Komparacija metoda za oblikovanje algoritma

Mandarić, Maja

Undergraduate thesis / Završni rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:287336

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-20**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

Maja Mandarić

KOMPARACIJA METODA ZA OBLIKOVANJE ALGORITAMA

Završni rad

Pula, rujan 2018. godine

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

Maja Mandarić

KOMPARACIJA METODA ZA OBLIKOVANJE ALGORITAMA

Završni rad

JMBAG: 0303061715, redoviti student

Studijski smjer: Sveučilišni preddiplomski studij informatike

Kolegij: Strukture podataka i algoritmi

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan 2018. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisana, Maja Mandarić, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, rujan 2018. godine

IZJAVA

o korištenju autorskog djela

Ja, Maja Mandarić, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom „Komparacija metoda za oblikovanje algoritama“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan 2018. godine

Potpis

_____

SAŽETAK

Ovaj rad odnosi se na teorijsku podlogu metoda za oblikovanje algoritama, a to su metoda podijeli pa vladaj, metoda dinamičkog programiranja, metoda pretraživanja s vraćanjem i metoda pohlepe. Također, rad donosi opis, implementaciju i složenost primjera za svaku navedenu metodu. Budući da su temeljni pojmovi ovog rada algoritam i analiza složenosti, početak rada nudi njihovo detaljno pojmovno određenje. Posebno poglavlje odnosi se i na rekurzivne funkcije koje imaju itekakvu važnost u rješavanju problema kod nekih metoda. Ključni dio rada sadržan je u posljednjem poglavlju koji se odnosi na komparaciju navedenih metoda. Ciljevi ovog rada su stjecanje osnovnih teorijskih spoznaja o metodama za oblikovanje algoritama te ukazivanje na sličnosti i razlike navedenih metoda na temelju prikazanih primjera.

Ključne riječi: algoritam, složenost, rekurzivna funkcija, metode za oblikovanje algoritama, podijeli pa vladaj, dinamičko programiranje, metoda pretraživanja s vraćanjem, metoda pohlepe.

ABSTRACT

This thesis is related to the theoretical background of algorithm design methods. These methods are: divide and conquer, dynamic programming, backtracking and the greedy algorithm. Also, the thesis gives a description, implementation and complexity of the examples for each of the above methods. Since the core concepts of this thesis are algorithm and complexity analysis they are described in detail. Some methods use recursive functions, which are very relevant, so a special chapter is dedicated to them. Last chapter is crucial because it focuses on the comparison of the methods. Goals of this thesis are acquisition of basic theoretical knowledge about algorithm design method as well as pointing out similarities and differences among them based on given examples.

Key words: algorithm, complexity, recursive function, algorithm design methods, divide and conquer, dynamic programming, backtracking, greedy algorithm.

SADRŽAJ

1. UVOD

Prije pojave suvremenoga računalnog doba ljudi su često zapisivali korake izvršavanja svakodnevnih zadataka kako bi mogli postići važne ciljeve. Na taj način mogao se smanjiti rizik od zaboravljanja nečega što je vrlo važno. Tako se zapisani koraci određenim redoslijedom odnose na ono što predstavlja algoritam. Algoritmi su, prije pojave računala, bili sveprisutni, od matematike pa sve do raznoraznih prilika iz svakodnevnog života. Pojavom prvih računala i njihovim ubrzanim razvojem algoritmi počinju predstavljati srce same računalne znanosti. U računalnoj znanosti od velike su važnosti, uz algoritme, i podatci kojima se manipulira, a to su strukture podataka. Oni zajedno stvaraju računalni program. Dizajneri, koji se bave izradom algoritama, imaju sličan pristup razvoju algoritama kao ljudi koji su zapisivali korake. Prvo sagledaju problem kojega, zatim, opisuju koracima koji su potrebni za njegovo rješavanje. Konačno razvijaju niz operacija za postizanje tih koraka. Navedeno se odnosi na oblikovanje algoritama koje predstavlja proces zapisivanja algoritma kojim se dolazi do nedvosmislenih instrukcija od kojih je građen algoritam. Kako bi se postupak nekog algoritma mogao precizno opisati, potrebne su određene notacije. One se odnose na prirodan jezik, grafički jezik, pseudokod i programski jezik. Ne postoji opća formula pomoću koje se mogu riješiti svi problemi, već se od samog dizajnera očekuje kreativnost, znanje, ali i iskustvo. Bez obzira na to, postoje standardne metode za oblikovanje algoritama pomoću kojih se mogu riješiti različiti problemi, a to su metoda podijeli pa vladaj, metoda dinamičkog programiranja, metoda pretraživanja s vraćanjem i metoda pohlepe. Od velike je važnosti za algoritme i postupak analize algoritama kojim se približno procjenjuju potrebni resursi nekog algoritma. Najskuplji resursi, koje je potrebno uštedjeti, odnose se na vrijeme i memoriju određenog algoritma. Prema tome veliku ulogu u analizi algoritama imaju vremenska i prostorna složenost.

Središnji dio rada sastoji se od pet poglavlja, a jedno od njih je razgranato na potpoglavlja čime je rad dobio na sistematičnosti i organizaciji. Prvo poglavlje donosi porijeklo algoritma, njeno detaljno pojmovno određenje te osnovna svojstva koja mora imati svaki algoritam, a to su svojstva konačnosti, određenosti, djelotvornosti, ulaza i izlaza. Navode se i dva primjera. Prvi je iz svakodnevnog života, dok se drugi odnosi na jednostavan primjer iz matematike, a riječ je o Euklidovom algoritmu. Drugo poglavlje odnosi se na analizu složenosti algoritma koja je važna za

procjenjivanje resursa algoritma. Prema tome postoje dvije vrste složenosti koje će biti opisane, a to su vremenska i prostorna složenost. *A priori* i *a posteriori* analizama može se promatrati vrijeme izvršavanja određenog algoritma pa će biti navedene specifičnosti svake analize. Zatim će biti riječi o asimptotskoj analizi kojom su dobivene standardne funkcije pomoću kojih se lakše mogu procijeniti performanse algoritama. Tri su notacije dobivene asimptotskom analizom, a to su notacije u najgorem, najboljem i prosječnom slučaju koje će, također, biti objašnjene. U trećem poglavlju iznose se rekurzivne funkcije koje predstavljaju temelj za funkcioniranje nekih od metoda za oblikovanje algoritama. Veliku ulogu ima četvrto poglavlje koje se bavi metodama za oblikovanje algoritama gdje će se navesti četiri poznate metode, a to su metoda podijeli pa vladaj, metoda dinamičkog programiranja, metoda pretraživanja s vraćanjem i metoda pohlepe. Za svaku od metoda navodi se teorijska podloga i problemi. Bit će navedene specifikacije svakog problema, prikazat će se implementacije napisane u programskom jeziku C++ te vremenska složenost. Posljednje poglavlje, odnosno ključni dio rada odnosi se na komparaciju navedenih metoda. Tako će se kod komparacije uočiti široki spektar primjenjivosti navedenih metoda. Također, na temelju složenosti uočit će se koja od metoda daje najbolje rješenje za prikazane probleme.

Ciljevi ovog rada su stjecanje teorijskih spoznaja o metodama za oblikovanje algoritama te uvid u to koja će metoda dati najbolje rješenje na temelju prikazanih problema. Uz navedeno je vrlo važno shvatiti koje vrste problema rješava bilo koja od navedenih metoda.

2. OPĆENITO O ALGORITMU

Pojam *algoritam* temeljni je pojam u računarstvu, a nastao je u 9. stoljeću, puno prije od pojave prvih računala (Divjak i Lovrenčić, 2005). Potječe od imena iranskog matematičara Abu Ja'fara Muhammada ibn Muse al-Khwarizmija koji je 825. godine napisao knjigu *Pravila restauracije i redukcije*. Smatrao je da se matematički problem može podijeliti na manje dijelove kako bi se olakšalo i ubrzalo samo rješavanje. Pojednostavljenje problema razlog je zašto se baš on smatra ocem algoritma (Hoško i Slamić, 2009). U razvoju algoritma od velike je važnosti i Ada Lovelace koja prva uspijeva prilagoditi algoritam računalu. Pokazala je veliko zanimanje za Charles Babbageovu ideju vezanu za analitički stroj koji je trebao imati osnovne elemente današnjeg računala, ali nije izrađen. Godine 1842. talijanski matematičar Luigi Federico Menabrea objavio je članak o analitičkom stroju. Ada ga je prevela te dodala vlastite bilješke koje su sadržavale upute za računanje Bernoullijevih brojeva. Navedene upute smatraju se prvim računalnim programom, dok se Ada Lovelace smatra prvim programerom. Prošlog stoljeća u čast Adi osmišljen je programski jezik Ada (Perišić, 2017). Ne treba ni izostaviti grčkog matematičara Euklida koji je u svojoj knjizi *Elementi* opisao algoritam koji se i danas koristi, a to je Euklidov algoritam. Njegova je svrha pronaći najveću zajedničku mjeru, odnosno djelitelj dvaju prirodnih brojeva. Oznaka za najveću zajedničku mjeru je *m(a,b)* (Divjak i Lovrenčić, 2005). Konkretan primjer i postupak rješavanja navedenog algoritma bit će prikazan u odjeljku nakon definiranja pojma *algoritam.*

Pojam *algoritam* može se definirati na nekoliko načina. Najjednostavnija definicija odnosi se na „naputak kako riješiti neki zadatak ili obaviti neki posao“ (Grundler i Blagojević, 2005, str. 144). Iz toga proizlazi da je algoritam postupak pomoću kojeg se mogu riješiti određeni problemi ili dostići određeni ciljevi. Odnosi se na detaljni opis postupka prilikom rješavanja problema koji se može podijeliti na manje i jednostavnije dijelove. Algoritam se može definirati i kao „konačan skup preciznih, razumljivih i jednoznačnih instrukcija koje djelotvorno i učinkovito dovode do rješenja u konačnom vremenu“ (Hoško i Slamić, 2009, str. 9). Za razumijevanje navedene definicije potrebno je objasniti njezine ključne pojmove. Vrlo je važno da su instrukcije algoritma precizne. Tako recept za pripremu ručka može sadržavati instrukciju „treba dodati malo začina“ koja nikako nije dobra, već bi trebala biti napisana točna količina začina. Dakle, prilikom zapisivanja algoritma određeni je korak vrlo važno zapisati

bez obzira što se taj korak podrazumijeva prilikom rješavanja određenog problema. Zatim, instrukcije algoritma moraju biti razumljive i ljudima i računalu te moraju biti jednoznačne što označuje da se uvijek trebaju izvoditi na isti način. Broj instrukcija algoritma zahtijeva konačnost, odnosno završetak instrukcije algoritma nakon konačnog broja koraka. Ako instrukcije algoritma nisu konačne, neće dati nikakav rezultat. Također, algoritam mora biti djelotvoran, tj. treba dati određeni rezultat te mora biti učinkovit što se tiče vremena i memorije (Hoško i Slamić, 2009). Velike zasluge u analizi algoritma pripisuju se Donaldu Knuthu, američkom računalnom znanstveniku, koji je iznio teoriju o pet svojstava svakog algoritma. Tako svaki algoritam mora zadovoljiti svojstva konačnosti, određenosti, djelotvornosti, ulaza i izlaza. Kao što je već spomenuto u ovom radu konačnost podrazumijeva završetak instrukcije algoritma nakon konačnog broja koraka što znači da bez tog svojstva nema ni konačnog rezultata. Svojstvo određenosti zahtijeva točno definirane korake, dok svojstvo djelotvornosti zahtijeva izvršavanje algoritma uz pomoć papira i olovke u konačnom vremenu. Također, svaki algoritam mora imati nula ili više ulaza te jedan ili više izlaza (Orehovački, 2016). Nastavak rada donosi primjere algoritama iz svakodnevnog života i u matematici koji su temeljeni na navedenim svojstvima čime će njihovo razumijevanje biti olakšano.

Za primjer algoritma iz svakodnevnog života navest će se priprema palačinki. Za pripremu palačinki potrebni su sastojci i njihova količina koji će biti navedeni u nastavku.

SASTOJCI:

- 2 jaja
- 150 g brašna
- 300 ml mlijeka
- prstohvat soli

Algoritam koji može pomoći osobi prilikom pripreme je: „U posudu za pripremu smjese za palačinke potrebno je dodati mlijeko s jajima te izmiješati. Treba dodati prstohvat soli te malo po malo dodavati brašno i miješati. Smjesa za palačinke gotova je kada postane gusta, ali opet dovoljno tekuća. Smjesa treba odstajati na sobnoj temperaturi 20 minuta. Treba zagrijati tavu i staviti nekoliko kapi ulja. Palačinku je potrebno peći dvije do tri minute sa svake strane. Premazati sa željenim

nadjevom.“ Algoritam je vrlo detaljno opisan i podijeljen u manje dijelove te se problem pripreme palačinki riješio. Time su zadovoljena Knuthova svojstva određenosti i konačnosti. Zadovoljeno je i svojstvo ulaza koje se odnosilo na sastojke potrebne za pripremu palačinki te svojstvo izlaza koje predstavlja pripremljene palačinke.

Budući da su se algoritmi prvo počeli primjenjivati u matematici, prikazat će se i jednostavan primjer iz tog područja. Riječ je o Euklidovom algoritmu koji se već spomenuo i koji služi za pronalaženje zajedničke mjere, odnosno djelitelja dvaju prirodnih brojeva. U nastavku rada slijedi slika 1 koja prikazuje postupak rješavanja Euklidova algoritma koristeći brojeve 20244 i 16802.

Slika 1. Postupak rješavanja Euklidova algoritma

$$20244 = 1 \cdot 16802 + 3442$$
$$16802 = 4 \cdot 3442 + 3034$$
$$3442 = 1 \cdot 3034 + 408$$
$$3034 = 7 \cdot 408 + 178$$
$$408 = 2 \cdot 178 + 52$$
$$178 = 3 \cdot 52 + 22$$
$$52 = 2 \cdot 22 + 8$$
$$22 = 2 \cdot 8 + 6$$
$$8 = 1 \cdot 6 + 2$$
$$6 = 3 \cdot 2$$
$$\mathbf{NZM\ (20244,\ 16802) = 2}$$

Izvor: prilagođeno prema (Bobičić, 2014, str. 16)

Također, i u ovom se primjeru mogu vidjeti zadovoljena Knuthova svojstva. Svojstvo konačnosti zadovoljeno je riješenim problemom nakon konačnog broja koraka, a to je dobivena najveća zajednička mjera. Svojstvo određenosti zadovoljeno je detaljnim prikazom algoritma koji je podijeljen u manje dijelove, dok se svojstvo djelotvornosti moglo pomoću papira i olovke riješiti u konačnom vremenu. Svojstvo ulaza svodi se na brojeve 20244 i 16802, dok se svojstvo izlaza odnosi na dobivenu najveću zajedničku mjeru, a to je broj 2.

