CHAPTER

6 Strings

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs naturally as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval.

— *"Algorithms For Finding Patterns in Strings,"*
A. V. AHO, 1990

Strings are ubiquitous in programming today—scripting, web development, and bioinformatics all make extensive use of strings.

A string can be viewed as a special kind of array, namely one made out of characters. We treat strings separately from arrays because certain operations which are commonly applied to strings—for example, comparison, joining, splitting, searching for substrings, replacing one string by another, parsing, etc.—do not make sense for general arrays.

You should know how strings are represented in memory, and understand basic operations on strings such as comparison, copying, joining, splitting, matching, etc. Advanced string processing algorithms often use hash tables (Chapter 12) and dynamic programming (Page 234). In this chapter we present problems on strings which can be solved using basic techniques.

Strings boot camp

A palindromic string is one which reads the same when it is reversed. The program below checks whether a string is palindromic. Rather than creating a new string for the reverse of the input string, it traverses the input string forwards and backwards, thereby saving space. Notice how it uniformly handles even and odd length strings.

```
def is_palindromic(s):
    # Note that s[~i] for i in [0, len(s) - 1] is s[-(i + 1)].
    return all(s[i] == s[~i] for i in range(len(s) // 2))
```

The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the length of the string.

Know your string libraries

The key operators and functions on strings are `s[3]`, `len(s)`, `s + t`, `s[2:4]` (and all the other variants of slicing for lists described on Page 39), `s in t`, `s.strip()`, `s.startswith(prefix)`, `s.endswith(suffix)`, `'Euclid,Axiom 5,Parallel Lines'.split(',')`, `3 * '01'`, `','.join(('Gauss', 'Prince of Mathematicians', '1777-1855'))`, `s.tolower()`, and `'Name {name}, Rank {rank}'.format(name='Archimedes', rank=3)`.

It's important to remember that strings are immutable—operations like `s = s[1:]` or `s += '123'` imply creating a new array of characters that is then assigned back to `s`. This implies that concatenating a single character n times to a string in a for loop has $O(n^2)$ time complexity. (Some

Similar to arrays, string problems often have simple brute-force solutions that use $O(n)$ space solution, but subtler solutions that use the string itself to **reduce space complexity** to $O(1)$. Understand the **implications** of a string type which is **immutable**, e.g., the need to allocate a new string when concatenating immutable strings. Know **alternatives** to immutable strings, e.g., a `list` in Python. Updating a mutable string from the front is slow, so see if it's possible to **write values from the back**.

Table 6.1: Top Tips for Strings

implementations of Python use tricks under-the-hood to avoid having to do this allocation, reducing the complexity to $O(n)$.)

6.1 Interconvert strings and integers

A string is a sequence of characters. A string may encode an integer, e.g., "123" encodes 123. In this problem, you are to implement methods that take a string representing an integer and return the corresponding integer, and vice versa. Your code should handle negative integers. You cannot use library functions like `int` in Python.

Implement an integer to string conversion function, and a string to integer conversison function. For example, if the input to the first function is the integer 314, it should return the string "314" and if the input to the second function is the string "314" it should return the integer 314.

Hint: Build the result one digit at a time.

Solution: Let's consider the integer to string problem first. If the number to convert is a single digit, i.e., it is between 0 and 9, the result is easy to compute: it is the string consisting of the single character encoding that digit.

If the number has more than one digit, it is natural to perform the conversion digit-by-digit. The key insight is that for any positive integer x, the least significant digit in the decimal representation of x is $x \mod 10$, and the remaining digits are $x/10$. This approach computes the digits in reverse order, e.g., if we begin with 423, we get 3 and are left with 42 to convert. Then we get 2, and are left with 4 to convert. Finally, we get 4 and there are no digits to convert. The natural algorithm would be to prepend digits to the partial result. However, adding a digit to the beginning of a string is expensive, since all remaining digit have to be moved. A more time efficient approach is to add each computed digit to the end, and then reverse the computed sequence.

If x is negative, we record that, negate x, and then add a '-' before reversing. If x is 0, our code breaks out of the iteration without writing any digits, in which case we need to explicitly set a 0.

