

The Complete Coding Interview Guide in Java

An effective guide for aspiring Java developers to ace their programming interviews



Packt

www.packt.com

Anghel Leonard

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/The-Complete-Coding-Interview-Guide-in-Java>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Chapter 6

Images

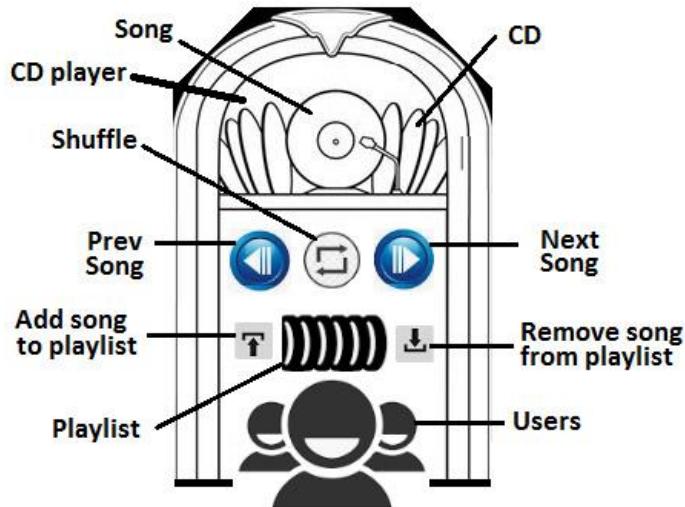


Figure 6.1 – Jukebox

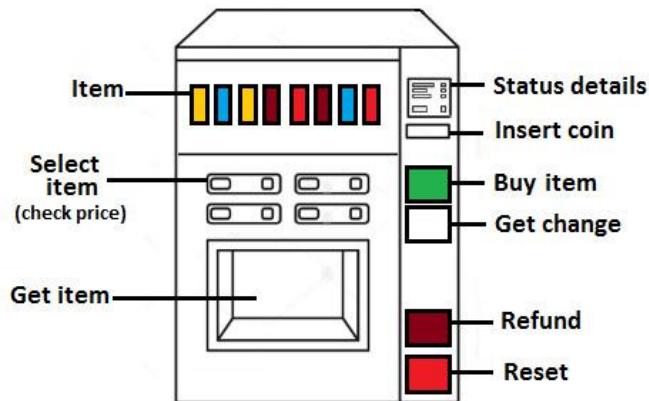


Figure 6.2 – Vending machine

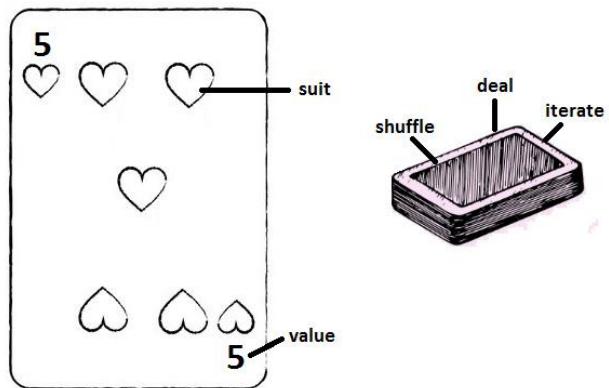


Figure 6.3 – A deck of cards

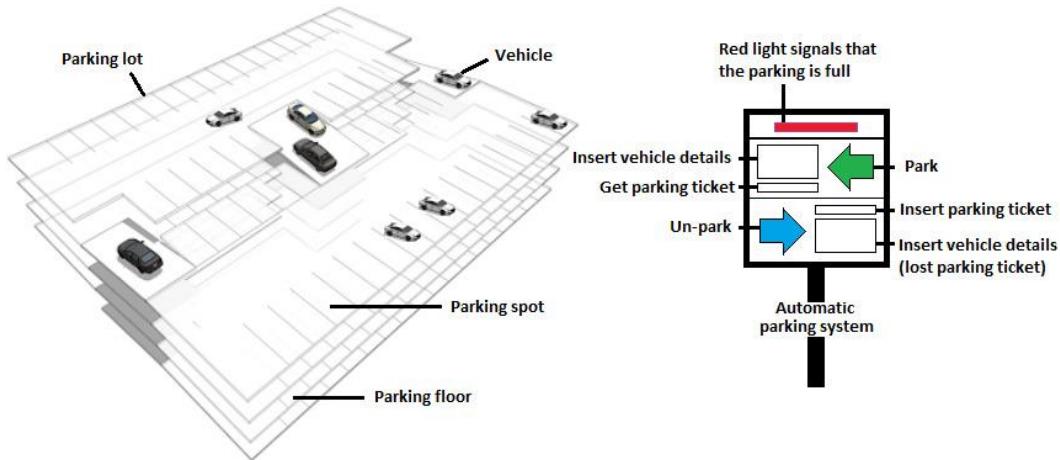


Figure 6.4 – A parking lot

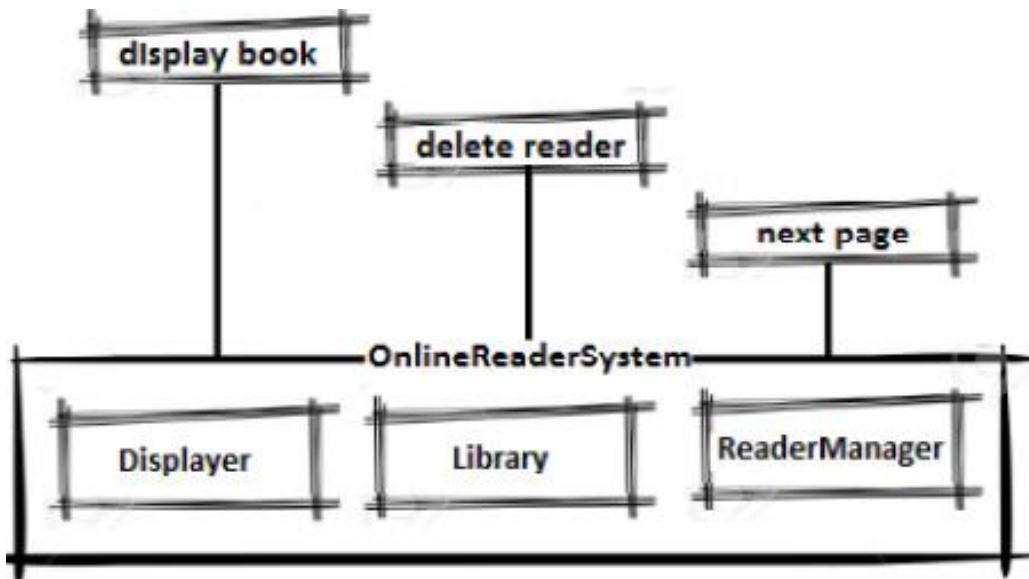


Figure 6.5 – An online reader system

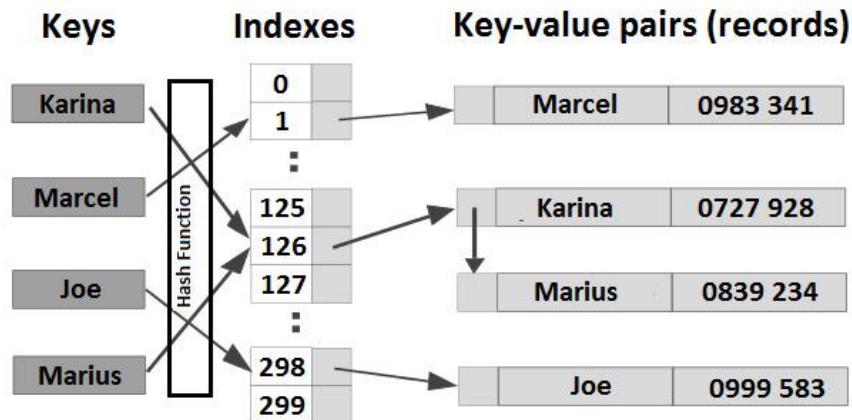


Figure 6.6 – A hash table

Code

Abstraction

Code 6.1

```
public interface Car {
    public void speedUp();
    public void slowDown();
    public void turnRight();
    public void turnLeft();
    public String getCarType();
}
```

Code 6.2

```
public class ElectricCar implements Car {
    private final String carType;
    public ElectricCar(String carType) {
        this.carType = carType;
    }
    @Override
    public void speedUp() {
        System.out.println("Speed up the electric car");
    }
}
```

```

@Override
public void slowDown() {
    System.out.println("Slow down the electric car");
}

@Override
public void turnRight() {
    System.out.println("Turn right the electric car");
}

@Override
public void turnLeft() {
    System.out.println("Turn left the electric car");
}

@Override
public String getCarType() {
    return this.carType;
}

}

```

Code 6.3

```

public class Main {

    public static void main(String[] args) {
        Car electricCar = new ElectricCar("BMW");
        System.out.println("Driving the electric car: "
+ electricCar.getCarType() + "\n");
        electricCar.speedUp();
        electricCar.turnLeft();
        electricCar.slowDown();
    }
}

```

Output - Code 6.4

Driving the electric car: BMW

```
Speed up the electric car  
Turn left the electric car  
Slow down the electric car
```

Encapsulation

Code 6.5

```
public class Cat {  
  
    private int mood = 50;  
  
    private int hungry = 50;  
  
    private int energy = 50;  
  
    public void sleep() {  
  
        System.out.println("Sleep ...");  
  
        energy++;  
  
        hungry++;  
  
    }  
  
    public void play() {  
  
        System.out.println("Play ...");  
  
        mood++;  
  
        energy--;  
  
        meow();  
  
    }  
  
    public void feed() {  
  
        System.out.println("Feed ...");  
  
        hungry--;  
  
        mood++;  
  
        meow();  
  
    }  
  
    private void meow() {  
  
        System.out.println("Meow!");  
  
    }  
  
    public int getMood() {
```

```

        return mood;
    }

    public int getHungry() {
        return hungry;
    }

    public int getEnergy() {
        return energy;
    }

}

```

Code 6.6

```

public static void main(String[] args) {
    Cat cat = new Cat();
    cat.feed();
    cat.play();
    cat.feed();
    cat.sleep();
    System.out.println("Energy: " + cat.getEnergy());
    System.out.println("Mood: " + cat.getMood());
    System.out.println("Hungry: " + cat.getHungry());
}

```

Output – code 6.7

```

Feed ...Meow!Play ...Meow!Feed ...Meow!Sleep ...
Energy: 50
Mood: 53
Hungry: 49

```

Inheritance

Code 6.8

```

public class Employee {
    private String name;
    public Employee(String name) {

```

```
        this.name = name;  
    }  
    // getters and setters omitted for brevity  
}
```

Code 6.9

```
public class Programmer extends Employee {  
    private String team;  
    public Programmer(String name, String team) {  
        super(name);  
        this.team = team;  
    }  
    // getters and setters omitted for brevity  
}
```

Code 6.10

```
public static void main(String[] args) {  
    Programmer p = new Programmer("Joana Nimar", "Toronto");  
    String name = p.getName();  
    String team = p.getTeam();  
    System.out.println(name + " is assigned to the "  
        + team + " team");  
}
```

Output – Code 6.11

```
Joana Nimar is assigned to the Toronto team
```

Polymorphism

Polymorphism via method overloading (compile time)

Code 6.12

```
public class Triangle {  
    public void draw() {  
        System.out.println("Draw default triangle ...");
```

```

    }

    public void draw(String color) {
        System.out.println("Draw a triangle of color "
            + color);
    }

    public void draw(int size, String color) {
        System.out.println("Draw a triangle of color " + color
            + " and scale it up with the new size of " + size);
    }
}

```

Code 6.13

```

public static void main(String[] args) {
    Triangle triangle = new Triangle();
    triangle.draw();
    triangle.draw("red");
    triangle.draw(10, "blue");
}

```

Output – Code 6.14

```

Draw default triangle ...

Draw a triangle of color red

Draw a triangle of color blue and scale it up
with the new size of 10

```

Polymorphism via method overriding (runtime)

Code 6.15

```

public interface Shape {
    public void draw();
}

```

Code 6.16

```

public class Triangle implements Shape {
    @Override

```

```

public void draw() {
    System.out.println("Draw a triangle ...");
}

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Draw a rectangle ...");
    }
}

public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Draw a circle ...");
    }
}

```

Code 6.17

```

public static void main(String[] args) {
    Shape triangle = new Triangle();
    Shape rectangle = new Rectangle();
    Shape circle = new Circle();
    triangle.draw();
    rectangle.draw();
    circle.draw();
}

```

Output – Code 6.18

```

Draw a triangle ...
Draw a rectangle ...
Draw a circle ...

```

Association

Code 6.19

```
public class Person {  
    private String name;  
    public Person(String name) {  
        this.name = name;  
    }  
    // getters and setters omitted for brevity  
}  
  
public class Address {  
    private String city;  
    private String zip;  
    public Address(String city, String zip) {  
        this.city = city;  
        this.zip = zip;  
    }  
    // getters and setters omitted for brevity  
}
```

Code 6.20

```
public static void main(String[] args) {  
    Person p1 = new Person("Andrei");  
    Person p2 = new Person("Marin");  
    Address a1 = new Address("Banesti", "107050");  
    Address a2 = new Address("Bucuresti", "229344");  
    // Association between classes in the main method  
    System.out.println(p1.getName() + " lives at address "  
        + a2.getCity() + ", " + a2.getZip()  
        + " but it also has an address at "  
        + a1.getCity() + ", " + a1.getZip());  
    System.out.println(p2.getName() + " lives at address "  
        + a1.getCity() + ", " + a1.getZip())
```

```
        + " but it also has an address at "
        + a2.getCity() + ", " + a2.getZip());
}
```

Output – Code 6.21

Andrei lives at address Bucuresti, 229344 but it also has an address at Banesti, 107050

Marin lives at address Banesti, 107050 but it also has an address at Bucuresti, 229344

Aggregation

Code 6.22

```
public class Racket {

    private String type;
    private int size;
    private int weight;

    public Racket(String type, int size, int weight) {
        this.type = type;
        this.size = size;
        this.weight = weight;
    }

    // getters and setters omitted for brevity
}
```

Code 6.23

```
public class TennisPlayer {

    private String name;
    private Racket racket;

    public TennisPlayer(String name, Racket racket) {
        this.name = name;
        this.racket = racket;
    }

    // getters and setters omitted for brevity
}
```

Code 6.24

```
public static void main(String[] args) {  
    Racket racket = new Racket("Babolat Pure Aero", 100, 300);  
    TennisPlayer player = new TennisPlayer("Rafael Nadal",  
        racket);  
    System.out.println("Player " + player.getName()  
        + " plays with " + player.getRacket().getType());  
}
```

Output – Code 6.25

```
Player Rafael Nadal plays with Babolat Pure Aero
```

Composition

Code 6.26

```
public class Engine {  
    private String type;  
    private int horsepower;  
    public Engine(String type, int horsepower) {  
        this.type = type;  
        this.horsepower = horsepower;  
    }  
    // getters and setters omitted for brevity  
}
```

Code 6.27

```
public class Car {  
    private final String name;  
    private final Engine engine;  
    public Car(String name) {  
        this.name = name;  
        Engine engine = new Engine("petrol", 300);  
        this.engine=engine;  
    }
```

```

public int getHorsepower() {
    return engine.getHorsepower();
}

public String getName() {
    return name;
}

}

```

Code 6.28

```

public static void main(String[] args) {
    Car car = new Car("MyCar");
    System.out.println("Horsepower: " + car.getHorsepower());
}

```

Output – Code 6.29

```
Horsepower: 300
```

Single Responsibility Principle

Code 6.30

```

public class RectangleAreaCalculator {

    private static final double INCH_TERM = 0.0254d;
    private final int width;
    private final int height;

    public RectangleAreaCalculator(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int area() {
        return width * height;
    }

    // this method breaks SRP

    public double metersToInches(int area) {
        return area / INCH_TERM;
    }
}

```

```
    }  
}
```

Code 6.31

```
public class RectangleAreaCalculator {  
  
    private final int width;  
  
    private final int height;  
  
    public RectangleAreaCalculator(int width, int height) {  
  
        this.width = width;  
  
        this.height = height;  
  
    }  
  
    public int area() {  
  
        return width * height;  
  
    }  
  
}
```

Code 6.32

```
public class AreaConverter {  
  
    private static final double INCH_TERM = 0.0254d;  
  
    private static final double FEET_TERM = 0.3048d;  
  
    public double metersToInches(int area) {  
  
        return area / INCH_TERM;  
  
    }  
  
    public double metersToFeet(int area) {  
  
        return area / FEET_TERM;  
  
    }  
}
```

Open Closed Principle

Code 6.33

```
public interface Shape {  
}  
  
public class Rectangle implements Shape {
```

```

    private final int width;
    private final int height;
    // constructor and getters omitted for brevity
}

public class Circle implements Shape {
    private final int radius;
    // constructor and getter omitted for brevity
}

```

Code 6.34

```

public class AreaCalculator {
    private final List<Shape> shapes;
    public AreaCalculator(List<Shape> shapes) {
        this.shapes = shapes;
    }
    // adding more shapes requires us to modify this class
    // this code is not OCP compliant
    public double sum() {
        int sum = 0;
        for (Shape shape : shapes) {
            if (shape.getClass().equals(Circle.class)) {
                sum += Math.PI * Math.pow(((Circle) shape)
                    .getRadius(), 2);
            } else
                if(shape.getClass().equals(Rectangle.class)) {
                    sum += ((Rectangle) shape).getHeight()
                        * ((Rectangle) shape).getWidth();
                }
        }
        return sum;
    }
}

```

```
}
```

Code 6.35

```
public interface Shape {  
    public double area();  
}
```

Code 6.36

```
public class Rectangle implements Shape {  
    private final int width;  
    private final int height;  
    public Rectangle(int width, int height) {  
        this.width = width;  
        this.height = height;  
    }  
    public double area() {  
        return width * height;  
    }  
}  
  
public class Circle implements Shape {  
    private final int radius;  
    public Circle(int radius) {  
        this.radius = radius;  
    }  
    @Override  
    public double area() {  
        return Math.PI * Math.pow(radius, 2);  
    }  
}
```

Code 6.37

```
public class AreaCalculator {  
    private final List<Shape> shapes;
```

```

public AreaCalculator(List<Shape> shapes) {
    this.shapes = shapes;
}

public double sum() {
    int sum = 0;
    for (Shape shape : shapes) {
        sum += shape.area();
    }
    return sum;
}

}

```

Liskov's Substitution Principle

Code 6.38

```

public abstract class Member {
    private final String name;
    public Member(String name) {
        this.name = name;
    }
    public abstract void joinTournament();
    public abstract void organizeTournament();
}

```

Code 6.39

```

public class PremiumMember extends Member {
    public PremiumMember(String name) {
        super(name);
    }
    @Override
    public void joinTournament() {
        System.out.println("Premium member joins
tournament");
    }
}

```

```

    }

    @Override

    public void organizeTournament() {
        System.out.println("Premium member organize
                           tournament");
    }

}

```

Code 6.40

```

public class FreeMember extends Member {

    public FreeMember(String name) {
        super(name);
    }

    @Override

    public void joinTournament() {
        System.out.println("Classic member joins tournament
                           ...");
    }

    // this method breaks Liskov's Substitution Principle

    @Override

    public void organizeTournament() {
        System.out.println("A free member cannot organize
                           tournaments");
    }

}

```

Code 6.41

```

List<Member> members = List.of(
    new PremiumMember("Jack Hores"),
    new VipMember("Tom Johns"),

```

```
    new FreeMember("Martin Vilop")
);
```

Code 6.42

```
for (Member member : members) {
    member.organizeTournament();
}
```

Code 6.43

```
public interface TournamentJoiner {
    public void joinTournament();
}

public interface TournamentOrganizer {
    public void organizeTournament();
}
```

Code 6.44

```
public abstract class Member
    implements TournamentJoiner, TournamentOrganizer {
    private final String name;
    public Member(String name) {
        this.name = name;
    }
}
```

Code 6.45

```
public class FreeMember implements TournamentJoiner {
    private final String name;
    public FreeMember(String name) {
        this.name = name;
    }
    @Override
    public void joinTournament() {
        System.out.println("Free member joins tournament ...");
    }
}
```

```
    }  
}
```

Code 6.46

```
List<TournamentJoiner> members = List.of(  
    new PremiumMember("Jack Hores"),  
    new PremiumMember("Tom Johns"),  
    new FreeMember("Martin Vilop")  
) ;
```

Code 6.47

```
// this code respects LSP  
for (TournamentJoiner member : members) {  
    member.joinTournament();  
}
```

Code 6.48

```
List<TournamentOrganizer> members = List.of(  
    new PremiumMember("Jack Hores"),  
    new VipMember("Tom Johns")  
) ;
```

Code 6.49

```
// this code respects LSP  
for (TournamentOrganizer member : members) {  
    member.organizeTournament();  
}
```

Interface Segregation Principle

Code 6.50

```
public interface Connection {  
    public void socket();  
    public void http();  
    public void connect();  
}
```

Code 6.51

```
public class WwwPingConnection implements Connection {  
    private final String www;  
    public WwwPingConnection(String www) {  
        this.www = www;  
    }  
    @Override  
    public void http() {  
        System.out.println("Setup an HTTP connection to "  
            + www);  
    }  
    @Override  
    public void connect() {  
        System.out.println("Connect to " + www);  
    }  
    // this method breaks Interface Segregation Principle  
    @Override  
    public void socket() {  
    }  
}
```

Code 6.52

```
WwwPingConnection www  
    = new WwwPingConnection('www.yahoo.com');  
www.socket(); // we can call this method!  
www.connect();
```

Code 6.53

```
public interface Connection {  
    public void connect();  
}
```

Code 6.54

```

public interface HttpConnection extends Connection {
    public void http();
}

public interface SocketConnection extends Connection {
    public void socket();
}

```

Code 6.55

```

public class WwwPingConnection implements HttpConnection {
    private final String www;
    public WwwPingConnection(String www) {
        this.www = www;
    }
    @Override
    public void http() {
        System.out.println("Setup an HTTP connection to "
            + www);
    }
    @Override
    public void connect() {
        System.out.println("Connect to " + www);
    }
}

```

Dependency Inversion Principle

Code 6.56

```

public class PostgreSQLJdbcUrl {
    private final String dbName;
    public PostgreSQLJdbcUrl(String dbName) {
        this.dbName = dbName;
    }
    public String get() {

```

```
        return "jdbc:// ... " + this.dbName;  
    }  
}  
  
public class ConnectToDatabase {  
  
    public void connect(PostgreSQLJdbcUrl postgresql) {  
  
        System.out.println("Connecting to "  
            + postgresql.get());  
    }  
}
```

Code 6.57

```
public interface JdbcUrl {  
  
    public String get();  
}
```

Code 6.58

```
public class PostgreSQLJdbcUrl implements JdbcUrl {  
  
    private final String dbName;  
  
    public PostgreSQLJdbcUrl(String dbName) {  
  
        this.dbName = dbName;  
    }  
  
    @Override  
  
    public String get() {  
  
        return "jdbc:// ... " + this.dbName;  
    }  
}
```

Code 6.59

```
public class ConnectToDatabase {  
  
    public void connect(JdbcUrl jdbcUrl) {  
  
        System.out.println("Connecting to " + jdbcUrl.get());  
    }  
}
```

Covariant method overriding

Code 6.60

```
public class Rectangle implements Cloneable {  
    ...  
    @Override  
    protected Rectangle clone()  
        throws CloneNotSupportedException {  
        Rectangle clone = (Rectangle) super.clone();  
        return clone;  
    }  
}
```

Code 6.61

Calling the `clone()` method doesn't require an explicit cast:

```
Rectangle r = new Rectangle(4, 3);  
Rectangle clone = r.clone();
```

Overriding a non-static method as static

Code 6.62

```
public interface Vehicle {  
    public void speedUp();  
    public void slowDown();  
}
```

Code 6.63

```
public class SteamCar implements Vehicle {  
    private String name;  
    // constructor and getter omitted for brevity  
    @Override  
    public void speedUp() {  
        System.out.println("Speed up the steam car ...");  
    }  
    @Override
```

```
public void slowDown() {  
    System.out.println("Slow down the steam car ...");  
}  
}
```

Code 6.65:

```
public interface Vehicle {  
    public void speedUp();  
    public void slowDown();  
    default double computeConsumption(int fuel,  
        int distance, int horsePower) {  
        // simulate the computation  
        return Math.random() * 10d;  
    }  
}
```

Code 6.66

```
public class ElectricCar implements Vehicle {  
    private String name;  
    private int horsePower;  
    // constructor and getters omitted for brevity  
    @Override  
    public void speedUp() {  
        System.out.println("Speed up the electric car ...");  
    }  
    @Override  
    public void slowDown() {  
        System.out.println("Slow down the electric car ...");  
    }  
    @Override  
    public double computeConsumption(int fuel,  
        int distance, int horsePower) {
```

```

        // simulate the computation

        return Math.random()*60d / Math.pow(Math.random(), 3);

    }

}

```

Code 6.67

```

public interface Vehicle {

    public void speedUp();

    public void slowDown();

    default double computeConsumption(int fuel,
        int distance, int horsePower) {
        return Math.random() * 10d;
    }

    static void description() {
        System.out.println("This interface control
            steam, petrol and electric cars");
    }
}

```

Method hiding

Code 6.68

```

public class Vehicle {

    public static void move() {
        System.out.println("Moving a vehicle");
    }
}

```

Code 6.69

```

public class Car extends Vehicle {

    // this method hides Vehicle#move()

    public static void move() {
        System.out.println("Moving a car");
    }
}

```

```
}
```

Code 6.70

```
public static void main(String[] args) {  
    Vehicle.move(); // call Vehicle#move()  
    Car.move(); // call Car#move()  
}
```

Output – Code 6.71

```
Moving a vehicle  
Moving a car
```

Coding challenges

Code 6.72

```
public interface Selector {  
    public void nextSongBtn();  
    public void prevSongBtn();  
    public void addSongToPlaylistBtn(Song song);  
    public void removeSongFromPlaylistBtn(Song song);  
    public void shuffleBtn();  
}  
  
public class Jukebox implements Selector {  
    private final CDPlayer cdPlayer;  
    public Jukebox(CDPlayer cdPlayer) {  
        this.cdPlayer = cdPlayer;  
    }  
    @Override  
    public void nextSongBtn() { ... }  
    // rest of Selector methods omitted for brevity  
}
```

Code 6.73

```
public class CDPlayer {  
    private CD cd;
```

```

private final Set<CD> cds;
private final Playlist playlist;
public CDPlayer(Playlist playlist, Set<CD> cds) {
    this.playlist = playlist;
    this.cds = cds;
}
protected void playNextSong() {...}
protected void playPrevSong() {...}
protected void addCD(CD cd) {...}
protected void removeCD(CD cd) {...}
// getters omitted for brevity
}

```

Code 6.74

```

public class Playlist {
    private Song song;
    private final List<Song> songs; // or Queue
    public Playlist(List<Song> songs) {
        this.songs = songs;
    }
    public Playlist(Song song, List<Song> songs) {
        this.song = song;
        this.songs = songs;
    }
    protected void addSong(Song song) {...}
    protected void removeSong(Song song) {...}
    protected void shuffle() {...}
    protected Song getNextSong() {...};
    protected Song getPrevSong() {...};
    // setters and getters omitted for brevity
}

```

Code 6.75

```
public enum Coin {  
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
    ...  
}  
  
public enum Item {  
    SKITTLES("Skittles", 15), TWIX("Twix", 35) ...  
    ...  
}
```

Code 6.76

```
public final class Inventory<T> {  
    private Map<T, Integer> inventory = new HashMap<>();  
    protected int getQuantity(T item) {...}  
    protected boolean hasItem(T item) {...}  
    protected void clear() {...}  
    protected void add(T item) {...}  
    protected void put(T item, int quantity) {...}  
    protected void deduct(T item) {...}  
}
```

Code 6.77

```
public interface Selector {  
    public int checkPriceBtn(Item item);  
    public void insertCoinBtn(Coin coin);  
    public Map<Item, List<Coin>> buyBtn();  
    public List<Coin> refundBtn();  
    public void resetBtn();  
}
```

Code 6.78

```
public class VendingMachine implements Selector {  
    private final Inventory<Coin> coinInventory
```

```

        = new Inventory<>();

private final Inventory<Item> itemInventory
        = new Inventory<>();

private int totalSales;

private int currentBalance;

private Item currentItem;

public VendingMachine() {
    initMachine();
}

private void initMachine() {
    System.out.println("Initializing the
        vending machine with coins and items ...");
}

// override Selector methods omitted for brevity
}

```

Code 6.79

```

public enum StandardSuit {
    SPADES, HEARTS, DIAMONDS, CLUBS;
}

public abstract class Card {
    private final Enum suit;
    private final int value;
    private boolean available = Boolean.TRUE;
    public Card(Enum suit, int value) {
        this.suit = suit;
        this.value = value;
    }
    // code omitted for brevity
}

public class StandardCard extends Card {

```

```

private static final int MIN_VALUE = 1;
private static final int MAX_VALUE = 13;
public StandardCard(StandardSuit suit, int value) {
    super(suit, value);
}
// code omitted for brevity
}

```

Code 6.80

```

public abstract class Pack<T extends Card> {
    private List<T> cards;
    protected abstract List<T> build();
    public int packSize() {
        return cards.size();
    }
    public List<T> getCards() {
        return new ArrayList<>(cards);
    }
    protected void setCards(List<T> cards) {
        this.cards = cards;
    }
}
public final class StandardPack extends Pack {
    public StandardPack() {
        super.setCards(build());
    }
    @Override
    protected List<StandardCard> build() {
        List<StandardCard> cards = new ArrayList<>();
        // code omitted for brevity
        return cards;
    }
}

```

```
    }  
}
```

Code 6.81

```
public class Deck<T extends Card> implements Iterable<T> {  
  
    private final List<T> cards;  
  
    public Deck(Pack pack) {  
  
        this.cards = pack.getCards();  
    }  
  
    public void shuffle() {...}  
  
    public List<T> dealHand(int numberOfCards) {...}  
  
    public T dealCard() {...}  
  
    public int remainingCards() {...}  
  
    public void removeCards(List<T> cards) {...}  
  
    @Override  
  
    public Iterator<T> iterator() {...}  
}
```

Code 6.82

```
// create a single classical card  
  
Card sevenHeart = new StandardCard(StandardSuit.HEARTS, 7);  
  
// create a complete deck of standards cards  
  
Pack cp = new StandardPack();  
  
Deck deck = new Deck(cp);  
  
System.out.println("Remaining cards: "  
    + deck.remainingCards());
```

Code 6.83

```
public enum VehicleType {  
  
    CAR(1), VAN(2), TRUCK(5);  
}  
  
public class Vehicle {  
  
    private final String licensePlate;
```

```

private final int spotsNeeded;
private final VehicleType type;
public Vehicle(String licensePlate,
               int spotsNeeded, VehicleType type) {
    this.licensePlate = licensePlate;
    this.spotsNeeded = spotsNeeded;
    this.type = type;
}
// getters omitted for brevity
// equals() and hashCode() omitted for brevity
}

```

Code 6.84

```

public class ParkingLot {
    private String name;
    private Map<String, ParkingFloor> floors;
    public ParkingLot(String name) {
        this.name = name;
    }
    public ParkingLot(String name,
                      Map<String, ParkingFloor> floors) {
        this.name = name;
        this.floors = floors;
    }
    // delegate to the proper ParkingFloor
    public ParkingTicket parkVehicle(Vehicle vehicle) {...}
    // we have to find the vehicle by looping floors
    public boolean unparkVehicle(Vehicle vehicle) {...}
    // we have the ticket, so we have the needed information
    public boolean unparkVehicle
        ParkingTicket parkingTicket) {...}
}

```

```
public boolean isFull() {...}  
protected boolean isFull(VehicleType type) {...}  
// getters and setters omitted for brevity  
}
```

Code 6.85

```
public class ParkingFloor {  
    private final String name;  
    private final int totalSpots;  
    private final Map<String, ParkingSpot>  
        parkingSpots = new LinkedHashMap<>();  
    // here, I use a Set, but you may want to hold the parking  
    // tickets in a certain order to optimize search  
    private final Set<ParkingTicket>  
        parkingTickets = new HashSet<>();  
    private int totalFreeSpots;  
    public ParkingFloor(String name, int totalSpots) {  
        this.name = name;  
        this.totalSpots = totalSpots;  
        initialize(); // create the parking spots  
    }  
    protected ParkingTicket parkVehicle(Vehicle vehicle) {...}  
    //we have to find the vehicle by looping the parking spots  
    protected boolean unparkVehicle(Vehicle vehicle) {...}  
    // we have the ticket, so we have the needed information  
    protected boolean unparkVehicle(  
        ParkingTicket parkingTicket) {...}  
    protected boolean isFull(VehicleType type) {...}  
    protected int countFreeSpots(  
        VehicleType vehicleType) {...}  
    // getters omitted for brevity
```

```

private List<ParkingSpot> findSpotsToFitVehicle(
    Vehicle vehicle) {...}

private void assignVehicleToParkingSpots(
    List<ParkingSpot> spots, Vehicle vehicle) {...}

private ParkingTicket releaseParkingTicket(
    Vehicle vehicle) {...}

private ParkingTicket findParkingTicket(
    Vehicle vehicle) {...}

private void registerParkingTicket(
    ParkingTicket parkingTicket) {...}

private boolean unregisterParkingTicket(
    ParkingTicket parkingTicket) {...}

private void initialize() {...}

}

```

Code 6.86

```

public class ParkingSpot {

    private boolean free = true;

    private Vehicle vehicle;

    private final String label;

    private final ParkingFloor parkingFloor;

    protected ParkingSpot(ParkingFloor parkingFloor,
        String label) {

        this.parkingFloor = parkingFloor;
        this.label = label;
    }

    protected boolean assignVehicle(Vehicle vehicle) {...}

    protected boolean removeVehicle() {...}

    // getters omitted for brevity

}

```

Code 6.87

```
public class ParkingSystem implements Parking {  
  
    private final String id;  
    private final ParkingLot parkingLot;  
    public ParkingSystem(String id, ParkingLot parkingLot) {  
        this.id = id;  
        this.parkingLot = parkingLot;  
    }  
    @Override  
    public ParkingTicket parkVehicleBtn(  
        String licensePlate, VehicleType type) {...}  
    @Override  
    public boolean unparkVehicleBtn(  
        String licensePlate, VehicleType type) {...}  
    @Override  
    public boolean unparkVehicleBtn(  
        ParkingTicket parkingTicket) {...}  
    // getters omitted for brevity  
}
```

Code 6.88

```
public class Reader {  
    private String name;  
    private String email;  
    // constructor omitted for brevity  
    // getters, equals() and hashCode() omitted for brevity  
}  
  
public class Book {  
    private final String author;  
    private final String title;  
    private final String isbn;
```

```
// constructor omitted for brevity  
  
public String fetchPage(int pageNr) {...}  
  
// getters, equals() and hashCode() omitted for brevity  
}
```

Code 6.89

```
public class Library {  
  
    private final Map<String, Book> books = new HashMap<>();  
  
    protected void addBook(Book book) {  
  
        books.putIfAbsent(book.getIsbn(), book);  
    }  
  
    protected boolean remove(Book book) {  
  
        return books.remove(book.getIsbn(), book);  
    }  
  
    protected Book find(String isbn) {  
  
        return books.get(isbn);  
    }  
}
```

Code 6.90

```
public class Displayer {  
  
    private Book book;  
  
    private Reader reader;  
  
    private String page;  
  
    private int pageNumber;  
  
    protected void displayReader(Reader reader) {  
  
        this.reader = reader;  
  
        refreshReader();  
    }  
  
    protected void displayBook(Book book) {  
  
        this.book = book;  
  
        refreshBook();  
    }
```

```

    }

protected void nextPage() {
    page = book.fetchPage(++pageNumber);
    refreshPage();
}

protected void previousPage() {
    page = book.fetchPage(--pageNumber);
    refreshPage();
}

private void refreshReader() {...}
private void refreshBook() {...}
private void refreshPage() {...}

}

```

Code 6.91

```

public class OnlineReaderSystem {

    private final Displayer displayer;
    private final Library library;
    private final ReaderManager readerManager;
    private Reader reader;
    private Book book;

    public OnlineReaderSystem() {
        displayer = new Displayer();
        library = new Library();
        readerManager = new ReaderManager();
    }

    public void displayReader(Reader reader) {
        this.reader = reader;
        displayer.displayReader(reader);
    }

    public void displayReader(String email) {

```

```
        this.reader = readerManager.find(email);

        if (this.reader != null) {
            displayer.displayReader(reader);
        }
    }

    public void displayBook(Book book) {
        this.book = book;
        displayer.displayBook(book);
    }

    public void displayBook(String isbn) {
        this.book = library.find(isbn);
        if (this.book != null) {
            displayer.displayBook(book);
        }
    }

    public void nextPage() {
        displayer.nextPage();
    }

    public void previousPage() {
        displayer.previousPage();
    }

    public void addBook(Book book) {
        library.addBook(book);
    }

    public boolean deleteBook(Book book) {
        if (!book.equals(this.book)) {
            return library.remove(book);
        }
        return false;
    }
}
```

```
public void addReader(Reader reader) {
    readerManager.addReader(reader);
}

public boolean deleteReader(Reader reader) {
    if (!reader.equals(this.reader)) {
        return readerManager.remove(reader);
    }
    return false;
}

public Reader getReader() {
    return reader;
}

public Book getBook() {
    return book;
}

}
```

Code 6.92

```
public class HashTable<K, V> {

    private static final int SIZE = 10;

    private static class HashEntry<K, V> {
        K key;
        V value;
        HashEntry <K, V> next;

        HashEntry(K k, V v) {
            this.key = k;
            this.value = v;
            this.next = null;
        }
    }
    ...
}
```

Code 6.93

```
private final HashEntry[] entries  
= new HashEntry[SIZE];  
...
```

Code 6.94

```
public void put(K key, V value) {  
  
    int hash = getHash(key);  
  
    final HashEntry hashEntry = new HashEntry(key, value);  
  
    if (entries[hash] == null) {  
  
        entries[hash] = hashEntry;  
  
    } else { // collision => chaining  
  
        HashEntry currentEntry = entries[hash];  
  
        while (currentEntry.next != null) {  
  
            currentEntry = currentEntry.next;  
  
        }  
  
        currentEntry.next = hashEntry;  
  
    }  
  
}  
  
public V get(K key) {  
  
    int hash = getHash(key);  
  
    if (entries[hash] != null) {  
  
        HashEntry currentEntry = entries[hash];  
  
        // Loop the entry linked list for matching  
        // the given 'key'  
  
        while (currentEntry != null) {  
  
            if (currentEntry.key.equals(key)) {  
  
                return (V) currentEntry.value;  
  
            }  
  
            currentEntry = currentEntry.next;  
  
        }  
  
    }  
}
```

```

    }

    return null;
}

```

Code 6.95

```

private int getHash(K key) {
    return Math.abs(key.hashCode() % SIZE);
}

```

Chapter 7

Images

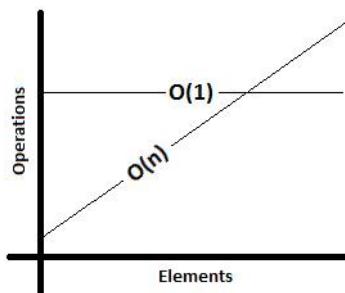


Figure 7.1 – The asymptotic runtime (Big O time)

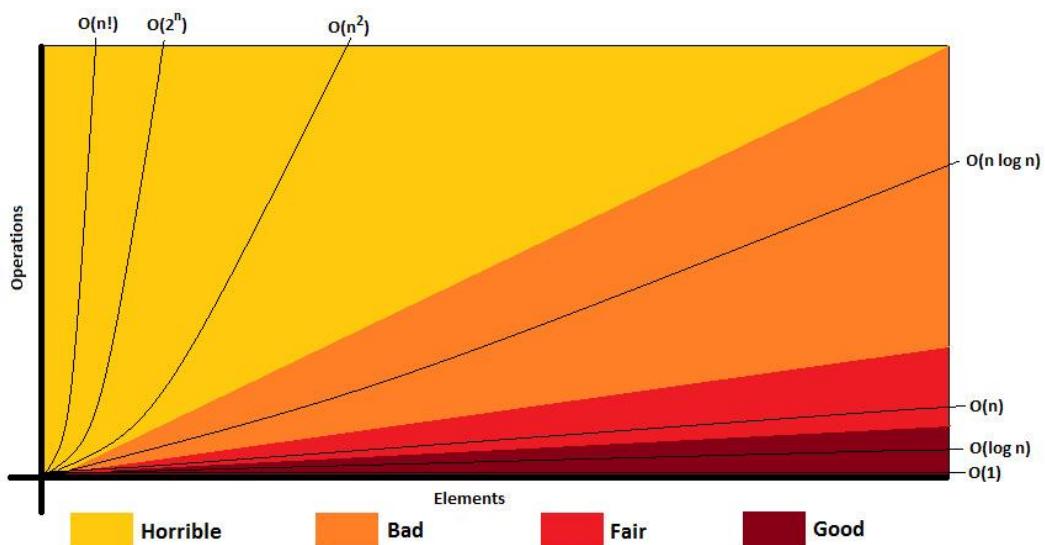


Figure 7.2 – Big O complexity chart

```

// snippet 1                                // snippet 2

int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int i = 0; i < a.length; i++) {
    if (a[i] < min) {
        min = a[i];
    }
    if (a[i] > max) {
        max = a[i];
    }
}

int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int i = 0; i < a.length; i++) {
    if (a[i] < min) {
        min = a[i];
    }
}
for (int i = 0; i < a.length; i++) {
    if (a[i] > max) {
        max = a[i];
    }
}

```

7.3 – Code Comparison

```

for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}

for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a.length; j++) {
        System.out.println(a[i] + a[j]);
    }
}

```

$O(n)$

$O(n^2)$

7.4 – Code snippet executed in $O(n)$

```

// snippet 1                                // snippet 2

for (int i=0; i<a.length; i++) {
}

for (int i=0; i<a.length; i++) {
}

```

$O(n)$

$O(n)$

7.5 – Code snippets 1 and 2

```
// snippet 1 // snippet 2
for (int i=0;i<a.length;i++) {
    System.out.println(a[i]);
}
for (int j=0;j<b.length;j++) {
    System.out.println(b[j]);
}

for (int i=0;i<a.length;i++) {
    for (int j=0;j<b.length;j++) {
        System.out.println(a[i]+b[j]);
    }
}
```

7.6 – Code snippet a and b

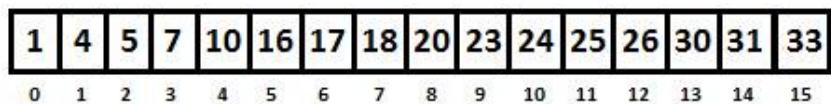


Figure 7.7 – Ordered array of 16 elements

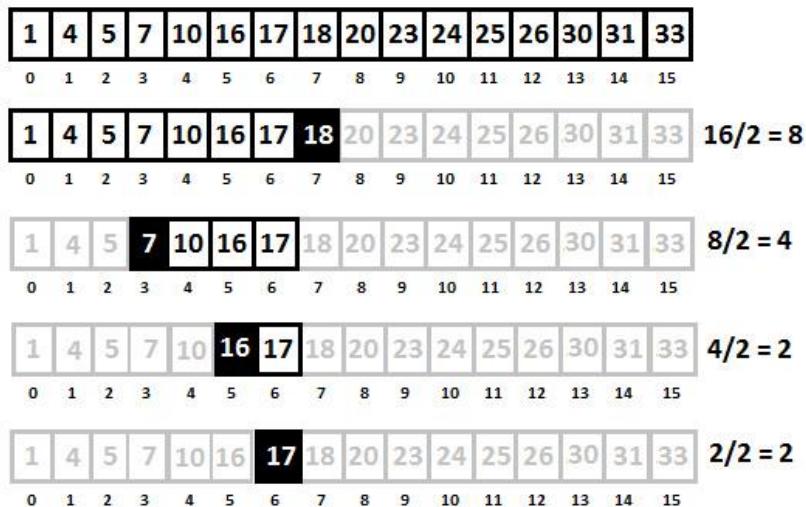


Figure 7.8 – The binary search algorithm

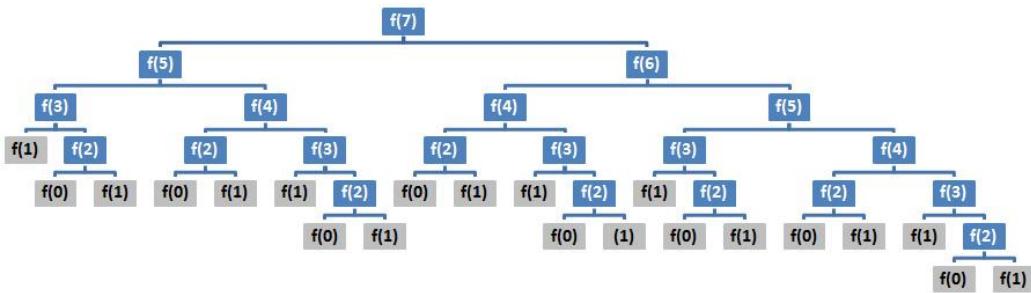


Figure 7.9 – Tree of calls

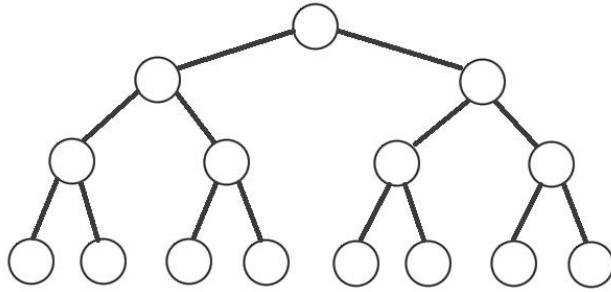


Figure 7.10 – Height-balanced binary search tree

$$2^{\log n} = X \equiv \log_2 X = \log n \equiv X = n \equiv O(X) = O(n)$$

Figure 7.11 – Big O expression

```
// snippet 1                                // snippet 2
for (int i=0;i<a.length;i++){           for (int i=0;i<a.length;i++){
    for (int j=0;j<a.length;j++){       for (int j=i+1;j<a.length;j++){
        System.out.println(a[i]+a[j]);   System.out.println(a[i]+a[j]);
    }
}
}
```

7.12 – Code snippets for Big O

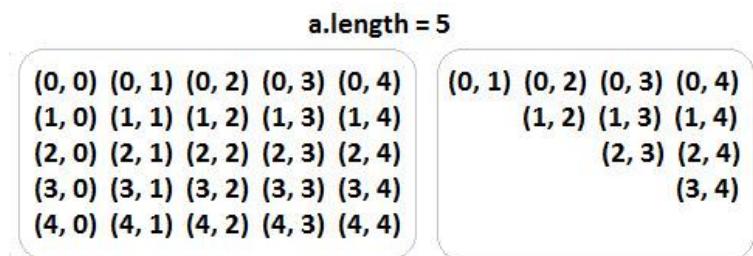


Figure 7.13 – Visualizing the runtime

$$2^4 = 16 \equiv \log_2 16 = 4$$

Figure 7.14 – Loop with O (log n)

```

int multiply(int x, int y) {      int powerxy(int x, int y) {
    int result = 1;                  if (y < 0) {
    for (int i=1; i<=y; i++) {        return 0;
        result *= x;                } else if (y == 0) {
    }                                return 1;
        return result;                } else {
    }                                return x*powerxy(x, y-1);
}
}

```

7.15 – Code snippets

$$10^d = n \equiv \log_2 n = d \equiv O(\log n)$$

Figure 7.16 – Digits relationship

Formula

$$16 * \frac{1}{2} = 8; 8 * \frac{1}{2} = 4; 4 * \frac{1}{2} = 2; 2 * \frac{1}{2} = 1$$

Formula 7.1

$$16 * \left(\frac{1}{2}\right)^4 = 1$$

Formula 7.2

$$n * \left(\frac{1}{2}\right)^k = 1 \equiv n * \frac{1}{2^k} = 1 \equiv 2^k * \frac{n}{2^k} = 2^k \equiv 2^k = n$$

Formula 7.3

$$2^k = n \rightarrow \log_2 n = k$$

Formula 7.4

$i = 0 \rightarrow \text{fibonacci}(0) \rightarrow 2^0 \text{ steps}$ $i = 1 \rightarrow \text{fibonacci}(1) \rightarrow 2^1 \text{ steps}$ $i = 2 \rightarrow \text{fibonacci}(2) \rightarrow 2^2 \text{ steps}$ \dots $i = k-1 \rightarrow \text{fibonacci}(k) \rightarrow 2^{k-1} \text{ steps}$	}	$= 2^0 + 2^1 + 2^2 + \dots + 2^{k-1} \text{ steps}$
---	---	---

Formula 7.5

$$n * n = n^2 \equiv O(n^2)$$

Formula 7.6

$$\frac{n*n}{2} = \frac{n^2}{2} = n^2 * \frac{1}{2} \equiv O(n^2)$$

Formula 7.7

Code

Code 7.1

```
// snippet 1
return 23;
```

code 7.2

```
// snippet 2 - 'cars' is an array
int thirdCar = cars[3];
```

code 7.3

```
// snippet 3 - 'cars' is a 'java.util.Queue'
Car car = cars.peek();
```

Code 7.4

```
// snippet 1 - 'a' is an array
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

Code 7.5

```
// snippet 1 - 'a' is an array
for (int i = 0; i < a.length; i++) {
```

```
        System.out.println("Current element:");
        System.out.println(a[i]);
        System.out.println("Current element + 1:");
        System.out.println(a[i] + 1);
    }
```

Code 7.6

```
search 17 in {1, 4, 5, 7, 10, 16, 17, 18, 20,
              23, 24, 25, 26, 30, 31, 33}
compare 17 to 18 -> 17 < 18
search 17 in {1, 4, 5, 7, 10, 16, 17, 18}
compare 17 to 7 -> 17 > 7
search 17 in {7, 10, 16, 17}
compare 17 to 16 -> 17 > 16
search 17 in {16, 17}
compare 17 to 17
return
```

Code 7.7

```
int fibonacci(int k) {
    if (k <= 1) {
        return k;
    }
    return fibonacci(k - 2) + fibonacci(k - 1);
}
```

Code 7.8

```
void printInOrder(Node node) {
    if (node != null) {
        printInOrder(node.left);
        System.out.print(" " + node.element);
        printInOrder(node.right);
    }
}
```

```
}
```

Code 7.9

```
void printFibonacci(int k) {  
    for (int i = 0; i < k; i++) {  
        System.out.println(i + ":" + fibonacci(i));  
    }  
}  
  
int fibonacci(int k) {  
    if (k <= 1) {  
        return k;  
    }  
    return fibonacci(k - 2) + fibonacci(k - 1);  
}
```

Code 7.10

```
void printFibonacci(int k) {  
    int[] cache = new int[k];  
    for (int i = 0; i < k; i++) {  
        System.out.println(i + ":" + fibonacci(i, cache));  
    }  
}  
  
int fibonacci(int k, int[] cache) {  
    if (k <= 1) {  
        return k;  
    } else if (cache[k] > 0) {  
        return cache[k];  
    }  
    cache[k] = fibonacci(k - 2, cache)  
        + fibonacci(k - 1, cache);  
    return cache[k];  
}
```

Code 7.11

```
Calling fibonacci(0):  
Result of fibonacci(0) is 0  
Calling fibonacci(1):  
Result of fibonacci(1) is 1  
Calling fibonacci(2):  
fibonacci(0)  
fibonacci(1)  
fibonacci(2) is computed and cached at cache[2]  
Result of fibonacci(2) is 1  
Calling fibonacci(3):  
fibonacci(1)  
fibonacci(2) is fetched from cache[2] as: 1  
fibonacci(3) is computed and cached at cache[3]  
Result of fibonacci(3) is 2  
Calling fibonacci(4):  
fibonacci(2) is fetched from cache[2] as: 1  
fibonacci(3) is fetched from cache[3] as: 2  
fibonacci(4) is computed and cached at cache[4]  
Result of fibonacci(4) is 3  
Calling fibonacci(5):  
fibonacci(3) is fetched from cache[3] as: 2  
fibonacci(4) is fetched from cache[4] as: 3  
fibonacci(5) is computed and cached at cache[5]  
Result of fibonacci(5) is 5  
Calling fibonacci(6):  
fibonacci(4) is fetched from cache[4] as: 3  
fibonacci(5) is fetched from cache[5] as: 5  
fibonacci(6) is computed and cached at cache[6]  
Result of fibonacci(6) is 8
```

Code 7.12

