Implementacija algoritama u programskom jeziku Scala

Brizanac, Ivan

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

IVAN BRIZANAC

IMPLEMENTACIJA ALGORITAMA U PROGRAMSKOM JEZIKU SCALA

Završni rad

Pula, 2020.

Sveučilište Jurja Dobrile u Puli
Fakultet informatike u Puli

IVAN BRIZANAC

IMPLEMENTACIJA ALGORITAMA U PROGRAMSKOM JEZIKU SCALA

Završni rad

JMBAG: 0303068950, redovan student

Studijski smjer: Informatika

Predmet: Strukture podataka i algoritmi

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: Izv. prof. dr. sc. Tihomir Orehovački

Pula, 2020.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Ivan Brizanac, 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, _____, _____ godine

IZJAVA
o korištenju autorskog djela

Ja, Ivan Brizanac dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom Implementacija algoritama u programskom jeziku Scala 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, _____(datum)

Potpis

_____

Implementacija algoritama u programskom jeziku Scala

Ivan Brizanac

Sažetak: U ovom radu objašnjeni su pojmovi algoritama, te je pobliže predočen programski jezik Scala. Definirane su najčešće tehnike oblikovanja algoritama, te prikazani su primjeri radi lakšeg poimanja. Također navedeni su algoritmi sortiranja i pretraživanja koji su elementarni kada se priča o računalnoj znanosti. Osim toga ponuđena su neka algoritamska rješenja u programskom jeziku Scala za neke od uobičajenih problema koji se javljaju u informatici.

Ključne riječi: Scala, oblikovanje algoritama, konjićev skok, algoritmi sortiranja i pretraživanja

Implementation of algorithms in Scala programming language

Abstract: This bachelor thesis explains concepts of algorithms, and presents Scala programming language in more detail. The most common algorithm design techniques are defined, with presented examples for ease of understanding. Sorting and searching algorithms are also mentioned which are elementary when talking about computer science. In addition, some algorithmic solutions in the Scala programming language are offered for some of the common problems that occur in informatics.

Keywords: Scala, algorithm design, knight's tour, sorting and searching algorithms

Sadržaj

1. Uvod

Cilj ovog završnog rada ponuditi je bliži uvid u noviji i nešto nepoznatiji programski jezik Scala, te prikazati implementacije nekih najčešće spomenutih algoritama. A što bi bili algoritmi? Algoritmi su skup naredbi čijim izvršavanjem se rješava neki problem, mada je nemoguće uvijek predvidjeti točno ponašanje pojedinih algoritama. Po pitanju jezika, sam jezik odlikuje njegova skalabilnost, što je kamen temeljac njegove popularizacije zadnjih godina. Uz konciznu sintaksu, ideja jezika je preuzeti najbolje osobine već viđenog i nadopuniti mane iz Jave, programskoga jezika na kojem se ovaj i temelji. Od poznatijih problema pokušano je predočiti primjere za svaku od tehnika oblikovanja algoritama poput Dijkstrinog algoritma gdje je potrebno pronaći najkraći put među čvorovima na grafu, zatim tzv. „konjićev skok“, gdje šahovski skakač obilazi pločom bez da se vraća na polja koja je posjetio (eng. Knight's tour), problem N kraljica, itd. Uz fundamentalne probleme, prikazana su popularna algoritamska rješenja za sortiranje i pretraživanje u okruženju Scale. Budući kako su podatci najbitnija stvar u svezi računala jer pri uporabi bilo kakvog programa ili softvera, računalo pohranjuje podatke stoga možemo shvatiti odakle potreba za spomenutim. Sortiranje je korisno jer računala raspolažu sa ogromnim količinama podataka, pa je nužno da su podatci sortirani, no osim toga, sortiranja mogu poslužiti i pri rješavanju mnogih kompleksnih problema. Također, sortirani podatci idu na ruku prilikom pretraživanja jer kao i u stvarnom životu, puno je lakše pretraživati po nečemu što je smisleno sortirano, nego nečemu bez pravila. Predočeni su neki korisni oblici sortiranja, poput mjehuričastoga sortiranja, sortiranja izborom, umetanjem, spajanjem i brzog sortiranja, a bitno je i spomenuti kako je za svaki zasebno predočena i njegova vremenska složenost. Za algoritme pretraživanja prikazano je linearno i binarno pretraživanje, a valja napomenuti kako svi primjeri sortiranja i pretraživanja koji su navedeni, osim što su konceptualno objašnjeni prikazana su i njihova nešto praktičnija rješenja. Scala je fleksibilan i moćan jezik opće namjene, izgrađen na Java Runtime Environment-u i može se lako ugraditi u postojeće Java aplikacije. Jezik nudi bogat niz mehanizama koji rješavaju mnoge nedostatke Java jezika, a zanimljiv je spoj funkcionalnih i objektno orijentiranih paradigmi programiranja. Taj spoj stilova programiranja ima komplementarne prednosti kada se priča o

skalabilnosti. Dok funkcionalni dio čini laganim i brzim izgradnju zanimljivih stvari iz jednostavnih dijelova, objektno-orijentirani dio olakšava strukturiranje ogromnih sustava i prilagodbu novim zahtjevima. Konačno, kako ovaj završni rad većinom obrađuje teme iz područja šaha, suludo je ne spomenuti da je jedna od najpoznatijih svjetskih platformi otvorenoga koda (eng. open-source) pod nazivom Lichess napisana upravo u Scali.

2. Što je Scala?

Scala je moderni programski jezik koji objedinjuje svojstva objektno-orijentiranoga i funkcionalnoga jezika. Dizajniran je kako bi integrirao s aplikacijama koje pokreću suvremeni virtualni strojevi, gdje je idejno fokusiran na „Java virtual machine“, a sam jezik 2003. razvila je grupa ljudi koju je predvodio Martin Odersky na institutu EPFL u Švicarskoj. Ime Scala preuzeto je od fraze Scalable Language radi njegove same skalabilnosti, tj. radi sposobnosti jezika da raste sa zahtjevima njegovih korisnika. „Scala se može koristiti od pisanja kraćih programskih skripti, do izrade većih sustava što najbolje svjedoči o širokoj primjenjivosti“ (Odersky, 2008, str. 3). To naravno ne znači kako ostali moderni programski jezici nisu skalabilni poput Pyhthona, Rubyja, Jave, itd..., dapače, no svi oni pri uporabi imaju određene restrikcije koje Scala nadilazi. Scala je statički pisani jezik inspiriran programskim jezikom Java, samim time ne čudi da je jezik interoperabilan s Javom jer postojeći kod zapisan u Javi može se pokrenuti u Scali, kao i vice versa. Ipak, dok u usporedbi s Javom brzina izvođenja u Scali je gotovo jednaka, Odersky tvrdi kako je upola manje koda potrebno za identičan u Javi. Unatoč tome što Scala izgleda kao interpretirani zapravo je kompajlirani jezik, gdje se sav uneseni kod komplajlira i pokreće unutar JVM, što se navodi kao i prednost jer se greška (eng. error) hvata kompajlerom, a ne za vrijeme izvođenja (eng. run-time). U usporedbi sa drugim jezicima instaliranje Scale jest malo neobično iz razloga što se postavlja zasebno za svaki projekt, a ne na razini sustava. Popularno rješenje za Scalu je IntelliJ IDEA okruženje (eng. integrated development environment), no valja napomenuti kako postoje i drugi načini pokretanja Scale, tipa preko Scala interpretera u naredbenom retku (eng. command line), pa čak i preko internet preglednika.

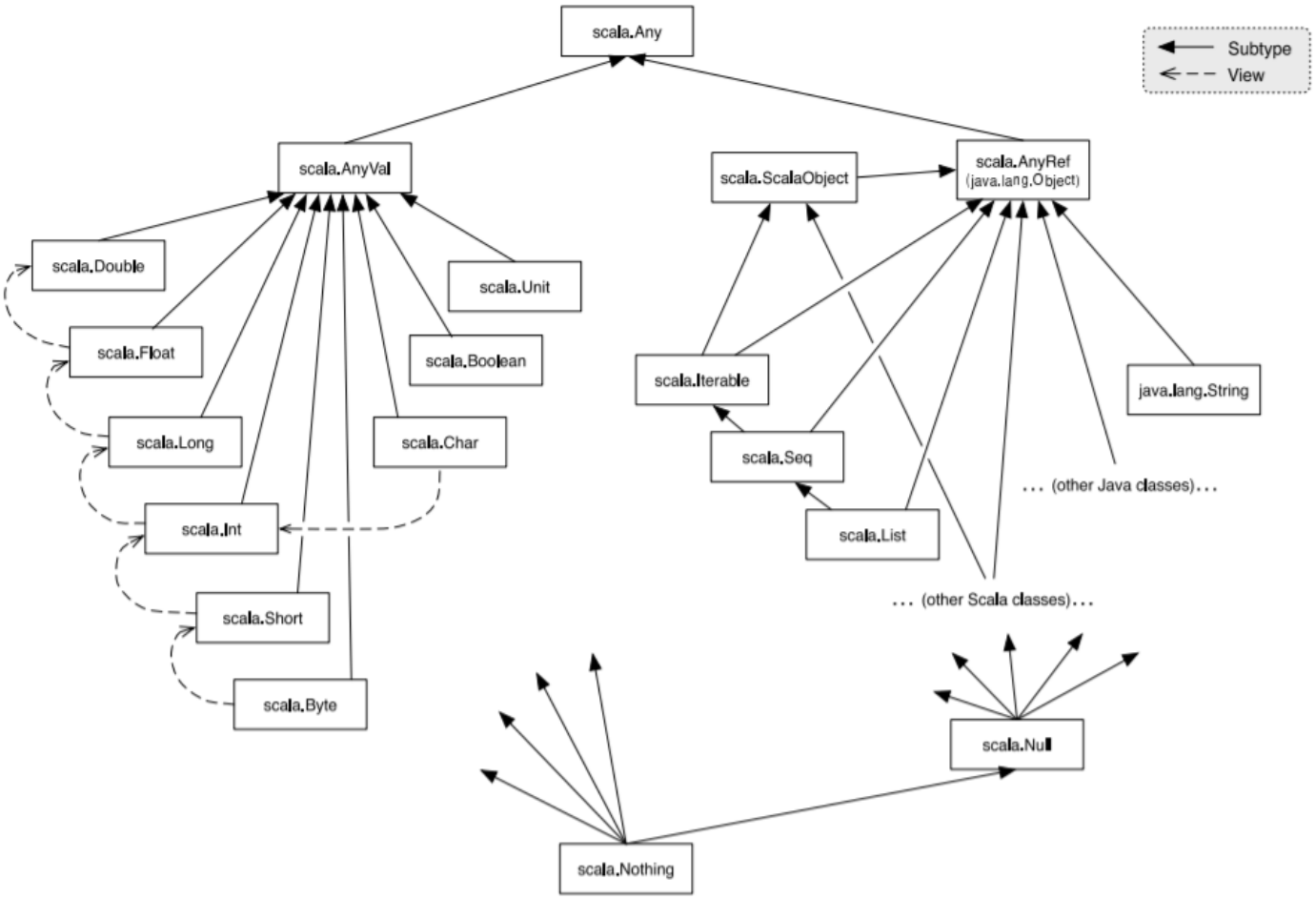
2.1. Svojstva Scale

Scala ima dvije vrste varijabli „val“ i „var“. Varijable tipa „val“ su konačne i nepromjenjive, tj. jednom kada su inicijalizirane vrijednost im se ne može ponovno dodijeliti, dok je „var“ upravo suprotan i vrijednost može biti kasnije mijenjana. Na slici 1. može se vidjeti „kostur“ Scale, tj. glavna metoda koja prikazuje kako ispisati famoznu krilaticu „Hello, world“, dok na slici 2. primjetna je hijerarhija svih tipova podataka na koje se nailazi u Scali. Budući kako se Scala zasniva na Javi, nije ni čudno kako neki tipovi koegzistiraju sa Javinim.

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Slika 1. „Hello World“ (Odersky, 2014)

Jedna od primarnih odlika Scale jest *lazy evaluation*, odnosno „lijena procjena“. Radi se o tome da varijablama koje su deklarirane na taj način se ne pristupa ili bilo kakva naredba se ne izvodi sve dok ju programer ne pozove. Jednostavnije rečeno, izvodi se na zahtjev, što štedi memoriju i resurse. Od struktura podataka što se rabe u Scali, uz polja i liste nešto nepoznatiji su „trait“ (skup metoda koje daju osobnost klasama) i „torke“ (eng. „tuples“), što su također nepromjenjjive varijable poput „val“, samo što sadržavaju objekte različitih tipova podataka. Scalu odlikuje i raznolikost tipova podataka jer tipovi mogu biti i strogo definirani, ali i nedefinirani.



