# A Complete Anytime Algorithm for Number Partitioning

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90095
korf@cs.ucla.edu

June 27, 1997

**Abstract**

Given a set of numbers, the two-way partitioning problem is to divide them into two subsets, so that the sum of the numbers in each subset are as nearly equal as possible. The problem is NP-complete, and is contained in many scheduling applications. Based on a polynomial-time heuristic due to Karmarkar and Karp, we present a new algorithm, called Complete Karmarkar Karp (CKK), that optimally solves the general number-partitioning problem, and significantly outperforms the best previously-known algorithm for large problem instances. By restricting the numbers to twelve significant digits, CKK can optimally solve two-way partitioning problems of arbitrary size in practice. For numbers with greater precision, CKK first returns the Karmarkar-Karp solution, then continues to find better solutions as time allows. Over seven orders of magnitude improvement in solution quality is obtained in less than an hour of running time. CKK is directly applicable to the subset sum problem, by reducing it to number partitioning. Rather than building a single solution one element at a time, or modifying a complete solution, CKK constructs subsolutions, and combines them together in all possible ways. This approach may be effective for other NP-hard problems as well.

# 1 Introduction and Overview

Most algorithms for combinatorial optimization problems can be divided into two classes: Complete algorithms that are guaranteed to find an optimal solution eventually, but that run in exponential time, and polynomial-time algorithms that only find approximate solutions. Since most of the latter run in low-order polynomial time, they often consume very little time on modern computers, with no way of improving their solutions given more running time. In between these two classes are the anytime algorithms[1], which generally find better solutions the longer they are allowed to run. One of the most common types of anytime algorithms are local search algorithms, which make incremental modifications to existing solutions to try to find better solutions. By maintaining the best solution found so far, the solutions returned by these algorithms can only improve with additional running time. On the other hand, there is no guarantee that additional running time will result in a better solution. Furthermore, if there is no solution of a given quality, these algorithms will never discover that.

In contrast, we present a case study of a different approach to algorithm design for such problems. We start with the best known polynomial-time approximation algorithm for a problem. We then construct a complete algorithm for the problem based on the heuristic approximation. The first solution found by the complete algorithm is the polynomial-time approximation, and as it continues to run it finds better solutions, until it eventually finds and verifies the optimal solution. We refer to such an algorithm as a *complete anytime algorithm*. Furthermore, our algorithm searches a different problem space than is normally searched by either constructive methods or local search techniques, and this problem space is applicable to other combinatorial problems as well. Our case study is the problem of number partitioning, and applies directly to the subset sum problem as well. More importantly, however, we believe that this algorithm design paradigm and/or problem space will be useful on other NP-hard problems as well.

Consider the following very simple scheduling problem. Given two identical machines, a set of jobs, and the time required to process each job on either machine, assign each job to one of the machines, in order to complete all the jobs in the shortest elapsed time. In other words, divide the job processing times into two subsets, so that the sum of the times in each subset are as nearly equal as possible. This is the two-way number partitioning problem, which is NP-complete[3]. The generalization to $k$-way partitioning with $k$ machines is straightforward, with the cost function being the difference between the largest and smallest subset sums. This basic problem is likely to occur as a subproblem in many practical scheduling applications.

For example, consider the set of integers $(4, 5, 6, 7, 8)$. If we divide it into the two subsets $(7, 8)$ and $(4, 5, 6)$, the sum of each subset is 15, and the difference between the subset sums is zero. In addition to being optimal, this is also a perfect partition. If the sum of all the integers is odd, a perfect partition will have a subset difference of one.

We first discuss the best existing algorithms for number partitioning, including several that are limited by their memory requirements to problems of less than 100 elements. We then present an elegant polynomial-time approximation algorithm due to Karmarkar and Karp[10], called set differencing or the KK heuristic, which dramatically outperforms the greedy heuristic. Our main contribution is to extend the KK heuristic to a complete algorithm, which we call Complete Karmarkar Karp (CKK), and which runs in linear space. The first solution returned by CKK is the KK solution, and as the algorithm continues to run it finds better solutions, until it eventually finds and verifies an optimal solution.

We present experimental results comparing CKK to the best previous algorithms for finding optimal solutions to large problem instances. For problem instances with more than 100 numbers, CKK appears to be asymptotically faster than the best existing algorithms, and provides orders of magnitude improvement when perfect partitions exist. Due to the existence of perfect partitions, it is possible in practice to optimally partition arbitrarily large sets of numbers, if the number of significant digits in each number is limited. This limit is currently about twelve decimal digits for two-way partitions, assuming we are willing to wait about an hour for a solution. This is not likely to be a limitation in practice, since no physical quantities are known with higher precision. On the other hand, number partitioning problems that are created by transformations from other NP-Complete problems may result in values with large numbers of digits.

We consider where the hardest problem instances are found, and show the performance of our algorithm on these instances. Next we consider the generalization of CKK to partitioning into more than two subsets. We describe a different search order, called limited discrepancy search, and show that it can improve the performance of CKK. We consider stochastic approaches to number partitioning, which do not find optimal solutions, and then describe the reduction of the subset sum problem to number partitioning.

CKK is the best existing algorithm for large number partitioning problems. Instead of incrementally building a single partition, CKK constructs a large number of subpartitions, and combines them together in all possible ways. This new problem space may be effective for other combinatorial optimization problems as well. Some of this work originally appeared in [12].

# 2 Previous Work

We begin with algorithms that find optimal solutions, but are limited in the size of problems that they can solve, then consider polynomial-time approximation algorithms, and then optimal algorithms for large problem instances.

## 2.1 Brute-Force Search

The most obvious algorithm for finding optimal solutions is to compute all possible subset sums, and return the subset whose sum is closest to one-half of the sum of all the elements. If there are $n$ elements to be partitioned, the time complexity of this algorithm is $O(2^n)$, since there are $2^n$ subsets of an $n$-element set. The space complexity is linear in the number of elements. This approach is impractical for problems larger than 40 elements, however, because of its time complexity.

## 2.2 Horowitz and Sahni

Horowitz and Sahni[6] showed how to dramatically reduce the running time of this algorithm by trading space for time, as follows: Arbitrarily divide the original set of $n$ numbers into two subsets, each containing $n/2$ numbers. For example, if we start with the set $(4, 5, 6, 7, 8)$, we might divide it into the subsets $(4, 6, 8)$ and $(5, 7)$. For each of the two subsets, compute and store all the possible subset sums achievable using numbers from only that subset. This would give us the subset sums $(0, 4, 6, 8, 10, 12, 14, 18)$ and $(0, 5, 7, 12)$. Sort these lists of subset sums. Every subset sum from the original set can be achieved by adding together a subset sum from one of these lists to a subset sum from the other. If a subset sum comes entirely from numbers in one of the lists, the value added from the other list is simply zero, for the subset sum of the null set. Our target value is a subset sum closest to half the sum of all the original numbers, which is 15 in this case.

We maintain a pointer into each of the sorted lists of subset sums. The pointer into one of the lists starts at the smallest element, say 0 on the first list, and only increases, while the pointer into the other list starts at the largest element, 12 in this case, and only decreases. If the sum of the two numbers currently pointed to, $0+12 = 12$ in this case, is less than the target, 15 in this case, increase the pointer that is allowed to increase. This would give us $4 + 12 = 16$ in this case. If the sum of the two numbers pointed to is greater than the target, as it is now, decrease the pointer that is allowed to decrease, giving us $4 + 7 = 11$ in this case. Since 11 is less than 15, we increase the increasing pointer, giving us $6+7 = 13$. Since 13 is still low, the next step gives us $8+7 = 15$, which is exactly the target, and terminates the algorithm. In general, we remember the subset sum closest to the target, and return that if we don't find a perfect partition. Of course, some additional bookkeeping is required to return the actual subsets.

