6.047/6.878/HST.507
Computational Biology: Genomes, Networks, Evolution

Lecture 3
Sequence Alignment II
Database search

Global vs. Local alignment

Exact string matching and Karp-Rabin

Database search and BLAST

Deterministic linear-time string matching

Module 1: Aligning and modeling genomes

- Module 1: Computational foundations
  - Dynamic programming: exploring exponential spaces in poly-time
  - Introduce Hidden Markov Models (HMMs): Central tool in CS
  - HMM algorithms: Decoding, evaluation, parsing, likelihood, scoring
- This week: Sequence alignment / comparative genomics
  - Local/global alignment: infer nucleotide-level evolutionary events
  - Database search: scan for regions that may have common ancestry
- Next week: Modeling genomes / exon / CpG island finding
  - Modeling class of elements, recognizing members of a class
  - Application to gene finding, conservation islands, CpG islands

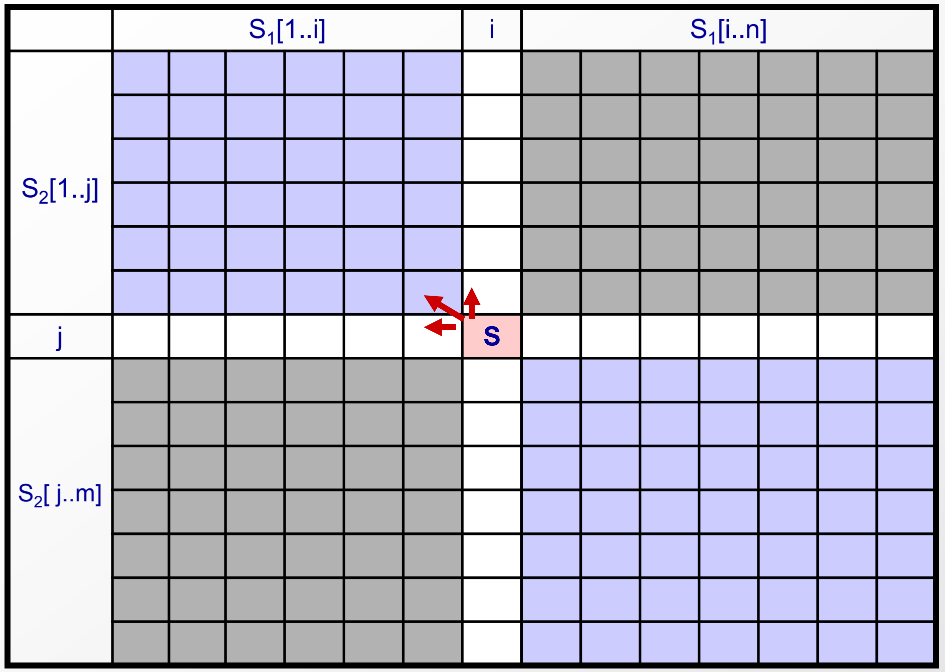
Remember Lecture 2

Sequence alignment

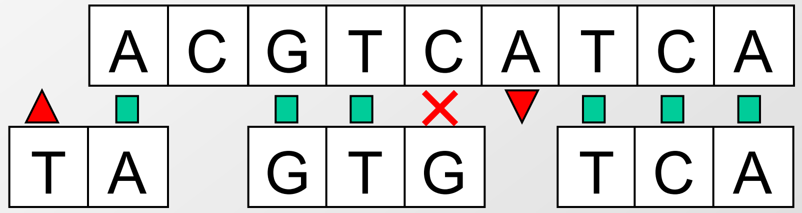
and

Dynamic programming

Duality: seq. alignment ⇔ path through the matrix

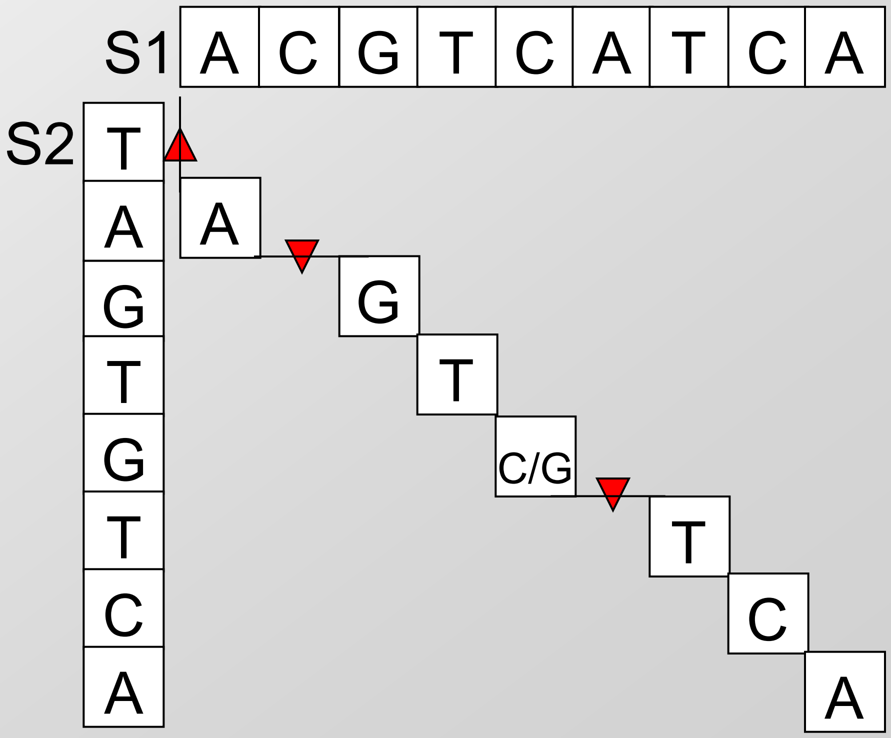


M[i,j] stores max score of prefix alignment of $S_1[1..i]$ and $S_2[1..j]$



Alignments ⇔Paths

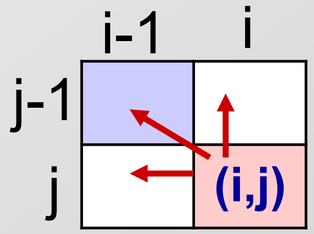
Prefix alignmt score ⇔M[i,j]



Best alignment ⇔ Best path through the matrix

Computing alignments recursively: M[i,j]=F(smaller)

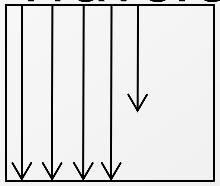
- Local update rules, only look at neighboring cells:
  - Compute next alignment based on previous alignment
  - Just like Fibonacci numbers: F[i] = F[i-1] + F[i-2]
  - Table lookup avoids repeated computation
- Computing the score of a cell from smaller neighbors
  - $$M(i,j) = \max\begin{cases} M(i-1, j) - \text{gap} \\ M(i-1, j-1) + \text{score} \\ M(i, j-1) - \text{gap} \end{cases}$$

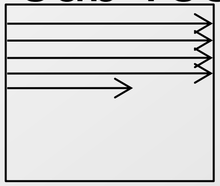


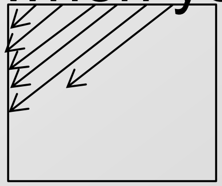
  - Only three possibilities for extending by one nucleotide: a gap in one species, a gap in the other, a (mis)match
- Compute scores for prefixes of increasing length
  - Start with prefixes of length 1, extend by one each time, until all prefixes have been computed
  - When you reach bottom right, alignment score of $S_1[1..m]$ and $S_2[1..n]$ is alignment of full S_1 and full S_2
  - (Can then trace back to construct optimal path to it)

Dynamic Programming for sequence alignment

- Setting up dynamic programming
  1. Find 'matrix' parameterization
     - Prefix parameterization. Score(S_1[1..i],S_2[1..j]) ➔ M(i,j)
     - (i,j) only prefixes vs. (i,j,k,l) all substrings ➔ simpler 2-d matrix
  2. Make sure sub-problem space is finite! (not exponential)
     - It's just n^2, quadratic (which is polynomial, not exponential)
  3. Traversal order: sub-results ready when you need them

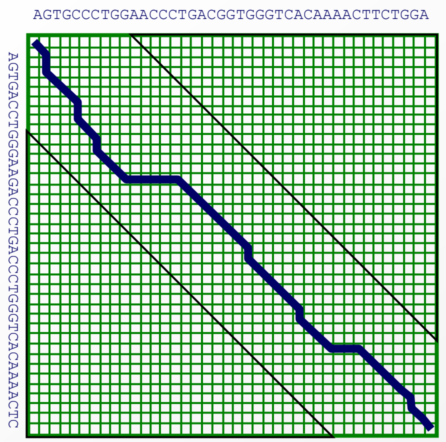

Cols L→R


Rows top→bot

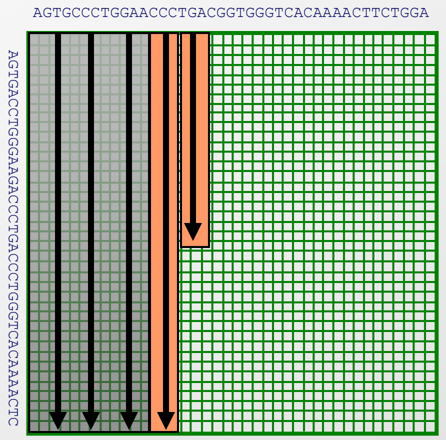

Diags topR→botL

  4. Recursion formula: larger problems = Func(subparts)
     - Need formula for computing M[i,j] as function of previous results
     - Single increment at a time, only look at M[i-1,j], M[i,j-1], M[i-1,j-1] corresponding to 3 options: gap in S_1, gap in S_2, char in both
     - Score in each case depends on gap/match/mismatch penalties
  5. Remember choice: F() typically includes min() or max()
     - Remember which of three cells (top,left,diag) led to maximum
     - Trace-back from max score to identify path leading to it

Algorithmic variations (save time and/or space)

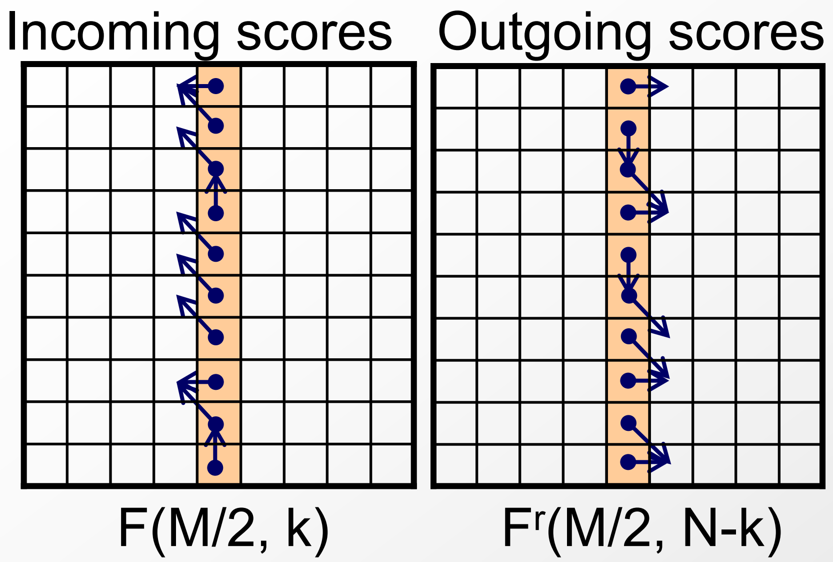

AGTGCCCTGGAACCCTGACGGTGGGTCACAAAACTTCTGGA
AGTGACCTGGGAAGACCCTGACCCTGGGTCACAAAACTC

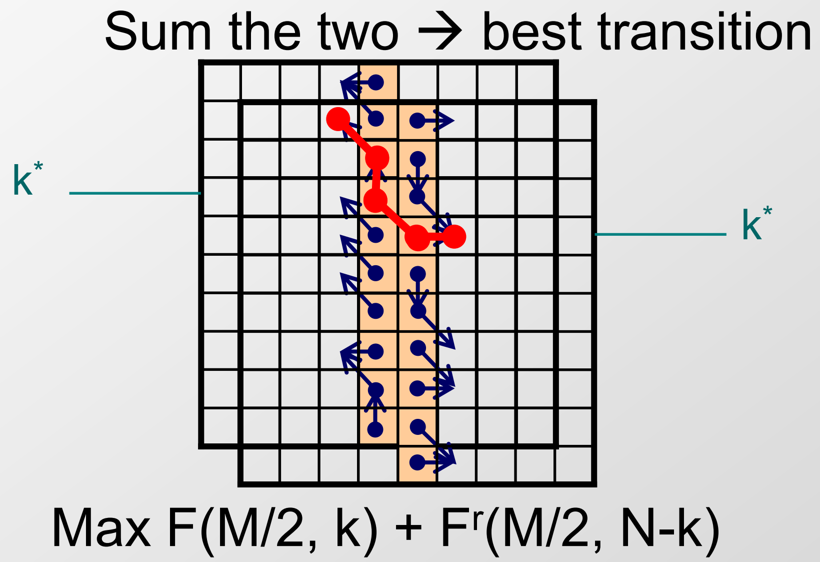
- Save time: Bounded-space computation
  - Space: O(k*m)
  - Time: O(k*m), where k = radius explored
  - Heuristic
    - Not guaranteed optimal answer
    - Works very well in practice
  - Practical interest

