COT5405 - Analys of Algorithms
Mid-Exam

Qinxuan Shi
UFID: 83518162

1 Problem 1

Consider the problem of providing change to an arbitrary amount N using US currency denominations, i.e \$0.01, \$0.05, \$0.10, \$0.25, \$1, \$5, \$10, \$20, \$50, \$100. Find a polynomial algorithm that, when given N, finds the exact change (or indicates that such change is not possible) using the minimum number of coins/banknotes.

1.1 Pseudo-code of the Algorithm

```
Algorithm 1 Provide_change(N, A)
Input: an arbitrary amount N, an sorted array A[0.01,0.05...50,100] contains all kinds of different currency denominations.
Output:
  using count to represent the sum of coins/banknotes.
  using m to represent the notes in A that is the largest notes less than N.

  count = count + N/m
  Provide_change(N%m, A)

  if N >0 then
      change is not possible
  end if
```

1.2 Proof of the algorithm's correctness

There are so many ways for us to provide change to N, but in order to get the minimum number of coins/banknotes, it is easy for us to come up in mind that we should use largest banknotes first, which is exactly a greedy choice.

Firstly, we are going to prove that this problem could be solved by greedy.

We assume that our first greedy choice is T, and the note is k, so the amount of money left is N-T_a = N'. It is clear that the subproblem of the money N' has the same structure of total problem, and could be solved as before. What's more, the optimal choice T' of N' fulfills that $T \cup T'$, which is exactly the optimal choice of the total problem.

Then, we are going to prove that the greedy choice is an optimal choice.

It is known that for N dollars, different coins/banknotes use a, b, c to j and 0.01a + 0.05b + 0.1c + ... + 20h + 50i + 100j = N. We have the case P, which uses the largest banknotes as much as possible.

Suppose there is an optimal case P', where the number of different coins and notes is A, B... J, fewer 100 dollars notes are used (J<j), and fewer total notes are used (A + B +...+ J<a + b +...+ j), that is, the optimal solution P' is obtained with fewer 100 dollars notes.

We make $k = 100 * (j - J)$, K dollar notes need (K / 50) numbers of 50 dollar notes, (K%50/20) numbers of 20 dollar notes and other notes like this. It is easy to know that to reduce (j-J) number of 100 dollar notes, you need to increase at least (100 * (j-J) / 50) number of 50 dollar notes, which means $2 * (j - J)$ numbers of 50 dollar notes are needed to replace (j-J) number of 100 dollar notes. It is contradict to the assume that P' has fewer notes than P, which means that there can be no better solution using less 100 dollar notes than P. Exactly, it is the same for other notes like 50 dollar notes when N is less than 50 but more than 20. Therefore, giving priority to the use of large banknotes is a correct greedy choice.

1.3 Algorithm's running time

According to the pseudo-code, the algorithm exactly only considers every element in A once, and compute the count as well as update the N once for every kind of note, which is O(1).

To conclude, the overall running time for the algorithm is O(n).

2 Problem 2

Given a tree, provide an efficient algorithm that finds the length of and the actual sequence for the longest path starting at the root and terminating at a leaf. If we now assume that tree edges have weights, how does the algorithm need to be modified to accommodate the generalization?

2.1 Pseudo-code of the Algorithm

Algorithm 2 Height(T)

```
Input: tree T
Output:
  using T(L) to represent the left subtree of the parent node, and T(R) to represent the right subtree.
  if T = $\emptyset$ then
      return 0
  else
      Height(T) = max(Height(T(L)), Height(T(R))) + 1
  end if
```

Algorithm 3 Sequence(T)

```
Input: tree T
Output:
  using T(L) to represent the left subtree of the parent node, and T(R) to represent the right subtree.
  using R[] to represent the sequence for the longest path.

  add root of T to R[]
  if T(L) or T(R) is not equal to $\emptyset$ then
      if Height(T(L)) $\geq$ Height(T(R)) then
          Sequence(T(L))
      else
          Sequence(T(R))
      end if
  end if
```

2.2 Proof of the algorithm's correctness and running time

1. Height(T)

The definition of binary tree itself recursively defines a binary tree as a tree composed of left and right subtrees, as a result, in order to get the height of the binary tree, it is easy for us to think about divide and conquer. We could divide the tree into two parts, and each part of the tree is also a binary tree, which means that the sub-problem is the same as the total problem. What's more, to get the height of the tree T', we only need to compare the left subtree with the right subtree and plus 1, since the sub-problems could finally combined together to get the total height of T.

Firstly, we unroll the tree, and we could find that for each node, we need to compare with empty set for three times(the node itself, its left subtree and right subtree). For each node, it also needs to add once. So, the overall T(n) = 3n, which equals to T(n)=O(n).

Secondly, according to the pseudo-code, we could get the structure of the algorithm Height(T):

$$T(n) = T(n_L) + T(n_R) + 1$$

Since the left subtree and right subtree both have at most half of n(nodes), then the structure can be tranformed to

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

According to the Master theorem, $T(n) = O(n)$.

2. Sequence(T)

To get the sequence, we only need to care about the subtree that has larger height. Since to each tree, the longest path must come from the larger height part. We get the root of the tree, then we replace the tree to the subtree that has larger height, we do not care about the situation of the other part since the longest path is not related to that part at all. This method is almost similar like binary seach, it only caculate one part instead of both two parts. When both T(L) and T(R) are empty set, we get the sequence for the longest path.

