Published in final edited form as:

Discrete Math Algorithms Appl. 2017 June ; 9(3): . doi:10.1142/S1793830917500422.

A Near-Optimal Algorithm to Count Occurrences of Subsequences of a Given Length

Jose Torres-Jimenez[a,*], **Idelfonso Izquierdo-Marquez**[a], **Daniel Ramirez-Acuna**[a], and **Rene Peralta**[b]

[a]CINVESTAV-Tamaulipas, Information Technology Laboratory, Km. 5.5 Carretera Cd. Victoria-Soto la Marina, 87130, Cd. Victoria Tamps., México

[b]National Institute of Standards and Technology, Gaithersburg, MD 20899-8910 USA

Abstract

For $k \in \mathbb{Z}^+$, define Σ_k as the set of integers $\{0, 1, \ldots, k-1\}$. Given an integer n and a string t of length $m \geq n$ over Σ_k, we count the number of times that each one of the k^n distinct strings of length n over Σ_k occurs as a subsequence of t. Our algorithm makes only one scan of t and solves the problem in time complexity mk^{n-1} and space complexity $m + k^n$. These are very close to best possible.

Keywords

Counting Subsequences; Perfect Tree

1. Introduction

An *alphabet* Σ is a finite set of symbols; a *string* over an alphabet Σ is a finite sequence of symbols from Σ; the *length* of a string is the number of symbols in the string; the set of strings of length n over Σ is denoted by Σ^n, and the set of all strings over Σ (including the empty string λ) is denoted by Σ^*.

The concepts of substring and subsequence are very important in computational problems defined on strings. Such problems are important in many fields, including coding theory, machine learning, and analysis of biological sequences (Elzinga et al., 2008). A substring is a contiguous part of a string, while the elements of a subsequence are not necessarily contiguous in the string. In formal terms, let x, y be two strings over an alphabet Σ, x is a substring of y if there exist strings $w_1, w_2 \in \Sigma^*$ such that $w_1 x w_2 = y$. Let $z = z_1, z_2, \ldots, z_n$ over Σ. z is a subsequence of a string y over Σ if there exist $n + 1$ strings $w_1, w_2, \ldots, w_{n+1} \in \Sigma^*$ such that $w_1 z_1 w_2 z_2 \cdots w_n z_n w_{n+1} = y$.

*Corresponding author: Jose Torres-Jimenez. Tel: (52) 834-1070220, Fax: (52) 834-3147392.

We are interested in strings over alphabets whose elements are integers. For an integer $k \in \mathbb{Z}^+$ let $\Sigma_k = \{0, 1, \ldots, k-1\}$ (we assume a coding that assigns a unique symbol to each integer). Given positive integers n, m with $n \leq m$, the problem addressed in this work is to count the number of times that each string in $\sum_k^n$ occurs as a subsequence of a string $t \in \sum_k^m$. This problem has three parameters: the integer k, the integer n, and the string t. For example, consider the case $k = 2$ (which defines $\Sigma_2 = \{0, 1\}$), $n = 3$, and $t = 0\,1\,0\,1\,0\,0\,1 \in \sum_2^7$. Table 1 shows the number of times that each string in $\sum_2^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ occurs as a subsequence of $t \in \sum_2^7$. Table 1 also shows the indices of t at which the subsequences occur, although we are interested only in the number of occurrences. For example, the first occurrence of the string $s = 110$ is at positions 2, 4, 5 of t, and the second occurrence is at positions 2, 4, 6.

2. Related Problems

For string subsequences the most studied problem is finding the longest common subsequence of two strings; the algorithm of (Wagner and Fischer, 1974) solves the problem in time $O(n \cdot \max(1, m/\log n))$ for strings with lengths n and m, $n > m$. The longest increasing subsequence problem is similar: given a string t of length m the objective is to find the largest j such that $i_1 < i_2 < \cdots < i_j$ and $t_{i_1} < t_{i_2} < \cdots < t_{i_j}$. The algorithm in (Crochemore and Porat, 2010) solves the problem in $O(m \log \log n)$, where n is the maximal length of increasing subsequences of t. The combination of the above two problems is the longest common increasing subsequence problem, i.e., the problem of computing the longest common subsequence that is also an increasing subsequence of the given strings. Some important works on this problem are (Yang et al., 2005; Brodal et al., 2006).

