GUSTAVE EIFFEL UNIVERSITY

Master 2 Mathematics and Computer Sciences

Programmation

Generate efficiently all words of length n over a fixed alphabet {1,...,k}

Mamadou DIOUF

Years : 2024 / 2025

Contents

List of Figures

1 Introduction

An original problem

One want to generate efficiently all words of length n over a fixed alphabet $\{1, ..., k\}$ but up to any permutation of the letter. This means, we don't want to generate 112, 113 and 332 : there are al equivalents up to permutation of the letters. We need only one of them, let choose 12 which is the minimum for the lexicographic order. This way, one can win close to a factor factorial (k). Solve this problem (Mathematically and using the machine) !

Original mail (sorry, in French)

Cher xxxxxx,
c'est yyyyyy (en CC) qui me conseille de m'adresser à toi. Pour des recherches expérimentations impliquant un peu de combinatoire des mots, je voudrais générer efficacement tous l e s mots de longueur n sur un alphabet 1,..,k fixé *mais à renommage des lettres près*. C.-a-d. que ce n'est pas la peine de générer 112, 113 et 332 : ils sont tous équivalents. Je n'ai besoin que de l'un d'entre eux, disons 112 qui est le premier lexicographiquement, et ainsi je gagne presque un facteur k!
zzzzzzz me dit que je peux définir un DFA qui reconnaît les formes normales (une par classe d'équivalence) et de fait c'est p.ex. le langage $1^*2^*...k^*$ et qu'ensuite tu connais des techniques génériques pour générer tous les mots de longueur n dans un langage régulier donné.

Based on this discussion, we will describe codes which generate all with given a fixed alphabet $\{1, 2, .., k\}$.

2 Definition

We apply the suggestions of zzzzzz. We reach to find normal forms of words as $1^{n_1}2^{n_2}...k^{n_k}$ with n_i number of repetition of the letter i for i in $\{1, 2, .., k\}$, $\sum_{i=1}^{k} n_i = n$ and their permutations.
Then we define a class by all permutations(anagrams) of the words.
Thus we resume the problem to find all words define by $1^{n_1}2^{n_2}...k^{n_k}$ such that $\sum_{i=1}^{k} n_i = n$. This we will give us one word for each class and we win close to $k!$ words(for each class). We can note that $1^{n_1}2^{n_2}...k^{n_k}$ is the first lexicographic word of his class. Then our goal is to generate all first lexigrographic for each class.

3 Methods to generate first lexigrographic words

3.1 Combinations with repetitions

A n-combination with repetitions, or n-multicombination, or multisubset of size n from a set S of size k is given by a set of n not necessarily distinct elements of S, where order is not taken into account: two sequences define the same multiset if one can be obtained from the other by permuting the terms. In other words, it is a sample of n elements from a set of k elements allowing for duplicates (i.e., with replacement) but disregarding different orderings (e.g. $\{2,1,2\} = \{1,2,2\}$). Associate an index to each element of S and think of the elements of S as types of objects, then we can let n_i denote the number of elements of type i in a multisubset. Suppose we want to code the above combinations using only two symbols, say 0 and 1. This can be done by letting 0 denote a letter, and letting 1 denote a change from one kind of letter to another. Then each combination will require n zeros, one for each letter, and $k-1$ ones which denotes a change from one kind of letter to another. Thus one n-ombinations will be coded as follows:

$$0^{n_1} \cdot 1 \cdot 0^{n_2} \cdot 1 \cdot \ldots \cdot 1 \cdot 0^{n_k} \text{ with } n_1 + n_2 + \ldots + n_k = n$$

The number of multisubsets of size k is then the number of nonnegative integer (so allowing zero) solutions of the Diophantine equation:

$$n_1 + n_2 + \ldots + n_k = n$$

If S has k elements, the number of such n-multisubsets is denoted by:

$$\left(\!\binom{k}{n}\!\right),$$

a notation that is analogous to the binomial coefficient which counts n-subsets. This expression, k multichoose n, can also be given in terms of binomial coefficients:

$$\left(\!\binom{k}{n}\!\right) = \binom{k+n-1}{n} = \binom{n+k-1}{k-1}.$$

3.2 Method generation n_i lists of length k

For this method, first we generate list of occurences letters $[n_1, n_2, ..., n_k]$, if $n_i = 0$ this word doesn't content the letter i otherwise it content n_i letter i. And we decode them to words in alphabet $\{1, 2, ..., k\}$.

