Prof. Sergio A. Alvarez
Maloney Hall, room 569
Computer Science Department
Boston College
Chestnut Hill, MA 02467 USA

http://www.cs.bc.edu/∼alvarez/
alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS243, Logic and Computation
Propositional Logic

1 Propositions

A *propositional statement* or *proposition* is a statement that is unambiguously either **true** or **false** in a given context. Examples include: *George Washington was born in Arizona* (**false**), $2 + 3$ *equals* 5 (**true**), and *Some humans are reptiles* (**false**). The truth or falsity of a given proposition is called its *truth value*. Statements that involve a subjective judgment, such as *Bob is a good guy*, are not propositional statements.

A well-formed boolean expression in a programming language such as Python is a concrete embodiment of a proposition. The following example uses an equality test to construct a boolean expression:

```
>>> for i in range(0,10):
...     if i%3 == 1:
...             print i
...
1
4
7
```

As you know from Python, propositions can also be built using boolean operators (e.g., `p and q`, where `p` and `q` are boolean variables). The truth value of such a proposition depends on the truth values of the symbols (boolean variables) that appear in it, and on the meanings of the particular operators used to combine these symbols. The most commonly used operators are `not, and, or`, with the usual meanings ("truth tables") described below using list comprehension notation in Python.

```
>>> [not x for x in [False,True]]
[True, False]
>>> [x and y for x in [False,True] for y in [False,True]]
[False, False, False, True]
>>> [x or y for x in [False,True] for y in [False,True]]
[False, True, True, True]
```

We'll describe propositions more formally below, in terms of a language with well-defined *syntax* (form, grammar) and *semantics* (meaning, truth). A recurring theme in this course is that formal descriptions provide transparency and allow *proving* assertions beyond a shadow of a doubt. The use of propositional logic for modeling purposes will also be discussed.

1.1 Syntax: recursive definition of proposition

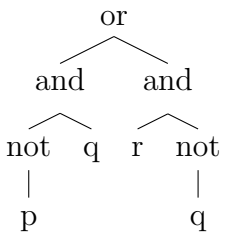
Start with a finite collection P of *propositional symbols* (boolean variables) (typically `p, q, r`, and so forth). We will define the set of propositions over P (also known as *propositional formulas* over P) as follows. Note that the precise ordering of symbols will be important, and that parentheses are explicitly required as indicated in each case.

1. (Basis) The *basic propositions* (also known as *atomic formulas*) over P are of the following two types:
   (a) The *constant symbols* **true** and **false** are (basic) propositions over P.
   (b) Every propositional symbol (variable) in P is a (basic) proposition over P.
2. (Recursion) If `p` and `q` are propositions over P (not necessarily variables), then so is each of the following: `(not p)`, `(p and q)`, `(p or q)`.
3. (Closure) Every proposition over P can be obtained from the basic propositions defined above by a finite number of applications of the recursive rules.

Parse trees. The syntactical structure of a proposition can be understood by drawing its *parse tree*, which is a tree that has the boolean operators as internal nodes, and propositional symbols as leaves. The root of the parse tree is the outermost operator, that is to say, the last operator to be applied in constructing the propositional formula. For example, the proposition below

```
((not p) and q) or (r and (not q))
```

has the following parse tree:



1.2 Semantics: truth assignments and evaluation

Truth assignments. Let P be a set of propositional symbols. A *truth assignment* over P is a function $t : P \to$ **false, true** that assigns a truth value to each symbol in P. For example, consider the following propositions:

```
p: The patient is over 50 years old.
q: The patient has a history of heart problems.
```

These propositions could be either **true** or **false** depending on the patient. Each of the possibilities can be represented by a truth assignment to the symbols `p` and `q`. For example, for a 20-year old patient with no prior heart problems, one would consider the truth assignment `t(p) = false`, `t(q) = false`.

Semantics of boolean operators: truth tables. A k-ary boolean operator, that is, an operator that takes k arguments, is just a function $f : \{\textbf{false, true}\}^k \to \{\textbf{false, true}\}$. Specifically, such an operator is uniquely determined by the truth values that it assigns to all of the possible combinations of **false, true** for its arguments. This truth function that represents the operator is also known as the *truth table* of the operator, in reference to a table listing with one combination of input values and corresponding truth value per row. The truth tables for not, and, or appear below.

```
>>> [not x for x in [False,True]]
[True, False]
>>> [x and y for x in [False,True] for y in [False,True]]
[False, False, False, True]
>>> [x or y for x in [False,True] for y in [False,True]]
[False, True, True, True]
```

Truth value of a complex proposition: recursive definition. Suppose that t is a truth assignment to a set P of propositional symbols. The truth value with respect to t of any propositional statement over P is then defined as follows.

