Programming Assignment 3
CS124: Data Structures and Algorithms
Prof. Mitzenmacher

Dhilan Ramaprasad
dhilanramaprasad@college.harvard.edu
Collaborators: Benny & Phil

1 Dynamic Programming Solution to Number Partition Problem

Note: Thank you to Amy and Esther for your additional guidance in generalizing my solution.

The `Number Partition Problem` which is NP-Complete takes a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers. The output is a sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in \{-1, +1\}$ such that the *residue*, u, is minimized, where:

$$u = \left| \sum_{i=1}^{n} s_i a_i \right|$$

For the sum b of integers in A, we will demonstrate a DP solution with time complexity $O(nb)$.

1.1 Defining a Recurrence

It is worth *first* exploring a constrained version of the number partition problem wherein we determine whether we can create equal subsets such that $u = 0$. For $u = 0$, each subset of elements (those assigned $s_i = -1$ versus those assigned $s_i = +1$) must contain elements such that the sum of each subset's elements is $\frac{1}{2}\sum_{i=1}^{n} a_i$, respectively.

In the case of our setup for minimization of residue, reaching exactly $\frac{1}{2}\sum_{i=1}^{n} a_i$ is not a requirement—instead, our goal is to approach that value as closely as possible (note this also removes any constraint on the sum of elements in A being even).

It follows that we seek to know, given a set of elements, the largest constructible sum which is $< \frac{1}{2}\sum_{i=1}^{n} a_i$. Naturally, the aforementioned inspires a building-up or *array-filling* strategy based on a constrained number of elements (in an ordered fashion such that we can re-construct the partitions and make use of symmetry). In essence, we will find the largest achievable sum given the option to select from elements $\{a_1, \ldots, a_i\}$. Our array will fill as such:

1. If a given sum can be reached using a selection of elements chosen among $\{a_1, \ldots, a_{i-1}\}$, then we know the sum is achievable with additional elements. (Element a_i need not be selected to achieve the sum since it was achievable on the more constrained range.)
2. If a given sum minus the value of the current element, a_i is achievable (i.e., we can achieve $Sum - A[i]$) using a selection of elements chosen among $\{a_1, \ldots, a_{i-1}\}$, then we know the whole sum is achievable by adding the

a_i element to the aforementioned selection which achieved $Sum - A[i]$. (Element a_i is required to achieve the sum, this time.)

Elts, Sum	0	1	2	...	b
$\{\}$					
$\{a_1\}$					
$\{a_1, a_2\}$					
...					
$\{a_1, ..., a_n\}$					

Table 1: *Array-Filling for DP Solution*

Written as a recurrence we can define **E[i, s]** to return a boolean {**TRUE**, **FALSE**} given the *reachability* conditions enumerated above and inputs i which corresponds to the indices 1 to i of list A on which elements may be selected to form a partition as well as s which represents the sum to be exactly reached by the partition of elements selected on the limited selection of indices given by i. For our case:

$$E[i, s] = \begin{cases} \text{True} & \text{if } (E[i-1, s] \textbf{ OR } E[i-1, s-a_i]) = \text{True} \\ \text{False} & \text{o.w.} \end{cases}$$

Using this recurrence, we'd set some base conditions:

$$E[0, s] = \textbf{False} \quad \forall s \neq 0$$
$$E[i, 0] = \textbf{True} \quad \forall i$$

We can find the minimal residual by traversing the final row with eligible elements $\{a_1, \dots, a_n\}$ to find the largest achievable sum $\leq \left\lfloor \frac{1}{2} b \right\rfloor$. In other words, let $S^* = \left\lfloor \frac{1}{2} b \right\rfloor$, start at cell $E[n, S^*]$ and traverse left toward $S = 0$ until reaching first {**True**} cell. Similarly, if implementing recursively, a call to $E[n, S^*]$ will fill all relevant prior cells.

If we prefer a "building-up" DP method to solve, we can fill all of row 0 and column 0 given the base cases, and then fill all rows of the array and all columns up to column $\left\lfloor \frac{1}{2} b \right\rfloor$ based on prior filled cells, and we should have all relevant information memoized for any relevant subsequent recursion to find minimal residue.

1.2 Constructing Partition of Minimized Residue

To return the elements which make up the smaller of the two subsets (could be equal) whose sum is as close as possible to, yet less than or equal to, $\left\lfloor \frac{1}{2} b \right\rfloor$, we simply must traverse our array starting at position $E[n, S^*]$ moving left (toward $Sum = 0$) until the first **True** is encountered (which may have been in the starting cell) whose column index indicates the starting partial sum $s = S_{start}$ and implies residue is $u = b - 2 \cdot S_{start}$). Then traverse the array upward (toward the empty set row) in the given column until we hit the first **False** cell in the row corresponding to a selection among elements $\{a_1, \dots, a_j\}$ noting that the a_{j+1} element was required to achieve the given starting partial sum; thereby, it is part of our set.

Now, in the same row (in which we found the first **False** cell) corresponding to a selection among elements $\{a_1, \dots, a_j\}$, find the column whose index is $S_{start} = S_{start} - A[j+1]$ whose corresponding cell in the row should read **True**. Traverse upward (toward the empty set row) until hitting the first **False** cell) corresponding to a selection among elements $\{a_1, \dots, a_k\}$ which indicates a_{k+1} was required to achieve the given starting partial sum; thereby, it is part of our set. Repeat the S_{start} -= $A[k + 1]$, column find, and traverse upward to False procedure until $S_{start} = 0$ at which point you will have marked all elements corresponding to the smaller set, leaving all remaining elements to the opposite set.

By construction, then, the minimized residual set can be constructed, and each entry in the array after the base cases is determined by two other elements whose entries will already have been filled; thereby, we find a constant time operation for each cell and there are O(nb) cells implying **O(nb)** run-time for array-filling. Traversal to reconstruct the partition is again **O(nb)** time as we only traverse the array once in reverse.

Technically, time and space is further constrained to $O\left(n\left\lfloor\frac{1}{2}b\right\rfloor\right)$.

2 Karmarkar-Karp in $O(n \log n)$ steps

The KK algorithm itself seems as though it runs in $O(n)$ steps; however, the finding and extracting of the largest two elements is a non-trivial task which does not take $O(1)$ time, but instead grows linearly with n if using an array or takes $O(\log n)$ time using a binary max heap.

Using a binary max heap, then. There are 3 $O(\log n)$ operations at each of n iterations: extract two max elements and insert their difference back into heap. Thereby we can achieve $O(n \log n)$ runtime.

The above assumes values in A are small enough such that arithmetic operations (subtraction/absolute value) can occur in $O(1)$ time.

3 partition.cpp

3.1 Implementation Choices

Despite only testing on **100** unsorted integers, I implemented a *binary max heap* for both coding practice and its marginal benefits:

- 1 `Build-Heap` operation $O(n \log n)$ initially
- 2 `Extract-Max` operations per iteration $O(2n \log n)$
- 1 `Insert` operation per iteration $O(n \log n)$

We'll find: $\underbrace{100\left(3\log_2(100)\right)}_{\text{3 extract/insert per iter.}} + \underbrace{100\left(\log_2(100)\right)}_{\text{build heap}} \quad < \quad \underbrace{100^2}_{\text{loose bound array ops}}$

My implementation of a C++ array-based binary max heap can be found in `maxheap.h` and `maxheap.cpp` or alternatively at the top of `partition.cpp` on the version of the code submitted to Gradescope. For general build, I used `#include "maxheap.h"` and a makefile to link the additional object files.

3.2 On reconstructing the partition

In order to optimize for speed in calculating residue, I did not prioritize keeping track of the set assignment. Reconstructing the set would require further implementation and a two-coloring.

4 100 Random Instances

4.1 Data Collected

On each of 100 trails, we generate integer values on the range $[0, 10^{12}]$, to populate an unsorted array for use in the Number Partition Problem. Using the 7 suggested heuristics/random approximation algorithms, the results are shown below.

		STANDARD		
ALGORITHM	Karmarkar-Karp	Repeated Random	Hill Climbing	Simulated Annealing
AVERAGE	175224	276188359	338691311	6451783199
MEDIAN	94056	223988585	260924961	2425259389
MINIMUM	738	1936774	98324	19022355
MAXIMUM	1161195	1050935270	1537435337	78881339669
STD DEV	212992	235161038	348597689	12891939792
RUNTIME (ms)	0.02	208.12	10.34	14.30

Table 2: *100 Random Instances of Number Partition Problem & Solutions without Prepartitioning*

In the data below, prepartitions were randomly generated at the start to provide variation to our random algorithms and heuristics.

		PREPARTITIONED		
ALGORITHM	Karmarkar-Karp	Repeated Random	Hill Climbing	Simulated Annealing
AVERAGE	175224	170	644	220
MEDIAN	94056	132	312	148
MINIMUM	738	4	1	2
MAXIMUM	1161195	789	7766	1419
STD DEV	212992	155	988	254
RUNTIME (ms)	0.02	650.23	433.27	445.62

Table 3: *100 Random Instances of Number Partition Problem & Solutions* ***with*** *Prepartitioning*

4.2 Discussion

From a cursory glance at the collected data, we see that the **prepartitioned repeated random** methodology provides, on average, the best solution, which follows from the standard implementations wherein repeated random provides the best solution given the options of two alternate local search algorithms. Prepartitioning, wherein we create a combination of some select elements (sharing set assignment) before beginning our routines, seemed to offer vastly better results than the random set assignment and local searches. Although coming at an expense of magnitudes higher time, with the given number of iterations, the prepartitioned methods likely came close to optimal residues.

Both local search algorithms appeared to perform worse than repeated random methods, which likely indicates that the `Number Partition Problem` is riddled with plenty of local minima. Given a better tuned *cooling schedule*, it is plausible that the simulated annealing could return better results in both the standard and prepartitioned strategies.

Finally, it is worth noting that the time for completion of all methods is under 1 second, while Karmarkar-Karp is a fraction of a millisecond and the standard local search algorithms of 10s of milliseconds. Naturally, the repeated random algorithms (pre-partitioned and standard) are much slower than the other implementations given excessive calls to our random number generator; however, they provide the best results, and seeing as all remain under 0.7 seconds; perhaps, time-permitting, it is best to start with the repeated random implementation.

5 Karmarkar-Karp as a Starting Point

From the data shown in Section 4.1, we find that the Karmarkar-Karp deterministic heuristic provides a meaningful improvement, *on average*, when compared to the outputs of our randomized algorithms and local search options (without prepartitions).

With regard to the repeated random algorithm, perhaps, we could use the Karmarkar-Karp solution as our base-line, and any overwriting solution will be required to be better.

Seeing as we have completely random assignments each time, starting with the Karmarkar-Karp solution does no harm.

In the other local search cases, however, we know that Karmarkar-Karp does not always return the optimal solution with minimized residue, and for Hill Climbing and Simulated Annealing, if we use the Karmarkar-Karp assignment (by performing a two-coloring and tracking set assignment) as a starting point, we may begin the search around a local minimum from which it is unlikely to escape (especially in the case of Hill Climbing).

In the case of the local search algorithms, it may not be wise to mix efforts and instead keep separate the starting position with a known base-line as returned by Karmarkar-Karp upon which to improve final residue. This way, we do not begin in the same local minimum as found by Karmarkar-Karp.