



# Counting array algorithms for the problem of finding appearances of all possible patterns of size $n$ in a sequence



Yuriy Fofanov, Viacheslav Fofanov, B. Montgomery Pettitt,

Department of Computer Science, University of Houston, Houston, Texas 77204

## Abstract

A new concept of a counting array allows us to map the problem of finding appearances of all possible patterns of size  $n$  ( $n$ -mers) in a sequence or text of size  $m$ , for large  $m$ , onto a useful data structure. The run time operation count estimation  $O(4^{n+m})$  makes it computationally convenient to accomplish calculation of the statistics of the presence of all possible 7-20-mers in more than 250 genomes, including human genome.

## Introducing the problem

Statistical analysis of appearance of short subsequences in different DNA sequences, from individual genes to full genomes, attracts attention for various reasons. An incomplete list of its applications includes PCR primer, microarray probe design, and genome identification based on frequency distribution of short subsequences ( $n$ -mers or motifs). Algorithmically, such type of analyses employ a repeatable search for the short patterns in genomes, also known as exact string matching problem.

Exact string matching is a well-developed area in computer science. The traditional definition of this problem is the following: Given a string  $P$  of size  $n$  called the pattern and the longer string  $T$  of size  $m$  called the text, the exact matching problem is to find all occurrences, if any, of pattern  $P$  in text  $T$ .

Algorithms applying precomputing to pattern or the sequence have been developed in the area of computer science and applied in the area of genomics. However, no studies have been performed for  $n > 11$  due to the rapid increase of the computational difficulties which appear because the total number of different  $n$ -mers grows exponentially fast when  $n$  increases.  $4^n$ . The operation count estimation  $O(4^n km)$  for existing algorithms which precompute patterns and  $O(4^n km)$  for algorithms which precompute sequence (here  $k$  is the number of texts that must be searched) is simply unacceptable.

## Introducing the solution

There are four specific points that characterize our problem versus previous algorithmic studies:

- Relatively short pattern lengths:  $\max(N)=25$ ;
- Only 4 characters in the alphabet. All DNA sequences contain only 4 nucleotides A (adenine), T (thymine), C (cytosine), and G (guanine);
- For each value of  $n$ , the search has to be performed simultaneously for all possible  $(4^n)$   $n$ -mers.
- Regarding each  $n$ -mer (pattern), we are interested only in its presence/absence in each genome (text), or in some rare cases how many times it appears in genome.

To take advantage of the specifics of our problem, in particular the fact that we can perform at once all calculation for all  $n$ -mers for each given value of  $n$ , (which is relatively small), we decided to employ an approach similar to the one used in the well-known counting-sort algorithm. The basic idea is to set in correspondence to each of  $4^n$   $n$ -mers a particular element of counting array  $A$  and define the procedure to convert the  $n$ -mer character sequence to the index of an element in such an array.

## Calculation of presence of all possible $n$ -mers in a given text

```

NUMBER-OF-N-MERS-IN-TEXT-2(T, n, n1)
1 sum ← 0
2 for prefix ← 0 to -1
3   do for i ← 0 to 4n-1
4     do A[i] ← 0
5   for j ← 0 to length(T)-n
6     if (prefix=CONVERT_TO_INTEGER_VALUE(T(j, j+n1-1))
7       then index=CONVERT_TO_INTEGER_VALUE(T(j+n1, j+n-1))
8         if A[index]=0
9           then sum ← sum+1
10        A[index]=1
11 return sum

```

In this algorithm,  $T(j_1, j_2)$  stands for substring of the string  $T$ , starting in position  $j_1$  and ending in position  $j_2$ . The function CONVERT\_TO\_INTEGER\_VALUE(s) is needed to convert string  $s$  of length  $n$ , which in our case is created using only a 4 character alphabet, to a unique integer value – corresponding to an index in the array  $A$ . Using a naive algorithm this function can be implemented to have running time  $O(n)$  but we can utilize the fact that only 2 bits are needed for each character and that we read the text  $T$  sequentially, so that each string  $T(j_1, j_2)$  already contains  $n-1$  elements of the next one  $T(j_1+1, j_2+1)$ . We can implement the function CONVERT\_TO\_INTEGER\_VALUE() using simple binary shift operations so it will be executed in  $O(1)$  time.

The operation count of this algorithm is  $O(4^{n_1}(4^{n-n_1} + m)) = O(4^n + 4^{n_1} m)$ . In fact, in the case mentioned above, if the available computer has more than 2.5 Gb RAM, which means it can handle an array  $A$  necessary to keep track of 17-mers, it takes only about 4 times longer to count all 18-mers and 64 times longer to count all 20-mers. In practice, using this algorithm, the number of 20-mers in *Mycobacterium tuberculosis* H37Rv genome can be calculated in less than a minute on a 1 GHz CPU. We also were able to perform such calculation for 20-mers for the human genome (available from GenBank at <http://www.ncbi.nih.gov/>) for all chromosomes in less than one hour.

