By: Kahunui Foster and Kate Schember

Part I: Dynamic Programming

To solve the Number Partition problem with Dynamic Programming, we require the use of two one-dimensional arrays. The first array, *X*, will indicate how evenly the partitions can be split, and the second array, *P*, will indicate which numbers from the given set *A* are in each of the two partitions.

We can define *X* [*i*] as the boolean indicating whether the sum *i* is possible to achieve with the numbers in *A*. *X* will be of size $\lfloor b/2 \rfloor$, so *i* ranges from 0 to $\lfloor b/2 \rfloor$. At each index *i*, the value will be *True* if some subset of *A* can produce the sum *i*. The goal is to find the *i* closest to $\lfloor b/2 \rfloor$. This is representing the idea that if partitions are equal, they will each sum to *b/2*, so we look at one subset of *A* and see how close to *b/2* that sum can be. This largest *i* will give the optimal sum of one partition, and therefore the residue would be $\lceil b/2 \rceil$ - *[largest i]*.

To recurse through this program, go through each member *a* in *A*. Since setting *a* as its own partition would give a total sum *a*, this is a possible sum that should be recorded in *X*, so set *X [a] = True*. Then, we want to see what other sums we can achieve using *a*. To do this, check every index of *X* from 0 to *a*. If that index *i* is *True*, set *X [a + i] = True* to indicate that if you put *a* in a partition with other members in *A* that were already checked, you could get a new sum *a + i*. As a base case, *X [0] = True* because if one partition is defined as no members of *A* then the sum of that would be 0. This can be formulated in the recurrence below:

$$X[i] = \begin{cases} True & \text{if } i \in A \text{ OR } \exists\ j \in A \text{ such that } X[i-j] = True \text{ OR } i = 0 \\ False & \text{otherwise} \end{cases}$$

Following this recursion, run through all elements in *A* and then get the *i* closest to $\lfloor b/2 \rfloor$. Call this *i* the solution *s*.

Once we know how large the partition should be to be optimal, we then need to know which exact numbers are in each partition. This is where we use the second array. Define *P [i]* as a number *j* that is an element of *A* that can be added to some other number to sum to *i*. This array should be filled while *X* is being filled, and will be the same size. If there are multiple ways to achieve the same sum, this 1-D array can hold linked lists with all the options. For every number *i* where X [i] = *True*, the number *j* that allowed *X [i - j] = True* should be the *j* that is set as *P [i] = j*. This is representing the idea that we want to keep track of what number from *A* allowed the sum *i* to be achieved. This can be formulated in the recurrence below:

$$P[i] = \begin{cases} j & \text{if } j \in A \text{ and } i = x + j \text{ for some nonnegative integer } x \\ \infty & \text{otherwise – to show that this is not a possible sum } i \end{cases}$$

Thus, to find all the numbers in one partition, when *X* is full and the solution *s* has been found, calculate *P [s].* Let the result of *P [s]* equal some number *j*. Based on our definition, *j* should be an element of *A*, and therefore is included in the first partition. Then, to continue finding the rest of the elements in the partition, split apart the leftover number added to *j* to get *s*. This is done by calculating *P [s - j].* Continue this pattern, taking note of each resulting number *j*, until the value *s - j = 0*. At this point, you know that you have taken apart the original sum *s* to be only elements of *A*, and therefore have found all the elements in the first partition. Putting together all the *j*'s you found during this process gives the first partition. The remaining elements of *A* are in the second partition.

The space complexity of this will simply be *O(b)* because both arrays will be of size $\lfloor b/2 \rfloor$, reducing to *O(b)*. The time complexity will be *O(nb)*. When filling *X*, you will need to go through all *n* elements in *A*, and you will need to go back and check the sum with at most *b/2* other sums that are *True* in *X*. *P* will be the same complexity because you will only add to *P* as often as you add to *X*, and retrieving the final partitions will always take less time than filling the array, so that won't add to the overall time complexity. Putting these procedures together, the total time complexity is *O(nb)*, a polynomial time.

