BC205: Algorithms for Bioinformatics. V. Rapid Searches

Christoforos Nikolaou

April 19th, 2018

In previous chapters

- We saw how:
  - We can align two sequences to each other to maximize their match based on some scoring scheme (substitution matrix)
  - We can align multiple sequences in a similar fashion
  - We can define the similarity between two sequences or a sequence and a motif

Why Searches? Why Rapid?

- We are not interested in the optimal alignment (pairwise all multiple) since we usually have one long "reference" sequence and many shorter queries. Thus the "Search"
- We often need to allow for variation but we need this to be coupled with speed and not through a "conventional" alignment strategy
- The number of queries is from very large to really, really large (of the order of millions). Thus there are bottle necks with many existing approaches. (What is the complexity of the Needleman-Wunsch Alignment Algorithm? Imagine that for a large number of sequence alignments)

Problem #1: Searching a string for a specific k-mer

- Given a sequence L and a pattern k how can we search for all instances of k in L?
  - Sliding window in Brute Force (Naive Approach)
  - Sliding window taking into account the context (Knuth-Morris-Pratt and Boyer-Moore algorithms)

Sliding window (Naive approach)

- It is a Brute-Force approach that tries all comparisons exhaustively
  - Requires n-k+1 shifts of the window with k calculations in each: $O(nk)$. But $O(nk)$ is prohibitive when:
    - $n > 10^6$
    - each comparison needs to be done $> 10^6$ times
  - We make a lot of shifts even when we know that there will be no match
- Write the code for the sliding-window and show why it is $O(nk)$

Knuth–Morris–Pratt (KMP)

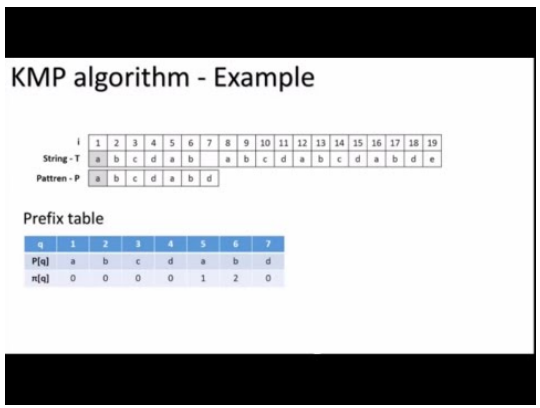
- Used in Plagiarism Check
- Reduces the number of shifts with a two-stage process:
  - A prefix function:
    The goal is to create a **prefix array** that will help us "jump" in the search without making all exhaustive shifts.
  - A string matching:
    It is basically a string matching that takes into account both the k-mer and the L string as well as the **prefix array**.

KMP: Prefix Array Construction

- Goal: To create an array that will tell us for **each** position if there is a *suffix* of size k that matches the prefix of the string up to that point.
- For each i<=k, compute the length of the *longer prefix of N[0..i]* that is also a *suffix of N[0..i]*.
- The **prefix array** construction is solely based on the pattern we are looking for.

KMP: Prefix Array Construction Explained

- Suppose you have a pattern K with length k
- The question is: Find the length of the longest *contiguous* substring s of K that is **at the same time** a prefix *and* a suffix of K
- The length of s is to be assigned in the prefix table of K
- This will help us "skip" $|s|$ elements in the search of K in a longer sequence L



An Example

Given the pattern K=ACACA calculate s:
1. All proper(=not containing itself) prefixes of K are: A, AC, ACA and ACAC
2. All proper suffixes of K are: CACA, ACA, CA, A
3. The longest substring that is ***both*** a prefix ***and*** a suffix of K is ACA. It has a length=3.
4. We assign 3 at the position where ACACA terminates in the prefix array.

