Introduction to Computer Science and Programming using Python

Status	Completed

Lecture 1: What is Computation?

11/28/23

What does a computer do?

- Fundamentally, a computer performs calculations and remembers results.
- Computers only know what you tell them.

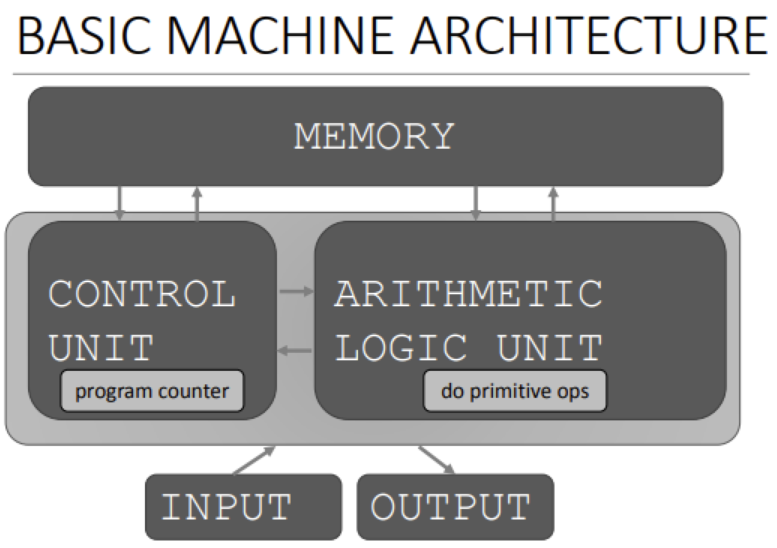
Types of knowledge

- Declarative knowledge is a statement of fact.
- Imperative knowledge is a “how-to”, a sequence of steps.
    - A “how-to” is:
        1. A sequence of steps.
        2. A flow of control.
        3. A means of determining when to stop.
    - This “how-to” is an algorithm.

Computers are machines

- Fixed program computers can only perform calculations.
    - For example, a four function calculator.
- Stored program computers are able to perform calculations, store information, and executer instructions.

Basic computer architecture



- The memory contains data and sequences of instructions.
- The control unit contains the program counter which fetches and executes instructions.
    - The control unit uses control flow to make decisions based on conditionals.
- The arithmetic logic unit (ALU) performs calculations.
- The interpreter executes each instruction in order.
    - It uses tests to change the flow of control through the sequence.
    - The interpreter stops when it is done.

Basic primitives

- Alan Turing showed that you can computer anything using 6 primitive operations.
- These primitives can be abstracted to create new primitives.
- Anything computable in one programming language is computable in any other programming language.

Creating recipes

- A programming language provides a set of primitive operations.
- Expressions are combinations of primitives in a programming language.
- Expressions and computations have values and meanings in programming languages.

Aspects of programming languages

- Programming languages have syntax, just like regular languages.
    - The syntax of a program can either be valid or invalid.
- Static semantics is when syntactically valid strings have meaning as well.
    - A program can have proper syntax, but not have proper semantics.
    - Semantics is the meaning associated with a string with no static semantic errors.

Where things go wrong

- Syntactic errors are common and easily caught.
- Static semantic errors are harder to catch and cause unpredictable behavior.
- Non-syntactic or semantic errors are errors that create a different result than what the programmer intends.
    - The program may run forever, crash, or give a different answer than expected.

Python programs

- A program is a sequence of definitions and commands.
    - Definitions are evaluated.
    - Commands are execute by the Python interpreter in a shell.

Objects

- Programs manipulate data objects.
- Object have a type that defines the kinds of things programs can do to them.
- Objects are either:
    - Scalar: cannot be subdivided.

    ```
    # Integers
    int
    # Real numbers
    float
    # Boolean values, True or False
    bool
    # Absence of a type
    None
    ```

    - Non-scalar: have an internal structure that can be accessed.
- The type of an object can be found using `type()`

```
# Find the type of 3:
type(3)
```

- Some types can be converted to other types using casts:

```
# Convert a float (2.9) to an integer (2)
int(2.9)
# Convert an int (3) to a float (3.0)
float(3)
```

Expressions

- Objects and operators can be combined to form expressions.
- An expression has a value, which has a type.

```
# <object> <operator> <expression>
3 + 2
```

Operator precedence

```
# Operator execution in order without parentheses
# Exponent
**
# Multiplication
*
# Division
/
# Addition and subtraction, executed left to right
+
-
```

Binding variables and values

- An equal sign `=` is an assignment of a value to a variable.
    - The value is stored in computer memory.
- Variables can be re-bound using a new assignment.
    - The previous value may still be stored in memory but the handle for it is lost.

Lecture 2: Branching and Iteration

11/28/23

A large portion of this lecture covers material that was already covered in CS50P. Reference Introduction to Programming, Lectures 0, 1, & 2 for what material is covered.

11/29/23

- The `range()` function allows one to iterate through a list.

```
# The range function allows start, end, and step arguments
# Start at 5, end at 9, and iterate by 2
for i in range(5, 11, 2):
    ...
```

- The `break` keyword only exits the innermost loop.
- `for` loops and `while` loops should be used in different contexts

`for` loops:

- The loops has a known number of iterations.
- The loop can end early with `break`.

`while` loops:

- Loops have an unbounded number of iterations.
- The loop can end early with `break`.

- The loop uses a counter.
- Any `for` loop can be rewritten using a `while` loop.

- The loop can use a counter but the counter must be initialized before the loop and incremented inside the loop.
- A `while` loop can not necessarily be rewritten as a `for` loop.

▼ Lecture 3: String Manipulation, Guess-and-Check, Approximations, Bisection

11/29/23

Strings

- Strings are a sequence of case sensitive characters.
- Strings can be compared using `==`, `>`, `<`, etc.
- Characters of strings can be indexed list any other sequence.

```
# Print certain characters of a string
string = "Hello World!"
# print "l"
print(string[2])
# print "!"
print(string[-1])
# print "d"
print(string[-2])
```

- Strings can be sliced using `[start:stop:step]`:

```
# Slice a string
s = "abcdefgh"
# def
s[3:6]
# df
s[3:6:2]
# abcdefgh (s[0:len(s):1])
s[::]
# hgfedcba (reverse)
s[::-1]
# ec
s[4:1:-2]
```

- Strings are immutable, and therefore cannot be modified.

For loops

- `for` loops have a loop variable that iterates over a set of values.

```
# This
for _ in range(4)
    ...
# is equivalent to this
for _ in [0, 1, 2, 3]
    ...
```

- `for` loops can iterate over any set of values, not just integers.

