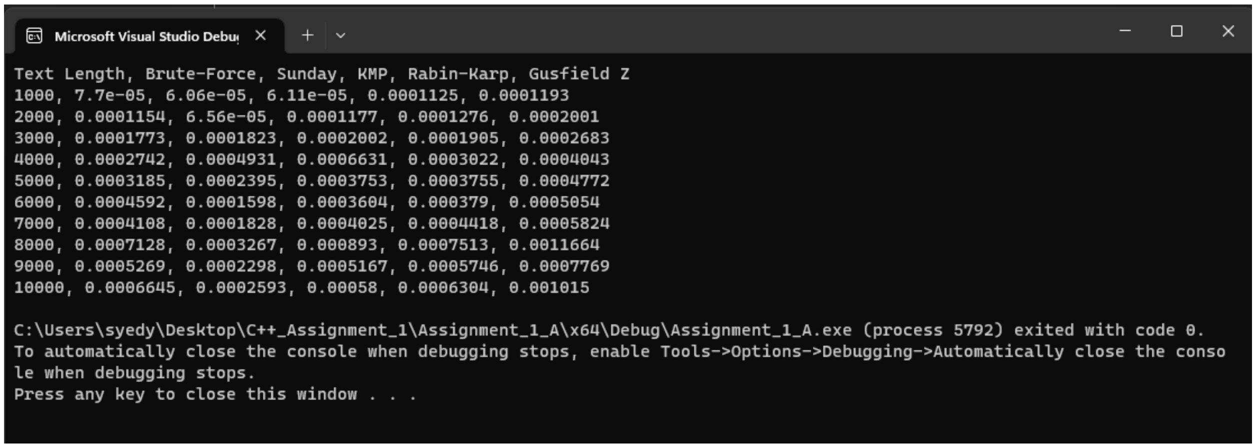
Contents

Assignment 1

Part One A:

OUTPUT:



```
Text Length, Brute-Force, Sunday, KMP, Rabin-Karp, Gusfield Z
1000, 7.7e-05, 6.06e-05, 6.11e-05, 0.0001125, 0.0001193
2000, 0.0001154, 6.56e-05, 0.0001177, 0.0001276, 0.0002001
3000, 0.0001773, 0.0001823, 0.0002002, 0.0001905, 0.0002683
4000, 0.0002742, 0.0004931, 0.0006631, 0.0003022, 0.0004043
5000, 0.0003185, 0.0002395, 0.0003753, 0.0003755, 0.0004772
6000, 0.0004592, 0.0001598, 0.0003604, 0.000379, 0.0005054
7000, 0.0004108, 0.0001828, 0.0004025, 0.0004418, 0.0005824
8000, 0.0007128, 0.0003267, 0.000893, 0.0007513, 0.0011664
9000, 0.0005269, 0.0002298, 0.0005167, 0.0005746, 0.0007769
10000, 0.0006645, 0.0002593, 0.00058, 0.0006304, 0.001015

C:\Users\syedy\Desktop\C++_Assignment_1\Assignment_1_A\x64\Debug\Assignment_1_A.exe (process 5792) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Algorithms:

Brute-Force:

To locate matches, the brute-force technique compares the pattern to every possible substring of the text. It has a temporal complexity of O(n * m), where n is the length of the text and m is the length of the pattern.

Sunday:

The Sunday method uses the pattern to preprocess a shift table to find the next probable location in the text for comparison. It takes O(n + m) seconds to complete, while n is the total length of the text and m is the length of the pattern that is created.

Knuth-Morris-Pratt (KMP):

The KMP algorithm employs a failure function to avoid superfluous text comparisons. The failure function is preprocessed depending on the pattern. It has a temporal complexity of O(n + m), where n is the text length and m is the pattern length.

Rabin-Karp:

The Rabin-Karp algorithm compares the pattern to the text by using hash values. It computes and compares the hash values for the pattern and text windows. It takes O(n * m) seconds to complete, where n is the length of the text and m is the length of the pattern.

