CS 124 Programming Assignment 3

Avinash Saraf and Ajay Nathan

April 18, 2016

Dynamic Programming Solution to Number Partition

Recurrence

In order to solve this problem, we consider the sums of the subsets we can create. For a set of numbers $\{a_1, a_2, \ldots, a_n\}$ we can consider the numbers $\{a_1, a_2, \ldots, a_{n-1}\}$ split into the two groups and the same numbers split into two groups with a_n added to a group.

To determine the best partition we can make, we define a number i that goes from 1 to $\lceil b/2 \rceil$ and we check whether one of the numbers in the set is equal to i or whether some combination of numbers in the set sums to i. We do this with the following recursion for $0 < j \leq n$, where $p(i, j)$ is equal to 1 if some subset of the numbers from a_0 to a_j sums to i:

$$p(i,j) = \begin{cases} 1 & i = 0 \\ max(p(i, j-1), p(i - a_j, j-1)) \end{cases}$$

Once we have filled out the $n \times b$ matrix for p, we scan the last column and determine the largest value that can be obtained by numbers in the set. Thus, this recurrence determines how close we can get to half of the sum of all of the numbers, which is the ideal sum for one partition.

To determine how to partition the set, we create an array of parent pointers that mirrors the array of $p(i, j)$ but only sets entries to 1 if $p(i - a_j, j - 1)$ was 1, and sets all of the other entries to -1. The algorithm starts at entry (i, n) for $i = b$ and looks backwards for the first row in column n that has a 1. It then moves to entry $(i - a_n, j - 1)$ and moves leftward until it finds the next 1, repeating the same procedure, while partitioning the elements based on the value in the column. Once $i = 0$, the algorithm puts the remaining $j - 1$ values to the -1 partition and returns the solution set.

Proof of Correctness

In our 2 dimensional matrix, all values in the first row will be 1 because of the first condition. For all other values, they will only be 1 in the following cases:

1. There is a value in the row that is equal to the current i. In this case, $i - a_j = 0$ and the 1 from the first row will carry over.
2. There is a subset of numbers from a_0 to a_{j-1} that sum to the current i. In this case, $p(i, j - 1)$ is met.
3. There is a subset of numbers in the current row that, combined with the current a_j, sums to i. In this case, $i - a_j$ will bring us to a previous value i' such that some other combination of numbers summed to i', which upon recursing all the way back will eventually lead to the first case. In this case, because we are including a_j, we set the value of i, j value in the parent pointer matrix to 1, as we want to include it in the partition that sums to i.

By finding the maximum value of i for which $p(i, n) = 1$, we can determine the best partition of numbers where $u \leq 2(b - i)$ for this max i.

For our parent pointer matrix, we search for a subset of numbers within the set that sums to this max i. By changing row only when we know the $p(i - a_j, j - 1)$ condition was met in the original array, we are searching for the exact values that got us to this max i, and we already know that this is the best way to partition the set based on the previous analysis. The other set of numbers will be determined automatically by the algorithm moving "left" past them or having already reached $i = 0$.

Runtime

We assume that addition is a constant time operation. Constructing the $p(i, j)$ matrix takes $O(nb)$ time, as there we perform a constant number of operations for each of the nb elements in the table. Constructing the parent pointer array takes the same amount of time. Determining the best sequence of signs for the solution takes $O(n + b)$ time to traverse the parent pointer array.

Karmarkar-Karp Runtime

In order to perform the Karmarkar-Karp algorithm in $O(n \log n)$ steps, we can use a binary max heap to keep track of the numbers in our set. Inserting all the numbers into the heap is $O(n \log n)$, as insertion into a binary heap is $O(\log n)$, and we are inserting n elements. To perform KK, we pop the two max values off the heap and insert their difference back into the heap. We should continue this process until there is only 1 non-zero element left in the heap. Insertions and deletions (popping) are both $O(\log n)$ operations. Thus, the number of operations for the KK algorithm is $3(n - 1)$, as for each iteration, we are deleting two elements and adding one, and therefore, reducing the size of the heap by 1 each iteration (we stop after $n - 1$ iterations). Thus, the runtime of the KK algorithm is $O(3(n - 1) \log n) = O(n \log n)$. It is worth noting that after each iteration, the number of elements in the heap actually decreases, so the runtime is actually less.

