<https://leetcode.com/problems/integer-to-english-words/>

Convert a non-negative integer num to its English words representation.

**Example 1:**

Input: num = 123

Output: "One Hundred Twenty Three"

**Example 2:**

Input: num = 12345

Output: "Twelve Thousand Three Hundred Forty Five"

**Example 3:**

Input: num = 1234567

Output: "One Million Two Hundred Thirty Four Thousand Five Hundred Sixty Seven"

**Constraints:**

0 <= num <= 2^31 - 1

**Attempt 1: 2024-06-01**

**Solution 1: DFS + Hash Table (30 min)**

class Solution {

    private final String[] LESS\_THAN\_20 = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};

    private final String[] TENS = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};

    private final String[] THOUSANDS = {"", "Thousand", "Million", "Billion"};

    public String numberToWords(int num) {

        if(num == 0) {

            return "Zero";

        }

        int i = 0;

        String words = "";

        // Process every 3 digits(~1000) from least(right) -> most(left)

        // significant digits

        while(num > 0) {

            if(num % 1000 != 0) {

                words = helper(num % 1000) + THOUSANDS[i] + " " + words;

            }

            num /= 1000;

            i++;

        }

        return words.trim();

    }

    private String helper(int num) {

        if(num == 0) {

            return "";

        } else if(num < 20) {

            return LESS\_THAN\_20[num] + " ";

        } else if(num < 100) {

            return TENS[num / 10] + " " + helper(num % 10);

        } else {

            return LESS\_THAN\_20[num / 100] + " Hundred " + helper(num % 100);

        }

    }

}

Time Complexity: O(1)

Space Compleixty: O(1)

Refer to

https://leetcode.com/problems/integer-to-english-words/solutions/70625/my-clean-java-solution-very-easy-to-understand/comments/139940

Technically the time complexity the above solution is O(1), since it has a constant upper bound

(a number can only be so long), but if we want to get an even tighter bound, then we can define

n to be the number of non-zero digits, in which the time complexity is O(n^2) where n is the number

of non-zero digits. This is because this solution doesn't use a StringBuilder, so each concat

operation needs to re-traverse every character of the previously concatenated strings, and the

number of characters is a function of the number of non-zero digits.

There is a good O(n) solution below by @airjordan919 where he uses a single string builder to

generate the solution.

**Refer to**

<https://leetcode.com/problems/integer-to-english-words/solutions/70625/my-clean-java-solution-very-easy-to-understand/>

class Solution {

    private final String[] LESS\_THAN\_20 = {"", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};

    private final String[] TENS = {"", "Ten", "Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};

    private final String[] THOUSANDS = {"", "Thousand", "Million", "Billion"};

    public String numberToWords(int num) {

        if (num == 0) return "Zero";

        int i = 0;

        String words = "";

        while (num > 0) {

            if (num % 1000 != 0)

                words = helper(num % 1000) +THOUSANDS[i] + " " + words;

            num /= 1000;

            i++;

        }

        return words.trim();

    }

    private String helper(int num) {

        if (num == 0)

            return "";

        else if (num < 20)

            return LESS\_THAN\_20[num] + " ";

        else if (num < 100)

            return TENS[num / 10] + " " + helper(num % 10);

        else

            return LESS\_THAN\_20[num / 100] + " Hundred " + helper(num % 100);

    }

}

**Refer to**

<https://algo.monster/liteproblems/273>

**Problem Description**

The LeetCode problem presented requires the conversion of a non-negative integer num into its English words representation. The task is to write a program that will take any non-negative integer and translate it into the corresponding words as one would read it in English. The algorithm needs to handle the structure of English numbering, including the placement of thousands, millions, and billions, as well as the rules for forming numbers less than one hundred.

**Intuition**

The solution to converting numbers into their English words representation relies on understanding the pattern and structure of English numbers. We can break down the problem into manageable parts by considering that:

Numbers from 1 to 19 have unique names which do not follow a particular pattern, so we prepare a list for those (lt20).

Tens from 20 to 90 also have distinct names, so we list them out (tens).

Complex numbers are generally combinations of smaller numbers. For example, 345 can be read as "Three Hundred Forty-Five", which combines "Three Hundred" with the smaller number "Forty-Five".

English names for numbers above 99 are formed by appending a scale word like "Thousand", "Million", or "Billion" and then saying the name of the number that follows.

Given this, we approach the problem by creating a recursive function transfer that can handle numbers less than a thousand. We then iteratively apply this function to parts of the input number, working our way from billions down to units, appending appropriate scale terms, and concatenating all parts to form the full English words representation.

To track units of thousand, we maintain an array thousands, which contains scale words each separated by a factor of a thousand, and an adjusted counter to work on portions of the number. Using division and modulo operations, we can slice the number into segments that can be processed by transfer before attaching the corresponding scale term (if any).

