Unranking Combinations Lexicographically: an efficient new strategy compared with others

Cyann Donnot, Antoine Genitrini, Yassine Herida

▶ **To cite this version:**

Cyann Donnot, Antoine Genitrini, Yassine Herida. Unranking Combinations Lexicographically: an efficient new strategy compared with others. 2020. hal-02462764

HAL Id: hal-02462764

https://hal.sorbonne-universite.fr/hal-02462764v1

Preprint submitted on 31 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Unranking Combinations Lexicographically: an efficient new strategy compared with others

Cyann Donnot[1], Antoine Genitrini[2], and Yassine Herida[1]

[1] Full-time master student at Sorbonne University
[2] Sorbonne University, LIP6. Antoine.Genitrini@lip6.fr

Abstract. We propose a comparative study of four algorithms dedicated to the lexicographic unranking of combinations. The three first ones are algorithms from the literature. We give some key ideas how to analyze them in average and describe their strengths and weaknesses. We then introduce a new algorithm based on a new strategy inside the classical factorial numeral system (or factoradics). We conclude the paper with an experimental study, in particular when $n = 10000; k = 5000$, our algorithm is 36 (resp. 53) times as fast as our C++ implementation of `Matlab`'s (resp. `Sagemath`'s) algorithm.

keywords. Unranking Algorithm; Combination; Lexicographic Order; Complexity Analysis.

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 rediscover an important property linking natural numbers and a mixed radius numeral system based on combinations. This relationship gives 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 about the combinations). This approach is substituted to the exhaustive generation once the latter is not possible anymore due to the *combinatorial explosion* of the number of objects when their sizes increase. In the context of combination, the explosion appears really quickly. 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 [18] but also e.g. in software testing [14].

In order to unrank elements one must first define an order over the elements. The one that is usually used is the *lexicographic* one that can been seen as a generalization of the alphabetical order. The lexicographic order is humanly easy to handle with, and thus has been extensively studied. But, as Ruskey [17, p. 59] mentions, the lexicographic generation is usually not the most efficient, thus a particular care must be taken while unranking in 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 [15]. They did not directly study the example of combinations, but the case of integer compositions is proposed and is not so far from our problem. The method has been then applied generically on decomposable objects in the sense of analytic combinatorics, first in the context of recursive generation [10], and then in the context of unranking approaches [12].

Aside such generic approaches there are several ad hoc algorithms but to the best of our knowledge no practical experiments does exist to determine which one should be used. We thus will start with te recall of three classical algorithms, supported by a revisited average complexity analysis. Then we will describe our new strategy and finally we will compare them in some experiments. In Fig. 1 we represent the time efficiency of some improved versions of 3 of these algorithms, our being depicted in green. The figure will be described later. But as a foretaste, when $n = 10000; k = n/2$, our algorithm is experimentally 36 (resp. 53) times as fast as our C++ implementation of `Matlab`'s (resp. `Sagemath`'s) actual algorithm. Along the paper, we represent combinations as follows.

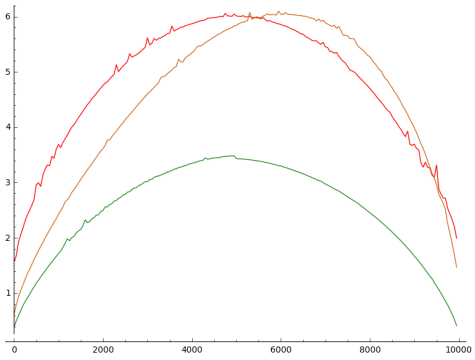

Fig. 1: Time (in ms) for unranking a combination, with $n = 10000$ and $k = 0..n$

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

For example, let n and k be respectively 5 and 3. The tuples $(0, 1, 2)$ and $(0, 2, 4)$ are combinations of k among n, but $(0, 2, 1)$ and $(0, 1, 2, 3)$ are not. There is another representation by using a 2-letters alphabet, but we do not deal with it through this paper. However we could rewrite the paper in this context also.

