CS 124 Programming Assignment 3: Spring 2023

Your name(s) (up to two): Michael Xiang, Marcos Johnson-Noya

Collaborators: None.

No. of late days used on previous psets: 0 for Michael Xiang, 2 for Marcos Johnson-Noya.
No. of late days used after including this pset: 0 for Michael Xiang, 2 for Marcos Johnson-Noya.

1 Introduction

In this programming assignment, we will implement numerous heuristics designed for the Number Partition problem. We will first start off by giving a pseudo-polynomial solution to the Number Partition problem. Finally, we will discuss and compare the results between the different heuristics that we have implemented.

2 Dynamic Programming Solution to Number Partition

Algorithm:

1. **Definition of Sub-Problems**
   Let us try to find the minimum residue by splitting up A into two subarrays A_1 and A_2, and let the sum of elements in A_1 be defined as S_1 and the sum of elements in A_2 be defined as S_2. If we are trying to find the minimum residue, then we are trying to find some A_1 and A_2 such that $|S_1 - S_2|$ is the minimum out of all possible choices of A_1 and A_2. If we define b to be the sum of A, then we see, by basic algebra, that $S_1 = b - S_2$. Thus, with substitution, we see that we get

   $$|S_1 - S_2| = |b - 2S_2|$$

   So, our goal is to minimize $|b - 2S_2|$. With that in mind, let us define our sub-problems so that we find out how to choose elements from A into A_2 wisely, so that $|b - 2S_2|$ is minimal.

   Let $D(i, j)$ be defined as an entry within our DP table that holds 3 values: first, a Boolean value that denotes whether it is possible to have a subset of $a_1, \ldots, a_i$ sum up to j; second, a value that is either None or the integer i that tells us that we must add a_i to a subset of $a_1, \ldots, a_i$ so that the subset can sum to j; and finally, a pointer value that, if it is possible to have a subset of $a_1, \ldots, a_i$ sum up to j, tells us which smaller sub-problem of D leads/points to $D(i, j)$ being true (more on the 3rd will be intuitive once we have our recurrence relation).

   We will explain how to solve the original problem in the following section.

2. **Recurrence Relation**
   Our base cases are $D(0, j)$ for $1 \leq j \leq \lfloor \frac{b}{2} \rfloor$, in which $D(0, j)$ for those values of j are such that the Boolean value dictating whether an (empty) subset can sum up to j is false, and following that, the pointer value is initialized to None and the value dictating if a_0 is included in our sum for j is None since such a sum is impossible. For $D(0, 0)$, we have that the Boolean value is true, and the pointer

values and value dictating if a_0 is included in some sum are false like the other elements in the same row.

To make our recurrence relation clearer, we will split it up amongst the three values that each entry $D(i,j)$ holds. However, note that all three values are calculated at the same time and not separately.

For the Boolean value that indicates the possibility of a subset sum.
We will denote the Boolean value for $D(i,j)$ as $D(i,j)[0]$. Our recurrence relation, for $1 \leq j \leq \lfloor \frac{b}{2} \rfloor$ and $0 \leq i \leq n$ is

$$D(i,j)[0] = \begin{cases} 1 & D(i-1,j)[0] = 1 \vee D(i-1,j-a_i)[0] = 1 \\ 0 & \text{else} \end{cases}$$

where 1 denotes the Boolean value "True" and 0 denotes the Boolean value "False."

For the value that indicates if a_i is included in the subset sum to j.
We will denote this value for $D(i,j)$ as $D(i,j)[1]$. Our recurrence relation, for $1 \leq j \leq \lfloor \frac{b}{2} \rfloor$ and $0 \leq i \leq n$ is

$$D(i,j)[1] = \begin{cases} i & D(i-1,j)[0] = 0 \wedge D(i-1,j-a_i)[0] = 1 \\ \text{NONE} & \text{else} \end{cases}$$