Kako bi programer mogao razviti računalni program, uz algoritme, treba biti još upućen i u strukture podataka. Strukture podataka nazivaju se još i statičkim, a

algoritmi dinamičkim aspektom programa. Tako se za strukture podataka može još reći da predstavljaju ono s čime se radi, dok su algoritmi ono što se radi. Prilikom rješavanja određenog problema ne treba se usredotočiti samo na algoritme, već i na strukture podataka koje, također, imaju veliki utjecaj na kvalitetno rješenje (Manger, 2013). S obzirom da su se algoritmi već definirali, u nastavku će biti navedena definicija za pojam strukture podataka. Pojam se odnosi na „skupinu varijabli u nekom programu zajedno s vezama između tih varijabli“. Svrha im je pružiti mogućnost pohrane određenih podataka, ali i izvršavanje operacija za upisivanje, promjenu, čitanje s tim podatcima i dr. Dakle, iz navedenog može se zaključiti da su algoritmi i strukture podataka međusobno povezani te, ako se govori o jednom, nije moguće ne spomenuti drugo (Manger, 2013).

3. ANALIZA SLOŽENOSTI ALGORITMA

Analiza algoritma postupak je kojim se približno procjenjuju potrebni resursi algoritma koji će biti utrošeni prilikom rada. Potrebni su resursi vrijeme i prostor te se procjenjuju na temelju veličine skupa ulaznih podataka. Ako se isti problem može riješiti uz pomoć više algoritama, u tom je slučaju cilj analize procijeniti koji će od tih algoritama biti najbolji, odnosno koji će najmanje potrošiti resurse (Manger, 2013). Složenost algoritma matematička je notacija pomoću koje se bilježi broj potrebnih operacija za izvršavanje algoritma (Carić i Ivanjko, 2013). Može se promatrati na temelju dva osnovna kriterija, a to su prostorna i vremenska složenost. Prostorna složenost odnosi se na potrebnu količinu memorije prilikom izvođenja algoritma (Carić i Ivanjko, 2013). Algoritam tijekom svog izvođenja može iskoristiti istu memorijsku lokaciju više puta. Prostorna složenost pomaže procijeniti postoji li dovoljna količina memorije za izvršavanje algoritma. Nastavak poglavlja temeljit će se na vremenskoj složenosti koja podrazumijeva vrijeme potrebno za izvršavanje algoritma. Može se izračunati tako da se svakoj naredbi pridoda složenost pa se na kraju računa ukupna složenost svih naredbi (Orehovački, 2016). Slijede tablica 1 i tablica 2 koje prikazuju jednostavne i složene naredbe i njihove vremenske složenosti.

Tablica 1. Vremenska složenost jednostavnih naredbi

NAREDBA	SLOŽENOST
Komentar	0
Deklarativne naredbe	0
Računske operacije	Konstanta
Operacije uspoređivanja	Konstanta
Pridruživanje vrijednosti	Konstanta
Čitanje vrijednosti	Konstanta
Pristupanje elementima polja	Konstanta
Operacije s pokazivačima	Konstanta

Izvor: Orehovački, 2016, str. 25

Tablica 2. Vremenska složenost složenih naredbi

NAREDBA	SLOŽENOST
Iteracije	Svaki korak petlje ima složenost jednaku složenosti naredbi unutar petlje, uvećanu za trajanje kontrole petlje. Ukupna složenost petlje izračunava se kao broj koraka petlje pomnožen sa složenošću jednog koraka petlje.
Selekcije	Složenost selekcije jednaka je složenosti njenog dijela koji se izvodi uvećanog za trajanje evaluacije izraza uvjeta.
Poziv procedure ili funkcije	Ako su svi argumenti elementarni tipovi podataka, onda je složenost poziva konstanta. Ako je jedan ili više argumenata polje, onda je konstanta pomnožena s veličinom polja.

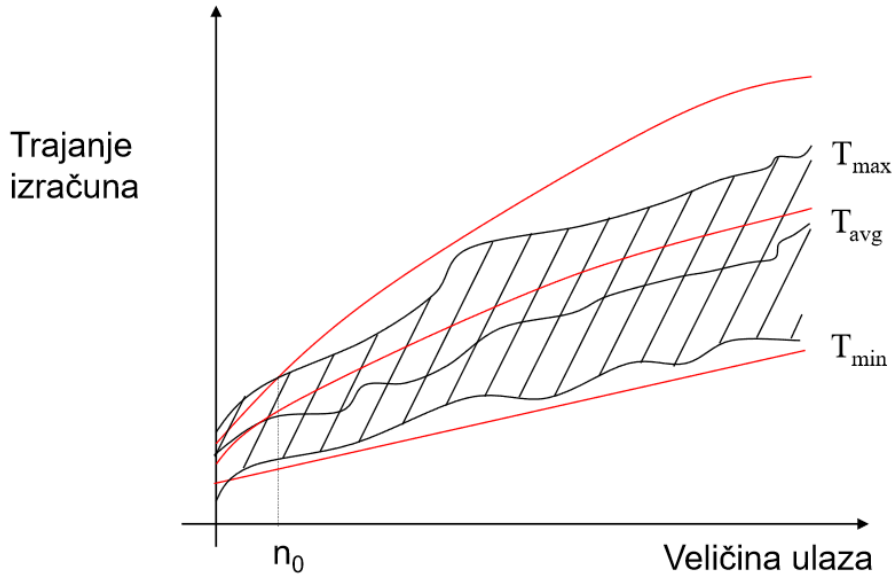
Izvor: Orehovački, 2016, str. 26

Vrijeme izvršavanja određenog algoritma može se promatrati dvjema vrstama analiza, a to su *a priori* i *a posteriori* analize. *A priori* analizom procjenjuje se vrijeme izvođenja algoritma pomoću matematičkih metoda gdje se broje naredbe koje se izvršavaju tijekom rada algoritma. Ideja je izbrojiti korake algoritma te zaključiti koje će biti vrijeme izvršavanja. Ova vrsta analize odvija se prije stvarnog mjerenja. Dakle, ne ovisi o programskom jeziku, prevoditelju ili vrsti računala. Ne može se dobiti precizno vrijeme izvršavanja algoritma, ali može se dobiti približna slika o učinkovitosti algoritma pa se zato i primjenjuje. Zatim se *a posteriori* analizom, također, procjenjuje vrijeme izvođenja algoritma, ali na odgovarajućem programu na računalu. Zato se ova analiza smatra jednostavnijom, ali i točnijom. Bez obzira na to, veliki je nedostatak mogućnost primjene jedino na program napisan u cijelosti pa se zato rijetko i koristi u praksi (Živković, 2010).

Analiza jednostavnih algoritama za procjenjivanje vremena izvršavanja algoritama može se dobiti brojanjem svake naredbe u algoritmu. Kod analize složenih

algoritama brojanje svake naredbe može dovesti do neispravnog određivanja vremena izvršavanja. Zato se analiza algoritama pojednostavljuje te se na taj način može pravilno ocijeniti vrijeme izvršavanja za velik broj ulaznih podataka. Prema tome, ne treba računati točno vrijeme izvršavanja određenog algoritma, već samo koliko brzo raste vrijeme izvršavanja algoritma ako se poveća veličina ulaza algoritma (Živković, 2010). Riječ je o asimptotskoj analizi koja se može vidjeti na grafikonu 1.

Grafikon 1. Asimptotska analiza algoritma



Izvor: Orehovački, 2016, str. 29

Dakle, cilj je asimptotske analize algoritama za ponuđenu funkciju *f(n)* pronaći funkciju *g(n)* koja će aproksimirati pri višim vrijednostima od *n.* Nije važno točno vrijeme trajanja algoritma, već njegov red veličine. Tako se složenost algoritama procjenjuje ocjenom brzine rasta funkcije (Orehovački, 2016). Asimptotskom analizom algoritama dobivene su standardne funkcije koje opisuju vrijeme izvršavanja algoritama. Pomoću njih se mogu lakše procijeniti performanse algoritama. Najčešće funkcije u analizi algoritama su konstantna, logaritamska, linearna, linearno-logaritamska, kvadratna, stupanjska ili polinomna, podeksponencijalna, eksponencijalna i faktorijelna. One i njihovo vrijeme izvršavanja algoritma bit će navedeni u tablici 3 i to od odozgo prema dolje po rastućim brzinama za velike vrijednosti (Živković, 2010).

Tablica 3. Standardne funkcije vremena izvršavanja algoritma

IME	VRIJEME IZVRŠAVANJA
Konstantna	1
Logaritamska	log *n*
Linearana	*n*
Linearno-logaritamska	*n* log *n*
Kvadratna	n^2
Stupanjska (polinomna)	n^m
Podeksponencijalna	$m^{\log n}$
Eksponencijalna	m^n
Faktorijelna	*n*

Izvor: Orehovački, 2016, str. 32

Dakle, algoritmi se uspoređuju na temelju njihovih funkcija asimptotskog vremena izvršavanja. Algoritam je bolji, odnosno brži ako je funkcija manja (Živković, 2010). Kod asimptotske analize algoritama postoje tri vrste notacija, a to su notacije u najgorem, najboljem i prosječnom slučaju. Notacija O ili *big-Oh* notacija koristi se za izvođenje algoritma u najgorem slučaju za čiju obradu treba najviše vremena. Notacija Ω ili *big-omega* notacija služi za izvođenje algoritma u najboljem slučaju za čiju obradu treba najmanje vremena, dok notacija Θ ili *big-theta* notacija prikazuje prosječno ponašanje algoritma (Orehovački, 2016).

4. REKURZIVNE FUNKCIJE

Budući da se neke od metoda za oblikovanje algoritama temelje na principu rekurzivnih funkcija, ovo poglavlje donosi objašnjenje navedenog pojma. Rekurzivne funkcije vrlo su česte funkcije u programiranju koje pozivaju same sebe zbog višestrukog izvršavanja. Korisne su za rješavanje problema koji se mogu podijeliti u jednostavnije ili manje potprobleme istog tipa. Ideja je rekurzivnih funkcija svaki uzastopni rekurzivni poziv dovesti bliže slučaju koji je lako rješiv. Slučaj koji je lako rješiv bazni je slučaj koji mora imati svaka rekurzivna funkcija te još mora imati i općeniti slučaj. Dakle, u rekurzivnim funkcijama nema iteracija, već se ponavljanje može postići tako da funkcija poziva samu sebe dok ne dođe do baznog slučaja koji omogućuje završetak ponavljanja. Rekurzivna funkcija mora imati završetak, a kada nema, nerješiva je jer se izvodi beskonačno (Živković, 2010). U nastavku je prikazan opći oblik rekurzivne funkcije.

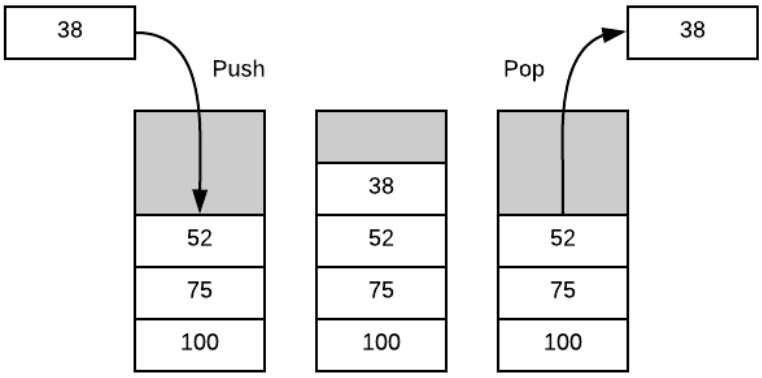
```
if (neki lako rješivi uvjet)    //bazni slučaj
    naredbe rješenja
else                            //općeniti slučaj
    rekurzivni poziv
```

Rekurzivne funkcije rade na način da se nove lokalne varijable spremaju u stog i to svaki put kada se funkcija pozove. Stog ili eng. *stack* struktura je podataka koja funkcionira prema principu LIFO odnosno *last in, first out.* Naziva se još i posebnom vrstom liste gdje se na vrh ubacuju i izbacuju elementi. Tako će element koji je zadnji ubačen, biti prvi izbačen, dok će element koji je prvi ubačen, biti zadnji izbačen (Manger, 2013). Na slici 2 prikazane su osnovne funkcije stoga *push,* koja umeće element u stog, te *pop,* koja izbacuje element sa stoga.

Određeni problemi koji se mogu riješiti uz pomoć rekurzivnih funkcija, također, mogu se još riješiti i uz pomoć iterativnih funkcija. Jedna i druga imat će isto rješenje, ali do njega dolaze na različite načine. Tako slijede pozitivne i negativne strane rekurzivnih funkcija u odnosu na iterativne funkcije. Rekurzivne funkcije imaju mogućnost jednostavnijeg zapisivanja koda za složenije probleme, dok bi iterativnim funkcijama zapisivanje bilo puno kompliciranije. Tako je veličina koda rekurzivnih funkcija puno manja u odnosu na iterativne funkcije. Navedeno se smatra glavnom

prednošću rekurzivnih funkcija. Nedostatak je što zahtijevaju puno memorijskog prostora i vremena za izvršavanje zbog spremanja lokalnih varijabli u stog. Tako se dolazi do zaključka da je rekurzivni kod kraći i jednostavniji, ali i sporiji za analizu te se može koristiti samo za određene probleme. Iterativne funkcije su kompliciranije, ali izvedba je puno bolja u usporedbi s rekurzijom (Mishra, n.d.).

Slika 2. Prikaz funkcija *push* i *pop* kod stoga



Izvor: prilagođeno prema (Soldaat, 2013)

5. METODE ZA OBLIKOVANJE ALGORITAMA

Metode za oblikovanje algoritama općenite su metode koje mogu riješiti široki spektar različitih problema. Prilikom oblikovanja algoritma, koji treba dovesti do rješenja nekog problema, treba se upitati kakvo će rješenje dati određena metoda. Ne postoji sigurnost da će odabrana metoda dovesti do korektnog rješenja pa je to potrebno tek provjeriti. Rješenje možda neće uvijek biti najkorektnije, ali će zato biti približno točno. Standardni se problemi mogu riješiti pomoću više metoda, ali najčešće se rješavaju pomoću jedne koja daje najbolje rješenje. Nastavak rada odnosit će se na najvažnije metode za oblikovanje algoritama, a to su metoda podijeli pa vladaj, metoda dinamičkog programiranja, metoda pretraživanja s vraćanjem te metoda pohlepe (Manger, 2013).

5.1. Metoda podijeli pa vladaj

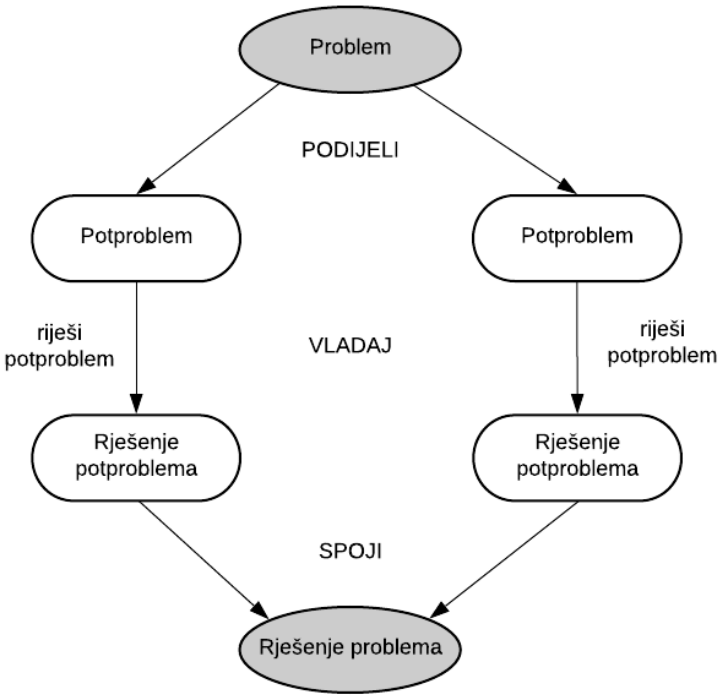
Metoda podijeli pa vladaj ili eng. *divide and conquer* najčešće je korištena metoda za oblikovanje algoritama. Nastavak rada donosi opće karakteristike metode podijeli pa vladaj. Tako će se navesti dvije podmetode ove metode, koraci izvedbe prikazani slikama i vremenska složenost *Master* i *Akra-Bazzi* teoremima. Najčešći primjeri koji se rješavaju pomoću metode podijeli pa vladaj su hanojski tornjevi, binarno pretraživanje, sortiranje spajanjem i brzo sortiranje. Prikazat će se primjeri koji se odnose na sortiranje spajanjem i Fibonaccijev niz, njihova implementacija u programskom jeziku C++ te vremenska složenost.

