CS 124 Programming Assignment 3: Spring 2023

Your name(s) (up to two):

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 2023-04-20 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

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.

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:

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

$$\rightarrow \quad (0,0,0,0,2)$$

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.

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:

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

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.

For the simulated annealing algorithm, you must choose a *cooling schedule.* That is, you must choose a function T(iter). We suggest $T(\text{iter}) = 10^{10}(0.8)^{\lfloor \text{iter}/300 \rfloor}$ 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.)

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?

Number Partition problem dynamic programming solution

Let's call the sum of all our elements in A as b and the number of elements in A as n. We begin by initializing a 2d array dp of 0s that is $\frac{b}{2}+1$ by $n+1$, where $dp[i][j]$ represents whether there's a subset of A up to the $j-1$th element that sums to i. We can start by looking at the case where $i = 0$, meaning the sum we are looking for is 0. We know that when we have an empty subset, we have a sum of 0, so we can set $dp[0][0]$ to 1. We also know that if there exists a subset of A up to $j-1$th element that sums to i, then there exists a subset of A up to jth, $j+1$th, ..., $n-1$th element that sums to i since it can simply be made up of the subset we found up to the $j-1$th element. Thus, we can set $dp[0][j]$ to 1 for all j up to $dp[0][n]$, since the empty subset will always be a part of every subset. We can follow this rule for the rest of the 2d array. We can also go ahead and set $dp[i][0]$ for all $i \neq 0$ to 0 since the empty subset will never sum to anything but 0. Now, we want to iterate from $i = 1$ to $i = \frac{b}{2}$, and within each iteration, we will iterate through $j = 1$ to $j = n$. At each $dp[i][j]$, we will check the condition mentioned previously, which is if $dp[i][j-1] = 1$ then $dp[i][j] = 1$. If this condition isn't satisfied, then we will move onto the second condition. For this condition, if these indices are within bounds, we want to check if $dp[i - a_{j-1}][j-1] = 1$, and if so, then $dp[i][j] = 1$. We know that if there's a subset that can sum to $i - a_{j-1}$ up to element $j-2$, then there's a subset that can sum to i up to element $j-1$ since it would be the same subset but including a_{j-1}. After dp is fully filled out, we can look at the i which is the highest value for which $dp[i][n] = 1$ since this represents the highest sum possible up to $\frac{b}{2}$ that a subset of all n elements of A could sum to. This means that all the elements not in that subset would sum to $b-i$. Thus, the residue found would be $(b-i)-i$, which is just $b-2i$.

Proof of Correctness:
First, we prove that the invariant $Dp[i][j]$ is 1 if and only if we can generate a sum of i using the first j elements through induction.

Base Case:
We start with $Dp[i][0]$ which asks if we can generate a sum of i using 0 elements or the empty set. We know that the only sum that we can make here is 0 so $Dp[0][0] = 1$ is set. All other $Dp[i][0] = 0$ as we cannot make any other i with the emptyset.

Inductive Step:
Assume by induction that row $Dp[i][j]$ is set correctly, for that i, and we want to show that row $Dp[i][j+1]$ is also set correctly. We must show both directions of our if and only if statement in order to ensure the induction is true.

Direction 1: if $Dp[i][j+1] = 1$ then we know that it is possible to get a sum of i using the first $j+1$ elements. Since we have now set $Dp[i][j+1] = 1$, we know that either one of two conditions exist: $Dp[i][j] = 1$ or $Dp[i - x_{j+1}][j] = 1$. Now, we can adopt casework here:
Case 1: $Dp[i][j] = 1$.
By our inductive hypothesis, there is some subset, which we will call S $\subseteq [x_1, x_2, ...x_n]$ that sums to i. We can use the same set S to show that we have found a set that uses the first $j+1$ elements that sums to i.
Case 2: $Dp[i - x_{j+1}][j] = 1$
By our inductive hypothesis, there is some subset, which we will call S $\subseteq [x_1, x_2, ...x_n]$, that sums to $i - x_{j+1}$. Let's look at a new set, $S' \subseteq Sx_i$. This sum will sum to i using the first $j+1$ elements so we have found a set that uses the first $j+1$ elements that sums to i.
This means that for all cases, if $Dp[i][j+1] = 1$, then it is possible to get a sum of i using the first $j+1$ elements.

