Efficient Generation of Subsets with a Given Sum

Dominique Roelants van Baronaigien *

Frank Ruskey †

Department of Computer Science
University of Victoria
Victoria, B.C., V8W 3P6, Canada

Abstract

Let $\mathbf{C}(n,p)$ denote the set of all subsets of $\{1,2,\ldots,n\}$ whose sum is p, and let $\mathbf{C}(n,k,p)$ denote the k element sets of $\mathbf{C}(n,p)$. We show that the elements of $\mathbf{C}(n,p)$ and $\mathbf{C}(n,k,p)$ can be generated efficiently by a simple recursive algorithms. The subsets are represented by characteristic bitstrings and by lists of elements. These representations can be generated in time that is proportional to the number of subsets generated.

1 Introduction

A subject dear to the heart of every computational combinatorist is that of generating combinatorial objects. Subsets of an n-set and k-subsets of an n-set, or combinations, are of fundamental importance and much has been written about generating them (see, for example, [2], [3], [4], [8], [11]). A natural restriction of the problem of generating all subsets is the problem of generating all subsets of $\{1,2,\ldots,n\}$ whose sum is, say, p.

The study of algorithms for generating combinatorial objects often leads to a deeper understanding of the objects themselves. Thus our motivations are twofold. First, we add to the catalog of elementary combinatorial objects that can be efficiently

*Research supported by NSERC grant 89767.

†Research supported by NSERC grant A3379.

generated. Secondly, we gain a little more insight into the combinatorics of subsets with a given sum.

Of course, we want our algorithms to be efficient. In the context of combinatorial generation, our goal is to come up with algorithms that run in *constant amortized time*. This means that the total amount of computation is proportional to the number of objects generated. Typically there are two parameters of interest: n, the size of the representation of an individual object and N, the total number of objects. An algorithm has constant amortized time complexity if its running time is $O(n + N)$; the term n is necessary to account for initialization. We also want the algorithm to use $O(n)$ space.

The algorithms that we develop are recursive, have no loops, and have no dead-ends. This means that the total amount of computation used in generating the objects is proportional to the number of recursive calls that are made.

There is a simple principle, which we call the CAT (for Constant Amortized Time) principle, that can be used to show that certain of these recursive generation algorithms run in constant amortized time. The *degree* of a call of a recursive algorithm is the number of "immediate" recursive calls initiated by the current call. For instance, the degree of a call of mergesort is either 2 or 0 and the degree of a call of binary search is either 1 or 0.

The CAT principle is given below. A precursor may be found in [9] as Theorem 2.1. The principle may be proven by considering the underlying computation tree.

> A recursive generation algorithm with the following three properties runs in constant amortized time.
>
> 1. Every call results in the output of at least one object.
> 2. Excluding the computation done by recursive calls, the amount of computation of any call is proportional to the degree of a call.
> 3. The number of calls of degree one is $O(N)$.

Let $\mathbf{C}(n, p)$ denote the set of all subsets of $\{1, 2, \ldots, n\}$ whose sum is p, and let $\mathbf{C}(n, k, p)$ denote the set of all k element subsets of $\{1, 2, \ldots, n\}$ whose sum is p. For example, $\mathbf{C}(5, 6)$ contains the three sets $\{1, 5\}$, $\{2, 4\}$, and $\{1, 2, 3\}$. An alternative perspective is that $\mathbf{C}(n, k, p)$ is the number of partitions of p into k distinct parts each of which is less than n.

There are two common ways to represent subsets. One representation is simply to list the elements of the subset in increasing order. Another representation is the characteristic bitstring in which the ith bit is 0 or 1 depending on whether i is in the subset or not. The subsets of our example of the last paragraph would be represented as $(1, 5)$, $(2, 4)$, and $(1, 2, 3)$, or 10001, 01010, and 11100.

As is frequently the case, the particular representation that is used will influence the complexity of the algorithms. In Section 2 we develop algorithms for generating the elements of $\mathbf{C}(n,p)$ and $\mathbf{C}(n,k,p)$, as represented by bitstrings, in time that appears experimentally to be constant amortized time. In Section 3 we develop an algorithm for generating the elements of $\mathbf{C}(n,p)$, as represented by lists of elements, in constant amortized time.