Guess-and-check algorithm

- This process is also called exhaustive enumeration.
- Given a problem:
    1. Guess a value for a solution.
    2. Check if the solution is correct
    3. Keep guessing systematically until a solution is found or all the possible values in the search space have been guessed.
- For example, find the cube root of a number

```
# Find the cube root of 8
cube = 8
# Guess numbers "guess" from 0 to 8
for guess in range(cube+1):
    # Check if "guess" cubed is 8:
    if guess**3 == cube:
        print("Cube root of", cube, "is", guess)
# If cube is not a perfect cube, say so
if guess**3 != cube
    print(cube, "is not a perfect cube")
```

Approximate solution algorithm

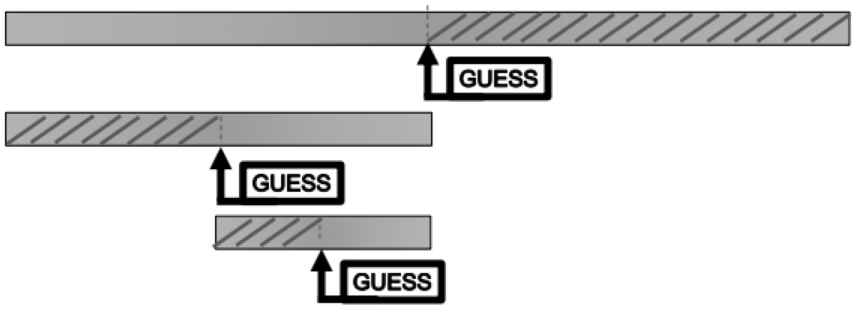
- A good enough solution.
- Given a problem:
    1. Start with a guess and increment by a small value.
    2. Keep guessing if not close enough for some small epsilon.
- A smaller increment size will result in a more accurate but slower program.
- A larger increment size will results in a less accurate but faster program.

```
# Find the cube root of 27
cube = 27
epsilson = 0.01
guess = 0.0
```

```
num_guesses = 0
# Check while the difference between the guess and the cube
# is larger than the epsilon, and smaller than the cube
while abs(guess**3 - cube) >= epsilon and guess <= cube:
    guess += increment
    # Track number of guesses
    num_guesses += 1
# Print number of guesses
print("Number of guesses =", num_guesses)
# If the epsilon is not hit with the increment, say so
if abs(guess**3 - cube) >= epsilon:
    print("Failed on cube root of", cube)
# Else, show the closest answer given the increment
else:
    print(guess, "is close to the cube root of", cube)
```

Bisection search algorithm

- A bisection search takes a half interval of the search space each iteration.
- Given a problem:
  1. Guess halfway between the interval on each iteration.
  2. Check if the guess is greater than or less than the solution.
  3. Take the correct half interval based on step two.
     - This eliminates half the search space on each iteration.



```
# Find the cube root of 27
sube = 27
epsilon = 0.01
num_guesses = 0
# Set boundries of search space
low = 0
high = cube
# Guess halfway between the interval
gess = (high + low) / 2.0
# Check while the difference between the guess and the cube
# is larger than the epsilon
while abs(guess**3 - cube) >= epsilon:
    # Take the top half if the cube is higher than the guess
    if guess**3 < cube:
        low = guess
    # Take the bottom half if the cube is lower than the guess
    else:
        high = guess
    # Track number of guesses
    num_guesses += 1
# Print number of guesses
print("Number of guesses =", num_guesses)
# Show the closest answer given the epsilon
print(guess, "is close to the cube root of", cube)
```

- For this algorithm, the size of the search space N can be calculated based on the number of searches k.
  - Size of search space on the k^{th} = $N/2^k$
- The guess converges on the order of $log_2 N$ steps.
- Bisection search works when the values of a function varies monotonically with input.
  - Monotonically means the function's output increases or decreases when the input increases or decreases respectively.

▼ Lecture 4: Decomposition, Abstraction, Functions

11/29/23

How do we write code?

- We have covered language mechanisms.
- We know how to write different files for each computation.
- Each file contains a sequence of instructions.

Problems with this approach

- It is messy for larger problems.
- It is hard to keep track of details.
- How does one know the right information is supplied to the right part of the code?

Good programming

- More code is not necessarily a good thing.
- Good programs are measured by their functionality.

Functions

- Functions are mechanisms to achieve decomposition and abstraction.
  - Abstraction means hiding calculations behind a "black box" to reduce the complexity of the program and increase efficiency.

        - This can be achieved with function specifications, also known as docstrings.
        - Functions do not need to know how other functions work.
    - Decomposition means breaking down a complex program into smaller, more manageable parts.
        - This can be achieved with functions or classes.
        - Different functions work together to achieve an end goal.
- Decomposition and abstraction are tools that are very powerful together.
- Modules are self-contained, used to break up code, intended to be reusable, keep code organized, and keep code coherent.
- Functions are called or invoked in a program.

```
# Anatomy of a function:
def <name>(<parameters>):
    """
    Docstring
    """
    <body>
    ...
    # Optional, will return None value if "return" keyword not used
    # None represents the absence of a value in Python
    return <rvalue>

# Invoke a function
<name>(<arguments>)
```

- In a function, the formal parameter (parameter) gets bound to the value of the actual parameter (argument) when the function is called.
- Functions can be nested as arguments.

```
def func_a(y):
    ...
    return z


def func_b(x):
    ...
    return y


# Function A takes the return value of function B as an argument
z = func_a(func_b(n))
```

- Functions can be passed into another function as an argument.

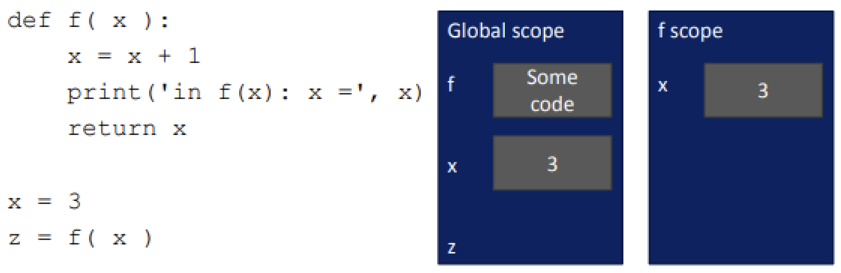
```
def func_a(func, x):
    # func calls function B, as that what was passed into the
    # argument for function A
    return func(x)


def func_b(x):
    ...
    return n


# Pass function B into function A to be used as a function
# within function A's scope
n = func_a(func_b, x)
```

Variable scope

- A new scope is created when the program enters a function.
  - Scopes refer to the environment that the program is in.
  - The program enters a local environment when entering a function, and exits that environment at the end of the function or when a value is returned using the `return` keyword.



```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

- Functions can access, but cannot modify a variable defined outside its scope.
- A function can modify global variables, but this practice is frowned upon.
  - Global variables are variables defined outside of all functions in the program.

`return` vs `print`:

- `return` only has meaning inside a function.
- only one `return` can be executed inside a function.
- Code inside the function after a `return` statement is not executed.
- `return` has a value associated with it, and that value is given to the function caller.

- `print()` can be used outside functions.
- `print()` can execute many print statements inside a function.
- Code inside a function can be executed after a `print()` statement.
- `print()` has a value associated with it, outputted to the console.

Python Tutor

- Python Tutor is a an online compiler a visual debugger for Python that can show one what is happening in their program.

https://pythontutor.com/

▼ Lecture 5: Tuples, Lists, Aliasing, Mutability, Cloning

12/01/23

Tuples

- Tuples are an *immutable* ordered sequence of elements, that can mix element types.
  - One swap tuple elements.

