CS124 Programming Assignment 3

Anusha Murali

April 17, 2024

1 Introduction

In this programming assignment, we implemented a number of heuristics for solving the Number Partition problem, which is known to be NP-complete. In a nutshell, the number partition problem can be described as follows: The number partition problem takes a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers as its input. The output is a sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in \{-1, +1\}$ such that the *residue*

$$u = \left| \sum_{i=1}^{n} s_i a_i \right|$$

is minimized. So, our goal is to split the set of numbers given by A into two subsets A_1 and A_2 with roughly equal sums. The absolute value of the difference of the sums is the residue.

2 Dynamic Programming Solution

We would like to find a dynamic programming solution to the Number Partition problem, which is to determine whether a set of non-negative numbers $A = (a_1, a_2, \ldots, a_n)$ can be divided into two subsets A_1 and A_2 with roughly equal sums, and to return the minimum residue. By using a second array as indicated at the end of the section below, the algorithm can be made to return the actual partitions A_1 and A_2 that result in the minimum residue.

Let us first consider the problem of determining whether the given input A can be divided into two subsets A_1 and A_2 such that the sums of A_1 and A_2 are equal. Let us assume that the sum of the elements in A is b, so our problem is to determine whether it is possible to generate a subset A_1, whose elements sum to $\lfloor b \rfloor$. In the case of an odd b, we will have the sum of elements in A_1 be $\lfloor b \rfloor$ and the sum of elements in A_2 be $\lceil b \rceil$. For simplicity, we assume that b is even.

Let $D(i, j)$ be an element of a 2-dimensional array, which tells us whether there is a subset of the set $\{a_1, a_2, \ldots, a_i\}$ such that the sum of its elements is equal to j. The value of $D(i, j)$ is 1 if there is a subset of the set $\{a_1, a_2, \ldots, a_i\}$ such that the sum of its elements is equal to j, and 0 otherwise. We are interested in the value $D(n, \lfloor b/2 \rfloor)$, as our original set A has n elements and its sum is b. We make the following observations:

1. Every subset has an empty subset, whose elements sum to 0. So $D(i, 0) = 1$.

2. If a sequence $\{a_1, a_2, \ldots, a_i\}$ has a subset, whose elements sum to j, then either the sequence $\{a_1, a_2, \ldots, a_{i-1}\}$ has a subset whose elements sum to j, or the sequence $\{a_1, a_2, \ldots, a_{i-1}\}$ contains a subset whose elements sum to $j - a_i$.

Our base case is $D(i, 0) = 1$ as all subsets of $A = \{a_1, a_2, \ldots, a_i\}$ contain an empty subset. Therefore, we have the following recurrence:

$$D(i, j) = \begin{cases} 1 & \text{if } D(i-1, j) = 1 \text{ or } D(i-1, j-a_i) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

We first fill our 2-dimensional array D of size $n \times \lfloor b \rfloor$ using the above recurrence, starting from the base case. Once the array is filled, we check the entry of $D(n, \lfloor b/2 \rfloor)$. If $D(n, \lfloor b/2 \rfloor) = 1$, then the optimal residue is 0 for even b, and 1 for odd b. If $D(n, \lfloor b/2 \rfloor) = 0$, then we check the value of $D(n, \lfloor b/2 \rfloor - 1)$. If $D(n, \lfloor b/2 \rfloor - 1) = 1$, then the optimal residue is 2 for even b, and 3 for odd b. If $D(n, \lfloor b/2 \rfloor - 1) = 0$, then we check the value of $D(n, \lfloor b/2 \rfloor - 2)$. If $D(n, \lfloor b/2 \rfloor - 2) = 1$, then the optimal residue is 4 for even b, and 5 for odd b. This pattern continues.

