Computing Partitions with Applications to the Knapsack Problem

by

Ellis Horowitz and Sartaj Sahni

Department of Computer Science
Cornell University
Ithaca, New York

Technical Report No.
72-134

Revised July 4, 1972

Computing Partitions with Applications to the Knapsack Problem

E. Horowitz and S. Sahni

Abstract

Given r numbers $s_1,\ldots,s_r$, algorithms are investigated for finding all possible combinations of these numbers which sum to M. This problem is a particular instance of the 0-1 unidimensional knapsack problem. All of the usual algorithms for this problem are investigated both in terms of asymptotic computing times and storage requirements, as well as average computing times. We develop a technique which improves all of the dynamic programming methods by a square root factor. Using this improvement a variety of new heuristics and improved data structures are incorporated for decreasing the average behavior of these methods. The resulting algorithms are then compared on a wide set of data. It is then shown how these improvements can be applied to various versions of the knapsack problem.

Key Words and Phrases: partitions, knapsack problem, dynamic programming, integer optimization

CR Categories: 5.25, 5.39, 5.42.

Computing Partitions with Applications to the Knapsack Problem

E. Horowitz and S. Sahni

Introduction

Given r numbers $s_1,\ldots,s_r$ we wish to find all possible combinations of these numbers which sum to M. This rather simply stated problem is at the root of several interesting problems in number theory, operations research and polynomial factorization. In the first case it is closely related to the classical number theory study of determining partitions. Phrased in our terminology, determining partitions of M would imply that $s_i = i$ and $s_r = M$. So here we are concerned with a more general problem than partitions. In [3], p. 273 Hardy and Wright provide generating functions but no good computational scheme for generating such partitions. If we restrict the s_i and M to be integers and include an additional set of numbers p_i then we have an integer programming form of what is usually referred to as the knapsack problem. In its simplest form one wishes to find the most desirable set of quantities a hiker should pack in his knapsack given a measure of the desirability of each item (p_i or *profit*) subject to its weight (s_i) and the maximum weight that the knapsack can hold (M). The partition problem is shown to be a special case of the 0-1 unidimensional knapsack problem and it will

be shown how a method for speeding up the partition problem can be more generally used to speed up the knapsack problem. In [13], Bradley shows how a class of problems can be reduced to knapsack problems. Thus, a more efficient method for knapsack solving algorithms is extremely useful. An implementation of this method has a wide variety of applications. In one reported case, [7] the motivation arose from capital budgeting problems in which investment projects are to be selected subject to expenditure limitations in several time periods. After solving our original partition problem we will show how our new techniques can be incorporated into an efficient knapsack algorithm. A survey of algorithms for the different variations of the knapsack problem is given in [5]. Much of the early work in the knapsack problem was done by Gilmore and Gomory, see[1] and [2]. Finally in [4] our original motivation for the partition problem arose as a subalgorithm for polynomial factorization where M is the degree of the given polynomial and the s_i's are suspected degrees of its irreducible factors.

At the moment all known methods for the partition and knapsack problems take exponential time. In [9] and [10] it is shown that both the 0-1 knapsack problem and the problem of finding one partition are p-complete, i.e. if one could find a polynomial time bounded algorithm for either of these problems then one would have polynomial algorithms for a wide variety of problems for which there is no known polynomial algorithm. Specifically this would lead to polynomial algo-

gorithms for the traveling salesman problem, multicommodity networks flows, simulation of polynomial time bounded nondeterministic Turing machines by deterministic ones, etc. A more complete list of p-complete problems can be found in [9] and [10]. In view of this theoretical result, it is clear that finding a polynomial algorithm for the 0-1 knapsack or partition problem would be difficult (if there exists such an algorithm). It is therefore of interest to obtain subexponential algorithms and to investigate the use of heuristics in an effort to improve the computing times for these problems. This is precisely the sort of development that this paper takes, first giving methods with reduced asymptotic bounds and then refining these algorithms with heuristics special data structures and testing.

In Section 2 we will precisely formulate the problem using the convenient concept of multisets. In Section 3 we will summarize the various algorithms that have been proposed, present our own refinements and then analyze the resulting computing times and storage requirements. A new technique which substantially reduces the worst case asymptotic computing time will be given. Also we will examine the same algorithm using different data representations on the computer. Then in Section 4 empirical studies will be examined so as to determine the best overall algorithm. Finally in Section 5 it will be shown how these new techniques can be easily incorporated into the 0-1 knapsack problem so as to maintain the same advantages of efficiency. An appendix contains the listings of the final programs that were developed.

2. Problem Definition

We begin with the mathematical formulation of our problem.

Definition 1 A multiset S is a collection of elements s_i* denoted by $S = \{s_i\}$.

Definition 2 A set S is a multiset whose elements satisfy $s_i \neq s_j$ if $i \neq j$.

Definition 3 The cardinality of a multiset S , denoted by $|S|$, is defined to be the number of elements in S . If $|S| = r$, then S will often be written as S_r .

Definition 4 An M-partition of a multiset $S_r = (s_1,\ldots,s_r)$ of cardinality r is an r-tuple

$$\left.\begin{array}{l} \delta = (\delta_1,\delta_2, \ldots, \delta_r) \text{ , where} \\ \delta_i \in (0,1) \quad 1 \leq i \leq r \\ \text{and} \\ \sum_{i=1}^{r} \delta_i s_i = M \end{array}\right\} \quad \ldots(1)$$

*Without loss of generality we shall restrict ourselves to the case where the s_i are positive integers.

Example

$S1 = \{1,9,1,5,4\}$ is a multiset but not a set.

$S2 = \{1,9,5\}$ is both a set and a multiset.

$|S1| = 5$ and $|S2| = 3$

$\delta = 11010$ is a 15-partition of S1.

The 15-partitions of S1 are 11010 and 01110.

Definition 5 An algorithm will be said to enumerate the M-partitions of S_r iff it generates all r-tuples δ satisfying (1) and no other δ's .

Lemma 1 There exist multisets and an M for which the number of M-partitions is exponential in the cardinality of the multiset.

Proof Consider $S_r = (1,1, \ldots, 1)$, r even and $M = r/2$. Then the number of r-tuples δ which satisfy (1) is

$$\binom{r}{m} = \binom{r}{r/2} = \frac{r!}{(\frac{r}{2})!(\frac{r}{2})!}$$

Using Stirling's approximation for r! we get

$$\frac{r!}{(\frac{r}{2})!(\frac{r}{2})!} > \frac{\sqrt{\pi r}\,(\frac{r}{e})^r}{\frac{\pi r}{2}\,(\frac{r}{2e})^r} = \frac{2^{r+1}}{\sqrt{\pi r}}$$

Corr. Any algorithm which enumerates the M-partitions of a multiset S_r must have a worst case computing time that is exponential in r.

