Artificial Intelligence

Lab Instructions

Table of Contents

LAB 1 Introduction to Python – I

The purpose of this lab is to familiarize you with this term's lab system and to diagnose your programming ability and facility with Python. This course uses Python for all of its labs, and you will be called on to understand the functioning of large systems, as well as to write significant pieces of code yourself. While coding is not, in itself, a focus of this class, artificial intelligence is a hard subject full of subtleties. As such, it is important that you be able to focus on the problems you are solving, rather than the mechanical code necessary to implement the solution.

Python resources

Udacity offers a really helpful Introductory course for Python Beginners
https://classroom.udacity.com/courses/ud1110

Python: There are a number of versions of Python available. This course will use standard Python from http://www.python.org/. If you are running Python on your own computer, you should download and install latest version (Currently Python 3.7.4) from http://www.python.org/download/ .

1.1 Essentials of a Python program:

In most of today's written languages, words by themselves do not make sense unless they are in certain order and surrounded by correct punctuation symbols. This is also the case with the Python programming language. The Python interpreter is able to interpret and run correctly structured Python programs. For example, the following Python code is correctly structured and will run:

```
print("Hello, world!")
```

Many other languages require a lot more structure in their simplest programs, but in Python this single line, which prints a short message, is sufficient. A very informative example of Python's syntax which does (almost) exactly the same thing:

```
# Here is the main function.
def my_function():
    print("Hello, World!")

my_function()
```

A hash (#) denotes the start of a comment. The interpreter will ignore everything that follows the hash until the end of the line.

1.1.1 Flow of control

In Python, statements are written as a list, in the way that a person would write a list of things to do. The computer starts off by following the first instruction, then the next, in the order that they appear in the program. It only stops executing the program after the last instruction is completed. We refer to the order in which the computer executes instructions as the flow of control. When the computer is executing a particular instruction, we can say that control is at that instruction.

1.1.2 Indentation and (lack of) semicolons

Many languages arrange code into blocks using curly braces (`{` and `}`) or `BEGIN` and `END` statements – these languages encourage us to indent blocks to make code easier to read, but indentation is not compulsory. Python uses indentation only to delimit blocks, so we must indent our code:

```
# this function definition starts a new block
def add_numbers(a, b):
    # this instruction is inside the block, because it's indented
    c = a + b
    # so is this one
    return c

# this if statement starts a new block
if it_is_tuesday:
    # this is inside the block
    print("It's Tuesday!")
# this is outside the block!
print("Print this no matter what.")
```

In many languages we need to use a special character to mark the end of each instruction – usually a semicolon. Python uses ends of lines to determine where instructions end (except in some special cases when the last symbol on the line lets Python know that the instruction will span multiple lines). We may optionally use semicolons – this is something we might want to do if we want to put more than one instruction on a line (but that is usually bad style):

```
# These all individual instructions -- no semicolons required!
print("Hello!")
print("Here's a new instruction")
a = 2

# This instruction spans more than one line
b = [1, 2, 3,
    4, 5, 6]

# This is legal, but we shouldn't do it
c = 1; d = 5
```

1.1.3 Built-in types

There are many kinds of information that a computer can process, like numbers and characters. In Python, the kinds of information the language is able to handle are known as types. Many common types are built into Python – for example integers, floating-point numbers and strings. Users can also define their own types using classes. In many languages a distinction is made between built-in types (which are often called “primitive types” for this reason) and classes, but in Python they are indistinguishable. Everything in Python is an object (i.e. an instance of some class) – that even includes lists and functions.

Python is a dynamically (and not statically) typed language. That means that we don’t have to specify a type for a variable when we create it – we can use the same variable to store values of different types. However, Python is also strongly typed – at any given time, a variable has a definite type. If we try to perform operations on variables which have incompatible types (for example, if we try to add a number to a string), Python will exit with a type error instead of trying to guess what we mean.

Python has large collection of built-in functions that operate on different kinds of data to produce all kinds of results. One function is called type, and it returns the type of any object.

```
>>> type(7)
<class 'int'>
>>>
```

1.1.3.1 Numbers:

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages.

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

1.1.3.2 Strings

A string is a sequence of characters. Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result. \ can be used to escape quotes. In the interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes.

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Some common escape sequences:

Sequence	Meaning
`\\`	literal backslash
`\'`	single quote
`\"`	double quote
`\n`	newline
`\t`	tab

Sometimes we may need to define string literals which contain many backslashes – escaping all of them can be tedious. We can avoid this by using Python's raw string notation. By adding an **r** before the opening quote of the string, we indicate that the contents of the string are exactly what we have written, and that backslashes have no special meaning. For example:

```
# This string ends in a newline
"Hello!\n"

# This string ends in a backslash followed by an 'n'
r"Hello!\n"
```

Consider another example:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

1.1.4 Files

Although the `print` function prints to the console by default, we can also use it to write to a file. Here is a simple example:

```
with open('myfile.txt', 'w') as myfile:
    print("Hello!", file=myfile)
```

In the **with** statement the file **myfile.txt** is opened for writing and assigned to the variable **myfile**. Inside the **with** block, **Hello!** followed by a newline is written to the file. The **w** character passed to open indicates that the file should be opened for writing. The **with** statement automatically closes the file at the end of the block, even if an error occurs inside the block.

As an alternative to **print**, we can use a file's **write** method as follows:

```
with open('myfile.txt', 'w') as myfile:
    myfile.write("Hello!")
```

Unlike **print**, the **write** method does not add a newline to the string which is written.

We can read data from a file by opening it for reading and using the file's **read** method:

```
with open('myfile.txt', 'r') as myfile:
    data = myfile.read()
```

This reads the contents of the file into the variable data. Note that this time we have passed **r** to the open function. This indicates that the file should be opened for reading.

1.1.5 Variable scope and lifetime

Where a variable is accessible and how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its scope, and the duration for which the variable exists its lifetime. A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace. A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

Here is an example of variables in different scopes:

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

1.2 Selection control statements

1.2.1 Selection: `if` statement

Perhaps the most well-known statement type is the `if` statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more **`elif`** parts, and the else part is optional. The keyword '**`elif`**' is short for '**`else if`**', and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

The interpreter will treat all the statements inside the indented block as one statement – it will process all the instruction in the block before moving on to the next instruction. This allows us to specify multiple instructions to be executed when the condition is met.

1.2.2 The **`for`** Statement

The **`for`** statement in Python differs a bit from what you may be used to in C. Rather than always giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
for i in range(1, 9):
    print(i)
```

Consider another example.

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

If you do need to iterate over a sequence of numbers, the built-in function **`range ()`** comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

1.3 Defining Functions

A function is a sequence of statements which performs some kind of task. Here is a definition of a simple function which takes no parameters and doesn't return any values:

```
def print_a_message():
    print("Hello, world!")
```

We use the **`def`** statement to indicate the start of a function definition. The next part of the definition is the function name, in this case `print_a_message`, followed by round brackets (the definitions of any parameters that the function takes will go in between them) and a colon. Thereafter, everything that is indented by one level is the body of the function.

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

1.3.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the **in** keyword. This tests whether or not a sequence contains a certain value.

In Python, there can only be one function with a particular name defined in the scope – if you define another function with the same name, you will overwrite the first function. You must call this function with the correct number of parameters, otherwise you will get an error. Sometimes there is a good reason to want to have two versions of the same function with different sets of parameters. You can achieve something similar to this by making some parameters optional. To make a parameter optional, we need to supply a default value for it. Optional parameters must come after all the required parameters in the function definition:

```
def make_greeting(title, name, surname, formal=True):
    if formal:
        return "Hello, %s %s!" % (title, surname)

    return "Hello, %s!" % name

print(make_greeting("Mr", "John", "Smith"))
print(make_greeting("Mr", "John", "Smith", False))
```

When we call the function, we can leave the optional parameter out – if we do, the default value will be used. If we include the parameter, our value will override the default value.

1.3.2 Lambdas

We have already seen that when we want to use a number or a string in our program we can either write it as a literal in the place where we want to use it or use a variable that we have already defined in our code. For example, `print("Hello!")` prints the literal string `"Hello!"`, which we haven't stored in a variable anywhere, but `print(message)` prints whatever string is stored in the variable message.

We have also seen that we can store a function in a variable, just like any other object, by referring to it by its name (but not calling it). Is there such a thing as a function literal? Can we define a function on the fly when we want to pass it as a parameter or assign it to a variable, just like we did with the string "Hello!"?

The answer is yes, but only for very simple functions. We can use the `lambda` keyword to define anonymous, one-line functions inline in our code:

```
a = lambda: 3

# is the same as

def a():
    return 3
```

Lambdas can take parameters – they are written between the lambda keyword and the colon, without brackets. A lambda function may only contain a single expression, and the result of evaluating this expression is implicitly returned from the function (we don't use the `return` keyword):

```
b = lambda x, y: x + y

# is the same as

def b(x, y):
    return x + y
```

1.4 Class Definitions:

A class serves as the primary means for abstraction in object-oriented programming. In Python, every piece of data is represented as an instance of some class. A class provides a set of behaviors in the form of member functions (also known as methods), with implementations that are common to all instances of that class. A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of attributes (also known as fields, instance variables, or data members).

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example: CreditCard Class

As a first example, we will define a `CreditCard` class. The instances defined by the `CreditCard` class provide a simple model for traditional credit cards. They have identifying information about the customer, bank, account number, credit limit, and current balance. The class restricts charges that would cause a card's balance to go over its spending limit, but it does not charge interest or late payments.

The construct begins with the keyword, class, followed by the name of the **`class`**, a colon, and then an indented block of code that serves as the body of the class. The body includes definitions for all methods of the class. These methods are defined as functions, yet with a special parameter, named **`self`**, that serves to identify the particular instance upon which a member is invoked.

By convention, names of members of a class (both data members and member functions) that start with a single underscore character (e.g., `_secret`) are assumed to be nonpublic and should not be relied upon.

1.4.1 The **`self`** Identifier:

In Python, the `self` identifier plays a key role. In the context of the `CreditCard` class, there can presumably be many different `CreditCard` instances, and each must maintain its own balance, its own credit limit, and so on. Therefore, each instance stores its own instance variables to reflect its current state.

Syntactically, `self` identifies the instance upon which a method is invoked. For example, assume that a user of our class has a variable, `my_card`, that identifies an instance of the `CreditCard` class. When the user calls `my_card.get_balance( )`, identifier self, within the definition of the `get_balance` method, refers to the card known as `my_card` by the caller. The expression, `self._balance` refers to an instance variable, named `_balance`, stored as part of that particular credit card's state.

```
class CreditCard:
  """A consumer credit card."""

  def __init__(self, customer, bank, acnt, limit):
    """Create a new credit card instance.

    The initial balance is zero.

    customer  the name of the customer (e.g., 'John Bowman')
    bank      the name of the bank (e.g., 'California Savings')
    acnt      the acount identifier (e.g., '5391 0375 9387 5309')
    limit     credit limit (measured in dollars)
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0

  def get_customer(self):
    """Return name of the customer."""
    return self._customer

  def get_bank(self):
    """Return the bank's name."""
    return self._bank

  def get_account(self):
    """Return the card identifying number (typically stored as a string)."""
    return self._account

  def get_limit(self):
    """Return current credit limit."""
    return self._limit

  def get_balance(self):
    """Return current balance."""
    return self._balance
```

Code Fragment 2.1: The beginning of the CreditCard class definition (continued in Code Fragment 2.2).

```
    def charge(self, price):
        """Charge given price to the card, assuming sufficient credit limit.

        Return True if charge was processed; False if charge was denied.
        """
        if price + self._balance > self._limit:    # if charge would exceed limit,
            return False                           # cannot accept charge
        else:
            self._balance += price
            return True

    def make_payment(self, amount):
        """Process customer payment that reduces balance."""
        self._balance −= amount
```

Code Fragment 2.2: The conclusion of the CreditCard class definition (continued from Code Fragment 2.1). These methods are indented within the class definition.

1.4.2 The Constructor

A user can create an instance of the `CreditCard` class using a syntax as:

```
cc = CreditCard( John Doe, 1st Bank , 5391 0375 9387 5309 , 1000)
```

Internally, this results in a call to the specially named `_ _init_ _` method that serves as the constructor of the class. Its primary responsibility is to establish the state of a newly created credit card object with appropriate instance variables. By the conventions, a single leading underscore in the name of a data member, such as `_balance`, implies that it is intended as nonpublic. Users of a class should not directly access such members.

Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `_ _init_ _()`, like this:

```
def __init__(self):
    self.data = []
```

Testing the Class:

We will demonstrate some basic usage of the `CreditCard` class, inserting three cards into a list named `wallet`. We use loops to make some charges and payments, and use various accessors to print results to the console. These tests are enclosed within a conditional, `if _ _name == '_ _main_ _':`, so that they can be embedded in the source code with the class definition.

