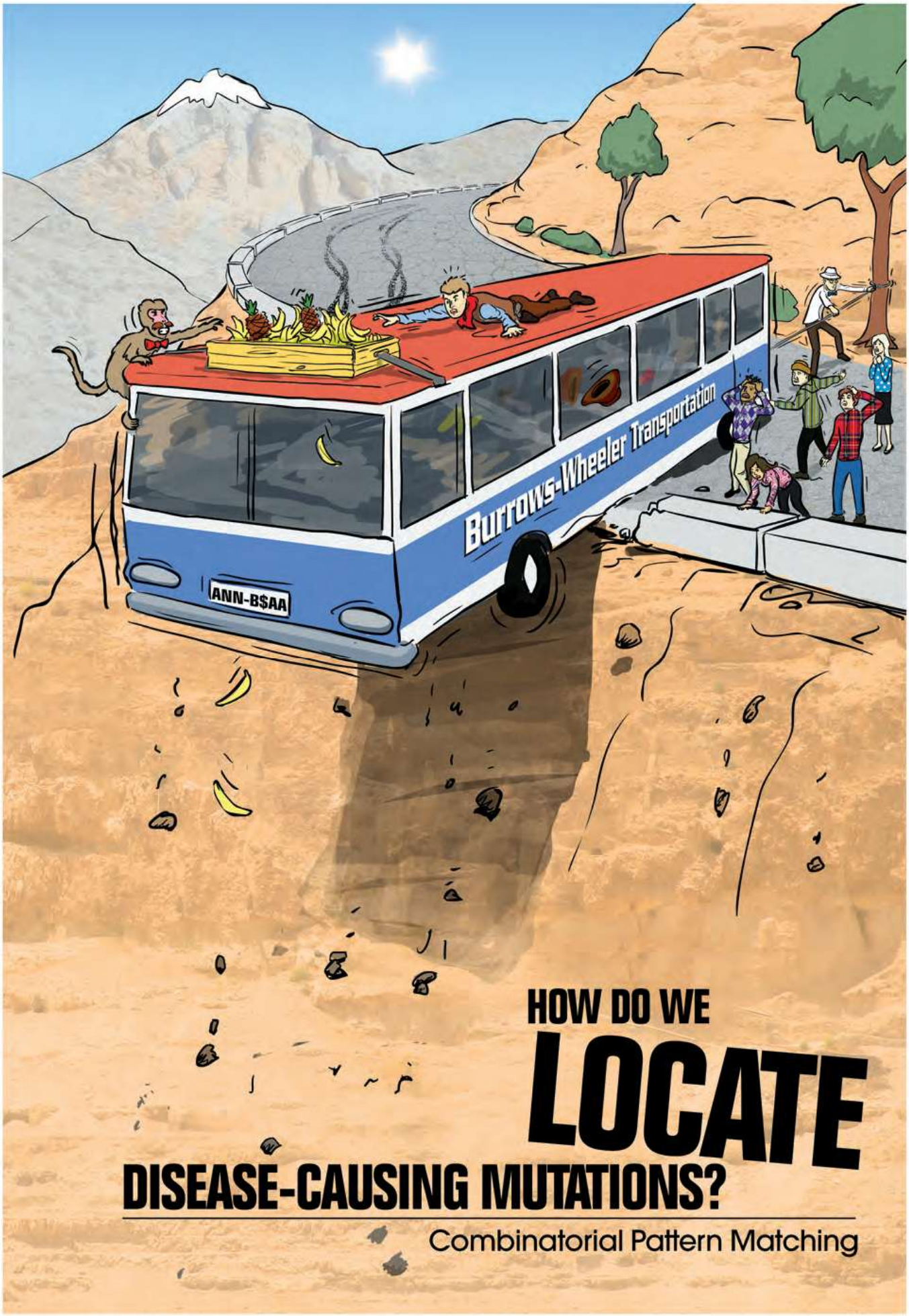

HOW DO WE LOCATE DISEASE-CAUSING MUTATIONS?

Combinatorial Pattern Matching

What Causes Ohdo Syndrome?

About 1% of babies are born with mental retardation, but this affliction remains poorly understood because it can be caused by a variety of different genetic disorders. One of these disorders is **Ohdo syndrome**, which causes an expressionless, "mask-like" face. In 2011, biologists solved the genetic puzzle underlying Ohdo syndrome by discovering a handful of mutations shared by multiple patients, which the researchers used to identify a single protein-truncating mutation responsible for Ohdo syndrome.

The discovery of Ohdo syndrome's root cause represents just one of many new discoveries arising from the use of **read mapping** to study genetic disorders. In read mapping, researchers compare sequenced DNA reads taken from an individual against a **reference human genome** (see DETOUR: The Reference Human Genome) in order to find which reads perfectly match the reference and which reads indicate mutations of one nucleotide into another (**single nucleotide polymorphisms**, or **SNPs**). The reference genome is a gross simplification of species identity, since in addition to about 3 million SNPs (0.1% of the human genome), humans differ by genome rearrangements, insertions, and deletions that can span thousands of nucleotides (see DETOUR: Rearrangements, Insertions, and Deletions in Human Genomes). However, in this chapter we will focus only on algorithms for finding SNPs.

PAGE 170

PAGE 170

But wait, you may say, *why not use one of the algorithms that we already covered?* After all, we could always sequence the entire genome of an individual and then compare it against the reference genome. However, sequencing methods are computationally intensive and not perfect, as they often generate error-prone contigs. As a result, it makes sense to map reads from an individual human to the reference human genome to find out the differences.

To see why read mapping should be easier than genome assembly, let us return to the analogy of a jigsaw puzzle, which is sold with a picture of the completed puzzle on its box. This photo makes reconstructing the puzzle far easier; for a simple example, if the completed puzzle shows a sun in a blue sky, then you can automatically move all of the bright yellow pieces and all of the light blue pieces to the top of the puzzle.

Yet aside from genome sequencing, two other methods come to mind for read mapping. First, you could align each read to the reference genome (using a fitting alignment, Chapter 5) to find the most similar region. Second, you could apply approximate pattern matching algorithms to match each read one at a time against the reference genome.

STOP and Think: What computational challenges might arise from using these methods to map millions of reads to a reference human genome?

Both of these methods are guaranteed to solve the problem of mapping reads to a reference genome, but their runtimes become a bottleneck when we scale to millions of reads. Therefore, our goal in this chapter is to figure out how to use the reference genome as a "photo on the box" shortcut to find SNPs.

Introduction to Multiple Pattern Matching

Recall from Chapter 3 that reads are typically a few hundred base pairs long. These reads will form a collection of strings *Patterns* that we wish to match against a genome *Text*. For each string in *Patterns*, we will first find all its *exact* matches as a substring of *Text* (or conclude that it does not appear in *Text*). When hunting for the cause of a genetic disorder, we can immediately eliminate from consideration areas of the reference genome where exact matches occur. In the epilogue, we will generalize this problem to find *approximate* matches, where single nucleotide substitutions in reads separate the individual from the reference genome (or represent errors in reads).

Multiple Pattern Matching Problem:
Find all occurrences of a collection of patterns in a text.

Input: A string *Text* and a collection *Patterns* containing (shorter) strings.
Output: All starting positions in *Text* where a string from *Patterns* appears as a substring.

A naive approach to the Multiple Pattern Matching Problem would attempt repeated applications of an algorithm for the (single) Pattern Matching Problem, which we encountered in Chapter 1. This algorithm, which we call BRUTEFORCEPATTERNMATCHING, would slide each *Pattern* along *Text*, checking whether the substring starting at each position of *Text* matches *Pattern*. Recall that the runtime of a naive algorithm for a single pattern is $\mathcal{O}(|Text| \cdot |Pattern|)$. Thus, the runtime of BRUTEFORCEPATTERNMATCHING for the Multiple Pattern Matching Problem is $\mathcal{O}(|Text| \cdot |Patterns|)$, where $|Text|$ is the length of *Text* and $|Patterns|$ is the sum of the lengths of all strings in *Patterns*.

The problem with applying BRUTEFORCEPATTERNMATCHING to read mapping is that $|Text|$ and $|Patterns|$ are both huge. In the case of the human genome (3 GB), the total length of all reads may exceed 1 TB; as a result, any algorithm with runtime $\mathcal{O}(|Text| \cdot |Patterns|)$ will be too slow.

STOP and Think: The estimate $\mathcal{O}(|Text| \cdot |Patterns|)$ presents the *worst-case* estimate of the runtime for **BRUTEFORCEPATTERNMATCHING**. What is the *average-case* estimate?

Herding Patterns into a Trie

Constructing a trie

The reason why the runtime of **BRUTEFORCEPATTERNMATCHING** is so high is that each string in *Patterns* must traverse all of *Text* independently. If you think about *Text* as a long road, then **BRUTEFORCEPATTERNMATCHING** is analogous to loading each pattern into its own car when driving down *Text*, an inefficient strategy. Instead, our goal is to herd the patterns onto a bus so that we only need to make one trip from the beginning to the end of *Text*. In more formal terms, we would like to organize *Patterns* into a data structure to prevent multiple passes down *Text* and to reduce the runtime. To this end, we will consolidate *Patterns* into a directed acyclic graph called a **trie** (pronounced "try"), which is written TRIE(*Patterns*) and has the following properties (Figure 9.1).

- The trie has a single root node with indegree 0, denoted *root*.
- Each edge of TRIE(*Patterns*) is labeled with a letter of the alphabet.
- Edges leading out of a given node have distinct labels.
- Every string in *Patterns* is spelled out by concatenating the letters along some path from the root downward.
- Every path from the root to a **leaf**, or node with outdegree 0, spells a string from *Patterns*.

Trie Construction Problem:
Construct a trie from a collection of patterns.

Input: A collection of strings *Patterns*.
Output: TRIE(*Patterns*).

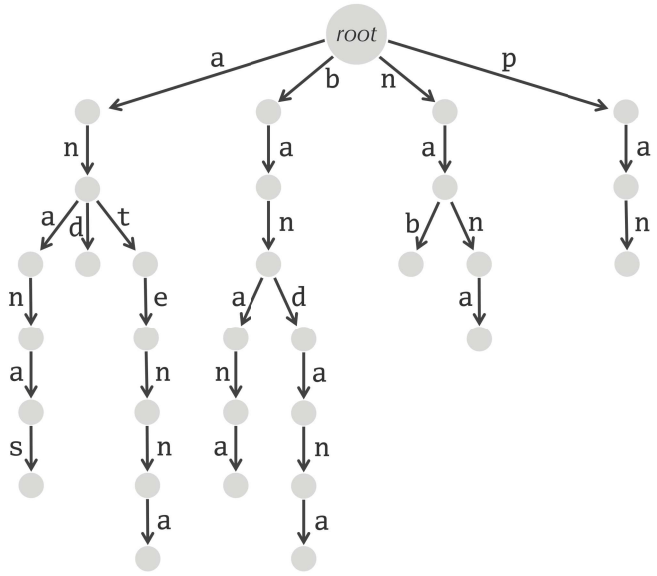

FIGURE 9.1 The trie for the following collection of strings *Patterns*: "ananas", "and", "antenna", "banana", "bandana", "nab", "nana", "pan".

The most obvious way to construct TRIE(*Patterns*) is by iteratively adding each string from *Patterns* to the growing trie, as implemented by the following algorithm.


```
TRIECONSTRUCTION(Patterns)
    Trie ← a graph consisting of a single node root
    for each string Pattern in Patterns
        currentNode ← root
        for i ← 1 to |Pattern|
            currentSymbol ← i-th symbol of Pattern
            if there is an outgoing edge from currentNode with label currentSymbol
                currentNode ← ending node of this edge
            else
                add a new node newNode to Trie
                add a new edge from currentNode to newNode with label currentSymbol
                currentNode ← newNode
    return Trie
```

STOP and Think: How can we use the trie to solve the Multiple Pattern Matching Problem?

Applying the trie to multiple pattern matching

Given a string *Text* and TRIE(*Patterns*), we can quickly check whether any string from *Patterns* matches a *prefix* of *Text*. To do so, we start reading symbols from the beginning of *Text* and see what string these symbols "spell" as we proceed along the path downward from the root of the trie, as illustrated in Figure 9.2 (left). For each new symbol in *Text*, if we encounter this symbol along an edge leading down from the present node, then we continue along this edge; otherwise, we stop and conclude that no string in *Patterns* matches a prefix of *Text*. If we make it all the way to a leaf, then the pattern spelled out by this path matches a prefix of *Text*. This algorithm is called **PREFIXTRIEMATCHING**.

```
PREFIXTRIEMATCHING(Text, Trie)
    symbol ← first letter of Text
    v ← root of Trie
    while forever
        if v is a leaf in Trie
            return the pattern spelled by the path from the root to v
        else if there is an edge (v, w) in Trie labeled by symbol
            symbol ← next letter of Text
            v ← w
        else
            output "no matches found"
            return
```

STOP and Think: For **PREFIXTRIEMATCHING** to work, we have made a hidden assumption that no string in *Patterns* is a prefix of another string in *Patterns*. How can this algorithm be modified when *Patterns* is an arbitrary collection of strings? Hint: consider adding `"pantry"` to the patterns in Figure 9.2 (left).

PREFIXTRIEMATCHING finds whether any strings in *Patterns* match a prefix of *Text*. To find whether any strings in *Patterns* match a substring of *Text* starting at position k, we chop off the first $k-1$ symbols from *Text* and run **PREFIXTRIEMATCHING** on the shortened string. As a result, to solve the Multiple Pattern Matching Problem, we simply iterate **PREFIXTRIEMATCHING** $|Text|$ times, chopping the first symbol off of *Text* before each new iteration (Figure 9.2 (right)).

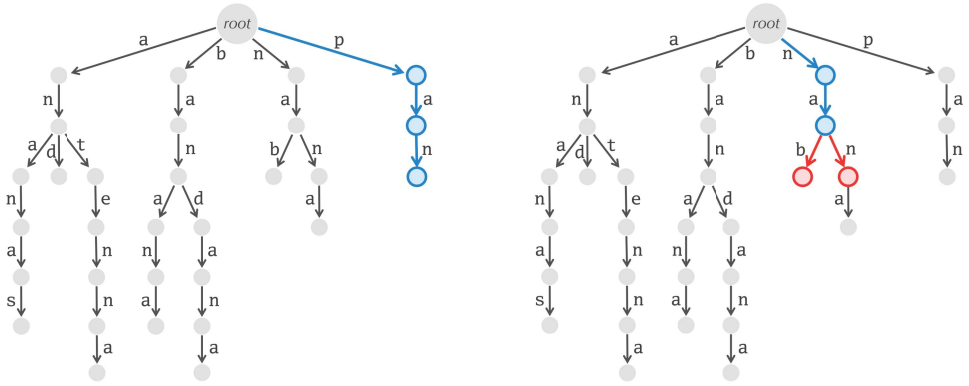

FIGURE 9.2 (Left) The pattern "pan" matches *Text* = "panamabananas" when starting at the beginning of *Text*. (Right) No pattern match is found in TRIE(*Patterns*) for the strings *Patterns* from Figure 9.1 when starting at the third symbol of "panamabananas".

TRIEMATCHING(*Text*, *Trie*)
  while *Text* is nonempty
    PREFIXTRIEMATCHING(*Text*, *Trie*)
    remove first symbol from *Text*

We need $|Patterns|$ steps to construct TRIE(*Patterns*), which contains at most $|Patterns|$ nodes. Each iteration of **PREFIXTRIEMATCHING** takes at most $|LongestPattern|$ steps, where *LongestPattern* is the longest string in *Patterns*. **TRIEMATCHING** makes $|Text|$ total calls to **PREFIXTRIEMATCHING**, making the total number of steps equal to $|Patterns| + |Text| \cdot |LongestPattern|$. This runtime offers a huge speed-up compared to the $|Text| \cdot |Patterns|$ steps required by **BRUTEFORCEPATTERNMATCHING**. The **Aho-Corasick algorithm**, developed in 1975, further reduces the number of steps required after constructing the trie from $\mathcal{O}(|Text| \cdot |LongestPattern|)$ steps to $\mathcal{O}(|Text|)$ steps (see DETOUR: The Aho-Corasick Algorithm).

STOP and Think: Do you see any computational challenges with using **TRIEMATCHING** to solve the Multiple Pattern Matching Problem?

Although **TRIEMATCHING** is fast, storing a trie consumes a lot of memory. Recall that **BRUTEFORCEPATTERNMATCHING** works with a single read at a time, which keeps the memory low because we only need to store the genome in memory. Yet **TRIEMATCHING** needs to store the entire trie in memory, which is proportional to

|*Patterns*|. Since a collection of reads for the human genome may consume upwards of 1 TB, the memory required to store the trie is prohibitive.

