CS 124 Programming Assignment 3

Harvard ID: 51170407, 81238782

April 21, 2017

1 DP Solution to Number Partition Problem

Define an $n \times (2b+1)$ matrix $split[][]$ and an $n \times (2b+1)$ matrix $set[][]$, where

$$b = \sum_{i=1}^{n} a_i$$

Each element $split[i][j]$ will keep track of the minimum achievable error for partitioning the first i numbers into sets A_1 and A_2 such that the difference between the sums of the elements in A_2 and A_1 is j (for the sake of convenience in this proof, we assume our j values run from $-b$ to b, rather than 0 to $2b$). $set[i][j]$ will keep track of whether the element a_i is in subset A_1 or A_2 Initialize all values $split[1][j]$ to equal $\min(|a_1 - j|, |a_1 + j|)$. Iterate through rows 2 through n and columns $-b$ through b and set each value in row i, column j to

$$split[i][j] = \min(split[i-1][j+a_i], split[i-1][j-a_i])$$

if $-b \leq j + a_i \leq b$ and $-b \leq j - a_i \leq b$, and set $set[i][j]$ equal to 1 if $split[i-1][j+a_i]$ is the minimum and 2 is $split[i-1][j-a_i]$ is the minimum. If $j + a_i$ or $j - a_i$ fall outside the range $-b$ to b, simply set $split[i][j] = \infty$ (or some large number). The value $split[n][0]$ is our desired error to the Number Partition problem, and the actual partitions into sets A_1 and A_2 can be found in the following manner: initialize some difference d to 0. For an index k ranging from n to 0, place a_k into set A_1 if $set[k][d] = 1$ and set A_2 if $set[k][d] = 2$. Then repeat for the next index $k \leftarrow k - 1$, and $d \leftarrow d + a_k$ if we placed a_k into set A_1 and $d \leftarrow d - a_k$ if we placed a_k into set A_2. Once we have reached $k = 0$, we will have placed all elements into one of the two sets.

This dynamic programming algorithm runs in time $O(nb)$ since each element in our two matrices $split[][]$ and $set[][]$ can be filled in constant time, so our entire algorithm takes $O(nb)$ time to run. Our algorithm also takes $O(nb)$ space due to our two matrices.

Proof of correctness: As stated above, this algorithm works by storing, in $split[i][j]$, the minimum possible error for partitioning the first i numbers into sets A_1 and A_2 such that the difference between the sums of the elements in A_2 and A_1 is as close to j as possible. We can prove this by induction on i then j. Clearly $split[1][j]$ stores the minimum possible error for partitioning a_1 into sets A_1 and A_2 such that the difference between the sets is as close to j as possible; namely, we place a_1

into set A_1 if a_1 and j have different signs, and we place a_1 into set A_2 if a_1 and j have the same sign. Then for every element $split[i][j]$, our recursion

$$split[i][j] = \min(split[i-1][j+a_i], split[i-1][j-a_i])$$

assigns the minimum possible error to our value $split[i][j]$, since all a_i are nonnegative integers, and therefore $split[i-1][j+a_i]$ is the error for achieving difference j if a_i is placed in set A_1, and $split[i-1][j-a_i]$ is the error for achieving difference j if a_i is placed in set A_2. For the values of j that go out of bounds (outside of the range $-b$ to b), we can essentially ignore those values since our desired value $split[n][0]$ only references the values $split[i][j]$ where $-b \leq j \leq b$, since the total sum of all elements a_j (which we know to be nonnegative integers) is b and thus our error will always be at most b. Finally, the actual partition can be found by backtracking through our values $split[i][j]$ starting from $spit[n][0]$ to determine the elements that fall into sets A_1 or A_2.

2 Implementation of Karmarkar-Karp in $O(n \log n)$ steps

Karmarkar-Karp can be implemented in $O(n \log n)$ steps by using a heap. First put all elements $a_1, a_2, \ldots, a_n$ onto a heap H. This can be done in $O(n)$ time. Then, to take the difference of the two maximum elements, we simply pop two elements off heap H, find their difference d, and push the d back onto the heap. This takes $O(\log n)$ times since the time to *insert* into and *pop* from a heap is $O(\log n)$. We do this n times until only one value remains, for a total time of $O(n \log n)$, as desired.

