

Simple Constant Amortized Time Generation of Fixed Length Numeric Partitions

John M. Boyer
jboyer@acm.org

Version: August 4, 2003

A numeric partition of n of fixed length k is an unordered sequence of k positive integers that sum to n . A recent implementation of Savage's constant amortized time (CAT) algorithm for generating numeric partitions in a minimal change order required over twenty-five pages of C code. According to Ruskey, there is no simple lexicographic algorithm that generates fixed length numeric partitions in the natural representation in constant amortized time (CAT) per partition. This paper describes such an algorithm and proves it is a CAT algorithm. As a byproduct, CAT generators are given for several other classes of numerical partitions, some of which have no prior CAT algorithm.

1. INTRODUCTION

A *numeric partition* of n is a representation of n as a sum of positive integers. The summands are called *parts*, and their order is ignored [4, p. 67]. The notation $p(n, k)$ denotes either the count or the set of numeric partitions of n with fixed length k (in this paper, $p(n, k)$ denotes the count unless otherwise indicated).

The *natural representation* of a numeric partition contains the actual parts. For example, a example 6-part partition of 12 is the sequence $\langle 4, 2, 2, 2, 1, 1 \rangle$. An alternative form known as the *multiplicity representation* lists each unique value in the partition along with the number of times the part occurs. For example, the above partition would be represented as $4(1), 2(3), 1(2)$. A third common form, known as a *Ferrers-Young diagram*, is essentially a base-1 version of the natural representation. If the parts are considered from greatest to least in value, a Ferrers-Young diagram for a partition is a boolean matrix of size k by p_1 where, in each row r , the leftmost p_r entries are true and the remaining entries are false. The Ferrers-Young diagram is useful in establishing a correspondence between k -length numeric partitions of n and numeric partitions of n with largest part k . Given a k length numeric partition π , the *conjugate* of π is a partition with largest part k that can be computed by performing a matrix transpose of the Ferrers-Young diagram of π .

There exist simple constant amortized time (CAT) algorithms for lexicographic generation of all numeric partitions of a number n and of all numeric partitions of n with largest part k [7]. The earliest algorithm for generating fixed length numeric partitions appears in [3], but it does not achieve CAT performance. More typically, fixed length numeric partitions in the natural representation are generated by converting from the multiplicity form [6] or by Ferrers-Young conjugation of partitions with largest part k [4]. However, in both cases, the conversion to the natural representation imposes a non-constant cost per fixed length partition that is generated.

Savage [8] created a CAT algorithm for generating fixed length numeric partitions in a minimal change order. It generates partitions with largest part k and uses reasoning about the Ferrers-Young conjugate to produce fixed length partitions. Unlike its lexicographic counterpart, the successive objects have the minimal change property, so the change necessary to produce the successive Ferrers-Young conjugates is also bounded above by a constant.

Generating objects in a minimal change order is typically more difficult than simple lexicographic generation, and Savage's algorithm is no exception. A recent implementation [1] required over twenty-five pages of C code. Due to this complexity, a simpler approach is desirable when a minimal change order is not required. According to [7], there exists no simple lexicographic generator for fixed length numeric partitions in the natural representation that has been *proven* to achieve constant amortized time— a curious gap in knowledge given that CAT lexicographic generators are typically easier and therefore created before their minimal change counterparts.

This paper describes a very simple algorithm for generating fixed length numeric partitions in the natural representation and proves its correctness in Section 2. Section 3 provides an interesting proof that the algorithm generates fixed length numeric partitions in constant amortized time per partition. Finally, Section 4 concludes with a discussion of similar objects that can be generated in constant amortized time with only minor modifications to the algorithm. Some results of this research were presented at the Tenth SIAM Conference on Discrete Mathematics [2].

2. CAT GENERATION OF THE SET $P(N, K)$

In algorithms for generating partitions with largest part k , which includes Savage's algorithm [8], the parts of a partition are typically given in non-increasing order. However, by generating the parts from smallest to largest, the algorithm in Figure 1 indirectly exploits the following well-known recurrence for fixed length numeric partitions:

Procedure: GenP(A, n, k, k_{orig}, l)
Input: Generation array A
integers n, k, k_{orig}
integer l giving least part value

```

if  $k$  is 1
   $A[k] \leftarrow n$ 
  ProcessSolution( $A, k_{orig}$ )
else
   $h \leftarrow \lfloor n/k \rfloor$ 

  for integers  $i$  from  $l$  to  $h$ 
     $A[k] \leftarrow i$ 
    GenP( $A, n - A[k], k - 1, k_{orig}, A[k]$ )