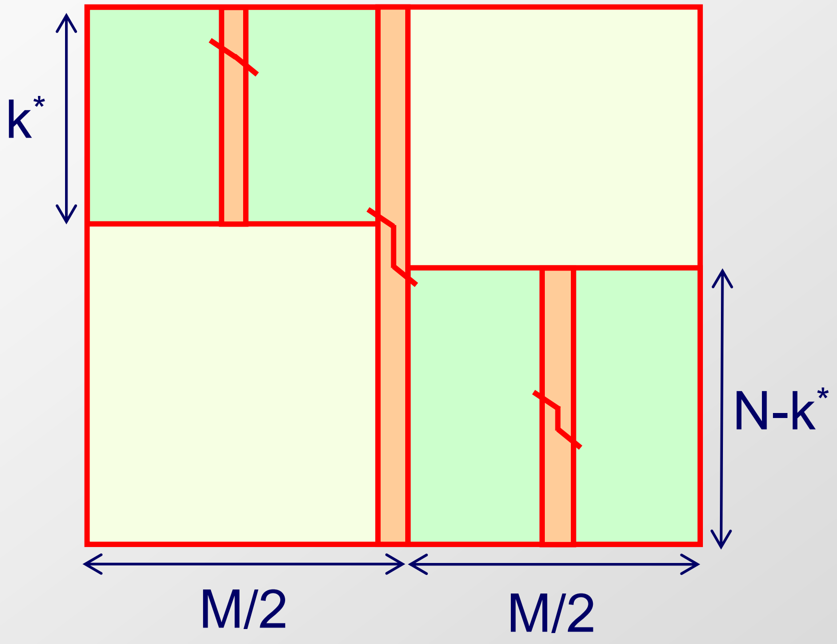

AGTGCCCTGGAACCCTGACGGTGGGTCACAAAACTTCTGGA
AGTGACCTGGGAAGACCCTGACCCTGGGTCACAAAACTC

- Save space: Linear-space computation
  - Save only one col / row / diag at a time
  - Computes optimal score easily
  - Theoretical interest
    - Effective running time slower
    - Optimal answer guaranteed
  - Recursive call modification allows traceback

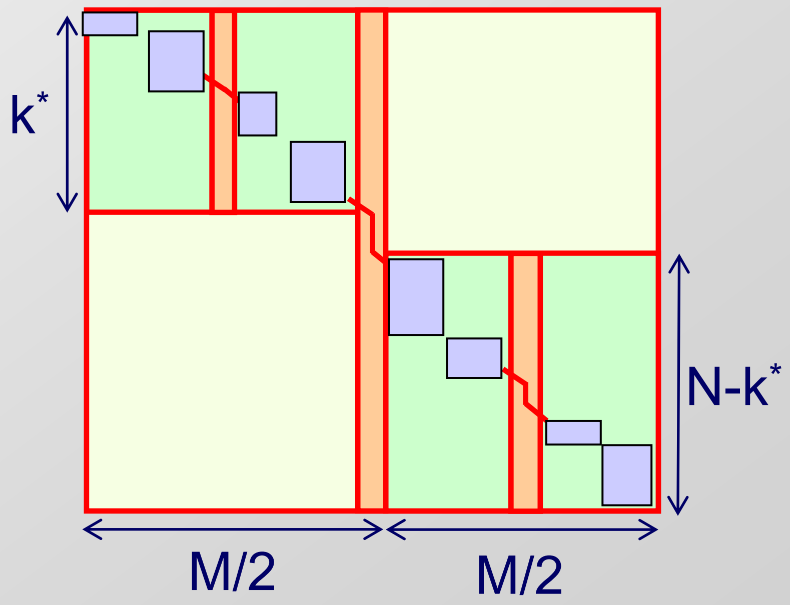
Finding optimal path using only linear space







Iterate procedure in corner quadrants



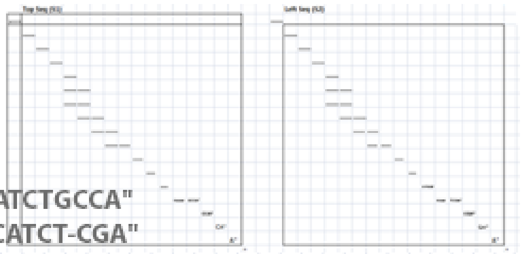
Total cost $MN(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots) \leq 2MN$

Genome alignment in an excel spreadsheet

	A	G	C	T	Gap
A	1.0	0.0	-1.0	-1.0	-1.0
G	0.0	1.0	-1.0	-1.0	
C	-1.0	-1.0	1.0	0.0	
T	-1.0	-1.0	0.0	1.0	

S1[1..i]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Prefixes	T	TA	TAA	TAAC	TAACC	TAACCT	TAACCTT	TAACCTTT	TAACCTTTA	TAACCTTTAT	TAACCTTTATC	TAACCTTTATCT	TAACCTTTATCTG	TAACCTTTATCTGC	TAACCTTTATCTGCC	TAACCTTTATCTGCCA

S2[1..j]	Prefixes	S2:" \ S1:"	T	A	A	C	C	T	T	T	A	T	C	T	G	C	C	A
1	T	T	1.0	-1.0	-1.0	0.0	0.0	1.0	1.0	1.0	-1.0	1.0	0.0	1.0	-1.0	0.0	0.0	-1.0
2	TA	A	-1.0	1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	1.0
3	TAA	A	-1.0	1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	1.0
4	TAAC	C	0.0	-1.0	-1.0	1.0	1.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	-1.0	1.0	1.0	-1.0
5	TAACG	G	-1.0	0.0	0.0	-1.0	-1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	0.0
6	TAACGG	G	-1.0	0.0	0.0	-1.0	-1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	0.0
7	TAACGGC	C	0.0	-1.0	-1.0	1.0	1.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	-1.0	1.0	1.0	-1.0
8	TAACGGCC	C	0.0	-1.0	-1.0	1.0	1.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	-1.0	1.0	1.0	-1.0
9	TAACGGCCC	C	0.0	-1.0	-1.0	1.0	1.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	-1.0	1.0	1.0	-1.0
10	TAACGGCCCA	A	-1.0	1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	1.0
11	TAACGGCCCAT	T	1.0	-1.0	-1.0	0.0	0.0	1.0	1.0	1.0	-1.0	1.0	0.0	1.0	-1.0	0.0	0.0	-1.0
12	TAACGGCCCATC	C	0.0	-1.0	-1.0	1.0	1.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	-1.0	1.0	1.0	-1.0
13	TAACGGCCCATCT	T	1.0	-1.0	-1.0	0.0	0.0	1.0	1.0	1.0	-1.0	1.0	0.0	1.0	-1.0	0.0	0.0	-1.0
14	TAACGGCCCATCTC	C	0.0	-1.0	-1.0	1.0	1.0	0.0	0.0	0.0	-1.0	0.0	1.0	0.0	-1.0	1.0	1.0	-1.0
15	TAACGGCCCATCTCG	G	-1.0	0.0	0.0	-1.0	-1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	0.0
16	TAACGGCCCATCTCGA	A	-1.0	1.0	1.0	-1.0	-1.0	-1.0	-1.0	-1.0	1.0	-1.0	-1.0	-1.0	0.0	-1.0	-1.0	1.0



S1:"TAAC-CTTTATCTGCCA"
S2:"TAACGGCCCATCT-CGA"

		T	A	A	C	C	T	T	T	A	T	C	T	G	C	C	A
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16
T	-1	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14
A	-2	0	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12
A	-3	-1	1	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
C	-4	-2	0	2	4	3	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8
G	-5	-3	-1	1	3	3	2	1	0	0	-1	-2	-3	-3	-4	-5	-6
G	-6	-4	-2	0	2	2	2	1	0	0	-1	-2	-3	-2	-3	-4	-5
C	-7	-5	-3	-1	1	3	2	2	1	0	0	0	-1	-2	-1	-2	-3
C	-8	-6	-4	-2	0	2	3	2	2	1	0	1	0	-1	-1	0	-1
C	-9	-7	-5	-3	-1	1	2	3	2	1	1	1	1	0	0	0	-1
A	-10	-8	-6	-4	-2	0	1	2	2	3	2	1	0	1	0	-1	1
T	-11	-9	-7	-5	-3	-1	1	2	3	2	4	3	2	1	1	0	0
C	-12	-10	-8	-6	-4	-2	0	1	2	2	3	5	4	3	2	2	1
T	-13	-11	-9	-7	-5	-3	-1	1	2	1	3	4	6	5	4	3	2
C	-14	-12	-10	-8	-6	-4	-2	0	1	1	2	4	5	5	6	5	4
G	-15	-13	-11	-9	-7	-5	-3	-1	0	1	1	3	4	6	5	5	5
A	-16	-14	-12	-10	-8	-6	-4	-2	-1	1	0	2	3	5	5	4	6

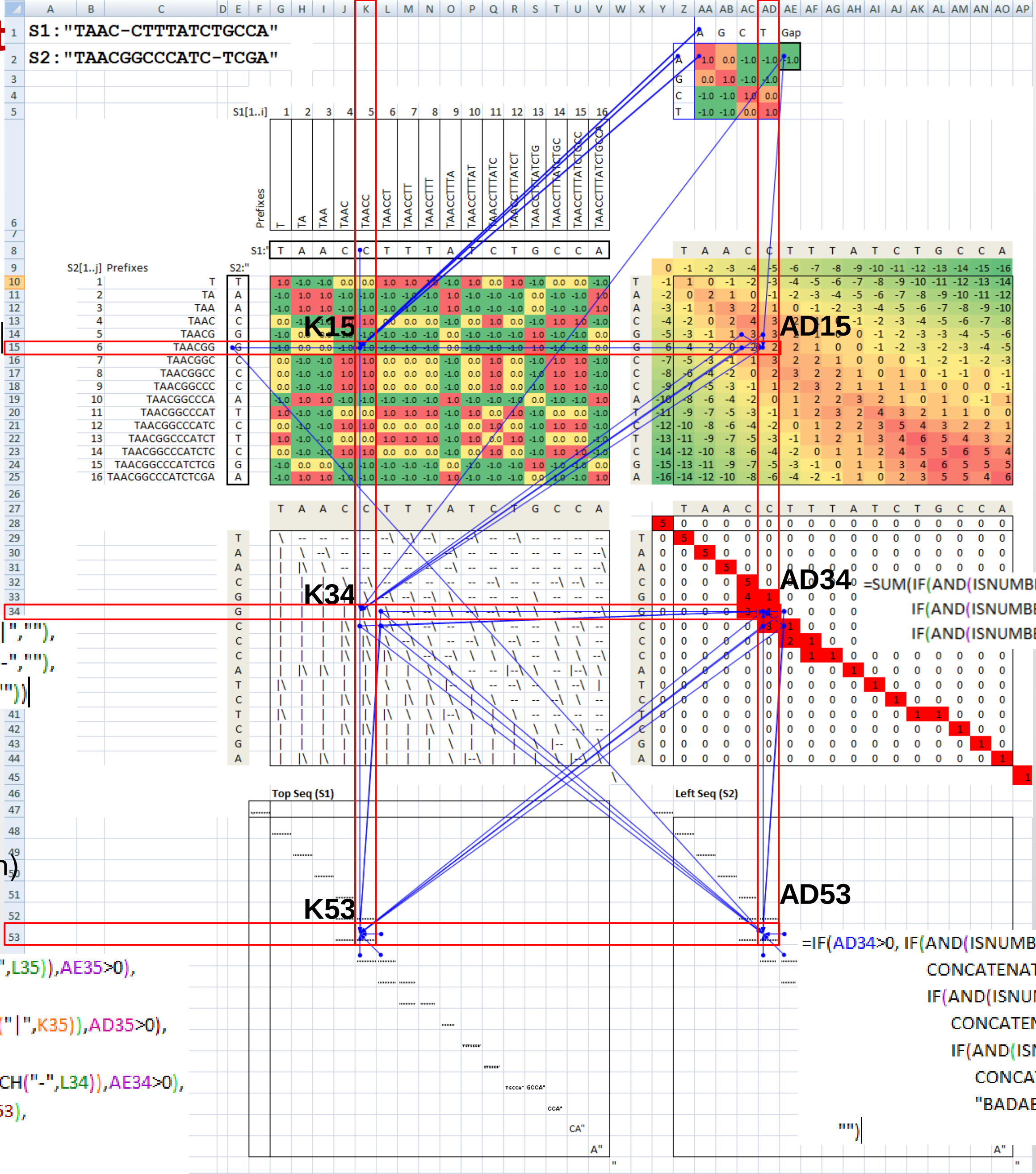
	T	A	A	C	C	T	T	T	A	T	C	T	G	C	C	A
T	\	--	--	--	--	--\	--\	--\	--	--\	--	--\	--	--	--	--
A	|	\	--\	--	--	--	--	--	--\	--	--	--	--	--	--	--\
A	|	|\	\	--	--	--	--	--	--\	--	--	--	--	--	--	--\
C	|	|	|	\	--\	--	--	--	--	--	--\	--	--	--\	--\	--
G	|	|	|	|	\	--\	--\	--\	\	--	--	--	\	--	--	--
G	|	|	|	|	|\	\	--\	--\	\	--\	--\	--\	\	--	--	--\
C	|	|	|	|\	\	--\	\	--\	--	\	\	--	--	\	--\	--
C	|	|	|	|\	|\	\	--\	\	--	--\	\	--\	--	\	\	--
C	|	|	|	|\	|\	|\	\	--\	--\	\	\	\	--	\	\	--\
A	|	|\	|\	|	|	|	|	\	\	--	--	|--\	\	--	|--\	\
T	|\	|	|	|	|	\	\	\	|--	\	--	--\	--	\	--\	|
C	|	|	|	|\	|\	|	|\	|\	\	|	\	--	--	--\	\	--
T	|\	|	|	|	|	|\	\	\	|--\	\	|	\	--	--	--	--
C	|	|	|	|\	|\	|	|	|\	\	|	\	|	\	\	--\	--
G	|	|	|	|	|	|	|	|	\	|	|	|	\	|--	\	\
A	|	|\	|\	|	|	|	|	|	\	|--\	|	|	|	\	|--\	\

\

		T	A	A	C	C	T	T	T	A	T	C	T	G	C	C	A
	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
T	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	4	1	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	3	1	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	3	1	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
T	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
T	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

9

Genome alignment in an excel spreadsheet



S1:"TAAC-CTTTATCTGCCA"
S2:"TAACGGCCCATC-TCGA"

