CS 124: Number Partition

Albert Qi and Steve Dalla

April 20, 2023

Contents

1. Introduction

In this write-up, we will compare the residues for the Number Partition problem outputted by the Karmarkar-Karp algorithm as well as a repeated random, a hill climbing, and a simulated annealing algorithm both with and without prepartitions. First, we will illustrate a dynamic programming algorithm that runs in pseudo-polynomial time. Then, we will explain how the Karmarkar-Karp algorithm can be implemented in $O(n \log n)$ steps and then compare the algorithm to a variety of randomized heuristic algorithms on random input sets. Finally, we will discuss our experiments in more depth, covering the intricacies of our algorithms and other intriguing details we discovered along the way.

2. Dynamic Programming Solution

Although the Number Partition problem is NP-complete, it can still be solved in pseudo-polynomial time. We will illustrate a dynamic programming algorithm that does so:

1. Let $dp[i][j]$ be TRUE if and only if there exists a subset of the first i elements of A whose sum equals j. Then, we have the following recurrence relation:

$$dp[i][j] = \begin{cases} \text{TRUE} & \text{if } i = 0 \land j = 0 \\ \text{FALSE} & \text{if } i = 0 \land j \neq 0 \\ dp[i-1][j] & \text{if } j < A[i] \\ dp[i-1][j] \lor dp[i-1][j - A[i]] & \text{otherwise} \end{cases}$$

2. To actually find a partition, however, we need to keep track of some extra information. Let $s[i][j]$ be some subset of the first i elements of A whose sum equals j, if one exists. Then, when we update $dp[i][j]$, we also need to update $s[i][j]$ appropriately:

$$s[i][j] = \begin{cases} \{\} & \text{if } i = 0 \\ s[i-1][j] & \text{if } dp[i-1][j] = \text{TRUE} \\ s[i-1][j - A[i]] \cup \{A[i]\} & \text{if } dp[i-1][j-A[i]] = \text{TRUE} \\ \{\} & \text{otherwise} \end{cases}$$

3. Loop from $i = 0$ to $i = n$ and on each iteration, loop from $j = 0$ to $j = \sum a_k$. On each of these inner iterations, calculate and update $dp[i][j]$ and $s[i][j]$ according to the recurrence relations.
4. Let $i = \lfloor (\sum a_k)/2 \rfloor$. Then, while $dp[n][i] = \text{FALSE}$, set $i = i - 1$.
5. Return $(s[n][i], A \setminus s[n][i])$.

Now, we will analyze the runtime of our algorithm:

1. Suppose that the sequence of terms in A sum up to some number b. Then, creating an array dp of size $O(n) \times O(b)$ takes $O(nb)$ time.
2. Because we are using memoization via dp, there are only $O(nb)$ subproblems that we need to solve. In each subproblem, we only perform a constant amount of work, meaning filling dp takes $O(nb)$ time.
3. Creating an array s of size $O(n) \times O(b)$ takes $O(nb)$ time.
4. Again, there are only $O(nb)$ subproblems that we need to solve. In each subproblem, updating s only takes a constant amount of time, meaning filling s takes $O(nb)$ time.
5. Each a_k has $O(\log b)$ bits, and we have $O(n)$ of these numbers in total. Thus, adding all of the a_k's takes $O(n \log b)$ time.
6. Dividing an $O(\log b)$-bit number by 2 takes $O(\log b)$ time.
7. Finding the floor takes $O(1)$ time.
8. Our while loop runs for $O(b)$ iterations, and we perform a constant amount of work on each iteration. Thus, our while loop takes $O(b)$ time overall.
9. Finding the set difference takes $O(n)$ time.

Our overall runtime for the algorithm is just the sum of the runtimes for these steps. As a result, the runtime for the algorithm is $O(nb) + O(nb) + O(nb) + O(nb) + O(n \log b) + O(\log b) + O(1) + O(b) + O(n) = O(nb)$ by asymptotic theory.

Furthermore, we know that our algorithm is correct because of the following:

1. Our dp is correct.
    - The first zero elements of A always sum to zero, and we correctly set $d[0][0] = \text{TRUE}$ and $d[0][j] = \text{FALSE}$ for $j \neq 0$.
    - If $j < A[i]$, then we cannot possibly get a sum of j by including $A[i]$. As a result, we must ignore $A[i]$ and try to find a sum of j by using the first $i - 1$ elements. Thus, we correctly set $dp[i][j] = dp[i-1][j]$ if $j < A[i]$.
    - If $j \geq A[i]$, then we have two possible cases in which the first i elements can sum to j. First, they can sum to j if the first $i-1$ elements sum to j, whereupon we will ignore $A[i]$. Second, they can sum to j if we include $A[i]$ in our sum and obtain a sum of $j - A[i]$ in the first $i - 1$ elements. Thus, we correctly set $dp[i][j] = dp[i-1][j] \vee dp[i-1][j - A[i]]$ if $j \geq A[i]$.
2. Our s is correct.
    - If $i = 0$, then we have no elements in our sum, meaning $s[0][j] = \{\}$ regardless of whether $dp[0][j]$ is TRUE or FALSE.
    - If $dp[i-1][j] = \text{TRUE}$, then there exists some subset of the first $i - 1$ elements that sum to j. We can reuse this set to have the first i elements also sum to j. Thus, we correctly set $s[i][j] = s[i-1][j]$ if $dp[i-1][j] = \text{TRUE}$.
    - If $dp[i-1][j - A[i]] = \text{TRUE}$, then we can find a subset that sums to j by taking a subset of the first $i - 1$ elements and adding the element $A[i]$. Thus, we correctly set $s[i][j] = s[i-1][j - A[i]] \cup \{A[i]\}$ if $dp[i-1][j - A[i]] = \text{TRUE}$.
    - Otherwise, there is no subset that sums to j, meaning we should set $s[i][j] = \{\}$.
3. We correctly initialize $i = \lfloor (\sum a_k)/2 \rfloor$. This represents the best case scenario in which the sums of the partitions are as close to each other as possible. Because we then slowly decrement i, we are guaranteed to find the partition that minimizes the residue.

