

**SVEUČILIŠTE U ZAGREBU**  
**FAKULTET ORGANIZACIJE I INFORMATIKE**  
**V A R A Ž D I N**

**Matija Belec**

# **PRIMJENA REKURZIVNIH ALGORITAMA U PROGRAMIRANJU**

**ZAVRŠNI RAD**

**Varaždin, 2016.**

**SVEUČILIŠTE U ZAGREBU**  
**FAKULTET ORGANIZACIJE I INFORMATIKE**  
**V A R A Ž D I N**

**Matija Belec**

**Matični broj: 39912/11**

**Studij: Informacijski sustavi**

**PRIMJENA REKURZIVNIH ALGORITAMA U**  
**PROGRAMIRANJU**

**ZAVRŠNI RAD**

**Mentor:**

Prof.dr.sc. Danijel Radošević

**Varaždin, rujan 2016.**

# Sadržaj

1. Uvod .....	1
2. Matematička indukcija .....	2
3. Rekurzija.....	3
3.1. Rekurzija u programskom jeziku C++ .....	3
3.2. Programski stog .....	3
4. „Školski“ primjeri rekurzija.....	5
4.1. Faktorijel.....	5
4.2. Fibonaccijev niz.....	6
5. Algoritmi podijeli pa vladaj.....	8
5.1. Algoritmi sortiranja .....	8
5.1.1. Quick sort .....	8
5.1.2. Merge sort.....	10
5.2. Algoritmi pretraživanja .....	12
5.2.1. Binarno pretraživanje .....	12
6. Pohlepni algoritmi .....	15
6.1. Vraćanje ostatka novca.....	15
7. Metoda pretraživanja s vraćanjem.....	17
7.1. Problem N-kraljica .....	17
8. Binarno stablo.....	20
8.1. Rekurzivna funkcija za dodavanje vrijednosti u stablo .....	20
8.2. Ispis binarnog stabla .....	21
8.2.1. Metoda preorder .....	21
8.2.2. Metoda postorder.....	22
8.2.3. Metoda inorder .....	22
8.3. Pretraživanje binarnog stabla .....	22
8.4. Dealokacija stabla (rekurzivno brisanje elemenata stabla) .....	23
9. Nedostaci rekurzivnog pristupa .....	24
10. Zaključak .....	25
11. Literatura .....	26
12. Prilozi .....	27
12.1. Programska rješenja korištena u ovom radu .....	27
12.1.1. Faktorijel – rekurzivni algoritam (faktorijel.cpp).....	27
12.1.2. Faktorijel – iterativni algoritam (faktorijel_iter.cpp) .....	27
12.1.3. Fibonaccijev niz – rekurzivni algoritam (fibo.cpp) .....	27
12.1.4. Fibonaccijev niz – iterativni algoritam (fibo_iter.cpp).....	28

12.1.5.	Quick sort (qs.cpp) .....	28
12.1.6.	Merge sort (ms.cpp).....	29
12.1.7.	Binarno pretraživanje – rekurzivni algoritam (bs.cpp).....	30
12.1.8.	Binarno pretraživanje – iterativni algoritam (bs_iter.cpp) .....	31
12.1.9.	Vraćanje novca (vracanje_novca.cpp).....	32
12.1.10.	Problem N-kraljica .....	33
12.1.11.	Binarno stablo (bs.cpp).....	35

# 1. Uvod

U ovom ću radu navesti nekoliko primjera rekurzivnih algoritama koje ću detaljno razraditi. Algoritmi su podijeljeni u nekoliko poglavlja ovisno o njihovoj namjeni. Za početak nešto ću reći o matematičkoj indukciji koja je temelj rekurzivnih algoritama.

Zatim ću navesti nekoliko jednostavniji rekurzivnih algoritama. Radi se o algoritmima koji se često stavljaju kao osnovni primjeri rekurzije u školstvu.

Zatim slijede algoritmi za pretraživanje i sortiranje koji koriste metodu „podijeli pa vladaj“. Oni su po svojoj naravi malo složeniji od klasičnih „školskih“ algoritama sa početka rada.

Na kraju dolaze i rekurzivni algoritmi s vraćanjem (*engl. backtracking*) gdje će se razraditi složeniji rekurzivni algoritmi.

U ovom radu se koriste programska rješenja koja su preuzeta iz osobnog github profila, odnosno repozitorija *cpp-algorithms*<sup>1</sup>.

---

<sup>1</sup> Navedeni repozitorij je dostupan na <https://github.com/matijabelec/cpp-algorithms>

## 2. Matematička indukcija

Kod stvaranja programskog rješenja i samog algoritma rješenja veliku ulogu najčešće odigrava matematička indukcija. Matematička indukcija služi za pojednostavljenje problema koji se rješava na način da se dokazuje u nekoliko manjih koraka početna tvrdnja za  $n = 1$ , te kroz  $k = n$  i  $n = k + 1$  dokazujemo da tvrdnja vrijedi za svaki prirodni broj.

Tako dolazimo do saznanja da su koraci kod dokazivanja matematičkom indukcijom, kako je ukratko navedeno iznad, sljedeći:

1. Baza indukcije – radi se o dokazivanju da određena tvrdnja vrijedi za  $n = 1$ ,
2. Pretpostavka indukcije – radi se pretpostavka da tvrdnja vrijedi i za proizvoljno odabrani  $k$ , čime se ustvari dobiva da je  $k = n$ ,
3. Korak indukcije – ovdje se radi dokaz da tvrdnja vrijedi i za  $k + 1$ .

Posljednji korak indukcije se svodi na rješavanje problem uvrštavanjem da je  $n = k + 1$ . Tako dolazimo do zaključka kojim potvrđujemo da tvrdnja vrijedi za svaki prirodni broj.

### 3. Rekurzija

Definicija rekurzije se sastoji od dva dijela. Prvi dio je baza rekurzije, kojim se prekida rekurzija i predstavlja osnovni element, odnosno rješenje kojim se rješavaju svi ostali problemi kod rekurzivnog rješavanja problema. Drugi dio je sam korak rekurzije kojim se definira kako se ponavlja rješavanje određenog problema ponovnim kreiranjem novog rekurzivnog koraka sve do bazičnog rješenja, odnosno baze rekurzije (Drozdek, 2013).

Rekurzija predstavlja ponavljanje. Kod programskog algoritma rekurzija se prikazuje kao funkcija koja poziva samu sebe unutar svog tijela.

Rekurzivni algoritam se sastoji od 2 dijela:

1. Baze rekurzije, odnosno osnovnog uvijeta kada rekurzija prestaje,
2. Koraka rekurzije

Baza rekurzije je obavezna kod rekurzivnog algoritma jer mora postojati način da se rekurzivni poziv funkcije prekine kako bi algoritam mogao završiti.

#### 3.1. Rekurzija u programskom jeziku C++

U programskom jeziku c++ primjer rekurzivne funkcije možemo napisati na sljedeći način:

```
int funk (int n) {  
    if(n<1) return 1;  
    return 1 + funka(n-1);  
}
```

Unutar funkcije „funk“ se nalazi poziv iste. Kao baza rekurzije ovdje je postavljen uvijet  $n < 1$  što znači da će se funkcija rekurzivno pozivati sve dok se  $n$  ne umanjuje na vrijednost 0.

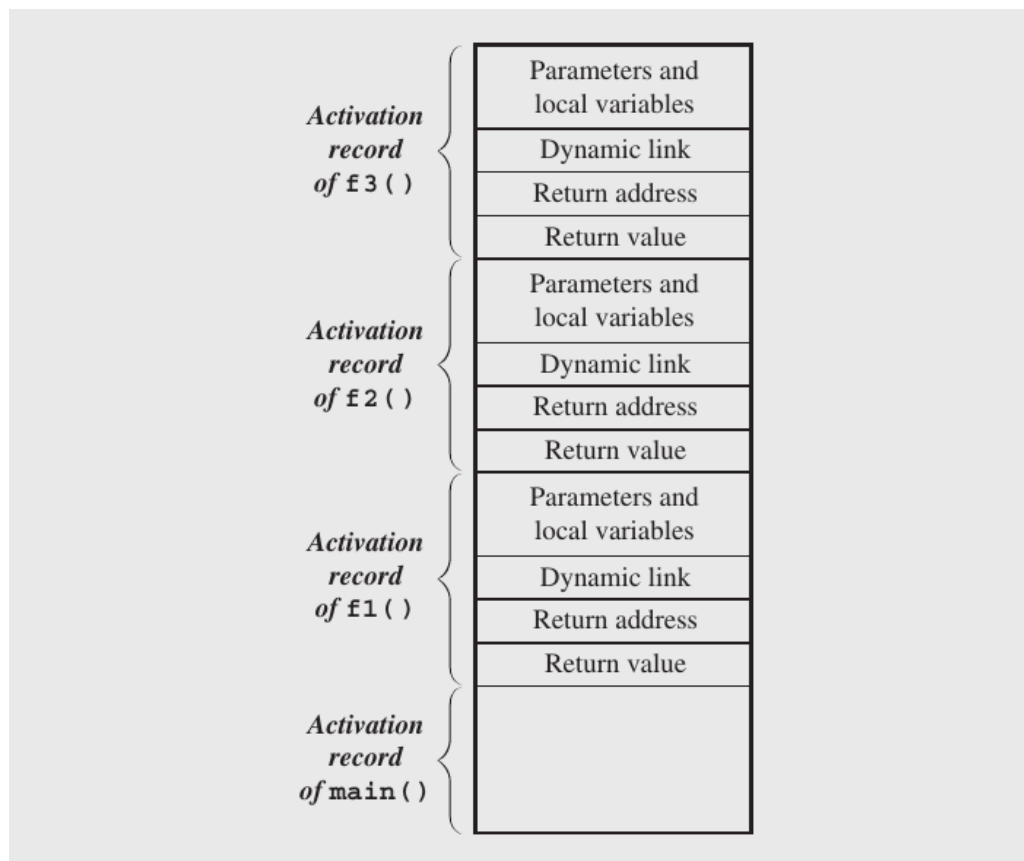
#### 3.2. Programski stog

Kod pisanja programskog rješenja za određeni problem koristeći rekurzivni algoritam važno je voditi računa i o programskom stogu. Kod rekurzivnih funkcija svakim novim rekurzivnim korakom, odnosno rekurzivnim pozivom funkcije, na programski stog se postavlja novi okvir koji zauzima memoriju.

Pošto je programski stog veoma ograničen na standardnim računalima, važno je pravilno odrediti za koje sve moguće ulazne argumente će se rekurzivna funkcija moći

koristiti, a za koje će se, ukoliko je potrebno, ipak morati napraviti iterativno rješenje problema.

Na sljedećoj slici prikazan je vizualni prikaz sadržaja stoga.



Slika 1 Programski stog (Drozdek, 2013)

Ovdje možemo primjetiti kako se stog puni. Svakim rekurzivnim pozivom funkcije na stog se dodaju varijable (argumenti i lokalne varijable funkcije) na stog što, ako se radi o objektima kao varijablama koji zauzimaju veći dio memorije, vrlo brzo može dovesti do zapunjenja programskog stoga i rušenja programa.

Zato je kod rekurzivnih funkcije vrlo važno voditi računa o dostupnoj veličini memorije programskog stoga koju program zauzima tokom izvršavanja algoritma.



## 4. „Školski“ primjeri rekurzija

U ovom poglavlju navesti će se nekoliko standardnih primjera rekurzije koji se uče u školama. Radi se o jednostavnijim algoritmima rekurzije koji služe za lakše razumijevanje kako rekurzivni algoritmi funkcioniraju.

### 4.1. Faktorijel

Prvi klasičan primjer rekurzije u ovom radu je izračun faktorijela broja. Faktorijel broja je matematička funkcija kojom se izračunava umnožak brojeva od 1 do odabranog broja  $n$ . Na primjer,  $!5$  iznosi 120:

$$!5 = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Primjer jednostavnog programskog koda koji koristi rekurzivni pristup je prikazan sljedećim programskim kodom:

```
// g++ faktorijel.cpp -o faktorijel.out
#include <iostream>
long long faktorijel(unsigned int n) {
    return (n>1 ? n * faktorijel(n-1) : 1);
}
int main(){
    int n;
    do std::cin >> n; while(n<1);
    std::cout << faktorijel(n) << std::endl;
    return 0;
}
```

Također, faktorijel je moguće programski izračunati i na iterativni način kako slijedi u donjem primjeru programskog koda.

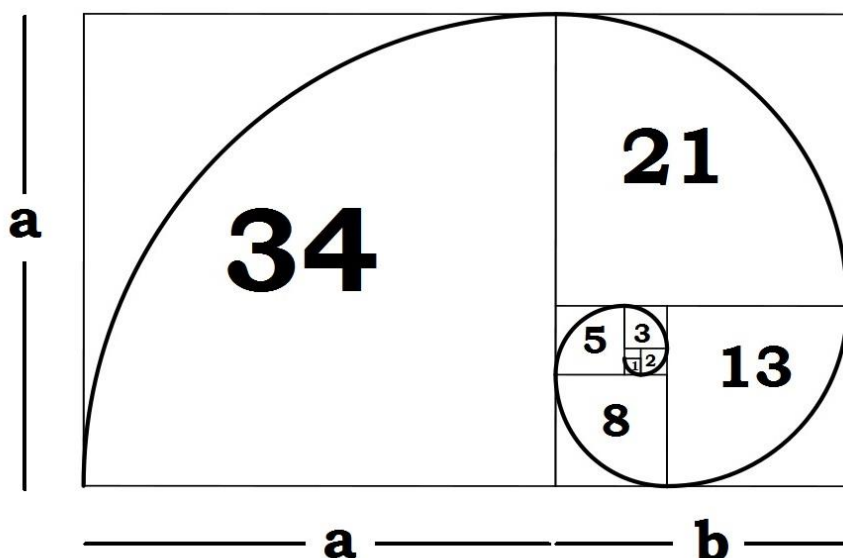
```
// g++ faktorijel_iter.cpp -o faktorijel_iter.out
#include <iostream>
int main(){
    int n;
    do std::cin >> n; while(n<1);

    long long f = 1;
    for(int i=2; i<=n; i++) {
        f *= i;
    }
    std::cout << f << std::endl;
    return 0;
}
```

## 4.2. Fibonaccijev niz

Drugi jednostavniji primjer rekurzije jest fibonaccijev niz.

Sam fibonaccijev niz spominje se od 1202. i vezan je uz osobu Leonardo Pisano (Leonardo of Pisa), poznatiji pod nadimkom Fibonacci. Europski matematičar Fibonacci autor knjige Liber Abbaci u kojoj je postavio zadatak: „Koliko zečeva se može umnožiti od samo jednog para zečeva u godini?“ tokom sljedećih godina dao je mnoge članke vezane uz navedeni niz u aritmetici i geometriji (Knuth, 1969).



Slika 2 Fibonaccijev niz (zlatni rez)<sup>2</sup>

Fibonaccijev niz određen je sljedećom matematičkom formulom:

$$F(n) = \begin{cases} 0, & \text{za } n = 0 \\ 1, & \text{za } n = 1 \\ F(n-1) + F(n-2), & \text{za } n > 1 \end{cases}$$

Prema navedenoj formuli možemo jednostavno izraditi programski kod koji slijedi u nastavku.

```
long long fibo(unsigned int n) {  
    return (n > 1 ? fibo(n-1) + fibo(n-2) : n);  
}
```

Razlog zašto je rekurzivni algoritam za izračun fibonaccijevog broja spor je upravo način na koji se rekruzija radi. Za izračun  $F_n$  postoje dva rekurzivna poziva za izračun  $F_{n-1}$  i  $F_{n-2}$ . Oba navedena rekurzivna poziva dalje rade nova dva rekurzivna poziva za izračun  $F_{n-2}$

<sup>2</sup> Preuzeto sa <https://fecdn-fractalenlighten.netdna-ssl.com/wp-content/uploads/2013/11/golden-ratio-fibonacci-sequence.jpg>

i  $F_{n-3}$ , odnosno  $F_{n-3}$  i  $F_{n-4}$ . Ovime se događa da se ustvari ponavljaju izračuni pojedinih problema pa se tako detaljnijom analizom algoritma dolazi do toga da se  $F_{n-3}$  računa tri puta,  $F_{n-4}$  se računa pet puta dok se  $F_{n-5}$  računa osam puta i tako dalje. Čime dolazimo do redundancije kod izračuna i razloga zašto je ovakav način izračuna spor (Weiss, 1992).

Pošto smo ustanovili da je rekurzivni pristup kod izračuna N-tog člana fibonaccijevog niza spor, slijedi iterativni način izračuna. Kod sljedećeg izračuna koristit ćemo samo petlju i tri varijable. Dvije varijable pamtit će  $F_{n-1}$ , te  $F_{n-2}$ , dok će treća sadržavati  $F_n$ . Upravo ta treća varijabla će nam nakon završetka izračuna koristiti za prikaz traženog člana fibonaccijevog niza.

```
// fibonacci iterativno
int rj[3] = {0, 1, 1};
if(n == 0) {
    rj[2] = 0;
} else {
    for(int i=1; i<n; i++) {
        rj[2] = rj[0] + rj[1];
        rj[0] = rj[1];
        rj[1] = rj[2];
    }
}
```

Iznad je naveden primjer koda (cijeli kod se nalazi u prilogima za fibonacci iterativno). Izračun fibonaccijevog niza iterativnim putem koristeći gore navedeni algoritam uvelike umanjuje poziv funkcija koji se kod rekurzivnog pristupa radi (ovdje nema poziva funkcije uopće), te se također koristi mnogo manje memorije: polje od tri elementa i dodatne varijable za prolazak kroz polje, te indeks traženog elementa. Za razliku od rekurzivnog pristupa gdje se kod svakog poziva funkcije također alociraju nove varijable.

## 5. Algoritmi podijeli pa vladaj

Metoda „podijeli pa vladaj“ (u nekim literaturama se spominje naziv „podijeli i ovladaj“) radi upravo to što se i iz samog naziva može isčitati: složeniji problem dijeli na više manjih problema od kojih se nadalje svaki zasebno rješava. Dobivanjem rješenja manjih problema, nadalje se dolazi do rješenja složenijeg (prvoizadanog) problema.

### 5.1. Algoritmi sortiranja

U programskim algoritmima često se dolazi do upotrebe određene vrste pretraživanja ili sortiranja. Kod sortiranja polja se radi pretraživanje polja ili dijela polja unutar kojeg tražimo elemente koji su veći ili manji od prethodno određenog elementa sa kojim ih uspoređujemo sa ciljem da se nakon procesa sortiranja, u spomenutom polju elemenata nalaze isti elementi ali sortirani redom od najmanjeg do najvećeg (ili obrnuto). Postoji mnogo vrsta sortiranja među kojima su više poznatiji:

- Sortiranje izborom,
- Sortiranje umetanjem,
- Sortiranje zamjenom,
- Quick sort,
- Sortiranje spajanjem (Merge sort).

Dok su prvi spomenuti oblici sortiranja koji se uglavnom implementiraju iterativnim pristupom, zadnja dva spomenuta, quick sort i sortiranje spajanjem, su dva najpoznatija algoritma sortiranja koja koriste metodu „podijeli pa vladaj“.

U nastavku se nalaze algoritamske reprezentacije i programska rješenja za oba navedena algoritma za sortiranje koji koriste navedenu metodu „podijeli pa vladaj“.

#### 5.1.1. Quick sort

Metoda sortiranja quick sort je izumljena 1962. od strane autora C. A. R. Hoare koji je dao odličan primjer u analizi algoritama. Navedena metoda sortiranja je u općoj upotrebi kod većine programskih jezika pa je sama analiza metode značajna (Sedgewick i Flajolet, 2013).

Kao što je navedeno, algoritam sortiranja quick sort je vrlo popularan u većini programskih jezika kao standardni način sortiranja elemenata unutar određenog polja, odnosno liste. Uglavnom se radi o jednoj od „poboljšanih“ varijanta quick sort-a gdje se polje,

umjesto na standardnih 2 dijela, dijeli na 3 dijela. U ovom radu ću se ograničiti na klasičan algoritam quick sort sortiranja i koristit ću samostalno napisan programski kod u programskom jeziku C/C++ kao primjer programskog rješenja navedenog sortiranja.

Za početak, ukratko o samom quick sort sortiranju. Navedeno sortiranje koristi sljedeće opisani algoritam:

1. Odredi „pivot“ element,
2. Podijeli polje na dva dijela tako da se u prvi (lijevi) dio polja prebace svi elementi manji od vrijednosti „pivot“ elementa, dok se preostali elementi postave u drugi (desni) dio polja,
3. Ponovi korak 2 za lijevi dio polja, te zatim za desni dio polja sve dok svaki dio polja ima elemenata.

Nakon što se korak 2 ponovi nad svakim dijelom polja rekursivno, te se u dijelu polja nađe samo jedan element, taj element je već sortiran i nalazi se na točnom indeksu na kojem se treba nalaziti u sortiranom polju. Time se nakon rekursivnog prebacivanja manjih elemenata u lijevi dio polja, a preostalih u desni dio polja dobiva sortiranje „pivot“ elementa u svakom koraku.

U nastavku se nalazi primjer programskog rješenja.

```
void qs(int* p, int a, int b) {
    int i=a, j=b;
    int m=p[(i+j)/2], s;

    while(i<=j) {
        while(p[i] < m) i++;
        while(p[j] > m) j--;
        if(i<=j) {
            s=p[i]; p[i]=p[j]; p[j]=s;
            i++; j--;
        }
    }

    if(a<j) qs(p, a, j);
    if(i<b) qs(p, i, b);
}

//qs(p, 0, n-1)
```

Na samom kraju programskog rješenja vidljiv je i sam poziv navedene funkcije qs. Prvi argument funkcije je polje (točnije pokazivač na prvi element polja), drugi argument je

indeks prvog elementa polja (ili dijela polja koji se želi sortirati), te posljednji, treći argument je indeks zadnjeg elementa polja (ili dijela polja koji se želi sortirati).

Pošto se kod sortiranja polja kao referentna brzina algoritma uzima najgori slučaj, a u ovom slučaju to je obrnuto sortirano polje (silazno sortirani elementi). Važno je primjetiti da, navedeno programsko rješenje, uzima središnji element dijela polja kao „pivot“ element (središnji prema indeksu). Naime, postoji nekoliko različitih implementacija algoritma gdje se na različiti način odabire „pivot“ element. Neki od načina su sljedeći:

- Odaberi N-ti element polja (najčešće se odabire središnji element dijela polja),
- Odaberi slučajnim odabirom element polja (u pravilu ovaj način daje najviše šanse da se dođe do sortirano polja na najbrži način u slučaju testiranja najgorog slučaja),
- Odabire element najbliže srednjoj vrijednosti svih elemenata u dijelu polja.

Ovdje se lako može primjetiti da se slučajnim načinom odabira elemenata u slučaju obrnuto sortirano polja može dogoditi najbolja situacija da se u svakom koraku odabire točno središnji element pa se kod dijeljenja polja na dva dijela uvijek dobiju podjednaki elementi.

Za razliku od pseudoslučajnog odabira elementa polja, u slučaju obrnuto sortirano polja, korištenjem navedenog primjera programskog rješenja dobiva se situacija da se polje dijeli na dva dijela od kojih jedan ne sadrži niti jedan element dok drugi dio polja sadrži N-1 elemenata. U ovom slučaju N predstavlja broj elemenata polja u K-tom koraku rekurzivnog poziva funkcije. Ovaj slučaj kod quick sort metode sortiranja daje najsporiji rezultat kod računanja brzine algoritma jednak  $O(n^2)$ .

Prosječna brzina algoritma je  $O(n \log n)$ , dok se u najboljem slučaju (kada se polje dijeli na dva podjednaka dijela kod svakog odabira „pivot“ elementa) dobiva brzina  $O(n \log n)$ .

### 5.1.2. Merge sort

Drugi algoritam sortiranja koji koristi metodu „podijeli pa vladaj“ koji ću obraditi u ovom radu je sortiranje spajanjem (poznatiji pod engleskim nazivom „merge sort“).

Ovaj algoritam može se opisati pomoću dva dijela algoritma, prvi dio je dijeljenje polja na dva podjednaka dijela sve dok polje ima elemenata. Drugi dio algoritma je spajanje po dva dijela natrag u jedan dio polja na način da se elementi iz jednog i drugog dijela polja

uzimaju prema veličini. Takvim spajanjem polja se dobiva novo polje koje ima svoje elemente sortirane.

Ovdje se nalazi primjer programskog rješenja sortiranja spajanjem:

```
void ms(int* p, int a, int b) {
    if(a<b) {
        int k = (a+b)/2;
        ms(p, a, k);
        ms(p, k+1, b);
        int i=a, j=k+1, c=0;
        int *p2 = new int[b-a+1];
        while(i<=k && j<=b)
            p2[c++] = p[ (p[i]<=p[j] ? i++ : j++) ];
        if(i>k)
            while(j<=b) p2[c++] = p[j++];
        else
            while(i<=k) p2[c++] = p[i++];
        for(int m=a; m<=b; m++) p[m] = p2[m-a];
        delete[] p2;
    }
}

//ms(p, 0, n-1)
```

U programskom rješenju vidljivi su spomenuti dijelovi algoritma. Prvi dio (dijeljenje polja na 2 dijela je prikazan rekurzivnim pozivom funkcije):

```
// (...)

int k = (a+b)/2;
ms(p, a, k);
ms(p, k+1, b);

// (...)
```

Dok se samo spajanje dijelova polja radi u ostatku programskog rješenja. Vidljivo je da se uspoređuju prvi sljedeći elementi iz oba dijela polja te se uvijek uzima manji element od dva. Time se polje koje se dobiva spajanjem popuni uzlazno sortiranim vrijednostima elemenata.

Brzina algoritma je  $O(n \log n)$ . Ovdje treba također spomenuti da se zbog specifičnosti algoritma svakim rekurzivnim dijeljenjem polja alociraju nova, manja polja što dodatno zauzima memoriju.

## 5.2. Algoritmi pretraživanja

U algoritmima pretraživanja također se može naći i algoritama koji koriste metodu „podijeli pa ovladaj“. Iako se iterativni algoritmi mogu lako prebaciti u rekurzivne, postoji standardni algoritam pretraživanja naziva binarno pretraživanje koji se jednostavnije implementira rekurzivno nego iterativno (iako je iterativni pristup gledajući sa aspekta memorije svakako manje zahtjevan nego rekurzivni).

### 5.2.1. Binarno pretraživanje

Specifičnost binarnog pretraživanja leži u klasičnom stilu metode „podijeli pa vladaj“. Radi se o algoritmu pretraživanja kojim se pronalazi traženi element u polju elemenata na način da se u svakom koraku polje podijeli na dva podjednaka dijela, te se zatim uspoređuje središnji element (element između dva dijela polja) sa traženim elementom. Ukoliko je traženi element jednak središnjem elementu tada je element pronađen u polju. Ukoliko je vrijednost traženog elementa manja od vrijednosti središnjeg elementa daljnja potraga za elementom nastavlja se nad prvim (lijevim) dijelom polja na isti princip kao i u prvom slučaju: odredi se središnji element i polje se time podijeli na dva dijela pa se radi navedena usporedba ponovo. Ukoliko je vrijednost traženog elementa veća od vrijednosti središnjeg elementa tada se traženje na isti princip ponavlja na drugom (desnom) dijelu polja. Takvo se rekurzivno traženje ponavlja sve dok se ne pronađe element ili dijelu polja ponestane elemenata što sugerira na to da traženi element u polju ne postoji.

Binarno pretraživanje je brzi algoritam pretraživanja sortirano polja ključeva<sup>3</sup>. Traženje ključa u polju se svodi na usporedbu traženog ključa sa vrijednošću srednjeg elementa polja. Ukoliko je traženi ključ manji tada se isti nalazi u prvoj polovici polja, a ako nije tada se traži u drugoj polovici polja. Rekurzivnim ponavljanjem navedenog traženja ključa dolazi se do velike razlike u brzini algoritma jer se traženje ključa smanjuje na ukupno  $\log n$  usporedbi (Skiena, 2008).

Iz zaključenog se može vidjeti da je važno da je polje prije pretraživanja sortirano. Obično se radi o uzlaznom sortiranju, no algoritam je vrlo jednostavno primjeniti i na silaznom sortiranju: potrebno je jedino zamijeniti uvijet veći i manje kod provjere u koji dio polja se ulazi rekurzivno. Prema svemu gore navedenom, algoritam možemo opisati sljedećim koracima:

---

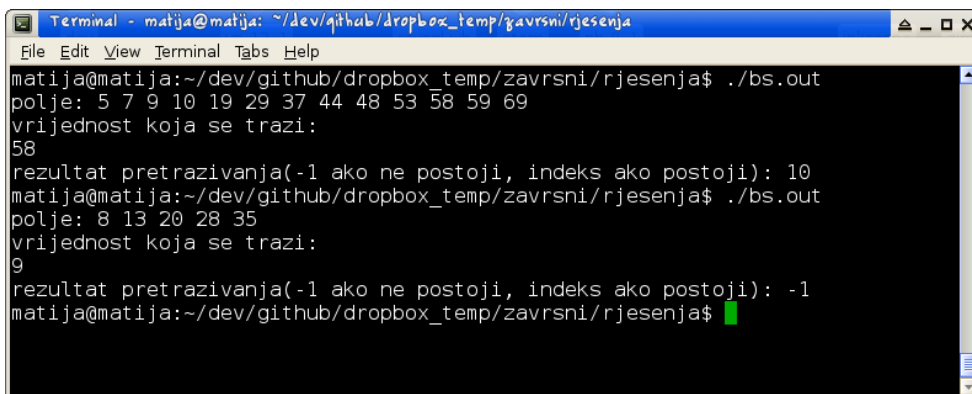
<sup>3</sup> Ključ u ovom kontekstu predstavlja samu vrijednost elementa u polju



1. Odredi središnji element polja, odnosno dijela polja u kojem tražimo traženi element.
2. Ako nema elemenata u polju, odnosno dijelu polja, vrati -1 (što predstavlja da traženi element, odnosno njegova vrijednost nije pronađena u polju). Završi pretraživanje.
3. Ako je vrijednost središnjeg elementa jednaka traženom elementu tada vrati indeks središnjeg elementa. I završi pretraživanje.
4. Ukoliko je traženi element polja manji od središnjeg ponovi sve od prvog koraka za prvi (lijevi) dio polja.
5. Ukoliko je traženi element polja veći od središnjeg ponovi sve od prvog koraka za drugi odnosno (desni) dio polja.

Postoji nekoliko varijanti algoritma ovisno o željenoj implementaciji: algoritam može vratiti vrijednost tipa bool, odnosno podatak da li se traženi element nalazi u polju ili ne, a može se implementirati i verzija u kojoj se vraća vrijednost -1 ako traženi element ne postoji u polju, odnosno indeks elementa u polju ako je isti pronađen. U nastavku se nalazi implementacija algoritma koja vraća indeks traženog elementa u polju ili -1 ukoliko traženi element ne postoji u polju.

```
int bs(int* polje, int x, int i, int j) {  
    if(i>j) return -1;  
    int m = (j-i)/2 + i;  
    if(x == polje[m]) return m;  
    if(x < polje[m]) return bs(polje, x, i, m-1);  
    return bs(polje, x, m+1, j);  
}
```



```
matija@matija:~/dev/github/dropbox_temp/zavrzni/rjesenja$ ./bs.out  
polje: 5 7 9 10 19 29 37 44 48 53 58 59 69  
vrijednost koja se trazi:  
58  
rezultat pretrazivanja(-1 ako ne postoji, indeks ako postoji): 10  
matija@matija:~/dev/github/dropbox_temp/zavrzni/rjesenja$ ./bs.out  
polje: 8 13 20 28 35  
vrijednost koja se trazi:  
9  
rezultat pretrazivanja(-1 ako ne postoji, indeks ako postoji): -1  
matija@matija:~/dev/github/dropbox_temp/zavrzni/rjesenja$
```

Slika 3 Primjer ispisa rješenja za binarno pretraživanje

Kod binarnog pretraživanja polja puno je bolja opcija korištenje iterativnog rješenja zbog navedenih problema sa programskim stogom u slučaju rekurzivnog primjera. Nadalje, zbog direktne promjene varijabli (bez poziva nove funkcije) algoritma je svakako efikasniji u odnosu na rekurzivni algoritam za navedeno pretraživanje. Programsko rješenje je također napravljeno u programskom jeziku c++, a cijeli kod se nalazi u prilogu na kraju ovog rada.

```
int bs(int* polje, int n, int x) {
    int i=0, j=n-1;
    int k = (i+j)/2;
    while(i<=j) {
        if(polje[k] == x) return k;
        if(polje[k] > x) j = k - 1;
        else i = k + 1;
        k = (i+j)/2;
    }
    return -1;
}
```

Algoritam se sastoji od petlje koja se ponavlja sve dok u dijelu polja u kojem tražimo vrijednost ima elemenata. Da li ima elemenata u polju provjeravamo pomoću krajnjih indeksa elemenata (varijabla i, te varijabla j) kojima određujemo dio polja za pretraživanje. U varijabli k spremamo indeks srednjeg elementa polja, te isti uspoređujemo sa traženom vrijednošću kako bi odredili da li smo pronašli traženu vrijednost u polju. Ukoliko algoritam izađe iz petlje, a da pritom nije vratio indeks nađenog elementa, tada zaključujemo da tražena vrijednost ne postoji u polju i vraćamo vrijednost -1.

## 6. Pohlepni algoritmi

Pohlepni algoritmi su algoritmi kod kojih se u svakom koraku algoritma traži optimalno rješenje za lokalni problem. Takvim određivanjem „najboljeg“ odabira kod svakog koraka pokušava doći do konačnog optimalnog rješenja kako se rekurzivni koraci ponavljaju (Cormen i sur., 2009).

U pripremljenim primjerima problema i rješenja pomoću pohlepnog algoritma prikazani su problemi korištenja navedenog tima algoritama.

### 6.1. Vraćanje ostatka novca

Kao jednostavniji primjer pohlepnog algoritma ovdje je naveden algoritam za vraćanje ostatka novca. Radi se o jednostavnom vraćanju određenog broja određenih vrste novčanica iz unaprijed poznatog skupa novčanica ovisno o ulaznom argumentu koji govori o ukupnom iznosu koji je potrebno vratiti.

```
const int n = 8;
int novcanice[n] = {100, 80, 50, 20, 10, 7, 5, 3};

int vrati_iznos(int iznos, int* rjesenje) {
    for(int i=0; i<n; i++) {
        if(iznos >= novcanice[i]) {
            rjesenje[i]++;
            return vrati_iznos(iznos-novcanice[i], rjesenje);
        }
    }
    return iznos;
}
```

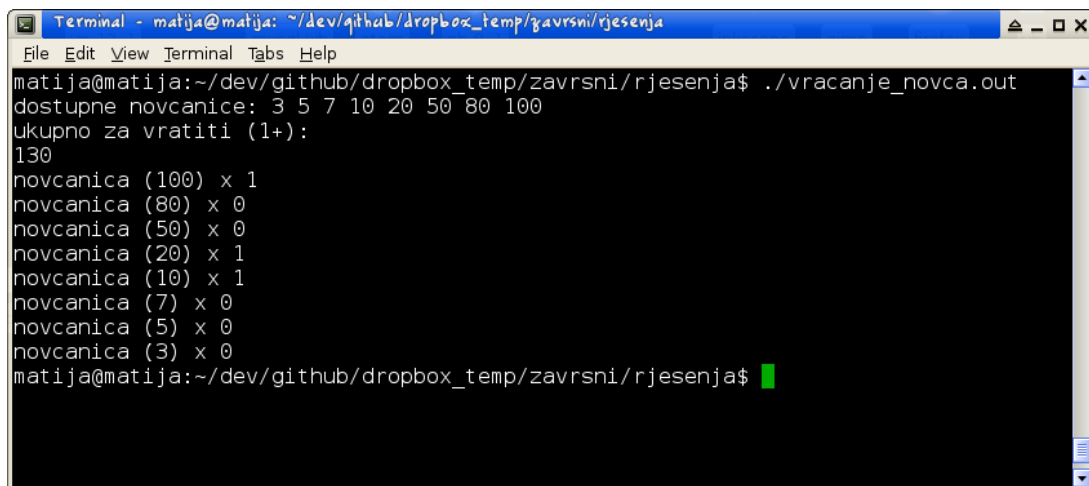
Ovdje je vidljiva rekurzivna funkcija koja rješava dani problem. Polje *novcanice* mora biti sortirano silazno zbog petlje unutar rekurzivne funkcije. Novčanice su namjerno postavljene na navedene vrijednosti kako bi se mogao prikazati nedostatak korištenja metode pohlepnog algoritma za navedeni problem vraćanja novca.

Ovdje je moguće vidjeti da ukoliko se unese iznos gdje je potrebno vratiti novčanice od 1 ili 2 tada algoritam vraća ostatak veći od nule jer nije moguće izvratiti određeni dio iznosa.

Drugi i veći problem je sama efikasnost algoritma. Naime, ovdje je lako uočiti da algoritam ne daje optimalno rješenje za svaku odabranu vrijednosti iznosa za vraćanje. Uzmimo na primjer vrijednost 130. Za vrijednost 130 algoritam će odraditi sljedeće:

1. U prvom pozivu funkcije algoritam odabire vrijednost 100 kao novčanicu za vratiti, te rekurzivno poziva funkciju sa preostalim iznosom  $130 - 100 = 30$ .
2. Odabire se najveća novčanica koja ne prelazi 30, a to je 20. Preostaje iznos od  $30 - 20 = 10$  te se rekurzivno poziva funkcija dalje.
3. Odabire se novčanica od 10 te se funkcija poziva sa iznosom 0.
4. Funkcija vraća 0 kao ostatak.

Prema navedenim koracima, vidimo da je rezultat koji algoritam vraća 1x100, 1x20 i 1x10. Primjer rezultata algoritma iz programskog koda (cijeli programski kod nalazi se u prilogima):



```
Terminal - matija@matija: ~/dev/github/dropbox_temp/zavrzni/rjesenja
matija@matija:~/dev/github/dropbox_temp/zavrzni/rjesenja$ ./vracanje_novca.out
dostupne novcanice: 3 5 7 10 20 50 80 100
ukupno za vratiti (1+):
130
novcanica (100) x 1
novcanica (80) x 0
novcanica (50) x 0
novcanica (20) x 1
novcanica (10) x 1
novcanica (7) x 0
novcanica (5) x 0
novcanica (3) x 0
matija@matija:~/dev/github/dropbox_temp/zavrzni/rjesenja$
```

Slika 4 Prikaz rješenja problema vraćanja novca za iznos=130

Ako pažljivo proučimo novčanice koje su zadane možemo vidjeti da je optimalno rješenje ustvari vraćanje dvije novčanice: 1x80 i 1x50.

## 7. Metoda pretraživanja s vraćanjem

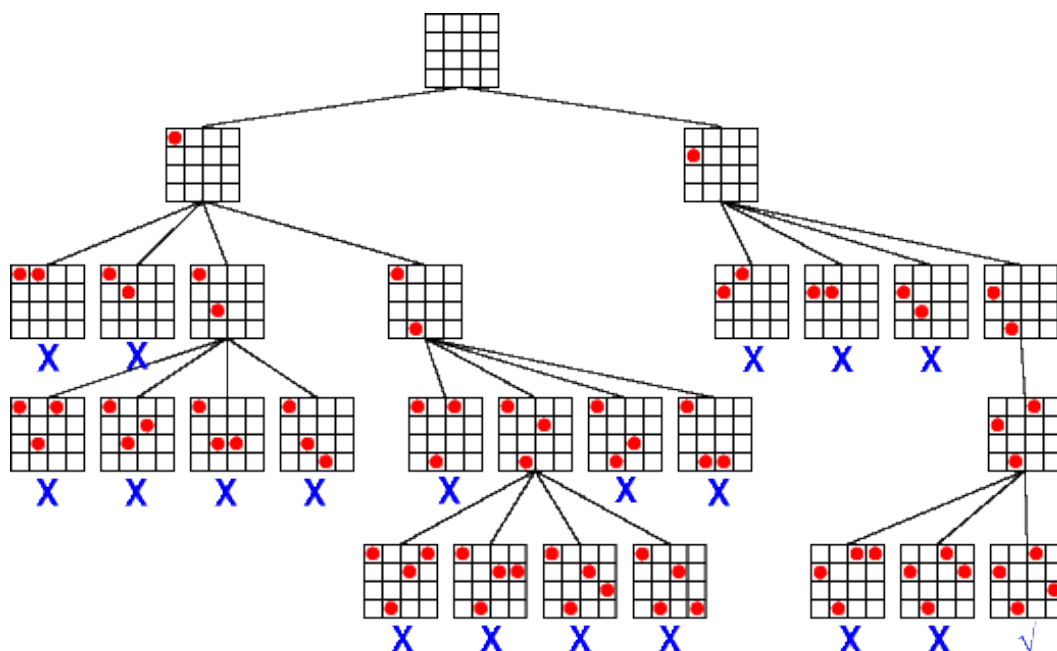
Metoda pretraživanja s vraćanjem (*engl. backtrack algorithm*) je specijalna verzija rekurzivnog algoritma gdje se može tražiti više zadovoljavajućih rješenja prema početnim parametrima. Svakim korakom ovakve vrste algoritma uklanjaju se potencijalna rješenja čim se dođe do saznanja da neko određeno rješenje više nije u granicama zadanih parametara.

Jedan od poznatijih problema kod pretraživanja s vraćanjem je problem N-kraljica. Ovaj problem je detaljno obrađen u slijedećem poglavlju.

### 7.1. Problem N-kraljica

Problema 8-kraljica predstavljen je 1848 i istražen je od strane nekoliko poznatih matematičara uključujući C. F. Gauss-a. Cilj problema je bio postaviti osam kraljica na standardnu šahovsku ploču 8x8 na takav način da niti jedna kraljica ne napada bilo koju drugu (Takefuji, 1992).

Ovdje uzimamo malo širi problem koji se odnosi na N-kraljica umjesto fiksno određenih osam kraljica. Problem možemo riješiti pomoću rekurzivnog algoritma s vraćanjem. U nastavku je vizualni prikaz kako se pomoću navedenog algoritma može tražiti rješenje problema za N=4 kraljice.



Slika 5 Rekurzivno pronalaženje pozicija kraljica za N=4 kraljice<sup>4</sup>

<sup>4</sup> Preuzeto sa <http://ktiml.mff.cuni.cz/~bartak/constraints/images/backtrack.gif>

U nastavku slijedi primjer programskog koda koji rješava navedeni problem. Programsko rješenje je napravljeno u programskom jeziku c++. Korišteno je jednodimenzionalno polje sa osam elemenata koje opisuje pozicije kraljica na ploči. Indeks elementa u polju predstavlja određeni stupac na ploči, dok vrijednost elementa polja određuje red na ploči. Zajedno, indeks elementa i vrijednost elementa, jednoznačno određuju poziciju kraljice na ploči.

```
int moze(int k[], int x, int y) {
    for(int b=0; b<x; b++)
        if(b==x || k[b]==y || b+y==x+k[b] || b-y==x-k[b])
            return 0;
    return 1;
}

void NKr(int k[], int i, int n, int& br_komb) {
    for(int a=0; a<n; a++)
        if(moze(k, i, a) ) {
            k[i] = a;
            NKr(k, i+1, n, br_komb);
        }

    if(i>=n) {
        cout << '#' << ++br_komb << ':';
        for(int i=0; i<n; i++)
            cout << ' ' << (char)(i+'a') << k[i]+1;
        cout << endl;
        // (...)
    }
}
```

Iznad je naveden primjer rekurzivnog rješenja problema<sup>5</sup>. Vidljive su dvije funkcije od kojih prva (funkcija *moze*) služi za provjeru da li je polje x,y slobodno za postaviti kraljicu ili navedeno polje x,y već napada neka od prethodno postavljenih kraljica. Druga funkcija (funkcija *Nkr*) nam je zanimljivija jer je upravo to rekurzivna funkcija koja traži rješenje navedenog problema.

U prvom dijelu funkcije Nkr se traži mjesto kraljicu u i-tom stupcu na ploči:

```
for(int a=0; a<n; a++)
    if(moze(k, i, a) ) {
        k[i] = a;
        NKr(k, i+1, n, br_komb);
    }
```

---

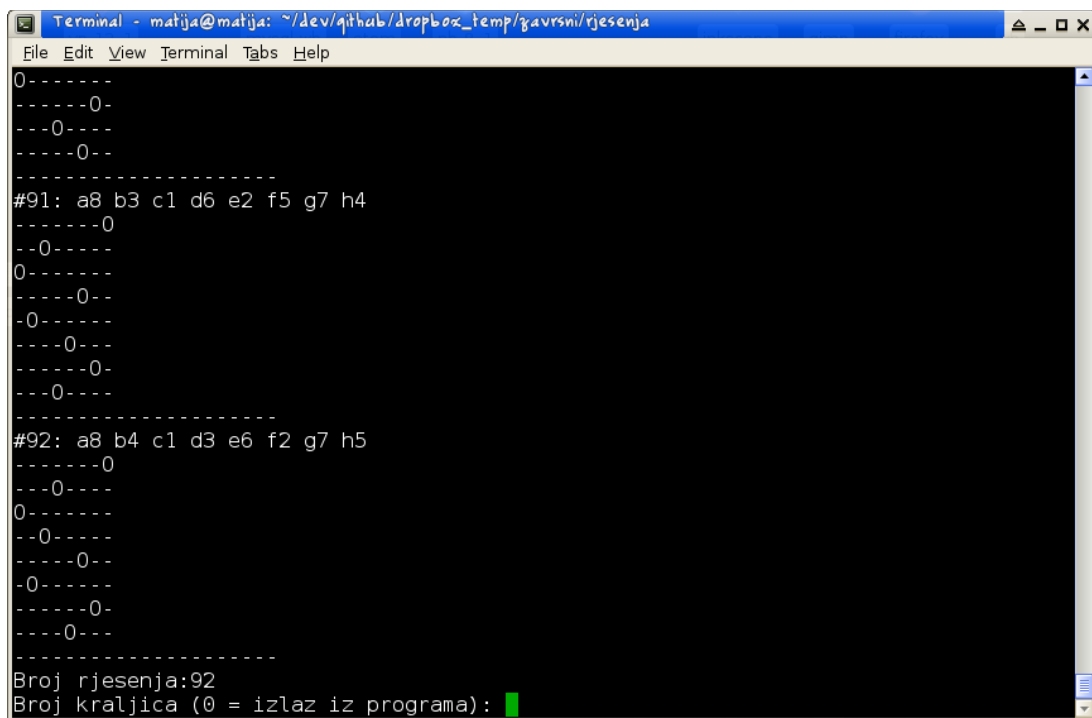
<sup>5</sup> Dostupno na <https://github.com/matijabelec/cpp-algorithms/blob/master/algorithms/n-queens-problem.cpp>, nalazi se i u prilogima na kraju rada

U drugom dijelu funkcije se radi o provjeri da li je nađeno rješenje (svi stupci su popunjeni sa kraljicama):

```
if(i>=n) {
    cout << '#' << ++br_komb << ':';
    for(int i=0; i<n; i++)
        cout << ' ' << (char)(i+'a') << k[i]+1;
    cout << endl;

    // (...)
}
```

Primjer ispisa programa za N=8 kraljica prikazan je ispod (Prikazano je samo zadnjih nekoliko rješenja jer je broj rješenja za problem N=8 kraljica jednak 92 što se vidi i na primjeru prikaza).



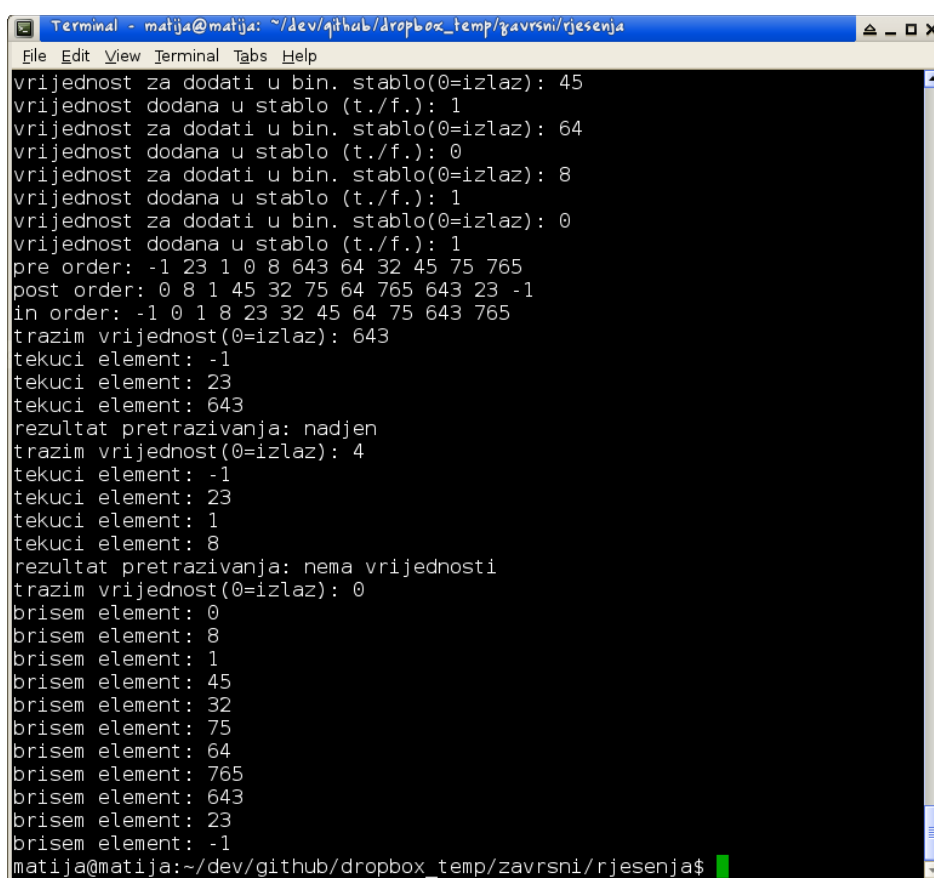
```
Terminal - matija@matija: ~/dev/github/dropbox_temp/gavrsni/rjesenja
File Edit View Terminal Tabs Help
0-----
-----0-
--0-----
-----0--
-----
#91: a8 b3 c1 d6 e2 f5 g7 h4
-----0
--0-----
0-----
-----0--
-0-----
----0---
-----0-
---0----
-----
#92: a8 b4 c1 d3 e6 f2 g7 h5
-----0
--0-----
0-----
-----0--
-0-----
----0---
-----0-
---0----
-----
Broj rjesenja:92
Broj kraljica (0 = izlaz iz programa):
```

Slika 6 Primjer rješenja problema N-kraljica za N=8

## 8. Binarno stablo

Kao završni primjer rekursije slijedi, po svojoj prirodi, rekurzivni apstraktni tip podataka. Ako uzmemo jedan element binarnog stabla tada vidimo da on ima pokazivač na lijevi, te pokazivač na desni element stabla. Lijevi element je manji element od tekućeg dok je desni element sigurno veći od tekućeg. Ovisno o implementaciji binarnog stabla, može postojati više istih vrijednosti u stablu, no, u pravilu se, kada govorimo o binarnom stablu radi o stablu čije su vrijednosti elemenata jedinstvene.

U ovom radu obradit ćemo jednostavan tip binarnog stabla. U prilogima se nalazi programsko rješenje pomoću kojeg se dolazi do sljedećeg primjera ispisa:



```
Terminal - matija@matija: ~/dev/github/dropbox_temp/završni/rjesenja
File Edit View Terminal Tabs Help
vrijednost za dodati u bin. stablo(0=izlaz): 45
vrijednost dodana u stablo (t./f.): 1
vrijednost za dodati u bin. stablo(0=izlaz): 64
vrijednost dodana u stablo (t./f.): 0
vrijednost za dodati u bin. stablo(0=izlaz): 8
vrijednost dodana u stablo (t./f.): 1
vrijednost za dodati u bin. stablo(0=izlaz): 0
vrijednost dodana u stablo (t./f.): 1
pre order: -1 23 1 0 8 643 64 32 45 75 765
post order: 0 8 1 45 32 75 64 765 643 23 -1
in order: -1 0 1 8 23 32 45 64 75 643 765
tražim vrijednost(0=izlaz): 643
tekuci element: -1
tekuci element: 23
tekuci element: 643
rezultat pretraživanja: nadjen
tražim vrijednost(0=izlaz): 4
tekuci element: -1
tekuci element: 23
tekuci element: 1
tekuci element: 8
rezultat pretraživanja: nema vrijednosti
tražim vrijednost(0=izlaz): 0
brise element: 0
brise element: 8
brise element: 1
brise element: 45
brise element: 32
brise element: 75
brise element: 64
brise element: 765
brise element: 643
brise element: 23
brise element: -1
matija@matija:~/dev/github/dropbox_temp/završni/rjesenja$
```

Slika 7 Primjer ispisa kod binarnog stabla

Kod ispisa vidimo nekoliko različitih dijelova ispisa koje ćemo kroz ovo poglavlje obraditi pošto su svi u programskom rješenju implementirani na rekurzivni način.

### 8.1. Rekurzivna funkcija za dodavanje vrijednosti u stablo

Prvi dio je dodavanje elementa u binarno stablo. Radi se o jednostavnom rekurzivnom algoritmu kojim se provjerava prvo slobodno mjesto za novi element u stablu gdje i dalje



ostaje vrijediti pravilo da svaki element sa lijevim pokazivačem (varijabla l) pokazuje na element čija je vrijednost manja od vrijednosti gledanog elementa, te da svaki element sa desnim pokazivačem (varijabla r) pokazuje na element čija je vrijednost veća od vrijednosti gledanog elementa. Programsko rješenje za prvi dio kod primjera je prikazano ispod.

```
bool dodaj(elem* bt, int v) {
    elem* tr = bt;
    if(v == bt->v) return false;
    if(v < bt->v) {
        // lijevo
        if(bt->l) return dodaj(bt->l, v);
        bt->l = new elem(v);
        return true;
    }

    // desno
    if(bt->r) return dodaj(bt->r, v);
    bt->r = new elem(v);
    return true;
}
```

Radi se o jednostavnoj rekurzivnoj funkciji koja se rekurzivno poziva sve dok se ne dođe do elementa kojem je lijevi, odnosno desni pokazivač postavljen na NULL (nema pod elementa) te se tamo postavlja novi element sa vrijednošću v. Također, kao što je na početku poglavlja spomenuto, navedena implementacija stabla koristi jedinstvene vrijednosti elemenata pa se u slučaju da vrijednost v već postoji u stablu jednostavno prekida rekurzija bez dodavanja kopije vrijednosti u stablo. Funkcija vraća podataka da li je element dodan u polje (true) ili nije (false).

## 8.2. Ispis binarnog stabla

Drugi dio algoritma je ispis binarnog stabla. Radi se o tri različite metode ispisa binarnog stabla:

1. Metoda preorder,
2. Metoda postorder,
3. Metoda inorder.

### 8.2.1. Metoda preorder

Radi se o ispisu binarnog stabla na način da se prvo ispisuje vrijednost elementa roditelja, a zatim se ispisuje vrijednost lijevog dijeteta, pa na kraju vrijednost desnog dijeteta

elementa roditelja. Korišteno programsko rješenje za navedenu metodu ispisa slijedi u nastavku.

```
void ispisi_pre(elem* bt) {
    cout << bt->v << " ";
    if(bt->l) ispisi_pre(bt->l);
    if(bt->r) ispisi_pre(bt->r);
}
```

### 8.2.2. Metoda postorder

Kod metode postorder radi se o sličnom ispisu. U ovom slučaju, prov se ispisuju vrijednosti elemenata djece i to redom: lijevo dijete, pa zatim desno dijete. Na kraju slijedi ispis vrijednosti roditelja. Programsko rješenje je sljedeće:

```
void ispisi_post(elem* bt) {
    if(bt->l) ispisi_post(bt->l);
    if(bt->r) ispisi_post(bt->r);
    cout << bt->v << " ";
}
```

### 8.2.3. Metoda inorder

Metodom ispisa inorder elemente binarnog stabla ispisujemo kao sortirani niz vrijednosti. Pošto metoda prati i održava slijedno povećanje vrijednosti elemenata, kao što je na početku poglavlja o binarnom stablu navedeno: lijevo dijete ima uvijek manju vrijednost od roditelja, a desno uvijek veću vrijednost od roditelja. Nije teško shvatiti da se ispis radi na način da se prvo ispiše najmanji član, odnosno lijevo dijete, zatim veći član, odnosno roditelja, te se na kraju ispisuje vrijednost desnog djeteta koje sadrži najveću vrijednost od tri navedene.

Programsko rješenje za metodu inorder je vrlo slično dvijema prethodno navedenim metodama, kao što je vidljivo ispod.

```
void ispisi_in(elem* bt) {
    if(bt->l) ispisi_in(bt->l);
    cout << bt->v << " ";
    if(bt->r) ispisi_in(bt->r);
}
```

## 8.3. Pretraživanje binarnog stabla

Slijedi dio algoritma koji je vezan uz pretraživanje binarno stabla. Zbog svog izgleda, binarno je stablo vrlo jednostavno pretraživati. Tako dolazimo do rekurzivnog algoritma pretraživanja koji u svakom svom koraku provjeri da li je vrijednost tekućeg elementa stabla

jednaka traženoj. Ako je jednaka tada vraća podatak da je element pronađen u polju. Ako je tražena vrijednost manja od tekućeg elementa tada se rekursivno prelazi na element koji je lijevo dijete od tekućeg elementa i ponavlja se rekursivni korak, a ako je tražena vrijednost veća od tekućeg elementa tada se rekursivno prelazi na element koji je desno dijete od tekućeg elementa.

Za gore opisani algoritam pretraživanja napravljeno je rekursivno programsko rješenje koje izgleda ovako:

```
bool pretrazi(elem* bt, int v) {
    cout << "tekuci element: " << bt->v << endl;
    if(bt->v == v) return true;
    if(bt->l && (bt->v > v) ) return pretrazi(bt->l, v);
    if(bt->r) return pretrazi(bt->r, v);
    return false;
}
```

## 8.4. Dealokacija stabla (rekursivno brisanje elemenata stabla)

Posljedni dio programskog rješenja je sama dealokacija stabla. Programsko rješenje sadrži sljedeći programski kod:

```
elem* brisi(elem* bt) {
    if(bt->l) brisi(bt->l);
    if(bt->r) brisi(bt->r);
    if(bt) {
        cout << "brisem element: " << bt->v << endl;
        delete[] bt;
    }
    return NULL;
}
```

Također je zgodno za spomenuti da se u objektno orijentiranom programiranju, kod programiranja klase za binarno stablo, pogotovo u programskom jeziku c++ u destruktor može dodati sljedeći kod (primjer je za klasu naziva bin\_stablo):

```
~bin_stablo(){
    if(l) delete l;
    if(r) delete r;
}
```

Te se ovakvim destruktorom može postići da se binarno stablo rekursivno dealocira, odnosno rekursivno se brišu svi elementi od listova stabla (krajnjih elemenata stabla koji nemaju djece) do samog korijena stabla (početni element stabla).

## 9. Nedostaci rekurzivnog pristupa

Za razliku od iterativnog pristupa kod većine korištenih algoritama u ovom radu iterativni pristup je mnogo manje čitljiviji nego rekurzivni. U ovom poglavlju obradit će se nedostaci rekurzivnog pristupa koji su navedeni kroz prijašnja poglavlja kroz korištene algoritme.

Kod svakog poziva metode<sup>6</sup> na programski stog se postavlja aktivacijski okvir<sup>7</sup> (*engl. activation record*). Kreiranje aktivacijskog okvira je brza operacija i ne stvara neko posebno ograničenje kod algoritama. No, kod neefikasnih algoritama, odnosno rekurzivnih algoritama koji teže prema velikom broju rekurzivnih poziva, vrlo brzo se dolazi do zapunjavanja programskog stoga što rezultira prekidanjem pokrenutog programa od strane operacijskog sustava (Barnett i Del Tongo, 2008).

---

<sup>6</sup> Metode ili funkcije – ovisno o programskom jeziku i metodi programiranja (objektno orijentirano ili ne)

<sup>7</sup> Activation record (negdje se spominje i activation frame) – struktura podataka koja se dodaje na programski stog kod poziva metode/funkcije

## 10. Zaključak

Nakon detaljnog objašnjenja što su to rekurzivni algoritmi, te kroz navedene primjere istih, dolazimo do nekoliko zanimljivih saznanja.

Iako su programska rješenja koja koriste rekurzivni pristup u pravilu kraća i jednostavnija za shvatiti (jer se problem rješava od osnovnih, bazičnih problema rekurzivno prema složenijima) su mnogo više intenzivnija što se tiče memorijskog zauzeća (svaki rekurzivni poziv funkcije zauzima novi okvir na stog-u (*engl. stack*)). Drugi problem je i sama brzina izvođenja pošto je poziv funkcije više zahtjevan od iterativnog prolaska na sljedeći korak.

## 11. Literatura

- [1] Barnett G, Del Tongo L (2008) Data Structures and Algorithms
- [2] Cormen T H, Leiserson C E, Rivest R L, Stein C (2009) Introduction to Algorithms (third edition)
- [3] Drozdek A (2013) Data Structures and Algorithms in C++
- [4] Knuth D E (1969) Art of Computer Programming
- [5] Sedgewick R, Flajolet P (2013) An Introduction to the Analysis of Algorithms
- [6] Skiena S S (2008) The Algorithm Design Manual
- [7] Takefuji Y (1992) Neural network parallel computing
- [8] Weiss M A (1992) Data Structures and Algorithm Analysis in C

## 12. Prilozi

### 12.1. Programska rješenja korištena u ovom radu

#### 12.1.1. Faktorijel – rekurzivni algoritam (faktorijel.cpp)

```
// g++ faktorijel.cpp -o faktorijel.out
#include <iostream>

long long faktorijel(unsigned int n) {
    return (n>1 ? n * faktorijel(n-1) : 1);
}

int main(){
    int n;
    do std::cin >> n; while(n<1);
    std::cout << faktorijel(n) << std::endl;
    return 0;
}
```

#### 12.1.2. Faktorijel – iterativni algoritam (faktorijel\_iter.cpp)

```
// g++ faktorijel_iter.cpp -o faktorijel_iter.out
#include <iostream>

int main(){
    int n;
    do std::cin >> n; while(n<1);

    long long f = 1;
    for(int i=2; i<=n; i++) {
        f *= i;
    }
    std::cout << f << std::endl;
    return 0;
}
```

#### 12.1.3. Fibonaccijev niz – rekurzivni algoritam (fibo.cpp)

```
// g++ fibo.cpp -o fibo.out
#include <iostream>

long long fibo(unsigned int n) {
    return (n > 1 ? fibo(n-1) + fibo(n-2) : n);
}

int main(){
    int n;
```

```

do std::cin >> n; while(n<0);
std::cout << fibo(n) << std::endl;
return 0;
}

```

#### 12.1.4.      **Fibonaccijev niz – iterativni algoritam (fibo\_iter.cpp)**

```

//g++ fibo_iter.cpp -o fibo_iter.out
#include <iostream>
using namespace std;

int main() {
    int n;

    cout << "unesi indeks clana niza: " << endl;
    cin >> n;

    // fibonacci iterativno
    int rj[3] = {0, 1, 1};
    if(n == 0) {
        rj[2] = 0;
    } else {
        for(int i=1; i<n; i++) {
            rj[2] = rj[0] + rj[1];
            rj[0] = rj[1];
            rj[1] = rj[2];
        }
    }

    cout << n << ". clan niza: " << rj[2] << endl;

    return 0;
}

```

#### 12.1.5.      **Quick sort (qs.cpp)**

```

#include <iostream>
using namespace std;

void qs(int* p, int a, int b) {
    int i=a, j=b;
    int m=p[(i+j)/2], s;

    while(i<=j) {
        while(p[i] < m) i++;
        while(p[j] > m) j--;
        if(i<=j) {

```



```

        s=p[i]; p[i]=p[j]; p[j]=s;
        i++; j--;
    }
}

if(a<j) qs(p, a, j);
if(i<b) qs(p, i, b);
}

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

    int* polje = new int[n];
    for(int i=0; i<n; i++) {
        cout << i+1 << ". vrijednost: " << endl;
        cin >> polje[i];
    }

    cout << "polje prije sortiranja: ";
    for(int i=0; i<n; i++) {
        cout << polje[i] << " ";
    }
    cout << endl;

    qs(polje, 0, n-1);

    cout << "polje nakon sortiranja: ";
    for(int i=0; i<n; i++) {
        cout << polje[i] << " ";
    }
    cout << endl;

    delete[] polje;
    return 0;
}

```

### 12.1.6. Merge sort (ms.cpp)

```

#include <iostream>
using namespace std;

void ms(int* p, int a, int b) {
    if(a<b) {
        int k = (a+b)/2;
        ms(p, a, k);
        ms(p, k+1, b);
        int i=a, j=k+1, c=0;
    }
}

```

```

        int *p2 = new int[b-a+1];
        while(i<=k && j<=b)
            p2[c++] = p[ (p[i]<=p[j] ? i++ : j++) ];
        if(i>k)
            while(j<=b) p2[c++] = p[j++];
        else
            while(i<=k) p2[c++] = p[i++];
        for(int m=a; m<=b; m++) p[m] = p2[m-a];
        delete[] p2;
    }
}

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

    int* polje = new int[n];
    for(int i=0; i<n; i++) {
        cout << i+1 << ". vrijednost: " << endl;
        cin >> polje[i];
    }

    cout << "polje prije sortiranja: ";
    for(int i=0; i<n; i++) {
        cout << polje[i] << " ";
    }
    cout << endl;

    ms(polje, 0, n-1);

    cout << "polje nakon sortiranja: ";
    for(int i=0; i<n; i++) {
        cout << polje[i] << " ";
    }
    cout << endl;

    delete[] polje;
    return 0;
}

```

### 12.1.7. Binarno pretraživanje – rekurzivni algoritam (bs.cpp)

```

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

```

```

int bs(int* polje, int x, int i, int j) {
    if(i>j) return -1;
    int m = (j-i)/2 + i;
    if(x == polje[m]) return m;
    if(x < polje[m]) return bs(polje, x, i, m-1);
    return bs(polje, x, m+1, j);
}

int main() {
    srand(time(0) );
    int n = 5 + rand()%10;

    int* polje = new int[n];
    polje[0] = 1 + rand()%10;
    cout << "polje: " << polje[0] << " ";
    for(int i=1; i<n; i++) {
        polje[i] = polje[i-1] + 1 + rand()%10;
        cout << polje[i] << " ";
    }
    cout << endl;

    int vrijednost;
    cout << "vrijednost koja se trazi: " << endl;
    cin >> vrijednost;

    int rezultat = bs(polje, vrijednost, 0, n-1);

    cout << "rezultat pretrazivanja(-1 ako ne postoji, indeks
ako postoji): " << rezultat << endl;

    delete[] polje;
    return 0;
}

```

### 12.1.8. Binarno pretraživanje – iterativni algoritam (bs\_iter.cpp)

```

//g++ bs_iter.cpp -o bs_iter.out
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int bs(int* polje, int n, int x) {
    int i=0, j=n-1;
    int k = (i+j)/2;
    while(i<=j) {
        if(polje[k] == x) return k;
        if(polje[k] > x) j = k - 1;
        else i = k + 1;
    }
}

```

```

        k = (i+j)/2;
    }
    return -1;
}

int main() {
    srand(time(0) );
    int n = 5 + rand()%10;

    int* polje = new int[n];
    polje[0] = 1 + rand()%10;
    cout << "polje: " << polje[0] << " ";
    for(int i=1; i<n; i++) {
        polje[i] = polje[i-1] + 1 + rand()%10;
        cout << polje[i] << " ";
    }
    cout << endl;

    int vrijednost;
    cout << "vrijednost koja se trazi: " << endl;
    cin >> vrijednost;

    int rezultat = bs(polje, n, vrijednost);

    cout << "rezultat pretrazivanja(-1 ako ne postoji, indeks
    ako postoji): " << rezultat << endl;

    delete[] polje;
    return 0;
}

```

### 12.1.9. Vraćanje novca (vracanje\_novca.cpp)

```

//g++ vracanje_novca.cpp -o vracanje_novca.out
#include <iostream>
using namespace std;

const int n = 8;
int novcanice[n] = {100, 80, 50, 20, 10, 7, 5, 3};

int vrati_iznos(int iznos, int* rjesenje) {
    for(int i=0; i<n; i++) {
        if(iznos >= novcanice[i]) {
            rjesenje[i]++;
            return vrati_iznos(iznos-novcanice[i], rjesenje);
        }
    }
    return iznos;
}

```

```

int main() {
    int rjesenje[n] = {0};
    int iznos;

    cout << "dostupne novcanice: ";
    for(int i=n-1; i>=0; i--) cout << novcanice[i] << " ";
    cout << endl;

    do {
        cout << "ukupno za vratiti (1+): " << endl;
        cin >> iznos;
    } while(iznos < 1);

    int ostatak = vrati_iznos(iznos, rjesenje);
    for(int i=0; i<n; i++) {
        cout << "novcanica (" << novcanice[i]
            << ") x " << rjesenje[i] << endl;
    }
    if(ostatak>0) cout << "ostatak: " << ostatak << endl;

    return 0;
}

```

#### 12.1.10. Problem N-kraljica

```

#include <iostream>
using namespace std;

bool vizualni_prikaz = false;

void info() {
    cout << endl;
    cout << "Problem n kraljica" << endl;
    cout << "  rjesenje by Matija Belec" << endl;
    cout << "-----" << endl << endl;
}

void prikazi_rjesenje(int k[], int n) {
    for(int i=0; i<n; cout<<endl, i++)
        for(int j=0; j<n; j++)
            cout << (k[i]==j?'0':'-');
    cout << "-----" << endl;
}

int moze(int k[], int x, int y) {
    for(int b=0; b<x; b++)
        if(b==x || k[b]==y || b+y==x+k[b] || b-y==x-k[b])
            return 0;
    return 1;
}

```

```

}
void NKr(int k[], int i, int n, int& br_komb) {
    for(int a=0; a<n; a++)
        if(moze(k, i, a) ) {
            k[i] = a;
            NKr(k, i+1, n, br_komb);
        }

    if(i>=n) {
        cout << '#' << ++br_komb << ':';
        for(int i=0; i<n; i++)
            cout << ' ' << (char)(i+'a') << k[i]+1;
        cout << endl;

        if(vizualni_prikaz)
            prikazi_rjesenje(k, n);
    }
}

int main(int argc, char *argv[]) {
    info();

    int n, * k, br_komb;
    char dn;
    do {
        cout << "prikaz punog rjesenja (d/n): ";
        cin >> dn;
    } while(dn!='d' && dn!='n');
    vizualni_prikaz = (dn=='d' ? true : false);

    while(1) {
        cout << "Broj kraljica (0 = izlaz iz programa): ";
        cin >> n;

        if(n < 1) break;

        k = new int[n];
        br_komb = 0;
        NKr(k, 0, n, br_komb);
        delete[] k;

        if(br_komb)
            cout << "Broj rjesenja:" << br_komb << endl;
        else
            cout << "Nema rjesenja!" << endl;
    }
    return 0;
}

```

### 12.1.11. Binarno stablo (bs.cpp)

```
//g++ bt.cpp -o bt.out
#include <iostream>
#include <cstdlib>
using namespace std;

struct elem {
    int v;
    elem* l;
    elem* r;

    elem(int _v=-1) {
        v = _v;
        l = r = NULL;
    }
};

elem* brisi(elem* bt) {
    if(bt->l) brisi(bt->l);
    if(bt->r) brisi(bt->r);
    if(bt) {
        cout << "brisem element: " << bt->v << endl;
        delete[] bt;
    }
    return NULL;
}

bool pretrazi(elem* bt, int v) {
    cout << "tekuci element: " << bt->v << endl;
    if(bt->v == v) return true;
    if(bt->l && (bt->v > v) ) return pretrazi(bt->l, v);
    if(bt->r) return pretrazi(bt->r, v);
    return false;
}

bool dodaj(elem* bt, int v) {
    elem* tr = bt;
    if(v == bt->v) return false;
    if(v < bt->v) {
        // lijevo
        if(bt->l) return dodaj(bt->l, v);
        bt->l = new elem(v);
        return true;
    }

    // desno
    if(bt->r) return dodaj(bt->r, v);
    bt->r = new elem(v);
    return true;
}
```

```

}

void ispisi_pre(elem* bt) {
    cout << bt->v << " ";
    if(bt->l) ispisi_pre(bt->l);
    if(bt->r) ispisi_pre(bt->r);
}

void ispisi_post(elem* bt) {
    if(bt->l) ispisi_post(bt->l);
    if(bt->r) ispisi_post(bt->r);
    cout << bt->v << " ";
}

void ispisi_in(elem* bt) {
    if(bt->l) ispisi_in(bt->l);
    cout << bt->v << " ";
    if(bt->r) ispisi_in(bt->r);
}

int main() {
    int v;

    // inicijalizacija praznog stabla
    elem* bt = new elem();

    // popuni polje
    do {
        cout << "vrijednost za dodati u bin. stablo(0=izlaz):
";
        cin >> v;
        cout << "vrijednost dodana u stablo (t./f.): " <<
dodaj(bt, v) << endl;
    } while(v > 0);

    // ispisi stablo
    cout << "pre order: ";
    ispisi_pre(bt); cout << endl;

    cout << "post order: ";
    ispisi_post(bt); cout << endl;

    cout << "in order: ";
    ispisi_in(bt); cout << endl;

    // pretrazivanje
    do {
        cout << "trazim vrijednost(0=izlaz): ";
        cin >> v;

```



```

        if(v > 0) {
            cout << "rezultat pretrazivanja: "
                 << (pretrazi(bt, v) ? "nadjeno" : "nema
vrijednosti")
                 << endl;
        }
    } while(v > 0);

    // dealokacija
    bt = brisi(bt);
    return 0;
}

```