Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

April 30, 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

```
funcion populateTableau(Array $A$)
Long $b = \sum_{a \in A} a$
Long minimum = A.minimum
boolean[] tableau = new boolean[$A.size$][$b+1$]
tableau[0][minimum] = true // initialize tableau
for (int i = 1; i < A.size; i++) do
    for (long j = 1; j $\leq$ b; j++) do
        if tableau[i - 1][j] or tableau[i - 1][j - A[i]] then
            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).

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

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 $x \in [1, \dots n]$. 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 $O(b)$

```
funcion populateTableau(Array $A$)
Long $b = \sum_{a \in A} a$
Long minimum = A.minimum
boolean[][] tableau = new boolean[$A.size$][$b + 1$]
boolean[] C = new boolean[$b + 1$]
tableau[0][minimum] = true // initialize tableau
for (int i = 1; i < A.size; i++) do
    for (long j = 1; j $\leq$ b; j++) do
        if tableau[i - 1][j] or tableau[i - 1][j - A[i]] then
            tableau[i][j] = true;
            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 $O(b)$

```
funcion findSumThatMinimizes(Long b)
Long $b = \sum_{a \in A} a$
for (long $m = b/2$; $m > 0$; $m = m - 1$) do
    if $C[m] ==$ true then
        return $|m - b/2|$
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.
Another tunning would be to reduce the size of the tableau from $n \cdot b$ to $n \cdot \lceil \frac{b}{2} \rceil$, since we do not require the values from $\lceil \frac{b}{2} \rceil$.

Populating the tableau requires a time in $O(n \cdot b)$ and getting the minimum element is in $O(b)$, so the total running time is $O(n \cdot b)$.

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).

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

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).

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

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

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

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)

$$
\begin{aligned}
T(1) &= \tfrac{1}{2}(1)^2 + \tfrac{3}{2}(1) - 1 \\
&= \tfrac{1}{2} + \tfrac{3}{2} - \tfrac{2}{2} \\
&= \tfrac{2}{2} \\
&= 1
\end{aligned}
\tag{9}
$$

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

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

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).

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

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).

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

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).

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

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 $O(n \log n)$.

1.4 Karmarkar-Karp algorithm and heuristics

You will compare the Karmarkar-Karp algorithm and a variety of randomized heuristic algorithms on random input sets. Let us first discuss two ways to represent to the problem and the state space based on these representations. Then we discuss heuristics search algorithms you will use.

The standard representation of a solution is simply as a sequence S of +1 and -1 values. A random solution can be obtained by generating a random sequence of n such values. Thinking of all possible solutions as a state space, a natural way to define neighbors of a solution S is as the set of all solutions that differ from S in either one or two places. This has a natural interpretation if we think of the +1 and -1 values as determining two subsets A_1 and A_2 of A. Moving from S to a neighbor is accomplished either by moving one or two elements from A_1 to A_2, or moving one or two elements from A_2 to A_1, or swapping a pair of elements where one is in A_1 and one is in A_2.

A *random move* on this state space can be defined as follows. Choose two random indices i and j from $[1, \ldots, n]$ with $i \neq j$. Set s_i to $-s_i$ and with probability 1/2, set s_j to $-s_j$.

An alternative way to represent a solution called *prepartitioning* is as follows. We represent a solution by a sequence $P = \{p_1, p_2, \ldots, p_n\}$ where $p_i \in \{1, \ldots, n\}$. The sequence P represents a repartitioning of the elements of A, in the following way: if $p_i = p_j$, then we enforce the restriction that a_i and a_j have the same sign. Equivalently, if $p_i = p_j$, then a_i and a_j both lie in the same subset, either A_1 or A_2.

We turn a solution of this form into a solution in the standard form using two steps:

1. We derive a new sequence A' from A which enforces the prepartitioning from P. Essentially A' is derived by resetting a_i to be the sum of all values j with $p_j = i$, using for example the pseudocode in algorithm 4.

