CS124 - Programming Assignment 3

Adolfo Roquero and Jerry Yang

April 2020

1 Dynamic Programming Solution

Suppose we have a list of n integers A. Define $D(i,j)$ to be true if we can get the value i by summing numbers in the subarray $A[0,\ldots,j]$, and false otherwise. Then

$$D(i,j) = D(i,j-1) || D(i - A[j], j-1)$$

A subset of $A[1,\ldots,j-1]$ is also a subset of $A[1,\ldots,j]$, so the first component of this statement is clearly true, and if a subset of $A[1,\ldots,j-1]$ sums to i, then we can get the sum $i + A[j]$ by adding a_j to the set. Hence, we can compute D using a bottom-up approach, since we can calculate the value for an entry based on the entries to its left and above.

Now suppose the integers in A sum to b, which we can calculate in linear time. We compute values $D(i,j)$ for $i \in [1, \frac{b}{2}]$ and $j \in [1,n]$. The largest value of i, call it i_m, such that $D(i_m, n)$ is true is the subset sum that minimizes the difference. To see this, suppose that P_1 and P_2 are subsets of A such that $P_1 \cup P_2 = A$ and $P_1 \cap P_2 = \emptyset$. The given problem is equivalent to minimizing

$$\left| \sum_{a_k \in P_1} a_k - \sum_{a_j \in P_2} a_j \right|$$

and since every element of A is either in P_1 or P_2 but not both, this can be rewritten as

$$\left| \sum_{a_k \in P_1} a_k - \big(b - \sum_{a_k \in P_1} a_k\big) \right|$$

Without loss of generality, assume that $\sum_{a_k \in P_1} a_k \leq \frac{b}{2}$. (This must be true of one of sets.) Then our goal is to minimize

$$b - 2\big(\sum_{a_k \in P_1} a_k\big)$$

which is equivalent to finding the largest possible subset sum that is less than or equal to $\frac{b}{2}$.

To construct the actual subarrays, we find the leftmost entry in the i_mth column that has value true. Suppose this entry is in the kth column. Referring back to our definition of $D(i,j)$, this means that a_k must be in the subset in order for its sum to equal i_m. Repeat this process with $i = i_m - A_k$ until every element in the desired subset X is found. Returning to the outlined problem, our solution would be to make $S_k = -1$ for $a_k \in X$, and 1 otherwise.

This algorithm takes time polynomial in nb because there are $n(\frac{b}{2})$ entries in the table and computing each entry requires a constant number of operations (i.e. one subtraction and looking up two previously calculated entries).

2 Karmarkar-Karp

We can implement this algorithm in $O(n \log n)$ steps using a maximum heap structure. We first insert each element of A into the heap. Then while the heap has more than one element, we call `deletemax()` twice, compute the difference of the two values, and insert it into the heap (with the index of the larger element as key). Each iteration of the loop takes $O(\log n)$ time since there are two `deletemax()` operations and one insert, and we do this $O(n)$ times for an overall runtime of $O(n \log n)$. To produce the actual partition, we keep a graph with the element indices as vertices, and add an edge between the indices of the two elements returned by `deletemax()`. We can then run DFS: when we get to a vertex, put it into a different set from the vertex it was explored from.

3 Results

3.1 Residue Performance

While none of the algorithms are guaranteed to find the minimum residue, we can evaluate the relative performance of the algorithms by the residues they return on the same array of numbers. We randomly generated 100 arrays of numbers in the interval $[1, 10^{12}]$, and ran all 7 algorithms on each array. Our results were as follows:

Algorithm	**Average Residue**	**Standard Dev**	**Max**	**Min**
Karmakar Karp	238749	303884	1593829	334
Standard Repeated Random	267553617	235965851	1186325938	981809
Standard Hill Climbing	309370810	290063349	1309119824	2699732
Standard Simulated Annealing	342180075	321989150	1369278127	1912261
Prepartitioned Repeated Random	155	139	704	1
Prepartitioned Hill Climbing	649	724	3830	2
Prepartitioned Simulated Annealing	204	218	920	2

It is important to note that different starting random solutions were generated in each algorithm, so the trial conditions are not identical. However, the average residue is still representative of how well each algorithm performs overall.

3.1.1 Standard vs Prepartitioned

We can see that the prepartitioned algorithms perform better on average than Karmakar-Karp and the standard algorithms by several orders of magnitude. The prepartitioned residues are $\sim 10^3$, while Karmakar-Karp is $\sim 10^6$ and the standard residues are $\sim 10^9$.

One explanation for this is that in the prepartitioned algorithms, each solution can be expected to have substantially more neighbors than in the standard algorithms. Each element in the prepartition has a value in $[0, 99]$, so there are at most $\sim 10^4$ neighboring prepartitions for a solution (in the case that each element is unique). With the standard solution, on the other hand, the number of neighbors is the sum of solutions with one element changed, 100, and two elements changed, $\binom{100}{2}$, but the neighbors with only one element changed are significantly more likely to be returned.

Hence, we can hypothesize that since the prepartitions have broader neighborhoods, the prepartitioned algorithms will take more variable paths from the starting solution (in the hill climbing and simulated annealing cases). Since the starting solution is random, this could enable finding better solutions, as a greater variety of changes to the solution are explored. Conversely, with the standard algorithms, it may be much more difficult to get out of the locally optimal solution in some cases. This is supported by the fact that the standard algorithms had average residues that were significantly higher than Karmarkar-Karp.

3.1.2 Repeated Random vs Hill Climbing vs Simulated Annealing

We can also observe that the repeated random algorithm performs better than the other approximation algorithms. This could be because the success of the hill climbing algorithm, and to a lesser extent simulated annealing, depend heavily on the random starting solution. The hill climbing algorithm in particular may get stuck in a local minimum. While the simulated annealing algorithm sometimes makes moves out of the locally optimal minimum, as the iterations increase, the increase in T(i) makes this less likely. This could explain why the prepartitioned simulated annealing algorithm performs better than the prepartitioned hill climbing algorithm (649 vs 204), although still worse than the prepartitioned repeated random algorithm (155). Interestingly, the standard simulated annealing algorithm performed worse than the standard hill climbing algorithm in our results. This could be explained by our hypothesis on the value of prepartitioning: in the standard simulated annealing case, the variability of the neighborhoods is not enough to make moves away from the local optimum effective most of the time.

3.2 Time Performance

However, the performance increase in minimizing the residue is shown to have trade-off in terms of the time performance.

We obtained the following time results:

Algorithm	**Average Time (in nanoseconds)**	**Standard Dev**
Karmakar Karp	22374	98564
Standard Repeated Random	39360966	2072316
Standard Hill Climbing	6457268	1287092
Standard Simulated Annealing	13865027	1205582
Prepartitioned Repeated Random	226405895	10773234
Prepartitioned Hill Climbing	177470834	4053176
Prepartitioned Simulated Annealing	185418708	40308032

We can see that the fastest algorithm to run is Karmakar-Karp. This makes sense since Karmakar Karp only runs once, while the other algorithms run for 25000 iterations. In addition, we can observe that the average time for the prepartitioned algorithms is almost 25000 times the average time for Karmakar-Karp which makes intuitive sense since for each iteration of the prepartitioned algorithms we run Karmakar-Karp to compute the residue. This also explains why the standard algorithms are faster than the prepartitioned algorithms, as computing the residue for standard solutions takes $O(n)$ (looping over the elements in a solution and performing addition or subtraction), while computing the residue for prepartitioned solutions takes $O(n \log n)$ (as shown in section 2).

4 Further Discussion

Our results indicate that Karmarkar-Karp provides a useful approximation for the optimal residue, and since the algorithm is deterministic, we could use it to determine a starting point that will be reliably better than random. In particular, if we redesign our Karmakar-Karp method to output the corresponding solution (as touched upon in section 2), then it would probably be better to start from this solution in our standard algorithms. Note that the minimum residue of a solution returned by our standard hill climbing and simulated annealing algorithms is greater than the maximum residue returned by Karmakar-Karp. Given our hypothesis that the poorer performance of these algorithms is related to a tendency to get trapped in local optimums, starting from the solution given by Karmakar-Karp would most likely give the algorithms greater success. Since the repeated random algorithm does not rely on the solution's neighbors, starting from the solution to Karmakar-Karp will not change the iterations of the algorithm, but it will ensure that the returned residue is at most as large as that of Karmakar-Karp.

If we were to use the solution given by Karmakar-Karp for our prepartitioned algorithms, however, we would have a starting prepartition where the elements are only one of two possible values, which is essentially a version of the standard solution. However, we can still use the residue returned by Karmakar-Karp to determine whether a random prepartition is a good starting point, and this might improve the speed of the prepartitioned hill climbing and simulated annealing algorithms without (in most cases) negatively impacting their residue performance.