Programming Assignment 3

Methods

The implementation for this assignment was fairly straightforward, with less runtime optimizations and more so making sure processes ran accordingly. We utilized C++ 17 for this assignment.

The first task was to implement Karmarkar-Karp. We chose not to implement an asymptotically efficient version of Karmarkar-Karp that would involve Max-Heaps (detailed below), as writing the methods for heaps would cause significant overhead. Rather, we took a longer but iterative approach: we obtain the two maximums by iterating twice over the array, subtracting them and resubstituting the result, and repeating. While this would take longer time in the long run, we found the overall runtimes to be within reason.

Next, we implemented the Repeated Random, Hill Climbing, and Simulated Annealing algorithms with and without without prepartitioning of solutions. Each algorithm was tested on 100 trials of 100 random signed 64-bit integers with 25,000 iterations in each trial. Signed 64-bit integers were used everywhere in order to ensure no overflow.

For Repeated Random without prepartitioning, we generate random values of -1 and 1 with equal probability, representing a solution S, and computed the residual and only updating if another trial generated a smaller residual. With prepartitioning, we instead generate random partition sequences P using integers from [1..100], run Karmarkar-Karp, and then only accept the solution if another trial generated a smaller residual. For Hill Climbing without prepartitioning, we begin with a random P, compute its residual using Karmarker-Karp, and pick a neighbor defined in the problem statement, again only updating if another trial generated a smaller residual. With prepartitioning, we do the same thing except pick a neighbor partition sequence P, defined exactly in the problem statement. And for Simulated Annealing, we again choose a neighbor partition sequence and compute using Karmarker-Karp. However, even if the residual is no longer greater than the best seen residual, we still "keep" the residual with some probability defined by the cooldown, but only update the best residual if a lower residual is seen. The cooldown used is the one provided in the problem statement.

Discussion

The dynamic programming solution to the Number Partition problem relies on a fundamental observation: that the best possible sum for at least one of the partitioned groups will be the achievable value closest to $\lfloor \frac{b}{2} \rfloor$), where b is the sum of the numbers in the original list, simply because in this way the difference will be closest to zero. This problem then reduces to the problem of finding a group of numbers that sums as close as possible to a given value (in this case $\lfloor \frac{b}{2} \rfloor$), among a list of numbers. We define our objective as a matrix $D[j][k]$, where j ranges from $[1..n]$ and k ranges from $[1..\lfloor \frac{b}{2} \rfloor]$, which should hold the maximum possible achievable sum using numbers up to j without the total going over k. Our recursive relationship is then the following, where $a[j]$ refers to the value at index j:

$$D(i) = \max \begin{cases} D[j-1][k] \\ D[j-1][k-a[j]] + a[j] \end{cases} \tag{1}$$

The pseudocode is as follows:

Algorithm 1 Number Partition

```
Require: List $a$
Ensure: Sequence of signs $s$
 1: $Total \leftarrow \textsc{sum}(a)$
 2: $n \leftarrow \textsc{len}(a)$
 3: $b \leftarrow \lfloor \frac{Total}{2} \rfloor$
 4: $D = [n][b]$
 5: $C \leftarrow \textsc{Ones}([n][b][n])$
 6: for $j \in [1..n]$ do
 7:     $D[j][0] = 0$
 8: for $k \in [1..b]$ do
 9:     $D[0][k] = 0$
10: for $j \in [1..n]$ do
11:     for $k \in [1..b]$ do
12:         if $a[j] > k$ then
13:             $D[j][k] = D[j-1][k]$
14:             $C[j][k] = C[j-1][k]$
15:         else
16:             $D[j][k] = \max(D[j-1][k-a[j]] + a[j], D[j-1][k])$
17:             if $D[i][j][k] = D[i][j-1][k]$ then
18:                 $C[j][k] = C[j-1][k]$
19:             else
20:                 $C[j][k] = C[j-1][k. - a[j]]$
21:                 $C[j][k][j] = -1$
22: $s \leftarrow N[n][b]$
23: return $s$
```

For each possible value of n and b, we consider whether or not we should add some element j (partitioned by index) into a group of integers that will add up to k. We compare the maximum sum achievable (without exceeding the total) by either including j or not including j. In this way, we build bottom-up the best possible combinations of integers to achieve any given sum lower than what is currently being iterated through, and at the end we simply look for the last element, which by definition must be the best possible combination of integers to achieve the value closest to $\lfloor \frac{b}{2} \rfloor$.