Lemma 2 The maximum number of distinct sums obtainable from a multiset S_r is 2^r. This number is in fact achieved by some S_r.

Proof (i) $\exists$ only 2^r distinct r-tuples δ for which $\delta_i \in (0,1)$, $1 \leq i \leq r$

(ii) let $S_r = (2^0, 2^1, \ldots, 2^{r-1})$

Each of the δ's in (i) is now the binary representation of the sum, $\sum \delta_i 2^i$ and so represents a distinct sum from the other δ's.

3. The Algorithms

We shall now look at several classical algorithms for enumerating M-partitions. Starting with the simple enumeration and branch and bound type algorithms 1(a) and (b), we shall go to the dynamic programming type algorithms 2(a), 3(a) and 4(a). We shall then show that by "splitting" the multiset S we can obtain algorithms that have a worst case computing time a square root of that for the dynamic programming algorithms. This is represented in the algorithms 2(b), 3(b) and 4(b). Improvements in the average behavior of the algorithms are obtained through the use of

heuristics. In section 4 empirical results are given to allow for comparing the usefulness of the heuristics used. The empirical results will show that the new algorithms are significantly better than the ones without splitting over a wide range of input data.

Definition 6 Union $\biguplus$, $S_{r_1} \biguplus S_{r_2}$ is a multiset such that $x \in S_{r_1} \biguplus S_{r_2}$ with n occurrences iff the number of occurrences of x in S_{r_1} plus the number of occurrences in S_{r_2} is n.

Definition 7 Ordered Union $\vec{\biguplus}$, $S_{r_1} \vec{\biguplus} S_{r_2}$ is a multiset such that $x \in S_{r_1} \vec{\biguplus} S_{r_2}$ under the same conditions as definition 6 and in addition the elements of $S_{r_1} \vec{\biguplus} S_{r_2}$ are ordered.

Example If $S_3 = \{1,2,1\}$ and $S_4 = \{1,2,2,3\}$

then $S_3 \biguplus S_4 = S_7 = \{1,2,1,1,2,2,3\}$

If $S_3 = \{1,3,5\}$ and $S_4 = \{2,3,4,4\}$

then $S_3 \vec{\biguplus} S_4 = S_7 = \{1,2,3,3,4,4,5\}$.

Algorithm 1(a)

Here we generate all 2^r possible δ's and determine which ones satisfy equation (1)

1) [Initialize] $\delta_r \leftarrow (0, \ldots, 0)$; Do step (2) $2^r - 1$ time

2) [find new δ] $\delta \leftarrow \delta+1$; (binary addition)

If $\sum_{1\le i\le r} \delta_i s_i = M$ then output

$\delta = (\delta_1, \ldots, \delta_r)$

Storage required: $O(r)$.

Computation time: $O(r2^r)$.

As we shall see from the empirical studies in Section 4 this method works extremely slowly for even small values of r. So despite the fact that its storage requirements are linear in the cardinality of the input set, its real effectiveness is severly limited because of time. We note that this algorithm could be somewhat speeded up through the use of heuristics as explained in [11]. However, we next give a backtracking or branch and bound algorithm 1(b), below, that is considerably superior to 1(a) and so we shall not concern ourselves further variations of 1(a).

Now we give a recursive algorithm which maintains the linear storage requirement and reduces the bound on the computation time from $r2^r$ to 2^r. This method is well-known and is perhaps the one most commonly employed for solving knapsack problems. A nonrecursive version without heuristics can be found in Beckenbach, [8], pp. 25-27. In the version we give here we have added several heuristics in steps (1) and (2). These do not change the order of the method, but do aid considerably in improving its overall performance. Similar heuristics have been used by Weingartner and Ness in [7].

Algorithm 1(b) PARTS(s, i, rem,δ) [Backtracking or Branch and Bound]

The generation of certain δ's is aborted by using heuristics in steps (1) and (2). It is assumed that the elements of $S_r = (s_1, \ldots, s_r)$ are initially ordered $(s_1 \leq s_2 \leq , \ldots, \leq s_r)$. (The choice of ordering is somewhat arbitrary. Had we ordered the s_i's in decreasing order then we would not have been able to use the heuristic of step 2 below.)

The specific heuristics used are:

1) Step 1 If the partial sum (s) plus the total sum left (rem) is not enough to reach M then abort;

(2) Step 2 If the partial sum (s) added to the next number s_i exceeds M then abort as all other s_i's are at least as large as this one (because s is ordered).

Let

s = the present partial sum;

i = index of the next s_i to be processed;

rem = the remaining sum, $\sum_{i+1 \leq j \leq r} s_j$

δ = the set of j such that $\sum_{j \in \delta} s_j = s$

The algorithm is recursive and is initially invoked as PARTS(0, 1, $\sum_{1 \leq i \leq r} s_i$, NULL).

1) [Test heuristics] If $s + rem < M$ then return
If $s + rem = M$ then output $\delta \cup \{i, i+1, \ldots r\}$ return.

2) [Try next s_i] If $s + s_i > M$ then return.
If $s + s_i = M$ then output $i \cup \delta$;
If $i < r$ GO TO step 4
else return.

3) [Recursion] If $i < r$ then CALL PARTS ($s + s_i$, $i + 1$, $rem - s_i$, $i \cup \delta$); else return.

4) [Recursion] CALL PARTS (s, $i + 1$, $rem - s_i$, δ);

5) [all done] return.

Storage required: $O(r)$
Computation time: $O(2^r)$

For each partition, this algorithm produces an r-tuple δ. Thus an additional time of rQ is required to print all the partitions, where Q is the total number of partitions. Though this method is much better than 1(a) in terms of the time requirements, let us now look at even faster methods.

In the next algorithm we compute the sums obtainable from all possible sub-multisets of S. Along with each sum is kept an encoding of the indices used to obtain that sum. Multiple copies of sums are retained.

Algorithm 2(a) $S_r = (s_1, \ldots, s_r)$

A is a multiset of 2-tuples (a_1, a_2) where

a_1 is a partial sum,

a_2 is an encoding of the j's such that $\sum_{(j \mid j-1 \in a_2)} s_j = a_1$.

The encoding used is $a_2 = \sum_{(j \mid s_j \in a_1)} 2^{j-1}$.

1) [Initialize] $i \leftarrow 0$, $A \leftarrow ((0, 0))$, $IC \leftarrow 1$,
Do step 2 for $i \leftarrow 1, \ldots, r$.

2) $A \leftarrow A \cup (A + (s_i, IC))$; $IC \leftarrow IC + IC$;

Note: In step 2 only those (a_1, a_2) for which $a_1 < M$ are retained. If $a_1 = M$, a_2 is output. (Strictly speaking we shall have to output decode (a_2)).

