**CSCE 1101-03, 04 Term Project Spring 2023**

**Guidelines**

The term project is **NOT OPTIONAL.** The project grade represents 20% of the total course grade.

**Guidelines:**

* The project is intended to be a team work
* **The project topic should be different from the topic of the term paper.**
* A group **of 1 - 4 members** is to complete the project by the deadline and report the work in a project report (format attached).
* All group members **must be students from the same section.**
* The project group can be the same as that presenting the term paper.
* Names of group members should be sent by e-mail to the course TA Eng. Mohamed Hany: mohamed.ihany@aucegypt.edu by **Monday April 17th, 2023. No allowance to change the names of group members once they have been sent to Eng. Hany.**
* **Names** of group members **must be submitted** in the final report with **no allowance to change the names from those sent before.**
* The final project report should be submitted on **blackboard.**
* **Final date** of submission of the complete project report is **Monday May 22, 2023**.
* There will be NO late policy for the submission of the project. **No submissions will be accepted after the due date.**

**Academic Integrity**

Students are expected to commit to the principles of academic integrity. Any plagiarism detected will result in zero grade for the project.

You should select **ONE** of the 3 projects presented here:

1. Experimental Verification and Analysis of Some Sorting Algorithms.
2. Simple Plagiarism Detection Utility using String Matching.
3. Simulation of Waiting Queues of Airplanes in an Airport

### **CSCE 1101-03,04 Spring 2023**

### **Term Project (1)**

##### Dr. Amr Goneid *Due: Monday May 22, 2023*

**Experimental Verification and Analysis**

**of Some Sorting Algorithms**

In this project, you are to implement, test and verify sorting algorithms by experiments. A sorting algorithm will be presented with a random array of integer and it should sort these elements in ascending order. In the experiments, you are to use counting to determine the number of array element comparisons *T(N)* done by the sorting algorithm as a function of the array size *N.* You are then requiredto compare experimental results with those derived from mathematical modeling of the algorithm.

**The problem**

Our problem is to compare between experimental results *Texp(N)* using counting of the number of array element comparisons done by a sorting algorithm with those expected from mathematical modeling of the algorithm *Tmodel(N).*

The algorithms subject to the experiments are:

* Selection Sort (Elementary Sorting Algorithm),
* Insertion Sort (Elementary Sorting Algorithm)
* Merge Sort (Divide & Conquer Algorithm),
* Quicksort (Divide & Conquer Algorithm) for three cases of pivot choice

**Methodology**

1. **Input**

In your experiments, the input to the sorting algorithm will be random integers. For that purpose, you will use Random Permutation Arrays (RPA) of the first N integers. Such arrays are often used in testing sorting algorithms. In these arrays, all integers from 1 to N are present but in a random order. No integer is missing and no integer is duplicated. Random permutation arrays are generated by the method given in **Appendix 1.**

Store the RPA in an array of size (N+1), starting from location 1 and ending at location N. Leave location (0) unused. In this way, the array after sorting will have integer (i) at location (i) in the array and you can use this feature to verify that the sorting algorithm is correct.

For example for N = 5, the following table shows the RPA before and after sorting

Table 1. RPA before and after sorting

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Array Location | 0 | 1 | 2 | 3 | 4 | 5 |
| RPA Before | 0 | 5 | 4 | 2 | 1 | 3 |
| RPA After | 0 | 1 | 2 | 3 | 4 | 5 |

**In each experiment, use 6 different array sizes (N = 1000, 2000, 3000, 5000, 7000, 10000).**

**Sorting Algorithms**

The algorithms to test are given below. In your implementation of each algorithm, add counters to **count the number of array element comparisons.**

You will conduct the following 6 experiments. In each experiment, you should find *Texp(N)* for each of the 6 array sizes N **= 1000, 2000, 3000, 5000, 7000, 10000.**

1. Selection Sort: Input is a Random Permutation Array A[1..N]
2. Insertion Sort: Input is a Random Permutation Array A[1..N]
3. Merge Sort: Input is a Random Permutation Array A[1..N]
4. Quick Sort (1): Input is a Random Permutation Array A[1..N], pivot is chosen to be the 1st element in the array / sub-array.
5. Quick Sort (2) (Randomized Quick Sort): Input is a Random Permutation Array A[1..N], pivot is chosen randomly.
6. **Implementing the Sorting Algorithms**

The implementations of all the above algorithms are given in the course slides.