STOP and Think: How can we avoid multiple passes through the genome without needing to consolidate all the reads into a huge data structure?

Preprocessing the Genome Instead

Introduction to suffix tries

Since storing TRIE(*Patterns*) requires so much memory, let's process *Text* into a data structure instead. Our goal is to compare each string in *Patterns* against *Text* without needing to traverse *Text* from beginning to end. In more familiar terms, instead of packing *Patterns* onto a bus and riding the long distance down *Text*, our new data structure will be able to "teleport" each string in *Patterns* directly to its occurrences in *Text*.

A **suffix trie**, denoted SUFFIXTRIE(*Text*), is the trie formed from all suffixes of *Text* (Figure 9.3). From now on, we append the dollar-sign ("$") to *Text* in order to mark the end of *Text*. We will also label each leaf of the resulting trie by the starting position of the suffix whose path through the trie ends at this leaf (using 0-based indexing). This way, when we arrive at a leaf, we will immediately know where this suffix came from in *Text*.

STOP and Think: How can we use the suffix trie for pattern matching?

Using suffix tries for pattern matching

To match a single string *Pattern* to *Text*, note that if *Pattern* matches a substring of *Text* starting at position i, then *Pattern* must also appear at the beginning of the suffix of *Text* starting at position i. We can therefore determine whether *Pattern* occurs in SUFFIXTRIE(*Text*) by starting at the root and spelling symbols of *Pattern* downward. If we can find a path in the suffix trie spelling out *Pattern*, then we know that *Pattern* must occur in *Text* (Figure 9.4). We can then iterate over all strings in *Patterns*.

STOP and Think: Figure 9.4 illustrates how to find the pattern "nanas" in SUFFIXTRIE("panamabananas$"), but it does not tell us where "nanas" occurs in *Text*. How can we obtain this information?

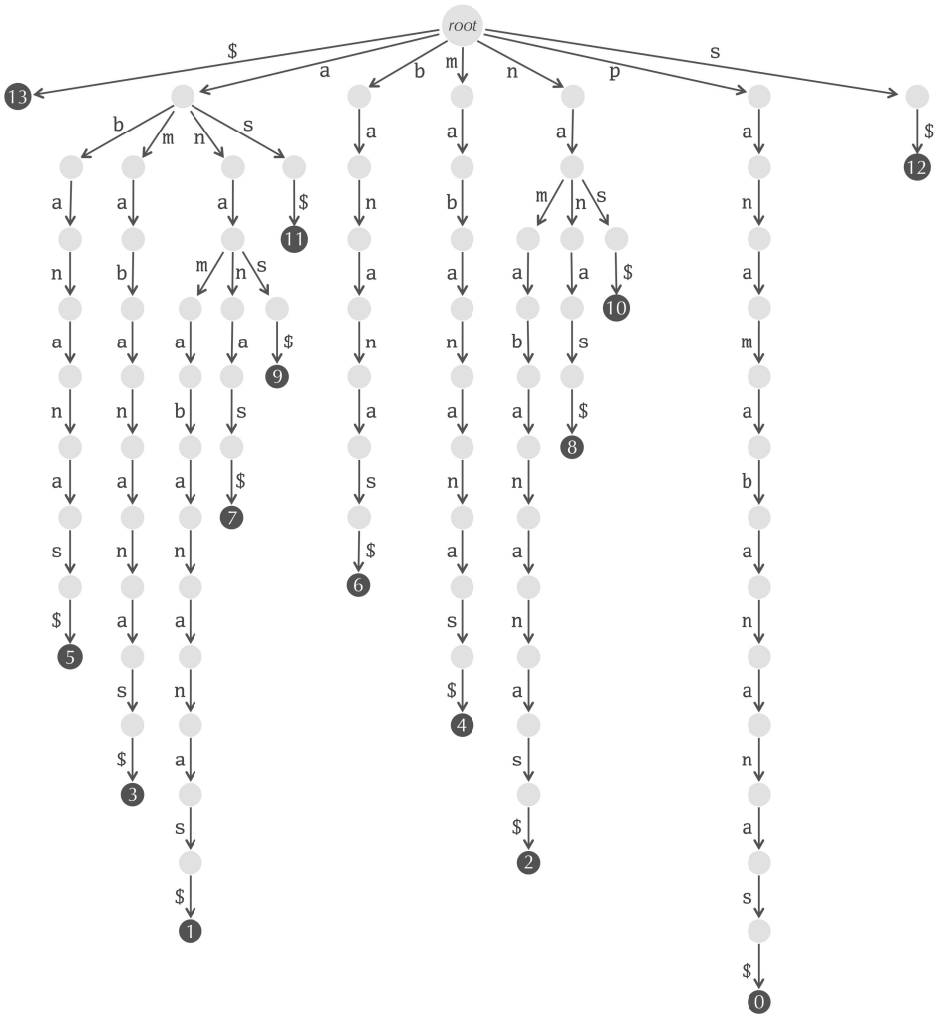

FIGURE 9.3 SUFFIXTRIE("panamabananas$"), with leaf labels (corresponding to starting positions of suffixes) varying from 0 to 13.

To determine where *Pattern* appears in *Text*, assume first that *Pattern* matches *Text* at a leaf of SUFFIXTRIE(*Text*). In this case, *Pattern* must appear in *Text* as a suffix, and we can consult the label at that leaf to determine the starting position of the suffix. For example, threading "nanas" into the suffix trie in Figure 9.4 shows that it matches the suffix starting at position 8 of "panamabananas$".

If the path spelling out *Pattern* stops before a leaf at some node v of SUFFIXTRIE(*Text*), then *Pattern* may occur more than once in *Text*. To locate these occurrences, follow all paths from v down to the leaves of SUFFIXTRIE(*Text*), which will indicate all starting

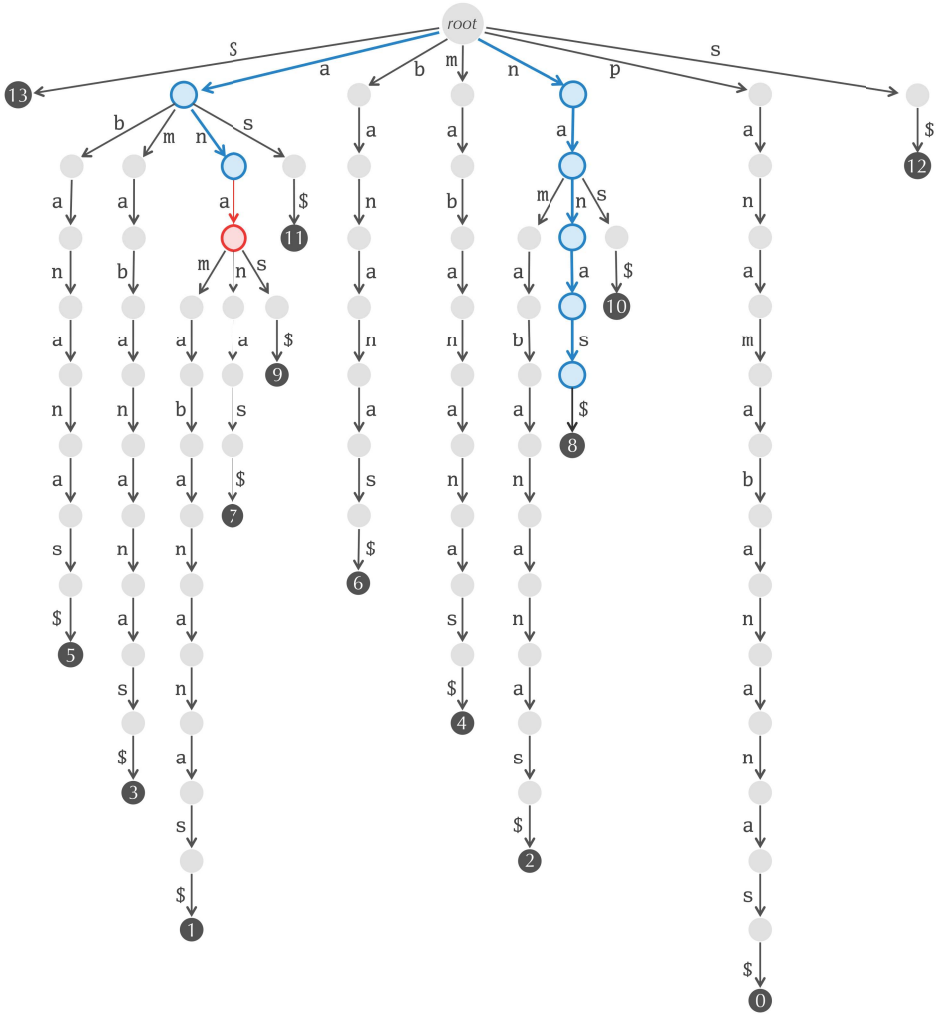

FIGURE 9.4 Threading "antenna" through SUFFIXTRIE("panamabananas$") fails to find a match because no suffix of "panamabananas$" begins with "ant"; however, threading "nana" through the suffix trie does find a match: "panamabananas$".

positions of *Pattern* in *Text*. For example, the pattern "ana" corresponds to a path in SUFFIXTRIE("panamabananas$") that can be extended to three different leaves with labels 1, 7, and 9, corresponding to three occurrences of "ana": "panamabananas$", "panamabananas$", and "panamabananas$".

STOP and Think: How much runtime and memory will it take to construct SUFFIXTRIE(*Text*)?

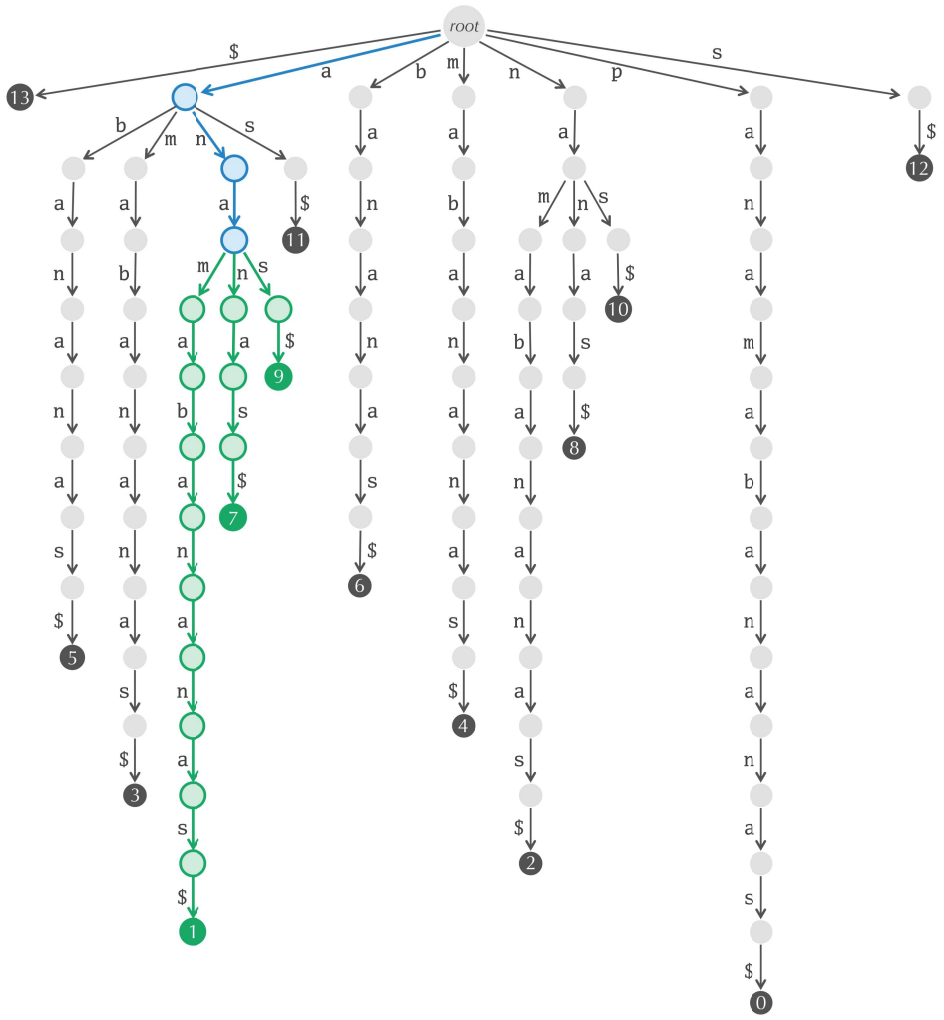

FIGURE 9.5 All paths starting with "ana" reveal the three occurrences of "ana" in "panamabanana$" . Extending these paths to the leaves (shown in green) reveals that the starting positions of these occurrences are 1, 7, and 9.

Recall that constructing TRIE(*Patterns*) required $\mathcal{O}(|Patterns|)$ runtime and memory. Accordingly, the runtime and memory required to construct SUFFIXTRIE(*Text*) are both equal to the combined length of all suffixes in *Text*. There are $|Text|$ suffixes of *Text*, ranging in length from 1 to $|Text|$ and having total length $|Text| \cdot (|Text| + 1)/2$, which is $\mathcal{O}(|Text|^2)$. Thus, we need to reduce both the construction time and memory requirements of suffix tries to make them practical.

Suffix Trees

Let's not give up hope on suffix tries. We can reduce the number of edges in the suffix trie by combining the edges on any non-branching path into a single edge. We then label this edge with the concatenation of symbols on the consolidated edges, as shown in Figure 9.6. The resulting data structure is called a **suffix tree**, written SUFFIXTREE(*Text*).

To match a single *Pattern* to *Text*, we thread *Pattern* into SUFFIXTREE(*Text*) by the same process used for a suffix trie. Similarly to the suffix trie, we can use the leaf labels to find starting positions of successfully matched patterns.

EXERCISE BREAK: Prove that SUFFIXTREE(*Text*) has exactly $|Text| + 1$ leaves and at most $|Text| + 1$ other nodes.

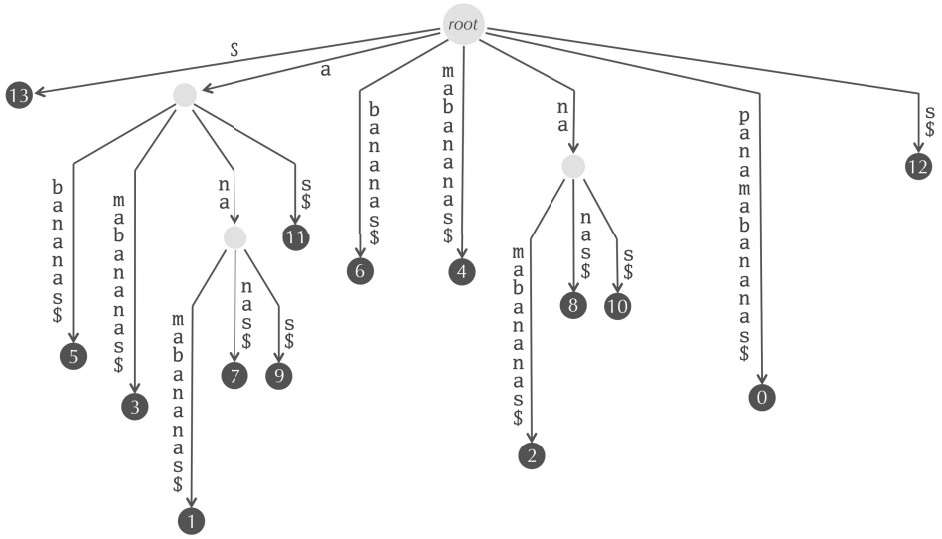

FIGURE 9.6 SUFFIXTREE("panamabananas$"), formed by compressing the edges on non-branching paths in Figure 9.3.

Compared to a suffix trie, which may have a quadratic number of nodes in the length of *Text*, the number of nodes in SUFFIXTREE(*Text*) is at most $2 \cdot |Text|$. Therefore, the memory required for SUFFIXTREE(*Text*) is $\mathcal{O}(|Text|)$.

STOP and Think: Wait a second. Suffix trees seem like a cosmetic modification of suffix tries. Since we still need to keep all concatenated edge labels in memory, why should suffix trees be more memory-efficient than suffix tries?

Suffix trees save memory because they do not need to store concatenated edge labels from each non-branching path. For example, a suffix tree does not need ten bytes to store the edge labeled "mabananas$" in Figure 9.6; instead, it suffices to store a **pointer** to position 4 of "panamabananas$", as well as the *length* of "mabananas$". Furthermore, suffix trees can be constructed in linear time, without having to first construct the suffix trie! We will not ask you to implement this fast suffix tree construction algorithm because it is quite complex.