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 $A = (a_0, a_1, \ldots, a_{k-1})$ and $B = (b_0, b_1, \ldots, b_{k-1})$ be two distinct combinations.*

- *In the* lexicographic *order, we say that A is smaller than B if and only if both combination have the same prefix (eventually empty) such that $(a_0, \ldots, a_{p-1}) = (b_0, \ldots, b_{p-1})$ and $a_p < b_p$.*
- *In the* co-lexicographic *order, we say that A is smaller than B if and only if the tuple $(a_{k-1}, \ldots, a_0)$ is smaller than $(b_{k-1}, \ldots, b_0)$ 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 n, k define combinations and A be a combination. 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 element.*

With these definitions, we can enter the core of the paper organized as follows. Section 1 describes a recursive algorithms solving the lexicographic combination unranking problem. Despite its simplicity due to the approach, the fact it is recursive is problematic to handle very big combinations. Then Section 2 describes two classical algorithms based on combinadics. It seems that it is the first time both are compared and the reason why one is better is explained. In Section 3 we enter the context of factoradics, and then describe our new unranking algorithm in this numeral system. Finally in Section 4 we compare all algorithms experimentally and give some possibility for really improve the time complexity of all approaches.

1 Unranking through recursion

We deal with a combinatorial structure: combinations, that is very well understood from a combinatorial sense. Thus while trying to develop an unranking algorithm the first idea consists in developing a recursive algorithm based on the classical recursive generation presented in [15]. Let us directly introduce such an algorithm.

Algorithm 1 Recursive Unranking

```
1: function UNRANKING_REC(n, k, u)
2:     L := REC_GENERATION(n, k, u)
3:     return (n − 1 − L[k − 1 − i] for i
                          from 0 to k − 1)
```

`binomial`(n, k) computes the value of $\binom{n}{k}$;
`append`(A, a): appends element a in A;
For $i \geq 0$, $L[i]$ is the element of index i in L.
In line 3 and 5, the outputs are tuples built by comprehension. The simple one in line 5 is $(0, 1, \dots, k-1)$.

```
1: function REC_GENERATION(n, k, u)
2:     if k = 0 then
3:         return ()
4:     if n = k then
5:         return (i for i from 0 to k − 1)
6:     b := binomial(n − 1, k − 1)
7:     if u < b then
8:         R := REC_GENERATION
                          (n − 1, k − 1, u)
9:         append(R, n − 1)
10:        return R
11:    else
12:        return REC_GENERATION
                          (n − 1, k, u − b)
```

Proposition 1. *The function* REC_GENERATION$(n, k, \cdot)$ *computes the combinations for n, k in the reverse co-lexicographic order.*

Corollary 1. *The function* UNRANKING_REC$(n, k, \cdot)$ *computes the combinations for n, k in the lexicographic order.*

The proposition is proved by induction and the corollary is a direct observation given in [11, p. 47].

In order to analyze the algorithm in details, 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.

Let us introduce the sequence $u_{n,k}$ computing the cumulative number of calls for the whole range for u.

Lemma 1. *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. The algorithm relies on the recursion underlying the binomial coefficients, with the additive cost 1 for each call to the function Rec_Generation$(n,k,\cdot)$ (in line 6). Furthermore, by calling the function once with the parameters n,k and u and a second time with n,k and v (such that $u\neq v$) then the following recursive calls will also be with distinct triplets. There is thus no multi-counting.

Theorem 1. *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*

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

The proof of Theorem 1 is given in Appendix A.1

This sequence is a shifted version of the sequence stored under the reference `OEIS A059797`[3]. 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.

0	0					
0	2	0				
0	5	5	0			
0	9	16	9	0		
0	14	35	35	14	0	
0	20	64	90	64	20	0

Fig. 2: First values of $u_{n,k}$ for $n=1..6$ and $k=0..n$

Corollary 2. *The function* Unranking_Rec$(n,k,\cdot)$ *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 $\alpha\in]0,1[$, we get*

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

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

This average value is interesting to be noted, because it indicates that obtaining an extreme case (for the recurrence) during the recursive calls is obtained in average after $(n + O(1))$ recursive calls.

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.

2 Unranking through combinadics

In 1887, E. Pascal [16] 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 combinations.

