40 algoritama

koje bi svaki programer trebalo da zna

Imran Ahmad

Unapredite svoje veštine rešavanja problema učenjem različitih algoritama i njihovom primenom u Pythonu

kompjuter biblioteka

Packt›

40 ALGORITAMA

koje bi svaki programer trebalo da zna

Imran Ahmad

kompjuter biblioteka

Packt>

Izdavač:

kompjuter biblioteka

Obalskih radnika 4a, Beograd

Tel: 011/2520272

e-mail: kombib@gmail.com

internet: www.kombib.rs

Urednik: Mihailo J. Šolajić

Za izdavača, direktor:
Mihailo J. Šolajić

Autor: Imran Ahmad

Prevod: Slavica Prudkov

Lektura: Miloš Jevtović

Slog: Zvonko Aleksić

Znak Kompjuter biblioteke:
Miloš Milosavljević

Štampa: „Pekograf“, Zemun

Tiraž: 500

Godina izdanja: 2020.

Broj knjige: 530

Izdanje: Prvo

ISBN: 978-86-7310-553-6

40 Algorithms
Every Programmer Should Know

Antonio Melé

ISBN 978-1-78980-121-7

O AUTORU

Imran Ahmad je sertifikovani Google instruktor i više godina podučava za Google i Learning Tree. Teme koje Imran podučava uključuju Python, mašinsko učenje, algoritme, big data i duboko učenje. U svom doktoratu predložio je novi algoritam zasnovan na linearnom programiranju, pod nazivom ATSRA, koji može da se upotrebi za optimalnu dodelu resursa u cloud računarskom okruženju. Poslednje 4 godine, Imran radi na visokoprofilnom projektu mašinskog učenja u naprednoj laboratoriji analitike kanadske savezne države. Projekat je razvoj algoritama mašinskog učenja koji mogu da automatizuju proces imigracije. Imran trenutno radi na razvoju algoritama za optimalnu upotrebu GPU-a za obučavanje kompleksnih modela mašinskog učenja.

O RECENZENTIMA

Benjamin Baka je full-stack softverski programer i strastven je po pitanju vrhunskih tehnologija i elegantnih tehnika programiranja. Ima 10 godina iskustva u različitim tehnologijama, od C++, Java i Ruby-a do Pythona i Qt-a. Neki od projekata na kojima radi mogu se pronaći na njegovoj GitHub stranici. Trenutno radi na uzbudljivim tehnologijama za mPedigree.

PACKT TRAŽI AUTORE KAO ŠTO STE VI

Ako ste zainteresovani da postanete autor za Packt, posetite stranicu authors.packtpub.com i prijavite se još danas. Mi smo radili sa hiljadama programera i tehničkih profesionalaca, kao što ste vi, da bismo im pomogli da podele svoj uvid sa globalnom tehnološkom zajednicom. Možete da popunite uopštenu prijavu, prijavite se za specifičnu temu za koju tražimo autore ili pošaljete neke svoje ideje.

Uvod

Algoritmi su uvek igrali važnu ulogu u nauci i praksi računarstva. U ovoj knjizi ćemo se fokusirati na upotrebu algoritama za rešavanje problema iz stvarnog sveta. Da biste dobili maksimum od ovih algoritama, razumevanje njihove logike i metamatike je neophodno. Prvo ćemo predstaviti algoritme i otkriti različite tehnike projektovanja algoritama. Zatim ćete učiti o linearnom programiranju, rangiranju stranice i grafovima, a koristićete i algoritme mašinskog učenja i razumećete matematiku i logiku u njima. Ova knjiga sadrži studije slučajeva, kao što je predviđanje vremenske prognoze, grupisanje tvitova i mehanizmi za preporučivanje filmova, koje će vam pokazati kako možete da primenite optimalno ove algoritme. Kada završite čitanje ove knjige postaćete vešti u upotrebi algoritama za rešavanje stvarnih računskih problema.

ZA KOGA JE OVA KNJIGA

Ova knjiga je za ozbiljne programere! Bez obzira da li ste iskusan programer, koji želi bolje da razume matematiku iza algoritama, ili imate ograničeno znanje na polju programiranja, ili nauke o podacima, a želite da naučite više o načinu na koji možete da iskorisite ove testirane algoritme, da biste poboljšali način na koji dizajnirate i pišete kod, ova knjiga će vam biti korisna. Iskustvo u Python programiranju je obavezno, a poznavanje nauke o podacima je korisno, ali nije neophodno.

ŠTA OBUHVATA OVA KNJIGA

Poglavlje 1, *Pregled algoritama* – u ovom poglavlju rezimiraćemo osnove algoritama. Poglavlje započinjemo odeljkom o osnovnim konceptima koji su potrebni da biste razumeli kako funkcionišu različiti algoritmi. Rezimiraćemo kako su ljudi počeli da koriste algoritme da bi matematički formulisali određene klase i

probleme. Takođe ćemo pomenuti ograničenja različitih algoritama. U sledećem odeljku ćemo objasniti različite načine specifikovanja logike algoritma. Pošto je u ovoj knjizi upotrebljen Python za pisanje algoritama, objašnjeno je i kako da podesite okruženje da biste pokrenuli primere. Zatim su opisani različitih načini na koje možete da kvantifikujete i uporedite performansu algoritma sa drugim algoritmima. Na kraju ovog poglavlja govorićemo o različitim načinima na koje određena implementacija algoritma može da bude potvrđena.

Poglavlje 2, *Strukture podataka upotrebljene u algoritmima* – u ovom poglavlju ćemo se fokusirati na potrebu algoritama za strukturama podataka u memoriji, koje mogu da skladište privremene podatke. Algoritmi mogu da budu veoma intenzivni u pogledu podataka ili izračunavanja, ili oba. Ali za sve različite tipove algoritama, biranje odgovarajuće strukture podataka je važno za njihovu optimalnu implementaciju. Mnogi algoritmi imaju rekurzivnu i iterativnu logiku i zahtevaju specijalizovane strukture podataka, koje su u osnovi iterativne po prirodi. Pošto u ovoj knjizi koristimo Python, u ovom poglavlju ćemo se fokusirati na Python strukture podataka, koje mogu da se upotrebe za implementiranje algoritama koji su opisani u ovoj knjizi.

Poglavlje 3, *Algoritmi sortiranja i pretraživanja* – u ovom poglavlju predstavićemo osnovne algoritme koji se koriste za sortiranje i pretraživanje. Ovi algoritmi mogu, kasnije, da postanu osnova za mnogo složenije algoritme. Poglavlje ćemo započeti predstavljanjem različitih tipova algoritama za sortiranje. Takođe ćemo uporediti peformanse različitih pristupa. Zatim ćemo predstaviti različite algoritme za pretragu. Uporedićemo ih i kvantifikovanti njihovu performansu i kompleksnost. Na kraju ovog poglavlja predstavićemo primenu ovih algoritama.

Poglavlje 4, *Dizajniranje algoritama* – u ovom poglavlju ćemo predstaviti osnovne koncepte dizajna različitih algoritama. Takođe ćemo opisati različite tipove algoritama i govoriti o njihovim vrlinama i slabostima. Razumevanje ovih koncepata je važno kada je reč o dizajniranju optimalnih kompleksnih algoritama. Poglavlje ćemo započeti opisom različitih tipova dizajna algoritma. Zatim ćemo predstaviti rešenje za poznati problem trgovačkog putnika. Zatim ćemo govoriti o linearnom programiranju i njegovim ograničenjima. Na kraju ćemo predstaviti praktični primer koji pokazuje kako linearno programiranje može da se upotrebi za planiranje kapaciteta.

Poglavlje 5, *Grafovski algoritmi* – u ovom poglavlju ćemo se fokurisati na algoritme za probleme grafa koji su uobičajeni u računarskoj nauci. Postoje mnogi problemi izračunavanja, koji mogu najbolje da se predstave u terminima grafova. U ovom poglavlju ćemo predstaviti metode za predstavljanje grafa i za pretraživanje grafa. Pretraživanje grafa podrazumeva sistematsko praćenje ivica grafa, da biste odredili vrh grafa. Algoritmi za pretraživanje grafa mogu da otkriju mnogo o strukturi grafa. Mnogi algoritmi započinju pretraživanjem ulaznog grafa za dobijanje informacija o njegovoj strukturi. Nekoliko drugih grafovskih algoritama razrađuju osnovno pretraživanje grafa. Tehnike za pretraživanje grafa nalaze

se u srcu polja grafovskih algoritama. U prvom odeljku ćemo opisati dve najčešće računske reprezentacije grafova: lista susedstva i matrice povezanosti. Zatim ćemo predstaviti jednostavan algoritam pretraživanja grafa, pod nazivom breadth-first search i prikazaćemo kako da kreirate breadth-first stablo. U sledećim odeljcima predstavljena je depth-first pretraga i obezbeđeni su neki standardni rezultati o redosledu u kojem depth-first pretraga doseže vrh grafa.

