CS124 Programming Assignment 3

Ryan Wallace

April 21, 2016

Dynamic Programming Solution

The number partition problem on a sequence of integers $A = (a_1, a_2, ..., a_n)$ can be solved using dynamic programming. Define

$$b = \sum_{i=1}^{n} a_n.$$

And define the output as the sequence $S = (s_1, s_2, ..., s_n)$ of signs $s_i \in \{1, +1\}$., which correspond to subsets S_1 and S_2 with sums A_1 and A_2, respectively, and without loss of generality call $A_1 \leq A_2$. Now, note that determining the optimum residue

$$\min\left(u = \sum_{i=1}^{n} |s_i * a_n|\right)$$

is equivalent to

$$\max(\min(A_1, A_2)).$$

Therefore, we wish to find the maximum value $F \leq \lfloor \frac{b}{2} \rfloor$ that can be written as the sum of elements of S, as well as these elements of S that form the sum.

We denote $f(i, j)$ as true if there if there is subset a subset of $(a_1, a_2, ..., a_i)$ with the sum j. Then f can be calculated recursively with the following recurrence:

$$f(i, 0) = \text{true}$$

$$f(0, n) = \text{false} : n \neq 0$$

$$f(i, j) = \begin{cases} \text{true} : \text{if } f(i-1, j) \text{ or } f(i-1, j-a_i) \\ \text{false} : \text{else} \end{cases}$$

Now, this recursion should be computed for $i \in (0, n)$ and $j \in (0, \lfloor \frac{b}{2} \rfloor)$ by filling an $(n+1) \times (\lfloor \frac{b}{2} \rfloor + 1)$ table starting with $(i, j) = (0, 0)$, and filling in row by row. While filling in the table, additionally store a a pointer from each element $(i, j) =$ true to the element (p, q) which set element (i, j) to true by the recurrence. Let $o = \max(j)$ such that $f(n, j) =$ true. The residue is then given by $u = b - o$. The sequence S defining the partition can be reconstructed by following the pointers back from the element (n, o) to element (u, v), taking the subset of S from $(1, u)$ less those elements corresponding to a horizontal move in the table during the pointer following.

The running time of this algorithm is $O(nb)$ because for each of the $(n+1) * (\lfloor \frac{b}{2} \rfloor + 1) \in O(nb)$ elements of the table, we perform constant time arithmetic and comparison operations. The space complexity of the algorithm is also $O(nb)$ as we must maintain a table of $(n+1) * (\lfloor \frac{b}{2} \rfloor + 1) \in O(nb)$ booleans.

Explanation of Karmarkar-Karp

First we derive the number of iterations in one call to Karmarkar-Karp. At each iteration of the K-K algorithm, two integers are replaced by one integer representing the difference, until only one integer remains. Therefore, if

there are n integers to start, K-K iterates $(n-1)$ times. Thus, when implementing K-K, it is important to realize that standard sorting or scanning for the largest two elements at each iteration will lead to at minimum an $O(n^2)$ algorithm because scanning the list of remaining integers for the two maximal elements takes time $O(n)$, and is performed once each $O(n)$ iterations.

Thus, we need a different method of finding the two maximum elements remaining at each iteration of the algorithm. Storing the elements in a max-heap is ideal for this job. The implementation is as follows: run max-heapify on the input set to construct a max heap of the input values. At each iteration of the algorithm, call get-max twice to find the largest two values remaining in the heap. Take the difference of these values and insert the difference into the heap. The algorithm terminates when there is one value remaining in the heap, which is the residue as found by K-K.

This implementation runs in $O(n\log(n))$ time because it consists of one call to max-heapify, which is $O(n\log(n))$, $O(2*(n-1))$ calls to get-max, which is $O(\log(n))$, and $O((n-1))$ calls to insert, which is $O(\log(n))$. Thus, the total runtime is $O(n\log(n)+2*(n-1)*\log(n)+(n-1)*\log(n))=O(n\log(n))$.

Experimental Work

Program Setup

Three .c programs were written for this assignment. The first, kk.c, is a program that takes an input file of 100 integers (one per line) and performs the standard Karmarkar-Karp algorithm. It prints to the screen the residue found by the algorithm. The program uses a max heap to "sort" the integers, so the program runs in $O(n\log(n))$ time.