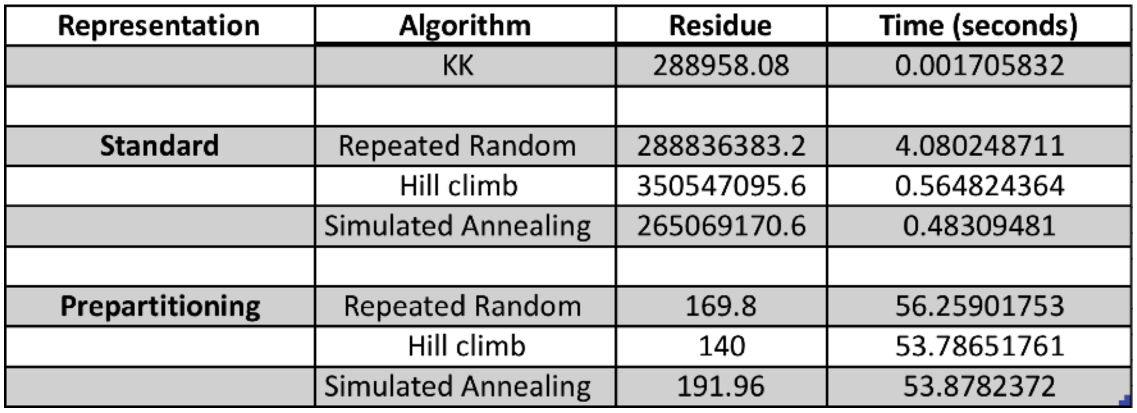
Comparison of Randomized Algorithms

We implemented the Karmarkar-Karp algorithm and the randomized algorithms in Python. We started the randomized algorithms with the same initial solution every time to allow for a more accurate comparison, especially in the cases of simulated annealing and hill climbing.

Summary

For the standard representation of the solution, the simulated annealing algorithm gave us the best results, followed by the repeated random, and then the hill climbing. With prepartitioning, all of the residues were much lower and quite close to each other, with hill climbing only slightly edging out repeated random. The run times for the standard representation were much faster.

To better understand what was happening, we graphed our results:



Representation	Algorithm	Residue	Time (seconds)
	KK	288958.08	0.001705832
Standard	Repeated Random	288836383.2	4.080248711
	Hill climb	350547095.6	0.564824364
	Simulated Annealing	265069170.6	0.48309481
Prepartitioning	Repeated Random	169.8	56.25901753
	Hill climb	140	53.78651761
	Simulated Annealing	191.96	53.8782372

Figure 1: Average residues and running times for each algorithm.

