COMBINATORIAL PATTERN MATCHING

Combinatorial pattern matching, on the other hand, looks for exact or approximate occurrences of given patterns in a long text. Although pattern matching is in some ways simpler than motif finding since we actually know what we are looking for, the large size of genomes makes the problem, in practice, difficult. The alignment techniques become impractical for whole genomes,particularly when one searches for approximate occurrences of many long patterns at one time. In this chapter we develop a number of ways to make pattern matching in a long string practical. For example, suppose you were in a library with 2 million volumes in no discernible order and you needed to find one particular title. The only way to guarantee finding the title would be to check every book in the library. However, if the books in the library were sorted by title, then the check becomes very easy.A sorted list is only one of many types of data structures, and significantly more sophisticated ways of organizing data.

Repeat Finding

Many genetic diseases are associated with deletions, duplications, and rearrangements of long chromosomal regions. These are dramatic events that affect the large-scale genomic architecture and may involve millions of nuleotides. For example, DiGeorge syndrome, which commonly results in an impaired immune system and heart defects, is associated with a large 3 Mbdeletion on human chromosome 22. A deletion of this size is likely to remove important genes and lead to disease.Such dramatic changes in genomic architecture often require—as in the case of DiGeorge syndrome—a pair of very similar sequences flanking the deleted segment. These similar sequences form a repeat in DNA, and it is important to find all repeats in a genome. Repeats in DNA hold many evolutionary secrets. A striking and still unexplained phenomenon is the large number of repeats in many genomes: for example, repeats account for about 50% of the human genome. Algorithms that search for repeats in the human genome need to analyze a 3 billion nucleotide genome, and quadratic sequence alignment algorithms are too slow for this. The simplest approach to finding exact repeats is to construct a table that holds, for each l-mer, all the positions where the l-mer is located in the genomic DNA sequence. Such a table would contain 4 containing some number between 0 and M of positions, where M is the frequency of occurrence of the most common l-mer in the genome. The average number of elements in each bin is n In many applications, the parameter l varies from 10 to 13, so this table is not unmanageably large. Although this tabulation approach allows one to quickly find all repeats of length l, such short repeats are not very interesting for DNA analysis. Biologists instead are interested in long maximal repeats,that is, repeats that cannot be extended to the left or to the right. To find maximal repeats that are longer than a predefined parameter L, each exact repeat of

length l must be extended to the left or to the right to see whether it is embedded in a repeat longer than L. Since typically l << L, there are many more repeats to be extended than there are maximal repeats to be found. For example, the bacterial Escherichia coli genome of 4.6 million nucleotides has millions of repeats of length l = 12 but only about 8000 maximal repeats of length L = 20 or longer.1 Thus, most of the work in this approach to repeat finding is wasted trying to pointlessly extend short repeats. The popular RE-Puter algorithm gets around this by using a suffix tree.

Hashing

Hashing is a very efficient way to store and retrieve data.

Hashing: Definitions

• **Records**: data stored in a hash table.

• **Keys**: Integers identifying set of records.

• **Hash Table**: The array of keys used in hashing.

• **Hash Function**: Assigns each record to a key.

• **Collision**: Occurs when more than one record is mapped to the same index in the hash table.

Hashing DNA sequences

• Each l-mer can be translated into a binary string:

• A can be represented as- 00

• T can be represented as -01

• G can be represented as- 10

• C can be represented as- 11

• Example: ATGCTA = 000110110100

After assigning a unique integer to each l-mer, it is easy to obtain all start locations of each l-mer in a genome. An Introduction to Bioinformatics Algorithms www.bioalgorithms.info Hashing: Maximal Repeats To find repeats in a genome:

