Chapter 1

Simple String Search

Searching a very long sequence for a fairly short query string is a fundamental operation in processing large genome sequences. The query string may occur exactly as it is in the target sequence, or there could be variants that do not match the query exactly, but which are very similar. The former case is called exact matching, while the latter is approximate matching. This chapter describes some simple, naive algorithms for exact matching.

1.1 Storing a String in an Array

Let us consider a search of the target string `ATAATACGATAATAA` using the query `ATAA`. It is obvious that `ATAA` occurs three times in the target, but queries do not always appear in the target; e.g., consider `ACGC`. Therefore, we need an algorithm that either enumerates all occurrences of the query string together with their positions in the target, or reports the absence of the query.

For this purpose, the target and query strings are stored in a data structure called an *array*. An array is a series of elements, and each element is an object, such as a character or an integer. In order to create an array in main memory, one must declare the number of elements to obtain the space. For example, storing the target string `ATAATACGATAATAA` requires an array of 15 characters.

To access an element in an array, one specifies the position of the element, which is called the *index*. There are two major ways of indexing. *One-origin* indexing is commonly used and requires that the head element of an array has index one, the second one has index two, and so on. In programming, however, it is more typical to use *zero-origin* indexing, in which the head element has index zero.

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

Fig. 1.1 Zero-origin and one-origin indexing.

Figure 1.1 illustrates the difference between the two types of indexing. According to zero-origin (and one-origin indexing, respectively), C is in the 6th (or 7th) position, and the G in the 7th (or 8th) position is the 1st (or 2nd) element from C. In general, the j-th element from the i-th element is the $(i + j)$-th ($(i + j - 1)$-th) element in terms of zero-origin (one-origin) indexing. If an array has n elements, the last element has index $n - 1$ with zero-origin indexing. Although readers who are not familiar with zero-origin indexing may find it puzzling initially, zero-origin indexing is used in this book because most software programs adopt this form.

Let `target` denote the array in Figure 1.1. The i-th element in the `target` is specified by `target[`i`]`. For example, `target[6]` and `target[7]` are C and G in terms of zero-origin indexing.

1.2 Brute-Force String Search

Figure 1.2 presents a simple, naive program called a *brute-force search*, which searches the target string in the `target` array for all occurrences of the query string in the `query` array. The Java programming language is used to describe programs in this book [1]. The program checks whether the query string matches the substring of the same length in the target that starts at the `i`-th position for each `i` $= 0, 1, \ldots$. The indexes `i` and `j` show the current positions in the `target` and `query` that are being compared.

The outer for-loop initializes `i` to zero, and it increments `i` until `i` reaches `targetLen - queryLen`. Inside the for-loop, `j` is initially set to zero. The while-loop checks whether the query matches the substring starting from the `i`-th position in the target string. Each step compares the `j`-th character in the query with the `i+j`-th one in the target, and then moves `j` to the next position if the two letters are equal. This step is iterated until a

[1] We adopted the Java programming language because of the availability of Eclipse, an extensible development platform and application framework for building software. We have found Eclipse to be effective teaching material in a Bioinformatics programming course that we have been teaching since 2003. For better computational efficiency, it is straightforward to transform all the Java programs in this book into C/C++ programs.

```
public static void bruteSearch( int[] target, int[] query ) {
     // "target.length" returns the size of the array target.
     int targetLen = target.length;
     int queryLen = query.length;
     for(int i = 0; i + queryLen <= targetLen; i++){ // i++ means i=i+1.
          int j = 0;
          while(j < queryLen && target[i+j] == query[j])  j = j+1;
          if(j == queryLen)  System.out.println(i);
     }
}
```

query	0	1	2	3
	A	T	A	A

target		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i+j	j	A	T	A	A	T	A	C	G	A	T	A	A	T	A	A
0, ..., 3	0, ...,3	A	T	A	A											
1	0		A													
2,3	0,1			A	T											
3, ..., 6	0, ..., 3				A	T	A	A								
4	0					A										
5,6	0,1						A	T								
6	0							A								
7	0								A							
8, ..., 11	0, ..., 3									A	T	A	A			
9	0										A					
10,11	0,1											A	T			
11, ..., 14	0, ..., 3												A	T	A	A

