Učinkovitost izvršavanja koda u programskim jezicima visoke razine

Acinger, Mateo

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Učinkovitost izvršavanja koda u programskim jezicima visoke razine

Rijeka, 07, 2023.

Mateo Acinger
0069089579

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Učinkovitost izvršavanja koda u programskim jezicima visoke razine

Mentor: doc.dr.sc. Goran Mauša

Rijeka, 07, 2023.

Mateo Acinger
0069089579

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA ZAVRŠNE ISPITE

Rijeka, 13. ožujka 2023.

Zavod: **Zavod za računarstvo**
Predmet: **Programsko inženjerstvo**
Grana: **2.09.06 programsko inženjerstvo**

ZADATAK ZA ZAVRŠNI RAD

Pristupnik: **Mateo Acinger (0069089579)**
Studij: Sveučilišni prijediplomski studij računarstva

Zadatak: **Učinkovitost izvršavanja koda u programskim jezicima visoke razine // Code execution efficiency in high-level programming languages**

Opis zadatka:

Istražiti podjelu programskih jezika s obzirom na specifičnosti njihova izvršavanja. Predložiti problemske zadatke koji će demonstrirati razlike u odabranom podskupu jezika visoke razine. Izračunati potrošnju energije i vrijeme izvršavanja u programskim kodovima različite algoritamske složenosti i veličine ulaznih podataka. Statistički analizirati rezultate mjerenja, ustanoviti korelaciju vremena izvršavanja i utrošene energije te rangirati jezike po kriteriju učinkovitosti.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 20. ožujka 2023.

Mentor:

Doc. Goran Mauša, dipl. ing.

Predsjednik povjerenstva za završni ispit:

Prof. dr. sc. Miroslav Joler

Izjava o samostalnosti pisanja rada

Izjavljujem da sam samostalno napisao ovaj rad.

Rijeka, srpanj 2023. Mateo Acinger

Zahvala

Velika zahvala mentoru doc. dr. sc. Goranu Mauši na podršci te usmjeravanju u izradi i pisanju ovog rada.

Sadržaj

Popis slika

Popis tablica

Poglavlje 1

Izvršavanje koda i potrebni alati

Programski jezik je set gramatičkih pravila koja računalu ili nekom drugom uređaju govore na koji način da se ponaša. Svaki programski jezik ima neke svoje ključne riječi i specijalnu sintaksu koja oblikuje instrukcije računalu [1]. Programski jezik je dizajniran da korisnik može komunicirati sa računalom preko instrukcija. Instrukcije računala su komande koje računalo može razumjeti i izvršiti. Sastoje se od niza bitova koji definiraju operaciju koja se treba izvršiti. Instrukcije su atomarne što znači da se izvršavaju kao cjelina koju nije moguće prekinuti između izvršavanja [2].

1.1 Podjela programskih jezika

Postoji više tipova programskih jezika. Prvi su programski jezici niske razine. Jezik niske razine je jezik koji je neovisan o arhitekturi računala i sadrži samo nule i jedinice. Računalo takve jezike izvršava bez kompajlera ili interpretera te ih izvršava jako velikom brzinom. Postoji i daljnja podjela jezika niske razine na strojni jezik i sklopovski jezik. Strojni jezik ili objektni jezik napisan je u binarnoj ili heksadecimalnoj formi i ne zahtijeva prevođenje jer ga računalo razumije. sklopovski jezik je jezik koji je dizajniran za

specifične procesore. Predstavlja set instrukcija u simbolima koji su u čovjeku razumljivoj formi. Takvi programi zahtijevaju korištenje assemlera za pretvorbu u strojni jezik. Prednost sklopovski jezika je ta što zauzima jako malo memorije i ima kraće vrijeme izvršavanja od jezika visoke razine. Assembler prevodi sklopovski jezik u binarni kod na jako kompleksan način. Potrebno je poznavati arhitekturu procesora i zatim pomoću dokumentacije i puno registara procesora vrši se postupak kodiranja u kojem se konkretni simboli pretvaraju u nizove nula i jedinica koji su zatim razumljivi računalu [3].

Jezici visoke razine su dizajnirani da pomoću korisnički prihvatljivih programa omoguće razvoj i kreiranje novih programa i web stranica. Da bi oni radili ispravno moraju koristiti kompajler ili interpreter da bi ih računalo moglo razumjeti. Prednost je lakše razumijevanje, pisanje i čitanje koda. Neki od najpoznatijih programskih jezika visoke razine su Python, Java, JavaScript, Julia, Perl, Swift, Ruby i mnogi drugi [4]. Jezici visoke razine mogu se podijeliti u četiri potkategorije [5].

- Proceduralni

  Proceduralni jezici su jezici koji su dizajnirani za izvođenje niza koraka i postupaka koji zahtijevaju slijedno izvođenje. Temelje se na konceptu proceduralnog programiranja te se programi pišu kao niz procedura ili funkcija. Uobičajeno podržavaju osnovne konstrukte kao što su grananja i petlje te neke osnovne strukture podataka. Najpoznatiji takav jezik je C [7].

- Funkcijski

  Funkcijski jezici se temelje na pozivima funkcija i na matematičkim izrazima. Glavne značajke takvih jezika su funkcijski pozivi gdje se funkcije tretiraju kao osnovne jedinice programiranja. U većini slučajeva glavni dio programa sadrži samo pozive te programi imaju malo vrijednosti koje se mijenjaju jer su najčešće vrijednosti definirane unutar samih funkcija. To svojstvo omogućava da se program provodi sa

manje grešaka nego inače te također omogućava češće korištenje rekurzije i struktura podataka kao što su stabla i liste. Najkorišteniji funkcijski jezici su Haskell, Lisp i Erlang [9].

- Skriptni

  Jezici koji omogućuju automatizaciju i olakšavanje zadataka zovu se skriptni jezici. Oni nude veoma dobre alate za obradu teksta, datoteka, slika i općenito za ponavljajuće zadatke. Zbog toga se najčešće koriste kod strojnog učenja i u umjetnoj inteligenciji. Njihova sintaksa je uobičajeno nešto lakša od drugih jezika pa je također dobar izbor prilikom povezivanja na baze podataka i razvijanja web aplikacija. Najpoznatiji skriptni jezici su Python, JavaScript, Perl, Bash i Ruby [6].

- Objektno-orijentirani

  Objektno orijentirani jezici su jezici koji se temelje na konceptu objekata. Objekt je instanca klase koja sadrži određene atribute i metode koje opisuju određeni podatak. Takvo programiranje omogućuje programerima bolju organizaciju koda te logičan i ponovo iskoristiv kod. U objektno orijentiranom programiranju postoje četiri osnovna koncepta.

  1. Apstrakcija

     Apstrakcija je teoretski pojam koji opisuje kako se kompleksni objekti mogu pojednostaviti tako da im se prezentiraju samo bitni podaci i funkcije. Najpoznatiji objektno orijentirani jezik je Java. Uz nju postoje još i C++, Python, Ruby, C# i ostali [8,10].

  2. Enkapsulacija

     Enkapsulacija je svojstvo koje osigurava da je podatak moguće izolirati od ostatka programa na način da je dostupan samo u svojoj klasi. Tako on postaje samo kontroliran pomoću metoda svoje klase.

3. Nasljeđivanje

   Nasljeđivanje je svojstvo koje omogućuje da se klase međusobno nasljeđuju te da na taj način klasa pokupi sve varijable i metode od druge klase.

4. Polimorfizam

   Polimorfizam je svojstvo koje omogućuje da se objekti ponašaju na različite načine ovisno o kontekstu za koji se koriste.

Uz ove četiri određene skupine postoje još stotine jezika koji se mogu koristiti za druge namjene te ih je teško razvrstati u neku od ovih specifičnih kategorija.

1.1.1 Usporedba jezika visoke i niske razine

Programski jezici visokih i niskih razina su fundamentalno vrlo različiti iz razloga što se izvršavaju na drugačije načine. U tablici 1.1 prikazane su osnovne razlike s obzirom na proces programiranja u tim jezicima. Čovjeku je prirodnije pisati kod u jeziku visoke razine iz razloga što mu je sintaksa sličnija svakodnevnom životu.

Visoki level	**Niski level**
Programerski orijentiran	Računalno orijentiran
Koristi kompajler ili interpreter	Koristi assembler
Prenosiv na druge uređaje	Neprenosiv
Lako razumljiv	Teško razumljiv
Lako se debugira	Teško se debugira
Koristi puno memorije	Koristi malo memorije

Tablica 1.1 Razlike između visokih i niskih programskih jezika

Općenito, ako je bitna brzina izvršavanja, većina programera će odabrati jezik niske razine iz razloga što su performanse znatno brže i bolje.

Postoje i jezici srednje razine koji imaju karakteristike visokih i niskih jezika. To znači

da je uz sve mogućnosti visokih jezika za jednostavno pisanje programa također moguće kontroliranje i upravljanje memorijom i ostalim hardverskim komponentama [4,11].

Razina jezika	**Primjer jezika**	**Opis**	**Primjer instrukcije**
Jezici visoke razine	Python, Visual Basic, Java, C++	Neovisni o hardveru. Prevedeni preko kompajlera ili interpretera. Jedna tvrdnja se manifestira u puno strojnih instrukcija.	plaćaPoSatu = 7.38 brojSati = 37.5 plaća = plaćaPoSatu * brojSati
Jezici niske razine	sklopovski jezik	Prevedeni koristeći assembler. Jedna tvrdnja se prevodi u jednu strojnu instrukciju.	LDA181 ADD93 STO185
	Strojni jezik	Izvodivi binarni kod koji napravi kompajler, interpreter ili assembler.	1010100011010101 0100100101010101

Tablica 1.2 Primjeri jezika zajedno s instrukcijama

U tablici 1.2 prikazan je tip jezika te određeni naziv pojedinih jezika za određeni tip. Uz to stoji i kratak opis u kojemu se vidi način na koji pojedini jezik prevodi kod u računalu razumljiv način. Uz sve to vidljiv je i prikaz instrukcija u kojemu se jasno vidi da je čovjeku jedino čitljiv kod u jeziku visoke razine.

1.2 Opis komponenti i njihovih karakteristika

Tijekom izvršavanja koda u većini koraka kod se izvršava pomoću računalni programskih alata. Od pisanja koda u nekom uređivaču teksta pa preko prevođenja u kompajleru ili interpreteru, kod se većinski izvršava u računalni programskim alatima. Kada se kod

prevede do instrukcija, tek onda ga računalo počinje čitati i to pomoću procesora. Procesor služi da bi zadani kod prikazao programeru na zaslonu i izvršio korisnički zahtjev.

Procesor je mozak računala i on je zaslužan za razumijevanje instrukcija. Procesor sadrži aritmetičko-logičku jedinicu koja izvršava matematičke i logičke operacije. Kontrolna jedinica je također dio procesora i ona je zaslužna za tumačenje uputa i upravljanja tokom podataka između različitih dijelova računala. Kada je kod spremljen u memoriju, procesor ga počinje obrađivati. Procesor ga obrađuje u nekoliko koraka.

1. Dohvaćanje instrukcije iz memorije: Prvu instrukciju programa iz memorije procesor dohvati i spremi u instrukcijski registar u procesoru. Registar je brza mala memorija koja se nalazi na procesoru ili na nekom mikrokontroleru te najčešće služi za pohranu privremenih podataka. Postoji više vrsta registara ovisno o namjeni. Većina procesora ima nekoliko registara opće namjene te registre za posebne namjene kao što su registri indeksa, pokazivača stoga ili programskog brojila. Procesor dohvati instrukciju na način da gleda adrese pohranjene u programskom brojilu. Programsko brojilo je registar u procesoru koji sadrži adresu sljedeće instrukcije koja se treba izvršiti. Na početku izvršavanja programa, operacijski sustav stavi prvu adresu u programsko brojilo te se onda krene izvršavati instrukcija po instrukcija.

2. Dekodiranje instrukcije: Upravljačka jedinica dekodira instrukciju i određuje koja operacija treba biti izvedena i nad kojim podacima. Postoje dva mjesta gdje su podaci pohranjeni, a to su: memorija i registri. To se događa na način da procesor pročita adresu iz programskog brojila i zatim u memoriji nađe instrukciju na toj adresi. Zatim procesor povećava vrijednost programskog brojila na sljedeću adresu te se trenutna instrukcija izvršava.

3. Izvršavanje trenutne instrukcije: Nakon što je uspješno dohvaćena, instrukcija se izvršava. Ovisno o arhitekturi procesora i računala, postoje razne instrukcije koje se mogu izvoditi. Složeniji procesori imaju razne metode predviđanja grana i opti-

mizacije izvođenja te na taj način poboljšavaju performanse. Osnovne instrukcije koje se nalaze u gotovo svim procesorima su:

- Aritmetičke instrukcije: Sastoje se od osnovnih matematičkih operacija kao što su zbrajanje, oduzimanje, množenje, dijeljenje.
- Logičke instrukcije: Instrukcije koje koriste osnovne sklopove digitalne logike kao što su I, ILI i isključivo ILI sklop.
- Usporedne instrukcije: Uspoređuju određene vrijednosti, ovisno o tome kakva je usporedba, mogu skakati s jedne adrese na drugu.
- Prijenosne instrukcije: Služe za premještanje podataka između dva registra.
- Kontrolne instrukcije: Koriste se ukoliko je u programu potrebno uvesti petlje, grananja ili prekide.

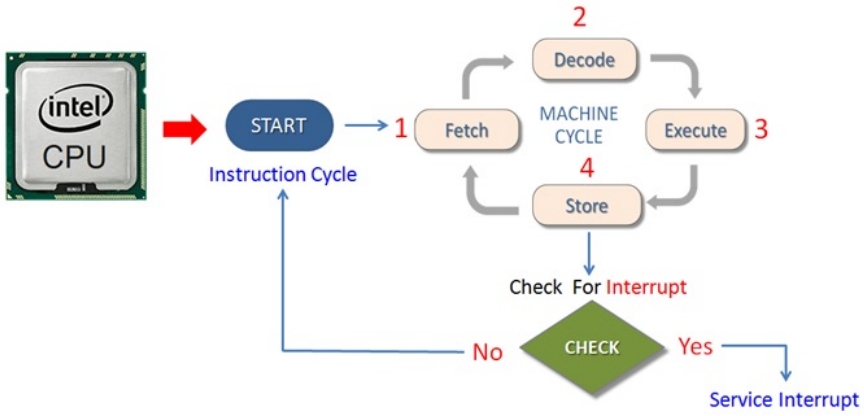
Procesor ponavlja postupak sve dok se ne dođe do prekidne točke u programu ili do kraja samog programa. Moguće je također da je prekidna točka upit prema korisniku za unos podataka ili greška u programu.

Periferni uređaji su povezani s procesorom preko posebnih hardverskih kontrolera za ulazno/izlazne operacije. Oni kontroliraju komunikaciju između procesora i perifernih uređaja. Podaci se spremaju u međuspremnik iz kojeg zatim procesor pomoću ulazno/izlaznih instrukcija čita te podatke te nastavlja s njihovim izvršavanjem.

Većina instrukcija ima podinstrukcije zbog različitih premještanja varijabli i podataka između registara. Svaka podinstrukcija je mikrooperacija koju procesor može izvršiti. Zbog toga instrukcija sama po sebi može izgledati jednostavno, no za njeno izvršavanje potrebno ju je još podijeliti na manje dijelove koje procesor može razumjeti.