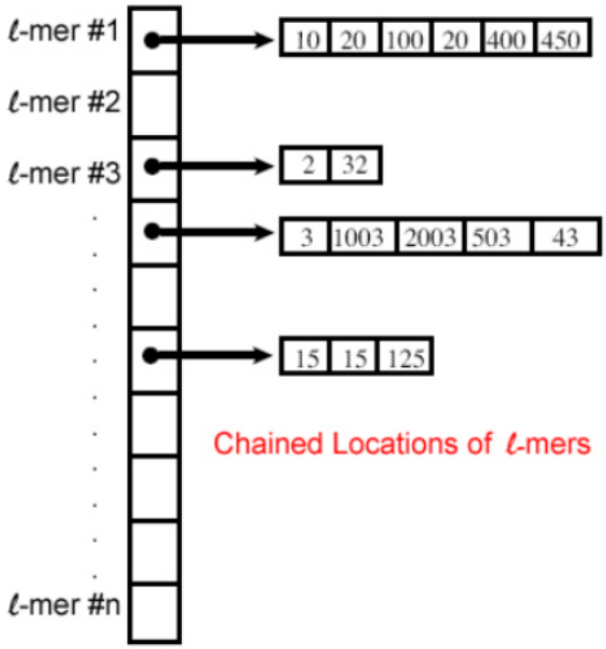
1. For all l-mers in the genome, note the start position and the sequence.

2. Generate a hash table index for each unique l-mer sequence.

3. At each index of the hash table, store all genome start locations of the l-mer that generated the index. 4. Anywhere there are collisions in the table, extend corresponding l-mers to maximal repeats.

Dealing with collisions

An Introduction to Bioinformatics Algorithms www.bioalgorithms.info Hashing: Collisions . In order to deal with collisions: "Chain" all start locations of l-mers. This can be done via a data structure called a linked list.



Exact Pattern Matching

A common problem in bioinformatics is to search a database of sequences for a known sequence. Given a pattern string p = p1 · · · pn and a longer text string t = t1 · · ·tm, the Pattern Matching problem is to find any and all occurrences of pattern p in text t.

Exact Pattern Matching Problem:

Goal: Find all occurrences of a pattern in a text.

Input:

• Pattern p = p1…pn of length n

• Text t = t1…tm of length m

• Output: All positions 1< i < (m – n + 1) such that the n-letter substring of text t starting at i matches the pattern p.

• Motivation: Searching database for a known pattern.

A Brute Force Algorithm

PatternMatching(p,t)

PatternMatching(p,t)

```
1  n=  length of pattern p
2  m= length of text t
3  for i =1  to (m – n + 1)
4   if ti …ti+n-1 = p
5  5 output i
```

Example

PatternMatching algorithm for:

• Pattern GCAT

• Text AGCCGCATCT

- AGCCGCATCT
- GCAT

• AGCCGCATCT

- GCAT

• AGCCGCATCT

- GCAT

• AGCCGCATCT

- GCAT

• AGCCGCATCT

- GCAT

• AGCCGCATCT

- GCAT

• AGCCGCATCT

- GCAT

Suffix Trees

Suffix trees allow one to preprocess a text in such a way that for any pattern of length n, one can answer whether or not it occurs in the text, using only O(n) time, regardless of how long

the text is. That is, if you created one gigantic book out of all the issues of the journal Science, you could search for a pattern p in time proportional to the length of p.

Use of Suffix Trees

- Suffix trees hold all suffixes of a text i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
- Builds in O(m) time for text of length m
- To find any pattern of length n in a text:
- Build suffix tree for text
- Thread the pattern through the suffix tree
- Can find pattern in text in O(n) time!
- O(n + m) time for "Pattern Matching Problem"
- Build suffix tree and lookup pattern

