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](#_Toc460972565)

[2. Matematička indukcija 2](#_Toc460972566)

[3. Rekurzija 3](#_Toc460972567)

[3.1. Rekurzija u programskom jeziku C++ 3](#_Toc460972568)

[3.2. Programski stog 3](#_Toc460972569)

[4. „Školski“ primjeri rekurzija 5](#_Toc460972570)

[4.1. Faktorijel 5](#_Toc460972571)

[4.2. Fibonaccijev niz 6](#_Toc460972572)

[5. Algoritmi podijeli pa vladaj 7](#_Toc460972573)

[5.1. Algoritmi sortiranja 7](#_Toc460972574)

[5.1.1. Quick sort 7](#_Toc460972575)

[5.1.2. Merge sort 9](#_Toc460972576)

[5.2. Algoritmi pretraživanja 10](#_Toc460972577)

[5.2.1. Binarno pretraživanje 11](#_Toc460972578)

[6. Pohlepni algoritmi 13](#_Toc460972579)

[6.1. Vraćanje ostatka novca 13](#_Toc460972580)

[6.2. Rastavljanje broja na egipatske razlomke 13](#_Toc460972581)

[7. Metoda pretraživanja s vraćanjem 14](#_Toc460972582)

[7.1. Problem N-kraljica 14](#_Toc460972583)

[8. Zaključak 15](#_Toc460972584)

[9. Literatura 16](#_Toc460972585)

[10. Prilozi 17](#_Toc460972586)

[10.1. Popis slika 17](#_Toc460972587)

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 riješenja koja su preuzeta iz osobnog github profila, odnosno repozitorija *cpp-algorithms*[[1]](#footnote-2).

1. Matematička indukcija

Kod stvaranja programskog riješenja i samo algoritma riješenja veliku ulogu najčešće odigrava matematička indukcija. Matematička indukcija služi za pojednostavljenje problema koji se riješava na način da se dokazuje u nekoliko manjih koraka početna tvrdnja za , te kroz i 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 ,
2. Pretpostavka indukcije – radi se pretpostavka da tvrdnja vrijedi i za proizvoljno odabrani k, čime se ustvari dobiva da je ,
3. Korak indukcije – ovdje se radi dokaz da tvrdnja vrijedi i za .

Posljednji korak indukcije se svodi na riješavanje problem uvrštavanjem da je . Tako dolazimo do zaključka kojim potvrđujemo da tvrdnja vrijedi za svaki prirodni broj.

1. Rekurzija

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.

* 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);

}

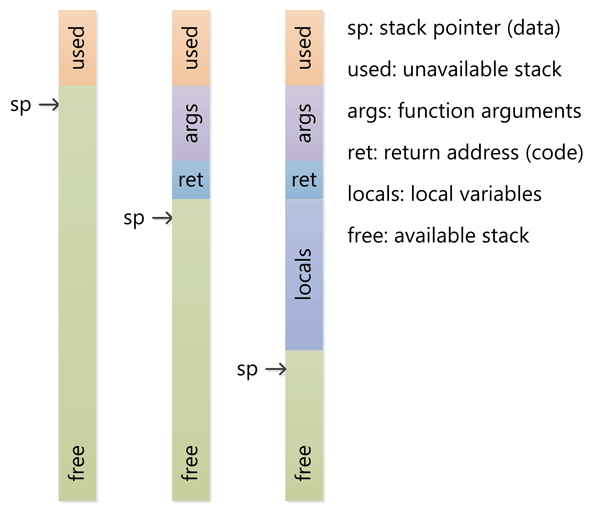
Unutar funkcije „funk“ se nalazi poziv iste. Kao baza rekurzije ovdje je postavljen uvijet što znači da će se funkcija rekurzivno pozivati sve dok se n ne umanji na vrijednost 0.

* 1. Programski stog

Kod pisanja programskog riješenja 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 riješenje problema.

Na sljedećoj slici prikazan je vizualni prikaz sadržaja stoga.



Slika Programski stog

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.

1. „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.

* 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, iznosi 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;

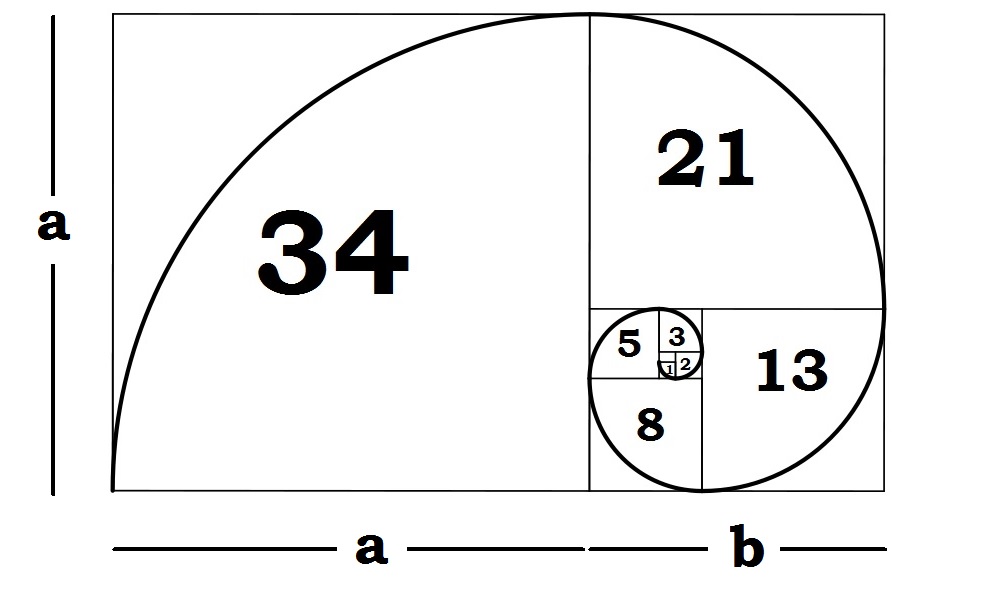
return 0;

}

* 1. 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). Navedena je osoba autor knjige Liber Abbaci u kojoj je postavio zadatak: „Koliko zečeva se može umnožiti od samo jednog para zečeva u godini?“. Europski matematičar Fibonacci dao je mnoge članke vezane uz navedeni niz u aritmetici i geometriji. (Knuth,1969).



Slika Fibonaccijev niz (zlatni rez)[[2]](#footnote-3)

Fibonaccijev niz određen je sljedećom matematičkom formulom:

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

// 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;

}

1. 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 riješava. Dobivanjem riješenja manjih problema, nadalje se dolazi do riješenja složenijeg (prvozadanog) problema.

* 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 soritiranja koja koriste metodu „podijeli pa vladaj“.

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

* + 1. Quick sort

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 riješenja 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 rekurzivno, 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 rekurzivnog 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 riješenja.

void qs(int\* p, int a, int b) {

int i=a, j=b;

int m=p[a], 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 riješenja 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 soritiranja polja kao referntna brzina algoritma uzima najgori slučaj, a u ovom slučaju to je obrnuto sortirano polje (silazno sortirani elementi). Važno je primjetiti da, navedeno programsko riješenje, uzima prvi element dijela polja kao „pivot“ element. 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 N=0 odnosno prvi element dijela polja),
* Odaberi slučajnim odabirom element polja (u pravilu ovaj način daje najviše šanse da se dođe do sortiranog polja na najbrži način u slučaju testiranja najgoreg slučaja).

Ovdje se lako može primjetiti da se slučajnim načinom odabira elemenata u slučaju obrnuto sortiranog 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 sortiranog polja, korištenjem navedenog primjera programskog riješenja 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 .

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

* + 1. 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 riješenja 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 riješenju 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 riješenja. 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 . Ovdje treba također spomenuti da se zbog specifičnosti algoritma svakim rekruzivnim dijeljenjm polja alociraju nova, manja polja što dodatno zauzima memoriju.

* 1. 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).

* + 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.

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 zamjeniti uvijet veći i manje kod provjere u koji dio polja se ulazi rekurzivno. Prema svemu gore navedenoom, algoritam možemo opisati sljedećim koracima:

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);

}

1. Pohlepni algoritmi

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

* 1. Vraćanje ostatka novca

T.

* 1. Rastavljanje broja na egipatske razlomke

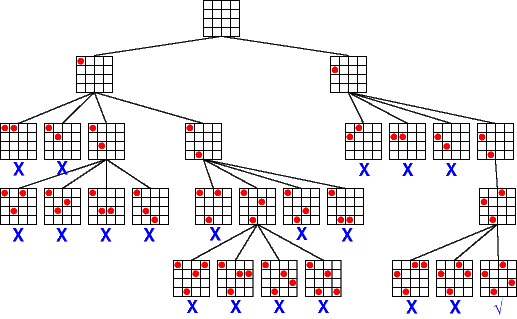
T.

1. Metoda pretraživanja s vraćanjem

T.

* 1. Problem N-kraljica

Problem N-kraljica možemo riješiti pomoću rekurzivnog algoritma s vraćanjem. U nastavku je vizualni prikaz kako se pomoću navedenog algoritma može tražiti riješenje problema za N=4 kraljice.



Slika Rekurzivno pronalaženje pozicija kraljica za N=4

U nastavku slijedi primjer programskog koda koji riješava navedeni problem.

1. Zaključak

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

Iako su programska riješenja koja koriste rekurzivni pristup u pravilu kraća i jednostavnija za shvatiti (jer se problem riješava od osnovnih, bazičnih problema rekurzivno prema složenijima) su mnogo više intenzivnija što se tiće 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.

Rekurzivni algoritmi imaju također i još jedan nedostatak, a to je upravo taj da se najčešće „riješava“ isti manji problem tokom dolaska do konačnog riješenja originalno postavljenog problema. Za ovo naime postoje neka dijelomična riješenja kao što je backtrack ili mapiranje pojedinih riješenja u pomoćno polje/listu, no tada se radi o složenijem pristupu riješavanja problema rekurzivno što direktno utječe na to da se originalno zamišljen princip rekurzivnog programiranja „podijeli problem na manje, jednostavnije probleme“ počinje lagano gubiti i prelazi se na hibridni spoj iterativnog i rekurzivnog pristupa riješavanja problema.

U pravilu je uvijek pametnije pokušati algoritam prevesti na iterativni pristup kako bi se izbjegle navedene situacije.

1. Literatura
2. \*Barnett G, Del Tongo L (2008.) Data Structures and Algorithms
3. Cormen T H, Leiserson C E, Rivest R L, Stein C (2009) Introduction to Algorithms (third edition)
4. \*Drozdek A (2013.) Data Structures and Algorithms in C++
5. Knuth D E (1969) Art of Computer Programming, Volume 3
6. \*Sedgewick R, Flajolet P (2013.) An Introduction to the Analysis of Algorithms.
7. \*Skiena S S (2008.) The Algorithm Design Manual
8. Prilozi
   1. Popis slika

* , <http://i.stack.imgur.com/9UshP.png> dostupno 06.09.2016.
* , <https://fecdn-fractalenlighten.netdna-ssl.com/wp-content/uploads/2013/11/golden-ratio-fibonacci-sequence.jpg>, dostupno 4.9.2016.
* , <http://ktiml.mff.cuni.cz/~bartak/constraints/images/backtrack.gif>, 3.9.2016.

1. Navedeni repozitorij je dostupan na: <https://github.com/matijabelec/cpp-algorithms> [↑](#footnote-ref-2)
2. Preuzeto sa <https://fecdn-fractalenlighten.netdna-ssl.com/wp-content/uploads/2013/11/golden-ratio-fibonacci-sequence.jpg>, dostupno 4.9.2016 [↑](#footnote-ref-3)