Computing Partitions with Applications to the Knapsack Problem

ELLIS HOROWITZ AND SARTAJ SAHNI

Cornell University, Ithaca, New York

ABSTRACT. Given r numbers $s_1, \cdots, s_r$, 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 in terms of both 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. Empirical studies indicate this new algorithm to be generally superior to all previously known algorithms. We then show how this improvement can be incorporated into the more general 0-1 knapsack problem obtaining a square root improvement in the asymptotic behavior. A new branch and search algorithm that is significantly faster than the Greenberg and Hegerich algorithm is also presented. The results of extensive empirical studies comparing these knapsack algorithms are given.

KEY WORDS AND PHRASES: partitions, knapsack problem, dynamic programming, integer optimization

CR CATEGORIES: 5.25, 5.39, 5.42

1. *Introduction*

Given r numbers $s_1, \cdots, s_r$ 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 [7, 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 [2], 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 [15], the motivation arose from capital budgeting problems in which invest-

Authors' present addresses: E. Horowitz, Computer Science Program, University of Southern California, Los Angeles, CA 90007; S. Sahni, Department of CICS, University of Minnesota, Minneapolis, MN 55455.

ment 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 [14]. Much of the early work in the knapsack problem was done by Gilmore and Gomory; see [4, 5]. Finally in [12] 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 [3, 9] 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 algorithms 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 [3, 9]. 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 such an algorithm exists). 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. A new branch and search algorithm for the knapsack problem is also presented. Empirical studies indicate that it is significantly faster than the Greenberg-Hegerich algorithm [6].

2. *Problem Definition*

We begin with the mathematical formulation of our problem.

Definition 1. A *multiset* S is a collection of elements[1] 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 $S_r = (s_1, \cdots, s_r)$ of cardinality r is an r-tuple $\delta = (\delta_1, \delta_2, \cdots, \delta_r)$, where

$$\delta_i \in (0, 1), \quad 1 \le i \le r \quad \text{and} \quad \sum_{i=1}^{r} \delta_i s_i = M. \tag{1}$$

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, 01110, and 11101.

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.

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

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 $S_r = (1, 1, \cdots, 1)$, r even, and $M = r/2$. Then the number of r-tuples δ which satisfy (1) is $\binom{r}{M} = \binom{r}{r/2} = r!/(r/2)!(r/2)!$. Using Stirling's approximation for $r!$ we get

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

Corollary. *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 $S_r = (2^0, 2^1, \cdots, 2^{r-1})$.

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 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. $\cup^*$, $S_{r_1} \cup^* S_{r_2}$ is a multiset such that $x \in S_{r_1} \cup^* S_{r_2}$ 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. $^*\cup^*$, $S_{r_1} {}^*\cup^* S_{r_2}$ is a multiset such that $x \in S_{r_1} {}^*\cup^* S_{r_2}$ under the same conditions as in Definition 6 and in addition the elements of $S_{r_1} {}^*\cup^* S_{r_2}$ are ordered.

Example. If $S_3 = \{1, 2, 1\}$ and $S_4 = \{1, 2, 2, 3\}$, then $S_3 \cup^* S_4 = S_7 = \{1, 2, 1, 1, 2, 2, 3\}$. If $S_3 = \{1, 3, 5\}$ and $S_4 = \{2, 3, 4, 4\}$, then $S_3 {}^*\cup^* S_4 = S_7 = \{1, 2, 3, 3, 4, 4, 5\}$.

Algorithm 1(a). Here we generate all 2^r possible δ's and determine which ones satisfy eq. (1).

```
1. [Initialize] $\delta_r \leftarrow (0, \cdots, 0)$; DO step 2: $2^r - 1$ times;
2. [Find new $\delta$] $\delta \leftarrow \delta + 1$; (binary addition)
        If $\sum_{1 \leq i \leq r} \delta_i s_i = M$ then output $\delta = (\delta_1, \cdots, \delta_r)$.
```

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 severely 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, which is considerably superior to 1(a), and so we shall not concern ourselves with 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 non-recursive version without heuristics can be found in Beckenbach, [1, 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 [15].