Therefore, we have a way to identify the most optimal residue by tracking the entries in the $D(n, \lfloor b/2 \rfloor - k)$ array, starting with $k = 0$, until we find the value of k for which $D(n, \lfloor b/2 \rfloor - k) = 1$. Hence, the smallest residue as $2k$ for even b and $2k + 1$ for odd b. □

We can also find the actual partitions A_1 and A_2 by maintaining a separate array P, whose entries are all first initialized to 0. If $P[i, j] = 1$, then the corresponding a_i belongs to A_1; if $P[i, j] = 2$, then the corresponding a_i belongs to A_2. If we found $D[i, j] = 1$ and $D[i - 1, \lfloor \frac{b}{2} \rfloor - a_i] = 1$, we set $P[i, j] = 1$ (which means this element belongs to A_1). If we found $D[i - 1, \lfloor \frac{b}{2} \rfloor - a_i] = 0$ and $D[i - 1, \lfloor \frac{b}{2} \rfloor] = 1$, we set $P[i, j] = 2$ (which means this element belongs to A_2). Suppose the first value of k for which $D(n, \lfloor b/2 \rfloor - k) = 1$ is K. Then we check $P[n, b-K]$. If $P[n, b-K] = 1$, then we place a_n in the set A_1 and recursively fill the partition by moving to $D[i - 1, \lfloor \frac{b}{2} \rfloor - a_i]$. On the other hand, if $P[n, b - K] = 2$, then we place it in A_2 and recursively fill the partition by moving to $D[i - 1, \lfloor \frac{b}{2} \rfloor]$. At the end of this process, we will have the desired partitions A_1 and A_2, with the lowest possible residue. □

2.1 Runtime and Space Complexities

In order to fill our two dimensional array $D(n, \lfloor b/2 \rfloor)$, assuming that takes constant time to turn each entry 1 or 0, it will take $O(nb)$. When we try to determine the optimal residue, in the worst case, we will travel at most b positions with the runtime of $O(b)$, which is less than $O(nb)$, the time to fill the array. Hence, our dynamic programming problem has a runtime of $O(nb)$. The space complexity is also $O(nb)$ as we require an $n \times \lfloor b/2 \rfloor$ array. □

3 Karmarkar-Karp Algorithm

Karmarkar-Karp (KK) algorithm is a deterministic heuristic algorithm for solving the Number Partition problem. This heuristic uses the method of differencing to arrive at the optimal residue. The differencing idea is to take two elements, say a_i and a_j, from the input sequence A, and replace the larger element by $|a_i - a_j|$, while replacing the smaller element by 0. The intuition is that if we decide to put a_i and a_j in different sets, then it is as though we have one element of size $|a_i - a_j|$around. In other words, the Karmarkar-Karp algorithm is an algorithm based on differencing, which repeatedly takes two elements from A and performs a differencing until there is only one element left; this element equals an attainable residue.

We can use a Max Heap to implement the KK algorithm in $O(n \log n)$ time. We assume that the arithmetic operations such as differencing take constant time. As we discussed in the earlier part of the course, the initial building of a Max Heap of n elements takes $O(n \log n)$ time. This is because an insertion into a Max Heap takes $O(\log n)$ time, with n elements to be inserted initially. During each step of the KK algorithm, we need to remove the two largest elements from the remaining sequence, which is equivalent to removing the top most element of the Max Heap successively twice. This of course takes constant time. However, inserting the two elements, $|a_i - a_j|$ and 0, back into the Max Heap takes $O(\log n)$ time. We run these sequence of operations a total of $(n - 1)$ times, as the last remaining element in the Max Heap is the solution (optimal residue) that we are looking for. Therefore, the total runtime of the KK algorithm using a Max Heap is $O(n \log n)$ as desired. □

4 Implementation

Our implementation of the various heuristics for solving the Number Partition problem is based on two different ways to represent solutions to the problem and the state space. The KK algorithm was implemented using a Max Heap to achieve an $O(n \log n)$ runtime.

4.1 The Solution Representations

For our implementation, we used two distinct representations for the solutions, namely (1) the standard representation and (2) the pre-partitioning representation. Each of the three heuristics (Repeated Random, Hill Climbing and Simulated Annealing) was implemented with the above two representations and their runtimes as well as the residues generated were compared. We first describe the two solution representations.

