# ESTIMATING THE WORK IN INTEGER PARTITIONING

*The authors developed an approach for estimating the work done by the best-known integer-partitioning methods for a given data set, and it works without running the partitioning program itself. This method can also estimate the number of perfect partitions in a given data set.*

Dividing a given set of positive numbers into two groups so that the sums of the numbers in each group differ by the smallest amount possible is called the *integer-partitioning problem*. In a perfect partition, the sums of the elements in each group differ by only zero or one. In a balanced partition, the number of elements in each group, the cardinalities, differ by only zero or one.

Finding such a partition is useful in multiprocessor load balancing,[1] certain thermodynamics problems,[2] and theoretical computer science (as an example of a provably hard problem).[3] A good estimate could help a processor decide how many resources it could allocate to solve a large problem. For example, we know that partitioning is a hard problem, and we might want to load-balance a multiprocessor. However, we would not want to do this if it would take a week to run a partitioning program but only a few hours to run the actual software programs.

1521-9615/03/$19.00 © 2003 US Gov't Work Not Protected by US Copyright
Copublished by the IEEE CS and the AIP

APRIL K. ANDREAS
*Southern Methodist University*
ISABEL BEICHL
*National Institute of Standards and Technology*

The integer-partitioning problem is known to be NP-complete,[3] and as a result, researchers have developed heuristics that provide minimized partition[4] and cardinality[5] differences over time. What is not clear, however, is how long these heuristics must be run to find an acceptable answer.

We have developed a method to estimate the amount of work required to find an optimal solution to the integer-partitioning problem. We have also developed a way to estimate how many balanced, perfect partitions exist. The key to our method is a variation of a technique Donald Knuth originally developed to estimate the size of the backtrack trees.[6]

## Why Is Partitioning Hard?

A naïve partitioning method looks at all the ways to assign elements to subsets. If we think of the integer-partitioning problem as assigning 0's and 1's to list elements, with a 0 meaning that element is assigned to the first set and a 1 meaning that the element is assigned to the second set, there are $2^n$ possible ways to assign the original list elements. With the constraint that the partition be balanced, only

$$\binom{n}{\lfloor n/2 \rfloor}$$

need be considered.

However, there are faster methods than the naïve method. Narendra Karmarkar and Richard Karp,[7] for example, have a set-differencing heuristic (the KK method) for partitioning that works in polynomial time. The KK heuristic sorts numbers in descending order. At each step, the two largest elements are placed in different subsets, although the decision on which particular subset each will go in is not made immediately. (We will address the question of which subset in the section "How to Get a Partition.") Putting the two elements in different groups is essentially equivalent to replacing the two numbers in the list by their difference. The list is then resorted with the new difference replacing the two numbers used, and the process continues until there is only one element left in the list. This last element is the final difference between the two subsets. Figure 1 shows this method used on the list 8, 7, 6, 5, 4. The KK heuristic is not guaranteed to give a good partition, but it does give a partition and a better one than a random one.

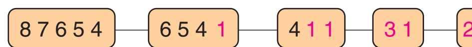



**Figure 1. The Karmarkar–Karp (KK) partitioning method on the list 8, 7, 6, 5, 4. The pink elements are differences of the first two elements from the previous steps.**

### The CKK Method

Richard Korf extended this heuristic to a complete anytime algorithm[4]—called the complete Karmarkar–Karp, or CKK method—that finds the KK partition in $n$ steps and then looks for better answers by also examining the sums of the numbers and not only their differences. The CKK gets better answers the longer it is allowed to run, but it could possibly run a long time, so it is reasonable to stop after a predetermined amount of computing has been done or until a perfect partition is found. Stephan Mertens's version of Korf's algorithm includes, among other improvements, constraints for finding a balanced partition.[5]

More specifically, CKK searches a binary tree from left to right where each node replaces the two largest numbers with either their difference (the left branch) or their sum (the right branch) and then resorts. There are also pruning rules if it is clear that an improved partition is not possible. To best follow the CKK partitioning algorithm, consider a sorted list of length six: 6, 5, 5, 4, 3, 3. The first two elements, 6 and 5, are removed. The two options available are either to put the two elements in different groups or to put them in the same group. Essentially, this is equivalent to replacing the two numbers by their difference, 1, or by their sum, 11, respectively. Either decision results in a new list of length five: 5, 4, 3, 3, 1 or 11, 5, 4, 3, 3.

