CSC148H1 - Introduction to Computer Science

Huzaim Malik

September 29, 2024

Contents

1 Introduction to Programming **2**
- 1.1 Python Syntax and Semantics . . . 2
- 1.2 Functions . . . 2

2 Object-Oriented Programming (OOP) **2**
- 2.1 Classes and Objects . . . 2
- 2.2 Inheritance Example . . . 3

3 Recursion **3**
- 3.1 Factorial Example . . . 3
- 3.2 Recursive vs Iterative Solutions . . . 3

4 Data Structures **3**
- 4.1 Lists . . . 3
- 4.2 Stacks and Queues . . . 4
- 4.3 Trees . . . 4

5 Searching and Sorting Algorithms **4**
- 5.1 Linear Search . . . 4
- 5.2 Binary Search . . . 4

6 Big-O Notation **5**

7 Hashing and Dictionaries **5**
- 7.1 Hash Function . . . 5

8 Linked Lists **5**
- 8.1 Singly Linked List . . . 5

9 Graphs **5**
- 9.1 Graph Representation . . . 6

10 Dynamic Programming **6**

1 Introduction to Programming

Programming is the process of writing instructions for a computer to perform tasks. A programming language provides a set of rules for how these instructions should be written.

1.1 Python Syntax and Semantics

Python is a high-level programming language that emphasizes readability and simplicity. Its basic components include:

- **Variables**: Used to store data values.
- **Data types**: Integers, floats, strings, and booleans.
- **Control structures**: if-else statements, loops (for, while).

1.2 Functions

Functions are reusable blocks of code that perform specific tasks. A Python function is defined using the `def` keyword:

```
bluedef add(a, b):
    bluereturn a + b
```

2 Object-Oriented Programming (OOP)

Object-oriented programming is a paradigm based on the concept of "objects," which can contain data and code. The four main principles of OOP are:

- **Encapsulation**: Bundling of data with methods that operate on that data.
- **Abstraction**: Hiding the complexity of certain operations.
- **Inheritance**: Creating a new class from an existing class.
- **Polymorphism**: The ability to process objects differently depending on their class.

2.1 Classes and Objects

A class is a blueprint for creating objects (instances). Here's an example of a class in Python:

```
blueclass Car:
    bluedef __init__(self, make, model, year):
        self.make = make
        self.model = model
```

```
        self.year = year

    bluedef display_info(self):
        blueprint(fred'red{redselfred.redyearred}red␣red{redselfred.redmakered}re
```

2.2 Inheritance Example

```
blueclass ElectricCar(Car):
    bluedef __init__(self, make, model, year, battery_size):
        bluesuper().__init__(make, model, year)
        self.battery_size = battery_size
```

3 Recursion

Recursion is a process where a function calls itself as a subroutine. This allows problems to be solved in smaller parts.

3.1 Factorial Example

The factorial of a number n is the product of all positive integers less than or equal to n. This can be computed using recursion:

```
bluedef factorial(n):
    blueif n == 0:
        bluereturn 1
    blueelse:
        bluereturn n * factorial(n - 1)
```

3.2 Recursive vs Iterative Solutions

While recursion can be elegant, it may lead to high memory consumption due to the call stack. Iterative solutions may be more memory efficient.

4 Data Structures

Data structures are ways to organize and store data efficiently.

4.1 Lists

Lists are ordered collections of items in Python. Lists are mutable and can hold different data types.

```
my_list = [1, 2, 3, red"redfourred"]
```

4.2 Stacks and Queues

- **Stack**: A Last-In-First-Out (LIFO) data structure.
- **Queue**: A First-In-First-Out (FIFO) data structure.

4.3 Trees

A tree is a hierarchical data structure with nodes. A binary tree is a type of tree where each node has at most two children.

```
blueclass Node:
    bluedef __init__(self, value):
        self.left = None
        self.right = None
        self.value = value
```

5 Searching and Sorting Algorithms

5.1 Linear Search

Linear search checks each element of a list until the target is found.

```
bluedef linear_search(lst, target):
    bluefor i bluein bluerange(bluelen(lst)):
        blueif lst[i] == target:
            bluereturn i
    bluereturn -1
```

5.2 Binary Search

Binary search is more efficient but requires a sorted list. It works by dividing the list into halves.

```
bluedef binary_search(lst, target):
    low = 0
    high = bluelen(lst) - 1
    bluewhile low <= high:
        mid = (low + high) // 2
        blueif lst[mid] == target:
            bluereturn mid
        blueelif lst[mid] < target:
            low = mid + 1
        blueelse:
            high = mid - 1
    bluereturn -1
```

6 Big-O Notation

Big-O notation is used to describe the time complexity of algorithms. It helps in analyzing the efficiency of an algorithm.

- $O(1)$: Constant time
- $O(n)$: Linear time
- $O(n^2)$: Quadratic time

7 Hashing and Dictionaries

Hashing allows for efficient data retrieval. Python's `dict` is a hash table, which allows for fast key-value lookups.

7.1 Hash Function

A hash function takes an input (or 'key') and returns an integer (hash value) which determines the position of the key-value pair in the table.

8 Linked Lists

A linked list is a linear data structure where each element is a separate object called a node. Each node contains the data and a reference to the next node.

8.1 Singly Linked List

```
blueclass Node:
    bluedef __init__(self, data):
        self.data = data
        self.bluenext = None

blueclass LinkedList:
    bluedef __init__(self):
        self.head = None
```

9 Graphs

Graphs consist of nodes (vertices) and edges. They can be directed or undirected, weighted or unweighted.

9.1 Graph Representation

- **Adjacency Matrix**: A 2D array to represent edges.
- **Adjacency List**: A list where each element is a list of nodes connected to a vertex.

10 Dynamic Programming

Dynamic programming is a technique used to solve complex problems by breaking them into smaller sub-problems, storing the solutions to sub-problems to avoid redundant calculations.

```
bluedef fib(n, memo={}):
    blueif n bluein memo:
        bluereturn memo[n]
    blueif n <= 2:
        bluereturn 1
    memo[n] = fib(n-1) + fib(n-2)
    bluereturn memo[n]
```