

Osnovne algoritamske paradigme kroz programski jezik Pajton

Marko Đukanović

Avgust 2023

Sadržaj

1	Neformalni uvod u teoriju algoritama	3
1.1	Računarski problem	3
1.2	Računarski program i pojam algoritma	5
1.2.1	Vrste algoritama	7
1.3	Teorijski RAM model	9
1.3.1	Formalizacija računanja na RAM-u	11
1.4	Pojam kompleksnosti	13
1.4.1	O -notacija	15
1.4.2	Neka svojstva O -notacije	18
1.4.3	Uputstva za procjenu kompleksnosti algoritama	18
1.5	Prostorna kompleksnost	21
1.6	Pseudokod	22
1.7	O valjanosti algoritma	25
1.8	Euklidov algoritam: kompleksnost	27
2	Uvod u programski jezik Pajton	35
2.1	Osnovne karakteristike	36
2.2	Pregled ugrađenih tipova podataka	37
2.3	Deklaracija varijabli i memorijska organizacija	38
2.4	Osnovni numerički tipovi podataka	42
2.5	Naredba grananja i petlje	42
2.5.1	Uslovna naredba If	43
2.5.2	For i While petlje	44
2.6	Složene strukture podataka	45
2.6.1	Stringovni tip podataka	45
2.6.2	Liste	47
2.6.3	Nizovi (Array)	49
2.6.4	Rječnici	51

2.6.5	Skupovi	53
2.6.6	Torke	56
2.7	Funkcije	57
2.8	Kompajliranje i izvršavanje programa	62
2.8.1	Stek vs. Hip	62
2.8.2	Kompajliranje i izvršavanje	63
2.9	O nekim programskim konceptima i ugrađenim funkcijama	65
2.9.1	Komprehenzija liste	65
2.9.2	Funkcije map, filter, reduce	66
2.9.3	Funkcije zip, enumerate i sort	67
2.10	Uopšteno o modulima	69
2.10.1	Instaliranje eksternih modula	70
2.10.2	Modul Math	70
2.10.3	Modul Random	71
2.10.4	Modul Time	72
2.10.5	Modul Os	73
2.11	Izuzeci	74
2.12	Rad sa datotekama	76
2.12.1	Modul Json	77
2.13	Regularni izrazi	78
2.13.1	Uopšteno o regularnim izrazima	78
2.13.2	Modul Re	79
2.14	Moduli Sys i Getopt	80
2.15	Modul Numpy	83
2.16	Modul Ctypes	86
2.17	Pisanje prilagodjenih modula	87
2.18	Postavljanje korisnički kreirani modul na PyPI repozitorij	88
3	Algoritamske paradigme	95
3.1	Rekurzivni algoritmi	97
3.2	Vrste rekurzija	99
3.2.1	Prednosti i nedostaci rekurzivnog nad iterativnim pristupom	100
3.2.2	Primjene rekurzivnog pristupa	101
3.2.3	Primjena rekurzivnog pristupa	102
3.3	Algoritmi grube sile	105
3.4	Algoritmi pretraživanja	109
3.4.1	Linearna pretraga	109

3.4.2	Binarna pretraga	110
3.4.3	Pretraga preskakanjem	112
3.4.4	Interpolacijska pretraga	113
3.4.5	Eksponencijalna pretraga	115
3.5	Algoritamska paradigma podijeli pa zavlada	116
3.6	Algoritamska paradigma vraćanja unazad	119
3.7	Pohlepni algoritmi	124
3.7.1	Problem razmjene novca	125
3.7.2	Dijeljeni problem ruksaka	127
3.8	Dinamičko programiranje	134
3.8.1	Rekurzija vs. Dinamičko programiranje	136
3.8.2	Primjene dinamičkog programiranja	137
4	Primjena algoritamskih paradigmi na probleme iz aritmetike	153
4.1	Eratostenovo sito	153
4.2	Faktorizacija broja	156
4.3	Izračunavanje binomnih koeficijenata	158
4.4	Izračunavanje Katalanovih brojeva	160
4.5	Još neki zadaci	162

Predgovor

Ova skripta je napisana za predmet Proceduralno programiranje, koji se obrađuje na prvoj godini Studijskog programa Matematike i informatike – smjer Informatika na Prirodno-matematičkom fakultetu, Univerziteta u Banjoj Luci. Skripta se sastoji od četiri dijela. Prvi dio obrađuje osnovne pojmove poput pojma programa, algoritma, modela izračunavanja, ali bez ulaženja u širinu i glavne detalje, koji se po planu izučavaju na višim godinama iz kurseva Algoritmike i srodnih kurseva. Drugi dio je vezan za osnove programskog jezika Pajton, prvo se obrađuju osnovni tipovi podatka, pa složeni tipovi podataka, bitni ugrađeni moduli, pa sve od pisanja korisničkih modula i njihovog postavljanja na zvanični repozitorij PyPI. Treći dio se odnosi na konceptualizaciju osnovnih algoritamskih paradigmi poput paradigme brutalne sile, podijeli pa zavladaaj, pohlepnih algoritama, dinamičkog programiranja, te njihovih primjena na rješavanje raznih tipova problema. Četvrti dio je vezan za probleme iz aritmetike i njihovo efikasno rješavanje pomoću raznih algoritamskih paradigmi.

Glava 1

Neformalni uvod u teoriju algoritama

1.1 Računarski problem

Računarski problem u računarskim naukama se jednostavno i slobodno može definisati kao zadatak koji je potrebno riješiti uz pomoć računara.

Formalna definicija ovog pojma zahtjeva uvođenje mnogih kompleksnih pojmova poput definicije jezika. U računarskom problemu, dat nam je ulaz za koji, bez gubitka opštosti, pretpostavljamo da je kodiran binarno kao niz iz skupa binarnih nizova proizvoljne dužine $L = \{0, 1\}^*$, a izlaz je rješenje koje zadovoljava neko svojstvo: računarski problem opisujemo pomoću svojstava koje izlaz mora zadovoljiti s obzirom na zadani ulaz. Ulaz se često naziva *instanca* problema, dok je izlaz *rješenje* problema s obzirom na ulaz. Primjer računarskog programa u računarskim naukama je sortiranje numeričkih nizova. Kao ulaz dat je niz brojeva dužine n . Kao izlaz problema sortiranja dobijamo uređen niz (po nekom unaprijed zadanom kriterijumu sortiranja). Međutim, ne moraju samo numerički nizovi da se sortiraju, već uopšteno, bilo koji objekti nad kojima možemo definisati relaciju totalnog uređenja. Zbog toga ima smisla govoriti o rješavanju računskog problema kao uparivanju ulazno-izlaznih parametara. Ulazni i izlazni parametri mogu biti predstavljeni nekim matematičkim objektima kao što su brojevi, skupovi, funkcije, nizovi itd. Ulazni dio problema tada nazivamo *generička instanca* problema, dok izlazni nazivamo *generiško rješenje* generičke instance problema.

Postoji nekoliko tipova problema u računarskim naukama; izdvajamo

četiri često prisutna u literaturi: problemi odlučivanja, problemi pretraživanja, problemi optimizacije i problemi prebrojavanja.

U problemu *odlučivanja*, s obzirom na ulaz $x \in \{0, 1\}^*$, od nas se traži izlaz DA/NE (*True/False*). Dakle, u problemu odlučivanja se traži da provjerimo da li instanca problema zadovoljava određeno svojstvo ili ne. Primjer problema odlučivanja je problem *3-bojanja* (3-COL): U ulazu problema imamo neusmjereni graf. Potrebno je odrediti da li postoji način dodjele “boje” iz skupa $\{1, 2, 3\}$ svakom čvoru grafa na takav način da nijedna dva susjedna čvora nemaju istu dodjeljenu boju.

Prikladan način za specificiranje problema odlučivanja je nuđenje skupa $L \subseteq \{0, 1\}^*$ ulaza za koje je odgovor DA. Podskup skupa $\{0, 1\}^*$ se takođe naziva jezikom, tako da, uz prethodnu konvenciju, svaki problem odlučivanja može biti specificiran pomoću jezika (a svaki jezik specificira problem odlučivanja). Na primjer, ako sa 3-COL imenujemo podskup skupa $\{0, 1\}^*$ koji sadrži (opise) 3-obojuje grafove, tada 3-COL predstavlja jezik koji specificira problem 3-bojanja.

U problemu pretraživanja, obzirom na ulaz $x \in \{0, 1\}^*$, potrebno je izračunati neki odgovor $y \in \{0, 1\}^*$ koji je u nekom odnosu sa x , ako takav postoji. Dakle, problem pretraživanja je definisan relacijom $R \subseteq \{0, 1\}^* \times \{0, 1\}^*$, gdje $(x, y) \in R$ ako i samo ako je y dopustiv odgovor za dati x .

Razmotrimo problem pretraživanja u vezi sa problemom 3-COL: dat je neusmjereni graf $G = (V, E)$ potrebno je pronaći, ukoliko postoji, bojanje $c: V \rightarrow \{1, 2, 3\}$ čvorova, tako da za svaki $(u, v) \in E$ vrijedi $c(u) \neq c(v)$. Ovakvo definisan problem je sada drugačiji (i zahtjevniji) od verzije odlučivanja, jer od nas se traži više od toga – pored utvrđivanja postoji li takav c , traži se da ga konstruišemo, pod uslovom da postoji. Formalno, problem 3-bojanja specificiran je relacijom R_{3-COL} koja sadrži sve parovi (G, c) gdje je G predstavlja 3-obojljivi graf, a c je valjano 3-bojanje grafa G .

Problemi optimizacije predstavljaju matematičke probleme koji uključuju pronalazak najboljeg rješenja iz skupa mogućih rješenja. Cilj je maksimizovati / minimizovati ciljnu funkciju, u odnosu na skup ograničenja dati definicijom problema. Drugim riječima, cilj je pronaći optimalnu vrijednost određene varijable ili skupa varijabli unutar zadanog skupa ograničenja. Primjer problema optimizacije je klasični “problem trgovačkog putnika”. U ovom problemu, osoba mora posjetiti skup gradova i vratiti se u početni grad, sa ciljem da minimizuje ukupnu prijeđenu udaljenost. Problem se sastoji u pronalasku optimalne rute kojom osoba može posjetiti sve gradove (tačno jednom) uz minimaliziranje ukupne pređene udaljenosti.

Problemi prebrojanja su matematički problemi koji uključuje prebrojavanje načina na koje se određeni događaj može dogoditi. Ovi problemi variraju od relativno jednostavnih zadataka kao što je prebrojavanje ljudi u prostoriji do mnogo složenijih scenarija kao što je brojanje mogućih kombinacija slova u riječi. Recimo da imamo četiri košulje različitih boja i troje pantalona različitih boja. Na koliko različitih načina se osoba može obući, ako nosimo jednu košulju i jedne hlače?

Za rješavanje ovog problema možemo upotrijebiti princip množenja, koje kaže da ako imamo n izbora za prvi zadatak i m izbora za drugi zadatak, tada je ukupan broj oblačenja koji se može izvesti je jednak $n \times m$.

Za kraj ovog poglavlja, formalno definišimo računarski problem.

Definicija 1.1 *Računarski problem je funkcija $f: \mathbb{N}^* \rightarrow 2^{\mathbb{N}^*}$. $f(x)$ predstavlja skup tačnih odgovora za ulaz x .*

1.2 Računarski program i pojam algoritma

Računarski program je (konačan) niz instrukcija napisanih u nekom programskom jeziku koje računar može da izvrši. Ove instrukcije govore računaru šta da radi i kako da to uradi u eksplicitno definisanom nizu.

Računarski programi mogu biti dizajnirani za obavljanje specifičnih zadataka ili funkcija, npr. obrada teksta, analiza podataka, ili pregledavanje veba. Razvijaju se za različite platforme i operativne sisteme, uključujući desktop računare, mobilne uređaje, ugrađene sisteme itd. Kompjuterski programi se mogu kreirati korišćenjem raznih programskih jezika, kao što su Pajton, Java, C++ i mnogi drugi. Kada je program napisan, potrebno ga je kompajlirati ili interpretirati u mašinski kôd koji je razumljiv procesoru računara. Rezultirajući izvršni kôd korisnici mogu pokrenuti kako bi izvršili željene zadatke ili funkcije.

Algoritam je korak-po-korak postupak za rješavanje problema ili postizanja određenog cilja. U informatici, algoritmi se su predstavljeni konačnim niza instrukcija koje kompjuterski program slijedi da bi riješio određeni problem ili izvršio određeni zadatak. Algoritam najčešće počinje sa inicijalnim ulazom, koji se zatim obrađenim nizom logičkih koraka ili operacija dovodi do očekivanog izlaza ili rezultata. Analogija algoritma u stvarnom svijetu se može predstaviti primjerom kuhanja po receptu. Da biste skuhalo po receptu, neophodno je čitati uputstva i korake te ih izvršavati jedno po jedno, u

datom redoslijedu. Dobijeni (očekivani) rezultat je jelo koje je kuhano.

Primjeri dobro poznatih algoritama uključuju algoritme za sortiranje koji se koriste za uređivanje podataka u odnosu na relaciju uređenja, algoritme pretraživanja koji služe za pronalaženje određenih informacija na velikom skupu podataka, zatim algoritme dešifrovanja i druge.

Algoritmi su korisni iz više razloga:

- Neophodni su za efikasno i efektivno rješavanje složenih problema.
- Pomažu u automatizaciji procesa – čine ih pouzdanijim, bržim i lakšim za izvođenje.
- Omogućavaju računarima da izvršavaju zadatke koje bi ljudima bilo teško ili nemoguće izvršiti ručno.

Ne određuje svaki niz instrukcija jedan algoritam. Da bi neke instrukcije činile algoritam, sljedeće karakteristike treba da budu ispoštovane:

- *Dobro definisani ulazi*: Ako algoritam zahtjeva podatke u ulazu, podaci bi trebali biti dobro definisani.
- *Dobro definisani rezultati*: Algoritam mora jasno definisati koji će se rezultat dobiti (kao izlaz) u zavisnosti od svakog ulaza.
- *Konačnost*: Algoritam mora biti konačan, tj. trebao bi se prekinuti nakon konačnog vremena za sve testne slučajeve. Beskonačne petlje ili rekurzivne funkcije bez baznih uslova ne posjeduju svojstvo konačnosti; o ovome ćemo govoriti u nastavku.
- *Dopustivost*: Algoritam mora biti generički i praktičan, u smislu da se može izvršiti u okviru dostupnih resursa.
- *Nezavisnost od jezika*: Algoritam koji je dizajniran mora biti nezavisan o jeziku u kojem se implementira; dakle, izlaz invarijantan od jezika imlementacije.
- *Definisanost*: Sve instrukcije u algoritmu moraju biti nedvosmislene, precizne i lake za interpretaciju. Pozivanjem bilo koje instrukcije algoritma, jasno se može razumjeti šta ona radi. Svaki osnovni operator u instrukciji mora biti definisan bez nejasnoća.

Koncizno rečeno, svaki algoritam posjeduje sljedeće karakteristike:

- Završava se nakon određenog vremena.
- Vraća barem jedan izlaz.
- Ima ulaz domenzije nula ili više.
- Posjeduje determinizam, tj. svaki put daje isti izlaz za isti ulazni slučaj.
- Svaki korak u algoritmu je efikasan, tj. svaki korak bi trebao obaviti određeni posao.

Veličina ulaza algoritma je defisana kao ukupan broj elemenata prisutnih u ulazu. Za dva problema, veličina ulaza n može biti okarakterisana na drugačiji način. Npr. u problemu sortiranja to je ukupan broj stavki za sortiranje, dok je to u problemima sa grafovima ukupan broj čvorova i grana, a u numeričkim problemima je to ukupan broj bitova potrebnih za predstavljanje broja.

1.2.1 Vrste algoritama

Na raspolaganju je nekoliko tipova (paradigmi) algoritama. Neke važnije paradigme nabrajamo u nastavku.

1. Algoritmi *brutalne sile* (eng. *brute force*): jedan od najjednostavnijih pristupa rješavanja problema. Ujedno je i prvi pristup koji se konstruiše za rješavanje problema jer je ideja rješenja u pozadini često vrlo naivna te se razmatraju svi mogući slučajevi koji potencijalno mogu predstaviti rješenje problema.
2. *Rekurzivni algoritmi*: Ovi algoritmi su zasnovani na rekurziji. U ovom slučaju, problem se dijeli na nekoliko podproblema te se ista funkcija poziva iznova za rješavanje tih podproblemima, dok god se ne dođe do nekih trivijano rješivih slučajeva.
3. *Algoritam vraćanja unazad*: Ovo je algoritamska tehnika rekurzivnog rješavanja problema koja pokušava postepeno izgraditi rješenje, dio po dio, isključujući momentalno ona rješenja koja ne uspijevaju zadovoljiti ograničenja problema.

