Lexicographic unranking of combinations revisited

Antoine Genitrini, Martin Pépin

LEXICOGRAPHIC UNRANKING OF COMBINATIONS REVISITED

ANTOINE GENITRINI AND MARTIN PÉPIN

Abstract. We propose a comparative study of four algorithms dedicated to the lexicographic unranking of combinations. Three of them are algorithms from the literature. We analyze their time complexity in average, with a uniform presentation, and describe their strengths and weaknesses. Furthermore we also introduce a new algorithm using a new strategy of computations inside the classical factorial numeral system (or factoradics). Then after proposing improvements for all implementations we present a detailed complexity analysis whose results are validated by an experimental analysis for actual use of combination unranking. Interestingly we show that even if the algorithms are based on different strategies all are doing very similar computations. Finally successfully apply our approach to the unranking of other classical combinatorial objects.

Keywords: Unranking Algorithm; Combination; Lexicographic Order; Complexity Analysis.

Contents

Date: December 4, 2020.

1. Introduction

One of the most fundamental combinatorial object is called combination. It consists of a selection of items from a collection. In many enumerating problems it appears either as the main combinatorial structure, or as a core fundamental block because of its simplicity and counting characteristics.

In the 60s while resolving some optimization problem about scheduling Lehmer rediscovered an important property linking natural numbers with a mixed radius numeral system based on combinations. This relation gave him the possibility to exhibit some greedy approach for a ranking algorithm that transforms (bijectively) a combination into an integer. This numeral system is now commonly called "combinatorial numbers system" or "combinadics". It is often used for the reverse of Lehmer's problem: generating the u-th combination (for a given order on the set of combinations). For efficiency reason this approach can be substituted to exhaustive generation once the latter is not possible anymore due to the *combinatorial explosion* of the number of objects when their size increases. In the context of combination, the explosion appears quickly: we recall $\binom{2n}{n} \sim (2\pi n)^{-\frac{1}{2}} 4^n$. This generation strategy of a single element is classically called an *unranking* method. It is today often used as a basic brick in scheduling problems [21] but also e.g. in software testing [17].

In order to unrank elements one must first define an order over these elements. The one that is usually used is the *lexicographic*. The lexicographic order is humanly easy to handle, and thus has been extensively studied. But, as Ruskey [20, p. 59] mentions, lexicographic generation is usually not the most efficient, thus a particular care must be taken while unranking for this order.

The classical approach for the construction of combinatorial structures presenting a recursive decomposition schema consists in taking advantage of this decomposition in order to build a bigger object from a smaller one. The method has been extensively detailed in the famous book of Nijenhuis and Wilf [18]. There, the authors are interested in an exhaustive generation and a uniform random sampling approach, but some ideas about the decomposition schema are also applicable in the context of unranking. The method has been then applied generically to decomposable objects in the sense of analytic combinatorics, first in the context of recursive generation [12], and then in the context of unranking approaches [15].

Aside such generic approaches there exist several *ad hoc* algorithms. The complexity analysis of these algorithms have been settled to be linear in n in average over all possible combinations when k is ranging from 0 to n. But to the best of our knowledge these complexity analyses are only computing the number of calls to the function that computes a binomial coefficient while having first all the possible coefficients pre-computed and stored (this pre-computation step is not included in the complexity analysis). From this fact, two questions arise. First, is this complexity analysis relevant? That is: does it reflect the actual runtime of the algorithms and can we afford the pre-computations of many binomial coefficients? And second, among the different existing algorithms, which one performs best in practice?

Using exact computations, it is actually not a problem to deal with combinations over sets of several thousands of objects. In this context using a table filled with all possible binomial coefficients that might be needed is not practical. Most classical computer algebra systems (CAS) can unrank combinations in a reasonable time though, which suggests that there are better approaches. In the following Table 1 we present some of our experimental results. We will detail everything in Section 4 but as a foretaste here we give some key points. In Section 2 we introduce a new unranking algorithm. The first column gives the typical run time of our C implementation of this algorithm. In the other columns, we give a rough performance comparison of four different CAS. For each one, we have compared the average run time in milliseconds to unrank a combination first by using the native algorithm of the CAS ("their algo.") and second by implementing our new algorithm in the high-level programming language of the CAS. The high diversity of the time used by the algorithms naturally militates in favor of providing a detailed analysis of the different methods that are used in practice.

There is a special case for Sagemath: since the version 9.1 our algorithm has been implemented (in Python 3) and is used as the native algorithm.

Implem. \ Sample	Our algo. in C	Sagemath v. 9.0 their algo.	Sagemath v. 9.2 new algo.	Maple v. 2020.0 their algo.	Maple v. 2020.0 our algo.	Mathematica v. 12.1.1.0 their algo.	Mathematica v. 12.1.1.0 our algo.	Matlab v. R2020b their algo.	Matlab v. R2020b our algo.
$n = 1,000$ $k = 100$	0.05464	2.6045	2.9672	78.2	2.12	0.44176	4.3145	3996.6	3041.2
$n = 1,000$ $k = 500$	0.06052	8.8903	2.4784	614	2.96	0.34608	3.9547	3520.6	3380.0
$n = 3,000$ $k = 300$	0.17496	15.8968	8.7929	1180	13.2	5.9131	11.823	11846	9315.2
$n = 3,000$ $k = 1,500$	0.27524	96.3589	8.0500	6130	19.2	4.9624	13.067	11087	9879.4
$n = 10,000$ $k = 1,000$	1.2554	191.03	31.665	too long	65.1	21.906	39.935	too long	too long
$n = 10,000$ $k = 5,000$	2.3849	2245.6	29.027	too long	97.9	29.916	46.452	too long	too long

Time in ms.

TABLE 1. Average time (in ms.) for the unranking of a combination among n, k

Along the paper, we represent combinations as follows.

Definition 1. Let n and k be two integers with $0 \leq k \leq n$. We represent a combination of k elements among n elements denoted by $\{0, 1, \dots, n-1\}$ as a finite sequence containing k distinct elements increasingly sorted from left to right.

For example, let n and k be respectively 5 and 3. The finite sequences $(0, 1, 2)$ and $(0, 2, 4)$ are combinations of k among n, but $(0, 2, 1)$ and $(0, 1, 2, 3)$ are not. There are another possible representations, notably by using a 2-letters alphabet, but we stick to the one given in Definition 1 throughout this paper.

There are several orders for comparing combinations. In the following we restrict our attention to orders comparing combinations of the same length, i.e. the same number of elements.

Definition 2. Let $A = (a_0, a_1, \dots, a_{k-1})$ and $B = (b_0, b_1, \dots, b_{k-1})$ be two distinct combinations of k elements among n.

- In the *lexicographic* order, we say that A is smaller than B if and only if both combinations have the same prefix (eventually empty) such that $(a_0, \dots, a_{p-1}) = (b_0, \dots, b_{p-1})$ and if in addition $a_p < b_p$.
- In the *co-lexicographic* order, we say that A is smaller than B if and only if the finite sequence $(a_{k-1}, \dots, a_0)$ is smaller than $(b_{k-1}, \dots, b_0)$ for the lexicographic order.
- An order being given such that A is smaller than B, then, for the *reverse* order, B is smaller that A.

Definition 3. Let $0 \leq k \leq n$ be two integers and let A be a combination of k elements among n. For a given order, the *rank* u of A belongs to $\{0, 1, \dots \binom{n}{k} - 1\}$ and is such that A is the u-th smallest combination.

With these definitions in mind, we can enter the core of the paper organized as follows. We first give the presentation and a first complexity analysis of our new algorithm in Section 2. Section 3 is dedicated to the survey of three classical algorithms. The first one is the algorithm based on the so-called recursive method, solving the lexicographic combination unranking problem. The two others algorithms are based on combinadics that correspond to a specific numeral system. It seems that it is the first time both are compared and the reason why one is better is explained. We also propose a new method for their analysis based on the now classical generating function approach.

Obviously we thus reprove their average complexity results but with more details. We then, in Section 4 compare their efficiency in some experiments and recall a classical way to improve such kind of combinatorial algorithms and thus apply it to the first codes we have presented. Surprisingly, once the improvements have been implemented in all algorithms, we observe deep similarities in the computations of all algorithms conducted during the unranking. Finally we extend our approach to solve the problem of unranking structures enumerated by multinomial coefficients and also objects counted by the k-permutations of n (also called arrangements).

This article is a long and extended version of the unpublished paper [7]. The implementation and the exhaustive material used for repeating the experiments are all available at `http://github.com/Kerl13/combination_unranking`.

2. Unranking through factoradics: a new strategy

The classical methods to unrank combination are relying on the combinatorial number system introduced in 1887, by E. Pascal [19] and later by D. H. Lehmer (detailed in the book [1, p. 27]). We survey these classical algorithms in the Section 3.2. But here we present a new strategy based on another number system that has never been used, to the best of our knowledge, for the unranking combinations: factoradics. The factorial number system, or factoradics, is a mixed radix numeral system in which the representation of integers relies on the use of factorial numbers.