Suffix Tree Construction Problem:
Construct the suffix tree of a string.

Input: A string *Text*.
Output: SUFFIXTREE(*Text*).

CHARGING STATION (Constructing a Suffix Tree): A memory-inefficient way of constructing the suffix tree is to first construct the suffix trie and then consolidate each non-branching path into a single edge, storing its label in memory. To implement a more memory-efficient solution, check out this Charging Station.

Although the suffix tree decreases memory requirements from $\mathcal{O}(|Text|^2)$ to $\mathcal{O}(|Text|)$, on average it still requires about 20 times as much memory as *Text*. In the case of a 3 GB human genome, 60 GB of RAM is a huge improvement over the 1 TB that we needed to work with TRIE(*Patterns*), but it still presents a memory challenge for most machines. This reveals a dark secret of big-O notation, which is that it ignores constant factors. For long strings such as the human genome, we will need to pay attention to this constant factor, since the expression $\mathcal{O}(|Text|)$ applies to both an algorithm with $2 \cdot |Text|$ memory and an algorithm with $1000 \cdot |Text|$ memory.

Yet before seeing how we can further reduce the memory needed for multiple pattern matching, we ask you to solve three problems for which suffix trees are useful.

Longest Repeat Problem:
Find the longest repeat in a string.

Input: A string *Text*.
Output: A longest substring of *Text* that appears in *Text* more than once.

Longest Shared Substring Problem:
Find the longest substring shared by two strings.

Input: Strings $Text_1$ and $Text_2$.
Output: The longest substring that occurs in both $Text_1$ and $Text_2$.

Shortest Non-Shared Substring Problem:
Find the shortest substring of one string that does not appear in another string.

Input: Strings $Text_1$ and $Text_2$.
Output: The shortest substring of $Text_1$ that does not appear in $Text_2$.

CHARGING STATION (Solving the Longest Shared Substring Problem): One way of solving the Longest Shared Substring Problem is to construct two suffix trees, one for $Text_1$ and one for $Text_2$. Check out this Charging Station to learn about a more elegant solution.

Suffix Arrays

Constructing a suffix array

In 1993, Udi Manber and Gene Myers introduced **suffix arrays** as a memory-efficient alternative to suffix trees. To construct SUFFIXARRAY(*Text*), we first sort all suffixes of *Text* lexicographically, assuming that "$" comes first in the alphabet (Figure 9.7). The suffix array is the list of starting positions of these sorted suffixes:

SUFFIXARRAY("panamabananas$") = [13, 5, 3, 1, 7, 9, 11, 6, 4, 2, 8, 10, 0, 12].

Suffix Array Construction Problem:
Construct the suffix array of a string.

Input: A string *Text*.
Output: SUFFIXARRAY(*Text*).

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

FIGURE 9.7 The sorted list of suffixes of *Text* = "panamabananas$", along with their starting positions in *Text*; these starting positions form the suffix array of *Text*.

The Suffix Array Construction Problem can easily be solved after sorting all suffixes of *Text*, but since even the fastest algorithms for sorting an array of n elements require $\mathcal{O}(n \cdot \log n)$ comparisons, sorting all suffixes takes $\mathcal{O}(|Text| \cdot \log(|Text|))$ comparisons. However, there exists a faster algorithm that constructs suffix arrays in linear time and requires only about a fifth as much memory as suffix trees, which knocks the 60 GB memory requirement for the human genome down to 12 GB.

STOP and Think: Given a suffix tree, can you quickly transform it into a suffix array? Given a suffix array, can you quickly transform it into a suffix tree?

As the preceding question suggests, suffix arrays and suffix trees are practically equivalent; every algorithm using suffix trees can be translated into an algorithm using suffix arrays (see **DETOUR: From Suffix Trees to Suffix Arrays**), and vice-versa (see **DETOUR: From Suffix Arrays to Suffix Trees**).

PAGE 171

PAGE 173

Pattern matching with the suffix array

Once we have constructed the suffix array of a string *Text*, we can use it to quickly locate every occurrence of a string *Pattern* in *Text*. First, recall that when pattern matching with the suffix trie, we observed that all matches of *Pattern* in *Text* must occur at the beginning of suffixes of *Text*. Second, note that after sorting the suffixes of *Text*, the

suffixes beginning with *Pattern* clump together. For example, in Figure 9.7, *Pattern* = "ana" occurs at the beginning of the suffixes "anamabananas$", "ananas$", and "anas$" of *Text* = "panamabananas$"; these suffixes occur in three consecutive rows and correspond to the starting positions 1, 7, and 9 in *Text*.

The question is how to find these starting positions for an arbitrary string *Pattern* without needing to store the sorted suffixes of *Text*. The following algorithm, called **PATTERNMATCHINGWITHSUFFIXARRAY**, identifies the first and last index of the suffix array corresponding to suffixes beginning with *Pattern* (these indices are denoted *first* and *last*, respectively). **PATTERNMATCHINGWITHSUFFIXARRAY** offers a variation of a general search technique called **binary search** that finds a data point in a sorted collection of data by iteratively dividing the data in half and determining the half in which the data point lies. See **DETOUR: Binary Search** for more details.

PAGE 176

```
PATTERNMATCHINGWITHSUFFIXARRAY(Text, Pattern, SUFFIXARRAY)
    minIndex ← 0
    maxIndex ← |Text|
    while minIndex < maxIndex
        midIndex ← (minIndex + maxIndex) / 2
        if Pattern > suffix of Text starting at position SUFFIXARRAY(midIndex)
            minIndex ← midIndex + 1
        else
            maxIndex ← midIndex
    first ← minIndex
    maxIndex ← |Text|
    while minIndex < maxIndex
        midIndex ← (minIndex + maxIndex) / 2
        if Pattern < suffix of Text starting at position SUFFIXARRAY(midIndex)
            maxIndex ← midIndex
        else
            minIndex ← midIndex + 1
    last ← maxIndex
    if first > last
        return "Pattern does not appear in Text"
    else
        return (first, last)
```

The Burrows-Wheeler Transform

Genome compression

Suffix arrays have greatly reduced the memory required for efficient text searches, and until the start of this century, they represented the state of the art in pattern matching. Can we be so ambitious as to look for a data structure that would encode *Text* using memory approximately equal to the length of *Text* while still enabling fast pattern matching?

To answer this question, we will digress to consider the seemingly unrelated topic of **text compression**. In one simple compression technique called **run-length encoding**, we replace a **run** of k consecutive occurrences of symbol s with only two symbols: k, followed by s. For example, run-length encoding would compress the string `TTTTTGGGAAAACCCCCCA` into `5T3G4A6C1A`.

Run-length encoding works well for strings having lots of long runs, but real genomes do not have many runs. What they do have, as we saw in Chapter 3, are *repeats*. It would therefore be nice if we could first manipulate the genome to convert repeats into runs and then apply run-length encoding to the resulting string.

A naive way of creating runs in a string is to reorder the string's symbols lexicographically. For example, `TACGTAACGATACGAT` would become `AAAAACCCGGGTTTT`, which we could then compress into `5A3C3G4T`. This method would represent a 3 GB human genome file using just four numbers.

STOP and Think: What is wrong with applying this compression method to genomes?

Ordering a string's symbols lexicographically is not suitable for compression because many different strings will get compressed into the *same* string. For example, the DNA strings `GCATCATGCAT` and `ACTGACTACTG` — as well as any string with the same nucleotide counts — get reordered into `AAACCCGGTTT`. As a result, we cannot **decompress** the compressed string, i.e., invert the compression operation to produce the original string.

Constructing the Burrows-Wheeler transform

Let's consider a different method of converting the repeats of a string into runs that was proposed by Michael Burrows and David Wheeler in 1994. First, form all possible **cyclic rotations** of *Text*; a cyclic rotation is defined by chopping off a suffix from the end of

Text and appending this suffix to the beginning of *Text*. Next — similarly to suffix arrays — order all the cyclic rotations of *Text* lexicographically to form a $|Text| \times |Text|$ matrix of symbols that we call the **Burrows-Wheeler matrix** and denote by M(*Text*) (Figure 9.8).

Cyclic Rotations	*M*("panamabananas$")
panamabananas$	$ p a n a m a b a n a n a **s**
$panamabananas	a b a n a n a s $ p a n a **m**
s$panamabanana	a m a b a n a n a s $ p a **n**
as$panamabanan	a n a m a b a n a n a s $ **p**
nas$panamabana	a n a n a s $ p a n a m a **b**
anas$panamaban	a n a s $ p a n a m a b a **n**
nanas$panamaba	a s $ p a n a m a b a n a **n**
ananas$panamab	b a n a n a s $ p a n a m **a**
bananas$panama	m a b a n a n a s $ p a n **a**
abananas$panam	n a m a b a n a n a s $ p **a**
mabananas$pana	n a n a s $ p a n a m a b **a**
amabananas$pan	n a s $ p a n a m a b a n **a**
namabananas$pa	p a n a m a b a n a n a s **$**
anamabananas$p	s $ p a n a m a b a n a n **a**

FIGURE 9.8 All cyclic rotations of "panamabananas$" (left) and the Burrows-Wheeler matrix M("panamabananas$") of all lexicographically ordered cyclic rotations (right). BWT("panamabananas$") is the last column of M("panamabananas$"): "**smnpbnnaaaaa$a**".

Notice that the first column of M(*Text*) contains the symbols of *Text* ordered lexicographically, which is just the naive rearrangement of *Text* that we already described. In turn, the second column of M(*Text*) contains the second symbols of all cyclic rotations of *Text*, and so it too represents a (different) rearrangement of symbols from *Text*. The same reasoning applies to show that any column of M(*Text*) is some rearrangement of the symbols of *Text*. We are interested in the last column of M(*Text*), called the **Burrows-Wheeler transform** of *Text*, or BWT(*Text*), which is shown in red in Figure 9.8.

STOP and Think: We have seen that the first column of M(*Text*) cannot be uniquely decompressed to yield *Text*. Do you think that some other column of M(*Text*) can be inverted to yield *Text*?

Burrows-Wheeler Transform Construction Problem:
Construct the Burrows-Wheeler transform of a string.

Input: A string *Text*.
Output: BWT(*Text*).

STOP and Think: Figure 9.8 suggests a simple algorithm for computing BWT(*Text*) based on constructing M(*Text*). Can you construct BWT(*Text*) using less memory given *Text* and SUFFIXARRAY(*Text*)?

From repeats to runs

If we re-examine the Burrows-Wheeler transform in Figure 9.8, we immediately notice that it has created the run `"aaaaa"` in BWT(`"panamabananas$"`) = `"smnpbnnaaaaa$a"`.

STOP and Think: Why do you think that the Burrows-Wheeler Transform produced this run?

Imagine that we take the Burrows-Wheeler transform of Watson and Crick's 1953 paper on the double helix structure of DNA. The word `"and"` is repeated often in English, which means that when we form all possible cyclic rotations of the Watson & Crick paper, we will witness a large number of rotations beginning with `"and..."` In turn, we will observe many rotations that begin with `"nd..."` and end with `"...a"`. When all the cyclic rotations of *Text* are sorted lexicographically to form M(*Text*), all rows that begin with `"nd..."` and end with `"...a"` will tend to clump together. As illustrated in Figure 9.9, this clumping produces runs of `"a"` in the final column of M(*Text*), which we know is BWT(*Text*).

The substring `"ana"` in `"panamabananas$"` plays the role of `"and"` in Watson and Crick's paper and explains three of the five occurrences of `"a"` in the repeat `"aaaaa"` in BWT(`"panamabananas$"`) = `"smnpbnnaaaaa$a"`. When the Burrows-Wheeler transform is applied to a genome, it converts the genome's many repeats into runs. As we already suggested, after applying the Burrows-Wheeler transform, we can apply an additional compression method such as run-length encoding in order to further reduce the memory.

```
nd Corey (1).  They kindly made their manuscript availa ..... a
nd criticism, especially on interatomic distances.  We  ..... a
nd cytosine.  The sequence of bases on a single chain d ..... a
nd experimentally (3,4) that the ratio of the amounts o ..... u
nd for this reason we shall not comment on it.  We wish ..... a
nd guanine (purine) with cytosine (pyrimidine).  In oth ..... a
nd ideas of Dr.  M. H. F. Wilkins, Dr.  R. E. Franklin  ..... a
nd its water content is rather high.  At lower water co ..... a
nd pyrimidine bases.  The planes of the bases are perpe ..... a
nd stereochemical arguments.  It has not escaped our no ..... a
nd that only specific pairs of bases can bond together  ..... u
nd the atoms near it is close to Furberg's 'standard co ..... a
nd the bases on the inside, linked together by hydrogen ..... a
nd the bases on the outside.  In our opinion, this stru ..... a
nd the other a pyrimidine for bonding to occur.  The hy ..... a
nd the phosphates on the outside.  The configuration of ..... a
nd the ration of guanine to cytosine, are always very c ..... a
nd the same axis (see diagram).  We have made the usual ..... u
nd their co-workers at King's College, London.  One of  ..... a
```

FIGURE 9.9 A few consecutive rows selected from M(*Text*), where *Text* is Watson and Crick's 1953 paper on the double helix. Rows beginning with "nd..." often end with "...a" because of the common occurrence of the word "and" in English, which causes runs of "a" in BWT(*Text*).

EXERCISE BREAK: There is only one run of length at least 10 in the *E. coli* genome. How many runs of length at least 10 do you find after applying the Burrows-Wheeler transform to the *E. coli* genome?

Inverting the Burrows-Wheeler Transform

A first attempt at inverting the Burrows-Wheeler transform

Before we get ahead of ourselves, remember that compressing a genome does not count for much if we cannot decompress it. In particular, if there exist a pair of genomes that the Burrows-Wheeler transform compresses into the same string, then we will not be able to decompress this string. But it turns out that the Burrows-Wheeler transform is reversible!

STOP and Think: Can you find the (unique) string whose Burrows-Wheeler transform is "enwvpeoseu$llt"? It could be "newtloveslupe$", "elevenplustwo$", "unwellpesovet$", or something else entirely.

Consider the toy example BWT(*Text*) = "ard$rcaaaabb". First, recall that the first column of M(*Text*) is the lexicographic rearrangement of symbols in BWT(*Text*), i.e., "$aaaaabbcdrr". For convenience, we will use the terms *FirstColumn* and *LastColumn* (i.e., BWT(*Text*)) when referring to the first and last columns of M(*Text*), respectively.

We know that the first row of M(*Text*) is the cyclic rotation of *Text* beginning with "$", which occurs at the end of *Text*. Thus, if we determine the first row of M(*Text*), then we can move the "$" to the end of this row and reproduce *Text*. But how do we determine the remaining symbols in this first row, if all we know is *FirstColumn* and *LastColumn*?

```
$ ? ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? b
```

STOP and Think: Using the first and last columns of the Burrows-Wheeler matrix shown above, can you find the first symbol of *Text*?

Note that the first symbol in *Text* must follow "$" in *any* cyclic rotation of *Text*. Because "$" occurs as the fourth symbol of *LastColumn* = "ard$rcaaaabb", we know that if we walk one symbol to the right from the end of the fourth row of M(*Text*), then we will "wrap around" and arrive at the fourth symbol of *FirstColumn*, which is "**a**" in "$aa**a**abbcdrr". Therefore, this "**a**" belongs in the first position of *Text*:

```
$ a ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? b
```

STOP and Think: Which symbol is hiding in the second position of *Text*?

Following the same logic of "wrapping around", the next symbol of *Text* should be the first symbol in a row of M(*Text*) that ends in "a". The only trouble is that five rows end in "a", and we don't know which of them is the correct one! If we guess that this "a" is the seventh symbol of "ard$rcaaaabb", then we obtain "b" in the second position of *Text* (Figure 9.10 (left)). On the other hand, if we guess that this "a" is the ninth symbol of "ard$rcaaaabb", then we obtain "c" in the second position of *Text* (Figure 9.10 (middle)). Finally, if we guess that this "a" is the tenth symbol of "ard$rcaaaabb", then we obtain "d" in the second position of *Text* (Figure 9.10 (right)).

```
$ a b ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? ? b
```

```
$ a c ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? ? b
```

```
$ a d ? ? ? ? ? ? ? ? a
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? d
a ? ? ? ? ? ? ? ? ? ? $
a ? ? ? ? ? ? ? ? ? ? r
a ? ? ? ? ? ? ? ? ? ? c
b ? ? ? ? ? ? ? ? ? ? a
b ? ? ? ? ? ? ? ? ? ? a
c ? ? ? ? ? ? ? ? ? ? a
d ? ? ? ? ? ? ? ? ? ? a
r ? ? ? ? ? ? ? ? ? ? b
r ? ? ? ? ? ? ? ? ? ? b
```

FIGURE 9.10 The three possibilities ("b", "c", or "d") for the third element of the first row of M(*Text*) when BWT(*Text*) is "ard$rcaaaabb". One of these possibilities must correspond to the second symbol of *Text*.

