CS124 Programming Project 3

Sean Ty, Brandon Tang

April 2021

1 Introduction

In this project, we explore different algorithms relating to the number partition problem. More specifically, given an array A of real numbers, we explore algorithms to partition A into two arrays S_1, S_2 such that the difference between the sum of the values in both arrays is minimized. For this project, Java is the programming language of choice due to its speed.

2 Dynamic Programming Approach

Let b be the sum of all the elements of A. We wish to find $\min |t_1 - t_2|$, where t_1, t_2 are sums of two subsets that partition A. Note that $t_1 + t_2 = b$ so it suffices to find $\min |b - 2t_1|$ where t_1 is the sum of elements of some subset of A. Also, note that the maximum possible sum of any subset is b by definition, which shows that we will never go out of bounds in our algorithm later.

Given this, we then perform the following algorithm: create an $(n+1) \times (b+1)$ array dp, for which each cell contains a boolean, a pointer to another cell to an array (which may be null), and an integer in the range $[0, n-1]$ or a `None`. Initialize the first field of the first row $dp[0]$ to be false everywhere except for $dp[0][0]$, since 0 is the only attainable sum using no elements. Furthermore, we initialize the pointers of every element in this row to `None`, and the third field to `None`.

We then fill out the first fields of the rest of dp through the recurrence $dp[i][j] = dp[i-1][j]$ OR $dp[i-1][j - A[i-1]]$. To define the pointers and the third fields, the left side takes precedence. That is, if $dp[i-1][j]$ is true, then the pointer in $dp[i][j]$ points to $dp[i-1][j]$, and the third field is set to `None`. If it is false but $dp[i-1][j - A[i-1]]$ is true, then the pointer points to $dp[i-1][j - A[i-1]]$ instead, and the third field is set to $i-1$, which is in $[0, n-1]$. Finally, if both are false, the pointer and the third field are initialized to `None`. This flow of logic is equivalent to the notion that whenever a sum of j is attainable with the first i numbers, then it can be done by either attaining j with the first $i-1$, or you would require the last entry $A[i-1]$ and attain the sum $j - A[i-1]$ with the other $i-1$ entries. Thus, the third field keeps track of which index we are adding into the sum, if we are. Further, note by the argument above that the pointers allow us to keep track of how the sums are constructed.

The algorithm terminates once every cell is filled. Then by the recurrence above, it follows that the bottom row $dp[n]$ describes which sums are attainable if we consider all entries of A. We then go through the bottom row starting at the index $\lfloor b/2 \rfloor$ (to minimize the residue $|b - 2s|$) and going down to zero, stopping once we hit an entry with a first field of true. Note that we only need to consider the possible sums at most $\lfloor b/2 \rfloor$ since a sum of s is attainable by some elements iff a

sum of $b - s$ is attainable by the rest of the elements. Now suppose the entry we stopped at is $dp[n][s_0]$, which we know exists because the trivial partition of having all elements in one subset yields $dp[n][0]$ is true. Then the desired residue is $|b - 2s_0|$.

To find the signs that yield this partition, we initialize an array res of length n, and let all its elements be -1. Since the cell earlier is initialized to true, it follows from the recurrence above that it points to some cell in the previous row that is also true, say $dp[n-1][s_1]$. Then if the third field of $dp[n][s_0]$ is not `None`, call it k, we set $res[k] = 1$. Continue this process inductively until we reach $dp[0][0]$, which we do since we know that the sum is attainable. Note that our backtracking should yield indices that are pairwise distinct, since in each step in the construction of the dp array we can only set the third field to be an integer strictly larger than the last (since the index must be strictly larger). Then, outputting res would yield the desired partition, since all other elements are valued at -1 while the elements we have in the sum are at 1.

This algorithm takes $O(nb)$ time, which comes from when we initialize the array and loop through each element. The space complexity here is $O(nb)$.

