Chapter 16 Practice Exercises

Disclaimer: These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Consider the following four strings:

```
std::string str1 = "a";
std::string str2 = "ab";
std::string str3 = "ab9";
std::string str4 = "ab10";
```

Which of the following correctly lists these strings in lexicographical order?
A) `str1 < str2 < str3 < str4`
B) `str1 < str2 < str4 < str3`
C) `str4 < str3 < str1 < str2`
D) `str4 < str3 < str2 < str1`
E) None of the above

2. For which of the following would `std::lexicographical_compare()` return false? Assume that the first word is entered as an argument before the second word.
A) Comparing the strings "`Apple`" and "`apple`"
B) Comparing the strings "`eecs280`" and "`eecs281`"
C) Comparing the strings "`lettuce`" and "`lettuce`"
D) More than one of the above
E) None of the above

3. Which of the following statements is/are **TRUE**?
I. If you want to compare the equality of two C strings (`char*`), you should choose `std::equal()` over `operator==`.
II. `std::equal()` can be used to check if a string shares a matching prefix with another string.
III. Only `operator<` and `operator==` are needed to implement the other four comparison operators.

A) I only
B) II only
C) I and II only
D) II and III only
E) I, II, and III

4. In which of the following cases will

```
std::equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2)
```

always return false? Note that `InputIterator1` and `InputIterator2` represent iterators to two different containers.
A) When the range `[first1, last1)` is larger than the container of `first2`
B) When the range `[first1, last1)` is smaller than the container of `first2`
C) When the container of `first1` and the container of `first2` are identical
D) When the container of `first1` and the container of `first2` are *not* identical
E) None of the above (i.e., `std::equal()` can return true for all of the above)

5. Consider the following snippet of code:

```
std::string str = "This pizza is really good!!!";
size_t size_init = str.size();

auto it = std::unique(str.begin(), str.end());
str.resize(it - str.begin());
size_t size_final = str.size();

std::cout << size_init - size_final << std::endl;
```

What does the above code output? *Hint: `std::unique()` returns an iterator that points **one past** the last element that is not removed.*
A) `0`
B) `4`
C) `5`
D) `9`
E) `10`

6. Consider the following snippet of code:

```cpp
constexpr size_t size = 10;
char a1[size] = {'E', 'E', 'C', 'S', '3', '7', '\0', 'F', '2', '1'};
char a2[size];
for (size_t i = 0; i < size; ++i) {
  a2[i] = a1[size - i - 1];
} // for i
std::cout << a2 << std::endl;
```

What does the above code output?
A) `EECS37`
B) `73SCEE`
C) `F21`
D) `12F`
E) `Segmentation fault`

7. Consider the following mystery function:

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

For which of the following strings would this function return true?
A) `abcdef`
B) `tartar`
C) `kayak`
D) `banana`
E) More than one of the above

8. If you are searching for a string of length n within a string of length h using brute force string searching, the average-case time complexity would be ___________ and the worst-case time complexity would be ___________.
A) $\Theta(n)$, $\Theta(h)$
B) $\Theta(n)$, $\Theta(n+h)$
C) $\Theta(n)$, $\Theta(n^2)$
D) $\Theta(h)$, $\Theta(h^2)$
E) $\Theta(h)$, $\Theta(nh)$

9. If you are searching for a string of length n within a string of length h using Rabin-Karp string searching, the worst-case time complexity would be ___________.
A) $\Theta(1)$
B) $\Theta(n)$
C) $\Theta(h)$
D) $\Theta(n+h)$
E) $\Theta(nh)$

10. What is the worst-case time complexity of calling `std::sort()` on a `std::vector<>` of n strings, each of length m characters?
A) $\Theta(m\log(n))$
B) $\Theta(n\log(n))$
C) $\Theta(mn\log(n))$
D) $\Theta(m^2n\log(n))$
E) $\Theta(mn^2)$

11. Which of the following is the best explanation as to why string fingerprinting is useful?
A) It reduces the cost of comparing strings by only checking the first character of each string rather than the entire string
B) It reduces the cost of comparing strings by traversing through only one string instead of both strings during a comparison
C) It reduces the cost of comparing strings by assigning a unique integer that can be used to distinctively identify different strings
D) It reduces the cost of comparing strings by performing a $\Theta(\log(n))$ binary search instead of a $\Theta(n)$ linear search on each string
E) It reduces the cost of comparing strings by mapping strings to an integer identifier that can be more efficiently compared

12. You are given the Rabin-Karp fingerprints of two non-empty strings. The original strings are unknown, but their fingerprints are the same. Which of the following statements are guaranteed to be true?
I. The strings are the same.
II. The strings contain the same characters.
III. The strings have the same length.
IV. The strings start with the same letter.

