# TCSS 343 – Programming Assignment

Nov 6th, 2024

## Guidelines

**This is a group assignment.** Each group may consist of either 1 to 3 students. In other words, you can choose to work by yourself, or with one colleague, or with two colleagues (but no more than this). *Please state the names of all your group members on your submission*. Also, all members of a group are required to upload their own copies of the joint work to Canvas.

**Details:** In this assignment, you will implement algorithmic techniques in **Java** (recommended) 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 *in a single zip file*. Each group is expected to submit the following material:

- **Report.** The submitted report *MUST* be typeset using any common software and submitted as a PDF. You can use any LaTeX tools such as Overleaf, MiKTeX/TeXworks, TexShop, etc. You are also free to use other tools like the embedded MSWord equation editor.
  When plotting your results in the report, *use log-scale charts and graphics* if/when results are too contrasting (tiny vs. huge) to coexist on the same linear-scale graphic.
- **Java or Python source code.** Prepare 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. NB: you must submit *ONLY the source file(s)* – you will be docked one point for each `.class` or `.jar` or similar non-source file included in the submitted zip file.

**Execution.** The following command(s) must be used 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. Failing to do so would constitute plagiarism.**

## Problem statement

Consider the *Subset Sum Problem*, formally defined as:

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

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 $\{0 \dots n-1\}$ into two sets of nearly equal size, $L = \{0 \dots \lfloor n/2 \rfloor\}$ and $H = \{\lfloor n/2 \rfloor + 1 \dots n-1\}$.
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. $(\sum_{i \in I} S[i]) + (\sum_{j \in J} S[j]) \leq t$. If equality holds for some $I$ and some $J$, i.e. $(\sum_{i \in I} S[i]) + (\sum_{j \in J} S[j]) = t$, then return TRUE and $I \cup J$, and stop.
6. If no subsets $I$ and $J$ yield equality, return FALSE, and stop.

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

## Your tasks

**1. BRUTE FORCE (4 points):** Design and implement a brute force 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.

**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.

**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. *Hint: before you do the complexity analysis, ask yourself these two questions: (1) Does the clever algorithm follow a divide-and-conquer approach? (2) Does it really? —or is it perhaps something else entirely?*

**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, … 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, … 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).

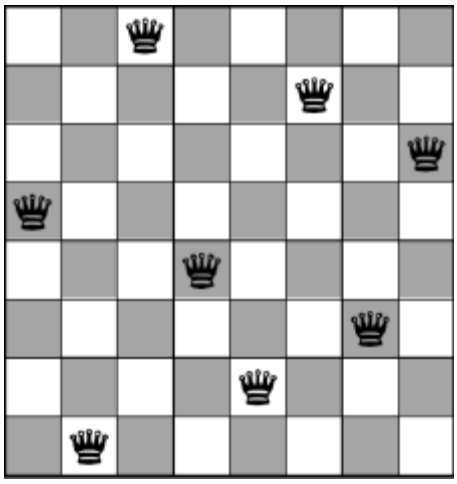
**5. ANALYSIS OF RESULTS (4 points):** Analyze the results using visualization (you may use MS Excel or equivalent spreadsheet software for that).

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 that 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 analysis should be included in the submitted document.

**6. DOCUMENTATION (4 points):** Provide a well-prepared document that describes your solutions, complexity analysis, and result analysis 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 report must briefly describe all these items*.

## Bonus challenge

**THE $d$-DIMENSIONAL $n$-QUEENS PROBLEM (5 bonus points):** 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 the figure below:



A 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 $n \times n \times \ldots \times n$ ($d$ times, thus consisting of $n^d$ 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 must be determined by direct inspection. For instance, it is possible to place 4 non-attacking queens on a 3-dimensional chessboard of size $3 \times 3 \times 3$, exceeding the expected 3 non-attacking queens.

**FORMALIZATION:** In general, a queen on a $d$-dimensional chessboard of size $n^d$ is specified by a tuple $q := (q_0, q_1, \ldots, q_{d-1})$ of coordinates where $0 \leq q_j < n$ for all $0 \leq j < d$. Let $\delta := (\delta_0, \delta_1, \ldots, \delta_{d-1})$ be a tuple of integers from the set $\{-1, 0, 1\}$ where not all of the $\delta_j$ 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 $\delta$ against all chessboard positions of form $q + s\delta = (q_0 + s\delta_0, q_1 + s\delta_1, \ldots, q_{d-1} + s\delta_{d-1})$, as long as none of these coordinates extrapolate the boundaries of the chessboard, that is, if $0 \leq q_j + s\delta_j < n$ for all $0 \leq j < d$.

Thus, another queen $p := (p_0, p_1, \ldots, p_{d-1})$ is in a mutual attacking configuration with respect to $q$ if $p = q + s\delta$ for *some* attack vector $\delta_{d-1}$ and *some* integer $0 < s < n$ (thus usually one must check all potential attack vectors $\delta_{d-1}$ and all potential integers $s$ in that range). For instance, for $d = 2$ the mutually attacking configurations are:

- queens on the same row:
  $(p_0, p_1) = (q_0, q_1) + s(0, -1) = (q_0, q_1 - s)$,
  $(p_0, p_1) = (q_0, q_1) + s(0, 1) = (q_0, q_1 + s)$,
- queens on the same column:
  $(p_0, p_1) = (q_0, q_1) + s(-1, 0) = (q_0 - s, q_1)$,
  $(p_0, p_1) = (q_0, q_1) + s(1, 0) = (q_0 + s, q_1)$,
- queens on the same main diagonal:
  $(p_0, p_1) = (q_0, q_1) + s(-1, -1) = (q_0 - s, q_1 - s)$,
  $(p_0, p_1) = (q_0, q_1) + s(1, 1) = (q_0 + s, q_1 + s)$,
- queens on the same anti-diagonal:
  $(p_0, p_1) = (q_0, q_1) + s(1, -1) = (q_0 + s, q_1 - s)$,
  $(p_0, p_1) = (q_0, q_1) + s(-1, 1) = (q_0 - s, q_1 + s)$,

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.

**THE TASK (FINALLY!):** Write a program that solves the $d$-Dimensional $n$-Queens problem for a dimension $d$ and board size $n \times n \times \ldots \times n$ where $d$ and $n$ are arbitrary inputs to the program.

Test your program for all combinations of dimensions $2 \leq d \leq d_{\max}$ and board sizes $2 \leq n \leq n_{\max}$ 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 $n$.

**Grading of the bonus challenge:** This is an *all-or-nothing* item: you get either full points or none. You will fail to get the bonus points in case of errors in your math, code disorganization and/or lack of clarity, or incomplete work.