Related to counting subsequences in a given string t are problems like counting the number of different subsequences of t, and counting the number of different subsequences of size n of t.

In (Elzinga et al., 2008) the number of distinct subsequences in a string t of length m is determined in time $\Theta(m)$ using a dynamic programming algorithm. Denote by t^i, $1 \leq i \leq m$, the prefix of t formed by the first i symbols of t; and let t^0 represent the empty string. Let $s \preceq t$ denote that string s is a subsequence of t. Before reading any character of t the number of subsequences is 1 since the empty string is counted as a subsequence of t. These are the rules applied for each character t_j read from the string t:

- If $t_j \not\preceq t^{j-1}$ the number of distinct subsequences doubles.
- If $t_j \preceq t^{j-1}$ the doubling is compensated by subtracting the number of subsequences until before the last occurrence of the character t_j.

For the problem of counting the number of distinct subsequences of length n in a string $t \in \Sigma^m$ we briefly describe the method given in (Rahmann, 2006).

Define $S_{i,j} = \{s \in \Sigma^i : s \preceq t_1 \cdots t_j\}$ as the set of subsequences of length i in the prefix t^j, and define $C_{i,j} = |S_{i,j}|$ as the cardinality of $S_{i,j}$. We refine the above definitions by imposing a

condition on the last character σ of each s in $S_{i,j}$ as follows: $S_{i,j}[\sigma] = \{s \in \Sigma^i : s \preceq t_1 \cdots t_j$ and $s_i = \sigma\}$, and $C_{i,j}[\sigma] = |S_{i,j}[\sigma]|$. In this manner $C_{0,j} = 1$ for all j because $S_{0,j} = \{t^0\}$; $C_{0,j}[\sigma] = 0$ since $S_{0,j}[\sigma] = \varnothing$ for all $\sigma \in \Sigma$; and $C_{i,j} = \Sigma_{\sigma \in \Sigma} C_{i,j}[\sigma]$ for $i > 0$ and all j. In addition, it is convenient to define $C_{i,j} = 0$ if $i > j$. The objective is to compute $C_{n,m}$ by using the following recurrence relation for $j = 1, \ldots, m$ and $i = 1, \ldots, j$ (if $j > n$ then it is enough to compute the values of the recurrence relation for $i = 1, \ldots, n$):

$$C_{i,j}[\sigma] = \begin{cases} C_{i,j-1}[\sigma] & if\ t_j \neq \sigma \\ C_{i-1,j-1} & if\ t_j = \sigma \end{cases}$$

This last problem is similar to the problem addressed in the present work. The difference is that our objective is to count the number of times that each string in the set Σ^n occurs as a subsequence of a string $t \in \Sigma^m$. If the alphabet Σ has k symbols then the output of the problem will be k^n values corresponding to the k^n strings in the set Σ^n.

3. The Algorithm

Given k and n, we use a perfect k-ary tree of height n to count the number of times that each string in $\sum_k^n$ occurs as a subsequence of a string $t \in \sum_k^m$. The perfect k-ary tree of height n has $\frac{k^{n+1}-1}{k-1}$ nodes of which k^n are leaf nodes. The tree has the following characteristics:

- The nodes are labelled with the integers from 1 to $\frac{k^{n+1}-1}{k-1}$. The labelling is done from left to right and from top to bottom.
- The k edges emanating from an internal node are labelled with the integers from 0 to $k - 1$.
- There is a counter associated with each node except the root node. The counter of a node stores the number of occurrences as a subsequence of t of the string formed by concatenating the label of the edges in the path from the root node to the node.