Suffix Trees: Example

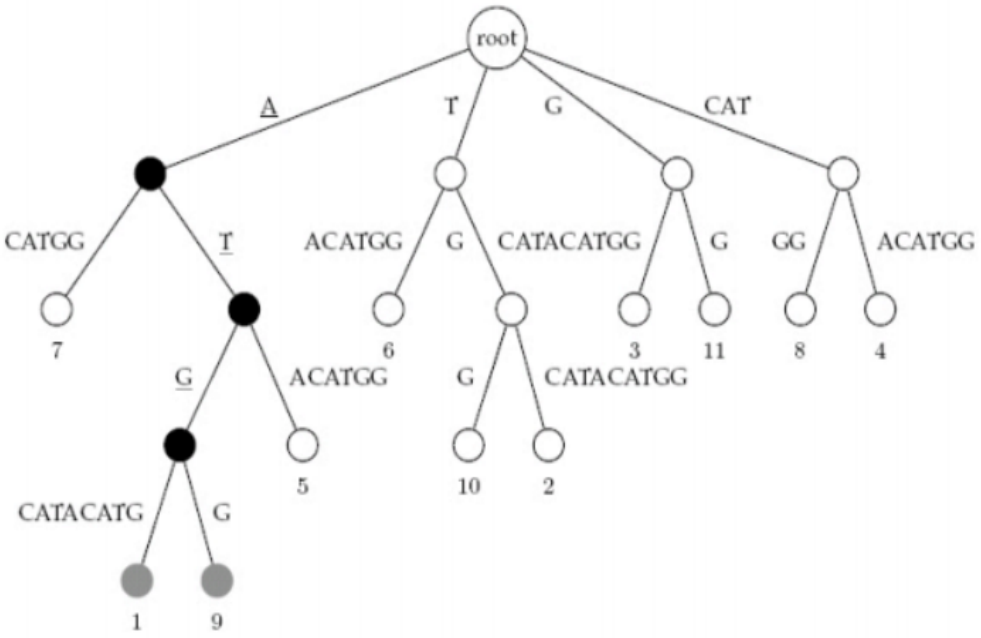


Figure 9.6 Threading the pattern ATG through the suffix tree for the text ATGCATACATGG. The suffixes ATGCATACATGG and ATGG both match, as noted by the gray vertices in the tree (the p-matching leaves). Each *p*-matching leaf corresponds to a position in the text where p occurs.

CLUSTERING PRINCIPLES

Homogeneity: elements of the same cluster are maximally close to each other

Separation: elements in separate clusters are maximally far apart from each other

CLUSTERING TECHNIQUES

Agglomerative: Start with every element in its own cluster, and iteratively join clusters together

Divisive: Start with one cluster and iteratively divide it into smaller clusters

Hierarchical: Organize elements into a tree, leaves represent genes and the length of the paths between leaves represents the distances between genes. Similar genes lie within the same subtrees

Applications

- Build regulatory networks
- Discover subtypes of a disease
- Infer unknown gene function
- Reduce dimensionality

Hierarchial Clustering:

Hierarchical clustering, the most frequently used mathematical technique, attempts to group genes into small clusters and to group clusters into higher-level systems. The resulting hierarchical tree is easily viewed as a dendrogram

Most studies involve comparing a series of experiments to identify genes that are consistently coregulated under some defined set of circumstances—disease state, increasing time, increasing drug dose, etc. A two-dimensional grid is constructed with each row corresponding to a different gene sequence and each column to a different set of experimental conditions. Each set of gene-

expression levels (each row in the matrix) is compared to every other set of expression levels in a pairwise fashion, and similarity scores are produced in the form of statistical correlation coefficients. These correlation coefficients can be thought of as representing the Euclidean distances between the rows in the matrix. The correlations are ordered, and a node is created between the highest-scoring (geometrically closest) pair of rows—the two gene sequences that were most nearly coregulated across each of the experiments. The matrix is then modified to represent the joined elements as a single node, and all distances between the newly formed node and other gene sequences (rows) in the matrix are calculated. It is not necessary to recalculate all correlations because only those involving the two rows joined in the new node have changed. Typically, the node is represented by a link in the dendrogram, the height of the link being directly proportional to the strength of the correlation. The process of creating proportional links and joining genes into clusters continues until all genes in the experiment have been joined into a single hierarchical cluster through links of appropriate length. If more than two nodes are related by the same correlation coefficient (same geometric distance), the conflict is resolved according to a predetermined set of rules.

CLUSTERING PROCEDURE

At each step, we select the two closest sequences and join them to form a clade.

We then replace the two just joined sequences with their ancestor

This reduces the size of the data matrix by one

We need to compute the distances from the new ancestor to the remaining sequences

UPDATING DISTANCES

There are multiple strategies for computing the distances to the new 'ancestral' sequence a that joins sequences m and n

Single linkage:

$d(x,a) = \min [d(x,m), d(x,n)]$

Complete linkage:

$d(x,a) = \max [d(x,m), d(x,n)]$

Advantages of Hierarchical Clustering

It is sometimes meaningful to cluster data at the experiment level rather than at the level of individual genes. Such experiments are most often used to identify similarities in overall gene-expression patterns in the context of different treatment regimens—the goal being to stratify patients based on their molecular-level responses to the treatments. The hierarchical techniques outlined earlier are appropriate for such clustering, which is based on the pairwise statistical comparison of complete scatterplots rather than individual gene sequences. The data are represented as a matrix of scatterplots, ultimately reduced to a matrix of correlation coefficients. The correlation coefficients are then used to construct a two-dimensional dendrogram

