CS 124 Programming Assignment 3: Spring 2022

Your name(s) (up to two): Ash Ahmed, George Popoola

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: 11 (George), 10 (Ash)
No. of late days used after including this pset: 11 (George), 10 (Ash)

Homework is due Wednesday 2022-04-20 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:

At a high level, we view the goal of this problem to be finding a specific sum of a subset of all n elements such that the sum is as close to $totalsum/2$ as possible (total sum is the sum of all n elements). This can be thought of as us "grabbing" some of the elements where they sum to half of the total sum of all elements; this minimizes the residue since we know the sum of all other elements we didn't grab is $totalsum - (grab)$, so if we reach $totalsum/2$, the remaining portion would also be $totalsum/2$.

Dynamic programming comes into play for our ability to know if we can reach a certain sum; we will keep track of all possible sum values that elements up to a certain index i can attain. That is, we have a matrix with n rows and $totalsum$ columns (note that finding totalsum is constant time according to the problem description), and we fill in indices with a list of elements up to that row's index that can attain the given sum.

Our specific algorithm is as follows: initialize the matrix described above with entries being all empty lists. Now begin at the second row, which represents sums attainable if we consider up to index 1 (just considering 1 element from our input). Of course, we know we can only attain a sum of 0 (0th column) and 1 (2nd column), but to make our algorithm clear for more complex steps, we fill in this row in the following

manner: first, copy over the previous row to be the current row. Next, iterate through the previous row and for all non-empty entries, we calculate the sum of the new element we're considering and that entry. We then index (constant time) to that sum's index in the current row, and check if it is already filled in; if it is, then we do nothing, but if it isn't (empty list), we modify it to be a list containing that previous row index and our new element. We also make the index for the current new value be filled in with just the value.

The logic behind this is simple: given that we are including a new element, the potential "new sums" we can now reach are every sum we could reach before plus the new element, and the new element itself alone. Thus, we can create new rows in linear time w.r.t. all possible previous sums, which is at most b. We do this for all rows 0 through n, and by the time we reach the nth row, we have tracked how to reach different potential sums.

Our final step, once we've filled in our DP-matrix as described above, is to index to column $totalsum/2$ in the nth (final) row, and check if there is a solution. If there is, we return a residue of 0 and all numbers stored in the list at that index comprise one group, and all numbers not in the list but included in our input comprise the other group. If $totalsum/2$ was not reachable, we consider the columns adjacent to it, which both represent a residue of 2 (since each group is 1 away from perfect balance). We keep doing this, considering 2 columns away from $totalsum/2$, 3, etc. until we reach a non-empty entry. Once we've reached one, we know that is the best solution we'll see (since we're working our way outward from optimal solutions).

Correctness: While implicitly described above, all that must be proven is that in our construction of the DP array, we are correctly finding all reachable sums up to a certain index. As stated two paragraphs above, the logic is simply that if we know all sums attainable before adding in a new element, the new element PLUS those sums, and the new element on its own, comprise all possible new sums we can now get. Thus, but iterating linearly through the previous row in our DP array, we can linearly find previous solutions, and in a constant time operation add our new candidate element to a previous solution, index to that sum in the previous row, and if it wasn't attained before, it now is (so we put this newly made list into that sum's index in the current row).

Time Complexity: This algorithm is $O(n * b)$ because for each of the n rows in our $n * b$ matrix, we are doing at most b work. To elaborate, for dp[k], we are adding in the kth element from the original list A. Then what we do is iterate through each of the $O(b)$ element s in the k-1th row. With them, we are adding this kth element to each of the sums in the previous row and checking the result of this new sum. From each of these, if the sum is equal to some number p, then we are adding this arrangement of numbers to dp[k][p], which, like the initial addition, is a constant operation (we know the index, and arithmetic operations to find the sum are assumed to be constant). The final step for the nth row once we've filled in our DP array is at most $O(b)$ (we check all columns for solutions), which is subsumed by the $O(n * b)$ complexity to create the table. Therefore we have an upper bound of $O(n * b)$.