STOP and Think: How would you choose among "b", "c", and "d" for the second symbol of *Text*?

The First-Last Property

To determine the remaining symbols of *Text*, we need to use a subtle property of M(*Text*) that may seem completely unrelated to inverting the Burrows-Wheeler transform. Below, we have indexed the occurrences of each symbol in *FirstColumn* with subscripts according to their order of appearance in this column. When *Text* = "panamabananas$", six instances of "a" appear in *FirstColumn*.

```
$  p a n a m a b a n a n a s
a1 b a n a n a s $ p a n a m
a2 m a b a n a n a s $ p a n
a3 n a m a b a n a n a s $ p
a4 n a n a s $ p a n a m a b
a5 n a s $ p a n a m a b a n
a6 s $ p a n a m a b a n a n
b  a n a n a s $ p a n a m a
m  a b a n a n a s $ p a n a
n  a m a b a n a n a s $ p a
n  a n a s $ p a n a m a b a
n  a s $ p a n a m a b a n a
p  a n a m a b a n a n a s $
s  $ p a n a m a b a n a n a
```

Consider "a_1" in *FirstColumn*, which occurs at the beginning of the cyclic rotation "$a_1$bananas$panam". If we cyclically rotate this string, then we obtain "panam$a_1$bananas$". Thus, "$a_1$" in *FirstColumn* is actually the third occurrence of "a" in "panamabananas$". We can now identify the positions of the other five instances of "a" in "panamabananas$":

$$pa_3na_2ma_1ba_4na_5na_6s\$$$

EXERCISE BREAK: Where are the three instances of "n" from *FirstColumn* (i.e., "n_1", "n_2", and "n_3") located in "panamabananas$"?

To locate "a_1" in *LastColumn*, we need to cyclically rotate the second row of the matrix M("$a_1$bananas$panam"), which results in "bananas$panam$a_1$". This rotation corresponds to the eighth row of M("panamabananas$"):

```
$  p a n a m a b a n a n a s
a1 b a n a n a s $ p a n a m
a2 m a b a n a n a s $ p a n
a3 n a m a b a n a n a s $ p
a4 n a n a s $ p a n a m a b
a5 n a s $ p a n a m a b a n
a6 s $ p a n a m a b a n a n
b  a n a n a s $ p a n a m a1
m  a b a n a n a s $ p a n a
n  a m a b a n a n a s $ p a
n  a n a s $ p a n a m a b a
n  a s $ p a n a m a b a n a
p  a n a m a b a n a n a s $
s  $ p a n a m a b a n a n a
```

EXERCISE BREAK: Where are the other five instances of "a" located in *LastColumn*?

You hopefully saw that *LastColumn* can be recorded as "smnpbnn$a_1a_2a_3a_4a_5$a_6$", as shown in Figure 9.11. Note that the six instances of "a" appear in exactly the same order in *FirstColumn* and *LastColumn*. This observation is not a fluke. On the contrary, it is a principle that holds for any string *Text* and any symbol that we choose.

First-Last Property: *The k-th occurrence of a symbol in FirstColumn and the k-th occurrence of this symbol in LastColumn correspond to the same position of this symbol in Text.*

$	p	a	n	a	m	a	b	a	n	a	n	a	s
a_1	b	a	n	a	n	a	s	$	p	a	n	a	m
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n
b	a	n	a	n	a	s	$	p	a	n	a	m	a_1
m	a	b	a	n	a	n	a	s	$	p	a	n	a_2
n	a	m	a	b	a	n	a	n	a	s	$	p	a_3
n	a	n	a	s	$	p	a	n	a	m	a	b	a_4
n	a	s	$	p	a	n	a	m	a	b	a	n	a_5
p	a	n	a	m	a	b	a	n	a	n	a	s	$
s	$	p	a	n	a	m	a	b	a	n	a	n	a_6

FIGURE 9.11 The six occurrences of "a" occur in the same order in *FirstColumn* as they do in *LastColumn*.

To see why the First-Last Property is true, consider the rows of M("panamabananas$") beginning with "a":

a_1	b	a	n	a	n	a	s	$	p	a	n	a	m
a_2	m	a	b	a	n	a	n	a	s	$	p	a	n
a_3	n	a	m	a	b	a	n	a	n	a	s	$	p
a_4	n	a	n	a	s	$	p	a	n	a	m	a	b
a_5	n	a	s	$	p	a	n	a	m	a	b	a	n
a_6	s	$	p	a	n	a	m	a	b	a	n	a	n

These rows are already ordered lexicographically, so if we chop off the "a" from the beginning of each row, then the remaining strings should still be ordered lexicographically:

```
b a n a n a s $ p a n a m
m a b a n a n a s $ p a n
n a m a b a n a n a s $ p
n a n a s $ p a n a m a b
n a s $ p a n a m a b a n
s $ p a n a m a b a n a n
```

Adding "a" back to the end of each row should not change the lexicographic ordering of these rows:

```
b a n a n a s $ p a n a m a1
m a b a n a n a s $ p a n a2
n a m a b a n a n a s $ p a3
n a n a s $ p a n a m a b a4
n a s $ p a n a m a b a n a5
s $ p a n a m a b a n a n a6
```

But these are just the rows of M("panamabananas$") containing "a" in *LastColumn*! As a result, the *k*-th occurrence of "a" in *FirstColumn* corresponds to the *k*-th occurrence of "a" in *LastColumn*. This argument generalizes for any *symbol* and any string *Text*, which establishes the First-Last property.

Using the First-Last property to invert the Burrows-Wheeler transform

The First-Last Property is interesting, but how can we use it to invert BWT(*Text*) = "ard$rcaaaabb"? Recalling Figure 9.10, let's return to where we were in our attempt to reconstruct the first row of M(*Text*) and index the occurrences of each symbol in *FirstColumn* and *LastColumn*:

```
$1 a ? ? ? ? ? ? ? ? ? ? a1
a1 ? ? ? ? ? ? ? ? ? ? ? r1
a2 ? ? ? ? ? ? ? ? ? ? ? d1
a3 ? ? ? ? ? ? ? ? ? ? ? $1
a4 ? ? ? ? ? ? ? ? ? ? ? r2
a5 ? ? ? ? ? ? ? ? ? ? ? c1
b1 ? ? ? ? ? ? ? ? ? ? ? a2
b2 ? ? ? ? ? ? ? ? ? ? ? a3
c1 ? ? ? ? ? ? ? ? ? ? ? a4
d1 ? ? ? ? ? ? ? ? ? ? ? a5
r1 ? ? ? ? ? ? ? ? ? ? ? b1
r2 ? ? ? ? ? ? ? ? ? ? ? b2
```

The First-Last Property reveals where "a_3" is hiding in *LastColumn*:

$$
\begin{array}{llllllllllll}
\$_1 & \mathbf{a} & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_1 \\
a_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & r_1 \\
a_2 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & d_1 \\
\mathbf{a_3} & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & \$_1 \\
a_4 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & r_2 \\
a_5 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & c_1 \\
b_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_2 \\
b_2 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & \mathbf{a_3} \\
c_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_4 \\
d_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_5 \\
r_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & b_1 \\
r_2 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & b_2
\end{array}
$$

Since we know that "$\mathbf{a_3}$" is located at the end of the eighth row, we can wrap around this row to determine that "$\mathbf{b_2}$" follows "$\mathbf{a_3}$" in *Text*. Thus, the second symbol of *Text* is "**b**", which we can now add to the first row of M(*Text*):

$$
\begin{array}{llllllllllll}
\$_1 & \mathbf{a} & \mathbf{b} & ? & ? & ? & ? & ? & ? & ? & ? & a_1 \\
a_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & r_1 \\
a_2 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & d_1 \\
\mathbf{a_3} & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & \$_1 \\
a_4 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & r_2 \\
a_5 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & c_1 \\
b_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_2 \\
\mathbf{b_2} & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & \mathbf{a_3} \\
c_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_4 \\
d_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & a_5 \\
r_1 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & b_1 \\
r_2 & ? & ? & ? & ? & ? & ? & ? & ? & ? & ? & b_2
\end{array}
$$

In Figure 9.12, we illustrate repeated applications of the First-Last Property to reconstruct more and more symbols from *Text*. Presto — the string that we have been trying to reconstruct is "abracadabra$".

EXERCISE BREAK: Reconstruct the string whose Burrows-Wheeler transform is "enwvpeoseu$llt".

STOP and Think: Can *any* string (having a single "$" symbol) be inverted using the inverse Burrows-Wheeler transform?

You are now ready to implement the inverse of the Burrows-Wheeler transform.

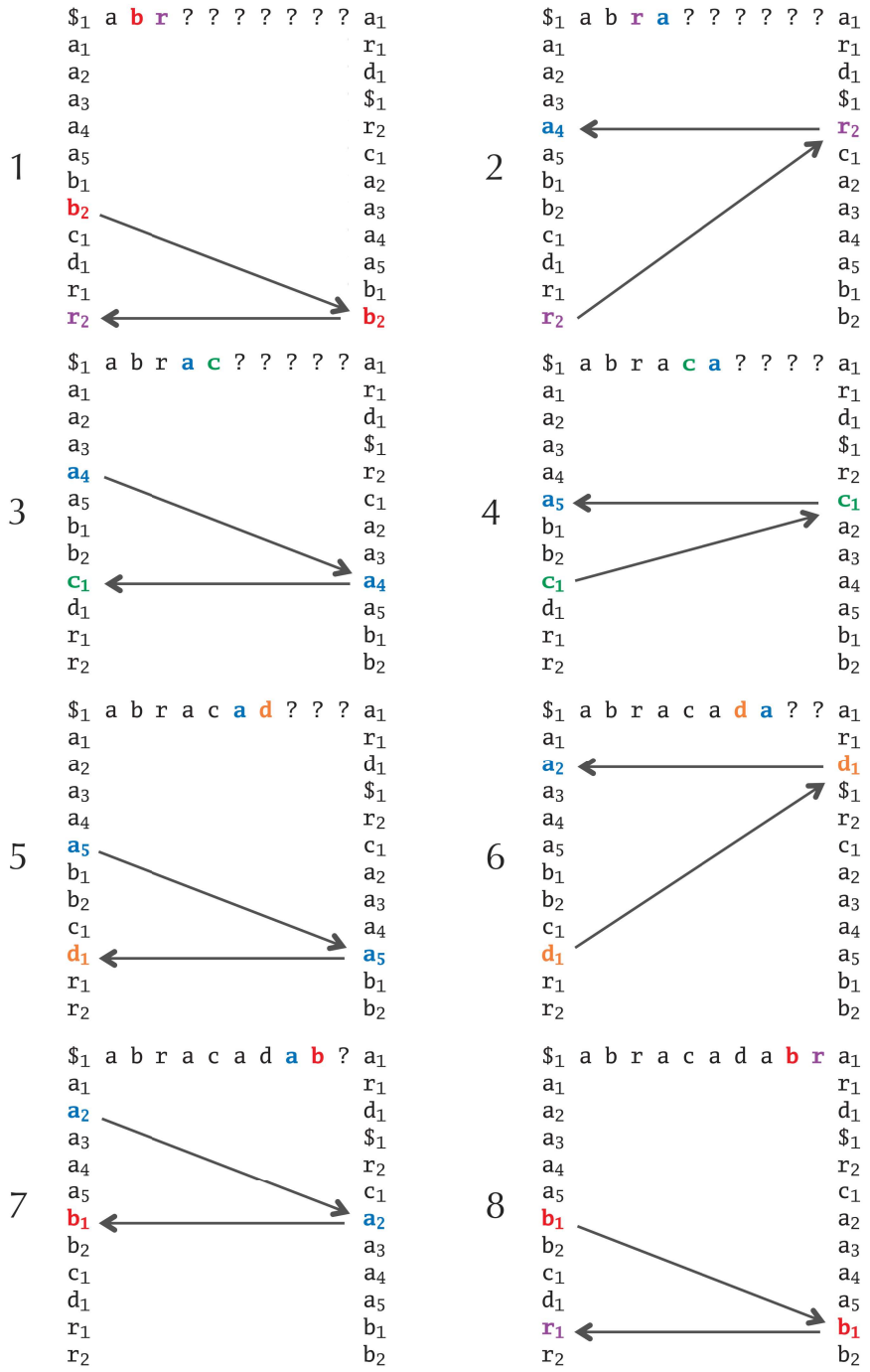

FIGURE 9.12 Repeated applications of the First-Last Property reconstruct the string "abracadabra$" from its Burrows-Wheeler transform "ard$rcaaaabb".

Inverse Burrows-Wheeler Transform Problem:
Reconstruct a string from its Burrows-Wheeler transform.

Input: A string *Transform* (with a single "$" symbol).
Output: The string *Text* such that BWT(*Text*) = *Transform*.

Pattern Matching with the Burrows-Wheeler Transform

A first attempt at Burrows-Wheeler pattern matching

The Burrows-Wheeler transform may be fascinating, but how could it possibly help us decrease the memory required for pattern matching? The idea motivating a Burrows-Wheeler-based approach to pattern matching relies on the observation that each row of M(*Text*) begins with a different suffix of *Text*. Since these suffixes are already ordered lexicographically, as we already noted when pattern matching with the suffix array, any matches of *Pattern* in *Text* will appear at the beginning of consecutive rows of M(*Text*), as shown in Figure 9.13.

M(*Text*)	SUFFIXARRAY(*Text*)
$ p a n a m a b a n a n a s	13
a b a n a n a s $ p a n a m	5
a m a b a n a n a s $ p a n	3
a n a m a b a n a n a s $ p	**1**
a n a n a s $ p a n a m a b	**7**
a n a s $ p a n a m a b a n	**9**
a s $ p a n a m a b a n a n	11
b a n a n a s $ p a n a m a	6
m a b a n a n a s $ p a n a	4
n a m a b a n a n a s $ p a	2
n a n a s $ p a n a m a b a	8
n a s $ p a n a m a b a n a	10
p a n a m a b a n a n a s $	0
s $ p a n a m a b a n a n a	12

FIGURE 9.13 (Left) Because the rows of M(*Text*) are ordered lexicographically, suffixes beginning with the same string ("**ana**") appear in consecutive rows of the matrix. (Right) The suffix array records the starting position of each suffix in *Text* and immediately tells us the locations of "**ana**".

We now have the outline of a method to match *Pattern* to *Text*. Construct M(*Text*), and then identify rows beginning with the first symbol of *Pattern*. Among these rows, determine which ones have a second element matching the second symbol of *Pattern*. Continue this process until we find which rows of M(*Text*) begin with *Pattern*.

STOP and Think: What is wrong with this approach?

Moving backward through a pattern

The issue with this proposed method for pattern matching is that we cannot afford storing the entire matrix M(*Text*), which has $|Text|^2$ entries. In an effort to reduce memory requirements, let's forbid ourselves from accessing any information in M(*Text*) other than *FirstColumn* and *LastColumn*. Using these two columns, we will try to match *Pattern* to *Text* by moving *backward* through *Pattern*. For example, if we want to match *Pattern* = "ana" to *Text* = "panamabananas$", then we will first identify rows of M(*Text*) beginning with "a", the last letter of "ana":

$_1$ p a n a m a b a n a n a s$_1$
a$_1$ b a n a n a s $ p a n a m$_1$
a$_2$ m a b a n a n a s $ p a n$_1$
a$_3$ n a m a b a n a n a s $ p$_1$
a$_4$ n a n a s $ p a n a m a b$_1$
a$_5$ n a s $ p a n a m a b a n$_2$
a$_6$ s $ p a n a m a b a n a n$_3$
b$_1$ a n a n a s $ p a n a m a$_1$
m$_1$ a b a n a n a s $ p a n a$_2$
n$_1$ a m a b a n a n a s $ p a$_3$
n$_2$ a n a s $ p a n a m a b a$_4$
n$_3$ a s $ p a n a m a b a n a$_5$
p$_1$ a n a m a b a n a n a s $$_1$
s$_1$ $ p a n a m a b a n a n a$_6$

As we are moving backward through "ana", we will next look for rows of M(*Text*) beginning with "na". To do this without knowing the entire matrix M(*Text*), we again use the fact that a symbol in *LastColumn* must precede the symbol of *Text* found in the same row in *FirstColumn*. Thus, we only need to identify those rows of M(*Text*) beginning with "a" and ending with "n":

$_1 p a n a m a b a n a n a s_1
a_1 b a n a n a s $ p a n a m_1
a_2 m a b a n a n a s $ p a n_1
a_3 n a m a b a n a n a s $ p_1
a_4 n a n a s $ p a n a m a b_1
a_5 n a s $ p a n a m a b a n_2
a_6 s $ p a n a m a b a n a n_3
b_1 a n a n a s $ p a n a m a_1
m_1 a b a n a n a s $ p a n a_2
n_1 a m a b a n a n a s $ p a_3
n_2 a n a s $ p a n a m a b a_4
n_3 a s $ p a n a m a b a n a_5
p_1 a n a m a b a n a n a s $_1
s_1 $ p a n a m a b a n a n a_6

The First-Last Property tells us where to find the three highlighted "n" in *FirstColumn*, as shown below. All three rows end with "a", yielding three total occurrences of "ana" in *Text*.

