CS 124 Programming Assignment 3: Spring 2022

Your name(s) (up to two): Shirley Zhu

Collaborators: (You shouldn't have any collaborators but the up-to-two of you, but tell us if you did.)

No. of late days used on previous psets: 8
No. of late days used after including this pset: 10

Give a dynamic programming solution to the Number Partition problem.

Answer: Suppose we have the sequence A = $(a_1, a_2, a_3, ..., a_n)$ of non-negative integers. Let $dp[i, j]$ be an array where the value is true when there exists a partition of the numbers $a_1, a_2, ..., a_i$ that have a residue j. The recursive definition of dp using subproblems is

$$dp[i, j] =$$
$$\text{True, if } i = 1 \text{ and } a_i = j$$
$$\text{True, if } i \neq 1 \text{ and } dp[i-1, |j - a_i|] \text{ is True or } dp[i-1, j + a_i \text{ is True}$$
$$\text{False, otherwise}$$

Algorithm:
Start populating at $dp[1, 0]$. Increase the i term and continue populating the dp array until you get to $i = n$. In other words, fill out dp in the order $dp[1, 0], dp[2, 0], dp[3, 0], ...dp[n, 0]$. Then increment the value of j and fill out each row in similar increasing i.

Stop when you find $dp[n, j] = T$ for some $j < b$ or when you reach $j = b$.

Proof of Correctness:
To prove that our dynamic programming definition for dp is accurate, we will use a proof by exhaustion. We will show that

$$dp[i, j] =$$
$$\text{True, if } i = 1 \text{ and } a_i = j$$
$$\text{True, if } i \neq 1 \text{ and } dp[i-1, |j - a_i|] \text{ is True or } dp[i-1, j + a_i \text{ is True}$$
$$\text{False, otherwise}$$

given that we have already filled out the array $dp[x, y]$ for $x < i$ when $y = j$ and for all $x \leq n$ when $y < j$.

First, we consider when $i = 1$. We are looking only at one element. With only one element, the only option for partitioning is to create one group containing a_i and the other group contains no elements. Then, the residue is $a_i - 0 = a_i$. If $a_i = j$, the residue is j and $dp[i, j]$ should be set to True. When $a_i \neq j$, we are not able to achieve a residue of j and $dp[i, j]$ should be set to False.

Second, we consider when $i \neq 1$. We will show there exists some way for us to partition $a_1, a_2, \ldots, a_i$ so that the residue is j, only if there is a way to partition $a_1, a_2, \ldots, a_{i-1}$ into groups with a residue of $j + a_i$ or $|j - a_i|$. Given that there exists a partition of $a_1, a_2, \ldots, a_{i-1}$ with residue j, there exists a partition into sets S_1 and S_2 where the sum of the elements in S_1 is some integer X and the sum of the elements in S_2

is $X + j$. There are two options for which set the element a_i is in.

Case 1: a_i is in S_1
Then, we can consider the partition of $a_1, a_2, \ldots, a_{i-1}$ where the first set is $S_1 \backslash a_i$ and the second set is S_2. The sum of elements of $S_1 \backslash a_i$ is $X - a_i$. Thus, the residue of the partition of $a_1, a_2, \ldots, a_{i-1}$ is $(X + j) - (X - a_i) = j + a_i$.

Case 2: a_i is in S_2
We can take out a_i from S_2 and have a partition for $a_1, a_2, \ldots, a_{i-1}$. The sum of elements in $S_2 \backslash a_i$ is $X + j - a_i$ and the sum of elements in S_i will still be X. The residue is the absolute value of the difference of $X + j - a_i$ and X, which is $|X + j - a_i - X| = |j - a_i|$. Thus, the residue of the partition of $a_1, a_2, \ldots, a_{i-1}$ is $|j - a_i|$.

We have shown that when there exists a partitioning of $a_1, a_2, \ldots, a_i$ that gives a residue j, then there must also exist a corresponding partitioning of $a_1, a_2, \ldots, a_{i-1}$ that has a residue of either $j + a_i$ or $|j - a_i|$. So, we should set $dp[i, j]$ to be True when $dp[i - 1, |j - a_i|]$ is True or $dp[i - 1, j + a_i]$ is True and $i \neq 1$. Otherwise, set $dp[i, j]$ to be False.

Since we have exhaustively proven the cases for when $i = 1$ and $i \neq 1$, we have proven that our dynamic programming definition works. The goal is to find a partition of the sequence A with the lowest possible residue. We want to partition the entire sequence, so we will be looking at $dp[i, j]$ where $i = n$. Since j represents the residue, we want to find the smallest value of j where $dp[n, j]$ is True.

