CS 304 Notes

Data Structures & Algorithms

Jaeden Bardati

Last modified January 28, 2022

1 Python and Object-Oriented Programming

Python is a *high-level programming language* which makes it useful when studying data structures and algorithms.

Python is an *interpreted* language, where *source code* (also referred to as *scripts*), in the form of files with the *.py* suffix, are run by an *interpreter*. It is common to use an integrated development environment (IDE) to aid in displaying and editing Python code. IDEs for Python include the built-in IDLE, PyCharm, Spyder, and others.

1.1 Objects in Python

Python is an *object oriented* language where **classes** are the basis of all data types. Examples of data types in Python include int, float, str.

An **assignment statement** assigns an **identifier** (or name) to an object. Identifiers are associated with a *memory address* and are similar to a pointer in languages such as C++ or Java. For example, the statement

```
1  temp = 98.6
```

associates the identifier "temp" to the float value 98.6.

Identifiers are **case-sensitive**. Namely, a variable named "temp" is different than one named "Temp".

Python is a **dynamically typed** language, unlike C++ or Java. That is to say, when making the association of an identifier in Python, the data type is not explicitly declared. In the code above, the data type is determined automatically by the interpreter to be a float.

It is possible to establish an **alias** by assigning a second identifier to an object as follows

```
temperature = temp
```

After an alias is made, either name can be used to refer to the object.

The process of creating a class is called **instantiation**. To do this, we call the **constructor** of a class. If we have defined the class called Animal then we would do this by

```
a = Animal()
```

Note that we can also pass parameters to the Animal constructor.

See Chapters 1 and 2 in the textbook for more.

2 Design and Analysis of Algorithms

2.1 Data structures vs. Abstract Data Types

There are many **abstract data types**: list, set, queue, stacks, dictionary, etc.

Data structures are the implementation of abstract data types. Examples include: arrays, trees, hash tables, etc.

Our goal is to find the best data structure to use. The best data structure is one that minimizes the **time complexity**.

For example, if we have a list $L = [1, 2, 3]$ and we want to insert an element. If we use an array abstract type, then it is easy to write to insert an element at the end of the list, but it is more complicated to insert it at the beginning of the list (must shift all other elements over 1).

Another example is the binary search algorithm (e.g. looking for a number in a phone book) has the time complexity of $\theta(log_2 n)$.

2.2 Insertion Sort

The insertion sort algorithm written in pseudocode in Algorithm 1.

We can check the code using a trace for A = [3, 2, 1]. The variables as they change are:

1. $j = 2$
2. key = 2
3. $i = 1$

4. $A = [3, 3, 1]$
5. $i = 0$
6. $A = [2, 3, 1]$
7. $j = 3$
8. key $= 1$
9. $i = 2$
10. $A = [2, 3, 3]$
11. $i = 1$
12. $A = [2, 2, 3]$
13. $i = 0$
14. $A = [1, 2, 3]$
15. $j = 4$

2.2.1 Algorithm efficiency

WE can go line by line through the program and associate an arbitrary time cost as well as a number of times that the line will run. We can represent this in Table 1.

The time that the algorithm takes is

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best case: $t_j = 0$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) = (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8)$$

Algorithm 1 InsertionSort(A)

```
1: for (j = 2 to length(A)) do
2:     set key = A[j]
3:     // insert A[j] into the
4:     // sorted sequence A[1..j − 1]
5:     set i = j − 1
6:     while (i > 0 and A[i] >key) do
7:         set A[i + 1] = A[i]
8:         set i = i − 1
9:         set A[i + 1] = key
10: stop
```

Table 1: Insertion Sort Algorithm Cost

Line	Cost	Times
1	c_1	n
2	c_2	$n-1$
3	0	0
4	0	0
5	c_3	$n-1$
6	c_4	$\sum_{j=2}^{n} t_j$
7	c_5	$\sum_{j=2}^{n}(t_j-1)$
8	c_6	$\sum_{j=2}^{n}(t_j-1)$
9	c_7	$n-1$

where we let t_j be the number of times that the while loop is executed for a given j.

This can be written in the form $an+b$. Namely, the best case is a linear function with n.

Worse case: $t_j = j$ for $j = 2, 3, ...n$, therefore,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

And,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Thus,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

This can be written in the form $an^2 + bn + c$. Namely, the worst case is a quadratic function with n.

2.3 Asymptotic Analysis

We say insertion sort has a worst-case running time of $\Theta(n^2)$. All we care about is the order of n as n asymptotically increases without bound. For example,

$\frac{n^3}{1000} - 100n^2 - 100n + 3$ has a time complexity of $\Theta(n^2)$.