4. *Algoritmi pretraživanja*: Ovi algoritmi se koriste za pretraživanje elemenata ili grupa elemenata iz određene strukture podataka. Mogu biti različitih tipova na osnovu njihovog pristupa ili strukture podataka u kojoj se element nalazi.
5. *Algoritmi sortiranja*: Sortiranje je uređivanje grupe podataka shodno nekom kriterijumu. Algoritmi koji pomažu u obavljanju ove funkcije nazivaju se algoritmi sortiranja.
6. *Algoritam podijeli i vladaj*: Ovaj algoritam razbija problem na podprobleme, rješava podprobleme i spaja njihova rješenja kako bi se došlo konačno rješenje. Sastoji se od sljedeće tri procedure: podijeli, riješi, kombinuj. Ovo je, inače, algoritamska paradigma koja se koristi u rješavanju raznih tipova problema poput sortiranja.
7. *Pohlepni algoritmi*: U ovom tipu algoritma rješenje se gradi konstruktivno, komponentu po komponentu. Komponenta koja daje najveću korist pri dodavanju na trenutno rješenje, biće i uzeta kao komponenta koja proširuje trenutno rješenje. Odabir narednog rješenja u nizu donosi se na osnovu nekog (pohlepnog) kriterijuma.
8. *Dinamičko programiranje*: Ovaj tip algoritma koristi koncept korištenja već pronađenog rješenja (koje se čuva u memoriji) kako bi se izbjeglo ponavljanje izračunavanja istog dijela problema. Ova paradigma dijeli problem na manje podprobleme, koji se preklapaju, a potom ih rekurzivno rješava.

Većina ovih tipova algoritama predstavljaju generičke sheme (paradigme) koje se mogu primijeniti na rješavanje većeg opsega različitih problema. Tu pripadaju algoritmi brutalne sile, rekurzivni algoritmi, podijeli i zavladaaj, algoritam vraćanja unazad, pohlepni algoritmi, dinamičko programiranje isl. U glavi 3 biće detaljno obrađene neke od pomenutih paradigmi.

Dizajniranje efikasnog algoritma je obično dugotrajan proces koji nije direktan, te zahtijeva veliko domensko znanje o problemu te domišljatost onoga koji rješava vezano za način rješavanja samog problema. Međutim, postoje neka pravila kojima se vodimo kada konstruišemo algoritam, kao što su korištenje paradigmi za koje znamo da su dovoljno efikasne u rješavanju određenog tipa problema.

U nastavku govorimo malo o istorijskoj pozadini nastanka riječi *algoritam*. Riječ “Algoritam” je iskrivljena transliteracija imena “al-Khwarizmi”. Al-Khwarizmi bio je persijski matematičar iz IX veka, rođen na prostorima današnjeg Uzbekistana, studirao i radio u Bagdadu za vrijeme Abasidskog kalifata; oko 820. godine nove ere imenovan je za šefa biblioteke “Kuće mudrosti” u Bagdadu. Napisao je nekoliko uticajnih knjiga, uključujući i jednu sa naslovom “O kalkulaciji sa hinduistanskim brojevima”, koja opisuje kako se izvodi aritmetika koristeći arapske brojeve (aka arapsko-hindu, a kraće hinduistički brojevi). Originalni rukopis je izgubljen, ali latinski prevod iz 1100. godine uveo je ovaj brojevni sistem u Evropu te se smatra odgovornim za današnje korišćenje arapskih brojeva. Stara francuska riječ *algorisme* značila je “arapski numerički sistem”, a tek kasnije je postao opšti recept za rješavanje računskih problema.

1.3 Teorijski RAM model

Pri razvoju algoritama, postavljaju se nekoliko ključnih pitanja:

1. Da li je naš algoritam efikasan?
2. Da li je jedan algoritam brži od drugog?
3. Koliko je najgore očekivano vrijeme izvršavanja algoritma za ulaze velikih dimenzija?

Odgovor na ova pitanja nije trivijalan, i zahtjeva uvođenje stroge matematičke notacije i modela izračunljivosti.

Najprije uvedimo pojam *mašine sa slučajnim pristupom* (eng. *random access machine*), teorijskog modela računara. Mašina sa slučajnim pristupom može da izvršava instrukcije i manipuliše podacima. Ova mašina pretpostavlja da je memorija podijeljena na diskretne ćelije, od kojih se svakoj može pristupiti i direktno modifikovati njen sadržaj, te da mašina izvršava instrukcije na sekvencijalni način (ne postoji paralelizam).

Mašina sa slučajnim pristupom predstavlja varijaciju RAM modela koji uključuje randomizaciju u svojim operacijama. Gledajući fizički RAM model, redoslijed instrukcija je fiksna, ali u RAM-u se neke operacije mogu nasumično rasporediti. Npr. RAM može imati “slučajnu” instrukciju koja nasumično odabira ćeliju u memoriji i izvodi odgovarajuću operaciju u odabranoj ćeliji.

Ova mašina predstavlja model (apstrakciju) koja se koristi u analizi algoritama i njihove složenosti jer omogućava *precizno* izračunavanje broja instrukcija potrebnih za obavljanje određenog zadatka. Ovaj model pretpostavlja da su vremena pristupa memoriji konstantna, što nije tačno na realnim računarima. U praksi, pristup memoriji može uzimati različito vrijeme ovisno o faktorima kao što su keširanje, pristup disku i kašnjenje mreže.

Krenimo sa formalizovanjem teorijskog modela koji se standardno koristi za analizu algoritma. Posmatrajući viši nivo, karakteristike modela su sljedeće:

- Glavna memorija je niz $M = (M[0], \dots, M[S-1])$ w -bitnih riječi, koje se takođe mogu prikazati kao brojevi u intervalu $\{0, \dots, 2^w - 1\}$.
- Niz $R = (R[0], \dots, R[r-1])$, gdje je r broj registara, koji su takođe w -bitne riječi. Registri se koriste za privremeno pohranjivanje podataka tokom izvođenja programa, te za pohranjivanje adresa za dohvaćanje podataka iz RAM-a.
- Algoritmima je dozvoljen samo direktan pristup registrima. Preostalim memorijskim lokacijama može se pristupiti samo indirektnim adresiranjem – intepretirajući riječi (u registru) kao pokazivač na drugu memorijsku lokaciju. Dakle, zahtjevamo $S \leq 2^w$.
- Program je konačan niz instrukcija.
- Skup instrukcija se sastoji od osnovnih aritmetičkih i bitskih operacija nad riječima (u registrima), kondicionalima (**if-else**), **goto**, kopiranja riječi između registara i glavne memorije, zaustavljanja.
- Dopuštamo *malloc* operaciju, tj. dodavanje dodatne memorijske ćelije, povećavajući S eksplicitno (i w ako je potrebno da osiguramo $S \leq 2^w$).
- Početna konfiguracija memorije je sljedeća: (i) Ulaz veličine n smješten je u glavne memorijske lokacije $(M[0], \dots, M[n-1])$; (ii) Veličina riječi postavljena je na $w = \lfloor \log_2 \max\{n+1, k+1\} \rfloor$, gdje je k gornja granica za brojeve koji se pojavljuju ili u ulazu ili kao konstanta u programu; (iii) Registar 0 se inicijalizira sa w , a registar 1 sa n .

Formalno, dopuštene se sljedeće instrukcije za indekse $i, j, k, m \in \mathbb{N}$:

- $R[i] \leftarrow m$
- $R[i] \leftarrow R[j] + R[k]$
- $R[i] \leftarrow R[j] - R[k]$
- $R[i] \leftarrow R[j] \cdot R[k]$
- $R[i] \leftarrow R[j] \wedge R[k]$ (bitwise AND)
- $R[i] \leftarrow \neg R[j]$ (bitwise negation)
- $R[i] \leftarrow \lfloor R[j] / R[k] \rfloor$
- $R[i] \leftarrow M[R[j]]$
- $M[R[j]] \leftarrow R[i]$
- IF $R[i] = 0$, GOTO ℓ .
- HALT
- MALLOC

Slika 1.1: Pregled instrukcija u RAM modelu.

Definicija 1.2 *w-RAM program je bilo koji konačan niz instrukcija $P = (P_1, P_2, \dots, P_q)$ gore navedenog tipa. Broj r registara je implicitno definiran kao najveći indeks direktne memorijske adrese (među indeksima i, j, k u skup instrukcija sa Slike 1.1) koji se javljaju u programu.*

1.3.1 Formalizacija računanja na RAM-u

Pojam konfiguracije ima za cilj da obuhvati cjelokupno stanje računanja u određenom trenutku – sve što je neophodno za određivanje budućeg ponašanja.

Definicija 1.3 *Konfiguracija w-RAM programa P je torka $K = (l, S, w, R, M)$, gdje je $l \in \{1, \dots, q+1\}$ programski brojač, $S \in \mathbb{N}$ je korišteni prostor, $w \in \mathbb{N}$ je veličina riječi, $R = (R[0], \dots, R[r-1]) \in \{0, \dots, 2^w - 1\}^r$ je niz registara, i $M = (M[0], \dots, M[|S| - 1]) \in \{0, \dots, 2^w - 1\}^{|S|}$ je memorijski niz.*

Definicija 1.4 Za konfiguraciju $K = (l, S, w, R, M)$ w -RAM programa (P_1, \dots, P_q) definišemo sljedeću konfiguraciju $K' = (l', S', w', R', M')$, u zapisu $K \Rightarrow_P K'$, na sljedeći način:

- Ako je $P_l = \text{"IF } R[i] = 0, \text{ GOTO } m\text{"}$ za neke $i < r$ i $m \in \{1, \dots, q\}$, onda $K' = (l', S, w, R, M)$ gdje je $l' = m$ ako je $R[i] = 0$, a $l' = l + 1$ inače.
- Ako je $P_l = \text{"}R[i] \leftarrow m\text{"}$ za $i < r$, onda $K' = (l + 1, S, w, R', M)$ gdje $R'[i] = m \bmod 2^w$ i $R'[j] = R[j]$ za sve $j \neq i$.
- Ako je $P_l = \text{"}R[i] \leftarrow R[j] + R[k]\text{"}$ za neki $i, j, k < r$, onda $K' = (l + 1, S, w, R', M)$ gdje je $R'[i] = (R[j] + R[k]) \bmod 2^w$ i $R'[j] = R[j]$ za sve $j \neq i$.
- Ako je $P_l = \text{"}M[R[j]] \leftarrow R[i]\text{"}$ za $i, j < r$ i $R[j] < S$, onda $K' = (l + 1, S, w, R, M')$ gdje je $M'[R[j]] = R[i]$ i $M'[k] = M[k]$ za sve $k \neq R[j]$.
- Ako je $P_l = \text{"}MALLOC\text{"}$, onda je $K' = (l + 1, S + 1, w', R, M')$, gdje je $w' = \max\{w, \lceil \log_2(S + 1) \rceil\}$, $M'[S] = 0$ i $M'[i] = M[i]$ za $i = 0, \dots, S - 1$.
- Ako je $P_l = \text{"}HALT \text{ ili } l = q + 1\text{"}$, onda je $K' = (q + 1, S, w, R, M)$.

Napomenimo da smo prethodno naveli samo neke od instrukcija, dok se ostale definišu slično.

Definišimo sada formalno pojam računarskog problema shodno prethodno definisanom pojmu RAM modela.

Definicija 1.5 RAM program $P = (P_1, \dots, P_q)$ sa $r \geq 2$ registara rješava problem $f: \mathbb{N}^* \rightarrow 2^{\mathbb{N}^*}$ ako za svaki ulaz $x = (x_1, \dots, x_n) \in \mathbb{N}^*$ i $k \geq \max\{x_1, \dots, x_n, m\}$ gdje je m najveća konstanta koja se pojavljuje u P , postoji niz konfiguracija K_0, \dots, K_t tako da:

- $K_0 = (1, n, w, R, M)$ gdje je $w = \lceil \log_2(\max\{n, k + 1\}) \rceil$, $R[0] = n$, $R[1] = k$, $R[2] = R[3] = \dots = R[r - 1] = 0$, $M[0] = x_1, M[1] = x_2, \dots, M[n - 1] = x_n$,
- $K_{i-1} \Rightarrow_P K_i$ za sve $i = 1, \dots, t$,

- $K_t = (q + 1, S, w, R, M)$ gdje je $(M[1], M[2], \dots, M[R[0] - 1]) \in f(x)$.

Komentar. Ključne tačke u vezi teorijskog RAM modela su:

- *Ujednačenost:* zahtijevamo da postoji konačan program koji bi trebao raditi za proizvoljne ulaze neograničene veličine (kad $n, k \rightarrow \infty$).
- Računanje se odvija nizom “baznih” operacija.
- Ne namećemo a priori ograničenje za vrijeme (broj koraka) ili prostor (memorija). Ove resurse želimo minimizirati, ali i dalje teorijski RAM model smatra algoritmom čak i ako on koristi ogromne količine vremena i prostora, što se, vidjećemo, razlikuje od praktičnog aspekta algoritama.

Pri računanju broja instrukcija izvršavanja programa P , u obzir uzimamo sljedeće operacije: aritmetičke i bitske operacije $(+, -, *, /, \&, |, \dots)$, logičke $(\wedge, \vee, \Rightarrow, \Leftrightarrow)$, operacija dodjele, te ulazno/izlazne operacije. Ove operacije nazivamo *jediničnim instrukcijama*. Pretpostavka je da se sve jedinične instrukcije izvršavaju u jediničnom vremenu. Vremenska složenost programa se upravo mjeri na osnovu ukupnog broja jediničnih instrukcija, o čemu ćemo više reći u narednoj sekciji.

1.4 Pojam kompleksnosti

Napomenimo da je jediničnim modelom kompleksnosti definisan (teorijski) RAM model gdje se svaka operacija izvršava u jediničnom vremenu. Bez obzira što operacije sabiranja i množenja nemaju isto vrijeme izvođenja na stvarnom računaru, to ne umanjuje značaj teorijskog RAM modela, što ćemo prikazati u nastavku.

Definišimo sada vremensku kompleksnost izvršavanja programa P na RAM modelu.

Definicija 1.6 *Vrijeme rada P za ulaz $x \in \mathbb{N}^*$ je broj t koraka prije nego P dostigne zaustavnu konfiguraciju (operacija “HALT”) za x . Dakle, najmanji t za koji postoji niz $K_0 \Rightarrow_P K_1 \Rightarrow_P \dots \Rightarrow_P K_t$ takav da je K_0 početna konfiguracija od P za ulaz x , a K_t je konfiguracija zaustavljanja (gdje se programski brojač ažurira na $q + 1$, a q označava broj instrukcija u programu P). Pogledajmo sljedeći primjer.*

```

algoritam max(a, n)
    maximum = a[0]
    for i in 1 to n-1 do
        if a[i] > maximum then
            maximum = a[i]
    return maximum

```

Izračunajmo tačan broj instrukcija programa konkretnog ulaza $a = [1\ 5\ 2\ 10\ 7]$ i $n = 5$. Imamo: prva linija sadrži jednu instrukciju (dodjela). Unutar petlje (koja se izvršava $n - 1$ puta, za koju se dodjeljuje nova vrijednost iteratoru i , te se potom provjerava da li je ona veća od n), broj instrukcija koji je uvjetovan prolazom elementima niza a je 5, 3, 5 i 3, što je ukupno $1 + 5 + 3 + 5 + 3 = 17$ izvršenih instrukcija. U kombinaciji sa instrukcijama za varijable i u *for* petlji, ovaj program izvršava ukupno $17 + 4 \cdot 2 + 1 = 26$ instrukcija.

Vremenska složenost izvršavanja programa se mjeri u odnosu na ukupan broj jediničnih instrukcija. Tačan broj instrukcija za svaki ulaz u kompleksnim programima je teško precizno izračunati. Zbog toga uvodimo pojam *najgoreg vremena izvršavanja* programa (eng. *worst-case running time*) koji nam pruža lakši način računanja broja instrukcija programa, ali opet dovoljno reprezentativan da ukaže na to koliko je algoritam efikasan.

Definicija 1.7 *Najgore vrijeme izvršavanja programa P je funkcija $T: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ gdje je $T(n, k)$ maksimalno vrijeme rada programa P u odnosu na sve moguće ulaze $x \in \{0, \dots, k\}^n$.*

Primjer 1.1 *Pogledajmo sljedeći program*

```

algoritam sort(a, n)
    for i in 1 to n-1 do
        for j in 0 to i-1 do
            if a[i] > a[j] then
                swap (a[i], a[j])

```

Najgori slučaj za ulazne podatke je kada se instrukcije pod *if*-petljom stalno izvršavaju; to je slučaj kada je na ulazu niz a kojem su elementi poredani u obrnutom poretku u odnosu na traženi poredak. Prema tome *swap*

operacija se tada izvršava u svakoj iteraciji, i ona sadrži 3 instrukcije (dodjela). Unutrašnja petlja se izvršava i puta za svaki $i = 1, \dots, n - 1$. Tada imamo da je

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 2 \cdot 3 = 6 \sum_{i=1}^{n-1} i = 3 \cdot (n-1) \cdot n$$

najveći (najgori) broj instrukcija koji algoritam izvršava za svaki ulaz dužine n .

Za najgore vrijeme izvršavanja uzimamo najgori slučaj ulaznih podataka određene dimenzije za koji će se izvršiti maksimalan broj instrukcija. Motivacija za računanje najgoreg vremena izvršavanja programa (algoritama) je sljedeća. Najgori slučaj izvršavanja algoritma je obično lakši za detektovanje i izračunavanje nego je to slučaj sa tačnim brojem koraka (instrukcija) za konkretnu instancu, koji ne govori mnogo o broju instrukcija programa za ulaze drugačijih distribucija. Prepoznavanje najgoreg slučaja za ulazne podatke određene dimenzije za koje program radi najduže nam omogućuje da utvrdimo gornju granicu za broj instrukcija svakog mogućeg izvršenja programa. Ovakvo vrijeme izvršavanja ne zavisi od konkretne (distribucije) ulazne instance, čime se dobija bolji osjećaj o efikasnosti programa u odnosu na veličinu ulaza.