Keeping track of the partitions' cardinality differences follows a similar approach to the one Mertens originally developed.[2] We now keep track of pairs of numbers—namely, the numbers themselves and their cardinality. The cardinalities of the original numbers are all initialized to 1. As list elements are put in separate or same groups, their cardinalities are subtracted or added accordingly. If elements 6 and 5 were put in the same group, their sum, 11, and the sum of their cardinalities, 2, would replace them in a new list. If they were put in different groups, their difference, 1, and the difference between their cardinalities, 0, would replace them. In Mertens's algorithm, however, the list is not resorted until its length is only half that of the original's length.

This process continues, producing a binary tree that finishes with 32 leaves that are each a list of length 1. The numbers in each of these lists represents the partition difference that we would get by distributing the original numbers into groups indicated by the path taken down the tree and the resulting cardinality difference. The pink boxes in Figure 2 indicate the locations of balanced, perfect partitions.

CKK excels because it can work faster; it knows where it *doesn't* have to look. For example, consider a list of length 5—with the numbers 9, 7, 6, 5, 3—and investigate balanced partitions that contain 9 and 7 in the same group. Remember, to have a balanced, perfect partition, each group can have no more than three elements. This leaves four options, each represented by a row (see Table 1).

In looking at the table, we see no balanced, perfect partitions, which should have been obvious from the start. Because 9 + 7 = 16 and 6 + 5 + 3 = 14, the smallest possible partition difference that includes 9 and 7 in the same group is 2. CKK does not need to waste time investigating these four possibilities; it can eliminate them immediately and move on.

This suggests two pruning conditions.[2,4] Let $(x_1, x_2, \dots x_i)$ be the list to be sorted, and let $m_i$ be