	A	G	C	T	Gap
A	1.0	0.0	-1.0	-1.0	-1.0
G	0.0	1.0	-1.0	-1.0	
C	-1.0	-1.0	1.0	0.0	
T	-1.0	-1.0	0.0	1.0	

K15

```
=INDEX($AA$2:$AD$5,
        MATCH(K$8,$Z$2:$Z$5,0),
        MATCH($E15,$AA$1:$AD$1,0))
```

Local score of matching characters $S_1[i]$ and $S_2[j]$

AD15

```
=MAX(AD14+$AE$2,
     AC15+$AE$2,
     AC14+K15)
```

Max alignment score of aligning prefix $S_1[1..i]$ and prefix $S_2[1..j]$

K34

```
=CONCATENATE(IF(AD15=AD14+$AE$2,"|",""),
             IF(AD15=AC15+$AE$2,"--",""),
             IF(AD15=AC14+K15,"\",""))
```

Is the max alignment score coming from the top ("|"), from the left ("--") or from the diagonal up ("\")
(show all of them, cuz we can)

AD34

```
=SUM(IF(AND(ISNUMBER(SEARCH("|",K35)),AD35>0),AD35,0),
     IF(AND(ISNUMBER(SEARCH("\",L35)),AE35>0),AE35,0),
     IF(AND(ISNUMBER(SEARCH("-",L34)),AE34>0),AE34,0))
```

Is the [i,j] part of an optimal path? (i.e. are chars $S_1[i]$ and $S_2[j]$ aligned to each other in an optimal path)
(also count number of optimal paths/alignment through [i.j], cuz we can)

K53

```
=IF(AD34>0, IF(AND(ISNUMBER(SEARCH("\",L35)),AE35>0),
        CONCATENATE(K$8,L54),
        IF(AND(ISNUMBER(SEARCH("|",K35)),AD35>0),
          CONCATENATE("-",K54),
          IF(AND(ISNUMBER(SEARCH("-",L34)),AE34>0),
            CONCATENATE(K$8,L53),
            "BADABOOM!"))),
  "")
```

Construct the optimal alignment for sequence S_1 by adding in characters or gaps to increasingly large suffixes (and arbitrarily choose one path when multiple using nested if's)

AD53

```
=IF(AD34>0, IF(AND(ISNUMBER(SEARCH("\",L35)),AE35>0),
        CONCATENATE($E15,AE54),
        IF(AND(ISNUMBER(SEARCH("|",K35)),AD35>0),
          CONCATENATE($E15,AD54),
          IF(AND(ISNUMBER(SEARCH("-",L34)),AE34>0),
            CONCATENATE("-",AE53),
            "BADABOOM!"))),
  "")
```

Construct the optimal alignment for sequence S_2 similarly to S_1

Today's Goal: Diving deeper into alignments

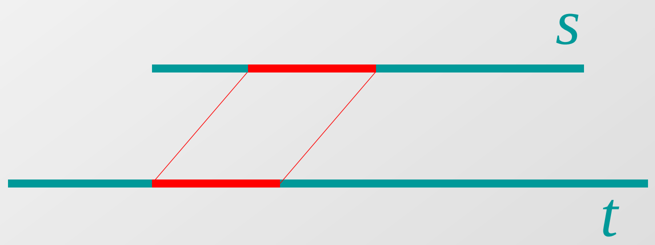
1. **Global alignment vs. Local alignment**
   - Variations on initialization, termination, update rule
   - Varying gap penalties, algorithmic speedups
2. Linear-time exact string matching (expected)
   - Karp-Rabin algorithm and semi-numerical methods
   - Hash functions and randomized algorithms
3. The BLAST algorithm and inexact matching
   - Hashing with neighborhood search
   - Two-hit blast and hashing with combs
4. Deterministic linear-time exact string matching
   - Key insight: gather more info from each comparison
   - Pre-processing, Z-algorithm, Boyer-More, KMP

Today's Goal: Diving deeper into alignments

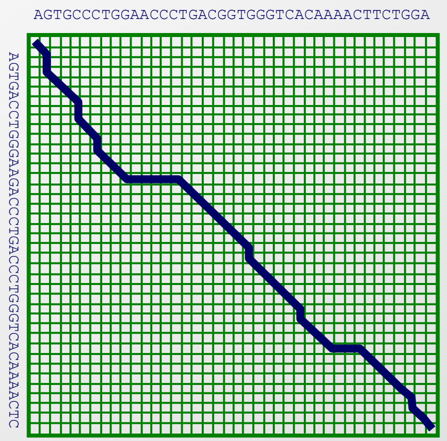
1. **Global alignment vs. Local alignment**
   - Variations on initialization, termination, update rule
   - Varying gap penalties, algorithmic speedups
2. **Linear-time exact string matching (expected)**
   - Karp-Rabin algorithm and semi-numerical methods
   - Hash functions and randomized algorithms
3. **The BLAST algorithm and inexact matching**
   - Hashing with neighborhood search
   - Two-hit blast and hashing with combs
4. **Probabilistic foundations of sequence alignment**
   - Mismatch penalties, BLOSUM and PAM matrices
   - Statistical significance of an alignment score

Intro to Local Alignments

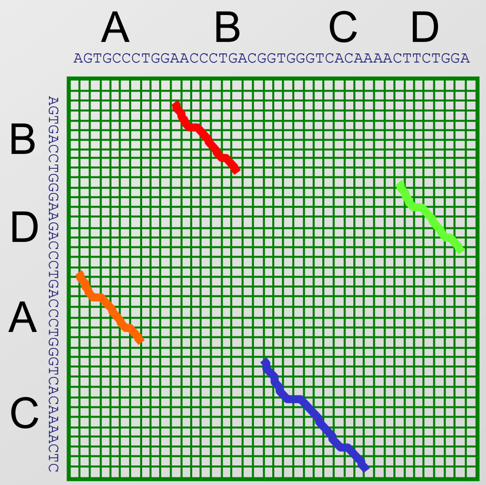
- Statement of the problem
  - A *local alignment* of strings *s* and *t* is an alignment of a substring of *s* with a substring of *t*



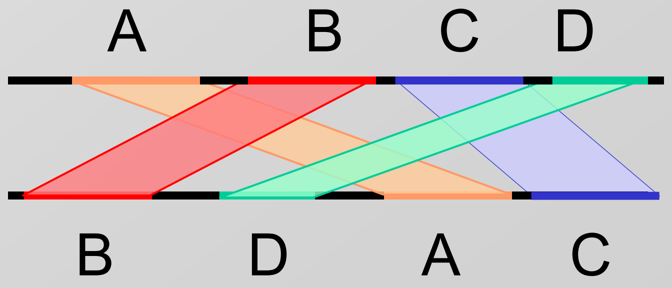
- Why local alignments?
  - Small domains of a gene may be only conserved portions
  - Looking for a small gene in a large chromosome (search)
  - Large segments often undergo rearrangements


AGTGCCCTGGAACCCTGACGGTGGGTCACAAAACTTCTGGA
AGTGACCTGGGAAGACCCTGACCCTGGGTCACAAAACTC

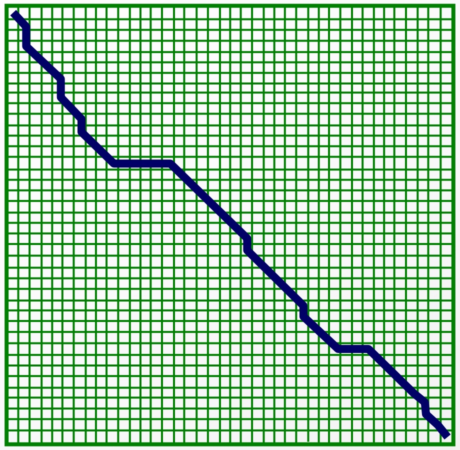
Global alignment

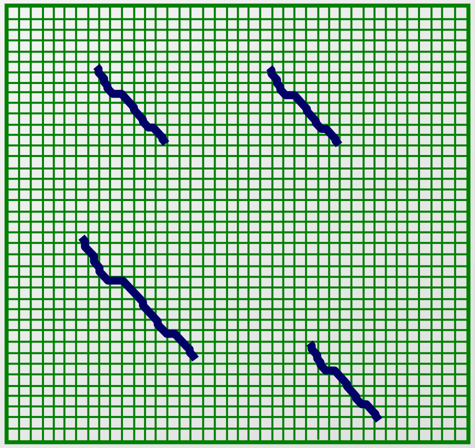

A
B
C
D
AGTGCCCTGGAACCCTGACGGTGGGTCACAAAACTTCTGGA
B
D
A
C
AGTGACCTGGGAAGACCCTGACCCTGGGTCACAAAACTC

Local alignment



Global Alignment vs. Local alignment





Needleman-Wunsch algorithm

Initialization: $F(0, 0) = 0$

Iteration:

$$F(i, j) = \max \begin{cases} F(i-1, j) - d \\ F(i, j-1) - d \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$$

Termination: Bottom right

Smith-Waterman algorithm

Initialization: $F(0, j) = F(i, 0) = 0$

Iteration:

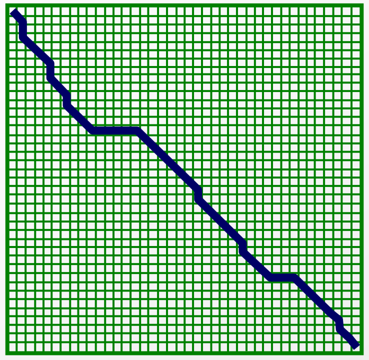
$$F(i, j) = \max \begin{cases} 0 \\ F(i-1, j) - d \\ F(i, j-1) - d \\ F(i-1, j-1) + s(x_i, y_j) \end{cases}$$

Termination: **Anywhere**

More variations on the theme: semi-global alignment

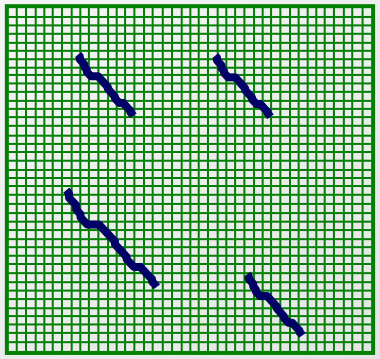
- Sequence alignment variations

Global



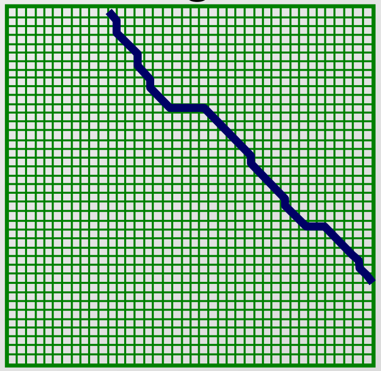
Complete alignment

Local



Stretches of similarity

Semi-global

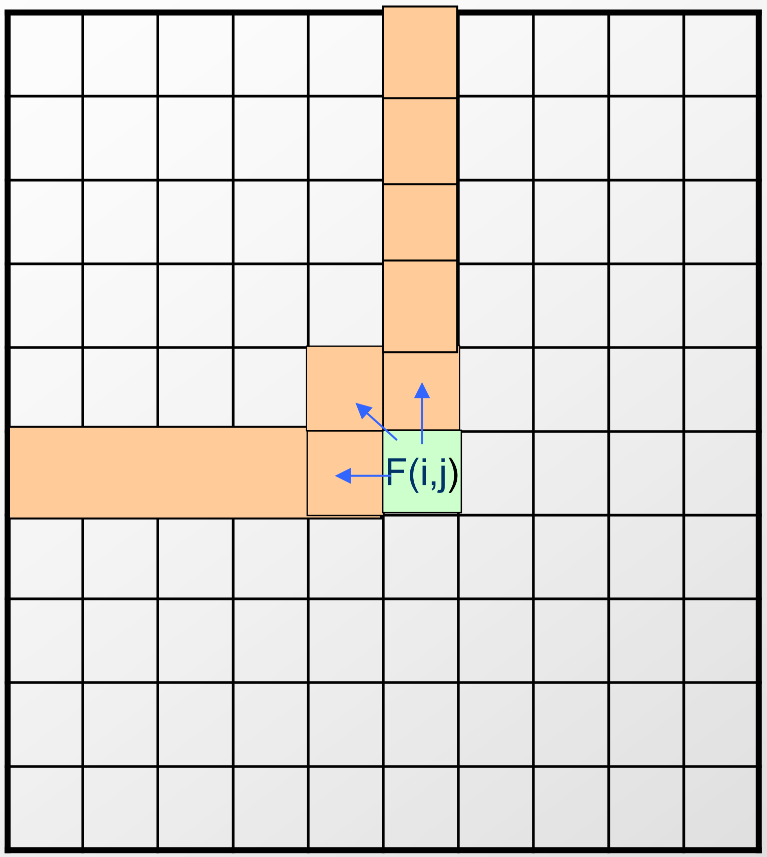


No end-gap penalty

Initialization	Top left	Top row/left col.	Top row or left column
Iteration:max	$F(i-1, j) - d$ $F(i, j-1) - d$ $F(i-1, j-1) + s(x_i, y_j)$	0 $F(i-1, j) - d$ $F(i, j-1) - d$ $F(i-1, j-1) + s(x_i, y_j)$	$F(i-1, j) - d$ $F(i, j-1) - d$ $F(i-1, j-1) + s(x_i, y_j)$
Termination	Bottom right	Anywhere	Bottom row or right column

Sequence alignment with generalized gap penalties

- **Implementing a generalized gap penalty function F(gap_length)**



Initialization: same

Iteration:

$$F(i, j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) \\ \max_{k=0\ldots i-1} F(k,j) - \gamma(i-k) \\ \max_{k=0\ldots j-1} F(i,k) - \gamma(j-k) \end{cases}$$

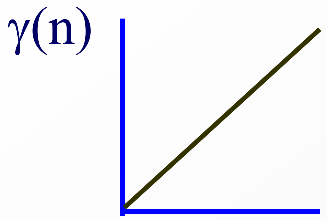
Termination: same

Running Time: $O(N^2M)$ (cubic)

Space: $O(NM)$

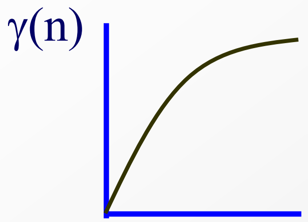
Do we have to be so general?