```
if __name__ == '__main__':
  wallet = [ ]
  wallet.append(CreditCard('John Bowman', 'California Savings',
                          '5391 0375 9387 5309', 2500) )
  wallet.append(CreditCard('John Bowman', 'California Federal',
                          '3485 0399 3395 1954', 3500) )
  wallet.append(CreditCard('John Bowman', 'California Finance',
                          '5391 0375 9387 5309', 5000) )

  for val in range(1, 17):
    wallet[0].charge(val)
    wallet[1].charge(2*val)
    wallet[2].charge(3*val)

  for c in range(3):
    print('Customer =', wallet[c].get_customer())
    print('Bank =', wallet[c].get_bank())
    print('Account =', wallet[c].get_account())
    print('Limit =', wallet[c].get_limit())
    print('Balance =', wallet[c].get_balance())
    while wallet[c].get_balance( ) > 100:
      wallet[c].make_payment(100)
      print('New balance =', wallet[c].get_balance())
    print()
```

Code Fragment 2.3: Testing the CreditCard class.

1.4.3 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

1.5 Lab Tasks

Exercise 1.1

Type each of the following expressions into python3. What value do each of the following Python expressions evaluate to? Is that value an integer or a floating point?

a. `250`

b. `28 % 5`

c. `2.5e2`

d. `3e5`

e. `3 * 10**5`

f. `20 + 35 * 2`
Why is this different from `(20 + 35) * 2`*?*

g. `2 / 3 * 3`

h. `2 // 3 * 3`
Why is this different from `2 / 3 * 3`*?*

i. `25 - 5 * 2 - 9`
Is this different from `((25 - 5) * 2) - 9` *and/or* `25 - ((5 * 2) - 9)`*? Why?*

Exercise 1.2

Suppose we are making ice cream sundaes. We have four flavors of ice cream: vanilla, chocolate, strawberry, and pistacchio. And we have three sauces: caramel, butterscotch, and chocolate. How many different ice cream sundaes can we make? Define a function `sundaes()` to systematically print out every possible combination, one per line. For example, the first line should say "vanilla ice cream sundae with caramel sauce". You should create a list to hold each class of ingredient, and use nested for loops to iterate over these lists to generate the combinations. At the end, your function should return an integer giving the total number of combinations.

To get you started here a few things you will need:

```
flavors = ["vanilla", "chocolate", "strawberry", "pistacchio"]
sauces  = ["caramel", "butterscotch", "chocolate"]

print(flavor + " ice cream sundae with " + sauce + " sauce")
```

Exercise 1.3

Consider the following output as shown:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31 32 33 34 35 36
```

```
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55
```

Complete the missing parts so that it creates the output above. Note that the columns of numbers do not need to line up perfectly. Run your program in using python3 to test your work. HINT: Row i has i columns.

```
def triangle():
    value = 1
    row = 1
    while row <= _____:
        column = 1
        while column <= _______:
            if column != _______:
                print(value, ' ', sep = '', end = '')
            else:
                print(value)
            value = value + 1
            column = column + 1
        row = row + 1
```

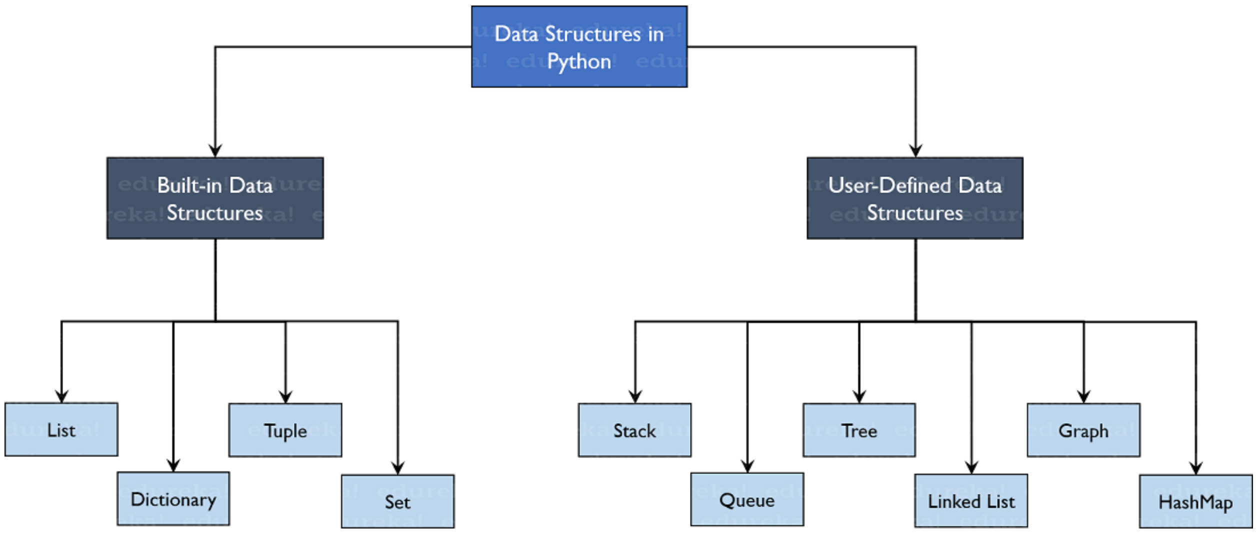
Exercise 1.4

1. Write the following functions:
   - `cube(n)`, which takes in a number and returns its cube. For example, cube(3) => 27.
   - `factorial(n)`, which takes in a non-negative integer n and returns n!, which is the product of the integers from 1 to n. (0! = 1 by definition.)
   - `count_pattern(pattern  lst)`, which counts the number of times a certain pattern of symbols appears in a list, including overlaps. So `count_pattern( ('a', 'b'), ('a','b', 'c', 'e', 'b', 'a', 'b', 'f'))` should return 2, and `count_pattern(('a', 'b', 'a'), ('g', 'a', 'b', 'a', 'b', 'a','b', 'a'))` should return 3.
   - Write a python program to print the multiplication table for the given number?
   - Write a python program to implement Simple Calculator program? (+, -, / ,*)
   - Write a python program to sort the sentence in alphabetical order?
2. Write a Python class to convert an integer to a roman numeral.
3. Write a Python class to find a pair of elements (indices of the two numbers) from a given array whose sum equals a specific target number. Input: numbers= [10,20,10,40,50,60,70], target=50 Output: 3, 4
4. Write a Python class to find the three elements that sum to zero from a set of n real numbers. Input array : [-25, -10, -7, -3, 2, 4, 8, 10] Output : [[-10, 2, 8], [-7, -3, 10]].
5. Write a Python class to reverse a string word by word. Input string : 'hello .py' Expected Output : '.py hello'
6. Count the numbers of characters in the string

   a. Read the string.
   b. Count the characters
   c. Display the result
7. Addition of two square matrices.
   d. Create a lists to read matrix elements
   e. Read the elements of to matrices add the elements
   f. Store the result in third matrix.
   g. Repeat steps 2 and 3 till the addition of all elements
8. Display the result Multiplication of two matrices
   h. Create a lists to read matrix elements
   i. Read the elements of two matrices, multiply the elements
   j. Store the result in third matrix.
   k. Repeat steps 2 and 3 till the multiplication of all elements
   l. Display the result.
9. Write a function called calculator. It should take the following parameters: two numbers, an arithmetic operation (which can be addition, subtraction, multiplication or division and is addition by default), and an output format (which can be integer or floating point, and is floating point by default). Division should be floating-point division. The function should perform the requested operation on the two input numbers, and return a result in the requested format (if the format is integer, the result should be rounded and not just truncated). Raise exceptions as appropriate if any of the parameters passed to the function are invalid.
10. Create a class called Numbers, which has a single class attribute called MULTIPLIER, and a constructor which takes the parameters x and y (these should all be numbers).
   m. Write a method called add which returns the sum of the attributes x and y.
   n. Write a class method called multiply, which takes a single number parameter a and returns the product of a and MULTIPLIER.
   o. Write a static method called subtract, which takes two number parameters, b and c, and returns b - c.
   p. Write a method called value which returns a tuple containing the values of x and y. Make this method into a property, and write a setter and a deleter for manipulating the values of x and y.

LAB 2 Introduction to Python – II

2.1 Data Structures in Python:



Built-in Data-structures:

- **Lists**: Stores indexed elements that are changeable and can contain duplicate items
- **Tuples**: Stores indexed, unchangeable elements that can have duplicate copies
- **Dictionaries**: Store key-value pairs that are changeable
- **Sets**: Contains unordered, unique elements that are mutable

User-defined Data-structures:

- **Arrays**: Similar to Lists, but store single type of elements
- **Stack**: Linear LIFO (Last-In-First-Out) Data structure
- **Queues**: Linear FIFO (First-In-First-Out) data structure
- **Trees**: Non-Linear data structures having a root and nodes
- **Linked Lists**: Linear data structures that are linked with pointers
- **Graphs**: Store a collection of points or nodes along with edges
- **Hash Maps**: In Python, Hash Maps are the same as Dictionaries

2.2 List

List is a type of sequence – we can use it to store multiple values, and access them sequentially, by their position, or index, in the list. We define a list literal by putting a comma-separated list of values inside square brackets:

```
# a list of strings
animals = ['cat', 'dog', 'fish', 'bison']

# a list of integers
numbers = [1, 7, 34, 20, 12]

# an empty list
my_list = []

# a list of variables we defined somewhere else
things = [
    one_variable,
    another_variable,
    third_variable, # this trailing comma is legal in Python
]
```

To refer to an element in the list, we use the list identifier followed by the index inside square brackets. Indices are integers which start from zero:

```
print(animals[0]) # cat
print(numbers[1]) # 7

# This will give us an error, because the list only has four elements
print(animals[6])
```

We can also count from the end:

```
print(animals[-1]) # the last element -- bison
print(numbers[-2]) # the second-last element -- 20
```

We can extract a subset of a list, which will itself be a list, using a slice. This uses almost the same syntax as accessing a single element, but instead of specifying a single index between the square brackets we need to specify an upper and lower bound. Note that our sublist will include the element at the lower bound, but exclude the element at the upper bound:

```
print(animals[1:3]) # ['dog', 'fish']
print(animals[1:-1]) # ['dog', 'fish']
```

How do we check whether a list contains a particular value? We use in or not in, the membership operators:

```
numbers = [34, 67, 12, 29]
my_number = 67

if number in numbers:
    print("%d is in the list!" % number)

my_number = 90
if number not in numbers:
    print("%d is not in the list!" % number)
```

The list data type has some more methods. An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

2.3 Stack and Queues:

A stack (sometimes called a "push-down stack") is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This ordering principle is sometimes called LIFO, last-in first-out. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base. There are two primary operations that are done on stacks: `push` and `pop`. When an element is added to the top of the stack, it is pushed onto the stack. When an element is taken off the top of the stack, it is popped off the stack.

Stack Operation	Stack Contents	Return Value
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.is_empty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Table 3.1: Sample Stack Operations

For example, if s is a stack that has been created and starts out empty, then Table 3.1 shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

2.3.1 Using List as Stack:

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

2.3.2 Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one). To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

2.4 Tuples

Python has another sequence type which is called tuple. Tuples are similar to lists in many ways, but they are immutable. We define a tuple literal by putting a comma-separated list of values inside round brackets:

```
WEEKDAYS = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday',
 ↪'Sunday')
```

What are tuples good for? We can use them to create a sequence of values that we don’t want to modify. For example, the list of weekday names is never going to change. If we store it in a tuple, we can make sure it is never modified accidentally in an unexpected place.

2.5 Sets

The Python set type is called set. A set is a collection of unique elements. If we add multiple copies of the same element to a set, the duplicates will be eliminated, and we will be left with one of each element. To define a set literal, we put a comma-separated list of values inside curly brackets.

```
animals = {'cat', 'dog', 'goldfish', 'canary', 'cat'}
print(animals) # the set will only contain one cat
```

Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we discuss in the next section. Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

2.6 Dictionaries

The Python dictionary type is called dict. We can use a dictionary to store key-value pairs. To define a dictionary literal, we put a comma-separated list of key-value pairs between curly brackets.

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

personal_details = {
    "name": "Jane Doe",
    "age": 38, # trailing comma is legal
}

print(marbles["green"])
print(personal_details["name"])

# This will give us an error, because there is no such key in the dictionary
print(marbles["blue"])

# modify a value
marbles["red"] += 3
personal_details["name"] = "Jane Q. Doe"
```

We use a colon to separate each key from its value. We access values in the dictionary in much the same way as list or tuple elements, but we use keys instead of indices. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys.

2.7 Two-dimensional sequences

What if we want to use a sequence to represent a two-dimensional data structure, which has both rows and columns? The easiest way to do this is to make a sequence in which each element is also a sequence. For example, we can create a list of lists:

```
my_table = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12],
]
```

The outer list has four elements, and each of these elements is a list with three elements (which are numbers). To access one of these numbers, we need to use two indices – one for the outer list, and one for the inner list:

```
print(my_table[0][0])

# lists are mutable, so we can do this
my_table[0][0] = 42
```

When we use a two-dimensional sequence to represent tabular data, each inner sequence will have the same length, because a table is rectangular – but nothing is stopping us from constructing two-dimensional sequences which don't have this property:

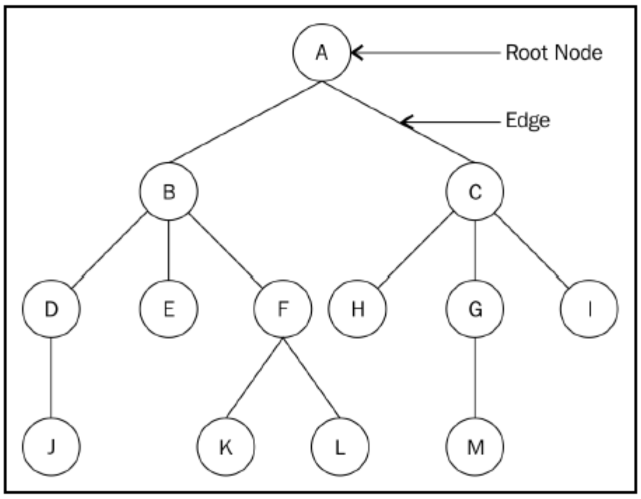
```
my_2d_list = [
    [0],
    [1, 2, 3, 4],
    [5, 6],
]
```

We can also make a three-dimensional sequence by making a list of lists of lists:

```
my_3d_list = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
]

print(my_3d_list[0][0][0])
```

2.8 Trees:

A tree is a hierarchical form of data structure. When we dealt with lists, queues, and stacks, items followed each other. But in a tree, there is a parent-child relationship between items. At the top of every tree is the so-called root node. This is the ancestor of all other nodes in the tree.



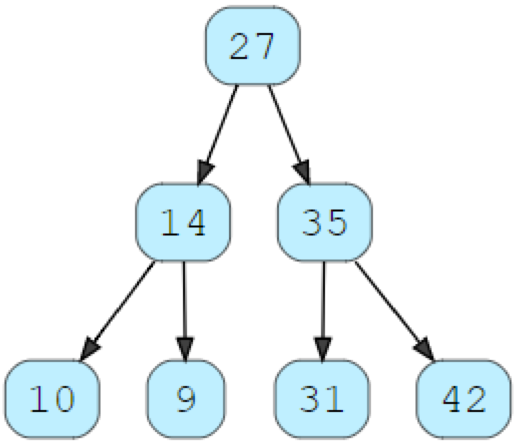
To understand trees, we need to first understand the basic ideas on which they rest. The following figure contains a typical tree consisting of character nodes lettered A through to M.

Here is a list of terms associated with a Tree:

- **Node**: Each circled alphabet represents a node. A node is any structure that holds data.
- **Root node**: The root node is the only node from which all other nodes come. A tree with an undistinguishable root node cannot be considered as a tree. The root node in our tree is the node A.
- **Sub-tree**: A sub-tree of a tree is a tree with its nodes being a descendant of some other tree. Nodes F, K, and L form a sub-tree of the original tree consisting of all the nodes.
- **Degree**: The number of sub-trees of a given node. A tree consisting of only one node has a degree of 0. This one tree node is also considered as a tree by all standards. The degree of node A is 2.
- **Leaf node**: This is a node with a degree of 0. Nodes J, E, K, L, H, M, and I are all leaf nodes.
- **Edge**: The connection between two nodes. An edge can sometimes connect a node to itself, making the edge appear as a loop.
- **Parent**: A node in the tree with other connecting nodes is the parent of those nodes. Node B is the parent of nodes D, E, and F.
- **Child**: This is a node connected to its parent. Nodes B and C are children of node A, the parent and root node.
- **Sibling**: All nodes with the same parent are siblings. This makes the nodes B and C siblings.
- **Level**: The level of a node is the number of connections from the root node. The root node is at level 0. Nodes B and C are at level 1.
- **Height of a tree**: This is the number of levels in a tree. Our tree has a height of 4.
- **Depth**: The depth of a node is the number of edges from the root of the tree to that node. The depth of node H is 2.

2.8.1 Binary tree

A tree whose elements have at most two children is called a binary tree. Each element in a binary tree can have only two children. A node's left child must have a value less than its parent's value, and the node's right child must have a value greater than its parent value.



Implementation: Here we have created a "node" class and assigned a value to the node.