```
for (int i = 0; i < a.length; i++) {  
    for (int j = 0; j < a.length; j++) {  
        for (int q = 0; q < 1_000_000; q++) {  
            System.out.println(a[i] + a[j]);  
        }  
    }  
}
```

Code 7.13

```
for (int i = 0; i < a.length; i++) {  
    for (int j = 0; j < a.length; j++) {  
        // O(1)  
    }  
}
```

Code 7.14

```
for (int i = 0; i < a.length / 2; i++) {  
    System.out.println(a[i]);  
}
```

Code 7.15

```
for (int i = 0; i < a.length; i++) {  
    for (int j = a.length; j > 0; j /= 2) {  
        System.out.println(a[i] + ", " + j);  
    }  
}
```

Code 7.16

```
String[] sortArrayOfString(String[] a) {  
    for (int i = 0; i < a.length; i++) {  
        // sort each string via O(n log n) algorithm  
    }  
    // sort the array itself via O(n log n) algorithm
```

```
    return a;  
}
```

Code 7.17

```
long factorial(int num) {  
    if (num >= 1) {  
        return num * factorial(num - 1);  
    } else {  
        return 1;  
    }  
}
```

Code 7.18

```
int div(int x, int y) {  
    int count = 0;  
    int sum = y;  
    while (sum <= x) {  
        sum += y;  
        count++;  
    }  
    return count;  
}
```

Code 7.19

```
int sqrt(int n) {  
    for (int guess = 1; guess * guess <= n; guess++) {  
        if (guess * guess == n) {  
            return guess;  
        }  
    }  
    return -1;  
}
```

Code 7.20

```

int sumDigits(int n) {
    int result = 0;
    while (n > 0) {
        result += n % 10;
        n /= 10;
    }
    return result;
}

```

Code 7.21

```

boolean matching(int[] x, int[] y) {
    mergesort(y);
    for (int i : x) {
        if (binarySearch(y, i) >= 0) {
            return true;
        }
    }
    return false;
}

```

Chapter 8

Images

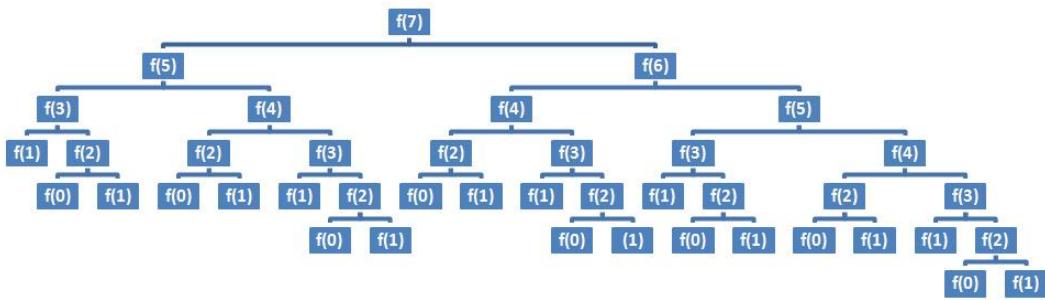


Figure 8.1 – Tree of calls (plain recursion)

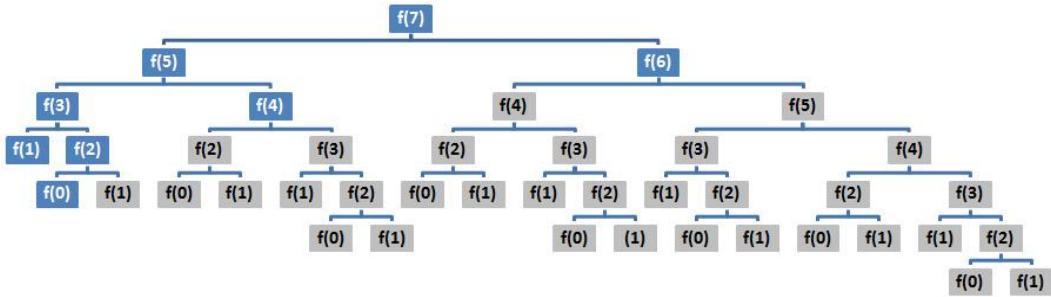


Figure 8.2 – Tree of calls (duplicate work)

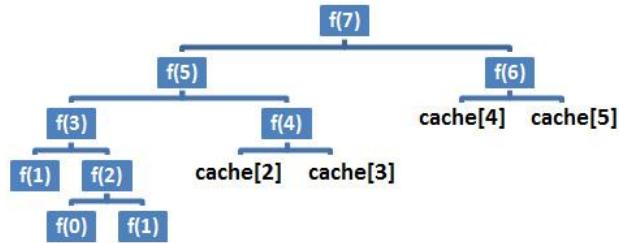


Figure 8.3 – Tree of calls (Memoization)

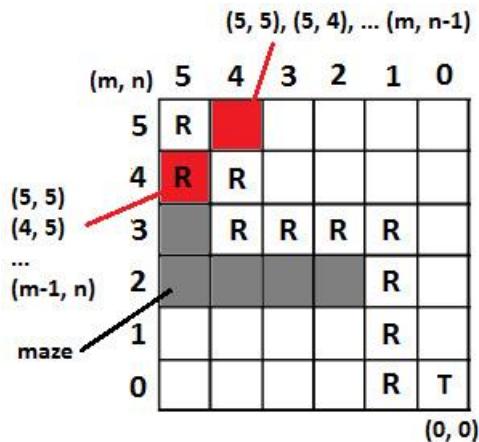


Figure 8.4 – Determining the moving pattern

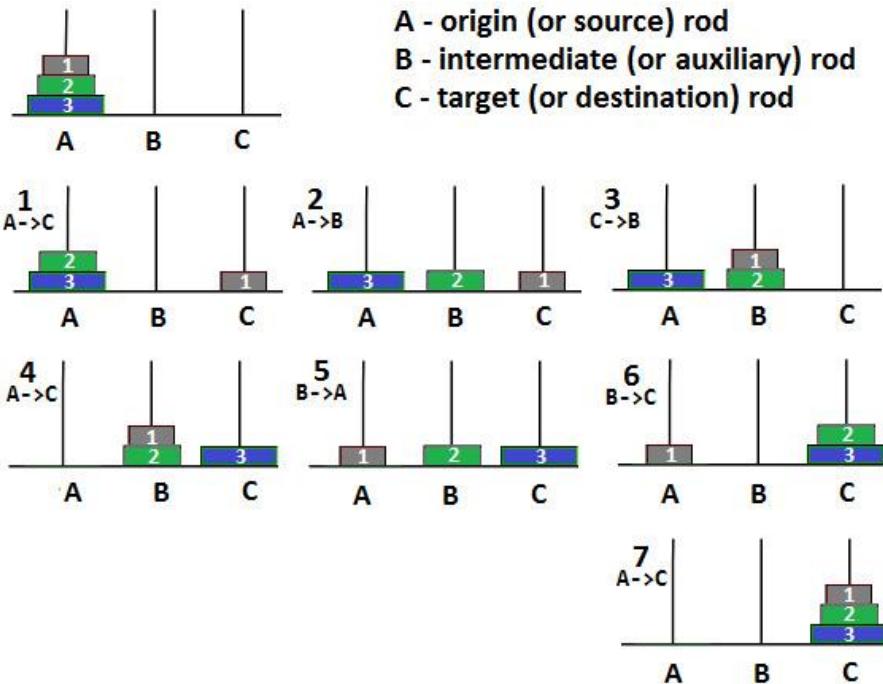


Figure 8.5 – Tower of Hanoi (three disks)

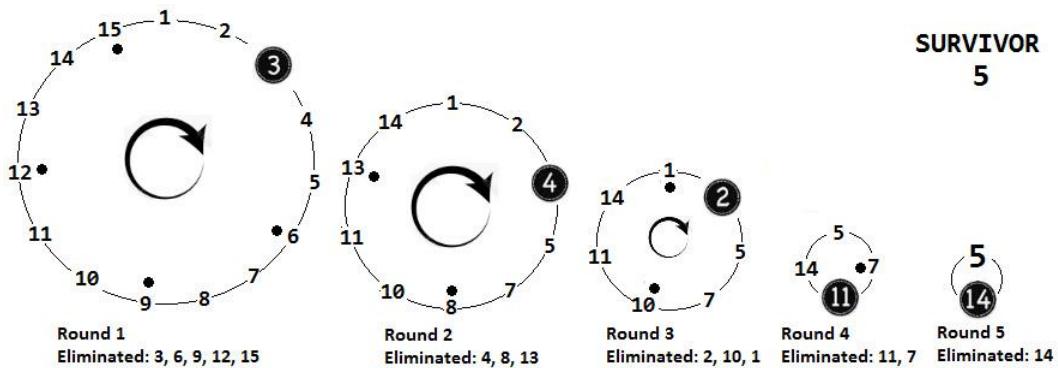


Figure 8.6 – Josephus for $n=15$ and $k=3$

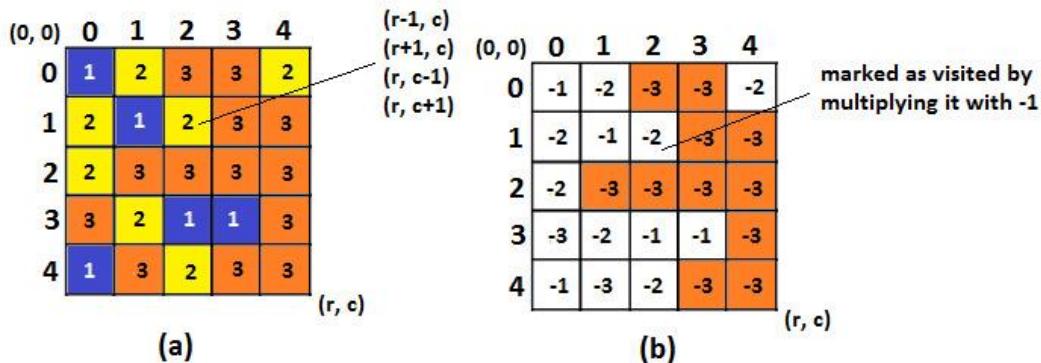


Figure 8.7 – Biggest color spot ((a) – initial grid, (b) – solved grid)

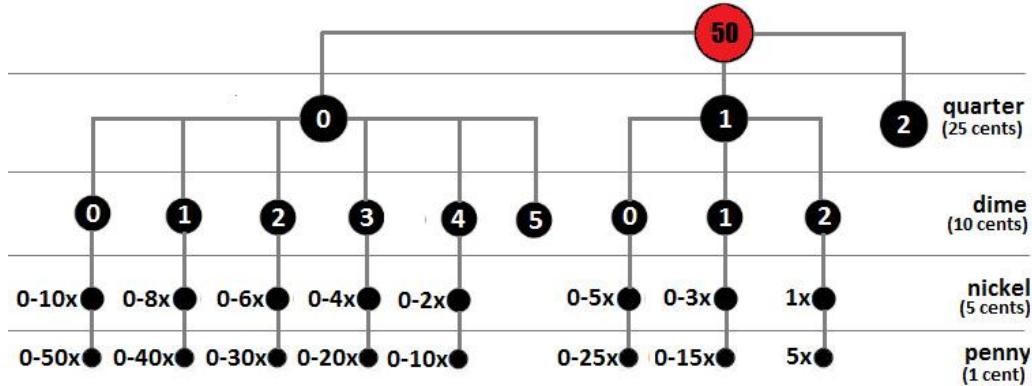


Figure 8.8 – Changing n cents into quarters, dimes, nickels, and pennies

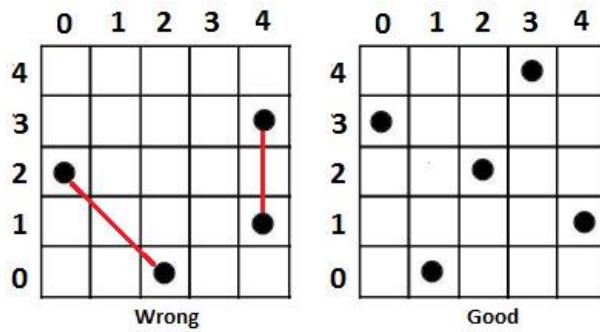


Figure 8.9(a) – Failure and solution

- Try to build the first tower at (0, 0), (0,1), (0, 2), (0, 3) or (0,4)
- The first tower can be build at (0, 0), (0,1), (0, 2), (0, 3) or (0,4)
 - The first tower was built at (0,2)

Figure 8.9(b): Part 1 of the logic to build the towers

First row:

- Try to build the first tower at (0, 0), (0,1), (0, 2), (0, 3) or (0,4)
- The first tower can be build at (0, 0), (0,1), (0, 2), (0, 3) or (0,4)
 - The first tower was built at (0, 2)

Second row:

- Try to build the second tower at (1, 0), (1,1), (1, 2), (1, 3) or (1,4)
- The second tower can be build at (1, 3) or (1, 4)
- The second tower was built at (1, 4)

Third row:

- Try to build the third tower at (2, 0), (2,1), (2, 2), (2, 3) or (2,4)
- The third tower can be build at (2, 0) or (2, 2)
- The third tower was built at (2, 2)

...

Figure 8.9(c): Part 2 of the logic to build the towers

if we half the array,
the magic index must be on right side

element																		
	-5	-4	-2	0	1	2	3	5	6	7	9	11	13	20	22	24	25	27
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Ordered array, no duplicates

if we half the array,
the magic index can be on both sides

element																		
	1	4	4	4	5	5	6	6	6	11	12	12	12	15	17	20	21	21
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Ordered array with duplicates

Figure 8.10 – Sorted array of 18 elements

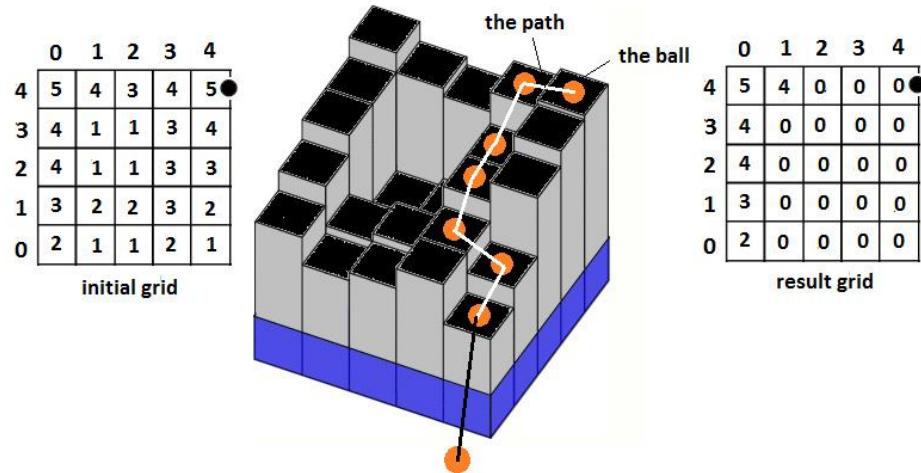


Figure 8.11 – The falling ball

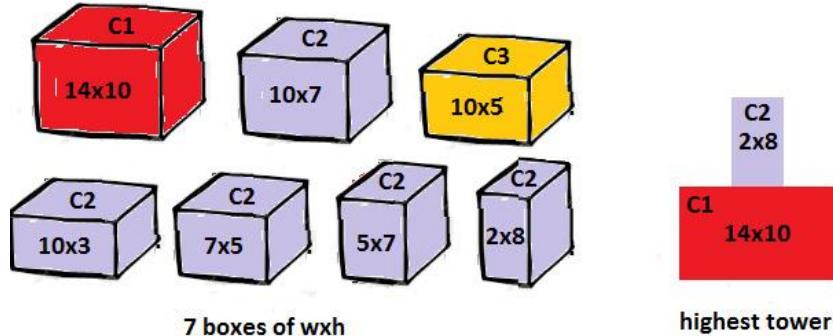


Figure 8.12(a) – The highest colored tower

- Choose the 14x10 box as the base box
 - On top of 14x10 box we can add the 10x7 box
 - On top of 10x7 box we cannot add another box; the total high is 17
 - On top of 14x10 box we can add the 10x5 box
 - On top of 10x5 we cannot add another box; the total high is 15, therefore we can ignore it since it is less than 17
 - ...
 - On top of 14x10 box we can add the 2x8
 - On top of 2x8 we cannot add another box; the total high is 18, therefore we update the high to 18, and this is the final result

Figure 8.12(b) The logic to select the boxes to build the highest tower

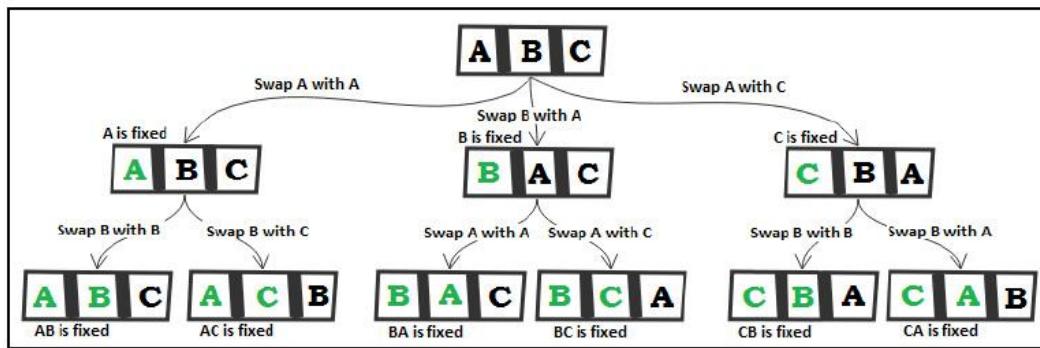


Figure 8.13 – Permuting ABC

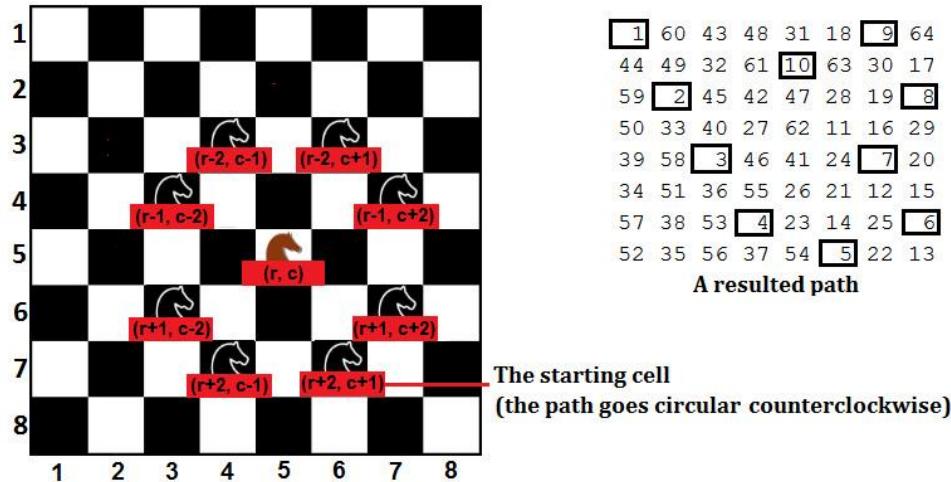


Figure 8.14 – Knight tour

For $n=2$, we have
 $\{\} \{\}$ $\{ \{\} \}$
 For $n=3$, we have
 $\{\} \{\} \rightarrow \{\} \{\} \{\}$ $\{ \{\} \} \{\}$ $\{\} \{ \{\} \}$
 $\{ \{\} \} \rightarrow \{\} \{ \{\} \}$ $\{ \{\} \} \{\}$ $\{\} \{ \{\} \}$

duplicate pair

Figure 8.15 – Curly braces duplicate pairs

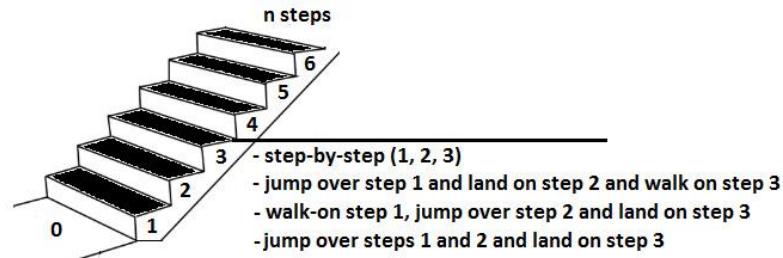


Figure 8.16 – Staircase (how to reach step 3)

0	1	2	3	4	5	6	7	8
3	2	7	4	5	1	6	7	9

Sum: 7 Subset: 2 4 1

Figure 8.17 – Subset of sum 7

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	5	T								
2	1	T								
3	6	T								
4	10	T								
5	7	T								
6	11	T								
7	2	T								

Figure 8.18 – Initial matrix

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	5	T	F	F	F	F	F	F	F	F
2	1	T								
3	6	T								
4	10	T								
5	7	T								
6	11	T								
7	2	T								

Figure 8.19 – Filling up the second row

	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
1	T	F	F	F	F	T	F	F	F	F
2	T									
3	T									
4	T									
5	T									
6	T									
7	T									
8	T									
9										

Figure 8.20 – Applying the algorithm to the second row

	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
1	T	F	F	F	F	T	F	F	F	F
2	T	T	F	F	F	T	T	F	F	F
3	T	T	F	F	F	T	T	T	F	F
4	T	T	F	F	F	T	T	T	F	F
5	T	T	F	F	F	T	T	T	T	F
6	T	T	F	F	F	T	T	T	T	F
7	T	T	F	F	F	T	T	T	T	T
8	T	T	T	T	F	T	T	T	T	T
9										

Figure 8.21 – Complete matrix

	0	1	2	3	4	5	6	7	8	9
0	T	F	F	F	F	F	F	F	F	F
1	T	F	F	F	F	T	F	F	F	F
2	T	T	F	F	F	T	T	F	F	F
3	T	T	F	F	F	T	T	T	F	F
4	T	T	F	F	F	T	T	T	F	F
5	T	T	F	F	F	T	T	T	T	F
6	T	T	F	F	F	T	T	T	T	F
7	T	T	F	F	F	T	T	T	T	T
8	T	T	T	T	F	T	T	T	T	T
9										

$9 = 2 + 6 + 1$, so a subset is {2, 6, 1}

Figure 8.22 – Subset solution path

Code

Code 8.1

```
int fibonacci(int k) {
    // base case
```

```
if (k <= 1) {  
    return k;  
}  
  
// recursive call  
return fibonacci(k - 2) + fibonacci(k - 1);  
}
```

Code 8.2

```
int fibonacci(int k) {  
    if (k <= 1) {  
        return k;  
    }  
  
    return fibonacci(k - 2) + fibonacci(k - 1);  
}
```

Code 8.3

```
int fibonacci(int k) {  
    return fibonacci(k, new int[k + 1]);  
}  
  
int fibonacci(int k, int[] cache) {  
    if (k <= 1) {  
        return k;  
    } else if (cache[k] > 0) {  
        return cache[k];  
    }  
  
    cache[k] = fibonacci(k - 2, cache)  
        + fibonacci(k - 1, cache);  
  
    return cache[k];  
}
```

Code 8.4

```
int fibonacci(int k) {  
    if (k <= 1) {
```

```

        return k;
    }

    int first = 1;
    int second = 0;
    int result = 0;
    for (int i = 1; i < k; i++) {
        result = first + second;
        second = first;
        first = result;
    }
    return result;
}

```

Code 8.5

```

public static boolean computePath(int m, int n,
    boolean[][] maze, Set<Point> path) {
    // we fell off the grid so we return
    if (m < 0 || n < 0) {
        return false;
    }
    // we cannot step at this cell
    if (maze[m][n]) {
        return false;
    }
    // we reached the target
    // (this is the bottom-right corner)
    if (((m == 0) && (n == 0)))
        // or, try to go to the right
        || computePath(m, n - 1, maze, path)
        // or, try to go to down
        || computePath(m - 1, n, maze, path));
}

```

```

        // we add the cell to the path
        path.add(new Point(m, n));
        return true;
    }
    return false;
}

```

Code 8.6

```

public static void moveDisks(int n, char origin,
    char target, char intermediate) {
    if (n <= 0) {
        return;
    }
    if (n == 1) {
        System.out.println("Move disk 1 from rod "
            + origin + " to rod " + target);
        return;
    }
    // move top n - 1 disks from origin to intermediate,
    // using target as a intermediate
    moveDisks(n - 1, origin, intermediate, target);
    System.out.println("Move disk " + n + " from rod "
        + origin + " to rod " + target);
    // move top n - 1 disks from intermediate to target,
    // using origin as an intermediate
    moveDisks(n - 1, intermediate, target, origin);
}

```

Code 8.7

```

public static int josephus(int n, int k) {
    if (n == 1) {
        return 1;
    }

```

```

    } else {
        return (josephus(n - 1, k) + k - 1) % n + 1;
    }
}

```

Code 8.8

```

public static void printJosephus(int n, int k) {
    Queue<Integer> circle = new ArrayDeque<>();
    for (int i = 1; i <= n; i++) {
        circle.add(i);
    }
    while (circle.size() != 1) {
        for (int i = 1; i <= k; i++) {
            int eliminated = circle.poll();
            if (i == k) {
                System.out.println("Eliminated: "
                    + eliminated);
                break;
            }
            circle.add(eliminated);
        }
    }
    System.out.println("Using queue! Survivor: "
        + circle.peek());
}

```

Code 8.9

```

public class BiggestColorSpot {
    private int currentColorSpot;
    void determineBiggestColorSpot(int cols,
        int rows, int a[][]) {
        ...
    }
}

```

```

}

private void computeColorSpot(int i, int j,
    int cols, int rows, int a[][], int color) {
    a[i][j] = -a[i][j];
    currentColorSpot++;
    if (i > 1 && a[i - 1][j] == color) {
        computeColorSpot(i - 1, j, cols,
            rows, a, color);
    }
    if ((i + 1) < rows && a[i + 1][j] == color) {
        computeColorSpot(i + 1, j, cols, rows, a, color);
    }
    if (j > 1 && a[i][j - 1] == color) {
        computeColorSpot(i, j - 1, cols,
            rows, a, color);
    }
    if ((j + 1) < cols && a[i][j + 1] == color) {
        computeColorSpot(i, j + 1, cols,
            rows, a, color);
    }
}
}
}

```

Code 8.10

```

void determineBiggestColorSpot(int cols,
    int rows, int a[][]) {
    int biggestColorSpot = 0;
    int color = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (a[i][j] > 0) {

```

```

       currentColorSpot = 0;

       computeColorSpot(i, j, cols,
                        rows, a, a[i][j]);

        if (currentColorSpot > biggestColorSpot) {

            biggestColorSpot = currentColorSpot;
            color = a[i][j] * (-1);

        }

    }

}

System.out.println("\nColor: " + color
+ " Biggest spot: " + biggestColorSpot);
}

```

Code 8.11

```

public static int calculateChangeMemoization(int n) {

    int[] coins = {25, 10, 5, 1};

    int[][] cache = new int[n + 1][coins.length];

    return calculateChangeMemoization(n, coins, 0, cache);
}

private static int calculateChangeMemoization(int amount,
                                              int[] coins, int position, int[][] cache) {

    if (cache[amount][position] > 0) {

        return cache[amount][position];
    }

    if (position >= coins.length - 1) {

        return 1;
    }

    int coin = coins[position];
    int count = 0;

    for (int i = 0; i * coin <= amount; i++) {

```

```

        int remaining = amount - i * coin;
        count += calculateChangeMemoization(remaining,
            coins, position + 1, cache);
    }
    cache[amount][position] = count;
    return count;
}

```

Code 8.12

```

protected static final int GRID_SIZE = 5; // (5x5)

public static void buildTowers(int row, Integer[] columns,
    Set<Integer[]> solutions) {
    if (row == GRID_SIZE) {
        solutions.add(columns.clone());
    } else {
        for (int col = 0; col < GRID_SIZE; col++) {
            if (canBuild(columns, row, col)) {
                // build this tower
                columns[row] = col;
                // go to the next row
                buildTowers(row + 1, columns, solutions);
            }
        }
    }
}

private static boolean canBuild(Integer[] columns,
    int nextRow, int nextColumn) {
    for (int currentRow=0; currentRow<nextRow;
        currentRow++) {
        int currentColumn = columns[currentRow];
        // cannot build on the same column
    }
}

```

```

        if (currentColumn == nextColumn) {
            return false;
        }

        int columnsDistance
            = Math.abs(currentColumn - nextColumn);

        int rowsDistance = nextRow - currentRow;

        // cannot build on the same diagonal
        if (columnsDistance == rowsDistance) {
            return false;
        }
    }

    return true;
}

```

Code 8.13

```

public static int find(int[] arr) {
    return find(arr, 0, arr.length - 1);
}

private static int find(int[] arr,
    int startIndex, int endIndex) {
    if (startIndex > endIndex) {
        return -1; // return an invalid index
    }

    // halved the indexes
    int middleIndex = (startIndex + endIndex) / 2;
    // value (element) of middle index
    int value = arr[middleIndex];
    // check if this is a magic index
    if (value == middleIndex) {
        return middleIndex;
    }
}

```

```

// search from middle of the array to the left
int leftIndex = find(arr, startIndex,
    Math.min(middleIndex - 1, value));
if (leftIndex >= 0) {
    return leftIndex;
}
// search from middle of the array to the right
return find(arr, Math.max(middleIndex + 1,
    value), endIndex);
}
}

```

Code 8.14

```

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        System.out.format("%2s", elevations[i][j]);
    }
    System.out.println();
}

```

Code 8.15

```

public static void computePath(
    int prevElevation, int i, int j,
    int rows, int cols, int[][][] elevations) {
    // ensure the ball is still on the grid
    if (i >= 0 && i <= (rows-1) && j >= 0 && j <= (cols-1)) {
        int currentElevation = elevations[i][j];
        // check if the ball can fall
        if (prevElevation >= currentElevation
            && currentElevation > 0) {
            // store the current elevation
            prevElevation = currentElevation;
        }
    }
}

```

```

        // mark this cell as visited
        elevations[i][j] = 0;

        // try to move the ball
        computePath(prevElevation, i, j-1,
                    rows, cols, elevations);
        computePath(prevElevation, i-1,
                    j, rows, cols, elevations);
        computePath(prevElevation, i, j+1,
                    rows, cols, elevations);
        computePath(prevElevation, i+1, j,
                    rows, cols, elevations);

    }

}

}

```

Code 8.16

```

// Memoization

public static int buildViaMemoization(List<Box> boxes) {
    // sorting boxes by width (you can do it by height as well)
    Collections.sort(boxes, new Comparator<Box>() {
        @Override
        public int compare(Box b1, Box b2) {
            return Integer.compare(b2.getWidth(),
                b1.getWidth());
        }
    });
    // place each box as the base (bottom box) and
    // try to arrange the rest of the boxes
    int highest = 0;
    int[] cache = new int[boxes.size()];
    for (int i = 0; i < boxes.size(); i++) {

```

```

        int height = buildMemoization(boxes, i, cache);

        highest = Math.max(highest, height);

    }

    return highest;
}

// Memoization

private static int buildMemoization(List<Box> boxes,
    int base, int[] cache) {

    if (base < boxes.size() && cache[base] > 0) {

        return cache[base];
    }

    Box current = boxes.get(base);

    int highest = 0;

    // since the boxes are sorted we don't
    // look in [0, base + 1)

    for (int i = base + 1; i < boxes.size(); i++) {

        if (boxes.get(i).canBeNext(current)) {

            int height = buildMemoization(boxes, i, cache);

            highest = Math.max(height, highest);
        }
    }

    highest = highest + current.getHeight();

    cache[base] = highest;
}

return highest;
}

```

Code 8.17

```

public static Set<String> permute(String str) {
    return permute("", str);
}

private static Set<String> permute(String prefix, String str) {

```

```

Set<String> permutations = new HashSet<>();

int n = str.length();
if (n == 0) {
    permutations.add(prefix);
} else {
    for (int i = 0; i < n; i++) {
        permutations.addAll(permute(prefix + str.charAt(i),
            str.substring(i + 1, n) + str.substring(0, i)));
    }
}
return permutations;
}

```

Code 8.18

```

private static Map<Character, Integer> charactersMap(
    String str) {
    Map<Character, Integer> characters = new HashMap<>();
    BiFunction<Character, Integer, Integer> count = (k, v)
        -> ((v == null) ? 1 : ++v);
    for (char c : str.toCharArray()) {
        characters.compute(c, count);
    }
    return characters;
}

```

Code 8.19

```

public static List<String> permute(String str) {
    return permute("", str.length(), charactersMap(str));
}

private static List<String> permute(String prefix,
    int strlength, Map<Character, Integer> characters) {
    List<String> permutations = new ArrayList<>();

```

```

if (strlength == 0) {
    permutations.add(prefix);
} else {
    // fetch next char and generate remaining permutations
    for (Character c : characters.keySet()) {
        int count = characters.get(c);
        if (count > 0) {
            characters.put(c, count - 1);
            permutations.addAll(permute(prefix + c,
                strlength - 1, characters));
            characters.put(c, count);
        }
    }
}
return permutations;
}

```

Code 8.20

```

public class KnightTour {
    private final int n;
    // constructor omitted for brevity
    // all 8 possible movements for a knight
    public static final int COL[]
        = {1,2,2,1,-1,-2,-2,-1,1};
    public static final int ROW[]
        = {2,1,-1,-2,-2,-1,1,2,2};
    public void knightTour(int r, int c,
        int cell, int visited[][])) {
        // mark current cell as visited
        visited[r][c] = cell;
        // we have a solution
    }
}

```

```

        if (cell >= n * n) {
            print(visited);
            // backtrack before returning
            visited[r][c] = 0;
            return;
        }

        // check for all possible movements (8)
        // and recur for each valid movement
        for (int i = 0; i < (ROW.length - 1); i++) {
            int newR = r + ROW[i];
            int newC = c + COL[i];
            // check if the new position is valid un-visited
            if (isValid(newR, newC)
                && visited[newR][newC] == 0) {
                knightTour(newR, newC, cell + 1, visited);
            }
        }

        // backtrack from current cell
        // and remove it from current path
        visited[r][c] = 0;
    }

    // check if (r, c) is valid chess board coordinates
    private boolean isValid(int r, int c) {
        return !(r < 0 || c < 0 || r >= n || c >= n);
    }

    // print the solution as a board
    private void print(int[][] visited) {
        ...
    }
}

```

```
}
```

Code 8.21

```
public static List<String> embrace(int nr) {  
    List<String> results = new ArrayList<>();  
    embrace(nr, nr, new char[nr * 2], 0, results);  
    return results;  
}  
  
private static void embrace(int leftHand, int rightHand,  
    char[] str, int index, List<String> results) {  
    if (rightHand < leftHand || leftHand < 0) {  
        return;  
    }  
    if (leftHand == 0 && rightHand == 0) {  
        // result found, so store it  
        results.add(String.valueOf(str));  
    } else {  
        // add left brace  
        str[index] = '{';  
        embrace(leftHand - 1, rightHand, str, index + 1,  
            results);  
        // add right brace  
        str[index] = '}';  
        embrace(leftHand, rightHand - 1, str, index + 1,  
            results);  
    }  
}
```

Code 8.22

```
public static int countViaMemoization(int n) {  
    int[] cache = new int[n + 1];  
    return count(n, cache);
```

```

}

private static int count(int n, int[] cache) {
    if (n == 0) {
        return 1;
    } else if (n < 0) {
        return 0;
    } else if (cache[n] > 0) {
        return cache[n];
    }
    cache[n] = count(n - 1, cache)
        + count(n - 2, cache) + count(n - 3, cache);
    return cache[n];
}

```

Code 8.23

```

/* Recursive approach */

public static void findSumRecursive(int[] arr, int index,
    int currentSum, int givenSum, int[] subset) {
    if (currentSum == givenSum) {
        System.out.print("\nSubset found: ");
        for (int i = 0; i < subset.length; i++) {
            if (subset[i] == 1) {
                System.out.print(arr[i] + " ");
            }
        }
    }
} else if (index != arr.length) {
    subset[index] = 1;
    currentSum += arr[index];
    findSumRecursive(arr, index + 1,
        currentSum, givenSum, subset);
    currentSum -= arr[index];
}

```

```

        subset[index] = 0;
        findSumRecursive(arr, index + 1,
                         currentSum, givenSum, subset);
    }
}

```

Code 8.24

```

/* Dynamic Programming (Bottom-Up) */
public static boolean findSumDP(int[] arr, int givenSum) {
    boolean[][] matrix
        = new boolean[arr.length + 1][givenSum + 1];
    // prepare the first row
    for (int i = 1; i <= givenSum; i++) {
        matrix[0][i] = false;
    }
    // prepare the first column
    for (int i = 0; i <= arr.length; i++) {
        matrix[i][0] = true;
    }
    for (int i = 1; i <= arr.length; i++) {
        for (int j = 1; j <= givenSum; j++) {
            // first, copy the data from the above row
            matrix[i][j] = matrix[i - 1][j];
            // if matrix[i][j] = false compute
            // if the value should be F or T
            if (matrix[i][j] == false && j >= arr[i - 1]) {
                matrix[i][j] = matrix[i][j]
                    || matrix[i - 1][j - arr[i - 1]];
            }
        }
    }
}

```

```

    printSubsetMatrix(arr, givenSum, matrix);
    printOneSubset(matrix, arr, arr.length, givenSum);
    return matrix[arr.length][givenSum];
}

```

Code 8.25

```

private static boolean breakItPlainRecursive(
    Set<String> dictionary, String str, int index) {
    if (index == str.length()) {
        return true;
    }
    boolean canBreak = false;
    for (int i = index; i < str.length(); i++) {
        canBreak = canBreak
            || dictionary.contains(str.substring(index, i + 1))
            && breakItPlainRecursive(dictionary, str, i + 1);
    }
    return canBreak;
}

```

Code 8.26

```

public static boolean breakItBottomUp(
    Set<String> dictionary, String str) {
    boolean[] table = new boolean[str.length() + 1];
    table[0] = true;
    for (int i = 0; i < str.length(); i++) {
        for (int j = i + 1; table[i] && j <= str.length(); j++) {
            if (dictionary.contains(str.substring(i, j))) {
                table[j] = true;
            }
        }
    }
}

```

```
    return table[str.length()];
}
```

Code 8.27

```
public class Trie {

    // characters 'a'-'z'

    private static final int CHAR_SIZE = 26;

    private final Node head;

    public Trie() {
        this.head = new Node();
    }

    // Trie node

    private static class Node {
        private boolean leaf;
        private final Node[] next;
        private Node() {
            this.leaf = false;
            this.next = new Node[CHAR_SIZE];
        }
    };

    // insert a string in Trie

    public void insertTrie(String str) {
        Node node = head;
        for (int i = 0; i < str.length(); i++) {
            if (node.next[str.charAt(i) - 'a'] == null) {
                node.next[str.charAt(i) - 'a'] = new Node();
            }
            node = node.next[str.charAt(i) - 'a'];
        }
        node.leaf = true;
    }
}
```

```

// Method to determine if the given string can be
// segmented into a space-separated sequence of one or
// more dictionary words

public boolean breakIt(String str) {
    // table[i] is true if the first i
    // characters of str can be segmented

    boolean[] table = new boolean[str.length() + 1];
    table[0] = true;

    for (int i = 0; i < str.length(); i++) {
        if (table[i]) {
            Node node = head;
            for (int j = i; j < str.length(); j++) {
                if (node == null) {
                    break;
                }
                node = node.next[str.charAt(j) - 'a'];
                // [0, i]: use our known decomposition
                // [i+1, j]: use this String in the Trie
                if (node != null && node.leaf) {
                    table[j + 1] = true;
                }
            }
        }
        // table[n] would be true if
        // all characters of str can be segmented
        return table[str.length()];
    }
}

```

Chapter 9

Images

00000000 00000000 00000000 00**110011**
... 2⁷ 2⁶ 2⁵ 2⁴ 2³ 2² 2¹ 2⁰
51

Figure 9.1 – Binary to decimal (32-bit integer)

X	Y	X&Y	X Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Figure 9.2 – Bitwise operators

X 0 _s = X	X ^ 0 _s = X	X & 0 _s = 0
X 1 _s = X	X ^ 1 _s = ~X	X & 1 _s = X
X X = X	X ^ X = 0	X & X = X

Figure 9.3 – Bitwise tips

23 << 3
00000000 00000000 00000000 00010111 = 23
00000000 00000000 00000000 10111000 = 184

Figure 9.4 – Signed Left Shift

-75 >> 1
10110101 = -75
10110100 = -38
MSB LSB

Figure 9.5 – Signed Right Shift

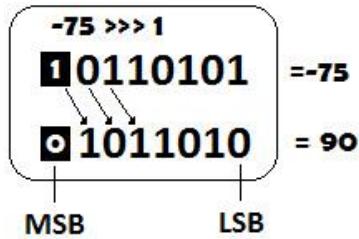


Figure 9.6 – Unsigned Right Shift

Expression	Bit-mask	Example
$1 \ll k$	0000...10000000...	$k=5, 000...100000$
$\sim(1 \ll k)$	111111...01111111...	$k=5, 111...011111$
$(1 \ll k) - 1$	000000...111111...	$k=5, 000...11111$
$-1 \ll (k + 1)$	11111...000000...	$k=5, 111...000000$

Figure 9.7 – Bit-masks

$$\begin{array}{cccccccccc}
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
 & & & & & & & & \\
 \hline
 & & & & & & & 1 & \& \\
 & & & & & & & & 1
 \end{array}$$

Figure 9.8 – Binary representation

$$\begin{array}{cccccccccc}
 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 \hline
 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1
 \end{array} \&$$

Figure 9.9 – Binary representation

$$\begin{array}{cccccccccc}
 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1
 \end{array} |$$

Figure 9.10 – Binary representation

$$\begin{array}{cccccccccc}
 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
 \hline
 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1
 \end{array} \&$$

Figure 9.11 – Binary representation

Figure 9.12 – Summing binary numbers

Figure 9.13 – Multiplying binary numbers

$$q * 1 = \sum_{i=0}^{30} q_{30} * 2^{30} + q_{29} * 2^{29} + \dots + q_0 * 2^0$$

Figure 9.14 – Multiplying binaries in a code

$$result = result + \sum_{i=0}^{30} q_{30} * 2^{30} + q_{29} * 2^{29} + \dots + q_0 * 2^0$$

Figure 9.15 – LSB of p is 1

Figure 9.16 – Subtracting binary numbers

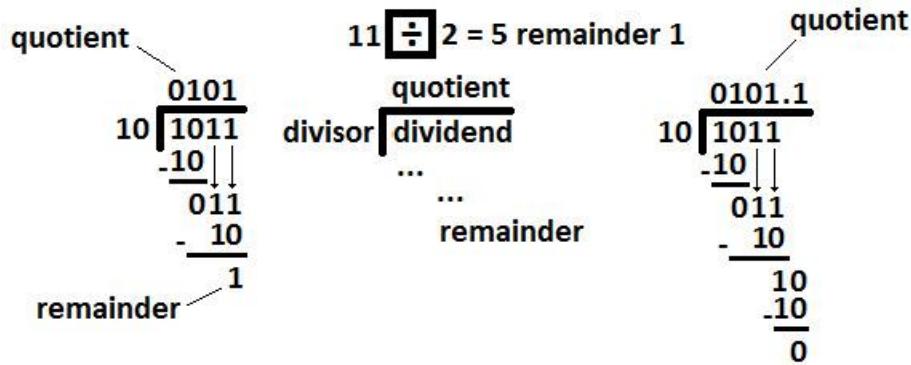


Figure 9.17 – Dividing binary numbers

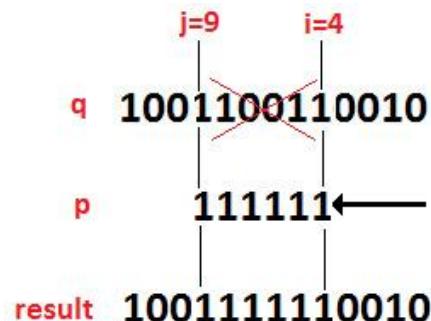


Figure 9.18 – Replacing the bits between **i** and **j**

$$\begin{array}{r}
 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \\
 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\
 \hline
 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0
 \end{array} \&$$

Figure 9.19 – Bit-mask (a)

$$\begin{array}{r}
 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\
 \hline
 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1
 \end{array} |$$

Figure 9.20 – Bit-mask (b)

$$\begin{array}{r}
 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \\
 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \\
 \hline
 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 0
 \end{array} |$$

Figure 9.21 – Binary representation

67534	100000 11111001110	5
67	10000 11	2
339809	10100 10111101100001	9

Figure 9.22 – Three examples

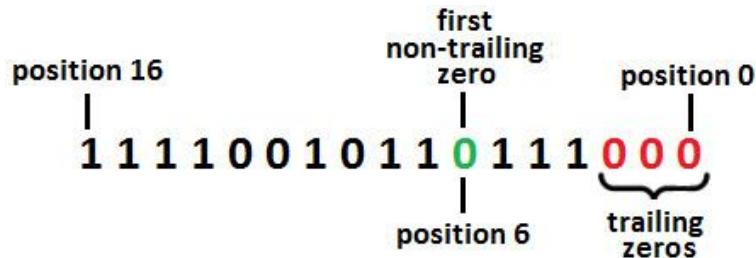


Figure 9.23 – The non-trailing zero

$11110010110111000_{124344} < 1111001011111000_{124408}$ (flip to 1 the bit at position 6)
 $1111001011111000_{124408} > 11110010111011000_{124376}$ (flip to 0 the bit at position 5)
 $11110010111011000_{124376} < 11110010111101000_{124392}$ (flip to 0 the bit at position 4)
 $11110010111101000_{124392} < 11110010111110000_{124400}$ (flip to 0 the bit at position 3)

Figure 9.24 – Several relationships

$$\begin{array}{cccccccccccccccc}
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \textcolor{red}{0} & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \textcolor{red}{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \textcolor{red}{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array} \mid$$

Figure 9.25 – Output (1)

$$\begin{array}{cccccccccccccccc}
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \textcolor{red}{1} & 1 & 1 & 1 & 0 & 0 & 0 \\
 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \textcolor{red}{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 \hline
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \textcolor{red}{1} & 0 & 0 & 0 & 0 & 0 & 0
 \end{array} \&$$

Figure 9.26 – Output (2)

$$\begin{array}{cccccccccccccccc}
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \textcolor{red}{1} & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \textcolor{red}{0} & 0 & 0 & 0 & 0 & 1 & 1 \\
 \hline
 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \textcolor{red}{1} & 0 & 0 & 0 & 0 & 0 & 1 & 1
 \end{array} \mid$$

Figure 9.27 – Output (3)

$$\begin{array}{cccccccccccccccc}
 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
 \hline
 0 & 0 & \textcolor{red}{1} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1
 \end{array} \wedge$$

Figure 9.28 – Conversion

$$(a - b)^2 \geq 0 \rightarrow (a + b)^2 - 4 * a * b \geq 0 \rightarrow a * b \leq \frac{(a + b)^2}{4}$$

Figure 9.29 – Maximizing expressions (1)

$$a * b = \frac{(a + b)^2}{4} \rightarrow 4 * a * b = a + 2 * a * b + b^2$$

Figure 9.30 – Maximizing expressions (2)

$$a^2 - 2 * a * b + b^2 = 0 \rightarrow (a - b)^2 = 0 \rightarrow a = b$$

Figure 9.31 – Maximizing expressions (3)

$$\begin{aligned}(1 \text{ AND } s) &= (1 \text{ AND } s) \rightarrow s = 0 \text{ or } 1 \\ \rightarrow s &= 0 \rightarrow (1 \& 0) = (1 \& 0) = 0 \\ \rightarrow s &= 1 \rightarrow (1 \& 1) = (1 \& 1) = 1 \text{ (we choose } s=1 \text{ to maximize)}\end{aligned}$$

Figure 9.32 – Right shifting q and p by 1 position

$$\begin{aligned}
 (0 \text{ AND } s) &= (1 \text{ AND } s) \rightarrow s = 0 \rightarrow (0 \& 0) = (1 \& 0) = 0 \\
 (0 \text{ AND } s) &= (0 \text{ AND } s) \rightarrow s = 0 \text{ or } 1 \\
 &\rightarrow s = 0 \rightarrow (0 \& 0) = (0 \& 0) = 0 \text{ (we choose } s=0 \text{ since they are the same)} \\
 &\rightarrow s = 1 \rightarrow (0 \& 1) = (0 \& 1) = 0
 \end{aligned}$$

Figure 9.33 – Two cases

$$\begin{array}{ccccccccc} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array} \text{ & }$$

Figure 9.34 – Answer

$$\begin{array}{r} 101010101010101010101010101010101010 \\ 1010010111 \\ \hline 1010000010 \end{array} \&$$

Figure 9.35 – Swapping odd and even bits (1)

$$\begin{array}{r} 1010101010101010101010101010101010101 \\ 1010010111 \\ \hline 0000010101 \end{array} \&$$

Figure 9.36 – Swapping odd and even bits (2)

$$\begin{array}{r}
 010100001 \\
 0000101010 \\
 \hline
 0101101011
 \end{array}$$

Figure 9.37 – Final result

- a) Given integer:
0 0 0 1 1 0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1
- b) Shift left with 10 positions:
1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
padded zeros
0 0 0 1 1 0 0 1 0 0 fallen bits
- c) Expected result:
1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 1 1 0 0 1 0 0
rotated bits

Figure 9.38 – Left rotating bits

$$\begin{array}{r}
 n \ll 10 \quad 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 \\
 n \gg 22 \quad 0 1 1 0 0 1 0 0 \\
 \hline
 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 1 0 0 0 1 1 0 0 1 0 0 0
 \end{array}$$

Figure 9.39 – Applying the OR[] operator

$$\begin{array}{r}
 0 0 0 1 0 0 0 0 0 0 0 0 \\
 0 0 0 0 0 0 0 1 0 0 0 \\
 \hline
 0 0 0 1 1 1 1 0 0 0
 \end{array}$$

Figure 9.40 – Subtraction

1. The given array
[51, 14, 14, 51, 98, 7, 14, 98, 51, 98]

	2. Sum bits:						3. Compute % 3	4. Result
1	1	1	0	0	1	1 (51)	3 % 3 = 0	0000111 = 7
			1	1	1	0 (14)	6 % 3 = 0	
			1	1	1	0 (14)	3 % 3 = 0	
	1	1	0	0	1	1 (51)	3 % 3 = 0	
	1	1	0	0	1	0 (98)	4 % 3 = 1	
				1	1	1 (7)	10 % 3 = 1	
			1	1	1	0 (14)	4 % 3 = 1	
	1	1	0	0	1	0 (98)		
	1	1	0	0	1	1 (51)		
	1	1	0	0	1	0 (98)		
		3	6	3	3	4	10	4

Figure 9.41 – Finding the unique element in the given array

<u>n</u>	<u>n</u>	<u>n-1</u>	<u>n&(n-1)</u>
0	0000	0000	0000 ←
1	0001	0000	0000 ←
2	0010	0001	0000 ←
3	0011	0010	0010
4	0100	0011	0000
5	0101	0100	0100
...			
8	1000	0111	0000 ←
9	1001	1000	1000
...			
15	1111	1110	1110
16	10000	1111	0000 ←
...			

Figure 9.42 – The n & (n-1) formula gives us the powers of two

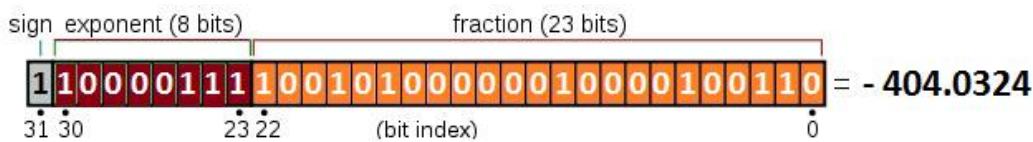


Figure 9.43 – IEEE 754 single-precision binary floating-point (binary 32)

$$v = -1^{sign} \times 2^{(e-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} \right)$$

Figure 9.44 – Float value

Formulas

$$p + q = 2 * (p \& q) + (p \wedge q)$$

Formula 9.1

$$p + q = 2 * and + xor$$

Formula 9.2

$$p + q = xor$$

Formula 9.3

$$p + q = 2 * (2 * and \& xor) + (2 * and \wedge xor)$$

Formula 9.4

*step 1 of recursion: $p + q = 2 * \text{and}\{1\} + \text{xor}\{1\}$*
*step 2 of recursion: $p + q = 2 * \text{and}\{2\} + \text{xor}\{2\}$*
 \dots
*step n of recursion: $p + q = 2 * \text{and}\{n\} + \text{xor}\{n\}$*

Formula 9.5

Mathematical demonstration

For those interested in the mathematical demonstration, let's say that we have the following:

$$(a - b)^2 \geq 0 \rightarrow (a + b)^2 - 4 * a * b \geq 0 \rightarrow a * b \leq \frac{(a + b)^2}{4}$$

Figure 9.29 – Maximizing expressions (1)

This means that we have the following:

$$a * b = \frac{(a + b)^2}{4} \rightarrow 4 * a * b = a + 2 * a * b + b^2$$

Figure 9.30 – Maximizing expressions (2)

Furthermore, this means that we have the following:

$$a^2 - 2 * a * b + b^2 = 0 \rightarrow (a - b)^2 = 0 \rightarrow a = b$$

Figure 9.31 – Maximizing expressions (3)

Code

Code 9.1

In Java, we can quickly see the binary representation of a number via `Integer#toString(int i, int radix)` or `Integer#toBinaryString(int i)`. For example, a radix of 2 means binary:

```
// 110011
System.out.println("Binary: " + Integer.toString(51, 2));
System.out.println("Binary: " + Integer.toBinaryString(51));
```

The reverse process (from binary to decimal) can be obtained via `Integer.parseInt(String nr, int radix)`:

```
System.out.println("Decimal: "
+ Integer.parseInt("110011", 2)); //51
```

Code 9.2

```
public static char getValue(int n, int k) {  
    int result = n & (1 << k);  
    if (result == 0) {  
        return '0';  
    }  
    return '1';  
}
```

Code 9.3

```
public static int setValueTo0(int n, int k) {  
    return n & ~(1 << k);  
}
```

Code 9.4

```
public static int setValueTo1(int n, int k) {  
    return n | (1 << k);  
}
```

Code 9.5

```
public static int clearFromMsb(int n, int k) {  
    return n & ((1 << k) - 1);  
}
```

Code 9.6

```
public static int clearFromPosition(int n, int k) {  
    return n & ~((1 << k) - 1);  
}
```

Code 9.7

```
public static int sum(int q, int p) {  
    int xor;  
    int and;  
    int t;  
    and = q & p;  
    xor = q ^ p;
```

```

// force 'and' to return 0

while (and != 0) {

    and = and << 1; // this is multiplication by 2

    // prepare the next step of recursion

    t = xor ^ and;

    and = and & xor;

    xor = t;

}

return xor;
}

```

Code 9.8

```

public static int multiply(int q, int p) {

    int result = 0;

    while (p != 0) {

        // we compute the value of q only when the LSB of p is
1

        if ((p & 1) != 0) {

            result = result + q;

        }

        q = q << 1; // q is left shifted with 1 position
        p = p >>> 1; // p is logical right shifted with 1 position
    }

    return result;
}

```

Code 9.9

```

public static int subtract(int q, int p) {

    while (p != 0) {

        // borrow the unset bits of q AND set bits of p

        int borrow = (~q) & p;

        // subtraction of bits of q and p
    }
}

```

```

    // where at least one of the bits is not set
    q = q ^ p;
    // left shift borrow by one position
    p = borrow << 1;
}
return q;
}

```

Code 9.10

```

private static final int MAX_BIT = 31;
...
public static long divideWithoutRemainder(long q, long p) {
    // obtain the sign of the division
    long sign = ((q < 0) ^ (p < 0)) ? -1 : 1;
    // ensure that q and p are positive
    q = Math.abs(q);
    p = Math.abs(p);
    long t = 0;
    long quotient = 0;
    for (int i = MAX_BIT; i >= 0; --i) {
        long halfdown = t + (p << i);
        if (halfdown <= q) {
            t = t + p << i;
            quotient = quotient | 1L << i;
        }
    }
    return sign * quotient;
}

```

Code 9.11

```

public static int replace(int q, int p, int i, int j) {
    int ones = ~0; // 11111111 11111111 11111111 11111111

```

```

int leftShiftJ = ones << (j + 1);

int leftShiftI = ((1 << i) - 1);

int mask = leftShiftJ | leftShiftI;

int applyMaskToQ = q & mask;

int bringPInPlace = p << i;

return applyMaskToQ | bringPInPlace;

}

```

Code 9.12

```

public static int sequence(int n) {

if (~n == 0) {
    return Integer.SIZE; // 32
}

int currentSequence = 0;
int longestSequence = 0;
boolean flag = true;

while (n != 0) {
    if ((n & 1) == 1) {
        currentSequence++;
        flag = false;
    } else if ((n & 1) == 0) {
        currentSequence = ((n & 0b10) == 0) // 0b10 = 2
            ? 0 : flag
            ? 0 : ++currentSequence;
        flag = true;
    }
    longestSequence = Math.max(
        currentSequence, longestSequence);
    n >>>= 1;
}
return longestSequence;
}

```

```
}
```

Code 9.13

```
int copyn = n;  
int zeros = 0;  
  
while ((copyn != 0) && ((copyn & 1) == 0)) {  
    zeros++;  
    copyn = copyn >> 1;  
}
```

Code 9.14

```
int ones=0;  
  
while ((copyn & 1) == 1) {  
    ones++;  
    copyn = copyn >> 1;  
}
```

Code 9.15

```
n = n | (1 << marker);
```

Code 9.16

```
n = n & (-1 << marker);
```

Code 9.17

```
n = n | (1 << (ones - 1)) - 1;
```

Code 9.18

```
public static int next(int n) {  
  
    int copyn = n;  
    int zeros = 0;  
    int ones = 0;  
  
    // count trailing 0s  
  
    while ((copyn != 0) && ((copyn & 1) == 0)) {  
        zeros++;  
        copyn = copyn >> 1;  
    }
```

```

// count all 1s until first 0

while ((copyn & 1) == 1) {
    ones++;
    copyn = copyn >> 1;
}

// the 1111...000... is the biggest number
// without adding more 1

if (zeros + ones == 0 || zeros + ones == 31) {
    return -1;
}

int marker = zeros + ones;

n = n | (1 << marker);
n = n & (-1 << marker);
n = n | (1 << (ones - 1)) - 1;

return n;
}

```

Code 9.19

```

public static int count(int q, int p) {

    int count = 0;

    // each 1 represents a bit that is
    // different between q and p

    int xor = q ^ p;

    while (xor != 0) {

        count += xor & 1; // only 1 & 1 = 1
        xor = xor >> 1;

    }

    return count;
}

```

Code 9.20

```

public static int swap(int n) {

```

```

int moveToEvenPositions
= (n & 0b10101010101010101010101010101010) >>> 1;

int moveToOddPositions
= (n & 0b1010101010101010101010101010101) << 1;

return moveToEvenPositions | moveToOddPositions;
}

```

Code 9.21

```

public static int leftRotate(int n, int bits) {
    int fallBits = n << bits;
    int fallBitsShiftToRight = n >> (MAX_INT_BITS - bits);
    return fallBits | fallBitsShiftToRight;
}

```

Code 9.22

```

public static int rightRotate(int n, int bits) {
    int fallBits = n >> bits;
    int fallBitsShiftToLeft = n << (MAX_INT_BITS - bits);
    return fallBits | fallBitsShiftToLeft;
}

```

Code 9.23

```

public static int setBetween(int left, int right) {
    return (1 << (right + 1)) - (1 << left);
}

```

Code 9.24

```

private static final int INT_SIZE = 32;
public static int unique(int arr[]) {
    int n = arr.length;
    int result = 0;
    int nr;
    int sumBits;
    // iterate through every bit

```

```

for (int i = 0; i < INT_SIZE; i++) {
    // compute the sum of set bits at
    // ith position in all array
    sumBits = 0;
    nr = (1 << i);
    for (int j = 0; j < n; j++) {
        if ((arr[j] & nr) == 0) {
            sumBits++;
        }
    }
    // the sum not multiple of 3 are the
    // bits of the unique number
    if ((sumBits % 3) == 0) {
        result = result | nr;
    }
}
return result;
}

```

Code 9.25

```

public static int unique(int arr[]) {
    int oneAppearance = 0;
    int twoAppearances = 0;
    for (int i = 0; i < arr.length; i++) {
        twoAppearances = twoAppearances
            | (oneAppearance & arr[i]);
        oneAppearance = oneAppearance ^ arr[i];
        int neutraliser = ~(oneAppearance & twoAppearances);
        oneAppearance = oneAppearance & neutraliser;
        twoAppearances = twoAppearances & neutraliser;
    }
}

```

```
    return oneAppearance;
}
```

Code 9.26

```
private static final int MAX_N = 32000;

public static void printDuplicates(int[] arr) {

    BitSet bitArr = new BitSet(MAX_N);

    for (int i = 0; i < arr.length; i++) {

        int nr = arr[i];

        if (bitArr.get(nr)) {

            System.out.println("Duplicate: " + nr);

        } else {

            bitArr.set(nr);

        }

    }

}
```

Code 9.27

```
public static void findNonRepeatable(int arr[]) {

    // get the XOR[^] of all elements in the given array

    int xor = arr[0];

    for (int i = 1; i < arr.length; i++) {

        xor ^= arr[i];

    }

    // get the rightmost set bit (you can use any other set bit)

    int setBitNo = xor & ~(xor - 1);

    // divide the elements in two sets by comparing the

    // rightmost set bit of XOR[^] with the bit at the same

    // position in each element

    int p = 0;

    int q = 0;

    for (int i = 0; i < arr.length; i++) {
```

```

if ((arr[i] & setBitNo) != 0) {
    // xor of the first set
    p = p ^ arr[i];
} else {
    // xor of the second set
    q = q ^ arr[i];
}
}

System.out.println("The numbers are: " + p + " and " + q);
}

```

Code 9.28

Compute the Power Set size as $2^{\text{size of } S}$

Iterate via i from 0 to Power Set size

 Iterate via j from 0 to size of S

 If j th bit in i is set then

 Add j th element from set to current subset

 Add the resulted subset to subsets

Return all subsets

Code 9.29

```

public static Set<Set<Character>> powerSet(char[] set) {
    // total number of subsets ( $2^n$ )
    long subsetsNo = (long) Math.pow(2, set.length);
    // store subsets
    Set<Set<Character>> subsets = new HashSet<>();
    // generate each subset one by one
    for (int i = 0; i < subsetsNo; i++) {
        Set<Character> subset = new HashSet<>();
        // check every bit of i
        for (int j = 0; j < set.length; j++) {
            // if  $j$ 'th bit of  $i$  is set,

```

```

    // add set[j] to the current subset
    if ((i & (1 << j)) != 0) {
        subset.add(set[j]);
    }
    subsets.add(subset);
}
return subsets;
}

```

Code 9.30

```

public static int findPosition(int n) {
    int count = 0;
    if (!isPowerOfTwo(n)) {
        return -1;
    }
    while (n != 0) {
        n = n >> 1;
        ++count;
    }
    return count;
}

private static boolean isPowerOfTwo(int n) {
    return (n > 0) && ((n & (n - 1)) == 0);
}

```

Chapter 10

Images

a b c d
豈更車賈滑更
a 豈 b 更

Figure 10.1 – Strings

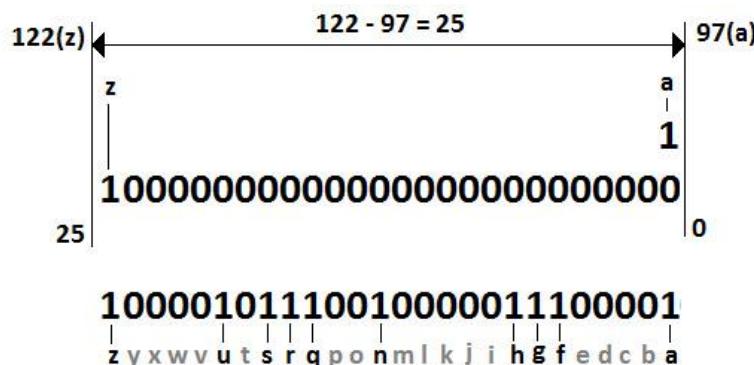


Figure 10.2 – Unique characters bit mask

Two Hearts	♥♥	Java Representation: \uD83D\uDC95
		Code point: 128149 (56469)
Musical Score	ℳ	Java Representation: \uD83C\uDFBC
		Code Point: 127932 (57276)
Smiley Face	😊	Java Representation: \uD83D\uDE0D
		Code Point: 128525 (56845)
Cyrillic Capital Letter ZHE with Diaeresis	Ӯ	Java Representation: \u04DC
		Code Point: 1244

Figure 10.3 – Unicode characters (surrogate pairs)

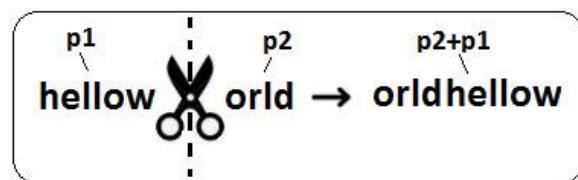


Figure 10.4 – Cutting str1 into two parts and rearranging them

$$[M^T]_{ij} = [M]_{ji}$$

Figure 10.5 – Matrix transpose relationship

M $[M]_{ji} = [M^T]_{ij}$	M^T Reversing columns of the transpose
------------------------------------	--

Figure 10.6 – The transpose of a matrix on the left and the final result on the right

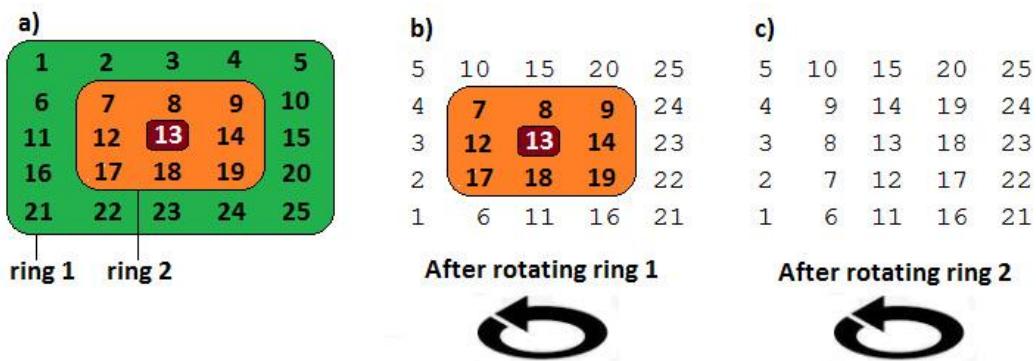


Figure 10.7 – Rotating a matrix ring by ring

Initial matrix

1	2	3	4	0	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	0	22
23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38

Solved matrix

0	0	0	0	0	0	0	0
8	9	10	11	0	13	0	15
0	0	0	0	0	0	0	0
23	24	25	26	0	28	0	30
31	32	33	34	0	36	0	38

Figure 10.8 – Matrix containing zeros

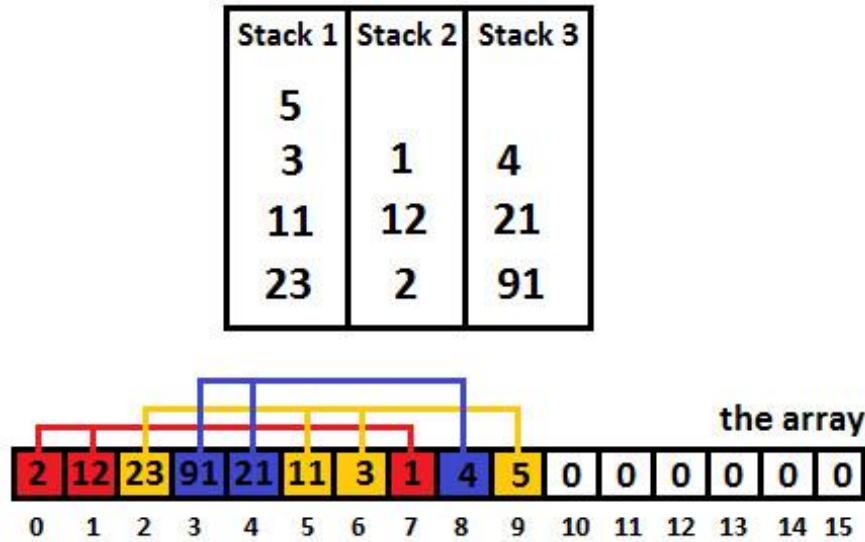


Figure 10.9 – Interleaving the nodes of the stacks

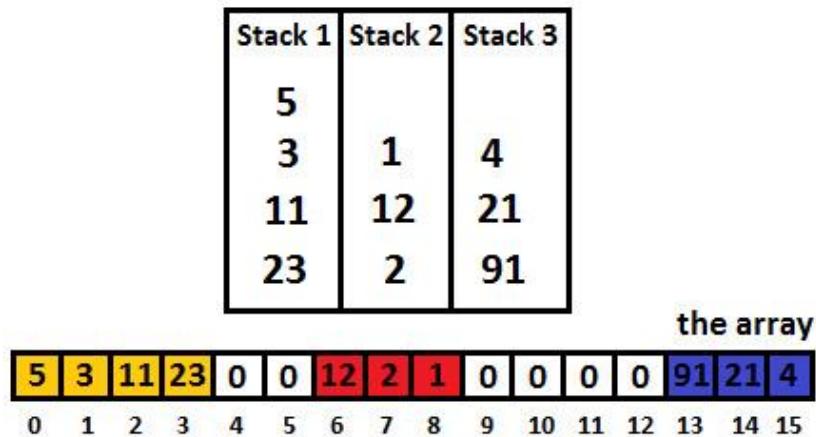


Figure 10.10 – Splitting the array into three zones

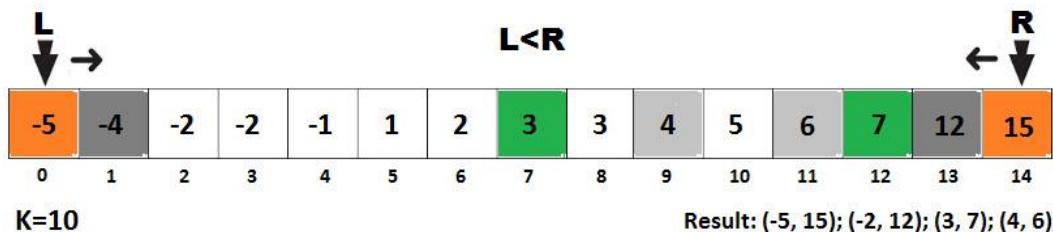


Figure 10.11 – Finding all pairs whose sum is equal to the given number

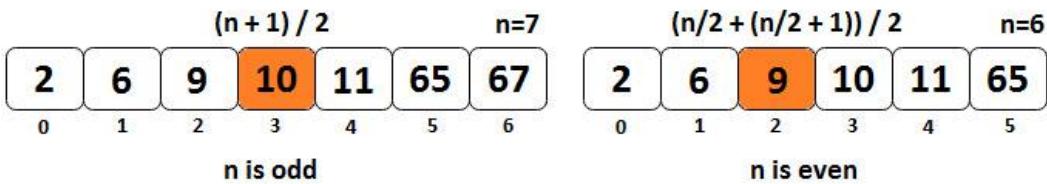


Figure 10.12 – Median values for odd and even arrays

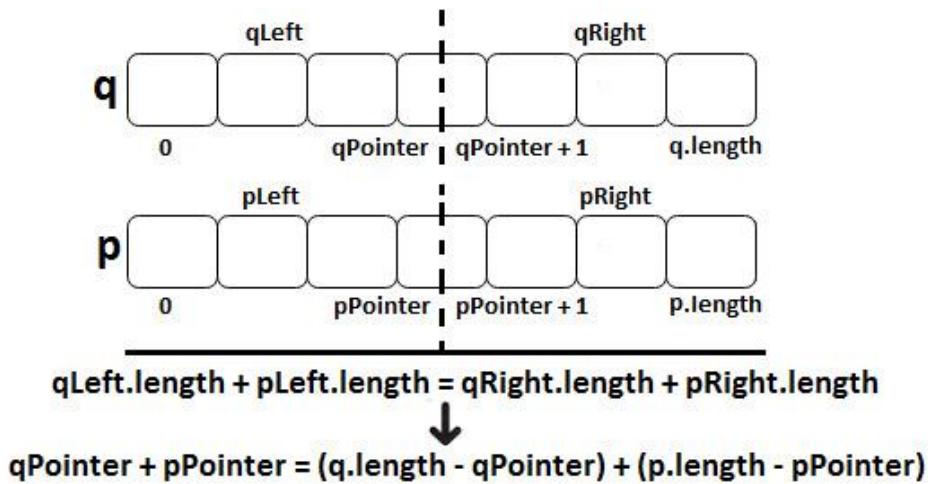


Figure 10.13 – Splitting arrays into two equal parts

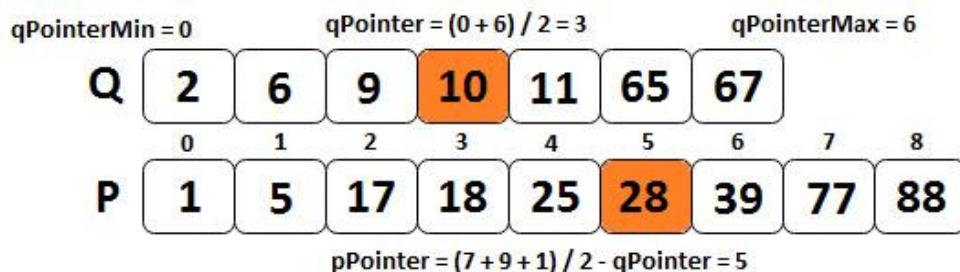


Figure 10.14 – Computing the median value (step 1)

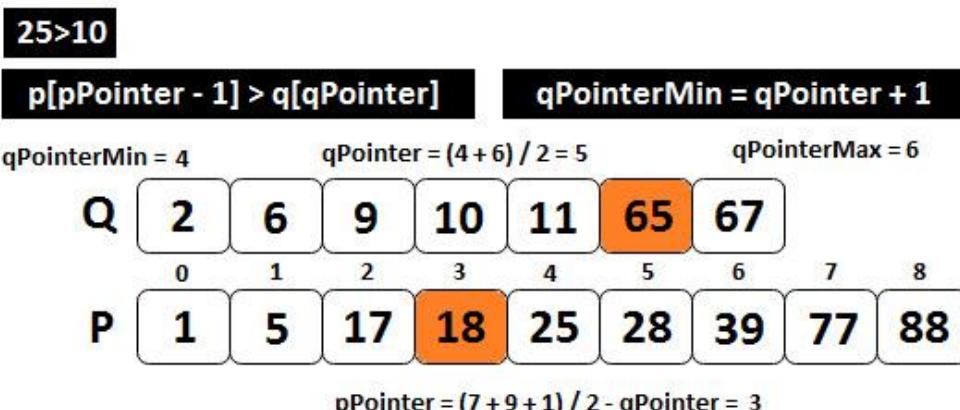


Figure 10.15 – Computing the median value (step 2)

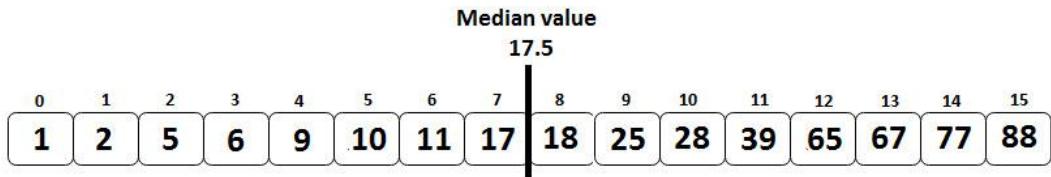


Figure 10.16 – Median value (final result)

0	0	1	1	0	0	0
0	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	0	1	1	1
0	1	1	0	0	0	0

Figure 10.17 – The given 5 x 7 binary matrix

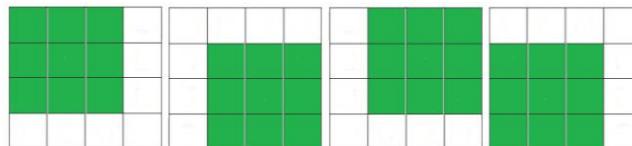


Figure 10.18 – Maxim sub-matrix of 1s in a 4 x 4 matrix

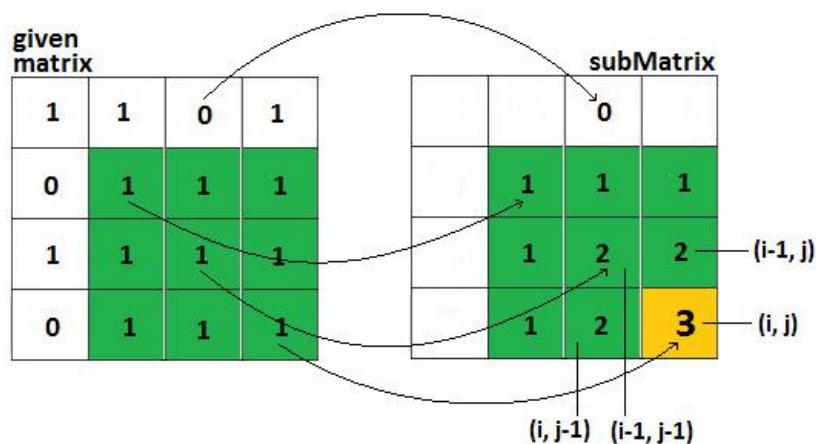


Figure 10.19 – Overall recurrence relation

0	0	1	1	0	0	0
0	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	0	1	1	1
0	1	1	0	0	0	0

given matrix

0	0	1	1	0	0	0
0	0	1	2	1	1	1
1	1	0	0	1	2	2
1	2	0	0	1	2	3
0	1	1	0	0	0	0

subMatrix

Figure 10.20 – Resolving our 5×7 matrix

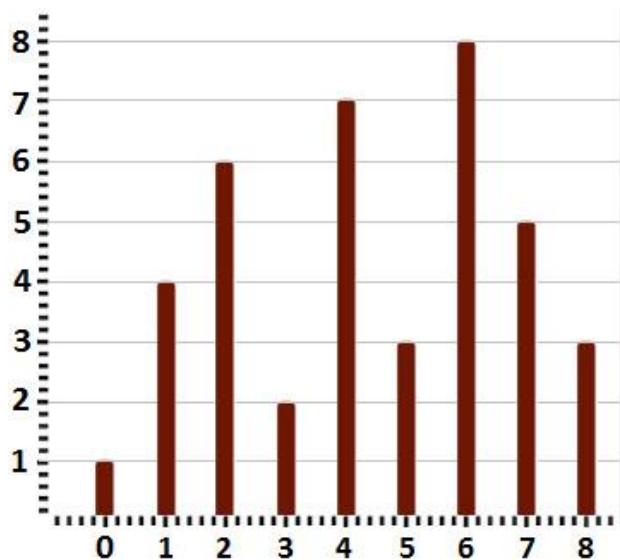


Figure 10.21 – The n vertical line representation

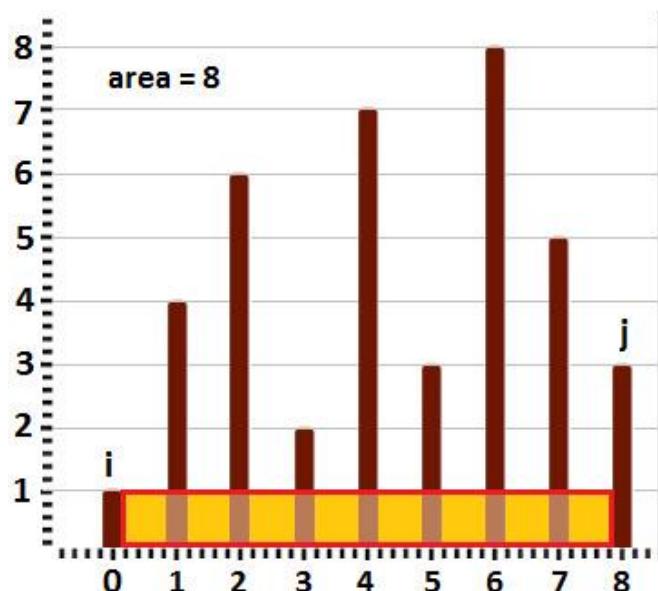


Figure 10.22 – Area with the biggest width

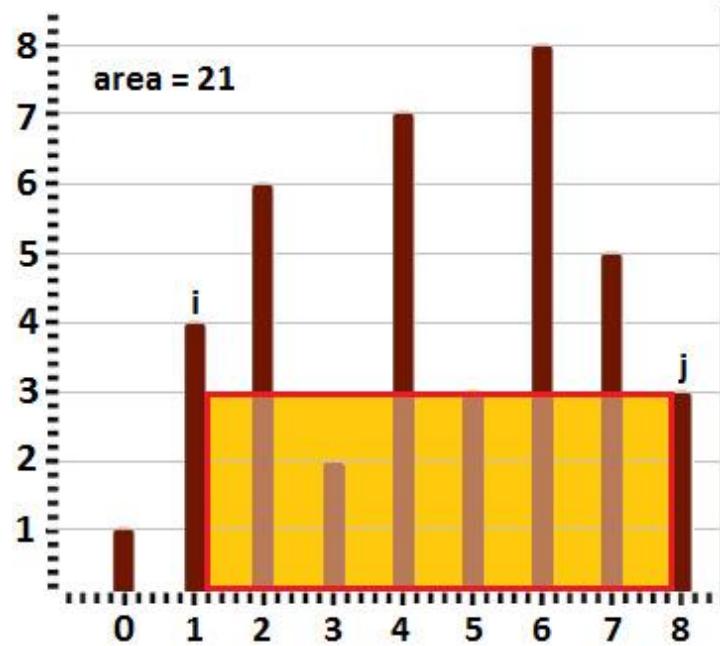


Figure 10.23 – Increasing i to obtain a bigger container

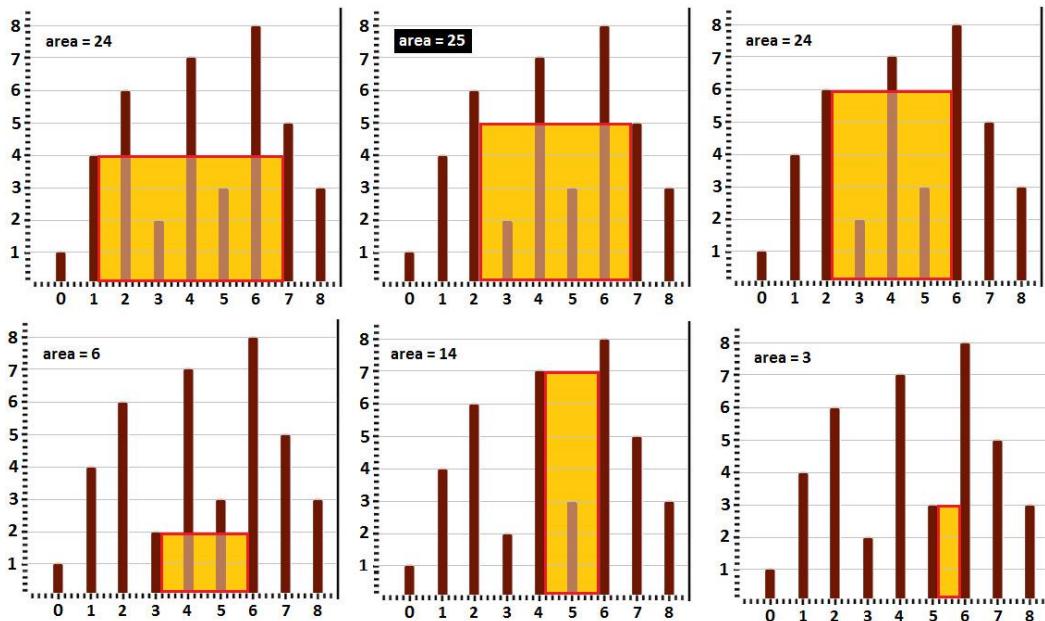


Figure 10.24 – Computing areas while increasing/decreasing i and j

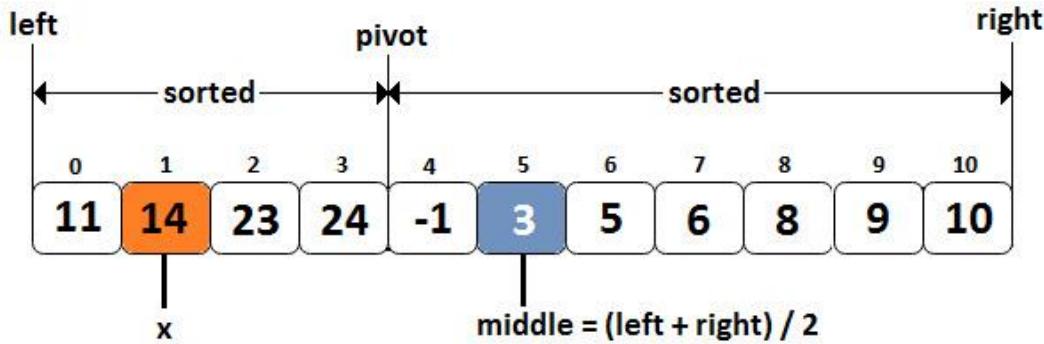


Figure 10.25 – Circularly sorted array and Binary Search algorithm

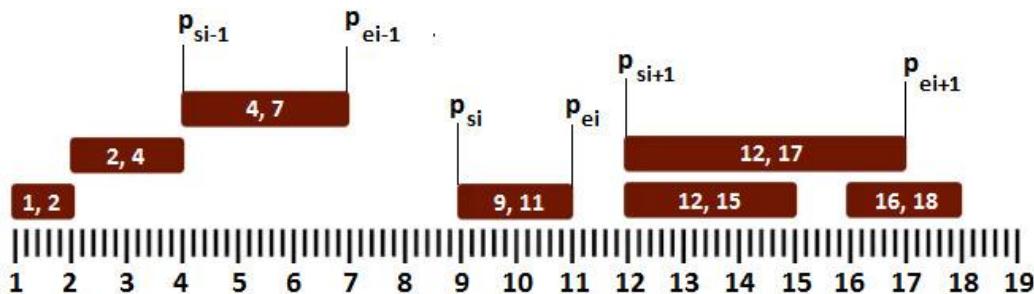


Figure 10.26 – Sorting the given intervals

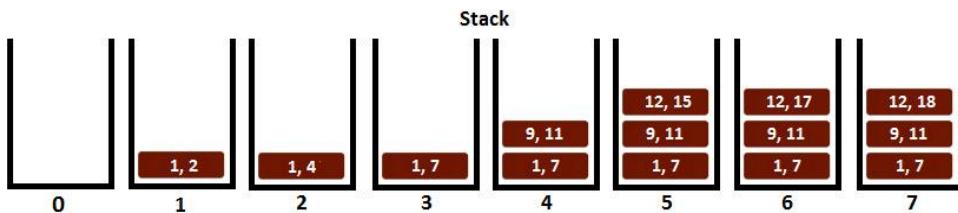


Figure 10.27 – Using a stack to solve the problem

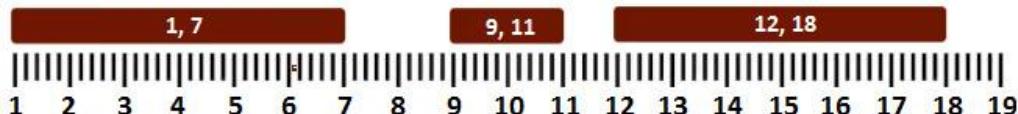


Figure 10.28 – The merged intervals

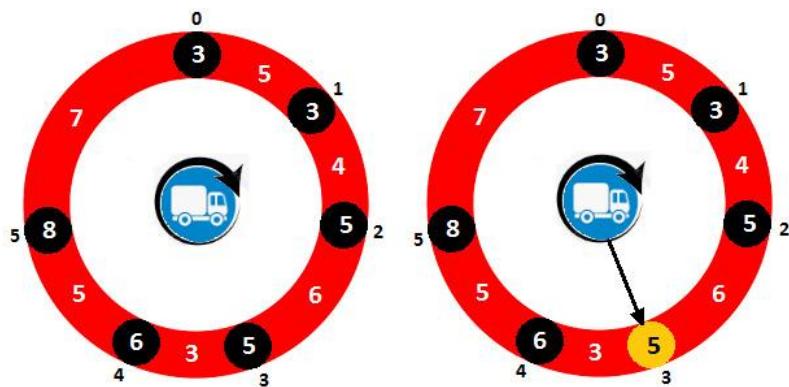


Figure 10.29 – Truck circular tour sample

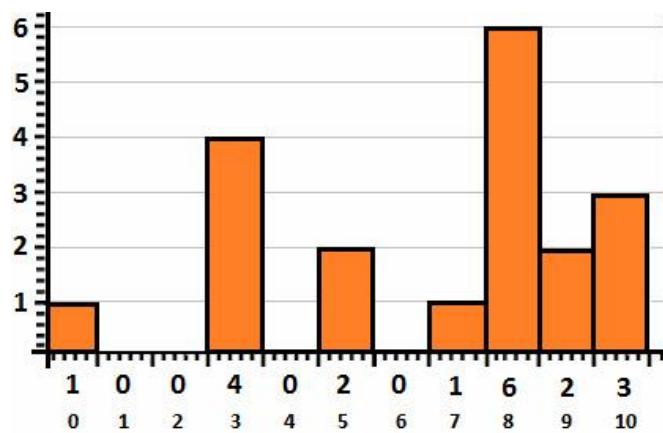


Figure 10.30 – The given set of bars

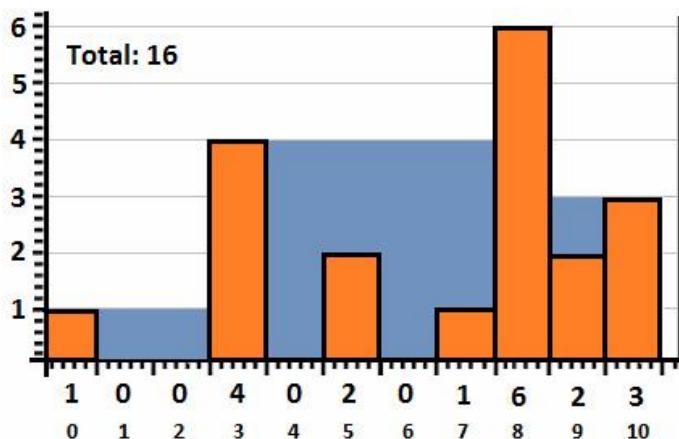


Figure 10.31 – The given bars after rain

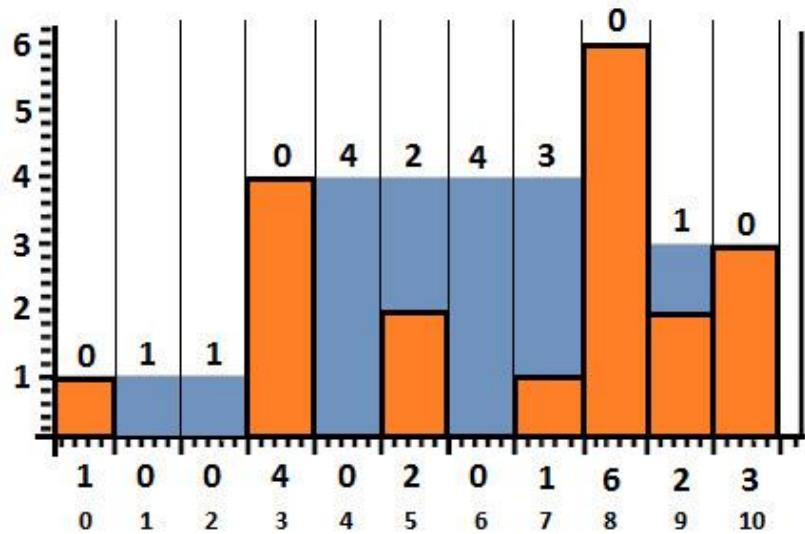


Figure 10.32 – Water on top of each bar

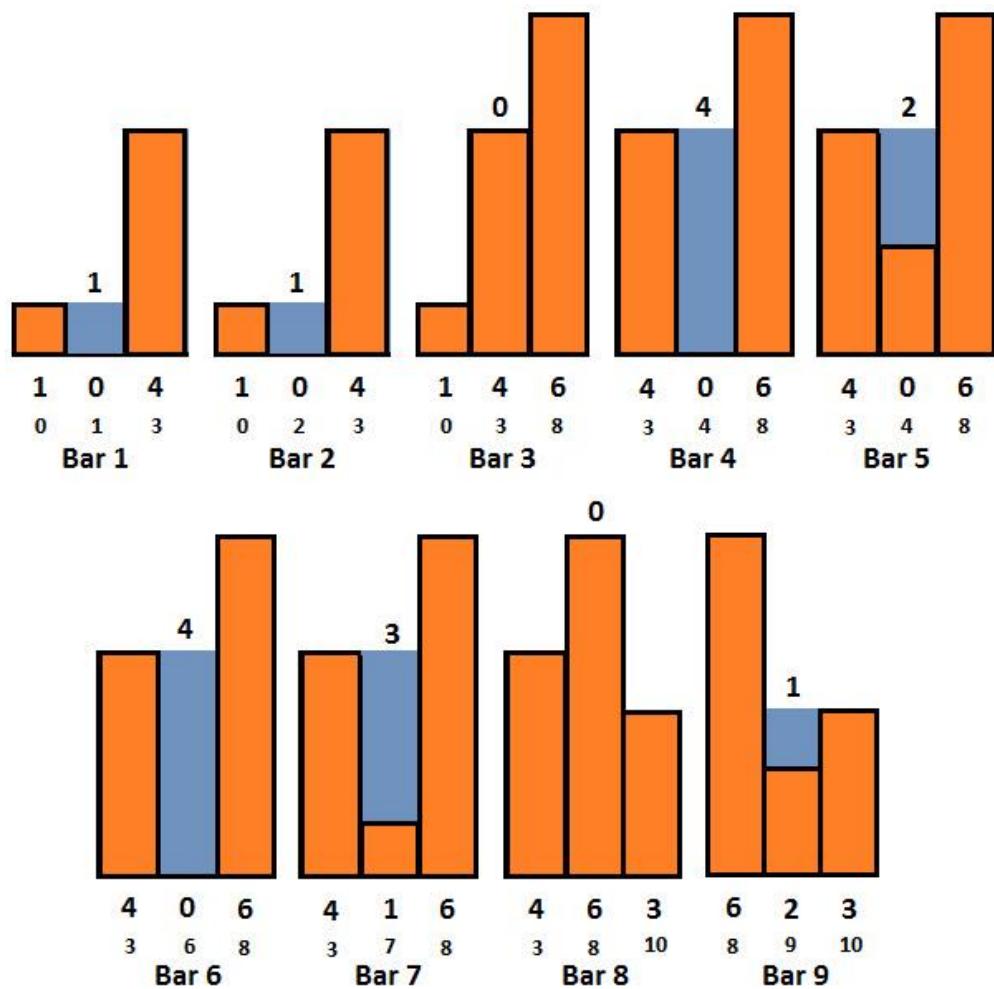


Figure 10.33 – Highest bars on the left- and right-hand sides

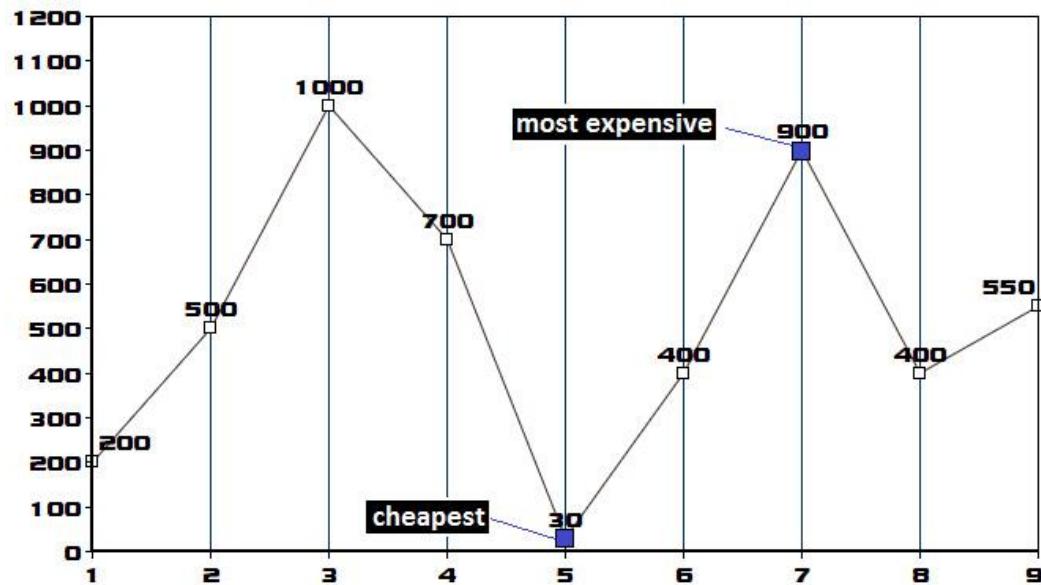


Figure 10.34 – Price-trend graph

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Price	200	500	1000	700	30	400	900	400	550
Min price (from day 1 till today)	200	200	200	200	30	30	30	30	30
Price - Min price	0	300	800	500	0	370	870	370	520
Max profit max(Max profit, Price - Min Price)	200	300	800	800	800	800	870	870	870

maximum profit

Figure 10.35 – Computing the maximum profit

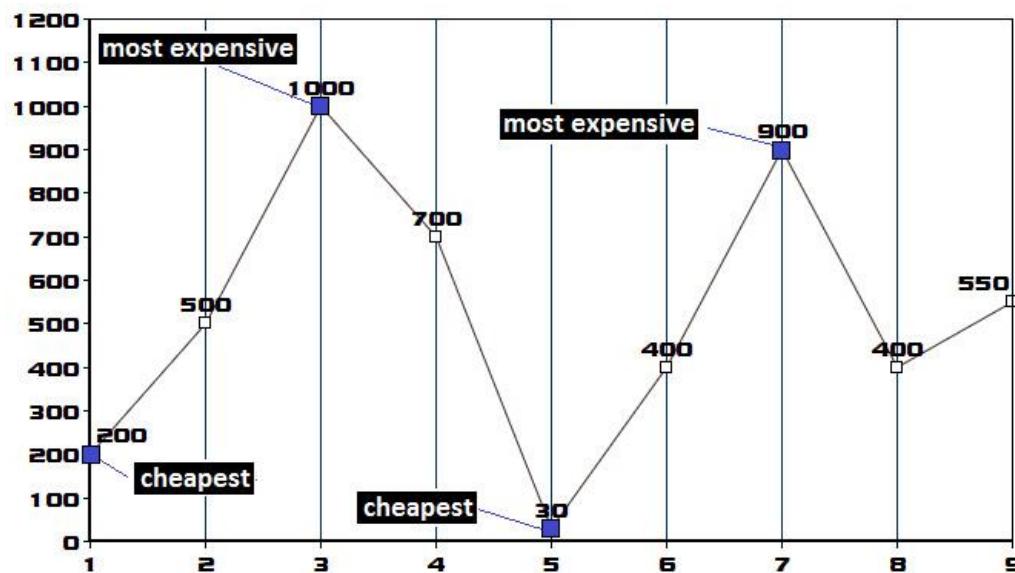


Figure 10.36 – Price-trend graph

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Price	200	500	1000	700	30	400	900	400	550
left[day]	0	300	800	800	800	800	870	870	870
maximum profit till this day	left[0]	left[1]	left[2]	left[3]	left[4]	left[5]	left[6]	left[7]	left[8]

Figure 10.37 – Computing the maximum profit before each day, starting from day 1

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Price	200	500	1000	700	30	400	900	400	550
right[day]	870	870	870	870	870	500	150	150	0
maximum profit after this day	right[0]	right[1]	right[2]	right[3]	right[4]	right[5]	right[6]	right[7]	right[8]

Figure 10.38 – Computing the maximum profit after each day, starting from the previous day

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Price	200	500	1000	700	30	400	900	400	550
transaction 1, left[day]	0	300	800	800	800	800	870	870	870
transaction2, right[day]	870	870	870	870	870	500	150	150	0
max(left[day]+right[day])	870	1170	1670	1670	1670	1300	1020	1020	870

maximum profit

Figure 10.39 – Computing the final maximum profit of transactions 1 and 2

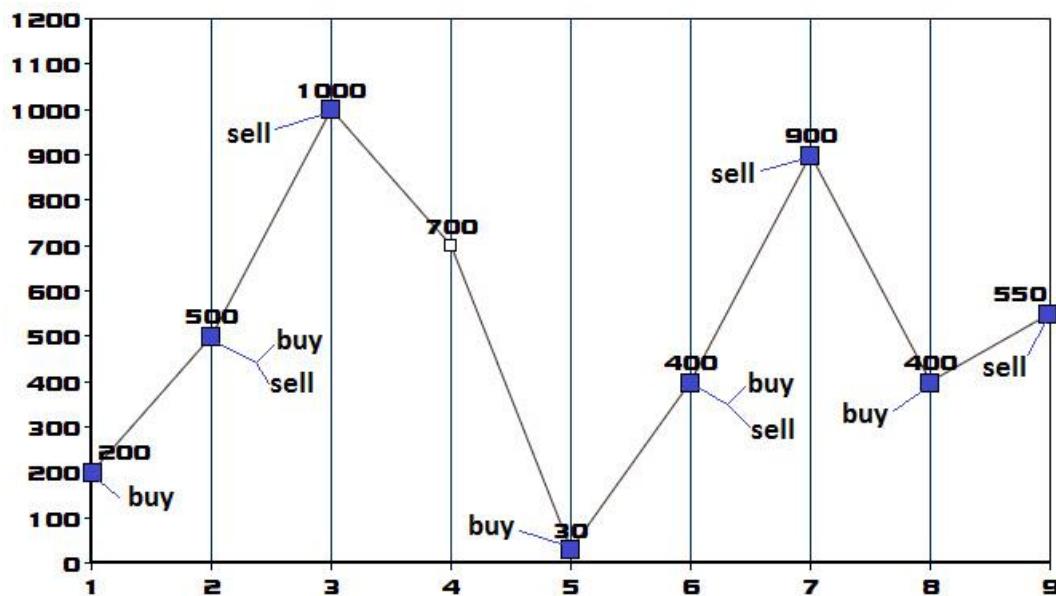


Figure 10.40 – Price-trend graph

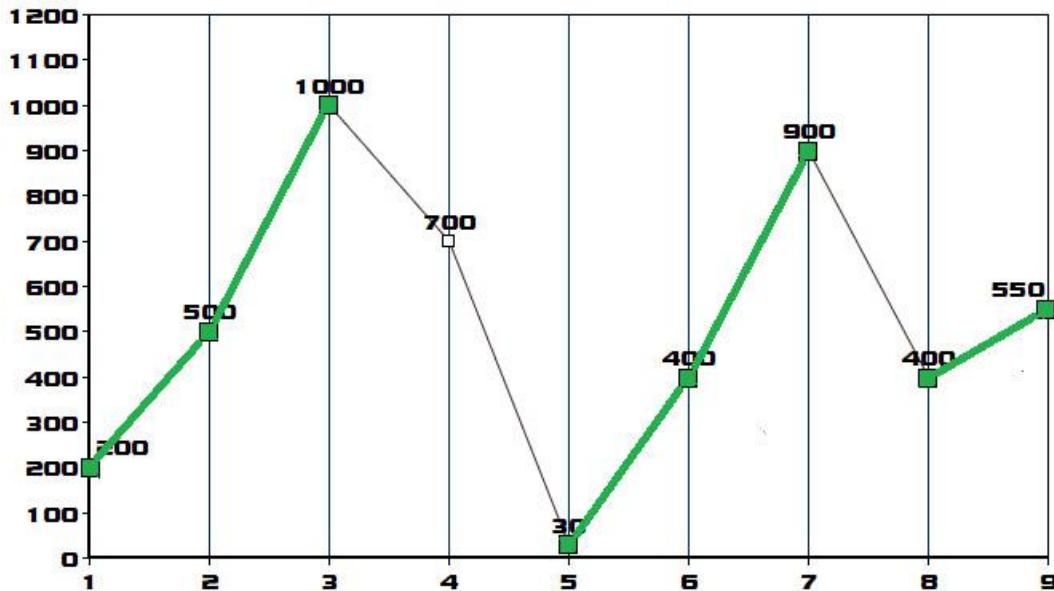


Figure 10.41 – Ascending sequences

	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9
Price	200	500	1000	700	30	400	900	400	550
Max profit	0	300	800	800	800	1170	1670	1670	1820

maximum profit

Figure 10.42 – Computing the final maximum profit

Given array:

[4, 2, 9, 5, 12, 6, 8] → Set:

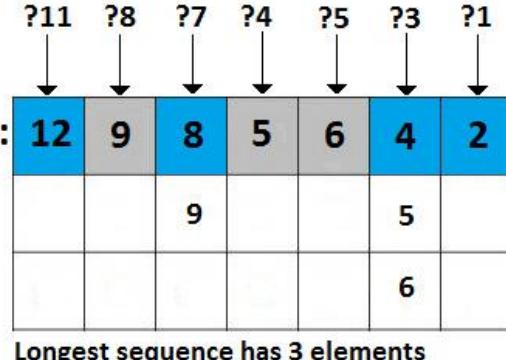


Figure 10.43 – Sequence set

Links

- Official Java documentation: <https://docs.oracle.com/javase/tutorial/java/>
- Java Coding Problems: <https://www.amazon.com/gp/product/1789801419>
- Sliding Window Technique by Zengrui Wang: <https://medium.com/@zengruiwang/sliding-window-technique-360d840d5740>

Code

Code 10.1

