CS 124 Programming Assignment 3: Spring 2024

Noah Chung[1] and Anaïs Killian[1]

[1]Harvard University, Class of 2026.

1 Abstract

In this assignment, our goal was to implement a number of heuristics for solving the Number Partition problem, which is NP-complete. As input, the number partition problem takes a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers. We output 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. Another way to view the problem is the goal is to split the set (or multi-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.

1.1 DP Solution

1.1.1 Algorithm:

For the purpose of our C++ code only, we assume A is a list of integers from index 0 to $n - 1$. Our algorithm is as follows:

```cpp
using namespace std;

int findMin(int A[], int n)
{
    // calculate b
    int b = 0;
    for (int i = 0; i < n; i++)
        b += A[i];

    bool dp[n + 1][b + 1];
    // fill in base cases
    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    // not possible when we have zero elements in one of the subsets
    for (int i = 1; i <= b; i++)
        dp[0][i] = false;

    // bottom-up dp
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= b; j++) {
            dp[i][j] = dp[i - 1][j];

            if (A[i - 1] <= j)
                dp[i][j] |= dp[i - 1][j - A[i - 1]];
        }
    }

    int residue = INT_MAX;

    // j loops from b/2 to 0
    for (int j = b / 2; j >= 0; j--) {
        if (dp[n][j] == true) {
            residue = b - 2 * j;
            break;
        }
    }
    return residue;
}
```

What this algorithm does is it creates a 2D array dp$[n+1][b+1]$. This is where b is still the total sum of A and n represents the subset of A from 1 to n. The piecewise nature is further described in the correctness section.

1.1.2 Correctness:

This solution is correct because we use the following piecewise function:

$$dp[i][j] = \begin{cases} 1 & \text{if some subset of A from 1 to i has a sum equal to j} \\ 0 & \text{otherwise} \end{cases}$$

Where 1 represents True and 0 represents False.

Here, note that:

- i ranges from $\{1, ..., n\}$
- j ranges from $0, ..., b$ where b is the total sum of integers in A.

Then, in order to achieve the residue that is minimized, we need

$$\min\{b - 2j \qquad \text{s.t. dp}[n][j] = 1\}, \quad 0 \leq j \leq \left\lfloor \frac{b}{2} \right\rfloor$$

This will result in the minimized residue since we only go from 0 to the floor of $\frac{b}{2}$. We want the minimized sum of A minus $2 \cdot j$ since this accounts for the subset of A from 1 to n.

Therefore, this indeed minimizes the residue and thus our code is correct.

1.1.3 Space Complexity:

Note that the matrix that we are building is dimensions b by n where b is the sum of A. This is because we build our matrix in a bottom-up manner with these dimensions in order to then find what is minimized when we took at some subset of A from 1 to n according to the equation above.

Therefore, we have a total space complexity of $O(n \cdot b) = O(nb)$ because we are using memoization in our table.

1.1.4 Runtime:

We now prove that the DP algorithm is polynomial in n and b.

Note that this solution is actually pseudo-poly time. This is because if we suppose the sequence of terms in A sum up to some number b, then each of the numbers in A has at most $\log(b)$ bits, so a polynomial time algorithm would take time polynomial in $n \log(b)$. Instead, we have found a dynamic programming algorithm that takes time polynomial in $O(nb)$.

As showed by our space complexity, we have found that our matrix is n by b, and after we fill in this matrix in a bottom-up manner, we then have to find what is minimized in the row where we take a subset of A from 1 to n for $0 \leq j \leq \lfloor \frac{b}{2} \rfloor$. Thus we have

$$O(n \cdot b) + O(n) = O(n \cdot b)$$

as our time complexity.

1.2 Karmarkar-Karp Algorithm

The Karmarkar-Karp differencing idea is to take two elements from A, call them a_i and a_j, and replace the larger by $|a_i - a_j|$ while replacing the smaller 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.

The Karmarkar-Karp algorithm can be efficiently implemented in $O(n \log n)$ steps using a max-heap. Recall in a heap that the insertion and extraction of elements both have a time complexity of $O(\log n)$.

The implementation steps are as follows:

1. We start by inserting all elements of the set A into the max-heap. Building the heap from an array of n elements requires $O(n \log n)$ time.
2. Then we run the differencing process:
   (a) While the size of the heap is greater than 1, we extract the two largest elements a_i and a_j from the heap, which takes $O(\log n)$ time for each extraction, leading to $O(\log n)$ for both.
   (b) We compute the difference $|a_i - a_j|$ and insert it back into the heap if it is non-zero, which takes $O(\log n)$ time.
   (c) We repeat this process until only zero or one element remains in the heap. Since each step decreases the heap size by one, there will be $O(n)$ iterations, resulting in a total of $O(n \log n)$ for the differencing process.
   (d) If there is zero items left in the heap, we return 0 as our residue.
3. The last element in the heap represents the residue. Extracting this element takes constant time, $O(1)$, as it is the only element left.