$_1 p a n a m a b a n a n a s_1
a_1 b a n a n a s $ p a n a m_1
a_2 m a b a n a n a s $ p a n_1
a_3 n a m a b a n a n a s $ p_1
a_4 n a n a s $ p a n a m a b_1
a_5 n a s $ p a n a m a b a n_2
a_6 s $ p a n a m a b a n a n_3
b_1 a n a n a s $ p a n a m a_1
m_1 a b a n a n a s $ p a n a_2
n_1 a m a b a n a n a s $ p a_3
n_2 a n a s $ p a n a m a b a_4
n_3 a s $ p a n a m a b a n a_5
p_1 a n a m a b a n a n a s $_1
s_1 $ p a n a m a b a n a n a_6

The highlighted occurrences of "a" in *LastColumn* correspond to the third, fourth, and fifth occurrences of "a" in this column, and the First-Last Property tells us that they should correspond to the third, fourth, and fifth occurrences of "a" in *FirstColumn* as well, which identifies the three matches of "ana":

$_1 p a n a m a b a n a n a s_1
a_1 b a n a n a s $ p a n a m_1
a_2 m a b a n a n a s $ p a n_1
a_3 n a m a b a n a n a s $ p_1
a_4 n a n a s $ p a n a m a b_1
a_5 n a s $ p a n a m a b a n_2
a_6 s $ p a n a m a b a n a n_3
b_1 a n a n a s $ p a n a m a_1
m_1 a b a n a n a s $ p a n a_2
n_1 a m a b a n a n a s $ p a_3
n_2 a n a s $ p a n a m a b a_4
n_3 a s $ p a n a m a b a n a_5
p_1 a n a m a b a n a n a s $_1
s_1 $ p a n a m a b a n a n a_6

EXERCISE BREAK: Match *Pattern* = "banana" to *Text* = "panamabananas$" by walking backward through *Pattern* using the Burrows-Wheeler transform of *Text*.

The Last-to-First mapping

We now know how to use BWT(*Text*) to find all matches of *Pattern* in *Text* by walking backward through *Pattern*. However, every time we walk backward, we need to keep track of the rows of M(*Text*) where the matches of a suffix of *Pattern* are hiding. Fortunately, we know that at each step, the rows of M(*Text*) that match a suffix of *Pattern* clump together in consecutive rows of M(*Text*). This means that the collection of all matching rows is revealed by only two pointers, *top* and *bottom*: *top* holds the index of the first row of M(*Text*) that matches the current suffix of *Pattern*, and *bottom* holds the index of the last row of M(*Text*) that matches this suffix. Figure 9.14 shows the process of updating pointers; after walking backward through *Pattern* = "ana", we have that *top* = 3 and *bottom* = 5. After traversing *Pattern*, we can compute the total number of matches of *Pattern* in *Text* by calculating *bottom* − *top* + 1 (e.g., there are 5 − 3 + 1 = 3 matches of "ana" in "panamabananas$").

Let's concentrate on how pointers are updated from one stage to the next. Consider the transition from the second to the third panel in Figure 9.14; how did we know to update the pointers (*top* = 1, *bottom* = 6) into (*top* = 9, *bottom* = 11)? We are looking for the first and last occurrence of "n" in the range of positions from *top* = 1 to *bottom* = 6 in *LastColumn*. The first occurrence of "n" in this range is "n_1" (in position 2) and the last is "n_3" (position 6).

In order to update the *top* and *bottom* pointers, we need to determine where "n_1" and "n_3" occur in *FirstColumn*. The **Last-to-First mapping**, denoted LastToFirst(*i*),

answers the following question: given a symbol at position i in *LastColumn*, what is its position in *FirstColumn*?

For our ongoing example, LASTTOFIRST$(2) = 9$, since the symbol at position 2 of *LastColumn* ("$\mathbf{n_1}$") occurs at position 9 in *FirstColumn*, as shown in Figure 9.15. Similarly, LASTTOFIRST$(6) = 11$, since the symbol at position 6 of *LastColumn* ("$\mathbf{n_3}$") occurs at position 11 in *FirstColumn*. Therefore, with the help of the Last-to-First mapping, we can quickly update the pointers ($top = 1$, $bottom = 6$) into ($top = 9$, $bottom = 11$).

We are now ready to describe **BWMATCHING**, an algorithm that counts the total number of matches of *Pattern* in *Text*, where the only information that we are given is *FirstColumn* and *LastColumn* in addition to the Last-to-First mapping. The pointers *top* and *bottom* are updated by the green lines in the following pseudocode.

```
BWMATCHING(FirstColumn, LastColumn, Pattern, LASTTOFIRST)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern is nonempty
            symbol ← last letter in Pattern
            remove last letter from Pattern
            if positions from top to bottom in LastColumn contain symbol
                topIndex ← first position of symbol among positions from top to bottom
                           in LastColumn
                bottomIndex ← last position of symbol among positions from top to
                              bottom in LastColumn
                top ← LASTTOFIRST(topIndex)
                bottom ← LASTTOFIRST(bottomIndex)
            else
                return 0
        else
            return bottom − top + 1
```

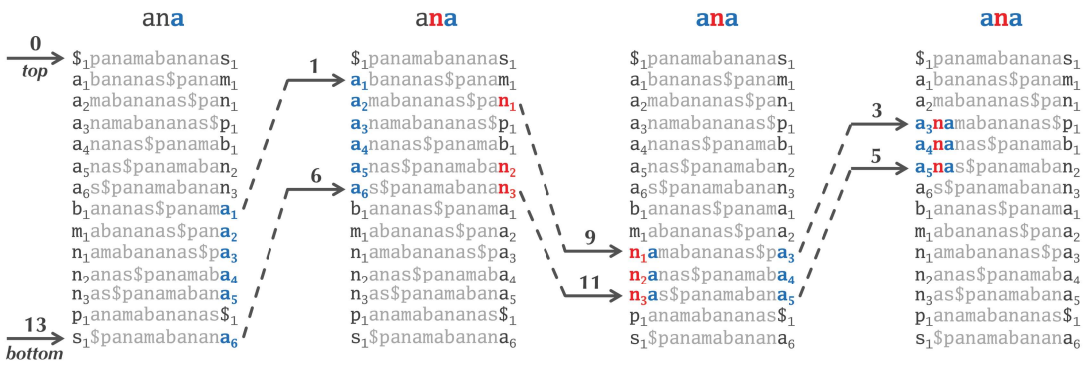

FIGURE 9.14 The pointers *top* and *bottom* hold the indices of the first and last rows of M(*Text*) matching the current suffix of *Pattern* = "ana". The above diagram shows how these pointers are updated when walking backwards through "ana" and looking for substring matches in "panamabananas$".

i	*FirstColumn*	*LastColumn*	LASTTOFIRST(i)	COUNT						
				$	a	b	m	n	p	s
0	$$_1$$	$$s_1$$	13	0	0	0	0	0	0	0
1	$$a_1$$	$$m_1$$	8	0	0	0	0	0	0	1
2	$$a_2$$	$$n_1$$	9	0	0	0	1	0	0	1
3	$$a_3$$	$$p_1$$	12	0	0	0	1	1	0	1
4	$$a_4$$	$$b_1$$	7	0	0	0	1	1	1	1
5	$$a_5$$	$$n_2$$	10	0	0	1	1	1	1	1
6	$$a_6$$	$$n_3$$	11	0	0	1	1	2	1	1
7	$$b_1$$	$$a_1$$	1	0	0	1	1	3	1	1
8	$$m_1$$	$$a_2$$	2	0	1	1	1	3	1	1
9	$$n_1$$	$$a_3$$	3	0	2	1	1	3	1	1
10	$$n_2$$	$$a_4$$	4	0	3	1	1	3	1	1
11	$$n_3$$	$$a_5$$	5	0	4	1	1	3	1	1
12	$$p_1$$	$$\$_1$$	0	0	5	1	1	3	1	1
13	$$s_1$$	$$a_6$$	6	1	5	1	1	3	1	1
				1	6	1	1	3	1	1

FIGURE 9.15 The Last-to-First mapping and count array. Precomputing the count array prevents time-consuming updates of the *top* and *bottom* pointers in **BWMATCHING**.

Speeding Up Burrows-Wheeler Pattern Matching

Substituting the Last-to-First mapping with count arrays

If you implemented **BWMATCHING** in the previous section, you probably found this algorithm to be slow. The reason for its sluggishness is that updating the pointers *top* and *bottom* is time-intensive, since it requires examining every symbol in *LastColumn* between *top* and *bottom* at each step. To improve **BWMATCHING**, we introduce a function $\text{COUNT}_{symbol}(i, LastColumn)$, which returns the number of occurrences of *symbol* in the first i positions of *LastColumn*. For example, $\text{COUNT}_{\texttt{"n"}}(10, \texttt{"smnpbnnaaaaa\$a"}) = 3$, and $\text{COUNT}_{\texttt{"a"}}(4, \texttt{"smnpbnnaaaaa\$a"}) = 0$. In Figure 9.15, we show arrays holding $\text{COUNT}_{symbol}(i, \texttt{"smnpbnnaaaaa\$a"})$ for every *symbol* occurring in "panamabananas$".

EXERCISE BREAK: Compute the arrays COUNT for BWT("abracadabra$").

We will say that the k-th occurrence of *symbol* in a column of a matrix has **rank** k in this column. For *Text* = "panamabananas$", note that the first and last occurrences of *symbol* in the range of positions from *top* to *bottom* in *LastColumn* have respective ranks

$$\text{COUNT}_{symbol}(top, LastColumn) + 1$$

and

$$\text{COUNT}_{symbol}(bottom + 1, LastColumn).$$

As illustrated in Figure 9.15, when $top = 1$, $bottom = 6$, and *symbol* = "n",

$$\text{COUNT}_{\texttt{"n"}}(top, LastColumn) + 1 = 1$$
$$\text{COUNT}_{\texttt{"n"}}(bottom + 1, LastColumn) = 3$$

The occurrences of "n" having ranks 1 and 3 are located at positions 2 and 6 of *LastColumn*, implying that we should update *top* to $\text{LASTTOFIRST}(2) = 9$ and *bottom* to $\text{LASTTOFIRST}(6) = 11$. Thus, the four green lines in the pseudocode for **BWMATCHING** can be rewritten as follows.

```
topIndex ← position of symbol with rank COUNT_symbol(top, LastColumn) + 1
           in LastColumn
bottomIndex ← position of symbol with rank COUNT_symbol(bottom + 1, LastColumn)
              in LastColumn
top ← LASTTOFIRST(topIndex)
bottom ← LASTTOFIRST(bottomIndex)
```

By eliminating the variables *topIndex* and *bottomIndex*, we can reduce these four lines of pseudocode to only two lines:

$top \leftarrow$ LastToFirst(position of *symbol* with rank $\text{Count}_{symbol}(top, LastColumn) + 1$ in *LastColumn*)
$bottom \leftarrow$ LastToFirst(position of *symbol* with rank $\text{Count}_{symbol}(bottom + 1, LastColumn)$ in *LastColumn*)

Note that these two lines of pseudocode merely compute the position of *symbol* with rank i in *FirstColumn* from its positions in *LastColumn*. This task can be compactly described without the Last-to-First mapping by the following two lines:

$top \leftarrow$ position of *symbol* with rank $\text{Count}_{symbol}(top, LastColumn) + 1$ in *FirstColumn*
$bottom \leftarrow$ position of *symbol* with rank $\text{Count}_{symbol}(bottom + 1, LastColumn)$ in *FirstColumn*

For $top = 1$, $bottom = 6$, and $symbol =$ "n", the occurrences of "n" having ranks $\text{Count}_{\text{"n"}}(top, LastColumn) + 1 = 1$ and $\text{Count}_{\text{"a"}}(bottom + 1, LastColumn) = 3$ are located in positions 9 and 11 of *FirstColumn*, respectively.

STOP and Think: Do we need to store all of *FirstColumn* in memory in order to execute the preceding two lines of pseudocode?

Getting rid of the first column of the Burrows-Wheeler matrix

BWMatching requires us to store *FirstColumn*, *LastColumn*, and LastToFirst. We can reduce the amount of memory needed to store the information contained in *FirstColumn* by defining FirstOccurrence(*symbol*) as the first position of *symbol* in *FirstColumn*. If *Text* = "panamabananas$", then *FirstColumn* is "$aaaaaabmnnnps", and the array holding all values of FirstOccurrence is $[0, 1, 7, 8, 9, 11, 12]$, as shown in Figure 9.16. For DNA strings of any length, the array FirstOccurrence contains only five elements. The previous two lines of code can now be rewritten as follows:

$top \leftarrow \text{FirstOccurrence}(symbol) + \text{Count}_{symbol}(top, LastColumn)$
$bottom \leftarrow \text{FirstOccurrence}(symbol) + \text{Count}_{symbol}(bottom + 1, LastColumn) - 1$

i	*FirstColumn*	FIRSTOCCURRENCE
0	$\$_1$	0
1	$\mathtt{a}_1$	1
2	$\mathtt{a}_2$	
3	$\mathtt{a}_3$	
4	$\mathtt{a}_4$	
5	$\mathtt{a}_5$	
6	$\mathtt{a}_6$	
7	$\mathtt{b}_1$	7
8	$\mathtt{m}_1$	8
9	$\mathtt{n}_1$	9
10	$\mathtt{n}_2$	
11	$\mathtt{n}_3$	
12	$\mathtt{p}_1$	12
13	$\mathtt{s}_1$	13

FIGURE 9.16 The array FIRSTOCCURRENCE has just seven elements, which is equal to the number of distinct symbols in `"panamabananas$"`.

When $top = 1$, $bottom = 6$, and $symbol =$ "n", we have that

$$\begin{aligned} \text{FIRSTOCCURRENCE}(\text{"n"}) &= 9 \\ \text{COUNT}_{\text{"n"}}(top, LastColumn) &= 0 \\ \text{COUNT}_{\text{"n"}}(bottom + 1, LastColumn) &= 3 \end{aligned}$$

Recalling Figure 9.14, this again implies that $(top = 1, bottom = 6)$ will be updated as

$$\begin{aligned} top &= 9 + 0 = 9 \\ bottom &= 9 + 3 - 1 = 11 \end{aligned}$$

In the process of simplifying the green lines of pseudocode from **BWMATCHING**, we have also substituted *FirstColumn* by FIRSTOCCURRENCE and LASTTOFIRST by COUNT, resulting in a more efficient algorithm called **BETTERBWMATCHING**, shown below.

You may be wondering why we call this algorithm "better", since on the one hand, if you have to compute the COUNT arrays as you go, then you will not obtain a runtime speedup. On the other hand, if you have to precompute these arrays, then you will need to store them in memory, which is space-intensive. Hold onto this thought.

```
BETTERBWMATCHING(FIRSTOCCURRENCE, LastColumn, Pattern, COUNT)
    top ← 0
    bottom ← |LastColumn| − 1
    while top ≤ bottom
        if Pattern is nonempty
            symbol ← last letter in Pattern
            remove last letter from Pattern
            top ← FIRSTOCCURRENCE(symbol) + COUNT_symbol(top, LastColumn)
            bottom ← FIRSTOCCURRENCE(symbol) + COUNT_symbol(bottom + 1,
                        LastColumn) − 1
        else
            return bottom − top + 1
    return
```

Where are the Matched Patterns?

We hope that you have noticed a limitation of **BETTERBWMATCHING** — even though this algorithm counts the *number* of occurrences of *Pattern* in *Text*, it does not tell us *where* these occurrences are located in *Text*! To locate pattern matches identified by the algorithm, we can once again use the suffix array, as shown in Figure 9.13 (right). In this figure, the suffix array immediately finds the three matches of "ana" in "panamabananas$".

