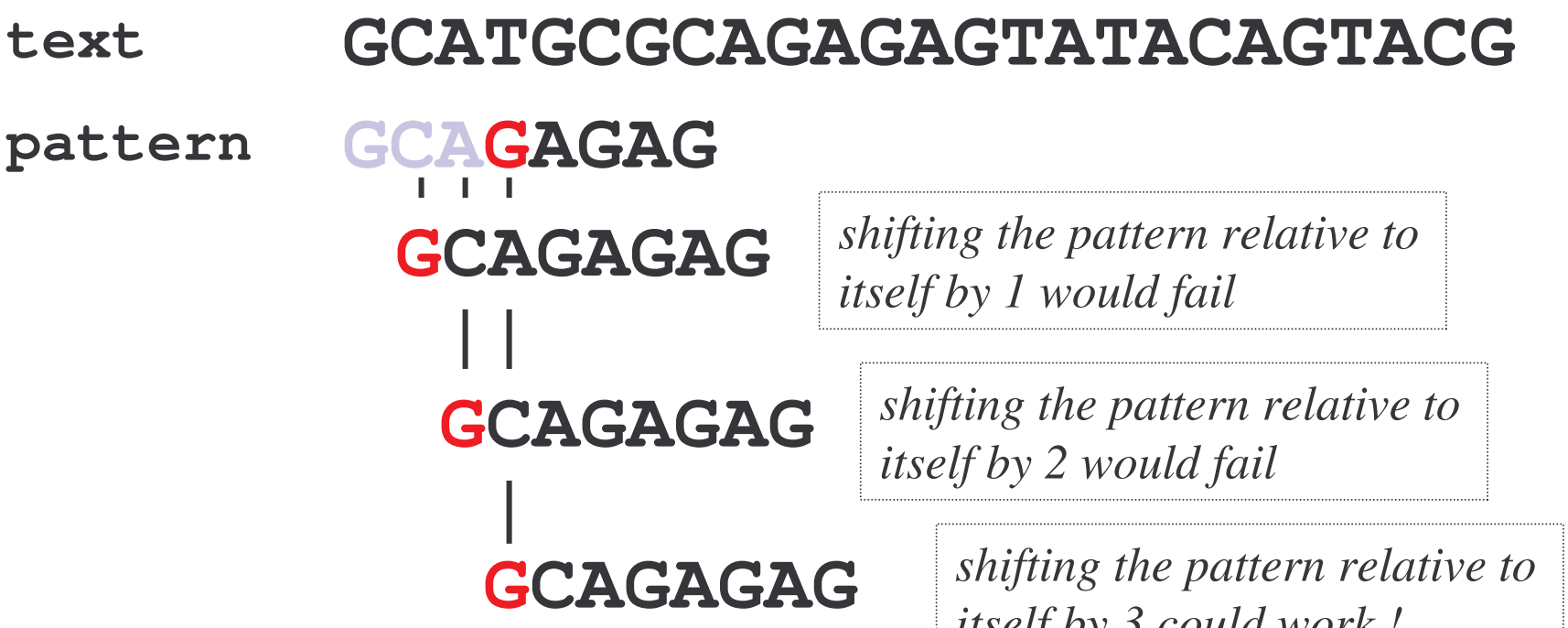
- Is there a better - i.e. more efficient - way to perform string matching than the brute-force approach?
- When a character mismatch is encountered, there should be a way that we can skip some of the comparisons!
- Example:

```
text      GCATGCGCAGAGAGTATACAGTACG

pattern   GCAGAGAG
```

Since **text** and **pattern** share the same prefix, we can compute a shift just based on the pattern.

Improved String Matching (2)

- Let's investigate this in more detail:



Improved String Matching (3)

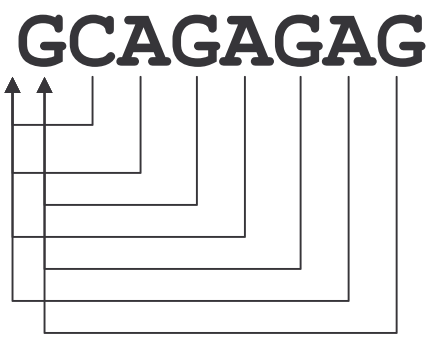
- Longer example:

```
GCATGCGCAGAGAGTATACAGTACG
GCAGAGAG
 GCAGAGAG
  GCAGAGAG
   GCAGAGAG
    GCAGAGAG
     GCAGAGAG
      GCAGAGAG
```

Morris - Pratt Algorithm (1)

- Key idea: pre-compute shifts for each "failing" character position based on **`pattern`** only!
- Store shifts in an array **`mpNext`**.
- In the example:

n:	0	1	2	3	4	5	6	7	8
mpNext[n]:	−1	0	0	0	1	0	1	0	1
shift:		1	1	2	3	3	5	5	7	7



Morris - Pratt Algorithm (2a)

- How are shifts computed?