Storage required: $O(2^r)$

Computation time: $O(\max(2^r, rQ))$

To see how algorithm 2a works, consider finding all the 8 partitions of $S_3 = \{1,3,4\}$

value of i	A
0	$\{(0,0)\}$
1	$\{(0,0),(1,2^0)\}$
2	$\{(0,0),(1,2^0),(3,2^1),(4,2^0 + 2^1)\}$
3	$\{(0,0),(1,2^0 0,(3,2^1),(4,2^0 + 2^1),(4,2^2),$ $(5,2^0 + 2^2),(7,2^1 + 2^2)\}$

and the vector (111) is output corresponding to the partition (1 + 3 + 4).

We note that while implementing the encoding scheme, above, one would use bit strings to represent the second component of the 2-tuples of A, with the jth bit set to 1 iff s_j was used in obtaining the corresponding sum. This has the advantage that no decoding is needed at the end to obtain the partition.

Theorem 3.1 In the worst case the computing time for algorithm (2a) is $O(\max\{2^r, rQ\})$ and its storage requirements are $O(2^r)$.

Proof Let $|A| = k$ when $i = j$. Then the cardinality of A for $i = j + 1$ is $\leq 2k$. The time taken for step 2 when $i = j$ is k and for $i = 1$, $k = 1$.

Therefore the total time is $\leq \sum_{i=1}^{r-1} 2^i = 0(2^r)$ and the decode time per partition is $0(r)$.

Though algorithm 2(a) has a much worse storage requirement than 1(b), it actually remains fairly competitive with 1(b) in terms of time. However, it is possible to make a significant improvement in method 2 by splitting the input into 2 sets as will be done in algorithm 2(b). The procedure of "splitting" is that rather than generate all possible sums for the given multiset S_r of cardinality r, we consider two smaller multisets T and U such that the union of the two gives S_r. Algorithm 2(a) is now applied to both T and U . However now the multiset of obtainable sums is maintained in increasing order in terms of the first component of the 2-tuples. It is now possible to combine the results of the two applications of method 2(a) to T and U to obtain all M-partitions, and in such a way that the entire process requires only a square root of the time and space required (in the worst case) if 2(a) were directly used on S_r.

Algorithm 2(b)

The multiset S_r is divided into two sub-multisets T, U such that[1]

$$|T| = t = \lfloor r/2 \rfloor \ , \ T = (s_1, \ldots, s_{r/2})$$
$$|U| = u = r - t \ , \ U = (s_{r/2+1}, \ldots, s_r)$$

As in 2a, A and B are multisets of 2-tuples. However now A and B are kept ordered, i.e. if $(a_{i_1}, a_{i_2}) \in A$ and $(a_{j_1}, a_{j_2}) \in A$ then $a_{i_1} < a_{j_1}$ implies $i_1 < j_2$ and similarly for B .

1) $i \leftarrow 0$, $A \leftarrow ((0, 0))$, $IC \leftarrow 1$
   Do step 2 t times for $i \leftarrow 1, \ldots, t$;

2) $A \leftarrow A \vec{\cup} (A + (t_1, IC))$; $IC \leftarrow IC + IC$

3) $i \leftarrow 0$, $B \leftarrow ((0, 0))$, $IC \leftarrow 1$
   Do step 4 r - t times for $i \leftarrow t + 1, \ldots, r$;

4) $B \leftarrow B \vec{\cup} (B + (u_1, IC))$, $IC = IC + IC$;

5) Pick off pairs $(a_{i_1}, a_{i_2}) \in A$

   $(b_{j_1}, b_{j_2}) \in B$

   such that $(a_{i_1} + b_{j_1}) = M$.

   Then output partition (a_{i_2}, b_{j_2}) .

---

[1] $\lfloor r/2 \rfloor$ = largest integer $\leq r/2$

$\lceil r/2 \rceil$ = smallest integer $\geq r/2$

As an example for 2b consider

$S_4 = \{1,2,4,8\}$, $M = 14$

then $T = \{1,2\}$

$U = \{4,8\}$

$A = \{(0,0),(1,2),(2,2^1),(3,2^0 + 2^1)\}$

$B = \{(0,0),(4,2^0),(8,2^1),(12,2^0 + 2^1)\}$

A search of A and B shows that the only M-partition of S_4 is 0111.

Storage required: $O(2^{\lceil r/2 \rceil})$.

Proof The multisets A and B cannot become larger than this by lemma 2.

Computation time: $O(\max(2^{\lceil r/2 \rceil}, rQ))$ where Q is the number of partitions.

Proof Since in steps 2 and 4, A and B are ordered and hence $A + (t_1, IC)$ and $B + (u_1, IC)$ are ordered the merging necessary to keep $A \vec{\cup} (A + (t_1, IC))$ and $B \vec{\cup} (B + (u_1, IC))$ ordered can be done in time proportional to $|A|$ and $|B|$ respectively. Therefore from algorithm 2(a) the time for steps 1 – 4 is

$$O(2^t + 2^u) = O(2^{\lceil r/2 \rceil}) .$$

Step 5 requires time $O(\max(2^{r/2}, rQ))$. To see this consider the algorithm below which realizes this step:

Let $|A| = a$, $|B| = b$, $A = ((a_i, p_i)\ 1 \le i \le a)$, $B = ((b_i, q_i)\ 1 \le i \le b)$ where p_i, q_i contain encodings of all combinations of elements that sum to a_i, b_i. Then

1) $i \leftarrow 1$; $j \leftarrow b$;

2) DO WHILE ($i \le a$ and $j \ge 1$);
   If $a_i + b_j < m$ then $i \leftarrow i+1$; go to (L)
   If $a_i + b_j > m$ then $j \leftarrow j-1$; go to (L)
   Output all combinations of p_i, q_j; $i \leftarrow i+1$
   L:END

3) end.

Thus the time required is $O(\max(a, b, rQ))$. Now $a, b < 2^{\lceil r/2 \rceil}$ so the time for step 5 is $O(\max(2^{\lceil r/2 \rceil}, rQ))$ and similarly for the entire algorithm.

Theorem 3.2 Algorithm 2(b) enumerates all the M-partitions of S_r.