The suffix array makes our job easy, but recall that our original motivation for using the Burrows-Wheeler transform was to *reduce* the amount of memory used by the suffix array for pattern matching. If we add the suffix array to Burrows-Wheeler-based pattern matching, then we are right back where we started!

The memory-saving device that we will employ is inelegant but useful. We will build a **partial suffix array** of *Text*, denoted SUFFIXARRAY$_K$(*Text*), which only contains values that are multiples of some positive integer K (Figure 9.17). In real applications, partial suffix arrays are often constructed for $K = 100$, thus reducing memory usage by a factor of 100 compared to a full suffix array. See **CHARGING STATION: Partial Suffix Array Construction** for more details.

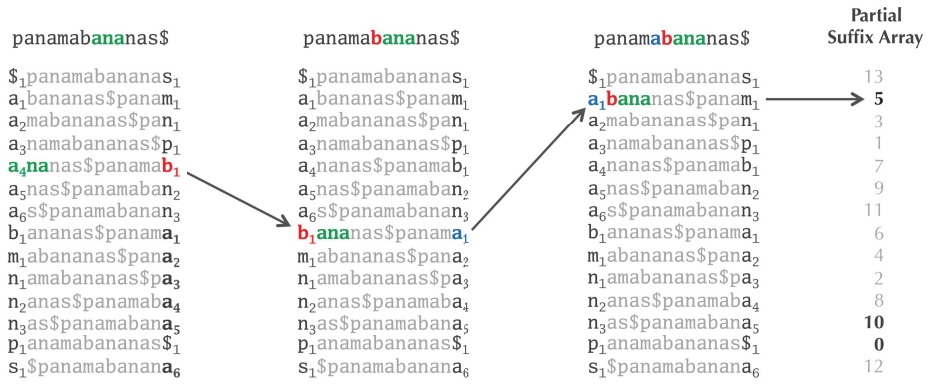

FIGURE 9.17 One of the matches of "ana" in "panamabananas$" is highlighted in the matrix on the right. By walking backward, we find that "ana" is preceded by "b_1", which in turn is preceded by "a_1". The partial suffix array above, generated for $K = 5$, indicates that "a_1" occurs at position 5 of "panamabananas". Since it took us two steps to walk backward to "a_1", we conclude that this occurrence of "ana" begins at position $5 + 2 = 7$.

Burrows and Wheeler Set Up Checkpoints

We will now discuss how to improve **BETTERBWMATCHING** by resolving the trade-off between precomputing the values of $\text{COUNT}_{symbol}(i, LastColumn)$ (requiring substantial memory) and computing these values as we go (requiring substantial runtime).

The balance that we strike is similar to the one used for the partial suffix array. Rather than storing $\text{COUNT}_{symbol}(i, LastColumn)$ for *all* positions i, we will only store the COUNT arrays when i is divisible by C, where C is a constant; these arrays are called **checkpoint arrays** (Figure 9.18). When C is large (C is typically equal to 100 in practice) and the alphabet is small (e.g., four nucleotides), checkpoint arrays require only a fraction of the memory used by BWT($Text$).

What about runtime? Using checkpoint arrays, we can compute the *top* and *bottom* pointers in a constant number of steps (i.e., fewer than C). Since each string $Pattern$ requires at most $|Pattern|$ pointer updates, the modified **BETTERBWMATCHING** algorithm now requires $\mathcal{O}(|Patterns|)$ runtime, which is the same as using a trie or suffix array.

Furthermore, we now only have to store the following data in memory: BWT($Text$), FIRSTOCCURRENCE, the partial suffix array, and the checkpoint arrays. Storing this data requires memory approximately equal to $1.5 \cdot |Text|$. Thus, we have finally knocked

i	*LastColumn*	COUNT						
		$	a	b	m	n	p	s
0	s_1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	m_1	0	0	0	0	0	0	1
2	n_1	0	0	0	1	0	0	1
3	p_1	0	0	0	1	1	0	1
4	b_1	0	0	0	1	1	1	1
5	n_2	**0**	**0**	**1**	**1**	**1**	**1**	**1**
6	n_3	0	0	1	1	2	1	1
7	a_1	0	0	1	1	3	1	1
8	a_2	0	1	1	1	3	1	1
9	a_3	0	2	1	1	3	1	1
10	$\mathbf{a_4}$	**0**	**3**	**1**	**1**	**3**	**1**	**1**
11	$\mathbf{a_5}$	0	4	1	1	3	1	1
12	$\mathbf{\$_1}$	0	5	1	1	3	1	1
13	a_6	1	5	1	1	3	1	1
		1	6	1	1	3	1	1

FIGURE 9.18 The COUNT checkpoint arrays for *Text* = "panamabananas$" and $C = 5$ are highlighted in bold. If we want to compute $\text{COUNT}_{\texttt{"a"}}(13, \texttt{"smnpbnnaaaaa\$a"})$, then the checkpoint array at position 10 tells us that there are 3 occurrences of "a" *before* position 10 of "smnpbnnaaaaa$a". We then check whether "a" is present at position **10** (yes), **11** (yes), and **12** (no) of *LastColumn* to conclude that $\text{COUNT}_{\texttt{"a"}}(13, \texttt{"smnpbnnaaaaa\$a"}) = \mathbf{3} + \mathbf{2} = 5$.

down the memory required for solving the Multiple Pattern Matching Problem for millions of sequencing reads into a workable range.

As the plummeting cost of DNA sequencing has made headlines, it is easy to fail to appreciate the progress that has been made in the computational side of read mapping. So, before continuing into approximate pattern matching, we would like to pause and reflect on just how far the algorithms for read mapping – and the hardware running them – have progressed. In 1975, the state-of-the-art Aho-Corasick algorithm was touted as requiring 15 minutes to map just 24 English words to a dictionary. Just a generation later, when Burrows-Wheeler based approaches were introduced into read mapping, the same 15 minutes was used to map almost *ten million* reads to a reference genome. Having seen how far we have come in the last four decades, we can only imagine where it will take us in the future.

Epilogue: Mismatch-Tolerant Read Mapping

Reducing approximate pattern matching to exact pattern matching

In this section, we will return to the goal of identifying SNPs in an individual genome when compared against the reference genome. To do so, we need to generalize the Approximate Pattern Matching Problem from Chapter 1 to the case of multiple patterns.

Multiple Approximate Pattern Matching Problem:
Find all approximate occurrences of a collection of patterns in a text.

Input: A string *Text*, a collection of shorter strings *Patterns*, and an integer d.
Output: All starting positions in *Text* where a string from *Patterns* appears as a substring with at most d mismatches.

We begin with the simple observation that if *Pattern* occurs in *Text* with a single mismatch, then we can divide *Pattern* into two halves, one of which occurs exactly in *Text*, as illustrated below.

```
Pattern        acttggct
Text    ...ggcacactaggctcc...
```

Thus, we can find whether *Pattern* occurs with one mismatch in *Text* by dividing *Pattern* into two halves and then searching for exact matches of these shorter strings. If we find a match with one of these halves of *Pattern*, we then check if the entire string *Pattern* occurs with a single mutation.

This method can easily be generalized for approximate pattern matching with $d > 1$ mismatches; if *Pattern* approximately matches a substring of *Text* with at most d mismatches, then *Pattern* and *Text* must share at least one k-mer for a sufficiently large value of k. For example, if we are looking for a pattern of length 20 with at most $d = 3$ mismatches, then we can divide this pattern into four parts of length $20/(3+1) = 5$ and search for exact matches of these shorter substrings:

```
Pattern      acttaggctcgggataatcc
Text    ...actaagtctcgggataagcc...
```

This observation is helpful because it reduces *approximate* pattern matching to the *exact* matching of shorter patterns, which allows us to use fast algorithms that are designed for exact pattern matching but are not applicable to approximate pattern matching, such as approaches based on suffix trees and suffix arrays.

STOP and Think: If *Pattern* has length 23 and appears in *Text* with 3 mismatches, can we conclude that *Pattern* shares a 6-mer with *Text*? Can we conclude that it shares a 5-mer with *Text*?

The question remains how to find the maximum value of k that guarantees that any *Pattern* of length n with d mismatches in *Text* will have a k-mer substring exactly matching *Text*.

Theorem. *If two strings of length n match with at most d mismatches, then they must share a k-mer of length $k = \lfloor n/(d+1) \rfloor$.*

Proof. Divide the first string into $d+1$ substrings, where the first d substrings have exactly k symbols and the final substring has at least k symbols. For the case $d = 3$, here is a subdivision of a 23-nucleotide string ($n = 23$) into $d+1 = 4$ substrings, where the first 3 substrings have $k = \lfloor n/(d+1) \rfloor = \lfloor 23/4 \rfloor = 5$ symbols and the last substring has 8 symbols.

acttaggctcgggataatccgga

If you distribute d mismatches among the positions of the string, then the mismatches may affect at most d substrings, which leaves at least one substring (of length at least k) unchanged. This substring is shared by both strings. □

We now have the outline of an algorithm for matching a string *Pattern* of length n to *Text* with at most d mismatches. We first divide *Pattern* into $d+1$ segments of length $k = \lfloor n/(d+1) \rfloor$, called **seeds**. After finding which seeds match *Text* exactly (**seed detection**), we attempt to extend seeds in both directions in order to verify whether *Pattern* occurs in *Text* with at most d mismatches (**seed extension**).

BLAST: Comparing a sequence against a database

Using shared k-mers to find similarities between biological sequences has some disadvantages. For example, two proteins may have similar functions but not share any k-mers, even for small values of k.

The **Basic Local Alignment Search Tool (BLAST)** is a heuristic that can find similarities between proteins, even if all amino acids in one protein have mutated compared to the other protein. BLAST is so fast that it is often used to query a protein against *all other* known proteins in protein databases. The paper introducing BLAST was released in 1990, and with over 40,000 citations, it has become one of the most cited scientific papers ever published.

To see how BLAST works, say we are given an integer k and strings $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_m$ to compare. We define a **segment pair** of x and y as a pair formed by a k-mer from x and a k-mer from y. The score of the segment pair corresponding to the k-mers starting at position i in x and position j in y is

$$\sum_{t=0}^{k-1} \text{SCORE}(x_{i+t}, y_{j+t}),$$

where $\text{SCORE}(x_{i+t}, y_{j+t})$ is determined by a scoring matrix such as the PAM scoring matrix that we introduced in Chapter 5. A **locally maximal segment pair** is a segment pair whose score cannot be increased by extending or shortening both strings in the segment pair. BLAST attempts to find not the highest-scoring segment pair of x and y, but rather all locally maximal segment pairs in these strings with scores above some threshold.

The key ingredient of BLAST is to first quickly find all k-mers that have scores above a given threshold when scored against some k-mer in the query string. If the score threshold is high, then the set of all resulting segment pairs formed between the query string and strings in the database is not too large. In this case, the database can be searched for exact occurrences of these high-scoring k-mers from this set, producing an initial set of seeds. This is an instance of the Multiple Pattern Matching Problem, which we have learned how to solve quickly.

EXERCISE BREAK: Given the PAM_{250} scoring matrix, only five 3-mers score higher than 23 against `CFC`: `CIC`, `CLC`, `CMC`, `CWC`, and `CYC`. How many 3-mers score higher than 20 against `CFC`?

After finding seeds, BLAST attempts to extend these seeds (allowing for insertions and deletions) in order to obtain locally maximal segment pairs.

EXERCISE BREAK: Given the PAM_{250} scoring matrix, an amino acid k-mer *Peptide*, and a threshold θ, develop an efficient algorithm for finding the exact number of k-mers scoring more than θ against *Peptide*.

Approximate pattern matching with the Burrows-Wheeler transform

To extend the Burrows-Wheeler approach to approximate pattern matching, we will not stop when we encounter a mismatch. On the contrary, we will proceed onward until we either find an approximate match or exceed the limit of d mismatches.

Figure 9.19 illustrates the search for "asa" in "panamabananas$" with at most 1 mismatch. Let's first proceed as in the case of exact pattern matching, threading "asa" backwards using the Burrows-Wheeler transform. After finding six occurrences of "a", we identify six inexact occurrences of "sa": "**p**a", "**m**a", "**b**a", and three occurrences of "**n**a". We note that these three strings have accumulated a mismatch and then continue with all six strings.

In the next step, five of the inexact occurrences of "sa" can be extended into inexact occurrences of "asa" with only a single mismatch: "a**m**a", "a**b**a", and three occurrences of "a**n**a". We fail to extend "**p**a", which we eliminate from consideration.

In practice, this heuristic faces complications. We do not want to start allowing mismatched strings at the early stages of **BETTERBWMATCHING**, or else we will have to consider too many frivolous candidate strings. We may therefore require that a suffix of *Pattern* of some threshold length matches *Text* exactly. Moreover, the method becomes time-intensive when using large values of d, as we must explore many inexact matches. Practical applications often limit the value of d to at most 3.

You should now be ready to design your own approach to solve the Multiple Approximate Pattern Matching Problem and use this solution to map real sequencing reads.

CHALLENGE PROBLEM: Given the Burrows-Wheeler transform and a partial suffix array of the bacterial genome *Mycoplasma pneumoniae* along with a collection of reads, find all reads occurring in the genome with at most one mismatch.

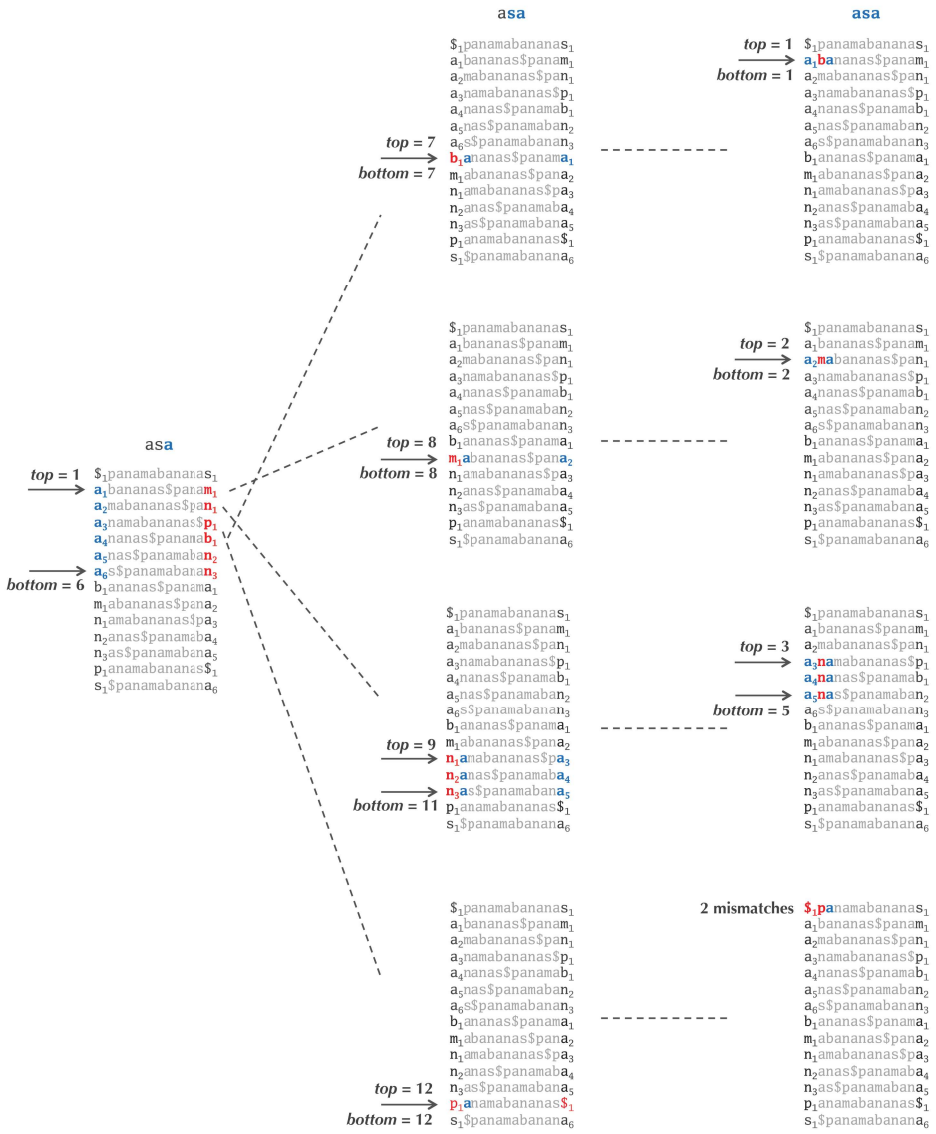

FIGURE 9.19 Using the Burrows-Wheeler transform to find approximate pattern matches of "asa" in "panamabananas$" with 1 mismatch. (Left) We identify six occurrences of "a". (Middle) Working backwards, we find four different inexact partial matches: "ba", "ma", "na" (three occurrences), and "pa". (Right) Additional mismatches are not allowed in the partial matches that we have found, but these partial matches can be extended to yield five inexact matches of "asa".

