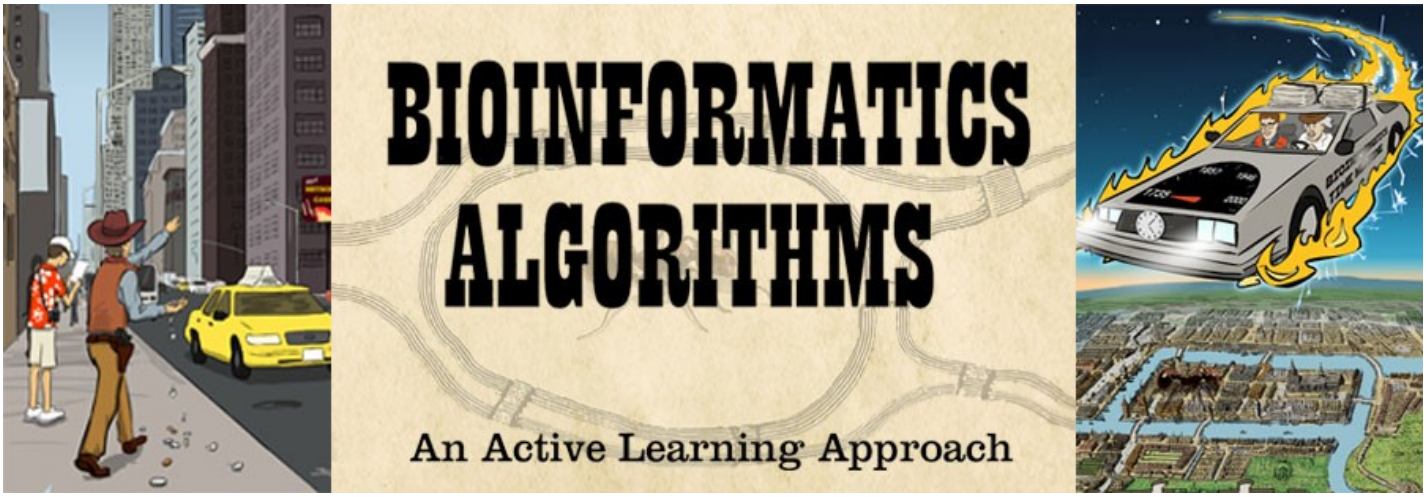


Chapter 9: How Do We Locate Disease-Causing Mutations?

(Coursera Week 1)

Does amalgamation of the reference human genome from various individuals cause problems? Can't such amalgamation produce a phenotype that does not occur naturally?

Yes, the reference human genome is a mosaic of various genomes that does not match the genome of any individual human. Since various human genomes differ by only 0.1%, however, the amalgamation does not cause significant problems.

How does the repeated triplet `"CAG"` affect the severity of Huntington's disease?

Huntington's disease is a rare genetic disease in that it is attributable to a single gene, called *Huntingtin*. This gene includes a trinucleotide repeat `"...CAGCAGCAG..."` that varies in length. Individuals with fewer than 26 copies of `"CAG"` in their *Huntingtin* gene are classified as unaffected by Huntington's disease, whereas individuals with more than 35 copies carry a large risk of the disease, and individuals with more than 40 copies will be afflicted. Moreover, an unaffected person can pass the disease to a child if the normal gene mutates and increases the repeat length. The reason why many repeated copies of `"CAG"` in *Huntingtin* leads to disease is that this gene produces a protein with many copies of glutamine (`"CAG"` codes for glutamine), which increases the decay rate of neurons.

Would it be better to use multiple reference genomes instead of a single reference genome?

Perhaps in theory, but in practice, biologists still use one reference genome, since comparison against thousands of reference genomes would be time-consuming.

What is the point of appending the "$" sign to *Text* when we construct *SuffixTrie*(*Text*)?

Construct the suffix trie for "papa" and you will see why we have added the "$" sign – without the "$" sign, the suffix "pa" will become a part of the path spelled by the suffix "papa".

Why do we construct the trie of all suffixes and not the trie of all prefixes for pattern matching?

If a pattern matches the text starting at position *i*, we want this pattern to correspond to a path starting at the root of a constructed tree. Therefore, we are interested in the suffix starting at position *i* rather than the prefix ending at position *i*.

