CS 124 Programming Assignment 3: Spring 2022

Your name(s) (up to two): Yooni Park and Chloe Loughridge

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: Yooni: 11
No. of late days used after including this pset: Yooni: 12

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).

Give a dynamic programming solution to the Number Partition problem.

For the Number Partition problem, we know that we must eventually create two sets of numbers such that the difference between the total sum of both groups and its values is minimized, as we are merely finding two values, a_i and a_j, with a minimal difference and placing them into two different sets and doing this repetitively for a total set A. If we say that the sum of all values in A is b, for example, then we know that both sets we end up creating must have sums as close to $b/2$ as possible.

In order to achieve this, we must populate some array, which we title NP, as a 2D array: the first index indicates the number of elements we have visited in the array so far, while the second index indicates the total sum that we desire to achieve. In this way, we can fill in a boolean value of true or false at each $NP[i][j]$ such that the first i elements do or do not sum to j.

We know that we can use recursion to prove that our algorithm will work: our base case, $i = 0, j = 0$ is true as an empty set of elements will sum to 0. We can then move to our recursive step, which essentially states that if $NP[i-1][j-1] = True$, if the i^{th} element of set A is 1, then it follows that $NP[i][j]$ is $True$ as well. Otherwise, it will be false. We will continue iterating until we reach some set such that $NP[i][\lceil b/2 \rceil] = True$, as we want to ensure that we are as close to half of b as possible. However, if this is not true, we can also backtrack to find an $NP[i][j]$ that IS True; the first one we find is the solution to the Number Partition problem.

Our solution is correct as it merely utilizes the Number Partiton algorithm that was given and uses iteration to solve it. We find the run time using what we know about iteration: we are choosing to fill an array of size that we aim to be at most $NP[|A|][\lceil b/2 \rceil]$. We know that this will take at most $O(nb)$ steps, where $n \leq |A|$. Since finding the sum of some n elements is at most $O(n)$ and backtracking to find a $True$ portion of the NP array takes at most $O(nb)$ steps, we are able to see trivially that our Number Partition problem runs in pseudo-polynomial time.

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.

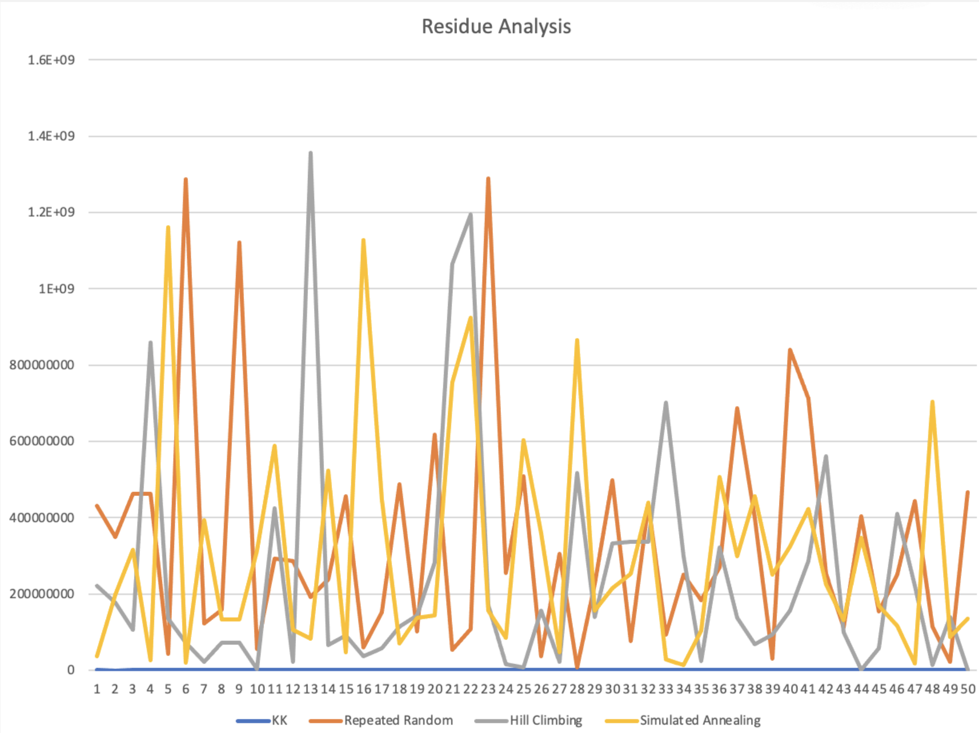
For our Karmarkar-Karp algorithm, we choose to create a MaxHeap in order to ensure that the algorithm will be implemented in $O(n \log n)$ steps. We do this trivially using the input we are given in $O(n)$ time, as the creation of a MaxHeap on a list w/ n elements is known to take $O(n)$ time.