Standard Representation Results

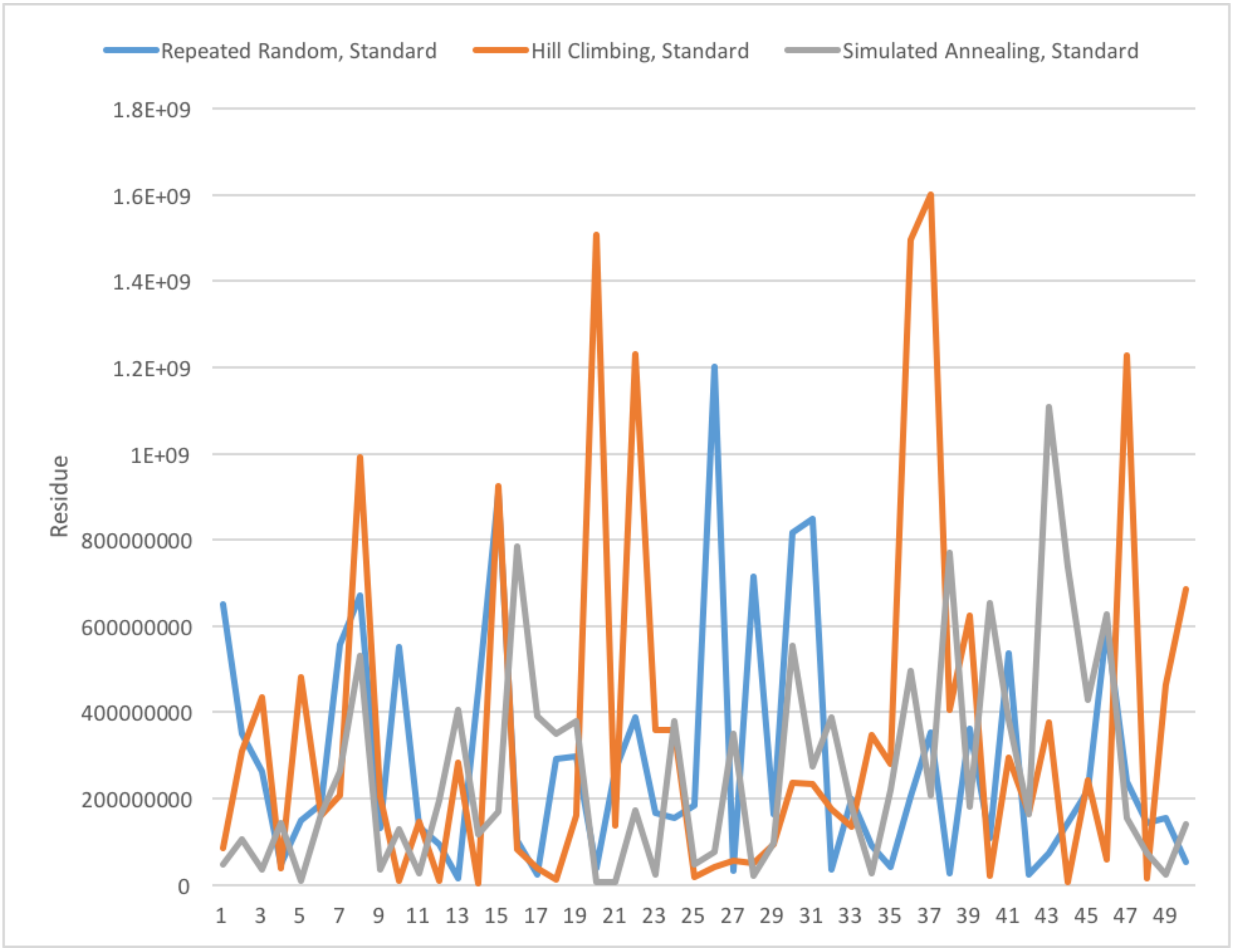


Figure 3: Residues for standard representation.