5.1.1. Općenito o metodi podijeli pa vladaj

Metoda podijeli pa vladaj odnosi se na podjelu složenog problema u potprobleme koji se rješavaju rekurzivno te se njihova rješenja na kraju spajaju u rješenje složenog problema (Cormen i Balkcom, n.d.). Može se zaključiti da postoje tri koraka izvedbe ove metode, a to su koraci podjele, vladanja i spajanja. Prvi je korak podjela koja rastavlja problem u potprobleme koji su zapravo manji primjerci izvornog problema. Drugi je korak vladanja koji potprobleme rješava rekurzivno, dok treći korak, tj. korak spajanja spaja riješene potprobleme kako bi se izvorni problem mogao riješiti (Moore et al., 2018). S obzirom da metoda rekurzivno rješava

potprobleme, svaki potproblem mora biti manji od prvobitnog problema te mora postojati baza za potprobleme (Cormen i Balkcom, n.d.). Na slici 3 prikazan je jedan korak dijeljenja koji stvara dva potproblema.

Slika 3. Koraci rješavanja problema s jednim rekurzivnim pozivom

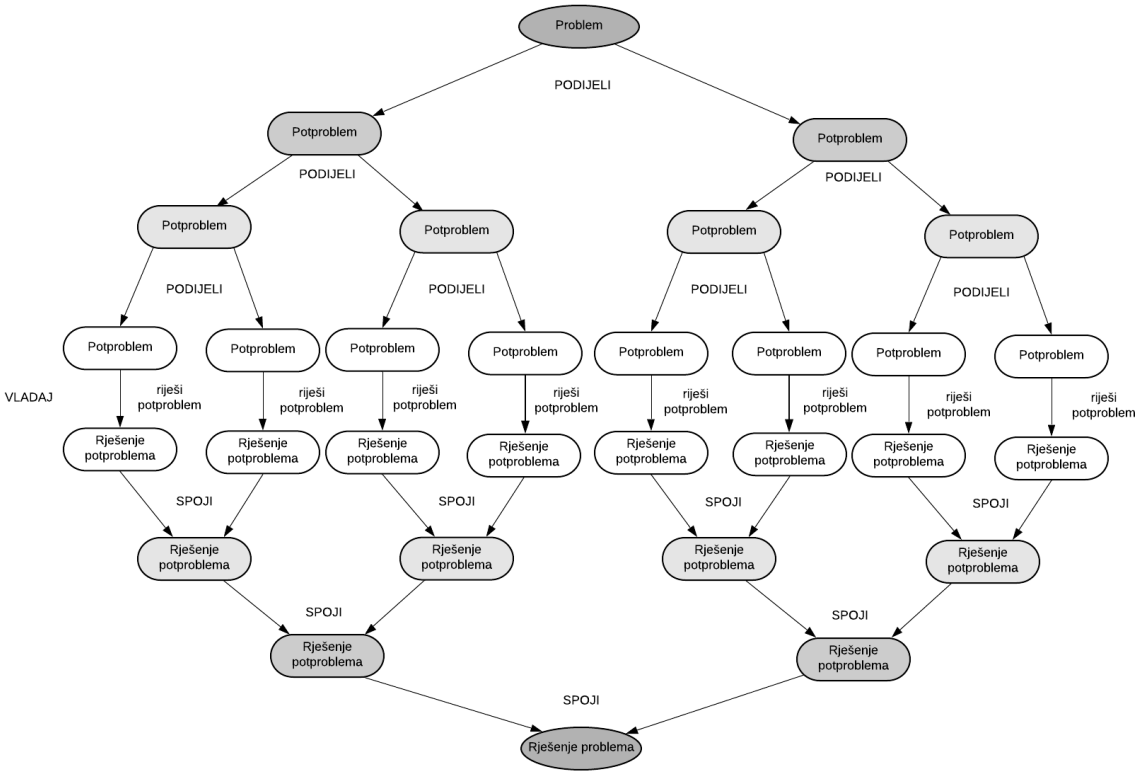


Izvor: prilagođeno prema (Cormen i Balkcom, n.d.)

Metoda podijeli pa vladaj stvara najmanje dva potproblema pa je tako i toliko rekurzivnih poziva (Cormen i Balkcom, n.d.). Ako se slika 2 proširi još s dvama rekurzivnim pozivima, ona će izgledati kao slika 4.

Dvije su podmetode metode podijeli pa vladaj, a to su smanji pa vladaj te podijeli pa vladaj. Kod podmetode smanji pa vladaj problem se treba riješiti smanjivanjem problema za određeni faktor koji može biti konstantan ili varijabilan. Tipični primjeri za ovu podmetodu su hanojski tornjevi i binarno pretraživanje. Kod podmetode podijeli pa vladaj problem se treba riješiti podjelom početnog problema na lakše, ali slične probleme koji se zatim rješavaju. Tipični primjeri za podmetodu podijeli pa vladaj su sortiranje spajanjem i brzo sortiranje (Orehovački, 2016).

Slika 4. Koraci rješavanja problema s više rekurzivnih poziva



Izvor: prilagođeno prema (Cormen i Balkcom, n.d.)

Složenost metode podijeli pa vladaj može se odrediti koristeći *Master* i *Akra-Bazzi* teoreme. *Master* teorem koristi se kod problema koji se mogu rekurzivno podijeliti na nekoliko dijelova, a koji su jednaki. Tada se složenost problema može prikazati na sljedeći način:

$$T(n) = a \cdot T \cdot \left(\frac{n}{b}\right) + f(n)$$

U navedenoj formuli oznaka *n* odnosi se na veličinu ulaza u problemu, dok oznaka *a* predstavlja broj potproblema u rekurziji. Oznaka *b* predstavlja veličinu potproblema koji su jednaki i koji se smanjuju u svakom rekurzivnom pozivu. Oznaka *f(n)* simbolizira asimptotsku složenost problema, ali bez rekurzivnih poziva. Odnosi se na složenost podjele problema i spajanje rješenja potproblema. Standardni problemi kod kojih se složenost može izračunati prema ovom teoremu su sortiranje spajanjem i binarno pretraživanje (Adamchik, 2014). *Akra-Bazzi* teorem koristi se kod problema koji nisu podijeljeni na jednake dijelove. Teorem je dobio ime po

matematičarima Mohamedu Akri i Louayu Bazziju. Složenost *Akra-Bazzi* teorema može se prikazati sljedećom formulom:

$$T(x) = f(x) + \sum_{i=1}^{k} a_i T\big(b_i x + g_i(x)\big) \text{ za } x \geq x_0$$

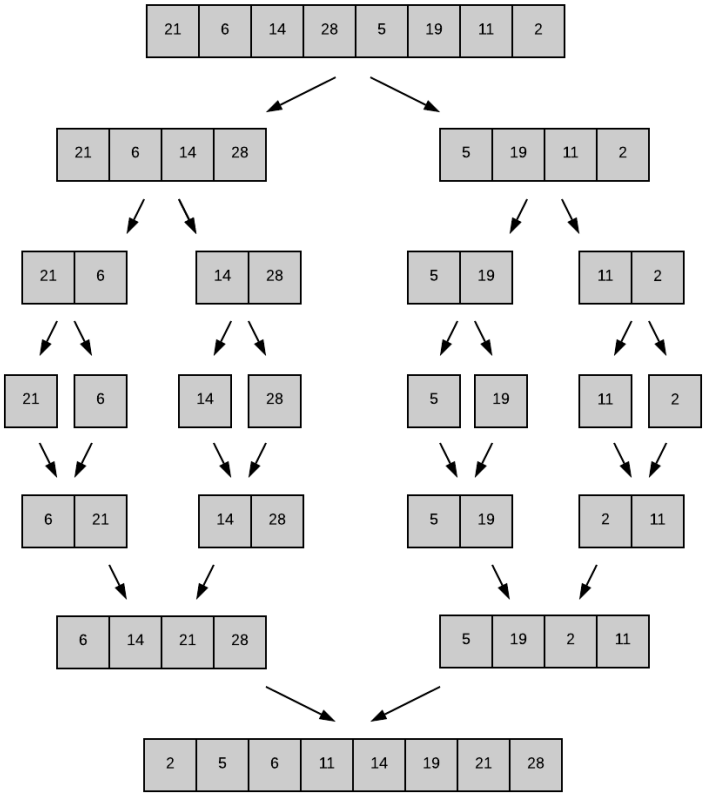
Uvjeti za korištenje *Akra-bazzi* teorema navode se u nastavku. Treba postojati dovoljan broj baznih slučajeva. Oznaka a_i i oznaka b_i odnose se na konstante za svaki *i* te oznaka a_i mora biti veća ili jednaka 0 za svaki *i.* Oznaka b_i mora biti veća ili jednaka 0 te manja ili jednaka 1 za svaki *i.* Ako $|f'(x)| \in O(x^c)$ onda se oznaka *c* odnosi na konstantu, a ako $|g_i(x)| \in O(\frac{x}{\log x)^2})$ onda se odnosi za svaki *i* te x_0 mora biti konstanta. *Akra-bazzi* teorem primjenjiviji je i korisniji teorem od *Master* teorema jer se može primijeniti na veći broj slučajeva kod kojih se pronalazi složenost (Leighton, 1996).

5.1.2. Opis, implementacija i složenost sortiranja spajanjem

Sortiranje spajanjem ili eng. *Merge sort* vrsta je algoritma koja radi po principu metode podijeli pa vladaj. Ima poprilično učinkovitu vremensku složenost o kojoj će biti riječi u nastavku ovog potpoglavlja. Sortiranje spajanjem počinje podjelom postavljenog niza na dva dijela, odnosno lijevu i desnu stranu sličnih veličina. Zatim je potrebno rekurzivno dijeliti lijevu i desnu stranu koje se na kraju sortirane uspoređuju i spajaju u jedan niz koji je konačno rješenje (Barnett i Del Tongo, 2008).

Na slici 5 prikazan je primjer sortiranja spajanjem. Dakle, elementi su brojevi 21, 6, 14, 28, 5, 19, 11 i 2 koji čine početni niz koji je potrebno sortirati. Niz će se podijeliti na dva dijela sličnih veličina. Tako će svaka strana sadržati četiri elementa. Svaka će se strana rekurzivno dijeliti sve dok se svaki element ne bude nalazio zasebno. Tada će brojevi biti sortirani te će se uspoređivati i spojiti u konačni niz koji predstavlja rješenje.

Slika 5. Primjer sortiranja spajanjem



Izvor: prilagođeno prema (Nayal, 2016)

Implementacija 1 odnosi se na sortiranje spajanjem, a napisana je u programskom jeziku C++. Od velike su važnosti dvije funkcije, *Merge i MergeSort.* Funkcija *Merge* spaja lijevu i desnu stranu, tj. potpolja u jedno polje, dok funkcija *MergeSort* rekurzivno dijeli polja na potpolja. Funkcija *MergeSort* još ima i ulogu pozvati funkciju *Merge.*

Implementacija 1. Sortiranje spajanjem riješeno metodom podijeli pa vladaj

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

void Merge(int *a, int low, int high, int mid)
{
	int i, j, k, temp[high-low+1];
	i = low;
	k = 0;
	j = mid + 1;
```

```
	while (i <= mid && j <= high)
	{
		if (a[i] < a[j])
		{
			temp[k] = a[i];
			k++;
			i++;
		}
		else
		{
			temp[k] = a[j];
			k++;
			j++;
		}
	}
	while (i <= mid)
	{
		temp[k] = a[i];
		k++;
		i++;
	}
	while (j <= high)
	{
		temp[k] = a[j];
		k++;
		j++;
	}
	for (i = low; i <= high; i++)
	{
		a[i] = temp[i-low];
	}
}

void MergeSort(int *a, int low, int high)
{
	int mid;
	if (low < high)
	{
		mid=(low+high)/2;
		MergeSort(a, low, mid);
		MergeSort(a, mid+1, high);
```

```
            Merge(a, low, high, mid);
        }
}

int main()
{
        int n, i;
        cout<<"\nEnter the number of data element to be sorted: ";
        cin>>n;
        int arr[n];
        for(i = 0; i < n; i++)
        {
                cout<<"Enter element "<<i+1<<": ";
                cin>>arr[i];
        }
        MergeSort(arr, 0, n-1);

        cout<<"\nSorted Data ";
        for (i = 0; i < n; i++)
          cout<<"->"<<arr[i];

