01. Čísla a číselné algoritmy

Obsah

Reprezentace celých a reálných čísel v paměti

Celá čísla

- `byte`: 8 bitů, rozsah -128 až 127
- `short`: 16 bitů, rozsah -32,768 až 32,767
- `int`: 32 bitů, rozsah -2^31 až 2^31-1
- `long`: 64 bitů, rozsah -2^63 až 2^63-1

Celá čísla jsou v paměti uložena ve dvojkovém doplňkovém kódu pro reprezentaci záporných čísel.

Reálná čísla

V počítači nejsou ukládána přesná reálná čísla, ale jen jejich racionální aproximace: - `float`: 32 bitů, standard IEEE 754 - `double`: 64 bitů, standard IEEE 754

Formát s plovoucí desetinnou čárkou: - Znaménkový bit (1 bit) - Exponent (8 bitů pro float, 11 bitů pro double) - Mantisa (23 bitů pro float, 52 bitů pro double)

Číslo = (-1)^znaménkový_bit × mantisa × 2^exponent

Čísla s velkým počtem číslic

Pro práci s čísly o velkém počtu číslic: - `BigInteger`: pro celá čísla neomezené velikosti - `BigDecimal`: pro desetinná čísla s libovolnou přesností

Algoritmy převodu mezi soustavami

Převod z desítkové do jiné soustavy

Algoritmus:

```
// Převod z desítkové soustavy do soustavy o základu b
String prevodDoDruheSoustavy(int cislo, int zaklad) {
    String vysledek = "";
    while (cislo > 0) {
        int zbytek = cislo % zaklad;
        // Pro základy > 10 konvertujeme číslice nad 9 na písmena
        char znak = (zbytek < 10) ? (char)('0' + zbytek) : (char)('A' + (zbytek - 10));
        vysledek = znak + vysledek;
        cislo = cislo / zaklad;
    }
    return vysledek;
}
```

Hornerovo schéma

Slouží k efektivnímu vyhodnocení polynomu nebo převodu mezi soustavami.

Pro vyhodnocení polynomu: Pro polynom $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$:

```
// Hornerovo schéma pro vyhodnocení polynomu
double hornerPolynom(double[] koeficienty, double x) {
    // koeficienty[i] je koeficient u x^i
    double vysledek = koeficienty[koeficienty.length - 1];

    for (int i = koeficienty.length - 2; i >= 0; i--) {
        vysledek = vysledek * x + koeficienty[i];
    }

    return vysledek;
}
```

Složitost: O(n), kde n je stupeň polynomu.

Pro převod mezi soustavami: Při převodu z jiné soustavy do desítkové:

```
// Převod z libovolné soustavy do desítkové pomocí Hornerova schématu
int prevodDoDesitkoveSoustavy(String cislo, int zaklad) {
    int vysledek = 0;

    for (int i = 0; i < cislo.length(); i++) {
        char znak = cislo.charAt(i);
        int hodnota;
        if (Character.isDigit(znak)) {
            hodnota = znak - '0';
        } else {
            hodnota = Character.toUpperCase(znak) - 'A' + 10;
        }

        vysledek = vysledek * zaklad + hodnota;
    }

    return vysledek;
}
```

Testy prvočíselnosti

Naivní test

```
boolean jePrvocislo(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    // Testujeme dělitelnost pouze lichými čísly většími než 3
    for (int i = 5; i * i <= n; i += 2) {
        if (n % i == 0) return false;
    }

    return true;
}
```

Složitost: O(√n)

Eratosthenovo síto

Efektivní algoritmus pro nalezení všech prvočísel do určitého limitu.

```
// Implementace Eratosthenova síta
boolean[] eratosthenovoSito(int n) {
    boolean[] jePrvocislo = new boolean[n + 1];

    // Inicializace - všechna čísla jsou potenciálně prvočísla
    for (int i = 2; i <= n; i++) {
        jePrvocislo[i] = true;
    }

    // Hlavní průchod sítem
    for (int p = 2; p * p <= n; p++) {
        // Pokud p je prvočíslo, škrtáme jeho násobky
        if (jePrvocislo[p]) {
            // Začneme od p*p, protože menší násobky už byly škrtnuty
            for (int i = p * p; i <= n; i += p) {
                jePrvocislo[i] = false;
            }
        }
    }

    return jePrvocislo;
}
```

Složitost: O(n log log n)

Výhody: - Velmi efektivní pro hledání prvočísel do velkého limitu - Jednoduchá implementace

Nevýhody: - Paměťově náročné pro velká čísla

Eukleidův algoritmus

Algoritmus pro nalezení největšího společného dělitele (NSD) dvou čísel.