Example: Number of words of length 3 in the alphabet $\{1,2,3\}$ with generate_class_words:
The list
[[0, 0, 3], [0, 1, 2], [0, 2, 1], [0, 3, 0], [1, 0, 2], [1, 1, 1], [1, 2, 0], [2, 0, 1], [2, 1, 0], [3, 0, 0]]
The words:
['333', '233', '223', '222', '133', '123', '122', '113', '112', '111']

3.2.1 Algo 1: generate_class_words

Listing 1: Algo 1

```
def generate_class_words(n, k):
    def generate_occur_letters(n, k, occur_counts, index):
        if index == k:
            if sum(occur_counts) == n:
                copi = occur_counts.copy()
                #coeff_sequence.append(copi)
                words.append(''.join(str(i+1) * copi[i] for i in range( k )))
            return
        for count in range(n + 1):
            occur_counts[index] = count
            generate_occur_letters(n, k, occur_counts, index + 1)

    #coeff_sequence=[]
    words = []
    generate_occur_letters(n, k, [0] * k, 0)
    return words

# Example usage:
n = 3  # Length of the words
k = 3   # Alphabet {1, 2, 3}
words = generate_class_words(n, k)
#print( coeff_sequence)
print(words)
print(len(words))
```

In this algo, we define a recursive fonction which generate list of occurences letters and decode them to words.
It does around 800 steps for the example $n = 3$ and $k = 3$, 400 steps for $n = 2$ and $k = 3$. The mechanism is showed in **figure 1**.

Here's how it works:

1. The **generate_occur_letters** function is a helper that recursively generates lists, occur_counts, where each element represents the count of occurrences of a character (from '1' to 'k') in the final "word."

2. The recursion:

   - **Base case**: When 'index == k', it checks if the sum of occur_counts equals n (ensuring the word length constraint).
   - **Recursive step**: For each character count, it tries all values from 0 to n to explore different ways to partition 'n' among 'k' characters.

