Problem 1 Solution

(a) Adapting Merge Sort for m-Tuples

The standard merge sort algorithm works on a list of numbers by recursively splitting the list into two halves, sorting each half, and then merging those halves together. To adapt this for m-tuples, we really only need to change **how** we compare two elements—because in this case, each "element" is now an m-tuple rather than a single number.

1. **Define a Lexicographic Comparison**

   Suppose we have two m-tuples:

   $$A = (a_1, a_2, \ldots, a_m) \quad \text{and} \quad B = (b_1, b_2, \ldots, b_m).$$

   We say $A < B$ in *lexicographic order* if, when we look at the first position where they differ (say at index j), we have $a_j < b_j$. If all components are the same, then $A = B$. This is exactly how we compare words in a dictionary: we compare letter by letter until we find a difference.

2. **Plug This Comparison into Merge Sort**

   The actual merge sort steps—divide, recursively sort, and merge—remain the same. Wherever the algorithm compares two "elements," we just use our new lexicographic comparison on the m-tuples.

   - **Divide**: Split the list of m-tuples into two sublists.
   - **Conquer**: Recursively sort each sublist (using the same comparison).
   - **Merge**: While merging, whenever we decide which tuple goes next, we compare the two "front" tuples from each sorted sublist using the lexicographic rule.

   We take the original merge sort structure and replace the numerical comparison with our custom tuple comparison.

(b) Worst-Case Time Complexity

1. **Number of Comparisons**

   In the worst case, merge sort does $O(n \log n)$ comparisons to sort n elements. Now, each element is an m-tuple.

2. **Cost of Each Comparison**

   In the worst case, comparing two m-tuples lexicographically could require looking at all m positions (imagine they match on the first $m-1$ components and differ only on the last one). Since each component-wise comparison is given as $O(1)$ in the worst case, one tuple-to-tuple comparison is $O(m)$.

3. **Putting It Together**

   Therefore, the total worst-case time is

   $$O(\#\text{ comparisons}) \times O(\text{cost per comparison}) = O(n \log n) \times O(m) = O(m \cdot n \log n).$$

   If m is a fixed constant (for example, if you always have 5-tuples), we often just write $O(n \log n)$. But in the more general sense, the complexity is $O(m \cdot n \log n)$.

Problem 2 Solution

(a) Sorting the Five Toys with Modified Merge Sort

Let's say each toy has attributes in some fixed order, for example:

$$(\text{Character, Color, Size, Shape, Texture}).$$

Alice's Priorities

Alice might care about the attributes in this order (from most important to least important):

1. **Color**
2. **Character**
3. **Size**
4. **Shape**
5. **Texture**

To sort by these priorities using the modified merge sort, we:

1. **Rewrite each toy** as a tuple in that exact order. For instance, if a toy's original description was (bird, blue, small, round, soft), we reorder it to (blue, bird, small, round, soft).
2. **Apply the lexicographic merge sort** exactly as in Problem 1, comparing first by color, then by character if colors match, then by size, and so on.

Bob's Priorities

Bob might rank the attributes differently. For example, maybe he cares about:

1. **Texture**
2. **Character**
3. **Color**
4. **Size**
5. **Shape**

Then we:

1. **Rewrite each toy** as (Texture, Character, Color, Size, Shape).
2. **Run the same modified merge sort**, but now comparing toys by texture first, then character if textures match, and so on.

(b) Using Radix Sort Instead

The **radix sort** can sort lists in "lexicographic order" if we do stable sorting passes from the least significant attribute to the most significant attribute. For example:

- **For Alice** (color , character , size , shape , texture), we:
  1. Stable sort by texture (least important).
  2. Stable sort by shape.
  3. Stable sort by size.
  4. Stable sort by character.
  5. Finally, stable sort by color (most important).
- **For Bob** (texture , character , color , size , shape), we:
  1. Stable sort by shape.
  2. Stable sort by size.
  3. Stable sort by color.
  4. Stable sort by character.
  5. Finally, stable sort by texture.

Since each pass is stable, the final ordering matches exactly what we would get by doing a single lexicographic comparison. The radix sort algorithm itself remains unchanged.