Debugging techniques:

1. **Print** particular variables and check if their values are as expected or not. The disadvantage is that you have to delete it in the future => garbage information.
    - print inside *if-else block, for, while loop, check if the function is called*.
2. **Assert and Check** can raise Assertion error whenever there is a violation in the program statement: *assert(isEven(4) == true)*
3. **Logging**

```
import logging

logging.basicConfig(filename='test.log', filemode='w', format='%(name)s - %(levelname)s
- %(message)s')
logging.warning('This will get logged to a file')
```

4. **PDB** - python debugger built-in build tool.

```
python -m pdb test.py
enter n to debug line by line
```

5. **IDE** to set breakpoints and single-step execution comfortably on source files.
   breakpoints:
    - **play**: choose the line and the program will be executed until the next breakpoint, and check if the data is passed to a variable at that state.
    - **skip**: moves one line forward
    - **step into**: moves to the next level of scope outside of the current function
6. **Pen and Paper**

Debugging strategies:

1. **Know what you're looking for**

   focus on what data you're expecting and what you're receiving, then go to the step that could be the cause.

2. **Explain the problem to someone**

   explain it to someone who's not a programmer, could be a friend but explain it line by line so you will find out where the bug is.

3. **Figure out how to reproduce the problem**

   In order to reproduce a problem, **you must cause the problem**. If you have caused the problem, you may be able to identify the particular thing you did that caused it.

4. **Find the exact location of the error**

Use the debugger or print statements to find where the output goes wrong and trace it back to the origin of the bug.

5. **Make methodical changes**
   Make one change at a time and test it, particularly if you're not confident how it will change the result. Otherwise you may not be sure what actually solved the bug.

6. **Ask fo help if you're stuck**
   Ask help if you're stuck there in 30 minutes.

Programming constructs (sequence, selections, loops) in designing algorithm:

Programs are designed using common building blocks(programming constructs). They form the basis of all programs.

1. **Sequence** statements are the order in which instructions occur and are processed. example: seq of statements.
2. **Selections** determine which path a program takes when it is running. example: if-else, switch-case.
3. **Iteration** is the repeated execution of a section of code when a program is running. example: for and while loop.
   definite iteration: count-controlled iteration : for loop
   indefinite iteration: condition-controlled iteration : while loop

Object orientation (OOP): `create object in python:`

```
class MyClass:
    x = 5

myobject = MyClass()
print(myobject.x) #5
```

`Self in python vs This in java;`
Technically both self and this are used for the same thing. They are used to access the variable associated with the current instance. Only difference is, you have to include self explicitly as the first parameter to an instance method in Python, whereas this is not the case with Java.

Inheritance

It uses **extends** as a keyword and allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called **base** class.
Child class is the class that inherits from another class, also called **derived** class.

```python
# The parent should be passed in as parameter
class Student(Person):

  # overrides the parent constructor function
  def __init__(self, fname, lname):

    # inherits all the methods and properties from the parent
    # self is passed out as a parameter
    Person.__init__(self, fname, lname)

    # inherits all the methods and properties from the parent
    # does not pass out self
    super().__init__(fname, lname)
```

Encapsulation - Abstract data types:
Encapsulation distinguishes code's public **interface** from its **private** implementation.

Polymorphism - using multiple child instances inherited(not necessarily) from parents

Interface

Interface is an abstract class. It doesn't have a body. To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement" class.

We can not use interfaces to create objects from itself directly.
When we implement the interface we should override all the methods.
Interface methods are by default abstract and public.
interface attributes are by default public static final.

Why interface?

1) To achieve security - hide certain details and only show the important details of an object (interface)
2) It allows multiple inheritance in Java

```
// Multiple inheritance
class Base1:
    pass

class Base2:
    pass

class MultiDerived(Base1, Base2):
    pass

// interface
```

Concurrency
not parallelism, but doing multiple sequences of operations that are run in overlapping periods of time. To avoid threads intertwining, we can add a **synchronized** keyword in the variable so that it can be accessed one by one(serialized) in the memory.