Thus the overall time complexity of the Karmarkar-Karp algorithm is $O(n \log n)$.

Runtime:

As described throughout the text above, we do two pop operations per iteration, and as described, we do $O(n)$ iterations. Since inserting and popping takes $O(\log n)$ time, then we overall have $O(3n \log n) = O(n \log n)$.

2 Experimental Results: 50 Random Instances

We generated 50 random instances of the Number Partition problem with different heuristics and the KK algorithm. For each instance, wwe run a repeated random, a hill climbing, and a stimulated annealing algorithm, using the two representations described below, each for at least 25,000 iterations.

The two representations are as follows: using prepartitioning, and not using prepartitioning.

Our experimental results for all of the algorithms as as follows:

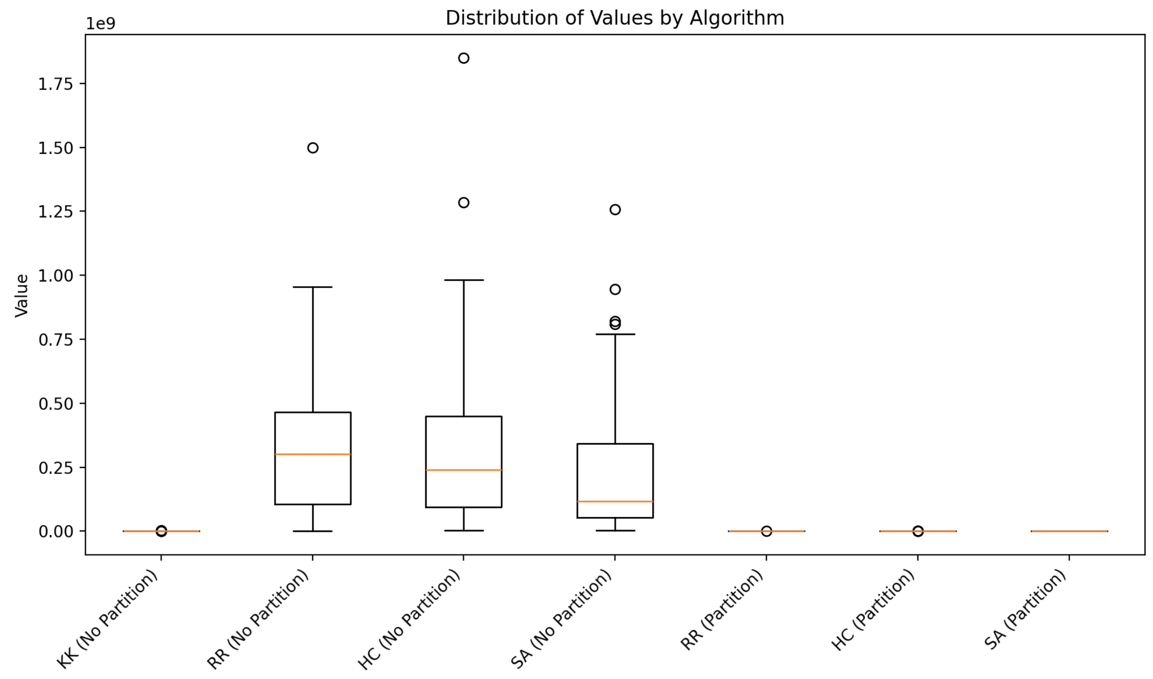


Table 1: Minimum Values for Each Algorithm Type

Algorithm	**Minimum Value**
KK	1084
RR (No Partition)	108471
HC (No Partition)	4022673
SA (No Partition)	4291112
RR (Partition)	2
HC (Partition)	1
SA (Partition)	2

