Article

Lexicographic Unranking of Combinations Revisited

Antoine Genitrini *[ID] and Martin Pépin[ID]

CNRS, Laboratoire de Paris 6—LIP6—UMR 7606, Sorbonne Université, F-75005 Paris, France;
martin.pepin@lip6.fr
* Correspondence: antoine.genitrini@lip6.fr

Abstract: In the context of combinatorial sampling, the so-called "unranking method" can be seen as a link between a total order over the objects and an effective way to construct an object of given rank. The most classical order used in this context is the lexicographic order, which corresponds to the familiar word ordering in the dictionary. In this article, we propose a comparative study of four algorithms dedicated to the lexicographic unranking of combinations, including three algorithms that were introduced decades ago. We start the paper with the introduction of our new algorithm using a new strategy of computations based on the classical factorial numeral system (or factoradics). Then, we present, in a high level, the three other algorithms. For each case, we analyze its time complexity on average, within a uniform framework, and describe its strengths and weaknesses. For about 20 years, such algorithms have been implemented using big integer arithmetic rather than bounded integer arithmetic which makes the cost of computing some coefficients higher than previously stated. We propose improvements for all implementations, which take this fact into account, and we give a detailed complexity analysis, which is validated by an experimental analysis. Finally, we show that, even if the algorithms are based on different strategies, all are doing very similar computations. Lastly, we extend our approach to the unranking of other classical combinatorial objects such as families counted by multinomial coefficients and *k*-permutations.

Keywords: unranking algorithm; combinatorial generation; combination; lexicographic order; complexity analysis

1. Introduction

One of the most fundamental combinatorial objects is called a combination. A *combination of k objects among n* is a subset of cardinality *k* of a set containing *n* objects. In many enumerating problems, it appears either as the main combinatorial structure or as a core building block because of its simplicity and counting characteristics.

In the 1960s while resolving some optimization problems related to 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 the "combinatorial number 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 reasons, this approach can be substituted to exhaustive generation once the latter is no longer possible due to the *combinatorial explosion* of the number of objects when their size increases. In the context of combinations, the explosion appears quickly (we recall that $\binom{2n}{n} \sim (2\pi n)^{-\frac{1}{2}} 4^n$). This generation strategy of a single element is classically called an *unranking* method. It appears as the fundamental problem in combinatorial generation such as in [1] and in optimization [2]. It is also used as a basic building block in scheduling problems [3], as well as, e.g., in software testing [4]. In other contexts, it appears as the core problem for the generation of complex structures: we can for example cite phylogenetics [5] and bioinformatics [6].

Prior to speaking of unranking one must specify a total order over the objects under consideration. The one that is frequently used is the *lexicographic* order; since it is humanly easy to handle, it has been extensively studied. However, as Ruskey mentioned [7] (p. 59), lexicographic generation is usually not the most efficient, thus particular care must be taken while unranking for this order. Knuth dedicated a section to the lexicographic generation of combinatorial objects, especially combination [8], relating it to the special case of Gray codes. Usually, while describing an unranking method, the dual approach for ranking, i.e., taking a built object and computing its rank, is also studied. The above reference to Knuth presents such algorithms. Other combinatorial objects are also studied in Ruskey's book about combinatorial generation [7] and in Skiena's book dedicated to the practical implementation of such algorithms [9]. For another ad hoc approach focused on the efficiency aspects of the ranking problem, one can refer to the work of Ryabko [10].

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 smaller ones. The method is detailed in the famous book by Nijenhuis and Wilf [11]. There, the authors were mostly interested in exhaustive generation and uniform random sampling, but some ideas about the decomposition schema are also applicable in the unranking context. The method has been 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 [13].

Aside from such generic approaches there exist several ad hoc algorithms. The paper by Kokosinski [14] presents a comparative study of several of them. The complexity analysis of these algorithms have been settled to be linear in n on average over all possible combinations when k is ranging from 0 to n.

However, to the best of our knowledge, these complexity analyses are only counting the number of calls to the function that computes a binomial coefficient, assuming that all the necessary coefficients have been 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 faithfully reflect the actual runtime of the algorithms and can we afford the pre-computation of that many binomial coefficients? 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 impractical. Most classical computer algebra systems (CAS) can unrank combinations in a reasonable time though, which suggests that there are better approaches. In Table 1, we present some of our experimental results. We 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 compare 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. We observe a great diversity of run times in Table 1. Unfortunately, Maple, Mathematica, and MATLAB (to deal with integers with arbitrary precision in MATLAB, we use symbolic computations by calling the function `sym`) are all closed source so we cannot take a look at their implementation to understand these differences. However, a detailed study of the different algorithms from the literature and some practical considerations presented in Section 4 provide some insights on this question. Sagemath is a special case to us. Since version 9.1, our algorithm has been implemented (in Python 3) and is used as the native algorithm.

Table 1. Average time (in ms) for the unranking of a combination of k elements among n (*too long* means that the average time is larger than 30,000 ms).