The second program is part.c, which is a program designed for experimentation. The program generates 50 files of 100 integers each on the range $[1, 10^{12}]$, and performs a total of seven algorithms for approximating the number partition optimum residue. These are: the standard Karmarkar-Karp algorithm, and the repeated random (rrb/rrp), hill-climbing (hcb/hcb), and simulated annealing (sab/sap) heuristics for each the binary (b) and pre-partitioning (p) representations. The program computes the average time taken by each algorithm and the average minimum residue found over a number of trials specified by the user. The heuristics are implemented to attempt 25,000 random iterations.

The final program is part_time.c, which is not a required or mentioned in the problem set specification, but the results from which are used in the discussion below. part_time.c is similar to part.c, but rather than complete 25,000 iterations of each random algorithm, an indefinite number of iterations are performed for a given number of seconds (MAX_TIME macro), and the minimum residue found after this interval is returned.

All programs can be compiled simultaneously by typing "$ make". kk.c can be run by typing "$ ./kk inputfile", where inputfile is the name of the file with 100 integers, one per line. part.c can be run by typing "$ ./part". part.c prints results in real time to the screen, as well as summary statistics about the residues and run times for each heuristic at completion of the program. part_time.c can be run by typing "$ ./part_time".

Numerical Results

The values of the residues generated by each of the seven algorithms (standard Karmarkar-Karp, repeated random with a binary representation (RRB), hill climbing with a binary representation (HCB), simulated annealing with a binary representation (SAB), repeated random with a pre-partitioning representation (RRP), hill climbing with a pre-partitioning representation (HCP), and simulated annealing with a pre-partitioning representation (SAP)) for each of the 50 randomly generated values are presented below in Figure 1. In Figure 2, the time taken by each algorithm in each file instance are is presented. In Figure 3, we present a summary statistics table with the mean and standard deviation across files of the residues and time taken by each algorithm. In Figure 4, we graph the residues found for each file by each algorithm. Finally, in Figure 5, we present the results from part_time.c, which reports the average residues found by each algorithm with 1 second of processing time.

Figure 1. Residues Found by Each of the Seven Algorithms.

-	KK	RRB	HCB	SAB	RRP	HCP	SAP
File 1	21333	403164717	743938733	692420123	279	2661	237
File 2	46036	47304578	858349508	250834992	70	424	422
File 3	214702	78042926	56301910	1070138886	300	2730	732
File 4	6892	255499158	573433446	641314874	32	690	422
File 5	141189	925668235	147158871	1684342125	635	89	309
File 6	144898	227902544	2447137766	499691862	64	246	98
File 7	106270	640227664	165743024	12679244	130	78	44
File 8	71702	9835508	243721804	32318150	114	454	84
File 9	504533	302876673	229759819	477111961	123	1023	119
File 10	264135	66105969	211024131	1472150189	65	523	133
File 11	165513	60038465	776340183	355282955	11	445	1
File 12	456054	574487534	152594322	509502430	128	352	184
File 13	33526	307991666	621406742	510380026	636	1280	738
File 14	1124	91205530	221488946	1035325216	164	220	388
File 15	13839	23198377	478937017	162864039	409	677	3
File 16	77999	132477747	76464633	194811293	49	941	1295
File 17	446989	431496777	802438961	430326859	25	961	5
File 18	47764	406592882	67683754	271789218	44	106	34
File 19	36135	206801001	63796961	892569061	5	575	131
File 20	248147	360776865	126391059	286800735	251	117	121
File 21	1E+06	1131968076	101369394	690477632	362	604	52
File 22	60615	411203791	738702245	646428459	179	2337	1
File 23	149980	119320946	342913194	840203354	20	630	46
File 24	166372	50601100	91607880	125317704	150	184	632
File 25	148872	553305554	1155765724	5777098	20	118	12
File 26	225597	169820281	270318965	1068339511	211	667	179
File 27	618338	261908648	37482296	2751813338	52	90	62
File 28	240832	1414766838	1184093794	581868392	116	902	244
File 29	24235	103432499	79616669	349998707	77	1391	3
File 30	725188	379431372	68831418	437385034	4	1886	274
File 31	5229	140263141	457964373	1426875017	235	291	25
File 32	480204	49440696	685237398	1111441958	126	188	116
File 33	3405	61863249	370368149	1153699167	33	255	349
File 34	29106	33560516	1316713494	537979056	36	1464	170
File 35	196980	56803616	160154200	522174216	422	112	58
File 36	211435	28686359	595378207	158277417	135	443	119
File 37	890753	256615521	1387613639	264560313	33	1927	367
File 38	68178	923605858	448466214	157697702	46	396	30
File 39	2E+06	214447159	154959257	324471567	143	143	167
File 40	180860	2855312	1804920750	535345032	48	678	250
File 41	20482	162784848	524072982	744848148	198	198	166
File 42	655039	276360021	278953379	1472515183	57	2995	59
File 43	133318	358614700	79977518	1428581838	196	78	152
File 44	268674	64803896	323540842	291734998	20	222	296
File 45	56446	470745962	206199708	217570242	44	860	52
File 46	388323	75787799	1017735711	150348349	97	171	419
File 47	92098	114745814	130469640	16511972	208	716	444
File 48	45971	81796603	64133575	388603587	161	157	53
File 49	86033	20189925	136569099	278136395	13	63	181
File 50	543342	295246494	951115136	163472384	200	248	446

Figure 2. Time Taken by Each of the Seven Algorithms (in seconds).

-	KK	RRB	HCB	SAB	RRP	HCP	SAP
File 1	4.5E-05	0.07198	0.02382	0.03085	2.10761	2.04882	2.07933
File 2	4.5E-05	0.06978	0.02165	0.03006	2.10987	2.05103	2.06484
File 3	3.9E-05	0.06972	0.02159	0.02961	2.11111	2.0412	2.0695
File 4	4.3E-05	0.07017	0.02178	0.03057	2.11476	2.03806	2.06289
File 5	4.5E-05	0.06961	0.02175	0.0304	2.10535	2.03676	2.0529
File 6	4.2E-05	0.06986	0.02171	0.03032	2.11484	2.08004	2.12169
File 7	0.00004	0.07215	0.02408	0.03554	2.21934	2.05763	2.06731
File 8	3.8E-05	0.06891	0.02145	0.03132	2.12561	2.10293	2.09033
File 9	3.6E-05	0.07087	0.02146	0.02967	2.12825	2.06782	2.07302
File 10	3.7E-05	0.06867	0.02184	0.02976	2.11972	2.07141	2.11428
File 11	3.7E-05	0.06975	0.02278	0.03096	2.24436	2.29636	2.26791
File 12	3.8E-05	0.06971	0.0222	0.0298	2.20721	2.30713	2.31872
File 13	3.8E-05	0.07775	0.02495	0.03368	2.30026	2.32539	2.18033
File 14	2.9E-05	0.06813	0.02149	0.02739	2.40899	2.19906	2.15709
File 15	4.1E-05	0.06849	0.02172	0.02855	2.20592	2.23497	2.14218
File 16	3.2E-05	0.07167	0.0241	0.02789	2.28095	2.36184	2.1458
File 17	4.5E-05	0.06895	0.02214	0.02884	2.22301	2.10628	2.0973
File 18	0.00003	0.06727	0.02131	0.02694	2.16117	2.14731	2.12871
File 19	0.00003	0.06769	0.02456	0.03127	2.21478	2.14415	2.09593
File 20	2.9E-05	0.06853	0.02264	0.02853	2.14026	2.11088	2.14496
File 21	0.00003	0.0693	0.02577	0.03432	2.21469	2.1319	2.12022
File 22	3.1E-05	0.06911	0.02397	0.02928	2.16052	2.10183	2.17338
File 23	3.7E-05	0.07671	0.02575	0.03143	2.19475	2.12376	2.11555
File 24	3.2E-05	0.07257	0.02792	0.03356	2.22497	2.17521	2.20402
File 25	5.1E-05	0.07188	0.02839	0.03506	2.21204	2.16315	2.13324
File 26	3.7E-05	0.07793	0.0233	0.03721	2.21865	2.15322	2.18292
File 27	0.00003	0.07273	0.0274	0.03217	2.29301	2.16219	2.1762
File 28	0.00003	0.07597	0.02669	0.03208	2.2256	2.18524	2.15853
File 29	0.00003	0.07289	0.02762	0.02996	2.15077	2.11907	2.15677
File 30	3.1E-05	0.07227	0.02639	0.03514	2.19981	2.14375	2.12974
File 31	3.2E-05	0.07378	0.02671	0.03051	2.2036	2.12949	2.15586
File 32	0.00003	0.07157	0.02683	0.03415	2.20359	2.16633	2.14894
File 33	3.3E-05	0.07766	0.0257	0.03538	2.23283	2.20507	2.18961
File 34	3.1E-05	0.0722	0.0254	0.03741	2.20687	2.1544	2.21155
File 35	0.00003	0.0714	0.03	0.0335	2.24959	2.1348	2.15879
File 36	3.1E-05	0.07527	0.02674	0.03294	2.26252	2.1257	2.16736
File 37	2.9E-05	0.07776	0.02601	0.03238	2.24519	2.14415	2.19152
File 38	3.3E-05	0.06923	0.02771	0.03186	2.17048	2.14252	2.17222
File 39	0.00003	0.07353	0.02836	0.03338	2.2424	2.16014	2.16791
File 40	4.2E-05	0.07355	0.02685	0.03228	2.24644	2.15628	2.18966
File 41	0.00003	0.07691	0.02556	0.03169	2.27669	2.21758	2.272
File 42	4.5E-05	0.07183	0.02591	0.03441	2.2349	2.12313	2.17575
File 43	2.9E-05	0.07679	0.0265	0.03368	2.20989	2.09666	2.20375
File 44	3.3E-05	0.06899	0.02174	0.02727	2.20239	2.22181	2.17229
File 45	3.4E-05	0.07123	0.02566	0.036	2.23372	2.18796	2.14463
File 46	3.8E-05	0.0696	0.02416	0.03331	2.23369	2.15351	2.14804
File 47	3.1E-05	0.0678	0.02184	0.03142	2.19501	2.20237	2.20315
File 48	0.00003	0.07245	0.02695	0.03006	2.24091	2.08633	2.13171
File 49	3.1E-05	0.07504	0.02507	0.03168	2.19508	2.09378	2.11556
File 50	3.1E-05	0.07079	0.02622	0.03261	2.23885	2.12032	2.20534

Figure 3. Mean and Standard Deviation of Residues Times.

-	KK	RRB	HCB	SAB	RRP	HCP	SAP
Mean Residue	252077.480	276733428.200	484587128.800	606502160.160	143.520	700.120	218.480
Std Dev Residue	328211.326	299483955.569	502453807.040	533434847.177	145.289	745.966	242.491
Mean Time	0.0000350	0.0718	0.0246	0.0318	2.2053	2.1462	2.1510
Standard Deviation Time	0.0000057	0.0030	0.0024	0.0025	0.0596	0.0724	0.0550

Figure 4. Plots of Residues Against Files for each Algorithm.

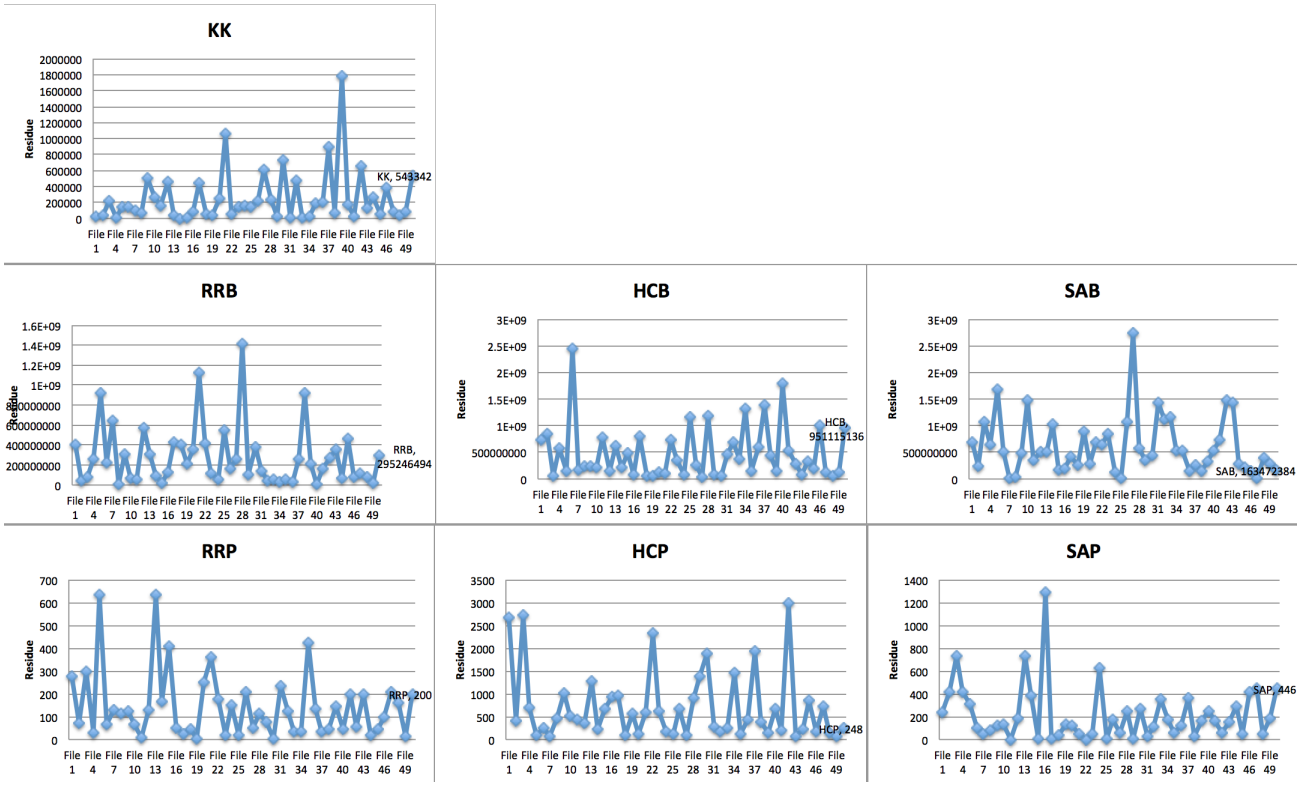

Figure 5. Mean Residues Found by Each Randomized Algorithm with 1 Second of Processing Time.

Algorithm	KK	RRB	HCB	SAB	RRP	HCP	SAP
Mean Residue	181200	18968605	24389553	23845218	432	1374	445

Discussion of Results

There are many interesting patterns and observations to note from these results. Firstly, in the given experiment of 25,000 iterations with each randomized algorithm, the algorithms using the pre-partitioning representation far outperform the algorithms using the binary representations, finding residues on average approximately six orders of magnitude less than those found by the binary representation algorithms. Meanwhile, the standard Karmarkar-Karp algorithm performed better than the binary representation randomized algorithms by approximately three orders of magnitude, but worse than the randomized pre-partitioning algorithms by approximately 3 orders of magnitude.