Proof Let $\delta = (\delta_1, \ldots, \delta_{r/2}, \delta_{r/2+1}, \ldots, \delta_r)$ be an M-partition of S_r. $\bar{\delta} = (\delta_1, \ldots, \delta_{r/2})$, $\bar{\bar{\delta}} = (\delta_{r/2+1}, \ldots, \delta_r)$. Then $\sum_{1 \le i \le r/2} a_i \delta_i \le M$ and

$$\sum_{r/2+1 \le i \le r} s_i \delta_i \le M .$$

Since all partitions $\le M$ of sets $T = (s_1, \ldots, s_{r/2})$ and $U = (s_{r/2+1}, \ldots, s_r)$ are produced by steps 2 and 4, then for any M-partition δ of S_r there must exist a $\overline{\delta}$ from A and $\overline{\overline{\delta}}$ from B such that $\delta = \overline{\delta} \cup \overline{\overline{\delta}}$. Therefore we must show that in step 5 every possible combination of $\overline{\delta} \in A$ and $\overline{\overline{\delta}} \in B : \delta = \overline{\delta} \cup \overline{\overline{\delta}}$ and δ is an M-partition, is found. By the previous proof, the a_i and b_i are ordered and suppose they are distinct. Associated with each a_i is the set of $\overline{\delta} : \sum_{1 \le j \le r/2} s_j \delta_j = a_i$. Similarly for $\overline{\overline{\delta}}$ in It is sufficient to show that if we are looking at a_i, b_j then every other $\overline{\delta}$ associated with a_k, $k < i$ such that $\overline{\delta} \cup \overline{\overline{\delta}}$ is an M-partition has already been output. If $a_i + b_j \le m$ then $a_k < a_i$ implies $a_k + b_j < m$ and hence there are no previous M-partitions. If $a_i + b_j > m$ then by the above algorithm either for all a_k, $a_k + b_j < m$ or $\exists k : a_k + b_j = m$. In this case all combinations of $p_i, q_i = (\delta, \overline{\delta})$ are output. Thus all previous M-partitions have been found and algorithm 2(b) produces them all.

The improvement in computing time exhibited by Algorithm 2(b), naturally, leads to the question of whether further improvements can be achieved by dividing the original set into more than two parts. If we divide the multiset into k parts then all the partial sums can be computed in $O(k\ 2^{r/k})$ time. However, there appears to be no way of combining the results of the k-parts in time less than $O(2^{r/2})$ to get the partitions. For example if we chose $k = 4$ then we would obtain four lists of partial sums of maximum length $2^{r/4}$ each. To obtain a partition of M we would choose one element from list 1, say x_1 and then determine all partitions of $M - x_1$ from the remaining three lists. Such a process requires more than $O(2^{r/2})$ time. Alternatively we could combine pairs of lists obtaining two lists of size $O(2^{r/2})$, but this just reduces to method 2b.

We have previously noted that a polynomially bounded algorithm for the partition problem would have important consequences on the existence of polynomially bounded algorithms for many other problems. Though the splitting technique cannot be iterated and further with a subsequent improvement it can be successfully applied to other p-complete problems. Thus, $O(2^{r/2})$ algorithms can be given for problems such as 1) finding an exact cover of a graph; 2) finding the hitting set of a graph.

Now we study an entirely different approach to this problem which avoids the generation of all partitions as in 2(a) and 2(b). Instead it first produces r sets $S^{(1)}, \ldots, S^{(r)}$ so that $S^{(1)}$ contains all possible combinations of $s_1, \ldots, s_i$. Then a retracing procedure is used to find those combinations which give M in $S^{(r)}$.

Definition 8 The sumset of S_r, denoted by $S^{(r)}$ is the set of all sums $\sum_{j \in J} s_j$ where $J \subset (1, \ldots, r)$.

Definition 9 Ordered Union on Sets $S_{r_1} \vec{\cup} S_{r_2}$ is a set such that $x \in S_{r_1} \vec{\cup} S_{r_2}$ implies either $x \in S_{r_1}$ or $x \in S_{r_2}$ and the elements are ordered.

Example $S_4 = (1,1,2,2)$

The sumsets are:

$S^{(0)} = \{0\}$

$S^{(1)} = \{0,1\}$

$S^{(2)} = \{0,1,2\}$

$S^{(3)} = \{0,1,2,3,4\}$

$S^{(4)} = \{0,1,2,3,4,5,6\}$

Algorithm 3(a) (Musser [4])

This works in essentially two stages

a) compute the sumsets of the sets,
$S_i = \{s_1, \ldots, s_i\}$ $1 \le i \le r$.

b) Where M appears in $S^{(r)}$ generate all partitions creating M by using the sumsets $S^{(1)}, \ldots, S^{(r-1)}$.

Generate sumsets

1. $S^{(0)} \leftarrow \{0\}$
2. For $j = 1, \ldots, r$ $S^{(j)} \leftarrow S^{(j-1)} \dot{\cup} (S^{j-1} + (s_j))$
   Generation of partitions, using $s^{(j)}$. This is a recursive procedure G(j, n, J) initially invoked as G(1, M, NULL).
3. If $n = 0$, output J , Return.
4. If $n - n_j \in S^{(j-1)}$ Call $G(j-1, n-n_j, \{j\} \cup J)$
5. If $n \in S^{(j-1)}$ Call G(j-1, n, J). Exit.

This algorithm differs from 2a chiefly in the scheme used for obtaining the indices that sum to a particular number (binary encoding in the case of 2a and trace back involving search in 3a). It should be clear that the binary encoding scheme would be superior when the number of partitions is large.

Theorem Algorithm 3(a) enumerates the M-partitions of S.

Proof See [4] p. 33.
Storage requirements: $O(\min(2^r, rM))$

Proof $|S^{(0)}| = 1$
If $|S^{(i)}| = k$ then $|S^{(i+1)}| \leq 2k$
Therefore the total space $= \sum_{0}^{r} 2^i = O(2^r)$.

Note however that the maximum sum in any of the $S^{(i)}$ is M (or $\cdot \sum s_i$ if no heuristics are used).
So we get another bound on the storage i.e. $O(rM)$.
Thus the storage required in $O(\min(2^r, rM))$

Computation time:
Steps 1 and 2: $O(\min(2^r, rM))$
Steps 3 through 5: $O(r^2Q)$, Q = number of partitions

Algorithm 3(b)

This is essentially 3(a) with S_r split into two parts as in 2(b). The worst case storage space is now

$$O(\min(2^{r/2}, rM))$$

and the computation time is

$$O(\max(2^{r/2}, r^2Q))$$

There are two strategies that can be employed for implementing method 3. Musser's implementation of algorithm 3(a) uses bit strings. The sets $S^{(i)}$ are bit strings in which the jth bit is a 1 iff j has a partition from the first i elements of the multiset. Such an implementation has a space requirement of $O(r M)$ and also an asymptotic computing time bound of $O(r M)$. This implementation is good when M is guaranteed to be small. However, the following example illustrates the drawbacks of this technique for large M.

Example: $S = (1, 10^5, 10^6)$, $M = 10^6 + 10^5$

The storage needed to handle this problem by the bit string technique is about 3×10^6 bits (careful programming could reduce this to around 10^6 bits). The computing time would be around 10^5 basic operations. However the implementation suggested by 3(a) needs only 8 machine words and about 8 units of time. Thus the dependency of the bit approach on the magnitude of the number can severely effect its general usefulness.