Algorithm 1(b). PARTS (s, i, rem, δ) [Backtracking or Branch and Search]. The generation of certain δ's is aborted by using heuristics in steps 1 and 2. It is assumed that elements of $S_r = (s_1, \cdots, s_r)$ are initially ordered $(s_1 \leq s_2 \leq, \cdots, \leq s_r)$. (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 $\delta \cup \{i, i+1, \cdots, r\}$ return.
2. [Try next $s_i$]   If $s + s_i > M$ then return.
                      If $s + s_i = M$ then DO;
                                output $i \cup \delta$;
                                If $i < r$ go to step 4;
                                      else return;
                                END;
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 $(s, i + 1, rem - s_i, \delta)$;
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 submultisets 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). $S_r = (s_1, \cdots, s_r)$. 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 $\sum_{(j|j-1 \in a_2)} s_j = a_1$. The encoding used is $a_2 = \sum_{(j|s_j \in a_1)} 2^{j-1}$.

1. [Initialize] $i \leftarrow 0$; $A \leftarrow \{(0, 0)\}$; $IC \leftarrow 1$; DO step 2 for $i \leftarrow 1, \cdots, r$.
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 2(a) works, consider finding all 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	$\{(0, 0), (1, 2^0), (3, 2^1), (4, 2^0 + 2^1), (4, 2^2), (5, 2^0 + 2^2), (7, 2^1 + 2^2)\}$

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 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 less than or equal to $2k$. The time taken for step 2 when $i = j$ is k and for $i = 1$, $k = 1$. Therefore the total time is less than or equal to $\sum_{i=1}^{r-1} 2^i = O(2^r)$ and the decode time per partition is $O(r)$.

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 two 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 submultisets T, U such that[2]

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

As in 2(a), 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 $i \leftarrow 1, \cdots, t$;
2. $A \leftarrow A$ *U* $\{A + (t_i, IC)\}$; $IC \leftarrow IC + IC$;
3. $i \leftarrow 0$; $B \leftarrow \{(0, 0)\}$; $IC \leftarrow 1$;
   DO step 4 $r - t$ times for $i \leftarrow t + 1, \cdots, r$;
4. $B \leftarrow B$ *U* $\{B + (u_i, IC)\}$; $IC = IC + IC$;
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 $(a_{i_2}, b_{j_2})$.
```

As an example for 2(b) 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 $A + (t_i, IC)$ and $B + (u_i, IC)$ are ordered, the merging necessary to keep A *U* $\{A + (t_i, IC)\}$ and

[2] $\lfloor r/2 \rfloor$ = largest integer less than or equal to $r/2$; $\lceil r/2 \rceil$ = smallest integer greater than or equal to $r/2$.

B *U* $\{B + (u_i, IC)\}$ 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 2. *Algorithm* 2(*b*) *enumerates all the M-partitions of* S_r.

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

Since all partitions less than or equal to M of sets $T = \{s_1, \cdots, s_{r/2}\}$ and $\mathsf{U} = \{s_{r/2+1}, \cdots, s_r\}$ are produced by steps 2 and 4, then for any M-partition δ of S_r there must exist a $\bar{\delta}$ from A and $\underline{\delta}$ from B such that $\delta = \bar{\delta} \cup \underline{\delta}$. Therefore we must show that in step 5 every possible combination of $\bar{\delta} \in A$ and $\underline{\delta} \in B : \delta = \bar{\delta} \cup \underline{\delta}$ 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 $\bar{\delta}$: $\sum_{1 \le j \le r/2} s_j \delta_j = a_i$. Similarly for $\underline{\delta}$ in B. It is sufficient to show that if we are looking at a_i, b_j then every other $\bar{\delta}$ associated with a_k, $k < i$ such that $\bar{\delta} \cup \underline{\delta}$ 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 $p_i, q_i = (\bar{\delta}, \underline{\delta})$ 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 2(b).

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 any 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 $S^{(1)}, \cdots, S^{(r)}$ so that $S^{(i)}$ contains all possible combinations of $s_1, \cdots, s_i$. 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 $J \subset \{1, \cdots, r\}$.

Definition 9. Ordered Union on Sets. S_{r_1} *U* S_{r_2} is a set such that $x \in S_{r_1}$ *U* S_{r_2} iff $x \in S_{r_1}$ or $x \in S_{r_2}$ and the elements are ordered.

Example. $S_4 = \{1, 1, 2, 2\}$. The sumsets are:

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

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

Algorithm **3** (a) (Musser [12]). This works in essentially two stages:
(a) Compute the sumsets of the sets, $S_i = \{s_1, \cdots, s_i\}$, $1 \leq i \leq r$.
(b) Where M appears in $S^{(r)}$ generate all partitions creating M by using the sumsets $S^{(1)}, \cdots, S^{(r-1)}$.