```
# node class
class Node:

    def __init__(self, data):
        # left child
        self.left = None
        # right child
        self.right = None
        # node's value
        self.data = data

    # print function
    def PrintTree(self):
        print(self.data)

root = Node(27)

root.PrintTree()
```

Being a binary tree node, these references are to the left and the right children. To test this class out, we first create a few nodes:

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

Next, we connect the nodes to each other. We let **n1** be the root node with **n2** and **n3** as its children. Finally, we hook **n4** as the left child to **n2**, so that we get a few iterations when we traverse the left sub-tree:

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

Once we have our tree structure set up, we are ready to traverse it.

2.8.2 Binary search tree implementation

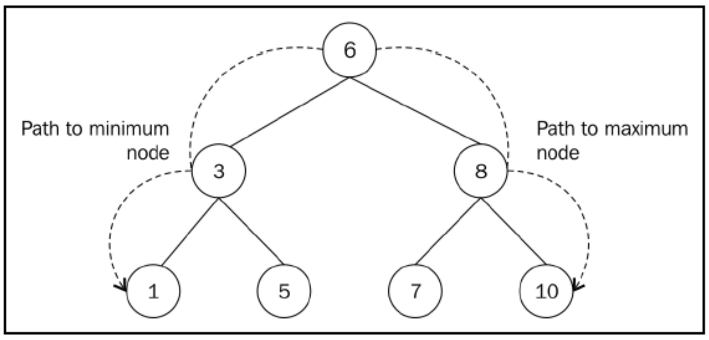
Let us begin our implementation of a BST. We will want the tree to hold a reference to its own root node:

```
class Tree:
    def __init__(self):
        self.root_node = None
```

That's all that is needed to maintain the state of a tree. Let's examine the main operations on the tree in the next section.

2.8.2.1 Finding the minimum and maximum nodes

The structure of the BST makes looking for the node with the maximum and minimum values very easy. To find the node with smallest value, we start our traversal from the root of the tree and visit the left node each time we reach a sub-tree. We do the opposite to find the node with the biggest value in the tree:



The method that returns the minimum node is as follows:

```
def find_min(self):
    current = self.root_node
    while current.left_child:
        current = current.left_child

    return current
```

The while loop continues to get the left node and visits it until the last left node points to None. It is a very simple method. The method to return the maximum node does the opposite, where `current.left_child` now becomes `current.right_child`.

2.8.2.2 Inserting nodes

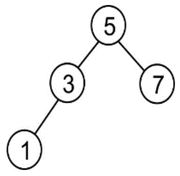
One of the operations on a BST is the need to insert data as nodes. Whereas in our first implementation, we had to insert the nodes ourselves, here we are going to let the tree be in charge of storing its data.

In order to make a search possible, the nodes must be stored in a specific way. For each given node, its left child node will hold data that is less than its own value, as already discussed. That node's right child node will hold data greater than that of its parent node.

We are going to create a new BST of integers by starting with the data 5. To do this, we will create a node with its data attribute set to 5.

Now, to add the second node with value 3, 3 is compared with 5, the root node. Since 5 is greater than 3, it will be put in the left sub-tree of node 5. The tree satisfies the BST rule, where all the nodes in the left sub-tree are less than its parent. To add another node of value 7 to the tree, we start from the root node with value 5 and do a comparison. Since 7 is greater than 5, the node with value 7 is situated to the right of this root. What happens when we want to add a node that is equal to an existing node? We will simply add it as a left node and maintain this rule throughout the structure.

If a node already has a child in the place where the new node goes, then we have to move down the tree and attach it. Let's add another node with value 1. Starting from the root of the tree, we do a comparison between 1 and 5. The comparison reveals that 1 is less than 5, so we move our attention to the left node of 5, which is the node with value 3. We compare 1 with 3 and since 1 is less than 3, we move a level below node 3 and to its left. But there is no node there. Therefore, we create a node with the value 1 and associate it with the left pointer of node 3 to obtain the following structure:



So far, we have been dealing only with nodes that contain only integers or numbers. For numbers, the idea of greater than and lesser than are clearly defined. Strings would be compared alphabetically, so there are no major problems there either. But if you want to store your own custom data types inside a BST, you would have to make sure that your class supports ordering.

Let's now create a function that enables us to add data as nodes to the BST. We begin with a function declaration:

```
def insert(self, data):
```

By now, you will be used to the fact that we encapsulate the data in a node. This way, we hide away the node class from the client code, who only needs to deal with the tree:

```
node = Node(data)
```

A first check will be to find out whether we have a root node. If we don't, the new node becomes the root node (we cannot have a tree without a root node):

```
if self.root_node is None:
    self.root_node = node
else:
```

As we walk down the tree, we need to keep track of the current node we are working on, as well as its parent. The variable `current` is always used for this purpose:

```
current = self.root_node
parent = None
while True:
    parent = current
```

Here we must perform a comparison. If the data held in the new node is less than the data held in the current node, then we check whether the current node has a left child node. If it doesn't, this is where we insert the new node. Otherwise, we keep traversing:

```
if node.data < current.data:
    current = current.left_child
    if current is None:
        parent.left_child = node
        return
```

Now we take care of the greater than or equal case. If the current node doesn't have a right child node, then the new node is inserted as the right child node. Otherwise, we move down and continue looking for an insertion point:

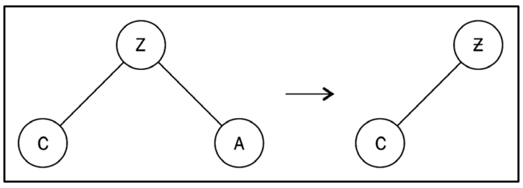
```
else:
    current = current.right_child
    if current is None:
        parent.right_child = node
        return
```

2.8.2.3 Deleting nodes

Another important operation on a BST is the deletion or removal of nodes. There are three scenarios that we need to cater for during this process. The node that we want to remove might have the following:

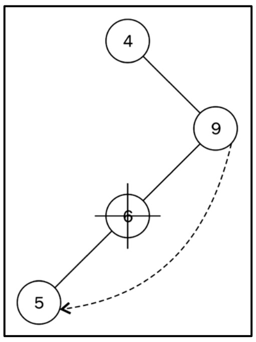
1. No children
2. One child
3. Two children

The first scenario is the easiest to handle. If the node about to be removed has no children, we simply detach it from its parent:



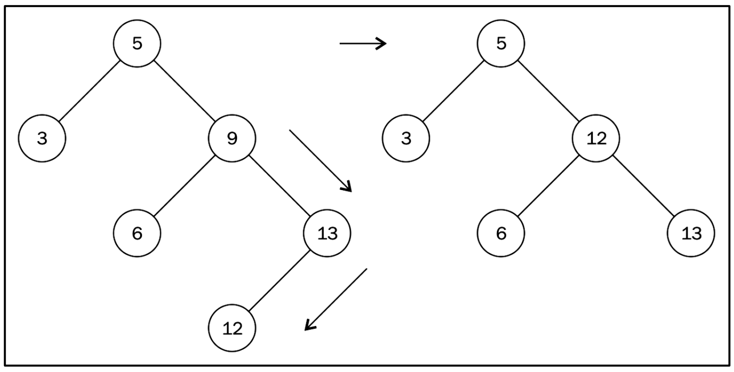
Because node A has no children, we will simply dissociate it from its parent, node Z.

On the other hand, when the node we want to remove has one child, the parent of that node is made to point to the child of that particular node:



In order to remove node 6, which has as its only child, node 5, we point the left pointer of node 9 to node 5. The relationship between the parent node and child has to be preserved. That is why we need to take note of how the child node is connected to its parent (which is the node about to be deleted). The child node of the deleted node is stored. Then we connect the parent of the deleted node to that child node.

A more complex scenario arises when the node we want to delete has two children:



We cannot simply replace node 9 with either node 6 or 13. What we need to do is to find the next biggest descendant of node 9. This is node 12. To get to node 12, we move to the right node of node 9. And then move left to find the leftmost node. Node 12 is called the in-order successor of node 9. The second step resembles the move to find the maximum node in a sub-tree.

We replace the value of node 9 with the value 12 and remove node 12. In removing node 12, we end up with a simpler form of node removal that has been addressed previously. Node 12 has no children, so we apply the rule for removing nodes without children accordingly.

Our **`node`** class does not have reference to a parent. As such, we need to use a helper method to search for and return the node with its parent node. This method is similar to the **`search`** method:

```
def get_node_with_parent(self, data):
    parent = None
    current = self.root_node
    if current is None:
        return (parent, None)
    while True:
        if current.data == data:
            return (parent, current)
        elif current.data > data:
            parent = current
            current = current.left_child
        else:
            parent = current
            current = current.right_child

    return (parent, current)
```

The only difference is that before we update the current variable inside the loop, we store its parent with `parent = current`. The method to do the actual removal of a node begins with this search:

```
def remove(self, data):
    parent, node = self.get_node_with_parent(data)

    if parent is None and node is None:
        return False

    # Get children count
    children_count = 0

    if node.left_child and node.right_child:
        children_count = 2
    elif (node.left_child is None) and (node.right_child is None):
        children_count = 0
    else:
        children_count = 1
```

We pass the parent and the found node to `parent` and `node` respectively with the line `parent, node = self.get_node_with_parent(data)`. It is helpful to know the number of children that the node we want to delete has. That is the purpose of the `if` statement.

```
if children_count == 0:
    if parent:
        if parent.right_child is node:
            parent.right_child = None
        else:
            parent.left_child = None
    else:
        self.root_node = None
```

In the case where the node about to be deleted has only one child, the `elif` part of the `if` statement does the following:

```
elif children_count == 1:
    next_node = None
    if node.left_child:
        next_node = node.left_child
else:
    next_node = node.right_child

if parent:
    if parent.left_child is node:
        parent.left_child = next_node
    else:
        parent.right_child = next_node
else:
    self.root_node = next_node
```

`next_node` is used to keep track of where the single node pointed to by the node we want to delete is. We then connect `parent.left_child` or `parent.right_child` to `next_node`.

Lastly, we handle the condition where the node we want to delete has two children:

```
...
else:
    parent_of_leftmost_node = node
    leftmost_node = node.right_child
    while leftmost_node.left_child:
        parent_of_leftmost_node = leftmost_node
        leftmost_node = leftmost_node.left_child

    node.data = leftmost_node.data
```

In finding the in-order successor, we move to the right node with `leftmost_node = node.right_child`. As long as there exists a left node, `leftmost_node.left_child` will evaluate to `True` and the `while` loop will run. When we get to the leftmost node, it will either be a leaf node (meaning that it will have no child node) or have a right child.

We update the node about to be removed with the value of the in-order successor with `node.data = leftmost_node.data`:

```
if parent_of_leftmost_node.left_child == leftmost_node:
   parent_of_leftmost_node.left_child = leftmost_node.right_child
else:
   parent_of_leftmost_node.right_child = leftmost_node.right_child
```

The preceding statement allows us to properly attach the parent of the leftmost node with any child node. Observe how the right-hand side of the equals sign stays unchanged. That is because the in-order successor can only have a right child as its only child.

2.8.3 Searching the tree

Since the `insert` method organizes data in a specific way, we will follow the same procedure to find the data. In this implementation, we will simply return the data if it was found or `None` if the data wasn't found:

```
def search(self, data):
```

We need to start searching at the very top, that is, at the root node:

```
current = self.root_node
while True:
```

We may have passed a leaf node, in which case the data doesn't exist in the tree and we return `None` to the client code:

```
if current is None:
    return None
```

We might also have found the data, in which case we return it:

```
elif current.data is data:
    return data
```

As per the rules for how data is stored in the BST, if the data we are searching for is less than that of the current node, we need to go down the tree to the left:

```
elif current.data > data:
    current = current.left_child
```

Now we only have one option left: the data we are looking for is greater than the data held in the current node, which means we go down the tree to the right:

```
else:
    current = current.right_child
```

Finally, we can write some client code to test how the BST works. We create a tree and insert a few numbers between 1 and 10. Then we search for all the numbers in that range. The ones that exist in the tree get printed:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)

for i in range(1, 10):
    found = tree.search(i)
    print("{}: {}".format(i, found))
```

2.8.4 Tree traversal

Visiting all the nodes in a tree can be done depth first or breadth first. These modes of traversal are not peculiar to only binary search trees but trees in general.

2.8.4.1 Depth-first traversal

In this traversal mode, we follow a branch (or edge) to its limit before recoiling upwards to continue traversal. We will be using the recursive approach for the traversal. There are three forms of depth-first traversal, namely `in-order`, `pre-order`, and `post-order`.

In-order traversal and infix notation

Most of us are probably used to this way of representing an arithmetic expression, since this is the way we are normally taught in schools. The operator is inserted (infixed) between the operands, as in `3 + 4`. When necessary, parentheses can be used to build a more complex expression: `(4 + 5) * (5 - 3)`.

In this mode of traversal, you would visit the left sub-tree, the parent node, and finally the right sub-tree.

The recursive function to return an in-order listing of nodes in a tree is as follows:

```
def inorder(self, root_node):
    current = root_node
    if current is None:
        return
    self.inorder(current.left_child)
    print(current.data)
    self.inorder(current.right_child)
```

We visit the node by printing the node and making two recursive calls with `current.left_child` and `current.right_child`.

Pre-order traversal and prefix notation

Prefix notation is commonly referred to as Polish notation. Here, the operator comes before its operands, as in `+ 3 4`. Since there is no ambiguity of precedence, parentheses are not required: `* + 4 5 - 5 3`.