There is significant variability in the performance of the algorithms between instances (files). For each method, the standard deviation of the residue is approximately equal to the mean of the residue. This implies that there is variability in the results from instance to instance proportional to the performance of the algorithm. That is, the worse performing algorithms (the binary representation algorithms) find higher residues on average, with correspondingly higher standard deviations of residues. Practically, this means that using a better performing algorithm confers the benefit of lower residues on average, but not the benefit of proportionally decreased variability in the results (bad results and good results, due both to variability in the input files and execution of the algorithms, are, in a sense,

equally likely). This idea can be seen in Figure 4, which appears to show similar distributions of residues against files for each of the randomized algorithms.

It is clear that the pre-partitioning representation is the superior solution space and neighborhood representation for this problem. The reason for this is at least two-fold. Firstly, and most significantly, the pre-partitioning representation incorporates the performance enhancing Karmarkar-Karp deterministic algorithm in its solution, rather than using pure randomness of the binary representation. In this sense, the pre-partitioning algorithms are partly randomized and partly deterministic, whereas the binary algorithms are purely randomized. Secondly, the notion of neighbors in the pre-partitioning representation allows for more variability between neighbors. This variability is then optimized by the use of the Karmarkar-Karp algorithm, whereas in the binary representations, variability between neighbors is smaller, and there is no space for it to be optimzed by an additional deterministic algorithm.

In addition to the relative performance of the two representations, it is interesting to note the relative performances of the flavors of algorithms within each representation scheme. For the binary representation, on average the repeated random algorithm found the lowest residue, followed by the hill climbing version, followed by the simulated annealing version. For the pre-partitioning representation, on average the repeated random again found the lowest residue, but the simulated annealing performed second best, with the hill climbing performing worst. One possible interpretation of these results is that for the binary representation, the repeated random method outperforms the hill climbing and simulated annealing methods because the solution space is not smooth, and because the the notion of neighbor is ill-formed. As far as the solution space, we know that it is possible that there are many local minima in the residues, such that no neighbor has a lower residue, but there exist lower residues with subsets that differ from the current subset by more than two substitutions, our definition of a neighbor. Therefore, it is possible that in our hill climbing algorithm, we get stuck in a local minimum and fail to break free to find a potentially smaller minimum found by the brute force repeated random method by chance. As far as the state space, each move to a neighbor changes the residue by on average three times the average value of the elements. This is a sizeable change and should not significantly negatively impact the performance of hill climbing and annealing. Instead, the major issue is likely that the solution space has many local minima, such that even simulated annealing, with the possibility of breaking out of these local minima is unable to overcome the non-smoothness of the solution space.

This problem is also seen in the pre-partitioning representation. In the pre-partitioning representation, the hill climbing performance also suffers because the solution function is not well behaved, but instead has many local minima. However, we note that the simulated annealing method confers a performance boost over normal hill climbing in the pre-partitioning representation, in contrast to the binary representation. This is potentially because the broader variability between neighbors in the pre-partitioning representation increases the likelihood that the a step in the opposite direction of the hill climb breaks free of a local minimum.

Finally, we note that in practical applications, the performance of these algorithms should likely be judged based on the residues found in a given period of time, not after a given number (say, 25,000) of iterations. Since in our results, we see that 25,000 iterations with the algorithms using the binary representations take significantly less time (on the order of 1/100) than the pre-partitioning algorithms, we must question if, given equal computing time, the binary representations would improve their relative performance, and do as well or better than the pre-partitioning representations. As we see in Figure 5, this is not so. When each randomized algorithm is given exactly one second of computing time, the pre-partitioning algorithms still significantly outperform the binary representation algorithms. However, the performance disparity is decreased from an average approximate 10^6 factor difference to an average approximate 10^5 factor difference. Nevertheless, the results are clear that in any application where the goal is to provide the best approximation to the partition problem in a given period of time, the pre-partitioning algorithms are superior to either the binary representation algorithms or the standard Karmarkar-Karp algorithm (which does not allow for performance increases with additional computing time whatsoever).

Karmarkar-Karp as Starting Point

For the binary representation algorithms, using Karmarkar-Karp as a starting point is straightforward. We run Karmarkar-Karp on the input elements, and assign the initial binary representation that coincides with the division into two subsets from Karmarkar-Karp. We then proceed with the algorithms generating new random binary sequences or neighbor sequences as specified by the particular method. Note that modification is logical for

the hill climbing and simulated annealing methods, but for the repeated random method, this reduces to simply the Karmarkar-Karp algorithm, so starting with Karmarkar-Karp is no different than running both repeated random and Karmarkar-Karp and taking the minimum resulting residue. For hill climbing and simulated annealing methods, this modification is likely to produce lower residuals on average than either the normal versions of hill climbing or simulated annealing or standard Karmarkar-Karp alone. This is because we are effictively giving the randomized algorithms a hint at where a local optimum may lie in the state space, and allow them to work from there. Nevertheless, since initializing with the results of Karmarkar-Karp may prevent the randomized algorithms from breaking out from that local optimum area, where they otherwise would have otherwise found other optima, the performance increase of these modified randomized algorithms over standard Karmarkar-Karp may be small.

For the pre-partitioning algorithms, we can use Karmarkar-Karp as a starting point for each of the repeated random, hill climbing, and simulated annealing by generating an initial partitioning that respects the grouping of the elements found by Karmarkar-Karp. To do so, at the beginning of the method, we run standard Karmarkar-Karp. Then we generate a partly-random partition on the input elements, which is a random sequence of n values on the range $(1, n)$, with the additional restriction that if two elements e_i and e_j are found in the same Karmarkar-Karp subset, the initial random indices assigned to them are equal, $r_i = r_j$. This modification will restrict the solution space over which Karmarkar-Karp can operate in the body of the algorithm, but does provide it a better starting point. This will likely lead to approximately a net neutral effect on performance compared to the unmodified pre-partitioning algorithms.