Analysis of Algorithms
CS 375, Fall 2018
Project 3
Due by the end of the day, Friday, December 7

Project 3: The Subset Sum Problem!

In this assignment, you'll work in teams of four—for you to choose on your own; please email me to let me know your team as soon as you've formed one[1]—to solve the *Subset Sum* problem 6 different ways! The purpose of this project is to give you practice understanding, implementing, testing, and analyzing the following kinds of algorithms:

1. exhaustive search
2. dynamic programming
3. greedy
4. random search
5. hill-climbing
6. simulated annealing

Definitions: Multiset

For this project, you'll work with the variant of the Subset Sum problem for *multisets*. So, before defining the Subset Sum problem, a definition of a multiset:

> **Definition**: A *multiset* is an unordered collection of elements, similar to a set, but with a significant difference: A multiset can contain the same item multiple times, whereas a set cannot. (See Levitin, page 36.) E.g., {1,1,2,3} is not a valid set, but it is a valid multiset.

The number of times an element x occurs in a multiset S is called the *multiplicity* of x in S. For examples, for $S_1 = \{i, j\}$, the elements i and j each have multiplicity 1 in S_1; for $S_2 = \{i, i, j\}$, the element i has multiplicity 2 in S_2 and element j has multiplicity 1 in S_2; and for $S_3 = \{i, i, i, j, j, j, k, k\}$, both i and j have multiplicity 3 in S_3 and k has multiplicity 2 in S_3.

With that, we can define the *cardinality* of a multiset S. Cardinality for multisets is a generalization of the idea of size for a set: The cardinality of multiset S is the sum of the multiplicities of its elements. For examples, given multisets S_1, S_2, S_3 defined above, the cardinality of S_1 is 2, the cardinality of S_2 is 3, and the cardinality of S_3 is 8.

In addition, we can define the important $\subseteq$ operation on multisets: For multisets S, T, $T \subseteq S$ exactly when, for every element x in T, the multiplicity of x in T is less than or equal to the multiplicity of x in S.

[1] If for any reason you can't find or form a team, let me know and I'll help you form one.

Definition: The Subset Sum Problem on Multisets

Using the above definitions, we can define Subset Sum on multisets:

> The *Subset Sum* problem has as input an integer k and a multiset S of integers; we'll let n stand for the cardinality of S. The output for Subset Sum is True exactly when there is a multiset $T \subseteq S$ such that when we add together all of the elements of T (accounting for their multiplicities in T), the sum is k; the output is False otherwise.

For example, consider $S = \{1, 1, 3, 9\}, k = 5$. In this case, the answer is True, because for $T = \{1, 1, 3\}$, $T \subseteq S$ and the elements of T sum to k. On the other hand, consider multiset $S = \{1, 2, 3, 9\}$ with $k = 8$. In this case, the answer is False—no collection of elements from S add up to 8. (Please ask me any questions you might have about this definition or these examples!)

For this project, we'll restrict the problem somewhat and assume that k is positive and all integers in S are positive.

Below and throughout this assignment, every time there is a reference to the Subset Sum problem ("Subset Sum", for short), it always refers to Subset Sum on multisets, not on sets. Moreover, below and throughout this assignment, every time there is a reference to a "subset" of some multiset S, that terminology will refer to some multiset T for which $T \subseteq S$, using the $\subseteq$ operation on multisets defined above.

The Assignment!

The assignment for this project has multiple parts.

1. Solve the Subset Sum problem (on multisets) 6 different ways:
   (a) Implement an *exhaustive search* algorithm that solves the problem exactly.
   (b) Implement a *dynamic programming* algorithm that solves the problem exactly.
   (c) Implement four *approximation algorithms* for the problem: one greedy, one random search, one hill-climbing, and one simulated annealing.
2. Test the four approximation algorithms to see how good of an approximation they give.
3. Test the speed of all six algorithms. This should be done in both of the following ways:
   (a) Using your understanding of the algorithms and the definitions of asymptotic complexity, give theoretical worst-case time complexity analyses for all 6 of your algorithms. (The more specific the complexity analyses are, the more credit they will earn.) Please explain your analyses, making sure the explanations demonstrate how the relevant definitions are applied.
   (b) In addition, empirically test the speed of your implementations of these algorithms for a variety of input values. (Use good code-testing practices, as always, to make sure the tested input values reflect the full range of possible inputs to your programs.) Create one or more tables comparing the speed of the algorithms and summarize your findings.

The algorithms are described in more detail below. Please use only Java or Python to implement your algorithms.

Exhaustive Search

The exhaustive search algorithm checks all possible multisets $T \subseteq S$ until finding a T (if any) with the appropriate sum. It should return True (it found such a multiset T) or False (it didn't).

It will be extremely slow. For maximum credit, come up with pruning techniques to shorten the search. For example, if all the multisets in the current branch of the search tree have a sum greater than the desired sum, then you can "prune" that branch of the search tree (i.e., ignore it).

Dynamic Programming

For input S and k, here's the dynamic programming algorithm to implement:

1. Order the elements of S (the input multiset with cardinality n) as $x_1, x_2, \ldots, x_n$.
2. Create a table Q with n rows and k columns.
3. Each entry in Q will be a boolean value. $Q[i][s]$ will have the value True if and only if there exists a multiset $T \subseteq \{x1, x2, ..., xi\}$ (i.e., T is fully included in the first i elements of S, in the ordering above) that has sum s. Otherwise, $Q[i][s]$ will have the value False.
4. Using that table, the solution to this instance of Subset Sum is the value of $Q[n][k]$, which your algorithm should return.
5. To compute $Q[n][k]$, use the recursive method following from this recurrence (be sure you understand why this recurrence works):
   - $Q[1][s] = (x_1 == s)$.
   - For $i > 1$, $Q[i][s] = (Q[i-1][s] == \text{True})$ or $(x_i == s)$ or $(Q[i-1][s-x_i] == \text{True})$.
6. Your implementation of this algorithm might waste space by creating an $n \times k$ matrix and then using only a few of the entries in the matrix. For extra credit, implement your algorithm so it doesn't waste such space and instead only uses as much memory as it needs.

