Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

April 27, 2014

1 Programming Assignment 03

1.1 Introduction

For this programming assignment, you will implement a number of heuristics for solving the NUMBER PARTITION problem, which is (of course) NP-complete. As input, the number partition problem takes a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers. The output is a sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in \{-1, +1\}$ such that the *residue*, is as shown in equation (1), is minimized.

$$u = \left| \sum_{i=1}^{n} s_i \cdot a_i \right| \tag{1}$$

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 which roughly equal sums. The absolute value of the difference of the sums is the residue.

In this programming assignment, we are going to analyze and implement several heuristic methods to solve this problem.

1.2 Number Partition in $O(n \cdot b)$

As a warm-up exercise, you will first prove that even though Number Partition is NP-complete, it can be solved in pseudo-polynomial time. That is, 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 $n \cdot \log b$. Instead you should find a dynamic programming algorithm that takes time polynomial in $n \cdot b$.

If we consider sum over the elements in A is b, then the program can be reduced as from a set of integers, find the subset whose elements sum b. To solve this problem using dynamic program, we are going to consider smaller problems which are given by finding

the subset whose elements sum $b-1$. If we continue this way simplifying this problem, we are going to reach to the simple problem of finding a subset over n elements, whose sum is 1.
This problem can be simplified even more if we consider a smaller subset, i.e., finding a subset over $n-1$ elements, whose sum is 1. If we continue reducing this problem, then we are going to reach the simple problem which is determine if we the first element, is equal to 1.
If we can solve this problem, we can solve the problem of the residue by finding the closest element to $b/2$ that can be yielded by summing a subset of A. For this, we are going to create a tableau with n rows, which are going to represent the elements of the set and b columns, that are going to represent the numbers that can be yielded by summing up a subset from A. So we have to create a n $n \cdot b$ boolean tableau D, in which each entry (i, j) is true if it is possible to sum take the elements in the set A from a_1 to a_j, i.e., from the first one, up to the j-th element.
The recurrence is going to be given by considering how can $D(i, j)$ be computed if I already know if $D(i-1, j)$ and $D(i-1, j-1)$. If we know that $D(i-1, j)$ is true, that means that we can yield the number j wit the previous $i-1$ elements, and we can do it with the first i elements just by not considering the i-th element in the subset.

If $D(i-1, j)$ is false, then it means that so far we have not been able to generate a subset whose elements sum j. Hence, we have to consider the information that $D(i-1, j-1)$ can give, which only tells us if the number $j-1$ was able to be computed with $i-1$ elements. If we were to add the element a_i to the set, we would be able to yield the value j as long as the current sum is $j-a_i$. Therefore if $D(i-1, j-a_i)$ is true, then $D(i, j)$ is true because we just need to add a_i to the subset in $D(i-1, j-a_1)$.

Before applying the recurrence formula in a nested loop, we need to consider a base case or a way to initialize the tableau with a sum that can be yielded. This initialization is going to be given by the minimum element in the array. This algorithm is shown in Algorithm 1.

Algorithm 1: Algorithm to populate the tableau to check if a sum is possible

```
1 funcion populateTableau(Array A)
2 Long b = ∑_{a∈A} a
3 Long minimum = A.minimum
4 boolean[] tableau = new boolean[A.size][b + 1]
5 tableau[0][minimum] = true // initialize tableau
6 for (int i = 1; i < A.size; i++) do
7     for (long j = 1; j ≤ b; j++) do
8         if tableau[i - 1][j] or tableau[i - 1][j - A[i]] then
9             tableau[i][j] = true;
```

So far we can only decide if based on the set A, we are able to compute a sum k, where

k can be b or $b/2$. We want to partition A into A_1 and A_2, such that if $S_1 = \sum_{a \in A_1} a$ and $S_2 = \sum_{a' \in A_2} a'$, then $|S_1 - S_2|$ is minimized. Assuming the best case is when $S_1 = m$, then $S_2 = b - m$, then the residue is going to be given by equation (2).

$$\begin{aligned} |S_1 - S_2| &= |\sum_{a \in A_1} - \sum_{a' \in A_2} a'| \\ &= |m - (b - m)| \\ &= |m - b + m| \\ &= |2m - b| \\ &= |2m - 2\frac{b}{2}| \\ &= \left|2\left(m - \frac{b}{2}\right)\right| \\ &= 2\left|m - \frac{b}{2}\right| \end{aligned} \tag{2}$$

The number partition problem now can be reduced to finding the element m that is possible to yield and is the closest to $b/2$, since the perfect scenario is when S_1 and S_2 are both equal to $b/2$. So we only have to iterate over the columns from $b/2$ down to 1, until we find that $D(i, j)$ is true. This search might be in $O(n \cdot b)$ but can be reduced if maintain an boolean array C of size equals to b, in which an element c is going to be true as long as $D(x, c)$ is true for any $x \in [1, \ldots n]$. The algorithm 2 shows the updated version to add the new array to keep track of the values that be generated. Notice that so far the complexity of the algorithm is $O(n \cdot b)$, which might be considered as the initialization and population of the tableau.

Algorithm 2: Algorithm to populate the tableau to check if a sum is possible in $O(b)$

```
1  funcion populateTableau(Array A)
2  Long b = sum_{a in A} a
3  Long minimum = A.minimum
4  boolean[][] tableau = new boolean[A.size][b + 1]
5  boolean[] C = new boolean[b + 1]
6  tableau[0][minimum] = true // initialize tableau
7  for (int i = 1; i < A.size; i++) do
8      for (long j = 1; j ≤ b; j++) do
9          if tableau[i - 1][j] or tableau[i - 1][j - A[i]] then
10             tableau[i][j] = true;
11             C[j] = true
```

Finally, we just need to find m such that $|m - b/2$ is minimized. This can be achieved in $O(b)$ by considering just the first value that can be computed starting from $b/2$ down to

1. The algorithm 3 shows how this search can be done and returns the absolute difference $|m - b/2|$.

Algorithm 3: Algorithm check if a sum is possible in $O(b)$

```
1 funcion findSumThatMinimizes(Long b)
2 Long b = ∑_{a∈A} a
3 for (long m = b/2; m > 0; m = m − 1) do
4     if C[m] == true then
5         return |m − b/2|
6 return null
```

Based on equation (2), the returned value just needs to be multiplied by 2 and we would have gotten the minimum residue possible. The only thing that we have to consider is when b is odd, which makes $b/2$ be not an integer. For this scenario we just need to return $2|m - b/2| + b \bmod 2$, to add the missing 1, due to the division.

The algorithm works in theory and in practice is going to work fine with small integers, but when we consider integers in the range of 10^{12}, then creating the tableau and populating it is going to take too long. Also languages like Java do not allow the creating of arrays that big. To solve this problem, we can create a class called *SparseMatrix* that is going to manage big matrices.
For this approach we considered an array of maps to store the values that are true for each column, and also an map MC to represent the array C. The map MC is going store only true for the columns j such that $D(x, j)$ is true.
We can tune the algorithm a little more by dropping the elements that are larger than $b/2$, which would help reduce the size of the tableau. Also when moving through the columns, we are not considering the case where $S_1 = \{\}$, this scenario should be validated before, so we can start populating the columns always from the minimum element in the set A.

1.3 Karmarkar-Karp Algorithm

One deterministic heuristic for the Number Partition problem is the Karmarkar-Karp algorithm, or the KK algorithm. This approach uses *differencing*. The 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 decided 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. An algorithm based on differencing repeatedly takes two elements from A and performs a differencing until there is only one element left; this element equals an attainable residue. (A sequence of signs s_i that yields this residue can be determined from the differencing operations performed in linear time by two-coloring the graph (A, E) that arises, where E is the set of pairs (a_i, a_j) that are used in the differencing steps. You will not need to construct the s_i for this assignment.)

For the Karmarkar-Karp algorithm suggests repeatedly taking the largest two elements remaining in A at each step and differencing them. For example, if A is initially $(10, 8, 7, 6, 5)$, then the KK algorithm proceeds as in equation (3).

$$\begin{array}{rcl} (10, 8, 7, 6, 5) & \rightarrow & (2, 0, 7, 6, 5) \\ & \rightarrow & (2, 0, 1, 0, 5) \\ & \rightarrow & (0, 0, 1, 0, 3) \\ & \rightarrow & (0, 0, 0, 0, 2) \end{array} \tag{3}$$

Hence the KK algorithm returns a residue of 2. The best possible residue for the example is 0.

The fact that we are required to take the largest two elements in each step gives a hint of a sorting involved. If we have n elements, then sorting is going to take $O(n \log n)$. After each sorting, we remove the two largest element and insert the new nonzero value. If we have a sorted linked list, the insertion is going to be in $O(n)$, and for each step, we remove 2 elements and we add 1, so we are reducing the set by 1. Hence, we have the recurrence $T(n) = T(n-1) + n + 1$, which considers at most $n - 2$ comparisons to find the sorted place for the new element, as well as retrieving the two largest elements from the sorted linked list, which is done in 2 operations and another operation for the difference. The recurrence can be solved by the characteristic equation method, considering the polynomial in equation (4).

$$(x-1)(x-1)^2 = (x-1)^3 \tag{4}$$

The polynomial in (4) has only one root of multiplicity 3, so the proposed solution for the recurrence is shown in equation (5).

$$T(n) = c_1 + c_2 n + c_3 n^2 \tag{5}$$

Considering the base case for $T(1) = 1$, which just returns the only element in the set, then $T(2) = 4$ and $T(3) = 8$ and the general solution (5), we get the linear system of equations in equation (6).

$$\begin{array}{rcrcrcl} c_1 & + & c_2 & + & c_3 & = & 1 \\ c_1 & + & 2c_2 & + & 4c_3 & = & 4 \\ c_1 & + & 3c_2 & + & 9c_3 & = & 8 \end{array} \tag{6}$$

The linear system in equation (6) can be solved using Gauss as shown in (7).

$$
\begin{array}{llllll}
\left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 4 \\ 1 & 3 & 9 & 8 \end{array}\right) & \begin{array}{l} \\ R_2 - R_1 \\ R_3 - R_1 \end{array} & \sim & \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 3 \\ 0 & 2 & 8 & 7 \end{array}\right) & R_3 - 2R_2 & \sim \\
\left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 3 \\ 0 & 0 & 2 & 1 \end{array}\right) & \frac{1}{2}R_3 & \sim & \left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 3 \\ 0 & 0 & 1 & \frac{1}{2} \end{array}\right) & R_2 - 3R_3 & \sim \\
\left(\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & \frac{3}{2} \\ 0 & 0 & 1 & \frac{1}{2} \end{array}\right) & R_1 - R_3 & \sim & \left(\begin{array}{ccc|c} 1 & 1 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & \frac{3}{2} \\ 0 & 0 & 1 & \frac{1}{2} \end{array}\right) & R_1 - R_2 & \sim \\
\left(\begin{array}{ccc|c} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & \frac{3}{2} \\ 0 & 0 & 1 & \frac{1}{2} \end{array}\right) & & & & &
\end{array}
\tag{7}
$$

Based on the reduced form yielded in (7), we get that $c_1 = -1$, $c_2 = 3/2$ and $c_3 = 1/2$, which means that the recurrence is given exactly by equation (8).

$$T(n) = \frac{1}{2}n^2 + \frac{3}{2}n - 1 \tag{8}$$

We proceed to compute the first 3 values of (8) to verify that they match $T(1)$, $T(2)$ and $T(3)$, which is shown in equations (9), (10) and (11)

$$
\begin{array}{rcl}
T(1) & = & \frac{1}{2}(1)^2 + \frac{3}{2}(1) - 1 \\
& = & \frac{1}{2} + \frac{3}{2} - \frac{2}{2} \\
& = & \frac{2}{2} \\
& = & 1
\end{array}
\tag{9}
$$

$$
\begin{array}{rcl}
T(2) & = & \frac{1}{2}(2)^2 + \frac{3}{2}(2) - 1 \\
& = & \frac{1}{2}(4) + \frac{6}{2} - \frac{2}{2} \\
& = & \frac{4}{2} + \frac{6}{2} - \frac{2}{2} \\
& = & \frac{8}{2} \\
& = & 4
\end{array}
\tag{10}
$$

$$
\begin{array}{rcl}
T(3) & = & \frac{1}{2}(3)^2 + \frac{3}{2}(3) - 1 \\
& = & \frac{1}{2}(9) + \frac{9}{2} - \frac{2}{2} \\
& = & \frac{9}{2} + \frac{9}{2} - \frac{2}{2} \\
& = & \frac{16}{2} \\
& = & 8
\end{array}
\tag{11}
$$

As induction hypothesis, we assume that the solution is true, so we have to prove that the recurrence equation is also valid for $n + 1$, which is done in equation (12).

$$\begin{aligned} T(n+1) &= T(n) + n + 2 \\ &= \tfrac{1}{2}n^2 + \tfrac{3}{2}n - 1 + n + 2 \\ &= \tfrac{1}{2}n^2 + n + \tfrac{1}{2} + \tfrac{3}{2}n + \tfrac{3}{2} - 1 \\ &= \tfrac{1}{2}(n^2 + 2n + 1) + \tfrac{3}{2}(n+1) - 1 \\ &= \tfrac{1}{2}(n+1)^2 + \tfrac{3}{2}(n+1) - 1 \end{aligned} \tag{12}$$

Therefore, the suggested approach has a running time of $O(n^2)$.

The algorithm in each step takes two elements and yields another 2 but we are going to consider only the nonzero, so we can place back into the set A the nonzero element, reducing the size of the set by 1. If we want to reduce the running time of the algorithm, we can consider the general recurrence $T(n) = T(n-1) + O(f(n))$, where $O(f(n))$ is the running time for deleting the two largest elements and inserting the new one.
Using a linked list, $f(n) = n$, which means that we must use a data structure which allows to delete an element and insert a new one in running time better than $O(n)$.