Naturally, if we were writing an algorithm for gener use we would avoid bit strings. The maximum storage gain t can be expected, for small M, through the use of bit stri is only a factor of β where β equals the number of bits word in the machine. Finally, it is often the case that th number of combinations which are generated is considerably less than 2^r. This will be reflected in our implementati by a decrease in storage needs whereas the bit approach is still dependent upon the magnitude of the number.

Now we present the last pair of algorithms. Later we shall see that their asymptotic bounds will be at least as good as all of the previously described methods and actual tests indicate that they are far superior.

Algorithms 4(a), 4(b)

These are the same as 2(a) and 2(b) respectively with the exception that A and B are now sets rather tha multisets. Eliminating multiple occurrences of the same sum at each stage easily overcomes the extra bookkeeping needed. Thus, encodings of all possible vectors resulting in a sum in A or B are kept in an auxiliary array with only 1 pointer; a pointer to the first partition of that su As for algorithms 2(a) and 2(b) the worst case storage and computing time bounds remain the same. However, in the next section we shall examine the extent to which these algorithms are an improvement.

Storage required: $O(2^{\lceil r/2 \rceil})$

Computing time: $O(\max(2^{\lceil r/2 \rceil}, rQ))$

Table 1 summarizes the upper bounds on the computing time and the storage requirements of algorithms 1 through 4. Estimates of the storage constants involved are also given.

Table 1

Algorithm	1a	1b	2a	2b	3a	3b	4a	4b
Time	$r2^r$	2^r	$\max(\frac{2^r}{rQ})$	$\max(2^{\lceil r/2 \rceil}, rQ)$	$\max r^2Q, 2^r)$	$\max(r^2Q, 2^{r/2})$	$\max(2^r, rQ)$	$\max(2^{r/2}, rQ)$
Storage	r	r	2^r	$2^{\lceil r/2 \rceil}$	$\min(2^r, rM)$	$\min(2^{r/2}, rM)$	2^r	$2^{r/2}$
Actual storage used	2r	2r	$(2\frac{1}{2})2^r$	$(4\frac{1}{2})2^{r/2}$	$\min(rM, 2^r)$	$\min(rM, 2^{r/2})$	$(3\frac{1}{2})2^r$	$(6\frac{1}{2})2^{r/2}$

Q = the number of partitions
M = the desired sum
r = the number of elements

4. Empirical Results

Algorithms 1 through 4 were programmed and tested extensively to determine their average relative performance as opposed to the theoretically obtained 'worst case' computing time and storage requirements. The programs were written in FORTRAN G and tested on an IBM 360/65.

Tests were performed using the following data sets for $S = (s_1, \ldots, s_r)$ and M :

I. $s_i = i \quad 1 \leq i \leq R$

$M = R, 2R, 3R, \quad R(R+1)/4$

II. s_i = random numbers in [1,100]

Let $m = \max(s_i)$

$M = m, 2m, 3m, \quad \lfloor s_i/3 \rfloor, \quad \lfloor s_i/2 \rfloor$

III. s_i = random numbers in [1,1000]

$M = m, 2m, 3m, \quad \lfloor s_i/3 \rfloor, \quad \lfloor s_i/2 \rfloor$

It should be noted that because of the heuristics used, the time to compute M-partitions for $M = \sum_{1 \leq i \leq r} s_i$ is essentially zero for algorithms 1(b), 2(a), 2(b), 4(a) and 4(b). The computing times reported for the cases where the s_i were random numbers is the mean of times obtained for 5 such tests.

The computing times are reported in Tables II (i), (ii), and (iii).

Despite the simplicity of 1(a) and the fact that it requires only linear storage, this method is far too slow for even small values of r. As the 3 tables show, an r of 15 took more than 21 seconds and higher values of r were subsequently much worse. Method 1(b) is a considerable improvement over 1(a), retaining the linear storage feature while its performance is superior to 1(a) by a factor of 10 or more. The combination of the heuristics helps to account for its dramatic improvement over 1(a). In the cases 2(a) versus 2(b), 3(a) versus 3 (b) and 4(a) versus 4(b) the (b) version with the multisets split was always superior. Thus let us compare 2(b), 3(b) and 4(b).

Examining all three tables we see that method 2(b) was faster than 3(b) in almost all circumstances showing the superiority of the binary encoding scheme. However, the ratio of improvement is not a constant but varies considerably with the input data. For instance in Table II (ii) method 2(b) is 10 times faster than 3(b) for $M = \max$, but both methods are about equal for $M = \text{sum}/2$. In any case method 2(b) is overall the more efficient, but its real difficulty is in storage. Note that in Table II (i) method 2(b) runs out of storage on all the data sets whereas 3(b) is able to continue. Therefore method 2(b) was modified to produce method 4(b) by changing the multisets into sets. Not only did this improvement allow 4(b) to continue for much greater r but also decreased the computing time, so that

4(b) is at least as good and often better than 2(b). The empirical results also show that 4(b) is considerably better than 3(b) even in cases where there are only a polynomial number of partitions.

Finally then we are left with algorithms 1(b) and 4(b). Looking at the tables we see that the computing time becomes prohibitive much earlier in 1(b) than in 4(b). In fact for $r = 60$, the maximum r that was tested, 4(b) was able to obtain the answers in 1.4 seconds and needed no more than 30K words. So despite the fact that $2^{r/2}$ is an upper bound on the number of partitions which may exist, empirical tests indicate that this limit is often not achieved. (Note that Lemma 1 in Section 2 shows when this limit will be reached.) Therefore an outright "best method" would probably be 4(b) although method 1(b) has the virtue of guaranteed linear storage.

Table II (i)

Sequential Numbers; Times in milliseconds

** means > 30K words required
* means exceeded time limit

N	R	1a	1b	2b	3b	4a	4b
Max	15	21099.5	16.6	–	66.5	49.9	–
	20		49.9	33.5	216.3	83.2	33.2
	25		116.4	100.2	582.4	249.6	83.2
	30		266.2	300.7	1098.2	648.9	216.3
	35		615.6	1015.9	2082.0	1480.9	416.0
	40		1464.3	2396.6	3910.4	3694.1	1015.0
	45		2462.7	**		6526(44)	1847(44)
						*	2163.0
2 Max	10	499.2	33.2		99.8	33.2	16.6
	15	21099.5	166.4	16.6(15)	382.7	216.3	49.9
	20		981.7	166.0	1382.7	965.4	116.4
	25		2912	599.0	6472.9	2645(23)	216(23)
	28		5890	**	14327	*	349.4
	30		11980.8				798.7
3 Max	10	499.2	66.5	16.6(10)	149.7	33.2	49.9
	15	21099.5	732.1	66.5	815.3	416.3	83.2
	20		6223.3	299.5	4183.2	665.6(16)	199.6
	25		13495(22)	**	7723(22)	**	648.8
	27		–		15128(24)		1198
Sum/2	10	499.2	49.9		116.4		33.2
	15	21099.5	1564.1	83.2	1031.6	599.2	66.5
	20		45643.5	732.1	7321.5	**	432.6
	23		*	**	49687		3111.6(24)