		Time in ms								
Sample		**Sagemath**			**Maple**		**Mathematica**		**Matlab**	
	Our Algo.	**v. 9.0**	**v. 9.2**		**v. 2020.0**		**v. 12.1.1.0**		**v. R2020b**	
Implem.	**in C**	**Their Algo.**	**New Algo.**	**Their Algo.**		**Our Algo.**	**Their Algo.**	**Our Algo.**	**Their Algo.**	**Our Algo.**
$n = 1000$ $k = 100$	0.05464	2.6045	2.9672	78.2		2.12	0.44176	4.3145	3996.6	3041.2
$n = 1000$ $k = 500$	0.06052	8.8903	2.4784	614		2.96	0.34608	3.9547	3520.6	3380.0
$n = 3000$ $k = 300$	0.17496	15.8968	8.7929	1180		13.2	5.9131	11.823	11,846	9315.2
$n = 3000$ $k = 1500$	0.27524	96.3589	8.0500	6130		19.2	4.9624	13.067	11,087	9879.4
$n = 10,000$ $k = 1000$	1.2554	191.03	31.665	too long		65.1	21.906	39.935	too long	too long
$n = 10,000$ $k = 5000$	2.3849	2 245.6	29.027	too long		97.9	29.916	46.452	too long	too long

Combinatorial Context

Throughout 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 $\{0, 1, \ldots, n-1\}$ as a finite increasingly sorted sequence containing k distinct elements.*

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* elements among *n*, but $(0, 2, 1)$ and $(0, 1, 2, 3)$ are not. Other representations are sometimes used, notably by using a two-letter alphabet, but we stick to the one given in Definition 1 in this paper.

There are several possible 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 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 (possibly empty) prefix such that $(a_0, \ldots, a_{p-1}) = (b_0, \ldots, b_{p-1})$ 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 $(a_{k-1}, \ldots, a_0)$ is smaller than $(b_{k-1}, \ldots, b_0)$ for the lexicographic order.*
- *If A is smaller than B for a given order, 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 $\{0, 1, \ldots \binom{n}{k} - 1\}$ and is such that A is the uth 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 solving the lexicographic combination unranking problem in Section 2. Section 3 is dedicated to the survey of three classical algorithms for the latter problem. The first one is based on the so-called recursive method and the two other algorithms are based on combinadics, which is a specific numeral system. It seems that it is the first time that both are compared and the reason one is better is explained. We also propose a new method for their analysis based on a generating function approach. Obviously, we obtain the same results as in the literature, but we manage also to obtain better asymptotic bounds than in the literature. In Section 4, we then compare their efficiency experimentally (the implementation and the exhaustive material used for repeating the experiments are all available at http://github.com/Kerl1 3/combination_unranking (accessed on 15 March 2021)) and propose a way to improve the computation of the binomial coefficients used in all four algorithms. Surprisingly, once the improvements have been implemented in all algorithms, we observe deep similarities in their computations, which is reflected by their observed run times. 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).

2. Unranking through Factoradics: A New Strategy

The classical methods to unrank combinations are relying on the combinatorial number system introduced in 1887, by E. Pascal [15] and later by D. H. Lehmer (detailed in the book [16] (p. 27)). We survey these classical algorithms in Section 3.2. In this section, we first present a new strategy based on another number system called *factoradics*. To the best of our knowledge, the latter has never been used for the unranking of combinations.

2.1. Link between the Factorial Number System and Permutations

We recall that 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. Its definition belongs to the folklore but already appeared in 1888 in [17].

Definition 4. *Let u be a positive integer and let n be 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 *(or factoradic) of u (note that $f_0 = 0$ for all u).*

Taking the number u = 2021 as an example, we obtain the following decomposition: $2021 = 0 \times 0! + 1 \times 1! + 2 \times 2! + 0 \times 3! + 4 \times 4! + 4 \times 5! + 2 \times 6!$. 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 of 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 [18].

Fact 1. *For all $0 \leq u < n!$, UNRANKINGPERMUTATION(n, u) returns the uth permutation in lexicographic order among the $n!$ permutations of n elements.*

Algorithm 1 Unranking a permutation.

1: **function** UNRANKINGPERMUTATION(n, u)
2:     $F \leftarrow$ 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:     $P \leftarrow [0, 1, \dots, n-1]$
3:     $L \leftarrow [0, \dots, 0]$ ▷ k components
4:     **for** i **from** 0 **to** $k-1$ **do**
5:         $L[i] \leftarrow P[F[n-1-i]]$
6:         remove$(P, F[n-1-i])$
7:     **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 obtain this permutation, we read the factoradic from right to left, and extract iteratively from the list $(0, 1, \dots, 8-1)$ 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 fourth element, which is 6, and so on.

Note that, for the sake of clarity, we present the function EXTRACT using a list for P, but a better data structure must be used to achieve good performance. Good candidates

are dynamic balanced trees, as presented in [19], or multisets with elements of weight 1 or 0, as presented in the Appendix of [20], 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 permutations in the lexicographic order; put differently: Durstenfeld's algorithm [21] cannot be easily adapted for the lexicographic order.

2.2. Combinations Unranking through Factoradics

We start with the definition of our algorithm to unrank *combinations* via the use of 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 6. *Let n and k be two integers with $0 \leq k \leq n$. We define $\mathcal{P}_{n,k}$ as the function which maps the combination $(\ell_0, \ell_1, \ldots, \ell_{k-1})$ to the size-n permutation $(\ell_0, \ell_1, \ldots, \ell_{k-1}, d_k, \ldots, d_{n-1})$ where the integers d_i are such that $d_k < d_{k+1} < \cdots < d_{n-1}$ and $\{\ell_0, \ldots, \ell_{k-1}, d_k, \ldots, d_{n-1}\} = \{0, 1, \ldots, n-1\}$.*

For instance, by definition, for $n = 5$ and $k = 3$, the permutations associated with 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 function $\mathcal{P}_{n,k}$ returns the smallest size-n permutation for the lexicographic order whose prefix is the given combination.