To convert from a string to an integer we recall the basic working of a positional number system. A base-10 number $d_2d_1d_0$ encodes the number $10^2 \times d_2 + 10^1 \times d_1 + d_0$. A brute-force algorithm then is to begin with the rightmost digit, and iteratively add $10^i \times d_i$ to a cumulative sum. The efficient way to compute 10^{i+1} is to use the existing value 10^i and multiply that by 10.

A more elegant solution is to begin from the leftmost digit and with each succeeding digit, multiply the partial result by 10 and add that digit. For example, to convert "314" to an integer, we

initial the partial result r to 0. In the first iteration, $r = 3$, in the second iteration $r = 3 \times 10 + 1 = 31$, and in the third iteration $r = 31 \times 10 + 4 = 314$, which is the final result.

Negative numbers are handled by recording the sign and negating the result.

```
def int_to_string(x):
    is_negative = False
    if x < 0:
        x, is_negative = -x, True

    s = []
    while True:
        s.append(chr(ord('0') + x % 10))
        x //= 10
        if x == 0:
            break

    # Adds the negative sign back if is_negative
    return ('-' if is_negative else '') + ''.join(reversed(s))


def string_to_int(s):
    return functools.reduce(
        lambda running_sum, c: running_sum * 10 + string.digits.index(c),
        s[s[0] == '-':], 0) * (-1 if s[0] == '-' else 1)
```

6.2 Base conversion

In the decimal number system, the position of a digit is used to signify the power of 10 that digit is to be multiplied with. For example, "314" denotes the number $3 \times 100 + 1 \times 10 + 4 \times 1$. The base b number system generalizes the decimal number system: the string "$a_{k-1}a_{k-2}\ldots a_1a_0$", where $0 \leq a_i < b$, denotes in base-b the integer $a_0 \times b^0 + a_1 \times b^1 + a_2 \times b^2 + \cdots + a_{k-1} \times b^{k-1}$.

Write a program that performs base conversion. The input is a string, an integer b_1, and another integer b_2. The string represents an integer in base b_1. The output should be the string representing the integer in base b_2. Assume $2 \leq b_1, b_2 \leq 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15. (For example, if the string is "615", b_1 is 7 and b_2 is 13, then the result should be "1A7", since $6 \times 7^2 + 1 \times 7 + 5 = 1 \times 13^2 + 10 \times 13 + 7$.)

Hint: What base can you easily convert to and from?

Solution: A brute-force approach might be to convert the input to a unary representation, and then group the 1s as multiples of b_2, b_2^2, b_2^3, etc. For example, $(102)_3 = (11111111111)_1$. To convert to base 4, there are two groups of 4 and with three 1s remaining, so the result is $(23)_4$. This approach is hard to implement, and has terrible time and space complexity.

The insight to a good algorithm is the fact that all languages have an integer type, which supports arithmetical operations like multiply, add, divide, modulus, etc. These operations make the conversion much easier. Specifically, we can convert a string in base b_1 to integer type using a sequence of multiply and adds. Then we convert that integer type to a string in base b_2 using a sequence of modulus and division operations. For example, for the string is "615", $b_1 = 7$ and $b_2 = 13$, then the integer value, expressed in decimal, is 306. The least significant digit of the result is 306 mod 13 = 7, and we continue with 306/13 = 23. The next digit is 23 mod 13 = 10, which we

denote by 'A'. We continue with 23/13 = 1. Since 1 mod 13 = 1 and 1/13 = 0, the final digit is 1, and the overall result is "1A7". The design of the algorithm nicely illustrates the use of reduction.

Since the conversion algorithm is naturally expressed in terms of smaller similar subproblems, it is natural to implement it using recursion.

```
def convert_base(num_as_string, b1, b2):
    def construct_from_base(num_as_int, base):
        return ('' if num_as_int == 0 else
                construct_from_base(num_as_int // base, base) +
                string.hexdigits[num_as_int % base].upper())

    is_negative = num_as_string[0] == '-'
    num_as_int = functools.reduce(
        lambda x, c: x * b1 + string.hexdigits.index(c.lower()),
        num_as_string[is_negative:], 0)
    return ('-' if is_negative else '') + ('0' if num_as_int == 0 else
                                           construct_from_base(num_as_int, b2))
```

The time complexity is $O(n(1 + \log_{b_2} b_1))$, where n is the length of s. The reasoning is as follows. First, we perform n multiply-and-adds to get x from s. Then we perform $\log_{b_2} x$ multiply and adds to get the result. The value x is upper-bounded by b_1^n, and $\log_{b_2}(b_1^n) = n \log_{b_2} b_1$.

