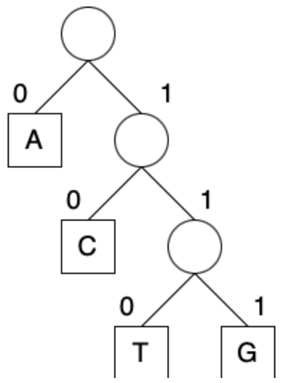
Romil V. Shah
20008692

Assignment 5
CS/CPE 600
Prof. Reza Peyrovian
Submission Date: 10/16/2022

Q1. No. 10.5.7

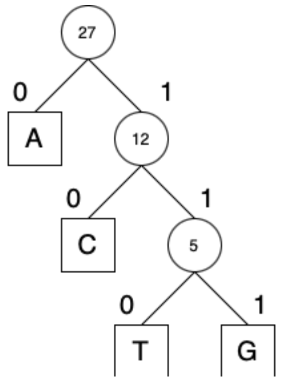
Fred says that he ran the Huffman coding algorithm for the four characters, **A**, **C**, **G**, and **T**, and it gave him the code words, 0, 10, 111, 110, respectively. Give examples of four frequencies for these characters that could have resulted in these code words or argue why these code words could not possibly have been output by the Huffman coding algorithm.

Ans.



WORD	Frequency 1	Frequency 2	Frequency 3	Frequency 4
A	100	40	130	15
C	45	30	70	7
T	10	10	40	2
G	20	10	30	3

Example for Frequency 4:



Here, the code of characters, A, C, T, G have different lengths and are assigned in a way that most frequent used character to character which uses fewest number of bits and least used characters to most used. It satisfies the prefix code, that no code is a prefix of any other code.

As a frequency to appear in a word is more so it has been assigned the smallest bit i.e. 0.

As G is the least used word, it is assigned the largest bit, i.e. 111.

This technique helps in achieving the maximum compression.

Q2. No. 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.

Ans.

Greedy Algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So, the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.

It is an approach for solving a problem by selecting the best option available now. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach. This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

So, let us consider the below example:

Suppose value of d1, d2, d3, d4, d5, d6, d7 is 1, 5, 10, 20, 30, 75, 100 and 250.

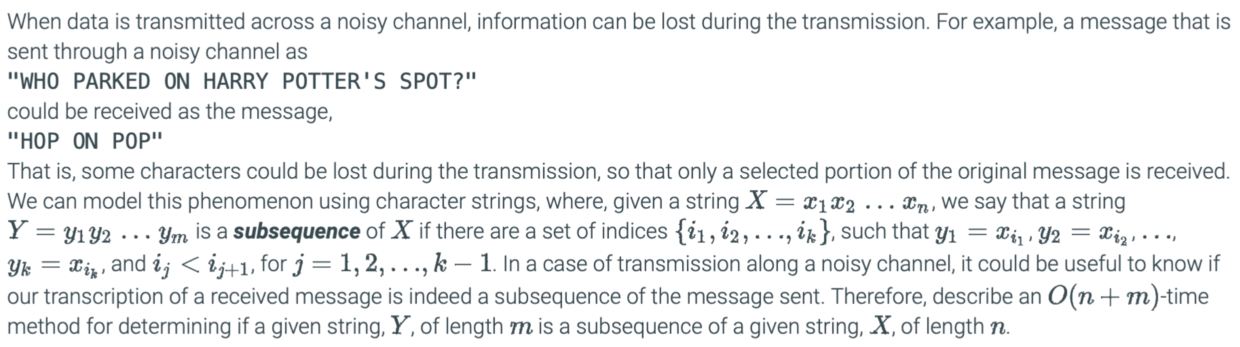
Now, we calcuclate the minimum numberof coins to get a value of 150 cents.

Using **Greedy Approach**, the minimum coins selected,
150 cents = 100 + 30 + 5 + 1 + 1 + 1 + 1 + 1

But the optimal solution will be,
150 cents = 75 + 75

So, we can say that the greedy algorithm will not use minimum number of coins.