Fig. 1.2 The upper half shows a brute-force string search algorithm for scanning the target string for the query. The lower half presents execution of the brute-force string search algorithm on the query and `target` arrays.

pair of characters disagrees, or all the pairs are equal. In the latter case, `j` is incremented until it reaches `queryLen`. Subsequently, the program exits the while-loop and prints the position `i` at which the query occurs in the target if `j` equals `queryLen`. "`System.out.println(i)`" prints the value of `i` to the standard output.

To illustrate how the brute-force search program works, consider the target string `ATAATACGATAATAA` and the query `ATAA` stored in the arrays `target` and `query` in Figure 1.2. The lower half of Figure 1.2 illustrates the execution of the program. The leftmost two columns present the ranges that `i + j` and `j` take until the program exits the while-loop. Letters in gray boxes indicate positions at which the target and query disagree. For example, consider the case when `i` ranges from 3 to 6 and `j` takes values from 0 to 3. The 6th character, `C`, in the target and the 3rd character, `A`, in the query differ, as indicated by the gray box.

1.3 Encoding Strings into Integers

The brute-force string search iterates the step of making pairwise comparisons between individual letters at the same positions in the query string and each substring of the same length in the target sequence. Since the maximum number of character comparisons in each step is determined by the length of the query, handling a fairly long query may increase the overall computation time. Here, we present an idea for improving this basic string comparison step by transforming strings into integer representations, thereby replacing the string comparison with one operation that compares two integers.

We attempt to encode a string into an integer so that the integer can be decoded to the original string, thereby making it possible to compare integer representations instead of strings themselves. Since this book mainly considers strings that are comprised of four nucleotide letters A, C, G, and T, string $b_0 b_1 \ldots b_{k-1} (b_i \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\})$ of length k is called a *k-mer* or a *k-mer string* in what follows. We inductively define a function that maps a k-mer to an integer called a *k-mer integer*. The basic step is to define *encode* for 1-mer that consists of one nucleotide letter. Since four integers are sufficient to provide unique representations for the four individual nucleotide letters, let us associate nucleotide letters with integers in alphabetical order:

$$encode(\mathtt{A}) = 0, encode(\mathtt{C}) = 1, encode(\mathtt{G}) = 2, encode(\mathtt{T}) = 3$$

In the inductive step, we extend the definition of *encode* to k-mer $s = b_0 b_1 \ldots b_{k-1} (b_i \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\})$ according to the following formula:

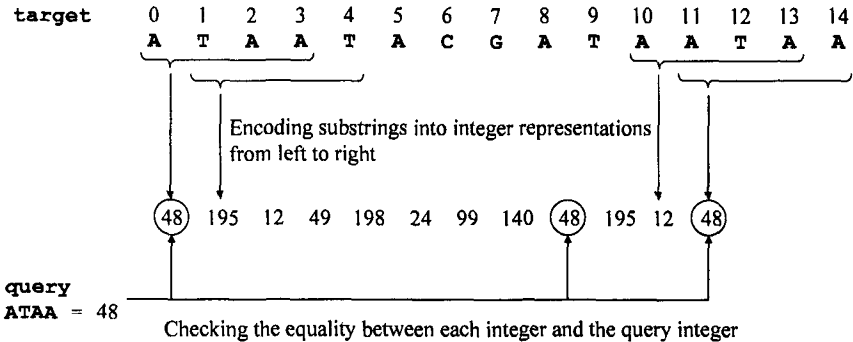
$$encode(s) = \sum_{i=0}^{k-1} 4^{k-1-i} encode(b_i).$$

$encode(s)$ is the k-mer integer. For example, we have

$$encode(\mathtt{ATAA}) = 0 \cdot 4^3 + 3 \cdot 4^2 + 0 \cdot 4^1 + 0 \cdot 4^0 = 48.$$

A k-mer is transformed into $2k$ bits, and its integer values range from 0 to $2^{2k} - 1$. If integers are coded in unsigned 32 bits, this transformation allows us to represent strings of at most 16 nucleotide letters.