Poglavlje 6, *Algoritmi nenadgledanog mašinskog učenja* – u ovom poglavlju predstavićemo algoritme nenadgledanog mašinskog učenja. Ovi algoritmi su klasifikovani kao nenadgledani jer model ili algoritam pokušava da nauči svojstvene strukture, obrasce i odnose iz datih podataka, bez ikakvog nadgledanja. Prvo su opisani metodi klasterovanja. To su metodi mašinskog učenja koji pokušavaju da pronađu obrasce sličnosti i odnosa među uzorcima podataka u skupu podataka, a zatim klasteruju ove uzorke u različite grupe, na primer, tako da svaka grupa ili klaster uzoraka podataka ima neke sličnosti, na osnovu svojstvenih karakteristika ili atributa. U sledećim odeljcima opisani su algoritmi redukcije dimenzionalnosti, koji se koriste kada postoji više atributa. Zatim su predstavljeni neki algoritmi koji se bave detekcijom anomalija. Na kraju ovog poglavlja predstavićemo mining pravila asocijacije, koji je metod data mininga koji se koristi za ispitivanje i analizu velikih transakcionih skupova podataka, za identifikovanje obrazaca i pravila. Ovi obrasci predstavljaju interesantne odnose i asocijacije između različitih stavki u transakcijama.

Poglavlje 7, *Algoritmi tradicionalnog nadgledanog učenja* – u ovom poglavlju opisaćemo algoritme tradicionalnog nadgledanog mašinskog učenja u odnosu na skup problema mašinskog učenja, u kojem postoji označeni skup podataka sa ulaznim atributima i odgovarajuće izlazne oznake ili klase. Ovi ulazi i odgovarajući izlazi se, zatim, koriste za obučavanje generalizovanog sistema, koji može da se upotrebi za predviđanje rezultata za prethodno neviđene tačke podataka. Prvo ćemo predstaviti koncept klasifikacije u kontekstu mašinskog učenja. Zatim ćemo predstaviti najjednostavnije algoritme mašinskog učenja, linearnu regresiju. Nakon toga ćemo govoriti o jednom od najvažnijih algoritama, stablu odlučivanja. Takođe ćemo govoriti o ograničenjima i moći algoritama stabla odlučivanja i opisaćemo dva najvažnija algoritma, SVM and XGBoost.

Poglavlje 8, *Algoritmi neuronske mreže* – prvo ćemo predstaviti glavne koncepte i komponente tipične neuronske mreže, koja postaje najvažniji tip tehnike mašinskog učenja. Zatim ćemo predstaviti različite tipove neuronskih mreža i takođe ćemo objasniti različite vrste aktivacionih funkcija koje se koriste za realizaciju ovih neuronskih mreža. Zatim je detaljno opisan backpropagation algoritam. Ovo je najčešće upotrebljavan algoritam za konvergenciju problema neuronske mreže. Zatim je objašnjena tehnika transfer učenja, koja može da se upotrebi za znatno pojednostavljivanje i delimičnu automatizaciju obučavanja modela. Na kraju je, kao primer stvarnog sveta, predstavljeno kako da upotrebite duboko učenje za detektovanje objekata u multimedijalnim podacima.

Poglavlje 9, *Algoritmi za obradu prirodnog jezika* – u ovom poglavlju ćemo predstaviti algoritme za obradu **prirodnog jezika (NLP)**. U ovom poglavlju obuhvaćena je teorija i praksa na progresivan način. Prvo ćemo predstaviti osnove, zatim osnovnu matematiku. Zatim ćemo govoriti o najčešće upotrebljavanim neuronskim mrežama za dizajniranje i implementiranje nekoliko važnih slučajeva upotrebe tekstualnih podataka. Ograničenja NLP-a su takođe opisana. Na kraju, predstavljena je studija slučaja u kojem je model obučen za detektovanje autora rada na osnovu stila pisanja.

Poglavlje 10, *Mehanizmi za preporučivanje* – u ovom poglavlju ćemo se fokusirati na mehanizme za preporučivanje, koji su način za modelovanje informacija dostupnih u odnosu na preference korisnika, a zatim upotreba ovih informacija, za obezbeđivanje informisanih preporuka na osnovu tih informacija. Osnova mehanizma za preporučivanje je uvek snimljena interakcija između korisnika i proizvoda. Ovo poglavlje ćemo započeti predstavljanjem osnovne ideje iza mehanizma za preporučivanje. Zatim ćemo govoriti o različitim tipovima mehanizma za preporučivanje. Na kraju ovog poglavlja opisaćemo kako se mehanizmi za preporučivanje koriste za predlaganje stavki i proizvoda različitim korisnicima.

Poglavlje 11, *Algoritmi podataka* – u ovom poglavlju ćemo se fokusirati na probleme vezane za algoritme usmerene ka podacima. Poglavlje ćemo započeti kratkim pregledom problema vezanih za podatke. Zatim ćemo predstaviti kriterijum za klasifikaciju podataka. Zatim ćemo opisati kako da primenite algoritme, da biste olakšali primenu podataka, a zatim ćemo predstaviti temu kriptografije. Na kraju ćemo predstaviti praktičan primer izdvajanja obrazaca iz Twitter podataka.

Poglavlje 12, *Kriptografija* – u ovom poglavlju ćemo predstaviti algoritme vezane za kriptografiju. Poglavlje ćemo započeti predstavljanjem pozadine. Zatim ćemo govoriti o algoritmima simetrične enkripcije. Objasnićemo MD5 i SHA algoritme heširanja i ograničenja i slabosti povezane sa implementiranjem simetričnih algoritama. Zatim ćemo govoriti o algoritmima asimentrične enkripcije i kako se oni koriste za kreiranje digitalnih sertifikata. Na kraju ćemo predstaviti praktičan primer koji rezimira sve ove tehnike.

Poglavlje 13, *Algoritmi velikih razmera* – u ovom poglavlju ćemo objasniti kako algoritmi velikih razmera obrađuju podatke koji ne mogu da se uklope u memoriju jednog čvora i uključivanje obrade koja zahteva više CPU-a. Poglavlje ćemo započeti opisom tipova algoritama koji su najprikladniji za paralelno pokretanje. Zatim ćemo govoriti o problemima vezanim za paralelizaciju algoritama. Takođe ćemo predstaviti CUDA arhitekturu i opisati kako jedan GPU ili niz GPU-ova može da se upotrebi za ubrzavanje algoritama i koje se promene moraju izvršiti u algoritmu da bismo efikasno iskoristili moć GPU-a. Na kraju ovog poglavlja ćemo govoriti o klaster računarstvu i opisaćemo kako Apache Spark kreira resilient distributed dataset-ove (RDD) za kreiranje ekstremno brzin paralelnih implementacija standardnih algoritama.

Poglavlje 14, *Praktična razmatranja* – ovo poglavlje ćemo započeti važnom temom objašnjenja, koja postaje sve važnija sada kada je objašnjena logika iza automatizovanog donošenja odluka. Zatim ćemo predstaviti etiku upotrebe algoritma i mogućnosti kreiranja biasa kada ih implementiramo. Zatim ćemo detaljno opisati tehnike za obradu NP problema. Na kraju ćemo opisati načine implementiranja algoritama i izazove iz stvarnog sveta povezane sa njima.

DA BISTE DOBILI MAKSIMUM IZ OVE KNJIGE

BROJ POGLAVLJA	POTREBAN SOFTVER (SA VERZIJOM)	BESPLATAN/ VLASNIČKI	SPECIFIKACIJE HARDVERA	POTREBAN OS
1-14	Python verzija 3.7.2 ili novija	Besplatan	Min 4GB RAM-a, 8GB+ preporučeno.	Windows/Linux/Mac

Preuzimanje fajlova primera koda

Možete da preuzmete fajlove sa primerima koda za ovu knjigu na adresi: https://bit.ly/3faHj7x

Kada su fajlovi preuzeti, raspakujte ili ekstrahujte direktorijum koristeći najnoviju verziju:

- WinRAR / 7-Zip za Windows
- Zipeg / iZip / UnRarX za Mac
- 7-Zip / PeaZip za Linux

Preuzimanje kolornih slika

Takođe smo obezbedili PDF fajl koji sadrži kolorne slike snimaka ekrana/ dijagrama koji su upotrebljeni u knjizi. Možete da preuzmete ovaj fajl sa adrese:
https://bit.ly/2X7cxGk

Upotrebljene konvencije

U ovoj knjizi upotrebljen je veliki broj konvencija za tekst.

`CodeInText`: ukazuje na reči koda u tekstu, u nazivima tabele baze podataka, u nazivima direktorijuma, u nazivima fajlova, u ekstenzijama fajlova, putanjama, lažnim URL-ovima, korisničkim unosima i Twiter postovima. Primer je sledeći: „Pogledajmo kako da dodamo novi element u stek, upotrebom `push` metoda, ili uklonimo element iz steka, upotrebom metoda `pop`".

Blok koda je postavljen na sledeći način:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Kada želimo da privučemo vašu pažnju na određeni deo bloka koda, relevantne linije ili stavke će biti ispisane zadebljanim slovima:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Svi unosi ili ispisi komandne linije napisani su na sledeći način:

```
pip install a_package
```

Zadebljana slova: Novi termini i važne reči koje vidite na ekranu su napisane podebljanim slovima. Na primer, u menijima ili okvirima za dijalog prikazaće se u tekstu na taj način. Na primer: „Jedan način da smanjite kompleksnost algoritma je da kompromitujete njegovu tačnost, kreirajući tip algoritma pod nazivom **približni algoritam**."

Upozorenja ili važne napomene se prikazuju na ovaj način.

Saveti i trikovi se prikazuju ovako.