1. **Counting the Number of Array Element Comparisons**

In order to find *Texp(N)* for a given array size N, you should introduce a counters in your implementation code for each algorithm. This counter will give the number of array element comparisons done by the algorithm at the end of the sorting process. Introducing the counter inside the code is shown by the example given in **Appendix 2.**

1. **Output**

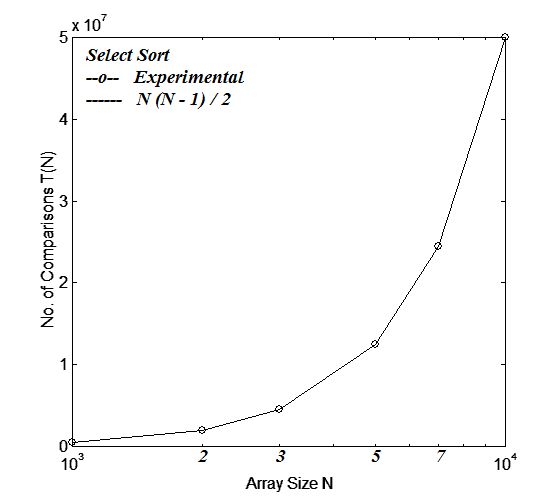
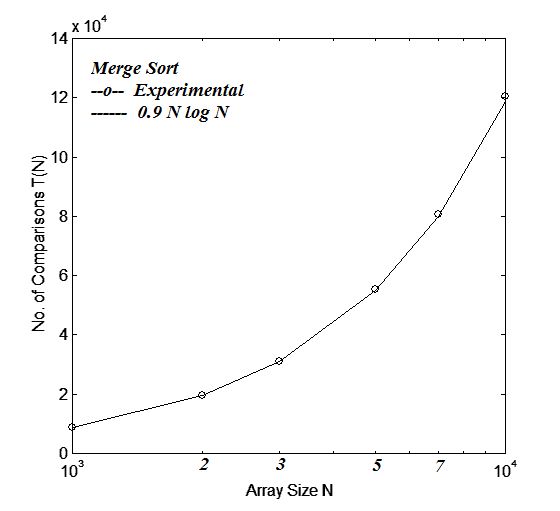
The output of each of the 6 experiments is *Texp(N)* = the number of array element comparisons that you counted as done by the algorithm for each of the 6 array sizes N.

You need also to compare the experimental results with those expected *Tmodel(N)* from the mathematical modeling of the algorithm (see **Appendix 3**). You may show the comparison in tables, for example:

Table 2. Selection Sort Algorithm

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N | 1000 | 2000 | 3000 | 5000 | 7000 | 10000 |
| T(N) experiment |  |  |  |  |  |  |
| T(N) model |  |  |  |  |  |  |

You may also show the comparison in graphical form as in the following two figures:



1. **Analysis**

Provide an analysis of the results in comparison with those expected from the mathematical modeling of the algorithms. **Appendix 3** gives *Tmodel(N)* for the tested algorithms.

1. **Deliverables**

You are to deliver a final report of the project as in the attached format.

**Appendices**

**Appendix 1. Generation of Random Permutation Arrays (RPA) of the first N integers**

Suppose you need to generate a *random* permutation of the first *N* integers. For example, {4, 3, 1, 5, 2,} and {3, 1, 4, 2, 5} are legal permutations, but {5, 4, 1, 2, 1} is not, because one number (1) is duplicated and another (3) is missing. Random permutation Arrays (RPA’s) are often used in testing sorting algorithms. We assume the existence of a random number generator, ***RandInt(i, j)*,** which generates integers between *i* and *j* with equal probability. Here is an efficient algorithm:

***Fill the array such that A[i ] = i. with i = 1, 2, ….N***

***for i = 2 to N {***

***m = RandInt(1,i);***

***swap ( A[i], A[m]);***

***}***

**Appendix 2. Introducing counters inside code:**

**Example:**

Find the location of the minimum element in an array A[1..N ]