Definition: For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) \equiv \{f(n) : \exists \text{ positive constants } c_1, c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \, \forall \, n \geq n_0\}$$

For all $n \geq n_0$ f(n) is equal to g(n) within a constant factor. Note that we often abuse the notation and write that $f(n) = \Theta(g(n))$ rather than $f(n) \in \Theta(g(n))$ as one might expect.

For example, let us justify that $\frac{n^2}{2} - 3n = \Theta(n^2)$. We must determine positive constants c_1, c_2, n_0 s.t. $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2 \; \forall \; n \geq n_0$.

Dividing by n^2, $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$, where we choose $c_2 = \frac{1}{2}$, $c_1 = \frac{1}{14}$ for $n_0 \geq 7$.

Definition: For a given function $g(n)$, we denote $O(g(n))$ as the set of functions

$$O(g(b)) \equiv \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \, \forall \, n \geq n_0\}$$

We call this "big O" notation. See Figure 3.5 in the textbook for a visualization. It is important to note that if $f(n) = \Theta(g(n))$, then necessarily $f(n) = O(g(n))$. Also note that the $\Omega(g(n))$ notation indicates a minimum asymptotic trend.

3 Array-Based Sequences

3.1 Sequence Types in Python

The main sequence types in Python are the **list**, **tuple** and **string**. These sequence types can be **referential** or **compact**, and **mutable** or **immutable**. Referential are distinguished from compact arrays later on in these notes. Mutable arrays are array that can be changes in some way (appended to, values changed, values inserted, etc.), whereas immutable arrays cannot be changed. The following are the properties of the sequence types.

- lists are *referential* and *mutable*
- tuples are *referential* and *immutable*
- strings are *compact* and *immutable*

We can do a variety of operations on these sequence types. Some of them include "len(data)", which finds the length of the array data, or "data1 == data2", which checks if the arrays data1 and data2 are equal. See Tables 5.3 and 5.4

from the textbook for a comprehensive list.

The non-mutating behaviors are

- len(data): Finds length of array
- data[j]: Returns the jth index
- data.cout(value): Number of times value is in data
- data.index(value): The index of the value in data
- Value in data: Same as above
- data1 == data2: Checks if the arrays are equal
- d2 = data[j:k]: Creates a new array which is a slice from index j (inclusively) to k (exclusively)
- d2 = data1 + data2: Creates a new array which is the concatenation of the arrays
- d2 = c*data: Creates a new array which is c copies of data

and the mutating behaviors are

- data[j] = val: Change the value at index j to val
- data.append(val): Append to the end of the array
- data.insert(k, val): Insert val in an element at index k (increases size)
- data.pop(): Removes (and returns) the last element of the array
- data.pop(k): Removes (and returns) the kth element of the array
- del data[k]: Removes the kth element of the array
- data.remove(val): Searches the array for val and removes it
- data1.extend(data2): Appends the array data2 to the array data1
- data.reverse(): Reverses the order of the array
- data.sort(): Sorts the array in increasing order

An implementation of data.insert in our previous array class in Python is

```
1  def insert (self, k, value):
2    if self._n == self._capacity:
3      self._resize(2*self._capacity)
4    for j in range(self._n, k, -1):
5      self._A[j] = self._A[j-1]
6    self._A[k] = val
7    self._n += 1
```

3.2 Low-Level Arrays

3.2.1 What is an Array?

First, we must define RAM. Random-access memory (RAM) is memory that can be randomly-accessed. Randomly-accessed means that any random element can be accessed just as easily as any other. We can visualize this in a table as follows

S	A	M	P	L	E

An array is a group of related values sorted one after another in a contiguous portion of RAM. To access an array, we refer to the memory address that the array begins at and add the index. Namely,

```
A[i] = start + index
```

3.2.2 Note for Python Arrays

Python arrays use pointers to refer to an array in memory. Therefore, we need to explicitly copy an array to make sure it is copied. In the following example, an alias is made and so the sorted array [1, 2, 3] would be outputted.

```
x = [3, 2, 1]
y = x
y.sort()
print(x)
```

3.2.3 Referential Arrays

Referential Arrays are arrays that store memory addresses (object references). For example, an array of strings is a referential array because strings are stored as pointers to another location in memory (to a compact array).

Let us say that we have an array of primes:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

Then, we can create a sub-array with

```
temp = primes[3:6]
```

The array temp is a referential array to a part of the array primes (or rather, to the objects to which the elements reference). See Figure 5.5 in the textbook. If then we set

```
temp[2] = 15
```

Then the temp array at index 15 is now pointing to another location in memory with a new entry 15, but the rest of the array will still point to their original addresses. See Figure 5.6 in the textbook.