## Calculation of frequency of presence of all possible $n$ -mers

```

FREQUENCY-OF-N-MERS-IN-TEXT(T, n)
1 for i ← 0 to 4n-1
2   do A[i] ← 0
3 sum ← 0
4 for j ← 0 to length(T)-n
5   do index=CONVERT_TO_INTEGER_VALUE(T(j, j+n-1))
6     if A[index]=0
7       then sum ← sum+1
8     A[index]=1
9 Use value of sum to reserve memory for arrays R and Q of size sum
10 counter ← 0
11 for i ← 0 to 4n-1
12   do if A[i]=1
13     then R[counter]=i
14     counter ← counter +1
15 for i ← 0 to sum
16   do Q[i] ← 0
17 for j ← 0 to length(T)-n
18   do index=CONVERT_TO_INTEGER_VALUE(T(j, j+n-1))
19     Q[R[index]]= Q[R[index]]+1
20 return Q, R

```

Here we use the fact that for  $4^n \gg m$  the majority of  $n$ -mers are absent from text, thus most of the counting array  $A$  will be filled with zeros (sparse). In fact, the number of different  $n$ -mers cannot be larger than  $m-n$ . We also cannot expect number of  $n$ -mers which appear more than once larger than  $(m-n)/2$ . Thus, to conveniently keep information about all present  $n$ -mers we need two arrays: one ( $Q$ ) to keep the "sequence" of  $n$ -mers and another ( $R$ ) to keep the integer number of times of appearance.

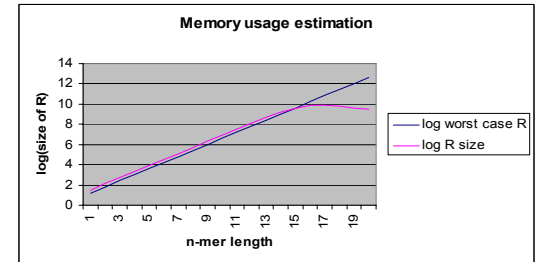
The estimation of the memory usage is straightforward. For the array  $R$  it is  $r_{integer}(m-n)/2 \approx r_{integer}m/2$ , where  $r_{integer}$  is the size reserved for the integer variable. For the array  $Q$  it is  $r_{(m-n)/2} \approx r_{m/2}$ , where  $r_{m/2}$  is the size reserved for the  $n$ -mer. In practice, memory usage is a lot less than worst case described above.

The Algorithm produces two synchronized arrays  $Q$  and  $R$  of dynamically defined size  $sum$ . The run time estimation of Algorithm 4 is  $O(4^n + m + 4^n + 4^n + m) = O(4^n + m)$ . Because the array  $Q$  of present  $n$ -mers appears to be sorted, this allows a logarithmic time for search  $O(\log(sum))$ , so that the worst case would be  $O(\log(m))$ .

$n$ -mer size	Number of different $n$ -mers $4^n$	Number of absent $n$ -mers		Number of $n$ -mers present only once		Number of $n$ -mers present more than once	
7	16,384	0	0.00%	0	0.00%	16,384	100.00%
8	65,536	0	0.00%	0	0.00%	65,536	100.00%
9	262,144	0	0.00%	0	0.00%	262,144	100.00%
10	1,048,576	0	0.00%	0	0.00%	1,048,576	100.00%
11	4,194,304	42	0.00%	324	0.01%	4,193,938	99.99%
12	16,777,216	42,501	0.25%	91,146	0.54%	16,643,569	99.20%
13	67,108,864	2,382,096	3.55%	2,642,582	3.94%	62,084,186	92.51%
14	268,435,456	41,634,971	15.51%	30,411,367	11.33%	196,389,118	73.16%
15	1,073,741,824	410,828,287	38.28%	166,998,278	15.55%	495,915,259	46.19%
16	4,294,967,296	2,717,880,983	63.28%	671,192,253	15.63%	905,694,060	21.09%
17	17,179,869,184	14,452,040,667	84.12%	1,790,043,813	10.42%	937,784,704	5.46%
18	68,719,476,736	65,147,357,575	94.80%	2,881,849,256	4.19%	650,229,905	1.00%
19	274,877,906,944	270,850,664,602	98.53%	3,538,156,028	1.29%	489,086,314	0.18%
20	1,099,511,627,776	1,095,257,688,530	99.61%	3,866,031,543	0.35%	387,907,703	0.04%

The presence/absence statistics for human genome of size 2,874,736,094 pairs of nucleotides. Calculation was performed using both (original and complementary) DNA sequences.

## Estimation of memory usage



Worst case memory usage for array  $R$  with size of  $R$  from Human genome in logarithmic scale. Note that due to extreme size of human genome the two estimators begin to differ only after 15-mers. In smaller genomes (microbials) these curves separate around  $n=11$ .

## Conclusion

The group of algorithms for the problem of finding appearances of all possible patterns of size  $n$  in a text or sequence of size  $m$  was presented. The operation count estimation  $O(4^n + m)$  makes it possible to accomplish the calculation of the statistics of the presence of all possible 7-20-mers ranging in size from viral to human genome. By using the concept of a counting array and parallel processing we can go to much higher lengths.

## Acknowledgements

Authors acknowledge the NIH for partial support and NPACI for computational support.