Algorithmic trade-offs of varying gap penalty functions



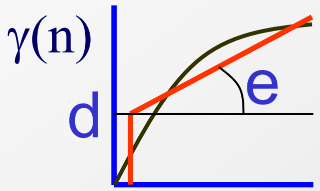
Linear gap penalty: w(k) = k*p

- State: Current index tells if in a gap or not
- Achievable using quadratic algorithm (even w/ linear space)



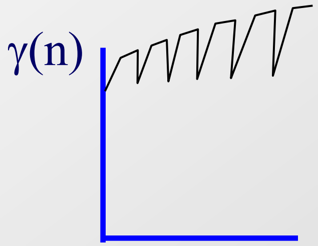
Quadratic: w(k) = p+q*k+rk^2.

- State: needs to encode the length of the gap, which can be O(n)
- To encode it we need O(log n) bits of information. Not feasible



Affine gap penalty: w(k) = p + q*k, where q<p

- State: add binary value for each sequence: starting a gap or not
- Implementation: add second matrix for already-in-gap (recitation)



Length (mod 3) gap penalty for protein-coding regions

- Gaps of length divisible by 3 are penalized less: conserve frame
- This is feasible, but requires more possible states
- Possible states are: starting, mod 3=1, mod 3=2, mod 3=0

Today's Goal: Diving deeper into alignments

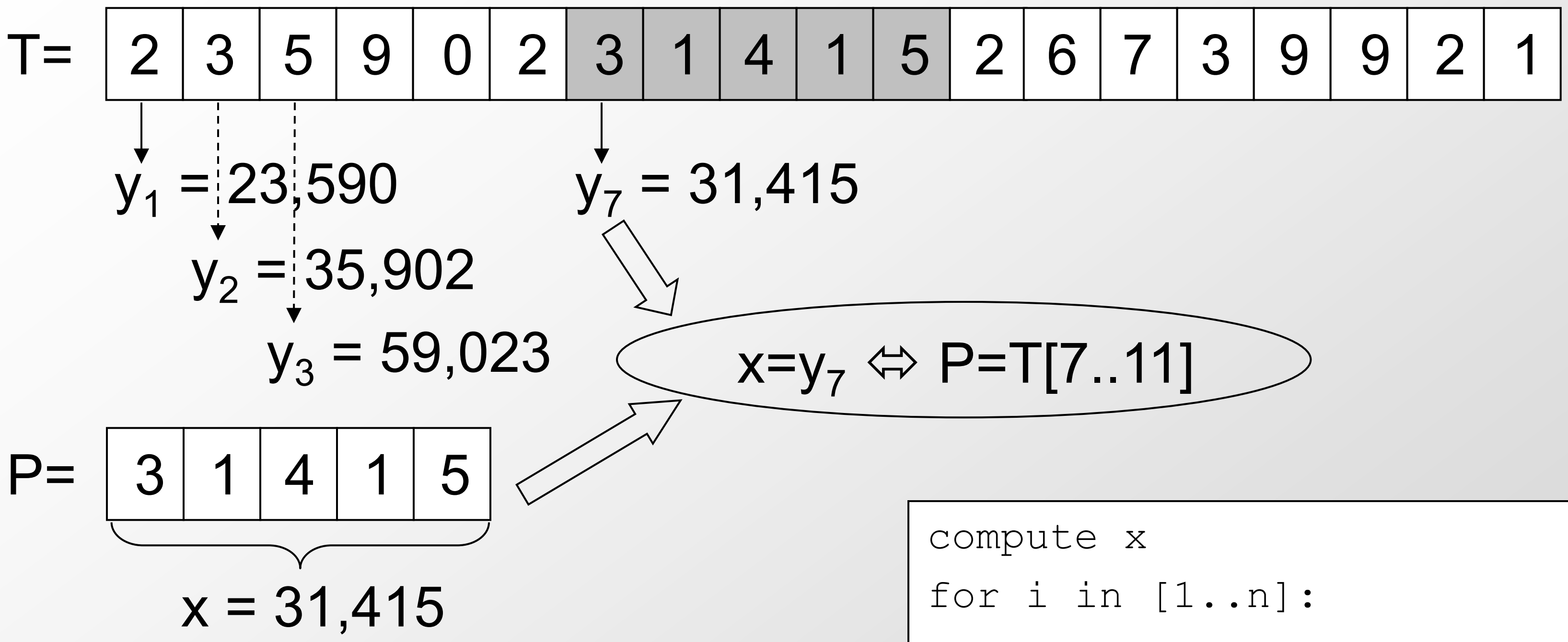
1. Global alignment vs. Local alignment
   - Variations on initialization, termination, update rule
   - Varying gap penalties, algorithmic speedups
2. **Linear-time exact string matching (expected)**
   - Karp-Rabin algorithm and semi-numerical methods
   - Hash functions and randomized algorithms
3. The BLAST algorithm and inexact matching
   - Hashing with neighborhood search
   - Two-hit blast and hashing with combs
4. Deterministic linear-time exact string matching
   - Key insight: gather more info from each comparison
   - Pre-processing, Z-algorithm, Boyer-More, KMP

Linear-time string matching

- When looking for exact matches of a pattern (no gaps)

- Karp-Rabin algorithm (probabilistic linear time):
  - Interpret String numerically
  - Start with ‘broken’ version of the algorithm
  - Progressively fix it to make it work
- Deterministicc linear-time solutions exist (not this term):
  - Z-algorithm / fundamental pre-processing, Gusfield
  - Boyer-Moore and Knuth-Morris-Pratt algorithms are earliest instantiations, similar in spirit
  - Suffix trees: beautiful algorithms, many different variations and applications, limited use in CompBio
  - Suffix arrays: practical variation, Gene Myers

Karp-Rabin algorithm

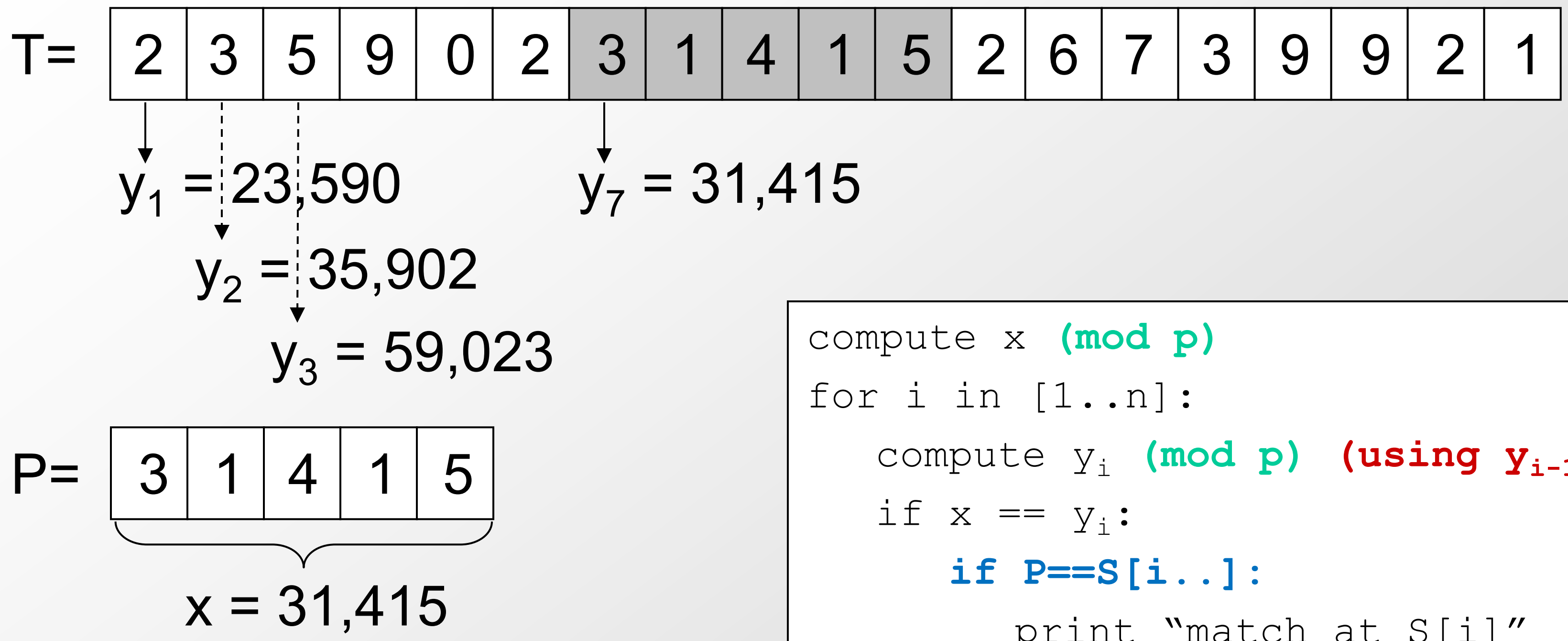


```
compute x
for i in [1..n]:
    compute y_i
    if x == y_i:
          print "match at S[i]"
```

(this does not actually work)

- Key idea:
  - Interpret strings as numbers: fast comparison

Karp-Rabin algorithm

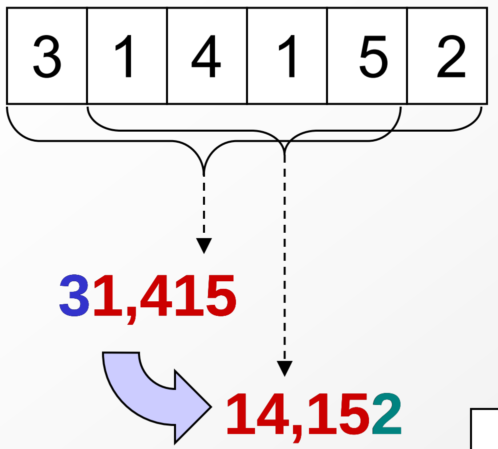


```
compute x (mod p)
for i in [1..n]:
   compute y_i (mod p) (using y_{i-1})
   if x == y_i:
      if P==S[i..]:
         print "match at S[i]"
      else:
         (spurious hit)
```

(this actually works)

- Key idea:
  - Interpret strings as numbers: fast comparison
- To make it work:
  - **(a) Compute next number based on previous one → O(1)**
  - **(b) Hashing (mod p) → keep the numbers small → O(1)**
  - **(c) Deal with spurious hits due to hashing collisions**

(a) Computing t_{s+1} based on t_s in constant time



old high-order bit

left shift

new low-order digit

14,152 = (31,415 - 3 * 10,000) * 10 + 2

14,152 =? function (31,415)

- Middle digits of the number are already computed

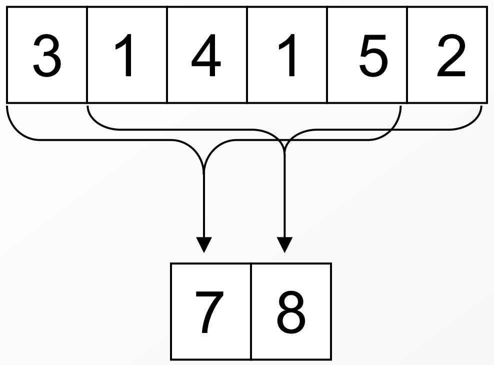
  Shift them to the left 🡐
- Remove the high-order bit
- Add the low-order bit

- General case:

$$t_s = \underline{T[s+1]2^{m-1}} + T[s+2]2^{m-2} + \ldots + T[s+m]2^0$$

$$t_{s+1} = T[s+2]2^{m-1} + T[s+3]2^{m-2} + \ldots + \underline{T[s+m+1]2^0}$$

(b) Dealing with long numbers in constant time



old high-order bit, left shift, new low-order digit

$$\mathbf{14{,}152} = (\mathbf{31{,}415} - 3 * \mathbf{10{,}000}) * 10 + 2 \ \mathbf{(mod\ 13)}$$
$$= (\mathbf{7}-3*\mathbf{3})*10+2 \ \mathbf{(mod\ 13)}$$
$$= \mathbf{8} \ \mathbf{(mod\ 13)}$$

Problem:

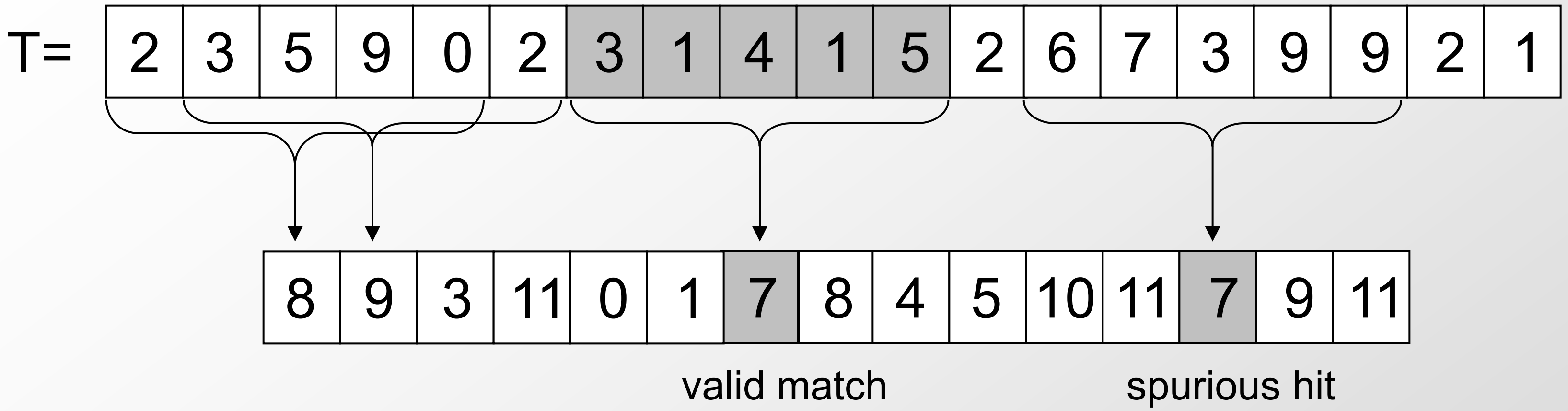
- To get O(n) time, need to perform each operation in O(1) time
- But if arguments are m-bit long (2^m range), can take O(m) time
- Need to reduce number range to something more manageable

