CS 124 Programming Assignment 3: Spring 2024

Your name(s) (up to two): Pedro Garcia Joshua Zhang

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:
No. of late days used after including this pset:

Homework is due Thursday 2024-04-18 at 11:59pm ET. You are allowed up to **twelve** late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two**.

For this programming assignment, you will implement a number of heuristics for solving the Number Partition problem, which is (of course) NP-complete. As input, the number partition problem takes a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers. The output is a sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in \{-1, +1\}$ such that the *residue*

$$u = \left| \sum_{i=1}^{n} s_i a_i \right|$$

is minimized. Another way to view the problem is the goal is to split the set (or multi-set) of numbers given by A into two subsets A_1 and A_2 with roughly equal sums. The absolute value of the difference of the sums is the residue.

As a warm-up exercise, you will first prove that even though Number Partition is NP-complete, it can be solved in pseudo-polynomial time. That is, suppose the sequence of terms in A sum up to some number b. Then each of the numbers in A has at most $\log b$ bits, so a polynomial time algorithm would take time polynomial in $n \log b$. Instead you should find a dynamic programming algorithm that takes time polynomial in nb.

Give a dynamic programming solution to the Number Partition problem.

Algo: Let our inputs be given by $\mathcal{I} : A = \{a_1, a_2, \ldots, a_n\}$ as our input array. Additionally, let's define the sum of a subset as the sum of all of its terms. We can set up a two dimensional memoization table DP, where $DP[i, j]$ is a boolean value 1 or 0, denoting True or False respectively. $DP[i, j] = 1$ if for the first i terms of A, $\{a_1, \ldots, a_i\} \subseteq A$, $\exists$ some subset $\alpha \subseteq \{a_1, \ldots, a_i\}$ such that $\text{sum}(\alpha) = j$. Conversely, $DP[i, j] = 0$ if there exists no such subset α.

Our memoization table will be n by $\lfloor \frac{b}{2} \rfloor$, where b is the sum of A. The base case is given by $DP[i, 0] = 1 \; \forall \; i \in \{0, 1, \ldots, n\}$ and $DP[0, j] = 0 \; \forall \; j \in \{0, 1, \ldots, \lfloor \frac{b}{2} \rfloor\}$. Then we can describe the recurrence equation as follows.

$$DP[i, j] = \begin{cases} 1 & \text{if } DP[i, j - a_i] \text{ or } DP[i-1, j] \\ 0 & \text{otherwise} \end{cases}$$

We iterate through DP from left to right and top to bottom. Once every entry in DP is filled with either a 0 or 1, the algorithm solves for the minimum residual. First, we check the value of $DP[n, \lfloor \frac{b}{2} \rfloor]$. If $DP[n, \lfloor \frac{b}{2} \rfloor] = 1$, then the minimum residual is 0 for even values of b and 1 for odd values of b. If $DP[n, \lfloor \frac{b}{2} \rfloor] = 0$, then we check the entry directly to its left in the table, or $DP[n, \lfloor \frac{b}{2} - 1 \rfloor]$. If $DP[n, \lfloor \frac{b}{2} \rfloor - 1] = 1$, then the minimum residual is 2 or 3 for even or odd b respectively. Again, if $DP[n, \lfloor \frac{b}{2} \rfloor - 1] = 0$, then we check the entry directly to its left.

In general, we keep decrementing the j-value in $DP[n, \lfloor \frac{b}{2} \rfloor]$ until we encounter a value of 1. More visually, this is shifting to the left along the n-th row of DP. Suppose that the first value of 1 we encounter is $DP[n, \frac{b}{2} \rfloor - k]$ for some whole number k, meaning we decremented k times or shifted k to the left on the n-th row. Then, the residual is given by $2k$ or $2k+1$ for even or odd b respectively. This is our final output.

Pf of Correctness Our proof will contain two parts. (i.) Let's first prove why the our process of memoization is correct. (ii.) Secondly, we will prove our use of the values of DP correctly outputs the minimum residual.

Pf of (i.) We will prove correctness of our recurrence equation by induction. *Base Case:* The boolean value at entry $DP[i, j]$ represents whether or not it is possible to construct a subset summing to j with the first i elements of our input A. In our notation, this is whether or not $\exists$ some subset $\alpha \subseteq \{a_1, \dots, a_i\} \subseteq A$ such that $\text{sum}(\alpha) = j$.

We can set up $DP[i, 0] = 1 \; \forall \; i \in \{0, \dots, n\}$. Visually, we establish that all the entries on the leftmost column in DP are 1. This is because no matter what subset of A we consider, we have $\alpha = \emptyset$, which is a subset of the first i terms of A. The sum of the null set is 0, which is the value of j in this base case.

We also set up $DP[0, j] = 0 \; \forall \; j \in \{1, \dots, \lfloor \frac{b}{2} \rfloor\}$. Visually, we establish that the top row of DP is 0, except for the leftmost term of that row. This is because this base case of $i = 0$ means that we consider the first 0 terms of A, which is the null set. The only subset of the null set is the null set itself, so there only exists $\alpha = \emptyset$, and we previously established that $\text{sum}(\emptyset) = 0$. Thus, if we consider any value $j \neq 0$, it is not possible by our definition of sum. Therefore, we have proven our base case.

Inductive Step: Now we want to prove our recurrence equation: how we solve for $DP[i, j]$. Our *inductive hypthesis* is that we have solved all of the entries of DP in the rows above $DP[i, j]$ as well as to the left in the same row. Getting more technical, the entries in rows above represent whether or not it is possible to find $\text{sum}(\alpha) = j$ for smaller values of i. The entries in the same row to the left represent if its possible to find $\text{sum}(\alpha) = j$ for smaller values of j with the same value of i.