```
# swap values in a tuple
(x, y) = (y, x)
```

  - A tuple with one element can be represented like so:

```
# Assign "x" to a single value tuple
(y,) = x
```

- `//` in Python is the operator for integer division.

```
# The assign 3 to "x"
x = 11 // 3
# Remainder is 2
remainder = 11 % 3
```

Lists

- Lists are a *mutable* ordered sequence of elements.
  - Lists are usually homogeneous, meaning they contain all the same type of elements.
  - However, they can contain mixed types.
  - Lists can be combined in Python with a built it `+` operator overload.

```
# Combine lists A and B
list_a = [1, 2, 3]
list_b = [4, 5, 6]
# List C is [1, 2, 3, 4, 5, 6]
list_c = list_a + list_b
```

- `del()` is used to delete a certain index in a list.
- `.pop()` removes the element from the end of a list and returns the removed element.

- `.remove()` is used to remove the first occurrence of a matching element in the list.

```
# Initial list
L = [2, 1, 3, 6, 3, 7, 0]
# L = [2, 3, 6, 3, 7, 0]
del(L[1])
# Assign 0 to n, L = [2, 1, 3, 6, 3, 7]
n = L.pop()
# L = [2, 1, 6, 3, 7, 0]
L.remove(3)
```

- Lists can be concatenated using `.join()`.

```
# Joined = My name is John Doe
L = ["My", "name", "is", "John", "Doe"]
# The separator is the object that the .join() method is called on
" ".join(L)
```

- `sorted()` does not mutate lists, where `.sort()` does.
    - `sorted()` returns a new list object.
    - `.sort()` returns None and keeps the same list object.

```
L = [1, 2, 3]
# Does not mutate "L", instead it assigns "L" to another variable
L_sorted = sorted(L)
# Mutates "L", and does not require reassignment
L.sort()
```

- `.extend()` is like append, but can add multiple values to a list.

```
# Initial list
L = [1, 2]
# L = [1, 2, [3, 4]]
L.append([3, 4])
# L = [1, 2, 3, 4]
L.extend([3, 4])
```

Mutation, aliasing, cloning

- Lists are mutable, and behave differently than immutable types.
- Lists are objects in memory.
    - The variable name points to the object.
    - Multiple variable names can point to the same object.
- Therefore, when the object is changed by affecting a variable, all the variables in the program change.

- A variable that points to the same object as another variable is called an alias.

```
# Create a list of warm colors
warm = ["red", "orange", "yellow"]
# Assign the list of warm colors to a list of hot colors
# Hot is an ALIAS for warm
hot = warm
# Add pink to the list of hot colors
hot.append("pink")
# Both hot and warm point to the same object
# warm = ["red", "orange", "yellow", "pink"]
# hot = ["red", "orange", "yellow", "pink"]
```

- If a separate list object object is needed, `[:]` can be used to slice the entire original list object.

```
# Initial list
list = [1, 2, 3]
# Create an entirely new object, but assign the same values from
# the original object
copy = list[:]
```

▼ Lecture 6: Recursion and Dictionaries

12/02/23

Recursion

- Recursion is the process of repeating items in a self-similar way.
    - Algorithmically, recursion is a way to design solutions to problems by "divide-and-conquer" or "decrease-and-conquer" methods.
    - Semantically, recursion is a programming technique where a function calls itself.
        - The goal is not to have infinite recursion.
        - There must be one or more base cases that are easy to solve.

```
# Multiply a * b through recurvise reduction
def mult(a, b):
    # If the base case is true, return a
    if b == 1:
        return a
    else:
        # Reduce b by 1
        return a + mult(a, b-1)
```

Iterative algorithms so far

- Looping leads to iterative algorithms.

- These algorithms can capture computation in a set of state variables that update on each iteration through the loop.

12/09/23

Example - Recursive factorial

```
# Find n factorial
def factorial(n)
    # Base case: n == 1
    if n == 1:
        return 1
    # Recursive step: return n * factorial(n - 1)
    else:
        return n * factorial(n - 1)
```

- Each recursive call creates its own scope.
- Bindings of variables in a scope are not changed by recursive calls.
- Flow of control passes back to the previous scope once the function call returns a value.

Inductive reasoning

- Recursive algorithms are based off of mathematical induction.
- If one wants to prove a statement for all integers, one only needs to prove its true for:
    - The smallest value of n (typically 0 or 1).
    - And given its true for n, it also needs to be true for $n + 1$.
- For the previous factorial example:
    - One can show that the base case returns the correct answer.
    - One can show that the recursive case correctly returns an answer for problems of size one smaller than `n`.
        - Then by the multiplication step, it must also return the correct answer for a problem of size n.

Example - Fibonacci

```
# Find the Fibonacci sequence of "n"
def fibonacci(n):
    # Base case:
    if n == 0 or n == 1:
        return 1
    # Recursive case: Add the numbers for the 2 previous steps
    else:
        return fibonacci(x - 1) + fibonacci(x - 2)
```

Recursion with strings

- Check if a `str` is a palindrome:

```
# Find if a string is a palindrome
def palindrome(string):
    # Base case: A string of length 0 or 1 is a palindrome
    if len(string) <= 1:
        return True
    # Recursive case: Does the first character match the last character
    else
        return string[0] == string[-1] and palindrome(s[1:-1])
```

- The key to recursive algorithms is reducing the problem down to a simpler version of itself.

<u>Dictionaries</u>

- Dictionaries are advantageous over lists because they can:
    - Index items of interest directly.
    - Combine multiple values from the same index into key-value pairs.
- When indexing a dictionary, the dictionary:
    1. Look up the key.
    2. Returns the value associated with the key.
    3. If a key isn't found, a `KeyError` is returned.
- One can test is a key is in a dictionary:

```
# Check if "Math" is in the subjects dictionary
if "Math" in subjects:
    return True
else:
    return False
```

- One can delete entries in a dictionary with `del()`:
    - `del()` mutates the dictionary directly.

```
# Delete the "English" key-value pair in the subjects dictionary
del(subjects["English"])
```

- One can iterate over a dictionary's keys and values separately:
    - A dictionary's keys are immutable.

```
# Print all of the subject dictionary's keys:
for key in subjects.keys():
    print(key)
# Print all of the subject dictionary's values:
for value in subjects.values():
    print(value)
```

- If one is not sure if a key is in a dictionary but one wants to access a value anyway, `.get()` can be used to avoid `KeyError`s.
    - The `default` parameter is what is returned if the key does not exists.
    - Even if `default` is not defined, `None` is returned to avoid a `KeyError`.

```
# Usual way to access a dictionary
dict["abc"]
# .get(key, default) implementation
dict.get("abc", 0)
```

Lists vs. dictionaries

`list`	`dict`
• Lists are ordered sequences of elements	• Dictionaries match keys to values.
• Elements are looked-up by an integer index.	• Dictionaries look up one item by another item.
• indices have an order.	• No order is guaranteed for dictionaries.
• indices are `int`s.	• Keys can be any immutable type.

Memoization

- Memoization is a technique for storing the results of functions calls to avoid repetition.
    - Dictionaries can be used to achieve memoization.