To count the number of array element comparisons we increment a counter whenever there is such comparison:

***int count = 0;***

***s = 1;***

***e = N;***

***ALGORITHM MinAt (A , s , e, count)***

***{ m = s ;***

***for j = s+1 to e***

***if (A[j] < A[m]) m = j ;***

***count++;***

***return m;***

***}***

**Appendix 3. Expected *Tmodel(N)* for the Different Algorithms**

Note that *Texp(N)* is the number of array element comparisons that you counted as done by an algorithm. Because the input arrays are all random, *Texp(N)* will represent the average case cost for the algorithm. In order to compare your results for *Texp(N)* with what is expected from a cost model of the algorithm, *Tmodel(N)* must also represent the expected average case cost. These are given in the following:

1. Selection Sort:

This algorithm does not have a best case or a worst case. The only case is the average case where *Tmodel(N) = N(N-1)/2 = O(N)*

1. Insertion Sort:

This algorithm has a best case of *O(N)* and a worst case of *O(N2)*. Its average case is also *O(N2)* and can be represented by a polynomial of degree 2:

*Tmodel(N)* = a *N2 + b N + c*

(see **Appendix 4** on how to get the coefficients a, b, c from your *Texp(N)* data)

1. Merge Sort:

This algorithm has an average case *Tmodel(N) = a N log2 N*

(see **Appendix 4** on how to get the coefficient a from your *Texp(N)* data)

1. , E. Quicksort:

The different variants of Quicksort all have an average case *Tmodel(N) = a N log2 N*

where *a = {a1 , a2 , a3)* for the 2 different variants (D, E), respectively.

(see **Appendix 4** on how to get the coefficients *a1 , a2 , a3* from your *Texp(N)* data)

**Appendix 4: Fitting *Texp(N)* Data**

You can fit your experimental data *Texp(N)* to obtain model coefficients for polynomial fits or other functions using on-line fitting facilities

(e.g. <https://mycurvefit.com/> or <https://planetcalc.com/8735/?license=1>)

You can aslo produce graphical comparisons between the *Texp(N)* data and the fitting curves representing *Tmodel(N)* using Microsoft Excel.

### **CSCE 1101-03,04 Spring 2023**

### **Term Project (2)**

##### Dr. Amr Goneid *Due: Monday May 22, 2023*

**Simple Plagiarism Detection Utility using String Matching**

**Introduction**

Plagiarism is a serious problem in study and research. In this project, you will implement a simple plagiarism detector. Your input will be a collection (corpus) of existing documents and a potentially plagiarized document. Your output will be the set of documents from which the document was plagiarized.

Given a test file, treat each sentence in the file as a potential pattern. Search for the pattern in the existing documents and find the matches.

This problem is typically approached through string matching. In the string matching problem, we are given a string of *n* characters called the *text* and a string of *m* characters (*m ≤ n*) called the *pattern* and we want to find a substring of the *text* that matches the *pattern*. More precisely, we want to find in the text the start index of the first substring that matches the pattern, if it exists.

**Algorithms**

There are several algorithms designed to find the location where one or several strings (patterns) are found within a larger string or text. A summary of the most famous methods for string matching is given in the following:

1. Brute Force (BF) Matching using Hamming Distance:

This method uses Brute–Force sliding window matching

(see document [BF String Matching.pdf](http://www1.aucegypt.edu/faculty/cse/goneid/csce1101/BFStrMat.pdf))

It can be used to detect approximate matches and to find suggested closest matchings. This method has the highest complexity among other algorithms.

The definition of closest follows the Hamming-distance concept. In this method, the Hamming Distance between two strings follows the algorithm below:

***For two strings X, Y, where length(X)| ≤ length (Y)***

***A match occurs at position k if A(k) = B(k).***

***M = Number of matches***

***Distance = Number of mismatches D = length(Y)) – M***

***D = 0 if X and Y are identical***

1. The Rabin-Karp Algorithm

This is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text.