Gusfield Z:

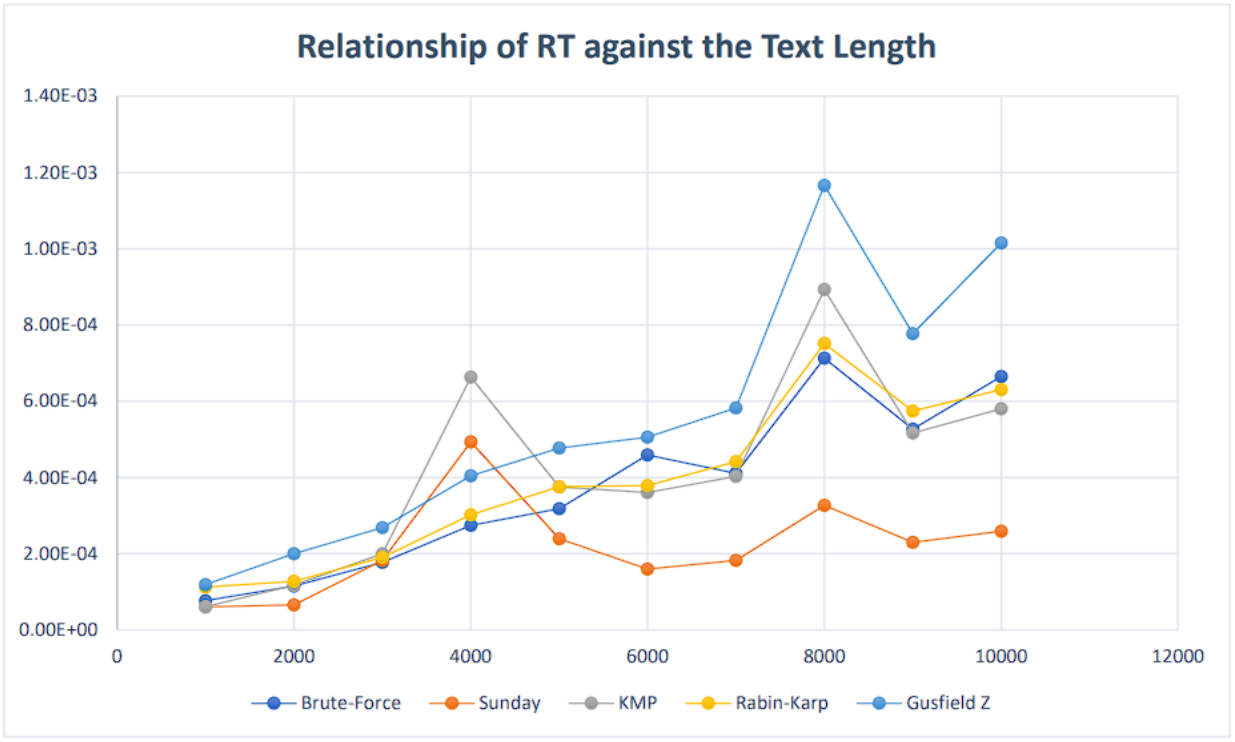
To locate matches, the Gusfield Z method creates a Z array. It combines the pattern, a delimiter, and the text before computing the Z values to find matches. It has a temporal complexity of O(n + m), where n is the text length and m is the pattern length.

Findings:

We may see the following patterns based on the output:

- The execution time of all algorithms typically increases as the length of the text grows.
- The brute-force approach has the shortest time to execute for shorter text lengths but grows slower as text length rises.
- The Sunday method regularly outperforms other algorithms over a wide range of text lengths.
- The KMP method also runs well and has a very consistent execution time across varying text lengths.
- For all text lengths, the Rabin-Karp method takes longer to execute than the other algorithms.
- The execution time of the Gusfield Z algorithm is comparable to that of the KMP and Sunday algorithms.

Chart:



Observations from the line graph:

- Brute-Force: The execution time increases linearly with the text length. This algorithm has the highest execution time among the tested algorithms.
- Sunday: The Sunday algorithm shows a relatively consistent execution time across different text lengths. It performs better than the Brute-Force algorithm in most cases.
- KMP: The Knuth-Morris-Pratt algorithm shows a relatively consistent execution time similar to the Sunday algorithm. It performs better than the Brute-Force algorithm but slightly worse than the Sunday algorithm.