Decoding a k-mer integer into the original string obviously involves iterating the step that divides a given integer (or its running quotient) by 4 and prints the letter corresponding to the reminder. The step is iterated




```
public static void intBruteSearch( int[] target, int[] query ) {
     // Exit if the target is shorter than the query.
     if(target.length < query.length) return;
     // Generate k-mer integer representations of the target and query.
     int[] intTarget = generateIntTarget(target, query.length);
     int intQuery = generateIntQuery(query);
     // Search intTarget for the query.
     for(int i = 0; i < intTarget.length; i++)
          if(intTarget[i] == intQuery) System.out.print(i+" ");
     System.out.println();
}
public static int[] generateIntTarget( int[] target, int queryLen ){
     // Exit if the target is shorter than the query.
     if(target.length < queryLen) return null;
     int intTargetLen = target.length - queryLen + 1;
     int[] intTarget = new int[intTargetLen];
     // Generate intTarget array.
     int tmp = 0;  int encode = 1;
     for(int i = 0; i < intTargetLen; i++){
          if(i == 0)
               for(int j=0; j<queryLen; j++){ // Initialize variables.
                    tmp = 4*tmp + target[j];  encode = encode*4;  }
          else // Compute the next value of tmp from the previous value.
               tmp = 4*tmp- encode*target[i-1] + target[i+queryLen-1];
          intTarget[i] = tmp;
     }
     return intTarget;
}
public static int generateIntQuery(int[] query){
     int intQuery = 0;
     for(int i = 0; i < query.length; i++) intQuery = 4*intQuery + query[i];
     return intQuery;
}
```

Fig. 1.3 The upper half illustrates how the program in the lower half operates on the target string when the query string is given.

k-times for a k-mer integer. For example, if 6 is a 3-mer integer, we perform the following steps to decode 6 into ACG:

Step	Quotient	Reminder	Print
1	$6/4 = 1$	$6 \bmod 4 = 2$	G
2	$1/4 = 0$	$1 \bmod 4 = 1$	C
3	$0/4 = 0$	$0 \bmod 4 = 0$	A

One k-mer integer can be decoded into different strings if k is changed. For example, we can decode the k-mer integer 1 into C, AC, and AAC respectively for $k = 1, 2, 3$. To avoid ambiguity, if the value of k is determined, it should be stated explicitly. However, in the general context in which the value of k should remain open, we will use k-mer integers to express this notion.

The upper half of Figure 1.3 illustrates the procedure used to encode all 4-mer substrings in ATAATACGATAATAA into 4-mer integers. Individual 4-mer integers are generated sequentially and are checked to see if they equal the 4-mer integer of the query ATAA, which is 48. Consider the computation of 195, the 4-mer integer of TAAT. Observe

$$\begin{aligned} encode(\texttt{ATAA}) &= 0 \cdot 4^3 + 3 \cdot 4^2 + 0 \cdot 4^1 + 0 \cdot 4^0 &= 48 \\ encode(\texttt{TAAT}) &= 3 \cdot 4^3 + 0 \cdot 4^2 + 0 \cdot 4^1 + 3 \cdot 4^0 &= 195 \end{aligned}$$

It is evident that the 4-mer integer of TAAT can be calculated from the value of the previous substring ATAA, i.e.,

$$encode(\texttt{TAAT}) = (encode(\texttt{ATAA}) - 0 \cdot 4^3) \cdot 4 + 3 \cdot 4^0.$$