Charging Stations

Constructing a suffix tree

To construct a suffix tree, we will first modify the construction of the suffix trie as follows. Although each edge *edge* in a suffix trie is labeled by a single symbol SYMBOL(*edge*), it is unclear where this symbol came from in *Text*. We will therefore add another label for each edge (denoted POSITION(*edge*)) referring to the position of this symbol in *Text*. If SYMBOL(*edge*) corresponds to more than one position in *Text*, then we will assign it its minimum starting position.

For example, consider the modified suffix trie for *Text* = "panamabananas$" shown in Figure 9.20. There are five edges labeled by "m" (colored purple), all of which are

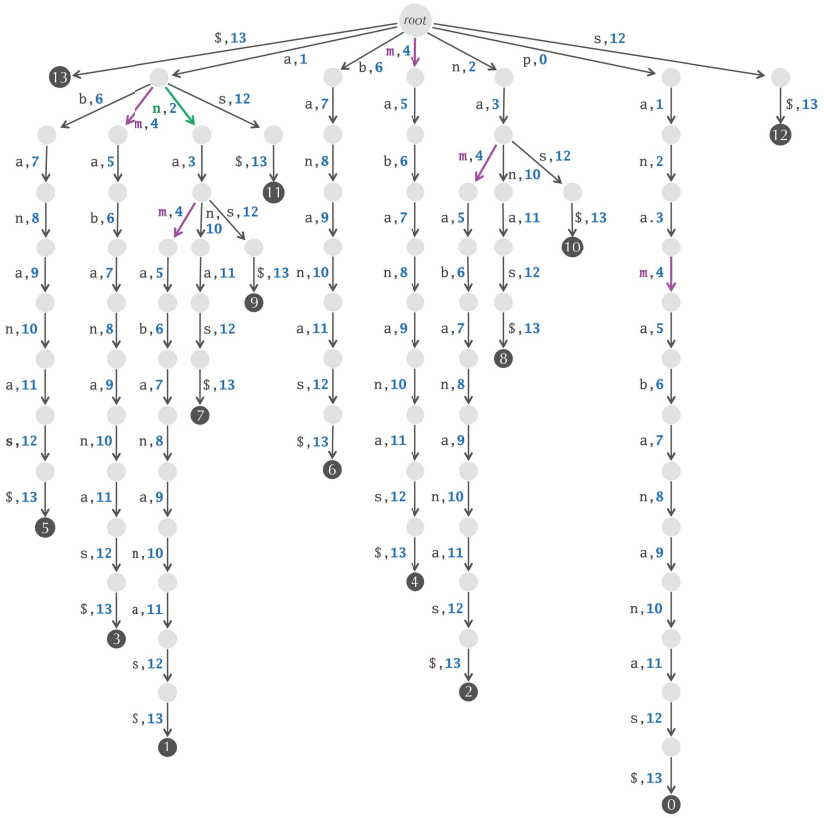

FIGURE 9.20 The modified suffix trie of *Text* = "panamabananas$". Each edge is assigned the minimum position to which the edge's symbol corresponds in *Text* (shown in blue).

labeled by position 4, because this is the only occurrence of "m" in *Text*. On the other hand, the green edge corresponding to "n" is a different story. By following every path from this edge down to the leaves, we see that it corresponds to occurrences of "n" in the suffixes "anamabananas$", "ananas$", and "anas$", at positions 2, 8, and 10. As a result, we assign this edge the minimum of these positions.

The following pseudocode constructs the modified suffix trie of a string *Text* by traversing the suffixes of *Text* from longest to shortest. Given a suffix, it attempts to spell the suffix by moving downward in the tree, following edge labels as far as possible until it can go no further. At that point, it adds the rest of the suffix to the trie in the form of a path to a leaf, along with the position of each symbol in the suffix.

```
ModifiedSuffixTrieConstruction(Text)
    Trie ← a graph consisting of a single node root
    for i ← 0 to |Text| − 1
        currentNode ← root
        for j ← i to |Text| − 1
            currentSymbol ← j-th symbol of Text
            if there is an outgoing edge from currentNode labeled by currentSymbol
                currentNode ← ending node of this edge
            else
                add a new node newNode to Trie
                add an edge newEdge connecting currentNode to newNode in Trie
                Symbol(newEdge) ← currentSymbol
                Position(newEdge) ← j
                currentNode ← newNode
        if currentNode is a leaf in Trie
            assign label i to this leaf
    return Trie
```

We can now transform a modified suffix trie into a suffix tree as follows. Note in Figure 9.6 (page 131) that each edge *edge* in SuffixTree("panamabananas$") is labeled by a string of symbols, denoted String(*edge*). As we mentioned in the main text, storing all these strings is memory-intensive, and so we will instead label *edge* by two integers: the starting position of the first occurrence of String(*edge*) in *Text*, denoted Position(*edge*), and its length, denoted Length(*edge*). For the modified suffix tree of *Text* = "panamabananas$" shown in Figure 9.21, these two integers are colored blue and red, respectively. For example, the edge labeled "mabananas$" in Figure 9.6 is labeled

by POSITION(*edge*) = 4 and LENGTH(*edge*) = 10 in Figure 9.21. There are two edges labeled "na" in Figure 9.6, and both of them are labeled by POSITION(*edge*) = 2 and LENGTH(*edge*) = 2 in Figure 9.21.

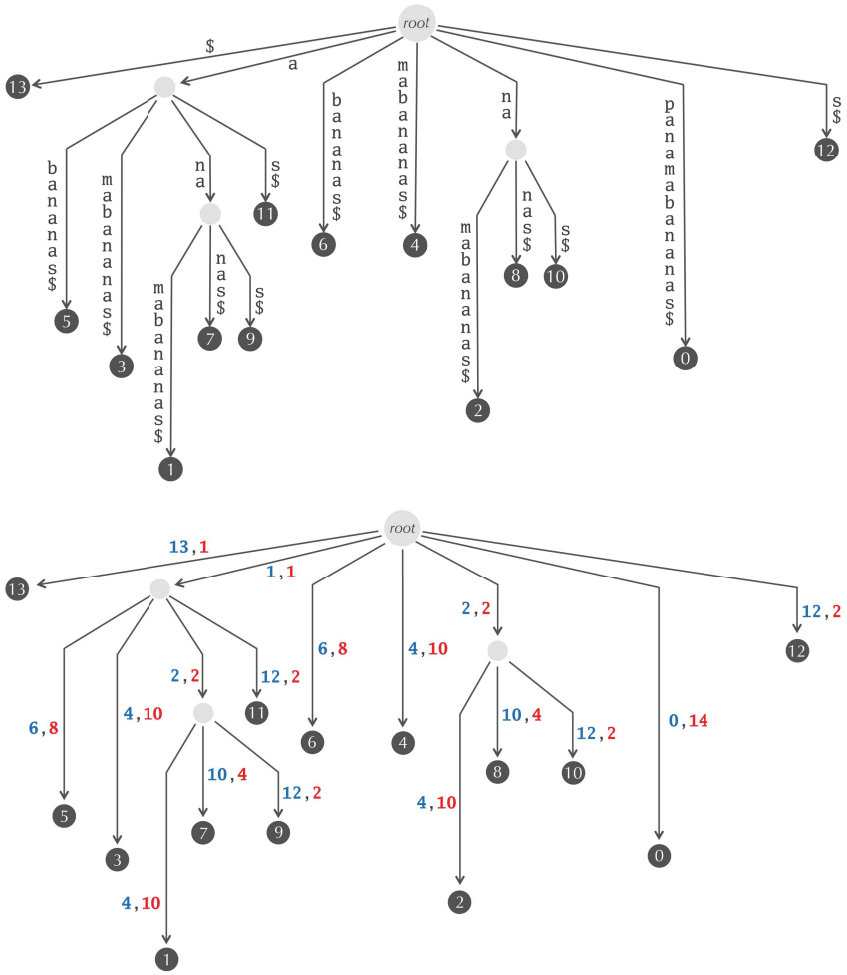

FIGURE 9.21 (Top) The suffix tree of *Text* = "panamabananas$. (Bottom) The modified suffix tree of *Text*. For each edge, the initial position of the substring to which it corresponds in *Text* is shown in blue, and the length of this substring is shown in red.

The following pseudocode constructs a suffix tree using the modified suffix trie constructed by MODIFIEDSUFFIXTREECONSTRUCTION. This algorithm will consolidate each non-branching path (i.e., a path whose intermediate nodes have indegree and

outdegree equal to 1) of the modified suffix trie into a single edge.

```
SUFFIXTREECONSTRUCTION(Text)
    Trie ← MODIFIEDSUFFIXTRIECONSTRUCTION(Text)
    for each non-branching path Path in Trie
        substitute Path by a single edge e connecting the first and last nodes of Path
        POSITION(e) ← POSITION(first edge of Path)
        LENGTH(e) ← number of edges of Path
    return Trie
```

Solving the Longest Shared Substring Problem

A naive approach for finding a longest shared substring of strings $Text_1$ and $Text_2$ would construct one suffix tree for $Text_1$ and another for $Text_2$. Instead, we will add "#" to the end of $Text_1$, add "$" to the end of $Text_2$, and then construct the single suffix tree for the concatenation of $Text_1$ and $Text_2$ (Figure 9.22). We color a leaf in this suffix tree blue if it is labeled by the starting position of a suffix starting in $Text_1$; we color a leaf red if it is labeled by the starting position of a suffix starting in $Text_2$.

We also color the remaining nodes of the suffix tree blue, red, and purple according to the following rules:

- a node is colored blue or red if all leaves in its subtree (i.e., the subtree beneath it) are all blue or all red, respectively;
- a node is colored purple if its subtree contains both blue and red leaves.

We use COLOR(v) to denote the color of node v.

There are three purple nodes in Figure 9.22 (other than the root), and the strings spelled from the root to each of these nodes are "a", "ana", and "na". Note that these three substrings are shared by $Text_1$ = "panama" and $Text_2$ = "bananas". This is no accident.

EXERCISE BREAK: Prove that a path ending in a purple node in the suffix tree of $Text_1$ and $Text_2$ spells out a substring shared by $Text_1$ and $Text_2$.

EXERCISE BREAK: Prove that a path ending in a blue (respectively, red) node in the suffix tree of $Text_1$ and $Text_2$ spells out a substring that appears in $Text_1$ but not in $Text_2$ (respectively, $Text_2$ but not in $Text_1$).

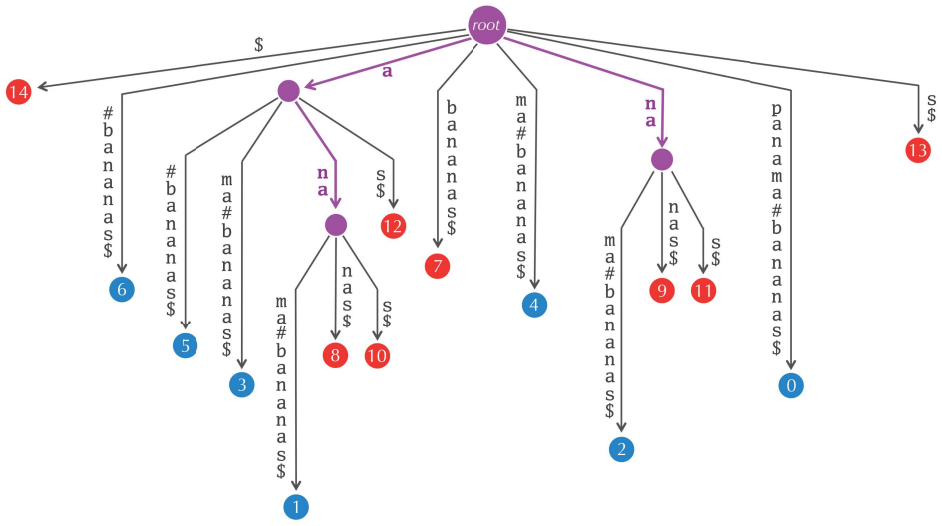

FIGURE 9.22 SUFFIXTREE("**panama#bananas$**"), constructed for $Text_1$ = "**panama**" and $Text_2$ = "**bananas**". Leaves corresponding to suffixes starting in "**panama#**" are colored blue; leaves corresponding to suffixes starting in "**bananas$**" are colored red. Every string of symbols spelled from the root to a purple node corresponds to a substring shared by $Text_1$ and $Text_2$.

The previous two exercises imply that in order to find the longest shared substring between $Text_1$ and $Text_2$, we need to examine all purple nodes as well as the strings spelled by paths leading to the purple nodes. A longest such string yields a solution to the Longest Shared Substring Problem.

TREECOLORING, which is illustrated in Figure 9.23, colors the nodes of a suffix tree from the leaves upward. This algorithm assumes that the leaves of the suffix tree have been labeled "blue" or "red" and all other nodes have been labeled "gray". A node in a tree is called **ripe** if it is gray but has no gray children.

```
TREECOLORING(ColoredTree)
    while ColoredTree has ripe nodes
        for each ripe node v in ColoredTree
            if there exist differently colored children of v
                COLOR(v) ← "purple"
            else
                COLOR(v) ← color of all children of v
    return ColoredTree
```

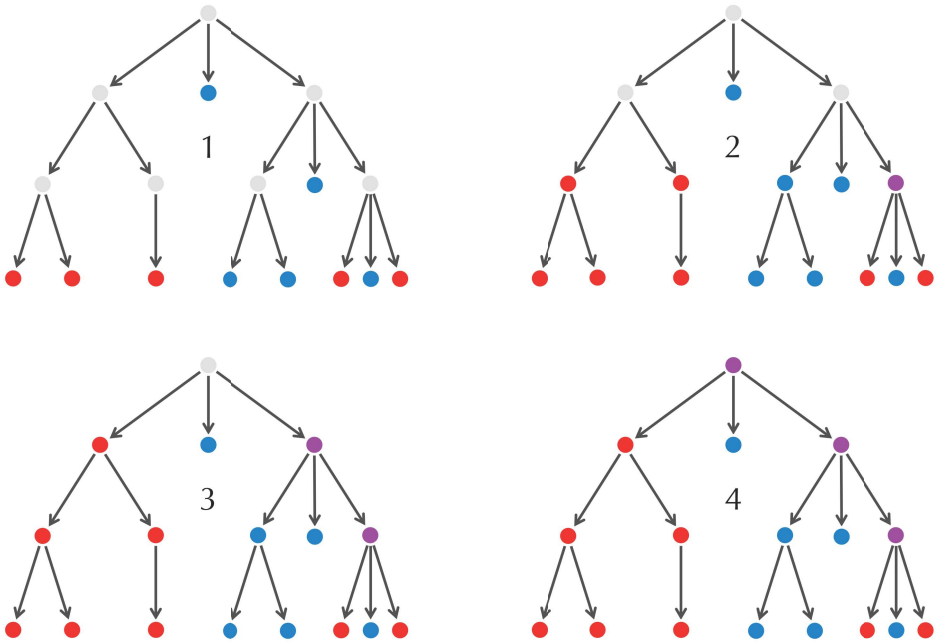

FIGURE 9.23 An illustration of the steps taken by **TREECOLORING** for a tree with an initial coloring of leaves as red or blue (top left).

9P

Partial suffix array construction

To construct the partial suffix array $\text{SUFFIXARRAY}_K(Text)$, we first need to construct the full suffix array and then retain only the elements of this array that are divisible by K, along with their indices i. This is illustrated in Figure 9.24 for $Text =$ `"panamabananas$"` and $K = 5$, where $\text{SUFFIXARRAY}_K(Text)$ corresponds to the bold elements.

9Q

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SUFFIXARRAY(*Text*)	13	5	3	1	7	9	11	6	4	3	8	10	0	12

FIGURE 9.24 Partial suffix array construction.

Detours

The reference human genome

When genomes are assembled from DNA taken from a number of donors, the reference genome represents a mosaic of donor genomes. The existing reference human genome was derived from thirteen volunteers in the United States; it continues to be improved by fixing errors and filling in the remaining gaps (there are currently over a hundred).

The reference genome is often used as a template on which new individual genomes can be rapidly assembled. Comparison of the reference genome and individual human genomes typically reveals about three million SNPs, and about 0.1% of an individual human genome cannot be matched to the reference genome at all.

In regions with high diversity, the reference genome may differ significantly from individual genomes. An example of a high-diversity region of the human genome is the **major histocompatibility complex**, a family of genes powering immune systems. Since these genes have an unusually large number of alternate forms, two individuals hardly ever have exactly the same set of genes from this complex.