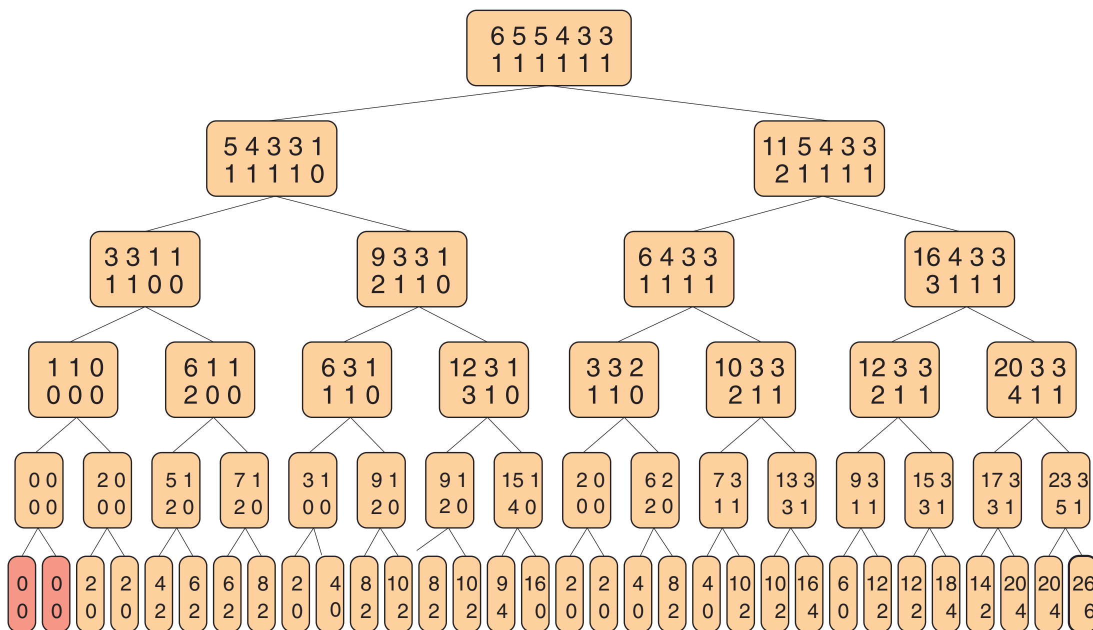



**Figure 2. The tree created by the complete Karmarkar–Karp (CKK) algorithm.**

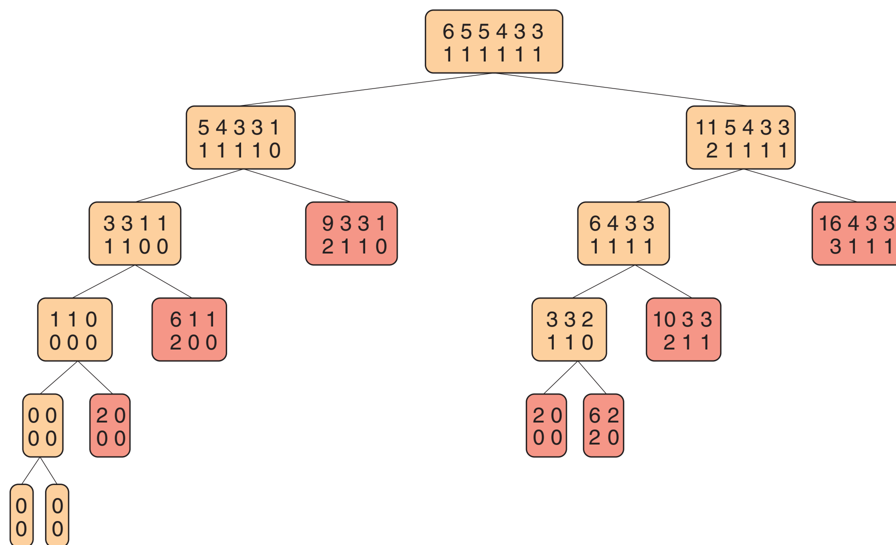



**Figure 3. The tree created by the complete Karmarkar–Karp (CKK) algorithm using pruning methods on a six-element list.**

the cardinality associated with element $x_i$. When finding a perfect, balanced partition, we can eliminate lists from consideration if either of the following conditions apply:

$$2 \bullet \max x_i - \Sigma x_i > 1$$

or

$$2 \bullet \max |m_i| - \Sigma |m_i| > 1.$$

By applying the pruning conditions, we can make the previous tree significantly shorter. The pink boxes in Figure 3 indicate nodes along the tree where the pruning conditions discussed earlier make looking at any nodes below that one unnecessary. Such pruning eliminates 47 possi-

ble steps, which is already a 75 percent reduction in work.

CKK could possibly find a perfect partition even sooner. Because it follows a postorder traversal of this tree, it will actually encounter the first 0-0 node in only six steps—thus it has a perfect answer, so it can stop.

We have used a modified CKK method that uses part of Mertens's method for finding balanced partitions. It is this method whose work we estimate. First, we always sort the list after every iteration. Second, we do not look to find anything but a perfect partition. That is, if pruning rules indicate that no perfect partition is possible, we don't go there.

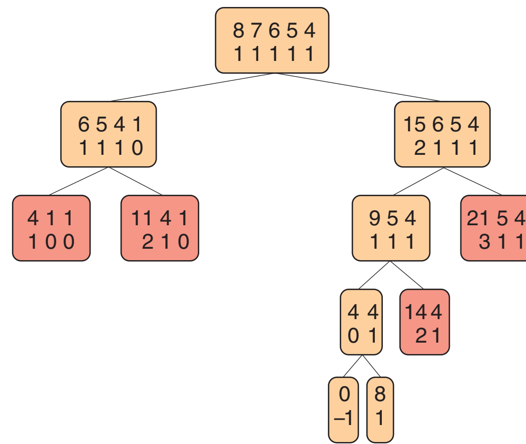



**Figure 4. The tree created by the CKK algorithm using pruning methods on a five-element list.**

**Table 1. Examples of different ways to partition 9, 7, 6, 5, 3 given that 9 and 7 are in the same set.**

| Group 1 | Group 2 | Difference |
|---|---|---|
| 9, 7 | 6, 5, 3 | 2 |
| 9, 7, 6 | 5, 3 | 14 |
| 9, 7, 5 | 6, 3 | 12 |
| 9, 7, 3 | 6, 5 | 8 |

**Table 2. Example of recovering the list partition.**

| Group A | Group B |
|---|---|
| ~~0~~ | |
| ~~4~~ | 4 |
| ~~9~~ | 4, 5 |
| ~~15~~ | 4, 5, 6 |
| 8, 7 | 4, 5, 6 |

## How to Get a Partition

Recovering the actual partition from the tree is a simple game of replacement. Consider the tree in Figure 4. (Recovering the partition is not necessary for understanding the rest of the article, but we thought it was interesting.)