```
Generate sumsets
1. $S^{(0)} \leftarrow \{0\}$;
2. For $j = 1, \cdots, r$, $S^{(j)} \leftarrow S^{(j-1)} {}^*\cup (S^{j-1} + \{s_j\})$. Generation of partitions, using
   $S^{(j)}$. This is a recursive procedure $G(j, n, J)$ initially invoked as $G(1, M, \text{NULL})$.
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 $G(j - 1, n, J)$. END.
```

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

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

PROOF. See [12, 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 $\sum s_i$ if no heuristics are used). So we get another bound on the storage, i.e. $O(rM)$. Thus the storage required is $O(\min\{2^r, rM\})$.

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 $O(rM)$ and also an asymptotic computing time bound of $O(rM)$. 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 affect its general usefulness.

Naturally, if we were writing an algorithm for general use we would avoid bit strings. The maximum storage gain that can be expected, for small M, through the use of bit strings is only a factor of β where β equals the number of bits/word in the machine. Finally, it is often the case that the number of combinations which are generated is considerably less than 2^r. This will be reflected in our implementation 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

TABLE I. Asymptotic Bounds for Partition Algorithms

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

Algorithm	1(a)	1(b)	2(a)	2(b)	3(a)	3(b)	4(a)	4(b)
Time	$r2^r$	2^r	$\max\left\{\begin{matrix}2^r\\ rQ\end{matrix}\right\}$	$\max\{2^{\lceil r/2\rceil}, rQ\}$	$\max\{r^2Q, 2^r\}$	$\max(r^2Q, 2^{r/2})$	$\max(2^r, rQ)$	$\max(2^{r/2}, rQ)$
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	$2r$	$2r$	$(2\frac{1}{2})2^r$	$(4\frac{1}{2})2^{r/2}$	$\min(rM, 2^r)$	$\min(rM, 2^{r/2})$	$(3\frac{1}{2})2^r$	$(6\frac{1}{2})2^{r/2}$

exception that A and B are now sets rather than 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 one pointer, a pointer to the first partition of that sum. 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 I 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.

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 $S = \{s_1, \cdots, s_r\}$ and M:

I. $s_i = i$, $1 \le i \le R$. $M = R, 2R, 3R, R(R+1)/4$.

II. s_i = random numbers in [1,100]. Let $m = \max\{s_i\}$. $M = m, 2m, 3m, \sum s_i/3, \sum s_i/2$.

III. s_i = random numbers in [1,1000]. $M = m, 2m, 3m, \sum s_i/3, \sum s_i/2$.

It should be noted that because of the heuristics used, the time to compute M-partitions for $M = \sum_{1\le i\le r} s_i$ 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 five such tests.

The computing times are reported in Tables II–IV.

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 Tables II–IV 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 III method 2(b) is 10 times faster than 3(b) for M = max, but both methods are about equal for M = 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 method 2(b) runs out of storage on all the data sets whereas 3(b) is able to continue. Therefore method 2(b)

TABLE II. SEQUENTIAL NUMBERS
Times in milliseconds

M	*R*	1(a)	1(b)	2(b)	3(b)	4(a)	4(b)
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	2080.0	1480.9	416.0
	40		1464.3	2396.6	3910.4	3694.1	1015.0
	45		2462.7	[a]		6506 (44)[b]	1847 (44)
						[c]	2163.0
2 max	10	499.2	33.2		99.8	33.2	16.6
	15	21099.5	166.4	16.0 (15)	382.7	216.3	49.9
	20		981.7	166.0	1397.7	965.4	116.4
	25		2912	599.0	6472.9	2645 (23)	216 (23)
	28		5890	[a]	14327	[c]	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	4193.2	665.6 (16)	199.6
	25		13495 (22)	[a]	7720 (22)	[a]	698.8
	27		—		15125 (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	7371.5	[a]	432.6
	25		[c]	[a]	49687		3111.6 (24)

[a] More than 30K words required.
[b] Parentheses indicate actual value of R.
[c] Exceeded time limit.

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.

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:

$$\begin{aligned} &\max \sum_{1 \le i \le r} p_i \delta_i \\ &\text{subject to } \sum_{1 \le i \le i} s_i \delta_i \le M, \end{aligned} \tag{2}$$

TABLE III. Random Numbers (1–100)
Times in milliseconds

M	R	1(a)	1(b)	2(b)	3(b)	4(a)	4(b)
Max	15	21099.5	39.9		36.6	33.2	23.2
	20		76.5		123.1	80.0	29.9
	25		186.3		485.8	176.3	56.5
	30		392.6		1158.9	402.7	96.5
	35		818.6	106.5	2193.7	755.4	163.0
	40		1720.5	123.1	4962.6	1428	216.1
	45		—	252.9	—	[a]	373.6
2 max	15	21099.5	216.2	20	173	180	49.9
	20		1171.4	150	805.3	609	153.1
	25		3120 (24)[b]	316.2	2921.7 (24)	1320 (23)	579.0
	30			858.6	[c]	[a]	1484.2
	35			[a]			1787.2
3 max	15	21099.5	838.6	76.5	329.5	346	76.5
	20		6951.3	472.6	2489.3	812 (17)	236.3
	25			[a]	13367.5	[a]	898.5
	30				[c]		1327.8 (28)
							[a]
Sum/2	15	21099.5	1311.2	552.1	552.1	462.6	83.2
	16		45527.5	1802.4		885	110
	20		[c]	6133.5	6133.5	[a]	372.7
	25			[c]	[c]		2506.2
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			[a]		[a]	1654.0
