1. Programmazione Strutturata

La programmazione strutturata è un paradigma che migliora la chiarezza e la qualità del codice attraverso l'uso di strutture di controllo ben definite e sottoprogrammi.

Principi fondamentali:

1. **Sequenza**: Esecuzione di istruzioni in ordine.
2. **Selezione**: Scelta tra alternative (if-then-else).
3. **Iterazione**: Ripetizione di blocchi di codice (cicli).
4. **Astrazione funzionale**: Uso di sottoprogrammi.

Esempio di codice strutturato (pseudocodice):

```
funzione calcola_media(numeri):
    somma = 0
    conteggio = 0
    per ogni numero in numeri:
        somma = somma + numero
        conteggio = conteggio + 1
    se conteggio > 0:
        return somma / conteggio
    altrimenti:
        return 0

principale:
    dati = [10, 15, 20, 25, 30]
    media = calcola_media(dati)
    stampa("La media è: " + media)
```

Vantaggi:

- Codice più leggibile e manutenibile
- Facilita il debug e il testing
- Promuove la riusabilità del codice

2. Quoziente Iterativo

Il quoziente iterativo è un metodo per calcolare la divisione intera usando sottrazioni ripetute.

Algoritmo:

1. Inizializza il quoziente e il resto al valore del dividendo
2. Finché il resto è maggiore o uguale al divisore:
   - Sottrai il divisore dal resto
   - Incrementa il quoziente di 1
3. Il risultato finale è il quoziente, con il resto rimanente

Implementazione in Python:

```
def quoziente_iterativo(dividendo, divisore):
    if divisore == 0:
        raise ValueError("Il divisore non può essere zero")

    quoziente = 0
    resto = abs(dividendo)
    divisore_abs = abs(divisore)

    while resto >= divisore_abs:
        resto -= divisore_abs
        quoziente += 1

    if (dividendo < 0) != (divisore < 0):
        quoziente = -quoziente

    return quoziente, resto

# Esempio di utilizzo
dividendo = 17
divisore = 5
quoziente, resto = quoziente_iterativo(dividendo, divisore)
print(f"{dividendo} diviso {divisore} dà quoziente {quoziente} e resto
{resto}")
```

Spiegazione:

- Gestiamo i numeri negativi prendendo il valore assoluto e aggiustando il segno alla fine
- Il ciclo `while` continua finché il resto è maggiore o uguale al divisore
- Ad ogni iterazione, sottraiamo il divisore dal resto e incrementiamo il quoziente
- Alla fine, il quoziente rappresenta quante volte il divisore "entra" nel dividendo

3. Tabelle di Traccia

Le tabelle di traccia sono strumenti per seguire l'esecuzione di un algoritmo, passo dopo passo.

Esempio: Algoritmo di Euclide per il MCD

```
def mcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Tabella di traccia per mcd(48, 18):

Passo	Istruzione	a	b
1	Inizio	48	18
2	a, b = b, a % b	18	12
3	a, b = b, a % b	12	6
4	a, b = b, a % b	6	0
5	Fine while	6	0

Risultato: MCD(48, 18) = 6

Utilizzo didattico:

- Aiuta gli studenti a visualizzare il flusso dell'algoritmo
- Utile per il debugging e la comprensione del codice
- Facilita l'identificazione di errori logici

4. Calcolo Iterativo della Radice Quadrata

Il metodo di Newton (o metodo delle tangenti) è un algoritmo iterativo efficiente per calcolare la radice quadrata.

Algoritmo:

1. Scegli un'approssimazione iniziale x0
2. Calcola la nuova approssimazione: x(n+1) = (xn + a/xn) / 2
3. Ripeti fino a raggiungere la precisione desiderata

Implementazione in Python:

```
def radice_quadrata(n, precisione=1e-10):
    if n < 0:
        raise ValueError("Non è possibile calcolare la radice quadrata di un
numero negativo")

    x = n  # Approssimazione iniziale
    while abs(x*x - n) > precisione:
        x = (x + n/x) / 2
    return x

# Esempio di utilizzo
numero = 16
radice = radice_quadrata(numero)
print(f"La radice quadrata di {numero} è approssimativamente {radice}")
```

Spiegazione:

- Iniziamo con un'approssimazione (qui usiamo il numero stesso)
- Ad ogni iterazione, calcoliamo una nuova approssimazione più precisa
- Il ciclo continua finché non raggiungiamo la precisione desiderata
- Questo metodo converge rapidamente per la maggior parte dei numeri

5. Vettori

I vettori (o array) sono strutture dati che contengono una sequenza di elementi dello stesso tipo.

Caratteristiche:

- Elementi memorizzati in locazioni contigue di memoria
- Accesso diretto agli elementi tramite indice
- Dimensione fissa in molti linguaggi (es. array in C), dinamica in altri (es. liste in Python)

Operazioni comuni:

1. **Inserimento**

```
def inserisci(vettore, elemento, posizione):
    return vettore[:posizione] + [elemento] + vettore[posizione:]

# Esempio
v = [1, 2, 3, 4, 5]
v = inserisci(v, 10, 2)
print(v)  # Output: [1, 2, 10, 3, 4, 5]
```

2. **Ricerca**

```
def ricerca(vettore, elemento):
    for i, e in enumerate(vettore):
        if e == elemento:
            return i
    return -1  # elemento non trovato

# Esempio
v = [1, 2, 3, 4, 5]
pos = ricerca(v, 3)
print(pos)  # Output: 2
```

3. **Eliminazione**

```
def elimina(vettore, posizione):
    return vettore[:posizione] + vettore[posizione+1:]

# Esempio
v = [1, 2, 3, 4, 5]
v = elimina(v, 2)
print(v)  # Output: [1, 2, 4, 5]
```

Vantaggi e svantaggi:

- **Pro**: Accesso rapido agli elementi, efficiente in memoria
- **Contro**: Dimensione fissa in alcuni linguaggi, inserimenti/eliminazioni possono essere costosi

6. Passaggio Parametri per Valore e per Riferimento

Il passaggio dei parametri determina come i dati vengono trasmessi alle funzioni.

Passaggio per Valore:

- Una copia del valore viene passata alla funzione
- Modifiche al parametro non influenzano la variabile originale

Passaggio per Riferimento:

- L'indirizzo di memoria della variabile viene passato
- Modifiche al parametro influenzano la variabile originale

Esempio in C++:

```
#include <iostream>
using namespace std;

void per_valore(int x) {
    x = x + 1;  // Non modifica la variabile originale
}

void per_riferimento(int &x) {
    x = x + 1;  // Modifica la variabile originale
}

int main() {
    int a = 5;

    per_valore(a);
    cout << "Dopo per_valore: " << a << endl;  // Output: 5
```

```
    per_riferimento(a);
    cout << "Dopo per_riferimento: " << a << endl;  // Output: 6

    return 0;
}
```

Implicazioni:

- Passaggio per valore: più sicuro ma può essere meno efficiente per grandi strutture dati
- Passaggio per riferimento: più efficiente ma richiede attenzione per evitare modifiche indesiderate

7. Ricerca Dicotomica

La ricerca dicotomica (o ricerca binaria) è un algoritmo efficiente per trovare un elemento in un array ordinato.

Algoritmo:

1. Confronta l'elemento centrale con il valore cercato
2. Se uguale, l'elemento è trovato
3. Se minore, cerca nella metà inferiore
4. Se maggiore, cerca nella metà superiore
5. Ripeti finché l'elemento è trovato o l'intervallo di ricerca è vuoto

Implementazione in Python:

```
def ricerca_dicotomica(array, valore):
    sinistra, destra = 0, len(array) - 1

    while sinistra <= destra:
        medio = (sinistra + destra) // 2
        if array[medio] == valore:
            return medio
        elif array[medio] < valore:
            sinistra = medio + 1
        else:
            destra = medio - 1
```

```
    return -1  # elemento non trovato

# Esempio di utilizzo
array_ordinato = [1, 3, 5, 7, 9, 11, 13, 15, 17]
valore_da_cercare = 7
risultato = ricerca_dicotomica(array_ordinato, valore_da_cercare)
if risultato != -1:
    print(f"Elemento {valore_da_cercare} trovato all'indice {risultato}")
else:
    print(f"Elemento {valore_da_cercare} non trovato")
```

Complessità:

- O(log n) nel caso peggiore e medio
- O(1) nel caso migliore (elemento trovato al centro)

Vantaggi:

- Molto efficiente per grandi dataset ordinati
- Riduce drasticamente il numero di confronti necessari rispetto alla ricerca lineare

Limitazioni:

- Richiede che l'array sia ordinato
- Non efficiente per piccoli dataset o per array che cambiano frequentemente

Java

Appunti Didattici di Programmazione Strutturata e Algoritmi in Java

1. Programmazione Strutturata

La programmazione strutturata in Java segue gli stessi principi fondamentali, utilizzando le strutture di controllo del linguaggio.

Esempio di codice strutturato in Java:

```
import java.util.List;

public class ProgrammazioneStrutturata {
    public static double calcolaMedia(List<Integer> numeri) {
        int somma = 0;
        int conteggio = 0;
        for (int numero : numeri) {
            somma += numero;
            conteggio++;
        }
        if (conteggio > 0) {
            return (double) somma / conteggio;
        } else {
            return 0;
        }
    }

    public static void main(String[] args) {
        List<Integer> dati = List.of(10, 15, 20, 25, 30);
        double media = calcolaMedia(dati);
        System.out.println("La media è: " + media);
    }
}
```

2. Quoziente Iterativo

Implementazione del quoziente iterativo in Java:

```
public class QuozienteIterativo {
    public static class RisultatoDivisione {
        public int quoziente;
        public int resto;

        public RisultatoDivisione(int quoziente, int resto) {
            this.quoziente = quoziente;
            this.resto = resto;
        }
    }

    public static RisultatoDivisione quozienteIterativo(int dividendo, int
divisore) {
```

```
        if (divisore == 0) {
            throw new ArithmeticException("Il divisore non può essere zero");
        }

        int quoziente = 0;
        int resto = Math.abs(dividendo);
        int divisoreAbs = Math.abs(divisore);

        while (resto >= divisoreAbs) {
            resto -= divisoreAbs;
            quoziente++;
        }

        if ((dividendo < 0) != (divisore < 0)) {
            quoziente = -quoziente;
        }

        return new RisultatoDivisione(quoziente, resto);
    }

    public static void main(String[] args) {
        int dividendo = 17;
        int divisore = 5;
        RisultatoDivisione risultato = quozienteIterativo(dividendo,
divisore);
        System.out.printf("%d diviso %d dà quoziente %d e resto %d%n",
                          dividendo, divisore, risultato.quoziente,
risultato.resto);
    }
}
```

3. Tabelle di Traccia

In Java, possiamo implementare l'algoritmo di Euclide per il MCD e creare una tabella di traccia:

```
public class AlgoritmoEuclide {
    public static int mcd(int a, int b) {
        System.out.println("Passo | a | b");
        System.out.println("------|---|---");
        int passo = 1;
        while (b != 0) {
            System.out.printf("%5d | %d | %d%n", passo, a, b);
            int temp = b;
```

```
            b = a % b;
            a = temp;
            passo++;
        }
        System.out.printf("%5d | %d | %d%n", passo, a, b);
        return a;
    }

    public static void main(String[] args) {
        int risultato = mcd(48, 18);
        System.out.println("MCD(48, 18) = " + risultato);
    }
}
```

4. Calcolo Iterativo della Radice Quadrata

Implementazione del metodo di Newton per il calcolo della radice quadrata in Java:

```
public class RadiceQuadrata {
    public static double radiceQuadrata(double n, double precisione) {
        if (n < 0) {
            throw new IllegalArgumentException("Non è possibile calcolare la
radice quadrata di un numero negativo");
        }

        double x = n; // Approssimazione iniziale
        while (Math.abs(x * x - n) > precisione) {
            x = (x + n / x) / 2;
        }
        return x;
    }

    public static void main(String[] args) {
        double numero = 16;
        double radice = radiceQuadrata(numero, 1e-10);
        System.out.printf("La radice quadrata di %.0f è approssimativamente
%.10f%n", numero, radice);
    }
}
```

5. Vettori

In Java, utilizzeremo gli ArrayList per implementare le operazioni sui vettori:

```
import java.util.ArrayList;

public class OperazioniVettori {
    public static <T> ArrayList<T> inserisci(ArrayList<T> vettore, T elemento,
int posizione) {
        vettore.add(posizione, elemento);
        return vettore;
    }

    public static <T> int ricerca(ArrayList<T> vettore, T elemento) {
        return vettore.indexOf(elemento);
    }

    public static <T> ArrayList<T> elimina(ArrayList<T> vettore, int
posizione) {
        vettore.remove(posizione);
        return vettore;
    }

    public static void main(String[] args) {
        ArrayList<Integer> v = new ArrayList<>(List.of(1, 2, 3, 4, 5));

        v = inserisci(v, 10, 2);
        System.out.println("Dopo inserimento: " + v);

        int pos = ricerca(v, 3);
        System.out.println("Posizione di 3: " + pos);

        v = elimina(v, 2);
        System.out.println("Dopo eliminazione: " + v);
    }
}
```

6. Passaggio Parametri per Valore e per Riferimento

In Java, i tipi primitivi sono passati per valore, mentre gli oggetti sono passati per riferimento. Ecco un esempio che illustra entrambi i casi:

```
public class PassaggioParametri {
    static class MioIntero {
        int valore;
```

```
        MioIntero(int valore) { this.valore = valore; }
    }

    public static void perValore(int x) {
        x = x + 1;  // Non modifica la variabile originale
    }

    public static void perRiferimento(MioIntero x) {
        x.valore = x.valore + 1;  // Modifica l'oggetto originale
    }

    public static void main(String[] args) {
        int a = 5;
        MioIntero b = new MioIntero(5);

        perValore(a);
        System.out.println("Dopo perValore: " + a);  // Output: 5

        perRiferimento(b);
        System.out.println("Dopo perRiferimento: " + b.valore);  // Output: 6
    }
}
```

7. Ricerca Dicotomica

Implementazione della ricerca dicotomica in Java:

```
public class RicercaDicotomica {
    public static int ricercaDicotomica(int[] array, int valore) {
        int sinistra = 0;
        int destra = array.length - 1;

        while (sinistra <= destra) {
            int medio = (sinistra + destra) / 2;
            if (array[medio] == valore) {
                return medio;
            } else if (array[medio] < valore) {
                sinistra = medio + 1;
            } else {
                destra = medio - 1;
            }
        }

        return -1;  // elemento non trovato
```

```
    }

    public static void main(String[] args) {
        int[] arrayOrdinato = {1, 3, 5, 7, 9, 11, 13, 15, 17};
        int valoreDaCercare = 7;
        int risultato = ricercaDicotomica(arrayOrdinato, valoreDaCercare);
        if (risultato != -1) {
            System.out.printf("Elemento %d trovato all'indice %d%n",
valoreDaCercare, risultato);
        } else {
            System.out.printf("Elemento %d non trovato%n", valoreDaCercare);
        }
    }
}
```

Questi appunti forniscono implementazioni Java per tutti i concetti precedentemente discussi. Gli studenti possono utilizzare questi esempi per comprendere come i principi di programmazione strutturata e gli algoritmi si applicano specificamente in Java. Ogni sezione include un esempio pratico che può essere eseguito e sperimentato.