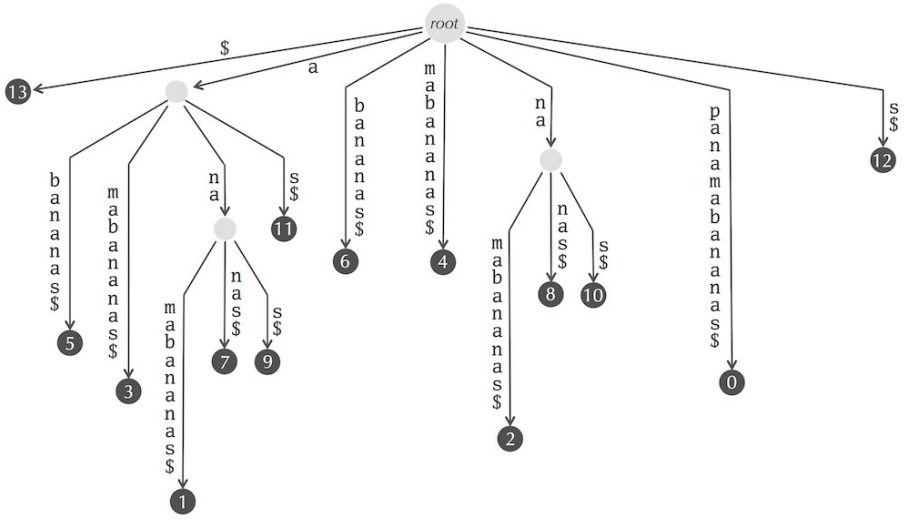
Why do we need an edge from the root to the leaf in the suffix tree (labeled by the "$" sign) if this edge is never traversed during pattern matching?

Although we indeed do not need this edge, it is included to simplify the description of the suffix tree.

What are the edge labels in the suffix tree for `"panamabananas$"`?

The suffix tree for `"panamabananas$"` reproduced below contains 17 edges with the following labels (note that different edges may have the same labels):

```
$
a
bananas$
mabananas$
na
mabananas$
nanas$
s$
s$
bananas$
mabananas$
na
mabananas$
nas$
s$
panamabananas$
s$
```



How does storing *SuffixTree*(*Text*) require memory on the order of 20·|*Text*| if the number of nodes in the suffix tree does not exceed 2·|*Text*|?

In addition to storing the nodes and edges of the suffix tree, we also need to store the information at the edge labels. Storing this information takes most of the memory allocated for the suffix tree.

How can I construct a suffix tree in linear time?

Suffix trees were introduced by Weiner, 1973. However, the original linear-time algorithm for building the suffix tree was extremely complex. Although the Weiner algorithm was greatly simplified by Esko Ukkonen in 1995, it is still non-trivial. Check out this excellent StackOverflow post by Johannes Goller if you are interested in seeing a full explanation.

Why does the suffix tree for the 3-billion nucleotide human require about 60 GB of memory?

A suffix tree for a string *Text* has |*Text*|+1 leaves and up to |*Text*| other nodes. The last figure in "Charging Station: Constructing a Suffix Tree" illustrates that we need to store two (rather large) numbers for each edge of the suffix tree, each requiring at least 4 bytes for the aprroximately 3

billion nucleotide human genome. Thus, since there are approximately 2·|*Text*| edges in the suffix tree of *Text*, the suffix tree for the human genome requires at least $3 \cdot 10^9 \cdot 2 \cdot (4+4) = 48$ GB. And we have not even taken into account some other things that need to be stored, such as the human genome itself :)

How do we construct the suffix tree for a human genome with 23 chromosomes?

You can simply construct the suffix tree for the contatenation of all 23 chromosomes. Don't forget to add 22 different delimiters (i.e., any symbols that differ from "`A`", "`C`", "`G`", and "`T`") when you construct the concatenated string.

Can I see an example of how PatternMatchingWithSuffixArray works?

We illustrate how **PatternMatchingWithSuffixArray** matches `"ana"` against the suffix array of `"panamabananas$"`, reproduced below from the main text (the suffix array is the column on the left).

