CS 124 Programming Assignment 3: Spring 2022

Your name: Tram B. Nham
Collaborators:
No. of late days used on previous psets: 31
No. of late days used after including this pset: 35

1 Dynamic Programming Algorithm

- **Definitions:** Let b be the sum of all the terms in A. Let $dp[][]$ be a 2D array of size $n \times b$, and $dp[i][j]$ denotes whether some subset from the first to the i^{th} element of A $(1 \leq i \leq n)$ has a sum equal to j $(1 \leq j \leq b)$. Thus, $dp[n][j]$ evaluates to True if some subset of all elements in A has a sum equal to j. We can find the residue by checking among the values of $dp[n][j]$'s, which j gives the closest sum to $b/2$, and the residue is $b - 2j$. In other words, we can look for some subset sum that is as closes to half the sum of A as possible, and the residue is the sum of A subtracting twice this subset sum.

- **Base case:** Set $dp[i][0] = 1$ because there always exists some subset with zero sum among all elements in A. Set $dp[0][j] = 0$ because without any element, no subset sum > 0 is achievable.

- **Recursion:** A subset sum j can be achieved by either including or excluding the i^{th} element, and we can set it to 1. If no subset among the first i elements sums up to j, set it to 0.

$$dp[i][j] = \begin{cases} \max(dp[i-1][j], dp[i-1][j - A[i]]) & \text{if } A[i] \leq j \\ dp[i-1][j] & \text{otherwise} \end{cases}$$

  In an outer loop, iterate i through all n elements of A. For each of the first i elements of A, check whether there exists some subset that sums up to $1, 2, ..., b$. Specifically, in the inner loop of j, iterate from 1 through b to check for every sum possible that is less than or equal to the total sum of A. If the i^{th} element of A is at most j, check if the subset sum j is achievable either with or without the i^{th} element. That is, check if the first $i-1$ elements $A[1 : i-1]$ contains a subset summing up to either $j - A[i]$ or j. If $A[i]$ is greater than j, check if $A[1 : i-1]$ contains a subset summing up to j. Exit the loop after iterating through all n elements and b subset sum values. Next, iterate j downward from $\lfloor \frac{b}{2} \rfloor$ to 1. If $dp[n][j]$ evaluates to True, then there exists some subset in A summing up to j. As soon as we find a True value, exit the loop since this is the sum closest to half the sum of A. The residue is thus calculated as $b - 2j$.

- **Analysis:** The recursion works since a subset sum j is only achievable at element i if among the first $i-1$ elements, either there already exists some subset summing up to j, or there exists some subset summing up to j less element i. Thus, $dp[n][j]$ should give us all subset sums achievable among all n elements of A. To minimize the residue, we just need to look for the subset sum j^* closest to half the sum of A. Since at least one of the two subsets that minimize the residue has to sum up to at most $b/2$, we can loop downward from $b/2$ to 1 to find j^*. The residue is thus calculated as $b - 2j^*$. There are n iterations of the outer loop and b iterations of the inner loop, so the runtime to calculate $dp[i][j]$ is $O(nb)$. The time to find one of the subset sums that minimize the residue is $O(b)$. The time to calculate the residue is constant. In total, the algorithm takes $O(nb)$ time. The space to store $dp[][]$ is $O(nb)$.

2 Karmarkar-Karp Algorithm

Start by putting elements from A in a binary heap, which takes $O(n)$ time. Extract the two largest elements and insert back their difference into the heap. Extraction and insertion both take $O(\log n)$. Repeat this process $n-1$ times. The residue is the value at the root after $2(n-1)$ extractions and $n-1$ insertions. In total, the algorithm takes $O(n \log n)$ time.