Morris - Pratt Algorithm (2b)

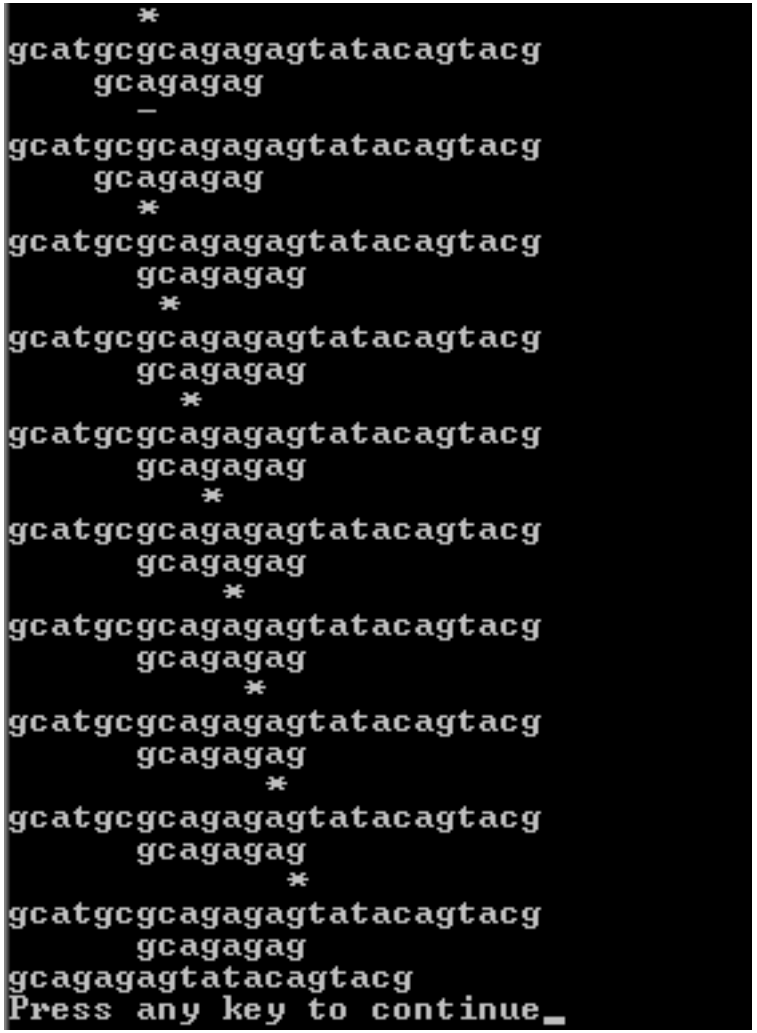
- How are shifts computed?

Morris - Pratt Algorithm (3)

- The search algorithm:

```
int MP(string text, string pattern)
{
   int i, j, n, m, *mpNext;
   n      = text.length();
   m      = pattern.length();
   mpNext = new int [m];

   //-- preprocessing
   preprocessMP(pattern, mpNext);

   //-- searching
   j = 0;
   for (i = 0; i < n; i++) {
      while (j >=0 && text[i] != pattern[j])
         j = mpNext[j];
      i++; j++;
      if (j >= m)
         return i-j;
   }
   return -1;
}
```

Morris - Pratt Algorithm (4)



