Programming Assignment 3
CS 124

Artidoro Pagnoni, Nisha Swarup

April 25, 2016

1 Dynamic Programming Solution to Number Partition Problem

1.1 General Idea

Although the Number Partition Problem is NP complete, we can solve it in pseudo-polynomial time. Let b be the sum of all n integer entries of A. The solution that we will propose is polynomial in nb. In the Number Partition Problem we are essentially creating two sets having the least possible difference.

Our solution lies on the computation of all possible partitions of the initial subset of elements into two sets. We only care of the possible values that can be reached by the sum of the elements in every partition. Since the sum of all entries of A is b, we are interested in knowing all the possible values in the range $[0, b]$ that can be reached by creating a subset of A and summing over all elements of the subset. We can imagine this information stored in an array of length b in which a 1 in position i means that i is the result of the sum of the elements of a subset of A. If i is not the sum of the elements of a subset, then the ith entry in the array will be 0. Given such an array we can find the value that is closest to $b/2$. The closer we get to $b/2$, clearly the lower the residue between the two subsets.

1.2 Recurrence

We can calculate this solution recursively. The recursion is done on the initial set. At every step we include one more element to the initial set, eventually covering the whole set. Given the solution array at one step, we can calculate the solution array at the next step in the following way. All the values that could be reached at the previous step can still be reached (we can choose the same subsets, and just discard the new element). In addition, we can choose subsets that include the new element. This will increase the value of the sum of their elements by the the new element.

We can formalize this by filling out a matrix M of size $n \times b$, such that the rows correspond to the solution arrays mentioned above. The first row is the base case, and corresponds to the empty set (when we don't include any element of A). As we increase the row number, we add more elements of A to the initial set. The columns correspond to the values of the sum that need to be reached. We therefore have:

Recurrence Relation:

$$
\begin{aligned}
M_{(0,0)} &= 1 && (1)\\
M_{(0,j)} &= 0 \qquad \text{for } j \neq 0 && (2)\\
M_{(i,j)} &= \max\{M_{(i-1,j)}, M_{(i-1,j-A_i)}\} && (3)
\end{aligned}
$$

The base case is clear, with no elements we can only reach value 0.

The recurrence equation is also quite straightforward. The first term is 1 when you could reach j at the previous step. The second term is 1 when we can reach j by adding A_i (the new element added at this last step) to any of of the values at the previous step. The max ensures that if any of the two terms is 1, we will get 1 out.

We fill out the matrix row by row from left to right, following the recurrence equation.

Once we have the matrix, we will only consider the last row, and find the closest reachable value to $b/2$. We can therefore run through the last row starting at $\lfloor b/2 \rfloor$ and finding the first non zero entry. One subset will have the value corresponding to the index of the column of the first non zero entry, and the other will have the complement of that index $(b - j)$.

The residue can be calculated by taking the absolute value of the difference of the values found for the two subsets.

1.3 Reconstruction of the Subsets

This algorithm finds the minimum residue but does not tell us how to partition the set A. If we want to construct the two subsets, we need to fill out an addition matrix, of the same size. The matrix will be composed of pointers to the parent entry. By parent we mean the entry that was equal to 1 in the equation $M_{(i,j)} = \max\{M_{(i-1,j)}, M_{(i-1,j-A_i)}\}$. If both terms were 1, pick the first.

To reconstruct the subset we need to find for every element if they belong to one set or the other. We will start with the last element in A. We take the last row in the Pointer's matrix, and the column that corresponds to one of the subsets,(the closest possible to $b/2$). If the pointer is pointing to the same column then the element in consideration is in set 1, if not it is in set 2. The next element to be considered is the one pointed by the pointer. And we repeat the same reasoning until we get to the first element.

1.4 Further Optimization

We notice that we don't really need to complete the right half of either matrices. It is a question of symmetry, if one set has a total sum below $b/2$ the other will be above. Therefore we need to make sure to fill out the matrix up to column $\lceil b/2 \rceil$. We will start at $b/2$ and go down in the last row when looking for the closest reachable entry to $b/2$. This saves half of the time and space, but will not change the asymptotic running time.