        return 0;
}
```

Izvor: Sanfoundry n.d.

Budući da je sortiranje spajanjem algoritam temeljen na rekurzivnim funkcijama te se rješava metodom podijeli pa vladaj, vremenska složenost može se dobiti korištenjem *Master* teorema. U nastavku slijedi funkcija temeljena na *Master* teoremu kojom se može dobiti vrijeme izvršavanja algoritma:

$$T(n) = 2 \cdot T \cdot \left(\frac{n}{2}\right) + O(n)$$

Rješenje koje navedena funkcija daje je *O(nLogn).* Tako vrijeme složenosti sortiranja spajanjem za sva tri slučaja, tj. najgori, najbolji i prosječni slučaj je *nLogn.* Razlog tome je što sortiranje spajanjem uvijek dijeli niz na dva dijela i uzima linearno logaritamsko vrijeme za spajanje dva dijela kako bi se dobilo konačno rješenje (Nayal, 2016). Ovaj je algoritam jedan od najbržih algoritama za sortiranje zbog

njegove vremenske složenosti u najgorem slučaju. Prednost ovog algoritma je što može sortirati velika polja koja su pohranjena u vanjskoj memoriji računala, dok je mana dodatno trošenje memorije (Manger, 2013).

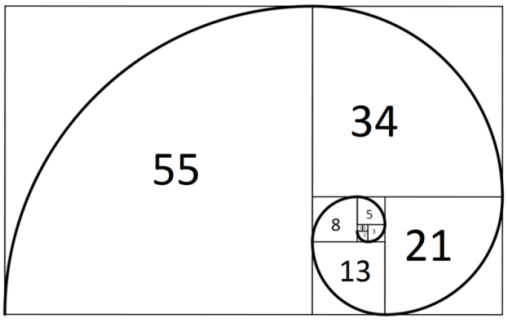
5.1.3. Opis, implementacija i složenost Fibonaccijeva niza

Fibonaccijev niz je niz brojeva u kojem se svaki sljedeći broj dobiva zbrajanjem prethodnih dvaju brojeva, a počinje od brojeva 0 i 1. Tako se dobiva niz 0, 1, 1, 2, 3, 5, 8, 13, 21, 43, 65 itd. U nastavku slijedi formula koja se odnosi na prethodno opisani postupak:

$$F_n = F_{n-1} + F_{n-2}$$

Fibonaccijevi brojevi od velike su važnosti jer se pojavljuju u prirodi, od suncokreta, uragana pa sve do galaksije. Poznata je i Fibonaccijeva spirala koja se odnosi na niz povezanih kvartalnih krugova koji su nacrtani u nizu kvadrata, a Fibonaccijevi brojevi su dimenzije. Kvadrati se uklapaju zbog prirode slijeda gdje je svaki sljedeći broj jednak zbroju dvaju brojeva prije njega. Bilo koja dva uzastopna Fibonaccijeva broja imaju omjer koji je blizu zlatnoga reza koji iznosi 1,618043. Što je veći par Fibonaccijevih brojeva, to je bliža aproksimacija. Spirala i konačni pravokutnik poznati su još kao i zlatni pravokutnik. Na slici 6 može se vidjeti prethodno opisani postupak Fibonaccijeve spirale (Hom, 2013).

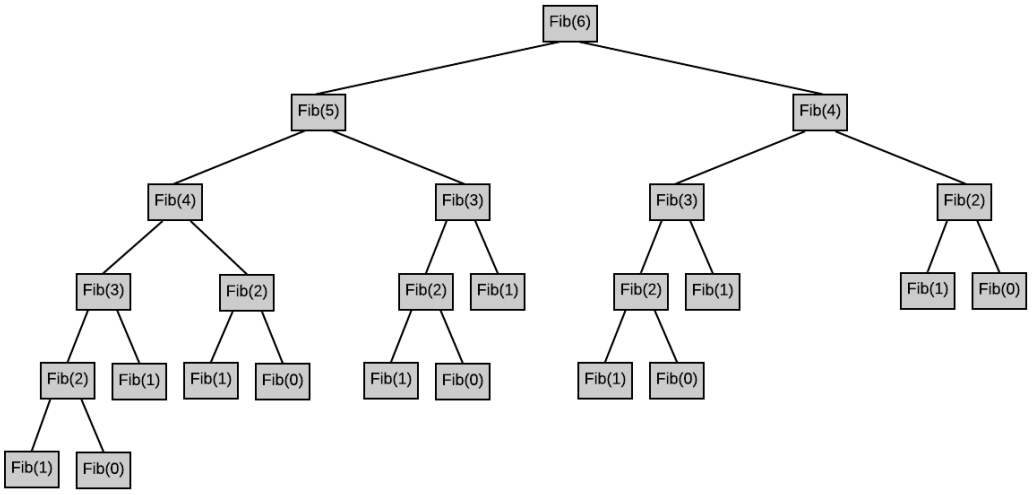
Slika 6. Prikaz Fibonaccijeve spirale



Izvor: Mullin, n.d.

Nastavak potpoglavlja donosi prikaz Fibonaccijeva niza koristeći metodu podijeli pa vladaj. Tako se *n* koji se traži dijeli rekurzivno na potprobleme sve dok svaki potproblem ne dođe do fib(0) ili fib(1). Ovaj način vrlo je spor jer se isti potproblemi pojavljuju više puta u rekurziji pa ih uvijek treba ispočetka rješavati. Navedeno je vidljivo na slici 7 gdje je *n* = 6. Tako se Fib(0) izračunava pet puta, Fib(1) osam puta, Fib(2) pet puta, Fib(3) puta, Fib(4) dva puta te Fib(5) i Fib(6) jedanput.

Slika 7. Prikaz Fibonaccijeva niza koristeći metodu podijeli pa vladaj



Izvor: prilagođeno prema (Suthers, 2015)

Implementacija 2 donosi Fibonaccijev niz riješen pomoću metode podijeli pa vladaj, a sastoji se od jedne funkcije *fib*. Ona za svaki *n* provjerava je li manji ili jednak 1. Ako je, vratit će *n*, dok će u suprotnom vratiti fib(n-1) + fib(n-2).

Implementacija 2. Fibonaccijev niz riješen metodom podijeli pa vladaj

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

int fib(int n)
{
   if (n <= 1)
   {
      return n;
   }
   else
   {
```

```
        return fib(n-1) + fib(n-2);
    }
}

int main(){
        int n;
        cout<<"Unesi n: ";
        cin>>n;
        int result = fib(n);
        cout<<result;
        return 0;
}
```

Izvor: Geeksforgeeks, n.d.

Gledajući na izraz fib(n-1) + fib(n-2) može se zaključiti da je vremenska složenost prethodno napisane implementacije eksponencijalna. Budući da je implementacija riješena rekurzivno, isti potproblemi riješeni su nekoliko puta što je loše za Fibonaccijev niz. Tako se ovaj problem rijetko rješava pomoću metode podijeli pa vladaj, a bolje rješenje donosi iduća metoda.

5.2. Metoda dinamičkog programiranja

Metoda podijeli pa vladaj katkad može biti neučinkovita jer potproblemi koje treba riješiti rastu eksponencijalno s veličinom zadanog problema. Tada ukupan broj potproblema možda nije velik, ali se isti potproblem može pojaviti na više mjesta u rekurziji pa ga tako uvijek treba ispočetka rješavati. U ovakvoj situaciji najbolje je koristiti dinamičko programiranje (Manger, 2013). Ovo potpoglavlje odnosit će se upravo na navedenu metodu za oblikovanje algoritama. Opisat će se ideja i dva pristupa, a to su odozgo prema dolje i odozdo prema gore. Prikazat će se primjeri problema riješeni dinamičkim programiranjem, njihova implementacija i vremenska složenost. Primjeri će se odnositi na Fibonaccijev niz i problem 0/1 naprtnjače.

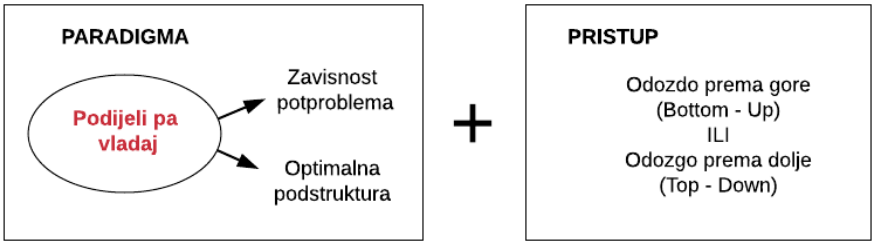
5.2.1. Općenito o metodi dinamičkog programiranja

Metoda dinamičkog programiranja ili eng. *dynamic programming* metoda je koja se koristi za rješavanje optimizacijskih problema koji mogu imati više rješenja. Cilj je

pronaći rješenje s optimalnom odnosno minimalnom ili maksimalnom vrijednošću (Stojanović, n.d). Ovom se metodom dolazi brže do rješenja nego ostalim metodama. Razlog tome je rješavanje svakog potproblema glavnog problema najviše jedanput. Iz toga proizlazi da je i smanjena složenost izvođenja algoritma (Tkalčec, 2017). Metoda s radom počinje rastavljanjem složenog problema na međusobno povezane potprobleme. Oni se rješavaju redom te se rješenja spremaju. Prvo se spremaju rješenja najmanjih potproblema pa sve veća i veća rješenja ostalih potproblema, dok se ne dođe do veličine zadanog problema. Ako rješenje određenog potproblema zatreba, neće se računati ponovno, već će se samo uzeti prethodno rješenje istog potproblema. Katkad će se riješiti manji potproblemi koji možda i neće trebati za rješenje zadanog problema. Bez obzira na to navedeni način bolji je od rješavanja istih potproblema više puta (Manger, 2013).

Metoda dinamičkog programiranja temelji se na dvama pristupima za pohranjivanje rješenja potproblema koji se poslije mogu upotrijebiti. Riječ je o pristupima odozgo prema dolje i odozdo prema gore. Oba pristupa rade istu stvar, a razlika je u načinu prijenosa rješenja potproblema. Pristup odozgo prema dolje ili eng. *top-down approach* omogućuje rastavljanje problema na potprobleme koji se rješavaju te se njihova rješenja pamte kako bi se poslije mogli iskoristiti. Ovaj pristup koristi kombinaciju rekurzije i memoizacije (Tkalčec, 2017). Memoizacija se definira kao struktura podataka u kojoj se čuvaju rješenja potproblema koja će trebati ako se isti potproblem opet pojavi. Uglavnom se koristi kod problema gdje je teško unaprijed odrediti ispravan redoslijed rješavanja potproblema. Prednost je još i jednostavan zapis koda, dok je mana sporost zbog rekurzije (Kumar, 2016). Pristup odozdo prema gore ili eng. *bottom-up approach* pristup je gdje se prvo rješavaju najjednostavniji potproblemi čija se rješenja koriste za rješavanje složenijih potproblema, dok se ne pronađe konačno rješenje. Pozitivna je strana ovog pristupa učinkovitost gledajući na potrošnju memorije, dok su negativne strane teško određenje potrebnih potproblema za rješavanje složenijih problema i kompliciranо zapisivanje koda. Na slici 8 prikazano je funkcioniranje dinamičkog programiranja. Tako kao paradigmu koristi metodu podijeli pa vladaj. Od velike važnosti za dinamičko programiranje još je i zavisnost potproblema te optimalna podstruktura koju ima onaj problem do čijeg se rješenja može doći uz pomoć optimalnih rješenja njegovih potproblema. Uz navedeno može se upotrijebiti jedan od dvaju spomenutih pristupa.

Slika 8. Funkcioniranje dinamičkog programiranja

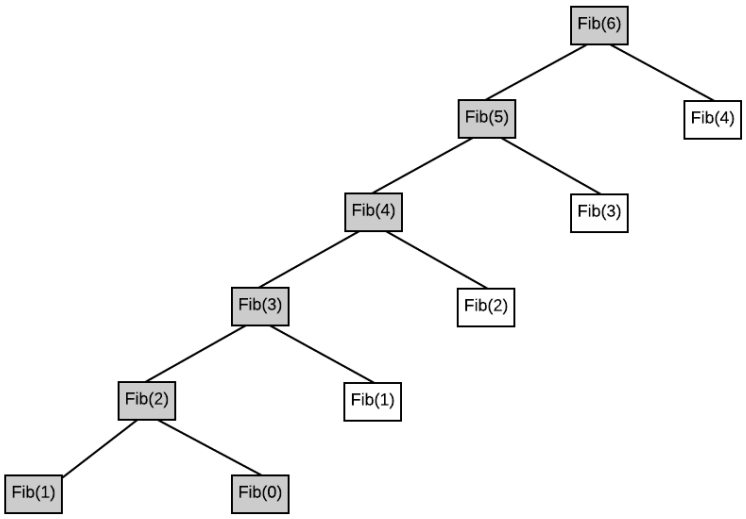


Izvor: prilagođeno prema (Trekhleb, n.d.)

5.2.2. Opis, implementacija i složenost Fibonaccijeva niza

Nastavak potpoglavlja donosi prikaz Fibonaccijeva niza koristeći pristupe odozgo prema dolje ili eng. *top-down approach* i odozdo prema gore ili eng. *bottom-up approach*. Pristup odozgo prema dolje naziva se još i memoizacijom jer se izračunati rezultat određenih argumenata funkcije pamti kako se ne bi ponovno trebala izračunati kad se pozove funkcija s istim argumentom. Primjenom ovog pristupa potproblemi se rješavaju samo jednom. Koristeći memoizaciju uklonila se potreba dijeljenja svih brojeva što se može vidjeti na slici 9 gdje je n = 6. Dakle, problem se razbio na dijelove. Prvo se rješava najveći broj pomoću Fib[n-1], zatim manji od njega pomoću Fib[n-2] itd. (Joshi, 2017).

Slika 9. Prikaz Fibonaccijeva niza koristeći pristup odozgo prema dolje



Izvor: prilagođeno prema (Sandam, 2015)

Slijedi implementacija 3 napisana u programskom jeziku C++. U funkciji *fib* pomoću selekcije *if* postavlja se uvjet. Ako f[n] nije jednak -1, tada postoji već izračunata vrijednost. Tada više ne treba računati, već samo vratiti istu vrijednost. U *main* funkciji iteracijom *for* najprije treba postaviti cijeli niz na -1 što označava da nema izračunate vrijednosti.

Implementacija 3. Fibonaccijev niz riješen pristupom odozgo prema dolje

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

int f[51];
int fib(int n) {
      if(f[n]!=-1){
            return f[n]; }
      f[n]=fib(n-1)+fib(n-2);
      return f[n];
}

int main(){
      for(int i=0;i<51;i++)
      {
            f[i]=-1;
      }

      f[0]=0;
      f[1]=1;

      int n;
      cout<<"Unesi n: ";
      cin>>n;
      int result = fib(n);
      cout<<result;
}
```

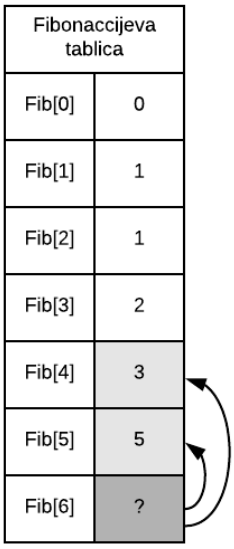
Izvor: Geeksforgeeks, n.d.

Budući da se ni jedan čvor ne poziva više od jednom, ovaj pristup za rješavanje Fibonaccijeva niza ima linearnu vremensku složenost *O(n).* Prostorna je složenost

zbog rekurzije linearna *O(n)*, ali postoji bolji način koji ima prostornu složenost konstantnu *O(1),* a on se navodi u nastavku (Joshi, 2017).

Drugi je pristup odozdo prema gore koji prvo rješava najjednostavnije potprobleme. Njihova rješenja koriste se za rješavanje složenijih potproblema sve dok se ne dođe do konačnog rješenja. Za ovaj pristup koristi se Fibonaccijeva tablica koja počinje s osnovnim slučajem, a to su brojevi 0 i 1. Ostatak tablice popunjava se odozgo prema gore te se na taj način gradi put do konačnog rješenja (Joshi, 2017). Navedeno se može vidjeti na slici 10.

Slika 10. Prikaz Fibonaccijeva niza koristeći pristup odozdo prema gore



Izvor: prilagođeno prema (Kumar, 2015)

Za implementiranje Fibonaccijeva niza ovim pristupom potrebna je jedna funkcija koja računa Fibonacijev niz. Navedeno se može vidjeti na implementaciji 4 gdje treba postaviti početne vrijednosti odnosno brojeve 0 i 1 u polja, a zatim uz pomoć iteracije postaviti formulu pomoću koje se može izračunati rješenje za određeni *i.*

Implementacija 4. Fibonaccijev niz riješen pristupom odozdo prema gore

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

int fib(int n)
{
      int fib[n];
```

```
        fib[0]=0;
        fib[1]=1;

for(int i=2; i<=n; i++)
        {
              fib[i]=fib[i-2]+fib[i-1];
        }
        return fib[n];
}

int main(){
        int n;
        cout<<"Unesi n: ";
        cin>>n;
        int result = fib(n);
        cout<<result;
}
```

Izvor: Geeksforgeeks, n.d.