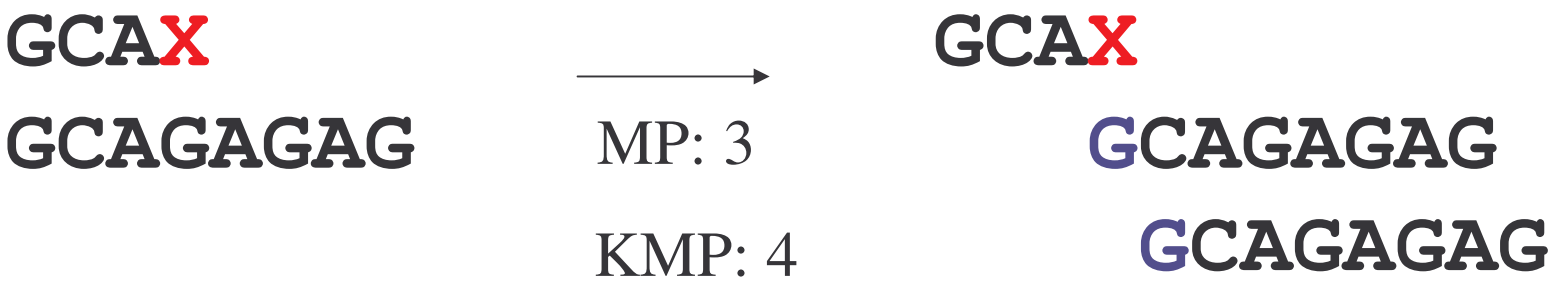
Morris - Pratt Algorithm (5)

- The pre-computation of the shifts can be done in a manner that is very similar to the actual string matching.

```
void precomputeMP(string x, int mpNext[])
{
   int i, j, m;
   i = 0;
   j = mpNext[0] = -1;
   m = x.length();
   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = mpNext[j];
      i++; j++;
      mpNext[i] = j;
   }
}
```

Knuth-Morris-Pratt Algorithm (1)

- An even better shift can be computed by avoiding to explicitly compare the same character in **`text`** twice in certain cases.
- This can be done, if the new character in the pattern based on MP would be the same as the failing one.
- The improved computation of the shifts leads to the KMP algorithm.
- Example:



Knuth-Morris-Pratt Algorithm (2)

- The search algorithm is the same, only the computation of the shift/next character function changes:

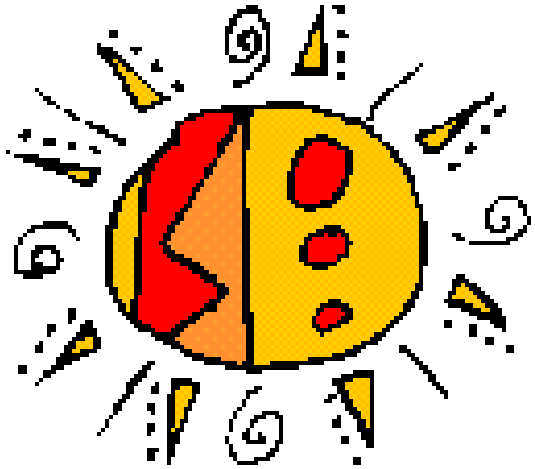
```
void precomputeMP(string x, int kmpNext[])
{
   int i, j, m;
   i = 0;
   j = kmpNext[0] = -1;
   m = x.length();

   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = kmpNext[j];
      i++; j++;
      if (x[i] == x[j]) kmpNext[i] = kmpNext[j];
      else              kmpNext[i] = j;
   }
}
```

Knuth-Morris-Pratt Algorithm (3)

- How many comparisons does this algorithm perform?
- One can show that the algorithm needs **`2n−1`** comparisons in the worst case to do the search.
- Is it possible to do better?
  - Further improvements: Colussi algorithm (left/right scanning), **`3/2 n`** comparisons.
  - Further refinement: Galil-Giancarlo algorithm **`4/3 n`** comparisons
  - Most efficient in most applications: Boyer-Moore algorithm **`3n`** worst case, proportional to **`n/m`** average case.
  - Lower bounds, e.g. **`[1+2/(m+3)] n`**
  - *It can sometimes be hard to find the most efficient algorithm even for a very simple problem ...*

break

- It’s time for a break – 10 minutes!



Computational Biology (1)

Strings are important in computational biology:

- **DNA**, the basis for heredity, is a polymer consisting of small molecules called **nucleotides**.
  - There are four of them: Adenine (A), cytosine (C), guanine (G) and thymine (T).
  - Hence a DNA sequence can be represented as a string over the alphabet {A,C,G,T}
- **Proteins**, the most important building blocks of living organisms, are composed of **amino acids**.
  - There are basically 20 amino acids.
  - Hence a protein (primary structure) can be represented as a string over a 20 letter alphabet.

Computational Biology (2)

- There is considerable variability in the DNA as well as in proteins.
  - Genes (certain substrings of DNA) may vary between species and even between individuals due to mutations etc.
  - Related proteins (families, homologies) will not be exactly identical, but will share certain subsequences. They will not be exact copies, but they will often be **similar**.
- Exact matching is not very helpful to compare and understand DNA sequences and proteins.
- One uses **inexact matching** techniques instead!