```
# Recursive fibonacci sequence, where the result of each function
# call is stored in a dictionary, and each function call is checked
# to see if it has been run before.
def fib_efficient(n, d):
    # If the function call has been run before with this specific value,
    # use the stored result from the dictionary
    if n in d:
        return d[n]
    # If the function has not been run with this value, run the function
    else:
        ans = fib_efficient(n - 1, d) + fib_efficient(n - 2, d)
        # Store the result in the dictionary
        d[n] = ans
        return ans
```

▼ Lecture 7: Testing, Debugging, Exceptions, and Assertions

12/10/23

Defensive programming

- Defensive programming means writing programs in a way to catch bugs and exceptions before they break a program.

    - One should write specifications for functions.
    - Programs should be modularized.
    - Input/output conditions should be checked.

Testing/validation

- Compare input/output pairs to the specifications.
- Think about what could break the program.

Debugging

- Study events leading up to an error.
- Think about how you can fix your program.

Testing and debugging

- From the start of writing a program, design your code to ease testing and debugging.
- Break your program into modules that can be tested and debugged individually.
    - Document the constraints on modules.
- Document your assumptions behind your code design.

Classes of tests

- Unit testing:
    - Validates each piece of a program by testing each function separately.
- Regression testing:
    - Adds tests for bugs as you find them, and catches reintroduced errors that were previously fixed.
- Integration testing:
    - Tests if the overall program works.

Testing approaches

- Black box testing:
    - Explore paths through specification based on the docstring.
    - The test is designed based on the specification, not the code itself.
    - The test can be reused even if the implementation changes.
- Glass box testing:
    - Come up with test cases that go through all the paths in the code.
    - The code directly guides the test cases.
    - The test is path-complete if every potential path through the code is tested at least once.
    - Drawback of glass box testing are:
        - The test can go through loops arbitrarily many times.
            - Loops should be entered 0, 1, and more than one times is separate tests.
        - Paths can be easily missed.

Debugging

- Debugging works towards the goal of having a bug-free program.
- Tools to debug include:
  - Python Tutor
  - Breakpoints/debuggers
    - Step through code one line at a time.
  - `print()`
    - `print()` statements are a good way to test hypothesis.
    - The bisection method can be used intuitively to find a bug in a program:
      1. Put a print statement halfway through the code with the value you want to debug.
      2. If the value is correct, put the print statement halfway between the current position and the end. If the value is incorrect, do that in the opposite direction towards the state of the program.
      3. Repeat until the line(s) where the bug is are located.
- Use the scientific method:
  - Study the available data.
    - Read error messages!
    - Look at the program code.
  - Form a hypothesis.
  - Repeat experiments.
    - Pick the simplest inputs to test with.

Error messages

- `IndexError`s happens when one try's to access beyond the limits of a list.

```
# Throw an IndexError
test = [1, 2, 3]
test[4]
```

- `TypeError`s happen when trying to convert a value to an inappropriate type.

```
# Throw a TypeError
test = "Hello World!"
int(test)
```

- `NameError`s happen when trying to reference an non-existent variable.

```
# Throw a NameError
a = 1
print(b)
```

- `TypeError`s happen when trying to mix inappropriate datatypes.

```
# Throw a TypeError
x = "3" / 4
```

- `SyntaxError`s happen when writing improper syntax.

```
# Throw a SyntaxError
print("Hello World!"
```

- `AttributeError`s happen when an attribute reference fails.

```
# Throw an AttributeError
class MyClass:
    pass

obj = MyClass()
# There is no attribute named "foo"
obj.foo()
```

- `IOError`s happen when an IO system report s malfunction or a file is not found.

```
# Throw an IOError
with open("non_existant_file_name.txt", "r") as file:
    ...
```

Logic errors

- Think before writing new code.
- Draw pictures.
- Explain the code out loud to either yourself, someone else, or a rubber ducky.
- DO NOT write an entire program at once and test it all at once.

Exceptions

- Python provides handlers for exceptions in `try` and `except` statements.
  - Exceptions raised by any statement in `try` are handled by the `except` statement.
  - Multiple `except` clauses with different handlers can be used to deal with multiple types of exceptions.

```
# Catch a ValueError if "x" is not an integer
try:
    x = int(input("What's x? "))
except ValueError:
    print("x is not an integer!")
```

- Unlike `else`, `finally` can be used to always execute some code no matter:
  - If `except` or `else` is called.

    - Or `break`, `continue`, or `return` is called.
- This is good for clean-up code that needs to be run no matter what, such as closing a file.

```
# Catch a ValueError if "x" is not an integer
try:
    x = int(input("What's x? "))
except ValueError:
    print("x is not an integer!")
    return
# Print "x" anyway
finally:
    print(f"x is {x}")
```

What to do with exceptions

- Failing silently means substituting values for a default value or just continuing.
    - This is a BAD IDEA.
- Return an error value.
    - This is also a bad idea, as is complicates code by having to check for a special value.
- Stop execution with a signal error condition (raising your own exception) with the `raise` keyword.

```
# Raise a ValueError
try:
    x = int(input("What's x? "))
except ValueError:
    raise ValueError("x is not an integer!")
```

Assertions

- For unit testing, one wants to be sure that assumptions about certain values are as expected.
- The `assert` statement is used to raise an `AssertionError`.
    - An `AssertionError` is raised if the assumption is not met.
    - Assertions are a good tool for defensive programming.
        - They check that pre- and post- conditions on functions are as expected.

```
# Assert that the square of 2 is 4
assert square(2) == 4
```

- Assertions:
    - Don’t allow a programmer to control the response to unexpected conditions.
    - Ensure that execution of a program halts whenever an expected condition is not met.
    - Typically are used to check inputs to functions, but can be used anywhere.

  - Can be used to check outputs of a function to avoid propagating bad values.
  - Can make it easier to locate a source of a bug.
  - Should be used as a supplement to testing.
  - Can raise exceptions if users supply bad data as an input.
  - Can check:
    - Datatypes of values.
    - Constraints on return values.
    - For violations of constraints.

▼ Lecture 8: Object Oriented Programming

12/10/23

Some of this content already is covered in CS50P's Introduction to Programming with Python Lecture 8.

Objects

- Every variable in Python is an object.
- Each object has a:
  - type,
  - internal data representation,
  - and a set of procedures for interaction with the object.
- An object is an instance of a type.

Object oriented programming (OOP)

- One can create new objects.
- One can manipulate objects.
- One can destroy objects.
  - `del` can be used.
  - Or Python will reclaim the objects with its garbage collector.

What are objects?

- Objects are data abstractions that have:
  1. An internal representation (attributes).
  2. An interface for interacting with the object (methods).

Advantages of OOP

- Data can be bundled into packages, that work on them through well-defined interfaces.
- Objects allow for "divide-and-conquer" development.
  - Behavior can be implemented and tested on each class separately.

    - The program becomes less complex and increasingly modularized.
- Classes make it easy to reuse code.
    - Inheritance allows subclasses to redefine or extend a superclass' behavior.

Attributes and methods

- Attributes are data and procedures that "belong" to the class.
- Data attributes are data that make up the class.
- Methods (procedural attributes) are functions that "belong" to the class.
    - Methods allow one to interact with an object.
    - Python always passes the object itself (`self`) as the first argument in any method.
    - The `.` operator is used to access any attribute.
    - The method `__init__` is used to initialize data attributes when an object is created from a class.
- One can call a method two ways:

```
# Conventional way
obj.method(args...)
# Alternative way
ClassName.method(obj, args...)
```

Printing the representation of an object

- By default, printing an object returns the pointer to the object's location in memory.
    - The information is generally unhelpful.
- The `__str__` method can be used to define what happens when `print()` is used on the class.

Operator overloads

- Much like `__str__` overrides what happens when `print()` operates on an object, other special operators methods can be used to change the function of other operators on objects.