Fact 2 *Let $n \geq k$ be two positive numbers. For all integers u, with $0 \leq u < \binom{n}{k}$, there exist a unique sequence $0 \leq c_1 < c_2 < \cdots < c_k < n$ such that*[4]

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

The writing $(c_1, \ldots, c_k)$ is called the combinatoric *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 2004, using this representation, McCaffrey exhibited in the MSDN article [13], 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 [11, p. 47] and can also be seen as an extension of the work of Lehmer. This algorithm is interesting in the sense it corresponds to the implementation used in the mathematics software `Sagemath` [19][5]. In Fig. 3, on the left handside we present the algorithm used in `Sagemath` and on the right handside we exhibit the combinadics of some ranks of combinations of 2 elements among 6. More precisely we have chosen the reverse of the ranks to be able to compare them with their combinadics and the corresponding combination (due to line 3 in the algorithm). The following algorithm is also close to Er's algorithm [8] 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 UNRANKING_COMBINADIC$(n, k, \cdot)$ computes the combinations for n, k in the lexicographic order but the core of the algorithm is reverse colexicographic (in fact, the tuple L represents the reverse of the combinadic for efficiency reasons). The correction of the algorithm is presented in [11].

[4] We extend the definition of binomial coefficients with $\binom{r-1}{r} = 0$.

[5] Unranking algorithm from Sagemath is stored in the Sofware Heritage Archive swh:1:cnt:c60366bc03936eede6509b23307321faf1035e23;lines=473-537

Algorithm 2 Unranking a combination

```
1: function UNRANKING_COMBINADIC(n, k, u)
2:     L := ()
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 − length(L))
8:         while u' < b do
9:             v := v − 1
10:            b := binomial(v, k − length(L))
11:        u' := u' − b
12:        append(L, v)
13:    return (n − 1 − ℓ for ℓ in L)
```

We take $n = 6$ and $k = 2$.

rev. rank	`combinadic`	`combination`
14	(4, 5)	(0, 1)
13	(3, 5)	(0, 2)
12	(2, 5)	(0, 3)
11	(1, 5)	(0, 4)
10	(0, 5)	(0, 5)
9	(3, 4)	(1, 2)
...	...	...
4	(1, 3)	(2, 4)
3	(0, 3)	(2, 5)
2	(1, 2)	(3, 4)
1	(0, 2)	(3, 5)
0	(0, 1)	(4, 5)

Fig. 3: Algorithm in `Sagemath`; Examples of combinadics and their combination

A first step in the complexity analysis of this algorithm consists in calculating the number of calls to the function `binomial` in Algorithm 2. The values n and k being given, the worst cases are obtained when v is as small as possible at then end of the loop, thus for all u whose combinadic satisfy $c_1 = 0$. To obtained a detailed analysis, we are also interested in the average number of calls to `binomial`, when u describes the whole range from 0 to $\binom{n}{k} - 1$. To reach this goal, let us introduce the function $u_{n,k}$[6] computing the cumulative number of calls for the whole range for u.

Lemma 2. *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*

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

The single difference with Lemma 1 relies on the extreme cases. The key-ideas for the proof are given in Appendix A.2.

The leftmost part in Fig. 4 corresponds to the sequence given in Lemma 2. Our 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.

1 2	0 0
1 5 3	0 0 1
1 9 11 4	0 0 4 2
1 14 26 19 5	0 0 10 10 3
1 20 50 55 29 6	0 0 20 30 18 4
1 27 85 125 99 41 7	0 0 35 70 63 28 5

Fig. 4: First values of $u_{n,k}$ for $n = 1..6$ and $k = 0..n$ (left: Algorithm 2; right: Algorithm 3)

[6] We use several times the same writings $u_{n,k}$ and $U(z, y)$ for distinct algorithms, but they rely on the same definition.

Note, 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 3. *Let $U(z,y)$ be the generating function associated to $(u_{n,k})$. Then*

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

Corollary 3. *The average number of calls to* `binomial` *in Algorithm 2 for n being large and k being of the form αn for $\alpha \in]0,1[$ 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 the combinadics. But 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 of further trials. In this algorithm, the variable L contains the combinadic (not its reverse). First the computations that are needed are based on some smaller values for the binomial coefficients: in fact, the trials start with small parameters for the binomials and further they are increasing until we reach the solution, thus the biggest (non necessary) trials are saved. And second the last coefficient is directly deduced. This algorithm should be faster in practice. Such a modification in the order of computations can be put in parallel in the context of recursive generation or unranking, with the Boustrophedon traversal presented in [10] whose improvement relies also in the order of the computations.

Algorithm 3 Unranking a combination faster

```
1: function UNRANKING_COMBINADIC2(n, k, u)
2:     L := (0)
3:     r := 0
4:     for i from 0 to k − 2 do
5:         while u >= r do
6:             L[i] := L[i] + 1
7:             b := binomial(n − L[i], k − i − 1)
8:             r := r + b
9:         append(L, L[i])
10:        L[i] := L[i] − 1
11:        r := r − b
12:    append(L, L[i] + u + 1 − r)
13:    return L
```

UNRANKING_COMBINADIC2$(n,k,\cdot)$ is a lexicographic unranking for combinations. This algorithm is presented by Buckles and Lybanon[5]. The correction of the algorithm is presented in [11]. Just as a comment, for the calculation of the next value $L[i+1]$, we start with the preceding value $L[i]+1$ (cf. line 9). Finally note it is approximately the implementation in `Matlab` [6] (the latter is a little bit less efficient: line 12 is omitted and some more trials are done in the loop).

Lemma 3. *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: $u_{n,k} = 0$ for all n, $k = 1, 2$ 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 the one in Lemma 2, summing over c_{k-1} instead of c_1. In Fig. 4 (right) we computed the first values of $(u_{n,k})$. We note the first values are smaller than the previous ones, but what about their asymptotic behavior?

Theorem 4. *Let $U(z,y)$ be the generating functions associated to $(u_{n,k})$. Then*

$$U(z,y) = \frac{z^2\, y^2}{(1-z)^2\,(1-z-zy)^2}; \quad u_{n,0} = 0; \forall k > 0,\ u_{n,k} = \binom{n}{k}\,\frac{k-1}{k+1}\,(n+1).$$

Corollary 4. *The average number of calls to* `binomial` *in Algorithm 3 for n being large and k being of the form αn for $\alpha \in]0,1[$ 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 cannot be deduced directly from the latter result in comparison to the one of Corollary 3.

In [17, p. 65] Ruskey presents an alternative implementation based on combinadics, in co-lexicographical order. But the first coefficients are computed starting by the largest ones, like in Algorithm 2.

3 Unranking through factoradics: a new strategy

Aside the combinatorial number system, there is another classical mixed radix numeral system: the factorial number system or factoradics. This decomposition relies on factorial numbers.

Fact 5 *For all positive integers u, we define n such that $(n-1)! < u \leq n!$, there exists a unique sequence $(f_\ell)_{\ell\in\{0,\dots,n-1\}}$, 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 writing $(f_0, f_1, \dots, f_{n-1})$ is called the factoradic decomposition *of u.*

First remark that $f_0 = 0$ for all u. Take the number $u = 2020$ as an example, we obtain its decomposition: $0 \cdot 0! + 0 \cdot 1! + 2 \cdot 2! + 0 \cdot 3! + 4 \cdot 4! + 4 \cdot 5! + 2 \cdot 6!$, thus its factoradic is $(0,0,2,0,4,4,2)$.

Definition 4. *Let n be a positive integer. A n-permutation is an ordering of the elements from $\{0,1,\dots,n-1\}$.*

We represent a n-permutation as a tuple of n components. For example the tuple $(2,4,0,3,1)$ is a 5-permutation.

The factorial number system is used to define a one-to-one correspondence between integers and permutations: it is thus an unranking method for permutation. The algorithm implemented in the function Unranking_Permutation is an obvious adaptation of Fisher-Yates random sampler for permutations [9].

Algorithm 4 Unranking a permutation

```
1: function UNRANKING_PERMUTATION(u, n)
2:     F := factoradic(u)
3:     while length(F) < n do
4:         append(F, 0)
5:     return EXTRACT(F, n, n)

1: function EXTRACT(F, n, k)
2:     P := (0, 1, ..., n − 1)
3:     L := ()
4:     for i from 0 to k − 1 do
5:         append(L, P[F[n−1−i]])
6:         remove(P, F[n − 1 − i])
7:     return L
```

`length`(F): computes the number of components in F;
`remove`(F, i): removes from F the element at index i;
`composition`(F): computes the integer whose factoradic is the *reverse* of F.

Algorithm 5 Unranking a combination

```
1: function UNRANKING_FACTORADIC(n, k, u)
2:     u' := RANK_CONVERSION(n, k, u)
3:     F := UNRANKING_PERMUTATION(u', n)
4:     return EXTRACT(F, n, k)

1: function RANK_CONVERSION(n, k, u)
2:     D := (0, ..., 0)        ▷ n components in D
3:     m := 0;  i := 0
4:     while i < k do
5:         b := binomial(n − 1 − m, k − 1 − i)
6:         if b > u then
7:             D[i] = m
8:             i := i + 1
9:             n := n − 1
10:        else
11:            u := u − b
12:            m := m + 1
13:    return composition(D)
       ▷ D contains the reverse factoradic tuple
```

Fig. 5: Algorithms for the unranking of combination through factoradics

Fact 6 UNRANKING_PERMUTATION$(\cdot, \ell)$ *returns permutations of ℓ elements in the lexicographic order.*

Since the factoradic (with 8 components) of 2020 is $(0, 0, 2, 0, 4, 4, 2, 0)$, the 8-permutation of rank 2020 is $(0, 3, 6, 7, 1, 5, 2, 4)$.

Note, in the function EXTRACT, the way the data structure P is handled in memory is very important for efficiency reasons. The best way is probably to build a dynamic balanced tree as presented in [3], or a multiset (whose elements have weight 1 or 0) as presented in the appendix of [2]. It seems there does not exist any algorithm based on some `swap` operation giving an in-place shuffle to unrank permutation in the lexicographic order: in fact Durstenfeld's algorithm [7] cannot be easily adapted for the lexicographic order.

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

1. we define a bijection among 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 k-prefix of the permutation of rank u' by using the Algorithm 4.

Definition 5. *Let n and k be two integers with $0 \leq k \leq n$. We define the application $\mathcal{P}$ such that to the combination $(\ell_0, \ell_1, \ldots, \ell_{k-1})$ it associates the n-permutation $(\ell_0, \ell_1, \ldots, \ell_{k-1}, d_k, \ldots d_{n-1})$ such that $d_k < d_{k+1} < \cdots < d_{n-1}$.*

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 $(0, 2, 4, 1, 3)$.

Proposition 2. *The integers n and k being given, the application $\mathcal{P}$ transforming a combination into a permutation is one-to-one.*

Remark that the 5-permutation $(0, 1, 2, 3, 4)$ is the permutation associated to both combinations $(0, 1)$ and $(0, 1, 2)$ but in these examples the values of k are distinct. In fact, there are exactly 6 combinations associated to the latter permutation: $\mathcal{P}$ depends in n and k, it is a bijection when n and k are given. The proof is obvious.

The algorithm implemented in the function Rank_Conversion (see Algorithm 5) transforms the rank of a combination for n, k into the rank of its associated permutation through the computation of its factoradic. The main idea of the rank conversion is similar to the one for determining the combinadic through trials. It consists in building the (reverse of the) factoradic by detailing each component by subtracting the permutation ranks that must be omitted, when the component is still too small.

Proposition 3. *The function* Unranking_Factoradic*$(n, k, \cdot)$ computes the combinations for n, k in the lexicographic order.*

During the conversion from the rank of the combination to the one of the associated permutation, the coefficients are obtained via trials for f_{n-1} to f_{n-k}, remarking that through our bijection $\mathcal{P}$ 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 $\binom{n}{k} - 1$ introduces again the following cumulative sequence.

0	1					
0	3	2				
0	6	8	3			
0	10	20	15	4		
0	15	40	45	24	5	
0	21	70	105	84	35	6

Fig. 6: First values of $u_{n,k}$ for $n = 1..6$ and $k = 0..n$

Lemma 4. *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: $u_{n,k} = 0$ and $u_{n,n+i} = 0$ for all n and $i > 0$ and otherwise*

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

Aside from the extreme cases, the recursive formula is identical to the previous ones. Here again the proof is obtained through the same kind of proof than the ones of Section 2. See the Appendix A.3 for key ideas for the proof.

In Fig. 6 we exhibit the first values for $u_{n,k}$. Here again we obtain a sequence stored under the reference `OEIS A127717`. The bijection between both structures is direct, and thus we have new information about this sequence in the following.

Theorem 7. *Let $U(z,y)$ be the generating functions associated to $(u_{n,k})$. Then*

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

The proof is similar to the one of Theorem 1.

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

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

Comparing the latter result with the asymptotic behavior for the second algorithm based on combinadics, even if the dominant parts are equal, our approach seems not as efficient. But again, our complexity measure assumes the binomial coefficients being pre-computed.

4 Experimental comparison and algorithm improvements

For all algorithms, n and k being given, we proved that the average number of calls to the function `binomial` is of the same order, equivalent to n.

In Fig.7 we have represented the following statistics[7]. We take $n = 10000$ and in the abscissa-axis we let the value k ranges from 25 to 9975 with an iteration step of 50. In the ordinate-axis we have represented the average time in ms to unrank one combination (for each step we sampled, uniformly at random, 100 distinct combinations and we computed the average time for one). We have implemented all algorithms in C++ using the classical `GMP` library for big integers. The time needed by the algorithms presented in the previous sections are depicted in dashed lines: the red color is for the recursive Algorithm 1, blue corresponds to Algorithm 2, brown corresponds to its improvement, Algorithm 3, and finally green is for our factoradic Algorithm 5. We first remark that Algorithm 2 is worse than the 3 others whose computation times are very close. Further, asymptotically, the average number of calls to `binomial` is linear in n, whatever the value of k, this characteristics is not reflected in the experiments.

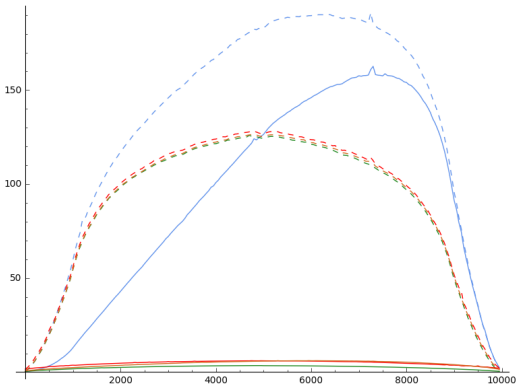

Fig. 7: Time (in ms) for unranking a combination, with $n = 10000$ and $k = 0..n$

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

Usually the way for evaluating $\binom{n}{k}$ is the following left to right computation: $n\cdot(n-1)/2\cdot(n-2)/3\cdots(n-k+1)/k$. Thus it needs $\Theta(k)$ arithmetic operations. It is done this way in `GMP`. There is a classical improvement used for iterated `binomial` evaluations. We compute the first coefficient we need, and then we correct this value to obtain the second one that is then modified to obtain the third one and so on. By studying all the previous algorithms we remark that the successive binomial coefficients are such that their parameters are only shifted by ± 1. Thus obtaining the next binomial coefficient is directly obtained with a multiplication and a division. For example, in Algorithm 5, we insert between line 3 and 4 the instruction $b :=$ `binomial`(n,k), then between line 9 and 10 we put $b := b\cdot(k-i)/(n-m)$ if $n \neq m$ else 1, and finally, between line 12 and 13 we put $b := b\cdot(n-m+1-k+i)/(n-m)$ if $n \neq m$ else 1.

Doing these improvements for all algorithms, we finally obtained the solid curves in Fig.7. Obviously all algorithms are faster than their first version, but Algorithm 2 is the less improved (in particular when $k > n/2$). The order of the computations is really penalizing in this approach. By focusing only on the new versions of Algorithms 1, 3 and 5, we obtain Fig. 1. We thus exhibit that our Algorithm 5, in green, is the most efficient while k ranges the interval.

As a final remark, we recall that we deal with very big numbers. For n and $k = \alpha n$ being large integers the unranking approaches deal with numbers that could need around $L = \log_2\binom{n}{k}$ bits to be written. Using Stirling approximation we have, by supposing $0 < \alpha < 1/2$ (and a symmetrical equation otherwise)

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

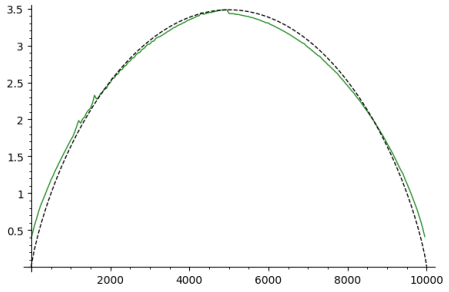

Fig. 8: Merge of Algorithm 5 and theoretical complexity

Theorem 8. *The bit complexity of the Algorithm 5 is*

$$\underset{\substack{n\to\infty\\ k=\alpha n}}{O}\left(n^2 L(n)\ \min\left(\alpha\log_2\left(\frac{1-\alpha}{\alpha}\right) - \log_2(1-\alpha)\,;(1-\alpha)\log_2\left(\frac{\alpha}{1-\alpha}\right) - \log_2(\alpha)\right)\right),$$

where $L(n)$ is a function in $\log_2 n$ depending on the algorithmic used for the integer multiplication.