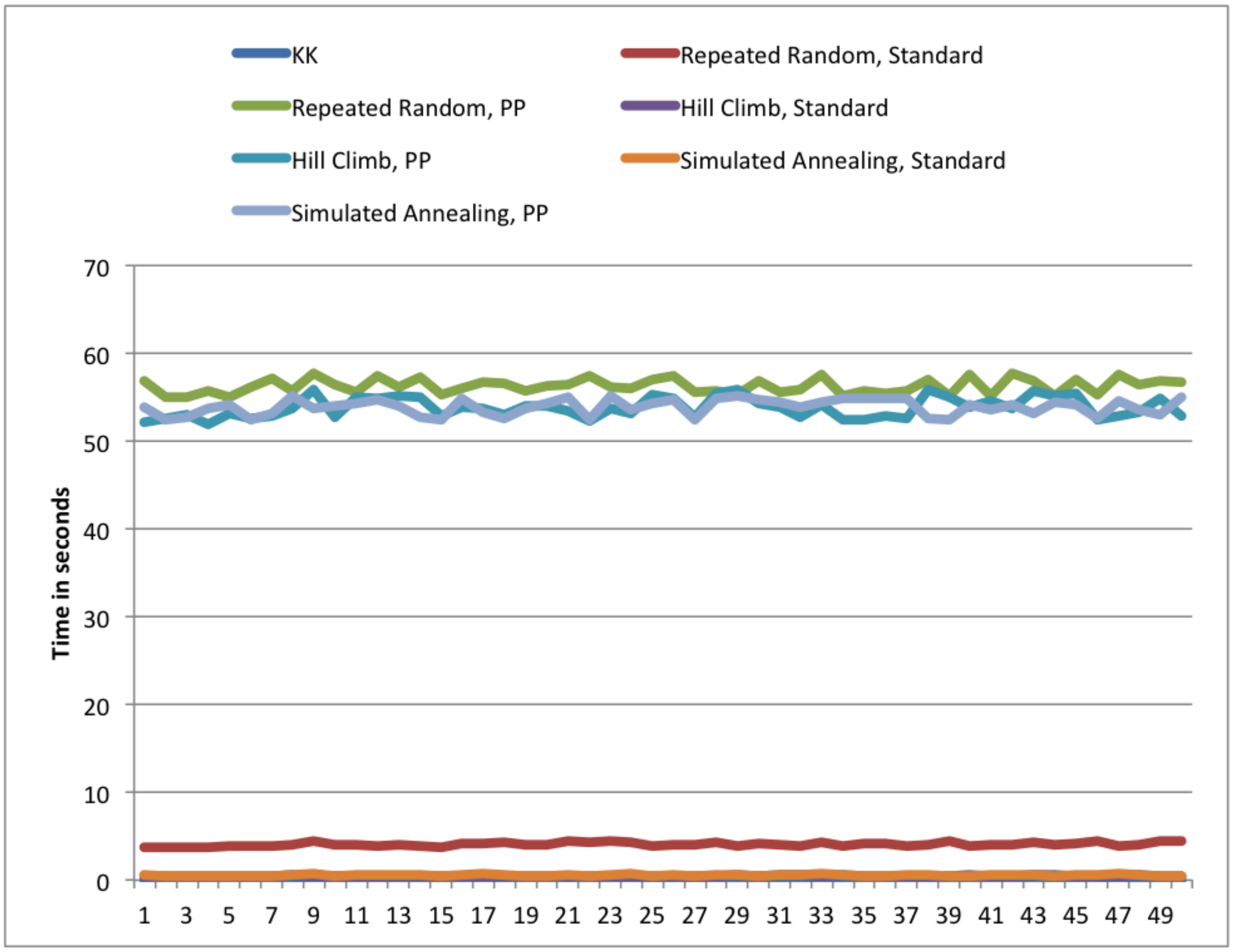


Figure 2: The standard representations ran much faster than the prepartitioned representations.

In the standard representation, we see that the best performances for all of the algorithms hovers well below 100,000,000 (the averages are around 300,000,000), but there is much more variation in their worst performances. The simulated annealing algorithm seems to have the most consistent performance, with fewer peaks above 600,000,000 than the other two algorithms. This intuitively makes sense because simulated annealing has the flexibility to both improve from a starting point (unlike repeated random) and can also "restart" (unlike hill climbing). By "restart" we mean that neighbors of worse neighbors might actually be better, and simulated annealing allows us to find those better solutions (in effect, increasing the size of the possible solution space). If the repeated random algorithm finds a good solution, instead of finding neighbors that could provide even better solutions, it jumps to a new solution that has a high likelihood of being poor. On average, it will find a relatively good solution (when compared to simulate annealing), but sometimes it does not. Conversely, the hill climbing algorithm has a lot of variation in results. We conjecture that this is because if hill climbing starts with a bad solution, it is "stuck" with that solution in the sense that it cannot move very far from it. Since the solution space is very large, the likelihood of starting with a bad solution is non-trivial. This is reflected by the many large peaks in our chart, where, presumably, the hill climbing algorithm started with a bad solution. That is, the success of the hill climbing algorithm is highly dependent on the initial solution, which is why it has the most variance in its results. Thus, the simulated annealing algorithm's consistency and lowest average residue over 50 trials reinforces its superiority over the other two algorithms in the standard representation. This is particularly important when comparing the results of simulated annealing on the sets where hill climbing produced an abnormally large value (the initial solution was bad), such as for the 20th set, where simulated annealing produced an abnormally low reside. This is presumably because simulated annealing was able to restart with a different solution.