We introduce the notation s_n for the sum of the first n positive integers.

$$s_n \;=\; 1+2+\cdots+n \;=\; n(n+1)/2$$

Clearly, $\mathbf{C}(n,p)$ is non-empty if and only if $0 \le p \le s_n$, and $\mathbf{C}(n,k,p)$ is non-empty if and only if $s_k \le p \le s_n - s_{n-k}$.

There is no closed form formula for $C(n,p)$ or $C(n,k,p)$, as far as we are aware, but there is a simple generating function whose coefficients give these numbers. Let $f(x,y)$ be the ordinary generating function defined below.

$$f(x,y) \;=\; \prod_{j=1}^{n}(1+xy^j)$$

Then $C(n,k,p) = \langle x^k y^p \rangle f(x,y)$ and $C(n,p) = \langle y^p \rangle f(1,y)$. Furthermore, there are some simple recurrence relations that these numbers satisfy. These recurrence relations are not stated in their simplest form, but rather are presented in a manner consistent with the recursive programs to be presented in the next section.

$$C(n,p) = \begin{cases} 1 & \text{if } n=p=1 \\ C(n-1,p) & \text{if } p<n \\ C(n-1,p-n) & \text{if } p>s_{n-1} \\ C(n-1,p)+C(n-1,p-n) & \text{if } n \le p \le s_{n-1} \end{cases} \tag{1}$$

Note that $C(n+i,n) = C(n,n)$ for $i \ge 0$ and that $C(n,n)$ is the number of partitions of n into distinct parts (which is the same as the number of partitions of n into odd parts). We now give a recurrence relation for $C(n,k,p)$.

$$C(n,k,p) = \begin{cases} 1 & \text{if } n=p=1 \\ C(n-1,k,p) & \text{if } p<n+s_{k-1} \\ C(n-1,k-1,p-n) & \text{if } p>s_{n-1}-s_{n-k-1} \\ C(n-1,k,p)+C(n-1,k-1,p-n) & \text{otherwise} \end{cases} \tag{2}$$

Note that $C(n,k,p) = C(p-s_{k-1},k,p)$ if $p < n+s_{k-1}$. The numbers $C(n,k,p)$ have three symmetries which are expressed in equations (3), (4), and (5). These equations will prove useful in the next section. The first (3) follows by taking complements, and the second (4) follows by the transformation $x \leftarrow n-x+1$ on each element x of a subset. The third (5) combines both bijections.

$$C(n,k,p) \;=\; C(n,n-k,s_n-p) \tag{3}$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1																
1	1	1															
2	1	1	1	1													
3	1	1	1	2	1	1	1										
4	1	1	1	2	2	2	2	2	1	1	1						
5	1	1	1	2	2	3	3	3	3	3	3	2	2	1	1	1	
6	1	1	1	2	2	3	4	4	4	5	5	5	5	4	4	4	3
7	1	1	1	2	2	3	4	5	5	6	7	7	8	8	8	8	8
8	1	1	1	2	2	3	4	5	6	7	8	9	10	11	12	13	13
9	1	1	1	2	2	3	4	5	6	8	9	10	12	13	15	17	18
10	1	1	1	2	2	3	4	5	6	8	10	11	13	15	17	20	22

Table 1: The numbers $C(n, p)$.

$$C(n,k,p) = C(n,k,(n+1)k-p) \tag{4}$$

$$C(n,k,p) = C(n,n-k,s_n-(n+1)k+p) \tag{5}$$

2 Fast Bitstring Algorithms

Based on the recurrence relations of the previous sections we are led to a simple algorithms for generating the elements of $\mathbf{C}(n,p)$. The algorithms represent the subsets as characteristic bitstrings $b_1 b_2 \cdots b_n$, and generate them in colex order. The algorithms appear to run in constant amortized time, independent of n, k, and p, but we are as yet unable to prove this.

2.1 Algorithm for $C(n,p)$

Our first algorithm is based directly on the recurrence relation (1) and is to be found in Figure 1. The running time of this algorithm is clearly proportional to the number of recursive calls it makes. Let $D(n,p)$ denote the number of calls to `C1` initiated by the call `C1(n,p)`. Then the amortized computation used by `C1(n,p)` is proportional to the ratio $D(n,p)/C(n,p)$. Unfortunately, this ratio is not bounded by any constant, independent of n and p. In particular, if p is held fixed and n increased then the ratio appears to be unbounded. On the other hand, if $p \geq \sqrt{n}$, then the ratio appears to be bounded.