Slika 2. Prikaz hijerarhije tipova podataka u Scali (Sharma, 2018)

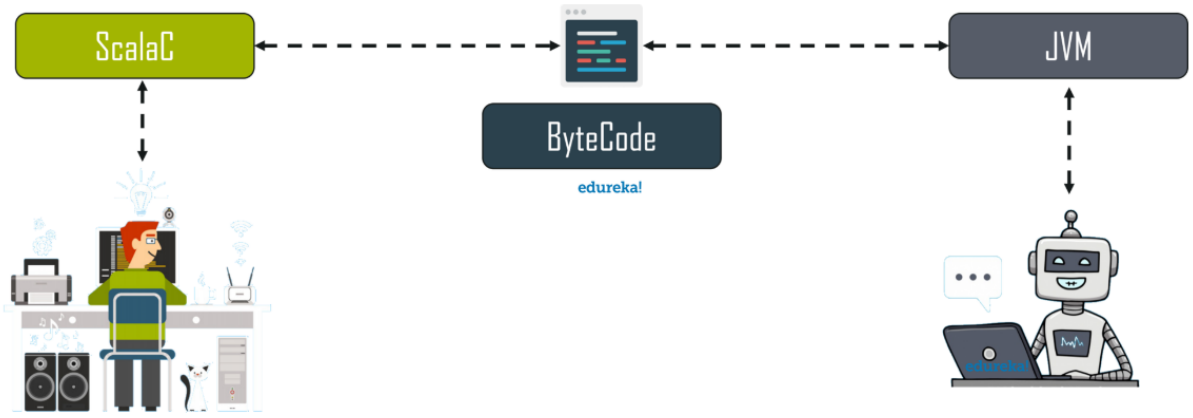
Među poznatijim programskim okvirima (eng. framework) okruženjima koji podržavaju Scalu izdvajaju se:

- Akka: besplatan alat otvorenoga koda namjenjen za izgradnju distribuiranih i otpornih na grešku aplikacija na JVM-u
- Spark: dizajniran za obradu i procesuiranje velikih količina podataka i općenito big data tehnologiju
- Play: framework za izradu web aplikacija
- Scalding: ekstenzija za Cascading koja omogućuje izradu posebne vrste aplikacija u Scali
- Neo4j: sustav za grafičko upravljanje bazama podataka

Slika 3. Najpopularnija framework rješenja u Scali (Izvor: edureka.org)

2.2. Scala i big data

Scala ima znatnu ulogu kada je riječ o tehnologiji velikih podataka (eng. big data) radi uočenih prednosti jezika kada se radi o ogromnim količinama podataka. Iako jest po naravi jezik koji se zasniva na kompajleru, glavna prednost Scale je Java Virtual Machine. Programski kod se prvo kompajlira, te se generira „byte code“ (zapis koda namijenjen pokretanju na virtualnim strojevima) koji se na kraju šalje na JVM kako bi se dobio ispis (eng. output).



Slika 4. Prikaz procesa pokretanja koda Scale (Izvor: edureka.org)

Postoji nekoliko razloga zbog kojih Scala zauzima superiorni položaj nad ostalim jezicima kad u pitanje dolazi „big data“, a neki koji se navode najčešće su:

1. Scala je sposobna raditi s podatcima koji se pohranjuju na distribuirani način, u stanju je pristupiti svim dostupnim podatcima, te podržava paralelnu obradu podataka.
2. Podržava nepromjenjive tipove podataka i ima podršku za funkcije višeg reda.
3. Scala se smatra nadograđenom verzijom Jave koja je osmišljena radi uklanjanja nepotrebnoga koda. Podržava brojne biblioteke i API sučelja što programerima omogućuje manje „downtime-a“ (vremena dok je sustav neaktivan, odnosno izvan uporabe).

Dakle rezimirano, kao i svaki programski jezik moguće je interpretirati prednosti i mane Scale, a neke od najočitijih su:

Prednosti:

- Objedinjuje objektno orijentirano programiranje sa funkcionalnim: kao OOP jezik sve vrijednosti su objekti opisani najčešće klasama, dok odlika funkcionalnog jest što svaka funkcija iskazuje neku vrijednost
- Koncizan i strogo definiran jezik
- Interoperabilnost sa Javom jer se pokreće na JVM, stoga može rabiti klase i biblioteke iz Jave, no opet u mogućnosti ispravljati nedostatke iz Jave

Mane:

- Zahtjevniji jezik za učenje i prilagodbu, pogotovo za početnike
- Slaba programska podrška u usporedbi s Javom, točnije rečeno slabije upotrebljiv IDE alat
- Zajednica ograničena brojem ljudi i resursima jer je jezik relativno nov

3. Algoritmi

Što bi bili algoritmi? Grubo govoreći, to je slijed nekih naredbi koje shodno svome izvršavanju dolaze do rješenja nekog jasno formuliranog problema. Jasno formuliran problem je nedvosmislen i precizan, radi izbjegavanja pogrešne interpretacije.

Dobitnik Turingove nagrade i čovjek kojeg razmatraju za Nobelovu nagradu iz područja računalnih znanosti Donald Knuth jednom prilikom definirao je algoritme kao: „skup pravila koja specificiraju redoslijed i vrstu aritmetičkih operacija koja se rabe na specificiranom setu podataka. Efektivna metoda za nadilaženje problema rabeći konačan slijed naredbi.“

„Dva bitna pitanja svezi kvalitete algoritma su radi li algoritam ispravno i koliko vremena mu je potrebno za izvršavanje, jer algoritam koji ispravan rezultat vraća u samo pola slučajeva, ili mu je potrebno 1000 godina za izvršavanje ne zadovoljava uvjete“ (Kulikov i Pevzner, 2018, str. 1).

„Riječ „algoritam“ potječe od perzijskog matematičara, astronoma i geografa iz devetog stoljeća al-Khwarizmija (puno ime u engleskoj transkripciji glasi Muhammad ibn Musa al-Khwarizmi). Napisao je knjigu u kojoj je opisao postupke za računanje u indijskom brojevnom sustavu. Original na arapskom jeziku nije sačuvan, a latinski prijevod proširio se Europom pod naslovom „Algoritmi de numero Indorum“ (Al-Khwarizmi o indijskim brojevima). Prema tom naslovu postupke za sustavno rješavanje problema danas nazivamo algoritmima“ (Krčadinac, 2016/2017, p. 4).

4. Tehnike oblikovanja algoritama

Kroz prošlost informatički stručnjaci su zamijetili kako razni algoritmi, unatoč tome što ne rješavaju slične probleme, koriste slične ideje pri rješavanju, stoga postoji nekoliko tehnika koje se najčešće rabe u postupku traženja algoritama za rješavanje novih problema. Treba naglasiti kako ne mora značiti da će neka tehnika dati uvijek točno rješenje, ali radi se o strategijama koje su se najčešće pokazale uspješnima, pa se iz toga razloga najčešće i rabe.

4.1 Metoda iscrpne pretrage

Metoda iscrpne pretrage, ili strogim prijevodom surove snage (eng. brute force) je vrsta algoritama gdje se pregledava svaka moguća alternativa dok se ne pronađe ispravna. Drugim riječima, stil programiranja koji ne sadržava nikakve „prečace“ pomoću kojih bi unaprijedio performanse, nego se oslanja na surovu snagu računala koje prolazi kroz sve mogućnosti dok ne dođe do rješenja. Ovo su najjednostavniji algoritmi za oblikovati, i ponekad služe za rješavanje nekih jednostavnih praktičnih problema. Ipak, valja naglasiti kako su algoritmi ove vrste kod mnogih slučajeva prespori da bi bili praktični i kako se savjetuje izbjegavanje uporabe ovih algoritama.

Jedan od primjera ovog principa su algoritmi za podudaranje znakovnih nizova (eng. string matching). Tu spadaju problemski zadatci koji iz jednoga ili više stringova traže identične uzorke unutar većih stringova, a u Scali je ponuđeno elegantno rješenje za ovu vrstu problema. Ubraja se u algoritme iscrpne pretrage iz razloga što provjerava svaki karakter zasebno i uspoređuje ga sa uzorkom.

```
def stringMatching(txt: String, uzorak: String): Unit = {

  txt.indices.filter(i => txt.substring(i).startsWith(uzorak))

  /*
  Funkcija za string Matching, gdje se traže poklapanja među stringovima

  Uz pomoć startsWith metode određuje se ima li string slijed
  traženih znakova gdje vraća true ili false za rezultat.
   */
}
```

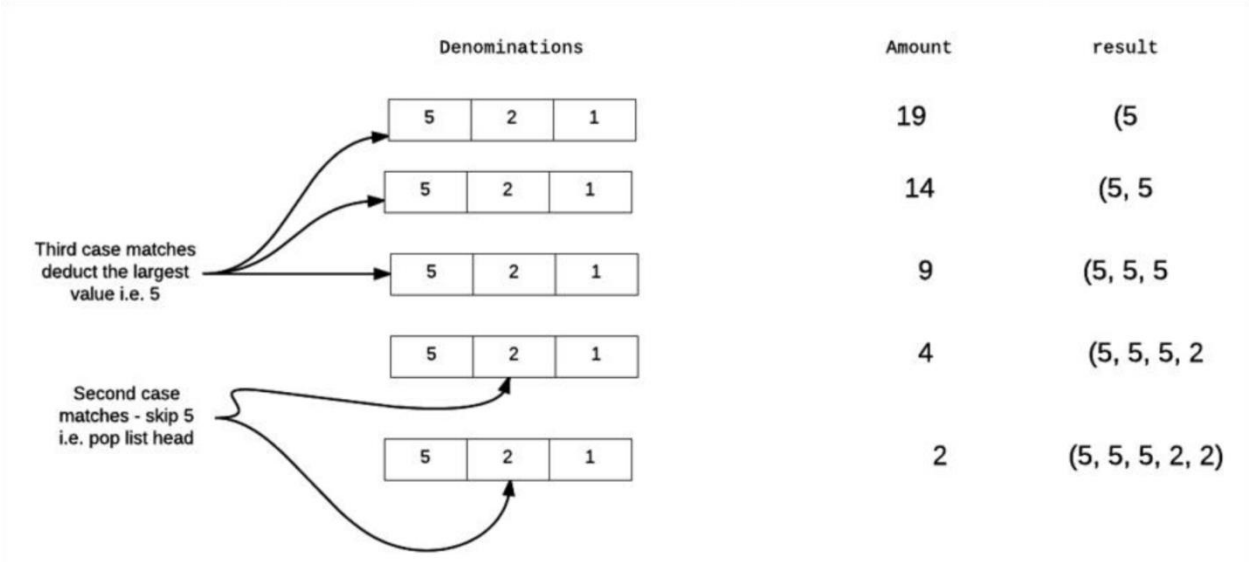
Slika 5. Koncizna metoda za podudaranje znakovnih nizova (izvor: autor)

4.2 Pohlepni algoritmi

Pohlepni algoritam jedan je od najjednostavnijih algoritama kombinatorne optimizacije. Sadrži optimalno rješenje problema radeći slijed izbora, tj. za svaku točku odluke u algoritmu odabire se izbor koji je u tome trenutku najbolji. Pohlepna paradigma često se koristi u kombinatornoj teoriji i praksi optimizacije. Naširoko je pretpostavljeno kako pohlepni algoritam rijetko pruža optimalna rješenja, a puno češće pruža neku vrstu približne vrijednosti, odnosno pruža rješenja koja su donekle srazmjerna.
Pohlepni algoritmi lagani su za provedbu i brzo rade, tj. smanjuju prostor pretraživanja, te također smanjuju i vremensku složenost. Naime, pohlepna strategija u svakom trenutku usmjerava na trenutno najbolje rješenje odnosno lokalni optimum, ne uzimajući u obzir da to rješenje ne mora voditi globalnom optimumu. Upravo zbog toga, pohlepni algoritmi ne moraju uvijek naći optimalno rješenje, ali su značajno brži od drugih algoritama.

