CS124 Programming Assignment 3

Liya Jin

Due April 20, 2022

No. of late days used on previous psets: 8

No. of late days used after including this pset: 10

1 Warm-up

1.1 Dynamic Programming Solution: Number Partition

We find a DP solution for the Number Partition problem. First, let us define syntax for the Number Parition problem as follows: A is the initial set to be partitioned, A_1 and A_2 are its residue-minimized partitions, and b is the sum of all the elements within A. The Number Partition problem is then whether there exists some A_1 that can be produced from A where the sum of the elements in A_1 is $\lfloor \frac{b}{2} \rfloor$. (We assume for the purposes of this solution even that if b is odd and the sum of the elements in A_2 is $\lceil \frac{b}{2} \rceil$, this is still a sufficiently accurate solution.)

Definition: Let $D(j, k)$ be the boolean value indicating whether there exists a subset of elements $(a_1, ..., a_j)$ whose elements sum to k. $D(n, \lfloor \frac{b}{2} \rfloor)$ would give us our final answer.

Recursion: All sets have a subset whose elements sum to zero (empty subset). Additionally, for some subset $(a_1, ..., a_j)$ to contain a subset whose elements sum to k, either the set $(a_1, ..., a_{j1})$ contains a subset whose elements sum to k, or that set contains a subset whose elements sum to ka_j. We set $D(j, k)$ to be true if $D(j, k1)$ is true, or if $D(ja_k, k1)$ is true. In other words, $D(j, k)$ is true if either the subset minus one element sums up to k, or if the subset itself sums to k. In mathematical terms:

$$D(j,k) = \begin{cases} D(j-1,k) \text{ or } D(j-1,k-a_i) & \text{if } k > 0 \\ \text{True} & \text{if } k = 0 \end{cases} \tag{1}$$

We end up filling in a table of dimensions n by $\lfloor \frac{b}{2} \rfloor$. Again, $D(n, \lfloor \frac{b}{2} \rfloor)$ gives us our final answer.

Analysis: To fill each cell of the output array, we index into two other values in the array: filling each cell takes constant time. We iterate over the entire array. The runtime of this function, then, is given linearly by $n * \lfloor \frac{b}{2} \rfloor) = O(nb)$.

1.2 Karmarker-Karp in $O(n \log n)$

Assuming that all arithmetic operations take constant time, we can implement Karmarker-Karp (KK) in $O(n \log n)$ if we represent the input with a max heap.

While initializing our max heap by inserting each node, in the worst case each takes the log of the number of elements already in the heap to find its appropriate location. There are n elements to be inserted, so we estimate the loose runtime of $O(n \log n)$ to initialize the max heap. Then, each step of the KK algorithm requires popping 2 nodes off the heap and inserting 1 (the difference of the two popped). Each of these actions in the worst case takes $O(\log n)$ time: since we do exactly 3 actions (a constant number) per step of KK, each step still runs in $O(\log n)$. Finally, the KK algorithm is run $n - 1$ times to remove every element until only one remains (our residue).

In total, then, we see that the entire KK algorithm runs in $O(n \log n)$ time.

2 Implementation

2.1 Karmarker-Karp

My Karmarker-Karp algorithm was built off a straightforward implementation of the max heap data structure, as aforementioned. I wrote a class `maxheap` with the standard max heap functions and a print function for testing. The actual Karmarker-Karp algorithm is implemented in `kk`: after inserting the given input into a max heap, I repeatedly popped two numbers and inserted their difference until there was only one node remaining: our residue.

2.2 Solution classes

After noting that the two representations of solutions, standard and prepartitioned, shared much of the same functionality—specifically generating random solutions, random neighbors, and residue—as well as identical implementations via the heuristics algorithms, I decided to create an umbrella `Solution` class and implement each solution representation as a child class—`Standard` and `Prepart`.