Ovisno o tome koja je instrukcija dohvaćena iz memorije, procesor obrađuje instrukcije koristeći skup registara, cache memoriju i aritmetičko-logičku jedinicu. Kada je instrukcija izvršena, njen rezultat se sprema u neki vid memorije, najčešće registre ili memoriju

[15]. Slika 1.1 prikazuje redoslijed događanja i na koji način procesor obrađuje instrukcije. Također, na slici se vidi da unutar tog procesa postoje i prekidi, koji se potencijalno mogu javiti.



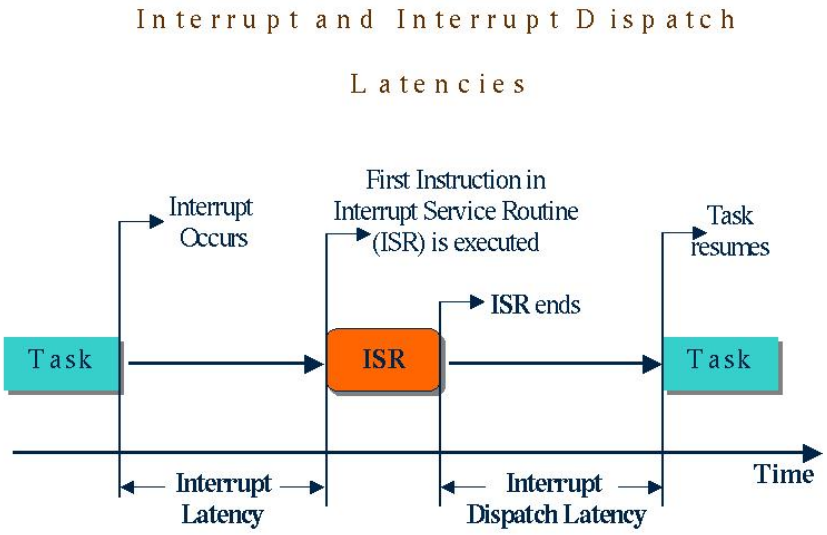
Slika 1.1 Proces izvršavanja instrukcija u procesoru.
Preuzeto iz Bibliografija slika [1]

1.3 Prekidi i prekidna rutina

Većina programa napisanih u nekom programskom jeziku služi za interakciju s korisnikom. Tijekom izvršavanja programa, procesor izvršava instrukcije te kada dođe do instrukcije koja zahtijeva unos, on čeka na ulazne podatke. Često se koriste instrukcije za ulazno/izlazne operacije koje služe za komunikaciju procesora s vanjskim uređajima kao što su miš i tipkovnica. Za komunikaciju s vanjskim uređajima, procesor može biti u dva načina rada: prekidi ili aktivno upravljanje. Korisnikov unos je jako spor u odnosu na brzinu rada procesora, te ukoliko bi procesor bio u načinu rada gdje aktivno čeka odgovor korisnika, gubio bi jako puno vremena i performanse bi bile jako loše. Zato se ovaj način ne koristi, nego jednostavno procesor ide dalje na izvršavanje trenutnih procesa u redu te

kada korisnik unese podatke, događa se prekid koji zatim procesor vidi te nastavlja dalje s njim.

Prekid je hardverski signal koji obavještava procesor o događaju koji zahtijeva njegovu pažnju. Procesor obrađuje prekid u prekidnoj rutini. Prekidna rutina je poseban dio programa u kojem se izvršavaju i obrađuju prekidi. Prekidna rutina se izvršava odmah nakon što procesor prekine trenutnu operaciju i prije nego što se vrati u prethodni program. Kada se prekidna rutina izvrši, procesor može obaviti potrebne operacije koje su u ovom slučaju čitanje ili pisanje podataka iz perifernog uređaja. Kada se ona završi, procesor se vraća na program koji je izvršavao prije prekida [14]. Slika 1.2 prikazuje redoslijed obrađivanja određenog procesa. Na grafu je prikazano putovanje kroz vrijeme na kojemu vidimo i prekidnu rutinu. Kada se prekidna rutina uspješno obavi, nastavlja se izvođenje obrade procesa.



Slika 1.2 Obrada prekida i prikaz prekidne rutine.
Preuzeto iz Bibliografija slika [2]

1.4 Računalno obrađivanje koda

Svaki kod prije nego što uopće dođe do arhitekture računala i procesora prolazi kroz prevođenje. Prevođenje koda je pretvorba koda napisanog u uređivaču teksta u neki oblik instrukcija koje računalo razumije. Postoje dva osnovna načina na koji se kod prevodi. To su interpretirani i kompajlirani način. Sve češće noviji jezici koriste hibridni način koji spaja interpretiranje i kompajliranje.

Interpretirani jezici su jezici koji se izvršavaju pomoću interpretera. To je računalni programski alat koji čita kod u stvarnom vremenu i izvršava ga. On analizira svaku liniju koda i odmah je izvršava nakon što je dohvati. To svojstvo omogućuje brzo pokretanje programa i jednostavno testiranje. Općenito, interpretirani programski jezici su sporiji iz razloga što se kod izvršava u stvarnom vremenu. Prednosti interpretera su u tome da, ukoliko se kod koristi na različitim uređajima, nema problema s prevođenjem tog programa iz razloga što se kod pomoću interpretera ne prevodi u cjelini, već linija po linija [16].Kompajlirani jezici su jezici koji se prije izvršavanja pretvaraju u strojni jezik pomoću kompajlera. Kompajler je računalni programski alat koji prevodi kod u jezik koji je razumljiv procesoru. Kompilacija se izvodi jednom prije pokretanja programa, što omogućuje brže izvršavanje koda. Ovisno o arhitekturi računala, strojni kod može imati prednost u performansama. Kompajler prolazi kroz tri faze. Prva faza je leksička faza u kojoj kompajler čita izvorni kod i dijeli ga na riječi i simbole. Sljedeća faza je sintaktička analiza, u kojoj se provjerava sintaksa koda i utvrđuje je li on ispravno napisan u skladu s gramatikom programskog jezika. Zadnja faza je semantička faza, u kojoj se provjerava ima li kod smisla u kontekstu programskog jezika. Ukoliko su sve faze uspješno završene, kompajler generira izvršni kod, koji je niz instrukcija strojnog jezika i može se izravno pokrenuti na računalu. Glavni nedostatak kompajlera je taj što se kod mora prevesti za svaku platformu na kojoj će se pokrenuti [17].

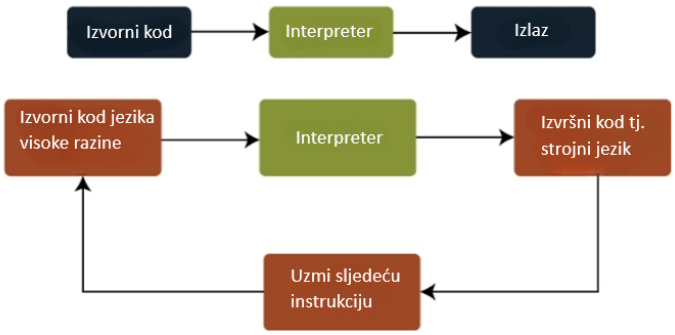
1.4.1 Razlike između kompajlera i interpretera

Glavna razlika između kompajlera i interpretera je način na koji se kod pretvara u strojni jezik. Kompajler prevodi cijeli kod odjednom, što omogućuje otkrivanje potencijalnih grešaka u fazi prevođenja. Kod koji se prevodi s interpreterom otkriva greške direktno na liniji na kojoj se greška nalazi kada ona dođe na red za izvršavanje [18]. U tablici 1.3 prikazane su osnovne razlike između kompajlera i interpretera.

Kompajler	**Interpreter**
Skenira kod i prevede ga u strojni jezik	Skenira i prevodi samo jednu tvrdnju
Koristi puno vremena za analizu koda	Koristi vrlo malo vremena za analizu koda
Cijeli proces izvršavanja je brži	Cijeli proces izvršavanja je sporiji
Generira objektni kod	Ne generira objektni kod
Teže se debugira jer skenira cijeli kod	Lako se debugira jer prevodi jednu liniju
Koristi puno memorije	Koristi malo memorije

Tablica 1.3 Razlike između kompajlera i interpretera

Slika 1.3 prikazuje grafički postupak izvršavanja koda te se jasno vidi da kompajler prevede cijeli kod dok s druge strane interpreter prevede jednu liniju pa ide natrag u izvorni kod po drugu i tako u krug dok ne dođe do kraja programa ili do prekida.



Slika 1.3 Obrada prekida i prikaz prekidne rutine.
Preuzeto iz Bibliografija slika [3]

Poglavlje 2

Izvođenje koda u popularnim programskim jezicima

Do sada je izvođenje koda bilo objašnjeno općenito bez konkretnih primjera kako to izgleda za pojedine jezike. U ovom poglavlju naglasak je na točnom izvođenju koda od izvornog pa sve do strojnog u jezicima kao što su Java, C++, Python i Julia. Za svaki od njih izvršavanje je jedinstveno i drugačije u odnosu jedni na druge.

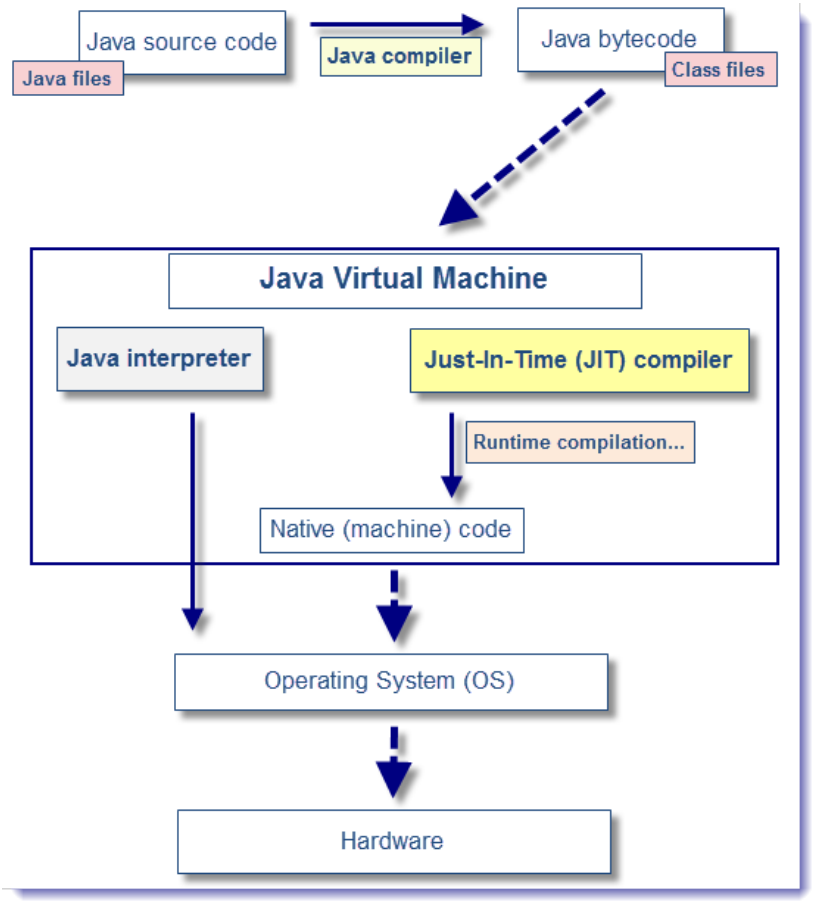
2.1 Obrada koda programskih jezikom Java

Java je programski jezik visoke razine. To znači da je pisanje koda omogućeno na način da sam kod ne ovisi o hardverskoj arhitekturi računala. Takvi programski jezici fokusiraju se na logiku programa te hardverske komponente stavljaju u drugi plan. Da bi se kod napisan u jeziku visoke razine mogao izvršiti, potrebno ga je prevesti u strojni jezik koji računalo razumije. Da bi se to moglo napraviti, računalu je potreban kompajler ili interpreter koji će to napraviti. Jednom kada se kod prevede u strojni jezik, moguće ga je pokrenuti koliko god puta je potrebno, ali to mora biti pokrenuto na računalu na kojem

je prevedeno. To znači da ukoliko se kod prevede, tj. kompajlira, na jednom računalu, nije ga moguće pokrenuti na drugom računalu te ga je potrebno ponovno prevoditi na računalu na kojem će se koristiti.

Tvorci Java programskog jezika izabrali su da će se kod izvršavati kao kombinacija interpretera i kompajlera. Kod se prevodi u strojni jezik za računalo koje fizički ne postoji, već je to računalo virtualno. To virtualno računalo zove se Java virtual machine, a strojni jezik za kod zoves se Java bytecode. Općenito, bytecode je objektni kod koji kompajler prevodi u binarni strojni jezik tako da ga računalo može čitati. Ime bytecode potječe od seta instrukcija koje imaju jedan bit te zatim slijede opcionalni parametri. Upravo zbog toga Java je jezik koji se danas koristi na više od tri bilijuna uređaja. Zbog Java virtual machine, kod napisan na jednom računalu može se koristiti na drugim uređajima bez ponovnog prevođenja. Sve što računalu treba je interpreter za Java bytecode.

Java virtual machine koristi JIT (just-in-time) kompajler i Javac kompajler. JIT omogućuje ubrzano izvršavanje bytecode-a i pretvara ga u binarni kod koji računalo razumije. On radi na način da kombinira interpretacijski i kompajlirani način te uzima prednosti jednog i drugog. On generira strojni kod u zadnji trenutak prije nego se izvršava. Postoji mogućnost da se JIT isključi te bi se tada kompletni bytecode prevodio pomoću interpretera. Oba načina rada prikazana su na slici 2.1 na kojoj se vidi kompletan postupak izvođenja. Cijeli postupak je isti u oba načina samo je taj korak drugačiji. Javac je kompajler koji prevodi Java izvorni kod u bytecode. Unutar JVM postoje razne dodatne funkcije kao što su garbage collection. Ta funkcija automatski upravlja oslobađanjem memorije koju program ne koristi, kao i mjerama sigurnosti od zlonamjernih računalni programa. JVM radi na način da klasa ClassLoader učitava bytecode u memoriju, a zatim JIT prevodi bytecode u strojni kod računala. JVM ima svoju memoriju za pohranu koja se dijeli između svih programa [19].



Slika 2.1 Kompletan prikaz izvođenja koda.
Preuzeto iz Bibliografija slika [4]

Osim načina izvršavanja Java koda koji je već spomenut, moguće je i samo prevođenje Java koda pomoću kompajlera ili interpretera na svakom pojedinačnom računalu, no takvim pristupom gubi se na performansama. Java bytecode interpreter je manje kompleksan i jednostavan program za izvršavanje strojnog jezika, dok je kompajler vrlo kompleksan. Također, na tržištu postoji veliki broj različitih uređaja koji ne moraju nužno imati kompleksan kompajler za prevođenje Java koda. Java kod u kombinaciji s Java bytecode-om omogućuje platformsku neovisnost, internetsku kompatibilnost i sigurnost.

