Brute-Force Algorithmic Approach

Solutions to the Practical Exercises

DA 2023 Instructors Team

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

Exercise 1

a) See source code.
b) See source code.

The improved solution generates candidate triplets (a,b,c) in a smarter manner:

- it generates all possible T values for A, between 1 and T.
- for each value of a, it generates all possible values for B, between 1 and (a - 1), to avoid symmetric solutions (given that (a,b,c) is a solution if and only if (b,a,c) because the addition operation is commutative).
- it does not need to test all possible values for c, since the correct value for c can be inferred, since $a + b + c = T$ is equivalent to $c = T - a - b$. All that is needed to have a valid solution is to check whether c > 0.

c) In the worst case, there is no solution, so the algorithm cannot stop earlier and must test all possible solutions.

The number of different candidates for the pair (a,b) is:

$$T + (T - 1) + (T - 2) + \ldots + 3 + 2 + 1$$

This value is due to there being T valid values for b when a = 1, (T-1) values for b when a = 2, and so on. The above sum is equivalent to:

$$\frac{T * (T + 1)}{2}$$

Testing each candidate takes O(1) time, as no operation in the innermost for-loop scales with T. Thus, the temporal complexity is, in the worst-case $O(T*(T + 1)/2) = O(T*(T + 1)) = O(T^2 + T) = O(T^2)$.

Exercise 2

a) The idea of the Brute-force solution is to represent the candidate solutions as a bitmask (*i.e.,* array of booleans) with the same length as the input array A. A value of **`true`** in a given position i of the bitmask means that the corresponding candidate solution uses the i-th element of A. Using a bitmask is useful as it makes it intuitive how to enumerate all the candidate solutions (*i.e.,* all the possible subsets).

Computing the next value of the bitmask intuitively corresponds to incrementing a binary number by 1, where the numbers are represented backwards (*i.e.*, the least significant bit is the leftmost bit, not the rightmost bit). Example of the enumeration of the 8 candidates for an array of length 3: 0 0 0 → 1 0 0 → 0 1 0 → 1 1 0 → 0 0 1 → 1 0 1 → 0 1 1 → 1 1 1

Pseudo-code for finding a subset with the correct sum:

Input:

- A: array of non-negative numbers
- T: desired sum

Output:

- boolean indicating if the subset exists, subset of A whose sum is T

```
subsetSum(A, T)
        // Create the first candidate
        for(i = 0; i < A.size(); i++)
                curCandidate[i] ← false

        // Iterate over all the candidates
        while(true)
           // Verify if the candidate is a solution
           sum ← 0
           for(i = 0; i < A.size(); i++)
                sum ← sum + A[i]*curCandidate[i] //only sums A[i] if curCandidate[i] = true
                if(sum = T)
                        // Build and return the solution
                        for(i = 0; i < A.size(); i++)
                            if(curCandidate[i])
                                subset.push(A[i])
                        return true, subset
                // Get the next candidate
                curIndex ← 0
                while(curCandidate[curIndex])
                        curIndex++
                        if(curIndex = A.size()) break
```

```
            if(curIndex = A.size()) break
            for(i = 0; i < curIndex; i++)
                    curCandidate[i] ← false
            curCandidate[curIndex] ← true
    return false, []
```

b) The temporal complexity of the algorithm in the worst case is $O(n*2^n)$, where n is array A's length. This is due to there being at most 2^n candidates to try out and processing each candidate takes O(n) since a fixed amount of for-cycles executes over n iterations at most.

   The spatial complexity of the algorithm is $\Theta(n)$, not only for the worst-case, but for any other case. This is due to space always being allocated to the current candidate, whose size scales with n. If there is a valid solution, then space is also allocated for the solution, which also scales with n. All the other variables have a fixed size with respect to n.

c) See source code.

Exercise 3

a) See source code.

b) The algorithm has a temporal complexity of $\Theta(n^3)$ since all n^2 possible contiguous subarrays are tested and processing each one takes $\Theta(n)$ time. This temporal complexity is valid for the best, worst and average cases, since all subarrays are tested, no matter what.

Exercise 4

a) See source code. The code contains comments explaining in detail the steps of the solution.

b) The algorithm has a temporal complexity of $O(n*S^n)$ since there are at most S^n candidate solutions (if all n coin denominations have an equal stock of S) and processing each one takes $\Theta(n)$ time (namely to obtain a candidate's total value and number of coins used).

Exercise 5

a) It is a solution to the **closest pair problem**, where the goal is to discover the two points that minimize the Euclidean distance between them, among a set of 2D points.
b) The algorithm uses a brute-force strategy since it exhaustively generates all possible solutions (i.e. pairs of points) and tests if they are better than the best solution found so far (*i.e.,* the pair of points that minimizes the Euclidean distance).
c) The temporal complexity is $\Theta(n^2)$, for the best, average and worst cases, given that all n^2 possible pairings of points are always tested and each one is processed in O(1) (comparing two points and evaluating the Euclidean distance takes constant time).
d) See source code.

Exercise 6

a) See source code.
b) The algorithm has a temporal complexity of $\Theta(n*2^n)$ since there are 2^n candidate solutions and processing each one takes $\Theta(n)$ time (namely to obtain a candidate's total value and weight). This temporal complexity is valid for the best, worst and average cases, since all subarrays are tested, no matter what.

Exercise 7

a) See source code.
b) The algorithm has a temporal complexity of $\Theta(n*n!)$ since all possible permutations of the path are considered and processing each one takes $\Theta(n)$ time. This temporal complexity is valid for the best, worst and average cases.