When we evaluate $DP[i, j]$, we are evaluating whether or not it is possible to find a sum j from the first i terms of A. There are two possible ways for verifying if this is true, which we will present as exhaustive cases, because this all depends on whether $a_i \in \alpha$ or $a_i \notin \alpha$. Note that if neither of these cases are satisfied, then it must be false.

Case 1: Suppose $a_i \in \alpha$. That means that in order to find a subset $\alpha \subseteq \{a_1, \dots, a_i\}$ such that $\text{sum}(\alpha) = j$, we include a_i and add it to our sum of j. That means that if we did not include a_i, choosing from the same set of the first i variables of A, we would have the sum $j - a_i$. Thus, $DP[i, j - a_i]$ must equal 1 in this case. Thus, $DP[i, j - a_i] = 1$ iff $DP[i, j] = 1$.

Case 2: Suppose $a_i \notin \alpha$. That means that we do not need to include a_i to reach our sum of j. Then, we may as well look at a smaller subset that did not have a_i in it, like $\{a_1, \dots, a_{i-1}\}$ to see if it is possible to reach a sum of j with this. That is the equivalent of checking if $DP[i-1, j] = 1$.

This aligns with an observation about our memoization table: once we encounter a 1, all of the entries below it in the same column are also 1. This is because even if we add more terms to our subset, we can always count on the original subset to contain our desired α. In this case, we choose to check right above the one we are currently one. Therefore we have established that $DP[i-1, j] = 1$ iff $DP[i, j] = 1$.

These two checks of $DP[i-1,j]$ and $DP[i,j-a_i]=1$ account for all cases of when $DP[i,j]=1$. Since we have proved the if and only if statements above, conversely, if neither are satisfied, then we know that $DP[i,j]=0$. Hence, our recurrence equation is correct. Therefore, we have proven by induction that our algorithm for filling out the memoization table is correct.

Pf of (ii.) By the proof of part (i.), we assume that we have correctly found values for all of DP. Next, our algorithm iterates along the bottom row of DP from right to left, starting at $DP[n,\lfloor\frac{b}{2}\rfloor]$ until we find the first entry $DP[n,\lfloor\frac{b}{2}\rfloor-k]$ that is equal to 1. Then, our output is $2k$ or $2k+1$ for even or odd values of $b=\text{sum}(A)$, where k is the number of squares that we travel left from $DP[n,\lfloor\frac{b}{2}\rfloor]$.

This holds because we can think about our set α as placing the term in one of the partitions A_1. If one of the terms is not in α, then it is in A_2. It would follow that $\text{sum}(\alpha)+\text{sum}(A\setminus\alpha)=b$. If that sum we are considering is $\text{sum}(\alpha)=\lfloor\frac{b}{2}\rfloor$, then we just describe the set complement as $A\setminus\alpha$. Since our $b/2$ value cut the total sum in half, then $\text{sum}(A\setminus\alpha)$ is within 1 of $\text{sum}(\alpha)$.

Generally, by decrementing our j value and shifting to the left, we consider if $\lfloor\frac{b}{2}\rfloor-k$ is a possible sum of some α. We should stop once we find $DP[n,\lfloor\frac{b}{2}\rfloor-k]=1$ for the smallest k, which is rightmost in that bottom row.

$$DP[n,\lfloor\frac{b}{2}\rfloor-k]=1\Rightarrow\text{sum}(\alpha)=\lfloor\frac{b}{2}\rfloor-k\Rightarrow\text{sum}(A\setminus\alpha)=\lfloor\frac{b}{2}\rfloor+k$$

This last equation holds because we established earlier that $\text{sum}(\alpha)+\text{sum}(A\setminus\alpha)=b$. Therefore, the difference between these two is $2k$ for even b or $2k+1$ for odd. Since we minimized k, we also minimized these residuals $2k$ or $2k+1$. Thus correctness is proven. ■

Pf of Runtime Our memoization table DP is n by b. The base cases are constant time because we already know in advance what they should be. The other entries of DP which we figure out inductively based on previous subproblem's results require checking two previous memoized entries of DP. We can describe checking two values as constant time operations, $O(1)$. Thus, constructing DP takes $O(bn)$ time.

Once we construct DP, we iterate through the bottom row to find the minimum residual. Thus, the worse case scenario is when we iterate through the entire row, which is $O(b)$. Hence, our entire algorithm's runtime as $O(bn+b)=O(bn)$. We describe this runtime as pseudo-polynomial. This is because we are simply multiplying two terms together, resembling some kind of polynomial runtime.

However, our sum b is potentially much larger than n. For example, consider $A=\{10^6,10^6,10^6\}$. Then, $|A|=n=3$, but $b=3\cdot10^6$. In essense, the sum of all our numbers can potentially be much larger than the number of numbers we have. More precisely, suppose that all our numbers can be stored as k-bit numbers. Then, b is upper bounded by 2^k. Thus, the runtime in the worse scenario creates a massive DP, with the length of DP growing exponentially. ■

One deterministic heuristic for the Number Partition problem is the Karmarkar-Karp algorithm, or the KK algorithm. This approach uses *differencing*. The differencing idea is to take two elements from A, call them a_i and a_j, and replace the larger by $|a_i-a_j|$ while replacing the smaller by 0. The intuition is that if we decide to put a_i and a_j in different sets, then it is as though we have one element of size $|a_i-a_j|$ around. An algorithm based on differencing repeatedly takes two elements from A and performs a differencing until there is only one element left; this element equals an attainable residue. (A sequence of signs s_i that yields this residue can be determined from the differencing operations performed in linear time by two-coloring the graph (A,E) that arises, where E is the set of pairs (a_i,a_j) that are used in the differencing steps. You will not need to construct the s_i for this assignment.)