1. (Basis) The truth value of each *basic proposition* is as given directly by t.
2. (Recursion) If `p` and `q` are propositions over P with truth values `t(p), t(q)`, then `t(not p) = notf(t(p))`, `t(p and q) = andf(t(p),t(q))`; `t(p or q) = ort(t(p),t(q))`, where `opf` in each case is the truth table for the operator `op`.

For example, consider a patient for which `t(p)=false, t(q)=false` above, and consider the following proposition:

```
z: (p or (not q))
```

The recursive extension of the truth assignment simply states that the corresponding truth value for `z` is obtained by "plugging in" the truth values for `t` into the definition of `z`:

```
t(z) = orf(t(p), notf(t(q))) = orf(false, notf(false)) = orf(false,true) = true
```

Boolean satisfiability. A propositional formula is *satisfiable* if there is some truth assignment that makes its truth value **true**. A propositional formula is a *tautology* if every truth assignment makes its truth value **true**.

Equivalent formulas. Two propositional formulas F and G are said to be *equivalent* if they have the same truth table. In other words, F and G are equivalent if (and only if) they have the same truth value $t(F) = t(G)$ for each truth assignment t of the set of propositional variables that appear in F and G. An equivalent statement is that F and G are equivalent formulas if, and only if, the formula $F \leftrightarrow G$ is a tautology.

1.3 Alternative notation for boolean operators

Negation. `not p` is often written as $\neg p$ or as $\sim p$ or as $\overline{p}$.

Conjunction (and). `(p and q)` is often written $(p \wedge q)$ or as (pq). The latter notation is consistent with the fact that ordinary multiplication on the values $0, 1$ as representations of the logical values **false, true** produces the same results as `and`.

Disjunction (or). `(p or q)` is often written $(p \vee q)$ or as $(p + q)$. The latter notation is consistent with the fact that addition on the values $0, 1$ as representations of the logical values **false, true** produces the same results as `or` (all nonzero values are interpreted as being **true**).

Logic gate notation is also used, particularly in the context of digital hardware. An example appears in Fig. 1. The small triangle with a single input and a ball at the output end represents negation, while the two-input gates represent conjunction (flat input end) and disjunction (rounded input end). The circuit's output at the far right is "on" (**true**) because the `and` gate's output is on for the given input values `A=0, B=1, C=0`.

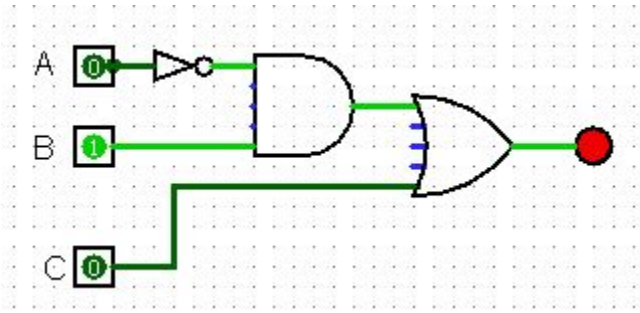


Figure 1: Sample boolean logic circuit. Screenshot from *Logisim*.

1.4 Dropping parentheses: precedence rules.

The strict use of all formally required parentheses gets tedious, and it is therefore customary to relax the rules a bit. Doing so requires agreeing on what is to be done in cases of ambiguity (multiple interpretations of what is intended). For example, consider this propositional formula:

```
not p and q
```

This could easily mean either of the following:

```
((not p) and q)
(not (p and q))
```

The two options are not equivalent. For example, if both `p` and `q` are **false**, then the upper formula is **false**, while the lower formula is **true**. Because of this ambiguity, we need to agree on which of the two options is to be used. The usual convention is that `not` binds more tightly than any binary operator (e.g., `and, or`). Hence, the original formula would be interpreted as the upper option here. By convention, each binary operator associates from left to right. For example, the formula

```
p and not q and r
```

is equivalent to the fully parenthesized version

```
(p and ((not q) and r))
```