Runtime:
There are n rows in the dp array because there are n total elements in A. The maximum sum of the elements is given to be b, so we have b columns. The total size of our array is nb. To populate the dp array, we have a runtime of $O(nb)$, because each calculation of $dp[i, j]$ is $O(1)$ time. In our recursive definition of dp, we check two previous elements in dp to determine $dp[i, j]$. Since we are populating the array in a bottom up fashion, the previous dp elements that we are examining will already be filled and the act of checking an element in an array is $O(1)$ time.

Space Complexity:
We have shown that the matrix would be size n by b so the space complexity is $O(nb)$.

Explain briefly how the Karmarkar-Karp algorithm can be implemented in $O(n \log n)$ steps, assuming the values in A are small enough that arithmetic operations take one step.
At each step of the Karmarkar-Karp algorithm, we take out the two largest numbers in the sequence and insert the absolute value of the difference. Given that we start with a sequence of size n, there are $n - 1$ iterations needed to get to only one non-zero number remaining in the sequence, because at each iteration we decrease the number of non-zeroes values by 1. The runtime of first building a max heap is $O(n)$. For every following step, we do two deletion operations and one insertion operation. Each operation is $O(\log n)$ time so the total of the three operations is $O(\log n)$ as well. The total of the Karmarkar-Karp algorithm where $O(\log n)$ operations are run for $n - 1$ iterations is $O(n \log n)$.

Generate 50 random instances of the problem as described above. For each instance, find

the result from using the Karmarkar-Karp algorithm. Also, for each instance, run a repeated random, a hill climbing, and a simulated annealing algorithm, using both representations, each for at least 25,000 iterations. Give tables and/or graphs clearly demonstrating the results. Compare the results and discuss.

ALGORITHM	Average Residue
Karmarkar-Karp	291035.67
Random	288915573.16
Hill Climbing	305729183.38
Sim Annealing	239582341.6
Prepartitioned Random	166.04
Prepartitioned Hill Climbing	194.75
Prepartitioned Sim Annealing	203.88

Analysis of Results:
We see that the prepartitioned algorithms have smaller residues than the algorithms that did not use prepartitioning. Prepartitioning leads to Kamarkar Karp being performed on the sequences, which leads to better results than generating random positive and negative signs on an entire sequence. Thus, the prepartitioned algorithms would have better average residues. Among the algorithms that do not use prepartitioning, the simulated annealing algorithm performs the best, meaning the output is the smallest residue. This makes sense because simulated annealing is optimized with keeping track of the best solution seen so far and choosing a random neighbor of the best solution. Then we see that Hill Climbing does not perform better than the repeated random algorithm, so using neighbors instead of randomly generating new sequences to try does not make a large impact on decreasing the residue.

Comparing the runtimes, I find that the simulated annealing algorithm has the slowest runtime, which aligns with what we would expect since the simulated annealing requires using a cooling schedule and checking multiple versions of the random solution sequences. Karmarkar Karp was the algorithm with the fastest runtime. This makes sense because the other prepartitioned algorithms call on Karmarkar Karp multiple times. The non partitioned algorithms take less time than the partitioned algorithms because they do not have the extra step of taking into account partitioning.

Discuss briefly how you could use the solution from the Karmarkar-Karp algorithm as a starting point the randomized algorithms, and suggest what effect that might have. (No experiments are necessary.)

Imagine that we could modify KK to return the partitioning rather than the value of the residue.

Repeated Random:
In our current implementation of the repeated random algorithm, we generate a random solution and compare the resulting residue with the residue of new randomly generated solutions. Instead, we could use Karmarkar-Karp to determine a residue that we set as the upper bound. Then we compare the residues of new randomly generated solutions to the upperbound residue. We are then guaranteed to return a solution that is at least as optimal as the KK algorithm partition. The repeated random algorithm would ultimately not be improved significantly by starting with KK because a large number of trials would likely return a

partition as good as the KK algorithm partition.

Hill Climbing:
We can modify the hill climbing algorithm by beginning with the KK algorithm output instead of randomly generating a solution. A randomly generated solution is likely to have a higher residue than the output of our KK algorithm. Since the hill climbing algorithm makes improvements on the best possible solution seen so far, starting with an improved solution will lead to a better answer after running through the iterations in simulated annealing.

Simulated Annealing:
A randomly generated solution is likely to have a higher residue than the output of our KK algorithm. We can modify the simulated annealing algorithm by beginning with the KK algorithm output instead of randomly generating a solution. Since the simulated annealing algorithm makes improvements on the best possible solution seen so far, starting with an improved solution will lead to a better answer after running through the iterations in simulated annealing.