To traverse a tree in pre-order mode, you would visit the node, the left sub-tree, and the right sub-tree node, in that order.

The recursive function for this traversal is as follows:

```
def preorder(self, root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    self.preorder(current.left_child)
    self.preorder(current.right_child)
```

Note the order in which the recursive call is made.

Post-order traversal and postfix notation.

Postfix or **reverse Polish notation** (**RPN**) places the operator after its operands, as in `3 4 +`. As is the case with Polish notation, there is never any confusion over the precedence of operators, so parentheses are never needed: `4 5 + 5 3 - *`.

In this mode of traversal, you would visit the left sub-tree, the right sub-tree, and lastly the root node.

The `post-order` method is as follows:

```
def postorder(self, root_node):
    current = root_node
    if current is None:
        return
    self.postorder(current.left_child)
    self.postorder(current.right_child)

    print(current.data)
```

2.8.4.2 Breadth-first traversal

This kind of traversal starts from the root of a tree and visits the node from one level of the tree to the other. This mode of traversal is made possible by using a queue data structure. Starting with the root node, we push it into a queue. The node at the front of the queue is accessed (dequeued) and either printed and stored for later use. The left node is added to the queue followed by the right node. Since the queue is not empty, we repeat the process.

The algorithm is as follows:

```
from collections import deque
class Tree:
    def breadth_first_traversal(self):
        list_of_nodes = []
        traversal_queue = deque([self.root_node])
```

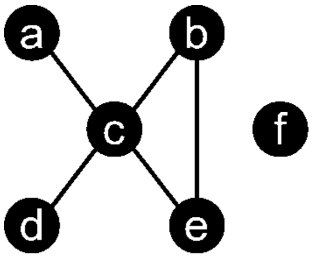
We enqueue the root node and keep a list of the visited nodes in the `list_of_nodes` list. The `dequeue` class is used to maintain a queue:

```
while len(traversal_queue) > 0:
    node = traversal_queue.popleft()
    list_of_nodes.append(node.data)

    if node.left_child:
        traversal_queue.append(node.left_child)

    if node.right_child:
        traversal_queue.append(node.right_child)
return list_of_nodes
```

If the number of elements in the `traversal_queue` is greater than zero, the body of the loop is executed. The node at the front of the queue is popped off and appended to the `list_of_nodes` list. The first `if` statement will `enqueue` the left child node of the node provided a left node exists. The second `if` statement does the same for the right child node. The `list_of_nodes` is returned in the last statement.

2.9 Graphs in Python

Before we start our possible Python representations of graphs, we want to present some general definitions of graphs and its components. A "graph" consists of "nodes", also known as "vertices". Nodes may or may not be connected with one another. The connecting line between two nodes is called an edge. If the edges between the nodes are undirected, the graph is called an undirected graph. If an edge is directed from one vertex (node) to another, a graph is called a directed graph. A directed edge is called an arc.



2.9.1 Graphs implementation using Dictionaries:

Python has no built-in data type or class for graphs, but it is easy to implement them in Python. One data type is ideal for representing graphs in Python, i.e. dictionaries. The graph in our illustration can be implemented in the following way:

```
graph = { "a" : ["c"],
          "b" : ["c", "e"],
          "c" : ["a", "b", "d", "e"],
          "d" : ["c"],
          "e" : ["c", "b"],
          "f" : []
        }
```

The keys of the dictionary above are the nodes of our graph. The corresponding values are lists with the nodes, which are connecting by an edge. There is no simpler and more elegant way to represent a graph. An edge can be seen as a 2-tuple with nodes as elements, i.e. ("a","b").

Function to generate the list of all edges:

```
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append((node, neighbour))

    return edges

print(generate_edges(graph))
```

This code generates the following output, if combined with the previously defined graph dictionary:

```
[('a', 'c'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('c', 'e'), ('b', 'c'), ('b', 'e'), ('e', 'c'), ('e', 'b'), ('d',
'c')]
```

As we can see, there is no edge containing the node "f". "f" is an isolated node of our graph. The following Python function calculates the isolated nodes of a given graph:

```
def find_isolated_nodes(graph):
    """ returns a list of isolated nodes. """
    isolated = []
    for node in graph:
        if not graph[node]:
            isolated += node
    return isolated
```

To generate the path from one node to the other node: Using Python dictionary, we can find the path from one node to the other in a Graph. The idea is similar to DFS in graphs. In the function, initially, the path is an empty list. In the starting, if the start node matches with the end node, the function will return the path. Otherwise the code goes forward and hits all the values of the starting node and searches for the path using recursion. If there is no such path, it returns “None”.

```
# Python program to generate the first
# path of the graph from the nodes provided

graph = {
        'a': ['c'],
        'b': ['d'],
        'c': ['e'],
        'd': ['a', 'd'],
        'e': ['b', 'c']
}

# function to find path
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath:
                return newpath
            return None


# Driver function call to print the path
print(find_path(graph, 'd', 'c'))
```

Program to generate all the possible paths from one node to the other: In the above discussed program, we generated the first possible path. Now, let us generate all the possible paths from the start node to the end node. The basic functioning works same as the functioning of the above code. The place where the difference comes is instead of instantly returning the first path, it saves that path in a list named as ‘paths’ in the example given below. Finally, after iterating over all the possible ways, it returns the list of paths. If there is no path from the starting node to the ending node, it returns “None”.

```python
# Python program to generate the all possible
# path of the graph from the nodes provided
graph ={
'a':['c'],
'b':['d'],
'c':['e'],
'd':['a', 'd'],
'e':['b', 'c']
}

# function to generate all possible paths
def find_all_paths(graph, start, end, path =[]):
  path = path + [start]
  if start == end:
    return [path]
  paths = []
  for node in graph[start]:
    if node not in path:
      newpaths = find_all_paths(graph, node, end, path)
    for newpath in newpaths:
      paths.append(newpath)
  return paths

# Driver function call to print all
# generated paths
print(find_all_paths(graph, 'd', 'c'))
```

Program to generate the shortest path: To get to the shortest from all the paths, we use a little different approach as shown below. In this, as we get the path from the start node to the end node, we compare the length of the path with a variable named as shortest which is initialized with the "None" value. If the length of generated path is less than the length of shortest, if shortest is not "None", the newly generated path is set as the value of shortest. Again, if there is no path, it returns "None".

```python
# Python program to generate shortest path

graph ={
'a':['c'],
'b':['d'],
'c':['e'],
'd':['a', 'd'],
'e':['b', 'c']
}

# function to find the shortest path
def find_shortest_path(graph, start, end, path =[]):
        path = path + [start]
        if start == end:
            return path
        shortest = None
        for node in graph[start]:
            if node not in path:
                newpath = find_shortest_path(graph, node, end, path)
                if newpath:
                    if not shortest or len(newpath) < len(shortest):
                        shortest = newpath
        return shortest

# Driver function call to print
# the shortest path
print(find_shortest_path(graph, 'd', 'c'))
```

2.10 Lab Tasks

Exercise 2.1.

Complete the following code which will perform a selection sort in Python. ”...” denotes missing code that should be filled in:

```
def selection_sort(items):
    """Sorts a list of items into ascending order using the
       selection sort algoright.
       """
    for step in range(len(items)):
        # Find the location of the smallest element in
        # items[step:].
        location_of_smallest = step
        for location in range(step, len(items)):
            # TODO: determine location of smallest
            ...
        # TODO: Exchange items[step] with items[location_of_smallest]
        ...
```

Exercise 2.2.

Write a Python function that implements merge sort. It may help to write a separate function which performs merges and call it from within your merge sort implementation.

Exercise 2.3.

Consider the list of characters: [‘P’,‘Y’,‘T’,‘H’,‘O’,‘N’]. Show how this list is sorted using the following algorithms:

- bubble sort
- selection sort
- insertion sort
- merge sort

Exercise 2.4.

Write a function to find mean, median, mode for the given set of numbers in a list.

1. Read the elements into a list.
2. Calculate the sum of list elements.
3. Calculate the mean, median.
4. Display the result.

Exercise 2.5.

Write a function dups to find all duplicates in the list.

1. Create a list to read elements.
2. Pass the list as parameter to dup function.
3. Define function dup to identify duplicate elements in the list.
4. Return the final list to called function.
5. Display the result

Exercise 2.6.

Implement the following instructions.

1. Create a list a which contains the first three odd positive integers and a list b which contains the first three even positive integers.
2. Create a new list **c** which combines the numbers from both lists (order is unimportant).
3. Create a new list d which is a sorted copy of c, leaving c unchanged.
4. Reverse d in-place.
5. Set the fourth element of c to 42.
6. Append 10 to the end of d.
7. Append 7, 8 and 9 to the end of c.
8. Print the first three elements of c.
9. Print the last element of d without using its length.
10. Print the length of d.

Exercise 2.7.

1. Create a list a which contains three tuples. The first tuple should contain a single element, the second two elements and the third three elements.
2. Print the second element of the second element of a.
3. Create a list b which contains four lists, each of which contains four elements.
4. Print the last two elements of the first element of b.

Exercise 2.8.

1. Write a program which uses a nested for loop to populate a three-dimensional list representing a calendar: the top-level list should contain a sub-list for each month, and each month should contain four weeks. Each week should be an empty list.
2. Modify your code to make it easier to access a month in the calendar by a human-readable month name, and each week by a name which is numbered starting from 1. Add an event (in the form of a string description) to the second week in July.

Exercise 2.9

Create a list whose elements contain the integers from 100,000 through 300,000, inclusive (be careful! what should the range be?) and set the variable big_list to contain that list.

NOTE: Put the assignment that creates big_list in your python file outside the scope of a function. Do **NOT** assign to big_list in the timer or search functions because it will invalidate the timing and search results.

a. Time how long it takes the search function above to find each of the following in big_list:

   i. element at front (100000)

   ii. element in middle (200000)

   iii. element at back (300000)

   iv. element not in the list (3)

How long does each take?

b. If you repeat the same searches, does it take exactly the same amount of time? Why do you think this is?

c. How would you describe the relationship between the position of the item you are searching for in the list and how long it takes to find that item? (Is the search time independent of the position, or is there some mathematical relationship between the two quantities?)

Creating a list containing a range of elements

```
list(range(0, 10))
```

Linear search function to find a list element equal to the key. Note that the same structure can be used to find a list element having some other property (e.g., is even, odd, positive, negative, etc.).

```
def search(integer_list, key):
  for i in range(0, len(integer_list)):
    if integer_list[i] == key:
      return i
  return None
```

Timing linear search for 700 in list(range(100,1000000)) by using time()

```
import time
def timer():
  start = time.time()
  """
    insert the code to be timed here, e.g. search(list, key)
    Make sure to remove the quote marks!
  """
  end = time.time()
  return end - start
```

LAB 3 Problem Formulation

3.1 Representing Search Problem:

The problem-solving agent performs precisely by defining problems and its several solutions. The main components of problem formulation are as follows:

- **Goal Formulation:** It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure.
- **Initial State:** It is the starting state or initial step of the agent towards its goal.
- **Actions:** It is the description of the possible actions available to the agent.
- **Transition Model:** It describes what each action does.
- **Goal Test:** It determines if the given state is a goal state.
- **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**

3.1.1 Explicit Representation of the Search Graph:

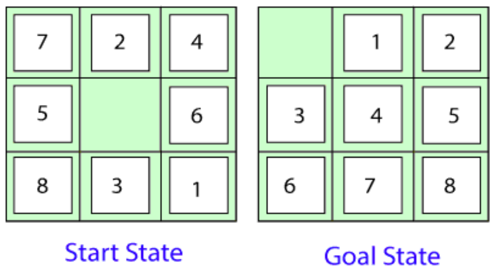
The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed). An **explicit graph** consists of

- a list or set of nodes
- a list or set of arcs
- a start node
- a list or set of goal nodes

To define a search problem, we need to define the start node, the goal predicate, and the neighbors function.

3.1.2 Example Problems

a) **8 Puzzle Problem:**
Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space. The objective is to reach a specified goal state similar to the goal state, as shown in the below figure. In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.



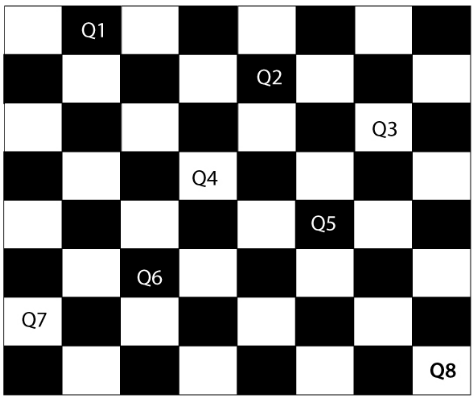
In the above figure, our task is to convert the current (Start) state into goal state by sliding digits into the blank space.

The problem formulation is as follows:

- **States:** It describes the location of each numbered tiles and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
- **Transition Model:** It returns the resulting state as per the given state and actions.
- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

b) **8-queens problem:**

The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.** From the following figure, we can understand the problem as well as its correct solution.



It is noticed from the above figure that each queen is set into the chessboard in a position where no other queen is placed diagonally, in same row or column. Therefore, it is one right approach to the 8-queens problem. For this problem, there are two main kinds of formulation:

- **Incremental formulation:** It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

- **States:** Arrangement of any 0 to 8 queens on the chessboard.
- **Initial State:** An empty chessboard
- **Actions:** Add a queen to any empty box.
- **Transition model:** Returns the chessboard with the queen added in a box.
- **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.
- **Path cost:** There is no need for path cost because only final states are counted.

In this formulation, there is approximately **1.8 x 10^{14}** possible sequence to investigate.