Međutim, na primjeru sa „kovanicama“ moguće je primjetiti kako to točno algoritam pohlepe griješi pri izvršavanju. Naime, ukoliko treba vratiti 19 jedinica po apoenima u vrijednosti od 5 i 2, to izvršava lako što je prikazano i na primjeru. Od 19 pa sve do 4

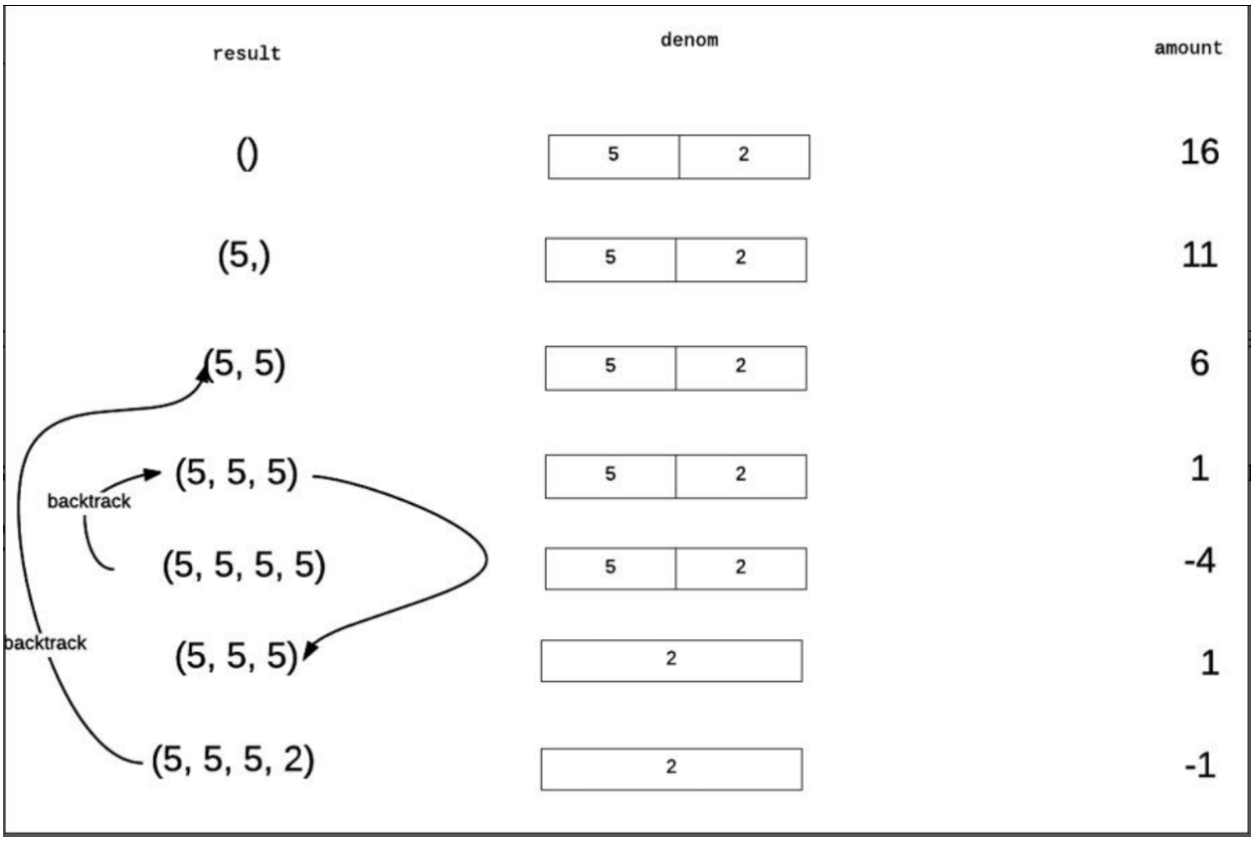
uzima apoene u vrijednosti od 5, te naposljetku kada mu preostane vrijednost od 4 vrlo jednostavno dolazi do optimalnoga rješenja s vraćanjem preostalih po 2.



Slika 6. Pohlepni algoritam na primjeru kovanica 1.(Khot i Mishra, 2017)

No, što da u problemu nije zadano 19, nego primjerice 16?

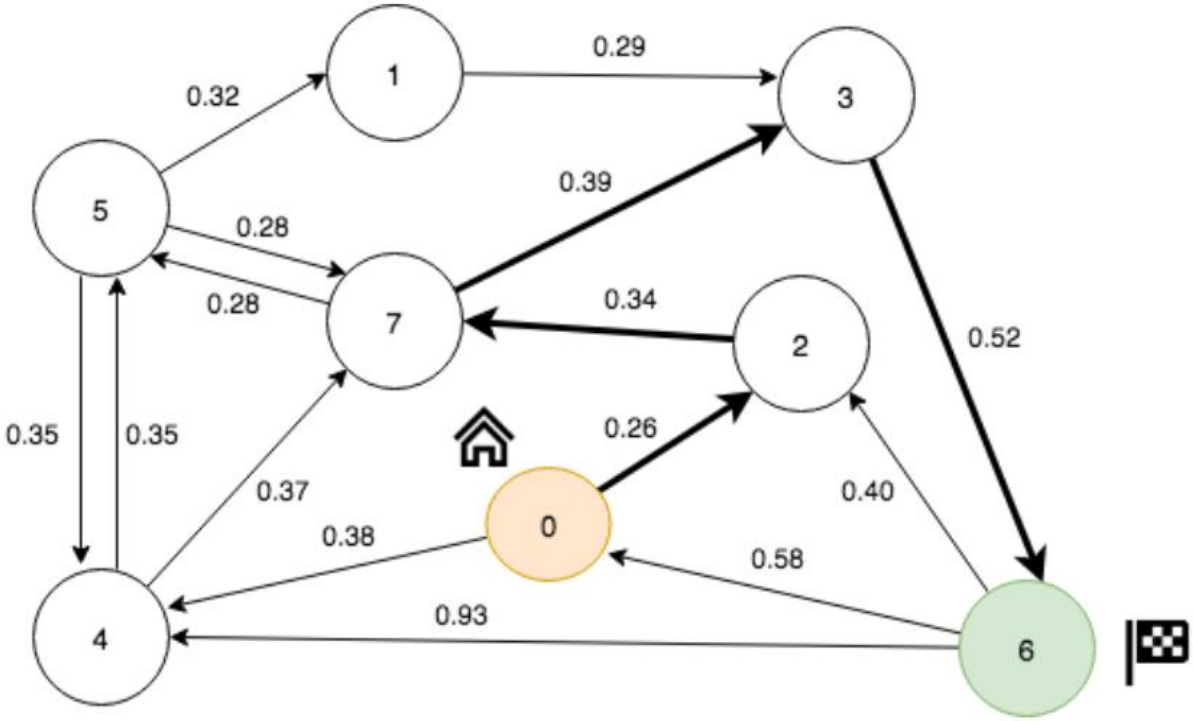
U tom slučaju prosti algoritam pohlepe je neefikasan, stoga da bi se došlo do solucije potrebni su algoritmi s pretraživanjem unazad (eng. backtracking). U primjeru sa 16 započinje na isti način, no primjetno je kako kada padne na -4, vraća se na prethodnu fazu gdje ispituje moguće varijante, no kada je i to pogrešno, vraća se za još jedan korak gdje postaje dostatan.



Slika 7. Pohlepni algoritam na primjeru kovanica 2. (Khot i Mishra, 2017)

Uz pomoć pohlepnih algoritama rješavamo raznolike problemske zadatke, a jedan od najpoznatijih prikazan je u nastavku.

4.2.1. Problem najkraćeg puta



Slika 8. Prikaz problema najkraćega puta (Sedgewick i Wayne, 2011)

Problem najkraćega puta glasi kako je potrebno pronaći najbližu relaciju od točke x do točke y, točnije na primjeru, treba pronaći najkraći put od točke 0 do točke 6. Rješenje problema bi trebalo glasiti [0, 2, 7, 3, 6], sa ukupnom udaljenosti u iznosu od 1.51. Izazovi pri rješavanju problema su:

- Budući kako postoji više puteva između x i y, potrebno je izabrati najkraći među njima
- Potrebno je poštivati smjer među čvorovima tražeći put između x i y
- Neki čvorovi su nedohvatni, tj. ne postoji direktna veza između dva čvora

4.2.2. Dijkstrin algoritam

Dijkstrin algoritam rješava problem najkraćeg puta u grafu s nenegativnim težinama bridova, sa napomenom kako graf ne mora nužno biti usmjeren. Algoritam sam pravi stablo najkraćeg puta koje sadrži najkraće putove od početnog pa do svih ostalih vrhova grafa. Isti algoritam temelji se na apstraktnoj strukturi podataka - redovima s prioritetima u kojima svaki element ima prioritet koji mu omogućuje prednost pred ostalim elementima. Odnosno, može se reći da redovi s prioritetima pohranjuju skup disjunktnih elemenata i da je svakom elementu pridružen ključ koji predstavlja njegovu prioritetnu vrijednost.

```scala
final case class UsmjereniVrh(from: Int, to: Int, weight: Double)

final case class DijkstraGraf(adj: Map[Int, List[UsmjereniVrh]] = Map.empty) {

}

/*    adj - rabi se za prikaz grafova, i svega sa čvorovima.
      Dva najčešća oblika su mape i matrice (adj. skraćeno od adjacency=susjedstvo). */

object DGrafPrikaz {
  implicit class DGrafPrikaz(g: DijkstraGraf) {

    def dodajVrh(v: UsmjereniVrh): DijkstraGraf = {

      val list = g.adj.getOrElse(v.from, List.empty)
      val adj = g.adj + (v.from -> (list :+ v))
      DijkstraGraf(adj)

    }
    /*  DGrafPrikaz prvo pomoću dodajVrh dodaje usmjereni vrh 'v' u DijkstraGraf
     *     tako što kreira listu sa susjednim vrhovima, te ostavlja trenutni
     *     graf 'g' nepromjenjivim
     *
     *     parametar v - pomoćni, služi pri dodavanju vrha iz UsmjereniVrh u graf 'g'
     *
     *     getOrElse - metoda se koristi kad je potrebno ili vratiti vrijednost ako je
     *     ista prisutna ili zadanu vrijednost (default) kada vrijednosti nema
     *
     *     Operator -> služi pridruživanju vrijednosti znaku, u kontekstu mapa
     *     Na kraju instanciranjem vraća novi graf sa dodanim vrhom 'v'
     *  */
}
```

Slika 9. Definiranje klasa za graf i dodavanje vrhova grafa u mapu uz komentare (izvor: autor)

Rješavanju problema se prvotno pristupa na način da se kreiraju „case klase“. U dokumentaciji Scale jest navedeno kako su to obične klase koje eksponiraju svoje parametre konstruktora i omogućuju rekurzivnu dekompoziciju pomoću podudaranja uzorka (eng. pattern matching) koje predstavljaju graf i njegove čvorove. Korištene su mape koje za razliku od polja su nešto fleksibilnije, jer broj čvorova nije nužno predodređen. Struktura podataka sa slike 9. preko liste dozvoljava dodavanje čvorova u mapu, a sam čin izvodi se pozivom metode *dodajVrh*. Napomene radi, inače kada je nešto potrebno dodati iznimno na početak liste koristi se „cons“ operator(::), koji će se rabiti u primjerima nekih metoda sortiranja.

```
import scala.collection.mutable.ArrayBuffer
import dijkstra.DGrafPrikaz._

val g = DijkstraGraf()
  .dodajVrh(UsmjereniVrh(4, 5, 0.35))
  .dodajVrh(UsmjereniVrh(5, 4, 0.35))
  .dodajVrh(UsmjereniVrh(4, 7, 0.37))
  .dodajVrh(UsmjereniVrh(5, 7, 0.28))
  .dodajVrh(UsmjereniVrh(7, 5, 0.28))
  .dodajVrh(UsmjereniVrh(5, 1, 0.32))
  .dodajVrh(UsmjereniVrh(0, 4, 0.38))
  .dodajVrh(UsmjereniVrh(0, 2, 0.26))
  .dodajVrh(UsmjereniVrh(7, 3, 0.39))
  .dodajVrh(UsmjereniVrh(1, 3, 0.29))
  .dodajVrh(UsmjereniVrh(2, 7, 0.34))
  .dodajVrh(UsmjereniVrh(6, 2, 0.40))
  .dodajVrh(UsmjereniVrh(3, 6, 0.52))
  .dodajVrh(UsmjereniVrh(6, 0, 0.58))
  .dodajVrh(UsmjereniVrh(6, 4, 0.93))
```

Slika 10. Način dodavanja čvorova u mapu (izvor: autor)

