CS 570: Analysis of Algorithms – H10

Submitted by: Indronil Bhattacharjee

Exercise 16.1-2

Question: Suppose that instead of always selecting the first activity to finish, you instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

The approach of selecting the last activity to start that is compatible with all previously selected activities is indeed a greedy algorithm. This approach ensures that at each step, we make the locally optimal choice by selecting the activity that maximizes the time available for subsequent activities.

```
Greedy-Last-Compatible-Activity-Selector(s, f):
1   n = length(s)
2   activities = []
3   activities.append(0)
4   last_activity_index = 0
5   for i from 1 to n-1:
6       if s[i] >= f[last_activity_index]:
7           activities.append(i)
8           last_activity_index = i
9   return activities
```

Problem 16-1

Question: Coin changing - Consider the problem of making change for n cents using the smallest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in denominations that are powers of c: the denominations are c_0, c_1, … , c_k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an O(nk)-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

(a) Always give the highest denomination coin that you can without going over. Then, repeat this process until the amount of remaining change drops to 0.

```
Greedy-Coin-Change(n, coins):
1   result = []
2   for coin in coins in descending order:
3       while n >= coin:
4           result.append(coin)
5           n -= coin
6   return result
```

(b) The steps of the greedy algorithm for selecting the last compatible activity. It iterates through the sorted activities and selects the last activity to start that is compatible with all previously selected activities. The selected activities form the maximum-size subset of mutually compatible activities.

```
Greedy-Coin-Change-Powers-Of-C(n, c):
1   result = []
2   i = 0
3   while n > 0:
4       if n >= c^i:
5           // If n is greater than or equal to c^i, add c^i to the solution
6           result.append(c^i)
7           // Reduce the remaining amount by the value of c^i
8           n -= c^i
9       else:
10          // Move to the next lower denomination
11          i -= 1
12  return result
```

At each step, the greedy algorithm selects the last activity to start that is compatible with all previously selected activities. This choice maximizes the remaining time available for subsequent activities, ensuring that the locally optimal choice is made at each step.

Let $A_1, A_2, \ldots, A_k$ be the activities selected by the greedy algorithm. Assume there exists an optimal solution $B_1, B_2, \ldots, B_m$ where m>k. Since the activities are sorted by finish times, we know that the finish time of Bm is greater than or equal to the finish time of A_k. Therefore, Bm is compatible with all activities $A_1, A_2, \ldots, A_k$, contradicting the optimality of $A_1, A_2, \ldots, A_k$. Thus, the greedy solution is optimal.

(c) Let the coin denominations be {1, 3, 4}, and the value to make change for be 6. The greedy solution would result in the collection of coins {1, 1, 4} but the optimal solution would be {3, 3}.

(d) O(nk)-time Algorithm for Arbitrary Coin Denominations with Dynamic Programming approach.

```
Dynamic-Coin-Change(n, coins):
1   dp[0] = 0
2   for i from 1 to n:
3       dp[i] = ∞
4       for coin in coins:
5           if i >= coin:
6               dp[i] = min(dp[i], dp[i - coin] + 1)
7   return dp[n]
```

This algorithm works because it considers all possible combinations of coins to find the minimum number needed to make change for each value from 0 to n. The time complexity is O(nk), where n is the target amount and k is the number of different coin denominations. Since the first for loop runs n times, and the inner for loop runs k times, and the later while loop runs at most n times, the total running time is O(nk).