A significant example that illustrates the utility of hierarchical clustering involves the identification of distinct tumor subclasses in diffuse large B-cell lymphoma (*DLBCL*). Two distinct forms of *DLBCL* have been identified using hierarchical clustering techniques, each related to a different stage of B-cell differentiation. The fact that the cluster correlates are significant is demonstrated by direct relationships to patient survival rates

Disadvantages of Hierarchical Clustering

Despite its proven utility, hierarchical clustering has many flaws. Interpretation of the hierarchy is complex and often confusing; the deterministic nature of the technique prevents reevaluation after points are grouped into a node; all determinations are strictly based on local decisions and a single pass of analysis; it has been demonstrated that the tree structure can lock in accidental features reflecting idiosyncrasies of the clustering rules; expression patterns of individual gene sequences become less relevant as the clustering process progresses; and an incorrect assignment made early in the process cannot be corrected [[14]]. These deficiencies have driven the development of additional clustering techniques that are based on multiple passes of analysis and utilize advanced algorithms borrowed from the artificial intelligence community. Two of these techniques, k-means clustering and self-organizing maps (SOMs), have achieved widespread acceptance in research oncology where they have been enormously successful in identifying meaningful genetic differences between patient populations.

Regardless of the graphical technique used, clustering algorithms can be collectively viewed as a mechanism for defining boundaries that partition vectors into meaningful groups. Datasets with many dimensions are often visualized in a simple two-dimensional plot—time or experiment number on the x-axis and expression ratio on the y-axis.

K-Means Clustering

K-means clustering is most useful when the number of clusters that should be represented is known. An example might include microarray classification of a group of patients that have morphologically similar diseases that fall into three clinically distinct categories (k=3). The clustering process would proceed as follows:

1. Each expression vector is randomly assigned to one of three groups or clusters (k=3).

2. An average expression vector (called the center) is calculated for each group, and these vectors are used to compute the distances between groups.

3. Each gene-expression vector is reassigned to the group whose center is closest. Expression vectors are allowed to remain in a cluster only when they are closer to the center of that cluster than to a neighboring one.

4. Inter- and intracluster distances are recomputed, and new expression vectors for the center of each cluster are calculated.

5. The process is repeated until all expression vectors are optimally placed. At this point, any additional changes would increase intracluster distances while decreasing intercluster dissimilarity.

EX: LLOYD'S ALGORITHM

1. Arbitrarily assign the K cluster centers (this can significantly influence the outcome)

2. while cluster centers keep changing

A. Compute the distance from each data point to the current cluster center Ci

$(1 \leq i \leq K)$ and assign the point to the nearest cluster

B. After the assignment of all datapoints, compute new centers for each cluster by taking the centroid of all the points in that cluster

3. Output cluster centers and assignments

Lloyd algorithm is fast but in each iteration it moves many data points not necessarily causing better convergence.

A more conservative method would be to move one point at a time only if it improves the overall clustering cost

The smaller the clustering cost of a partition of data points is the better that clustering is

Different methods (e.g. the squared error distortion) can be used to measure this clustering cost

The Advantages of K-Means Clustering

K-means clustering has proven to be a valuable tool for identifying coregulated genes in systems where biochemical or clinical knowledge can be used to predict the appropriate number of clusters. The tool is also useful when the number of appropriate clusters is unknown if the researcher experiments with different values of *k*.

The Disadvantages of K-Means Clustering

Despite these advantages, the unstructured nature of the k-means clustering technique tends to proceed in a local fashion, and this effect intensifies as additional clustering centers are added to the analysis. Excessive locality eventually leads to incorrect groupings, and important gene associations can be lost. It follows that as the number of clustering centers is increased, initial placement of the centers becomes increasingly critical; for analyses that involve large numbers of clustering centers, it makes sense to use more-structured techniques. Algorithms based on

self-organizing maps solve many of these problems and have demonstrated tremendous utility in anticancer drug discovery and research oncology.