U istomenom objektu definirano je kako pronaći najkraći put od izvorišne točke sourceV, do svih drugih čvorova u grafu. Osim što ne može biti negativne vrijednosti, potrebno je

odrediti put kojim se prolazi točkama, te ukupna udaljenost među pređenim točkama. Vrijednosti u varijable se pohranjuju pomoću strukture „ArrayBuffera“, koji za razliku od klasičnih polja nisu fiksne veličine. Polaskom iz ishodišne točke pretpostavljena udaljenost iznosi 0.0 u tome trenutku. Susjedni čvorovi se sortiraju u red po njihovoj veličini pri polasku iz čvora pomoću „PriorityQueue naredbe“ koja slaže u red nadolazeće čvorove, a izguruje prijeđene na osnovu najmanje veličine. Jednom kada su prijeđeni svi dohvatljivi čvorovi ne polažu se više u pomoćni red, a vrhovi se smještaju na odgovarajuće mjesto. Skraćena sintaksa funkcije „=>“ predstavlja pogodan način definiranja neimenovane funkcije koja se može proslijediti ili kao varijabla ili kao parametar pozivu metode.

```
object NajkraciPut {

  def kreni(g: DijkstraGraf, sourceV: Int): Either[String, NajkraciPutIzracun] = {
    val velicina = g.adj.size

    if (sourceV >= velicina) Left(s"Izvorišna točka mora biti u rasponu [0, $velicina)")
    else {
      val doVrha = ArrayBuffer.fill[Option[UsmjereniVrh]](velicina)(None)
      val udaljenost = ArrayBuffer.fill(velicina)(Double.PositiveInfinity)

      udaljenost(sourceV) = 0.0
      val izvorisnaUdaljenost = (sourceV, udaljenost(sourceV))
      val sortirajPoVelicini = Ordering[(Int, Double)] = (a, b) => a._2.compareTo(b._2)
      val queue = mutable.PriorityQueue[(Int, Double)](izvorisnaUdaljenost)(sortirajPoVelicini)

      while (queue.nonEmpty) {
        val (minUdaljenost, _) = queue.dequeue()
        val vrhovi = g.adj.getOrElse(minUdaljenost, List.empty)

        vrhovi.foreach { v =>
          if (udaljenost(v.to) > udaljenost(v.from) + v.weight) {
            udaljenost(v.to) = udaljenost(v.from) + v.weight
            doVrha(v.to) = Some(v)
            if (!queue.exists(_._1 == v.to)) queue.enqueue((v.to, udaljenost(v.to)))
          }
        }
      }
      Right(new NajkraciPutIzracun(doVrha, udaljenost))
    }
  }
}
```

Slika 11. Dijkstrin algoritam (izvor: autor)

Dok za očitanje udaljenosti do destinacijskog čvora je trivijalno, jer je vidljivo u polju *udaljenost,* prepoznavanje najkraćeg puta jest nešto nespretnije, no ipak nije komplicirano kao što bi moglo biti. Nakon što je ispisano polje doVrha, nastavlja se iščitavati od posljednjeg čvora, pa sve do startnog korak po korak, gdje redoslijed čvorova prepoznajemo po parametru „v“ u listi. Na kraju je vidljivo kako redoslijed [0, 2, 7, 3, 6] i jest najkraći mogući put.

1. v4(3, 6, 0.52)
2. v3(7, 3, 0.39), *v4*(3, 6, 0.52)
3. v2(2, 7, 0.34), *v3*(7, 3, 0.39), *v4*(3, 6, 0.52)
4. v1(0, 2, 0.26), *v2*(2, 7, 0.34), *v3*(7, 3, 0.39), *v4*(3, 6, 0.52)

4.3 Algoritmi pretraživanja s vraćanjem i konjićev skok

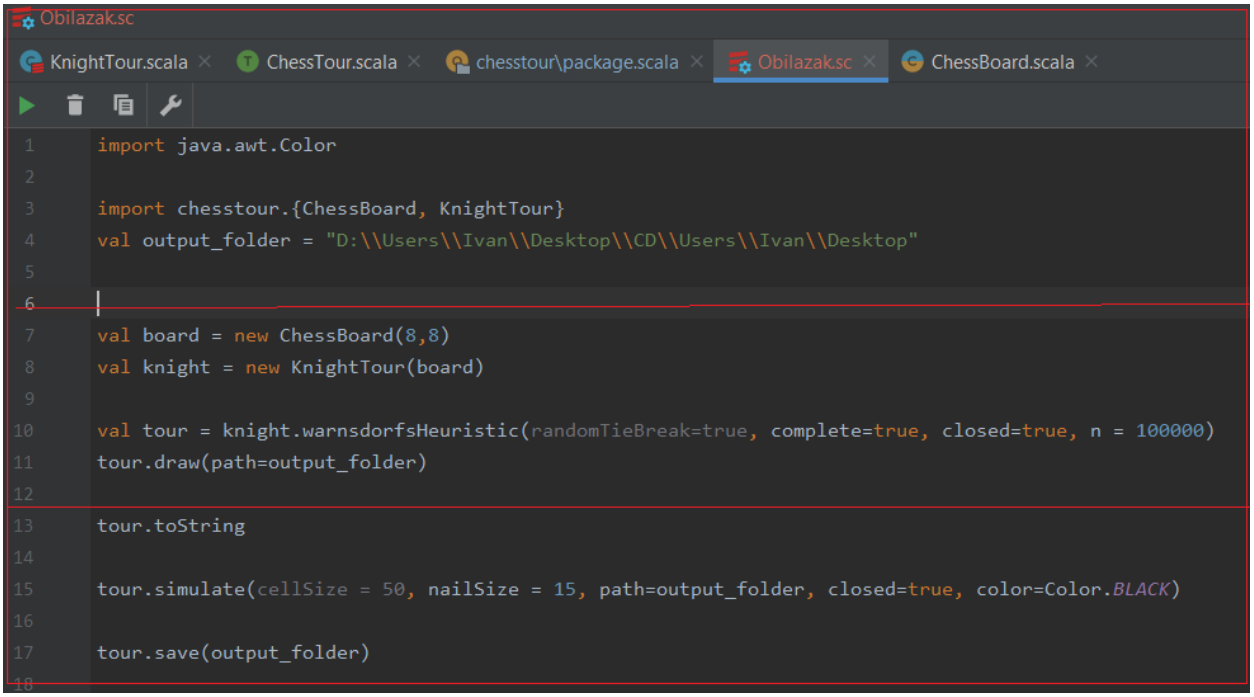
Za razliku od algoritama pohlepe koji ne mogu biti primijenjeni na svakom problemu, to nije slučaj za algoritme pretraživanja s vraćanjem (eng. backtracking). Inače, to je rekurzivna tehnika kod oblikovanja algoritama, i rješavanje vrši na način da svakim idućim korakom uklanja nemoguća rješenja, pritom pazeći na zadana ograničenja problema.

„Algoritmi pretraživanja s vraćanjem su algoritamska paradigma koja ispituje razne solucije sve dok ne dođe do one ispravne. Ovaj tip problema može biti dokučen samo na način ispitivanja svake moguće konfiguracije samo jedanput, što nije slučaj pri „naivnim algoritmima“ koji ispituju svaku moguću konfiguraciju i njezin „output“ za određeno problemsko ograničenje“ (Khot i Mishra, 2017, str. 113).

Od problema za koje se primijenjuju algoritmi pretraživanja s vraćanjem jest konjićev skok (eng. knight's tour), odnosno obilazak šahovskog skakača.

Problem obilaska šahovskog skakača jedan je od poznatijih problema u računalnoj znanosti i šahu, ljudima poznat više od tisuću godina. Sastoji se od pronalaženja slijeda

poteza skakača na šahovskoj ploči dimenzija 8x8, na način da svako polje posjeti samo jedanput. Postoje otvoreni i zatvoreni tip obilaska pločom. Dok je otvoreni pragmatičan, zatvoreni tip zahtjeva da je početna pozicija dohvatljiva sa konačne, što dodatno otežava situaciju. Pri rješavanju se primjenjuje Warnsdorffovo pravilo koje tvrdi kako skakač treba napredovati na ono polje s kojeg će imati manje opcija za skok poslije toga.



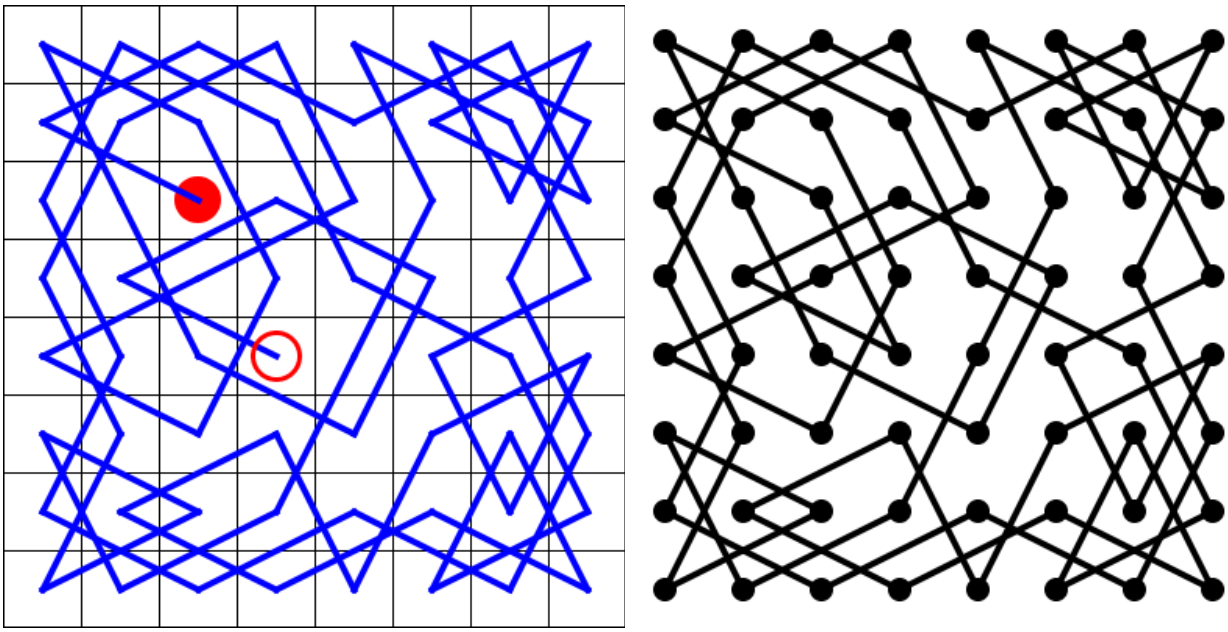
Slika 12. Implementacija algoritama za obilazak skakača (Izvor: autor)

Izvorni kod za šahovsku ploču, skakača i definirano Warnsdorffovo pravilo preuzeti su sa https://github.com/edouardfouche/chesstour, a iznad je prikazano kako rabiti navedeno. Radi jednostavnijeg shvaćanja program je podijeljen na tri sekcije.

U novootvorenom listu (eng. worksheet), u prvom dijelu potrebno je uvesti klase za boju iz Javinog paketa, definirati mapu gdje će se output pohranjivat, te uvesti ploču i kretanje skakača iz kloniranog chesstour paketa. Zatim je potrebno instancirati dimenzije ploče, dodati skakača na ploču, i primjeniti Warnsdorffovo pravilo sa uvjetom

da se radi o zatvorenom tipu obilaska, a pomoću *draw* metode producira se output u prethodno definiranu mapu.

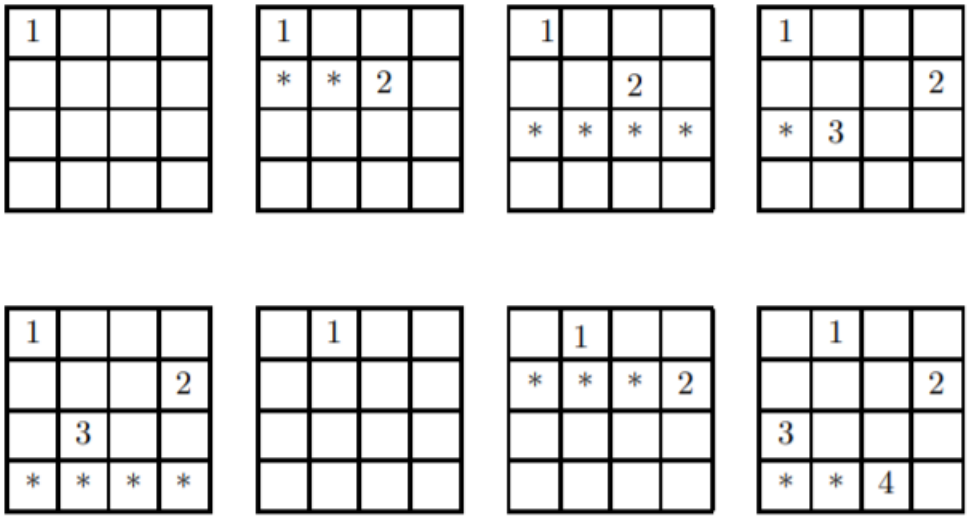
U zadnjoj sekciji određuje se ispis, veličina polja i „čavla“ za pređeno polje u prikazu simulacije, te se pohranjuje naredbom *save* da bi se mogla primjeniti ista simulacija.



Slika 13. Prikaz rješenja u PNG formatu iz output mape sa računala (Izvor: autor)

4.3.1. Problem N-kraljica

Još jedan od problema iz backtracking domene je problem N-kraljica, gdje je kraljice potrebno smjestiti da se međusobno ne napadaju na šahovskoj ploči n * n veličine, a osmislio ga je Max Bezzel još 1848. godine na standardnoj 8 * 8 ploči. Sama problematika leži u tome što povećanjem broja N i broj solucija raste eksponencijalno, tako da primjera radi, na 15 * 15 ploči za isto toliko kraljica postoji 2,279,184 mogućih kombinacija. Problemu je uspješno nadišao Edsger Dijkstra 1971. godine, kada je algoritmom pretraživanja s vraćanjem na primjeru ovoga problema utvrdio moguća rješenja. Najveće opisano rješenje je za N = 27.



Slika 14. Koraci algoritma za problem N kraljica (Manger i Marušić, 2003)

Prikazana je ilustracija slijeda koraka algoritma gdje brojevi označavaju pozicionirane kraljice, a zvjezdice mjesta gdje algoritam pokušava smjestiti iduće, no u konačnici odustaje jer je polje napadnuto drugom kraljicom. Kako bi se uspješno implementirao postupak potrebno je odrediti klasu sa torkama (eng. tuples) uz pomoć koje se unosi broj, te metode za nenapadnuto polje i izračun rješenja permutacijama koje provodi „iterator“ koji je dio Scaline standardne biblioteke. Također kako navodi dokumentacija,

u situacijama gdje podatak ne bi trebao biti mijenjan meritorno je rabiti torke, a ne primjerice listu jer torke koriste manje memorije, pa samim time i izvedba je brža.

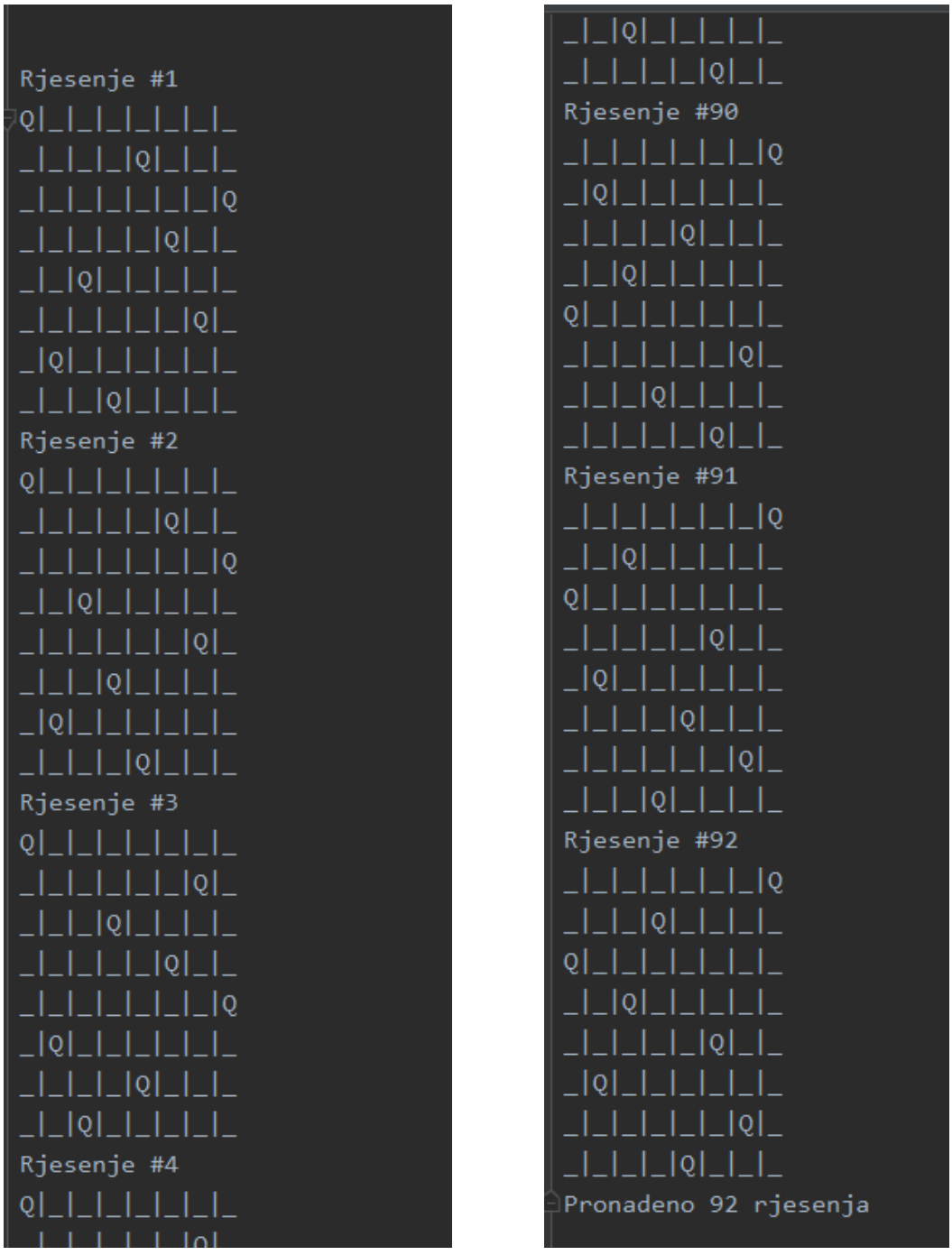
```
object NKraljica {

  private implicit class ParBrojeva[T](
                                       par: (T,T))(
                                       implicit broj: Numeric[T]
                                     ) {
    import broj._

    def polje(x: T, y: T): Boolean =
      par._1 - par._2 != abs(x - y)
  }

  def izracunaj(n: Int): Iterator[Seq[Int]] = {
    (0 to n-1)
      .permutations
      .filter { v =>
        (0 to n-1).forall { y =>
          (y+1 to n-1).forall { x =>
            (x,y).polje(v(x),v(y))
          }
        }
      }
  }

```

Slika 15. Klasa i metode za problem N kraljica (izvor: autor)

U osnovnoj funkciji main su osim spomenutog naredbe za ispis rednog broja rješenja i zbroja svih varijanti rješenja za N=8, što je u ovom slučaju šahovska ploča standardne veličine, dok je ispod naknadno definirana metoda za ispis virtualne ploče i notacija za damu „Q“ (eng. queen). Na slici 17. prikazano je kako izgleda konačan ispis za neke od kombinacija.

```scala
  def main(args: Array[String]): Unit = {
    val n = args.headOption.getOrElse("8").toInt
    val (rjesenje1, rjesenje2) = izracunaj(n).duplicate
    rjesenje1
      .zipWithIndex
      .foreach { case (rjesenje, i) =>
        Console.out.println(s"Rjesenje #${i+1}")
        ispis(n)(rjesenje)
      }
    /*headOption - funkcija koja vraća : Some(vrijednost)
    * ukoliko ima stavki u strukturi podataka, ili None
    * u suprotnome
    *
    * zipWithIndex - funkcija iteracije koja kreira nove parove ili torke
    * sastojeći se od istih elemenata i njihovih odgovarajućih indeksa
    */

    val n_rjesenja = rjesenje2.size
    if (n_rjesenja == 1) {
      Console.out.println("Pronadeno 1 rjesenje")
    } else {
      Console.out.println(s"Pronadeno $n_rjesenja rjesenja")
    }
  }

  def ispis(n: Int)(ploca: Seq[Int]): Unit = {
    ploca.foreach { kraljica =>
      val row =
        "_|" * kraljica + "Q" + "|_" * (n-kraljica-1)
      Console.out.println(row)
    }
  }
}

NKraljica.main(Array("8"))

```

Slika 16. Funkcije main i ispis sa komentarima (izvor: autor)



Slika 17. Prikaz početka i kraja ispisa rješenja (izvor: autor)

4.4. Algoritmi dinamičkog programiranja i rekurzivni algoritmi

Kako neki algoritmi dijele problem na manje podprobleme, te potom koriste rješenja podproblema kako bi konstruirali pravo rješenje, u tome procesu broj podproblema može postati glomazan, pa vrijeme izvršavanja algoritma dolazi u pitanje. Među temeljne probleme spada masovno ponovno izračunavanje već znanih vrijednosti, što nije slučaj u dinamičkom programiranju gdje se vrijednosti pohranjuju što štedi vrijeme.

„Dinamičko programiranje je uglavnom optimizacija rekurzivnih algoritama, odnosno gdje god vidimo rekurzivno rješenje koje se ponavlja za iste unose (eng. input), moguće ga je optimizirati uz pomoć dinamičkog programiranja. Ideja je pohraniti rezultate podproblema, kako računalo ne mora više puta vršiti iste kalkulacije“ (Kulikov i Pevzner, 2018).

U matematici, neobičan matematički niz definiran kao Fibonaccijev, jest niz u kojem nakon dvije početne vrijednosti, svaka sljedeća ima vrijednost zbroja dvaju prethodnika. Idealan je primjer dinamičkog programiranja iz razloga što izbjegava konstantno ponavljanje kalkulacija čime ubrzava izvođenje i povećava efikasnost.

Slika 18. Fibonaccijev niz while petljom (Izvor: autor)

Na slici 18. evidentno je kako za definirati Fibonaccijev niz su dostatni rekurzija i while petlja. No, postoji puno elegantnije rješenje, a to je uporabom repne rekurzije.

„Rekurzija je jedna od najprisutnijih metoda među algoritamskim konceptima. Inače, algoritam je rekurzivan kada poziva samoga sebe“ (Kulikov i Pevzner, 2018, str. 12).

Repna rekurzija, ili *tail recursion* se odvija kada u funkciji, posljednja operacija vraća rezultat pozivajući samu sebe. U dolje prikazanom primjeru primjetno je kako u slučaju nule, prethodna znamenka postaje nula, a u svim ostalima slučajevima ona iznosi n-1, gdje je b vrijednost prije nje. Kao konačnu vrijednost ova rekurzivna funkcija vraća n, što je broj na zadanom mjestu u Fibonaccijevu nizu. Shodno tome „običnom“ rekurzijom vidljivo je da na šestom mjestu niza nalazi se broj 8, a uporabom repne rekurzije izračunati su brojevi na 7. i 28. poziciji.

```
14
15      def fibonacci_niz_tail( n : Int) : Int = {
16        def fib_tail( n: Int, a:Int, b:Int): Int = n match {
17          case 0 => a
18          case _ => fib_tail( n-1, b, (a+b)%1000000 )
19        }
20        return fib_tail( n, 0, 1)
21      }
```

Slika 19. Fibonaccijev niz repnom rekurzijom (Izvor: autor)

```
23  fibonaccijev_niz(n = 6)                    res0: Int = 8
24
25  fibonacci_niz_tail(n = 7)                  res1: Int = 13
26
27  fibonacci_niz_tail(n=28)                   res2: Int = 317811
28
```

Slika 20. Output dobiven iz oba primjera (Izvor: autor)

4.5. Metoda podijeli pa vladaj

„Dok je jedan veliki problem težak za rješiti, dva upola manja su znatno jednostavnija. U tom slučaju, metoda podijeli pa vladaj radi upravo to. Cjepanjem problema na podprobleme pojednostavljuje se težina istih, a spajanjem tih rezultata lakše se dolazi do rješenja izvornog problema. U praksi je situacija puno kompliciranija od ovoga, pa nakon prve podjele, podijeli pa vladaj algoritam nastavlja dijeliti podprobleme na još manje do točke u kojoj mu više nije potrebno“ (Kulikov i Pevzner, 2018, str. 18).

Ova metoda prisutna je u algoritmima sortiranja spajanjem i brzog sortiranja. Ti algoritmi se navode među bržim opcijama, a njihovi primjeri su objašnjeni u idućem poglavlju.

Valja još spomenuti i randomizirane algoritme. To je tip algoritama koji uz deterministička, u svojoj logici primjenjuje jedan ili više slučajnih faktora. Primjera radi kod randomiziranog quicksorta, pivot nije prethodno definiran, nego ga algoritam odabire slučajno.