4.1.1 The Standard Representation

The standard representation of a solution is simply as a sequence S of $+1$ and -1 values. A random solution can be obtained by generating a random sequence of n such values. Thinking of all possible solutions as a state space, a natural way to define neighbors of a solution S is as the set of all solutions that differ from S in either one or two places. This has a natural interpretation if we think of the $+1$ and -1 values as determining two subsets

A_1 and A_2 of A. Moving from S to a neighbor is accomplished either by moving one or two elements from A_1 to A_2, or moving one or two elements from A_2 to A_1, or swapping a pair of elements where one is in A_1 and one is in A_2. We can define a *random move* on this state space can be defined as follows. Choose two random indices i and j from $[1, n]$ with $i \neq j$. Set s_i to $-s_i$ and with probability 1/2, set s_j to $-s_j$.

4.1.2 The Pre-Partitioning Representation

Pre-partitioning is an alternative way to represent a solution. We represent a solution by a sequence $P = \{p_1, p_2, \ldots, p_n\}$, where $p_i \in \{1, \ldots, n\}$. The sequence P represents a pre-partitioning of the elements of the original set A, in the following way: if $p_i = p_j$, then we enforce the restriction that a_i and a_j have the same sign. In other words, if $p_i = p_j$, then we consider both a_i and a_j to be in the same subset, either A_1 or A_2.

4.2 The Heuristics

In addition to the Karmarkar-Karp algorithm, we used three heuristics, namely (1) Repeated Random, (2) Hill Climbing and (3) Simulated Annealing, to solve the Number Partition problem. They are briefly described below.

4.2.1 Repeated Random

In this heuristic, we repeatedly generate random solutions to the problem for a predetermined number of iterations (eg: 25,000 iterations). At the end of each iteration, we compare the residue with the best residue so far. If the current residue is smaller than the best residue so far, we update the best residue with the current residue. The most optimal residue is the best residue found so far by the algorithm at the end of all the iterations.

4.2.2 Hill Climbing

In this heuristic, we first generate a random solution to the problem. Then we attempt to improve it through moves to better neighbors for a predetermined number of iterations (eg: 25,000 iterations). At the end of each iteration, we compare the residue with the best residue so far. If the current residue is smaller than the best residue so far, we update the best residue with the current residue. The most optimal residue is the best residue found so far by the algorithm at the end of all the iterations.

4.2.3 Simulated Annealing

In this heuristic, we first generate a random solution to the problem. As in the case of Hill Climbing, we attempt to improve it through moves to neighbors for a predetermined number of iterations (eg: 25,000 iterations). However, the neighbors may not always be better. Here, we selected a *cooling schedule*, which is actually a function T(iter) defined by $\mathrm{T(iter)} = 10^{10}(0.8)^{\lfloor \mathrm{iter}/300 \rfloor}$. This function allows us to probabilistically select the neighbor.

4.3 Implementation of the Algorithms

We implemented the Karmarkar-Karp and the two representations of the three heuristics using Python. As per the specification described in the assignment, the main driver of our code takes three arguments: a flag, an algorithm code and an input file. The algorithm argument is one of the seven algorithms described in Table 1 of the assignment. We randomly generated input files, each containing 100 random numbers chosen uniformly in the range $[1, 10^{12}]$. Each algorithm was iterated a maximum of 25,000 times to find the most optimal residue. We generated 50 random instances of the problem for each of the 7 variants of the implementations, each time using 25,000 iterations to find the optimal solution. In addition to the required functions to implement the algorithms, we also wrote a few utility functions to analyze our results for each of the method.