Edit Distance and Edit Costs

- One would like to define a distance function between strings.
- Distances will be based on certain editing operations and the associated goodness: how can one string be transformed into the other with the smallest number of edits or the optimal edit sequence.
- Example:
  - Correct match: `1.0`
  - Incorrect match: `-1.0`
  - Insertion of a blank: `-0.5`

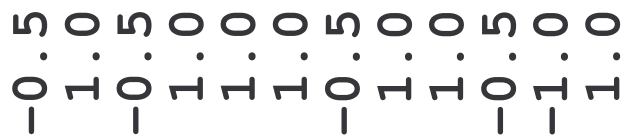
TGCTATCGAA

?

ATCTACGGGA

––TGCTATCG–AA

AT–CTA–CGGGA–



+4.0

Optimal Alignment (1)

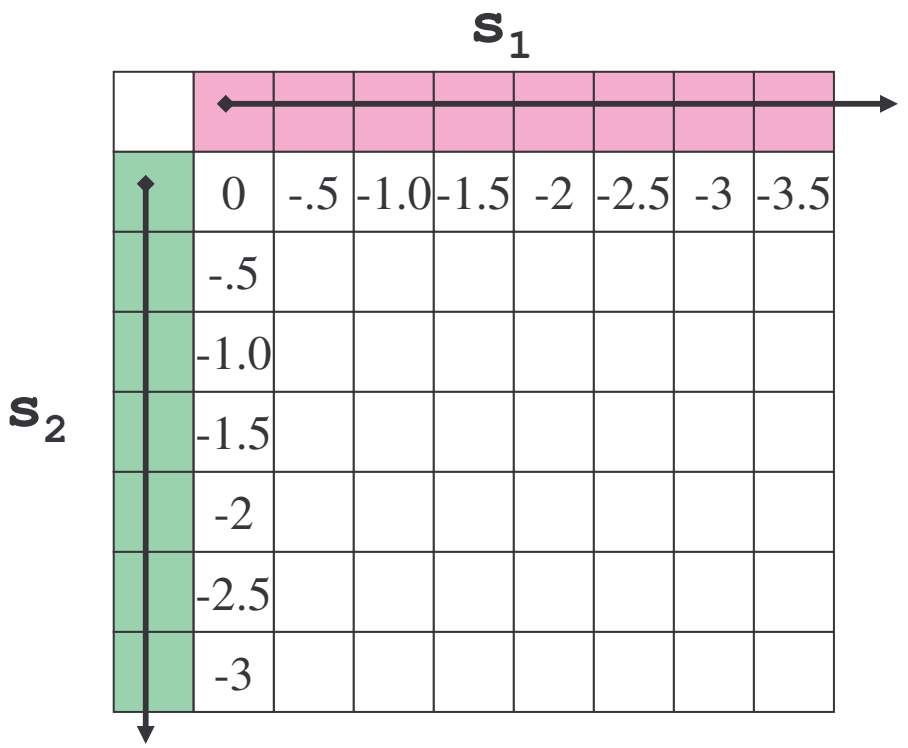
- There are typically very many ways to align two strings $\mathbf{s}_1$ and $\mathbf{s}_2$ by inserting blanks. Brute-force search will not be successful.
- Instead one can use a powerful technique known as **dynamic programming**.
- Compute a matrix $\mathbf{A}$ with elements $\mathbf{A}_{ij}$ such that $\mathbf{A}_{ij}$ is defined as the best score obtainable for a prefix of $\mathbf{s}_1$ and $\mathbf{s}_2$ of length $\mathtt{i}$ and $\mathtt{j}$, respectively.

Optimal Alignment (2)

- We would like to obtain suitable recurrence relations...
- A_{ij} denotes the cost associated with the optimal alignment of the first i characters of the first string and the first j characters of the second string
- Assume that these numbers for i<k and j<l were known and we want to compute A_{kl}
- How can we obtain an alignment of a k-prefix of the first string with a l-prefix of the second one? 3 Ways:
  - align i=k-1 prefix with j=l-1 prefix and match the last two characters
  - align i=k prefix with j=l-1 prefix and match the l-th letter in the second string with a gap in the first
  - align i=k-1 prefix with j=l prefix and match the kl-th letter in the first string with a gap in the second

Optimal Alignment (3)

- We start by computing the first row and the first column of the matrix/table.
  - $A_{0i} = A_{i0}$ = i*penalty for insertion

