SADRŽAJ

Niz u C++

Niz (eng. *array*) je struktura podataka koja omogućava da se više elemenata istog tipa (npr. int, float, char) skladišti na jednoj lokaciji u memoriji. Niz se koristi za organizaciju podataka u sekvencu.

Deklaracija i inicijalizacija niza:

```
int niz[5]; // Deklaracija niza od 5 elemenata tipa int
```

- Indeksi u nizu u C++ počinju od **0**.
- Vrednosti elemenata u nizu možete postaviti prilikom deklaracije:

```
int niz[5] = {10, 20, 30, 40, 50}; // Inicijalizacija
```

Pristup elementima niza:

Da pristupite nekom elementu niza, koristite indeks:

```
cout << niz[0]; // Ispisuje prvi element (10)
niz[2] = 100;   // Menja treći element na 100
```

Iteracija kroz niz:

Možete koristiti petlje za rad sa svim elementima:

```
for (int i = 0; i < 5; i++) {
    cout << niz[i] << " ";
}
```

Matrica u C++

Matrica je zapravo **dvodimenzionalni niz**. Možete je zamisliti kao tabelu sa redovima i kolonama.

Deklaracija matrice:

```
int matrica[3][3]; // Deklaracija matrice od 3 reda i 3 kolone
```

Inicijalizacija matrice:

Matrica se može inicijalizovati na sledeći način:

```
int matrica[3][3] = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
```

Pristup elementima matrice:

Elementima matrice pristupate pomoću dva indeksa: jedan za red, drugi za kolonu.

```
cout << matrica[1][2]; // Ispisuje element iz drugog reda i treće kolone (6)
```

Iteracija kroz matricu:

Kada radite sa matricom, koristite **ugnežđene petlje**:

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
      cout << matrica[i][j] << " ";
   }
   cout << endl; // Prelazak u novi red
}
```

String kao niz karaktera (char[])

String se može definisati kao niz karaktera koji se završava posebnim karakterom '\0' (null terminator). Ovo je tradicionalan način predstavljanja stringova u jezicima poput C i starijih verzija C++.

Deklaracija i inicijalizacija:

```
char str1[6] = "Hello"; // 5 karaktera + '\0'
char str2[] = "World";  // Kompajler automatski određuje veličinu
```

- String str1 zauzima 6 bajtova: 'H', 'e', 'l', 'l', 'o', '\0'.
- Ako sami definišete niz, morate osigurati mesto za '\0'.

Pristup karakterima:

Koristite indeks da biste pristupili karakterima:

```
cout << str1[0]; // Ispisuje 'H'
str1[1] = 'a';   // Menja drugi karakter u 'a', pa string postaje "Hallo"
```

Korišćenje biblioteke <cstring>:

Biblioteka <cstring> sadrži funkcije za rad sa stringovima:

```
#include <cstring>

char str1[] = "Hello";
char str2[] = "World";
strcat(str1, str2); // Konkatenacija: "HelloWorld"
cout << strlen(str1); // Dužina stringa: 10
```

String kao std::string

Klasa std::string iz biblioteke <string> pruža jednostavniji i fleksibilniji način rada sa stringovima.

Deklaracija i inicijalizacija:

```
#include <string>

std::string str1 = "Hello";
std::string str2("World");
```

Operacije sa std::string:

Konkatenacija:

```
std::string str3 = str1 + " " + str2; // "Hello World"
```

Pristup karakterima:

```
cout << str1[0]; // 'H'
str1[1] = 'a';   // Menja u "Hallo"
```

Dužina stringa:

```
cout << str1.length(); // 5
```

Dodavanje teksta:

```
str1 += " World!"; // "Hello World!"
```

Poređenje stringova:

```
if (str1 == "Hello World!") {
    cout << "Stringovi su jednaki!";
}
```

Podstring:

```
std::string deo = str1.substr(0, 5); // "Hello"
```

Poređenje char[] i std::string

Osobina	char[]	std::string
Fleksibilnost	Ograničena (mora se ručno pratiti veličina)	Veoma fleksibilan
Lakoća upotrebe	Teže (zahteva ručno upravljanje nizom)	Jednostavnije i intuitivnije
Funkcionalnosti	Osnovne funkcije u <cstring>	Bogat skup metoda i operatora
Bezbednost	Veća šansa za greške (npr. prepisivanje memorije)	Bezbedniji zbog upravljanja memorijom

---

Primer korišćenja:

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

int main() {
    string ime = "Marko";
    string prezime = "Petrović";

    string punoIme = ime + " " + prezime; // Konkatenacija
    cout << "Puno ime: " << punoIme << endl;

    // Provera dužine
    cout << "Dužina imena: " << ime.length() << endl;

    // Podstring
    cout << "Prva tri slova: " << punoIme.substr(0, 3) << endl;

    return 0;
}
```

Stek (Stack)

Stek je struktura podataka sa principom **LIFO** (*Last In, First Out*), što znači da poslednji element koji je ubačen u stek prvi biva uklonjen.

Operacije sa stekom:

- **Push**: Dodavanje elementa na vrh steka.
- **Pop**: Uklanjanje elementa sa vrha steka.
- **Top/Peek**: Prikazivanje elementa na vrhu steka.

Stek u C++ sa std::stack:

U C++ stek se može implementirati pomoću klase std::stack iz biblioteke <stack>.

Primer:

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

int main() {
    stack<int> stek;

    // Dodavanje elemenata (Push)
    stek.push(10);
    stek.push(20);
    stek.push(30);

    // Prikazivanje vrha steka
    cout << "Vrh steka: " << stek.top() << endl; // 30

    // Uklanjanje elemenata (Pop)
    stek.pop();
    cout << "Novi vrh steka: " << stek.top() << endl; // 20

    // Provera da li je stek prazan
    if (stek.empty()) {
        cout << "Stek je prazan." << endl;
    } else {
        cout << "Stek nije prazan." << endl;
    }

    // Broj elemenata u steku
    cout << "Veličina steka: " << stek.size() << endl;

    return 0;
}
```

Red (Queue)

Red je struktura podataka sa principom **FIFO** (*First In, First Out*), što znači da prvi element koji je ubačen u red prvi biva uklonjen.

Operacije sa redom:

- **Enqueue**: Dodavanje elementa na kraj reda.
- **Dequeue**: Uklanjanje elementa sa početka reda.
- **Front**: Prikazivanje elementa na početku reda.
- **Back**: Prikazivanje elementa na kraju reda.

Red u C++ sa std::queue:

U C++ red se implementira pomoću klase std::queue iz biblioteke <queue>.

Primer:

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

int main() {
    queue<int> red;

    // Dodavanje elemenata (Enqueue)
    red.push(10);
    red.push(20);
    red.push(30);

    // Prikazivanje početka i kraja reda
    cout << "Prvi element reda: " << red.front() << endl; // 10
    cout << "Poslednji element reda: " << red.back() << endl; // 30

    // Uklanjanje elemenata (Dequeue)
    red.pop();
    cout << "Novi prvi element: " << red.front() << endl; // 20

    // Provera da li je red prazan
    if (red.empty()) {
        cout << "Red je prazan." << endl;
    } else {
        cout << "Red nije prazan." << endl;
    }

    // Broj elemenata u redu
    cout << "Veličina reda: " << red.size() << endl;

    return 0;
}
```

Razlike između steka i reda

Osobina	Stek	Red
Princip	LIFO (Last In, First Out)	FIFO (First In, First Out)
Operacija dodavanja	push()	push()
Operacija uklanjanja	pop()	pop()
Pristup elementima	top()	front() za početak, back() za kraj

Kada koristiti stek i red?

- **Stek** se koristi kada je potrebno da obavljate operacije u obrnutom redosledu, npr. u parsiranju izraza, implementaciji funkcijskog poziva ili pretrazi dubine (*DFS*).
- **Red** se koristi kada su elementi organizovani po prioritetu dolaska, npr. u simulacijama, redovima čekanja ili pretrazi širine (*BFS*).

Primena formula radi izbegavanja iteracija

Zbir prvih n prirodnih brojeva

Zbir prvih nn prirodnih brojeva može se izračunati iterativno koristeći petlju, ali postoji formula koja daje direktan rezultat:

$$S = \frac{n \cdot (n+1)}{2}$$

Iterativni pristup:

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

int main() {
    int n;
    cout << "Unesite broj n: ";
    cin >> n;

    int zbir = 0;
    for (int i = 1; i <= n; i++) {
        zbir += i;
    }

    cout << "Zbir prvih " << n << " prirodnih brojeva je: " << zbir << endl;
    return 0;
}
```

Ovaj pristup koristi petlju i zahteva O(n) iteracija.

Primena formule:

Umesto iteracija, možemo koristiti direktno formulu:

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

int main() {
    int n;
    cout << "Unesite broj n: ";
    cin >> n;

    int zbir = n * (n + 1) / 2; // Primena formule
    cout << "Zbir prvih " << n << " prirodnih brojeva je: " << zbir << endl;

    return 0;
}
```

Prednosti formule:

- Brža izvedba (konstantno vreme O(1)).
- Jednostavniji i sažetiji kod.