We need to repeatedly take the largest two elements at each step, of which there are at most $O(n)$ trivially. By nature of the fact that we are utilizing a MaxHeap, we know that finding the two maximum elements of the heap takes $O(\log n)$ time. Insertion of the difference of these elements also takes $O(\log n)$ time. As such, we have ensured by using the Karmarkar-Karp algorithm on a MaxHeap that we can implement it in $O(n \log n)$ steps.

We know the correctness of our algorithm trivially; converting our input into a MaxHeap does not change the list itself, and as such finding the two maximum elements and finding the difference does not change. We are merely emulating the Karmarkar-Karp algorithm given to us in the problem statement, which is why we can be reassured that our output at the end is what we desire.

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.

We did runs on 50 random instances for KK, repeated random, hill climbing, and simulated annealing. We will currently discuss the non-prepartitioned portion of our algorithm. Here is a graph of the residues:

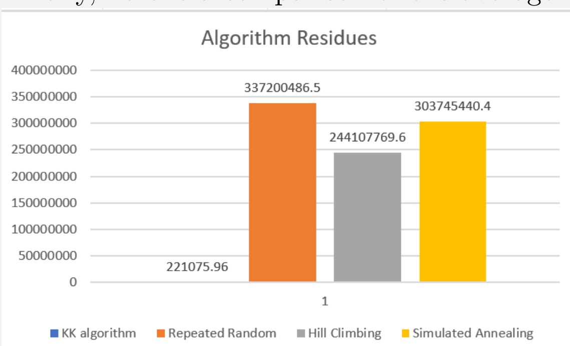


Here is a corresponding table of values for the non-prepartitioned algorithms:

KK	Repeated Random	Hill Climbing	Simulated Annealing
188927	431448759	221584357	37071550
7693	349891311	177940660	197474797
259043	463060450	105670016	314835051
214746	462831434	858032059	25234025
355441	43660178	134137212	1161178590
136770	1286056363	71268537	19999902
867570	122812971	22109819	392338644
92319	158724515	71823534	133757416
154735	1121073067	71034567	133771463
112615	55899326	2574104	312973703
77269	291479287	424166461	588694084
46008	287030119	21596927	105925191
129031	192085890	1355660930	83134809
140653	237473700	66091644	522398738
63029	456549334	90392750	46669462
47489	58846751	36091332	1127270668
24722	152286665	57080367	447878198
85627	487643441	113996077	69676317
56290	102161127	143899179	136308809
152728	617999446	280976393	143276041
25808	52708826	1064952830	754762895
843601	107349969	1195226860	924095116
302111	1289110618	169982065	156873649
423430	253670751	15745995	83712948
189225	507620193	6086809	603021229
35558	36219517	156390401	359209290

325625	304552890	21259053	47055477
217906	7472918	516314109	865083493
265521	225801031	139900840	156321585
15408	497479012	332217615	215571247
630400	76714117	335791910	252328783
389751	435810714	335694607	438375513
221219	93683457	701258457	28981042
218729	249701300	298832129	13736251
184162	182674128	24535532	103575835
613481	270026025	321346907	505367248
274305	687455879	137365834	297932081
166302	398875127	67981270	455743891
123036	29199505	93884717	250378935
245424	840502576	156022065	323821994
154594	712724117	285414542	422302491
142965	252871583	561772095	226017535
75296	112522303	99609336	130426157
339406	404579757	295590	345934034
37262	154386126	57789442	167981832
624857	249749590	410540236	115421594
65167	444338899	220011487	18546136
132650	114656585	12849297	703051246
220023	22603378	139760532	87683453
337871	465949301	428995	134091580