Fact 4. *For all positive integers u, we define n as the unique integer satisfying $(n-1)! \leq u < n!$. Then there exists a unique sequence of integers $(f_\ell)_{\ell \in \{0,\dots,n-1\}}$, with $0 \leq f_\ell \leq \ell$, for all ℓ such that:*

$$u = f_0 \cdot 0! + f_1 \cdot 1! + \cdots + f_{n-2} \cdot (n-2)! + f_{n-1} \cdot (n-1)!$$

The finite sequence $(f_0, f_1, \dots, f_{n-1})$ is called the factoradic decomposition *of u (note that $f_0 = 0$ for all u).*

Take the number $u = 2021$ as an example, we obtain the following decomposition: $2021 = 0 \cdot 0! + 1 \cdot 1! + 2 \cdot 2! + 0 \cdot 3! + 4 \cdot 4! + 4 \cdot 5! + 2 \cdot 6!$, thus its factoradic is $(0, 1, 2, 0, 4, 4, 2)$.

Definition 5. Let n be a positive integer. A permutation of size n is an ordering of the elements from the set $\{0, 1, \dots, n-1\}$.

We represent a permutation of size n as a finite sequence of length n indicating the order of its elements. For example the sequence $(2, 4, 0, 3, 1)$ is a permutation of size 5.

The factorial number system is particularly suitable to define a one-to-one correspondence between integers and permutations and thus can be used as an unranking method for permutations. The algorithm implemented in the function UnrankingPermutation in Algorithm 1 is a straightforward adaptation of the Fisher-Yates random sampler for permutations [10].

Fact 6. *For all $0 \leq u < n!$,* UnrankingPermutation(n, u) *returns the u-th permutation in lexicographic order among the $n!$ permutations of n elements.*

Algorithm 1 Unranking a permutation

function UnrankingPermutation(n, u)
  $F \leftarrow$ `factoradic`(u)
  while `length`$(F) < n$ **do**
    `append`$(F, 0)$
  return Extract(F, n, n)

function Extract(F, n, k)
  $P \leftarrow [0, 1, \dots, n-1]$
  $L \leftarrow [0, \dots, 0]$ ▷ k components
  for i **from** 0 **to** $k-1$ **do**
    $L[i] \leftarrow P[F[n-1-i]]$
    `remove`$(P, F[n-1-i])$
  return L

`factoradic`(u): computes the factoradic of u;
`length`(F): computes the number of components in F;
`append`(F, i): appends the element i at the end of F;
`remove`(F, i): removes from F the element at index i.

Since the factoradic (with 8 components) of 2021 is $(0, 1, 2, 0, 4, 4, 2, 0)$, the permutation (of size 8) of rank 2021 is $(0, 3, 6, 7, 1, 5, 4, 2)$. To reach this permutation, we read the factoradic from right to left, and extract iteratively from the list $(0, 1, \dots, n-1)$ the element whose index is the coefficient read in the factoradic. This goes on until the list is empty and we reach the leftmost component of the factoradic. Thus, in our example we start by extracting the element of index 0, which is 0. Then the list P becomes $(1, 2, \dots, 7)$ and we extract the element of index 2, which is 3. Then P becomes $(1, 2, 4, 5, 6, 7)$ and we extract the 4-th element which is 6, and so on.

Note that for the sake of clarity we presented the function EXTRACT using a list for P, but a better data structure must be used in order to achieve good performance. Good candidates are dynamic balanced trees as presented in [3], or multisets with elements of weight 1 or 0 as presented in the appendix of [2], since both provide logarithmic access and removal. Unfortunately it seems that there is no algorithm based on some `swap` operation giving an in-place shuffle to unrank permutation in the lexicographic order, put differently: Durstenfeld's algorithm [8] cannot be easily adapted for the lexicographic order.

We now turn to the *unranking of a combination through factoradics*. The basic ideas driving our algorithm are the following:

(1) we define a bijection between the combinations of k elements among n and a subset of the permutations of n elements;
(2) we transform the combination rank u into the rank u' of the appropriate permutation;
(3) we build (the prefix of) the permutation of rank u' by using Algorithm 1.

Definition 7. Let n and k be two integers with $0 \le k \le n$. We define $\mathcal{P}_{n,k}$ as the application which maps the combination $(\ell_0, \ell_1, \dots, \ell_{k-1})$ to the size-n permutation $(\ell_0, \ell_1, \dots, \ell_{k-1}, d_k, \dots, d_{n-1})$ where the integers d_i are such that $d_k < d_{k+1} < \dots < d_{n-1}$ and $\{\ell_0, \dots, \ell_{k-1}, d_k, \dots, d_{n-1}\} = \{0, 1, \dots, n-1\}$.

Thus, by definition, for $n = 5$ and $k = 3$, the permutations associated to the combinations $(0, 1, 2)$ and $(0, 2, 4)$ are respectively $(0, 1, 2, 3, 4)$ and $(0, 2, 4, 1, 3)$. Note that the application $\mathcal{P}_{n,k}$ returns the smallest size-n permutation for the lexicographic order whose prefix is the given combination.

Proposition 8. *For all integers $0 \le k \le n$, the application $\mathcal{P}_{n,k}$ is a bijection from the set of (n, k) combinations to the set of size-n permutations whose prefix of length k and suffix of length $n-k$ are both increasingly sorted.*

Remark that the permutation (of size 5) $(0, 1, 2, 3, 4)$ is the permutation associated to combinations $(0, 1)$ and $(0, 1, 2)$ by respectively $\mathcal{P}_{2,5}$ and $\mathcal{P}_{3,5}$. In fact, there are exactly 6 combinations associated to the latter permutation, but for different values of k. The proof of Proposition 8 is straightforward.

Fact 9. *For any integers $m \ge 0$, the number of sequences $(f_i)_{0 \le i < n}$ satisfying $n-k \ge f_{n-k} \ge f_{n-k+1} \ge \dots \ge f_{n-1} \ge m$ is given by $\binom{n-m}{k}$.*

In fact, we get the result using a cardinality argument: a sequence of integers of the form $x_0 = 0 \le x_1 \le x_2 \le \dots \le x_k \le x_{k+1} = m$ corresponds to a weak composition (consider the differences $x_{i+1} - x_i$) of the integer m into $k+1$ terms. The number of such compositions is given by $\binom{m+k}{k}$. Hence, the number of sequences matching the description of Fact 9 is given by $\binom{n-k+k}{k}$.

We now exhibit how to transform a combination rank into its corresponding permutation rank.

Lemma 10. *For any given $0 \le k \le n$, the factoradic decompositions of the ranks of the permutations obtained as the image by $\mathcal{P}_{n,k}$ of some combination of k elements among n are all the finite sequences of the form $(0, \dots, 0, f_{n-k}, \dots, f_{n-1})$ with $n-k \ge f_{n-k} \ge f_{n-k+1} \ge \dots \ge f_{n-1} \ge 0$.*

Proof. Let u be an integer whose factoradics is $(0, \dots, 0, f_{n-k}, \dots f_{n-1})$ as in the Lemma. Due to the constraint $n-k \ge f_{n-k} \ge f_{n-k+1} \ge \dots \ge f_{n-1} \ge 0$, the permutation corresponding to u has for prefix of length k the sequence $(f_{n-1}, f_{n-2}+1, f_{n-3}+2, \dots, f_{n-k}+k-1)$ which is increasingly sorted. The rest of the permutation (the suffix of length $n-k$) corresponds to the increasing sequence of elements that have not been taken yet. Thus the result corresponds to a combination via $\mathcal{P}_{n,k}$.

Fact 9 completes the proof. □

So in order to convert the combination rank u into its corresponding permutation rank u', it is sufficient to find the u-th sequence satisfying Lemma 10 in *co-lexicographic order*. This is presented in Algorithm 2 where the RankConversion function implements the conversion from u to u' and the UnrankingCombination function implements the whole unranking procedure.

The key to the rank conversion is also Fact 9 As a consequence, we get that the first $\binom{n-1}{k-1}$ such sequences (in col-lexicographic order) have $f_{n-1} = 0$ and that the $\binom{n-1}{k}$ following have $f_{n-1} \geq 1$.

Algorithm 2 Unranking a combination

```
1: function UnrankingCombination(n, k, u)
2:     u' ← RankConversion(n, k, u)
3:     p ← UnrankingPermutation(n, u')
4:     return the first k elements of p
```

```
1: function RankConversion(n, k, u)
2:     F ← [0, ..., 0]                  ▷ n components in F
3:     i ← 0
4:     m ← 0
5:     while i < k do
6:         b ← binomial(n − 1 − m − i, k − 1 − i)
7:         if b > u then
8:             F[n − 1 − i] ← m
9:             i ← i + 1
10:        else
11:            u ← u − b
12:            m ← m + 1

13:            ▷ F is the factoradic decomposition
14:    return composition(F)
```

`binomial`(n,k) computes the value of $\binom{n}{k}$;
`composition`(F): computes the integer whose factoradic is F.

Once again we opted for a simple presentation here where the rank conversion and the unranking part of the algorithm are clearly separated. There is much room for improvement here: for instance, note that at the end of the function RankConversion a factoradic decomposition is transformed into the integer it represents, but then at the beginning of UnrankingPermutation this integer will be decomposed again in factoradics. In fact, instead of storing of storing m into F at line 8, we could directly compute the i-th component of the combination as $m+i$ by using the remark at the beginning of the proof of Lemma 10. In Section 4.2 we provide a more efficient way to implement the algorithm including this optimization among other things.

Proposition 11. *The function* UnrankingCombination(n,k,u) *computes the u-th combination of k elements among n in lexicographic order.*

Proof. The algorithm, and thus its proof, relies heavily on Fact 9. The key to prove the correctness of the RankConversion function is the following loop invariant:

- the values of f_{n-1-j} for all $0 \leq j < i$ have been computed and stored in F;
- the value of f_{n-i-1} (which has not been determined yet, as we enter the loop) is at least m;
- the variable u' holds the rank of the sequence $(f_j)_{0 \leq j < n-i}$ (note that $j < i$) among all sequences satisfying the condition of Fact 9 with $k = k - i$ and $n = n - i$.