Figure 1 shows the perfect tree created by the algorithm for $k = 2$ and $n = 3$. The concatenations of the edge labels in the paths from the root node to the nodes in the i-th level produce the elements in the set $\sum_k^i$. In particular, the concatenation of the n edge labels in the path from the root node to a leaf node gives a string of $\sum_k^n$. Thus there is a correspondence between the strings formed by concatenating the labels of the edges in the paths to the leaf nodes and the elements in the set $\sum_k^n$. In this example, the paths from the root node to the leaf nodes produce the strings of $\sum_2^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

Inside each node (other than the root node) there are two integers, the first one is the label of the node and the second one is the current value of the counter of the node. The root node has no counter and its label is 1.

The algorithm reads the symbols of the string t one at a time. Let t_i denote the i-th symbol of the string t. Before processing any symbol of t each string in $\cup_{j=1}^{n} \sum_{k}^{j}$ has occurred zero times. Thus the counters of all nodes are initially zero. For the i-th symbol read from the string t the algorithm performs the following two steps to update the counters:

1. From level n to level 2 update the counter of each node whose edge from the parent node is labelled with the symbol t_i according to:

$$\text{counter}(\text{node}) = \text{counter}(\text{node}) + \text{counter}(\text{parent})$$

2. In level 1 increment by one unit the counter of the node whose edge from the root node has the label t_i.

By performing these steps for every symbol of t the algorithm counts all occurrences of the strings in $\cup_{j=1}^{n} \sum_{k}^{j}$ In what follows we describe how the updates of the counters of the tree of Figure 1 take place as the first three symbols of the string t = 0 1 0 1 0 0 1 are read:

1. $t_1 = 0$:
   - **a.** Counters of nodes 8, 10, 12, 14 in the third level are updated; in all cases the result is 0 since the counters of those nodes and the counters of their parents are 0.
   - **b.** Counters of nodes 4, 6 in the second level are updated; in all cases the result is 0.
   - **c.** The counter of node 2 in the first level is incremented by one unit; now its value is 1.
2. $t_2 = 1$:
   - **a.** Counters of nodes 9, 11, 13, 15 in the third level are updated; in all cases the result is 0.
   - **b.** Counters of nodes 5, 7 in the second level are updated; in this case the counter of node 5 becomes 1 and the counter of node 7 remains at 0.
   - **c.** The counter of node 3 in the first level is incremented by one unit; now its value is 1.
3. $t_3 = 0$:
   - **a.** Counters of nodes 8, 10, 12, 14 in the third level are updated; this time the counter of node 10 becomes 1, and the counters of nodes 8, 12, 14 remain at 0.

b. Counters of nodes 4, 6 in the second level are updated; in this case both counters are set to 1.

c. The counter of node 2 in the first level is incremented by one unit; now its value is 2.

Figure 2 shows the status of the tree after processing the first three symbols of $t = 0\ 1\ 0\ 1\ 0\ 0\ 1$. The nodes with counters distinct from 0 are the nodes 2, 3, 4, 5, 6, 10. The strings represented by these nodes are: $node(2) = 0$, $node(3) = 1$, $node(4) = 00$, $node(5) = 01$, $node(6) = 10$, $node(10) = 010$.

Once the algorithm processes all symbols of t, the counter of each leaf node gives the number of occurrences in t of the string $s \in \sum_k^n$ formed by concatenating the label of the edges in the path from the root node to the leaf node. The algorithm is able to count all occurrences of each string in $\cup_{j=1}^{n} \sum_k^j$, but we are interested only in the occurrences of the strings of $\sum_k^n$. In the above example only the string 010 has occurred after reading the first three symbols of $t = 0\ 1\ 0\ 1\ 0\ 0\ 1$.

For each symbol t_i of the string t, the algorithm updates all nodes that are endpoints of an edge with label t_i. In the perfect tree constructed by the algorithm for k and n, the total number of nodes that are endpoints of an edge is the total number of nodes minus the root node. Then, for every symbol σ of the alphabet $\Sigma_k = \{0, 1, \ldots, k-1\}$ there are $\left(\frac{k^{n+1}-1}{k-1} - 1\right)/k$ nodes in the tree each of which satisfies that its edge from the parent node has label σ. Simplifying this expression we have:

$$\frac{\frac{k^{n+1}-1}{k-1} - 1}{k} = \frac{\frac{\left(k^{n+1}-1\right)-(k-1)}{k-1}}{k} = \frac{k^{n+1}-k}{k\,(k-1)} = \frac{k^n - 1}{k-1}.$$

Therefore, for every symbol read from the string t the algorithm updates the counters of $(k^n - 1)/(k - 1)$ nodes of the tree. The updating of a node takes one operation, either a sum of two terms or an increment of one unit. Accordingly, if the string t has m symbols then the algorithm counts all occurrences of all strings of length n over Σ_k in $m\left(\frac{k^n - 1}{k-1}\right)$ operations.