3. Time Complexity

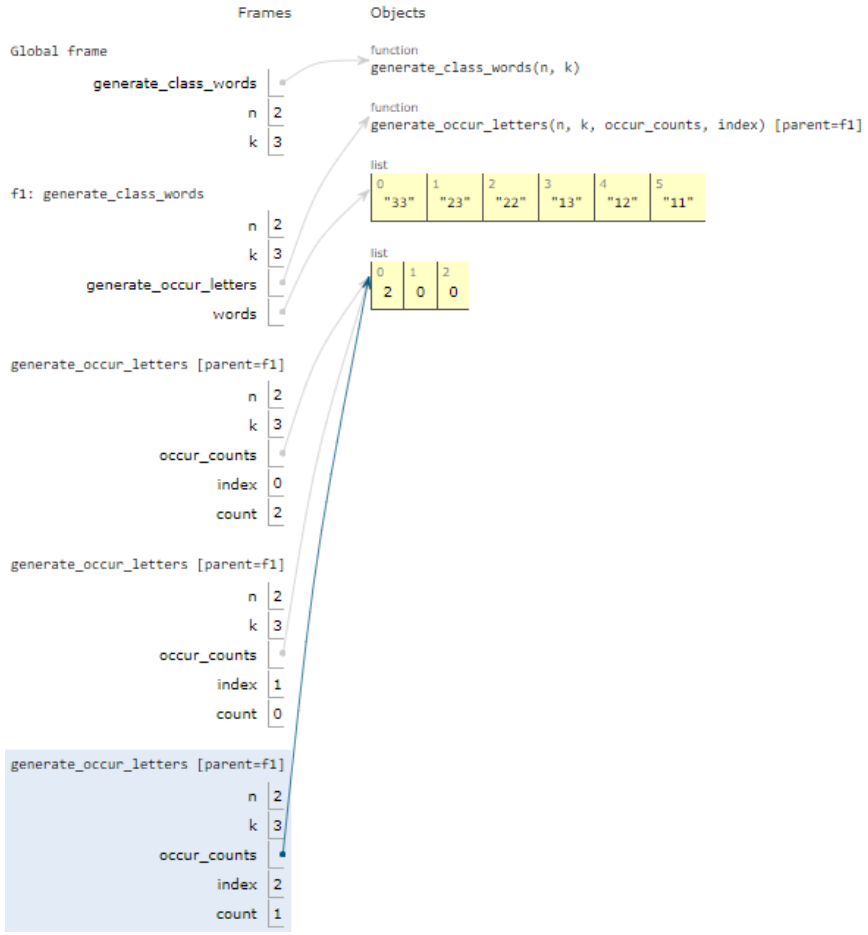

Figure 1: Algo 1 n=2,k=3

- Combinations of Counts : The function generates all ways to partition n "slots" (or positions in the word) into k groups, which is related to the combination with repetition.
  - This problem has $\binom{n+k-1}{k-1}$ solutions by "combinations with repetition").
  - Thus, there are roughly $O((n+k-1)^{k-1})$ recursive calls to **generate_occur_letters** in the worst case.
- String Generation in Base Case: For each valid partition of n characters into k groups, a string of length n is generated by repeating each character according to **occur_counts[i]**. This takes $O(n)$ time per valid combination.
- Total Complexity:Combining both, the time complexity is:

$$O(n \cdot (n+k-1)^{k-1})$$

3.2.2 Algo 2: enumerate_words

For this fonction, two functions which will call:

- **chose(elements,k)** : a recursive fonction which return all subsets of length **k** in elements.
- **enumerate_zeros_ones(n,k)**: a recursive generates binary arrays (lists of zeros and ones) that represent all possible ways to insert $k-1$ ones into $n+k-1$ positions. It

uses **chose(r,k)**, it generates all possible combinations of $k-1$ indices from r. Each combination corresponds to the positions where 1's should appear in the binary array.

- **enumerate_words(n,k,zeros_ones**: it loop over zeros_ones:

  For each binary list (zo) in zeros_ones, it constructs a "word" string by iterating through each 0 and 1. 0s are replaced with the current integer i, and 1s prompt an increment in i. Each complete "word" is added to words.

Listing 2: Algo 2

```
def chose(elements, k): #All subsets of length k in elements
    output = []
    if k == 1:
        return [[t] for t in elements]
    else:
        for i in range(len(elements)):
            head = elements[i]
            tails = chose(elements[i+1:], k-1)
            output += [[head] + tail for tail in tails]
        return output

def enumerate_zeros_ones(n,k):  #words with 0s and 1s
  list_zero_ones = []
  r = [i for i in range(n+k-1)]
  for comb in chose(r, k-1):
    lis = [0]*(n+k-1)
    for i in comb:
      lis[i] = 1
    list_zero_ones.append(lis)
  return list_zero_ones

def enumerate_words(n,k,zeros_ones): #all words of length n in the alphal

  words =[]
  for zo in zeros_ones:
    word=''
    i = 1
    for j in zo:
      if j == 0:
        word+=str(i)
      else:
        i+=1
    words.append(word)
  return words
```

Example: enumerate_words(4,3) done around 600 steps enumerate_words(3,3) around 400 steps and enumerate_words(2,3) around 250 steps.
Answers for $(3,3)$: binary lists [[1, 1, 0, 0, 0], [1, 0, 1, 0, 0], [1, 0, 0, 1, 0], [1, 0, 0, 0, 1], [0, 1, 1, 0, 0], [0, 1, 0, 1, 0], [0, 1, 0, 0, 1], [0, 0, 1, 1, 0], [0, 0, 1, 0, 1], [0, 0, 0, 1, 1]]

Words: [’333’, ’233’, ’223’, ’222’, ’133’, ’123’, ’122’, ’113’, ’112’, ’111’]