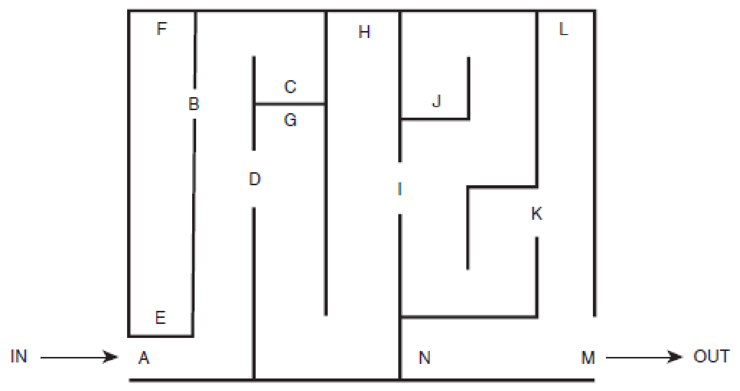
- **Complete-state formulation:** It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

Following steps are involved in this formulation

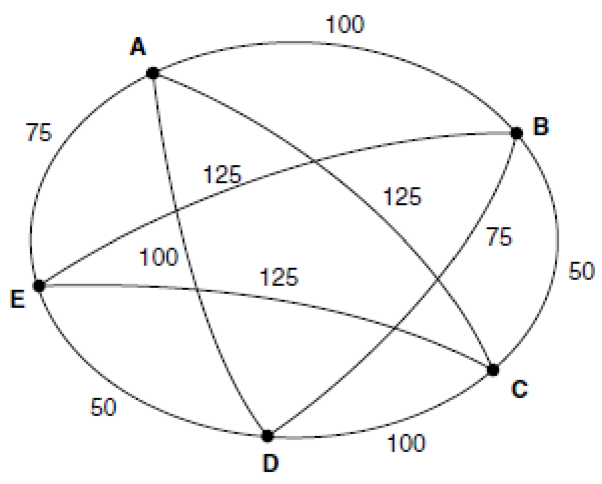
- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.

This formulation is better than the incremental formulation as it reduces the state space from **1.8 x 10^{14}** to **2057**, and it is easy to find the solutions.

c) **Traversing a Maze:**
When traversing a maze, most people will wander randomly, hoping they will eventually find the exit. This approach will usually be successful eventually but is not the most rational and often leads to what we call "going round in circles." This problem, of course, relates to search spaces that contain loops, and it can be avoided by converting the search space into a search tree. An alternative method that many people know for traversing a maze is to start with your hand on the left side of the maze (or the right side, if you prefer) and to follow the maze around, always keeping your left hand on the left edge of the maze wall. In this way, you are guaranteed to find the exit.



d) **Traveling salesperson problem (TSP):**
It is a **touring problem** where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city. Suppose a salesperson has five cities to visit and then must return home. The goal of the problem is to find the shortest path for the salesperson to travel, visiting each city, and then returning to the starting city. The nodes of the graph represent cities, and each arc is labeled with a weight indicating the cost of traveling that arc. This cost might be a representation of the miles necessary in car travel or cost of an air flight between the two cities.


A
B
C
D
E
100
75
125
125
75
50
100
125
50
100

3.1.3 Code Structure:

```
#
class Problem:
    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
    __init__, goal_test, and path_cost. Then you will create instances
    of your subclass and solve them with the various search functions."""

    def __init__(self, initial, goal=None):
        """The constructor specifies the initial state, and possibly a goal
        state, if there is a unique goal. Your subclass's constructor can add
        other arguments."""
        self.initial = initial
        self.goal = goal

    def actions(self, state):
        """Return the actions that can be executed in the given
        state. The result would typically be a list, but if there are
        many actions, consider yielding them one at a time in an
        iterator, rather than building them all at once."""
        raise NotImplementedError

    def result(self, state, action):
        """Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state)."""
        raise NotImplementedError

    def goal_test(self, state):
        """Return True if the state is a goal. The default method compares the
        state to self.goal or checks for state in self.goal if it is a
        list, as specified in the constructor. Override this method if
        checking against a single self.goal is not enough."""
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Return the cost of a solution path that arrives at state2 from
        state1 via action, assuming cost c to get up to state1. If the problem
        is such that the path doesn't matter, this function will only look at
        state2. If the path does matter, it will consider c and maybe state1
        and action. The default method costs 1 for every step in the path."""
        return c + 1

    def value(self, state):
        """For optimization problems, each state has a value.
        Some algorithms try to maximize this value."""
        raise NotImplementedError
#
```

3.2 Lab Tasks

Exercise 3.1.

In this task you will consider 8-puzzle problem. The 8-puzzle problem is a puzzle played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in specified order. You are permitted to slide blocks horizontally or vertically into the blank square.

3	4	1
2	5	6
7	8	

Initial State (Randomly Chosen)

1	2	3
4	5	6
7	8	

Goal State

Create a State Space for 8-puzzle problem. Define it in such a way that it may act as a tree.

Exercise 3.2.

Design a decision tree that enables you to identify an item from a category in which you are interested (e.g., cars, animals, pop singers, films, etc.).

Exercise 3.3.

Devise your own representation for the Missionaries and Cannibals problem and implement it. Use it to solve the problem. How efficient is your representation

Exercise 3.4.

Design a suitable representation and draw the complete search tree for the following problem: A farmer is on one side of a river and wishes to cross the river with a wolf, a chicken, and a bag of grain. He can take only one item at a time in his boat with him. He can't leave the chicken alone with the grain, or it will eat the grain, and he can't leave the wolf alone with the chicken, or the wolf will eat the chicken. How does he get all three safely across to the other side?

LAB 4 Uninformed Search – I

In Lab-03, we introduced methods and representations that are used for solving problems using Artificial Intelligence techniques such as search. In this Lab, we will implement an Uninformed Search Algorithm to search for solving a particular problems.

4.1 Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed.

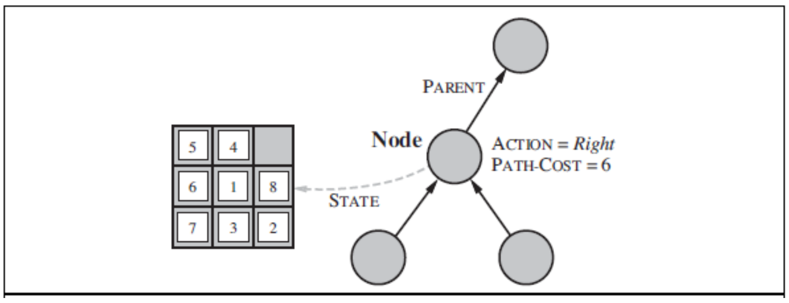


Figure 4-1. Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

For each node n of the tree, we have a structure that contains four components:

- `n.STATE`: the state in the state space to which the node corresponds;
- `n.PARENT`: the node in the search tree that generated this node;
- `n.ACTION`: the action that was applied to the parent to generate the node;
- `n.PATH-COST`: the cost, traditionally denoted by `g(n)`, of the path from the initial state to the node, as indicated by the parent pointers.

4.2 Breadth First Search:

A pseudocode implementation of breadth-first search is given below. BFS is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. In Breadth-first search the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is *generated* rather than when it is selected for expansion.

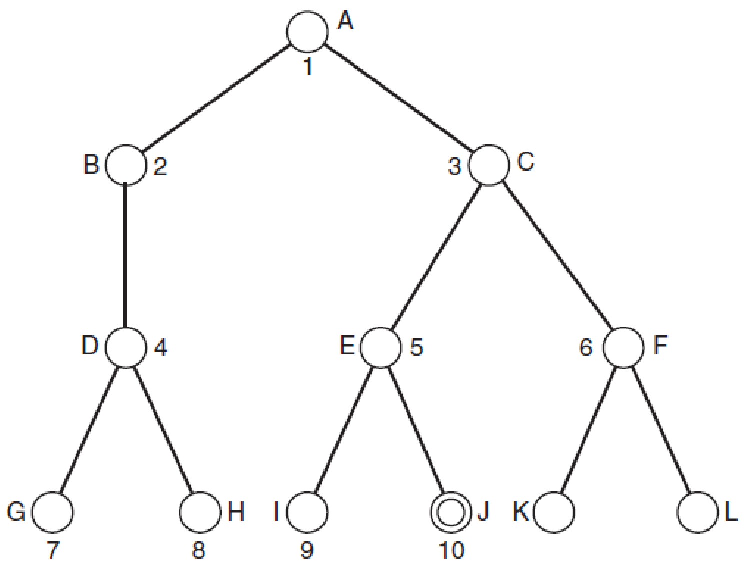


Figure 4-2 Illustration of Breadth First Search

```
Function breadth ()
{
     queue = [];       // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
{
          if is_goal (state)
               then return SUCCESS
          else add_to_back_of_queue (successors (state));
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

Figure 4-3 Breadth First Search Algorithm Pseudocode

4.3 Depth-First Search:

Depth-first search always expands the *deepest* node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors. Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

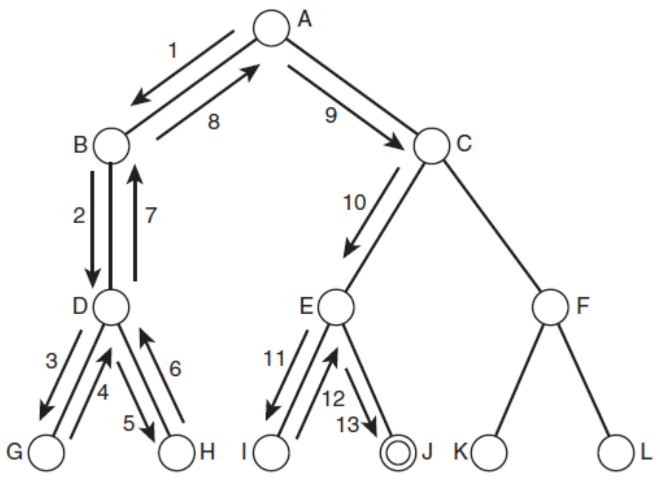


Figure 4-4 Illustration of Depth First Search

```
Function depth ()
{
      queue = [];      // initialize an empty queue
      state = root_node;    // initialize the start state
      while (true)
      {
            if is_goal (state)
                  then return SUCCESS
            else add_to_front_of_queue (successors (state));
            if queue == []
                  then report FAILURE;
            state = queue [0]; // state = first item in queue
            remove_first_item_from (queue);
      }
}
```

Figure 4-5 Depth First Search Algorithm Pseudocode

4.4 Code Structure:

```
class Node:
    """A node in a search tree. Contains a pointer to the parent (the node
    that this is a successor of) and to the actual state for this node. Note
    that if a state is arrived at by two paths, then there are two nodes with
    the same state. Also includes the action that got us to this state, and
    the total path_cost (also known as g) to reach the node. Other functions
    may add an f and h value; see best_first_graph_search and astar_search for
    an explanation of how the f and h values are handled. You will not need to
    subclass this class."""

    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, derived from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

    def __repr__(self):
        return "<Node {}>".format(self.state)

    def __lt__(self, node):
        return self.state < node.state

    def expand(self, problem):
        """List the nodes reachable in one step from this node."""
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]


    def child_node(self, problem, action):
        """[Figure 3.10]"""
        next_state = problem.result(self.state, action)
        next_node = Node(next_state, self, action, problem.path_cost(self.path_cost,
        self.state, action, next_state))
        return next_node

    def solution(self):
        """Return the sequence of actions to go from the root to this node."""
        return [node.action for node in self.path()[1:]]

    def path(self):
        """Return a list of nodes forming the path from the root to this node."""
        node, path_back = self, []
        while node:
            path_back.append(node)
            node = node.parent
        return list(reversed(path_back))
```

```python
    # We want for a queue of nodes in breadth_first_graph_search or
    # astar_search to have no duplicated states, so we treat nodes
    # with the same state as equal. [Problem: this may not be what you
    # want in other contexts.]

    def __eq__(self, other):
        return isinstance(other, Node) and self.state == other.state

    def __hash__(self):
        # We use the hash value of the state
        # stored in the node instead of the node
        # object itself to quickly search a node
        # with the same state in a Hash Table
        return hash(self.state)
# ______________________________________________________________________________
```

4.4.1 Breadth First Search – Sample Code:

```python
# ______________________________________________________________________________
# Uninformed Search algorithms

def breadth_first_tree_search(problem):
    """
    Search the shallowest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = deque([Node(problem.initial)])  # FIFO queue

    while frontier:
        node = frontier.popleft()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
# ______________________________________________________________________________
```

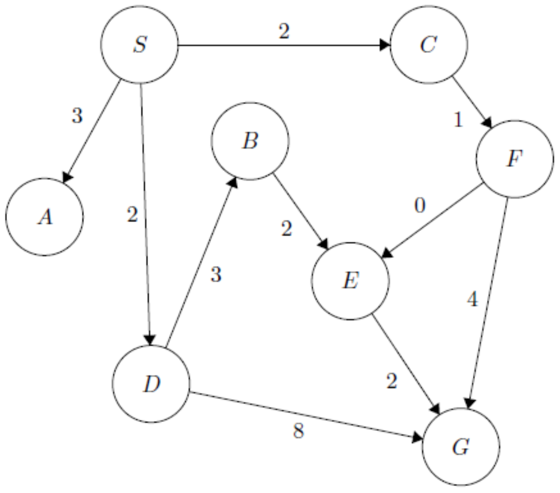
4.4.2 Depth First Search – Sample Code:

```python
# ______________________________________________________________________________
def depth_first_tree_search(problem):
    """
    Search the deepest nodes in the search tree first.
    Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Repeats infinitely in case of loops.
    """

    frontier = [Node(problem.initial)]  # Stack

    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None
# ______________________________________________________________________________
```

4.5 Lab Tasks

Exercise 4.1.

Using Breadth First Search (BFS) algorithm, write out the order in which nodes are added to the explored set, with **start state S** and **goal state G**. Break ties in alphabetical order. Additionally, what is the path returned by each algorithm? What is the total cost of each path?



Exercise 4.2.

The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. Implement and solve the problem optimally using following search algorithm. Is it a good idea to check for repeated states?

a. BFS
b. DFS

Exercise 4.3.

In Figure 1 you will find a matrix (list of lists) that represents a simple labyrinth. 0 represents a wall and 1 represents a passage. 2 represents a door that requires a key to be opened and 3 represents a supply of keys.