Algorithms and Complexity

We will see how popular bioinformatics algorithms work and we will see what principles drove their design. It is important to understand how an algorithm works in order to be confident in its results; it is even more important to understand an algorithm's design methodology in order to identify its potential weaknesses and fix them.

Before considering any algorithms in detail, we need to define loosely what we mean by the word "algorithm" and what might qualify as one. In many places throughout this text we try to avoid tedious mathematical formalisms, yet leave intact the rigor and intuition behind the important concept.

What Is an Algorithm?

Roughly speaking, an *algorithm* is a sequence of instructions that one must perform in order to solve a well-formulated *problem*. We will specify problems in terms of their *inputs* and their *outputs*, and the algorithm will be the method of translating the inputs into the outputs. A well-formulated problem is unambiguous and precise, leaving no room for misinterpretation.

In order to solve a problem, some entity needs to carry out the steps specified by the algorithm. A human with a pen and paper would be able to do this, but humans are generally slow, make mistakes, and prefer not to perform repetitive work. A computer is less intelligent but can perform simple steps quickly and reliably. A computer cannot understand English, so algorithms must be rephrased in a programming language such as C or Java in order to give specific instructions to the processor. Every detail must be specified to the computer in exactly the right format, making it difficult to describe algorithms; trifling details that a person would naturally understand must be specified. If a computer were to put on shoes, one would need to tell it to find a pair that both matches and fits, to put the left shoe on the left foot, the right shoe on the right, and to tie the laces.[1] In this book, however, we prefer to simply leave it at "Put on a pair of shoes."

However, to understand how an algorithm works, we need some way of listing the steps that the algorithm takes, while being neither too vague nor too formal. We will use *pseudocode*, whose elementary operations are summarized below. Pseudocode is a language computer scientists often use to describe algorithms: it ignores many of the details that are required in a programming language, yet it is more precise and less ambiguous than, say, a recipe in a cookbook. Individually, the operations do not solve any particularly difficult problems, but they can be grouped together into minialgorithms called *subroutines* that do.

In our particular flavor of pseudocode, we use the concepts of *variables*, *arrays*, and *arguments*. A variable, written as x or $total$, contains some numerical value and can be assigned a new numerical value at different points throughout the course of an algorithm. An array of n elements is an ordered collection of n variables $a_1, a_2, \ldots, a_n$. We usually denote arrays by boldface letters like $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and write the individual elements as a_i where i is between 1 and n. An algorithm in pseudocode is denoted by a name, followed by the list of arguments that it requires, like $\textsc{Max}(a, b)$ below; this is followed by the statements that describe the algorithm's actions.

Arithmetic

Format: $a + b$, $a - b$, $a \cdot b$, a/b, a^b

Effect: Addition, subtraction, multiplication, division, and exponentiation of numbers.

Example: $\textsc{Dist}(x1, y1, x2, y2)$
1 $dx \leftarrow (x2 - x1)^2$
2 $dy \leftarrow (y2 - y1)^2$
3 **return** $\sqrt{(dx + dy)}$

Result: $\textsc{Dist}(x1, y1, x2, y2)$ computes the Euclidean distance between points with coordinates $(x1, y1)$ and $(x2, y2)$. $\textsc{Distance}(0, 0, 3, 4)$ returns 5.

Conditional

Format:
if A is true
    B
else
    C

Effect: If statement A is true, executes instructions **B**, otherwise executes instructions **C**. Sometimes we will omit "**else C**," in which case this will either execute **B** or not, depending on whether A is true.

Example:
```
MAX(a, b)
1 if a < b
2     return b
3 else
4     return a
```

Result: MAX(a, b) computes the maximum of the numbers a and b. For example, MAX$(1, 99)$ returns 99.

`for` loops

Format:
for $i \leftarrow a$ **to** b
    B

Effect: Sets i to a and executes instructions **B**. Sets i to $a + 1$ and executes instructions **B** again. Repeats for $i = a + 2, a + 3, \ldots, b - 1, b$.[3]

Example:
```
SUMINTEGERS(n)
1 sum ← 0
2 for i ← 1 to n
3     sum ← sum + i
4 return sum
```

Result: SUMINTEGERS(n) computes the sum of integers from 1 to n. SUMINTEGERS(10) returns $1 + 2 + \cdots + 10 = 55$.