The theorem is a by product of Lemma 5 saying that the average number of calls to `binomial` is equivalent to n that is amended to obtain the average number of multiplications or divisions being equivalent to $\Theta(n)$ through the previous improvements and the result about the bit complexity for the computation of $n!$ (cf. [4, p. 270]). In Fig.8 we merge the time complexity of our algorithm (in green) with a normalized curve (in black) from Theorem 8.

References

1. Beckenbach, E.F., Pólya, G.: Applied Combinatorial Mathematics. R.E. Krieger Publishing Company (1981)
2. Bodini, O., Genitrini, A., Peschanski, F.: A Quantitative Study of Pure Parallel Processes. Electronic Journal of Combinatorics **23**(1), P1.11, 39 pages (2016)
3. Bonet, B.: Efficient algorithms to rank and unrank permutations in lexicographic order. AAAI Workshop - Technical Report pp. 18–23 (2008)
4. Bostan, A., Chyzak, F., Giusti, M., Lebreton, R., Lecerf, G., Salvy, B., Schost, E.: Algorithmes Efficaces en Calcul Formel (2017), `https://hal.archives-ouvertes.fr/AECF/`, 686 pages. Édition 1.0
5. Buckles, B.P., Lybanon, M.: Algorithm 515: Generation of a Vector from the Lexicographical Index [G6]. ACM Trans. Math. Softw. **3**(2), 180–182 (1977)
6. Butler, B.: Function kSubsetLexUnrank, MATLAB central file exchange (2020)
7. Durstenfeld, R.: Algorithm 235: Random permutation. ACM **7**(7), 420– (1964)
8. Er, M.C.: Lexicographic ordering, ranking and unranking of combinations. International Journal of Computer Mathematics **17**(3-4), 277–283 (1985). https://doi.org/10.1080/00207168508803468
9. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Oliver & Boyd, London (1948)
10. Flajolet, P., Zimmermann, P., Van Cutsem, B.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science **132**(1-2), 1–35 (1994)
11. Kreher, D.L., Stinson, D.R.: Combinatorial Algorithms: generation, enumeration, and search. CRC Press (1999)
12. Martínez, C., Molinero, X.: A generic approach for the unranking of labeled combinatorial classes. Random Structures & Algorithms **19**(3-4), 472–497 (2001)
13. McCaffrey, J.: Generating the mth Lexicographical Element of a Mathematical Combination. MSDN (2004), `http://visualstudiomagazine.com/articles/2012/08/01/biginteger-data-type.aspx`
14. Myers, A.F.: k-out-of-n:g system reliability with imperfect fault coverage. IEEE Transactions on Reliability **56**(3), 464–473 (2007)
15. Nijenhuis, A., Wilf, H.S.: Combinatorial algorithms. Computer science and applied mathematics, Academic Press, New York, NY (1975)
16. Pascal, E.: Sopra una formula numerica. Giornale di Matematiche **25**, 45–49 (1887)
17. Ruskey, F.: Combinatorial Generation (2003)
18. Tamada, Y., Imoto, S., Miyano, S.: Parallel algorithm for learning optimal bayesian network structure. J. Mach. Learn. Res. **12**, 2437–2459 (2011)
19. The Sage Developers: SageMath, the Sage Mathematics Software System (Version 8.9), `http://www.sagemath.org`

A Appendix

A.1 Appendix dedicated to Section 1

Proof (of Theorem1). 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 1. 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 $u_{n,k} = \binom{n}{k} + u_{n-1,k-1} + u_{n-1,k}$.

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 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 by taking care of the extreme cases we have:

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

$$U(z,y) = \tilde{B}(z,y) + z\, y\, U(z,y) + z\, U(z,y).$$

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:

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

The latter result correspond to the distribution when k ranges from 0 to n. But we can further extract the coefficient of $z^n\, y^k$:

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

Using Hockey-Stick identity we obtain

$$\sum_{\ell=k}^{n-1} \binom{\ell}{k} = \binom{n}{k+1}, \qquad \sum_{\ell=n-k}^{n-1} \binom{\ell}{n-k} = \binom{n}{n-k+1} = \binom{n}{k-1}.$$

Thus we conclude

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

The result is proved.