- The array for *all substrings* of K=ACACACCAT is the following:

A	C	A	C	A	C	C	A	T
0	0	1	2	3	2	0	0	0

- It takes $O(l)$ time to create the prefix array. This is done as in the following:

KMP: Prefix Array Construction Step-by-Step

P	String	P(#)	Longest matching prefix-suffix
0	“a”	0	""
1	“ab”	0	""
2	“aba”	1	“a” – **a** b **a**
3	“abab”	2	“ab” – **ab ab**
4	“ababa”	3	“aba” – **ab a ba**
5	“ababac”	0	""
6	“ababaca”	1	“a” – **a** babac **a**

KMP Prefix Function

```
def kmpPattern(pattern):
    kmp=[]
    for i in range(len(pattern)+1):
        subpattern=pattern[:i]
        maxlen=0
        for j in range(len(subpattern)):
            if (subpattern[:j]==subpattern[-j:]) +
            + and (len(subpattern[j:])>maxlen):
                maxlen=len(subpattern[:j])
        kmp.append(maxlen)
    kmp.pop(0)
    return(kmp)
```

KMP: Search and Match

- Start with:

1. The sequence L
2. The pattern K
3. The prefix table P

- Use the prefix table to dynamically traverse L and k. The steps are:

1. For i=1 and j=1, compare L[i] and K[j] until you find a mismatch
2. Once you find a mismatch **go back one** position and look-up its value in the prefix table P
3. Shift the pattern so that you are now comparing with i=P[j].
4. Exit if you 've traversed all of K with a full match. Report j-k. (see more here: https://www.youtube.com/watch?v=BQBOETwrVpI)

KMP Search and Match: Pseudocode

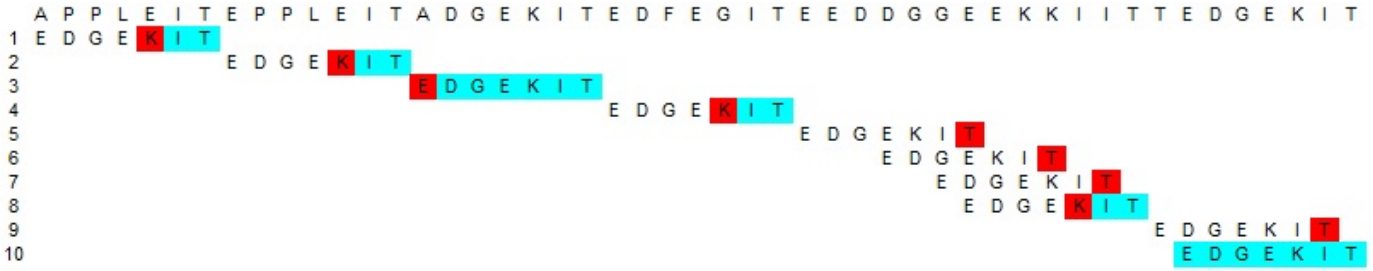
```
def kmpSearch(sequence, pattern):
    successes=[]
    kmppattern=kmpPattern(pattern)
    s=len(sequence);p=len(pattern)
    i=0; j=0
    while i < s:
        if (sequence[i] == pattern[j]):
            i += 1; j += 1
        else:
            if j != 0:
                j = kmppattern[j-1]
            else:
                i = i + 1
        if j == p:
            successes.append(i-j)
            j = kmppattern[j-1]
    return(successes)
```

KMP: Worst case scenario

- What is the worst case scenario for a KMP search?
  - Imagine a prefix table where all values are $=0$
  - When will this happen?
  - What will this mean for our search and match?
- How will KMP perform under the worst case scenario?
- How can we go around this problem?

Boyer-Moore (BM)

- Used in every CTRL+F search you've performed
- Instead of prefix we use a suffix approach. We thus search based on the right-most end of the pattern



Boyer-Moore Explained

Starting from a sequence L and a pattern K we first align L and K starting from $i = 1$ all the way to $i = k$, wher k is the length of K.

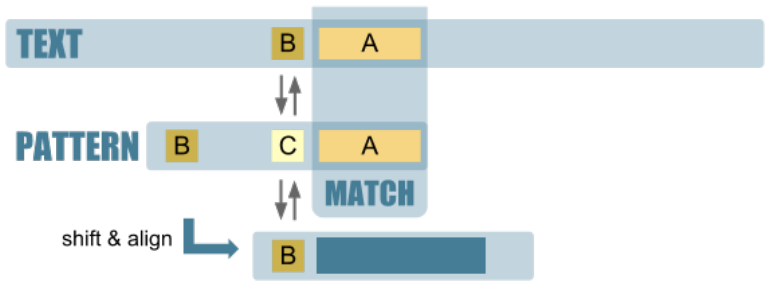
1. From $i = k$ and **backwards** we check if $L[i] = K[i]$
2. Upon hitting a mismatch for which $L[i]! = K[i]$ we make a number of shifts defined by specific tables and resume steps 1-2.