Q3. No. 10.5.27



When data is transmitted across a noisy channel, information can be lost during the transmission. For example, a message that is sent through a noisy channel as
`"WHO PARKED ON HARRY POTTER'S SPOT?"`
could be received as the message,
`"HOP ON POP"`
That is, some characters could be lost during the transmission, so that only a selected portion of the original message is received. We can model this phenomenon using character strings, where, given a string $X = x_1 x_2 \ldots x_n$, we say that a string $Y = y_1 y_2 \ldots y_m$ is a ***subsequence*** of X if there are a set of indices $\{i_1, i_2, \ldots, i_k\}$, such that $y_1 = x_{i_1}$, $y_2 = x_{i_2}$, ..., $y_k = x_{i_k}$, and $i_j < i_{j+1}$, for $j = 1, 2, \ldots, k-1$. In a case of transmission along a noisy channel, it could be useful to know if our transcription of a received message is indeed a subsequence of the message sent. Therefore, describe an $O(n+m)$-time method for determining if a given string, Y, of length m is a subsequence of a given string, X, of length n.

Ans.

Let us consider a string K that is a string of length n and a subsequent string L of length m.

Using a greedy approach, we solve the problem in linear time.

a. We first start with the first character of the message received at the receiver end and finding the same character in the input transmission.
b. Now, we find the second character “O” which is the third character of the first word in the original transmission while ignoring the character which are already matched.
c. We keep calling the above 2 steps until all the characters in the output transmission have been received and matched,
d. Finally, we end the matching if there are no characters left to match with the original string K.

The matching will be done with the original string K of length n and subsequent string L of length m. So, total comparisons made will be O(n+m) which will be the time complexity for this algorithm.

Q4. No. 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) $T(n) = 2T(n/2) + \log\ n$

(b) $T(n) = 8T(n/2) + n^2$

(c) $T(n) = 16T(n/2) + (n\ \log\ n)^4$

(d) $T(n) = 7T(n/3) + n$

(e) $T(n) = 9T(n/3) + n^3 \log\ n$

Ans.

Master Theorem is a method for solving divide-and-conquer recurrence equations that is quite general and does not require explicit use of induction to apply correctly. The master theorem is a "cookbook" method for determining the asymptotic characterization of a wide variety of recurrence equations.

Namely, it is used for recurrence equations of the form

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

where $d \geq 1$ is an integer constant, $a \geq 1$, $c > 0$, and $b > 1$ are real constants, and f(n) is a function that is positive for $n \geq d$.

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

   $n^{\log_b a} = n^{\log_2 2} = n$
   $\log n = O(n^{\log_b a - \varepsilon})$. For $\varepsilon > 0$
   $= O(n^{1-\varepsilon})$ (Using case 1 of master method)
   $T(n) = \theta(n^{\log_b a})$
   $= \theta(n)$

b) $T(n) = 8T(n/2) + n^2$
   a= 8, b= 2, f(n) = n^2

   $n^{\log_b a} = n^{\log_2 8} = n^3$

   $n^2 = O(n^{\log_b a - \varepsilon})$. For $\varepsilon > 0$
   $= O(n^{3-\varepsilon})$ (Using case 1 of master method)
   $T(n) = \theta(n^{\log_b a})$

= $\theta(n^3)$

c) $T(n) = 16T(n/2) + (n \log n)^4$

a= 16, b= 2, f(n) = $(n \log n)^4$

$n^{\log_b a} = n^{\log_2 16} = n^4$

$(n \log n)^4 = \theta(n^{\log_b a} \log^k n)$

$T(n) = \theta(n^{\log_b a} \log^{k+1} n)$ (Using case 2 of master method)

= $\theta\ n^4(\log n)^5$

d) $T(n) = 7T(n/3) + n$

a= 7, b= 3, f(n) = n

$n^{\log_b a} = n^{\log_3 7} = n^{1.77}$

$n = O(n^{\log_b a-\varepsilon})$. For $\varepsilon > 0$

= $O(n^{1.77-\varepsilon})$ (Using case 1 of master method)

