# Ali Abdullah Ahmad
# 20031246
# Assignment 5
# CS 600

**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.**

The Huffman coding algorithm generates an optimal prefix code based on character frequencies, ensuring that the most frequent characters get shorter codes. Fred's claimed code assignments:

- A → 0
- C → 10
- G → 111
- T → 110

**Step 1: Checking the Prefix Property**

The given codes follow the **prefix property**, meaning no code is a prefix of another. This is a necessary condition for a valid Huffman code.

**Step 2: Constructing a Huffman Tree**

Huffman coding follows a **greedy** approach where the two least frequent characters are merged iteratively. To verify whether Fred's encoding is valid, let's reconstruct the tree:

1. **Step 1: Smallest frequencies merge**
    - The longest codes (111 and 110) suggest that G and T should have the lowest frequencies since Huffman merges the least frequent symbols first.
    - Suppose G and T are combined first into an internal node.
2. **Step 2: Next merge**
    - The next longest code is C → 10, meaning that C is slightly more frequent than G and T.
    - This suggests that the **GT node is merged with C next**.
3. **Step 3: Final merge with A**
    - The final step in Huffman coding would merge the A node with the (C + GT) node, where A is the most frequent character (since it gets the shortest code "0").

**Step 3: Assigning Example Frequencies**

Since A is the most frequent and G/T are the least frequent, we assign relative frequencies in a way that justifies this tree structure. Example frequencies satisfying this hierarchy could be:

- **A = 40**
- **C = 25**
- **G = 15**
- **T = 20**

These frequencies ensure that the merging process follows the Huffman algorithm's rules and produces the given codes.

**Conclusion**

Thus, yes, these codewords could have been produced by the Huffman algorithm, provided that the character frequencies followed a hierarchy like the example above.

**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.**

**Denominations: {1, 3, 4}**
**Target amount: 6**
**Greedy approach:**

- Pick the largest coin $\leq 6 \rightarrow$ **4** (1 coin)
- Remaining: 6 - 4 = **2**
- Pick the largest coin $\leq 2 \rightarrow$ **1** (1 coin)
- Remaining: 2 - 1 = **1**
- Pick **1** (1 coin)

**Total coins used by greedy: 3** (4, 1, 1)
**Optimal solution:**

- Pick **3** (1 coin)
- Pick **3** (1 coin)

**Total coins used optimally: 2** (3, 3)
Thus, the greedy algorithm does **not** always give the minimum number of coins.

**Q3. No. 10.5.27**

EXERCISE | 10.5.27

(a) 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 $X = x_1x_2 \ldots x_n$, we say that a string $Y = y_1y_2 \ldots y_m$ is a ***subsequence*** of $X$ if there are a set of indices $\{i_1, i_2, \ldots, i_k\}$, such that $y_1 = x_{i_1}, y_2 = x_{i_2}, \ldots, y_k = x_{i_k}$, 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.**

We use a greedy approach to determine if a string L (length m) is a subsequence of a string K (length n) in O(n + m) time.

Algorithm:

1. Start with the first character of L and scan K from left to right to find a match.
2. Once a match is found, move to the next character in L while continuing to scan K forward.
3. Repeat this process until all characters in L are found in order within K, or K is fully traversed.
4. If all characters of L are matched, return true; otherwise, return false.

Time Complexity Analysis:

- Each character in K is scanned at most once → O(n).
- Each character in L is scanned at most once → O(m).
- Total comparisons = O(n + m).

Thus, this algorithm efficiently determines whether L is a subsequence of K in linear time.

**Q4. No. 11..6.1**

EXERCISE | 11.6.1

Characterize each of the following recurrence equations using the master theorem (assuming that $T(n) = k$ 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.**

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 $\varepsilon > 0$

$= O(n^{1-\varepsilon})$ (case 1)

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

$=$ **$O(n)$**

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 $\varepsilon > 0$

$= O(n^{3-\varepsilon})$ (case 1)

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

$=$ **$O(n^3)$**

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

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

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

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

$T(n) = O(n^{\log_b a} \log^{k+1} n)$ (case 1)

$=$ **$O(n^4 (\log^5 n))$**

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 $\varepsilon > 0$

$= O(n^{1.77-\varepsilon})$ (case 1)

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

**= O( $n^{1.77}$)**

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 $\varepsilon = 1$

T(n) = **$O(n^3 \log n)$** (case 3)