Part II: Karmarkar-Karp Time

Karmarkar-Karp can be solved in *O(n log n)* steps if it is solved using a heap. If a Max-Heap is implemented, it is very fast to extract the two largest numbers and also to insert the difference of those numbers, and therefore is an efficient method. To create a max heap, the function *Build-Heap* can be used, which takes *O(n log n)* steps. Then *Extract-Max* can be used twice on every iteration of Karmarkar-Karp to get the two largest numbers from that heap. Since each call to *Extract-Max* takes *O(log n)* time, this total process will take *O(2n log n)*. The last step is to insert the difference between the two largest numbers extracted, which will be done no more than *n* times, so this will take a total of *O(n log n)* steps. Adding all these times together still gives *O(n log n)* time.

Part III: Implementation and Analysis

We implemented these algorithms entirely in C, using arrays of *long long int*s to allow for enough space for 64-bit numbers. The functions are relatively straightforward given the assignment directions and pseudocode. One of the most challenging issues with C that we had to address was generating random numbers between 1 and 10^{12}, since there C *rand()* function is not always entirely random and also only generates 32-bit numbers. In order to fix this issue, we used the commonly effective *srand(time(NULL))* seeding for the random generator and used bitwise operators to transform a random 32-bit number to a 64-bit number no larger than 10^{12}. All code is attached in the "kk" file.

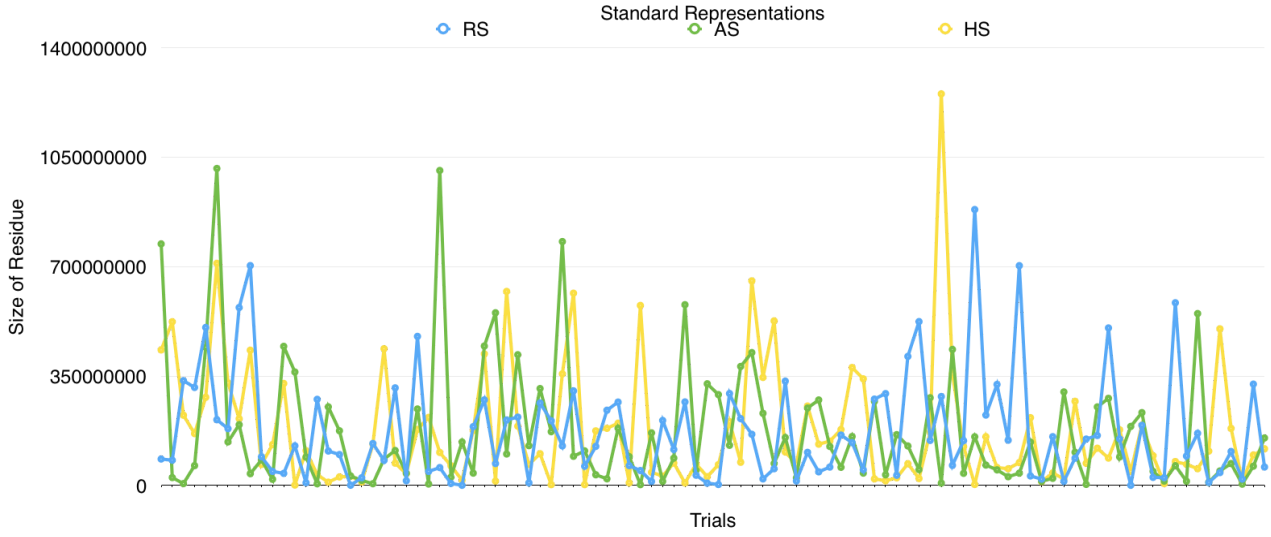
Our numerical results of all the tests are displayed in the graphs and tables below:

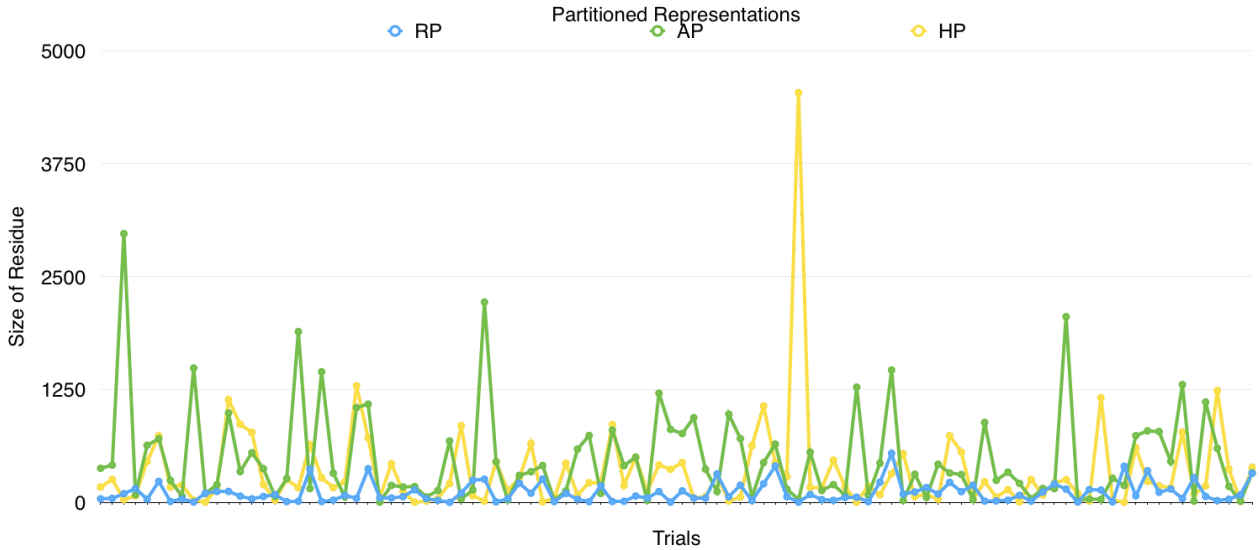
Average Residues

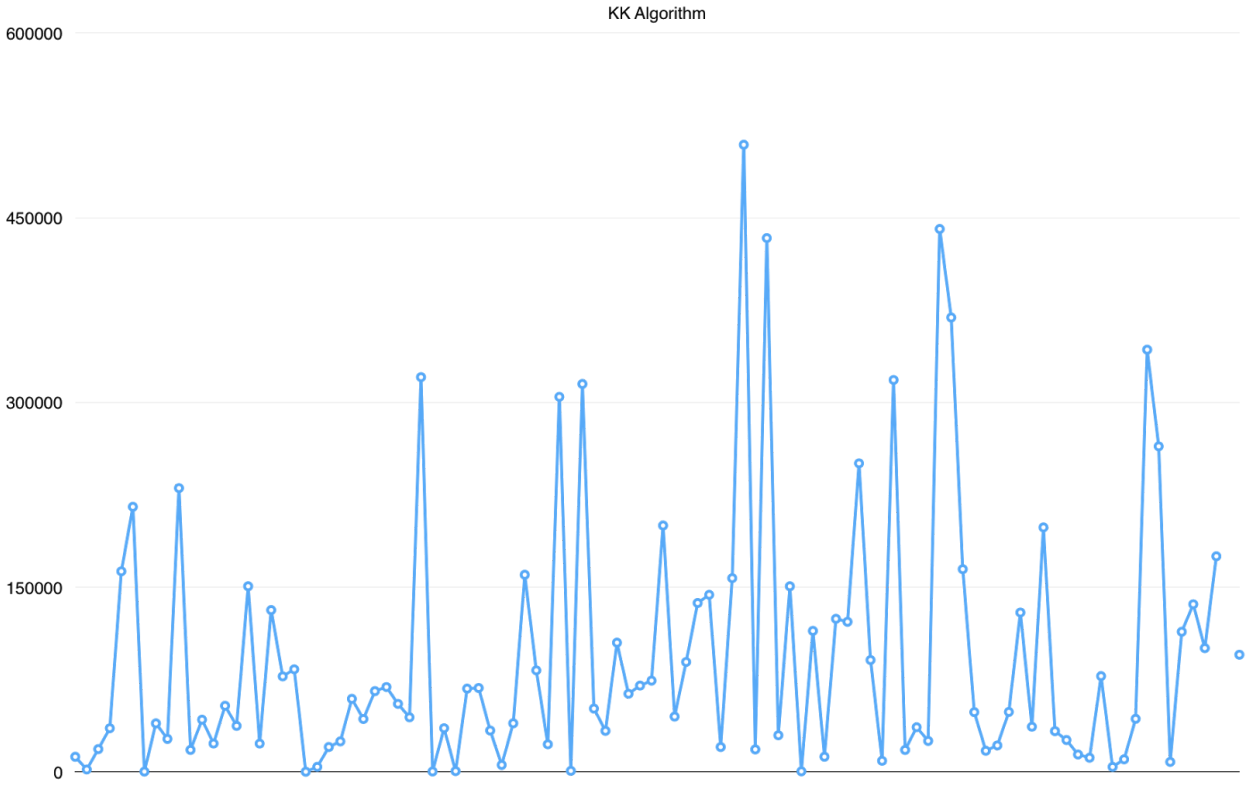
RS	**172557216.1**
RP	**107.48**
HS	**177055075.36**
HP	**340.82**
AS	**175203957.08**
AP	**483.96**
KK	**95257.08**