Table II (11)

Random Numbers (1–100)

Times in milliseconds; ** means > 30K words required

* means exceeded time limit

M	R	1a	1b	2b	3b	4a	4b
Max	15	21099.5	39.9		36.6	33.2	23.2
	20		76.5		123.1	80.3	29.9
	25		186.3		485.8	176.3	59.6
	30		392.6		1158.9	402.7	86.9
	35		818.6	106.5	2193.7	755.4	163.0
	40		1720.5	123.1	4962.6	1428	226.1
	45		–	252.9	–	**	373.6
2 Max	15	21099.5	216.2	20	173	180	39.9
	20		1171.4	150	805.3	609	153.1
	25		3120(24)	316.2	2921.7(24)	1320(23)	579.1
	30			858.6	*	**	1484.2
	35			**			2787.2
3 Max	15	21099.5	838.6	76.5	329.5	346	76.5
	20		6951.3	472.6	2489.3	812(17)	256.3
	25			**	13367.5	**	883.6
	30				*		1307.6(28)
Sum/2	15	21099.5	1311.2	552.1	552.1	462.6	83.2
	16		45527.5	1802.4		885	120
	20		*	6133.5	6133.5	**	372.7
	25			*	*		2526.2
Sum/3	15	21099.5	572.4	56.6	426	279.5	69.8
	20		13961.9	286.2	3494.3	825.3	266.2
	25			**		**	1654.0
							2083(26)

Table II (iii)

Random Numbers (1–1000)

Times in milliseconds; ** means > 30 K words required.

* means exceeded time limit

M	R	1a	1b	2b	3b	4a	5b
Max	15	21099.5	26.6	16.6	32.2	36.5	16.6
	20		59.9	26.6	37.2	83.2	36.5
	25		116.5	36.6	53.3	183	73.2
	30		292.9	43.2	129.3	466	99.8
	35		639.0	136.2	176.4	832	196.3
	40		1484.3	136.4	512.5	1534	229.5
	45			269.0		1864(42)	366
	50			469.0		2612(43)	622.3
	55			715.5			726.5
	60			1239.2			1[illegible].6
2 Max	15	21099.5	371.4	43.3	33.3	186.3	46.6
	20		3511	89.9	79.9	642	123
	25		12070.7	—	183	1371(23)	253.6
	30		*	266.2	1357.8	1431(24)	539.1
	35			678.9	3630.9	**	1[illegible].5
	40			**	[illegible].9		12[illegible].6(36)
3 Max	15	21099.5	582.2	49.9	39.9	376	86.5
	20		5544.4	213.0	269.6	1126(18)	253
	25		39137.2	758.8	386.1	**	662.2
	30		*	**	6[illegible].2		1[illegible].2(28)
Sum/2	15	21099.5	1364.5	63.2	36.6	502.5	96.5
	20		38382.4	322.8	2539.3	941.8(16)	382.7
	25		*	**	1[illegible].6	**	1[illegible].6(24)
Sum/3	15	21099.5	525.8	53.2	176.4	326.1	66.5
	20		12625.6	239.6	133.1	772(17)	253.8
	25		*	**	499.2	**	1[illegible]
	30				4[illegible]70		14[illegible]06

5. The Knapsack Problem

The general knapsack problem may be stated as the following integer optimization problem: let p_i be the profits or returns gained by including project i; s_i the amount of resource required for project i; M the total amount of resource that can be allocated, and δ_i the fraction of project i that is accepted. Then we wish to solve:

$$\max \sum_{1 \le i \le r} p_i \delta_i$$

subject to

$$\sum_{1 \le i \le r} s_i \delta_i \le M$$

where δ_i is a non-negative integer. If we restrict δ_i to be the integer 0 or 1 this is called the 0/1 knapsack problem. In this paper we shall be concerned only with this form of the problem. In particular we shall consider applying the methods of the previous sections for computing partitions to produce more efficient knapsack methods. In terms of the knapsack problem we may formulate the partition problem as

$$\max \sum_{1 \le i \le r} s_i \delta_i$$

subject to

$$\sum_{1 \le i \le r} s_i \delta_i \le M$$

$$\delta_i = 0,1$$

Clearly, there is a partition of M in $S = \{s_1, \ldots, s_r\}$ iff $\max \sum_{1 \le i \le r} s_i \delta_i = M$. If we want all the partitions then we

look for all δ for which $\sum s_i\delta_i$ attains its maximum of M. Thus we see that the partitions problem discussed earlier is really a very important instance of the 0/1 knapsack problem

Let us briefly examine the new complications produced by the knapsack. We now have a profit associated not only with each s_i , but subsequently with each partial sum If we adopt algorithm 4(b), then at each iteration for every multiple occurrence of a partial sum we need only retain the one partition which yields the maximum profit. At least this eliminates having to keep multiple copies of either the partial sums or the profits. But in algorithm 3(b) the approac of generating the sumsets and then tracing back to find all existing partitions now seems more attractive. Since there is only <u>one</u> solution in the knapsack problem we can entirely eliminate the overhead of maintaining <u>all</u> possible partition as we generate sums (as in 4(b)) and instead use 3(b) where we need only trace back once. Furthermore, in order to assure that the splitting procedure takes no longer than $O(2^{r/}$ we must now keep not only the sums but their associated prof in increasing order. This can clearly be done, for suppose that for some i we have sums $a_i < a_{i+1}$ but profits $p_i \leq p_{i+}$ Then we can reject the pair (a_{i+1}, p_{i+1}) as not yielding a maximum profit. For,every further possible combination of s $i+2\leq j \leq r$ which would be added to a_{i+1} giving a sum $\leq M$ can as well be added to a_i yielding a greater profit. Therefore the method we first suggest is an adaptation of algorithm 3 the dynamic programming approach where we initially split t set of weights and profits. This method is now given:

Algorithm Knap (1) [Splitting]

Step 1 Divide the multiset S, of weights, into two multisets T and U as in 2(b). Let the associated profit sets be PT and PU respectively.

$$\text{Set} \quad F_o(i) = o \qquad o \le i \le M$$
$$\text{and} \quad G_o(i) = o \qquad o \le i \le M$$

Step 2 Compute $F_k(x) = \max(F_{k-1}(x), F_{k-1}(x-t_k) + PT_k)$

$$1 \le k \le \lfloor r/2 \rfloor$$

$$G_k(x) = \max(G_{k-1}(x), G_{k-1}(x-u_k) + PU_k)$$

$$1 \le k \le r - \lfloor r/2 \rfloor$$

Step 3 [find an optimal solution]

Search $F_{\lfloor r/2 \rfloor}$ and $G_{r-\lfloor r/2 \rfloor}$ in a manner similar to algorithm 2(b) to find an optimal pair x,y such that $x+y \le M$ and $F_{\lfloor r/2 \rfloor}(x) + G_{r-\lfloor r/2 \rfloor}(y)$ is a maximum.