							2083 (26)

[a] More than 30K words required.
[b] Parentheses indicate actual value of R.
[c] Exceeded time limit.

where δ_i is a nonnegative 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

$$\max \sum_{1 \le i \le r} s_i\delta_i \qquad \text{subject to} \qquad \sum_{1 \le i \le r} s_i\delta_i \le M, \quad \delta_i = 0, 1.$$

Clearly, there is a partition of M in $S = \{s_1, \cdots, s_r\}$ 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 that 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 approach of generating the sumsets and then tracing back to find all existing partitions now seems more attractive. Since we want only *one* solution to the knapsack problem we can entirely eliminate the overhead of maintaining *all* possible partitions as we generate sums (as in 4(b)) and instead use 3(b) where we need only

TABLE IV. RANDOM NUMBERS (1–1000)
Times in milliseconds

M	*R*	1(a)	1(b)	2(b)	3(b)	4(a)	4(b)
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	129.8	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)[a]	386
	50			469.0		2612 (43)	622.3
	55			715.5			725.5
	60			1239.1			1414.6
2 max	15	21099.5	871.4	43.3	33.3	186.3	46.6
	20		3511	89.9	79.9	642	123
	25		12070.7	—	183	1371 (23)	259.6
	30		[b]	266.2	1357.8	1431 (24)	539.1
	35			678.9	3630.9	[c]	1111.5
	40			[c]	8186.9		1244.6 (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	[c]	662.2
	30		[b]	[c]	6659.3		1138.2 (28)
Sum/2	15	21099.5	1364.5	63.2	36.6	502.5	96.5
	20		38022.4	322.8	2539.3	941.8 (16)	382.7
	25		[b]	[c]	13023.6	[c]	1284.6 (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)	282.8
	25		[b]	[c]	499.2	[c]	1045
	30				49770		14144 (26)

[a] Parentheses indicate actual value of *R*.
[b] Exceeded time limit.
[c] More than 30K words required.

trace back once. Furthermore, in order to assure that the splitting procedure takes no longer than $O(2^{r/2})$ we must now keep not only the sums but their associated profits in increasing order. This can clearly be done, for suppose that for some i we have sums $a_i < a_{i+1}$ but profits $p_i \geq p_{i+1}$. 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_j, $i + 2 \leq j \leq r$ which would be added to a_{i+1} giving a sum less than or equal to M can just as well be added to a_i yielding as much profit at less expense. Therefore, the method we first suggest is an adaptation of Algorithm 3(b), the dynamic programming approach where we initially split the 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. Set $F_0(i) = 0$, $0 \leq i \leq M$, and $G_0(i) = 0$, $0 \leq i \leq M$.

Step 2. Compute $F_k(x) = \max\{F_{k-1}(x), F_{k-1}(x - t_k) + PT_k\}$, $1 \leq k \leq \lfloor r/2 \rfloor$; $G_k(x) = \max\{G_{k-1}(x), G_{k-1}(x - u_k) + PU_k\}$, $1 \leq k \leq r - \lfloor r/2 \rfloor$.

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 $x + y \leq M$ 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; i.e. $F_4(x)$ is computed only at those x which can be represented as the sum of a submultiset of the weights t_1, t_2, t_3, and t_4.

It follows immediately from the previous sections that the worst case time and storage requirements for KNAP(1) are $O(\min\{2^{r/2}, rM\})$. We note that previous dynamic programming algorithms for the Knapsack problem (see [5, 13–15]) require $O(\min\{2^r, rM\})$ time and space. Thus for large M our algorithm again represents a square root improvement.

For comparison purposes let us now consider the branch and search algorithms. We will see that their storage requirement will be guaranteed to be linear. This may prove to be the deciding factor, especially for a problem which generates a large number of intermediate possibilities using KNAP(1). We now present two branch and search algorithms that differ only in the heuristic employed by each. KNAP(2) is a new modification of of Kolesar's [10] Branch and Bound Algorithm so that it now requires only linear storage. Kolesar's original algorithm is a mixed breadth-wise and depth-wise tree search requiring exponential storage while KNAP(2) is a depth-first search. KNAP(3) is the popular Greenberg and Hegerich branch and search algorithm. The only difference between KNAP(2) and KNAP(3) is that in KNAP(2) branching is done in order of decreasing profit densities (i.e. decreasing p_i/s_i) while in KNAP(3) the noninteger variable, in the linear program solution to (2) with added constraints, is chosen as the next variable to branch on. In [6] Greenberg and Hegerich presented empirical results indicating that KNAP(3) was better than the Kolesar algorithm by a factor of two or three.

We now present these two branch and search algorithms. It is assumed that $s_i \leq M$, $1 \leq i \leq r$ and that $\sum_1^r s_i > M$ (if this is not the case then we either have a trivial solution or objects can be trivially deleted to obtain an equivalent problem in this form). It is also assumed that $p_i, s_i > 0$, $1 \leq i \leq r$.