Since each of the two lists of numbers is of length $n/2$, generating all their subset sums takes $O(2^{n/2})$ time. They can be sorted in $O(2^{n/2} \log 2^{n/2})$ or $O(n2^{n/2})$ time. Finally, the two lists of subset sums are scanned in linear time, for an overall time complexity of $O(n2^{n/2})$ time. In fact, this algorithm can be improved by generating the lists of subset sums in sorted order initially, resulting in a running time of $O(2^{n/2})$.

The main drawback of this algorithm is the space needed to store the lists of subset sums. Each list is of size $O(2^{n/2})$, so the overall space complexity is $O(2^{n/2})$. On current machines, this limits us to problems no larger than about $n = 50$. However, for problems of this size or smaller, we have reduced the time complexity from $O(2^n)$ to $O(2^{n/2})$, a very significant reduction.

## 2.3 Schroeppel and Shamir

Schroeppel and Shamir[16] improved on the algorithm of Horowitz and Sahni by reducing its space complexity from $O(2^{n/2})$ to $O(2^{n/4})$, without increasing its asymptotic time complexity. What the Horowitz and Sahni algorithm requires is all possible subset sums, in sorted order, for each half of the original numbers. It accomplishes this by explicitly creating and storing them all. The Schroeppel and Shamir algorithm generates these numbers in order on demand without storing them all.

It works as follows: Arbitrarily divide the original set of numbers into four equal sized sets, called $A$, $B$, $C$, and $D$. We need to generate all possible subset sums of numbers from $A$ and $B$ in sorted order, and similarly for $C$ and $D$. To do this, generate and store all possible subset sums from numbers in $A$, and all possible subset sums from numbers in $B$, and sort both of these lists. Every possible subset sum of numbers from $A$ and $B$ can be represented as the sum of two numbers, one from the subset sums generated by $A$, and the other from the subset sums generated by $B$. Represent such a value as the ordered pair $(a, b)$, where $a$ and $b$ are members of the subset sums from $A$ and $B$ respectively. Note that $a + b_i \leq a + b_j$ if and only if $b_i \leq b_j$.

Initially, create the ordered pairs $(a, b)$ where $a$ ranges over all possible subset sums generated from $A$, and $b$ is the smallest subset sum from elements in $B$, namely zero for the null set. Place these ordered pairs in a heap data structure, ordered by their sum. Thus, the root element will be the smallest such ordered pair. Whenever the next larger subset sum from $A$ and $B$ is required, the root of the heap, containing the element $(a, b_i)$ is returned. Then, this element is replaced in the heap by the pair $(a, b_j)$, where $b_j$ is the next larger element after $b_i$ in the collection of subset sums from $B$. In this way, all the subset sums from $A$ and $B$ can be generated in sorted order. The same algorithm is applied to $C$ and $D$ but in decreasing order of size.

The asymptotic time complexity of this algorithm is $O(2^{n/2})$ since potentially all $2^{n/2}$ subset sums from $A$ and $B$, and also from $C$ and $D$, may have to be generated. While the asymptotic time complexity of the Schroeppel and Shamir algorithm is the same as for the Horowitz and Sahni algorithm, the constant factors are considerably greater, due to the heap operations. The big advantage of the Schroeppel and Shamir algorithm, however, is that its space complexity is only $O(2^{n/4})$ because only the lists of subset sums generated by the numbers in $A$, $B$, $C$, and $D$ must be stored. Since each of these sets of numbers is of size $n/4$, the number of subset sums they each generate is $2^{n/4}$. The heaps are also the same size, for an overall space complexity of $O(2^{n/4})$, compared to $O(2^{n/2})$ for the Horowitz and Sahni algorithm. Thus, in practice, this increases the size of problems that can be solved optimally to from about 50 to 100 numbers. Most current machines don't have sufficient memory to solve larger problems using this algorithm.

## 2.4 Dynamic Programming

There is also one other algorithm for finding optimal solutions that is based on dynamic programming[3]. It requires a bit array $a[i]$ whose size is on the order of the number of achievable subset sums. Assuming integer values, if $a[i]$ is equal to one, that means that the subset sum $i$ is achievable. We describe here a simplified version of the algorithm, albeit not the most efficient. Start with the array initialized to all zeros, and set $a[0] = 1$. Then for each integer $x$ in the original set, scan the array, and for each element $a[i]$ equal to one, set $a[i + x]$ equal to one. Continue until all the numbers are exhausted.

The space complexity of this algorithm is proportional to the number of achievable subset sums. Thus, it is only practical for partitioning problems with a small number of values, or alternatively where the values have limited precision. On most current machines this limit is about 7 decimal digits.

With the exception of the brute-force algorithm described first, all the above algorithms are limited by their space complexities to problems of less than about 100 numbers, or problems where the individual numbers have limited precision. We now to turn to algorithms for solving large problem instances, with numbers of arbitrarily high precision, which are the most difficult to solve. We begin with polynomial-time algorithms that return only approximate solutions, then consider complete versions of these algorithms.

## 2.5 Greedy Heuristic

The obvious greedy heuristic for this problem is to first sort the numbers in decreasing order, and arbitrarily place the largest number in one of two subsets. Then, place each remaining number in the subset with the smaller total sum thus far, until all the numbers are assigned.

For example, given the sorted integers $(8, 7, 6, 5, 4)$, the greedy algorithm would proceed through the following states, where the integers outside the parentheses are the current subset sums: $8, 0(7, 6, 5, 4)$, $8, 7(6, 5, 4)$, $8, 13(5, 4)$, $13, 13(4)$, $13, 17()$, for a final subset difference of 4. Note that the greedy algorithm does not find the optimal solution in this case. The above notation maintains both subset sums, but to find the value of the final difference, we only need to store the difference of the two subset sums. Thus we can rewrite the above trace as: $8(7, 6, 5, 4)$, $1(6, 5, 4)$, $5(5, 4)$, $0(4)$, $4()$. In practice, we would keep track of the actual subsets as well.

This algorithm takes $O(n \log n)$ time to sort the numbers, and $O(n)$ time to assign them, for a time complexity of $O(n \log n)$. It requires $O(n)$ space.

## 2.6 Set Differencing (Karmarkar-Karp Heuristic)

The set differencing method of Karmarkar and Karp[10], also called the KK heuristic, is another polynomial-time approximation algorithm. It also begins by sorting the numbers in decreasing order. At each step, the algorithm commits to placing the two largest numbers in different subsets, while deferring the decision about which subset each will go in. In the above example, if we place 8 in the left subset, and 7 in the right subset, this is equivalent to placing their difference of 1 in the left subset, since we can subtract 7 from both subsets without affecting the final difference. Similarly, placing 8 in the right subset and 7 in the left subset is equivalent to placing 1 in the right subset. The algorithm removes the two largest numbers, computes their difference, and then treats the difference just like any other number to be assigned, inserting it in sorted order in the remaining list of numbers. The algorithm continues removing the two largest numbers, replacing them by their difference in the sorted list, until there is only one number left, which is the value of the final subset difference.

For example, given the sorted integers $(8, 7, 6, 5, 4)$, the 8 and 7 are replaced by their difference of 1, which is inserted in the remaining list, resulting in $(6, 5, 4, 1)$. Next, the 6 and 5 are replaced by their difference of 1, yielding $(4, 1, 1)$. The 4 and 1 are replaced by their difference of 3, giving $(3, 1)$, and finally the difference of these last two numbers is the final subset difference of 2. The KK heuristic also fails to find the optimal partition in this case, but does better than the greedy heuristic.

To compute the actual partition, the algorithm builds a tree, with one node for each original number. Each differencing operation adds an edge between two nodes, to signify that the corresponding numbers must go in different subsets. The resulting graph forms a spanning tree of the original nodes, which is then two-colored to determine the actual subsets, with all the numbers of one color going in one subset.

