CS 124 Programming Assignment 3
Harvard IDs: 10939860, 80977824
April 21, 2017

Dynamic Programing Solution

Our dynamic programming solution is inspired by the solution discussed during section.

Definition:
Let $A = (a_1, a_2, \ldots a_n)$, where A is the given sequence of nonnegative integers. Let t be the sum of all the elements in A, which can be represented as $\sum_{i=1}^{n} a_i$. In this problem, we want to divide A into two sets A_1 and A_2 , where the sum of each set's elements is as close to half of t as possible. This is because if we want the residue to be as close to 0 as possible, we want each set to have a sum as close to half of the overall sum t as possible. This is similar to the subset sum problem (which the number partition problem reduces to), where in this case the sum we are finding is equal to half t. For the definition of our dynamic programming solution, we let X[i, $\lfloor \frac{t}{2} \rfloor$] be a boolean value, where it is True if there exists a subset of the first i numbers of A that add up to $\lfloor \frac{t}{2} \rfloor$, and False if not.

Recurrence:

$$T(n) = \begin{cases} True & \text{if } i = 0, \lfloor \frac{t}{2} \rfloor = 0 \\ False & \text{if } i = 0, \lfloor \frac{t}{2} \rfloor \neq 0 \\ False & \text{if } \lfloor \frac{t}{2} \rfloor < 0 \\ X[i-1, \lfloor \frac{t}{2} \rfloor - a_i] \| X[i-1, \lfloor \frac{t}{2} \rfloor] & \text{otherwise} \end{cases}$$

For an element a_i, we have to determine if we want to include it in set A_1 or A_2. If we include it in set A_1 , we evaluate X[i-1, $\lfloor \frac{t}{2} \rfloor$ -a_i]. If we include a_i in set A_2, we evaluate X[i-1, $\lfloor \frac{t}{2} \rfloor$]. This is our recurrence. However, we also must construct a table Y[i,s] to determine the actual partitionings. Y[i,s] is None if X[i,s] is False. If X[i,s] is True and X[i-1, $\lfloor \frac{t}{2} \rfloor$ -a_i] is also True, then Y[i,s] stores "A_1". If X[i-1, $\lfloor \frac{t}{2} \rfloor$ -a_i] is False and X[i-1, $\lfloor \frac{t}{2} \rfloor$] is True, then Y[i,s] stores "A_2". Finally, we evaluate X[n,b] to see if it is True. If not, we move on to X[n,b-1], and then on to X[n,b-2], and then so on until we find the first instance where it evaluates to True. We can generalize this, where X[n,b-c] is the first instance of a True value. We then check the second table Y at Y[n,b-c]. There are two possibilities of what it will store: "A_1" or "A_2". If Y[n,b-c] contains "A_1", we place a_n into set A_1 and recurse on X[i-1, $\lfloor \frac{t}{2} \rfloor$ -a_i]. However, if [n,b-c] contains "A_2", we place a_n into set A_2 and recurse on

$X[i-1, \lfloor \frac{t}{2} \rfloor]$. At the end, we should have two sets A_1 and A_2, each with elements that sum as close to *t* as possible, making the residue as close to 0 as possible.

Time and Space Complexity:
The time complexity is O(nb) since there are O(nb) possible inputs to X and each input takes O(1) time to calculate. Furthermore, the space complexity is also O(nb) since our final solution requires the use of both X and Y.

Karmarkar-Karp Runtime Question

The Karmarkarp-Karp algorithm can be implemented in O(nlogn) time by using a max-heap. We will describe the operations in one iteration of the algorithm. We begin my building a max heap from the array A which takes O(nlogn) time. However, this is not more significant than the run time of the following steps which take O(nlogn) in total. The first step of Karmarkar-Karp is removing the max element which takes O(1) time. This rebalances the heap with MAX-HEAPIFY which takes O(logn) time. We remove another max element which takes O(1) time, which rebalances the heap again with MAX-HEAPIFY, taking O(logn) time. We then take the difference of those two elements, which is an arithmetic operation taking O(1) time. In total, one iteration of Karmarkar-Karp takes O(2logn) time. We do *n* iterations of this, where the runtimes can be represented by 2log(n) + 2log(n-1) + 2log(n-2)...2log(1). This sequence is equivalent to $O(\frac{2[log(n) + log(1)]n}{2})$ which simplifies to O(nlogn).

Results

Our *kk.py* file runs the Karmarkar-Karp algorithm on an input file of 100 random numbers, as well as the repeated random, hill climbing, and simulated annealing algorithms for both the standard and prepartition representations (which are written in *standard_solution.py* and *prepartition_solution.py*), for seven results in all. The file prints the residue and runtime for each algorithm. (Note: our file is compiled with "chmod +x kk.py" and run with "./kk.py <inputfile>".)

