Chapter 16
Strings and Sequences

16.1 C Strings

In this chapter, we will look at the **string** data type, which can be used to represent text data using sequences of characters. Strings are one of the most important components of programming, and much of the technology that we use today relies on strings and string operations. In fact, for many developers, a string object (`"Hello, World!"`) was the thing that welcomed them to the programming world. Since strings are so ubiquitous and useful, the course would not be complete without a discussion on how strings and string algorithms work.

In C++, there exist two primary abstractions for string objects, **C strings** and **C++ strings** (`std::string`). For the majority of this class, we will be working with C++ strings. However, this section will provide a brief overview of C strings, which serve as a foundation for the more complex string objects that we will cover later.

A C string is an array of characters that is terminated by a null-character, or **sentinel** character (`'\0'`). The sentinel is important because it allows the program to determine where a C string ends in memory, and therefore it must be included at the end of every C string.

A C string can be declared in many ways. The following methods can all be used to declare a C string that stores the word "EECS". Note that, in the first two methods, the language is smart enough to add a null terminator to the end of the string for you. However, if you explicitly specify the size of the character array like in methods three and four, you will need to explicitly assign the null character to the last index.

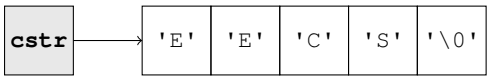
```
const char* cstr = "EECS";  // read-only
```

```
char cstr[] = "EECS";
```

```
char cstr[5] = {'E', 'E', 'C', 'S', '\0'}
```

```
char* cstr = new char[5];
cstr[0] = 'E'; cstr[1] = 'E'; cstr[2] = 'C'; cstr[3] = 'S'; cstr[4] = '\0';
```

In memory, `cstr` is a pointer to a character array that stores the characters of the string "EECS":

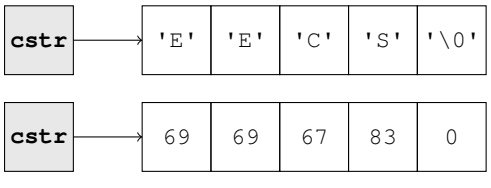


Behind the scenes, each `char` is actually stored as a numerical value. This is because C strings use ASCII, a standard that dictates what numerical value each character is assigned to. These values are shown in the table below:

32	Space	**48**	0	**64**	@	**80**	P	**96**	`	**112**	p
33	!	**49**	1	**65**	A	**81**	Q	**97**	a	**113**	q
34	"	**50**	2	**66**	B	**82**	R	**98**	b	**114**	r
35	#	**51**	3	**67**	C	**83**	S	**99**	c	**115**	s
36	$	**52**	4	**68**	D	**84**	T	**100**	d	**116**	t
37	%	**53**	5	**69**	E	**85**	U	**101**	e	**117**	u
38	&	**54**	6	**70**	F	**86**	V	**102**	f	**118**	v
39	'	**55**	7	**71**	G	**87**	W	**103**	g	**119**	w
40	(	**56**	8	**72**	H	**88**	X	**104**	h	**120**	x
41	)	**57**	9	**73**	I	**89**	Y	**105**	i	**121**	y
42	*	**58**	:	**74**	J	**90**	Z	**106**	j	**122**	z
43	+	**59**	;	**75**	K	**91**	[	**107**	k	**123**	{
44	,	**60**	<	**76**	L	**92**	\	**108**	l	**124**	|
45	-	**61**	=	**77**	M	**93**	]	**109**	m	**125**	}
46	.	**62**	>	**78**	N	**94**	^	**110**	n	**126**	~
47	/	**63**	?	**79**	O	**95**	_	**111**	o	**127**	DEL

ASCII characters 0-31 have been omitted from the table because they are unprintable characters. However, some notable characters in this range are the null character `'\0'` (ASCII value 0), the tab character `'\t'` (ASCII value 9), the newline character `'\n'` (ASCII value 10), and the carriage return character `'\r'` (ASCII value 13).

Using the table, we can convert the characters of our C string to their respective ASCII values. This is what the array above looks like under the hood:



Because characters are actually numbers behind the scenes, we can compare characters using comparison operators and iterate through letters in a loop, as follows:

```
for (char current = 'a'; current <= 'z'; ++current) {
  std::cout << current;  // prints "abcdefghijklmnopqrstuvwxyz"
} // for current
```

Because C strings are arrays of characters, they behave just like arrays. Thus, if you try to access the value of a C string variable, you will end up getting a pointer to the first character. As a result, you cannot compare two C strings using built-in operators like `operator==`; special operations are required to work with C strings. Some of these operations are detailed below, and they can be used if you `#include <cstring>` in your code:

`size_t strlen(const char* str);`
Returns the length of the C string `str`.

```
char str[] = "apple";
std::cout << strlen(str) << '\n';    // prints 5
```

`char* strcpy(char* dest, const char* source);`
Copies the C string pointed to by `source` into the array pointed to by `dest`, including the sentinel character. The size of the array pointed to by `dest` needs to be long enough to hold the contents of `source`, and it should not overlap with `source` in memory. After the copy, `dest` is returned.

```
char s1[] = "hi";
char s2[3];           // allocate enough space
strcpy(s2, s1);       // s2 also stores {'h','i','\0'} since s1 was copied over
```

char* strncpy(**char*** dest, **const char*** source, **size_t** num);
Copies the first num characters of source to dest. If the entirety of source is copied before num characters have been copied over (i.e., if source has fewer than num characters), dest is padded with zeros until a total of num characters have been written. If source is longer than num characters, only the first num characters are copied over and *not* the sentinel character! After the copy, dest is returned.

```
char s1[] = "eecs281";
char s2[5];
strncpy(s2, s1, 4);  // s2 now stores "eecs" but with no sentinel
s2[4] = '\0';        // sentinel must be added for s2 to be a valid C string
```

char* strcat(**char*** dest, **const char*** source);
Concatenates a copy of the source C string to the end of the dest C string. The sentinel of dest is overwritten by the first character of source, but a new sentinel is added to the end of the resultant C string after concatenation. After the concatenation, dest is returned.

```
char s[8];
strcpy(s, "eecs");   // copies {'e','e','c','s','\0'} to s
strcat(s, "281");    // appends {'2','8','1','\0'}, s now stores {'e','e','c','s','2','8','1','\0'}
```

char* strncat(**char*** dest, **const char*** source, **size_t** num);
Concatenates the first num characters of source to dest, plus a sentinel. If source has fewer than num characters, only the characters up to the sentinel is copied. After the concatenation, dest is returned.

```
char s1[] = "281282283";
char s2[] = "eecs";
strncat(s2, s1, 3);       // s2 now stores {'e','e','c','s','2','8','1','\0'}
```

int strcmp(**const char*** str1, **const char*** str2);
Compares the two C strings str1 and str2. If the two C strings are equal, the function returns 0. If str1 is smaller than str2 (using ASCII values to determine ordering), a negative number is returned. If str1 is larger than str2, a positive number is returned. To determine which C string comes first, the function compares the characters of the strings one-by-one until a mismatch (or sentinel) is encountered. The ASCII values of the mismatched characters are then compared to determine which C string comes first.

```
char s1[] = "apple";
char s2[] = "apple";
char s3[] = "banana";
if (strcmp(s1, s2) == 0) {
  std::cout << "This prints if s1 == s2" << '\n';
} // if
if (strcmp(s1, s3) < 0) {
  std::cout << "This prints if s1 < s3" << '\n';
} // if
```

int strncmp(**const char*** str1, **const char*** str2, **size_t** num);
Compares up to num characters of the two C strings to determine equality. The return value rules for this function are the same as the rules fo strcmp() (i.e., 0 if equal, etc.).

```
char s1[] = "apple";
char s2[] = "application";
if (strncmp(s1, s2, 4) == 0) {
  std::cout << "This prints because first four chars are equal" << '\n';
} // if
if (strncmp(s1, s2, 5) < 0) {
  std::cout << "This prints because apple < appli" << '\n';
} // if
```

This is not a comprehensive list of all the functions that are provided by the <cstring> library. Additional C string operations can be found by looking through documentation. However, we will not be going over these additional operations because C strings will not be a focus of this class. Instead, you will primarily be working on C++ strings, which will are covered in the next section.

Lastly, it should be noted that C strings exhibit special behavior when used with C++ output streams like std::cout. If std::cout is passed in a char*, it will treat the pointer as a C string and print out every character up until the first null character ('\0') it encounters. If a char* that is not null-terminated is passed into std::cout, you would get undefined behavior.

```
char str1[] = {'e', 'e', 'c', 's', '\0'};
char str2[] = {'e', 'e', 'c', 's'};

std::cout << str1 << '\n';  // prints "eecs"
std::cout << str2 << '\n';  // undefined behavior
```

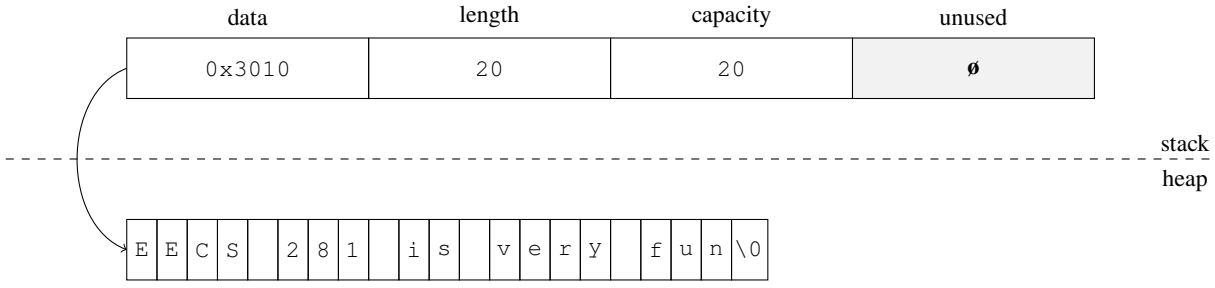
16.2 C++ Strings

※ 16.2.1 *String Implementations and Short String Optimization* (✻)

When you work with C strings, you are directly working with the character values in memory. On the other hand, a C++ string, or `std::string`, is a class-type object that encapsulates a string's underlying character array and manages its memory for you. The `std::string` is defined in the `<string>` library. You can think of a `std::string` as similar in concept to a `std::vector<>` — like a vector, the string class stores an internal array with its underlying data and provides member functions that can be used to operate on that data.

To gain insight into how a `std::string` works under the hood, let's look at a common implementation of the C++ string class in memory.[1] In g++ version 5 and above, a `std::string` uses 32 bytes of memory on the stack to store the following information: (1) a pointer to the beginning of the underlying character array, (2) the length of the string, and (3) the capacity of the underlying character array (not including the sentinel). An illustration is shown below:

```
std::string str = "EECS 281 is very fun";
```



The pointer, size, and capacity values each take up 8 bytes of memory. This adds up to a total of 24 bytes. So, why are there 8 bytes of unused stack space floating around in each string? It turns out that the `capacity` member variable actually does double duty. If we were to look at what a C++ string stores internally, we would essentially see something like this (this is a simplified version of the actual implementation):

```
1  class string {
2    char* data;
3    size_t length;
4    union {
5      size_t capacity;
6      char local_buffer[16];
7    };
8  };
```

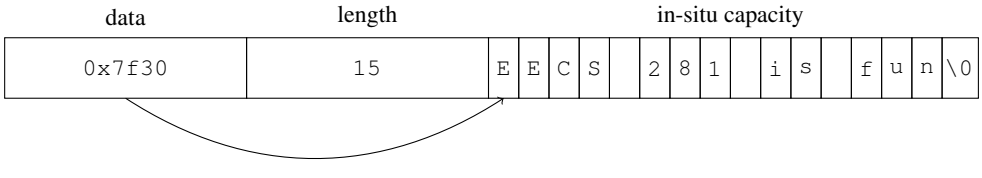
You do not need to know this for this class, but a `union` is a special type of object that can only take on one of its data members at any time. In this case, the `string` object holds three member variables: a data pointer, the string's length, and a special variable that can store one of either:

- a 8 byte `size_t` value that stores the capacity of the heap-allocated array *OR*
- a 16 byte character array

This additional memory allows the `std::string` to take advantage of something known as **short string optimization** (SSO). As mentioned in chapter 6, the program stack size is rather limited compared to the heap, and arrays on the stack need to have fixed sizes at compile time. Thus, C++ strings typically initialize their data on the heap (and keep stack usage capped at 32 bytes) to avoid potential overflow issues. However, there is a trade-off: dynamically-allocated arrays tend to have slower performance compared to arrays on the stack.

For larger strings, there is no way around this. The majority of strings, however, tend to be short, and it would be inefficient to dynamically allocate them every time. This is where the additional stack memory comes in handy. If the length of a `std::string` is fewer than 16 characters, the memory that would have been used to store the capacity value (along with the extra 8 bytes of memory at the end) is used to store the actual string instead.

```
std::string str = "EECS 281 is fun";
```



By repurposing the last 16 bytes of stack space to hold short strings, the `std::string` is able to take advantage of performance benefits from arrays that live on the stack. Only when the length of the string exceeds 15 characters does the string begin allocating memory on the heap. Note that a string of length 16 does not fit on the stack, since an additional byte is needed to store the sentinel character (and thus requires 17 bytes in total).

[1] Like with a vector, a string's standard library implementation may be platform-specific. In this chapter, we will be looking at the GCC string implementation in libstdc++, which is commonly used by many platforms.

※ 16.2.2 *String Library Operations*

Objects of the `std::string` class support assignment using `operator=` and comparisons using operators <, >, and ==. Strings also support random access iterators, which can be initialized using variations of `.begin()` and `.end()` (similar to other random access containers).