We start creating our two lists by leaving one empty and placing our known partition difference, 0, in the other. Looking up to the next level of the tree, we get a final partition difference of 0 by subtracting 4 from 4. As such, we can maintain our partition difference by removing the 0 and placing a 4 in each of the two groups. We know that one of the 4's was part of the original list but the other came from subtracting 5 from 9. Hence, we can remove one of the 4's and instead place the 9 where the 4 had been and the 5 in the other group. The sum of each group is 9, and the partition difference is still 0. We can continue this work up the tree until we have only original elements in our two groups (see Table 2; note that because 15 came from adding 8 and 7, we must replace 15 by placing 8 and 7 in the same group to maintain the partition difference).

Programming this process is only slightly more complicated. We have to keep track of how we get to our perfect partition. For the example in Figure 4 (and also Table 2), the path is right, left, left, left. Once we have the path to a perfect partition, we begin at the root element and follow it down the tree. However, instead of having just a cardinality number associated with each element, now we also keep track of whether it is an element original to our list and, if it is not, what its two parent numbers are.

Once we have recreated the tree and made it back down to the 0-1 list, everything is set to traverse backward, up the tree. The basic idea is that if something is an original element, it is placed in either the A or B list—if not, it is pushed onto an A or B stack. The nonoriginal 0 is pushed onto the A stack. The A stack gets popped, reads the 0, and sees that it comes from a nonoriginal 4 and an original 4. Because 4 – 4 = 0, the function knows to put the 4's in different groups. The nonoriginal 4 is pushed onto the A stack, while the original 4 is placed in the B list. The A stack is popped again, and the same process is followed. If the difference of the two parent elements equals the child element, they are placed in opposing stacks or lists; otherwise, they are placed in the same stack or list. The process continues until both stacks are empty.

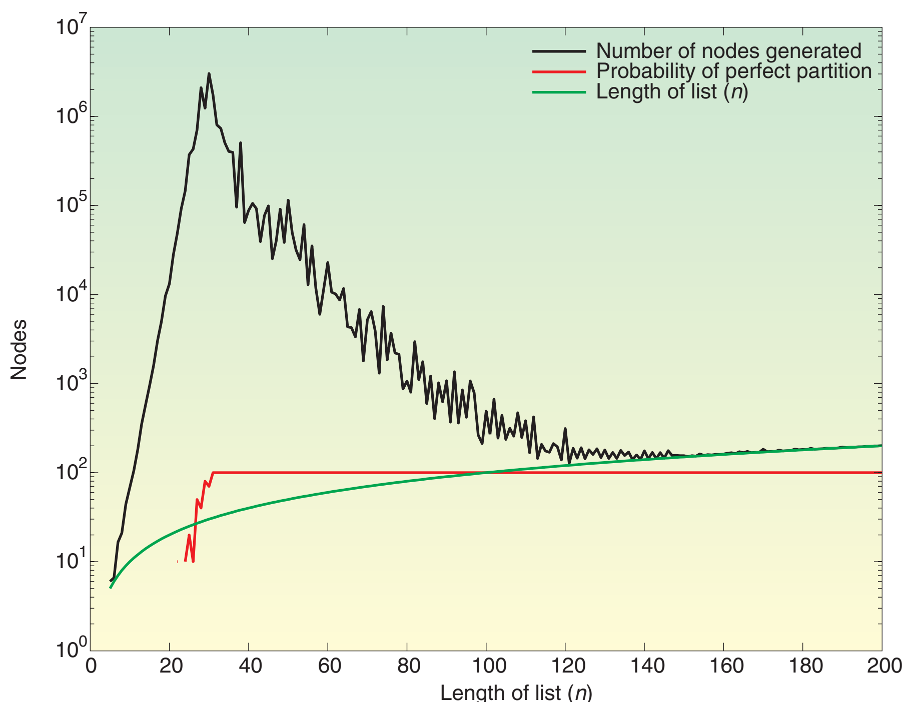



**Figure 5. The plot of the number of nodes that must be visited to find a perfect partition or to determine that none exist.**

## Estimating the Work Done

Because solving large problems can take a great deal of time, we have developed a way to estimate the number of nodes that must be visited in our modified CKK algorithm, using a variation on a technique Knuth developed for estimating the size of backtrack trees. We do this in two ways, depending on the list's length and critical point.