Veliki dio prenošenja Java koda u današnje vrijeme odvija se putem interneta, gdje

korisnik direktno preuzima program na svoj uređaj. Da bi se uspješno preuzela aplikacija putem interneta, potrebno je imati Java Development Kit (JDK) okruženje na računalu. JDK sadrži potrebne alate za kompiliranje i pokretanje Java koda. Sastoji se od tri glavne komponente. Prva komponenta je kompajler koji prevodi kod u Java bytecode, što omogućuje korištenje tog koda na bilo kojem uređaju koji ima Java virtualnu mašinu. Također postoji i Java runtime okruženje koje je odgovorno za uspješno izvršavanje i korištenje Java aplikacije. Treća komponenta je razvojno okruženje u kojem programer piše i razvija kod. Java Development Kit je dostupan na svim operacijskim sustavima [18, 19, 25].

2.2 Obrada koda programskih jezikom C++

C++ je programski jezik visoke razine koji se koristi za razvoj računalni programskih aplikacija. C++ je jezik koji je potekao od C jezika. U odnosu na C ima novije koncepte kao što su objektno orijentirana paradigma, generičko programiranje i programiranje sa šablonama. Poznat je i po performansama jer omogućuje izravno upravljanje hardverskim resursima računala. Programi su brzi i učinkoviti. Performanse omogućuju optimalni način za upravljanje hardverskim resursima kao što su procesorski registri, memorija i slično. Programer ima mogućnost manualnog alociranja i realociranja memorije te također može i osloboditi memoriju.

Prilikom izvršavanja koda, kod treba prevesti u računalu razumljiv jezik. Prvi korak u izvršavanju koda radi predprocesor koji obrađuje i manipulira izvorni kod prije nego što kompajler krene s prevođenjem. Predprocesorske naredbe služe kako bi se kod mogao kompajlirati prema nekim kriterijima te ako je potrebno dodati određene datoteke u kod. Kompajler prevodi kod u strojni jezik koji računalo razumije. Postoje koraci po kojima se vrši prevođenje.

1. Kod prolazi kroz leksički analizator koji ga pretvara u niz leksema ili tokena. Tokeni su jedinice koda koje predstavljaju ključne riječi i simbole.

2. Zatim kod prolazi kroz sintaktički analizator koji provjerava je li kod gramatički ispravan prema pravilima jezika.

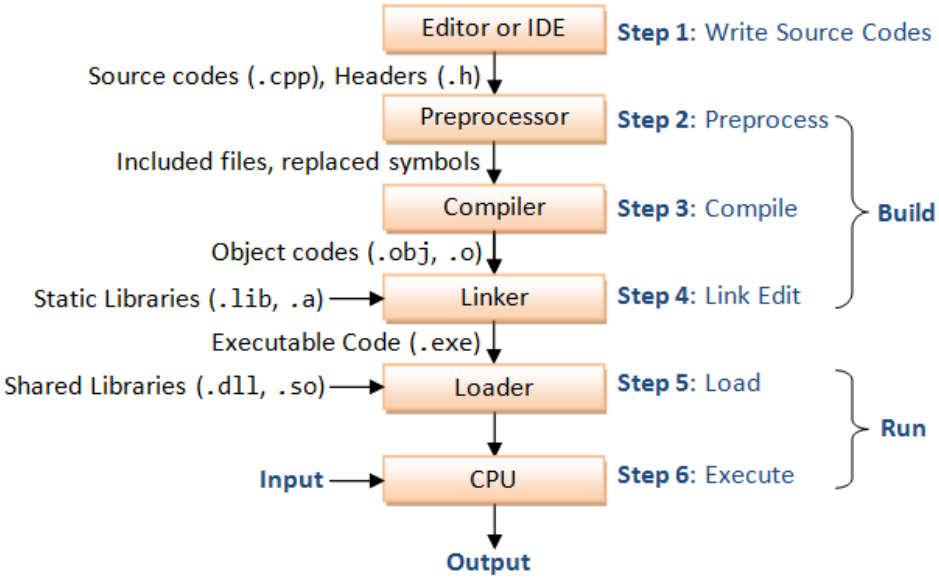
3. Zadnja faza je semantički analizator koji provjerava ponaša li se kod u skladu sa jezikom i logikom. Semantički analizator provjerava tipove podataka, pravila nasljeđivanja, domenu varijabli itd. Ako je kod prošao uspješno kroz sve faze, generira se strojni kod koji sadrži instrukcije, ali nije još kompletni program.

Da bi bio kompletni program, potrebno je koristiti povezivač. Povezivač je program koji se koristi u procesu generiranja strojnog jezika. On ima ulogu povezivanja objektnih datoteka koje su kreirane u tri faze kompajliranja u jednu datoteku koja se može pokrenuti na računalu. Uz to služi i kako bi povezao vanjske knjižnice koje vrlo često olakšavaju kompleksnije zadatke prevođenja.

Nakon što ih poveže u jednu datoteku, poziva se program koji učitava tu datoteku i zapisuje ju u memoriju. On radi na način da prvo čita izvršnu datoteku s diska i učitava ju u memoriju. Zatim u memoriji dijeli memorijska mjesta ovisno o tome koji dio programa mora ići u koji dio memorije. Popunjava tablice simbola koje su ostale neriješene u fazi povezivanja. Tablica simbola je tablica koja pokazuje informacije o simbolima koji korišteni u programu. Simboli se koriste kao identifikatori varijabli, funkcija i drugih oznaka. Svaki simbol se sastoji od imena, tipa, adrese i drugih bitnih informacija. Vrlo su korisne kod kodova niskih levela jer omogućuju bržu dijagnostiku problema i uvid u način izvršavanja programa. Zatim učitavač inicijalizira sve varijable i postavlja početne vrijednosti te poziva sve konstruktore ukoliko je to potrebno.

Nakon svih ovih koraka vraća se na početak programa i počinje sa izvršavanjem programa. Učitavač je najčešće automatiziran od strane operacijskog sustava i programeri

ne moraju obraćati pažnju na njega. Najpoznatiji C++ kompajleri su GNU Compiler Collection (GCC), Clang i Microsoft Visual C++ [21,22].



Slika 2.2 Kompletan prikaz izvršavanja koda.
Preuzeto iz Bibliografija slika [5]

Slika 2.2 grafički prikazuje cijeli postupak izvršavanja koda u C++-u. Također, prikazuje i koje ekstenzije se koriste za pojedine datoteke u tom postupku. Uz sve to prikazano je i koji se koraci koriste prilikom prevođenja, a koji prilikom pokretanja programa.

2.3 Obrada koda programskih jezikom Python

Python je programski jezik visoke razine koji je sve zastupljeniji u svim dijelovima ljudskih života. Osnovna namjena Python-a bila je kreirati jezik koji ljudi mogu koristiti iako nemaju programersku pozadinu. To omogućuje jako jednostavna sintaksa jezika zajedno s puno dostupnih knjižnica i objašnjenja kako se one koriste. U današnje vrijeme

koristi se svugdje, ali najčešće kod problema gdje se mora baratati s velikim količinama podataka i kod strojnog učenja.

Često se može pročitati da je Python jezik koji kod prevodi pomoću interpretera ili preko kompajlera te zapravo nije jasno definirano na koji način Python zapravo prevodi kod. Kod se prevodi koristeći interpreter i kompajler, no korištenje kompajlera je svedeno na minimum. Općenito, Python je jezik kod kojeg se kod prevodi liniju po liniju. To je odlika interpretiranog jezika. Izvorni kod napisan u nekom od uređivača teksta se prvo prevodi u bytecode umjesto u strojni kod kao kod jezika kao što je `C++`. Za to je zaslužan Python interpreter koji vrši postupak kompajliranja. Zatim se taj bytecode može koristiti na bilo kojoj platformi koja posjeduje Python virtualnu mašinu. Ovakav pristup se koristi najviše iz razloga što to omogućuje kodu da je platformski nezavisan i da se može vrlo jednostavno koristiti na više fizičkih uređaja. Osim toga, prednost je i u tome što se ne mora definirati tip varijable u kodu te je ta zadaća prepuštena interpreteru, umjesto programeru, kao što je to u većini drugih jezika [23].

Interpreter koji prevodi izvorni kod u bytecode u Pythonu je nazvan CPython i napisan je u C-u. Postoje i druge vrste, kao što su PyPy ili Jython. Kada se kod krene izvršavati, izvršava se u tri faze.

- Inicijalizacija
- Kompajliranje
- Interpretiranje

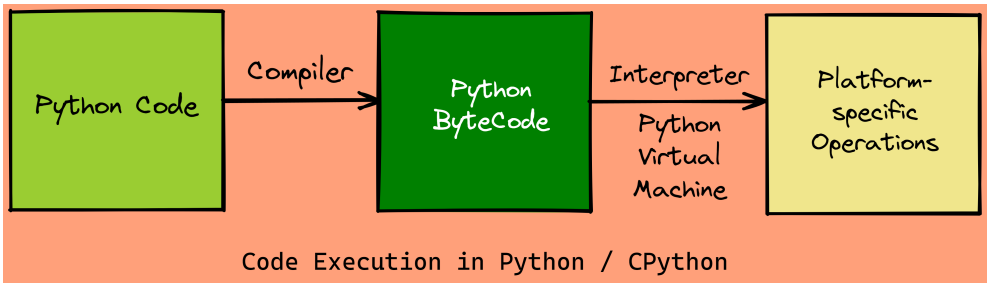
Tijekom faze inicijalizacije, CPython kreira određene strukture podataka koje omogućuju da se kod pokrene. Zatim se izvorni kod prevede u sintaksu apstraktnih stabala, što inače rade svi kompajleri, te se iz tog razloga kaže da Python koristi i kompajler. Onda se iz tih kreiranih stabala dolazi do bytecode-a. Virtualna mašina je sastavni dio CPython-a. Ona koristi stog pomoću kojeg uzima jednu po jednu liniju te je stavlja na

stog i prima povratne podatke. To sve omogućuju instrukcije kao što su `LOAD_FAST`, `LOAD_CONST`, `RETURN_VALUE` i `BINARY_ADD`. Za prepoznavanje varijabli, referenci na određene lokacije i kada koristiti koje funkcije, virtualna mašina posjeduje nekoliko osnovnih koncepata.

- Objekt koda
- Objekt funkcije
- Okviri

U objekt koda spremaju se informacije koje govore na koji način kod radi te varijable i osnovne strukture. Funkcije se spremaju u objekt funkcija umjesto u objekt koda iz razloga što uz varijable i ostale atribute, funkcije imaju svoje ime te određene argumente i povratne vrijednosti. Za to sve su zaslužne instrukcije koje to izvode. Okviri omogućuju stanje u kojem se objekt koda može izvršiti. Za svaki zaseban objekt koda postoje zasebni okviri. Svaki okvir ima pokazivač na prijašnji, što onda tvori svojevrsni stog. Na taj način Python virtualna mašina čita i izvršava bytecode.

Uz sve to, osim bytecode-a, učitavaju se i moduli određenih knjižnica, te zatim Python runtime pretvara takav kod u strojni jezik koji računalo razumije. Python runtime je dio Python virtualne mašine koji je zaslužan za izvršavanje i pokretanje aplikacija. Interpreter prevodi bytecode u strojni kod koji Python runtime izvrši. Na taj način bilo koji Python kod se može pokrenuti na bilo kojem računalu, jer kada se kreira bytecode, Python virtualna mašina zajedno s Python runtime-om taj bytecode čita i prevodi na svakom uređaju zasebno, te se tako postiže platformska nezavisnost Python programa [24, 25]. Na slici 2.3 vidi se grafički prikaz cijelog postupka izvršavanja koda zajedno s alatima koji rade određeni korak. Prikaz je specifično napravljen za CPython obradu.



Slika 2.3 Izvršavanje koda u Pythonu.
Preuzeto iz Bibliografija slika [6]

2.4 Obrada koda programskih jezikom Julia

Julia je dinamički programski jezik visoke razine koji je razvijen s ciljem da bude lagan za korištenje i da ima intuitivnu sintaksu. Osim toga, glavna motivacija pri razvijanju ovog jezika bila je potreba za brzim izvršavanjem programa. Julia se najčešće koristi prilikom rješavanja matematičkih problema i za rad s velikim skupovima podataka. Sintaksa je vrlo slična Pythonu i MATLAB-u te je iz tog razloga vrlo jednostavna za razumijevanje. Pruža vrlo bogat skup alata i biblioteka za rad sa statistikom, algebrom te numeričkom analizom općenito. Julia ima ugrađenu podršku za paralelno izvršavanje, što znači da je vrlo lako moguće stvoriti više niti za izvršavanje programa i na taj način dodatno optimizirati performanse.

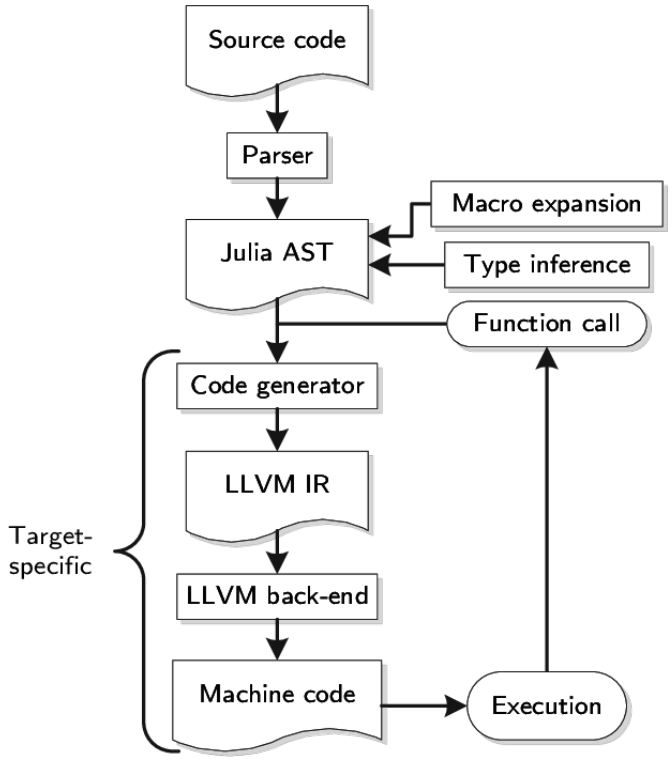
Julijin kompajler se razlikuje od interpretera koji se koristi u Pythonu, pa performanse ne moraju biti najbolje na početku korištenja jezika. Općenito, tvorci jezika očekuju da će se u većem dijelu izvršavanja postići performanse slične C++ kodu. Da bi se postigle takve performanse, potrebno je slijediti određene korake i pravila pri pisanju Julia programa. Neki od tih zahtjeva su da se kritični kod mora pisati unutar funkcija, treba izbjegavati globalne varijable bez definiranih tipova podataka te mnoge druge smjernice koje se mogu pronaći u službenoj Julia dokumentaciji. Ovi zahtjevi su važni jer Julijin

kompajler radi na određeni način, te je otežano i sporije izvršavanje koda koji ne prati ta pravila.