For the Karmarkar-Karp algorithm suggests repeatedly taking the largest two elements remaining in A at each step and differencing them. For example, if A is intially $(10, 8, 7, 6, 5)$, then the KK algorithm proceeds as follows:

$$\begin{aligned} (10,8,7,6,5) &\rightarrow (2,0,7,6,5) \\ &\rightarrow (2,0,1,0,5) \\ &\rightarrow (0,0,1,0,3) \\ &\rightarrow (0,0,0,0,2) \end{aligned}$$

Hence the KK algorithm returns a residue of 2. The best possible residue for the example is 0.

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.

The Karmarkar-Karp algorithm finds the largest two numbers in the set at each iteration. This is done most efficiently with a maximum heap. We covered this earlier in the class. Retrieving the maximum, reheapifying, and then retrieving the maximum again are all $O(\log n)$ operations. Thus, every iteration where we find the largest two numbers and perform two arithmetic operations is is $O(3\log n + 2) = O(\log n)$. Additionally, every iteration includes subtraction that reduces one of the terms to 0. The algorithm does not stop until there is only one non-zero value left. Thus, we have $n - 1$ iterations. Hence, we can describe the runtime as $O((n-1)\log n) = O(n \log n)$.

You will compare the Karmarkar-Karp algorithm and a variety of randomized heuristic algorithms on random input sets. Let us first discuss two ways to represent solutions to the problem and the state space based on these representations. Then we discuss heuristic search algorithms you will use.

The standard representation of a solution is simply as a sequence S of $+1$ and -1 values. A random solution can be obtained by generating a random sequence of n such values. Thinking of all possible solutions as a state space, a natural way to define neighbors of a solution S is as the set of all solutions that differ from S in either one or two places. This has a natural interpretation if we think of the $+1$ and -1 values as determining two subsets A_1 and A_2 of A. Moving from S to a neighbor is accomplished either by moving one or two elements from A_1 to A_2, or moving one or two elements from A_2 to A_1, or swapping a pair of elements where one is in A_1 and one is in A_2.

A *random move* on this state space can be defined as follows. Choose two random indices i and j from $[1, n]$ with $i \neq j$. Set s_i to $-s_i$ and with probability $1/2$, set s_j to $-s_j$.

An alternative way to represent a solution called *prepartitioning* is as follows. We represent a solution by a sequence $P = \{p_1, p_2, \ldots, p_n\}$ where $p_i \in \{1, \ldots, n\}$. The sequence P represents a prepartitioning of the elements of A, in the following way: if $p_i = p_j$, then we enforce the restriction that a_i and a_j have the same sign. Equivalently, if $p_i = p_j$, then a_i and a_j both lie in the same subset, either A_1 or A_2.

We turn a solution of this form into a solution in the standard form using two steps:

- We derive a new sequence A' from A which enforces the prepartioning from P. Essentially A' is derived by resetting a_i to be the sum of all values j with $p_j = i$, using for example the following pseudocode:

```
$A' = (0, 0, \ldots, 0)$
```

```
for $j = 1$ to $n$
    $a'_{p_j} = a'_{p_j} + a_j$
```

- We run the KK heuristic algorithm on the result A'.

For example, if A is initially $(10, 8, 7, 6, 5)$, the solution $P = (1, 2, 2, 4, 5)$ corresponds to the following run of the KK algorithm:

$$\begin{aligned} A = (10,8,7,6,5) &\rightarrow A' = (10,15,0,6,5) \\ (10,15,0,6,5) &\rightarrow (0,5,0,6,5) \\ &\rightarrow (0,0,0,1,5) \\ &\rightarrow (0,0,0,0,4) \end{aligned}$$

Hence in this case the solution P has a residue of 4.

Notice that all possible solution sequences S can be generated using this prepartition representation, as any split of A into sets A_1 and A_2 can be obtained by initially assigning p_i to 1 for all $a_i \in A_1$ and similarly assigning p_i to 2 for all $a_i \in A_2$.

A random solution can be obtained by generating a sequence of n values in the range $[1, n]$ and using this for P. Thinking of all possible solutions as a state space, a natural way to define neighbors of a solution P is as the set of all solutions that differ from P in just one place. The interpretation is that we change the prepartitioning by changing the partition of one element. A *random move* on this state space can be defined as follows. Choose two random indices i and j from $[1, n]$ with $p_i \neq j$ and set p_i to j.

You will try each of the following three algorithms for both representations.

- Repeated random: repeatedly generate random solutions to the problem, as determined by the representation.

```
Start with a random solution $S$
for iter = 1 to max_iter
    $S'$ = a random solution
    if residue($S'$) < residue($S$) then $S = S'$
return $S$
```

- Hill climbing: generate a random solution to the problem, and then attempt to improve it through moves to better neighbors.

```
Start with a random solution $S$
for iter = 1 to max_iter
    $S'$ = a random neighbor of $S$
    if residue($S'$) < residue($S$) then $S = S'$
return $S$
```

- Simulated annealing: generate a random solution to the problem, and then attempt to improve it through moves to neighbors, that are not always better.

```
Start with a random solution $S$
$S'' = S$
**for** iter = 1 to max_iter
    $S'$ = a random neighbor of $S$
    **if** residue($S'$) $<$ residue($S$) **then** $S = S'$
    **else** $S = S'$ with probability exp($-$(res($S'$)-res($S$))/T(iter))
    **if** residue($S$) $<$ residue($S''$) **then** $S'' = S$
return $S''$
```

Note that for simulated annealing we have the code return the best solution seen thus far.