5 Experimental Results and Analysis

We generated 50 sets of 100 random numbers in the range of $[1, 10^{12}]$ and ran all the heuristics as specified in the assignment. We averaged the results for all seven runs (as per Table 1 in the assignment) and generated, (1) Mean Residue, (2) Mean Runtime, (3) Min Residue, (4) Max Residue, and (5) Standard Deviation of the Residues. We ran the Karmarkar-Karp algorithm 50 times and generated the above statistics. For each of the 50 trials, we ran each algorithm 25,000 times to find the most optimal residues under each category. For each trial, using the randomly generated list of 100 numbers (which we call the sequence A in the Introduction section), we randomly generated a sequence solution and a pre-partition representation. The following table shows the average statistics for the 50 trials.

Algo Code	Mean Residue	Mean Runtime (s)	Min	Max	Std. Dev
0	309783.04	0.00022	4670	1545082	334805.11
1	286604325.52	0.56275	4742684	1452929110	304471726.52
2	300498292.44	0.13917	3494079	1190991645	290581213.24
3	310266360.52	0.25827	54578	1532677765	307759669.38
11	157.92	3.49252	3	1008	181.05
12	212.64	2.94797	0	1107	239.75
13	217.48	3.07041	1	877	203.96

5.1 Residue Values

As we can see from the above table, the pre-partitioned representation of the solution leads to a significantly more optimal mean residue for the three heuristics, namely repeated random, hill climbing and simulated annealing. In fact, the mean residues for the pre-partitioned representations are almost 10^6 times smaller than their standard representation counterparts. The lowest mean residue was obtained using the pre-partitioned repeated random algorithm. The pre-partitioned hill climbing and simulated annealing came in close second and third in terms of the mean residue. The pre-partitioned version of the three heuristics also discovered

the near perfect partitioning as their minimum residue values are 3, 0 and 1 respectively for these algorithms. The standard deviations for the pre-partitioned versions are also orders of magnitude smaller than their standard representation counterparts.

The reason why the pre-partitioned representation of the three heuristics performed significantly better than their standard representation counterparts is that a pre-partitioned representation already merges one or more pairs of elements in the set, thereby effectively moving elements into two separate subsets A_1 and A_2 even before the actual heuristic is invoked. This pre-merging of elements from the initial set A gives a significant advantage for the heuristics to continue to partition the elements further. In fact the mean residues obtained for the three pre-partitioned heuristics were 10^6 times smaller than their respective standard representation counterparts.

