Homework 3:

Due: October 2nd, 2025 at 2:30p.m.

This homework must be typed in LaTeX and submitted via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before your write. Except where indicated, consider every problem as asking for a proof.

Problem 1.

1. Consider a partition of the vertices of a graph $G = (V, E)$ into two disjoint subsets U and W, $U \cup W = V$. Let e be an edge of minimum weight across the partition. Prove that there is a minimum spanning tree of G containing edge e.
2. Formulate and analyze a greedy algorithm for minimum cost spanning tree based on the above property.

Solution. 1. **Cut Property** Let (U, W) be a cut of G, and let $e = xy$ be a minimum-weight edge among all edges with one endpoint in U and the other in W. We prove that there exists an MST that contains e.

Proof. Take any minimum spanning tree T of G. If $e \in T$ we are done. Otherwise, adding e to T creates a unique cycle C (since T is a tree). This cycle must contain at least one other edge f crossing the same cut (U, W) (as x and y lie on different sides of the cut, any $x \to y$ path in T must cross the cut an odd number of times; in particular, at least once). Remove such an f from $T \cup \{e\}$ to obtain a new spanning tree $T' = T \cup \{e\} \setminus \{f\}$.

By the choice of e as a minimum-weight edge crossing the cut, $w(e) \leq w(f)$. Therefore

$$w(T') \;=\; w(T) + w(e) - w(f) \;\leq\; w(T).$$

Since T is minimum, $w(T') = w(T)$, so T' is also a minimum spanning tree and it contains e. □

(i) Kruskal's algorithm. Sort all edges by nondecreasing weight; scan in this order, adding an edge if and only if it does not create a cycle with the edges already chosen. Implement cycle detection with a disjoint-set union (Union–Find) data structure.

Algorithm 1 Kruskal($G = (V, E)$)

```
Initialize a forest F ← ∅; make-set(v) for all v ∈ V
Sort E as e_1, ..., e_m by nondecreasing weight
for j = 1 to m do
    let e_j = (u, v)
    if find(u) ≠ find(v) then                    ▷ adding e_j won't form a cycle
        F ← F ∪ {e_j}; union(u, v)
return F
```

Correctness sketch. At every step, consider the cut induced by the connected components (current forest) that *respects* the chosen edges. Among all edges crossing that cut, the first edge by weight is safe by the Cut Property; Kruskal picks exactly such safe edges, hence returns an MST.

Running time. Sorting takes $O(m \log m) = O(m \log n)$. Total $O(m \log n)$ time and $O(n)$ extra space.

□

Problem 2. Tianren *really* likes **reversible** words, i.e., words that read the same forwards and backwards (e.g., racecar, kayak, level).

1. Given a string with n lower-case English letters, design an efficient dynamic program that finds the longest substring (a consecutive sequence of characters) that is a *reversible word* (i.e., a palindrome).
2. Provide proof of correctness and analyze time and memory.

Solution. Let $s[0..n-1]$ be the string. Define a DP table

$$P[i,j] \;=\; \begin{cases} \text{true} & \text{if } s[i..j] \text{ is a palindrome,} \\ \text{false} & \text{otherwise,} \end{cases} \qquad 0 \le i \le j < n.$$

Recurrences (process by increasing length $\ell = j - i + 1$):

$$\begin{aligned} \text{Base:} \quad & P[i,i] = \text{true} \quad \text{(1-char)}, \\ & P[i,i+1] = (s[i] = s[i+1]) \quad \text{(2-char)}, \\ \text{Step:} \quad & P[i,j] = (s[i] = s[j]) \;\wedge\; P[i+1,j-1] \quad (\ell \ge 3). \end{aligned}$$

Track the longest $(i^{,j)}$ with $P[i^{,j]=\text{true}}$.

