Brian Ton

Exam 2 -- Discrete Structures

Essay 1: Turing's Halting Problem

Prior to any discussion of the Halting Problem should start with the idea of a decision problem. Intuitively, a decision problem is a type of problem that asks a "yes or no" question on some given set of inputs. For instance, some examples of decision problems include "are all odd integers prime?", "is this integer prime?", and "is this system of boolean expressions satisfiable?" This idea of a decision problem then leads to the problem posed by Hilbert the (Entscheidungsproblem, itself a decision problem) that Turing seeked to answer: *Is there a procedure (an algorithm) that given a set of axioms and a statement that can decide if the statement is true or false?* Turing approached this problem computationally, using the concept of what would be later known as a *Turing Machine* to arrive at an answer to this problem. Turing would then rephrase this idea into a question of whether or not a program exists such that it can determine whether or not another program will terminate given a certain input. Using a proof by contradiction, Turing first assumed that the halting problem was solvable (and thus that there did exist such an algorithm) and then by following its logical consequences, derived a contradiction that showed that the halting problem was unsolvable. Since Turing showed that there does exist a problem that cannot be solved using a procedure, Turing thus showed that the Entscheidungsproblem had a negative answer. Turing's solution to the halting problem has many implications. For instance, using a technique known as a proof by reduction, one can show that the state entry problem (also known as the "dead code problem"), which asks if one can determine whether or not a Turing machine enters a certain state, requires a solution to the

Halting problem and is thus unsolvable. More practically, one can reduce the problem of determining whether or not a program runs malicious code into a state entry problem, which the Halting problem shows is unsolvable. In sum, the Halting Problem, as shown by Turing is unsolvable, which has implications in many areas of computing.

Essay 2: Turing's Halting Problem (Rigorous)

In order to present a rigorous proof to the halting problem, one must first define the idea of a Turing machine. However, in doing so, one must also define several key ideas. Firstly, an alphabet is defined as a set of elements that can be used to form strings. A Turing machine T can then be defined as $T=(S, I, f, s_0)$, a four-tuple consisting of a finite set S of states, an alphabet I containing the blank symbol B, a partial function f from $S \times I$ to $S \times I \times \{R, L\}$, and a starting state s_0. Somewhat moving away from the formal definition, a Turing Machine can be thought of as a control unit with an infinitely long tape (that has only a finite number of nonblank symbols at a time). At each step, the control unit reads the current tape symbol x. If the control unit is in state s and if the partial function f is defined for the pair (s, x) where $f(s, x) = (s', x', d)$, then the control unit does the corresponding actions. Here, the control unit first enters the state s', then writes the symbol x' into the current cell, and moves right or left one cell depending on the value of d (if $d=R$, then it moves right, and if $d=L$ then it moves left). If the partial function f is not defined for the pair (s, x), then the machine will halt. Additionally, a decision problem (also known as a *yes-or-no problem*) is formally defined as a problem that asks whether statements from a particular class of statements are true. The halting problem, then, can be defined in terms of a Turing machine T as the decision problem that asks whether a Turing machine T eventually

halts when given an input string *x*. Theorem 1 of the text then states that the halting problem is an unsolvable decision problem. That is, no Turing machine exists that, when given an encoding of a Turing machine *T* and its input string *x* as input, can determine whether *T* eventually halts when started with *x* is written on its tape. Like many other proofs regarding specific properties of something (like the irrationality of $\sqrt{2}$), a proof by contradiction can be used to prove Theorem 1. At a high level, this proof is done by assuming that a Turing machine exists that can be used to determine if another Turing machine halts, then deriving a contradiction. In sum, after defining the halting problem in terms of a Turing machine, using contradiction, one can show that the halting problem is unsolvable.