Starting Positions	Sorted Suffixes
13	$
5	abananas$
3	amabananas$
1	anamabananas$
7	ananas$
9	anas$
11	as$
6	bananas$
4	mabananas$
2	namabananas$
8	nanas$
10	nas$
0	panamabananas$
12	s$

- It first initializes *minIndex* = 0 and *maxIndex* = |*Text*| = 13 and computes *midIndex* = ⌊(0+13)/2⌋ = 6. It then compares *Pattern* with the suffix `"as$"` of *Text* starting at position *SuffixArray*(6). Since `"ana"` < `"as$"`, it assigns *maxIndex* = *midIndex* - 1 = 5 and computes *midIndex* = ⌊(0+5)/2⌋ = 2.
- Since `"ana"` is larger than suffix `"amabananas$"` of *Text* starting at position *SuffixArray*(2), it assigns *minIndex* - *midIndex* + 1 = 3 and computes *midIndex* = ⌊(3+5)/2⌋ = 4.
- Since `"ana"` is smaller than the suffix `"ananas$"` of *Text* starting at position *SuffixArray*(4), it assigns *maxIndex* = *midIndex* - 1 = 3 and computes *midIndex* = ⌊(3+3)/2⌋ = 3.
- Since `"ana"` is smaller than the suffix `"anamabananas$"` of *Text* starting at position *SuffixArray*(3), it assigns *maxIndex* = *midIndex* - 1 = 2.

The last assignment breaks the first while loop since *maxIndex* is now smaller than *minIndex*. As a result, after the first while loop ends, we have *maxIndex* = 2, *minIndex* = 3, and

- suffix of *Text* starting at position *SuffixArray*(2) = `"amabananas$"` < `"ana"`
- suffix of *Text* starting at position *SuffixArray*(3) = `"anamabananas$"` > `"ana"`

Therefore, the first index of the suffix array corresponding to a suffix beginning with `"ana"` is *first* = 3.

The second while loop finds the last index of the suffix array corresponding to a suffix beginning with `"ana"`.

- PatternMatchingWithSuffixArray first sets *minIndex* = *first* = 3, *maxIndex* = |*Text*| = 13, and computes *midIndex* = ⌊(minIndex + maxIndex)/2⌋ = ⌊(3+13)/2⌋ = 8.
- Since `"ana"` does not match the suffix `"mabananas$"` of *Text* starting at position *SuffixArray*(8), it assigns *maxIndex* = *midIndex* - 1 = 7 and computes *midIndex* = ⌊(3+7)/2⌋ = 5.
- Since `"ana"` matches the suffix `"anas$"` of *Text* starting at position *SuffixArray*(5), it assigns *minIndex* = *midIndex* + 1 = 6 and computes *midIndex* = ⌊(6+7)/2⌋ = 6.
- Since `"ana"` does not match the suffix `"as$"` of *Text* starting at position *SuffixArray*(6), it assigns *maxIndex* = *midIndex* - 1 = 5.

The last assignment breaks the second while loop and assigs *last* = *maxIndex* = 5 as the last index of the suffix array corresponding to a suffix beginning with `"ana"`.

To better understand the logic of **PatternMatchingWithSuffixArray**, you may want to check the FAQ on modifying **BinarySearch** for determining how many times a key is present in an array with duplicates.

Why does the suffix array for the 3-billion nucleotide human require 12 GB of memory? Is about 3 GB not sufficient?

The suffix array for the human genome indeed has about 3 billion elements. However, since each element represents one of $3 \cdot 10^9$ positions in the human genome, we need 4 bytes to store each element.

Why do we need the LCP array to transform a suffix array into a suffix tree?

It may seem easier to subsequently add each suffix to the growing suffix tree by finding out where each newly added suffix branches from the growing suffix tree. But this straightforward approach results in quadratic running time, compared to the linear time algorithm in the textbook. (Keep in mind that the LCP array can be constructed in linear time.)

(Coursera Week 2)

Is it possible to construct the Burrows-Wheeler Transform in linear time and space?

Our naive approach to constructing BWT(*Text*) requires constructing the matrix M(*Text*) of all cyclic rotations, which requires $O(|Text|^2)$ time and space. However, there exist algorithms constructing BWT(*Text*) in linear time and space. One such algorithm first constructs the suffix array of *Text* in linear time and space, then uses this suffix array to construct BWT(*Text*).

What is special about the final column of the Burrows-Wheeler matrix? Why not work with some other column?

In short, the last column is the only invertible column of the Burrows-Wheeler matrix. In other words, it is the only column from which we are always able to reconstruct the original string *Text*.

For example, strings `001` and `100` have identical third columns in the Burrows-Wheeler matrix, as shown below.

```
$001      $100
001$      0$10
01$0      00$1
1$00      100$
```

Doesn't the Last-to-First mapping require a lot of memory?