```
Algorithm KNAP(2) [Branch and Search].
$p$ = profit associated with this sum,
$i$ = index of next $s_i$ to be processed,
$P$ = maximum profit obtainable,
$\Delta$ = set of $j$ that yield this profit.
  1. [Initialize] Order the $s_i$ in decreasing order of $p_i/s_i$. Set $P, p, \Delta, \delta = 0$ and $i = 1$.
  2. [Test heuristic] Solve the corresponding linear programming problem:
     max $Z = \sum_{k=1}^r p_k\delta_k$
     subject to  $\sum_{k=1}^r s_k\delta_k \leq M,\quad 0 \leq \delta_k \leq 1,\quad 1 \leq k \leq r$
     **If** $P \geq \lfloor Z \rfloor + p$ **go to** 5;
  3. [Put in items until next one that does not fit]
     DO WHILE($s_1 \leq M$ **and** $i \leq r$);
       $M = M - s_i$;
       $p = p + p_i$;
       $\delta_i = 1$;
       $i = i + 1$;
     END;
     [set index of object that does not fit to 0]
     **If** $i \leq r$ **then** DO;
                   $\delta_i = 0$;
                   $i = i + 1$;
                   END;
     [any objects left?]
     **If** $i < r$ **then go to** 2;
     **If** $i = r$ **then go to** 3;
```

