Generating Permutations and Combinations in Lexicographical Order

Alon Itai
Computer Science Department
Technion Haifa Israel

Abstract

We consider producing permutations and combinations in lexicographical order. Except for the array that holds the combinatorial object, we require only $O(1)$ extra storage. The production of the next item requires $O(1)$ amortized time.
Keywords: Permutations, combinations, amortized complexity.

1 Introduction

Let n and $p < n$ be integers, and let N denote $\{1, \ldots, n\}$. A *p-combination* is a subset of N of size p. A p-combination, σ, may be represented by a boolean array of size n, i.e., $\sigma_i = 1$ if and only if $i \in \sigma$. A permutation π over N can also be represented by an array of n distinct integers, $\pi_1 \cdots \pi_n \in N$. An array $a = a_1 \cdots a_n$ is *lexicographically less than* $b = b_1 \cdots b_n$ if for some i: $a_i < b_i$ and for all $j < i$, $a_j \leq b_j$. A permutation (combination) π is lexicographically less than σ if the array that represents π is lexicographically less than the one that represents σ. Thus the permutations are ordered

$$\begin{aligned} 12\cdots n &<_L 12\cdots(n-2)n(n-1) \\ &<_L \cdots \\ &<_L n(n-1)\cdots 21 \ ; \end{aligned}$$

and the p-combinations are ordered

$$\begin{aligned} 00\cdots0\underbrace{11\cdots1}_{p} &<_L 00\cdots010\underbrace{11\cdots1}_{p-1} \\ &<_L \cdots \\ &<_L \underbrace{11\cdots1}_{p}00\cdots0 \ . \end{aligned}$$

We consider producing permutations of N and p-combinations in lexicographical order. Producing the next object might require $O(n)$ time: consider the consecutive permutations $1n(n-1)\cdots2$ and $2134\cdots n$, since these two permutations differ in all positions, given the first permutation $\Omega(n)$ time is required to produce the second one. We show algorithms whose amortized time complexity is $O(1)$, i.e., the time required to produce m consecutive objects is $O(n+m)$.

While previous algorithms required auxiliary data structures of size $\theta(n)$, our algorithms require at most $O(1)$ additional space. Thus our data structure is *implicit* [4]. An algorithm for generating combinatorial objects is *memoryless* if its input is a single object with no additional structure and its output is the next object according to some prespecified order. In our case, an algorithm is memoryless if its input consists of no data other than the array required to store the permutation or combination.

We present a memoryless algorithm to produce permutations. While, for combinations, our algorithm requires retaining in addition to the combination two integer variables. Moreover, these modest extra space requirements cannot be improved without sacrificing time—we show that every memoryless algorithm for producing p-combinations requires non-constant time.

1.1 Previous research

There is a considerable body of research in constructing combinatorial objects. Knuth and Szwarcfiter [3] produce all topological sortings, Squire [5] generated all acyclic orientations of an undirected graph, and Szwarcfiter and Chaty [6] generated all kernels of a directed graph with no directed cycles.

Walsh [7] presents two non-recursive algorithms to produce all well-formed parenthesis strings. The first generates each string in $O(n)$ worst-case time and requires space for only $O(1)$ extra integer variables, and the other generates each string in $O(1)$ worst-case time and uses $O(n)$ extra space. Thus he too shows a time/space tradeoff.

Ehrlich [2] and [1] generate permutations and combinations in $O(1)$ worst time complexity. They both use $O(n)$ time and the objects are not generated in lexicographic order.

2 Permutations

The lexicographically first permutation is

$$12\cdots n$$

and the last is

$$n\,n-1\cdots 1.$$

Let $\pi_1\cdots\pi_{i-1}*$ denote the set of all permutations of N whose prefix is $\pi_1\cdots\pi_{i-1}$. In the lexicographical order these permutations appear consecutively, i.e., if $\pi^1,\pi^2\in\pi_1\cdots\pi_{i-1}*$ and $\pi^1<_L\sigma<_L\pi^2$ then $\sigma\in\pi_1\cdots\pi_{i-1}*$.

The first permutation, π^1, of $\pi_1\cdots\pi_{i-1}*$ satisfies $\pi^1_i<\ldots<\pi^1_n$. The last one, π^2, satisfies $\pi^2_i>\ldots>\pi^2_n$. Let π^3 be the permutation immediately following π^2. The permutation π^3 cannot belong to $\pi_1\cdots\pi_{i-1}*$. If $\pi_{i-1}>\pi^2_i$ then π^2 is the last permutation of $\pi_1\cdots\pi_{i-2}*$. Hence $\pi^3\notin\pi_1\cdots\pi_{i-2}*$. If, however, $\pi_{i-1}<\pi_i$ then exchanging π_{i-1} and π_i yields a lexicographically larger permutation. Thus $\pi^3\in\pi_1\cdots\pi_{i-2}*$, and is the first permutation of $\pi_1\cdots\pi_{i-2}*$ which is lexicographically larger than π^2. Thus $\pi^3_{i-1}\in\{\pi^2_i\cdots\pi^2_n\}=\{\pi_i\cdots\pi_n\}$, i.e.,

$$\pi^3_{i-1}=\min\{\pi_j : j\ge i \text{ and } \pi_j>\pi_{i-1}\}. \qquad (1)$$

Since any permutation $\sigma\in\pi_1\cdots\pi_{i-2}\pi^3_{i-1}*$ is lexicographically greater than π^2, the permutation π^3 is the minimum permutation of $\pi_1\cdots\pi_{i-2}\pi^3_{i-1}*$, i.e., $\pi^3_i<\pi^3_{i+1}<\ldots<\pi^3_n$.