A) III only
B) IV only
C) II and III only
D) I, II, III, and IV
E) None of the above

13. You are given the Rabin-Karp fingerprints of two non-empty strings. The original strings are unknown, but their fingerprints are different. Under which of the following conditions could it be possible for these two strings to be identical?
    A) The two strings are palindromes (i.e., spelled the same forward and backward)
    B) The length of each string is a prime number
    C) The number chosen as the fingerprint modulus is a prime number
    D) The number chosen as the fingerprint modulus is a power of two
    E) None of the above

14. Given a sorted array of n strings, each of length m, what is the worst-case time complexity of finding whether a given string of length m exists in the array, if you use the most efficient algorithm?
    A) $\Theta(\log(n))$
    B) $\Theta(n)$
    C) $\Theta(n\log(m))$
    D) $\Theta(m\log(n))$
    E) $\Theta(nm)$

15. You are given a string S of length n, and you want to return the longest repeated substring in S. A repeated substring is a substring that occurs more than once in a string, and occurrences of a repeated substring may overlap. If you *already know* the length of the longest repeated substring in S, what is the *average-case* time complexity of returning a longest repeated substring, if you use the most efficient algorithm? For example, given the string $S =$ "soonoonoontm", you would return "oonoon", since it is the longest substring that occurs more than once in S: "s**oonoon**oontm" and "soon**oonoon**tm".
    A) $\Theta(n)$
    B) $\Theta(n\log(n))$
    C) $\Theta(n\log^2(n))$
    D) $\Theta(n^2)$
    E) $\Theta(n^2\log(n))$

16. You are given two identical strings of length n, except that one string is in the form of a C string (`const char*`) while the other is in the form of a `std::string`. What are the time complexities of finding the length of the two strings, provided that you use `strlen()` on the C string and `.size()` on the `std::string`?
    A) C string: $\Theta(1)$, `std::string`: $\Theta(1)$
    B) C string: $\Theta(1)$, `std::string`: $\Theta(n)$
    C) C string: $\Theta(n)$, `std::string`: $\Theta(1)$
    D) C string: $\Theta(n)$, `std::string`: $\Theta(n)$
    E) None of the above

17. Consider the following possible implementation of `std::lexicographical_compare()` in the standard algorithm library:

```
1  template <class InputIterator1, class InputIterator2>
2  bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
3                               InputIterator2 first2, InputIterator2 last2) {
4    for (; (first1 != last1) && (first2 != last2); ++first1, (void) ++first2) {
5      if (__I__)
6        return true;
7      if (__J__)
8        return false;
9    } // for
10
11   return (first1 == last1) && (first2 != last2);
12 } // lexicographical_compare()
```

Which of the following could be correctly inserted into the missing condition statements `I` and `J` on lines 5 and 7?
    A) `I: *first1 < *first2`
    `J: *first1 == *first2`
    B) `I: *first1 == *first2`
    `J: *first1 < *first2`
    C) `I: *first1 > *first2`
    `J: *first2 > *first1`
    D) `I: *first1 < *first2`
    `J: *first2 < *first1`
    E) None of the above

18. You are given two strings, as well as a fingerprint function that can be used to convert a string into an integer in constant time.

```
int32_t get_string_fingerprint(const std::string& str);
```

Your friend implements the following function, which they claim is able to check if two strings are equal in constant time.

```
bool are_strings_equal(const std::string& s1, const std::string& s2) {
  int32_t fingerprint1 = get_string_fingerprint(s1);
  int32_t fingerprint2 = get_string_fingerprint(s2);
  if (fingerprint1 == fingerprint2) {
    return true;
  } // if
  return false;
} // are_strings_equal()
```

Is your friend's implementation guaranteed to work as intended? Why or why not?