The `<string>` library provides many operations that can be used to operate on a `std::string` object. An outline of several string member functions is provided below. Several of these operations have different variations depending on the parameters that are passed in, but this list will only cover the most common ones (you are encouraged to look at other documentation for these additional behaviors).

```
size_t std::string::size();
size_t std::string::length();
```
Returns the number of characters in the string, not including the sentinel. Both functions do the same thing.

```
std::string str = "EECS281";
std::cout << str.size() << '\n';     // prints 7
std::cout << str.length() << '\n';   // prints 7
```

```
void std::string::clear();
```
Clears the contents of a string.

```
bool std::string::empty();
```
Returns whether a string is empty.

```
std::string str = "EECS281";
str.clear();
std::cout << std::boolalpha << str.empty() << '\n';   // prints "true"
```

```
char& std::string::operator[](size_t idx);
```
Returns a reference to the character at index `idx` of the string.

```
std::string str = "EECS281";
std::cout << str[2] << '\n';   // prints "C"
```

```
std::string& std::string::operator+=(const std::string& str);
std::string& std::string::operator+=(const char* cstr);
std::string& std::string::operator+=(char c);
```
Appends the given value to the back of the string and returns the modified string.

```
std::string str = "hello ";
str += "world";
std::cout << str << '\n';      // prints "hello world"
```

```
void std::string::push_back(char c);
```
Appends the character `c` to the back of the string and increases length by one.

```
void std::string::pop_back();
```
Deletes the character at back of string and reduces length by one. This operation causes undefined behavior if used on an empty string.

```
std::string str = "EECS28";
str.push_back('1');            // str now stores "EECS281"
str.pop_back();                // str now stores "EECS28"
```

```
char& std::string::front();
```
Returns a reference to the first character of the string. Causes undefined behavior if used on an empty string.

```
char& std::string::back();
```
Returns a reference to the last character of the string. Causes undefined behavior if used on an empty string.

```
std::string str = "EECS281";
std::cout << str.front() << '\n';    // prints the first character, or 'E"
std::cout << str.back() << '\n';     // prints the last character, or '1'
```

```
std::string& std::string::insert(size_t pos, const std::string& str);
```
Inserts a copy of `str` directly before the character at position `pos` and returns the modified string.

```
std::string str = "281";
str.insert(0, "EECS");
std::cout << str << '\n';    // prints "EECS281"

std::string str1 = "pot";
std::string str2 = "arr";
std::string str3 = str1.insert(1, str2);
std::cout << str3 << '\n';   // prints "parrot"
```

`std::string& std::string::erase(size_t pos = 0, size_t len = std::string::npos);`
Erases the portion of the string that starts at index `pos` and spans `len` characters, or until the end of the string, whichever comes first. If not specified, `pos` defaults to 0 and `len` defaults to `npos`. Returns a reference to the modified string.

`iterator std::string::erase(iterator pos);`
Erases the character pointed to by `pos`. Returns an iterator pointing to the character directly following the character erased, or the `end` iterator if no such character exists.

`iterator std::string::erase(iterator first, iterator last);`
Erases all characters in the range `[first, last)`. Returns an iterator referencing the character `last` pointed to before the erase, or the `end` iterator if no such character exists.

```
std::string str1 = "carrot";
str1.erase(2, 3);
std::cout << str1 << '\n';     // prints "cat"

std::string str2 = "carrot";
str2.erase(3);                 // erases everything starting from index 3
std::cout << str2 << '\n';     // prints "car"

std::string str3 = "carrot";
str3.erase(str3.begin(), str3.begin() + 3);
std::cout << str3 << '\n';     // prints "rot"
```

`std::string& std::string::replace(size_t pos, size_t len, const std::string& str);`
Replaces the portion of the string that begins at index `pos` and spans `len` characters with the contents of `str` and returns a reference to the modified string.

`std::string& std::string::replace(iterator i1, iterator i2, const std::string& str);`
Replaces the portion of the string that falls in iterator range `[i1, i2)` with the contents of `str` and returns the modified string.

```
std::string str = "eecs 370 is fun";
str.replace(5, 3, "281");      // replaces 3 chars starting at index 5
std::cout << str << '\n';      // prints "eecs 281 is fun"
```

`const char* std::string::c_str();`
Returns a pointer to the underlying C string that stores the current data of the string object.

```
std::string str = "eecs 281";
const char* cstr = str.c_str();
std::cout << cstr << '\n';       // prints "eecs 281"
```

`size_t std::string::find(const std::string& str, size_t pos = 0);`
Searches the string starting at index `pos` for the first occurrence of `str` and returns the index position of the first match (or `npos` if no matches were found). `pos` defaults to 0 if not specified.

```
std::string str = "pineapple";
size_t idx1 = str.find("apple", 0);   // idx1 == 4
size_t idx2 = str.find("orange");     // idx2 == npos
size_t idx3 = str.find("apple", 6);   // idx3 == npos
```

`size_t std::string::find_first_of(const std::string& str, size_t pos = 0);`
Searches the string starting at index `pos` for the first character that matches any of the characters in `str` and returns the position of the first match (or `npos` if no matches were found). `pos` defaults to 0 if not specified.

```
std::string str = "pineapple";
size_t idx1 = str.find_first_of("apple");    // idx1 == 0 ('p')
size_t idx2 = str.find_first_of("orange");   // idx2 == 2 ('n')
size_t idx3 = str.find_first_of("pie", 4);   // idx3 == 5 ('p')
```

`size_t std::string::find_last_of(const std::string& str, size_t pos = std::string::npos);`
Searches the string for the last character that matches any of the characters in `str` and returns position of first match (or `npos` if no matches were found). Only characters at or before index `pos` are considered. `pos` defaults to 0 if not specified.

```
std::string str = "pineapple";
size_t idx1 = str.find_last_of("apple");     // idx1 == 8 ('e')
size_t idx2 = str.find_last_of("orang");     // idx2 == 4 ('a')
size_t idx3 = str.find_last_of("pie", 4);    // idx3 == 5 ('p')
```

`size_t std::string::find_first_not_of(const std::string& str, size_t pos = 0);`
Searches the string starting at index `pos` for the first character that does not match any of the characters in `str` and returns its position (or `npos` if no matches were found). `pos` defaults to 0 if not specified.

`size_t std::string::find_last_not_of(const std::string& str, size_t pos = std::string::npos);`
Searches the string for the last character that does not match any of the characters in `str` and returns its position (or `npos` if no matches were found). Only characters at or before index `pos` are considered. `pos` defaults to `npos` if not specified.

`std::string std::string::substr(size_t pos = 0, size_t len = std::string::npos);`
Returns a newly constructed string that is a substring of the original string. The substring begins at index `pos` and spans `len` characters (or until the end of the string, whichever comes first).

```
std::string str = "pineapple";
std::string substring = str.substr(4, 3);   // start at index 4 with length 3
std::cout << substring << '\n';             // prints "app"
```

`std::string operator+(const std::string& lhs, const std::string& rhs);`
Can be used to concatenate two strings (this is not a member function of the string class, but it is useful to know).

```
std::string str1 = "pine";
std::string str2 = "apple";
std::string str3 = str1 + str2;
std::cout << str3 << '\n';       // prints "pineapple"
```

This is not a comprehensive list of all available string operations, but these are the most common. You may have noticed a value known as `std::string::npos` in the function definitions above. The value `npos` is a special constant that is used when working with strings. This variable stores the value of the largest possible `size_t`, and its purpose depends on how it is used. When used as the value for the `len` or `pos` parameter of a string member function, it tells the function to complete an operation until the end of the string. If `npos` is returned by a variant of the `find()` function, it indicates that no match was found.

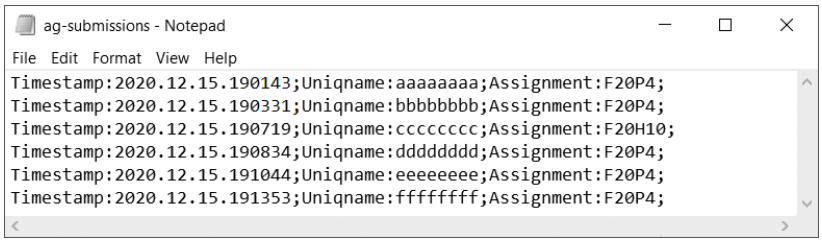
※ 16.2.3 *std::string_view* (✻)

One major pitfall of strings is that they can be quite expensive to copy, especially if dynamic memory allocation is involved (recall that heap allocation is not a cheap process). However, there are cases where an algorithm may only need a *read-only* copy of the string it is working with. In these situations, there is no need to give a separate copy of a string to every method that needs to use it. Instead, since usage of the string is intended to be read-only, we can simply pass in a "view" of the exact same string object to all the methods that need it.

This behavior was simplified in C++17 with the introduction of `std::string_view`. When you give something a `std::string_view`, you are giving it *a view of an existing string that is owned by something else*, instead of making a separate copy of that string. Not only are string views more performant, they are more lightweight in size as well. If you look at what a string view stores internally, you would see something like this (again, this is an oversimplification, but it provides an insight into what this type provides):

```
class string_view {
  size_t len;         // length of the string view
  const char* str;    // pointer to the first character
};
```

If used correctly, string views can greatly improve the performance of many string operations, from substring searches to string comparisons and concatenation. Let's look at an example where usage of a `std::string_view` may be prudent. Suppose you have a giant file containing autograder submission data that you want to process:

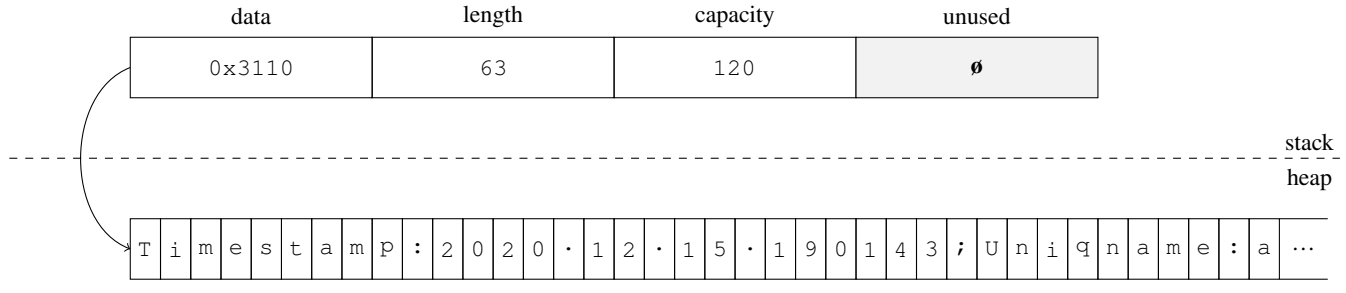



```
void process_data(const std::string& timestamp, const std::string& uniqname,
                  const std::string& assignment) {
  // do stuff...
} // process_data()

int main() {
  std::string line;
  while (std::getline(std::cin, line)) {
    size_t last = 0, next = 0;

    // get timestamp
    next = line.find(";", last);
    std::string timestamp_str = line.substr(last, next - last);
    std::string timestamp = timestamp_str.substr(timestamp_str.find(":") + 1);
    last = next + 1;

    // get uniqname
    next = line.find(";", last);
    std::string uniqname_str = line.substr(last, next - last);
    std::string uniqname = uniqname_str.substr(uniqname_str.find(":") + 1);
    last = next + 1;

```

```
23        // get assignment
24        next = line.find(";", last);
25        std::string assignment_str = line.substr(last, next - last);
26        std::string assignment = assignment_str.substr(assignment_str.find(":") + 1);
27        last = next + 1;
28
29        process_data(timestamp, uniqname, assignment);
30      } // while
31    } // main()
```

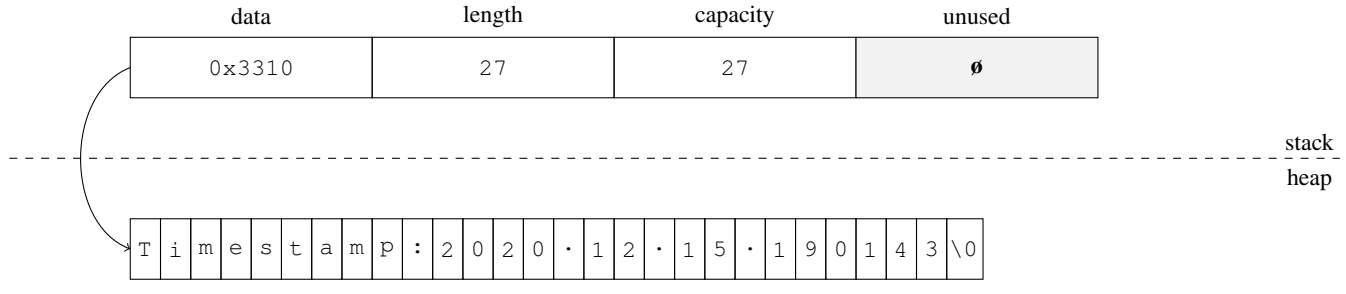
Notice that this code allocates more strings than necessary, which results in a significant performance overhead. First, we read in the contents of a line using `std::getline()` on line 8, which invokes a heap allocation:

```
std::getline(std::cin, line);
```



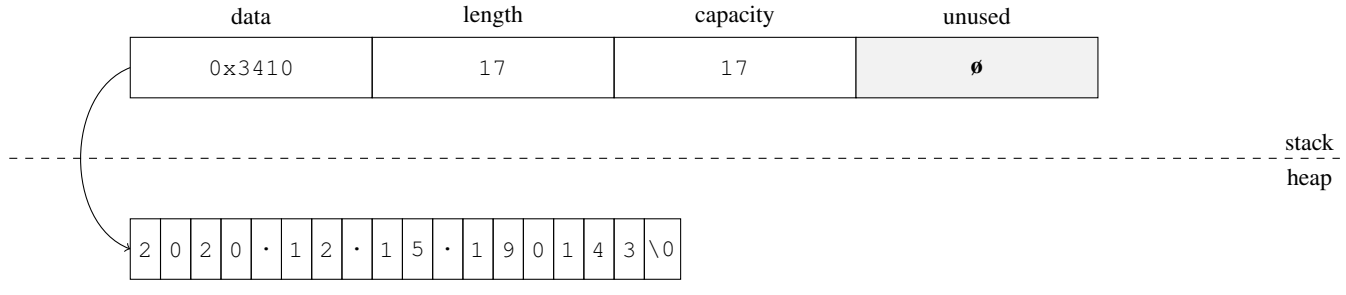
Then, on line 13, we perform another heap allocation to initialize `timestamp_str`:

```
std::string timestamp_str = line.substr(last, next - last);
```



On line 14, we perform yet another heap allocation to initialize the `timestamp` string to be passed into the `process_data()` method:

```
std::string timestamp = timestamp_str.substr(timestamp_str.find(":") + 1);
```

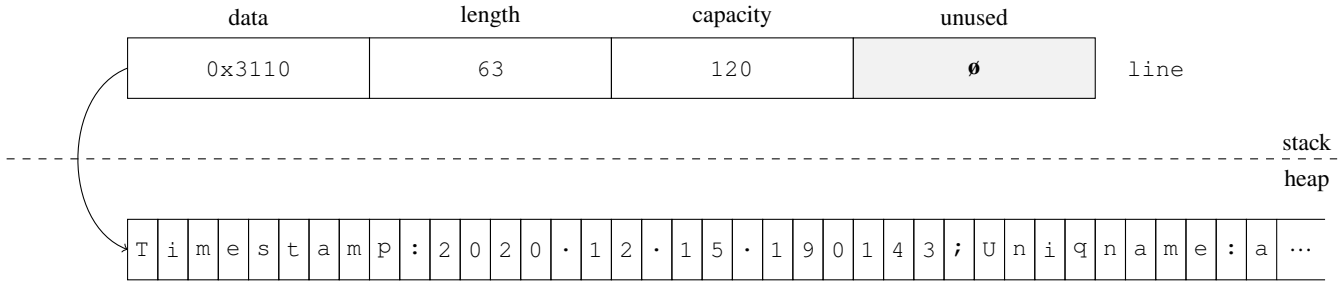


This same inefficiency also applies to the extraction of uniqname and assignment ID from each line. However, since the `process_data()` method does not need to change the contents of any of its string parameters (hence the const reference), we do not need to allocate entirely new strings to pass into the function. Rather, it is sufficient to pass in a view of the original string that we initially read into the variable `line`.