Algorithm 4: Pseudocode to derive A'

```
1 $A' = (0, 0, \ldots, 0)$
2 for $j = 1$ to $n$ do
3     $a'_{p_j} = a'_{p_j} + a_j$
```

2. We run the KK heuristic algorithm on the result A'.

For example, if A is initially (10, 8, 7, 6, 4), the solution $P = (1, 2, 2, 4, 5)$ corresponds to the following run of the KK algorithm:

$$\begin{aligned} A = (10, 8, 7, 6, 5) &\rightarrow A' = (10, 15, 0, 6, 5) \\ (10, 15, 0, 6, 5) &\rightarrow (0, 5, 0, 6, 5) \\ &\rightarrow (0, 0, 0, 1, 5) \\ &\rightarrow (0, 0, 0, 0, 4) \end{aligned} \tag{16}$$

Hence in this case the solution P has a residue of 4.

Notice that all possible solution sequences S can be regenerated using this prepartition representation, as any split of A into sets A_1 and A_2 can be obtained by initially assigning p_i to 1 for all $a_i \in A_1$ and similarly assigning p_i to 2 for all $a_i \in A_2$.

A random solution can be obtained by generating a sequence of n values in the range $[1, \ldots, n]$ and using this for P. Thinking of all possible solutions as a state space, a natural way to define neighbors for a solution P is as the set of all solutions that differ from P in just one place. The interpretation is that we change the repartitioning by changing the partition of one element. A *random move* on this state space can be defined as follows. Choose two random indices i and j from $[1, \ldots, n]$ with $p_i \neq j$ and set p_i to j.

You will try each of the following three algorithms for both representations.

- *Repeated random:* Repeatedly generate random solutions to the problem, as determined by the representations, as seen in the pseudocode in algorithm 5

Algorithm 5: Repeated random

```
1 Start with a random solution $S$
2 for $iter = 1$ to $max\_iter$ do
3     $S' =$ a random solution
4     if $residue(S') < residue(S))$ then
5         $S = S'$
6 return $S$
```

- *Hill climbing:* Generate a random solution to the problem, and then attempt to improve it through moves to better neighbors. The pseudocode can be seen in algorithm 6.

Algorithm 6: Hill climbing

```
1 Start with a random solution $S$
2 for $iter = 1$ to $max\_iter$ do
3     $S' =$ a random neighbor of $S$
4     if $residue(S' < residue(S))$ then
5         $S = S'$
6 return $S$
```

- *Simulated annealing:* Generate a random solution to the problem, and then attempt to improve it through moves to neighbors, that are not always better. The pseudocode can be seen in algorithm 7.

  Note that for simulated annealing, we have the code return the best solution seen thus far.

Algorithm 7: Simulated annealing

```
1 Start with a random solution $S$
2 $S'' = S$
3 for $iter = 1$ to $max\_iter$ do
4     $S'$ = a random neighbor of $S$
5     if $residue(S' < residue(S))$ then
6         $S = S'$
7     else
8         $S = S'$ with probability $e^{-\frac{\text{residue}(S') - \text{residue}(S)}{\text{T(iter)}}}$
9 return $S$
```

You will run experiments on sets of 100 integers, with each integer being a random number chosen uniformly from the range $[1, \ldots, 10^{12}]$. Note that these are big numbers. You should use 64 bit integers. Pay attention to things like whether your random number generator works on ranges this large!

Now we proceed to generate 50 random instances of the problem as described below. 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 simulating annealing algorithm, using both representations, each for at least 25,000 iterations. We are going to give tables and/or graphs clearly demonstrating the results, giving both the numerical results, and the time taken by the algorithms. We are going to compare the results and discuss.

For the simulated annealing algorithm, you must choose a *cooling schedule.* That is, you must choose a function T(tier). We suggest $\text{T(iter)} = 10^{10}(0.8)^{\lfloor \text{tier}/300 \rfloor}$ for numbers in the range $[1, \ldots 10^{12}]$, but you can experiment with this as you please.

Note that, in our random experiments, we began with a random initial starting point.

The approach to develop this section is first to generate the random numbers and store them in a file. We are going to generate 50 plain text files and for each file, 100 integers in the range $[1, \ldots, 10^{12}]$. The integers are going to be distinct since having a repeated element would reduce the complexity of the problem from 100 to 98 since we can place one of the repeated elements in A_1 and the other in A_2.

The program is developed in Java SE 1.6 on a Macbook late 2008, which presents some constraints due to the size of the integers. The range of the integers are not considered integers in Java, but long integers. This puts a constraint if we would want to also run the experiments using the dynamic programming approach, to validate the results from the heuristics, since Java cannot create an array whose size is in the range of long integers. The sparse matrix approach might reduce the complexity but it still takes too long since it has to populate a big tableau.

Java provides a way to generate a big random integer using the BigInteger class along with the Random class. Also, if we consider all the numbers in $[1, \ldots, 10^{12}]$ as 13 digits number with padding 0s in the left, then a random long integer can be seen as an array of 13 digits in which the first digit can be 0 or 1 and the other 12 digits can take the random values in $[0, 2, \ldots, 9]$.

The heap to implement the Karmarkar-Karp heuristic uses the priority queue that is already built in Java. The algorithm was implemented in a recursive way and in each step, the largest two elements are removed from the heap. Only the difference is added to the heap since the 0 value does not add any value to the algorithm. Also if the difference is 0, it will also not be added to the heap.
The stop condition is if the heap is empty, which means that the residue found by the algorithm is the minimum possible and it returns 0. If the heap only has one element, then that is the residue and the algorithm finishes.

For Random Repeated, we start with two lists representing A_1 and A_2 and we iterate over all the elements in A and randomly we select if we put the element a_i in A_1 or in A_2. For all the algorithms, the list A is sorted first, so we start with possible residue equals to the greatest element in A. We get the sum S_1 for A_1 and the sum S_2 for A_2 and we the residue, if the residue is less than our possible residue, then we replace it. If the residue is 0, then we stop, otherwise, we continue until we finish iterating and then we return the possible residue.

For Hill Climbing we start with a random solution similar to how we randomly generate a solution in Random Repeated. The neighbor is found by randomly generate a number i between 0 and $|A_1|$ and another number j between 0 and $|A_2|$. In our case we iterate until j is not the same as i but it does not matter if $i = j$ since the indices are for two different sets.
After the indices have been generating, the i-th element in A_1 is swap with the j-th element in A_2 and then the residue is generating by getting the total sums S_1 and S_2 and then we get the absolute difference. If the neighbor solution is better than the proposed solution, then the proposed solution is equal to the neighbor solution and the minimum variable is updated.

For Simulated Annealing, we start with a generated random solution as in Hill Climbing, but besides keeping tracking of the proposed solution and the neighbor solution, we also keep track of another solution S''.The neighbor is generated as in Hill Climbing but if the neighbor solution is not better than the proposed solution, we move to the neighbor solution as long as the probability based on the *cooling schedule* $\mathrm{T(tier)} = 10^{10}(0.8)^{\lfloor \mathrm{tier}/300 \rfloor}$. Finally we keep track of the best solution seen so far with S''. If at some point by moving to the neighbor solution, the residue is better than the residue in S'', then we replace S'' with this solution.

For the methods that use the Karmarkar-Karp algorithm, we prefix them with *Mixed*. Instead of randomly swapping elements between A_1 and A_2, we randomly generate a array of size $|A|$, with elements in $[1, \ldots, |A|]$. After the partition is generated, we get an array A' and then from the array we create list of non zero elements and we apply the Karmarkar-Karp program that we already have. If we residue that we have is better than the current one (which initially is the largest element in the array), then we replace it, otherwise we continue until the iterations are completed or we reach a residue of 0.

For Mixed Hill Climbing, we start with a random partition generated just as in Mixed Random Repeated and then we get a random neighbor by generating a random number i which will be the value to be replaced and then we generate another random integer j such that it is not the same as $S[i]$ and then we assign j to $S[i]$. Notice that i and j are in the range $[1, \ldots, |A|]$.
After each random partition is generated, then we apply KK and compare the residue with the current minimum, if the residue is less than the current minimum, then it is replaced, as well as the current partition by the neighbor partition.

For Mixed Simulated Annealing, it is like Simulated Annealing but for the partition as in Hill Climbing.

To run the experiments, several test cases were written. The methods that generated the same value for the same input, have test cases to validate the result. The other heuristic methods have test cases only to run the experiment.

1.5 Results

In Table 1 we can appreciate that KK algorithm runs faster than any other proposed heuristic, which makes sense considering the simple operations that are done compared to Random Repeated and Hill Climbing which have to compute the residue of each sum at each step.
The best residue that K was able to produce is 758 for file 46, for which Random Repeated and Hill Climbing produced terrible residues. The execution time for Random Repeated and Hill Climbing did not change that much.

Since Random Repeated does not consider the current state of the solution, we are practically just iterating and hope to be lucky enough to get a combination that would generate a better residue.
Hill Climbing also is based on random swapping but it will moving to a better neighbor and iterating for another one might get us stuck into a local optima that is far from the one that we would expect.

Table 1: Comparing results with KK, Random Repeated and Hill Climbing

	KK		Random Repeated		Hill Climbing	
File	Residue	Milliseconds	Residue	Milliseconds	Residue	Milliseconds
1	114182	92	124122290	513	323327088	645
2	142542	51	687210366	397	106455160	276
3	218382	52	36606166	269	288046066	256
4	172671	18	673420835	268	50523635	245
5	18946	43	118084310	260	75153670	265
6	77096	40	36200566	298	472988564	251
7	141706	17	290976336	279	68465818	255
8	59282	14	313058340	407	168428118	229
9	165544	60	377751230	341	9664978	253
10	1094350	8	82735178	349	520089022	275
11	157322	6	137124322	362	523831054	243
12	17389	24	246837135	338	57352739	258
13	133522	6	519902896	416	305093332	228
14	426552	20	168683632	312	325835110	262
15	121555	10	79513781	241	25971579	247
16	359984	19	456108672	423	72090110	251
17	92510	4	526761610	266	331717516	255
18	174338	15	216579114	255	142002284	232
19	482968	5	604771234	287	196379512	240
20	113902	15	546133570	254	182198260	255
21	191888	6	322730786	268	74419550	255
22	149623	5	532788945	268	70517311	240
23	1174258	15	446543408	277	359606686	324
24	39179	4	75216663	271	632681317	319
25	70712	14	176375630	276	93852932	259
26	161571	4	41853471	259	1903253797	248
27	56562	16	353456552	263	740153880	299
28	2760	4	170741138	284	539330572	280
29	114192	4	164227170	267	84164316	281
30	41626	38	11218250	264	198030510	271
31	52594	3	481985390	249	187076024	238
32	397885	9	709146039	286	125592209	254
33	451550	4	154726820	266	907916102	262
34	1073370	28	321383432	243	459510096	260
35	209575	4	358859047	270	9427843	237
36	182143	18	1336597985	331	312750889	249
37	41566	3	263114974	258	727491300	240
38	109685	4	63529643	262	38336147	294
39	34183	19	169109701	267	542152423	264
40	419170	4	382161400	260	625390808	243
41	59050	4	142152116	271	821492128	247

42	115695	11	535822457	250	111365141	254
43	23795	4	208174055	280	784144699	263
44	532410	3	296235488	256	556561466	245
45	82125	47	87212219	273	224917259	247
46	758	3	93110072	257	273460596	241
47	314069	11	80326695	273	334603447	252
48	3541	65	457741769	271	621652513	253
49	222488	3	94748350	263	176140458	254
50	70647	2	289828027	266	224036297	240

In Table 2, we can appreciate that also Simulated Annealing did not perform very well in terms of residue. Moving to worse neighbors really impact in the result generating extremely bad residues.