```
4. [Save new solution]
   If $P \geq p$ then DO;
                  $P = p$;
                  $\Delta = \delta$;
                  END;
   $i = r$;
5. [Backtrack]
   Find largest $k < i$ for which $\delta_k = 1$.
   If no such $k$ we are done with optimal solution $\Delta$
   Else $M = M + s_k$, $p = p - p_k$, $\delta_k = 0$, $i = k + 1$, go to 2.
```

The linear program of step 2 is simply solved by setting

$$\delta_i , \delta_{i+1} , \cdots , \delta_l = 1, \; \delta_{l+1} = \left(M - \sum_i^l s_k\right) \Big/ s_{l+1}$$

where l is the largest index for which $\sum_i^l s_k \leq M$ (if $l = r$ then just use $\delta_i , \cdots , \delta_r = 1$ with $Z = \sum_i^r p_k$ as the solution).

Algorithm KNAP(3) [Greenberg and Hegerich]. The exact specification of their procedure can be found in [6, pp. 329–330].

Empirical tests to determine the relative efficiencies of KNAP(2) and KNAP(3) were carried out using random p_i, s_i in the range [1, 100] and $M = 100$. This corresponds to the data used by Greenberg and Hegerich [6, p. 331]. The relevant section of the table in [6] together with our computing times for their method on our machine plus our times for KNAP(2) are presented in Table V.

Table V indicates that KNAP(2) is uniformly better than KNAP(3) on this data set. Further results reported later in Table VI will show that KNAP(2) performs several times better than KNAP(3) on the other data sets tested as well.

We note that Algorithms KNAP(2) and KNAP(3) take at most $O(r2^r)$ in computing time and so asymptotically KNAP(1) is certainly superior.

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. The data sets considered are as follows:

I(a):	random weights s_i and random profits p_i; $1 \leq s_i , \; p_i \leq 100$.
I(b):	random weights s_i and random profits p_i ; $1 \leq s_i , \; p_i \leq 1000$.
II(a):	random weights s_i , $1 \leq s_i \leq 100$, $p_i = s_i + 10$.
II(b):	random weights s_i , $1 \leq s_i \leq 1000$, $p_i = s_i + 100$.
III(a):	random profits p_i , $1 \leq p_i \leq 100$, $s_i = p_i + 10$.
III(b):	random profits p_i , $1 \leq p_i \leq 1000$, $s_i = p_i + 100$.
IV:	random p_i , $s_i = p_i$, $1 \leq i \leq r$, and size M such that there is no M-partition in $\{s_i\}$ but $\sum s_i > M$.

For each of the data sets I, II, and III, five problems for each $r = 15, 20, 25, \cdots , 60$ were solved. The knapsack sizes considered were (1) $M = 2*\sum s_i$ and (2) $M = \sum s_i/2$.

Table VI gives the total time needed to solve the fifty problems as described above for each data set. From this table it is clear that for each of the data sets KNAP(2) is significantly better than KNAP(3). KNAP(1) and KNAP(2) are highly competitive, with KNAP(1) generally being superior for $M = \sum/2$. Tables VII–IX expand the computing times given in Table VI for data sets I(a), II(a), and III(a) with $M = \sum/2$. Results are reported only for KNAP(1) and KNAP(2) since from Table VI it is clear that KNAP(3) is comparatively very poor on all the data sets tested. From these tables it becomes evident that KNAP(1) is relatively insensitive to the data and is thus a more stable algorithm. However, it suffers from large storage requirements and so while it may be expected to perform better than KNAP(2), as far as time required is concerned, storage limitations leave KNAP(2) as the only algorithm that may be feasible for certain problems.