- Rabin-Karp: The Rabin-Karp algorithm exhibits a varied execution time, with fluctuations as the text length increases. It performs reasonably well in most cases.
- Gusfield Z: The Gusfield Z algorithm has the highest execution time among all the tested algorithms. Its execution time increases significantly with the text length.

Part One B:

Code Explanation:

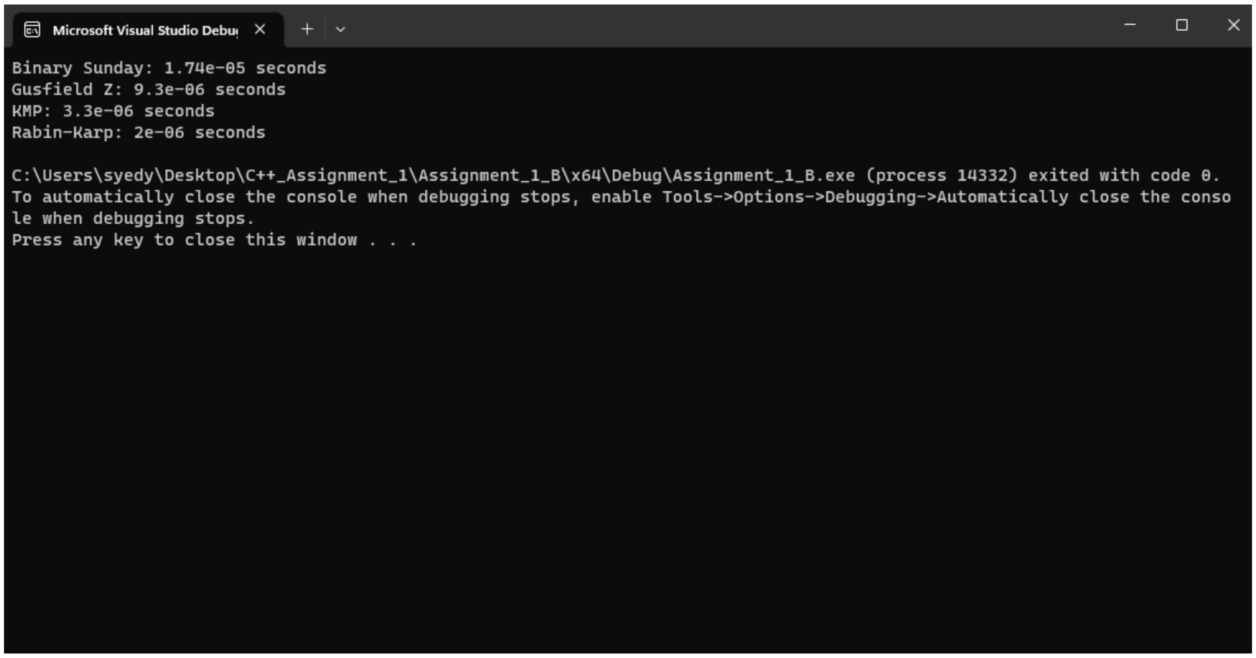
Implementations of four string pattern matching algorithms are included in the code: Binary Sunday, Gusfield Z, Knuth-Morris-Pratt (KMP), and Rabin-Karp. The main function calculates the time it takes each method to match a given pattern inside a text.

The following is an overview of the code structure:

- The code starts by adding the C++ libraries required for input/output, vector manipulation, timing, and random number generation.
- The Binary Sunday pattern matching technique is implemented by the binarySunday function. It accepts a text and a pattern as input and returns a vector of integer points within the text where the pattern may be found.
- The Gusfield Z pattern matching algorithm is implemented via the gusfieldZ function. It also accepts a text and a pattern as input and returns a vector of integer points within the text where the pattern is detected.
- Knuth-Morris-Pratt (KMP) pattern matching is implemented via the kmp function. It accepts a text and a pattern as input and returns a vector of integer locations where the pattern is found inside the text, just like the preceding functions.