For example, Figure 1 shows the final tree for the example above. First, replacing 8 and 7 by their difference creates an edge between their nodes. The larger of the two, node 8, represents their difference of 1. Next, replacing 6 and 5 by their difference adds an edge between their nodes, with node 6 representing their difference of 1. We then take the difference of 4 and 1, representing the difference between 7 and 8, and add an edge between node 4 and node 8, since node 8 represents the difference of 1. Since 4 is larger than 1, node 4 represents their difference of 3. Finally, an edge is added between node 4 and node 6, representing 3 and 1 respectively.

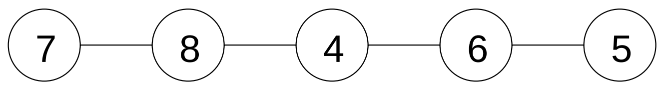


Figure 1: Tree from KK partitioning of (4,5,6,7,8)

In general, the resulting graph forms a spanning tree of the original nodes, since all the numbers must eventually be combined, and $n-1$ edges are created, one for each differencing operation. We then color the nodes of the tree with two colors, so that no two adjacent nodes receive the same color, to get the final partition itself. To two-color a tree, color one node arbitrarily, and then color any node adjacent to a colored node the opposite color. Two-coloring the above tree results in the subsets $(7,4,5)$, and $(8,6)$, whose subset sums are 16 and 14, respectively, for a final partition difference of 2.

The running time of this algorithm is $O(n \log n)$ to sort the $n$ numbers, $O(n \log n)$ for the differencing, since each difference must be inserted into the sorted order, using a heap for example, and finally $O(n)$ to two-color the graph, for an overall time complexity of $O(n \log n)$.

The KK heuristic finds much better solutions on average than the greedy heuristic. Figure 2 compares the two algorithms, partitioning random integers uniformly distributed from 0 to 10 billion. The horizontal axis is the number of values partitioned, and the vertical axis is the difference of the final subset sums, on a logarithmic scale. Each data point in the top two lines is an average of 1000 random problem instances, while those in the bottom line are averages of 100 problems. As the number of values increases, the final difference found by the KK heuristic becomes orders of magnitude smaller than for the greedy heuristic. We also show the optimal solution quality. With 40 or more 10-digit integers, a perfect partition difference of zero or one was found in every case. By about 300 integers, the KK line nearly joins the optimal line, finding a perfect partition almost every time. The greedy line, however, drops only slightly.

The explanation for the difference between the quality of the greedy and

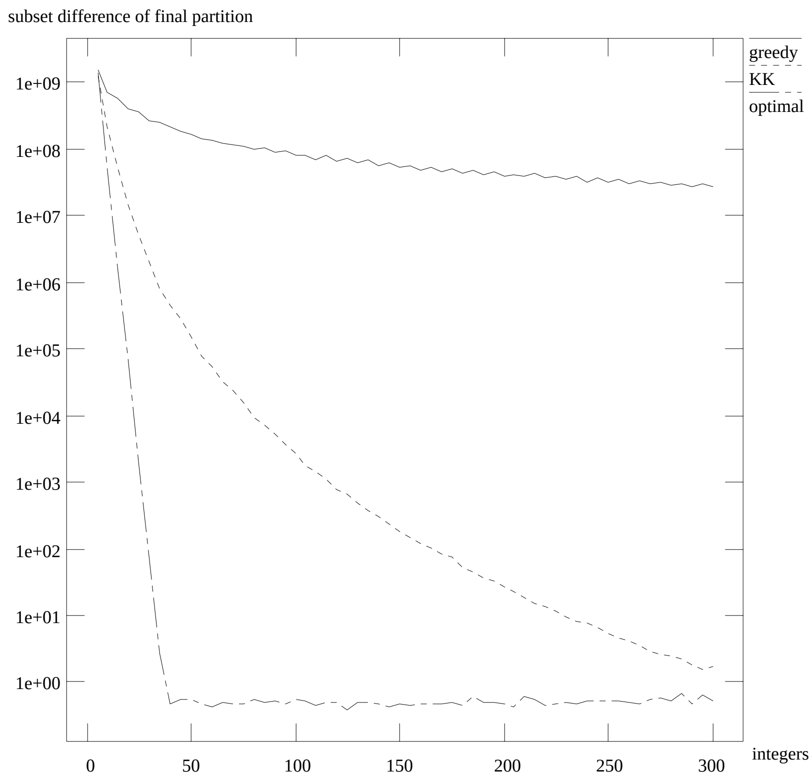


Figure 2: Greedy, KK, and optimal solution quality for 10-digit numbers

KK solutions is quite simple. The difference of the final partition is on the order of the size of the last number to be assigned. For the greedy heuristic, this is the size of the smallest original number. This explains the small improvement with increasing problem size, since the more values we start with, the smaller the smallest of them is likely to be. For $n$ numbers uniformly distributed between zero and one, the greedy method produces a final difference of $O(1/n)$. The sawtooth shape for large numbers of values is due to the fact that successive data points represent alternating odd and even numbers of values, and the even cases result in smaller final differences.

For the KK method, however, repeated differencing operations dramatically reduce the size of the remaining numbers. The more numbers we start with, the more differencing operations, and hence the smaller the size of the last number. Yakir[17] recently confirmed Karmarkar and Karp's conjecture that the value of the final difference is $O(1/n^{\alpha \log n})$, for some constant $\alpha$[10].

## 2.7 Making the Greedy Heuristic Complete

Both these algorithms run in $O(n \log n)$ time and $O(n)$ space, but only find approximate solutions. To find optimal solutions, the obvious algorithm is to search a binary tree, where at each node one branch assigns the next number to one subset, and the other branch assigns it to the other subset. We return the best final difference found during the search, and the actual subsets.

The running time of this algorithm is $O(2^n)$, since we are searching a binary tree of depth $n$, and its space complexity is $O(n)$, since we search it depth-first. There are two ways to prune the tree, however. At any node where the difference between the current subset sums is greater than or equal to the sum of all the remaining unassigned numbers, the remaining numbers are placed in the smaller subset. For example, in the state $15, 0(6, 5, 4)$, the sum of 6, 5, and 4 is no greater than the current subset difference of 15, so the best we can do is to put all the remaining numbers in the smaller subset. This pruning doesn't depend on any solutions found so far, and thus the size of the resulting tree is independent of the order in which it is searched.

Furthermore, if we reach a terminal node whose subset difference is zero or one, representing a perfect partition, then we can terminate the entire search. The above example illustrates this as well, since once we put the remaining integers in the smaller subset, the resulting complete partition has a difference of zero. If a perfect partition exists, then the search order matters, since the sooner we find it, the sooner we can quit. The obvious way to order the search is to always put the next number in the smaller subset so far, before putting it in the larger subset. This algorithm produces the greedy solution first, and continues to search for better solutions, until an optimal solution is eventually found and verified.

Several additional optimizations deserve mention. One is that the first number should only be assigned to one subset, cutting the search space in half. The second is that whenever the current subset sums are equal, the next number should only be assigned to one subset, cutting the remaining subtree in half. Finally, when only one unassigned number remains, it should be assigned only to the smaller subset. Figure 3 shows the resulting binary tree for the integers $(4, 5, 6, 7, 8)$, where the number in front of the parentheses is the difference between the current subset sums, and the numbers below the leaf nodes are the corresponding final partition differences. We refer to this algorithm as the complete greedy algorithm (CGA).

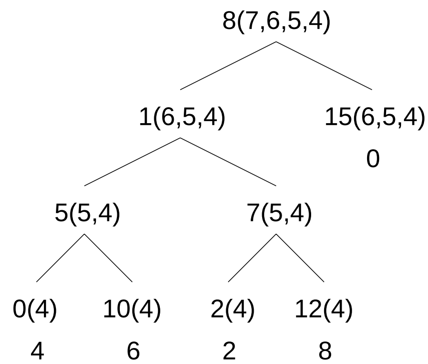


Figure 3: Tree Generated by Complete Greedy Algorithm (CGA)

# 3 Complete Karmarkar-Karp (CKK)

Similar to the extension of the greedy heuristic to a complete algorithm, the main contribution of this paper is to extend the KK heuristic to a complete algorithm. While the idea is extremely simple, it doesn't appear in Karmarkar and Karp's paper[10], nor in any subsequent papers on the problem[15, 8].

At each cycle, the KK heuristic commits to placing the two largest numbers in different subsets, by replacing them with their difference. The only other option is to place them in the same subset, replacing them by their sum. The resulting algorithm, which we call Complete Karmarkar-Karp (CKK), searches a binary tree depth-first from left to right, where each node replaces the two largest remaining numbers. The left branch replaces them by their difference, while the right branch replaces them by their sum. The difference is inserted in sorted order in the remaining list, while the sum is simply added to the head of the list, since it will be the largest element. Thus, the first solution found by CKK is the KK solution, and as it continues to run it finds better solutions, until an optimal solution is found and verified.

In our implementation, the list is maintained in a simple array, with a linear scan for insertion of the differences. While this might seem to take $O(\log n)$ time for each insertion, it amounts to only a constant factor, since most of the nodes in the tree are near the bottom, where the lists are very short. In fact, the average height of a node in a complete binary tree approaches one as the height of the tree goes to infinity. Thus, the worst-case running time of CKK is $O(2^n)$.

Similar pruning rules apply as in CGA, with the largest element playing the role of the current subset difference. In other words, a branch is pruned when the largest element is greater than or equal to the sum of the remaining elements, since the best one can do at that point is to put all the remaining elements is a separate subset from the largest element. Figure 4 shows the resulting binary tree for the integers $(4, 5, 6, 7, 8)$. Note that the tree in Figure 4 is smaller than that in Figure 3, even though both find optimal solutions.

CKK is more efficient than CGA for two reasons. If there is no perfect partition, then both algorithms must search the whole tree. Consider the left subtrees in Figures 3 and 4, where both algorithms place the 8 and 7 in different subsets. This state is represented by $1(6, 5, 4)$ in Figure 3, where 1 is the current subset difference, and by $(6, 5, 4, 1)$ in Figure 4. The distinction between these two nodes is that in the latter case, the difference of 1 is treated like any other number, and inserted in the sorted order, instead of being the current subset difference. Thus, at the next level of the tree, represented by nodes $(4, 1, 1)$ and $(11, 4, 1)$ in Figure 4, the largest number is greater than the sum of the remaining numbers, and these branches are pruned. In CGA, however, the two children of the left subtree, $5(5, 4)$ and $7(5, 4)$ in Figure 3, have to be expanded further. Thus, CKK prunes more of the tree than CGA.

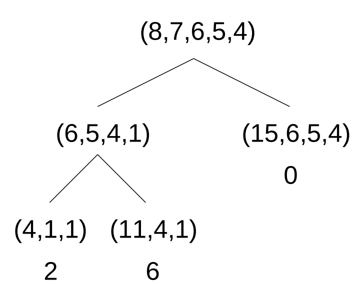


Figure 4: Tree generated by CKK algorithm to partition (4,5,6,7,8)

The second reason that CKK is more efficient occurs when a perfect partition exists. In that case, since the KK heuristic produces better solutions than the greedy heuristic, the CKK algorithm finds better solutions sooner, including the perfect solution. This allows it to terminate the search much earlier than the complete greedy algorithm, on average.

# 4 Experimental Results: Optimal Solutions

We implemented CGA and CKK, both of which find optimal solutions. The results for two-way partitioning are shown in Figure 5. We chose random integers uniformly distributed from 0 to 10 billion, which have ten significant decimal digits of precision. Each data point is the average of 100 random problem instances. The horizontal axis shows the number of integers partitioned, with data points for sets of size 5 to 100, in increments of 5, and from 30 to 40 in increments of 1. The vertical axis shows the number of nodes generated by the two algorithms. The descending line shows the average optimal partition difference on the vertical axis, fortuitously representable on the same scale in this case.

To make the algorithm more efficient, CKK directly computes the optimal partition when there are four numbers left, since the KK heuristic is optimal in that case. CGA continues until there are only two unassigned numbers left before directly computing the optimal partition. To some extent, the choice of what constitutes a terminal node of these search trees is arbitrary and implementation dependent. We set the terminal level of each tree at a point where both algorithms generate roughly the same number of nodes per second, so that a comparison of nodes generated is also a fair comparison of running time. In particular, by our accounting, both algorithms generate two nodes to partition five elements.

Both algorithms were coded in C, and generate about 150 million nodes per minute on a SUN ULTRASPARC model 1 workstation. Thus, the entire vertical axis represents less than seven minutes of computation time.

There are two different regions of this graph, depending on how many values are partitioned. With less than 30 integers, no perfect partitions were found, while with 40 or more integers, a perfect partition was found in every case. The optimal subset difference averages .5 beyond 40 integers, since there are roughly equal numbers of differences of zero and one. CKK dominates CGA over the entire range.

Without a perfect partition, the ratio of the number of nodes generated by CGA to those generated by CKK grows linearly with the number of values.

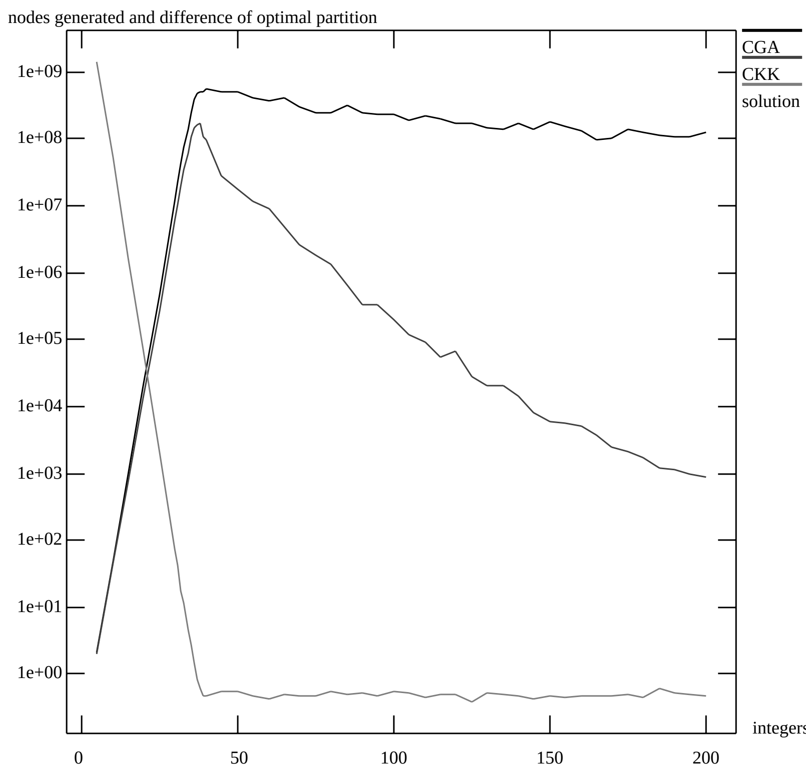


Figure 5: Nodes generated to optimally partition 10-digit integers

This suggests that CKK is asymptotically more efficient than CGA. Without a perfect partition, the performance of both algorithms is independent of the precision of the numbers, assuming that double-precision arithmetic is used throughout. To optimally partition 40 48-bit double-precision integers with CGA requires an average of 56 minutes, while CKK requires an average of only 22 minutes on the same problems, a factor of 2.5 improvement.

The performance improvement is more dramatic when a perfect partition exists. In that case, CKK finds the perfect partition much sooner than CGA, and hence terminates the search earlier. As the problem size increases, the running time of CGA drops gradually, but the running time of CKK drops precipitously, resulting in orders of magnitude improvement. We have run CKK on 10-digit problems up to size 300, at which point the KK heuristic solution is almost always optimal, and the number of nodes generated approaches the number of integers being partitioned. At that point, the running time of CKK is dominated by the $O(n \log n)$ time to find the KK solution.

