CPE593 2021S Project Descriptions

Your team may select one of the projects here, or you may define an equivalent project of your own and get it approved. Your project must involve considerable code implementation (minimum 500 lines of real code, not fluff). If you turn in a minimal project, you will receive a minimal grade. If you really have a project that is very theoretical you may do less code and more of a paper, but then your theory and analysis must be impeccable.

Projects I have defined are one per page below:

|  |
| --- |
| Compressed Trie Dictionaries (english) |
| Compressed Trie Dictionaries (Chinese) |
| Document and re-implement bzip2 algorithm |
| Document and re-implement LZMA algorithm |
| Factoring Common Text in Documents |
| Gravity Simulator |
| Factoring Common Text in Documents |
| RopeLike Text Buffer |
| Implementing a lexical analyzer with combined regex engines |
| Binary Diff/Patch |
| Automated Exploratory Data Analysis |

|  |
| --- |
| Compressed Trie Dictionaries (english) |
| Given an English dictionary with a set of words, compress that dictionary using at least two schemes. Compare how much compression is obtained with each method as well as with gzip, bzip2, lzma and paqh8x |
| Method 1: create a trie of hashes where the first 3 letters are used to select which hashmap is used. There are 26\*26\*26 hashmaps, with one special one for storing all words < 3 letters long. All words are encoded using arithmetic encoding. If the node is not a leaf, then it is followed by all the trie nodes for each child. |
| Method 2: Create a trie of hashes but this time, the top level is a single character, and at each level, a trie node consisting of a 1-bit flag (isWord) 1-bit flag (leafNode) and a bit vector of each letter a-z. If the node is a leaf, then a single byte encodes how many words there are in the leaf, followed by the words encoded using arithmetic encoding. If the node is not a leaf, then it is followed by all the trie nodes for each child for whom a 1-bit is specified. |
| Method 3: come up with your own idea to represent the dictionary and store it as efficiently as possible. |
| Compare the three methods, the existing methods of compression for how much they compress the dictionary, how much CPU it takes to uncompress and load into memory. |
| All methods should write a binary compressed dictionary to disk and be able to read it back in and convert it back to the input for testing purposes. |

|  |
| --- |
| Compressed Trie Dictionaries (Chinese) |
| Using the compressed Trie dictionary in English as an example, write code which does the same for a dictionary in Chinese. Note that you will have to learn about Chinese encodings, find a dictionary before you can even begin. The purpose can be to aid in auto-completion, or anything else, but the main purpose of this project is to store a Chinese dictionary in a compact binary file. |

|  |
| --- |
| Document and re-implement bzip2 algorithm |
| Explore the bzip2 compression program and find out how it works (hopefully not by reading the code, that would be the most difficult way). Then explain the algorithm and duplicate it. Your test is whether you can compress a file with your program and decompress it with bzip2, and vice versa. |
| This includes the core algorithm of Burrows-Wheeler that rearranges blocks into a form more amenable to compression, but also run-length encoding, Huffman and lzw |
| see: <https://en.wikipedia.org/wiki/Bzip2#:~:text=bzip2%20is%20a%20free%20and,version%200.15%2C%20in%20July%201996>. |

|  |
| --- |
| Document and re-implement LZMA algorithm |
| In a manner exactly analogous to bzip2 above, implement and describe the LZMA algorithm:  see: <https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm#Compression_algorithm_details> |

|  |
| --- |
| Gravity Simulator |
| Given an example n-body simulator using simple Euler numerics, set up an example of a 2-body system (ie earth and moon). Calculating forward in time will result in errors due to roundoff. Write code to measure the change in total energy for the system as a check. Implement methods rkf45 (Runge-Kutta-Fehlberg) and Predictor Corrector (Adams-Bashforth) and first make sure both work. |
| Make sure your code works on a multi-body system as well. Use sun and planets out to Saturn as a test case. You can restrict your study of the numerical stability to the simple 2-body case however. |

|  |
| --- |
| Factoring Common Text in Documents |
| Get at least 20 English documents from project gutenberg. Notice that the top and bottom of each document contains boilerplate. I hate this redundancy. Unfortunately over the years, they have changed their boilerplate so there are multiple versions so make sure you have at least two versions with multiple files from each. Write a program that identifies common text at top and bottom, separates it into a separate file and replaces the text with a hypertext link to the file. You can keep only a single copy of one kind of the file and let every document point to the same one. Analyze the complexity of this code as well as implementing it. |