### What Is a Critical Point?

Although it seems that the longer the list of numbers, the more difficult it would be to find a perfect partition, the opposite is actually true. For example, for each length between 5 and 200, we created 100 lists of random 25-bit numbers and submitted them to the partitioning method described earlier. Figure 5 plots the average number of nodes that must be visited to find one perfect partition, if one exists. Once we reach a critical length (around 29), the problem complexity drops sharply, and soon the number of nodes generated acts as $n$.

Mertens gives the formula for finding this critical point as

$$n_c - \log_2(n_c) = \log_2\left(\pi \cdot \sqrt{<x^2> - <x>^2}\right)$$

where $\sqrt{<x^2> - <x>^2}$ is the standard deviation of the $x_i$'s and $n_c$ is the critical point.[2]

### Calculating the Critical Point

After the critical point, the probability that the list of numbers has a perfect partition equals 1. (This was demonstrated empirically in Mertens's paper.[2] He also used methods of statistical mechanics to prove it in another work.[5]) In our experiments, we used a combination of the secant and bisection rule to calculate the critical points.[8] Our estimation method is in two parts: one if $n < n_c$ and one if $n > n_c$.

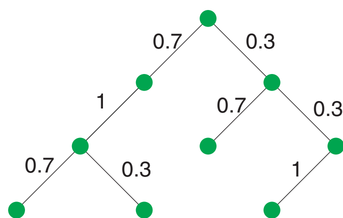



**Figure 6. An example for estimating the size of backtrack trees.**

## How the Knuth Method Works

The Knuth estimating method systematically estimates a tree's size without actually counting all the nodes.[6] Consider the tree in Figure 6, with the probabilities of going either direction indi-

**Table 3. Example of using the Knuth method for estimating the size of a tree.**

| Step | Direction | Probability | Estimate |
|---|---|---|---|
| 1 | Left | 0.7 | $1 + 1 \times 10/7$ |
| 2 | Left | 1 | $1 + 1 \times 10/7 + 1 \times 10/7 \times 1$ |
| 3 | Left | 0.7 | $1 + 1 \times 10/7 + 1 \times 10/7 \times 1$ $+ 1 \times 10/7 + 1 \times 10/7 \times 1 \times 10/7 = 5.90$ |
| 1 | Left | 0.7 | $1 + 1 \times 10/7$ |
| 2 | Left | 1 | $1 + 1 \times 10/7 + 1 \times 10/7 \times 1$ |
| 3 | Left | 0.3 | $1 + 1 \times 10/7 + 1 \times 10/7 \times 1$ $+ 1 \times 10/7 \times 1 \times 10/3 = 8.62$ |
| 1 | Right | 0.3 | $1 + 1 \times 10/3$ |
| 2 | Left | 0.7 | $1 + 1 \times 10/3 + 1 \times 10/3 \times 10/7 = 9.10$ |
| 1 | Right | 0.3 | $1 + 1 \times 10/3$ |
| 2 | Left | 0.3 | $1 + 1 \times 10/3 + 1 \times 10/3 \times 10/3$ |
| 3 | Left | 1 | $1 + 1 \times 10/3 + 1 \times 10/3 \times 10/3$ $+ 1 \times 10/3 \times 10/3 \times 1 = 26.56$ |

cated. We begin with an estimate of 1, for the root. Depending on which direction we choose to go, we multiply 1 by the inverse of the probability of going that direction, then add that number to 1. For the next step, in whichever direction we've taken, we multiply what was added previously by the inverse of the probability of going that direction, then add that number to the previous sum (see Table 3).

The estimates of 5.90, 8.62, 9.10, or 26.56 are all valid for the total number of nodes in the tree. In fact, the sum of the products of the estimates and their respective probabilities yields the correct number of nodes in the tree: $(5.90 \times 0.7 \times 0.7) + (8.62 \times 0.7 \times 0.3) + (9.10 \times 0.7 \times 0.3) + (26.56 \times 0.3 \times 0.3) = 9$.