1.5 Space and Time Complexity

In terms of space complexity, this algorithm is $O((nb)^2)$, since we need to fill out two matrices of size $n \times b/2$.

The time complexity is polynomial in nb more precisely it is $O((nb)^2)$ since we need to fill out the entire matrix and every entry requires a constant number of operations (two comparisons and writing to both the solution and the pointer matrix). This confirms that we have a pseudo-polynomial solution for the Number Partition Problem.

2 Karma Karp Algorithm Time Complexity

The Karma Karp algorithm can be implemented in $O(n \log n)$. We will describe the algorithm in that we have implemented.

We construct a binary Max Heap with the initial set. Inserting something in the Max Heap takes at most $O(\log n)$, and we are doing it for every element in the set. Therefore the total time from the construction of the Max Heap is $O(n \log n)$.

Once we have the Max Heap constructed, we need to extract the two largest elements. This corresponds to two Delete Min operations, which take $O(\log n)$ each. We then take the difference of the two elements and insert it back to the Max Heap, also with time $O(\log n)$.

We repeat the last step exactly $n-1$ times. At every step we reduce the size of the Max Heap by 1 (two delete min and one insert). We have an initial size of n and we need to get down to 1. Which takes $n-1$ steps.

Now putting everything together we have a run time of:

$$
\begin{aligned}
& \text{Construct Max Heap} + (n-1)(2 \text{ Delete Min} + \text{Insert}) && (4)\\
= \; & O(n \log n) + (n-1)(2 * O(\log n) + O(\log n))) && (5)\\
= \; & O(n \log n) + O(n \log n) && (6)\\
= \; & \boxed{O(n \log n)} && (7)
\end{aligned}
$$

Which shows that our algorithm implements Karma Karp in $O(n \log n)$.

3 Karma Karp as Starting Point for the Randomized Algorithms

The implementation of Karma Karp that we have provided returns a residue but does not construct the corresponding partition. We could easily construct it with a slight variation, as suggested in the prompt.

Supposing that we get the partition from the Karma Karp algorithm, we can use that partition as a starting point instead of generating a random one.

The first thing to notice is that the three algorithms always return the best solution found. Using the Karma Karp partition as a starting point ensures that we will never get a worst solution. Starting with Karma Karp will have different effects for the two representations.

Standard vs Prepartition: In the Standard Representation starting with Karma Karp would start our optimization from that point in solution space. We therefore expect our solution to slightly improve

from the Karma Karp solution, but only slightly. It would only slightly improve because it would quickly get stuck in local minima, the solution space is not very well defined.

With the Prepartition representation, the optimization from the Karma Karp solution would still reduce the residue significantly. We expect the solution to be lower than the solutions obtained without starting with Karma Karp. Here it is harder to get stuck in local minima because the solution space is defined better.

Repeated Random: Starting Repeated Random with Karma Karp has the only effect of adding among to the randomly generated solution the Karma Karp solution. As we have seen in the previous graphs, Karma Karp offers usually a significantly better solution than the Repeated Random Algorithm for the standard representation. Therefore, on average, we will get the Karma Karp solution back when using the variation of Repeated Random with the standard representation.

With the prepartition representation instead, Repeated Random usually generates better results than Karma Karp. So adding the Karma Kart solution to the solution set won't affect the result of the algorithm.

Hill Climbing: When we start Hill Climbing with Karma Karp we are actually starting with an already fairly optimized solution. What will happen is that improvements will be less frequent. This means that the curve of residue in time will be flatter. As we get closer to the optimized value the solution space gets flatter and flatter.

Simulated Annealing: Similarly, starting with the Karma Karp solution corresponds to starting with an already fairly optimized solution. Here as well, the frequency of the improvements will be lower. This means that the curve of residue in time will be flatter.