3 Karmarkar-Karp in $O(n \log n)$

To implement Karmarkar-Karp in $O(n \log n)$, we create a max heap storing the numbers in the array. We then pop the largest number twice, which takes $O(\log n)$, which gives us the two largest values in the array. Afterwards, we subtract the smaller number from the larger one and reinsert the difference into the array. We repeat this process until there is one number left in the heap, which we know will happen in $O(n)$ steps since each step decreases the number of elements by 1. Assuming the numbers are sufficiently small, subtracting takes $O(1)$ time, and inserting the difference takes $O(\log n)$ time. Since we repeat this $O(n)$ times, the run time here is $O(n \log n)$.

Correctness follows from the fact that we are using a max heap–in each step, popping the first two elements would yield the largest followed by the second largest, and we push back in their (nonnegative) difference, which is exactly what the KK algorithm requires.

4 Algorithm Comparison

4.1 Random number generator

We generate integer arrays of length 100, where each entry is a random integer in the interval $[1, 10^{12}]$. Unfortunately, the `Random` class does not work for generating random numbers larger than Java's integer max value, which 10^{12} certainly is. As a result, we opted to use the `ThreadLocalRandom` library under `Java.util`, which does allow us to generate large random integers through the `nextLong()` method. This mimics a discrete uniform random variable in a specific range.

We however still used the `Random` class to generate random solutions and neighbors in the simulations. We set the seed to be the current time in milliseconds for the results to appear random.

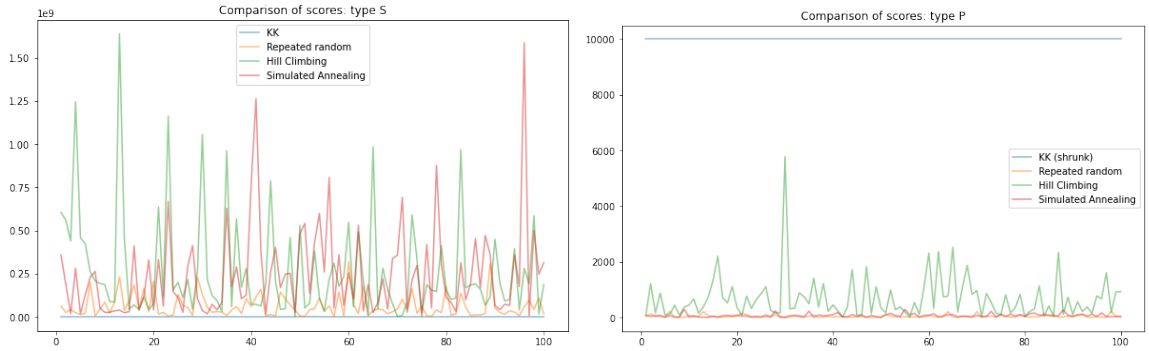
Finally, each trial of each of the randomized algorithms was run for 100000 iterations, which increases performance at the expense of run time.[1] As we will see below, the run time is still rather

[1]The submitted code on Gradescope, however, used 25000 iterations only, as we were falsely flagged as incorrect due to having a very low residue. We suspect that this is because we are doing this for much more iterations, giving

short, so we opted to sacrifice some run time for a lower residue. We also used $T(i) = 10^{10}0.8^{\lfloor i/300 \rfloor}$ as the cooling schedule for simulated annealing, as suggested by the project description.

4.2 Comparison by scores

We first define the *type* of a randomized algorithm to be P if it uses the prepartitioning scheme, and S otherwise. Thus, an array P here will be an array where each entry is an integer in $[1, n]$, while an array S here is an array where each entry is in $\{-1, 1\}$. In this section, we will be comparing the residues obtained by the randomized algorithms against each other and KK:



In the graph at the right, we shrunk the residues of KK, which were usually over 10^6 to make the details more noticeable. For concreteness, the following table summarizes the mean residues per algorithm:[2]

Type/Algorithm	KK	Repeated random	Hill climbing	Simulated annealing
S	240044.79	63045780.07	272753565.29	246327814.51
P	240044.79	43.55	678.19	65.97

For type S, we see that hill climbing performs the worst, followed by simulated annealing, then repeated random, which KK having residues well below the randomized algorithm residues. On the other hand, for type P, we see that repeated random performs best, followed by simulated annealing, then hill climbing, with KK having residues much larger than the randomized algorithms.

