<https://leetcode.com/problems/remove-k-digits/description/>

Given string num representing a non-negative integer num, and an integer k, return *the smallest possible integer after removing* k *digits from* num.

**Example 1:**

Input: num = "1432219", k = 3

Output: "1219"

Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.

**Example 2:**

Input: num = "10200", k = 1

Output: "200"

Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.

**Example 3:**

Input: num = "10", k = 2

Output: "0"

Explanation: Remove all the digits from the number and it is left with nothing which is 0.

**Constraints:**

1 <= k <= num.length <= 10^5

num consists of only digits.

num does not have any leading zeros except for the zero itself.

**Attempt 1: 2023-03-24**

**Solution 1: Monotonic Increasing Stack (30 min)**

**Style 1: Use Stack and StringBuilder together**

class Solution {

    public String removeKdigits(String num, int k) {

        Stack<Character> stack = new Stack<Character>();

        int len = num.length();

        int i = 0;

        while(i < len) {

// Similar handling way happen in L1673 for Monotonic Increasing Stack

// especially for k > 0, the similar condition in L1673 is

// 'k - stack.size() <= len - i - 1'

            while(k > 0 && !stack.isEmpty() && stack.peek() > num.charAt(i)) {

                stack.pop();

                k--;

            }

            stack.push(num.charAt(i));

            i++;

        }

        // Why it requires additional handling of k > 0?

        // Input num="1111", k=3

        // Output "1111"

        // Expected "1"

// Another case requires this additional handling is given number doesn't

// have enough k 'decreasing point', e.g "123456" has NO 'decreasing point'

// at all, since its monoploy increasing from 1 to 6, it requires 3 removes

// in below logic, e.g "123546" has only one 'decreasing point' between 5

// to 4, less than k = 3, it requires 2 removes in below logic.

        while(k > 0) {

            stack.pop();

            k--;

        }

        StringBuilder sb = new StringBuilder();

        while(!stack.isEmpty()) {

            sb.append(stack.pop());

        }

        sb.reverse();

        // Why it requires remove heading 0s?

        // Input num="10200", k=1

        // Output "0200"

        // Expected "200"

        while(sb.length() > 1 && sb.charAt(0) == '0') {

            sb.deleteCharAt(0);

        }

        // Why it requires make up empty string as 0?

        // Input num="10", k=2

        // Output ""

        // Expected "0"

        return sb.toString() == "" ? "0" : sb.toString();

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/remove-k-digits/discuss/88708/Straightforward-Java-Solution-Using-Stack>

class Solution {

    public String removeKdigits(String num, int k) {

        if(num == null || num.length() == 0) {

            return "";

        }

        if(num.length() == k) {

            return "0";

        }

        // Better than Stack<Integer> since no need calculate

        // num.charAt(i) - '0'

        Stack<Character> stack = new Stack<Character>();

        // The given num does not contain any leading zero

        // so we can directly put the 1st char on stack

        stack.push(num.charAt(0));

        int count = 0;

        for(int i = 1; i < num.length(); i++) {

            char curr = num.charAt(i);

            // ??? This way not work ??? Instead remove the initial 0 after all

            // Because second condition as '(stack.size() == 1 && stack.peek() == '0')'

            // should not count into remove operation since 0200 need auto remove

            // leading 0

            //    || (stack.size() == 1 && stack.peek() == '0')) {

            //    stack.pop();

            //    count++;

            // }

            // Whenever meet a digit which is less than the previous digit,

            // discard the previous one

                stack.pop();

                count++;

            }

            stack.push(curr);

        }

        // Handle corner case as num = 112 and k = 1

        while(count < k) {

            stack.pop();

            count++;

        }

        StringBuilder sb = new StringBuilder();

        while(!stack.isEmpty()) {

            sb.insert(0, stack.pop());

        }

        // Remove all the 0 at the head

        while(sb.length() > 1 && sb.charAt(0) == '0') {

            sb.deleteCharAt(0);

        }

        return sb.toString();

    }

}

Just one suggestion though.

Instead of performing a reverse first and then removing the characters at head, you could remove the characters at the tail and then reverse the string.

deleteCharAt is a linear time operation. If you remove a character at the head, the characters will have to be moved forward by one position each time. On the other hand, removing the tail character can be achieved in constant time as there would be no shifting involved! :)

The change can be incorporated as follows:

while(sb.length() > 1 && sb.charAt(sb.length()-1) == '0')

    sb.deleteCharAt(sb.length()-1);

return sb.reverse().toString();

**Style 2: Use StringBuilder only to simulate Stack**

class Solution {

    public String removeKdigits(String num, int k) {

        // Create a StringBuilder to use as a stack to keep track of digits.

        StringBuilder stack = new StringBuilder();

        for(char digit : num.toCharArray()) {

            // While the current digit is smaller than the last digit in the stack

            // and we still have digits to remove (k > 0), remove the last digit.

            while(k > 0 && stack.length() > 0 && stack.charAt(stack.length() - 1) > digit) {

                stack.deleteCharAt(stack.length() - 1);

                k--;

            }

            stack.append(digit);

        }

        // If after the iteration we still need to remove more digits, remove from the end.

        while(k > 0) {

            stack.deleteCharAt(stack.length() - 1);

            k--;

        }

        // Remove leading zeros by finding the index of the first non-zero digit.

        int i = 0;

        while(i < stack.length() && stack.charAt(i) == '0') {

            i++;

        }

        // Create a new string starting from the first non-zero digit.