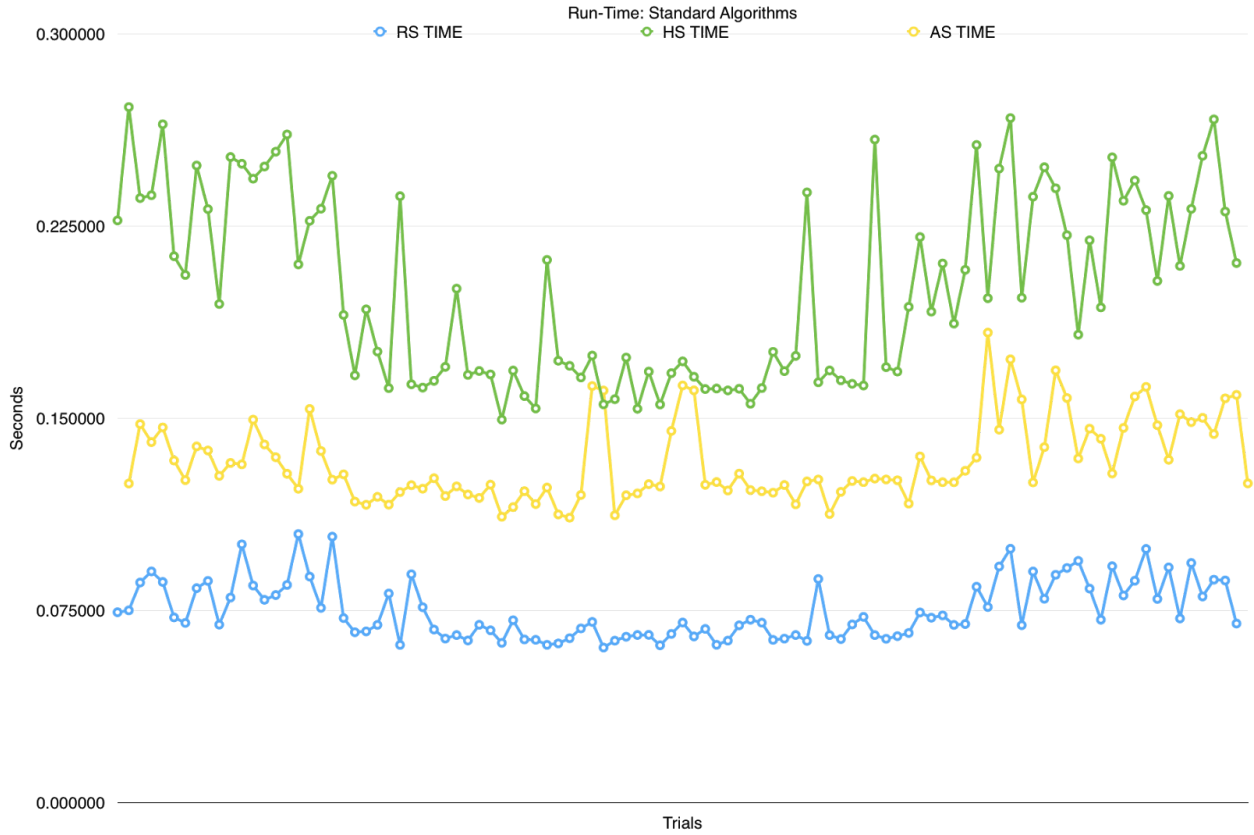
Average Times

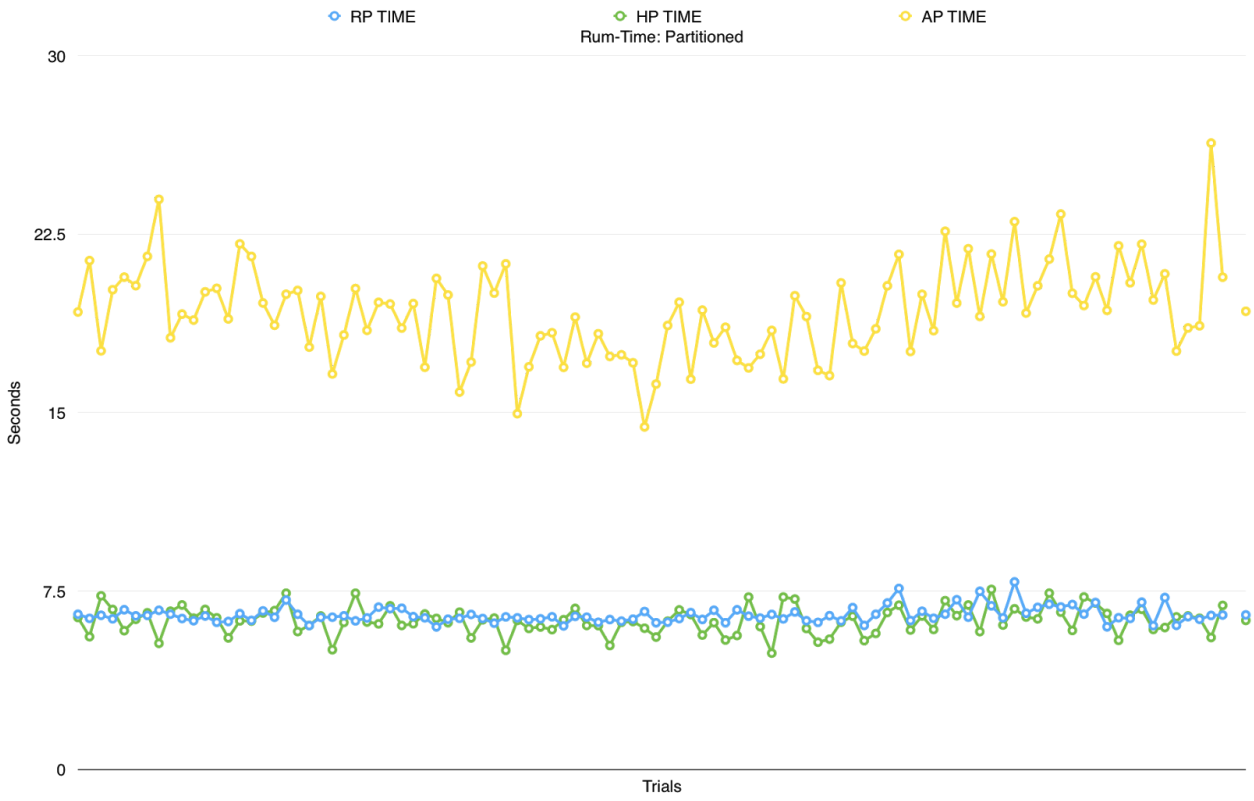
RS	**0.1329106**
RP	**6.49420371**
HS	**0.07498933**
HP	**6.2637493**
AS	**0.20089256**
AP	**19.26002056**
KK	**0.00033851**