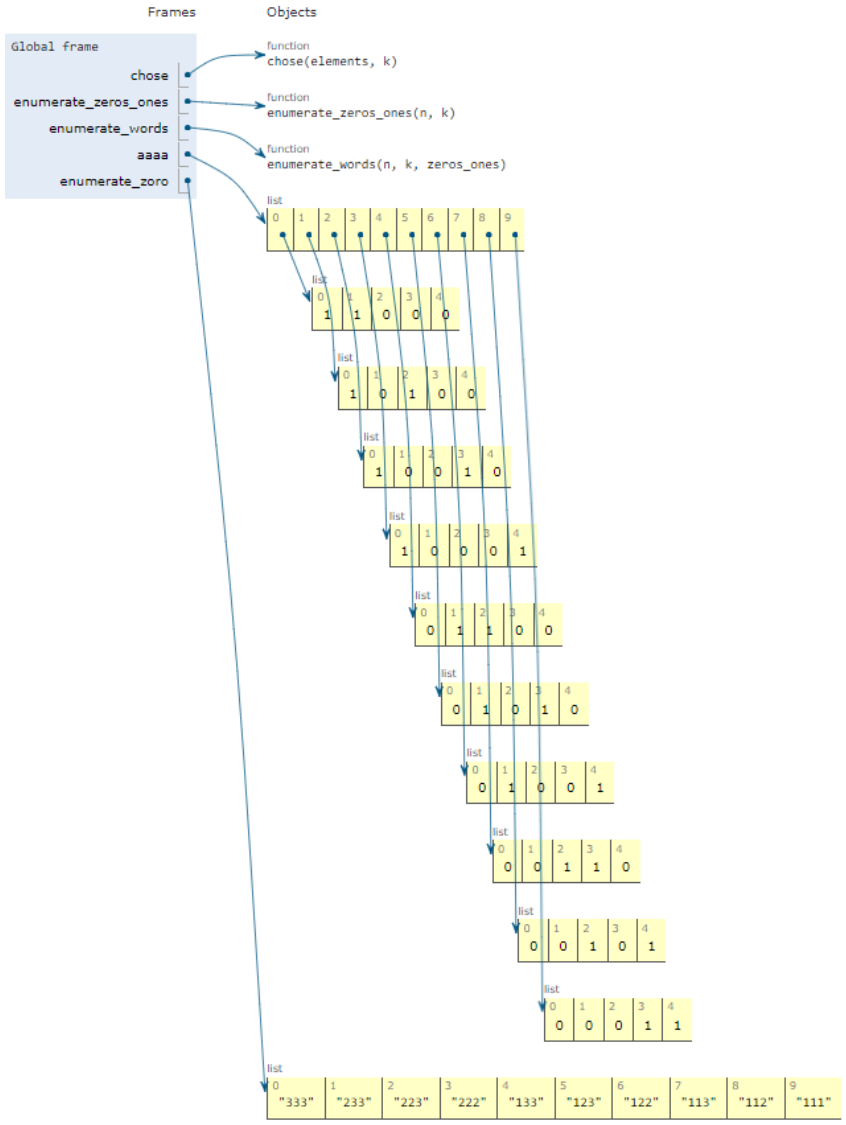

Figure 2: Algo 2 enumerate_words(3,3)

We can see it more fast than the Algo 1 in this cases.
Here’s how it works:

1. Loop over zeros_ones:

   - For each binary list (zo) in zeros_ones, it constructs a "word" string by iterating through each ’0’ and ’1’.

   - 0s are replaced with the current integer ’i’, and 1s prompt an increment in ’i’.

   - Each complete "word" is added to ’words’.

2. Constructing Words:

   - Starting from i = 1, the function increments i each time it encounters a ’1’ in zo, effectively shifting to the next integer in the sequence.

3. Complexity Analysis:

   - The function processes each binary list in zeros_ones, which has length $O((n+k-1)^{k-1})$, as determined by the combination count in the previous function.

   Generating Each Word:

   - For each binary list, the function constructs a word of length n, taking $O(n)$ time per word.