1. The KMP Algorithm

This is a string-searching algorithm created by Donald Knuth, James H. Morris, and Vaughan Pratt (1977). The algorithm searches for occurrences of a pattern string within a main "text string" by employing the observation that when a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

1. The Boyer–Moore Algorithm

This is an efficient string-searching algorithm that is the standard benchmark for practical string-search. It was developed by Robert S. Boyer and J Strother Moore in 1977. The algorithm preprocesses the string being searched for (the pattern), but not the string being searched in (the text). It is thus well-suited for applications in which the pattern is much shorter than the text or where it persists across multiple searches. The Boyer–Moore algorithm uses information gathered during the preprocess step to skip sections of the text, resulting in a lower constant factor than many other string search algorithms. In general, the algorithm runs faster as the pattern length increases. The key features of the algorithm are to match on the tail of the pattern rather than the head, and to skip along the text in jumps of multiple characters rather than searching every single character in the text.

**The Project:**

Your project is to build a **Simple Plagiarism Detection Utility** using string matching algorithms.

You are required to:

* Build a collection (corpus) of existing documents and a potentially plagiarized document (the choice and number of corpus documents, the size of the document and the potentially plagiarized document are left to your discretion).
* Implement the following basic detectors:

1. Approximate string matching using Brute Force and Hamming Distance
2. Rabin–Karp algorithm

* Using the corpus, find the documents from which the potential document was plagiarized. Given a test file, treat each sentence in the file as a potential pattern. Using the above algorithms, search for the pattern in the existing documents and find the matches.
* Compare the performance of the implemented algorithms (measures of performance are left to your discretion).

**Submission Instructions**

Your source code and executable binary file(s) should be submitted on blackboard by the due date, along with a README file that describes how to use the utility.

**Grading**

The program will be tested against a sample potentially plagiarized document. If your program works, it should be capable of detecting plagiarism. In this case, grading will be on both the program and the project report.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### **CSCE 1101-03,04 Spring 2023**

### **Term Project (3)**

##### Dr. Amr Goneid *Due: Monday May 22, 2023*

**Simulation of Waiting Queues of Airplanes in an Airport**

**Introduction**

In this project, you are to develop a program to simulate the queueing operations of an airport. The objective is to estimate the average time an airplane has to wait in the landing queue (airborne) before it can land on a runway. For this purpose, you are to develop a Double-ended Queue data structure and use it to build the simulation program.

**Part 1: The *DEQ* ADT**

We recall that the ***Stack*** and ***Queue*** ADT’s are sequential containers. For the ***Stack***, both adding and removing an element occur at the same end (Top), while for the ***Queue***, adding an element occurs at one end (Rear) and removal occurs at the other end (Front). As an ADT, the Double-Ended Queue (***DEQ***) is also a sequential container that may function like a Queue or a Stack at both ends. Therefore, it can be used either as a Stack or as a Queue.



**Add**

**Add**

**Remove**

**Remove**

The ***DEQ*** ADT can be implemented using a Simple Linked List (***SLL***) with the following member functions:

* **Costructor**: Construct an empty DEQ
* **Destructor**: Destroy DEQ
* **DEQisEmpty**: Test if DEQ is empty
* **Add\_Front**: Add an element at the front
* **Add\_Rear**: Add an element at the rear
* **Remove\_Front**: Remove the element at the front
* **Remove\_Rear**: Remove the element at the rear
* **View\_Front**: Retrieve the front element without removal
* **View\_Rear**: Retrieve the rear element without removal
* **DEQ\_Length**: Number of elements in the DEQ

**Required Implementation:**

Design and implement a template class ***DEQ*** using a SLL with a minimum of the above member functions.

**Part 2: Simulation of Waiting Queues of Airlanes in an Airport**

A certain airport has **a single** landing runway. When planes arrive near the airport, they will have to join **a queue**. A plane arriving near the airport at a random time ***Tarrival***, will be instructed to join the queue and it might have to wait (remain airborne) in that landing queue a time ***Twait*** until the runway becomes free and ready to receive it.

Once a plane lands on the beginning of a runway, that runway becomes occupied for a fixed time ***Tlanding*** until the plane docks (This is the service time).

Use the ***DEQ*** template class you implemented in Part 1 above to develop a program to simulate the airport queue operations with the objective of computing the average wait time in the landing queue.

