Seonhye Yang (sy3420) Homework 5

Problem 1

coins - 25 - quater
        10 - dimes
        5 - nickel
        1 - penny

Greedy method
- This will solve greatest value coins to use without exceeding the given amount.

```
coin = [1,5,10,25]
result = 0

MakeChang(n):
    coin = [1,5,10,25]      = our coins
    n = len(coin)           = 4
    result = []             = empty array to append results.
    i = n-1
    while i ≥ 0 :
        while n ≥ coin[i] :
            n = n - coin[i]
            result = coin[i]    ← append result into array
        i - 1
```

- This sorts array of coins in decreasing order. It finds the largest denomination that is smaller than given amount. Then we add the denomination to our empty array, and minus value of the resulting denomination from the given amount.
- If the amount becomes zero, the algorithm ends.

b. Given our coins, $[C_n, C_{n-1}, \dots, 1]$ such that for $1 \le k \le n$, $C_k > C_{k-1}$. This algorithm gives the minimum number of coins If $C_k > C_{k-1} + C_{k-2}$ for $n = C_k + C_{k-1}$. The algorithm to coins $[C_n, \dots, 1]$, where $C_k \le n$, will give fewer coins than the number from using this algorithm to subset of coin $[C_{k-1}, C_{k-2} \dots 1]$.

c. coin = [9, 6, 5, 1]
- Given amount = 11
- Our algorithm will give 9, 1, 1.
- However, it should be 6, 5.

Problem 2

- Frog is on last index of array. A[1.... n]
  frog's position

① if last index = 0, then frog cannot make any jumps.
② if the value A[i] > n-i , the frog can jump to first index.

```
CanReach ( A[1.... n]):
        if len(n) ≤ 1 :
              return TRUE
        if A[n] == 0 :          can't make jumps.
              return FALSE
        jump = 0
        current = 0
        maxjump = 0
        for i=1 to n :
              maxjump = max(i - A[i] , 1)
              if i == current :
                    jump= jump +1
                    current = maxjump
        return maxjump
```

b. TC = O(n) because there are n elements in the array and we iterate through each element in the array once.

- Greedy algorithm will always make locally optimal decisions

→ For k ranging from 1 to # jumps in greedy method.
- $a_k + n(a_k) \le g_k + n(g_k)$.
  - → a_k is index of jump sequences at jump k.
  - → g_k is index of greedy solution at jump k .
- for all $0 \le m \le g_k$, $m + n(m) \le g_k + n(g_k)$

Greedy selects local optimum and adds it to current solution. This will repeat until global optimum or final result is obtained.

$\boxed{\text{Problem 3}}$

- Given a set I of n intervals on real line
  - → start $= S[1 \dots n]$
  - → finish $= f[1 \dots n]$

  $S[i]$, $f[i]$ denote start and finish time of the i^{th} interval of I.

Goal: color the intervals in I so that no 2 overlapping intervals are assigned the same color.

a. Compute min numbers of colors need to color I so that overlapping intervals are given different colors.

$I_i = (s_i, f_i)$, $s_i < f_i$. I_i, I_j are compatible if they don't intersect ($s_i < f_j$ or $f_i < s_j$).

```
for i=1 to n:
    maxcount = 0
        count = 0
            Q = ∅
            Q = {s_i, f_i}
  • while Q is not equal to ∅
        starting_point = Min(Q)
        count = 1 + count
        if maxcount < count :
                maxcount = count
        else :
            count = count - 1

        Return maxcount
```

- This algo goes over the starting time and finishing time in increasing order. It keeps track of # colors used and the max # colors used.
- If the current point is starting time, it increases count by 1. It also updates the maximum.
- If current point is finishing time, it substracts count by 1.

b. We know that the number of colors colored is the overlapping intervals. This uses a priority queue method. The to queue and not queue takes $O(\log(n))$ per operation.

The endpoints take $O(n)$ and queue takes $O(\log(n))$.

Total $TC = O(n \cdot \log(n))$.

$\boxed{\text{Problem 4}}$

- G be directed graph on n vertices which given by input adjacent matrix V[1..n][1..n].

Example:
input :

- V2 → V1
- V3 → V1
- V4 → V1
- V5 → V1

has edges coming to it from all other vertices of G but no edges going out from it.

This is also called a sink. We need to get rid of the vertices and to do so, we need to check whether a certain index in the adjacent matrix cannot sink in i or j.

- If adjacent matrix = 1, then i can't be a sink because an edge is going from it
- If adjacent matrix = 0, then j there's no edge from i to j. j can't be sink. However, if we want j to be sink, then j should have edges coming towards it from every vertex.

- By doing so, we can just get rid of n-1 vertices by looking at n-1 entries of the matrix. We continue getting rid of the vertices until i or j gets to the end. OR increase i and j until one of them exceeds the number of vertices.

- Now that we've gotten rid of the non-sink, we can look if the last vertex is a non-sink or a sink. This allows us to do sink test for one vertex instead of all n vertices.

This algorithm runs on O(n) because we eliminated (n-1) non-sink vertex and checks for vertices for sink properties.

① TC for eliminating non-sink = O(n-1)
② TC for checking if vertex is sink or non-sink = O(n)

Total = ① + ② = O(n-1) + O(n) = O(n)