We selected the probabilities 0.7 and 0.3 in this example for illustrative purposes. The probability of going either direction could just as easily have been 0.5 and 0.5. These probabilities are often called *importance functions*, and choosing them wisely can sometimes help an estimate converge faster.[9]

### Estimating the Work to the Left of $n_c$

We use these same techniques to estimate the number of steps required to find a perfect partition or decide that we have the best one possible. Beginning at the top of the tree, a check is made to see if pruning conditions make either direction a dead end. If only one way is open, we proceed in that direction, performing the Knuth calculation with a probability of 1. If either way is available, we choose a random branch and perform the Knuth calculation with a probability of 0.5. This is continued until no further movement down the tree is allowed.

This estimate is applied to the same 100 lists of 25-bit numbers used before, taking 10 samples on each list. For $n < n_c$, the maximum percent error hovers around 50 percent, but increases dramatically for $n > n_c$. This is because what the Knuth method actually estimates is the number of nodes that must be visited to find *all* the perfect partitions. Before the critical point, there are usually only one or two, if any, perfect partitions, so this presents no problem. After the critical point, however, we begin to have more than one or two perfect partitions—that is, more paths down the tree become viable options, so the estimate explodes.

### Estimating the Number of Perfect Partitions

The next logical step to correct the estimate on the right side of $n_c$ is to find a good estimate for the number of perfect partitions. For this estimate, we use the same method from Knuth, except that instead of using the multiply-then-add method, which counts all the tree's nodes, we use the multiply-only method, which counts just the leaves at the bottom of the tree. Also, because we know that we are only guaranteed the existence of perfect partitions for lengths larger than the critical point, we only multiply while the length of the list in the current node (which decreases by 1 as we go down each level of the tree) is greater than the critical point.

We must make one final adjustment: If the original list's length is even, everything is fine, but if it's odd, we multiply our final perfect es-

```
Estimate (MainList[(x1, m1 ), ..., (xk, mk )], stepEstimate, nextTerm,
perfectEstimate)
```

*For the first call,* `stepEstimate` *is initialized to 1 and* `nextTerm` *to 1, and* `perfectEstimate` *is initialized to 1 for even lengths and 2 for odd lengths of list.*

```
If k = 1 then return;

List1 = sort[ (x1 − x2, m1 − m2), (x3, m3), ..., (xk, mk) ]
List2 = sort[ (x1 + x2, m1 + m2), (x3, m3), ..., (xk, mk) ]

if 2 · max_{x∈List1} x − Σ_{x∈List1} x ≤ 1 and 2 · max_{m∈List1} |m| − Σ_{m∈List1} |m| ≤ 1
     goLeft = TRUE;

if 2 · max_{x∈List2} x − Σ_{x∈List2} x ≤ 1 and 2 · max_{m∈List2} |m| − Σ_{m∈List2} |m| ≤ 1
     goRight = TRUE;

if !goLeft and !goRight
     return;

if goLeft and goRight
     if length(List1) ≥ ceiling(CRITICAL_POINT)
                perfectEstimate = perfectEstimate · 2
     nextTerm = nextTerm · 2 (since 1/probability = 1/.5 = 2)
     pivot = randomNumber();

if goLeft and !goRight
     pivot = 0

if !goLeft and goRight
     pivot = 1

stepEstimate = stepEstimate + nextTerm

if (pivot <.5)
     Estimate(List1, stepEstimate, nextTerm)
Else
     Estimate(List2, stepEstimate, nextTerm)
```

**Figure 7. Our modified estimating algorithm.**

timate by 2. The reasoning here is that to find a balanced partition for an even list, we must have exactly the same number of elements in each group—a cardinality difference of 0. However, for an odd list, we could have a cardinality difference of 1 or –1: twice as many possibilities. This is the equivalent of simply initializing the perfect estimate to 1 for even numbers and 2 for odd numbers. With these modifications, Figure 7 demonstrates our estimating algorithm.

Figure 8 shows the results of using our algorithm to find the number of perfect partitions of lists of 10-bit numbers. We used 100 samples on a single list for each length.

### Estimating the Work to the Right of $n_c$

As mentioned earlier, the Knuth method actually estimates the number of nodes that must be visited to find not just one perfect partition but all of them. As such, we must adjust the estimate for length $n > n_c$. Imagine visiting all the

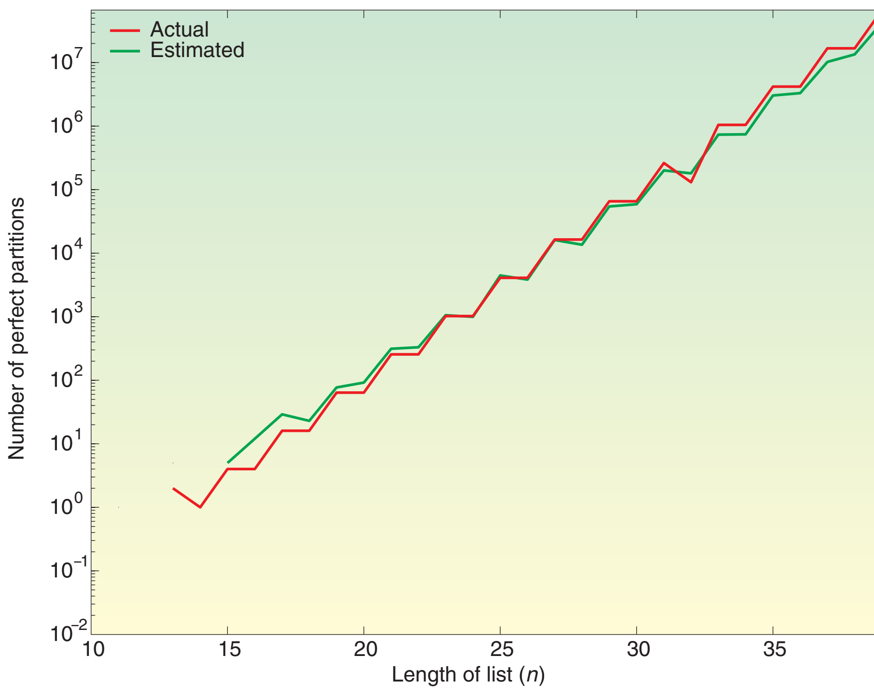



**Figure 8. An estimate of the number of perfect partitions and the actual number of perfect partitions. Results are for single lists of 10-bit random numbers. Estimated results were made by averaging the results of 100 samples taken on those same lists.**

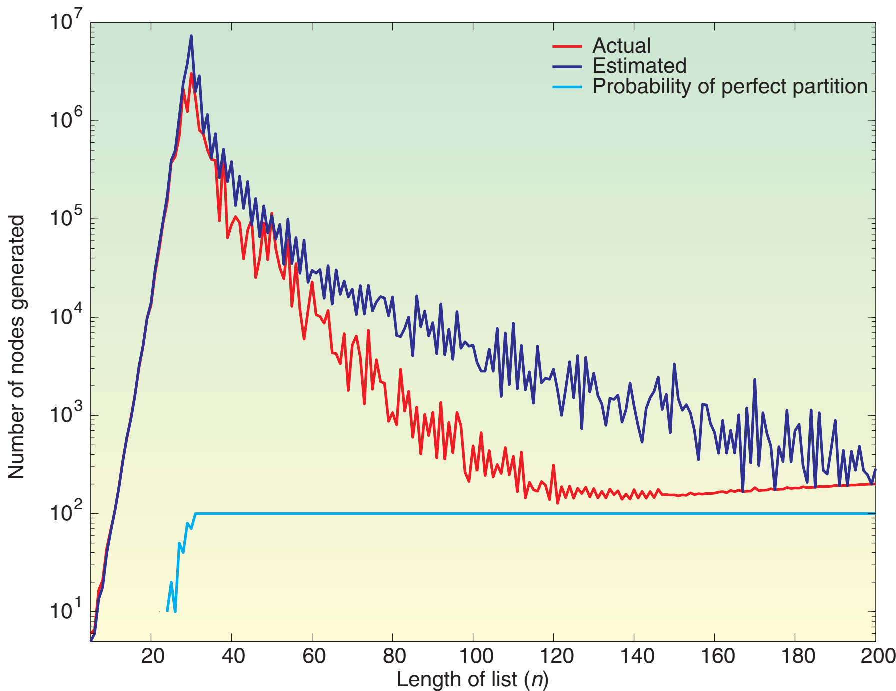



**Figure 9. The actual and estimate work done using our modified CKK method. Estimated results are the averages of 100 samples taken on those same lists.**

branches that would be required to find all the perfect partitions. Let $x$ be the number of perfect partitions, and let our Knuth estimate be $T$. Then, the effective work for getting from the root to any of the leaves is $\log_2(x)$ if we assume a binary subtree. Thus, the amount of traversal work done to actually lead to these different leaves is $x \bullet \log_2(x)$.