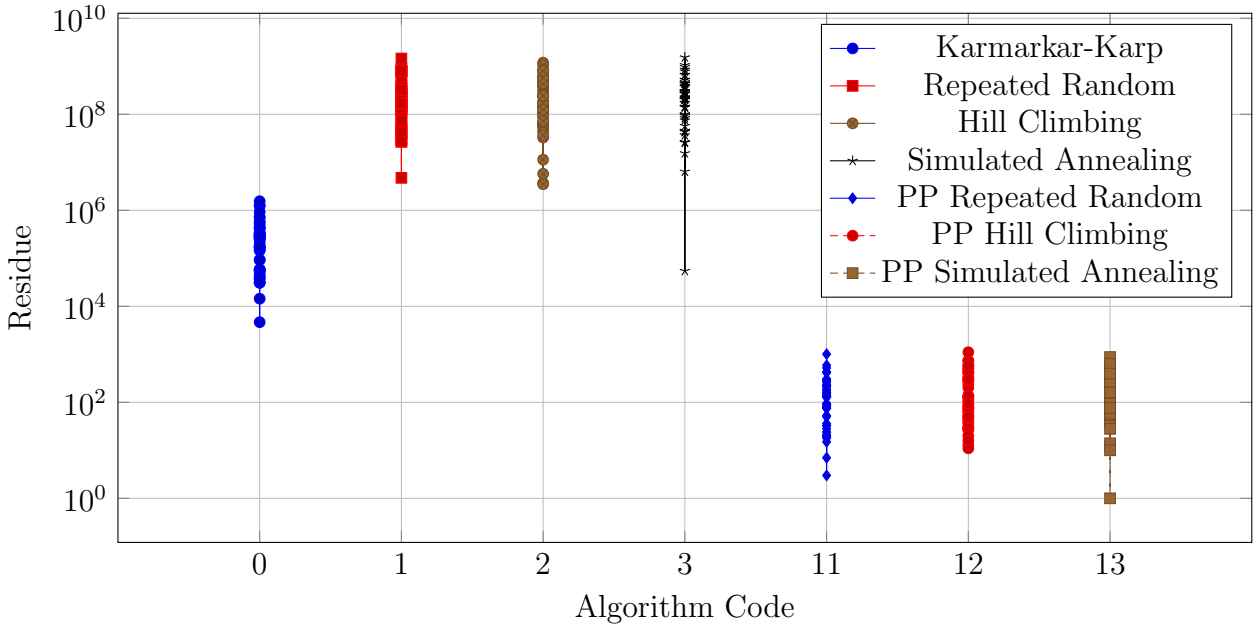
Among the three heuristics, we anticipated that the Simulated Annealing algorithm would produce the smallest residue, followed by the Hill Climbing Algorithm and then by the Repeated Random algorithm. However, we found that the Repeated Random algorithm was the most effective under the pre-partitioned representation for the solutions than the other two. The likely reason for this is that the Repeated Random algorithm was able to generate many different varied sets of pre-partitioned representations, which helped it to lower the residue better than the other two heuristics. Our results also showed that the pre-partitioned version of the Hill Climbing algorithm performed better than the pre-partitioned version of the Simulated Annealing. However, we were expecting the Simulated Annealing to perform better than the Hill Climbing algorithm, as the Hill Climbing algorithm can get stuck in local minima, whereas the Simulated Annealing algorithm can move to different positions with varying probability with the help of the cooling schedule. The likely reason for this experimental results is that the $T(\text{iter})$ function that we used for the cooling schedule may not have been sufficiently optimal.

5.2 Runtimes of the Algorithms

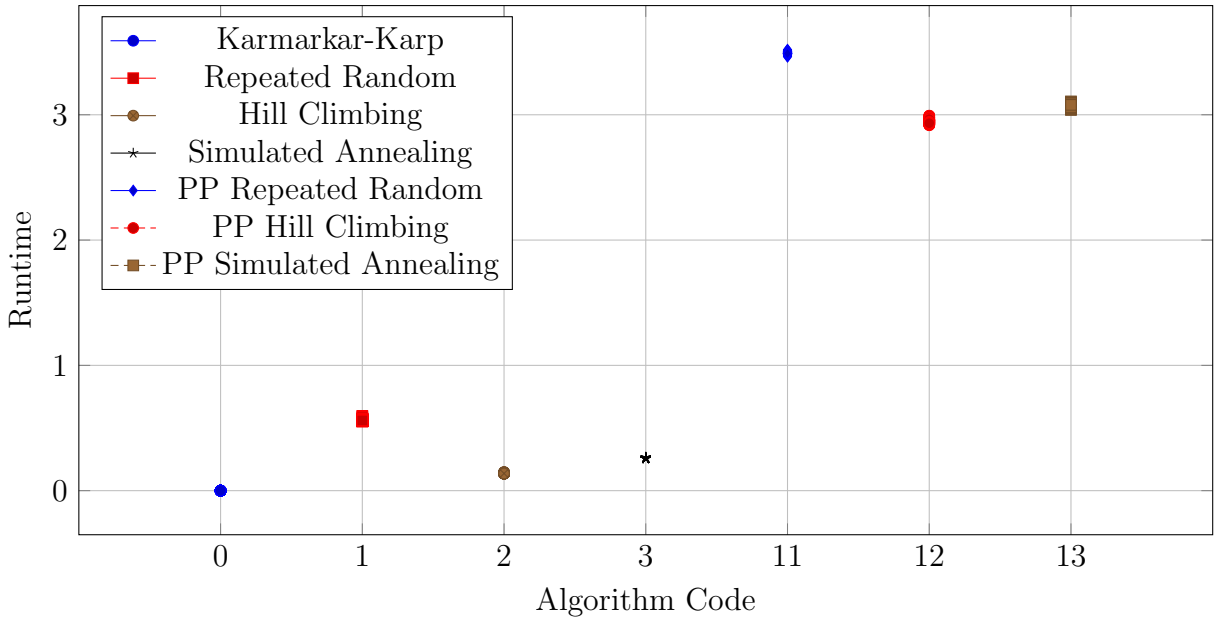
The runtime of Karmarkar-Karp was the lowest, as it has a runtime of $O(n \log n)$ using a Max Heap. The runtimes of the standard representation version of the three heuristics are significantly lower than that of their pre-partitioned counterparts. The increased runtime of the pre-partitioned versions are due to the fact pre-partitioning does not directly produce a solution. After finding a pre-partition, the algorithm still needs to run the Karmarkar-Karp algorithm to find the final residue value. Therefore, we found from our experimental results that all three pre-partitioned algorithms took more than 10 times to find the residues compared to their standard representation counterparts.