```

FIG. 1 Generator for the set $p(n, k)$ of k -length numeric partitions of n (though division is used to simplify the presentation, relatively little effort is required to eliminate it).

$$p(n, k) = p(n - 1, k - 1) + p(n - k, k) \quad (1)$$

The term $p(n - 1, k - 1)$ counts the number of partitions with a first part of one (the smallest part is first). The remaining partitions in which the first part is greater than one are counted by $p(n - k, k)$. The proof of this can be seen by adding one to each part of each partition in the set $p(n - k, k)$. The result is a set of k -part partitions of n with the least part being greater than one.

In the algorithm of Figure 1, the k parts of each partition are generated in positions 1 to k of the integer array A . The variable k_{orig} contains the size of A , which is necessary since the parameters n and k are reduced in the lower levels of recursion to reflect the work already done in generating a partition.

The algorithm *GenP* generates the partitions in co-lexicographic order (for any partition with a successor in the list, the partition is lexicographically less than the successor if the parts are considered from greatest to least index). For example, in generating $p(12, 6)$, the partition $\langle 7, 1, 1, 1, 1, 1 \rangle$ is generated first and $\langle 2, 2, 2, 2, 2, 2 \rangle$ is generated last. Co-lex generators result in more readable pseudocode, and a lexicographic listing can easily be created by replacing all occurrences of $A[k]$ with $A[k_{orig} - k + 1]$.

When k is greater than one, the least part value l and the greatest part value h are calculated. Then, a loop iterates the integers from l to h , assigning each integer to the part k then making a recursive call

to generate all parts in positions $k - 1$ down to 1 that, together with the parts in positions k to k_{orig} , form a partition expressed as a non-decreasing sequence (considered from positions k down to 1 due to co-lexicographic generation). Recursive termination occurs when k drops to 1.

The correctness of the algorithm rests almost entirely on the proper selection of l and h , combined with an inductive argument that becomes straightforward once l is considered along with the other parameters, as shown in Theorem 1.

THEOREM 1. *The algorithm invocation $GenP(A, n, k, k, 1)$ correctly generates the set $p(n, k)$.*

Proof. For all n , observe that the algorithm behaves correctly at $k = 1$ (regardless of the value assigned to l , which is ignored) since a single part that must sum to n must in fact be n . In an induction on n , this is useful both during the inductive step and in establishing the inductive basis at $n = 1$ (since k must be one).

For some constant c such that $1 \leq n < c$, assume the algorithm correctly generates all k -part partitions of n with least part greater than or equal to l , where $1 \leq l \leq \lfloor n/k \rfloor$. Now we consider the case $n = c$. If $k = 1$, then the algorithm behavior is correct by the aforementioned observation. If $k > 1$, then the algorithm loop selects every integer value i in the range l to $h = \lfloor n/k \rfloor$ for the least part, then recurses to generate $(k - 1)$ -part partitions of $n - i$ with least part greater or equal to i . By the inductive hypothesis, it does this correctly for each value of i .

Since the parts must be generated in order from least to greatest, it is correct to restrict the parts generated by the lower levels of recursion to be i or greater. Moreover, since the k parts must sum to n , the least part cannot exceed $\lfloor n/k \rfloor$. Thus, the loop iterates every possible value for the least part, and by induction every possible k -part partition having that least value is generated in the recursion.

Therefore, the algorithm correctly generates all k -part partitions of n since this is a special case of the induction above with $l = 1$. ■

3. ANALYSIS

In the algorithm of Figure 1, the function body runs in constant time because the cost of each loop iteration is associated with the recursive call in the loop body. The algorithm running time is analyzed by comparing the number of recursive function calls in the algorithm's computation tree to the number of numeric partitions generated. As an example, Figure 2 shows the computation tree for $p(12, 6)$.