- The Rabin-Karp pattern matching method is implemented via the rabinKarp function. It accepts a text and a pattern as input and returns a vector of integer locations where the pattern appears in the text.
- The measureExecutionTime function is a utility function that accepts a function pointer to a pattern-matching algorithm and estimates its execution time on a specified text and pattern.
- A text string and a pattern string are initialised in the main function. The execution time of each algorithm is then measured by executing measureExecutionTime with the appropriate pattern-matching function.

- Finally, each algorithm's execution timings are printed to the console.

OUTPUT:



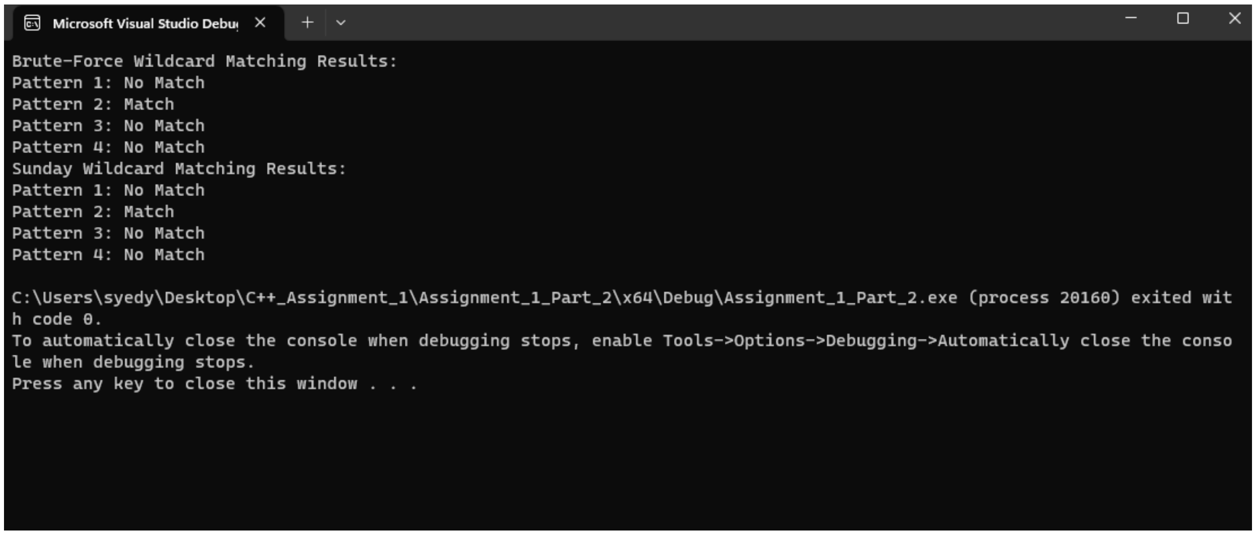
```
Binary Sunday: 1.74e-05 seconds
Gusfield Z: 9.3e-06 seconds
KMP: 3.3e-06 seconds
Rabin-Karp: 2e-06 seconds

C:\Users\syedy\Desktop\C++_Assignment_1\Assignment_1_B\x64\Debug\Assignment_1_B.exe (process 14332) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Explanation:

The result gives the execution times in seconds for each algorithm. Because the execution times are so short (on the order of microseconds), they are shown in scientific notation. The output shows that the execution times for the algorithms drop from Binary Sunday to Gusfield Z, KMP, and Rabin-Karp, in that order. This shows that Rabin-Karp is the quickest of the four algorithms for the given input.

Part Two:

OUTPUT:



```
Brute-Force Wildcard Matching Results:
Pattern 1: No Match
Pattern 2: Match
Pattern 3: No Match
Pattern 4: No Match
Sunday Wildcard Matching Results:
Pattern 1: No Match
Pattern 2: Match
Pattern 3: No Match
Pattern 4: No Match

C:\Users\syedy\Desktop\C++_Assignment_1\Assignment_1_Part_2\x64\Debug\Assignment_1_Part_2.exe (process 20160) exited wit
h code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Explanation:

The given code supports the Brute-Force and Sunday wildcard matching algorithms, as well as the escape character '' and wildcard characters '?' and '*'. The program then runs these algorithms on a given text using four different patterns and reports the results.