You will run experiments on sets of 100 integers, with each integer being a random number chosen uniformly from the range $[1, 10^{12}]$. Note that these are big numbers. You should use 64 bit integers. Pay attention to things like whether your random number generator works on ranges this large!

Below is the main problem of the assignment.

First, write a routine that takes three arguments: a flag, an algorithm code (see Table 1), and an input file. We'll run typical commands to compile and execute your code, as in programming assignment 2; for example, for C/C++, the run command will look as follows:

$ ./partition flag algorithm inputfile

The flag is meant to provide you some flexibility; the autograder will only pass 0 as the flag but you may use other values for your own testing, debugging, or extensions. The algorithm argument is one of the values specified in Table 1. You can also assume the inputfile is a list of 100 (unsorted) integers, one per line. The desired output is the residue obtained by running the specified algorithm with these 100 numbers as input.

Code	Algorithm
0	Karmarkar-Karp
1	Repeated Random
2	Hill Climbing
3	Simulated Annealing
11	Prepartitioned Repeated Random
12	Prepartitioned Hill Climbing
13	Prepartitioned Simulated Annealing

Table 1: Algorithm command-line argument values

If you wish to use a programming language other than Python, C++, C, Java, and Go, please contact us first. As before, you should submit either 1) a single source file named one of partition.py, partition.c, partition.cpp, partition.java, Partition.java, or partition.go, or 2) possibly multiple source files named whatever you like, along with a Makefile (named makefile or Makefile).

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.

Response: The following table and the two Graphs are the results of our experimentation. Essentially, these are the minimum residue found by each of the algorithms in each of the 50 trials, with max_iters= 25000.

Table 2: Residues by algorithm and iteration

	KK	RR	HC	SA	PP RR	PP HC	PP SA
0	138637	20569525	3422449	954700051	59	47	305
1	10220	541464544	1181348718	268978506	110	1226	20
2	6432	78770926	324150690	70777770	176	152	54
3	740	367540	51158818	349212602	258	114	456
4	100000	389922250	88042098	56637240	76	84	402
5	1091665	762820213	355678681	889453889	69	181	475
6	290259	16932369	127110101	159168637	219	203	147
7	1418384	167987642	286018144	1047416124	8	112	26
8	203449	56682075	154459581	66651667	47	7	449
9	187278	613846902	235059942	392119886	18	366	372
10	669217	966583	85476727	2602856365	25	265	33
11	12111	462109025	329521753	817188713	91	7	341
12	145317	594859175	499634593	471074265	53	215	49
13	325213	924009429	484030243	87857065	81	381	281
14	70981	320681301	569139511	228586639	191	5	79
15	317415	259949819	44640705	215417385	319	1255	39
16	3416	375332178	739176486	411213896	220	52	408
17	35233	37711347	717443191	519547143	113	13	197
18	57334	49050946	454939742	374634066	98	228	56
19	66835	180092445	185576447	205814849	27	9	131
20	170214	78904724	538485598	524298410	10	194	374
21	128774	402148096	259801080	1686509662	118	10	338
22	65866	90658640	230453932	1155268156	326	824	754
23	98509	585796693	116017119	238552153	39	485	45
24	38349	336787397	260373767	704651473	17	585	437
25	18428	1167056796	57212672	716951558	774	246	318
26	41910	537107102	884718858	384410336	86	120	20
27	662473	1012732033	49782923	255178963	45	15	99
28	145415	87415547	139558535	601813743	73	217	5
29	164180	818365508	85569058	49664804	22	136	456
30	171041	168119645	57845687	731803889	311	515	945
31	44821	125617035	132928035	237290045	55	79	801
32	254083	8265883	24811507	620064077	355	111	79
33	285289	895917819	33067667	265044893	9	263	501
34	17085	749087329	237194519	365811119	379	153	207
35	22839	269501395	631826451	80001355	87	79	173
36	38275	45670113	148825925	102815045	63	213	147
37	46933	44292585	167050613	46893799	177	559	133
38	113191	127188669	7074483	453681431	81	131	295
39	8224	192095996	6502980	817203156	330	1924	120
40	191810	493563948	427548384	391374660	244	442	54
41	256067	292867303	509090817	173798971	3	97	95
42	129754	1006691208	180674142	151077076	440	28	32
43	1256645	428864183	104383881	1591587843	23	749	59

Continued on next page

Table 2: Residues by algorithm and iteration

	KK	RR	HC	SA	PP RR	PP HC	PP SA
44	181892	498478938	80117898	51913172	358	94	182
45	915927	302620741	281522391	481852951	3	115	129
46	3326	7338764	297675858	388319286	2	236	128
47	92774	300381002	9626002	207030382	166	72	304
48	766772	469577728	147158812	101604798	10	64	128
49	133364	15039036	734435736	238545004	90	50	14