Vremenska je složenost Fibonaccijeva niza za ovaj pristup linearna odnosno *O(n).* Tako će primjerice za pronalaženje 20-og Fibonaccijeva broja trebati i toliko koraka. Što se tiče prostorne složenosti, ona je konstantna odnosno *O(1).* Tome pridonosi brži pristup memoriji jer se izbjegavaju pozivi funkcija (Joshi, 2017).

5.2.3. Opis, implementacija i složenost 0/1 problema naprtnjače

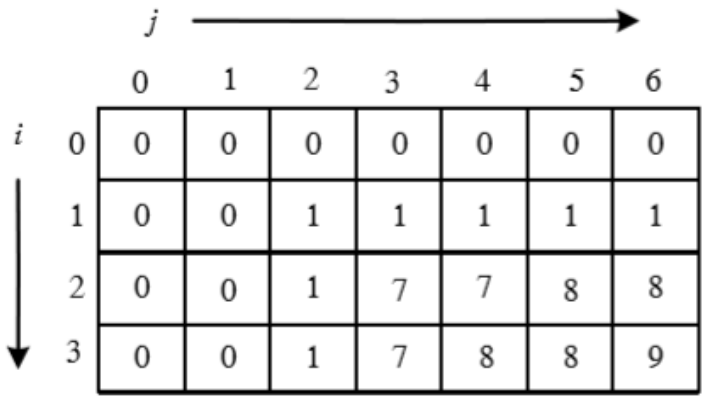
Problem 0/1 naprtnjače ili eng. *0/1 knapsack problem* pripada problemima kombinatorne optimizacije. Za rješavanje problema potrebno je zadati n predmeta O_1, O_2, . . . , O_n i naprtnjaču. Svaki predmet O_i ima težinu w_i i vrijednost p_i. Kapacitet naprtnjače predstavlja c težinskih jedinica. Cilj je ovoga problema staviti određene predmete u naprtnjaču, ali uz uvjete da ukupna težina ne prijeđe kapacitet c te da ukupna vrijednost bude maksimalna. Također, treba uzeti u obzir nedjeljivost predmeta pa se tako predmet može uzeti u cijelosti ili se ne može uzeti nikako. Budući da je riječ o problemu optimizacije, od svih rješenja koja su dopustiva, tj. zadovoljavaju ograničenja problema, traže se samo ona koja su optimalna. Optimalnost se može izmjeriti funkcijom cilja koja se minimizira ili maksimizira. Tako kod problema 0/1 naprtnjače jedno je od dopustivih rješenja ono koje neće prijeći

kapacitet naprtnjače, dok se funkcija cilja, koja će se maksimizirati, odnosi na ukupnu vrijednost predmeta u naprtnjači (Manger, 2013).

Metoda koja je pogodna za rješavanje ovog problema je metoda dinamičkog programiranja. Tako oznaka K_{ij} predstavlja maksimalnu vrijednost koja se dobiva biranjem predmeta iz skupa $O_1, O_2, \ldots, O_i$ uz kapacitet *j.* Predmet O_i kod postizanja K_{ij} može ili ne mora biti postavljen u naprtnjaču. Ako nije postavljen, tada je $K_{ij} = K_{i-1,j}$, a ako je postavljen izabrani su predmeti optimalni izbor skupa $O_1, O_2, \ldots, O_i$ uz kapacitet $j - w_i$ (Manger, 2013). Tako se na slici 11 može vidjeti rješavanje problema 0/1 naprtnjače uz pomoć tablice koja se ispunjava vrijednostima K_{ij}, a traži se K_{nc}. Računanje se izvršava redak po redak gdje se *j* brže mijenja nego *i.* U nastavku navode se ulazni podatci pomoću kojeg će se riješiti problem 0/1 naprtnjače.

n = 3
(w1,w2,w3) = (2,3,4)
(p1,p2,p3) = (1,7,8)
c = 6

Slika 11. Prikaz problema 0/1 naprtnjače koristeći metodu dinamičkog programiranja



Izvor: Manger, 2013, str. 132

Gledajući na sliku 11 zaključuje se da maksimalna vrijednost naprtnjače iznosi 9 što se postiglo biranjem prvog i trećeg predmeta čija je ukupna težina jednaka 6.

Implementacija 5 koja rješava problema 0/1 naprtnjače sadrži dvije funkcije, a to su *max* i *knapSack.* Svrha je funkcije *max* vratiti najveća dva cjelobrojna broja, dok funkcija *knapsack* vraća maksimalnu vrijednost koja se može staviti u naprtnjaču kapaciteta *W*. Unutar funkcije *knapsack* nalazi se polje K[][] koje sprječava ponovno rješavanje potproblema.

Implementacija 5. 0/1 problem naprtnjače riješen dinamičkim programiranjem

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

int max(int a, int b) { return (a > b)? a : b; }

int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }
   return K[n][W];
}

int main()
{
    cout << "Enter the number of items in a Knapsack:";
    int n, W;
    cin >> n;
    int val[n], wt[n];
    for (int i = 0; i < n; i++)
```

```
    {
        cout << "Enter value and weight for item " << i << ":";
        cin >> val[i];
        cin >> wt[i];
    }
    cout << "Enter the capacity of knapsack";
    cin >> W;
    cout << knapSack(W, wt, val, n);
    return 0;
}
```

Izvor: Sanfoundry, n.d.

Vremenska složenost ove implementacije jednaka je *O(nc),*gdje *n* predstavlja broj predmeta, a *W* se odnosi na kapacitet naprtnjače.

5.3. Metoda pretraživanja s vraćanjem

Metoda pretraživanja s vraćanjem metoda je koja rješava teške kombinatorne probleme. Smatra se jednom od učinkovitijih metoda (Manger, 2013). Nastavak potpoglavlja donosi opće karakteristike ove metode uz koju će se navesti i metoda *branch and bound* koja se smatra optimizacijskom metodom *backtrackinga*. Bit će prikazani problemi koji se mogu riješiti metodom pretraživanja s vraćanjem, implementacija u programskom jeziku C++ i vremenska složenost.

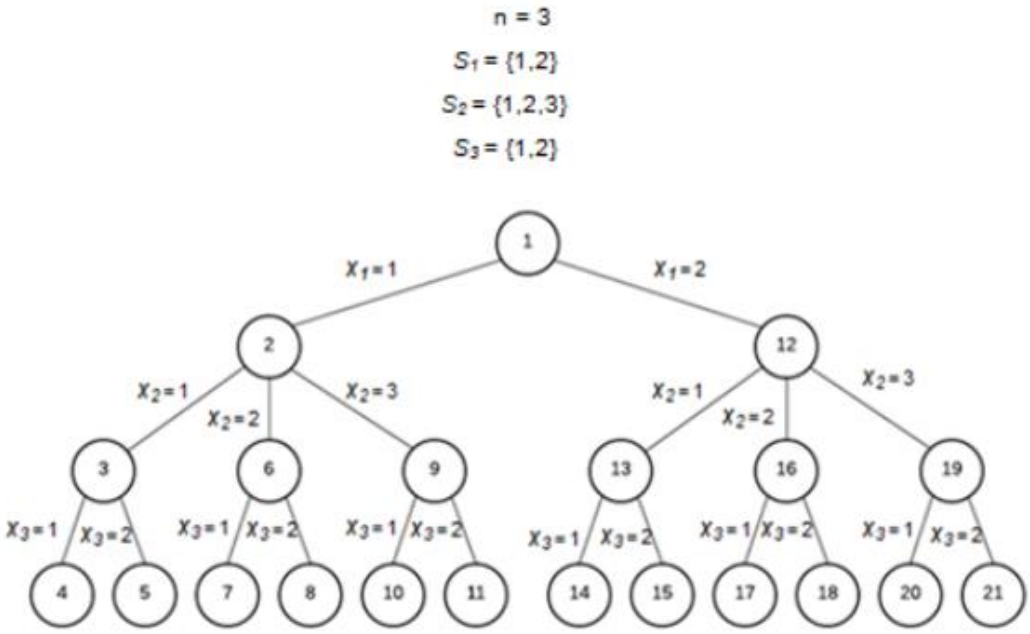
5.3.1. Općenito o metodi pretraživanja s vraćanjem

Metoda pretraživanja s vraćanjem ili eng. *backtracking* metoda je koja se temelji na potpunom pretraživanju stabla prostora stanja problema, tj. svih mogućih rješenja nekog problema. Prilikom pronalaženja rješenja algoritam stvara kandidate, ali i izbacuje potencijalne kandidate koji ne mogu stvoriti ispravno rješenje. Na taj se način smanjuje broj mogućih rješenja te se može brže doći do ispravnih rješenja. Prednost je metode unatražno pretraživanje koje se odnosi na postavljanje početnih uvjeta prije izvođenja algoritma koji će smanjiti broj mogućih rješenja (Dunić, 2013). Dakle, ideja je ove metode stvarati kandidate pomoću rekurzije i tijekom procesa uvidjeti koja će rješenja biti neispravna, prestati računati na njih te se vraćati na

prethodne korake kako bi se pronašlo ispravno rješenje. Ova metoda trebala bi uvijek dati ispravna rješenja (Iacono, 2017).

Rješenje određenog problema izražava se kao *n*-torka koja se zapisuje u obliku ($x_1, x_2, \ldots, x_n$). Oznaka x_i element je konačnog skupa S_i, a Kartezijev produkt $S_1 \cdot S_2 \cdot \ldots \cdot S_n$ prostor je rješenja. Konkretna *n*-torka, koja je u prostoru rješenja, treba ispuniti određena ograničenja koja ovise o samom problemu. Prostor rješenja zamišlja se kao uređeno stablo rješenja. Korijen stabla odnosi se na sve moguće *n*-torke, tj. na skup svih mogućih rješenja. Čvorovi nižih razina odnose se na manje skupove izoliranih rješenja, a razlikuju se po svojstvima. Tako dijete korijena predstavlja sve *n*-torke kod koje prva komponenta x_1 sadrži određenu vrijednost. Zatim, unuk korijena odnosi se na sve *n*-torke kod kojeg su prve dvije komponente x_1 i x_2 fiksirane na neki određeni način. List stabla odnosi se na jednu konkretnu *n*-torku (Manger, 2013). Na slici 12 prikazan je jednostavan primjer stabla rješenja.

Slika 12. Prikaz primjera stabla rješenja



Izvor: prilagođeno prema (Manger, 2013, str. 136)

Dakle, *backtracking* metoda radi po principu rekurzije gdje se istovremeno stvaraju i ispituju čvorovi u stablu rješenja. Budući da je riječ o rekurziji, čvorovi se postavljaju na stog. Uvijek se uzima čvor s vrha stoga te se provjerava predstavlja li taj čvor rješenje problema. Ako predstavlja, izvršit će se odgovarajuća akcija kao što

je primjerice ispis rješenja, a ako ne predstavlja, generirat će se njegova djeca koja se stavljaju na stog. Dakle, algoritam će započeti korijenom stabla na stogu, a završit će kad se pronađe rješenje ili kad stog postane prazan. Na gornjoj slici 6 numeracija čvorova preklapa se s redoslijedom obrađivanja čvorova, odnosno redoslijedom skidanja sa stoga (Manger, 2013).

Budući da će veličina rješenja rasti eksponencijalno s veličinom problema *n,* dobar algoritam neće generirati cijelo stablo rješenja, već će rezati ona podstabla koja neće dovesti do rješenja. Kod generiranja čvorova treba provjeriti ograničenja *n*-torke koja moraju biti zadovoljavajuća da bi bila rješenje. Na *i*-toj razini čvor predstavlja *n*-torke gdje je prvih *i* komponenti $x_1, x_2, \ldots, x_i$ fiksirano na određeni način. Ako na temelju tih vrijednosti ograničenja nisu ispunjena, tada taj čvor ne treba generirati jer ni djeca neće dati rješenje (Manger, 2013). Navedeno se može primijeniti na slici 6. Ako je zadano ograničenje gdje su kompnente x_1, x_2 i x_3, međusobno različite, tada lijeva strana koja je generirana postaje ona za $x_1 = 1$, $x_2 = 3$, $x_3 = 2$, a broj generiranih čvorova je 11.

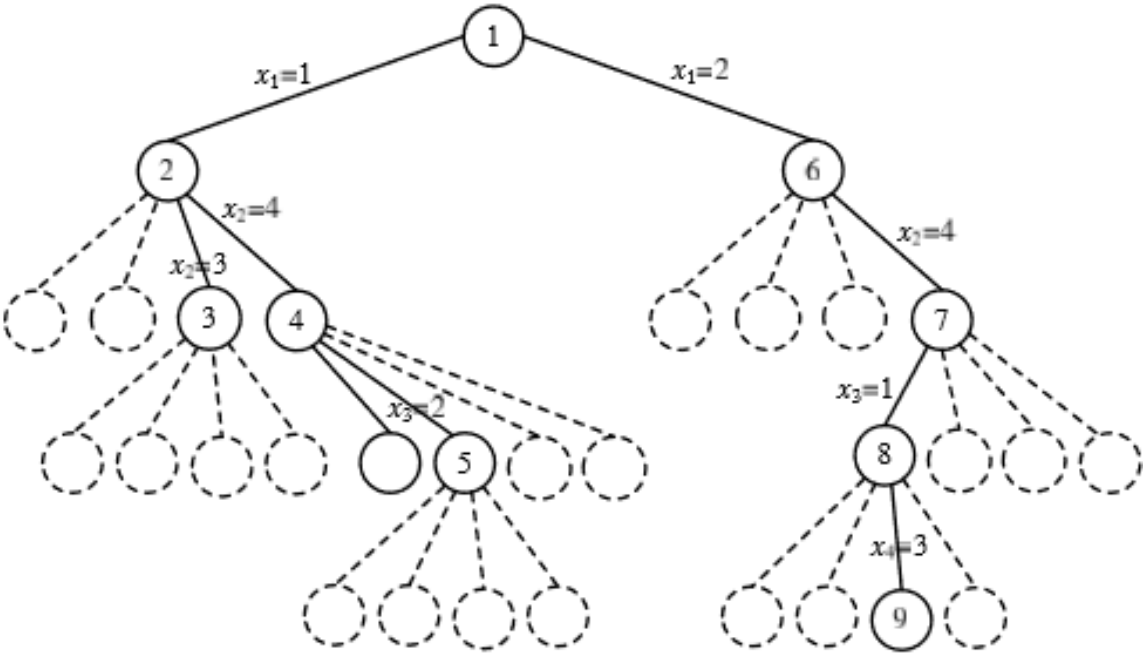
Metoda *backtracking* može se koristiti za optimizacijske probleme i to onda kada treba napraviti određene promjene koje će dodatno ubrzati rad. Kod standardnog *backtrackinga* u stablu rješenja režu se grane koje ne vode do rješenja, dok postoji način rezanja onih grana koje ne vode do boljeg rješenja u odnosu na one koje postoje. Taj način odnosi se na *branch and bound* metodu koja je poboljšana verzija *backtrackinga.* Problemi koji se rješavaju ovom metodom obično su eksponencijalni u smislu vremenske složenosti i mogu zahtijevati istraživanje svih mogućih permutacija u najgorem slučaju. *Branch and bound* metoda relativno brzo rješava ove probleme. Cilj je ove poboljšane verzije što prije pronaći dobro rješenje (Manger, 2013).

5.3.2. Opis, implementacija i složenost *n*-kraljice

N kraljica je problem postavljanja *n* kraljica na šahovsku ploču dimenzije *n x n* uz uvjet da dvije kraljice ne napadaju jedna drugu. Ovaj problem ima rješenje za sve *n-ove* koji su veći od 3. Što više raste *n,* put do rješenja postaje sve kompleksniji. Tako je najjednostavnije i najbrže naći rješenje za *n* koji iznosi 4. Zbog toga se ovaj algoritam najčišće prikazuje koristeći vrijednost *n* = 4. Ako se u obzir uzme *n* = 8, tada će postojati 92 rješenja gdje je njih 12 različito, a ostala proizlaze iz tih 12 rotacija ili simetrija šahovske ploče (Manger, 2013).

Nastavak rada donosi opis i način rješenja *n* kraljice koristeći metodu *backtracking.* Svaka kraljica mora biti u posebnom retku ploče pa se zato uzima da je *i*-ta kraljica u *i*-tom retku. Rješenje se problema može prikazati *n*-torkom ($x_1, x_2, \ldots, x_n$), a x_i predstavlja indeks stupca u kojem se nalazi *i*-ta kraljica. Tako je S_1 ($1, 2, \ldots, n$) za svaki *i,* a broj je *n*-torki u prostoru rješenja n^2. Rješenje ($x_1, x_2, \ldots, x_n$) mora zadovoljiti određena ograničenja, a to je da dvije kraljice ne smiju biti u istom stupcu ni na istoj dijagonali. Na slici 13 može se vidjeti algoritam koji generirana stablo rješenja za *n* = 4. Algoritam prestaje raditi onda kada se pronađe prvo rješenje. Čvorove koji su označeni isprekidanim linijama algoritam je poništio jer ne ispunjavaju ograničenja (Manger, 2013).

Slika 13. Prikaz stabla rješenja *n*-kraljice

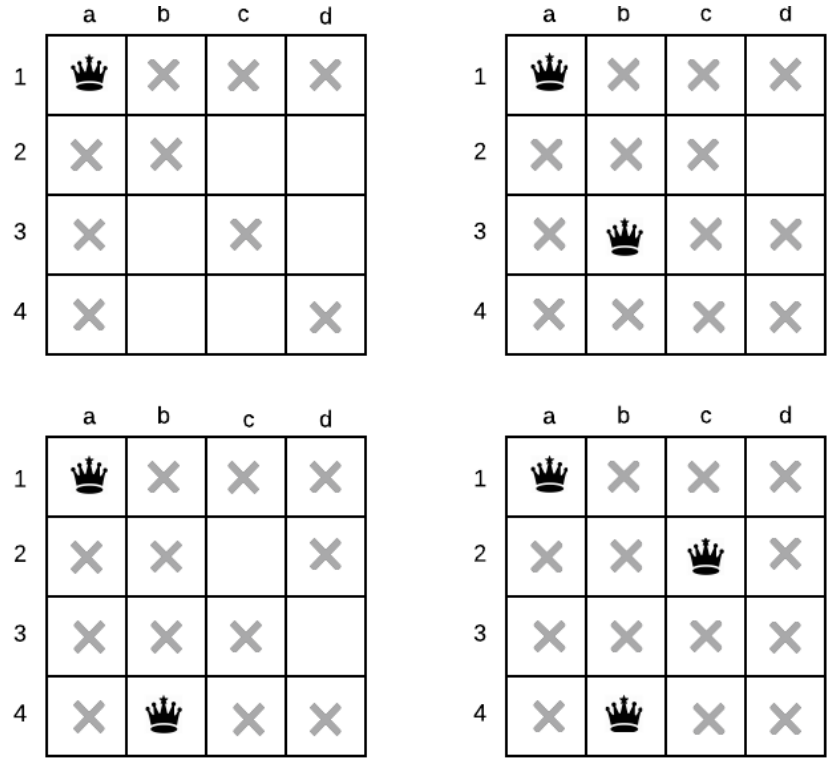


Izvor: prilagođeno prema (Manger, 2013, str. 137)

Na slici 14 mogu se vidjeti koraci algoritma za *n* kraljicu koji započinje s prvim slobodnim mjestom. Dimenzija šahovske ploče je 4 x 4. Tako algoritam za pronalaženje rješenja *n* kraljice počinje postavljanjem kraljice *k1* na poziciju *a1*. Nakon toga potrebno je prekrižiti cijeli stupac, red i dijagonalu na kojima se nalazi kraljica *k1*. Nakon toga kraljicu *k2* treba postaviti na prvo slobodno mjesto u stupcu *b*, a to je *b3.* Također, trebaju se prekrižiti stupac, redak i dijagonale. Može se vidjeti da pozicija *b3* za kraljicu *k2* ne odgovara jer je nemoguće postaviti kraljicu *k3* zbog

cijeloga prekriženog stupca *c.* Zato je potrebno vratiti se unatražno te potražiti sljedeće slobodno mjesto u stupcu *b*, a to je *b4.* Križanjem redaka, stupaca i dijagonala može se vidjeti da za kraljicu *k3* u stupcu c ostaje jedno slobodno mjesto. Tako se *k3* postavlja na c2. Kraljicu *k4* nije moguće smjestiti u stupac *d.* Budući da jedno unatražno vraćanje nije bilo dovoljno, trebat će se vratiti još jedan korak, odnosno na *k1* što će prikazivati slika 13.

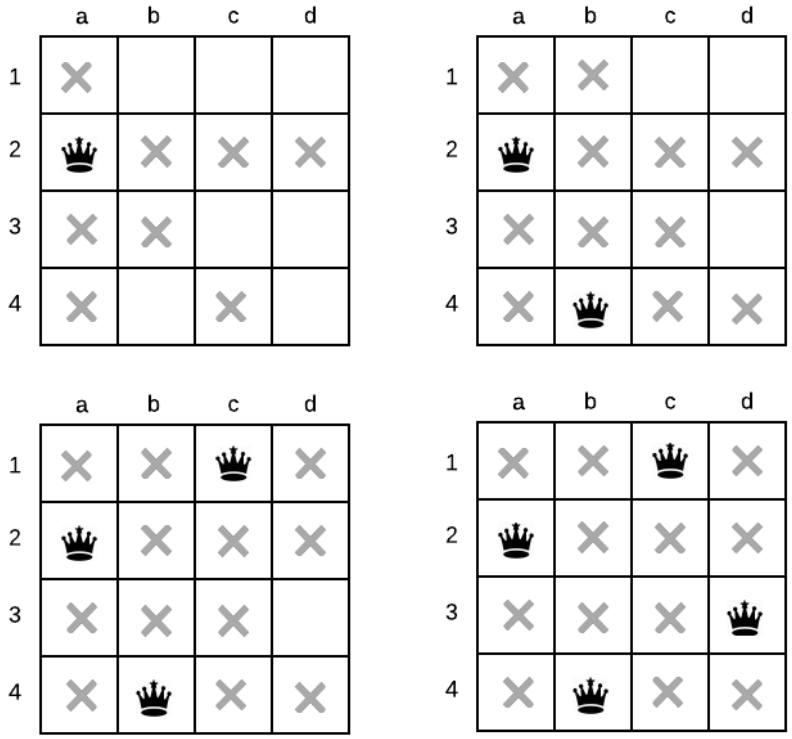
Slika 14. Koraci algoritma za *n*-kraljicu koji započinje s prvim slobodnim mjestom



Izvor: prilagođeno prema (Dunić, 2013, str. 5)

Dakle, slika 15 odnosi se na korake algoritma za *n* kraljicu nakon prvih unatražnih pretraživanja. Tako se *k1* postavlja na mjesto *a2* jer jedino *a2* dopušta postavljanje kraljice *k2* na jedino slobodno mjesto, a to je *b4*. U trećem stupcu jedino mjesto koje je slobodno *c1* gdje se kraljica k3 i postavlja. Također, jedino slobodno mjesto u četvrtom stupcu *d3* pa se tamo smješta kraljica *k4*. Tako se došlo do rješenja za koje se trebalo vratiti dva puta unatrag. Za pronalaženje svih mogućih rješenja, također, koristi se metoda *backtracking.*

Slika 15. Koraci algoritma za *n* kraljicu nakon prvih unatražnih pretraživanja



Izvor: prilagođeno prema (Dunić, 2013, str 6)

Implementacija 6 koja predstavlja rješavanje *n* kraljice sadrži nekoliko funkcija. Prva je funkcija *printSolution, a* služi za ispisivanje rješenja *n* kraljice. Od velike je važnosti *bool* funkcija *isSafe* koja provjerava može li se kraljica staviti na šahovsku ploču gledajući na stupac i redak. Funkcija *solveNQUtil* rješava problem *n* kraljice, dok funkcija *solveNQ* rješava problem *n* kraljice koristeći *backtracking*. Vremenska je složenost *n* kraljice eksponencijalna odnosno $O(n^n)$.

Implementacija 6. *N*-kraljica riješena metodom pretraživanja s vraćanjem

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#define N 8
using namespace std;

void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
```

```
    {
        for (int j = 0; j < N; j++)
            cout<<board[i][j]<<"  ";
        cout<<endl;
    }}
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
    {
        if (board[row][i])
            return false;
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j])
            return false;
    }

    for (i = row, j = col; j >= 0 && i < N; i++, j--)
    {
        if (board[i][j])
            return false;
    }
    return true;
}

bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if ( isSafe(board, i, col) )
        {
            board[i][col] = 1;
            if (solveNQUtil(board, col + 1) == true)
                return true;
            board[i][col] = 0;
        }
    }
```

```
    return false;
}

bool solveNQ()
{
    int board[N][N] = {0};
    if (solveNQUtil(board, 0) == false)
    {
        cout<<"Solution does not exist"<<endl;
        return false;
    }
    printSolution(board);
    return true;
}

int main()
{
    solveNQ();
    return 0;
}
```