The process starts with the largest possible unit, billions, and strips away parts of the number progressively until the entire number has been converted. We then join the formed segments and clean up any extra spaces to get the final English words representation.

**Solution Approach**

The solution implementation begins with setting up the base case for the number zero, returning 'Zero' if num is equal to zero.

**Data Structures Used:**

lt20: A list containing the English words for numbers 1 through 19.

tens: A list containing the English words for multiples of ten, from 20 to 90.

thousands: A list to denote the scale terms (Billion, Million, Thousand).

res: A list that will accumulate the segments of the word representation.

**Algorithm:**

A recursive helper function transfer is defined to convert numbers less than 1000 into words:

If num is less than 20, it returns the corresponding word from lt20.

If num is less than 100, it returns the word from tens for the tens place followed by the recursive call for the remainder (num % 10).

For larger numbers, it returns the word for hundreds place from lt20, followed by 'Hundred', and then the recursive call for the remainder of dividing by 100.

The main function works with these steps:

Initialization by checking if the input num is zero, subsequently returning 'Zero'.

A loop is set to iterate through scales (billion to thousand) using variables i and j where i starts at 10^9 and j is index position in thousands.

Inside the loop:

Check if current segment (num // i) is non-zero.

If so, call transfer for that segment, and append the result along with the scale term (thousands[j]) to res.

Use modulo operation num % i to move to the next segment.

Increment j and divide i by 1000 to proceed to the next lower scale term.

After processing all segments, join the parts in res stripping trailing or duplicate spaces.

**Patterns Used:**

**Divide and conquer:** The number is divided into smaller parts, each of which is processed independently (billion, million, thousand, less than a thousand).

**Recursion:** A recursive function transfer is used to build up the word representation of numbers less than 1000.

**Modular arithmetic:** Division (num // i) and modulo (num % 10, num % 100, num % i) operations extract specific digits or segments from the number.

**Array indexing:** Lists lt20, tens, and thousands are accessed via indices to retrieve the English words for numbers and scale terms.

**Performance and Complexity:**

The algorithm runs in O(n) where n is the number of digits in the input number since each digit or group of digits is processed once. The space complexity is also O(n) due to the storage required for the word representation of the number.

By breaking down the problem into smaller subproblems and following the structure of English number words, the algorithm effectively converts any non-negative integer into its English words counterpart.

**Example Walkthrough**

Let's take the number 12345 as an example to illustrate the solution approach.

We start by checking if num is zero. Our number isn't zero, so we move to the next step.

We prepare our data structures lt20, tens, and thousands, and an empty list res for accumulating results.

The transfer function is ready to be used, but we start with the scale terms.

We look at our largest unit, billions (10^9). Since 12345 is smaller than one billion, we move to the next scale—millions. It's not a million either, so we proceed to thousands.

We see that our number is greater than 1000, so we process this segment first.

We divide 12345 by 1000, giving us 12 with a remainder. Using transfer, we turn 12 into "Twelve" and append "Thousand" to get "Twelve Thousand". We add this to our res list.

The remainder is now 345, which we process next. It's less than 1000, so the transfer function will handle it directly.

Inside transfer, 345 is not less than 20 or 100, so we further break it down to "Three Hundred" ("Three" from lt20 and "Hundred") and then recursively process 45.

For the number 45, we again call transfer. It's less than 100 but more than 20, so the function returns the corresponding word for the tens place, "Forty", and the word for the ones place, "Five".

We concatenate "Three Hundred" with "Forty-Five" to create "Three Hundred Forty-Five".

We append "Three Hundred Forty-Five" to our res list.

Now our res list is ["Twelve Thousand", "Three Hundred Forty-Five"]. We join these with a space to make the final output: "Twelve Thousand Three Hundred Forty-Five".

We followed the approach, using recursion for numbers under 1000, modular arithmetic to divide the number into manageable segments, and array indexing to map numbers to their respective words. The process broken down into the loop and conditional checks ensures that the number is systematically converted into its English words representation.

**Solution Implementation**

import java.util.HashMap;

import java.util.Map;

public class Solution {

// Static map to hold number to words mapping

private static final Map<Integer, String> NUMBER\_TO\_WORDS\_MAP;

// Static initializer block used to populate the map

static {

NUMBER\_TO\_WORDS\_MAP = new HashMap<>();

// Single digit mappings

NUMBER\_TO\_WORDS\_MAP.put(1, "One");

NUMBER\_TO\_WORDS\_MAP.put(2, "Two");

NUMBER\_TO\_WORDS\_MAP.put(3, "Three");

NUMBER\_TO\_WORDS\_MAP.put(4, "Four");

NUMBER\_TO\_WORDS\_MAP.put(5, "Five");