My `Solution` class contains three classes meant to be overwritten by its children's divergent implementations: `randsol` for generating random solutions given a solution size, `randneighbor` for generating random neighbors of a given solution, and `residue` for generating the residue of a given solution. `Solution` also contains 3 functions for the heuristics algorithms, inherited by both `Standard` and `Prepart`: `repeatrand`, `hillclimb`, and `simanneal` (functionality self explanatory).

With this class implementation, I only had to write each heuristics algorithm and each random solution/neighbor/residue generation function *once*, saving a lot of time and greatly simplifying my code.

2.2.1 Random neighbors for standard solution

@LIYA TBD

2.3 Extras

Learning from programming assignment 1, I made sure to seed my random number generator every time a solution was called to ensure fair results (which sometimes required importing the `random` module within functions in a class!).

I also wrote a `runTest` function to generate the data for my experiments.

3 Results

Using `runTest`, I ran 50 trials, each with an input of 100 randomized integers on the range of $[1, 10^{12}]$. Each trial outputted the residue given by KK, the initial residue of the first randomly-generated starting solution, and the residues given by the 3 heuristic algorithms for standard and prepartitioned solution representation.

I made sure to use the same randomized input to calculate each respective residue for within trial, generating a newly-randomized input for each trial. The total averages over all 50 trials is shown below.

	standard				*partition*			
KK	**random**	**rrand**	**hillclimb**	**simanneal**	**random**	**rrand**	**hillclimb**	**simanneal**
23408	5536609249854	356754156	1295247892	442201276	7883516	415	918	374

Figure 1: Averages at the bottom. All averaged residues across 50 trials, from all heuristics algorithms for both standard and partition solution representations, as well as KK and random residue values.

I also looked at the runtimes of all the heuristic and the KK algorithm. The average runtimes across all 50 trials are below.

KK	*standard*			*partition*		
	rrand	**hillclimb**	**simanneal**	**rrand**	**hillclimb**	**simanneal**
4	74	31	19	2032	1893	1903

Figure 2: All average runtimes for both solution representations and all heuristics algorithms across 50 trials.

4 Discussion

4.1 Residue comparison

The prepartitioned representation of the solution performed better than the standard representation, even without any heuristic algorithm being run. This is likely because prepartitioning allows for the KK algorithm to do the bulk of the residue-finding. When compared to a fixed set partition, like the one generated by the standard reprsentation, the KK algorithm is likely to find a much more efficient one once run on that generated prepartition.

Based on our averages, in fact, prepartitioning produces better residues by a factor around 1-million.

4.2 Runtime comparison

We also see, however, that these better residue values come at the cost of computing time: the sequence solution merely has to sum its solution output multiplied with the input. Prepartitioning requires we run the more time-costly KK algorithm for each of the 25,000 iterations of each heuristic algorithm.

We see based on our averages that prepartitioning takes around 100 times longer than the standard representation does.

4.3 Heuristic comparison

We can see that hill climbing generally produced the worst residues for both the standard and prepartitioned solution representations. This might suggest that local minima (in terms of residue value) are more frequent than optimal for the hill-climbing technique: it seems we must be getting stuck at one of these such minima.

4.4 Using KK to bolster other algorithms

We see that many of the final residues from each of the three heuristic methods for both solution representations are actually higher than the KK-generated residue. This implies that—regardless of which solution representation we use—if we used the KK-generated solution as the starting solution (as opposed to a random solution) for our heuristic functions, our final residues might be lower.

Whether or not starting with KK would significantly and consistently lower the final residue is different for each method. Residues yielded by repeated random would likely remain around the solution KK produced, since finding a better solution randomly is improbable. The results of hill climbing—namely, that the ultimate residues are higher than other methods—suggests that we may be getting stuck at local minima, so starting at a lower residue value with KK should help (unless KK is itself a local minimum). Simulated annealing has the best chance of finding a better final solution, since our starting, KK-generated residue would be low enough such that the probability of taking a worse solution would be higher than it would be for my current trials. This would likely give the algorithms greater flexibility in its search for a lower residue.