Izvor: Sanfoundry, n.d.

5.3.3. Opis, implementacija i složenost 0/1 problema naprtnjače

Opis problema 0/1 naprtnjače već je opisan u jednom od prethodnih poglavlja pa tako ovo potpoglavlje donosi opće značajke, implementaciju i složenost navedenog problema koristeći poboljšanu verziju *backtrackinga*, a to je *branch and bound* metoda. Kod korištenja *backtrackinga* ako se dođe do točke u kojoj rješenje nije izvedivo, nema potrebe nastaviti s rješavanjem. Dakle, režu se grane koje neće dovesti do rješenja. Bolji način od navedenog je *branch and bound* koji će rezati grane koje neće dati bolje rješenje u odnosu na one koje postoje.

Implementacija 7 donosi riješen problem naprtnjače metodom pretraživanja s vraćanjem odnosno *brach and bound* metodom. Najprije se definiraju dvije strukture, a to su *Item* i *Node.* Struktura *Item* sadrži varijable za spremanje težinu i vrijednosti podataka, dok struktura *Node* pohranjuje informacije o odlukama. Zatim se navode tri funkcije, a to su *cmp*, *bound* i *knapsack.* Funkcija *cmp* uspoređuje sortirane predmete prema vrijednost / težina. Funkcija *bound* vraća vrijednost koja je u korijenu te

uglavnom koristi rješenje pohlepnog algoritma za pronalaženje maksimalne vrijednosti. Funkcija *knapsack* vraća maksimalnu ukupnu vrijednost koja se dobiva s kapacitetom *W*.

Implementacija 7. 0/1 problem naprtnjače riješen metodom pretraživanja s vraćanjem

```
#include <bits/stdc++.h>
using namespace std;

struct Item
{
    float weight;
    int value;
};

struct Node
{
    int level, profit, bound;
    float weight;
};

bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

int bound(Node u, int n, int W, Item arr[])
{
    if (u.weight >= W)
        return 0;

    int profit_bound = u.profit;
    int j = u.level + 1;
    int totweight = u.weight;

    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight    += arr[j].weight;
        profit_bound += arr[j].value;
```

```
        j++;
    }
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /
                                        arr[j].weight;
    return profit_bound;
}
int knapsack(int W, Item arr[], int n)
{
    sort(arr, arr + n, cmp);

    queue<Node> Q;
    Node u, v;

    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    int maxProfit = 0;
    while (!Q.empty())
    {
        u = Q.front();
        Q.pop();

        if (u.level == -1)
            v.level = 0;

        if (u.level == n-1)
            continue;

        v.level = u.level + 1;
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;

        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;

        v.bound = bound(v, n, W, arr);

        if (v.bound > maxProfit)
            Q.push(v);
```

```
            v.weight = u.weight;
            v.profit = u.profit;
            v.bound = bound(v, n, W, arr);
            if (v.bound > maxProfit)
                Q.push(v);
        }
        return maxProfit;
}

int main()
{
        int W = 6;
        Item arr[] = {{2, 1}, {3, 7}, {4, 8}
                      };
        int n = sizeof(arr) / sizeof(arr[0]);

        cout << "Maximum possible profit = "
             << knapsack(W, arr, n);

        return 0;
}
```

Izvor: Geeksforgeeks, n.d.

Vremenska je složenost *branch and bound* *O(2n)* jer se generiraju svi čvorovi i listovi kod stabla.

5.4. Metoda pohlepe

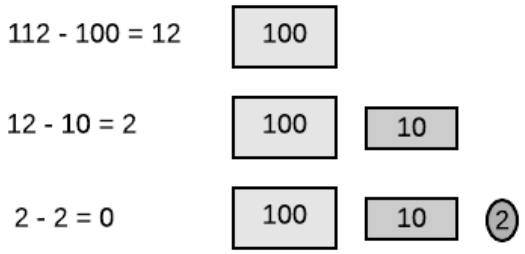
Metoda pohlepe jedna je od najjednostavnijih i najbržih metoda koja se primjenjuje samo na probleme optimizacije. Nastavak rada donosi karakteristike ove metode, Primov algoritam i kontinuirani problem naprtnjače, njihovu implementaciju i vremensku složenost.

5.4.1. Općenito o metodi pohlepe

Metoda pohlepe ili eng. *greedy algorithm* metoda je koja se može jednostavno izvesti, brzo radi, ali je i smanjena vremenska složenost. Rješenja se mogu dobiti vrlo

brzo jednostavnim koracima. U svakom koraku bira se mogućnost koja je u tom trenutku lokalno optimalna, tj. najbolja. Tim načinom dolazi se do globalno optimalnog ukupnog rješenja. Rješenje je najčešće približno. Ako trgovac kupcu treba vratiti 112 kn, novčanice koje mu stoje na raspolaganju su od 100, 50, 20 i 10 kn te kovanice od 5, 2 i 1 kn. Tom problemu trgovac može pristupiti vraćanjem jedne novčanice od 100 kn, jedne novčanice od 10 kn te jedne kovanice od 2 kn. Na taj je način trgovac, osim što je vratio točan iznos, izabrao i algoritam koji je vratio iznos s najkraćom mogućom listom novčanica i kovanica, a to su tri komada (Manger, 2013). Na slici 16 prikazan je postupak izvedbe pohlepnog algoritma za vraćanje novčanica i kovanica.

Slika 16. Prikaz najboljeg rješenja za vraćanje novčanica



Izvor: prilagođeno prema (Algoskills, n.d.)

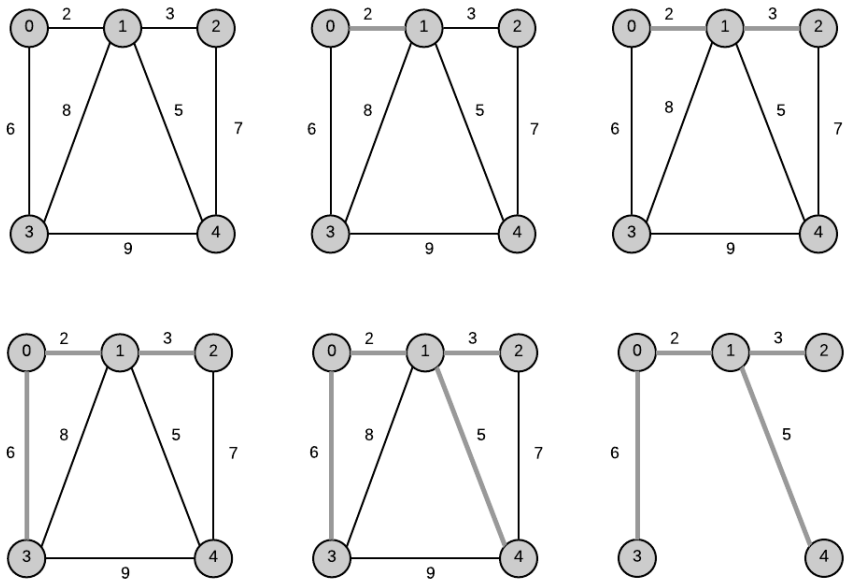
Ovaj algoritam funkcionira tako da se prvo treba izabrati najveća novčanica koja stoji na raspolaganju, uz uvjet da ne prijeđe ukupnu sumu. Zatim je treba staviti na listu za vraćanje te je oduzeti od ukupnog iznosa. Ovaj se postupak ponavlja dok se ne vrati odgovarajući iznos. Cilj je da izbor vraćenih novčanica bude minimalan. Tek će tada rješenje biti najučinkovitije (Manger, 2013).

Ova metoda neće baš uvijek dati optimalno rješenje. U nastavku se navodi primjer kod kojeg pohlepni pristup neće funkcionirati. Trgovac kupcu treba vratiti 25 kn, a na raspolaganju su mu novčanice od 10 i 20 kn te kovanice od 5 i 1 kn. Pohlepnim algoritmom može se vratiti prvo novčanica od 20 kn te pet kovanica od 1 kn. Tada bi bilo ukupno šest komada, dok je optimalno rješenje s dva komada od 20 i 5 kn (Manger, 2013).

5.4.2. Opis, implementacija i složenost Primova algoritma

Dobri primjeri algoritama koji koriste metodu pohlepe su Kruskalov i Primov algoritam. Svrha im je pronaći minimalno razapinjuće stablo u težinskom grafu. Prilikom izvedbe Kruskalovog algoritma treba pripaziti da se ne stvori ciklus. Bira grane najmanje težine ne gledajući u kakvom je odnosu s prethodno izabranim granama. Razapinjuće stablo ne mora biti povezano za vrijeme izvođenja algoritma. Kod Primova je algoritma suprotno. Ovo potpoglavlje odnosit će se upravo na Primov algoritam. Minimalno razapinjuće stablo stvara se proširivanjem početnog i najjednostavnijeg stabla koji se odnosi na proizvoljno izabrani čvor. U svakom se koraku stvorenom stablu dodaje novi čvor na drugom kraju grane najmanje težine koja izlazi iz tog stabla što se može vidjeti na slici 17 (Živković, 2010).

Slika 17. Koraci rješavanja Primova algoritma



Izvor: prilagođeno prema (Živković, 2010, str. 234)

Slijedi implementacija 8 koja se odnosi na Primov algoritam riješen uz pomoć pohlepne metode. Funkcije koje su korištene za implementaciju su *minKey, printMST* i *primMST.* Svrha je funkcije *minKey* pronaći vrh koji ima minimalnu vrijednost gledajući na skup vrhova koji još nisu uključeni u minimalno razapinjuće stablo. Zatim, funkcija *printMST* služi za ispis stvorenoga minimalnog razapinjućeg stabla koji je pohranjen u *parent [].* Posljednja je funkcija *primMST* koja služi za

konstruiranje i ispis minimalnoga razapinjućeg stabla za graf koji se prikazuje pomoću susjeda.

Implementacija 8. Primov algoritam riješen metodom pohlepe

```
#include <stdio.h>
#include <limits.h>
#include <iostream>

using namespace std;
#define V 5

int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
    if (mstSet[v] == false && key[v] < min)
    min = key[v], min_index = v;
    return min_index;
}

int printMST(int parent[], int n, int graph[V][V])
{
    cout<<"Edge   Weight\n";
    for (int i = 1; i < V; i++)
        printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

void primMST(int graph[V][V])
{
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++)
```

```
    {
        int u = minKey(key, mstSet);
        mstSet[u] = true;

        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
     }
    printMST(parent, V, graph);
}

int main()
{
    int graph[V][V] = { { 0, 2, 0, 6, 0 }, { 2, 0, 3, 8, 5 },
            { 0, 3, 0, 0, 7 }, { 6, 8, 0, 0, 9 }, { 0, 5, 7, 9, 0 }, };

    primMST(graph);
    return 0;
}
```

Izvor: Geeksforgeeks, n.d.

Vremenska je složenost Primova algoritma za navedenu implementaciju u programskom jeziku C++ jednaka $O(V^2)$.

5.4.3. Opis, implementacija i složenost kontinuiranog problema naprtnjače

Kontinuirani problem naprtnjače ili eng. *continuous knapsack problem,* također, pripada problemima optimizacije. Vrlo je sličan opisanom 0/1 problemu naprtnjače, a razlika je u stavljanju predmeta u ruksak. U ruksak je moguće staviti dio svih predmeta i na taj način maksimizirati ukupnu vrijednost naprtnjače. Tako se dobiva proporcionalan dio vrijednosti. Poimanje predmeta je isto kao kod 0/1 problema naprtnjače. Metodom pohlepe za svaki je predmet potrebno izračunati njegovu profitabilnost p_i / w_i, odnosno vrijednost po jedinici težine. Sljedeći je korak predmete sortirati silazno po profitabilnosti. Zatim se sortirane po tom redoslijedu stavljaju u naprtnjaču sve dok ima mjesta. Kod svakog stavljanja u naprtnjaču nastoji se spremiti što je moguće veći dio predmeta (Manger, 2013). Nastavak rada donosi objašnjenje

postupka rješavanja kontinuiranog problema naprtnjače. Na slici 18 vidljivi su svi parametri. Tako se može vidjeti da *n* iznosi 3, odnosno u naprtnjaču je moguće staviti tri ponuđena predmeta. Težine predmeta *w1, w2* i *w3* iznose 2, 3 i 4, dok su vrijednosti *p1, p2* i *p3 jednake* 1, 7 i 8. Kapacitet odnosno *c* težinskih jedinica je 6. U zadnjem redu izračunata je profitabilnost p_i / w_i svih predmeta Tako je profitabilnost prvog predmeta jednaka 0,5, drugog predmeta 2,33 te trećeg predmeta 2.

Slika 18. Prikaz parametara u tablici

c = 6

Predmet *i*	1	2	3
Vrijednost *p*	1	7	8
Težina *w*	2	3	4
p/w	0,5	2,33	2

Izvor: prilagođeno prema (Algoskills, n.d.)

Najprofitabilniji je drugi predmet, za njim slijedi treći predmet, a na posljednjem mjestu je prvi predmet. Oni se tim redoslijedom i sortiraju. Dakle, sortiraju se silazno po profitabilnosti što se može vidjeti na slici 19.

Slika 19. Sortiranje predmeta silazno po profitabilnosti

c = 6

Predmet *i*	1	2	3
p/w	2,33	2	0,5
Vrijednost *p*	7	8	1
Težina *w*	3	4	2

Izvor: prilagođeno prema (Algoskills, n.d.)

Na slici 20 vidljivi su koraci spremanja predmeta u naprtnjaču. Najprije se sprema prvi predmet koji je sortiran silazno po profitabilnosti koja iznosi 2,33, dok je

vrijednost jednaka 7, a težina 3. Kapacitet prije spremanja iznosio je 6, a nakon spremanja prvog predmeta iznosi 3. Idući predmet koji se treba spremiti je profitabilnosti 2, vrijednosti 6 te težine 4. Može se zaključiti da ovaj predmet ne može u cijelosti stati pa se uzima njegov dio odnosno ¾ predmeta. Nakon što su oba predmeta spremljena, kapacitet je popunjen, a vrijednost je 13.

Slika 20. Koraci spremanja predmeta u naprtnjaču

Kapacitet (c)	Ukupna vrijednost	*p/w*
6	0	0
6 - 3 = 3	7	2,33
0	7 + 2 * 3 = 13	2

Izvor: prilagođeno prema (Algoskills, n.d.)

Implementacija 9 rješava prethodno opisani problem kontinuirane naprtnjače pomoću metode pohlepe. Prvo se definira struktura za predmet u kojoj se sprema težina i vrijednost predmeta. Zatim se navode dvije funkcije, *cmp* i *fractionalKnapsack.* Svrha je funkcije *cmp* sortirati predmete prema profitabilnosti koja se dobiva podjelom vrijednosti na težinu. Funkcija *fractionalKnapsack* rješava problem metodom pohlepe.

Implementacija 9. Kontinuirani problem naprtnjače riješen metodom pohlepe

```
#include <bits/stdc++.h>
using namespace std;

struct Item
{
    int value, weight;

    Item(int value, int weight) : value(value), weight(weight)
    {}
};
```

```
bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(int W, struct Item arr[], int n)
{
    sort(arr, arr + n, cmp);

    int curWeight = 0;
    double finalvalue = 0.0;

    for (int i = 0; i < n; i++)
    {
        if (curWeight + arr[i].weight <= W)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }

        else
        {
            int remain = W - curWeight;
            finalvalue += arr[i].value * ((double) remain / arr[i].weight);
            break;
        }
    }
    return finalvalue;
}

int main()
{
    int W = 6;
    Item arr[] = {{1, 2}, {7, 3}, {8, 4}};

    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum value we can obtain = "
         << fractionalKnapsack(W, arr, n);
```

```
    return 0;
}
```

Izvor: Geeksforgeek, n.d.

U ovoj implementaciji sortiranje igra glavnu ulogu u određivanju vremenske složenosti algoritma. Prema tome može se zaključiti da je vremenska složenost linearno-logaritamska *O(nLogn).*

Na sličan se način pomoću pohlepne metode može riješiti i 0/1 problem naprtnjače, ali on nema previše smisla jer ne bi dao optimalno rješenje. Gledajući na prethodni primjer u naprtnjaču bi se mogao spremiti drugi predmet koji je najprofitabilniji s težinom 3 i vrijednošću 7. Nakon toga sljedeći najprofitabilniji predmet s težinom 4 i vrijednosti 8 ne bi mogao stati pa bi se jedino mogao dodati posljednji predmet težine 2 i vrijednosti 1. Tada bi konačna vrijednost iznosila 8, a pomoću dinamičkog programiranja dobiveno je bolje rješenje koje iznosi 9. Pomoću pohlepne metode 0/1 problem naprtnjače imao bi linearno logaritamsku vremensku složenost (Manger, 2013).

6. KOMPARACIJA METODA ZA OBLIKOVANJE ALGORITAMA

Najpoznatije metode za oblikovanje algoritama su metoda podijeli pa vladaj, metoda dinamičkog programiranja, metoda pretraživanja s vraćanjem i metoda pohlepe. Prvo što se može uočiti kod uspoređivanja ovih metoda je široki spektar primjene. Prema tome metoda podijeli pa vladaj najčešće se koristi kod problema koji se rješavaju uz pomoć rekurzije, a potproblemi su međusobno nezavisni. Metoda dinamičkog programiranja koristi se kod višedimenzionalnih optimizacijskih problema kod kojih su problemi međusobno zavisni i imaju optimalnu podstrukturu. Metodu pretraživanja s vraćanjem najbolje je iskoristiti za teške kombinatorne probleme, dok se metoda pohlepe primjenjuje na optimizacijske probleme gdje se pronalazi rješenje koje je u tom trenutku lokalno najbolje.

Metoda podijeli pa vladaj može se usporediti s dinamičkim programiranjem. Problem obiju metoda odnosi se na rastavljanje glavnog problema na potprobleme koji se rješavaju te se spajanjem njihovih rješenja dobiva rješenje glavnog problema. Temeljna je razlika što su potproblemi metode podijeli pa vladaj nezavisni, dok su potproblemi dinamičkoga programiranja zavisni, odnosno međusobno povezani te imaju optimalnu podstrukturu. Problemi koji se rješavaju metodom podijeli pa vladaj mogu rasti eksponencijalno s veličinom zadanog problema. Tako se isti potproblem može pojaviti na više mjesta u rekurziji pa isti problem treba riješiti više puta. U ovakvoj situaciji najučinkovitije je iskoristiti dinamičko programiranje koje će isti problem riješiti samo jedanput, a pritom će rješenje problema biti spremljeno. Ponovnom pojavom istog potproblema iskoristit će se sačuvano rješenje bez ponovnog rješavanja istog problema. Prema tome može se zaključiti da je složenost izvođenja algoritma dinamičkog programiranja u ovakvoj situaciji smanjena. Problemi koji su spomenuti u radu, a tiču se prethodne usporedbe su sortiranje spajanjem i Fibonaccijev niz. Dobar primjer riješen metodom podijeli pa vladaj je sortiranje spajanjem jer su potproblemi nezavisni, nema optimalne podstrukture te ponavljanja istih potproblema, a složenost je linearno-logaritamska. Primjer riješen metodom podijeli pa vladaj i metodom dinamičkog programiranja je Fibonaccijev niz. Tako su metodom podijeli pa vladaj isti potproblemi riješeni nekoliko puta te je vremenska složenost eksponencijalna što je loše za rješavanje ovog problema. Bolje rješenje donosi metoda dinamičkog programiranja i to pomoću dva pristupa, a to su odozdo prema gore i odozgo prema dolje. Oba pristupa rade identičnu stvar, ali je razlika u

načinu prijenosa rješenja potproblema. Pristupom odozgo prema dolje problem se rastavlja na potprobleme koji se rješavaju, a rješenja se pamte pomoću strukture podataka u kojoj se čuvaju rješenja, a navedeno se naziva memoizacija. Na taj način mogu se ponovno iskoristiti već riješeni potproblemi. Ovaj pristup još i koristi rekurziju. Pristupom odozdo prema gore prvo se rješavaju jednostavniji potproblemi, a njihova rješenja koriste se za rješavanje složenih problema dok se ne dođe do konačnog rješenja. Tako je prednost pristupa odozgo prema dolje jednostavan kod, dok se manom može smatrati sporo izvršavanje algoritma zbog korištenja rekurzije. Prednost je pristupa odozdo prema gore učinkovitost gledajući na prostornu složenost, a mane se odnose na komplicirano zapisivanje koda te teško određivanje potproblema za rješavanje složenih problema. Prema tome Fibonaccijev niz riješen pristupom odozgo prema dolje koristi memoizaciju kojom se isti potproblemi rješavaju što nije bio slučaj kod metode podijeli pa vladaj. Ovaj pristup ima linearnu vremensku složenost jer se ni jedan čvor ne poziva više od jednom, dok je prostorna složenost, također, linearna zbog rekurzije. Pristup koji je još bolji za rješavanje ovog problema je pristup odozdo prema gore jer je prostorna složenost konstantna. Prvo su se riješili najjednostavniji problemi čija su rješenja poslije poslužila za rješavanje složenijih potproblema. Kod ovog je pristupa vremenska složenost ista kao i kod prethodnog pristupa, ali prostorna je složenost konstantna odnosno bolja zbog bržeg pristupa memoriji jer se izbjegavaju pozivi funkcija. Tako se može zaključiti da je najbolje za problem Fibonaccijeva niza iskoristiti pristup odozdo prema gore.