Essay 3: Recursive Algorithms

An algorithm is defined as a finite sequence of instructions for performing a computation or solving a problem. Algorithms typically share several properties. Firstly, an algorithm has *input* values from a specified set. Secondly, from each set of input values, an algorithm produces *output* values from a specified set. The output values are the solution to the problem. Thirdly, the steps of an algorithm must be *defined precisely*. Fourthly, an algorithm should produce the desired output after a *finite* number of steps for any input in the set. Fifthly, it must be possible to perform each step of an algorithm exactly an in a finite amount of time (typically referred to as *effectiveness*). Lastly, the algorithm should be applicable for all problems of the desired form, not just for a particular set of input values (typically referred to as *generality*). An algorithm must also be correct, meaning that the algorithm produces the correct output values for each set of input values. Perhaps the most famous algorithm is so-called Bubble Sort, which sorts a

general array of elements. One particular type of algorithm is the recursive algorithm. A recursive algorithm is defined as an algorithm that solves a problem by reducing it to an instance of the same problem with a smaller input. A natural example of a recursive algorithm would be the factorial, as computing $n!$ is the same as $n \cdot (n-1)!$ which is the same as $n \cdot (n-1) \cdot (n-2)!$ and so on. Another famous example of a recursive algorithm would be merge sort, which splits a given list into two sublists, sorts them individually using merge sort, and then merges them together (hence the name merge sort). Due to its similar nature to induction, recursive algorithms are often proven correct using mathematical and strong induction. An example of this proof is shown here.

ALGORITHM - Recursive Exponentiation With Integer Power.

procedure *power(b, n: $b \in \mathbb{R}$, $n \in \mathbb{Z}$, $n \geq 0$)*
if n == 0 then
  return 1
else if *n is even* **then**
  return *power(b, $\frac{n}{2}$)^2*
else
  return *$b \cdot$ power(base, $\lfloor \frac{n}{2} \rfloor$)^2*
{output is b^n}

Proof of Correctness (using Strong Induction)

Base Case: When $n = 0$, the procedure outputs 1. This is trivially correct since $b^0=1$ for all b. The base case is thus correct.

Inductive Hypothesis: Assume that *power(b, m)* $= b^m$ for all integers $0 \leq m < k$ for a fixed integer k where b is a positive integer.

Inductive Step: Since the algorithm handles odd and even values of k differently, the inductive step must be split into two cases.

1. When k is even, *power(b, k) = power(b, $\frac{k}{2}$)^2* $= (b^{k/2})^2 = b^k$, where the inductive hypothesis was used to substitute $b^{k/2}$ for *power(b, $\frac{k}{2}$)*. Hence, the algorithm is correct when k is even.
2. When k is odd, *power(b, k) = b $\cdot$ power(b, $\lfloor \frac{k}{2} \rfloor$)^2* $= b \cdot (b^{\lfloor k/2 \rfloor})^2 = b^{2\lfloor k/2 \rfloor + 1}$, where the inductive hypothesis was used to substitute $b^{\lfloor k/2 \rfloor}$ for *power(b, $\lfloor \frac{k}{2} \rfloor$)*. Here, note that since k is odd, it can be written as $k=2a+1$ for some integer a. It follows that $2\lfloor \frac{k}{2} \rfloor + 1 = 2\lfloor \frac{2a+1}{2} \rfloor + 1 = 2\lfloor \frac{2a}{2} + \frac{1}{2} \rfloor + 1 = 2\lfloor a + \frac{1}{2} \rfloor + 1$. Since a is an integer,

$2\lfloor a+\frac{1}{2}\rfloor+1=2a+1$ (because the floor of a real number x is the greatest integer less than or equal to x). However, note that $2a+1=k$. Hence, when k is odd, $2\lfloor \frac{k}{2}\rfloor+1 = k$. Thus, $b^{2\lfloor k/2\rfloor+1}=b^k$, which is correct.

This thus completes the inductive step. Therefore, by strong induction, the given algorithm is correct (i.e. *power(b, n)* correctly computes b^n for all positive integers n and real numbers b).