Postanite član Kompjuter biblioteke

Kupovinom jedne naše knjige stekli ste pravo da postanete član Kompjuter biblioteke. Kao član možete da kupujete knjige u pretplati sa 40% popusta i učestvujete u akcijama kada ostvarujete popuste na sva naša izdanja. Potrebno je samo da se prijavite preko formulara na našem sajtu.
Link za prijavu: http://bit.ly/2TxekSa

Skenirajte QR kod
registrujte knjigu
i osvojite nagradu

Deo I

Osnove i osnovni algoritmi

U ovom odeljku ćemo predstaviti osnovne aspekte algoritama. Istražićemo šta je algoritam i kako da ga dizajniramo, a takođe ćemo učiti o strukturama podataka koje se koriste u algoritmima. U ovom odeljku ćemo, takođe, objasniti algoritme za sortiranje i pretraživanje, zajedno sa algoritmima za rešavanje grafičkih problema. Poglavlja obuhvaćena u ovo delu su:

- **Poglavlje 1**, Pregled algoritama
- **Poglavlje 2**, Strukture podataka upotrebljene u algoritmima
- **Poglavlje 3**, Algoritmi za sortiranje i pretraživanje
- **Poglavlje 4**, Dizajniranje algoritama
- **Poglavlje 5**, Grafovski algoritmi

1

Pregled algoritama

U ovoj knjizi su obuhvaćene informacije za razumevanje, klasifikovanje, selekciju i implementiranje važnih algoritama. Osim što ćemo objasniti njihovu logiku, takođe ćemo govoriti o strukturama podataka, razvojnim okruženjima i proizvodnim okruženjima koja su pogodna za različite klase algoritama. Fokusiraćemo se na moderne algoritme mašinskog učenja, koji postaju sve važniji. Zajedno sa logikom, predstavljeni su i praktični primeri upotrebe algoritama za rešavanje stvarnih svakodnevnih problema.

U ovom poglavlju ćemo obezbediti uvid u osnove algoritama. Započećemo sa odeljkom o osnovnim konceptima koji su potrebni da biste razumeli kako funkcionišu različiti algoritmi. U ovom odeljku ćemo rezimirati kako su ljudi počeli da koriste algoritme za matematičko formulisanje određene klase problema. Takođe ćemo govoriti o ograničenjima različitih algoritama. U sledećem odeljku ćemo objasniti različite načine za specifikovanje logike algoritma. Pošto u ovoj knjizi koristimo Python za pisanje algoritama, objašnjeno je kako da podesite okruženje za pokretanje primera. Zatim ćemo opisati različite načine kvantifikovanja performansi algoritma u poređenju sa drugim algoritmima. Na kraju ovog poglavlja ćemo govoriti o različitim načinima na koje možemo da izvršimo validaciju određene implementacije algoritma.

Da rezimiramo, u ovom poglavlju obuhvaćene su sledeće glavne teme:

- Šta je algoritam?
- Specifikovanje logike algoritma
- Predstavljanje Python paketa
- Tehnike dizajniranja algoritma

- Analiza performansi
- Validacija algoritma

ŠTA JE ALGORITAM?

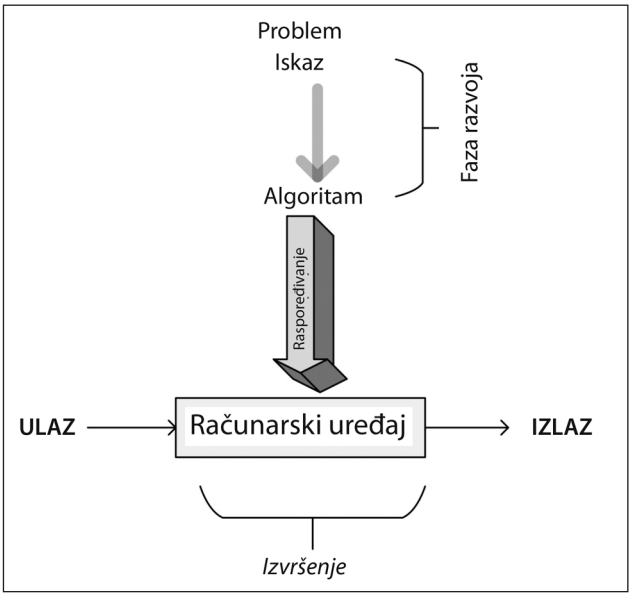
Najjednostavnije rečeno, algoritam je skup pravila za izvršavanje nekih izračunavanja za rešavanje problema. Algoritam je dizajniran da daje rezultate za bilo koji validan unos, u skladu sa precizno definisanim instrukcijama. Ako potražite reč algoritam u rečniku engleskog jezika (kao što je American Heritage), koncept je definisan na sledeći način:

> *"Algoritam je konačan skup nedvosmislenih instrukcija koje, s obzirom na neki skup početnih uslova, mogu da budu izvršene u unapred određenoj sekvenci za postizanje određenog cilja i koje imaju prepoznatljiv skup krajnjih uslova."*

Dizajniranje algoritma je, u stvari, kreiranje matematičkog recepta na najefikasniji način, koji može efikasno da se upotrebi za rešavanje problema iz stvarnog sveta. Ovaj recept može da se upotrebi kao osnova za razvoj višestruko upotrebljivog i generičkog matematičkog rešenja, koje se može primeniti na širi skup sličnih problema.

Faze algoritma

Različite faze razvoja, raspoređivanja i konačne upotrebe algoritma prikazane su u sledećem dijagramu:



Kao što možemo da vidimo, proces započinje razumevanjem zahteva od iskaza problema, koji detaljno opisuje šta treba da bude izvršeno. Kada je problem jasno iskazan, vodi nas u fazu razvoja.

Faza razvoja se sastoji iz dve faze:

- **Faza dizajna**: U fazi dizajna, predviđeni su i dokumentovani detalji o arhitekturi, logici i implementaciji algoritma. Dok dizajniramo algoritam, trebalo bi da imamo na umu tačnost i performanse. Međutim, pretraživanje rešenja za dati problem će, u mnogim slučajevima, dovesti do toga da pronađemo više od jednog alternativnog algoritma. Faza dizajna algoritma je iterativni proces koji uključuje upoređivanje različitih kandidata za algoritme. Neki algoritmi mogu da obezbeđuju jednostavna i brza rešenja, ali mogu kompromitovati tačnost. Drugi algoritmi mogu da budu veoma tačni ali, zbog složenosti, njihovo pokretanje može značajno potrajati. Neki od ovih složenih algoritama mogu da budu efikasniji od drugih. Pre nego što donesete odluku, treba pažljivo da razmotrite inherentne kompromise kandidata za algoritme. Posebno za složene probleme, dizajniranje efikasnog algoritma je veoma važno. Pravilno dizajniran algoritam će rezultirati efikasnim rešenjem, koje će biti u mogućnosti da obezbedi, istovremeno, zadovoljavajuću performansu i razumnu tačnost.
- **Faza kodiranja**: U fazi kodiranja, dizajnirani algoritam je konvertovan u računarski program. Važno je da aktuelni program implementira svu logiku i arhitekturu predloženu u fazi dizajna.

Faze dizajniranja i kodiranja algoritma su, po prirodi, iterativne. Kreiranje dizajna koji ispunjava i funkcionalne i nefunkcionalne zahteve može zahtevati mnogo vremena i truda. Funkcionalni zahtevi su oni zahtevi koji diktiraju koji je odgovarajući izlaz za dati skup ulaznih podataka. Nefunkcionalni zahtevi algoritma su, uglavnom, vezani za performansu za datu veličinu podataka. Validacija i analiza performanse algoritma opisani su kasnije u ovom poglavlju. Validacija algoritma podrazumeva potvrđivanje da algoritam ispunjava svoje funkcionalne zahteve. Analiza performanse algoritma podrazumeva potvrđivanje da algoritam ispunjava svoj glavni nefunkcionalni zahtev: performansu.

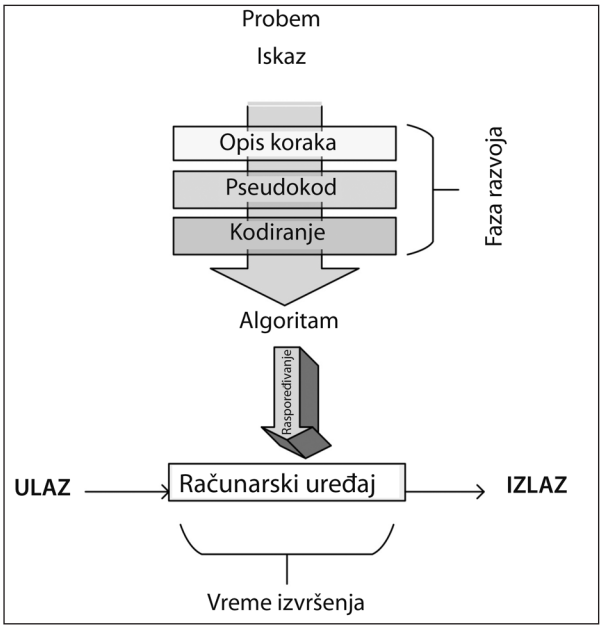
Kada je dizajniran i implementiran u programskom jeziku po vašem izboru, kod algoritma je spreman da bude raspoređen. Raspoređivanje algoritma uključuje projektovanje aktuelnog proizvodnog okruženja u kojem će se kod pokretati. Proizvodno okruženje trebalo bi da bude projektovano u skladu sa podacima i potrebama obrade algoritma. Na primer, za algoritme koji se mogu paralelizovati, biće potreban klaster sa odgovarajućim brojem računarskih čvorova za efikasno izvršenje algoritma. Za algoritme koji intenzivno koriste podatke bi trebalo da bude projektovan pipeline, za dolazne podatke, i strategija za keširanje i skladištenje podataka. Dizajniranje proizvodnog okruženja je detaljno opisano u Poglavlju 13, *Algoritmi velikih razmera*, i u Poglavlju 14, *Praktična razmatranja*. Kada je proizvodno okruženje projektovano i implementirano algoritam je raspoređen i koristi ulazne podatke, obrađuje ih i generiše izlaz prema zahtevima.

