CSE321 HW3

Elifnur Kabalcı
1801042617

1)
```
def max_discount(stores, current_set=[], max_set=[], max_discount=0):
    if not stores:
        current_discount = calc_discount(current_set)
        if current_discount > max_discount:
            return current_set, current_discount
        else:
            return max_set, max_discount
    for i in range(len(stores)):          # recursive part
        store = stores[i]
        remaining_stores = stores[i+1:]
        new_set = current_set + [store]
2 T(n-1) {
        max_set, max_discount = max_discount(remaining_stores, new_set, max_set, max_discount)
        max_set, max_discount = max_discount(remaining_stores, current_set, max_set, max_discount)
    return max_set, max_discount
```

=> for each store, we two recursive calls (one where we visit the store, and one where we don't), and the calc_discount function has a time complexity of $O(1)$.

=> The recurrence relation for this algorithm is:

$T(n) = 2 \cdot T(n-1) + O(1)$

=> The average-case time complexity of the derived recurrence relation is $O(2^n)$, as we are generating all possible subsets of the set of stores, which is a power set, and the size of a power set is 2^n.

2)
```
def min_cost(users, processes, processors, current_alloc=[], min_alloc=[], min_cost=float('inf')):
    if not users or not processes:
        current_cost = calc_cost(current_alloc)
        if current_cost < min_cost:
            return current_alloc, current_cost
        else:
            return min_alloc, min_cost
n^3 T(n-1) {
    for i in range(len(users)):
        for j in range(len(processes)):
            for k in range(len(processors)):
                user = users[i]
                process = processes[j]
                processor = processors[k]
                remaining_users = users[:i] + users[i+1:]
                remaining_processes = processes[:j] + processes[j+1:]
                remaining_processors = processors[:k] + processors[k+1:]
                new_alloc = current_alloc + [(user, process, processor)]
                min_alloc, min_cost = min_cost(remaining_users, remaining_processes, remaining_processors,
                                               new_alloc, min_alloc, min_cost)
    return min_alloc, min_cost
```

$T(n) = n^3 \cdot T(n-1) + O(1)$

(2)

=> Each user-process pair, we make n^3 recursive calls (one for each possible assignment to a processor), and the calc_cost function has a time complexity of $O(1)$.

=> The worst-case, best-case and average-case time complexity of the derived recurrence relation are all $O(n^4)$, as we are generating all possible assignments of user-process pairs to processors, which is a permutation, and the size of a permutation is n^4.

3)
```
def min_energy(parts, current_seq=[], min_seq=[], min_energy=float('inf')):
    if not parts:
        current_energy = calc_energy(current_seq)
        if current_energy < min_energy:
            return current_seq, current_energy
        else:
            return min_seq, min_energy
    for i in range(len(parts)):
        part = parts[i]
        remaining_parts = parts[:i] + parts[i+1:]
        new_seq = current_seq + [part]
        min_seq, min_energy = min_energy(remaining_parts, new_seq, min_seq, min_energy)
    return min_seq, min_energy
```

$n \cdot T(n-1)$

=> We make n recursive calls (one for each possible assembly sequence) and the calc_energy function has a time complexity of $O(1)$.

$T(n) = n \cdot T(n-1) + O(1)$

=> The worst-case, best-case and average-case time complexity of the derived reccurence relation are all $O(n!)$, as we are generating all possible sequences of assembling parts, which is a permutation, and the size of a permutation is $n!$.

4)
```
def min_coins(coins, target, current_count=0, min_count=float('inf')):
    if target == 0:
        return min(current_count, min_count)
    if not coins or target < 0:
        return min_count
    for i in range(len(coins)):
        coin = coins[i]
        remaining_coins = coins[i:]
        new_count = current_count + 1
        new_target = target - coin
        min_count = min_coins(remaining_coins, new_target, new_count, min_count)
        min_count = min_coins(remaining_coins, target, current_count, min_count)
    return min_count
```

$n \cdot T(n-1)$

=> Each coin, we make two recursive calls (one where we use the coin, and one where we don't) and the comparison operation has a time complexity of $O(1)$.

=> Worst case is $O(2^n)$, as we are generating all possible combinations of using or not using each coin. The best case time complexity is $O(n)$, which occurs when the target amount is exactly divisible by the largest coin denomination. The average case time complexity is also $O(2^n)$ as on average, we still need to explore all possible combinations of using or not using each coin.

5) find_min_max: Divide and conquer algorithm that finds the minimum and maximum values in an array. The function divides the array into two halves and recursively finds the minimum and maximum values in each half.

- The base case is when the array has one or two elements, in which case the function directly returns the minimum and maximum values.

=> Function divides the array into two halves (leading to the $2T(n/2)$ term) and performs a constant amount of work to find the minimum and maximum values in each half (leading to the $O(1)$ term).

$$T(n) = 2T(n/2) + O(1)$$

=> The average case time complexity of the function can be found by solving the relation using the master teorem.

$a = 2$, $b = 2$, $f(n) = O(1)$, so $c = 0$ -> $c < \log_b^{(a)}$ => $\Theta(n^{\log_b^a}) = O(n)$