**COMP S265F Design and Analysis of Algorithms**
**Assignment 1 – Suggested Solution**

**Question 1 (15 marks).** Let $gcd(x, y)$ be the greatest common divisor of $x$ and $y$.

(a) Using Euclid's algorithm with subtractions:

$$\begin{aligned} gcd(265, 380) &= gcd(265, 115) = gcd(150, 115) = gcd(35, 115) = gcd(35, 80) = gcd(35, 45) \\ &= gcd(35, 10) = gcd(25, 10) = gcd(15, 10) = gcd(5, 10) = gcd(5, 5) = 5 \end{aligned}$$

(b) Using the Euclid's algorithm with modulus operations:

$$\begin{aligned} gcd(380, 265) &= gcd(265, 115) = gcd(115, 35) = gcd(35, 10) \\ &= gcd(10, 5) = gcd(5, 0) = 5 \end{aligned}$$

**Question 2 (35 marks).**

(a) The main operation of the algorithm is additions, so its time complexity equals the number of additions.

Line 3 loops through and adds the entries $X[i]$ to $X[j]$, so it takes $j - i + 1$ additions. Line 4 stores the result to $Y[i][j]$ and needs no addition. Therefore, for a particular value of $i$ in the outer for-loop, the number of additions performed by lines 2 to 5 is

$$\sum_{j=i}^{n-1}(j - i + 1) = 1 + 2 + 3 + \cdots + (n - i) = \frac{(n - i)(n - i + 1)}{2}$$

We can easily find an upper bound of the above term, as follows:

$$\begin{aligned} \frac{(n - i)(n - i + 1)}{2} &\leq \frac{n(n + 1)}{2} && (\text{as } i \geq 0) \\ &\leq \frac{n(n + n)}{2} && (\text{as } n \geq 1) \\ &= n^2 \end{aligned}$$

Therefore, the total time complexity is at most

$$\sum_{i=0}^{n-1} n^2 = n^3 = O(n^3) \ .$$

**Remark.** To see that this is a tight bound, we find a lower bound, as follows:

$$\frac{(n - i)(n - i + 1)}{2} \geq \frac{(n - i)^2}{2}$$

Therefore, the total time complexity is at least

$$\begin{aligned} \sum_{i=0}^{n-1} \frac{(n - i)^2}{2} &= \frac{1^2}{2} + \frac{2^2}{2} + \frac{3^2}{2} + \cdots + \frac{n^2}{2} \\ &= \frac{1}{2}\left(1^2 + 2^2 + 3^2 + \cdots + n^2\right) \\ &= \frac{1}{2} \cdot \frac{n(n + 1)(2n + 1)}{6} = \Omega(n^3) \ . \end{aligned}$$

(b) The bottleneck is to sum up all the entries $X[i]$ to $X[j]$ for a particular entry $Y[i][j]$, which leads to the cubic time complexity, i.e., $O(n^3)$.

(c) To improve the time complexity, we need at most 1 addition (instead of $j - i + 1$ additions) to compute the entry $Y[i][j]$, as shown below:

- $Y[i][i] = X[i]$
- $Y[i][j] = Y[i][j-1] + X[j]$ for $j > i$

This leads to the following implementation in q2ans.py (note that any entry $Y[i][j]$, where $j < i$, is 0):

```
1  def func(X, n):
2      Y = [[0 for i in range(n)] for j in range(n)]
3      for i in range(n):
4          Y[i][i] = X[i]
5          for j in range(i+1, n):
6              Y[i][j] += Y[i][j-1] + X[j]
7      return Y
8
9  if __name__ == "__main__":
10     n = 500
11     X = list(range(n))
12     for i in range(30, 50):
13         print(X[i], end=" ")
14     print()
15
16     print(func(X, n)[30][49])
```

(d) We analyze the time complexity of `func(X, n)` in q2ans.py, as follows:

Line 2 initializes the 2D array `Y` to all zeros, which takes $O(n^2)$ time.
Lines 4, 6 and 7 take $O(1)$ time.

The inner for-loop (lines 5 to 6) takes $O((n-1) - (i+1) + 1) = O(n - i - 1)$ time.

Therefore, the outer for-loop (lines 3 to 6) has the time complexity of

$$\sum_{i=0}^{n-1} O(1 + (n - i - 1)) = O\left(\sum_{i=0}^{n-1} (n - i)\right) = O\left(1 + 2 + 3 + \cdots + n\right) = O\left(\frac{n(n+1)}{2}\right) = O(n^2) \ .$$

The total time complexity of `func(X, n)` is

$$O(n^2 + n^2 + 1) = O(n^2) \ .$$

**Question 3 (25 marks).**

(a)
- Left tree (any one): 0, 1
- Right tree (any one): 0, 1, 2, 3

(b) We can find a local minimum starting from the root node, as follows:

1. If the current node is a leaf node, return its value.
2. Otherwise, the current node is an internal node and we check whether it has a smaller value than both of its children.
3. If yes, return the current node value.
4. If no, go to the child node with smaller value, and repeat Step 1.