6.3 Compute the spreadsheet column encoding

Spreadsheets often use an alphabetical encoding of the successive columns. Specifically, columns are identified by "A", "B", "C", ..., "X", "Y", "Z", "AA", "AB", ..., "ZZ", "AAA", "AAB", ...

Implement a function that converts a spreadsheet column id to the corresponding integer, with "A" corresponding to 1. For example, you should return 4 for "D", 27 for "AA", 702 for "ZZ", etc. How would you test your code?

Hint: There are 26 characters in ["A", "Z"], and each can be mapped to an integer.

Solution: A brute-force approach could be to enumerate the column ids, stopping when the id equals the input. The logic for getting the successor of "Z", "AZ", etc. is slightly involved. The bigger issue is the time-complexity—it takes 26^6 steps to get to "ZZZZZZ". In general, the time complexity is $O(26^n)$, where n is the length of the string.

We can do better by taking larger jumps. Specifically, this problem is basically the problem of converting a string representing a base-26 number to the corresponding integer, except that "A" corresponds to 1 not 0. We can use the string to integer conversion approach given in Solution 6.1 on Page 68.

For example to convert "ZZ", we initialize result to 0. We add 26, multiply by 26, then add 26 again, i.e., the id is $26^2 + 26 = 702$.

Good test cases are around boundaries, e.g., "A", "B", "Y", "Z", "AA", "AB", "ZY", "ZZ", and some random strings, e.g., "M", "BZ", "CCC".

```
def ss_decode_col_id(col):
    return functools.reduce(
        lambda result, c: result * 26 + ord(c) - ord('A') + 1, col, 0)
```

The time complexity is $O(n)$.

Variant: Solve the same problem with "A" corresponding to 0.

Variant: Implement a function that converts an integer to the spreadsheet column id. For example, you should return "D" for 4, "AA" for 27, and "ZZ" for 702.

6.4 Replace and remove

Consider the following two rules that are to be applied to an array of characters.

- Replace each 'a' by two 'd's.
- Delete each entry containing a 'b'.

For example, applying these rules to the array $\langle a, c, d, b, b, c, a \rangle$ results in the array $\langle d, d, c, d, c, d, d \rangle$.

Write a program which takes as input an array of characters, and removes each 'b' and replaces each 'a' by two 'd's. Specifically, along with the array, you are provided an integer-valued size. Size denotes the number of entries of the array that the operation is to be applied to. You do not have to worry about preserving subsequent entries. For example, if the array is $\langle a, b, a, c, \_ \rangle$ and the size is 4, then you can return $\langle d, d, d, d, c \rangle$. You can assume there is enough space in the array to hold the final result.

Hint: Consider performing multiples passes on s.

Solution: Library array implementations often have methods for inserting into a specific location (all later entries are shifted right, and the array is resized) and deleting from a specific location (all later entries are shifted left, and the size of the array is decremented). If the input array had such methods, we could apply them—however, the time complexity would be $O(n^2)$, where n is the array's length. The reason is that each insertion and deletion from the array would have $O(n)$ time complexity.

This problem is trivial to solve in $O(n)$ time if we write result to a new array—we skip 'b's, replace 'a's by two 'd's, and copy over all other characters. However, this entails $O(n)$ additional space.

If there are no 'a's, we can implement the function without allocating additional space with one forward iteration by skipping 'b's and copying over the other characters.

If there are no 'b's, we can implement the function without additional space as follows. First, we compute the final length of the resulting string, which is the length of the array plus the number of 'a's. We can then write the result, character by character, starting from the last character, working our way backwards.

For example, suppose the array is $\langle a, c, a, a, \_, \_, \_ \rangle$, and the specified size is 4. Our algorithm updates the array to $\langle a, c, a, a, \_, \mathbf{d}, \mathbf{d} \rangle$. (Boldface denotes characters that are part of the final result.) The next update is $\langle a, c, a, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d} \rangle$, followed by $\langle a, c, \mathbf{c}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d} \rangle$, and finally $\langle \mathbf{d}, \mathbf{d}, \mathbf{c}, \mathbf{d}, \mathbf{d}, \mathbf{d}, \mathbf{d} \rangle$.

