CS 124 Programming Assignment 3: Spring 2023

Your name(s) (up to two): Luke Richey

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: 9
No. of late days used after including this pset: 10

Homework is due Thursday 2023-04-20 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

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*

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

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

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 $n \log b$. Instead you should find a dynamic programming algorithm that takes time polynomial in nb.

Give a dynamic programming solution to the Number Partition problem.

Solution:

Since the Number Partition problem has overlapping subproblems, it lends itself to a dynamic programming solution. Let $D[i][j]$ be a boolean representing whether the sum of the first i elements of A is j. Then, we will define the recurrence of $D[i][j]$ as follows:

$$D[i][j] = \begin{cases} \text{true} & \text{if } i = 0, j = 0 \\ \text{false} & \text{if } i = 0, j \neq 0 \\ D[i-1][j] & \text{if } j < A[i] \\ D[i-1][j] \vee D[i-1][j - A[i]] & \text{otherwise} \end{cases}$$

Now, we need to actually find these subsets. Let $S[i][j]$ be a subset of A consisting of its first i elements that sums to j. $D[i][j]$ tells us whether there existence of such a subset, but not which one. Then, we will define the recurrence of $S[i][j]$ as follows:

$$S[i][j] = \begin{cases} \emptyset & \text{if } i = 0 \\ S[i-1][j] & \text{if } D[i-1][j] = \text{true} \\ S[i-1][j - A[i]] \cup \{A[i]\} & \text{if } D[i-1][j - A[i]] = \text{true} \\ \emptyset & \text{otherwise} \end{cases}$$

Now that we have defined the recurrences, we can find the partitions. The algorithm works as follows:

1. For each $i = 0$ to n elements in A, we will then iterate through all possible sums j from 0 to $\sum_{k=0}^{k=n} A[k]$, each time calculating $D[i][j]$ and $S[i][j]$, where i is the outer loop and j is the inner loop.
2. We will then try to find a subset of A of size n that sums to $a = \frac{\sum_{k=0}^{k=n} A[k]}{2}$, by checking if $D[n][a]$ is true and decrementing a until we find a subset of size n that sums to a. Note that we will round a so that it is an integer.
3. If we find a subset of size n that sums to a, then we will return the subset $D[n][a]$ as the partition with the smallest residue. Otherwise, we will return the empty set. Additionally, we will return $A \setminus S[n][a]$ as the other partition.

Proof of correctness:

Proof. We will prove the correctness of our algorithm by checking the correctness of our recurrences.

D[i][j]

In the base case, that is, $i = 0$ and $j = 0$, we have that $D[0][0] = \text{true}$, since the sum of the first 0 elements of A is 0. Additionally, we have that $D[0][j] = \text{false}$ for all $j \neq 0$, since the sum of the first 0 elements of A is 0 and $j \neq 0$.

In the case that $i \neq 0$ and $j < A[i]$, we have that $D[i][j] = D[i-1][j]$, since we cannot add $A[i]$ to the sum of the first $i-1$ elements of A to get j. We then correctly search for a subset of A of size $i-1$ that sums to j.

In the case that $i \neq 0$ and $j \geq A[i]$, we have that $D[i][j] = D[i-1][j] \vee D[i-1][j - A[i]]$, since we can either add $A[i]$ to the sum of the first $i-1$ elements of A to get j, or we can search for a subset of A of size $i-1$ that sums to $j - A[i]$. We then correctly search for a subset of A of size $i-1$ that sums to j or a subset of A of size $i-1$ that sums to $j - A[i]$.

Thus, we have proven that $D[i][j]$ is correctly defined.

S[i][j]

In the base case, that is, $i = 0$, we have that $S[0][j] = \emptyset$, since there is no subset of A of size 0 that sums to j.

In the case that $i \neq 0$ and $D[i-1][j] = \text{true}$, we have that $S[i][j] = S[i-1][j]$, since there exists a subset of A of size $i-1$ that sums to j, by the correctness of $D[i-1][j]$.

In the case that $i \neq 0$ and $D[i-1][j-A[i]] = \text{true}$, we have that $S[i][j] = S[i-1][j-A[i]] \cup \{A[i]\}$, since there exists a subset of A of size $i-1$ that sums to $j - A[i]$ and then add $A[i]$ to it to get a subset of A of size i that sums to j.

Otherwise, we have that $S[i][j] = \emptyset$, since there is no subset of A of size i that sums to j. Thus, we have proven that $S[i][j]$ is correctly defined.

We will now prove that our algorithm correctly finds the partition with the smallest residue. We start our search by setting our target sum a to be $\frac{\sum_{k=0}^{k=n} A[k]}{2}$. This initial value is correct since the maximum possible sum for the two partitions of A is half of the sum of all the elements in A. We then check if $D[n][a]$ is true. If it is, then we have found a subset of A of size n that sums to a. As we decrement a, we are bound to find the partition minimizing the residue of A.

□

We will now prove that, even though the Number Partition problem is NP-complete, our algorithm runs in pseudopolynomial time.

Proof. We will now prove the runtime of our algorithm. Let's first suppose that the elements of A sum to some number b. It then follows that each element of A has at most $\log b$ bits.

In the first step of our algorithm, we must first initialize the arrays D and S. There are then n elements summing to b, so we initialize both arrays to be of size at most $n \times b$. Thus, the initialization of D and S takes $\mathrm{O}(nb)$ time.

Additionally, we must also fill these arrays as described in our algorithm. Since there are nb subproblems that we are solving, and each takes $\mathrm{O}(1)$ time, then the runtime to fill the arrays is $\mathrm{O}(nb)$.