`while` loops

Format: **while** A is true
B

Effect: Checks the condition A. If it is true, then executes instructions **B**. Checks A again; if it's true, it executes **B** again. Repeats until A is not true.

Example:
```
ADDUNTIL(b)
1 i ← 1
2 total ← i
3 while total ≤ b
4       i ← i + 1
5       total ← total + i
6 return i
```

Result: ADDUNTIL(b) computes the smallest integer i such that $1 + 2 + \cdots + i$ is larger than b. For example, ADDUNTIL(25) returns 7, since

$1+2+\cdots+7 = 28$, which is larger than 25, but $1+2+\cdots+6 = 21$, which is smaller than 25.

Array access

Format: a_i

Effect: The ith number of array $\mathbf{a} = (a_1, \ldots a_i, \ldots a_n)$. For example, if $\mathbf{F} = (1, 1, 2, 3, 5, 8, 13)$, then $F_3 = 2$, and $F_4 = 3$.

Example: FIBONACCI(n)

```
1 F_1 ← 1
2 F_2 ← 1
3 for i ← 3 to n
4       F_i ← F_{i-1} + F_{i-2}
5 return F_n
```

Result: FIBONACCI(n) computes the nth Fibonacci number. FIBONACCI(8) returns 21.

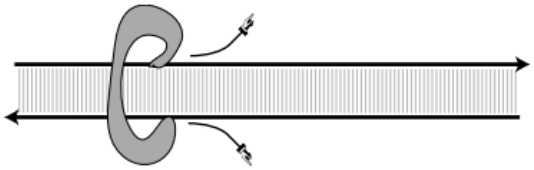
While computer scientists are accustomed to the pseudocode jargon above, we fear that some biologists reading it might decide that this book is too cryptic and therefore useless. Although modern biologists deal with algorithms on a daily basis, the language they use to describe an algorithm might be closer to the language used in a cookbook, like the pumpkin pie recipe in figure 2.1. Accordingly, some bioinformatics books are written in this familiar lingo as an effort to make biologists feel at home with different bioinformatics concepts. Unfortunately, the cookbook language is insufficient to describe more complex algorithmic ideas that are necessary for even the simplest tools in bioinformatics. The problem is that natural languages are not suitable for communicating algorithmic ideas more complex than the pumpkin pie.

Biological Algorithms versus Computer Algorithms

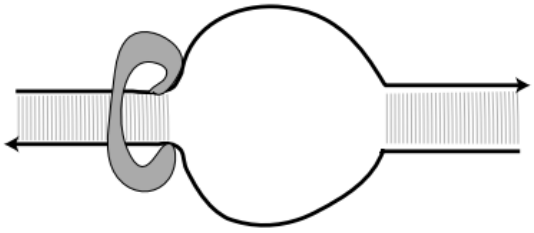
Nature uses algorithm-like procedures to solve biological problems, for example, in the process of *DNA replication*. Before a cell can divide, it must first make a complete copy of all its genetic material.

DNA replication proceeds in phases, each of which requires an elaborate cooperation between different types of molecules. For the sake of simplicity, we describe the replication process as it occurs in bacteria, rather than the replication process in humans or other mammals, which is quite a bit more involved. The basic mechanism was proposed by James Watson and Francis Crick in the early 1950s, but could only be verified through the ingenious Meselson-Stahl experiment of 1957. The replication process starts from a pair of complementary[5] strands of DNA and ends up with two pairs of complementary strands.[6]

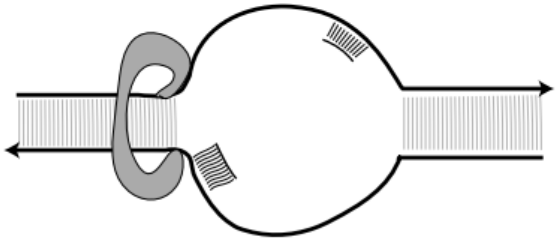
1. A molecular machine (in other words, a protein complex) called a *DNA helicase*, binds to the DNA at certain positions called *replication origins*.