```
labyrinth = \
    [[0,0,0,0,0,0,1,0],
     [0,1,0,1,1,1,1,0],
     [0,1,1,1,0,1,0,0],
     [0,1,0,0,0,0,0,0],
     [0,1,1,0,1,1,3,0],
     [0,0,1,1,1,0,0,0],
     [0,1,2,0,1,1,1,0],
     [0,1,0,0,0,0,0,0]]
```

```
############. ##
##  ##. . . . ##
##. . . ##  ####
##. ############
##. . ##. . .K##
####. . . ######
##. .D##      ##
##. ############
```

a. Write a function that takes a labyrinth matrix like the one shown above as argument and draws it using the characters `##` for walls, two spaces for empty passages, `D` for doors and `K` for keys. This function should also accept a list of `(x,y)` tuples that represent positions visited by a robot in the labyrinth, you should print the character . at these points (note that you should print either a whitespace, D or K in addition to the dot).

b. Write a function that takes a labyrinth matrix like the one above and the x and y coordinates for a place in the labyrinth as arguments, and returns a list of the coordinates (tuples of x and y) of all adjacent places (vertically and horizontally) that are passages (i.e. have value 1). Note that the function must handle all input coordinates in a consistent way, and not access memory outside the size of the labyrinth array under any circumstance.

c. Write a recursive function that finds a way through the labyrinth. The entrance is at (6,0) the exit at (1,15) for the big labyrinth. The function should effectively perform a depth-first search. The base case is obviously that the goal is reached, and should return the path leading there. Therefore, you need to keep track of the path traversed so far. In the recursive case, you need to try all possible ways you can take the next step, except for those cases when you return to a state previously visited.

LAB 5 Uninformed Search – II

5.1 Uniform Cost Search:

Instead of expanding the shallowest node as in BFS, **uniform-cost search** expands the node n with the *lowest path cost* g(n). This is done by storing the frontier as a priority queue ordered by g. The algorithm is shown below. At each stage, the path that has the lowest cost so far is extended. In this way, the path that is generated is likely to be the path with the lowest overall cost.

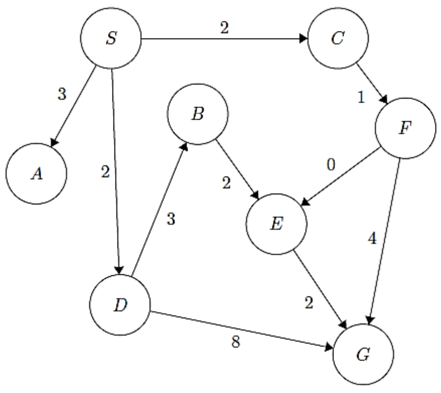
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Figure 5-1. Uniform-cost search on a graph use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered.

5.2 Lab Tasks

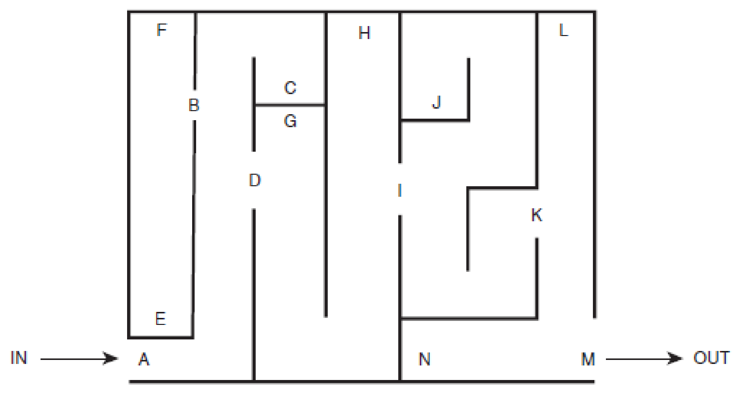
Exercise 5.1.

Modify the code of Breadth First Search implemented in LAB 04 to Uniform Cost Search. Display the order in which nodes are added to the explored set, with **start state S** and **goal state G**. Path costs are mentioned on the arcs. Break ties in alphabetical order. What is the total cost of path found using UCS?



Exercise 5.2.

An alternative method that many people know for traversing a maze is to start with your hand on the left side of the maze (or the right side, if you prefer) and to follow the maze around, always keeping your left hand on the left edge of the maze wall. In this way, you are guaranteed to find the exit. Implement the UCS algorithm to find the shortest path in the following maze.



LAB 6 Informed Search

In **informed search** strategy we use problem-specific knowledge beyond the definition of the problem itself so that we can find solutions more efficiently than can an uninformed strategy.

6.1 Greedy Best First Search

In Best-first search algorithm a node is selected for expansion based on an **evaluation function**, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. **Greedy best-first search** tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

```
Function best ()
{
    queue = [];     // initialize an empty queue
    state = root_node;   // initialize the start state
    while (true)
    {
        if is_goal (state)
            then return SUCCESS
        else
        {
            add_to_front_of_queue (successors (state));
            sort (queue);
        }
        if queue == []
            then report FAILURE;
        state = queue [0]; // state = first item in queue
        remove_first_item_from (queue);
    }
}
```

6.2 A* Search

The most widely known form of best-first search is called **A* search**. It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

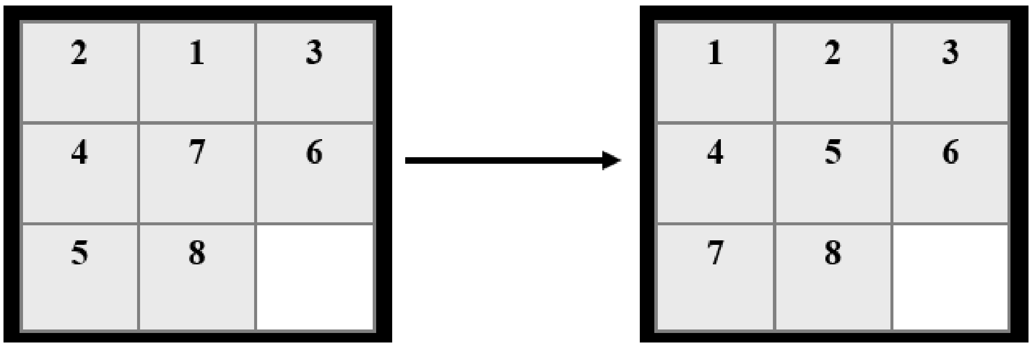
$$f(n) = g(n) + h(n)$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n. Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n).

6.3 Lab Tasks

Exercise 6.1.

Using state representation, goal_test() and operators from previous labs, implement Greedy Best First Search to solve *8-Puzzle* problem. You should write a function *GreedySearch(state)* that accepts any initial state and returns the result.



The first heuristic we consider is to count how many tiles are in the wrong place. We will call this heuristic, $h_1(n)$. An improved heuristic, $h_2(n)$, takes into account how far each tile had to move to get to its correct state. This is achieved by summing the **Manhattan distances** of each tile from its correct position. (Manhattan distance is the sum of the horizontal and vertical moves that need to be made to get from one position to another). Implement both the heuristic to solve the given 8-puzzle problem.

Exercise 6.2.

Solve above problem using A* Search algorithm.

LAB 7 Hill Climbing Search

In examining a search tree, hill climbing will move to the first successor node that is “better” than the current node—in other words, the first node that it comes across with a heuristic value lower than that of the current node. Now hill climbing proceeds as with depth-first search, but at each step, the new nodes to be added to the queue are sorted into order of distance from the goal.

7.1 Hill Climbing Algorithm

```
Function hill ()
{
     queue = [];     // initialize an empty queue
     state = root_node;   // initialize the start state
     while (true)
     {
          if is_goal (state)
               then return SUCCESS
          else
          {
               sort (successors (state));
               add_to_front_of_queue (successors (state));
          }
          if queue == []
               then report FAILURE;
          state = queue [0]; // state = first item in queue
          remove_first_item_from (queue);
     }
}
```

7.1.1 Simple Hill Climbing

Simple hill climbing is the simplest way to implement a hill-climbing algorithm. It only evaluates the neighbour node state at a time and selects the first one which optimizes current cost and set it as a current state. It only checks it’s one successor state, and if it finds better than the current state, then move else be in the same state.

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
  - i. If it is goal state, then return success and quit.
  - ii. else if it is better than the current state then assign new state as a current state.
  - iii. else if not better than the current state, then return to step 2.
- **Step 5:** Exit.

7.1.2 Steepest-Ascent hill climbing

The steepest-Ascent algorithm is a variation of the simple hill-climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors.

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make the current state as your initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
  i. Let **S** be a state such that any successor of the current state will be better than it.
  ii. For each operator that applies to the current state;
     - Apply the new operator and generate a new state.
     - Evaluate the new state.
     - If it is goal state, then return it and quit, else compare it to the **S**.
     - If it is better than **S**, then set new state as **S**.
     - If the **S** is better than the current state, then set the current state to **S**.
- **Step 5:** Exit.

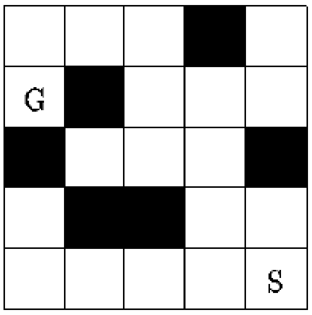
7.1.3 Stochastic hill climbing

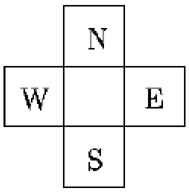
Stochastic hill climbing does not examine for all its neighbors before moving. Rather, this search algorithm selects one neighbor node at random and evaluate it as a current state or examine another state.

7.2 Lab Tasks

Exercise 7.1.

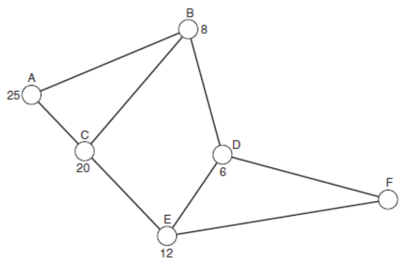
Consider the following maze. The problem is to get from the start node S to the goal node G, by moving horizontally and vertically and avoiding the black obstacles in the above maze. Implement the two variants of hill climbing algorithm discussed above.





Exercise 7.2. ***Travelling Salesman Problem***

Suppose a TCS delivery boy has to deliver parcels from *Head Office (MM Alam Road)* to 7 different locations in Lahore *(Johar Town, Shahdara, DHA Phase 6, Wapda Town, Askari 10, Allama Iqbal Town, Mall Road)* and then return back to the *Head Office.* He wants to find the route with least travelling distance. Can you help him with that using Hill Climbing Algorithm?



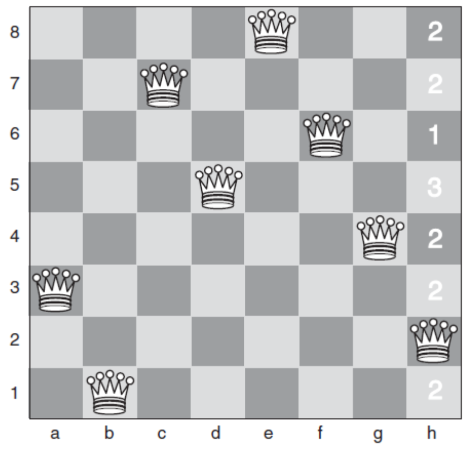
[You can construct distance matrix using google maps OR you can take random values (between 0-50) for distances between any two spots.]

Exercise 7.3.

Consider the function $f(x) = x^2 + 3x + 5$ defined on integer numbers in the interval $[-20,20]$. Use the hill-climbing algorithm to find the function's minimum value on this interval.

Exercise 7.4.

The eight queen puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.



a. Evaluate the heuristics for this problem.
b. Solve the problem using Hill Climbing Search algorithm

LAB 8 Adversarial Search – I

In this lab we consider two-player zero-sum games. Here a player only wins when another player loses. This can be modeled as where there is a single utility which one agent (the maximizing agent) is trying minimize and the other agent (the minimizing agent) is trying to minimize.

A game can be formally defined as a kind of search problem with the following elements:

- S0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. A **zero-sum game** is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game.

8.1 Minimax Algorithm:

The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. The minimax algorithm performs a complete depth-first exploration of the game tree.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

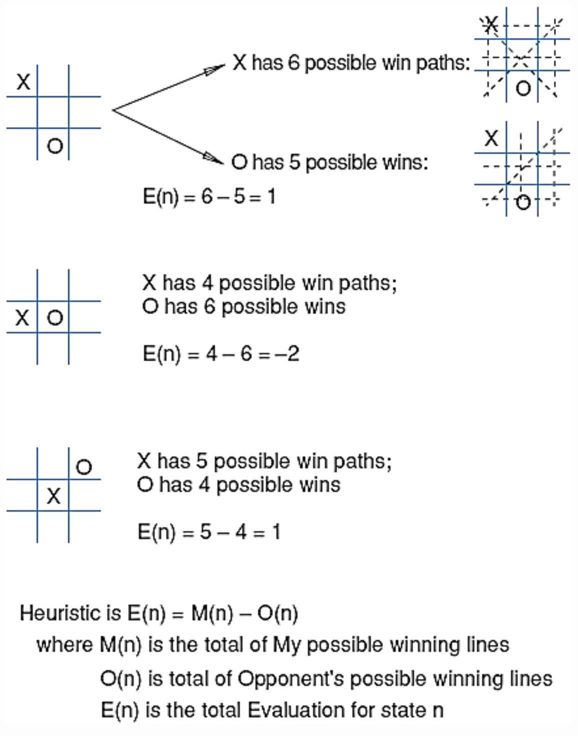
8.1.1 TIC-TAC-TOE Game:

A Tic-Tac-Toe is a 2 player game, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. The game is to be played between two people (in this program between HUMAN and COMPUTER). One of the player chooses 'O' and the other 'X' to mark their respective cells. The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X'). If no one wins, then the game is said to be draw. In our program the moves taken by the computer and the human are chosen randomly. We use rand () function for this.

0	X	0
0	X	X
X	0	X

8.1.2 Bounded Look ahead

Minimax, as we have defined it, is a very simple algorithm and is unsuitable for use in many games where the game tree is extremely large. The problem is that in order to run Minimax, the entire game tree must be examined, this is not possible due to the potential depth of the tree and the large branching factor. In such cases, **bounded look ahead** is very commonly used and can be combined with Minimax. The idea of bounded look ahead is that the search tree is only examined to a particular depth. All nodes at this depth are considered to be leaf nodes and are evaluated using a static evaluation function. In other words, the suggestion is to alter minimax in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL.


X has 6 possible win paths:
O has 5 possible wins:
E(n) = 6 – 5 = 1
X has 4 possible win paths;
O has 6 possible wins
E(n) = 4 – 6 = –2
X has 5 possible win paths;
O has 4 possible wins
E(n) = 5 – 4 = 1
Heuristic is E(n) = M(n) – O(n)
where M(n) is the total of My possible winning lines
O(n) is total of Opponent's possible winning lines
E(n) is the total Evaluation for state n

8.2 Lab Tasks

Exercise 8.1. Tic-Tac-Toe

Formulate the problem.

Exercise 8.2. Tic-Tac-Toe

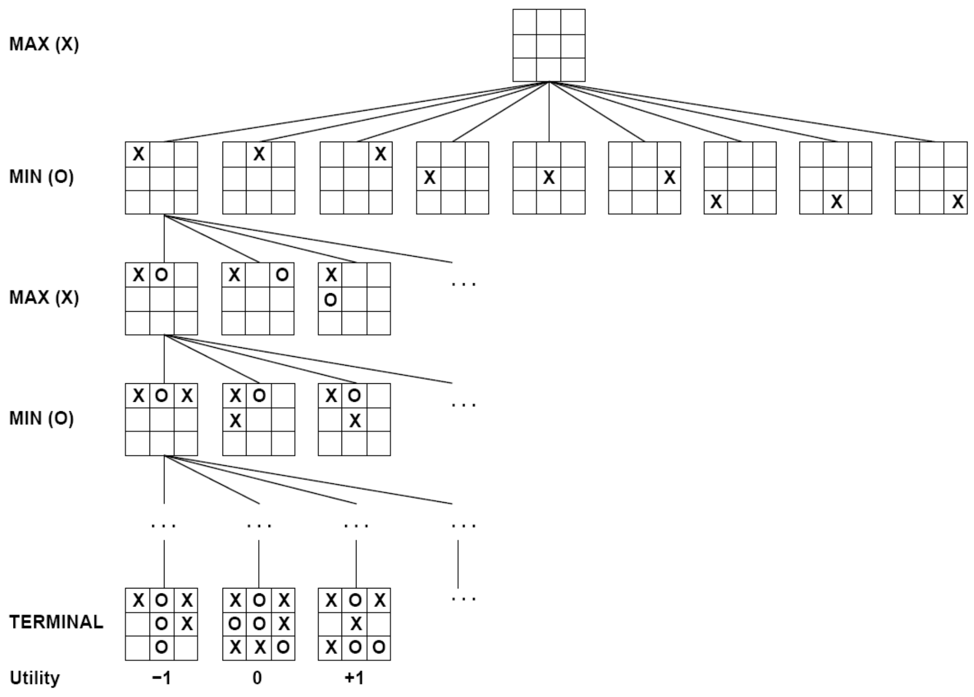
Implement the Minimax algorithm to rank the score of each possible move that an agent can do.

Exercise 8.3. Tic-Tac-Toe

Implement Minimax algorithm to solve the game. It should have a look-ahead of at least 3 half moves (ply's). In case of multiple moves giving same score a random choice should be made.

Exercise 8.4. Tic-Tac-Toe

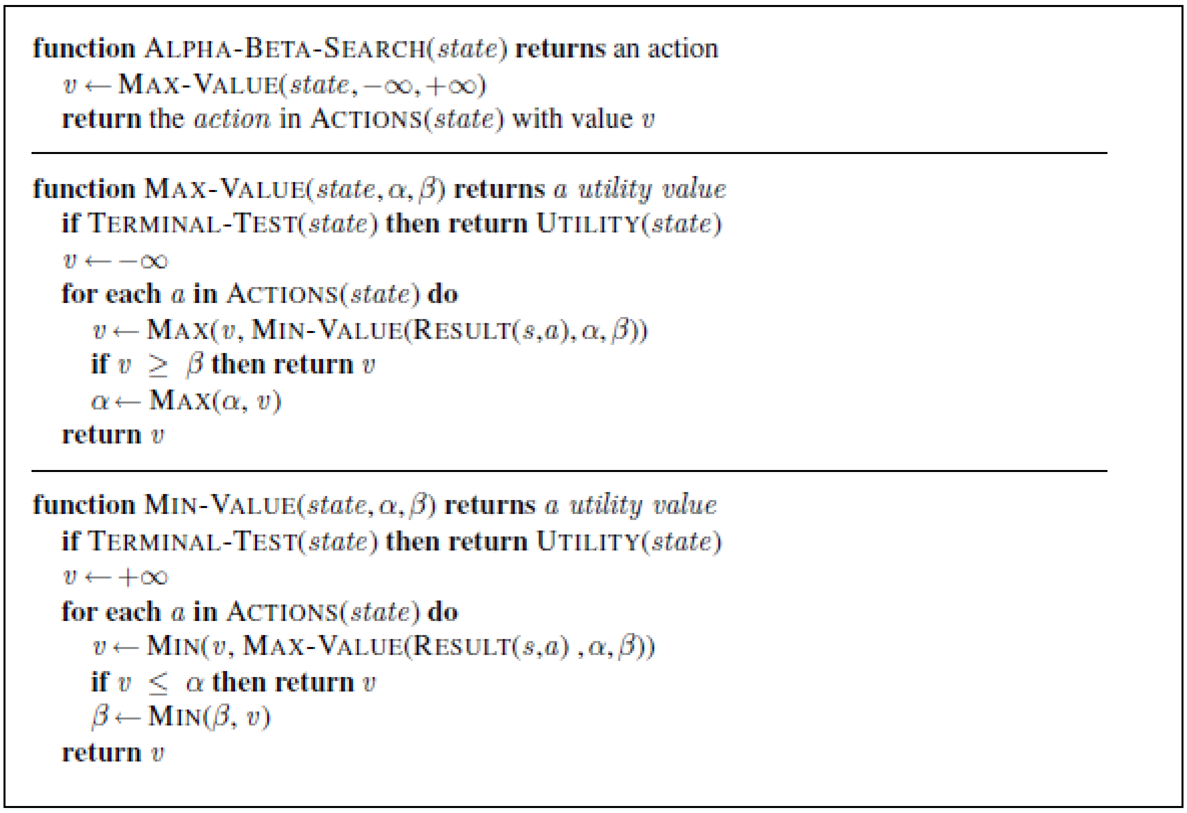
You should make your agent more intelligent by employing a heuristic value that rewards each scenario. For this evaluate the relevant heuristic and implement the minimax algorithm with defined heuristic values.



LAB 9 Adversarial Search – II

9.1 Alpha-Beat Pruning:

In some cases, it is extremely useful to be able to **prune** sections of the game tree. Using alpha–beta pruning, it is possible to remove sections of the game tree that are not worth examining, to make searching for a good move more efficient. The principle behind alpha–beta pruning is that if a move is determined to be worse than another move that has already been examined, then further examining the possible consequences of that worse move is pointless.