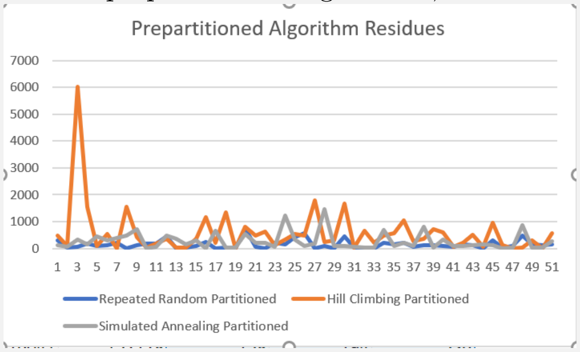
Finally, here is a comparison of the average residue produced by each non-partitioned algorithm:



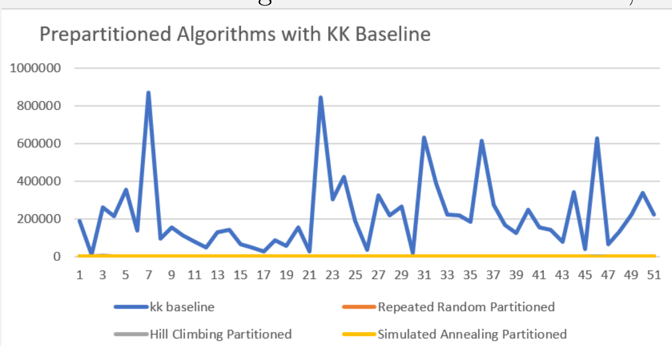
We see that KK had the lowest residues overall and performed fairly consistently, whereas all three other algorithms that we utilized had fairly varying residues. Based upon inspection of the averages, however, we see that hill climbing performed slightly better than repeated random and simulated annealing. (244107769.6 vs. 337200486.5 for repeated random and 303745440.4 for simulated annealing)

In regards to run time for the Python implementation, it is also true that KK was the fastest to run at 2.31E-03 seconds on average. This makes sense as KK takes $O(n \log n)$ steps whereas our other algorithms require a decent amount of iteration. Repeated random was also relatively efficient, averaging at a run time of 10.62544959 seconds, and so was simulated annealing at 20.9017696 seconds. Hill climbing by far took the longest for us, which we found to be a bit unusual as our implementation seemed to be far more efficient than that of something such as repeated random, which requires a new random implementation every time. However, we predict that perhaps hill climbing took longer due to the fact that we need to search for better neighbors, which takes a lot of iteration on top of the iteration we already do for it.

For the prepartitioned algorithms, we obtain the following graph of residues:



If we add the KK algorithm residue as a baseline, we obtain the following graph:



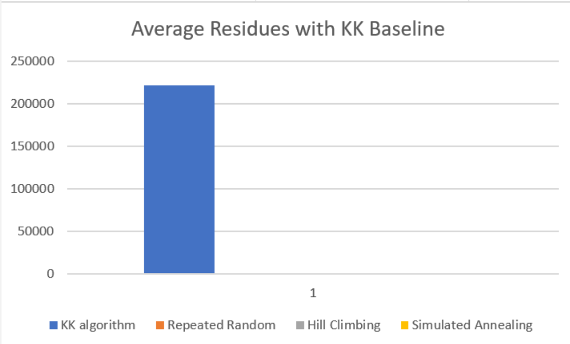
Clearly, the pre-partitioned algorithms outperform the standard KK algorithm. This makes sense when we consider that representing solutions in pre-partitioned form gives us greater flexibility over how we explore all possible groupings for the integers. Indeed, the normal form only allows us to change one integer from one side of the partition to the other per representation. By changing the groupings of the integers in pre-partitioned form, however, we can ensure that certain groups of integers will always remain on the same side of the partition together, and furthermore we can influence the side on which an entire group of integers ends up, not just a single integer (for example, if changing the group ID of one integer significantly changes the sum of the group in which it ends up, that group may end up on a different side of the partition thanks to the KK algorithm's role in pre-partitioning). This greater flexibility in expression means that we can express regions of the state space that are more optimal but unreachable by the standard representation (or at least "unreachable" in a reasonable number of attempted states without pre-partitioning).