The intuition behind the observation that large problems are easier to solve than those of intermediate size is quite simple. Given $n$ integers, the number of different subsets of those integers is $2^n$. If the integers range from 0 to $m$, the number of different possible subset sums is less than $nm$, since $nm$ is the maximum possible subset sum. If $m$ is held constant while $n$ is increased, the number of different subsets grows exponentially, while the number of different subset sums grows only linearly. Thus, there must be many subsets with the same subset sum. In particular, the frequency of perfect partitions increases with increasing $n$, making them easier to find.

The data in Figure 5 are for integers with ten decimal digits of precision, to allow running many trials with different numbers of values. Arbitrary-size single problem instances with up to twelve digits of precision can be solved by CKK in a matter of hours. For example, this represents an accuracy of one second in over 30,000 years. Since no physical quantities are known with higher precision, any two-way partitioning problems that arise in practice can be optimally solved, regardless of problem size. While all our experiments were run on uniformly distributed values, we believe that the same results will apply to other naturally occurring distributions as well.

# 5 Where the Hardest Problems Are

Figure 5 shows that for integers of fixed precision, increasing the problem size makes the problem more difficult up to a point, and then easier beyond that point. The reason for the decrease in problem difficulty with increasing problem size is that perfect partitions become more common with increasing problem size. Once a perfect partition is found, the search is terminated.

This phenomenon has been observed in a number of different constraint-satisfaction problems, such as graph coloring and boolean satisfiability, and has been called a *phase transition*[7, 2, 14]. In a constraint-satisfaction problem, the difficulty increases with increasing problem size as long as no solution exists, since the entire problem space must be searched. For some problems however, as problem size increases further, solutions begin to appear more frequently. In that case, the problem gets easier with increasing size, since once any solution is found, the search can be terminated.

This complexity transition also appears in optimization problems[2, 18], as long as there exist optimal solutions that can be recognized as such without comparison to any other solutions. This is the case with number partitioning, where a subset difference of zero or one is always an optimal solution. As another example, when solving a minimization problem with non-negative costs, a zero-cost solution is always optimal.

In most of these problems, the hardest problem instances occur where the probability that an exact or perfect solution exists is one-half. In our experiments, 38% of random sets of 35 10-digit integers had a perfect partition, and 63% of problem instances of size 36 could be partitioned perfectly, suggesting that these should be the hardest problems. In fact, the problem instances that generate the largest median number of nodes are those of size 36. If we look at mean node generations instead, the hardest problems are of size 38, since the outliers have a larger effect on the mean than the median. See [4] for more detail on this complexity transition in number partitioning.

We would like to predict where the hardest problems are, for a given precision of values. To do this, we need to know the value of the optimal subset difference for a given problem class. Karmarkar and Karp et al[11] showed that for a set of independent trials of partitioning $n$ real numbers from zero to one, the median value of the minimum subset difference is $\Theta(\sqrt{n}/2^n)$, or $c\sqrt{n}/2^n$ for some constant $c$. We can use our data to estimate the value of this constant. Since we used integers, we multiply this by the maximum value $m$ of an integer. For example, $m = 10^{10}$ for the experiments in Figure 5. If a majority of a set of problem instances yield a perfect partition, then the median value of the optimal subset differences will almost certainly be 1. In our experiments, this occurred between $n = 35$ and $n = 36$. Solving for $c$ in the equation $c\sqrt{35} \cdot 10^{10}/2^{35} = 1$, gives $c = 1.72$. Using $n = 36$ yields $c = .873$. Thus, we can estimate $c$ as 1. Then, to find the hardest problems for integers up to size $m$, we simply solve for $n$ in the equation $\sqrt{n}m/2^n$.

# 6 Finding Approximate Solutions

While this formula tells us where to find the hardest problems for a given precision of values, the easiest way to generate hard problems is to increase the precision. Most of the work on number partitioning has focussed on problems without perfect partitions. To generate large such problem instances, integers with up to 36 decimal digits have been used[15].

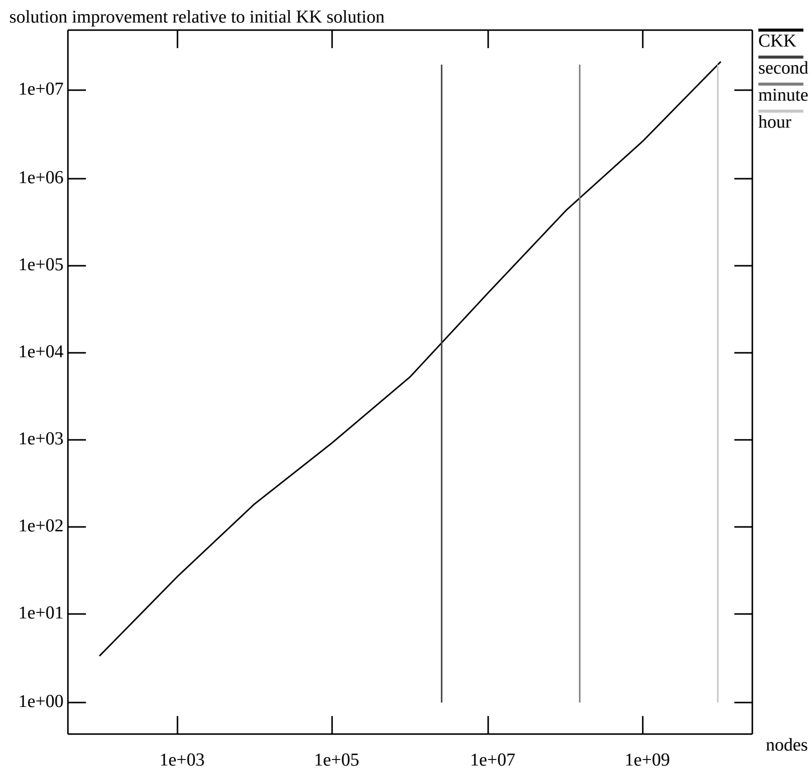


Figure 6: Solution quality relative to KK solution for 50 48-bit integers

For large problems with very high precision values, we must settle for approximate solutions. In that case, we can run CKK for as long as time allows, and return the best solution found. The first solution found is the KK solution, and as CKK continues to run, it finds better solutions. This technique is very effective, with much of the improvement in solution quality occurring early in the run. Figure 6 shows the average improvement as a function of running time for 100 trials of partitioning 50 48-bit integers. The horizontal axis is the number of nodes generated, and the vertical axis is the ratio of the initial KK solution to the best solution found in the given number of node generations, both on a logarithmic scale. The entire horizontal scale represents about an hour of real time, and shows over seven orders of magnitude improvement, relative to the initial KK solution. Four orders of magnitude improvement is obtained in less than a second of running time.

# 7 Multi-Way Partitioning

So far, we have considered partitioning into two subsets. Here we discuss the generalization of these techniques to partitioning into multiple subsets. The task is to partition a set of numbers into $k$ mutually exclusive and collectively exhaustive subsets, so that the difference between the largest and smallest subset sums is minimized. This also minimizes the largest subset sum.

## 7.1 Greedy Algorithm

The generalization of the greedy heuristic to $k$-way partitioning is straightforward. We sort the numbers in decreasing order, and always place the next unassigned number in the subset with the smallest sum so far, until all the numbers have been assigned. Since we can always subtract the smallest subset sum from each of the others without affecting the final partition difference, we only have to maintain $k-1$ values, the normalized differences between each subset sum and the smallest one.

