|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Resultado de imagen para tec de monterrey | **Materia**: TC1018, ED | **Profesor**: Mónica Larre | **Tipo**:  Actividad colaborativa de diseño de código en salón de clase | **Fecha**:  2 sep 2019 |
| **Tema**:  Algoritmos de búsqueda en strings | | | |
| **Objetivo**:  Aprender sobre la eficiencia en los algoritmos para localizar ocurrencias de substrings en strings. | | | |
| **Descripción**:  String matching is an important problem in computer science: it is central to most text processing applications. Exact string-matching searches for occurrences of a substring (the "pattern") within a larger string (the "text"). Approximate string matching, which is considerably more complex, is used for things like spelling correction.  **Part 1**  **Brute Force String Matching Algorithm**  The simplest way to find all occurrences of string pat within string text is simply match pat against text at every position. This is known as the "brute force" approach, since it tries every possible solution. It is easy to program using method *compare*() provided by class string. This method is overloaded, with several different types of arguments.  The form used here will be "*compare(const& pat, size\_t pos, size\_t len,*)", where parameter POs is the position within the string to start the comparison, parameter Len is the number of characters of the string to check (starting from POs), and parameter pat is the pattern. The method returns value 0 if the match is exact, or a positive (or negative) value if the pattern is lexicographically greater than (or less than) the string.  Here is a function in C++ to print out all the positions of text that match pat.  void search\_all(string const& text, string const& pat) {  int const pat\_size(pat.size());  int const endpos(text.size() - pat\_size + 1);  for (int POs(0); POs < endpos; ++POs) {  if (text.compare(POs, pat\_size, pat) == 0) {  cout << "Match at: " << POs << endl;  }  }  }  **Algorithm 1.**  Compile and run the program in file compare\_demo for a verbose demonstration of this algorithm. The expected output must be like:    You can use:    void search\_all(string const&, string const&);  int main()  {  string the\_text = "panamanian banana fanatics can manage anacondas";  string the\_pattern = "ana";  search\_all(the\_text, the\_pattern);  system("pause");  return EXIT\_SUCCESS;  }  **Question 1**  If text is of length n, and pat is of length m, how many calls to *compare()* will be made by *search\_all(text,pat)*?  Method *compare*() compares characters one at a time until it either finds a mismatch or the match is complete. Comparing string "abcd" against string "abcd" results in 4-character comparisons, while comparing "abcd" against "axyz" makes just 2-character comparisons before rejecting the match.  **Question 2**  How many total character comparisons would be made by *search\_all*("aabcd","abcd")?  **Question 3**  How many total character comparisons would be made by *search\_all("aaaab","aaab")*? The text string has four a's followed by one b. Now, how many comparisons would be made if the text string had five, six, seven, or eight A's followed by one b? String Matching Using Iterators Iterators can be used to compare characters directly instead of calling method *compare*(). If "it1" is a string iterator, then "\*it1" is the character to which it points. A version of function *search\_all*() that performs its own character-by-character comparisons requires a nested loop.  **Algorithm 2**  In a file named "*iter\_demo*", write a program that performs the same brute force search as the program in file *compare\_demo.cpp*, but uses iterators to keep track of positions in the text and in the pattern. Do not be concerned with lexicographic ordering for mismatches when outputting the result of the comparison; just output 1 for a match, 0 for a mismatch. Include code to count each character comparison performed and output the total count at the end.  **Part II** Smarter String Matching Brute force algorithms are inefficient because they do not make use of knowledge about the pattern for which the search is conducted; they march blindly through the text string trying every comparison possible. They can be improved. Whenever a match terminates, we can exploit information about the part of the pattern that matched successfully to skip some later comparisons for which we already know the answer.  