Zbir kvadrata prvih n prirodnih brojeva

Za zbir kvadrata prvih nn prirodnih brojeva:

$$S = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Primena formule:

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

int main() {
    int n;
    cout << "Unesite broj n: ";
    cin >> n;

    int zbirKvadrata = n * (n + 1) * (2 * n + 1) / 6;
    cout << "Zbir kvadrata prvih " << n << " prirodnih brojeva je: " << zbirKvadrata << endl;

    return 0;
}
```

Zbir prvih n parnih brojeva

Prvih nn parnih brojeva su: 2,4,6,…,2n2, 4, 6,…, 2n. Njihov zbir je:

$$S = n \cdot (n+1)$$

Primena formule:

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

int main() {
    int n;
    cout << "Unesite broj n: ";
    cin >> n;

    int zbirParnih = n * (n + 1);
    cout << "Zbir prvih " << n << " parnih brojeva je: " << zbirParnih << endl;

    return 0;
}
```

Zbir prvih n neparnih brojeva

Prvih nn neparnih brojeva su: 1,3,5,…,2n−11, 3, 5,…, 2n - 1. Njihov zbir je:

$$S = n^2$$

Primena formule:

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

int main() {
    int n;
    cout << "Unesite broj n: ";
    cin >> n;

    int zbirNeparnih = n * n;
    cout << "Zbir prvih " << n << " neparnih brojeva je: " << zbirNeparnih << endl;

    return 0;
}
```

Prednosti korišćenja formula:

1. **Efikasnost:** Ne koristimo petlje, pa je vreme izvršavanja konstantno (O(1)O(1)).
2. **Jednostavnost:** Kod je kraći i lakši za razumevanje.
3. **Pouzdanost:** Formula je matematički tačna i izbegava greške koje mogu nastati tokom iteracija (npr. prekorak broja).

Prefiksni i sufiksni zbirovi

Prefiksni i sufiksni zbirovi su korisni koncepti koji omogućavaju efikasno izračunavanje zbira elemenata niza u određenom opsegu. Ovo se postiže unapred računajući zbirove za sve prefikse ili sufikse niza, što omogućava odgovore na upite u konstantnom vremenu.

Prefiksni zbir

Prefiksni zbir za niz je niz u kojem je svaki element zbir svih prethodnih elemenata originalnog niza do tog indeksa, uključujući i njega samog.

Definicija:

Ako imamo niz a[1],a[2],…,a[n]a[1], a[2],…, a[n], prefiksni zbir P[i] definišemo kao:

$$P[i] = a[1] + a[2] + \ldots + a[i]$$

Korišćenje:

Uz pomoć prefiksnih zbirova, zbir elemenata niza a[l],a[l+1],…,a[r]a[l], a[l+1],…, a[r] može se efikasno izračunati kao:

$$Zbir(l, r) = P[r] - P[l - 1]$$

Gde P[l−1] predstavlja zbir elemenata pre l.

Implementacija prefiksnog zbira u C++:

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

int main() {
    vector<int> a = {3, 2, 1, 4, 5}; // Originalni niz
    int n = a.size();

    vector<int> prefiksni(n + 1, 0); // Prefiksni niz (0-indeksiran)

    // Računanje prefiksnih zbirova
    for (int i = 1; i <= n; i++) {
        prefiksni[i] = prefiksni[i - 1] + a[i - 1];
    }

    // Ispisivanje prefiksnih zbirova
    cout << "Prefiksni zbirovi: ";
    for (int i = 1; i <= n; i++) {
        cout << prefiksni[i] << " ";
    }
    cout << endl;

    // Efikasno računanje zbira elemenata na opsegu (l, r)
    int l = 2, r = 4; // Indeksi 1-bazirani
    int zbir = prefiksni[r] - prefiksni[l - 1];
    cout << "Zbir elemenata od " << l << " do " << r << " je: " << zbir << endl;

    return 0;
}
```

Sufiksni zbir

Sufiksni zbir za niz je niz u kojem je svaki element zbir svih elemenata originalnog niza od tog indeksa do kraja niza.

Definicija:

Ako imamo niz a[1],a[2],…,a[n]a[1], a[2],…, a[n], sufiksni zbir S[i] definišemo kao:

$$S[i] = a[i] + a[i+1] + \ldots + a[n]$$

Korišćenje:

Sufiksni zbirovi se koriste kada je potrebno računati zbirove elemenata od određenog indeksa do kraja niza.

Implementacija sufiksnog zbira u C++:

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

int main() {
    vector<int> a = {3, 2, 1, 4, 5}; // Originalni niz
    int n = a.size();

    vector<int> sufiksni(n + 1, 0); // Sufiksni niz (0-indeksiran)

    // Računanje sufiksnih zbirova
    for (int i = n - 1; i >= 0; i--) {
        sufiksni[i] = sufiksni[i + 1] + a[i];
    }

    // Ispisivanje sufiksnih zbirova
    cout << "Sufiksni zbirovi: ";
    for (int i = 0; i < n; i++) {
        cout << sufiksni[i] << " ";
    }
    cout << endl;

    // Zbir elemenata od indeksa `i` do kraja niza
    int i = 2; // Indeks 0-baziran
    int zbir = sufiksni[i];
    cout << "Zbir elemenata od indeksa " << i << " do kraja je: " << zbir << endl;

    return 0;
}
```

Primene prefiksnih i sufiksnih zbirova:

1. Efikasni upiti na opsegu:
    - Korišćenjem prefiksnih zbirova, opseg zbirova O(1) može se izračunati umesto O(n) iteracija.
2. Algoritmi pretraživanja:
    - Prefiksni i sufiksni zbirovi su ključni u mnogim algoritmima za rešavanje problema poput podnizova sa maksimalnim zbirom.
3. Razni problemi:
    - Problemi sa ravnotežom niza, simetrijom ili analiza subnizova.

Inkrementalnost

Inkrementalnost je tehnika u programiranju gde koristimo prethodno izračunate veličine da bismo izbegli nepotrebno ponovno računanje u sledećim koracima. Ova tehnika često poboljšava efikasnost algoritama i smanjuje vreme izvršavanja.

Osnovni koncept inkrementalnosti

Umesto da ponovo izračunavamo sve vrednosti od početka, inkrementalni pristup koristi ranije izračunate rezultate i prilagođava ih prema promenama u ulazu. Ovaj koncept je često primenjen u raznim problemima poput:

- Računanja zbirova u opsegu
- Generisanja sekvenci
- Algoritama za pretragu i sortiranje
- Dinamičkog programiranja

Primer 1: Zbir niza sa inkrementalnim pristupom

Ako želimo da izračunamo zbir elemenata niza nakon svake promene, možemo sačuvati prethodni zbir i samo dodati ili oduzeti novu vrednost.

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

int main() {
    vector<int> niz = {1, 2, 3, 4, 5}; // Originalni niz
    int zbir = 0;

    // Računanje početnog zbira
    for (int broj : niz) {
        zbir += broj;
    }
    cout << "Početni zbir: " << zbir << endl;

    // Dodavanje novog elementa u niz
    int noviBroj = 6;
    niz.push_back(noviBroj);
    zbir += noviBroj; // Ažuriramo zbir inkrementalno
    cout << "Zbir nakon dodavanja " << noviBroj << ": " << zbir << endl;

    // Uklanjanje poslednjeg elementa
    zbir -= niz.back(); // Oduzimamo poslednji element
    niz.pop_back();     // Uklanjamo element iz niza
    cout << "Zbir nakon uklanjanja poslednjeg elementa: " << zbir << endl;

    return 0;
}
```

Prednosti:

- Umesto da prolazimo kroz ceo niz za svaku izmenu (O(n)), ažuriranje je konstantno (O(1)).

Primer 2: Prefiksni zbir kao osnova inkrementalnog računanja

Prefiksni zbirovi (koje smo ranije objasnili) su prirodna primena inkrementalnosti jer svaka nova vrednost koristi prethodni zbir i dodaje trenutni element.

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

int main() {
    vector<int> niz = {1, 2, 3, 4, 5};
    vector<int> prefiksni(niz.size(), 0);

    // Računanje prefiksnog zbira inkrementalno
    prefiksni[0] = niz[0];
    for (size_t i = 1; i < niz.size(); i++) {
        prefiksni[i] = prefiksni[i - 1] + niz[i];
    }

    // Ispisivanje prefiksnih zbirova
    cout << "Prefiksni zbirovi: ";
    for (int broj : prefiksni) {
        cout << broj << " ";
    }
    cout << endl;

    return 0;
}
```

Primer 3: Fibonacci niz sa inkrementalnim računanjem

Umesto da ponovo računamo sve prethodne vrednosti, možemo koristiti samo poslednje dve vrednosti da bismo izračunali sledeću.

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

int main() {
    int n = 10; // Prvih n brojeva Fibonacci niza

    int prethodni = 0, trenutni = 1;
    cout << "Fibonacci niz: " << prethodni << " " << trenutni << " ";

    for (int i = 2; i < n; i++) {
        int sledeci = prethodni + trenutni;
        cout << sledeci << " ";
        prethodni = trenutni;
        trenutni = sledeci;
    }
    cout << endl;

    return 0;
}
```

Prednosti:

- Umesto čuvanja celog niza, koristimo samo dve promenljive za poslednje dve vrednosti (O(1) memorijski prostor).

Primer 4: Dinamičko programiranje (npr. problem knapsaka)

U dinamičkom programiranju, rezultati manjih podproblema se koriste za rešavanje većih. Ova tehnika je prirodna primena inkrementalnosti.

Primer problema: Maksimizacija vrednosti predmeta u ruksaku sa ograničenom težinom.