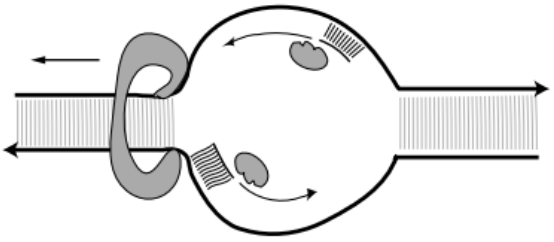
2. Helicase wrenches apart the two strands of DNA, creating a so-called *replication fork*. The two strands are complementary and run in opposite directions (one strand is denoted $3' \rightarrow 5'$, the other $5' \rightarrow 3'$). Two other molecular machines, *topoisomerase* and *single-strand binding protein* (not shown) bind to the single strands to help relieve the instability of single-stranded DNA.



3. *Primers*, which are short single strands of RNA, are synthesized by a protein complex called *primase* and latch on to specific positions in the newly opened strands, providing an anchor for the next step. Without primers, the next step cannot begin.

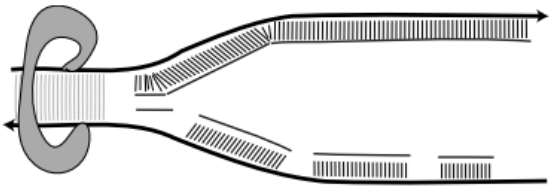


4. A *DNA polymerase* (yet another molecular machine) binds to each freshly separated *template* strand of the DNA; the DNA polymerase traverses the parent strands only in the $3' \rightarrow 5'$ direction. Therefore, the DNA polymerases attached to the two DNA strands move in opposite directions.

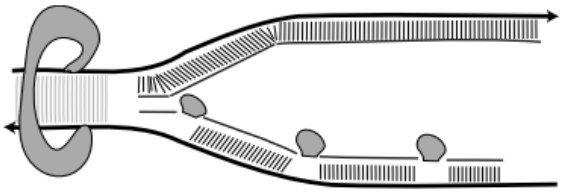


5. At each nucleotide, DNA polymerase matches the template strand's nucleotide with the complementary base, and adds it to the growing synthesized chain. Prior to moving to the next nucleotide, DNA polymerase checks to ensure that the correct base has been paired at the current position; if not, it removes the incorrect base and retries.

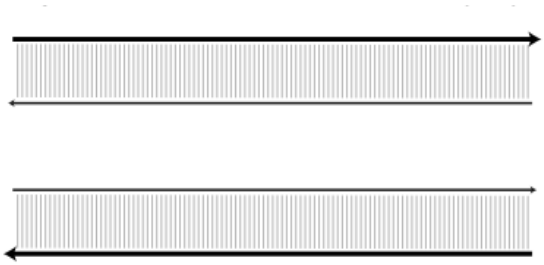
   Since DNA polymerase can only traverse DNA in the $3' \rightarrow 5'$ direction, and since the two strands of DNA run in opposite directions, only one strand of the template DNA can be used by polymerase to continuously synthesize its complement; the other strand requires occasional stopping and restarting. This results in short segments called *Okazaki fragments*.



6. Another molecular machine, *DNA ligase*, repairs the gaps in the newly synthesized DNA's backbone, effectively linking together all Okazaki fragments into a single molecule and cleaning any breaks in the primary strand.



7. When all the DNA has been copied in such a manner, the original strands separate, so that two pairs of DNA strands are formed, each pair consisting of one old and one newly synthesized strand.



Obviously, an astounding amount of molecular logistics is required to ensure completely accurate DNA replication: DNA helicase separates strands, DNA polymerase ensures proper complementarity, and so on. However, in terms of the logic of the process, none of this complicated molecular machinery actually matters—to mimic this process in an algorithm we simply need to take a string which represents the DNA and return a copy of it.

String Duplication Problem:
Given a string of letters, return a copy.

Input: A string $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ of length n, as an array of characters.

Output: A string representing a copy of $\mathbf{s}$.

Of course, this is a particularly easy problem to solve and yields absolutely no interesting algorithmic intuition. However it is still illustrative to write the pseudocode. The STRINGCOPY program below uses the string $\mathbf{t}$ to hold a copy of the input string $\mathbf{s}$, and returns the result $\mathbf{t}$.

```
STRINGCOPY(s, n)
1  for i ← 1 to n
2        t_i ← s_i
3  return t
```

While STRINGCOPY is a trivial algorithm, the number of operations that a real computer performs to copy a string is surprisingly large. For one partic-

ular computer architecture, we may end up issuing thousands of instructions to a computer processor. Computer scientists distance themselves from this complexity by inventing programming languages that allow one to ignore many of these details. Biologists have not yet invented a similar "language" to describe biological algorithms working in the cell.