The runtime of this algorithm is polynomial in nb, where n is the total Count and b is the floor of the total sum divided by 2; more specifically, $O(nb)$, since we loop to b n times, doing a constant amount of work per loop. The space complexity is also polynomial in nb; more specifically, $O(n^2b)$, in order to store the arrays that hold the sequences that comprise the solution to each possible sum. We hold one for each value up to nb, and since each sum can have at most n elements (ie. all), the space complexity is $O(n^2b)$.

Instead, if we turn to Karmarkar-Karp, we can implement this in $O(nlog(n))$ steps. First, we add all items to a max-heap, taking $O(n)$ steps. Then, we remove the maximum twice, subtract the two numbers, and reinsert into the heap, taking a total of $O(3log(n))$ steps. We do this until one number remains (ie. the residue), producing a total asymptotic step count of $O(nlog(n))$.

Algorithm	Average Residual	Average Time (ms)
Karmarkar-Karp	216999.84	0.11688
Repeated Random	313540714	379.14142
Hill Climbing	290156716.3	132.67351
Simulated Annealing	278932148.3	132.6967
PP Repeated Random	135.08	2895.33967
PP Hill Climbing	852.48	2642.38362
PP Simulated Annealing	155.68	2674.61829

Table 1: Average residuals and times for each algorithm using 25000 iterations

Examining the results of the simulation methods detailed in the Methods, we can immediately see that the prepartitioned algorithms run much more reliably (ie. is able to produce a residual value closer to 0) than without prepartitioning, and still performs significantly better than Karmarkar-Karp alone; however, this appears to come at the cost of runtime. Instead of generating random integers of -1 and 1 100 times as a sample solution S or switching two index values as a neighbor of S, we instead have to generate random and neighboring P sequences, convert this solution to the relevant solution space, and run Karmarkar-Karp - thus all adding signficantly to the runtime (with Karmarkar-Karp adding the most). But the effect of this is extremely apparent. As Karmarkar-Karp is already significantly more efficient than any of the non-prepartitioned algorithms, it makes sense that prepartitioning would improve upon this. By loosely grouping certain integers together, we iteratively improve upon the best possible input P that Karmarkar-Karp can then use to produce the residual of. In doing so, we allow Karmarkar-Karp to calculate the best possible residual given an P, instead of relying on a pre-grouped sequence of integers S by which we cannot optimize any further (and can only sum to get the residual).

We were also interested in the distributions of the simulations. In terms of the residuals, we see that the spread does not deivate significantly; the lower and upper quartile appear to be a factor of roughly 0.75 orders of magnitude apart. There is considerable difference between the largest and smallest residuals computed for each method, but we see that even the largest residual produced by the prepartitioned algorithms is still significantly smaller than the smallest residual of the non-prepartitioned algorithms. The spread of Karmarker-