3 Experimentation

Table 1: Summary Statistics of Residue and Runtime

	Residue					Runtime (milliseconds)	
Algorithm	Mean	SD	Min	Median	Max	Mean	SD
KK	308,994.84	542,755.65	4,310	142,638.5	3,023,001	0.18	1.14
RAND	273,413,705.16	354,334,032.00	2,337,164	193,900,305.5	2,060,156,150	1,802.66	149.11
HILL	314,047,990.36	244,849,794.13	13,799,868	227,109,967	1,079,957,584	746.18	88.39
SIMUL	252,207,769.28	225,448,609.49	11,134,442	179,543,563	999,974,723	2,063.53	96.06
PRE-RAND	192.32	212.97	6	115	921	11,493.55	328.50
PRE-HILL	897.84	925.43	2	594.5	4,536	9,575.83	456.82
RE-SIMUL	179.48	189.30	1	114	821	26,763.96	1,369.16

Table 1 shows the summary statistics for the residue and runtime in milliseconds from each of the 7 algorithms. The three algorithms using prepartition perform significantly better than the rest in terms of minimizing the residues, with much smaller residue distribution (3-digit residues on average) compared to Karmarkar-Karp (6-digit), followed by the three algorithms using the standard representation (9-digit). The relatively small standard deviation observed in each the three prepartition algorithms also show that the distribution center more closely around the mean, indicating a more reliable performance overall. The standard algorithms, on the other hand, have very wide standard deviations. While the Repeated Random standard algorithm, for example, yield a mean of 273,413,705.2, the residue could get up to 2,060,156,150 in the worst case. For both representations, Simulated Annealing on average yield the smallest residues, followed by Repeated Random and Hill Climbing respectively. However, among the three prepartition algorithms, Hill Climbing perform significantly worse on average compared to the other two ($Mean(SD)$ = 897.8 (925.4) vs. 192.3 (213) and 179.5 (189.3)).

The residue results are reasonable. Karmarkar-Karp works better than the standard algorithms because it's deterministic, while the standard algorithms all start with a purely random solution and then only make very small random moves on their state spaces. Thus, it is much less likely for these algorithms to approximate closely to the optimal residue. The prepartition algorithms have the best performance since they not only rely on Karmarkar-Karp to emulate a deterministic solution, but also improve upon it by searching for better solutions. For both representations, Simulated Annealing perform the best since it's more likely than Repeated Random to move to a better solution if it already finds a good one, while also avoiding getting stuck at a local optimum. Repeated Random perform almost as well as Simulated Annealing since it could leverage the high number of iterations to jump across different solutions to find the closest approximation. Hill Climbing is the worst since it only moves among neighbors with very little difference, and thus is more likely to get stuck at a local optimum.

In terms of runtime, Karmarkar-Karp is the fastest (0.18 milliseconds) since it only runs on one iteration, while the others run on 25000. The prepartition algorithms takes much longer to run compared to the standard algorithms (9.6-26.8 seconds vs. 0.7-2.1 seconds) since they all involve transforming the solution and deriving A' on very large numbers, and repeated calling Karmarkar-Karp, which also costs significant runtime. For both representations, Hill Climbing takes the least amount of time since it moves from one solution to another by changing only one or two numbers, which is close to constant time. Repeated Random takes longer to run since it generates a new solution for every iteration. Simulated Annealing is the slowest since it involves multiple residue calculations and comparisons between solutions within each iteration.

I should note that my heap implementation of Karmarkar-Karp, while still produces the same (correct) output as if I use the **heapq** package in Python, is very slow and I have not had a chance to figure out why. The version I use to submit to the autograder and to produce *Table 1* above uses Python list instead (since heap packages are not allowed). In each iteration, I sort the list of elements in A in descending order, remove the first (largest) two elements, and add (*append*) their difference back to the list. Sorting takes $O(n \log n)$ time amortized since we keep sorting on smaller lists after the first time, while popping and appending takes $O(1)$ time amortized [1]. I also experimented with the **heapq** package on my own and found the runtime to be significantly better for the prepartition algorithms, cutting down time as much as a third of the list implementation (2.9-7.5 seconds).

4 Karmarkar-Karp Extension

As evident in the results above, Karmarkar-Karp produces a relatively good approximation, especially when compared to the standard algorithms that all start from purely random solutions. Therefore, if we start with a Karmarkar-Karp solution, and then use Simulated Annealing or Hill Climbing to search for a better neighbor, our algorithm would be improving upon an already decent solution instead of a random one. If we use Hill Climbing, while it is still possible to get stuck at a local optimum, statistically it would still be better than being stuck at a random local optimum. If we use Simulated Annealing and pick a good cooling schedule, not only do we start on a good local optimum, we could also move to a better local optimum. For Repeated Random, starting with KarmarKar-Karp would not have any effect since a completely random solution is generated at every iteration.

[1] `https://wiki.python.org/moin/TimeComplexity`