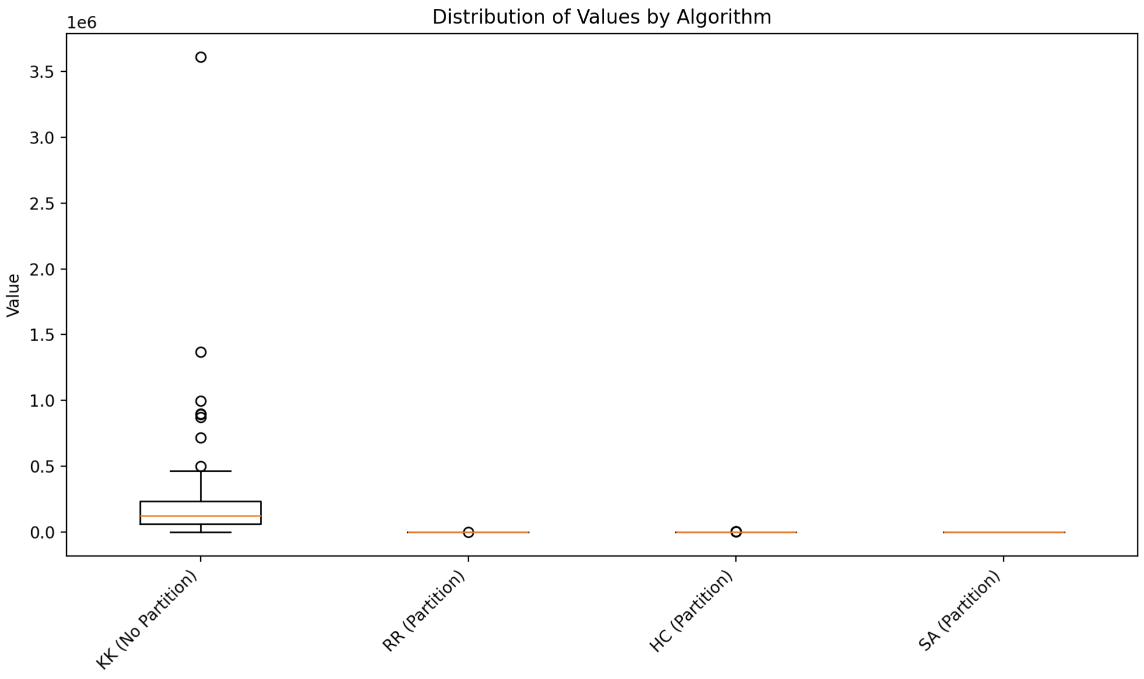
Using Non-Partitioning

Note that in the graph above, we can clearly see the results for the non-partition algorithms.

We find that the non partitioned algorithms all give values that are on the order of 1e9, which is orders of magnitude greater than all of the other algorithms.

With regards to RR, HC, and SA, note that RR and HC are pretty similar. HC seems to have greater outliers, which makes sense because that suggests that in some cases, the algorithm might have got stuck at local maximum. RR gives significantly lower minimum values, but the randomized nature of the algorithm means that the average residue values seem to be around the same as the rest. SA seems to have similar performance to HC in terms of minimum value because in most cases, HC getting lucky is the same as SA getting lucky. However, SA has a slightly lower mean value, which makes sense because the probability of not greedily moving to a certain state increases the chances that we don't get stuck in a local minimum.

Indeed, the repeated random algorithm looks at 25000 different assignments, and thus we have a chance to obtain a good state, however with the HC and SA algorithms, if they start out with a bad assignment it will be difficult to obtain a smaller residue. More specifically with HC, if we start in a state that is non-optimal, it will be difficult to get an optimal state. Note that HC has the greatest range in the partitioned algorithms graph. With non-partitioned SA, with have a bit more room to move around to different states than HC probabilistically.

