CS 600 Advanced Algorithms

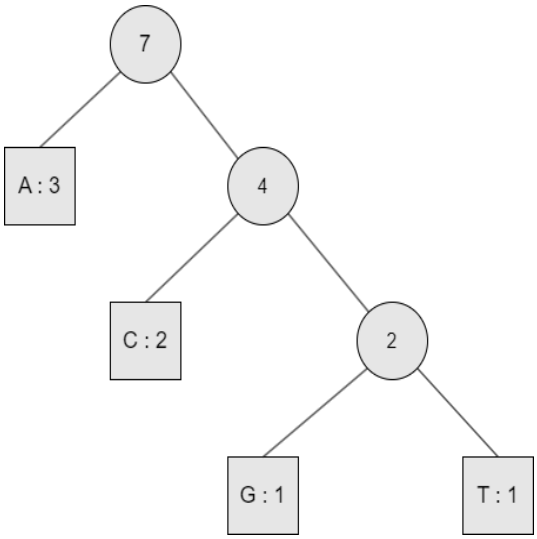
Sri Dhanush Reddy Kondapalli

CWID: 10476150

Homework 5

1 R-10.5.7 Fred says that he ran the Huffman coding algorithm for the four characters, A, C, G, and T, and it gave him the code words, 0, 10, 111, 110, respectively.

Yes the code mentioned by Fred is possible for the following frequencies of the four characters A, C, G, and T
A : 3, C : 2, G : 1, T : 1 and the following binary tree



2 C-10.5.16 Give an example set of denominations of coins so that a greedy change making algorithm will not use the minimum number of coins.

To reduce the number of coins, a greedy algorithm selects the highest denomination coin first. A greedy algorithm will not be able to utilize the fewest coins if it cannot choose the coins with the highest denomination first, i.e. we don't have any coins with higher denominations than the others, which is only feasible when all the coins have the same denomination. As a result, we just need pennies because we can alter any value with a penny, hence the denomination set will be {1}.

3 A-10.5.27 Describe an $O(n + m)$-time method for determining if a given string, Y, of length m is a sub-sequence of a given string, X, of length n.

To determine if string Y is an isSubsequence of string X, we use a greedy approach.
Algorithm isSubsequence X, Y:

```
Input: A string X, and a string Y
Output: True if Y is a subsequence of X, else False
Initialize a counter j = 1
for i = 1 to X.length() do
    if j <= Y.length() and X[i] == Y[j]
        j = j + 1
if j == Y.length() + 1
    return True
else
    return False
```

In the above algorithm, the for loop runs n times since the length of the string X is n, and the increment of j happens m times in the best case scenario. As a result, the method has a time complexity of O(n + m).

4 R-11.6.1 Characterize each of the following recurrence equations using the master theorem (assuming that $T(n) = c$ for $n < d$, for constants $c > 0$ and $d \geq 1$).

a) $T(n) = 2T(n/2) + \log n$
Here a = 2, b = 2 and f(n) = log n

According to Master Theorem $n^{\log_b a} = n$. Thus we are in Case 1 where f(n) is $O(n^{1-\epsilon})$ for $0 < \epsilon < 1$. This means that
T(n) is $\Theta(n)$ by Master theorem.

b) $T(n) = 8T(n/2) + n^2$
Here a = 8, b = 2 and f(n) = n^2
According to Master Theorem $n^{\log_b a} = n$. Thus we are in Case 1 where f(n) is $O(n^{3-\epsilon})$ for $\epsilon = 1$. This means that
T(n) is $\Theta(n^3)$ by Master theorem.

c) $T(n) = 16T(n/2) + (n \log n)^4$
Here a = 16, b = 2 and f(n) = $n^4 \log^4 n$
According to Master Theorem $n^{\log_b a} = n^4$. Thus we are in Case 2 with k = 4, for f(n) is $\Theta(n^4 \log^4 n)$ for $\epsilon = 1$. This means that
T(n) is $\Theta(n^4 \log^5 n)$ by Master theorem.

d) $T(n) = 7T(n/3) + n$
Here a = 7, b = 3 and f(n) = n
According to Master Theorem $n^{\log_b a} = n^{\log_3 7}$. Thus we are in Case 1 where f(n) is $O(n^{1-\epsilon})$ for $\epsilon = \log_3 7 - 1$. This means that
T(n) is $\Theta(n^{\log_3 7})$ by Master theorem.

e) $T(n) = 9T(n/3) + n^3 \log n$
Here a = 9, b = 3 and f(n) = $n^3 \log n$
According to Master Theorem $n^{\log_b a} = n^2$. Thus we are in Case 3, since f(n) is $O(n^{2+\epsilon})$ for $\epsilon = 1$ and
$f(n/b) = 9((n^3 \log(n/3))/27) = 1/3f(n) - \log 3/3$. This means that
T(n) is $\Theta(n^3 \log n)$ by Master theorem.

