RAMAIAH UNIVERSITY OF APPLIED SCIENCES

# ASSIGNMENT

| | |
|---|---|
| **Course Code** | CSC201A |
| **Course Name** | Discrete Mathematics - I |
| **Programme** | B.Tech |
| **Department** | CSE |
| **Faculty** | FET |

| | |
|---|---|
| **Name of the Student** | Satyajit Ghana |
| **Reg. No** | 17ETCS002159 |
| **Semester/Year** | 03/2018 |
| **Course Leader/s** | Ms. Sahana |

| Declaration Sheet | | | |
|---|---|---|---|
| Student Name | Satyajit Ghana | | |
| Reg. No | 17ETCS002159 | | |
| Programme | B.Tech | Semester/Year | 03/2018 |
| Course Code | CSC201A | | |
| Course Title | Discrete Mathematics - I | | |
| Course Date | | to | |
| Course Leader | Ms. Sahana | | |

**Declaration**

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

| | | | |
|---|---|---|---|
| Signature of the Student | | Date | |
| Submission date stamp (by Examination & Assessment Section) | | | |

| Signature of the Course Leader and date | Signature of the Reviewer and date |
|---|---|
| | |

<u>**Question No. 1**</u>

## <u>Solution to Question No. 1 Part A:</u>

### A 1.1 Timed Logics and their CSE Applications

Temporal logic is a branch of symbolic logic which is concerned with problems on propositions that have truth values dependent on time. Temporal logic is considered a variant of modal logic, which is a branch of logic dealing with propositions that can be expressed as a set of possible worlds. Temporal logic is used to touch all approaches to reasoning and representation based on time.

Non-determinism is an important issue in computer science applications, and hence much use has been made of branching time models. Two important such systems are CTL (Computation Tree Logic) and a more expressive system CTL*; these correspond very nearly to the Ockhamist and Peircean semantics discussed above.

Further branches of CSE that make use of Temporal Logic are in Natural Language Processing and Artificial Intelligence,

**Prior (1967)** lists amongst the precursors of Tense Logic **Hans Reichenbach's (1947)** analysis of the tenses of English, according to which the function of each tense is to specify the temporal relationships amongst a set of three times related to the utterance, namely S, the speech time, R, the reference time, and E, the event time. In this way Reichenbach was neatly able to distinguish between the simple past “I saw John”, for which $R = E < S$, and the present perfect “I have seen John”, for which $E < R = S$, the former statement referring to a past time coincident with the event of my seeing John, the latter referring to the present time, relative to which my seeing John is past.

Much of the work on temporal reasoning in AI has been closely tied up with the notorious frame problem, which arises from the necessity for any automated reasoner to know, or be able to deduce, not only those properties of the world which do change as the result of any event or action, but also those properties which do not change. In everyday life, we normally handle such facts fluently without consciously adverting to them: we take for granted without thinking about it, for example, that the colour of a car does not normally change when one changes gear. The frame problem is concerned with how to formalise the logic of actions and events in such a way that indefinitely many inferences of this kind are made available without our having to encode them all explicitly.

Likewise, to ensure correctness of a concurrent program, performed by two or more processors working in parallel, it is necessary to formally specify and verify their synchronization, i.e., the way in which the actions of the various processors are interrelated, and to formally describe the acceptable infinite executions of the program. The relative timing of the actions must be carefully co-ordinated so as to ensure that integrity of the information shared amongst the processors is maintained.

Key temporal patterns of importance in specifying such programs are:

*"liveness"* properties, or *"eventualities"*, involving temporal patterns $Fp, q \rightarrow Fp$, or $G(q \rightarrow Fp)$, which ensure that if a specific precondition (q) is initially satisfied, then a desirable state (satisfying p) will eventually be reached in the course of the computation. For example, such are "If a message is sent, eventually it will be delivered" and "Whenever a printing job is activated, eventually it will be completed".

*"safety"* or *"invariance"* properties, involving temporal patterns $Gp, q \rightarrow Gp$, or $G(q \rightarrow Gp)$, which ensure that if a specific precondition (q) is initially satisfied, then undesirable states (violating the safety condition p) will never occur. Examples are: "No more than one process will be in its critical section at any moment of time", "A resource will never be used by two or more processes simultaneously", or, for more practical examples: "The traffic lights will never show green in both directions", "A train will never pass a red semaphore".