In general, as a symbol σ is read from the string t, the number of occurrences of the strings in $\sum_k^n$ whose last character is σ grows, and so the counters of those strings must be updated. For each symbol $\sigma \in \Sigma_k$ there are k^{n-1} strings of $\sum_k^n$ whose last symbol is σ, and therefore at least k^{n-1} counters must be updated each time a character from t is processed. Thus, any algorithm that processes sequentially the m characters of t will solve the problem in $m(k^{n-1})$ operations. This complexity is for the general case (nothing is assumed about the distribution and frequency of the characters of t). For example, if $t = \sigma\ \sigma \cdots \sigma$ is formed by m occurrences of the character σ, then the result can be computed easily: the value of the

counter associated with the string $s \in \sum_k^n$ formed by n characters σ is $\binom{m}{n}$, while the value of the counters for the other strings in $\sum_k^n$ is zero.

The execution time $m(k^n - 1)/(k - 1)$ of our algorithm is greater than the minimum possible $m(k^{n-1})$ only by a factor $k/(k - 1)$. The memory required by our algorithm is $O(m)$ to store the string t, and $O((k^{n+1} - 1)/(k - 1))$ to store the perfect k-ary tree of height n. Any algorithm requires $O(m)$ to store the string t, and $O(k^n)$ to store the counters associated with each string of $\sum_k^n$. Again, for large k the spatial complexity of our algorithm is $O(m)$ plus $O((k^{n+1} - 1)/(k - 1)) = O(k^{n+1}/k) = O(k^n)$.

4. Implementation of the Algorithm

Since all internal nodes have exactly k children, there is a straight-forward way to store the nodes of the tree in an array. The perfect tree constructed for the given values of k and n has $\frac{k^{n+1}-1}{k-1}$ nodes, and they are labelled from 1 to $\frac{k^{n+1}-1}{k-1}$. The tree can be stored in an array of size equal to the number of nodes of the tree by storing the node with label i in the i-th position of the array. The positions of the array correspond to the labels of the nodes, and the contents of the positions correspond to the counters of the nodes.

Let $d = k - 2$ be a value dependent of the given k. If the node at position i is an internal node then its children nodes are in positions $(i)(k) - d$, $(i)(k) - d + 1$, …, $(i)(k) - d + k - 1$ of the array; and the parent of the node at position i, $i \neq 1$, is at position $\left\lfloor \frac{i+d}{k} \right\rfloor$ of the array.

To perform the updates of the counters associated with each node in an efficient way, we use the fact that nodes that are endpoints of edges with equal labels are stored in positions separated by a distance of k in the array. For example, in a tree for $k = 3$ the nodes at positions 2, 5, 8, 11, … are endpoints of an edge with label 0, the nodes at positions 3, 6, 9, 12, … are endpoints of and edge with label 1, and the nodes at positions 4, 7, 10, 13, … are endpoints of an edge with label 2.

The algorithm described in the previous section first updates counters of nodes in the last level of the tree, then updates counters of nodes in the next level, and so on until reaching level 1. To preserve this order, the implementation of the algorithm first updates the last node of the array that is an endpoint of an edge with label t_i, where t_i is the symbol currently read from the string t. Then, starting from this node the algorithm updates the nodes separated by k positions, going from the end to the beginning of the array.

Algorithm 1 shows pseudocode for the algorithm. The operation $(j + d)/k$ in line 11 is an integer division and computes the parent of node j. The last three lines of Algorithm 1 print the number of occurrences of the strings in $\sum_k^n$. The nodes from position $total\_nodes - k^n + 1$ to position total nodes correspond to the k^n elements of $\sum_k^n$ listed in lexicographic order.