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:
The Karmarkar-Karp algorithm can be implemented in $O(n \log n)$ steps if we represent our list A as a Max Heap. Specifically, if given a list of numbers A, we first use Build-Heap on it. Then, we can simply do the constant operations of extracting the maximum element twice, subtracting them, and inserting the result of $|A_{largest} - A_{2^{nd}largest}|$ back into the heap. Then we repeat this series of operations above until the first time the second extract max is 0 (or $A_{2^{nd}largest} = 0$), at which point we know to stop and only one non-zero number remains, which is $A_{largest}$, our residue.

Correctness: We claim the correctness of this approach by assuming the correctness of the Build-Heap and Extract-Max, all of which we know to be correct from class material. Since we are just using a series of these correct functions, our algorithm should also be correct. At a step-level, we're still removing the largest and 2nd largest elements, finding their difference, and adding that back to the heap (in line with the pset description).

Running Time: The running time of Build-Heap is $O(log(n)$, and it is run once on the given list A. Then we have, Extract-Max, which has a run time of $O(log(n)$ as well. However, it is run $2(n-1)$ times (as we will have $n-1$ total $A_{largest}, A_{2^{nd}largest}$ pairs). Then the total running time is $O(2(n-1)+1(n(log(n))))$, or simply $O(nlog(n))$.

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

- 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 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++ (or if you provide a Makefile), 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.

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,

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

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.

Solution:

(Tests pass on gradescope)

Before detailing the experimental results, it serves to discuss some of the implementation details when coding these algorithms and heuristics. For the random numbers, we used a Mersenne Twister, a more sophisticated random number generator that utilized random seeding. Since we can only generate numbers in a certain range, we used it to randomly generate 0's and 1's, which served as indices into the vector -1, 1 (to randomly generate -1 or 1 to be appended to the solution index). We also used this same generator for generating our pre-partition vector (in range $[0, 99]$) and the list A itself (in range $[1, 10^{12}]$.

The 6 heuristics along with the Karmarkar-Karp algorithm that were implemented were done following the pseudo-code laid out in this document.

Writing the actual functions was fairly straightforward, with most issues being debugging. A major change we made was abstracting a lot of repetitive functions (generating a random solution, for both standard and pre-partitioning, calculating residues, etc.). At a high level, we relied on our heap implementation and C++ vectors being so easy to use without making us worry about active memory management. We decided to use doubles as the de facto datatype for values since we knew we'd need to store fairly large values; that said, we could have also gone with long longs or something else.

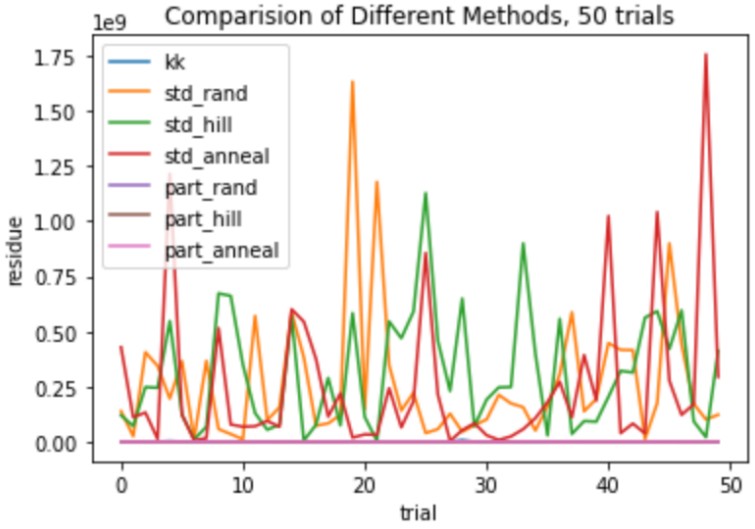
Our heap implementation was meant to provide us exactly what we needed it for (just grabbing and removing max element and inserting elements into heap); we followed the structure of a max-binary-heap, where every node has two children and it must be greater than both children. A key convention the heap uses is that the left child is at index 2 times root + 1, and the right child is at index 2 times root + 2. By having this convention, all sorts of helpful operations (like tracing down a tree with recursive heapify calls, back up the tree during our pop() operation, etc.) are easy to write. Note that the heapify function we wrote assumes that everything in the heap is sorted except for the current "triad" of root, L child and R child; it recursively "follows" the incorrect element down the tree until it is in the correct position.