```
# self + other
__add__(self, other)
# self - other
__sub__(self, other)
# self * other
__mul__(self, other)
# self == other
__eq__(self, other)
# self < other
__lt__(self, other)
# self <= other
__le__(self, other)
# self > other
__gt__(self, other)
```

```
# self >= other
__ge__(self, other)
# self != other
__ne__(self, other)
# len(self)
__len__(self)
```

The power of OOP

- Object that share common attributes and procedures to operate on those attributes can be bundled together.
- Abstraction can be used to make a distinction between how to implement an object versus how to use an object.
- One can build layers of object abstractions that inherit behaviors from other classes of objects.

▼ Lecture 9: Classes and Inheritance

12/12/23

Implementation vs usage

Implementing a class:

- Implementing a new object type with a class means defining data attributes and methods (procedural attributes).
- The class name is the type.
- The class defines the data common across all instances.

Using an object:

- Using a new object creates instances of the object type, and allows one to do operations with them.
- An instance is one specific object.
- An instance has the structure of a class, but data values vary between instances.

Getter and setter methods

- Getters and setters are used to “get” and “set” the values of data attributes.
  - This creates an interface to interact with the class without accessing its data attributes directly.
  - This defensive programming can prevent bugs.

```
# Class getters and setters
class Foo:
    ...
    # Getter
    def get_bar(self):
        return self.bar
    # Setter
    def set_bar(self, bar):
        self.bar = bar
```

Dot notation

- Instantiation creates an instance of an object.
- Dot notation is used to access data and procedural attributes of an instance of a class.

```
# Create an instance of the Foo class
foo = Foo()
# Access the bar() procedural attributes
foo.bar()
```

Information hiding

- An author of a class definition may change data attribute variable names.
- Errors may be raised if one is accessing data attributes outside the class and class definition changes.
- Getters and setter prevent this by not directly interacting with the implementation. They allow for:
    - good style,
    - easy to maintain code,
    - and preventing bugs.

Python is not great at information hiding

- Python allows you to:
    - access data from outside the class definition,
    - write to data from outside the class definition,
    - and create data attributes for an instance from outside the class definition.
- It is not good style to do any of these actions.

Default arguments

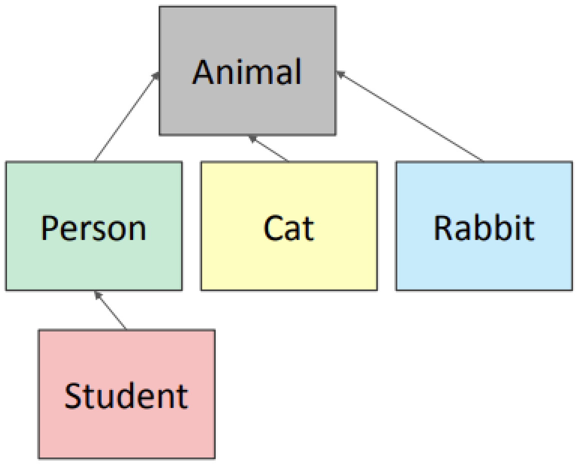
- Default arguments for formal parameters are used if no actual argument is given.

```
# If arg2 is not passed, use "" as arg2
def concat(arg1, arg2="")
    return arg1 + arg2

# Return "Hello"
string = concat("Hello")
```

Hierarchies

- Hierarchies allow classes to inherit data and behaviors.
- The parent (superclass) passes data to its child (subclass).
    - The inheritance can:
        - add more data,
        - add more behavior,
        - or override behavior.



Which method to use?

- Subclasses can have methods with the same name as super-classes.
- For an instance of a class, look for a method name in the current class definition.
- If the method name is not found, look for the method name up the hierarchy.
- The first method up the hierarchy that is found is the method that will be used.

Class variables

- Unlike instance variables, class variables and their values are shared between all instances of a class.
    - Class variables are defined inside the class definition, but outside the `__init__` method.

```
# This class has the class variable "bar"
class Foo:
    # Define the class variable "bar"
    bar = 1
    def __init__(self, args...):
        ...

    ...
    def baz(self, arg):
        # Access and modify the class variable "bar"
        Foo.bar += arg


# Access "bar" outside the class definition
foo_bar = Foo.bar
```

Recap

- OOP allows:
    - one to create their own collections of data,

    - intuitive organization of data,
    - division of work,
    - access to information in a consistent manner,
    - layers of complexity,
    - a mechanism for decomposition and abstraction in programming.

▼ Lecture 10: Understanding Program Efficiency, Part 1

12/13/23

How to understand the efficiency of programs?

- There is a need to separate time and space efficiency of a program.
    - One might pre-compute results and store them then use a "lookup" to increase time efficiency but decrease space efficiency.
- Challenges relating to understanding the efficiency of an implementation of a computational problem are:
    - A program can be implemented many different ways.
    - One can solve a problem using only a handful of different algorithms.
    - We would like to separate choices of implementation from choices of a more abstract algorithm.
        - How efficient is the algorithm, not the implementation?

How to evaluate the efficiency of programs?

- One can measure the time a program takes with a timer.
- One can count the operations.
- One can use the abstract notion of order of growth.
    - This is also called "Big O Notation".
    - This method may be the most appropriate way of assessing the impact of choices of an algorithm in solving a problem.
    - It measures the inherent difficulty of solving a problem.

Timing a problem

- In Python one can use the `time` module.

```
# Import the time module
import time

# Function being tracked
def c_to_f(c):
    return (c * 9 / 5) + 32

# Start the clock
t0 = time.clock()
```

```
# Run the function
c_to_f(100)
# Stop the clock (measure the time difference, really)
t1 = time.clock() - t0
```

- Unfortunately, timing a program can be inconsistent.
  - Running time varies between algorithms, implementations, and computers.
  - Running time is not predictable based on small inputs.
  - This method can not really express a relationship between input and time.

Counting operations in a problem

- Assume all steps take a constant time, then count the number of operations executed as a function of size input.