In many lexicographic algorithm analyses, it is possible to prove CAT behavior by associating single-child vertices in the computation tree with descendant vertices that have a higher branching factor. As Figure 2 shows, this is difficult or impossible to do for the algorithm of the previous section

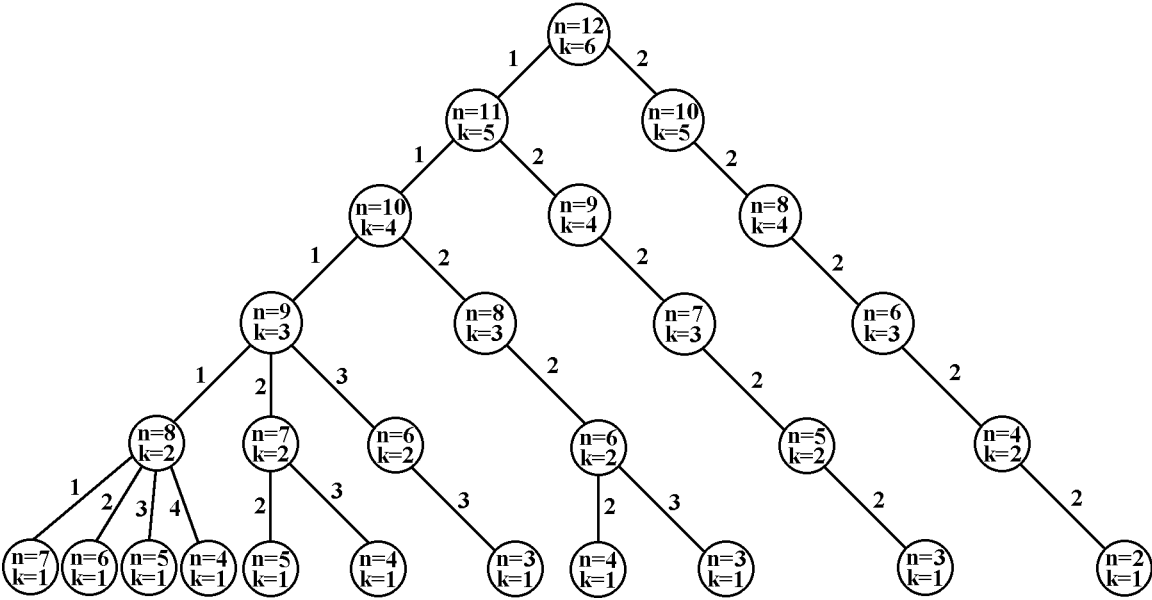


FIG. 2 The computation tree for $p(12, 6)$. Each vertex represents a function call and shows the values of n and k for that call. The edge labels leading from a non-leaf vertex to its children in the computation tree show the partition value placed into the k^{th} position by the loop before the call to the child function is made.

because many single-child vertices have no descendants of higher degree. It is also clear from Figure 2 that the long single-child paths are composed of different numbers during the generation of $p(n, k)$. For example, $p(12, 6)$ contains single-child paths of 2's, but the subtree that computes $p(9, 3)$ contains a single-child path of 3's. Consequently, the techniques often used for other lexicographic algorithms, such as padding the generation array with a useful value, will not work on this algorithm. Instead, a more involved mathematical argument must be developed to show that computation subtrees with low branching factors are balanced by subtrees with high branching factors.

According to Equation 1 above, an induction on both n and k is required. As mentioned above, the amount of work $w(n, k)$ performed by the algorithm can be measured in function call units since the cost of each function invocation is constant. Lemma 1 helps to set the inductive basis for k by showing that, regardless of n , if k is fixed, then the number of function calls to *GenP* is at most a fixed multiple of the number of partitions generated, denoted $p(n, k)$.

LEMMA 1. *For any fixed k , the algorithm GenP generates the set $p(n, k)$*

in constant amortized time per numeric partition.

Proof. The $p(n, k)$ leaves occur at depth $k - 1$. The maximum number of nodes in such a tree is $1 + (k - 1) \cdot p(n, k)$. ■

The main result, Theorem 2, establishes that the number of recursive calls for generating $p(n, k)$ is at most a constant multiple of numeric partitions generated if the depth k of the computation tree does not exceed $n/2$. When k exceeds $n/2$, algorithm *GenP* generates a sequence of ones until the remaining number to partition is at least twice the number of remaining parts. We deal with the cost of generating this path after the main result.

THEOREM 2. *For $k \leq \lfloor n/2 \rfloor$, the algorithm GenP generates the set $p(n, k)$ in constant amortized time per numeric partition.*

Proof. First, the inductive step is made assuming that the size of the function call tree $w(n, k)$ for generating the partitions does not exceed a constant multiple C of the number of partitions $p(n, k)$. An additional term $k^2 - \frac{n^2}{4}$ is needed to make the inductive step but never exceeds zero due to the bound on k .