We were initially surprised to see that hill climbing and simulated annealing performed worse on average than repeated random, but soon realized that the algorithm might be getting stuck at a locally optimal solution. That is, the "greedy" heuristic of the hill climbing and simulated annealing makes their performance improvement worse than repeated random given more iterations. This could explain why we see this ordering–indeed, if the greedy approach does lead to only a locally optimal solution, then we would expect hill climbing to perform worse than simulated annealing, followed by the repeated random, since hill climbing would be "stuck" in a locally optimal solution, but simulated annealing has a chance of leaving this solution in search of better ones. Meanwhile, repeated random is not affected as it looks for another completely random solution (i.e. the solutions are independent).

us more possibilities to try and reduce our score. Especially helpful for repeated random, since it won't get "stuck" in a locally optimal solution.

[2]Note that KK has the same residues for S and P. This is because the algorithm is independent on the choice of type, and we added this twice for convenience.

We were also surprised to see that the type S algorithms performed far *worse* than type P. Upon further thought, this is likely due to the fact that by selecting our choice of S, there is no other way for us to optimize–we have to directly calculate our residue. On the other hand, selecting P only forces us to *group* certain terms together–we can still optimize the residue through some other means such as KK, which is exactly what we did. Thus, in terms of how much leeway we have in minimizing the residue, the P method gives us more than the S method, and so we should expect the P method to do much better than the S.

We can in fact extend this argument even further to explain why type P performs much better than KK: type P allows us to "force" group some terms before considering KK. That is, we are effectively doing KK but giving ourselves more leeway by altering the array, which in turn leads to better possibilities, as we see in the plots above. Likewise, S performs much more poorly than KK because by choosing S we are "restricting" what we are able to do, thus leading to worse solutions overall.

4.3 Comparison by run times

The following are the run times per algorithm, per type:

For concreteness, the table below shows the mean run time (in milliseconds) per algorithm:

Type/Algorithm	KK	Repeated random	Hill climbing	Simulated annealing
S	0.05	138.14	10.24	17.31
P	0.05	909.0	586.66	638.8

This is about what we would expect. KK is very fast since all it does is create a heap and continuously subtract two elements. Hill climbing is rather fast (compared to the rest) since it only looks at the best neighbor. Simulated annealing is roughly the same, only a little higher due to the fact that we need to simulate probabilistically whether or not we go down to a less optimal solution. Finally, repeated random is slower compared to the other algorithms since we need to constantly generate other completely random solutions and take their residues.[3]

[3]We did several reruns of this with only 25000 iterations and repeated random sometimes performed worse than simulated annealing for type S–they appear to be about the same in performance. Repeated random is much better here since we are generating more solutions independently, while simulated annealing still only looks at some neighborhood.

4.4 Karmarkar-Karp as a starting point

All the randomized algorithms we've implemented so far have used a random solution as a starting point. It is possible that we instead *start* with the solution from the KK algorithm, which intuitively, may be advantageous because we likely start off with a smaller residue.

For the repeated random algorithm, this is especially advantageous because it gives us a lower bound of our best residue. Indeed, since the other solutions are independently sampled, we are effectively running repeated random for one less iteration, but with the guarantee that we will perform at least as well as KK. As seen from the experiments above, this helps for instance for the case of type S.

For the hill climbing and simulated annealing algorithm, the effect is a lot more grey, since by starting in a specific location, we are looking only at a neighborhood around that location. Using KK as a starting point would give us a region in the solution set for which we search on given our iterations and "greedy" heuristic of looking for valleys (i.e. locally minimum residues), so it may take less iterations to reach a local minimum. Thus, it may be advantageous to do so if we do not have a large number of iterations. This approach, however, runs us the risk of getting stuck in a local minimum/valley, as we have observed in our hill climbing experiments (and simulated annealing to some extent). As a result, for more iterations, it may be more advantageous to start with a random solution, so as to not get "stuck" in a local minimum. It would be also be advantageous to have random restarts so that we are more likely to obtain a lower local minimum.

In all cases, running KK as a starting point would increase run times, since we will have to keep track of the resulting solution. However, this extra cost should not be significant, as it is an additive increase to the run time.