Proposition 1. *For all integers $0 \leq k \leq n$, the function $\mathcal{P}_{n,k}$ 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 with combinations $(0, 1)$ and $(0, 1, 2)$ by, respectively, $\mathcal{P}_{2,5}$ and $\mathcal{P}_{3,5}$. In fact, there are exactly six combinations associated with the latter permutation, but for different values of k. The proof of Proposition 1 is straightforward.

Fact 2. *For any integer $m \geq 0$, the number of sequences $(f_i)_{0 \leq i < n}$ satisfying $n - k \geq f_{n-k} \geq f_{n-k+1} \geq \cdots \geq f_{n-1} \geq m$ is given by $\binom{n-m}{k}$.*

In fact, we get this result using a classical cardinality argument: a sequence of integers of the form $x_0 = 0 \leq x_1 \leq x_2 \leq \cdots \leq x_k \leq x_{k+1} = \ell$ corresponds to a weak composition (consider the differences $x_{i+1} - x_i$) of the integer ℓ into $k + 1$ terms. The number of such compositions is given by $\binom{\ell+k}{k}$. Hence, the number of sequences matching the description of Fact 2 is given by $\binom{n-k-m+k}{k}$.

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

Lemma 1. *For any given $0 \leq k \leq n$, 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 $(0, \ldots, 0, f_{n-k}, \ldots, f_{n-1})$ with $n - k \geq f_{n-k} \geq f_{n-k+1} \geq \cdots \geq f_{n-1} \geq 0$.*

Proof. Let u be an integer whose factoradic is $(0, \ldots, 0, f_{n-k}, \ldots f_{n-1})$ as in the lemma. Due to the constraint $n - k \geq f_{n-k} \geq f_{n-k+1} \geq \cdots \geq f_{n-1} \geq 0$, the permutation corresponding to u has for prefix of length k the sequence $(f_{n-1}, f_{n-2} + 1, f_{n-3} + 2, \ldots, f_{n-k} + k - 1)$ 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 2 completes the proof. □

Thus, to convert the combination rank u into its corresponding permutation rank u', it is sufficient to find the uth sequence satisfying Lemma 1 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 2. As a consequence, we get that the first $\binom{n-1}{k-1}$ such sequences (in co-lexicographic order) have $f_{n-1} = 0$ and that the $\binom{n-1}{k}$ following have $f_{n-1} \geq 1$.

Once again, we opt 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 m into F on Line 8 we could directly compute the ith component of the combination as $m + i$ by using the remark at the beginning of the proof of Lemma 1. In Section 4.2, we provide a more efficient way to implement this algorithm including this particular optimization and other improvements.

Proposition 2. *The function* UNRANKINGCOMBINATION(n, k, u) *computes the uth combination of k elements among n in lexicographic order.*

Proof. The algorithm, and thus its proof, relies heavily on Fact 2. 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 2 with $k = k - i$ and $n = n - i$.

With this invariant at hand, the combinatorial argument behind the condition $u < \binom{n-m-i-1}{k-i-1}$ in the algorithm becomes more apparent; that is, 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 possible value for m. □

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.

The usual way to evaluate the efficiency of such an algorithm is to count the number of times the `binomial` function is called (see, e.g., the book [7] (p. 66) or the papers [22,23]). 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 $n-k$. In these cases, the number of calls to `binomial` is n.

To study the *average* number of calls to `binomial`, when u describes the whole range from 0 to $\binom{n}{k} - 1$, we introduce the following cumulative sequence.

Lemma 2. *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, we give the first values of $u_{n,k}$, when n is less than 9. The obtained sequence is stored under the reference `OEIS A127717` (throughout this paper, a reference `OEIS A`$\cdots$ points to Sloane's Online Encyclopedia of Integer Sequences www.oeis.org (accessed on 15 March 2021)). The bijection between both structures is direct, and thus we provide new information about this sequence in the following.

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

n \ 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

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 the rest of the execution corresponds to unranking a combination of $k - 1$ elements among $n - 1$. This is accounted for 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 a combination of k elements among $n - 1$, which is accounted for by $u_{n-1,k}$. □

We turn to bivariate generating functions to encode the sequence $(u_{n,k})$ as a power series and express the above recurrence as a simple equation satisfied by this function, which can be solved explicitly. The reader can refer to the two books of Flajolet and Sedgewick [24,25] for a global presentation of such method.

Theorem 3. *Let $U(z, y)$ be the generating functions associated with $(u_{n,k})$. Then,*

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

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

Proof. The first step of the proof consists in exhibiting the ordinary generating function associated with $U(z, y)$. To obtain a functional equation satisfied by U, we start from the result presented in Lemma 2. The extreme cases are $u_{n,0} = 0$ and $u_{n,n+i} = 0$ for all n and $i > 0$. 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 of binomial coefficients. Let us denote it by $B(z, y)$, which is equal to

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

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:

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

$$= \left(\sum_{r \geq 0}z^r(1 + y)^r\right) \cdot \left(\sum_{r \geq 0}z^r(1 + y)^r - \sum_{r \geq 0}z^r\right)$$

$$= \left(\sum_{r \geq 0}z^r(1 + y)^r\right) \cdot \left(1 - 1 + \sum_{r \geq 1}z^r\left((1 + y)^r - 1\right)\right).$$

By extraction, the coefficient in front of z^n is:

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

$$= n(1 + y)^n - \frac{(1 + y)^n - 1}{y}.$$

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 1. *To unrank a combination of k elements among n, the function* UNRANKINGCOMBINATION$(n, k, \cdot)$ *needs on average* $u_{n,k}/\binom{n}{k}$ *calls to the function* `binomial`. *For n being large and k being of the form αn for $0 < \alpha < 1$, this 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 3. Since we have the exact value of $u_{n,k}$, the mean value can easily be computed in other cases such as $k = o(n)$ or $k = n - o(n)$.