5. Algoritmi sortiranja

Neka od najosnovnijih stvari što računala čine jest sortiranje podataka po određenim vrijednostima, te pretraživanje kroz podatke radi pronalaženja informacija. Zbog važnosti sortiranja i pretraživanja, puno je truda uloženo kako bi algoritmi postali što brži i efikasniji, a navedeni su neki od najčešćih pristupa sa njihovim vremenskim zahtjevima najboljih i najgorih slučajeva, ali i zahtjevima prosjeka.

„Postoje razni načini na koje sortirati podatke. Oni najučestaliji nemaju posebne zahtjeve za posebnost podataka. Sve što im je potrebno jest sposobnost usporedbe elemenata kako bi se primjetilo koji element treba doći ispred kojega. Ovi načini sortiranja se rabe najčešće, upravo zbog njihove fleksibilnosti. Postoje ograničenja na temelju toga koliko brzo mogu izvršavati sortiranje jer nisu prilagođeni podatcima, ali generalno njihova općenitost ide u prilog brzini izvršavanja“ (Lewis i Lacher, 2016, str. 401).

5.1 Mjehuričasto sortiranje

Mjehuričasto sortiranje (eng. bubble sort) stabilna je i jednostavna metoda sortiranja. „Metoda uzastopno pomjera najveći (ili najmanji) element na najveći index u polju. Radi na principu usporedbe svakoga elementa sa njegovim susjednim, te ih sortira prema tome. Ako su dva susjedna polja nesortirana, zamjenjuje ih. Međutim, metoda baš zbog toga nije efikasna jer zahtjeva dosta ponavljanja, i vjerojatnost da se polje podataka sortira u jednome pokušaju jest jedino ukoliko je na početku bilo približno sortirano“ (Lewis i Lacher, 2016, str. 402).

```
def bubbleSort(a:Array[Double]):Unit = {
  for (j <- 0 until a.length-1) {
    for (i <- 0 until a.length-1-j) {
      if (a(i) > a(i+1)) {
        val tmp = a(i)
        a(i) = a(i+1)
        a(i+1) = tmp
      }
    }
  }
}
```

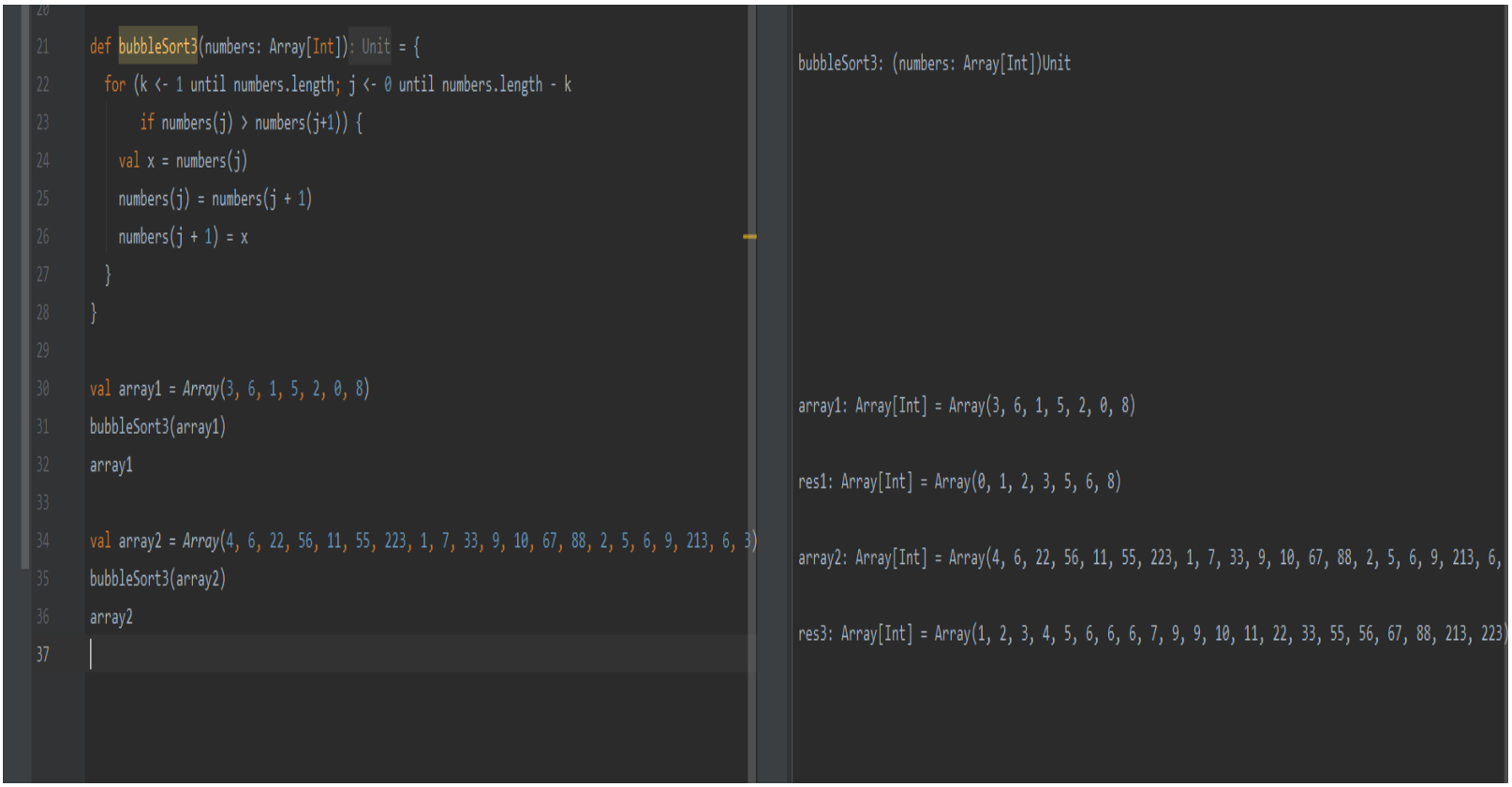
Slika 21. Mjehuričasto sortiranje (Lewis i Lacher, 2016)

„U ovome kodu primjetno je kako se vanjska petlja izvodi n-1 puta, gdje se vrijednost za „j“ zbraja od nule do n-2. U unutrašnjoj petlji fraza „a.length-1“, se proširuje na „a.length-1-j“ radi toga što primjerice nakon prvog prolaska najveći element se nalazi na kraju. Isto tako nakon drugog prolaska drugi najveći zauzima svoje mjesto, pa iz tog razloga unutrašnja petlja smanjuje algoritam za mjesto, jer nema potrebe provjeravati vrijednosti koje su sigurno sortirane, iako nije pogrešno ni bez unutrašnje petlje“ (Lewis i Lacher, 2016, str. 403).

Sama zamjena mjesta vrši se ispod petlji gdje se ispituje „if“ izrazom, odnosno je li element veći od onoga nakon njega, te ukoliko je, uz pomoć privremene varijable dolazi do zamjene. Valja primjetiti isto kako je na primjeru prikazan tip podataka polja „double“, pa ništa osim vrijednosti „double“ neće ni raditi. Iz navedenih razloga posljednji prikazani primjer sortira podatke tipa „integer“, gdje su zadana dva niza brojeva u polju, koje sortiranje mjehurića ispravno sortira.

Uz mjehuričasto sortiranje često se nadovezuje problem zečeva i kornjača koji nosi takav naziv iz razloga što elementi većih vrijednosti prolaze listom brže od onih s manjim vrijednostima. Radi na principu ponavljanja prolaska kroz niz, pa proporcionalno tome što je manji broj elemenata za sortiranje metoda je efikasnija, a za veći broj elemenata sortiranje mjehurića je presporo. Najgori slučaj na koji metoda može naići bio bi potpuno nesortirano polje odnosno sortirano silazno, gdje bi se svaki put morala

izvoditi i usporedba i zamjena elemenata stoga vremenska kompleksnost najgoreg slučaja sortiranja mjehurića iznosi $O(n^2)$, gdje je „n“ broj elemenata u nizu. Iz istog razloga performans prosjeka također iznosi $O(n^2)$ jer bez obzira bio niz više ili manje raštrkan, algoritam će morati proći usporediti elemente jednak broj puta. Usporedno svemu spomenutome najbolji slučaj bio bi već sortirano polje i u tom slučaju proći će nizom samo jednom i neće izvršiti niti jednu zamjenu, tako da i vremenska kompleksnost prosjeka iznosi O(n).



Slika 22. Primjer mjehuričastoga sortiranja (Izvor: autor)

5.2. Sortiranje izborom (Min/Max sortiranje)

„Algoritam sortiranja izborom (eng. selection) radi na način da u polju podataka pronalazi najmanji element iz nesortiranog dijela (ili najveći, s obzirom na to sortira li ga uzlazno ili silazno), i polaže ga na početak. Algoritam pokušava održati dva podpolja u zadanom polju:

1) Podpolje u kojem se nalaze sortirani elementi
2) Podpolje sa preostalim nesortiranim elementima

Budući kako uvijek grabi samo jedan, algoritam prolazi poljem onoliko puta koliko je polje veliko, te prebacuje elemente iz nesortiranog u sortirani dio“ (Khot i Mishra, 2017, str. 275).

```
def minSort(a:Array[Double]):Unit = {
  for (j <- 0 until a.length-1) {
    var min = j
    for (i <- j+1 until a.length) {
      if (a(i) < a(min)) min = i
    }
    if (min != j) {

      val tmp = a(j)
      a(j) = a(min)
      a(min) = tmp
    }
  }
}
```

Slika 23. Sortiranje izborom najmanjeg elementa (Lewis i Lacher, 2016)

Kreirana minSort metoda klase, s poljem postavljenim na double, definira se u povratnom tipu *Unit.* Usporedbe radi, „unit“ je nešto poput funkcije „void“ u C/C++ jeziku, a svrha mu je kako bi dao do znanja da metoda ne treba vratiti nikakvu povratnu informaciju. Ipak, kako svaka metoda u Scali vraća vrijednost, definira se Unit, da bi kompajler prepoznao kako ne treba vratiti ništa. Drugim riječima, ako ne treba vratiti ništa, vraća se unit. Na prikazanom primjeru unutarnja petlja prolazi nesortiranim

dijelom i traži najmanji element. Ukoliko taj element već nije na svome mjestu, uz pomoć privremene varijable mijenja mjesto do pretpostavljene pozicije. Kao i kod sortiranja mjehurića, vanjska petlja se odvija „n-1“ puta jer toliko algoritmu treba da sortira sve elemente gdje i pripadaju. Algoritam se definira kao adaptivan, no jako neučinkovit kako količina podataka raste.

Vremenska kompleksnost ove metode sortiranja u najboljem i najgorem slučaju iznosi $O(n^2)$, s obzirom kako uvijek mora proći čitav nesortirani dio kako bi pronašao najveći (ili najmanji) broj u nizu. Iz istih razloga prosječna složenost također iznosi $O(n^2)$.

5.3. Sortiranje umetanjem

Sortiranje umetanjem (eng. insertion sort) naziva se tako jer funkcionira na način da kreira sortiranu grupu elemenata na početku polja, i svaki novi element se pozicionira, odnosno ubacuje na pogodno mjesto u toj skupini. Prednost mu je što izvodi manje provjera, pa je samim time i nešto brži, jer jednom kada pronađe ispravno mjesto za umetnuti element, može stati i započeti sa idućim“ (Lewis i Lacher, 2016, str. 405).

Budući kako sortiranje umetanjem grabi elemente iz nesortiranog dijela i stavlja ih na odgovarajuće mjesto u sortiranom dijelu, najbolji mogući slučaj bio bi da su elementi već sortirani, pa i sama vremenska složenost iznosila bi O(n), jer proći će redom samo potvrditi kako su svi dobro pozicionirani. Ipak najgora i prosječna složenost iznosit će $O(n^2)$ unatoč tome što grupom elemenata prolazi samo jednom. U algoritmu se rabi ugniježđena petlja gdje se vanjska petlja ponavlja „n“ puta, a unutarnja petlja „n * n“ puta, te iz toga razloga konačna složenost najgoreg i prosječnog slučaja je $O(n^2)$.

