1st Lab Class – Brute Force and Greedy Algorithms – Solutions

---

Note: Keep in mind that there may be several possible solutions to these problems. The solutions herein presented are suggested as possible alternatives.

---

1. The 3-sum problem

a)

```
/**
 * Brute force algorithm.
 * Time: O(T^3). Space: O(1)
 */
bool sum3(unsigned int T, unsigned int nums[3]) {
    for(unsigned int i = 1; i < T; i++) {
            for(unsigned int j = 1; j < i; j++) {
                    for(unsigned int k = 1; k < j; k++) {
                            if(i + j + k == T) {
                                    nums[0] = i;
                                    nums[1] = j;
                                    nums[2] = k;
                                    return true;
                            }
                    }
            }
    }
    return false;
}
```

b)

```
/**
 * Brute force algorithm.
 * Time: O(T^2). Space: O(1)
 */
bool sum3(unsigned int T, unsigned int nums[3]) {
    for(unsigned int i = 1; i < T; i++) {
            for(unsigned int j = 1; j < i; j++) {
                    if(int(T - i - j) > 0) {
                            nums[0] = i;
                            nums[1] = j;
                            nums[2] = T - i - j;
                            return true;
                    }
            }
    }
    return false;
}
```

2. The maximum subarray problem

```
/**
 * Brute force algorithm.
 * Time: O(n^3). Space: O(n)
 */
int maxSubsequence(int A[], unsigned int n, unsigned int &i, unsigned int &j) {
    bool firstSum = true;
    int maxSum;
    for(unsigned int a = 0; a < n; a++) {
           for(unsigned int b = a + 1; b < n; b++) {
                  int sum = 0;
                  for(unsigned int c = a; c <= b; c++) {
                         sum += A[c];
                  }
                  if(firstSum) {
                         firstSum = false;
                         maxSum = sum;
                         i = a;
                         j = b;
                  }
                  else if(sum > maxSum) {
                         maxSum = sum;
                         i = a;
                         j = b;
                  }
           }
    }
    return maxSum;
}
```

3. Changing making problem (brute force)

```
/**
 * Brute force algorithm.
 */
bool changeMakingBF(unsigned int C[], unsigned int Stock[], unsigned int n, unsigned
int T, unsigned int usedCoins[]) {
    // Static memory allocation is used since it's faster but this assumes there are
at most 20 coin values (n <= 100).
    unsigned int curCandidate[20]; // current solution candidate being built
    // Prepare the first candidate
    for(unsigned int i = 0; i < n; i++) {
        curCandidate[i] = 0;
    }
    // Iterate over all the candidates
    bool foundSol = false;
    unsigned int minCoins; // value of the best solution found so far
    while (true) {
        // Verify if the candidate is a solution
        unsigned int totalValue = 0;
        unsigned int totalCoins = 0;
        for(unsigned int k = 0; k < n; k++) {
            totalValue += C[k]*curCandidate[k];
            totalCoins += curCandidate[k];
        }
        if(totalValue == T) {
```

```
            // Check if the solution is better than the previous one (or if it's the
first one)
            if(!foundSol || totalCoins < minCoins) {
                foundSol = true;
                minCoins = totalCoins;
                for(unsigned int k = 0; k < n; k++) {
                    usedCoins[k] = curCandidate[k];
                }
            }
        }

        // Get the next candidate
        unsigned int curIndex = 0;
        while(curCandidate[curIndex] == Stock[curIndex]) {
            curIndex++;
            if(curIndex == n) break;
        }
        if(curIndex == n) break;
        // Set the stock of the higher coin values in the candidate solution back to
0.
        // Example with 1 stock per coin value: 1 1 0 1 -> 0 0 1 1.
        // Enumeration of the 8 candidates for 3 coin values with 0-1 stock:
        // 0 0 0 -> 1 0 0 -> 0 1 0 -> 1 1 0 -> 0 0 1 -> 1 0 1 -> 0 1 1 -> 1 1 1
        // (it's like incrementing by 1 numbers in binary written backwards)
        for(unsigned int i = 0; i < curIndex; i++) {
            curCandidate[i] = 0;
        }
        curCandidate[curIndex]++;
    }
    return foundSol;
}
```

4. Changing making problem (greedy)

```
/**
 * Greedy algorithm.
 * Time: O(D*S). Space: O(D), D-number of coin values, S-maximum stock of any value
 */
bool changeMakingGreedy(unsigned int C[], unsigned int Stock[], unsigned int n,
unsigned int T, unsigned int usedCoins[]) {
    for(unsigned int i = 0; i < n; i++) {
        usedCoins[i] = 0;
    }
    for(int i = n - 1; i >= 0; i--) {
        while(usedCoins[i] < Stock[i] && T >= C[i]) {
            usedCoins[i]++;
            T -= C[i];
        }
    }
    return T == 0;
}
```

5. Canonical coin systems

```
/**
 * Auxiliary function
 */
unsigned int sumArray(unsigned int a[], unsigned int n) {
    // Returns sum of array a
}

/**
 * Brute force algorithm.
 * Time: O(2^(D*S)). Space: O(D*S), D-number of coin values, S-maximum stock of any
value
 */
bool isCanonical(unsigned int C[], unsigned int n) {
    if(n <= 2) return true;
    int minVal = C[2] + 2;
    int maxVal = C[n-2] + C[n-1] - 1;
    for(unsigned int T = minVal; T <= maxVal; T++) {
        // Define the maximum stocks
        unsigned int Stock[10000]; // static memory allocation is faster
        unsigned int usedCoins[10000];
        for(unsigned int i = 0; i < n; i++) {
            Stock[i] = T / C[i];
            //std::cout << Stock[i] << " ";
        }
        std::cout << std::endl;
        changeMakingBF(C, Stock, n, T, usedCoins);
        unsigned int numCoinsBF = sumArray(usedCoins, n);
        changeMakingGreedy(C, Stock, n, T, usedCoins);
        unsigned int numCoinsGreedy = sumArray(usedCoins, n);
        if(numCoinsBF < numCoinsGreedy) return false;
    }
    return true;
}
```

6. The activity selection problem

```
/**
 * Operators
 */
bool Activity::operator==(const Activity &a2) const {
    return start == a2.start && finish == a2.finish;
}

bool Activity::operator<(const Activity &a2) const {
    return finish < a2.finish;
}

/**
 * Greedy algorithm.
 * Time: O(n*log(n)). Space: O(n), n-number of activities
 */
std::vector<Activity> earliestFinishScheduling(std::vector<Activity> A) {
    std::sort(A.begin(),A.end());
```

```
    std::vector<Activity> selected;
    unsigned int latestTime = 0;
    for(size_t i = 0; i < A.size(); i++) {
        if(A[i].start >= latestTime) {
            latestTime = A[i].finish;
            selected.push_back(A[i]);
        }
    }
    return selected;
}
```

7. Minimum Average Completion Time

a) Consider a set T={ a_1 , a_2 , …, a_n } of n tasks in which task a_i requires t_i units of processing time and c_i represents the completion time of task a_i .

Input: T, t_i with $i \in [1,n]$

Restrictions: Consider an ordered set of tasks each with starting time s_i and finish time f_i for task a_i. $\forall$ a_i with $i \in [1,n]$, $s_{i+1} \geq f_i$. Consider $c_k = t_k + c_{k-1}$.

Objective Function: $min\ \frac{1}{n} \times \sum_{i=1}^{n} c_i$

Output: Average completion time, $\frac{1}{n} \times \sum_{i=1}^{n} c_i$

b) Considering $c_k = t_k + c_{k-1}$, and adopting a greedy strategy in which we always choose the available task with the smallest processing time t, it is trivial to see that $\forall\ c_k, k \in [1,n]$ is minimized. If each c is minimized, its sum, $\sum_{k=1}^{n} c_k$, is also minimized. Since n is a constant, minimizing the sum, $\sum_{k=1}^{n} c_k$, is equivalent to minimizing the averaged sum, $\frac{1}{n} \times \sum_{i=1}^{n} c_i$. Thus, this greedy strategy will provide an optimal solution.

c)

```
/**
 * Greedy algorithm.
 * Time: O(n*log(n)). Space: O(n), n-number of tasks
 */
double minimumAverageCompletionTime(std::vector<unsigned int> tasks,
std::vector<unsigned int> &orderedTasks) {
    std::sort(tasks.begin(),tasks.end());
    orderedTasks = tasks;
    unsigned int latestTime = 0;
    double sumEndTimes = 0.0;
    for(size_t i = 0; i < orderedTasks.size(); i++) {
        latestTime += orderedTasks[i];
        sumEndTimes += latestTime;
    }
    return sumEndTimes / orderedTasks.size();
}
```