Homework 5 - Solution

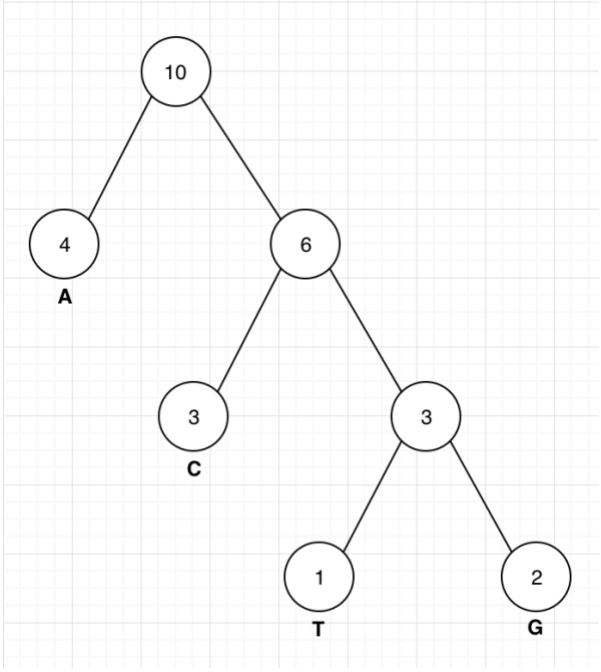
R-10.7

An interesting greedy approach to solving the text compression problem using variable length encoding approach is Huffman coding. This method produces a variable-length prefix code for *X* based on the construction of a proper binary tree *T* that represents the code. Each edge in *T* represents a bit in a code word, with each edge to a left child representing a "0" and each edge to a right child representing a "1." Each external node *v* is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the edges in the path from the root of *T* to *v*.

To generate the Huffman encoding tree, it requires all the frequencies to be sorted in an increasing order. Thus, with each run we pick the two lowest frequencies and connect them with a parent node, assigning its frequency as the summation of the frequencies of its child elements.

So, let's consider that the frequency distribution for a given string containing *A*, *C*, *G*, and *T* is given as: [A,4], [C,3], [G,2], [T,1].

The Huffman tree for the above frequency distribution can be given as:



Thus we can say that the codes generated by Fred can be a result of Huffman coding algorithm output.

C-10.6

Consider a situation where the denominations available are {5, 20, 30}. Now, suppose you want to get the change for 40 bucks. The greedy change making algorithm will look for the biggest denomination less that the amount and use that toward the coin changes to be provided. Thus in

the above case it will return us with 1 coin of 30 bucks and 2 coins of 5 bucks, as a total 3 coins for 40 bucks. But notice that we had a denomination (coin) for 20 bucks and if we used that we could get 40 bucks with 2 coins of 20 bucks. Thus, in this situation we can say that the greedy change making algorithm does not return us with the minimum number of coins.

A-10.7

This problem is an example of a subsequence problem. We need to design an algorithm for determining if a given string, *Y*, of length *m* is a subsequence of a given string, *X*, of length *n*. We can do this recursively, by initializing two pointers to point to start of each strings and matching the values in those strings at those pointer locations. If we find a match we move both pointers to the next location in the string, and if they do not match then we just move by one character in string *X* as it is the original reference string. With this approach, we would be traversing through each of the character of both strings *X* and *Y* once, thus the total running time of this algorithm is $O(n + m)$.

Algorithm isSubSequence (string array *X*, string array *Y*, int *n*, int *m*):
Input: Original string *X* of length *n*, transmitted string *Y* of length *m*
Output: If string *Y* is subsequence of string *X*

```
if m=0 then
        return true;
if n=0 then
        return false;
if Y[m-1]=X[n-1] then
        return isSubSequence(X, Y, n-1, m-1);

return isSubSequence(X, Y, n-1, m);
```

R-11.1

For divide and conquer recurrence equations of the form

$$T(n) = \begin{cases} c, & n < d \\ aT(b) + f(n), & n \geq d \end{cases}$$

the master theorem has 3 cases,

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\theta(n^{\log_b a})$
2. if $f(n)$ is $\theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$

a. $T(n) = 2T(n/2) + \log n$
$n^{\log_b a} = n^{\log_2 2} = n$, and $f(n) = \log n$.
Here, $n^{\log_b a} > f(n)$, thus we are in case 1.
Thus, $T(n) = \theta(n^{\log_b a}) = \theta(n)$.