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

int main() {
    int tezinaRuksaka = 10;
    vector<int> tezine = {1, 3, 4, 6};
    vector<int> vrednosti = {10, 40, 50, 70};

    vector<int> dp(tezinaRuksaka + 1, 0); // Dinamička tabela

    // Popunjavanje tabele inkrementalno
    for (size_t i = 0; i < tezine.size(); i++) {
        for (int w = tezinaRuksaka; w >= tezine[i]; w--) {
            dp[w] = max(dp[w], dp[w - tezine[i]] + vrednosti[i]);
        }
    }

    cout << "Maksimalna vrednost ruksaka: " << dp[tezinaRuksaka] << endl;

    return 0;
}
```

Prednosti inkrementalnog pristupa

1. Efikasnost: Eliminacija nepotrebnog računanja smanjuje kompleksnost algoritma.
2. Manje memorije: Manje podataka se čuva kada koristimo prethodne vrednosti umesto celih struktura.
3. Lakoća implementacije: Korišćenje prethodnih rezultata često pojednostavljuje kod.

Tehnika dva pokazivača u C++

Tehnika dva pokazivača (engl. *Two Pointer Technique*) je algoritamski pristup koji koristi dva pokazivača (indeksa) za efikasno rešavanje problema sa nizovima ili listama. Ova tehnika je posebno korisna kada radimo sa sortiranim podacima ili problemima koji uključuju opsege, podnizove ili parove elemenata.

Osnovni koncept

Dva pokazivača se postavljaju na različite pozicije u nizu:

1. Levi pokazivač (*left pointer*): često započinje na početku niza.
2. Desni pokazivač (*right pointer*): može započeti na kraju niza (ili na drugom mestu).

Pokazivači se pomeraju prema određenim pravilima kako bi postigli željeni rezultat, često smanjujući broj potrebnih iteracija.

Primene tehnike dva pokazivača

1. Pronalaženje parova sa određenim zbirom u sortiranom nizu
2. Pronalaženje maksimalnog ili minimalnog podniza sa određenim svojstvom
3. Proveravanje da li je niz palindrom
4. Spajanje dva sortirana niza
5. Rešavanje problema sa podnizovima ili podsekvencama

Primer 1: Pronalaženje para sa određenim zbirom

Ako je niz sortiran, možemo koristiti tehniku dva pokazivača za pronalaženje para elemenata čiji zbir daje zadatu vrednost S.

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

bool pronadjiPar(const vector<int>& niz, int S) {
    int left = 0;                  // Levi pokazivač
    int right = niz.size() - 1;    // Desni pokazivač

    while (left < right) {
        int zbir = niz[left] + niz[right];
        if (zbir == S) {
            cout << "Par pronađen: (" << niz[left] << ", " << niz[right] << ")" << endl;
            return true;
        } else if (zbir < S) {
            left++; // Povećaj levi pokazivač
        } else {
            right--; // Smanji desni pokazivač
        }
    }
    cout << "Nije pronađen par sa zbirom " << S << endl;
    return false;
}

int main() {
    vector<int> niz = {1, 2, 3, 4, 6, 8, 10};
    int S = 10;
    pronadjiPar(niz, S);
    return 0;
}
```

Objašnjenje:

- Levi i desni pokazivači se pomeraju prema vrednosti zbira.
- Kompleksnost je O(n)O(n), jer svaki element niza bude obrađen najviše jednom.

Primer 2: Proveravanje palindroma

Tehnika dva pokazivača može se koristiti za proveru da li je niz ili string palindrom (čita se isto s leva na desno i obrnuto).

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

bool jePalindrom(const string& s) {
    int left = 0;
    int right = s.size() - 1;

    while (left < right) {
        if (s[left] != s[right]) {
            return false; // Nije palindrom
        }
        left++;
        right--;
    }
    return true;
}

int main() {
    string rec = "radar";
    if (jePalindrom(rec)) {
        cout << "String \"" << rec << "\" je palindrom." << endl;
    } else {
        cout << "String \"" << rec << "\" nije palindrom." << endl;
    }
    return 0;
}
```

Objašnjenje:

- Levi pokazivač kreće od početka, a desni od kraja stringa.
- Pokazivači se približavaju jedan drugom dok se ne susretnu.

Primer 3: Maksimalni podniz sa zbirom manjim od SS

Koristi se dva pokazivača za pronalaženje podniza čiji zbir elemenata ostaje manji od zadate vrednosti S.

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

int maksimalniPodniz(const vector<int>& niz, int S) {
    int left = 0, zbir = 0, maxDuzina = 0;

    for (int right = 0; right < niz.size(); right++) {
        zbir += niz[right];

        while (zbir >= S) {
            zbir -= niz[left];
            left++; // Pomeri levi pokazivač
        }

        maxDuzina = max(maxDuzina, right - left + 1);
    }

    return maxDuzina;
}

int main() {
    vector<int> niz = {1, 2, 3, 4, 5};
    int S = 8;
    cout << "Maksimalna dužina podniza sa zbirom manjim od " << S << " je: "
         << maksimalniPodniz(niz, S) << endl;

    return 0;
}
```

Objašnjenje:

- Levi pokazivač pomera se kada zbir postane veći ili jednak SS.
- Desni pokazivač iterira kroz niz i širi opseg.

Prednosti tehnike dva pokazivača

1. Efikasnost:
    - Obično radi u O(n) vremenskoj kompleksnosti za mnoge probleme.
2. Jednostavnost:
    - Pogodna za iteracije kroz niz ili string sa logikom koja zavisi od relativnih pozicija elemenata.
3. Manje memorije:
    - Ne zahteva dodatne strukture podataka.

<u>Kada koristiti tehniku dva pokazivača?</u>

- Kada su podaci sortirani ili je moguće postići sortiranje pre obrade.
- Kada je cilj pronaći parove, podnizove ili sekvence koje zadovoljavaju određene uslove.

Odsecanje u linearnoj pretrazi u C++

Odsecanje (engl. *pruning*) je tehnika u programiranju kojom se izbegava pretraga u delu podataka za koji znamo ili možemo zaključiti da ne sadrži rešenje. Ovo smanjuje vreme izvršavanja pretrage tako što eliminišemo nepotrebne korake.

Osnovni koncept

U klasičnoj linearnoj pretrazi, prolazimo kroz svaki element niza dok ne pronađemo cilj. Kod odsecanja, koristimo logiku ili dodatne informacije o podacima kako bismo prekinuli pretragu ranije ili preskočili nepotrebne delove.

Odsecanje se najčešće primenjuje kada:

1. Niz ili podaci imaju neku strukturu (npr. sortirani niz).
2. Postoje ograničenja koja omogućavaju isključivanje određenih delova pretrage.

Primer 1: Linearna pretraga sa odsecanjem u sortiranim nizovima

Ako tražimo određeni element u **sortiranom nizu**, možemo prekinuti pretragu čim naiđemo na element koji je veći od traženog.

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

bool linearnaPretragaOdsecanje(const vector<int>& niz, int cilj) {
    for (int broj : niz) {
        if (broj > cilj) {
            // Odsecanje: nema smisla nastaviti pretragu
            break;
        }
        if (broj == cilj) {
            return true;
        }
    }
    return false;
}

int main() {
    vector<int> niz = {1, 3, 5, 7, 9}; // Sortiran niz
    int cilj = 5;

    if (linearnaPretragaOdsecanje(niz, cilj)) {
        cout << "Element " << cilj << " pronađen u nizu." << endl;
    } else {
        cout << "Element " << cilj << " nije pronađen." << endl;
    }

    return 0;
}
```

Objašnjenje:

- Kad naiđemo na element veći od traženog, prekinemo pretragu jer znamo da nema smisla nastaviti (zbog sortiranja).

Primer 2: Traženje zbira u opsegu sa odsecanjem

Kada tražimo podniz sa zbirom manjim od zadate vrednosti, možemo prestati da dodajemo elemente kada zbir pređe zadatu granicu.

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

bool postojiZbirManjiOd(const vector<int>& niz, int granica) {
    int zbir = 0;

    for (int broj : niz) {
        zbir += broj;

        if (zbir >= granica) {
            // Odsecanje: zbir je već veći ili jednak granici
            return false;
        }
    }

    return true; // Ako smo prošli kroz niz i zbir ostao ispod granice
}

int main() {
    vector<int> niz = {1, 2, 3, 4};
    int granica = 10;

    if (postojiZbirManjiOd(niz, granica)) {
        cout << "Zbir elemenata je manji od " << granica << "." << endl;
    } else {
        cout << "Zbir elemenata prelazi " << granica << "." << endl;
    }

    return 0;
}
```

Objašnjenje:

- Prekidamo dodavanje u zbir čim pređe granicu jer nema smisla nastaviti dalje.

Primer 3: Pretraga maksimalnog podniza sa ograničenjem

Tražimo najduži podniz čiji zbir ne prelazi zadatu vrednost SS. Koristimo odsecanje da preskočimo nepotrebne iteracije.