SPECIFIKOVANJE LOGIKE ALGORITMA

Kada dizajniramo algoritam, važno je da pronađemo različite načine za specifikovanje detalja. Potrebna je mogućnost pronalaženja njegove logike i arhitekture. Generalno, kao kada gradimo kuću, važno je da specifikujemo strukturu algoritma pre nego što ga implementiramo. Za složenije distribuirane algoritme, planiranje načina na koji će njihova logika biti distribuirana u klasteru u vreme izvršenja je važno za iterativno efikasan proces dizajna. Pomoću pseudokoda i planova izvršenja, oba ova zahteva su ispunjena i opisana su u sledećem odeljku.

Razumevanje pseudokoda

Najjednostavniji način da specifikujemo logiku za algoritam je da napišemo opis višeg nivoa za algoritam na polu-strukturirani način, što se naziva **pseudokod**. Pre nego što napišemo logiku u pseudokodu, korisno je da prvo opišemo glavni tok, pisanjem glavnih koraka na engleskom jeziku. Zatim, ovaj opis na engleskom jeziku je konvertovan u pseudokod, koji je strukturiran način pisanja ovog opisa na engleskom jeziku, koji najbliže predstavlja logiku i tok za algoritam. Dobro napisan pseudokod bi trebalo razumno da opiše korake algoritma visokog nivoa, čak i ako detaljni kod nije relevantan za glavni tok i strukturu algoritma. Na sledećoj slici prikazan je tok koraka:



Imajte na umu da kada je pseudokod napisan (kao što ćemo videti u sledećem odeljku), spremni smo da kodiramo algoritam, upotrebom programskog jezika po našem izboru.

Praktičan primer za pseudokod

U sledećem primeru prikazan je pseudokod algoritma za dodelu resursa pod nazivom **SRPMP**. U klaster računarstvu, postoje mnoge situacije u kojima postoje paralelni zadaci koji treba da se pokrenu na skupu dostupnih resursa, koji se kolektivno nazivaju **skladište resursa**. Ovaj algoritam dodeljuje zadatke za resurse i kreira skup mapiranja, pod nazivom Ω. Imajte na umu da predstavljeni pseudokod preuzima logiku i tok algoritma, koji su dalje opisani u sledećem odeljku:

```
1: BEGIN Mapping_Phase
2: Ω = { }
3: k = 1
4: FOREACH Ti∈T
5:       ωi = RA(Δk, Ti)
6:       add {ωi, Ti} to Ω
7:       state_changeTi [STATE 0: Idle/Unmapped] → [STATE 1: Idle/Mapped]
8:       k=k+1
9:       IF (k>q)
10:          k=1
11:      ENDIF
12: END FOREACH
13: END Mapping_Phase
```

Sada ćemo analizirati ovaj algoritam, liniju po liniju:

1. Započinjemo mapiranje izvršavanjem algoritma. Skup mapiranja Ω je prazan.
2. Prva particija je selektovana kao skladište resursa za T_1 zadatak (vidite liniju 3 u prethodnom kodu). **Television Rating Point (TRPS)** iterativno poziva **Rheumatoid Arthritis (RA)** algoritam za svaki T_i zadatak sa jednom od particija selektovanom kao skladište resursa.
3. RA algoritam vraća skup resursa izabran za T_i zadatak, predstavljen sa ωi (vidite liniju 5 u prethodnom kodu).
4. T_i i ωi su dodati u skup mapiranja (vidite liniju 6 u prethodnom kodu).
5. Stanje zadatka T_i je promenjeno sa `STATE 0:Idle/Mapping` na `STATE 1:Idle/Mapped` (vidite liniju 7 u prethodnom kodu).
6. Imajte na umu da za prvu iteraciju, `k=1` i selektovana je prva particija. Za svaku sledeću iteraciju, vrednost k se povećava do `k>q`.
7. Ako k postane veći od q, ponovo je resetovan na 1 (vidite linije 9 i 10 u prethodnom kodu).
8. Ovaj proces se ponavlja dok se mapiranje između svih zadataka i skupa resursa koji će upotrebiti ne odredi i sačuva u skupu mapiranja pod nazivom Ω.
9. Izvršenje se dešava kada je svaki od zadataka mapiran u skup resursa u fazi mapiranja.

Upotreba odlomaka koda

Sa popularnošću jednostavnog ali moćnog jezika za kodiranje, kao što je Python, alternativni pristup postaje popularan, a to je predstavljanje logike algoritma direktno u programskom jeziku, u nekako pojednostavljenoj verziji. Kao i pseudokod, ovaj selektovani kod preuzima važnu logiku i strukturu predloženog algoritma, izbegavajući detaljni kod. Ovaj selektovani kod se ponekad zove **odlomak** koda (eng. *snippet)*. U ovoj knjizi, odlomci koda su upotrebljeni umesto pseudokoda kada god je to moguće, jer oni štede jedan dodatni korak. Na primer, pogledajmo primer odlomka koda koji govori o Python funkciji koja se može upotrebiti za zamenu dve promenljive:

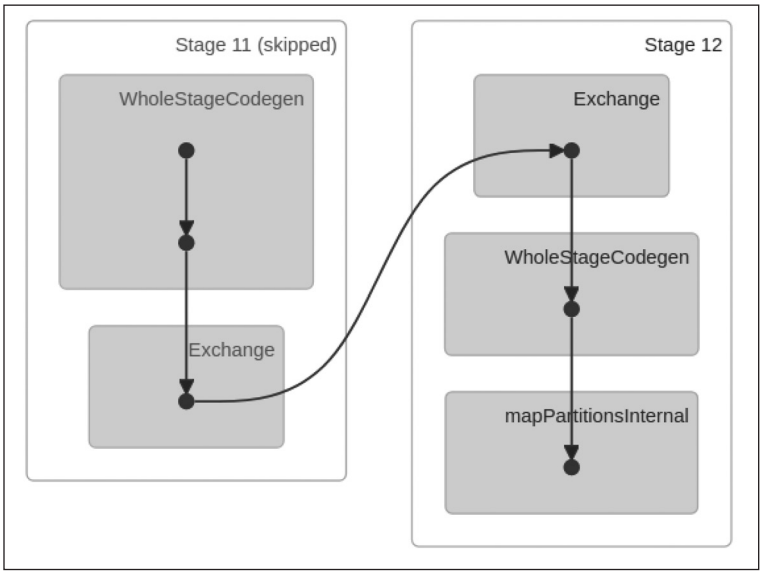
```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Imajte na umu da odlomci koda ne mogu uvek da zamene pseudokod. U pseudokodu, ponekad, izdvajamo mnogo linija koda kao jednu liniju pseudokoda, izražavajući logiku algoritma bez ometanja od nepotrebnih detalja kodiranja.

Kreiranje plana izvršenja

Pseudokod i odlomci koda nisu uvek dovoljni za specifikovanje logike koja se odnosi na složenije distribuirane algoritme. Na primer, distribuirani algoritmi, obično, treba da budu podeljeni u različite faze kodiranja, u vreme izvršenja, koje imaju redosled prioriteta. Odgovarajuća strategija za deljenje većeg problema na optimalan broj **faza** (eng. *stage*) sa odgovarajućim ograničenjima prioriteta je važna za efikasno izvršenje algoritma.

Treba da pronađemo način da predstavimo ovu strategiju, kao i da u potpunosti predstavimo logiku i strukturu algoritma. Plan izvršenja je jedan od načina prikaza detalja kako će algoritam biti podeljen u grane zadataka. Zadatak mogu da budu maperi ili reduktori, koji mogu da budu grupisani zajedno u blokove pod nazivom faze. Sledećim dijagramom prikazan je plan izvršenja koji generiše Apache Spark izvršenje pre izvršavanja algoritma. Detaljno su prikazani zadaci izvršenja, na koje će biti podeljen posao kreiran za izvršenje algoritma:



Vidite da prethodni dijagram ima pet zadataka koji su podeljeni u dve različite faze: **Stage 11** i **Stage 12**.

PREDSTAVLJANJE PYTHON PAKETA

Kada su dizajnirani, algoritmi treba da budu implementirani u programski jezik prema dizajnu. Za ovu knjigu ja sam izabrao programski jezik Python. Izabrao sam ga zato što je Python fleksibilan programski jezik otvorenog koda. Python je, takođe, jezik po izboru sve važnijih cloud računarskih infrastruktura, kao što su **Amazon Web Services (AWS)**, **Microsoft Azure i Google Cloud Platform (GCP)**.

Zvanična Python početna stranica je dostupna na https://www.python.org/ adresi, na kojoj se, takođe, nalaze i instrukcije za instalaciju i koristan vodič za početnike.

Ako niste ranije koristili Python, dobra ideja je da potražite vodiče za početnike, da biste ga sami naučili. Osnovno razumevanje Pythona će vam pomoći da bolje razumete koncepte predstavljene u ovoj knjizi.

Za ovu knjigu očekujem da koristite najnoviju verziju Python 3. U vreme pisanja ove knjige, najnovija verzija je 3.7.3 i tu verziju ćemo koristiti za pokretanje primera u ovoj knjizi.

Python paketi

Python je jezik opšte namene. Dizajniran je tako da dolazi sa minimalnom funkcionalnošću. Na osnovu slučaja upotrebe za koji nameravate da koristite Python, potrebno je da instalirate dodatne pakete pomoću pip instalacionog programa. Sledeća `pip` komanda može da se upotrebi za instaliranje dodatnih paketa:

```
pip install a_package
```

Paketi koji su već instalirani trebalo bi periodično da budu ažurirani, da biste dobili najnovije funkcionalnosti. To ćete uraditi upotrebom `upgrade` indikatora:

```
pip install a_package --upgrade
```

Još jedna Python distribucija za naučno računarstvo je Anaconda, koju možete da preuzmete sa http://continuum.io/downloads adrese.

Osim upotrebe `pip` komande za instaliranje novih paketa, za Anaconda distribuciju takođe, imamo i opciju za upotrebu sledeće komande za instaliranje novih paketa:

```
conda install a_package
```

Da biste ažurirali postojeće pakete, Anaconda distribucija obezbeđuje opciju za upotrebu sledeće komande:

```
conda update a_package
```

Postoji mnogo vrsta Python paketa koji su dostupni. Neki od važnih paketa koji su relevantni za algoritme opisani su u sledećem odeljku.

SciPy ekosistem

Scientific Python (SciPy) – izgovara se *sigh pie* – je grupa Python paketa kreiranih za naučnu zajednicu. Sadrži mnoge funkcije, uključujući i širok raspon generatora nasumičnih brojeva, rutine linearne algebre i optimizatore. SciPy je obiman paket i, vremenom, ljudi su razvili mnoge ekstenzije, da bi prilagodili i proširili paket u skladu sa svojim potrebama.

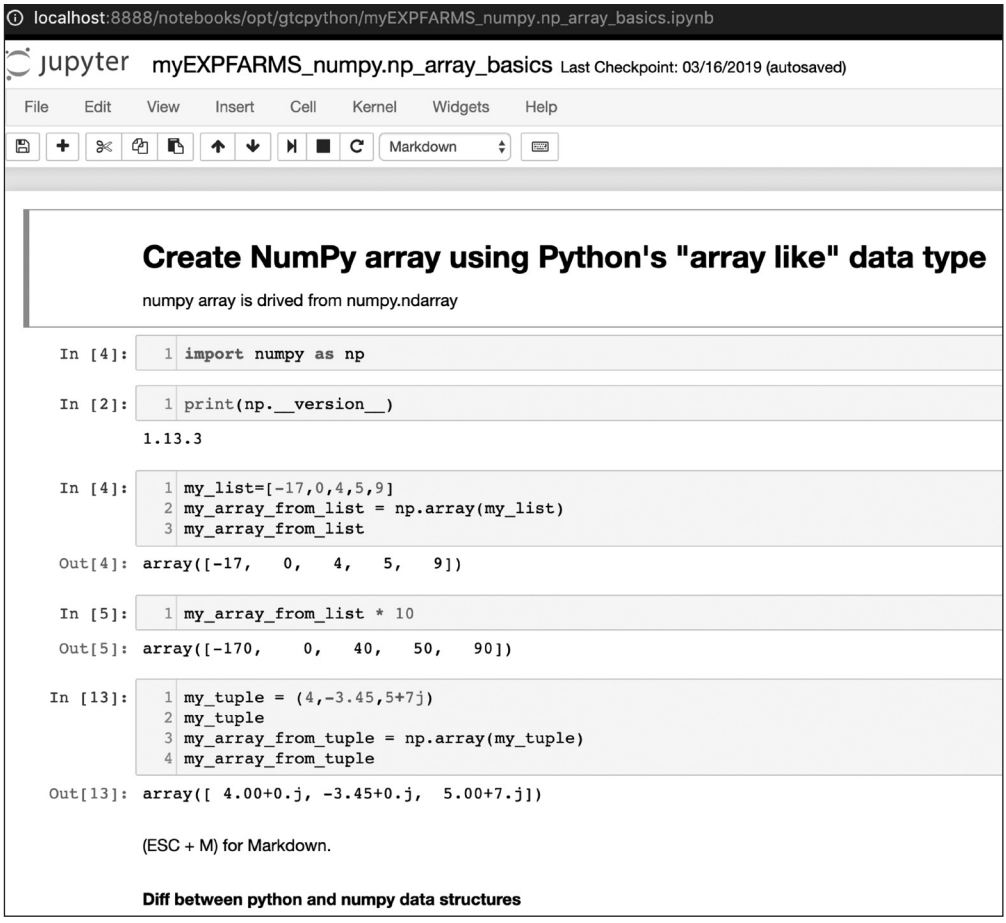
Slede glavni paketi koji su deo ovog ekosistema:

- **NumPy**: Za algoritme je veoma važna mogućnost kreiranja višedimenzionalnih struktura podataka, kao što su nizovi i matrice. NumPy obezbeđuje skup nizova i tipova podataka matrice koji su važni za statistiku i analizu podataka. Detalje o NumPy paketu možete pronaći na adresi http://www.numpy.org/.
- **scikit-learn**: Ova ekstenzija mašinskog učenja je jedna od najpopularnijih ekstenzija SciPy ekosistema. Scikit-learn obezbeđuje širok raspon važnih algoritama mašinskog učenja, uključujući klasifikaciju, regresiju, klasterovanje i validaciju modela. Možete da pronađete više informacija o scikit-learn paketu na adresi http://scikit-learn.org/.
- **pandas**: pandas je softverska biblioteka otvorenog koda. Sadrži tabelarne složene strukture podataka koje se koriste za ulaz, izlaz i obradu tabelarnih podataka u različitim algoritmima. Pandas biblioteka sadrži mnogo korisnih funkcija i, takođe, obezbeđuje visoko optimizovane performanse. Više detalja o pandas biblioteci možete pronaći na adresi http://pandas.pydata.org/.
- **Matplotlib**: Matplotlib obezbeđuje alatke za kreiranje moćnih vizuelizacija. Podaci mogu da budu predstavljeni kao linijski grafikoni, grafikoni rasturanja, trakasti dijagrami, histogrami, kružni dijagrami i tako dalje. Više informacija možete pronaći na adresi https://matplotlib.org/.
- **Seaborn**: Seaborn možete zamisliti kao sličan popularnoj `ggplot2` biblioteci u R-u. Zasnovana je na Matplotlib-u i obezbeđuje napredni interfejs za crtanje brilijantnih statističkih grafikona. Više detalja možete pronaći na adresi https://seaborn.pydata.org/.
- **iPython**: iPython je poboljšana interaktivna konzola koja je dizajnirana da bi olakšala pisanje, testiranje i ispravljanje grešaka Python koda.
- **Pokretanje Python programa**: Interaktivni režim programiranja je koristan za učenje i eksperimentisanje sa kodom. Python programi mogu da budu snimljeni u tekstualni fajl upotrebom ekstenzije .py, a taj fajl može da se pokrene iz konzole.

Implementiranje Pythona pomoću Jupyter Notebooka

Još jedan način da pokrenete Python programer je pomoću Jupyter Notebooka. Jupyter Notebook obezbeđuje korisnički interfejs zasnovan na pretraživaču za razvoj koda. Jupyter Notebook je upotrebljen za predstavljanje primera koda u ovoj knjizi. Mogućnost označivanja i opisivanja koda pomoću tekstova i grafika čini ga odličnom alatkom za predstavljanje i objašnjenje algoritama i odlična je alatka za učenje.

Da bismo započeli notebook, potrebno je da pokrenemo `Jupyter-notebook` proces, a zatim da otvorimo omiljeni pretraživač i u njemu stranicu http://localhost:8888:



Imajte na umu da se Jupyter Notebook sastoji iz različitih blokova koji se nazivaju ćelije.

TEHNIKE DIZAJNIRANJA ALGORITMA

Algoritam je matematičko rešenje za problem iz stvarnog sveta. Kada dizajniramo algoritam, trebalo bi da imamo na umu sledeća tri problema dizajna, dok radimo na dizajnu i podešavamo algoritme:

- **Problem 1**: Da li algoritam proizvodi rezultate koje očekujemo?
- **Problem 2**: Da li je ovo najoptimalniji način da dobijemo ove rezultate?
- **Problem 3**: Kako će se algoritam izvršiti na većim skupovima podataka?

Važno je da razumemo složenost samog problema, pre nego što dizajniramo rešenje za njega. Na primer, korisno je da dizajniramo odgovarajuće rešenje ako karakterišemo problem u pogledu njegovih potreba i složenosti. Generalno, algoritmi mogu da budu podeljeni na sledeće tipove, na osnovu karakteristika problema:

- **Algoritmi koji intenzivno koriste podatke**: Algoritmi koji intenzivno koriste podatke su dizajnirani da obrađuju velike količine podataka. Očekuje se da imaju relativno jednostavne zahteve obrade. Algoritam kompresije primenjen na ogroman fajl je dobar primer algoritama koji intenzivno koriste podatke. Za takve algoritme, veličina podataka se očekuje da bude mnogo veća od memorije procesnog mehanizma (jedan čvor ili klaster) i možda će morati da bude razvijen dizajn iterativne obrade, za efikasnu obradu podataka u skladu sa zahtevima.
- **Računski intenzivni algoritmi**: Računski intenzivni algoritmi imaju značajne zahteve obrade, ali ne uključuju velike količine podataka. Jednostavan primer je algoritam za pronalaženje veoma velikih prostih brojeva. Pronalaženje strategije za deljenje algoritma u različite faze, tako da su barem neke od faza paralelizovane, je ključ za maksimiziranje performanse algoritma.
- **Računski intenzivni algoritmi koji intenzivno koriste podatke**: Postoje određeni algoritmi koji koriste velike količine podataka i, takođe, imaju značajne računske zahteve. Algoritmi upotrebljeni za izvršavanje analize sentimenta u živim video snimcima su dobar primer gde su zahtevi za podatke i za obradu ogromni za izvršavanje zadatka. Takvi algoritmi su algoritmi koji veoma intenzivno koriste resurse i zahtevaju pažljivo projektovanje algoritma i inteligentnu dodelu dostupnih resursa.

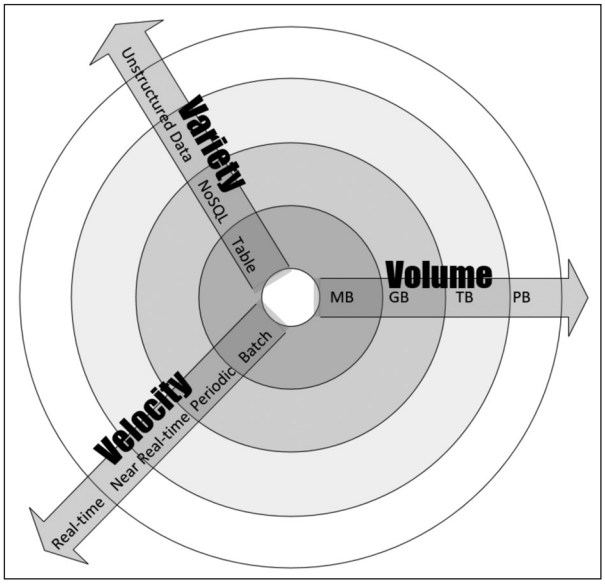
Da bismo karakterisali problem u pogledu njegove složenosti i potreba, korisno je da istražimo njegove podatke i izračunamo dimenzije detaljnije, što ćemo i uraditi u sledećem odeljku.

Dimenzija podataka

Da bismo kategorizovali dimenziju podataka problema, pogledaćemo njegov **obim**, **brzinu** i **brojnost tipova (3Vs)**, koji su definisani na sledeći način:

- **Obim** (eng. *Volume*): Obim je očekivana veličina podataka koje će algoritam obraditi.
- **Brzina** (eng. *Velocity*): Brzina je očekivana brzina generisanja novih podataka kada je upotrebljen algoritam. Može da bude nula.
- **Brojnost tipova** (eng. *Variety*): Brojnost tipova označava koliko različitih tipova podataka se očekuje da će dizajnirani algoritam obraditi.

Na sledećoj slici detaljnije je prikazano 3V-a podataka. Centar ovog dijagrama prikazuje najjednostavnije moguće podatke, sa malim obimom i niskom brojnošću tipova i brzine. Kako se udaljavamo od centra, složenost podataka se povećava. Može da se poveća u jednoj ili u više od tri dimenzija. Na primer, u dimenziji brzine, imamo **Batch** proces kao najjednostavniji, zatim **Periodic**, a zatim **Near Real--Time proces**. Na kraju imamo Real-Time proces, koji je najsloženiji za obradu u kontekstu brzine podataka. Na primer, kolekcija od živih video snimaka sakupljenih grupom praćenih kamera će imati veliki obim, veliku brzinu i veliku brojnost tipova i možda će biti potreban odgovarajući dizajn da bismo mogli efikasno da skladištimo i obradimo podatke. Sa druge strane, jednostavan `.csv` fajl kreiran u Excelu će imati mali obim, malu brzinu i malu brojnost tipova:



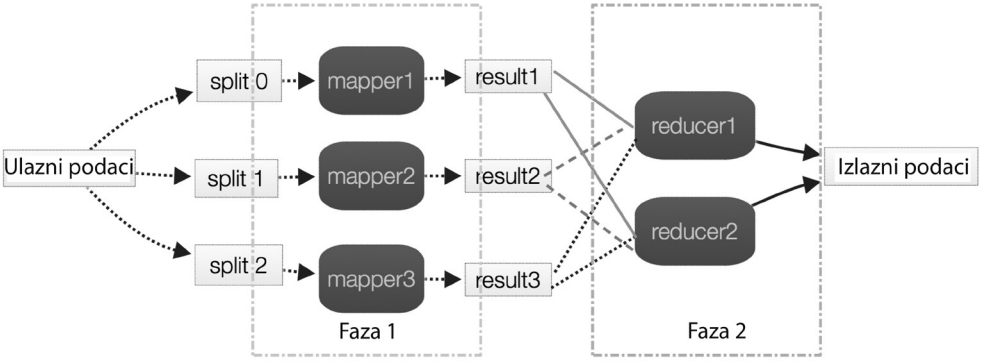
Na primer, ako je ulazni podatak jednostavan `csv` fajl, onda će obim, brzina i brojnost tipova podataka biti niska. Sa druge strane, ako su ulazni podaci živi strim bezbednosnih video kamera, onda će obim, brzina i brojnost tipova podataka biti prilično visoki i ovaj problem bi trebalo da imamo na umu dok dizajniramo algoritam za njega.

Izračunavanje dimenzije

Izračunavanje dimenzije podrazumeva potrebe obrade i izračunavanja za dati problem. Zahtevi obrade algoritma će odrediti koja vrsta dizajna je najefikasnija za njega. Na primer, algoritmi dubokog učenja, generalno, zahtevaju mnogo procesne moći. To znači da je za algoritme dubokog učenja važno imati paralelnu arhitekturu, sa više čvorova, gde god je to moguće.

Praktičan primer

