CS124–Spring, 2020 Programming Assignment 3 Out: April 13, 2020
Due: April 28, 2020

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 the sums of A_1 and A_2 being as close to equal as possible. 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 integers 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.

Task 1: Give a dynamic programming solution to the Number Partition problem.

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 instead of having two elements we have one element of size $|a_i - a_j|$. 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, although you may find it helpful to do so for debugging purposes.)

The Karmarkar-Karp algorithm repeatedly takes the largest two elements remaining in A at each step and uses differencing on 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. However, the best possible residue for the example is 0.

Task 2: 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.

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 $-s_j$. That is, with probability $1/2$, we just try to move one element to the other set; with probability $1/2$, we try to swap two elements. (The two elements we try to swap may be from the same set or different sets.)

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. You can think of prepartitioning as being the opposite of differencing: where differencing splits two elements apart, prepartioning is a way of forcing elements together.

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
  $\quad a'_{p_j} = a'_{p_j} + a_j$

- 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 which partition one element lies in. 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 for ranges this large!

Below are the main tasks of the assignment.

Task 3: Write a program in either Python 3, C, C++, or Java called `partition.py`, `partition.c`, `partition.cpp`, or `partition.java` (`Partition.java` also works if you prefer). The program should take in 3 command line arguments:

- a flag: we will test your program with a flag of 0. You may choose to use other values of the flag for other purposes.
- an algorithm: this argument will simply be a number that indicates what algorithm should be used to compute the residue. The following the table lists what the algorithms the codes correspond to. You may support additional algorithms if you'd like, but we will only test these 7.

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

- an inputfile: this will be the path to a file that contains a list of 100 (unsorted) integers, one per line.

Your program should print out a single number: the residue after running the requested partioning algorithm on the numbers in the inputfile. Your code will be automatically tested to verify that the residues you obtain with each algorithm are within reason, so don't print out anything extraneous when the flag is set to 0.

So, as an example, if you wrote your program in C, we would test your prepartioned hill climbing algorithm with: `./partition 0 12 numbers.txt`.

Task 4: Generate 100 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 – give both the numerical results, and the time taken by the algorithms. 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) = $10^{10}(0.8)^{\lfloor \text{iter}/300 \rfloor}$ 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.

Task 5: 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.)

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, no additional credit: Can you design a BubbleSearch based heuristic for this problem? You may want to read the BubbleSearch paper that is online at the course website, and then consider the following. 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 proabilistically 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 – or even do something exhaustive.) 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?