Bez obzira na ovakva pojednostavljenja, i dalje je egzaktno računanje najgoreg vremena izvršavanja težak posao u većini kompleksnih programa. Stoga, u nastavku govorimo o pojmu *asimptotskog* vremena izvršavanja programa. Gotovo uvijek se broj koraka izvršavanja u algoritmu povećava shodno povećavanju dimenzije ulaza. Prema tome, ima smisla posmatrati ponašanje algoritma na ulazima velike dimenzije jer se često programi čiji su ulazi malih dimenzija ionako efikasno izvršavaju (imaju kratko vrijeme izvršavanja) nezavisno od algoritamske tehnike rješavanja. U sljedećoj podsekciji, formalno uvedimo pojam O -notacije.

1.4.1 O -notacija

O -notacija dodatno pojednostavljuje ocjenjivanje broja izvršenih instrukcija programa, fokusirajući se na ponašanje algoritma na ulazima velikih dimenzija. Pretpostavimo da imamo sljedeću situaciju o ocjeni najgoreg scenarija izvršavanja dva algoritma (na istom posmatranom problemu):

- $T_1(n) = 2n^{\frac{3}{2}} + 3n + 10$
- $T_2(n) = 4n \cdot \log n + \lceil \sqrt{n} \rceil + 22$

Upoređivanje ovakvih funkcija zahtjeva analitičku analizu obje funkcije, pri čemu se može utvrditi da je prvi algoritam (koji se izvršava u T_1 vremenu u odnosu na ulaz veličine n) brži – zahtjeva izvođenje manjeg broja instrukcija – za manje n , dok je za veće n očigledno u prednosti algoritam sa vremenom izvršavanja T_2 ($n \log n$ dosta sporije raste od funkcije $n^{\frac{3}{2}}$).

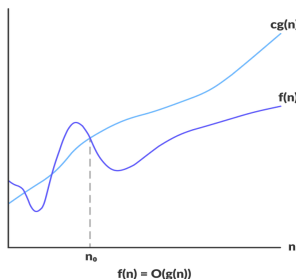
Dakle, cilj je je pojednostaviti računanje izvršenog broja instrukcija, koje se postiže uvođenjem sljedećih konvencija u svrhu ocjenjivanja ponašanja algoritma na instancama velikih dimenzija.

1. Konstante uz svaki od termova se ignorišu.
2. Dominantni termovi se uzimaju, dok se ostali (nedominirajući) ignorišu. Dominantan je onaj term koji najbrže raste sa porastom dimenzije ulaza. Dakle, procjenjuje se asimptotsko ponašanje broja instrukcija u zavisnosti od veličine ulaza programa.

Za $T_1(n)$ je dominantni term funkcija $n^{\frac{3}{2}}$, dok je to za $T_2(n)$ funkcija $n \cdot \log n$.

Nadalje pretpostavimo da radimo sa pozitivnim funkcijama, sa pozitivnim vrijednostima argumenata.

Definicija 1.8 *Neka su date dvije funkcije f i g . Kažemo da $f = O(g)$ akko postoji konstanta $c > 0$ tako da $f(n) \leq c \cdot g(n)$ počevši od nekog dovoljno velikog N . tj. za $n \geq N$. Kažemo da je g (gornja) asimptotska granica za f .*



Slika 1.2: O -notacija: vizuelizacija.

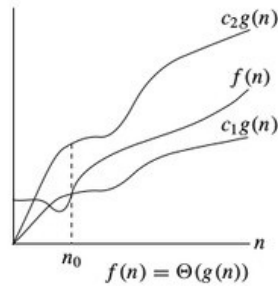
Primjer 1.2 *Jasno je da $T_1 = O(n^2)$, ali i $T_1 = O(n^{\frac{3}{2}})$. Takođe, $T_2 = O(n \log n)$.*

U osnovi, dominirajući term nam govori o asimptotskoj granici čitavog izraza (dobijen računanjem (najgoreg mogućeg) broja instrukcija izvršenog u programu), jer kad je n dovoljno veliko, vrijednost izraza dominantno zavisi od dominirajućeg terma, dok se ostali termi doprinose u ne tako značajnoj mjeri da bi se promijenio zaključak o asimptotskom ponašanju porasta broja instrukcija programa sa porastom dimenzije ulaza.

O -notacija, bez obzira na praktičnost i jednostavnost, dovodi i do određenih dvosmislenosti. Npr. $n^2 + n = O(n^2)$, ali vrijedi i $n^2 + n = O(n^3)$. U svakom slučaju, korisniji zaključak dobijamo uzimajući prvi od ova dva. Ovo nas dovodi do definisanja Θ -notacije (eng. Big-Theta).

Definicija 1.9 *Neka su date dvije funkcije f i g . Ako $f = \Theta(g)$, onda postoje konstante $c_1, c_2 > 0$ tako da $c_1g(n) \leq f(n) \leq c_2g(n)$, počevši od nekog dovoljno velikog n .*

U prevodu, funkcije f i g imaju isto asimptotsko ponašanje (dakle, za velike vrijednosti ulaza n), vidjeti Sliku 1.3.



Slika 1.3: Θ -notacija: vizuelizacija.

Primjer 1.3 *Imamo $T_1 = \Theta(n^{\frac{3}{2}})$, dok je $T_2 = \Theta(n \log n)$.*

Postoji još nekoliko notacija za kompleksnost algoritama, a to su Ω i o -notacija.

Definicija 1.10 *Neka su date dvije funkcije f i g . Kažemo da $f = \Omega(g)$ akko postoji konstanta $c > 0$ tako da je $f(n) \geq cg(n)$, počevši od nekog dovoljno velikog n .*

Ω -notacija govori o funkcijama za koje je g asimptotska donja granica, budući da ograničava rast vremena rada odozdo za dovoljno velike veličine ulaza.

Primjer 1.4 Vrijedi $n^3 = \Omega(n^2 + n)$ i $n \log n = \Omega(n)$.

Definicija 1.11 Neka su date dvije funkcije f i g . Kažemo da $f = o(g)$ akko za sve $c > 0$ postoji $N > 0$ takav da je $0 \leq f(n) < cg(n)$ za sve $n \geq N$.

Napomena. Vrijednost k ne smije zavisiti od n , ali može da zavisi od c . Neformalno, ako $f(n) = o(g(n))$, to znači da f postaje beznačajno mala u odnosu na g kako se n približava beskonačnosti.

Primjer 1.5 Vrijedi $n = o(n^2)$, kao i $n = o(n^3)$, ali ne vrijedi $n = o(n)$.

1.4.2 Neka svojstva O -notacije

Teorema 1.1 Neka su f, g i h pozitivne funkcije, pozitivnih vrijednosti argumenata. Vrijede sljedeće tvrdnje:

- $f = O(c \cdot f)$ za svako $c > 0$ (konstantni faktori su irelevantni);
- Ako $f = O(g+h)$ i $h = O(g)$, onda je i $f = O(g)$ (pravilo dominantnog terma);
- $O(f + g) = O(f) + O(g)$ (pravilo zbira);
- $O(f \cdot g) = O(f) \cdot O(g)$ (pravilo proizvoda);
- $f = O(g)$ i $g = O(h)$ onda vrijedi i $f = O(h)$ (pravilo tranzitivnosti).

Dokaz Dokazi tvrdnji se direktno izvode na osnovu definicije O -notacije.

1.4.3 Uputstva za procjenu kompleksnosti algoritama

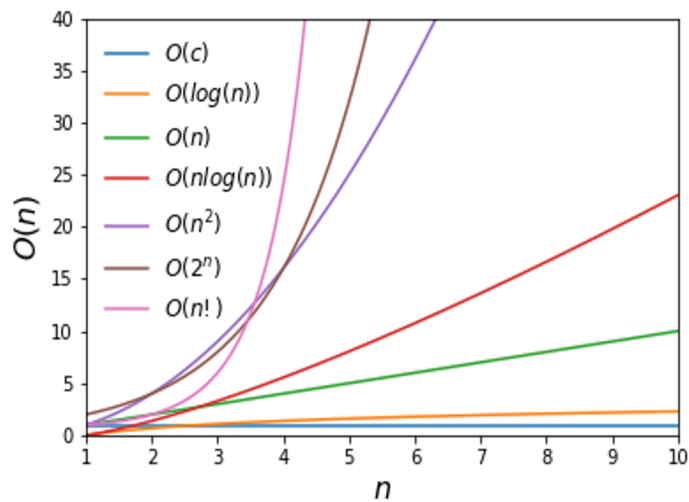
Prethodna sekcija nam daje na uvid neke od osnovnih notacija za ocjenjivanje kompleksnosti algoritma, tj. izvršenja broja instrukcija programa koja se obično razmatra u zavisnosti od veličine ulaza.

Najčešće funkcije koje se pominju u analizi kompleksnosti izvršavanja algoritama su nabrojane u Tabeli ??.

Tabela 1.1: Rast funkcija.

Funkcija	Kompleksnost
1	konstantna
$\log n$	logaritamska
n	linearna
$n \log n$	super-linearna
n^2	kvadratna
n^3	kubna
2^n	eksponencijalna
$n!$	faktorijel

Rast svake od ovih funkcija sa porastom ulaza (n) je prikazan na Slici 1.4



Slika 1.4: Rast funkcija.

U praksi, neželjena kompleksnost algoritma je eksponencijalna, dok su logaritamska, linearna i log-linearna kompleksnost poželjne i takvi algoritmi su obično veoma efikasni u rješavanju problema, tj. instanci velikih dimenzija. U zavisnosti od tipa problema, kvadratna kompleksnost može ali i ne mora biti garant za rješavanje realnih instanci većih dimenzija razmatranog problema.

U nastavku izlažemo neka uputstva koja olakšavaju računanje kompleksnosti algoritma. Za (pod)program P , označimo njegovu kompleksnost izvođenja

sa funkcijom $T(P)$ koja (direktno ili indirektno) zavisi od dimenzije ulaza (n).

Tabela 1.2: Uputstva za računanje kompleksnosti algoritma.

Konstrukcija	Računanje kompleksnosti
Jedinične instrukcije	1
Serijska naredba S : $N_1; N_2$;	$T(S) = T(N_1) + T(N_2)$
Naredba grananja S : IF C then N_1 ELSE N_2	$T(S) = T(C) + \max\{T(N_1), T(N_2)\}$
Naredba petlje S : 1) WHILE U DO N_1 2) FOR $i = j$ to p DO N_1	n – maksimalan broj iteracija $T(S) = n \cdot (T(U) + T(N_1))$ $T(S) = n \cdot T(N_1)$

Odgovorimo sada na pitanja da li je jedan algoritam efikasniji od drugoga? Upravo, odgovor nam daju notacije za kompleksnost koje smo naučili u ovoj glavi, i to dvije najznačajnije: O i Θ –notacije. Kažemo da je jedan algoritam X *asimptotski efikasniji* od drugog algoritma Y akko sa porastom veličine ulaza u oba algoritma, vrijeme rada algoritma X postaje kraće od vremena rada Y . To znači da X pripada nižoj O –klasi složenosti od Y . Prema tome, za velike ulaze, algoritam X je uvijek bolji izbor od algoritma Y u smislu efikasnosti (vremena izvođenja). Međutim, za male ulaze moguće je da je Y i dalje brži od X . Stoga se ne može reći da je X uvijek bolji izbor za sve ulazne vrijednosti, već da će X biti bolji izbor za sve ulaze osim eventualno malih ulaza.

U narednom primjeru dajemo jedan zadatak, rješavamo ga na dva načina, te izvodimo kompleksnost oba programa.

Primjer 1.6 *Problem nalaska (neprekidnog) podniza maksimalne sume. Od svih neprekidnih podnizova datog niza, naći onaj čija je suma elemenata najveća moguća.*

Rješenje. Neka je dat ulaz $a = [1, 4, -2, 2, 10, -7, 1]$. Jedan neprekidan podniz ovog niza je $[-2, 2, 10]$ (početne pozicije 2, te završne pozicije 4), ali $[-2, -7, 10]$ nije neprekidan podniz.

Koristimo se naivnim pristupom rješavanja problema. Definišimo

$$S(i, j) := \max\{a[i], \dots, a[j]\},$$

gdje je $S(i, i) = a[i]$. Funkcija $S(i, j)$ računa sumu elemenata neprekidnog podniza niza a , sa početnom pozicijom i , te završnom pozicijom j . Za

izračunavanje funkcije $S(i, j)$, potrebno je $j - i + 1$ instrukcija, što definitivno pripada klasi $O(n)$. Da bismo riješili problem na ovaj način, treba da pozovemo funkciju $S(i, j)$ za svaki par (i, j) , $i, j \in \{0, \dots, n-1\}$, $i \leq j$, a to je ukupno $\binom{n}{2} + n$ puta, što odgovara klasi kompleksnosti $O(n^2)$. Dakle, kompleksnost ovakvog algoritma je $O(n) \cdot O(n^2) = O(n^3)$ što je kubna kompleksnost.

Pokušajmo konstruisati “pametniji” algoritam. Iskoristimo činjenicu da postoji jasna rekurzivna zavisnost između vrijednosti dva susjedna $S(i, j)$ i $S(i, j+1)$, tj. da je $S(i, j+1) = a[j+1] + S(i, j)$, za $j \in \{i, \dots, n-2\}$. Definišimo sada:

$$S(i) := \max\{S(i, i), S(i, i+1), \dots, S(i, n-1)\}.$$

Izvršavanje funkcije $S(i)$ se odvija u linearnoj kompleksnosti, tj. $O(n)$. Kako se ona poziva n puta (za svako $i \in \{0, \dots, n-1\}$ po jednom), ukupna kompleksnost ovog pristupa je $n \cdot O(n) = O(n^2)$, što je znatno efikasnije od prethodnog, naivnog, pristupa.

1.5 Prostorna kompleksnost

Termin prostorna kompleksnost se na mnogim mjestima pogrešno koristi asocirajući na pomoćni prostor. *Pomoćni prostor* je dodatni ili privremeni prostor koji koristi algoritam.

Prostorna složenost algoritma je ukupan prostor koji algoritam zauzima u odnosu na veličinu ulaza. Prostorna složenost uključuje i pomoćni prostor i prostor koji koristi ulaz.

Prostorna složenost se može posmatrati kao pandan koncepta vremenske složenosti. Za kreiranje niza veličine n , zahtijeva se dodatan prostor od $O(n)$. Za kreiranje dvodimenzionalnog niza veličine $n \times n$, zahtijeva se $O(n^2)$ prostora. Napomenimo da prostorna kompleksnost zavisi i od programskog jezika, kompajlera, pa čak i mašine koja pokreće algoritam.

Efikasnost algoritma uglavnom je definisana u odnosu na dva faktora, korišteni prostor i vrijeme. “Dobar” algoritam je onaj koji koristi manje vremena i prostora; međutim, to nije moguće stalno postići. Zbog toga postoji kompromis između vremena i prostora. Ako želimo skratiti vrijeme izvršavanja algoritma, korišteni prostor se obično povećava. Slično tome, ako želimo smanjiti prostor, vrijeme izvršenja algoritma se obično povećava.

Primjer 1.7 *Posmatrajmo sljedeći pseudokod, te odredimo prostornu kompleksnost datog programa.*

```
1: procedure PAIRSUM( $x, y$ )  
2:   return  $x + y$   
3: end procedure
```

```
1: procedure ADDSEQUENCE( $n$ )  
2:    $sum \leftarrow 0$   
3:   for  $i = 0$  to  $n - 1$  do  
4:      $sum \leftarrow sum + \text{pairSum}(i, i + 1)$   
5:   end for  
6:   return  $sum$   
7: end procedure
```

Prostorna kompleksnost procedure ADDSEQUENCE je jednaka $O(1)$, bez obzira što se funkcija PAIRSUM poziva $O(n)$ puta. Zašto?

Napomena. U standardnom programiranju, podrazumijeva se korištenje 256MB prostora za određeni program. Dakle, kreiranje niza veličine veće od 10^8 nije dozvoljeno, jer je na raspolaganju data gornja granica za količinu memorije na raspolaganju. Takođe, kreiranje niza veličine veće od 10^6 u tijelu funkcije nije moguće, jer je maksimalni prostor dodijeljen funkciji (u lokalnom steku) 4MB. Dakle, da bismo koristili niz veće veličine, potrebno je kreirati globalni niz.

1.6 Pseudokod

Pseudokod je jednostavan prikaz implementacije algoritma u obliku anotacija i informativnog teksta napisanog na običnom jeziku. On nema sintaksu kao bilo koji programski jezik i stoga ga računar ne može kompajlirati ili interpretirati. Pseudokod, kao što sam naziv sugerise, je reprezentacija kôda koja je prilagođena da ga razumiju i osobe bez velikog znanja o programiranju i algoritmima. Za razliku od pseudokoda, algoritam je organizirani niz instrukcija koji rješava određen problem pri implementiranju u nekom programskom

jeziku, te se kompajliranjem ili interpretiranjem prevodi u mašinski kôd kojeg mašina može da izvrši.

Prednosti korištenja pseudokoda su sljedeće.

- Poboljšava čitljivost i razumijevanje bilo kojeg pristupa kojim se rješava problem.
- Djeluje kao spona između programa i dijagrama toka.
- Služi kao gruba dokumentacija, da bi se program mogao lakše razumjeti. U industriji je vođenje dokumentacije od suštinskog značaja i tu se potreba za pseudokodom pokazuje krucijalnim.