Direction 2: If we can get a sum of i using the first $j+1$ elements, this must mean that $Dp[i][j+1] = 1$. First, we will assume it is possible to get a sum of i using the first $j+1$ elements. Therefore, we know that there exists some $S \subseteq [x_1, x_2, ...x_n]$ which sums to i. Now, we have two cases of whether or not $x_{j+1} \in S$.
Case 1: $x_{j+1} \in S$
Here, we consider the set without the element $S - x_{j+1}$. This set, which we will call S', sums to $j - x_{j+1}$. Using our inductive hypothesis, $Dp[i - x_{j+1}][j] = 1$ and by our algorithm, $Dp[i][j+1] = 1$.
Case 2: $x_{j+1} \notin S$
We can still see that $S \subseteq [x_1, x_2, ...x_n]$ and it sums to i. Using our inductive hypothesis here, $Dp[i][j] = 1$ and by our algorithm, $Dp[i][j+1] = 1$.

Using our base case and looking at both directions that the inductive step covers, we have proven, through induction, that $D[i][j]$ is 1 if and only if we can generate a sum of i using the first j elements. Now, we have to show that we indeed extract the true solution from our dynamic programming array. We will say that $T(S)$ denotes the sum of the elements in S. Now, we want a partition that generates S and its complement S^c that will minimize the difference of their sums. We will let $T(S) \leq T(S^c)$, so $T(S) \leq \lfloor T/2 \rfloor$. Now, row $D[n]$ displays all possible sums up to $\lfloor T/2 \rfloor$ using all of our n elements. Since we want the highest j for $D[i][j] = 1$ to minimize the difference of $T(S)$ and $T(S^c)$ and since we also know that $T(S^c) = T - T(S)$, we are left with an answer of $T - 2j'$

Runtime:
First, we create our dynamic programming matrix D which has dimension of b by n. To fill each entry $D[i][j]$, we perform an addition. If we assume that addition takes $O(1)$ time, we know that generating D takes time $O(n \cdot b)$, however, if we don't assume constant time for addition, each addition takes $O(log(b))$ time so we are left with a runtime of $O(n \cdot b \cdot log(b))$ to generate our matrix. Once we have filled the matrix, we scan for the first instance of 1 and do a subtraction. Scanning takes time $O(b)$ and subtraction takes time $O(1)$ under the assumption of constant time or $O(log(b))$ if not. Then, we return our solution. Overall, if we assume that addition and subtraction takes constant time, we are left with a runtime of $O(n \cdot b)$ otherwise, we are left with a runtime of $O(n \cdot b \cdot log(b))$.

Karmarker-Karp algorithm implementation in O(n log(n))

We can do implement this algorithm quite easily. We will use a max heap to store all the elements within our list. This takes time $O(n \cdot log(n))$ as the insert function in heaps takes $O(log(n))$. Next, we can access each element for differencing by using the extract max function which will take $O(log(n))$ time. Since there are a total of n pairs that we will need to difference at most, we must perform the extract max function at most 2n times which will give us a runtime of $O(2n \cdot log(n))$ for differencing. To access the residue, we can just extract max from our heap after performing all operations. Thus, the total time will be $O(n \cdot log(n) + (2n+1) \cdot log(n)) = O(n \cdot log(n))$.

Karmarker-Karp algorithm for randomized algorithms

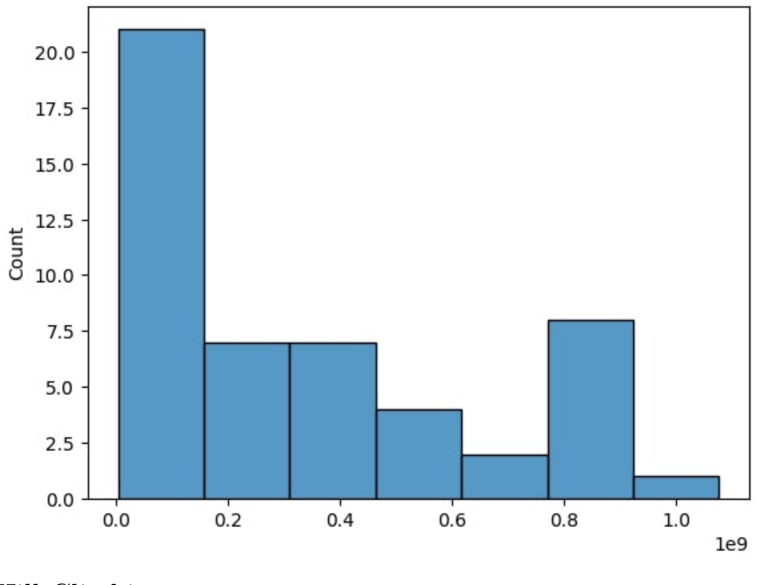
We could use the KK algorithm to create a more optimal starting point for our randomized algorithms. If we use the KK algorithm for randomized algorithms, we are essentially starting at a better starting point. We can view the list of possible residues as a hill. By using the KK algorithm we are starting at a point that is higher on the hill as we are subtracting the largest elements at a time. This means that on average, our randomized algorithms will return smaller residues as they start with a better starting residue and only update when an even better residue is discovered.