Assume the following:

* The time (clock) unit is one minute
* A fixed simulation period ***Tmax***
* A fixed time ***Tlanding*** to complete landing (This is the service time).
* A random arrival time ***Tarrival*** with a fixed average inter-arrival time ***ΔT***
* No plane will leave the queue until it lands.

You might start your simulation using a “standard run” with:

***Tmax*** = 6 hours, ***Tlanding*** = 10 minutes, ***ΔT*** = 6 minutes

After that, you might investigate the effect of varying inter-arrival time ***ΔT*** simulate prime and slack times of the day, or if the amount of time to complete landing ***Tlanding*** is changed.

See the document: [QueueSim.pdf](http://www1.aucegypt.edu/faculty/cse/goneid/csce1101/QueueSim.pdf) to guide you through the design of the program.

Allow your program to produce a “log” of the events of arrival and landing in each run (see a simple example of a “log” in the above document).

**Note on using the C++ Random Number Generator (RNG):**

Many C++ programs use random numbers generated by a Random Number Generator (RNG). The RNG in C++ is a function **rand( )** that returns a random integer from 0 to 32,767 with equal probability.

To obtain random floating point numbers 0 ≤ R < 1.0 with equal probability, use

**float R = rand( ) / float(32767);**

To obtain random integers from 1 through n, use

**int r = rand( ) % n + 1.**

Generally, you may implement a function ***RandInt (i, j)*** that generates an integer between *i* and *j* with equal probability. This is implemented simply as follows:

***int RandInt (int i , int j)***

***{***

***return rand( ) % (j-i+1) + i ;***

***}***

To obtain a random sequence you need to first initialise the RNG using the time of the machine as a seed. This is done so that we do not get the same sequence every time we run the program.

The following is an **example** of how to generate a random sequence of pairs of integers with values between 1 and n:

**#include <time.h>**

**int x , y, n ;**

**…………….**

**srand ( (unsigned) time (NULL) ); //Initialize RNG**

**repeat as needed // Loop over the sequence**

**{**

**x = *RandInt (1 , n)*; // Generate 1st number**

**y = *RandInt (1 , n)*; // Generate 2nd number**

**……………….. // Do something with x and y**

**}**

**One Method to build your own RNG**

You may use a random congruence method to code your own RNG. Here is one algorithm to generate a random sequence of Large Integers:

With ***x0 = some large int value*** representing the seed, then a random ***large integer*** sequence is obtained as:

***xi+1 = (α xi + β) % m***

***α = 25173 β = 13849 m = 65536***

The values of x will be between 0 and 65535. You may divide it by m to obtain a random sequence of 0 ≤ R < 1.0 with equal probability.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSCE 1101-03,04 Spring 2023 Term Project Report**

**Project Title**

Group Name/s

Department of Computer Science and Engineering, AUC

***Abstract:***

*This is just a suggested format. You may change or modify it in the way that suits the project work*

**…………………………………………………………………………………………………………………………………………………………………**

*Keywords:*

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Introduction**

**(Introduce the topic of the project here)**

**……………………………………………………………………………………………………………**

1. **Problem Definition**

**(Present here a more detailed definition of the topic or problem)**

**……………………………………………………………………………………………………………**

1. **Methodology**

**(Outline the methodology chosen to solve the problem)**

**……………………………………………………………………………………………………………**

1. **Specification of Algorithms to be used**

**……………………………………………………………………………………………………………**

1. **Data Specifications**

**(Specify the input data to be used in your work)**

**……………………………………………………………………………………………………………**

1. **Experimental Results**

**(Give your results of processing the input data in text, tabular or graphical forms) ……………………………………………………………………………………………………………**

1. **Analysis and Critique**

**(Your own analysis and critique of the output results and the methodology/algorithms used)**

**……………………………………………………………………………………………………………**

1. **Conclusions**

**……………………………………………………………………………………………………………**

**Acknowledgements**

**(Acknowledge other people who helped you in producing the project)**

**…………………………….**

**References**

**(List here the references you used to produce the project)**

**…………..………………………………………….**

**Appendix: Listing of all Implementation Codes**

**…………..**

**…………..**