What you need is a lookup table that tells you how many positions you need to skip before carrying on with the alignment. The look-up table is based on two rules. Skipping is done on the basis of the rule that allows you to make the longer pattern shift

Boyer-Moore: Bad Character Rule

1. The bad character rule:

- Upon hitting a mismatch shift the pattern to the right *as many positions as necessary to make a match on the mismatch*.
- If no match can be found shift pattern untill the whole of it is just beyond the mismatch (treat the empty [-1] position as a match).



Boyer-Moore: Creation of the Bad Character look-up table

As with the Prefix Array of KMP it is solely based on the pattern and is constructed in O(kA) time with A being the size of the alphabet of the pattern. The pseudocode looks like this:

```
p=pattern
m=length(p)
a=sizeofalphabet(p)
t=dim[m,a](0)
for i=1 to i<=a;
  for j=1 to j<=m;
    if(p[j]==a[i])
      j--
    if(p[j]!=a[i])
      t[i,j]+=j
```

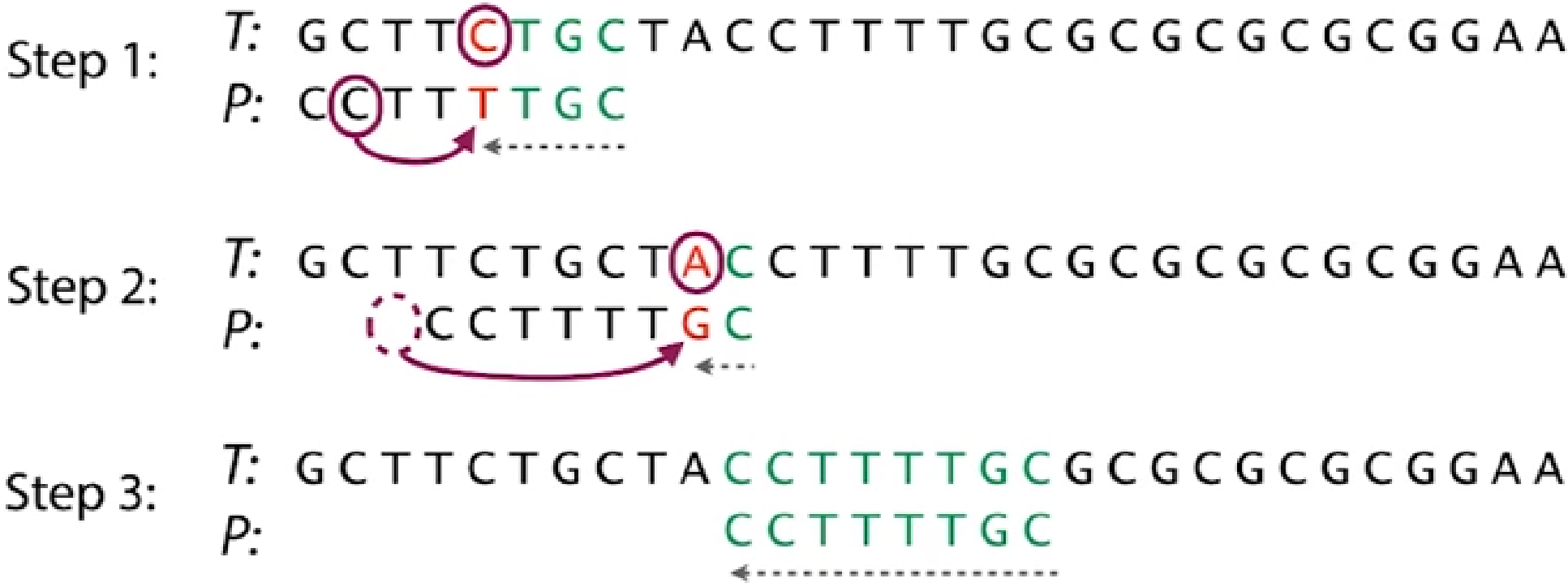
Can you write the code for the above look up table construction?

