AOA midterm

Name: Shiyin Lin
UFID: 49619012

1 Coin Change

1.1 Pseudo-code

algorithm 1 CoinChange

Input: *float* money
Output: *int* NumOfCoin, *Array* changes

```
function CoinChange(float money)
    Coins := {100, 50, 20, 10, 5, 1, 0.25, 0.10, 0.05, 0.01}
    Changes := {}
    for each x in Coins do
        while money > x do
            tmp := 0
            tmp = money / x
            money = money
            NumOfCoin += tmp
            changes = APPEND(changes, {x} * tmp)
        end while
    end for
    if money ≠ 0 then
        NumOfCoin = -1
        changes = {}
        return NumOfCoin, changes
    end if
    return NumOfCoin, changes
end function
```

1.2 Proof of Correctness

Set the denomination of the dollar is $c_0, c_1, c_2, ..., c_k$
The change method can be expressed by the following formula

$$m_0c_0 + m_1c_1 + \ldots + m_kc_k = S$$

m_i = number of dollars of each denomination
c_i = the face value of the dollar
S = total amount
Assume there is a non-greedy algorithm for the optimal change solution

$$S_1 = m_0c_0 + m_1c_1 + ... + m_kc_k$$

and a greedy algorithm for the change solution

$$S_2 = n_0c_0 + n_1c_1 + ... + n_kc_k$$

Assume that starting from k, up to $x(x <= k)$ corresponding to the face value of the dollar, $m_x \neq n_x$
Because the greedy algorithm uses the largest denomination of dollars possible to make change each time, so $n_x > m_x$ (because the change solution for S_2 is different from S_1, there must be such an x that satisfies the condition)
Consider the minimum case, $n_x - m_x = 1$

$$1c_k = c_0+(c-1)c_0+(c-1)c_1+\ldots+(c-1)c_{k-1} > (c-1)c_0+(c-1)c_1+\ldots+(c-1)c_{k-1}$$

In S_1's change solution, $m(m < k)$ cannot be greater than or equal to c (because when $m_x > c$, it is necessary to replace c m_x with a higher denomination)

$$S_1 = m_0c_0+m_1c_1+\ldots+m_kc_{k-1} <= (c-1)c_0+(c-1)c_1+\ldots+(c-1)c_{k-1} < 1c_k$$

The sum of the remaining dollar denominations in S_1 that are less than c_k, and will not be greater than the denomination of one c_k
⇒Contradiction
⇒S1 not exist, so the Greedy algorithm is the optimal solution

1.3 Algorithm's running time

In the dollar change problem of the greedy algorithm, the worst case is (dollar value n)/(minimum denomination), i.e., the worst case is O(100n) = $O(n)$

2 Find Depth Find Longest Path

2.1 Pseudo-code

algorithm 2 Find Depth & Find Longest Path

Input: *Array* Tree

Output: *int* Depth or *int* LongestPath and *Array* Seq

```
function FindDepth(Array Tree)
    if Tree = null then
        return 0
    end if
    left, right := FindDepth(Tree.left), FindDepth(Tree.right)
    if l theneft ¿ right
        Depth = left + 1
        Seq = Append(Seq, Tree.left.Weight)
    else
        Depth = right + 1
        Seq = Append(Seq, Tree.right.Weight)
    end if
    return Depth, Seq
end function

function FindLongestPath(Array Tree)
    if Tree = null then
        return 0
    end if
    left, right := FindDepth(Tree.left), FindDepth(Tree.right)
    // The difference between find max depth and find longest path is that
    // the maximum depth is increasing the depth one layer at a time, and
    // the longest path is increasing the weight of the node at a time.
    if l theneft ¿ right
        LongestPath = left + Tree.left.Weight
        Seq = Append(Seq, Tree.left.Weight)
    else
        LongestPath = right + Tree.right.Weight
        Seq = Append(Seq, Tree.right.Weight)
    end if
    return LongestPath, Seq
end function
```

