Regex & Text Manipulation Cheatsheet

A comprehensive guide to regular expressions and text manipulation techniques, essential for algorithms and interview preparation.

Regex Fundamentals

Basic Metacharacters

`.` (Dot)	Matches any single character except newline. **Example:** `a.c` matches “abc”, “aac”, “adc”, etc.
`^` (Caret)	Matches the beginning of the string. **Example:** `^abc` matches “abc” only if it’s at the beginning.
`$` (Dollar)	Matches the end of the string. **Example:** `xyz$` matches “xyz” only if it’s at the end.
`*` (Asterisk)	Matches 0 or more occurrences of the preceding character or group. **Example:** `ab*c` matches “ac”, “abc”, “abbc”, “abbbc”, etc.
`+` (Plus)	Matches 1 or more occurrences of the preceding character or group. **Example:** `ab+c` matches “abc”, “abbc”, “abbbc”, etc., but not “ac”.
`?` (Question Mark)	Matches 0 or 1 occurrence of the preceding character or group. **Example:** `ab?c` matches “ac” or “abc”.
`[]` (Character Set)	Matches any single character within the set. **Example:** `[aeiou]` matches any vowel.
`[^]` (Negated Character Set)	Matches any single character *not* within the set. **Example:** `[^aeiou]` matches any character that is not a vowel.
`|` (Pipe)	Acts as an “OR” operator, matching either the expression before or after the pipe. **Example:** `cat|dog` matches either “cat” or “dog”.

Quantifiers and Grouping

`{n}`	Matches exactly `n` occurrences. **Example:** `a{3}` matches “aaa”.
`{n,}`	Matches `n` or more occurrences. **Example:** `a{2,}` matches “aa”, “aaa”, “aaaa”, etc.
`{n,m}`	Matches between `n` and `m` occurrences. **Example:** `a{2,4}` matches “aa”, “aaa”, or “aaaa”.
`( )` (Grouping)	Groups patterns together, allowing you to apply quantifiers or other operations to the entire group. **Example:** `(ab)+` matches “ab”, “abab”, “abababa”, etc.
`\` (Escape)	Escapes special characters, allowing you to match them literally. **Example:** `\*` matches a literal asterisk.

Character Classes

`\d`	Matches any digit (0-9). **Example:** `\d+` matches one or more digits.
`\w`	Matches any word character (letters, digits, and underscores). **Example:** `\w+` matches one or more word characters.
`\s`	Matches any whitespace character (space, tab, newline, etc.). **Example:** `\s+` matches one or more whitespace characters.
`\D`	Matches any non-digit character. **Example:** `\D+` matches one or more non-digit characters.
`\W`	Matches any non-word character. **Example:** `\W+` matches one or more non-word characters.
`\S`	Matches any non-whitespace character. **Example:** `\S+` matches one or more non-whitespace characters.

Advanced Regex Concepts

Lookarounds (Zero-Width Assertions)

`(?=pattern)` (Positive Lookahead)	Asserts that the pattern is followed by the specified `pattern`, but doesn't include the `pattern` in the match. **Example:** `\w+(?=\s)` matches a word followed by a space, but the space isn't part of the match.
`?!pattern` (Negative Lookahead)	Asserts that the pattern is *not* followed by the specified `pattern`. **Example:** `\w+(?!\s)` matches a word not followed by a space.
`(?<=pattern)` (Positive Lookbehind)	Asserts that the pattern is preceded by the specified `pattern`, but doesn't include the `pattern` in the match. Requires fixed width pattern in some languages. **Example:** `(?<=\s)\w+` matches a word preceded by a space, but the space isn't part of the match.
`?<!pattern` (Negative Lookbehind)	Asserts that the pattern is *not* preceded by the specified `pattern`. Requires fixed width pattern in some languages. **Example:** `(?<!\s)\w+` matches a word not preceded by a space.