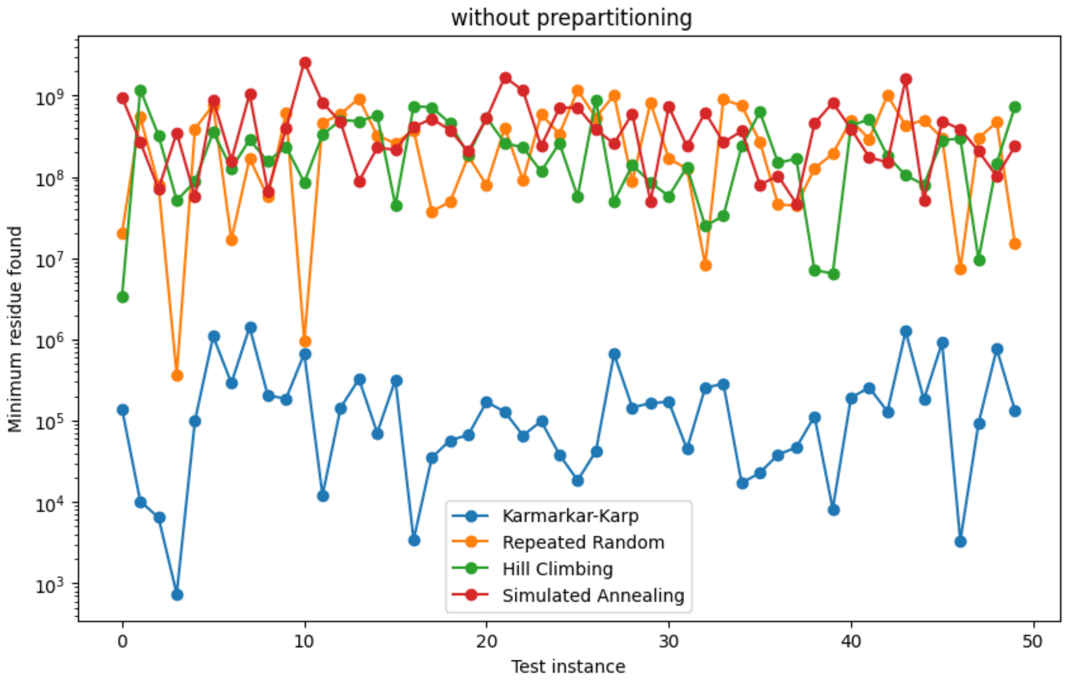


Figure 1: Enter Caption

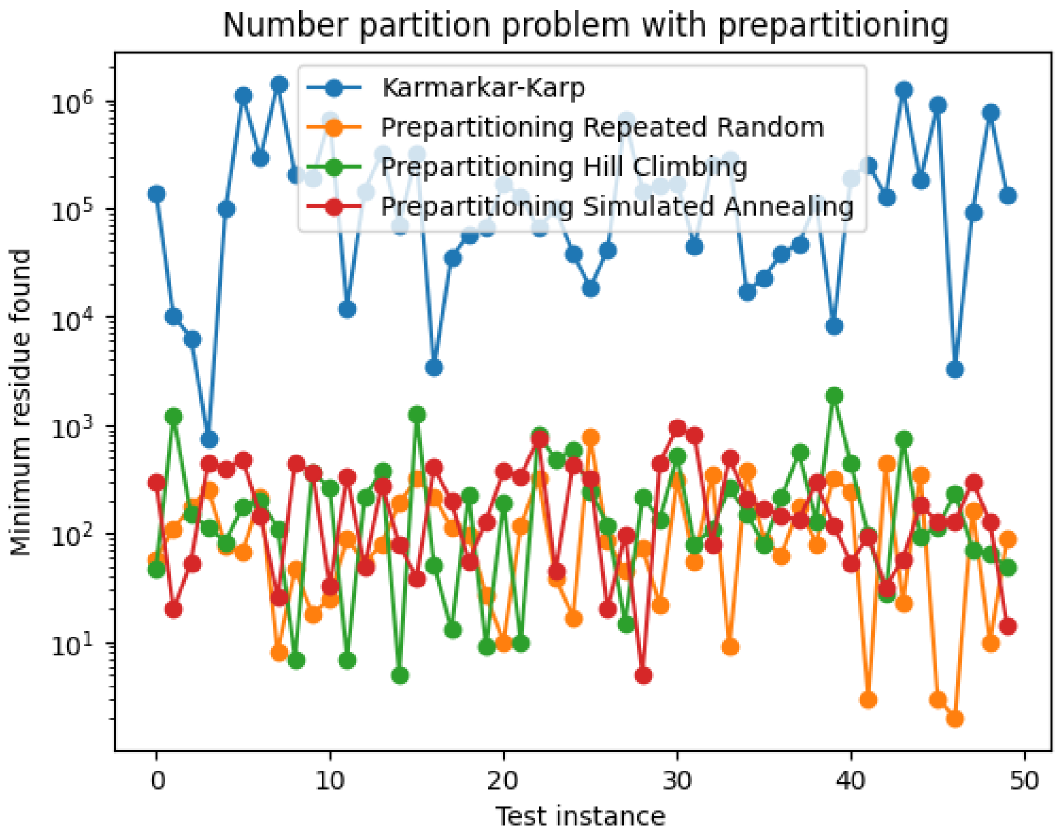
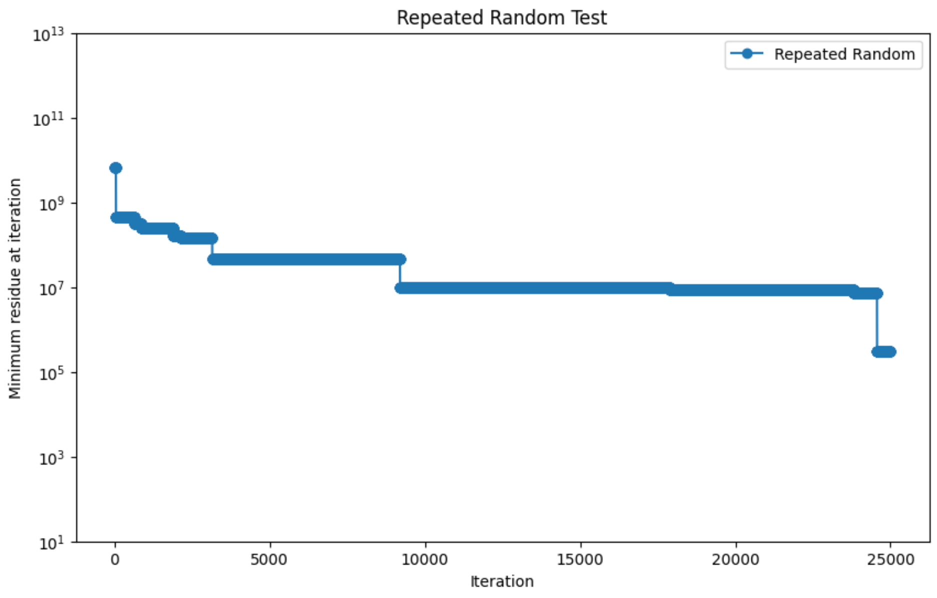
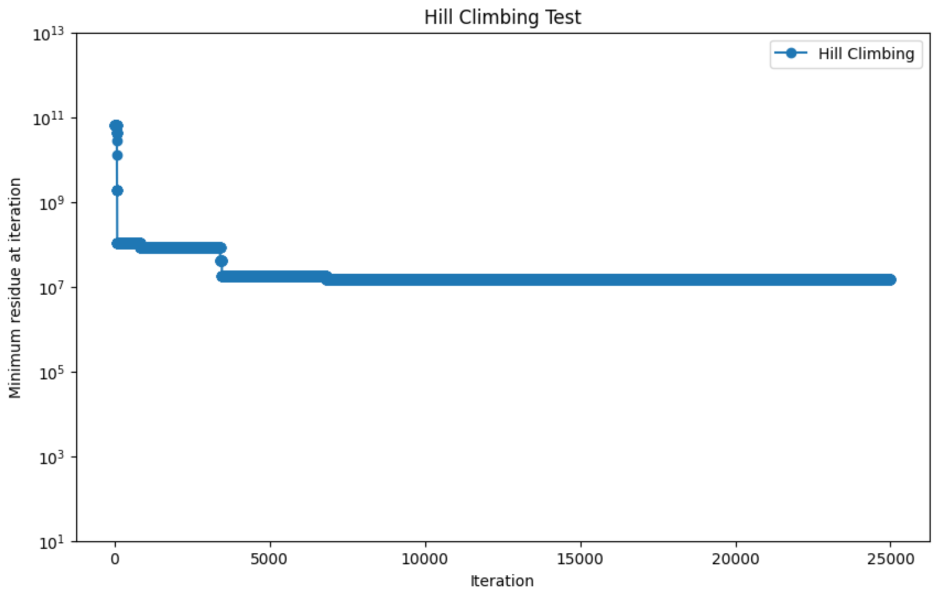