*"fairness"* properties, involving combinations of temporal patterns of the form $GFp$ ("infinitely often p") or $FGp$ ("eventually always p"). Intuitively, fairness requires that in a system where several processes sharing resources are run concurrently, they must be treated 'fairly' by the operating system, scheduler, etc. A typical fairness requirement says that if a process is persistent enough in sending a request (e.g., keeps sending it over and over again) then eventually its request will be granted. **(Goranko, Valentin and Galton, Antony, "Temporal Logic", 1999)**

Another application of temporal logic is in databases in which a temporal database is generally understood as a database capable of supporting storage and reasoning of time-based data. For example, medical applications may be able to benefit from temporal database support a record of a patient's medical history has little value unless the test results, e.g. the temperatures, are associated to the times at which they are valid, since we may wish to do reasoning about the periods in time in which the patient's temperature changed.

Temporal databases store temporal data, i.e. data that is time dependent (timevarying). Typical temporal database scenarios and applications include time-dependent/time-varying economic data, such as Share prices, Exchange rates, interest rates and Company profits.

The desire to model such data means that we need to store not only the respective value but also an associated date or a time period for which the value is valid. Typical queries expressed informally might include:

- Give me last month's history of the Dollar-Pound Sterling exchange rate.
- Give me the share prices of the NYSE on October 17, 1996.

<u>**Question No. 2**</u>

## Solution to Question No. 1 Part B:

**B 1.1 Translate a set of at-least five functional requirements into formal specifications using propositional logic:**

The given Functional Requirements for the Thesis: Design and Implementation of an Intrusion Tolerant Web Server are:

1. The design must withstand attacks
2. The design must have low latency and high availability
3. The solution must tolerate attacks and serve users continuously
4. The solution should withstand internal attacks
5. The IP lists should block compromised clients and free them from the list after time limit

$p : the\ design\ must\ withstand\ attacks$

$q :\ the\ design\ must\ have\ low\ latency$

$r :\ the\ design\ must\ have\ high\ availability$

$s :\ the\ solution\ must\ tolerate\ attacks$

$t :\ the\ solution\ must\ serve\ users\ continuously$

$u : the\ solution\ should\ withstand\ internal\ attacks$

$v : the\ ip\ list\ should\ block\ compromised\ clients$

$w : the\ ip\ list\ should\ free\ the\ compromised\ clients\ from\ the\ list\ after\ time\ limit$

**B 1.2 Use Propositional Logic to verify the consistency of the identified specifications:**

$$p$$

$$q \wedge r$$

$$s \wedge t$$

$$u$$

$$v \wedge w$$

The System is consistent for the following conditions:

| $p$ | $q$ | $r$ | $s$ | $t$ | $u$ | $v$ | $w$ | $q \wedge r$ | $s \wedge t$ | $v \wedge w$ | $res$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**B 1.3 Interpret the result:**

To create an intrusion tolerant web-server the design must withstand attacks and must have low latency and high availability and it must tolerate attacks and serve users continuously and it must withstand internal attacks and ip list should block compromised clients and free the compromised clients from the list after time limit.

The result is consistent, since there is a combination for which all of the propositions are true and the output hence is true.

## Solution to Question No. 2 Part B:

**B 2.1 Develop a program that takes the set of given DNA strings as input and generate a lexicographically ordered list of all the r-subsets:**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <string>

std::vector<std::string> data; // = {"GAACTG", "CGCGTT", "AGATCC", "CTCCCC", "AGAATG",
"AGTGTC"};

void combination(int n, int r) {
    std::string bitmask(r, 1);
    bitmask.resize(n, 0);

    /* while there is a permutation generate the bit-string and print the permutation */
    do {
        for (int i = 0; i < n; i++)
            if (bitmask[i]) std::cout << " {"<< data[i] << "} ";
        std::cout << std::endl;
    } while (std::prev_permutation(bitmask.begin(), bitmask.end()));
}

int main() {
    std::cout << "----------- n-Combination-r Generator --------------------------------
------\n" << std::endl;
    std::cout << "Enter the data elements : " << std::endl;
    std::string input;
    while (std::getline(std::cin, input)) {
        if (input.size() > 0)
            data.push_back(input);
        else
            break;
    }
    std::cin.clear();
    /* Sort them lexicographically */
    std::sort(data.begin(), data.end());
    std::cout << "Data in the Set :  " << std::endl;
    std::copy(data.begin(), data.end(), std::ostream_iterator<std::string>(std::cout, "
"));
    std::cout << std::endl;
    int r = 3;
    std::cout << "Enter the value of r : ";
    std::cin >> r;
    combination(data.size(), r);
    return 0;
}