For the pointer value.
We will denote this pointer value for $D(i,j)$ as $D(i,j)[2]$. Our recurrence relation, for $1 \leq j \leq \lfloor \frac{b}{2} \rfloor$ and $0 \leq i \leq n$ is

$$D(i,j)[2] = \begin{cases} D(i-1,j) & D(i-1,j)[0] = 1 \\ D(i-1,j-a_i) & D(i-1,j)[0] = 0 \wedge D(i-1,j-a_i)[0] = 1 \\ \text{NONE} & \text{else} \end{cases}$$

To solve the original problem, we start from the row $D(n,j)$ where $j = \lfloor \frac{b}{2} \rfloor$ and descend until $j = 0$, halting once we find some $D(n,j)$ such that $D(n,j)[0]$ is true, meaning that it is possible to achieve a sum of j with a subset from all elements of A. Let $D(n,j_0)$ denote this point where we halt our descent. If we wanted to find the minimum residue, then, via our formula above, the minimum residue is equal to

$$|b - 2j_0|$$

But we also need to find the exact partition to get this minimum residue. We initialize our return array R to be an array of length $n = |A|$, where all elements of R are equal to -1. We start at $D(n,j_0)$. Since it is assumed to have $D(n,j_0)[0]$ equal to true (denoting whether a subset is possible), it also has a pointer, $D(n,j_0)[2]$, that points to the previous sub-problem that allowed a subset sum to be possible and another value, $D(n,j_0)[1]$, that tells us if a_n must be included in the subset sum to j_0. If the latter value $D(n,j_0)[1]$ tells us that a_n must be included, then we set the nth element of R equal to 1. If not, then nothing changes. Then, we move towards another entry in D, say $D(k,\ell)$, via the pointer value in $D(i,j_0)$. We repeat the same process at $D(k,\ell)$, in which we check if a_k must be included, and we repeat the process until we reach $D(0,0)$.

Finally, our algorithm returns our partition R, in which $R[i] = 1$ means we add $A[i]$ to our sum and $R[i] = -1$ means that we subtract $A[i]$ from our sum.

3. **Dependency Graph is a DAG**
   We see that, since our recurrence relations operate only on smaller sub-problems, our dependency graph is acyclic.

Proof:
Our algorithm is correct by exhaustion. If we consider a case $D(i, j)$, then we know that, when determining if a subset of $a_1, \ldots, a_i$ can sum up to j, we can either include a_i or not include a_i. If we do not include a_i, then a subset of $a_1, \ldots, a_{i-1}$ must sum up to j, which $D(i-1, j)$ tells us. If we do include a_i, then a subset of $a_1, \ldots, a_{i-1}$ must sum up to $j - a_i$, so that adding a_i will ensure a sum of j, which $D(i-1, j-a_i)$ tells us about. Thus, our recurrence relation for $D(i, j)[0]$ is correct since there can only be two ways for $D(i, j)[0]$ to be true. Our recurrence relation for the pointer variable works the same way, but we prefer pointing to $D(i-1, j)$ if possible, because there would be no need to add a_i and thereby point to $D(i-1, j-a_i)$ since both sub-problems work as building blocks for our solution to $D(i, j)$. Similarly, our recurrence relation for the value that tells us if a_i must be included when summing to j works the same way. If that value $D(i, j)[1]$ is equal to i, then that means the *only* possible way that we can have a subset sum from $a_1, \ldots, a_i$ equal to j is if a_i is included, which our algorithm does. Our base cases our correct, because only $D(0, 0)$ has a valid subset sum, as you can only sum up to 0 with 0 elements of A. Thus, our recurrence relations solve correctly up to $D(n, \lfloor \frac{b}{2} \rfloor)$.

We also correctly get the minimum residue and signs for our number partition. Given any A_1, A_2, such that A is split into A_1 and A_2, we see that a minimal residue happens when A_1 and A_2 are as close in value as possible. WLOG, let us assume that $A_1 \geq A_2$. Thus, via algebraic observation, we observe that $A_2 \leq \lfloor \frac{b}{2} \rfloor$. So, for $D(n, j)$, starting from $j = \lfloor \frac{b}{2} \rfloor$ and descending towards 0 is correct, because we want to stop at the highest value of j possible, so that the sums of A_1 and A_2 can be as close as possible. The signs of our number partition follow from the pointers of this this highest value j, so reconstructing the solution via backtracking correctly gives us the elements in A_2. Since we are asking for absolute difference, it doesn't matter which set of elements have a -1 sign/belong to A_2 and which have a $+1$ sign and belong to A_1, as long as the same set of elements have the same sign for the optimal solution (i.e. A_1 and A_2 are interchangeable). Thus, our algorithm is correct.
Runtime:
We are building an $(n+1) \times (b+1)$ size matrix, which takes $O(nb)$ time. Logically, then, there are (nb) sub-problems. Since each sub-problem is comparing between a constant number of previous sub-problems, it takes $O(1)$ time to solve each sub-problem. Thus, the time complexity of our algorithm is

$$O(nb) \cdot O(1) = O(nb)$$

so our algorithm is polynomial in n and b.

3 Karmarkar-Karp Implementation

Algorithm:
Let us use a binary max-heap. Let $|A| = n$. Let us first heapify A. For $n-1$ iterations, we shall do the following.

1. We pop the largest element, a, from the heap (and the heap will heapify itself after).
2. We pop "another largest element" from the heap, b, which is equivalent to the second largest element after the first element that we popped from the heap (and the heap heapifys itself after).

3. We push the difference $(a - b)$ into the heap (and the heap will heapify itself after).

After our $n - 1$ iterations, we return the largest/only element from the heap - this is our residue from our heuristic.

Proof:

This is a heuristic, so it is not guaranteed to be correct. We are implementing it as given in the problem description, and via the rubric, we don't need to explain correctness.

Note that we are correctly following the instructions of the problem with our implementation. Heapifying A guarantees that we can access all heap operations. The description wants us to get the two largest elements and replace it with its difference. Thus, we decrease the size of our heap by one each time. And if we want to run this operation until there is only 1 element left, then running it $|A| - 1$ operations will ensure that there is only one element left. Also, popping two elements from our max-heap, via analysis given in section, gives us the two largest elements. So, taking their difference and adding the difference back to the heap correctly follows the description given.

Runtime:

We first run a heapify operation on A, which, if $|A| = n$, takes $O(n \log n)$ operations. Then, we run a for loop for $n - 1 = O(n)$ iterations. For each iteration, we extract the max element twice, taking $O(\log n)$ time for each extraction, and we insert the difference of the two elements, in which subtraction takes $O(1)$ time assuming small numbers and inserting into the heap takes $O(\log n)$ time as given in section. Thus, we do three operations that take $O(\log n)$ time, so each iteration takes $O(\log n)$ time. Thus, our for loop takes

$$O(n) \cdot O(\log n) = O(n \log n)$$

Finally, we return our only element left in the heap of A, which takes $O(1)$ time.

In total, our dominating term is $O(n \log n)$ in both heapify and our for loop, so our time complexity for implementing this algorithm is

$$O(n \log n)$$

4 Discussion and Comparison of Different Heuristics

4.1 Implementation of Heuristics

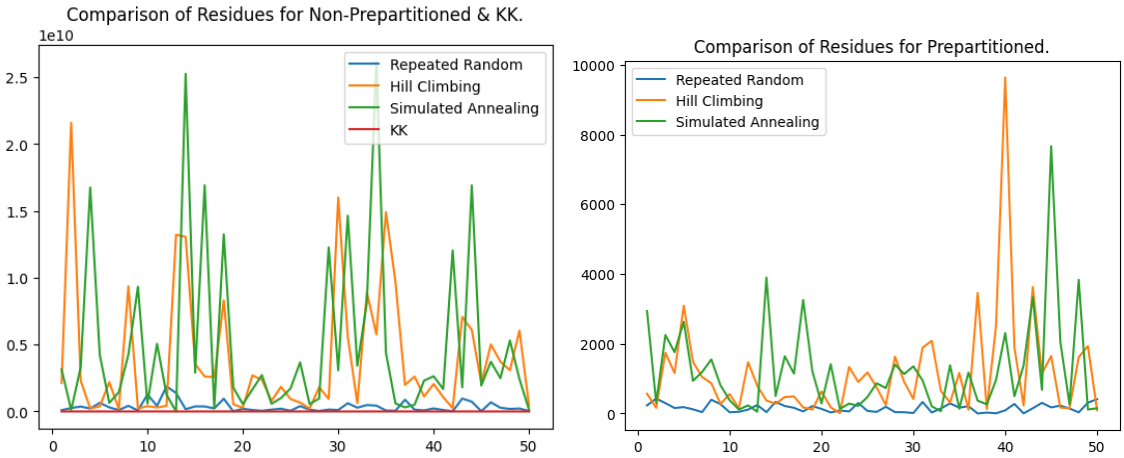
We implemented all randomized heuristics according to the pseudocode. Python3 can handle integers as big as 10^{12}, so we used the *randint* method from python's random library to generate integers in the range of $[0, 10^{12}]$. When finding random neighbors, we used the *sample* method from python's random class to generate 2 unique numbers $i, j \in \{0, \ldots, n - 1\}$, in which $|A| = n$ and our arrays representing our solution for the minimum residue are zero-indexed (so n - 1 is the last index). And when generating random solutions, we used the *choice* method for non-prepartioned algorithms to determine if the sign of a_i should be $+1$ or -1. For prepartitioned algorithms, we still used the *randint* method to generate some random number between 0 and $n - 1$. For the Karmarkar-Karp algorithm, we implemented it as described above, in which we used a heap to pop the largest two numbers from A and add its difference back into the heap, until we only had one element remaining, in which case we returned that element as our residue. We used python's standard heapq library to implement the heap, and since heapq is a min-heap, we took the additive inverse of all elements of A to transform it into a max-heap, in which the smallest (most negative) numbers are the largest elements of A.

We then simulated 50 different trials, in which we generated a random 100-integer array and ran all algorithms on it. We discuss our results below.

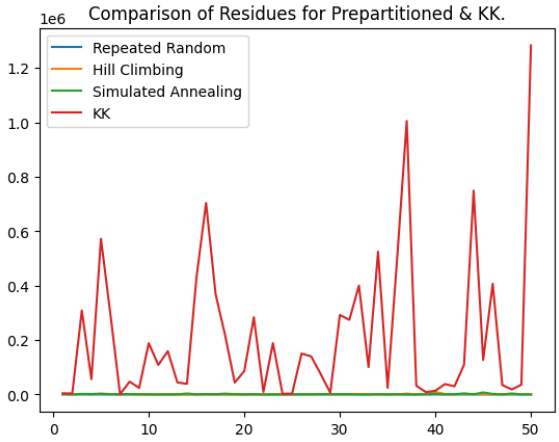
4.2 Comparing between the 6 Randomized Heuristic Algorithms and the Karmarkar-Karp Algorithm

4.2.1 Comparing the Residues

We used Deepnote to help graph our results. We get the following results.



For the graph with prepartitioned residues, we decided not to include the KK algorithm residues because they were far greater than the other 3 randomized heuristics. For proof, this is how the KK algorithm compares to the other 3 algorithms with prepartitioning.



To show how the algorithms compare to each other numerically, we took the average of the residues for each algorithm, and we get that the average residues for each algorithm are