Approximation Algorithms

The four approximation algorithms do not solve the problem exactly and so they should not return True or False. Instead, they should return the *residue*, which, in general terms, means the absolute value of the difference between the value found by the approximation algorithm and the desired value. In our case, the residue is the absolute value of the difference between the desired value k and the sum of the elements in the "subset" (actually a multiset, as noted in Section **Definition: The Subset Sum Problem on Multisets**, above) that the algorithm found.

Create one or more tables comparing the accuracy of the four approximation algorithms, and in the report you'll submit (see below), summarize your findings. To get data for these tables, create 50 instances of the Subset Sum problem and run all four approximation algorithms on each instance. Each instance should include a multiset S of 100 integers, each

a random number in the range 1 to 10^{12}. The value of k should be $25 \cdot 10^{12}$. (You will need to use type `long` instead of type `int` in a Java implementation of these algorithms.)

As noted in Section **What To Submit**, below, the report should include all of the relevant tables, charts, and data, along with a discussion of the algorithms, their speed, and their accuracy.

Greedy Algorithm

The greedy algorithm to implement is as follows:

1. Start with an empty subset T (actually a multiset) of S.
2. Sort multiset S from largest down to smallest.
3. Repeatedly try adding to T the next integer in the sorted list.
   (a) If the integer will not give T too large a sum, add it to T.
   (b) Otherwise, ignore that integer and move on to the next integer in the sorted list.
4. Return the residue of T.

Random Search Algorithm

The random search algorithm to implement is as follows:

1. This algorithm repeatedly chooses a random subset (multiset) T of S and checks its residue. It keeps track of the smallest residue it found as it is doing so.
2. The number of repetitions to use is provided as additional input to the algorithm.

Hill-Climbing

The *hill-climbing* algorithm to implement is as follows:

1. The algorithm should take four inputs: as always, there will be a multiset S and integer k, which are the "Subset" and "Sum" for the Subset Sum problem; in addition, there will be two integers q and r, with roles defined below.
2. Do the following q times:
   (a) Choose a random subset (multiset) S' of S as the "current" subset.
   (b) Do the following ("hill climbing") r times:
      i. Find a random *neighbor* T (see definition of *neighbor* below) of the current subset.
      ii. If neighbor T has smaller residue, then make T the current subset.
   (c) Keep track of the residue of the final "current" subset when starting with subset S'.
3. Return the smallest residue of the q subsets tested by the algorithm.

Definition: Subset (multiset) $B \subseteq S$ is a *neighbor* of a subset A of S if you can transform A into B by moving one or two integers from A to B, or by moving one or two integers from B to A, or by swapping one integer in A with one integer in B.

An easy way to generate a random neighbor B of a subset A of S is as follows:

1. Order the elements of S as $x_1, x_2, ..., x_n$.
2. Initialize B to be a clone of A.
3. Choose two distinct random indices i and j, where $1 \leq i, j \leq n$.
4. if x_i is in A, remove it from B. Otherwise, add x_i to B.
5. if x_j is in A, then with probability 0.5, remove it from B. If x_j is not in A, then with probability 0.5, add x_j to B.

Simulated Annealing

The simulated annealing algorithm to implement is as follows:

1. Choose a random subset (multiset) of S as the current subset.
2. Repeatedly do the following:
   (a) Find a random neighbor A of the current subset. (See the definition of neighbor above, along with an algorithm for finding a random neighbor.)
   (b) If neighbor A has smaller residue, then make it the current subset.
   (c) If neighbor A has larger or equal residue, then with probability e^{-X}, make it the current subset. (See below for the value of X.)
3. Return the smallest residue of all the subsets that at some time were the current subset.
4. The number of repetitions to use is provided as additional input to the algorithm.
5. The value of exponent X is computed using the following formula, where i is the current iteration, `current` is the residue of the current subset, and `neighbor` is the residue of the neighbor subset:

$$(\texttt{neighbor} - \texttt{current})/(10000000000 * 0.8^{i/300})$$

   In general, the constants to use in this formula vary depending on the application and the size of the problem, but for multisets of size 100 with integers in the range 1 to 10^{12}, this formula is appropriate.

What To Submit

Write a report including all relevant tables, charts, and data. Include a discussion of your algorithms, their speed, and their accuracy, along with details about the methods and analyses you employed to reach your conclusions. (In this report as in other scientific papers, enough detail should be provided so that a reader could independently replicate or validate the findings.) As always, part of your grade will be determined by the report's clarity, completeness, correctness, and depth of conceptual understanding and analysis demonstrated.

Submission instructions: Given the remarkable success we've had submitting to a Dropbox folder this semester, we'll try something different this time. This time, each group should have one "designated submitter" to handle submission of their materials.

Please create a zipped folder, containing your report (in PDF) and all of your code; please include all four of the team members' names in the name of the zipped folder and the filename for the PDF report. Then, the designated submitter should submit that folder in two ways:

1. Email a copy to me, just in case the other way is as successful as the Dropbox was; and

2. Submit a copy online to the designated submitter's `Private` folder in the CS375 filespace. (Please see me if additional instructions are needed on how to access that folder!)

In addition, each student is required to individually email me within a day after the project was turned in, telling me:

- how well your group worked together;
- how much (percentage) each member of your group contributed to the project; and
- how many hours you (individually) spent on this project.