The attached Excel spreadsheet, *CS124_PA3_Results.xlsx*, contains the residues and runtimes for the seven methods, for 100 random instances of 100 random numbers on the interval $[0,10^{12}]$. For simplicity, here are the averages of 100 trials for each method:

Residues:

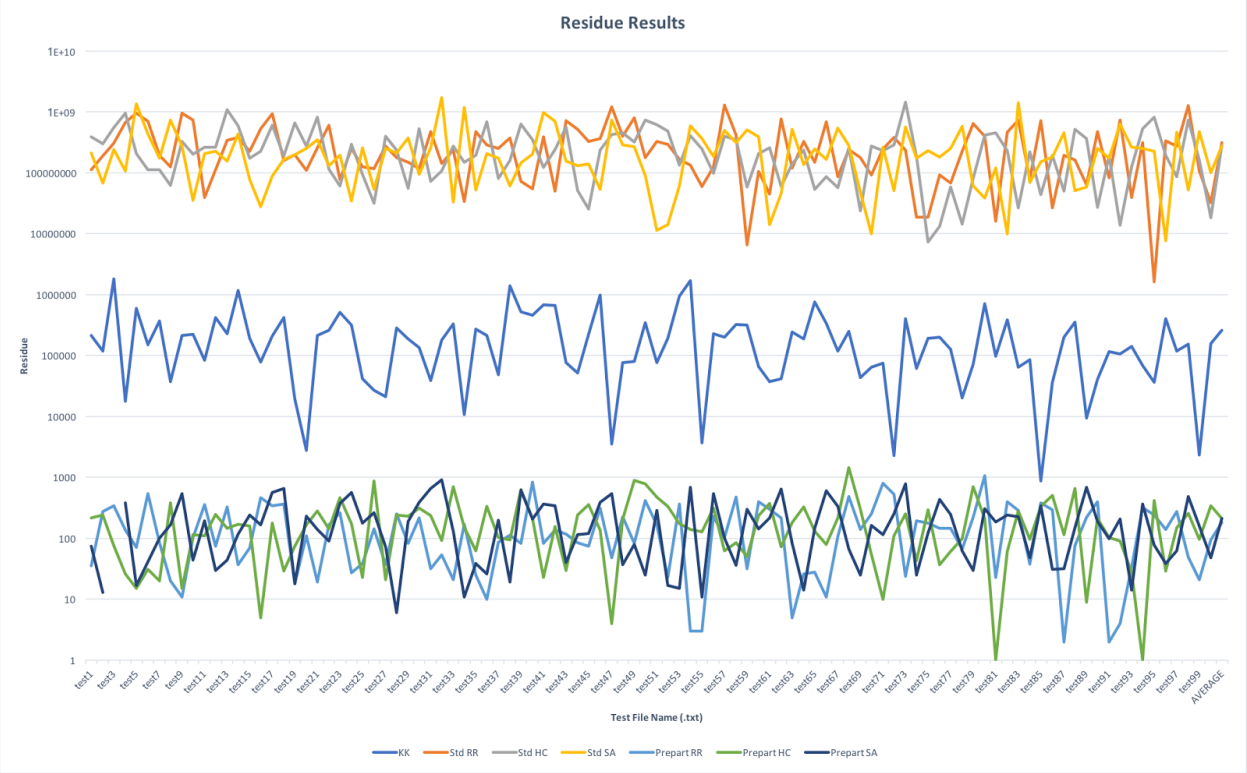
Karmarkar-Karp: 258,394.28
Standard Repeated Random: 315,334,261.4
Standard Hill Climbing: 282,310,367.2

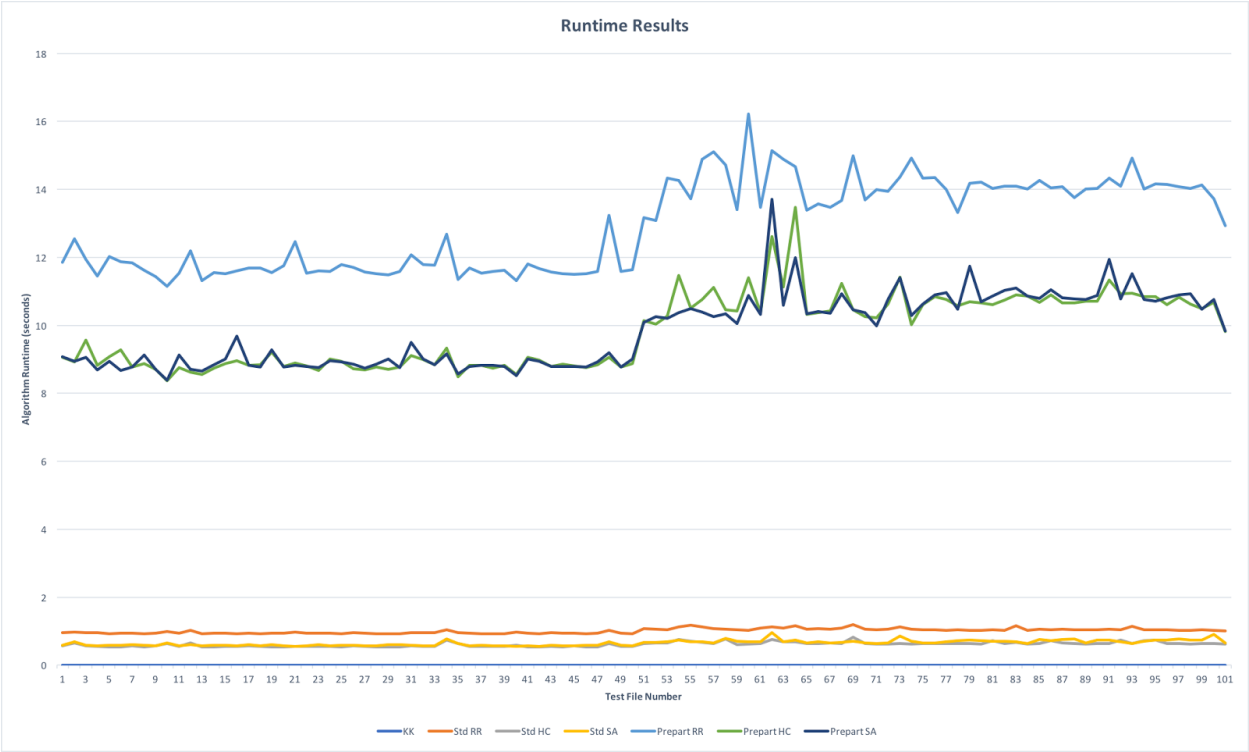
Standard Simulated Annealing: 277,117,175.5
Prepartition Repeated Random: 180.58
Prepartition Hill Climbing: 214.06
Prepartition Simulated Annealing: 213.32