## 7.2 Complete Greedy Algorithm

The complete greedy algorithm also readily generalizes to $k$-way partitioning. Again we sort the numbers in decreasing order, and each number in turn is placed in each of $k$ different subsets, generating a $k$-ary tree. The left-most branch places the next number in the subset with the smallest sum so far, the next branch places it in the next larger subset, etc. Thus, the first solution found is the greedy solution. By never placing a number in more than one empty subset, we avoid generating duplicate partitions that differ only by a permutation of the subsets, and produce all $O(k^n/k!)$ distinct $k$-way partitions of $n$ elements. More generally, by never placing a number in two different subsets with the same subset sum, we avoid generating different partitions that have the same partition difference.

To prune the tree, we use branch-and-bound, and maintain the smallest difference found so far for a complete partition. Given the largest current subset sum, the best we can do is to bring each of the remaining subset sums up to the value of the largest. To see if this is possible, we add all the subset sums except the largest together, add to this the sum of the remaining unassigned numbers, and divide the result by $k-1$. If this quotient is less than the largest subset sum, then the difference between them is the best possible final difference we could achieve. There is no guarantee that we can actually achieve this, since it represents a perfect solution to a $k-1$-way partitioning problem, but it is a lower bound. If the resulting difference is greater than the best complete partition difference found so far, we can prune this branch of the tree, since we can't improve on the existing partition.

Formally, let $s_1, s_2, \ldots, s_k$ be the current subset sums, let $s_1$ be the largest of these, and let $r$ be the sum of the remaining unassigned numbers. If

$$s_1 - \frac{\sum_{i=2}^{k} s_i + r}{k-1}$$

is greater than or equal to zero, then this is the best possible completion of this partial partition. If it is greater than or equal to the best complete partition difference found so far, then the corresponding branch is pruned.

Finally, a perfect partition will have a difference of zero if the sum of the original integers is divisible by $k$, and a difference of one otherwise. Once a perfect partition is found, the algorithm is terminated.

## 7.3 Karmarkar-Karp Heuristic

Karmarkar and Karp also generalized their set differencing method to $k$-way partitioning. A state is represented by a collection of subpartitions, each with $k$ subset sums. The initial numbers are each represented by a subpartition with the number itself in one subset, and the remaining subsets empty. For example, a three-way partitioning of the set $(4, 5, 6, 7, 8)$ would initially be represented by the subpartitions $((8,0,0),(7,0,0),(6,0,0),(5,0,0),(4,0,0))$, which are sorted in decreasing order. The two largest numbers are combined into a single subpartition by putting them in different subsets, resulting in the list $((8,7,0),(6,0,0),(5,0,0),(4,0,0))$. The new subpartition still has the largest subset sum, and hence the next smaller subpartition, $(6,0,0)$, is combined with it by placing the 6 in the smallest subset, resulting in the subpartition $(8,7,6)$. Since we are only interested in the difference between the largest and the smallest subset sums, we normalize by subtracting the smallest sum, 6, from each of the subsets, yielding the subpartition $(2,1,0)$.

This subpartition is then inserted into the remaining sorted list, in decreasing order of largest subset sum, resulting in $((5,0,0)(4,0,0)(2,1,0))$. Again, the two largest subpartitions are combined, yielding $((5,4,0)(2,1,0))$. Finally, these last two subpartitions are combined by merging the largest subset sum with the smallest $(5 + 0)$, the smallest with the largest $(0 + 2)$, and the two medium subset sums together $(4 + 1)$, yielding $(5,5,2)$. Subtracting the smallest from all three subset sums results in the final subpartition of $(3,3,0)$, which has a difference of 3, and happens to be optimal in this case. While we have shown all three subset sums for clarity, our implementation only maintains the two normalized non-zero subset sums for each subpartition.

The actual partition is reconstructed as follows. Each subset sum in each subpartition represents a set of original numbers. Whenever we combine two subset sums, we merge the corresponding sets. For example, in the state $((5,4,0)(2,1,0))$ above, the 5 represents the original 5, the 4 represents the 4, the 2 represents the 8, the 1 represents the 7, and the 0 in $(2,1,0)$ represents the 6, since we subtracted 6 from each of the subset sums in this subpartition. At the last step, we combine the 5 with the 0, resulting in the set $\{5,6\}$, the 4 with the 1, resulting in the set $\{4,7\}$, and the 2 with the 0, resulting in the singleton set $\{8\}$. Thus, the final partition is $(\{8\}\{7,4\}\{6,5\})$, with subset sums of 8, 11, and 11, and a final difference of 3.

## 7.4 Complete Karmarkar-Karp Algorithm

The CKK algorithm also generalizes to multi-way partitioning. Instead of combining subpartitions in only one way, to make the algorithm complete we must combine them together in all possible ways. Again consider three-way partitioning. A particular subpartition represents a commitment to keep the elements in the different subsets separate. There are three cases to consider in combining a pair of subpartitions. In the first case, both subpartitions have only a single non-zero subset sum, say $(A,0,0)$ and $(X,0,0)$. We can combine these in only two different ways, either putting the non-zero elements together, $(X+A,0,0)$, or keeping them apart, $(X,A,0)$. In the second case, one subpartition has one non-zero subset sum and the other has two, say $(A,0,0)$ and $(X,Y,0)$. In this case we can combine them in three different ways, putting the single non-zero element in any of the three subsets of the other subpartition, resulting in the subpartitions $(X,Y,A)$, $(X,Y+A,0)$, and $(X+A,Y,0)$. Finally, both subpartitions can have two non-zero subset sums, say $(A, B, 0)$ and $(X, Y, 0)$. In this case, there are six different ways to combine them: $(X, Y + B, A)$, $(X, Y + A, B)$, $(X + B, Y, A)$, $(X + A, Y, B)$, $(X + B, Y + A, 0)$, and $(X + A, Y + B, 0)$. At each step of the algorithm, the two subpartitions with the largest normalized subset sums are combined in each possible way, and the resulting subpartitions are normalized and inserted in the sorted order of remaining subpartitions. The resulting child nodes are then searched in increasing order of the largest normalized subset sum. Thus, the first solution found is the KK solution.

Figure 7 shows the tree that is generated to optimally partition the integers $(4, 5, 6, 7, 8)$ into three subsets. At the root, each original integer is in its own subpartition. Each subpartition contains only two subset sums, since the third is normalized to zero. At each node, the two subpartitions with the largest subset sums are combined in all possible ways. The nodes at depth $n-1$ are complete partitions, and their subset difference is the largest subset sum, since the smallest is zero. Pruning is discussed below.

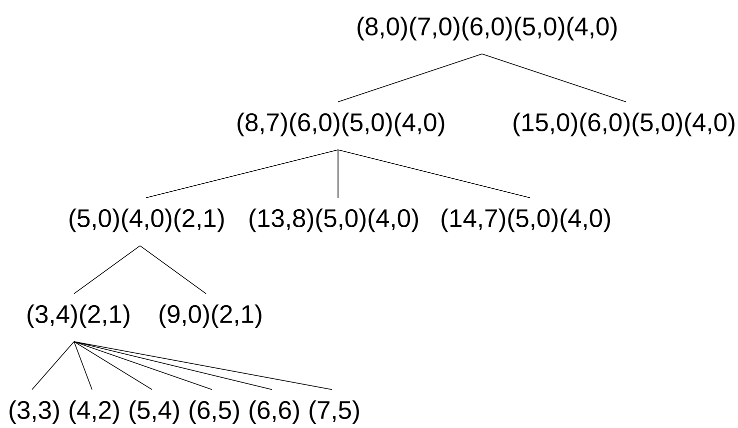


Figure 7: Tree generated to partition 4,5,6,7,8 into three subsets

The resulting tree has depth $n - 1$, and for three-way partitioning, nodes with branching factors two, three, and six, depending on whether the two subpartitions being combined have two, three, or four non-zero subset sums, respectively. The number of leaf nodes, however, is the same as in the complete greedy tree, assuming no normalization or pruning. In other words, if all $k$ subset sums are maintained, and every branch proceeds to depth $n - 1$, there is one leaf node for every distinct partition of the original numbers.