```
# Function being tracked (3x + 1 operations)
def mysum(x):
    total = 0  # 1 operation
    for i in range(x + 1): # x operations
        total += i # 2x operations
    return total
```

- This is an improvement, as the counting depends on the algorithm, not the computer.
- Count varies for different inputs, and therefore a relationship between inputs and count can be reached.
- Unfortunately, the count is still dependent on the implementation.
  - There is also no clear definition on which operations to count.

Still need a better way

- We want to evaluate the algorithm based on scalability, and in terms of input size.
- We should focus on the idea of counting operations in an algorithm, rather than the number of operations in an implementation.
- We will focus on how algorithms perform when the size of it gets arbitrarily large.
  - We need to relate the time needed to complete a computation against the size of the input of the problem.

Orders of growth

- Efficiency will be expressed in terms of the size of the input(s).
  - The input(s) could be an integer, the length of a list, etc.
- Different inputs can change how a program runs.

```
# When "e" is the first element, that's the BEST CASE
# When "e" not in the list, that's the WORST CASE
# When "e" is through about half the list, that's the AVERAGE CASE
def search_for_element(L, element)
    for i in L:
        if i = element:
            return True
    return False
```

- Our goal is to measure the behavior of algorithms in a general way.
    - The best case is the minimum running time over all possible inputs of a given size.
    - The worst case is the maximum running time over all possible inputs of a given size.
    - The average case is the average running time over all possible inputs of a given size.
- Generally, the worst case will be the case that is focused on.
- The goals of orders of growth are to:
    - Evaluate a programs efficiency when an input is very big.
    - Express the growth of a program's run time as input size grows.
    - Put an upper bound on growth.
    - Not need to be precise
    - Look at the largest factors in run time.
- Generally, we want a tight upper bound on growth, as a function of size input, in the worst case.

Measuring order of growth: "big O notation"

- Big O notation measures an upper bound on the asymptotic growth, often called the order of growth.
- Big O is used to describe the worst case for an algorithm.
    - It evaluates the algorithm, not the machine or implementation.
    - We focus on the term that grows most rapidly, ignoring additive and multiplicative constants.

```
# The worst case asymptotic complexity for this algorithm is O(n)
# This means that this algorithm scales linearly with the input "n"
def fact_iter(n):
    while n > 1:
        answer *= n
        n -= 1
    return answer
# Even though the counting complexity is 5n + 2, the additive
# and multiplicative constants are ignored for arbitrarily large numbers
```

What does O(n) measure?

- O(n) describes the amount of time needed based on the size of the input.
- Thus, given an expression for the number of operations needed to compute an algorithm, we want to know the asymptotic behavior as the size of the problem gets large.
- Hence, we will focus on the term that grows the most rapidly in a sum of terms.
- And we will ignore multiplicative constants, since we want to know how rapidly the time required increases as the size of the input increases.

```
# O() focusses on the dominant terms as :n" gets arbitrarily large
# O(n^2)
n^2 + 2n + 2
# O(n^2)
n^2 + 100000n + 3^10000
# O(n)
log(n) + n + 4
# O(n * log(n))
(0.0001 * n * log(n)) + 300n
# O(3^n)
2n^30 + 3^n
```

Analyzing programs and their complexity

- Big O can be combined from different pieces of a program.
  - One should analyze statements inside functions.
  - The dominant term should be focused on.
- Law of addition of Big O:
  - This is used with sequential statements.
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  - For example, if two loops in sequence are $O(n) + O(n * n)$