While actually implementing Step 2 we do not compute F and G for all $x \in [0,M]$ but only at those points x for which there is an x - partition in the weights currently considered in $F_4(x)$ is computed only at those x which can be represented as the sum of a sub-multiset of the weights t_1, t_2, t_3 and t_4.

It follows immediately from the previous sections that this algorithm has a worst case computing time and storage requirement of $O(\min\{2^{r/2}, rM\})$. We note that previous dynamic programming algorithms for the knapsack problem, see [2,5,6,and 7] require $O(\min\{2^{r}, rM\})$ time and space. Thus for large M our algorithm again represents a square root improvement.

For a thorough comparison we now consider the best branch and bound procedure that has been proposed, algorithm 1(b) as applied to the knapsack problem. We include the heuristic as given by Kolesasar in [12], who suggests solving a simple linear programming problem at each stage of the branch. While he had considerable success comparing 1(b) with his heuristic to method 1, he did not compare it with any dynamic programming algorithms. Asymptotically KNAP(1) is superior to algorithm 1(b), but we will also test these 2 methods extensively to determine more precisely their expected behavior. We now present the branch and bound algorithm 1(b) with Kolesasar's heuristic for the Knapsack problem

Algorithm Knap(2) [Branch and Bound]

s = the present partial sum

p = profit associated with this sum

i = index of next s_i to be processed

δ = set of j such that $\sum_{j\in\delta} s_j = s$

P = maximum profit obtainable

A = set of j that yield this profit

$W = \sum_{j\in A} s_j$

1) [Initialize] Order the s_i in decreasing order of p_i/s_i.

Set P,p, W,s,A,δ = 0

and i = 1

2) [Test Heuristic] Solve the corresponding linear program

$$\max: Z = \sum_{k=i}^{r} p_k\delta_k$$

$$\text{subject to: } \sum_{k=i}^{r} s_k\delta_k \le M$$

$$0 \le \delta_k \le 1 \qquad i \le k \le r$$

IF $P \ge Z + p$ GO TO(5)

3) [Put in next item that fits]

Check for first k for which $s_k \le M$

If none, set $\delta_i,\ldots,\delta_r = 0$, GO TO(4)

Set $M = M-s_k$, $p = p + p_k$, $s = s + s_k$,

$\delta_i,\delta_{i+1},\ldots \delta_{k-1} = 0$, $\delta_k = 1$

i = k + 1. If $i \le r$ go to (2) else go to (4)

4) [Save new solution] i = r + 1

IF $P \ge p$ go to (5)

ELSE P = p, A = δ, W = s, go to (5)

5) [Backtrack]

Find largest $k<i$ for which $\delta_k=1$.

If no such k we are done with optimal solution ℓ.

ELSE $M=M+s_k$, $p=p-p_k$, $s=s-s_k$, $\delta_k=0$, $i=k+1$, go to (2).

The linear program of step (2) is simply solved by setting $\delta_i, \delta_{i+1},\ldots\delta_\ell=1$, $\delta_{\ell+1}=(M-\sum_i^\ell s_k)/s_{\ell+1}$ where ℓ is the largest index for which $\sum_i^\ell s_k \leq M$ (lb $\ell=r$ then just use $\delta_i,\ldots\delta_r=1$ with $z=\sum_i^r p_k$ as the solution)

We note that Algorithm KNAP(2) takes $O(r2^r)$ in computing time and so asymptotically KNAP(1) is certainly superior. We will now test these 2 algorithms empirically on 2 variations of the 0/1 knapsack problem: a) finding a single solution with maximum profit; and b) finding the optimal solution with maximum profit and minimum weight. We note that a 3rd variation, namely finding all solutions is essentially equivalent to (b).

For both variations a variety of data sets were constructed to reflect the several degrees of freedom which are possible, i.e. we can choose the weights, the profits, and the size of the knapsack. Data set I consists of random weight, s_i, and random profits p_i. Data sets II (a) and II(b) consist of random weights, s_i and profits p_i such that

$$p_i = s_i + \begin{cases} 10 & \text{, if } 1 \le s_i \le 100 \quad \text{II(a)} \\ 100 & \text{, if } 1 \le s_i \le 1000 \quad \text{II(b)} \end{cases}$$

Thus as the weights increase the profits are accordingly increased. Data sets III(a) and III(b) are constructed by choosing random profits, p_i and then choosing the weights s_i such that

$$s_i = p_i + \begin{cases} 10 & \text{if } 1 \le p_i \le 100 \\ 100 & \text{if } 1 \le p_i \le 1000 \end{cases}$$

Thus correspondingly greater profits have greater weights. Finally we consider 2 special types of data which serve both to exploit KNAP(2) to its fullest potential and to show how disastrous it can be. Data set IV has sequential $s_i=i$, $M=2*\max\{s_i\}$ and the ratios p_i/s_i are all equal. Data set V has random weights s_i, equal ratios p_i/s_i but there exists no partition.

Table I represents KNAP(1) versus KNAP(2) as tested for finding a single solution on data sets I,II and III. For each choice

of M, the size of the knapsack, the time given is the total time that was needed to solve the generated knapsack problems for sizes r=15-60 in steps of 5. In all, 50 knapsack problems were solved for each test. Table II compares these 2 methods for finding the optimal solution using the same data sets I, II and III.