With this invariant at hand, the combinatorial argument behind the condition $u < \binom{n-m-i-1}{k-i-1}$ becomes more apparent: the binomial coefficient counts the number of such sequences ending with m. Hence if $u < \binom{n-m-i-1}{k-i-1}$ then $f_{n-1-i} = m$ and we move to the evaluation of the next coefficient (f_{n-i-2}), otherwise we try the next value of m. □

The usual way to evaluate the efficiency of such algorithms is to count the number of times the function `binomial` is called (see e.g. the book [20, p. 66] or the papers [5, 9]). During the conversion from the rank of the combination to the one of the associated permutation, the coefficients are obtained via trials (in the factoradic notation) for f_{n-1} to f_{n-k}, remarking that through our bijection $\mathcal{P}_{n,k}$ the latter sequence is weakly increasing. Thus the worst cases are obtained when the value f_{n-k} is as large as possible, that is $n-k$. Thus for such elements, the number of calls to `binomial` is n.
For the average number of calls to the function `binomial`, unranking all combinations u when it describes the whole range from 0 to $\binom{n}{k}-1$, we introduce the following cumulative sequence.

Lemma 12. *Let $u_{n,k}$ be the cumulative numbers of calls to `binomial` while unranking all possible combinations u from 0 to $\binom{n}{k}-1$. The sequence satisfies: $u_{n,0}=0$ and $u_{n,n+i}=0$ for all n and $i>0$, and otherwise*

$$u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}.$$

In Table 2 you get the first values for $u_{n,k}$ when n is less than 9. We obtain a sequence stored under the reference `OEIS A127717`[1]. The bijection between both structures is direct, and thus we have new information about this sequence in the following.

$n \backslash k$	0	1	2	3	4	5	6	7	8
1	0	1							
2	0	3	2						
3	0	6	8	3					
4	0	10	20	15	4				
5	0	15	40	45	24	5			
6	0	21	70	105	84	35	6		
7	0	28	112	210	224	140	48	7	
8	0	36	168	378	504	420	216	63	8

TABLE 2. First values of $u_{n,k}$ for $n = 1..8$ and $k = 0..n$ (Algorithm 2)

Proof. The recurrence can be observed by unrolling the first iteration of the while loop. In the first iteration of the loop, a binomial coefficient b is always computed (regardless the value of k and n) which accounts for the term $\binom{n}{k}$ in the recurrence relation. Then, for all the ranks u such that $u<b$ we choose $f_{n-1}=0$ and increment i, so that rest of the execution corresponds to unranking a combination of $k-1$ elements among $n-1$. This is accounted by $u_{n-1,k-1}$. Conversely, for all ranks u such that $u \geq b$, the value of m is incremented and the rest of the execution corresponds to unranking an combination of k elements among $n-1$, which is accounted by $u_{n-1,k}$. □

We turn to bivariate generating function to encode the sequence $(u_{n,k})$. The reader can for example refer to the two books of Flajolet and Sedgewick [13, 11] for a global presentation of such method.

Theorem 13. *Let $U(z,y)$ be the generating functions associated to $(u_{n,k})$. Then*

$$U(z,y) = \frac{1}{1-z-zy}\left(\frac{1}{1-z-zy} - \frac{1}{1-z}\right); \text{ thus}$$
$$u_{n,k} = \binom{n}{k}\frac{k}{k+1}(n+1).$$

[1]Throughout this paper, a reference `OEIS A···` points to Sloane's Online Encyclopedia of Integer Sequences `www.oeis.org`.

Proof. The first step of the proof consists in exhibiting the ordinary generating function associated to $U(z, y)$. In order to obtain a functional equation satisfied by U, we start from the result presented in Lemma 12. The extreme cases are $u_{n,0} = 0$ and $u_{n,n+i} = 0$ for all n and $i > 0$. And the recursive equation is $u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}$.

We remark the constant $\binom{n}{k}$ in the equation. We thus need the bivariate generating function for binomial coefficient. Le us denote it by $B(z, y)$, it is equal to

$$B(z,y) = \sum_{n\geq 0}\sum_{k=0}^{n}\binom{n}{k} z^n\ y^k = \frac{1}{1-z-zy}.$$

In order to take into account the extreme cases, we must remove the terms corresponding to $k = 0$:

$$\tilde{B}(z,y) = \frac{1}{1-z-zy} - \frac{1}{1-z}.$$

By summing both sides of the recursive relation and by taking care of the extreme cases we get:

$$\sum_{n\geq 1}\sum_{k=1}^{n} u_{n,k} z^n\ y^k = \tilde{B}(z,y) + \sum_{n\geq 1}\sum_{k=1}^{n} u_{n-1,k-1} z^n\ y^k + \sum_{n\geq 1}\sum_{k=1}^{n} u_{n-1,k} z^n\ y^k$$
$$U(z,y) = \tilde{B}(z,y) + z\ y\ U(z,y) + z\ U(z,y).$$

We thus deduce

$$U(z,y) = \frac{1}{1-z-zy}\left(\frac{1}{1-z-zy} - \frac{1}{1-z}\right).$$

The second step in the proof consists in extracting the coefficient $u_{n,k}$. We rewrite $U(z, y)$ as:

$$\begin{aligned} U(z,y) &= \frac{1}{1-z(1+y)}\left(\frac{1}{1-z(1+y)} - \frac{1}{1-z}\right) \\ &= \left(\sum_{r\geq 0} z^r(1+y)^r\right)\cdot\left(\sum_{r\geq 0} z^r(1+y)^r - \sum_{r\geq 0} z^r\right) \\ &= \left(\sum_{r\geq 0} z^r(1+y)^r\right)\cdot\left(1-1+\sum_{r\geq 1} z^r\left((1+y)^r-1\right)\right). \end{aligned}$$

By extraction the coefficient in front of z^n:

$$\begin{aligned} [z^n]U(z,y) &= \sum_{\ell=0}^{n-1}(1+y)^\ell\left((1+y)^{n-\ell}-1\right) = \sum_{\ell=0}^{n-1}(1+y)^n - (1+y)^\ell \\ &= n(1+y)^n - \frac{(1+y)^n-1}{y}. \end{aligned}$$

The latter result corresponds to the distribution of the costs when k varies from 0 to n. We can then exactly extract the coefficient of $z^n\ y^k$:

$$[z^n\ y^k]U(z,y) = n\cdot\binom{n}{k} - \binom{n}{k+1} = \binom{n}{k}\frac{k}{k+1}(n+1).$$

□

Corollary 14. *In order to unrank a combination of* k *elements among* n*, the function* UnrankingCombination$(n, k, \cdot)$ *needs in average* $u_{n,k}/\binom{n}{k}$ *calls to the function* `binomial`*. For* n *being large and* k *being of the form* αn *for* $0 < \alpha < 1$*, the average number of calls is*

$$\frac{u_{n,k}}{\binom{n}{k}} \underset{\substack{n\to\infty \\ k=\alpha n}}{=} n + 1 - \frac{1}{\alpha} + O\left(\frac{1}{n}\right).$$

The result is direct by using Theorem 13. Since we have the exact value of $u_{n,k}$ the mean value can easily be computed in other cases like $k = o(n)$.

3. Classical unranking algorithms

We present now a survey of the usual approaches to unrank combinations. The motivation behind this section is threefold. First the classical algorithms are old, they have been developed in the 70's and 80's and it is a hard task to get access to the papers we will mention. Second, although they have been analyzed according to the number of calls to the binomial coefficient computations we present here a standardization of the analysis using generating functions like in the previous section. Finally, as we will see in Section 4 and in the conclusion of the paper a detailed analysis of all the possible approaches is necessary to well understand the behaviors of the computations.

3.1. **Unranking through the recursive method.** We are dealing with a combinatorial structure here, combinations, that is very well understood in the combinatorial sense. Thus when trying to develop an unranking algorithm the first idea consists in developing one based on the classical recursive generation method presented in [18]. This type of algorithm is based on a recursive decomposition of the structure into smaller parts. Here, this idea is to use the following fact: a combination of k elements among $[\![0; n-1]\!]$ either contains $n-1$ or does not. In the first case, the rest of the combination can be seen as a combination of $k-1$ elements among $n-1$ and in the second case the combination is a combination of k elements among $n-1$. From a counting point of view, this translates into the well-known identity $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ and from an unranking point of view, this translates into Algorithm 3.

Algorithm 3 Recursive Unranking

```
1: function UnrankingRecursive(n, k, u)
2:     L ← RecGeneration(n, k, u)
3:     L' ← [0, ..., 0]                ▷ k components
4:     for i from 0 to k − 1 do
5:         L'[i] ← n − 1 − L[k − 1 − i]
6:     return L'
```

```
1: function RecGeneration(n, k, u)
2:     if k = 0 then
3:         return []
4:     if n = k then
5:         return [0, 1, 2, ..., k − 1]
6:     b ← binomial(n − 1, k − 1)
7:     if u < b then
8:         R ← RecGeneration(n − 1, k − 1, u)
9:         append(R, n − 1)
10:        return R
11:    else
12:        return RecGeneration(n−1, k, u−b)
```

Remark 15. An alternative choice would have been to test whether $b < \binom{n-1}{k}$ at line 6. This corresponds to putting first the combinations that do not contain $n-1$ and then those that contain $n-1$. In this case the unranking order is different but the performance are similar.