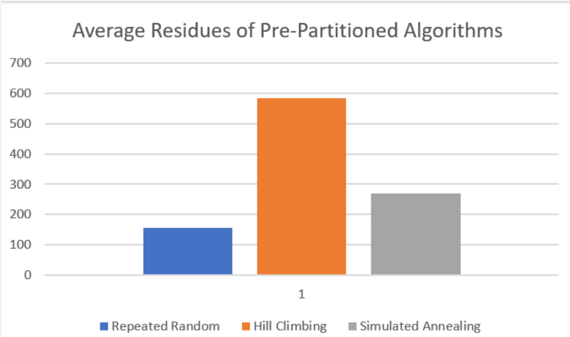
When we compare the pre-partitioned algorithms to themselves (excluding the KK baseline), the most noteworthy qualitative observation is that the hill climbing algorithm has higher variance in its returned residue values. This makes sense since the hill climbing algorithm is prone to falling into local optima instead of global optima. Similarly, simulated annealing appears to have slightly higher variance than the repeated random strategy; perhaps again this is because repeated random does not depend on finding direct neighbors of a given solution and so it is not so restrained to moving within a local area and hence it is more immune to falling into local optima.

Clearly, the most striking take-away is that the pre-partitioned algorithms do orders of magnitude better than the standard algorithms. As discussed above, this is most likely because the pre-partitioned "language" allows for more flexibility in expressing potentially optimal solutions.

For completeness, we include a table of the residue value returned by each pre-partitioned algorithm for each trial below, and we also show a bar chart of the average residue value for each pre-partitioned algorithm.

kk baseline	Repeated Random Partitioned	Hill Climbing Partitioned	Simulated Annealing Partitioned
188927	299	489	134
7693	42	80	74
259043	70	6012	327
214746	191	1557	154
355441	87	95	461
136770	121	544	296
867570	225	6	377
92319	0	1540	472
154735	117	429	708
112615	195	37	17
77269	171	183	48
46008	464	369	472
129031	12	40	368
140653	46	45	159
63029	94	355	309
47489	246	1180	5
24722	4	218	660
85627	40	1347	32
56290	14	14	40
152728	758	794	536
25808	70	485	214
843601	10	636	215
302111	204	126	59
423430	159	326	1222
189225	433	530	329
35558	573	494	83
325625	13	1791	196
217906	98	249	1462
265521	10	287	78
15408	464	1662	89
630400	30	35	49
389751	32	654	43
221219	17	211	34
218729	226	467	674
184162	142	563	93
613481	202	1043	225
274305	49	279	80
166302	112	362	810
123036	108	722	33
245424	96	599	322
154594	76	74	103
142965	159	218	78
75296	117	519	129
339406	12	60	150
37262	294	960	111
624857	20	111	2
65167	107	3	26
132650	469	40	877
220023	158	291	26
337871	124	3	32
221075.96	155.6	582.68	269.86





Briefly, we conclude with an observation on runtimes: pre-partitioned algorithms take significantly longer to run than their normal-form counterparts to get through 25000 iterations since the KK algorithm must be run to compute the residue of each prepartitioned state. As proved previously, this adds an extra runtime factor of at least $O(nlogn)$ to the prepartitioned algorithms, depending on how the KK algorithm is implemented.

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.)

We see that there are several ways in which starting with the KK algorithm may help us find a better solution, especially since KK on average we have fairly strong confidence can find lower residues.

For something such as repeated random, for example, we need a starting point to improve on using random solutions. Using KK, which we know to have a fairly low residue average as opposed to something such as repeated random, we can create a tight upper bound that may help us find a better solution. In the case of hill climbing, we would similarly be able to use our solution from KK to have an initial "best solution" that we would only be able to improve by moves to better neighbors. In the case of simulated annealing, however, it is unclear whether or not it is necessarily the case that we would find a better solution; simulated annealing functions in such a way that we do not always move to better neighbors, implying that the algorithm could jump to a worse solution instead.