According to the pseudo-code, we could get the structure of the algorithm Sequence(T):

$T(n) = T(n_L) + O(n)$ (suppose that left subtree has larger height, it is the same with right)

the structure can be tranformed to

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

According to the Master theorem, $T(n) = O(n)$.

2.3 Modification

If we now assume that tree edges have weights, and we want to get the heaviest sum of the weights, we only need to change the step that add 1 to the height.

Algorithm 4 Weight(T)

```
Input: tree T
Output:
  using T(L) to represent the left subtree of the parent node, and T(R) to represent the right subtree.
  using W(T) to represent the weight of the way from the root of T to it's parent.
  The total root has W(T) = 0;

  if T = ∅ then
      return 0
  else
      Weight(T)=max(Weight(T(L)), Weight(T(R))) + W(T)
  end if
```

Since we only change the compute step, there is exactly a few differences to get the sequence of the heaviest path, so I'm not going to explain it.

3 Problem 3

Suppose you are given an array A[1..n] of distinct sorted integers that have been circularly shifted k positions to the right (for an unknown k). For example, [35, 42, 5, 15, 27, 29] is a sorted array that has been circularly shifted k = 2 positions, while [27, 29, 35, 42, 5, 15] has been shifted k = 4 positions. We can obviously find the largest element in A in O(n) time. Describe an O(log n) algorithm.

3.1 Pseudo-code of the Algorithm

Algorithm 5 FindLargest(A, left, right)

```
Input: Array A[1..n] of distinct sorted integers that have been shifted k positions to the right, left be the index
  of the left boundary, and right be the index of the right boundary.
Output:
  mid = (left+right)/2
  if A[mid] > A[left] then
      if A[mid] > A[mid+1] then
          largest=A[mid]
      else if A[mid] < A[mid+1] then
          FindLargest(A, mid+1, right)
      end if
  else
      if A[mid] > A[mid-1] then
          FindLargest(A, left, mid-1)
      else if A[mid] < A[mid-1] then
          largest=A[mid-1]
      end if
  end if
  return largest
```

3.2 Proof of the algorithm's correctness

Although the array A[1..n] has been circularly shifted k positions to the right and the k is unknown, there are still characteristics that we could find. Firstly, the array is still sorted, but is partly sorted, A[0..k-1] is sorted and A[k..n-1] is also sorted. Secondly, the smallest number is always behind the largest number(when k=0, we could suppose that the largest number is A[-1]. When we emplement the algorithm, we need to consider this kind of exceptional cases, but I do not conclude this in pseudo-code. Exactly, in some programming languages like python, people can get the last element of the array by using A[-1]). Then, we can figure out that any i in array A, if A[i] >A[i+1], A[i] must be the largest number and A[i+1] must be the smallest number. What's more, there are only three situations of the position of these two numbers.

1. Both the largest number and smallest number are in the left part of array.

2. Both the largest number and smallest number are in the right part of array.

3. The largest number or the smallest number is the middle of array.

According to three different positions, we could find that A[left] is greater than A[i] when i>k-1(A[left] represents the number has the smallest index of array). We could firstly compare the middle element of the array A[mid] with the element A[left]. If the position is like situation 1, then it is easy to know that A[left] is greater than A[mid]; if the position is like situation 2, then A[left] is less than A[mid](situation 3 can be concluded into the situation 1 and 2). After that, we are going to consider the situation 3, if there is A[mid] greater than A[mid+1] under situation 2 or A[mid] less than A[mid-1] under situation 1, the largest number and smallest number are found. If not found, we will divide the problem into two parts just like binart search and we are going to only consider the subproblem which is the part that the two numbers are in. The subproblem could be solved by recursion.

3.3 Algorithm's running time

When $n > 2$, we could get the structure from the pseudo-code:

$$T(n) = T(n/2) + k, (k \in N)$$

We supposed that $T(n) \leq klog_b n$, we are going to try to prove this.

$n = 1, T(n) = 1$, it is clearly to know this.

$n = 2, T(n) \leq c$, when there are two elements in array, we only need to compare two numbers once to get the largest number.

$$n > 2,\ T(n) \leq T(n/2) + c \leq klog_b(n/2) + c$$

we could attempt to choose b=2, so we could get:

$$T(n) \leq klog_2(n/2) + c$$

$$= k(log_2(n) - 1) + c$$

$$= klog_2(n) - k + c$$

So, it is clear that we just need to choose k that is at least as large as c, and we get

$$T(n) \leq klog_2(n)$$

To conclude, $T(n) = O(logn)$.

This could also be proved by unrolling the recurrence, but I'm not going to prove this since the structure of the algorithm is almost similar like Binary Search's.

4 problem 4

To problem 1, the greedy solution does not means fit for all kinds of notes. For example, there are three kinds of notes, 1, 5 and 7. When you need to change 10 dollars. If 7 dollar notes are used first, the number of sheets is 4 (1 + 1 + 1 + 7). However, if only 5 dollar notes are used, the number of sheets is 2 (5 + 5). So the algorithm is no correct.

So, For problem 1, the most general set of currency, we use C[0..n] to represent this set.

1. For each i and j, i=j+1, C[j]>2*C[i].

2. And there must have odd number in C[0..n], if not, there must exist the a*C[i]+b*c[j]+...+x*c[k]=1.