To understand, consider matching the pattern "*quartz*" against the text "*quartet of quality quartz questioners*". Begin by searching at position 0 of the text. The first five characters, "q", "u", "a", "r", and "t", match the text, but the fifth character, "z", does not match, failing at the "e". The brute force match algorithm would simply move on to position 1 of the text and try again, comparing "q" in the pattern against "u" in the text. Here lies an opportunity for improvement. Notice that "q" does not appear anywhere else in the pattern except position 0. Since we know that the first five characters of the text matched the pattern, positions 1 through 4 of the text are known, and do not contain a "q". So, that part of the text can be safely skipped, beginning the next search at position 5. Unnecessary comparisons of "q" against the already-examined characters in positions 1 through 4, are avoided.  Now consider another situation where comparisons can be avoided: when a pattern contains a repeated element. Try to match the pattern "abcabdabp" against the text "abcabdabcabdabpa". The match is illustrated below, where the vertical bar indicates the point there the match terminates, and the numbers indicate position.  0 1 2 3 4 5 6 7 | 8  Pattern: a b c a b d a b | p    Text: a b c a b d a b | c a b d a b p a  0 1 2 3 4 5 6 7 | 8 9 10 11 12 13 14 15  The characters at pattern positions 0 through 7 match the text, but at position 8, the pattern's "p" failed to match the text's "c". Call the portion of the pattern that matched successfully the head of the match. In the illustration, the *head* is the portion of the pattern to the left of the vertical bar. Its value is "abcabdab". Since the pattern has failed to match the text when starting at position 0, the search moves on. A clever observation reveals that the *head* indicates the length of the move. Notice that the *head* begins with "a", and the next "a" does not occur until 3 characters later. Therefore, the next "a" in the text is 3 characters beyond the current position, so the intervening text characters can be skipped, and the next search can begin at position 3. Can a longer length be skipped?  Look more closely at the example above, where the head "abcabdab" is 8 characters long. If skipping to position 3, position 0 of the pattern ("a") would be matched against position 3 of the text, position 1 of the pattern ("b") to position 4 of the text, and position 2 of the pattern ("c") to position 5 of the text ("d"), which fails. Shown below.  0 1 | 2 3 4 5 6 7 8  Pattern: a b | c a b d a b p  |  Text: a b c a b | d a b c a b d a b p a ...  0 1 2 3 4 | 5 6 7 8 9 10 11 12 13 14 15  Design an algorithm; call it *smarter\_search.cpp* that improves the brute force method using the strategic explained earlier. Include code to count each character comparison performed, and output the total count at the end.  Exploring other ideas. The failure could has been predicted by scrutinizing more carefully the head of the previous match, as illustrated below:  0 1 2 3 4 5 6 7 |  Head: a b c a b d a b |  If skipping ahead three positions, then position 0 of the head will have to match what used to be position 3 of the head ("a" = "a"), position 1 of the head will have to match what used to be position 4 ("b" = "b"), and position 2 of the head will have to match what used to be position 5, but "c" does not equal "d", so the match was destined to fail. Now notice that there is a third "a" in this head, as position 6. Consider, then, skipping six positions. Position 0 would have to match position 6 ("a" and "a"), and position 1 would have to match position 7 ("b" and "b"), and that's the end of the head. By skipping six positions to restart the search, all knowledge about the text characters examined so far without finding a reason for the match to fail have been exhausted. Therefore, the next search should be conducted at an offset of six from the previous search. In fact, since the first two characters of the pattern match the last two characters of the head, eight characters can be skipped (the length of the head) and the next search can begin by using position 2 of the pattern (character "c") against position 8 of the text, as illustrated below.  0 1 2 3 4 5 6 7 8 |  Pattern: a b c a b d a b p |  Old head:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |  Text: a b c a b d a b c a b d a b p | a ...  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 |15  ^ ^  ^ ^  ^ ^ resumed comparing characters here  ^  ^ new match positioned here  Consider determining a general rule for exploiting the head of a failed match. Find a skip-ahead position i, greater than 0, that has the same character which appears at position 0. Furthermore, position i+1 must have the same character as position 1, position i+2 must have the same character as position 2, and so on, or the match is destined to fail. So, when choosing a position i, all the characters from there to the end of the head must match, correspondingly, the characters starting at the beginning of the head. More abstractly, identify the longest prefix of the head that is also a suffix of the head. Call the length of this sequence L. Given a head of length H, find the longest length such that the characters at positions 0 through L-1 match those at positions H-L through H-1.  