Note that the above formula uses only three arithmetic operations: subtraction, multiplication, and addition, if 4^3 has been computed once previously. In general, for string $s_k = b_k b_{k+1} \ldots b_{k+l-1}$ and $s_{k+1} = b_{k+1} b_{k+2} \ldots b_{k+l}$, we have

$$encode(s_{k+1}) = (encode(s_k) - encode(b_k) \cdot 4^{l-1}) \cdot 4 + encode(b_{k+l}).$$

Therefore, even if longer substrings are processed, it holds that three arithmetic operations are required before moving on to handle the next substring, which is more efficient than scanning both the query string and each substring in the target. The lower half of Figure 1.3 presents a program that implements the brute-force string search algorithm empowered with k-mer integers of substrings.

1.4 Sorting k-mer Integers and a Binary Search

To search the target string for one query string, scanning `target` from the head to the tail for each query is not a demanding task. However, if a large number of queries has to be processed and the target string is extremely long, like the human genome sequence, we need to accelerate the overall performance. Here, we introduce a technique for preprocessing the target sequence, so that the task of searching for one query string can be done more efficiently at the expense of using more main memory.

The upper half of Figure 1.4 shows a table in which the 4-mer integer for the substring starting at the `i`-th position is put into `intTarget[i]`. The size of `intTarget` is `targetLen - queryLen + 1`, which is denoted by `intTargetLen` in what follows. Occurrences of `ATAA` in the target string can be found by searching `intTarget` for the 4-mer integer of `ATAA`, 48. However, putting all the 4-mer integers into an array does not improve the performance.

Our key idea is to sort the values of `intTarget` in ascending order, as illustrated in the lower half of Figure 1.4. Basic efficient algorithms for sorting a list of values will be introduced in Chapter 2. Simple application of the sorting process may lose information regarding the location of each element `intTarget`. In order to memorize the position (index) of each element, we use another array named `posTarget`. For example, suppose that after `intTarget` has been sorted, `intTarget[i]` is assigned to the `j`-th position of `intTarget*`, where `intTarget*` denotes the result of sorted `intTarget`. Then, we simultaneously assign `i` to `posTarget[j]`.

In a sorted list of values, a query value can be located quickly using a binary search. A binary search probes the value in the middle of the list, compares the probed value and the query to decide whether the query is in the lower or upper half of the list, and moves to the appropriate half to search for the query. For example, let us search the sorted `intTarget*` for 24. The binary search calculates a middle position by dividing 0+11 by 2 and obtains the quotient 5. The search then probes the 5-th position and finds that its value, 48, is greater than 24. Therefore, it moves to the lower half and probes the 2nd position ($(0+5)/2 = 2$) to identify 24. In order to memorize the 5th position, at which 24 is located in the original `intTarget` array, `posTarget[2]` stores the position 5.

When the binary search processes a list of n elements, each probing step narrows the search space to its lower or upper half. After k probing steps, the binary search focuses on approximately $n/2^k$ elements, and it stops

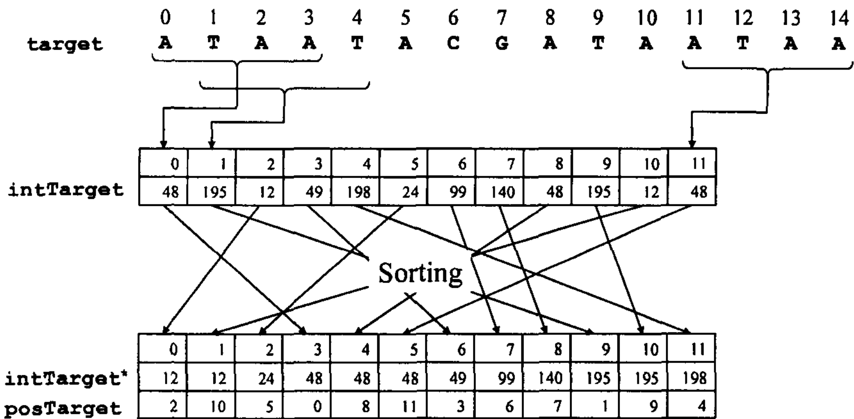


Fig. 1.4 The 4-mer substrings in the top array **target** are transformed into 4-mer integers in the middle array **intTarget**. The elements in **intTarget** are sorted in ascending order and are put into **intTarget***. **posTarget** memorizes the original positions of the elements in **intTarget** before they are sorted.