Prepartitioning Representation Results

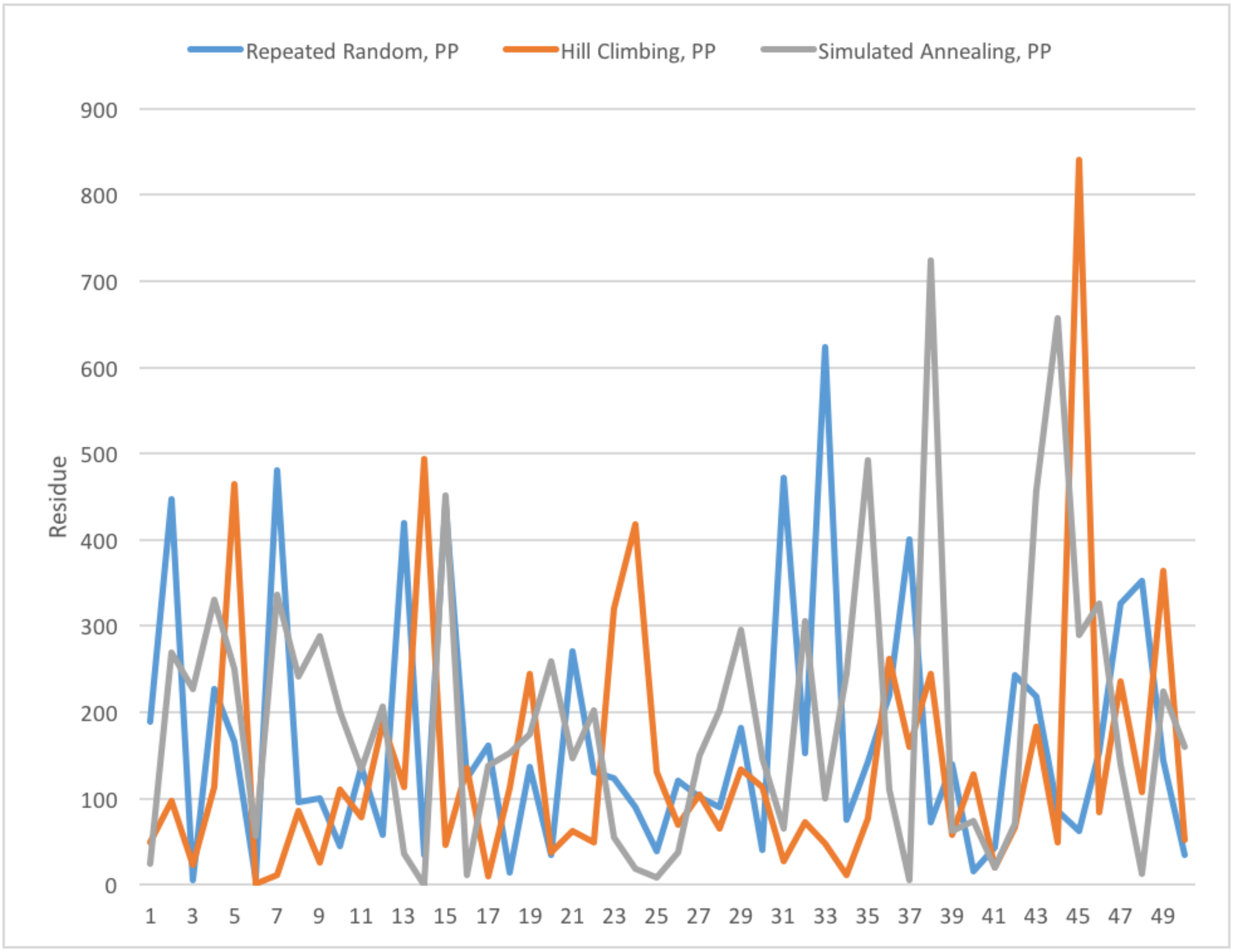


Figure 4: Residues for prepartitioning.

In the prepartioning representation, we see that our results are much better than with the standard representation. One explanation for this might be that prepartioning significantly reduces the size of our solution space, so if we run 25000 iterations of any of the algorithms, we are more likely to find a good solution. Prepartioning decreases the size of the solution space because we force elements to be in the same partition, effectively reducing the size of our list of numbers.

We also see that there is less variation in the worst performances of all the algorithms than in the standard representation. Additionally, the algorithms are of more or less the same consistency. Presumably, this is because the solution space is much smaller, so the differences between the algorithms start to fade away after 25000 iterations. However, it seems that simulated annealing performs worse than hill climbing if we prepartition. This might be because simulated annealing switches to a worse initial prepartition when it could have improved more on the current prepartition, whereas hill climbing sticks to the initial prepartition, continues to improve on it, and never goes backwards. We expect, as for the standard representation, repeated random to fall in the middle, as the solutions it produces are indeed random.

Using Karmarkar-Karp as a Starting Point

In order to use KK as a starting point for the standard representation, we can sort our original list and assign 1 and -1 to the elements in our list in an alternating fashion to create our starting solution. For the algorithms that use prepartitioning, we assign numbers 1 and 2 to alternating numbers in the sorted list so that they are in the same partition (this is our starting prepartition). For our repeated random algorithm, the starting point with which to compare further random solutions should be the KK starting point. For hill climbing and simulated annealing, we should start testing random neighbors of the initial solution produced by KK.