3. Classical Unranking Algorithms

This section is dedicated to the presentation of a survey of the usual approaches to unrank combinations in the lexicographic order. The motivation behind this section is threefold. First, the classical algorithms were developed in the 1970s and 1980s and it is a hard task to get access to the papers we 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 such as in the previous section. Finally, as shown 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 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 [11]. 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 $\{0, 1, \ldots, n-1\}$ 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: 　　$L \leftarrow$ RECGENERATION(n, k, u)
3: 　　$L' \leftarrow [0, \ldots, 0]$　　　　　▷ k components
4: 　　**for** i **from** 0 **to** $k-1$ **do**
5: 　　　　$L'[i] \leftarrow n - 1 - L[k-1-i]$
6: 　　**return** L'

1: **function** RECGENERATION(n, k, u)
2: 　　**if** $k = 0$ **then**
3: 　　　　**return** $[\,]$
4: 　　**if** $n = k$ **then**
5: 　　　　**return** $[0, 1, 2, \ldots, k-1]$
6: 　　$b \leftarrow$ `binomial`$(n-1, k-1)$
7: 　　**if** $u < b$ **then**
8: 　　　　$R \leftarrow$ 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 1. *An alternative choice would have been to test whether $b < \binom{n-1}{k}$ on 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 is similar.*

Proposition 3. *The function* RECGENERATION(n, k, u) *computes the uth combination of k elements among n for the reverse co-lexicographic order.*

Corollary 2. *The function* UNRANKINGRECURSIVE(n, k, u) *computes the uth combinations of k elements among n for the lexicographic order.*

Proof. The proposition is proved by induction and the corollary is a direct observation given in [26] (p. 47). □

Here, again, we are interested in the average number of calls to the `binomial` function, when u describes the whole range of integers from 0 to $\binom{n}{k} - 1$. Ruskey justified such a measure by supposing the table of all binomial coefficients pre-computed, thus each call is equivalent. In Section 4, we discuss this measure. We introduce the sequence (to simplify the notations, we use several times the notations $u_{n,k}$ and $U(z)$ for distinct sequences and series) $u_{n,k}$ equal to the cumulative number of calls to `binomial` for the whole range of possible values for u.

Lemma 3. *Let $u_{n,k}$ be the cumulative number of calls to `binomial` for the recursive unranking 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 $u_{n-1,k}$. □

In the following Table 3, we give the first values of $u_{n,k}$, when n is less than 9.

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

n \ 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

Theorem 4. *Let $U(z, y)$ be the ordinary generating function associated with $(u_{n,k})$, 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 4 is very similar to the one of Theorem 3.

Proof. The first step of the proof consists in exhibiting the ordinary generating function associated with $U(z, y)$. To obtain the equation for U, we start from the result presented in Lemma 3. The extreme cases are $u_{n,0} = 0$ and $u_{n,n+i} = 0$ for all n and $i \geq 0$. 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. Let us denote it by $B(z, y)$. It satisfies:

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

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 of the proof consists in extracting the coefficient $u_{n,k}$.

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

$$= \left(\sum_{r \geq 0} z^r (1 + y)^r \right) \cdot \left(\sum_{r \geq 0} z^r (1 + y)^r - \sum_{r \geq 0} z^r - \sum_{r \geq 1} z^r y^r \right)$$

$$= \left(\sum_{r \geq 0} z^r (1 + y)^r \right) \cdot \left(1 - 1 + \sum_{r \geq 1} z^r \left((1 + y)^r - 1 - y^r \right) \right).$$

By extraction the coefficient in front of z^n is:

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

$$= \sum_{\ell=0}^{n-1} (1 + y)^n - (1 + y)^\ell - y^{n-\ell} (1 + y)^\ell$$

$$= n(1 + y)^n - \frac{(1 + y)^n - 1}{y} - y(1 + y)^n + y^{n+1}.$$

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

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

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

which completes the proof. $\square$

This sequence $u_{n,k}$ is a shifted version of the sequence stored under the reference `OEIS A059797`. We thus can complete its properties in the `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 that $u_{n,k} = u_{n,n-k}$, reflecting the symmetry of the binomial coefficients.

Corollary 3. *The function* UNRANKINGRECURSIVE$(n, k, \cdot)$ *needs on 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, 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. The recursive approach is a drawback for some programming languages that do not handle recursion efficiently (due to the depth of the stack). Naturally, other strategies have been suggested in the literature.

3.2. Unranking through Combinadics

In 1887, E. Pascal [15] and later D. H. Lehmer (detailed in the book [16] (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 so-called combinatorial number system, or combinadics. The decomposition relies on binomial coefficients.

Fact 5. *Let* $n \geq k$ *be two positive integers. 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 (we extend the definition of binomial coefficients with* $\binom{n}{k} = 0$ *when* $k > n$)

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

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 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.