The best that we can do is to have $f(n) = c$, so we could have that the recurrence will be $T(n) = T(n-1) + O(1)$, which can be solved considering the polynomial $(x-1)^2$, and hence we have that $T(n) = c_1 + c_2 n$. The base case is still $T(1) = 1$, when we have only one element, and when we have $T(2) = c + 1$, then we have the linear system shown in equation (13).

$$\begin{aligned} T(1) &= c_1 + c_2 = 1 \\ T(2) &= c_1 + 2c_2 = c + 1 \end{aligned} \tag{13}$$

If we consider $c_1 = 1 - c_2$, then we have that $1 - c_2 + 2c_2 = c + 1$, from which we get that $c_2 = c$ and $c_1 = 1 - c$. Therefore the recurrence has the solution $T(n) = c \cdot n + 1 - c$, and we can prove it by considering the base case $T(1) = c(1) + 1 - c = c + 1 - c = 1$ and then assuming that it is true for n. For inductive step, we have to prove that $T(n+1) = c \cdot n + 1$, which is done in equation (14).

$$\begin{aligned} T(n+1) &= T(n) + c \\ &= c \cdot n + 1 - c + c \\ &= c \cdot n + 1 \end{aligned} \tag{14}$$

Therefore, the best theoretical running time for the Karmarkar-Karp algorithm is $O(n)$. This theoretical running time can be achieve if we are able to design a data structure that can be built in $O(n)$ and the operations findMax, insert and delete are all of them in $O(1)$. This bound can be achieved with count sort (pigeonhole sort) as long as the values of the elements are in the size of $O(n)$.
We first iterate over the elements to find the maximum value, which is done in $O(n)$, then we create an array A of elements of dimension $n + 1$ to include the number 0. Then we iterate over each element i and then increase its corresponding bucket in array A, as $A[i] = A[i] + 1$; this also takes $O(n)$. The way usually this algorithm is used, the elements in A are placed again in the original array so we waste unnecessary space. We are going to continue using the array A, since after each step of the Karmarkar-Karp algorithm, a

new element is inserted into the array. The insertion is going to be in $O(1)$ as well as deletion with the operation $A[i] = A[i] - 1$.
The only thing that remains is the findMax operation in $O(1)$. In the worst case, this is going to take $O(n)$, which takes us to our original with case of $f(n)$ in $O(n)$. We can amortized the cost of findMax by considering that the size of the array is in $O(n)$, i.e., at the end of the Karmarkar-Karp algorithm, we would have traverse the whole array by assigning a constant cost for each findMax operation equal to the constant k which bounds the size of the array to less than or equal to $k \cdot n$. Therefore, we have the Karmarkar-Karp algorithm running in $O(n)$, which is also in $O(n \log n)$.

In case that the size of the integers is not in (n), then we would need to use a different approach. Since we require the largest two elements, this might involve sorting in $\Omega(n \log n)$ or building a heap in $O(n \log n)$. With either option, the operation findMax can be achieve in $O(1)$ but insertion takes $O(\log n)$ with a binary heap and $O(n)$ if the sorted elements are stored a linked list. If the sorted elements are stored in an AVL three the insertion and deletion run in $O(\log n)$ but findMax will also be in $O(\log n)$. If we choose to use a heap to store the elements, then we will have that $f(n) = 3 \log n + 3$, because 2 deletions, 2 findMax and 1 insertion. This yields the recurrence $T(n) = T(n-1) + 3 \log n + 3$ that can be solved by expanding the recurrence until we reach a general case as in equation (15).

$$
\begin{aligned}
T(n) &= T(n-1) + 3\log n + 3 \\
&= T(n-2) + 3\log(n-1) + 3 + 3\log n + 3 \\
&= T(n-2) + 3\log(n-1) + 3\log n + 2(3) \\
&= T(n-3) + 3\log(n-2) + 3 + 3\log(n-1) + 3\log n + 2(3) \\
&= T(n-3) + 3\log(n-2) + 3\log(n-1) + 3\log n + 3(3) \\
&= T(n-4) + 3\log(n-3) + 3 + 3\log(n-2) + 3\log(n-1) + 3\log n + 3(3) \\
&= T(n-4) + 3\log(n-3) + 3\log(n-2) + 3\log(n-1) + 3\log n + 4(3) \\
&\vdots \\
&= T(n-i) + 3\sum_{k=0}^{i-1} \log(n-k) + 3i \qquad (15)
\end{aligned}
$$

If we consider $i = n-1$, we would have that $T(n) = T(1) + 3\sum_{k=0}^{n-2} \log(n-k) + 3(n-1)$, which can be expressed as $T(n) = T(1) + 3\sum_{k=2}^{n} \log(k) + 3(n-1)$. By previous analysis we know that $\sum_{k=2}^{n} \log(k)$ is in $\Theta(n \log n)$, which implies that using a heap structure, the Karmarkar-Karp algorithm runs in $O(n \log n)$.