CS 124 Programming Assignment 3: Spring 2024

Your name(s) (up to two): Andy Martinez

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: 13
No. of late days used after including this pset: 13

Homework is due Thursday 2024-04-18 at 11:59pm ET. You are allowed up to **twelve** late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two**.

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.
Algorithm:
This problem is similar to the knapsack problem, the main difference is that the size and value are equal in this case. Since our optimal solution is one where each subset has $\frac{1}{2}$ of our total sum of the sets we can interpret this as the maximum weight for our knap sack. Thus, we start by creating a 2-D array such D[i, j] represents the maximum weight (below $\frac{1}{2}$ total sum) from A[0] to A[i] given a maximum total weight of j. Such that $i \in [-1, n)$ and $j \in [0, \frac{\text{Total sum}}{2}]$ where n = length of input sequence. The base case occurs when i = -1, this is because the only valid set is the empty set. So D[-1, j] = 0 for all j. Now, to merge our recursive cases we will have 3 total cases, one of which is the base case mentioned.

(A) If $i \geq 0$ and $j \geq s_i$: D[i, j] = max(D[i - 1, j], D[i - 1, j - $A[i]$] + $A[i]$)
(B) If $i \geq 0$ and $j < s_i$: D[i, j] = D[$i - 1, j$]
(C): If $i == -1$: D[i, j] = 0

Run this algorithm until we fully populate the 2-D array. The minimal residual would be in D[n, $\frac{\text{Total sum}}{2}$]. One alteration is that we can store tuples instead of just the values. In these tuples the first entry will

be the residual and the second entry will be the element added ($A[i]$). Doing this follows us to return not only the residual values but also the sets themselves. So we can backtrack using these last elements added, and for each element that shows up in the path backwards. All the entries along the path in the table D will go into A_1 and the ones that are excluded will go into A_2. Then we can return both the residual and the sets themselves.

Space and Time Complexity:

All the operations for each iteration takes O(1) seeing as it just involves comparisons, additions, and checking conditions. Seeing as we have to fill the entire table (D) which has n rows and $\frac{\text{Total Sum}}{2}$ columns. So the overall runtime of filling the table is $O(\frac{n}{2} \cdot \text{Total Sum})$ which just reduces to $O(n \cdot \text{Total Sum})$. In this case $n = n$ and $b = \text{Total Sum}$ giving us the $O(nb)$ from the question. This is a pseudo-polynomial because it runs in polynomial of the total sum but not in terms of the number of digits of the total sum.

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:

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

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.

To implement Karmarkar-Karp in $O(n \log n)$ time the trick is to use a max-heap. Start by taking the sequence as input and heapify it, this process takes $O(n)$ time. Let's start by assuming there are at least 2 elements in the max-heap. So in this case, we'd use heap-pop and calculate the residual of the top 2 largest elements. The heap-pop takes $O(\log n)$ for each pop, and the subtraction takes O(1). Now, if the residual is non-zero we heap-push it back onto the heap. Thus, worst case we have to execute code that runs in $O(2 \log n) = O(\log n)$ per iteration. Additionally, we can only have a maximum of $O(n)$ iterations because even in the worst case we are taking 2 elements and adding 1 back, resulting in the number of iterations being upper bounded by O(n). At the end, we have 3 possible cases. One, subtracting the last two elements results in zero in which case we just return zero. Two, the last two elements results in a non-zero number, in this case we can just return thie residual. Finally, let's say we get to the last 3 elements, and in subtracting the largest 2 we get 0, meaning only one number is left in the max-heap. In

this case we should break and just return the last element in the max-heap. All three of these final steps take O(1), so the overall runtime of the algorithm looks like $O(n) \cdot O(\log n)$ or $O(n \log n)$.

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

- 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(−(residue(S')-residue(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.
Response.

	Minimum Residual	Average Residual	Runtime
Karmarkar-Karp	3138	209220.24	3.589630126953125e-05
Repeated Random	9048590	208421007.92	1.304308352470398
Hill Climb	1290993	165470547.08	0.032649765014648447
Simulated Annealing	351829	171497045.14	0.05913392066955567
Prepartitioned Repeated Random	15	159.52	2.450779089927673
Prepartitioned Hill Climbing	0	168.72	1.3267576456069947
Prepartitioned Simulated Annealing	6	157.92	1.3540440750122071

In the graph above we can see a few general trends. For instance, the worst algorithm is Repeated Random in pretty much all respects. We see that its minimum and average residual values are the largest and is among the slowest of the algorithms. This tells us that blindly choosing solutions and getting the best out of all of them isn't a great way of solving the partitioning problem. This is because of its slow runtime and it also not being deterministic or returning consistently low residuals. Instead, we should run the KK

algorithm or run both in conjunction and take the minimum of the two. The second worst out of the set is likely the Hill Climb algorithm. Again, this algorithm relies on randomness and is not deterministic. Its runtime is relatively competitive so its speed is not holding it back. Hill climb being faster than repeated random makes sense because for each iteration repeated random must come up with a total new partition each time whereas hill climb only alters 1 slot at most. Hill climb is likely better than repeated random because it leverages randomness to guide it to a near optimal solution. However, the reason this is the second worst is likely because the algorithm is prone to getting stuck on local extremes (local mins in this case). This claim is corroborated by Simulated Annealing being overall better than hill climb. Simulated Annealing is similar to Hill Climb but combats the issue of getting stuck at local extremes by incentivizing exploration a percentage of the time, allowing it to escape local extremes and possibly find better partitions. Again, running KK along side all of these random algorithms could be good because KK is quick and provides an upper bound for the residual value.

Now, for the prepartitioned algorithms, these are the best 3 algorithms out of the ones tested, even better than the KK algorithm. This benefit comes at the expense of time complexity seeing as these are among the slowest algs. These pre-partitioned They introduce more randomness by merging certain groups together but they also leverage the fact the act of partitioning leaves a lot of empty slots. This effectively reduces the size of the input sequence and results in a less complex problem. These are all return approximately the same residual values but we again see that out of all of them the repeated random is the worst. This is because despite having similar minimum and average residuals, its runtime is the slowest, but still better than KK. Out of pre-partitioned Hill Climbing and Simulated Annealing, we see they are approximately the same accross the board, but if I had to choose one I would select prepartitioned simulated annealing. That is because it has a lower risk getting stuck at a local maximum because of the extra bit of randomness added by the extra conditions in the code. Ultimately, if you need a quick algorithm and are okay with slightly higher residual values, then KK is the best. However, if you want to approach the optimal solution, and are okay with some extra runtime, then either the prepartitioned versions of Hill Climbing or Simulated Annealing will work.

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.)

Response.

In this case, using KK as a starting point is great because it can not do any harm to our residual values. This is because for most of these algorithms we can run the KK algorithm along with the random algorithms and just choose the one that minimizes the residual. Thus, at its worst KK will only serve as an upper bound and at its best it could improve our residual values. All of this in exchange for not much downside seeing as KK is deterministic and pretty quick to run.

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?