2.2 Proof of Correctness

Inductive method
We can simply set that $F(n)$ is the depth of node n ($F(0)$ is the root node), then the depth of node $n+1$ can be simply introduced as $F(n) = F(n+1) + 1$; for the longest path, then $F(n)$ is the longest path from the leaf node to node n, and simply introduce $F(n) = F(n+1) + n.weight$.
Initialization
From the root node, iterate over the left and right child nodes
Recursive
For the nth node, iterate over the left and right child nodes. For the maximum depth, take $thelargervalue + 1$; for the longest path, take $thelargervalue + node.weight$ and return the value which is the left or right value of the $n-1$ node.
Recursive termination condition
Node is null, return 0

2.3 Algorithm's running time

Due to the use of depth-first search traversing the entire binary tree, assuming that the tree has n nodes, the complexity is $O(n)$

3 Dichotomous search

3.1 Pseudo-code

algorithm 3 DichotomousSearch

Input: *Array* array

Output: *int* MaxNum

```
function DichotomousSearch(Array array)
    start, end := 0, len(array) - 1
    while start < end do
        mid = (start + end)/2
        if array[mid] < array[end] then
            end = mid
        else
            start = mid
        end if
    end while
    MaxNum = array[start]
    return MaxNum
end function
```

3.2 Proof of Correctness

Cyclic invariants
The subarray $A[start, end]$ must contain the maximum value, which can be understood as:
1. the search range $[start, end]$ is not empty, i.e. $low <= high$
2. all elements in the left side of the search range (i.e., within the range $[start0, start - 1]$) are less than the maximum value, where $start0$ is the initial value of $start$
3. all elements on the right side of the search range (i.e., within the range $[end + 1, end0]$) are less than the maximum value, where $end0$ is the initial value of end
Initialization
Initialize the array to the whole array, the search range is $[start0, end0]$, at this time both sides of the search range are the empty set.
Iteration
Suppose the invariant holds in the nth iteration, then in the n+1st iteration, if
1. $A[mid] < A[end]$, then $end = mid$ and all elements in $[mid + 1, end0]$ are smaller than $A[mid]$, while the right search range remains the same, then the invariant is valid
2. $A[mid] > A[end]$, then $start = mid$, all elements in $[start0, mid - 1]$ are smaller than $A[mid]$, while the left search range remains unchanged, then the invariant is valid.
Termination Conditions
1. when $start = end$, there is only one number in the array $A[start, end]$, which is the maximum value
2. when $start > end$, the array $A[start, end]$ is empty and the whole array is outside the array, according to the invariant, there is no such case

3.3 Algorithm's running time

There are n elements in total, and the size of the interval for each lookup is $n, n/2, n/4, ..., n/2^k$, where k is the number of cycles.
Since $n/2^k$ is rounded $>= 1$, i.e., let $n/2^k = 1$.
$k = log2n$, so the time complexity can be expressed as $O(logn)$

4 BONUS

Suppose the array of currency denominations is a[...] (arranged from small to large)
First of all, the simplest mathematical derivation leads to the fact that when the previous currency denomination is a multiple of the latter, it satisfies $a[i] * n = a[i+1]$, which is a sure way to satisfy the greedy algorithm.
But this is only a sufficient condition, that is, satisfying this requirement must be able to use greedy, but the greedy case is not only this condition. Take the case of 20 and 50 dollar denominations, for example, they are not multiples, but they can satisfy the greedy algorithm.
For this case where there is no multiplicative relationship, the condition should be satisfied that for $a[i] < a[i+1]$, we must be able to find an integer $k_i >= 1$ that satisfies $(k_i - 1) * a[i] < a[i+1]$ and $k_i * a[i] > a[i+1]$.
At this point for a change problem like $k_i * a[i]$, we can use $a[i+1] + k_{i-1} * a[i-1] + ... + k_0 * a[0]$ instead, and the number of sheets cannot be greater than k_i.

Briefly.

If the greedy algorithm is satisfied when the adjacent currencies are multiplicative

$$a[i] * n = a[i+1]$$

If the adjacent currencies do not satisfy the multiplicative relationship, then

$$k_i a[i] = a[i+1] + k_{i-1} a[i-1] + ... + k_0 a[0]$$

and k_i satisfies

$$k_i >= 1 + k_{i-1} + ... + k_0$$