3 Description of Programs

Our implementation of the various heuristics was very straightforward. We started by implementing our own heap class using the notes from section. Once we verified the correctness of the heap, we simply converted the provided pseudocode into a correct Java implementation.

When implementing the heuristics, we made a few small optimizations. First, since we weren't actually tasked with calculating the partition, only the residue, we could save on space by not storing the generated random or neighboring solutions. Instead, we saved only the residue that we calculated. Additionally, we took care to calculate the residue only once per iteration. Initially, we were naively calculating the residue of the solutions many times per iteration. During the prepartitioning steps, this took especially long. When we only calculated the residue once per iteration, we experienced a drastic speed up in our average timing, particularly for the prepartitioning strategies.

4 Results

We summarize the results of applying each algorithm to an input of 100 randomly generated integers in the range $[1, 10^{12}]$ in the following table, where the average error and time was computed by averaging 100 random instances of the problem for $25,000$ iterations. Our full data is included at the end of the report.

Strategy	Average residue	Average time (ms)
Karmarkar-Karp	221582	0.091769
Repeated Random	241831173	50.813113
Hill Climbing	367598256	17.079770
Simulated Annealing	287477721	24.206134
Repeated Random with Preparititioning	163	722.322031
Hill Climbing with Prepartitioning	732	600.094982
Simulated Annealing with Prepartitioning	200	614.660164

The tables show that the average error for our Simulated Annealing, Repeated Random, and Hill Climbing algorithms without prepartioning is substantially higher than the error for Karmarkar-Karp or their respective prepartitioned versions. In contrast, the prepartitioned versions of our algorithms all perform much more slowly than our standard algorithms.

Furthermore, we notice that the Karmarkar-Karp algorithm performs less optimally than the prepartitioned versions of our random algorithms. Amongst the random algorithms, Hill Climbing with prepartitioning is less optimal than Repeated Random or Simulated Annealing.

As for run time, we notice that the Karmarkar-Karp algorithm is substantially faster than the others, with Hill Climbing performing second best and the Simulated Annealing and Repeated Random algorithms following afterwards.

5 Discussion of Results

The above results fall in line with our expectations for the various algorithms. The running time for Karmarkar-Karp is substantially faster than the other strategies since we are only running the algorithm once over the 100 integers, whereas in the other algorithms, we are attempting 25,000 iterations over the 100 integers.

It also makes sense that Karmarkar-Karp performs better than the purely random algorithms but worse than the prepartitioned versions of the algorithms. The purely random algorithms are relatively naive and arrive at a solution by starting at a random solution and then progressively getting better. Karmarkar-Karp, in contrast, employs a heuristic and splits apart the largest values, which turns out to be a fairly good guess. On the other hand, the prepartitioned versions of the algorithm *use* Karmarkar-Karp as a subroutine to calculate the residue of various partitions. Thus, these prepartitioned algorithms get the benefit of both the Karmarkar-Karp heuristic and the robustness of randomness, resulting in some very impressive results.

The fact that Karmarkar-Karp is a subroutine of the prepartitioned algorithms also help explain why the prepartioned algorithms are about 10 to 30 times slower than the purely randomized algorithms. In the random algorithms, calculating the residue is a linear time operation. In contrast, calculating the residue for the prepartitioned algorithms requires the use of Karmarkar-Karp, which is an $O(n \log n)$ time algorithm. This difference builds up over the course of 25,000 iterations and has a substantial effect on the average time.