Also the running time for Simulated Annealing increases due to the operations that have to be computed to decide if we should move to a bad neighbor or not.
The residues really improve when we mixed heuristics. As can be appreciated in the Mixed Random Repeated (MRR) column, the residues are very small compared to KK. The best result was for file 18, with a residue of 4, and for the file 46, the residue was 156 which beats KK in its base case, and every other one.

The only inconvenience with MRR is its running time. The running time of MRR is almost as twice of Simulated Annealing and almost as 4 times what Repeated Random and Hill Climbing take. This makes sense considering that Repeated Random executes in $O(|A|)$, since the generation of random indices and swapping takes constant time, the only thing that takes time for Repeated Random is calculating the sum over A.
For the mixed scenario, we do not sum over A, but we apply the algorithm over an array with size in $O(|A|)$, since after prepartitioning we might remove some elements and the array might be smaller.

The cost of MRR resides in KK, so it runs in $O(|A| \log |A|)$, which we considered very good according to the residues that it gets.

Table 2: Comparing results with KK, Simulated Annealing and Mixed Random Repeated

	KK		Simulated Annealing		Mixed Random Repeated	
File	Residue	Milliseconds	Residue	Milliseconds	Residue	Milliseconds
1	114182	92	5628844163290	886	262	1191
2	142542	51	7217281758456	620	372	885
3	218382	52	2878871546202	573	102	881
4	172671	18	666258108611	610	307	869
5	18946	43	7632874636326	590	408	957
6	77096	40	934620286296	603	410	1148
7	141706	17	7162116942232	582	474	868
8	59282	14	597396995426	656	38	835
9	165544	60	11892733048032	581	6	852

10	1094350	8	2301236163138	579	56	875
11	157322	6	568843491078	587	24	842
12	17389	24	7565861670717	570	53	879
13	133522	6	4683919243710	579	82	856
14	426552	20	3424946559084	613	478	849
15	121555	10	583856455495	639	31	853
16	359984	19	9885670576046	556	474	909
17	92510	4	3498918646720	607	6	838
18	174338	15	4529287452558	589	4	880
19	482968	5	237340093792	560	226	866
20	113902	15	6422793728994	597	144	1064
21	191888	6	8016240362984	676	142	861
22	149623	5	499952415977	564	25	862
23	1174258	15	4072419307264	592	240	857
24	39179	4	2867083895877	576	155	873
25	70712	14	130915334860	591	172	861
26	161571	4	8096375661177	586	21	852
27	56562	16	182986537564	580	390	867
28	2760	4	362811874734	577	230	1005
29	114192	4	2435593865940	569	642	918
30	41626	38	7206058598972	602	138	822
31	52594	3	2675618126524	579	200	926
32	397885	9	1704894217047	556	131	1136
33	451550	4	3683242392588	579	110	948
34	1073370	28	4386435495758	579	28	884
35	209575	4	4349655004671	570	105	839
36	182143	18	13260978395405	607	13	881
37	41566	3	9691243560948	592	50	843
38	109685	4	5041972691309	610	61	866
39	34183	19	891678421687	764	179	834
40	419170	4	2776743313208	629	38	887
41	59050	4	92442205666	637	360	830
42	115695	11	112816335279	567	253	867
43	23795	4	3134687481497	590	163	824
44	532410	3	1062467313616	582	52	955
45	82125	47	1403159039103	598	105	891
46	758	3	561503245280	555	156	876
47	314069	11	8590215375767	562	195	872
48	3541	65	6012728597293	584	73	836
49	222488	3	5091469271684	586	56	877
50	70647	2	5774852634595	566	191	856

In Table 3 and based on the results from Table 2, we can appreciate that the mixed heuristics produced betters residues.