$$D(n,p) = \begin{cases} 1 & \text{if } n = p = 1 \\ 1 + D(n-1,p) & \text{if } p < n \\ 1 + D(n-1,p-n) & \text{if } p > s_{n-1} \\ 1 + D(n-1,p) + D(n-1,p-n) & \text{if } n \leq p \leq s_{n-1} \end{cases} \tag{6}$$

```
procedure C1 ( n,p : integer );
begin
  if n = 0 then Output( b )
  else begin
    if s[n-1] >= p then begin
       b[n] := 0;  C1( n-1, p );
    end;
    if n <= p then begin
       b[n] := 1;  C1( n-1, p-n );
    end;
  end;
end {of C1};
```

Figure 1: The procedure `C1(n,p)`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1																
1	2	2															
2	3	3	3	3													
3	4	4	4	7	4	4	4										
4	5	5	5	8	9	9	9	8	5	5	5						
5	6	6	6	9	10	15	15	14	14	15	15	10	9	6	6	6	
6	7	7	7	10	11	16	22	21	21	25	26	26	25	21	21	22	16
7	8	8	8	11	12	17	23	29	29	33	37	38	42	44	43	44	42
8	9	9	9	12	13	18	24	30	38	42	46	50	55	62	67	74	72
9	10	10	10	13	14	19	25	31	39	52	56	60	68	76	86	99	103
10	11	11	11	14	15	20	26	32	40	53	67	71	79	90	101	119	129

Table 2: The original numbers $D(n,p)$.

```
procedure C2 ( n,p : integer );
begin
  if p = 0 then Output( b )
  else begin
    if  p < n  then begin
       b[n] := 0;  C2( p, p );
    end else
    if  p > s[n-1]  then begin
       b[n] := 1;  C2( n-1, p-n );
    end
    else begin
       b[n] := 0;  C2( n-1, p   );
       b[n] := 1;  C2( n-1, p-n );
    end;
    b[n] := 0;
  end;
end {of C2};
```

Figure 2: Improved version of `C1(n,p)`.

Procedure `C1` satisfies conditions 1 and 2 of the CAT principle but, by the previous discussion, does not appear to satisfy condition 3, at least for small values of p.

If possible, we would like to eliminate calls of degree 1. Many calls of degree 1 are eliminated in procedure `C2(n,p)`. These eliminated calls skip over blocks of 0's, so we must ensure that 0's are indeed in the positions that are skipped over. This is accomplished by initializing `b` to contain all 0's and by the statement `b[n] := 0` at the end of `C2`. The initial call is `C2(n,p)` if $p \le s_n/2$. If $p > s_n/2$, then the roles of 0 and 1 are interchanged and `C2(n,s[n]-p)` is called.

With this modification, the second case of the recursion for $D(n,p)$ becomes $1 + D(p,p)$ instead of $1 + D(n-1,p)$. To avoid confusion let us denote the resulting numbers $E(n,p)$. We have observed experimentally that the ratio $E(n,p)/C(n,p)$ is less than 4.5 if $p \le s_n/2$, but we have no proof. The maximum value of this ratio appears to occur at $p = s_n/2$ if $n \ge 10$.

Conjecture 1: For all $0 \le p \le s_n/2$,

$$\frac{E(n,p)}{C(n,p)} \le 4.5.$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1																
1	1	2															
2	1	3	2	3													
3	1	3	3	5	4	3	4										
4	1	3	3	6	6	7	8	6	5	4	5						
5	1	3	3	6	7	9	12	10	12	11	13	9	7	6	5	6	
6	1	3	3	6	7	10	14	14	16	18	21	19	20	17	18	18	14
7	1	3	3	6	7	10	15	16	20	22	28	27	31	32	33	35	33
8	1	3	3	6	7	10	15	17	22	26	32	34	39	43	49	52	54
9	1	3	3	6	7	10	15	17	23	28	36	38	46	51	60	68	72
10	1	3	3	6	7	10	15	17	23	29	38	42	50	58	68	79	88

Table 3: The improved numbers $E(n,p)$.

2.2 Algorithm for $\mathbf{C}(n,k,p)$

In this section we extend the results of the previous section to $\mathbf{C}(n,k,p)$. We immediately adopt the idea that was used to improve the algorithm and the result is quite similar to `C2(n,p)`. See Figure 3 for `C3(n,k,p)`.

The following table indicates which version of `C3(n,k,p)` should be called for maximum efficiency.