Note: The Appendix section includes a full set of 50-runs of the seven heuristics and their mean residue values.

We have used scatter plots to display both the residue ranges and runtimes in the next two graphs. The y-axis for the first plot (Residue versus Algorithm) is on a logarithmic scale in order to provide a better visual comparison. The plot shows that the pre-partitioned versions lead to significantly lower residue values than their standard representation counterparts. The scatter plots of the individual algorithms help visualize the range of the residue values.



We also see from the plot below (Runtime versus Algorithm) that the pre-partitioned versions take significantly longer runtimes than their standard representation counterparts.



5.3 KK as a Starting Point for the Randomized Algorithms

The solution to the Karmarkar-Karp algorithm can be used as a starting point for the randomized algorithms such as Hill Climbing and Simulated Annealing. This is because both the Hill Climbing and the Simulated Annealing algorithms iteratively decrease the value of the residue until the most optimal residue is found. Therefore, starting from an improved solution, which was already computed by KK, will have higher likelihood of finding a more optimal residue than otherwise. This approach will also likely reduce the runtime of the heuristic algorithm itself (compared to the runtime of the original heuristic algorithm), although there will be an additional $O(n \log n)$ runtime (using a Max Heap) introduced by the initial KK run. In fact, the pre-partition representation uses an intermediate solution from the KK algorithm, which is why it results in a significantly more optimal residue than the standard representation.

Unlike the Hill Climbing and the Simulated Annealing algorithms, the Repeated Random algorithm simply generates a new solution each time. Therefore, the Repeated Random algorithm is unlikely to benefit by using KK as a starting point.

6 Conclusion

In this programming assignment, we implemented a number of heuristics for solving the Number Partition problem, which is known to be NP-complete. We first implemented the Karmarkar-Karp algorithm using a Max Heap that has a runtime of $O(n \log n)$. In addition, we implemented three heuristics, namely, (1) Repeated Random, (2) Hill Climbing and (3) Simulated Annealing. We used two different representations for the solutions of these heuristics - the first one was the standard representation, which is just a sequence of $+1$ and -1 values, and the second one was the pre-partitioning representation, which groups two or more elements into the final partitions, and thereby has the potential to significantly improve the quality of the solutions.

Our experiments showed that the pre-partitioned representations of the three heuristics performed significantly better than their standard representation counterparts by many orders of magnitude. The reason why the pre-partitioned representation of the three heuristics performed significantly better than their standard representation counterparts is that a pre-partitioned representation already merges one or more pairs of elements in the set, thereby effectively moving elements into two separate subsets A_1 and A_2 even before the actual heuristic is invoked. The randomized algorithms provided further advantage by allowing better input solutions to be selected for iteratively selecting lower residues.

Even though the residues generated by the Karmarkar-Karp algorithm were better than those generated by the three heuristics using the standard representation, its mean residue was 10^3 times larger than the three heuristics using the pre-partitioned representation. Due to the deterministic nature of the Karmarkar-Karp algorithm, it could not always identify the optimal residues for random input sets. This explains why the pre-partitioned randomized algorithms were able to outperform the Karmarkar-Karp algorithm.