Given a permutation $\pi=\pi_1\cdots\pi_n$, to find the next permutation π^3, we first find the last decreasing subsequence $\pi_i>\pi_{i+1}>\ldots>\pi_n$ by scanning from position n; then we find π^3_{i-1} by equation (1); swap positions $i-1$ and j. Now $\pi^2_i>\ldots>\pi^2_{j-1}>\pi^2_{i-1}>\pi^2_{j+1}>\ldots>\pi^2_n$. The final sequence is obtained by reversing $\pi^2_i\cdots\pi^2_n$. See Program 1.

Finding the index i requires $\theta(n-i)$ time. Likewise for j. Reversing the sequence also requires $\theta(n-i)$ time. For $i=1$ or $j=1$ the amount of work is therefore $\theta(n)$. We next show that the amortized complexity is much lower.

Theorem 2.1 *Procedure* `next_perm()` *requires* $O(1)$ *amortized time.*

Proof: For $i=1,\ldots,n-1$ let

$$c_i=\begin{cases}1 & \text{if } \pi_i>\pi_{i+1}\\ 0 & \text{otherwise.}\end{cases}$$

And define a potential fuction

$$\Phi(\pi)=\sum_{i=1}^{n-1}c_i.$$

```
int n,v[n];
int next_perm()
{
  int i, j, k;
  for(i=n-1;i>0 && v[i-1]>v[i];i--)
    ;
  if(i == 0)
    return 0;
  for(j=i+1; j<n && v[i-1]<v[j]; j++)
    ;
  swap v[i-1] and  v[j-1];
  reverse v[i..n-1];
  return 1;
}
```

Program 1

The actual time is $t=n-i$ and since $\pi'_i<\pi'_{i+1}<\ldots<\pi'_n$, $\Delta\Phi=-(n-i-1)$. Thus the amortized time is

$$a=t+\Delta\Phi=O(1)\ .$$

□

Note that since two consecutive permutations may differ by up to n places (e.g. $n-1\,n\,n-2\,n-3\cdots1$ and $n\,1\,2\cdots n-1$) the worst case time is $\theta(n)$. Thus in order to achieve $O(1)$ time, we must settle for amortized complexity.

3 Combinations

3.1 Upper bound

We will use the following notation: 1^k will denote a run (a consecutive block) of k "1"s and 0^k a run of k "0"s. In addition to the boolean array of size n that represents the p-combination, we keep two counters q and r. The counter $0\le r\le p$ counts the number of "1"s that follow the last "0", and the counter $1\le q\le n-p$ counts the number of consecutive "0"s before the last run of "1"s, i.e., $\pi=\alpha0^q1^r$.

Let π be the current combination: There are two cases to consider:
Case 1: $r>0$, i.e., π ends with a "1".
Exchange the first "1" of the last run of "1"s with the last "0" of the last run of "0"s.

$$\alpha0^q1^r=\alpha0^{q-1}\mathbf{0}1^{r-1}\ \Rightarrow\alpha0^{q-1}\mathbf{1}0^{r-1}=\alpha'0^{q'}1^{r'}\ ,$$

where $\alpha'=\alpha0^{q-1}1$, $q'=1$ and $r'=r-1$.
Case 2: $r=0$, i.e., π ends with a "0".

```
int n, v[n], p, r, q;
    /* v contains exactly p ``1"s,
          v = ...10^r 1^q        */
int next_comb()
    {
    int s,last1, f;

    if(r > 0){
        v[n-r] = 0;
        v[n-r-1] = 1;
        r--;
        q = 1;
        return 1;
    }

    if(q == n-p)
        return 0;

    last1 = n-q-1;
    for(f = last1; v[f]; f--)
        ;
    v[f++] = 1;
    v[f] = 0;
    s = last1 - f + 1;
    swap v[f+1..f+1+min(s-1,q)] and
        v[n-min(s-1,q)..n-1]
    q++;
    r = s-1;
    return 1;
}
```

Program 2

If $q = n - p$, then $\pi = 1^p 0^{n-p}$ is the last combination. Otherwise, the next combination is produced by exchanging the first "1" of the last run with the "0" on its left and moving all the remaining "1"s of the last run all the way right, i.e.,

$$\pi = \alpha 01^s 0^q \;\Rightarrow\; \alpha 10^{q+1}1^{s-1} = \alpha' 0^{q'}1^{r'} = \pi' \;,$$

where $\alpha' = \alpha 1$, $q' = q + 1$ and $r' = s - 1$.
See Program 2.