	$k \leq n/2$	$k \geq n/2$
$p \leq (s_n + s_k - s_{n-k})/2$	$C(n,k,(n+1)k-p)$	$C(n,n-k,s_n-p)$
$p \geq (s_n + s_k - s_{n-k})/2$	$C(n,k,p)$	$C(n,n-k,s_n-(n+1)k+p)$

A recurrence relation for $D(n,k,p)$ is given below.

$$D(n,k,p) = \begin{cases} 1 & \text{if } n = p = 1 \\ 1 + D(p - s_{k-1}, k, p) & \text{if } p < n + s_{k-1} \\ 1 + D(n-1, k-1, p-n) & \text{if } p > s_{n-1} - s_{n-k-1} \\ 1 + D(n-1,k,p) + D(n-1,k-1,p-n) & \text{otherwise} \end{cases}$$

Experimentally, we have observed that the worst case occurs when k and p are in the middle of their ranges; that is, when we are generating $\mathbf{C}(n, n/2, s_n/2)$ (again, n must be large enough). It appears in this case that the ratio is bounded by 6, but we have no proof that it is bounded by any constant.

Conjecture 2: For all $(s_n+s_k-s_{n-k})/2 \leq p \leq s_n+s_k$ and $k \leq n/2$, if $C(n,k,p) > 1$, then

$$\frac{D(n,k,p)}{C(n,k,p)} \leq 6.$$

```
procedure C3 ( n,k,p : integer );
begin
  if k = 0 then Output( b )
  else begin
    if  p < n + s[k-1]  then begin
       b[n] := 0;  C3( p-s[k-1], k, p );
    end else
    if  p > s[n-1] - s[n-k-1]  then begin
       b[n] := 1;  C3( n-1, k-1, p-n );
    end
    else begin
       b[n] := 0;  C3( n-1, k,   p   );
       b[n] := 1;  C3( n-1, k-1, p-n );
    end;
    b[n] := 0;
  end;
end {of C3};
```

Figure 3: The procedure `C3(n,k,p)`.

3 A Provably Fast Subset Algorithm

In this section we present an algorithm for generating the elements of $\mathbf{C}(n,p)$ in constant amortized time, where the subsets are represented by a list of the elements in the subset.

The algorithms presented in the previous section made use of the fact that $\mathbf{C}(n,p) = \mathbf{C}(p,p)$ if $p < n$. This fact allowed the algorithm to *skip* the redundant recursive calls where a series of possible subset elements, specifically the elements $p+1, p+2, ..., n$, must be excluded. This technique was used previously by Roelants van Baronaigien and Ruskey [10] and Ko and Ruskey [6]. The algorithm we present here is slightly more complicated because we must skip both redundant recursive calls where elements *must* be excluded and where elements *must* be included. We refer to the elements that must be included or must be excluded as *forced elements*. To be more precise, j must be included if $s_{j-1} < p$ and must be excluded if $j > p$.

The trick to avoiding the processing of the forced elements is to consider *complement* $\overline{\mathbf{C}}(n,p) = \mathbf{C}(n, s_n - p)$ of $\mathbf{C}(n,p)$. If k must be included in $\mathbf{C}(n,p)$ then it must be excluded in $\overline{\mathbf{C}}(n,p)$, and vice-versa. Thus, the problem of determining if there is a sequence of elements that must be included in $\mathbf{C}(n,p)$ is equivalent to the problem of determining if there is a sequence of elements that must be excluded from $\overline{\mathbf{C}}(n,p)$, and we have already seen how to do that in Section 2.

Of course, if we are listing $\overline{\mathbf{C}}(n,p)$ then we must include elements that we are

```
procedure C4( n,p : integer; comp : boolean );
begin
  if n = 0 then Output( b )
  else begin
     if p > s[n]/2 then begin
        p := s[n] - p; comp := not comp;
     end;
     if p < n then
        if comp then begin
           list[n] := list[0];  list[0] := p+1;
           C( p, p, comp );
           list[0] := list[n];  list[n] := n+1;
        end else
           C( p, p, comp )
     else begin
        list[n] := list[0];  list[0] := n;
        if comp then begin
           C( n-1, p, comp );
           list[0] := list[n];  list[n] := n+1;
           C( n-1, p-n,   comp );
        end else begin
           C( n-1, p-n, comp );
           list[0] := list[n];  list[n] := n+1;
           C( n-1, p,   comp );
        end;
     end;
  end;
end {of C4};
```

Figure 4: The procedure C4(n,p).

skipping over and if we are listing $\mathbf{C}(n, p)$ we do not include the elements that we skip over. It is therefore necessary to pass a third parameter, *comp*, to the procedure. If *comp* is true then we must include the forced elements in the subset and if *comp* is false we must exclude the forced elements from the subset.