        String result = stack.substring(i);

        // If the resulting string is empty, return "0" instead; otherwise, return the string.

        return result.isEmpty() ? "0" : result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://algo.monster/liteproblems/402>

**Problem Description**

This LeetCode problem asks you to find the smallest possible integer after removing exactly k digits from a string num that represents a non-negative integer. The goal is to reduce the size of the number while keeping the remaining digits in the same order as they were in the original number.

**Intuition**

The intuition behind the solution is to use a greedy algorithm. If we want the resulting number to be the smallest possible, we should ensure that the higher place values (like tens, hundreds etc.) have the smallest possible digits. **Therefore, while parsing the string from left to right, if we encounter a digit that is larger than the digit following it, we remove the larger digit (which is at a higher place value). This decision is greedy because it makes the best choice at each step, aiming to keep the smallest digits at higher place values.**

To efficiently perform removals and keep track of the digits, a stack is an excellent choice. Each time we add a new digit to the stack, we compare it to the element on top of the stack (which represents the previous digit in the number). If the new digit is smaller, it means we can make the number smaller by popping the larger digit off the stack. This process is repeated up to k times as required by the problem statement.

The stack represents the digits of the resulting number with the smallest digits at the bottom (higher place values). When k removals are done, or the string is fully parsed, we take the bottom n - k digits from the stack (where n is the length of num), since k digits have been removed, and that forms our result. Leading zeroes are removed as they do not affect the value of the number. If all digits are removed, we must return '0', which is the smallest non-negative integer.

**Solution Approach**

The implementation of this algorithm is straightforward once you understand the intuition:

We create an empty list called stk, which we will use as a stack to keep track of the valid digits of the smallest number we are constructing.

We need to retain len(num) - k digits to form the new number after we have removed k digits. The variable remain holds this value.

We iterate over each character c in the string num:

While we still have more digits k to remove, and the stack stk is not empty, and the digit at the top of the stack stk[-1] is greater than the current digit c, we pop the top of the stack. This is because keeping c, which is smaller, will yield a smaller number.

We also decrement k by 1 each time we pop a digit off the stack since that counts as one removal.

After the check (and potential removal), we append the current digit c to the stack. This digit is now part of the new number.

After we finish iterating over num, the stack contains the digits of the resulting number, but it might have more digits than necessary if we didn’t need to remove k digits. Thus, we slice the stack up to remain digits.

Next, we need to convert the list of digits into a string. We join the digits in stk up to the remain index and then we remove any leading zeros with .lstrip('0').

The last step is to handle the case where all digits are removed, resulting in an empty string. If that happens, we return '0' because we must return a valid number and 0 is the smallest non-negative integer. In any other case, we return the joined string of digits that now represents the smallest possible integer after the removal of k digits.

This algorithm makes use of a stack, which is a classic data structure that operates on a Last In, First Out (LIFO) principle. It's an ideal choice to store the digits of the new number because it allows for easy removal of the last added digit when a smaller digit comes next. The process is greedy and makes local optimum choices by preferring smaller digits in the higher place values.

Remember, in Python, a list can act as a stack with the append method to push elements onto the stack and the pop method to remove the top element of the stack.

**Example Walkthrough**

Let's consider a small example to illustrate the solution approach. Suppose the input string num is "1432219" and k is 3. We want to remove 3 digits to make the number as small as possible.

Here's the step-by-step process:

Initialize an empty list stk to represent the stack. The number of digits we want to remain in the final number is remain = len(num) - k = 7 - 3 = 4.

Iterate over each digit in "1432219":

Start with the first digit '1'. Since the stack is empty, we add '1' to stk.

Next is '4'. '4' is greater than '1', so we keep it and push '4' to stk.

Then comes '3'. '3' is smaller than '4' and k > 0, so we pop '4' out of the stack. Now stk = ['1'] and k = 2.

Now we have '2'. '2' is smaller than '3', so we pop '3'. Now, stk = ['1'] and k = 1.

Add '2' to the stack. stk = ['1', '2'].

Another '2' comes, which is the same as the last digit, so we push '2' to stk. stk = ['1', '2', '2'].

Finally, '1' is smaller than '2', so we pop the last '2' from stk. stk = ['1', '2'], and k = 0 (no more removals allowed).

Since we've already removed 3 digits, just push '1' and then '9' to the stack. Now, stk = ['1', '2', '1', '9'].

We've finished processing each digit and our stack stk represents the smallest number we could make. However, we need to make sure we have the right number of digits, which should be remain = 4. Since stk already contains 4 digits, there's no need to slice.

Join the stack to form a number and strip leading zeros (if any). result = ''.join(stk).lstrip('0'). In this case, '1219'.