Rearrangements, insertions, and deletions in human genomes

Until recently, biologists focused primarily on SNPs in the human genome, assuming that rearrangements and indels are relatively rare. In 2005, Evan Eichler surprised biologists when he found hundreds of rearrangements and indels separating the genomes of two individuals. This finding was important because rearrangements and indels are often hallmarks of disease; for example, repeated insertions of the nucleotide triplet `CAG` increases the severity of Huntington's disease.

In 2013, Gerton Lunter revealed the true extent of indels in the human population by identifying over a *million* indels in a cohort of over a hundred individuals. Intriguingly, he found that over half of the indels occurred in just 4% of the genome; in other words, some regions of the human genome represent "indel hotspots". As the catalog of human rearrangements and indels grows, biologists are gaining the ability to identify frequently mutated genes as well as implicate rearrangements and indels when diagnosing complex disorders.

The Aho-Corasick algorithm

The **Aho-Corasick algorithm** for the Multiple Pattern Matching Problem was invented by Alfred Aho and Margaret Corasick in 1975. The runtime of their algorithm is $\mathcal{O}(|Patterns| + |Text| + m)$, where m is the number of output matches.

Imagine sliding the trie in Figure 9.25 against *Text* = "bantenna". Similarly to TRIEMATCHING, the Aho-Corasick algorithm starts at the root and attempts to build a path spelling a prefix of "bantenna". This attempt fails after three nodes ("bantenna"). In TRIEMATCHING, we would begin again at the root and attempt to find a match starting at the second symbol of "bantenna".

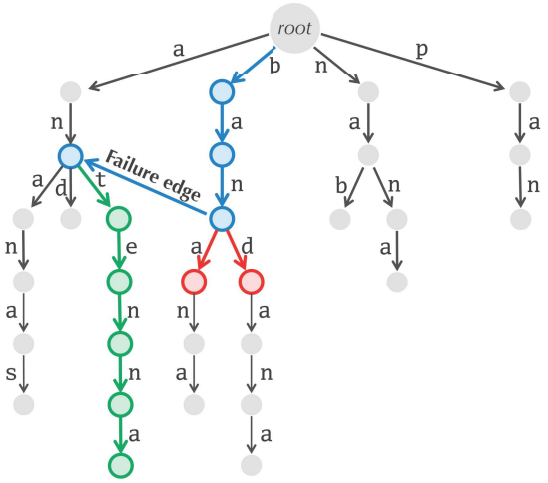

FIGURE 9.25 The trie from Figure 9.1 with an additional failure edge. The failure edge allows us to jump from the path beginning with "ban" to the path beginning with "an".

However, we have thrown away some important information: we *already matched* the first two symbols of "antenna" when searching through "bantenna". Thus, a more sensible strategy is to jump directly to the node "an" and then continue downward in the trie. Indeed, this strategy will eventually match the pattern "antenna". We can implement this "jumping ahead" strategy by augmenting the tree with a **failure edge** connecting node "ban" to node "an" (Figure 9.25). More generally, failure edges are formed by connecting node v to node w if w is the longest suffix of v that appears in the trie. The Aho-Corasick algorithm follows failure edges whenever a mismatch is found in order to avoid going all the way back to the root, thus saving time.

From suffix trees to suffix arrays

We can construct SUFFIXARRAY(*Text*) from SUFFIXTREE(*Text*) by applying a **preorder traversal** of SUFFIXTREE(*Text*) as long as the outgoing edges from every node of the suffix tree are arranged lexicographically.

Given a rooted tree, a node w is called a **child** of a node v if there is an edge connecting v to w in the tree. The preorder traversal of a tree involves visiting a node of the tree, starting at the root, and then recursively preorder traversing the subtrees rooted at each of its children from left to right (Figure 9.26), which is accomplished by the following pseudocode. By taking the order of leaves visited in a preorder traversal of the suffix tree, we obtain the suffix array (consult Figure 9.6 and Figure 9.7).

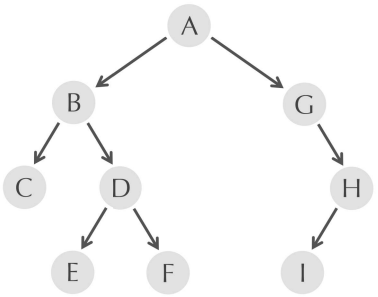

FIGURE 9.26 The preorder traversal of the above tree visits its nodes in increasing order of their labels.

```
PREORDER(Tree, Node)
    visit Node
    for each child Node' of Node from left to right
        PREORDER(Tree, Node')
```

Conversely, SUFFIXTREE(*Text*) can be constructed from SUFFIXARRAY(*Text*) in linear time by using the **longest common prefix (LCP) array** of *Text*, LCP(*Text*), which stores the length of the longest common prefix shared by consecutive lexicographically ordered suffixes of *Text*. For example, LCP(`"panamabananas$"`) is $(0, 0, 1, 1, 3, 3, 1, 0, 0, 0, 2, 2, 0, 0)$, as shown in Figure 9.27.

Suffix Tree Construction from Suffix Array Problem:
Construct a suffix tree from the suffix array and LCP array of a string.

Input: A string *Text*, its suffix array, and its LCP array.
Output: The suffix tree of *Text*.

LCP Array	Sorted Suffixes
0	$
0	abananas$
1	amabananas$
1	anamabananas$
3	ananas$
3	anas$
1	as$
0	bananas$
0	mabananas$
0	namabananas$
2	nanas$
2	nas$
0	panamabananas$
0	s$

FIGURE 9.27 The LCP array of "panamabananas$" is formed by sorting the suffixes of "panamabananas$" lexicographically and then finding the length of the longest common prefix shared by consecutive suffixes according to the lexicographic order.

From suffix arrays to suffix trees

Given the suffix array SUFFIXARRAY and LCP array LCP of a string *Text*, the suffix tree SUFFIXTREE(*Text*) can be constructed in linear time using the algorithm illustrated in Figure 9.28. After constructing a **partial suffix tree** for the i lexicographically smallest suffixes (denoted SUFFIXTREE$_i$(*Text*)), this algorithm iteratively inserts the $(i+1)$-th suffix into this tree to form SUFFIXTREE$_{i+1}$(*Text*).

We define the **descent** of a node v in a suffix tree, denoted DESCENT(v), as the length of the concatenation of all path labels from the root to this node. We assume that the descents of all nodes in the growing partial suffix tree have been precomputed.

We start with SUFFIXTREE$_0$(*Text*), which we define as the tree consisting only of the root. To insert the $(i+1)$-th suffix (corresponding to the element SUFFIXARRAY$(i+1)$ in the suffix array of *Text*) into SUFFIXTREE$_i$(*Text*), we need to know where the path representing this suffix "splits" from the already constructed partial suffix tree

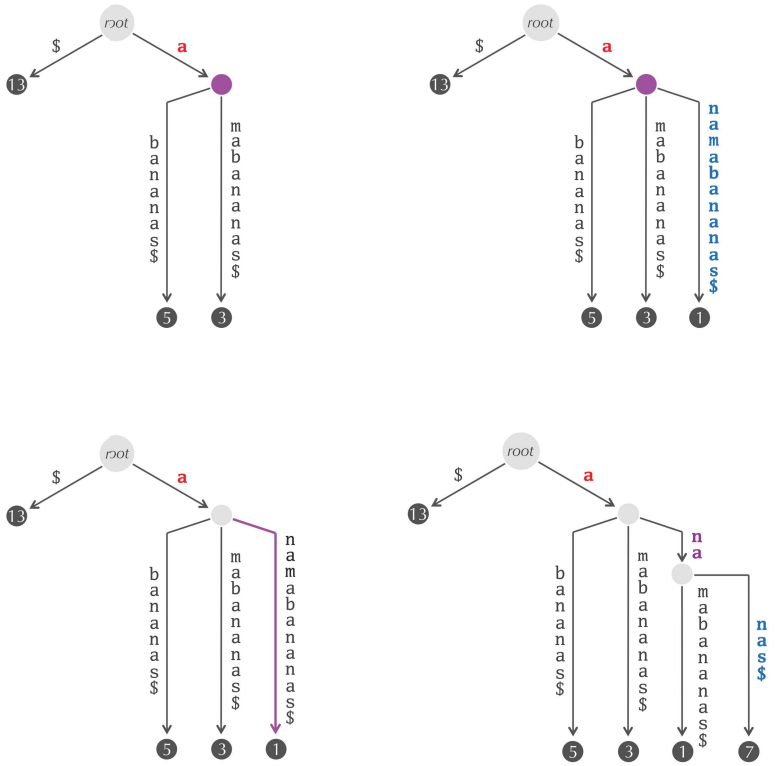

FIGURE 9.28 (Top left) SUFFIXTREE$_3$(*Text*) for *Text* = "panamabananas$". The fourth lexicographically ordered suffix of *Text* is "**anamabananas$**", and we can thread only its first letter into this suffix tree. Our stopping point is at the purple node, so we create a new node branching off this node with edge labeled "**namabananas$**" to obtain SUFFIXTREE$_4$(*Text*) (top right). (Bottom left) The fifth lexicographically ordered suffix of *Text* is "**ananas$**", the first three symbols of which we can thread into SUFFIXTREE$_4$(*Text*) until we reach a stopping point in the middle of the edge "**na**mabananas$". (Bottom right) To form SUFFIXTREE$_5$(*Text*), we create a new node in the middle of the purple edge and branch from this node into two edges: one labeled "mabananas$" in order to retain the suffix "anamabananas$" and another labeled "**nas$**" to spell the new suffix "**ananas$**". In general, the number of red symbols that we spell down in SUFFIXTREE$_i$(*Text*) to form SUFFIXTREE$_{i+1}$(*Text*) is given by the $(i+1)$-th entry in the LCP array from Figure 9.27.

SUFFIXTREE$_i$(*Text*). For example, this split happens at the purple node while inserting "anamabananas$" into SUFFIXTREE$_3$(*Text*) (Figure 9.28 (top)) and at the purple edge labeled "namabananas$" while inserting "ananas$" into SUFFIXTREE$_4$(*Text*), thus breaking this edge into edges labeled "na" and "mabananas$" in SUFFIXTREE$_5$(*Text*) (Figure 9.28 (bottom)).

STOP and Think: How would you find the node or edge where the partial suffix tree splits while constructing the suffix tree from the suffix array and LCP array?

To find the node/edge where the partial suffix tree splits, we will walk up the rightmost path (i.e., the last added path in the partial suffix tree) in the partial suffix tree beginning at the previously inserted leaf, labeled SUFFIXARRAY(i), to the root. We stop when we encounter the first node v such that DESCENT$(v) \leq$ LCP$(i+1)$. Afterwards, we have to consider two cases depending on whether DESCENT$(v) =$ LCP$(i+1)$ (the split occurs at the node v) or DESCENT$(v) <$ LCP$(i+1)$ (the split occurs on the edge leading from v):

- DESCENT$(v) =$ LCP$(i+1)$: the concatenation of the labels on the path from the root to v equals the longest common prefix of suffixes corresponding to SUFFIXARRAY(i) and SUFFIXARRAY$(i+1)$. We insert SUFFIXARRAY$(i+1)$ as a new leaf x connected to v, and we label the edge (v, x) with the suffix of *Text* starting at position SUFFIXARRAY$(i+1)$ + LCP$(i+1)$. Thus, the edge label consists of the remaining symbols of the suffix corresponding to SUFFIXARRAY$(i+1)$ that are not already represented by the concatenation of the labels of the path connecting the root to v. This completes the construction of the partial suffix tree SUFFIXTREE$_{i+1}$(*Text*) (see Figure 9.28 (top) for an example).
- DESCENT$(v) <$ LCP$(i+1)$: the concatenation of the labels on the path from the root to v has fewer symbols than the longest common prefix of the suffixes corresponding to SUFFIXARRAY(i) and SUFFIXARRAY$(i+1)$. The question therefore arises of how to recover these missing symbols. We denote the rightmost edge leading from v in SUFFIXTREE$_i$(*Text*) as (v, w) and argue that the missing symbols represent a prefix of this edge's label. In this case, we split this edge and construct SUFFIXTREE$_{i+1}$(*Text*) as described below (see Figure 9.28 (bottom) for an example):

  1. Delete the edge (v, w) from SUFFIXTREE$_i$(*Text*).
  2. Add a new internal node y and a new edge (v, y) labeled by a substring of *Text* starting at position SUFFIXARRAY$(i+1)$ + DESCENT(v). The new label is formed by the final LCP$(i+1)$ − DESCENT(v) symbols of the longest common prefix of SUFFIXARRAY(i) and SUFFIXARRAY$(i+1)$. Thus, the concatenation of the labels on the path from the root to y is now the longest common prefix of SUFFIXARRAY(i) and SUFFIXARRAY$(i+1)$.

3. Define DESCENT(y) as LCP$(i+1)$.
4. Connect w to the newly created internal node y by an edge (y, w) that is labeled by a substring of *Text* starting at position SUFFIXARRAY(i) + LCP$(i+1)$ and ending at position SUFFIXARRAY(i) + DESCENT$(w) - 1$. The new label consists of the remaining symbols of the deleted edge (v, w) that were not used as the label of edge (v, y).
5. Add SUFFIXARRAY$(i+1)$ as a new leaf x as well as an edge (y, x) that is labeled by a suffix of *Text* beginning at position SUFFIXARRAY$(i+1)$ + LCP$(i+1)$. The label of this edge consists of the remaining symbols of the suffix corresponding to SUFFIXARRAY$(i+1)$ that are not already represented by the concatenation of the labels on the path from the root to v.

EXERCISE BREAK: Prove that the running time of this algorithm is $\mathcal{O}(|Text|)$.

Binary search

The game show *The Price is Right* features a timed challenge called the "Clock Game" in which a contestant makes repeated guesses at the price of an item, with the host telling the contestant only whether the true price is higher or lower than the most recent guess.

An intelligent strategy for the Clock Game is to pick a sensible range of prices within which the item's price must fall, and then guess a price halfway between these two extremes. If this guess is incorrect, then the contestant has immediately eliminated half of the set of possible prices. The contestant then makes a guess in the middle of the remaining possible prices, eliminating half of them again. Iterating this strategy quickly yields the price of the item.

This strategy for the Clock Game motivates a binary search algorithm finding the position of an element *key* within a sorted array ARRAY. This algorithm, called **BINARYSEARCH**, is initialized by setting *minIndex* equal to 0 and *maxIndex* equal to the length of ARRAY. It sets *midIndex* equal to $(minIndex + maxIndex)/2$ and then checks to see whether *key* is greater than or less than ARRAY$(midIndex)$. If *key* is larger than this value, then **BINARYSEARCH** iterates on the subarray of ARRAY from *minIndex* to $midIndex - 1$; otherwise, **BINARYSEARCH** iterates on the subarray of ARRAY from $midIndex + 1$ to *maxIndex*. Iteration eventually identifies the position of *key*.

For example, if $key = 9$ and ARRAY $= (1, 3, 7, 8, 9, 12, 15)$, then **BINARYSEARCH** would first set *minIndex* equal to 0, *maxIndex* equal to 6, and *midIndex* equal to 3. Because *key* is greater than ARRAY$(midIndex) = 8$, we examine the subarray whose elements

are greater than ARRAY($midIndex$) by setting *minIndex* equal to 4, so that *midIndex* is recomputed as $(4+6)/2 = 5$. This time, *key* is smaller than ARRAY($midIndex$) $= 12$, and so we examine the subarray whose elements are smaller than this value. This subarray consists of only a single element, which is *key*.

```
BINARYSEARCH(ARRAY, key, minIndex, maxIndex)
    while maxIndex ≥ minIndex
        midIndex ← (minIndex + maxIndex)/2
        if ARRAY(midIndex) = key
            return midIndex
        else if ARRAY(midIndex) < key
            minIndex ← midIndex + 1
        else
            maxIndex ← midIndex − 1
    return "key not found"
```

Bibliography Notes

The Aho-Corasick algorithm was introduced by Aho and Corasick, 1975. Suffix trees were introduced by Weiner, 1973. Suffix arrays were introduced by Manber and Myers, 1990. The Burrows-Wheeler Transform was introduced by Burrows and Wheeler, 1994. An efficient implementation of the Burrows-Wheeler transform was described by Ferragina and Manzini, 2000. The genetic cause of Ohdo syndrome was elucidated by Clayton-Smith et al., 2011. Rearrangements and indels in the human genome were studied by Tuzun et al., 2005 and Montgomery et al., 2013. BLAST, the dominant database search tool in molecular biology, was developed by Altschul et al., 1990.