```
function ALPHA-BETA-SEARCH(state) returns an action
  $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
  return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha, \beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$,$a$), $\alpha, \beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE(state, $\alpha, \beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$,$a$) , $\alpha, \beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Figure 9-1. The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

9.2 Lab Tasks

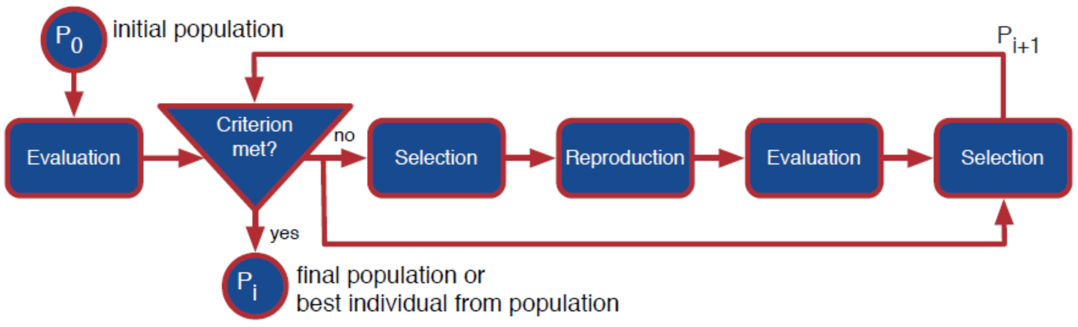
Exercise 9.1.

Modify the program to implement Tic-Tac-Toe game from LAB 08 to incorporate alpha-beta pruning.

LAB 10 Genetic Algorithm

10.1 Genetic Algorithm

Genetic algorithms are usually used to identify optimal solutions to complex problems. This can clearly be easily mapped to search methods, which are aiming toward a similar goal. Genetic algorithms can thus be used to search for solutions to multi-value problems where the closeness of any attempted solution to the actual solution (**fitness**) can be readily evaluated. In short, a **population** of possible solutions (**chromosomes**) is generated, and a fitness value for each chromosome is determined. This fitness is used to determine the likelihood that a given chromosome will survive to the next generation, or reproduce. Reproduction is done by applying **crossover** to two (or more) chromosomes, whereby features (**genes**) of each chromosome are combined together. Mutation is also applied, which involves making random changes to particular genes.



10.2 Lab Tasks

Exercise 10.1.

Consider the problem of maximizing the function

$$f(x) = \frac{-x^2}{10} + 3x$$

where x is allowed to vary between 0 and 31. You must perform following tasks in the code.

a. Representation of states (solutions): To solve this using a genetic algorithm, we must encode the possible values of x as chromosomes. For this problem, we will encode x as a binary integer of length 5. Thus the chromosomes for our genetic algorithm will be sequences of 0's and 1's with a length of 5 bits, and have a range from 0 (00000) to 31 (11111).

b. Fitness function:

The fitness function for it will be:

$$f(x) = \frac{-x^2}{10} + 3x$$

To begin the algorithm, we select an initial population of 10 chromosomes at random. The resulting initial population of chromosomes is shown in Table 1. Next we take the x-value that each chromosome represents and test its fitness with the fitness function. The resulting fitness values are recorded in the third column of Table 1.

Chromosome Number	Initial Population	x-Value	Fitness Value $f(x)$	Selection Probability
1	0 1 0 1 1	11	20.9	0.1416
2	1 1 0 1 0	26	10.4	0.0705
3	0 0 0 1 0	2	5.6	0.0379
4	0 1 1 1 0	14	22.4	0.1518
5	0 1 1 0 0	12	21.6	0.1463
6	1 1 1 1 0	30	0	0
7	1 0 1 1 0	22	17.6	0.1192
8	0 1 0 0 1	9	18.9	0.1280
9	0 0 0 1 1	3	8.1	0.0549
10	1 0 0 0 1	17	22.1	0.1497

c. **Operators:**
   i. Apply cross over in every generation.
   ii. Apply mutation after every 3 generations.

d. **Termination criteria:** Your loop should stop when the value of one of your candidate's fitness function is greater or equal to 90%.

Exercise 10.2. Travelling Salesman Problem

Suppose a TCS delivery boy has to deliver parcels from *Head Office (MM Alam Road)* to 7 different locations in Lahore *(Johar Town, Shahdara, DHA Phase 6, Wapda Town, Askari 10, Allama Iqbal Town, Mall Road)* and then return back to the *Head Office.* He wants to find the route with least travelling distance. You helped him in finding the route using *Hill Climbing Algorithm*. Now use *Genetic Algorithm* instead of *Hill Climbing* to solve this problem. Design choices should be as per class discussion. [You can construct distance matrix using google maps OR you can take random values (between 0-50) for distances between any two spots.]

Exercise 10.3.

For any given 'target' number, find an expression involving any legal combination of addition(+), subtraction(-), multiplication(*) and division(/) on digits that represents the given target number. For example, for the target 15, "9 + 3 * 2" is a solution."

Exercise 10.4.

n-Queen Problem using Genetic Algorithm.

Exercise 10.5.

You have 10 cards numbered 1 to 10. You have to divide them into two piles so that:

- The sum of the first pile is as close as possible to 36.
- And the product of all in the second pile is as close as possible to 360.

Well, all that is being done is the following :

Loop through the population member's genes

- If the current gene being looked at has a value of 0, the gene is for the sum pile (pile 0), so add to the running calculation
- If the current gene being looked at has a value of 1, the gene is for the product pile (pile 1), so add to the running calculation
- Calculate the overall error for this population member. If this member's geneotype has an overall error of 0.0, then the problem domain has been solved

LAB 11 Propositional Logic

If you use python go to https://github.com/aimacode/aima-python/blob/master/intro.ipynb for instructions and links to the files you need. As always, look at https://github.com/aimacode/aima-python/blob/master/tests/test_logic.py for examples on how to use the code. The code (which includes examples) is at https://github.com/aimacode/aima-python/blob/master/logic.py. [1]

11.1 Propositions

A *propositional statement* or *proposition* is a statement that is unambiguously either true or false in a given context. Examples include: "*Quaid-e-Azam was born in July*" (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 *"Ali is a good guy"*, are not propositional statements.

Following are some basic facts about propositional logic:

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "Where is Ali", "How are you", "What is your name", are not propositions.

11.1.1 Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

i. **Atomic Propositions**

ii. **Compound propositions**

Atomic Proposition: Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false. For example

- 2+2 is 4, it is an atomic proposition as it is a **true** fact.

[1] Notes: The aima.logic package requires that propositional variables (i.e., those whose values must be True or False) begin with an uppercase letter and logic variables (i.e., those that can take on any value) begin with a lowercase letter. See here for a an example of using logic.py on gl. The Python examples below assume you are using Python 3.X and have done "from logic import *". The aima's logic.py package is well commented, so take a look if you have questions. The answer might be in the comments.

- "The Sun is cold" is also a proposition as it is a **false** fact.

Compound proposition: Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives. For example:

- "It is raining today, and street is wet."
- "Ankit is a doctor, and his clinic is in Mumbai."

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:

11.1.2 Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has ∧ connective such as, **P ∧ Q** is called a conjunction. For example “Ali is intelligent and hardworking”. It can be written as,
   P= Ali is intelligent,
   Q= Ali is hardworking. → P∧ Q.
3. **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions. For example: "Qasim is a doctor or Engineer**"**, Here
   P= Qasim is Doctor.
   Q= Qasim is an Engineer, so we can write it as **P ∨ Q**.
4. **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as
   If it is raining, then the street is wet.
   Let P= It is raining, and Q= Street is wet, so it is represented as P → Q
5. **Biconditional:** A sentence such as **P⇔ Q is a Biconditional sentence,** for example If I am breathing, then I am alive
   P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ B

11.1.3 Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called **Truth table**. Following are the truth table for all logical connectives:

11.1.3.1 *Truth table with three propositions:*

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8n Tuples as we have taken three proposition symbols.