Proposition 16. *The function* RecGeneration(n, k, u) *computes the u-th combination for of k elements among n in reverse co-lexicographic order.*

Corollary 17. *The function* UnrankingRecursive(n, k, u) *computes the u-th combinations of k elements among n in lexicographic order.*

Proof. The proposition is proved by induction and the corollary is a direct observation given in [14, p. 47]. □

Here again, we are interested in the average number of calls to the function `binomial`, when u describes the whole range of integers from 0 to $\binom{n}{k} - 1$. Ruskey justifies such a measure by supposing the table of all binomial coefficients precomputed, thus each call is equivalent. Later, in Section 4 we will discuss this measure. We introduce the sequence[2] $u_{n,k}$ equal to the cumulative number of calls to `binomial` for the whole range of possible values for u.

[2] In order to simplify the notations we use several times the notations $u_{n,k}$ and $U(z)$ for distinct sequences and series.

Lemma 18. *Let $u_{n,k}$ be the cumulative number of calls to* `binomial` *while unranking all possible u from 0 to $\binom{n}{k}-1$. The sequence satisfies: $u_{n,0}=0$ and $u_{n,n+i}=0$ for all n and $i \geq 0$ and otherwise:*

$$u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}.$$

Proof. If $0 < k < n$, calling RecGeneration(n,k,u) incurs one call to binomial and a recursive call. The cumulative cost of the first call to binomial is $\binom{n}{k}$, the cumulative cost of the recursive calls for $u < \binom{n-1}{k-1}$ is $u_{n-1,k-1}$ and the cumulative costs of the recursive calls for $u \geq \binom{n-1}{k-1}$ is $u_{n-1,k}$. □

$n \backslash k$	0	1	2	3	4	5	6	7	8
1	0	0							
2	0	2	0						
3	0	5	5	0					
4	0	9	16	9	0				
5	0	14	35	35	14	0			
6	0	20	64	90	64	20	0		
7	0	27	105	189	189	105	27	0	
8	0	35	160	350	448	350	160	35	0

Table 3. First values of $u_{n,k}$ for $n = 1..8$ and $k = 0..n$ (Algorithm 3)

Theorem 19. *Let $U(z,y)$ be the ordinary generating function associated to $(u_{n,k})$, such that $U(z,y) = \sum_{n\geq 0}\sum_{k=0}^{n} u_{n,k}\, y^k\, z^n$. Then,*

$$U(z,y) = \frac{1}{1-z-zy}\left(\frac{1}{1-z-zy} - \frac{1}{1-z} - \frac{zy}{1-zy}\right), \text{ thus,}$$
$$u_{n,k} = \binom{n}{k}\, k \left(\frac{n+1}{k+1} - \frac{1}{n-k+1}\right).$$

The proof of Theorem 19 is very similar to the one of Theorem 13.

Proof. The first step of the proof consists in exhibiting the ordinary generating function associated to $U(z,y)$. In order to obtain the equation for U, we start from the result presented in Lemma 18. The extreme cases are $u_{n,0}=0$ and $u_{n,n+i}=0$ for all n and $i \geq 0$. And the recursive equation is $u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}$.

Again, the numbers $\binom{n}{k}$ appear in the equation, thus we need the bivariate generating function of binomial coefficients. Le us denote it by $B(z,y)$. It satisfies:

$$B(z,y) = \sum_{n\geq 0}\sum_{k=0}^{n}\binom{n}{k} z^n\, y^k = \frac{1}{1-z-zy}.$$

In order to follow the extreme cases, we must remove the first column $k=0$ and the diagonal $k=n$:

$$\tilde{B}(z,y) = \frac{1}{1-z-zy} - \frac{1}{1-z} - \frac{zy}{1-zy}.$$

By summing the recursive equation and taking care of the extreme cases we get:

$$\sum_{n\geq 1}\sum_{k=1}^{n} u_{n,k} z^n\, y^k = \tilde{B}(z,y) + \sum_{n\geq 1}\sum_{k=1}^{n} u_{n-1,k-1} z^n\, y^k + \sum_{n\geq 1}\sum_{k=1}^{n} u_{n-1,k} z^n\, y^k$$
$$U(z,y) = \tilde{B}(z,y) + z\, y\, U(z,y) + z\, U(z,y).$$

We thus deduce

$$U(z,y) = \frac{1}{1-z-zy}\left(\frac{1}{1-z-zy} - \frac{1}{1-z} - \frac{zy}{1-zy}\right).$$

The second step in the proof consists in extracting the coefficient $u_{n,k}$.

$$\begin{aligned}
U(z,y) &= \frac{1}{1-z(1+y)}\left(\frac{1}{1-z(1+y)} - \frac{1}{1-z} - \frac{zy}{1-zy}\right)\\
&= \left(\sum_{r\geq 0} z^r(1+y)^r\right)\cdot\left(\sum_{r\geq 0} z^r(1+y)^r - \sum_{r\geq 0} z^r - \sum_{r\geq 1} z^r y^r\right)\\
&= \left(\sum_{r\geq 0} z^r(1+y)^r\right)\cdot\left(1 - 1 + \sum_{r\geq 1} z^r\left((1+y)^r - 1 - y^r\right)\right).
\end{aligned}$$

By extraction the coefficient in front of z^n:

$$\begin{aligned}
[z^n]U(z,y) &= \sum_{\ell=0}^{n-1}(1+y)^\ell\left((1+y)^{n-\ell} - 1 - y^{n-\ell}\right)\\
&= \sum_{\ell=0}^{n-1}(1+y)^n - (1+y)^\ell - y^{n-\ell}(1+y)^\ell\\
&= n(1+y)^n - \frac{(1+y)^n - 1}{y} - y(1+y)^n + y^{n+1}.
\end{aligned}$$

The latter result corresponds to the distribution of the costs when k varies from 0 to n. We can then extract the coefficient of $z^n\ y^k$:

$$\begin{aligned}
[z^n\ y^k]U(z,y) &= n\cdot\binom{n}{k} - \binom{n}{k+1} - \binom{n}{k-1}\\
&= \binom{n}{k}\left(n - \frac{n-k}{k+1} - \frac{k}{n-k+1}\right)
\end{aligned}$$

which completes the proof. □

This sequence $u_{n,k}$ is a shifted version of the sequence stored under the reference `OEIS A059797`. We thus can complete the properties in `OEIS` using our results.

Due to the values of the extreme cases when $k=0$ and $k=n$ and the symmetry in the recurrence we obviously obtain the fact that $u_{n,k} = u_{n,n-k}$, reflecting the symmetry of the binomial coefficients.

Corollary 20. *The function* UNRANKINGRECURSIVE$(n,k,\cdot)$ *needs in average* $u_{n,k}/\binom{n}{k}$ *calls to the function* ***binomial***. *For n being large and k being of the form $\alpha\cdot n$ for $0<\alpha<1$, we get:*

$$\frac{u_{n,k}}{\binom{n}{k}} \underset{\substack{n\to\infty\\ k=\alpha n}}{=} n + 2 - \frac{1}{\alpha(1-\alpha)} + O\left(\frac{1}{n}\right).$$

Note that for this algorithm too, the average complexity is only below the worst-case complexity by a constant (when $k\sim\alpha n$).

Naturally, in order to be able to handle large values of n and k, a tail-recursive variant of this algorithm, or an iterative version should be preferred over the straightforward implementation. In the latter strategy, the recursive approach is a drawback for some programming languages that do not handle recursion efficiently (due to the depth of the stack). In fact, today the computer are able to handle combinations for very big values of n and k thus the recursive approach should be avoided. Naturally other strategies have been suggested in the literature.

3.2. **Unranking through combinadics.** In 1887, E. Pascal [19] and later D. H. Lehmer (detailed in the book [1, p. 27]) presented an interesting way to decompose a natural number, in what we call today a mixed radix numeral system. In their case it is the combinatorial number system, or combinadics. The decomposition relies on binomial coefficients.

Fact 21. *Let $n \geq k$ be two positive numbers. For all integers u, with $0 \leq u < \binom{n}{k}$, there exists a unique sequence $0 \leq c_1 < c_2 < \cdots < c_k < n$ such that*[3]

$$u = \binom{c_1}{1} + \binom{c_2}{2} + \cdots + \binom{c_{k-1}}{k-1} + \binom{c_k}{k}.$$

The finite sequence $(c_1, \ldots, c_k)$ is called the combinadic *of u.*

For example when $n = 5$ and $k = 3$, the number 8 is represented as $\binom{1}{1} + \binom{3}{2} + \binom{4}{3}$, thus the combinadic of 8 is $(1, 3, 4)$. In the following Table 4 we present for various values of u the combinadic of u and the combination of rank u for $n = 6$ and $k = 2$. Here we observe that the exhibited ranking is co-lexicographic and that the combination of rank u can be deduced from the combinadic of u by reversing it and applying the transformation $x \mapsto n - 1 - x$ to each of its components.

rank	reverse rank	`combinadic`	`combination`
0	14	(4, 5)	(0, 1)
1	13	(3, 5)	(0, 2)
2	12	(2, 5)	(0, 3)
3	11	(1, 5)	(0, 4)
4	10	(0, 5)	(0, 5)
5	9	(3, 4)	(1, 2)
6	8	(2, 4)	(1, 3)
7	7	(1, 4)	(1, 4)
8	6	(0, 4)	(1, 5)
9	5	(2, 3)	(2, 3)
10	4	(1, 3)	(2, 4)
11	3	(0, 3)	(2, 5)
12	2	(1, 2)	(3, 4)
13	1	(0, 2)	(3, 5)
14	0	(0, 1)	(4, 5)

TABLE 4. Combinadics and their combination for $n = 6$ and $k = 2$

In 2004, using this representation, McCaffrey exhibited in the MSDN article [16], an algorithm to build the u-th element (in lexicographic order) of the combinations of k elements among n. But in fact, this algorithm was already published in [14, p. 47] and can also be seen as an extension of the work of Lehmer. This algorithm is interesting in the sense that it corresponds to the previous implementation used in the mathematics software system `Sagemath` [22][4]. In the beginning of 2020, we replaced the Sagemath implementation by the algorithm presented in Section 2 [5].