Algorithm 2 LongestPalSubstring(s)

```
   n ← |s|; initialize P[0..n−1][0..n−1] ← false
 2: bestLen ← 1, best(i, j) ← (0, 0)
 3: for i = 0 to n − 1 do P[i, i] ← true
 4: for i = 0 to n − 2 do
 5:     if s[i] = s[i + 1] then P[i, i + 1] ← true; bestLen← 2; best← (i, i + 1)
 6: for ℓ = 3 to n do
 7:     for i = 0 to n − ℓ do
 8:         j ← i + ℓ − 1
 9:         if s[i] = s[j] and P[i + 1][j − 1] then
10:             P[i, j] ← true
11:             if ℓ > bestLen then bestLen← ℓ; best← (i, j)
12: return s[best(i)..best(j)]
```

Correctness. We prove by induction on substring length ℓ that $P[i,j]$ is true iff $s[i..j]$ is a palindrome.

Base $\ell = 1, 2$. Trivial by definition. *Inductive step.* Assume correctness for all lengths $< \ell$. For length $\ell \ge 3$, $s[i..j]$ is a palindrome iff its endpoints match and the interior $s[i+1..j-1]$ is a palindrome. By the IH, $P[i+1, j-1]$ correctly captures palindromicity of the interior; thus the recurrence is correct.

Since we enumerate all (i,j) in increasing ℓ and maintain the best true entry, the returned substring is the longest palindrome.

Complexity. The DP fills $\Theta(n^2)$ entries; each in $O(1)$ time. Hence time $O(n^2)$, space $O(n^2)$. (If desired, a non-DP "expand around centers" method yields $O(n^2)$ time and $O(1)$ extra space; Manacher's algorithm achieves $O(n)$ time.)

□

Problem 3. Design a Huffman code for this paragraph. What is the number of bits required to encode this paragraph using the Huffman code vs. the standard ASCII code (8 bits per character)?

1. *Solution.* Let the paragraph contain characters from an alphabet Σ. For each $c \in \Sigma$, compute its frequency $f(c)$ and probability $p(c) = f(c)/N$, where $N = \sum_c f(c)$ is the total number of characters (including spaces and punctuation, if specified).

   Constructing the Huffman code.

   1. Build a min-heap keyed by $f(c)$ for all $c \in \Sigma$.
   2. While the heap has at least two nodes, extract the two minimum-frequency nodes x, y, create a new node z with weight $f(z) = f(x) + f(y)$ and children x, y, and insert z back into the heap.
   3. The final node is the root of the Huffman tree. Assign binary codewords by labeling one child edge 0 and the other 1 on every branch; a character's code is the bitstring from root to its leaf.

   Bit counts. Let $\ell(c)$ be the codeword length of c. Then

   $$\text{Bits}_{\text{Huffman}} = \sum_{c \in \Sigma} f(c) \cdot \ell(c), \qquad \text{Bits}_{\text{ASCII}} = 8N.$$

   Average code length is $\bar{\ell} = \sum_c p(c)\,\ell(c)$, so $\text{Bits}_{\text{Huffman}} = N \cdot \bar{\ell}$.
   Running time. $O(|\Sigma| \log |\Sigma|)$ to build the heap plus $O(|\Sigma| \log |\Sigma|)$ merges; overall $O(|\Sigma| \log |\Sigma|)$. Counting frequencies takes $O(N)$. **Step 1: Frequency count.**

   The paragraph contains $N = 177$ characters (including spaces and punctuation). The alphabet size is 33 distinct symbols.

   Step 2: ASCII encoding size. In standard ASCII, each character takes 8 bits:

   $$\text{Bits}_{\text{ASCII}} = 177 \times 8 = 1416 \text{ bits.}$$

   Step 3: Huffman coding. Constructing the Huffman tree from the frequency distribution yields code lengths $\ell(c)$ for each character c. The total number of bits is

   $$\text{Bits}_{\text{Huffman}} = \sum_c f(c)\,\ell(c).$$

   For this paragraph, the result is:

   $$\text{Bits}_{\text{Huffman}} = 777 \text{ bits.}$$

   Step 4: Comparison.

   $$\text{ASCII bits} = 1416, \qquad \text{Huffman bits} = 777.$$

   Thus, the Huffman encoding uses only about 54.8% of the space required by ASCII.

   Conclusion. Encoding this paragraph with a Huffman code nearly halves the storage cost compared to fixed-length 8-bit ASCII. □