Now, we can discuss the differences in performance between repeated random, hill climbing, and simulated annealing, regardless of prepartitioning. We see that repeated random actually performs best, simulated annealing second-best, and hill climbing coming in last. Again, this aligns with our intuitions. Repeated random has many opportunities to simply "get lucky" and stumble across a very good solution. Hill climbing, on the other hand, can get stranded at local optimums since it is greedy and chooses whichever neighbor improves the residue. Thus, it may never find a neighbor which is closer to the optimum. Simulated annealing attempts to strike the balance between the two. In the first few iterations, when it is very "hot," simulated annealing allows itself to take suboptimal local steps, therefore avoiding the traps of local optimums. However, the temperature decreases very rapidly and once things cool off, simulated annealing essentially resembles hill climbing. Thus, simulated annealing can avoid most local optimums, but it still may inevitably get stuck, which is why it still does not perform as well as repeated random.

The time differences are also explainable. Repeated random always took the longest time, hill climbing always took the shortest time, and simulated annealing took just slightly more time than hill climbing. Repeated random takes the longest time because at every iteration, we have to generate a brand new random solution. This means we have to generate 100 random numbers! In contrast, for both simulated annealing and hill climbing, we only need to generate a neighbor, which means that we really only have to generate, on average, two new random numbers. This saves a lot of time. The difference between the time for hill climbing and simulated annealing can probably be attributed to simulated annealing's calculation of the temperature. This calculation is somewhat costly and actually using the probability requires the generation of a new random number. This small cost causes the small time increase necessary for simulated annealing.

6 Optimizations

6.1 Changing the Method of Initialization

One way to use the solution to the Karmarkar-Karp algorithm as a starting point for randomized algorithms (in particular, Hill Climbing or Simulated Annealing) would be to initialize our sequence S to the output of the Karmarkar-Karp algorithm, rather than generating a completely random sequence.

This might be advantageous since it would reduce the amount of time it would take to reach the optimal solution and it would also start the algorithms in a better "neighborhood" where they have a better chance of finding optimal solutions. This could could also be suboptimal since it might cause our algorithm to find the local minimum of the area nearby the Karmarkar-Karp solution, rather than the true global minimum that would be desired.

Of course, starting the repeated random algorithm from the output of the Karmarkar-Karp algorithm would be a mostly pointless endeavor since the solutions generated between iterations are completely independent of one another. Hill climbing would most likely benefit from starting from the Karmarkar-Karp output, but it would be susceptible to getting stuck at local optimums. Simulated annealing might be perfectly suited for starting at the Karmarkar-Karp output since in the first few iterations, it might accept taking locally suboptimal steps and thus avoid local optimums. It has the flexibility to "escape" from the local minimum in the first few iterations before degenerating to hill climbing in later iterations.

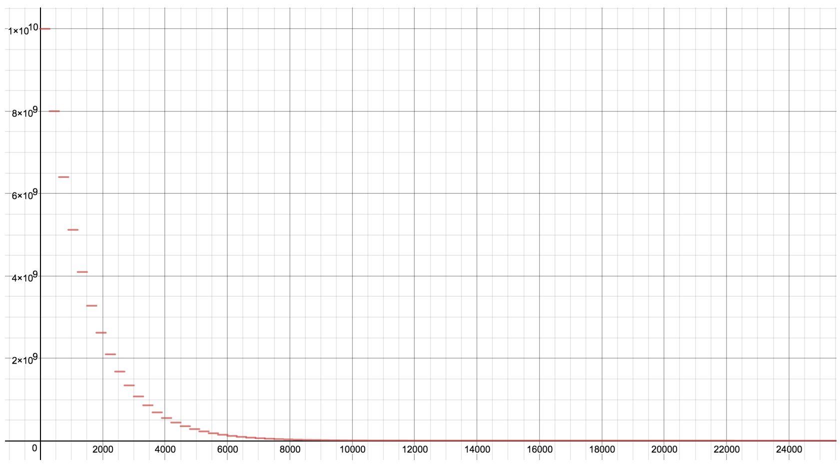
We conclude that it is likely that starting from Karmarkar-Karp's output would most likely benefit the hill climbing and simulated annealing algorithms, although we should be wary of getting trapped in local optimums. This issue is partially addressed by simulated annealing's high temperature.