TABLE V. MEAN COMPUTING TIMES
Random p_i, $s_i \in [1,100]$; $M = 100$; times in milliseconds

	r									
	15	20	25	30	35	40	45	50	55	60
IBM 360/67 times from [6]	[a]	36.0	[a]	97	[a]	181	[a]	163	[a]	[a]
IBM 360/65 times KNAP(3)	36.4	49.8	49.8	119.8	179.6	156.2	169.8	163.4	269.6	226.4
IBM 360/65 times KNAP(2)	22.8	46.4	43.2	66.6	99.6	80.0	76.4	80.0	83.2	86.8

[a] Times not given in [6].

TABLE VI. TOTAL TIMES FOR 50 PROBLEMS
Times in seconds

Data set	M	KNAP(1)	KNAP(2)	KNAP(3)
I (a)	2 max	6.8	10.45	22.53
	$\sum/2$	17.7	16.91	271
I (b)	2 max	9.45	10.58	43.13
	$\sum/2$	15.94[a]	10.22	109.12
II (a)	2 max	8.37	9.2	157.93
	$\sum/2$	27.94	164.04	>653.2[b]
II (b)	2 max	8.15	13.53	338.57
	$\sum/2$	29.5	305.11	>600[c]
III (a)	2 max	8.04	5.42	10.13
	$\sum/2$	27.22	46.8	>1021.5
III (b)	2 max	8.6	7.6	23.85
	$\sum/2$	32.38	71.1	>900[d]

[a] Time for 49 problems. Last problem could not be solved because of excessive storage required.
[b] First 34 problems solved in this time.
[c] First 28 problems solved in 230 seconds.
[d] First 37 problems solved in 775 seconds.

TABLE VII. MAX TIMES
$M = \sum/2$; times in milliseconds

r	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)
15	33	33	33	83	50	50
20	50	33	83	465	66	83
25	99	99	116	233	133	116
30	169	100	199	3,361	233	150
35	200	216	316	1,564	316	882
40	299	433	499	217	466	166
45	383	615	715	13,063	715	1,397
50	532	616	915	566	1,015	549
55	782	899	1,148	62,699	1,215	932
60	882[a]	2,446	1,464	3,727	1,631	14,643
Data set	I(a)		II(a)		III(a)	

[a] Max of the four problems solved.

Finally in Table X we show what can happen if one tries to use KNAP(2) to solve partition problems. When several partitions exist, one may be found early and KNAP(2) terminates quickly. However, in case there is no partition KNAP(2) gets bogged down. This phenomenon may be expected to occur when the profit densities are equal or nearly equal and the optimal solution does not fill the knapsack. Thus KNAP(1) becomes the more attractive method for this sort of data.

Finally we should like to mention that Ingargiola and Korsh in [8] have developed a clever way to preprocess the input of a knapsack problem before applying the actual algorithm. Given r inputs which are already ordered according to decreasing profit densities ($p_i/s_i \geq p_{i+1}/s_{i+1}$), their method splits the variables into three groups: (a) those which must be contained in the optimal solution, (b) those which cannot be contained in the optimal solution, and (c) those which may or may not end up in the optimal solution. The method takes no more than $O(r^2)$ time and can generally reduce an r-quantity problem to a k-quantity problem where k is significantly smaller than r. After using the Ingargiola-Korsh preprocessor one can then use either a branch and search or a dynamic programming method to solve the remaining knapsack problem.

TABLE VIII. MEAN COMPUTING TIMES
$M = \sum/2$; times in milliseconds

r	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)
15	19.4	16.6	26.2	56.4	36.2	26.6
20	33.2	19.8	65.0	202.8	62.6	53.4
25	69.9	63.0	106.0	196.6	119.4	86.2
30	129.7	76.6	189.2	742.0	199.4	76.6
35	169.7	162.8	295.6	988.6	309.2	665.6
40	282.8	319.6	475.4	133.2	448.6	93.2
45	356.0	366.0	645.2	6,835.8	675.0	981.4
50	475.9	299.4	858.2	429.4	946.0	252.8
55	668.9	586.0	1,104.6	21,682.0	1,141.0	635.6