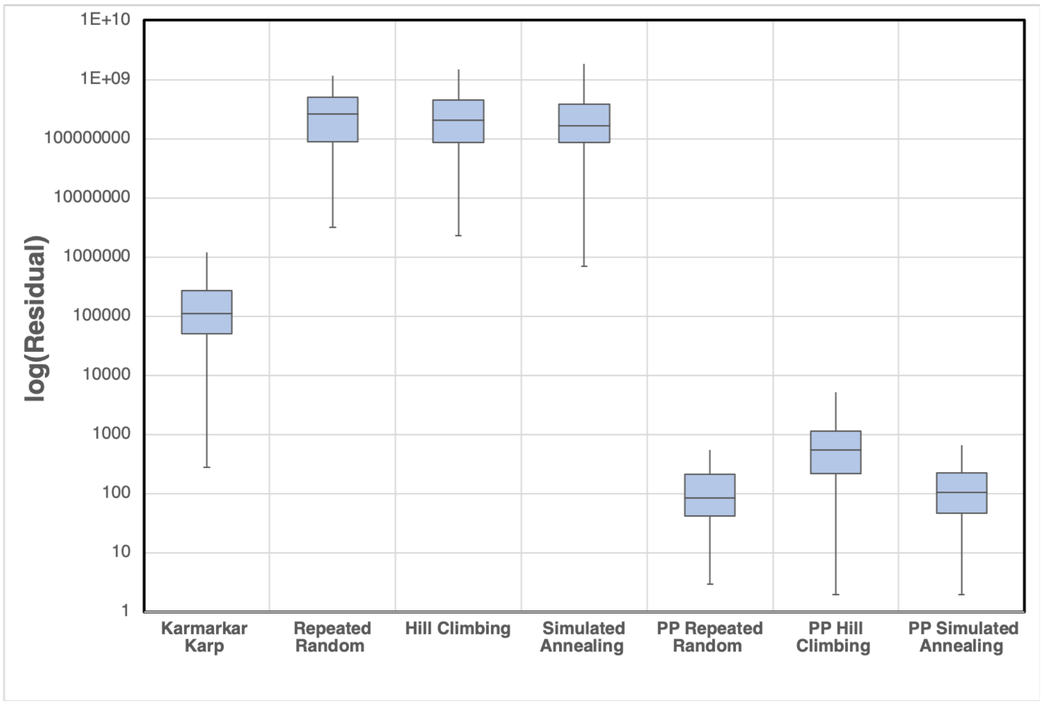

Figure 1: Residuals for each algorithm

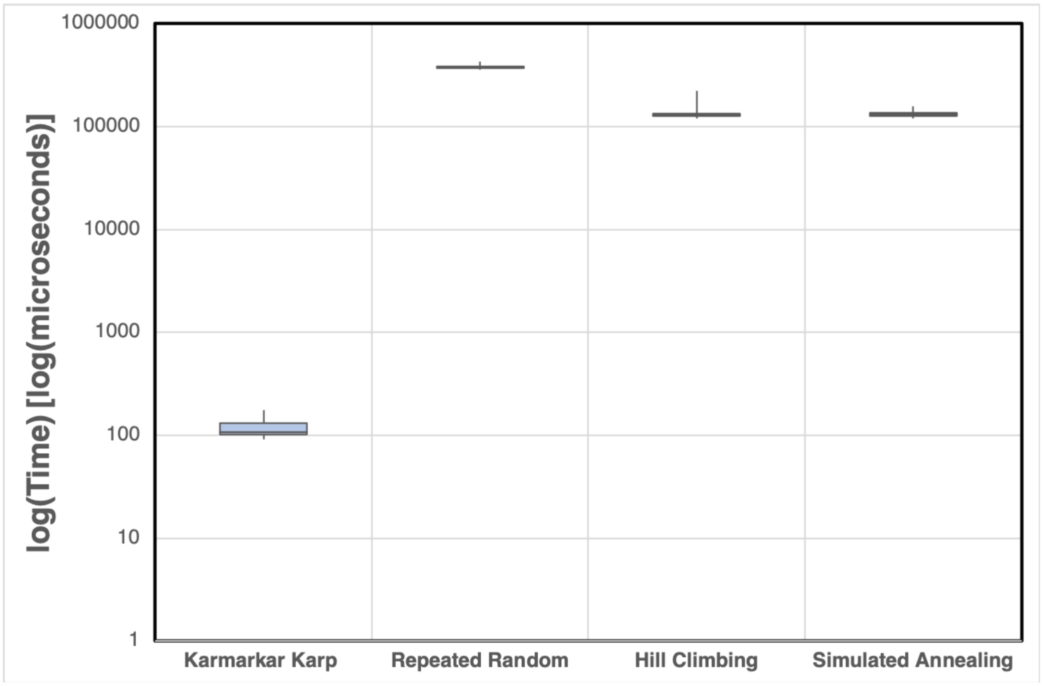

Figure 2: Log(Runtime) for Karmarkar-Karp, Repeated Random, Hill Climbing, and Simualted Annealing algorithms