6.2 Different Temperature Curve

If we plot the given temperature curve,

$$T(\text{iter}) = 10^{10}(0.8)^{\lfloor \frac{\text{iter}}{300} \rfloor}$$

we get the following graph

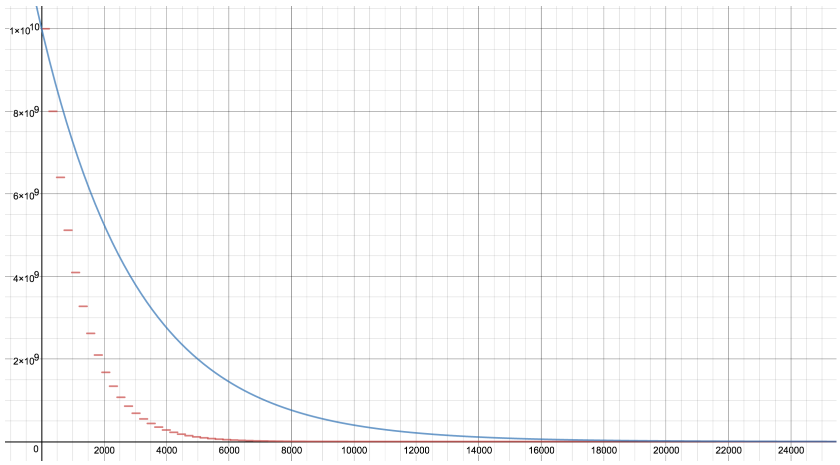


We immediately notice two things. First, the temperature moves in steps, which is to be expected, given the presence of the floor function. Second, the temperature cools off extremely quickly. After iteration 10,000, it is getting very close to 0. After the temperature has cooled, simulated annealing degenerates into hill climbing since it will no longer accept suboptimal local solutions. To attempt to combat this problem, we wanted to create a temperature that was smooth and that cooled more slowly.

The result of these efforts is the new curve

$$T(\text{iter}) = 10^{10}e^{x/3106.674673}$$

which we overlay in blue below.



We arrived at the strange decimal by calculating

$$\frac{5000}{\ln 5} \approx 3106.674673$$

which allows the temperature to drop by exactly one-fifth every 5,000 iterations. We can see that this curve is now smoothed out (since we dropped the floor function) and it decays slower than previously. When using this temperature function in our simulated annealing algorithms, we get the following table

Strategy	Average residue	Average time (ms)
Karmarkar-Karp	227074	0.080084
Repeated Random	268774446	43.786513
Hill Climbing	320302850	5.605623
Simulated Annealing	126054397	8.256100
Repeated Random with Prepartitioning	184	688.159942
Hill Climbing with Prepartitioning	664	526.576007
Simulated Annealing with Prepartitioning	213	560.061376

The relative timings have remained exactly the same, which is to be expected. Strangely, we see that the new temperature curve has not done much to improve the simulated annealing with prepartitioning with respect to the repeated random with prepartitioning. It seems that the Karmarkar-Karp heuristic has a larger effect on the performance than the selection of random neighbors. However, in the regular simulated annealing, we see a drastically different story. Here, the simulated annealing has an average residue which is twice as good as the repeated random algorithm. This is a complete reversal of the situation with the previous temperature curve where the simulated annealing yielded an average residue that was slightly greater than repeated random.

Overall, we conclude that the slower and smoothed cooling somewhat helps the simulated annealing algorithm avoid local minimum during its run. This is especially noticeable in the cases without prepartitioning where the simulated annealing algorithm was able to halve the residue of the repeated random algorithm. In the prepartitioning cases, the effect was too minute to overcome the power of the Karmarkar-Karp heuristic, so we observed little to no effect.

6.3 Bubble Search

Just for fun, we also decided to implement the bubble search variant of Karmarkar-Karp as described in the assignment. We ran the algorithm for 100 trials. For each trial, we attempted bubble search 100 times on the same array and took the best result. We have appended the results to the end of our previous table. Note that the timings are per trial, not per single run of bubble search.