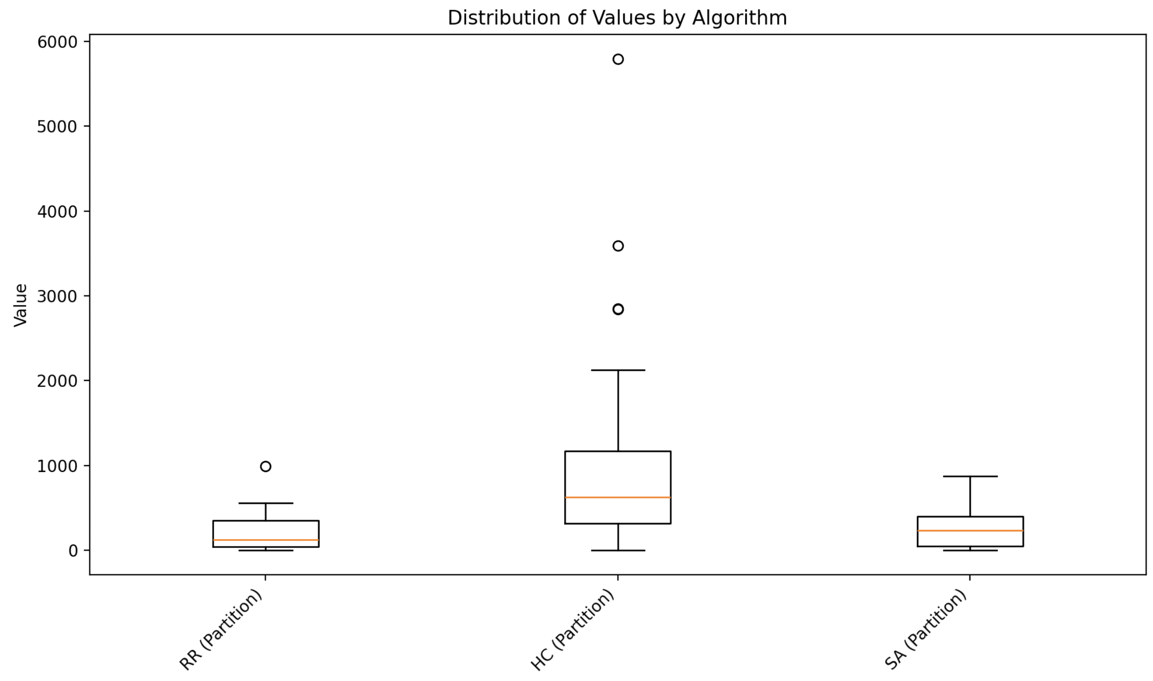


Karmarkar-Karp

Karmarkar-Karp seems to give a solution that is an order of magnitude better than the non partitioned solutions but orders of magnitude worse than any of the partitioned ones. Looking at the minimum values, it seems to give a relatively low minimum, 1084. However, it seems to have a high amount of upper level outliers. This is most likely because KK is essentially a greedy algorithm, taking the two largest numbers and differencing them. If it happens to work out that that was the correct solution, then our residue will be low. If we get unlucky, however, it could become very wrong. It is still generally better than the non partitioned solutions because we are following a defined heuristic that is intuitive, rather than a random selection. It also makes sense that this is worse than the pre partitioned solutions, as those run KK multiple times per run, which increases the chance that the residue becomes lower and lower.

Using Pre-Partitioning

Our zoomed-in results for partitioning are as follows:



In comparison to not using partitioning, these results make sense: with prepartitioning, we use KK, which helps to determine the residue, whereas non-partioning does not use KK. With prepartitioning, we descrease the number of inputs we can consider, as we essentially see “more elements” at once by combining two elements in one set.

As noted, the partitioned algorithms all give values on the order of 1e4, significantly lower than all of the other methods. It seems that RR (Partition) and SA (Partition) give lower values consistently, and HC with partition gives a slightly wider boxplot with a higher median value, which suggests that HC with partition is slightly less consistent than RR (Partition), which makes sense because there is a chance that you keep making the wrong decision.

Among those three algorithms using prepartitioning, it seems that repeated random has slightly better performance in terms of getting lower values. The Karmarkar-Karp algorithm has a separate distribution that gives values around 1e6. However, it notably has a significant number of outliers, suggesting that the algorithm is not very consistent.

The simulated annealing algorithm has better performance than HC, and this makes sense because the T function forces the algorithm to not make the greedy decision with some probability, which most likely prevents the algorithm from getting stuck in a local minima, and over 25000 iterations, it yields a much better result than HC. That is with RR, we have more freedom in moving around and finding lower residues because we can change our neighbors.

On the other hand with HC, we are likely to get stuck in local minima as a result of staying in a constrained space of neighbors and not going to worse neighbors.

2.1 Running Times

The average timings over the 50 trials are also below. Karmarkar-Karp essentially takes 0 time. The largest difference is caused by partitioning, as partitioning seems to add a little over a second to the average runtime. This makes sense because it is essentially extra steps. That is, this makes sense since using prepartitioning requires more steps including running the KK heuristic algorithm on the result A'.

Within each partition or no partition group, repeated random seems to take the longest, which makes sense because you regenerate the entire solution every time. And then HC is slightly faster than SA, which also makes sense because there is an extra probability check that requires some runtime.

Algorithm (Partition)	Average Time (s)
KK (no partition)	0.00000
RR (no partition)	0.15628
HC (no partition)	0.04006
SA (no partition)	0.04138
RR (partition)	1.52778
HC (partition)	1.49966
SA (partition)	1.50514

Table 2: Average running times for various algorithms with and without partitioning.

3 KK as Starting Point

As the Karmarkar-Karp algorithm seems to give values that are an order of magnitude lower than the non partitioned random algorithms, it seems that using the solution from the Karmarkar-Karp algorithm as a starting point for those algorithms would improve the performance of the non partitioned algorithms. This is because the randomized algorithms always compare to the best solution they have seen so far. Since Karmarkar-Karp would give the randomized algorithm a starting point that is order of magnitudes better than if the algorithm had run by itself, running Karmarkar-Karp first would consistently give a better solution. Thus, using the Karmarkar-Karp solution should provide better performance. Runtime shouldn't be affected too much because as we can see that Karmarkar-Karp essentially runs instantly. Since Karmarkar-Karp seems to give worse solutions than what we end up in when partitioning, it wouldn't make much sense to use those. In addition, since Karmarkar-Karp doesn't give a solution that is a partition, it also doesn't really make sense to use it for the random algorithms that are partitioned.

3.1 Acknowledgements

We would like to thank Ed for the opportunity to complete this Progset. We would also like to thank the CS 124 TFs for accommodating us during their lovely office hours. Special thanks to the professors!