The algorithm simply performs the combinadic decomposition of u and then applies the aforementioned transformation. The idea to get the combinadic of an integer $0 \leq u < \binom{n}{k}$ is the following: c_k is obtained as the maximum integer such that $u \geq \binom{c_k}{k}$, then the remaining part $u - \binom{c_k}{k}$ is smaller than $\binom{n-1}{k-1}$ so its can be decomposed recursively into a combinadic with $k-1$ components smaller than $n-1$. McCaffrey's algorithm is described in Algorithm 4 below.

[3]We extend the definition of binomial coefficients with $\binom{n}{k} = 0$ when $k > n$.

[4]The previous unranking algorithm from Sagemath is stored in the Software Heritage Archive swh:1:cnt:c60366bc03936eede6509b23307321faf1035e23;lines=539-605

[5]The new unranking algorithm from Sagemath is stored in the Software Heritage Archive swh:1:cnt:b2a68056554dbf90fa55e76820f348d9d55019e3;lines=539-653

Algorithm 4 Unranking a combination

```
1: function UnrankingViaCombinadic(n, k, u)
2:     L ← [0, ..., 0]                              ▷ k components
3:     u' ← binomial(n, k) − 1 − u
4:     v ← n
5:     for i from 0 to k − 1 do
6:         v ← v − 1
7:         b ← binomial(v, k − i)
8:         while u' < b do
9:             v ← v − 1
10:            b ← binomial(v, k − i)
11:        u' ← u' − b
12:        L[i] ← n − 1 − v
13:    return L
```

As we noted while explaining Table 4, we work with the reverse of the rank u (see line 3 in the algorithm) in order to unrank combination in lexicographic order. The presented algorithm is also close to Er's algorithm [9] whose representation for combinations is distinct but the computations are analogous; furthermore in his paper, Theorem 2 corresponds exactly to the combinadic decomposition.

The function UnrankingViaCombinadic(n, k, u) computes the combinations of k elements among n of rank u in lexicographic order, though the core of the algorithm is reverse co-lexicographic. The correctness of the algorithm is stated in [14].

Again, we express the complexity of this algorithm as its number of calls to the function `binomial`. First note that, the values n and k being given, the worst cases are obtained when v gets as small as possible at then end of the loop, thus for all u whose combinadic satisfy $c_1 = 0$. Hence, the worst case complexity is $n - 1$. Again, we complete this analysis, by computing the average complexity of the algorithm. To reach this goal, we introduce the sequence $u_{n,k}$ computing the cumulative number of calls to `binomial` when u ranges from 0 to $\binom{n}{k} - 1$.

Lemma 22. *Let $u_{n,k}$ be the cumulative numbers of calls to `binomial` while unranking all possible u from 0 to $\binom{n}{k} - 1$. The sequence satisfies: $u_{n,0} = 1$ and $u_{n,n+i} = 0$ for all n and $i > 0$ and otherwise*

$$u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}.$$

$n \backslash k$	0	1	2	3	4	5	6	7	8
1	1	2							
2	1	5	3						
3	1	9	11	4					
4	1	14	26	19	5				
5	1	20	50	55	29	6			
6	1	27	85	125	99	41	7		
7	1	35	133	245	259	161	55	8	
8	1	44	196	434	574	476	244	71	9

TABLE 5. First values of $u_{n,k}$ for $n = 1..8$ and $k = 0..n$ (Algorithm 4)

Table 5 presents the sequence given in Lemma 22. The difference with the two sequences studied before lies in the extreme cases. This sequence is a shifted version of the sequence `OEIS A264751`. Both combinatorial objects can be put in bijection, and thus some conjectures stated there, are solved in the following.

Proof. Note that $n - c_k$ calls to `binomial` are necessary to determine c_k, then $c_k - c_{k-1}$ calls to determine c_{k-1}, ..., $c_2 - c_1$ to determine c_1. Hence, the total number of binomial coefficients evaluations necessary to compute the combinadic of u is $n - c_1$ (and thus only depends on c_1). Besides, for a given $j \geq 0$, the number of finite sequences $c_1 = j < c_2 < c_3 < \cdots < c_k < n$ is equal to the number of sequences $0 \leq c'_2 < c'_3 < \cdots < c_k < n - j - 1$ by the change of variable $c'_i \leftarrow c_i - j - 1$. Hence, this number is equal to $\binom{n-1-j}{k-1}$. In addition, there is a first call to `binomial` at the beginning of the algorithm to reverse the rank, regardless of the value of u. We thus obtain:

$$u_{n,k} = \binom{n}{k} + \sum_{j=0}^{n-k} (n-j) \cdot \binom{n-j-1}{k-1}.$$

Using the latter equation, the recursive equation is directly proved by induction. □

Note that in this case the cumulative numbers are not symmetrical $u_{n,k} \neq u_{n,n-k}$. In fact the computation of the combinadics is not symmetrical.

Theorem 23. *Let $U(z, y)$ be the generating function associated to $(u_{n,k})$. Then*

$$U(z, y) = \frac{1}{1 - z - zy} \left(\frac{1}{1 - z - zy} - \frac{z}{1 - z} \right); \text{ thus}$$
$$u_{n,k} = \binom{n}{k} \left(n + 1 - \frac{n-k}{k+1} \right).$$

The proof is the same as the one of Theorem 13. The values for $u_{n,k}$ are a bit different due to the extreme cases $u_{n,0}$.

Corollary 24. *The average number of calls to* `binomial` *in Algorithm 4 for n being large and k being of the form αn for $0 < \alpha < 1$ is*

$$\frac{u_{n,k}}{\binom{n}{k}} \underset{\substack{n \to \infty \\ k = \alpha n}}{=} n + 2 - \frac{1}{\alpha} + O\left(\frac{1}{n}\right).$$

In the literature there is another algorithm based on combinadics given in [5]. We provide a pseudo-code equivalent of the original Fortran algorithm in Algorithm 5. Note that the algorithm does not handle the case $k = 1$, which should thus be treated separately. There, in the computation of the combinadic for a given rank, the coefficients are computed from the smallest one, c_1, to the second largest one, c_{k-1}, and finally the value for c_k is directly deduced with no need for further trials. In this algorithm, the variable L contains the combinadic of u (not its reverse). We note two differences with Algorithm 4. First, last coefficient (c_k) is directly computed without "trying" the different possible values as for the previous coefficients (see line 13). Second, it uses a combinatorial argument to find the value of the c_i coefficients that is the complementary of the argument used in the privous algorithm: the number of sequences $0 \leq c_1 < c_2 < \cdots < c_k < n$ with $c_1 \geq j$ is equal to $\sum_{i=0}^{j} \binom{n-i-1}{k-1}$, hence the accumutation performed in the r variable. The fact that the same combinatorial argument can be used in two different ways here has to be put in parallel with Remark 15 at the beginning of this section.

The first point mentioned above is an improvement over Algorithm 4, but in fact this second algorithm is penalised by the extra addition that it has to perform and then undo at line 12 to find each c_i. With this approach it is mandatory to compute the accumulation of binomial coefficients in r until it becomes greater than u to know when to exit the loop. It will appear in our experimentations in the next section that this has a noticeable impact on performance.

Algorithm 5 Unranking a combination (alternative algorithm)

```
1: function UnrankingViaCombinadic2(n, k, u)
2:     L ← [0, . . . , 0]                                   ▷ k components
3:     r ← 0
4:     for i from 0 to k − 2 do
5:         if i ≠ 0 then L[i] ← L[i − 1]
6:         else L[i] ← −1
7:         while true do
8:             L[i] ← L[i] + 1
9:             b ← binomial(n − L[i] − 1, k − i − 1)
10:            r ← r + b
11:            if r > u then exit the loop

12:        r ← r − b
13:    L[k − 1] ← L[k − 2] + u − r + 1
14:    return L
```

UnrankingViaCombinadic2(n, k, u) is a lexicographic unranking for combinations. This algorithm is presented by Buckles and Lybanon in [5] and its correctness is presented in [14]. Finally note it is approximately the implementation in `Matlab` [6] whose code is also presented by Ruskey in [20, p. 65]. The latter approach does also trials to find the last coefficient of the combination instead of computing it directly line in line 15.

Lemma 25. *Let $u_{n,k}$ be the cumulative numbers of calls to `binomial` while unranking all possible u from 0 to $\binom{n}{k} - 1$. For all n, the sequence satisfies $u_{n,k} = 0$ when $k = 1, 2$ or $k > n$ and otherwise:*

$$u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}.$$

The result is proved in an analogous way as Lemma 22, summing over c_{k-1} instead of c_1. In Table 6 we compute the first values of $(u_{n,k})$. We note the first values are smaller than the previous ones, Theorem 26 gives their asymptotic behavior.