Za izvršavanje programa koristi se JIT (Just-in-Time) kompilacija, koja radi na drugačiji način nego JIT u Javi. Prvo Julijin interpreter prolazi kroz izvorni kod i koristi sintaktnu analizu kako bi razumio strukturu koda i pretvorio ga u apstraktno sintaksno stablo. Zatim se vrši inferencija tipova, gdje se analiziraju tipovi varijabli i izraza kako bi se omogućilo efikasnije generiranje strojnog koda. Generiranje strojnog koda obavlja JIT, koji koristi LLVM (Low Level Virtual Machine). LLVM je kolekcija alata i biblioteka za generiranje, optimizaciju i izvršavanje statički i dinamički tipiziranog koda. Općenito, on služi kao backend koji generira strojne instrukcije za izvršavanje programa. Kada je generiran strojni kod, on se izvršava izravno na računalu[27].

Prema ovim koracima, može se zaključiti da Julia koristi kombinaciju kompiliranja i interpretiranja, s tim da JIT kompilacija pretvara izvorni kod u strojni kod. Osim izvršavanja koda, Julia pruža i mogućnost paralelizma. U pozadini, Julia koristi `THREAD` modul, koji je dio biblioteke za implementaciju paralelizma pomoću niti. Zadužen za dodjeljivanje poslova određenoj niti je Julia interpreter. Koriste se mehanizmi sinkronizacije kako bi se osiguralo ispravno izvršavanje niti. Unatoč tome, programer treba biti oprezan pri korištenju više niti i pratiti smjernice koje su dostupne u Julia službenoj dokumentaciji[28].

Na slici 2.4 može se vidjeti grafički prikaz tijeka izvođenja Julia programa. Na grafu su prikazane sve već opisane komponente i njihov ispravan redoslijed.



Slika 2.4 Izvršavanje koda u Juliji.
Preuzeto iz Bibliografija slika [7]

Poglavlje 3

Opisi i rezultati mjerenja

Mjerenja su napravljena na nekoliko različitih algoritama koji sadrže različite algoritamske složenosti. Algoritmi su pisani na način da su u svakom programskom jeziku oni u potpunosti isti te je razlika jedino u sintaksi jezika, a ne u samom radu algoritma. Iz tog razloga mjerenja su optimalna jer se mjeri identična stvar u različitim programskim jezicima. Ulazni setovi podataka također su jednaki te je za svaki ulaz pojedinog algoritma provedeno mjerenje deset puta te se uzima prosječna vrijednost tih deset mjerenja.

3.1 pyRAPL alat za mjerenje performansi koda

pyRAPL je alat za mjerenje energetske potrošnje procesora koji je temeljen na Intelovoj RAPL (Running Average Power Limit) tehnologiji. Općenito RAPL je dio Intel-ove arhitekture procesora koja omogućava programski pristup mjerenju i upravljanju energetskom potrošnjom procesora. Pomoću RAPL-a moguće je mjeriti performanse raznih dijelova procesora uključujući DRAM, GPU i u radu korištenu procesorsku utičnicu (CPU socket).

pyRAPL je napisan i koristi sintaksu programskog jezika Python te radi na način da

očitava spremljena mjerenja i očitane vrijednosti iz sistemske datoteke. Za korištenje ovog alata potrebno je Linux okruženje te se ne može koristiti na Windowsima. Na početku rada potrebno je u terminal upisati naredbu `sudo chmod -R a+r /sys/class/powercap/intel-rapl` koja rješava problem s dozvolama te se zatim RAPL može neometano koristiti. Prilikom mjerenja performansi Python koda nema problema te se dodatne mogućnosti i detalji mogu pronaći na službenoj pyRAPL web stranici. Ukoliko se koristi za očitavanje nekog drugog koda kao što je na primjer C++ kod potrebno je pronaći odgovarajući Python omot te pokrenuti postupak kompajliranja pomoću podprocesa te zatim pozvati prevedeni kod za koji će se izmjeriti energetska i vremenska potrošnja.

3.2 Algoritam složenosti O(n)

Ulazni set podataka za mjerenje ovog algoritma je bila varijabla n koja se odnosi na duljinu niza. Cilj zadatka je bio da program nasumično kreira niz koji sadrži objekte klase čovjek. Objekti klase čovjek imaju nekoliko atributa uključujući i atribut OIB koji može poprimiti nasumičnu vrijednost između 0 i 10000. Program je trebao pomoću funkcije linearnog pretraživanja pronaći objekt čiji je OIB jednak nasumično odabranom OIB-u na početku izvršavanja programa. Ukoliko bi ga pronašao vratio bi indeks na kojem se mjestu u nizu nalazi taj objekt, a ukoliko ga ne bi pronašao vratio bi da broj nije pronađen u nizu. Ovaj zadatak prikazuje sljedeći pseudokod:

Pseudokod O(N)

```
Klasa Covjek:
    Metoda init(oib):
        Spremi oib u instancu klase
```

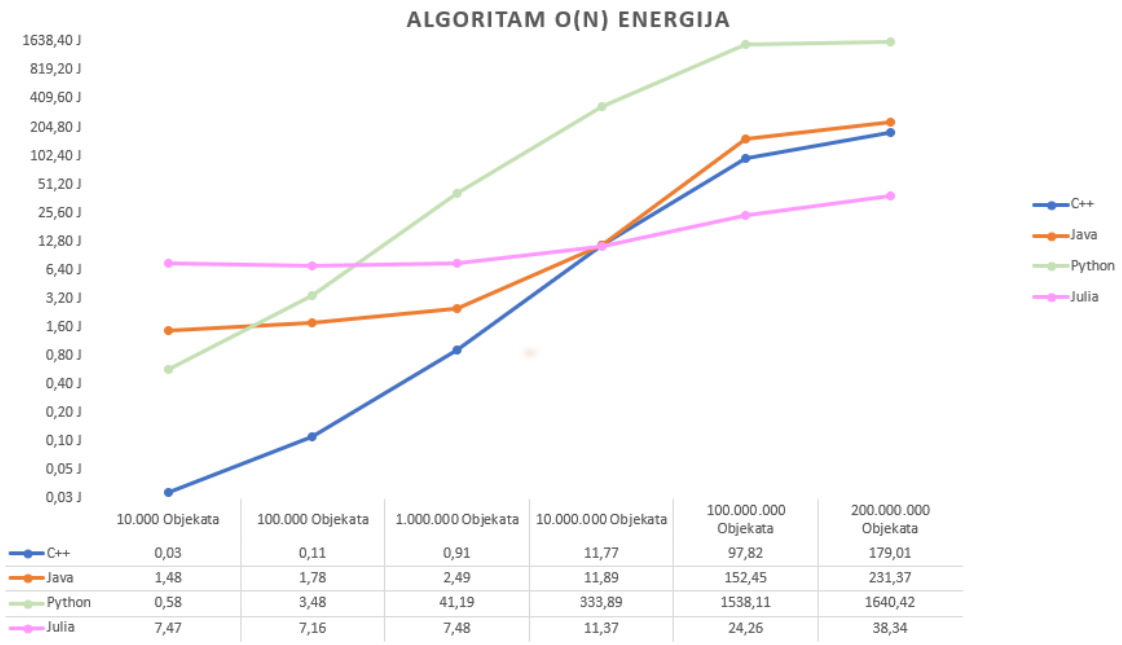
Pseudokod O(N)

```
Klasa LinearSearch:
    Staticka metoda linear_search(arr, target):
        Inicijaliziraj potrebne varijable
        Petlja za svaki i u rasponu od 0 do duljine arr:
            Ako arr[i].oib jednak target:
                Postavi indeks na i
                Postavi pronaden na True
        Vrati indeks

if je glavni program jednak "main" then
    Inicijaliziraj n na ulaznu vrijednost i slucajni OIB
    Inicijaliziraj arr s objektima Covjek sa slucajnim OIB-ovima
    Izvrsi linear_search nad arr s target
    Ispisati rezultat
end if
```

Algoritam složenosti O(n) znači da se vremenski i prostorni resursi povećavaju proporcionalno s vremenom. Kada se govori o algoritamskim složenostima često se čuju pojmovi najbolji, najgori i prosječni slučaj. U ovom slučaju svi slučajevi su složenosti O(n) iz razloga što neovisno o tome je li pronašao OIB ili ne program je napravljen da prođe kroz čitav niz. Ovo nije optimalno rješenje no daje točnija mjerenja te bolju usporedbu. Ovo se postiže na način da for petlja u klasi LinearSearch prolazi od indeksa 0 sve to duljine niza.

Na slici 3.1 grafički su prikazani rezultati mjerenja energije. Ispod grafičkog prikaza nalazi se tablica ulaznih podataka i konkretni rezultati mjerenja. Energija je mjerena u J(Džulima). Zbog vrlo velikih oscilacija mjerenja grafički prikaz je napravljen na logari-

tamskoj skali.



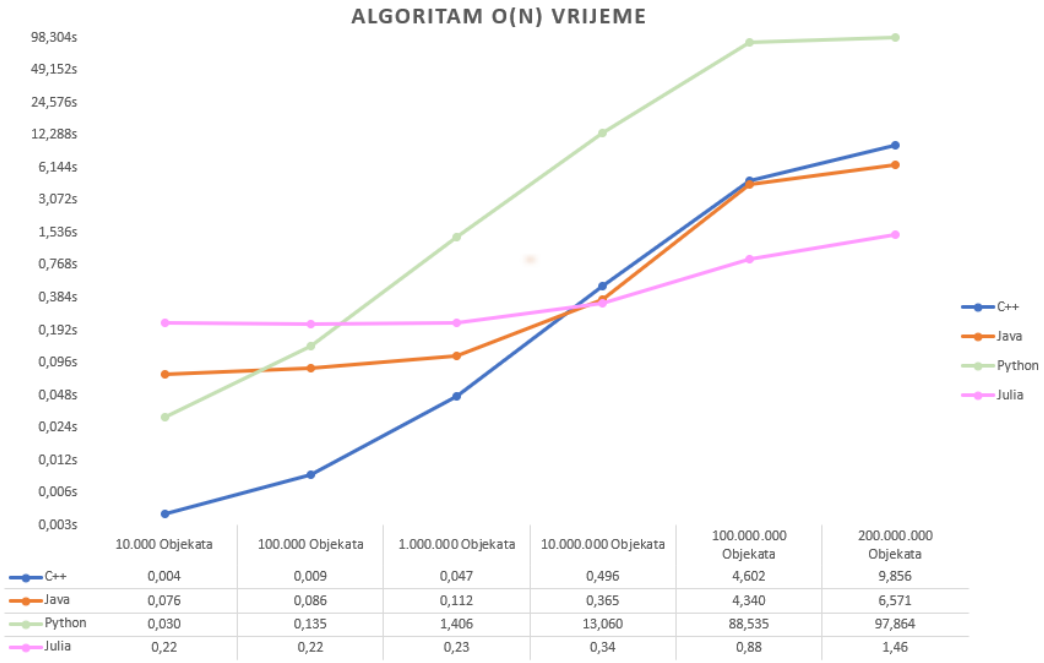
	10.000 Objekata	100.000 Objekata	1.000.000 Objekata	10.000.000 Objekata	100.000.000 Objekata	200.000.000 Objekata
C++	0,03	0,11	0,91	11,77	97,82	179,01
Java	1,48	1,78	2,49	11,89	152,45	231,37
Python	0,58	3,48	41,19	333,89	1538,11	1640,42
Julia	7,47	7,16	7,48	11,37	24,26	38,34

Slika 3.1 Algoritam složenosti O(n) energija

Grafički prikaz pokazuje kako skoro svim jezicima potrošnja energije raste s porastom duljine niza. Potrošnja energije kod Julije je znatno manjih vrijednosti nego kod drugih jezika kod vrlo velikih ulaznih podataka. To nije u potpunosti očekivani ishod iz razloga što bi Julia trebala biti otprilike brza kao C++, no ne i ovako znatno brža. Ovakav rezultat dogodio se zbog načina rada Julia kompajlera. Julia kompajler ima ugrađenu podršku za vektorske operacije i indeksiranje nizova što je vrlo pogodno za ovakav program koji koristi Linear Search algoritam. Također, moguće je primijetiti da su Java i C++ sličnih performansi te da Python očekivano troši puno energije. Zanimljivo je za primjetiti kako kod najmanjih ulaza Java troši najviše energije. To se događa iz razloga što su Python i C++ kompajleri optimizirani kako bi brže obrađivali mali skup ulaznih podataka.

Slika 3.2 prikazuje grafički prikaz rezultata mjerenja vremena kod ovog algoritma.

Vrijeme je izmjereno u sekundama. Prikaz također sadrži i tablicu ulaznih podataka te detaljno napisane vrijednosti izmjerenih vremena. Kod mjerenja vremena također se može primjetiti linearan rast u odnosu na ulazne veličine. Iz već opisanih razloga Julia je najbrža a Python najsporiji. Rezultati Python-a jako odskaču u odnosu na druge jezike, no to je očekivano ponašanje zbog interpretiranog načina rada. Zanimljivo je primjetiti kako su Java i C++ opet vrlo sličnih performansi. Kod malih ulaza C++ je znatno brži od Jave zbog svog kompajlera. Kada se promatraju vrlo velike ulazne veličine Java je nešto brža od C++-a. Iz toga se može zaključiti kako ne mora uvijek kompajlirani jezik biti brži od interpretiranog jezika.



	10.000 Objekata	100.000 Objekata	1.000.000 Objekata	10.000.000 Objekata	100.000.000 Objekata	200.000.000 Objekata
C++	0,004	0,009	0,047	0,496	4,602	9,856
Java	0,076	0,086	0,112	0,365	4,340	6,571
Python	0,030	0,135	1,406	13,060	88,535	97,864
Julia	0,22	0,22	0,23	0,34	0,88	1,46

Slika 3.2 Algoritam složenosti O(n) vrijeme

Uspoređivanjem izgleda grafova energije i vremena može se vidjeti da su krivulje vrlo sličnog izgleda. Kod ovog algoritma energija i vrijeme oboje rastu proporcionalno te iz tog razloga dolazi do sličnosti. Energetski najučinkovitiji program ne mora uvijek biti i najbrži. Energija i brzina nisu međusobno zavisne varijable te je moguće da dolazi do

oscilacija[1]. Oscilacije u ovom primjeru nisu prevelike, no vidljive su ako se gledaju Java i C++. Na prijašnjim slikama kod vrlo velikih ulaznih podataka može se primjetiti da je Java brža u odnosu na C++ koji je energetski učinkovitiji.

Svako mjerenje je ponovljeno deset puta te su sva mjerenja unutar dopuštenih granica. Granica je +/- 5% u odnosu na aritmetičku sredinu. Iz tog razloga ne postoje određeni zaključci koji bi opisivali taj segment mjerenja.