1. Repeated Random: After the first iteration, the KK starting point will have little impact because a completely random solution will be tested every iteration. It is possible that the initial KK solution is the best, especially considering how much smaller the average for our KK results are compared to the repeated random in the standard representation. Thus, the algorithm may just return the KK result.
2. Hill-climbing: As mentioned before, the success of the hill climbing algorithm depends heavily on the starting point. Thus, the KK starting point will probably have the biggest impact on the hill climbing algorithm because it limits the solution space to better solutions. By giving the algorithm a "good" starting point, it significantly reduces the possibility of the algorithm falling into a trap and being stuck looking for a good solution when there are only bad solutions available.
3. Simulated Annealing: Similarly to hill climbing, the simulated annealing algorithm depends on the starting point, though less so because it can switch to worse neighbors. By making the solution space better by starting the algorithm with a better solution, it will likely reduce the average residue.

Appendix

Trial	KK	RR, Standard	RR, PP	HC, Standard	HC, PP	SA, Standard	SA, PP
1	614585	651788523	189	85354135	49	46401323	25
2	84041	350455761	447	309070109	97	106450793	269
3	53621	261489655	5	434984237	23	36343395	227
4	49547	46280603	227	38070973	113	141975313	331
5	87968	148587604	166	481497606	464	7920640	250
6	40249	187264675	7	160202237	1	163561299	57
7	58829	556274679	481	206308723	11	262321379	337
8	78338	670208184	96	991389422	86	530763280	242
9	87398	131136208	100	204631960	26	36772856	288
10	414945	550521751	45	7693341	111	129686627	201
11	73436	133630488	136	145779792	78	26542386	134
12	83252	92756496	58	8163392	188	195631436	206
13	1978376	14888896	420	284301570	114	406180918	36
14	628778	447920768	34	3215056	494	115602208	0
15	1716362	906513562	442	923257678	46	169730160	452
16	91347	101353929	123	82413723	135	783775027	11
17	201782	22795858	162	39142534	10	391715276	138
18	79584	290749096	14	11714890	112	349261476	152
19	75144	297455488	136	159632864	244	379146064	174
20	721843	38933699	35	1506265281	37	5264163	259
21	44393	264966481	271	137927799	63	7505473	147
22	233969	387706417	131	1231131637	49	171508803	203
23	84305	166674739	123	358795093	321	24519181	55
24	68336	155470460	90	359620774	418	378813894	18
25	15515	182960467	39	19154535	131	47586407	9
26	56138	1202606210	120	40255306	70	77301424	38
27	8930	31961594	102	56542216	104	351139838	150
28	2978	715713332	90	51267140	66	21469114	202
29	243730	164624608	182	93714734	134	96492824	296
30	150511	815412755	41	237558939	113	553930827	147
31	17784	849494834	472	232526324	28	274066794	66
32	1134	34149336	152	176114338	72	387253398	306
33	123966	197696738	624	134276880	48	181863262	100
34	205962	89522548	76	346861250	12	25511862	244
35	506761	40767663	143	280595385	77	219566721	493
36	50252	204768710	220	1495155764	262	496528672	110
37	2696924	353258836	400	1601005346	160	208143678	6
38	470214	27988310	72	405793674	244	770894036	724
39	105210	362103156	140	623542768	58	182406712	62
40	302830	107340056	16	21269616	128	653416656	74
41	31772	537381506	44	295901774	20	381375778	20
42	454977	25099609	243	166816359	67	164807171	71
43	365564	73614156	218	376214472	184	1108979026	458
44	31513	143429087	85	6170711	49	737845327	657
45	51340	220225412	62	243694610	840	427820634	290
46	20590	599119176	154	59043958	84	625901872	326
47	592926	238038212	326	1227990294	236	155417120	140
48	238253	143296203	353	16141807	107	73137879	13
49	33842	153692218	144	463983086	364	23904276	224
50	17860	51730410	34	685198666	52	139303854	160

Figure 5: Table of residues for 50 trials of all algorithms.