Backreferences

`\1`, `\2`, etc.	Refers to the captured group with the corresponding number. Useful for matching repeated patterns. **Example:** `(.)\1+` matches two or more consecutive identical characters.

Flags/Modifiers

`i` (Case-insensitive)	Makes the regex case-insensitive. **Example:** `/abc/i` matches "abc", "Abc", "ABC", etc.
`g` (Global)	Finds all matches rather than stopping after the first. **Example:** `/abc/g` finds all occurrences of "abc" in a string.
`m` (Multiline)	Treats the string as multiple lines, allowing `^` and `$` to match the start and end of each line. **Example:** `/^abc$/m` matches "abc" at the beginning of any line.
`s` (Dotall)	Allows the `.` to match newline characters as well. **Example:** `/a.c/s` matches "a\nc".

Text Manipulation Techniques

String Splitting

Splitting by a delimiter	Use the `split()` method (or equivalent) to divide a string into an array based on a delimiter. **Example (Python):** ``` text = "apple,banana,orange" result = text.split(",") # Output: ['apple', 'banana', 'orange'] ```
Splitting by Regex	Use regex for more complex splitting scenarios. **Example (JavaScript):** ``` const text = "one two three four"; const result = text.split(/\s+/); // Split by one or more spaces // Output: ['one', 'two', 'three', 'four'] ```

String Replacement

Basic Replacement	Replace a substring with another string. **Example (Java):** `String text = "Hello World"; String result = text.replace("World", "Java"); // Output: Hello Java`
Regex Replacement	Use regex for more powerful replacement operations. **Example (C#):** `using System.Text.RegularExpressions;` `string text = "123-456-7890"; string result = Regex.Replace(text, "[\\d-]", "X"); // Output: XXX-XXX-XXXX`

Substring Extraction

Using indices	Extract a portion of a string using start and end indices. **Example (C++):** `#include <iostream> #include <string>` `int main() { std::string text = "Hello World"; std::string result = text.substr(6, 5); // Start at index 6, length 5 std::cout << result << std::endl; // Output: World return 0; }`
Regex-based extraction	Use regex groups to extract specific parts of a string. **Example (Ruby):** `text = "My phone number is 123-456-7890" match = text.match(/.*(\d{3}-\d{3}-\d{4})/) #Capture group if match puts match[1] # Output: 123-456-7890 end`

Regex & Text Manipulation in Algorithms

Palindrome Check

Use regex to preprocess the string by removing non-alphanumeric characters and converting to lowercase. Then, compare the string to its reverse.

Example (Python):

```
import re

def is_palindrome(s):
  processed_string = re.sub(r'[^a-zA-Z0-9]', '', s).lower()
  return processed_string == processed_string[::-1]

print(is_palindrome("A man, a plan, a canal: Panama")) # Output: True
```

Validating User Input

Regex is excellent for validating formats such as email addresses, phone numbers, or passwords.

Example (JavaScript):

```
function isValidEmail(email) {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return emailRegex.test(email);
}

console.log(isValidEmail("test@example.com")); // Output: true
console.log(isValidEmail("invalid-email")); // Output: false
```

Parsing Log Files

Regex can be used to extract relevant information from log files.

Example (Python):

```
import re

log_line = "2023-10-26 10:00:00 INFO: User logged in"
match = re.search(r'INFO: (.*)$', log_line)
if match:
  print(match.group(1))  # Output: User logged in
```

String Compression/Decompression

Text manipulation techniques can be used in string compression and decompression algorithms, such as Run-Length Encoding (RLE).

Example (Python):

```
def compress_string(s):
    compressed = ''
    count = 1
    for i in range(len(s)):
        if i + 1 < len(s) and s[i] ==
s[i + 1]:
            count += 1
        else:
            compressed += s[i] +
str(count)
            count = 1
    return compressed

print(compress_string("AAABCCDAA")) #
Output: A3B1C2D1A2
```