Among the 3 mixed heuristics, Mixed Repeated Random produced the best results. Mixed Hill Climbing produced better results than simple KK and the simple heuristics but the execution time is close to Mixed Repeated Random but the results are not that good.
Still the best result for Mixed Hill Climbing was for file 17, which beat Mixed Random Repeated and Mixed Simulated Annealing. Still Mixed Random Repeated produced better results than Hill Climbing but with the trade-off of running slower.
The best result for Mixed Simulated Annealing was file 10, which also beats Mixed Hill Climbing and Mixed Random Repeated. Mixed Hill Climbing and Mixed Simulated Annealing generate kind of similar results but over different files. The only inconvenience with Mixed Simulated Annealing is the running time which is the heuristic that performed poorly, given that it took around 3 times what Mixed Random Repeated took. This might be able to be improved if we simplify the function that determines the probability to move to a bad neighbor, as well as the function T.

Table 3: Comparing results with KK, Mixed Hill Climbing and Mixed Simulated Annealing

	KK		Mixed Hill Climbing		Mixed Simulated Annealing	
File	Residue	Milliseconds	Residue	Milliseconds	Residue	Milliseconds
1	114182	92	600	887	614	3118
2	142542	51	1202	718	94	2935
3	218382	52	1148	651	636	2499
4	172671	18	553	703	261	2696
5	18946	43	54	712	204	2660
6	77096	40	994	743	186	2776
7	141706	17	3034	719	1166	2816
8	59282	14	638	754	18	2757
9	165544	60	1088	784	282	2816
10	1094350	8	566	678	4	2861
11	157322	6	1394	683	262	2804
12	17389	24	9	804	113	2723
13	133522	6	468	716	234	3171
14	426552	20	1468	769	8	3050
15	121555	10	561	631	33	2790
16	359984	19	1170	793	402	2660
17	92510	4	4	678	722	2572
18	174338	15	328	988	106	2672
19	482968	5	244	847	400	2832
20	113902	15	66	987	890	2583
21	191888	6	76	696	50	2716
22	149623	5	4693	666	409	2707
23	1174258	15	42	698	22	2858
24	39179	4	1189	675	169	2835
25	70712	14	476	683	88	2799
26	161571	4	245	721	189	2649

27	56562	16	466	704	276	2807
28	2760	4	18	697	504	2862
29	114192	4	20	774	232	2735
30	41626	38	2548	768	22	2743
31	52594	3	1322	625	214	2657
32	397885	9	113	672	131	2714
33	451550	4	2858	672	170	2871
34	1073370	28	54	801	102	3247
35	209575	4	327	737	11	2883
36	182143	18	1835	732	119	2838
37	41566	3	398	694	18	2700
38	109685	4	153	767	333	3078
39	34183	19	1259	765	27	2801
40	419170	4	58	749	36	3130
41	59050	4	1358	739	166	2651
42	115695	11	869	751	303	2814
43	23795	4	963	737	167	2816
44	532410	3	2258	728	774	2623
45	82125	47	103	683	261	2519
46	758	3	228	624	58	2853
47	314069	11	449	864	447	2800
48	3541	65	1591	729	285	2757
49	222488	3	3194	726	998	2914
50	70647	2	183	630	17	3157

1.6 Mixing Karmarkar-Karp with heuristics

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

1.7 Running the Karkarmar-Karp program

The program is already compiled and package in the root directory so it can run as follows:

```
 java -jar NumberPartition.jar input.txt
```

If the user would like to try it in a different environment or recompile it again from scratch, you would only need to run the following two commands:

```
mvn clean compile assembly:single
mvn assembly:assembly
```

After the command second command is executed, it can be killed when running the test cases.

The program was tested in the NICE system by connecting through the command:

```
ssh cgarciavazquez@nice.fas.harvard.edu
```