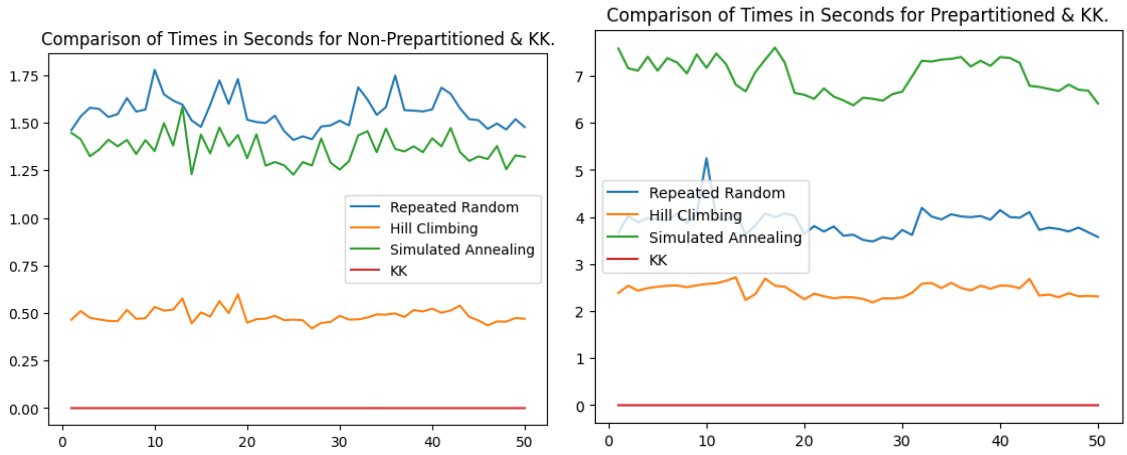
Algorithm	No Prepartitioning	Prepartition
KK	$2.12 \cdot 10^5$	$2.12 \cdot 10^5$
Repeated Random	$3.52 \cdot 10^8$	157.8
Hill Climbing	$4.00 \cdot 10^9$	1157.88
Simulated Annealing	$5.01 \cdot 10^9$	1257.36

For the non-prepartitioning case, we see that the KK algorithm performs the best, followed by the Repeated Random algorithm, Hill Climbing algorithm, and with the Simulated Annealing algorithm performing the worst but on the same order as the Hill Climbing Algorithm. It makes sense that the KK algorithm performed the best, given that the KK algorithm is a credible heuristic that should outperform random algorithms that are not optimized. The Repeated Random algorithm should be the most optimal random algorithm, which we see here in the graph and table, because we are choosing a random solution everytime. Thus, we can never get stuck in some local neighborhood, in which we can only inch towards some local optima instead of a global optima. The Repeated Random algorithm does not have any constraints for the "next" random solution, so we can always jump to a better, globally optimal solution. And by the same logic, the Hill Climbing algorithm should perform worse than the Repeated Random algorithm, because the Hill Climbing algorithm only considers local neighbors; it can only reach some locally optimal solution, not a globally optimal one. The Simulated Annealing algorithm appears to perform worse than the Hill Climbing one even though the Simulated Annealing algorithm doesn't require the solution to jump to a locally optimal neighbor (preserving the chance to go towards some global minima). However, both the Simulated Annealing algorithm and the Hill Climbing algorithm have the same order of residue, so the results are similar enough that we can attribute the Simulated Annealing algorithm's status as less optimal than the Hill Climbing algorithm to the prospect that, whenever the Simulated Annealing algorithm jumped to a non-optimal local neighbor, it, on average, jumped to a worse neighborhood than it was before.

We see that the prepartioned algorithms outperformed its non-prepartitioned counterparts. This makes sense because prepartitioning imposes less "constraints" than not prepartitioning. Given by the description in the programming set, prepartitioning allows us to partition or group different elements together. Thus, this allows us more freedom in choosing which elements to group together before doing the KK algorithm. Thus, it makes sense that we outperform KK in this scenario, since we are doing the KK algorithm, just with more freedom with regards to the grouping of the elements. Also, it makes sense that we outperform the non-prepartitioned counterparts because we are doing the KK algorithm as an additional "optimization" when calculating our residue, compared to the non-partitioned algorithms, in which we have to manually calculate the residue without any additional optimizations. Also, when we calculate random neighbors, we can switch around groups of elements that we partition together, but with non-partitioning, we can only switch 1 or 2 elements. Thus, we get more freedom by prepartitioning, which helps us not get stuck at some locally optimal solution.