19. Implement a function that, given a string `str`, returns the longest prefix of the string that is also a suffix (excluding the entire word itself). If no such prefix exists, return an empty string.

Example: Given the string `"edited"`, you would return `"ed"`, since that is the longest prefix (`"`**`ed`**`ited"`) that is also a suffix of the same word (`"edit`**`ed`**`"`).

```
std::string longest_prefix_suffix(const std::string& str);
```

You should implement your solution so that the average-case time complexity is better than $\Theta(n^2)$, where n is the length of the input string. (Reducing the worst-case time complexity below $\Theta(n^2)$ is also possible, but not required.)

Chapter 16 Exercise Solutions

1. **The correct answer is (B).** The correct order is `"a"`, then `"ab"`, then `"ab10"`, then `"ab9"` (`"ab10"` comes before `"ab9"` since 9 is larger lexicographically compared to 1).

2. **The correct answer is (C).** Option (A) returns true because capital letters come before lowercase letters lexicographically. Option (B) returns true because the first character difference between the two strings (`"0"` and `"1"`) puts `"eecs280"` before `"eecs281"` lexicographically. Option (C) returns false because `std::lexicographical_compare()` returns false if two strings are identical.

3. **The correct answer is (E).** All three statements are true. `std::equal()` works on C strings while `operator==` does not, and it can also be used to check for a matching prefix (since you can pass in an iterator range representing the prefix you want to search). Statement III is true since only < and == are needed to implement the other four comparison operators:
   - a != b is the same as !(a == b)
   - a > b is the same as b < a
   - a <= b is the same as !(b < a)
   - a >= b is the same as !(a < b)

4. **The correct answer is (A).** If you want to find an input range A within another sequence B, then you would never be able to find A in B if the length of B is smaller than A. Thus, `std::equal()` is guaranteed to return false if the input range you are searching for `[first1, last1)` is larger than the container you are searching in (represented by `first2`).

5. **The correct answer is (C).** The `std::unique()` function eliminates all but the first element from every consecutive group of equivalent elements in the provided range `[first, last)`. Thus, the following five letters of `str` are removed: `"This piz`**`z`**`a is real`**`l`**`y go`**`o`**`d!`**`!!`**`"`.

6. **The correct answer is (D).** The loop on line 4 reverses the contents of `a1` and inserts them into `a2`. Therefore, when the loop completes its last iteration, the contents of `a2` are `{'1', '2', 'F', '\0', '7', '3', 'S', 'C', 'E', 'E'}`. Printing `a2` would read every character up to the first sentinel character `'\0'`, so the output would be `"12F"`.

7. **The correct answer is (C).** The function returns true if the back half of the string is equal to the front half (notice the use of reverse iterators in the input range). Thus, the function would only return true if given a palindrome (a word spelled the same forward and backward), which is only true for option (C).

8. **The correct answer is (E).** With brute force string searching, we take the needle of length n and use it as a "window" to compare with substrings of length n within the haystack. For instance, if we had a needle of length 4 and a haystack of length 20, we would first check if the needle matches the haystack from characters 1-4, then if the needle matches with characters 2-5, then 3-6, then 4-7, etc., until we find a match. In the average case, we would be able to tell quickly if a match is found, as we expect the first letter of the needle and the haystack substring to be different if there is no match. Thus, we would check the first letter of the needle with each letter of the haystack, moving forward if a match is not found. We would only have to do approximately h comparisons while doing this, where h is the length of the haystack, so this gives us an average-case time complexity of $\Theta(h)$. However, in the worst case, we would have to check all n characters in the needle for *every* substring of length n in the haystack. Thus, for the approximately h comparisons we have to do when traversing through the haystack, we would have to do an additional n comparisons in the worst case (e.g., in our previous example, we would have to compare the 4 characters of the needle with *every* haystack character from 1-4 to see if there is a match, then with *every* haystack character from 2-5, then 3-6, then 4-7, etc.). This gives us a worst-case time complexity of $\Theta(nh)$. See section 16.5 for a more in-depth explanation.