```
private static final int MAX_CODE = 65535;  
...  
public static boolean isUnique(String str) {  
    Map<Character, Boolean> chars = new HashMap<>();  
    // or use, for(char ch : str.toCharArray()) { ... }  
    for (int i = 0; i < str.length(); i++) {  
        if (str.codePointAt(i) <= MAX_CODE) {  
            char ch = str.charAt(i);  
            if (!Character.isWhitespace(ch)) {  
                if (chars.put(ch, true) != null) {  
                    return false;  
                }  
            }  
        } else {  
            System.out.println("The given string  
                contains unallowed characters");  
            return false;  
        }  
    }  
    return true;  
}
```

Code 10.2

```
private static final char A_CHAR = 'a';

...

public static boolean isUnique(String str) {

    int marker = 0;

    for (int i = 0; i < str.length(); i++) {

        int s = str.charAt(i) - A_CHAR;

        int mask = 1 << s;

        if ((marker & mask) > 0) {

            return false;

        }

        marker = marker | mask;

    }

    return true;

}
```

Code 10.3

```
char[] str = " String with spaces ".toCharArray();
```

Code 10.4

```
public static char[] encodeWhitespaces(char[] str) {

    // count whitespaces (step 1)

    int countWhitespaces = 0;

    for (int i = 0; i < str.length; i++) {

        if (Character.isWhitespace(str[i])) {

            countWhitespaces++;

        }

    }

    if (countWhitespaces > 0) {

        // create the encoded char[] (step 2)

        char[] encodedStr = new char[str.length

            + countWhitespaces * 2];


```

```

// populate the encoded char[] (step 3)

int index = 0;

for (int i = 0; i < str.length; i++) {

    if (Character.isWhitespace(str[i])) {

        encodedStr[index] = '0';

        encodedStr[index + 1] = '2';

        encodedStr[index + 2] = '%';

        index = index + 3;

    } else {

        encodedStr[index] = str[i];

        index++;

    }

}

return encodedStr;

}

return str;
}

```

Code 10.5

```

if (Math.abs(q.length() - p.length()) > 1) {

    return false;

}

```

Code 10.6

```

public static boolean isOneEditAway(String q, String p) {

    // if the difference between the strings is bigger than 1
    // then they are at more than one edit away

    if (Math.abs(q.length() - p.length()) > 1) {

        return false;

    }

    // get shorter and longer string

    String shorter = q.length() < p.length() ? q : p;

```

```

String longer = q.length() < p.length() ? p : q;
int is = 0;
int il = 0;
boolean marker = false;

while (is < shorter.length() && il < longer.length()) {
    if (shorter.charAt(is) != longer.charAt(il)) {
        // first difference was found
        // at the second difference we return false
        if (marker) {
            return false;
        }
        marker = true;
    }
    if (shorter.length() == longer.length()) {
        is++;
    } else {
        is++;
    }
    il++;
}
return true;
}

```

Code 10.7

```

public static String shrink(String str) {
    StringBuilder result = new StringBuilder();
    int count = 0;
    for (int i = 0; i < str.length(); i++) {
        count++;
        // we don't count whitespaces, we just copy them
        if (!Character.isWhitespace(str.charAt(i))) {

```

```

        // if there are no more characters
        // or the next character is different
        // from the counted one
        if ((i + 1) >= str.length())
            || str.charAt(i) != str.charAt(i + 1)) {
            // append to the final result the counted character
            // and number of consecutive occurrences
            result.append(str.charAt(i))
                .append(count);
            // reset the counter since this
            // sequence was appended to the result
            count = 0;
        }
    } else {
        result.append(str.charAt(i));
        count = 0;
    }
}
// return the result only if it is
// shorter than the given string
return result.length() > str.length()
    ? str : result.toString();
}

```

Code 10.8

```

public static List<Integer> extract(String str) {
    List<Integer> result = new ArrayList<>();
    StringBuilder temp = new StringBuilder(
        String.valueOf(Integer.MAX_VALUE).length());
    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);

```

```

// or, if (((int) ch) >= 48 && ((int) ch) <= 57)
if (Character.isDigit(ch)) {
    temp.append(ch);
} else {
    if (temp.length() > 0) {
        result.add(Integer.parseInt(temp.toString()));
        temp.delete(0, temp.length());
    }
}
return result;
}

```

Code 10.9

```

char[] musicalScore = new char[]{'\uD83C', '\uDFBC'};

char[] smileyFace = new char[]{'\uD83D', '\uDE0D'};

char[] twoHearts = new char[]{'\uD83D', '\uDC95'};

char[] cyrillicZhe = new char[]{'\u04DC'};

String str = "is" + String.valueOf(cyrillicZhe) + "zhe"
+ String.valueOf(twoHearts) + "two hearts"
+ String.valueOf(smileyFace) + "smiley face and, "
+ String.valueOf(musicalScore) + "musical score";

```

Code 10.10

```
String str = String.valueOf(Character.toChars(128149));
```

Code 10.11

```

public static List<Integer> extract(String str) {
    List<Integer> result = new ArrayList<>();
    for (int i = 0; i < str.length(); i++) {
        int cp = str.codePointAt(i);
        if (i < str.length()-1
            && str.codePointCount(i, i+2) == 1) {

```

```

        result.add(cp);

        result.add(str.codePointAt(i+1));

        i++;

    }

}

return result;

}

```

Code 10.12

```

public static List<Integer> extract(String str) {

    List<Integer> result = new ArrayList<>();

    for (int i = 0; i < str.length(); i++) {

        int cp = str.codePointAt(i);

        // the constant 2 means a surogate pair

        if (Character.charCount(cp) == 2) {

            result.add(cp);

            result.add(str.codePointAt(i+1));

            i++;

        }

    }

    return result;

}

```

Code 10.13

```

public static boolean isRotation(String str1, String str2) {

    return (str1 + str1).matches("(?i).*"
        + Pattern.quote(str2) + ".*");

}

```

Code 10.14

```

private static void transpose(int m[][])

{
    for (int i = 0; i < m.length; i++) {

        for (int j = i; j < m[0].length; j++) {

```

```

        int emp = m[j][i];
        m[j][i] = m[i][j];
        m[i][j] = temp;
    }
}
}

```

Code 10.15

```

public static boolean rotateWithTranspose(int m[][])) {
    transpose(m);
    for (int i = 0; i < m[0].length; i++) {
        for (int j = 0, k = m[0].length - 1; j < k; j++, k--) {
            int temp = m[j][i];
            m[j][i] = m[k][i];
            m[k][i] = temp;
        }
    }
    return true;
}

```

Code 10.16

```

public static boolean rotateRing(int[][] m) {
    int len = m.length;
    // rotate counterclockwise
    for (int i = 0; i < len / 2; i++) {
        for (int j = i; j < len - i - 1; j++) {
            int temp = m[i][j];
            // right -> top
            m[i][j] = m[j][len - 1 - i];
            // bottom -> right
            m[j][len - 1 - i] = m[len - 1 - i][len - 1 - j];
            // left -> bottom
        }
    }
}

```

```

        m[len - 1 - i][len - 1 - j] = m[len - 1 - j][i];
        // top -> left
        m[len - 1 - j][i] = temp;
    }
}

return true;
}

```

Code 10.17

```

boolean firstRowHasZeros = false;

boolean firstColumnHasZeros = false;

// Search at least a zero on first row
for (int j = 0; j < m[0].length; j++) {
    if (m[0][j] == 0) {
        firstRowHasZeros = true;
        break;
    }
}

// Search at least a zero on first column
for (int i = 0; i < m.length; i++) {
    if (m[i][0] == 0) {
        firstColumnHasZeros = true;
        break;
    }
}

```

Code 10.18

```

// Search all zeros in the rest of the matrix
for (int i = 1; i < m.length; i++) {
    for (int j = 1; j < m[0].length; j++) {
        if (m[i][j] == 0) {
            m[i][0] = 0;
        }
    }
}

```

```

        m[0][j] = 0;
    }
}
}

```

Code 10.19

```

for (int i = 1; i < m.length; i++) {
    if (m[i][0] == 0) {
        setRowOfZero(m, i);
    }
}

for (int j = 1; j < m[0].length; j++) {
    if (m[0][j] == 0) {
        setColumnOfZero(m, j);
    }
}

```

Code 10.20

```

if (firstRowHasZeros) {
    setRowOfZero(m, 0);
}

if (firstColumnHasZeros) {
    setColumnOfZero(m, 0);
}

```

Code 10.21

```

private static void setRowOfZero(int[][] m, int r) {
    for (int j = 0; j < m[0].length; j++) {
        m[r][j] = 0;
    }
}

private static void setColumnOfZero(int[][] m, int c) {
    for (int i = 0; i < m.length; i++) {

```

```
    m[i][c] = 0;  
}  
}
```

Code 10.22

```
public class StackNode {  
    int value;  
    int backLink;  
    StackNode(int value, int backLink) {  
        this.value = value;  
        this.backLink = backLink;  
    }  
}
```

Code 10.23

```
public class ThreeStack {  
    private static final int STACK_CAPACITY = 15;  
    // the array of stacks  
    private final StackNode[] theArray;  
    ThreeStack() {  
        theArray = new StackNode[STACK_CAPACITY];  
        initializeSlots();  
    }  
    ...  
    private void initializeSlots() {  
        for (int i = 0; i < STACK_CAPACITY; i++) {  
            theArray[i] = new StackNode(0, i + 1);  
        }  
    }  
}
```

Code 10.24

```
public class ThreeStack {
```

```
private static final int STACK_CAPACITY = 15;

private int size;

// next free slot in array

private int nextFreeSlot;

// the array of stacks

private final StackNode[] theArray;

// maintain the parent for each node

private final int[] backLinks = {-1, -1, -1};

...

public void push(int stackNumber, int value)

    throws OverflowException {

    int stack = stackNumber - 1;

    int free = fetchIndexOfFreeSlot();

    int top = backLinks[stack];

    StackNode node = theArray[free];

    // link the free node to the current stack

    node.value = value;

    node.backLink = top;

    // set new top

    backLinks[stack] = free;

}

private int fetchIndexOfFreeSlot()

    throws OverflowException {

    if (size >= STACK_CAPACITY) {

        throw new OverflowException("Stack Overflow");

    }

    // get next free slot in array

    int free = nextFreeSlot;

    // set next free slot in array and increase size

    nextFreeSlot = theArray[free].backLink;
```

```
    size++;
    return free;
}
}
```

Code 10.25

```
public class ThreeStack {
    private static final int STACK_CAPACITY = 15;
    private int size;
    // next free slot in array
    private int nextFreeSlot;
    // the array of stacks
    private final StackNode[] theArray;
    // maintain the parent for each node
    private final int[] backLinks = {-1, -1, -1};
    ...
    public StackNode pop(int stackNumber)
        throws UnderflowException {
        int stack = stackNumber - 1;
        int top = backLinks[stack];
        if (top == -1) {
            throw new UnderflowException("Stack Underflow");
        }
        StackNode node = theArray[top]; // get the top node
        backLinks[stack] = node.backLink;
        freeSlot(top);
        return node;
    }
    private void freeSlot(int index) {
        theArray[index].backLink = nextFreeSlot;
        nextFreeSlot = index;
    }
}
```

```
    size--;
}
}
```

Code 10.26

```
public static List<String> pairs(int[] m, int k) {
    if (m == null || m.length < 2) {
        return Collections.emptyList();
    }
    List<String> result = new ArrayList<>();
    java.util.Arrays.sort(m);
    int l = 0;
    int r = m.length - 1;
    while (l < r) {
        int sum = m[l] + m[r];
        if (sum == k) {
            result.add("(" + m[l] + " + " + m[r] + ")");
            l++;
            r--;
        } else if (sum < k) {
            l++;
        } else if (sum > k) {
            r--;
        }
    }
    return result;
}
```

Code 10.27

```
public class MinHeap {
    int data;
    int heapIndex;
```

```

int currentIndex;

public MinHeap(int data, int heapIndex,
               int currentIndex) {
    this.data = data;
    this.heapIndex = heapIndex;
    this.currentIndex = currentIndex;
}

}

```

Code 10.28

```

public static int[] merge(int[][][] arrs, int k) {
    // compute the total length of the resulting array
    int len = 0;
    for (int i = 0; i < arrs.length; i++) {
        len += arrs[i].length;
    }
    // create the result array
    int[] result = new int[len];
    // create the min heap
    MinHeap[] heap = new MinHeap[k];
    // add in the heap first element from each array
    for (int i = 0; i < k; i++) {
        heap[i] = new MinHeap(arrs[i][0], i, 0);
    }
    // perform merging
    for (int i = 0; i < result.length; i++) {
        heapify(heap, 0, k);
        // add an element in the final result
        result[i] = heap[0].data;
        heap[0].currentIndex++;
        int[] subarray = arrs[heap[0].heapIndex];

```

```

    if (heap[0].currentIndex >= subarray.length) {
        heap[0].data = Integer.MAX_VALUE;
    } else {
        heap[0].data = subarray[heap[0].currentIndex];
    }
}

return result;
}

```

Code 10.29

```

public static float median(int[] q, int[] p) {
    int lenQ = q.length;
    int lenP = p.length;
    if (lenQ > lenP) {
        swap(q, p);
    }
    int qPointerMin = 0;
    int qPointerMax = q.length;
    int midLength = (q.length + p.length + 1) / 2;
    int qPointer;
    int pPointer;
    while (qPointerMin <= qPointerMax) {
        qPointer = (qPointerMin + qPointerMax) / 2;
        pPointer = midLength - qPointer;
        // perform binary search
        if (qPointer < q.length
            && p[pPointer-1] > q[qPointer]) {
            // qPointer must be increased
            qPointerMin = qPointer + 1;
        } else if (qPointer > 0
            && q[qPointer-1] > p[pPointer]) {

```

```

    // qPointer must be decreased
    qPointerMax = qPointer - 1;

} else { // we found the proper qPointer
    int maxLeft = 0;

    if (qPointer == 0) { // first element on array 'q'?
        maxLeft = p[pPointer - 1];
    } else if (pPointer == 0) { // first
element                                // of array 'p'?

        maxLeft = q[qPointer - 1];
    } else { // we are somewhere in the middle -> find max
        maxLeft = Integer.max(q[qPointer-1], p[pPointer-1]);
    }

    // if the length of 'q' + 'p' arrays is odd,
    // return max of left

    if ((q.length + p.length) % 2 == 1) {

        return maxLeft;
    }

    int minRight = 0;

    if (qPointer == q.length) { // last element on 'q'?
        minRight = p[pPointer];
    } else if (pPointer == p.length) { // last
element                                // on 'p'?

        minRight = q[qPointer];
    } else { // we are somewhere in the middle -> find min
        minRight = Integer.min(q[qPointer], p[pPointer]);
    }

    return (maxLeft + minRight) / 2.0f;
}

}

return -1;
}

```

Code 10.30

```
public static int ofOneOptimized(int[][] matrix) {  
    int maxSubMatrixSize = 1;  
    int rows = matrix.length;  
    int cols = matrix[0].length;  
    int[][] subMatrix = new int[rows][cols];  
    // copy the first row  
    for (int i = 0; i < cols; i++) {  
        subMatrix[0][i] = matrix[0][i];  
    }  
    // copy the first column  
    for (int i = 0; i < rows; i++) {  
        subMatrix[i][0] = matrix[i][0];  
    }  
    // for rest of the matrix check if matrix[i][j]==1  
    for (int i = 1; i < rows; i++) {  
        for (int j = 1; j < cols; j++) {  
            if (matrix[i][j] == 1) {  
                subMatrix[i][j] = Math.min(subMatrix[i - 1][j - 1],  
                    Math.min(subMatrix[i][j - 1],  
                        subMatrix[i - 1][j])) + 1;  
                // compute the maximum of the current sub-matrix  
                maxSubMatrixSize = Math.max(  
                    maxSubMatrixSize, subMatrix[i][j]);  
            }  
        }  
    }  
    return maxSubMatrixSize;  
}
```

Code 10.31

```

public static int maxArea(int[] heights) {

    int maxArea = 0;

    for (int i = 0; i < heights.length; i++) {
        for (int j = i + 1; j < heights.length; j++) {
            // traverse each (i, j) pair
            maxArea = Math.max(maxArea,
                Math.min(heights[i], heights[j]) * (j - i));
        }
    }
    return maxArea;
}

```

Code 10.32

```

public static int maxAreaOptimized(int[] heights) {

    int maxArea = 0;

    int i = 0; // left-hand side pointer
    int j = heights.length - 1; // right-hand side pointer
    // area cannot be negative,
    // therefore i should not be greater than j
    while (i < j) {
        // calculate area for each pair
        maxArea = Math.max(maxArea, Math.min(heights[i],
            heights[j]) * (j - i));
        if (heights[i] <= heights[j]) {
            i++; // left pointer is small than right pointer
        } else {
            j--; // right pointer is small than left pointer
        }
    }
    return maxArea;
}

```

Code 10.34

```
public static int find(int[] m, int x) {  
    int left = 0;  
    int right = m.length - 1;  
    while (left <= right) {  
        // half the search space  
        int middle = (left + right) / 2;  
        // we found the searched value  
        if (m[middle] == x) {  
            return middle;  
        }  
        // check if the right-half is sorted (m[middle] ... right])  
        if (m[middle] <= m[right]) {  
            // check if n is in m[middle] ... right]  
            if (x > m[middle] && x <= m[right]) {  
                left = middle + 1; // search in the right-half  
            } else {  
                right = middle - 1; // search in the left-half  
            }  
        } else { // the left-half is sorted (A[left] ... middle])  
            // check if n is in m[left] ... middle]  
            if (x >= m[left] && x < m[middle]) {  
                right = middle - 1; // search in the left-half  
            } else {  
                left = middle + 1; // search in the right-half  
            }  
        }  
    }  
    return -1;  
}
```

Code 10.34

```
public static void mergeIntervals(Interval[] intervals) {  
    // Step 1  
    java.util.Arrays.sort(intervals,  
        new Comparator<Interval>() {  
            public int compare(Interval i1, Interval i2) {  
                return i1.start - i2.start;  
            }  
        } );  
  
    Stack<Interval> stackOfIntervals = new Stack();  
    for (Interval interval : intervals) {  
        // Step 3a  
        if (stackOfIntervals.empty() || interval.start  
            > stackOfIntervals.peek().end) {  
            stackOfIntervals.push(interval);  
        }  
        // Step 3b  
        if (stackOfIntervals.peek().end < interval.end) {  
            stackOfIntervals.peek().end = interval.end;  
        }  
    }  
    // print the result  
    while (!stackOfIntervals.empty()) {  
        System.out.print(stackOfIntervals.pop() + " ");  
    }  
}
```

Code 10.35

```
public static void mergeIntervals(Interval intervals[]) {  
    // Step 1  
    java.util.Arrays.sort(intervals,
```

```

        new Comparator<Interval>() {
    public int compare(Interval i1, Interval i2) {
        return i2.start - i1.start;
    }
});

int index = 0;
for (int i = 0; i < intervals.length; i++) {
    // Step 2a
    if (index != 0 && intervals[index - 1].start
        <= intervals[i].end) {
        while (index != 0 && intervals[index - 1].start
            <= intervals[i].end) {
            // merge the previous interval with
            // the current interval
            intervals[index - 1].end = Math.max(
                intervals[index - 1].end, intervals[i].end);
            intervals[index - 1].start = Math.min(
                intervals[index - 1].start, intervals[i].start);
            index--;
        }
    }
    // Step 2b
} else {
    intervals[index] = intervals[i];
}
index++;
}

// print the result
for (int i = 0; i < index; i++) {
    System.out.print(intervals[i] + " ");
}

```

```
}
```

Code 10.36

```
public static int circularTour(int[] fuel, int[] dist) {  
    int sumRemainingFuel = 0; // track current remaining fuel  
    int totalFuel = 0; // track total remaining fuel  
    int start = 0;  
  
    for (int i = 0; i < fuel.length; i++) {  
        int remainingFuel = fuel[i] - dist[i];  
        //if sum remaining fuel of (i-1) >= 0 then continue  
        if (sumRemainingFuel >= 0) {  
            sumRemainingFuel += remainingFuel;  
            //otherwise, reset start index to be current  
        } else {  
            sumRemainingFuel = remainingFuel;  
            start = i;  
        }  
        totalFuel += remainingFuel;  
    }  
    if (totalFuel >= 0) {  
        return start;  
    } else {  
        return -1;  
    }  
}
```

Code 10.37

```
// start point 1  
  
int[] dist = {2, 4, 1};  
int[] fuel = {0, 4, 3};  
  
// start point 1  
  
int[] dist = {6, 5, 3, 5};
```

```
int[] fuel = {4, 6, 7, 4};  
// no solution, return -1  
  
int[] dist = {1, 3, 3, 4, 5};  
int[] fuel = {1, 2, 3, 4, 5};  
// start point 2  
  
int[] dist = {4, 6, 6};  
int[] fuel = {6, 3, 7};
```

code 10.38

```
public static int trap(int[] bars) {  
    int n = bars.length - 1;  
  
    int water = 0;  
  
    // store the maximum height of a bar to  
    // the left of the current bar  
  
    int[] left = new int[n];  
    left[0] = Integer.MIN_VALUE;  
  
    // iterate the bars from left to right and  
    // compute each left[i]  
  
    for (int i = 1; i < n; i++) {  
        left[i] = Math.max(left[i - 1], bars[i - 1]);  
    }  
  
    // store the maximum height of a bar to the  
    // right of the current bar  
  
    int right = Integer.MIN_VALUE;  
  
    // iterate the bars from right to left  
    // and compute the trapped water  
  
    for (int i = n - 1; i >= 1; i--) {  
        right = Math.max(right, bars[i + 1]);  
  
        // check if it is possible to store water  
        // in the current bar  
  
        if (Math.min(left[i], right) > bars[i]) {
```

```

        water += Math.min(left[i], right) - bars[i];
    }
}

return water;
}

```

Code 10.39

```

public static int trap(int[] bars) {
    // take two pointers: left and right pointing
    // to 0 and bars.length-1
    int left = 0;
    int right = bars.length - 1;
    int water = 0;
    int maxBarLeft = bars[left];
    int maxBarRight = bars[right];
    while (left < right) {
        // move left pointer to the right
        if (bars[left] <= bars[right]) {
            left++;
            maxBarLeft = Math.max(maxBarLeft, bars[left]);
            water += (maxBarLeft - bars[left]);
        } else {
            right--;
            maxBarRight = Math.max(maxBarRight, bars[right]);
            water += (maxBarRight - bars[right]);
        }
    }
    return water;
}

```

Code 10.40

```

public static int maxProfitOneTransaction(int[] prices) {
    int min = prices[0];
    int result = 0;
    for (int i = 1; i < prices.length; i++) {
        result = Math.max(result, prices[i] - min);
        min = Math.min(min, prices[i]);
    }
    return result;
}

```

Code 10.41

```

public static int maxProfitTwoTransactions(int[] prices) {
    int[] left = new int[prices.length];
    int[] right = new int[prices.length];
    // Dynamic Programming from left to right
    left[0] = 0;
    int min = prices[0];
    for (int i = 1; i < prices.length; i++) {
        min = Math.min(min, prices[i]);
        left[i] = Math.max(left[i - 1], prices[i] - min);
    }
    // Dynamic Programming from right to left
    right[prices.length - 1] = 0;
    int max = prices[prices.length - 1];
    for (int i = prices.length - 2; i >= 0; i--) {
        max = Math.max(max, prices[i]);
        right[i] = Math.max(right[i + 1], max - prices[i]);
    }
    int result = 0;
    for (int i = 0; i < prices.length; i++) {
        result = Math.max(result, left[i] + right[i]);
    }
}

```

```

    }

    return result;
}

```

Code 10.42

```

public static int maxProfitUnlimitedTransactions (
    int[] prices) {

    int result = 0;

    for (int i = 1; i < prices.length; i++) {

        int diff = prices[i] - prices[i - 1];

        if (diff > 0) {

            result += diff;
        }
    }

    return result;
}

```

Code 10.43

```

temp[p] = Math.max(result[p - 1]
                    + Math.max(diff, 0), temp[p] + diff);

result[p] = Math.max(temp[p], result[p]);

```

Code 10.44

```

public static int maxProfitKTransactions (
    int[] prices, int k) {

    int[] temp = new int[k + 1];
    int[] result = new int[k + 1];

    for (int q = 0; q < prices.length - 1; q++) {

        int diff = prices[q + 1] - prices[q];

        for (int p = k; p >= 1; p--) {

            temp[p] = Math.max(result[p - 1]
                                + Math.max(diff, 0), temp[p] + diff);

            result[p] = Math.max(temp[p], result[p]);
        }
    }
}

```

```

        }
    }

    return result[k];
}

```

Code 10.45

```

public static int findLongestConsecutive(int[] sequence) {

    // construct a set from the given sequence
    Set<Integer> sequenceSet = IntStream.of(sequence)
        .boxed()
        .collect(Collectors.toSet());

    int longestSequence = 1;

    for (int elem : sequence) {
        // if 'elem-1' is not in the set then      // start a new sequence
        if (!sequenceSet.contains(elem - 1)) {
            int sequenceLength = 1;
            // lookup in the set for elements
            // 'elem + 1', 'elem + 2', 'elem + 3' ...
            while (sequenceSet.contains(elem + sequenceLength)) {
                sequenceLength++;
            }
            // update the longest consecutive subsequence
            longestSequence = Math.max(
                longestSequence, sequenceLength);
        }
    }
    return longestSequence;
}

```

Code 10.46

```

public static int count(int n) {
    int[] table = new int[n + 1];

```

```
table[0] = 1;

for (int i = 3; i <= n; i++) {
    table[i] += table[i - 3];
}

for (int i = 5; i <= n; i++) {
    table[i] += table[i - 5];
}

for (int i = 10; i <= n; i++) {
    table[i] += table[i - 10];
}

return table[n];
}
```

Code 10.47

```
public static boolean checkDuplicates(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Code 10.48

```
public static boolean checkDuplicates(int[] arr) {
    java.util.Arrays.sort(arr);
    int prev = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] == prev) {
            return true;
        }
    }
}
```

```
        }

        prev = arr[i];

    }

    return false;

}
```

Code 10.49

```
public static boolean checkDuplicates(int[] arr) {

    Set<Integer> set = new HashSet<>();

    for (int i = 0; i < arr.length; i++) {

        if (set.contains(arr[i])) {

            return true;

        }

        set.add(arr[i]);

    }

    return false;

}
```

Code 10.50

```
public static boolean checkDuplicates(int[] arr) {

    for (int i = 0; i < arr.length; i++) {

        if (arr[Math.abs(arr[i])] > 0) {

            arr[Math.abs(arr[i])] = -arr[Math.abs(arr[i])];

        } else if (arr[Math.abs(arr[i])] == 0) {

            arr[Math.abs(arr[i])] = -(arr.length-1);

        } else {

            return true;

        }

    }

    return false;

}
```

Chapter 11

Images

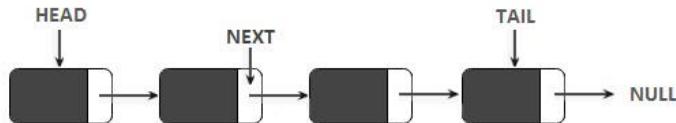


Figure 11.1 – A singly linked list

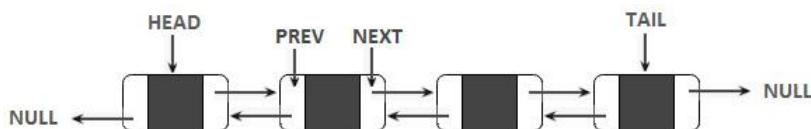


Figure 11.2 – A doubly linked list

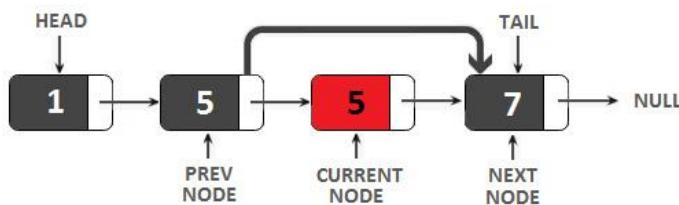


Figure 11.3 – Removing a node from a singly linked list

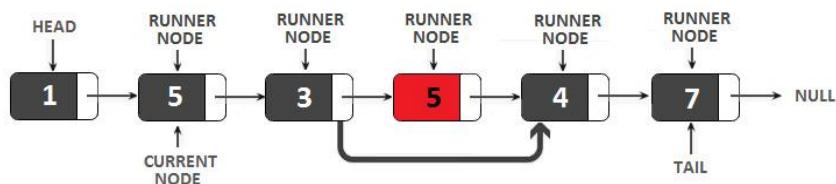


Figure 11.4 – Removing a node from a singly linked list

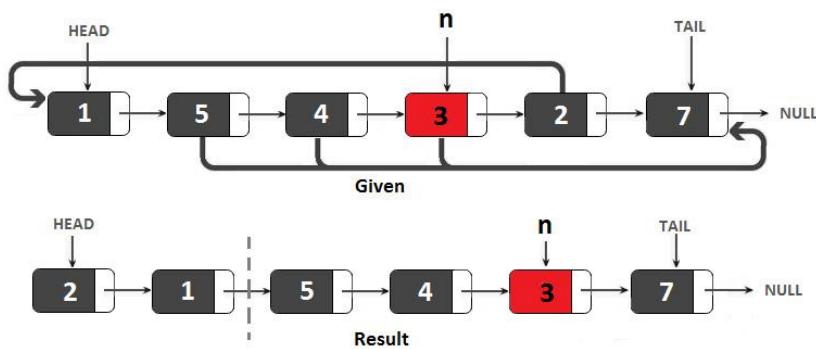


Figure 11.5 – Linked list rearranging

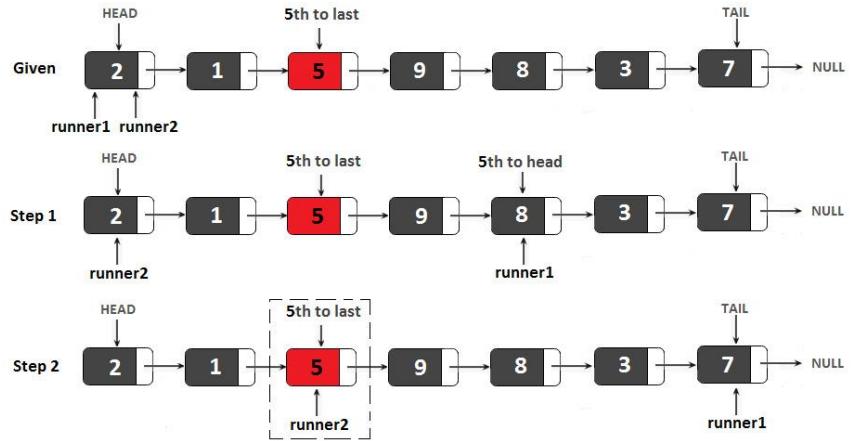


Figure 11.6 – The nth to last node

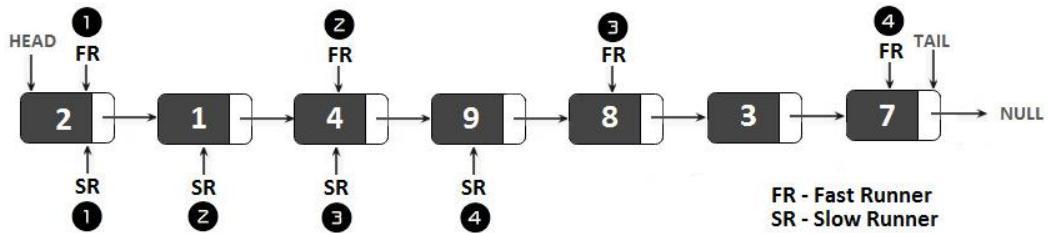


Figure 11.7 – Fast Runner/Slow Runner example

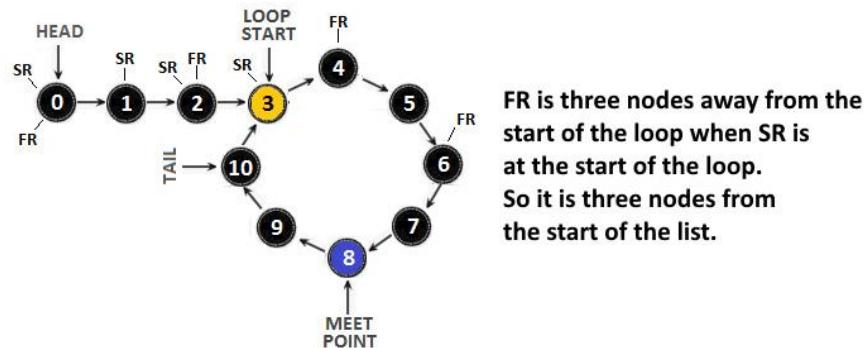


Figure 11.8 – Linked list with a loop

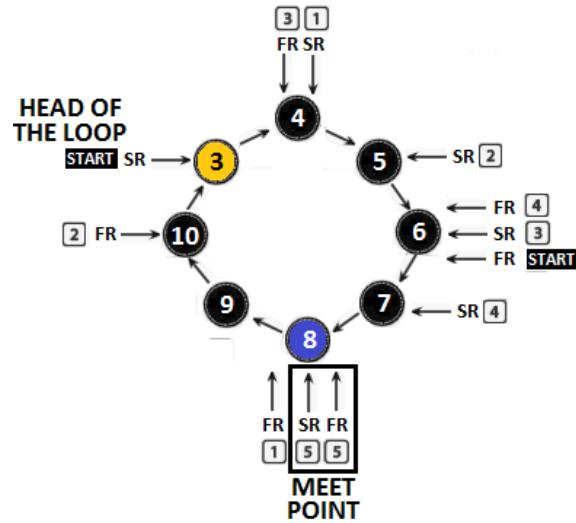


Figure 11.9 – FR and SR collision

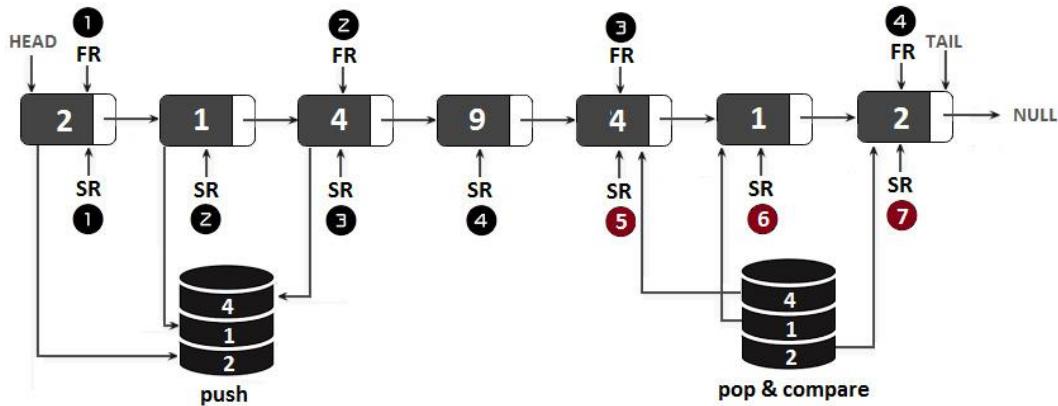


Figure 11.10 – Linked list palindrome using the Fast Runner/Slow Runner approach

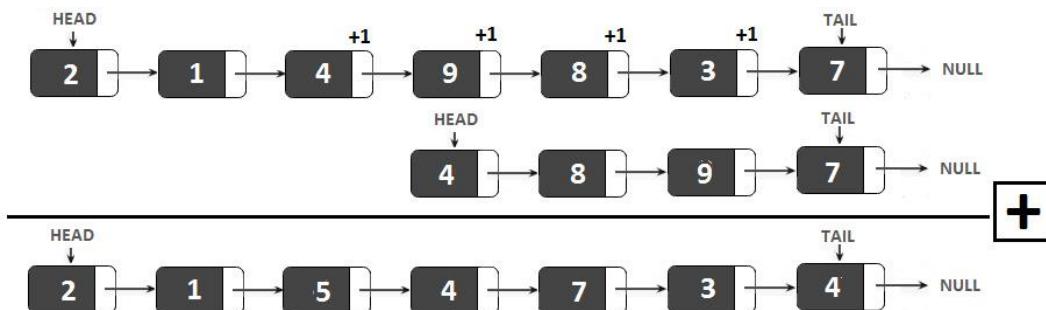


Figure 11.11 – Summing two numbers as linked lists

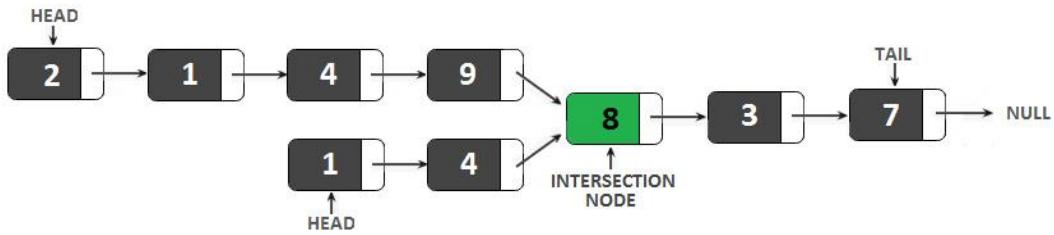


Figure 11.12 – The intersection of two lists

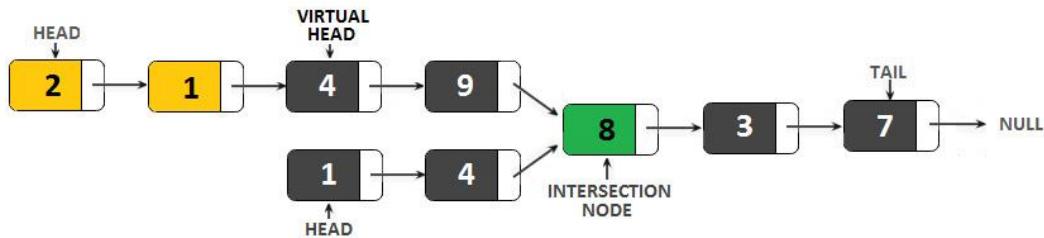


Figure 11.13 – Removing the first two nodes of the top list

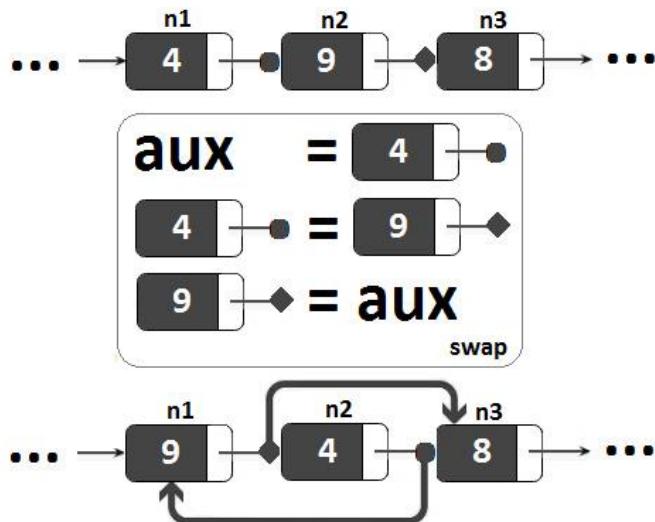


Figure 11.14 – Plain swapping with broken links (1)

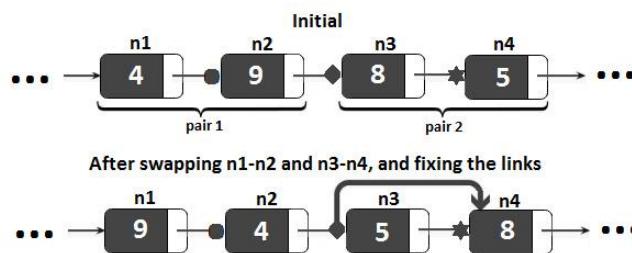


Figure 11.15 – Plain swapping with broken links (2)

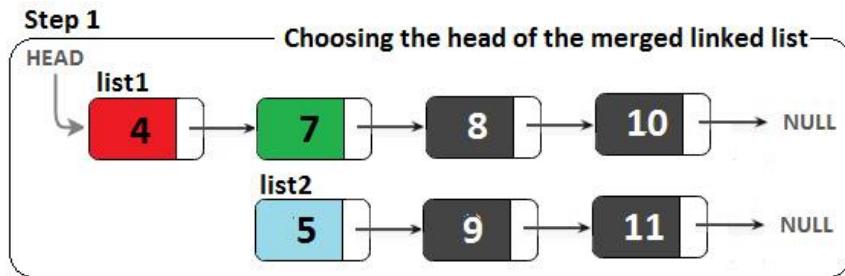


Figure 11.16 – Merging two sorted linked lists (step 1)

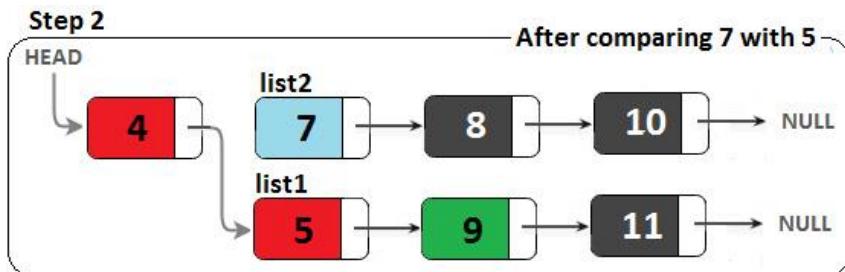


Figure 11.17 – Merging two sorted linked lists (step 2)

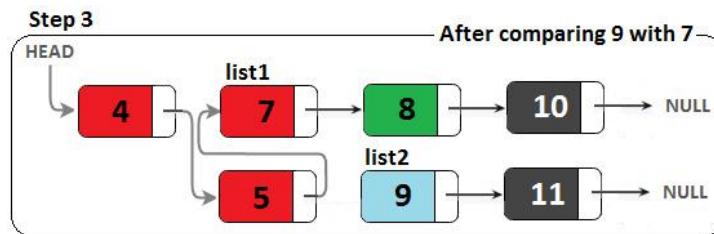


Figure 11.18 – Merging two sorted linked lists (step 3)

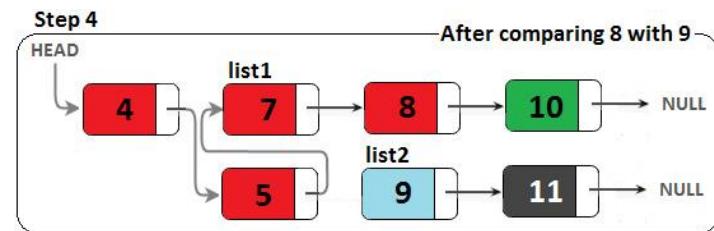


Figure 11.19 – Merging two sorted linked lists (step 4)

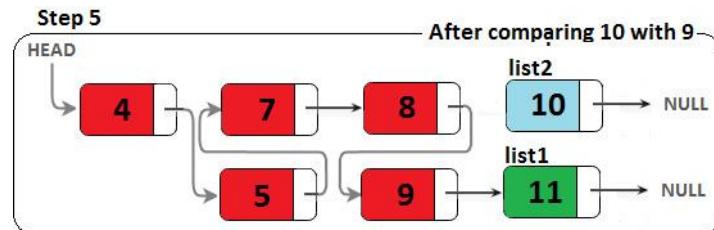


Figure 11.20 – Merging two sorted linked lists (step 5)

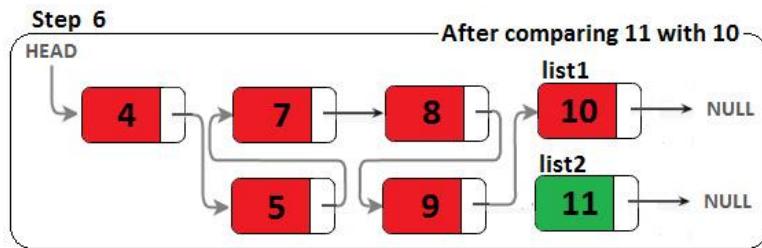


Figure 11.21 – Merging two sorted linked lists (step 6)

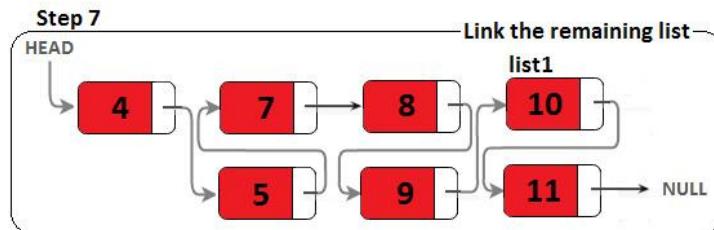


Figure 11.22 – Merging two sorted linked lists (last step)

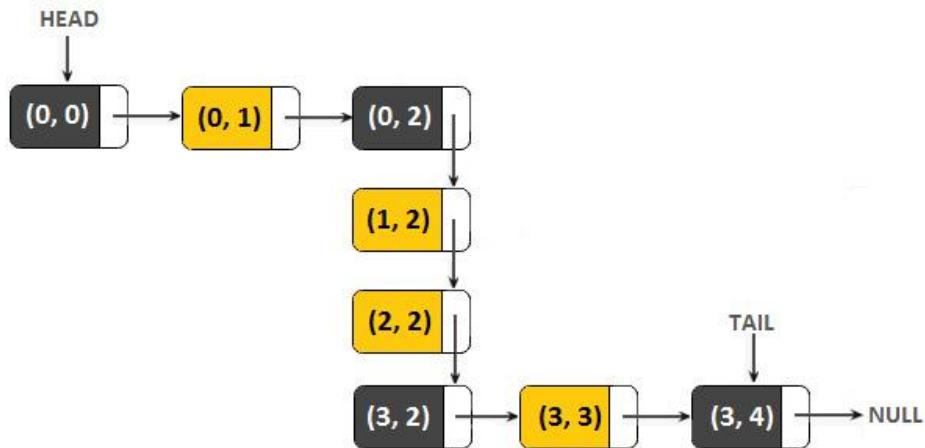


Figure 11.23 – The redundant path

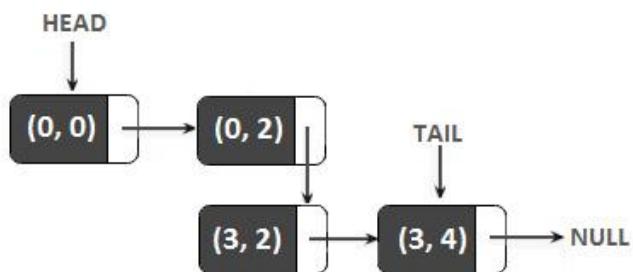


Figure 11.24 – The remaining path after removing the redundancy

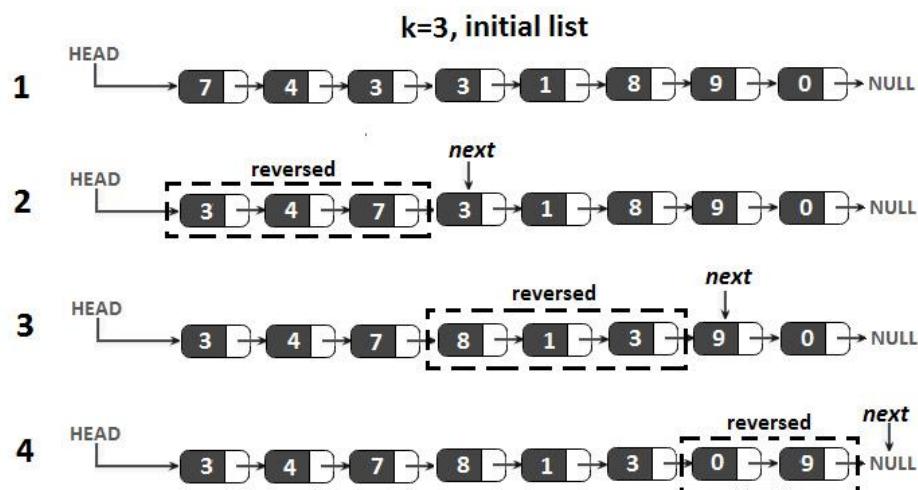


Figure 11.25 – Reversing the list in k groups ($k=3$)

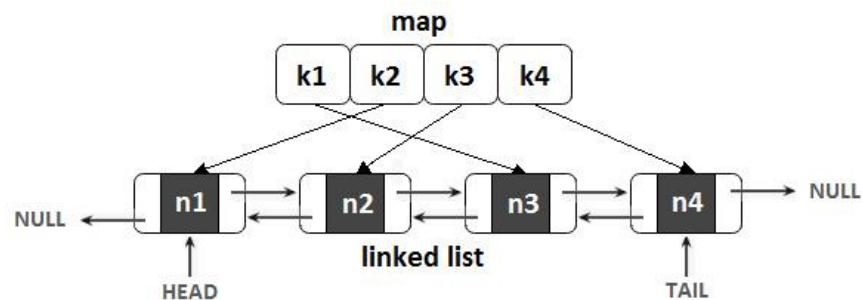


Figure 11.26 – An LRU cache using a HashMap and doubly linked list

Code

Code 11.1

```
private final class Node {
    private int data;
    private Node next;
}
```

Code 11.2

```
private final class Node {
    private int data;
    private Node next;
    private Node prev;
}
```

Code 11.3

```
private final class MyEntry<K, V> {  
    private final K key;  
    private V value;  
    public MyEntry(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
    // getters and setters omitted for brevity  
}
```

Code 11.4

```
private static final int DEFAULT_CAPACITY = 16;  
private MyEntry<K, V>[] entries  
    = new MyEntry[DEFAULT_CAPACITY];
```

Code 11.5

```
private int size;  
public void put(K key, V value) {  
    boolean success = true;  
    for (int i = 0; i < size; i++) {  
        if (entries[i].getKey().equals(key)) {  
            entries[i].setValue(value);  
            success = false;  
        }  
    }  
    if (success) {  
        checkCapacity();  
        entries[size++] = new MyEntry<>(key, value);  
    }  
}
```

Code 11.6

```
private void checkCapacity() {  
    if (size == entries.length) {  
        int newSize = entries.length * 2;  
        entries = Arrays.copyOf(entries, newSize);  
    }  
}
```

Code 11.7

```
public V get(K key) {  
    for (int i = 0; i < size; i++) {  
        if (entries[i] != null) {  
            if (entries[i].getKey().equals(key)) {  
                return entries[i].getValue();  
            }  
        }  
    }  
    return null;  
}
```

Code 11.8

```
public void remove(K key) {  
    for (int i = 0; i < size; i++) {  
        if (entries[i].getKey().equals(key)) {  
            entries[i] = null;  
            size--;  
            condenseArray(i);  
        }  
    }  
}  
  
private void condenseArray(int start) {  
    int i;  
    for (i = start; i < size; i++) {
```

```
    entries[i] = entries[i + 1];  
}  
  
entries[i] = null; // don't forget this line  
}
```

Code 11.9

```
public Set<K> keySet() {  
  
    Set<K> set = new HashSet<>();  
  
    for (int i = 0; i < size; i++) {  
  
        set.add(entries[i].getKey());  
    }  
  
    return set;  
}
```

Code 11.10

```
public Collection<V> values() {  
  
    List<V> list = new ArrayList<>();  
  
    for (int i = 0; i < size; i++) {  
  
        list.add(entries[i].getValue());  
    }  
  
    return list;  
}
```

Code 11.11

```
char[] nuts = {'$', '%', '&', 'x', '@'};  
char[] bolts = {'%', '@', 'x', '$', '&'};
```

Code 11.12

```
public static void match(char[] nuts, char[] bolts) {  
  
    // in this map, each nut is a key and  
    // its position is as value  
  
    Map<Character, Integer> map = new HashMap<>();  
  
    for (int i = 0; i < nuts.length; i++) {  
  
        map.put(nuts[i], i);  
    }
```

```

}

//for each bolt, search a nut

for (int i = 0; i < bolts.length; i++) {

    char bolt = bolts[i];

    if (map.containsKey(bolt)) {

        nuts[i] = bolts[i];

    } else {

        System.out.println("Bolt " + bolt + " has no nut");

    }

}

System.out.println("Matches between nuts and bolts: ");

System.out.println("Nuts: " + Arrays.toString(nuts));

System.out.println("Bolts: " + Arrays.toString(bolts));

}

```

Code 11.13

```

// 'size' is the linked list size

public void removeDuplicates() {

    Set<Integer> dataSet = new HashSet<>();

    Node currentNode = head;

    Node prevNode = null;

    while (currentNode != null) {

        if (dataSet.contains(currentNode.data)) {

            prevNode.next = currentNode.next;

            if (currentNode == tail) {

                tail = prevNode;

            }

            size--;

        } else {

            dataSet.add(currentNode.data);

            prevNode = currentNode;

        }

    }

}

```

```
        }

        currentNode = currentNode.next;

    }

}
```

Code 11.14

```
public void removeDuplicates() {

    Node currentNode = head;

    while (currentNode != null) {

        Node runnerNode = currentNode;

        while (runnerNode.next != null) {

            if (runnerNode.next.data == currentNode.data) {

                if (runnerNode.next == tail) {

                    tail = runnerNode;

                }

                runnerNode.next = runnerNode.next.next;

                size--;

            } else {

                runnerNode = runnerNode.next;

            }

        }

        currentNode = currentNode.next;

    }

}
```

Code 11.15

```
public void rearrange(int n) {

    Node currentNode = head;

    head = currentNode;

    tail = currentNode;

    while (currentNode != null) {

        Node nextNode = currentNode.next;
```

```

if (currentNode.data < n) {
    // insert node at the head
    currentNode.next = head;
    head = currentNode;
} else {
    // insert node at the tail
    tail.next = currentNode;
    tail = currentNode;
}
currentNode = nextNode;
}
tail.next = null;
}

```

Code 11.16

```

public int nthToLastIterative(int n) {
    // both runners are set to the start
    Node firstRunner = head;
    Node secondRunner = head;
    // runner1 goes in the nth position
    for (int i = 0; i < n; i++) {
        if (firstRunner == null) {
            throw new IllegalArgumentException(
                "The given n index is out of bounds");
        }
        firstRunner = firstRunner.next;
    }
    // runner2 run as long as runner1 is not null
    // basically, when runner1 cannot run further (is null),
    // runner2 will be placed on the nth to last node
    while (firstRunner != null) {

```

```

        firstRunner = firstRunner.next;
        secondRunner = secondRunner.next;
    }
    return secondRunner.data;
}

```

Code 11.17

```

public void findLoopStartNode() {
    Node slowRunner = head;
    Node fastRunner = head;
    // fastRunner meets slowRunner
    while (fastRunner != null && fastRunner.next != null) {
        slowRunner = slowRunner.next;
        fastRunner = fastRunner.next.next;
        if (slowRunner == fastRunner) { // they met
            System.out.println("\nThe meet point is at
                the node with value: " + slowRunner);
            break;
        }
    }
    // if no meeting point was found then there is no loop
    if (fastRunner == null || fastRunner.next == null) {
        return;
    }
    // the slowRunner moves to the head of the linked list
    // the fastRunner remains at the meeting point
    // they move simultaneously node-by-node and
    // they should meet at the loop start
    slowRunner = head;
    while (slowRunner != fastRunner) {
        slowRunner = slowRunner.next;
    }
}

```

```

        fastRunner = fastRunner.next;
    }

    // both pointers points to the start of the loop
    System.out.println("\nLoop start detected at
        the node with value: " + fastRunner);
}

```

Code 11.18

```

public boolean isPalindrome() {
    Node fastRunner = head;
    Node slowRunner = head;
    Stack<Integer> firstHalf = new Stack<>();
    // the first half of the linked list is added into the stack
    while (fastRunner != null && fastRunner.next != null) {
        firstHalf.push(slowRunner.data);
        slowRunner = slowRunner.next;
        fastRunner = fastRunner.next.next;
    }
    // for odd number of elements we to skip the middle node
    if (fastRunner != null) {
        slowRunner = slowRunner.next;
    }
    // pop from the stack and compare with the node by node of
    // the second half of the linked list
    while (slowRunner != null) {
        int top = firstHalf.pop();
        // a mismatch means that the list is not a palindrome
        if (top != slowRunner.data) {
            return false;
        }
        slowRunner = slowRunner.next;
    }
}

```

```
    }

    return true;
}
```

Code 11.19

```
private Node sum(Node node1, Node node2, int carry) {
    if (node1 == null && node2 == null && carry == 0) {
        return null;
    }

    Node resultNode = new Node();
    int value = carry;
    if (node1 != null) {
        value += node1.data;
    }
    if (node2 != null) {
        value += node2.data;
    }
    resultNode.data = value % 10;
    if (node1 != null || node2 != null) {
        Node more = sum(node1 == null
            ? null : node1.next, node2 == null
            ? null : node2.next, value >= 10 ? 1 : 0);
        resultNode.next = more;
    }
    return resultNode;
}
```

Code 11.20

```
public int intersection() {
    // this is the head of first list
    Node currentNode1 = {head_of_first_list};
    // this is the head of the second list
```

```

Node currentNode2 = {head_of_second_list};

// compute the size of both linked lists
// linkedListSize() is just a helper method

int s1 = linkedListSize(currentNode1);
int s2 = linkedListSize(currentNode2);

// the first linked list is longer than the second one

if (s1 > s2) {

    for (int i = 0; i < (s1 - s2); i++) {

        currentNode1 = currentNode1.next;
    }
} else {

    // the second linked list is longer than the first one

    for (int i = 0; i < (s2 - s1); i++) {

        currentNode2 = currentNode2.next;
    }
}

// iterate both lists until the end or the intersection node

while (currentNode1 != null && currentNode2 != null) {

    // we compare references not values!

    if (currentNode1 == currentNode2) {

        return currentNode1.data;
    }

    currentNode1 = currentNode1.next;
    currentNode2 = currentNode2.next;
}
return -1;
}

```

Code 11.21

```

public void swap() {

    if (head == null || head.next == null) {

```

```
    return;
}

Node currentNode = head;
Node prevPair = null;

// consider two nodes at a time and swap their links
while (currentNode != null && currentNode.next != null) {

    Node node1 = currentNode;                      // first node
    Node node2 = currentNode.next;                 // second
node
    Node node3 = currentNode.next.next; // third node

    // swap node1 node2
    Node auxNode = node1;
    node1 = node2;
    node2 = auxNode;

    // repair the links broken by swapping
    node1.next = node2;
    node2.next = node3;

    // if we are at the first swap we set the head
    if (prevPair == null) {
        head = node1;
    } else {
        // we link the previous pair to this pair
        prevPair.next = node1;
    }

    // there are no more nodes, therefore set the tail
    if (currentNode.next == null) {
        tail = currentNode;
    }

    // prepare the prevNode of the current pair
    prevPair = node2;
    // advance to the next pair
}
```

```
    currentNode = node3;
}
}
```

Code 11.22

```
Node auxNode = list1.next; // auxNode = node with value 7
list1.next = list2;           // list1.next = node with value 5
list2 = auxNode;             // list2 = node with value 7
```

Code 11.23

```
public void merge(SinglyLinkedList sll) {
    // these are the two lists
    Node list1 = head;          // the merged linked list
    Node list2 = sll.head;       // from this list we add nodes at
                                // appropriate place in list1
    // compare heads and swap them if it is necessary
    if (list1.data < list2.data) {
        head = list1;
    } else {
        head = list2;
        list2 = list1;
        list1 = head;
    }
    // compare the nodes from list1 with the nodes from list2
    while (list1.next != null) {
        if (list1.next.data > list2.data) {
            Node auxNode = list1.next;
            list1.next = list2;
            list2 = auxNode;
        }
        // advance to the last node in the merged linked
        list
    }
}
```

```

        list1 = list1.next;
    }

    // add the remaining list2
    if (list1.next == null) {
        list1.next = list2;
    }
}

```

Code 11.24

```

public void removeRedundantPath() {

    Node currentNode = head;

    while (currentNode.next != null
            && currentNode.next.next != null) {

        Node middleNode = currentNode.next.next;

        // check for a vertical triplet (triplet with same column)
        if (currentNode.c == currentNode.next.c
                && currentNode.c == middleNode.c) {

            // delete the middle node
            currentNode.next = middleNode;
        } // check for a horizontal triplet
        else if (currentNode.r == currentNode.next.r
                && currentNode.r == middleNode.r) {

            // delete the middle node
            currentNode.next = middleNode;
        } else {
            currentNode = currentNode.next;
        }
    }
}

```

Code 11.25

```

public void moveLastToFront() {

```

```
Node currentNode = head;  
// step 1  
while (currentNode.next.next != null) {  
    currentNode = currentNode.next;  
}  
// step 2  
Node nextNode = currentNode.next;  
// step 3  
currentNode.next = null;  
// step 4  
nextNode.next = head;  
head = nextNode;  
}
```

Code 11.26

```
public void moveLastToFront() {  
    Node currentNode = head;  
    // step 1  
    while (currentNode.next.next != null) {  
        currentNode = currentNode.next;  
    }  
    // step 2  
    currentNode.next.next = head;  
    // step 3  
    head = currentNode.next;  
    // step 4  
    currentNode.next = null;  
}
```

Code 11.27

```
public void reverseInKGroups(int k) {  
    if (head != null) {
```

```

        head = reverseInKGroups(head, k);
    }
}

private Node reverseInKGroups(Node head, int k) {
    Node current = head;
    Node next = null;
    Node prev = null;
    int counter = 0;

    // reverse first 'k' nodes of linked list
    while (current != null && counter < k) {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next;
        counter++;
    }

    // 'next' points to (k+1)th node
    if (next != null) {
        head.next = reverseInKGroups(next, k);
    }

    // 'prev' is now the head of the input list
    return prev;
}

```

Code 11.28

```

public void reverse() {
    Node currentNode = head;
    Node prevNode = null;
    while (currentNode != null) {
        // swap next and prev pointers of the current node
        Node prev = currentNode.prev;

```

```

        currentNode.prev = currentNode.next;
        currentNode.next = prev;
        // update the previous node before moving to the next node
        prevNode = currentNode;
        // move to the next node in the doubly linked list
        currentNode = currentNode.prev;
    }
    // update the head to point to the last node
    if (prevNode != null) {
        head = prevNode;
    }
}

```

Code 11.30

```

public final class LRUCache {

    private final class Node {
        private int key;
        private int value;
        private Node next;
        private Node prev;
    }

    private final Map<Integer, Node> hashmap;
    private Node head;
    private Node tail;
    // 5 is the maximum size of the cache
    private static final int LRU_SIZE = 5;
    public LRUCache() {
        hashmap = new HashMap<>();
    }

    public int getEntry(int key) {
        Node node = hashmap.get(key);

```

```
// if the key already exist then update its usage in cache
if (node != null) {
    removeNode(node);
    addNode(node);
    return node.value;
}

// by convention, data not found is marked as -1
return -1;
}

public void putEntry(int key, int value) {
    Node node = hashmap.get(key);
    // if the key already exist then update
    // the value and move it to top of the cache
    if (node != null) {
        node.value = value;
        removeNode(node);
        addNode(node);
    } else {
        // this is new key
        Node newNode = new Node();
        newNode.prev = null;
        newNode.next = null;
        newNode.value = value;
        newNode.key = key;
        // if we reached the maximum size of the cache then
        // we have to remove the Least Recently Used
        if (hashmap.size() >= LRU_SIZE) {
            hashmap.remove(tail.key);
            removeNode(tail);
            addNode(newNode);
        }
    }
}
```

```
    } else {
        addNode(newNode);
    }
    hashmap.put(key, newNode);
}
}

// helper method to add a node to the top of the cache
private void addNode(Node node) {
    node.next = head;
    node.prev = null;
    if (head != null) {
        head.prev = node;
    }
    head = node;
    if (tail == null) {
        tail = head;
    }
}

// helper method to remove a node from the cache
private void removeNode(Node node) {
    if (node.prev != null) {
        node.prev.next = node.next;
    } else {
        head = node.next;
    }
    if (node.next != null) {
        node.next.prev = node.prev;
    } else {
        tail = node.prev;
    }
}
```

```
    }  
}
```

Chapter 12

Images

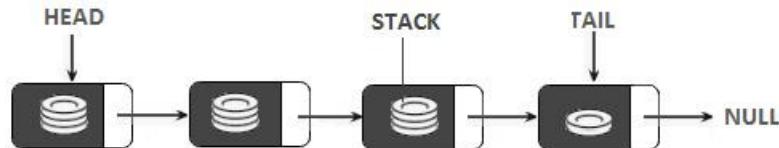


Figure 12.1 – Linked list of stacks

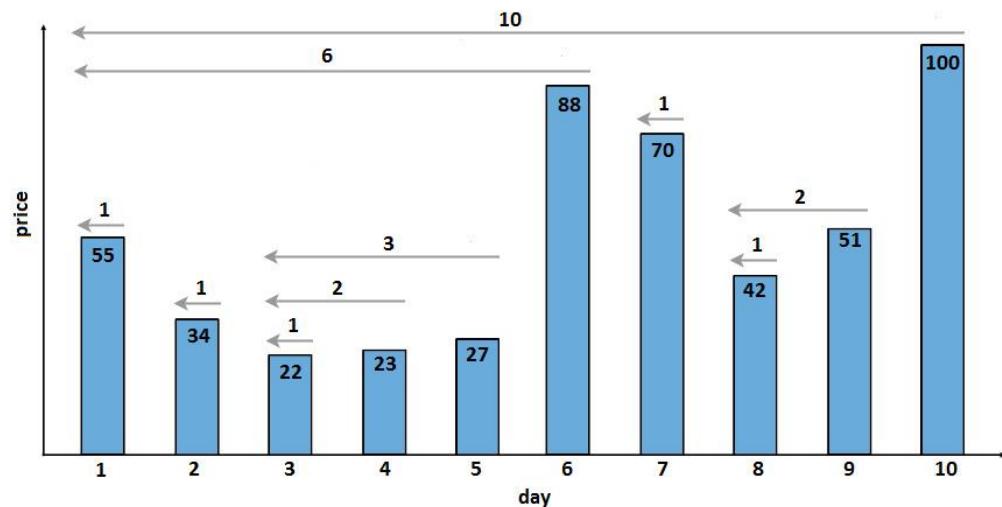


Figure 12.2 – Stock span for 10 days

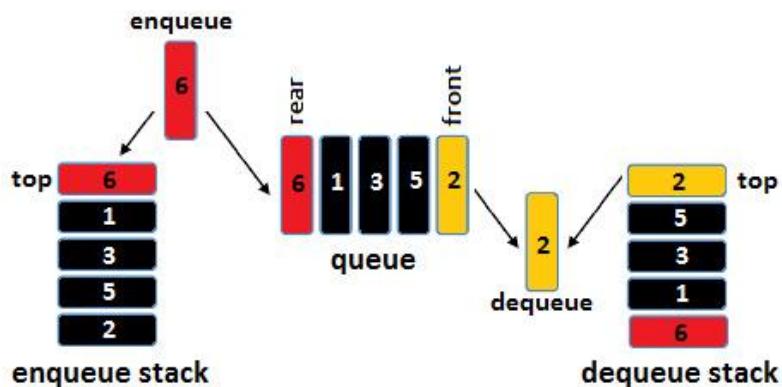


Figure 12.3 – Cue via two stacks

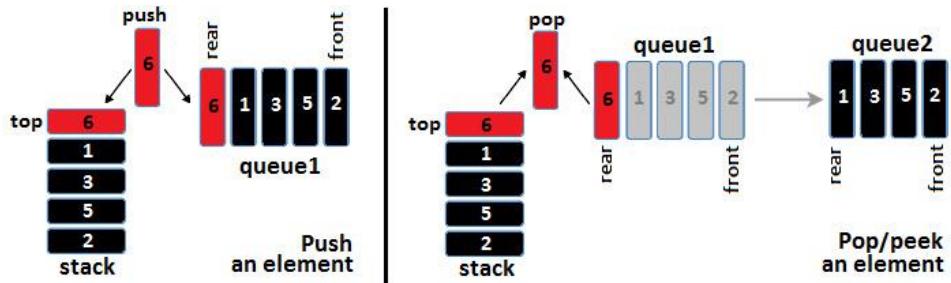


Figure 12.4 – Stack via two queues

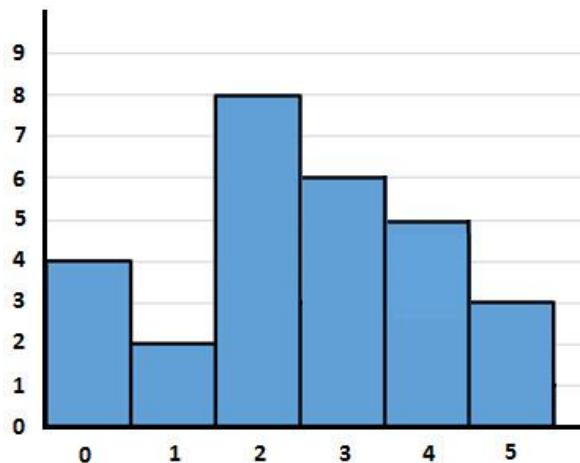


Figure 12.5 – Histogram with the class interval equal to 1

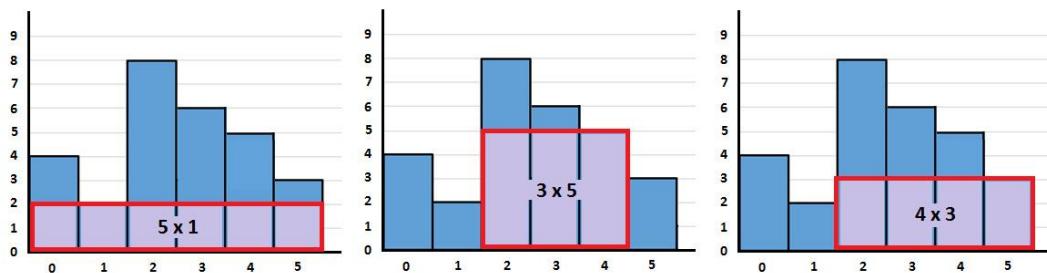


Figure 12.6 – Rectangles of a histogram

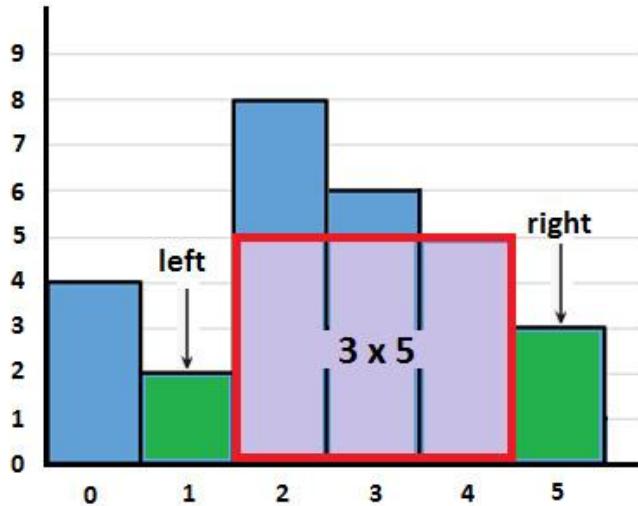


Figure 12.7 – Left and right boundaries

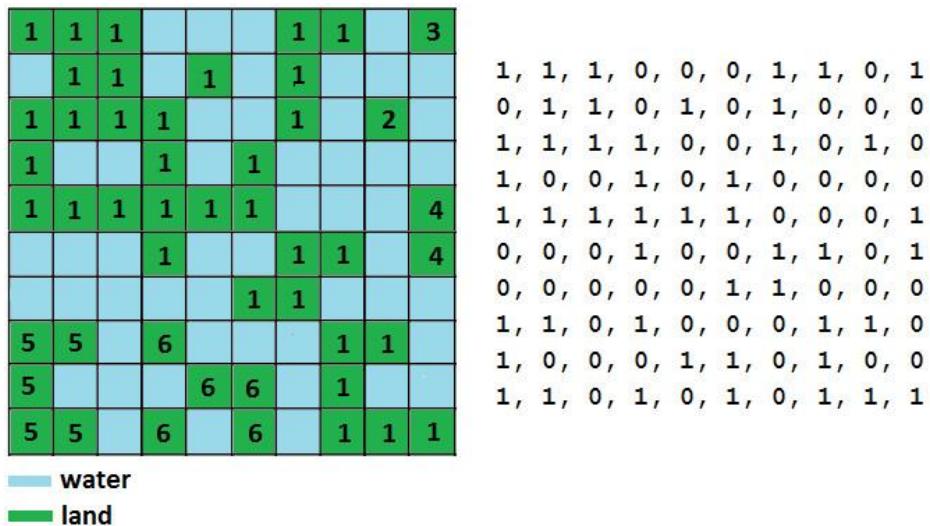


Figure 12.8 – Islands via a 10x10 matrix

	0	1	2	3	4	5	6	7	8	9
0	0	1	1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
2	1	1	0	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	0	1	1	0	1
5	1	1	1	1	1	1	1	1	1	1
6	1	0	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	0
8	1	1	1	1	1	0	1	1	1	1
9	1	1	1	1	1	1	1	1	1	1

	0	1	2	3	4	5	6	7	8	9
0	0	0	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1
2	1	0	0	0	1	1	1	1	1	1
3	1	0	0	0	0	0	0	0	0	0
4	1 ⁰	1 ¹	1 ²	1 ³	0	0	0	0	0	0
5	0	0	0	1 ⁴	0	0	0	0	0	0
6	0	0	0	1 ⁵	1 ⁶	1 ⁷	1 ⁸	1 ⁹	1 ¹⁰	0
7	0	0	0	1 ¹¹	0	0	0	1 ¹²	0	0
8	1	1	1	1 ¹³	0	0	0	1 ¹⁴	0	0
9	1	1	1	1	0	0	0	1	1	1

Shortest path: 15

Figure 12.9 – The given matrix (left-hand side) and the resolved matrix (right-hand side)

Infix	Postfix	Prefix
(a + b) * c	a b + c *	* + a b c
a + (b * c)	a b c * +	+ a * b c

Figure 12.10 – Infix, postfix, and prefix

Code

Code 12.1

```
public final class MyStack<E> {

    private static final int DEFAULT_CAPACITY = 10;

    private int top;
    private E[] stack;

    MyStack() {
        stack = (E[]) Array.newInstance(
            Object[].class.getComponentType(),
            DEFAULT_CAPACITY);
        top = 0; // the initial size is 0
    }

    public void push(E e) {}

    public E pop() {}

    public E peek() {}
}
```

```
public int size() {}

public boolean isEmpty() {}

public boolean isFull() {}

private void ensureCapacity() {}

}
```

Code 12.2

```
// add an element 'e' in the stack

public void push(E e) {

    // if the stack is full, we double its capacity
    if (isFull()) {

        ensureCapacity();

    }

    // adding the element at the top of the stack
    stack[top++] = e;

}

// used internally for doubling the stack capacity

private void ensureCapacity() {

    int newSize = stack.length * 2;

    stack = Arrays.copyOf(stack, newSize);

}
```

Code 12.3

```
// pop top element from the stack

public E pop() {

    // if the stack is empty then just throw an exception
    if (isEmpty()) {

        throw new EmptyStackException();

    }

    // extract the top element from the stack
    E e = stack[--top];

    // avoid memory leaks
```

```
    stack[top] = null;
    return e;
}
```

Code 12.4

```
// return but not remove the top element in the stack
public E peek() {
    // if the stack is empty then just throw an exception
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    return stack[top - 1];
}
```

Code 12.5

```
public final class MyQueue<E> {
    private static final int DEFAULT_CAPACITY = 10;
    private int front;
    private int rear;
    private int count;
    private int capacity;
    private E[] cue;
    MyQueue() {
        cue = (E[]) Array.newInstance(
            Object[].class.getComponentType(),
            DEFAULT_CAPACITY);
        count = 0; // the initial size is 0
        front = 0;
        rear = -1;
        capacity = DEFAULT_CAPACITY;
    }
    public void enqueue(E e) { }
```

```

public E dequeue() {}

public E peek() {}

public int size() {}

public boolean isEmpty() {}

public boolean isFull() {}

private void ensureCapacity() {}

}

```

Code 12.6

```

// add an element 'e' in the cue

public void enqueue(E e) {

    // if the cue is full, we double its capacity
    if (isFull()) {

        ensureCapacity();

    }

    // adding the element in the rear of the cue
    rear = (rear + 1) % capacity;
    cue[rear] = e;

    // update the size of the cue
    count++;

}

// used internally for doubling the cue capacity

private void ensureCapacity() {

    int newSize = cue.length * 2;

    cue = Arrays.copyOf(cue, newSize);

    // setting the new capacity
    capacity = newSize;

}

```

Code 12.7

```

// remove and return the front element from the cue

public E dequeue() {

```

```

// if the cue is empty we just throw an exception
if (isEmpty()) {
    throw new EmptyStackException();
}

// extract the element from the front
E e = cue[front];
cue[front] = null;

// set the new front
front = (front + 1) % capacity;
// decrease the size of the cue
count--;
return e;
}

```

Code 12.8

```

// return but not remove the front element in the cue
public E peek() {
    // if the cue is empty we just throw an exception
    if (isEmpty()) {
        throw new EmptyStackException();
    }
    return cue[front];
}

```

Code 12.9

```

public static String reverse(String str) {
    Stack<Character> stack = new Stack();
    // push characters of the string into the stack
    char[] chars = str.toCharArray();
    for (char c : chars) {
        stack.push(c);
    }
}

```

```

// pop all characters from the stack and
// put them back to the input string
for (int i = 0; i < str.length(); i++) {
    chars[i] = stack.pop();
}
// return the string
return new String(chars);
}

```

Code 12.10

```

public static boolean bracesMatching(String bracesStr) {
    Stack<Character> stackBraces = new Stack<>();
    int len = bracesStr.length();
    for (int i = 0; i < len; i++) {
        switch (bracesStr.charAt(i)) {
            case '{':
                stackBraces.push(bracesStr.charAt(i));
                break;
            case '}':
                if (stackBraces.isEmpty()) { // we found a mismatch
                    return false;
                }
                // for every match we pop the corresponding '{'
                stackBraces.pop();
                break;
            default:
                return false;
        }
    }
    return stackBraces.empty();
}

```

Code 12.11

```
private static final int STACK_SIZE = 3;

private final LinkedList<Stack<Integer>> stacks
    = new LinkedList<>();

public void push(int value) {
    // if there is no stack or the last stack is full
    if (stacks.isEmpty() || stacks.getLast().size()
        >= STACK_SIZE) {
        // create a new stack and push the value into it
        Stack<Integer> stack = new Stack<>();
        stack.push(value);
        // add the new stack into the list of stacks
        stacks.add(stack);
    } else {
        // add the value in the last stack
        stacks.getLast().push(value);
    }
}
```

Code 12.12

```
public Integer pop() {
    // find the last stack
    Stack<Integer> lastStack = stacks.getLast();
    // pop the value from the last stack
    int value = lastStack.pop();
    // if last stack is empty, remove it from the list of stacks
    removeStackIfEmpty();
    return value;
}

private void removeStackIfEmpty() {
    if (stacks.getLast().isEmpty()) {
```

```

        stacks.removeLast();
    }
}

```

Code 12.13

```

public Integer popAt(int stackIndex) {
    // get the value from the correspondind stack
    int value = stacks.get(stackIndex).pop();
    // pop an element -> must shift the remaining elements
    shift(stackIndex);
    // if last stack is empty, remove it from the list of stacks
    removeStackIfEmpty();
    return value;
}

private void shift(int index) {
    for (int i = index; i < stacks.size() - 1; ++i) {
        Stack<Integer> currentStack = stacks.get(i);
        Stack<Integer> nextStack = stacks.get(i + 1);
        currentStack.push(nextStack.remove(0));
    }
}

```

Code 12.14

```

public static int[] stockSpan(int[] stockPrices) {
    Stack<Integer> dayStack = new Stack();
    int[] spanResult = new int[stockPrices.length];
    spanResult[0] = 1; // first day has span 1
    dayStack.push(0);
    for (int i = 1; i < stockPrices.length; i++) {
        // pop until we find a price on stack which is
        // greater than the current day's price or there
        // are no more days left
    }
}

```

```

while (!dayStack.empty())
    && stockPrices[i] > stockPrices[dayStack.peek()])
{
    dayStack.pop();
}

// if there is no price greater than the current
// day's price then the stock span is the numbers of days
if (dayStack.empty()) {
    spanResult[i] = i + 1;
} else {
    // if there is a price greater than the current
    // day's price then the stock span is the
    // difference between the current day and that day
    spanResult[i] = i - dayStack.peek();
}
// push current day onto top of stack
dayStack.push(i);
}

return spanResult;
}

```

Code 12.15

```

public class MyStack extends Stack<Integer> {
    Stack<Integer> stackOfMin;
    public MyStack() {
        stackOfMin = new Stack<>();
    }
    public Integer push(int value) {
        if (value <= min()) {
            stackOfMin.push(value);
        }
        return super.push(value);
    }
}
```

```

    }

@Override

public Integer pop() {
    int value = super.pop();
    if (value == min()) {
        stackOfMin.pop();
    }
    return value;
}

public int min() {
    if (stackOfMin.isEmpty()) {
        return Integer.MAX_VALUE;
    } else {
        return stackOfMin.peek();
    }
}
}

```

Code 12.16

```

public class MyStack {

    private int min;

    private final Stack<Integer> stack = new Stack<>();

    public void push(int value) {
        // we don't allow values that overflow int/2 range
        int r = Math.addExact(value, value);
        if (stack.empty()) {
            stack.push(value);
            min = value;
        } else if (value > min) {
            stack.push(value);
        } else {

```

```

        stack.push(r - min);

        min = value;

    }

}

// pop() doesn't return the value since this may be a wrong
// value (a value that was not pushed by the client)!

public void pop() {

    if (stack.empty()) {

        throw new EmptyStackException();

    }

    int top = stack.peek();

    if (top < min) {

        min = 2 * min - top;

    }

    stack.pop();

}

public int min() {

    return min;

}

}

```

Code 12.17

```

public class MyQueueViaStack<E> {

    private final Stack<E> stackEnqueue;

    private final Stack<E> stackDequeue;

    public MyQueueViaStack() {

        stackEnqueue = new Stack<>();

        stackDequeue = new Stack<>();

    }

    public void enqueue(E e) {

        stackEnqueue.push(e);

```

```

    }

    public E dequeue() {
        reverseStackEnqueue();
        return stackDequeue.pop();
    }

    public E peek() {
        reverseStackEnqueue();
        return stackDequeue.peek();
    }

    public int size() {
        return stackEnqueue.size() + stackDequeue.size();
    }

    private void reverseStackEnqueue() {
        if (stackDequeue.isEmpty()) {
            while (!stackEnqueue.isEmpty()) {
                stackDequeue.push(stackEnqueue.pop());
            }
        }
    }
}

```

Code 12.18

```

public class MyStackViaQueue<E> {

    private final Queue<E> queue1;
    private final Queue<E> queue2;
    private E peek;
    private int size;

    public MyStackViaQueue() {
        queue1 = new ArrayDeque<>();
        queue2 = new ArrayDeque<>();
    }
}

```

```
public void push(E e) {
    if (!queue1.isEmpty()) {
        if (peek != null) {
            queue1.add(peek);
        }
        queue1.add(e);
    } else {
        if (peek != null) {
            queue2.add(peek);
        }
        queue2.add(e);
    }
    size++;
    peek = null;
}

public E pop() {
    if (size() == 0) {
        throw new EmptyStackException();
    }
    if (peek != null) {
        E e = peek;
        peek = null;
        size--;
        return e;
    }
    E e;
    if (!queue1.isEmpty()) {
        e = switchQueue(queue1, queue2);
    } else {
        e = switchQueue(queue2, queue1);
    }
}
```

```

        }

        size--;
        return e;
    }

    public E peek() {
        if (size() == 0) {
            throw new EmptyStackException();
        }

        if (peek == null) {
            if (!queue1.isEmpty()) {
                peek = switchQueue(queue1, queue2);
            } else {
                peek = switchQueue(queue2, queue1);
            }
        }

        return peek;
    }

    public int size() {
        return size;
    }

    private E switchQueue(Cue from, Cue to) {
        while (from.size() > 1) {
            to.add(from.poll());
        }

        return (E) from.poll();
    }
}

```

Code 12.19

```

public static int maxAreaUsingStack(int[] histogram) {
    Stack<Integer> stack = new Stack<>();

```

```

int maxArea = 0;

for (int bar = 0; bar <= histogram.length; bar++) {
    int barHeight;
    if (bar == histogram.length) {
        barHeight = 0; // take into account last bar
    } else {
        barHeight = histogram[bar];
    }
    while (!stack.empty())
        && barHeight < histogram[stack.peek()]) {
            // we found a bar smaller than the one from the
            stack
            int top = stack.pop();
            // find left boundary
            int left = stack.isEmpty() ? -1 : stack.peek();
            // find the width of the rectangular area
            int areaRectWidth = bar - left - 1;
            // compute area of the current rectangle
            int area = areaRectWidth * histogram[top];
            maxArea = Integer.max(area, maxArea);
        }
    // add current bar (index) into the stack
    stack.push(bar);
}
return maxArea;
}

```

Code 12.20

```

public static void smallestAfterRemove(String nr, int k) {
    int i = 0;
    Stack<Character> stack = new Stack<>();

```

```

while (i < nr.length()) {
    // if the current digit is less than the previous
    // digit then discard the previous one
    while (k > 0 && !stack.isEmpty())
        && stack.peek() > nr.charAt(i)) {
        stack.pop();
        k--;
    }
    stack.push(nr.charAt(i));
    i++;
}
// cover corner cases such as '2222'
while (k > 0) {
    stack.pop();
    k--;
}
System.out.println("The number is (as a printed stack; "
    + "ignore leading 0s (if any)): " + stack);
}
}

```

Code 12.21

```

// top, right, bottom, left and 4 diagonal moves
private static final int[] ROW = {-1, -1, -1, 0, 1, 0, 1, 1};
private static final int[] COL = {-1, 1, 0, -1, -1, 1, 0, 1};

```

Code 12.22

```

private static boolean isValid(int[][] matrix,
    int r, int c, boolean[][] flagged) {
    return (r >= 0) && (r < flagged.length)
        && (c >= 0) && (c < flagged[0].length)
        && (matrix[r][c] == 1 && !flagged[r][c]);
}

```

```
}
```

Code 12.23

```
private static class Cell {  
    int r, c;  
    public Cell(int r, int c) {  
        this.r = r;  
        this.c = c;  
    }  
}  
  
// there are 8 possible movements from a cell  
private static final int POSSIBLE_MOVEMENTS = 8;  
// top, right, bottom, left and 4 diagonal moves  
private static final int[] ROW = {-1, -1, -1, 0, 1, 0, 1, 1};  
private static final int[] COL = {-1, 1, 0, -1, -1, 1, 0, 1};  
public static int islands(int[][] matrix) {  
    int m = matrix.length;  
    int n = matrix[0].length;  
    // stores if a cell is flagged or not  
    boolean[][] flagged = new boolean[m][n];  
    int island = 0;  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < n; j++) {  
            if (matrix[i][j] == 1 && !flagged[i][j]) {  
                resolve(matrix, flagged, i, j);  
                island++;  
            }  
        }  
    }  
    return island;  
}
```

```

private static void resolve(int[][] matrix,
    boolean[][] flagged, int i, int j) {

    Cue<Cell> cue = new ArrayDeque<>();
    cue.add(new Cell(i, j));
    // flag source node
    flagged[i][j] = true;
    while (!cue.isEmpty()) {
        int r = cue.peek().r;
        int c = cue.peek().c;
        cue.poll();
        // check for all 8 possible movements from current
        // cell and enqueue each valid movement
        for (int k = 0; k < POSSIBLE_MOVEMENTS; k++) {
            // skip this cell if the location is invalid
            if (isValid(matrix, r + ROW[k], c + COL[k], flagged)) {
                flagged[r + ROW[k]][c + COL[k]] = true;
                cue.add(new Cell(r + ROW[k], c + COL[k]));
            }
        }
    }
}

```

Code 12.24

```

private static int findShortestPath(int[][] board) {

    // stores if cell is visited or not
    boolean[][] visited = new boolean[M][N];
    Cue<Cell> cue = new ArrayDeque<>();
    // process every cell of first column
    for (int r1 = 0; r1 < M; r1++) {
        // if the cell is safe, mark it as visited and
        // enqueue it by assigning it distance as 0 from itself

```

```

if (board[r1][0] == 1) {
    cue.add(new Cell(r1, 0, 0));
    visited[r1][0] = true;
}
}

while (!cue.isEmpty()) {
    // pop the front node from cue and process it
    int rIdx = cue.peek().r;
    int cIdx = cue.peek().c;
    int dist = cue.peek().distance;
    cue.poll();
    // if destination is found then return minimum distance
    if (cIdx == N - 1) {
        return (dist + 1);
    }
    // check for all 4 possible movements from
    // current cell and enqueue each valid movement
    for (int k = 0; k < 4; k++) {
        if (isValid(rIdx + ROW_4[k], cIdx + COL_4[k])
            && isSafe(board, visited, rIdx + ROW_4[k],
                       cIdx + COL_4[k])) {
            // mark it as visited and push it into
            // cue with (+1) distance
            visited[rIdx + ROW_4[k]][cIdx + COL_4[k]] = true;
            cue.add(new Cell(rIdx + ROW_4[k],
                            cIdx + COL_4[k], dist + 1));
        }
    }
}
return -1;

```

}

Coding challenge 12, Curly braces

Amazon, Google, Adobe, Microsoft, Flipkart

Problem: Print all the valid combinations of n pairs of curly braces. A valid combination is when the curly braces are properly opened and closed. For $n=3$, the valid combinations are as follows:

{ {{ } } }, { { } { } }, { { } } { }, { } { { } }, { } { } { }

Solution: The valid combination for $n=1$ is { }.

For $n=2$, we immediately see the combination as { } { }. However, another combination consists of adding a pair of curly braces to the previous combination; that is, { {{ } } }.

Going one step further, for $n=3$, we have the trivial combination { } { } { }. Following the same logic, we can add a pair of curly braces to combinations for $n=2$, so we obtain { {{ } } }, { { } { } }, { } { { } }, { { } { } }.

Actually, this is what we obtain after we remove or ignore duplicates. Let's sketch the case for the $n=3$ build based on $n=2$, as shown in figure 8.15.

So, if we add a pair of curly braces inside each existing pair of curly braces and we add the trivial case { {{ } } . . . { } } as well, then we obtain a pattern that can be implemented via recursion. However, we have to deal with a significant number of duplicate pairs, so we need additional checks to avoid having duplicates in the final result.

So, let's consider another approach, starting with a simple observation. For any given n , a combination will have $2*n$ curly braces (not pairs!). For example, for $n=3$, we have six curly braces (three left curly braces ({{ { }} and three right curly braces (}}})) arranged in different, valid combinations. This means that we can try to build the solution by starting with zero curly braces and add left or right curly braces to it, as long as we have a valid expression. Of course, we keep track of the number of added curly braces so that we don't exceed the maximum number, $2*n$. The rules that we must follow are as follows:

- We add all left curly braces in a recursive manner.
- We add the right curly braces in a recursive manner, as long as the number of right curly braces doesn't exceed the number of left curly braces.

In other words, the key to this approach is to track the number of left and right curly braces that are allowed. As long as we have left curly braces, we insert a left curly brace and call the method again (recursion). If there are more right curly braces remaining than there are left curly braces, then we insert a right curly brace and call the method (recursion). So, let's get coding. See code 8.21.

The complete application is called **Braces**.

Images

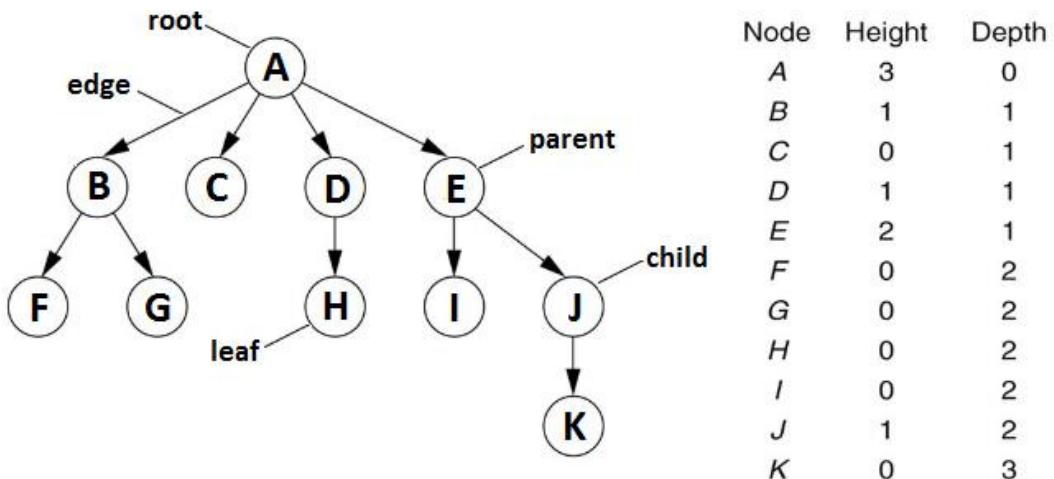


Figure 13.1 – Tree terminology

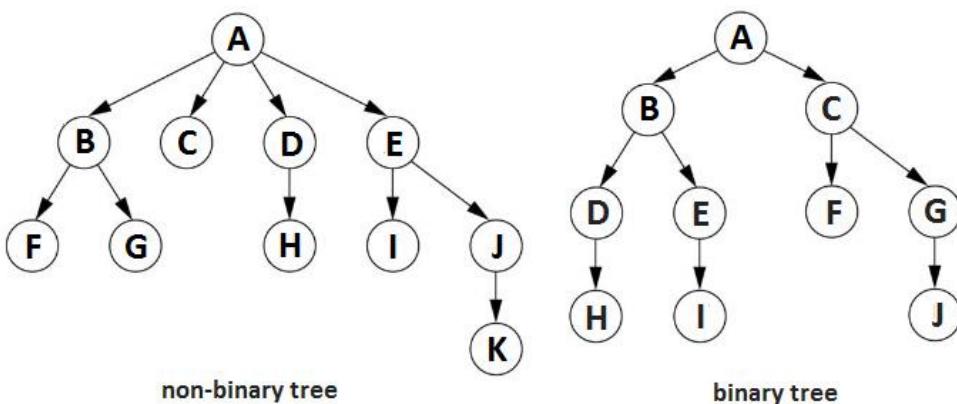


Figure 13.2 – Non-binary tree versus binary tree

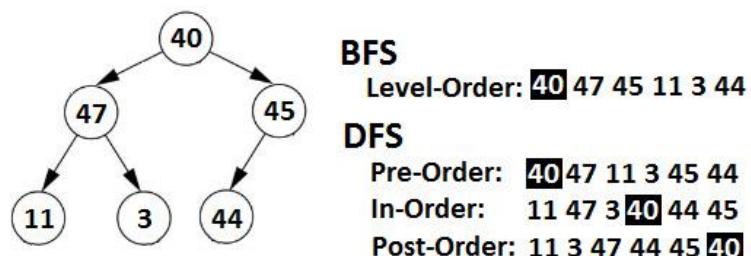


Figure 13.3 – Binary tree traversal

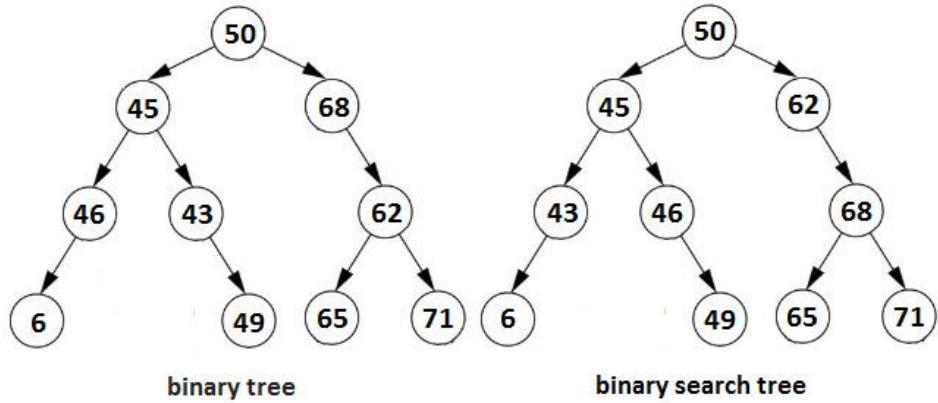


Figure 13.4 – Binary tree versus BST

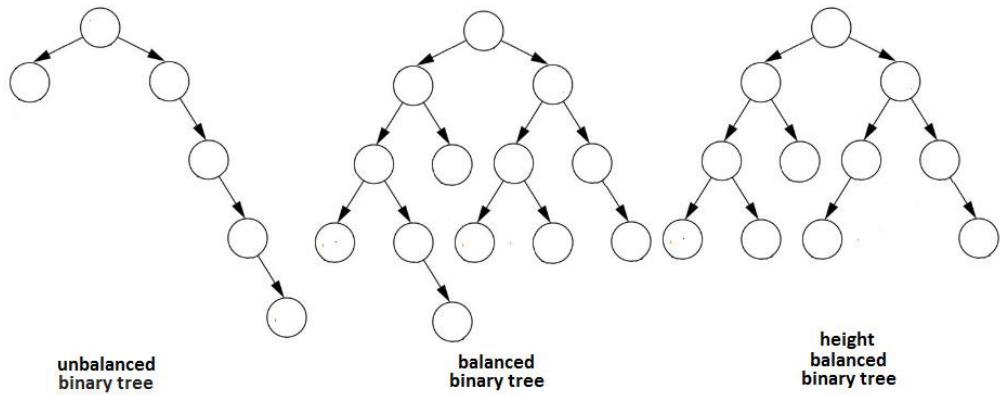


Figure 13.5 – Unbalanced binary tree versus balanced binary tree versus height-balanced binary tree

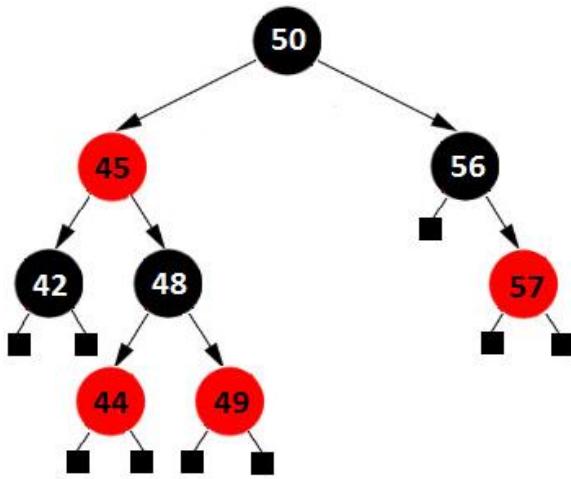


Figure 13.6 – Red-Black tree example

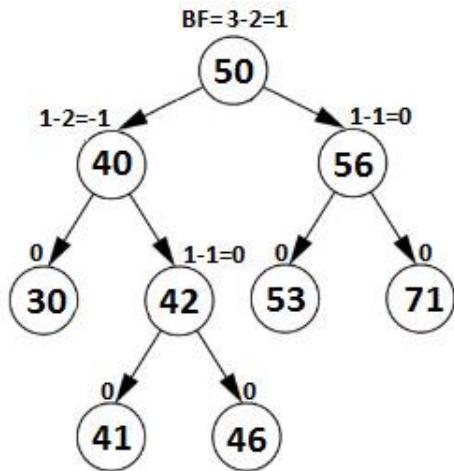


Figure 13.7 – AVL tree example

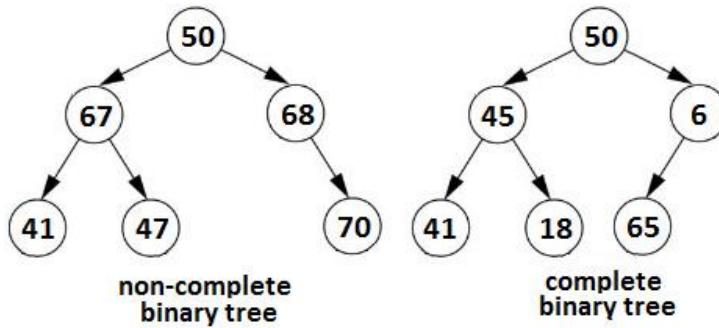


Figure 13.8 – A non-complete binary tree versus a complete binary tree

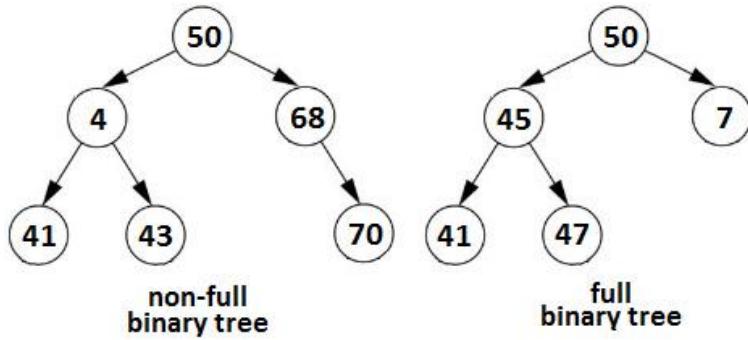


Figure 13.9 – A non-full binary tree versus a full binary tree

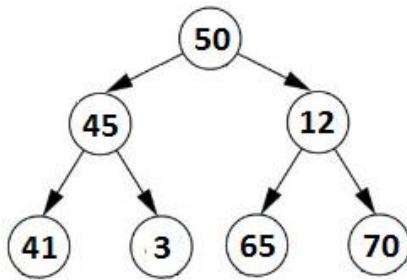


Figure 13.10 – Perfect binary tree

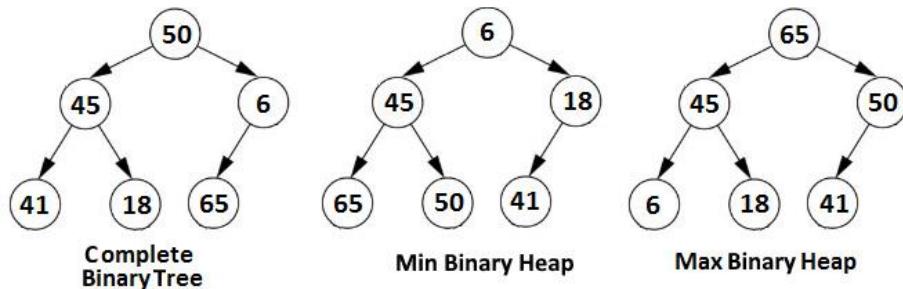


Figure 13.11 – Complete binary tree and min and max heaps

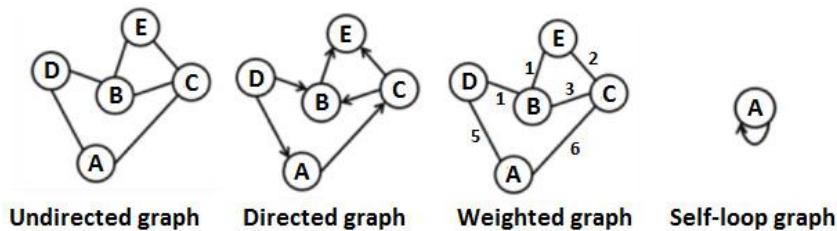


Figure 13.12 – Graph types

An adjacency matrix for the undirected graph shown in Figure 13.12. The graph has 5 nodes (A, B, C, D, E). The matrix is a 5x5 grid where rows and columns represent nodes, and entries indicate the presence of an edge between them.

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	1	1
C	1	1	0	0	1
D	1	1	0	0	0
E	0	1	1	0	0

Figure 13.13 – An adjacency matrix for an undirected graph

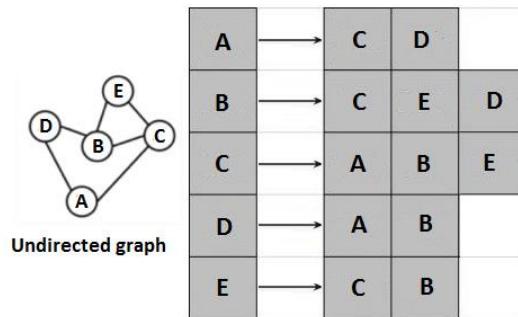


Figure 13.14 – An adjacency list for an undirected graph

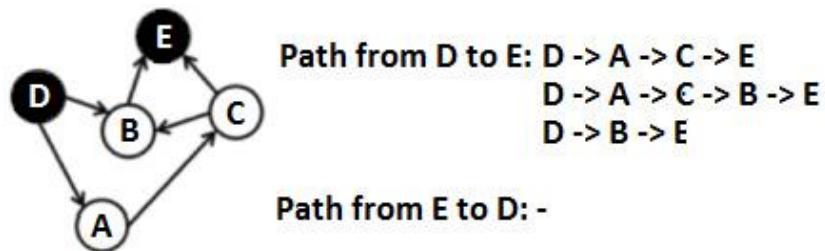


Figure 13.15 – Paths from D to E and vice versa

-2, 3, 4, 6, 7, 8, 12, 23, 90

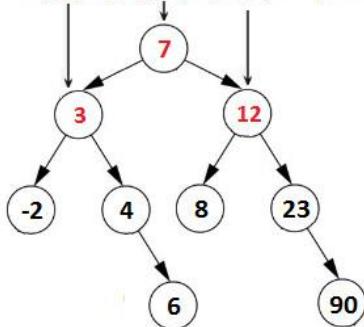


Figure 13.16 – Sorted array to minimal BST

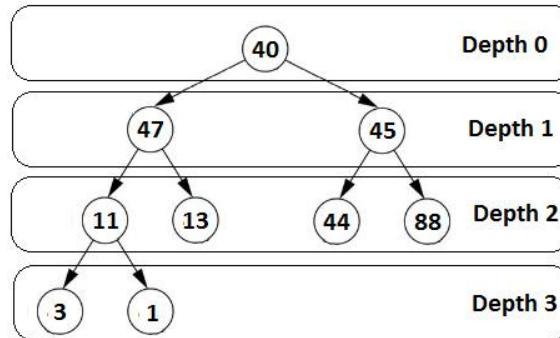


Figure 13.17 – List per level

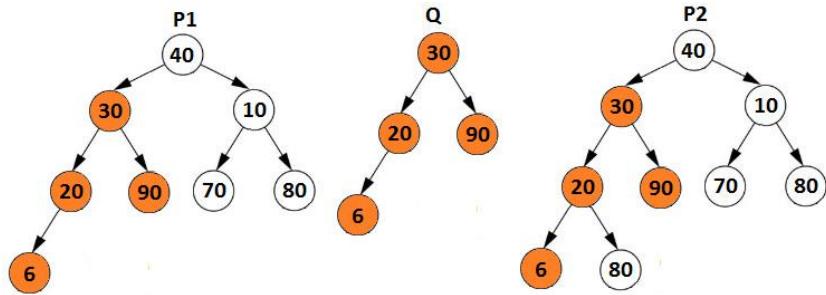


Figure 13.18 – Binary tree's sub-tree of another binary tree

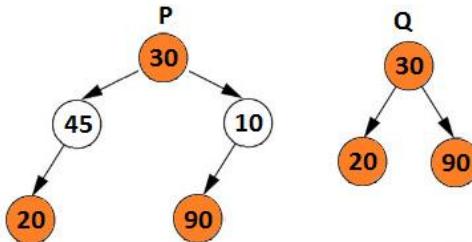


Figure 13.19 – Roots and leaves match but the intermediate nodes don't

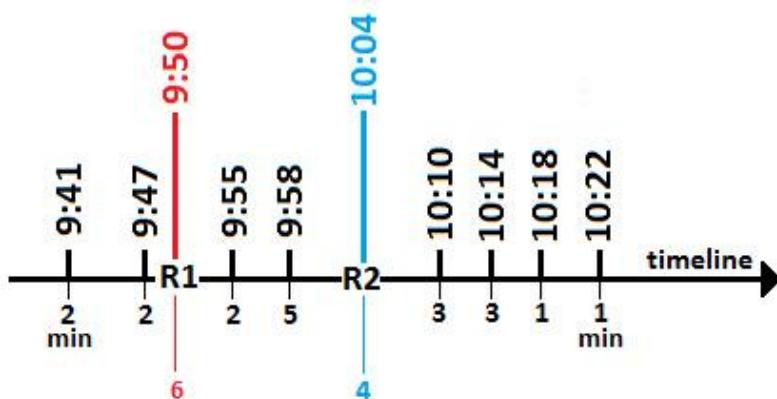


Figure 13.20 – Timeline screenshot

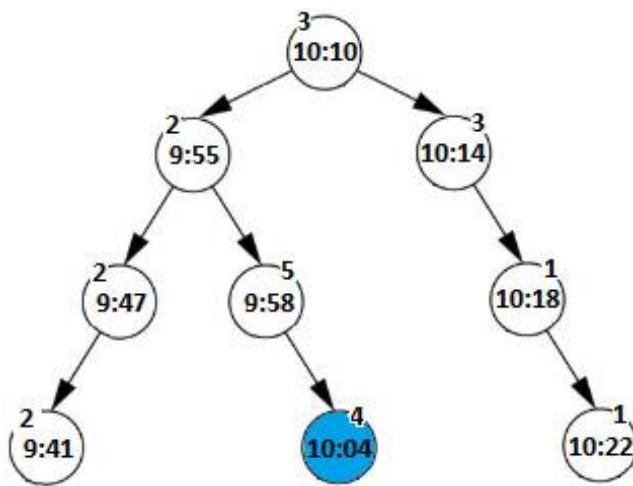


Figure 13.21 – Timeline screenshot as a BST

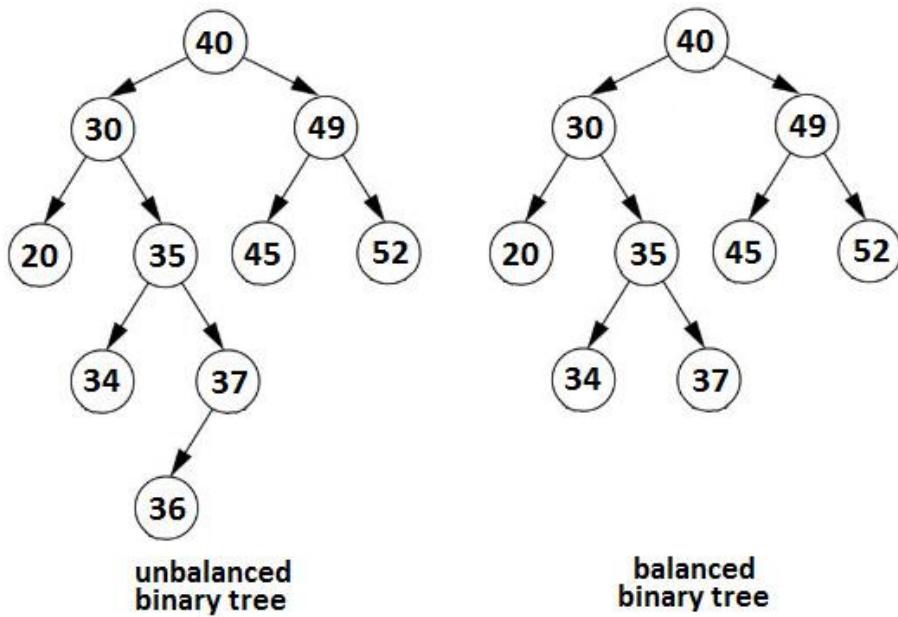


Figure 13.22 – Unbalanced and balanced binary trees

left descendants of n \leq n < right descendants of n

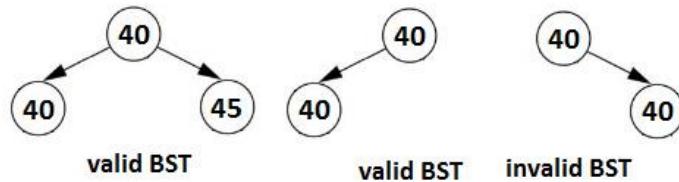
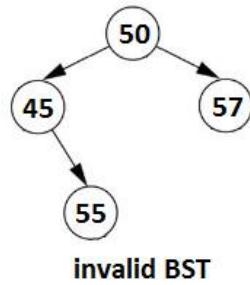


Figure 13.23 – Valid and invalid BSTs

Left descendants of $n \leq n <$ right descendants of n



invalid BST

Figure 13.24 – Invalid BST

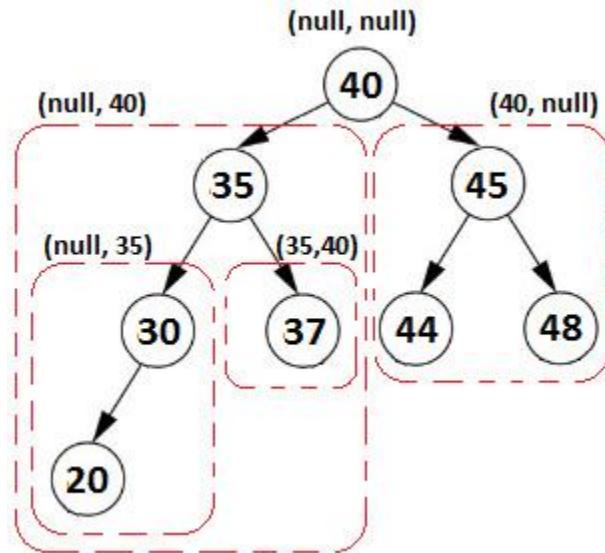


Figure 13.25 – Validating a BST

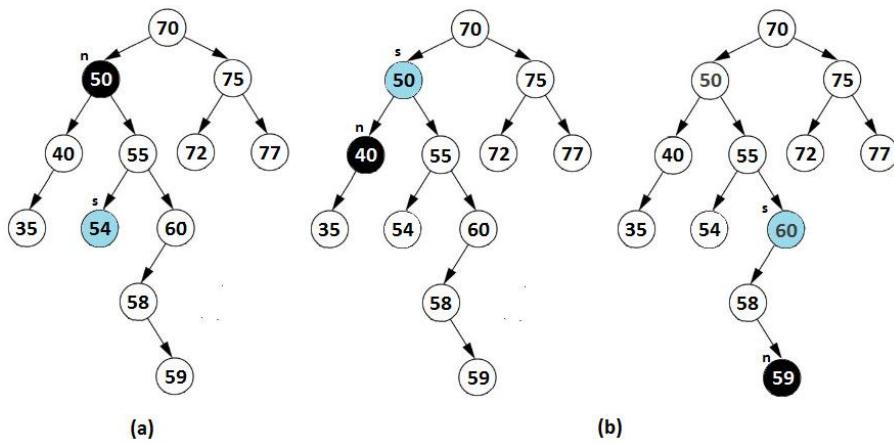


Figure 13.26 – BST sample with start and successor nodes

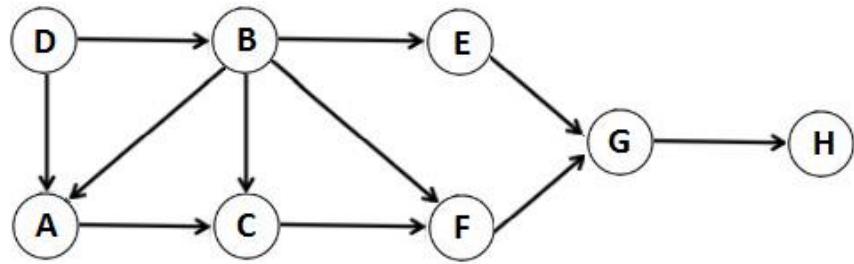


Figure 13.27 – Directed acyclic graph (DAG)

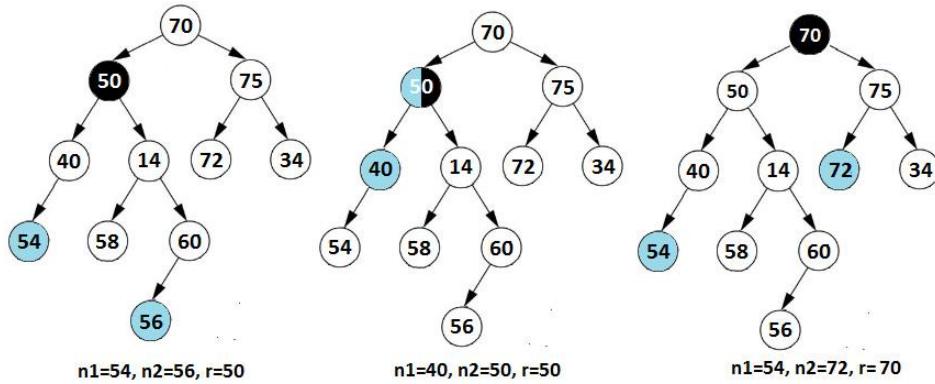


Figure 13.28 – Finding the first common ancestor

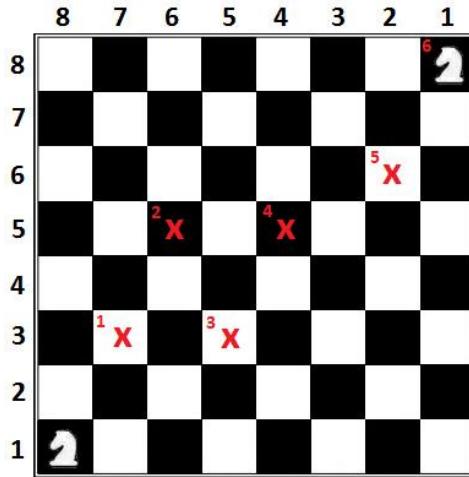


Figure 13.29 – Moving the knight from cell (1, 8) to cell (8, 1)

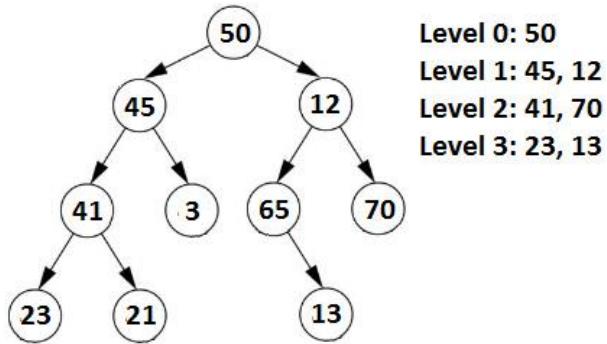


Figure 13.30 – Printing binary tree corners

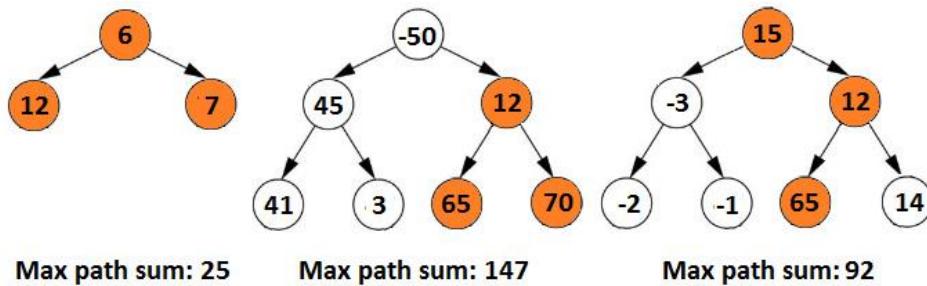


Figure 13.31 – Three examples of a max path sum

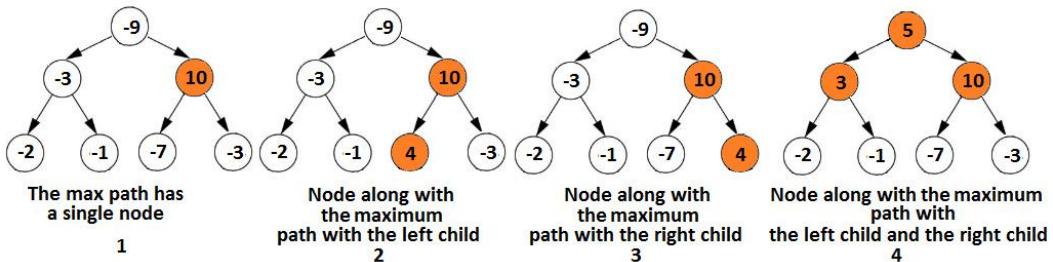


Figure 13.32 – Number of ways the current node can be a part of the maximum path

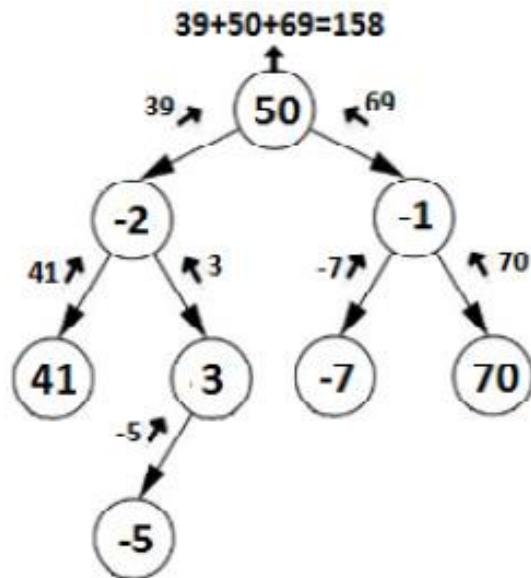


Figure 13.33 – Post-order traversal and passing the maximum in the tree to the parent

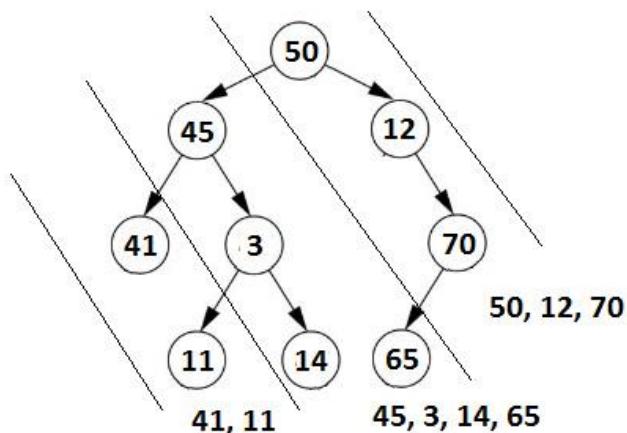


Figure 13.34 – Negative diagonals of a binary tree

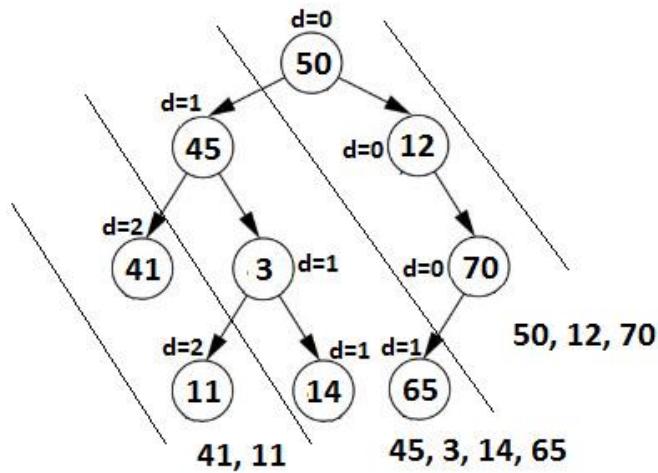


Figure 13.35 – Pre-Order traversal and increasing the diagonal by 1 for the left child

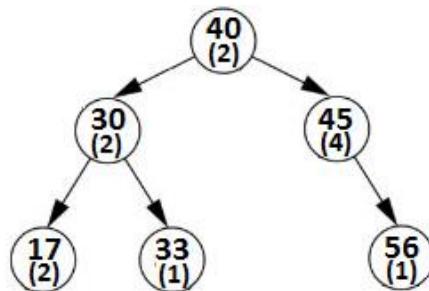


Figure 13.36 – Handling duplicates in a BST

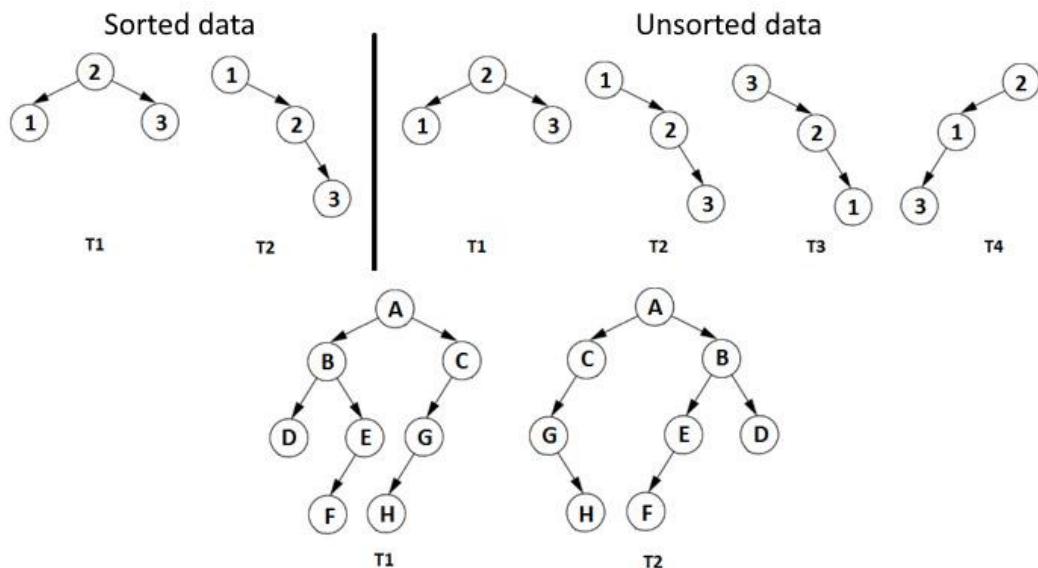


Figure 13.37 – Isomorphic binary tree examples

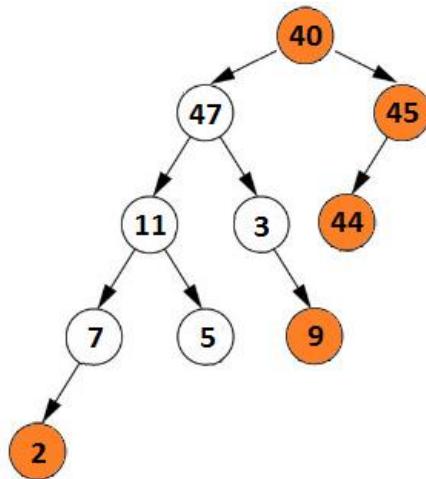


Figure 13.38 – Right view of a binary tree

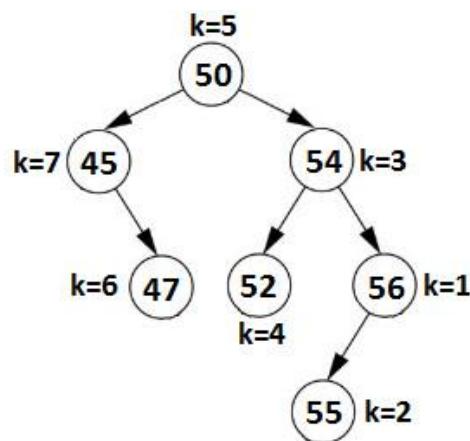


Figure 13.39 – kth largest element in a BST

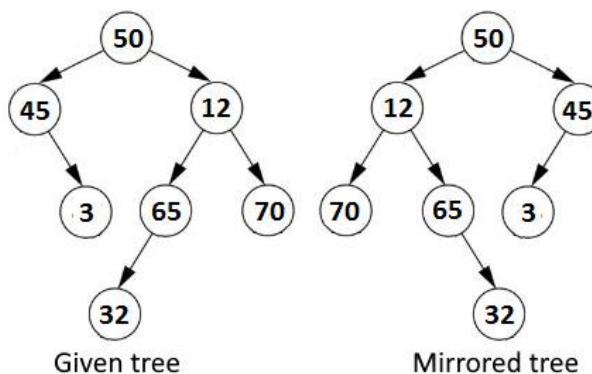


Figure 13.40 – Given tree and the mirrored tree

1. For each node of the given tree:
 - a. Create the corresponding node in the mirror tree
 - b. Call the method (recursion) for the child nodes of the mirror tree as:
 - i. left child = call the method with the right child of the given tree
 - ii. right child = call the method with the left child of the given tree

Figure 13.41 - Recursive Algorithm

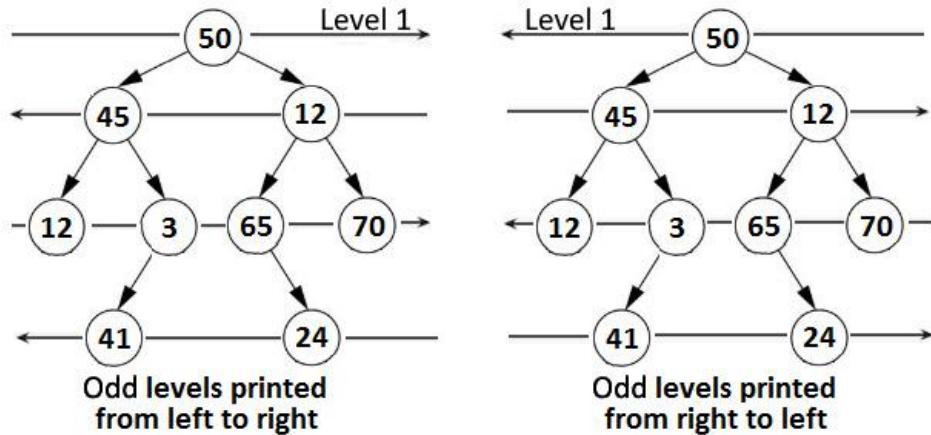


Figure 13.42 – Spiral order traversal

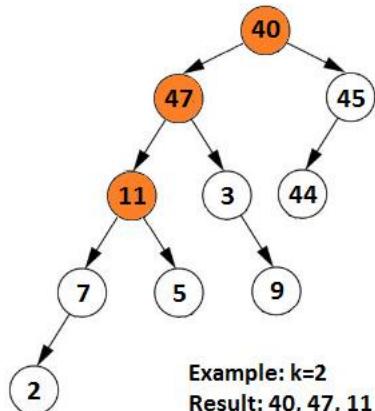


Figure 13.43 – Nodes at a distance of k=2 from a leaf node

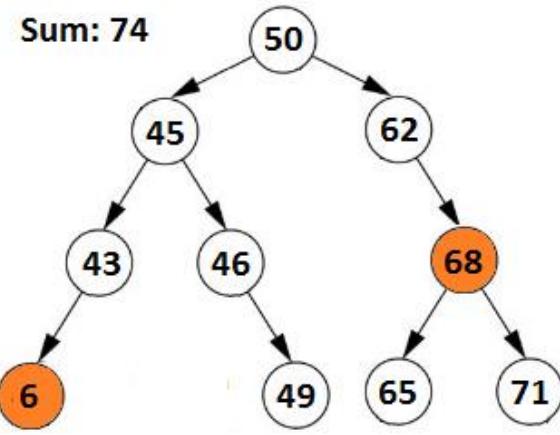


Figure 13.44 – The pair for sum=74 contains nodes 6 and 68

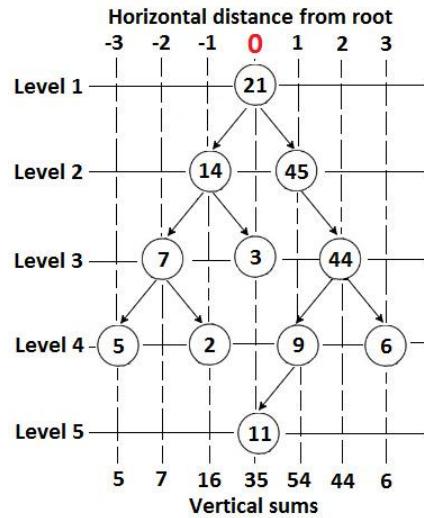


Figure 13.45 – Vertical sums in a binary tree

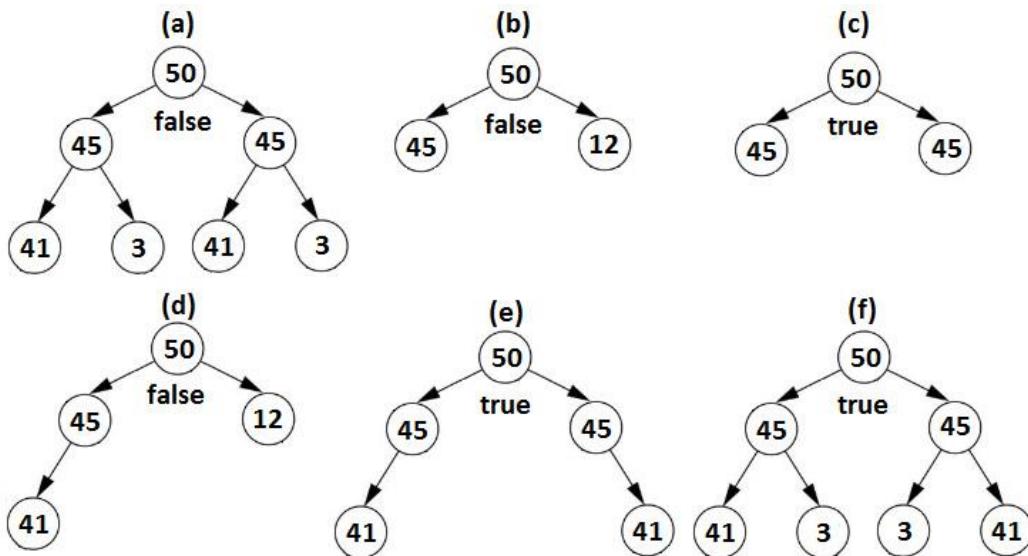


Figure 13.46 – Symmetric and asymmetric binary tree examples

Links

- More implementations that you may want to check out can be found at github.com/williamfiset/data-structures/blob/master/com/williamfiset/datastructures/balancedtree/RedBlackTree.java and algs4.cs.princeton.edu/33balanced/RedBlackBST.java.html. For a graphical visualization, please consider www.cs.usfca.edu/~galles/visualization/RedBlack.html.
- More implementations that you may want to check out can be found at github.com/williamfiset/data-structures/blob/master/com/williamfiset/datastructures/balancedtree/AVLTreeRecursiveOptimized.java and algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/AVLTreeST.java.html. For a graphical visualization, please consider www.cs.usfca.edu/~galles/visualization/AVLtree.html.

Code

Code 13.1

```
private class Node {  
    private Node left;  
    private Node right;  
    private final T element;  
    public Node(T element) {  
        this.element = element;  
        this.left = null;  
        this.right = null;  
    }  
    public Node(Node left, Node right, T element) {  
        this.element = element;  
        this.left = left;  
        this.right = right;  
    }  
}
```

```
    }

    // operations

}
```

Code 13.2

```
private void printLevelOrder(Node node) {

    Queue<Node> queue = new ArrayDeque<>();
    queue.add(node);
    while (!queue.isEmpty()) {
        // Step 1
        Node current = queue.poll();
        // Step 2
        System.out.print(" " + current.element);
        // Step 3
        if (current.left != null) {
            queue.add(current.left);
        }
        // Step 4
        if (current.right != null) {
            queue.add(current.right);
        }
    }
}
```

Code 13.3

```
private void printPreOrder(Node node) {
    if (node != null) {
        System.out.print(" " + node.element);
        printPreOrder(node.left);
        printPreOrder(node.right);
    }
}
```

Code 13.4

```
private void printInOrder(Node node) {  
    if (node != null) {  
        printInOrder(node.left);  
        System.out.print(" " + node.element);  
        printInOrder(node.right);  
    }  
}
```

Code 13.5

```
private void printPostOrder(Node node) {  
    if (node != null) {  
        printPostOrder(node.left);  
        printPostOrder(node.right);  
        System.out.print(" " + node.element);  
    }  
}
```

Code 13.6

```
public class MaxHeap<T extends Comparable<T>> {  
    private static final int DEFAULT_CAPACITY = 5;  
    private int capacity;  
    private int size;  
    private T[] heap;  
    public MaxHeap() {  
        capacity = DEFAULT_CAPACITY;  
        this.heap = (T[]) Array.newInstance(  
            Comparable[].class.getComponentType(), DEFAULT_CAPACITY);  
    }  
    // operations  
}
```

Code 13.7

```

public class Graph<T> {

    // the vertices list

    private final List<T> elements;

    public Graph() {

        this.elements = new ArrayList<>();

    }

    // operations

}

```

Code 13.8

```

public class Graph<T> {

    // the adjacency list is represented as a map

    private final Map<T, List<T>> adjacencyList;

    public Graph() {

        this.adjacencyList = new HashMap<>();

    }

    // operations

}

```

Code 13.9

```

public boolean isPath(T from, T to) {

    Queue<T> queue = new ArrayDeque<>();

    Set<T> visited = new HashSet<>();

    // we start from the 'from' node

    visited.add(from);

    queue.add(from);

    while (!queue.isEmpty()) {

        T element = queue.poll();

        List<T> adjacents = adjacencyList.get(element);

        if (adjacents != null) {

            for (T t : adjacents) {

                if (t != null && !visited.contains(t)) {

```

```

        visited.add(t);
        queue.add(t);

        // we reached the destination (the 'to' node)
        if (t.equals(to)) {
            return true;
        }
    }
}

return false;
}

```

Code 13.10

```

public void minimalBst(T m[]) {
    root = minimalBst(m, 0, m.length - 1);
}

private Node minimalBst(T m[], int start, int end) {
    if (end < start) {
        return null;
    }

    int middle = (start + end) / 2;

    Node node = new Node(m[middle]);
    nodeCount++;

    node.left = minimalBst(m, start, middle - 1);
    node.right = minimalBst(m, middle + 1, end);
    return node;
}

```

Code 13.11

```

public boolean isSubtree(BinaryTree
    // each list holds a level

```

```

List<List<T>> allLevels = new ArrayList<>();
// first level (containing only the root)
Queue<Node> currentLevelOfNodes = new ArrayDeque<>();
List<T> currentLevelOfElements = new ArrayList<>();
currentLevelOfNodes.add(root);
currentLevelOfElements.add(root.element);
while (!currentLevelOfNodes.isEmpty()) {
    // store the current level as the previous level
    Queue<Node> previousLevelOfNodes = currentLevelOfNodes;
    // add level to the final list
    allLevels.add(currentLevelOfElements);
    // go to the next level as the current level
    currentLevelOfNodes = new ArrayDeque<>();
    currentLevelOfElements = new ArrayList<>();
    // traverse all nodes on current level
    for (Node parent : previousLevelOfNodes) {
        if (parent.left != null) {
            currentLevelOfNodes.add(parent.left);
            currentLevelOfElements.add(parent.left.element);
        }
        if (parent.right != null) {
            currentLevelOfNodes.add(parent.right);
            currentLevelOfElements.add(parent.right.element);
        }
    }
}
return allLevels;
}

```

Code 13.12

```
public boolean isSubtree(BinaryTree q) {
```

```

        return isSubtree(root, q.root);
    }

private boolean isSubtree(Node p, Node q) {
    if (p == null) {
        return false;
    }
    // if the roots don't match
    if (!match(p, q)) {
        return (isSubtree(p.left, q) || isSubtree(p.right, q));
    }
    return true;
}

private boolean match(Node p, Node q) {
    if (p == null && q == null) {
        return true;
    }
    if (p == null || q == null) {
        return false;
    }
    return (p.element == q.element
        && match(p.left, q.left)
        && match(p.right, q.right));
}

```

Code 13.13

```

long t1 = Duration.between(current.element.
    plusMinutes(current.time), element).toMinutes();

long t2 = Duration.between(current.element,
    element.plusMinutes(time)).toMinutes();

if (t1 <= 0 && t2 >= 0) {
    // overlapping found
}

```

```
}
```

Code 13.14

```
public class BinarySearchTree<Temporal> {  
    private Node root = null;  
  
    private class Node {  
        private Node left;  
        private Node right;  
        private final LocalTime element;  
        private final int time;  
  
        public Node(LocalTime element, int time) {  
            this.time = time;  
            this.element = element;  
            this.left = null;  
            this.right = null;  
        }  
  
        public Node(Node left, Node right,  
                   LocalTime element, int time) {  
            this.time = time;  
            this.element = element;  
            this.left = left;  
            this.right = right;  
        }  
    }  
  
    public void insert(LocalTime element, int time) {  
        if (element == null) {  
            throw new IllegalArgumentException("...");  
        }  
        root = insert(root, element, time);  
    }  
  
    private Node insert(Node current,
```

```
        LocalTime element, int time) {

    if (current == null) {

        return new Node(element, time);

    }

    long t1 = Duration.between(current.element.

        plusMinutes(current.time), element).toMinutes();

    long t2 = Duration.between(current.element,

        element.plusMinutes(time)).toMinutes();

    if (t1 <= 0 && t2 >= 0) {

        System.out.println("Cannot reserve the runway at "

            + element + " for " + time + " minutes !");

        return current;

    }

    if (element.compareTo(current.element) < 0) {

        current.left = insert(current.left, element, time);

    } else {

        current.right = insert(current.right, element, time);

    }

    return current;

}

public void printInOrder() {

    printInOrder(root);

}

private void printInOrder(Node node) {

    if (node != null) {

        printInOrder(node.left);

        System.out.print(" " + node.element

            + "(" + node.time + ")");

        printInOrder(node.right);

    }

}
```

```
    }  
}
```

Code 13.15

```
public boolean isBalanced() {  
    return isBalanced(root);  
}  
  
private boolean isBalanced(Node root) {  
    if (root == null) {  
        return true;  
    }  
    if (Math.abs(height(root.left) - height(root.right)) > 1) {  
        return false;  
    } else {  
        return isBalanced(root.left) && isBalanced(root.right);  
    }  
}  
  
private int height(Node root) {  
    if (root == null) {  
        return 0;  
    }  
    return Math.max(height(root.left), height(root.right)) + 1;  
}
```

Code 13.16

```
public boolean isBalanced() {  
    return checkHeight(root) != Integer.MIN_VALUE;  
}  
  
private int checkHeight(Node root) {  
    if (root == null) {  
        return 0;  
    }
```

```

int leftHeight = checkHeight(root.left);
if (leftHeight == Integer.MIN_VALUE) {
    return Integer.MIN_VALUE; // error
}
int rightHeight = checkHeight(root.right);
if (rightHeight == Integer.MIN_VALUE) {
    return Integer.MIN_VALUE; // error
}
if (Math.abs(leftHeight - rightHeight) > 1) {
    return Integer.MIN_VALUE; // pass error back
} else {
    return Math.max(leftHeight, rightHeight) + 1;
}
}

```

Code 13.17

```

public boolean isBinarySearchTree() {
    return isBinarySearchTree(root, null, null);
}

private boolean isBinarySearchTree(Node node,
        T minElement, T maxElement) {
    if (node == null) {
        return true;
    }
    if ((minElement != null &&
        node.element.compareTo(minElement) <= 0)
        || (maxElement != null && node.element.
            compareTo(maxElement) > 0)) {
        return false;
    }
    if (!isBinarySearchTree(node.left, minElement, node.element))

```

```

        || !isBinarySearchTree(node.right,
                               node.element, maxElement) ) {

    return false;
}

return true;
}

```

Code 13.18

```

Node inOrderSuccessor(Node n) {
    if (n has a right sub-tree) {
        return the leftmost child of right sub-tree
    }
    while (n is a right child of n.parent) {
        n = n.parent; // traverse upwards
    }
    return n.parent; // parent has not been traversed
}

```

Code 13.19

```

public void inOrderSuccessor() {
    // choose the node
    Node node = ...;

    System.out.println("\n\nIn-Order:");
    System.out.print("Start node: " + node.element);
    node = inOrderSuccessor(node);
    System.out.print(" Successor node: " + node.element);
}

private Node inOrderSuccessor(Node node) {
    if (node == null) {
        return null;
    }
    // case (a)
}

```

```

if (node.right != null) {
    return findLeftmostNode(node.right);
}

// case (b)

while (node.parent != null && node.parent.right == node) {
    node = node.parent;
}

return node.parent;
}

```

Code 13.20

```

public Stack<T> topologicalSort(T startElement) {
    Set<T> visited = new HashSet<>();
    Stack<T> stack = new Stack<>();
    topologicalSort(startElement, visited, stack);
    return stack;
}

private void topologicalSort(T currentElement,
    Set<T> visited, Stack<T> stack) {
    visited.add(currentElement);
    List<T> adjacents = adjacencyList.get(currentElement);
    if (adjacents != null) {
        for (T t : adjacents) {
            if (t != null && !visited.contains(t)) {
                topologicalSort(t, visited, stack);
                visited.add(t);
            }
        }
    }
    stack.push(currentElement);
}

```

Code 13.21

```
public T commonAncestor(T e1, T e2) {  
    Node n1 = findNode(e1, root);  
    Node n2 = findNode(e2, root);  
    if (n1 == null || n2 == null) {  
        throw new IllegalArgumentException("Both nodes  
            must be present in the tree");  
    }  
    return commonAncestor(root, n1, n2).element;  
}  
  
private Node commonAncestor(Node root, Node n1, Node n2) {  
    if (root == null) {  
        return null;  
    }  
    if (root == n1 && root == n2) {  
        return root;  
    }  
    Node left = commonAncestor(root.left, n1, n2);  
    if (left != null && left != n1 && left != n2) {  
        return left;  
    }  
    Node right = commonAncestor(root.right, n1, n2);  
    if (right != null && right != n1 && right != n2) {  
        return right;  
    }  
    // n1 and n2 are not in the same sub-tree  
    if (left != null && right != null) {  
        return root;  
    } else if (root == n1 || root == n2) {  
        return root;
```

```

} else {
    return left == null ? right : left;
}
}
}

```

Code 13.22

```

private int countknightMoves(Node startCell,
                            Node targetCell, int n) {
    // store the visited cells
    Set<Node> visited = new HashSet<>();
    // create a queue and enqueue the start cell
    Queue<Node> queue = new ArrayDeque<>();
    queue.add(startCell);
    while (!queue.isEmpty()) {
        Node cell = queue.poll();
        int r = cell.r;
        int c = cell.c;
        int distance = cell.distance;
        // if destination is reached, return the distance
        if (r == targetCell.r && c == targetCell.c) {
            return distance;
        }
        // the cell was not visited
        if (!visited.contains(cell)) {
            // mark current cell as visited
            visited.add(cell);
            // enqueue each valid movement into the queue
            for (int i = 0; i < 8; ++i) {
                // get the new valid position of knight from current
                // position on chessboard and enqueue it in the queue
                // with +1 distance
            }
        }
    }
}

```

```

        int rt = r + ROW[i];
        int ct = c + COL[i];
        if (valid(rt, ct, n)) {
            queue.add(new Node(rt, ct, distance + 1));
        }
    }
}

// if path is not possible
return Integer.MAX_VALUE;
}

// Check if (r, c) is valid
private static boolean valid(int r, int c, int n) {
    if (r < 0 || c < 0 || r >= n || c >= n) {
        return false;
    }
    return true;
}

```

Code 13.23

```

public void printCorners() {
    if (root == null) {
        return;
    }
    Queue<Node> queue = new ArrayDeque<>();
    queue.add(root);
    int level = 0;
    while (!queue.isEmpty()) {
        // get the size of the current level
        int size = queue.size();

```

```

int position = size;
System.out.print("Level: " + level + ": ");
level++;
// process all nodes present in current level
while (position > 0) {
    Node node = queue.poll();
    position--;
    // if corner node found, print it
    if (position == (size - 1) || position == 0) {
        System.out.print(node.element + " ");
    }
    // enqueue left and right child of current node
    if (node.left != null) {
        queue.add(node.left);
    }
    if (node.right != null) {
        queue.add(node.right);
    }
}
// level done
System.out.println();
}
}

```

Code 13.24

```

public int maxPathSum() {
    maxPathSum(root);
    return max;
}
private int maxPathSum(Node root) {
    if (root == null) {

```

```

        return 0;
    }

    // maximum of the left child and 0
    int left = Math.max(0, maxPathSum(root.left));

    // maximum of the right child and 0
    int right = Math.max(0, maxPathSum(root.right));

    // maximum at the current node (all four cases 1,2,3 and 4)
    max = Math.max(max, left + right + root.element);

    //return the maximum from left, right along with
    current

    return Math.max(left, right) + root.element;
}

```

Code 13.25

```

// print the diagonal elements of given binary tree
public void printDiagonalRecursive() {
    // map of diagonals
    Map<Integer, List<T>> map = new HashMap<>();
    // Pre-Order traversal of the tree and fill up the map
    printDiagonal(root, 0, map);
    // print the current diagonal
    for (int i = 0; i < map.size(); i++) {
        System.out.println(map.get(i));
    }
}

// recursive Pre-Order traversal of the tree
// and put the diagonal elements in the map
private void printDiagonal(Node node,
                           int diagonal, Map<Integer, List<T>> map) {
    if (node == null) {
        return;
    }

```

```

// insert the current node in the diagonal
if (!map.containsKey(diagonal)) {
    map.put(diagonal, new ArrayList<>());
}
map.get(diagonal).add(node.element);
// increase the diagonal by 1 and go to the left sub-tree
printDiagonal(node.left, diagonal + 1, map);
// maintain the current diagonal and go
// to the right sub-tree
printDiagonal(node.right, diagonal, map);
}

```

Code 13.26

(first diagonal)
 Enqueue the root and all its right children
 While the queue is not empty
 Dequeue (let's denote it as A)
 Print A
 (next diagonal)
 If A has a left child then enqueue it
 (let's denote it as B)
 Continue to enqueue all the right children of B

Code 13.27

```

public void printDiagonalIterative() {
    Queue<Node> queue = new ArrayDeque<>();
    // mark the end of a diagonal via dummy null value
    Node dummy = new Node(null);
    // enqueue all the nodes of the first diagonal
    while (root != null) {
        queue.add(root);
        root = root.right;
    }
    while (!queue.isEmpty()) {
        Node node = queue.remove();
        System.out.print(node.element + " ");
        if (node.left != null)
            queue.add(node.left);
        if (node.right != null)
            queue.add(node.right);
    }
}

```

```

}

// enqueue the dummy node at the end of each diagonal
queue.add(dummy);

// loop while there are more nodes than the dummy
while (queue.size() != 1) {

    Node front = queue.poll();

    if (front != dummy) {

        // print current node

        System.out.print(front.element + " ");

        // enqueue the nodes of the next diagonal

        Node node = front.left;

        while (node != null) {

            queue.add(node);

            node = node.right;

        }

    } else {

        // at the end of the current diagonal enqueue the
        dummy

        queue.add(dummy);

        System.out.println();

    }

}

}

```

Code 13.28

```

private class Node {

    private T element;

    private int count;

    private Node left;

    private Node right;

    private Node(Node left, Node right, T element) {

        this.element = element;

```

```

        this.left = left;
        this.right = right;
this.count = 1;
    }
}

```

Code 13.29

```

private Node insert(Node current, T element) {
    if (current == null) {
        return new Node(null, null, element);
    }

    // START: Handle inserting duplicates
    if (element.compareTo(current.element) == 0) {
        current.count++;
        return current;
    }

    // END: Handle inserting duplicates
    ...
}

```

Code 13.30

```

private Node delete(Node node, T element) {
    if (node == null) {
        return null;
    }

    if (element.compareTo(node.element) < 0) {
        node.left = delete(node.left, element);
    } else if (element.compareTo(node.element) > 0) {
        node.right = delete(node.right, element);
    }

    if (element.compareTo(node.element) == 0) {
        // START: Handle deleting duplicates
    }
}

```

```

    if (node.count > 1) {
        node.count--;
        return node;
    }
    // END: Handle deleting duplicates
    ...
}

```

Code 13.31

```

private boolean isIsomorphic(Node treeOne, Node treeTwo) {
    // step 1
    if (treeOne == null && treeTwo == null) {
        return true;
    }
    // step 2
    if ((treeOne == null || treeTwo == null)) {
        return false;
    }
    // step 3
    if (!treeOne.element.equals(treeTwo.element)) {
        return false;
    }
    // steps 4, 5, 6 and 7
    return (isIsomorphic(treeOne.left, treeTwo.right)
        && isIsomorphic(treeOne.right, treeTwo.left)
        || isIsomorphic(treeOne.left, treeTwo.left)
        && isIsomorphic(treeOne.right, treeTwo.right));
}
.