1.5 Additional boolean operators

Implication. $(p \to q)$, read "p implies q", is shorthand for `((not p) or q)`.

```
>>> [not p or q for p in [False,True] for q in [False,True]]
[True, True, False, True]
```

Equivalence. $(p \leftrightarrow q)$ is shorthand for $((p \to q)$ and $(q \to p))$.

```
>>> [(not p or q) and (not q or p) for p in [False,True] for q in [False,True]]
[True, False, False, True]
```

Exclusive OR. $p \oplus q$, read "p x-or q", is shorthand for `((p and (not q)) or ((not p) and q))`. This coincides with mod 2 addition on the values $0, 1$.

```
>>> [(p and not q) or (not p and q) for p in [False,True] for q in [False,True]]
[False, True, True, False]
```

1.6 Complete sets of boolean operators.

A set S of boolean operators is *complete* when any truth table can be obtained as the truth table of a formula written using only boolean operators from among those in S. We will give a few examples of complete sets of boolean operators below.

p_1	p_2	p_3	$T(p_1, p_2, p_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 1: Sample truth table. $1, 0$ represent **true**, **false**.

Negation, conjunction, and disjunction. DNF and CNF. We claim that any truth table is the truth table of a formula in which the only boolean operators are $\neg$, $\wedge$, $\vee$. To see this, consider an arbitrary truth table T that corresponds to a formula with k distinct propositional variables $p_1, \cdots, p_k$. Therefore, T will have an entry (truth value) $T(t)$ for each of the 2^k possible truth assignments, t, of $p_1, \cdots p_k$. Each such truth assignment is an ordered sequence $(t(p_1), \cdots, t(p_k))$ of the truth values **true** and **false** assigned to the various p_i by t. We will concoct a formula F in which $\neg$, $\wedge$, $\vee$ are the only operators, such that F has the same truth table, T. To do so, we first form a smaller formula F_t for each row of the truth table, that is, for each truth assignment, t:

$$F_t = \bigwedge_{\{i|t(p_i)=\textbf{true}\}} p_i \ \wedge \bigwedge_{\{i|t(p_i)=\textbf{false}\}} \neg p_i$$

This formula just describes the particular truth assignment, t, by requiring that each propositional variable have the appropriate truth value that is given to it by that truth assignment. The formula that will represent the entirety of T simply combines the formulas F_t of the rows t for which $T(t) = \textbf{true}$:

$$F = \bigvee_{\{t|T(t)=\textbf{true}\}} F_t$$

This completes the proof of the completeness of the set $\{\neg, \wedge, \vee\}$. By the way, any formula like F above that is an OR of ANDs, where each AND portion includes only variables and their negations, is said to be in *disjunctive normal form* (DNF).[1]

For example, consider the truth table for three propositional variables p_1, p_2, p_3 in Table 1.6. The corresponding DNF formula using only negation, conjunction, and disjunction appears below:

$$(\neg p_1 \wedge \neg p_2 \wedge p_3) \vee (\neg p_1 \wedge p_2 \wedge p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge p_3) \vee (\neg p_1 \wedge p_2 \wedge \neg p_3)$$

There is also a *conjunctive normal form* (CNF), which is an AND of ORs, where each OR portion only includes variables and their negations. This can be obtained from the DNF version of the negation of the formula in question. See the Exercises.

[1] Not to be confused with a disappointing outcome in some athletic events.

Implication and negation. We claim that the set $\{\neg, \rightarrow\}$ is a complete set of boolean operators. Since the set $\{\neg, \wedge, \vee\}$ was already shown above to be complete, it is enough to show that each of the three boolean operators $\neg$, $\wedge$, $\vee$ can be represented in terms of the boolean operators $\neg$, $\rightarrow$. We do this below.

1. $\neg$ is automatically representable, since we are including it in the set that we intend to show completeness of.
2. Since $(\neg p) \vee q$ is equivalent to $p \rightarrow q$, we see that $p \vee q$ is equivalent to $(\neg p) \rightarrow q$.
3. Since $\neg(p \wedge q)$ is equivalent to $\neg p \vee \neg q$ by DeMorgan's law, and since $\vee$ can be represented in terms of $\neg$ and $\rightarrow$ as shown above, it follows that there is a formula in terms of $\neg$ and $\rightarrow$ that represents $\wedge$. What is it?

This completes the proof of the completeness of the set $\{\neg, \rightarrow\}$.