9. **The correct answer is (E).** Although improbable, two strings having the same fingerprint does not guarantee that they are equal. As a result, even if you encounter a fingerprint match while searching the string, you would still need to do a string comparison to determine if the two strings are actually equal. In the worst case, every fingerprint is a false positive, which forces you to do a $\Theta(n)$ comparison for all $\Theta(h)$ positions of the haystack. This yields the same worst-case time complexity as brute force, which is $\Theta(nh)$.

10. **The correct answer is (C).** The worst-case time complexity of sorting a vector of n elements is $\Theta(n\log(n))$ if comparisons take $\Theta(1)$ time. In our case, the worst-case time complexity of comparing two strings of length m is $\Theta(m)$. The overall worst-case time complexity occurs when every comparison during the sort takes $\Theta(m)$ time, which yields a result of $\Theta(m \times n\log(n))$.

11. **The correct answer is (E).** String fingerprinting maps strings to an integer comparator that improves the efficiency of string comparisons. For instance, checking if the strings `"aaaaaaaaaa"` and `"aaaaaaaaab"` are identical may be inefficient using the standard method of comparing characters up to the first mismatch. If we map these strings to a unique fingerprint, however, we can simplify this process by checking if the two integers are identical, as integer comparisons are a lot more lightweight. Note that string fingerprints are not guaranteed to be unique, as two different strings can still map to the same integer value.

12. **The correct answer is (E).** If two strings have the same fingerprint, they may be the same, but that is not a guarantee. This is due to the chance that two different strings could still be mapped to the same fingerprint value. You cannot conclude any of the statements provided without actually checking the characters of the two strings for equivalence.

13. **The correct answer is (E).** If two strings have different fingerprints, the contents of the strings themselves *must* also be different, as it is impossible for the same string to map to two different fingerprint values. This holds in all cases, so the provided scenarios are all irrelevant.

14. **The correct answer is (D).** If the array is sorted, you can find whether a string exists or not using a binary search, which involves $\Theta(\log(n))$ comparisons. Each comparison takes $\Theta(m)$ time, so the overall time complexity would be $\Theta(m\log(n))$.

15. **The correct answer is (A).** Since repeated substrings may overlap, Rabin-Karp can be a useful strategy for this problem. To solve this problem efficiently, you can keep track of a rolling fingerprint value for substrings of length k, where k is the length of the longest repeated substring. Slide through the original string, and once you find a fingerprint you have seen before, check the strings for equality and return if the strings are equal. If the fingerprint function is good (and there are few collisions), each window in the string can be checked in average-case $\Theta(1)$ time. Since there are $\Theta(n)$ windows, the average-case time complexity of the entire problem is $\Theta(n)$.

16. **The correct answer is (C).** A `std::string` stores information internally that allows it to calculate the length of its stored string in $\Theta(1)$. This is not true for C strings, which are essentially plain character arrays — to find its length, you would need to iterate over the characters until you encounter the sentinel character, which would take $\Theta(n)$ time for a C string of length n.

17. **The correct answer is (D).** The `std::lexicographical_compare()` function returns true if the contents of the first iterator range compares lexicographically less than the contents of the second iterator range. Therefore, we want to return true if `*first1` is less than `*first2`, and false if `*first1` is greater than `*first2`. This matches option (D).

18. Your friend's implementation is not guaranteed to work in the case where `get_string_fingerprint()` returns the same fingerprint for two different strings. Even if two fingerprints match, the actual contents of the strings should still be compared with each other to ensure that the strings themselves also match.