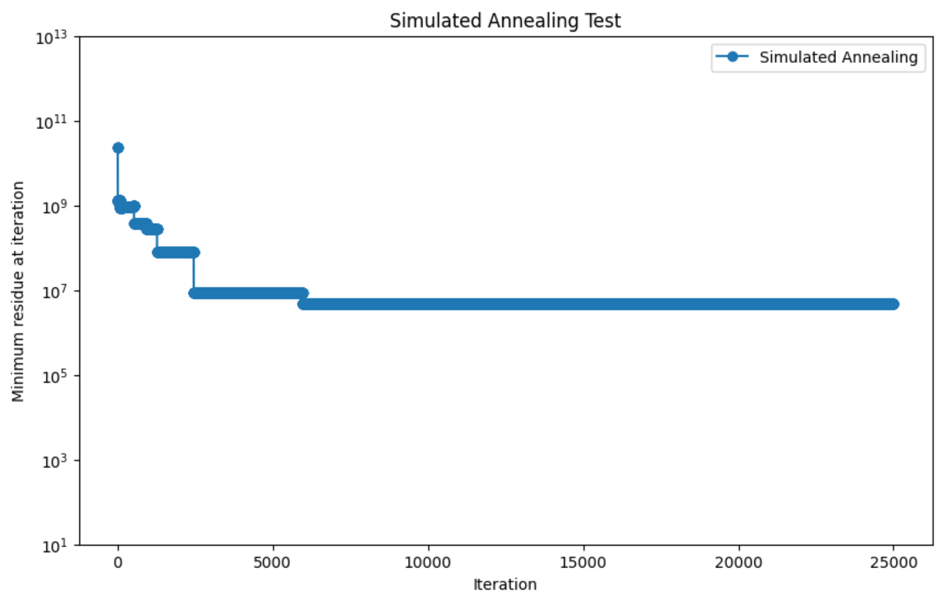
Figure 2: Enter Caption

Analysis: We notice that without prepartitioning, the heuristics we ran (repeated random, hill climbing and simulated annealing) performed very similar as they were tend to be on the same magnitude of 10. However, all of them performed significantly worse than the Karmarkar-karp algorithm, which is consistent with our previous analysis. We get a good understanding of what the upper bound of the solution could be through Karmarkar-karp algorithm. So randomly assigning our subsets (S_1 and S_2) through -1, and 1 leads to a slower stepping algorithm that takes long to converge to the optimal/minimal residue. We will we see this very shortly.
Thus, when we prepartion our arrays and then run our RR, HC, and SA algorithms again with a redefinition of states we see these algorithms start to out perform Karmarkar-karp and their non-pre-partitioned variants. Of course, this is no surprise since these pre-partitioned heuristics are very similar to KK since they no longer have the number of sets as just 2, but instead len(array). This empirically appeared to be an optimization as we can take a closer look to our convergence.

