Brute-Force and Greedy Algorithmic Approach

Practical Exercises

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

Spring 2025

Exercise 1

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 `maxSubSequence` 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). 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. Is it possible to implement an algorithm with a quadratic temporal complexity? If yes, implement it.
b) Indicate and justify the temporal complexity of the algorithm, with respect to the array's size, n.

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 **`subsetSum`**, using a simple brute-force approach.
d) Could you think of an improvement to your execution time, but being clever about when to give up the exploration? **Hint:** What if the elements of the array are sorted? Optimize your previous code solution.

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

Exercise 3

Consider the **0-1 Knapsack problem**. This 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 uses a brute-force strategy.
b) Derive and justify the temporal complexity of the algorithm, with respect to the number of items, n.
c) Try using the simplest greedy strategy of selecting the most valuable item first. Will this yield in general the optimal solution? If not, present a counter-example. Implement the knapsackGreedyValue method, why the unitary tests fail?
d) Repeat c) with the greedy strategy of picking the lightest element first. Will this yield in general the optimal solution? If not, present a counter-example. Implement the knapsackGreedyWeight method, why the unitary tests fail?

Exercise 4

Consider the **fractional knapsack problem**. This is a variant of the 0-1 knapsack problem where only a percentage of an item can be placed in the knapsack. For instance, if an item has a value of 4 and a weight of 3 and only 50% of the item is used, then it adds a value of 2 and a weight of 1.5 to the knapsack. Consider the function `fractionalKnapsack` below.

```
double fractionalKnapsackGR(unsigned int values[], unsigned int
weights[], unsigned int n, unsigned int Weight, double usedItems[])
```

Input example: **values** = [60, 100, 120], **weights** = [10, 20, 30], **n** = 3, **Weight** = 50

Expected result: [1, 1, $\frac{2}{3}$] (the total value is 60*1 + 100*1 + 120*$\frac{2}{3}$ = 240)

a) Formalize this problem.
b) Implement `fractionalKnapsackGR` using a greedy strategy.
c) Indicate and justify the algorithm's time complexity, in the worst case, with respect to the number of items, n.
d) Prove that this greedy strategy leads to the optimal solution.

Exercise 5

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 `changeMaking` 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 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 `changeMaking`BF 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.
c) Develop a greedy variant of your algorithm. Implement the method changeMakingGR, which uses a greedy approach. Since you want to minimize the number of coins, maybe choosing the coins of the largest denomination first will lead to the optimal solution? Will this work for these examples? and in general? What properties does your currency system need to have for this greedy strategy to work?