# Programming Assignment 3

Albert Young and Peregrine Badger
CS 124: Data Structures and Algorithms

April 30, 2013

## Dynamic Programming Solution

We are given a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers and their sum $b$. We initialize a two-dimensional array $X$ of dimensions $[n+1][b+1]$. Define $X(i, j)$ to be true if there exists some subset of $a_1, \ldots, a_i$ that has sum $j$, false otherwise. Then, our recursion is:

$$X(i, j) = X(i-1, j) || X(i-1, j - A_i)$$

Initialize $X(0, 0) =$ true, and the rest of the first row $X(0, j) =$ false, $\forall j \geq 1$, since the sum of an empty set must be 0. Fill up all other $X(i, j)$ until $i = n, j = b$. We next traverse the last ($n$th) row of the array and find the maximum $j \leq b/2$ such that $X(n, j)$ is true. Let us call this cell $X(n, j_0)$. We then calculate the residue $u = b - 2j_0$.

There are $nb$ array entries, each of which takes constant time to fill, so the overall time complexity is $O(nb)$.

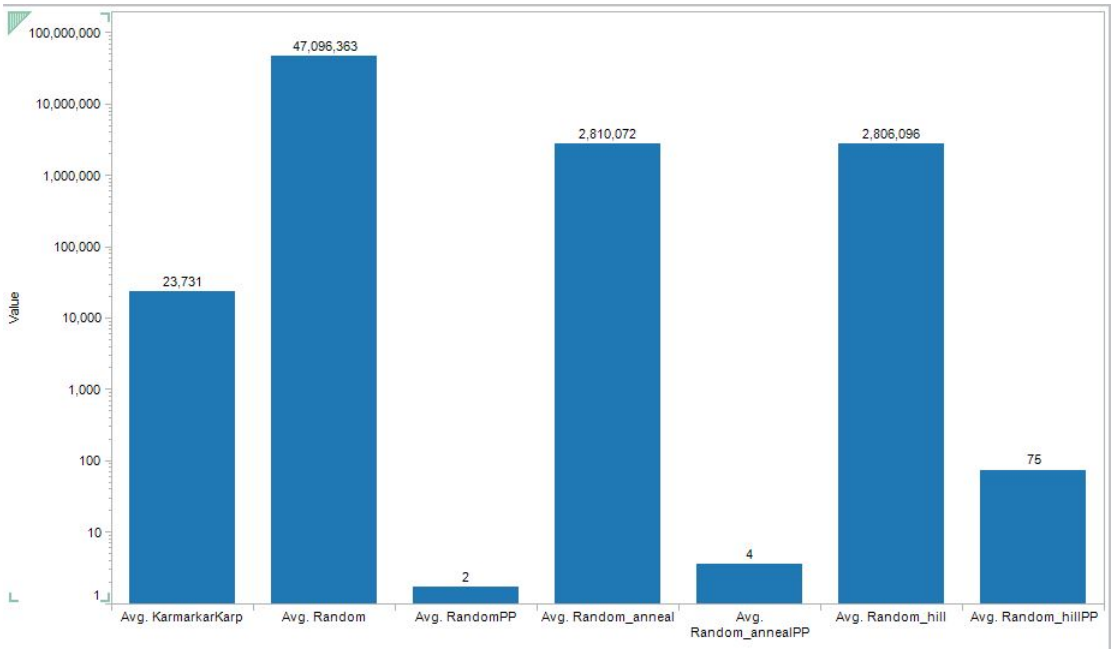
## Karmarkar Karp Analysis

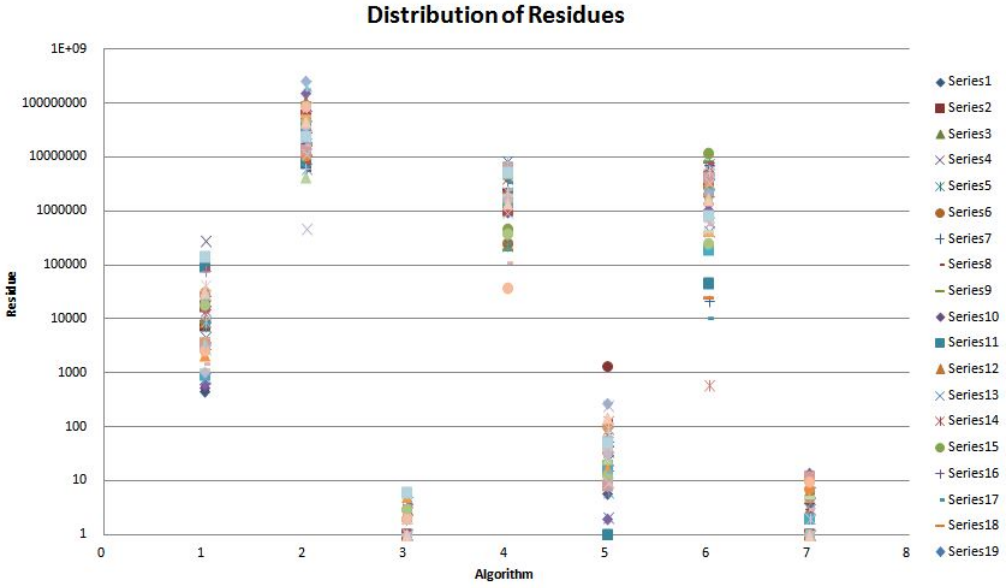
The algorithm starts by sorting the sequence in decreasing order in a max-heap. The algorithm then repeatedly pops off the largest two numbers and inserts their difference back into the heap in sorted order. With each iteration, the number of elements in the heap decreases by 1. The algorithm returns when the residue is the single remaining element in the heap. Sorting takes $O(n \log n)$ time and inserting the differences in sorted order takes $O(n \log n)$ time, so overall the algorithm takes $O(n \log n)$ steps.

The correctness of the algorithm follows from the implementation of the heap. The two elements removed each time must be the largest, and their difference is always inserted in correct order into the remainder of the heap. When only one element remains, the residue is returned.

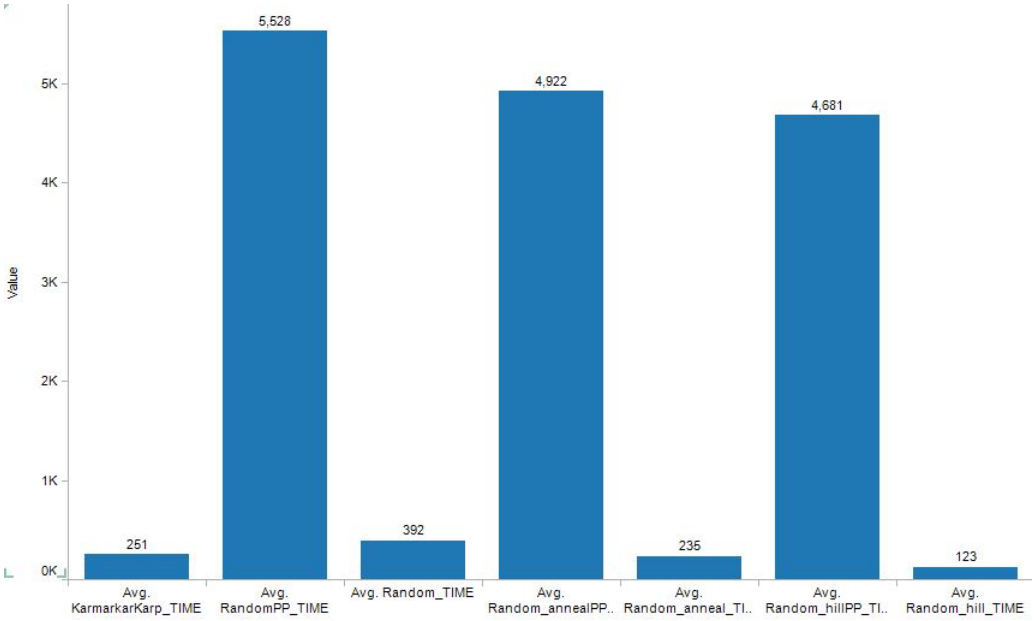
# Randomized Algorithms

### Average residues



**Average running times (in milliseconds)**




We generated and ran 50 random instances of 100 integers in the range $[1, 10^{12}]$ for 25,000 iterations for each algorithm. Our results show that by the value of the residue returned, the algorithms can be ranked from lowest to highest as follows:

1. Repeated Random with Prepartitioning
2. Simulated Annealing with Prepartitioning
3. Hill-climbing with Prepartitioning
4. Karmarkar-Karp
5. Hill-climbing (Random Signed)
6. Simulated Annealing (Random Signed)
7. Repeated Random (Random Signed)

The algorithms can be ranked by average running time (cumulative for 25,000 iterations) from lowest to highest:

1. Hill-climbing (Random Signed)
2. Simulated Annealing (Random Signed)
3. Karmarkar-Karp
4. Repeated Random with Prepartitioning
5. Hill-climbing with Prepartitioning
6. Simulated Annealing with Prepartitioning
7. Repeated Random with Prepartitioning

*Note: the Karmarkar-Karp algorithm was run for 25,000 iterations even though the result it produces does not change between iterations in order to have an accurate running time comparison with the other algorithms*

These results make several things clear. First, we can see that prepartitioning and Karmarkar-Karp yields much lower residues than simply running a random signed algorithm, even one using simulated annealing. It seems that the best solutions take advantage of the Karmarkar-Karp heuristic, as well as some random variation. This makes sense because the Karmarkar-Karp heuristic improves upon the random signed solutions by several orders of magnitude, and each of the solutions using pre-partitioning call Karmarkar-Karp 25,000 times.

Secondly, although the pre-partition algorithms clearly give better results, they also take much longer to run (up to 25-30× slower), so that on some systems or for some data sets, or depending on the required precision, it may be preferable to only run Karmarkar-Karp with no pre-partitioning.

Finally, it is interesting to note that the random simulated annealing algorithm is slightly less effective than the random hill climbing algorithm, which is probably due to random variation. This suggests that for these data sets, random variation does not accomplish much in arriving at an advantageous place in the solution landscape.

# Using KK to Improve Randomized Algorithms

The Karmarkar-Karp algorithm could be used as a starting point for the three non-prepartitioned algorithms that were tested. This could be implemented by running Karmarkar-Karp and saving the solution that Karmarkar-Karp can generate when calculating a residue. Since Karmarkar-Karp can be used to generate a residue and two sets that will yield the residue, we could use these sets to generate a solution of +1s and -1s, such that we could then run the random algorithms on this solution. On average, this would probably yield residues somewhat smaller than Karmarkar-Karp on its own, since by attempting to exchange values between the two sets, a smaller residue would be found some of the time, if not every time.

Additionally, using the result of Karmarkar-Karp as an input to one of the randomized algorithms would be an effective upper bound on the residue that is returned, since the randomized algorithms will only try to improve upon the Karmarkar-Karp solution. Using the Karmarkar-Karp solution will also dictate where in the solution landscape the hill-climbing and simulated-annealing algorithms begin. They will begin searching locally starting from the Karmarkar-Karp solution. In this way, the randomized algorithms can be guaranteed to perform better than the Karmarkar-Karp algorithm alone.

Karmarkar-Karp calculates much better outcomes than the randomized algorithms alone,

but the values in Karmarkar-Karp are also far from optimal, as we can see upon comparison to the prepartitioned examples. Thus, it seems likely that swapping values randomly could result in significantly more efficient partitions of the set.