7 Appendix

Following is a set of results collected from another experiment of 50 runs of the seven algorithms. Note that Algo 0 = Karmarkar-Karp, Algo 1, Algo 2, and Algo 3 are standard representation of Repeated Random, Hill Climbing and Simulated Annealing respectively, and Algo 11, Algo 12, and Algo 13 are pre-partitioned representation of Repeated Random, Hill Climbing and Simulated Annealing respectively.

Run	Algo 0	Algo 1	Algo 2	Algo 3	Algo 11	Algo 12	Algo 13
1	300828	321330336	138785656	230400888	404	584	46
2	31182	134242124	57391418	490527826	420	8	94
3	30815	95471559	397198019	260715779	843	47	385
4	54749	37577307	375700115	543297891	877	407	635
5	342601	79115229	266734935	59498383	45	165	1341
6	177307	19908341	504950329	613037987	17	289	111
7	58387	1097409043	129454819	440084953	137	47	173
8	93323	283655879	7799211	672222957	393	57	173
9	1268060	479496870	102944098	321624430	164	574	254
10	269370	433104898	131971258	609162822	100	268	20
11	255440	34623516	51964684	415760646	308	48	0
12	486009	296548959	432978705	442770201	15	157	65
13	1226603	71768987	546361363	292403635	99	23	355
14	38530	594502216	46545030	239735172	452	44	136
15	589289	208388483	32288069	237918873	25	167	633
16	168393	56142437	297685383	269501985	43	91	47
17	425645	433845043	57819271	576035	31	159	907
18	303099	185857513	88357309	127109953	199	133	45
19	441708	520443658	298091664	337774578	118	232	16
20	4670	50657504	21791210	254934690	24	10	198
21	14391	257093235	80037441	337752099	193	681	177
22	146623	268820233	405328675	1277953473	135	3	269
23	246434	721496212	305132646	353612294	74	186	510
24	413963	248176749	327405555	16599049	189	165	361
25	55428	547293532	164553604	309279354	136	312	24
26	333927	77816769	75442709	519316677	225	65	135
27	283632	427501502	99090536	25053964	34	194	144
28	550998	24510130	558601862	405403532	92	8	366
29	58387	7649327	756294803	176064567	455	397	325
30	259454	363478318	230279698	163597594	48	80	22

Run	Algo 0	Algo 1	Algo 2	Algo 3	Algo 11	Algo 12	Algo 13
31	147671	378881481	791889729	673799187	179	15	7
32	165812	140531834	20304748	171993504	236	30	136
33	212653	151248603	65097909	311317697	231	47	451
34	56913	581863775	154554115	268969	31	179	207
35	323299	654347689	35038341	128760533	107	325	61
36	1545082	67257110	183498248	29011414	408	6	104
37	729069	212058619	456559531	90636401	115	285	123
38	47177	17730221	366621867	316542043	43	45	439
39	42592	122455600	135975182	846187418	26	40	156
40	263295	89328247	604519957	440541463	333	71	11
41	92182	165477908	112710914	155127832	32	204	54
42	38240	306641010	583016520	781593564	254	222	90
43	714966	180889964	133261522	3367302	380	70	190
44	173826	194679124	187283648	9317752	54	110	14
45	314036	3653754	1058438322	225190870	30	52	228
46	273524	19273412	353917712	317735424	204	24	174
47	917863	127158527	460920043	694467589	5	501	33
48	291776	347342200	386924464	21104920	86	166	26
49	31844	5386328	76066252	203496040	184	158	94
50	178087	16686849	612501345	248538629	79	33	735

The following is the mean residues of the seven algorithms for the above experimental results:

	Algo 0	Algo 1	Algo 2	Algo 3	Algo 11	Algo 12	Algo 13
Mean	309783	243216363	275361608	322253816	186	163	226