Data Set	M	KNAP(1)	KNAP(2)
I	2*Max	6.8	8.45
	$\sum/2$	17.7	15.11
II(a)	2*Max	8.37	7.9
	$\sum/2$	27.94	142.2
II(b)	2*Max	8.15	10.9
	$\sum/2$	29.50	276.0
III(a)	2*Max	8.04	4.56
	$\sum/2$	27.22	41.93
III(b)	2*Max	8.60	7.47
	$\sum/2$	32.38	60.74

Times in Seconds

Table I: Single Solution

Data Set	M	KNAP(1)	KNAP(2)
I	2*Max	6.8	10.7
	$\sum/2$	17.7	16.8
IIa	2*Max	8.37	10.18
	$\sum/2$	27.94	263.94
IIb	2*Max	8.15	11.30
	$\sum/2$	29.50	311.09
IIIa	2*Max	8.04	6.11
	$\sum/2$	29.22	163.49
IIIb	2*Max	8.60	6.28
	$\sum/2$	32.38	70.77

Times in seconds

Table II: Optimal Solution

For the problem of finding a single solution, examination of Table I shows that both methods are often competitive. However, whenever there is a significant difference, as when M=$\sum/2$, KNAP(1) is consistently faster than KNAP(2) often by a factor of 2 or greater. In examining Table II, we again see that KNAP(1) remains far more stable than KNAP(2) and is faster in almost every category. Again for the case when M=$\sum/2$ we see that KNAP(1) is often 5-10 times faster than KNAP(2). In order to see what is happening more concretely, Table III expands the data of Table I for the case: data sets I, IIIa with M=$\sum/2$.

r	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)
15	46.50	73.22	53.24	119.80
20	89.86	209.66	76.56	222.98
25	136.46	242.94	133.12	246.28
30	227.40	775.44	216.32	818.68
35	332.80	975.10	359.42	309.50
40	522.50	183.04	565.76	352.78
45	708.86	5973.76	778.74	4226.56
50	908.56	389.38	1028.36	4156.64
55	1164.80	18720.00	1381.12	858.64
60	1454.32	1865.28	1883.64	835.34
	Data Set I		Data Set IIIa	

Time in milliseconds

Table III: Single Solution Expanded

Table III clearly reveals how KNAP(1) remains stable with increasing r while KNAP(2) experiences wide variation. For r=45, KNAP(2) needs 5900 milliseconds but for r=50 KNAP(2) drops down to 389 milliseconds. We note that this phenomenon occurs i the data is such that a single solution is discovered quickly. Then KNAP(2) abruptly terminates, but one cannot easily determin a priori if such a situation exists. On the average KNAP(1) doe better than KNAP(2). Thus we conclude from Tables I, II and III that KNAP(1) is either as fast and sometimes much faster than KNAP(2).

Let us now look at the behavior of these methods on th data sets IV and V in Table IV. Where the ratios are equal and the data sequential as in set IV, KNAP(2) finds a solution quick- ly and hence works extremely well. However, to find the optiral solution KNAP(2) must do all of the work and the heuristic is no longer helpful. Thus we see that the times for KNAP(2) grow pro- hibitively large. In data set V, where no partition exists, we again see that KNAP(2) works extremely poorly whereas KNAP(1) is very stable. Therefore, both for the straightforward data sets as well as for the specially concocted ones, KNAP(1) with split- ting is almost always faster and often much faster. The empir- ical data combined with the fact that in the worst case we know KNAP(2) to be worse than KNAP(1) combines to make KNAP(1) our first choice.

Data Set	r	KNAP(1)	KNAP(2) (Single Solution)	KNAP(2) (Optimal Solution)
IV	15	16.6	0.0	632.3
	20	33.2	16.6	3394.5
	25	66.5	49.8	14776.3
	30	83.2	33.2	50302.7
	35	166.4	16.6	159760.6
	40	199.6	16.6	> 400000.0
	45	183.0	33.2	
	50	249.6	33.2	
	55	316.1	49.8	
	60	382.7	49.8	
V	15	50.0	2300.0	2912.0
	20	99.8	80437.7	92834.5
	25	200.0	> 600000.0	> 600000.0

Times in milliseconds

Table IV: Single and Optimal Solutions

6. Conclusion

We have considered the problem of finding all combinations of r numbers which sum to M and shown how to reduce the computing time and storage requirements for algorithms which solve this problem by a square root factor. Then we have studied additional improvements such as heuristics and special data structures. The resulting algorithms were then extensively tested and compared. Algorithm 4(b) turned out to be superior in almost every case and often far superior than all of the others. Also it is empirically established that binary encoding as in 4(b) is better than the conventional implicit encoding scheme of algorithm 3(b). Only under special circumstances of the input will algorithm 1(b) even be competitive with 4(b) and these cases are outlined.

Then we have presented the 0/1 knapsack problem and shown how the square root improvement seen before can be directly generalized to its solution. The 2 standard methods, branch and bound and dynamic programming for the knapsack problem (with the inclusion of the square root improvement technique and other heuristics) were programmed and tested. The empirical results showed that the knapsack problem ran 10 times faster than the partition problem. In comparison, KNAP(1) uniformly out-performed its main competitor KNAP(2). Though for certain types of input KNAP(2) is extremely fast for others it is disastrously slow whereas KNAP(1) remains stable as a function of the size of the input. Further for finding ei-

ther the optimal or equivalently all solutions, KNAP(1) was far more efficient than KNAP(2). Therefore, unless one is guaranteed that many solutions exist and that they all will be found early by KNAP(2), KNAP(1), with the splitting technique, is both asymptotically and empirically the better choice.

References

1. Gilmore, P.C. and Gomory, R.E., "Multistage Cutting Stack Problems of Two and More Dimensions", Operations Research 13, pp. 94-120.

2. Gilmore, P.C. and Gomory, R.E., "The Theory and Computation of Knapsack Functions", Operations Research 14, pp. 1045-1074.

3. Hardy, G.H. and Wright, E.M., An Introduction to the Theory of Numbers. Oxford at the Clarendon Press, 4th edition, July 1959.

4. Musser, David R., Algorithms for Polynomial Factorization, PH.D. Thesis, Computer Sciences Department, University of Wisconsin, T.R. #134, September 1971.

5. Nemhauser, G.L. and Ullman, Z., "Discrete Dynamic Programming and Capital Allocation", Management Science, Vol. 15, No. 9, May 1969, pp. 494-505.

6. Nemhauser, G.L. and Garfinkel, R., Integer Programming, John Wiley & Sons, New York, 1972 (to appear).

7. Weingartner, H. Martin and Ness, David N., "Methods for the Solution of the Multi-Dimensional 0/1 Knapsack Problem", Operations Research Vol.15, No.1, Jan.-Feb. 1967, pp. 83-103.

8. Bechenbach, E.F. (ed), Applied Combinatorial Mathematics J. Wiley & Sons, New York, 1964.

9. Cook, Stephen A., "The Complexity of Theorem Proving Procedures", Conference Record of Third ACM Symposium on Theory of Computing, 1970, pp. 151-158.

10. Karp, R., "Reducibility Among Combinatorial Problems", University of California, Berkeley, Tech. Rpt. 3, April 1972.

11. Lawler, E.L. and Bell, M.D. "A Method for Solving Discrete Optimisation Problems", Operations Research 14, 1098-1112.

12. Kolesasar, P.J. (1967), "A Branch and Bound Algorithm for the Knapsack Problem", Management Science 13, 723-735.

13. Bradley, G.H. "Transformation of Integer Programs to Knap sack Problems", Discrete Mathematics, vol 1, No.1, May 1971, pp. 29-45.