What your program does (high level)

You have two layers:

1. **Permutation layer (build all orderings of p)**

- You compute how many permutations should exist (permutations as a BigInteger).
- You then **randomly generate permutations** until your Set<String> st reaches that count.
- Result: valuesSet[] = all permutations (unique strings).

2. **Scan layer (for each permutation, scan s)**

- For each permutation valuesSet[entry], you scan s starting from each startPos (counter).
- You match characters **in the exact order of the permutation**.
- You allow "skips" in s (you move i forward and keep searching).
- Your *subsequence mode* is **PREFIX**:
    - you only start "counting" once you hit the **first matched character** (your hasFirstMatch gate),
    - and you only match from the beginning of the permutation (you don't delete from the front of Y unless using Strategy A).

When a scan attempt finishes (either perfect match or partial), you store:

- indices of the window in s (firstIndexLocationStringS, lastIndexLocationStringS),
- the permutation,
- and the subsequence details (what matched, what didn't, lengths, etc).

Your "mental model" (how you seem to think while coding this)

Your code reflects a consistent mindset:

- **Accountability first**: you want to *prove* you explored all permutations (set size equals the theoretical permutation count), not "probably enough".
- **State machine thinking**: you treat matching like a "consume from template" process:
    - sb is the "remaining work to do".
    - When a char matches, you delete it from sb.
    - When sb is empty → the permutation is fully satisfied.
- **Window + evidence logging**: you don't just want "true/false".
    - You want the substring window, the subsequence extracted, and indexes in both strings.
- **Performance guardrails**: you added cycle counters and set-size checkpoints, and you avoid printing too frequently.

That's why your design works well as a "debuggable engine" rather than a minimal solution.

Key variables and what they mean (Main class)

Global (static) fields

- **permutations (BigInteger)**
  Target number of unique permutations you expect for p (factorials + repetition adjustment in your Permutation helper).

- **row**
  Output record pointer. Every time you finish storing a scan result, you do row++.

- **col**
  Always used as the base column index (effectively 0). You use col, col+1, col+2 to keep the code readable.

- **firstIndexLocationStringS, lastIndexLocationStringS**
  The window bounds in s for the current attempt.
    - firstIndex... = where the *first matched character* occurred
    - lastIndex... = where the *last matched character* occurred

- **store[1000][5]**
  A table of "core results" per row. In this final version you mainly store:
    - store[row][0] = first index in s
    - store[row][1] = last index in s
    - store[row][2] = permutation string (the template that was attempted)
      (cols 3–4 exist but aren't heavily used in this file)

- **subsequences[1000][6] (StringJoiner[][])**
  Per-row subsequence details. Your 6 columns behave like:

1. **Matched chars in X with X-index**

Example: a index(2) b index(3)

2. **Unmatched chars in X with X-index** *(only after first match, because of hasFirstMatch gate)*

Example: c index(1) as unmatched

3. **Presented window string (X-window)**

You build up the window content you "saw" (characters you processed once matching started). This is what you print as "Presented window".

4. **Length of the subsequence**

Stored as a number string (you later parse/use it to compute maximum length).

5. **Subsequence characters only (no indices)**

Example: ab

6. **Matched characters with Y-indices**

Example: a index: (0) b index: (1)

- **maximumSubsequenceLength**
  You compute the maximum value of subsequences[c][3] across rows, then print ties.

- **windowSize**
  Used for printing (size: ...) and window summaries.

Key local variables inside findPermutations(s,p)

Permutation building stage

- **stRandomNums**
  Ensures you don't reuse the same random index twice while building a single permutation.

- **st**
  The set of unique permutation strings built so far.

- **sj**
  Builds one permutation candidate (characters of p in random order) before inserting into st.

- **cycles, totalcycles**
  How many attempts you made to populate the permutation set.

- **valuesSet[]**
  Final array of permutations from the set.

Scan/match stage

- **sb (StringBuilder)**
  The "remaining letters of current permutation that still need to be matched".
    - When you match the next needed character, you delete it from sb.
    - Empty sb means perfect match.
- **counter**
  Drives different startPos positions (try matching starting from s[0], then s[1], ...).
- **startPos**
  Current starting index in s for the scan attempt.
- **i**
  The running index scanning through s.
- **hasCharFound**
  Flag used to indicate you matched at least something during this attempt (used in your result classification).
- **hasFirstMatch** *(important for your PREFIX subsequence semantics)*
  This is your "gate".
    - Before the first match, you don't treat any characters as part of the window/unmatched set.
    - After the first match, you start recording the window and unmatched characters.
- **lengthSubsequence**
  How many characters you successfully matched (in order) for that attempt.

What your PREFIX subsequence rule really means

Your design is effectively:

"For each permutation Y, I will only start recording the subsequence/window **after the first matched character of Y is found in X**. I do **not** try to rescue the permutation by skipping its leading characters (unless using Strategy A)."

So for p="ba", s="a":

- permutation "ba" → no first match (b) → subsequence length 0
- permutation "ab" → first match found (a) → subsequence length 1

That's why permutations can "rescue" you even in prefix mode.