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

int najduziPodnizSaGranicom(const vector<int>& niz, int S) {
    int left = 0, zbir = 0, maxDuzina = 0;

    for (int right = 0; right < niz.size(); right++) {
        zbir += niz[right];

        // Odsecanje: smanjujemo zbir dok ne bude ispod granice
        while (zbir > S) {
            zbir -= niz[left];
            left++;
        }

        maxDuzina = max(maxDuzina, right - left + 1);
    }

    return maxDuzina;
}

int main() {
    vector<int> niz = {1, 2, 3, 4, 5};
    int S = 8;

    cout << "Najduži podniz sa zbirom manjim ili jednakim " << S << " je: "
         << najduziPodnizSaGranicom(niz, S) << endl;

    return 0;
}
```

Objašnjenje:

- Levi pokazivač se pomera kad zbir pređe zadatu granicu, čime preskačemo nepotrebne elemente.

Primer 4: Backtracking sa odsecanjem

Kod algoritama poput rešavanja slagalice (npr. *N-queens problem*), koristimo odsecanje da prekinemo pretragu kad znamo da trenutni put ne vodi rešenju.

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

bool validnoPostavljanje(const vector<int>& tabla, int red, int kolona) {
    for (int i = 0; i < red; i++) {
        if (tabla[i] == kolona || abs(tabla[i] - kolona) == abs(i - red)) {
            return false; // Odsecanje: sukob sa postojećom kraljicom
        }
    }
    return true;
}

void resiNQueens(int n, vector<int>& tabla, int red) {
    if (red == n) {
        // Pronađeno rešenje
        for (int kolona : tabla) {
            cout << kolona << " ";
        }
        cout << endl;
        return;
    }

    for (int kolona = 0; kolona < n; kolona++) {
        if (validnoPostavljanje(tabla, red, kolona)) {
            tabla[red] = kolona;
            resiNQueens(n, tabla, red + 1);
        }
    }
}

int main() {
    int n = 4; // Broj kraljica
    vector<int> tabla(n, -1);

    cout << "Rešenja za " << n << "-queens problem:" << endl;
    resiNQueens(n, tabla, 0);

    return 0;
}
```

Objašnjenje**:**

- Sukobi među kraljicama prepoznaju se unapred, čime se eliminišu grane pretrage koje ne vode rešenju.

Prednosti odsecanja

1. Efikasnost: Izbegavanje nepotrebnih iteracija značajno smanjuje vreme izvršavanja.
2. Jednostavnija implementacija: Logičko odsecanje olakšava rukovanje velikim nizovima podataka.
3. Manje memorijsko opterećenje: Odsecanje može smanjiti potrebu za dodatnim podacima.

Brojevni sistemi u C++

Brojevni sistemi su različiti načini predstavljanja brojeva korišćenjem različitih baza. Najčešći sistemi su:

1. **Decimalni sistem (osnova 10):** koristi cifre 0−90-9.
2. **Binarni sistem (osnova 2):** koristi cifre 00 i 11.
3. **Oktalni sistem (osnova 8):** koristi cifre 0−70-7.
4. **Heksadekadni sistem (osnova 16):** koristi cifre 0−90-9 i slova A−F (gde A=10, B=11, itd.).

C++ omogućava rad sa ovim brojevnim sistemima na više načina, uključujući unos, prikaz i konverziju između sistema.

Rad sa literalima u različitim sistemima

U C++, brojevi u različitim bazama mogu se zapisivati na sledeće načine:

- **Decimalni:** Podrazumevani format.
  Primer: int broj = 42;
- **Binarni:** Dodavanjem prefiksa 0b ili 0B.
  Primer: int broj = 0b101010; (ekvivalentno decimalnom broju 42)
- **Oktalni:** Dodavanjem prefiksa 0.
  Primer: int broj = 052; (ekvivalentno decimalnom broju 42)
- **Heksadekadni:** Dodavanjem prefiksa 0x ili 0X.
  Primer: int broj = 0x2A; (ekvivalentno decimalnom broju 42)

Prikaz brojeva u različitim sistemima

Biblioteka <iomanip> omogućava formatiranje izlaza brojeva. Koriste se manipulatore std::dec, std::hex i std::oct za prikaz u decimalnom, heksadekadnom i oktalnom formatu.

Primer:

```cpp
#include <iostream>
#include <iomanip> // Za manipulatore
using namespace std;

int main() {
    int broj = 42;

    cout << "Decimalni: " << dec << broj << endl;
    cout << "Heksadekadni: " << hex << broj << endl;
    cout << "Oktalni: " << oct << broj << endl;

    return 0;
}
```

Izlaz:

Decimalni: 42

Heksadekadni: 2a

Oktalni: 52

Konverzije između brojevnih sistema

Možemo ručno konvertovati brojeve ili koristiti ugrađene funkcije za manipulaciju stringovima. Na primer, std::bitset za binarne prikaze i std::stringstream za heksadekadne konverzije.

Konverzija decimalnog broja u binarni:

```
#include <iostream>
#include <bitset> // Za binarni prikaz
using namespace std;

int main() {
    int broj = 42;

    cout << "Decimalni: " << broj << endl;
    cout << "Binarni: " << bitset<8>(broj) << endl; // Prikazuje 8 bitova

    return 0;
}
```

Izlaz:

Decimalni: 42

Binarni: 00101010

Konverzija heksadekadnog u decimalni i obrnuto:

```
#include <iostream>
#include <sstream> // Za string manipulaciju
using namespace std;

int main() {
    stringstream ss;
    int decimalniBroj = 42;

    // Decimalni u heksadekadni
    ss << hex << decimalniBroj;
    string heksadekadniBroj = ss.str();
    cout << "Heksadekadni: " << heksadekadniBroj << endl;

    // Heksadekadni u decimalni
    int konvertovaniBroj;
    ss.clear(); // Čisti stream
    ss.str("2A"); // Heksadekadni broj kao string
    ss >> hex >> konvertovaniBroj;
    cout << "Decimalni: " << konvertovaniBroj << endl;

    return 0;
}
```

Izlaz:

Heksadekadni: 2a

Decimalni: 42

Operacije u različitim sistemima

Brojevi u različitim sistemima mogu se direktno koristiti u izrazima jer se svi brojevi interno čuvaju kao binarni.

Primer: Operacije sa binarnim brojevima

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

int main() {
    int a = 0b1010; // 10 u decimalnom
    int b = 0b0101; // 5 u decimalnom

    cout << "a + b = " << (a + b) << endl;  // Decimalni zbir
    cout << "a & b = " << bitset<4>(a & b) << endl; // Binarni "AND"
    cout << "a | b = " << bitset<4>(a | b) << endl; // Binarni "OR"

    return 0;
}
```

Izlaz:

a + b = 15

a & b = 0000

a | b = 1111

Primer za konverziju brojeva

Konverzija binarnog u decimalni:

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

int binarniUDekadni(int binarni) {
    int decimalni = 0, baza = 1;

    while (binarni > 0) {
        int poslednjaCifra = binarni % 10;
        decimalni += poslednjaCifra * baza;
        baza *= 2;
        binarni /= 10;
    }

    return decimalni;
}

int main() {
    int binarniBroj = 101010;
    cout << "Decimalni ekvivalent binarnog broja " << binarniBroj << " je: "
         << binarniUDekadni(binarniBroj) << endl;

    return 0;
}
```

Izlaz:

Decimalni ekvivalent binarnog broja 101010 je: 42

Zaključak

- Literalni prefiksi: 0b za binarni, 0 za oktalni, 0x za heksadekadni.
- Prikaz brojeva: Koristite manipulatore std::hex, std::oct, std::dec.
- Konverzije: Ugrađene klase (std::bitset) ili ručne funkcije za prelaz između sistema.

Upotreba algoritma za sortiranje podataka u C++

C++ standardna biblioteka (<algorithm>) pruža moćne alate za sortiranje podataka bilo kog tipa. Glavna funkcija za sortiranje je **std::sort**, koja omogućava sortiranje nizova, vektora i drugih kontejnera po zadatim kriterijumima.

Osnovna sintaksa std::sort

Funkcija std::sort se nalazi u <algorithm> biblioteci i ima sledeću sintaksu:

```
std::sort(početak, kraj, kriterijum);
```

- **početak** i **kraj** su iteratori koji definišu opseg elemenata za sortiranje.
- **kriterijum** je opciona funkcija ili lambda izraz koji određuje pravilo sortiranja. Ako se izostavi, koristi se podrazumevani redosled (uzlazno).

Sortiranje osnovnih tipova podataka

Primer: Sortiranje vektora celih brojeva

```
#include <iostream>
#include <vector>
#include <algorithm> // Za std::sort
using namespace std;

int main() {
    vector<int> brojevi = {5, 2, 9, 1, 5, 6};

    // Sortiranje u rastućem redosledu
    sort(brojevi.begin(), brojevi.end());

    cout << "Sortirano (rastuće): ";
    for (int broj : brojevi) {
        cout << broj << " ";
    }
    cout << endl;

    // Sortiranje u opadajućem redosledu
    sort(brojevi.begin(), brojevi.end(), greater<int>());

    cout << "Sortirano (opadajuće): ";
    for (int broj : brojevi) {
        cout << broj << " ";
    }
    cout << endl;

    return 0;
}
```

Izlaz:

Sortirano (rastuće): 1 2 5 5 6 9

Sortirano (opadajuće): 9 6 5 5 2 1

Sortiranje po prilagođenom kriterijumu

Možemo definisati sopstvene kriterijume sortiranja koristeći funkcije ili lambda izraze.