3.3 Algoritam složenosti O(N * log N)

Ulazni set podataka za program složenosti O(N * log N) bila je varijabla N. Varijabla predstavlja broj objekata klase Student u nizu. Klasa Student ima nekoliko atributa, no najvažniji je broj dresa koji se dodjeljuje prilikom punjenja niza objektima. Svaki student nasumično dobije broj dresa veličine između 1 i 10. Zatim kada postoji niz sa objektima klase Student poziva se iterativni Quicksort algoritam koji sortira sve objekte prema broju dresa. Osim samog sortiranja radi se i podjela studenata u dva tima: crveni i plavi tim. Cilj programa je da oba tima imaju isti broj studenata te da su zbrojevi svih dresova težinski isti u oba tima. Kao izlaz programa napisani su brojevi članova svakoga tima te zbroj svih dresova. Što se tiče same složenosti ona je O(N * log N) u najboljem i u prosječnom slučaju dok je u najgorem slučaju moguće da je O(N * N). Ovakva složenost u najgorem slučaju moguća je jedino ako se odabere najgori pivot u QuickSort algoritmu. U samoj implementaciji programa pivot je uvijek dobro odabran tako da nikada u praksi neće doći do najgoreg slučaja prema složenosti. Općenito ovakva izvedba programa s QuickSort-om može biti optimizirana, no za potrebe mjerenja je dobra jer se uvijek prolazi i sortira cijeli niz te je pivot uvijek na istom mjestu. Pseudokod navedenog programa izgleda ovako:

Pseudokod O(N * log N)

Klasa Student:

Metoda ___init___(id, jerseyNumber):

Spremi id i broj dresa u instancu klase

Funkcija quickSort(arr, low, high):

Inicijaliziraj prazan stog

Dok god stog nije prazan:

Pop element s vrha stoga

Ako low < high:

Pushaj potrebne elemente u stog i zovi partition

Funkcija partition(arr, low, high):

Petlja za svaki j u rasponu od low do high:

Ako arr[j].getJerseyNumber() < pivot.getJerseyNumber():

Zamijeni arr[i] s arr[j]

Zamijeni arr[i + 1] s arr[high]

Vrati i + 1

Petlja za svaki i u rasponu od ulazne varijable:

Dodaj novog Studenta s id-jem i i nasumicnim brojem dresa u listu students

Izvrsi quickSort nad listom students, od indeksa 0 do indeksa len(students) - 1

Pseudokod O(N * log N)

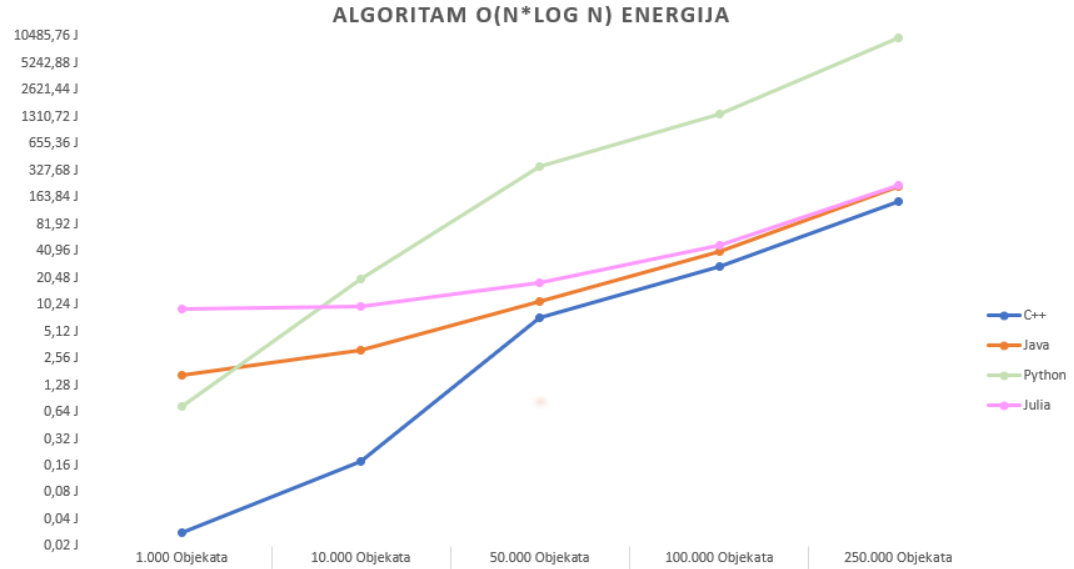
```
Petlja za svaki i u rasponu od 0 do len(students):
    Rasporedi studenta u tim

Ispisati rezultat
if je glavni program jednak "main" then
    Inicijaliziraj n na ulaznu vrijednost i slucajni OIB
    Inicijaliziraj arr s objektima Student sa slucajnim OIB-ovima
    Izvrsi linear_search nad arr s target
    Ispisati rezultat
end if
```

U glavnom dijelu programa postoji nekoliko glavnih petlji. Petlja za punjenje niza objektima koja je složenosti O(N), quickSort koji u sebi sadrži while složenosti O(N) zajedno sa ugnježdenim for-om koji u svakoj iteraciji smanjuje pomoću pivota listu na pola. Zbog toga je taj dio složenosti O(log N) što da je quickSort-u ukupnu složenost O(N * log N). Nakraju postoji i petlja koja raspoređuje objekte po timovima te je ona također O(n). Sveukupno ovakva raspodijela petlji donosi ukupnu složenost ovog algoritma na O(N * log N).

Složenost O(N * log N) znači da vremenska i energetska složenost raste proporcionalno s umnoškom broja N i njegovog logaritma. Ovakav rast je nešto sporiji nego O(n^2), ali je brži od O(N). Ovakva složenost često se javlja u algoritmima sortiranja, pretraživanja i podjele podataka[26].

Na slici 3.3 grafički je prikaz mjerenja energije. Prikaz također koristi logaritamsku skalu zbog velikog odstupanja vrijednosti. Energetske vrijednosti Jave, C++-a i Julije su vrlo slične na svakoj ulaznoj vrijednosti. Može se primjetiti kako prilikom vrlo malih ulaza C++ troši zanemarivo malo energije dok Julia pri vrlo malim ulazima troši otpri-

like isto energije kao i za srednje velike ulaze. Što se tiče Pythona može se primjetiti kako je Python brži od Jave i Julije na vrlo malim ulazima. To je zbog optimiziranog Python interpretera. Kada su u pitanju veće vrijednosti potrošnja energije u Pythonu je u prosjeku veća čak 50 puta nego kod ostala tri jezika. Ovo se događa zbog načina na koji Python radi. Moguće optimizacije bi se mogle napraviti kod korištenja globalnih i statičkih varijabli te bi trebalo pripaziti i na dijeljenje jednog velikog niza na više malih. Kada bi se optimizirao, Python program bi trošio manje energije, ali svejedno puno više od ostalih jezika.

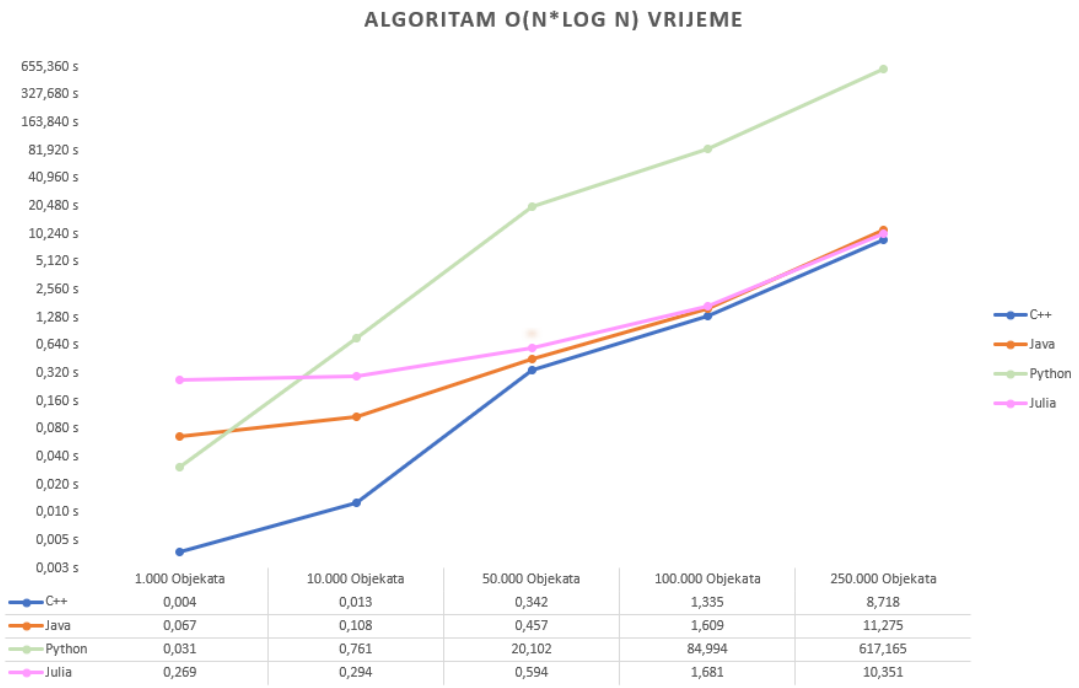


	1.000 Objekata	10.000 Objekata	50.000 Objekata	100.000 Objekata	250.000 Objekata
C++	0,03	0,18	7,37	27,66	148,41
Java	1,68	3,19	11,03	39,99	215,14
Python	0,75	19,67	358,76	1409,06	10101,09
Julia	9,06	9,67	17,85	48,40	227,60

Slika 3.3 Algoritam složenosti O(N * log N) energija

Na slici 3.4 nalazi se grafički prikaz mjerenja vremena izvršavanja. Zanimljivo je za primjetiti vremena izvršavanja kod vrlo malih ulaznih podataka. Uz C++, Python je nabrži dok su Java i Julia nešto sporije, no to je sve zanemarivo mala razlika jer se radi o vrijednostima manjima od jedne sekunde. Kod većih ulaznih podataka Java, C++ i Julia imaju vrlo slična vremena izvršavanja no C++ je ipak malo brži. Python iz

već spomenutih razloga jako uspori te je za izvršavanje Python programa na najvećem ulaznom setu podataka potrebno prosječno 62 puta više vremena nego kod ostalih.



	1.000 Objekata	10.000 Objekata	50.000 Objekata	100.000 Objekata	250.000 Objekata
C++	0,004	0,013	0,342	1,335	8,718
Java	0,067	0,108	0,457	1,609	11,275
Python	0,031	0,761	20,102	84,994	617,165
Julia	0,269	0,294	0,594	1,681	10,351

Slika 3.4 Algoritam složenosti O(N * log N) vrijeme

Ovo je odličan primjer koji pokazuje kako energija i vrijeme izvršavanja ne moraju uvijek biti isti. Energija se u prosjeku podigla 50 puta dok vrijeme čak 62 puta. Kod ostalih programa krivulje su vrlo slične te se svejedno u većini slučajeva porast energije manifestira i u proporcionalnom porastu vremena. Svako mjerenje je ponovljeno deset puta te su sva ta mjerenja unutar dopuštenih granica. Granica je +/- 5% u odnosu na aritmetičku sredinu. Iz tog razloga ne postoje određeni zaključci koji bi opisivali taj segment mjerenja.

3.4 Algoritam složenosti O(n^2)

Ulazni set podataka za ovaj program je varijabla n koja prikazuje veličinu dvodimenzionalnog niza. Kada se kao ulaz pošalje određena varijabla n stvara se dvodimenzionalni niz veličine n^2. Zatim se taj niz puni objektima klase čovjek gdje se prilikom kreiranja objekta da nasumično odabrani broj između 1 i 100. Cilj programa je da se spiralno sortira dvodimenzionalni niz prema nasumično odabranim brojevima svakog objekta. Sortiranje se radi na način da se kreće iz gornjeg lijevog kuta matrice te se zatim ide niz zadnji stupac i tako dalje dok se ne dođe do središnjeg elementa u sredini matrice. Kao izlaz programa ispisuje se novo sortirana matrica. Ovako izgleda pseudokod opisanog programa:

Pseudokod - O(n^2)

Klasa Covjek:
    Metoda ___init___(broj):
        Spremi broj u instancu klase

Funkcija spiral_order(matrix):
    Dok god left $\leq$ right i top $\leq$ bottom:
        Petlja za svaki i u rasponu od left do right + 1:
            Dodaj matrix[top][i].get_broj() u res
        Povecaj top za 1
        Petlja za svaki i u rasponu od top do bottom + 1:
            Dodaj element u res
        Smanji right za 1
        Ako left $\leq$ right i top $\leq$ bottom:
            Petlja za svaki i u rasponu od right do left - 1 s korakom -1:

Pseudokod - O(n^2)

```
            Dodaj element u res
        Smanji bottom za 1
        Petlja za svaki i u rasponu od bottom do top - 1 s korakom -1:
            Dodaj element u res
        Povecaj left za 1
    Vrati res

if je glavni program jednak "___main___" then
    Inicijaliziraj matrice

    Petlja za svaki i u rasponu od 0 do n:
        Petlja za svaki j u rasponu od 0 do n:
            Stvori novi objekt Covjek s nasumicnim brojem izmedu 1 i 100

    Petlja za svaki i u rasponu od 0 do n:
        Petlja za svaki j u rasponu od 0 do n:
            Ispisati matrix[i][j].get_broj() i razmak

    Ispitaj funkciju spiral_order() za matricu matrix
    Ispisati result
end if
```

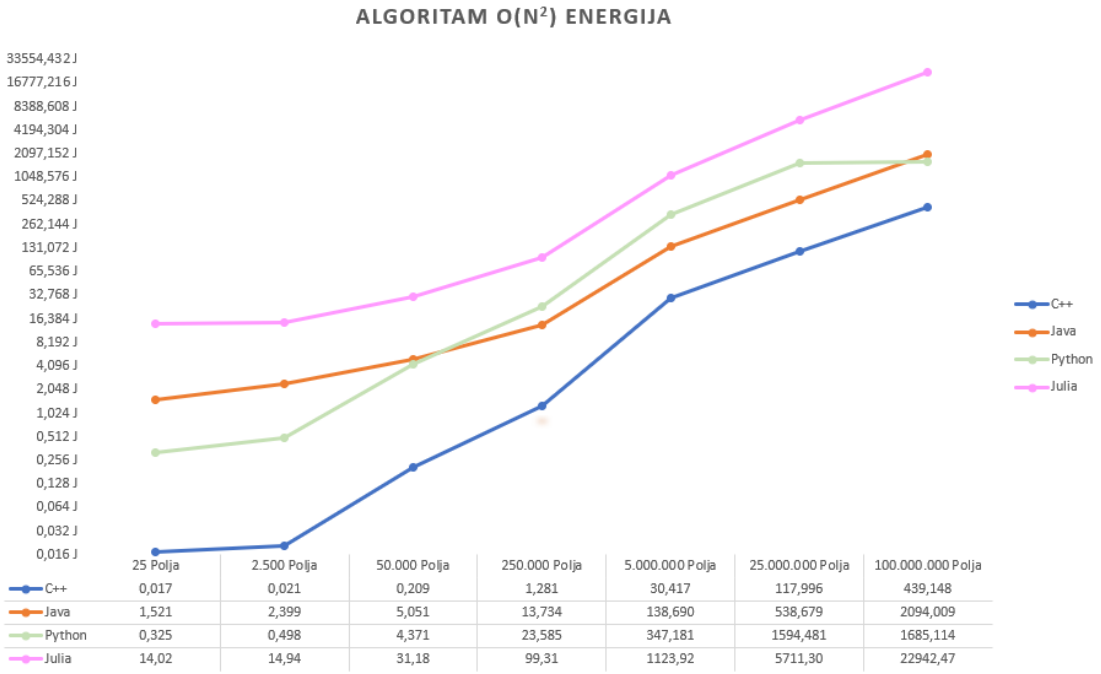
Složenost ovog programa je O(n^2) u svim slučajevima jer neovisno o veličini matrice svaki će element biti posjećen barem jednom te je minimalna složenost za prolazak i zamjenu elemenata u dvodimenzionalnom nizu O(n^2). Iz pseudokoda vidljivo je da se iz funkcije spiral_order vraća tek kada se prođe kroz svaki element dvodimenzionalnog niza. Moguće je da je složenost O(1) ukoliko se kao ulazni parametar da n koji je veličine

1. Takvi slučajevi neće se razmatrati te je složenost sa sigurnošću uvijek O(n^2). U main dijelu programa postoje dvije ugnježdene petlje koje daju spomenutu složenost. Osim njih već spomenuta funkcija za prolazak kroz niz sastoji se od while petlje sa for petljom unutra što također daje složenost O(n^2).