We return '1219', which is the smallest number possible after removing 3 digits from "1432219".

This example illustrates how the stack helps efficiently manage the digits of the new number, ensuring that smaller digits remain at the higher place values whenever possible.

**Solution Implementation**

class Solution {

public String removeKdigits(String num, int k) {

// Create a StringBuilder to use as a stack to keep track of digits.

StringBuilder stack = new StringBuilder();

// Iterate through each character in the input string.

for (char digit : num.toCharArray()) {

// While the current digit is smaller than the last digit in the stack

// and we still have digits to remove (k > 0), remove the last digit.

while (k > 0 && stack.length() > 0 && stack.charAt(stack.length() - 1) > digit) {

stack.deleteCharAt(stack.length() - 1);

k--;

}

// Append the current digit to the stack (StringBuilder).

stack.append(digit);

}

// If after the iteration we still need to remove more digits, remove from the end.

while (k > 0) {

stack.deleteCharAt(stack.length() - 1);

k--;

}

// Remove leading zeros by finding the index of the first non-zero digit.

int nonZeroIndex = 0;

while (nonZeroIndex < stack.length() && stack.charAt(nonZeroIndex) == '0') {

nonZeroIndex++;

}

// Create a new string starting from the first non-zero digit.

String result = stack.substring(nonZeroIndex);

// If the resulting string is empty, return "0" instead; otherwise, return the string.

return result.isEmpty() ? "0" : result;

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of the given code can be analyzed based on the operations performed. The code iterates over each character in the string num which has a length of n. In the worst case, each character may be pushed to and popped from the stack stk once. Pushing and popping from the stack are O(1) operations, but since the inner while loop could run up to k times for each character, it might appear at first as if the complexity is O(nk). However, each element is pushed and popped at most once, resulting in a time complexity of O(n) overall because the while loop can't execute more than n times over the course of the entire function.

Therefore, the total time complexity of the algorithm is: O(n)

**Space Complexity**

The space complexity is determined by the space used by the stack stk, which in the worst case could contain all characters if k is zero or if all characters are in increasing order. Therefore, the space complexity is proportional to the length of the input string num.

Thus, the space complexity of the algorithm is: O(n)

**Why we can directly compare character with character (stack.peek()>num.charAt(i)), no need to transfer to integer ?**

**Because character '0' mapping to decimal value 48, '1' mapping to 49, '2' mapping to 50 .... '9' mapping to 57, monotonically increasing sequence**

**Refer to**

<https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html>

ASCII Table

Dec  = Decimal Value

Char = Character

'5' has the int value 53

if we write '5'-'0' it evaluates to 53-48, or the int 5

if we write char c = 'B'+32; then c stores 'b'

Dec  Char                          Dec  Char    Dec  Char    Dec  Char

---------                          ---------    ---------    ----------

  0  NUL (null)                      32  SPACE    64  @        96  `

  1  SOH (start of heading)          33  !        65  A        97  a

  2  STX (start of text)            34  "        66  B        98  b

  3  ETX (end of text)              35  #        67  C        99  c

  4  EOT (end of transmission)      36  $        68  D        100  d

  5  ENQ (enquiry)                  37  %        69  E        101  e

  6  ACK (acknowledge)              38  &        70  F        102  f

  7  BEL (bell)                      39  '        71  G        103  g

  8  BS  (backspace)                40  (        72  H        104  h

  9  TAB (horizontal tab)            41  )        73  I        105  i

10  LF  (NL line feed, new line)    42  \*        74  J        106  j

11  VT  (vertical tab)              43  +        75  K        107  k

12  FF  (NP form feed, new page)    44  ,        76  L        108  l

13  CR  (carriage return)          45  -        77  M        109  m

14  SO  (shift out)                46  .        78  N        110  n

15  SI  (shift in)                  47  /        79  O        111  o

16  DLE (data link escape)          48  0        80  P        112  p

17  DC1 (device control 1)          49  1        81  Q        113  q

18  DC2 (device control 2)          50  2        82  R        114  r

19  DC3 (device control 3)          51  3        83  S        115  s

20  DC4 (device control 4)          52  4        84  T        116  t

21  NAK (negative acknowledge)      53  5        85  U        117  u

22  SYN (synchronous idle)          54  6        86  V        118  v

23  ETB (end of trans. block)      55  7        87  W        119  w

24  CAN (cancel)                    56  8        88  X        120  x

25  EM  (end of medium)            57  9        89  Y        121  y

26  SUB (substitute)                58  :        90  Z        122  z

27  ESC (escape)                    59  ;        91  [        123  {

28  FS  (file separator)            60  <        92  \        124  |

29  GS  (group separator)          61  =        93  ]        125  }

30  RS  (record separator)          62  >        94  ^        126  ~

31  US  (unit separator)            63  ?        95  \_        127  DEL

**Refer to**

[L1673.Find the Most Competitive Subsequence (Ref.L84,L402)](note://WEBd8e85b624521547b3108241deb26a6cb)

[L84.Largest Rectangle in Histogram](note://E2FC22D9715540C5A8F06F60FBBE886F)