Boyer-Moore: Bad Character Rule Look-up table

- \- means there is a match so no shift is needed
- 0 denotes one shift to the right

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

Boyer-Moore: Bad Character Rule Example



Boyer-Moore: Good Suffix Rule

2. The good suffix rule:

- Upon hitting a mismatch, shift the pattern to the right as many positions as necessary in order to match a substring of the pattern to the suffix of the matched proportion of the pattern.
- If no match can be found shift pattern untill the whole of it is just beyond the mismatch.

Creation of the look-up table is too complex for our needs and is omitted.

Boyer Moore advantages

- Is faster than both naive and KMP searches
- Works even for patterns that have no proper prefixes
- How long will the algorithm take to run in the case there are no matches? (See Exercises)

Problem #2: Searching for interrupted matches of a pattern in a sequence

- Given a sequence L and another sequence K how can we search for substring matches between K and L?
  - Comparison by content. Break L and K in n-mers. Compare n-mer existence.
  - FASTA searches of aligned k-mers

Comparison by Content

- Take L and K with a word size of $n = 2$ L=GCATCGCCATCG K=GCATCGGC
- Create a (sparse) table holding all 2-mers and their positions for L and K. This takes O(l+k) time to construct.

	AT	CA	CC	CG	GC	GG	TC
L	3,9	2,8	1,7	5,11	6	-	4,10
K	3	2	-	5	1,7	6	4

We can now use this table to compare L and K on the basis of their content with a simple statistic such as: $\sum_{i=1}^{4^n} L[i] * K[i]$

Binary Searches

- Consider the example above for $n = 10$ and for much longer L and K. The size of the table will make the summation process too slow and its sparseness will make things worse (too many 0s).
- A way to go around this is the following:

1. Start by creating an ordered table for the n-mer occurrences of L
2. Now take K and search for each n-mer in the table of L. $+1$ if it is there or leave 0 otherwise:
   Q1: What is the time complexity of this search?
   Q2: How can we speed it up?

Binary Searches

- Binary Searches are the approach described above only instead of exhaustively searching for each n-mer, we search in a binary way:

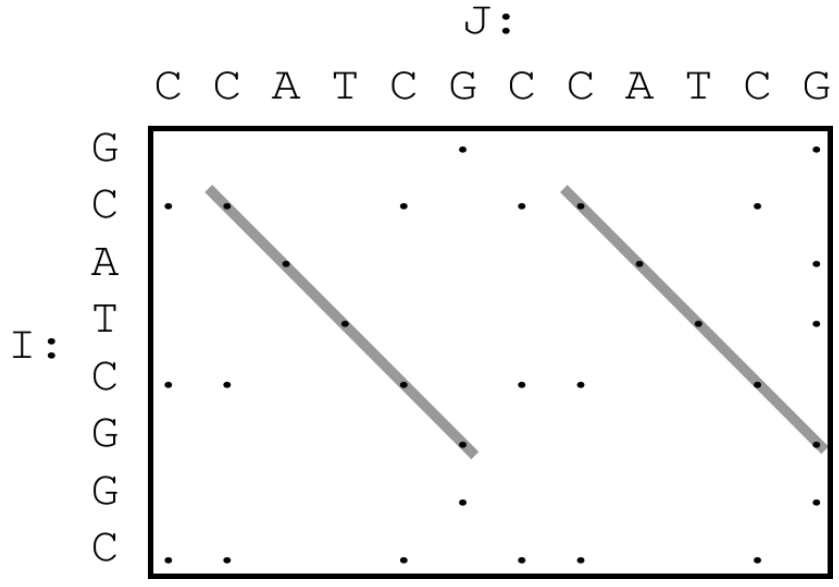
Q: Is n-mer in the top 50% of the ordered list?
- **yes**: Make list equal to the top 50%. Go to 1
- **no**: Make list equal to the bottom 50%. Go to 1

This takes $2^{log(n)}$ time to complete, but we are still missing out on the context.
We need to resort to some form of alignment of L and K

Dot-matrices

In order to make it more sensitive to context we can think of the following:

1. Take the two sequences L and K and plot them in a matrix just like in the pairwise alignment strategy
2. Mark points in diagonals that represent identical substring matches
3. Try to locate diagonal elements above a given threshold in size



The FASTA Algorithm #1: Outline

- Is based on the original FastP algorithm (Wilbur and Lipman) that assumes indels are much more common than substitutions and tries to locate identical matches.
- Our goal is to count the size of all contiguous diagonal elements
- We need:
  1. A way to identify diagonal indexes
  2. A way to score them