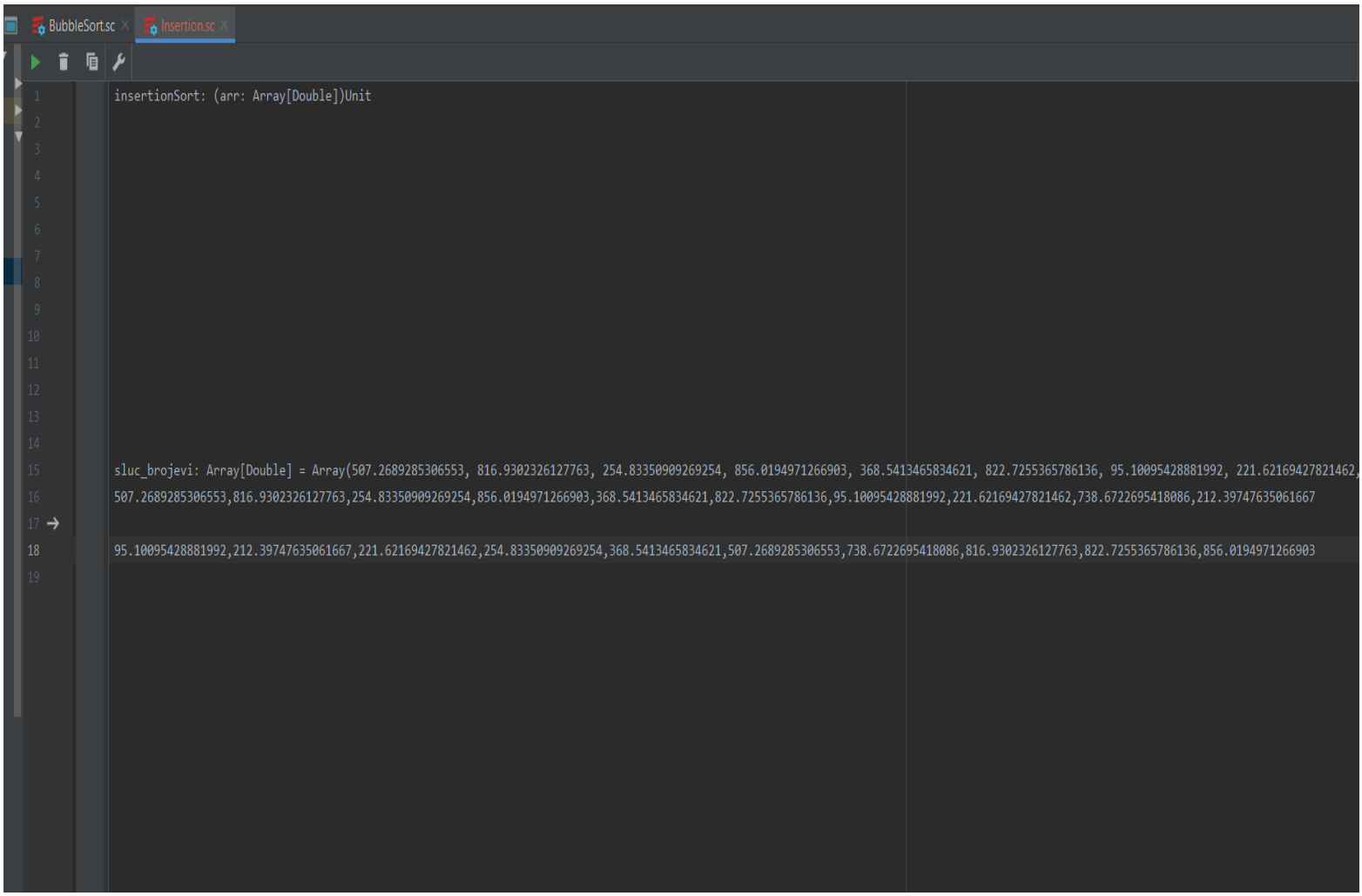
```
def insertionSort(a:Array[Double]):Unit = {
  for (j <- 1 until a.length) {
    var i = j-1
    val tmp = a(j)
    while (i >= 0 && a(i) > tmp) {
      a(i+1) = a(i)
      i -= 1
    }
    a(i+1) = tmp
  }
}
```

Slika 24. Sortiranje metodom umetanja (Lewis i Lacher, 2016)

Ideja ovog tipa sortiranja također jest da se izvodi n-1 puta, no na drukčiji način i s drukčijim razlogom. Umjesto da kreće od nule, te da se zaustavlja kada mu ponestane elemenata, ovdje vanjska petlja kreće od jedan, i ponavlja se sve dok ne dođe do posljednjeg elementa u polju. Programski kod unutar vanjske petlje započinje sa deklariranjem varijable „j“, koja se postavlja na „i-1“. To je indeks idućeg elementa kojeg treba usporediti sa onim koji je potrebno umetnuti. Također se deklarira i privremena nepromjenjiva varijabla „tmp“, koja pohranjuje vrijednost elementa iz polja koji se pomiče. Privremena varijabla čuva vrijednost i prenosi je do odgovarajućeg mjesta, dok se ostali elementi pomiču i prave prostor za njegovo umetanje.

Slika 25. Primjer sortiranja umetanjem (Izvor: autor)

Na prikazanom primjeru može se vidjeti kako izgleda algoritam sortiranja metodom umetanja izrađenom u Intellij okruženju, sa slučajno generiranim brojevima *double* vrijednosti, uz prikaz ispisa na idućoj slici.



Slika 26. Dobiveni rezultat sortiranja umetanjem (Izvor: autor)

5.4. Sortiranje spajanjem

Sortiranje spajanjem (eng. merge sort) i brzo sortiranje (eng. quick sort) koriste istu analogiju podijeli pa vladaj, sa temeljnom razlikom što sortiranje spajanjem doslovce razdvaja polje na dva dijela potom izvodi sortiranje, a brzo sortiranje kao okosnicu usporedbe uzima element nazvan pivot.

Strategija metode podijeli pa vladaj jest točno onakva kako i zvuči. Veliki problem se dijeli, ustanove se manja rješenja, te potom tvore finalnu soluciju.

„Generalna ideja sortiranja spajanjem jest da se zadani skup elemenata može razdvojiti na ugrubo dva jednaka dijela. Svaki od tih dijelova se sortira individualno, te se sortirani rezultati spajaju na kraju. Radi se o brzom algoritmu sortiranja, a razlog tome je što kada je potreban najmanji element iz svake kolekcije, nije nužno pregledavati kroz sve elemente, nego samo uzeti u obzir one sa krajeva gdje su male vrijednosti u obje kolekcije, što znači kako pri pronalasku elementa usporedba se izvodi samo jedanput. Isto tako za idući element, uspoređuju se najmanji s oba kraja, što daje ukupan performans sortiranja O(n log(n)), što je za veće vrijednosti od n puno bolje“ (Lewis i Lacher, 2016, str. 470).

„Mana sortiranja spajanjem jest što ne može biti sproveden „na licu mjesta“ (kada bi se sortiranje odvijalo u samoj kolekciji bez uporabe dodatne memorije), stoga sortiranje spajanjem zahtjeva memorije barem u protuvrijednosti proporcionalnoj od n za izvršavanje. Ipak, dodavanjem dodatnog polja duljine n bi se moglo nadići problemu, no tu se već radi o kompleksnijoj razini uporabe memorije“ (Lewis i Lacher, 2016, str. 470).

Vremenska složenost u sva tri slučaja (najbolji, najgori i prosječni) daje za rezultat O(n log (n)), jer sortiranje spajanjem uvijek dijeli niz na dvije polovice za čije spajanje će mu biti potrebno jednako vremena.

```
def merge(lst1:List[Int], lst2:List[Int]):List[Int] = (lst1,lst2) match {
  case (Nil,_) => lst2
  case (_,Nil) => lst1
  case (h1::t1, h2::t2) =>
    if (h1<h2) h1 :: merge(t1, lst2)
    else h2 :: merge(lst1, t2)
}
```

Slika 27. Sortiranje spajanjem (Lewis i Lacher, 2016)

„Iako obje funkcije rabe podudaranje uzoraka u rekurziji, rekurzija u ovome primjeru mora alocirati okvir podataka u stogu, za svaki element koji se spaja, što znači kako ova verzija koda nije skalabilna za veće veličine. Kako bi se zaobišla ta prepreka koristi se

while petlja u korist imperativnijeg spajanja. Iako verzija nije toliko pregledna i duplo je duža, u stanju je sortirati puno veće okvire podataka“ (Lewis i Lacher, 2016, str. 482).

```
def merge(lst1:List[Int], lst2:List[Int]):List[Int] = {
  var l1 = lst1
  var l2 = lst2
  var ret = List[Int]()
  while (l1.nonEmpty && l2.nonEmpty) {
    if (l1.head<l2.head) {
      ret ::= l1.head
      l1 = l1.tail
    } else {
      ret ::= l2.head
      l2 = l2.tail
    }
  }
  if (l1.nonEmpty) ret :::= l1.reverse
  else ret :::= l2.reverse
  ret.reverse
}
```

Slika 28. Sortiranje spajanjem while petljom (Lewis i Lacher, 2016)

Postoji još jedna, nešto preglednija od while varijante, a to je sa indeksiranim sekvencama nad listom koje su dio Scaline standardne biblioteke, što omogučuje uporabu vektora iz iste, pa bi zato bila i najoptimiziranija varijanta.

```scala
def mergeSort(input: IndexedSeq[Int]): List[Int] = {
  if(input.length == 1) List(input.head)
  else {
    val (left, right) = input.splitAt(input.length / 2)
    val sortedLeft = mergeSort(left)
    val sortedRight = mergeSort(right)
    merge(sortedLeft, sortedRight)
  }
}

val nizbrojeva = Vector(3, 6, 1, 5, 2, 0, 8)
mergeSort(nizbrojeva)

val nizbrojeva = Vector(4,6,22,56,11,55,223,1,7,33,9,10,67,88,2,5,6,9,213,6,3)
mergeSort(nizbrojeva)
```

Slika 29. Sortiranje spajanjem indeksiranim sekvencama (Izvor: autor)

```
nizbrojeva: scala.collection.immutable.Vector[Int] = Vector(3, 6, 1, 5, 2, 0, 8)
res0: List[Int] = List(0, 1, 2, 3, 5, 6, 8)

nizbrojeva: scala.collection.immutable.Vector[Int] = Vector(4, 6, 22, 56, 11, 55, 223, 1, 7, 33, 9, 10, 67, 88, 2, 5, 6, 9, 213, 6, 3)
res1: List[Int] = List(1, 2, 3, 4, 5, 6, 6, 6, 7, 9, 9, 10, 11, 22, 33, 55, 56, 67, 88, 213, 223)
```

Slika 30. Output sortiranja spajanjem (Izvor: autor)

5.5. Brzo sortiranje

Brzo sortiranje (eng. quick sort) jedan je od najbržih poznatih algoritama relativno lagane implementacije, a 1960. osmislio ga je britanski znanstvenik Tony Hoare. Što se tiče metode brzog sortiranja, kao što je već gore spomenuto, ideja je izabrati jedan element koji se naziva pivot, te sve elemente manje od njega staviti prije, a veće poslije njega. Ti elementi neće biti sortirani, no zbog toga što ne zahtjevaju dodatnu uporabu memorije su izvedivi „na licu mjesta“, a sortiranje rekurzijama će biti brže jer se ne mora pregledavati cijeli skup elemenata, nego samo do/od pivota.

```
def quicksort(lista:List[Int]):List[Int] = lista match {
  case Nil => Nil
  case x::Nil => lista
  case _=>
    val p = lista.head
    val (before, after) = lista.tail.partition(_<p)
quicksort(before) ++ (p :: quicksort(after))
}

val brojevi = List.fill(10)(util.Random.nextInt(100))
println(brojevi.mkString(", "))
println(quicksort(brojevi).mkString(", "))
```

Slika 31. Brzo sortiranje (Izvor: autor)

```
brojevi: List[Int] = List(20, 65, 11, 45, 46, 22, 92, 18, 40, 4)
20, 65, 11, 45, 46, 22, 92, 18, 40, 4
4, 11, 18, 20, 22, 40, 45, 46, 65, 92
```

Slika 32. Output brzog sortiranja (Izvor: autor)

Najbolji slučaj je varijanta kada se kao središte uvijek odabire srednji element za pivot, dok za rezultat vremenske složenosti prosjeka koristi se matematička analiza. U oba slučaja složenost iznosi O(n log n). Međutim, iako slovi za jednu od bržih metoda sortiranja, ukoliko pivot bude loš (a u najgorem slučaju izabere najveći ili najmanji element), samim time i efikasnost sortiranja pada u vodu, pa je i vremenska složenost najgoreg slučaja $O(n^2)$. Iz tog razloga može se uzeti dva ogledna pivota, tzv. *dual pivot quicksort*, kako bi se smanjila vjerojatnost tako čega.