   - Thus, the total time complexity is:

$$O(n \cdot (n+k-1)^{k-1})$$

This two algorithms are same complexity (for worst case).

3.3 Method generate sets of numbers which have as sum n

Firstly, we generate all sets(without zeros) which have as sum n. Secondly, we decode them to lists of zeros and ones. Here each value set represent the occurence of one number. Finally, we transform each list to a word in alphabet $\{1, 2, ..., k\}$.

Each step is done respectively by the functions **subset_sum**, **enumerate_zeros_ones2** and **enumerate_words2**
Example: subsets (of sum 3)[[1, 2], [1, 1, 1], [2, 1], [3]]
Words : ['333', '233', '222', '133', '122', '111', '223', '123', '113', '112']

Listing 3: Algo 3

```
def subset_sum(numbers, target, partial, lis):
    s = sum(partial)

    # check if the partial sum is equals to target
    if s == target :
        #print ("sum(%s)=%s" % (partial, target))

        lis.append(partial)


    if s > target:

        return  # if we reach the number why bother to continue

    for i in range(len(numbers)):
        n = numbers[i]
        remaining = numbers[i+1:]
        if partial + [n] not in lis:
          subset_sum(remaining, target, partial + [n], lis)

    return lis
```

```
def enumerate_words2(n,k,zeros_ones2): #words of length n in the alphabet

  words =[]
  for zo in zeros_ones2:
    word=''
    i = 1
    for j in zo:
      #print(j)
      if '0' in j:
        word+=str(i)*len(j)
      if '1' in j:
        i+=1
        #print(word)
    if word not in words:
      words.append(word)
  return words
```

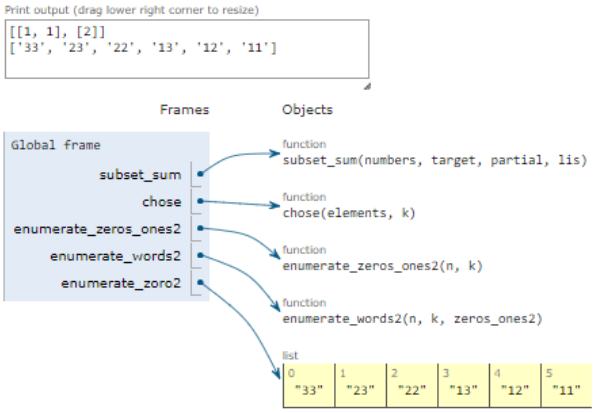

Figure 3: enumerate_words2(2,3,..)

It makes around 750 steps for $(2, 3)$.
Example: Answers for $(4, 3)$:
Subsets: [[1, 2, 1], [1, 3], [1, 1, 2], [1, 1, 1, 1], [2, 1, 1], [2, 2], [3, 1]]
Words: ['3333', '2333', '2223', '2222', '1333', '1223', '1222', '1113', '1112', '1111', '2233', '1233', '1133', '1122', '1123']

Complexity

- Loop over zeros_ones2 :

  Each zo in zeros_ones2 represents a possible pattern. The size of zeros_ones2 is exponential, up to $O((n+k-1)^{k-1})$ due to the combinations generated by the previous function enumerate_zeros_ones2.

- Constructing Words:

  For each zo, constructing a word takes $O(n)$ time, as it iterates over each position to add the characters according to the groupings.

- Checking :

  Each word is checked for uniqueness before being added to words. In the worst case, this can take $O(2^k)$ comparisons due to exponential growth in the number of generated words.

- Total Time Complexity:

  Combining the steps, the time complexity is approximately:

$$O(n \cdot (n + k - 1)^{k-1} \cdot 2^k)$$

4 Conclusion

The suggestion of zzzzzz is a good idea to generate all words of length n given an alphabet $\{1, 2, \cdots, k\}$. And it ensure us the win close $k!$. But the classes in method is not the same of the classes of yyyyyy.
We can also conclude the algorithm 2 is better than the two others to solve the original problem.

5 Annexe

https://en.wikipedia.org/wiki/Combination