The FASTA Algorithm #2: Initialization

- For a pair of sequences with lengths l and k there are $l + k - 1$ diagonals in the matrix
- We create an array S of size $l + k - 1$ indexed from $1 - l$ to $k - 1$. Element 0 is the main(longer) diagonal of the dot matrix
- We then create a table like the one we saw earlier. For L=GCATCGCCATCG, K=GCATCGGC and $n = 2$:

	AT	CA	CC	CG	GC	GG	TC
L	3,9	2,8	1,7	5,11	6	-	4,10
K	3	2	-	5	1,7	6	4

The FASTA Algorithm #3: Strategy

- Starting from sequence K:
  a. Take the first ($i = 1$) 2-mer $K[i] = GC$
  b. Search for GC in L (found at position $p = 6$)
  c. Index d is $i - p = 1 - 6$
  d. Increment S[d]=S[d]+1
- Repeat for all 2-mers of K

The FASTA Algorithm. #4: Pseudocode

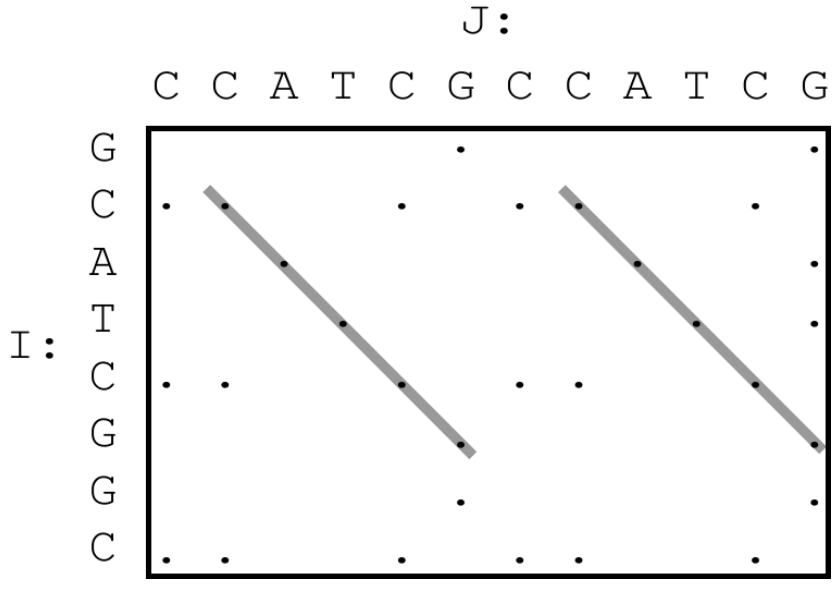
```
input(L,K)
l=length(L)
k=length(K)
n=2 # choice of n
S=[0 for i=range(1-l,k-1)] # creation of S
# find positions of each nmer occurrence in L
nmers=position_of_nmers_in[L]
# now, loop over nmers in K
for i in range(1,length(K)-n):
  # locate (if exists) nmer in L, obtain position in L
  p=nmers[K[i..i+n-1]]
  # find index based on i, p
  d=i-p
  # increment S
  S[d]++
return S
```

The FASTA Algorithm #5: Output

S now is (non-zero elements)

d	-6	-5	0	1
S	4	1	4	1

which match the diagonal elements we saw in the dot matrix



The FASTA algorithm. Questions

1. How did the choice of n affect the running time and the output?
2. What are the limitations of a FASTA approach?
3. Can we search for mismatches with FASTA?

Problem #3: Allowing for mismatches

- Going beyond FASTA. Searching with mismatches
- BLAST. Basic-Local Alignment Tool
- BLAT. BLAST-like Alignment Tool