Primer: Sortiranje niza stringova po dužini

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

int main() {
    vector<string> reci = {"banana", "jabuka", "kivi", "ananas", "jagoda"};

    // Sortiranje po dužini stringa (rastuće)
    sort(reci.begin(), reci.end(), [](const string& a, const string& b) {
        return a.size() < b.size();
    });

    cout << "Sortirano po dužini (rastuće): ";
    for (const string& rec : reci) {
        cout << rec << " ";
    }
    cout << endl;

    // Sortiranje po dužini (opadajuće)
    sort(reci.begin(), reci.end(), [](const string& a, const string& b) {
        return a.size() > b.size();
    });

    cout << "Sortirano po dužini (opadajuće): ";
    for (const string& rec : reci) {
        cout << rec << " ";
    }
    cout << endl;

    return 0;
}
```

Izlaz:

Sortirano po dužini (rastuće): kivi banana jabuka jagoda ananas

Sortirano po dužini (opadajuće): ananas banana jabuka jagoda kivi

Sortiranje struktura i objekata

Kod sortiranja struktura ili objekata, često je potrebno definisati kriterijum na osnovu jednog ili više atributa.

Primer: Sortiranje struktura po jednom atributu

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

struct Student {
    string ime;
    int poeni;
};

int main() {
    vector<Student> studenti = {
        {"Ana", 85}, {"Marko", 92}, {"Jovana", 76}, {"Petar", 92}, {"Milica", 89}};

    // Sortiranje po broju poena (opadajuće)
    sort(studenti.begin(), studenti.end(), [](const Student& a, const Student& b) {
        return a.poeni > b.poeni; // Veći poeni imaju prednost
    });

    cout << "Sortirani studenti po poenima:" << endl;
    for (const Student& s : studenti) {
        cout << s.ime << " - " << s.poeni << endl;
    }

    return 0;
}
```

Izlaz:

Sortirani studenti po poenima:

Marko - 92

Petar - 92

Milica - 89

Ana - 85

Jovana - 76

Sortiranje po više kriterijuma

Ako je potrebno sortirati po više atributa (npr. prvo po poenima, zatim po imenu), kombinujemo uslove:

```
sort(studenti.begin(), studenti.end(), [](const Student& a, const Student& b) {
    if (a.poeni == b.poeni) {
        return a.ime < b.ime; // Sortiraj po imenu ako su poeni isti
    }
    return a.poeni > b.poeni; // Inače, sortiraj po poenima
});
```

Stabilno sortiranje sa std::stable_sort

Funkcija **std::stable_sort** čuva relativni redosled elemenata koji su jednaki prema kriterijumu. Ovo je korisno kada želimo očuvati inicijalni redosled.

Primer: Stabilno sortiranje

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

struct Osoba {
    string ime;
    int godina;
};

int main() {
    vector<Osoba> osobe = {
        {"Marko", 25}, {"Ana", 30}, {"Jovana", 25}, {"Petar", 20}};

    // Stabilno sortiranje po godinama
    stable_sort(osobe.begin(), osobe.end(), [](const Osoba& a, const Osoba& b) {
        return a.godina < b.godina;
    });

    cout << "Sortirane osobe po godinama (stabilno):" << endl;
    for (const Osoba& o : osobe) {
        cout << o.ime << " - " << o.godina << endl;
    }

    return 0;
}
```

Izlaz:

Sortirane osobe po godinama (stabilno):

Petar - 20

Marko - 25

Jovana - 25

Ana - 30

Napredni primer: Sortiranje mapa prema vrednostima

Sortiranje mapa prema vrednostima zahteva kopiranje elemenata u vektor parova.

Primer: Sortiranje mape prema vrednostima

```
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    map<string, int> mapa = {{"Ana", 85}, {"Marko", 92}, {"Jovana", 76}, {"Petar", 89}};

    // Kopiraj elemente mape u vektor parova
    vector<pair<string, int>> parovi(mapa.begin(), mapa.end());

    // Sortiraj po vrednostima (opadajuće)
    sort(parovi.begin(), parovi.end(), [](const pair<string, int>& a, const pair<string, int>&
b) {
        return a.second > b.second;
    });

    cout << "Sortirana mapa prema vrednostima:" << endl;
    for (const auto& p : parovi) {
        cout << p.first << " - " << p.second << endl;
    }

    return 0;
}
```

Izlaz:

Sortirana mapa prema vrednostima:

Marko - 92

Petar - 89

Ana - 85

Jovana - 76

Zaključak

1. std::sort omogućava jednostavno i fleksibilno sortiranje podataka.
2. Lambda izrazi su vrlo korisni za definisanje prilagođenih kriterijuma.
3. std::stable_sort čuva inicijalni redosled elemenata jednakih prema kriterijumu.
4. Sortiranje se može prilagoditi za rad sa strukturama, objektima i čak složenim kontejnerima poput mapa.

Sortiranje prebrojavanjem (Counting Sort)

Sortiranje prebrojavanjem je algoritam za sortiranje koji funkcioniše tako što prebrojava pojave svakog elementa i koristi te informacije za rekonstrukciju sortiranog niza. Efikasan je za podatke s ograničenim opsegom vrednosti.

Karakteristike:

- **Kompleksnost:** O(n+k), gde je nn broj elemenata, a k opseg vrednosti (maksimalna vrednost - minimalna vrednost).
- **Stabilan:** Može biti stabilan ako se implementira uz očuvanje relativnog redosleda.
- **Primena:** Najpogodniji za sortiranje celih brojeva u poznatom opsegu.

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

void countingSort(vector<int>& arr) {
    // Pronađi maksimalni element
    int maxElement = *max_element(arr.begin(), arr.end());

    // Kreiraj pomoćni niz za prebrojavanje
    vector<int> count(maxElement + 1, 0);

    // Prebroj pojave svakog elementa
    for (int num : arr) {
        count[num]++;
    }

    // Rekonstruiši sortiran niz
    int index = 0;
    for (int i = 0; i <= maxElement; ++i) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}

int main() {
    vector<int> arr = {4, 2, 2, 8, 3, 3, 1};

    cout << "Originalni niz: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    countingSort(arr);

    cout << "Sortiran niz: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Izlaz:

Originalni niz: 4 2 2 8 3 3 1

Sortiran niz: 1 2 2 3 3 4 8

Razvrstavanje (Radix Sort)

Radix Sort je algoritam za sortiranje koji radi tako što sortira elemente prema njihovim ciframa, od najmanje značajne do najznačajnije (ili obrnuto). Najčešće koristi **Sortiranje prebrojavanjem** kao pomoćni algoritam.

Karakteristike:

- **Kompleksnost:** O(d×(n+b)), gde je d broj cifara u najvećem broju, n broj elemenata, a b baza (najčešće 10).
- **Stabilan:** Da, ako je interno sortiranje stabilno.
- **Primena:** Koristi se za brojeve, stringove ili podatke koji mogu biti predstavljeni ciframa.

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

// Pomoćna funkcija za sortiranje po cifri
void countingSortByDigit(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n); // Sortirani niz
    vector<int> count(10, 0); // Brojač za cifre (0-9)

    // Prebroj pojave svake cifre
    for (int num : arr) {
        int digit = (num / exp) % 10;
        count[digit]++;
    }

    // Modifikuj brojač za pozicije
    for (int i = 1; i < 10; ++i) {
        count[i] += count[i - 1];
    }

    // Popuni izlazni niz
    for (int i = n - 1; i >= 0; --i) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Kopiraj rezultat u originalni niz
    for (int i = 0; i < n; ++i) {
        arr[i] = output[i];
    }
}

// Glavna funkcija za Radix Sort
void radixSort(vector<int>& arr) {
    // Pronađi maksimalni element
    int maxElement = *max_element(arr.begin(), arr.end());
```

```
    // Sortiraj po svakoj cifri (jedinice, desetice, stotine...)
    for (int exp = 1; maxElement / exp > 0; exp *= 10) {
        countingSortByDigit(arr, exp);
    }
}

int main() {
    vector<int> arr = {170, 45, 75, 90, 802, 24, 2, 66};

    cout << "Originalni niz: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    radixSort(arr);

    cout << "Sortiran niz: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Izlaz:

Originalni niz: 170 45 75 90 802 24 2 66

Sortiran niz: 2 24 45 66 75 90 170 802

Ključne razlike između Counting Sort i Radix Sort

Karakteristika	**Counting Sort**	**Radix Sort**
Kompleksnost	O(n+k)	O(d×(n+b))
Primena	Celobrojni podaci u malom opsegu	Veliki brojevi, stringovi
Stabilnost	Stabilan uz prilagodbu	Stabilan
Potrošnja memorije	Može zahtevati veliki pomoćni niz	Više iteracija po manjem opsegu

Efikasna pretraga sortiranog niza i binarna pretraga u C++

Binarna pretraga je algoritam za brzo pronalaženje pozicije ciljne vrednosti u sortiranom nizu. Funkcioniše tako što se srednji element niza poredi s ciljnim elementom i zatim se pretraga nastavlja samo u delu niza gde bi cilj mogao biti.

Karakteristike:

1. **Kompleksnost:**
   - **Najbolji slučaj:** O(1) (cilj je srednji element).
   - **Prosečni i najgori slučaj:** O(log n), gde je n broj elemenata.
2. **Ograničenja:** Niz mora biti sortiran pre primene algoritma.
3. **Primena:** Koristi se za pretragu u velikim sortiranima nizovima ili strukturama.

Osnovna implementacija binarne pretrage

Iterativna verzija:

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

int binarySearch(const vector<int>& arr, int target) {
    int low = 0, high = arr.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2; // Izbegavanje prelivanja

        if (arr[mid] == target) {
            return mid; // Nađeno
        }
        if (arr[mid] < target) {
            low = mid + 1; // Traži u desnoj polovini
        } else {
            high = mid - 1; // Traži u levoj polovini
        }
    }
    return -1; // Nije pronađeno
}

int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11, 13, 15};

    int target = 7;
    int result = binarySearch(arr, target);

    if (result != -1) {
        cout << "Element " << target << " pronađen na indeksu " << result << endl;
    } else {
        cout << "Element " << target << " nije pronađen." << endl;
    }

    return 0;
}
```

Izlaz: Element 7 pronađen na indeksu 3

Rekurzivna verzija:

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

int binarySearchRecursive(const vector<int>& arr, int low, int high, int target) {
    if (low > high) {
        return -1; // Nije pronađeno
    }

    int mid = low + (high - low) / 2;

    if (arr[mid] == target) {
        return mid; // Nađeno
    }
    if (arr[mid] < target) {
        return binarySearchRecursive(arr, mid + 1, high, target); // Desna polovina
    } else {
        return binarySearchRecursive(arr, low, mid - 1, target); // Leva polovina
    }
}

int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11, 13, 15};

    int target = 11;
    int result = binarySearchRecursive(arr, 0, arr.size() - 1, target);

    if (result != -1) {
        cout << "Element " << target << " pronađen na indeksu " << result << endl;
    } else {
        cout << "Element " << target << " nije pronađen." << endl;
    }

    return 0;
}
```

Izlaz:

Element 11 pronađen na indeksu 5

Varijante problema s binarnom pretragom

1. **Prva pojava elementa:** Modifikovana binarna pretraga koja pronalazi prvu pojavu ciljne vrednosti u nizu sa duplikatima.