$T(n) = \theta(n^{\log_b a})$

= $\theta(n^{1.77})$

e) $T(n)=9T(n/3) +n^3 \log n$

a= 9, b= 3, f(n) = $n^3 \log n$

$n^{\log_b a} = n^{\log_3 9} = n^2$

$n^3 \log n = O(n^{\log_b a+\varepsilon})$. For $\varepsilon = 1$

Given, $a.f(n/b) \leq \delta f(n)$ for some $\delta < 1$

$a.f(n/b) = 9(n/3)^3 \log(n/3)$

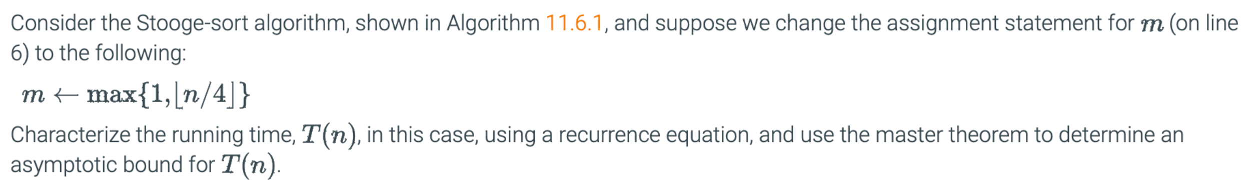
$= n^3/3 \log(n/3)$

$\leq \delta f(n)$. (when $\delta = 1/3$ and $n \geq 1$)

(Using case 3 of master method)

$T(n) = \theta(n^3 \log n)$

Q5. No. 11.6.10

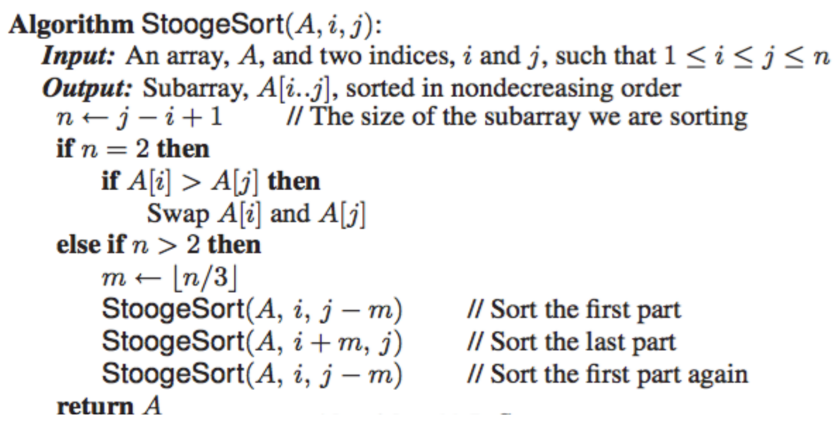


Consider the Stooge-sort algorithm, shown in Algorithm 11.6.1, and suppose we change the assignment statement for m (on line 6) to the following:

$m \leftarrow \max\{1, \lfloor n/4 \rfloor\}$

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

Ans.



```
Algorithm StoogeSort($A, i, j$):
  Input: An array, $A$, and two indices, $i$ and $j$, such that $1 \le i \le j \le n$
  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 $A$
```

The Stooge-Sort Algorithm says that if the input size, n = 1 or 2, then the algorithm sorts the input immediately. But, for n > 3, we call stooge-sort algorithm recursively into parts each of length ((3(n)) / 4) and in one loop we call stooge-sort 3 times recursively.

Recurrence Relation for the above algorithm will be:
T(n) = 3T ((3(n)) / 4) + cn

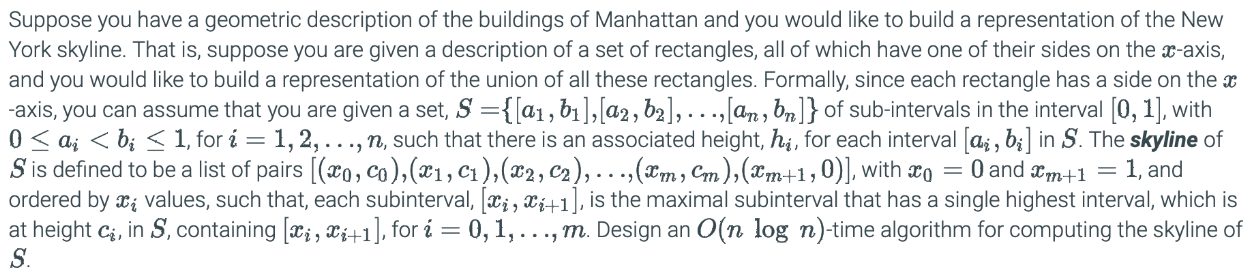
Using master method:
a = 3, b = 4, f(n) = n