Općenito složenost O(n^2) znači da vremenska i energetska komponenta rastu zajedno s kvadratom ulazne varijable n. Osobito treba voditi računa prilikom provedbe ovakvih algoritama jer se jako brzo povećaju matrice te dolazi do velikog problema s efikasnošću. Program nije optimiziran nego je napravljen na ovaj način ne bi li se što zornije vidjelo ponašanje svakog jezika u normalnim ne optimiziranim programima.

Na slici 3.5 može se vidjeti grafički prikaz energetskih mjerenja ovog programa zajedno sa tablicom koja detaljno pokazuje rezultate samog mjerenja. Odmah pri prvom gledanju grafa može se zaključiti kako su krivulje skoro identične u svim jezicima samo je promjena u vrijednostima na y osi. C++ je očekivano najučinkovitiji zbog toga što je najniži jezik po hijerarhiji. U ovom programu po prvi put se vidi da C++ i Java zapaženije variraju u vrijednostima. Također, zanimljivo je za primjetiti rezultat Julije prilikom sortiranja najveće matrice. Rezultat je skoro 11 puta veći nego kod ostalih jezika. To je zapravo očekivano ponašanje jer u pravilima korištenja Julia jezika striktno je naglašeno kako u mainu dijelu programa ne smiju biti petlje. U ovom programu u main dijelu su dvije petlje. Jedna petlja puni početnu matricu objektima a druga ispisuje sortiranu matricu. Iz tog razloga ovako velika odstupanja su očekivana. Osim Julije zanimljivo je da Java troši više energije od Pythona prilikom mjerenja na najvećoj matrici. Općenito za zadnja dva ulaza Python potrošnja energije je otprilike identična i to se događa zbog izvedbe Python interpretera.

Na slici 3.6 prikazani su rezulati mjerenja vremena za ovaj program zajedno s tablicom koja detaljnije prikazuje vrijednosti tih mjerenja. Kod rezultata mjerenja također se jasno mogu vidjeti identičnosti u krivuljama s drugim vrijednostima isto kao i kod energije. Julia je iz već opisanih razloga najsporija te kada bi se provela ispravna optimizacija

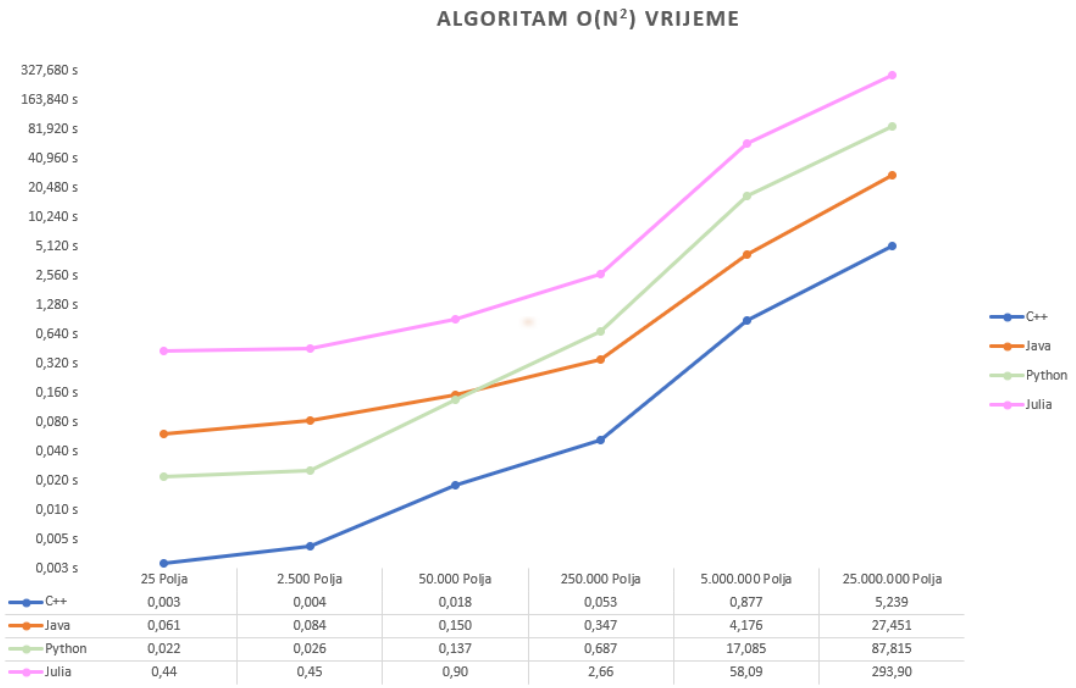


	25 Polja	2.500 Polja	50.000 Polja	250.000 Polja	5.000.000 Polja	25.000.000 Polja	100.000.000 Polja
C++	0,017	0,021	0,209	1,281	30,417	117,996	439,148
Java	1,521	2,399	5,051	13,734	138,690	538,679	2094,009
Python	0,325	0,498	4,371	23,585	347,181	1594,481	1685,114
Julia	14,02	14,94	31,18	99,31	1123,92	5711,30	22942,47

Slika 3.5 Algoritam složenosti O(n^2) energija

rezultati bi bili puno bolji. Python je pri malim veličinama matrice brži od Jave, no kod velikih je Java ipak brža. Iako Java troši više energije nego Python što se vidi iz slike 3.5 brža je skoro 3 puta od Pythona što je svakako zanimljiv rezultat mjerenja. C++ je najučinkovitiji i najbrži jezik te je na najvećoj matrici skoro 6 puta brži od Jave.

Teško je zorno prikazati sve jezike optimalno jer bi se na taj način mogle dogoditi različitosti u izvršavanju. Također, ako se uspoređuju složenost samog algoritma sa rezultatima može se vidjeti kako krivulje rastu kvadratno te da su krivulje oštrijeg kuta nego u prijašnjim algoritmima. To je malo teže za prikazati jer se koristi logaritamska skala. Još jedan primjer nepovezanosti energije i vremena izvođenja vidi se u primjerima za Javu i Python. Java troši više energije nego Python, ali je zato i brža pa bi ovo itekako trebalo uzeti u obzir prilikom odabira jezika za rad. Također, kao i prije pošto se uvijek prolazi kroz cijelu matricu nema odstupanja od granica aritmetičke sredine te su



	25 Polja	2.500 Polja	50.000 Polja	250.000 Polja	5.000.000 Polja	25.000.000 Polja
C++	0,003	0,004	0,018	0,053	0,877	5,239
Java	0,061	0,084	0,150	0,347	4,176	27,451
Python	0,022	0,026	0,137	0,687	17,085	87,815
Julia	0,44	0,45	0,90	2,66	58,09	293,90

Slika 3.6 Algoritam složenosti O(n^2) vrijeme

sva mjerenja unutar dopuštenih odstupanja od +/- 5%.

3.5 Algoritam složenosti O(2^n)

Ulazni podatak za program koji predstavlja ovaj algoritam je varijabla n koja označava duljinu pina. Program napočetku radi pin duljine n koji se sastoji od nasumično odabranih znamenki 0 i 1. Kada se generira pin program Brute force metodom pretražuje znamenke sve dok ne pronađe odgovarajući pin. Na početku prvo pretražuje po jednu znamenku pa zatim dvije i tako dalje prođe kroz sve kombinacije. Kada pronađe odgovarajući pin napiše izlaznu poruku kako je pronašao pin. Program svejedno prođe kroz sve moguće kombinacije iz razloga kako bi se moglo bolje usporediti različite jezike. Ovo nije optimizirano rješenje jer program bespotrebno pretražuje kombinacije bez cilja. Takvo rješenje prikazano je pseudokodom:

Pseudokod O(2^n)

```
Funkcija generate_pin(length)
Petlja i u rasponu od pin_length
Dodaj slucajno odabranu znamenku iz mogucih znamenki u pin
Return pin
Inicijalizacija pocetnih i pomocnih stringova
for i in raspon od 2^(pin_length×duljina_mogucih_znamenki) do
  for j in raspon od pin_length do
    Odredi digit_index kao (i/(2^(j×duljina_mogucih_znamenki)))
  end for
  Provjeri je li pin pronaden
end for
Ispisi rezultat
```

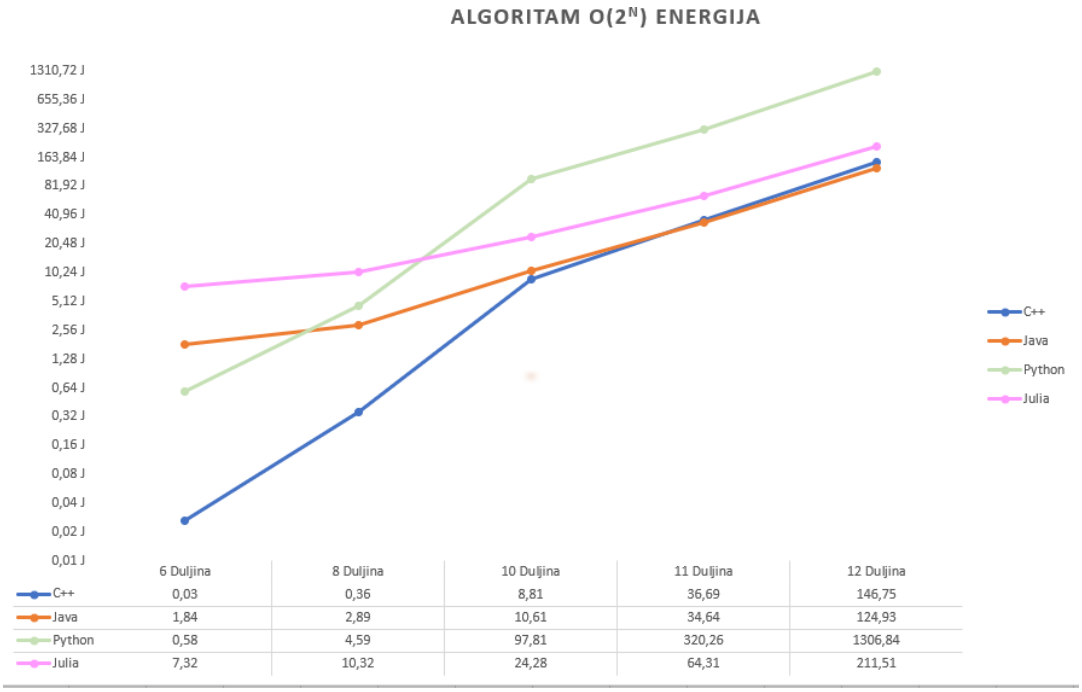
Složenost algoritma O(2^n) znači da energetska i vremenska potrošnja rastu eksponencijalno te da se u svakoj iteraciji gdje se poveća n broj 2 toliko puta pomnoži sam sa

sobom. Iz pseudokoda i funkcije findPin vidljivo je da se kroz petlju prolazi na način da se u svakom koraku iteracije povećava eksponent. Zbog te for petlje ukupna složenost ovog koda je $O(2^n)$. Ovakvu složenost bi u praksi trebalo izbjegavati iz razloga što se jako brzo složenost poveća te su energetska i vremenska potrošnja vrlo visoke te troše puno resursa.

Algoritam koji se mjeri u svim slučajevima ima složenost $O(2^n)$ zato što čak i ako pronađe pin iz prvog pokušaja ne izlazi iz funkcije nego prolazi kroz sve ostale kombinacije. Na ovaj način mjerenja su točnija. U tom slučaju da se izlazi iz funkcije složenost bi bila O(1) ,a u najgorem slučaju ostaje ista jer svejedno mora proći kroz sve kombinacije.

Na slici 3.7 prikazani su rezultati mjerenja energije i tablica sa detaljnim izmjerenim vrijednostima. Energija je izmjerena u J. Zanimljivost ovog mjerenja je već viđena mogućnost Python programa da bude energetski učinkovit za male ulaze te jako neučinkovit za velike ulaze. C++ je ponovno najučinkovitiji za male ulazne vrijednosti i to je u prosjeku učinkovitiji skoro 20 puta od drugog Pythona. Java je za najveće ulaze najefikasnija i čak je efikasnija od C++-a što je vrlo zanimljiv rezultat ovog mjerenja. Osim C++-a i Jave tu je i Julia koja je malo manje efikasna, ali i dalje vrlo blizu tim jezicima dok Python odskače te je sporiji 7 puta od ostalih jezika za velike ulaze. Ukoliko se pogleda način rada Java interpretera ovaj rezultat čak i nije toliko neočekivan. Java virtualna mašina ima dodatnu razinu apstrakcija koja pomaže kod algoritama s visokom složenošću. Također Java garbage collector skuplja neiskorištene resusrse i objekte iz memorije te na taj način omogućava brže izvođenje programa di se resusrsi eksponencijalno skupljaju[19].

Na slici 3.8 mogu se vidjeti vremena izvođenja ovog algoritma te tablica sa detaljnim vrijednostima u sekundama. Krivulje vremena izgledaju isto kao i energije samo s drugim vrijednostima na y osi. Java je najbrža dok su C++ i Julia nešto sporiji. Python je najsporiji za velike ulaze, no opet je vrlo brz kod malih ulaza zbog načina rada Python optimiziranog interpretera. Kod Julije se opet može primjetiti kako za male ulaze treba čak 44 puta više vremena za izvođenje nego C++-u. S druge strane za najveći ulaz Julia

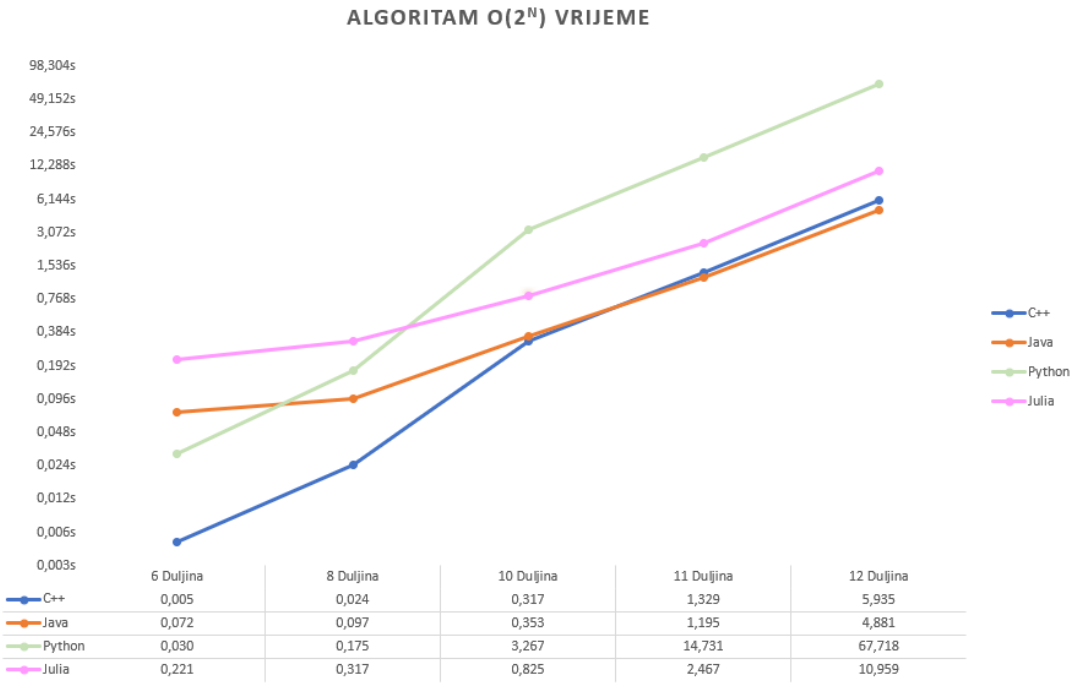


	6 Duljina	8 Duljina	10 Duljina	11 Duljina	12 Duljina
C++	0,03	0,36	8,81	36,69	146,75
Java	1,84	2,89	10,61	34,64	124,93
Python	0,58	4,59	97,81	320,26	1306,84
Julia	7,32	10,32	24,28	64,31	211,51

Slika 3.7 Algoritam složenosti $O(2^n)$ energija

je sporija samo 2 puta od C++ što je očekivani rezultat iz već opisanih razloga. Također, može se primjetiti i to da u ovom programu ne postoje petlje u main dijelu te su Julia performanse odmah znatno bolje te su vrlo konkurentne Javi i C++-u što nije bio slučaj u prethodno opisanom algoritmu.