Metoda dinamičkog programiranja, metoda pretraživanja s vraćanjem i metoda pohlepe mogu se, također, međusobno usporediti. Kao što je već navedeno, metoda dinamičkog programiranja rješava optimizacijske probleme. Tako je cilj ove metode pronaći rješenje s optimalnom vrijednošću. Prednost je ove metode brže dolaženje do rješenja u odnosu na ostale metode pa je i smanjena složenost izvođenja algoritma. Metoda pretraživanja s vraćanjem rješava teške kombinatorne probleme. Temelji se na pretraživanju svih mogućih rješenja pomoću stabla. Tada algoritam stvara kandidate, ali i izbacuje one koji ne mogu stvoriti ispravno rješenje. Na taj se način smanjuje broj mogućih rješenja pa se brže može doći do rješenja. Jedan od problema koji je riješen ovom metodom je problem *n* kraljice, a vremenska je složenost eksponencijalna. Postoji i optimizacijska varijanta ove metode, a naziva se *branch and bound.* Standardni *backtracking* u stablu rješenja je rezao grane koje nisu vodile do rješenja, dok *branch and bound* metoda reže one grane koje ne vode

do boljeg rješenja u odnosu na one koje postoje. Problemi koji se rješavaju *branch and bound* metodom uglavnom su eksponencijalni i ova metoda relativno brzo uspijeva riješiti takve probleme. Ova metoda uglavnom se primjenjuje na one optimizacijske probleme koji su neuspješno riješeni metodom dinamičkog programiranja ili metodom pohlepe. U odnosu na navedene dvije metode *branch and bound* je relativno spor, ali ako se pažljivo koristi može dovesti do algoritama koji u prosjeku rade brzo. Metoda pohlepe primjenjuje se samo na optimizacijske probleme, a jedna je od najjednostavnijih i najbržih metoda za navedenu skupinu problema. Prednost je i smanjena vremenska složenost. Rješenja se dobivaju jednostavnim koracima gdje se u svakom od koraka bira mogućnost koja u trenutku lokalno optimalna. Problem koji je naveden u radu, a odnosi se na metodu pohlepe je Primov algoritam. Njegova je vremenska složenost kvadratna. Problem koji se riješio navedenim metodama je problem naprtnjače. U radu se navode dvije varijante naprtnjače. Prva se odnosi na problem 0/1 naprtnjače, dok se drugi odnosi na kontinuirani problem naprtnjače. Cilj je problema 0/1 naprtnjače postaviti cjelobrojne predmete u naprtnjaču uz uvjet da ukupna vrijednost bude maksimalna i da ukupna težina ne prijeđe kapacitet naprtnjače. Gledajući na primjer riješen u radu, metodom dinamičkog programiranja svaki se problem riješio pojedinačno, a zatim su se rješenja spojila kako bi se dobilo optimalno rješenje. Dakle, jednom se riješio svaki potproblem, njihova rješenja su se pohranila u memoriji pa se pojavom istog potproblema uzelo spremljeno rješenje. Ova metoda daje optimalno, tj. najbolje rješenje, a složenost je linearna. Problem 0/1 naprtnjače mogao se riješiti pomoću pohlepne metode, ali on nema previše smisla jer ne daje optimalno rješenje, a i složenost je linearno-logaritamska što je iznimno loše jer postoji bolje rješenje. Metodom pohlepe najbolje je riješiti kontinuirani problem naprtnjače gdje je moguće staviti dio svih predmeta i na taj način maksimizirati ukupnu vrijednost naprtnjače. Budući da sortiranje ima glavnu ulogu, vremenska je složenost kontinuiranog problema naprtnjače linearno-logaritamska. Optimizacijskom metodom *backtrackinga* odnosno metodom *branch and bound* može se, također, riješiti problem 0/1 naprtnjače. U ovom slučaju rješenje je optimalno, a složenost je linearna isto kao kod metode dinamičkog programiranja. No, inače se *branch and bound* primjenjuje samo onda kada pohlepna metoda i dinamičko programiranje ne uspijevaju. *Branch and bound* zna dovesti do eksponencijalne vremenske složenosti u najgorem slučaju, ali ako se pažljivo primjeni može dovesti do algoritma koji relativno brzo radi.

7. ZAKLJUČAK

Na kraju ovog rada može se zaključiti da algoritmi igraju veliku ulogu, kako u svakodnevnom životu, tako i u informacijsko komunikacijskoj tehnologiji i drugim granama znanosti. Ne gledajući na područja primjene, vrlo su praktični pa njihovo poznavanje može biti od velike koristi za svakodnevno rješavanje različitih zadataka koji se nude. Gledajući na algoritme u svijetu programiranja, njihovo oblikovanje se može smatrati jedinim od kreativnijih poslova. Zato se od svakog dizajnera koji se opušta u oblikovanje, uz kreativnost, još i očekuje upornost, volja i znanje. Ne postoji opće pravilo kako riješiti sve probleme, ali postoje standardne metode za oblikovanje algoritama koje mogu pomoći pri rješavanju problema. Te metode detaljno su objašnjene u radu, a pogodne su za rješavanje širokog spektra različitih problema. Tako metoda podijeli pa vladaj rješava probleme koji mogu iskoristiti rekurziju. Glavni problem dijeli se na manje potprobleme koji se rješavaju rekurzivno te se njihova rješenja spajaju u jedno rješenje. Kod ove metode problemi su međusobno nezavisni. Metoda dinamičkog programiranja može se primijeniti kod višedimenzionalnih optimizacijskih problema. Problemi su međusobno zavisni te imaju optimalnu podstrukturu. Specifičnost ove metode je spremanje rezultata potproblema te njegovo ponovno korištenje ako se u daljnjem rješavanju naiđe na isti potproblem. Metoda pretraživanja s vraćanjem koristi se za teške kombinatorne probleme, ali postoji i optimizacijska verzija koja se naziva *branch and bound*. Kod standardnog *backtrackinga* režu se grane koje ne vode do rješenja, dok *branch and bound* metoda reže one rane koje ne vode do boljeg rješenja u odnosu na one koje postoje. Posljednja je metoda pohlepe koja služi samo za optimizacijske probleme, a pronalazi rješenje koje je u tom trenutku lokalno najbolje.

Određeni problem može se riješiti nekim od metoda za oblikovanje algoritama, međutim nije pametno osloniti se samo na jednu metodu, već pokušati riješiti problem i ostalim metodama koje će možda rezultirati učinkovitijim rješenjima. Tako je Fibonaccijev niz, koji je spomenut u radu, prvobitno riješen metodom podijeli pa vladaj koja je rezultirala eksponencijalnom vremenskom složenošću. Korištenjem metode dinamičkog programiranja odnosno pristupa odozgo prema dolje došlo se do boljeg rješenja gdje je vremenska složenost bila linearna, a prostorna složenost, također, linearna. Treći način, pomoću pristupa odozdo prema gore imao je bolju prostornu složenost od prethodno spomenutog pristupa, a to je konstantna složenost.

Tako se pristup odozdo prema gore smatra najučinkovitijim pristupom za rješavanje Fibonaccijeva niza. Prema tome može se još i zaključiti da analiza složenosti ima ključnu ulogu u biranju najučinkovitijeg algoritma. Pomoću nje moguće je okvirno predvidjeti koja je metoda najbolja za pojedinu situaciju, ali ipak analiza stvarnog vremena to može najbolje pokazati.

8. POPIS LITERATURE

Knjige:

Barnett, G., Del Tongo, L. (2008). *Data Structures and Algorithms: Annotated Reference with Examples.* DotNetSlackers.

Divjak, B., Lovrenčić, A. (2005). *Diskretna matematika s teorijom grafova.* Varaždin: TIVA-FOI.

Grundler, D., Blagojević, L. (2005). *Informatika: udžbenik za 1. razred gimnazije.* Zagreb: Školska knjiga.

Hoško, T., Slamić, M. (2009). *Strukture podataka i algoritmi - Priručnik.* Zagreb: Algebra d.o.o.

Manger, R. (2013). *Strukture podataka i algoritmi Nastavni materijal*. Četvrto izdanje. Zagreb: Element d.o.o.

Živković, D. (2010). *Uvod u algoritme i strukture podataka.* Beograd: Univerzitet Singidunum.

Internetski izvori:

Adamchik, V. (2014). *Solving Divide-and-Conquer Recurrences.* http://www.cs.cmu.edu/afs/cs/academic/class/15451-s14/LectureNotes/lecture1_supplement.pdf (pristupljeno 15. kolovoza 2018).

Algoskills. (n.d.). *Greedy algorithm on Minimum Coin Change Problem.* https://www.algoskills.com/g_coinchange.php *(*pristupljeno 27. kolovoza 2018).

Algoskills. (n.d.). *Fractional Knapsack Problem.* https://www.algoskills.com/f_knapsack.php (pristupljeno 8. rujna 2018).

Cormen, T., Balkcom D. (n.d.) *Divide and conquer algorithms.* https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms (pristupljeno 15. kolovoza 2018).

Geeksforgeeks. (n.d.). *Implementation of 0/1 Knapsack using Branch and Bound.* https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/ (pristupljeno 11. rujna 2018).

Geeksforgeeks. (n.d.). *Prim's Minimum Spanning Tree (MST) | Greedy Algo-5* https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/ (pristupljeno 25. kolovoza 2018).

Geeksforgeeks. (n.d.). *Program for Fibonacci numbers.* https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/ (pristupljeno 17. kolovoza 2018).

Hom, E. J. (2013). *What is the Fibonacci Sequence?.* https://www.livescience.com/37470-fibonacci-sequence.html (pristupljeno: 17. kolovoza 2018).

Iacono, A. (2017). *Backtracking explained.* https://medium.com/@andreaiacono/backtracking-explained-7450d6ef9e1a (pristupljeno: 20. kolovoza 2018).

Joshi, V. (2017). *Less Repetition, More Dynamic Programming.* https://medium.com/basecs/less-repetition-more-dynamic-programming-43d29830a630 (pristupljeno 18. kolovoza 2018).

Kumar, N. (2016). *Tabulation vs Memoization*. https://www.geeksforgeeks.org/tabulation-vs-memoizatation/ (pristupljeno 18. kolovoza 2018).

Leighton, T. (1996). *Notes on Better Master Theorems for Divide-and-Conquer. Recurrences.* http://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf (pristupljeno 15. kolovoza 2018).

Mishra, N. (n.d.). *Difference Between Recursion and Iteration* https://www.thecrazyprogrammer.com/2017/05/difference-between-recursion-and-iteration.html (pristupljeno 14. kolovoza 2018).

Moore, K., Rizzante, C., Khim, J. (2018). *Divide and Conquer.* https://brilliant.org/wiki/divide-and-conquer/ (pristupljeno 15. kolovoza 2018).

Mullin, M. (n.d.) *Fibonacci, spirals and human movement—Part I.* https://www.mjmatc.com/positional-integration/2017/8/27/fibonacci-spirals-and-human-movement (pristupljeno 17. kolovoza 2018).

Nayal, C. (2016). *Merge Sort.* https://www.geeksforgeeks.org/merge-sort/ (pristupljeno 16. kolovoza 2018).

Perišić, L. (2017). *Ada Lovelace – čarobnica brojeva i proročica računalne ere*. https://www.voxfeminae.net/strasne-zene/item/9023-ada-lovelace-carobnica-brojeva-i-prorocica-racunalne-ere (pristupljeno 5. kolovoza 2018.).

Sandam, V. (2015). *Dynamic Programming – Introduction and Fibonacci Numbers.* http://theoryofprogramming.com/2015/03/02/dynamic-programming-introduction-and-fibonacci-numbers/ (pristupljeno 17. kolovoza 2018).

Sanfoundry. (n.d.). *0 1 Knapsack Problem – Dynamic Programming Solutions.* https://www.sanfoundry.com/dynamic-programming-solutions-0-1-knapsack-problem/ (pristupljeno 9. rujna 2018).

Sanfoundry. (n.d.). *C++ Program to Implement Merge Sort.* https://www.sanfoundry.com/cpp-program-implement-merge-sort/ (pristupljeno 15. kolovoza 2018).

Sanfoundry. (n.d.). *C++ Program to Solve N-Queen Problem.* https://www.sanfoundry.com/cpp-program-solve-n-queen-problem/ (pristupljeno 23. kolovoza 2018).

Soldaat, X. (2013). *Tutorial: Stack in robotic.* http://botbench.com/blog/2013/01/20/tutorial-stacks-in-robotc/ (pristupljeno 14. kolovoza 2018).

Stojanović, B. (n.d.). *Algoritamske strategije.* https://imi.pmf.kg.ac.rs/index-old.php?option=com_docman&task=doc_view&gid=452&Itemid=198 (pristupljeno 18. kolovoza 2018).

Suthers, D. (2015). *Multithreaded Algorithms.* https://www2.hawaii.edu/~janst/311/Notes/Topic-22.html (pristupljeno 17. kolovoza 2018).

Trekhleb, O. (n.d.). *Dynamic Programming vs Divide-and-Conquer.* https://itnext.io/dynamic-programming-vs-divide-and-conquer-2fea680becbe (pristupljeno 17. kolovoza 2018).

Prezentacije s predavanja:

Carić, T., Ivanjko, E. (2013). *Složenost algoritama i rekurzije*. Algoritmi i strukture. Sveučilište u Zagrebu, Fakultet prometnih znanosti, Zagreb.

Orehovački, T. (2016). *Analiza složenosti algoritma*. Strukture podataka i algoritmi. Sveučilište Jurja Dobrile u Puli, Fakultet informatike, Pula.

Orehovački, T. (2016). *Metoda podijeli pa vladaj*. Strukture podataka i algoritmi. Sveučilište Jurja Dobrile u Puli, Fakultet informatike, Pula.

Kvalifikacijski radovi

Bobičić, D. (2014). *Eratostenovo sito i Euklidov algoritam* (Diplomski rad). Prirodoslovno-matematički fakultet, Osijek, Sveučilište J.J. Strossmayera u Osijeku. http://www.mathos.unios.hr/~mdjumic/uploads/diplomski/BOB04.pdf (pristupljeno 9. kolovoza 2018).

Dunić, S. (2013). *Problem osam kraljica* (Završni rad). Prirodoslovno-matematički fakultet, Osijek, Sveučilište J.J. Strossmayera u Osijeku. http://mapmf.pmfst.unist.hr/~ani/radovi/zavrsni/Stefan-Dunic-zavrsni.pdf (pristupljeno 23. kolovoza 2018).

Tkalčec, D. (2017). *Dinamičko programiranje* (Završni rad). Fakultet elektrotehnike, računarstva i informacijskih tehnologija, Osijek, Sveučilište J.J. Strossmayera u Osijeku. https://repozitorij.etfos.hr/islandora/object/etfos:1549/preview (pristupljeno 17. kolovoza 2018).

9. POPIS SLIKA, IMPLEMENTACIJA, TABLICA I GRAFIKONA

Popis slika:

Popis implementacija:

Popis tablica:

Popis grafikona: