CS 124 Programming Assignment 3: Spring 2022

Your name(s) (up to two): Rei Yatsuhashi, Evan Jiang

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:
No. of late days used after including this pset: *medical extension due to concussion

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.

Consider the optimal partition of the sequence A into subsets O_1 and O_2 such that (sum of O_1) $\leq$ (sum of O_2). If the partition is optimal, then we know that because:

$$\sum_{a \in O_1} a \leq \frac{S}{2} \leq \sum_{a \in O_2} a = \left(S - \sum_{a \in O_1} a \right) \text{ where } S = \sum_{i=1}^{n} a_i$$

finding the optimal partition O_1, O_2 is equivalent to finding the subset $O \subseteq A$ with the greatest possible sum $\leq S/2$. We propose the following algorithm. Given a sequence $A = (a_1, a_2, \ldots, a_n)$ with sum b:

1. Construct arrays T of size $(\lfloor b/2 \rfloor + 1) \times n$ and S of length n initialized to 0.
2. Initialize a counter ranging between the values $0 \leq i < n$:
   (a) For $i = 0$, set $T[0, 0]$ and $T[a_1, 0]$ to 1.
   (b) For $i > 0$, set $T[0, i]$ and $T[a_{i+1}, i]$ to 1. For all values s such that $T[s, i-1] = 1$, set $T[s, i]$ to 1. If $s + a_{i+1} \leq \lfloor b/2 \rfloor$, then also set $T[s + a_{i+1}, i]$ to 1.
3. Find the largest value m such that $T[m, n-1] = 1$. Initialize $j = m$ and $k = n - 1$. While $j > 0$:

(a) If $T[j, k-1] = 1$, decrement k by 1.

(b) If $T[j, k-1] = 0$, set $S[k] = 1$. Then, decrease j by a_{k+1} ($j = j - a_{k+1}$) and decrement k by 1.

4. For indices ℓ such that $S[\ell] \neq 1$, set $S[\ell] := -1$. Return S

Complexity: The initialization of the two arrays in step (1) (T and S) have size $O(nb)$ and $O(n)$ respectively, for a space complexity $O(nb)$. Regarding time complexity, the loop in step (2) has n iterations, with each iteration having $\leq b$ operations. Finally, the loop in step (3) has at most n iterations (one per column) with constant time per iteration. Thus, our time complexity is $O(nb)$, which is pseudo-polynomial dynamic programming algorithm.

Correctness: Our proof of correctness relies on the following recursive definition. If the sums attainable using a subset of the first k elements of A are represented by a set S_k, then we set the base case $S_0 = \{0\}$ and the recursion for $1 \leq k \leq n$:

$$S_k = S_{k-1} \cup \{a_k + s \mid s \in S_{k-1}\}$$

as a non-negative integer can only be expressed as the sum of a subset of the first k elements of A if it can either be expressed as the sum of a subset of the first $k-1$ elements of A, or expressed as the sum of a_k and a subset of the first $k-1$ elements of A.

We claim that after step (2) of our algorithm, our array T satisfies the following property: $T[i, j] = 1 \iff$ there exists a subset of the first $j+1$ elements of A that sum to i. In step (2) of our algorithm, we see that the 0-column of T (corresponding to the subset of A containing (a_1) only) satisfies $T[0, 0] = 1$ and $T[0, a_1] = 1$ only, which is consistent with:

$$S_1 = S_0 \cup \{a_1 + s \mid s \in S_0\} = \{0\} \cup \{a_1\}$$

Furthermore, the for-loop for updating T as set in step (2) is consistent with the recursion for S_k. Thus, we know that $T[i, j] = 1 \iff$ there exists a subset of the first $j+1$ elements of A that sum to i, as desired. Therefore, we know that the entries with value 1 in the rightmost column of T (with index $n-1$) are the sums $\leq \lfloor b/2 \rfloor$ that can be attained using some subset of the first n elements of A, which is A itself. Thus, it suffices to recover the indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that:

$$\sum_{j=1}^{k} a_{i_j} = m$$

where m is the largest value $\leq \lfloor b/2 \rfloor$ such that m can be expressed as the sum of a subset of the elements in A. We ahve shown above that this is equivalent to the largest m such that $T[m, n-1] = 1$.

We show that our algorithm correctly identifies the requisite indices. We note that by the construction of our algorithm, if $T[i, j] = T[i, j+1] = 1$, this implies that i can be expressed as the sum of a subset of both the first j elements and first $j+1$ elements of A. Thus, a_{j+1} is not necessary in the sum used to create i. However, if $T[i, j] = 0$ and $T[i, j+1] = 1$, this implies that i can only be expressed as the sum of a subset of both the first $j+1$ elements. Thus, a_{j+1} is necessary in the sum used to create i. From this property, we can see that our method of descent as detailed in step (3) of our algorithm is correct, as it sets $S[j] := 1$ when a_{j+1} is necessarily in the sum representation of m. The corresponding residue is $b - 2m$. One can modify the algorithm above (steps 3 and 4) to return the residue without affecting its asymptotic time complexity. Thus, our algorithm is pseudo-polynomial and correct.

