CSCI-311 Project 1

Part 1. In this Project, you will implement the tree algorithms: QuickSort, Insertion and MergeSort. Each of these algorithms will sort suffixes of a given string S in alphabetical order. You are not allowed to store separate suffixes of S, instead, you will learn how to sort suffixes (or any strings) using indirect access to suffixes via pointers. In our case, we don't need to use pointers, we can just use the starting positions of suffixes to access each suffix inside S. Once your program will read S from an input file, and will convert S to lower case, it will create a vector of integers *indices* that will hold the starting positions of suffixes of S; initially, *indices*[i] = i. Whenever an algorithm needs to swap suffixes, instead, it will swap two indices, the starting positions of the corresponding suffixes. This design technique saves space (an index of a suffix takes only 4 bytes, whereas a suffix of a long string might take much more space) and saves time (to swap two strings, one needs to copy a string that takes O(n)-time, where n is the length of a string, but copying an integer index takes only constant O(1)-time).

Step 1: you need to write four functions:

1. Function ***readFromFile*** will take as a parameter an empty string S by reference, and a string, name of a file. This function will open the given file, and will read all the lines from the file and concatenate the lines into S. At the end of the function, S will be the concatenation of all lines from the given file.
   void *readFromFile*(string &S, string filename)

2. Function ***convertToLower*** will take a string S passed by reference as a parameter and will convert all the characters in S to lower case (you may use std function ***tolower***).
   void *convertToLower*(string &S)

3. Function ***lessThan*** will take as parameters a constant string passed by reference *S*, and two integers, *first* and *second*, the starting indices of two suffixes in *S*. This function will compare the two suffixes of S starting at their starting positions and to the end of S one character at a time. The function will return true if the suffix starting at *first* is less than the suffix starting at position *second.*
   IMPORTANT: do not use ***substr*** to extract suffixes from the given string; do not copy suffixes into a separate strings – this takes O(n) time and O(n) space. We would like to save time and space.
   bool *lessThan*(const string &S, int first, int second)

4. Function ***partition*** that takes as parameters (1) string ***S*** of size *n*, passed constant by reference (2) vector ***indices*** of integers of size *n*, where each integer is the starting position of a suffix in S; *indices* are passed by reference (3) integers ***low*** and ***high,*** the start and the end indices of a given range; and (4) an integer ***pivotIndex***, which is an arbitrary index of the vector *indices*.
   int *partition*(const string &S, vector<int> &indices, int low, int high, int pivotIndex)

   - ✓ First, this function swaps *indices*[*pivotIndex*] and *indices*[*high*], where *high* is the last index in the range.
   - ✓ Then, this function will partition suffix indices (use code of Partition provided in lecture notes as an example) so that the first half contains positions of suffixes that are less than suffix *pivot* and the second contains indices of suffixes that are greater than suffix *pivot.* This function will call the functions *lessThan* to accomplish this task.

Step 2: Re-write QuickSort, Insertion and MergeSort to sort suffixes of a given string S. You will use *lessThan* function to compare two suffixes given by the starting positions.

Algorithm	Description	Test files
Quicksort	Input parameters: 1. Constant string S by reference. 2. Vector of integers *indices* holding indices of suffixes of S. 3. Integer *low,* the lowest index in the range. 4. Integer *high*, the highest index in the range. Output: Use ***cout*** to print out the indices from the vector *indices* in this format: <index><space>…<index><space><endl> QuickSort must sort suffixes of the input string S in alphabetical order. Instead of storing each suffix separately, and swapping suffixes (strings), it will access suffixes of S indirectly via indices that are stored in *indices.* The indices of suffixes represent the starting positions of suffixes in S. Use ***partition*** function that you wrote (above) inside this QuickSort.	t15 t16 t17
Insertion	Input parameters: 1. Constant string S by reference. 2. Vector of integers *indices* holding indices of suffixes of S. 3. Integer *low*, the lowest index of the range. 4. Integer *high,* the highest index of the range. Output: Use ***cout*** to print out the indices from the vector *indices* in this format: <index><space>…<index><space><endl> Insertion must sort suffixes of the input string S in alphabetical order. The function will sort the suffixes of S by calling ***lessThan*** function and swapping positions of the suffixes.	t01, t02, t03, t04, t05, t06, t07, t08
MergeSort	Input parameters: 1. Constant string S by reference. 2. Vector of integers *indices* holding indices of suffixes of S. 3. Integer *low,* the lowest index in the range. 4. Integer *high*, the highest index in the range. Output: Use ***cout*** to print out the indices from the vector *indices* in this format: <index><space>…<index><space><endl> The function will sort the suffixes of S by calling ***lessThan*** function and swapping positions of the suffixes.	t09, t10, t11, t12, t13, t14