Locks could cause Deadlock when two or more threads are blocked forever, waiting for each other. **Deadlock occurs** when multiple threads need the same locks but obtain them in different order. **Changing the order of locks** prevents the program from deadlock.

Locks are one synchronization technique. A lock is an abstraction that allows at most one thread to *own* it at a time. *Holding a lock* is how one thread tells other threads: “I’m changing this thing, don’t touch it right now.”

The difference between lock and synchronized is you can **release** and **acquire** locks whenever you want.

Threads
Race conditions

Deadlock vs starvation:
Deadlock:
- no process proceeds and get blocked

Starvation:

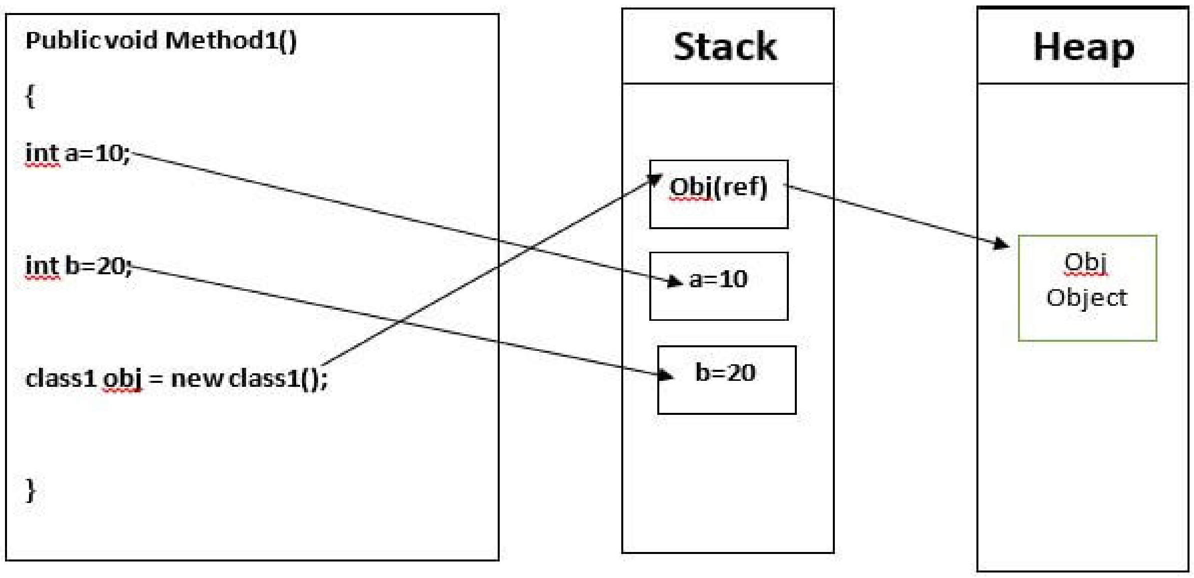
- low priority processes get blocked by high priority and never run or have access to the resource

Binding: linking between method call and method definition. It answers the question for: For particular method calls which specific method is invoked?
Static vs Dynamic binding
Static: uses **type information** for binding, done at **compile-time**, method **overloading**.
Dynamic: uses **objects** to resolve binding, done at **run-time**, method **overriding**.

Static vs Dynamic typing

statically typed languages perform type checking at compile time, to optimize hardware efficiency. example lang: java, c++.
dynamically typed languages perform type checking at run-time, to optimize programmer efficiency. example lang: python. the type of the variable can be changed directly over time.

Stack vs heap



Strong vs weak typing

Strong vs Weak Typing

Strong typing:

Every variable must be declared with a type.

A variable may receive only values of its type.

Type-related bugs can be reduced.

Type identification tags are not needed a run time.

Weak typing:

Variables need not be declared with particular types.

The type of value held by a variable can change dynamically.

Values must be tagged during execution with their types.