We can combine these two approaches to get a complete algorithm. First, we delete 'b's and compute the final number of valid characters of the string, with a forward iteration through the string. Then we replace each 'a' by two 'd's, iterating backwards from the end of the resulting string. If there are more 'b's than 'a's, the number of valid entries will decrease, and if there are more 'a's than 'b's the number will increase. In the program below we return the number of valid entries in the final result.

```
def replace_and_remove(size, s):
    # Forward iteration: remove 'b's and count the number of 'a's.
```

```
    write_idx, a_count = 0, 0
    for i in range(size):
        if s[i] != 'b':
            s[write_idx] = s[i]
            write_idx += 1
        if s[i] == 'a':
            a_count += 1

    # Backward iteration: replace 'a's with 'dd's starting from the end.
    cur_idx = write_idx - 1
    write_idx += a_count - 1
    final_size = write_idx + 1
    while cur_idx >= 0:
        if s[cur_idx] == 'a':
            s[write_idx - 1:write_idx + 1] = 'dd'
            write_idx -= 2
        else:
            s[write_idx] = s[cur_idx]
            write_idx -= 1
        cur_idx -= 1
    return final_size
```

The forward and backward iterations each take $O(n)$ time, so the total time complexity is $O(n)$. No additional space is allocated.

Variant: You have an array C of characters. The characters may be letters, digits, blanks, and punctuation. The telex-encoding of the array C is an array T of characters in which letters, digits, and blanks appear as before, but punctuation marks are spelled out. For example, telex-encoding entails replacing the character "." by the string "DOT", the character "," by "COMMA", the character "?" by "QUESTION MARK", and the character "!" by "EXCLAMATION MARK". Design an algorithm to perform telex-encoding with $O(1)$ space.

Variant: Write a program which merges two sorted arrays of integers, A and B. Specifically, the final result should be a sorted array of length $m + n$, where n and m are the lengths of A and B, respectively. Use $O(1)$ additional storage—assume the result is stored in A, which has sufficient space. These arrays are C-style arrays, i.e., contiguous preallocated blocks of memory.

6.5 Test palindromicity

For the purpose of this problem, define a palindromic string to be a string which when all the nonalphanumeric are removed it reads the same front to back ignoring case. For example, "A man, a plan, a canal, Panama." and "Able was I, ere I saw Elba!" are palindromic, but "Ray a Ray" is not.

Implement a function which takes as input a string s and returns true if s is a palindromic string.

Hint: Use two indices.

Solution: The naive approach is to create a reversed version of s, and compare it with s, skipping nonalphanumeric characters. This requires additional space proportional to the length of s.

We do not need to create the reverse—rather, we can get the effect of the reverse of s by traversing s from right to left. Specifically, we use two indices to traverse the string, one forwards, the other backwards, skipping nonalphanumeric characters, performing case-insensitive comparison on the

alphanumeric characters. We return false as soon as there is a mismatch. If the indices cross, we have verified palindromicity.

```
def is_palindrome(s):
    # i moves forward, and j moves backward.
    i, j = 0, len(s) - 1
    while i < j:
        # i and j both skip non-alphanumeric characters.
        while not s[i].isalnum() and i < j:
            i += 1
        while not s[j].isalnum() and i < j:
            j -= 1
        if s[i].lower() != s[j].lower():
            return False
        i, j = i + 1, j - 1
    return True
```

We spend $O(1)$ per character, so the time complexity is $O(n)$, where n is the length of s.

6.6 Reverse all the words in a sentence

Given a string containing a set of words separated by whitespace, we would like to transform it to a string in which the words appear in the reverse order. For example, "Alice likes Bob" transforms to "Bob likes Alice". We do not need to keep the original string.

Implement a function for reversing the words in a string s.

Hint: It's difficult to solve this with one pass.

Solution: The code for computing the position for each character in the final result in a single pass is intricate.

However, for the special case where each word is a single character, the desired result is simply the reverse of s.