To get a sense of the time required by the algorithm for some values of k and n, we performed the experiments shown in Table 2. The string t in all cases was generated randomly and had 100,000 symbols. The machine on which the program was run has an Intel® Core™ 2 Duo processor at 3.06 GHz.

5. Conclusions

We developed an algorithm to count the number of times that each string of length n over the alphabet $\Sigma_k = \{0, 1, \ldots, k-1\}$ occurs as a subsequence of a string $t \in \sum_k^m$, where $n, k, m \in \mathbb{Z}^+$, $m \geq n$. The algorithm uses a perfect k-ary tree of height n to count in one scanning of the symbols of t all occurrences of the strings in $\sum_k^n$. The execution time of the algorithm is $m\left(\frac{k^n-1}{k-1}\right)$, and this execution time is greater than the minimum possible mk^{n-1} (for an algorithm based on scanning sequentially the characters of t) only by a factor $k/(k-1)$. The algorithm was implemented using an array of size $\frac{k^{n+1}-1}{k-1}$. Each element of the array is a counter of the number of times that a string from $\cup_{j=1}^n \sum_k^j$ has occurred as a subsequence t; the last k^n elements of the array correspond to the elements of $\sum_k^n$ in lexicographic order. The algorithm is able to solve in a reasonable amount of time (less than five minutes) cases like $k = 4$, $n = 10$, and a string t of length 100,000.

Algorithm 1

count_subsequences(k, n, t)

```
1:  total_nodes ← (k^(n+1) − 1)/(k − 1)
2:  v ← array(total_nodes)
3:  for i ← 1 to total_nodes do
4:      v[i] ← 0
5:  end for
6:  d ← k − 2
7:  m ← length(t)
8:  for i ← 1 to m do
9:      j ← total_nodes − k + t_i + 1
10:     while j > k + 1 do
11:         v[j] ← v[j] + v[(j + d)/k]
12:         j ← j − k
13:     end while
14:     v[j] ← v[j] + 1
15: end for
16: for i ← total_nodes − k^n + 1 to total_nodes do
17:     print(v[i])
18: end for
```

Acknowledgments

The authors acknowledge: ABACUS-CINVESTAV, CONACYT grant EDOMEX-2011-COI-165873; and General Coordination of Information and Communications Technologies (CINVESTAV-CGSTIC "Xiuhcoatl") for providing access to high performance computing. The following project partially funded this research: 238469 - CONACyT Métodos Exactos para Construir Covering Arrays Óptimos.

References

Brodal, G., Kaligosi, K., Katriel, I., Kutz, M. Faster algorithms for computing longest common increasing subsequences. In: Lewenstein, M., Valiente, G., editors. Combinatorial Pattern Matching. Springer Berlin Heidelberg; 2006. p. 330-341.Vol. 4009 of Lecture Notes in Computer Science

Crochemore M, Porat E. Fast computation of a longest increasing subsequence and application. Information and Computation. 2010; 208(9):1054–1059.

Elzinga C, Rahmann S, Wang H. Algorithms for subsequence combinatorics. Theoretical Computer Science. 2008; 409(3):394–404.

Rahmann, S. Subsequence combinatorics and applications to microarray production, DNA sequencing and chaining algorithms. In: Lewenstein, M., Valiente, G., editors. Combinatorial Pattern Matching. Springer Berlin Heidelberg; 2006. p. 153-164.Vol. 4009 of Lecture Notes in Computer Science

Wagner RA, Fischer MJ. The string-to-string correction problem. J ACM. Jan; 1974 21(1):168–173.

Yang IH, Huang CP, Chao KM. A fast algorithm for computing a longest common increasing subsequence. Information Processing Letters. 2005; 93(5):249–253.