For $k = \lfloor n/2 \rfloor$, the computation tree has the form depicted in Figure 2. There is a single path of length k on the right that is filled with 2's (and a final 3 if n is odd). The remainder of the computation tree takes the same amount of work required to generate $p(n - 1, k - 1)$. Therefore, we can use the inductive hypothesis to show that the desired bound holds as follows:

$$\begin{aligned}
 w(n, k) &= w(n - 1, k - 1) + k \\
 &\leq Cp(n - 1, k - 1) + (k - 1)^2 - \frac{(n - 1)^2}{4} + k \\
 &\leq Cp(n, k) - C + k^2 - k + 1 - \frac{n^2}{4} + \frac{n}{2} - \frac{1}{4} \\
 &\leq Cp(n, k) + k^2 - \frac{n^2}{4}
 \end{aligned}$$

For $\lfloor n/3 \rfloor < k < \lfloor n/2 \rfloor$, the computation tree has a form similar to that described above. As shown in Figure 3, the left subtree T_1 is the same as above but the rightmost path of 2's eventually branches off into a subtree T_2 .

The depth of T_2 is $n \bmod k$ because this is the remainder after k 2's are placed along the rightmost path of the computation tree. To simplify the calculations, we will substitute $n - 2k$ for $n \bmod k$, which is equivalent when $\lfloor n/3 \rfloor < k < \lfloor n/2 \rfloor$. Thus, the depth of T_2 is $n - 2k$ and the length of the path leading to T_2 can be expressed as $3k - n$.

Note that the rightmost path in T_2 will be composed of 3's. Moreover, the value being partitioned at the root of T_2 is $n - 2(k - n \bmod k)$. By subtracting

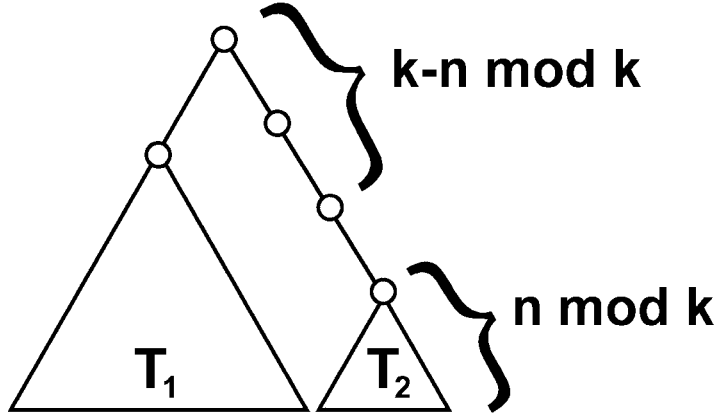


FIG. 3 Computation tree for $\lfloor n/3 \rfloor < k < \lfloor n/2 \rfloor$

one from each part in T_2 , one can see that the shape of the computation subtree T_2 is equivalent to the one for generating $p(n - 2(k - n \bmod k) - n \bmod k, n \bmod k)$, which algebraically reduces to $p(n - 2k + n \bmod k, n \bmod k)$ and finally to $p(2(n - 2k), n - 2k)$. Therefore, we can use the inductive hypothesis to show that the desired bound holds as follows:

$$\begin{aligned}
w(n, k) &= w(n - 1, k - 1) + 3k - n + w(2(n - 2k), n - 2k) \\
&\leq Cp(n - 1, k - 1) + (k - 1)^2 - \frac{(n - 1)^2}{4} \\
&\quad + 3k - n + Cp(n - k, k) + (n - 2k)^2 - \frac{(2(n - 2k))^2}{4} \\
&\leq Cp(n, k) + k^2 - 2k + 1 - \frac{n^2}{4} + \frac{n}{2} - \frac{1}{4} + 3k - n \\
&\leq Cp(n, k) + k^2 - \frac{n^2}{4} + \left(k + \frac{3}{4} - \frac{n}{2}\right) \\
&\leq Cp(n, k) + k^2 - \frac{n^2}{4}
\end{aligned}$$

For $3 < k \leq \lfloor n/3 \rfloor$, the computation tree has a form similar to that described above. As shown in Figure 4, the left subtree T_1 is the same as above but the right subtree T_2 has the same root as the entire computation tree.