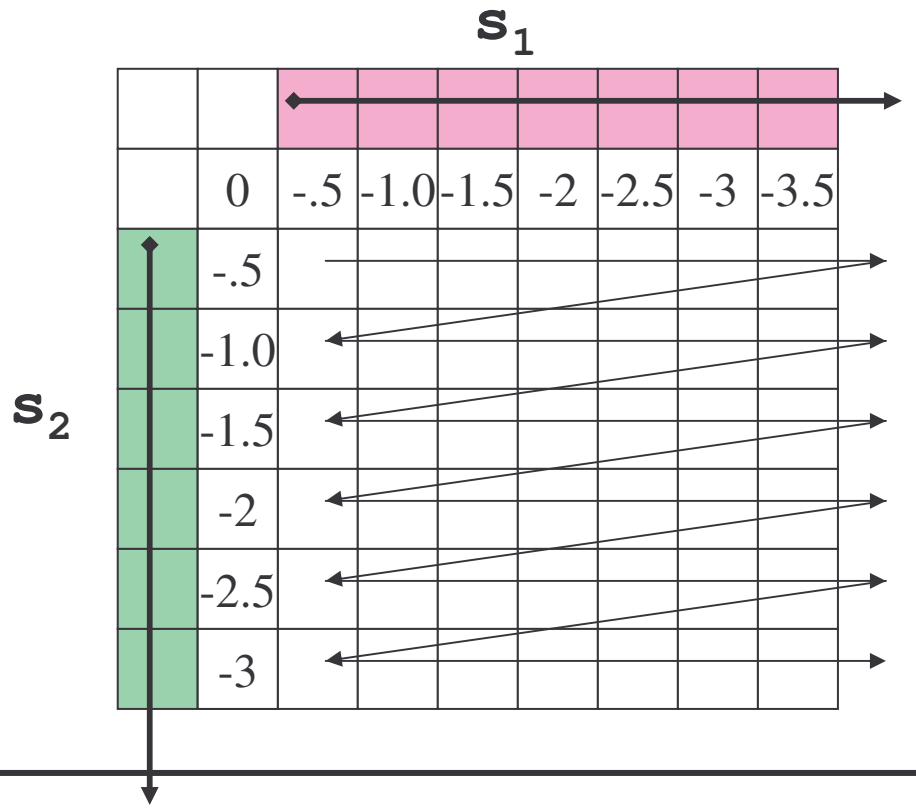


Optimal Alignment (4)

- Filling the matrix iteratively:

$$A_{ij} = \max \begin{cases} A_{i-1,j-1} + \text{score}(s_1[i], s_2[j]) \\ A_{i-1,j} + \text{penalty(insertion)} \\ A_{i,j-1} + \text{penalty(insertion)} \end{cases}$$

- We have to use a correct scanning order, e.g. the left-to-right row scanning procedure shown in the figure



Optimal Alignment (5)

- Let's fill in some values

$$A_{ij} = \max \begin{cases} A_{i-1,j-1} + \text{score}(s_1[i], s_2[j]) \\ A_{i-1,j} + \text{penalty(insertion)} \\ A_{i,j-1} + \text{penalty(insertion)} \end{cases}$$

- The optimal score can be found in the lower right corner of the table.
- The solution can be found by tracing your way backward in the matrix

–TGCTATC

AT–CTA–C

s_1

		T	G	C	T	A	T	C
	0	-.5	-1.0	-1.5	-2	-2.5	-3	-3.5
A	-.5	-1.0	-1.5	-2.0	-2.5	-1.0	-1.5	-2.0
T	-1.0	0.5	0.0	-0.5	-1.0	-1.5	0.0	-0.5
C	-1.5	0.0	-0.5	1.0	0.5	0.0	-0.5	1.0
T	-2	-0.5	-1.0	0.5	2.0	1.5	1.0	0.5
A	-2.5	-1.0	-1.5	0.0	1.5	3.0	2.5	2.0
C	-3	-1.5	-2.0	-0.5	1.0	2.5	2.0	3.5

s_2

Afternoon Lab

Part 1:

- Specify cost (match, mismatch, gap) by hand
- Read in two strings
- Compute the values of the alignment table
- Read-off the optimal score

Afternoon Lab

Part 2:

- Reconstruct an optimal alignment from the table
- Output the optimal alignment between the strings by using – for gaps

Summary

- Exact string matching can be done slow and efficient
  - Brute-force matching vs.
  - Knuth-Morris-Pratt
  - Rabin-Karp
- Inexact string matching
  - Very important, in particular in computational biology (but also in language processing)
  - Can be implemented very efficiently using dynamic programming.