A.2 Appendix dedicated to Section 2

Proof (ideas of Lemma 2). Once c_1 is given, all possible compositions of $(n-c_1)$ in k parts are possible for the values for $c_2, \ldots, c_k$, and for each composition, we need $(n-c_1)$ calls to `binomial`. Furthermore, for each rank we computes its reverse, with one call to `binomial`. Thus

$$u_{n,k} = \binom{n}{k} + \sum_{c_1=0}^{n-k} (n-c_1) \cdot \binom{n-c_1-1}{k-1}.$$

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

A.3 Appendix dedicated to Section 3

To prove the correction of our Algorithm 5 and also its average complexity, we rely on the following lemma.

Lemma 5. *The factoradics of the combinations k elements among n are all the tuples $(0, \ldots, 0, f_{n-k}, \ldots f_{n-1})$ with $n-k \geq f_{n-k} \geq f_{n-k+1} \geq \cdots \geq f_{n-1} \leq 0$.*

From the later lemma we deduce the proof of Proposition 3 and Lemma 4. In fact, the rank conversion consists in enumerating the factoradics corresponding to combinations, thus avoiding permutations that do not represent combinations. In fact we can directly omit them by using their numbers given by binomial coefficients.

Proof (ideas for Lemma 4). Here the last coefficient to be determined is f_{n-k}, it is the greatest one. Once it is given, all other values $f_{n-k+1}, \ldots, f_{n-1}$ are just determined through a weak composition of the available integers between 0

and f_{n-k}. In this case (in front of Lemma 2) we need weak compositions since the sequence $f_{n-k}, f_{n-k+1}, \ldots, f_{n-1}$ is weakly decreasing. Furthermore for each weak composition determining a combination, we need $(f_{n-k} + 1 + k)$ calls to `binomial` to determine the combination. We thus obtain:

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

An induction completes the proof.

A.4 Optimized factoradics algorithm

In this following new version of Algorithm 5 we present 4 improvements:

- The algorithm is not symmetrical anymore. In fact, we observe that when $k > 2$ then the previous version is faster than when $k < 2$. Thus when unranking in the range $0..n/2$, with some suitable substitutions we perform the unranking in the other range. As a consequence we get a small stall in this curve around $k = n/2$ in Fig. 1.
  Let C be a combination for $k < n/2$, then its complement $\bar{C}$ is somehow computed, but we store C and not $\bar{C}$ (lines 5-9 and 23-26 and 39-44).
- Only the first binomial coefficient is computed. Then it is adjusted to obtain the right value (lines 20 and 36).
- The last component is computed without any trial, just as a consequence of the other components (line 49).
- We avoid the call to Extract. In fact, when unranking the first part of a combination, the sequence of values we extract is an increasing sequence, we thus can completely avoid the expensive call to Extract just be saving how many values have already been extracted (lines 28 and 46 when $k > n/2$ and the more complicated cases lines 23-25 and lines 39-44).

```
1: function UNRANKING_FACTORADIC(n0, k, r)
2:     n := n0
3:     D := (0, ..., 0)                          ▷ the tuple containds k elements
4:     inverse := False
5:     if k < n/2 then
6:         inverse := True
7:         B := binomial(n, k)
8:         k := n − k;    r := B − 1 − r
9:         B := B · k/n
10:    else
11:        B := binomial(n − 1, k − 1)
12:    i := 0;    j := 0    d := 0
13:    m := 0    m2 := 0
14:    while d < k − 1 do
15:        if B > r then
16:            if i < k − 2 then
17:                if n − 1 − m = 0 then
18:                    B := 1
19:                else
20:                    B := B · (k − 1 − i)/(n − 1 − m)
21:            d := d + 1
22:            if inverse then
23:                for e from m2 to m + i do
24:                    D[j] := e
25:                    j := j + 1
26:                m2 := m + i + 1
27:            else
28:                D[i] := m + i
29:            i := i + 1
30:            n := n − 1
31:        else
32:            r := r − B
33:            if n − 1 − m = 0 then
34:                B := 1
35:            else
36:                B := B · (n − m − k + i)/(n − 1 − m)
37:            m := m + 1
38:    if inverse then
39:        for e from m2 to n + r + i − b do
40:            D[j] := e
41:            j := j + 1
42:        for e from n + r + i − B to n0 do
43:            D[j] := e
44:            j := j + 1
45:    else
46:        D[k − 1] := n + r + k − 1 − B
47:    return D
```