CS 124
Programming Assignment 3

Curren Iyer & Paul Lisker

April 25, 2016

1 Dynamic Programming Solution for Number-Partition

Let A be a sequence of n integers that sums up to some number b. Let us construct an n by b matrix M. Here $M(i, j)$ will be used to determine whether a subset of i terms from the sequence A $(s_1, ...s_i)$ sums up to j, in which case the value of $M(i, j)$ will be 1, and 0 otherwise. We can summarize this property with the following recurrence equation:

$$M(i, j) = \max(M(i-1, j), M(i-1, j-s_i))$$

Once the table has been filled in, we can then limit our scope to the last row, $M(n, b_s)$, which denotes whether it is possible to reach some sum b_s using at most all of the n terms in the sequence. We then check to see if $M(n, \frac{b}{2}) = 1$ (this is the case where it is possible to create a sequence summing to $\frac{b}{2}$, in which case there is 0 residue since there would exist two equally summing partitions). If this is true, then we return the residue (0). Otherwise, iterate backwards from $\frac{b}{2}$ for all $x \leq \frac{b}{2}$, stopping at the first value of x such that $M(n, x) = 1$. Finally, return the residue, which is equal to $x - (b - x) = 2x - b$.

In terms of running time, creating the array takes time $O(nb)$, finding the value x takes at most time $O(b)$, and backtracking to find the elements in each subset takes time $O(n)$. Thus the total algorithmic running time is $O(nb)$.

2 Implementing Karmarkar-Karp Algorithm in $O(n \log n)$ Steps

Before we describe the solution let us make the assumption that the values in sequence A are small enough that arithmetic operations take one step. We can implement KK in $O(n \log n)$ steps by transforming the input array containing the sequence A into a max-heap, an algorithm that has a running time of $O(n \log n)$. We then go through the differencing process as described in the problem spec, with the only difference being that the two largest elements a_i and a_j are popped off the heap ($O(2\log n)$), and their difference $|a_i - a_j|$ is inserted back into the heap $O(\log n)$. Thus at most this process is run $\frac{n}{2}$ times, making the total algorithmic running time on the order of $O(n \log n)$ overall.

3 Discussion of Results

As can be seen in the tables found at the end of this section, there is a significant difference in both residues calculated and running time of the various algorithms. Overall, however, two general conclusions can be derived:

1. Pre-Partitioning consistently returns *significantly* better residues than Random
2. Random is consistently faster in running time than Pre-Partitioning

These results should not be surprising. I'll analyze both separately.

(1) Each of the random algorithms proceeds by randomly choosing a solution (denoted by an array of $+1$ and -1) and determines the residue by adding/subtracting values according to the solution. Then, via some process specific to the algorithm, it determines another solutions, checks to see whether the residue is smaller, and if so, it then (with some probability) switches. Not surprisingly, this entirely random approach provides a worse solution than the K-K algorithm, which proceeds via a deterministic heuristic, thus resulting in consistently better results. Since the pre-partitioning algorithms determine a pre-partition first and then a residue with K-K, switching to other pre-partitions (with some probability) only if the K-K residue is smaller, it is not surprising that the overall residues by these pre-partition algorithms are better.

(2) Following the above explanation, it is easy to see why each of the pre-partitioning algorithms, though resulting in better residues, also take longer; in each iteration, they calculate residues via the K-K algorithm, which in our implementation takes $O(n^2)$, though it can be implemented in $O(n \log n)$. Thus, the overall runtime of these algorithms are $O(n^2)$, whereas the random algorithms are simply $O(n)$, on the presumption that arithmetic operations take $O(1)$.

Comparing each of the algorithms, we can see that repeated random provided a better solution, on average, than the average of simulated annealing, which was better than the average of hill climbing. This is ultimately because simulated annealing and hill climbing are entirely dependent on the initial randomized guess; if they guess well, then the residue can be extremely low (with simulated annealing we got 1 once); on the other hand, a poor initial randomized guess will result in a poor residue since each better guess must be a neighbor of the original guess. On the other hand, repeated random is unencumbered by the initial guess, thus allowing it the flexibility of reaching much better solutions.

Table 1: Residues of Algorithms

	KK	Repeated Random		Hill Climbing		Simulated Annealing	
	–	**Random Move**	**Pre-Partition**	**Random Move**	**Pre-Partition**	**Random Move**	**Pre-Partition**
1	92468	126767794	340	192763984	696	40894316	364
2	67336	576168954	168	51437702	730	41436818	136
3	386590	51334430	26	257323932	512	132055062	186
4	44376	39937528	292	111233184	408	482838262	42
5	43404	14767428	24	106050972	232	146761806	224
6	4773	29101283	1	1053263161	483	238159113	1979
7	685323	591427349	183	187749373	421	67587629	55
8	159703	3169689	245	271782181	53	38979105	601
9	118384	720881738	8	460355474	682	385550650	158
10	125768	344659834	44	321743746	688	16517560	332
11	91814	152180948	64	544223036	574	251018090	134
12	75352	315924360	30	740878158	416	981606876	206
13	584562	104704806	400	2084864	526	15529826	46
14	282627	301518893	161	68643519	409	517428657	71
15	93787	226703193	291	141474963	111	5485521	217
16	9353	74772903	335	150849079	1423	419158697	31
17	234769	454171193	221	225596129	1003	31068953	31
18	107000	62900010	350	58078076	1158	636062166	358
19	643448	142758952	8	122650148	558	72666022	460
20	167887	21939941	81	130416247	519	789114943	301
21	18833	496652521	221	214623647	453	165260553	189
22	22047	40347767	121	1273399013	49	499193463	347
23	8076	580036046	114	9671768	208	666986900	108
24	48193	11050617	167	465907959	689	28562841	125
25	155324	71648884	120	403402374	824	29221524	564
26	40600	638825052	36	4615402	268	1169465050	70
27	34801	116381983	39	36239955	543	67487995	117
28	37767	443882223	309	28883573	3263	102738995	183
29	467420	445971314	50	156378880	98	70279766	164
30	736825	319585073	129	85443211	511	74260525	59
31	117929	1185309299	331	901367949	35	522442917	419
32	556744	113609978	222	663760080	1028	178587600	250
33	82811	173870329	177	642811099	993	336841041	273
34	19491	75034761	103	768327447	1115	126868735	1
35	77468	216556444	316	75716328	426	160576888	22
36	1100	106433984	82	202636430	82	380721464	252
37	126661	37386413	217	511737345	235	53804937	455
38	213455	571636187	279	439425125	135	138682931	21
39	1214	686821004	124	94534658	70	490599160	40
40	784405	52548873	427	70242679	1021	29156967	1055
41	61858	355692892	238	15325588	288	111801460	6
42	730473	392055243	285	187834339	113	289063445	81
43	229368	629303700	178	96861678	1506	1021582262	38
44	78379	9678801	49	616778051	347	47950631	13
45	89000	683836354	300	182132136	148	655792064	612
46	24901	114405869	107	227156101	1755	484360413	187
47	43419	129413855	271	280115103	395	93769359	201
48	151337	6882309	5	685620899	987	574204267	277
49	148460	215414888	462	340729066	1662	1985738	528
50	66196	530343560	40	5158498	1242	78772590	200
Average	**183865.58**	**276128149**	**175.82**	**297708686.2**	**641.82**	**279218851.1**	**255.78**

Table 2: Timing of Algorithms

	Repeated Random		**Hill Climbing**		**Simulated Annealing**	
	Random Move	**Pre-Partitioning**	**Random Move**	**Pre-Partitioning**	**Random Move**	**Pre-Partitioning**
1	1.624	11.357	1.225	6.029	3.053	17.423
2	1.683	11.901	1.214	6.882	3.279	17.984
3	1.810	15.245	1.478	7.191	3.621	18.459
4	1.713	12.066	1.328	6.243	3.388	23.799
5	2.261	14.322	1.373	6.852	3.337	18.882
6	1.862	12.754	1.142	6.511	3.240	18.003
7	1.701	12.180	1.312	6.660	3.270	18.092
8	1.644	12.055	1.275	6.691	3.337	18.166
9	1.764	13.270	1.413	7.793	3.724	17.966
10	1.919	13.523	1.283	6.493	3.174	19.029
11	1.845	14.133	1.147	6.467	3.339	18.053
12	1.672	12.714	1.289	7.085	4.541	17.988
13	1.837	12.457	1.206	6.806	4.015	16.708
14	1.549	12.496	1.306	7.470	3.787	19.678
15	1.630	12.842	1.244	6.170	3.345	19.188
16	1.760	11.484	1.124	7.638	3.188	18.199
17	1.845	12.175	1.173	6.367	3.084	17.990
18	1.686	12.092	1.573	6.774	3.109	19.195
19	2.165	15.772	1.214	6.535	3.235	20.834
20	1.864	11.986	1.069	6.084	3.184	16.772
21	1.591	11.439	1.100	5.835	2.984	17.577
22	1.546	11.660	1.122	5.774	3.124	16.745
23	1.629	11.440	1.219	5.730	2.964	16.797
24	1.574	11.427	1.087	6.018	3.236	17.676
25	1.627	11.339	1.204	6.687	3.371	17.972
26	1.571	12.169	1.088	5.973	3.017	17.212
27	1.718	12.124	1.187	6.220	3.146	17.377
28	1.601	11.768	1.245	6.105	2.920	17.126
29	1.783	12.275	1.329	6.542	3.016	17.973
30	1.699	12.287	1.159	6.610	3.108	16.850
31	1.597	12.003	1.104	6.108	3.108	17.215
32	1.629	11.842	1.142	6.877	3.481	19.382
33	1.787	12.568	1.179	7.114	3.540	20.375
34	1.769	12.258	1.101	7.279	3.550	20.576
35	1.755	11.982	1.300	6.302	3.164	17.508
36	1.688	12.087	1.286	6.499	3.427	18.410
37	1.724	12.484	1.272	6.727	3.297	18.729
38	1.833	12.799	1.501	12.022	4.902	23.656
39	1.877	15.128	1.409	6.935	3.657	19.142
40	1.729	13.128	1.193	6.874	4.116	21.917
41	1.943	12.272	1.238	6.322	3.326	18.204
42	1.890	12.289	1.042	6.268	3.535	21.559
43	1.948	14.638	1.230	7.010	3.400	18.319
44	1.759	12.039	1.126	8.983	3.536	17.950
45	1.781	12.742	1.230	6.369	3.206	18.557
46	1.761	11.987	1.106	6.165	3.196	19.090
47	1.571	11.459	1.262	6.113	3.270	17.434
48	1.613	12.167	1.262	6.323	3.127	17.755
49	1.705	11.707	1.082	6.170	3.175	18.026
50	1.837	12.056	1.310	6.163	3.206	18.448
Average	**1.747**	**12.488**	**1.230**	**6.697**	**3.367**	**18.559**

Figure 1: A comparison of the relative residues for the three algorithms, using random move

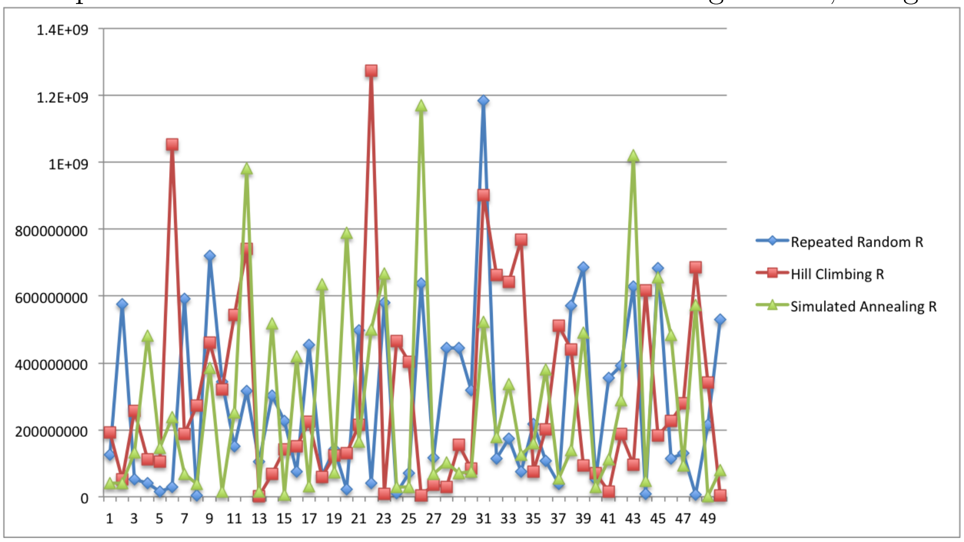
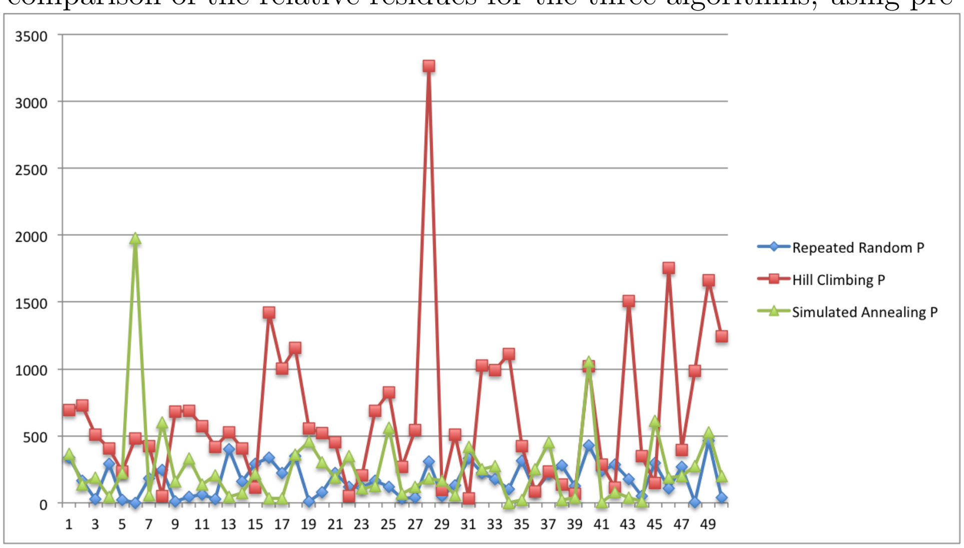


Figure 2: A comparison of the relative residues for the three algorithms, using pre-partitioning



4 Using the Karmarkar-Karp Solution as a Starting Point for the Randomized Algorithms

As per our results, the Karmarkar-Karp (KK) algorithm provided a residue that was consistently lower than the algorithms involving random move. Thus it could be useful to generate an initial solution S using the KK algorithm rather than generating a solution at random. Given that the KK algorithm yielded a better solution (i.e. lower residue) than did the random move, we have reason to believe that running the three iterative algorithms (repeated random, hill climbing, and simulated) after an initial run of KK would yield even better solutions than those that the three algorithms yielded using random move. Alternatively, running KK in combination with these three algorithms could yield a similarly viable solution in fewer iterations.

Using KK would be particularly useful for the hill-climbing and simulated annealing algorithms, as these two involve randomly selecting from the neighbors of the initial solution, and so a better initial solution would yield to a better overall residue (or fewer iterations). Repeated random, on the other hand, involves iterations in which random solutions in the space are generated, and so the initial solution is not as impactful (apart from checking the relative residues, which is done in all 3 algorithms).