Question 1. The result of the Huffman tree algorithm is not unique. Please add some constraints to let the result unique.

We can make the Huffman tree deterministic by applying a **tie-breaking rule**.

Tie-breaker Strategy: Use Insertion Order

We modify the comparison logic so that if two nodes have the same frequency, the one that was inserted **earlier** has higher priority.

Implementation Details

1. Modify the Node Structure

```
struct Node {
  Node *e0, *e1;
  int freq;
  char ch = -1;
  int id; // Unique insertion ID for tie-breaking
};
```

2. Update the Comparator

```
struct node_comp {
  bool operator()(const Node* a, const Node* b) const {
    if (a->freq == b->freq)
      return a->id > b->id; // Smaller ID means higher priority
    return a->freq > b->freq;
  }
};
```

3. Assign Unique IDs When Creating Nodes

```
int uid = 0;
for (auto& [ch, freq] : ch_count) {
  trees.push(new Node{nullptr, nullptr, freq, ch, uid++});
}

// For internal nodes:
trees.push(new Node{t0, t1, t0->freq + t1->freq, -1, uid++});
```

With this modification:

- The **selection order** of nodes becomes deterministic
- The **structure of the Huffman tree is unique**

- The **binary codes** assigned to characters are consistent across runs

Question 2. std::map is Red-Black-Tree. std::priority_queue is heap. Assume length of `text` is n, which contains m kinds of characters. What is the time complexity of `HuffmanCode(text)`?

Assume:

- `n` = length of input string `text`
- `m` = number of **distinct characters** in `text`

We analyze the time complexity of the constructor `HuffmanCode(text)` step by step:

---

1. Frequency Counting with `std::map`

```
for (char ch : text) {
  ch_count[ch]++;
}
```

- `std::map` is implemented as a Red-Black Tree.
- Each insertion or lookup takes **O(log m)**.
- Loop runs `n` times → **O(n log m)**.

---

2. Pushing Nodes into `std::priority_queue`

```
for (auto node : ch_count) {
  trees.push(new Node{...});
}
```

- Each `push` is **O(log m)** in a binary heap.
- `m` distinct characters → **O(m log m)**.

---

3. Building the Huffman Tree

```
while (trees.size() > 1) {
  Node* t0 = trees.top(); trees.pop();
  Node* t1 = trees.top(); trees.pop();
  trees.push(new Node{t0, t1, t0->freq + t1->freq, -1, uid++});
}
```

- Loop runs `m - 1` times (creating `m - 1` internal nodes).
- Each pop and push operation is **O(log m)**.
- Total → **O(m log m)**.

---

4. DFS Tree Traversal to Generate Code

```
travel(root, "");
```

- Visits all `2m - 1` nodes → **O(m)**.

Total Time Complexity:

```
O(n log m + m log m)
```

This complexity is efficient for practical use, especially when `m` is small (e.g., 256 ASCII characters).

Question 3. Please modify above algorithm to choose the minimum number of activities to cover the whole time range.

```cpp
// Returns the minimum number of activities required to cover the interval
// [S, T)
int min_cover(std::vector<Activity> A, int S, int T) {
  // Sort activities by start time in ascending order
  std::sort(A.begin(), A.end(),
            [](const Activity &a, const Activity &b) { return a.s < b.s; });

  int count = 0;    // Number of activities selected
  int now = S;      // Current time point we need to cover
  int i = 0;        // Index to scan through the activity list
  int n = A.size(); // Total number of activities

  // Keep selecting activities until we cover the interval [S, T)
  while (now < T) {
    int max_end = -1; // Farthest reachable end time from current point

    // Iterate over all activities that start at or before 'now'
    while (i < n && A[i].s <= now) {
      max_end = std::max(max_end, A[i].e); // Pick the one that ends the latest
      i++;
    }

    // If no activity can extend the coverage, return -1 (impossible)
```

```
    if (max_end <= now)
      return -1;

    // Move the current point forward to the end of the selected activity
    now = max_end;
    count++; // One more activity selected
  }

  return count; // Return the minimum number of activities used
}
```

time complexity:

- sort -> $O(nlogn)$
- scan one time -> $O(n)$
- total time complexity -> $O(nlogn)$

Question 4. Activities with overlapping in time cannot be in the same room. Please design an algorithm to find how many rooms we need to hold all the given activities.

> Hint: Sort $s(A) \cup e(A)$ rather than sort A

```
// Return the minimum number of rooms needed to schedule all activities
int min_rooms(std::vector<Activity> A) {
  std::vector<std::pair<int, int>> events;

  for (auto &act : A) {
    events.emplace_back(act.s, +1); // start event
    events.emplace_back(act.e, -1); // end event
  }

  // Sort by time; if equal, end (-1) before start (+1)
  std::sort(events.begin(), events.end(), [](auto &a, auto &b) {
    return a.first == b.first ? a.second < b.second : a.first < b.first;
  });

  int ongoing = 0;   // current number of active rooms
  int max_rooms = 0; // maximum rooms used at any time

  for (auto &[time, delta] : events) {
    ongoing += delta;
    max_rooms = std::max(max_rooms, ongoing);
  }

  return max_rooms;
}
```

Question 5. We know that the greedy approach cannot always guarantee the optimal answer. Please discuss when we have multiple items with the same value/weight ratio, should we prioritize items with smaller weights or larger weights?

In the classic **Fractional Knapsack Problem**, the greedy algorithm chooses items based on **value-to-weight ratio** in decreasing order. But when multiple items have the **same value/weight ratio**, the greedy algorithm's decision is **no longer unique** — and it can impact performance depending on what we prioritize.

Which should we prioritize: smaller weights or larger weights?

Answer: Prioritize smaller weights

Reason:

- Smaller items with the same value/weight ratio allow **finer granularity**.
- We can fill the knapsack more precisely without overshooting the capacity.
- Choosing smaller weights increases the **flexibility** of the remaining space.

Example:

Suppose we have a capacity of 50, and two items:

- Item A: weight = 30, value = 60 (ratio = 2)
- Item B: weight = 10, value = 20 (ratio = 2)

If you choose A first, you'll use up 30 units of capacity. But if B is chosen first, you still have room to consider more options (e.g., five B's total if available).

Prioritizing larger weights first:

- May lead to **early saturation** of the knapsack
- Reduces the chance to include combinations that use space more efficiently

Conclusion:

> When items have equal value/weight ratios, **prioritizing smaller weights leads to better flexibility and often better final value**, especially in fractional settings.