Brute-Force and Greedy Algorithmic Approach

Practical Exercises

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

Spring 2024

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 maxSubsequence(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.

The provided maxSubsequence function solves the maximum sum subarray problem using a brute-force approach. It iterates through all possible subarrays and calculates their sums, updating the maximum sum and corresponding indices i and j whenever a new maximum sum is found.

Temporal Complexity Analysis:
Let's analyze the temporal complexity of the algorithm:

Outer Loop (start): The outer loop iterates n times, where n is the size of the input array A[]. This loop represents the starting index of the subarray.

Inner Loop (end): The inner loop also iterates n times in the worst case. It represents the ending index of the subarray and starts from the current starting index determined by the outer loop.

Operations inside the loops: Inside the nested loops, the only operations are basic arithmetic operations (addition, comparison) and updating values, all of which are constant time operations (O(1)).

Total Complexity:
The total number of iterations performed by the nested loops is approximately n^2. Since the operations inside the loops are constant time, the temporal complexity of the algorithm is quadratic in terms of the size of the input array n.

Therefore, the temporal complexity of the maxSubsequence function is O(n^2).

Despite being a straightforward brute-force solution, this algorithm might not be the most efficient for very large arrays due to its quadratic complexity. For larger inputs, more efficient algorithms like Kadane's algorithm, which achieves linear time complexity (O(n)), would be more suitable.

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

b) Temporal Complexity : O(2^n) . The spatial complexity is relatively low and doesn't depend on the input size, apart from some constant overhead for auxiliary variables.
d) One improvement to the subset sum problem, especially when the elements of the array are sorted, is to use a technique called binary search along with backtracking. Sort the Array. Backtracking with Binary Search: Start by selecting elements from the sorted array and checking if they contribute to the sum. If the current sum exceeds the target sum T, then backtrack to the previous element and explore other possibilities. Here, we can use binary search to efficiently find the position of the next element to consider. Subset Size

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?

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

a)

```
Function subsetSumExists(A[], n,
T):
    for each subsetSize from 0 to n:
        for each combination C of subsetSize elements from A:
            if sum(C) equals T:
                return true
    return false

Function findSubset(A[], n, T):
    for each subsetSize from 0 to n:
        for each combination C of subsetSize elements from A:
            if sum(C) equals T:
                return C
    return empty subset
```

Explanation:

subsetSumExists: This function iterates through all possible subsets of different sizes and checks if the sum of any subset equals T. If such a subset is found, it returns true; otherwise, it returns false.
findSubset: This function is similar to subsetSumExists, but instead of returning just a boolean, it returns the subset itself if it exists, otherwise an empty subset.
This brute-force approach checks all possible combinations of elements in the array, making it quite inefficient for large arrays due to its exponential time complexity. However, it provides a straightforward way to understand the problem and serves as a baseline for more efficient algorithms

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

```
unsigned int integerKnapsack(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 *integerKnapsack*, 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.
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.

b) The function it explores all possible combinations of items to find the subset that maximizes the total value without exceeding the maximum weight capacity of the knapsack. Subset Generation: The function uses a recursive approach to generate all possible subsets of items. For each item, there are two choices: include it in the knapsack or exclude it. Therefore, there are a total of 2^n subsets that need to be explored, where n is the number of items.Subset Evaluation: For each subset, the function calculates the total value and the total weight. This involves iterating through the items in the subset, which takes O(n) time. Exponential
c) "Greedy by value", the counter-example: Item A: Value = 6, Weight = 5, Item B: Value = 5, Weight = 5, Item C: Value = 4, Weight = 4. select Item A first because it has the highest value. However, the weight of Item A is 5 units, which already fills up the knapsack to its maximum capacity. After selecting Item A, we are left with a remaining capacity of 5 units. Now, based on the greedy strategy, we would select Item B next since it has the next highest value. However, including Item B would exceed the maximum weight capacity of the knapsack. Thus, the greedy by value strategy would select Items A and B, resulting in a total value of 11, but it violates the weight constraint. The optimal solution, in this case, is to select only Item C, which has a value of 4 and a weight of 4, satisfying both the value and weight constraints simultaneously. Therefore, the optimal solution has a total value of 4.
d) Using the greedy strategy of picking the lightest element first, also known as "greedy by weight," does not necessarily yield the optimal solution for the 0-1 Knapsack problem. This strategy involves selecting items based solely on their weights, without considering their values relative to their weights. Item A: Value = 2, Weight = 9, Item B: Value = 10, Weight = 5Item C: Value = 6, Weight = 6

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 fractionalKnapsack(unsigned int values[], unsigned int
weights[], unsigned int n, unsigned int maxWeight, double usedItems[])
```

Input example: **values** = [60, 100, 120], **weights** = [10, 20, 30], **n** = 3, **maxWeight** = 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 *fractionalKnapsack* 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 the greedy strategy leads to the optimal solution.

a) An array values[] representing the values of n items.
An array weights[] representing the weights of n items.
An integer n representing the number of items available.
An integer maxWeight representing the maximum weight capacity of the knapsack.
Objective:

Maximize the total value of the items that can be placed in the knapsack.
Constraints:

Each item can be partially included in the knapsack, allowing fractions of items to be placed.
The total weight of items in the knapsack cannot exceed the maximum weight capacity maxWeight.
Decision Variables:

usedItems[]: An array indicating the portion of each item used in the knapsack. For each item i, usedItems[i] represents the fraction of item i used in the knapsack.
Formulation:

Let v_i be the value of item i.
Let w_i be the weight of item i.
Let x_i be the fraction of item i used in the knapsack.
The objective is to maximize the total value
c) The sorting of items dominates the time complexity of the algorithm, as it takes O(n log n) time. Iterating through the sorted items and performing arithmetic operations in each iteration takes linear time, O(n). Other operations, such as calculating value-to-weight ratios and filling the usedItems array, contribute negligibly to the overall time complexity compared to the sorting step.
d) Greedy Solution (S): We know that the greedy strategy sorts the items based on their value-to-weight ratios in non-increasing order and selects items greedily until the knapsack is filled. Let's denote the total value of this solution as V(S).
Alternate Solution (S):* Let's assume there exists an alternate solution S* that yields a higher total value than the greedy solution, i.e., V(S*) > V(S).

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 *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.
c) Develop a greedy variant of your algorithm (*changeMakingGR*). 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 needs to have for this greedy strategy to work?

b) The temporal complexity of the provided brute-force solution (changeMakingBF) can be analyzed as follows:

Nested Loops: The algorithm uses nested loops to iterate through all possible combinations of coins. The number of iterations of these loops depends on the maximum stock of any coin denomination (S). For each coin denomination, the loop iterates from 0 to the stock of that denomination. Since there are 'n' denominations, the total number of iterations for each denomination would be at most S. Therefore, the total number of iterations across all denominations is S^n.

Total Combinations: In each iteration of the nested loops, the algorithm calculates the total value of coins for the current combination. This involves summing the values of all coin denominations, which requires O(n) operations.

Overall Complexity: Considering the above points, the overall temporal complexity of the brute-force algorithm can be expressed as O(n * S^n), where 'n' is the number of coin denominations and 'S' is the maximum stock of any coin denomination.