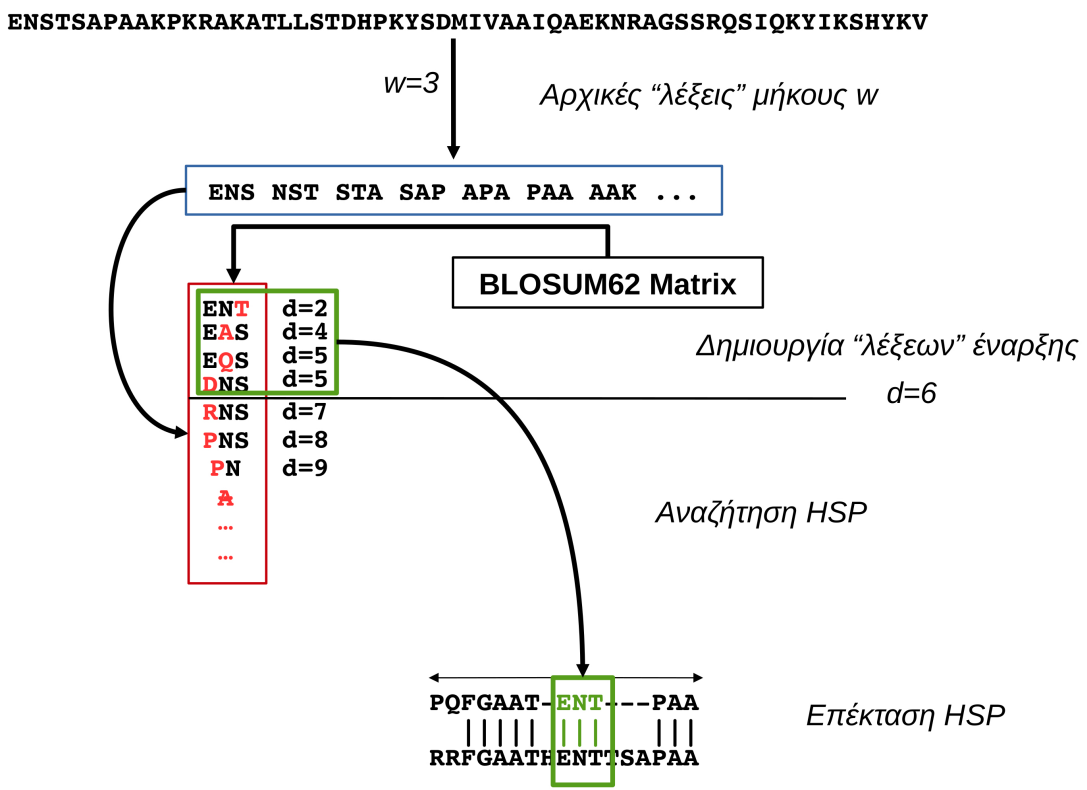
BLAST: Basic Local Alignment Tool

- Is by far the most widely-used (and cited) Bioinformatics algorithm.
- Finds local alignments with mismatches and extends them by joining related (closely located) matched elements (HSP: high scoring segment pairs)