Re-write ***Insertion*** sort algorithm to sort suffixes of a given string S. Similarly to QuickSort, your ***insertion*** function will take a string S (passed constant by reference), and a vector of integers with the starting positions of suffixes of S. The function will sort the suffixes of S in the range from *low* to *high* (inclusive indices of the range). Insertion will call ***lessThan*** function (from Assignment 5) and will swap positions of the suffixes instead of suffixes. Here is the header of this function:

```
void insertion(const string &S, vector<int> &indices, int low, int high)
```

Sample input:

S="abracadabra", indices = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, low = 3, high = 7.

insertion(S, indices, low, high); //will sort suffixes starting at positions 3, 4, 5, 6, and 7.

Sample output:

After insertion is called on the *sample input*, vector *indices* will have the following order of suffix positions: indices = {0, 1, 2, **7, 3, 5, 4, 6**, 8, 9, 10}. Only suffixes at indices 3, 4, 5, 6, and 7 are ordered alphabetically.

Main.cpp must include: The file *main.cpp* must include the code for Insertion and Selection and QuickSort in it, as well as `main(int argc, char* argv[])` function that calls these algorithms.

1) Read the input file, argv[1], and concatenate all lines into a single string S. Call function *readFromFile*
1) Convert all characters of S to lower case. Call function *convertToLower*
2) Create a vector of integers *indices* and initialize it *indices*[i] = i (*indices* has the same size as S).
3) Let string "command" be argv[2]. Write if/else if/else statement that depends on the *command.* If *command* is equal to *QuickSort,* then call QuickSort to sort suffixes; similarly, if *command* is *Insertion,* then call Insertion algorithm; if *command* is *MergeSort,* then call MergeSort algorithm.

Error messages:

1) If the input file cannot be opened, print out this message:
```
cerr <<"ERROR: could not open file " << file1 << endl;
```

2) If ***argc*** is less than 3, print out this message:
```
cerr << "ERROR: provide more arguments." << endl;
```

Submission: Submit *main.cpp* file to *Project1* on ***turnin***.

Part 2.
In this part, you will run your program developed in Part 1 to compare running time of QuickSort, Insertion and MergeSort on the given data sets.

Test files for Part 2: download **time_tests.tar**.

Submission: Submit your printed report in person by the deadline in class.

Grading:

Project 1 is worth 200pts: (1) 100pts for Part 1, (2) 100pts for the report.

For each algorithm:

- Read a test file, concatenate all lines into a single string S;
- Then make a vector of integers with the suffix indices for each algorithm (the length of a vector equals to the length of S).
- Measure time that it takes to run an algorithm;
- Output the results to the screen in the format: length of S, run time.
- Record this information for all five input files.
- Build a graph using Excel (or a similar tool) that shows the length of S on x-axis and time in seconds on y-axis.

How to measure time in seconds inside a program:
Include <ctime> header.
Assume that you need to measure how long does it take to run function *partition*, then you need to declare two variables of type ***clock_t***, place one before the function, and the other after the function:

```
clock_t t1 = clock();
partition(… parameters …);
clock_t t2 = clock();
double elapsed = double(t2 – t1)/CLOCKS_PER_SEC; //elapsed is the time in seconds
cerr << "Time to run partition is " << elapsed << " seconds." << endl;
```

Fill in this Report (type and print) **Student's Name:_______________________**

Fill in the table below stating the theoretical asymptotic time-complexity and space-comlexity (using big-O notation) for each of the algorithms (look up the lecture notes):

Algorithm	**Big-O of Time**	**Big-O of Space**
QuickSort		
Insertion		
MergeSort		

Experiment1: Comparison of QuickSort and Insertion to sort all the suffixes of a given string S

Input File	**Size of S**	**Time, sec, QuickSort**	**Time, sec, Insertion**	**Time, sec, MergeSort**
chrY_50.txt				
chrY_100.txt				
chrY_150.txt				
chrY_200.txt				
chrY_250.txt				

Insert your **graph** below (make sure to use different line markers to distinguish lines from different algorithms):

Based on your experimental results, write a conclusion below about which algorithm performed better in terms of time in this experiment, and briefly explain why do you think this is the case: