

1st Lab class – Brute Force and Greedy Algorithms

Instructions

- In this first practical class, students are required to implement the following exercises in C++ using any IDE of their choice; CLion is recommended as most of the exercises in this course will resort to the Google Test's unit testing library for the C++ programming language. A possible CLion project is provided in the compressed support files, which can be used throughout the semester. In case the student opts to use the CLion project environment provided, the support files for each lab class can be added to the project in a separate folder; the CMakeLists.txt file should be edited accordingly (please see comments in the file).

Exercises

1. The 3-sum problem

Implement the function *sum3* below.

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

The function finds three positive integers whose sum is equal to T . The function returns *true* and initializes the *selected* array with the three integers summing up to T . Otherwise, the function returns *false* (and the *selected* array is not initialized).

For example: $T = 10$

Solutions: *selected* = {1, 1, 8}, ..., *selected* = {2, 3, 5}, ...

- Implement *sum3* using an exhaustive search strategy (i.e. brute force) with $O(T^3)$ temporal complexity.
- Improve the temporal efficiency of *sum3* by implementing another brute-force solution with a lower temporal complexity.

2. The maximum subarray problem

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 $1 \leq i \leq j \leq n$. Implement the function *maxSubsequence* 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.

For example: $A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$

Solution: $[0, 0, 0, 1, 1, 1, 1, 0, 0]$, as subsequence $[4, -1, 2, 1]$ ($i = 3, j = 6$) produces the largest sum, 6.

3. Changing making problem (brute force)

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). Implement the function *changeMakingBF* below using a brute force strategy, considering a limited stock of coins of each denomination c_i , in *stock_i*, respectively.

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

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, then *usedCoins* is an array of the total number of coins used for each denomination c_i .

For example: $C = [1, 2, 5, 10]$, **Stock** = $[3, 5, 2, 1]$, $n=4$, $T = 8$

Result: $[1, 1, 1, 0]$

For example: $C = [1, 2, 5, 10]$, **Stock** = $[1, 2, 4, 2]$, $n=4$, $T = 38$

Result: $[1, 1, 3, 2]$

4. Changing making problem (greedy)

Considering the same description for the change-making problem as in the previous exercise, implement the function *changeMakingGreedy* below using a greedy strategy instead.

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

For example: $C = [1, 2, 5, 10]$, **Stock** = $[3, 5, 2, 1]$, $n=4$, $T = 8$

Result: $[1, 1, 1, 0]$

For example: $C = [1, 2, 5, 10]$, **Stock** = $[1, 2, 4, 2]$, $n=4$, $T = 38$

Result: $[1, 1, 3, 2]$

5. Canonical coin systems

Given a coin system C , with denominations (monetary labels) $C = \{1, c_2, \dots, c_n\}$, C is considered to be canonical if there is always a minimum combination of coins summing up x , with $c_3 + 1 < x < c_{n-1} + c_n$, resulting from a *greedy* strategy. If a greedy solution is not able to find the minimum amount of coins summing up x , the C is said non-canonical. Implement function *isCanonical* below.

```
bool isCanonical(int C[], int n)
```

The function uses an exhaustive search (i.e. brute force) to find any counter-example for the change x that might contradict the solution resulting from a greedy algorithm. Note: you can combine the functions implemented in exercises 3 and 4 above.

For example: If $C = \{1, 4, 5\}$, then any counter-example that might contradict the canonical nature of C would be between $6 < x < 9$.

Result: if $x = 7$, a greedy algorithm yields the optimum solution $\{5, 1, 1\}$; if $x = 8$, a greedy algorithm yields $\{5, 1, 1, 1\}$, whereas the optimum solution would be $\{4, 4\}$, in which case C is non-canonical.

6. The activity selection problem

The activity selection problem is concerned with the selection of non-conflicting activities to perform within a given time frame, given a set A of activities (a_i), each marked by a start time (s_i) and finish time (f_i). The problem is to select the maximum number of activities that can be performed by a single person or machine, assuming a given priority and that a person can only work on a single activity at a time. Implement the function *earliestFinishScheduling* below, using a greedy strategy, in which priority is given to activities with the earliest finish time (see slides of the theory class).

```
vector<Activity> earliestFinishScheduling(vector<Activity> A)
```

Consider a class *Activity*, as follows.

```
class Activity {
public:
    unsigned int start = 0;
    unsigned int finish = 0;
    Activity(unsigned int s, unsigned int f): start(s), finish(f) {};
    //other details omitted...
};
```

For example: $A = \{ a_1(10, 20), a_2(30, 35), a_3(5, 15), a_4(10, 40), a_5(40, 50) \}$

Result: $\{a_3, a_2, a_5\}$

7. Minimum Average Completion Time

Consider a machine on a factory line that needs to have its tasks scheduled in order to minimize their average completion time. The machine can only process one task at a time and each task has a predefined quantity of time needed for completion. For example, imagine the machine has two tasks to carry out, **a** and **b**, and we know that each task takes exactly 2 and 4 units of time respectively. The best scheduling option that minimizes the average completion time would be **{a, b}** (task **a** followed by task **b**) since the average completion time is 4, $(2 + 2 + 4) / 2$. On the contrary, **{b, a}** would give an average completion time of 5, $(4 + 4 + 2) / 2$.

- Formulate the postulated problem mathematically.
- Convince yourself that a greedy algorithm would give an optimal solution to this problem.
- Implement a greedy algorithm to find the optimal solution:

```
double minimumAverageCompletionTime (vector<unsigned int> tasks,  
                                     vector<unsigned int> &orderedTasks)
```

The function returns the minimum average task completion time and returns the optimal task ordering on the second argument (*orderedTasks*).