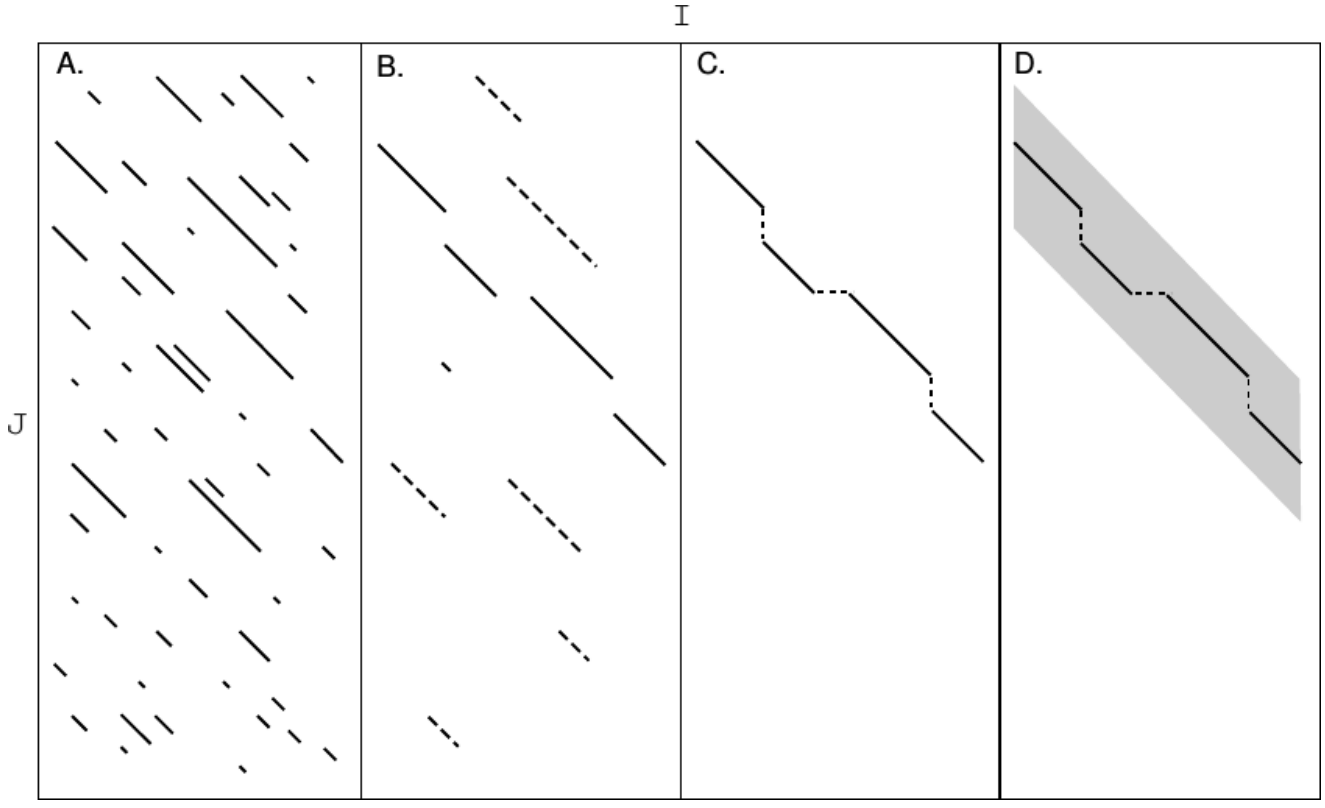
The BLAST Algorithm #1: Strategy

- BLAST is based on four parameters:

1. word length (w)
2. distance (d) of created dictionary (assumes a substitution table)
3. extension (e) for the distance within which HSP are joined
4. E-value is the value on which the statistical inference of the final match is based (we won't be discussing that)



The BLAST Algorithm #2: Output



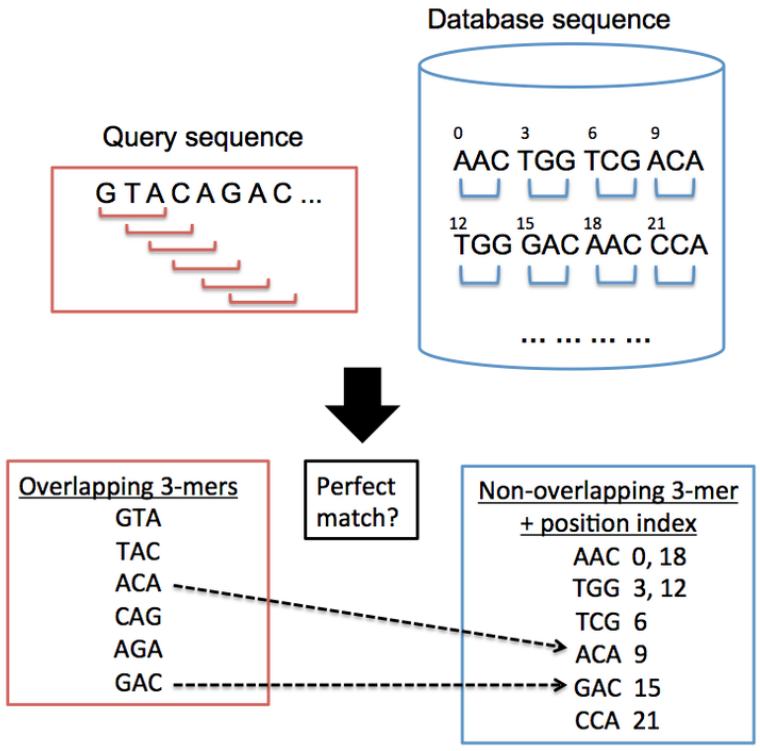
- Question: How does extension (e) parameter affect the image above?

