Brute-Force Algorithmic Approach

Practical Exercises

Gonçalo Leão

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

Spring 2023

Exercise 1
Consider the function *sum3* below.

```
bool sum3(unsigned int T, unsigned int selected[3])
```

The function finds three positive integers whose sum is equal to the value of the argument `T`. The function returns **`True`** and initializes the *`selected`* array with the three integers that add up to `T`. Otherwise, the function returns **`False`** (and the array named *`selected`* is not initialized).

Input example: `T` = 10
Expected result: `selected` = {1, 1, 8}, ..., `selected` = {2, 3, 5}, …

a) Implement *sum3* using an exhaustive search strategy (i.e., Brute-force) with $O(T^3)$ temporal asymptotic complexity.
b) Improve the temporal efficiency of *sum3* by implementing a more efficient, but still brute-force approach with a lower temporal complexity.
c) Indicate and justify the temporal complexity of the improved algorithm, in the worst case, with respect to `T`.

Exercise 2

The **subset sum problem** consists of determining if there is a subset of a given array of non-negative numbers `A` whose sum is equal to a given integer `T`.

Note: To simplify this exercise, and should multiple subsets be valid, returning any of them will be acceptable. Any order for the valid elements of a subset is also acceptable.

Input example: `A` = [3, 34, 4, 12, 5, 2], `n`=6, `T` = 9
Expected result: [4, 5]

Input example: `A` = [3, 34, 4, 12, 5, 2], `n`=6, `T` = 20
Expected result: [3, 12, 5]

Input example: `A` = [3, 34, 4, 12, 5, 2], `n`=6, `T` = 30
Expected result: no solution

a) Propose in pseudo-code a Brute-force algorithm solution for this problem. Your algorithm should return two outputs: a boolean indicating if the subset exists and, if so, the subset itself.
b) Indicate and justify the algorithm's temporal and spatial complexity, in the worst case, with respect to the array's size, n.
c) Implement *subsetSumBF*, which executes your algorithmic solution.

```
bool subsetSumBF(unsigned int A[], unsigned int n, unsigned int T,
                 unsigned int subset[], unsigned int &subsetSize)
```

Exercise 3

Given any one-dimensional array `A`[1..*n*] of integers, the **maximum sum subarray problem** tries to find a contiguous subarray of `A`, starting with element *i* and ending with element *j*, with the largest sum: $max \sum_{x=i}^{j} A[x]$ with $1 \leq i \leq j \leq n$. Consider the *maxSubsequenceBF* function below.

```
int maxSubsequenceBF(int A[], unsigned int n , int &i, int &j)
```

The function returns the sum of the maximum subarray, for which *i* and *j* are the indices of the first and last elements of this subsequence (respectively), starting at 0. The function uses an exhaustive search strategy (*i.e.*, Brute-force) so as to find a subarray of `A` with the largest sum, and updates the arguments *i* and *j*, accordingly.

Input example: A = [−2, 1, −3, 4, −1, 2, 1, −5, 4]
Expected result: [0, 0, 0, 1, 1, 1, 1, 0, 0], as subsequence [4, −1, 2, 1] (i = 3, j = 6) produces the largest sum, 6.

a) Implement *maxSubsequence* using a brute-force strategy.
b) Indicate and justify the temporal complexity of the algorithm, with respect to the array's size, n.

Exercise 4

The **change-making problem** is the problem of representing a target amount of money, T, with the fewest number of coins possible from a given set of coins, C, with *n* possible denominations (monetary value). Consider the function *changeMakingBF* below, which considers a limited stock of coins of each denomination C_i, in Stock, respectively.

```
bool changeMakingBF(unsigned int C[], unsigned int Stock[],
 unsigned int n, unsigned int T, unsigned int usedCoins[])
```

The arguments C and Stock are unidimensional arrays of size *n*, and *T* is the target amount for the change. The function returns a boolean indicating whether or not the problem has a solution. If so, it set the *usedCoins* array with the total number of coins used for each denomination C_i.

Input example: C = [1, 2, 5, 10], Stock = [3, 5, 2, 1], n =4, T = 8
Expected result: [1, 1, 1, 0]

Input example: C = [1, 2, 5, 10], Stock = [1, 2, 4, 2], n =4, T = 38
Expected result: [1, 1, 3, 2]

a) Implement *changeMakingBF* using a brute force strategy.
b) Indicate and justify the temporal complexity of the algorithm, in the worst case, with respect to the number of coin denominations, n, and the maximum stock of any of the coins, S.

Exercise 5
Consider the following pseudo-code:

```
mystery_func(S):
   d ← +∞
   p1 ← NULL
   p2 ← NULL
   foreach (x in S)
      foreach (y in S)
         if(x ≠ y && euclidean_dist(x,y) < d)
            d ← euclidean_dist(x,y)
            p1 ← x
            p2 ← y
return d, p1, p2
```

a) What does this algorithm do?
b) Explain why the algorithm follows a brute-force approach.
c) Derive and justify the temporal complexity of the algorithm, with respect to the array S's size, n.
d) Implement the *mysteryFunc* in C++.

```
std::vector<int> mysteryFunc(const std::vector<int> &A)
```

Exercise 6
The **0-1 Knapsack problem** consists in selecting a subset of items from a set so that the total value is maximized without exceeding the Knapsack's maximum weight capacity. Each item has a value and a weight. Each item can only be placed in the Knapsack at most once.

Consider the *knapsackBF* function below, which solves the 0-1 knapsack problem.

```
unsigned int knapsackBF(unsigned int values[], unsigned int weights[],
      unsigned int n, unsigned int maxWeight, bool usedItems[])
```

Input example: `values` = [10, 7, 11, 15], `weights` = [1, 2, 1, 3], `n` = 4, `maxWeight` = 5
Expected result: [1, 0, 1, 1] (the total value is 10 + 11 + 15 = 36)

Input example: `values` = [1, 2, 5, 9, 4], `weights` = [2, 3, 3, 4, 6], `n` = 5, `maxWeight` = 10
Expected result: [0, 1, 1, 1, 0] (the total value is 2 + 5 + 9 = 16)

a) Implement *knapsackBF*, which using a brute-force strategy.
b) Derive and justify the temporal complexity of the algorithm, with respect to the number of items, n.

Exercise 7

The **Traveling Salesman Problem (TSP)** consists in finding the path in a weighted graph that traverses each vertex once and exactly once and then returns to the initial node such that the sum of the weights of the edges used is minimized.

To simplify, consider that the initial city is always node 0 and that the graph is complete (*i.e.*, it has one edge connecting every node to every other node). The distances may not be symmetric, that is, the distance from node A to B may be different from the distance from B to A.

Consider the *tspBF* function below, which solves the TSP problem.

```
unsigned int tspBF(const unsigned int **dists, unsigned int n, unsigned
                                        int path[])
```

Input example: `dists` = [[0, 10, 15, 20], [5, 0, 9, 10], [6, 13, 0, 1], [8, 8, 9,0]], **n**=4
Expected result: 35, `path` = [0, 1, 3, 2]

a) Implement *tspBF*, using a brute-force strategy.
b) Derive and justify the temporal complexity of the algorithm, with respect to the number of vertices, n.

Suggestion: Iterate through all possible path permutations. The next permutation can be obtained using *std::next_permutation* from the <*algorithm*> library:
https://cplusplus.com/reference/algorithm/next_permutation/