|  |
| --- |
| RopeLike Text Buffer |
| Rope is a tree-structure that holds text in order to insert into large strings faster than a string.  Your goal is to create an object that can manipulate large (100MB+) strings rapidly and compare speed to traditional strings. |
| First create a class that can load a large text file from disk. You will benchmark the following actions. For all actions that are too fast, like inserting 1 byte, do the action n times and divide by n so the benchmark is accurate:   * load 100Mb file of text * get the kth byte * get a string from byte m to n * replace a string from byte m to n * inserting 1 byte in the middle of the file (offset=50 million for example) * inserting 10 bytes, 100 bytes, 1k bytes 10k bytes. (same as above) * appending to end of file 1, 10, 100, 1k, 10k bytes. * deleting 1, 10, 100, 1k, 10k bytes and the entire file. * saving the entire text to a file   Method 1: Create a tree with degree k. You will try k=2, k=4, k=16. For example, with degree 4, each node has 4 children. At the lowest level, the nodes contain 4 strings, each of which is a string for a line. Implement all the same methods described above and compare the Ropelike data structure speed to the original string implementation. |

|  |
| --- |
| Implementing a lexical analyzer with combined regex engines |
| Existing tools like lex, flex etc are complicated. The purpose of this project is to make a very high speed lexical analyzer that implements regular expressions and allows a simple program to determine which of several regular expressions matches a pattern without having to execute multiple regex matches.  This project should include a symbol table for storing values but need not support a full grammar with a stack. This project requires that you IMPLEMENT a complicated regex engine, not use one from your programming language. That’s the data structures component of the project: an efficient regex implementation that uses 1 rather than n separate regex parsings. |
| Example: I write a simple language containing a number of statements. I can currently write a regex parser for each kind of statement, but in order to parse my language I must repeatedly apply each regex and only match the one.  The following example shows a config file. The goal is to store symbols with various types including strings, integers, double, lists, regular expression patterns and objects which are hash maps.  Write a parsing tool that allows the programmer to parse a line and have a simple switch statement depending on which case it is:  int cmd = parser.parse(currentLine); if (cmd == NOMATCH) {  // forget it…. }  string sym = parser.get(1); // symbol is the first switch (cmd) { case COMMENT:  break; case INT:  symtab.put(sym, convertStringToInt(parser.get(2)));  break; case DOUBLE:  symtab.put(sym, convertSTringToDouble(parser.get(2))); …   } |

// comment goes here

port 8000

Na 6.022e23

s “this is a test”

r /ab\*c/  
list [1, 2, 3, 5, 19]

|  |
| --- |
| Binary Diff/Patch |
| In order to patch programs, binary data must be changed. The worst case is to simply transmit a completely new executable. However, most of the time programs are changed only slightly. Find papers on binary differencing, and implement an efficient algorithm to perform it.   Should support bytes being inserted, deleted or changed.  See existing utilities bspatch and bsdiff for ideas and sample source code. |
| Testing: The binary patching code should be capable of modifying an existing executable into a new one by applying changes. This may be tested by generating an executable, modifying the code slightly and generating it again, then demonstrating the ability to ship a patch file which can turn the old executable into a new one.  Generate at least four executables. Take a small sample program, use the optimizer and make sure that the three executables are different -- if the optimizer eliminates the differences then your code is meaningless.   * Write a small program * Insert a new line of code into the source that computes an expression from a variable and prints it so the optimizer cannot remove the code. Write this executable out. Verify that the two executables are different using the unix utility cmp. * Generate a patch using your software to create the second executable from the first. * Delete a line of code. (same procedure). * Take a larger program composed of many files from CPE553. Make the difference substantial, with 10k source in the first, and 20k in the second so that the time is more significant. * Should also work on Binary data (try writing a binary file of integers, and inserting in some places, deleting in others) |

|  |
| --- |
| Automated Exploratory Data Analysis |
| Given a file of data, try to find hidden patterns within. Write all the algorithms individually and determine the complexity to execute them all and determine the complexity of the overall algorithm. |
| Algorithms include the following. For each matcher, return the RMS error as the metric of quality to compare what is the best fit   * Floating point (double)   + Constant value   + Find a linear sequence   + Find a quadratic sequence   + Find a 2d linear sequence (gradient in horizontal and vertical)   + Find a 2d quadratic sequence   + An exponential fit AeBt+C   + A sine wave fit Asin(Bt + C) including phase   + Fitting a cubic spline to data for data which should be smooth   + Fitting a Piecewise Cubic Hermite (PCHIP) to data for data where derivatives can be disjoint * Integer   + constant value   + Find a linear sequence   + Find a quadratic sequence   + Find a 2d linear sequence (gradient in horizontal and vertical)   + Find a 2d quadratic sequence   Write a driver that attempts to find the best fit in given data, and then write out the information with the fit followed by the residual error. Using bzip, compress the original file, and compare to the output of this program and the output of this program piped to bzip2. |
| Testing: Create the following data in binary files in order to test your code:  A linear series 1, 2, 3, 4, ...n to which you add normally distributed random data.  A linear series a, a+k, a+2k, … a+nk to which you add normally distributed random data.  Repetitive elliptic motion of the form x = Acos(t), y = B sin(t) + noise |