Glavni cilj pseudokoda je da nedvosmisleno objasni suštinu svakog dijela programa, čime se olakšava faza izgradnje kôda te smanjuje mogućnost pravljenja grešaka u implementaciji algoritma.

Pisanje pseudokoda počinje razumijevanjem problema koji se rešava. Nakon jasnog definisanja problema, slijedi razmatranje koraka koje treba preduzeti kako bi se došlo do rešenja. Te korake zatim zapisujemo uz pomoć pseudokoda.

Nekoliko saveta u vezi prakse pisanja “dobrog” pseudokoda su:

- Koristiti jasne i precizne nazive za promenljive i funkcije.
- Koristiti opise koji jasno ukazuju na to šta se dešava u svakom koraku algoritma.
- Razmisliti o slučajevima u kojima algoritam može da se zaustavi i načinima na koje se to može desiti.
- Koristiti odgovarajuće sintakse za kontrolne strukture kao što su petlje i uslovi: `if-else`, `for`, `while` petlje. Koristiti mehanizam uvlačenja naredbi koje se izvršavaju u okviru istog bloka, jer pomaže lakšem shvatanju kontrolnog mehanizma i izvršavanja odluka. Ovakva praksa u velikoj mjeri poboljšava čitljivost kôda. Pogledajmo u nastavku primjer jednog jednostavnog pseudokoda.

```
tip = unos sa tastature
if tip == "1"
    ispiši odgovor "Unesen je broj 1"
```

```

if tip == "2"
    ispiši odgovor "Unesen je broj 2"

```

- Dakle, pseudokod nije potrebno pisati na potpuno programski način. Njegova svrha je jednostavnost i razumijevanje čak i za osobe kojima programiranje nije struka, stoga ne bi trebalo sadržati previše tehničkih detalja.

Primjer 1.8 *Napisati pseudokod za nalazak najvećeg elementa u nizu.*

Inicijalizuj najveći broj na prvi element niza.

Proći sve preostale elemente niza:

*Ako je trenutni element veći od trenutno najvećeg broja,
postavi trenutni element kao novi najveći broj.*

Kraj petlje.

Ispiši najveći broj.

U nastavku navodimo (koncizniji) pseudokod, koji rješava problema nalaska podniza maksimalne sume na efikasniji način. Ovdje će biti korišten čest format koji je više tehnički, a i više upotrebljavan u literaturi.

Algoritam 1 Funkcija $S_i(i)$

```

1: procedure S( $i$ , niz)
2:    $s_{maxi}, s_{next} \leftarrow niz[i]$ 
3:   for  $j = i + 1$  to  $n - 1$  do
4:      $s_{next} \leftarrow s_{next} + niz[j]$ 
5:     if  $s_{next} > s_{maxi}$  then
6:        $s_{maxi} \leftarrow s_{next}$ 
7:     end if
8:   end for
9:   return  $s_{maxi}$ 
10: end procedure

```

Algoritam 2 Funkcija S

```

1: procedure  $S(niz)$ 
2:    $s_{max} \leftarrow niz[0]$ 
3:   for  $i = 0$  to  $n - 1$  do
4:      $s_i \leftarrow S_i(i)$ 
5:     if  $s_{max} < s_i$  then
6:        $s_{max} \leftarrow s_i$ 
7:     end if
8:   end for
9:   return  $s_{max}$ 
10: end procedure

```

1.7 O valjanosti algoritma

Postoji nekoliko načina da se formalno provjeri valjanost algoritama; tri su osnovna načina:

- *Matematički dokaz.* Predstavlja jedan od najrigoroznijih načina za provjeru valjanosti algoritma. Ovdje se koriste matematičke tehnike za analizu logike algoritma i za utvrđivanje da algoritam radi ispravno za sve moguće ulaze. Jedana od metoda je *metoda invarijantne petlje*, kojom ćemo se pozabaviti u nastavku ove sekcije.
- *Formalna verifikacija.* Formalna verifikacija je proces upotrebe matematičkih tehnika i alata za provjeru da li algoritam ispunjava određene specifikacije. Ovo uključuje modelovanje algoritma korištenjem formalnog jezika, a zatim korištenje automatiziranih alata za provjeru da li model zadovoljava određena svojstva. U osnovi, ovi metodi su iz domena vještaške inteligencije specijalno namijenjeni za verifikaciju softvera.
- *Testiranje.* Iako testiranje nije formalna metoda već empirijska, ona predstavlja važan način za provjeru valjanosti algoritma. Pokretanjem algoritma sa raznim ulaznim vrijednostima i provjeravanjem izlaza u odnosu na očekivane rezultate, može se steći povjerenje o tome da li je algoritam ispravan.

Metoda *invarijantne petlje* (eng. *loop invariant*) predstavlja jednu od matematičkih tehnika koja se koristi u dokazivanju ispravnosti algoritama. In-

varijanta petlje je tvrdnja koja ostaje istinita tokom svake iteracije petlje programa, uključujući i prije i poslije izvršavanja naredbi u iteraciji.

Da bismo koristili metod invarijantne petlje u dokazivanju ispravnosti algoritma, najčešće se kombinuju sljedeći koraci.

- Definisanje invarijante petlje: potrebno je definisati tvrdnju koja ostaje istinita prije i poslije svake izvršene iteracije petlje.
- Dokazivanje inicijalne invarijante: treba pokazati da je invarijanta petlje istinita prije prve iteracije petlje.
- Dokazivanje održavanja invarijante: potrebno je dokazati da ako je invarijanta istinita prije određene iteracije petlje, onda je istinita i nakon izvršavanja te iteracije.
- Dokazivanje da se petlja završava: pokazati da se nakon konačnog broja iteracija, petlja prekida.
- Dokazati da algoritam rješava problem: potrebno dokazati da, ako je invarijanta istinita, nakon posljednje iteracije petlje, algoritam uistinu rješava problem.

Ova metoda je korisna u dokazivanju ispravnosti algoritama koji se sastoje od petlji, jer omogućava dokazivanje da algoritam radi ono što je i očekivano, tj. da ispravno rješava problem za sve moguće ulaze.

Primjer 1.9 *Demonstrirajmo pokazivanje ispravnosti programa za nalazak najvećeg zajedničkog djelioca (NZD) dva prirodna broja, datog Algoritmom 3, uz pomoć metoda invarijantne petlje.*

Algoritam 3 NZD dva broja.

```

1: procedure NZD( $n, m$ )
2:    $nzd \leftarrow \min(n, m)$ 
3:   while  $!(n \% nzd == 0 \ \& \ m \% nzd == 0)$  do
4:      $nzd \leftarrow nzd - 1$ 
5:   end while
6:   return  $nzd$ 
7: end procedure

```

Ovaj program je konačan, jer je broj iteracija u glavnoj while-petlji najviše nzd , dok je broj instrukcija u svakoj iteraciji također konačan (jednak $5+2=7$). Dalje, odredimo uslov invarijantne petlje. Neka je nzd^* konačan rezultat koji program vraća (koji je zasigurno zajednički djelilac, po uslovu prekida petlje). Tada invarijantnu petlju definišemo sa

$$nzd^* \leq nzd.$$

Prije ulaska u prvu iteraciju petlje imamo $nzd^* \leq nzd = d_0 = \min(n, m)$. Neka je prije ulaska u k -tu iteraciju petlje $d_{k-1} = nzd$. Tada je $nzd^* < d_{k-1}$ jer se ulaskom u k -tu iteraciju petlje uvjeravamo da d_{k-1} ne može biti NZD ulaznih brojeva (jer tada zadovoljava uslov while-petlje). Tada imamo $d_k = d_{k-1} - 1 \geq nzd^*$ nakon završetka k -te iteracije, čime je uslov invarijantne petlje i tada zadovoljen. Pretpostavimo da se u iteraciji k^* program prekida. To znači da je d_{k^*-1} zasigurno zajednički djelilac. Kako tražimo najveći takav, onda je i $d_{k^*-1} \leq nzd^*$. S obzirom da je $d_{k^*-1} \geq nzd^*$ zbog uslova invarijantne petlje, slijedi da je $d_{k^*-1} = nzd^*$, čime smo pokazali valjanost algoritma.

Pokazivanje valjanosti kompleksnih algoritama pomoću metoda invarijantne petlje je nerijetko nepraktično. Jedan od razloga je taj što definisanje samog uslova invarijantnosti nije trivijalan zadatak, kao i to da instrukcije koje se izvršavaju u svakoj iteraciji nisu trivijalne (kao što je to slučaj sa prethodnim programom), te mogu uključivati pozivanje složenijih pomoćnih funkcija. U takvim slučajevima, valjanost algoritma provjeravamo empirijski, izvršavajući ga nad instancama različitih veličina i distribucija te provjeravajući validnost vraćenih rezultata (za instance za koje znamo tačan rezultat, dok za one za koje ne znamo, provjeravamo ispunjivost uslova zadatka).

1.8 Euklidov algoritam: kompleksnost

Klasični algoritam za nalazak NZD je Euklidov algoritam, dat Algoritmom 4. On predstavlja jedan od najpoznatijih algoritama u polju aritmetike. Izvedimo njegovu vremensku kompleksnost.

Rješenje. Prvo izvedimo korake algoritma na konkretnoj instanci. Neka je $p = 400$, a $q = 24$. Tada imamo korake:

$$(400, 24) \rightarrow (24, 16) \rightarrow (16, 8) \rightarrow (8, 0),$$

Algoritam 4 NZD dva broja.

```

1: procedure NZD( $p, q$ ) # pretpostavka je da  $p \geq q$ 
2:    $x \leftarrow p$ 
3:    $y \leftarrow q$ 
4:   while  $y > 0$  do
5:      $x \leftarrow y$ 
6:      $y \leftarrow x \% y$ 
7:   end while
8:   return  $x$ 
9: end procedure

```

odakle slijedi da je $\text{NZD}(400, 24) = 8$. Za razliku od naivnog algoritma za nalazak NZD iz prethodnog poglavlja, koji bi za ovu instancu izveo 17 iteracija do nalaska rješenja, vidimo da za Euklidov algoritam treba svega 4 iteracije da nađe tačno rješenje.

Prvo, Euklidov algoritam je konačan, jer je broj iteracija ograničen sa $\min(p, q)$. Drugo, algoritam ne zavisi direktno od veličine ulaza (koji je konstantan, tj. veličine $2 = |\{p, q\}|$), već od veličine brojeva, ili dužine binarne reprezentacije (dekadnih) brojeva u ulazu. Ako je p broj u dekadnom zapisu, dužina njegovog binarnog zapisa jednaka je $\lfloor \log_2(p) \rfloor + 1$. Dalje, označimo sa n_1 broj bitova u zapisu broja p , a sa n_2 broj bitova u zapisu broja q . Tada je

$$n_1 \approx \log_2(p), n_2 \approx \log_2(q), n = n_1 + n_2 = \log_2(p \cdot q).$$

Vrijednost n je najveća kada su p i q susjedni brojevi, odakle je $n_1 \approx n_2 \approx n/2$, pa za kompleksnost algoritma $T(n)$ vrijedi da je proporcionalna vrijednosti manjeg od ta dva broja. Dakle, imamo $T(n) = \Omega(q) = \Omega(2^{n_2}) = \Omega(2^{n/2})$.

U svrhu pokazivanja kompleksnosti, iskoristimo sljedeću teoremu, koju i formalno pokazujemo.

Teorema 1.2 *Za svaka dva broja p i q vrijedi*

$$x \% y \leq x/2.$$

Dokaz Razlikujemo dva slučaja:

- $x/2 < y \Rightarrow x \% y = x - y \leq x/2$;

- $x/2 \geq y \Rightarrow x \% y < y \leq x/2$.

Procijenimo broj koraka algoritma. Algoritam izvodi sljedeći niz koraka:

$$(p_0, q_0) \rightarrow (p_1, q_1) \rightarrow \cdots \rightarrow (p_m, q_m) \rightarrow (p_{m+1}, q_{m+1} = 0),$$

gdje je $p_0 = p, q_0 = q$, nakon čega algoritam prekida sa radom. Procijenimo vrijednost broja koraka m .

Imamo: $p_1 = q_0, q_1 = p_0 \% q_0$. Dalje, primjenom prethodne teoreme imamo:

$$p_1 q_1 = q_0 p_0 \% q_0 \leq q_0 p_0 / 2.$$

Slično je

$$p_2 q_2 \leq q_0 p_0 / 2^2.$$

Takođe je

$$1 \leq p_m q_m \leq q_0 p_0 / 2^m,$$

odakle dobijamo $p q = q_0 \cdot p_0 \geq 2^m$, odakle nakon primjene logaritma dobijamo

$$m < \log(pq) = n,$$

odakle imamo $m = O(n)$.

U svakom koraku algoritma, najskuplji korak je računanje modula, za koji je neophodno dijeljenje dva binarna broja. Taj proces zahtjeva $O(n^2)$ instrukcija. Iz svega imamo da je kompleksnost algoritma:

$$O(n^2) \cdot O(n) = O(n^3) = O(\log \max\{p, q\}).$$

Primijetimo da je prostorna kompleksnost Euklidovog algoritma konstanta, tj. jednaka je $O(1)$.

Napomenimo da valjanost Euklidovog algoritma slijedi iz algebarskog svojstva da je skup djelilaca brojeva p i q , jednak skupu djelilaca brojeva q i $p \% q$, pa su stoga transformacije parova brojeva za koje se razmatra NZD u svakom koraku validne, dok se ne dođe od trivijanog slučaja, tj. dok drugi broj ne bude jednak nuli, čime se vraća vrijednost prvog broja kao krajnji rezultat.

Kako smo vidjeli, iako je Euklidov algoritam relativno jednostavnog kôda, njegovu kompleksnost nije bilo trivijalno odrediti. Napomenimo da Euklidov algoritam predstavlja jedan vid rekurzivnog algoritma. U osnovi, za

računanje kompleksnosti ovakvih algoritama, o kojima će biti više riječi u Glavi 3, koristi se poznata Master teorema. Ovu teoremu navodimo bez dokaza, uz napomenu da se ona navodi i formalno dokazuje na nekim od kurseva algoritmike na višim godinama studija. Mi ćemo je koristiti isključivo samo kao pomoćni alat u već pomenutoj Glavi 3, kod određivanja kompleksnosti algoritama problema koje budemo rješavali.

Teorema 1.3 (Master teorema) *Neka je data jednačina*

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

gdje je n veličina ulaza, broj a predstavlja broj podproblema, n/b veličinu svakog podproblema, $f(n)$ broj instrukcija koji je potreban pored rekursivnog poziva, uz $a \geq 1$ i $b > 1$, dok je $f(n) \geq 0$. Vrijedi sljedeće:

- *Ako je $f(n) = O(n^{\log_b(a)-\epsilon})$ za neki $\epsilon > 0$, onda $T(n) = \Theta(n^{\log_b(a)})$;*
- *Ako je $f(n) = \Theta(n^{\log_b(a)})$, onda je $T(n) = \Theta(n^{\log_b(a)} \log(n))$;*
- *Ako je $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ za neki $\epsilon > 0$, onda je $T(n) = \Theta(f(n))$.*

Na sljedećim primjerima pokazujemo kako funkcioniše primjena Master teoreme.

Primjer 1.10 *Posmatrajmo formulu:*

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

Kako je $a = 2, b = 2, f(n) = n$, te $f(n) = n = n^{\log_b(a)} = n^{\log_2(2)} = n$, iz Master teoreme, na osnovu slučaja 2, možemo zaključiti da je

$$T(n) = \Theta(n^{\log_b(a)} \log(n)) = \Theta(n \log n).$$

Primjer 1.11 *Neka je data formula $T(n) = T(\frac{n}{2}) + O(1)$ Imamo $a = 1, b = 2, f(n) = 1$, pa je*

$$f(n) = 1 = n^{\log_b(a)} = n^{\log_2(1)} = n^0 = 1.$$

Na osnovu Master teoreme, slučaja 2, imamo

$$T(n) = \Theta(n^{\log_b(a)} \log(n)) = \Theta(\log n).$$

Primjer 1.12 *Neka je data sljedeća rekurzija:*

$$T(n) = T\left(\frac{n}{2}\right) + cn, c > 0.$$

Imamo $a = 1, b = 2, f(n) = cn$, pa je

$$f(n) = cn = n^{\log_b(a)} = n^{\log_2(1)+\varepsilon} = n^\varepsilon \text{ za neki } \varepsilon \leq 1.$$

Na osnovu Master teoreme, slučaj 3, imamo $T(n) = \Theta(cn) = \Theta(n)$.