  ```
  # O(n)
  for i in range(n):
      ...
  # O(n^2)
  for j in range(n*n)
      ...
  ```

  - This simplifies to $O(n + n^2) = O(n^2)$ because only the dominant term is tracked.
- Law of multiplications for Big O:
  - This is used with nested statements/loop.
  - $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
  - For example, if two nested loops are $O(n) * O(n)$

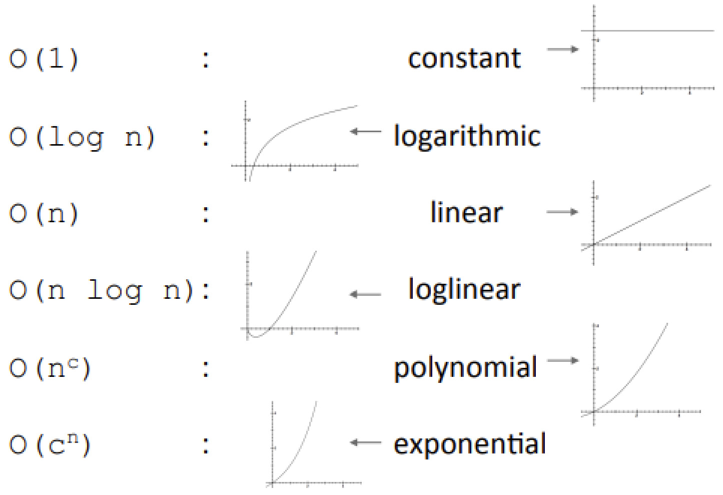
  ```
  # O(n)
  for i in range(n):
      # O(n)
      for j in range(n):
          ...
  ```

  - This simplifies to $O(n * n) = O(n^2)$,

        - n^2 being kept as it is the dominant term.

Complexity classes and types of growth

- The orders of complexity from lowest to highest are:
    - $O(1)$ denotes constant running time.
        - Algorithms that reference a list of allocated data are typically constant in complexity.
    - $O(log(n))$ denotes logarithmic running time.
    - $O(n)$ denotes linear running time.
        - Simple iterative loop algorithms are typically linear in complexity.
            - For example, searching through a list to see if an element is present.
    - $O(n * log(n))$ denotes log-linear running time.
    - $O(n^2)$ denotes quadratic running time, c a constant.
        - One loop nested in another loop is typically quadratic in complexity.
    - $O(n^c)$ denotes polynomial running time, c a constant and $c > 2$.
    - $O(c^n)$ denotes exponential running time, c a constant.



▼ Lecture 11: Understanding Program Efficiency, Part 2

12/13/23

Why we want to understand efficiency of programs

- How can we reason about an algorithm in order to predict the amount of time it will need to solve a problem of a particular size?
- How can we relate choices in algorithm design to the time efficiency of the resulting algorithm?

Constant complexity $O(1)$

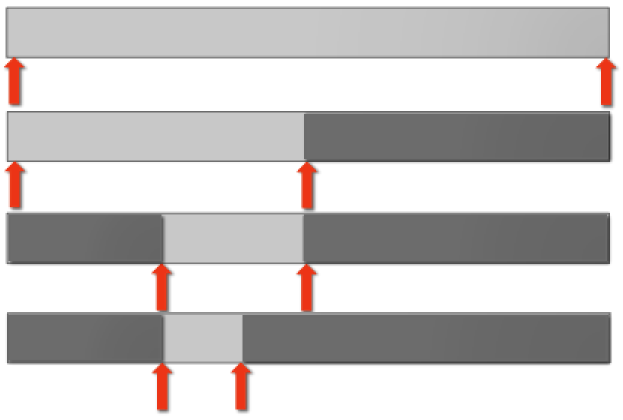
- The complexity is independent of input.
- Constant complexity algorithms can have loops or recursive calls, but only if the number of iterations or calls in independent of the size of the input.

Logarithmic complexity $O(log(n))$

- The complexity grows logarithmically based on the size of the input.
- The problem is reduced in half each time through the process.
- Example:

  Bisection search

  - Suppose we want to know if a particular element is present in a list `L`.
  - We can iterate through it, but that would result in a complexity of $O(n)$.
  - If the list is sorted from smallest to largest, we can use bisection search.
  - In the worst case, the for the resulting number of iterations i, $1 = n/2^i$.
  - Simplify to $i = log(n)$.
  - So the complexity is $O(log(n))$, when n is `len(L)`.



Linear complexity $O(n)$

- The complexity grows linearly based on the size of the input.
- Examples:
  - Searching a list in sequence
  - Iterative loops
  - Recursive programs

Log-linear complexity $O(n * log(n))$

- The complexity grows log-linearly based on the size of the input.
- Example:

    - Merge sort

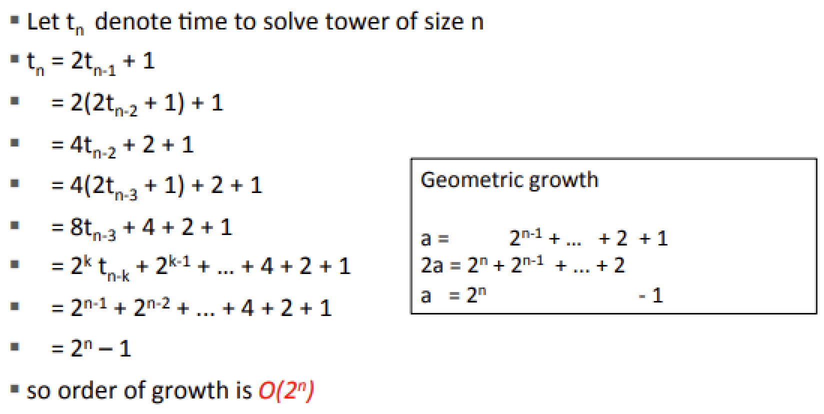
<u>Polynomial complexity $O(n^c)$</u>

- The complexity grows to the power of c, a constant, based on the size of the input.
- Examples:
    - Nested loops
    - Nested recursive function calls

<u>Exponential complexity $O(c^n)$</u>

- The complexity grows to the power of n, ,the size of the input.
- Typically, there are multiple loops or recursive calls at each level.
- Example:

  <u>Towers of Hanoi:</u>



- Let t_n denote time to solve tower of size n
- $t_n = 2t_{n-1} + 1$
- $= 2(2t_{n-2} + 1) + 1$
- $= 4t_{n-2} + 2 + 1$
- $= 4(2t_{n-3} + 1) + 2 + 1$
- $= 8t_{n-3} + 4 + 2 + 1$
- $= 2^k t_{n-k} + 2^{k-1} + \ldots + 4 + 2 + 1$
- $= 2^{n-1} + 2^{n-2} + \ldots + 4 + 2 + 1$
- $= 2^n - 1$
- so order of growth is *$O(2^n)$*

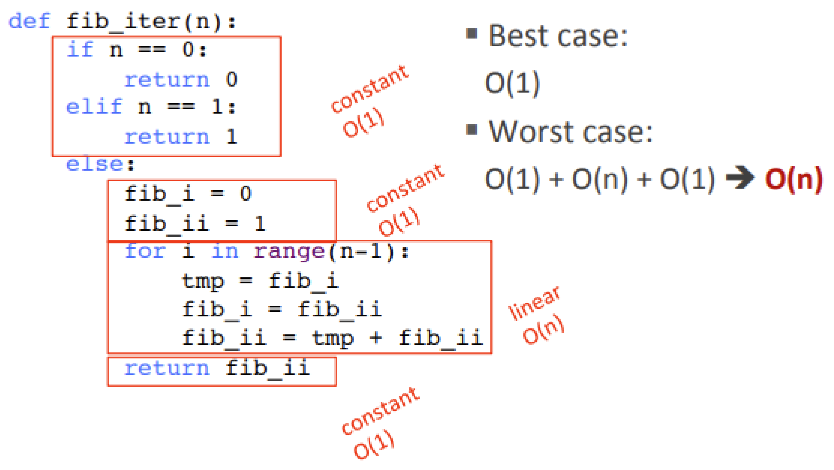
Geometric growth

$a = 2^{n-1} + \ldots + 2 + 1$
$2a = 2^n + 2^{n-1} + \ldots + 2$
$a = 2^n - 1$

- Many important problems are inherently exponential.
    - This is unfortunate, as the cost can be high.
    - This leads us to consider approximate solutions, as they may provide reasonable answers more quickly.

<u>How implementation can affect complexity</u>

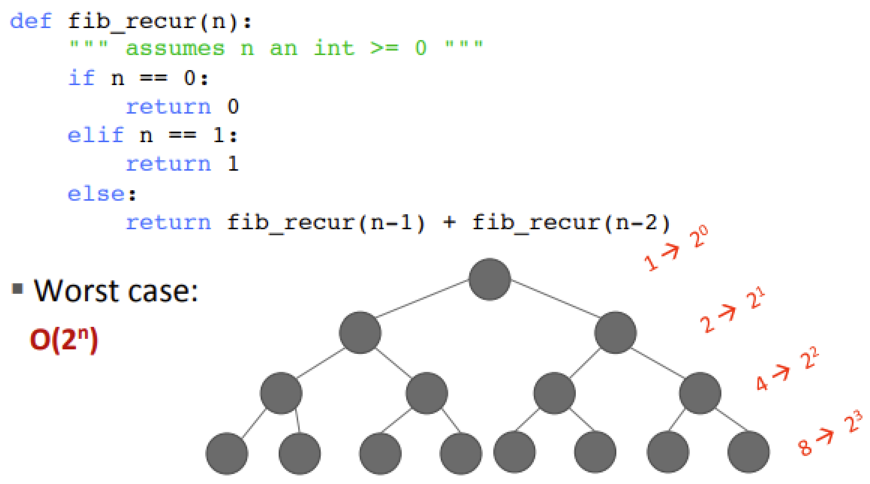
- In this example, the complexity of an iterative Fibonacci algorithm is $O(n)$, where the complexity of a recursive Fibonacci algorithm is $O(n^2)$.
- Iterative implementation:



```python
def fib_iter(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fib_i = 0
        fib_ii = 1
        for i in range(n-1):
            tmp = fib_i
            fib_i = fib_ii
            fib_ii = tmp + fib_ii
        return fib_ii
```

- Best case:
  O(1)
- Worst case:
  O(1) + O(n) + O(1) ➔ **O(n)**

- Recursive implementation:



```python
def fib_recur(n):
    """ assumes n an int >= 0 """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_recur(n-1) + fib_recur(n-2)
```

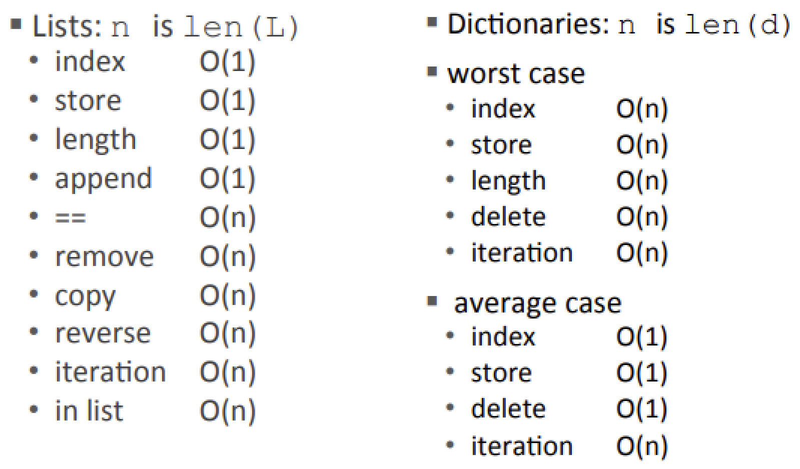
- Worst case:
  O(2ⁿ)

Big O summary

- Big O compares the efficiency of algorithms.
  - It is a notation that describes growth.
  - A lower order of growth is better.
  - It describes the algorithm’s complexity independent of machine or specific implementation.
- Big O should be used:
  - To describe the order of growth of an algorithm.
  - For asymptotic notation.
  - To find an upper bound.
  - For worst case analysis.

Complexity of common Python functions

- This chart shows the algorithmic complexity of many common Python functions that operate on `list`s and `dict`s.



Lists: `n` is `len(L)`

index	O(1)
store	O(1)
length	O(1)
append	O(1)
==	O(n)
remove	O(n)
copy	O(n)
reverse	O(n)
iteration	O(n)
in list	O(n)

Dictionaries: `n` is `len(d)`

worst case

index	O(n)
store	O(n)
length	O(n)
delete	O(n)
iteration	O(n)

average case

index	O(1)
store	O(1)
delete	O(1)
iteration	O(n)

Lecture 12: Searching and Sorting

12/17/23

Search algorithms

- A search algorithm is a method for finding an item or group of items with specific properties within a collection of items.
- A collection could be implicit, as in it is between certain values.
  - For example, an algorithm could search for the square root of a number.
- A collection could be explicit, as in not necessarily in the search space.
  - For example, an algorithm that searches for a student record in a collection of data.

Searching algorithms

- Linear search:
  - $O(n)$ complexity
  - Brute force
  - List does not have to be sorted
- Bisection search:
  - $O(log(n))$ complexity
  - List must be sorted
- It makes sense to sort a list then search when:

  $SORT + O(log(n)) < O(n)$
  - When sorting is less that $O(n)$.
  - This is NEVER true.

Amortized cost

- Why bother sorting first then?
- Because is many cases, it makes sense to sort a list once then do many searches on it.
- You are essentially amortizing (spreading out) the cost of the sort over many searches.
  $SORT + K * O(log(n)) < K * O(n)$, K being the amount of searches.
    - For a large K, the sorting time becomes irrelevant.

Bogo sort

- Bogo sort, also known as monkey sort, stupid sort, slow sort, permutation sort, shotgun sort, etc., it the process of randomly selecting permutations of a sequence until the correct sequence is found.
- The complexity is $O(n!)$.

Bubble sort

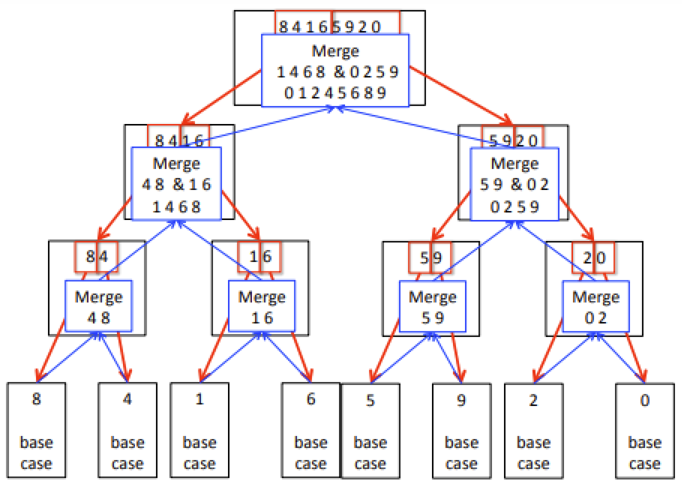
- Bubble sort compares consecutive pairs of elements.
    1. Element pairs are swapped so that the smaller is first.
    2. When the end of the list is reached, the sort is started over again.
    3. The sort is stopped when there are no more swaps.
- There are always at most n passes.
    - This is because the largest unsorted element is always at the end after a pass.
- The complexity is $O(n^2)$.

Selection sort

- Steps:
    1. Selection sort extracts the minimum element in the list and swaps it with the element at index 0.
    2. This is repeated with each remaining sub-list of everything but that element.
- The loop “prefix” `L[0:i]` will be sorted and the “suffix” `L[i+1:len(L)]` elements will be larger than any element in the prefix.
- The complexity is $O(n^2)$.

Merge sort

- Merge sort uses a divide-and-conquer approach.
    1. If the list is of length 0 or 1, it is already sorted.
    2. Split the list into two lists, and sort each.
    3. Merge the sorted sub-lists.
        a. Look at the first element of each list, and move the smaller to the end of the result.
        b. When one list is empty, copy the rest of the other list.
- The complexity is $O(n * log(n))$.
    - This is the fastest a sort can be.



6.0001 Introduction to Computer Science recap

- Topics:
  - Knowledge can be represented using data structures.
  - Iteration and recursion can be used to implement algorithms.
  - Abstraction can be used to simplify procedures and datatypes.
  - Object classes and methods can be used to organize and modularize programs.
  - There a different classes of algorithms, some of which can be used for searching and sorting.
  - Algorithms can be measured by complexity.
- Themes:
  - We learned about computational modes of thinking.
  - We began to master the art of computational problem solving.
  - Computers can be made to do what you want them to do.

What do computer scientists do?

- Computer scientists think computationally using:
  - Abstractions:
    - Choosing the right abstractions.
    - Operating in multiple layers of abstraction.
    - Defining the relationships between the abstraction layers.
  - Algorithms:
    - Thinking in terms of mechanizing our abstractions.
      - This is possible due to precise and exacting notations and models.
      - There is a machine, a computer, than can interpret these notions and models.

    - Automated execution:
        - Languages for describing automated processes.
        - Languages that allow abstraction of details.
        - Languages for communicating ideas and processes.
- Computational thinking is becoming a fundamental skill that every well-educated person will need in the future.
    - One should think about how difficult a problem is and how it can best be solved.
        - Theoretical computer science gives precise meaning to these and related questions and their answers.
    - One should think recursively.
        - Reformulating seemingly difficult problems into one which we know how to solve.
            - This involves reduction, embedding, transformation, and simulation.