CS 304 Notes

Data Structures & Algorithms

Jaeden Bardati

Last modified January 21, 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

```
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.

TO BE CONTINUED . . .

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. $A = [3, 3, 1]$

Algorithm 1 InsertionSort(A)

```
for ($j = 2$ to length($A$)) do
    set key = $A[j]$
    // insert $A[j]$ into the
    // sorted sequence $A[1..j-1]$
    set $i = j - 1$
    while ($i > 0$ and $A[i]$ >key) do
        set $A[i+1] = A[i]$
        set $i = i - 1$
        set $A[i+1]$ = key
stop
```

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

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

<u>**Best case**</u>: $t_j = 0$

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

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.

<u>Worse case</u>: $t_j = j$ 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 Notation for Algorithm efficiency

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 $\Theta(n^2)$.

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

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

INSERT GRAPH 1

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. $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2 \; \forall \; n \geq n_0$.

Diving 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 $n_0 \geq 7$.

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

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

INSERT GRAPH 2 HERE

It is important to note that if $f(n) = \Theta(g(n))$, then necessarily $f(n) = O(g(n))$.