Cheat Sheet String

Topics: (String Basics, Hashing, Frequency Arrays; Palindromic Substrings, Longest Palindrome; Anagram Grouping, Sliding Window on Strings; Manacher's Algorithm, Z-algorithm (Pattern Matching)).

1. String Basics, Hashing, Frequency Arrays

Java Essentials

```
String s = "example";
s.length();              // String length
s.charAt(0);             // Character at index
s.substring(1, 4);       // Substring
s.indexOf("a");          // First occurrence
s.equals("text");        // Exact comparison
s.equalsIgnoreCase("TEXT");
s.contains("amp");
s.replace("e", "E");     // Replace character
s.split("e");            // Split by delimiter

StringBuilder sb = new StringBuilder(s);
sb.reverse().toString();  // Efficient reversal
```

Explanation: These are common operations on strings. StringBuilder is used when frequent modification is needed due to its mutable nature.

Frequency Array Template

```
int[] freq = new int[26]; // For lowercase letters
for (char c : s.toCharArray()) {
    freq[c - 'a']++;
}
```

Explanation: A basic frequency counter assuming all lowercase letters. Useful in problems involving anagrams or character counts.

HashMap Character Frequency

```
Map<Character, Integer> map = new HashMap<>();
for (char c : s.toCharArray()) {
    map.put(c, map.getOrDefault(c, 0) + 1);
}
```

Explanation: More flexible than frequency arrays. Works for any character range.

2. Palindromic Substrings, Longest Palindrome

Count Palindromic Substrings (LC 647)

```
int count = 0;
for (int i = 0; i < s.length(); i++) {
    count += expandAroundCenter(s, i, i);     // Odd
    count += expandAroundCenter(s, i, i + 1); // Even
}

int expandAroundCenter(String s, int l, int r) {
    int count = 0;
    while (l >= 0 && r < s.length() && s.charAt(l--) == s.charAt(r++)) {
        count++;
    }
    return count;
}
```

Explanation: This technique expands outward from every character (and pair) to find palindromes. Handles both odd and even length cases.

Longest Palindromic Substring (LC 5)

```
String longest = "";
for (int i = 0; i < s.length(); i++) {
    String odd = expand(s, i, i);
    String even = expand(s, i, i + 1);
    if (odd.length() > longest.length()) longest = odd;
    if (even.length() > longest.length()) longest = even;
}

String expand(String s, int l, int r) {
    while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
        l--; r++;
    }
    return s.substring(l + 1, r);
}
```

Explanation: Similar to above, but returns the actual longest substring. Uses a helper method to expand from the center.

3. Anagram Grouping, Sliding Window on Strings

Group Anagrams (LC 49)

```
Map<String, List<String>> map = new HashMap<>();
for (String word : words) {
    char[] chars = word.toCharArray();
```

```
    Arrays.sort(chars);
    String key = new String(chars);
    map.computeIfAbsent(key, k -> new ArrayList<>()).add(word);
}
```

Explanation: Anagrams share the same sorted form. Sort each string and use as a key to group anagrams.

Sliding Window: Longest Substring w/o Repeating (LC 3)

```
Map<Character, Integer> map = new HashMap<>();
int maxLen = 0, left = 0;

for (int right = 0; right < s.length(); right++) {
    if (map.containsKey(s.charAt(right))) {
        left = Math.max(left, map.get(s.charAt(right)) + 1);
    }
    map.put(s.charAt(right), right);
    maxLen = Math.max(maxLen, right - left + 1);
}
```

Explanation: Maintain a sliding window and a hashmap of seen characters to ensure no repeats. Adjust the window as needed.

Minimum Window Substring (LC 76)

```
Map<Character, Integer> tMap = new HashMap<>();
for (char c : t.toCharArray()) tMap.put(c, tMap.getOrDefault(c, 0) + 1);

Map<Character, Integer> window = new HashMap<>();
int left = 0, minLen = Integer.MAX_VALUE, start = 0, matched = 0;

for (int right = 0; right < s.length(); right++) {
    char c = s.charAt(right);
    window.put(c, window.getOrDefault(c, 0) + 1);

    if (tMap.containsKey(c) && window.get(c).intValue() == tMap.get(c).intValue()) {
        matched++;
    }

    while (matched == tMap.size()) {
        if (right - left + 1 < minLen) {
            minLen = right - left + 1;
            start = left;
        }
        char leftChar = s.charAt(left);
        if (tMap.containsKey(leftChar)) {
            if (window.get(leftChar).intValue() == tMap.get(leftChar).intValue()) {
                matched--;
```

```
        }
      }
      window.put(leftChar, window.get(leftChar) - 1);
      left++;
    }
}
String result = minLen == Integer.MAX_VALUE ? "" : s.substring(start, start + minLen);
```

Explanation: Use two maps to track the required and current window characters. Slide the window until all requirements are matched.

4. Manacher's Algorithm, Z-Algorithm (Pattern Matching)

Manacher's Algorithm (Longest Palindrome in O(n))

```
String preprocess(String s) {
   StringBuilder sb = new StringBuilder("^");
   for (char c : s.toCharArray()) {
      sb.append("#").append(c);
   }
   sb.append("#$");
   return sb.toString();
}

int manacher(String s) {
   String T = preprocess(s);
   int[] P = new int[T.length()];
   int C = 0, R = 0;

   for (int i = 1; i < T.length() - 1; i++) {
      int iMirror = 2 * C - i;
      if (R > i)
         P[i] = Math.min(R - i, P[iMirror]);

      while (T.charAt(i + 1 + P[i]) == T.charAt(i - 1 - P[i]))
         P[i]++;

      if (i + P[i] > R) {
         C = i;
         R = i + P[i];
      }
   }

   int maxLen = 0;
   for (int p : P) maxLen = Math.max(maxLen, p);
   return maxLen;
}
```

Explanation: Preprocess string with separators to handle odd/even palindromes. Efficient O(n) solution using symmetry.

Z-Algorithm (Pattern Matching)

```
int[] zAlgorithm(String s) {
    int n = s.length();
    int[] z = new int[n];
    int l = 0, r = 0;

    for (int i = 1; i < n; i++) {
        if (i <= r)
            z[i] = Math.min(r - i + 1, z[i - l]);

        while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i]))
            z[i]++;

        if (i + z[i] - 1 > r) {
            l = i;
            r = i + z[i] - 1;
        }
    }
    return z;
}
```

Explanation: Computes length of longest substring starting at each index that matches prefix. Useful for pattern matching.

Quick Decision Table

Problem Type	Best Technique	Time
Unique Characters Substring	Sliding Window	O(n)
Count All Palindromes	Expand Around Center	$O(n^2)$
Longest Palindromic Substring	Manacher's	O(n)
Group Anagrams	HashMap + Sort	O(n·k log k)
Pattern Search	Z / KMP / Rabin-Karp	O(n + m)