*Proof of Correctness.* Consider the following two possible cases:

**Case 1: A leaf node value is returned.** The leaf node has a smaller value than its parent (if there is one), so it is a local minimum.

**Case 2: An internal node value is returned.** The internal node has a smaller value than both of its children and its parent (if there is one), so it is a local minimum.

This leads to the following implementation in `q3ans.py`:

```
1  def traverse(node):
2      if node.left == None:
3          return node.label
4      else:
5          if node.left.label < node.right.label:
6              min_node = node.left
7          else:
8              min_node = node.right
9
10         if node.label < min_node.label:
11             return node.label
12         else:
13             return traverse(min_node)
```

(c) Consider the function `traverse(node)` in Q3(b).

Let $T(n)$ be its time complexity when `node` is the root of a $n$-node complete binary tree.

If `node` is a leaf node, then $n = 1$ and by lines 2 to 3 in the if-part, $T(1) = O(1)$.

If `node` is an internal node, lines 5 to 8 identify the child node with smaller value and set it as `min_node`, which takes $O(1)$ time. There are two cases:

- If `node` has a smaller value than `min_node`, we just return the value of `node`, which by lines 10 to 11, takes $O(1)$ time.
- If `min_node` has a smaller value than `node`, we make the recursive call `traverse(min_node)` and return its value.
  As `node` is the root of a $n$-node complete binary tree, its child `min_node` is also the root of a complete binary tree, which has $\frac{n-1}{2} \leq \frac{n}{2}$ nodes. Thus, `traverse(min_node)` takes at most $T\left(\frac{n}{2}\right)$ time.
  Therefore,
  $$T(n) = T\left(\frac{n}{2}\right) + O(1) = 1 \cdot T\left(\frac{n}{2}\right) + O(n^0) \ .$$
  Recall the Master Theorem that
  $$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases} .$$
  In this case, we have $a = 1$, $b = 2$ and $d = 0$, thus
  $$d = 0 = \log_2 1 = \log_b a \ .$$
  By the Master Theorem, $T(n) = O(n^d \ \log n) = O(n^0 \ \log n) = O(\log n)$.

**Question 4 (25 marks).**

(a) This is a coin change problem, which should generally be solved using dynamic programming but not greedy algorithms.

Yet for the set of coins $\{\$1, \$2, \$5, \$10\}$, a greedy algorithm works: to minimize the number of coins, we change each coin one by one and always select the coin with the maximum possible value.

In other words, given the positive integer account balance $\$b$, we select the coins, as follows:

1. Select $\left\lfloor \frac{b}{10} \right\rfloor$ \$10 coins, and define the remaining value $r_{10} = b - 10 \times \left\lfloor \frac{b}{10} \right\rfloor$.
2. Select $\left\lfloor \frac{r_{10}}{5} \right\rfloor$ \$5 coins, and define the remaining value $r_5 = r_{10} - 5 \times \left\lfloor \frac{r_{10}}{5} \right\rfloor$.
3. Select $\left\lfloor \frac{r_5}{2} \right\rfloor$ \$2 coins, and define the remaining value $r_2 = r_5 - 2 \times \left\lfloor \frac{r_5}{2} \right\rfloor$.
4. Select $r_2$ \$1 coins.

This leads to the following implementation in q4ans.py:

```
from sys import stdin

def coin_change(b):
    coins = [10, 5, 2, 1]
    num = []
    for c in coins:
        num.append(b//c) # we need floor(b/c) coins of value c
        b = b % c

    answer = []
    for i in range(len(coins)):
        answer.extend([coins[i] for j in range(num[i])])
    return answer

if __name__ == '__main__':
    b = int(stdin.readline())
    print(coin_change(b))
```

(b) Let $OPT = (n'_{10}, n'_5, n'_2, n'_1)$ be the optimal solution such that $n'_i$ is the number of coin \$$i$. In other words,

$$10 \cdot n'_{10} + 5 \cdot n'_5 + 2 \cdot n'_2 + 1 \cdot n'_1 = b \ .$$

**Property 1: $\mathbf{n'_1 \leq 1}$.** We can always replace two \$1 coins by one \$2 coin to reduce the total number of coins. Therefore, in $OPT$, the maximum value achieved by \$1 coins is $\$1 \cdot 1 = \$1$.