For the general case, reversing s gets the words to their correct relative positions. However, for words that are longer than one character, their letters appear in reverse order. This situation can be corrected by reversing the individual words.

For example, "ram is costly" reversed yields "yltsoc si mar". We obtain the final result by reversing each word to obtain "costly is ram".

```
# Assume s is a string encoded as bytearray.
def reverse_words(s):
    # First, reverse the whole string.
    s.reverse()

    def reverse_range(s, start, end):
        while start < end:
            s[start], s[end] = s[end], s[start]
            start, end = start + 1, end - 1

    start = 0
    while True:
        end = s.find(b' ', start)
        if end < 0:
            break
```

```
            # Reverses each word in the string.
            reverse_range(s, start, end - 1)
            start = end + 1
        # Reverses the last word.
        reverse_range(s, start, len(s) - 1)
```

Since we spend $O(1)$ per character, the time complexity is $O(n)$, where n is the length of s. The computation in place, i.e., the additional space complexity is $O(1)$.

6.7 Compute all mnemonics for a phone number

Each digit, apart from 0 and 1, in a phone keypad corresponds to one of three or four letters of the alphabet, as shown in Figure 6.1. Since words are easier to remember than numbers, it is natural to ask if a 7 or 10-digit phone number can be represented by a word. For example, "2276696" corresponds to "ACRONYM" as well as "ABPOMZN".

Figure 6.1: Phone keypad.

Write a program which takes as input a phone number, specified as a string of digits, and returns all possible character sequences that correspond to the phone number. The cell phone keypad is specified by a mapping that takes a digit and returns the corresponding set of characters. The character sequences do not have to be legal words or phrases.

Hint: Use recursion.

Solution: For a 7 digit phone number, the brute-force approach is to form 7 ranges of characters, one for each digit. For example, if the number is "2276696" then the ranges are 'A'–'C', 'A'–'C', 'P'–'S', 'M'–'O', 'M'–'O', 'W'–'Z', and 'M'–'O'. We use 7 nested for-loops where the iteration variables correspond to the 7 ranges to enumerate all possible mnemonics. The drawbacks of such an approach are its repetitiveness in code and its inflexibility.

As a general rule, any such enumeration is best computed using recursion. The execution path is very similar to that of the brute-force approach, but the compiler handles the looping.

```
# The mapping from digit to corresponding characters.
MAPPING = ('0', '1', 'ABC', 'DEF', 'GHI', 'JKL', 'MNO', 'PQRS', 'TUV', 'WXYZ')


def phone_mnemonic(phone_number):
    def phone_mnemonic_helper(digit):
        if digit == len(phone_number):
```

```
            # All digits are processed, so add partial_mnemonic to mnemonics.
            # (We add a copy since subsequent calls modify partial_mnemonic.)
            mnemonics.append(''.join(partial_mnemonic))
        else:
            # Try all possible characters for this digit.
            for c in MAPPING[int(phone_number[digit])]:
                partial_mnemonic[digit] = c
                phone_mnemonic_helper(digit + 1)

    mnemonics, partial_mnemonic = [], [0] * len(phone_number)
    phone_mnemonic_helper(0)
    return mnemonics
```

Since there are no more than 4 possible characters for each digit, the number of recursive calls, $T(n)$, satisfies $T(n) \leq 4T(n-1)$, where n is the number of digits in the number. This solves to $T(n) = O(4^n)$. For the function calls that entail recursion, the time spent within the function, not including the recursive calls, is $O(1)$. Each base case entails making a copy of a string and adding it to the result. Since each such string has length n, each base case takes time $O(n)$. Therefore, the time complexity is $O(4^n n)$.

Variant: Solve the same problem without using recursion.

6.8 The look-and-say problem

The look-and-say sequence starts with 1. Subsequent numbers are derived by describing the previous number in terms of consecutive digits. Specifically, to generate an entry of the sequence from the previous entry, read off the digits of the previous entry, counting the number of digits in groups of the same digit. For example, 1; one 1; two 1s; one 2 then one 1; one 1, then one 2, then two 1s; three 1s, then two 2s, then one 1. The first eight numbers in the look-and-say sequence are $\langle 1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211 \rangle$.

Write a program that takes as input an integer n and returns the nth integer in the look-and-say sequence. Return the result as a string.

Hint: You need to return the result as a string.

Solution: We compute the nth number by iteratively applying this rule $n-1$ times. Since we are counting digits, it is natural to use strings to represent the integers in the sequence. Specifically, going from the ith number to the $(i+1)$th number entails scanning the digits from most significant to least significant, counting the number of consecutive equal digits, and writing these counts.

```
def look_and_say(n):
    def next_number(s):
        result, i = [], 0
        while i < len(s):
            count = 1
            while i + 1 < len(s) and s[i] == s[i + 1]:
                i += 1
                count += 1
            result.append(str(count) + s[i])
            i += 1
        return ''.join(result)
```

```
    s = '1'
    for _ in range(1, n):
        s = next_number(s)
    return s


# Pythonic solution uses the power of itertools.groupby().
def look_and_say_pythonic(n):
    s = '1'
    for _ in range(n - 1):
        s = ''.join(
            str(len(list(group))) + key for key, group in itertools.groupby(s))
    return s
```

The precise time complexity is a function of the lengths of the terms, which is extremely hard to analyze. Each successive number can have at most twice as many digits as the previous number—this happens when all digits are different. This means the maximum length number has length no more than 2^n. Since there are n iterations and the work in each iteration is proportional to the length of the number computed in the iteration, a simple bound on the time complexity is $O(n2^n)$.

6.9 Convert from Roman to decimal

The Roman numeral representation of positive integers uses the symbols I, V, X, L, C, D, M. Each symbol represents a value, with I being 1, V being 5, X being 10, L being 50, C being 100, D being 500, and M being 1000.

In this problem we give simplified rules for representing numbers in this system. Specifically, define a string over the Roman number symbols to be a valid Roman number string if symbols appear in nonincreasing order, with the following exceptions allowed:

- I can immediately precede V and X.
- X can immediately precede L and C.
- C can immediately precede D and M.

Back-to-back exceptions are not allowed, e.g., IXC is invalid, as is CDM.

A valid complex Roman number string represents the integer which is the sum of the symbols that do not correspond to exceptions; for the exceptions, add the difference of the larger symbol and the smaller symbol.

For example, the strings "XXXXXIIIIIIIII", "LVIIII" and "LIX" are valid Roman number strings representing 59. The shortest valid complex Roman number string corresponding to the integer 59 is "LIX".

Write a program which takes as input a valid Roman number string s and returns the integer it corresponds to.

Hint: Start by solving the problem assuming no exception cases.

Solution: The brute-force approach is to scan s from left to right, adding the value for the corresponding symbol unless the symbol subsequent to the one being considered has a higher value, in which case the pair is one of the six exception cases and the value of the pair is added.

A slightly easier-to-code solution is to start from the right, and if the symbol after the current one is greater than it, we subtract the current symbol. The code below performs the right-to-left

iteration. It does not check that when a smaller symbol appears to the left of a larger one that it is one of the six allowed exceptions, so it will, for example, return 99 for "IC".

```
def roman_to_integer(s):
    T = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}

    return functools.reduce(
        lambda val, i: val + (-T[s[i]] if T[s[i]] < T[s[i + 1]] else T[s[i]]),
        reversed(range(len(s) - 1)), T[s[-1]])
```

Each character of s is processed in $O(1)$ time, yielding an $O(n)$ overall time complexity, where n is the length of s.

Variant: Write a program that takes as input a string of Roman number symbols and checks whether that string is valid.

Variant: Write a program that takes as input a positive integer n and returns a shortest valid simple Roman number string representing n.

6.10 Compute all valid IP addresses

A decimal string is a string consisting of digits between 0 and 9. Internet Protocol (IP) addresses can be written as four decimal strings separated by periods, e.g., 192.168.1.201. A careless programmer mangles a string representing an IP address in such a way that all the periods vanish.

Write a program that determines where to add periods to a decimal string so that the resulting string is a valid IP address. There may be more than one valid IP address corresponding to a string, in which case you should print all possibilities.

For example, if the mangled string is "19216811" then two corresponding IP addresses are 192.168.1.1 and 19.216.81.1. (There are seven other possible IP addresses for this string.)

Hint: Use nested loops.