$n \backslash k$	0	1	2	3	4	5	6	7	8
1	0	0							
2	0	0	1						
3	0	0	4	2					
4	0	0	10	10	3				
5	0	0	20	30	18	4			
6	0	0	35	70	63	28	5		
7	0	0	56	140	168	112	40	6	
8	0	0	84	252	378	336	180	54	7

Table 6. First values of $u_{n,k}$ for $n = 1..8$ and $k = 0..n$ (Algorithm 5)

Theorem 26. *Let $U(z,y)$ be the generating functions associated to $(u_{n,k})$. Then*

$$U(z,y) = \frac{z^2\ y^2}{(1-z)^2\ (1-z-zy)^2};\ \textit{thus}$$
$$u_{n,k} = \binom{n}{k} \frac{k-1}{k+1}\ (n+1).$$

Corollary 27. *The average number of calls to* `binomial` *in Algorithm 5 for n being large and k being of the form αn for $0 < \alpha < 1$ is*

$$\frac{u_{n,k}}{\binom{n}{k}} \underset{\substack{n\to\infty\\ k=\alpha n}}{=} n + 1 - \frac{2}{\alpha} + O\left(\frac{1}{n}\right).$$

The improvement for the efficiency of the algorithm is seen only on the second order in comparison to the one of Corollary 24.

Let us conclude this part with an interesting remark. In our Algorithm 3 while unranking k elements among n, we combinatorially see $\binom{n}{k}$ as first $\binom{n-1}{k-1}$ and otherwise at $\binom{n-1}{k}$. This is the same approach as in Algorithm 5. But as we presented after Algorithm 3, we could have first be interested in $\binom{n-1}{k}$ and then in $\binom{n-1}{k-1}$. Then the algorithm would have been in a reverse co-lexicographic fashion, and it would follwo exactly the same approach as Algorithm 4.

4. Improving efficiency and realistic complexity analysis

We will now show that the complexity study mentioned above is not adequate anymore. Although the approximation that computing one binomial coefficient has a constant cost seems to have been sufficient in the past (due the limitation of the implementations to 32-bits integers, see [16] who seems to be the first to introduce combination unranking using big integer computations), this is not a valid model anymore as big integers are now widely used. We will discuss the impact of big integer manipulation and some optimizations that significantly improve the performance of all algorithms.

Using the two last sections we know the different algorithms that are mostly used in practice. But having in mind the results exhibited in Table 1 there are probably deep modifications in the codes that are implemented. Obviously by using an implementation calling directly big integers through the `GMP` libray in `C` is the best way too compute very fast, but the distinct behaviors presented in the table militate in favor of other improvements.

For all algorithms we proved that the average number of calls to the function `binomial` is equivalent to n (when k grows linearly). A first question to investigate is about this complexity measure: is it reasonable? An obvious approach consists in analyzing the average time spent for the computation of uniformly sampled combinations.

4.1. First experiments to visualize the time complexity in real context.

In the two next plots in Figure 1 we have represented the average time needed for the computations of 500 combinations for each pair $(n, k = n/2)$ while n is ranging from 250 to $10,000$ with a step size of 250. The choice $k = n/2$ has been done because it corresponds to the worst complexity cases while k is ranging 0 to n. To conduct experiments we have implemented all algorithms in `C` using the classical `GMP` library for big integers[6]. On the leftmost plot we represent the average time for the unranking of a combination for n ranging from 250 to $10,000$ by steps of 250 too. Note that all the curves are merged: all algorithms seem to need the almost exactly the same time to unrank a combination.

In the rightmost plot, we present a version with memoization for the algorithms: we first run a pre-computation step storing all binomial coefficients that will be used. The second step consists in unranking the combinations by using the pre-calculated values for the binomial coefficients. On our plot we represent only the time used for the second step. We remark that the recursive Algorithm 3

[6]The implementation and the exhaustive material used for repeating the experiments are available at `http://github.com/Kerl13/combination_unranking`.

The experiments have been driven through a standard laptop PC with an I7-8665U CPU, 32Gb RAM running Ubuntu Linux 2020.

is not as efficient as the others. Later on we will be able to state it is due to its recursive nature. As we have earlier explained, Algorithm 5 should be less efficient as Algorithm 4: this is the case. Our Algorithm 2 and Algorithm 4 are the most efficient when used in this setup.

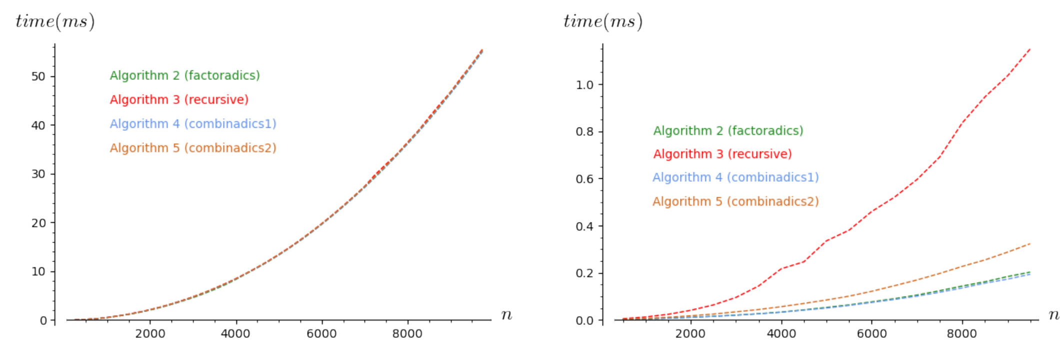

FIGURE 1. Time (in ms) for unranking a combination, with $n = 250..10,000$ and $k = n/2$.

While the number of calls to the binomial coefficients is linear in n, it is clear that it is not the case for the time complexity of the algorithms (see on the leftmost plot). But, once the binomial coefficients have been pre-computed, then running the four algorithms without computing binomial coefficients is closer to a linear function but they still are not really linear.

While memoizing all binomial coefficients when n is of order of some hundreds is possible, it is not the case anymore when n is of the order of several thousands[7]. While such cases did not occur when the methods (e.g. [5, 9]) were derived, it is now necessary to take the cost of arithmetic operations into account now that big integers are mode widely used (the first use of big integers for combination unranking seems to fall back to 2004 [16]). A more detailed analysis of the time complexity is necessary.

4.2. **Improving the implementations of the algorithms.** Before going further with the experiments, we propose an improvement in the computation of the binomial coefficient, that is applicable to all the algorithms presented in this paper.

In all of the presented algorithms, a binomial coefficient is computed at each step of the generation. There are various ways to implement those. One possibility is to compute the two products $n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$ and $k!$ separately using a divide and conquer strategy as described in [4, Section 15.3] and then to compute the division.

In the unranking algorithms, instead of doing the "full" evaluation of the binomial coefficient at each step, it is possible to reuse the computations from the previous step and to deduce the value of the new coefficient by a constant number of multiplications or divisions by a small integer. This is possible because in all algorithms the parameters of the binomial coefficients vary only by ± 1 from one step to the other. For instance, in Algorithm 2, just before incrementing i at line 9 the coefficient for the *next* iteration can be obtained by multiplying b by $k-1-i$ and dividing it by $n-1-m-i$. Similarly, in the other branch of the if, just before incrementing m at line 12, the value of the coefficient for the next iteration can be obtained by multiplying b by $n-m-k$ and dividing it by $n-1-m-i$. In the end, only the first binomial coefficient is computed from scratch and all others are obtained as described above. This lowers the amortized cost of computing one coefficient to $\Theta(1)$ rather than $\Theta(k)$. This optimization is applicable to all the algorithms presented in this paper.

In Algorithm 6, we propose an optimized version of our Algorithm 2 based on the above remarks on binomial coefficients. It also includes some other enhancements. First, instead of computing the permutation rank of the combination and then unranking the combination as a permutation, it is possible to process the coefficients of the factoradic decomposition on the fly to extract the

[7]For the experiments of Figure 1 we have only computed the necessary binomial coefficients using a lazy approach.

right value from the set $\{0, 1, 2, \ldots, n-1\}$. Second, we can note that we are in a special use-case of the EXTRACT function where values are extracted in increasing order. Hence, there is no need to explicitly store the set of remaining values (not yet in the combination) to get access its m-th value: it is $m+i$ where i is the number of already extracted values. Finally, the last component of the combination can be deduced without computing more binomial coefficients (line 15) in order to leave the loop earlier.

Algorithm 6 Unranking a combination with optimization

1: **function** OPTIMIZEDUNRANKINGCOMBINATION(n, k, u)
2: $\quad L \leftarrow [0, \ldots, 0]$ ▷ k components
3: $\quad b \leftarrow$ `binomial`$(n-1, k-1) \cdot n$
4: $\quad m \leftarrow 0; \quad i \leftarrow 0;$
5: $\quad$ **while** $i < k-1$ **do** ▷ Invariant: $b = \binom{n-m-i-1}{k-1-i} \cdot (n-m-i)$
6: $\qquad b \leftarrow b/(n-m-i)$
7: $\qquad$ **if** $b > u$ **then**
8: $\qquad\quad L[i] \leftarrow m+i$
9: $\qquad\quad b \leftarrow b \cdot (k-i-1)$
10: $\qquad\quad i \leftarrow i+1$
11: $\qquad$ **else**
12: $\qquad\quad u \leftarrow u-b$
13: $\qquad\quad b \leftarrow b \cdot (n-m-k)$
14: $\qquad\quad m \leftarrow m+1$
15: $\quad$ **if** $k > 0$ **then** $L[k-1] \leftarrow n+u-b$
16: $\quad$ **return** L

Before going on with the efficiency comparisons, we use the remarks related to the binomial coefficient computation to improve Algorithm 4, Algorithm 5 and Algorithm 3. Furthermore, in order to make the comparison fair, we used a variant of Algorithm 3 for the recursive approach in which the array storing the result is allocated with the right size at the beginning of the execution, like in the other algorithms. Also, in order not to penalize it due to its recursive nature, the order of some instructions have been changed so that it is tail-recursive. The optimized version is given in Algorithm 7.