Explanation of the extensions:

The given code adds support for the wildcard characters '?' and '*' to the Brute-force and Sunday algorithms. It also allows you to escape wildcard characters and the backslash character '' by preceding them with a backslash.

Brute-force Wildcard Matching:

The bruteForceWildcard function extends the Brute-force technique to support wildcards. It compares the text and pattern characters one by one as it iterates through them. The algorithm exits the loop if the pattern character is not a wildcard '?' and does not match the associated text character. If all of the characters in the pattern match until the conclusion, the function returns true, signifying a match.

Sunday Wildcard Matching:

The sundayWildcard function adds support for wildcard characters to the Sunday algorithm. It functions similarly to the original Sunday method, but with changes to account for wildcard behaviour. When there is a mismatch between the pattern and the text characters, it checks to see if the next character in the text is a wildcard '?' or not in the pattern. Because the pattern can match any character at that place, it moves the pattern by m + 1 position (where m is the pattern length). If the following character is not a wildcard and occurs in the pattern, the shift is determined by the character's rightmost occurrence in the pattern. This shifting operation is repeated by the algorithm until a match is discovered or the end of the text is reached.

The backslash letter '' is utilized as an escape character in both algorithms. If a backslash comes before a wildcard character or another backslash, the algorithm takes it as a literal character and compares it to the relevant text character directly.

Multiple test patterns containing wildcard characters are generated in the main function, and each pattern is matched against the provided text using both the Brute-force and Sunday wildcard matching methods. The results (match or no match) for each pattern are then written to the console.

Part Three:

Explanation:

The function attempts to identify whether a duplicate corner exists in a given image. A corner is defined as a KxK submatrix in the upper-right corner of the image. The code determines if the image has two identical corners.

compareCorner function:

- This function accepts as input the image, two corner points (x1, y1) and (x2, y2), and the size K.
- It compares the comparable elements in the KxK submatrices at (x1, y1) and (x2, y2) in the image.
- It returns false if any element in the submatrices varies, indicating that the corners are not identical. If not, it returns true.

checkDuplicateCorner function:

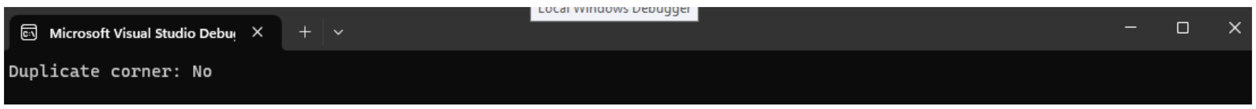
- This function accepts as input the image, the number of rows (M), the number of columns (N), and the corner size (K).
- It initially determines if the image dimensions are adequate to have a corner of size K. If not, it returns false immediately.
- It creates a corner hash variable to hold the top-right corner's hash value.
- It calculates the first corner hash value by iterating over the first K rows and final K columns using a rolling hash technique.
- The corner hash values and their places are then stored in an unordered map called cornerHashes.
- The map now includes the first row's corner hash.
- The rolling hash is then calculated for each succeeding row by subtracting the previous row's contribution and adding the current row's contribution.
- It saves the corner hash and the position in the map for each row.
- Following that, it looks for matches in the top-right corner of the image.
- Starting with the Kth row, it computes the corner hash value.

- If the corner hash matches any of the hashes stored in cornerHashes, the function iterates over the matching corners and compares them using the compareCorner function.
- If an identical corner is identified, it returns true, indicating that a duplicate corner exists.
- It returns false if no duplicate corners are identified.

main function:

- A sample image is defined as a 2D vector of numbers.
- The image is used to calculate the number of rows (M), columns (N), and corner size (K).
- It uses the image, M, N, and K as inputs to the checkDuplicateCorner function.
- Finally, if a duplicate corner is detected, it writes "Duplicate corner: Yes" and "Duplicate corner: No" if no duplicate corner is found.

OUTPUT:



```
Duplicate corner: No
```

Explanation:

The code returns "Duplicate corner: No," indicating that there are no duplicate corners in the provided image.