In practice, it is possible to compute the Last-to-First mapping of a given position of BWT(*Text*) with very low runtime and memory using the array holding the first occurrence of each symbol in the sorted string. Unfortunately, the analysis is beyond the scope of this class. For details, please see Ferragina and Manzini, 2000 (click here for full text).

Why does *FirstColumn* appear among the arguments in **BWMatching** if it is never used in the **BWMatching** pseudocode?

We indeed do not use *FirstColumn* in **BWMatching**. Although it seemingly does not make sense, we prefer this because we use *FirstColumn* in a modification of of **BWMatching** in a later section.

How does the array *FirstOccurrence* reduce memory if we still need the larger array *FirstColumn* to construct *FirstOccurrence* in the first place?

When we use *FirstColumn* to construct *FirstOccurrence*, we can immediately release the memory taken by *FirstColumn*. Furthermore, there is an alternative way to construct *FirstOccurrence* without using *FirstColumn*.

Why do the first and last occurrences of *symbol* in the range of positions from *top* to *bottom* in *LastColumn* have respective ranks $Count_{symbol}(top, LastColumn)+1$ and $Count_{symbol}(bottom+1, LastColumn)$?

Given an index *ind* in the array *LastColumn* (varying from 0 to 13 in the example shown in the text), the number of occurrences of *symbol* before position *ind* (i.e., in positions with indices less than *ind*) is defined by $Count_{symbol}(ind, LastColumn)$. Since the number of occurrences of *symbol* starting before position *ind* is equal to $Count_{symbol}(ind, LastColumn)$, the rank of the first occurrence of *symbol* starting from position *ind* is

$$Count_{symbol}(ind, LastColumn) + 1$$

To be more precise, it is $Count_{symbol}(ind, LastColumn) + 1$ if *symbol* occurs in *LastColumn* at or after position *ind*.

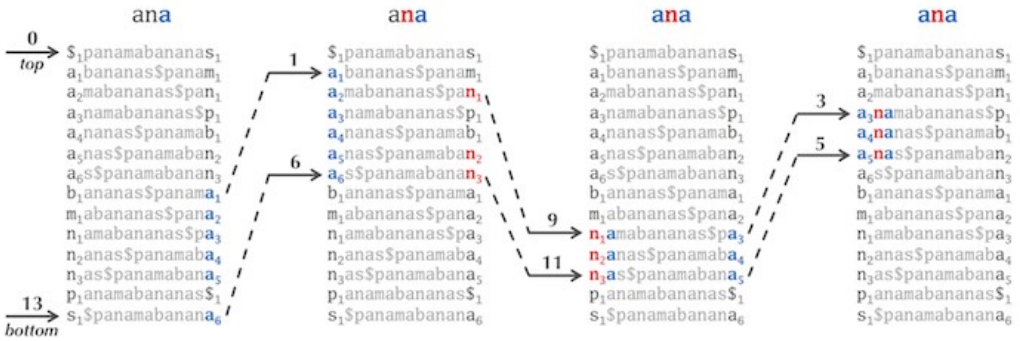
Similarly, the rank of the last occurrence of symbol starting before or at position *ind* is given by

$$Count_{symbol}(ind + 1, LastColumn)$$

For example, when $ind = 5$, the rank of the first occurrence of "n" starting at position 5 is $Count_{"n"}(5, LastColumn) + 1 = 1 + 1 = 2$. On the other hand, the rank of the last occurrence of "p" starting before or at position *ind* is $Count_{"p"}(6, LastColumn) = 1$.

Would it be possible to match a pattern using the Burrows-Wheeler Transform by moving forward through the pattern instead of moving backward?

Let's match `"nam"` against $BWT($`"panamabananas$"`$)$, but instead of matching backward (like in the figure below, reproduced from the text), let's try to match it forward. We can easily find the three occurrences of `"n"` in the first column to start this matching, but afterwards we need to match the next symbol `"a"` in `"nam"`. However, this symbol is "hiding" in the second column, and we have no clue what is in the second column - the Burrows Wheeler Transform does not reveal this information! And there is no equivalent of the "First-Last Property" for the second column to help us.



Is **BetterBWMatching** guaranteed to terminate?

The condition "$top \leq bottom$" is a **loop invariant**, or a property that holds before and after each iteration of the loop. In this case, if pattern matches have been found, the number of matches is equal to $bottom - top + 1$. If pattern matches are not found, then at some point in the loop, *bottom* becomes equal to $top - 1$, in which case $top \leq bottom$ and the loop terminates.