In 2004, using this representation, McCaffrey exhibited, in the MSDN article [27], an algorithm to build the uth element (in lexicographic order) of the combinations of k elements among n. However, in fact, this algorithm was already published by Kreher and Stinson [26] (p. 47) and can also be seen as an extension of the work of Lehmer. This algorithm is interesting as it corresponds to the previous implementation used in the mathematics software system `Sagemath` [28] (the previous unranking algorithm from Sagemath is stored in the Software Heritage Archive swh:1:cnt:c60366bc03936eede6509b23307321faf1035e23;lines=539-605). In the beginning of 2020, we replaced the Sagemath implementation by the algorithm presented in Section 2 (the new unranking algorithm from Sagemath is stored in the Software Heritage Archive swh:1:cnt:b2a68056554dbf90fa55e76820f348d9d55019e3;lines=539-653 (accessed on 15 March 2021)).

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 $u \geq \binom{c_k}{k}$, and

then the remaining part $u - \binom{c_k}{k}$ is smaller than $\binom{n-1}{k-1}$ so it can be decomposed recursively into a combinadic with $k-1$ components smaller than $n-1$. McCaffrey's algorithm is described in Algorithm 4.

Table 4. Combinadics and their combinations for $n = 6$ and $k = 2$.

Rank, u	Reversed Rank, $u' = \binom{6}{2} - 1 - u$	Combinadic of u'	Combination of Rank u
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)

Algorithm 4 Unranking a combination.

1: **function** UNRANKINGVIACOMBINADIC(n, k, u)
2: 　　$L \leftarrow [0, \ldots, 0]$ 　　　　　　　　　　　　　　　　　　　　▷ k components
3: 　　$u' \leftarrow \mathtt{binomial}(n, k) - 1 - u$
4: 　　$v \leftarrow n$
5: 　　**for** i **from** 0 **to** $k - 1$ **do**
6: 　　　　$v \leftarrow v - 1$
7: 　　　　$b \leftarrow \mathtt{binomial}(v, k - i)$
8: 　　　　**while** $u' < b$ **do**
9: 　　　　　　$v \leftarrow v - 1$
10: 　　　　　$b \leftarrow \mathtt{binomial}(v, k - i)$
11: 　　　$u' \leftarrow u' - b$
12: 　　　$L[i] \leftarrow n - 1 - v$
13: 　　**return** L

As noted while explaining Table 4, we work with the reverse of the rank u (see Line 3 in the algorithm) in order to unrank combinations in lexicographic order. The presented algorithm is also close to Er's algorithm [23] 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, although the core of the algorithm is reverse co-lexicographic. The correctness of the algorithm is stated in [26].

Again, we express the complexity of this algorithm as its number of calls to the `binomial` function. First note that, the values n and k being given, the worst cases are obtained when v gets as small as possible at the end of the loop, thus for all u whose combinadic satisfies $c_1 = 0$. 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 again 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 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: $u_{n,0} = 1$ and $u_{n,n+i} = 0$ for all n and $i > 0$, and otherwise*

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

Table 5 presents the sequence given in Lemma 4. 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.

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

k n	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

Proof. Note that $n - c_k$ calls to `binomial` are necessary to determine c_k, then $c_k - c_{k-1}$ calls to determine $c_{k-1}, \ldots$, and finally $c_2 - c_1$ 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 6. *Let $U(z, y)$ be the generating function associated with $(u_{n,k})$. 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 for Theorem 3. The values for $u_{n,k}$ are a bit different due to the extreme cases $u_{n,0}$.

Corollary 4. *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, another algorithm based on combinadics is given in [22]. 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, the 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 previous algorithm: the number of sequences $0 \le c_1 < c_2 < \cdots < c_k < n$ with $c_1 \ge j$ is equal to $\sum_{i=0}^{j} \binom{n-i-1}{k-1}$, hence the accumulation 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 1 at the beginning of this section.

The first point mentioned above is an improvement over Algorithm 4, but in fact this second algorithm is penalized by the extra addition that it has to perform and then undo on 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 appears 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:     $L \leftarrow [0, \ldots, 0]$ ▷ k components
3:     $r \leftarrow 0$
4:     **for** i **from** 0 **to** $k - 2$ **do**
5:         **if** $i \ne 0$ **then** $L[i] \leftarrow L[i-1]$
6:         **else** $L[i] \leftarrow -1$
7:         **while** true **do**
8:             $L[i] \leftarrow L[i] + 1$
9:             $b \leftarrow$ `binomial`$(n - L[i] - 1, k - i - 1)$
10:             $r \leftarrow r + b$
11:             **if** $r > u$ **then** exit the loop

12:             $r \leftarrow r - b$

13:     $L[k-1] \leftarrow L[k-2] + u - r + 1$
14:     **return** L

UNRANKINGVIACOMBINADIC2(n, k, u) is a lexicographic unranking for combinations. This algorithm was presented by Buckles and Lybanon [22], and its correctness is presented in [26]. Finally, note it is approximately the implementation in `Matlab` [29] whose code was also presented by Ruskey ([7], p. 65). The latter approach also does trials to find the last coefficient of the combination instead of computing it directly on Line 13.

Lemma 5. *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 4, 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, and Theorem 7 gives their asymptotic behavior.

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

	k								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
n									
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

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

$$U(z, y) = \frac{z^2 y^2}{(1 - z)^2 (1 - z - zy)^2}; \text{ thus}$$

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

Corollary 5. *The average number of calls to* binomial *in Algorithm 5 for n being large and k being of the form αn for $0 < \alpha < 1$ 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 4 on average. Let us conclude this part with an interesting remark.