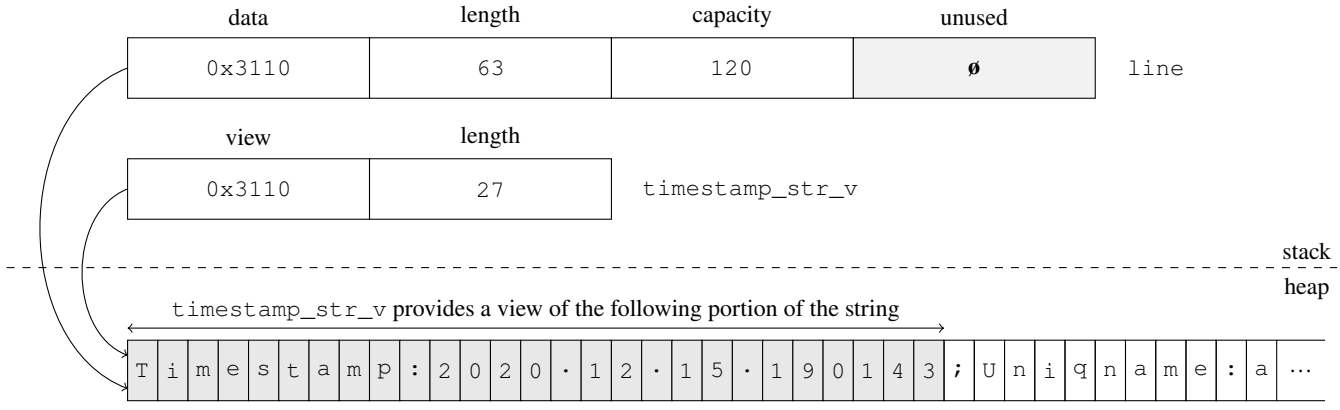
A modified implementation using `std::string_view` is shown below, which does just that:

```
1  void process_data(std::string_view timestamp, std::string_view uniqname,
2                    std::string_view assignment) {
3    // do stuff...
4  } // process_data()
5
6  int main() {
7    std::string line;
8    while (std::getline(std::cin, line)) {
9      size_t last = 0, next = 0;
10
11     // get timestamp
12     next = line.find(";", last);
13     std::string_view timestamp_str_v{line.c_str() + last, next - last};
14     std::string_view timestamp = timestamp_str_v.substr(timestamp_str_v.find(":") + 1);
15     last = next + 1;
16
17     // get uniqname
18     next = line.find(";", last);
19     std::string_view uniqname_str_v{line.c_str() + last, next - last};
20     std::string_view uniqname = uniqname_str_v.substr(uniqname_str_v.find(":") + 1);
21     last = next + 1;
22
23     // get assignment
24     next = line.find(";", last);
25     std::string_view assignment_str_v{line.c_str() + last, next - last};
26     std::string_view assignment = assignment_str_v.substr(assignment_str_v.find(":") + 1);
27     last = next + 1;
28
29     process_data(timestamp, uniqname, assignment);
30   } // while
31 } // main()
```

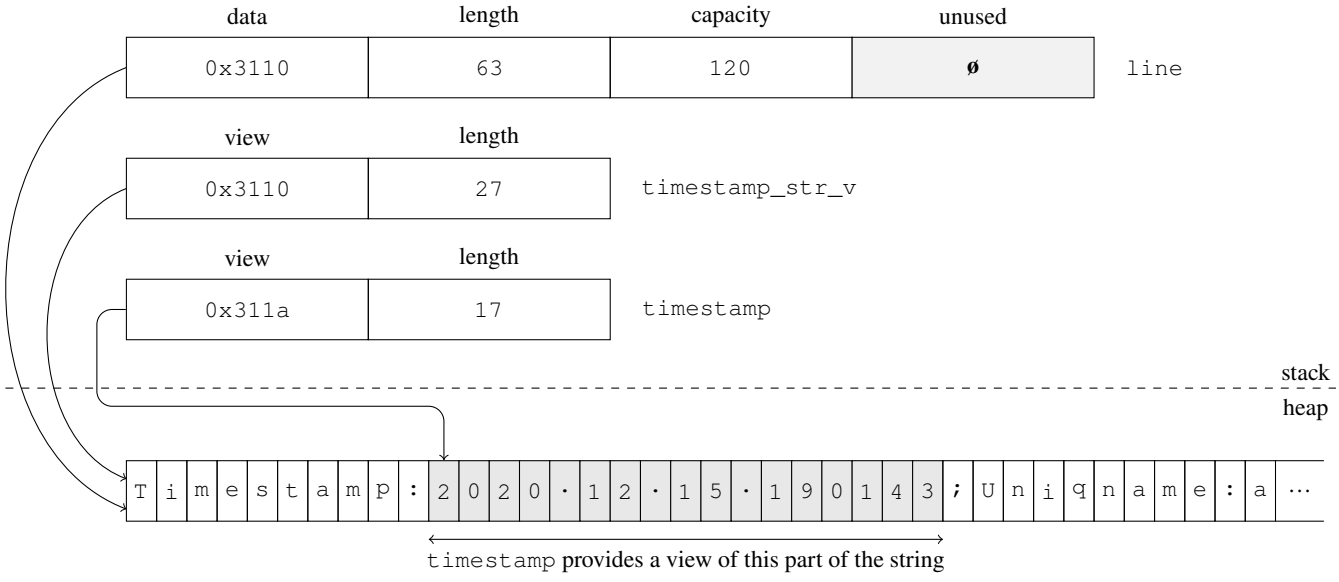
Instead of performing multiple string allocations, this implementation only makes a single allocation for each line of the file. All of the string views that are initialized (on lines 13-14, 19-20, and 25-26) simply refer to the contents of this first allocation of the string instead of making a new copy of their own. This improves the overall performance, since allocating a new string can be an expensive process. To illustrate how this implementation works, consider the value of `line` after the first iteration of the loop:



On line 13, we create a `std::string_view` corresponding to the timestamp portion of the original string. Instead of making a brand new string, this expression simply creates a read-only view of a subsection of the original string.



Likewise, line 14 does the same thing and creates a new view using a substring of the `timestamp_str_v` view.



If you are using C++17 or beyond, you can use `std::string_view` by including the `<string_view>` header. As shown in the previous code, you can create a string view of an existing string by passing in a pointer to the first character along with the size of the view:

```
1  std::string str = "EECS281";
2  std::string_view str_v{str.c_str() + 4, 3};
3  std::cout << str_v << std::endl;  // prints "281"
```

There is a lot of overlap between the methods provided by `std::string_view` and the methods provided by `std::string`. As shown previously, we can use `.substr()` to get a substring of a `std::string_view`, much like how we can use the same method to get a substring of a `std::string`. Other read-only methods, like the find methods, are also supported by string views.

Two additional methods provided by string views are `.remove_prefix()` and `.remove_suffix()`, which can be used to shrink the size of a view. As their names imply, `.remove_prefix()` shrinks a view by moving its start position forward, while `.remove_suffix()` shrinks a view by moving its end position backward. Examples of these methods are shown below:

```
1  std::string str = "xxxEECS281xx";
2  std::string_view str_v = str;
3  str_v.remove_prefix(3);  // trim first three characters of view
4  str_v.remove_suffix(2);  // trim last two characters of view
5  std::cout << str_v << std::endl;  // prints "EECS281"
```

Remark: With the introduction of `std::string_view`, we have an alternative way to express the existence of a read-only string. Now, there are cases where immutable references to strings (e.g., `const std::string&`) may be replaced with a `std::string_view`:

```
1  // before C++17: use immutable reference to std::string
2  std::vector<std::string> strs = { /* values */ };
3  for (const std::string& str : strs) {
4    // do stuff
5  } // for str
6
7  // after C++17: use std::string_view
8  for (std::string_view str_v : strs) {
9    // do stuff
10 } // for str_v
```

There are situations where using a `std::string_view` may be faster than using an immutable `std::string` reference. Consider the following function definition, which takes in a `const std::string&`:

```
void foo(const std::string& str);
```

Now, consider what happens when we pass a string literal into `foo()`:

```
foo("This string is way too long to qualify for SSO");
```

The `foo()` method expects a `std::string`, so the string literal (stored as `const char*`) is first converted into a `std::string` object, which is then destroyed after the function returns. However, this forces a string allocation on the heap, which is not efficient. Instead, we can define the `foo()` method to take in a `std::string_view` instead:

```
void foo(std::string_view str);
```

Here, an object is still created to hold the string literal, but an expensive string allocation is no longer needed since the parameter of `foo()` does not require a `std::string`. Instead, `foo()` simply takes in a pointer to a view of the literal, which is much more efficient.

There are a few things to be aware of when working with string views. First, a string view only provides a read-only view of a string, but it does *not* own any string data on its own. Because of this, you have to make sure that a string view does not outlive its underlying string! Second, a string view is not guaranteed to be null-terminated, as it instead uses its length to identify where its contents end. As a result, string views will not be safe if you need to work with C-style strings and methods that expect null termination.

When passing immutable, read-only strings into functions, a `std::string_view` may be faster, and they are more flexible with C-style string arguments (such as string literals). However, if your function expects null termination or calls some other function that takes in a `std::string` parameter, then passing in a `const std::string&` may be a better option, as `std::string_view` is not guaranteed to be null-terminated, and it would be wasteful to have to convert a view back into a `std::string` if it needs to be in that form anyway.

Remark: In general, `std::string_view` should be passed by value instead of const reference. This is because string views are small, trivially copyable types that can be optimized by compilers if passed by value, and doing so also removes a level of pointer indirection when accessing the contents of the view (i.e., if you pass a string view by reference, the function that is called (or the callee) would need to access the address of the reference first before it can access the address of the view's data itself).

※ 16.2.4 *Additional String Formatting Methods* (✻)

C++20 introduced the `std::format()` method in the `<format>` header, which makes it easier to format strings with custom values. This method takes in a format string that consists of ordinary characters, plus the special characters of `{` and `}` that delineate the position of custom arguments. For instance, the following uses `std::format()` to format a string using custom arguments:

```
// This prints "My favorite class is EECS 281!"
std::string category = "EECS";
std::string number = "281";
std::cout << std::format("My favorite class is {} {}!", category, number) << '\n';
```

You can also specify the order of the arguments in the format string by including an index within the curly braces. This is shown below:

```
// This prints "Have you taken EECS 281? There is no doubt that 281 is the best class in EECS!"
std::string category = "EECS";
std::string number = "281";
std::cout <<
  std::format("Have you taken {0} {1}? There is no doubt that {1} is the best class in {0}!",
              category, number) << '\n';
```

C++23 takes string formatting up a notch with the introduction of `std::print()`, which can be used to format a string and also print it to a customized output stream. We will not go over `std::format()` and `std::print()` in too much detail here, but they are definitely worthwhile to understand if you are using versions of C++ that support them.