Would **BetterBWMatching** work properly if *Pattern* contains symbols that do not appear in *Text*?

No; however, you can easily modify **BetterBWMatching** by first checking whether *Pattern* contains symbols not present in *Text* and immediately returning 0 in this case.

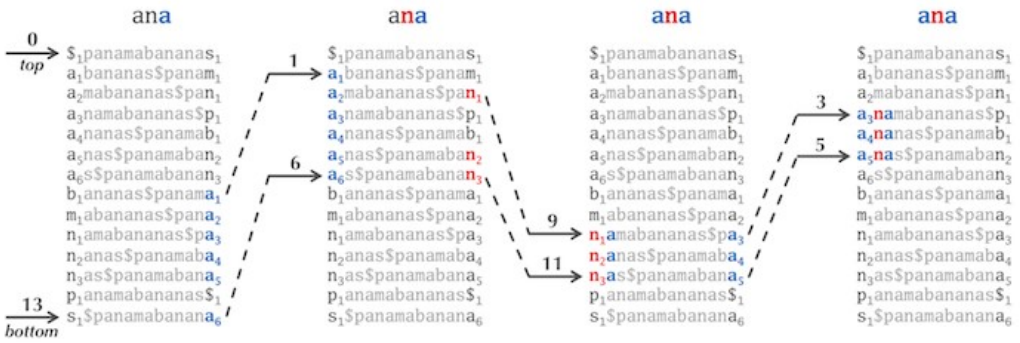
In the main text, you illustrated walking backward in **BetterBWMatching** with the pattern "ana", which is a palindrome. How can we match a non-palindromic pattern?

Try "walking backwards" to find the one pattern match of "ban" in "panamabananas$".

(Coursera Week 3)

How do we select the constant *K* for constructing the partial suffix array?

Selecting a large value of *K* reduces the memory allocated to the partial suffix array by a factor of *K* but increases the time needed to "walk backward" during the pattern matching process (see the figure below, reproduced from the main text). This backward walk may take up to *K*-1 steps ((*K*-1)/2 steps on average). Thus, it makes sense to select the minimum value of *K* that allows fitting the BWT pattern matching code into the memory on your machine.

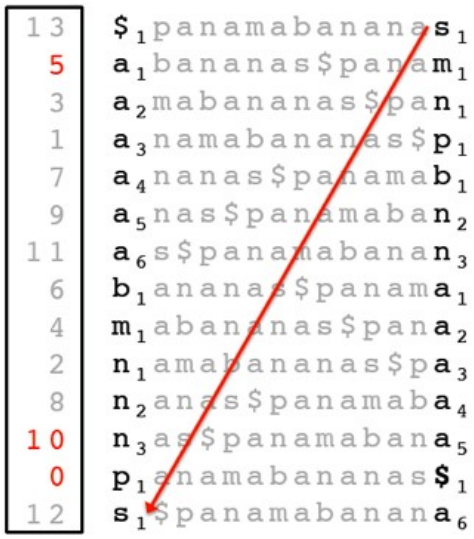


If we have to construct the entire suffix array to build the partial suffix array, how can it save memory?

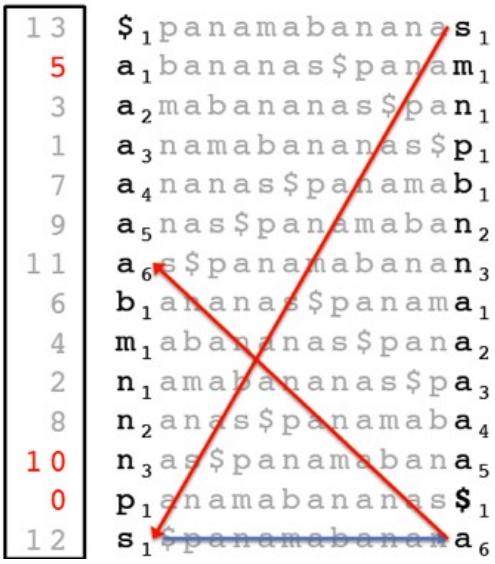
The partial suffix array can indeed be constructed without constructing the (full) suffix array first. The figure below shows the partial suffix array of *Text* = "panamabananas$" with *K* = 5. Although it is unclear how to find where the entries 0, 5, and 10 in the suffix array are located without constructing this array first, we know that the top-most entry of the suffix array is 13, corresponding to the length of *Text*):