/* Source Code for prev_permutation */
/*bool prev_permutation(BidirIt first, BidirIt last)
{
    if (first == last) return false;
    BidirIt i = last;
    if (first == --i) return false;

    while (1) {
        BidirIt i1, i2;

        i1 = i;
        if (*i1 < *--i) {
            i2 = last;
            while (!(*--i2 < *i))
                ;
            std::iter_swap(i, i2);
            std::reverse(i1, last);
            return true;
        }
        if (i == first) {
            std::reverse(first, last);
            return false;
        }
    }
}*/
```

**OUTPUT:**

```
----------- n-Combination-r Generator -----------------------------------

Enter the data elements :
GAACTG
CGCGTT
AGATCC
CTCCCC
AGAATG
AGTGTC

Data in the Set :
AGAATG AGATCC AGTGTC CGCGTT CTCCCC GAACTG
Enter the value of r : 3
 {AGAATG}  {AGATCC}  {AGTGTC}
 {AGAATG}  {AGATCC}  {CGCGTT}
 {AGAATG}  {AGATCC}  {CTCCCC}
 {AGAATG}  {AGATCC}  {GAACTG}
 {AGAATG}  {AGTGTC}  {CGCGTT}
 {AGAATG}  {AGTGTC}  {CTCCCC}
 {AGAATG}  {AGTGTC}  {GAACTG}
 {AGAATG}  {CGCGTT}  {CTCCCC}
 {AGAATG}  {CGCGTT}  {GAACTG}
 {AGAATG}  {CTCCCC}  {GAACTG}
 {AGATCC}  {AGTGTC}  {CGCGTT}
 {AGATCC}  {AGTGTC}  {CTCCCC}
 {AGATCC}  {AGTGTC}  {GAACTG}
 {AGATCC}  {CGCGTT}  {CTCCCC}
 {AGATCC}  {CGCGTT}  {GAACTG}
 {AGATCC}  {CTCCCC}  {GAACTG}
 {AGTGTC}  {CGCGTT}  {CTCCCC}
 {AGTGTC}  {CGCGTT}  {GAACTG}
 {AGTGTC}  {CTCCCC}  {GAACTG}
 {CGCGTT}  {CTCCCC}  {GAACTG}
```

**B 2.2 Determine the time and space complexities of the program:**

`prev_permuations` uses a linear method to calculate the permutation, we first create a bit string where 1 represents that the element is present in the set and 0 denotes that it is not, this bit string is then permuted to obtain the permutation, from this the combination of the actual set is found and displayed. The Algorithm being

1. Search back from the end, find the first greater character compared to the one immediately after it.
2. Search back from the end again, find the first smaller one compared to the great character just found.
3. Swap the 2 characters found from above 2 steps.
4. Reverse the rest string after the character location found in Step 1.

Looking at the source code the time complexity depends upon the number of swaps that are taking place which by definition is at most $(finish - start)/2$ , hence the time complexity is linear, the space complexity depends upon the size of the input and does not change, hence it's constant, since we used a vector it's variable size and hence we can consider the space complexity of the overall program to be n.

$$Time\ Complexity : O(n)$$

In the worst case, only two scans of the whole array is needed

$$Space\ Complexity\ (Algorithm) : O(1)$$

Since no extra space is used and the replacements/swaps are done in-place

$$Space\ Complexity\ (Program) : O(n)$$

## Bibliography

1. Pnueli, A., 1977, “The temporal logic of programs”, Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pages 46-67.
2. Prior, A. N., 1957, *Time and Modality*, Oxford: Clarendon Press.
3. https://www.cs.uct.ac.za/mit_notes/database/htmls/chp18.html#temporal-databases-the-complexities-of-time
4. Goranko, Valentin and Galton, Antony, "Temporal Logic", Stanford Encyclopedia, Temporal Logic, First published Nov 29, 1999
   https://plato.stanford.edu/entries/logic-temporal/