```
int firstOccurrence(const vector<int>& arr, int target) {
    int low = 0, high = arr.size() - 1;
    int result = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target) {
            result = mid; // Zabeleži indeks
            high = mid - 1; // Nastavi traženje u levoj polovini
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}
```

2. **Najmanji element veći od cilja (Lower Bound):** Pronalaženje najmanjeg elementa koji je veći ili jednak cilju.

```
int lowerBound(const vector<int>& arr, int target) {
    int low = 0, high = arr.size() - 1;
    int result = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] >= target) {
            result = mid; // Zabeleži indeks
            high = mid - 1; // Traži u levoj polovini
        } else {
            low = mid + 1;
        }
    }
    return result;
}
```

3. **Najveći element manji od cilja (Upper Bound):** Pronalaženje najvećeg elementa koji je manji ili jednak cilju.

```
int upperBound(const vector<int>& arr, int target) {
    int low = 0, high = arr.size() - 1;
    int result = -1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] <= target) {
            result = mid; // Zabeleži indeks
            low = mid + 1; // Traži u desnoj polovini
        } else {
            high = mid - 1;
        }
    }
    return result;
}
```

Prednosti i mane binarne pretrage

Prednosti	**Mane**
Brza (O(log n))	Niz mora biti sortiran
Jednostavna implementacija	Neefikasna za male nizove
Pogodna za statične podatke	Teška za dinamičke strukture

Pronalaženje poslednjeg elementa koji ne zadovoljava ili prvog koji zadovoljava uslov u uređenom nizu

Ovaj problem se često rešava modifikovanom binarnom pretragom. Ako imamo sortiran niz u kojem su elementi podeljeni prema nekom uslovu, cilj je pronaći:

- **Poslednji element koji ne zadovoljava uslov**.
- **Prvi element koji zadovoljava uslov**.

Problem u kontekstu:

Pretpostavimo da niz ima sledeću strukturu:

- Svi elementi pre nekog indeksa ii **ne zadovoljavaju** uslov.
- Svi elementi od indeksa ii nadalje **zadovoljavaju** uslov.

Binarnom pretragom možemo efikasno pronaći tačku prelaza između elemenata koji ne zadovoljavaju i onih koji zadovoljavaju uslov.

Generalni algoritam

1. Inicijalizujemo dve granice: low=0 i high=n−1.
2. Izračunavamo sredinu: mid=low+(high−low)/2.
3. Koristimo uslov da odlučimo da li da pretragu suzimo na levi ili desni deo niza:
    - Ako element **zadovoljava** uslov, tražimo dalje u levoj polovini.
    - Ako element **ne zadovoljava**, tražimo dalje u desnoj polovini.

Na kraju dobijamo:

- **Poslednji element koji ne zadovoljava uslov** tako što čuvamo indeks dok pretraga napreduje.
- **Prvi element koji zadovoljava uslov** direktno kroz granicu lowlow.

Pronalaženje poslednjeg elementa koji ne zadovoljava uslov

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

// Uslov za proveru
bool zadovoljava(int x) {
    return x >= 10; // Primer: zadovoljava ako je >= 10
}

int poslednjiKojiNeZadovoljava(const vector<int>& arr) {
    int low = 0, high = arr.size() - 1;
    int result = -1; // Indeks poslednjeg koji ne zadovoljava

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (zadovoljava(arr[mid])) {
            high = mid - 1; // Idi levo, jer mid zadovoljava
        } else {
            result = mid; // Zabeleži indeks
            low = mid + 1; // Idi desno
        }
    }

    return result;
}

int main() {
    vector<int> arr = {1, 2, 3, 5, 7, 9, 10, 12, 15};

    int result = poslednjiKojiNeZadovoljava(arr);
    if (result != -1) {
        cout << "Poslednji element koji ne zadovoljava je: " << arr[result] << " na indeksu "
<< result << endl;
    } else {
        cout << "Svi elementi zadovoljavaju uslov." << endl;
    }

    return 0;
}
```

Izlaz:

Poslednji element koji ne zadovoljava je: 9 na indeksu 5

Pronalaženje prvog elementa koji zadovoljava uslov

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

// Uslov za proveru
bool zadovoljava(int x) {
    return x >= 10; // Primer: zadovoljava ako je >= 10
}

int prviKojiZadovoljava(const vector<int>& arr) {
    int low = 0, high = arr.size() - 1;
    int result = -1; // Indeks prvog koji zadovoljava

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (zadovoljava(arr[mid])) {
            result = mid; // Zabeleži indeks
            high = mid - 1; // Idi levo
        } else {
            low = mid + 1; // Idi desno
        }
    }

    return result;
}

int main() {
    vector<int> arr = {1, 2, 3, 5, 7, 9, 10, 12, 15};

    int result = prviKojiZadovoljava(arr);
    if (result != -1) {
        cout << "Prvi element koji zadovoljava je: " << arr[result] << " na indeksu " << result
<< endl;
    } else {
        cout << "Nijedan element ne zadovoljava uslov." << endl;
    }

    return 0;
}
```

Izlaz:

Prvi element koji zadovoljava je: 10 na indeksu 6

Analiza kompleksnosti

- **Vremenska složenost:** O(log n) zbog binarne pretrage.
- **Prostorna složenost:** O(1), jer koristimo konstantan prostor.

Primena u praksi

1. **Određivanje opsega:** Koristi se u problemima gde treba pronaći početak ili kraj intervala zadovoljenja uslova.
2. **Pragovi:** Identifikacija prvog elementa iznad ili ispod praga.
3. **Optimizacija:** Koristi se u optimizaciji gde treba pronaći najmanju vrednost parametra koja zadovoljava zadati kriterijum.

Pohlepni (gramzivi) algoritmi u C++

Pohlepni algoritmi (greedy algorithms) su klasa algoritama koji donose sekvencijalne odluke tako što na svakom koraku biraju **najbolju lokalnu opciju** u nadi da će rezultirati **optimalnim globalnim rešenjem**.

Karakteristike pohlepnih algoritama

1. **Lokalna optimalnost:** Na svakom koraku bira se trenutna najbolja odluka (najpovoljniji element).
2. **Irreverzibilnost:** Jednom donesena odluka se ne preispituje.
3. **Efikasnost:** Obično imaju vremensku složenost O(n log n) (ako je potrebno sortiranje) ili O(n).

Kada koristiti pohlepni algoritam?

1. Problem ima **svojstvo pohlepnosti** (greedy-choice property), tj. lokalna optimalnost vodi ka globalnoj optimalnosti.
2. Problem ima **svojstvo optimalne podstrukture**, tj. optimalno rešenje problema može se konstruisati iz optimalnih rešenja njegovih podproblema.

Ako problem ne zadovoljava ova svojstva, pohlepni algoritmi možda neće dati tačno rešenje.

Primer problema i implementacija

Aktivnosti (Activity Selection Problem)

Izabrati najveći broj aktivnosti koje se ne preklapaju, gde je svaka aktivnost definisana početkom i krajem.

Opis:

- Sortiramo aktivnosti prema vremenu završetka.
- Biramo prvu aktivnost koja se ne preklapa s prethodno odabranom.

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

// Struktura za aktivnost
struct Aktivnost {
    int pocetak, kraj;
};

// Funkcija za poređenje po završetku aktivnosti
bool compare(const Aktivnost& a1, const Aktivnost& a2) {
    return a1.kraj < a2.kraj;
}

int main() {
    vector<Aktivnost> aktivnosti = {{1, 3}, {2, 5}, {4, 6}, {6, 8}, {5, 9}, {8, 10}};

    // Sortiranje aktivnosti prema vremenu završetka
    sort(aktivnosti.begin(), aktivnosti.end(), compare);

    // Izbor aktivnosti
    int brojIzabranih = 1;
    int poslednjiKraj = aktivnosti[0].kraj;

    cout << "Izabrane aktivnosti: " << endl;
    cout << "(" << aktivnosti[0].pocetak << ", " << aktivnosti[0].kraj << ")" << endl;

    for (int i = 1; i < aktivnosti.size(); i++) {
        if (aktivnosti[i].pocetak >= poslednjiKraj) {
            cout << "(" << aktivnosti[i].pocetak << ", " << aktivnosti[i].kraj << ")" << endl;
            poslednjiKraj = aktivnosti[i].kraj;
            brojIzabranih++;
        }
    }

    cout << "Ukupno izabranih aktivnosti: " << brojIzabranih << endl;
    return 0;
}
```

Izlaz:

Izabrane aktivnosti:

(1, 3)

(4, 6)

(6, 8)

(8, 10)

Ukupno izabranih aktivnosti: 4

Prednosti i mane pohlepnih algoritama

Prednosti	**Mane**
Jednostavna implementacija	Ne garantuje optimalno rešenje za sve probleme
Efikasan za određene probleme	Zahteva svojstvo pohlepnosti i optimalne podstrukture
Brz u odnosu na dinamičko programiranje	Nije uvek fleksibilan za promene u problemu

Provera da li je broj prost i faktorizacija broja sa složenošću

1. Provera da li je broj prost

Broj je **prost** ako ima tačno dva delitelja: 1 i sebe. Da bismo proverili da li je broj prost, možemo koristiti sledeću optimizaciju:

1. Proveriti deljivost broja sa 2 i 3.
2. Iterirati kroz potencijalne delioce od 5 do sqrt{n}, povećavajući za 6 (i=i+6), jer delitelji prostih brojeva mogu biti oblika 6k±1.