If the probability of something happening is $p$, and we do independent trials, then the expected number of trials before that something happens is $1/p$. For example, double 6's should show up once every 36 rolls of two dice. In our case, the number of nodes that lead to a solution is $x \bullet \log_2(x)$, and the total number of nodes is $T$, so the probability that any node leads to a so-

lution is $x \bullet \log_2(x)/T$. Hence, the number of nodes that must be looked at to get a solution is

$$\frac{T}{x \cdot \log_2(x)}.$$

When $n \geq n_c$, the work is known to be $n$ and, in fact, can never be less than $n$. So, if the estimate is ever less than $n$, we set the estimate to $n$. Using this final estimation method, we can find the close-fitting estimate in Figure 9.

## Putting the Pieces Together

To run the method, do the following:

1. Calculate the critical point $n_c$
2. Run the pseudo code to get `Estimate`
3. If $n > n_c$, `Estimate = Estimate/(perfectEstimate*log2(perfectEstimate))`
4. If `Estimate` < $n$, `Estimate` = $n$

We plan to examine the same problem for more than two sets and to include other features of Mertens's method. In future investigations, we plan to look for a correction factor to the estimate for $n > n_c$.


## Acknowledgments

*The SURF program, a NIST–NSF partnership, partially supported Andreas's research.*


## References


1. L. Tsai, "Asymptotic Analysis of an Algorithm for Balanced Parallel Processor Scheduling," *SIAM J. Computing*, vol. 21, no. 1, 1992, pp. 59–64.
2. S. Mertens, "Phase Transition in the Number Partitioning Problem," *Physical Rev. Letters*, vol. 81, no. 20, 1998, pp. 4281–4284.
3. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1997.
4. R.E. Korf, "A Complete Anytime Algorithm for Number Partitioning," *Artificial Intelligence*, vol. 106, 1998, pp. 181–203.
5. S. Mertens, *A Complete Anytime Algorithm for Balanced Number Partitioning*, 1999, unpublished preprint at xxx.lanl.gov/abs/cs.DS/9903011; http://itp.nat.uni-magdeburg.de/mertens/papers/cckk.shtml.
6. D.E. Knuth, "Estimating the Efficiency of Backtrack Programs," *Selected Papers on Analysis of Algorithms*, CSLI Publications, Stanford, Calif., 2000.
7. N. Karmarkar and R.M. Karp, *The Differencing Method of Set Partitioning*, tech. report UCD/CSD 82/113, Computer Science Division, Univ. of California, Berkeley, Calif., 1982.
8. L.F. Allen et. al., *Fundamentals of Numerical Computing*, John Wiley & Sons, 1996.
9. I. Beichl and F. Sullivan, "The Importance of Importance Sampling," *Computing in Science & Eng.*, vol. 1, no. 2, 1999, pp. 71–73.



**April K. Andreas** is a graduate student in mathematics at Southern Methodist University, where she also received a BS in mathematics. She performed the work for this article while she was a summer undergraduate research fellow (SURF) at the National Institute of Standards and Technology. Contact her at Southern Methodist Univ., PO Box 750156, Dallas, Texas 75275; andreas@mail.smu.edu.

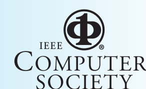
**Isabel Beichl** is a mathematician in the Information Technology Laboratory at the National Institute of Standards and Technology, where she works on probabilistic methods for physical problems. She received a BA from the University of Pennsylvania and an MA and PhD from Cornell University, all in mathematics. She is a member of the American Mathematical Society and an editor of the Computing Prescriptions department for *CiSE* magazine. Contact her at NIST, Gaithersburg, MD 20899; isabel.beichl@nist.gov.



**Eleven good reasons why close to 100,000 computing professionals join the IEEE Computer Society**

**Transactions on**

- *Computers*
- *Information Technology in Biomedicine*
- *Knowledge and Data Engineering*
- *Mobile Computing*
- *Multimedia*
- *Networking*
- *Parallel and Distributed Systems*
- *Pattern Analysis and Machine Intelligence*
- *Software Engineering*
- *Very Large Scale Integration Systems*
- *Visualization and Computer Graphics*




*computer.org/publications/*