	kk	std_rand	std_hill	std_anneal	part_rand	part_hill	part_anneal
Mean	437871.64	266470162.24	311119500.4	255680488.88	146.92	252.52	199.24
SD	1026495.3	303274465.7	260995542.1	356835603.9	141.9	304.2	198.4
Min	1451	11466386	3897337	3850692	1	0	1

Insertion has us add the new element to the end of the heap (easy to do with vector pushback op), and then we "follow" it up the binary tree until its bigger than both children and less than parent. Pop has similar logic, where we swap the root element with the last element (once again, this is because vectors have an easy popback function for us to grab it), and then we call heapify on the value we swapped it with (it needs to go all the way back down the heap since it was smallest before the insert).

Finally, we included some helper functions for printing vectors/heaps and making txt files of our function outputs at the bottom of heap.cc (commented out to suppress compiler warnings).

See the table for summary statistics of the 50 random instances, and the graph for residue values by trial (same data detailed in both table and graph). For some reason, latex is acting weird and putting the table above and the graph below, but both are for this question!



To analyze the data obtained from the experimental trials of the 7 techniques, we have provided both a table and a graph. The utility of these two representations is for the table to show the data that is not as apparent on the graph.
From the start, what we can see is that std-hill, on average, gives us the highest residues out of any method. This may be the case for several reasons, but it is likely that it is getting stuck at some local minimum in the total solution space that is nowhere near the best solution, simply a small dip in the total space. Hence, it is showing us one of the main drawbacks of using such heuristics to find good solutions; the good solutions are only relative.
We see that after this, both std-rand and std-0annealing perform around the same in terms of the average value of their residues. This comes as a surprise, as our initial hypothesis would have been that both hill-climbing and annealing would have outperformed the random heuristic because they look for solutions in more deliberate manners. However, in retrospect, it seems to make sense as, unlike the latter 2, it has more freedom to look for better solutions, and therefore escapes the trap that hill-climbing gets caught

in with local minima. Still, it does slightly worse than annealing, which is likely due to annealing's more exploratory nature in looking at worse solutions with some probability, helping it to escape that trap as well.

Next, we discuss another glaring observation, which is how all of the heuristics performed drastically better with the pre-partitioning representation. We see that all three of them have solutions in the hundreds in the table and are basically invisible in the graph. This seems to be because, for each of the pre-partitioned heuristics, we are running KK for each iteration, which in turn

Finally, Karmarkar-Karp showed itself to be the better than the standard and worse than the pre-partioned of all approaches. This is because we only do it for one trial instead of the 25000 repetitions, so it had less opportunity to obtain a lower average value.

For the simulated annealing algorithm, you must choose a *cooling schedule.* That is, you must choose a function T(iter). We suggest $T(\text{iter}) = 10^{10}(0.8)^{\lfloor \text{iter}/300 \rfloor}$ 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.

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:
In running the Karmarkar-Karp Algorithm, we could take its solution and use it as a starting point. However, since this algorithm returns a residue, we need to specify the solution it proposes in terms of an array of signs and as an array of partition labels. As the former, the Karmarkar-Karp Algorithm solution, as stated above, can be derived from a 2-coloring on the positions in the original A, where the edges are defined as the differencing pairs that are used throughout the algorithm. If we represent it as the latter, we would take this 2-coloring and give those one color one partition label and those with the other color another label, giving us partitions with two distinct numbers.

If we start with this solution, then we know that for any heuristic, we are most likely starting with a better solution than we otherwise would have if we started with a random solution. We expect this to be true because of the way the algorithm runs. Specifically, since we are separating the two largest numbers into the positive and negative groups, we are attempting to even out the groups as effectively as possible in an effort to minimize the residue. This shows us that there is some strategy behind this algorithm that is working towards a lower solution. Because of this, we could expect that each of the heuristics would run quicker if we start with the Karmarkar-Karp solution because we are probably starting closer to (or even at) a local minimum rather than if we start randomly. By this same reasoning, it may even increase our odds of finding the optimal solution if we start from Karmarkar-Karp. Furthermore, because we start with this approximation, we know that we can do no worse than it following all of the heuristics. These are just some of the effects of running these heuristics starting with the Karmarkar-Karp solution.

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? 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 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?