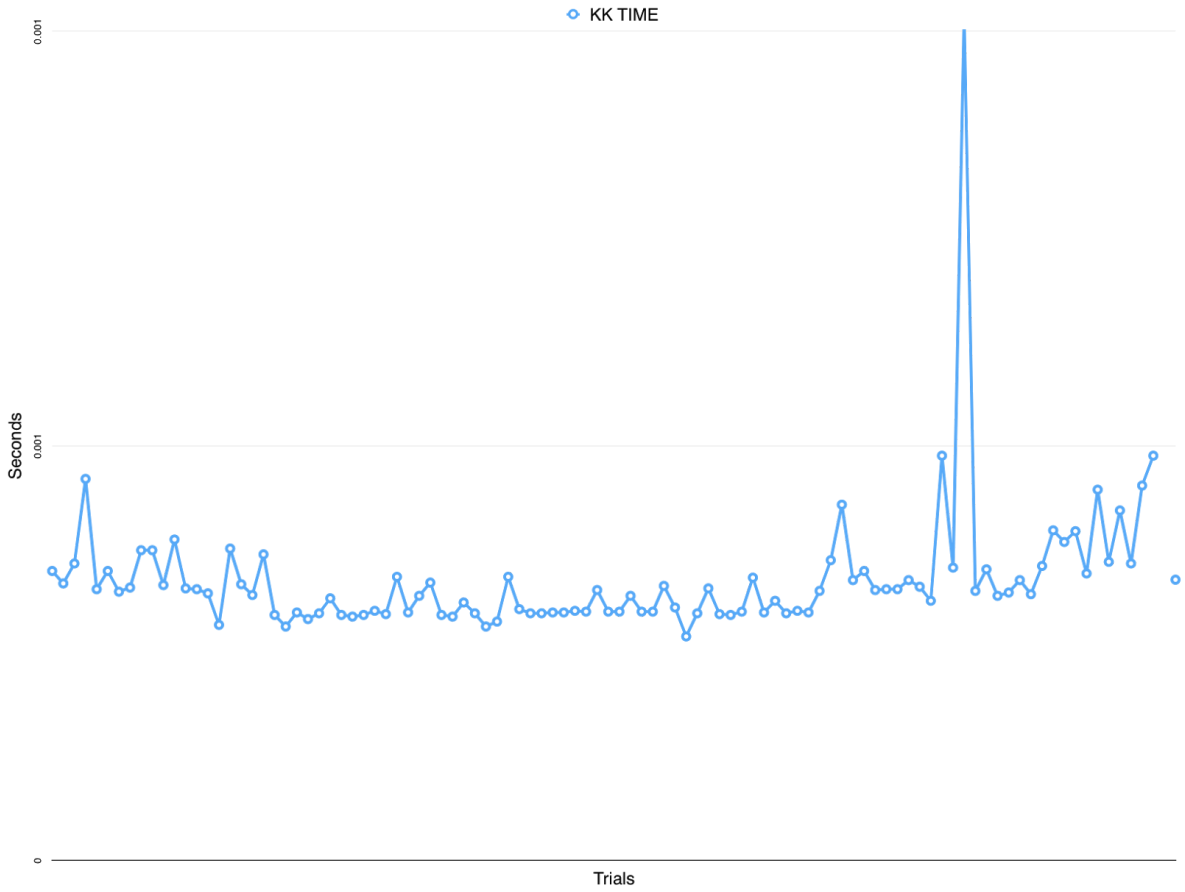





KK Algorithm
600000
450000
300000
150000
0


Run-Time: Standard Algorithms
RS TIME
HS TIME
AS TIME
0.300000
0.225000
0.150000
0.075000
0.000000
Seconds
Trials