If we have an array,

```
counters = [0]*8
```

This will create an array of *pointers* to the same 0 object. See Figure 5.7 in the textbook. Adding 1 to one of the elements will create a new 1 object and make that element point to it (see Figure 5.8).

3.2.4 Compact Arrays

Compact arrays are when the elements are stored directly in the array (not an array of pointers). An example of this is a string, with is an array of characters (not pointers of characters). Compact arrays have advantages over referential arrays. Namely, they are good because they

1 Have less memory usage: No overhead devoted to storage of memory references
2 Have higher performance:
2a) direct access
2b) principle of locality (if we access a certain location in memory, we are likely to access the array again nearby)

To use a compact array, we can use the array module. To create one, we must pass a list of a certain type, along with the type code for that type. For a list of signed integers, we use the type code 'i'. For example, we could write

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

3.3 Dynamic Arrays and Amortization

Dynamic Arrays are arrays that can dynamically increase in time. To do this, dynamic arrays reserve a certain amount of space in memory for future appends until it runs out of reserved space, in which case it rewrites the new array along with more reserved space. Python's list class is a dynamic array. We can explore the relation with the code

```
for k in range(10):
  a = len(data)
  b = sys.getsizeof(data)
  print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
  data.append(None)
```

We can see that the size in memory increases exponentially. This is because the number of spots added is proportional to the size of the array at a time in Python lists. The amount of space that is dynamically added to the array at a time increases as the list grows (it will not always be 4).

Let us now implement a dynamic array ourselves

```
import ctypes

class DynamicArray:
  """A Dynamic array class akin to a simplified python list"""

  def __init__(self):
    """Create an empty array"""
    self._n = 0
    self._capacity = 1
    self._A = self._make_array(self._capacity)

  def __len__(self):
    """Return number of elements stored in array"""
    return self._n

  def __getitem__(self, k):
    """Return item at index k"""
    if not 0 <= k < self._n:
      raise IndexError("Invalid index")
    return self._A[k]

  def _resize(self, c)
    """Resize internal array to capacity c"""
    B = self._make_array(c)
    for k in range(self._n):
      B[k] = self._A[k]
    self._A = B
    self._capacity = c

  def _make_array(self, c):
    """Return a new array with capacity c"""
    return (C * ctypes.py_object)()

  def append(self, obj):
    """adds object to end of the array"""
    if self._n == self._capacity:
      self._resize(2*self._capacity)
      self._A[self._n] = obj
      self._n += 1

```

We can test it out using:

```
arr = DynamicArray()
arr.append(1)
```

We now would like to analyze the efficiency.

Reminder: $O(n)$ means *at most* n efficent and $\Omega(n)$ means *at least* n. Here we have $\Omega(n)$ efficiency.

Using amortization, we can show that an append is $O(1)$, and n appends are $O(n)$. Namely, a single append does not depend on the size of the array.

Proposition 5.1: Let S be a sequence implemented using a dynamic array with initial capacity 1. Using the strategy of doubling the array size when full, the total time to perform n operations is $O(n)$.

Justification: Assume 1 cyber-dollar pays for each append where we do not expand array. Assume growing array from size k to $2k$ requires k cyber-dollars. We charge each append 3 cyber-dollars. An overflow occurs when array S has 2^i elements for some integer $i \geq 0$ and the size of the array representing S is 2^i. Namely, doubling the size of the array will cost 2^i cyber-dollars. These cyber-dollars can be found in cells 2^{i-1} through $2^i - 1$.

Proposition 5.2: Performing a series of n append operations using a fixed increment with each resize takes $\Omega(n^2)$.

Justification: Let $c \geq 0$ represent a fixed increment in capacity used for each resize event. During the series of append operations, time will be spent initializing arrays of size c, $2c$, $3c$, ..., mc for $m = \frac{n}{2}$. So the overall time is proportional to $c + 2c + 3c + ... + mc = \sum_{i=1}^{m} ci = c\sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq \frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2}$. This is $\Omega(n^2)$.

3.4 Multidimensional Arrays

If we do

```
l = [[9]*10]*5
l[0][3] = 5
```

It will change the values of throughout the array. Instead, the correct method to instantiate a multidimensional array is

```
l = [[9]* 10 for j in range(5)]
l[0][3] = 5
```

This will work by ensuring that they do not all point to the same value.

3.5 Numpy Arrays

For big data, we can use the NumPy module. We can make an nd-array with

```
z = np.zeros((10, 10)) # will create a 10x10 array of 0s
```

There are many different methods that NumPy has which are highly optimized in precomplied C-libraries.