Solution:

- Hashing: Mapping keys ***k*** from large universe U (of strings/numbers) into the '**hash**' of each key ***h(k)***, in smaller space [1..m]
- Many hash functions possible, w/ theoretical & practical properties:
  - Reproducibility: x=y➔h(x)=h(y) (hash of x always the same)
  - Uniform output distrib: x≠y➔P(h(x)=h(y))=1/m, for any input dist

New problem: Collisions

(c) Dealing with collisions, due to hashing



- Consequences of (mod p) 'hashing'
  - Good: Enable fast computation (use small numbers)
  - Bad: Leads to spurious hits (collisions)
- Dealing with the bad:
  1. Verify that a <u>**hit**</u> correspond to valid <u>**match**</u>

     → re-compute equality for entire string (not just hash)
  2. Avoid worst-case behavior of many collisions w/ bad m

     → Choose **random m**
- Algorithm and its analysis becomes more complex:
  1. Compute <u>expected run time</u>, include expected <u>cost of verification</u>
  2. Show probability of spurious hits is small, expected run time is <u>linear</u>

Karp-Rabin algorithm: Putting it all together

T=

2	3	5	9	0	2	3	1	4	1	5	2	6	7	3	9	9	2	1

$y_1 = 23{,}590$ $\quad y_7 = 31{,}415$

$y_2 = 35{,}902$

$y_3 = 59{,}023$

P=

3	1	4	1	5

$x = 31{,}415$

```
compute x (mod p)
for i in [1..n]:
    compute $y_i$ (mod p)  (using $y_{i-1}$)
    if x == $y_i$:
        if P==S[i..]:
            print "match at S[i]"
        else:
            (spurious hit)
```

(this actually works)

- Key idea: Semi-numerical computation
  - **Idea: Interpret strings as numbers => fast comparison**
    (other semi-numerical methods: Fast Fourier Transform, Shift-And)
- To make it work:
  (a) Compute next number based on previous one → O(1)
  (b) Hashing (mod p) → keep the numbers small → O(1)
  (c) Dealing with collisions → Randomized p, expected run time → O(1) exp

Today's Goal: Diving deeper into alignments

1. Global alignment vs. Local alignment
   - Needleman-Wunsch and Smith-Waterman
   - Varying gap penalties and algorithmic speedups
2. Linear-time exact string matching (expected)
   - Karp-Rabin algorithm and semi-numerical methods
   - Hash functions and randomized algorithms
3. **The BLAST algorithm and inexact matching**
   - Hashing with neighborhood search
   - Two-hit blast and hashing with combs
4. Deterministic linear-time exact string matching
   - Key insight: gather more info from each comparison
   - Pre-processing, Z-algorithm, Boyer-More, KMP

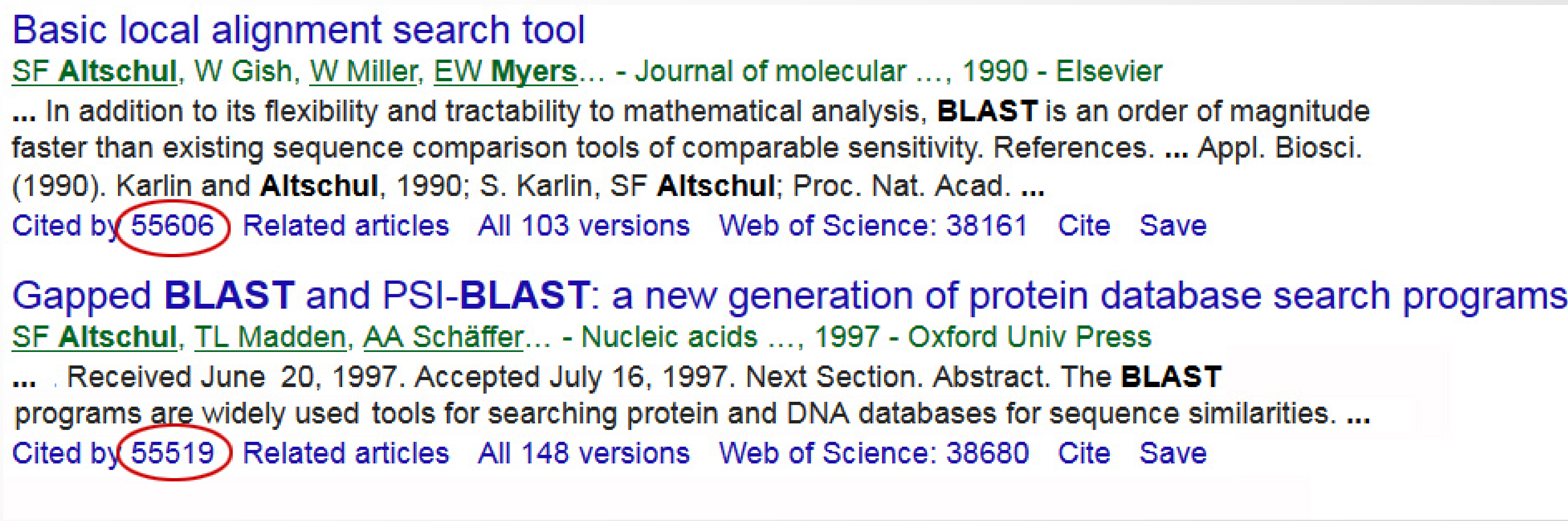
Sequence Alignment vs. Sequence Database Search

- Sequence Alignment
  - Assumes sequences have some common ancestry
  - Finding the “right” alignment between two sequences
  - Evolutionary interpretation: min # events, min cost
- Sequence Database Search
  - Given a query (new seq), and target (many old seqs), ask: which sequences (if any) are related to the query
  - Individual alignments need not be perfect: Once initial matches are reported, we can fine-tune them later
  - Query must be very fast for a new sequence
  - Most sequences will be completely unrelated to query
- Exploit distinct nature of database search problem

Speeding up your searches in dB setting

- Exploit nature of the problem (many spurious hits)
  - If you're going to reject any match with idperc <= 90, then why bother even looking at sequences which don't have a stretch of 10 nucleotides in a row.
  - Pre-screen sequences for common long stretches
- Put the speed where you need it (pre-processing)
  - Pre-processing the database is off-line.
  - Once the query arrives, must act fast
- Solution: content-based indexing and BLAST
  - Example: index 10-mers.
  - Only one 10-mer in 4^{10} will match, one in a million (even with 500 k-mers, only 1 in 2000 will match).
  - Additional speedups are possible

BLAST



Basic local alignment search tool

SF **Altschul**, W Gish, W Miller, EW **Myers**... - Journal of molecular ..., 1990 - Elsevier

... In addition to its flexibility and tractability to mathematical analysis, **BLAST** is an order of magnitude faster than existing sequence comparison tools of comparable sensitivity. References. ... Appl. Biosci. (1990). Karlin and **Altschul**, 1990; S. Karlin, SF **Altschul**; Proc. Nat. Acad. ...

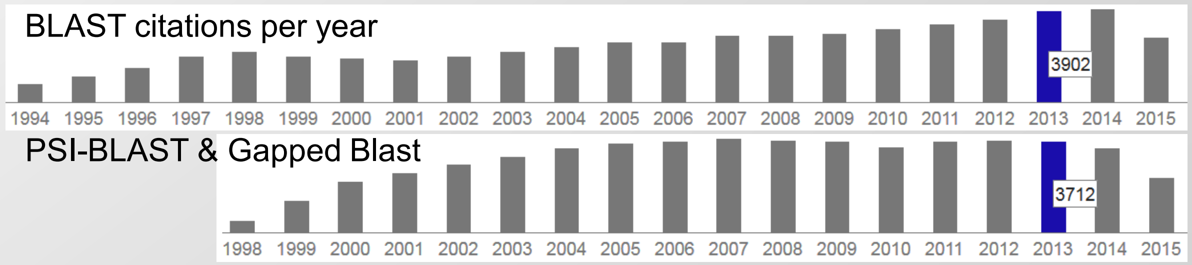
Cited by 55606 Related articles All 103 versions Web of Science: 38161 Cite Save

Gapped **BLAST** and PSI-**BLAST**: a new generation of protein database search programs

SF **Altschul**, TL Madden, AA Schäffer... - Nucleic acids ..., 1997 - Oxford Univ Press

... Received June 20, 1997. Accepted July 16, 1997. Next Section. Abstract. The **BLAST** programs are widely used tools for searching protein and DNA databases for sequence similarities. ...

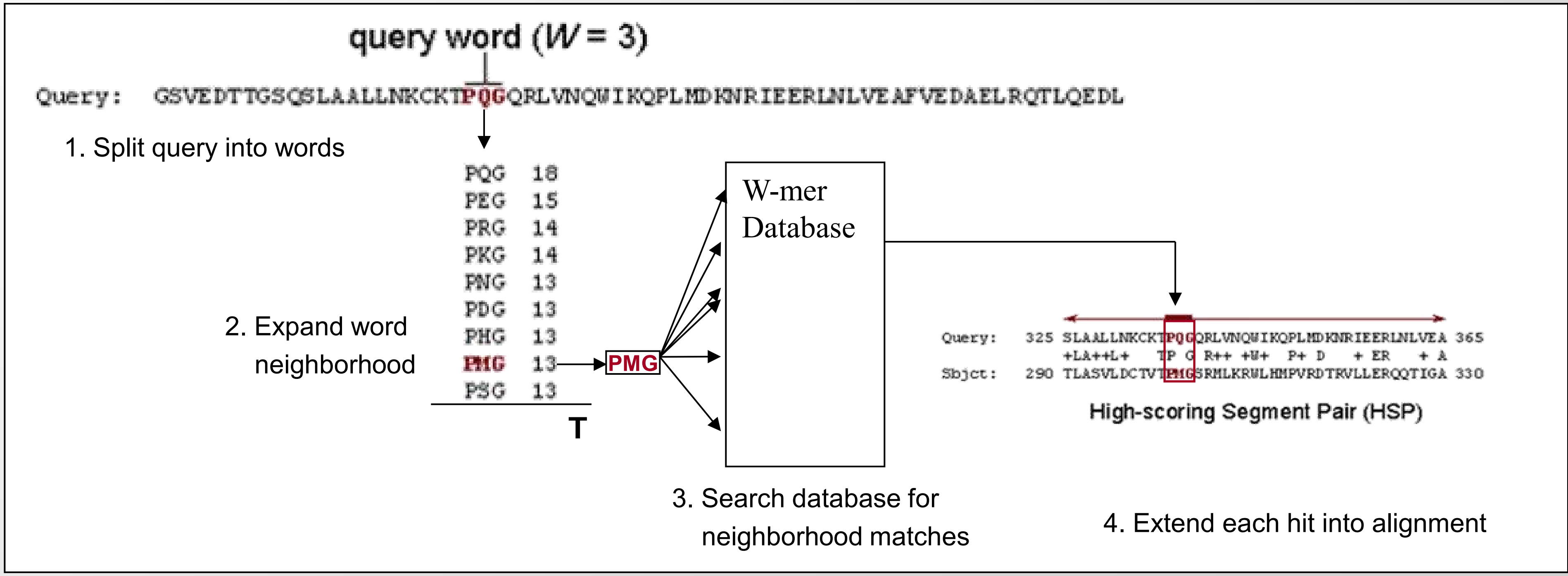
Cited by 55519 Related articles All 148 versions Web of Science: 38680 Cite Save



- Two key insights:
- Hashing:
  - Like Karp-Rabin, semi-numerical string matching
- Neighborhood search:
  - Can find hits even when no exact k-mer matches

Blast Algorithm Overview

- Receive query
  1. Split query into overlapping words of length W
  2. Find neighborhood words for each word until threshold T
  3. Look in table where these neighbor words occur: seeds S
  4. Extend seeds S until score drops off under X
- Report significance and alignment of each match



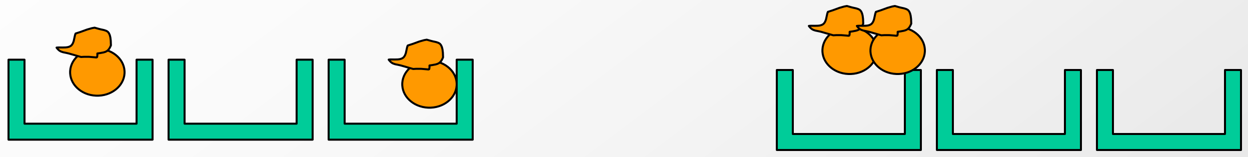
```
Query:  GSVEDTTGSQSLAALLNKCKTPQGQRLVNQWIKQPLMDKNRIEERLNLVEAFVEDAELRQTLQEDL
```

PQG	18
PEG	15
PRG	14
PKG	14
PNG	13
PDG	13
PHG	13
PMG	13
PSG	13

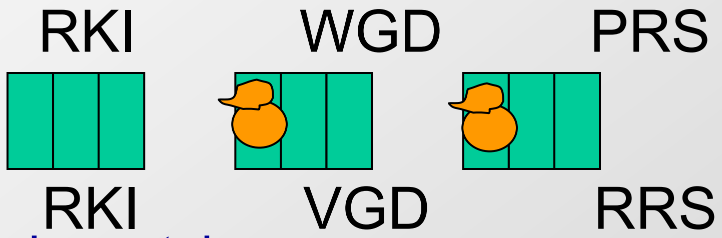
```
Query:  325 SLAALLNKCKTPQGQRLVNQWIKQPLMDKNRIEERLNLVEA 365
            +LA++L+    TP G R++ +W+  P+ D   + ER   + A
Sbjct:  290 TLASVLDCTVTPMGSRMLKRWLHMPVRDTRVLLERQQTIGA 330
```

query word (W = 3)
1. Split query into words
2. Expand word neighborhood
T
PMG
W-mer Database
3. Search database for neighborhood matches
High-scoring Segment Pair (HSP)
4. Extend each hit into alignment

Why BLAST works(1): Pigeonhole and W-mers



- Pigeonhole principle
  - If you have 2 pigeons and 3 holes, there must be at least one hole with no pigeon



- Pigeonholing mis-matches
  - Two sequences, each 9 amino-acids, with 7 identities
  - There is a stretch of 3 amino-acids perfectly conserved
- In general:
  - Sequence length: n
  - Identities: t
  - Can use W-mers for W= [n/(n-t+1)]

Extensions to the basic algorithm

- Ideas beyond W-mer indexing? Desirata:
  - Faster
  - Better sensitivity (fewer false negatives)

1) Filtering: Low complexity regions cause spurious hits
   - Filter out low complexity in your query
   - Filter most over-represented items in your database
2) Two-hit BLAST
   - Two smaller W-mers are more likely than one longer one
   - Therefore it's a more sensitive searching method to look for two hits instead of one, with the same speed.
   - Improves sensitivity for any speed, speed for any sensitivity
3) Beyond W-mers, hashing with non-consecutive k-mers (combs)
   - Next slide

Extension 3: Combs and Random Projections

Key idea:

- No reason to use only consecutive symbols
- Instead, we could use combs, e.g.,

  RGIKW → R*IK* , RG**W, …
- Indexing same as for W-mers:
  - For each comb, store the list of positions in the database where it occurs
  - Perform lookups to answer the query
- How to choose the combs? At random
  - Random projections: Califano-Rigoutsos’93, Buhler’01, Indyk-Motwani’98
  - Choose the positions of * at random
  - Analyze false positives and false negatives

Performance Analysis:

- Assume we select k positions, which do not contain *, at random with replacement
- What is the probability of a false negative ?
  - At most: 1-idperc^k
  - In our case: $1-(7/9)^4 = 0.63...$
- What is we repeat the process l times, independently ?
  - Miss prob. = 0.63^l
  - For l=5, it is less than 10%

Query: RKIWGDPRS

Datab.: RKIVGDRRS

$\downarrow$ k=4

Query: *KI*G***S

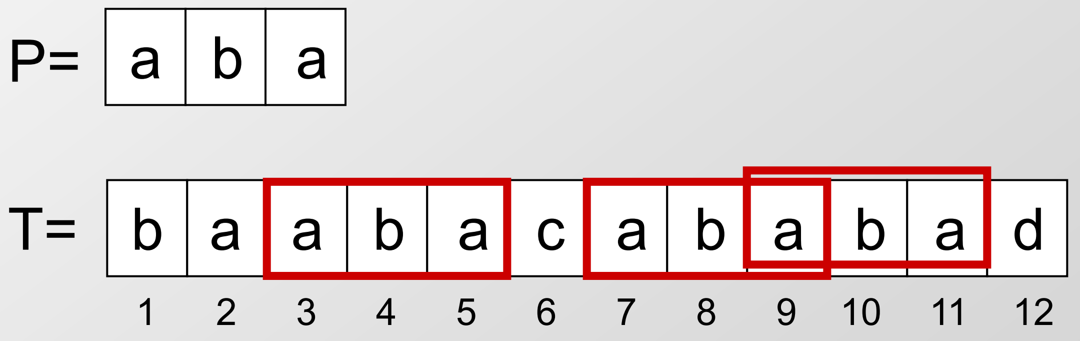
Datab.: *KI*G***S

Today's Goal: Diving deeper into alignments

1. Global alignment vs. Local alignment
   - Needleman-Wunsch and Smith-Waterman
   - Varying gap penalties and algorithmic speedups
2. Linear-time exact string matching (expected)
   - Karp-Rabin algorithm and semi-numerical methods
   - Hash functions and randomized algorithms
3. The BLAST algorithm and inexact matching
   - Hashing with neighborhood search
   - Two-hit blast and hashing with combs
4. **Deterministic linear-time exact string matching**
   - Key insight: gather more info from each comparison
   - Pre-processing, Z-algorithm, Boyer-More, KMP

The exact matching problem

- Inputs:
  - a string ***P***, called the pattern
  - a longer string ***T***, called the text
- Output:
  - Find all occurrences, if any, of pattern ***P*** in text ***T***
- Example



Basic string definitions

S=	b	a	a	b	a	c	a	b	a	b	a	d
	1	2	3	4	5	6	7	8	9	10	11	12

- A *string* S
  - Ordered list of characters
  - Written contiguously from left to write
- A *substring* S[i..j]
  - all contiguous characters from i to j
  - Example: S[3..7] = abaxa
- A *prefix* is a substring starting at 1
- A *suffix* is a substring ending at |S|
- |S| denotes the number of characters in string S

The naïve string-matching algorithm

- NAÏVE STRING MATCHING

```
1     – n ← length[T]
2     – m ← length[P]
3     – for shift ← 0 to n
4         • do if P[1..m] == T[shift+1 .. shift+m]
5             – then print “Pattern occurs with shift” shift
```

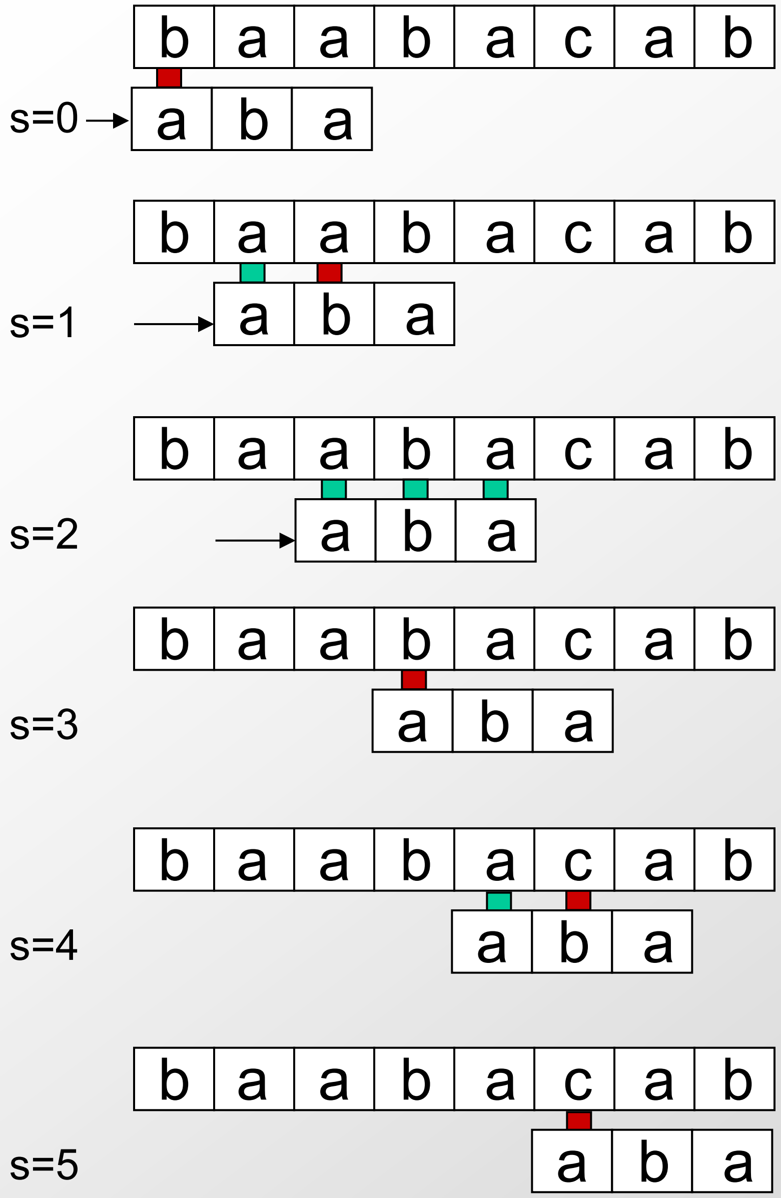
Running time:
O(n)
$\rightarrow$ O(m)

- Where the test operation in line 4:
  - Tests each position in turn
    - If match, continue testing
    - else: stop
- Running time ~ number of comparisons

  number of shifts (with one comparison each)

  + number of successful character comparisons

Comparisons made with naïve algorithm

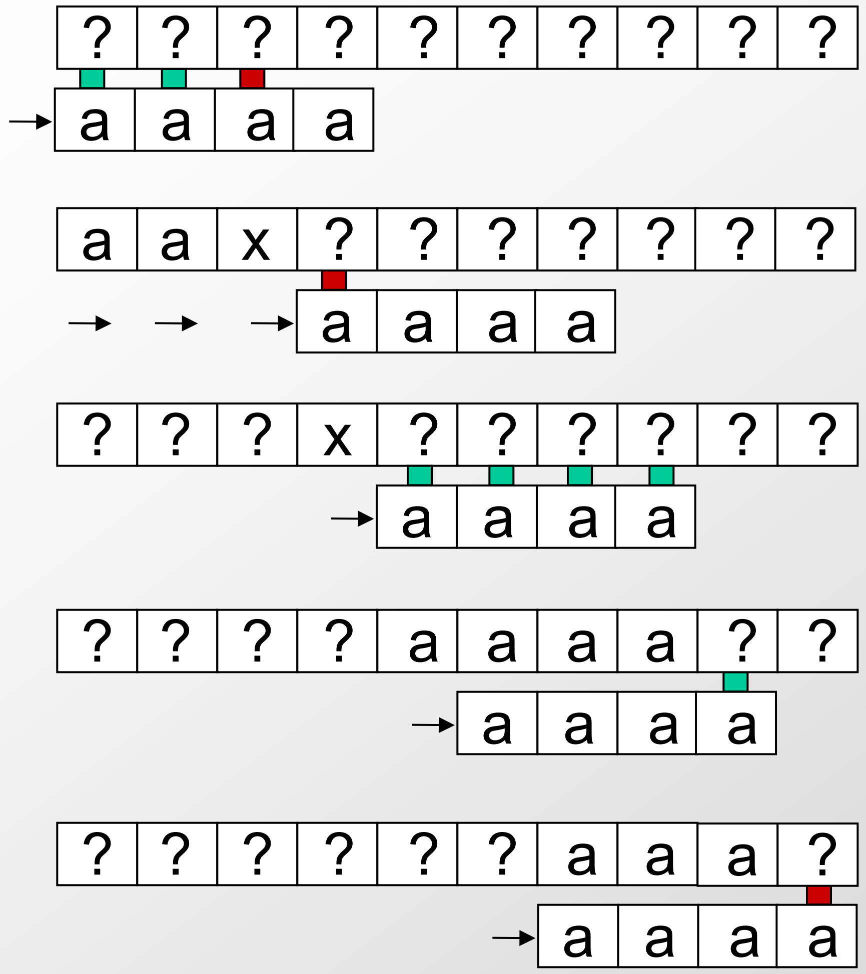


- Worst case running time:
  - Test every position
  - P=aaaa, T=aaaaaaaaaaa
- Best case running time:
  - Test only first position
  - P=bbbb, T=aaaaaaaaaaa

Can we do better?

Key insight: make bigger shifts!

- If all characters in the pattern are the **same**:



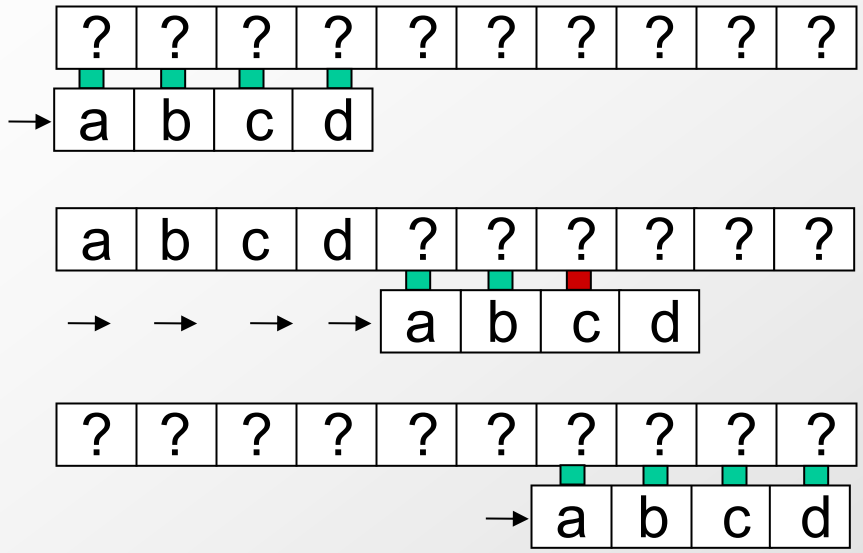
Information gathered at every comparison

Knowledge of the internal structure of P

Number of comparisons: O(n)

Key insight: make bigger shifts!

- If all characters in the pattern are **different**:



Number of comparisons:

•At most n matching comparisons

•At most n non-matching comparisons

➔ Number of comparisons: O(n)

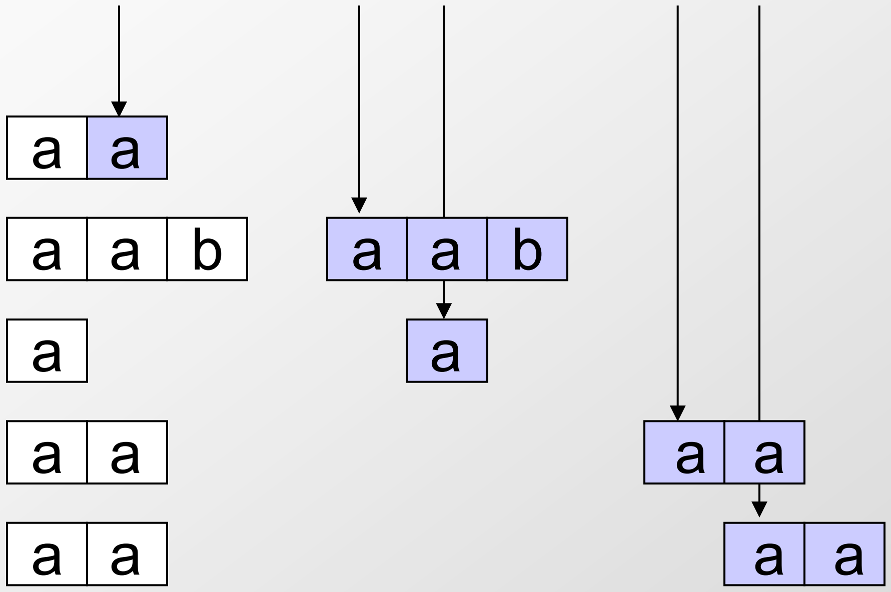
Key insight: make bigger shifts!

- Special case:
  - If all characters in the pattern are **the same**: O(n)
  - If all characters in the pattern are **different**: O(n)
- General case:
  - Learn internal redundancy structure of the pattern
  - Pattern pre-processing step
- Methods:
  - Fundamental pre-processing
  - Knuth-Morris-Pratt
  - Finite State Machine

Fundamental pre-processing

- Learning the redundancy structure of a string S

	1	2	3	4	5	6	7	8	9	10	11
S =	a	a	b	c	a	a	b	x	a	a	a
Z =	0	**1**	0	0	**3**	**1**	0	0	**2**	**2**	1



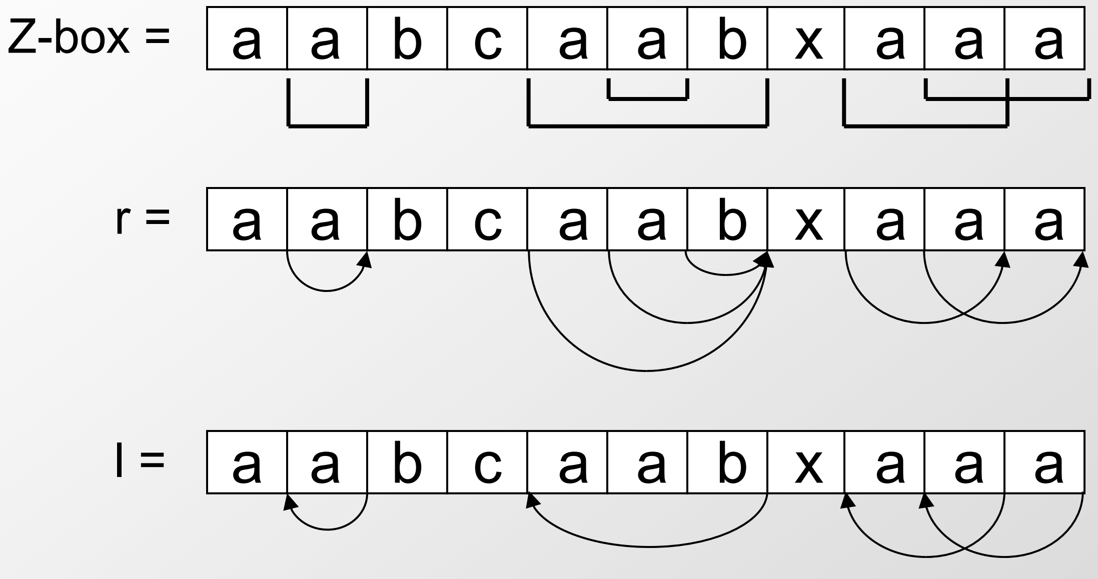
- Zi = length of longest prefix in common for S[i..] and S

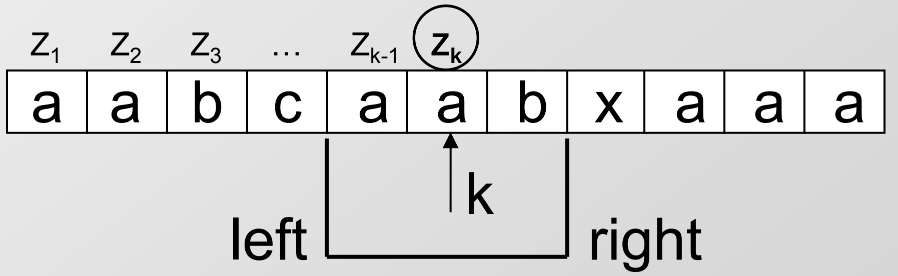
  (Length of the longest prefix of S[i..] that's also a prefix of S)

Fundamental pre-processing

- Learning the redundancy structure of a string S

	1	2	3	4	5	6	7	8	9	10	11
S =	a	a	b	c	a	a	b	x	a	a	a
Z =	0	**1**	0	0	**3**	**1**	0	0	**2**	**2**	1

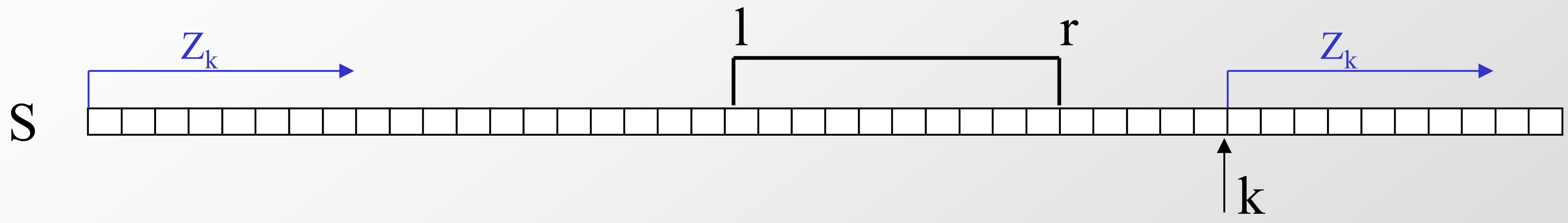




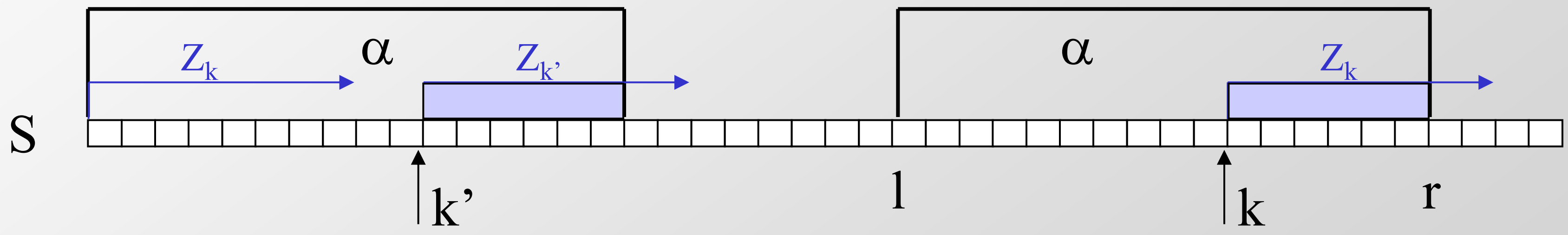
Can we compute Z, r, l in linear time O(|S|)?

Computing Z_k given Z_1 .. Z_{k-1}

- Case 1: k is outside a Z-box: simply compute Z_k



- Case 2: k is inside a Z-box: Look up $Z_{k'}$

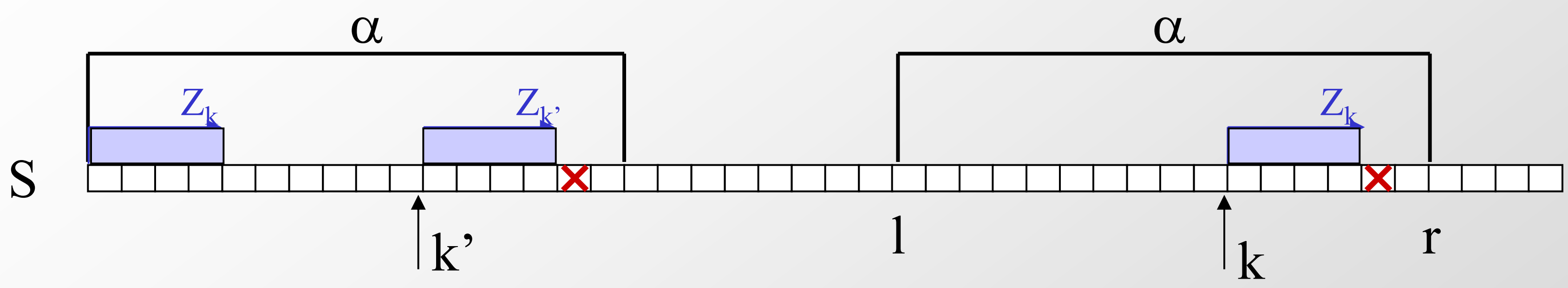


→ Case 2a: Zk' < r-k

→ Case 2b: Zk' >= r-k

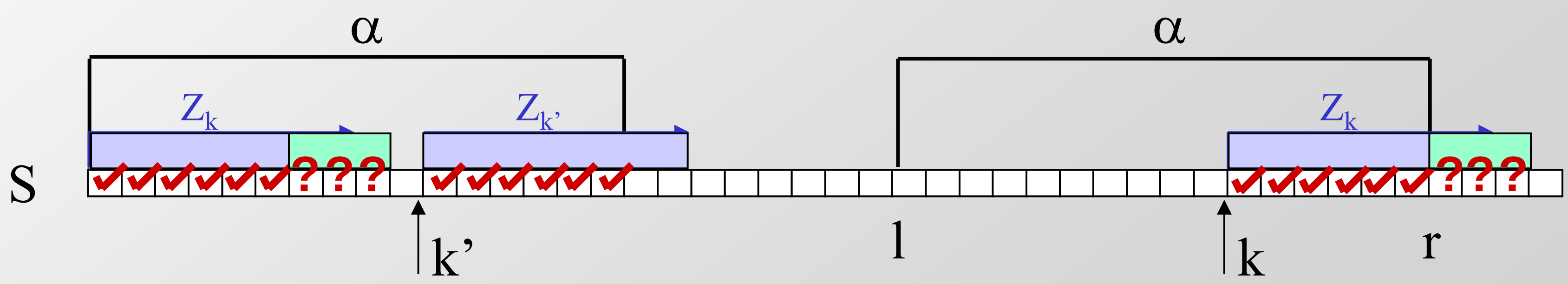
Computing Z_k given Z_1 .. Z_{k-1}

Case 2a: $Z_{k'} < r-k$



Set $Z_k = Z_{k'}$

Case 2b: $Z_{k'} >= r-k$



Explicitly compare starting at r+1

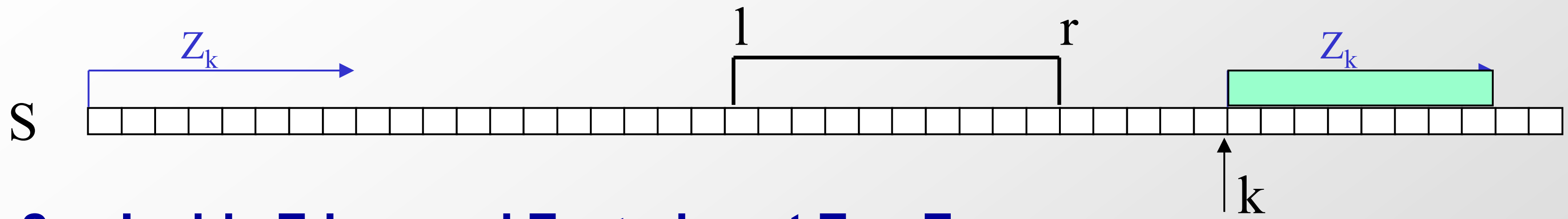
Putting it all together

- FUNDAMENTAL-PREPROCESSING(S):

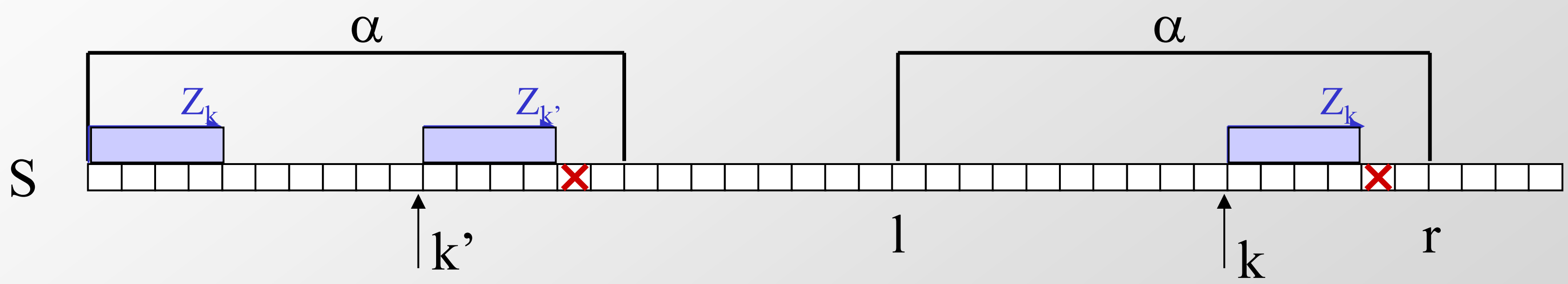
```
$Z_2$,l,r = explicitly compare S[1..] with S[2..]
**for** k in 2..n:
    **if** k > r:  $Z_k$,l,r = explicitly compare S[1..] with S[k..]
    **if** k <= r:
        **if** $Z_{k'}$<(r-k): $Z_k$ = $Z_{k'}$
        **else**:
            $Z_k$ = explicitly compare S[r+1..] with S[(r-k)+1..]
            l = k
            r = l+$Z_k$
```

Correctness of Z computation

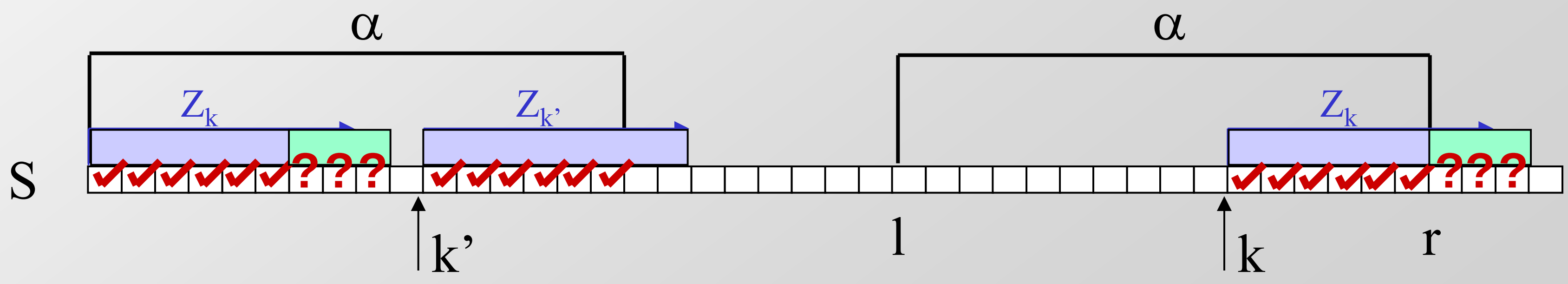
Case 1: k is outside a Z-box: explicitly compute Z_k