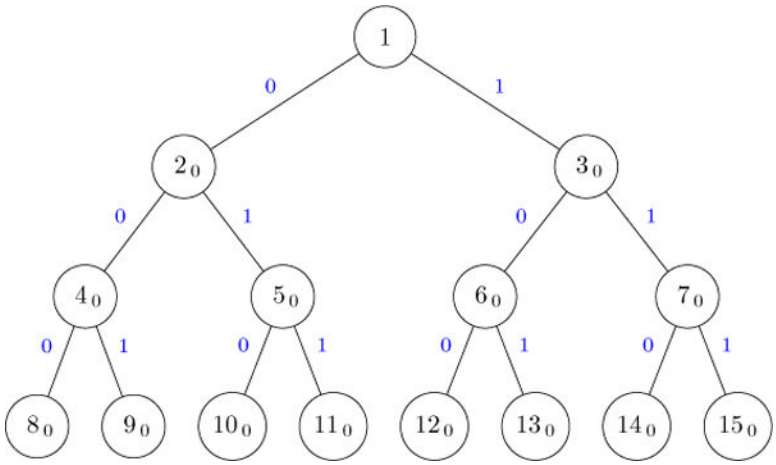

Figure 1.
Perfect tree created by the algorithm for $k = 2$ and $n = 3$.

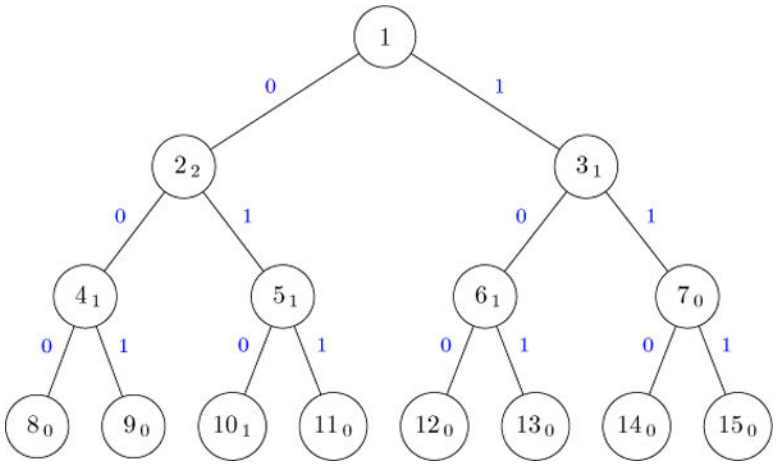

Figure 2.
Status of the tree for $k = 2$ and $n = 3$ after reading the first three symbols of $t = 0\ 1\ 0\ 1\ 0\ 0\ 1$.

Table 1

Occurrences of the elements in $\sum_2^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ in the string $t = 0\ 1\ 0\ 1\ 0\ 0\ 1$.

s	Occurrences	Indices
000	4	{1, 3, 5}, {1, 3, 6}, {1, 5, 6}, {3, 5, 6}
001	7	{1, 3, 4}, {1, 3, 7}, {1, 5, 7}, {1, 6, 7}, {3, 5, 7}, {3, 6, 7}, {5, 6, 7}
010	7	{1, 2, 3}, {1, 2, 5}, {1, 2, 6}, {1, 4, 5}, {1, 4, 6}, {3, 4, 5}, {3, 4, 6}
011	4	{1, 2, 4}, {1, 2, 7}, {1, 4, 7}, {3, 4, 7}
100	4	{2, 3, 5}, {2, 3, 6}, {2, 5, 6}, {4, 5, 6}
101	6	{2, 3, 4}, {2, 3, 7}, {2, 5, 7}, {2, 6, 7}, {4, 5, 7}, {4, 6, 7}
110	2	{2, 4, 5}, {2, 4, 6}
111	1	{2, 4, 7}

Table 2

Execution time of the algorithm for some values of k and n; the string t has 100,000 symbols.

k	n	Nodes	Time (in seconds)
2	6	127	0.013061
2	9	1 023	0.077676
2	12	8 191	0.584635
2	15	65 535	4.86974
2	18	524 287	43.4101
3	6	1 093	0.08307
3	8	9 841	0.741019
3	10	88 573	6.91008
3	12	797 161	66.0708
4	6	5 461	0.22226
4	8	87 381	3.62427
4	10	1 398 101	256.516
5	6	19 531	0.850961
5	8	488 281	21.7044