Algorithm 7 Recursive method with optimizations

function OPTIMIZEDUNRANKING-RECURSIVE(n, k, u)
$\quad L \leftarrow [0, \ldots, 0]$ ▷ k components
$\quad b \leftarrow$ `binomial`(n, k)
$\quad$ UNRANKTR$(L, 0, 0, n, k, u, b)$
$\quad$ **return** L

function UNRANKTR(L, i, m, n, k, u, b)
$\quad$ **if** $k = 0$ **then** do nothing
$\quad$ **else if** $k = n$ **then**
$\qquad$ **for** j **from** 0 **to** $k-1$ **do**
$\qquad\quad L[i+j] \leftarrow m+j$
$\quad$ **else**
$\qquad b \leftarrow b/n$
$\qquad$ **if** $u < b$ **then**
$\qquad\quad L[i] \leftarrow m$
$\qquad\quad b \leftarrow (k-1) \cdot b$
$\qquad\quad$ UNRANKTR$(L, i+1, m+1, n-1, k-1, u, b)$
$\qquad$ **else**
$\qquad\quad u \leftarrow u-b$
$\qquad\quad b \leftarrow (n-k) \cdot b$
$\qquad\quad$ UNRANKTR$(L, i, m+1, n-1, k, u, b)$

In UnrankTR, the variable i represents the position in L of the next value to be computed and the variable m represents the next candidate to be the value of $L[i]$. The invariant satisfied by b is that UnrankTR is always called with $b = n \cdot \binom{n-1}{k-1}$.

We propose a second time efficiency comparison for some algorithms with their optimizations in the Figure 2. Comparing this plot with the leftmost one of Figure 1 we note that when $n = 10000$ the algorithms run approximately 45 times faster. Again we note that Algorithm 5 is still less efficient than the others that all seem to be equivalent.

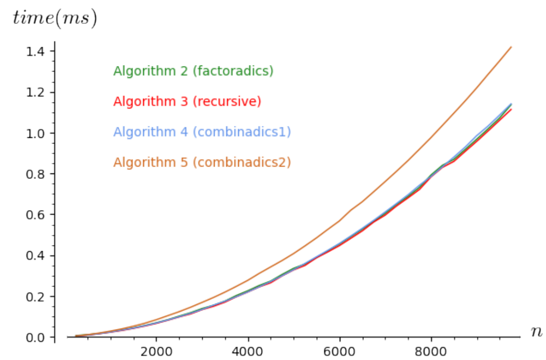

Figure 2. Time (in ms) for unranking a combination with the optimized algorithms

In order to understand the behavior of the curves of the previous plot, we introduce another way to analyze the time complexity in Figure 3. Now we take $n = 10,000$ and we let k ranging from 250 to 9750 with an iteration step of 250. For each step we represent an average value for 500 tests. Again (on the leftmost plot) the dashed lines correspond to the first version of each algorithm, and the solid ones to the optimized versions. On the rightmost plot we only focus on the optimized versions of the four algorithms. We remark the worst time complexity if when k reach $n/2$. Algorithm 6, the tail-recursive Algorithm 7 and Algorithm 4, do behave almost in the same way. While Algorithm 4 is a little bit more efficient when $k < n/2$ the two other are a bit more efficient for the second half range.

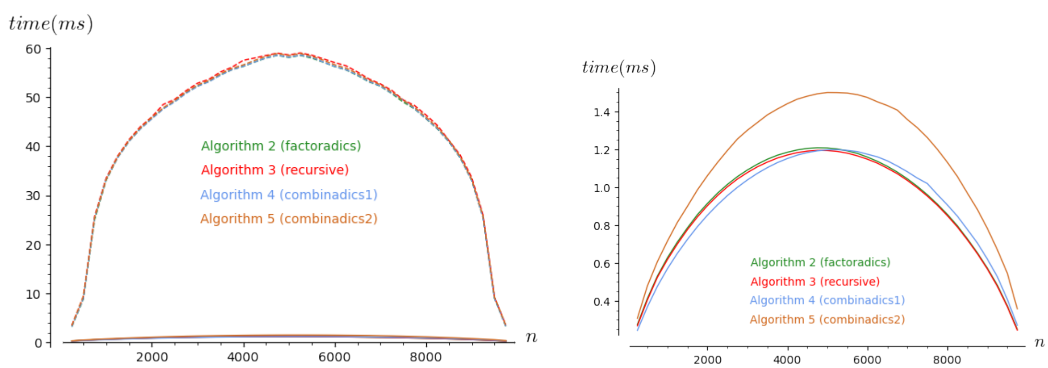

Figure 3. Time (in ms) for unranking a combination, with $n = 10,000$ and $k = 25..9975$

So the curves for Algorithm 7, Algorithm 6 and Algorithm 4 (once optimized) are hard to distinguish. There is a surprising explanation for this. In fact, it can be shown that all three algorithms perform the exact same arithmetic operations on big integers except for a few terms at then end of their execution, due to their base cases. In the case of the recursive and factoradics-based algorithms, the similarity goes further. Algorithm 7 being tail-recursive, it can be automatically translated into the imperative style[8] and the result of the automatic translation is an algorithm which is almost identical to Algorithm 2. In the case of Algorithm 4 (once optimized), although the computations are the same, they are used to obtain the values of the c_i coefficients in a different order, which makes the parallel less obvious.

As a result, the only fundamental differences between all these algorithms are their base cases. Since they are all based on a different combinatorial point of view, their base cases have been characterized slightly differently but, as we can see by comparing the results of their complexity analyses (establishing how many calls to `binomial` are done), this only impacts the second order of their complexity.

Besides, for all algorithms, a significant speedup can be achieved by re-using the value of the most recently computed binomial coefficient. To the best of our knowledge, this is the first time this trick is used for the unranking of combinations.

4.3. **Realistic complexity analysis.** We now propose a more precise complexity analysis based on a more realistic cost model. Recall that we deal with big integers. More precisely for n and k being given, the ranks as well as the binomial coefficients computed during the generation can have up to $L_{n,k} = 1 + \log_2 \binom{n}{k}$ bits. Using Stirling's approximation we get that

$$L_{n,k} \underset{\substack{n\to\infty \\ k=\alpha n}}{\sim} n \left(\alpha \log_2 \frac{1}{\alpha} + (1-\alpha) \log_2 \frac{1}{1-\alpha} \right).$$

Besides, the cost of the multiplication of a big integer with $O(n)$ bits with a smaller integer of $O(\ln n)$ bits can be bounded by $O(\frac{n}{\ln n} M(\ln n))$ where $M(x)$ is the cost of multiplying two x-bits integers. This can be achieved by writing the big integer in base $2^{\log_2 n} = n$ and performing the multiplication using the naive "textbook" algorithm in this base. The first term $\frac{n}{\ln n}$ counts the number of operations done in base n and the second term $M(\ln)$ is the cost of one single multiplication in this base.

A rough upper bound for $M(x)$ is x^2, obtained by using the naive multiplication algorithm. Actually, the naive algorithm is often used in practice for small values of x since the asymptotically more efficient algorithms only become better above a given threshold. For our use-case n is likely to fit in a machine word in practice and thus the naive algorithm must be used. Hence, the upper bound of $O(n \ln n)$ for the cost of the multiplication of a small integer by a big integer should faithfully reflect the actual runtime of our implementations, although it is theoretically not optimal. A tighter bound of $O(n(\ln\ln n)(\ln\ln\ln n))$ can be obtained by using the Schönhage-Strassen algorithm, though it is not advisable in practice.

In addition to the cost of the multiplications discussed above, a linear number of comparisons and additions are performed. The cost of one such operation is linear in n and thus negligible compared to the cost of the multiplications. Finally, the first binomial coefficient must be computed from scratch which can be done at negligible cost compared to $\frac{n^2}{\ln n} M(\ln n)$.

By combining the above discussion with the results from the previous sections, we get the average bit-complexity of all algorithms when k grows linearly with n as presented in Theorem 28.

Theorem 28. *For all the optimized algorithms of the present paper, there exist a constant $c > 0$ such that for all n large enough and $k = \alpha n$ for $0 < \alpha < n$, the bit-complexity of the algorithm is bounded by:*

$$c \cdot n^2 \ln n \cdot \left(\alpha \log_2 \frac{1}{\alpha} + (1-\alpha) \log_2 \frac{1}{1-\alpha} \right)$$

In Figure 4 we display the time complexity of our algorithm (in green) with the graph of the function $\alpha \mapsto C \cdot \left(\alpha \ln \frac{1}{\alpha} + (1-\alpha) \ln \frac{1}{1-\alpha} \right)$ where the constant C has been chosen so that the maximum values of both curves coincide.

This validates experimentally our complexity results. Besides, we checked by profiling the optimized algorithms that most of the run time is spent in the arithmetic operations, which also confirms the validity of this result.

[8]The conversion of a tail-recursive algorithm into its imperative version is called "tail-call" optimization and is implemented by most compilers for languages with recursion.

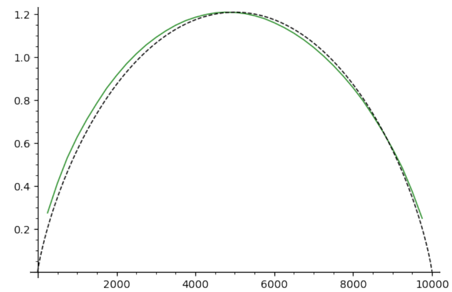

FIGURE 4. Merge of Algorithm 2 and theoretical complexity

5. Extensions and conclusion of the algorithmic context

5.1. **Objects counted by multinomial coefficients.** Let n and m be two positive integers and $K = (k_1, k_2, \dots, k_m)$ be a finite sequence of non-negative integers whose sum equals to n. The multinomial coefficient $\binom{n}{k_1,\dots,k_m}$ counts the number of ways of depositing n distinct objects into m distinct bins such that there are k_i objects in the i-th bin. It can also be interpreted as combination with repetitions: we have a pool of m kinds of different objects, we must pick a finite sequence of n objects such that k_1 of them are of the first kind, k_2 of the second kind and so on.

Note that when $m = 2$ the multinomial coefficient corresponds to a binomial coefficient. Informally, Proposition 8 (related to combinations) states that the combinations are in one-to-one correspondence with permutations containing two increasing runs. We have an analogous interpretation here: the ranks from 0 to $\binom{n}{k_1,\dots,k_m} - 1$ are in one-to-one correspondence with size-n permutations composed of m increasing runs. We now exhibit formally this link.

Proposition 29. *Let n and m be two positive integers and $K = (k_1, k_2, \dots, k_m)$ each object enumerated by $\binom{n}{k_1,\dots,k_m}$ is represented by a finite sequence $(\ell_1, \ell_2, \ell_{k_m})$ where for all $1 \le i \le m$, ℓ_i is an increasing finite sequence of length k_i. Furthermore, the union over i of all elements of ℓ_i is exactly $\{0, 1, \dots, n-1\}$.*

Our unranking algorithms relies on the classical point of view relating a multinomial coefficient to a product of binomial coefficients.

$$\binom{n}{k_1,\dots,k_m} = \binom{k_m}{k_m} \cdot \binom{k_m + k_{m-1}}{k_{m-1}} \cdots \binom{k_m + \cdots + k_2}{k_2} \cdot \binom{k_m + \cdots + k_1}{k_1}.$$

Proposition 30. *The function* UNRANKINGCOMBINATIONWITHREPETITIONS$(n, (k_1, \dots, k_m), u)$ *computes the u-th object counted by $\binom{n}{k_1,\dots,k_m}$ in lexicographic order.*

The core of the algorithm consists in computing the rank of the permutation, written in factoradics, associated to the combination with repetitions we are interested in. It remains then to unrank a permutation.

Proof. Based on Proposition 29 the core of the algorithm computes the rank of a permutation containing m increasing runs respectively of lengths $k_1, k_2, \dots, k_m$. Determining the contribution of each run in the factoradic decomposition of the permutation rank is done in the external loop starting in line 5, from k_m downto k_1. The correctness of our algorithms relies on the following loop invariant:

- the values of f_j for all $0 \le j < k_m + \cdots + k_{i+1}$ have been computed and stored in F;
- the values of $f_{k_m+\cdots+k_{i+1}}, \dots, f_{k_m+k_{i+1}+k_i-1}$ (which has not been determined yet, as we enter the loop) are equal to the factoradics othe the rank $u' \mod \binom{k_m+\cdots+k_i}{k_i}$ in the combinations of k_i elements among $k_m + \cdots + k_i$ possible elements.
- the variable u' holds the rank of the runs that must be still unranked.

Algorithm 8 Unranking a combination with repetitions

1: **function** UnrankingCombinationWithRepetitions($n, K = (k_1, \ldots, k_m), u$)
2: $F \leftarrow [0, \ldots, 0]$ ▷ n components in M
3: $u' \leftarrow u$
4: $n' \leftarrow k_m$
5: **for** i **from** $m-1$ **downto** 1 **do**
6: $n' \leftarrow n' + k_i$
7: $b \leftarrow$ `binomial`(n', k_i)
8: $(u', u'') \leftarrow$ `division`(u', b)
9: $F' \leftarrow$ `factoradic`(RankConversion(n', k_i, u''))
10: **for** j **from** 0 to $k_i - 1$ **do**
11: $F[n' - k_i + j] \leftarrow F'[n' - k_i + j]$
12: $r \leftarrow$ `composition`(F)
13: **return** UnrankingPermutation(n, r)

`division`(s,t): returns the pair (q,r) corresponding respectively to the quotient and the remainder of the integer division from s by t.

Once the factoradic F' of the run under consideration has been computed (line 10), it remains to update F according to the k_i last components of F'. □

5.2. **Objects counted by k-permutations.** Let n and k be two positive integer. A k-permutation is a combination of k elements that are ordered among n elements. Objects that are counted by this notion have a cardinality corresponding to

$$k!\binom{n}{k} = \binom{n}{1, \ldots, 1, n-k}.$$

Proposition 31. *The function* UnrankingKPermutation(n, k, u) *returns the result to the call of* UnrankingCombinationWithRepetitions$(n, (1, \ldots, 1, n-k), u)$*. Thus it computes the u-th k-permutation among n elements in lexicographic order.*

Proof. The facts that both combinatorial classes have the same cardinality and that the algorithm for unranking combination with repetitions is lexicographic induce the correctness of the function. □

6. Conclusion

As a surprising result we thus have remarked that all usual algorithms for the unranking of combinations (first the recursive method but also the algorithms using combinadics, and also our new algorithm based on factoradics) share a very similar core, doing almost the same computations in order to reconstruct the combination under consideration.

One interesting remark is that understanding in details the core computations that are necessary to unrank combinations, it is possible to significantly improve all algorithms. This understanding, joint with a detailed and realistic theoretical complexity analysis leads to a prediction of the run time of the algorithm that matches completely the actual run time of their implementations.

However due to details that are neglected in practice, we realize in Table 1 that some improvements are still necessary in various Computer Algebra Systems in order to get the most efficient implementations possible for the unranking of combinatorial objects.

Acknowledgment. We thank C. Donnot and Y. Herida for having conducted the preliminary experiments about our algorithm.

References

[1] E. F. Beckenbach and G. Pólya. *Applied Combinatorial Mathematics*. R.E. Krieger Publishing Company, 1981.

[2] O. Bodini, A. Genitrini, and F. Peschanski. A Quantitative Study of Pure Parallel Processes. *Electronic Journal of Combinatorics*, 23(1):P1.11, 39 pages, 2016.

[3] B. Bonet. Efficient algorithms to rank and unrank permutations in lexicographic order. *AAAI Workshop - Technical Report*, pages 18–23, 2008.

[4] A. Bostan, F. Chyzak, M. Giusti, R. Lebreton, G. Lecerf, B. Salvy, and É. Schost. *Algorithmes Efficaces en Calcul Formel*. 2017. 686 pages. Édition 1.0.

[5] B. P. Buckles and M. Lybanon. Algorithm 515: Generation of a Vector from the Lexicographical Index [G6]. *ACM Trans. Math. Softw.*, 3(2):180–182, 1977.

[6] B. Butler. Function kSubsetLexUnrank, MATLAB central file exchange, 2020.

[7] C. Donnot, A. Genitrini, and Y. Herida. Unranking combinations lexicographically: an efficient new strategy compared with others, 2020. http://hal.archives-ouvertes.fr/hal-02462764v1.

[8] R. Durstenfeld. Algorithm 235: Random permutation. *ACM*, 7(7):420–, 1964.

[9] M. C. Er. Lexicographic ordering, ranking and unranking of combinations. *International Journal of Computer Mathematics*, 17(3-4):277–283, 1985.

[10] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver & Boyd, London, 1948.

[11] P. Flajolet and R. Sedgewick. *An Introduction to the Analysis of Algorithms (2nd Edition)*. Addison Wesley, 2013.

[12] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1-2):1–35, 1994.

[13] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

[14] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: generation, enumeration, and search*. CRC Press, 1999.

[15] C. Martínez and X. Molinero. A generic approach for the unranking of labeled combinatorial classes. *Random Structures & Algorithms*, 19(3-4):472–497, 2001.

[16] J. McCaffrey. Generating the mth Lexicographical Element of a Mathematical Combination. *MSDN*, 2004. http://docs.microsoft.com/en-us/previous-versions/visualstudio/aa289166(v=vs.70).

[17] A. F. Myers. k-out-of-n:g system reliability with imperfect fault coverage. *IEEE Transactions on Reliability*, 56(3):464–473, 2007.

[18] A. Nijenhuis and H. S. Wilf. *Combinatorial algorithms*. Computer science and applied mathematics. Academic Press, New York, NY, 1975.

[19] E. Pascal. Sopra una formula numerica. *Giornale di Matematiche*, 25:45–49, 1887.

[20] F. Ruskey. Combinatorial Generation, 2003.

[21] Y. Tamada, S. Imoto, and S. Miyano. Parallel algorithm for learning optimal bayesian network structure. *J. Mach. Learn. Res.*, 12:2437–2459, 2011.

[22] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.2)*.

Antoine Genitrini and Martin Pépin. Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6 -LIP6- UMR 7606, F-75005 Paris, France.

Email address: `{Antoine.Genitrini; Martin.Pepin}@lip6.fr`