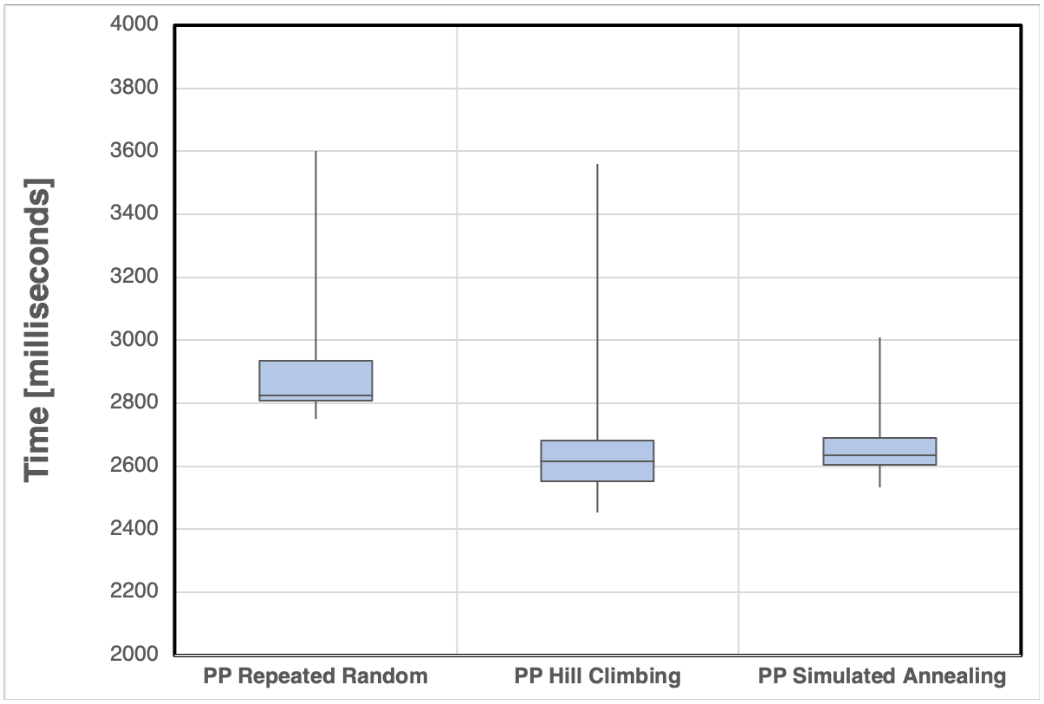

Figure 3: Runtime for prepartitioned algorithms

Karp seems to straddle the difference between these two categories. On average, though, it appears that all non-prepartitioned algorithms perform equally badly. It also appears that prepartitioned Repeated Random or Simulated Annealing would be the most efficient overall. This would demonstrate the utility of not simply traversing by neighbors. The Repeated Random algorithm generates random sequences without consideration of its neighbors, preventing it from being caught in local minima, but also preventing it from closely reaching global minima unless the iteration count is significant. The Simulated Annealing algorithm does traverse through its most optimal neighbors but also traverses non-optimal neighbors with some probability defined by the cooldown function. This function allows the algorithm to explore significantly beyond its own neighbors at low iteration count (when the probability of choosing a non-optimal neighbor is high, and there is time to explore), but then narrows the search space to its neighbors at high iteration count (when the probability of choosing a non-optimal neighbor is now low) - thus more likely leading towards the global minimum.

In contrast to the spread of the residuals, we see that the spread of the times is significantly less. This makes sense, as we anticipate all the numbers to be fairly large and for the runtime to not depend significantly on the differences in magnitude of these numbers. We see that Karmarkar-Karp is the lowest (obviously), but aside from that non-prepartitioned Hill Climbing and Simulated Annealing run the fastest. This is only slightly faster than prepartitioned Repeated Random, simply because we do not have to generate an entirely new random solution S each time. The prepartitioned Repeated Random appeared to run the slowest, likely due to the aforementioned reasons and that it is the only one of the prepartitioned algorithms that requires the generation of entirely new random solution P.

As stated before, the benefits of utilizing Karmarkar-Karp are obvious. Were we to instead begin with a Karmarkar-Karp solution, we would expect that the average residual drop down to at least the Karmarkar-Karp solution. However, a Karmarkar-Karp solution is not necessarily indicative of the global minimum, and so even with this addition we don't expect to obtain residuals significantly close to the prepartitioned algorithms, since the same problem arises as stated previously. We expect the same overall trend to hold across the algorithms, in terms of residual magnitudes, with Repeated Random still being the highest. For the prepartitioned algorithms, we don't expect that beginning with a Karmarkar-Karp solution would benefit the average residual significantly. The algorithms already utilize Karmarkar-Karp to a significant degree, and from Table 1, we see that the residual is already extremely low.

With respect to code design, we noticed and corrected using two interesting phenomena during development. We initially seeded the random number generator multiple times, believing that by default the generator would seed using the clock time. However, this turned out to not be the case; we were essentially reseeding the generator to the same starting point every time we called on the function to generate a random S or P sequence, thus breaking our Repeated Random algorithms that rely most heavily on generating randomized sequences. We also noticed that using the default random engine was not sufficient in producing high-quality random numbers and residuals varied extremely widely; instead, we utilized mt19937_64, the 64-bit version of the Mersenne Twister pseudorandom number generator.

References

[1] C++ 17 Docs.

[2] Computer Science 124, Harvard University.