To see this, note that every node in the tree represents a complete collection of subpartitions. A single subpartition is a division of some of the original numbers into $k$ different subsets, and represents a commitment to keep the numbers in different subsets apart in every partition arising from it. A subset sum of zero represents an empty set, since we don't normalize. Each node takes the two subpartitions with the largest subset sums, and combines them together in all possible ways. For example, the root node in Figure 7 combines the two subpartitions containing just the 8 and the 7 into a single subpartition in two different ways. Either the 8 and 7 will go into different subsets, or in the same subset. Since all the original numbers must eventually be combined into a single partition of $k$ subsets, combining the subpartitions pairwise in all possible ways guarantees that every partition will eventually be generated at some leaf node of the tree.

To see that no partition is generated more than once, note that each node combines only two subpartitions, the ones with the largest subset sums. The new subpartitions that result are all different, representing different choices about assigning the numbers in the combined subpartitions to the same or different subsets. For example, consider the root of the tree in Figure 7. Its left child puts the 8 and 7 in different subsets, and they will stay in different subsets in every partition generated below that node, since numbers in different subsets of the same subpartition are never combined. Conversely, the right child of the root puts the 8 and 7 in the same subset, and they will stay in the same subset in every partition below that node. Thus, the partitions generated below the left and right children are completely disjoint. This is true in general, and each distinct partition appears only once.

Normalization can reduce the size of the tree. For example, if the two smallest subsets in a subpartition have the same sum, after normalization a zero will occur in the combined subpartition, which does not represent an empty subset. When another value is combined with this subpartition, it will only be placed in one of the smallest subsets. Since the two subsets have the same sum, which subset a new value is placed in has no affect on the final partition difference, even though it may generate different partitions. This normalization savings also applies to the complete greedy algorithm as well.

Pruning the CKK tree is similar to pruning in the complete greedy algorithm. We add up all the subset sums except for the largest one, and evenly divide this total among $k-1$ subsets. If the difference between the largest subset sum and this quotient is greater than or equal to the best complete partition difference found so far, we can prune this branch, since this is the best we could possibly do below that node. For example, consider the sub-partition $(13, 8)(5, 0)(4, 0)$, near the middle of Figure 7. The largest subset sum is 13, and the sum of the remaining values is $8+5+4 = 17$. If we divide 17 among the two remaining subsets, the best we could do is to have 9 in one of the subsets, and 8 in the other. The best possible partition difference would then be $13 - 8 = 5$. Since the leftmost leaf node has a partition difference of only 3, we can prune this node. Finally, a complete partition with a difference of zero or one is perfect, and terminates the search.

## 7.5 Experimental Results

We implemented both CKK and CGA for three-way partitioning, using integers uniformly distributed from zero to $100,000$. Figure 8 shows the results, in the same format as Figure 5. Each data point is an average of 100 random trials. The horizontal axis is the number of integers being partitioned, and the vertical axis for the CGA and CKK algorithms is the number of nodes generated. We also show the value of the optimal subset difference on the same scale, indicating that the hardest problems occur where the probability of a perfect partition is about one-half. The results are very similar to those for two-way partitioning. Namely, CKK appears asymptotically more efficient than CGA when no perfect partition exists, and is orders of magnitude more efficient when there are perfect partitions.

While the constant factors for CKK and CGA are similar for two-way partitioning, the three-way version of CKK is more complex. Our three-way implementation of CKK runs about 33% slower per node generation than CGA. While this reduces the absolute performance of CKK, it still appears asymptotically more efficient than CGA, and runs faster in practice.

In order to run large numbers of three-way partitioning problems of different sizes, we used integers with five significant decimal digits. Single instances of arbitrary size with six digits of precision can be solved in practice, however. While two-way partitioning problems with up to twelve digits can be optimally solved, three-way partitioning is computationally more difficult, since the number of $k$-way partitions is $O(k^n/k!)$.

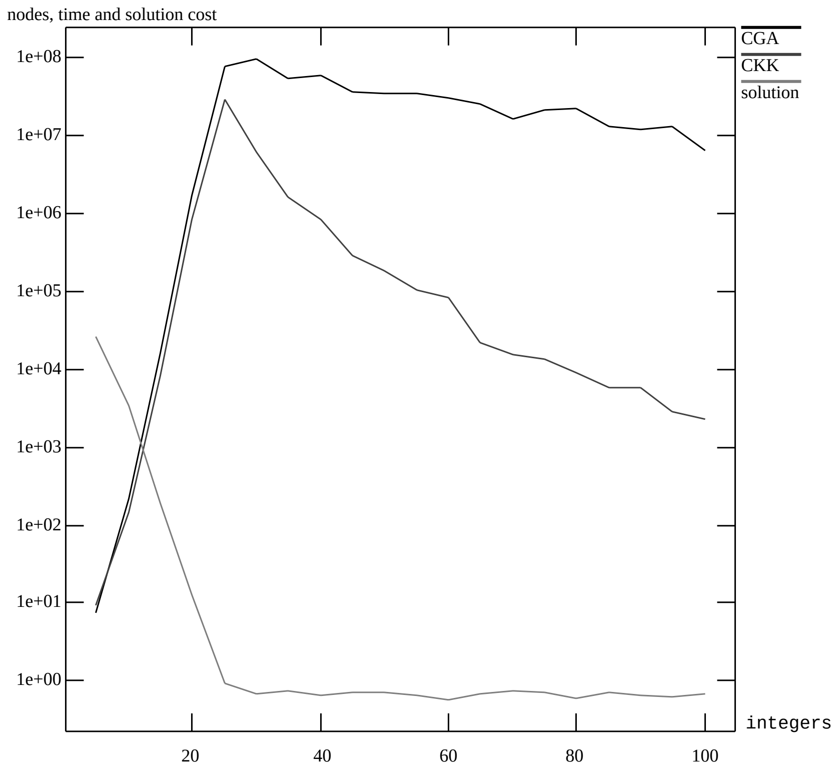


Figure 8: Nodes generated to partition 5-digit integers into three subsets

# 8 Limited Discrepancy Search

So far, all the trees generated by CKK and CGA have been searched depth-first, from left to right. The leftmost branch at each node is the branch recommended by the heuristic, either the greedy heuristic of placing the next number in the smallest subset so far, or the KK heuristic of separating the two largest numbers in different subsets. An alternative search strategy, which has the same linear space requirement as depth-first search, is called *limited discrepancy search*[5]. Limited discrepancy search (LDS) is based on the idea that in a heuristically ordered search tree, a left branch is preferable to a right branch. Instead of searching the tree left to right, LDS searches the paths of the tree in increasing order of the number of right branches, or discrepancies with the heuristic recommendation. The first path it generates is the leftmost, as in depth-first search. Then, however, it searches all those paths with one right branch in them, followed by those paths with two right branches, etc. This results in a different search order than depth-first search. For example, the path that goes right from the root and then left at every remaining branch is generated during the first iteration of LDS, rather than after the entire left subtree has been searched as in depth-first search.

Searching the tree in this order involves some overhead relative to depth-first search[13]. Thus, in the cases where no perfect partition exists, and the entire tree must be searched, depth-first search is preferable. However, in cases where there is a perfect partition, LDS often finds it faster than depth-first search, and hence is more efficient. Furthermore, in those difficult problem instances where finding an optimal solution is not practical, LDS oftens finds better solutions faster than depth-first search. See [13] for some experimental results with this algorithm on number partitioning.

# 9 Stochastic Approaches

There have been at least three studies applying stochastic algorithms to number partitioning, none of which can guarantee optimal solutions. Johnson et al[8] applied simulated annealing to the problem, but found that it was not competitive with even the Karmarkar-Karp heuristic solution. Ruml et al[15] applied various stochastic algorithms to some novel encodings of the problem, but their best results outperform the KK solution by only three orders of magnitude, compared to the seven orders of magnitude CKK achieves in less than an hour. Jones and Beltramo[9] applied genetic algorithms to the problem, but don't mention the Karmarkar-Karp heuristic. Their technique fails to find an optimal solution to the single problem instance they ran, while the KK solution to this instance is optimal.

