CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

August 28, 2017

Outline

Textbook

Data Structures and Algorithms Using Python and C++
David M. Reed and John Zelle

Outline

Functional Abstraction

Functional Abstraction

A **functional abstraction** (or simply **abstraction**) is the **external** view of a function, as seen from the program that calls it. This view is sometimes called the function's **interface**.

Interface

The function interface consists of its **specification** (what the function does) and its **signature** (function name, list of parameter types, and the type of the value returned by the function).

Interface Example: sqrt

```
def sqrt(x):
    """Computes the square root of x"""
```

Functional Abstraction

Functional Abstraction

A **functional abstraction** (or simply **abstraction**) is the **external** view of a function, as seen from the program that calls it. This view is sometimes called the function's **interface**.

Interface

The function interface consists of its **specification** (what the function does) and its **signature** (function name, list of parameter types, and the type of the value returned by the function).

Interface Example: `sqrt`

```
def sqrt(x):
    """Computes the square root of x"""
```

Functional Abstraction

Functional Abstraction

A **functional abstraction** (or simply **abstraction**) is the **external** view of a function, as seen from the program that calls it. This view is sometimes called the function's **interface**.

Interface

The function interface consists of its **specification** (what the function does) and its **signature** (function name, list of parameter types, and the type of the value returned by the function).

Interface Example: `sqrt`

```
def sqrt(x):
    """Computes the square root of x"""
```

Design by Contract

Design by Contract

The program calling a function can be considered its **client**. **Design by contract** is formal specification of the **preconditions** (true before the function call) and **postconditions** (true after the function call).

Preconditions

Preconditions are what the client must provide if the function is to be expected to perform its task, as the client's part of the contract.

Postconditions

Postconditions must be true after the function call, as the function's part of the contract.

Design by Contract

Design by Contract

The program calling a function can be considered its **client**.
Design by contract is formal specification of the **preconditions** (true before the function call) and **postconditions** (true after the function call).

Preconditions

Preconditions are what the client must provide if the function is to be expected to perform its task, as the client's part of the contract.

Postconditions

Postconditions must be true after the function call, as the function's part of the contract.

Design by Contract

Design by Contract

The program calling a function can be considered its **client**. **Design by contract** is formal specification of the **preconditions** (true before the function call) and **postconditions** (true after the function call).

Preconditions

Preconditions are what the client must provide if the function is to be expected to perform its task, as the client's part of the contract.

Postconditions

Postconditions must be true after the function call, as the function's part of the contract.

Defensive Programming

Defensive Programming

Defensive Programming is writing code that checks that all preconditions are met before allowing the program to proceed.

Testing for preconditions

Testing can be performed in various ways: conditional statements, raising exceptions, and making assertions.

Testing for preconditions

Conditional Statements

A negative value returned means error. Each violation of a function precondition needs its own conditional statement in the calling program, which is inefficient.

```
def sqrt(x):
    if x < 0:
        return -1
    ...
```

Testing for preconditions

Exception Handling

This is better. Different types of error are handled together.

```
try:
   y = sqrt(x)
except ValueError:
   print('bad parameter for sqrt')

def sqrt(x):
   if x < 0:
      raise ValueError('math domain error')
   if type(x) not in (type(1), type(1l),
type(1.0)):
      raise TypeError('number expected')
   ...
```

Testing for preconditions

Assertions

With assertions, checking can be turned off when the code is compiled for production, but this is sometimes risky.

```
   def sqrt(x):
      assert x >= 0 and type(x) in (type(1), type(1l),
type(1.0))
      ...
```

Top-down Design

Top-down Design

Top-down Design is the decomposition of a single task into several smaller ones. Each task can the be written as a single function. This can be repeated, producing an abstraction hierarchy in which high-level functions call lower level functions until the programming language itself provides its built-in functions at the bottom level.

Top-down Design

API

An **application programming interface** or **API** is the interface for a collection of functions performing related tasks. They are at the same level of abstraction, and when implemented they can call each other.

Top-down Design

Example

Example: The task is to calculate some statistics (high score, low score, mean, standard deviation) for a given set of data (student exam scores). This overall task can be divided up:

```
get scores
calculate minimum score
calculate maximum score
calculate average score
calculate standard deviation
```

Once this “top level” has been designed, each subtask can be designed separately. The top level only uses the abstraction of each function.

Top-down Design

Side Effects

A **side effect** is an unexpected result of a function call which modifies some variable or object in the environment of the function call. These should always be documented as postconditions. Many bugs are caused by undocumented side effects.

Algorithm Analysis

Algorithms

An **algorithm** is a step-by-step procedure defined for a mathematical model of computing. It can use variables, conditional expressions, looping, and sequences of steps. It can be expressed in pseudocode, flowcharts, or languages such as Python, C++ or even English.

Algorithm Analysis

Program Performance

A program will take a certain amount of time to run, and it will use other resources as well, such as space (memory). This depends on the size of the input to the program (how many data items are being processed) and other factors as well, such as the model of the computer. It is important to use an algorithm which will guarantee to the user that for reasonably sized inputs the program will run quickly.

Algorithm Analysis

Asymptotic Analysis

Asymptotic analysis is a way to compare algorithms to decide which one will have a faster running time for most inputs. This is done by determining the function $T(n)$ (which gives the running time of a given algorithm, depending on the input size n) and then determining the behavior of this function as n becomes large.

Algorithm Analysis

Some Math

Suppose that for one algorithm the running time is $T_1(n) = 4n + 2$ while for a second the running time is $T_2(n) = 3n^2$. Recall from precalculus that the rational function

$$r(x) = \frac{T_1(x)}{T_2(x)} = \frac{4x+2}{3x^2} = \frac{4x}{3x^2} + \frac{2}{3x^2} = \frac{4}{3x} + \frac{2}{3x^2}$$

approaches zero as x approaches infinity. (So the graph of $y = r(x)$ has an asymptote on the x-axis, the line $y = 0$). The numerator, $4x + 2$, becomes insignificant when compared with the denominator, $3x^2$ as x goes to infinity. So for large enough n, $T_2(n)$ is greater than $T_1(n)$ because it grows faster. If we input size by a factor of 10, the second algorithm gets about 100 times slower; the first only gets 10 times slower.

Algorithm Analysis

Theta (Θ) Notation

Θ-Notation is an abbreviation for the ideas in the last slide. In that slide, we would say that $T_1(n) = 4n + 2$, a linear function, is a $\Theta(n)$ function, and $T_2(n) = 3n^2$, a quadratic function, is $\Theta(n^2)$. In other words, to get the Θ of a polynomial function we drop the lower degree terms and ignore the leading coefficient.
Formally, a function $T(n)$ is $\Theta(f(n))$ if there are constants c_1, c_2 and n_0 such that for all $n > n_0$, $T(n) < c_1 f(n)$ and $T(n) > c_2 f(n)$.
In practical terms, a function $T(n)$ is $\Theta(f(n))$ if it “does what $f(n)$ does” as n gets large.

Algorithm Analysis

Some Facts About Θ Notation

- A $\Theta(n^2)$ running time will dominate (be eventually slower than) a $\Theta(n)$ running time.
- A $\Theta(1)$ running time is a constant (times 1) so it does not grow as n increases. (This is the fastest kind of algorithm.)
- A $\Theta(log\ n)$ algorithm is faster than a $\Theta(n)$ algorithm.

Algorithm Analysis

Big O Notation

There is a less precise way to classify functions called "big-O" or simply O-notation. A function $T(n)$ is $O(n^2)$, for example, if it is eventually less than cn^2 for some c. This limits how large the running time is for large n, (i.e.how slow the algorithm is), but it gives less information: there is no lower bound on the growth of the function $T(n)$–it could actually be faster than cn for some c. Thus, $T_2(n) = 3n^2$ is $O(n^2)$, but so is $T_1(n) = 4n + 2$. (Any $O(n)$ function is automatically $O(n^2)$, since it can be bounded above by a quadratic function.)

Algorithm Analysis

There are three types of analysis, possibly giving different running time functions:

- Best case is the fastest possible time. We don't use this, since all it tells us is what will happen if you are lucky. This gives no guarantee to the customer.
- Average case is how fast the algorithm runs on average, over different sets of inputs. It gives a good idea of how much time the program will use over a long period of time.
- Worst case is how fast the algorithm is guaranteed to run.

To get the running time, count each single step as 1 time unit. Then, using math, calculate how many time units it takes the entire algorithm to run if the size of the input is n.

Algorithm Analysis

First Example: Linear Search

```
def search(items, target):
    i = 0
    while i < len(items):
        if items[i] == target:
            return i
        i += 1
    return -1
```

Algorithm Analysis

Analysis of Linear Search (worst case)

- Multiply the steps in the body of the loop (2) times the number of times the loop executes (the worst case is n), giving $2n$.
- Add this to the steps outside the loop (2): $T(n) = 2n + 2$. The algorithm is $\Theta(n)$. (We see that we could have ignored the number of steps outside the loop, 2, since it is a lower degree term than $2n$.)

Algorithm Analysis

Second Example: Binary Search (Precondition: items list is sorted)

```
def search(items, target):
   low = 0
   high = len(items) - 1
   while low <= high:
      mid = (low + high) // 2
      item = items[mid]
      if target == item :
         return mid
      elif target < item:
         high = mid - 1
      else:
         low = mid + 1
   return -1
```

Algorithm Analysis

Analysis of Binary Search (worst case)

- Each time the loop executes, the list (between `high` and `low`) starts twice as big as after the loop executes.
- The algorithm starts with list size of n; it ends when the list has size 1 (worst case).
- So if $l =$ number of times the loop executes, then $n = 2^l$.
- In logarithm form, this gives $l = log_2\, n$.
- The number of steps in the loop is, on average, 5. Two steps are outside the loop. The total is $5l + 2 = 5(log_2\, n) + 2$.
- So this algorithm is $\Theta(log\ n)$. (In Θ notation, we don't care what the logarithm base is, since changing from base a to base b is just multiplying by a constant, $log_b\, a$.)
- The moral: binary search ($\Theta(log\ n)$) is faster than linear search ($\Theta(n)$).