Graphing Stuff

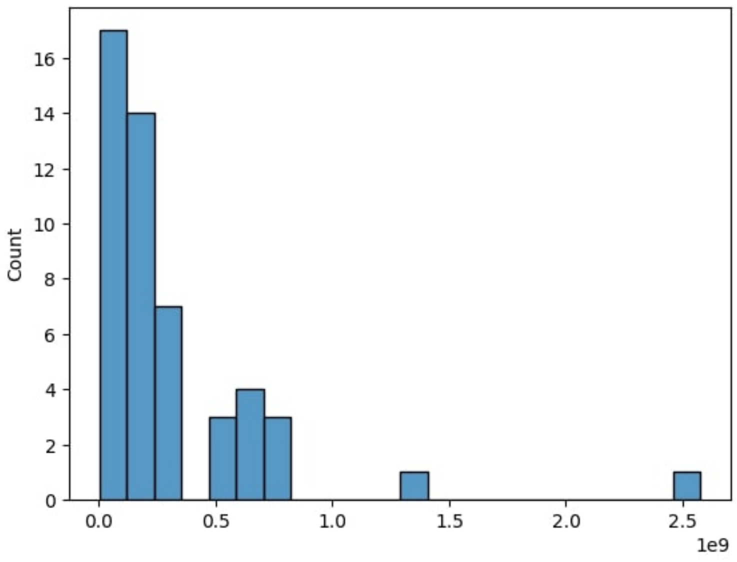
We can see that the KK algorithm and the prepartitioned algorithms gave significantly better residues than the other algorithms. Moreover, the prepartitioned algorithms were better at coming up with lower residue solutions that the KK algorithms. Within the specific prepartitioned algorithms, the random and annealing algorithms had better residues than hillclimb algorithm. The following graphs support the conclusions made above.

We see that KK is the fastest by far, meanwhile the prepartitioned versions of the random, hillclimbing, and simulated annealing algorithms take longer than their non-prepartitioned counterparts.

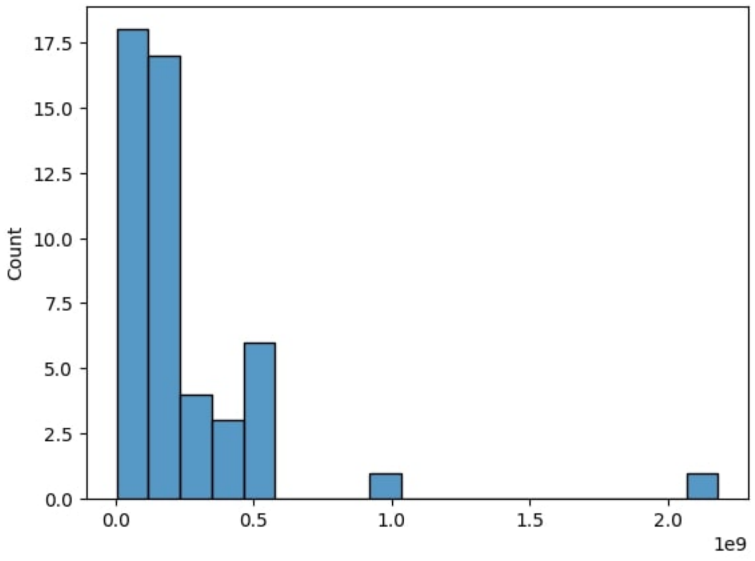
Repeated Random:



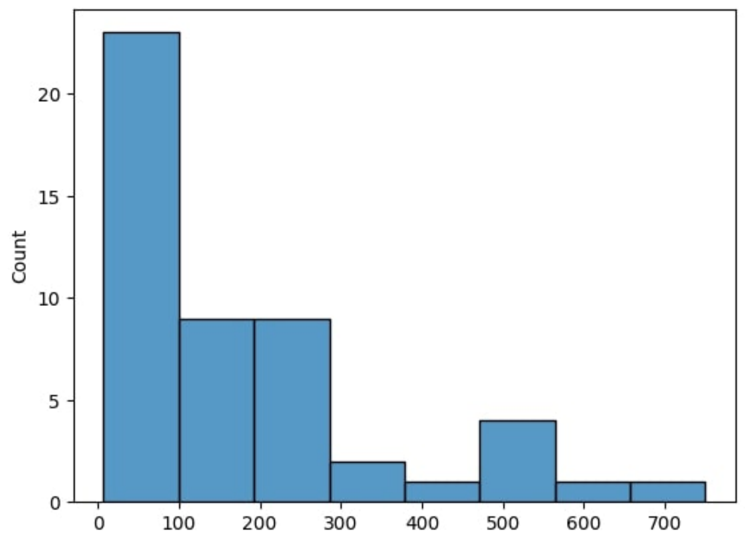
Hill Climbing:



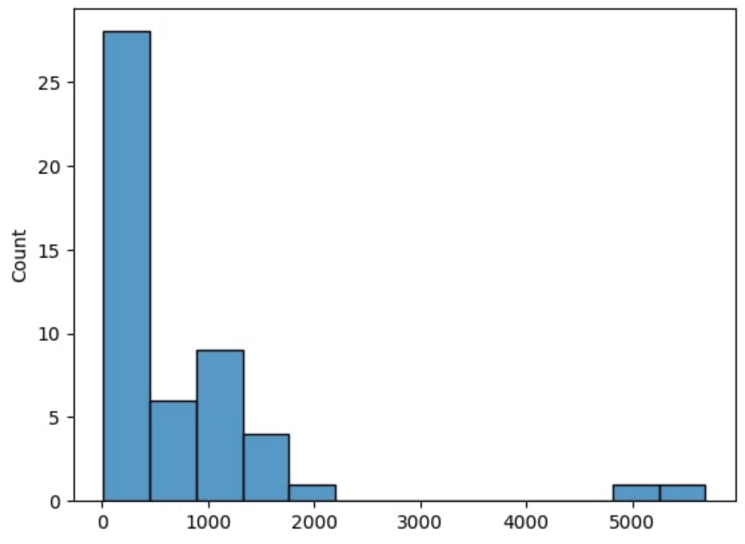
Simulated Annealing:



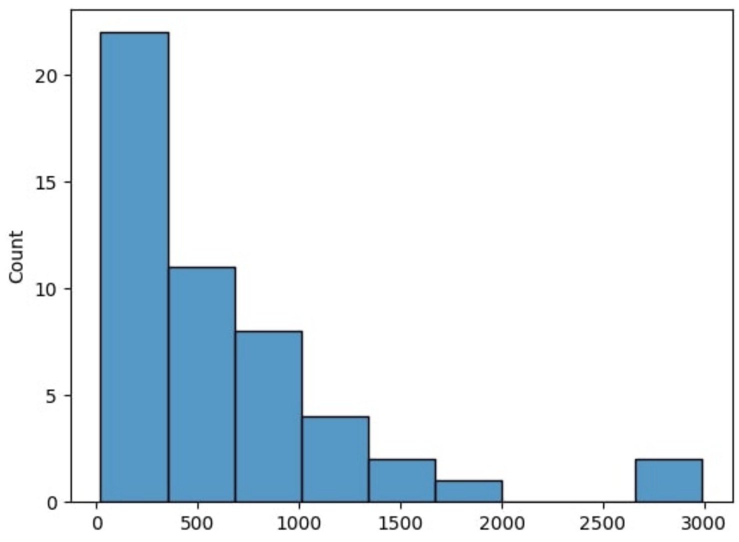
Repeated Random(prepartition):



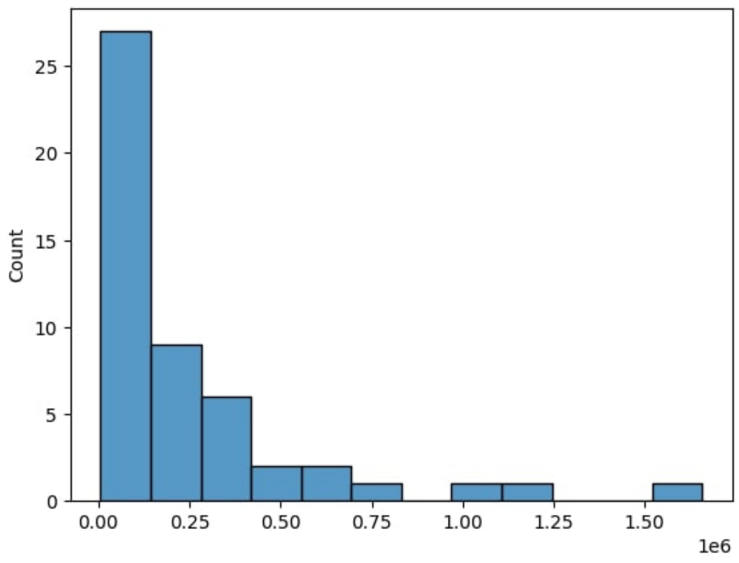
Hill Climbing(prepartition):



Simulated Annealing(prepartition):



Karmarker-Karp Algorithm


Count
1e6