16.3 Lexicographical String Comparisons

C++ strings have overloaded comparison operators that allow them to be compared. The ordering of string objects is determined by ASCII value. As a result, uppercase letters are considered to be "less" than lowercase letters, since uppercase letters have lower ASCII values. Otherwise, strings follow lexicographical (dictionary) order, e.g.,

`""< "Zebra"< "a"< "ab"< "b"< "bb"< "bc"< "bc0"`

The C++ `<algorithm>` library provides the `std::lexicographical_compare()` function, which can be used to check if the contents of one iterator range is lexicographically less than the contents of a second iterator range. A lexicographical comparison involves going through the elements in both iterator ranges sequentially until a character in the first range does not compare equal to one in the second. The values of these two elements are then compared, and the result determines which range comes first lexicographically.

```
template <typename InputIterator1, typename InputIterator2, typename OutputIterator, typename Compare>
bool std::lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Returns true if the range `[first1, last1)` compares lexicographically less than the range `[first2, last2)`, and false otherwise. The comparator is optional, and `operator<` is used if it is not provided.

For example, the following code checks if the string `"apple"` is lexicographically less than the string `"banana"`:

```
std::string str1 = "apple";
std::string str2 = "banana";
bool b = std::lexicographical_compare(str1.begin(), str1.end(), str2.begin(), str2.end());  // true
```

The following rules are used when two strings are lexicographically compared:

- Two ranges are compared element by element.
- The first mismatching element defines which range is lexicographically less or greater than the other.
- If one range is a prefix of another, the shorter range is lexicographically less than the other.
- If two ranges have equivalent elements and are the same length, the ranges are lexicographically equal.
- An empty range is lexicographically less than any non-empty range.
- Two empty ranges are lexicographically equal.

Using these rules, the following lexicographical comparison would be false, as uppercase letters have lower ASCII values than lowercase ones:

```
std::string str1 = "apple";
std::string str2 = "Banana";
bool b = std::lexicographical_compare(str1.begin(), str1.end(), str2.begin(), str2.end());  // false
```

Similarly, using the prefix rule, the string `"app"` would compare lexicographically less than the string `"apple"`, and an empty string would compare lexicographically less than the string `"app"`.

Because `std::lexicographical_compare()` checks if a range is strictly lesser, the function will return false if both strings passed in are equal. Furthermore, the function is templated so that it can take in any container that supports input iterators. For instance, the following is a valid usage of the function, even though we are passing in iterators belonging to different containers:

```
std::string str1 = "apple";
std::deque<char> str2 = {'b', 'a', 'n', 'a', 'n', 'a'};
bool b = std::lexicographical_compare(str1.begin(), str1.end(), str2.begin(), str2.end());  // true
```

If you do something like this, make sure to watch the types! For instance, the character `'3'` compares greater than the integer 3, since `'3'` has an ASCII value of 51. Thus, a 3 in a container of integers is very different from a `'3'` in a container of characters!

16.4 STL Sequence Operations

The STL `<algorithm>` library provides several functions that can be used to work with sequences of data. In this section, we will discuss two of these functions, `std::equal()` and `std::unique()`.

※ 16.4.1 *std::equal*

The `std::equal()` function can be used to check if the contents of two interator ranges are equal. Two ranges are considered equal if they have the same number of elements *and* every character in one range matches the character at the same relative position of the other range.

```
template <typename InputIterator1, typename InputIterator2, typename BinaryPredicate>
bool std::equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);
```

Returns `true` if the elements in the iterator range `[first, last)` are equal to the elements in the iterator range `[first2, first2 + (last1 - first1))`, and `false` otherwise. The predicate is optional, and `operator==` is used to compare values if not provided.

Consider the following snippet of code:

```
std::string str1 = "app";
std::string str2 = "apple";
std::cout << std::equal(str1.begin(), str1.end(), str2.begin());   // true
```

When `std::equal()` is called, it checks if all the characters in the range `[str1.begin(), str1.end())` match the characters in the range beginning at `str2.begin()`. Note that the first range determines the length of the sequence that is checked. Since the string `"app"` is used to determine the length of the iterator range, only the first three characters of `"apple"` are checked to see if it matches with `"app"`. As a result, `std::equal()` returns `true` in this example. Using similar logic, the following `std::equal()` call returns `false`, since there exists no match for the last two letters of "apple" in the string "app":

```
std::string str1 = "app";
std::string str2 = "apple";
std::cout << std::equal(str2.begin(), str2.end(), str1.begin());   // false
```

The `std::equal()` operation is helpful if you want to check if one string is a prefix of another:

```
bool is_prefix(const std::string& prefix, const std::string& full_str) {
  return std::equal(prefix.begin(), prefix.end(), full_str.begin());
} // is_prefix()
```

As an additional example, the following function checks if a string is a palindrome using `std::equal()` in just one line:

```
bool is_palindrome(const std::string& str) {
  return std::equal(str.begin(), str.begin() + str.size() / 2, str.rbegin());
} // is_palindrome()
```

※ 16.4.2 *std::unique*

Another useful STL sequence operation is `std::unique()`, which filters out duplicates in a *sorted* container of values.

```
template <typename ForwardIterator, typename BinaryPredicate>
ForwardIterator std::unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);
```

Removes all duplicates in a sorted iterator range by eliminating all but the first element from every consecutive group of identical elements from the range `[first, last)`. An iterator pointing one past the last non-removed element is returned. The predicate is optional, and `operator==` is used to compare values if not provided.

Like with other STL algorithms, the function cannot physically modify the container that the underlying data is stored in. As a result, `std::unique()` only ensures that the elements to be kept are moved to the front of the container. This is why the function returns an iterator one past the last element not removed. After calling `std::unique()`, you must separately erase all the elements from this returned iterator to the end using the `.erase()` member of the container the data is stored in.

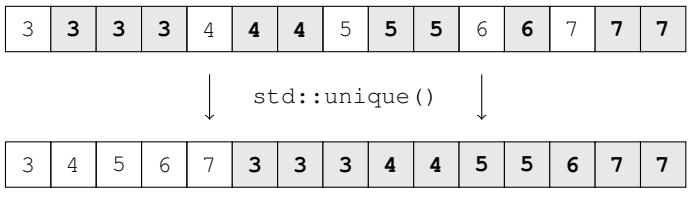
It is also important to note that the `std::unique()` function does not sort the elements for you. If you want to remove duplicates using this function, you must sort the underlying container before passing it into `std::unique()`. Otherwise, `std::unique()` would behave as if the data were sorted, which could lead to incorrect results. Consider the following code:

```
std::vector<int32_t> vec = {5, 5, 5, 3, 4, 4, 4, 3, 3, 3, 6, 7, 7, 6, 7};
std::unique(vec.begin(), vec.end());
```

Here, the vector was not sorted prior to passing it into `std::unique()`. What happens here? When the function is called, it completes a linear pass of the sequence. If it sees a consecutive sequence of elements that are the same, it only keeps the first of the sequence and flags the remaining elements for removal. For example, the bolded elements would be "removed" by `std::unique` on this unsorted container.

Since `std::unique()` only looks at *consecutive* elements that are identical, there is no guarantee that all duplicates will be removed if the underlying container is not sorted. In the above example, there are multiple 3s, 6s, and 7s left over. To ensure all duplicates are removed, the range passed into `std::unique()` must be sorted.

```
std::vector<int32_t> vec = {5, 5, 5, 3, 4, 4, 4, 3, 3, 3, 6, 7, 7, 6, 7};
std::sort(vec.begin(), vec.end());
std::unique(vec.begin(), vec.end());
```

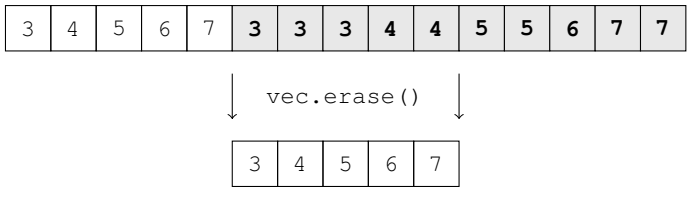


Since `std::unique()` cannot physically remove elements from the vector itself, they must be removed explicitly, as shown:

```
std::vector<int32_t> vec = {5, 5, 5, 3, 4, 4, 4, 3, 3, 3, 6, 7, 7, 6, 7};
std::sort(vec.begin(), vec.end());
auto it = std::unique(vec.begin(), vec.end());  // iterator to first elt to erase
vec.erase(it, vec.end());  // erases all elts from 'it' to the end
```

The last two lines of the above code can be combined (skipping the temporary variable). The following code does the same thing as above:

```
std::vector<int32_t> vec = {5, 5, 5, 3, 4, 4, 4, 3, 3, 3, 6, 7, 7, 6, 7};
std::sort(vec.begin(), vec.end());
vec.erase(std::unique(vec.begin(), vec.end()), vec.end());
```



16.5 Brute Force String Searching

String searching algorithms are undoubtedly one of the most important groups of algorithms in the field of computer science. These algorithms can be used to search for specific text patterns within a large body of text, and they play a pivotal role in search engines, word processing software, and even the field of bioinformatics, just to name a few.

The process of searching for specific sequences of characters in a larger string is sometimes denoted as a *needle in a haystack* problem. This is because string searching is akin to searching for a needle in a haystack, where the needle is the string pattern you are trying to find, and the haystack is the text corpus that you are trying to search in. The goal is to search for instances of the needle in the haystack and return the position of any match. Consider the following needle and haystack:

Haystack: abcdabcdabcdabcbadbccacbabdcabcadabcd
Needle: abca

How would you find the needle in the haystack? A naïve approach is to use a brute force **sliding window** approach. In this solution, the needle "abca" would be compared with all substrings of length 4 by sliding through the haystack until a match is found.

Haystack: **abcd**abcdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: a**bcda**bcdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: ab**cdab**cdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: abc**dabc**dabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

...

Haystack: abcdabcdabcdabcbadbccacbabdc**abca**dabcd
Needle: **abca** → *(match found!)*

What is the time complexity of this brute force algorithm? In the average case, only the first few letters of the needle are enough to determine that a window does not match. For example, any haystack window that begins with a letter other than 'a' can be eliminated immediately in constant time:

Haystack: a**bcda**bcdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(only one character needs to be checked to see that this won't work)*

With this information, we can show that the time required to check each window is constant in the average case, since only the first few letters of the needle are often enough to determine a mismatch. If we denote the length of the needle as n and the length of the haystack as h, the average-case time complexity of the sliding window approach is $\Theta(h)$, since there are a total of $\Theta(h)$ windows that can each be checked in average-case $\Theta(1)$ time.

What about the worst-case time complexity? This happens when every character in the needle needs to be compared for every window in the haystack.

Haystack: **aaaaaaa**aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Needle: **aaaaaab** → *(not equal, so slide window)*

Here, we need to check every character of the needle "aaaaaab" before we can tell that the window does not match. Since the worst-case time complexity of checking each window is $\Theta(n)$, and there are a total of $\Theta(h)$ windows that need to be checked, the worst-case time complexity of the brute force string searching algorithm is $\Theta(nh)$. The code for brute force is shown below:

```
1  std::vector<size_t> sliding_window(const std::string& needle, const std::string& haystack) {
2    std::vector<size_t> matches;
3    const size_t len_needle = needle.length();
4    const size_t len_haystack = haystack.length();
5    for (size_t i = 0; i <= len_haystack - len_needle; ++i) {
6      size_t j = 0;
7      for (; j < len_needle; ++j) {
8        if (haystack[i + j] != needle[j]) {
9          break;  // character mismatch
10       } // if
11     } // for
12     // if for loop completes, the needle matches the window, so store starting index in solution
13     if (j == len_needle) {
14       matches.push_back(i);
15     } // if
16   } // for i
17   return matches;
18 } // sliding_window()
```

Even though a worst-case time complexity of $\Theta(nh)$ is not ideal, the brute force approach is by no means a terrible algorithm, and it can be quite fast in practice. This is because the worst-case scenario rarely happens. However, if the worst case does show up, there are ways to improve performance. One such method is the Rabin-Karp algorithm, which we will cover in the next section.

String Search Method	Average-Case Time	Worst-Case Time
Sliding Window	$\Theta(h)$	$\Theta(nh)$

16.6 Rabin-Karp String Searching

※ 16.6.1 *Rabin Fingerprinting*

If we use the brute force approach, the worst-case time complexity for finding a match is $\Theta(nh)$. This is because a single window comparison may take up to $\Theta(n)$ time if every character of the needle needs to be compared. The **Rabin-Karp string searching algorithm** addresses this inefficiency by guaranteeing a $\Theta(1)$ comparison for each window, regardless of the contents of the string.

In the previous example, checking the equality of "aaaaaaa" and "aaaaaab" took linear time because we had to compare every letter. The Rabin-Karp algorithm avoids this problem by comparing *string fingerprints* instead of the actual strings themselves. In this approach, each string is first converted into an integer in $\Theta(1)$ time. Then, the integers are compared to determine the equality of the strings. This improves the worst-case performance because an integer comparison, unlike a string comparison, takes $\Theta(1)$ time. The function we use to convert each string into an integer is our **fingerprint function**, and the integer that the string gets converted to is its **fingerprint**. A string's fingerprint value may also be referred to as a *rolling hash*.

To gain an intuition for how string fingerprinting works, consider a simple fingerprint function that converts a string to an integer by adding the ASCII value of each character.