Finally, we must also search for the partition with the smallest residue. We calculate $a = \frac{b}{2}$. As stated earlier, each element of A has at most $\log b$ bits of which there are n. Thus, the runtime to compute b is $\mathrm{O}(n \log(b))$. We then divide b by 2, which takes $\mathrm{O}(\log b)$ time. Rounding a down to the nearest integer takes $\mathrm{O}(1)$ time.

We then check if $D[n][a]$ is true for the largest possible a. In the worst case, we must decrement a until it is 0. Since a is at most $\frac{b}{2}$ the runtime to find $D[n][a]$ is $\mathrm{O}(b)$. We also take the set difference to get the other partition of A, which takes $\mathrm{O}(n)$ time.

Our overall runtime can be found by summing the runtimes of each step. Thus, we have that:

$$\text{Runtime} = O(nb) + O(nb) + O(n \log(b)) + O(\log b) + O(b) + O(n) = O(nb)$$

□

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 decide 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 intially $(10, 8, 7, 6, 5)$, then the KK algorithm proceeds as follows:

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

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

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.

Solution: We would like to be able to take the two largest elements in A efficiently. We can do so by using a max-heap, which takes $O(n)$ time to construct. We can then take the two largest elements in A using the heap's extract-max function, which will take $O(\log n)$ time. We can then perform the differencing operation on these two elements, say a_i and a_j which will take $O(1)$ time. We can then insert these two elements back into the heap, which will take $O(\log n)$ time. We will continue this process until there remains one nonzero element in A. Since there are n elements in A and our operations take $O(\log n)$ time, then the runtime of the KK algorithm is $O(n \log n)$.

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 solutions to the problem and the state space based on these representations. Then we discuss heuristic 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 $[1, n]$ 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 prepartitioning 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:

- We derive a new sequence A' from A which enforces the prepartioning 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 following pseudocode:

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

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

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

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

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

Notice that all possible solution sequences S can be generated 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 $[1, n]$ and using this for P. Thinking of all possible solutions as a state space, a natural way to define neighbors of 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 prepartitioning 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 $[1, n]$ 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 representation.

Start with a random solution S
for iter = 1 to max_iter
    S' = a random solution
    if residue(S') < residue(S) **then** $S = S'$
return S

- Hill climbing: generate a random solution to the problem, and then attempt to improve it through moves to better neighbors.

Start with a random solution S
for iter = 1 to max_iter
    S' = a random neighbor of S
    if residue(S') < residue(S) **then** $S = S'$
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.

Start with a random solution S
$S'' = S$
for iter = 1 to max_iter
    S' = a random neighbor of S
    if residue(S') < residue(S) **then** $S = S'$
    else $S = S'$ with probability exp(−(res(S')-res(S))/T(iter))
    if residue(S) < residue(S'') **then** $S'' = S$
return S''

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

You will run experiments on sets of 100 integers, with each integer being a random number chosen uniformly from the range $[1, 10^{12}]$. 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!

Below is the main problem of the assignment.

First, write a routine that takes three arguments: a flag, an algorithm code (see Table 1), and an input file. We'll run typical commands to compile and execute your code, as in programming assignment 2; for example, for C/C++, the run command will look as follows:

$ ./partition flag algorithm inputfile

The flag is meant to provide you some flexibility; the autograder will only pass 0 as the flag but you may use other values for your own testing, debugging, or extensions. The algorithm argument is one of the values specified in Table 1. You can also assume the inputfile is a list of 100 (unsorted) integers, one per line. The desired output is the residue obtained by running the specified algorithm with these 100 numbers as input.

Code	Algorithm
0	Karmarkar-Karp
1	Repeated Random
2	Hill Climbing
3	Simulated Annealing
11	Prepartitioned Repeated Random
12	Prepartitioned Hill Climbing
13	Prepartitioned Simulated Annealing

Table 1: Algorithm command-line argument values

If you wish to use a programming language other than Python, C++, C, Java, and Go, please contact us first. As before, you should submit either 1) a single source file named one of partition.py, partition.c, partition.cpp, partition.java, Partition.java, or partition.go, or 2) possibly multiple source files named whatever you like, along with a Makefile (named makefile or Makefile).

Second, 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.

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

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

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.)

Solution:
Currently, we generate a random solution for each algorithm. We could have a much better initial approximation if we ran the Karmarkar-Karp algorithm and used the result as the initial solution for the randomized algorithms. We would have to edit the KK algorithm slightly to return partitions instead of the residue. We could then use the partitions as the initial solution for the randomized algorithms.
However, this would not affect the repeated random algorithm, since it generates a random solution each time. As such, the initial solution does not really matter. However, it would affect the hill climbing and simulated annealing algorithms. They would both start at a state closer to the solution state and then reach a better solution faster (i.e. would not need as many iterations).

Finally, the following is entirely optional; you'll get no credit for it. But if you want to try something else, it's interesting to do.

Optional: Can you design a BubbleSearch-based heuristic for this problem? The Karmarkar-Karp algorithm greedily takes the top two items at each step, takes their difference, and adds that difference back into the list of numbers. A BubbleSearch variant would not necessarily take the top two items in the list, but probabilistically take two items close to the top. (For instance, you might "flip coins" until the the first heads; the number of flips (modulo the number of items) gives your first item. Then do the same, starting from where you left off, to obtain the second item. Once you're down to a small number of numbers – five to ten – you might want to switch back to the standard Karmarkar-Karp algorithm.) Unlike the original Karmarkar-Karp algorithm, you can repeat this algorithm multiple times and get different answers, much like the Repeated Random algorithm you tried for the assignment. Test your BubbleSearch algorithm against the other algorithms you have tried. How does it compare?