Number Partition Problem

CS 124 - Sabrina Hu & Smriti Somasundaram

4/19/23

No. of late days used on previous psets: 9

No. of late days used after including this pset: 10

1 Introduction

In this assignment, we evaluate a number of different heuristics to solve the np-complete Number Partition Problem. For this problem, we take a sequence A or nonnegative integers and output a sequence S composed of signs $s_i \in \{+1, -1\}$ such that the residue $u = |\sum_{i=1}^{n} s_i a_i|$ is minimized.

In our assignment, we first give a dynamic programming solution to this problem.

Then, we evaluate several deterministic heuristics for the problem, including the Karmarkar-Karp algorithm, as well as the "repeated random", "hill climbing", and "simulated annealing" heuristics with both pre-partitioning and no pre-partitioning.

We discuss our findings and results below.

2 Dynamic Programming Solution

2.1 Algorithm

Let us first give a DP approach to solve this problem.

To set up, consider the interpretation of the number partition problem where we are looking to find how we split our sequence A up into two subsets A_1 and A_2 such that we minimize $|\sum A_1 - \sum A_2|$, where $\sum A_1$ and $\sum A_2$ are the sum over the elements of A_1 and A_2, respectively.

To find our minimum residue, let us consider when the residue is 0. This happens when we have $\sum A_1 = \sum A_2 = B$, where $B = \frac{\sum A}{2}$. To approach this problem, we can consider that if we can find a subset of elements of A that sums to B, we will have a residue of 0. Otherwise, we want to find the greatest possible value less than B that a subset of elements of A can sum to; by maximizing this value, we can minimize the difference in sums of the two subsets A_1 and A_2 and thus minimize our residue.

2.1.1 Definition of Subproblems

Let $A = (a_1, a_2, ..., a_n)$.

Let us define our subproblem to be whether or not a subset of $(a_1, ..., a_i)$ can sum to value j, where j increments from 0 to $C = \lfloor B \rfloor$.

Then, we can set up a DP table with $n+1$ rows and $C+1$ columns. We define each entry of the table, $dp[i][j]$, to be 1 if a subset of the first i elements of A (where i goes from 0 to n) can sum to value j, and 0 otherwise. In other words:

$$dp[i][j] = \begin{cases} 1 \text{ if a subset of the elements up to the i-th element can sum to j} \\ 0 \text{ otherwise.} \end{cases}$$

2.1.2 Recurrence Relation

Then, let us define a recurrence relation to find $dp[i][j]$. Notice that $dp[n][C]$ is 1 if and only if there exists a subset of A that sums to C, and thus tells us that the minimum residue is 0 (or 1 if $\sum A$ is odd).

We notice that

$$dp[0][0] = 1,$$

since there must exist a subset of the first 0 elements that sum to 0.

For our recurrence relation, we have:

$$dp[i][j] = \begin{cases} 1 \text{ if } dp[i-1][j] = 1 \text{ or } dp[i-1][j-a_i] = 1 \\ 0 \text{ otherwise.} \end{cases}$$

First, $dp[i][j] = 1$ if $dp[i-1][j] = 1$ because, if we can find a subset of the first $i-1$ elements that sums to j, then we can clearly also find a subset of the first i elements that sums to j (ex: we simply use the same subset).

Additionally, $dp[i][j] = 1$ if $dp[i-1][j-a_i] = 1$ because if a subset of the first $i-1$ elements of A sum to $j - a_i$, we can add a_i to that subset to get a sum of j with i elements.

2.1.3 Final Steps

Using the above recursive rules, we can fill out our dp table for all $i \in [0, n]$ and $j \in [0, C]$.

Then, if $dp[n][C] = 1$ when $\sum A$ is even, our minimum residue is 0, so our algorithm returns 0. If $dp[n][C] = 1$ when $\sum A$ is odd, our minimum residue is 1, so our algorithm returns 1. Otherwise, our algorithm keeps iterating down from $j = C$ to $j = 0$ until it finds a j such that $dp[n][j] = 1$. This allows us to maximize the sum of one of our subsets (below C) to minimize the residue. When it does find such j, we record that j as j_{max}.

Then, our algorithm returns $2(C - j_{max})$ as our minimum residue for our final answer if $\sum A$ is even, and $2(C - j_{max}) + 1$ as our minimum residue if $\sum A$ is odd. This is the residue because if a subset of elements of A sums to j_{max}, the other subset of elements sums to $\sum A - j_{max} = 2C - j_{max}$. Then, the difference between these two sums is $2(C - j_{max})$, which gives us our minimum residue as we seek to maximize j_{max}.

2.2 Runtime Analysis

We see that our algorithm is essentially the construction of a $(n+1) \times (C+1)$ table, where $C = \lfloor \frac{b}{2} \rfloor$ where b is the sum of the terms of A.

We see that the calculation of each entry of our table is $O(1)$ time, so to calculate approximately $n \cdot \frac{b}{2}$ entries takes $O(nb)$ time, so our DP algorithm has a time complexity of $\boxed{O(nb)}$.

3 Karmarkar-Karp Implementation

The Karmarkar-Karp algorithm is a deterministic heuristic for solving the number partition. The algorithm uses differencing, where it takes two elements a_i and a_j and takes the difference $|a_i - a_j|$. Then, the algorithm replaces the larger of a_i and a_j with $|a_i - a_j|$ and the smaller with 0. Repeating this process until there is one value left, that leftover value serves as a heuristic for the minimum residue.

For our implementation, we wanted to find a $O(n \log n)$ implementation of Karmarkar-Karp. To do so, we used a binary max-heap to store our sequence values.

For our implementation, we followed the following steps:

1. Add all values of our sequence A into a maxheap (which we made by using negative values in Python's minheap). We initialized the maxheap by heapifying our values
2. Then, we popped from the heap twice, extracting the two largest values in the sequence. Popping values automatically re-heapifies the heap so that it maintains its structure.
3. Finally, we push the difference $a - b$ into the heap. This automatically gets rid of one element (similar to setting the smaller element to 0) and replaces the larger element with this difference.
4. Repeat this process $n - 1$ iterations. At the end of $n - 1$ iterations, there will be only one value left in our heap (since each iteration we get rid of one value), and that value will be returned as our minimum residue.

Evaluating the runtime of this implementation, we see that first, each maxheap operation is $O(\log n)$ time for a sequence A of size n. Then, we have $O(n)$ iterations of our process, which contains three $O(\log n)$ maxheap operations, so the total runtime of our implementation is $O(n \log n)$.

4 Discussion

4.1 Implementation of Other Heuristics

Besides the Karmarkar-Karp algorithm, we also evaluating three other heuristics, with both pre-partitioning and no pre-partitioning. We implemented all of our algorithms with Python3, and wrote three main helper functions for our other heuristics, including *calc_residues*, *generate_solution*, and *generate_neighbor*. These three functions, which we wrote variations of for the pre-partitioning case, perform the precise function their names describe. Respectively, calculate residues calculated the residues of a given solution, generate solution generates a random solution using Python's random library and "choice" function, and generate neighbor generates a neighbor of a solution once again using Python's random library as described in the assignment description. We implemented all three heuristics based on the pseudocode given in the assignment description.

4.1.1 Repeated Random

For the repeated random heuristic, we repeatedly generate random sequences of signs as our solution, replacing the current solution if the residue from the randomly generated solution is less than the current residue. We repeat this process for our max iterations and return the final solution that gives the minimum residue.

4.1.2 Hill Climbing

For the hill climbing heuristic, instead of generating a random solution every iteration, we generate random neighbors of the current solution. So, we start with a random solution, and then generate a random neighbor, and replace the current solution with the neighbor if it gives a smaller residue. We repeat this process for max iterations as well to get our final solution.

4.1.3 Simulated Annealing

Simulated annealing is similar to hill climbing, in that it starts with a random solution and generates random neighbors, replacing the current solution with the random neighbor if that residue is smaller. However, it also will replace the current solution with a random neighbor even if it has a greater residue for a certain probability (calculated with the cooling function). Once again, we repeat this process for max iterations as well to get our final solution.

4.1.4 Pre-partitioning

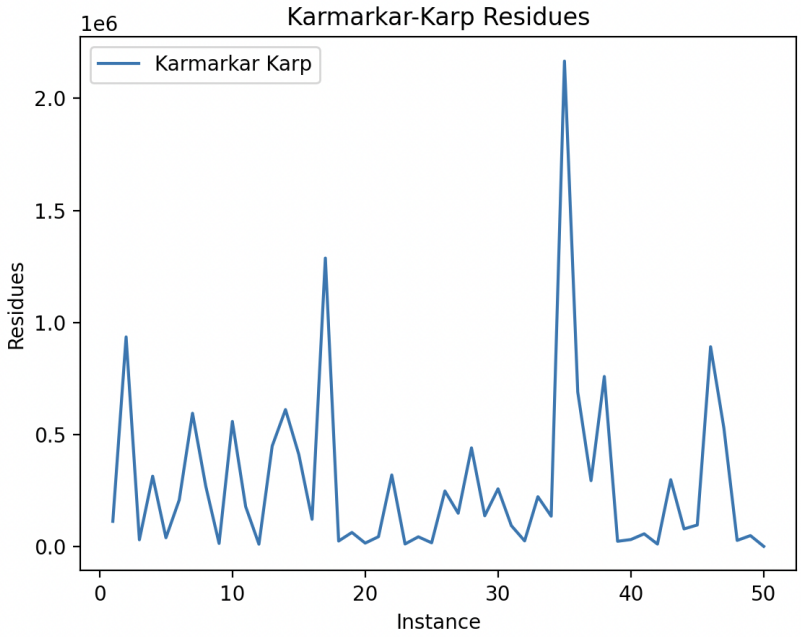
Beyond our simple sequential random solution, where each value is either -1 or +1, we also introduce a new heuristic of pre-partitioning. In this heuristic, instead of generating a random sequence of signs, we generate a random partition, where we have a sequence of numbers where each number is randomly selected from 1 to n. Then, when we generate a random neighbor, we randomly select an index in our sequence to assign to a new partition. Then, to calculate residues we create a new sequence A' that sums all the values in a specific partition, and run the Karmarkar-Karp algorithm.