Once again, by subtracting one from each part in T_2 , we see that it has the same form as the computation tree for $p(n - k, k)$. Thus, the work to generate both T_1 and T_2 can be accounted for using the inductive hypothesis as follows:

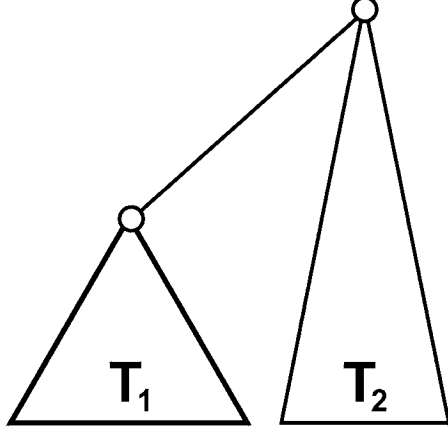


FIG. 4 Computation tree for $3 < k \leq \lfloor n/3 \rfloor$

$$\begin{aligned}
w(n, k) &= w(n-1, k-1) + w(n-k, k) \\
&\leq Cp(n-1, k-1) + (k-1)^2 - \frac{(n-1)^2}{4} \\
&\quad + Cp(n-k, k) + k^2 - \frac{(n-k)^2}{4} \\
&\leq Cp(n, k) + k^2 - 2k + 1 - \frac{n^2}{4} + \frac{n}{2} - \frac{1}{4} \\
&\quad + k^2 - \frac{n^2}{4} + \frac{kn}{2} - \frac{k^2}{4} \\
&\leq Cp(n, k) + k^2 - \frac{n^2}{4} \\
&\quad - 2k + 1 + \frac{n}{2} - \frac{1}{4} + k^2 - \frac{n^2}{4} + \frac{kn}{2} - \frac{k^2}{4} \\
&\leq Cp(n, k) + k^2 - \frac{n^2}{4} - \frac{((n-3k)(n+k-2) + 2k-3)}{4} \\
&\leq Cp(n, k) + k^2 - \frac{n^2}{4}
\end{aligned}$$

Finally, we come to the matter of establishing the inductive basis in k . For $k = 3$, we solve for C using the relationship between $w(n, 3)$ and $p(n, 3)$ established in Lemma 1 in conjunction with the following well-known equivalence [4, p. 67].

$$p(n, 3) = \left\lfloor \frac{n^2 + 4}{12} \right\rfloor \quad (2)$$

According to Lemma 1, we must have

$$w(n, 3) \leq 2p(n, 3) + 1$$

In order to conform to this inductive proof, we must also have

$$w(n, 3) \leq Cp(n, 3) + 3^2 - \frac{n^2}{4}$$

Both constraints on $w(n, 3)$ are satisfied if we select a constant C such that

$$2p(n, 3) + 1 \leq Cp(n, 3) + 3^2 - \frac{n^2}{4}$$

Substituting the result of Equation 2, we find that the smallest satisfying integer is $C = 5$. In essence, C is scaled up such that the boundary established by Lemma 1 is no greater than the boundary given by this theorem.

The smallest value of n to which the theorem applies is $n = 6$ (at $k = 3$), but the inductive basis is $n = 7$ (at $k = 3$) since no larger problems reduce to $p(6, 3)$. In both cases, the theorem holds since the value for C is deliberately selected to work for $k = 3$. ■

Remark: Theorem 2 shows that generating $p(n, k)$ requires no more than 5 times as many invocations of *GenP* as there are solutions generated if $k \leq \lfloor n/2 \rfloor$. However, in practice the worst case appears to be a little over 2.5 times as many function calls required to generate numeric partitions, which occurs at $p(14, 7)$. A further analysis to improve the result of Lemma 1 for the case $k = 3$ would allow a lower constant to be derived.

To conclude the analysis, note that Theorem 2 proves CAT behavior for $k \leq \lfloor n/2 \rfloor$, yet the algorithm *GenP* is correct for all values of k . For $k > \lfloor n/2 \rfloor$, *GenP* generates a sequence of $2k - n$ ones, then it essentially generates $p(2(n - k), n - k)$ with CAT performance. Often, special implementation techniques can be used to short-circuit single-child paths in the computation tree that generate a specific value, such as the initial sequence of $2k - n$ ones. While such a modification would technically extend CAT performance of *GenP* to the case $k > \lfloor n/2 \rfloor$, Ruskey [7] notes that the use of these techniques often results in much lower generator performance in practice. In the particular case of *GenP*, Theorem 3 shows that modification is almost always unnecessary since the additional work of generating the initial sequence of $2k - n$ ones can almost always be amortized as a constant additional cost per numeric partition generated.