```
1  int32_t simple_fingerprint(const std::string& str) {
2    int32_t sum = 0;
3    for (char c : str) {
4      sum += static_cast<int32_t>(c);
5    } // for c
6    return sum;
7  } // simple_fingerprint()
```

Using this fingerprint function, the string "cat" would get converted to the integer 312. This is because 'c' has an ASCII value of 99, 'a' has an ASCII value of 97, and 't' has an ASCII value of 116 — the sum of these three numbers is 312. With string fingerprinting, we can use the integer 312 to do comparisons instead of the string "cat". *If any string does not have a fingerprint of 312, it cannot possibly be the string "cat"!*

However, what if we were given another string that also has a fingerprint of 312? Can we conclude that this other string also has the value "cat"? In this case, no — *fingerprints can only tell us if strings differ, but they cannot tell us if they match*. If a string does not have a fingerprint of 312, it cannot be the string "cat", but if a string does have a fingerprint of 312, we would not know anything without first comparing the two strings themselves! The other string could be "cat", but it could also be any string that has the letters 'c', 'a', and 't' (e.g., "act"). In fact, the other string does not even need this combination of letters; any string whose ASCII values sum to 312 would work (e.g., "kid" = 107 + 105 + 100 = 312). Thus, if two strings have the same fingerprint, they are not guaranteed to be identical — we must check the character compositions of the two strings for equality before we can make such a conclusion.

It turns out that simply adding ASCII values produces a really bad fingerprint function. This is because it produces a lot of *collisions*, where two different strings are assigned to the same fingerprint value. Consider the following:

FP 312:
- "act"
- "are"
- "cat"
- "doe"
- "ear"
- "gap"
- "gel"
- "jam"
- "kid"
- "leg"
- "sad"

FP 430:
- "best"
- "draw"
- "even"
- "fast"
- "frog"
- "girl"
- "hero"
- "link"
- "memo"
- "park"
- "them"

FP 536:
- "alert"
- "bound"
- "drink"
- "equal"
- "fresh"
- "index"
- "level"
- "light"
- "paper"
- "shell"
- "there"

FP 641:
- "beyond"
- "broken"
- "center"
- "choose"
- "either"
- "finish"
- "medium"
- "method"
- "prince"
- "random"
- "recent"

FP 769:
- "airport"
- "control"
- "convert"
- "instant"
- "present"
- "protect"
- "protein"
- "running"
- "through"
- "violent"
- "working"

FP 843:
- "accuracy"
- "critical"
- "diameter"
- "distance"
- "electric"
- "flexible"
- "generate"
- "intended"
- "leverage"
- "marginal"
- "practice"

The table above provides lists of strings that all map to the same fingerprint value. For instance, all of the words in the first column map to a fingerprint of 312. Collisions are not good because they result in false positives, which result in unnecessary string comparisons.

A good fingerprint function should minimize collisions. For instance, if "cat" is assigned a fingerprint of 312, the fingerprint function should (ideally) not assign any other string a fingerprint of 312. That way, it would be extremely rare to encounter a string with a fingerprint of 312 not equal to "cat". To do this, we will use a technique known as **Rabin fingerprinting**. Unlike the previous approach, the Rabin fingerprinting approach also takes into account the *position* of each character alongside its ASCII value when calculating the fingerprint of a string.

Consider the string "cat" again. Instead of simply adding the ASCII values together, we will also multiply each intermediate result with a multiplier value that depends on a character's position. In the example below, we will use 10 as the multiplier:

Old fingerprinting function (position not considered): "cat" → 99 + 97 + 116 = 312

New fingerprinting function (multiply fingerprint by a constant value k every time a new letter is added; here $k = 10$):
"c" → **99** (since 'c' has an ASCII value of 99)
"ca" → (10 × 99) + 97 = **1087** (multiply the fingerprint of "c" by 10, then add the ASCII value of 'a')
"cat" → (10 × 1087) + 116 = **10986** (multiply the fingerprint of "ca" by 10, then add the ASCII value of 't')

Using this method, the string "cat" has a fingerprint of 10986. Since we taking the position of each character into account, it is much more difficult for two different strings to end up with the same fingerprint. Consider the string "act", which had the same fingerprint as "cat" using our initial fingerprinting approach.

Old fingerprinting function (position not considered): "act" → 97 + 99 + 116 = 312

New fingerprinting function (multiply fingerprint by a constant value k every time a new letter is added; here $k = 10$):
"a" → **97** (since 'c' has an ASCII value of 97)
"ac" → (10 × 97) + 99 = **1069** (multiply the fingerprint of "a" by 10, then add the ASCII value of 'c')
"act" → (10 × 1069) + 116 = **10806** (multiply the fingerprint of "ac" by 10, then add the ASCII value of 't')

Notice that "act" and "cat" now have different fingerprints, even though they are made up of the same characters. In fact, if we look at the new fingerprints of the strings in the previous list, we can see that they are now very different:

FP 312:
- "act" → 10806
- "are" → 10941
- "cat" → 10986
- "doe" → 11211
- "ear" → 11184
- "gap" → 11382
- "gel" → 11418
- "jam" → 11679
- "kid" → 11850
- "leg" → 11913
- "sad" → 12570

FP 430:
- "best" → 109366
- "draw" → 112489
- "even" → 113920
- "fast" → 112966
- "frog" → 114613
- "girl" → 114748
- "hero" → 115351
- "link" → 119707
- "memo" → 120301
- "park" → 122947
- "them" → 127519

FP 536:
- "alert" → 1089356
- "bound" → 1103900
- "drink" → 1125707
- "equal" → 1135778
- "fresh" → 1145354
- "index" → 1171130
- "level" → 1193918
- "light" → 1196456
- "paper" → 1229324
- "shell" → 1265288
- "there" → 1275341

FP 641:
- "beyond" → 10943300
- "broken" → 11062820
- "center" → 11032724
- "choose" → 11063351
- "either" → 11277524
- "finish" → 11371754
- "medium" → 12021779
- "method" → 12037610
- "prince" → 12457091
- "random" → 12491219
- "recent" → 12520316

FP 769:
- "airport" → 108764356
- "control" → 111328618
- "convert" → 111329356
- "instant" → 117276916
- "present" → 124536316
- "protect" → 124637206
- "protein" → 124637260
- "running" → 126921703
- "through" → 127663834
- "violent" → 129729316
- "working" → 131358703

FP 843:
- "accuracy" → 1080194811
- "critical" → 1115775978
- "diameter" → 1115903724
- "distance" → 1117769091
- "electric" → 1129218549
- "flexible" → 1139415981
- "generate" → 1143134961
- "intended" → 1172731110
- "leverage" → 1193934831
- "marginal" → 1199547078
- "practice" → 1244817591

Now, it is very difficult for two differing strings to have the same fingerprint. If you wanted to search for the word "cat" in a body of text, the likelihood of encountering a different string with the same fingerprint of 10986 is near zero. What are the implications of this? Consider the worst-case scenario we introduced earlier:

Haystack: **aaaaaaa**aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Needle: **aaaaaab**

If we used brute force, we would have to check all the characters of the needle before we can determine that the two strings are different. However, if we instead converted "aaaaaaa" and "aaaaaab" to their respective fingerprints, 107777767 and 107777768, comparing the equality of these two integers only takes constant time!

But wait... doesn't the fingerprint conversion process take linear time? If we wanted to convert "aaaaaaa" to its fingerprint, we would need to visit every letter of the string, multiply it by a constant, and add an ASCII value. Wouldn't we be doing the same amount of work as brute force, since we need to visit every character of each haystack window regardless? It is indeed true that we need to visit every character of a string to calculate its fingerprint. However, the math behind Rabin fingerprinting works out so that we only need to calculate a fingerprint *twice* from scratch: once for the needle and once for the first haystack window we check. All other fingerprints needed for the algorithm can be computed in $\Theta(1)$ time. Consider the following example:

Haystack: **abc**de
Needle: **cde**

First, let's calculate the fingerprint of the needle, using a multiplier of $k = 10$:
"c" → **99** (since 'c' has an ASCII value of 99)
"cd" → $(10 \times 99) + 100 =$ **1090** (multiply the fingerprint of "c" by 10, then add the ASCII value of 'd')
"cde" → $(10 \times 1090) + 101 =$ **11001** (multiply the fingerprint of "cd" by 10, then add the ASCII value of 'e')

We then repeat this process to find the fingerprint of our first haystack window, "abc":
"a" → **97** (since 'a' has an ASCII value of 97)
"ab" → $(10 \times 97) + 98 =$ **1068** (multiply the fingerprint of "a" by 10, then add the ASCII value of 'b')
"abc" → $(10 \times 1068) + 99 =$ **10779** (multiply the fingerprint of "ab" by 10, then add the ASCII value of 'c')

Because the two fingerprints differ (10779 != 11001), we shift to the next window:

Haystack: **abc**de
Needle: **cde** → *(*$\underbrace{10779}_{window}$ *!=* $\underbrace{11001}_{needle}$*, so slide window)*

Now, we have to compare the fingerprints of "cde" and "bcd" to determine if there is a match.

Haystack: a**bcd**e
Needle: **cde**

We know from before that the fingerprint of "cde" is 11001 since we calculated it previously. However, what is the fingerprint of "bcd"? Do we need to calculate this fingerprint from scratch?

It turns out that we can do some math to elegantly obtain the fingerprint value of "bcd" in $\Theta(1)$ time, given that we already know the fingerprint of the previous haystack window (in this case, "abc"). To do so, we first need to remove the leftmost character of the previous window (i.e., the character that is leaving the window as we slide it forward) from the fingerprint value. To remove this character, we can first multiply its ASCII value by k^{n-1}, where k is the base multiplier and n is the length of the needle. Then, we subtract this value from the previous fingerprint value, multiply the result by the multiplier, and add the ASCII value of the new character that entered the window.

Let's go through the previous example and look at how the fingerprint of "abc" (10779) can be converted to the fingerprint of "bcd" in constant time. In our example, we are using $k = 10$ and $n = 3$, so we need to multiply the leftmost character by $k^{n-1} = 10^{3-1} = 100$ and subtract it from the fingerprint of "abc".

"abc" → **10779**
"bc" → 10779 - $(100 \times 97) =$ **1079** (subtract the contribution of 'a' from the fingerprint)

We are now able to add 'd' using the standard fingerprint calculation process.

"bc" → **1079**
"bcd" → $(10 \times 1079) + 100 =$ **10890** (multiply the fingerprint of "bc" by 10, then add the ASCII value of 'd')

Thus, "bcd" has a fingerprint of 10890. We were able to obtain this value from the fingerprint of "abc" in $\Theta(1)$ time.

Remark: Why were we able to calculate the fingerprint of "bcd" in constant time, given that we already know the fingerprint of "abc"? Notice what is happening when we calculate our Rabin fingerprints:

"a" → **97** (since 'a' has an ASCII value of 97)
"ab" → (10 × 97) + 98 = **1068** (multiply the fingerprint of "a" by 10, then add the ASCII value of 'b')
"abc" → (10 × 1068) + 99 = **10779** (multiply the fingerprint of "ab" by 10, then add the ASCII value of 'c')

If we combined all three operations into one equation, we get

$$\text{fingerprint of "abc"} = 10\times((10\times97)+98)+99 = \underbrace{10\times10\times97}_{\text{contribution of 'a'}} + \underbrace{10\times98}_{\text{contribution of 'b'}} + \underbrace{99}_{\text{contribution of 'c'}}$$

To obtain the fingerprint of "bcd" from the fingerprint of "abc", we need to

1. Remove the contribution of 'a' from the fingerprint of "abc" to get the fingerprint of "bc".
2. Add the contribution of 'd' to the fingerprint of "bc" to get the fingerprint of "bcd".

In general, if you have a string of length n with characters that have ASCII values c_1, c_2, c_3, …, c_n and a base multiplier of k, the string's fingerprint can be calculated using the following equation:

$$k^{n-1}c_1 + k^{n-2}c_2 + k^{n-3}c_3 + \ldots + k^1c_{n-1} + k^0c_n$$

Every time we shift the sliding window, we remove a character from the left of the window and add one on the right. We can easily manipulate the formula above to obtain the fingerprint of one window from the fingerprint of another. The process of turning the fingerprint of "abc" into the fingerprint of "bcd" is shown again below (where the ASCII values of 'a', 'b', and 'c' are 97, 98, and 99, respectively):

Original fingerprint of "abc":

$$\underbrace{10^2\times97}_{\text{contribution of 'a'}} + \underbrace{10^1\times98}_{\text{contribution of 'b'}} + \underbrace{10^0\times99}_{\text{contribution of 'c'}} = \mathbf{10779}$$

Remove the contribution of 'a' from the fingerprint by subtracting $10^2 \times 97$:

$$\underbrace{10^2\times97-10^2\times97}_{\text{remove contribution of 'a'}} + \underbrace{10^1\times98}_{\text{contribution of 'b'}} + \underbrace{10^0\times99}_{\text{contribution of 'c'}} = \mathbf{10779 - 10^2 \times 97}$$

$$\underbrace{10^1\times98}_{\text{contribution of 'b'}} + \underbrace{10^0\times99}_{\text{contribution of 'c'}} = \mathbf{1079}$$

Add 'd' to the fingerprint by multiplying the current fingerprint by the multiplier and adding the ASCII of 'd':

$$\underbrace{10\times(10^1\times98+10^0\times99)}_{\text{fingerprint of "bc" multiplied by 10}} + \underbrace{100}_{\text{ASCII of 'd'}} = \mathbf{18090}$$

This simplifies to the following, which is the fingerprint value of "bcd".

$$\underbrace{10^2\times98}_{\text{contribution of 'b'}} + \underbrace{10^1\times99}_{\text{contribution of 'c'}} + \underbrace{10^0\times100}_{\text{contribution of 'd'}} = \mathbf{10890}$$

Before we continue, there are two things that should be mentioned. First, even though we used 10 as our multiplier for the above examples, it is often better to use a power of two instead. This is because multiplication is faster with powers of two (due to a process known as bit shifting, which you do not have to worry about for this class).

Second, for extremely long strings, fingerprint calculations can lead to overflow errors if you end up with a fingerprint that is larger than the largest number representable using its underlying data type. To account for this, we can use modular arithmetic to ensure our fingerprints do not exceed the largest possible value that we can represent. This is done by picking a very large prime number and taking the modulus of the result with the prime number after each calculation (i.e., result % prime). (Why prime? We will get into more detail when we discuss hash tables in the next chapter, but prime numbers are better at producing uniformly distributed modulus results, and they reduce the likelihood of collisions.) An example of this process is shown below:

"potatobo" → **1243698191**
"potatobot" → (10 × 1243698191) + 116 = **12436982026** (multiply the fingerprint of "potatobo" by 10, then add the ASCII value of 't')

If you are using a 32-bit integer to represent your fingerprint, this value is above the largest possible value that the integer can hold. To address this, we can take the modulus of each computed fingerprint with a large prime number to get a more reasonable value, as shown:

"potatobo" → **1243698191**
"potatobot" → ((10 × 1243698191) + 116) % $\underbrace{183203281}_{\text{large prime}}$ = **162362199**

※ 16.6.2 *Implementing Rabin-Karp String Search*

Now that we have covered Rabin fingerprinting, we can use it to implement the **Rabin-Karp string searching** algorithm, which can be used to solve the needle-in-a-haystack problem more efficiently than brute force. Recall that brute force compares the needle with every possible window in the haystack:

Haystack: **abcd**abcdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: a**bcda**bcdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: ab**cdab**cdabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: abc**dabc**dabcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

Haystack: abcd**abcd**abcdabcbadbccacbabdcabcadabcd
Needle: **abca** → *(not equal, so slide window)*

...

Haystack: abcdabcdabcdabcbadbccacbabdc**abca**dabcd
Needle: **abca** → *(match found!)*

However, this could potentially take worst-case $\Theta(nh)$ time if every window had to check all n characters in the needle.

The Rabin-Karp algorithm addresses this issue by comparing integers instead of strings, and it does so by using Rabin fingerprinting to convert strings to integers. Rabin-Karp is implemented using the following procedure:

1. The fingerprint of the needle is calculated in $\Theta(n)$ time, where n is the length of the needle.
2. The fingerprint of the first n characters of the haystack is calculated, also in $\Theta(n)$ time.
3. The fingerprint of the needle is compared with the fingerprint of the window.
4. If the two fingerprint values match, a brute force comparison is done to confirm that the contents of the two strings are indeed the same (this needs to be done in case two different strings end up with the same fingerprint).
5. If the two fingerprints do not match, the needle slides to the right, and the fingerprint is updated for the new window. This is done by
   - Removing the character on the left of the window by subtracting ($k^{n-1}\times$ ASCII) from the existing fingerprint.
   - Multiplying this intermediate value by k (the base multiplier).
   - Adding the ASCII value of the new character in the window to the result.
6. As long as there exist more characters in the haystack, repeat steps 4 and 5.

The Rabin-Karp process is shown below, using a base multiplier of $k = 32$.

Haystack: **abcd**abcdabcdabcbadbccacbabdcabcadabcd
Needle: **abca**

First, we calculate the fingerprints of the needle ("abca") and the first window of the haystack ("abcd"):

$\text{FP("abca")} = 32^3 \times 97 + 32^2 \times 98 + 32^1 \times 99 + 32^0 \times 97 = 3282113$
$\text{FP("abcd")} = 32^3 \times 97 + 32^2 \times 98 + 32^1 \times 99 + 32^0 \times 100 = 3282116$

We then compare the two fingerprints. Since they do not match, we know that the strings in the first window also do not match.

Haystack: $\overbrace{\textbf{abcd}}^{3282116}$ abcdabcdabcbadbccacbabdcabcadabcd
Needle: $\underbrace{\textbf{abca}}_{3282113}$

Because the window does not match, we slide the needle one character to the right and update the fingerprint of the window, which can be done in constant time. This process is repeated until a match is found.

Haystack: $\overbrace{\textbf{abcd}}^{3282116}$ abcdabcdabcbadbccacbabdcabcadabcd
Needle: $\underbrace{\textbf{abca}}_{3282113}$ → *(3282116 != 3282113, so slide window and update fingerprint)*

Haystack: a $\overbrace{\textbf{bcda}}^{3315937}$ bcdabcdabcbadbccacbabdcabcadabcd
Needle: $\underbrace{\textbf{abca}}_{3282113}$ → *(3315937 != 3282113, so slide window and update fingerprint)*

Haystack: ab $\overbrace{\underline{\textbf{cdab}}}^{3349634}$ cdabcdabcbadbccacbabdcabcadabcd
Needle: $\underbrace{\textbf{abca}}_{3282113}$ → *(3349634 != 3282113, so slide window and update fingerprint)*

Haystack: abc $\overbrace{\underline{\textbf{dabc}}}^{3379363}$ dabcdabcbadbccacbabdcabcadabcd
Needle: $\underbrace{\textbf{abca}}_{3282113}$ → *(3379363 != 3282113, so slide window and update fingerprint)*

...

Haystack: abcdabcdabcdabcbadbccacbabdc $\overbrace{\underline{\textbf{abca}}}^{3282113}$ dabcd
Needle: $\underbrace{\textbf{abca}}_{3282113}$ → *(3282113 == 3282113, match found!)*

Let's code the implementation of Rabin-Karp. First, let's initialize a few variables that we will need throughout the algorithm:

```
constexpr int32_t base = 128;                       // this is k, or our multiplier
constexpr int64_t prime = 376370281280203183;       // this prime will be used for our modulus
const size_t len_needle = needle.length();          // length of needle
const size_t len_haystack = haystack.length();      // length of haystack
std::vector<size_t> matches;                        // stores indices of matches
```

To remove the leftmost character from our window, we will need to subtract ($k^{n-1}\times$ ASCII) from our fingerprint. Let's create a variable to store the value of k^{n-1} (as mentioned, we take its modulus with `prime` to prevent overflow).

```
const int64_t left_multiplier = static_cast<int64_t>(std::pow(base, len_needle - 1)) % prime;
```

Then, we will calculate the fingerprint of the needle and the first search window of the haystack.

```
int64_t fp_needle = 0;
int64_t fp_haystack = 0;
for (size_t i = 0; i < len_needle; ++i) {
  fp_needle = (base * fp_needle + needle[i]) % prime;
  fp_haystack = (base * fp_haystack + haystack[i]) % prime;
} // for i
```

After this, we will need to iterate through all the windows of the haystack, as shown. We iterate up to `len_haystack - len_needle` so that we do not iterate off the end of the haystack when comparing with the needle.

```
for (size_t i = 0; i <= len_haystack - len_needle; ++i) {
  if (fp_needle == fp_haystack) {
    // do work if fingerprints are equal
  } // if
  // slide the needle one window to the right and calculate the fingerprint
  // of the new window (subtract left char and add right char)
} // for i
```

If the fingerprints ever match, we will still need to do a string comparison to make sure the strings are actually the same. This is because it is still possible for two strings to end up with the same fingerprint value.

```
for (size_t i = 0; i <= len_haystack - len_needle; ++i) {
  if (fp_needle == fp_haystack) {
    size_t j = 0;
    for (; j < len_needle; ++j) {
      if (haystack[i + j] != needle[j]) {
        break;
      } // if
    } // for
    // if for loop completes without breaking, the strings are equal
    if (j == len_needle) {
      matches.push_back(i);
    } // if
  } // if
  ...
} // for i
```

Then, we would check the next window by sliding the needle rightward and recalculating the window's fingerprint:

```
for (size_t i = 0; i <= len_haystack - len_needle; ++i) {
  ...  // continued from line 14 previously
  // fingerprint with leftmost character removed
  int64_t fp_no_left = fp_haystack - haystack[i] * left_multiplier;
  fp_haystack = (base * fp_no_left + haystack[i + len_needle]) % prime;
  // if fingerprint becomes negative, add the prime to make it positive
  // this is okay because of modular arithmetic
  if (fp_haystack < 0) {
    fp_haystack = fp_haystack + prime;
  } // if
} // for i
```

Putting this all together, we have the following Rabin-Karp function to find all instances of a needle within a larger string:

```
std::vector<size_t> rabin_karp(const std::string& needle, const std::string& haystack) {
  std::vector<size_t> matches;
  constexpr int32_t base = 128;
  constexpr int64_t prime = 376370281280203183;
  const size_t len_needle = needle.length();
  const size_t len_haystack = haystack.length();
  const int64_t left_multiplier = static_cast<int64_t>(std::pow(base, len_needle - 1)) % prime;
  int64_t fp_needle = 0;
  int64_t fp_haystack = 0;
  for (size_t i = 0; i < len_needle; ++i) {
    fp_needle = (base * fp_needle + needle[i]) % prime;
    fp_haystack = (base * fp_haystack + haystack[i]) % prime;
  } // for i
  for (size_t i = 0; i <= len_haystack - len_needle; ++i) {
    if (fp_needle == fp_haystack) {
      size_t j = 0;
      for (; j < len_needle; ++j) {
        if (haystack[i + j] != needle[j]) {
          break;
        } // if
      } // for
      if (j == len_needle) {
        matches.push_back(i);
      } // if
    } // if
    int64_t fp_no_left = fp_haystack - haystack[i] * left_multiplier;
    fp_haystack = (base * fp_no_left + haystack[i + len_needle]) % prime;
    if (fp_haystack < 0) {
      fp_haystack = fp_haystack + prime;
    } // if
  } // for i
  return matches;
} // rabin_karp()
```

※ 16.6.3 *Rabin-Karp Complexity Analysis*

The efficiency of Rabin-Karp depends on the frequency at which collisions occur. Every time two strings have the same fingerprint, the algorithm must do a $\Theta(n)$ string comparison to ensure the strings are indeed equal. Ideally, two different strings should never share the same fingerprint; if this is the case, the runtime of Rabin-Karp would be $\Theta(h)$ in the worst case, since checking each window would only take $\Theta(1)$ time.

However, this is not entirely a guarantee; if the fingerprinting function is awful and produces a false positive at every step, then you could end up with a worst-case time complexity of $\Theta(nh)$. When this happens, the algorithm would think it found a match at every position of the haystack and could potentially perform a $\Theta(n)$ comparison to verify that the needle actually matches for all $\Theta(h)$ positions of the haystack. This matches the worst-case time complexity of brute force! That being said, while this scenario is theoretically possible, it is extremely improbable in practice if a good fingerprinting function is used, especially since Rabin-Karp is designed to safeguard against this happening.

The average-case time complexity of Rabin-Karp is $\Theta(n+h)$. This is because it takes $\Theta(1)$ time to check each window if a good fingerprinting function is used (which makes it rare for two different strings to share the same fingerprint). There are a total of $\Theta(h)$ windows that need to be checked in the haystack, and each match requires a $\Theta(n)$ pass over the needle to verify that the window actually matches (to make sure it is not a false positive where two differing strings end up with the same fingerprint).

String Search Method	Average-Case Time	Worst-Case Time
Rabin-Karp	$\Theta(n+h)$	$\Theta(nh)$

16.7 Knuth-Morris-Pratt (KMP) String Searching (⁎)

※ 16.7.1 *Prefix Tables* (⁎)

In this section, we will be discussing the **Knuth-Morris-Pratt (KMP) string searching** algorithm, which is another important string matching algorithm that can be used to improve the worst-case performance of string searching. The KMP algorithm differs from brute force in that it keeps track of information from previous comparisons to reduce the number of comparisons that may be needed later on. To understand how KMP works, consider the following example that was introduced earlier:

Haystack: **aaaaaaa**aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Needle: **aaaaaab** → *(not equal, so slide window)*

If we used the brute force approach, we would compare "aaaaaaa" with "aaaaaab", see that the two strings are not equal, and slide the window forward by one. Then, we would compare "aaaaaaa" with "aaaaaab" again, as shown.

Haystack: a**aaaaaaa**aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
Needle: **aaaaaab** → *(not equal, so slide window)*

However, much of this work is unnecessary. When we detected a mismatch in the first window (‘a’ != ‘b’), we had already discovered that the first six characters matched. Thus, we do not need to compare these characters again when we move on to the second window, since we already know they match from the comparisons we made in the previous window. This is where the KMP algorithm comes into play.

To better illustrate this, consider the following modified example:

Haystack: **abdcdabdc**abdbcdbccaabdcdabcabdcdabcbab
Needle: **abdcdabcb** → *(not equal, so slide window)*

Here, the first difference is at the second to last position of the needle, which is ‘d’ in the haystack but ‘c’ in the needle.

Haystack: **abdcdab d c**abdbcdbccaabdcdabcabdcdabcbab

Needle: **abdcdab c b**

Notice that the sequence of characters that comes directly before the mismatch is "ab" (which is underlined below).

Haystack: **abdcdab d c**abdbcdbccaabdcdabcabdcdabcbab

Needle: **abdcdab c b**

However, this same sequence "ab" also appears at the *beginning* of our window.

Haystack: **abdcdab d c**abdbcdbccaabdcdabcabdcdabcbab

Needle: **abdcdab c b**

We can take advantage of this information to skip several comparisons. Since the needle starts with the characters "ab", and the characters directly before the mismatch are also "ab", we can align the first "ab" of the needle with the last "ab" of our previous haystack window. Furthermore, since we already know that the "ab" sequences match, we can start our comparisons at the position of the two ‘d’ characters (which are boxed below) instead of the beginning of the window:

Haystack: abdcd**ab d cabdbc**dbccaabdcdabcabdcdabcbab

Needle: → **ab d cdabcb**

With this optimization, we are able to traverse the haystack without ever having to move backwards. This idea forms the basis for how KMP works. Now, let’s look at how KMP is implemented.

First, we will introduce some terms that are important for the problem. The **prefix** of a string is any substring of that string that begins at the first character. For example, the following are prefixes of the string "abdcdabcb":

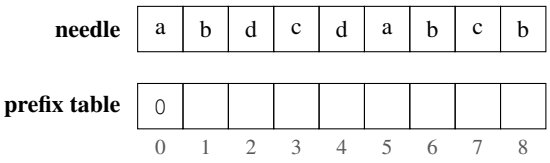
```
"a", "ab", "abd", "abdc", "abdcd", "abdcda", "abdcdab", "abdcdabc", "abdcdabcb"
```

On the other hand, the **suffix** of a string is any substring of that string that ends at the last character. For example, the following are suffixes of the string "abdcdabcb":

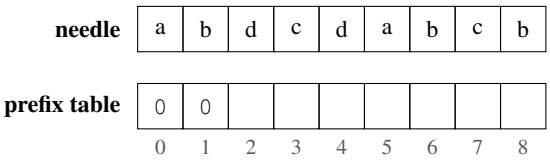
```
"b", "cb", "bcb", "abcb", "dabcb", "cdabcb", "dcdabcb", "bdcdabcb", "abdcdabcb"
```

To perform KMP string search, we must first preprocess our needle and store additional information that can be used to determine how far we can shift the needle after a mismatch. This is done by constructing something known as a **prefix table** (also known as a π *table* or an *LPS table*, where LPS stands for "longest prefix suffix"). The prefix table is an array that has the same length as the needle, and each index i of the prefix table stores the length of the longest suffix that is also a prefix of the substring up to index i (ignoring the suffix that matches with the entire string). Because we never consider the suffix that matches with the entire string, index 0 of the prefix table will always have a value of 0.

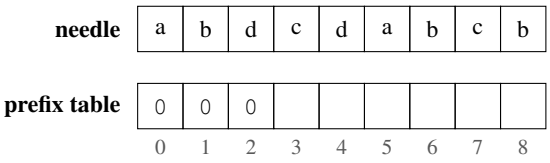
For example, let's use the string "abdcdabcb" to construct a prefix table. Since this string has a length of 9, the prefix table will also have a length of 9. The value at index 0 of the prefix table is set to 0. This is shown below:



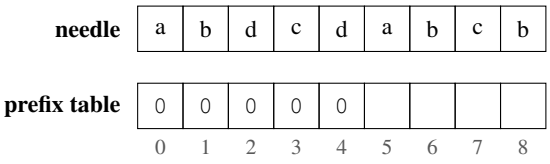
We will now go through all the prefixes of the string and identify the length of the longest suffix that is also a prefix. Index 1 of the prefix table stores the length of the longest suffix that is also a prefix of the substring up to index 1 ("ab"). Ignoring the case where the entire string matches with itself, there is no suffix that is also a prefix. Thus, index 1 of the prefix table stores **0**.



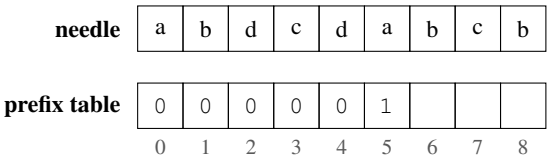
Index 2 of the prefix table stores the length of the longest suffix that is also a prefix of the substring up to index 2 ("abd"). Ignoring the self-match case, there is no suffix that is also a prefix. Thus, index 2 of the prefix table also stores **0**.



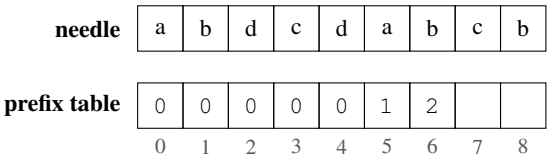
If we continue this process for indices 3 and 4 (the substrings "abdc" and "abdcd"), we would see that there is no suffix that is also a prefix of the string (ignoring the self-match case). As a result, indices 3 and 4 of the prefix table are both **0**.



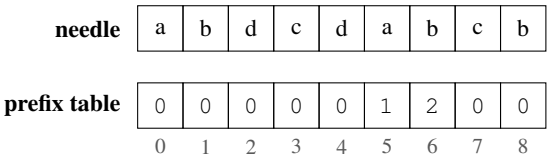
Index 5 of the prefix table stores the length of the longest suffix that is also a prefix of the substring up to index 5, or "abdcda". Here, there is a suffix that matches with a prefix: "**a**bdcd**a**". This suffix has a length of 1, so index 5 of the prefix table stores **1**.



Index 6 of the prefix table stores the length of the longest suffix that is also a prefix of the substring up to index 6, or "abdcdab". Here, there is a suffix that matches with a prefix: "**ab**dcd**ab**". This suffix has a length of 2, so index 6 of the prefix table stores **2**.



Index 7 of the prefix table stores the length of the longest suffix that is also a prefix of the substring up to index 7, or "abdcdabc". Here, there is no longer a suffix that matches a prefix (ignoring the self-match case). Thus, index 7 of the prefix table stores **0**. This is also true for index 8.



This is our final prefix table for the needle "abdcdabcb".

Example 16.1 Suppose you are using the KMP string searching algorithm to find instances of the string "aabcbaaabcab" (the needle) within a larger body of text (the haystack). Construct the prefix table for this given needle.

To solve this problem, find the longest suffix that is also a prefix for the substring up to each index of the string. This process is shown below (remember that index 0 of the prefix table will always have a value of 0):

- Index 1: "**a** **a**" → longest matching suffix has a length of 1
- Index 2: "aab" → no suffix matches a prefix, so longest matching suffix has a length of 0
- Index 3: "aabc" → no suffix matches a prefix, so longest matching suffix has a length of 0
- Index 4: "aabcb" → no suffix matches a prefix, so longest matching suffix has a length of 0
- Index 5: "**a**abcb**a**" → longest matching suffix has a length of 1
- Index 6: "**aa**bcb**aa**" → longest matching suffix has a length of 2
- Index 7: "**aa**bcba**aa**" → longest matching suffix has a length of 2
- Index 8: "**aab**cba**aab**" → longest matching suffix has a length of 3
- Index 9: "**aabc**ba**aabc**" → longest matching suffix has a length of 4
- Index 10: "**a**abcbaaabc**a**" → longest matching suffix has a length of 1
- Index 11: "aabcbaaabcab" → no suffix matches a prefix, so longest matching suffix has a length of 0

Thus, the final prefix table is:

needle	a	a	b	c	b	a	a	a	b	c	a	b

prefix table	0	1	0	0	0	1	2	2	3	4	1	0
	0	1	2	3	4	5	6	7	8	9	10	11

Example 16.2 Suppose you are using the KMP string searching algorithm to find instances of the string "aabaabaaa" (the needle) within a larger body of text (the haystack). Construct the prefix table for this given needle.

To solve this problem, find the longest suffix that is also a prefix for the substring up to each index of the string. This process is shown below (remember that index 0 of the prefix table will always have a value of 0):

- Index 1: "**a** **a**" → longest matching suffix has a length of 1
- Index 2: "aab" → no suffix matches a prefix, so longest matching suffix has a length of 0
- Index 3: "**a**ab**a**" → longest matching suffix has a length of 1
- Index 4: "**aa**b**aa**" → longest matching suffix has a length of 2
- Index 5: "**aab** **aab**" → longest matching suffix has a length of 3
- Index 6: "**aabaaba**" → longest matching suffix has a length of 4

  Remark: Notice that overlaps are fine, as long as you are *not* overlapping the suffix with the entire string (e.g., for index 2, you cannot match "aab" with "aab" and claim a longest matching suffix length of 3).

- Index 7: "**aabaabaa**" → longest matching suffix has a length of 5
- Index 8: "**aa**baaba**aa**" → longest matching suffix has a length of 2

Thus, the final prefix table is:

needle	a	a	b	a	a	b	a	a	a

prefix table	0	1	0	1	2	3	4	5	2
	0	1	2	3	4	5	6	7	8

※ 16.7.2 *Implementing Knuth-Morris-Pratt String Search* (✻)

After we construct the prefix table for the needle, we can now begin our KMP string search. We will iterate through the haystack, using the prefix table to identify how many comparisons we can skip after each mismatch. Let's return to our previous haystack and needle as an example:

Haystack: **abdcdabdc**abdbcdbccaabdcdabcabdcdabcbab
Needle: **abdcdabcb**

needle	a	b	d	c	d	a	b	c	b

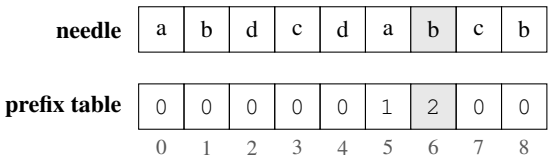
prefix table	0	0	0	0	0	1	2	0	0
	0	1	2	3	4	5	6	7	8

We first begin by comparing the characters in the haystack one by one, until we encounter a mismatch. The first mismatch we encounter is the second to last letter in the needle, which is 'd' in the haystack but 'c' in the needle.

Haystack: **abdcdab d c**abdbcdbccaabdcdabcabdcdabcbab

Needle: **abdcdab c b**

After encountering this mismatch, we go to the prefix table and look up the value associated with the character directly before the mismatch. In this case, the character before the mismatch is the 'b' highlighted below:



The value in the prefix table is 2, so we will align the character at index 2 of the needle (the 'd') with the mismatched character in the haystack (shown below). Then, we will continue comparing the characters in the haystack, *starting from the position of the mismatch*. We do not compare the first "ab" again because we already know they match! This allows us to traverse the haystack without ever moving backwards to check characters that have already been compared before.

Haystack: abdcd**ab d cabdbc**dbccaabdcdabcabdcdabcbab

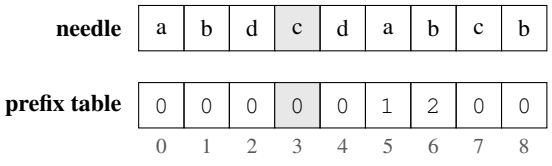
Needle: → **ab d cdabcb**

The next mismatch happens at this point:

Haystack: abdcd**abdc a bdbc**dbccaabdcdabcabdcdabcbab

Needle: **abdc d abcb**

The character in the needle directly before the mismatch is the 'c', which has a prefix table value of 0.



Thus, we will align the character at index 0 of the needle with the mismatched character in the haystack, as shown:

Haystack: abdcdabdc **a bdbcdbcc**aabdcdabcabdcdabcbab

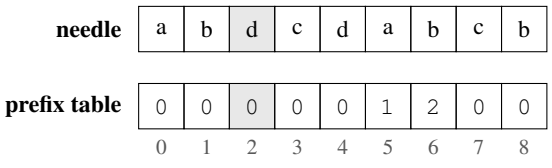
Needle: → **a bdcdabcb**

The next mismatch happens at this point:

Haystack: abdcdabdc**abd b cdbcc**aabdcdabcabdcdabcbab

Needle: → **abd c dabcb**

The character directly before the mismatch is the 'd', which has a prefix value of 0.



Thus, we will align the character at index 0 of the needle with the mismatched character in the haystack, as shown:

Haystack: abdcdabdcabd **b cdbccaab**dcdabcabdcdabcbab

Needle: → **a bdcdabcb**

Notice here that there is a mismatch on the first character of the needle. When this happens, you would just slide the needle forward by one character, since there are no comparisons that you are able to immediately skip.

Haystack: abdcdabdcabdb **c dbccaabd**cdabcabdcdabcbab

Needle: → **a bdcdabcb** *(mismatch on first character, so slide window forward by one)*

Haystack: abdcdabdcabdbc **d** **bccaabdc**dabcabdcdabcbab

Needle: → **a** **bdcdabcb** *(mismatch on first character, so slide window forward by one)*

Haystack: abdcdabdcabdbcd **b** **ccaabdcd**abcabdcdabcbab

Needle: → **a** **bdcdabcb** *(mismatch on first character, so slide window forward by one)*

Haystack: abdcdabdcabdbcdb **c** **caabdcda**bcabdcdabcbab

Needle: → **a** **bdcdabcb** *(mismatch on first character, so slide window forward by one)*

Haystack: abdcdabdcabdbcdbc **c** **aabdcdab**cabdcdabcbab

Needle: → **a** **bdcdabcb** *(mismatch on first character, so slide window forward by one)*

Haystack: abdcdabdcabdbcdbcc **a** **abdcdabc**abdcdabcbab

Needle: → **a** **bdcdabcb**

In this window, the next mismatch is at the second character:

Haystack: abdcdabdcabdbcdbcca **a** **bdcdabc**abdcdabcbab

Needle: → **a** **b** **dcdabcb**

The prefix value of the character before the mismatch is 0:

needle	a	b	d	c	d	a	b	c	b
prefix table	0	0	0	0	0	1	2	0	0
	0	1	2	3	4	5	6	7	8

Thus, we will align the character at index 0 of the needle with the mismatched character in the haystack, as shown:

Haystack: abdcdabdcabdbcdbcca **a** **bdcdabca**bdcdabcbab

Needle: → **a** **bdcdabcb**

At this point, our next mismatch occurs at the last letter of the needle:

Haystack: abdcdabdcabdbcdbcca**abdcdabc** **a** bdcdabcbab

Needle: → **abdcdabc** **b**

The character directly before the mismatch is the 'c', which has a prefix value of 0.

needle	a	b	d	c	d	a	b	c	b
prefix table	0	0	0	0	0	1	2	0	0
	0	1	2	3	4	5	6	7	8

Thus, we will align the character at index 0 of the needle with the mismatched character in the haystack, as shown.

Haystack: abdcdabdcabdbcdbccaabdcdabc **a** **bdcdabcb**ab

Needle: → **a** **bdcdabcb**

We now have a match, so we record its position. To continue finding matches, we would align the character in the needle whose index is the last value in the prefix table (in this case, index 0) with the character directly after the current window of the haystack and repeat the above procedure, as shown below. We use the last value of the prefix table to determine where to continue comparisons; for instance, if the last value in the prefix table had been something else like 2, we would instead align the next character in the haystack with index 2 of our needle.

needle	a	b	d	c	d	a	b	c	b
prefix table	0	0	0	0	0	1	2	0	0
	0	1	2	3	4	5	6	7	8

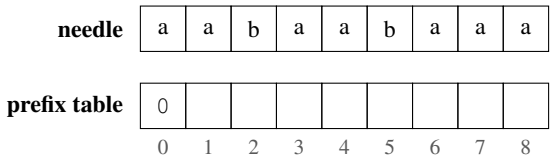
Haystack: ... ccaabdcdabcabdcdabcb **a** **b ...**

Needle: → **a** **bdcdabcb**

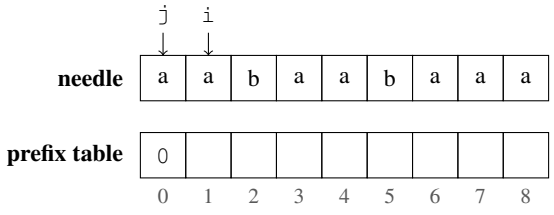
The code for the KMP string search algorithm is provided below:

```
1   std::vector<size_t> compute_prefix_array(const std::string& needle) {
2     std::vector<size_t> prefix_arr(needle.length());
3     size_t j = 0;
4     for (size_t i = 1; i < needle.length(); ++i) {
5       while (j > 0 && needle[j] != needle[i]) {
6         j = prefix_arr[j - 1];
7       } // while
8       if (needle[j] == needle[i]) {
9         ++j;
10      } // if
11      prefix_arr[i] = j;
12    } // for i
13    return prefix_arr;
14  } // compute_prefix_array()
15
16  std::vector<size_t> knuth_morris_pratt(const std::string& needle, const std::string& haystack) {
17    std::vector<size_t> matches;
18    std::vector<size_t> prefix_arr = compute_prefix_array(needle);
19    size_t idx = 0;
20    for (size_t curr = 0; curr < haystack.length(); ++curr) {
21      while (idx > 0 && needle[idx] != haystack[curr]) {
22        idx = prefix_arr[idx - 1];
23      } // while
24      if (needle[idx] == haystack[curr]) {
25        ++idx;
26      } // if
27      if (idx == needle.length()) {
28        matches.push_back(curr - needle.length() + 1);
29      } // if
30    } // for curr
31    return matches;
32  } // knuth_morris_pratt()
```

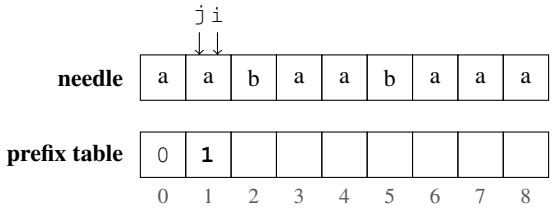
To illustrate how this works, let's walk through the code for building a prefix table (lines 1-14) using the following example:



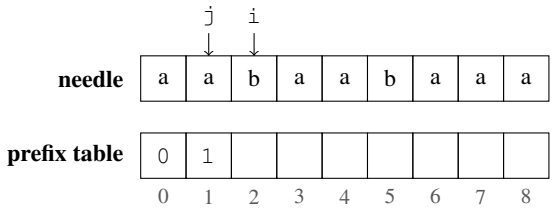
We first initialize two indices, `i` and `j`, that are used to iterate through the needle. `i` is initially set to 1 and `j` is initially set to 0.



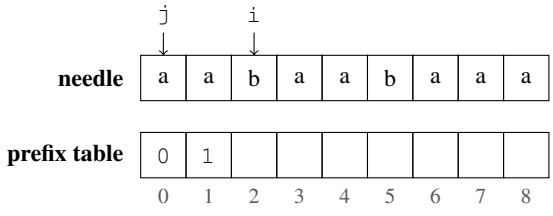
Because `j` is 0, the `while` loop on line 5 does not run. We hit the `if` check on line 8, which returns `true`, so `j` is incremented to index 1, and `prefix_arr[1]` is set to `j` = 1.



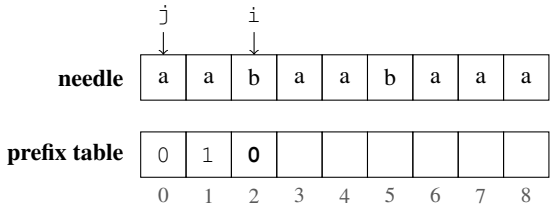
`i` is then incremented for the next iteration of the `for` loop.



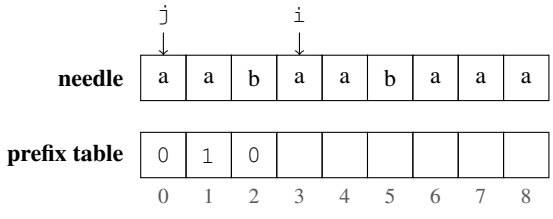
`j` is greater than 0, and `needle[j]` and `needle[i]` do not match. As a result, `j` gets set to `prefix_arr[j - 1]`, or index 0.



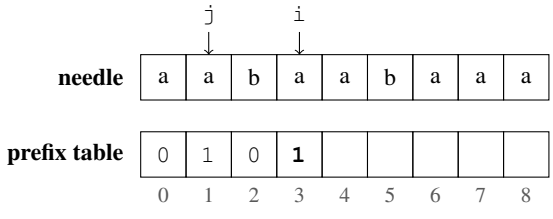
The characters at `i` and `j` still do not match, so the `if` check on line 8 does not run, and `prefix_arr[2]` gets set to `j`, or 0.



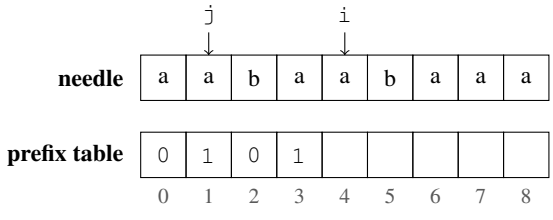
`i` is incremented for the next iteration of the `for` loop. `j` is 0, so the `while` loop on line 5 does not run.



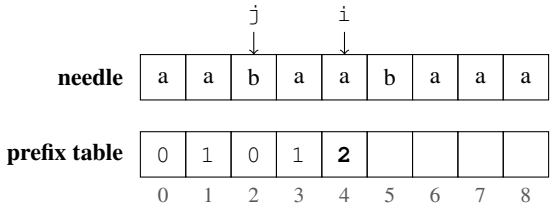
The characters at positions `i` and `j` are the same (both 'a'), so `j` is incremented to index 1, and `prefix_arr[3]` is also set to 1.



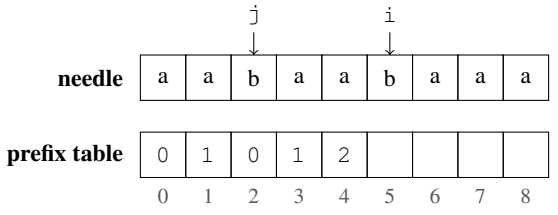
`i` is incremented for the next iteration of the `for` loop.



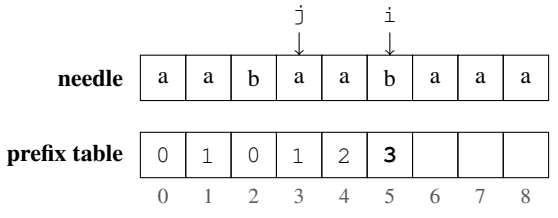
The characters at positions `i` and `j` are the same (both 'a'), so the `while` loop does not run. The `if` condition returns `true`, so `j` is incremented to index 2, and `prefix_arr[4]` is also set to 2.



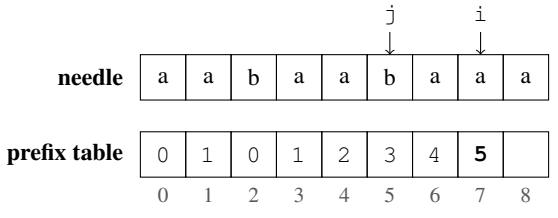
`i` is incremented for the next iteration of the `for` loop.



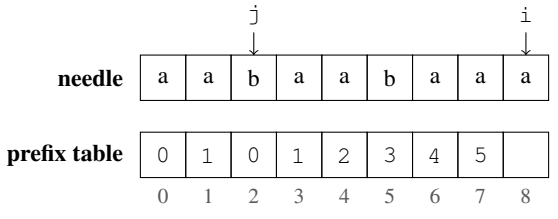
The characters at positions `i` and `j` are the same (both 'b'), so the `while` loop does not run. The `if` condition returns `true`, so `j` is incremented to index 3, and `prefix_arr[5]` is also set to 3.



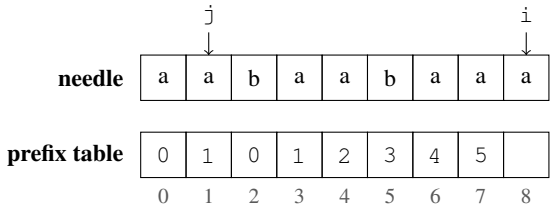
Repeating the process for the next two iterations, we would end up with the following:



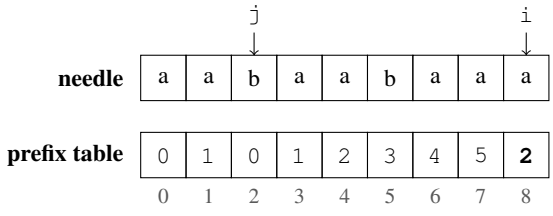
`i` is incremented one more time for the last iteration of the `for` loop. The characters at positions `i` and `j` are not the same after the incrementation, and `j` is not 0. Thus, the `while` loop runs, and `j` moves back to `prefix_arr[j - 1]`, or the prefix value of the character directly before the mismatch. In this case, `prefix_arr[j - 1]` is 2, so `j` moves back to index 2.



The characters at `i` and `j` still do not match, so the `while` loop continues iterating, and `j` moves back to `prefix_arr[j - 1]` again. In this example, `prefix_arr[j - 1]` is 1, so `j` moves back to index 1.



The characters at `i` and `j` now match, so the `while` loop completes. Since the characters at these two positions are the same (both 'a'), the `if` statement returns `true`, and `j` is incremented to index 2. The value of `prefix_arr[8]` is then set to 2 as well.



This is our final prefix table. The code for the actual KMP search (lines 16-32) is fairly straightforward and closely follows the description of the algorithm on the previous few pages. The `for` loop on line 20 iterates through the haystack one character at a time. If there is a mismatch between the needle and the haystack, the `while` loop on line 21 correctly aligns the needle with the haystack based on the prefix value of the character directly before the mismatch (`prefix_arr[idx - 1]`). The `if` check on line 24 takes care of the comparison process, incrementing the index of the needle as long as it matches the corresponding character in the haystack. If the index of the needle ever reaches the end (line 27), the entire window must have matched, so we record the position of the match (i.e., the haystack index of the first character in the match) in the output vector. The code is reproduced below for convenience.

```
1  std::vector<size_t> compute_prefix_array(const std::string& needle) {
2    std::vector<size_t> prefix_arr(needle.length());
3    size_t j = 0;
4    for (size_t i = 1; i < needle.length(); ++i) {
5      while (j > 0 && needle[j] != needle[i]) {
6        j = prefix_arr[j - 1];
7      } // while
8      if (needle[j] == needle[i]) {
9        ++j;
10     } // if
11     prefix_arr[i] = j;
12   } // for i
13   return prefix_arr;
14 } // compute_prefix_array()
15
16 std::vector<size_t> knuth_morris_pratt(const std::string& needle, const std::string& haystack) {
17   std::vector<size_t> matches;
18   std::vector<size_t> prefix_arr = compute_prefix_array(needle);
19   size_t idx = 0;
20   for (size_t curr = 0; curr < haystack.length(); ++curr) {
21     while (idx > 0 && needle[idx] != haystack[curr]) {
22       idx = prefix_arr[idx - 1];
23     } // while
24     if (needle[idx] == haystack[curr]) {
25       ++idx;
26     } // if
27     if (idx == needle.length()) {
28       matches.push_back(curr - needle.length() + 1);
29     } // if
30   } // for curr
31   return matches;
32 } // knuth_morris_pratt()
```

※ 16.7.3 *Knuth-Morris-Pratt Complexity Analysis* (✻)

What are the time and space complexities of the KMP algorithm? The KMP process consists of two steps: constructing the prefix table and iterating through the haystack. If the needle has length n and haystack has length h, the overall time complexity of KMP is $\Theta(n+h)$, since building the prefix table takes $\Theta(n)$ time, and iterating through the haystack takes $\Theta(h)$ time.

Why is this the case? First, let's consider the work required to build the prefix table. Since the number of times the inner `while` loop runs (line 5) depends on the outer `for` loop (line 4), we have a loop dependency, so we will analyze the runtime of the two loops together. Notice that `j` starts at 0 and is only incremented *at most once* per iteration of the outer loop (on line 9). Thus, the value of `j` cannot exceed $n-1$, which is the total number of iterations the outer loop runs. Furthermore, notice that `i` is *always* incremented once per iteration of the outer loop. Because `j` is incremented *at most* once per iteration of the outer loop, and `j` starts off smaller than `i`, the value of `j` cannot ever be larger than `i`.

```
3   size_t j;
4   for (size_t i = 1; i < needle.length(); ++i) {
5     while (j > 0 && needle[j] != needle[i]) {
6       j = prefix_arr[j - 1];
7     } // while
8     if (needle[j] == needle[i]) {
9       ++j;
10    } // if
11    prefix_arr[i] = j;
12  } // for i
```

Since `prefix_arr[i]` is set to `j` (line 11), and `j` cannot be larger than `i`, we can conclude that `prefix_arr[j - 1]` must be less than `j`. As a result, each iteration of the inner `while` loop ends up *decreasing* the value of `j`, since we are assigning `j` to `prefix_arr[j - 1]` (whose value must be smaller than the original value of `j`). However, in order for `j` to decrease in value from the assignment on line 6, it must have increased its value earlier in the algorithm (for example, in order for `j` to fall from a value of 2 to 0 in the inner loop, it must have incremented to a value of 2 sometime earlier in the algorithm, which requires at least two iterations of the outer loop to run). Because `j` cannot fall below 0, the inner loop can only decrease `j` at most as many times as the total increase of `i` in the outer loop. Thus, the number of iterations completed by the inner `while` loop is bounded by the total increase of `i` in the outer loop, or $n-1=\Theta(n)$. Since the remaining work done in the `for` loop (lines 9 to 11) takes $\Theta(1)$ time, the total work done by the entire loop dependency (lines 4 to 12) is $\Theta(n)+\Theta(1)=\Theta(n)$.

We can use a similar analysis to show that the time complexity of iterating through the haystack (after calculating the prefix table) is $\Theta(h)$. For each iteration of the `for` loop on line 20, the value of `idx` increases by at most 1 (on line 25). Because `prefix_arr[idx - 1]` is always less than `idx`, the inner `while` loop on line 21 always decreases the value of `idx`. Since `idx` cannot be negative, the number of times we can decrease `idx` is bounded by the value of `idx`, which is itself bounded by the number of times the outer `for` loop has run. Since the outer `for` loop runs at most $\Theta(h)$ times, and the remaining work in the outer loop runs in constant time, the time complexity of the entire loop dependency (lines 20 to 30) is also $\Theta(h)$.

```
19  size_t idx = 0;
20  for (size_t curr = 0; curr < haystack.length(); ++curr) {
21    while (idx > 0 && needle[idx] != haystack[curr]) {
22      idx = prefix_arr[idx - 1];
23    } // while
24    if (needle[idx] == haystack[curr]) {
25      ++idx;
26    } // if
27    if (idx == needle.length()) {
28      matches.push_back(curr - needle.length() + 1);
29    } // if
30  } // for curr
```

Lastly, it should be noted that the auxiliary space used by the KMP algorithm is $\Theta(n)$, where n is the length of the needle. This is because we need to construct a separate prefix table of length n to run the algorithm.

String Search Method	Average-Case Time	Worst-Case Time
Knuth-Morris-Pratt	$\Theta(n+h)$	$\Theta(n+h)$