Negated conjunction (NAND). The NAND of two propositional formulas is the negation of their conjunction. We will denote the NAND of p and q simply as $\text{NAND}(p, q)$. The standard logic circuit symbol for NAND is shown in Fig. 2.

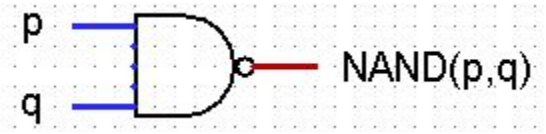


Figure 2: NAND gate. Screenshot from *Logisim*.

We will show that the set consisting of only NAND is a complete set of boolean operators. In other words, NAND gates suffice to construct any combinational logic circuit. In order to show completeness of $\{NAND\}$, it is enough to show that each of the three operators of the complete set $\{\neg, \wedge, \vee\}$ can be represented in terms of NAND.

1. $\neg$ is easy to represent: $\neg p$ is equivalent to $\text{NAND}(p, p)$.
2. $\wedge$ is now easy, too: $p \wedge q$ is equivalent to $\neg(p \text{ NAND } q)$, and $\neg$ is representable in terms of NAND (above).
3. In light of the above, $\vee$ can be represented in terms of NAND by using DeMorgan's laws.

2 Some algorithms in propositional logic

2.1 Evaluation of propositional formulas

Theorem 2.1. *The truth value of a propositional formula may be computed in at most Cn^2 steps, where n is the length of the formula and C is a finite constant independent of n.*

Proof. Algorithm 1 provides a specific way to evaluate a propositional formula. The strategy followed is simply to successively evaluate the leftmost complete subformula by referring to the truth table for the appropriate operator.

Algorithm 1: Boolean propositional evaluation
Input: propositional formula, ϕ; truth assignment, t, to the variables of ϕ
Output: the truth value $t(\phi)$ of ϕ under t
EVALUATE(ϕ, t)
(1) **while** the length of ϕ is greater than 1
(2) let i be the index of the leftmost ')' in ϕ
(3) let j be the index of the rightmost '(' among $j < i$
(4) replace the substring $\phi_{j\cdots i}$ by its truth value $t(\phi_{j\cdots i})$
(5) **return** ϕ

Here is a sample run of a program that implements Algorithm 1. The top line shows the original formula, and successive lines show the reduced formula obtained by replacing the leftmost subformula at each stage. Truth values are represented as 0 and 1 in order to simplify string processing.

```
(1>((((0v1)^(1v0))^(1^0))>0))
(1>(((1^(1v0))^(1^0))>0))
(1>(((1^1)^(1^0))>0))
(1>((1^(1^0))>0))
(1>((1^0)>0))
(1>(0>0))
(1>1)
1
```

As you can infer from this example, the number of iterations of the algorithm (lines in the sample run) is less than n, the length of the original formula, because each iteration reduces the length of the formula. Each iteration requires finding the leftmost subformula and replacing it with its truth value. Let's break this per-iteration time down further: scanning for the leftmost ')' takes at most n steps, finding the matching '(' takes just three or four steps; replacing the subformula can take a number of steps proportional to n, depending on the internal string representation used. In any case, the per-iteration time is at most Cn for some finite constant C. It follows that the total number of steps needed by the algorithm is at most Cn^2, as claimed. Sample elapsed times for different formula lengths appear below. The ratio between consecutive lengths in this list is approximately 2.

```
length, time (s)
33, 8.29696655273e-05
69, 0.000124931335449
```

```
141, 0.000247001647949
285, 0.000566959381104
573, 0.0011088848114
1149, 0.0022439956665
2301, 0.00455713272095
4605, 0.00929999351501
9213, 0.0196549892426
18429, 0.0431261062622
36861, 0.107383966446
73725, 0.287739038467
147453, 0.87535905838
294909, 4.60914897919
589821, 14.7984681129
1179645, 49.4246001244
2359293, 182.577973843
4718589, 685.218982935
9437181, 2976.95564008
```

2.2 Satisfiability of propositional formulas

Theorem 2.2. *There is an algorithm that solves the problem of determining if a given propositional formula is satisfiable, and runs in at most $Cn^2 2^k$ steps, where n is the length of the input formula, k is the number of boolean variables in the input formula, and C is a finite constant independent of n and k.*

Proof. Algorithm 2 solves the satisfiability problem very straightfowardly, by trying all possible truth assignments: if any one of these makes the input formula **true**, the algorithm reports that the formula is satisfiable; if none of the candidate truth assignments makes the formula **true**, then the algorithm reports that the formula is not satisfiable.