Remark 2. *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. However, as explained after Algorithm 3, we could have first been 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 follow exactly the same approach as Algorithm 4.*

4. Improving Efficiency and Realistic Complexity Analysis

Finally, we show that the complexity model used above is no longer adequate. 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-bit integers; see the work of McCaffrey [27], who seems to be the first to introduce combination unranking using big integer computations), this is no longer a valid model as

big integers are now more widely used. We discuss the impact of big integer arithmetic and some optimizations that significantly improve the performance of all algorithms.

From the two last sections, we know the different algorithms that are mostly used in practice. However, having in mind the results exhibited in Table 1, implementations may vary a lot. Obviously, using a specialized library for big integer arithmetic such as the `GMP` `C` library is the best way to carry out fast operations, but the distinct behaviors observed in the table suggests that deeper differences exists between the various implementations than just the choice of the library.

For all algorithms, we prove that the average number of calls to the `binomial` function 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 comparing these results with the actual run time of the algorithms.

4.1. First Experiments to Visualize the Time Complexity in a Real Context

In the two next plots in Figure 1, we present the average time needed for the computations of 500 combinations for each pair $(n, k = n/2)$ where *n* ranges from 250 to 10,000 with a step size of 250. The choice $k = n/2$ was made because it corresponds to the worst complexity cases when *k* varies from 0 to *n*. To conduct our experiments, we implemented all algorithms in `C` using the classical `GMP` library for big integers arithmetic (the implementation and the exhaustive material used for repeating the experiments are available at http://github.com/Kerl13/combination_unranking (accessed on 15 March 2021)). The experiments were run on a standard laptop PC with an I7-8665U CPU, 32Gb RAM running Ubuntu Linux 2020. Moreover, except for Table 1, which required running different computer algebra systems and thus the X server, the experiments were run in the console while the X server and other time-consuming services (wifi, sound, etc.) were off. On the leftmost plot, we present 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 almost exactly the same time to unrank a combination. In the rightmost plot, we present a version with memoization (memoization is an optimization technique consisting in storing the results of expensive function calls and returning the cached values when the function is called again with the same arguments; this technique is sometimes called "tabling") 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. In our plot, we present only the time used for the second step. We remark that the recursive Algorithm 3 is not as efficient as the others. Below, we show that this is due to its recursive nature and that this can be easily avoided. As explained above, Algorithm 5 should be less efficient than Algorithm 4, which is indeed apparent in the plots. Our Algorithms 2 and 4 are the most efficient when used in this setup.
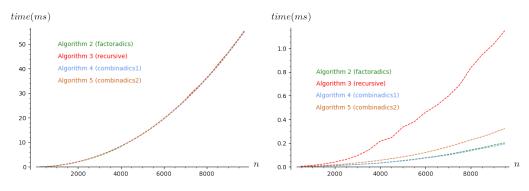

Figure 1. Time (in ms) for unranking a combination, with $n = 250 \ldots 10,000$ and $k = n/2$.

While the number of evaluations of binomial coefficients is linear in *n*, it is clear that it is not the case for the time complexity of the algorithms (see the leftmost plot). However,

once the binomial coefficients have been pre-computed, 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 no longer the case when n is of the order of several thousands (for the experiments in Figure 1, we only computed the necessary binomial coefficients using a lazy approach). Such use cases did not occur when the methods (e.g., [22,23]) were derived. However, now that big integers are more widely used, it becomes reasonable to unrank combinations of large size and it is thus necessary to take the cost of arithmetic operations into account. The first use of big integers for combination unranking appears to date back to 2004 [27].

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, which is applicable to all 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 [30] (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 obtain 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 next. For instance, in Algorithm 2, just before incrementing i on 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 on 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 $\Theta(1)$ multiplication of a big integer by a small integer. 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. 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 right value from the set $\{0, 1, 2, \dots, n-1\}$. 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 to its mth 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.
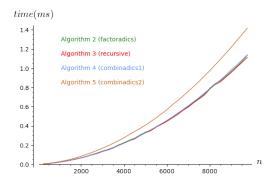
Before going on with the efficiency comparisons, we use the remarks related to the binomial coefficient computation to improve Algorithms 3–5. Furthermore, to make the comparison fair, we used a variant of Algorithm 3 for the recursive approach in which the array storing the result is allocated with the right size at the beginning of the execution, as in the other algorithms. In addition, 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.
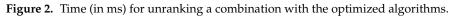
In UNRANKTR, the variable i represents the position in L of the next value to be computed and the variable m represents the next candidate to be the value of $L[i]$. The invariant satisfied by b is that UNRANKTR is always called with $b = n \cdot \binom{n-1}{k-1}$.

Algorithm 6 Unranking a combination with optimization.

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

We propose a second time efficiency comparison for some algorithms with their optimizations in Figure 2. Comparing this plot with the leftmost one of Figure 1, we note that, when $n = 10,000$, the algorithms run approximately 45 times faster. Again, we note that Algorithm 5 is still less efficient than the others, which all seem to be equivalent.

Algorithm 7 Recursive method with optimizations.