13	$ panamabananas
5	a bananas$panam
3	a mabananas$pan
1	a namabananas$p
7	a nanas$panamab
9	a nas$panamaban
11	a s$panamabanan
6	b ananas$panama
4	m abananas$pana
2	n amabananas$pa
8	n anas$panamaba
10	n as$panamabana
0	p anamabananas$
12	s $panamabanana

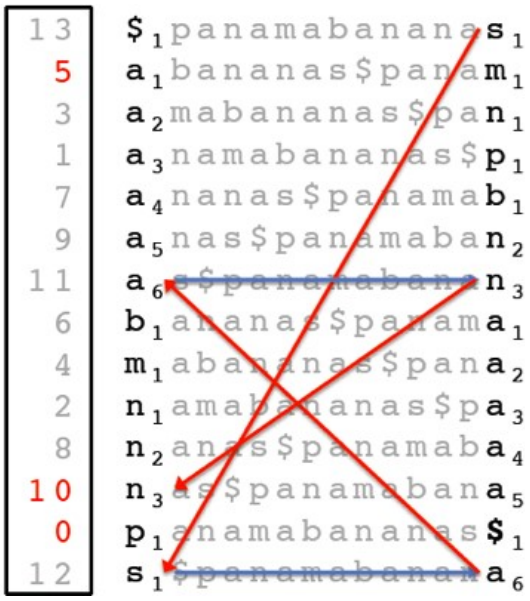
Using the *LastToFirst* function, we can find out where entry 12 is located:



And in turn, we can find entry 11:



At last, we find entry 10, which is the one we were looking for!



After entry 10 in the partial suffix array is found, we can apply the *LastToFirst* function five more times and find the position of 5 in the partial suffix array. Slowly but surely, after another round of five applications of the *LastToFirst* function, we will find the position of 0.

However, since the *LastToFirst* function consumes a lot of memory, we cannot explicitly use it to construct the partial suffix array. We could also use the *Count* arrays, but these require a lot of memory too… yet we can substitute the *Count* arrays by memory-efficient checkpoint arrays (which can also be computed without needing the original *Count* arrays) to achieve the same goal, and therefore construct the partial suffix array efficiently.

It seems as though the partial suffix array will require using the *LastToFirst* mapping. But we got rid of the *LastToFirst* mapping in order to speed up pattern matching and save memory! Why do we do this?

We indeed got rid of the *LastToFirst* array; however, in the same section we saw how the *Count* arrays can be used as a substitute for *LastToFirst*.

How do we select the constant *C* when constructing checkpoint arrays?

Selecting a large value of *C* reduces the memory allocated to the checkpoint arrays by a factor of *C* but increases the time for computing *top* and *bottom* pointers by a factor of *C* in the worst case. Thus, it makes sense to select the minimum value of *C* that allows fitting the BWT pattern matching code into the memory on your machine.

What modifications of BetterBWMatching are needed to make it work with checkpoint arrays instead of count arrays?

To explain how to modify BetterBWMatching for working with checkpoint arrays, we explain how to quickly compute each value in the count array given the checkpoint arrays and *LastColumn*.

To compute $Count_{symbol}(i, LastColumn)$, we represent i as $t \cdot K + j$, where $j < K$. We can then compute $Count_{symbol}(i, LastColumn)$ as $Count_{symbol}(t \cdot K, LastColumn)$ (contained in the checkpoint arrays) plus the number of occurrences of *symbol* in positions $t \cdot K + 1$ to i in *LastColumn*.

How do biologists determine the maximum allowable number of mismatches while mapping reads to the human genome?

Biologists usually set a small threshold for the maximum number of mismatches, since otherwise read mapping becomes too slow.

Can reads that "fall off the edges of the text" form approximate matches?

For example, does *Pattern* = "TTACTG" match *Text* = "ACTGCTGCTG" with *d* = 2 mismatches? Not according to the statement of the Multiple Approximate Pattern Matching Problem, since there is no *starting position* in *Text* where *Pattern* appears as a substring with at most *d* mismatches. However, if you want to count approximate matches falling off the edges of *Text*, you can simply add short strings formed by "$" signs before the start and after the end of *Text*.

What is the running time of approximate pattern matching using the Burrows Wheeler Transform?