Algorithm 2:
Input: a propositional formula, ϕ
Output: **true** if ϕ is satisfiable, **false** otherwise
isSatisf(ϕ)
(1) determine the set of distinct variables in ϕ
(2) **foreach** truth assignment t of the variables in ϕ
(3) **if** $t(\phi)$ is **true**
(4) **return true**
(5) **return false**

How many steps are required by Algorithm 2? Finding the variables in the input formula, ϕ, can be accomplished in one pass over the formula: a list is kept of the variables encountered so far, and each symbol of ϕ is examined to see if it is either an operator or constant or else if the symbol belongs to the list of variables so far. If the list of variables is represented as an array, a membership test requires examining at most as many positions as variables. Thus, counting the distinct variables in ϕ takes at most Cnk steps for some finite C, where k is the

total number of variables, and that is less than $Cn^2 2^k$. There are 2^k truth assignments of the k variables in ϕ. Extending any such truth assignment to ϕ itself can be accomplished in Cn^2 steps, by Theorem 2.1. It follows that the total number of steps required for the main loop in the algorithm is at most $Cn^2 2^k$. This completes the proof.

3 Applications of propositional logic

3.1 Proving that code is correct

A *loop invariant* for a particular loop construct within a program is a propositional statement that evaluates to **true** each time the loop guard condition is evaluated. The use of loop invariants allows proving iterative code to be correct (or discovering that it needs to be fixed).

3.1.1 Example: accumulating a sum

For example, consider the following code:

```
>>> sum = 0
>>> i = 1
>>> while i <= 20:
...     sum = sum + i
...     i = i + 1
```

Let p denote the propositional statement

> *The value of the* `sum` *variable is the sum of all integers between* 0 *and* $i - 1$*, inclusive.*

Invariance of p. The above proposition p is **true** when the loop is first entered, as the values when the loop condition is first evaluated are `sum=0, i=1`. Assuming that p is **true** on a given pass, it will also be **true** on the next, as `sum` will now have accumulated the value `i` just before `i` is incremented by 1. It follows by induction that p is **true** each time the loop condition is evaluated. That is, p is a loop invariant for the above loop. This fact allows one to conclude that the value of `sum` will be the sum of all integers between 1 and 20 after the loop, since the loop condition is last evaluated when `i` is 21.

3.1.2 Example: decoding binary positional notation

Consider the problem of determining the (decimal integer) value of a string of binary digits in standard (unsigned) binary positional notation. For example, the bit sequence 11001 represents the value $2^4 + 2^3 + 2^0 = 25$. Here is a Python function that performs this conversion, assuming that the original bit sequence is given in the form of a list such as $[1, 1, 0, 0, 1]$.

```
>>> def readBinary(bits):
...     n = 0
...     while len(bits) > 0:
...             n = 2*n + bits[0]
...             bits = bits[1:]
...     return n
```

We will find a loop invariant for the main loop in this function, and use it to show the correctness of the function itself.

Discovering a loop invariant. Consider the process of converting $[1, 1, 0, 0, 1]$ to an integer, as followed in the above function `readBinary`. We show the values of the variables `n` and `bits` each time the loop entry condition is evaluated:

```
n      bits
0     [1, 1, 0, 0, 1]
1     [1, 0, 0, 1]
3     [0, 0, 1]
6     [0, 1]
12    [1]
25    []
```

What we need is a boolean propositional statement that is **true** on each of the lines in any run of the function, the above example included. This statement should be phrased in terms of the variables `n` and `bits`. In order to develop an intuitive idea of what may work, notice in the above example that the `bits` list gets shorter as the value of `n` increases. This suggests that there may be some combination of the two that doesn't change from one line to the next. What is it? Well, you see that the bits are being accounted for in decreasing order of significance, starting with the most significant bit: the value of `n` at each point of the computation reflects the bits that have been accounted for so far. Thus, on the first line no bits have been accounted for, and the value of `n` is 0, while on the third line, the two most significant bits have been accounted for somehow by the value 3 for `n`. The remaining bits, 001, have yet to be "folded into" `n`. Intuitively, the total numerical value of the original bit string is spread across the value of `n` so far, on one hand, and the remaining bits, on the other. How would one reconstruct this numerical value on any given line? The remaining bits should be taken at face value: they represent powers of 2 according to their position within the list. The current value of `n`, however, will actually end up representing a larger magnitude in the final answer, since it will be doubled on each subsequent pass through the loop body. By fleshing out this thought process, one concludes that the following statement should be **true** whenever the loop entry condition is evaluated:

> The numerical value n_0 of the original bit string may be recovered as $2^{\text{len(bits)}}n$ plus the numerical value represented by `bits`.