1: **function** OPTIMIZEDUNRANKING-
      RECURSIVE(n, k, u)
2:   $L \leftarrow [0, \ldots, 0]$ $\triangleright$ k components
3:   $b \leftarrow \texttt{binomial}(n, k)$
4:   UNRANKTR$(L, 0, 0, n, k, u, b)$
5:   **return** L

1: **function** UNRANKTR(L, i, m, n, k, u, b)
2:     **if** $k = 0$ **then** do nothing
3:     **else if** $k = n$ **then**
4:         **for** j **from** 0 **to** $k - 1$ **do**
5:             $L[i + j] \leftarrow m + j$
6:     **else**
7:         $b \leftarrow b / n$
8:         **if** $u < b$ **then**
9:             $L[i] \leftarrow m$
10:           $b \leftarrow (k - 1) \cdot b$
11:           UNRANKTR$(L, i + 1, m + 1, n - 1, k - 1, u, b)$
12:         **else**
13:           $u \leftarrow u - b$
14:           $b \leftarrow (n - k) \cdot b$
15:           UNRANKTR$(L, i, m + 1, n - 1, k, u, b)$

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

To understand the behavior of the curves of the above plot, we introduce another way to analyze the time complexity in Figure 3. Now, we take $n = 10{,}000$ and we let k range from 250 to 9750 with an iteration step of 250. For each step, we present an average value for 500 tests. Again (in the leftmost plot), the dashed lines correspond to the first version of each algorithm and the solid lines to the optimized versions. In the rightmost plot, we only focus on the optimized versions of the four algorithms. We remark that the worst time complexity is obtained when $k = n/2$. In addition, Algorithm 6 and the tail-recursive Algorithms 4 and 7 behave in almost the same way. Whereas 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 $k = 25 \ldots 9975$.

Thus, the curves for Algorithms 4, 6, and 7 (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 different base cases. In the case of the recursive and factoradic-based algorithms, the similarity goes further. Algorithm 7, being tail-recursive, it can be automatically translated into the imperative style (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) 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 speed-up can be achieved by re-using the value of the most recently computed binomial coefficient.

4.3. Realistic Complexity Analysis

We now propose a more precise complexity analysis based on a more realistic cost model. Recall that we are dealing 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, although 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 8.

Theorem 8. *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 was 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 our complexity model.
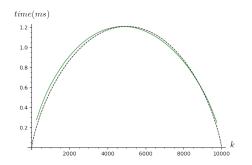

Figure 4. Merge of Algorithm 2 and its theoretical complexity.

5. Extensions of the Algorithmic Context

5.1. Objects Counted by Multinomial Coefficients

Let n and m be two positive integers and let $K = (k_1, k_2, \ldots, k_m)$ be a finite sequence of non-negative integers whose sum equals n. The multinomial coefficient $\binom{n}{k_1, \ldots, k_m}$ counts the number of ways of depositing n distinct objects into m distinct buckets such that there are k_i objects in the ith bucket. It can also be interpreted as combination with repetitions as follows. Consider we have a (large enough) pool of m kinds of different objects and we must pick a finite sequence of n objects from this pool 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 1 (related to combinations) states that 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 this link more formally.

Proposition 4. *Let n and m be two positive integers and let $K = (k_1, k_2, \dots, k_m)$ be a sequence of non-negative integers such that $\sum_i k_i = n$. 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 algorithm relies on the classical formula expressing the multinomial coefficient as a product of binomial coefficients.

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

Algorithm 8 Unranking a combination with repetitions.

1: **function** UNRANKINGCOMBINATIONWITHREPETITIONS($n, K = (k_1, \dots, k_m), u$)
2:     $F \leftarrow [0, \dots, 0]$ ▷ 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 of s by t.

Proposition 5. *The function* UNRANKINGCOMBINATIONWITHREPETITIONS$(n, (k_1, \dots, k_m), u)$ *computes the uth 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 with the combination with repetitions we are interested in. It remains then to unrank a permutation.

Proof. Based on Proposition 4, 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 down to k_1. The correctness of our algorithm 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 of the rank $u' \bmod \binom{k_m + \cdots + k_i}{k_i}$ 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.

  □

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 given by a combination of k elements among n elements, and an ordering of these k elements. The number of k-permutation over a set of n elements is given by:

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

Proposition 6. *The function* UNRANKINGKPERMUTATION(n, k, u) *returns the result of a call to* UNRANKINGCOMBINATIONWITHREPETITIONS$(n, (1, \ldots, 1, n-k), u)$*. Thus, it computes the uth 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. Conclusions

In the present article, we first exhibit, in Section 2, a new algorithm based on the classical factoradic numeral system for the lexicographic unranking of combinations. We also give a first step complexity analysis of the algorithm.

Section 3 is dedicated to the survey of three classical algorithms. The first one is the algorithm based on the so-called recursive method. The other two algorithms are based on combinadics, which is a numeral system particularly suited to representing combinations. To the best of our knowledge, it is the first time that both algorithms are compared. During the survey, we extend our complexity analysis based on a generating function approach, as presented in Section 2, in order to obtain a uniform presentation of the analysis of all four algorithms. Obviously, we thus reprove their average complexity results but with more details. Then, in Section 4, we compare their efficiency experimentally and propose a way to improve all these algorithms based on classical formulas of the binomial coefficient.

As a surprising result, we find that all the usual algorithms 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 detail the core computations that are necessary to unrank combinations, it is possible to significantly improve all algorithms. This understanding, joined with a detailed and realistic theoretical complexity analysis, leads to a prediction of the run time of the algorithm that closely matches 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.

Finally, in Section 5, we extend our approach to solve the problem of unranking structures enumerated by multinomial coefficients and objects counted by the k-permutations of n elements (also called arrangements).