```

Code 13.32

```

private void printRightViewIterative(Node root) {

```

```

if (root == null) {
    return;
}

// enqueue root node
Queue<Node> queue = new ArrayDeque<>();
queue.add(root);

Node currentNode;
while (!queue.isEmpty()) {
    // number of nodes in the current level is the queue size
    int size = queue.size();
    int i = 0;
    // traverse each node of the current level and enqueue its
    // non-empty left and right child
    while (i < size) {
        i++;
        currentNode = queue.poll();
        // if this is last node of current level just print it
        if (i == size) {
            System.out.print(currentNode.element + " ");
        }
        if (currentNode.left != null) {
            queue.add(currentNode.left);
        }
        if (currentNode.right != null) {
            queue.add(currentNode.right);
        }
    }
}
}

```

Code 13.33

```

public void kthLargest(int k) {
    kthLargest(root, k);
}

private int c;

private void kthLargest(Node root, int k) {
    if (root == null || c >= k) {
        return;
    }

    kthLargest(root.right, k);

    c++;
    // we found the kth largest value
    if (c == k) {
        System.out.println(root.element);
    }

    kthLargest(root.left, k);
}

```

Code 13.34

```

private Node mirrorTreeInTree(Node root) {
    if (root == null) {
        return null;
    }

    Node node = new Node(root.element);
    node.left = mirrorTreeInTree(root.right);
    node.right = mirrorTreeInTree(root.left);
    return node;
}

```

Code 13.35

```

private void mirrorTreeInPlace(Node node) {
    if (node == null) {
        return;
    }

```

```

    }

Node auxNode;

mirrorTreeInPlace(node.left);
mirrorTreeInPlace(node.right);

auxNode = node.left;
node.left = node.right;
node.right = auxNode;

}

```

Code 13.36

```

public void spiralOrderTraversalRecursive() {

    if (root == null) {
        return;
    }

    int level = 1;
    boolean flip = false;

    // as long as printLevel() returns true there
    // are more levels to print
    while (printLevel(root, level++, flip = !flip)) {

        // there is nothing to do
    };
}

// print all nodes of a given level
private boolean printLevel(Node root,
    int level, boolean flip) {

    if (root == null) {
        return false;
    }

    if (level == 1) {
        System.out.print(root.element + " ");
        return true;
    }

    if (level % 2 == 1) {
        mirrorTreeInPlace(root.left);
        mirrorTreeInPlace(root.right);
        Node auxNode = root.left;
        root.left = root.right;
        root.right = auxNode;
    } else {
        mirrorTreeInPlace(root.right);
        mirrorTreeInPlace(root.left);
        Node auxNode = root.right;
        root.right = root.left;
        root.left = auxNode;
    }
}

```

```

    }

    if (flip) {
        // process left child before right child
        boolean left = printLevel(root.left, level - 1, flip);
        boolean right = printLevel(root.right, level - 1, flip);
        return left || right;
    } else {
        // process right child before left child
        boolean right = printLevel(root.right, level - 1, flip);
        boolean left = printLevel(root.left, level - 1, flip);
        return right || left;
    }
}

```

Code 13.37

```

private void printSpiralTwoStacks(Node node) {
    if (node == null) {
        return;
    }

    // create two stacks to store alternate levels
    Stack<Node> rl = new Stack<>(); // right to left
    Stack<Node> lr = new Stack<>(); // left to right
    // Push first level to first stack 'rl'
    rl.push(node);

    // print while any of the stacks has nodes
    while (!rl.empty() || !lr.empty()) {
        // print nodes of the current level from 'rl'
        // and push nodes of next level to 'lr'
        while (!rl.empty()) {
            Node temp = rl.peek();
            rl.pop();

```

```

        System.out.print(temp.element + " ");
        if (temp.right != null) {
            lr.push(temp.right);
        }
        if (temp.left != null) {
            lr.push(temp.left);
        }
    }
    // print nodes of the current level from 'lr'
    // and push nodes of next level to 'rl'
    while (!lr.empty()) {
        Node temp = lr.peek();
        lr.pop();
        System.out.print(temp.element + " ");
        if (temp.left != null) {
            rl.push(temp.left);
        }
        if (temp.right != null) {
            rl.push(temp.right);
        }
    }
}

```

Code 13.38

```

private void leafDistance(Node node,
    List<Node> pathToLeaf, Set<Node> nodesAtDist, int dist) {
    if (node == null) {
        return;
    }
    // for each leaf node, store the node at distance 'dist'

```

```

        if (isLeaf(node) && pathToLeaf.size() >= dist) {
            nodesAtDist.add(pathToLeaf.get(pathToLeaf.size() - dist));
            return;
        }

        // add the current node into the current path
        pathToLeaf.add(node);

        // go to left and right subtree via recursion
        leafDistance(node.left, pathToLeaf, nodesAtDist, dist);
        leafDistance(node.right, pathToLeaf, nodesAtDist, dist);
        // remove the current node from the current path
        pathToLeaf.remove(node);
    }

    private boolean isLeaf(Node node) {
        return (node.left == null && node.right == null);
    }

```

Code 13.39

```

public boolean findPairSum(int sum) {
    return findPairSum(root, sum, new HashSet());
}

private static boolean findPairSum(Node node,
        int sum, Set<Integer> set) {
    // base case
    if (node == null) {
        return false;
    }

    // find the pair in the left subtree
    if (findPairSum(node.left, sum, set)) {
        return true;
    }

    // if pair is formed with current node then print the pair

```

```

if (set.contains(sum - node.element)) {
    System.out.print("Pair (" + (sum - node.element) + ", "
        + node.element + ") = " + sum);
    return true;
} else {
    set.add(node.element);
}
// find the pair in the right subtree
return findPairSum(node.right, sum, set);
}

```

Code 13.40

```

public boolean findPairSumTwoStacks(int sum) {
    return findPairSumTwoStacks(root, sum);
}

private static boolean findPairSumTwoStacks(
    Node node, int sum) {
    Stack<Node> fio = new Stack<>(); // fio - Forward In-Order
    Stack<Node> rio = new Stack<>(); // rio - Reverse In-Order
    Node minNode = node;
    Node maxNode = node;
    while (!fio.isEmpty() || !rio.isEmpty()
        || minNode != null || maxNode != null) {
        if (minNode != null || maxNode != null) {
            if (minNode != null) {
                fio.push(minNode);
                minNode = minNode.left;
            }
            if (maxNode != null) {
                rio.push(maxNode);
                maxNode = maxNode.right;
            }
        }
        if (fio.isEmpty() && rio.isEmpty())
            return false;
        if (fio.peek().element + rio.peek().element == sum)
            return true;
        if (fio.peek().element + rio.peek().element < sum)
            rio.pop();
        else
            fio.pop();
    }
    return false;
}

```

```

    }

} else {

    int elem1 = fio.peek().element;
    int elem2 = rio.peek().element;
    if (fio.peek() == rio.peek()) {
        break;
    }
    if ((elem1 + elem2) == sum) {
        System.out.print("\nPair (" + elem1 + ", "
            + elem2 + ") = " + sum);
        return true;
    }
    if ((elem1 + elem2) < sum) {
        minNode = fio.pop();
        minNode = minNode.right;
    } else {
        maxNode = rio.pop();
        maxNode = maxNode.left;
    }
}
return false;
}

```

Code 13.41

```

private void verticalSum(Node root,
    Map<Integer, Integer> map, int dist) {
    if (root == null) {
        return;
    }
    if (!map.containsKey(dist)) {

```

```

        map.put(dist, 0);

    }

    map.put(dist, map.get(dist) + root.element);

    // or in functional-style

    /*
     BiFunction <Integer, Integer, Integer> distFunction
     = (distOld, distNew) -> distOld + distNew;
     map.merge(dist, root.element, distFunction);
     */

    // decrease horizontal distance by 1 and go to left
    verticalSum(root.left, map, dist - 1);

    // increase horizontal distance by 1 and go to right
    verticalSum(root.right, map, dist + 1);

}

```

Code 13.42

```

public static void convertToMinHeap(int[] maxHeap) {

    // build heap from last node to all

    // the way up to the root node

    int p = (maxHeap.length - 2) / 2;

    while (p >= 0) {

        heapifyMin(maxHeap, p--, maxHeap.length);

    }

}

// heapify the node at index p and its two direct children

private static void heapifyMin(int[] maxHeap,
                               int p, int size) {

    // get left and right child of node at index p

    int left = leftChild(p);

    int right = rightChild(p);

```

```

int smallest = p;

// compare maxHeap[p] with its left and
// right child and find the smallest value
if ((left < size) && (maxHeap[left] < maxHeap[p])) {
    smallest = left;
}

if ((right < size)
    && (maxHeap[right] < maxHeap[smallest])) {
    smallest = right;
}

// swap 'smallest' with 'p' and heapify
if (smallest != p) {
    swap(maxHeap, p, smallest);
    heapifyMin(maxHeap, smallest, size);
}
}

/* Helper methods */

private static int leftChild(int parentIndex) {
    return (2 * parentIndex + 1);
}

private static int rightChild(int parentIndex) {
    return (2 * parentIndex + 2);
}

// utility function to swap two indices in the array
private static void swap(int heap[], int i, int j) {
    int aux = heap[i];
    heap[i] = heap[j];
    heap[j] = aux;
}

```

Code 13.43

```

private boolean isSymmetricRecursive(
    Node leftNode, Node rightNode) {
    boolean result = false;
    // empty trees are symmetric
    if (leftNode == null && rightNode == null) {
        result = true;
    }
    // conditions 1, 2, and 3 from above
    if (leftNode != null && rightNode != null) {
        result = (leftNode.element.equals(rightNode.element))
            && isSymmetricRecursive(leftNode.left, rightNode.right)
            && isSymmetricRecursive(leftNode.right, rightNode.left);
    }
    return result;
}

```

Code 13.44

```

public boolean isSymmetricIterative() {
    boolean result = false;
    Queue<Node> queue = new LinkedList<>();
    queue.offer(root.left);
    queue.offer(root.right);
    while (!queue.isEmpty()) {
        Node left = queue.poll();
        Node right = queue.poll();
        if (left == null && right == null) {
            result = true;
        } else if (left == null || right == null
            || left.element != right.element) {
            result = false;
            break;
        }
    }
    return result;
}

```

```

    } else {
        queue.offer(left.left);
        queue.offer(right.right);
        queue.offer(left.right);
        queue.offer(right.left);
    }
}

return result;
}

```

Code 13.45

```

public int minimumCost(int[] ropeLength) {
    if (ropeLength == null) {
        return -1;
    }

    // add the lengths of the ropes to the heap
    for (int i = 0; i < ropeLength.length; i++) {
        add(ropeLength[i]);
    }

    int totalLength = 0;
    while (size() > 1) {
        int l1 = poll();
        int l2 = poll();
        totalLength += (l1 + l2);
        add(l1 + l2);
    }

    return totalLength;
}

```

Chapter 14

Images

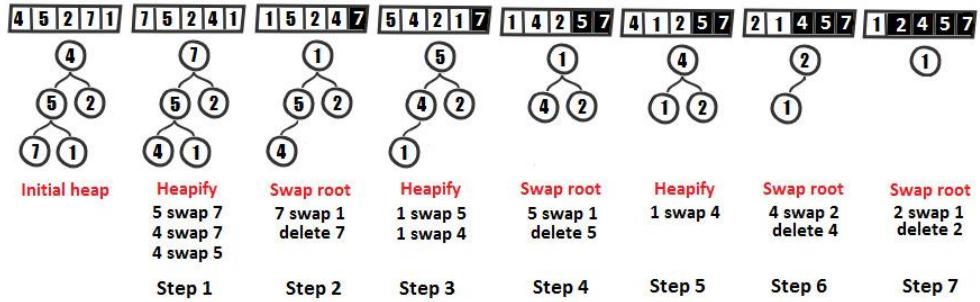


Figure 14.1 – Heap Sort

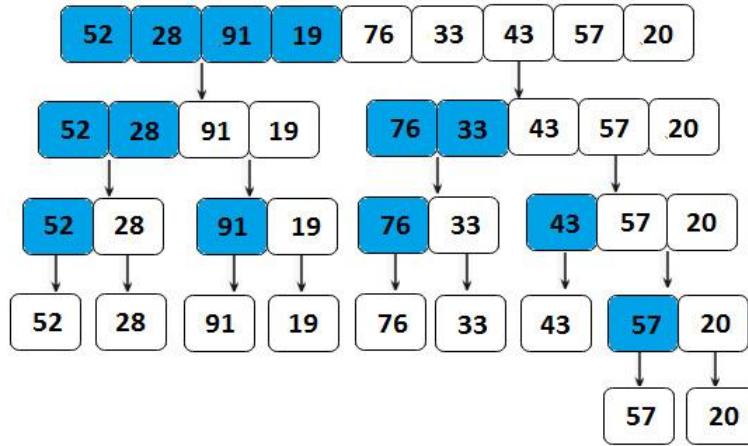


Figure 14.2 – Splitting the given array in the Merge Sort algorithm

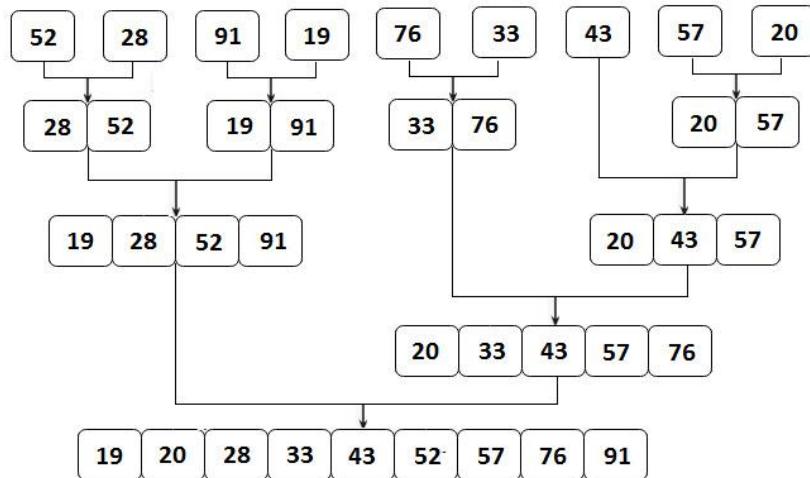


Figure 14.3 – Merging operation for Merge Sort

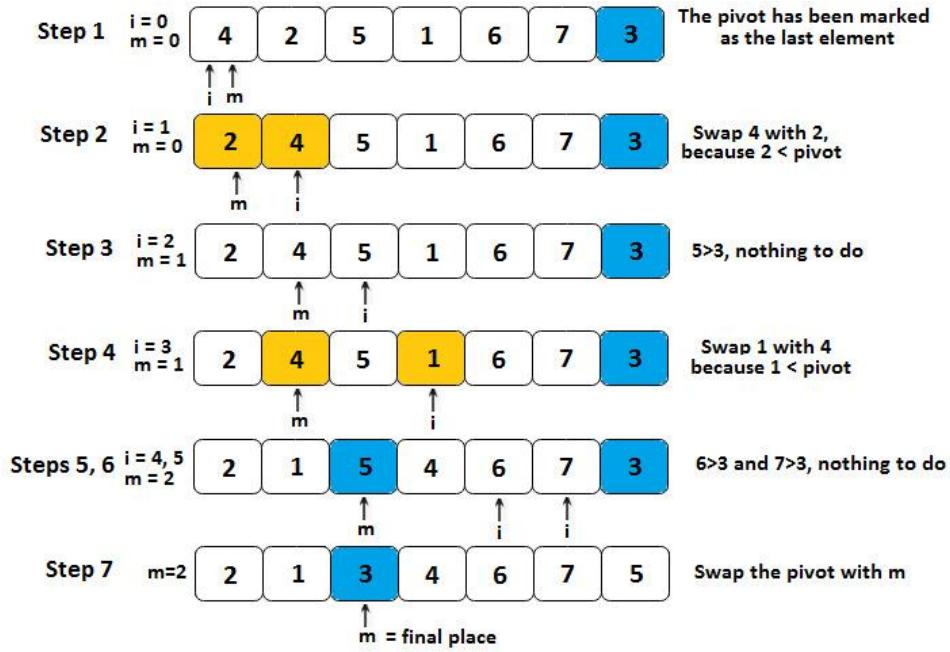


Figure 14.4 – Quick Sort

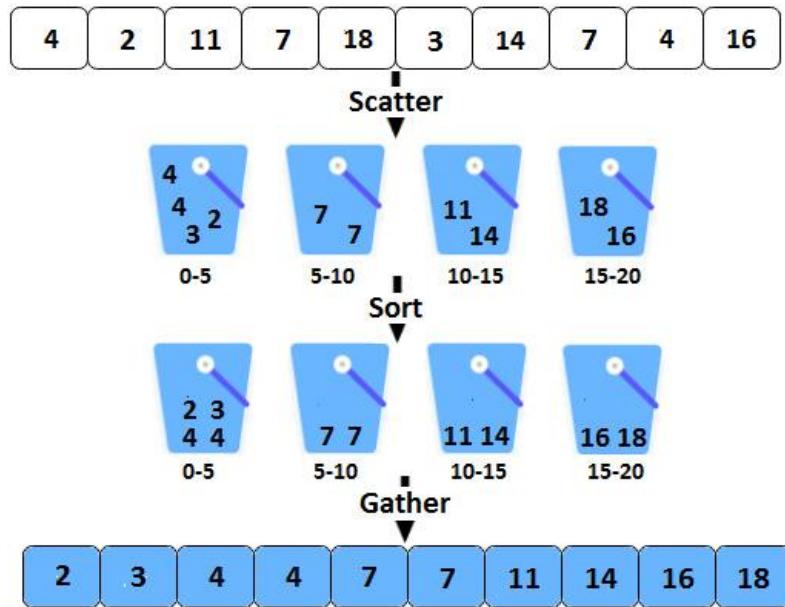


Figure 14.5 – Bucket Sort via the scatter-sort-gather approach

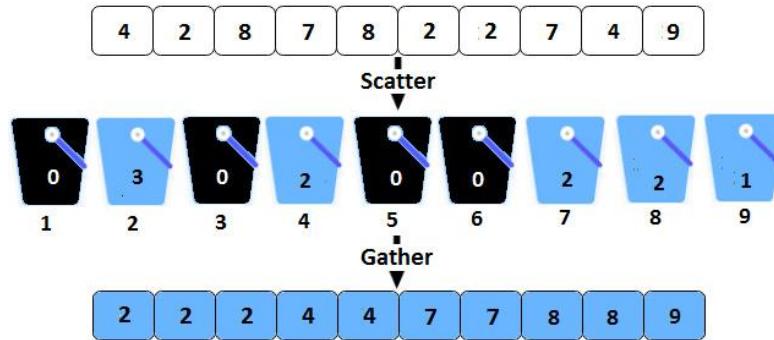


Figure 14.6 – Bucket Sort via the scatter-gather approach

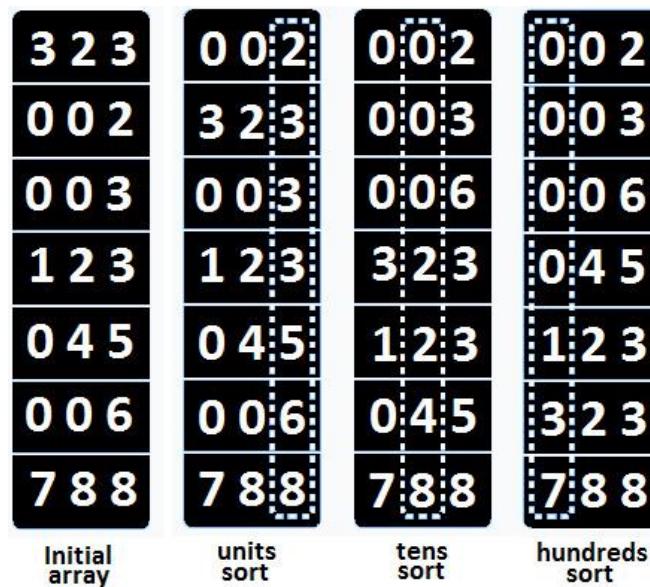


Figure 14.7 – Radix Sort

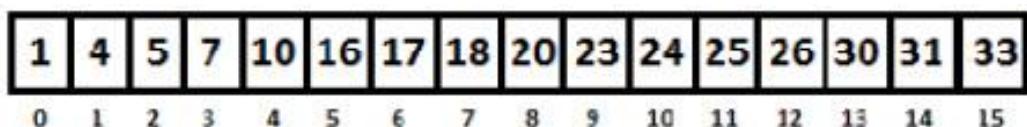


Figure 14.8 – Ordered array containing 16 elements

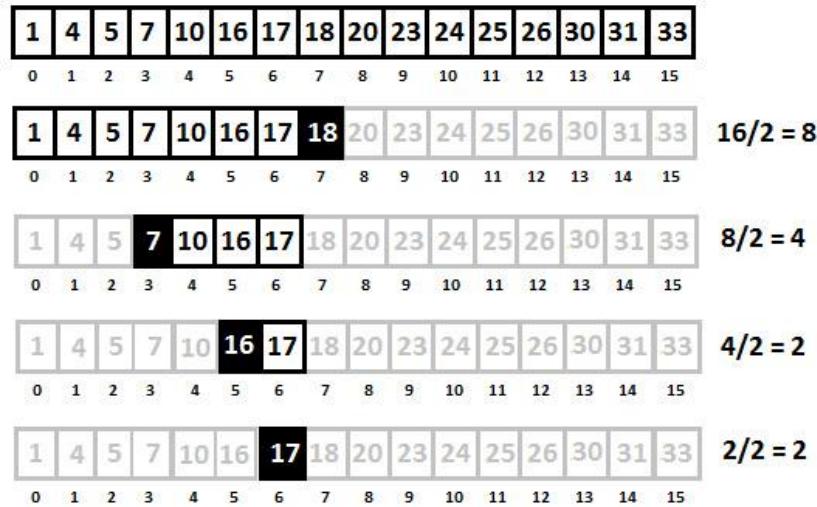


Figure 14.9 – The Binary Search algorithm

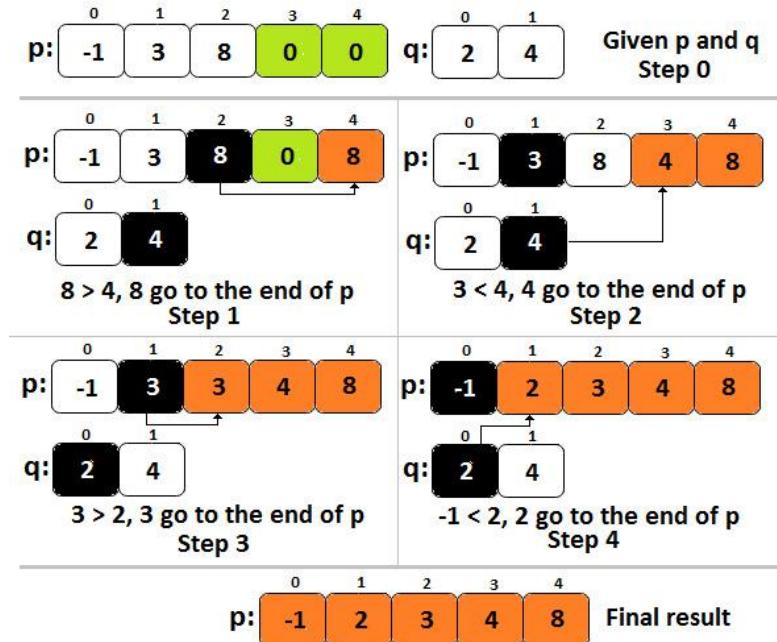


Figure 14.10 – Merging two sorted arrays

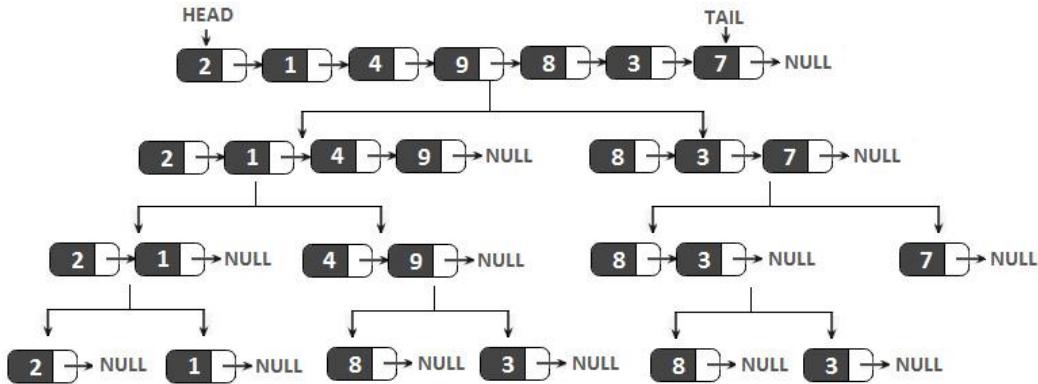


Figure 14.11 – Using divide and conquer on a linked list

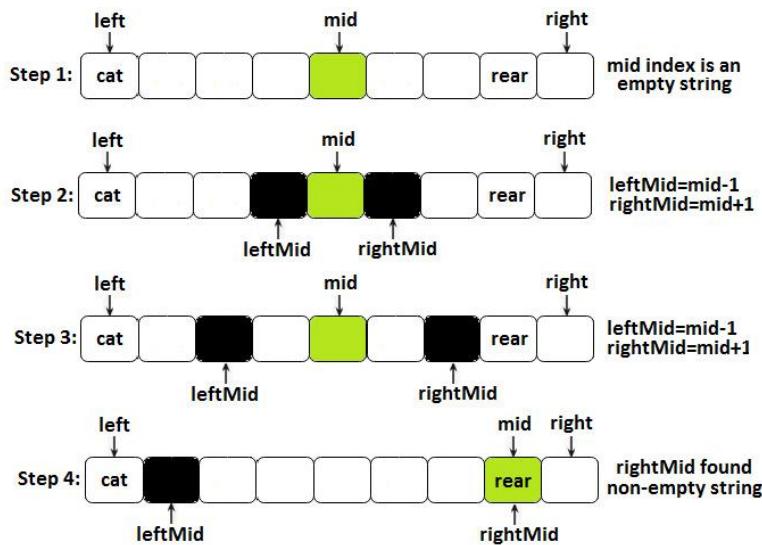


Figure 14.12 – Computing the middle point in the case of an empty string



Figure 14.13 – The given queue and the extra queue

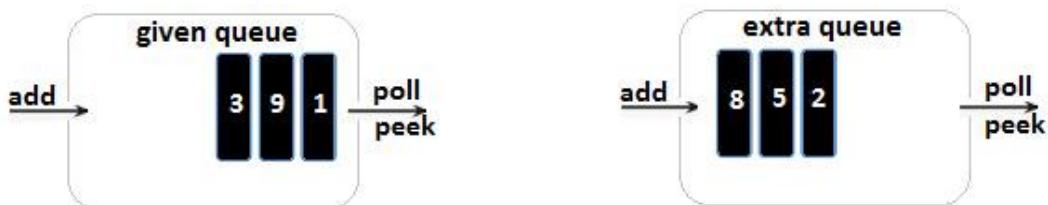


Figure 14.14 – Enqueuing 2, 5, and 8 in the extra queue



Figure 14.15 – Dequeueing and enqueueing 1 in the given queue



Figure 14.16 – Enqueueing 9 in the extra queue



Figure 14.17 – Dequeueing and enqueueing 3 in the given queue

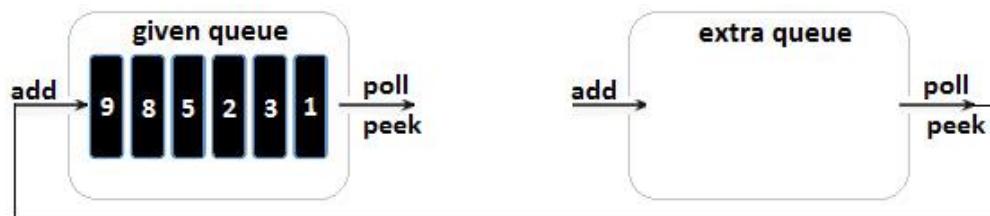


Figure 14.18 – Dequeueing from the extra queue and enqueueing in the given queue

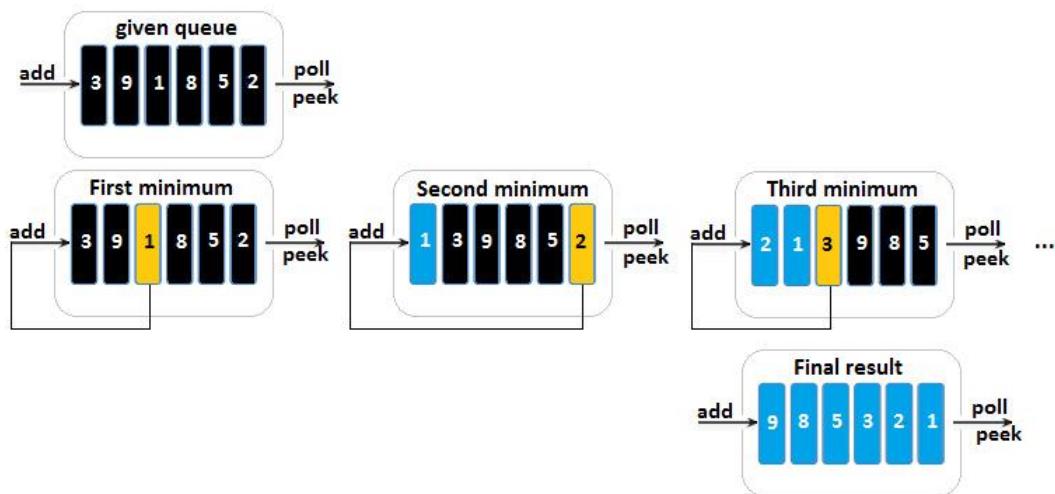


Figure 14.19 – Sorting a queue without extra space

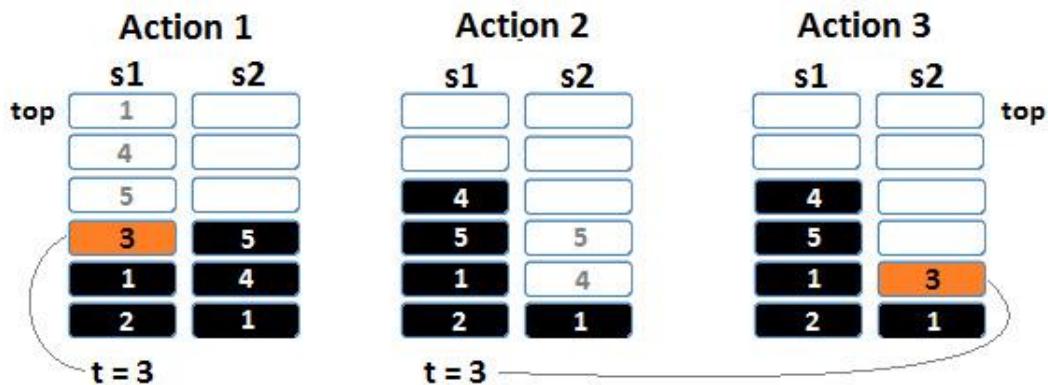


Figure 14.20 – Sorting a stack

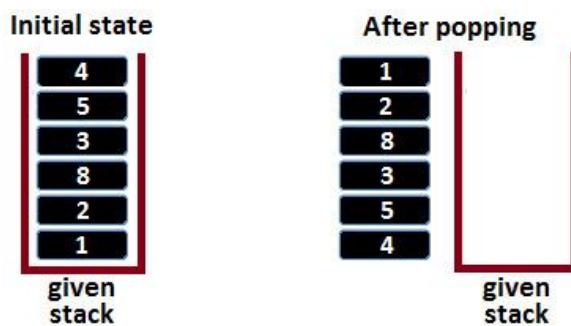


Figure 14.21 – Sorting the stack in place (1)

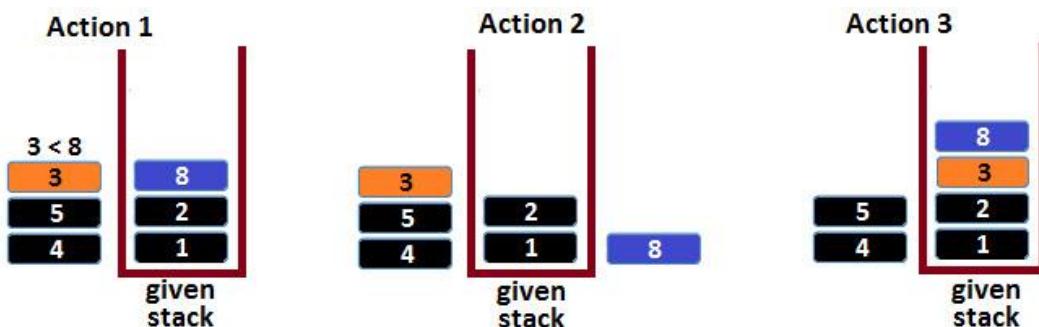


Figure 14.22 – Sorting the stack in place (2)

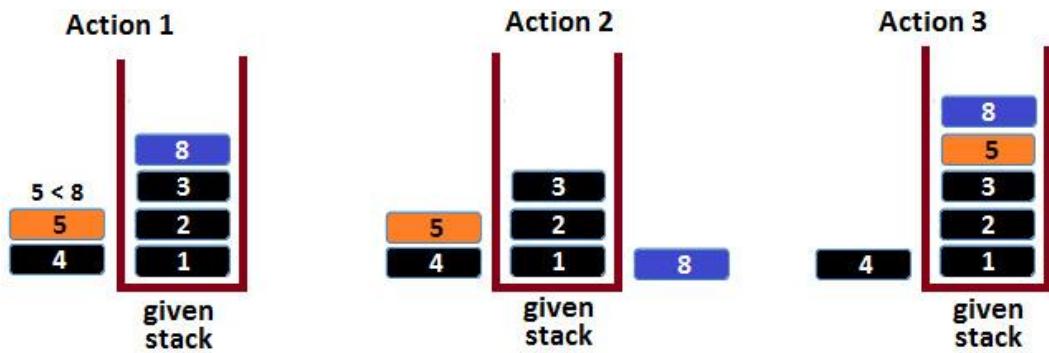


Figure 14.23 – Sorting the stack in place (3)

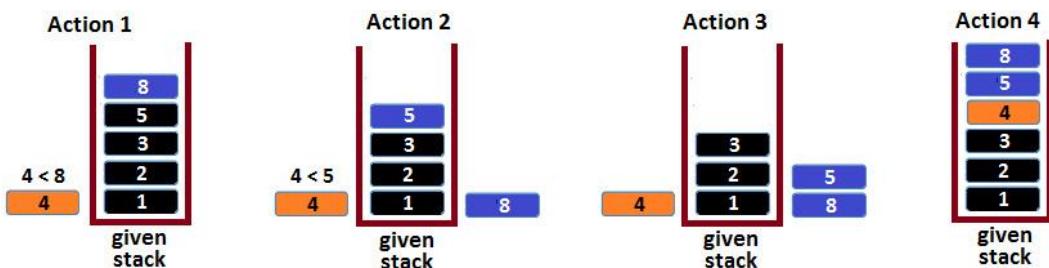


Figure 14.24 – Sorting the stack in place (4)

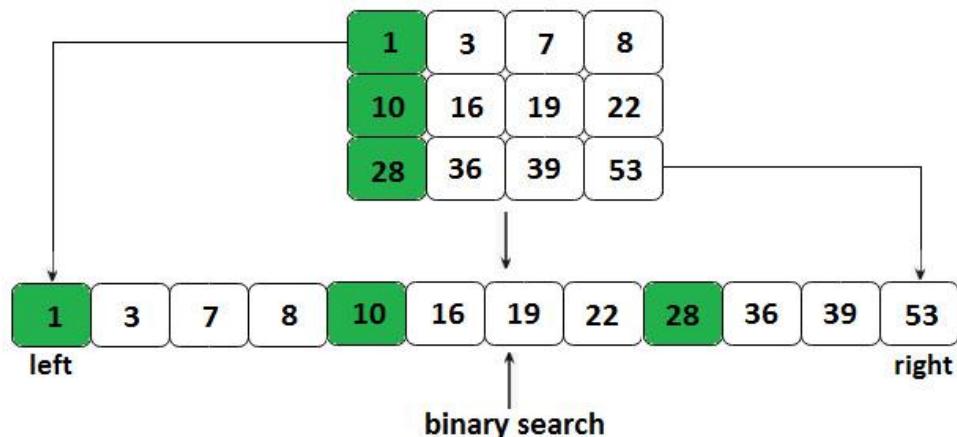


Figure 14.25 – Fully sorted matrix as an array

	0	1	2	3	4	5
0	11	22	48	77	78	84
1	12	24	55	78	83	90
2	25	56	58	80	85	95
3	33	57	60	85	86	99

Figure 14.26 – Searching in a sorted matrix

	0	1	2	3	4	5
0	11	22	48	77	78	84
1	12	24	55	78	83	90
2	25	56	58	80	85	95
3	33	57	60	85	86	99

Figure 14.27 – Path to the solution

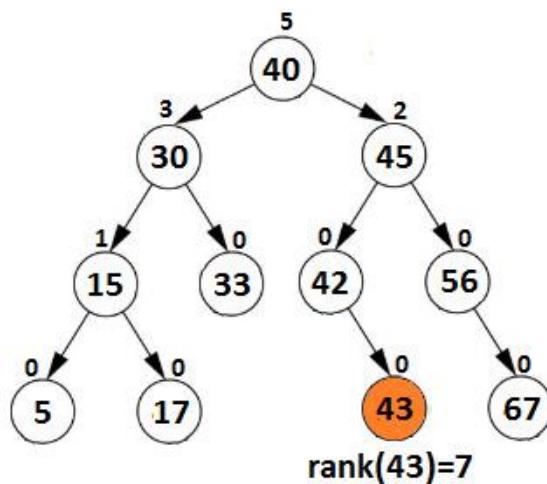


Figure 14.28 – BST for stream ranking

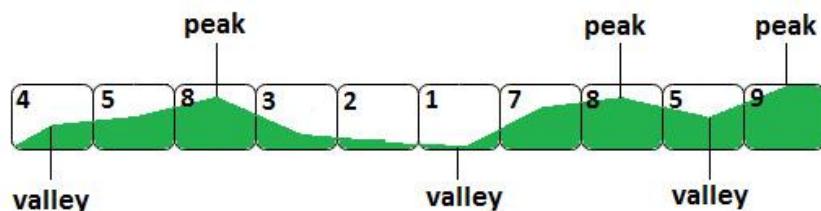


Figure 14.29 – Given array of terrain elevations

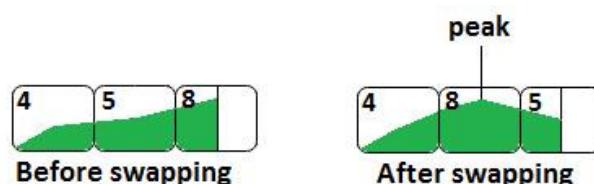


Figure 14.30 – Swapping 5 with 8

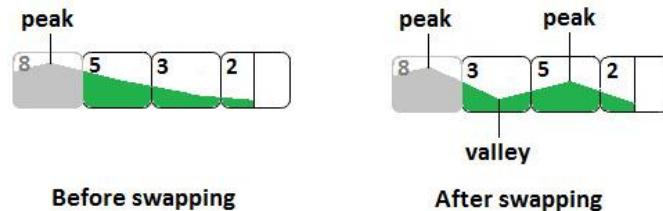


Figure 14.31 – Swapping 3 with 5

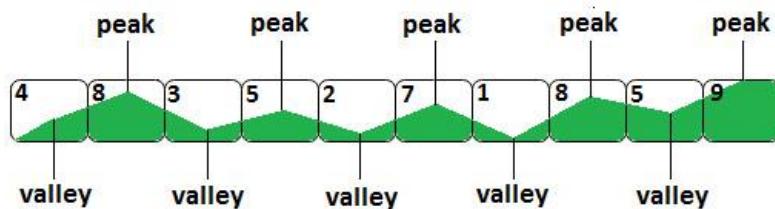


Figure 14.32 – Final result

0	1	2	0	1	2	0	1	2
T	A	C	T	A	C	T	A	C
A	B	L	A	B	L	A	B	L
X	I	E	X	I	E	X	I	E

Figure 14.33 – Board sample

Code

Code 14.1

```
public static void sort(int[] arr) {
    int n = arr.length;
    buildHeap(arr, n);
    while (n > 1) {
        swap(arr, 0, n - 1);
        n--;
        heapify(arr, n, 0);
    }
}
```

```

private static void buildHeap(int[] arr, int n) {
    for (int i = arr.length / 2; i >= 0; i--) {
        heapify(arr, n, i);
    }
}

private static void heapify(int[] arr, int n, int i) {
    int left = i * 2 + 1;
    int right = i * 2 + 2;
    int greater;
    if (left < n && arr[left] > arr[i]) {
        greater = left;
    } else {
        greater = i;
    }
    if (right < n && arr[right] > arr[greater]) {
        greater = right;
    }
    if (greater != i) {
        swap(arr, i, greater);
        heapify(arr, n, greater);
    }
}

private static void swap(int[] arr, int x, int y) {
    int temp = arr[x];
    arr[x] = arr[y];
    arr[y] = temp;
}

```

Code 14.2

```

public static void sort(int[] arr) {
    if (arr.length > 1) {

```

```

        int[] left = leftHalf(arr);
        int[] right = rightHalf(arr);
        sort(left);
        sort(right);
        merge(arr, left, right);
    }

}

private static int[] leftHalf(int[] arr) {
    int size = arr.length / 2;
    int[] left = new int[size];
    System.arraycopy(arr, 0, left, 0, size);
    return left;
}

private static int[] rightHalf(int[] arr) {
    int size1 = arr.length / 2;
    int size2 = arr.length - size1;
    int[] right = new int[size2];
    for (int i = 0; i < size2; i++) {
        right[i] = arr[i + size1];
    }
    return right;
}

```

Code 14.3

```

private static void merge(int[] result,
    int[] left, int[] right) {
    int t1 = 0;
    int t2 = 0;
    for (int i = 0; i < result.length; i++) {
        if (t2 >= right.length
            || (t1 < left.length && left[t1] <= right[t2])) {

```

```

        result[i] = left[t1];
        t1++;
    } else {
        result[i] = right[t2];
        t2++;
    }
}
}
}

```

Code 14.4

```

sort(array, left, right)
if left < right
    m = partition(array, left, right)
    sort(array, left, m-1)
    sort(array, m+1, right)
end
partition(array, left, right)
pivot = array[right]
m = left
for i = m to right-1
    if array[i] <= pivot
        swap array[i] with array[m]
        m=m+1
    end
end
swap array[m] with array[right]
return m
end

```

Code 14.5

```
public static void sort(int[] arr, int left, int right) {
```

```

    if (left < right) {
        int m = partition(arr, left, right);
        sort(arr, left, m - 1);
        sort(arr, m + 1, right);
    }
}

private static int partition(int[] arr, int left, int right) {
    int pivot = arr[right];
    int m = left;
    for (int i = m; i < right; i++) {
        if (arr[i] <= pivot) {
            swap(arr, i, m++);
        }
    }
    swap(arr, right, m);
    return m;
}

```

Code 14.6

```

sort(array)
    create N buckets each of which can hold a range of elements
    for all the buckets
        initialize each bucket with 0 values
    for all the buckets
        put elements into buckets matching the range
    for all the buckets
        sort elements in each bucket
        gather elements from each bucket
end

```

Code 14.7

```
/* Scatter-Sort-Gather approach */
```

```

public static void sort(int[] arr) {
    // get the hash codes
    int[] hashes = hash(arr);
    // create and initialize buckets
    List<Integer>[] buckets = new List[hashes[1]];
    for (int i = 0; i < hashes[1]; i++) {
        buckets[i] = new ArrayList();
    }
    // scatter elements into buckets
    for (int e : arr) {
        buckets[hash(e, hashes)].add(e);
    }
    // sort each bucket
    for (List<Integer> bucket : buckets) {
        Collections.sort(bucket);
    }
    // gather elements from the buckets
    int p = 0;
    for (List<Integer> bucket : buckets) {
        for (int j : bucket) {
            arr[p++] = j;
        }
    }
}

```

Code 14.8

```

sort(array)
create N buckets each of which can track a
    counter of a single element
for all the buckets
    initialize each bucket with 0 values

```

```

for all the buckets

    put elements into buckets matching a single

        element per bucket

for all the buckets

    gather elements from each bucket

end

```

Code 14.9

```

/* Scatter-Gather approach */

public static void sort(int[] arr) {

    // get the maximum value of the given array
    int max = arr[0];

    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    // create max buckets
    int[] bucket = new int[max + 1];
    // the bucket[] is automatically initialized with 0s,
    // therefore this step is redundant
    for (int i = 0; i < bucket.length; i++) {
        bucket[i] = 0;
    }

    // scatter elements in buckets
    for (int i = 0; i < arr.length; i++) {
        bucket[arr[i]]++;
    }

    // gather elements from the buckets
    int p = 0;
    for (int i = 0; i < bucket.length; i++) {

```

```

        for (int j = 0; j < bucket[i]; j++) {
            arr[p++] = i;
        }
    }
}

```

Code 14.10

```

public static void sort(int[] arr, int radix) {
    int min = arr[0];
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < min) {
            min = arr[i];
        } else if (arr[i] > max) {
            max = arr[i];
        }
    }
    int exp = 1;
    while ((max - min) / exp >= 1) {
        countSortByDigit(arr, radix, exp, min);
        exp *= radix;
    }
}

private static void countSortByDigit(
    int[] arr, int radix, int exp, int min) {
    int[] buckets = new int[radix];
    for (int i = 0; i < radix; i++) {
        buckets[i] = 0;
    }
    int bucket;
    for (int i = 0; i < arr.length; i++) {

```

```

        bucket = (int) (((arr[i] - min) / exp) % radix);
        buckets[bucket]++;
    }

    for (int i = 1; i < radix; i++) {
        buckets[i] += buckets[i - 1];
    }

    int[] out = new int[arr.length];
    for (int i = arr.length - 1; i >= 0; i--) {
        bucket = (int) (((arr[i] - min) / exp) % radix);
        out[--buckets[bucket]] = arr[i];
    }

    System.arraycopy(out, 0, arr, 0, arr.length);
}

```

Code 14.11

```

search 17 in {1, 4, 5, 7, 10, 16, 17, 18, 20,
              23, 24, 25, 26, 30, 31, 33}

compare 17 to 18 -> 17 < 18

search 17 in {1, 4, 5, 7, 10, 16, 17, 18}

    compare 17 to 7 -> 17 > 7

        search 17 in {7, 10, 16, 17}

            compare 17 to 16 -> 17 > 16

                search 17 in {16, 17}

                    compare 17 to 17

                    return

```

Code 14.12

```

public static int runIterative(int[] arr, int p) {
    // the search space is the whole array
    int left = 0;
    int right = arr.length - 1;
    // while the search space has at least one element

```

```

while (left <= right) {
    // half the search space
    int mid = (left + right) / 2;
    // if domain overflow can happen then use:
    // int mid = left + (right - left) / 2;
    // int mid = right - (right - left) / 2;
    // we found the searched element
    if (p == arr[mid]) {
        return mid;
    } // discard all elements in the right of the
    // search space including 'mid'
    else if (p < arr[mid]) {
        right = mid - 1;
    } // discard all elements in the left of the
    // search space including 'mid'
    else {
        left = mid + 1;
    }
}
// by convention, -1 means element not found into the array
return -1;
}

```

Code 14.13

```

public static void merge(int[] p, int[] q) {
    int pLast = p.length - q.length;
    int qLast = q.length;
    if (pLast < 0) {
        throw new IllegalArgumentException("p cannot fit q");
    }
    int pIdx = pLast - 1;

```

```

int qIdx = qLast - 1;
int mIdx = pLast + qLast - 1;
// merge p and q
// start from the last element in p and q
while (qIdx >= 0) {
    if (pIdx >= 0 && p[pIdx] > q[qIdx]) {
        p[mIdx] = p[pIdx];
        pIdx--;
    } else {
        p[mIdx] = q[qIdx];
        qIdx--;
    }
    mIdx--;
}
}

```

Code 14.14

```

String[] words = {
    "calipers", "caret", "slat", "cater", "thickset",
    "spiracle", "trace", "last", "salt", "bowel", "crate",
    "loop", "polo", "thickest", "below", "thickets",
    "pool", "elbow", "replicas"
};

```

Code 14.15

```

// helper method for sorting the chars of a word
private static String sortWordChars(String word) {
    char[] wordToChar = word.toCharArray();
    Arrays.sort(wordToChar);
    return String.valueOf(wordToChar);
}

```

Code 14.16

```
public class Anagrams implements Comparator<String> {  
    @Override  
    public int compare(String s1, String s2) {  
        return sortStringChars(s1).compareTo(sortStringChars(s2));  
    }  
}
```

Code 14.8

```
Arrays.sort(words, new Anagrams());
```

Code 14.9

```
/* Group anagrams via hashing (O(nm log m) */  
public void printAnagrams(String words[]) {  
    Map<String, List<String>> result = new HashMap<>();  
    for (int i = 0; i < words.length; i++) {  
        // sort the chars of each string  
        String word = words[i];  
        String sortedWord = sortWordChars(word);  
        if (result.containsKey(sortedWord)) {  
            result.get(sortedWord).add(word);  
        } else {  
            // start a new group of anagrams  
            List<String> anagrams = new ArrayList<>();  
            anagrams.add(word);  
            result.put(sortedWord, anagrams);  
        }  
    }  
    // print the result  
    System.out.println(result.values());  
}
```

Code 14.17

```
/* Group anagrams via hashing (O(nm) */
```

```

public void printAnagramsOptimized(String[] words) {
    Map<String, List<String>> result = new HashMap<>();
    for (int i = 0; i < words.length; i++) {
        String word = words[i];
        char[] wordToChar = new char[RANGE_a_z];
        // count up the number of occurrences (frequency)
        // of each letter in 'word'
        for (int j = 0; j < word.length(); j++) {
            wordToChar[word.charAt(j) - 'a']++;
        }
        String computedWord = String.valueOf(wordToChar);
        if (result.containsKey(computedWord)) {
            result.get(computedWord).add(word);
        } else {
            List<String> anagrams = new ArrayList<>();
            anagrams.add(word);
            result.put(computedWord, anagrams);
        }
    }
    System.out.println(result.values());
}

```

Code 14.18

```

public class SizelessList {
    private final int[] arr;
    public SizelessList(int[] arr) {
        this.arr = arr.clone();
    }
    public int peekAt(int index) {
        if (index >= arr.length) {
            return -1;
        }
    }
}

```

```
    }

    return arr[index];
}

}
```

Code 14.19

```
public static int search(SizelessList sl, int element) {

    int index = 1;

    while (sl.peekAt(index) != -1
        && sl.peekAt(index) < element) {

        index *= 2;
    }

    return binarySearch(sl, element, index / 2, index);
}

private static int binarySearch(SizelessList sl,
    int element, int left, int right) {

    int mid;

    while (left <= right) {

        mid = (left + right) / 2;

        int middle = sl.peekAt(mid);

        if (middle > element || middle == -1) {

            right = mid - 1;
        } else if (middle < element) {

            left = mid + 1;
        } else {

            return mid;
        }
    }

    return -1;
}
```

Code 14.20

```

// Divide the given linked list in two equal sub-lists.

// If the length of the given linked list is odd,
// the extra node will go in the first sub-list

private Node[] divide(Node sourceNode) {

    // length is less than 2

    if (sourceNode == null || sourceNode.next == null) {
        return new Node[]{sourceNode, null};
    }

    Node fastRunner = sourceNode.next;
    Node slowRunner = sourceNode;

    // advance 'firstRunner' two nodes,
    // and advance 'secondRunner' one node

    while (fastRunner != null) {
        fastRunner = fastRunner.next;

        if (fastRunner != null) {
            slowRunner = slowRunner.next;
            fastRunner = fastRunner.next;
        }
    }

    // 'secondRunner' is just before the middle point
    // in the list, so split it in two at that point

    Node[] headsOfSublists = new Node[]{
        sourceNode, slowRunner.next};

    slowRunner.next = null;
    return headsOfSublists;
}

```

Code 14.21

```

// sort the given linked list via the Merge Sort algorithm

public void sort() {
    head = sort(head);

```

```
}

private Node sort(Node head) {
    if (head == null || head.next == null) {
        return head;
    }

    // split head into two sublists
    Node[] headsOfSublists = divide(head);
    Node head1 = headsOfSublists[0];
    Node head2 = headsOfSublists[1];
    // recursively sort the sublists
    head1 = sort(head1);
    head2 = sort(head2);

    // merge the two sorted lists together
    return merge(head1, head2);
}

// takes two lists sorted in increasing order, and merge
// their nodes together (which is returned)
private Node merge(Node head1, Node head2) {
    if (head1 == null) {
        return head2;
    } else if (head2 == null) {
        return head1;
    }

    Node merged;
    // pick either 'head1' or 'head2'
    if (head1.data <= head2.data) {
        merged = head1;
        merged.next = merge(head1.next, head2);
    } else {
        merged = head2;
    }
}
```

```

        merged.next = merge(head1, head2.next);
    }

    return merged;
}

```

Code 14.22

```

public static int search(String[] stringsArr, String str) {
    return search(stringsArr, str, 0, stringsArr.length - 1);
}

private static int search(String[] stringsArr,
    String str, int left, int right) {
    if (left > right) {
        return -1;
    }

    int mid = (left + right) / 2;

    // since mid is empty we try to find the
    // closest non-empty string to mid
    if (stringsArr[mid].isEmpty()) {
        int leftMid = mid - 1;
        int rightMid = mid + 1;

        while (true) {
            if (leftMid < left && rightMid > right) {
                return -1;
            } else if (rightMid <= right
                && !stringsArr[rightMid].isEmpty()) {
                mid = rightMid;
                break;
            } else if (leftMid >= left
                && !stringsArr[leftMid].isEmpty()) {
                mid = leftMid;
                break;
            }
        }
    }
}

```

```

        }

        rightMid++;
        leftMid--;
    }

}

if (str.equals(stringsArr[mid])) {
    // the searched string was found
    return mid;
} else if (stringsArr[mid].compareTo(str) < 0) {
    // search to the right
    return search(stringsArr, str, mid + 1, right);
} else {
    // search to the left
    return search(stringsArr, str, left, mid - 1);
}
}
}

```

Code 14.23

```

public static void sort(Queue<Integer> queue) {
    if (queue == null || queue.size() < 2) {
        return;
    }

    // this is the extra queue
    Queue<Integer> extraQueue = new ArrayDeque();
    int count = 0;           // count the processed elements
    boolean sorted = false;   // flag when sorting is done
    int queueSize = queue.size(); // size of the given queue
    int lastElement = queue.peek(); // we start from the front
                                  // of the given queue

    while (!sorted) {
        // Step 1

```

```

if (lastElement <= queue.peek()) {
    lastElement = queue.poll();
    extraQueue.add(lastElement);
} else { // Step 2
    queue.add(queue.poll());
}
// still have elements to process
count++;
if (count != queueSize) {
    continue;
}
// Step 4
if (extraQueue.size() == queueSize) {
    sorted = true;
}
// Step 3
while (extraQueue.size() > 0) {
    queue.add(extraQueue.poll());
    lastElement = queue.peek();
}
count = 0;
}
}

```

Code 14.24

```

public static void sort(Queue<Integer> queue) {
    // traverse the unsorted part of the queue
    for (int i = 1; i <= queue.size(); i++) {
        moveMinToRear(queue, queue.size() - i);
    }
}

```

```

// find (in the unsorted part) the minimum
// element and move this element to the rear of the queue

private static void moveMinToRear(Queue<Integer> queue,
        int sortIndex) {

    int minElement = Integer.MAX_VALUE;
    boolean flag = false;
    int queueSize = queue.size();
    for (int i = 0; i < queueSize; i++) {
        int currentElement = queue.peek();
        // dequeue
        queue.poll();
        // avoid traversing the sorted part of the queue
        if (currentElement <= minElement && i <= sortIndex) {
            // if we found earlier a minimum then
            // we put it back into the queue since
            // we just found a new minimum
            if (flag) {
                queue.add(minElement);
            }
            flag = true;
            minElement = currentElement;
        } else {
            // enqueue the current element which is not the minimum
            queue.add(currentElement);
        }
    }
    // enqueue the minimum element
    queue.add(minElement);
}

```

Code 14.25

```

public static void sort(Stack<Integer> stack) {
    Stack<Integer> auxStack = new Stack<>();
    // Step 1 (a, b and c)
    while (!stack.isEmpty()) {
        int t = stack.pop();
        while (!auxStack.isEmpty() && auxStack.peek() > t) {
            stack.push(auxStack.pop());
        }
        auxStack.push(t);
    }
    // Step 2
    while (!auxStack.isEmpty()) {
        stack.push(auxStack.pop());
    }
}