Implementacija za proveru prostosti

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

bool jeProst(int n) {
    if (n <= 1) return false; // Brojevi <= 1 nisu prosti
    if (n <= 3) return true;  // 2 i 3 su prosti brojevi
    if (n % 2 == 0 || n % 3 == 0) return false; // Deljivost sa 2 ili 3

    // Provera deljivosti od 5 do sqrt(n) sa korakom 6
    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) {
            return false;
        }
    }
    return true;
}

int main() {
    int broj;
    cout << "Unesite broj: ";
    cin >> broj;

    if (jeProst(broj)) {
        cout << broj << " je prost broj." << endl;
    } else {
        cout << broj << " nije prost broj." << endl;
    }

    return 0;
}
```

Primer izlaza:

Unesite broj: 29

29 je prost broj.

2. Faktorizacija broja

Faktorizacija broja razlaže ga na njegove proste činioce. Efikasna implementacija radi na sledeći način:

1. Ukloniti sve faktore 2.
2. Iterirati kroz potencijalne faktore od 3 do sqrt{n}, povećavajući za 2 (neparni brojevi).
3. Na kraju, ako preostali broj n>2, onda je on prost faktor.

Implementacija faktorizacije

```cpp
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

vector<int> faktorizacija(int n) {
    vector<int> faktori;

    // Ukloni sve faktore 2
    while (n % 2 == 0) {
        faktori.push_back(2);
        n /= 2;
    }

    // Proveri sve neparne faktore do sqrt(n)
    for (int i = 3; i * i <= n; i += 2) {
        while (n % i == 0) {
            faktori.push_back(i);
            n /= i;
        }
    }

    // Ako je preostali broj > 2, onda je prost faktor
    if (n > 2) {
        faktori.push_back(n);
    }

    return faktori;
}

int main() {
    int broj;
    cout << "Unesite broj: ";
    cin >> broj;

    vector<int> faktori = faktorizacija(broj);
    cout << "Faktori broja " << broj << " su: ";
    for (int faktor : faktori) {
        cout << faktor << " ";
    }
    cout << endl;

    return 0;
}
```

Primer izlaza:

Unesite broj: 60

Faktori broja 60 su: 2 2 3 5

Analiza složenosti

- **Provera prostosti:**
  - Broj iteracija je proporcionalan sqrt{n}, pa je vremenska složenost O(sqrt{n}).
- **Faktorizacija:**
  - Takođe ima vremensku složenost O(sqrt{n}) jer svaki prost faktor p eliminišemo sa n podelama dok n ne postane manji od sqrt{n}.

Praktične primene

1. **Provera prostih brojeva:** Koristi se u kriptografiji (RSA algoritam), teoriji brojeva i generisanju prostih brojeva.
2. **Faktorizacija:** Korisna za analizu brojeva, traženje najvećeg zajedničkog delioca i rešavanje zadataka baziranih na deljivosti.

Euklidov algoritam za NZD i NZS

Euklidov algoritam je efikasan postupak za pronalaženje **najvećeg zajedničkog delioca (NZD)** dva broja. On se zasniva na svojstvu da se najveći zajednički delilac dva broja ne menja ako veći broj zamenimo ostatkom pri deljenju manjeg broja.

1. Pronalaženje NZD

Osnovna ideja algoritma:

1. Ako je $a = 0$, onda je $NZD(a, b) = b$.
2. Ako je $b = 0$, onda je $NZD(a, b) = a$.
3. U suprotnom, $NZD(a, b) = NZD(b, a\%b)$.

Rekurzivna implementacija za NZD

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

// Rekurzivna funkcija za pronalaženje NZD
int NZD(int a, int b) {
    if (b == 0) {
        return a;
    }
    return NZD(b, a % b);
}

int main() {
    int a, b;
    cout << "Unesite dva broja: ";
    cin >> a >> b;

    cout << "Najveći zajednički delilac (NZD) brojeva " << a << " i " << b << " je: " << NZD(a,
b) << endl;
    return 0;
}
```

Primer izlaza:

Unesite dva broja: 48 18

Najveći zajednički delilac (NZD) brojeva 48 i 18 je: 6

2. Pronalaženje NZS

Najmanji zajednički sadržilac (NZS) može se izračunati koristeći NZD:

$$NZS(a, b) = \frac{|a \cdot b|}{NZD(a, b)}$$

Funkcija za NZS koristeći NZD

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

// Funkcija za NZD
int NZD(int a, int b) {
    if (b == 0) {
        return a;
    }
    return NZD(b, a % b);
}

// Funkcija za NZS
int NZS(int a, int b) {
    return (a / NZD(a, b)) * b;
}

int main() {
    int a, b;
    cout << "Unesite dva broja: ";
    cin >> a >> b;

    cout << "Najmanji zajednički sadržilac (NZS) brojeva " << a << " i " << b << " je: " <<
NZS(a, b) << endl;
    return 0;
}
```

Primer izlaza:

Unesite dva broja: 48 18

Najmanji zajednički sadržilac (NZS) brojeva 48 i 18 je: 144

Iterativna implementacija NZD

Ako ne želite koristiti rekurziju, algoritam se može implementirati iterativno.

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

// Iterativna funkcija za pronalaženje NZD
int NZD_iterativno(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main() {
    int a, b;
    cout << "Unesite dva broja: ";
    cin >> a >> b;
```

```
    cout << "Najveći zajednički delilac (NZD) brojeva " << a << " i " << b << " je: " <<
NZD_iterativno(a, b) << endl;
    return 0;
}
```

Analiza složenosti

- **Vremenska složenost:** O(log (min (a,b))), jer se veličina brojeva smanjuje približno za polovinu na svakom koraku.
- **Prostorna složenost:** O(1) za iterativnu implementaciju, O(log (min (a,b))) za rekurzivnu (zbog steka funkcija).

Praktične primene

1. **Razlomci:** Svodjenje razlomaka na najjednostavniji oblik.
2. **Kriptografija:** Koristi se u algoritmima kao što su RSA za pronalaženje relativno prostih brojeva.
3. **Matematičke operacije:** Potreban za mnoge zadatke vezane za deljivost i faktorizaciju.

Eratostenovo sito za generisanje prostih brojeva u C++

Eratostenovo sito je efikasan algoritam za pronalaženje svih prostih brojeva do nekog broja n. Ovaj algoritam koristi princip eliminacije višekratnika brojeva koji nisu prosti, počevši od najmanjeg prostog broja 2.

Koraci algoritma

1. Napraviti niz logičkih vrednosti (ili celobrojni niz) od 0 do n, gde svaki element označava da li je broj potencijalno prost.
2. Inicijalizovati sve brojeve kao potencijalno proste (osim 0 i 1, koji nisu prosti).
3. Počevši od 2 (prvi prost broj), eliminisati sve njegove višekratnike iz niza označavanjem da nisu prosti.
4. Ponoviti postupak za sledeći broj u nizu koji je označen kao prost.
5. Brojevi koji ostanu označeni kao prosti nakon eliminacije su prosti brojevi.