As such, we can see that our algorithm is indeed correct. Thus, we have proven that there exists a dynamic programming algorithm to solve the Number Partition problem in pseudo-polynomial time.

3. Karmarkar-Karp Algorithm

Assuming that the values in A are small enough such that arithmetic operations take one step, the Karmarkar-Karp algorithm can be implemented in $O(n \log n)$ steps.

The Karmarkar-Karp algorithm works by continually choosing the two largest elements in the sequence, taking their absolute difference, and adding the difference back into the sequence. By utilizing the max-heap data structure, we can ensure that each iteration only takes $O(\log n)$ time. Below are the runtimes for the heap operations:

- Make: $O(n)$
- Pop: $O(\log n)$
- Insert: $O(\log n)$

Each iteration requires two pops and one insert operation, meaning the runtime of each iteration is $O(2 \log n) + O(\log n) = O(\log n)$ by asymptotic theory. The size of our sequence shrinks by 1 after each iteration, so we have a total of $O(n)$ iterations. Furthermore, we only make the heap once at the very beginning, meaning the Karmarkar-Karp algorithm should take $O(n) + O(n \log n) = O(n \log n)$ steps overall.

4. Results

We perform 50 trials on sets of 100 integers, with each integer being a random number chosen uniformly from the range $[1, 10^{12}]$. For each trial, we calculate the residue with each of the algorithms. Note that we run 25000 iterations for each of the randomized heuristic algorithms.

The following table shows the residues for each algorithm averaged across 50 trials.

Algorithm	Average Residue
Karmarkar-Karp	170646.44
Repeated Random	310812988.92
Hill Climbing	370117223.52
Simulated Annealing	299258510.08
Prepartitioned Repeated Random	181.20
Prepartitioned Hill Climbing	643.36
Prepartitioned Simulated Annealing	236.48

From this, we can see three major groups emerge: the Karmarkar-Karp algorithm, randomized heuristic algorithms without prepartitions, and randomized heuristic algorithms with prepartitions.

It seems that the prepartitioned algorithms perform significantly better (i.e., $\approx 10^6$ times better) than their non-prepartitioned counterparts. This is likely due to the fact that the prepartitioned algorithms utilize the Karmarkar-Karp algorithm, which will ultimately lead to better residues than those from solutions consisting of random signs. Thus, the prepartitioned algorithms result in much lower residues.

Furthermore, we can see that simulated annealing seems to perform the best out of the non-prepartitioned randomized heuristic algorithms. This makes sense; it outperforms the repeated random algorithm by consistently improving a solution, and it outperforms the hill climbing algorithm by being able to escape local minima that are not globally optimal.

Interestingly, the hill climbing algorithm seems to lag behind both the repeated random and simulated annealing algorithms, regardless of whether or not we decide to prepartition. One possible explanation is that the algorithm is highly dependent on its randomly chosen starting point. If we choose a poor starting solution by chance, then we may be stuck with a residue that is locally optimal but still very far from the global optimum.

5. Additional Observations

In our experiments, we always utilize a random initial starting point. However, we could have used the solution from the Karmarkar-Karp algorithm as our starting point for the randomized algorithms.

To do so, we would first need to slightly modify the Karmarkar-Karp algorithm to return not just the residue but also the partition of numbers. One way of doing this would be to build some sort of tree and keep track of the relationships between different elements.

Now, suppose that we start the repeated random algorithm with the solution from the Karmarkar-Karp algorithm. That is, instead of initially generating a random solution, we utilize the one obtained from the Karmarkar-Karp algorithm. Believe it or not, there is not much that would change since the candidates that we generate do not rely on our current solution. The only guarantee that we really have is that the repeated random algorithm would return a solution at least as good as the one from the Karmarkar-Karp algorithm.

However, if we start with the solution from the Karmarkar-Karp algorithm for the hill climbing or simulated annealing algorithms, then we would notice more of a difference. This is because these algorithms do not randomly generate solutions but rather look at their neighbors. By starting at the solution from the Karmarkar-Karp algorithm, we would be able to build off of this solution and improve it greatly. Thus, we would notice more of a difference.

6. Discussion of Experiments

Notice that our implementation of `simulated_annealing` allows us to define `hill_climbing` in terms of `simulated_annealing`. Due to the similarities between the hill climbing and simulated annealing algorithms, we are able to make use of some abstraction. This helps us significantly simplify the code.

Furthermore, as a minor optimization, we pass objects by reference whenever possible. This reduces the number of unnecessary copies and slightly improves the runtimes of almost all of our algorithms.

Lastly, note that we generate random values through the `<random>` header because we trust this more than the C standard library function `rand`. Our generator is also seeded via `random_device`, ensuring that each trial has independent randomness.