Unfortunately, the running time scales roughly as A^k, where *A* is the alphabet size and *k* is the number of mismatches. This is why the existing read matching tools based on the Burrows Wheeler Transform become prohibitively slow when the number of mismatches increases. For more details, see N. Zhang, A. Mukherjee, D.Adjeroh, T. Bell. "Approximate Pattern Matching using the Burrows-Wheeler Transform." *Data Compression Conference*, 2003, 458.

How does seed extension work?

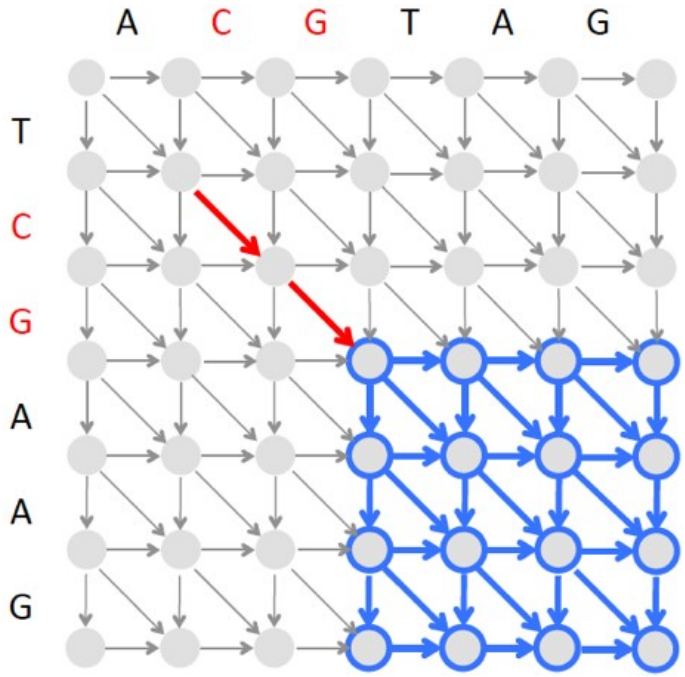
After finding a seed of length *k* that starts at position *i* in *Pattern* and position *j* in *Text*, approximate pattern matching algorithms find a substring of *Text* of length |*Pattern*| that starts at position *j-i*. If this substring matches *Pattern* with at most *d* mismatches, then it is reported as an approximate match between *Pattern* and *Text*.

How does BLAST find all *k*-mers that have scores above a given threshold when scored against a *k*-mer in the query string?

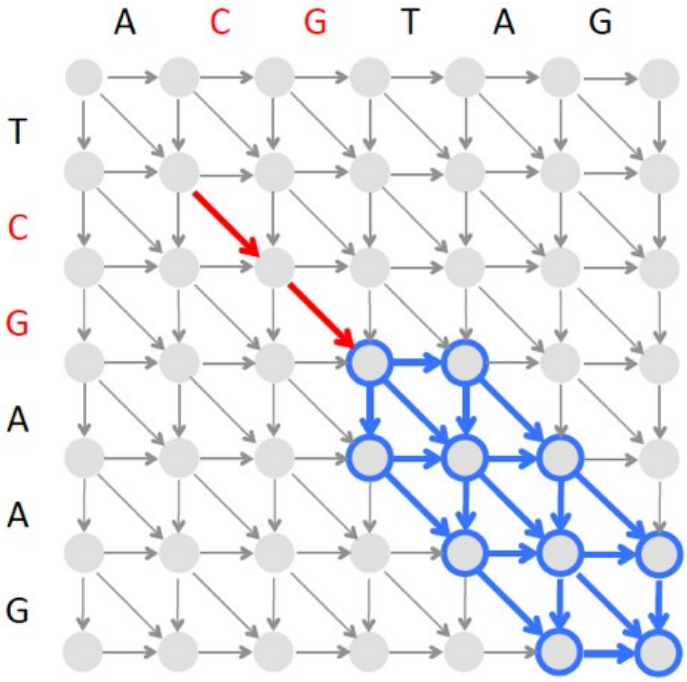
To find all *k*-mers that score above a threshold against a given *k*-mer *a*, we can generate the 1-neighborhood of *a* and retain only *k*-mers from this set that score above the threshold. Iterate by applying the same procedure to each *k*-mer in the constructed set. See the BLAST paper for a more efficient algorithm.