4.2.2 Comparing the Performance/Speed

Again, we use deepnote to graph. We get the following graphs.



Calculating the average times for each algorithm in seconds, we get that

Algorithm	No Prepartitioning	Prepartition
KK	0.000048	0.000048
Repeated Random	1.556052	3.885834
Hill Climbing	0.485730	2.440329
Simulated Annealing	1.364365	7.000116

The results that we get from both the graph and table correlate with our logical observations about each algorithm. The KK algorithm should run the fastest, because it simply uses a heap to linearly subtract two elements and runs in $O(n \log n)$ time. For the No Prepartitioning case, it makes sense that the Repeated Random algorithm runs the slowest because it has to generate a random solution each iteration, so each iteration takes $O(n)$ time. The Hill Climbing should run faster than the Repeated Random algorithm because the Hill Climbing algorithm only has to search for neighboring solutions and thus, doesn't need to generate a completely new solution as we do in the Repeated Random algorithm. By the same logic, the Simulated Annealing algorithm should run faster than the Repeated Random algorithm, but it should be slower than the Hill Climbing algorithm since we need to probabilistically determine if we should move to a neighboring solution, even if that solution is worse than the current solution, which adds on to the time that we take for the Hill Climbing algorithm.

For the Prepartitioning case, it makes sense that all algorithms run slower than their non-prepartitioning counterparts because we need to generate solutions and run the KK algorithm each iteration, which takes $O(n \log n)$ time each iteration. Interestingly, we see that the Simulated Annealing algorithm in the Prepartitioning case is the slowest algorithm. This could be due to the fact that our computation for the probability, due to how large the numbers are, is very slow on our respective devices.

5 Using the Karmarkar-Karp Algorithm as a Starting Point

If we were to start off our randomized algorithms with the KK algorithm solution, then our solutions/minimum residues would be bounded by the residue that we get from the KK algorithm. Given that all of our randomized algorithms begin with some random solution, it may be better to use the KK algorithm as a

starting point, since the answer from a credible, deterministic heuristic like the KK algorithm will likely have a lower residue that some random solution. For the prepartitioned case, we shall set the numbers of partition P equal to 1 or 2, so that we split A into two subsets such that its residue is equal to what we get with the KK algorithm, something that we can do by labeling elements in the heap when we run the KK algorithm. With all that in mind, let us examine how this will affect each of the 3 randomized heuristics.

For the repeated random algorithm, it is particularly likely to lead to a better solution if we start with the KK algorithm solution, since, as discussed above, starting from the KK solution gives us a lower bound. Since we are repeatedly generating random solutions each step, independent of our initial solution that we get from the KK algorithm, we see that, by starting off with the KK algorithm, we are only lower bounding our algorithm without sacrificing the chance to improve our residue. In other words, since our repeated random algorithm will randomly generate solutions at each iteration, it never gets stuck at a local state, so starting off with the KK algorithm can only improve the minimum residue that we get.

For the hill-climbing and simulated annealing algorithms, the benefits of starting off with the KK algorithm are not as clear, because both algorithms rely on reaching neighbors of some random solution. On one hand, via past analysis, it lower bounds the residue that we get, which may be good if our random solutions and their neighbors are not great at all. If the KK algorithm residue is in a neighborhood such that it leads to a local minima, then starting from the KK algorithm may not be as optimal as starting from another random solution that leads to a better minima. The simulated annealing algorithm will do better than the hill-climbing algorithm in this case, since the simulated annealing algorithm can probabilistically move to a worse solution that lies in a more optimal neighborhood. However, starting from the KK algorithm can be better than some other random starting solution, in which case it may both take less iterations to reach the local minima and also lead to a better minima than other starting solutions.

For runtime, starting off with the KK algorithm solution would **increase** the runtime for all algorithms. This is because each algorithm would have to start off with running the KK algorithm, which would add $O(n \log n)$ time to the runtime of all randomized algorithms. However, given that this is just an extra additive term to our runtime, this runtime change should not be substantial.