Implementacija u C++

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

// Funkcija koja implementira Eratostenovo sito
vector<int> eratostenovoSito(int n) {
    vector<bool> prost(n + 1, true); // Niz koji označava da su svi brojevi potencijalno prosti
    prost[0] = prost[1] = false;    // 0 i 1 nisu prosti brojevi

    for (int i = 2; i * i <= n; i++) {
        if (prost[i]) {
            // Eliminacija svih višekratnika broja i
            for (int j = i * i; j <= n; j += i) {
                prost[j] = false;
            }
        }
    }

    // Prikupljanje prostih brojeva
    vector<int> prostiBrojevi;
    for (int i = 2; i <= n; i++) {
        if (prost[i]) {
            prostiBrojevi.push_back(i);
        }
    }
    return prostiBrojevi;
}

int main() {
    int n;
    cout << "Unesite gornju granicu (n): ";
    cin >> n;

    vector<int> prostiBrojevi = eratostenovoSito(n);

    cout << "Prosti brojevi do " << n << " su: ";
    for (int broj : prostiBrojevi) {
        cout << broj << " ";
    }
    cout << endl;

    return 0;
}
```

Primer izlaza

Unesite gornju granicu (n): 30

Prosti brojevi do 30 su: 2 3 5 7 11 13 17 19 23 29

Analiza složenosti

1. **Vremenska složenost:**
    - Petlja za iteraciju kroz brojeve od 2 do sqrt{n} ima složenost O(sqrt{n}).
    - Eliminacija višekratnika unutar svake iteracije ima ukupnu složenost O(n log (log (n))).
    - Ukupna vremenska složenost: O(n log (log (n))).
2. **Prostorna složenost:**
    - Memorija za niz veličine n + 1, što je O(n).

Optimizacije

1. **Eliminacija parnih brojeva:**
    - Možemo izbeći označavanje parnih brojeva tako što ćemo raditi samo sa neparnim brojevima.
2. **Početak od i·* i:**
    - Eliminacija višekratnika broja i može početi od i * i jer su svi manji višekratnici već eliminisani.

Praktične primene

1. Generisanje prostih brojeva za kriptografiju.
2. Matematičke analize velikih skupova brojeva.
3. Efikasno rešavanje problema na takmičenjima, poput "najteži prost broj ispod n".

Predstavljanje osnovnih geometrijskih objekata u C++ programskom jeziku uključuje definisanje struktura ili klasa za tačke, duži, prave i kružnice, zajedno s funkcijama koje omogućavaju operacije nad tim objektima.

1. Tačka

Tačka u koordinatnoj ravni definiše se pomoću dve koordinate x i y.

Implementacija:

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

// Struktura za tačku
struct Tacka {
    double x, y;

    // Konstruktor
    Tacka(double _x = 0, double _y = 0) : x(_x), y(_y) {}

    // Funkcija za izračunavanje udaljenosti između dve tačke
    double udaljenost(const Tacka& druga) const {
        return sqrt(pow(x - druga.x, 2) + pow(y - druga.y, 2));
    }
};
```

Primer korišćenja:

```
Tacka t1(1, 2), t2(4, 6);
cout << "Udaljenost između tačaka: " << t1.udaljenost(t2) << endl;
```

2. Duž

Duž je definisana s dve tačke koje predstavljaju njene krajeve.

Implementacija:

```
struct Duz {
    Tacka pocetak, kraj;

    // Konstruktor
    Duz(Tacka _pocetak, Tacka _kraj) : pocetak(_pocetak), kraj(_kraj) {}

    // Funkcija za izračunavanje dužine duži
    double duzina() const {
        return pocetak.udaljenost(kraj);
    }
};
```

Primer korišćenja:

```
Tacka t1(0, 0), t2(3, 4);
Duz duz(t1, t2);
cout << "Dužina duži: " << duz.duzina() << endl;
```

3. Prava

Prava u ravni može se predstaviti u obliku Ax+By+C=0 ili pomoću tačke i vektora koji definišu pravac.

Implementacija:

```
struct Prava {
    double A, B, C; // Koeficijenti pravca Ax + By + C = 0

    // Konstruktor
    Prava(double _A, double _B, double _C) : A(_A), B(_B), C(_C) {}

    // Funkcija za proveru da li tačka pripada pravoj
    bool pripadaPravi(const Tacka& t) const {
        return (A * t.x + B * t.y + C == 0);
    }
};
```

Primer korišćenja:

```
Prava p(1, -1, -1); // Prava: x - y - 1 = 0
Tacka t(1, 2);
cout << "Tačka pripada pravoj: " << (p.pripadaPravi(t) ? "Da" : "Ne") << endl;
```

4. Kružnica

Kružnica se definiše centrom (tačkom) i radijusom.

Implementacija:

```
struct Kruznica {
    Tacka centar;
    double radijus;

    // Konstruktor
    Kruznica(Tacka _centar, double _radijus) : centar(_centar), radijus(_radijus) {}

    // Funkcija za proveru da li tačka leži na kružnici
    bool pripadaKruznici(const Tacka& t) const {
        return abs(centar.udaljenost(t) - radijus) < 1e-9; // Uzimamo u obzir grešku u
računanju
    }
};
```

Primer korišćenja:

```
Tacka centar(0, 0);
Kruznica k(centar, 5);
Tacka t(3, 4);
cout << "Tačka pripada kružnici: " << (k.pripadaKruznici(t) ? "Da" : "Ne") << endl;
```

Proširenja

1. **Dodavanje metoda za međusobne relacije:**
    - Da li se dve duži seku.
    - Da li se prava i kružnica seku.
    - Da li se dve kružnice dodiruju ili seku.
2. **Efikasna provera pozicije tačke:**
    - Da li je tačka unutar kružnice.
    - Da li je tačka iznad ili ispod prave.
3. **Korišćenje C++ STL biblioteka:**
    - STL funkcije poput std::pair za efikasno predstavljanje tačaka.

Euklidska geometrija i osnovne operacije poput računanja euklidskog rastojanja i korišćenja Pitagorine teoreme su korisni alati u rešavanju problema vezanih za geometrijske objekte. U nastavku ću objasniti kako ih možete implementirati u C++ programskom jeziku.

1. Euklidsko rastojanje

Euklidsko rastojanje između dve tačke A(x1,y1) i B(x2,y2) u ravni izračunava se pomoću formule:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Implementacija u C++:

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

// Struktura za tačku
struct Tacka {
    double x, y;

    Tacka(double _x = 0, double _y = 0) : x(_x), y(_y) {}
};

// Funkcija za izračunavanje euklidskog rastojanja
double euklidskoRastojanje(const Tacka& A, const Tacka& B) {
    return sqrt(pow(B.x - A.x, 2) + pow(B.y - A.y, 2));
}

int main() {
    Tacka A(1, 2), B(4, 6);
    cout << "Euklidsko rastojanje između A i B: " << euklidskoRastojanje(A, B) << endl;
    return 0;
}
```

Primer izlaza:

Euklidsko rastojanje između A i B: 5

2. Pitagorina teorema

Pitagorina teorema kaže da je u pravouglom trouglu kvadrat dužine hipotenuze jednak sumi kvadrata dužina kateta:

$$c^2 = a^2 + b^2$$

Provera da li je trougao pravougli:

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

// Funkcija za proveru pravouglosti trougla
bool jePravougli(double a, double b, double c) {
    // Poređenje hipotenuze sa katetama uzimajući u obzir numeričke greške
    return abs(c * c - (a * a + b * b)) < 1e-9;
}

int main() {
    double a = 3, b = 4, c = 5; // Katete i hipotenuza pravouglog trougla
    if (jePravougli(a, b, c)) {
        cout << "Trougao je pravougli." << endl;
    } else {
        cout << "Trougao nije pravougli." << endl;
    }
    return 0;
}
```

Primer izlaza:

Trougao je pravougli.

3. Poluprečnik opisane kružnice pravouglog trougla

Poluprečnik opisane kružnice za pravougli trougao je polovina dužine hipotenuze:

$$R = \frac{c}{2}$$

Implementacija u C++:

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

double poluprecnikOpisaneKruznice(double c) {
    return c / 2;
}

int main() {
    double c = 5; // Hipotenuza pravouglog trougla
    cout << "Poluprečnik opisane kružnice: " << poluprecnikOpisaneKruznice(c) << endl;
    return 0;
}
```

Primer izlaza:

Poluprečnik opisane kružnice: 2.5

4. Površina trougla

Površina pravouglog trougla može se izračunati kao:

$$P = \frac{1}{2} \cdot a \cdot b$$

Implementacija u C++:

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

double povrsinaPravouglogTrougla(double a, double b) {
    return 0.5 * a * b;
}

int main() {
    double a = 3, b = 4; // Katete pravouglog trougla
    cout << "Površina pravouglog trougla: " << povrsinaPravouglogTrougla(a, b) << endl;
    return 0;
}
```

Primer izlaza:

Površina pravouglog trougla: 6

5. Opšta primena sa geometrijskim objektima

Kombinovanjem euklidskog rastojanja i Pitagorine teoreme možemo rešavati složenije probleme, kao što je određivanje da li tačke formiraju pravougli trougao:

Provera pravouglosti trougla preko rastojanja:

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

// Struktura za tačku
struct Tacka {
    double x, y;

    Tacka(double _x = 0, double _y = 0) : x(_x), y(_y) {}
};

// Funkcija za izračunavanje rastojanja između tačaka
double rastojanje(const Tacka& A, const Tacka& B) {
    return sqrt(pow(B.x - A.x, 2) + pow(B.y - A.y, 2));
}

// Funkcija za proveru pravouglosti trougla
bool jePravougliTrougao(const Tacka& A, const Tacka& B, const Tacka& C) {
    double d1 = rastojanje(A, B);
    double d2 = rastojanje(B, C);
    double d3 = rastojanje(A, C);

    // Provera Pitagorine teoreme za sve moguće kombinacije
    return abs(d1 * d1 - (d2 * d2 + d3 * d3)) < 1e-9 ||
           abs(d2 * d2 - (d1 * d1 + d3 * d3)) < 1e-9 ||
           abs(d3 * d3 - (d1 * d1 + d2 * d2)) < 1e-9;
}

int main() {
    Tacka A(0, 0), B(3, 0), C(0, 4); // Tačke pravouglog trougla
    cout << "Tačke čine pravougli trougao: " << (jePravougliTrougao(A, B, C) ? "Da" : "Ne") <<
endl;
    return 0;
}
```

Primer izlaza:

Tačke čine pravougli trougao: Da