60	835.3	832.0	1,400.8	925.2	1,500.2	6,103.4
Data set	I(a)		II(a)		III(a)	

TABLE IX. STANDARD DEVIATION
$M = \sum/2$; times in milliseconds

r	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)	KNAP(1)	KNAP(2)
15	15.3	10.5	8.3	17.0	6.5	13.4
20	10.5	6.7	10.8	201.7	6.8	21.9
25	19.5	19.2	13.2	37.1	12.6	30.6
30	12.5	17.2	13.2	1,309.8	20.7	37.3
35	36.5	54.8	12.7	489.5	8.3	145.6
40	32.5	80.5	25.6	57.9	14.8	56.3
45	13.6	223.6	56.8	5,114.0	17.8	375.0
50	48.9	214.4	38.8	140.6	47.3	241.9
55	64.9	278.2	51.1	21,508.5	50.0	291.8
60	28.6	840.5	44.9	1,601.3	81.1	4,775.6
Data set	I(a)		II(a)		III(a)	

TABLE X. DATA SET IV
Equal densities but no M-partition; times in milliseconds

r	KNAP(1)	KNAP(2)
15	50.0	2300
20	99.8	80438
25	200.0	>600,000

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. A new branch and search method is presented which is a modification of the Kolesar branch and bound algorithm, but is guaranteed to require only linear storage. The branch and search algorithm is shown to be superior to the Greenberg-Hegerich algorithm. Then the branch and search method (KNAP(2)) is tested against the dynamic programming method with splitting (KNAP(1)). KNAP(1) usually ran faster, often by a factor of two or better. Though for certain types of input KNAP(2) is extremely fast, for others it is disastrously slow (data set IV), whereas KNAP(1) remains stable as a function of the size of the problem.

REFERENCES

1. Beckenbach, E. F. (Ed). *Applied Combinatorial Mathematics*. Wiley, New York, 1964.
2. Bradley, G. H. Transformation of integer programs to knapsack problems. *Discrete Math 1*, 1 (May 1971), 29–45.
3. Cook, Stephen A. The complexity of theorem proving procedures. Conf. Record of Third ACM Symposium on Theory of Computing, 1970, pp. 151–158.
4. Gilmore, P. C., and Gomory, R. E. Multistage cutting stock problems of two and more dimensions. *Oper. Res. 13* (1965), 94–120.
5. Gilmore, P. C., and Gomory, R. E. The theory and computation of knapsack functions. *Oper. Res. 14* (1966), 1045–1074.
6. Greenberg, H., and Hegerich, R. L. A branch search algorithm for the knapsack problem. *Manage. Sci. 16*, 5 (Jan. 1970), 327–332.
7. Hardy, G. H., and Wright, E. M. *An Introduction to the Theory of Numbers*, 4th ed. Clarendon Press, Oxford, England, 1959.
8. Ingargiola, G. P., and Korsh, J. F. A reduction algorithm for zero-one single knapsack problems. *Manage. Sci.* (to appear).
9. Karp, R. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thather, Eds., Plenum Press, N. Y., 1972, pp. 85–104.
10. Kolesar, P. J. A branch and bound algorithm for the knapsack problem. *Manage. Sci. 13* (May 1967), 723–735.
11. Lawler, E. L., and Bell, M. D. A method for solving discrete optimisation problems. *Oper. Res. 14* (1966), 1098–1112.
12. Musser, David R. Algorithms for polynomial factorization. Ph.D. Th., T.R. #134, Computer Sciences Dep., U. of Wisconsin, Sept. 1971.
13. Nemhauser, G. L., and Ullman, Z. Discrete dynamic programming and capital allocation. *Manage. Sci. 15*, 9 (May 1969), 494–505.
14. Nemhauser, G. L., and Garfinkel, R. *Integer Programming*. Wiley, New York, 1972.
15. Weingartner, H. Martin, and Ness, David N. Methods for the solution of the multi-dimensional 0/1 knapsack problem. *Oper. Res. 15*, 1 (Jan.-Feb. 1967), 83–103.

RECEIVED JULY 1972; REVISED APRIL 1973