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.

Initialize a max-heap with the elements of A, which takes $O(n)$ time if $|A| = n$. At every iteration, take the two maximum values a_i and a_j in our heap, and replace them with $|a_i - a_j|$ and 0 which takes $O(1)$ time (given that arithmetic operations take $O(1)$ time). Note that this is equivalent to the description of the Karmarkar-Karp algorithm as written above. Then, run max-heapify which takes $O(\log n)$ time. As this must be repeated $n - 1$ times, we have that our total time complexity 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++ (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.

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.

Our experimental procedure was the following:

1. Create a random array of 100 integers using Java's ThreadLocalRandom (compatible with large numbers), chosen uniformly from the range $[1, 10^{12}]$.
2. Run all seven algorithms/heuristics and record the runtime and residue.
3. Repeat (1), (2) for 50 randomly-generated arrays.

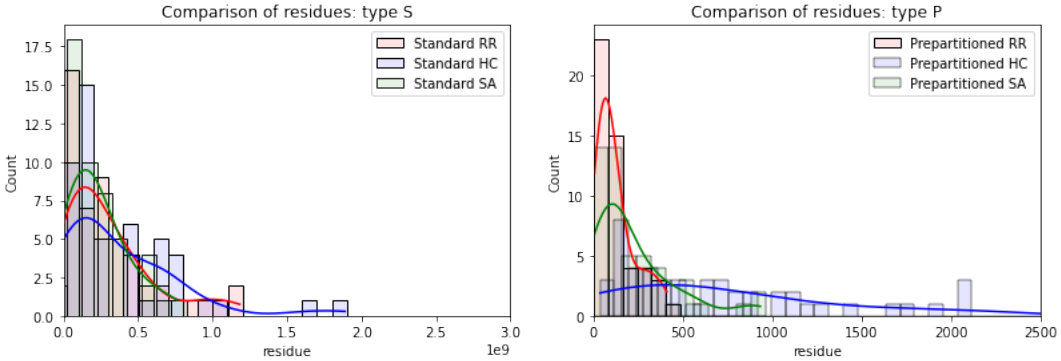
For conciseness, we refer to the algorithms tested as KK (Karmarkar-Karp), RR (Repeated Random), HC (Hill Climbing), and SA (Simulated Annealing). Furthermore, for the standard version, we use the label S, and for the prepartitioned version, we use the label P.

Comparison of Residues: Below, we have a table detailing the mean residues across 50 trials, by algorithm:

\	KK	RR	HC	SA
S	$2.808 \cdot 10^5$	$3.028 \cdot 10^8$	$3.803 \cdot 10^8$	$1.884 \cdot 10^8$
P	$2.808 \cdot 10^5$	112.2	781.8	225.4

*Note how although we did not run KK for the prepartitioned version, since KK is deterministic we can assume that its residue will be the same as the output from KK run on the standard version.

We also have a histogram for the residues across the six algorithms. As seen above, we divide the data into two groups (standard and prepartitioned) because the data differs on a order of magnitude of 10^6 across groups. Moreover, we do not include the results of Karmakar-Karp in either histogram, as its value hovers between the standard residues ($\sim 10^8$) and the prepartitioned residues ($\sim 10^2$):



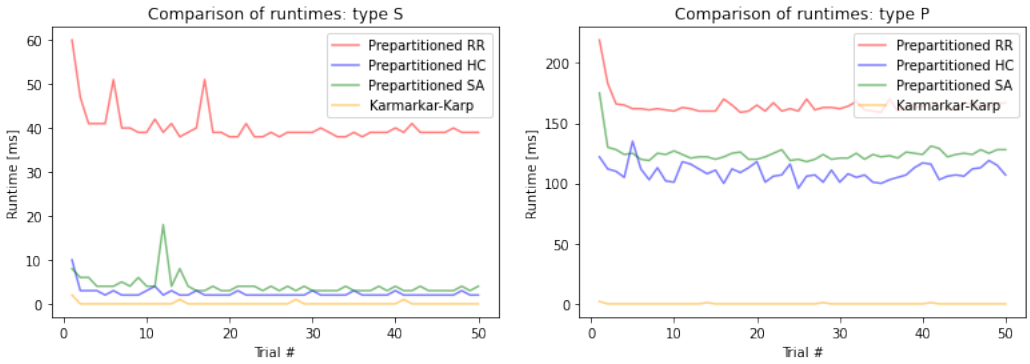
We analyze the variance across the six methods (standard vs. prepartioned), as well as the differences with Karmakar-Karp. We first examine why Karmakar-Karp performs better than the standard algorithms, but worse than the prepartitioned algorithms. An intuitive explanation of why KK outperforms the standard algorithms is as follows. Through a greedy estimation method, the KK heuristic provides a reasonable method of constructing the partition $\{s_i\}$. Conversely, without optimizations, the three standard algorithms are sampling from all 2^{100} possibilities, most of which have large residues. Consider how we expect the optimal partition to be two sets of around 50 elements, and it would be unlikely for one set to have ≥ 55 elements because the numbers are uniformly sampled from the same range. However, we know from the binomial theorem that out of 2^{100} possibilities, approximately 37 percent of partitions have at least 55 elements in one half. This observation also explains why hill-climbing performs the worst out of our three standard algorithms, as there is greater risk of the hill-climbing algorithm getting stuck in a locally-optimal solution. Because the initial plan is randomly selected in the standard variation, there is a non-negligible chance that this locally-optimal residue is far from the globally-optimal residue, and our results reflect this fact. The repeated random algorithm does not fare much better, as it is sampling over all 2^{100} partitions, most of which are largely irrelevant, as explained above. The SA algorithm is also prone to converging and navigating around locally-optimal (but not necessarily globally-optimal) neighborhoods. However, there is a higher chance of an SA chain to escape a local minimum and explore other areas, whereas this is less common in hill-climbing.