Strategy	Average residue	Average time (ms)
Karmarkar-Karp	221582	0.091769
Repeated Random	241831173	50.813113
Hill Climbing	367598256	17.079770
Simulated Annealing	287477721	24.206134
Repeated Random with Preparititioning	163	722.322031
Hill Climbing with Prepartitioning	732	600.094982
Simulated Annealing with Prepartitioning	200	614.660164
Bubble Search	**20371**	**17.399578**

These are some impressive results! With this added randomness into the Karmarkar-Karp algorithm, we have achieve a ten-fold improvement in the average residue. This comes at the cost of time; the bubble search algorithm is about 100 times slower than the traditional Karmarkar-Karp algorithm. Of course, this makes perfect sense since in one trial of bubble search, we run a modified version of Karmarkar-Karp 100 times!

The bubble search algorithm performs substantially better than the random algorithms without prepartitioning and they operate at the same speeds. Bubble search is much worse than the prepartitioning algorithms, but those algorithms are much slower.

Overall, this optimization is effective if we don't need the absolute best speed nor the absolute best residue. The bubble search finds a fantastic happy balance between the two. It is clear that you would never want to choose the random algorithms without prepartitioning over this.

7 Conclusion

During this programming assignment, we observed some pretty interesting results! Despite the number partitioning problem being an NP-complete problem, we were able to derive some very good approximations for the solutions that run in a reasonable amount of time.

We see that the Karmarkar-Karp heuristic is an incredibly fast approximation algorithm that gives us very reasonable results. The standard randomized algorithms took longer to calculate a residue, and they did not perform as well as the pure Karmarkar-Karp heuristic. However, when we augmented the randomized algorithms with prepartitioning and Karmarkar-Karp, they far outperformed the pure Karmarkar-Karp algorithm. This performance comes with the trade-off of time; the prepartioning algorithms were over ten times slower.

By experimenting with some small optimizations - changing the temperature curve for simulated annealing, using the output of Karmarkar-Karp as the start of hill climbing or simulated annealing, and implementing bubble search - we were able to get various improvements on performance.

Changing the temperature curve vastly improved simulated annealing's performance in the standard algorithms but had little improvement on the prepartitioning version. Bubble search was particularly impressive since it was very fast and offered results better than pure Karmarkar-Karp.

In the end, all of these algorithms are good and each one fits a particular use case. Some are fast yet inaccurate. Others are slow yet give very good approximations. Some find that sweet spot in between. Whatever the task, you can find the right tools for the job.

8 Residue Data (Standard Temperature Curve)

KK	RR	HC	SA	RR with PP	HC with PP	SA with PP
219917	513056635	448726341	95007777	45	55	99
5145	165925481	200302483	640304349	321	293	497
37098	625391866	16465944	110982988	264	926	904
50548	437788604	344150364	270352160	90	92	234
1715089	165959405	250899483	134906657	197	723	59
21643	179900977	105236431	166718999	3	1153	101
28233	324588253	273413483	89460333	149	1115	7
308466	143612462	344231888	206837854	48	2414	134
6415	245032019	1073052695	185952685	293	355	927
727589	98936049	213884675	42956363	89	253	457
570618	483802660	454390540	252273084	14	40	302
33369	202196483	950845993	55138435	423	1529	113
129754	125920338	462763360	313963130	214	870	304
122688	49260032	110374774	207021956	80	612	18
54481	106543123	1375518713	75330997	5	3273	1059
73247	138907475	722624071	35598899	97	81	349
98395	20681473	495957805	62549527	59	485	203
196018	480998314	16557118	116161926	174	278	118
91240	435455110	283462080	103915278	248	384	158
823	344833285	130773157	10107507	13	15	245
165917	36004959	717040435	666440159	31	1877	9
9239	269202069	406872553	206740735	257	1087	323
41019	80957781	79697005	65721783	235	75	313
176251	92660247	332708187	376719125	25	711	245
44879	49944943	770264033	21513931	57	93	299
120999	660333017	422510495	88064191	111	341	171
1170533	208060287	543808537	615417713	89	181	5
2541083	69728215	429177011	102297531	49	1795	243
27061	840041981	473933121	548817531	7	333	289
98037	8479217	430597207	51515909	149	543	125
115127	484752437	108775035	373268529	101	1015	355
4350	125711160	389404284	1038491340	112	1374	590
961017	7820267	262434271	189001933	21	353	369
32837	264792827	502525119	493935771	115	1667	935
2194	165258378	177923900	912056486	250	3640	642
81099	4887593	678147541	432159925	37	851	37
26396	630117842	682758478	87768078	374	1612	66
431821	10665589	80076645	8549653	665	1713	361
32463	788736729	48661005	73512299	169	1737	97
37309	184175903	37290325	65690403	33	131	61
250443	537012117	684380049	96983649	391	1671	61
905162	290840230	251179182	6622936	62	372	0