```

Code 14.26

```

public static void sort(Stack<Integer> stack) {
    // stack is empty (base case)
    if (stack.isEmpty()) {
        return;
    }
    // remove the top element
    int top = stack.pop();
    // apply recursion for the remaining elements in the stack
    sort(stack);
    // insert the popped element back in the sorted stack
    sortedInsert(stack, top);
}

private static void sortedInsert(
    Stack<Integer> stack, int element) {

```

```

// the stack is empty or the element
// is greater than all elements in the stack (base case)
if (stack.isEmpty() || element > stack.peek()) {
    stack.push(element);
    return;
}

// the element is smaller than the top element,
// so remove the top element
int top = stack.pop();
// apply recursion for the remaining elements in the stack
sortedInsert(stack, element);
// insert the popped element back in the stack
stack.push(top);
}

```

Code 14.27

```

public static boolean search(int[][] matrix, int element) {
    int rows = matrix.length;      // number of rows
    int cols = matrix[0].length; // number of columns
    // search space is an array as [0, (rows * cols) - 1]
    int left = 0;
    int right = (rows * cols) - 1;
    // start binary search
    while (left <= right) {
        int mid = (left + right) / 2;
        int midElement = matrix[mid / cols][mid % cols];
        if (element == midElement) {
            return true;
        } else if (element < midElement) {
            right = mid - 1;
        } else {

```

```
    left = mid + 1;

}
}

return false;
}
```

Code 14.28

```
public static boolean search(int[][] matrix, int element) {

    int row = 0;

    int col = matrix[0].length - 1;

    while (row < matrix.length && col >= 0) {

        if (matrix[row][col] == element) {

            return true;

        } else if (matrix[row][col] > element) {

            col--;

        } else {

            row++;

        }

    }

    return false;

}
```

Code 14.29

```
public static int firstOneIndex(int[] arr) {

    if (arr == null) {

        return -1;

    }

    int left = 0;

    int right = arr.length - 1;

    while (left <= right) {

        int middle = 1 + (right - left) / 2;

        if (arr[middle] == 0) {


```

```

        left = middle + 1;
    } else {
        right = middle - 1;
    }
    if (arr[left] == 1) {
        return left;
    }
}
return -1;
}

```

Code 14.30

```

public static int maxDiff(int arr[]) {
    int len = arr.length;
    int maxDiff = arr[1] - arr[0];
    int marker = arr[0];
    for (int i = 1; i < len; i++) {
        if (arr[i] - marker > maxDiff) {
            maxDiff = arr[i] - marker;
        }
        if (arr[i] < marker) {
            marker = arr[i];
        }
    }
    return maxDiff;
}

```

Code 14.31

```

public class Stream {
    private Node root = null;
    private class Node {
        private final int element;

```

```
private int leftTreeSize;  
private Node left;  
private Node right;  
private Node(int element) {  
    this.element = element;  
    this.left = null;  
    this.right = null;  
}  
}  
/* add a new node into the tree */  
public void generate(int element) {  
    if (root == null) {  
        root = new Node(element);  
    } else {  
        insert(root, element);  
    }  
}  
private void insert(Node node, int element) {  
    if (element <= node.element) {  
        if (node.left != null) {  
            insert(node.left, element);  
        } else {  
            node.left = new Node(element);  
        }  
        node.leftTreeSize++;  
    } else {  
        if (node.right != null) {  
            insert(node.right, element);  
        } else {  
            node.right = new Node(element);  
        }  
    }  
}
```

```

        }
    }
}

/* return rank of 'element' */
public int getRank(int element) {
    return getRank(root, element);
}

private int getRank(Node node, int element) {
    if (element == node.element) {
        return node.leftTreeSize;
    } else if (element < node.element) {
        if (node.left == null) {
            return -1;
        } else {
            return getRank(node.left, element);
        }
    } else {
        int rightTreeRank = node.right == null
            ? -1 : getRank(node.right, element);
        if (rightTreeRank == -1) {
            return -1;
        } else {
            return node.leftTreeSize + 1 + rightTreeRank;
        }
    }
}
}

```

Code 14.32

```

public static void sort(int[] arr) {
    for (int i = 1; i < arr.length; i += 2) {

```

```

        int maxFoundIndex = maxElementIndex(arr, i - 1, i, i + 1);

        if (i != maxFoundIndex) {
            swap(arr, i, maxFoundIndex);
        }
    }

private static int maxElementIndex(int[] arr,
int left, int middle, int right) {
    int arrLength = arr.length;
    int leftElement = left >= 0 && left < arrLength
        ? arr[left] : Integer.MIN_VALUE;
    int middleElement = middle >= 0 && middle < arrLength
        ? arr[middle] : Integer.MIN_VALUE;
    int rightElement = right >= 0 && right < arrLength
        ? arr[right] : Integer.MIN_VALUE;
    int maxElement = Math.max(leftElement,
        Math.max(middleElement, rightElement));
    if (leftElement == maxElement) {
        return left;
    } else if (middleElement == maxElement) {
        return middle;
    } else {
        return right;
    }
}

```

Code 14.33

```

public static void leftSmaller(int arr[]) {
    Stack<Integer> stack = new Stack<>();
    // While the top element of the stack is greater than
    // equal to arr[i] remove it from the stack

```

```

for (int i = 0; i < arr.length; i++) {
    while (!stack.empty() && stack.peek() >= arr[i]) {
        stack.pop();
    }
    // if stack is empty there is no left smaller element
    if (stack.empty()) {
        System.out.print("_, ");
    } else {
        // the top of the stack is the left smaller element
        System.out.print(stack.peek() + ", ");
    }
    // push arr[i] into the stack
    stack.push(arr[i]);
}
}

```

Code 14.34

```

int[] firstArr = {4, 1, 8, 1, 3, 8, 6, 7, 4, 9, 8, 2, 5, 3};
int[] secondArr = {7, 4, 8, 11, 2};

```

Code 14.35

```

public static void custom(int[] firstArr, int[] secondArr) {
    // store the frequency of each element of first array
    // using a TreeMap stores the data sorted
    Map<Integer, Integer> frequencyMap = new TreeMap<>();
    for (int i = 0; i < firstArr.length; i++) {
        frequencyMap.putIfAbsent(firstArr[i], 0);
        frequencyMap.put(firstArr[i],
            frequencyMap.get(firstArr[i]) + 1);
    }
    // overwrite elements of first array
    int index = 0;

```

```
for (int i = 0; i < secondArr.length; i++) {
    // if the current element is present in the 'frequencyMap'
    // then set it n times (n is the frequency of
    // that element in the first array)
    int n = frequencyMap.getOrDefault(secondArr[i], 0);
    while (n-- > 0) {
        firstArr[index++] = secondArr[i];
    }
    // remove the element from map
    frequencyMap.remove(secondArr[i]);
}
// copy the remaining elements (the elements that are
// present in the first array but not present
// in the second array)
for (Map.Entry<Integer, Integer> entry :
    frequencyMap.entrySet()) {
    int count = entry.getValue();
    while (count-- > 0) {
        firstArr[index++] = entry.getKey();
    }
}
```

Chapter 15

Images

SYMBOL	VALUE
I	1
IV	4
V	5
IX	9
X	10
XL	40
L	50
XC	90
C	100
CD	400
D	500
CM	900
M	1000

Figure 15.1 – Roman numbers

Digitized by srujanika@gmail.com

Figure 15.2 – All the doors are closed (initial state)

Figure 15.3 – All the doors are opened (step 1)

Figure 15.4 – The even doors are closed and the odd ones are opened (step 2)

Figure 15.5 – The result of applying the third visit (step 3)

$10010000100000010000000010000000001000000000010000000000010000000000001000000000000010000000000000010000000000000001$

$2^2 3^3 4^4 5^5 6^6 7^7 8^8 9^9 10^{10}$

Figure 15.6 – The opened doors are all perfect squares (last visit)

Step 1	111111111111	1	111111111111 ...
Step 2	1010101010101	0	101010101010 ...
Step 3	10001110001	1	1000111000 ...
Step 4	10011111001	0	1001111100 ...
	10010111011	0	1011111100 ...
Step 6	10010011011	1	1011110100 ...
	10010001011	1	1111110101 ...
	10010000011	1	1110101010 ...
	10010000111	1	1110111101 ...
	10010000101	1	1110111111 ...
	10010000100	1	1110111111 ...
Step 12	10010000100	0	1110111111 ...
	10010000100	0	0110111111 ...
	10010000100	0	0010111111 ...
	10010000100	0	0000111111 ...

Figure 15.7 – Door #12 after 15 steps

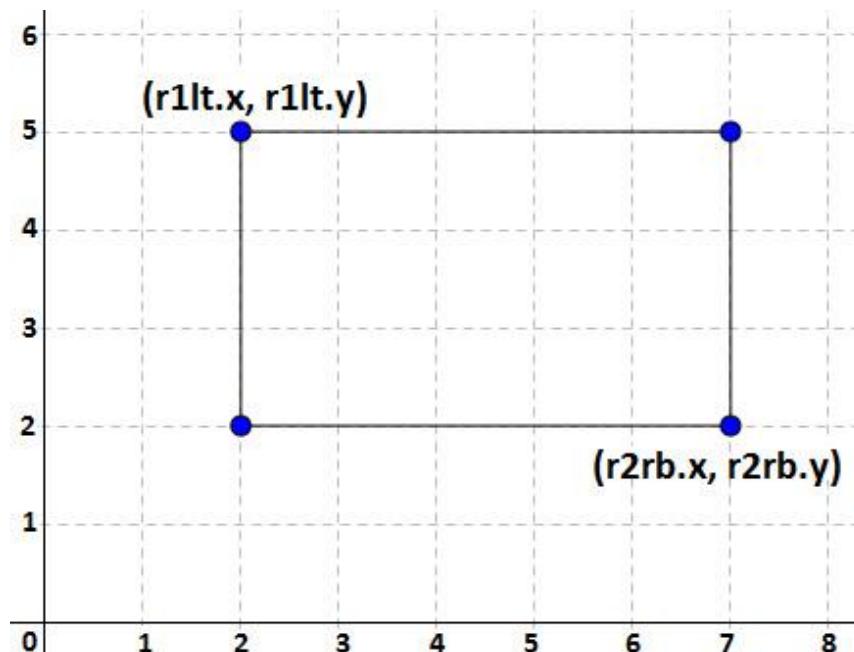


Figure 15.8 – Rectangle coordinates

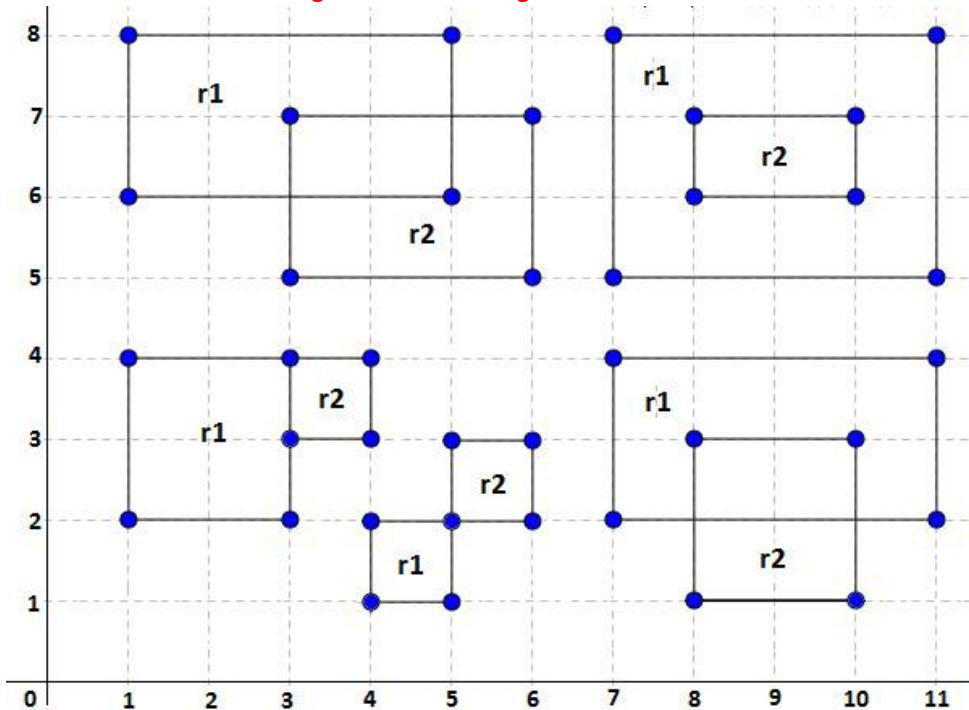


Figure 15.9 – Overlapping rectangles

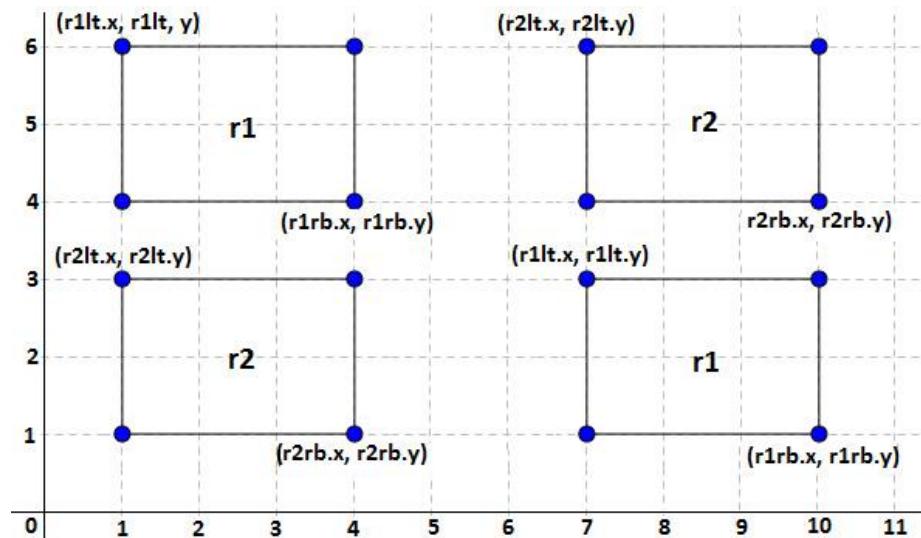


Figure 15.10 – Non-overlapping rectangles

$$\begin{array}{r}
 4145775 \\
 771467 \\
 \hline
 29020425 = 4145775 \times 7 \\
 248746500 = 4145775 \times 60 \\
 1658310000 = 4145775 \times 400 \\
 4145775000 = 4145775 \times 1000 \\
 290204250000 = 4145775 \times 70000 \\
 2902042500000 = 4145775 \times 700000 \\
 \hline
 3198328601925
 \end{array}$$

Figure 15.11 – Multiplying two large numbers

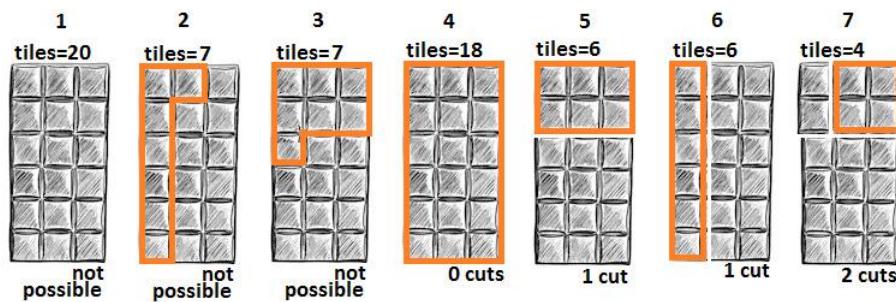


Figure 15.12 – A 3×6 chocolate bar

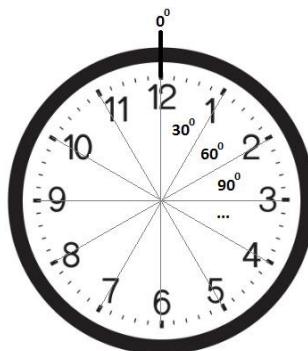


Figure 15.13 – 360 degree split at 12 hours

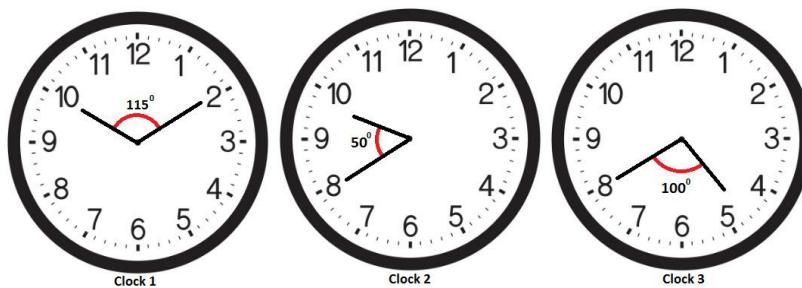


Figure 15.14 – Three clocks

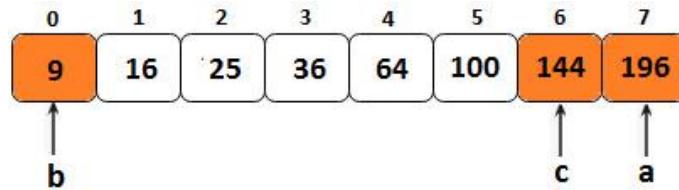


Figure 15.15 – Setting a, b, and c

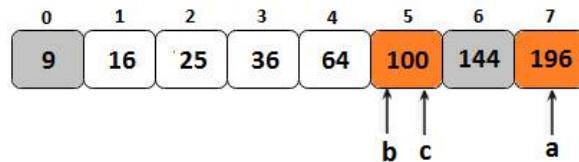


Figure 15.16 – b and c at the end of the loop

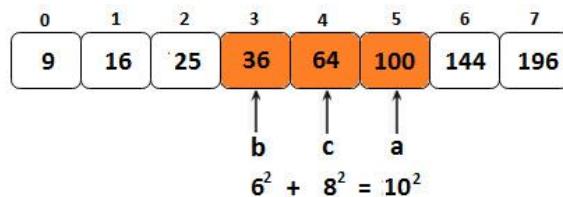


Figure 15.17 – A Pythagorean triplet

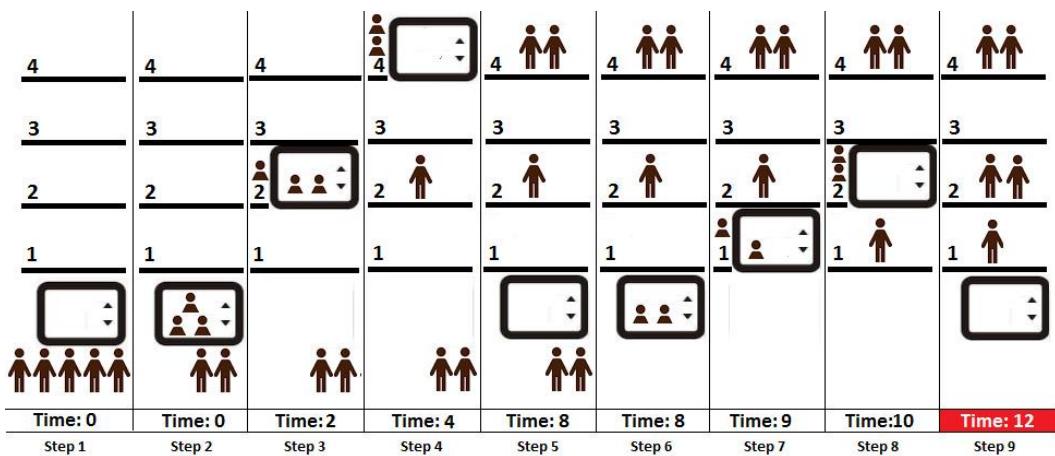


Figure 15.18 – Scheduling an elevator example

Code

Code 15.1

```
public static void print(int n) {  
    for (int i = 1; i <= n; i++) {  
        if (((i % 5) == 0) && ((i % 7) == 0)) { // multiple of  
5&7  
            System.out.println("fizzbuzz");  
        } else if ((i % 5) == 0) { // multiple of 5  
            System.out.println("fizz");  
        } else if ((i % 7) == 0) { // multiple of 7  
            System.out.println("buzz");  
        } else {  
            System.out.println(i); // not a multiple of 5 or 7  
        }  
    }  
}
```

Code 15.2

```
private static final String HUNDREDTHS[]  
= {"", "C", "CC", "CCC", "CD", "D",  
    "DC", "DCC", "DCCC", "CM"};  
  
private static final String TENS[]  
= {"", "X", "XX", "XXX",  
    "XL", "L", "LX", "LXX", "LXXX", "XC"};  
  
private static final String ONES[]  
= {"", "I", "II", "III", "IV", "V",  
    "VI", "VII", "VIII", "IX"};  
  
public static String convert(int n) {  
    String roman = "";  
    // Step 1  
    while (n >= 1000) {
```

```

    roman = roman + 'M';
    n -= 1000;
}

// Step 2

roman = roman + HUNDREDTHS[n / 100];
n = n % 100;

// Step 3

roman = roman + TENS[n / 10];
n = n % 10;

// Step 4

roman = roman + ONES[n];

return roman;
}

```

Code 15.3

```

private static final int DOORS = 100;

public static int[] visitToggle() {
    // 0 - closed door
    // 1 - opened door

    int[] doors = new int[DOORS];
    for (int i = 0; i <= (DOORS - 1); i++) {
        doors[i] = 0;
    }

    for (int i = 0; i <= (DOORS - 1); i++) {
        for (int j = 0; j <= (DOORS - 1); j++) {
            if ((j + 1) % (i + 1) == 0) {
                if (doors[j] == 0) {
                    doors[j] = 1;
                } else {
                    doors[j] = 0;
                }
            }
        }
    }
}

```

```

        }
    }
}

return doors;
}

```

Code 15.4

```

public static int kth(int k) {
    int count3 = 0;
    int count5 = 0;
    int count7 = 0;
    List<Integer> list = new ArrayList<>();
    list.add(1);
    while (list.size() <= k + 1) {
        int m = min(min(list.get(count3) * 3,
                         list.get(count5) * 5),
                    list.get(count7) * 7);
        list.add(m);
        if (m == list.get(count3) * 3) {
            count3++;
        }
        if (m == list.get(count5) * 5) {
            count5++;
        }
        if (m == list.get(count7) * 7) {
            count7++;
        }
    }
    return list.get(k - 1);
}

```

Code 15.5

```

public static int decoding(char[] digits, int n) {

```

```

// base cases
if (n == 0 || n == 1) {
    return 1;
}

// if the digits[] starts with 0 (for example, '0212')
if (digits == null || digits[0] == '0') {
    return 0;
}

int count = 0;
// If the last digit is not 0 then last
// digit must add to the number of words
if (digits[n - 1] > '0') {
    count = decoding(digits, n - 1);
}

// If the last two digits represents a number smaller
// than or equal to 26 then consider last two digits
// and call decoding()
if (digits[n - 2] == '1'
    || (digits[n - 2] == '2' && digits[n - 1] < '7')) {
    count += decoding(digits, n - 2);
}

return count;
}

```

Code 15.6

```

public static int decoding(char digits[]) {
    // if the digits[] starts with 0 (for example, '0212')
    if (digits == null || digits[0] == '0') {
        return 0;
    }

    int n = digits.length;

```

```

// store results of sub-problems

int count[] = new int[n + 1];

count[0] = 1;

count[1] = 1;

for (int i = 2; i <= n; i++) {

    count[i] = 0;

    // If the last digit is not 0 then last digit must
    // add to the number of words

    if (digits[i - 1] > '0') {

        count[i] = count[i - 1];

    }

    // If the second last digit is smaller than 2 and
    // the last digit is smaller than 7, then last
    // two digits represent a valid character

    if (digits[i - 2] == '1' || (digits[i - 2] == '2'
        && digits[i - 1] < '7')) {

        count[i] += count[i - 2];

    }

}

return count[n];
}

```

Code 15.7

```

public static void find() {

    for (int i = 1000; i < 2499; i++) {

        int p = i;

        int q = i * 4;

        String m = String.valueOf(p);

        String n = new StringBuilder(String.valueOf(q))

            .reverse().toString();

        p = Integer.parseInt(m);
    }
}

```

```

q = Integer.parseInt(n);

if (p == q) {
    System.out.println("\n\nFound: " + p + " : " + (q * 4));
    break;
}
}
}

```

Code 15.8

```

public static boolean overlap(Point r1lt, Point r1rb,
    Point r2lt, Point r2rb) {
    // r1 is totally to the right of r2 or vice versa
    if (r1lt.x > r2rb.x || r2lt.x > r1rb.x) {
        return false;
    }
    // r1 is totally above r2 or vice versa
    if (r1rb.y > r2lt.y || r2rb.y > r1lt.y) {
        return false;
    }
    return true;
}

```

Code 15.9

```

public static boolean overlap(Point r1lt, Point r1rb,
    Point r2lt, Point r2rb) {
    return (r1lt.x <= r2rb.x && r1rb.x >= r2lt.x
        && r1lt.y >= r2rb.y && r1rb.y <= r2lt.y);
}

```

Code 15.10

```

public static String multiply(String a, String b) {
    int lenA = a.length();
    int lenB = b.length();

```

```
if (lenA == 0 || lenB == 0) {
    return "0";
}

// the result of multiplication is stored in reverse order
int c[] = new int[lenA + lenB];
// indexes to find positions in result
int idx1 = 0;
int idx2 = 0;
// loop 'a' right to left
for (int i = lenA - 1; i >= 0; i--) {
    int carry = 0;
    int n1 = a.charAt(i) - '0';
    // used to shift position to left after every
    // multiplication of a digit in 'b'
    idx2 = 0;
    // loop 'b' from right to left
    for (int j = lenB - 1; j >= 0; j--) {
        // current digit of second number
        int n2 = b.charAt(j) - '0';
        // multiply with current digit of first number
        int sum = n1 * n2 + c[idx1 + idx2] + carry;
        // carry of the next iteration
        carry = sum / 10;
        c[idx1 + idx2] = sum % 10;
        idx2++;
    }
    // store carry
    if (carry > 0) {
        c[idx1 + idx2] += carry;
    }
}
```

```

    // shift position to left after every
    // multiplication of a digit in 'a'
    idx1++;
}

// ignore '0's from the right
int i = c.length - 1;
while (i >= 0 && c[i] == 0) {
    i--;
}

// If all were '0's - means either both or
// one of 'a' or 'b' were '0'
if (i == -1) {
    return "0";
}

String result = "";
while (i >= 0) {
    result += (c[i--]);
}
return result;
}

```

Code 15.11

```

public static void findNextGreater(int arr[]) {
    int min = -1;
    int len = arr.length;
    int prevDigit = arr[arr.length - 1];
    int currentDigit;
    // Step 1: Start from the rightmost digit and find the
    // first digit that is smaller than the digit next to it.
    for (int i = len - 2; i >= 0; i--) {
        currentDigit = arr[i];

```

```
if (currentDigit < prevDigit) {  
    min = i;  
    break;  
}  
}  
  
// If 'min' is -1 then there is no such digit.  
// This means that the digits are in descending order.  
// There is no greater number with same set of digits  
// as the given one.  
if (min == -1) {  
    System.out.println("There is no greater number with "  
        + "same set of digits as the given one.");  
} else {  
    // Steps 2 and 3: Swap 'min' with 'len-1'  
    swap(arr, min, len - 1);  
    // Step 4: Sort in ascending order all the digits  
    // to the right side of the swapped 'len-1'  
    reverse(arr, min + 1, len - 1);  
    // print the result  
    System.out.print("The next greater number is: ");  
    for (int i : arr) {  
        System.out.print(i);  
    }  
}  
}  
  
private static void reverse(int[] arr, int start, int end) {  
    while (start < end) {  
        swap(arr, start, end);  
        start++;  
        end--;
```

```

    }
}

private static void swap(int[] arr, int i, int j) {
    int aux = arr[i];
    arr[i] = arr[j];
    arr[j] = aux;
}

```

Code 15.12

```

public static boolean isDivisible(int n) {
    int t = n;
    while (n > 0) {
        int k = n % 10;
        if (k != 0 && t % k != 0) {
            return false;
        }
        n /= 10;
    }
    return true;
}

```

Code 15.13

```

public static int breakit(int width, int height, int nTiles) {
    if (width <= 0 || height <= 0 || nTiles <= 0) {
        return -1;
    }
    // case 1
    if (width * height < nTiles) {
        return -1;
    }
    // case 4
    if (width * height == nTiles) {

```

```

        return 0;
    }

    // cases 5 and 6
    if ((nTiles % width == 0 && (nTiles / width) < height)
        || (nTiles % height == 0 && (nTiles / height) < width)) {
        return 1;
    }

    // case 7
    for (int i = 1; i <= Math.sqrt(nTiles); i++) {
        if (nTiles % i == 0) {
            int a = i;
            int b = nTiles / i;
            if ((a <= width && b <= height)
                || (a <= height && b <= width)) {
                return 2;
            }
        }
    }

    // cases 2 and 3
    return -1;
}

```

Code 15.14

```

public static float findAngle(int hour, int min) {
    float angle = (float) Math.abs(((30f * hour)
        + (0.5f * min)) - (6f * min));
    return angle > 180f ? (360f - angle) : angle;
}

```

Code 15.15

```

public static void triplet(int arr[]) {
    int len = arr.length;

```

```

// Step1
for (int i = 0; i < len; i++) {
    arr[i] = arr[i] * arr[i];
}

// Step 2
Arrays.sort(arr);

// Steps 3, 4, and 5
for (int i = len - 1; i >= 2; i--) {
    int b = 0;
    int c = i - 1;
    // Step 6
    while (b < c) {
        // Step 6c
        if (arr[b] + arr[c] == arr[i]) {
            System.out.println("Triplet: " + Math.sqrt(arr[b])
                + ", " + Math.sqrt(arr[c]) + ", "
                + Math.sqrt(arr[i]));
            b++;
            c--;
        }
        // Steps 6a and 6b
        if (arr[b] + arr[c] < arr[i]) {
            b++;
        } else {
            c--;
        }
    }
}

```

Code 15.16

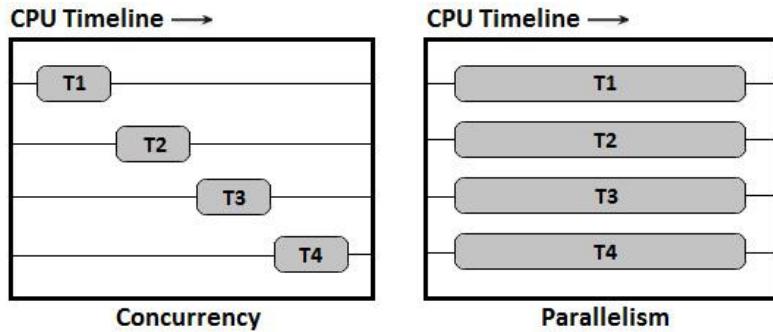
```

public static int time(int k, int floors[]) {
    int aux;
    for (int i = 0; i < floors.length - 1; i++) {
        for (int j = i + 1; j < floors.length; j++) {
            if (floors[i] < floors[j]) {
                aux = floors[i];
                floors[i] = floors[j];
                floors[j] = aux;
            }
        }
    }
    // iterate the groups and update
    // the time needed for each group
    int time = 0;
    for (int i = 0; i < floors.length; i += k) {
        time += (2 * floors[i]);
    }
    return time;
}

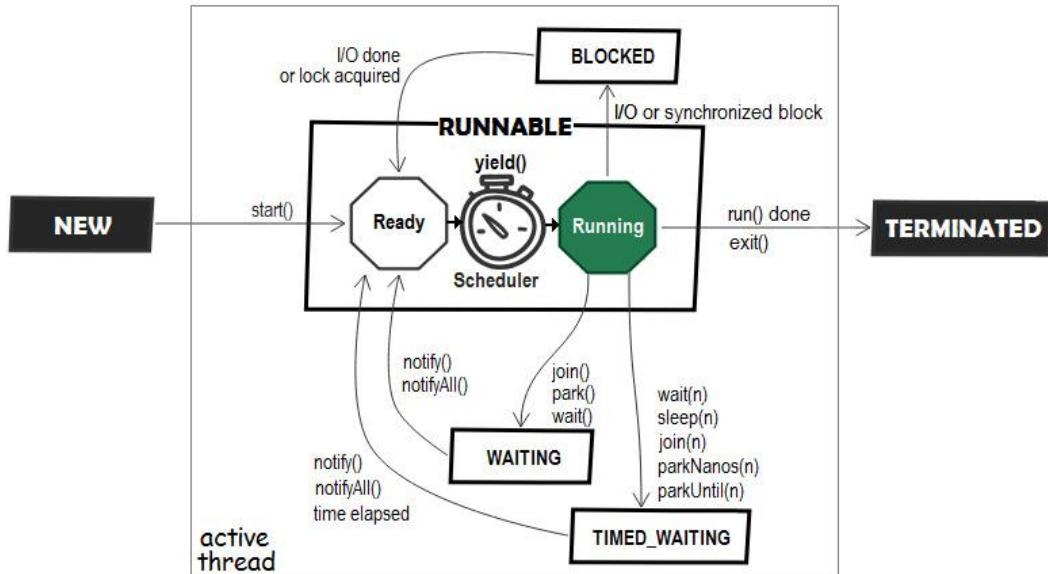
```

Chapter 16

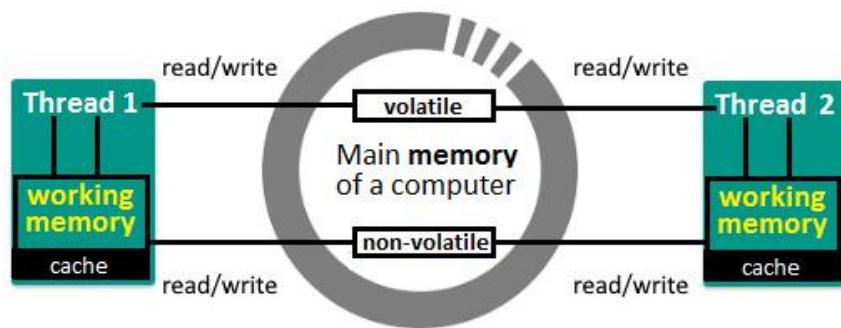
Images



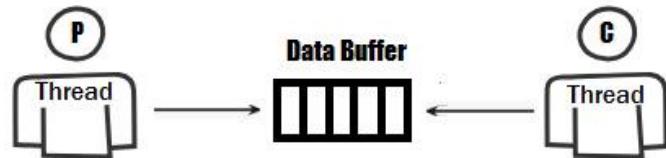
16.1 – Concurrency versus parallelism



16.2 – Java thread states



16.3 – Volatile flag read/write



16.4 – Producer-Consumer design pattern

Code

Code 16.1

```
void run()
```

Code 16.2

```
V call() throws Exception
```

Code 16.3

```
synchronized (queue) {  
    while (!queue.isEmpty()) {  
        logger.info("Queue is not empty ...");  
        queue.wait();  
    }  
}
```

Code 16.4

```
synchronized (queue) {  
    String product = "product-" + rnd.nextInt(1000);  
    // simulate the production time  
    Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS));  
    queue.add(product);  
    logger.info(() -> "Produced: " + product);  
    queue.notify();  
}
```

Code 16.5

```
synchronized (queue) {  
    while (queue.isEmpty()) {  
        logger.info("Queue is empty ...");  
        queue.wait();  
    }  
}
```

Code 16.6

```
synchronized (queue) {  
    String product = queue.remove(0);  
    if (product != null) {  
        // simulate consuming time
```

```

        Thread.sleep(rnd.nextInt(MAX_CONS_TIME_MS));

        logger.info(() -> "Consumed: " + product);

        queue.notify();

    }

}

```

Code 16.7

```

while (queue.hasWaitingConsumer()) {

    String product = "product-" + rnd.nextInt(1000);

    // simulate the production time

    Thread.sleep(rnd.nextInt(MAX_PROD_TIME_MS));

    queue.add(product);

    logger.info(() -> "Produced: " + product);

}

```

Code 16.8

```

// MAX_PROD_TIME_MS * 2, just give enough time to the producer

String product = queue.poll(
    MAX_PROD_TIME_MS * 2, TimeUnit.MILLISECONDS);

if (product != null) {

    // simulate consuming time

    Thread.sleep(rnd.nextInt(MAX_CONS_TIME_MS));

    logger.info(() -> "Consumed: " + product);

}

```

Chapter 17

Images



Figure 17.1 – Lambda parts

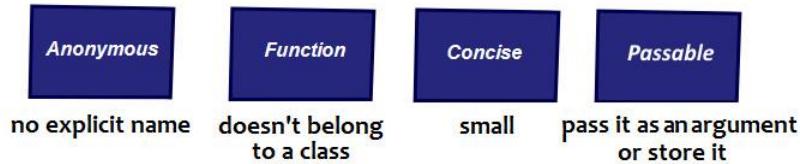


Figure 17.2 – Lambda characteristics

Code

Code 17.1

```
public class Calculator {  
    public int sum(int x, int y) {  
        return x + y;  
    }  
}
```

Code 17.2

```
Collections.sort(list, (String x, String y) -> {  
    return x.compareTo(y);  
});
```

Code 17.3

```
static long factorial(long n) {  
    long result = 1;  
    for (; n > 0; n--) {  
        result *= n;  
    }  
    return result;  
}
```

Code 17.4

```
static long factorial(long n) {
```

```
    return n == 1 ? 1 : n * factorial(n - 1);  
}
```

Code 17.5

```
static long factorialTail(long n) {  
    return factorial(1, n);  
}  
  
static long factorial(long acc, long v) {  
    return v == 1 ? acc : factorial(acc * v, v - 1);  
}
```

Code 17.6

```
static long factorial(long n) {  
    return LongStream.rangeClosed(1, n)  
        .reduce(1, (n1, n2) -> n1 * n2);  
}
```

Code 17.7

```
FilenameFilter filter = new FilenameFilter() {  
    @Override  
    public boolean accept(File folder, String fileName) {  
        return folder.canRead() && fileName.endsWith(".pdf");  
    }  
};
```

Code 17.8

```
@FunctionalInterface  
public interface Callable<V> {  
    V call() throws Exception;  
}
```

Code 17.9

```
List<String> strList = Arrays.asList("1", "2", "3");  
List<Integer> intList = strList.stream()  
    .map(Integer::parseInt)
```

```
.collect(Collectors.toList()));
```

Code 17.20

```
List<List<Object>> list = ...  
List<Object> flatList = list.stream()  
    .flatMap(List::stream)  
    .collect(Collectors.toList());
```

Code 17.21

```
List<List<String>> melonLists = Arrays.asList(  
    Arrays.asList("Gac", "Cantaloupe"),  
    Arrays.asList("Hemi", "Gac", "Apollo"),  
    Arrays.asList("Gac", "Hemi", "Cantaloupe"));
```

Code 17.22

```
melonLists.stream()  
    .map(Collection::stream) // Stream<Stream<String>>  
    .distinct();
```

Code 17.23

```
List<String> distinctNames = melonLists.stream()  
    .flatMap(Collection::stream) // Stream<String>  
    .distinct()  
    .collect(Collectors.toList());
```

Code 17.24

```
List<Integer> ints  
= Arrays.asList(1, 2, -4, 0, 2, 0, -1, 14, 0, -1);
```

Code 17.25

```
List<Integer> result = ints.stream()  
    .filter(i -> i != 0)  
    .collect(Collectors.toList());
```

Code 17.26

```
addresses.stream()  
    .peek(p -> System.out.println("\tstream(): " + p))
```

```
.filter(s -> s.startsWith("c"))
.sorted()
.peek(p -> System.out.println("\tsorted(): " + p))
.collect(Collectors.toList());
```

Code 17.29

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    // default and static methods omitted for brevity
}
```

Code 17.30

```
public static <T> Stream<T> toStream(T[] arr) {
    return Arrays.stream(arr);
}
```

Code 17.32

```
public static <T> Stream<T> toStream(T[] arr) {
    return Stream.of(arr);
}
```

Code 17.33

```
public static <T> Stream<T> toStream(T[] arr) {
    return Arrays.asList(arr).stream();
}
```

Code 17.34

```
public static IntStream toStream(int[] arr) {
    return Arrays.stream(arr);
}
```

Code 17.35

```
public static IntStream toStream(int[] arr) {
    return IntStream.of(arr);
}
```

Code 17.36

```
public interface Polygon {  
    public double area();  
    default double perimeter(double... segments) {  
        return Arrays.stream(segments)  
            .sum();  
    }  
}
```

Code 17.37

```
Spliterators.spliteratorUnknownSize(  
    your_Iterator, your_Properties);
```

Code 17.38

```
Optional<User> userOptional = Optional.empty();
```

Code 17.39

```
User user = new User();  
Optional<User> userOptional = Optional.of(user);
```

Chapter 18

Images



Figure 18.1 – Unit testing flow



Figure 18.2 – Functional testing

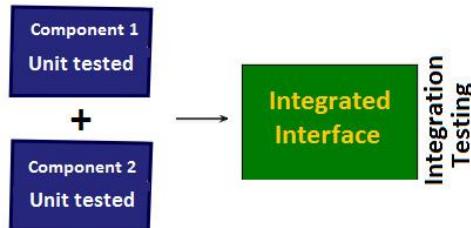


Figure 18.3 – Integration testing

Unit test	Integration test
Typically, useful to developers.	Useful to QA, DevOps, and Help Desk as well
Results depend only on Java code.	Results may depend on external systems.
Fairly easy to write.	May be quite complicated to set up
Units are tested in isolation.	One or more components are tested.
Can use mocking for dependencies.	Mocking should be avoided.
Only test the implementation of code.	Test the implementation of the components and the interconnection behavior between them
Uses JUnit/TestNG.	Uses real environments via tools such as Arquillian and DbUnit.
A failed test is a regression problem.	A failed test can be caused by changes in the environment
Such tests shouldn't take too long to run.	Such tests can take quite a long time (for example, 1 hour)

Figure 18.4 – Comparison between unit tests and integration tests

Code

Code 18.1

```

@Test
public void givenStreamWhenSumThenEquals6() {
    // Arrange
    Stream<Integer> theStream = Stream.of(1, 2, 3);
}

```

```

// Act

int sum = theStream.mapToInt(i -> i).sum();

// Assert

assertEquals(6, sum);

}

```

Code 18.2

```

@Test

public void givenStreamWhenGetThenException() {

    Stream<Integer> theStream = Stream.of();

    try {
        theStream.findAny().get();
        fail("Expected a NoSuchElementException to be thrown");
    } catch (NoSuchElementException ex) {
        assertThat(ex.getMessage(), is("No value present"));
    }
}

```

Code 18.3

```

@Test(expected = NoSuchElementException.class)

public void givenStreamWhenGetThenException() {

    Stream<Integer> theStream = Stream.of();

    theStream.findAny().get();
}

```

Code 18.4

```

@Rule

public ExpectedException thrown = ExpectedException.none();

@Test

public void givenStreamWhenGetThenException()

    throws NoSuchElementException {
    Stream<Integer> theStream = Stream.of();
    thrown.expect(NoSuchElementException.class);
}

```

```
        thrown.expectMessage("No value present");

        theStream.findAny().get();

    }
```

Code 18.6

```
@Test

public void givenStreamWhenGetThenException() {
    assertThrows(NoSuchElementException.class, () -> {
        Stream<Integer> theStream = Stream.of();
        theStream.findAny().get();
    });
}
```

Code 18.7

```
@Test

public void givenStreamWhenGetThenException() {
    Throwable ex = assertThrows(
        NoSuchElementException.class, () -> {
            Stream<Integer> theStream = Stream.of();
            theStream.findAny().get();
        });
    assertEquals(ex.getMessage(), "No value present");
}
```

Code 18.9

```
@RunWith(Suite.class)

@Suite.SuiteClasses({
    TestConnect.class,
    TestHeartbeat.class,
    TestDisconnect.class
})

public class TestSuite {
    // this class was intentionally left empty
}
```

```
}
```

Code 18.10

```
@RunWith(JUnitPlatform.class)
@SuiteDisplayName("TEST LOGIN AND CONNECTION")
@SelectPackages({
    "coding.challenge.connection.test",
    "coding.challenge.login.test"
})
public class TestLoginSuite {
    // this class was intentionally left empty
}
```

Code 18.11

```
@RunWith(JUnitPlatform.class)
@SuiteDisplayName("TEST CONNECTION")
@SelectClasses({
    TestConnect.class,
    TestHeartbeat.class,
    TestDisconnect.class
})
public class TestConnectionSuite {
    // this class was intentionally left empty
}
```

Code 18.12

```
@Test
public void givenFolderWhenGetAbsolutePathThenSuccess() {
    assumeThat(File.separatorChar, is('/'));
    assertThat(new File(".").getAbsolutePath(),
        is("C:/SBPBP/GitHub/Chapter18/junit4"));
}
```

Code 18.13

```
@Test
public void givenFolderWhenGetAbsolutePathThenSuccess() {
    assumingThat(File.separatorChar == '/',
    () -> {
        assertThat(new File(".").getAbsolutePath(),
        is("C:/SBPBP/GitHub/Chapter18/junit5"));
    });
    // run these assertions always, just like normal test
    assertTrue(true);
}
```

Code 18.14

```
1: @TestFactory
2: Stream<DynamicTest> dynamicTestsExample() {
3:
4:     List<Integer> items = Arrays.asList(1, 2, 3, 4, 5);
5:
6:     List<DynamicTest> dynamicTests = new ArrayList<>();
7:
8:     for (int item : items) {
9:         DynamicTest dynamicTest = dynamicTest(
10:             "pow(" + item + ", 2):", () -> {
11:                 assertEquals(item * item, Math.pow(item, 2));
12:             });
13:         dynamicTests.add(dynamicTest);
14:     }
15:
16:     return dynamicTests.stream();
17: }
```

Code 18.15

```
@RunWith(JUnitPlatform.class)
```

```
public class NestedTest {  
    private static final Logger log  
        = Logger.getLogger(NestedTest.class.getName());  
    @DisplayName("Test 1 - not nested")  
    @Test  
    void test1() {  
        log.info("Execute test1() ...");  
    }  
    @Nested  
    @DisplayName("Running tests nested in class A")  
    class A {  
        @BeforeEach  
        void beforeEach() {  
            System.out.println("Before each test  
method of the A class");  
        }  
        @AfterEach  
        void afterEach() {  
            System.out.println("After each test  
method of the A class");  
        }  
        @Test  
        @DisplayName("Test2 - nested in class A")  
        void test2() {  
            log.info("Execute test2() ...");  
        }  
    }  
}
```

Chapter 19

Images



Figure 19.1 – Latency versus bandwidth versus throughput

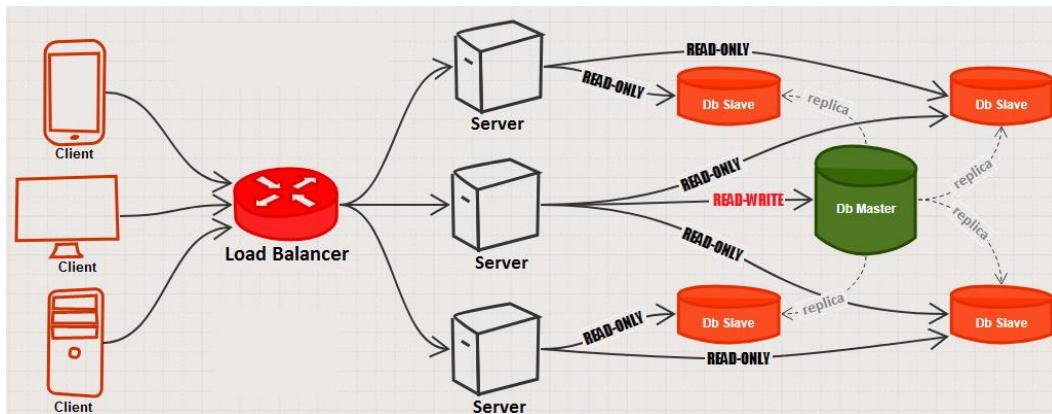


Figure 19.2 – Load balancer in a master-slave architecture

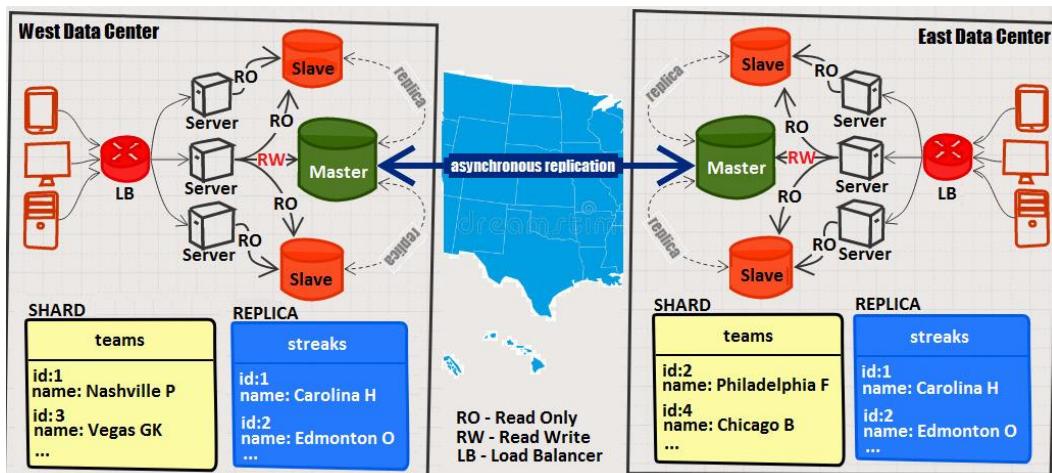


Figure 19.3 – Sharding

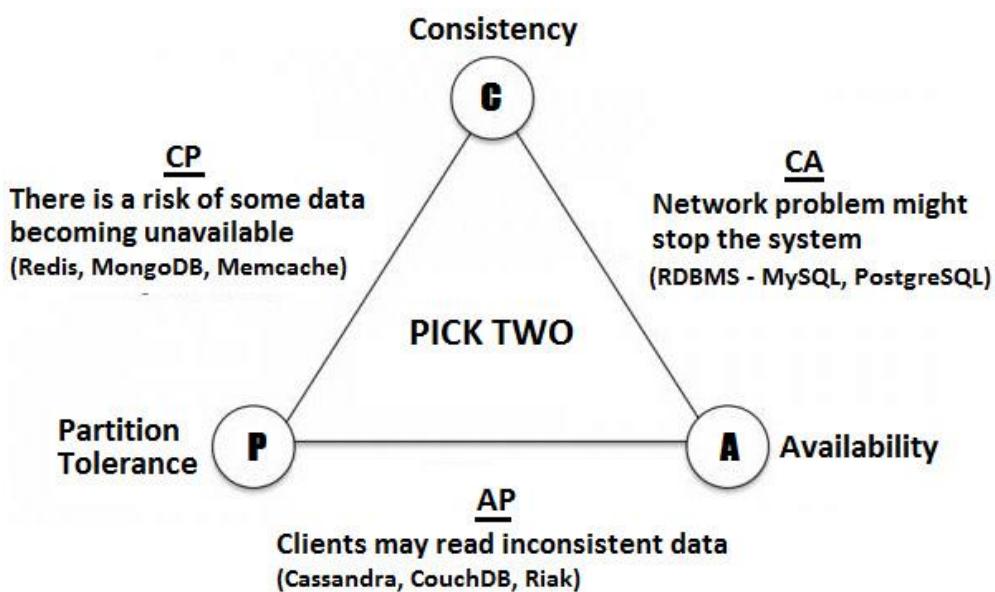


Figure 19.4 – The CAP theorem

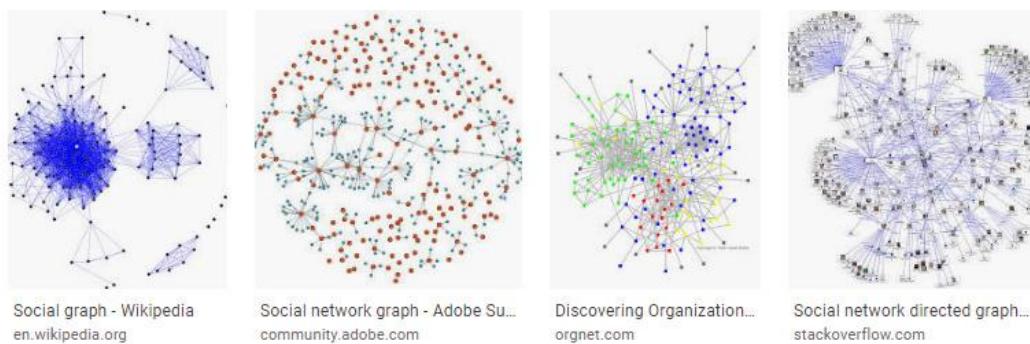


Figure 19.5 – Social network graph