19. One strategy for solving this problem is to incrementally compare the ends of the input string and keep track of the longest matching prefix/suffix encountered so far. For instance, consider the string `"abcdabc"`. Here, we would perform the following comparisons:

$n = 1$:	"**a**bcdabc"	"abcdab**c**"	✗
$n = 2$:	"**ab**cdabc"	"abcda**bc**"	✗
$n = 3$:	"**abc**dabc"	"abcd**abc**"	✓
$n = 4$:	"**abcd**abc"	"abc**dabc**"	✗
$n = 5$:	"**abcda**bc"	"ab**cdabc**"	✗
$n = 6$:	"**abcdab**c"	"a**bcdabc**"	✗

However, comparing strings can be expensive, and this would give us an $\Theta(n^2)$ algorithm on average. If we recall the rolling hash strategy employed in the Rabin-Karp algorithm, we can actually calculate the fingerprint of the first n characters of a string in $\Theta(1)$ time, provided that we know the fingerprint of the first $n-1$ characters. See the first remark of section 16.6 for why this is the case, but the idea is that we can easily convert our prefixes and suffixes into integers, which are more efficient to compare than the contents of the strings themselves. Given a string of length n with characters that have ASCII values $c_1, c_2, \ldots, c_n$ and a base multiplier of k, a string's fingerprint can be calculated using the following equation:

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

If we want to increase the length of our prefix by one character to the right, we would just need to multiply our existing prefix fingerprint by the base multiplier k and add the value of the new character (e.g., for a string of length 2, the fingerprint of the prefix of length 2 is $k^1c_1 + k^0c_2$, the fingerprint of the prefix of length 3 is $k^2c_1 + k^1c_2 + k^0c_3$, and $(k^1c_1 + k^0c_2) * k + c_3 = k^2c_1 + k^1c_2 + k^0c_3$ for $n = 2 \rightarrow 3$). On the other hand, if we want to increase the length of our suffix by one character to the left, we would just need to multiply the new character by k^m and add it to the existing suffix fingerprint value, where m is the distance from the last character (e.g., for a string of length 3, the fingerprint of the suffix of length 2 is $k^1c_2 + k^0c_3$, the fingerprint of the suffix of length 3 is $k^2c_1 + k^1c_2 + k^0c_3$, and $(k^1c_2 + k^0c_3) + k^2c_1 = k^2c_1 + k^1c_2 + k^0c_3$ for $n = 2 \rightarrow 3$). An implementation of this solution is shown below; note that a prefix and suffix sharing the same hash value does not imply that they are necessarily equal (due to the potential of two differing strings mapping to the same integer value via a hash collision), so a string comparison must still be made if the fingerprints of a prefix and suffix match.

```
std::string longest_prefix_string(const std::string& str) {
  constexpr int32_t base = 128;         // this is k, or our multiplier (can be different)
  constexpr int64_t prime = 281280281;  // large prime to be used as modulus (can be different)

  int64_t prefix_hash = 0, suffix_hash = 0, suffix_multiplier = 1;
  int32_t best_prefix_length = 0;  // length of best match encountered
  for (size_t i = 0; i < str.length() - 1; ++i) {
    prefix_hash = (base * prefix_hash + str[i]) % prime;
    suffix_hash = (suffix_hash + suffix_multiplier * str[str.length() - i - 1]) % prime;
    suffix_multiplier = suffix_multiplier * base % prime;
    if (prefix_hash == suffix_hash && str.substr(0, i + 1) == str.substr(str.length() - i - 1)) {
      best_prefix_length = i + 1;
    } // if
  } // for

  return str.substr(0, best_prefix_length);
} // longest_prefix_string()
```

Not required for this problem:

Since string comparisons cannot be avoided even in the case where two fingerprints match, the worst-case time complexity of this rolling hash solution is still $\Theta(n^2)$. If we want to drop this worst-case complexity down, we can instead use the KMP approach, which allows us to obtain a linear time solution. Notice that the prefix table that needs to be constructed for KMP solves the same prefix problem: 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. Thus, the solution for this problem can be obtained by simply constructing the prefix table and returning the prefix of that length. An implementation is shown below (this is the same implementation as the prefix table code explained in section 16.7 — refer to that section for an explanation of how this works):

```
std::string longest_prefix_string(const std::string& str) {
  std::vector<size_t> prefix_arr(str.length());
  size_t j = 0;
  for (size_t i = 1; i < str.length(); ++i) {
    while (j > 0 && str[j] != str[i]) {
      j = prefix_arr[j - 1];
    } // while
    if (str[j] == str[i]) {
      ++j;
    } // if
    prefix_arr[i] = j;
  } // for i

  return str.substr(0, j);
} // longest_prefix_string()
```