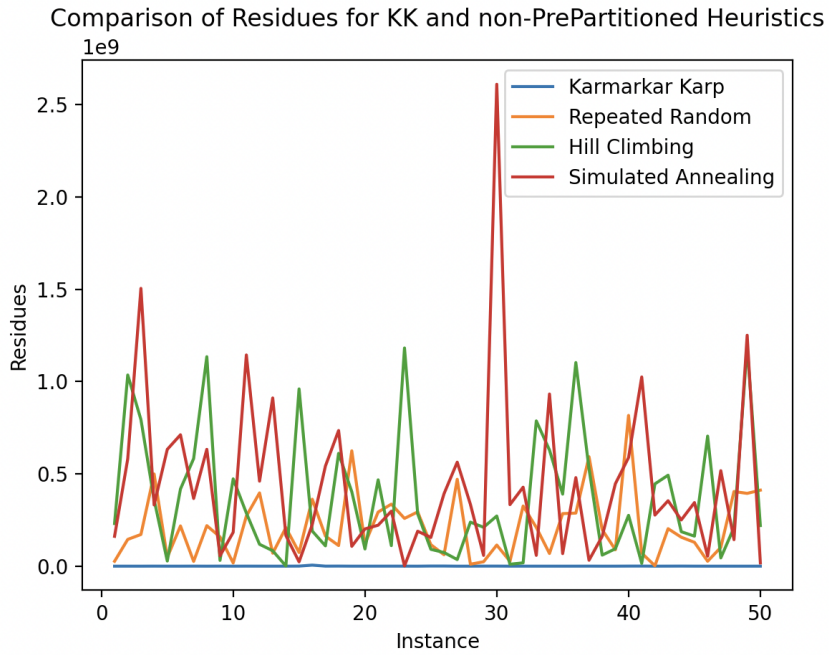
4.2 Results

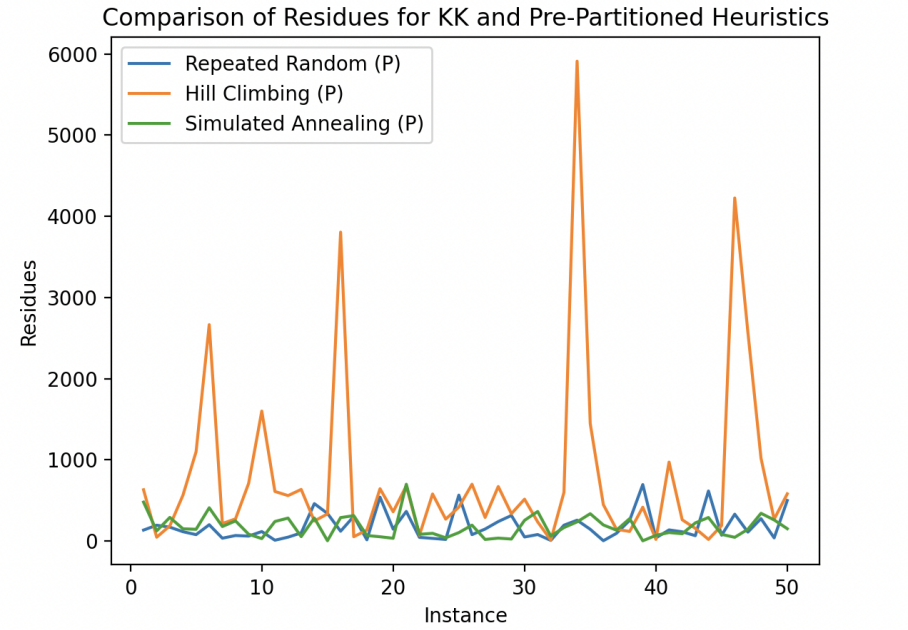
4.3 Comparison of Heuristics

To test our heuristics, we generated sequences of 100 integers, from 0 to 10^{12}, and tested each heuristic over 50 random instances. The graphs below compare the results of calculated residues and the runtimes among the different heuristics. We generated these graphs with numpy and matplotlib.