Ukoliko bi se promatrao odnos energije i vremena na ovom algoritmu moglo bi se zaključiti da porastom energije dolazi do istog porasta u vremenu iz razloga što kod velikih ulaznih vrijednosti odnosi energije i vremena su isti. Ovo je vrlo zanimljiv rezultat jer kod nekih prethodnih algoritama ovo nije bio slučaj. Iako energija i vrijeme nisu vezane varijable vrlo često prate jedna drugu te se porastom jedne isto toliko ili slično povećava i druga.



	6 Duljina	8 Duljina	10 Duljina	11 Duljina	12 Duljina
C++	0,005	0,024	0,317	1,329	5,935
Java	0,072	0,097	0,353	1,195	4,881
Python	0,030	0,175	3,267	14,731	67,718
Julia	0,221	0,317	0,825	2,467	10,959

Slika 3.8 Algoritam složenosti O(2^n) vrijeme

3.6 Algoritam složenosti N!

Ulazni podatak za program koji opisuje ovaj algoritam je varijabla N koja opisuje duljinu string-a. Program kada primi ulaznu varijablu kreira nasumično odabrani string. U taj string mogu dospjeti sva moguća slova engleske abecede. Zatim kada postoji kreiran početni string vrši se ispis svih mogućih permutacija tog string-a. Permutacija stringa znači da se traže sve moguće kombinacije slova koje su moguće za ispisati od tog string-a. Program radi na način da se permutacije pretražuju rekurzivno. Kao izlaz ispisuju se sve permutacije. U najboljem slučaju ovaj program bi mogao imati složenost O(1) kada bi N bio duljine 1. Ukoliko je N nekakve veće vrijednosti algoritam će u srednjem i u najgorem slučaju imati složenost N!. Iz pseudokoda se vidi da složenost daje funkcija find_permutation kojoj se u svakoj iteraciji mijenja vrijednost varijabli l i r. Pošto je

funkcija rekurzivna ovakva implementacija daje složenost O(N!).

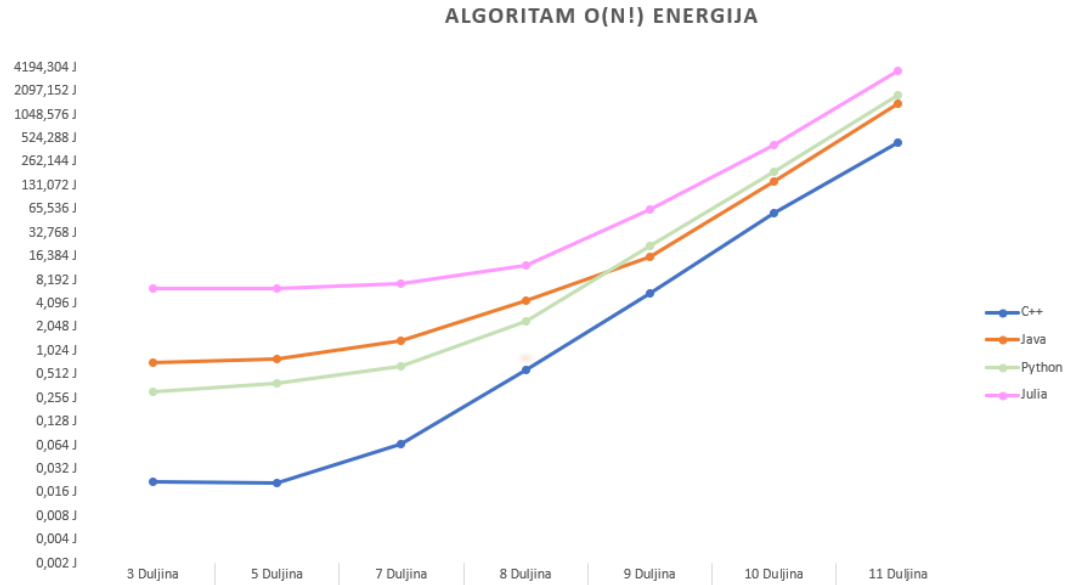
Pseudokod - N!

```
for i u rasponu od 1 do n do
    Postavi randStringArr[i] na nasumicno odabrano slovo iz slova
end for
Pozovi funkciju FIND_PERMUTATIONS s argumentima randStringArr, 0 i n − 1
if l je jednak r then
    Ispisati spojeni niz elemenata iz arr
else
    for i u rasponu od l do r + 1 do
        Zamijeni arr[l] s arr[i]
        Pozovi funkciju FIND_PERMUTATIONS s argumentima arr, l + 1 i r
        Zamijeni arr[l] s arr[i]
    end for
end if
if je glavni program jednak "___main___" then
    Pozovi funkciju MAIN()
end if
```

Složenost N! predstavlja faktorijelni rast gdje se gleda umnožak svih brojeva od 1 do N. Ovakva složenost je daleko najgora promatrana složenost. Programi koji imaju ovakvu složenost trebaju se detaljno analizirati te pronaći optimalnije rješenje. Osim toga rekurzivni način izvođenja ovakvog programa je također ne praktičan iz razloga što se jako puno puta poziva unutarnja funkcija te se vrlo brzo troše resusrsi i puni se stog s tim pozivima[26].

Na slici 3.9 se vidi grafički prikaz mjerenja energije zajedno s tablicom koja prikazuje detaljne vrijednosti mjerenja. Grafički se odmah može primjetiti kako krivulje imaju dosta veći rast nego li u prijašnjim algoritmima što je logično zato što je ova složenost

daleko najneefikasnija. Osim toga konkretna mjerenja su očekivana te C++ ponovno troši manje energije od ostalih dok su Python i Java relativno sličnih rezulata. Python je također brži za male ulaze zbog svog interpretera. Pošto u main dijelu programa postoji petlja u kojoj se stvara string duljine N Julia program je energetski najzahtjevniji. Promatraju li se rezultati za najveći ulaz može se primjetiti kako nijedan program ne odskače kao u prijašnjim primjerima te kako su mjerenja uobičajenih vrijednosti.

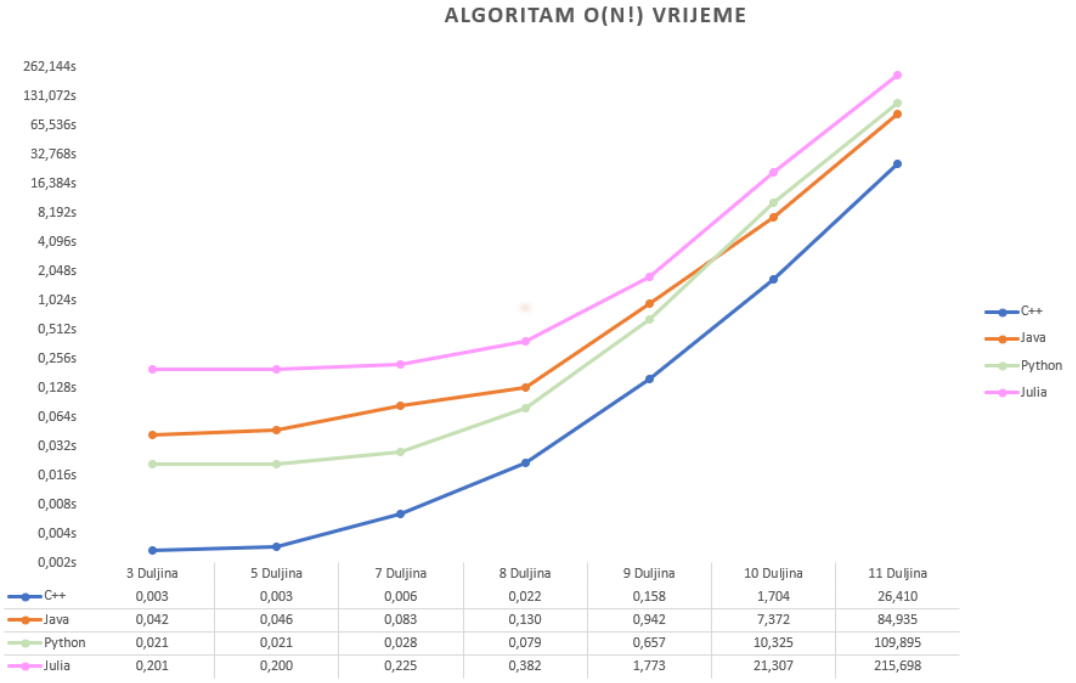


	3 Duljina	5 Duljina	7 Duljina	8 Duljina	9 Duljina	10 Duljina	11 Duljina
C++	0,022	0,021	0,066	0,589	5,511	58,738	460,145
Java	0,725	0,813	1,387	4,465	15,965	149,482	1425,574
Python	0,309	0,397	0,663	2,486	22,262	196,703	1864,334
Julia	6,365	6,481	7,411	12,834	64,310	430,529	3869,916

Slika 3.9 Algoritam složenosti O(N!) vrijeme

Promatraju li se vremenske vrijednosti mjerenja može se primjetiti da su krivulje otprilike istog izgleda no drugih vrijednosti. Slika 3.10 nam prikazuje upravo to zajedno s tablicom detaljnih vrijednosti za svako mjerenje. Kod svih programa vidi se nagli porast vremena izvršavanja za velike vrijednosti te se između ulaza duljine 9, 10 i 11 vidi vrlo veliki skok. Za najveći ulaz C++ je najbrži i brži je skoro 3 puta od Jave. C++ je najbrži iz razloga što je vrlo nizak jezik po hijerarhiji i na taj način može bolje i kvalitetnije upravljati memorijom i resursima koji su kod ovog programa vrlo bitni.

Python mjerenja su neočekivano malih vrijednosti u odnosu na kakva su mjerenja bila kod prijašnjih algoritama. To se događa iz razloga što Python interpreter ima posebnu podršku za reguliranje zbivanja na stogu što je vrlo korisno kod ovakvih rekurzivnih programa koji imaju puno poziva iste funkcije.



	3 Duljina	5 Duljina	7 Duljina	8 Duljina	9 Duljina	10 Duljina	11 Duljina
C++	0,003	0,003	0,006	0,022	0,158	1,704	26,410
Java	0,042	0,046	0,083	0,130	0,942	7,372	84,935
Python	0,021	0,021	0,028	0,079	0,657	10,325	109,895
Julia	0,201	0,200	0,225	0,382	1,773	21,307	215,698

Slika 3.10 Algoritam složenosti O(N!) vrijeme

Ovakva izvedba algoritma nije pretjerano dobra, no omogućava konkretnija mjerenja jer se uvijek prolazi kroz sve moguće permutacije i na taj način se stvara podloga za usporedbe. Iako su krivulje na logaritamskoj skali te se ne može vidjeti stvari kut pod kojem rastu može se svejedno primjetiti da je rast puno veći nego kod ostalih. Osim toga ovo je prvo mjerenje kod kojeg ni jedan jezik ne odskače pretjerano s performansama. Tome uvelike pridonosi i složenost ovog algoritma. Ukoliko se vrijeme i energija promatraju zajedno može se primjetiti kako su porasti otprilike istih veličina. Tako da se iz toga može zaključiti da kod ovakve implementacije porastom vremena izvršavanja otprilike isto toliko raste i energija što je vrlo korisno za znati ukoliko se rade mjerenja

s vrlo velikim ulaznim podacima. Pošto svaki program prolazi kroz sve permutacije sva provedena mjerenja su unutar granica aritmetičke sredine te se iz tog segmenta ne mogu napraviti konkretni zaključci koji bi ukazivali na nekakve anomalije.

3.7 Prikaz učinkovitosti i trajanja između programa

Po završetku mjerenja svi dobiveni rezultati prikazani su u odnosu na druge rezultate kako bi se dobila jasnija slika o provedenim mjerenjima. Tablica 3.1 prikazuje koliko su puta jezici u prosjeku lošiji od najučinkovitijeg jezika za taj algoritam. Najučinkovitiji program za svaki algoritam ima pridodijeljenu vrijednost 1. Na ovaj način se može jasno zaključiti kako je C++ u prosjeku najučinkovitiji jezik. Osim toga ovakav prikaz zornije i jasnije prikazuje podatke te se može vidjeti kako su u projesku Julia i Python za pojedine algoritme vrlo neučinkoviti. Podaci su dobiveni na način da su se zbrojile sve vrijednosti za pojedini jezik i podijelile sa brojem ulaznih setova podataka. Ovakav način prikaza ima mane zbog toga što sve vrijednosti svih ulaznih podataka imaju istu težinu. Za detaljnije i još jasnije zaključke pojedinim ulaznim podacima bi se mogla dodijeliti težina. Na taj način mogu se provesti jasniji zaključci ovisno treba li se računati učinkovitost za velike, srednje ili male setove podataka.

	O(n)	O(N * log N)	O(n^2)	O(2^n)	O(N!)
C++	3.02	1	1	1.11	1
Java	4.18	1.48	5.26	1	3.04
Julia	1	1.69	58.19	1.49	8.44
Python	37.05	64.91	17.21	9.93	4.01

Tablica 3.1 Prikaz učinkovitosti između jezika

Tablica 3.2 prikazuje trajanja programa u odnosu na druge programe za isti algoritam. Vrijednost 1 dodijeljena je najbržem programu te se u ovisnosti o njemu gledaju koliko su puta sporiji ostali programi. U većini slučajeva usporedbom tablica 3.1 i 3.2 može

se zaključiti kako jezici zadržavaju poredak te je najučinkovitiji jezik i najbrži. Ovaj zaključak je donesen već prije u radu, no ove tablice detaljnije prikazuju to zajedno sa anomalijama gdje se mijenja poredak.