probing when two elements remain to be investigated. Solving $n/2^k = 2$ shows that k is almost equal to $\log_2 n - 1$, which indicates the approximate number of probing steps that the binary search needs to take. For instance, when $n = 2^{30} \approx 10^9$, $k \approx 29$.

1.5 Binary Search for the Boundaries of Blocks

The binary search described above works properly when the sorted list contains no duplicates. However, the sorted `intTarget*` in Figure 1.4 contains multiple occurrences of the same numbers, and the program is required to output all the positions of occurrences of the query. For example, 48 appears from positions 3 to 5 successively in the sorted `intTarget*`, and it occurs at positions 0, 8, and 11 in the original array, as `posTarget` indicates. Since occurrences of the query 4-mer integer are successive in the sorted list, it is sufficient to calculate the leftmost and rightmost positions of the successive occurrences, and each of the two boundary positions can be calculated in the binary search program given in Figure 1.5.

In the binary search for the leftmost position, the program uses the variable `left` to approach the leftmost index by scanning the array from the left, as illustrated in Figure 1.5. In order to implement this idea,

```
public static void binarySearch( int[] target,  int[] query ){
    // Exit if the target is shorter than the query.
    if(target.length < query.length) return;
    // Generate k-mer integer representations of target and query.
    int[] intTarget = generateIntTarget(target, query.length);
    int intQuery = generateIntQuery(query);
    // Sort elements in intTarget and
    // memorize the original positions in posTarget.
    int targetLen = intTarget.length;
    int[] posTarget = new int[targetLen];
    for(int i=0; i<targetLen; i++) posTarget[i]=i;
    randomQuickSort(intTarget, posTarget, 0, targetLen-1);
    // Search for the left boundary.
    int left, right, middle;
    for(left=0, right=targetLen; left < right; ){
        middle = (left + right) / 2;
        if( intTarget[middle] < intQuery) left = middle + 1;
        else right = middle; }
    int leftBoundary = left;
    // Search for the right boundary.
    // We use "right" to approach the index next to the right boundary.
    for(left=0, right=targetLen; left < right; ){
        middle = (left + right) / 2;
        if( intTarget[middle] <= intQuery ) left = middle + 1;
        else right = middle; }
    // Print positions in the range between the two boundaries.
    for(int i = leftBoundary; i < right; i++)
        System.out.print(posTarget[i]+" ");
    System.out.println();
}
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
intTarget*	12	12	24	48	48	48	49	99	140	195	195	198	
leftmost	l_0		l_1	l_2									
				r_2			r_1						r_0
rightmost	l_0				l_1		l_2						
							r_1						r_0

Fig. 1.5 The upper half presents a binary search algorithm for seeking the leftmost and rightmost boundary positions of contiguous occurrences of the query k-mer integer. The lower half shows the operation of the program on the sorted `intTarget*` for the query 4-mer integer, 48.

the program sets `left` to the position next to the middle position when the query 4-mer integer is higher than the target 4-mer interger in the middle. In the leftmost and rightmost search rows, l_0 and r_0 show the initial positions of `left` and `right`, respectively, while l_i and r_i indicate

the i-th updated values during the execution. When the program exits the for-loop, observe that `left` contains the leftmost index where the query 4-mer integer appears in the leftmost search of Figure 1.5. In contrast, when searching for the rightmost location, `right` is intended to store the position next to the rightmost position in the final step.