Essay 4: Program Correctness

Program verification uses the rules of inference and proof techniques in order to show a program's correctness. A program is defined to be correct if it produces a correct output for every possible input. A proof of correctness is made up of two parts. Firstly, it must be shown that the correct answer is obtained if the program terminates. This step also establishes what is known as partial correctness of the program. Secondly, in a proof of correctness, it must be shown that the program always terminates. To specify what it means to have a correct output, two propositions are defined. The initial assertion describes the properties that the input values must have, and the final assertion gives the properties that the output of the program should have, if the program did what was intended. Programs are often split into a sequence of subprograms, which when using the rules of inference, can show whether or not the entire program is correct. Additionally, rules of inference are also be used to verify programs where conditional statements occur. Loop invariants are used to prove correctness of programs involving while loops. A loop invariant is a condition that is true during every traversal of a loop. This loop invariant leads to a rule of inference, which allows one to prove correctness of an entire program. A typical proof of a loop invariant has three parts. Firstly, it must be shown that the loop invariant is true before execution of the loop. Secondly, it must be shown that the loop invariant is true during the execution of the loop. Finally, it must be shown that the loop invariant is true after execution of

the loop. In sum, techniques used to prove program correctness are somewhat similar to techniques we have seen before in class, especially with logical inferences and loop invariance.

Essay 5: SQL

An *n-ary relation* on sets $A_1, A_2 ... A_n$ is defined as a subset of $A_1 \times A_2 ... \times A_n$ .The sets $A_1, A_2 ... A_n$ are considered the domains of the relation, and *n* is considered its degree. One of the most widely used applications of an n-ary relation is a relational database. Such a database is made up of records, which are n-tuples where each element in the tuple is a part of a field. These databases are also often called tables, as they are often displayed as tables. The domain of a relation is considered a primary key when it can be used to identify an n-tuple (i.e. it is unique). In a database of student records, an example of such a primary key would be the student ID number. A composite key is a collection of values that can determine an n-tuple in a relation (i.e. are unique). For a database of people in the United States, a composite key could be the combination of name and address (assuming that there is not a household where people share the exact same name). Certain operations can be done on n-ary relations. Firstly, there is a selection operator s_c that selects all the n-tuples from a relation that satisfy a certain condition. Secondly, there is a projection operator that can remove components of an n-tuple. Thirdly, there is the join operator, which joins relations to create a new relation by combining all tuples of the original relations together (where the two tuples have agreeing values). SQL (or Structured Query Language) is a computer language that allows for the creation and management of relational databases. The operators on n-ary relations above can also be implemented in SQL. For instance, an example SQL query would look like:

SELECT Major *FROM* Enrollment *WHERE* name=’Brian’

Here, the use of *SELECT* indicates a projection operator, where SQL is chooses certain components of the n-tuples from the Enrollment database. The *WHERE* clause specifies the selection operation, in this case choosing the entry with the name Brian. Additionally, the join operation is implemented through a *JOIN* clause in the query. Some other SQL operations include *INSERT*, which allows for the insertion of new tuples into the database, and *DROP* which completely erases a database. In class, a student mentioned that in his experience with SQL, he encountered a case where even though two queries achieved equivalent results, one was much slower than the other. This result comes from how SQL is implemented (in fact, one SQL implementation can differ greatly from another). Consider a search for an employee by his / her number (note that this will be the primary key in this instance). There are three ways that an SQL implementation can find such an employee. Firstly, if the implementation hashes the primary key and uses a hash table lookup, the time complexity would be O(1). Secondly, if the implementation does a linear search, the time complexity would be O(n). Finally, if the implementation does a binary search, the time complexity would be O(log n). When the student changed his query, what he did was better leverage how the SQL implementation worked. In sum, *n-ary* relations have many useful characteristics, leading to its most popular application, relational databases, typically written in SQL.