	O(n)	O(N * log N)	O(n^2)	O(2^n)	O(N!)
C++	4.46	1	1	1.19	1
Java	3.38	1.31	4.76	1	3.27
Julia	1	1.22	50.95	2.34	8.54
Python	59.55	69.56	6.24	13.49	4.4

Tablica 3.2 Prikaz trajanja programa između jezika

Poglavlje 4

Zaključak

Po završetku mjerenja te analize samih mjerenja mogu se donijeti konkretni zaključci te odgovori na ranija pitanja. Za početak iz samih mjerenja može se zaključiti kako su jezici koji koriste kompajler umjesto interpretera brži i energetski učinkovitiji. To je bilo i za očekivati, a mjerenja su samo potvrdila tu tezu. Vremenski i energetski rezultati između sebe vuku još svakakve druge varijable te je teško povući poveznicu samo između njih [1]. Programeri sve više teže ka manjoj energetskoj potrošnji programa. Kako bi to postigli osim same optimizacije koda potrebno je poznavati točan rad svakog programskog jezika jer svaki jezik radi na specifični način. Ukoliko se koristi hibridno prevođenje korištenjem kompajlera i interpretera potrebno je voditi računa i o garbage collection-u, vanjskim knjižnicama te o složenosti algoritma. Primjeri za Juliju vrlo su konkretno prikazali da ukoliko se koriste pravilno kompajleri se mogu jako optimizirati.

- Je li brži kod i energetski učinkovitiji?

  Vrlo česta zabluda je da ukoliko se napravi brži kod da će on biti i energetski učinkovitiji. Mjerenja složenosti O(n) pokazuju kako kod vrlo velikih ulaza trajanje raste znatno više nego energetska potrošnja. Mjerenja složenosti O(n^2) pokazala su da Python koji je prema energiji rangiran kao drugi, u vremenskom promatranju

dolazi na treće mjesto. Oscilacije između energije i vremena normalna su stvar te ovise o puno faktora. Najveći dio mjeranja ipak pokazuje da porastom energije raste i vrijeme izvršavanja te obrnuto. Stoga bi se u svakodnevnom programiranju moglo voditi po tom načelu te dodatnim mjerenjima vidjeti je li došlo do odstupanja te istražiti zašto. Iako vrlo često energija i vrijeme prate jedan drugog općeniti zaključak je da brži kod nije uvijek energetski učinkovitiji. Točne vrijednosti i ovisnosti pokazuju tablice 3.1 i 3.2.

- Rangiranje jezika prema energiji i vremenu izvršavanja

  Prilikom rangiranja jezika mora se paziti na puno stvari koje se uzimaju u obzir. Ovdje će se jezici rangirati isključivo prema energetskom i vremenskom izvršavanju. Kada se gleda i ukupno korištena memorija poredak može biti malo drugačiji, no ako se gledaju samo energija i vrijeme uvijek je moguće rangirati jezike. Iz provedenih mjerenja C++ je najbrži i najučinkovitiji jezik te to prikazuje tablica 3.1. Može se vidjeti kako je C++ najučinkovitiji i najbrži u čak 3 algoritma dok su mu performanse za ostala dva vrlo konkuretne prvom mjestu. C++ zauzima sigurno prvo mjesto i zbog već objašnjenih razloga u radu je boljih performansi od ostalih. Java je druga i performanse Jave su vrlo konstatne pri velikim ulaznim podacima te nema velikih oscilacija. Za zadnja dva mjesta javlja se problem. Pošto su mjerenja napravljena na istim programima bez dodatne optimizacije ovisno o algoritamskom složenosti negdje bolje performanse ima Julia, a negdje Python. No gledajući arhitekturu te rezultate kada se poštivaju pravila Julia kompajlera, Julia zauzima treće mjesto iz razloga što su performanse u 3 algoritma vrlo konkurente C++ i Javi te su bolje od Python-a . Python je zadnji te je energetski najneučinkovitiji i najsporiji jezik te se to jasno može vidjeti iz tablice 3.2. Python je najsporiji u čak 4 algoritma te jako odstupa od ostalih jezika. Pri vrlo velikim ulaznim podacima brojke jako odskaču te izvođenje programa postaje vrlo naporno.

Prilikom odabira jezika za programiranje programer neće uvijek gledati samo energiju

i vrijeme izvršavanja. Vrlo često programeri neće raditi na ovako apstrakno velikim ulaznim podacima prikazanima u radu. Pri malim ulazima svi jezici su vrlo brzi i energetski efikasni. Optimizacija kompajlera i interpretera omogućuje takve rezultate. Ukoliko se jezici promatraju prema zajednicama, dostupnim knjižnicama, sintaksi , prenosivosti i ostalim metrikama poredak se znatno mijenja. C++ koji je po performansama najbolji vjerovatno će biti najmanje izabran od strane programera zbog svoje teške sintakse, manjka zajednice te zbog nezahvalnog broja vanjskih knjižnica. Zatim će programeri odabrat Juliju zbog mogućnosti paralelizma te jednostavne sintakse. Julia je relativno novi jezik te nema puno knjižnica i razvijenu zajednicu, no često se koristi zbog vrlo lagane sintakse i dobrih performansi. Zatim će programeri odabrati Javu koja je vrlo dobrih performansi te je u potpunosti objektno orijentirani jezik. Osim toga skoro svi uređaji koriste Java programe te je vrlo raširena zajednica te postoji puno vanjskih knjižnica i API-a. Python je trenutno jezik kojeg svi programeri koriste. Iako je po performansama daleko najgori vrlo široka zajednica s bezbroj mogućih vanjskih knjižnica te vrlo jednostavnom sintaksom omogućuje ljudima koji nikad nisu programirali da vrlo lagano pišu programe[5][18].

Svako ima nekakvu svoju preferencu jezika. Ovaj rad detaljno je objasnio načine izvršavanja te performanse u različitim slučajevima i potencijalno pomogao nekome u izboru programskog jezika s obzirom na probleme koje treba rješiti.

Bibliografija

[1] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, ...(2021). Ranking programming languages by Energy Efficiency. https://haslab.github.io/SAFER/scp21.pdf.

[2] University of Babylon (2020). Lecture about computer architecture. https://www.uobabylon.edu.iq/eprints/publication_4_4134_49.pdf.

[3] Dr. Raaid Alubady, (2021). Programming Languages: Classification, Execution Model, and Errors . https://www.uobabylon.edu.iq/eprints/publication_11_23917_6270.pdf.

[4] Isaac computer science, (Pristupljeno 15.03.2023.) , Internetski članak . Low level languages. https://isaaccomputerscience.org/concepts/sys_proglang_low_level?examBoard=all&stage=all&topic=programming_languages.

[5] Coursera, (Pristupljeno 15.03.2023.) , Internetski članak . 5 Types of programming languages. https://www.coursera.org/articles/types-programming-language.

[6] TechTarget Contributor,(Pristupljeno 25.03.2023.), Internetski članak . Scripting languages. https://www.techtarget.com/whatis/definition/scripting-language.

[7] javaTpoint,(Pristupljeno 25.03.2023.), Internetski članak . What is Procedural language?. https://www.javatpoint.com/what-is-procedural-language.

[8] Erin Doherty, Educative ,(Pristupljeno 25.03.2023.), Internetski članak . What is object-oriented programming? OOP explained in depth. https://www.educative.io/blog/object-oriented-programming.

[9] tutorialspoint, (Pristupljeno 02.04.2023.) , Internetski članak . Functional Programming - Introduction. https://www.tutorialspoint.com/functional_programming/functional_programming_introduction.htm.

[10] Khalil Stemmler, khalilstemmler, (Pristupljeno 02.04.2023.) , Internetski članak . 4 Principles of Object-oriented programming?. https://khalilstemmler.com/articles/object-oriented/programming/4-principles/.

[11] Margaret Rouse, tectopedia, (Pristupljeno 12.04.2023.) , Internetski članak . Medium-Level language. https://www.techopedia.com/definition/20789/medium-level-language-mll.

[12] University of Oxford, Department of Engineering science,(2022), The CPU, instruction Fetch & Execute . https://www.robots.ox.ac.uk/~dwm/Courses/2CO_2014/2CO-N2.pdf.

[13] James Peckol, Wiliam Stallings, (2021), Characteristics and Functions Addressing Modes. https://redirect.cs.umbc.edu/~tinoosh/cmpe311/notes/instruction%20types.pdf.

[14] IBM documentation, (2001-2010) ,What is interrupt processing. https://www.ibm.com/docs/en/zos-basic-skills?topic=system-what-is-interrupt-processing.

[15] Sebastion Schmielder, Programmathically, (Pristupljeno 20.04.2023.) , Internetski članak .How does a CPU Execute Instructions : Understanding Instruction Cycles. https://programmathically.com/how-does-a-cpu-execute-instructions-understanding-instruction-cycles/.

[16] David Evans, Computingbook, (2021.) , Interpreters. https://computingbook.org/Interpreters.pdf.

[17] Prof. Douglas Thain, University of Notre Dame, (2020.) , Introduction to Compilers and language design. https://www3.nd.edu/~dthain/compilerbook/compilerbook.pdf.

[18] Faisal Chughtai, Theory Notes, (Pristupljeno 20.04.2023.) , Internetski članak .Compiler and Interpreter. https://faisalchughtai.com/media/resources/CH%3A%202%20%7C%20Theory%20notes%20-%20Paper%201%20%7C%20IT%20(9626)%20%7C%20A%20Level/Compiler_and_interpreter.pdf.

[19] Davis Gries, Cornell Bowers Computer Science, (2018.) , How Java works. https://www.cs.cornell.edu/courses/JavaAndDS/files/howJavaWorks.pdf.

[20] Tarun Luthra, Scaler Topics, (Pristupljeno 05.05.2023.) , Internetski članak, How Java program works. https://www.scaler.com/topics/java/how-java-program-works/.

[21] Prateek Narang, Scaler Topics, (Pristupljeno 05.05.2023.) , Internetski članak, How is a C++ program compiled and executed. https://www.scaler.com/topics/how-to-compile-cpp/.

[22] Sabah Al-Fedaghi, Kuwait University, (2013.) , Conceptual Understanding of Computer Program Execution: Application to C++. https://www.researchgate.net/publication/236248406_Conceptual_Understanding_of_Computer_Program_Execution_Application_to_C.

[23] Dhruvil Karani, Towards data science, (Pristupljeno 05.05.2023.) , Internetski članak, How does Python work?. https://towardsdatascience.com/how-does-python-work-6f21fd197888.

[24] Victor Skvortsov,Ten thousand meters, (Pristupljeno 06.05.2023.) , Internetski članak, Python behind the scenes: how the CPython VM works. https://tenthousandmeters.com/blog/python-behind-the-scenes-1-how-the-cpython-vm-works/.

[25] Ambert Lency ,morioh, (Pristupljeno 06.05.2023.) , Internetski članak, PVM vs JVM - What the difference?. https://morioh.com/p/01d84b0bab10.

[26] David Harvey,Joris van Der Hoeven,(2020) , Integer multiplication in time O(n log n). https://hal.science/hal-02070778v2/file/nlogn.pdf.

[27] University of Pensilvania,(2021) , Julia tutorial . https://www.sas.upenn.edu/~jesusfv/Chapter_HPC_8_Julia.pdf.

[28] Tim Besard, Bjorn De Sutter, Ghent University , (2016) , High-Level GPU programming in Julia . https://www.researchgate.net/figure/Julia-compiler-components-and-interactions_fig1_301876510.

Bibliografija slika

[1] Slika 1.1 preuzeta je sa `https://www.learncomputerscienceonline.com/instruction-cycle/` Pristupljeno (15.03.2023.)

[2] Slika 1.2 preuzeta je sa `https://slideplayer.com/slide/5981610/` Pristupljeno(15.03.2023.)

[3] Slika 1.3 preuzeta je sa `https://www.javatpoint.com/compiler-vs-interpreter` Pristupljeno (20.04.2023.)

[4] Slika 2.1 preuzeta je sa `https://devtut.github.io/java/just-in-time-jit-compiler.html` Pristupljeno (05.05.2023)

[5] Slika 2.2 preuzeta je sa `https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp0_Introduction.html` Pristupljeno (05.05.2023.)

[6] Slika 2.3 preuzeta je sa `https://stackoverflow.com/questions/17130975/python-vs-cpython` Pristupljeno (06.05.2023.)

[7] Slika 2.4 preuzeta je sa `https://www.researchgate.net/figure/Julia-compiler-components-and-interactions_fig1_301876510` Pristupljeno (06.05.2023.)

Sažetak

Programiranje te razvoj aplikacija često sa sobom donosi pitanja kakve su performanse određenih aplikacija. Postoje mnoge metode usporedbe performansi , no dvije najpoznatije su mjerenje energije i vremena izvršavanja. U ovom radu istražena je podjela programskih jezika s obzirom na specifičnosti njihova izvršavanja po koracima. Tema rada je učinkovitost izvršavanja koda u programskim jezicima visoke razine. Osim toga detaljno je opisan način izvršavanja svakog programa u svih odabranim programskim jezicima. Odabran je podskup programskih jezika visoke razine koji se sastoji od nekih najpopularnijih jezika a to su C++, Java, Python i Julija. Na tom podskupu napravljena su mjerenja energetske i vremenske potrošnje na algoritmima različite složenosti i različitih veličina ulaznih podataka. Za kraj, napravljena je statistička analiza izmjerenih rezultata te se istražuju korelacije energije i vremena te se rangiraju jezici s obzirom na konkretne zaključke. U radu se kroz detaljne opise i izvedbe koraka izvođenja pokušava odgovoriti na neka vrlo česta pitanja. Među tim pitanjima je i pitanje ukoliko je brži program i energetski učinkovitiji te jesu li vrijeme i energija povezani pojmovi. Također, rad uvelike pomaže pri odabiru jezika pod konkretnim zahjevima koji se tiču performansi. Ovaj završni rad izrađen je u okviru ERASMUS+ projekta `Promoting Sustainability as a Fundamental Driver in Software development Training and Education` sa oznakom 2020-1-PT01-KA203-078646.

Ključne riječi **— Python, Java, Julia, C++, Učinkovitost, pyRAPL, Brzina izvršavanja, Performanse**

Abstract

Programming and application development often bring questions about the performance of specific applications. There are many methods for comparing performance, but the two most well-known are energy and execution time measurements. This paper explores the categorization of programming languages based on the specifics of their execution steps. The topic of the paper is the efficiency of code execution in high-level programming languages. Additionally, a detailed description of the execution process for each program in the selected programming languages is provided. A subset of high-level programming languages consisting of popular languages such as C++, Java, Python, and Julia was chosen. Measurements of energy and time consumption were conducted on algorithms of different complexities and various input data sizes within this subset. Finally, a statistical analysis of the measured results was performed, exploring the correlations between energy and time, and ranking the languages based on specific conclusions. The paper attempts to answer some common questions through detailed descriptions and step-by-step executions. Among these questions is the inquiry into whether a faster program is also more energy-efficient and whether time and energy are interconnected concepts. Furthermore, the paper greatly assists in language selection based on specific performance requirements. This final paper was developed within the framework of the ERASMUS+ project `Promoting Sustainability as a Fundamental Driver in Software development Training and Education` with the code 2020-1-PT01-KA203-078646.

Keywords **— Python, Java, Julia, C++, Efficiency, pyRAPL, Execution speed, Performance**