Case 2a: Inside Z-box and $Z_{k'} < r\text{-}k$: set $Z_k = Z_{k'}$

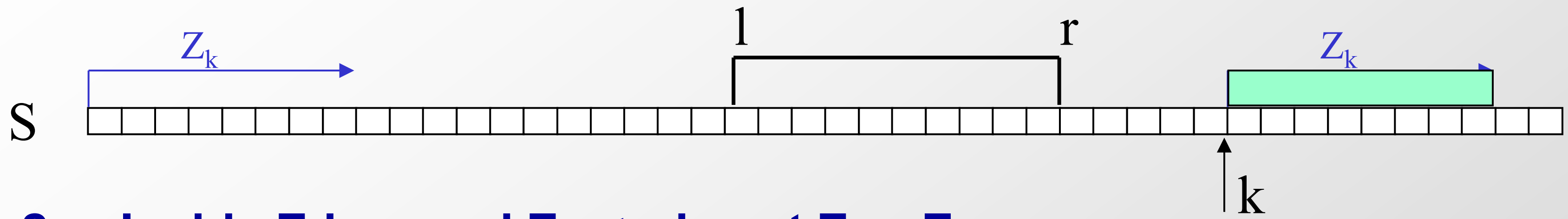


Case 2b: Inside Z-box and $Z_{k'} >= r\text{-}k$: explicitly compute starting at r+1

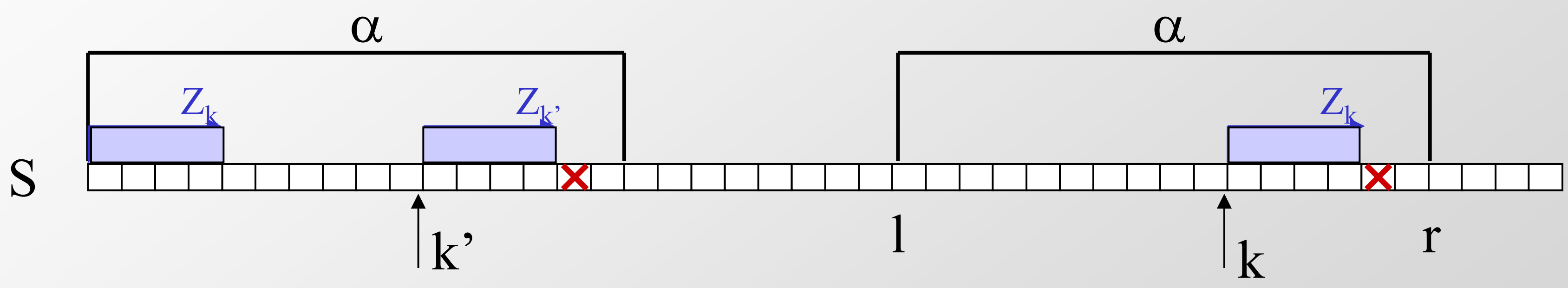


Running time of Z computation

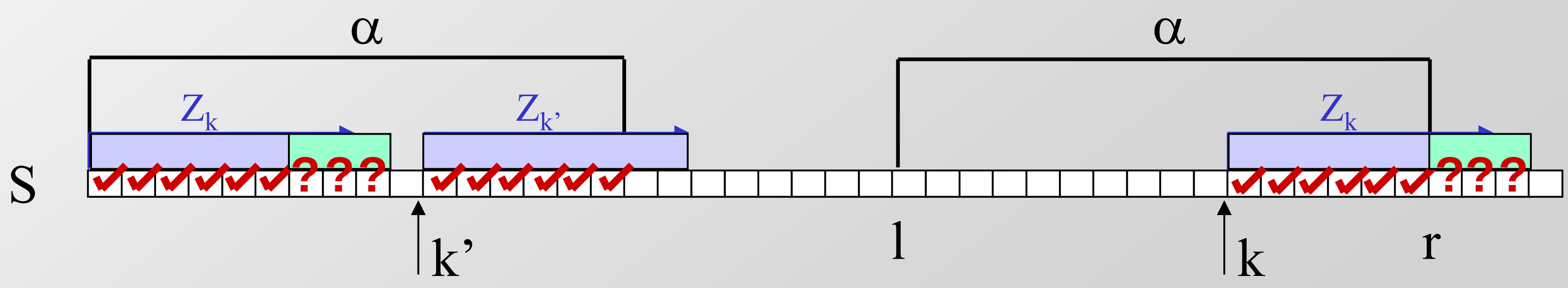
Case 1: k is outside a Z-box: explicitly compute Z_k



Case 2a: Inside Z-box and $Z_{k'}$ < r-k: set Z_k = $Z_{k'}$

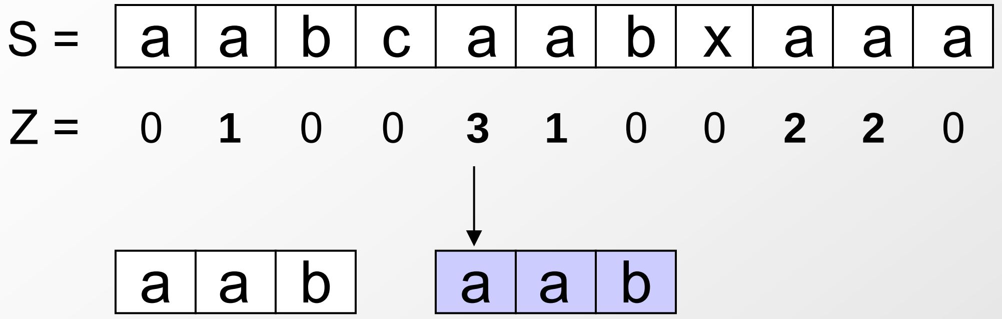


Case 2b: Inside Z-box and $Z_{k'}$ >= r-k: explicitly compute starting at r+1



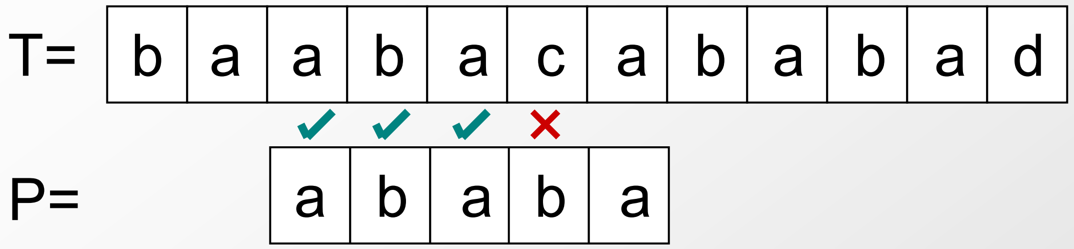
What's so fundamental about Z?

- Learning the redundancy structure of a string S



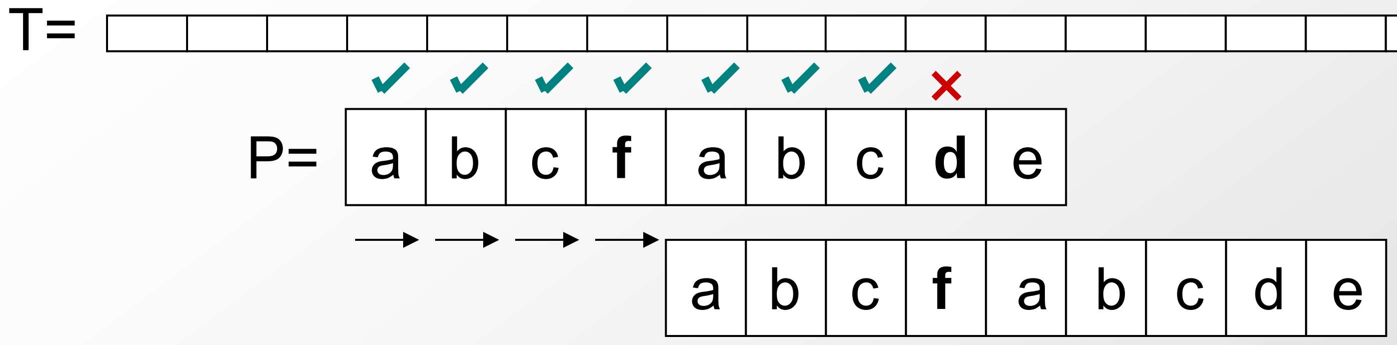
- Z_i = fundamental property of internal redundancy structure
- Most pre-processings can be expressed in terms of Z
  - Length of the longest **prefix** starting/ending at position i
  - Length of the longest **suffix** starting/ending at position i

Back to string matching

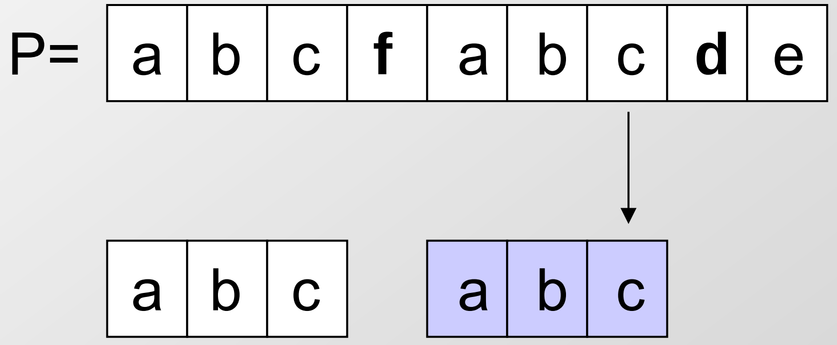


- Given the fundamental pre-processing of pattern P
  - Compare pattern P to text T
  - Shift P by larger intervals based on values of Z
- Three algorithms based on these ideas
  - Knuth-Morris-Pratt algorithm
  - Boyer-Moore algorithm
  - Z algorithm

Knuth-Morris-Pratt algorithm

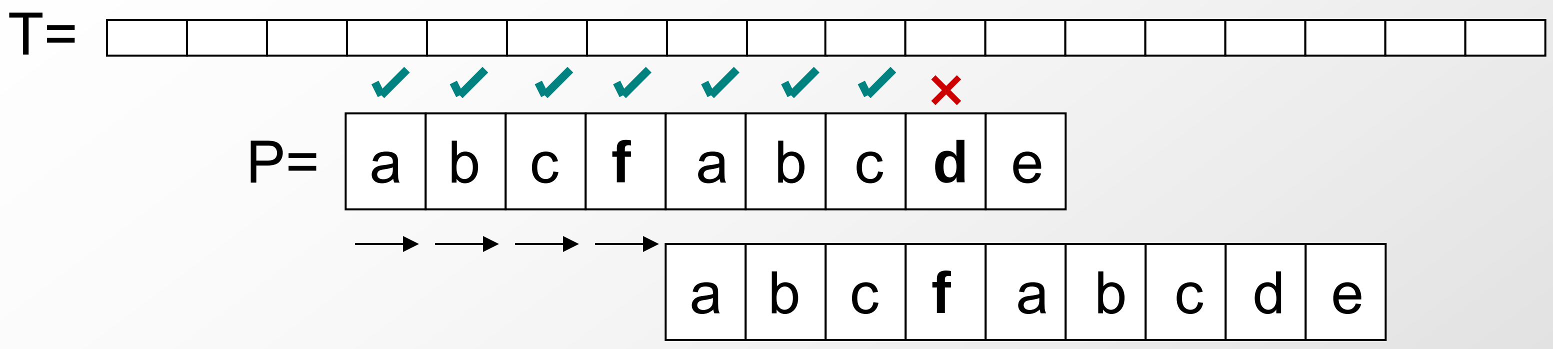


- Pre-processing:
  - $Sp_i(P)$ = length of longest proper suffix of P[1..i] that matches a prefix of P



– No other than the right-hand-side of the Z-boxes

Knuth-Morris-Pratt running time



- Number of comparisons bounded by characters in T
  - Every comparison starts at text position where last comparison ended
  - Every shift results in at most one extra comparison
  - At most |T| shifts → Running time bounded by 2*|T|

Boyer-Moore algorithm

T=	b	a	a	b	x	c	a	b	a	b	a	d
P=	a	b	a	b	x							

- Three fundamental ideas:
  1. Right-to-left comparison
  2. Alphabet-based shift rule
  3. Preprocessing-based shift rule
- Results in:
  – Very good algorithm in practice
  – Rule 2 results in large shifts and sub-linear time
    - for larger alphabets, ex: English text
  – Rule 3 ensures worst-case linear behavior
    - even in small alphabets, ex: DNA sequences

The Z algorithm

P+T=	a	b	a	b	a	$	b	a	a	b	a	c	a	b	a	b	a	d

- The Z algorithm
  - Concatenate P + ‘$’ + T
  - Compute fundamental pre-processing O(m+n)
  - Report all starting positions *i* for which Z_i=|P|

Today's Goal: Diving deeper into alignments

1. **Global alignment vs. Local alignment**
   - Needleman-Wunsch and Smith-Waterman
   - Varying gap penalties and algorithmic speedups
2. **Linear-time exact string matching (expected)**
   - Karp-Rabin algorithm and semi-numerical methods
   - Hash functions and randomized algorithms
3. **The BLAST algorithm and inexact matching**
   - Hashing with neighborhood search
   - Two-hit blast and hashing with combs
4. **Deterministic linear-time exact string matching**
   - Key insight: gather more info from each comparison
   - Pre-processing, Z-algorithm, Boyer-More, KMP