Pretpostavimo da želimo da izvršimo analizu sentimenta na video snimku. Analiza sentimenta je mesto gde pokušavamo da označimo različite delove video snimka sa ljudskim emocijama tuge, sreće, straha, radosti, frustracije i ekstaze. To je računski intenzivan zadatak za koji je potrebno mnogo računaske snage. Kao što ćete videti na sledećoj slici, da bismo dizajnirali izračunavanje dimenzije, podelili smo obradu u pet zadataka koji se sastoje iz dve faze. Sve transformacije podataka i priprema su implementirane u tri mapera. Za to ćemo podeliti video na tri različite particije, koje se nazivaju **razdelnici** (eng. *split*). Nakon što su maperi izvršeni, rezultirajući obrađeni video snimak je ubačen u dva agregatora, koji se nazivaju **reduktori** (eng. *reducer*). Da bismo izvršili potrebnu analizu sentimenta, reduktori grupišu video snimak u skladu sa emocijama. Na kraju, rezultati su kombinovani u izlaz:



Imajte na umu da se broj mapera direktno prevodi u izvršni paralelizam algoritma. Optimalan broj mapera i reduktora zavisi od karakteristike podataka, tipa algoritma koji treba da se upotrebi i broja dostupnih resursa.

ANALIZA PERFORMANSE

Analiza performanse algoritma je važan deo njegovog dizajna. Jedan od načina da procenimo performansu algoritma je da analiziramo njegovu složenost.

Teorija složenosti je istraživanje koliko su komplikovani algoritmi. Da bi bio koristan, algoritam bi trebalo da ima tri ključne funkcije:

- Trebalo bi da bude tačan. Algoritam neće biti koristan ako vam ne daje odgovarajuće odgovore.
- Dobar algoritam bi trebalo da bude razumljiv. Najbolji algoritam na svetu neće biti koristan ako je previše komplikovan za implementiranje na računaru.
- Dobar algoritam bi trebalo da bude efikasan. Čak i ako algoritam proizvodi tačan rezultat, neće vam mnogo pomoći ako se izvršava hiljadu godina ili ako zahteva milijardu terabajtova memorije.

Postoje dva moguća tipa analize za kvantifikovanje složenosti algoritma:

- **Analiza prostorne složenosti**: Procenjuje zahteve memorije potrebne za izvršavanje algoritma.
- **Analiza vremenske složenosti**: Procenjuje vreme koje je algorimu potrebno za pokretanje.

Analiza prostorne složenosti

Analiza prostorne složenosti procenjuje količinu potrebne memorije da bi algoritam obradio ulazne podatke. Dok obrađuje ulazne podatke, algoritam treba da sačuva privremene strukture podataka tranzijenta u memoriji. Način na koji je algoritam dizajniran utiče na broj, tip i veličinu ovih struktura podataka. U doba distribuiranog računarstva i sa sve većim količinama podataka koje je potrebno obraditi, analiza prostorne složenosti postaje sve važnija. Veličina, tip i broj ovih struktura podataka će odrediti zahteve za memoriju za osnovni hardver. Moderne strukture podataka u memoriji upotrebljene za distribuirano izračunavanje – kao što je **Resilient Distributed Datasets (RDD)** – trebalo bi da imaju efikasne mehanizme za dodelu resursa, koji su svesni zahteva za memorijom u različitim fazama izvršenja algoritma.

Analiza prostorne složenosti je obavezna za efikasno dizajniranje algoritama. Ako nije izvršena pravilna analiza prostorne složenosti tokom dizajniranja određenog algoritma, nedovoljno dostupne memorije za privremene strukture podataka tranzijenta može da dovede do nepotrebnog prelivanja diska, što potencijalno može znatno da utiče na performansu i efikasnost algoritma.

U ovom poglavlju ćemo detaljnije govoriti o vremenskoj složenosti. Prostorna složenost će biti detaljno opisana u Poglavlju 13, *Algoritmi velikih razmera*, gde ćemo opisati distribuirane algoritme velikih razmera sa složenim memorijskim zahtevima izvršenja.

Analiza vremenske složenosti

Analiza vremenske složenosti procenjuje koliko vremena je potrebno algoritmu da izvrši dodeljeni zadatak na osnovu svoje strukture. Za razliku od prostorne složenosti, vremenska složenost ne zavisi od hardvera na kojem će se algoritam pokretati. Analiza vremenske složenosti zavisi samo od strukture samog algoritma. Uopšteni cilj analize vremenske složenosti je da pokuša da odgovori na ova važna pitanja – da li će se algoritam skalirati? Koliko dobro će algoritam rukovati većim skupovima podataka?

Da bismo odgovorili na ova pitanja, trebalo bi da odredimo efekat algoritma na performansu dok se veličina podataka povećava i da se uverimo da je algoritam dizajniran na način koji, ne samo da ga čini tačnim, već čini da se algoritam dobro skalira. Performansa algoritma postaje sve važnija za veće skupove podataka u današnjem svetu „velikih podataka“.

U mnogim slučajevima, možemo imati više od jednog pristupa koji su dostupni za dizajniranje algoritma. Cilj izvršavanja analize vremenske složenosti, u ovom slučaju, biće sledeći:

> *„Uzimajući u obzir određeni problem i više od jednog algoritma, koji je najefikasniji za upotrebu u pogledu vremenske efikasnosti?"*

Mogu postojati dva osnovna pristupa za izračunavanje vremenske složenosti algoritma:

- **Pristup profilisanjem nakon implementacije**: U ovom pristupu, implementirani su različiti kandidati algoritama i upoređene su njihove performanse.
- **Teoretski pristup pre implementacije**: U ovom pristupu, matematički je približno određen performans svakog algoritma pre pokretanja algoritma.

Prednost teoretskog pristupa je što zavisi samo od strukture samog algoritma. Ne zavisi od hardvera koji će biti upotrebljen za pokretanje algoritma, izbora softverskog steka prilikom izvršenja, ili programskog jezika koji je upotrebljen za implementiranje algoritma.

Procena performansi

Performans tipičnog algoritma će zavisiti od tipa podataka koji su mu dati kao ulaz. Na primer, ako su podaci već sortirani u skladu sa kontekstom problema koji pokušavamo da rešimo, algoritam može da se izvršava neverovatno brzo. Ako je upotrebljen sortiran ulaz za testiranje određenog algoritma, tada će rezultat biti nerealno dobra performansa, što nije stvarna refleksija njegovih realnih performansi, u većini situacija. Da bismo obradili ovu zavisnost algoritma u ulaznim podacima, trebalo bi da razmotrimo različite tipove slučajeva kada izvršavamo analizu performansi.

Najbolji slučaj

U najboljem slučaju, podaci koji su dati kao ulaz su organizovani tako da će algoritam pružiti svoj najbolji performans. Analiza najboljeg slučaja daje gornju granicu performansi.

Najgori slučaj

Drugi način da procenimo performans algoritma je da pokušamo da pronađemo maksimalno moguće vreme koje je potrebno za izvršenje zadatka pod datim skupom uslova. Ova analiza najgoreg slučaja algoritma je prilično korisna jer možemo da garantujemo, bez obzira na uslove, da će performans algoritma uvek biti bolji od brojeva koje dobijemo u rezultatu analize. Analiza najgoreg slučaja je posebno korisna za procenu performansa kada rešavamo složene probleme sa većim skupovima podataka. Analiza najgoreg slučaja daje donju granicu performansi algoritma.

Prosečan slučaj

Prosečan slučaj započinje deljenjem različitih mogućnih ulaza u različite grupe. Zatim se izvršava analiza performansi iz jednog reprezentativnog ulaza iz svake grupe. Na kraju, izračunava se prosek performansi svake grupe.

Analiza prosečnog slučaja nije uvek tačna, jer treba da razmotrimo sve različite kombinacije i mogućnosti ulaza za algoritam, što nije uvek jednostavno.

Selektovanje algoritma

Kako da znate koji algoritam je bolje rešenje? Kako da znate koji se algoritam pokreće brže? Vremenska složenost i Big O notacija (opisana kasnije u ovom poglavlju) su stvarno dobre alatke za odgovor na ove tipove pitanja.

Da biste videli gde može da bude koristan, uzmimo jednostavan primer u kojem je cilj da sortiramo listu brojeva. Postoji nekoliko dostupnih algoritama koji mogu da izvrše zadatak. Problem je kako da izaberemo pravi.

Prvo, primedba koju možemo da damo je da ako nema previše brojeva na listi, onda nije važno koji ćemo algoritam da izaberemo za sortiranje liste brojeva. Dakle, ako postoji samo 10 brojeva u listi (n=10), onda nije važno koji ćemo algoritam izabrati, jer verovatno izvršenje neće potrajati više od nekoliko milisekundi, čak i sa veoma loše dizajniranim algoritmom. Ali, ako veličina liste postaje milion brojeva, onda je izbor pravog algoritma veoma važna. Veoma loše napisan algoritam može da potraje i nekoliko sati za pokretanja, dok dobro dizajnirani algoritam može da izvrši sortiranje liste za nekoliko sekundi. Dakle, za veće ulazne skupove podataka, ima smisla investirati vreme i trud, izvršiti analizu performansi i izabrati korektno dizajniran algoritam, koji će izvršiti potreban zadatak na efikasan način.

Big O notacija

Big O notacija se koristi za kvantifikovanje performansi različitih algoritama kako se povećava veličina ulaza. Big O notacija je jedna od popularnih metodologija koja se koristi za izvršavanje analize najgoreg slučaja. Različite vrste Big O notacija opisane su u ovom odeljku.

Konstantna vremenska (O(1)) složenost

Ako algoritam koristi istu količinu vremena za pokretanje, nezavisno od veličine ulaznih podataka, kaže se da se pokreće u konstantnom vremenu i predstavljeno je sa O(1). Pogledajmo primer pristupa n-tom elementu niza. Bez obzira na veličinu niza, potrebno je konstantno vreme da se dobiju rezultati. Na primer, sledeća funkcija će vratiti prvi element niza i ima složenost O(1):