KK	RR	HC	SA	RR with PP	HC with PP	SA with PP
934653	958176421	560136419	139327727	241	75	9
9824	56634210	645164780	65118116	224	232	228
53738	63223676	265901672	146993252	144	1288	34
85337	320056481	29882413	643751585	73	919	111
183442	48299230	300717116	163877618	160	14	110
21497	177226133	34473671	384543395	81	1403	51
107585	85004955	99540945	473890835	185	2265	109
83660	155051616	433679762	296299280	206	218	122
3505574	1501233066	201653044	401462262	118	104	4
101785	120646169	462533907	260353331	137	171	183
4156922	76850818	481618810	120989570	196	380	224
242570	674336848	58350054	130611960	150	956	1288
213649	631956329	44767591	36016257	33	913	105
161958	37585716	185536736	81678090	110	992	140
135673	333155351	695526277	869580155	425	793	143
68266	287278432	21707218	2037670	54	80	20
93040	448312052	291004128	473482932	230	156	20
1020161	37199221	216111099	344197489	289	11	69
60935	521573485	183208607	199526277	37	205	81
132841	211403419	404103895	598260153	103	1169	39
24569	172373007	243604359	177396763	61	289	247
114963	153484547	38592403	27695295	121	1105	219
15638	463373556	87421354	66524950	270	288	440
47120	88788316	298033650	141821906	18	860	170
208486	295488318	134385928	465002406	134	448	40
34583	691726903	161333191	392688081	47	511	1
155049	191363593	312191981	242981825	473	2839	75
564550	126347630	170978	253405558	8	644	202
2665013	25744367	283380959	470987201	199	655	157
198390	456208620	47011714	13495454	82	142	92
11490	303814608	167473738	150882006	10	394	136
17338	1272008098	211601522	729228182	70	862	172
214352	320613268	327295480	361170526	192	8	12
196312	419354498	707956208	50500232	96	314	142
48260	102087620	109464606	11476666	620	942	174
21220	434881600	99091572	181652220	512	2164	260
22314	177663278	403486750	573835540	12	24	62
190390	91050034	232767478	547614382	18	12	116
39332	47764546	328247634	164164366	128	220	188
305952	1047563326	583203644	131669824	702	1076	162
65177	36367721	610214911	526483051	55	1515	337
693486	18377402	87105428	610091630	16	166	16
87675	860034401	9647751	252300013	61	1361	9
79634	1389048	172537972	102592128	60	148	400
204334	548289884	76122664	400248402	4	2858	122

KK	RR	HC	SA	RR with PP	HC with PP	SA with PP
144876	9071152	114402790	421850468	318	266	506
469754	2998636	169845452	130258354	152	422	32
41357	3969823	83708431	140000085	347	389	545
230826	1086208842	357762340	319010304	372	488	126
858878	779465678	51640492	494729498	572	436	310
138848	325291280	165791044	55293886	144	302	74
185322	128187606	157550662	117204146	32	454	520
47976	141275762	62524578	915586710	62	454	336
12455	184634293	248734517	400068825	147	1439	197
437392	824043816	396271204	1148624312	60	204	126
32813	230390649	78487525	33816871	113	165	135
321755	538965563	225149615	124285957	103	137	81
49330	77418214	218482448	114160330	40	2460	70