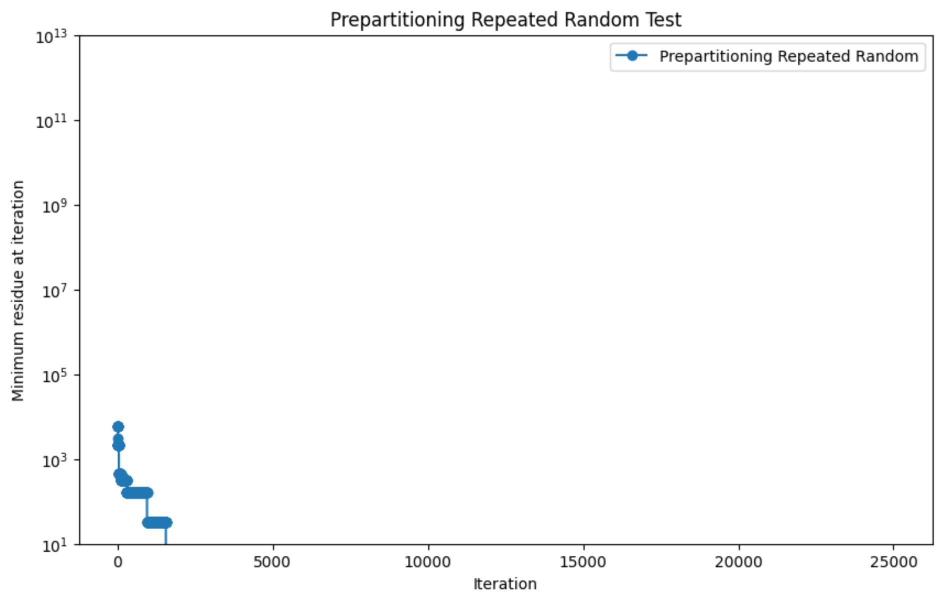


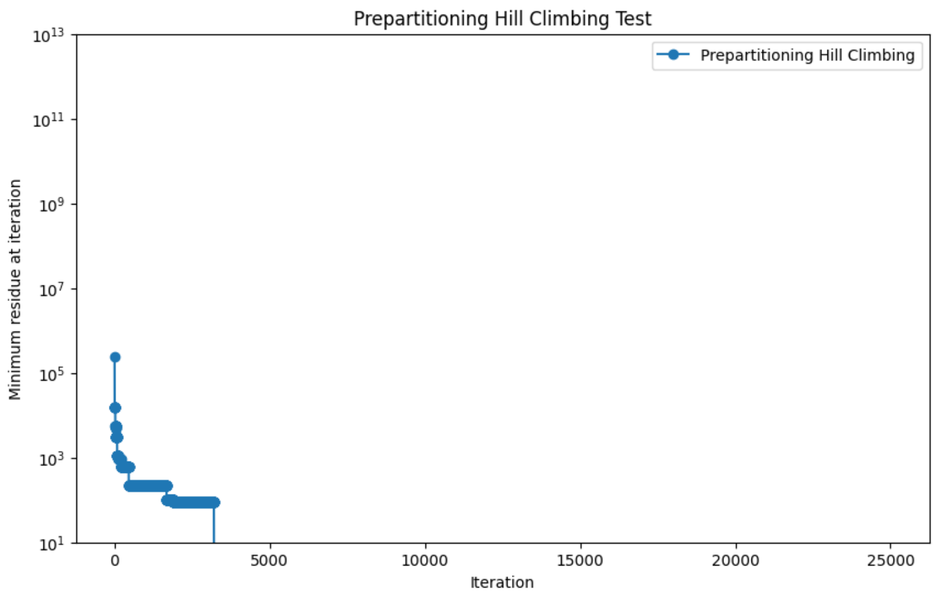


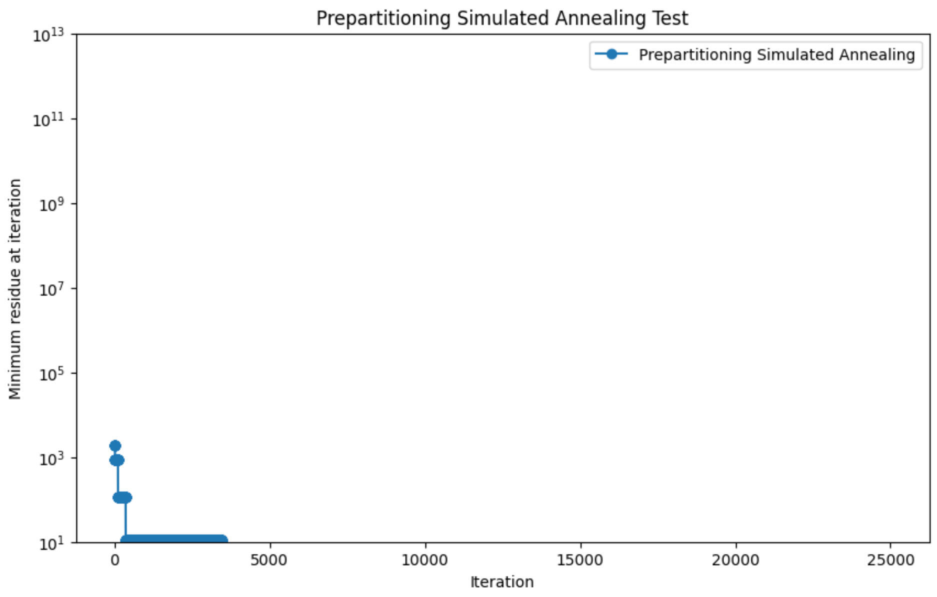
Analysis: When looking at the standard variation of RR and Hill climbing, we see the function of decreasing minimum residue as iteration increases are similar in shape. Despite their similarity, we can see that our Hill Climbing algorithm is stuck at a local minimum since we see the repeated random algorithm find some smaller value and actual improve while Hill Climb is unable to improve its solution. This is the issue with hill climbing since it will get stuck depending on the starting state, which inhibits its ability to perform well in certain cases.