Rekurzivní varianta

```
// Rekurzivní Eukleidův algoritmus
int nsd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return nsd(b, a % b);
}
```

Iterativní varianta

```
// Iterativní Eukleidův algoritmus
int nsd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

Složitost: O(log(min(a, b)))

Rozklad na prvočísla

Algoritmus pro rozklad čísla na prvočísla.

```
// Rozklad čísla na prvočísla
void rozkladNaPrvocisla(int n) {
    // Zpracování násobků 2
    while (n % 2 == 0) {
        System.out.print(2 + " ");
        n /= 2;
    }

    // Testování lichých čísel od 3
    for (int i = 3; i * i <= n; i += 2) {
        while (n % i == 0) {
            System.out.print(i + " ");
            n /= i;
        }
    }

    // Pokud zbylo prvočíslo větší než 2
    if (n > 2) {
        System.out.print(n);
    }
}
```

Složitost: O(√n)

Bitové operace

Bitové operace pracují s čísly na úrovni jednotlivých bitů:

- & (AND): pokud jsou oba bity 1, výsledek je 1, jinak 0
- | (OR): pokud je alespoň jeden bit 1, výsledek je 1, jinak 0
- ^ (XOR): pokud jsou bity různé, výsledek je 1, jinak 0
- ~ (NOT): invertuje všechny bity
- << (levý posun): posune bity doleva
- >> (pravý posun): posune bity doprava (zachovává znaménko)
- >>> (pravý posun bez znaménka): posune bity doprava a doplní nuly

Příklady použití

1. **Test sudosti/lichosti čísla**:

```
boolean jeSude(int n) {
    return (n & 1) == 0;
}
```

2. **Výpočet 2^n**:

```
int mocnina2(int n) {
    return 1 << n;
}
```

3. **Nastavení bitu na určité pozici**:

```
int nastavBit(int cislo, int pozice) {
    return cislo | (1 << pozice);
}
```

4. **Test, zda je bit na určité pozici nastaven**:

```
boolean jeBitNastaven(int cislo, int pozice) {
    return (cislo & (1 << pozice)) != 0;
}
```

5. **Přepnutí hodnoty bitu na určité pozici**:

```
int prepniBit(int cislo, int pozice) {
    return cislo ^ (1 << pozice);
}
```

Využití bitových operací

- Rychlá matematika: násobení/dělení mocninami 2 pomocí posunů
- Optimalizace paměti: ukládání příznaků do jedné proměnné
- Datové struktury: implementace Eratosthenova síta pomocí bitového pole
- Kryptografie: šifrování dat
- Počítačová grafika: míchání barev, masek apod.
- Hašování a kontrolní součty

Rozdíl mezi dělením a celočíselným dělením

Dělení (/ s reálnými čísly)

- Pracuje s reálnými čísly (float, double)
- Vrací výsledek jako reálné číslo s desetinnou částí
- Příklad: 5.0 / 2.0 = 2.5

Celočíselné dělení (/ s celými čísly)

- Pracuje s celými čísly (int, long)
- Vrací pouze celočíselnou část (zaokrouhluje dolů k nule)
- Příklad: 5 / 2 = 2
- Zbytek po dělení se získá operátorem modulo %
- Příklad: 5 % 2 = 1

```
// Ukázka rozdílu
int celočíselnéDělení = 5 / 2;        // Výsledek: 2
double reálnéDělení = 5.0 / 2.0;      // Výsledek: 2.5
int zbytek = 5 % 2;                   // Výsledek: 1
```