We now examine why the prepartitioned variations of the three algorithms perform magnitudes better than their standard counterparts. We conjecture that this is because the prepartitioning process requires that certain elements are grouped together (the process in which we group together certain elements can be optimized using Karmarkar-Karp), thus restricting the state space (possible partitions) to one that is smaller and on average, more optimal (KK optimizations for prepartition makes the resulting residues generally lower). Note that if just two terms are grouped together, then this halves the size of the state space. Any subsequent grouping exponentially decreases the number of possibilities, we need to evaluate, meaning that the 25000 iterations can find a locally minimal residue that is much loser to the actual global minimum, compared to the standard variant. This explains the improved residues across all 3 algorithms. Another interesting observation is that among the prepartitioned variants, the repeated random algorithm performs the best. This is likely due to the fact that it independently samples across a smaller state space, meaning that it is more likely to be able to identify small residues across a larger variety of partitions, compared to its state-dependent counterparts HC and SA.

Comparison of Runtimes: Below, we have a table detailing the mean runtime [in ms] across 50 trials, by algorithm:

\	KK	RR	HC	SA
S	< 1	40.3	2.44	4.14
P	———	164.68	109.08	124.62

We also have line graphs (x-axis represents trial number) for the runtimes across the seven algorithms. As seen above, we divide the data into two groups (standard and prepartitioned) because the data differs on a order of magnitude of 10 across groups:



KK is expected to be very fast, as it has $O(n \log n)$ runtime, and simply has to create a heap with the elements in A, then repeat the process as described in the section above (remove two elements, replace with difference, max-heapify). HC is faster than SA, as it considers a single neighbor per iteration and does one simple numerical comparison between residues. Simulated annealing is slightly slower, as each iteration requires for a probabilistic evaluation of whether to move from state S to S' or stay at S. Furthermore, both of these algorithms are faster than repeated random, as moving from a state to a neighboring state takes less time than simulating across the entire state space at each iteration.

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

All six randomized algorithms start with a random solution S. If we were to start with the solution S', the values s_i from the Karmarkar-Karp algorithm (KK), we are expected to start with a solution with significantly lower residue than a randomly sampled solution S.
We expect this to help greatly in the repeated random algorithm, as we are guaranteed to perform at least as well as KK. Because all other solutions are sampled uniformly and independently from the solution space, initializing with KK does not affect the performance of the rest of the algorithm. Though the fact that we are guaranteed to find. a residue at least as good as KK holds for both the hill climbing and simulated annealing algorithms, we cannot definitively say that it will improve the performance of our algorithm, i.e. find a sequence s_i that has lower residue than that which could be found using a random initialization. Because these two algorithms explore locally, and not uniformly over the entire solution space, by setting the KK solution as initial, the greedy method (checking whether a neighbor has lower residue than the current) is prone to converge to a locally minimal residue in a small neighborhood of KK. If we have a strict time constraint and cannot run the algorithm for many iterations, starting at a semi-optimal initial position like that returned by KK could be advantageous in converging to a local minimum relatively quickly. However, for large iteration values, this advantage becomes less relevant relative to the risk of converging on a local minimum near the KK solution. However, we note that the same risks are present in a randomized initialization as well, and the only way to fix this issue would be to combine the advantages of uniform sampling (as seen in repeated random) and the advantages of carefully exploring local minima (as seen in hill-climbing, simulated annealing), by periodically restarting the algorithm on a randomly chosen solution R, then exploring the neighborhood around R using some variation of simulated

annealing.

Regarding its effects on runtime, the initialization process takes a negligible amount of time compared to the rest of the algorithm, especially for a large number of iterations. However, we note that the benefits of a KK-initialization are more relevant when the algorithm is only run for a small number of iterations, in which case the impact on run-time is larger. This tradeoff between time and space ought to be considered when deciding between a random initialization vs. one determined by the result of a heuristic.