Proving invariance of the proposed invariant. We've done the first task of formulating the loop invariant carefully. That should facilitate the job of actually *proving* its invariance beyond a shadow of a doubt. Let's go. We will prove that the proposed loop invaiant is **true** whenever the loop condition is evaluated. We will do this by induction in the number of previous evaluations of the loop entry condition.

- (basis) Suppose that the loop entry condition has not yet been evaluated at all. In that case, `bits` contains the full original bit string that was passed to the `readBinary` function, and `n` has the value 0. Therefore, the proposed loop invariant states that the numerical value n_0 of the original bit string may be recovered as 0 plus the numerical value represented by the original bit string. Can't argue with that.

- (inductive step) Suppose that the loop entry condition has previously been evaluated some number of times, and that it has had the value **true** each one of those times. We claim that the loop condition will also be **true** the next time that it is evaluated. Since the loop condition evaluated to **true** on the previous evaluation, this means that, if p was the length of `bits` at that time, and if n was the value of `n` on the previous evaluation, then the numerical value represented by the original bit string may be expressed as $2^p n$ plus the numerical value represented by `bits` on the previous evaluation. Accounting for the most significant bit `bits[0]` among those remaining, this value is the same as $2^p n$ plus 2^{p-1}bits[0] plus the numerical value represented by `bits[1:]`, which can be expressed as $2^{p-1}(2n + \texttt{bits[0]})$ plus the numerical value represented by `bits[1:]`. However, because of the way in which `bits` and `n` are updated in the loop body, the exponent $p - 1$ is precisely the length of the updated bit string, and $2n + \texttt{bits[0]}$ is precisely the updated value of `n`; these values are the ones in place at the time of the current evaluation. We therefore conclude that the numerical value of the original bit string equals $2^{\text{current len(bits)}}$ times the current value of `n` plus the numerical value represented by the current value of `bits`. This proves the inductive step, and completes the proof of the invariance of the proposed loop invariant.

Using the loop invariant to prove correctness of the binary to decimal integer function. In the end, there is a particular interest in showing that programs actually behave the way that they *should*. Here, we will show that `readBinary(L)` actually does what it is supposed to, namely that it returns the numerical value of the bit string contained in the list `L`. This is pretty easy at this point, since we have a good invariant in hand. We know that, at each evaluation of the loop entry condition during the invocation `readBinary(L)`, the numerical value represented by the bits in `L` may be expressed as follows:

$$\text{value of L} = 2^{\text{len(bits)}} n + \text{numerical value of current bits list}$$

We know also that the final evaluation of the loop entry condition occurs when that loop entry condition evaluates to **false** for the first time. That occurs when the `bits` list is empty, which represents a numerical value of 0. At that point, the loop invariant is **true**, as it is

at the time of each loop entry evaluation, and states the following: the numerical value represented by the bits in `L` is equal to

$$\text{value of L} = 2^0 n + 0 = n$$

Since the value `n` is returned immediately after this final loop condition evaluation, the function correctly performs its job.

4 Exercises

1. Provide a detailed definition by recursion for the parse tree of a propositional statement, based directly on the definition of propositional statement.

2. Prove in detail that a propositional formula is satisfiable if, and only if, its negation is not a tautology.

3. Show that every formula is equivalent to a formula in conjunctive normal form. *Hint. First, find a DNF formula that is equivalent to the* negation *of the target formula, as described in these notes, and then use DeMorgan's laws.*

4. Implement Algorithm 1 as a program (script) in Python. Submit documented source code and screenshots from sample runs.

5. Consider the following Python code (shown on the command line, so ignore the prompt symbols and other punctuation):

```
>>> p=0
>>> while n>=1:
...     p = p+1
...     n = n/2
```

   (a) Define a loop invariant for the loop shown, as a specific expression in terms of n and p. Explain.

   (b) Prove by induction that the expression from the preceding part has the same value every time the loop condition is evaluated.

   (c) Express the final value of p when the loop is exited in terms of the initial value of n. Explain.