Zadaci

1. Izračunati vremensku kompleksnost sljedećeg programa:

```

1:  $a \leftarrow 0$ 
2: for  $i = 0$  to  $N$  do
3:   for  $j = N$  downto  $i$  do
4:      $a \leftarrow a + i + j$ 
5:   end for
6: end for

```

2. Izračunati vremensku kompleksnost sljedećeg programa

```

1:  $i, j, k \leftarrow 0$ 
2: for  $i = n/2$  to  $n$  do
3:   for  $j = 2; j \leq n; j = j * 2$  do
4:      $k \leftarrow k + n/2$ 
5:   end for
6: end for

```

3. Izračunati vremensku kompleksnost sljedećeg programa

```

1:  $a \leftarrow 0, i \leftarrow N$ 
2: while  $i > 0$  do
3:    $a \leftarrow a + i$ 
4:    $i \leftarrow i/2$ 
5: end while

```

4. Izračunati vremensku kompleksnost sljedećeg programa

```

1: for  $i = 0$  to  $n - 1$  do
2:    $i \leftarrow i * k$ 
3: end for

```

5. Izračunati vremensku kompleksnost sljedećeg programa

```

1:  $sum \leftarrow 0$ 
2: for  $i = 1$  to  $n$  do
3:   for  $j = 1$  to  $i \cdot i$  do
4:     if  $j \bmod i == 0$  then
5:       for  $k = 1$  to  $j$  do
6:          $sum \leftarrow sum + 1$ 
7:       end for
8:     end if
9:   end for
10: end for

```

6. Izračunati kompleksnost sljedeće rekurzije:

```

1: procedure BUBBLESORT( $arr[]$ ,  $size$ )
2:   for  $i = 0; i < size - 1; ++i$  do
3:     for  $j = 0; j < size - 1; ++j$  do
4:       if  $arr[j] > arr[j + 1]$  then
5:          $swap(arr[j], arr[j + 1])$ 
6:       end if
7:     end for
8:   end for
9: end procedure

```

7. Izračunati kompleksnost sljedeće rekurzije:

$$T(n) = \begin{cases} 3T(n-1), & \text{ako } n > 0, \\ 1, & \text{inače} \end{cases}$$

8. Izračunati kompleksnost sljedeće rekurzije:

$$T(n) = \begin{cases} 2T(n-1)-1, & \text{ako } n > 0, \\ 1, & \text{inače} \end{cases}$$

9. Izračunati prostornu kompleksnost programa datog sljedećim pseudokodom

```
1: procedure SUM(arr[], N)
2:   ans ← 0
3:   for i = 0 to N do
4:     ans ← ans + arr[i]
5:   end for
6:   ispiši ans
7: end procedure
```
















10. Izračunati prostornu kompleksnost programa datog sljedećim pseudokodom

```
1: procedure FAKTORIJEL(N)
2:   fact ← 1
3:   for i = 1 to N do
4:     fact ← fact * i
5:   end for
6:   return fact
7: end procedure
```

Glava 2

Uvod u programski jezik Pajton

Pajton je objektno-orjentisani programski jezik visokog nivoa, opšte namjene, pušten u upotrebu u februaru 1991. godine. Kreiran je od strane Guido van Rossum-a. Naziv Pajton potiče od britanske komičarske grupe *Monty Python*. Od svog nastanka, Pajton je imao nekoliko izdanja; danas je jedan od najpopularnijih programskih jezika na svijetu, pogledati Sliku 2¹.

Pos. Jan 2022	Pos. Jan 2021	Programming Language	Ratings	Chart Ratings	Variations
1	3	Python	13.58%		+1.86%
2	1	C	12.44%		-4.94%
3	2	Java	10.66%		-1.30%
4	4	C++	8.29%		+0.73%
5	5	C#	5.68%		+1.73%
6	6	Visual Basic	4.74%		+0.90%
7	7	JavaScript	2.09%		-0.11%
8	11	Assembly language	1.85%		+0.21%
9	12	SQL	1.80%		+0.19%
10	13	Swift	1.41%		-0.02%
11	8	PHP	1.40%		-0.60%
12	9	R	1.25%		-0.65%
13	14	Go	1.04%		-0.37%
14	19	Delphi/Object Pascal	0.99%		+0.20%
15	20	Classic Visual Basic	0.98%		+0.19%

Slika 2.1: Statistički pregled najpopularnijih programskih jezika (godina 2021–2022).

¹Podaci preuzeti sa <https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2022-new-update/>.

U nastavku navodimo nekoliko važnih informacija vezanih za istoriju razvoja programskog jezika Pajton.

Godine 1989. Guido van Rossum je samoinicijativno započeo rad na kreiranju novog programskog jezika kao poboljšanoj verziji ABC jezika, koji se uglavnom koristio za podučavanje osnovama programiranja. Van Rossum je izgradio Pajton uzimajući po njemu pozitivne strane ABC jezika kao bazu, a poboljšavajući ili iznova implementirajući lošije koncepte tog jezika.

Prva verzija Pajtona, verzija 0.9.0, je objavljena u februaru 1991. godine. Pajton 1.0 je objavljen u januaru 1994. godine. Uključio je nekoliko novih funkcionalnosti, preuzimajući uglavnom dobre koncepte funkcionalnih jezika, kao što su lambda, mapa, filter i funkcije redukcije. Pajton 2.0 je objavljen 2000. godine i uveo je komprehenziju liste, sakupljač otpadaka i mnoga druga poboljšanja. Pajton 3.0 je objavljen 2008. godine kao veliko ažuriranje koje je uvelo nekoliko izmjena koje su nekompatibilne sa prethodnim verzijama.

Glavna odlika verzije Pajton 3.0 bio je pročišćavanje jezika i uklanjanje redundantnih funkcija. Od pojavljivanja Pajton 3, razvojni pajtonov tim pušta glavne verzije jezika svakih 18-24 mjeseca. Svaka nova verzija uključuje nove funkcije i poboljšanja, čuvajući kompatibilnosti sa prethodnim verzijama. Danas se Pajton koristi u gotovo svemu, od razvoja veb aplikacija, do naučnog računarstva, analize podataka, vještačke inteligencije, mašinskog učenja itd. Prednost Pajtona je velika i aktivna zajednicu programera koja svakodnevno doprinosi jeziku kreiranjem biblioteka, frejmworka i drugih alata. Neke od popularnih biblioteka i frejmworka za Pajton uključuju NumPy, Pandas, Matplotlib, Django, TensorFlow.

Razvoj Pajtona karakteriše njegova sintaksna jednostavnost, čitljivost i fleksibilnost. Ovi aspekti su pomogli da Pajton postane popularan jezik među početnicima ali i iskusnijim programerima.

2.1 Osnovne karakteristike

- Pajton je *jezik visokog nivoa* (eng. *high-level language*): programski jezik koji je dizajniran tako da ga ljudi lako čitaju, pišu i razumiju. U prevodu, obezbjeđuje koncepte koje omogućavaju pisanje kôda na višem nivou apstrakcije, čime ga približava sintaksi prirodnog jezika, za razliku od jezika niskog nivoa, kao što su asemblerski ili mašinski kôd.

- Pajton je *dinamički tipizirani jezik*. To znači da se tip varijable određuje u vrijeme izvođenja programa, a ne u vrijeme kompajliranja. Drugim riječima, tip varijable se može promijeniti tokom izvršavanja programa. Nasuprot ovome, statički tipiziranim jezicima, tip varijable je fiksiran u vrijeme kompajliranja i nepromjenljiv je tokom vremena izvršavanja. U statički tipiziranim jezicima, tip svake varijable se mora deklarirati prije upotrebe, a potom kompajler provjerava da li se tip varijable dosljedno upotrebljava u cijelom programu.
- Pajton je *objektno-orijentisani jezik*. Objektno orijentisani programski (OOP) jezik je tip programskog jezika koji koristi objekte kao svoje osnovne gradivne blokove. Objekti su instance klasa koje enkapsuliraju podatke i operacije (metode) koje se mogu izvršiti na tim podacima. U osnovi, OOP koncept potiče prirodniji način razmišljanja o problemima tako što ih modeluje u smislu objekata i njihovih interakcija.
- Pajton je *skriptni jezik*. Programski jezik koji je dizajniran dijelom za skriptiranje, pisanja tipa programa koji se koristi za automatizaciju zadataka, kao što je izvođenje niza naredbi ili manipulaciju podacima. Skriptni jezici se obično interpretiraju, a ne kompajliraju, što znači da se izvorni kod može izvršiti direktno, bez prethodnog prevođenja u mašinski kôd. Jezici za skriptiranje se često koriste za automatizacija zadataka koji se ponavljaju, pisanje malih potpornih programa ili ugrađivanje funkcionalnosti u druge aplikacije. Jedna od glavnih prednosti skriptnih jezika je njihova jednostavnost upotrebe i fleksibilnost.
- Pajton posjeduje *automatsko čišćenje memorije* (eng. *garbage-collected*). Pajton automatski upravlja identifikovanjem i oslobađanjem memorije koju program više ne koristi. Oslobađena memorija stavljanja se na raspolaganje drugim dijelovima programa za korištenje. Ovaj mehanizam omogućuje programerima da se usredotoče na pisanje koda bez vođenja brige o ručnom dodeljivanju i oslobađanju memorije (dok to nije slučaj u, recimo, programskom jeziku C).

2.2 Pregled ugrađenih tipova podataka

Pajton nudi nekoliko *ugrađenih* (eng. *built-in*) tipova podataka koji su dio osnovnog jezika. Oni su dostupni bez potrebe za dodatnom instalacijom ili

bibliotekama. Među ugrađeni tipovima podataka (koji predstavljaju klase), od naročite bitnosti su sljedeći tipovi:

- numerički tipovi: `int`, `float`, `complex`;
- tekstualni sekvencijalni tipovi: `str`, `bytes`, `bytearray`;
- sekvencijalni tipovi: `list`, `tuple`, `range`;
- mapirajući tipovi: `dict`;
- skupovni tipovi: `set`, `frozenset`;
- boolean tip: `bool`;
- binarni tipovi: `memoryview`.

Svaki od ovih tipova posjeduje različite funkcionalnosti za skladištenje i manipulaciju podacima, te se koriste za kreiranje složenih struktura podataka i algoritama. Osim toga, Pajton pruža širok raspon biblioteka koje proširuju funkcionalnost ovih ugrađenih tipova, time omogućavaju kreiranje moćnih i fleksibilnih aplikacija sa lakoćom.

2.3 Deklaracija varijabli i memorijska organizacija

Dodjela vrijednosti varijablama se izvršava pomoću operatora (dodjele) “=”, kao i kod većine programskih jezika. Da bismo kreirali varijablu, prvo odaberemo naziv varijable, a potom joj dodijelimo vrijednost pomoću operatora “=”. Npr. za kreiranje varijable pod nazivom *x* sa vrijednošću 5, koristimo sljedeći kod:

```
x = 5
```

Da bismo kreirali varijablu pod nazivom *y* i dodijelili joj stringovnu vrijednost “abc”, napišemo:

```
y = "abc"
```

Više o cjelobrojnim i stringovnim tipovima podataka biće riječi u nastavku.

Pajton je objektno-orientisani jezik, gdje se svaka varijabla tretira kao objekat neke klase. Objekti koji su varijable sa cjelobrojnomo vrijednošću se tretiraju kao objekat tipa `int`. Dalje, varijabla koja je referencirana na string se tretira kao objekat tipa `str`. Inače, gledajući memorijsku organizaciju, pojednostavljeno govoreći varijable predstavljaju reference na memorijsko mjesto gdje je upisan sadržaj na koji referenciraju. Detaljnije, pajtonov interpreter (o kome će biti više riječi u nastavku) je napisan u C-u, pa su svi pajtonovi objekti prikrivena verzija C struktura, stoga sadrže ne samo svoju vrijednost, već i druge informacije. Objekat nije samo sirovi cijeli broj (ili string ili drugi tip podatka), već pokazivač na složenu C strukturu (npr. `PyLongObject`) koja sadrži nekoliko vrijednosti. Ako istražimo dublje, možemo saznati kako ova C struktura izgleda.

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

gdje je

1. *ob_refcnt* – brojač referenci koji upravlja dodjelom i oslobađanjem memorije.
2. *ob_type* – tip varijable.
3. *ob_size* – veličina podatkovnih članova.
4. *ob_digit* – čuva stvarnu vrijednost koju varijabla predstavlja.

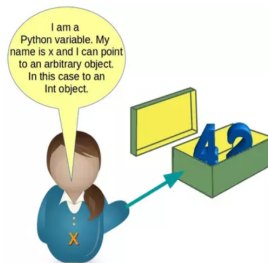
Sve ove dodatne informacije iziskuju dodatne troškove u smislu memorijskih i računarskih resursa. Stoga je pajtonov objekat (nekog tipa podatka) ništa drugo nego pokazivač na poziciju u memoriji koja sadrži sve informacije o toj varijabli, uključujući količinu memorije sa stvarnom vrijednošću tog tipa. Dakle, sve ove dodatne informacije su ono što omogućuje da slobodno kodiramo u pajtonu, bez vođenja brige o tipovima podataka varijabli. Ipak, navedimo da se tip varijable može eksplicitno dodijeliti, kao u narednom primjeru

```
x: str = "abc"  
y: int = 23
```

Jednostavnosti radi, sljedeći kôd

```
x = 42
```

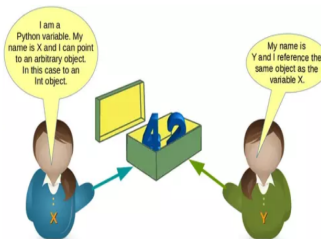
pojednostavljeno možemo predstaviti sljedećom vizuelizacijom:



U slučaju da dodamo sljedeću liniju koda

```
y = x
```

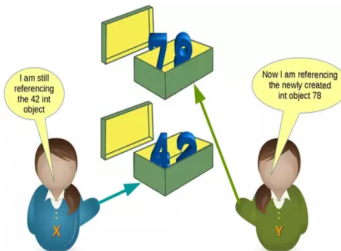
dobijamo situaciju koja odgovara slici



U slučaju da dodamo sljedeću liniju koda na prethodni kôd

```
x = 72
```

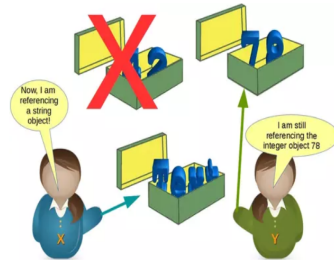
dobijamo sljedeći prikaz



Konačno, ako postavimo `y` na novu vrijednost, tj.

```
x = "abcd"
```

imamo odgovarajuću sliku



Primijetimo da na memorijsko mjesto gdje je broj 42 bio upisan ne pokazuje niti jedan pokazivač. Pajton koristi tehniku zvanu *brojač referenci* (atribut *ob_refcnt*) za upravljanje memorijom. Za svaki objekt, uključujući i složenije strukture kao što su kolekcije, vezuje se jedan broj referenci koji prati broj referenci na objekt. Kada referentni broj postane nula, objekt se briše iz memorije uz pomoć sakupljača otpadaka (eng. *garbage collector*).

Sakupljač otpadaka povremeno skenira hip u potrazi za objektima sa referentnim brojem nula te ih briše iz memorije. Na ovaj način se osigurava i slučaj ako postoje kružne reference, gdje dva ili više objekata referenciraju jedan na drugog.

Očitavanje broja memorijske ćelije te heksadekadnog broja na koju pokazuje varijabla `x` se izvršava pomoću koda:

```
idnum = id(x)
idhex = hex(id(x))
```

Napomenimo da se ispisivanje rezultata na ekranu vrši pomoću naredbe *print()*. Poruka može biti string ali i bilo koji drugi objekat. Svaki objekat će biti konvertovan u string prije nego što se ispiše na ekran. Ukoliko konverzija nije moguća, javiće se poruka o greški. Pisanje komentara u jeziku Pajton je realizovano na više načina: jednolinijski i višelinijski. Jednolinijski komentari počinju sa simbolom tab (`#`), nakon čega slijedi opis komentara. U slučaju višelinijskog komentara, početak komentara počinje i završava sa tri uzastopna apostrofa (`'''`), koji se moraju pravilno uravnati (sa ostatkom kôda), da ne bi došlo do pojave sintaksne greške.

2.4 Osnovni numerički tipovi podataka

Numerički tipovi. `int` (skraćeno od *integer*) predstavlja pozitivne ili negativne cijele brojeve, dok `float` (skraćeno od *floating-point number*) predstavlja realne brojeve sa decimalnim zarezima ili eksponencijalnom notacijom.

Slijede primjeri inicijalizacije ovakvih tipova.

```
x = 5          # int
y = -3         # int
z = 3.14       # float-point number
w = -0.325     # float-point number
```

Nad ovim numeričkim tipovima je moguće primjenjivati razne aritmetičke operacije, koje su manje-više sintaksno i smenatnički iste kao i kod većine ostalih programskih jezika. Napomenimo jednu suštinsku razliku, a to je operator `'/'` koji podrazumijeva realno dijeljenje, dok `'//'` označava cjelobrojno dijeljenje. Npr. `5/2` vraća rezultat 2.5, dok `5//2` vraća 2.

Pored ovih osnovnih aritmetičkih operacija, postoje i druge ugrađene funkcije i metode koje se mogu koristiti sa `int` i `float` vrijednostima, kao što su `abs()`, `round()`, `min()`, `max()`, itd.

Numerički tip **`complex`** predstavlja kompleksne brojeve koji su dati u obliku `a+bj`, gdje su `a` i `b` oba realni brojevi, dok je `j` imaginarna jedinica.

```
x = 1 + 1j
y = 2 + j
print(x + y) # Output: 3 + 2j
z = x + y
print(z.real) # Output: 3.0
print(z.imag) # Output: 2.0
```

Napomenimo da se kompleksni brojevi mogu inicijalizovati pomoću funkcije `complex(x, y)`, gdje je `x` vrijednost realnog, a `y` vrijednost imaginarnog dijela broja.

2.5 Naredba grananja i petlje

U ovoj sekciji navodimo sintaksu naredbe grananja `if`, te osnovnih petlji `for` i `while`.

Prije toga, u Tabeli 2.2 navodimo osnovne operatore poređenja u jeziku Pajton. Napomenimo da svaki ovakav izraz vraća bulove konstante *True* ili *False*, u zavisnosti od toga da li je tačan ili ne, respektivno.

Matematika	Pajton	Značenje
$a < b$	<code>a < b</code>	a je manje od b
$a \leq b$	<code>a <= b</code>	a je manje ili jednako b
$a > b$	<code>a > b</code>	a je veće od b
$a \geq b$	<code>a >= b</code>	a je veće ili jednako b
$a = b$	<code>a == b</code>	a je jednako b
$a \neq b$	<code>a != b</code>	a nije jednako b

Slika 2.2: Osnovni operatori poređenja u Pajtonu.

2.5.1 Uslovna naredba If

Sintaksa bazne naredbe `if` (sa neobaveznom alternativom) je data sa:

```
if uslov:
    naredba_1
else:
    naredba_2
```

Tok naredbe je sljedeći: ako je *uslov* ispunjen, izvršava se *naredba_1* (koja može da bude i blok naredbi), inače se izvršava blok *naredba_2*. Primijetimo pravila uvlačenja (indentacije) kôda pod `if` blokom. Nakon izvršavanja jedne od naredbi, program izlazi iz ugnježdenog bloka `if` naredbe, te prelazi na izvršavanje sljedeće naredbe po redu u programu. Kao i u svakom programskom jeziku, tok izvršavanja naredbi je odozgo prema dolje. Primijetimo da prethodna sintaksa za `if` radi sam samo dvije alternative. `If` koji dozvoljava više alternativa se poziva na sljedeći način.

```
if uslov_1:
    naredba_1
elif uslov_2:
    naredba_2
...
elif uslov_k:
    naredba_k
```



```

else:
    naredba_n

```

Program provjerava redom uslove krenuvši od prvog. U slučaju da se naide na uslov koji je tačan, izvršavaju se odgovarajuće naredbe (u sklopu ugnježdenog bloka), te program nastavlja sa izvršavanjem prve naredne naredbe poslije `if` naredbe. U slučaju da niti jedan od uslova nije tačan, izvršava se *naredba_n* (ulazi u ugnježdeni blok pod *else* naredbom). Nakon toga, izlazi se iz `if` bloka te program nastavlja sa prvom narednom naredbom ispod.

2.5.2 For i While petlje

Petlje se koriste za izvršavanje bloka kôda uzastopno, sve dok se ne ispuni određeni uslov (prekida). Pajton sadrži dvije vrste petlji: `for` i `while` petlja.

Petlja `for` se koristi za iterisanje kroz nizovne objekte (kao što je lista, torka, ili string) ili duge iterabilne objekte (rječnici i skupovi).

Sintaksa za petlju `for` je data sa:

```

for item in iterable:
    naredba # blok koda koji se izvršava

```

gdje je *item* varijabla kojoj se dodjeljuju elementi iterabilnog objekta *iterable* (lista, torka), jedan po jedan, dok se za svaku takvu dodjelu izvrši blok kôd *naredba*.

Sintaksa za petlju `while` je data sa:

```

while uslov:
    naredba # blok koda koji se izvršava

```

Uslov podrazumijeva bulov izraz, koji se provjerava prije svakog izvršavanja blok koda *naredba*, koja se potom izvršava ukoliko izraz *uslov* vraća vrijednost *True*, inače se izlazi iz petlje i tok izvršavanja programa seli na prvu narednu naredbu u programu van ugnježdenih naredbi pod tom petljom.

Npr. ispisivanje prvih 10 parnih cijelih brojeva pomoću *while* petlje se dobija izvršavanjem sljedećeg kôda.

```

i = 1
while i <= 10:
    print(2*i)
    i += 1 # Komentar: ekvivalentno dodjeli i = i + 1

```

2.6 Složene strukture podataka

Postoji nekoliko osnovnih struktura podataka u Pajtonu koje su fundamentalne za primjenu i rješavanje problema, a to su: stringovi, liste, rječnici (heš mape), torke i skupovi, između ostalih.

2.6.1 Stringovni tip podataka

Str je ugrađeni tip podataka (klasa) koji služi za predstavljanje niza Unicode karaktera. Stringovi u Pajtonu su nepromjenjivi; u prevodu, kada se string kreira, ne može se više mijenjati.

Inicijalizacija. Da biste se kreirao string, stavimo niz karaktera u jednostrukom (') ili dvostrukom navodniku (").

```
string0 = 'hello'
string1 = "world"
```

Višelinijski stringovi se kreiraju koristeći trostruke navodnike.

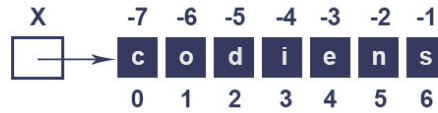
```
multiline_str2 = '''Ovo je
višelinijski string.'''
```

Metode.

- Konkatenacija stringova se realizuje pomoću operatora '+'.
 - Pristupanje karakterima stringa je obezbjeđeno operatorom '['. Karakteri stringova su indeksirani brojevima 0, 1, ... idući od početka stringa ka njegovom kraju, ili sa -1, -2, ... idući od kraja stringa ka početku (reverzno).
- Rezanje stringa (eng. *slicing*) se realizuje sa '[' na sljedeći način:

```
string1 = 'world'
substr1 = string1[1:3] # Output: 'or'
substr2 = string1[:3] # Output: 'wor'
```

- Binarnim operatorom '*' "lijepimo" više istih kopija stringa u jedan novi. Drugim argumentom operatora definišemo broj kopija datog stringa.
- Dužina stringa se dobija primjenom funkcije `len()`.



Slika 2.3: Pojednostavljen prikaz unutrašnje memorijske strukture jednog stringa u pajtonu.

- Brisanje stringa iz memorije se realizuje pomoću ključne riječi *del* koja se pozicionira ispred stringa koji se briše.
- Postoji mnogo pomoćnih funkcija koje nabrojavamo ovdje bez detalja: *lower()*, *upper()*, *strip()*, *replace()*, *find()*, itd.

Memorijska organizacija. Objekat tipa **str** je predstavljen nizom Unicode karaktera koji je smješten u neprekidnom bloku memorije. Svaki karakter u tom nizu je predstavljen brojem (kodom) tog karaktera u Unicode standardu. Svaki takav kôd je predstavljen fiksni broj bajtova, u zavisnosti od platforme, što je obično ili 2 ili 4 bajta. Stringovi se često prosljeđuju kao reference na originalni stringovni objekat.

Dodatne napomene. Objekat tipa **str** u Pajtonu predstavljen je u jeziku *C* kao struktura koja sadrži pokazivač na početak niza podataka (string), kao i druge metapodatke kao što su dužina stringa i trenutna količina dodijeljene memorije za niz.

Definicija odgovarajuće *C* strukture za pajtonov **str** objekt izgleda ovako:

```
typedef struct {
    long ob_refcnt;
    PyTypeObject *ob_type;
    Py_hash_t ob_shash;
    Py_UNICODE* st_val
} PyUnicodeObject;
```

Polje *ob_shash* se koristi za pohranjivanje keširane vrijednosti niza, a tip *Py_UNICODE* se koristi za predstavljanje Unicode znakova u procesu interpretacije programa.

Jednostavno rečeno, string pohranjuje podatke kao neprekidni niz Unicode znakova. Struktura *PyUnicodeObject* sadrži pokazivač na ove podatke.

Kada se objekt tipa `str` kreira, prvo se memorija dodjeljuje nizu, a potom se inicijalizuju i metapodaci. Pri nizovnoj manipulaciji, interni bafer se eventualno mijenja kako bi se prilagodio novim podacima. Objekt `str` je optimiziran za dijeljenje, kako je već napomenuto. Za stringove je vezan i proces *interniranja stringova* što označava ponovnu upotrebu postojećih stringovnih objekata umjesto stvaranja novih sa istom vrijednošću. Ovaj proces se izvodi globalnim keširanjem string objekata i provjerom keš memorije prije kreiranja novog string objekta. Ako string objekt sa istom vrijednošću već postoji u kešu, on se vraća umjesto kreiranja novog objekta. Ovaj proces štedi memoriju i poboljšava performanse programa, posebno kada se radi sa većim brojem manjih stringovnih objekata.

2.6.2 Liste

Lista je nehomogena, uređena, indeksirana i promjenljiva kolekcija elemenata. U jeziku Pajton se inicijalizuje kao objekat klase `list`. Elementi ove kolekcije su indeksirani brojevima, krenuvši od 0. Pajton, za razliku od većine programskih jezika, dopušta indeksiranje negativnim brojevima gdje je u tom slučaju indeks posljednjeg elementa liste -1, pretposljednjeg -2, itd.

Inicijalizacija.

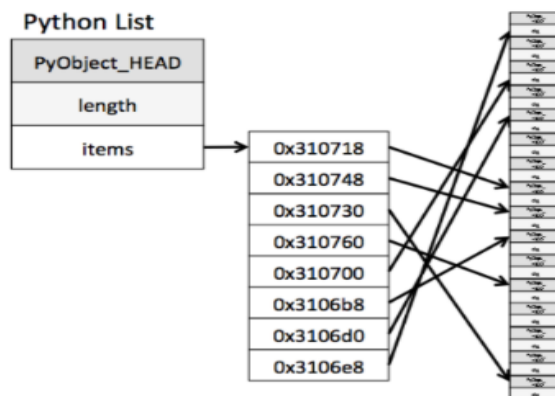
```
lista = list()
lista1 = ["a", "b", 2.0, 3]
lista2 = list(("abc", "sef", 2))
```

Osnovne metode.

- Operator `[index]`: vraća element liste na poziciji *index*;
- Operator otkidanja `[indeks1 : indeks2 : k]`: vraća listu uzimajući svaki *k*-ti element date liste krenuvši sa elementom na poziciji *indeks1* (0 ako nije naveden), pa sve do elementa sa indeksom *indeks2* (dužina liste, ako nije naveden) isključno;
- Operator *in*: ispituje da li je element u listi – vraća *True* ili *False*;
- `for x in lista`: iterisanje kroz listu prema indeksu;
- `append(elem)`: dodaje se element na kraj liste;

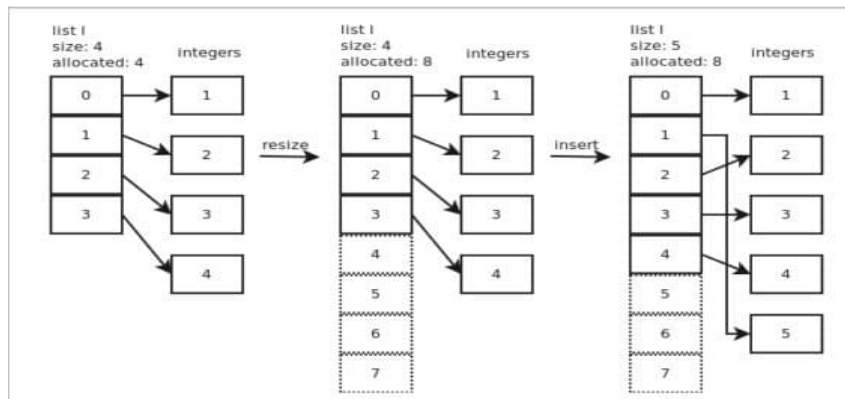
- `remove(elem)`: briše element *elem* iz liste;
- `pop()`: briše posljednji element iz liste;
- `insert(index, elem)`: postavlja element *elem* na poziciji *index* u listi;
- `reverse()`: vraća reverznu listu date liste;
- `copy()`: pravi kopiju elemenata liste u drugu listu.

Memorijska organizacija. U pajtonu su liste realizovane kao *dinamički* nizovi. To znači da mogu povećavati ili smanjivati veličinu shodno dodavanjem ili brisanjem elemenata, respektivno. Upravljanje memorijom listi se obezbeđuje interpreterom. Pri inicijalizaciji liste, jedan blok memorije se određuje za skladištenje njenih elemenata, pogledati Sliku 2.4. Veličina ovog bloka memorije zavisi od broja elemenata u listi i korištenog pajtonovog interpretera. Npr. u interpreteru (CPython) memorija je unaprijed dodijeljena u komadima (eng. *chunk*).



Slika 2.4: Liste u pajtonu: memorijska organizacija

Kada se lista promijeni, interpreter po potrebi dodjeljuje novi blok memorije veće veličine gdje se postojeći elementi liste kopiraju u novododijeljeni blok. Ovaj proces se naziva *realokacija*. U slučaju kada je potrebno osigurati čuvanje niza, a realokacija nema toliko prostora, stvoriće se nova memorija i kopija što će rezultirati veoma skupom operacijom. Da bismo se to izbjegli, možemo unaprijed rezervirati potrebnu memoriju. To se može uraditi sljedećim kodom:



Slika 2.5: Liste u pajtonu (pojednostavljena prezentacija): dodavanje elementa 5 na početnu listu

```
lista = [None] * max_broj_elemenata
```

Da bi se elementi liste automatski obrisali od strane pajtonovog sakupljača otpadaka, listi je potrebno dodijeliti `None` vrijednost. U tom slučaju, svi elementi liste koji nemaju drugih (aktivnih) pokazivača koji pokazuju na date elemente se automatski brišu iz memorije.

Dodatne napomene. Kao što smo napomeuli, za razliku od nizova u programskom jeziku *C*, liste mogu biti heterogene. Međutim, ova fleksibilnost je prilično skupa. Da bi lista bila heterogena, svaki od elemenata liste mora sadržavati informaciju o vlastitom tipu, broju referenci, stvarnu vrijednost na koju pokazuje itd. Drugim riječima, svaka stavka je kompletan pajtonov objekat. Dakle, ako dalje raščlanimo, lista u pajtonu sadrži pokazivač koji pokazuje na blok pokazivača, a unutar svakog bloka, pokazivači redom pokazuju na odvojeni puni pajtonov objekat. Ako su sve varijable istog tipa, većina ovih informacija postaje suvišna. Alternativa u takvom slučaju je korištenje *array* ugrađenog tipa, o kojem više govorimo u sljedećem odjeljku.

2.6.3 Nizovi (Array)

Pajton sadrži ugrađeni modul *array* koji je sličan nizovima u jeziku *C* ili *C++*. U ovom kontejneru, podaci se smještaju u neprekidnom bloku memorije. Kao i nizovi u jeziku *C* ili *C++*, oni podržavaju unos jednog tipa podataka u isto vrijeme, dakle nisu heterogeni poput listi u Pajtonu, već homogene strukture. Indeksiranje je slično listama. Tip elemenata niza

mora biti specificiran pomoću koda, koji su dati u Tabeli 2.1 (podešavajući vrijednost parametra `typecode`).

Code	Datatype	Python type	Size (Byte)
'b' / 'B'	signed / unsigned char	int	1
'h' / 'H'	signed / unsigned short	int	2
'i' / 'I'	signed / unsigned int	int	2
'l' / 'L'	signed / unsigned long	int	4
'q' / 'Q'	signed / unsigned long long	int	8
'f'	float	float	4
'F'	double	double	8

Tabela 2.1: Tip podataka u objektu *array*

Da bismo koristili modul `array`, prvo ga uvedemo sa naredbom `import`:

```
import array
```

Inicijalizacija.

```
my_array = array('i', [1, 2, 3, 4, 5])
```

Osnovne metode.

- Pristup svakom elementu se postiže korištenjem uglaste zagrade; indeksiranje elemenata kreće od 0.

```
print(my_array[0]) # Output: 1
print(my_array[2]) # Output: 3
```

- Modifikacija elemenata se vrši pomoću operatora dodjele (kao i u listama):

```
my_array[0] = 7

print(my_array) # Output: array('i', [7, 2, 3, 4, 5])
```

- Funkcije `append()`, `extend()`, `insert()`, `remove()` i `pop()`: imaju istu ulogu kao i odgovarajuće funkcije u listi. Pogledajmo primjer sa `extend()`:

```

floats = array('f', [2.0, 3.2, 1.4 ])

tuple_floats = (1.1, 4.1, 5.2, 9.0)

floats.extend(tuple_floats) # dodavanje (kolekcije) elemenata

```

2.6.4 Rječnici

Rječnici su neuređene, promjenljive, i indeksirane kolekcije elemenata koji su dati u obliku para ključ–vrijednost. Za indeksiranje elemenata upravo služi par ključ–vrijednost koji je jedinstven na nivou objekta ove strukture. Često ime za rječnik je heš mapa. Klasa koja podržava kreiranje objekata koji predstavljaju rječnike je `dict`.

Inicijalizacija.

```

# Vitičaste zagrade
dct = {"ime": "Mirko", "godine": 30, "grad": "Banja Luka"}

# dict() konstruktor
dct1 = dict(ime="Mirko", godine=30, grad="Banja Luka")

```

Metode.

- Pristup vrijednosti odgovarajućeg ključa:

```

my_dict = {"ime": "Mirko", "age": 30, "grad": "Banja Luka"}
print(my_dict["ime"]) # Izlaz: Mirko

```

- Ažuriranje vrijednosti:

```
my_dict["godine"] = 31
```

U slučaju da dati ključ ne postoji u mapi, par (“godine”, 31) se dodaje u mapu. Ako ne želimo posljednji scenario dodavanja, koristi se metoda `update()`.

- Brisanje elementa: koristi se metod `pop(key)` ili ključna riječ `del`:


```
dct = {"ime": "Mirko", "godine": 30, "grad": "Banja Luka"}
del dct["grad"]
print(dct)  # Output: {"ime": "Mirko", "godine": 30}

dct = {"ime": "Mirko", "godine": 30, "grad": "Banja Luka"}
dct.pop("grad")
print(dct)  # Output: {"ime": "Mirko", "godine": 30}
```

- Provjera da li ključ postoji u heš-mapi: koristimo operator *in*:

```
print("ime" in dct)  # Output: True
print("pol" in dct)  # Output: False
```

- Ostale korisne metode:

- *len(dct)*: vraća broj elemenata rječnika *dct*;
- *keys()*: vraća ključeve mape;
- *values()*: vraća sve vrijednosti koji dolaze u paru sa ključevima mape
- *items()*: vraća sve parove (ključ, vrijednost) date mape:

```
for x, y in dct.items():
    print("Ključ=", x, " vrijednost=", y)
```

Memorijska organizacija rječnika. U Pajtonu, rječnici su implementirani preko heš tabela, koje su predstavljene nizom parova ključ–vrijednost, gdje su ključevi jedinstveni na nivou objekta i koriste se za vraćanje odgovarajućih vrijednosti. Proces unosa, brisanja i traženja elemenata u heš tabeli je obično veoma brz, sa očekivanim konstantnim vremenom za svaku od operacija.

Kada se rječnik kreira u Pajtonu, prvo se heš tabela određene veličine inicijalizuje u memoriji koja skladišti parove ključ–vrednost. Veličina heš tabele je obično veća od broja inicijalno unesenih elemenata, kako bi se spriječilo često (dinamičko) menjanje veličine heš mape.

Svaki ključ u rječniku se preslikava u cjelobrojnu vrijednost koristeći heš funkciju, a potom se dobijeni broj koristi kao indeks u heš tabeli. Ako se dva ključa heširaju istom heš vrijednosti, dolazi do pojave heš kolizija. Heš tabela tu situaciju rješava tako što oba para ključ—vrijednost čuva u istom indeksu, stvarajući povezanu listu parova ključ–vrijednost, pogledati Sliku 2.6.

Da bi se dohvatila vrijednost pridružena ključu, Pajton prvo izračunava heš vrijednost ključa pa ga potom koristi za lociranje indeksa u heš tabeli u kojoj je pohranjen odgovarajući par ključ-vrijednost. Ako je ključ pronađen, vraća se odgovarajuća vrijednost; u suprotnom se javlja greška.

Kada se novi par ključ-vrijednost treba dodati u rečnik, pajton izračunava heš vrijednost ključa kojeg koristi za pronalazak indeksa u heš tabeli na čiju poziciju treba da bude pohranjen novi par ključ-vrednost. Ako je indeks već zauzet drugim parom ključ/vrijednost, novi par ključ-vrijednost se dodaje u listu povezanu sa tim indeksom. Ako broj parova ključ-vrijednost u rječniku premašuje određenu granicu, heš tabela može promijeniti svoju dužinu kako bi se smanjila vjerovatnoća pojavljivanja heš kolizija u svrhu poboljšavanja performansi.

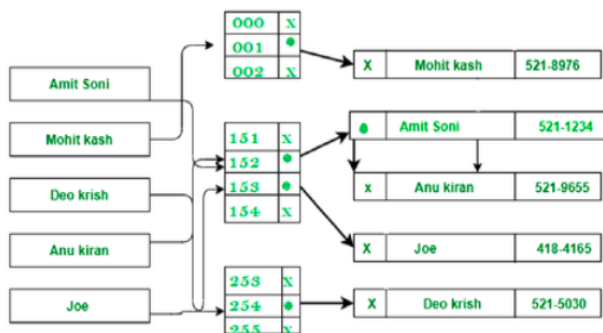
Što se tiče heš funkcija, Pajton koristi heš funkciju zvanu **hash()** za ugrađene tipove kao što su stringovi (**str**), cijeli brojevi (**int**) i torke (**tuple**). Međutim, za prilagođene objekte, može se definisati vlastita heš funkcija implementacijom metode **__hash__()**. Funkcija **hash()** vraća cjelobrojni vrijednost koja predstavlja heš vrijednost objekta. Objekti koji se upoređuju kao jednaki treba da imaju istu heš vrijednost, radi ispravnog funkcioniranja rječnika. Napomenimo da za stringove i bajtove, Pajton koristi algoritam *MurmurHash3* za hešovanje vrijednosti. Ovaj algoritam je konstruisan tako da uniformno raspoređuje heš vrijednosti minimizirajući broj kolizija. Za korisnički definisane klase, Pajton koristi zadanu **hash()** funkciju, koja generiše jedinstvenu heš vrijednost za svaku instancu klase na osnovu memorijske adrese pridružena instanci. Međutim, kao što je ranije rečeno, ovo ponašanje se može nadjačati definisanjem vlastite heš metode.

2.6.5 Skupovi

Skupovi predstavljaju neindeksirane, neuređene, promjenjive strukture podataka čija je osnovna namjena da se na efikasan način izvrši operacija provjere pripadnosti elementa. Ova struktura odgovara matematičkom pojmu skupa, što znači da duplikati u njoj nisu dozvoljeni. Skup može sadržati bilo koju vrstu elementa kao što je **int**, **float**, **tuple**, **complex**, itd. ali promjenjive kolekcije kao što su iste, rječnici, skupovi ne mogu biti elementi skupa.

Inicijalizacija.

```
skup = set()
```



Slika 2.6: Pojednostavljen prikaz unutrašnje memorijske strukture jedne heš mape (rječnika) u Pajtonu.

```
skup1 = {} #inicijalizacija praznog skupa
skup2 = set((1, 2, "3"))
skup3 = set([1, 2, 3])
```

Osnovne metode.

- *add(elem)*: dodavanje elementa *elem* u skup;
- *remove(elem)*: brisanje elementa *elem* iz skupa; izbacit će se greška, ukoliko element ne postoji u skupu (*KeyError* izuzetak, o izuzecima više u narednim sekcijama);
- *discard(elem)*: Uklanja element iz skupa ako je prisutan. Ne aktivira grešku ako element nije u skupu;
- *pop()*: Uklanja i vraća proizvoljan element iz skupa;
- *clear()*: Uklanja sve elemente iz skupa;
- *union()*: Vraća novi skup koji je unija elemenata datog skupa i drugog skupa (koji se prenosi kao vrijednost argumenta metoda); kraći zapis ove metode je pomoću znaka “|”.
- *intersection()*: Vraća novi skup koji je presjek datog skupa i drugog skupa; kraći zapis ove metode je pomoću znaka “&”.

- *difference()*: Vraća novi skup koji sadrži razliku između datog skupa i drugog skupa. Kraći zapis ove metode je pomoću znaka “-”.

Memorijska organizacija. U Pajtonu se skupovna struktura podataka implementira kao neuređena kolekcija jedinstvenih elemenata. Za implementaciju se koristi heš tabela ili rječnik; elementi skupa su ključevi rječnika dok odgovarajuće vrijednosti njima pridružene nisu od važnosti (mogu se pridružiti *None* vrijednosti, koje se za definisanje nula vrijednosti ili vrijednosti koja u suštini ne označava “ništa”. Napomenimo da je *None* vlastiti tip podatka (*NoneType*)).

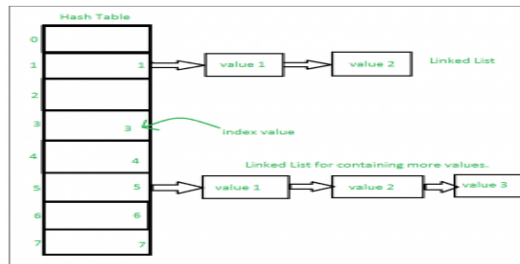
Kada se skupu doda novi element, Pajton uz pomoć odgovarajuće heš funkcije izračunava heš vrijednost koja odgovara njegovoj poziciji u heš tabeli. Ako na dobijenoj poziciji nema elementa, novi element se jednostavno dodaje u skup. U slučaju da već postoji element na toj poziciji, provjerava se da li je novi element jednak postojećem (koristeći metodu `__eq__`). Ako su dva elementa jednaka, odbija se dodavanje elementa skupu. Inače, po defaultnom principu se element dodaje u linkovanu listu pridružena datom indeksu. Takođe, određeni kompajleri koriste algoritam otvorenog adresiranja da se pronađe prazna poziciju u heš tabeli gdje bi se potom dodao novi element. Otvoreno adresiranje koristi sekvencijalno ispitivanje pozicija da bi se pronašla prazna pozicija u tabeli. Pozicije se provjeravaju sekvencijalno po nekom pravilu dok se ne pronađe prva prazna. Postoji nekoliko pravila definisanja sekvence probavanja pozicija, kao što je linearno, kvadratno ispitivanje ili dvostruko heširanje. Npr. ako heš pozicija nije prazna, izračunamo sljedeću poziciju u nizu dodavanjem neke konstantne vrijednosti (recimo 1) trenutnoj heš poziciji kretajući se ciklično po pozicijama heš tabele. Ako je trenutna pozicija 4, a konstantna vrijednost 1, sljedeća pozicija koja se ispituje je 5. Ako je sljedeća pozicija prazna, element se umeće na tu poziciju. Inače, ponovimo prethodni korak dok god se ne pronađe prazna pozicija ili dok se ne pretraži cijela heš tabela.

Slično, u slučaju kada se element uklanja iz skupa, Pajton prvo izračunava njegovu heš vrijednost pri čemu pronalazi njegovu poziciju u heš tabeli. Ako postoji element na toj poziciji, provjerava se da li je jednak elementu koji se želi ukloniti. U slučaju da jeste, element se uklanja iz skupa.

Pošto su skupovi implementirani kao heš tabele, prosječna vremenska složenost skupovnih operacija dodavanja, uklanjanja i provjere pripadnosti je konstanta, dakle $O(1)$. Međutim, u najgorem slučaju, kada postoji mnogo kolizija u heš tabeli, vremenska složenost može da bude i $O(n)$.

Sljedeći kod pokazuje tip strukture podataka u Pajtonu koja predstavlja skup, a to je upravo ugrađeni tip podataka 'set'.

```
s = {1, 2, 3}
print(type(s)) # <class 'set'>
```



Slika 2.7: Prikaz unutrašnje memorijske strukture jednog skupa (set) u pajtonu.

2.6.6 Torke

U pajtonu, torka (`tuple`) je uređena, indeksirana, nepromjenjiva (eng. *immutable*) sekvencijalna kolekcija elemenata, predstavljena jednim objektom u memoriji (na koga pokazuje), dok se ostali objekti određuju implicitnim mehanizmom Pajtona. Ova struktura dopušta unos duplikata.

Inicijalizacija.

```
my_tuple = ("aa", "bb", 2.0, 3)
my_tuple1 = tuple(["abc", 3.2, 'a'])
```

Osnovne metode.

- Pristup elementu torke:

```
my_tuple = (1, 2, 3, 2, 4, 2, 5)
print(my_tuple[2]) # Output: 3
```

- Računanje broja pojavljivanja pojedinog elementa:

```
my_tuple = (1, 2, 3, 2, 3, 5)
print(my_tuple.count(2)) # Output: 2
```

- Vraćanje indeksa prvog pojavljivanja elementa u torci:

```
print(my_tuple.index(2)) # Output: 1
```

- Vraćanje broja elemenata u torci:

```
print(len(my_tuple)) # Output: 6
```

- Spajanje dvije torke:

```
my_tuple1 = ("a", "bb", 2)
my_tuple1 = ("ee", 2, 3.4)
my_tuple2 = my_tuple + my_tuple1
print(my_tuple2) # Output: ("a", "bb", 2, "ee", 2, 3.4)
```

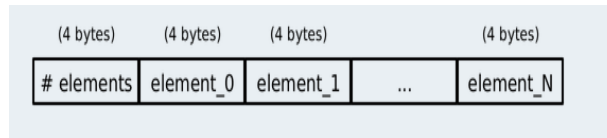
Memorijska organizacija. Pri kreiranju torke, u memoriji se rezerviše fiksna količina memorije za čuvanje svih njenih elemenata. Svaki njen element je pohranjen u kontinuiranom (neprekidnom) bloku memorije, sa fiksnom količinom prostora (nepromjenljiv objekat) dodijeljena svakom elementu na osnovu njegovog tipa.

Memorijska adresa prvog elementa torke se koristi kao referenca na cijeli objekat. U prevodu, pristup pojedinačnom elementu torke zahtijeva izračunavanje memorijske adrese željenog elementa na osnovu njegovog indeksa i veličine elemenata u torci. Za posljedicu, pristup pojedinačnim elementima torke se izvršava mnogo efikasnije od pristupa elementima liste jer indeks svakog elementa torke može da se izračuna direktno na osnovu njegove pozicije u memoriji.

Pošto su objekti tipa `tuple` nepromjenljive strukture, elementi se ne mijenjaju nakon što su kreirani. Ako je potrebno izmijeniti određene elemente, treba da se kreira novi objekat ntorka sa željenim promjenama. Strukture ntorki su efikasnije u smislu upotrebe memorije u poređenju sa promjenjivim strukturama podataka (liste, rječnici), jer nema potrebe za dodjelom dodatnih memorijskih lokacija da bismo prilagodili promjene u podacima.

2.7 Funkcije

Programski jezik Pajton svaku funkciju predstavlja kao jedan objekat. Prema tome, funkcije se mogu dodijeliti varijablama, proslijediti kao argument drugim funkcijama, a takođe vratiti kao vrijednost drugih funkcija.



Slika 2.8: Pojednostavljen prikaz unutrašnje memorijske strukture jednog objekta torke (tuple) u Pajtonu – `element_0`, `element_1`, ... predstavljaju pokazivače na vrijednosti (smješteni na uzastopnim lokacijama).

Preciznije, funkcije su implementirane korištenjem pajtonove interne klase **function**. Ova klasa je dio ugrađenih tipova ovog jezika koja pruža osnovnu funkcionalnost za kreiranje i izvršavanje funkcija. Kada se funkcija definiše, interpreter inicijalizuje novi objekat klase **function** koji predstavlja tu funkciju – sadrži informacije o nazivu funkcije, argumentima, kôdu i drugim metapodacima.

Funkcija je predstavljena blokom kôda koji obavlja određeni zadatak a može se pozivati i izvršavati više puta kroz program. Uloga funkcija je da se kôd učini modularnijim, efikasnijim i čitljivijim, jer omogućavaju podijelu složenih programa na manje dijelove kojima je lakše upravljati.

Sintaksno, funkcije se definišu pomoću ključne riječi "*def*" nakon čega slijedi naziv funkcije, zagrade (u kojima se opciono navode argumenti) pa dvotočka. Tijelo funkcije se uvlači ispod zaglavlja. Naredba **return** vraća rezultat date funkcije, nakon čega se izlazi iz tijela funkcije. Evo primjera jednostavne funkcije koja računa proizvod dva broja:

```
def proizvod_brojeva(x, y):
    prod = x * y
    return prod
```

Da bismo pozvali funkciju, koristimo njen naziv nakon čega u zagradama navodimo vrijednosti svih potrebnih argumenata. Npr. da bismo pozvali funkciju iz prethodnog primjera, pišemo sljedeći kôd:

```
res = proizvod_brojeva(4, 3)
print(res) # Output: 12
```

Funkcije mogu imati defaultne vrijednosti dodijeljene parametarima, koje se koriste ako se funkcija pozove bez specificiranja vrijednosti tim parametrima. Pogledajmo sljedeći primjer

```
def umnozi(ime="Mirko", umnozi_puta = 1):
    umnozeno = ime * umnozi_puta
    print(umnozeno)
```

Ako se prethodna funkcija pozove bez vrijednosti argumenata, defaultne vrijednosti se prosleđuju parametrima (vrijednosti “Mirko” i 1, redom), a ispisuje se poruka “Mirko”.

Dodatno, funkcije mogu imati i opcione parametre, koji su specificirani pomoću operatora zvjezdice (*) ili dvostruke zvjezdice (**). Sljedeći primjer pokazuje upotrebu ovih operatora:

```
def print_brojevi(*nums):
    for num in nums:
        print(num)
def print_vrijednosti_dict(**key_vals):
    for key, vals in key_vals.items():
        print("Ključevi: ", key, " vrijednosti: ", vals)

print_vrijednosti_dict({"ime": "Mirko", "prezime": "Mirkovic"})
```

Funkcije takođe mogu da vraćaju nekoliko vrijednosti koristeći torke:

```
def visestruko_vracanje(x, y):
    dio = x / y
    ostatak = x % y
    return dio, ostatak

d, o = visestruko_vracanje(10, 3)
print(d, " ", o) # Output: 3 1
```

U jeziku Pajton, vrijednosti argumenata funkcije se prosleđuju referencom objekta. Dakle, kada se funkcija pozove sa svojim argumentom, referenca na objekt se prosleđuje funkciji, a ne kopija samog objekta.

Ovo može dovesti do neočekivanog ponašanja u odnosu na neke druge programske jezike, posebno u radu promjenjivim objektima kao što su liste, rječnici i skupovi. Pogledajmo sljedeći primjer:

```
def dodaj_elem_u_listu(elem, lista):
```



```
lista.append(elem)

lista = list(range(1, 4))
dodaj_elem_u_listu(5, lista)
print(lista) # Output: [1, 2, 3, 5]
```

Kada se pozove funkcija `dodaj_elem_u_listu` sa vrijednostima 5 i lista, funkciji prosljeđujemo referencu na objekat lista. Izvršavanjem funkcije, mijenja se lista dodavanjem novog elementa 5. U slučaju da ne želimo da funkcija modifikuje (promjenljivi) objekt, treba da se napraviti kopija objekta prije njegovog prosljeđivanja funkciji. Više o kopiranju objekata liste će biti riječi u narednim sekcijama.