Solution: There are three periods in a valid IP address, so we can enumerate all possible placements of these periods, and check whether all four corresponding substrings are between 0 and 255. We can reduce the number of placements considered by spacing the periods 1 to 3 characters apart. We can also prune by stopping as soon as a substring is not valid.

For example, if the string is "19216811", we could put the first period after "1", "19", and "192". If the first part is "1", the second part could be "9", "92", and "921". Of these, "921" is illegal so we do not continue with it.

```
def get_valid_ip_address(s):
    def is_valid_part(s):
        # '00', '000', '01', etc. are not valid, but '0' is valid.
        return len(s) == 1 or (s[0] != '0' and int(s) <= 255)

    result, parts = [], [None] * 4
    for i in range(1, min(4, len(s))):
        parts[0] = s[:i]
        if is_valid_part(parts[0]):
            for j in range(1, min(len(s) - i, 4)):
                parts[1] = s[i:i + j]
                if is_valid_part(parts[1]):
```

```
                for k in range(1, min(len(s) - i - j, 4)):
                    parts[2], parts[3] = s[i + j:i + j + k], s[i + j + k:]
                    if is_valid_part(parts[2]) and is_valid_part(parts[3]):
                        result.append('.'.join(parts))
    return result
```

The total number of IP addresses is a constant (2^{32}), implying an $O(1)$ time complexity for the above algorithm.

Variant: Solve the analogous problem when the number of periods is a parameter k and the string length is unbounded.

6.11 Write a string sinusoidally

We illustrate what it means to write a string in sinusoidal fashion by means of an example. The string "Hello␣World!" written in sinusoidal fashion is

```
                e       ␣       l
  H     l     o     W     r     d
              l       o       !
```

(Here ␣ denotes a blank.)

Define the snakestring of s to be the left-right top-to-bottom sequence in which characters appear when s is written in sinusoidal fashion. For example, the snakestring string for "Hello␣World!" is "e␣lHloWrdlo!".

Write a program which takes as input a string s and returns the snakestring of s.

Hint: Try concrete examples, and look for periodicity.

Solution: The brute-force approach is to populate a $3 \times n$ $2D$ array of characters, initialized to null entries. We then write the string in sinusoidal manner in this array. Finally, we read out the non-null characters in row-major manner.

However, observe that the result begins with the characters $s[1], s[5], s[9], \ldots$, followed by $s[0], s[2], s[4], \ldots$, and then $s[3], s[7], s[11], \ldots$. Therefore, we can create the snakestring directly, with three iterations through s.

```
def snake_string(s):
    result = []
    # Outputs the first row, i.e., s[1], s[5], s[9], ...
    for i in range(1, len(s), 4):
        result.append(s[i])
    # Outputs the second row, i.e., s[0], s[2], s[4], ...
    for i in range(0, len(s), 2):
        result.append(s[i])
    # Outputs the third row, i.e., s[3], s[7], s[11], ...
    for i in range(3, len(s), 4):
        result.append(s[i])
    return ''.join(result)


# Python solution uses slicing with right steps.
def snake_string_pythonic(s):
    return s[1::4] + s[::2] + s[3::4]
```

Let n be the length of s. Each of the three iterations takes $O(n)$ time, implying an $O(n)$ time complexity.

6.12 Implement run-length encoding

Run-length encoding (RLE) compression offers a fast way to do efficient on-the-fly compression and decompression of strings. The idea is simple—encode successive repeated characters by the repetition count and the character. For example, the RLE of "aaaabcccaa" is "4a1b3c2a". The decoding of "3e4f2e" returns "eeefffee".

Implement run-length encoding and decoding functions. Assume the string to be encoded consists of letters of the alphabet, with no digits, and the string to be decoded is a valid encoding.

Hint: This is similar to converting between binary and string representations.

Solution: First we consider the decoding function. Every encoded string is a repetition of a string of digits followed by a single character. The string of digits is the decimal representation of a positive integer. To generate the decoded string, we need to convert this sequence of digits into its integer equivalent and then write the character that many times. We do this for each character.