**Property 2: $\mathbf{n'_1 + n'_2 \leq 2}$.** If we have three coins of \$1 and \$2, by Property 1, it must be either $\$1, \$2, \$2$ or $\$2, \$2, \$2$. We can reduce the total number of coins in both cases: For the first case, we can replace $\$1, \$2, \$2$ by one \$5 coin; for the second case, we can replace $\$2, \$2, \$2$ by one \$5 coin and one \$1 coin. Therefore, in $OPT$, the maximum value achieved by $\$1, \$2$ coins is $\$2 \cdot 2 = \$4$.

**Property 3: $\mathbf{n'_5 \leq 1}$.** We can always replace two \$5 coins by one \$10 coin to reduce the total number of coins. Therefore, in $OPT$, the maximum value achieved by $\$1, \$2$, \$5 coins is $\$5 \cdot 1 + \$2 \cdot 2 = \$9$.

*Solution 1: Proof by contradiction*

Let $ALG = (n_{10}, n_5, n_2, n_1)$ be the greedy solution such that $n_i$ is the number of coin \$$i$.

Suppose, for the sake of contradiction, $ALG$ and $OPT$ are different. Let $k$ be the largest $k$ where $n_k \neq n'_k$.

**Case 1: $k = 10$.**

If $n_{10} < n'_{10}$, then by the greedy algorithm, in ALG, the value achieved by $\$1, \$2, \$5$ coins must be less than \$10 (otherwise, ALG can have one more \$10 coin). Thus, it is impossible for $OPT$ to have more \$10 coins, contradicting that $n'_{10} > n_{10}$.

If $n_{10} > n'_{10}$, then in OPT, the value achieved by $\$1, \$2, \$5$ coins is at least 10, contradicting Property 3 of $OPT$ that this value is at most 9.

**Case 2: $k = 5$.** Note that $n_{10} = n'_{10}$.

If $n_5 < n'_5$, then by the greedy algorithm, in ALG, the value achieved by $\$1, \$2$ coins must be less than \$5 (otherwise, ALG can have one more \$5 coin). Thus, it is impossible for $OPT$ to have more \$5 coins, contradicting that $n'_5 > n_5$.

If $n_5 > n'_5$, then in OPT, the value achieved by $\$1, \$2$ coins is at least 5, contradicting Property 2 of $OPT$ that this value is at most 4.

**Case 3:** $k = 2$**.** Note that $n_{10} = n'_{10}$ and $n_5 = n'_5$.

If $n_2 < n'_2$, then by the greedy algorithm, in ALG, the value achieved by \$1 coins must be less than \$2 (otherwise, ALG can have one more \$2 coin). Thus, it is impossible for $OPT$ to have more \$2 coins, contradicting that $n'_2 > n_2$.

If $n_2 > n'_2$, then in OPT, the value achieved by \$1 coins is at least 2, contradicting Property 1 of $OPT$ that this value is at most 1.

**Case 4:** $k = 1$**.** Note that $n_{10} = n'_{10}$, $n_5 = n'_5$ and $n_2 = n'_2$. This implies that $n_1 = n'_1$, contradicting that $n_1 \neq n'_1$.

*Solution 2: Proof by mathematical induction*

Let $c_1 = 1, c_2 = 2, c_3 = 5, c_4 = 10$, and for any $i \geq 5$, $c_i = c_{i-1} + 10$. We prove that for any integer $k \geq 1$, the greedy algorithm makes the minimal coins for any positive integer account balance \$$b$ such that $c_k \leq b < c_{k+1}$.

**Base case:** $k = 1$**.** The only possible value for $b$ such that $c_1 = 1 \leq b < c_2 = 2$ is 1. Then, both the greedy algorithm and $OPT$ make one \$1 coin, which is the minimal.

**Induction step:** Assume that, for some integer $k \geq 1$, the greedy algorithm makes the minimal coins for any balance \$$b$ such that $c_k \leq b < c_{k+1}$.

Now, consider any balance \$$b$ such that $c_{k+1} \leq b < c_{k+2}$.

Let $M = min\{10, c_{k+1}\}$ be the maximum coin value less than or equal to $c_{k+1}$.

By definition, the greedy algorithm makes at least one coin of value \$$M$. The optimal algorithm must also make at least one coin of value \$$M$; otherwise, by the property of $OPT$, it cannot use coins with less values to make up for the value \$$M$.

Thus, the problem is reduced to a coin change problem for account balance \$$(b - M)$.

If $b = M$, the greedy algorithm makes one coin, which is the minimal. Otherwise, we have $c_k \leq b - M < c_{k+1}$. By the induction hypothesis, the greedy algorithm makes the minimal coins for \$$(b - M)$ and thus \$$b$ (as both the greedy and optimal algorithm make at least one \$$M$ coin), completing the proof.