b. $T(n) = 8T(n/2) + n^2$
$n^{log_b a} = n^{log_2 8} = n^3$, and $f(n) = n^2$.
Here, $n^{log_b a} > f(n)$, thus we are in case 1.
Thus, $T(n) = \theta(n^{log_b a}) = \theta(n^3)$.

c. $T(n) = 16T(n/2) + (n \log n)^4$
$n^{log_b a} = n^{log_2 16} = n^4$, and $f(n) = n^4 \, log^4 n$.
Here, $n^{log_b a} \approx f(n)$, thus we are in case 2.
Thus, $T(n) = \theta(n^{log_b a} \, log^{k+1} n) = \theta(n^4 \, log^5 n)$.

d. $T(n) = 7T(n/3) + n$
$n^{log_b a} = n^{log_3 7} = n^{1.77}$, and $f(n) = n$.
Here, $n^{log_b a} > f(n)$, thus we are in case 1.
Thus, $T(n) = \theta(n^{log_b a}) = \theta(n^{log_3 7}) = \theta(n^{1.77})$.

e. $T(n) = 9T(n/3) + n^3 \log n$
$n^{log_b a} = n^{log_3 9} = n^2$, and $f(n) = n^3 \log n$.
Here, $n^{log_b a} < f(n)$, thus we are in case 3.
Thus, $T(n) = \theta(n^{log_b a}) = \theta(n^3 \log n)$.

C-11.4
The algorithm for Stooge-sort is give as:

Algorithm StoogeSort(A, i, j):
Input: An array, A, and two indices, i and j, such that $1 \leq i \leq j \leq n$
Output: Subarray, $A[i..j]$, sorted in nondecreasing order
$n \leftarrow j - i + 1$ // The size of the subarray we are sorting
if $n = 2$ **then**
  if $A[i] > A[j]$ **then**
    Swap $A[i]$ and $A[j]$
else if $n > 2$ **then**
  $m \leftarrow \lfloor n/3 \rfloor$
  StoogeSort$(A, i, j - m)$ // Sort the first part
  StoogeSort$(A, i + m, j)$ // Sort the last part
  StoogeSort$(A, i, j - m)$ // Sort the first part again
return A

Algorithm 11.5: Stooge-sort.

If $n < 4$, then the algorithm works pretty straight forward in constant running time. For $n > 4$, we will break $T(n)$ recursively into 3 parts, each of size $3n/4$. Thus the divide and conquer recurrence equation for the given method can be written as:

$$T(n) = \begin{cases} c, & n \leq 4 \\ 3T(3n/4) + cn, & n > 4 \end{cases}$$

Thus, in this case by master theorem,

$n^{log_b a} = n^{log_{4/3} 3} = n^{3.82}$, and $f(n) = cn$.

Here, $n^{log_b a} > f(n)$, thus we are in case 1.

Thus, $T(n) = \theta(n^{log_b a}) = \theta\left(n^{log_{4/3} 3}\right) = \theta(n^{3.82})$.

A-11.6

Every building has one side on the *x*-axis (2 co-ordinates) & a height *h* which is associated to each building. Moreover, all buildings have a common bottom with one side on the *x*-axis. Each building has 3 parameters.

Algorithm computeSkyline(S, low, high):

Input: Array of building S with each building having 3 parameters (left x co-ordinate, height, right x co-ordinate) of size n.

Output: Array of rectangular strips where each strip has a left x co-ordinate & height

```
If low==high then
    return [ ( S[low].left, S[low].height ) , ( S[low].right, 0) ]
mid ➔ (low + high) /2
SkylineOne ➔ computeSkyline (S, low, mid)
SkylineTwo ➔ computeSkyline (S, mid+1, high)
return merge (SkylineOne, SkylineTwo)
```

Algorithm merge (SkylineOne, SkylineTwo):

Input: Two different Skylines

Output: Merged Skyline with strips (x co-ordinate, height)

```
Initialize heightOne, heightTwo ➔ 0
InitializeArray final [ ]        // empty array
While i <len(SkylineOne) and j < len(SkylineTwo) do
    If SkylineOne[ i ][0] <  SkylineTwo[ j ][0] then
        Height1 ➔ SkylineOne[ i ][1]
        final.append( SkylineOne[ i ][0], max(Height1, Height2) )
        Increment i
    else do
        Height2 ➔ SkylineOne[ j ][1]
        final.append( SkylineOne[ j ][0], max(Height1, Height2) )
        Increment j
While i < len(SkylineOne) do
    final.append(SkylineOne[ i ])
    Increment i
While j < len(SkylineOne) do
    final.append(SkylineOne[ j ])
    Increment j
return final
```

The idea is to strip into two skylines and do merging. In the merging we compare the smaller *x* co-ordinate strip and add into the result and the height will be current maximum height from both the skylines. We are dividing the array *S* into two equal half- and each-time merge n elements.