The encoding function requires an integer (the repetition count) to string conversion.

```
def decoding(s):
    count, result = 0, []
    for c in s:
        if c.isdigit():
            count = count * 10 + int(c)
        else:  # c is a letter of alphabet.
            result.append(c * count)  # Appends count copies of c to result.
            count = 0
    return ''.join(result)


def encoding(s):
    result, count = [], 1
    for i in range(1, len(s) + 1):
        if i == len(s) or s[i] != s[i - 1]:
            # Found new character so write the count of previous character.
            result.append(str(count) + s[i - 1])
            count = 1
        else:  # s[i] == s[i - 1].
            count += 1
    return ''.join(result)
```

The time complexity is $O(n)$, where n is the length of the string.

6.13 Find the first occurrence of a substring

A good string search algorithm is fundamental to the performance of many applications. Several clever algorithms have been proposed for string search, each with its own trade-offs. As a result, there is no single perfect answer. If someone asks you this question in an interview, the best way to approach this problem would be to work through one good algorithm in detail and discuss at a high level other algorithms.

Given two strings s (the "search string") and t (the "text"), find the first occurrence of s in t.

Hint: Form a signature from a string.

Solution: The brute-force algorithm uses two nested loops, the first iterates through t, the second tests if s occurs starting at the current index in t. The worst-case complexity is high. If t consists of n 'a's and s is $n/2$ 'a's followed by a 'b', it will perform $n/2$ unsuccessful string compares, each of which entails $n/2 + 1$ character compares, so the brute-force algorithm's time complexity is $O(n^2)$.

Intuitively, the brute-force algorithm is slow because it advances through t one character at a time, and potentially does $O(m)$ computation with each advance, where m is the length of s.

There are three linear time string matching algorithms: KMP, Boyer-Moore, and Rabin-Karp. Of these, Rabin-Karp is by far the simplest to understand and implement.

The Rabin-Karp algorithm is very similar to the brute-force algorithm, but it does not require the second loop. Instead it uses the concept of a "fingerprint". Specifically, let m be the length of s. It computes hash codes of each substring whose length is m—these are the fingerprints. The key to efficiency is using an incremental hash function, such as a function with the property that the hash code of a string is an additive function of each individual character. (Such a hash function is sometimes referred to as a rolling hash.) For such a function, getting the hash code of a sliding window of characters is very fast for each shift.

For example, let the strings consist of letters from $\{A, C, G, T\}$. Suppose t is "GACGCCA" and s is "CGC". Define the code for "A" to be 0, the code for "C" to be 1, etc. Let the hash function be the decimal number formed by the integer codes for the letters. The hash code of s is 121. The hash code of the first three characters of t, "GAC", is 201, so s cannot be the first three characters of t. Continuing, the next substring of t is "ACG", whose hash code can be computed from 201 by subtracting 200, then multiplying by 10, and finally adding 2. This yields 12, so there no match yet. We then reach "CGC" whose hash code, 121, is derived in a similar manner. We are not done yet—there may be a collision. We check explicitly if the substring matches s, which in this case it does.

For the Rabin-Karp algorithm to run in linear time, we need a good hash function, to reduce the likelihood of collisions, which entail potentially time consuming string equality checks.

```
def rabin_karp(t, s):
    if len(s) > len(t):
        return -1  # s is not a substring of t.

    BASE = 26
    # Hash codes for the substring of t and s.
    t_hash = functools.reduce(lambda h, c: h * BASE + ord(c), t[:len(s)], 0)
    s_hash = functools.reduce(lambda h, c: h * BASE + ord(c), s, 0)
    power_s = BASE**max(len(s) - 1, 0)  # BASE^|s-1|.

    for i in range(len(s), len(t)):
        # Checks the two substrings are actually equal or not, to protect
        # against hash collision.
        if t_hash == s_hash and t[i - len(s):i] == s:
            return i - len(s)  # Found a match.

        # Uses rolling hash to compute the hash code.
        t_hash -= ord(t[i - len(s)]) * power_s
        t_hash = t_hash * BASE + ord(t[i])
```

```
    # Tries to match s and t[-len(s):].
    if t_hash == s_hash and t[-len(s):] == s:
        return len(t) - len(s)
    return -1  # s is not a substring of t.
```

For a good hash function, the time complexity is $O(m+n)$, independent of the inputs s and t, where m is the length of s and n is the length of t.