How does BLAST extend the seeds that it identifies? Does it not require constructing an optimal alignment, thus significantly slowing down the algorithm?

After finding a seed, BLAST does construct an alignment but imposes a limit on the number of insertions and deletions in this alignment. For example, after finding the two amino acid-long seed CG below, we can find the best local alignment starting at the "end" of this seed (shown by the blue rectangle shown below) and another local alignment ending at the "beginning" of this seed (the corresponding rectangle is not shown). The resulting local alignments extending this seed from both sides form a full-length alignment.



However, since finding an optimal local alignment in the blue rectangle above is time consuming, BLAST instead finds a highest-scoring alignment path in a narrow band as illustrated below. Since the band is narrow, the algorithm constructing this banded alignment is fast.



How can I modify the approximate pattern matching with the Burrows-Wheeler transform to account for patterns whose last symbols do not appear in *Text*?

The algorithm illustrated in the epilogue would fail to find an approximate match of `"nad"` because the final symbol of `"nad"` does not appear in `"panamabananas$"`. To address this complication, we can modify the algorithm for finding a pattern of length *m* with up to *k* mismatches as follows.

We first run the algorithm described in the main text to find all approximate instances of a *Pattern* of length *k* against *Text*. However, this algorithm does not actually find all approximate matches of *Pattern* – since we do not allow mismatched strings in the early stages of BetterBWMatching, we miss those matches where the last letter of *Pattern* does not match *Text*. To fix this shortcoming, we can simply find all locations in *Text* where the prefix of *Pattern* of length *k* - 1 has *d* - 1 mismatches. Yet this algorithm fails to find matches where the last two letters of *Pattern* do not match *Text*. Thus, we need to run the algorithm again, finding all locations in *Text* where the prefix of *Pattern* of length *k* - 2 has *d* - 2 mismatches. We then find all locations in *Text* where the prefix of *Pattern* of length *k* - 3 occurs with *d* - 3 mismatches, and so on, finally finding all locations in *Text* where the prefix of *Pattern* of length *k* - *d* occurs exactly.

When we approximately match patterns with the Burrows-Wheeler transform, we consider possibilities of mismatches in all positions but the first one. Wouldn't this strategy fail to match a read with an error at the first position?

Yes, this strategy would fail to match a read with an error at the first position. However, as noted in the main text, if we start considering mismatches at the first position, the running time will significantly increase. As is, the running time explodes with the increase in the maximum number of errors. If one wants to alow mismatches at the first position, a more sensible strategy would be to trim the first position of the read.

To see how modern read mapping algorithms get around this limitation, see B.Langmead, C. Trapnell, M. Pop, S. L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to human genome (2009) *Genome biology*, 10, R25.

Why do we add both the # and $ symbols to the texts when solving the Longest Shared Substring Problem?

Consider the strings "mapa" and "pepap", with longest shared substring “pa”. If we use both “#” and “$”, then we construct the suffix tree for "mapa#pepap$", which reveals the longest shared substring “pa”. If we do not use “#”, then we construct the suffix tree for "mapapepap$", falsely concatenating the two strings. In this suffix tree, we would find the erroneous longest shared substring “pap” in the concatenated string "mapapepap$".

How should I modify BinarySearch if I want to determine how many times a key is present in an array with duplicates?

If *key* appears in *Array[minIndex,…, maxIndex]*, BinarySearchTop and BinarySearchBottom find the indexes of its first and last occurrences, respectively. Note that in the case of BinarySearchTop, *key* >*Array(i)* for all *i* < *minIndex* and *key* ≤ *Array(i)* for all *i*>*maxIndex*. In contrast, in the case of BinarySearchBottom, *key* ≥ *Array(i)* for all *i*<*minIndex* and *key* < *Array(i)* for all *i*>*maxIndex*.

```
BinarySearchTop(Array, key, minIndex, maxIndex)
  while maxIndex ≥ minIndex
    midIndex = (minIndex + maxIndex)/2
    if Array(midIndex) ≥ key
      maxIndex = mid-1
    else
      minIndex = mid+1
  return minIndex
```

```
BinarySearchBottom(Array, key, minIndex, maxIndex)
  while maxIndex ≥ minIndex
    midIndex = (minIndex + maxIndex)/2
    if Array(midIndex) > key
      maxIndex = mid-1
    else
      minIndex = mid+1
  return minIndex
```


ISBN: 978-0-9903746-3-3