P	Q	R	¬R	Pv Q	PvQ→¬R
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

11.1.3.2 *Precedence of connectives:*

Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

Precedence	Operators
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

11.1.4 Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as A⇔B. In below truth table we can see that column for ¬AV B and A→B, are identical hence A is Equivalent to B

A	B	¬A	¬AV B	A→B
T	T	F	**T**	**T**
T	F	F	**F**	**F**
F	T	T	**T**	**T**
F	F	T	**T**	**T**

11.2 Lab Tasks

Exercise 11.1. Checking Validity

Use the functions in aima's logic.py to see which of the following are valid, i.e., true in every model. You will have to

i. convert these sentences to the appropriate string form that the python code uses (see the comments in the code) and
ii. use the `expr()` function in `logic.py` to turn each into an Expr object, and
iii. use the `tt_true()` function to check for validity. Provide text from a session that shows the result of checking the validity of each. We've done the first one as an example.

a. P v ¬P

```
>>> tt_true(expr('P | ~P'))
True
```

b. P → P

c. P→ (P v Q)

d. (P v Q) → P

e. ((A ∧B) →C) ↔ (A → (B → C))

f. ((A →B) →A) →A

Exercise 11.2. Satisfiability

Use the functions in logic.py to see which of the following are are satisfiable. We've done the first one as an example.

a. P ∧ Q

```
>>> dpll_satisfiable(expr('P & Q'))
{P: True, Q: True}
```

b. ALIVE→ ¬DEAD ∧ ¬ALIVE ∧ ¬DEAD

c. P → ¬ P v P

d. ~ (P v ¬ P)

Exercise 11.3. Propositional Consequence

For each of the following entailment relations, say whether or not it is true. The text on the left of the entailment symbol (⊨) represents one or more sentences (separated by commas) that constitute a knowledge base. We've done the first one for you.

a. P ∧ Q ⊨ P

b. P ⊨ P ∧ Q

c. P ⊨ P v Q

d. P ⊨ ¬ ¬ P

e. $P \rightarrow Q \vDash \neg P \rightarrow \neg Q$

f. $\neg P \vDash P \rightarrow Q$

g. $\neg Q \vDash P \rightarrow Q$

h. $P \wedge (P \rightarrow Q) \vDash Q$

i. $(\neg P) \wedge (Q \rightarrow P) \vDash \neg Q$

LAB 12 First Order Logic

First-order logic (FOL) is a formal system used in artificial intelligence for knowledge representation and reasoning. It extends propositional logic by allowing the use of quantifiers and predicates, enabling the expression of more complex statements about objects and their relationships. FOL is powerful for modeling natural language and supports inference and automated reasoning.

FOL finds extensive applications in **artificial intelligence**, particularly in **knowledge representation, expert systems**, automated theorem proving, and **natural language understanding**. It allows AI systems to model complex relationships and properties, making it possible to reason about the knowledge they possess.

The computational complexity of reasoning in FOL can be significant, especially when working with large knowledge bases, posing challenges for real-time applications. Despite these limitations, first-order logic remains a cornerstone of AI, providing a robust framework for representing and reasoning about knowledge in a structured and meaningful way.

Key Components of First-Order Logic

1. **Constants**: Specific objects in the domain (e.g., 1, New York, Elie).
2. **Variables**: Symbols that can represent any object (e.g., x, y, a).
3. **Predicates**: Functions that express properties or relationships (e.g., greaterThan, brother).
4. **Functions**: Mappings from objects to objects (e.g., fatherOf, bestFriend).
5. **Quantifiers**: Symbols that express the quantity of objects (e.g., ∀ for "for all", ∃ for "there exists").
6. **Logical Connectives**
   - **And (∧):** Conjunction operator.
   - **Or (∨):** Disjunction operator.
   - **Not (¬):** Negation operator.
   - **Implies (→):** Implication operator.
   - **Biconditional (↔):** Logical equivalence operator.

Applications of First-Order Logic in AI

- **Knowledge Representation:** FOL is widely used in knowledge-based systems, allowing for the representation of complex relationships and properties.
- **Expert Systems:** Many expert systems use FOL to represent domain knowledge and perform reasoning to provide solutions or recommendations.
- **Automated Theorem Proving:** FOL is used in systems designed to prove mathematical theorems automatically.
- **Natural Language Understanding:** FOL helps in parsing and understanding the semantics of natural language statements, enabling AI systems to process and respond to human language.

Syntax of First-Order Logic

The syntax of FOL consists of well-formed formulas (WFFs) that can be constructed using the components mentioned above. Here are some examples:

- **Atomic Formulas:** These are the simplest forms, consisting of predicates applied to terms.
  Example:* *Loves(Alice, Bob), IsHuman(x)
- **Complex Formulas:** These can be built using logical connectives and quantifiers.
  Example: ∀x (IsHuman(x) → Loves(x, Bob)) means "For all x, if x is a human, then x loves Bob."

Semantics of First-Order Logic

The semantics of FOL involves interpreting the symbols in a way that assigns meanings to them. This typically involves:

- **Domain of Discourse:** The set of objects that the variables can refer to.
- **Interpretation:** A mapping that assigns a specific meaning to the constants, predicates, and functions in the logic.

For example, if the domain is the set of all people, we might interpret:

- Alice as a specific person,
- Loves as a relation that represents love between two people.

Practice Questions:

Part-1

Use the file logics.py uploaded on your portal and implement the following;

a) Write a function AND(p, q) that takes two boolean values p and q, and returns the result of the logical AND operation.
b) Write a function OR(p, q) that takes two boolean values p and q, and returns the result of the logical OR operation.
c) Define predicates P(x) and Q(x) for a set of integers where P(x) is true if x is odd and Q(x) is true if x is positive. Write a function EvaluatePredicates that takes an integer x and evaluates the following statement: [P(x) \lor (¬Q(x))]

Part-2

Use the file logics.py uploaded on your portal and implement the following statments;

a) She is a talented musician but not a disciplined practice.
b) The weather is nice today, but it is not warm enough to go swimming
c) He is a skilled player but not a team leader
d) This book is interesting, but it is not easy to read.
e) If she is a good teacher, then she is respected by her students, but she is not appreciated by her colleagues.
f) He is a successful entrepreneur if and only if he is innovative, but he is not a good manager.
g) If he studies hard, he will pass the exam, but he is not confident about his preparation
h) He will be successful if he is persistent and learns from his mistakes, but he does not take feedback well.

Part-3

A= He is a good boy, B= He is not hard worker, C= He is talented.

a) $(A \wedge B) \rightarrow (C \vee A)$
b) $((A \rightarrow B) \wedge) \wedge (C \leftrightarrow B)$
c) $(\sim A \leftrightarrow B) \vee (C \wedge (A \sim B))$

LAB 13 Bayes Classifier

Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

Task: It is a classification technique based on Bayes" Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as „Naive".

Dataset : Pima-indians-diabetes.csv

It is a classification technique based on Bayes" Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as „Naive".

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:

1) **Handling Of Data:**
   - Load the data from the CSV file and split in to training and test data set.
   - Training data set can be used to by Naïve Bayes to make predictions.
   - And Test data set can be used to evaluate the accuracy of the model.

2) **Summarize Data:**

The summary of the training data collected involves the mean and the standard deviation for each attribute, by class value.

- These are required when making predictions to calculate the probability of specific attribute values belonging to each class value.
- Summary data can be break down into the following sub-tasks:

- **Separate Data By Class**: The first task is to separate the training dataset instances by class value so that we can calculate statistics for each class. We can do that by creating a map of each class value to a list of instances that belong to that class and sort the entire dataset of instances into the appropriate lists.
- **Calculate Mean**:We need to calculate the mean of each attribute for a class value.

The mean is the central middle or central tendency of the data, and we will use it as the middle of our gaussian distribution when calculating probabilities.

- **Calculate Standard Deviation**: We also need to calculate the standard deviation of each attribute for a class value. The standard deviation describes the variation of spread of the data, and we will use it to characterize the expected spread of each attribute in our Gaussian distribution when calculating probabilities.
- **Summarize Dataset**: For a given list of instances (for a class value) we can calculate the mean and the standard deviation for each attribute.
- The zip function groups the values for each attribute across our data instances into their own lists so that we can compute the mean and standard deviation values for the attribute.
- **Summarize Attributes By Class**: We can pull it all together by first separating our training dataset into instances grouped by class. Then calculate the summaries for each attribute.

3) Make Predictions:

- ❖ Making predictions involves calculating the probability that a given data instance belongs to each class,
- ❖ then selecting the class with the largest probability as the prediction.
- ❖ Finally, estimation of the accuracy of the model by making predictions for each data instance in the test dataset.

4) Evaluate Accuracy: The predictions can be compared to the class values in the test dataset and a classification\ accuracy can be calculated as an accuracy ratio between 0& and 100%.

Naïve Bayes Program:

```
import csv
import random
import math
def safe_div(x,y):
    if y == 0:
        return 0
    return x / y
def loadCsv(filename):
       lines = csv.reader(open(filename, "r"))
       dataset = list(lines)
       for i in range(len(dataset)):
                dataset[i] = [float(x) for x in dataset[i]]
       return dataset

def splitDataset(dataset, splitRatio):
       trainSize = int(len(dataset) * splitRatio)
       trainSet = []
    copy = list(dataset)
    while len(trainSet) < trainSize:
            index = random.randrange(len(copy))
            trainSet.append(copy.pop(index))
    return [trainSet, copy]

def separateByClass(dataset):
    separated = {}
    for i in range(len(dataset)):
            vector = dataset[i]
            if (vector[-1] not in separated):
                    separated[vector[-1]] = []
            separated[vector[-1]].append(vector)

    return separated

def mean(numbers):
    return sum(numbers)/float(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)

def summarize(dataset):
    summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(*dataset)]
    del summaries[-1]
    return summaries

def summarizeByClass(dataset):
    separated = separateByClass(dataset)
    summaries = {}
    for classValue, instances in separated.items():
            summaries[classValue] = summarize(instances)
    return summaries

def calculateProbability(x, mean, stdev):
    exponent = math.exp(-(math.pow(x-mean,2)/(2*math.pow(stdev,2))))
    return (1 / (math.sqrt(2*math.pi) * stdev)) * exponent

def calculateClassProbabilities(summaries, inputVector):
    probabilities = {}
    for classValue, classSummaries in summaries.items():
            probabilities[classValue] = 1
            for i in range(len(classSummaries)):
                mean, stdev = classSummaries[i]
                x = inputVector[i]
                probabilities[classValue] *= calculateProbability(x, mean, stdev)
        return probabilities

    def predict(summaries, inputVector):
        probabilities = calculateClassProbabilities(summaries, inputVector)
        bestLabel, bestProb = None, -1
        for classValue, probability in probabilities.items():
            if bestLabel is None or probability > bestProb:
                bestProb = probability
                bestLabel = classValue
        return bestLabel

    def getPredictions(summaries, testSet):
        predictions = []
        for i in range(len(testSet)):
            result = predict(summaries, testSet[i])
            predictions.append(result)

        return predictions

    def getAccuracy(testSet, predictions):
      correct = 0
      for i in range(len(testSet)):
        #print(testSet[i][-1]," ",predictions[i])
        if testSet[i][-1] == predictions[i]:
          correct += 1

        return (correct/float(len(testSet))) * 100.0

    def main():
        filename = 'pima-indians-diabetes.data.csv'
        splitRatio = 0.67
        dataset = loadCsv(filename)
        trainingSet,testSet=splitDataset(dataset, splitRatio) #dividing into training and test data
        #trainingSet = dataset #passing entire dataset as training data
        #testSet=[[8.0,183.0,64.0,0.0,0.0,23.3,0.672,32.0]]
        print('Split {0} rows into train={1} and test={2} rows'.format(len(dataset),
    len(trainingSet), len(testSet)))
        # prepare model
    summaries = summarizeByClass(trainingSet)
    # test model
    predictions = getPredictions(summaries, testSet)
    accuracy = getAccuracy(testSet, predictions)
    print('Accuracy: {0}%'.format(accuracy))

main()
```

Input:

Pima-indians-diabetes.csv

OUTPUT:

```
Split 768 rows into train=576 and test=192 rows
Accuracy: 77.604 %
```

LAB 14 KNN

Write a program to implement k-Nearest Neighbor algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

TASK: The task of this program is to classify the IRIS data set examples by using the k-Nearest Neighbour algorithm. The new instance has to be classified based on its k nearest neighbors.

Dataset: iris.csv

<u>ALGORITHM</u>

Let m be the number of training data samples. Let p be an unknown point.

```
1. Store the training samples in an array of data points arr[]. This means each element of
   this array represents a tuple (x, y).
2. for i=0 to m:
     Calculate Euclidean distance d(arr[i], p).
3. Make set S of K smallest distances obtained. Each of these distances correspond to
   an already classified data point.
4. Return the majority label among S.
```

Implement the KNN algorithm for given dataset. For k = 7 and 11

LAB 15 K-Means

The K-Means algorithm is an unsupervised machine learning algorithm used for clustering. It aims to partition a dataset into K distinct clusters, where each data point belongs to the cluster with the nearest mean (centroid). K-Means is an iterative algorithm that converges to a local optimum.

- **Explanation of how the K-Means algorithm works with an example:**

Step 1: Set the Number of Clusters

Start by selecting the desired number of clusters, denoted as K. This value is determined based on prior knowledge or by using domain expertise or techniques such as the elbow method.

Step 2: Initialize Cluster Centers

Randomly initialize K cluster centers in the feature space. These centers act as the initial centroids for the clusters.

Step 3: Assign Data Points to Clusters

Assign each data point to the cluster whose centroid is closest to it. This assignment is based on a distance metric, typically the Euclidean distance.

Step 4: Update Cluster Centers

Calculate the new centroid for each cluster by computing the mean of all data points assigned to that cluster. This step updates the cluster centers.

Step 5: Repeat Steps 3 and 4

Iteratively repeat steps 3 and 4 until convergence or a maximum number of iterations is reached. Convergence is achieved when the cluster assignments and cluster centers no longer change significantly.

Step 6: Finalize Clustering

The final result of the K-Means algorithm is a set of K clusters, each represented by its centroid. The data points are clustered based on their proximity to the centroids.

- **Example to illustrate the K-Means algorithm:**

Suppose we have a dataset of customer data with two numerical features: annual income and spending score. We want to segment the customers into distinct groups based on their income and spending habits.

- We start by setting the number of clusters, K, to, let's say, 3.
- We randomly initialize three cluster centers in the feature space.
- For each data point in the dataset, we calculate its distance to each of the three cluster centers and assign it to the nearest cluster.
- Once all data points are assigned to clusters, we update the cluster centers by calculating the mean of the data points in each cluster.
- We repeat steps 3 and 4 iteratively, updating the cluster assignments and cluster centers, until convergence.
- When convergence is reached, we have our final clustering result. Each customer is assigned to one of the three clusters based on proximity to the cluster centers.

The K-Means algorithm works well for datasets where the clusters are well-separated and have a spherical shape. However, it may struggle with datasets that contain irregularly shaped or overlapping clusters. It is also sensitive to the initial random assignment of cluster centers, which can result in different clustering results.

It's important to note that the K-Means algorithm is an unsupervised learning algorithm, meaning it doesn't require labeled data. It is solely based on the patterns and structure within the input data.