```
def getFirst(myList):
    return myList[0]
```

Rezultat je prikazan kao:

```
In [2]:  1 getFirst([1,2,3])
Out[2]: 1

In [3]:  1 getFirst([1,2,3,4,5,6,7,8,9,10])
Out[3]: 1
```

- Dodatak novog elementa u `stek`, upotrebom komande `push`, ili uklanjanje elementa iz steka, upotrebom komande `pop`. Bez obzira na veličinu steka, potrebna je ista količina vremena za dodavanje ili uklanjanje elementa.
- Pristupanje elementu heš tabele (opisano u Poglavlju 2, *Strukture podataka upotrebljene u algoritmima*).
- Bucket sortiranje (opisano u Poglavlju 2, *Strukture podataka upotrebljene u algoritmima*).

Linearna vremenska (O(n)) složenost

Za algoritam se kaže da ima linearnu vremensku složenost, predstavljenu sa O(n), ako je vreme izvršenja direktno proporcionalno veličini ulaza. Jednostavan primer je da dodamo elemente u jednodimenzionalnu strukturu podataka:

```
def getSum(myList):
     sum = 0
     for item in myList:
          sum = sum + item
     return sum
```

Vidite glavnu petlju algoritma. Broj iteracija u glavnoj petlji se povećava linearno sa povećanjem vrednosti *n*, kreirajući O(n) složenost na sledećoj slici:

```
In [5]:  1 getSum([1,2,3])
Out[5]: 6

In [6]:  1 getSum([1,2,3,4])
Out[6]: 10
```

Neki drugi primeri operacija niza su sledeći:

- Pretraživanje elementa
- Pronalaženje minimalne vrednosti među svim elementima niza.

Kvadratna vremenska (O(n2)) složenost

Za algoritam se kaže da se pokreće u kvadratnom vremenu ako je vreme izvršenja algoritma proporcionalno sa kvadratom ulazne veličine; na primer, jednostavna funkcija koja rezimira dvodimenzionalni niz je sledeća:

```
def getSum(myList):
     sum = 0
     for row in myList:
          for item in row:
               sum += item
     return sum
```

Vidite da je ugnežđena unutrašnja petlja unutar druge, glavne petlje. Ova ugnežđena petlja daje prethodnom kodu složenost O(n2):

```
In [8]:   1 getSum([[1,2],[3,4]])
Out[8]: 10

In [9]:   1 getSum([[1,2,3],[4,5,6]])
Out[9]: 21
```

Još jedan primer je algoritam mehurastog sortiranja (opisano u Poglavlju 2, *Strukture podataka upotrebljene u algoritmima*).

Logaritamska vremenska (O(logn)) složenost

Za algoritam se kaže da se pokreće u logaritamskom vremenu ako je vreme izvršenja algoritma proporcionalno sa logaritmom ulazne veličine. Sa svakom iteracijom, ulazna veličina se samnjuje za konstantan višestruki faktor. Primer logaritamske složenosti je binarna pretraga. Algoritam binarne pretrage se koristi za pronalaženje određenog elementa u jednodimenzionalnoj strukturi podataka, kao što je Python lista. Elementi unutar strukture podataka treba da budu sortirani u opadajućem redosledu. Algoritam binarne pretrage je implementiran u funkciji pod nazivom `searchBinary`, na sledeći način:

```
def searchBinary(myList,item):
    first = 0
    last = len(myList)-1
    foundFlag = False
    while( first<=last and not foundFlag):
        mid = (first + last)//2
        if myList[mid] == item :
            foundFlag = True
        else:
            if item < myList[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return foundFlag
```

Glavna petlja koristi činjenicu da je lista uređena i deli listu na pola sa svakom iteracijom, dok ne dođe do rezultata:

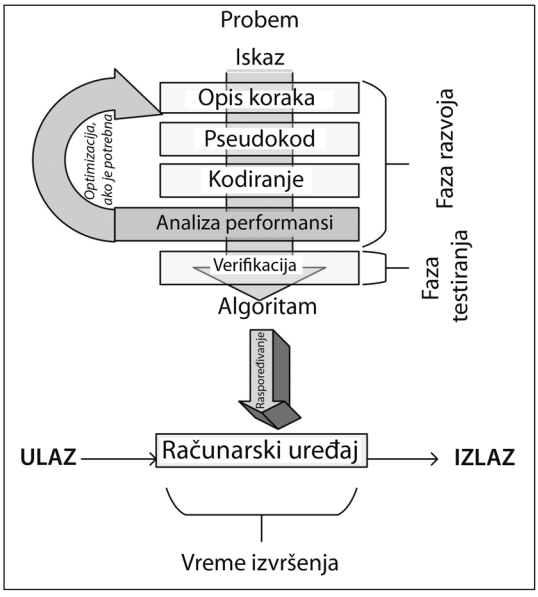
```
In [11]:  1 searchBinary([8,9,10,100,1000,2000,3000], 10)
          2
Out[11]: True

In [12]:  1 searchBinary([8,9,10,100,1000,2000,3000], 5)
Out[12]: False
```

Nakon definisanja funkcije, algoritam je testiran za pretragu određenog elementa u linijama 11 i 12. Algoritam binarne pretrage je detaljnije opisan u Poglavlju 3, *Algoritmi za sortiranje i pretraživanje*.

Imajte na umu da među četiri tipa Big O notacije koji su predstavljeni, O(n2) ima najgore performanse a O(logn) ima najbolje performanse. U stvari, performans O(logn) tipa može da se smatra zlatnim standardom za performans bilo kog algoritma (što se ne postiže uvek). Sa druge strane, $O(n^2)$ nije toliko loš kao $O(n^3)$ ali, ipak, algoritmi koji pripadaju ovoj klasi ne mogu da se upotrebe u velikim skupovima podataka, jer vremenska složenost postavlja ograničenja u količini podataka koji se stvarno mogu obraditi.

Jedan način da smanjite složenost algoritma je da kompromitujete njegovu tačnost, kreirajući tip algoritma pod nazivom **aproksimacioni algoritam**.

Ceo proces procene performansi algoritama je iterativne prirode, kao što je i prikazano na sledećoj slici:



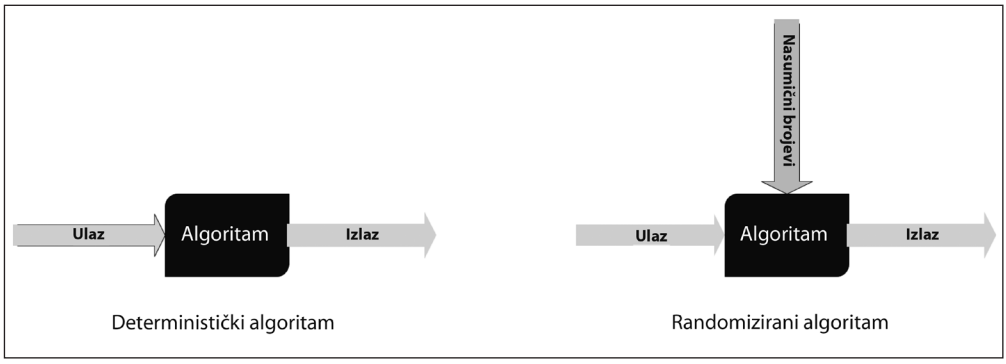
VALIDACIJA ALGORITMA

Validacija algoritma potvrđuje da obezbeđuje matematičko rešenje za problem koji pokušavamo da rešimo. Proces validacije bi trebalo da proverava rezultate za što je moguće više vrednosti i tipova ulaznih vrednosti.

Tačni, aproksimacioni i randomizirani algoritmi

Validacija algoritma, takođe, zavisi i od tipa algoritma, jer se tehnike testiranja razlikuju. Prvo ćemo objasniti razliku između determinističkih i randomiziranih algoritama.

Za deterministične algoritme, određeni unos uvek generiše potpuno isti izlaz. Ali za određene klase algoritama, sekvenca nasumičnih brojeva je, takođe, upotrebljena kao ulaz, što izlaz čini različitim svaki put kada se algoritam pokrene. Algoritam klasterovanja k-srednjih vrednosti, koji je detaljno opisan u Poglavlju 6, *Algoritmi nenadgledanog mašinskog učenja,* je primer takvog algoritma:



Algoritmi takođe mogu da budu podeljeni u sledeća dva tipa, na osnovu pretpostavke ili aproksimacije upotrebljene za pojednostavljenje logike, što čini da se pokreću brže:

- **Tačan algoritam**: Za tačne algoritme se očekuje da kreiraju precizno rešenje bez uvođenja ikakvih pretpostavki i aproksimacija.
- **Aproksimacioni algoritam**: Kada je složenost problema prevelika za dati resurs, pojednostavićemo problem nekim pretpostavkama. Algoritmi zasnovani na ovim pojednostavljenjima ili pretpostavkama se nazivaju aproksimacioni algoritmi, koji nam ne daju precizno rešenje.

Pogledajmo primer da bismo razumeli razliku između tačnih i aproksimacionih algoritama – poznati problem trgovačkog putnika, koji je predstavljen 1930.godina. Trgovački putnik vas izaziva da pronađete najkraći put za određenog trgovca koji posećuje svaki grad (iz liste gradova), a zatim se vraća na početnu tačku, zbog čega se i naziva trgovački putnik. Prvi pokušaj da obezbedimo rešenje će uključivati generisanje svih permutacija gradova i biranje kombinacije gradova koja ja najjeftinija. Složenost ovog pristupa za obezbeđivanje rešenja je O(n!), gde n predstavlja broj gradova. Očigledno je da vremenska složenost postaje teška za upravljanje za više od 30 gradova.

Ako je broj gradova veći od 30, jedan način da smanjimo složenost je da predstavimo neke aproksimacije i pretpostavke.

Za aproksimacione algoritme, važno je da podesimo očekivanja za tačnost kada prikupljamo zahteve. Validacija aproksimacionog algoritma je, u stvari, verifikovanje da je greška rezultata unutar prihvatljivih granica.

Objašnjenje

Kada su algoritmi upotrebljeni za važne slučajeve, važno imati mogućnost objašnjenja razloga svakog rezultata, kada god je to potrebno. To je važno da bismo se uverili da odluke zasnovane na rezultatima algoritama ne predstavljaju odstupanja.

Mogućnost tačnog identifikovanja funkcija koje se koriste direktno ili indirektno za donošenje određene odluke naziva se **objašnjenje algoritma**. Algoritmi, kada se koriste za važne slučajeve upotrebe, trebalo bi da budu procenjeni za odstupanja i predrasude. Etička analiza algoritama postala je standardni deo validacionog procesa za one algoritme koji mogu da utiču na donošenje odluka koje se odnose na život ljudi.

Za algoritme koji se koriste za duboko učenje, objašnjenje je teško postići. Na primer, ako se algoritam koristi za odbijanje prijave za hipoteku osobi, važno je da imamo transparentnost i mogućnost objašnjenja razloga.

Algoritamsko objašnjenje je aktivna oblast istraživanja. Jedna od efikasnih tehnika koja je nedavno razvijena je **Local Interpretable Model-Agnostic Explanations (LIME)**, kao što je predstavljeno na **22. Association for Computing Machinery (ACM)** na **Special Interest Group on Knowledge Discovery (SIGKDD)** međunarodnoj konferenciji otkrića i istraživanja podataka 2016.godine. LIME je zasnovana na konceptu u kojem se male promene uvode u ulaz za svaku instancu, a zatim se mapira lokalna granica odluke za tu instancu. Zatim se kvantifikuje uticaj svake promenljive za datu instancu.

REZIME

U ovom poglavlju smo učili o osnovama algoritama. Prvo smo učili o različitim fazama razvoja algoritma. Govorili smo o različitim načinima specifikovanja logike algoritma, koja je potrebna za njegovo dizajniranje. Zatim smo opisali kako da dizajniramo algoritam. Učili smo o dva različita načina analiziranja performanse algoritma. Na kraju smo učili o različitim aspektima validacije algoritma.

Nakon ovog poglavlja bi trebalo da razumemo pseudokod algoritma. Trebalo bi da razumemo različite faze u razvoju i raspoređivanju algoritma. Takođe smo učili kako da koristimo Big O notaciju za procenu performansi algoritma.

U sledećem poglavlju ćemo govoriti o strukturama podataka koje se koriste u algoritmima. Prvo ćemo govoriti o strukturama podataka koje su dostupne u Pythonu. Zatim ćemo pregledati kako možemo da upotrebimo ove strukture podataka za kreiranje sofisticiranijih struktura podataka, kao što su stekovi, redovi i stabla, koji su nam potrebni da bismo kreirali složene algoritme.