BLAT: A BLAST-like approach that is faster (for some cases)

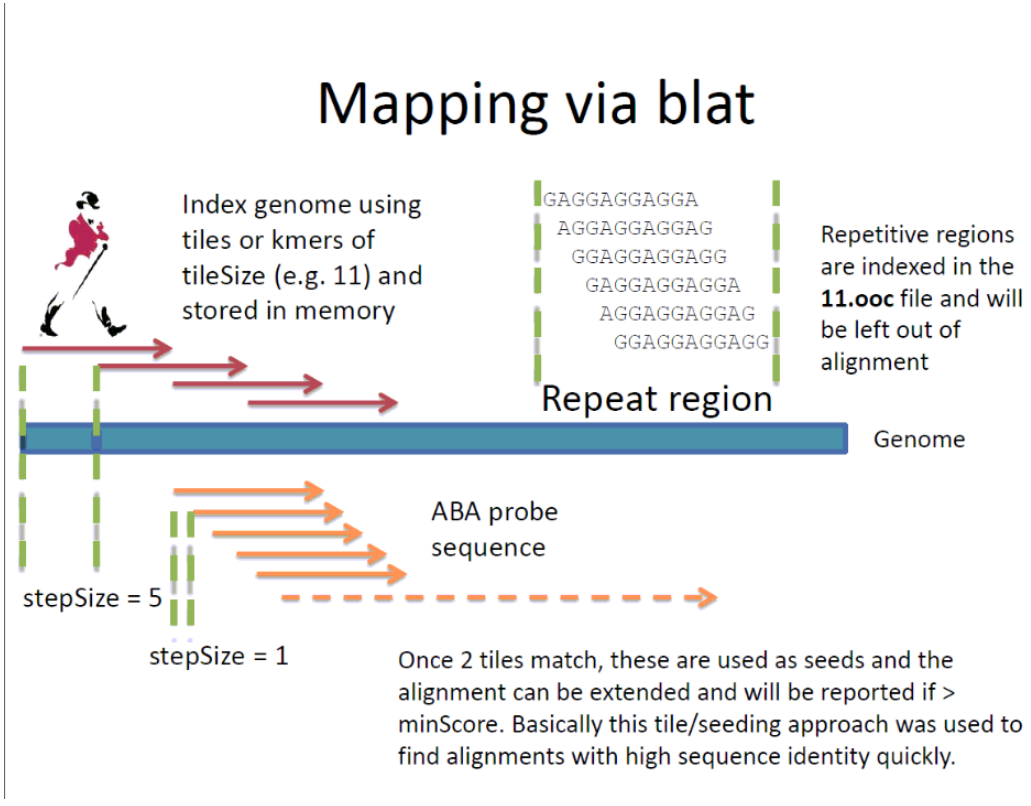
- BLAST is widely used to find matches between a sequence and entire databases.
  - it is fast but not very sensitive when it comes to closely related sequences and
  - it is not fast enough for a large number of repeated searches
- BLAT is the solution for:
  - Closely related sequences (species)
  - Large number of searches

The BLAT algorithm #1: Outline

1. Find perfect matches (like in FASTA) => Increased sensitivity for closely related sequences
2. Require there are a lot of HSP (like in BLAST) => Allows for indels which are common in evolution, even for closely related species
3. Require that they are very close to each other (very small e) => Increased specificity, a lot of false positives are discarded



The BLAT Algorithm #2: Strategy



BLAT vs BLAST and beyond

- Even though BLAT can be ~800 times faster than BLAST (and BLAST is fast) it cannot handle the amount of input generated by NGS experiments (of the order of 10^6 of sequences)
- We need different approaches that make more efficient use of memory.
- Next time: We will see how efficient data structures allow us to achieve even faster matching for huge numbers of searches

Thanks to:

Ben Langmead for input and ideas on the slides
(https://github.com/BenLangmead/ads1-slides)

Questions/Exercises

- The Bad Character Look up table is constructed in O(kA) time. What is the complexity of the Prefix Array Construction in the KMP algorithm?
- What is the time complexity of BM in case the pattern does not exist in the sequence? How much time will the algorithm take to exit for sequence length l and pattern length p.
- Write the code to construct the Bad Character Rule look-up table for a BM implementation on a protein sequence.
- In the FASTA algorithm, how does the choice of n affect the running time and the output of the algorithm?
- How are FASTA, BLAST and BLAT approaches described in this lecture different than the Needlemann-Wunsch and Smith-Waterman alignments?