4.3.1 Residues







Residues for Heuristics, Averaged over 50 Random Instances

Algorithm	No Pre-Partitioning	Pre-Partitioning
KK	2.37×10^5	N/A
RR	2.30×10^6	242.9
HC	2.67×10^7	833.7
SA	7.36×10^7	163.5

We see that among the non pre-partitioned cases, Karmarkar-Karp performs the best in terms of minimizing residues. After Karmarkar-Karp, Repeated Random is the best, followed by Hill Climbing and Simulated Annealing. It makes sense that Karmarkar-Karp performed the best without pre-partitioning among these heuristics, because Karmarkar-Karp actually starts with the initial sequence and reduces the sequence to a residue by differencing. Because this algorithm takes the two largest elements and replaces with their difference, this is synonymous with simply putting them in two different subsets. This algorithm is not randomized but rather deterministic based on the input sequence. Thus, it makes sense that it is the most accurate. Among the other three heuristics, it also makes sense that the repeated random algorithm performs better than hill climbing and simulated annealing. With the repeated random algorithm, we go with the best randomly generated solution out of 25000 iterations. However, with hill climbing/simulated annealing, we only look at neighbors of a solution; thus, we can get stuck in the neighborhood of that solution, which can lead to a worse residue if the random solution is not good. Between hill climbing and simulated annealing, the results are similar, but for simulated annealing, the results could be worse if, when it happens to jump to a new random solution, it ends up in a worse neighborhood.

On the other hand, for the prepartitioned heuristics, we have a significantly better performance. This also makes sense, because when we pre-partition our elements, every time we generate a new solution or generate a neighbor, we reallocate a whole group of elements rather than just one at a time. Thus, we have much more flexibility with shifting our solution and thus are more likely to find a more accurate one. We see that for prepartitioning, simulated annealing actually performed the best. This could be the case if simulated annealing happens to jump to a really good solution; then, the random neighbor aspect will allow it to both be in a good neighborhood and optimize its solution in the neighborhood, which prepartitioning will allow more freely because of its flexibility.

4.3.2 Runtime

Runtimes for Heuristics in Seconds, Averaged over 50 Random Instances

Algorithm	No Pre-Partitioning	Pre-Partitioning
KK	0.000043	N/A
RR	1.461	3.916
HC	0.608	2.692
SA	1.7022	8.129

Once again, these time observations make sense. Karmarkar is significantly faster than all other heuristics, which makes sense because it runs in $O(n \log n)$ time, with our MaxHeap implementation. On the other hand, Repeated Random generates a new solution each iteration, which takes $O(n)$ time, and since we perform 25000 iterations, the time is significantly greater than the $n - 1$ iterations of Karmarkar-Karp. It also makes sense that Hill Climbing is faster than Repeated Random, since at each iteration it generates a neighbor rather than a completely new solution, which is only $O(1)$ time. However, since it also takes 25000 iterations, it will still be slower than Karmarkar-Karp. For Simulated Annealing, the runtime should be slower than Hill Climbing because it sometimes generates new solutions as well. The reason why it's also slower than Repeated Random may be because of the time taken to calculate the cooling function and check whether or not to go with a worse neighbor, which is a more complex calculation and can add up in time over 25,000 iterations.

5 Karmarkar-Karp as Starting Point

Finally, we can consider using Karmarkar-Karp as our starting point in these heuristics, and the effect that would have on both the performance and runtime of the heuristics.

For repeated random, using KK as a starting point would not necessarily have an effect on the performance, since running the heuristic would still just generate a new random solution and compare it against the KK solution.

For hill climbing, however, using KK as a starting point would be very beneficial to its performance. Since KK is already closer in accuracy, starting with the KK partition will help the solution keep moving towards the optimal from KK, so it will significantly help.

For simulated annealing, using KK as a starting point could also be beneficial, but not as much as for hill climbing. This is because simulated annealing does not always go with the best neighbor, so it doesn't always go to the local optimal from the KK solution. However, because it still searches for a local optimal most of the time, starting with KK can be helpful.

For runtime, starting with KK would actually increase the runtime of the algorithm, since the KK algorithm would run for $O(n \log n)$ time, and then we would simply still perform the heuristic implementation as normal, and their time complexity would not be changed. Thus, with the addition of the KK algorithm, our runtime would only increase.