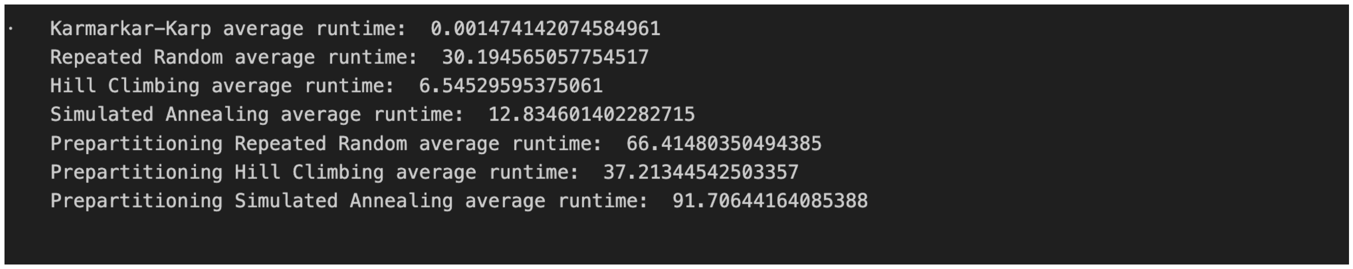
Analysis: We see the same issue as hill climbing in the simulated annealing heuristic as it is almost the same, except we flip some probability to actually maybe go to some other state in our algorithm. We don't much of a difference in this scenario.







Analysis: While the general trends were described earlier about the algorithms themselves, when we prepartition we notice such a faster convergence to some minimal value of residue. This might be due to the fact that there exists greater change from one state to another, which might lead to this convergence a lot faster for the preparititioned algorithms. Additionally, these algorithms improve on the already upper bound found by the starting point of the Karmarkar-karp algorithm. Thus, we get these results.



```
·   Karmarkar-Karp average runtime:  0.001474142074584961
    Repeated Random average runtime:  30.194565057754517
    Hill Climbing average runtime:  6.54529595375061
    Simulated Annealing average runtime:  12.834601402282715
    Prepartitioning Repeated Random average runtime:  66.41480350494385
    Prepartitioning Hill Climbing average runtime:  37.21344542503357
    Prepartitioning Simulated Annealing average runtime:  91.70644164085388
```

Analysis: These were our runtimes for our algorithms. Clearly we have Karmarkar-Karp algorithm as the clear fastest runtime simply because only run it once with $O(n \log(n)$ described earlier. Meanwhile, for the non-prepartitioned algorithms we see that Hill climbing average to be the smallest, then Simulated annealing, and then the most time costly being Repeated random. These of course are implementation specific, but this is most likely because creating a random state requires me to re-iterate through the array just to create the subsets corresponding to $S = \{-1, 1\}$. In both prepartitioning and non-prepartitioning, Hill climbing average runtime was the determined to be the lowest. Furthermore, prepartitioning simulated annealing average runtime was the largest, as it only it also has to compute a probability function everytime there is a worse solution found.

For the simulated annealing algorithm, you must choose a *cooling schedule.* That is, you must choose a function T(iter). We suggest $\text{T(iter)} = 10^{10}(0.8)^{\lfloor \text{iter}/300 \rfloor}$ for numbers in the range $[1, 10^{12}]$, but you can experiment with this as you please.

Note that, in our random experiments, we began with a random initial starting point.

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.)

Even without a randomized prepartitioning, as we saw in the concrete examples provided to us above, the Karmarkar-Karp (KK) algorithm itself does some implicit partitioning. At every iteration of KK, when we take the largest two and do our subtraction, we are implicitly partitioning into two groups. Thus, this gives us some idea of how we could start to estimate a split and calculate residue. For example, we expect the largest pair of numbers to be split up so as not to make one group contain all the biggest numbers.

I thought of a pretty funny story to explain why this is a decently reliable heuristic. Suppose that we are at recess and we want to make two teams to play a game. We want to make the teams as even as possible. We should quantify the skill of all the players and sort them into groups so that the sum of the skill of the two teams has a minimal difference.

We should line all of the players up from most skilled to least skilled. In this line, we tell all the players to find a pair-partner who is right next to them in line. Each player has a similar skill level to their partner relative to other players. The logic of the KK algorithm essentially iterates through all of the pairs and splits the partners up. Intuitively, you make sure that each team gets one of each pair, who is similar in skill to the partner assigned to the opponent. In the end, we end up with pretty evenly matched teams.

Finally, the following is entirely optional; you'll get no credit for it. But if you want to try something else, it's interesting to do.

Optional: Can you design a BubbleSearch-based heuristic for this problem? The Karmarkar-Karp algorithm greedily takes the top two items at each step, takes their difference, and adds that difference back into the list of numbers. A BubbleSearch variant would not necessarily take the top two items in the list, but probabilistically take two items close to the top. (For instance, you might “flip coins” until the the first heads; the number of flips (modulo the number of items) gives your first item. Then do the same, starting from where you left off, to obtain the second item. Once you’re down to a small number of numbers – five to ten – you might want to switch back to the standard Karmarkar-Karp algorithm.) Unlike the original Karmarkar-Karp algorithm, you can repeat this algorithm multiple times and get different answers, much like the Repeated Random algorithm you tried for the assignment. Test your BubbleSearch algorithm against the other algorithms you have tried. How does it compare?