Runtimes (in seconds):

Karmarkar-Karp: 0.000483866
Standard Repeated Random: 1.005331123
Standard Hill Climbing: 0.611045532
Standard Simulated Annealing: 0.650936347
Prepartition Repeated Random: 12.93445673
Prepartition Hill Climbing: 9.819396634
Prepartition Simulated Annealing: 9.834228227

The following two graphs represent the residues (logarithmic y-axis scale) and the runtimes for each of the seven methods over the course of 100 trials, respectively.





Analysis

Let us first look at the results for residues. Looking at the graph, we see that the three standard methods' results are all near each other, as well as the three prepartition methods. The two groups are very far apart, with the prepartition representation giving significantly better results than the standard representation, in the hundreds as opposed to the hundreds of millions. The Karmarkar-Karp algorithm sits in between the two groups, hovering in the hundreds of thousands.

We expected the prepartition representation to be much more effective than the standard representation, because it uses the Karmarkar-Karp heuristic in its algorithms. We know the KK algorithm is an effective approximation, giving the prepartition algorithms an advantage in how the lists are represented. Through 25,000 iterations, all of the algorithms under this representation eventually give relatively low and accurate results. The standard representation partitions the given list into two parts at the start, but the prepartitioning representation allows it to be split in a number of partitions up to the number of elements in the set, allowing for more flexibility and efficiency in representing set partitions. Additionally, we expected that Karmarkar-Karp by itself would be somewhere in the middle of the two representations, though the actual results were much closer to those of the prepartition representation than the standard representation, to an extent by which we were surprised.

Between the three algorithms, we anticipated that simulated annealing would be the most effective, followed by hill climbing, and then repeated random. In reality, this ordering proved true for the standard representation. However, to our surprise, the repeated random algorithm was most effective for the prepartition representation, by a notable margin. We are not entirely sure of the cause, though we theorize that it may be because of the significant number of iterations. Because the prepartition representation allows for many different sets to be generated on each iteration, the RR algorithm can eventually find a good solution. Further, the hill climbing was the most ineffective, likely due to the algorithm's tendency to hover around local minima and get stuck. The simulated annealing algorithm avoids this by sometimes jumping to a different location with some decreasing probability, leading to better results.

As for the runtimes, we expected that the prepartition representation algorithms would take longer than the standard representation, due to the need to account for a large number of partitions as opposed to a fixed number in the standard representation. This was the case, though we were still surprised at the stark difference. The standard methods took between 0.6 to 1 seconds on average, while the prepartition methods took between 9.8 to 13 seconds on average. Within the prepartition representation, the HC and SA algorithms took around the same time (9.8 seconds), while the RR algorithm took on average a full 3 seconds longer. While surprising, this may be a result of the RR loop constantly generating new partitions to test, as opposed to the HC and SA algorithms generating random neighbor solutions in their loop, a process that takes less time in the long run. Additionally, since Python is known as a relatively slow programming language, these times could possibly be cut down in a different language.

Theoretically, we could use solution from the Karmarkar-Karp solution as a starting point for the other randomized algorithms. Simulated annealing and hill climbing iteratively decrease the residue until the best possible residue is found, so picking a more optimal starting point will give the algorithms a better chance of estimating the true residue. Having used this heuristic, the standard representation algorithms would have likely given much better results for the residue, which we can see from the fact that the prepartition representation used a version of the KK heuristic while the standard representation did not. The prepartition representation algorithm would also likely improve with KK as a starting point, though not as dramatically as the standard representation algorithms. However, using KK as a starting point, it is likely that the repeated random algorithm in either representation would not significantly improve, because it simply generates a new solution each time instead of updating iteratively, so the starting point would not make a major difference.