For example, given the head "abcabdab", the longest prefix that is also a suffix is "ab", so L=2. Therefore, after failing to match the pattern at position 0 of any text, skip forward by the length of the head (to position 8 for "abcabdab") and start matching the text there against position L of the pattern, which would be "c".  Returning to the "quartz" example, when the match failed at position 0 of the text, the head was "quart". Since only the null prefix is also a suffix of this head, L=0, and the algorithm will skip ahead by 5 characters (the length of the head) and begin looking for a "q" (position 0 in the pattern.) The Morris-Pratt Algorithm The Morris-Pratt algorithm automates the exploitation of information about the pattern and its set of possible heads. For a pattern of length m there are m+1 heads, reflecting the number of characters that matched successfully, from 0 through m. An MPtable of size m, where MPtable[H] indicates the point, L, in the pattern from where to begin a new search, where H is the length of the head. The new match will begin H-MPtable[H] characters ahead in the text. After a failed match, compare the character at position L of the pattern with the character L positions ahead of the new starting location of the search.  Each entry in the *MPtable* is the length of the longest proper prefix of a head of length H that is also a proper suffix of the head. (A proper prefix is one whose length is less than the length the entire head.) Following is the *MPtable* for the pattern "abcabdabp". The first entry, for H=0, has been defined as L=-1 since, if the very first character of the pattern fails to match, in which case the head has length 0, the skip length HL should be 1. (Of course, a search cannot begin at position -1 of the pattern since there is no such position. The search must start at position 0. That aspect will have to be handled specially by the code that reads the table.)   |  |  |  | | --- | --- | --- | | **Head** | **H** | **MPtable[H]** | | - | 0 | -1 | | a | 1 | 0 | | ab | 2 | 0 | | abc | 3 | 0 | | abca | 4 | 1 | | abcab | 5 | 2 | | abcabd | 6 | 0 | | abcabda | 7 | 1 | | abcabdab | 8 | 2 | | abcabdabp | 9 | 0 |   Study the table to understand how the values are computed. Notice that, in the last entry, the head is equal to the entire pattern. Although the match can never fail if the entire pattern has matched successfully, this entry is necessary in order to restart the search for the next occurrence of the pattern in the text.  **Question 4**  Write the MPtable for the pattern "jarjar".  **Algorith 3**  In a file named "mp\_demo", write a program that accepts a string argument on the command line and outputs the MPtable for the string. The program must include a function "int MPentry(string const& pat, int i)" that returns the ith entry in the MPtable for parameter *pat*. The Knuth-Morris-Pratt Algorithm The Knuth-Morris-Pratt algorithm makes one improvement to the Morris-Pratt algorithm. Consider the pattern "abcabc". Suppose a match fails at position 5 of the pattern, which is the second "c". The head is thus "abcab", and the longest prefix that is also a suffix is "ab", so MPtable[5] would be 2. But notice that the character that comes after this head in the pattern, meaning the character on which the match failed, is the same character that occurs after the prefix: a "c". Given that the match failed on a "c", restarting the match HL=3 characters ahead is also guaranteed to fail. Therefore, "ab" is not an acceptable prefix/suffix. A smarter value for this table entry would be zero. (Notice that, since HL characters are skipped, a smaller L value results in more characters to skip.) Here is a comparison of the tables used by the two algorithms for the pattern "abcabc". Notice that the entries for "abca" and "abcab" skip ahead by 6-0=6 characters according to the KMPtable. But the last entry, "abcabc", still only skips ahead by 6-3=3 characters. This is because there was no character that failed to match. The head equals the entire pattern, so text beyond the head has not yet been examined.   |  |  |  |  | | --- | --- | --- | --- | | **head** | **H** | **MPtable[H]** | **KMPtable[H]** | | - | 0 | -1 | -1 | | a | 1 | 0 | 0 | | AB | 2 | 0 | 0 | | ABC | 3 | 0 | 0 | | abca | 4 | 1 | 0 | | abcab | 5 | 2 | 0 | | abcabc | 6 | 3 | 3 |   **Question 5**  Draw a table containing MPtable values and KMPtable values for the pattern "aaaaaa" (six a's in a row).  **Algorithm 4**  Begin with a copy of file mp\_demo naming it "kmp\_demo", and modify it to use the Knuth-Morris-Pratt algorithm.  **PART III**  **Green Gene Machine** Objectives Implement the Knuth-Morris-Pratt string matching algorithm and use it to search for patterns in a DNA sequence. Description The "genetic code" is the means by which DNA encodes the instructions for making proteins. Strands of DNA can be regarded as strings. While C++ strings are sequences of characters, DNA strings are sequences of small molecules called "bases" or "nucleotides". English uses a twenty-six-letter alphabet, but DNA is restricted to just four bases: G, C, T, and A.  In this exercise, the chloroplast genome of Chlorella vulgaris, a type of green algae, will be examined. The chloroplast is the structure within the cell responsible for photosynthesis. Its genome contains a variety of genes related to that process, including the gene for chlorophyll, which gives the algae its greenish color. The genome contains 150,613 bases. It can be thought of as a string of length 150,613 composed of the letters G, C, T, and A. The genome is provided in the file chlorella\_vulgaris.dat.  One-way scientists study long DNA sequences is to apportion them up using restriction enzymes. A restriction enzyme snips a DNA strand at places where a specific pattern occurs. Biologists have isolated restriction enzymes from various organisms and discovered the patterns used by each enzyme. The biologists use the enzymes to cut DNA they wish to analyze. Observing at the sizes of the fragments produced by different restriction enzymes provides important clues about the layout of genes on the DNA strand.  Following is a table of three restriction enzymes and their associated sequences.   |  |  |  | | --- | --- | --- | | **Enzyme** | **Source** | **Sequence** | | MstII | Micocoleus | CCTAAGG | | NotI | Nocardia otitidis | GCGGCCGC | | TaqI | Thermus aquaticus | TCGA |   The task will be to find, for each restriction enzyme, all the locations in the chloroplast genome where the enzyme could make a cut.  **Algorithm 5.**  Write a program that locates DNA sequences in the genome in file chlorella\_vulgaris.dat. The program must accept one or two command line arguments. If only one argument is provided, it is assumed a DNA sequence. The program must, then, open and internally store the genome in file chlorella\_vulgaris.dat, and output the starting index of each occurrence of the DNA sequence found in the genome. The search must be conducted twice. One must use the Knuth-Morris-Pratt string matching algorithm, and the other must use the brute force string matching algorithm also described in the Algorithms part of this assignment. Both searches must report (output) the total number of comparisons used. In summary, the output must include both the list of each starting index of the DNA sequence found in the genome, and the total number of comparisons used by each of the two algorithms.  For smaller, more localized searches, the program must accept an optional second argument. The second argument is to be used in place of the genome found in file chlorella\_vulgaris.dat. For example, to find "a" in "abracadabra", supply "a" as the first argument and "abracadabra" as the second argument. Note that, when two arguments are supplied, opening and loading the genome in file chlorella\_vulgaris.dat is not necessary. Save the program in a file named "search\_chlorella\_vulgaris.cpp Processing chlorella\_vulgaris.dat As required for the one-argument behavior, the genome in file chlorella\_vulgaris.dat must be internally stored and searched. Due to the format of file chlorella\_vulgaris.dat, storing the genome will require some processing.  Open and observe the format of file chlorella\_vulgaris.dat. Notice that the genome is broken into lines, and further broken into "chunks" on each line. Notice, too, that each line is preceded with a line number. Use a statement like "cin >> junk" to extract (and ignore) the line number. Then use function *getline*() to extract the rest of the line. For each extracted line, use function *remove*() to remove spaces and carriage returns ("'\r'"). Remember that function remove() repositions elements and returns an iterator pointing to the new end of the sequence, placing all "removed" elements into a position that lies after the iterator returned. Then, concatenate the useful part of the new sequence (up to the position marked by the iterator returned) to a master string to accumulate the entire genome (devoid of line numbers, spaces, and carriage returns) in the master string.  Provide the usual error messages about program usage and about failed file open attempts. | | | | |
| **Entregables:**  **Un proyecto con un menú que incluya las opciones:**   1. *compare\_demo* 2. *iter\_demo* 3. mp\_demo 4. *kmp\_demo* 5. *search\_chlorella\_vulgaris* 6. *theory1 (print here your answers to questions 1, 2 and 3)* 7. *theory2 (print here your answers to questions 4 and 5)* | | | | |
| **Fecha y forma de entrega**:  En la plataforma  Parte I  Parte II  Parte III | | | | |