Before discussing how to deal with the forced elements, we briefly discuss the data structure used to represent the subset. In procedure C4, the array *list* is used to represent the list of elements in the subset. Consider the set $\mathbf{a} = \{a_1 < a_2 < \ldots < a_j\}$. We represent that set by setting $list[0] = a_1$, $list[a_i] = a_{i+1}$ for $i = 1, 2, \ldots j-1$, $list[a_j] = n+1$ and for all $i \notin \mathbf{a}$, $list[i] = i+1$. As an example, if $list = 2, 2, 3, 5, 5, 8, 7, 8, 9$ and $n = 8$ then the subset that *list* represents is $\{2, 3, 5, 8\}$. Each element $list[i]$ is initialized to $i+1$ except $list[0]$ which is initialized to $n+1$.

We now consider an efficient method of skipping over forced elements. If $list[i] = i+f$, then $i+1, i+2, \ldots, i+f-1$ are excluded; furthermore, if i is included then $i+f$ is included ($list[0]$ is always included). Clearly a sequence of forced elements can be either included or excluded in constant time regardless of the length of the sequence.

The algorithm is presented in Figure 4. We have already discussed variables *list*, and *comp*. The global array s, and the parameters n and p are the same as defined in section 2.

The description of procedure C4 and a simple inductive proof can be used to show that procedure C4 lists all subsets of an n-set that sum to p. What remains to be discussed is the time complexity of procedure C4.

To analyse the algorithm, we use the CAT principle. It is clear that every call of the procedure will result in some output, and that the amount of computation done in each call of the procedure, excluding recursive calls, is constant. If there is a call of degree one (i.e., `p < n` and so the procedure directly calls itself only once), then either the next call produces output, or the next call is of degree 2. Thus by the CAT principle, the generation algorithm, procedure C4, runs in constant amortized time.

4 Summary and Further Research

We have given a provably constant amortized time algorithm for generating all subsets of an n-set with a given sum. This algorithm uses a data structure similar to the stack simulation data structure used by Reingold, Neivergelt and Deo [2] to implement a Gray Code generation algorithm.

We also presented a simpler algorithm that experimentally appears to run in constant amortized time. An algorithm for listing all subsets of an n-set of a specific size that have a given sum is presented as well.

The problem of listing these objects using a loopless algorithm [5] remains open as does the resolution of Conjectures 1 and 2 of Section 2.

5 Acknowledgement

We wish to thank Brendan McKay for suggesting the problem.

References

[1] S. Akl, *A Comparison of Combination Generation Methods*, ACM Transactions on Mathematical Software, 7 (1981) 42-45.

[2] J. Bitner, G. Ehrlich, and E. Reingold, *Efficient Generation of the Binary Reflected Gray Code and its Applications*, Communications of the ACM, Vol. 19, No. 9, Sept. 1976, pp 517-521.

[3] P. Eades and M. Carkeet, *Performance of Subset Generating Algorithms*, Annals of Discrete Math., 26 (1985) 49-58.

[4] P. Eades and B. McKay, *An Algorithm for Generating Subsets of Fixed Size with a Strong Minimal Change Property*, Information Processing Letters, 19 (1984) 131-133.

[5] G. Ehrlich, *Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations*, Journal of the ACM, 20 (1973) 500-513.

[6] C.W. Ko and F. Ruskey, *Generating Permutations of a Bag by Interchanges*, Information Processing Letters, 41 (1992) 263-269.

[7] C.W.H. Lam and L.H. Soicher, *Three New Combination Algorithms with the Minimal Change Property*, Communications of the ACM, 25 (1982) 555-559.

[8] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, 1977.

[9] F. Ruskey and D. Roelants van Baronaigien, *Fast Recursive Algorithms For Generating Combinatorial Objects*, Congressus Numerantium, 41, 1984, pp 53-62.

[10] D. Roelants van Baronaigien and F. Ruskey, *Generating Permutations With Given Ups and Downs*, Discrete Applied Math., 36 (1992) 57-65.

[11] H.S. Wilf, *Combinatorial Algorithms: An Update*, SIAM, CBMS 55, 1989.