CSE 321 HOMEWORK 3

① Pseudo code for first question:

```
def find_max_discount (store_list, current_index, current_set, max_discount_set):
    if current_index == len(store_list)
        current_discount = calc_discount (current_set)

        if current_discount > calc_discount (max_discount_set):
            max_discount_set = current_set.copy()
        return max_discount_set

    max_discount_set = find_max_discount (store_list, current_index +1, current_set + [store_list[current_index]], max_discount_set)
    max_discount_set = find_max_discount (store_list, current_index+1, current_set, max_discount_set)
    return max_discount_set
```

Recurrence relation is : $\underbrace{2}_{\text{two subproblems}}T(\underbrace{n-1}_{\text{size}}) + \underbrace{O(1)}_{\text{calc\_discount time complexity}}$.   by applying master's theorem for decreasing functions:

if: $T(n)$: $a \cdot T(n-b) + f(n)$

$\boxed{a=2 \quad b=1 \quad d=0}$

if $a>1 \Rightarrow \Theta(n^d \cdot a^{n/b})$

$\boxed{\Theta(2^n)}$

---

② Pseudo code for second question:

```
def exhaustive_search (cost_matrix):
    n= len (cost_Matrix)
    best_assignment = None
    min_cost = float ('inf')

    permutations = []
    generate_permutations (n, [], permutations)

    for perms in permutations
        assignment = list (enumerate (perms))
        total_cost = calculate_cost (assignment, cost_matrix)

        if total_cost < min_cost:
            min_cost = total_cost
            best_assignment = assignment

    return best_assignment, min cost
```

Time complexity analysis: Best, Worst and Average cases complexityies are same. Let n be number of users (or processors), and factorial (n) be the number of permutations. For each permutation, the function calculates the cost using 'calculate_cost' that it has $O(n)$ complexity. Therefore, the overall time complexity is $\boxed{O(n.n!)}$

③ Pseudo-code for third question

```
def exhaustive_search (current_sequence, remaining_parts, energy_so_far, best_sequence, min_energy):
    if len(remaining_parts) == 0:
        if energy_so_far < min_energy:
            best_sequence = current_sequence.copy()
            min_energy = energy_so_far
        return best_sequence, min_energy

    for part in remaining parts:
        next_sequence = current_sequence + [part]
        next_remaining_parts = [p for p in remaining parts if p != part]

        energy_cost = calculate_energy_cost (current_sequence [-1], part) if current_sequence else 0
        best_sequence, min_energy = exhaustive_search ( next_sequence, next_remaining_parts, energy_so_far +
                                                        energy_cost, best_sequence, min_energy )
    return best_sequence, min energy.
```

Time Complexities : Best-case : When the optimal sequence is found early in the search. But we need to explore all possible sequences so time complexity is $\boxed{O(n!)}$

Worst-case : Explores all possible permutations of the assembly sequence. The num of permutations for N parts is N!. So, worst-case is $\boxed{O(n!)}$

Average-case : Average-case needs to explore roughly half of the possible sequences. Therefore time complexity is $\boxed{O(n!)}$

---

④ Pseudo-code for fourth question.

```
def exhaustive_search (coins, target_amount, current_amount=0, current_count=0, remaining_coins=None):
    if remaining_coins is None:
        remaining_coins = coins

    if current_amount == target_amount:
        return current_count
    elif current_amount > target_amount or not remaining_coins:
        return float('inf')

    use_current_coin = exhausive_search (coins, target_amount, current_amount + remaining_coins [0], current_count+1,
                                                                                        remaining coins)
    skip_current_coin = exhausive_search (coins, target_amount, current_amount, current_count, remaining_coins [1:])

    return min (use_current_coin, skip_current_coin)
```

Time Complexity Analysis : At each step, the algorithm explores two possibilities : using or skipping the current coin. The number of recursive calls grows exponentially with the number of coins. Therefore the time complexity of this function is $\boxed{O(2^n)}$

⑤ Write recurrence relation for the following function when called with an array of length n. Provide the average-case complexity by solving the recurrence relation.

```
def find_min_max(arr, low, high):
    if low = high:
        return arr[low], arr[low]

    if high - low == 1:
        if arr[low] < arr[high]:
            return arr[low], arr[high]
        else:
            return arr[high], arr[low]

    mid = (low+high) //2

    left_min, left_max = find_min_max(arr, low, mid)
    right_min, right_max = find_min_max(arr, mid+1, high)

    min_val = min(left_min, right_min)
    max_val = max(left_max, right_max)

    return min_val, max_val
```

Recurrence relation is:

$$T(n) = \underbrace{2}_{\text{two subproblems}} \cdot T(\underbrace{n/2}_{\text{half size}}) + \underbrace{C}_{\text{constant time}}$$

by applying masters theorem,

$$= a \cdot T(n/b) + f(n)$$

$a = 2 \quad b = 2 \quad d = 0$

$\checkmark\ a > b^d \Rightarrow \theta(n^{\log_b a}) \rightarrow n^{\log_2 2}$

$= n$

$\boxed{\Rightarrow \theta(n)}$