NUMBER\_TO\_WORDS\_MAP.put(6, "Six");

NUMBER\_TO\_WORDS\_MAP.put(7, "Seven");

NUMBER\_TO\_WORDS\_MAP.put(8, "Eight");

NUMBER\_TO\_WORDS\_MAP.put(9, "Nine");

// Teen mappings

NUMBER\_TO\_WORDS\_MAP.put(10, "Ten");

NUMBER\_TO\_WORDS\_MAP.put(11, "Eleven");

NUMBER\_TO\_WORDS\_MAP.put(12, "Twelve");

NUMBER\_TO\_WORDS\_MAP.put(13, "Thirteen");

NUMBER\_TO\_WORDS\_MAP.put(14, "Fourteen");

NUMBER\_TO\_WORDS\_MAP.put(15, "Fifteen");

NUMBER\_TO\_WORDS\_MAP.put(16, "Sixteen");

NUMBER\_TO\_WORDS\_MAP.put(17, "Seventeen");

NUMBER\_TO\_WORDS\_MAP.put(18, "Eighteen");

NUMBER\_TO\_WORDS\_MAP.put(19, "Nineteen");

// Tens place mappings

NUMBER\_TO\_WORDS\_MAP.put(20, "Twenty");

NUMBER\_TO\_WORDS\_MAP.put(30, "Thirty");

NUMBER\_TO\_WORDS\_MAP.put(40, "Forty");

NUMBER\_TO\_WORDS\_MAP.put(50, "Fifty");

NUMBER\_TO\_WORDS\_MAP.put(60, "Sixty");

NUMBER\_TO\_WORDS\_MAP.put(70, "Seventy");

NUMBER\_TO\_WORDS\_MAP.put(80, "Eighty");

NUMBER\_TO\_WORDS\_MAP.put(90, "Ninety");

// Scale mappings

NUMBER\_TO\_WORDS\_MAP.put(100, "Hundred");

NUMBER\_TO\_WORDS\_MAP.put(1000, "Thousand");

NUMBER\_TO\_WORDS\_MAP.put(1000000, "Million");

NUMBER\_TO\_WORDS\_MAP.put(1000000000, "Billion");

}

// Converts a number to words

public String numberToWords(int num) {

// Special case for zero

if (num == 0) {

return "Zero";

}

StringBuilder wordsBuilder = new StringBuilder();

// Process the number for billions, millions, and thousands

for (int i = 1000000000; i >= 1000; i /= 1000) {

if (num >= i) {

wordsBuilder.append(processThreeDigits(num / i)).append(" ").append(NUMBER\_TO\_WORDS\_MAP.get(i));

num %= i;

}

}

// Append the remaining words for numbers less than a thousand

if (num > 0) {

wordsBuilder.append(processThreeDigits(num));

}

// Remove the leading space and return the result

return wordsBuilder.substring(1);

}

// Helper function to process up to three digits of the number

private String processThreeDigits(int num) {

StringBuilder threeDigitsBuilder = new StringBuilder();

if (num >= 100) {

threeDigitsBuilder.append(" ")

.append(NUMBER\_TO\_WORDS\_MAP.get(num / 100))

.append(" ")

.append(NUMBER\_TO\_WORDS\_MAP.get(100));

num %= 100;

}

if (num > 0) {

// Direct mapping for numbers less than 20 or multiples of 10

if (num < 20 || num % 10 == 0) {

threeDigitsBuilder.append(" ").append(NUMBER\_TO\_WORDS\_MAP.get(num));

} else {

// Combine the tens and ones place for other numbers

threeDigitsBuilder.append(" ")

.append(NUMBER\_TO\_WORDS\_MAP.get(num / 10 \* 10))

.append(" ")

.append(NUMBER\_TO\_WORDS\_MAP.get(num % 10));

}

}

return threeDigitsBuilder.toString();

}

}

**Time and Space Complexity**

**Time Complexity**

The time complexity of this function is primarily influenced by the division and modulo operations inside the while loop. The loop itself is executed a constant number of times (4 iterations, one for each group of digits corresponding to "Billion", "Million", "Thousand", and the sub-thousand level). Each iteration involves constant-time arithmetic operations and a recursive call to the transfer function, which has a worst-case scenario of constant time (since numbers less than 1000 are processed directly, with at most two recursive calls for numbers <100).

Therefore, the overall time complexity is O(1), as it does not scale with the input number.

**Space Complexity**

The space complexity is largely governed by the storage of intermediate string results and the use of recursion for numbers less than 1000. However, the depth of recursion does not exceed a constant (with a maximum recursion depth = 3 for numbers less than 1000). The array res holds a fixed number of strings corresponding to the digit group labels and their English representations.

Consequently, the space complexity is also O(1) since the algorithm allocates a constant amount of additional space.