Trial	KK	RR, Standard	RR, PP	HC, Standard	HC, PP	SA, Standard	SA, PP
1	0.002078	3.7703	56.8903	0.4353	52.1813	0.6136	53.8237
2	0.001849	3.8087	55.0440	0.4629	52.6242	0.4718	52.4126
3	0.001643	3.7868	55.0109	0.4451	52.9516	0.4817	52.7442
4	0.001638	3.7416	55.7116	0.4627	51.9054	0.4672	53.7854
5	0.001500	3.8203	54.9515	0.4503	53.2009	0.4877	54.1475
6	0.001559	3.9437	56.0811	0.4400	52.5009	0.4428	52.4248
7	0.001658	3.8597	57.1386	0.4620	52.7922	0.4748	53.0782
8	0.001540	4.0042	55.6602	0.5464	53.6593	0.6307	55.1842
9	0.001466	4.4346	57.7005	0.4396	55.8926	0.6795	53.7400
10	0.001473	4.0053	56.4107	0.4709	52.7280	0.4876	54.0546
11	0.001708	4.0449	55.5623	0.4554	54.9733	0.6688	54.2982
12	0.001514	3.8249	57.4467	0.4985	54.8599	0.6676	54.6678
13	0.001432	4.0364	56.0889	0.5178	55.1746	0.6269	54.0452
14	0.001441	3.8950	57.2992	0.4608	55.0446	0.5598	52.7506
15	0.001660	3.8045	55.3121	0.4576	52.8702	0.4466	52.4794
16	0.001590	4.1059	55.9897	0.4698	53.8528	0.5370	54.8511
17	0.001946	4.1968	56.6454	0.4580	53.6806	0.6709	53.2606
18	0.001952	4.3661	56.5696	0.4597	53.0713	0.6480	52.5515
19	0.001485	4.0243	55.7025	0.4799	53.9464	0.5043	53.6516
20	0.001548	4.0424	56.2495	0.4654	53.9633	0.5219	54.3200
21	0.001567	4.4003	56.4156	0.5005	53.4807	0.5697	55.0516
22	0.001467	4.3019	57.4435	0.4394	52.2880	0.4880	52.4161
23	0.001455	4.4328	56.0868	0.4673	53.7489	0.5899	55.1103
24	0.001708	4.3493	55.9960	0.4356	53.1741	0.6717	53.5254
25	0.002166	3.8367	56.9580	0.5264	55.2508	0.4501	54.2311
26	0.001547	4.0781	57.3964	0.4658	54.7860	0.6620	54.6460
27	0.001636	4.0563	55.6090	0.5233	52.7122	0.5010	52.4260
28	0.001580	4.3336	55.7544	0.4587	55.4152	0.6347	54.8555
29	0.001451	3.9269	55.3380	0.5446	55.8140	0.6545	55.0947
30	0.001493	4.1833	56.8574	0.4596	54.3311	0.4786	54.7496
31	0.001558	4.0203	55.5698	0.5374	53.7874	0.6435	54.4437
32	0.001534	3.8805	55.9214	0.5315	52.7856	0.6517	53.8301
33	0.002866	4.3123	57.5897	0.5164	54.1387	0.6792	54.4763
34	0.001689	3.9428	55.1653	0.5280	52.3797	0.6670	54.8501
35	0.002061	4.1318	55.7389	0.5017	52.4749	0.4851	54.8394
36	0.001605	4.2286	55.4218	0.4837	52.8215	0.5064	54.8062
37	0.001445	3.8846	55.7322	0.4388	52.5907	0.5609	54.9079
38	0.002374	4.0866	56.9765	0.4412	55.8357	0.5902	52.5965
39	0.001444	4.4011	55.1331	0.4661	54.9527	0.4759	52.4260
40	0.001486	3.8285	57.5934	0.5407	53.9276	0.4855	54.1831
41	0.001559	4.0154	55.1632	0.4857	54.5678	0.6280	53.6145
42	0.001538	4.0740	57.6561	0.5102	53.6900	0.5380	54.1878
43	0.001488	4.3653	56.8634	0.5371	55.7011	0.6698	53.0980
44	0.001619	4.0061	55.1303	0.5303	55.1967	0.4971	54.3848
45	0.001630	4.2395	57.0353	0.5133	55.3664	0.6042	54.1744
46	0.002162	4.4269	55.2921	0.4455	52.3996	0.5631	52.5619
47	0.001852	3.9366	57.5815	0.4934	52.8912	0.6774	54.6089
48	0.001667	3.9760	56.4741	0.5304	53.3364	0.5571	53.5466
49	0.003337	4.4227	56.8589	0.5112	54.8181	0.5124	53.0323
50	0.001601	4.4144	56.7309	0.4507	52.7873	0.4569	54.9633

Figure 6: Table of running times in seconds for 50 trials of all algorithms.

How to Run

Running kk.c:

1. Type ”make” into terminal
2. Type ”python generate_inputfile.py”
3. Type ”./kk input.txt”

Running the randomized algorithms:
Type ”python randomized_algorithms.py”