The amount of "intelligence" that the simplest organism, such as a bacterium, exhibits to perform any routine task—including replication—is amazing. Unlike StringCopy, which only performs abstract operations, the bacterium really *builds* new DNA using materials that are floating near the replication fork. What would happen if it ran out? To prevent this, a bacterium examines the surroundings, imports new materials from outside, or moves off to forage for food. Moreover, it waits to begin copying its DNA until sufficient materials are available. These observations, let alone the coordination between the individual molecules, lead us to wonder whether even the most sophisticated computer programs can match the complicated behavior displayed by even a single-celled organism.

Recursive Algorithms

Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.

The *Towers of Hanoi* puzzle, introduced in 1883 by a French mathematician, consists of three pegs, which we label from left to right as 1, 2, and 3, and a number of disks of decreasing radius, each with a hole in the center. The disks are initially stacked on the left peg (peg 1) so that smaller disks are on top of larger ones. The game is played by moving one disk at a time between pegs. You are only allowed to place smaller disks on top of larger ones, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3.

Towers of Hanoi Problem:
Output a list of moves that solves the Towers of Hanoi.

Input: An integer n.

Output: A sequence of moves that will solve the n-disk Towers of Hanoi puzzle.

Iterative versus Recursive Algorithms

Recursive algorithms can often be rewritten to use iterative loops instead, and vice versa; it is a matter of elegance and clarity that dictates which technique is easier to use. Consider the problem of sorting a list of integers into ascending order.

Sorting Problem:
Sort a list of integers.

Input: A list of n distinct integers $\mathbf{a} = (a_1, a_2, \ldots, a_n)$.

Output: Sorted list of integers, that is, a reordering $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ of integers from $\mathbf{a}$ such that $b_1 < b_2 < \cdots < b_n$.

The following algorithm, called SELECTIONSORT, is a naive but simple iterative method to solve the Sorting problem. First, SELECTIONSORT finds the smallest element in $\mathbf{a}$, and moves it to the first position by swapping it with whatever happens to be in the first position (i.e., a_1). Next, SELECTIONSORT finds the second smallest element in $\mathbf{a}$, and moves it to the second position, again by swapping with a_2. At the ith iteration, SELECTIONSORT finds the ith smallest element in $\mathbf{a}$, and moves it to the ith position. This is an intuitive approach at sorting, but is not the best-known one. If $\mathbf{a} = (7, 92, 87, 1, 4, 3, 2, 6)$, SELECTIONSORT$(\mathbf{a}, 8)$ takes the following seven steps:

$$(\overbrace{7, 92, 87, 1}, 4, 3, 2, 6)$$
$$(1, \overbrace{92, 87, 7, 4, 3, 2}, 6)$$
$$(1, 2, \overbrace{87, 7, 4, 3}, 92, 6)$$
$$(1, 2, 3, \overbrace{7, 4}, 87, 92, 6)$$
$$(1, 2, 3, 4, \overbrace{7, 87, 92, 6})$$
$$(1, 2, 3, 4, 6, \overbrace{87, 92, 7})$$
$$(1, 2, 3, 4, 6, 7, \overbrace{92, 87})$$
$$(1, 2, 3, 4, 6, 7, 87, 92)$$

Fast versus Slow Algorithms

Real computers require a certain amount of time to perform an operation such as addition, subtraction, or testing the conditions in a **while** loop. A supercomputer might take 10^{-9} second to perform an addition, while a hand calculator might take 10^{-5} second. Suppose that you had a computer that took 10^{-9} second to perform an elementary operation such as addition, and that you knew how many operations a particular algorithm would perform. You could estimate the running time of the algorithm simply by taking the product of the number of operations and the time per operation. However, computing devices are constantly improving, leading to a decreasing time per operation, so your notion of the running time would soon be outdated. Rather than computing an algorithm's running time on every computer, we rely on the total number of operations that the algorithm performs to describe its running time, since this is an attribute of the algorithm, and not an attribute of the computer you happen to be using.