THEOREM 3. *For $k \leq n - \delta$ where δ is $\omega(\log(n))$, the algorithm *GenP* generates the set $p(n, k)$ in constant amortized time per numeric partition.*

Proof. Since Theorem 2 proves the result for $k \leq \lfloor n/2 \rfloor$, we consider the case $k > \lfloor n/2 \rfloor$. A sequence of $2k - n = \Theta(k)$ ones is generated, followed

by $p(2(n - k), n - k) = p(2\delta, \delta)$, which produces a number of solutions exponential in δ [4, p.67]. If δ is $\omega(\log(n))$, then the number of solutions is $\omega(k)$, and the result follows. ■

4. CONCLUSION

This paper presented a very simple generator for fixed length numeric partitions and proved that it achieves the best possible performance, which filled a conspicuous gap in the collection of constant amortized time algorithms for lexicographically generating elementary combinatorial objects. The proof is longer and more interesting than might be expected given the algorithm's simplicity because the typical optimization techniques could not be used to eliminate most of the single-child paths in the computation tree. Since the single-child paths could not be accounted for by algorithmic means, it was necessary to account for them mathematically.

Relative to the 25 page implementation of Savage's algorithm in [1], the simplicity of the algorithm *GenP* is clearly desirable. However, in practice one can also achieve approximately 20 percent better performance with a few simple modifications to *GenP* to make it avoid using division and multiplication. Of course, this improvement is only helpful if minimal change order is not required.

Aside from simplicity and speed, the algorithm formulation and analysis of *GenP* derive additional value from the wealth of similar problems that can be solved by simple modifications to the algorithm, then shown to achieve constant amortized time by simple corollaries of Theorems 2 and 3 that exploit computation tree symmetries. In [5], minimal change order algorithms that achieve CAT performance are presented for generating several classes of numeric partitions of n , including strictly odd parts, strictly even parts, and distinct parts. In [5], it is also conjectured that similar methods can be used to develop corresponding minimal change algorithms for fixed length numeric partitions. The remainder of this paper describes the lexicographic (co-lex) counterparts.

Fixed length numeric partitions of n into strictly odd or even parts simply requires that the loop in Figure 1 consider only odd or even numbers. In both cases, the loop's step value should be changed from one to two, and in the even case the assignment to l in the computation tree root should be changed from one to two. As well, the assignment to h should be decremented if it is not odd in the odd case or even in the even case, though nothing is technically wrong with omitting this step. Constant amortized time behavior is established by symmetry with the computation trees for generating the set $p(\frac{n+k}{2}, k)$ in the odd case and the set $p(\frac{n}{2}, k)$ in the even case.

Generating fixed length numeric partitions with distinct parts, given as another open problem in [7], requires only a little additional effort. In Figure 1, the parameter value associated with l in non-root computation

tree nodes should change from $A[k]$ to $A[k] + 1$, and the assignment to h must be changed to

$$\left\lfloor \frac{n - \binom{k}{2}}{k} \right\rfloor$$

which subtracts from n a value that accounts for the changed assignment to l . Similarly, by subtracting from each part of a partition with distinct parts a value equal to the depth in the computation tree of the function call that assigns the part (starting with zero for the root), the constant amortized time behavior of this modified generator is established by symmetry with the computation tree for generating the set $p(n - \binom{k}{2}, k)$.

REFERENCES

- [1] J. Bingham. *Implementation of Savage's CAT numeric partition generator*, Personal communication, 1999. Available at www.theory.cs.uvic.ca/cos/inf/nump/NumPartition.html
- [2] J. Boyer. Simple constant amortized time generation of fixed length numeric partitions. Presentation at *Tenth SIAM Conference on Discrete Mathematics*. Minneapolis, MN, June 2000.
- [3] M. Coleman and M. Taylor. Algorithm 403: Circular integer partitioning. *Communications of the ACM*, 14(1):48, 1971.
- [4] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press, Boca Raton, Florida, 1999.
- [5] J. Rasmussen, C. Savage, and D. West. Gray code enumeration of families of integer partitions. *Journal of Combinatorial Theory, Series A*, 70(2):201-229, 1995.
- [6] E. Reingold, J. Nievergelt and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
- [7] F. Ruskey. *Combinatorial Generation*. University of Victoria Computer Science Department CSC 528 Course Book, 1996.
- [8] C. Savage. Gray code sequences of partitions. *Journal of Algorithms*, 10:577-595, 1989.