RP TIME
HP TIME
AP TIME
Rum-Time: Partitioned
30
22.5
15
7.5
0
Seconds
Trials


KK TIME
0.001
0.001
0
Seconds
Trials

The most important take-away from these tests is that the standard representation gives very inaccurate results and the prepartitioning is always effective. This makes sense and was mostly expected, since standard does not use Karmarkar-Karp. Our tests showed that Karmarkar-Karp is quite a good approximation – the residues were within the 100,000s and 10,000s, which is relatively low, and it took by far the least amount of time to run. It follows, then, that the algorithm that computes residues without the use KK would be worse. Standard is not able to benefit from the advantages of KK, and the consequences of that are obvious in our data, as Standard residues all linger in the 9-digit numbers.

Prepartitioning was vastly better than standard for all three heuristics because it did use KK, consistently producing residues in the hundreds. This, as we predicted, is because prepartitioning is able to take the already reasonable residues from KK, and continually try to improve them. With so many iterations, prepartitioning is able to get all the way down to a 2- or 3-digit residue.

Another important advantage of preparitioning is that it has flexibility to partition the numbers into *n* groups. As explained in the assignment, prepartitioning is like forcing numbers into the same group, whereas standard is splitting numbers into two separate groups. Having this flexibility for prepartition could be advantageous for general uses of computing the number partition problem.

On the other hand, however, standard has the advantage of running very quickly. While all standard algorithms took well under 1 second to partition 100 64-bit numbers, the prepartition algorithms took anywhere from 5 to 20 seconds each. We guess that this is because prepartition has the flexibility explained above. Standard has a fixed number of partitions to break into, but prepartition has to be able to partition into as many as *n* groups without being fixed. While this is helpful for correctness, it adds a lot of complexity to prepartition that takes a significant amount of time.

Within each representation, the heuristics produced surprisingly similar results. For standard, hill climbing was the worst and repeated random was the best. It is logical that simulated annealing would be better than hill climbing in this situation because hill climbing only looks for the locally best moves, whereas simulated annealing ensures that the algorithm will never get stuck in a local minimum residue and stop improving. Because annealing is constantly making new moves, despite whether it looks beneficial at that step. We were not expecting repeated random to be best, but it is not unreasonable that generating random numbers, if done on 25,000 iterations, would eventually find a better solution. Overall, however, each of the heuristics for standard were very similar.

For prepartitioning, repeated random was the best of the three heuristics again, most likely for the same reason as for standard. This time, annealing was slightly worse than hill climbing, which was a bit surprising, but since each of these residues are so low, they are not extremely different. Overall, it is clear that running KK is by far the most effective way to approximate the

number partition problem, and when run on many iterations, randomly guessing solutions can be very effective.

Part IV: Karmarkar-Karp and Randomized Algorithms

Karmarkar-Karp could be used in each of the randomized algorithms to help provide a possible residue that could be compared against. Understanding that Karmarkar-Karp is a reasonably accurate algorithm to find residues, than using KK as a starting point for randomized algorithms could potentially provide a better base case to try to improve. From our data, it is clear that the standard representation algorithms were far less accurate than Karmarkar-Karp, suggesting that using KK as a starting point could be quite advantageous for these.

The randomized algorithms could use the result from Karmarkar-Karp as a replacement to a random solution – instead of comparing random solutions to another random solution, compare random solutions with KK, and see if it is possible to improve the residue from there. Since the standard representations produced average residues far larger than KK, then starting with Karmarkar-Karp would reduce the residue from a value around 10^9 to around 10^6, which is a very significant improvement.

Using Karmarkar-Karp as a starting point for any of the randomized algorithms with the prepartitioning representation would have a smaller effect. On average, all the prepartitioning residues were significantly lower than the residue from KK, meaning that even if we started these algorithms with KK, that the algorithms would almost certainly find a better residue anyway, so it wouldn't be particularly advantageous.