References

1. Shablya, Y.; Kruchinin, D.; Kruchinin, V. Method for Developing Combinatorial Generation Algorithms Based on AND/OR Trees and Its Application. *Mathematics* **2020**, *8*, 962. [CrossRef]
2. Grebennik, I.; Lytvynenko, O. Developing software for solving some combinatorial generation and optimization problems. In Proceedings of the 7th International Conference on Application of Information and Communication Technology and Statistics in Economy and Education, Sofia, Bulgaria, 3–4 November 2017; pp. 135–143.
3. Tamada, Y.; Imoto, S.; Miyano, S. Parallel Algorithm for Learning Optimal Bayesian Network Structure. *J. Mach. Learn. Res.* **2011**, *12*, 2437–2459.
4. Myers, A.F. *k*-out-of-*n*:G System Reliability With Imperfect Fault Coverage. *IEEE Trans. Reliab.* **2007**, *56*, 464–473. [CrossRef]
5. Bodini, O.; Genitrini, A.; Naima, M. Ranked Schröder Trees. In Proceedings of the Sixteenth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2019, San Diego, CA, USA, 6 January 2019; pp. 13–26. [CrossRef]
6. Ali, N.; Shamoon, A.; Yadav, N.; Sharma, T. Peptide Combination Generator: A Tool for Generating Peptide Combinations. *ACS Omega* **2020**, *5*, 5781–5783. [CrossRef] [PubMed]
7. Ruskey, F. *Combinatorial Generation*; 2003. (unpublished). Available online: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.5967 (accessed on 15 March 2021).
8. Knuth, D.E. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms*; Addison-Wesley Professional: Boston, MA, USA, 2011.
9. Skiena, S. *The Algorithm Design Manual, Third Edition*; Texts in Computer Science; Springer: Berlin/Heidelberg, Germany, 2020. [CrossRef]
10. Ryabko, B.Y. Fast enumeration of combinatorial objects. *Discret. Math. Appl.* **1998**, *8*, 163–182. [CrossRef]
11. Nijenhuis, A.; Wilf, H.S. *Combinatorial Algorithms*; Computer Science and Applied Mathematics; Academic Press: New York, NY, USA, 1975.
12. Flajolet, P.; Zimmermann, P.; Van Cutsem, B. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.* **1994**, *132*, 1–35. [CrossRef]
13. Martínez, C.; Molinero, X. A generic approach for the unranking of labeled combinatorial classes. *Random Struct. Algorithms* **2001**, *19*, 472–497. [CrossRef]
14. Kokosinski, Z. Algorithms for Unranking Combinations and their Applications. In Proceedings of the Seventh IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems, Washington, DC, USA, 19–21 October 1995; Hamza, M.H., Ed.; IASTED/ACTA Press: Calgary, AB, Canada, 1995; pp. 216–224.
15. Pascal, E. Sopra una formula numerica. *G. Di Mat.* **1887**, *25*, 45–49.
16. Beckenbach, E.F.; Pólya, G. *Applied Combinatorial Mathematics*; R.E. Krieger Publishing Company: Malabar, FL, USA, 1981.
17. Laisant, C.A. Sur la numération factorielle, application aux permutations. *Bull. Soc. Math. Fr.* **1888**, *16*, 176–183. [CrossRef]
18. Fisher, R.A.; Yates, F. *Statistical Tables for Biological, Agricultural and Medical Research*; Oliver & Boyd: London, UK, 1948.
19. Bonet, B. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI Workshop on Search in AI and Robotics—Technical Report*; The AAAI Press: Menlo Park, CA, USA, 2008; pp. 142–151.
20. Bodini, O.; Genitrini, A.; Peschanski, F. A Quantitative Study of Pure Parallel Processes. *Electron. J. Comb.* **2016**, *23*, 39. [CrossRef]
21. Durstenfeld, R. Algorithm 235: Random Permutation. *Commun. ACM* **1964**, *7*, 420. [CrossRef]
22. Buckles, B.P.; Lybanon, M. Algorithm 515: Generation of a Vector from the Lexicographical Index [G6]. *ACM Trans. Math. Softw.* **1977**, *3*, 180–182. [CrossRef]
23. Er, M.C. Lexicographic ordering, ranking and unranking of combinations. *Int. J. Comput. Math.* **1985**, *17*, 277–283. [CrossRef]
24. Flajolet, P.; Sedgewick, R. *Analytic Combinatorics*; Cambridge University Press: Cambridge, UK, 2009.
25. Flajolet, P.; Sedgewick, R. *An Introduction to the Analysis of Algorithms,* 2nd ed.; Addison Wesley: Boston, MA, USA, 2013.
26. Kreher, D.L.; Stinson, D.R. *Combinatorial Algorithms: Generation, Enumeration, and Search*; CRC Press: Boca Raton, FL, USA, 1999.
27. McCaffrey, J. Generating the mth Lexicographical Element of a Mathematical Combination. MSDN. 2004. Available online: http://docs.microsoft.com/en-us/previous-versions/visualstudio/aa289166(v=vs.70) (accessed on 15 March 2021).
28. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.2)*. 2020. Available online: https://www.sagemath.org/ (accessed on 15 March 2021).
29. Butler, B. Function kSubsetLexUnrank, MATLAB Central File Exchange. 2015. Available online: http://fr.mathworks.com/matlabcentral/fileexchange/53976-combinatorial-numbering-rank-and-unrank (accessed on 15 March 2021).
30. Bostan, A.; Chyzak, F.; Giusti, M.; Lebreton, R.; Lecerf, G.; Salvy, B.; Schost, E. *Algorithmes Efficaces en Calcul Formel*, 10th ed.; Printed by CreateSpace: Palaiseau, France, 2017; 686p.