$n^{\log_b a} = n^{\log_3 4} = n^{1.26}$

n = O($n^{\log_b a - \varepsilon}$). For $\varepsilon > 0$
  = O($n^{1.26-\varepsilon}$)   (Using case 1 of master method)

T(n) = θ($n^{\log_b a}$)
  = θ($n^{1.26}$)

Q6. No. 11.6.17



Suppose you have a geometric description of the buildings of Manhattan and you would like to build a representation of the New York skyline. That is, suppose you are given a description of a set of rectangles, all of which have one of their sides on the x-axis, and you would like to build a representation of the union of all these rectangles. Formally, since each rectangle has a side on the x-axis, you can assume that you are given a set, $S = \{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$ of sub-intervals in the interval $[0, 1]$, with $0 \leq a_i < b_i \leq 1$, for $i = 1, 2, \ldots, n$, such that there is an associated height, h_i, for each interval $[a_i, b_i]$ in S. The ***skyline*** of S is defined to be a list of pairs $[(x_0, c_0), (x_1, c_1), (x_2, c_2), \ldots, (x_m, c_m), (x_{m+1}, 0)]$, with $x_0 = 0$ and $x_{m+1} = 1$, and ordered by x_i values, such that, each subinterval, $[x_i, x_{i+1}]$, is the maximal subinterval that has a single highest interval, which is at height c_i, in S, containing $[x_i, x_{i+1}]$, for $i = 0, 1, \ldots, m$. Design an $O(n \log n)$-time algorithm for computing the skyline of S.

Ans.

The skyline is a collection of rectangular strips, each rectangle has a side on the x-axis, we assume that we are given a set, S = {[a1, b1], [a2, b2], . . . , [an, bn]} of sub- intervals in the interval [0,1], with 0 ≤ ai < bi ≤ 1, for i = 1, 2, … , n, such that there is an associated height, hi, for each interval [ai, bi] in S.

Here, we can use divide and conquer approach to solve the skyline of S.

1. First, we divide the set **S** into two subsets S_1 and S_2, then we keep dividing further recursively in two halves until a single element is left in the set.
2. Then, we start merging the elements in bottom-up fashion.
3. Merging will be done like merge sort. Start from the first strips of two skyline, compare their ***x*** coordinates. Pick the strip which has smaller ***x*** coordinate and add it to the result.
4. The height of added strip is considered as maximum of current heights from set S_1 and S_2.

As the algorithm divides the set into two halves every time, it will form the recurrence T(n) = 2 T(n/2) and takes O(log n) time (as the height of the tree). Merging will be done in O(n) time as the work done at the nodes of depth i.

So, the total running time for the above algorithm is O(n log n).

Q7. No. 12.8.5

Let $S = \{a, b, c, d, e, f, g\}$ be a collection of objects with benefit-weight values, $a:(12, 4), b:(10, 6), c:(8, 5), d:(11, 7), e:(14, 3), f:(7, 1), g:(9, 6)$. 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.

Ans.

Total Weight object can hold is 18.
Here,

1. a: (12, 4)
2. b: (10, 6)
3. c: (8, 5)
4. d: (11, 7)
5. e: (14, 3)
6. f: (7, 1)
7. g: (9, 6)

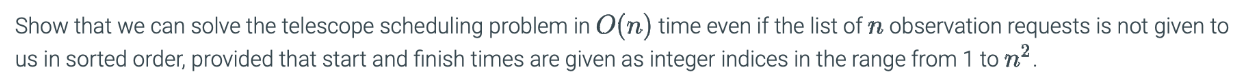
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
B, W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12, 4	1	0	0	0	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
10, 6	2	0	0	0	0	12	12	12	12	12	12	22	22	22	22	22	22	22	22	22
8, 5	3	0	0	0	0	12	12	12	12	12	20	22	22	22	22	22	30	30	30	30
11, 7	4	0	0	0	0	12	12	12	12	12	20	22	23	23	23	23	30	31	33	33
14, 3	5	0	0	0	14	14	14	14	26	26	26	26	26	34	36	37	37	37	37	44
7,1	6	0	0	0	14	14	14	14	26	26	26	26	26	34	36	37	37	37	37	44
9,6	7	0	0	0	14	14	14	14	26	26	26	26	26	34	36	37	37	37	37	44

As we fill up the values as per the formula:

$$B[k, w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1, w], B[k-1, w-w_k] + b_k\} & \text{else}. \end{cases}$$

We see that the maximum benefit is 44.
We will get the optimal solution from {(12,4), (10,6), (8,5), (14,3)}

Q8. No. 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.

Ans.

Algorithm for telescope scheduling:

Ordering of requests by finish times and an array, P, so that P[i] = pred(i), then we can fill in the array, B

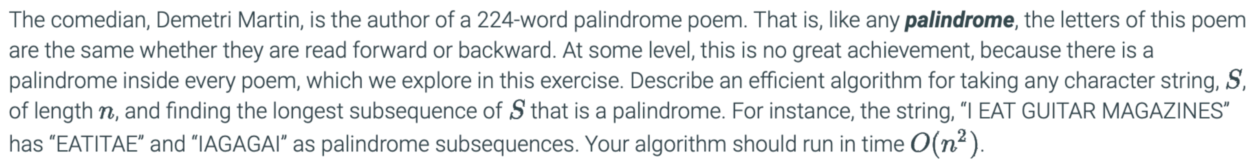
```
$B[0] \leftarrow 0$
for i = 1 to n do
        $B[i] \leftarrow \max\{B[i-1], B[P[i]] + b_i\}$ A
```

As the list of n observations is not sorted, it will take O(n log n) time to sort and O(n) time for the for loop in the algorithm. So, the total running time of this algorithm will be O(n log n + n).

For large value of n, we consider the value O(n) neglecting log n. Sorting of n observation can be easily done in O(n) time. So, we can compute the entire array B in O(n) time.

For input 1 to n^2, we implement dynamic programming to find the optimal solution. But, if the size varies to 2^n then, the brute force algorithm will work better.

Q9. No. 12.8.30



The comedian, Demetri Martin, is the author of a 224-word palindrome poem. That is, like any ***palindrome***, the letters of this poem are the same whether they are read forward or backward. At some level, this is no great achievement, because there is a palindrome inside every poem, which we explore in this exercise. Describe an efficient algorithm for taking any character string, S, of length n, and finding the longest subsequence of S that is a palindrome. For instance, the string, "I EAT GUITAR MAGAZINES" has "EATITAE" and "IAGAGAI" as palindrome subsequences. Your algorithm should run in time $O(n^2)$.

Ans.

Algorithm to find the longest subsequence of S that is a palindrome:

Input: Palindrome of string S of length [0 … n-1].
Output: The length L[i, j] of longest common sequence of a palindrome S[0 … n-1]

```
for i ← 1 to n do
        L[i, i] ← 1
for i ← 2 to n do
        L[i, i-1] ← 0
for i ← n-1 to 2 do
        for j ← i + 1 to n do
                if  L[i] = L[j] then
                        L[i, j] = L [i+1, j-1] + 2
                else
                        L[i, j] = max {L[i+1, j], L [i, j-1]}
return array L
```

Here, first, we check the first and the last character of the sequence. This will give us that either both characters are same, or they are not.

If both characters are same, we add 2 to the result and remove both characters and put it in the array L.

If both characters are different, we solve the problem by considering the first character and we compute the remaining subsequence. Then, we repeat the previous step choosing the last character and solve it.

We will record the maximum of both the results.

The algorithm uses 2 dominated nested for loops, with both inner and outer iterating n times. It has an if statement and assignment inside the loop which requires O(1) time.
So, the total running time of the algorithm is O(n^2).