6. Algoritmi pretraživanja

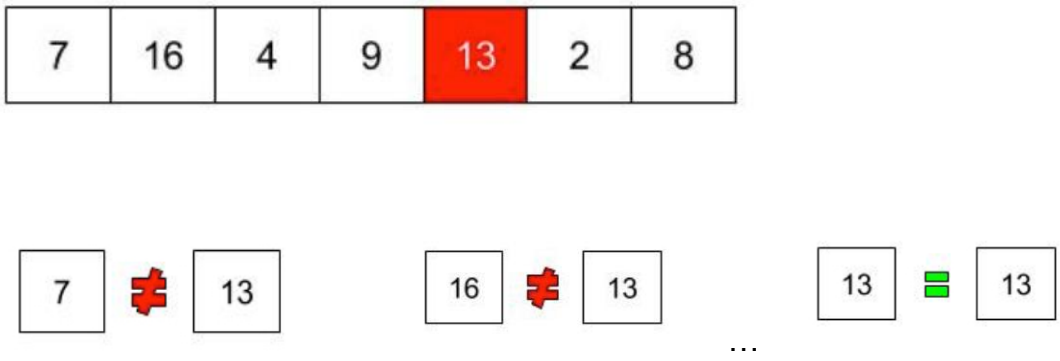
„U računalnoj znanosti algoritam pretrage je svaki algoritam koji vraća povratnu informaciju pohranjenu u nekoj od struktura podataka. Kako bi algoritam izvršio propisno pretraživanje potrebno je poznavati koja se struktura podataka pretražuje, no i poznavanje pretraživanoga podatka“ (Lewis i Lacher, 2016, str. 414).

Nakon sortiranja podataka, elementarno je poznavati i kako ih pretraživati, a dva osnovna oblika algoritama pretraživanja jesu algoritmi linearnog i binarnog pretraživanja.

6.1. Linearno pretraživanje

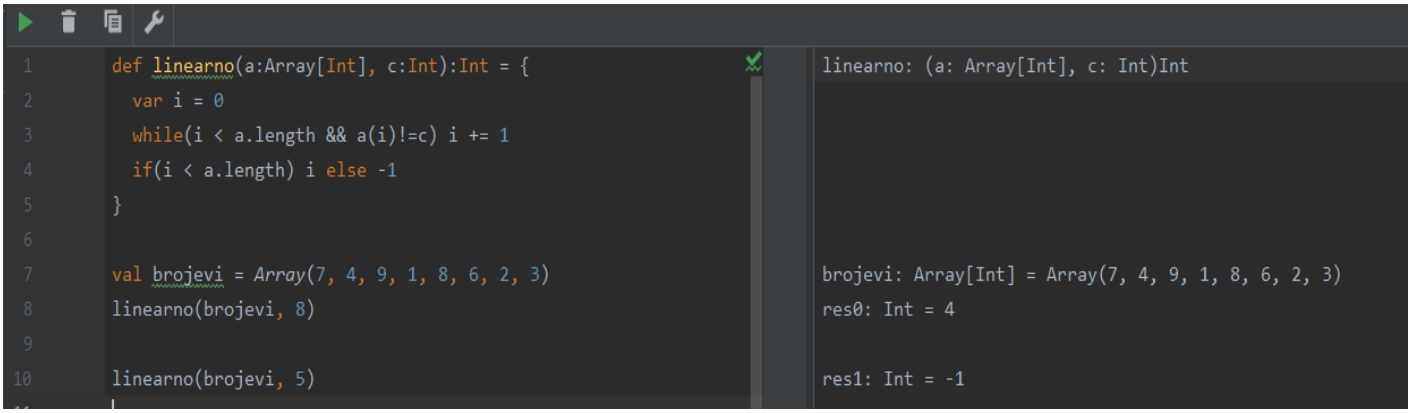
Linearno pretraživanje je način pretraživanja elemenata na osnovu sekvencijalnog pregledavanja. Redom pregledava svaki element u listi ili polju podataka i tako sve dok traženi element nije pronađen, ili dok cijela struktura podataka nije pregledana.

Radi lakšeg shvaćanja, ako se pretražuje broj 13 na zadanom primjeru, algoritam linearnog pretraživanja najprije uzima prvi element, zatim drugi, i tako sve redom dok ne pronađe što mu treba.



Slika 33. Prikaz linearnog pretraživanja (freecodecamp.org)

Sličan problem prikazan je i u Scali. Za metodu „linearno“, inicijalizirana je varijabla i, koja sve dok je manja od iznosa elemenata u polju, uz pomoć while petlje pretražuje polje i traži element. Kada algoritam pronađe element ispisuje njegovo mjesto u polju (napomena: redoslijed brojanja u polju kreće od 0), a ukoliko elementa nema algoritam ispisuje -1.



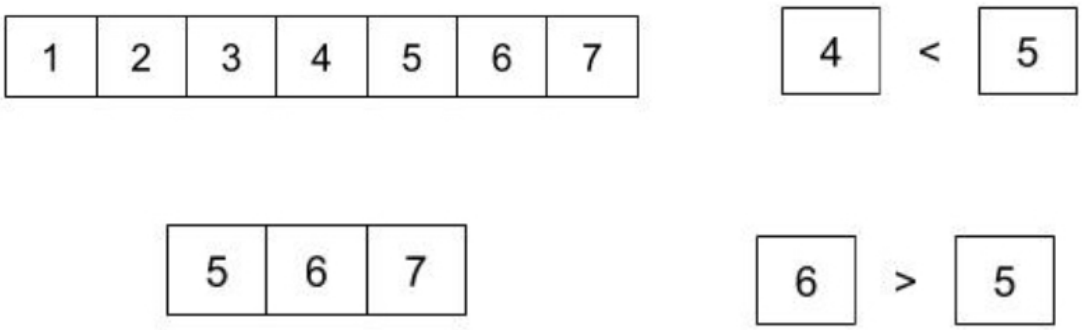
Slika 34. Linearno pretraživanje (Izvor: autor)

Iako je linearno pretraživanje jednostavno, kompleksnost algoritma je O(n), upravo iz tog razloga što pregledava element po element. Samim time algoritam s povećanjem elemenata nije efikasan i proces postaje dugotrajan.

6.2. Binarno pretraživanje

Binarno pretraživanje uspješno nadilazi problemu dugotrajnosti linearnog pretraživanja. Ono pretražuje sortirano polje neprestanim dijeljenem na pola s obzirom je li element veći ili manji od središnjega. Podijeli pa vladaj pristupom dobilo se elegantno rješenje, gdje je kompleksnost algoritma u najgorem slučaju $O(log_2 N)$

Dakle kao što stoji prikazano na slici 35., u sortiranom polju ako se binarno pretražuje broj 5, pronalazi se središnji element koji je 4, pa ako središnji element nije traženi, odbacuje suprotnu stranu i istim načinom nastavlja pretragu dok ne pronađe broj 5.



Slika 35. Prikaz binarnog pretraživanja (freecodecamp.org)

```
def binarno(a:Array[Int], value:Int):Int = {
  var start = 0
  var end = a.length
  var sredina = (start + end) / 2
  while (end > start && a(sredina) != value) {
    if (value < a(sredina)) {
      end = sredina
    } else {
      start = sredina + 1
    }
    sredina = (start + end) / 2
  }
  if (end > start) sredina else -1
}
val porBrojevi = Array(1, 3, 4, 5, 7, 8, 9)

binarno(porBrojevi, value = 2)

binarno(porBrojevi, value = 5)
```

```
binarno: (a: Array[Int], value: Int)Int

porBrojevi: Array[Int] = Array(1, 3, 4, 5, 7, 8, 9)

res2: Int = -1

res3: Int = 3
```

Slika 36. Binarno pretraživanje (Izvor: autor)

U funkciji je najprije potrebno deklarirati varijable za početak, završetak i središnji element. If petljom polje se cijepa na pola zavisno gdje je potrebno, a petlja while nastavlja cijepanje sve dok ne pronađe element, ili ne dođe do kraja. U slučaju da nije pronađen, algoritam ispisuje -1.

7. Zaključak

Scala uz sve vrline i mane pokazala se efikasnim jezikom za uporabu, i interesantnim za učenje i prilagodbu. Ovaj rad je uvod u neke od fascinantnih mehanizama i značajki ovog opsežnog programskog jezika koji je primjenjiv u rasponu od strojnog učenja pa sve do razvoja web aplikacija. Znajući kako su u radu objašnjene najčešće varijante oblikovanja algoritama sa primjerima nekih poznatih problema, može se zaključiti nekoliko stvari. Uz pomoć Dijsktrinog algoritma definiralo se kako odrediti optimalni put u grafu. Isto tako, na primjeru skakača prikazana je jedna kompleksna varijanta algoritma pretraživanja s vraćanjem. Zatim osebujnost algoritma Fibonaccijevog niza, uz pomoć repne rekurzije kao odlike funkcionalnog programiranja u Scali. Uvid u sve te programske probleme daje ideju za ubuduće rješavanje i primjenu pri sličnim problemima. S druge strane, ne treba zaboraviti kako jedan od najbitnijih zadataka koja računala obrađuju jest jednostavnije snalaženje po podatcima, a kako bi se lakše rukovalo podatcima, i jednostavnije pretraživalo po njima, potrebno ih je sortirati. Od algoritama sortiranja navedeni su oni koji predstavljaju primjenu standardnih metoda izgradnje algoritama, a kako se rabe, prikazano je i teorijski i na praktičnom primjeru. Naposljetku, te sortirane strukture služile su i za pokazivanje linearnog i binarnog pretraživanja. Po pitanju Scale i nekih prednosti koje se mogu uočiti, a i sama zajednica navodi su statičko tipkanje, odnosno sposobnost jezika da ne zahtjeva definiranje varijabli prije njihove uporabe, nego da ih se može inicijalizirati bilo gdje prije nego ih se planira koristiti. Zatim „Java virtual machine“ platforma koja je moderna, pouzdana i stabilna, te dozvoljava razvoj aplikacija bilo kakve kompleksnosti i svrhe. Dakako i podudaranje uzoraka kao jedno od najrasprostranjenijih svojstava ovoga jezika eksperti navode kao ogroman plus. Od onoga što se može primijetiti kao nedostatak je definitivno limitiran broj ljudi i primjera pomoću kojih bi bilo lakše savladati ovakav jezik, tako da za adaptirati se jezik nije lagan. Također, budući da je jezik hibrid funkcionalnog i objektno orijentiranog programiranja, ponekad se može činiti konfuznim i težim za razumjeti. Samim time teško se oteti dojmu da jezik odbija nove ljude, no upravo u tome i je poanta, što pogotovo danas ovaj jezik neće nitko učiti bez da je svjestan iz kojeg razloga mu je zaista potreban. Ipak, u svrhu poboljšanja vještina programiranja, nije na

odmet proučiti barem jedan jezik drukčijih paradigmi gdje Scala kao hibrid nudi upravo to. Sve u svemu, od elementarne definicije algoritama, do shvaćanja principa koje neki algoritmi poštuju, ponuđeno je osnovno za razumijevanje ovog programskog jezika.

8. Literatura

1. Martin Odersky (2008), Programming in Scala

2. Vikash Sharma (2018), Learning Scala Programming

3. Robert Sedgewick and Kevin Wayne (2011), The textbook *Algorithms, 4th Edition*

4. Krčadinac V. (2016/2017), Osnove algoritama: https://web.math.pmf.unizg.hr/nastava/oa/oa-skripta.pdf

5. Robert Manger i Miljenko Marušić (2003), Strukture podataka i algoritmi

6. Jonas Boner (2009), Pragmatic Real-World Scala

7. Mark C. Lewis i Lisa L. Lacher (2016), Introduction to programming and problem solving using Scala

8. Mark C. Lewis i Lisa L. Lacher (2016), Object-Orientation, Abstraction, and Data-Structures Using Scala

9. A.S. Khot i R.K. Mishra (2017), Learning Functional Data Structures and Algorithms, Packt Publishing

10. A. Kulikov i P. Pevzner (2018), Learning algorithms through programming and puzzle solving

11. Mark C. Lewis (2012), Introduction to the Art of Programming Using Scala

12. Martin Odersky (2014), Scala by Example

13. Donald Knuth (1968), The Art of Computer Programming

14. https://www.edureka.co/blog/what-is-scala/ 28.8.2020.

15. https://docs.scala-lang.org/ 29.8.2020.

16. https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html 29.8.2020.

Github:

17. https://github.com/edouardfouche/chesstour 20.8.2020.

18. https://github.com/scala-native/scala-native/issues/731 23.8.2020.

19. https://github.com/novakov-alexey/scala-dijkstra 25.8.2020.

9. Popis slika