Number Partition Heuristics

Peter Chang and Tosin Alabi

1 Introduction

In this project, we will implement several heuristics for the Number Partition Problem. We will compare results using the Karmarkark-Karp algorithm and the Repeated Random, Hill Climbing, and Simulate Annealing heuristics.

The standard Karmarkar-Karp algorithm is located in kk.py and is executable through the command line with:

python kk.py inputfile

Where inputfile is a text file with 100 integers, one per line. The output is the residue of this list of 100 integers.

Our additional code (used to test the other methods) is contained in the file everything.py. It is not executable via command line but it is available for reference.

2 Number Partition Problem

The Number Partition problem is trying to determine if a set of numbers $T = a_1, ..., a_n$ can be evenly divided into two sets A_1 and A_2. Thus, we are trying to determine whether we can create a set A_1 such that $A_1 = \lfloor \frac{A}{2} \rfloor$, where A is the sum of all of terms in T. If we can do so, and A is an even number, then $A_2 = \lfloor \frac{A}{2} \rfloor$. If A is odd, then $A_2 = \lceil \frac{A}{2} \rceil$ and this is the best we can do in this case.

We will solve this problem dynamically:

• Define $X(i, j)$ as true if a subset of $a_1, ..., a_j$ sums up to i.

• Recursively define $X(i, j)$ as true if either $X(i, j-1)$ is true or if $X(i - a_j, j - 1)$ is true. In other words, it is true if either the set minus one element sums up to i or if the whole set was equal to i in the first place.

• We will start at $i = j = 0$ and build up a table by working backwards. This table will be of size $\lfloor \frac{A}{2} \rfloor$ by n. We then return $X(\lfloor \frac{A}{2} \rfloor, n)$

After this part, we have a set which contains all the elements that we want in one partition A_1. So our output sequence S will return (without loss of generality) 1 if the element is in A_1 and -1 if it is not in A_1.

Since the check time for this is constant, this algorithm takes time $\lfloor \frac{A}{2} \rfloor * n \equiv n * b$, where b is the sum of the terms in T.

3 Karmarkar-Karp Algorithm

We can implement the Karmarkar-Karp algorithm in $O(nlog(n))$ time by loading the input in a max heap.

At each step, we will max heapify, and take the largest two numbers from the top of the heap. This takes $log(n)$ time. Since at each step, one of the terms will be differenced to zero, we must do this n times.

Thus, by using a max heap, we get a run time of $O(nlog(n))$.

4 Heuristics

4.1 Residue Data

This table includes all the residues that were returned by the various algorithms run on 50 different trials using a maximum iteration value of 25,000:

ALG	Mean	Median	Min	Max	Std. Dev
KK	249304.5	128470.5	2932.0	1282498.0	301879.9
Pre-partition Representation					
RAND	169.8	115.0	1.0	775.0	167.0
HILL	627.7	441.5	6.0	3932.0	669.5
SIMUL	246.5	205.5	10.0	1030.0	227.3
Standard Representation					
RAND	372387037	340636726	2892820	1402064651	296901789
HILL	313560763	293533580	7559494	1014290301	257899495
SIMUL	271537797	189833882	2917455	1233763347	269502681

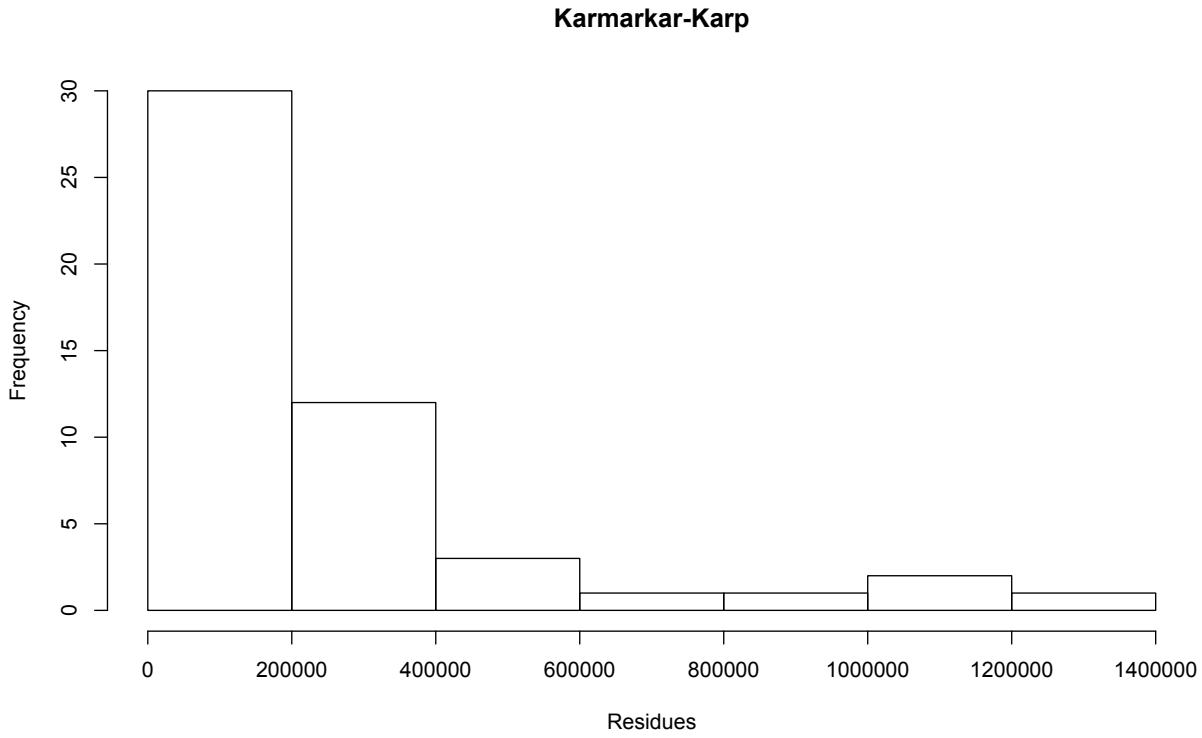
As we can clearly see, the pre-partition representation is significantly better than the standard Karmarkar-Karp algorithm as well as the standard representations. Within these, the Repeated Random algorithm returned the best results, followed by the Simulated Annealing. The Hill Climbing method was clearly the worst with this representation.

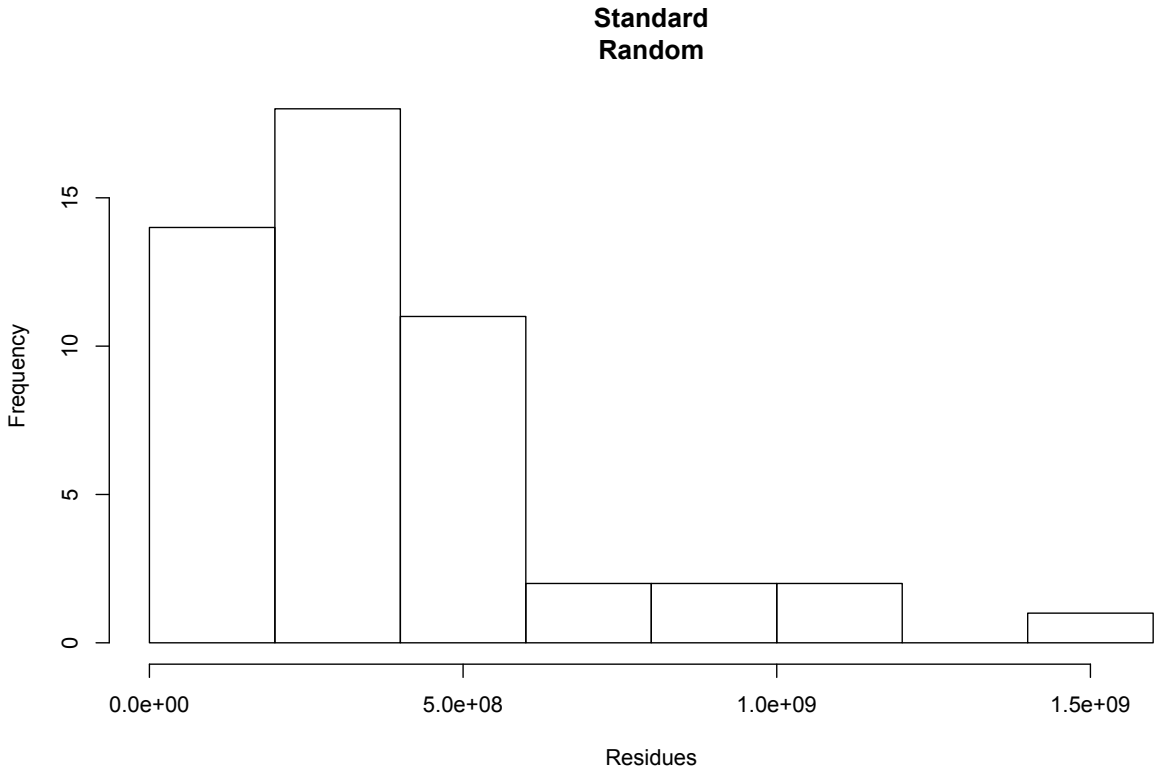
Running the algorithms on the standard representations returned results that were much worse than the Karmarkar-Karp algorithm (by a factor of 10^3). With these representations, the Simulated Annealing was the fastest, followed by the Hill Climbing method, and then the Repeated Random Algorithm.

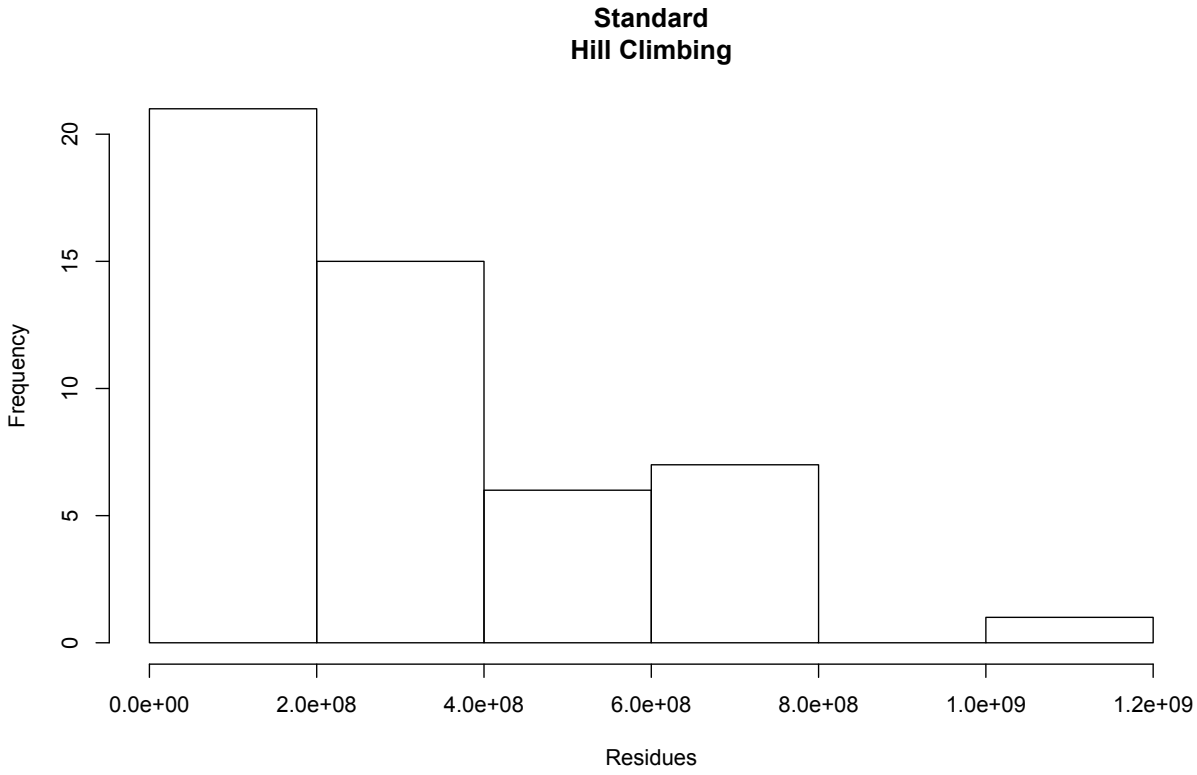
In both of the representations, the median also followed the same pattern. This means that none of the methods were significantly affected by any outlier cases.

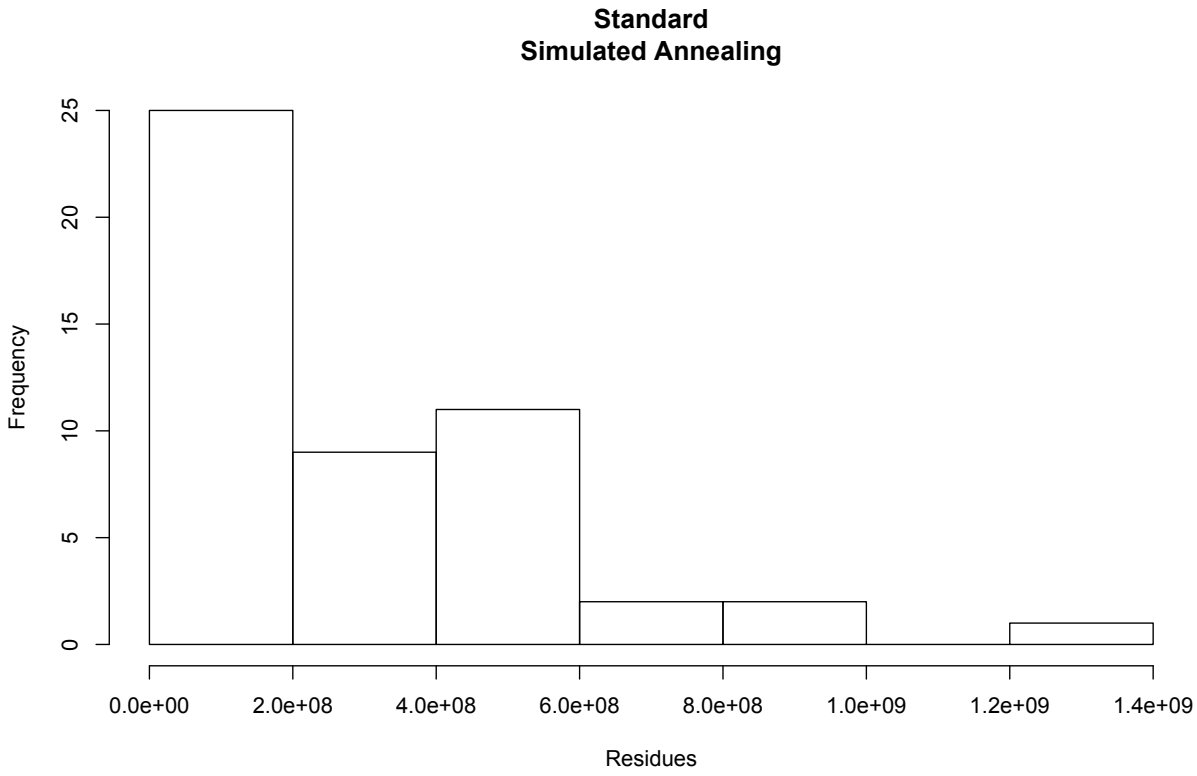
Concerning the pre-partitioning representation the algorithm returned results that were somewhat unexpected. The random algorithm came in first place with a mean residue obtained of 170, the simulated annealing algorithm in second place with with a mean of 247, and the hill climbing algorithm cam in last place with a mean of 628. That simulated annealing was found to be worse than using repeated randomness can be attributed to the specific conditions of our program. For example, only 25,000 iterations were used on each problem instance.It's possible that given more iterations, simulated annealing would have shown itself as the best possible approach.

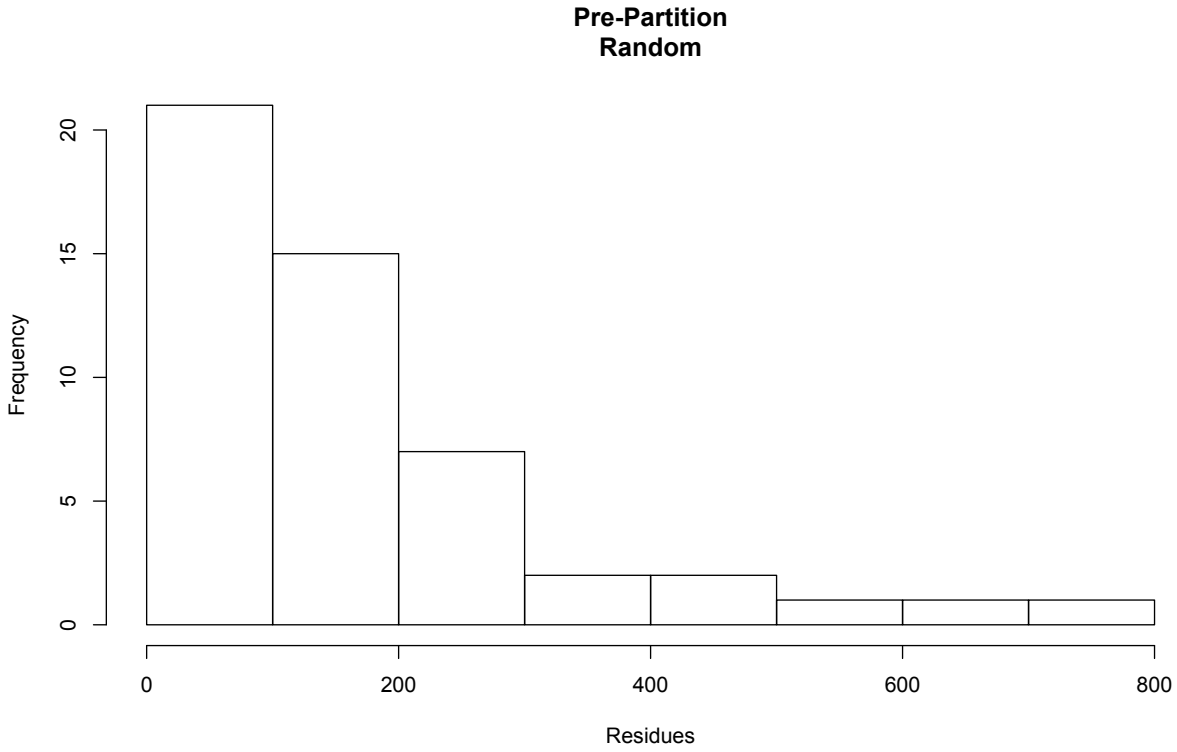
4.2 Histograms

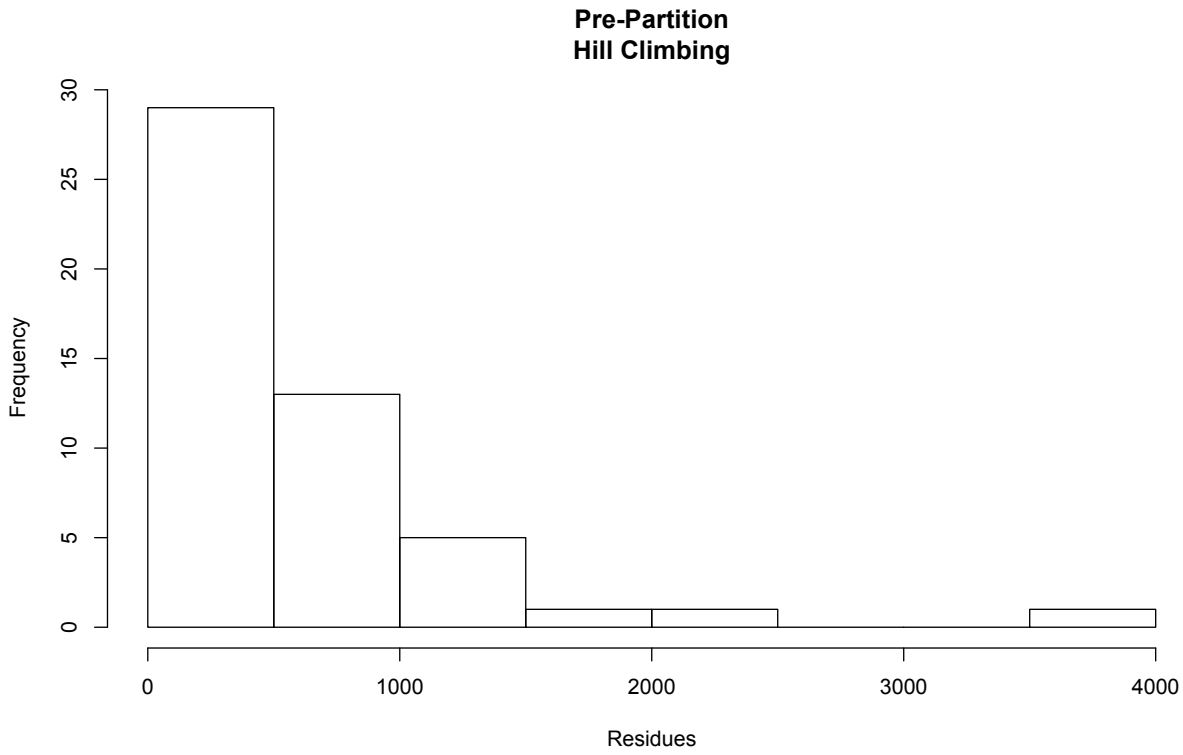


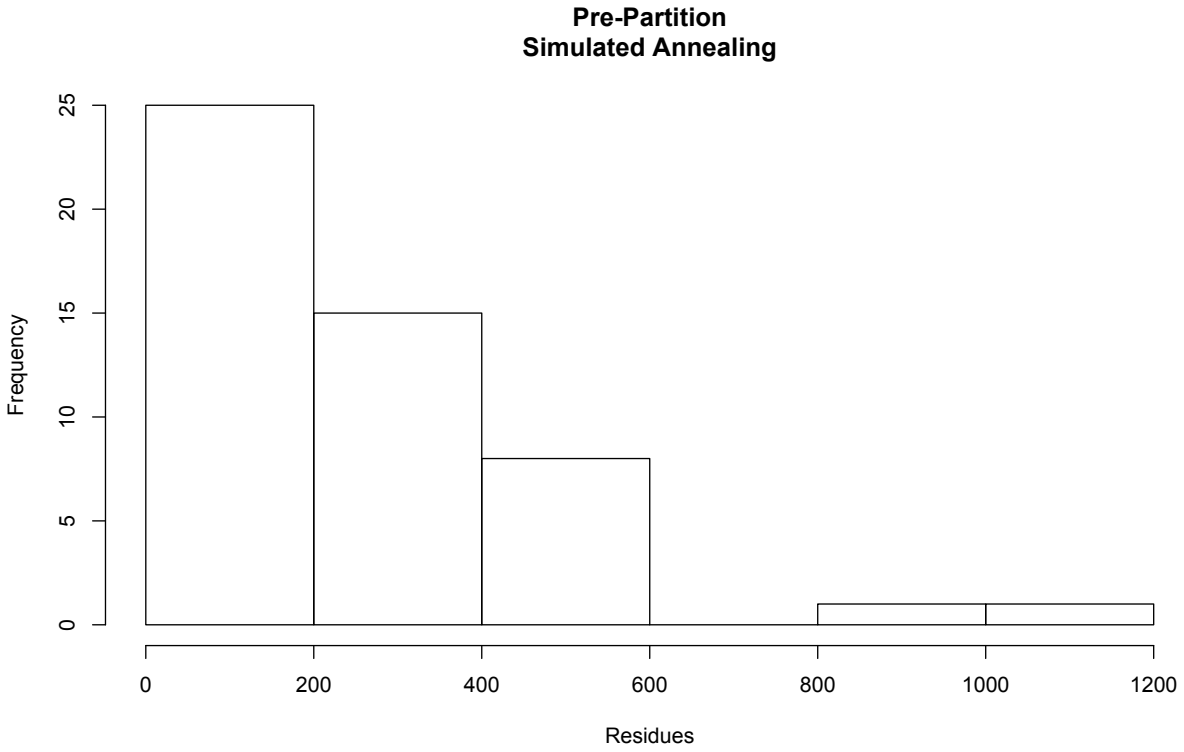



Standard
Hill Climbing
Frequency
0
5
10
15
20
0.0e+00
2.0e+08
4.0e+08
6.0e+08
8.0e+08
1.0e+09
1.2e+09
Residues


Standard
Simulated Annealing
Frequency
0
5
10
15
20
25
0.0e+00
2.0e+08
4.0e+08
6.0e+08
8.0e+08
1.0e+09
1.2e+09
1.4e+09
Residues


Pre-Partition
Random
Frequency
0
5
10
15
20
0
200
400
600
800
Residues


Pre-Partition
Hill Climbing
Frequency
0
5
10
15
20
25
30
0
1000
2000
3000
4000
Residues



4.3 Time Data

Presented here is data for the runtimes (in seconds) of the algorithms:

Algorithm	Total Time	Time per trial
KK	.039	.00078
Pre-partition Representation		
RAND	820.86	16.42
HILL	1322.94	26.46
SIMUL	1912.17	38.24
Standard Representation		
RAND	448.62	8.97
HILL	77.58	1.55
SIMUL	202.40	4.05

The Karmarkar-Karp algorithm was clearly the fastest of all the methods, which makes sense because we had to run 25,000 iterations on all of the other methods.

As we can see, the standard representations were all faster than the pre-partitioning. This can be attributed to their respective residual functions. While the pre-partition methods must constantly call the Karmarkar-Karp algorithm, the standard methods could simply multiply the solution array cells by the input array cells and sum them together.

Within the standard representations, the Repeated Random was the slowest, which could be attributed to the speed of the random number generation. The random number generator is called 25,000 times in Repeated Random while it is only called once in the other two methods.

In both methods, the Hill Climbing algorithm was faster than the Simulated Annealing method. This is because the Simulated Annealing runs the exact same way as Hill Climbing, only it also has to compute a probability function every single time it finds a worse solution.

As for the pre-partitioning times, we found that repeated randomness was faster than hill climbing which was faster than simulated annealing. One cause of the differences was due to implementation details. Just looking at the pseudocode, one might think that hill climbing ought to be faster than repeated randomness, but our hill climbing algorithm made calls to a user defined function which had to do an extra O(n) work during every iteration of its big loop, slowing its execution. Simulated annealing was slowest because it made the most calls to residue. In the clause in which $S = S'$ with a certain probability, residue is called two extra times compared to the random and hill climbing algorithms. Our residue function worked via recursive calls, and so many calls to this function proved expensive.

5 Using Karmarkar-Karp

This Karmarkar-Karp algorithm returns an approximated residue from the given input. While this residue is not necessarily optimal, it is a good approximation. We can use this to improve our other algorithms.

This would be most useful in the Hill Climbing algorithm, where we could add the conditional that the starting random solution must have a residue less than that given by the Karmarkar-Karp algorithm. This would allow the algorithm to quickly start at a point that is good and allow for more searches for better neighbors.

This would likewise have a positive effect on the Simulated Annealing Algorithm, however it would have less of an effect because of the property of moving to worse neighbors.

Since the Repeated Random process is entirely random, it would not have an effect on this process.

If we built the Karmarkar-Karp algorithm to also return the solution array, the effect would be even better. That is, we could begin with the solution returned by the Karmarkar-Karp algorithm, instead of a random solution.

In the case of the Repeated Random algorithm, this would simply provide a bound for our residue output. In the cases of Hill Climbing and Simulated Annealing, it would save a lot of steps and allow for more searches of better solutions. The downside is that we would always start in the same location (not a random one), which could lead the algorithm getting "trapped" in a sub-optimal solution (similar to how greedy algorithms fail sometimes). Statistically though, this would lead to a better output.