Thus the recurrence equation would be $T(n) = 2T(n/2) + cn$.

Thus, in this case by master theorem,

$n^{log_b a} = n^{log_2 2} = n$, and $f(n) = cn$.

Here, $n^{log_b a} \approx f(n)$, thus we are in case 2.

Thus, $T(n) = \theta(n^{log_b a} log^{k+1} n) = \theta(\text{n log}\, n)$.

R-12.5

We can derive the solution for this by using algorithm 12.13 of the textbook for the optimal solution for above problem. It is {a, b, c, e} with total weight 18 and maximum benefit 44. As it is a 0-1 knapsack problem, we cannot consider fractional parts of weights.

Another solution is {a, d, e, f} with total weight 15 and maximum benefit 44.

C-12.5

The telescoping algorithm produces $O(n)$ timing assuming the start and finish times are provided in non-decreasing order. Since finish times are between 1 and n^2, the start times must be between 0 and n^2 too. So the problem reduces to sorting a list of non-negative INTEGERS (if not integers, we have to somehow clearly map them first to a set of non-negative integers) less than n^2. We can represent any such integer *k* as an ordered pair of binary numbers (i, j), where *i* is the half higher order bits and *j* is the half lower order bits, both of integer values $\leq n$. Now we can use radix sort on these tuples and provide non-decreasing start and finish times in $O(n)$.

So, the total running time in worst case would also be $O(n)$.

A-12.3

We have a string, *S*, of *n* uppercase letters and a function *valid(s)* which can check whether a character string, *s*, is a valid English word in $O(1)$ time. We want to break *S* into a sequence of valid English words. We observe for some $i < n$, either $S[0: i-1]$ is a valid English word or $S[i: n-1]$ is breakable and contains a valid English word. If one of these conditions fail then our input is not valid. Here is our algorithm. If *S* is already a valid word, we simply return *S*. Otherwise we traverse *S* with an index *i*. If *valid*($S[0: i-1)$ is true, we recursively try to break the substring $S[i: n-1]$ into a set of English words. If this recursion fails, we record the substring $S[i: n-1]$ in a set (this is the memorization step in our algorithm). This is so that when we get back from the recursion, we won't have to check $S[i: n-1]$ again. At each successful recursive call, we record the string formed by the sub-string $S[0: i-1]$, a space, and the substring $S[i: j]$, for some $i < j \leq n$.

When there are no more substrings to break, we return the accumulated result. Here is a pseudocode description.

Algorithm Break-String(S, M, O):
Parameters: String S, table M, output O

```
If S is in M then
        Return Fail
else if valid(S) then
        return S
end
for i= 1...n do
        L ← S[0: i − 1]
        R ← S[i: n − 1]
        If valid(L) then
                R′ ← Break-String(R)
                If R′ is not Fail then
                        O ← L+" "+R′
                Else
                        Add R to M
                End
        End
end
return O
```

Our worst case would be if we had a string for which each character was a valid word, such as AAAA...We perform the iteration *n* times and at each iteration, *i*, we recursively traverse a substring of size $n - i$. The complexity is $O(n^2)$.

A-12.4
We can solve this problem by using dynamic programming tabulation method. Initially, we construct a 2D Array of size equal to the length of the string *S* of length *n*.

Each character itself is a palindrome so we start by assigning 1 to all the diagonal elements in the table. This will take $O(n)$.

Suppose we have a *n*-characters sequence *S*, denoted as $S[1,n]$, $L(1,n)$ to be the length of the longest palindromic subsequence of $S[1,n]$. If the first and the last characters of *S* are the same, then $L(1,n) = L(2,n-1) + 2$. Otherwise, $L(1,n) = max(L(1,n-2), L(2,n-1))$. Then it comes to be a recurrence method. We will fill the rest of the 2D Array to store overlapping sub problems, which is also called memorization.

Since there are n^2 elements in the 2D array and calculating the value of each element takes $O(1)$ time, the time complexity of this method will be $O(n^2)$.