SCHOOL OF ENGINEERING OF TECHNOLOGY

TCSS 343 – Programming Assignment

November 22, 2020

1 GUIDELINES

This is a group assignment. Each group may consist of 1, 2, or 3 students. In other words, you can choose to work by yourself, or with one colleague, or with two (but not more than two). Please state the names of your group members on your submission. All member of a group are required to upload their own copies of the joint work to Canvas, though.

Details: In this assignment, you will implement algorithmic techniques in **Java** or **Python**.

There is also a bonus challenge that also involves these techniques and an implementation in Java or Python.

Homework should be electronically submitted via Canvas before midnight on the due date. Each group is expected to submit the following files:

- Report. The submitted report MUST be typeset using any common software and submitted as a PDF. I strongly recommend using LaTeX to prepare your solution: you could use any LaTeX tools such as Overleaf, MiKTeX/TeXworks, TexShop etc. However, you afre free to use other tools like the embedded MS Word equation editor.

  When presenting your results, use log-scale graphics if/when results are too different (tiny vs. huge) to coexist on the same linear-scale graphic.

- Java/Python source code. Submit one single Java or Python file containing all your code, except for the challenge which must be in one single separate source file. You must name your source code **tcss343.java** or **tcss343.py** for the normal part of this assignment, and **challenge.java** or **challenge.py** for the bonus challenge.

Execution. We will use the following command(s) to execute your program for the normal part of this assignment:

```
java tcss343
```

or

```
python tcss343
```

The challenge has its own format described below.

Remember to cite all sources you use other than the programming assignment text, the course materials, or your own notes.

2 Problem Statement

Consider the Subset Sum Problem, formally defined as:

Subset Sum (SS):
INPUT: a list $S[0 \ldots n-1]$ of n positive integers, and a target integer $t \geq 0$.
OUTPUT: TRUE and a set of indices $A \subseteq \{0 \ldots n-1\}$ such that $\sum_{i \in A} S[i] = t$ if such an A exists, or FALSE otherwise.

This problem can be solved by brute force or with dynamic programming. The following clever algorithm has also been proposed to solve SS:

1. Split the indices $\{0 \ldots n-1\}$ into two sets of nearly equal size, $L = \{0 \ldots \lfloor n/2 \rfloor\}$ and $H = \{\lfloor n/2 \rfloor + 1 \ldots n-1\}$.
2. Compute a table T of all subsets of L that yield a subset of S of weight not exceeding t (that is, a table containing all $I \subseteq L$ such that $\sum_{i \in I} S[i] \leq t$). If equality holds for some I, i.e. $\sum_{i \in I} S[i] = t$, then return TRUE and I, and stop.
3. Compute a table W of all subsets of H that yield a subset of S of weight not exceeding t (that is, a table containing all $J \subseteq H$ such that $\sum_{j \in J} S[j] \leq t$). If equality holds for some J, i.e. $\sum_{j \in J} S[j] = t$, then return TRUE and J, and stop.
4. Sort table W in ascending order of weights.
5. For each entry I in table T, find the subset $J \subseteq H$ that yields the maximum weight not exceeding t when joined to I, i.e. $\left(\sum_{i \in I} S[i]\right) + \left(\sum_{j \in J} S[j]\right) \leq t$. If equality holds for some I and some J, i.e. $\left(\sum_{i \in I} S[i]\right) + \left(\sum_{j \in J} S[j]\right) = t$, then return TRUE and $I \cup J$, and stop.
6. If no subsets I and J yield equality, return FALSE and the empty set, and stop.

The overall goal of this programming assignment is to test and compare these three proposed solutions.

3 Your Tasks

3.1 Brute Force

(**4 points**): Design and implement a brute force solution for this problem. Run it according to the testing procedure described below in section 3.4.

What is the asymptotic running time complexity of this algorithm? What is its asymptotic space complexity (memory requirements for tables and similar data structures)? Justify your complexity analysis.

Solution:

For every subset of S, compute its sum and compare it against t.

Since there are 2^n subsets, each containing up to n elements (so that evaluating each sum costs $O(n)$ additions), the complexity is $\Theta(n2^n)$ arithmetic operations.

Each subset can be represented as a sequence of n bits (meaning that each element either is or is not in the subset), and the sum of each fits $\lceil \lg t \rceil$ bits, which is the space needed to represent t itself (any sum exceeding this would be too large and could be disregarded anyway). Since $\lg t \in O(n)$ (that is, the space needed to store t is in the same ballpark as the space needed to store the remaining input), the space complexity is $O(n)$ bits, or $O(1)$ elements. □

3.2 Dynamic Programming (DP)

(**6 points**): Design and implement a DP solution for this problem. Run it according to the testing procedure described below.

What is the asymptotic running time complexity of this algorithm? What is its asymptotic space complexity (memory requirements for tables and similar data structures)? Justify your complexity analysis.

Solution:

The DP algorithm has been analyzed in the corresponding lecture. Since it consists of two nested loops, one of n steps and one of t steps, the total number of steps (each taking constant time) is nt, and the time complexity is $\Theta(nt)$ Because $\lg t \in O(n)$ as pointed out above, this simplifies to $\Theta(n2^n)$ arithmetic operations.

The algorithm requires an $n \times t$ matrix of integers large enough to store a value comparable to t, that is, $O(\lg t) = O(n)$ bits. Hence the space complexity is $O(nt \lg t) = O(n^2 2^n)$ bits, or $O(n2^n)$ elements. □

3.3 Clever algorithm

(**8 points**): Design and implement a solution for this problem based on the (allegedly) clever algorithm.

What is the asymptotic running time complexity of this algorithm? What is its asymptotic space complexity (memory requirements for tables and similar data structures)? Justify your complexity analysis.

Solution:

Step 1 clearly takes no more than $O(n)$ time, since it only needs to split a list of size n. Step 6, which is even simpler, costs only $O(1)$.

Step 2 scans over all subsets $I \subseteq L$. Since L contains $n/2$ elements, there are $2^{n/2}$ such subsets, and for each one the algorithm computes $\sum_{i \in I} S[i]$ consisting of up to $n/2$ elements. Thus, this step performs $\Theta(n2^{n/2})$ additions.

By the same token as step 2, step 3 performs $\Theta(n2^{n/2})$ additions.

Since the table size is $N = 2^{n/2}$, step 4 can be implemented with a $\Theta(N \lg N)$ sorting algorithm (like heapsort), and the cost is hence $\Theta(n2^{n/2})$ comparisons.

In step 5, finding the subset $J \subseteq H$ with maximum weight not exceeding $t - \text{weight}(I)$ can be performed with binary search on table W at cost $\Theta(\lg N)$. Since the table size is $N = 2^{n/2}$, the cost is $\Theta(N)$ comparisons per search, and since a search is performed for each of the $2^{n/2}$ elements of $I \subseteq L$, the overall cost of this step is $\Theta(n2^{n/2})$ comparisons.

Summing up, the total time complexity is $\Theta(n2^{n/2})$ arithmetic operations.

This algorithm requires two tables of subsets of lists of length $n/2$ elements. There are $2^{n/2}$ such subsets for each set, totaling $2 \cdot 2^{n/2}$ list entries. Each subset can be represented by a sequence of $n/2$ bits, so the overall space is $(n/2) \cdot 2 \cdot 2^{n/2}$, or $O(n2^{n/2})$ bits, or simply $O(n2^{n/2})$ elements. □

3.4 Testing

(**4 points**): Design and implement a method `Driver` that, given two integers n and r and a Boolean value v as input, creates a sequence S of n random elements sampled from range 1 to r, then tests each of the above algorithms on S for a target t chosen as, for $v =$ TRUE, the sum of a random subset of S (so that a solution is guaranteed to exist), and for $v =$ FALSE, a random value larger than the sum of all values on S (so that there is no solution).

The `Driver` method must measure the time (in milliseconds) each of the three algorithms takes to complete, and record how much table space each algorithm needs for that particular combination of n and t. When the tests are done, `Driver` must also print the values of n and r, the generated sequence S and, for each of the three algorithms, the subsequence of elements from S each algorithm found that sums up to t (or an indication that none such subsequence exists), and the running time and table space requirements of that algorithm for that S.

Run `Driver` for each of the following combinations:

- $r = 1{,}000$, $v =$ TRUE or FALSE (test both), and $n = 5, 6, 7, \ldots$ up to the largest value of n each algorithm can test so that it takes no more than 5 minutes for that n;
- $r = 1{,}000{,}000$, $v =$ TRUE or FALSE (test both), and $n = 5, 6, 7, \ldots$ up to the largest value of n each algorithm can test so that it takes no more than 5 minutes for that n.

Notice that the largest value of n may be different for each algorithm. If a certain combination of n, r, and v is infeasible for some algorithm, skip that algorithm but test the other one(s).

3.5 Analysis of results

(**4 points**): Analyze the results using visualization (you may use MS Excel or equivalent spreadsheet software for that).

Plot the running time (in milliseconds) as a function of n for the range of n values described above and for each r and v.

Also plot the table sizes (in number of entries) as a function of n for the range of n values described above and for each r and v.

Use log-scale graphics whenever the discrepancy between the cases is so large than linear-scale graphics will make it too hard to compare those cases visually, or if the graphics simply do not fit the visual space.

Your analyses should be included in the submitted document.

3.6 Documentation

(**4 points**): Provide a well prepared document that describes your solutions, complexity analyses, and result analyses including the graphics.

You must submit your code, which should be well documented as well. If there is any known error in the code, you must point that out.

Also, document the division of labor (who did what) in your report.

Remember: your presentation must briefly describe all of these items.

4 Bonus challenge (5 points): the d-dimensional n-Queens problem

The well-known 8-Queens problem consists of placing 8 queens on an otherwise empty chessboard in such a way that no two queens attack each other (hence, there are no two queens on the same row, column, or diagonal, and no other pieces on the chessboard), as shown in fig. 4.1.

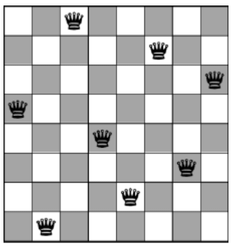

Figure 4.1: Sample solution to the 8-Queens problem

A trivial generalization is the n-Queens problem, where the task is to place n queens on an $n \times n$ board instead. This problem has real-world applications like managing distributed memory storage or preventing deadlocks in computer networks.

A not-so-trivial generalization is the d-dimensional n-Queens problem, which simply asks how many queens can be placed on a d-dimensional chessboard of size $n \times n \times \cdots \times n$ (d times, i.e. n^d cells). The answer is easy for $d = 2$, since one can place exactly n queens on a 2-dimensional chessboard in non-attacking configuration (every queen must be alone on each row and each column, and there are n rows or columns on a 2-dimensional chessboard).

But for $d > 2$ the answer is not known exactly, and in general has to be determined by direct inspection. For instance, it is possible to place 4 non-attacking queens on a 3-dimensional chessboard of size $3 \times 3 \times 3$.

4.1 Formalization

In general, a queen on a d-dimensional chessboard of size n^d is specified by a tuple $q := (q_0, q_1, \dots, q_{d-1})$ of coordinates where $0 \le q_j < n$ for all $0 \le j < d$.

Let $\delta := (\delta_0, \delta_1, \dots, \delta_{d-1})$ be a tuple of integers from the set $\{-1, 0, 1\}$ where not all of the δ_j are zero. Such a tuple is called an attack vector. For any integer $0 < s < n$, the queen at position q is said to be in attacking configuration along the attack vector δ against all chessboard positions of form $q + s\delta = (q_0 + s\delta_0, q_1 + s\delta_1, \dots, q_{d-1} + s\delta_{d-1})$, as long as none of these coordinates extrapolate the boundaries of the chessboard, that is, if $0 \le q_j + s\delta_j < n$ for all $0 \le j < d$.

Thus, another queen $p := (p_0, p_1, \dots, p_{d-1})$ is in a mutual attacking configuration with respect to q if $p = q + s\delta$ for some attack vector δ_{d-1} and some integer $0 < s < n$.

For instance, for $d = 2$ the mutually attacking configurations are:

- queens on the same row:

$$\begin{aligned} (p_0, p_1) &= (q_0, q_1) + s(0, -1) = (q_0, q_1 - s), \\ (p_0, p_1) &= (q_0, q_1) + s(0, 1) = (q_0, q_1 + s), \end{aligned}$$

- queens on the same column:

$$\begin{aligned} (p_0, p_1) &= (q_0, q_1) + s(-1, 0) = (q_0 - s, q_1), \\ (p_0, p_1) &= (q_0, q_1) + s(1, 0) = (q_0 + s, q_1), \end{aligned}$$

- queens on the same main diagonal:

$$\begin{aligned} (p_0, p_1) &= (q_0, q_1) + s(-1, -1) = (q_0 - s, q_1 - s), \\ (p_0, p_1) &= (q_0, q_1) + s(1, 1) = (q_0 + s, q_1 + s), \end{aligned}$$

- queens on the same anti-diagonal:

$$\begin{aligned} (p_0, p_1) &= (q_0, q_1) + s(1, -1) = (q_0 + s, q_1 - s), \\ (p_0, p_1) &= (q_0, q_1) + s(-1, 1) = (q_0 - s, q_1 + s), \end{aligned}$$

for some $0 < s < n$.

The d-dimensional n-Queens problem is then to determine how many queens can be placed on a d-dimensional chessboard of size n^d in such a way that no two queens are in mutual attack configuration.

4.2 The task (finally!)

Write a program that solves the d-dimensional n-Queens problem for a dimension d and board size $n \times n \times \cdots \times n$ where d and n are arbitrary inputs to the program.

Test your program for all combinations of dimensions $3 \le d \le d_{\text{max}}$ and board sizes $3 \le n \le n_{\text{max}}$ for the largest possible d_{max} and n_{max} that constrain the running time to within a reasonable limit (say, 15 minutes, but you are free to try longer times), and plot the observed running times on a graph as a function of d and n.