5 C-11.6.10 Characterize the running time, T(n), in this case, using a recurrence equation, and use the master theorem to determine an asymptotic bound for T(n).

If we make the adjustments listed above to the Stooge-sort method, we will solve 3/4th of the issue three times. As a result, the new recurrence relation would be

$$T(n) = 3T(3n/4) + O(1)$$

Here a = 3, b = 4 / 3 and f(n) = O(1).
According to Master Theorem $n^{\log_b a} = n^{\log_{4/3} 3}$ Thus we are in Case 1 where f(n) is $O(n^{\log_{4/3} 3-\epsilon})$ for $\epsilon = \log_{4/3} 3$. This means that T(n) is $\Theta(\log_{4/3} 3)$ which can be further by simplified to $O(n^{\log 3/\log 1.33}) = O(n^{3.82})$

6 A-11.6.17 Design an $O(n \log n)$-time algorithm for computing the skyline of S.

Algorithm Skyline(S):

```
Input: A Set S of sub intervals with n tuples (ai, bi, hi)
       where 0 <= ai < bi <= 1 and hi is the
       height of the interval [ai, bi]
Output: The skyline of S
Sort the tuples by their x co-ordinates
Create a BST T containing the tuples sorted by height h
Create set O containing the output
for i = 1 to n do
    T.add(Si)
    if Si is the highest node in T then
        O.add([ai, hi])
    t  T .remove(Si)
    if t is the highest node in T then
         t 2 Highest  node in T
        if t2 is not empty then
            O.add([bi, t2.h ])
        else
            O.add([bi, 0])
return O
```

The Sorting takes $O(n \log n)$ time.
Looping through the objects of S will take O(n) time.
Adding and removing the objects of S in T will take O(log n) time.
Adding to the output set O will take O(1) time.
Therefore the whole algorithm runs in O(n log n) time.

7 R-12.8.5 What is an optimal solution to the 0-1 knapsack problem for S assuming we have a sack that can hold objects with total weight 18? Show your work.

Solving the above problem using Algorithm 12.13 the array B is as given below:

Items	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
a(12,4)	0	0	0	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
b(10,6)	0	0	0	0	12	12	12	12	12	12	22	22	22	22	22	22	22	22	22
c(8,5)	0	0	0	0	12	12	12	12	12	20	22	22	22	22	22	30	30	30	30
d(11,7)	0	0	0	0	12	12	12	12	12	20	22	23	23	23	23	30	31	33	33
e(14,3)	0	0	0	14	14	14	14	26	26	26	26	26	34	36	37	37	37	37	44
f(7,1)	0	7	7	14	21	21	21	26	33	33	33	34	34	41	43	44	44	44	44
g(9,6)	0	7	7	14	21	21	21	26	33	33	33	34	34	41	43	44	44	44	44

Therefore the maximum benefit than can be achieved is 44 with objects {e, c, b, a}

8 C-12.8.14 Show that we can solve the telescope scheduling problem in $O(n)$ time even if the list of n observation requests is not given to us in sorted order, provided that start and finish times are given as integer indices in the range from 1 to n^2.

Algorithm P(L,i):

```
Input: List L of n observation requests and
       request i whose predecessor is to be searched
Output: Predecessor of request i
p = 0
for k = 1 to n:
    if p < endk and endk <= starti then
        p = endk
return p
```

With this algorithm we can still use the telescope scheduling algorithm to find the max benefit in O(n + n) = O(n) time.

```
B[0] = 0
for i = 1 to n do
```

```
B[i] = max{B[i - 1], B[P(i)] + bi }
```

9 A-12.8.30 Describe an efficient algorithm for taking any character string, S, of length n, and finding the longest subsequence of S that is a palindrome.

Algorithm:

```
Input: String S of length n,
       considering our example: 'I EAT GUITAR MAGAZINES'
Output: Longest palindromic subsequence
We use:
L - left index of current index
R - right index of current index
dp - a 2d array that stores the answers to state
Solve:
    if L > R, then:
        return 0
    else if L = R, then:
        return 1
    else if dp[L][R] not equals -1:
        return dp[L][R]
    else if S[L] = S[R] , then:
        dp[L][R] = 2 + solve(dp, S, L+1, R-1)
    else:
        moveL = solve(dp, S, L+1, R)
        mover = solve(dp, S, L, R-1)
        dp[L][R] = maximum(moveL, moveR)
    return dp[L][R]
```

We make certain to iterate over all conceivable lengths. Then we explore the lengths in increasing order, starting with smaller subproblems and progressively progressing to larger ones.

When the end of S matches, we proceed to the next element by increasing the length + 2. If they do not match, we search for narrower ranges (beginning with L or L+1). We save the result of the comparison in dp[1][n].

This costs us $O(n^2)$ time, where n is the length of the string.