# 10 Subset Sum Problem

Given a set of integers, and a constant $c$, the subset sum problem is to find a subset of the integers whose sum is exactly $c$. We can reduce this problem to the two-way partition problem, and hence apply CKK to this problem as well. Let $s$ be the sum of all the integers. If $c$ is less than $s/2$, use $s-c$ for $c$.

Add a new integer $d$ such that $(s+d)/2 = c$, or $d = 2c - s$. If this augmented set can be perfectly partitioned with a difference of zero, then the subset of the perfect partition that does not contain $d$ is a subset of the original numbers whose sum is exactly $c$, and hence a solution to the subset sum problem. Conversely, if this augmented set cannot be perfectly partitioned, then there is no subset of the original numbers that sum to exactly $c$, and hence no solution to the subset sum problem.

# 11 Generalization: A New Problem Space for Combinatorial Optimization

Most algorithms for combinatorial problems search one of two different problem spaces. The first, searched by CGA for example, is a constructive space where each node is a partial solution, and the operation is to make an assignment to another element of the problem, in this case assign a new number to one of the subsets. In the second space, typically searched by the stochastic methods described above, each node is a complete solution, and the operators are to change one complete solution into the other. The space searched by the CKK algorithm is neither of these, however, and represents a new problem space which is applicable to other combinatorial problems as well. While it appears to be a constructive space for number partitioning, the operators are not to assign a number to a subset, but rather commit to either separating two numbers, or combining them together into the same subset.

To see the generalization of this idea, note that a solution to a two-way number partitioning problem can be represented as a bit string, with one bit for each number, the value of which specifies which subset it is assigned to. While CGA successively assigns the values of these bits one at a time, CKK decides at each point that two bits will either have the same value or different values, without making an explicit assignment. When $n-1$ such decisions have been made, for every pair of bits we know whether they have the same or different values, and only two possible complete solutions remain, which are complements of each other, and equivalent in this case.

The solutions to many other combinatorial problems can be represented as bit strings as well, and the same space could be searched. For example, the graph bisection problem is to partition the nodes of a graph into two equal size subsets, so that the number of edges that go from a node in one subset to a node in the other is minimized. Clearly any solution can be represented by a bit string, with one bit for each node, and the space searched by CKK could be searched here as well. As another example, consider the problem of boolean satisfiability. Any solution can be represented as a bit string, with a bit for each variable. Again, we could search a space where at each point we decide that two variables will have the same or different values. We leave for further research the question of whether searching such a space is worthwhile in these other problems, and merely claim that this approach suggests an entirely new problem space for a wide variety of combinatorial optimization problems.

# 12 Summary and Conclusions

The main contribution of this paper is to extend an elegant and effective polynomial-time approximation algorithm for number partitioning, due to Karmarkar and Karp, to a complete algorithm, CKK. The first solution it finds is the KK heuristic solution, and as it continues to run it finds better solutions, until it eventually finds and verifies an optimal solution. For problems with less than 100 numbers, or for problems where the numbers have low precision, there exist more efficient algorithms. However, for large problem instances with high precision, which are the most difficult to solve, CKK is more efficient than the complete greedy algorithm (CGA), the best existing alternative. When a perfect partition exists, CKK outperforms CGA by orders of magnitude. We showed results for both two-way and three-way partitioning. In practice, two-way partitioning problems of arbitrary size can be solved if the numbers are restricted to no more than twelve significant digits of precision, while arbitrary-sized three-way partitioning problems can be optimally solved with six significant digits. For large problems with higher precision values, CKK can be run as long as time is available, returning the best solution found when time runs out.

What contribution does this work make beyond the specific problem of number partitioning? First, CKK is directly applicable to the subset sum problem, and may apply to other related problems as well. Secondly, it presents an example of an approach that may be effective on other combinatorial problems. Namely, we took a good polynomial-time approximation algorithm, and made it complete, so that the first solution found is the approximation, and then better solutions are found as long as the algorithm continues to run, eventually finding and verifying an optimal solution. We refer to such an algorithm as a complete anytime algorithm. Thirdly, it represents an example of a new problem space for combinatorial optimization problems. Most existing algorithms either construct a solution to a problem incrementally, adding one element at a time to a single partial solution, or perturb a complete solution into another complete solution. The former is the case for CGA, and the latter is the approach taken by most stochastic algorithms. The CKK algorithm, on the other hand, constructs a large number of partial solutions, and combines them together in all possible ways. In the case of number partitioning, this latter strategy is much more effective, and may be for other problems as well.

## 13 Acknowledgements

Thanks to Wheeler Ruml for introducing me to number partitioning, and the Karmarkar-Karp heuristic. Thanks to Wheeler, Ken Boese, Alex Fukunaga, and Andrew Kahng for helpful discussions concerning this research, and to Pierre Hasenfratz for comments on an earlier draft. This work was supported by NSF Grant IRI-9119825, and a grant from Rockwell International.

## References

[1] Boddy, M., and T. Dean, Solving time-dependent planning problems, in *Proceedings of the International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, August, 1989, pp. 979-984.

[2] Cheeseman, P., B. Kanefsky, and W.M. Taylor, Where the *really* hard problems are, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-91)*, Sydney, Australia, Aug. 1991, pp. 331-337.

[3] Garey, M.R., and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[4] Gent, I.P., and T. Walsh, The number partition phase transition, Technical Report RR-95-185, Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, Scotland, May, 1995.

[5] Harvey, W.D., and M.L. Ginsberg, Limited discrepancy search, in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995, pp. 607-613.

[6] Horowitz, E., and S. Sahni, Computing Partitions with Applications to the Knapsack Problem, *Journal of the A.C.M.*, Vol. 21, No. 2, April 1974, pp. 277-292.

[7] Huberman, B., and T. Hogg, Phase transitions in artificial intelligence systems, *Artificial Intelligence*, Vol. 33, No. 2, Oct. 1987, pp. 155-171.

[8] Johnson, D.S., C.R. Aragon, L.A. McGeoch, and C. Schevon, Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning, *Operations Research*, Vol. 39, No. 3, 1991, pp. 378-406.

[9] Jones, D.R., and M.A. Beltramo, Solving partitioning problems with genetic algorithms, in Belew, R.K., and L.B. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, Ca., 1991, pp. 442-449.

[10] Karmarkar, N., and R.M. Karp, The differencing method of set partitioning, Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, Ca., 1982.

[11] Karmarkar, N., R.M. Karp, G.S. Lueker, and A.M. Odlyzko, Probabilistic analysis of optimum partitions, Journal of Applied Probability, Vol. 23, 1986, pp. 626-645.

[12] Korf, R.E., From approximate to optimal solutions: A case study of number partitioning, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995, pp. 266-272.

[13] Korf, R.E., Improved limited discrepancy search, *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, Aug. 1996, pp. 286-291.

[14] Mitchell, D., B. Selman, and H. Levesque, Hard and easy distributions of SAT problems, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-94)*, San Jose, Ca., July 1992, pp. 459-465.

[15] Ruml, W., J.T. Ngo, J. Marks, S. Shieber, Easily searched encodings for number partitioning, to appear, *Journal of Optimization Theory and Applications*, vol 89, number 2, 1996.

[16] Schroeppel, R., and A. Shamir, A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-Complete Problems, *SIAM Journal of Computing*, Vol. 10, No. 3, Aug. 1981, pp. 456-464.

[17] Yakir, B., The differencing algorithm LDM for partitioning: A proof of a conjecture of Karmarkar and Karp, *Mathematics of Operations Research*, Vol. 21, 1996, pp. 85-99.

[18] Zhang, W., and R.E. Korf, A study of complexity transitions on the asymmetric traveling salesman problem, *Artificial Intelligence*, Vol. 81, March 1996, pp. 223-239.