Theorem 3.1 *Procedure* `next_comb()` *requires* $O(1)$ *amortized time.*

Proof: Consider the potential function $\Phi = p - r$. Let r_i denote the value of r after the i-th operation.
Case 1 ($r > 0$): $\Phi = p - r$, $\Phi' = p - r' = p - (r - 1)$. Then $\Delta\Phi = \Phi' - \Phi = 1$. Since in this case the procedure next_comb only exchanges two positions, the actual time is $t = 1$. Hence, the amortized time is $a = t + \Delta\Phi = O(1)$.
Case 2 ($r = 0$): $\Phi = p$, $\Phi' = p - (s - 1) = p - s + 1$. $\Delta\Phi = \Phi' - \Phi = (p - s + 1) - p = -s + 1$. In this case, The actual time $t = 1 + \min\{s, q\} \le 1 + s$. Thus, the amortized time is $a = t + \Delta\Phi = O(1)$.

□

3.2 Lower bounds

First note that the amortized time bound cannot be replaced by a "worst case" bound since the number of bits that have to be changed between two consecutive combinations is $1 + \min\{q, r\}$. In particular, the combination $01^p 0^{n-p-1}$ is followed by $10^{n-p}1^{p-1}$ which involves $1 + \min\{p, n - p\}$ changes. This is maximized when $p = n/2$.

Consider a memoryless scheme that operates on $\pi = \alpha 0^q 1^r$. If the sequence ends with a "1" ($r > 0$), we must find the first "0" before the the last run of "1"s, and since we have no extra data we must scan π from its right-hand end until finding a "0", i.e., scan r "1"s. If the sequence ends with a "0" ($r = 0$) then we have to find the first "1" before the last run of "0"s, i.e., the time is q. Since there are p "1"s the probability that the last digit is "1" is p/n, and the probability that the sequence ends with a "0" is $1 - p/n$. The average time is therefore:

$$\begin{aligned} A_n(p) &= P(r > 0)E(r|r > 0) + P(r = 0)E(q|r = 0) \\ &= \frac{p}{n}E(r|r > 0) + (1 - \frac{p}{n})E(q|r = 0) \;. \end{aligned}$$

When $r > 0$ the last digit is "1", and we have $p - 1$ other "1"s which are partitioned by the $n - p$ "0"s to $n - p + 1$ runs (some of which might have zero length). The average length of such a run is therefore $\frac{p-1}{n-p+1}$. Since the last run of "1"s ends with an additional "1", its average length is $1 + \frac{p-1}{n-p+1} = \frac{n}{n-p+1}$.

When $r = 0$ the last digit is "0", and the remaining $n - p - 1$ "0"s are partitioned by the p "1"s to $p+1$ runs of "0"s. Their average length is $\frac{n-p-1}{p+1}$. The average length of the last run of "0"s is $1 + \frac{n-p-1}{p+1} = \frac{n}{p+1}$.

$$\begin{aligned} A_n(p) &= \frac{p}{n}\frac{n}{n-p+1} + \left(1 - \frac{p}{n}\right)\frac{n}{p+1} \\ &= \frac{p}{n-p+1} + \frac{n-p}{p+1} \;. \end{aligned}$$

The asymptotic value of $A_n(p)$ depends on p. The first term monotonically increases while the second decreases. For $p = cn$, $A_n(p) = O(1)$, for $p = \sqrt{n}$ or $p = n - \sqrt{n}$, $A_n(p) = \theta(\sqrt{n})$ and for $p = 1$ or

$p = n - 1$, $A_n(p) = \theta(n)$. In general, for a function $1 \leq f(n) < n$, and $p = f(n)$ or $p = n - f(n)$, we have that $A_n(p) = \theta(n/f(n))$.

Thus for many values of p the average time required to produce the next combination is not constant but an increasing function on n. Hence, for memoryless algorithms the amortized complexity is also non-constant.

References

[1] Nachum Dershowitz. A simplified loop-free algorithm for generating permutations. *BIT*, 15(2):158–164, 1975.

[2] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM*, 20(3):500–513, 1973.

[3] D. E. Knuth and J. L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 2:153–157, 1974.

[4] J. L. Munro. An implicit data structure for the dictionary problem that runs in polylog time. In *FOCS*, volume 25, pages 369–374, 1984.

[5] Matthew B. Squire. Generating the acyclic orientations of a graph. *Journal of Algorithms*, 26(2):275–290, 1998.

[6] J. L. Szwarcfiter and G. Chaty. Enumerating the kernels of a directed graph with no odd circuits. *Information Processing Letters*, 51:149–153, 1994.

[7] Timothy R. Walsh. Generation of well-formed parenthesis strings in constant worst-case time. *Journal of Algorithms*, pages 165–17, 1998.