**Q5. No. 11.6.10**
**Consider the Stooge-sort algorithm, and suppose we change the assignment statement for m(on line 6) to the following:**
**m←max1,n/4**
**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.**

We know that the standard recurrence relation for stooge sorting algorithm is given as
$T(n) = 3T(2N/3) + O(1)$

In this case if we set $m \leftarrow \max\{1, n / 4\}$

We get the relation as
$T(N) = 3T(3N/4) + O(1)$

Let us calculate the complexity using master's theorem
$a = 3$ $b = 4/3 = 1.25$ and $d = 0$

Since $d < \log_b a$
$T(n) = O(n^{(\log_b a)}) = O(n^{\log 1.333}) = O(n^{3.852})$

## Q6. No. 11.6.17

(a) 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.**

Algorithm (Sweep Line with Max-Heap)

1. Preprocess Events:
    - For each rectangle [ai, bi] with height hi, create two events:
        - Start event (ai, hi, "start")
        - End event (bi, hi, "end")
    - Sort events by x-coordinate:
        - If two events have the same x, process "start" events before "end" events (to avoid removing a height before considering a new one).
        - If two start or end events have the same x, sort by height in descending order for starts, and ascending order for ends.
2. Sweep Line Processing:
    - Maintain a max-heap (priority queue) to store active heights.
    - Iterate through sorted events:
        - If start event (x, h, "start"), insert h into the heap.
        - If end event (x, h, "end"), remove h from the heap.
        - After each update, check the current maximum height:
            - If the max height changes, add (x, new max height) to the skyline.
3. Output the Skyline:
    - The resulting skyline consists of all points where the height changes.

Time Complexity Analysis

- Sorting events: O(nlogn)

**Q7. No. 12.9.5**

(a) 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,

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

| B, W | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12, 4 | A | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| 10, 6 | B | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 12 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 | 22 |
| 8, 5 | C | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 20 | 22 | 22 | 22 | 22 | 22 | 30 | 30 | 30 | 30 |
| 11, 7 | D | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 20 | 22 | 23 | 23 | 23 | 23 | 30 | 31 | 33 | 33 |
| 14, 3 | E | 0 | 0 | 0 | 14 | 14 | 14 | 14 | 26 | 26 | 26 | 26 | 26 | 34 | 36 | 37 | 37 | 37 | 37 | 44 |
| 7,1 | F | 0 | 7 | 7 | 14 | 21 | 21 | 21 | 26 | 33 | 33 | 33 | 34 | 34 | 43 | 44 | 44 | 44 | 44 | 44 |
| 9,6 | G | 0 | 7 | 7 | 14 | 21 | 21 | 21 | 26 | 33 | 33 | 33 | 34 | 41 | 43 | 44 | 44 | 44 | 44 | 44 |

We see that the maximum benefit is 44.
We will get the optimal solution from **{(12,4), (10,6), (8,5), (14,3)} items {a,b,c,e}**
**Total weight = 18** ($\leq$ 18, so valid)

**Q8. No. 12.9.14**

**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:

1. Sort the observation requests by finish times.
    - Since start and finish times are integers in the range 1, $n^2$ ; use counting sort to achieve O(n) sorting instead of O(nlogn).
2. Compute the array B using dynamic programming:
    - Initialize B[0]=0
    - Iterate through the sorted observations and compute:
      B[i] = max(B[i−1], B[P[i]] +bi)
      where P[i] stores the index of the last non-overlapping observation before i.
    - This loop runs in O(n) time.

Time Complexity Analysis:

- Sorting (Counting Sort): O(n)
- Dynamic Programming Computation: O(n)
- Total Complexity: O(n)

**Q9. No. 12.9.30**

(a) 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.**

```
def longest_palindromic_subsequence(S):
    n = len(S)
    dp = [[0] * n for _ in range(n)]

    # Base case: Single character palindromes
    for i in range(n):
        dp[i][i] = 1

    # Fill DP table
    for L in range(2, n+1):  # Length of substring
        for i in range(n - L + 1):
            j = i + L - 1
            if S[i] == S[j]:
                dp[i][j] = dp[i+1][j-1] + 2
            else:
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])

    return dp[0][n-1]  # Length of LPS
```

The total running time of the algorithm is $O(n^2)$.