CSE 3150 — C++ Essentials — Yufeng Wu — Fall 2024
Programming Assignment 3. Due: 11:59 pm @ 09/20/2024.

These are the objectives in this assignments.

1. Learn how to work with C++ strings.
2. Continue working on basic programming logic.
3. Learn how to implement given algorithms.
4. Practice on debugging your code.

String matching is one of the most common functions about strings and texts. A common setting is as follows. You are given a (potentially very long) string T called text, and another (often shorter) string P called pattern. You want to determine whether P is a substring of T. For example, let $T = mississippi$, and $P = iss$. Then the answer is yes: P is a substring of P. However, if $P = ssp$, then the answer is no: ssp is not a substring of $mississippi$. String matching is a classic algorithmic problem and there are tons of string matching algorithms. For more background on string matching algorithms, I would refer you to the Wikipedia page: `https://en.wikipedia.org/wiki/String-searching_algorithm`. In this assignment, you are to deal with the following three approaches for performing string matching for C++ std::string.

1. Brute-force comparison: this is called the "naive string search" in the above Wikipedia page. Basically, you are to check at each position of T, one by one, whether P starts at this position or not.
2. C++ built-in string matching: C++ string class provides a function called "find" which does exactly the string matching task. You can simply just invoke this implemented function directly. It is so easy, isn't it?
3. A numeric-based comparison algorithm. This algorithm is somewhat more complex. However, I hope after trying it yourself, you will agree that this algorithm can be a lot faster than other methods, including the one provided by C++! The algorithm is a little involved. So I am going to describe it in more depth in the following.

1 The numeric-based comparison

The basic idea is viewing P and all substrings of T that are the same length as P as **numbers**. To see what this means, we first suppose P consists of digits from 0 to 9. For example, $P = 1211$ and $T = 1235211215$. Here, it is obvious that P can be viewed as a decimal number. Now we consider each position i of T, and view the four (the length of P) digits starting from this position as a number T_i. For example, for the first position $i = 1$, $T_i = 1235$; $T_2 = 2352$; $T_6 = 1121$ and so on. Then we can decide whether P is a substring starting from this position by directly comparing the two numbers P and T_i are the same number. When the pattern is not too long, comparing two integers is very easy in C++: we can simply perform comparison between two C++ integers.

Of course, things are not always this easy. The first question is, what if the strings are not in digits but rather some general texts like $P = mississppi$? This is actually not that hard: recall that each character of a C++ string is encoded as a ASCII number. That is, a string can be viewed as a number consisting of a sequence of digits, where each digit is in ASCII code. The more difficult problem is about the size of the numbers corresponding to a longer string. C++ integers have fairly small range: int is of 4 bytes. Therefore, often treating strings as integer

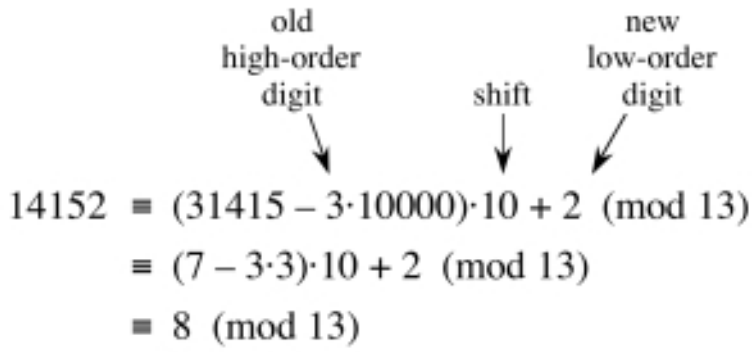

Figure 1: Construing numbers efficiently from the ones computed before. Here the numbers are decimal. We use $q = 13$ for the range of hash (for modulo).

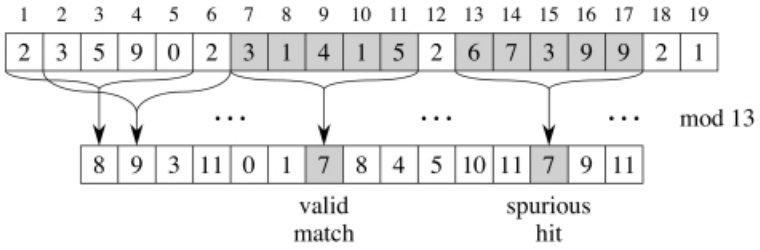

Figure 2: Hash collision can lead to "spurious" match.

directly will lead to numerical overflow. How can we deal with this problem? The idea is imposing a range on the integers by imposing a "modulo" operation on all the numbers we are to consider. The modulo would be taken against a user-specified number q. Doing modulo leads to another problem: hash collision. By doing modulo, we essentially compute a hash value for the strings we are comparing. Recall what you have learned about hashing. When two hash values match, the underlying strings may or may not match. To address this issue, the simple approach is, performing a direct comparison *only* when the hash values of the pattern and the text substring match. This is to ensure there is no hash collision.

There is one more issue. Construing numbers from strings may be slow if you are to do that for each position one by one. In fact, constructing a number can take $O(length(P)$ time, which is no faster than the brute-force string comparison! We will speed up this number construction using a technique that is similar to the well-known "running sum" approach. We consider $P = 31415$ and $T = 2359023141526739921$. Now, $T_7 = 31415$. To construct T_8, we are going to *reuse* part of T_7 by noting that T_7 and T_8 share the middle digits. That is, T_7 and T_8 share 1415. This would allow us to quickly compute the number for T_8 from T_7 by discarding the leading digit (3) of T_1 and appending a new last digit (2). This can be done in constant time. Refer to Figure 1 to see how the approach works.

Figure 2 shows a more complete picture about the issue of hash collision.

The following is the pseudo code for the numeric-based string matching. Here T is the text, P is the pattern, d is the base (radix) of the numbers (for decimal, $d = 10$; for binary, $d = 2$), and q is the range of hash (all numbers will be from 0 to $q - 1$).

```
1  n = T.length
2  m = P.length
3  h = d^(m−1) mod q
4  p = 0
5  t_0 = 0
6  for i = 1 to m                    // preprocessing
7       p = (dp + P[i]) mod q
8       t_0 = (dt_0 + T[i]) mod q
9  for s = 0 to n − m                // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n − m
14          t_{s+1} = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

2 What do you need to implement?

1. Brute-force comparison: you need to implement this very simple algorithm, which is used as the baseline.

2. C++ built-in string matching: I have already provided this function in the starter code, which you can use.

3. A numeric-based comparison algorithm. You need to implement this function. Note that this function takes four parameters: in addition to T and P, it also takes d (the base/radix) and q (the hash value range).

I have provided some test cases. Please note: we will perform running time check to make sure your algorithm actually works as expected.