Nepromjenjivi objekti kao što su cijeli brojevi, decimalni, stringovi i torke se isto prosljeđuju referencom objekta. Međutim, kako su nepromjenjivi, dakle ne mogu se mijenjati, to znači da sve promjene koje radimo na njima unutar funkcije automatski stvaraju novi objekat (na lokalnom steku).

```
def inkrementiraj(x):
    x += 1
    print("Unutar funkcije:", x)

y = 5
inkrementiraj(y) # Output: 6
print(y) # Output: 5
```

Dakle, varijabla *x* unutar metoda referiše na novi objekat (i nema veze sa proslijeđenim objektom, osim što su istog naziva). Prema tome, iako se nepromjenljivi objekti takođe prenose putem reference na objekat, bilo kakvo djelovanje funkcije neće izmijeniti originalni objekat. Napomenimo da pomoću ključne riječi *global* ispred naziva varijable možemo preinačiti lokalno ponašanje varijable (koja dozvoljava pristup čitanju sadržaja u funkciji) u globalno (gdje je dozvoljeno i pisanje – mijenjanje sadržaja na koji varijabla referencira).

```
def my_func():
    global x
    x = 10
    print(" Vrijednost unutar funkcije :", x)
```

```
x = 20
my_func()
print(x) # Output: 10
```

Kao što smo napomenuli, funkcija je jedan objekat te se ona može proslijediti kao vrijednost argumentu druge funkcije, kako je to prikazano sljedećim primjerom

```
def dodaj(a, b):
    return a + b
def primjena(func, x, y):
    return func(x, y)

result = primjena(dodaj, 2, 3)
print(result) # Output: 5
```

Anonimne funkcije. Anonimne funkcije su definirane pomoću ključne riječi *lambda*, tako da se ponekad nazivaju i lambda funkcije. Naziv anonimne funkcije su dobile jer ne zahtijevaju eksplicitnu dodjelu naziva kao u regularnoj funkciji. Sintaksa lambda funkcije je data sa:

```
lambda argumenti: izraz
```

Pod *argumenti* se podrazumijevaju argumenti funkcije, a *izraz* predstavlja operaciju koja se izvršava nad ulaznim parametrima. Funkcija vraća rezultat koji se dobija izvršavanjem tog izraza uvrštavajući vrijednosti proslijeđenih argumenata.

Sljedeći primjer daje lambda funkciju za operaciju sabiranja.

```
suma = lambda x, y: x + y
print(suma(4, 3)) # Output: 7
```

Ovdje je lambda funkcija pridružena varijabli *suma*, što je moguće jer je svaka funkcija u programskom jeziku Pajton objekat. Potom su ovoj funkciji proslijeđene reference (na vrijednosti 4 i 3).

Lambda funkcije su često korisne u situaciji kada je potrebno definisati jednostavne (i sadržinski male) funkcije ukoje se koriste u kraćem periodu pa nema smisla da se definišu kao funkcije punog imena. Pored toga, lambda funkcije su često korištene u kombinaciji sa funkcijama, koje su opštepoznate iz funkcijskih jezika, kao što su: *map()*, *filter()*, *reduce()*, o kojima će biti više riječi u nastavku ove knjige, konkretno u Sekciji 2.9.

2.8 Kompajliranje i izvršavanje programa

2.8.1 Stek vs. Hip

Prilikom izvođenja programa, računarska memorija se dijeli na različite dijelove. Dva važna memorijska dijela su stek i hip memorija.

Pajton interpreter učitava funkcije i lokalne varijable (reference) u memoriju steka. Stek memorija je statična i privremena. Statična znači da se veličina objekata pohranjenih u steku ne može da bude promijenjena. Riječ privremeno znači da čim pozvana funkcija vrati svoju vrijednost, funkcija i varijable vezane za funkciju se uklanjaju sa steka. Pristup memoriji steka nije direktno obezbjeđen programeru, već briga o ovom dijelu memorije je dodijeljena Pajton interpreteru i OS-u. Količina memorije koja se dodjeljuje je poznata interpreteru i kad god se pozove funkcija, njene varijable zauzimaju memoriju koja se rezerviše na steku.

```
def func():  
  
    # Dve varijable dobijaju memoriju  
    # alociranu na steku  
    a = 10  
    b = []  
    c = "abc"
```

Kao što smo prethodno rekli, varijable (ili reference generalno) skladište samo memorijske adrese objekata. Dakle, postavlja se pitanje gdje se skladište sami objekti? Oni se nalaze u drugoj memoriji koja se zove *hip* (eng. *Heap memory*). Napomenimo da riječ hip nema veze sa istoimenom strukturom podataka već zbog dostupnosti gomile memorijskog prostora u svrhu pohranjivanja i brisanja objekata. Za pohranjivanje objekata potrebna nam je memorija sa dinamičkom alokacijom – veličina memorije i objekata se mogu mijenjati. Pajton interpreter je zadužen za aktivno dodeljivanje memorije hipu (Primijetimo da u jeziku C/C++ to se izvršava od strane programera!). Varijable koje su pohranjene u memoriji hipa su: (i) izvan poziva metoda ili funkcija ili (ii) dijele se unutar više funkcija, dakle globalnog svojstva.

```
# Memorija za 10 brojeva  
# je alocirana na hipu  
a = [0]*10
```

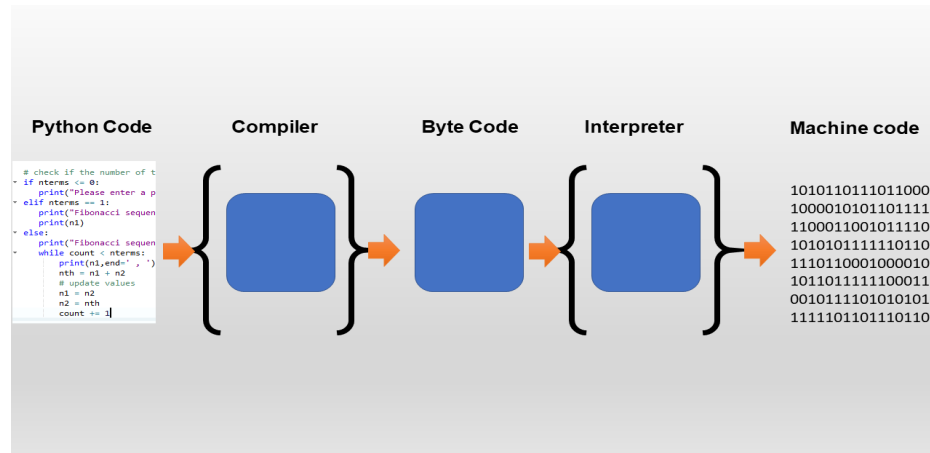
Napomenimo još jednom da Pajton koristi algoritam za sakupljanje otpadaka (eng. *garbage collector*) koji održava memoriju hipa čistom, periodično uklanjajući objekte koji više nisu potrebni.

2.8.2 Kompajliranje i izvršavanje

Proces kompilacije i izvršavanja kôda u programskom jeziku Pajton uključuje sljedeće korake:

- *Leksička analiza* – Izvorni kôd se prvo prosljeđuje leksičkom analizatoru, koji razlaže kôd na pojedinačne tokene kao što su ključne riječi, identifikatori, operatori, itd.
- *Parisiranje* – Tokeni se zatim prosleđuju parseru, koji koristi gramatiku bez konteksta za izgradnju *apstraktnog sintaksnog stabla* (AST) koje predstavlja strukturu kôda. U detalje ovog procesa nećemo ulaziti jer izlazi iz domena ove knjige.
- *Kompilacija* – AST se kompajlira u bajtkod, koji je jezik nižeg nivoa, nezavisan od platforme. Ovaj bajtkod se čuva u *.pyc* fajlovima kojeg izvršava pajtonova virtuelna mašina (VM).
- *Izvođenje* – Pajtonova VM izvršava dobijeni bajtkod tako što ga interpretira generišući mašinske instrukcije u hodu. Bajtkod se izvršava u zaštićenom okruženju virtuelne mašine (obezbjeđuje sigurnost i izolaciju od osnovnog OS-a).

Napomenimo da tokom procesa izvršavanja, interpreter izvodi nekoliko optimizacija u kodu (npr. eliminaciju repne rekurzije), što može poboljšati performanse kôda.



Slika 2.9: Vizuelizacija procesa izvršavanja pajton koda.

Recimo sada malo više o pajtonovoj virtuelnoj mašini (PVM), bazičnoj komponenti ovog programskog jezika. Kako smo naveli, PVM je odgovorna za interpretiranje bajtkoda i izvršavanje kôda. Bajtkod se generiše od strane pajtonovog kompajlera (npr. CPython, Brython) pri kompajliranju izvornog kôda.

Ključne uloge PVM su navedene u nastavku.

- *Kompatibilnost između platformi:* PVM omogućava izvršavanje pajtonovog kôda na više različitih platformi bez ponovne kompilacije. Ovo olakšava pisanje i distribuciju aplikacija.
- *Poboljšane performanse izvršavanja:* PVM koristi bajtkod interpreter, koji je brži od direktnog interpretera izvornog kôda. PVM može koristiti *Just-In-Time* (JIT) kompilaciju za dalje poboljšanje performansi.
- *Dinamičko tipiziranje:* Pajton je dinamički tipiziran jezik, što znači da se tip varijable dodjeljuje u vremenu izvođenja programa. PVM je dizajnirana da efikasno upravlja ovim procesom.
- *Upravljanje memorijom:* PVM automatski upravlja memorijom, što može olakšati pisanje sigurnog memorijskog kôda. PVM automatski oslobađa memoriju koja se više ne koristi.
- *Lak pristup eksternim bibliotekama:* PVM se intenzivno koristi u pajton ekosistemu, koji uključuje veliki broj otvorenih biblioteka. Ove biblioteke pružaju širok spektar funkcionalnosti vezano za veb programiranje,

mašinsko učenje, itd. Importovanje biblioteka se vrši pomoću naredbe `import`.

Na osnovu izloženog, Pajton je i kompajliran (prevodi se u bajtni kôd) i interpretiran jezik. Vizuelizaciju procesa prevodenja programa možete da vidite na Slici 2.9.

2.9 O nekim programskim konceptima i ugrađenim funkcijama

U ovoj sekciji ćemo demonstrirati rad sa nekoliko bitnih i često korištenih ugrađenih funkcija koje su dio standardne pajton biblioteke. Dodatno, biće objašnjen i koncept komprehenzije liste (eng. *list comprehension*) koji doprinosi modularnosti kôda i približava ga konceptima funkcijskih programskih jezika (kao što je Haskel).

2.9.1 Komprehenzija liste

Komprehenzija liste predstavlja koncizan način za kreiranje liste. To je konstrukcija koja omogućava kreiranje liste u jednoj liniji kôda, umjesto da se koriste petlje (i odgovarajuće metode) za iterativnu konstrukciju liste.

Osnovna sintaksa ovog koncepta je sljedeća:

```
lista = [expr for item in iterable]
```

Ovdje *expr* označava izraz koji se izračunava do vrijednosti koja se smiješta u listu. Dalje, *item* je varijabla koja uzima vrijednosti iz (iterabilnog) objekta *iterable*. Rezultat primjene je nova lista sastavljena od vrijednosti dobivenih primjenom izraza *expr* na svaki elemenat objekta *iterable*.

Npr. recimo da želimo kreirati listu prvih 10 parnih brojeva. Kôd bi mogao da ide ovako:

```
parni = []
for i in range(1, 11):
    parni.append(2*i)
```

Kraći zapis koda, pomoću komprehenzije liste, izgleda ovako:

```
parni = [2*i for i in range(1, 11)]
```

Takođe se uslovni izrazi mogu uključiti u koncept komprehenzije liste. Sintaksa je sljedeća:

```
lista = [expr for item in iterable if uslov]
```

U ovom slučaju, uslovna komanda se odnosi na elemente koje uzima varijabla *item* – to su samo oni koji zadovoljavaju *uslov*.

2.9.2 Funkcije map, filter, reduce

Funkcije *map()*, *filter()* i *reduce()* su ugrađene funkcije koje se opštepoznate u funkcijskom programiranju. One su korisne kao alati za obradu podataka, posebno kada se radi o velikim skupovima podataka.

Funkcija *map()* uzima dva argumenta: funkciju i iterabilni objekat. Primjenjuje funkciju na svaki element iterabilnog objekta i vraća iterator na nove elemente kao rezultat.

```
brojevi = [1, 2, 3, 4, 5]
doubled = map(lambda x: x*2, brojevi)
print(list(doubled)) # Output: [2, 4, 6, 8, 10]
```

Funkcija *filter()* uzima dva argumenta: funkciju koja vraća logičku vrijednost i iterabilni objekat. Primjenjuje funkciju na svaki element iterabilnog objekta i vraća iterator na one elemente za koje je funkcija vratila *True*.

```
brojevi = [1, 2, 3, 4, 5]
parni = filter(lambda x: x % 2 == 0, brojevi)
```

Funkcija *reduce()*: Sintaksa ove funkcije je

```
reduce(func, iterable[, initial ])
```

Ona uzima dva (obavezna) argumenta, funkciju i iterabilni objekat, te opcionalno *initial*. Inicijalno, funkcija *func* se primjenjuje na element *initial* i prvi element iterabilnog objekta, te vraća rezultat. Potom, primjenjuje istu funkciju na vraćeni rezultat i sljedeći element, i tako dalje, sve dok se svi elementi *iterable* objekta ne obrade.

Pogledajmo sljedeći primjer:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
```

Ovo će primijeniti funkciju množenja na prva dva elementa liste, zatim na rezultat i sljedeći element, i tako dalje, sve dok se svi elementi ne obrade, što rezultira proizvodom svih brojeva liste. Pirmijetimo da funkcija *reduce()* nije dio standardnog pajtonovog modula, već se mora uključiti direktno pozivom *functools* modula. Više o modulima i radu sa njima ćemo govoriti u narednim sekcijama.

2.9.3 Funkcije *zip*, *enumerate* i *sort*

Funkcije *zip()* i *enumerate()* su ugrađene funkcije korisne za obradu i iterisanje preko lista ili drugih iterabilnih objekata.

Funkcija *zip()* uzima jedan ili više iterable objekata i vraća iterator koji generiše torke koje sadrže uparene elemente svakog iterabilnog objekta (prvi elementi svih listi u jednu torku, drugi elementi svih listi u novu torku, itd.). Sintaksa funkcije je:

```
zip(iterable1, iterable2, ...)
```

gdje su *iterable1*, *iterable2*,... iterabilni objekti.

Npr. ako imamo dvije liste brojeva i želite da uparite odgovarajuće elemente, imamo sljedeći kod:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
parovi = zip(list1, list2)
print(list(parovi)) # Output: (1, 5), (2, 6), (3, 7), (4, 8).
```

Funkcija *enumerate()* uzima iterativni objekat i vraća iterator koji generiše parove koji sadrže indeks svakog elementa i samog elementa. Sintaksa funkcije je:

```
enumerate(iterable, start=0)
```

gdje je *iterable* jedan iterabilni objekat, a *start* je početna vrijednost indeksa (podrazumijeva se 0). Pogledajmo sljedeći primjer.


```
voce = ['jabuka', 'banana', 'kruska', 'breskva']
for i, vocka in enumerate(voce):
    print(i, vocka) # (0, 'jabuka'), (1, 'banana'), (2, 'kruska'), (3, 'breskva')
```

Funkcija `sort()` je ugrađena funkcija koja se koristi za sortiranje elemenata liste po nekom kriterijumu. Ona je metoda liste koja ne vraća ništa već je njen zadatak sortirati listu (u memoriji koja je već dodijeljena).

Sintaksa ove funkcije data je u nastavku

```
sort(reverse=False, key=None)
```

Argument *reverse* prosljeđuje poredak sortiranja u listi na način da ukoliko mu je dodijeljena vrijednost *True*, lista će biti sortirana u opadajućem, a inače (po defaultu) redoslijed sortiranja je rastući. Kriterijum sortiranja (kao funkcija) se prosljeđuje argumentu funkcije *key*. Primijetimo da ukoliko se kriterijum sortiranja ne proslijedi eksplicitno, Pajton koristi interne relacije poretka za tip objekata, ukoliko su oni uporedivi (predefinisanom relacijom totalnog uređenja, kao kod osnovnih tipova podataka). U slučaju da objekti nisu uporedivi, vraća se poruka o greški. Jedna od mogućih grešaka koja se može javiti je sljedeća:

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

Primjer sortiranja numeričkih vrijednosti je dat kôdom:

```
brojevi = [6, 3, 8, 3, 1, 7]
brojevi.sort(reverse=True)
print(brojevi) # Output: [8, 7, 6, 3, 3, 1]
```

Takođe je moguće koristiti funkciju `sorted()` za sortiranje liste i vraćanje nove sortirane liste bez mijenjanja originalne liste:

```
brojevi = [5, 2, 8, 3, 1, 7]
sort_brojevi = sorted(brojevi)
print(sort_brojevi)
```

Postoji i niz drugih pomoćnih funkcija od opšte koristi, koje ćemo pobrojati u nastavku bez ulaska u detalje o njihovoj sintaksi i korišćenju:

- `sum(iterable)`: vraća sumu vrijednosti elemenata iterabilnog objekta *iterable*;

- *eval*(*expr*): vraća vrijednost izraza *expr* zapisan u stringovnom zapisu;
- *round*(*num*, *dec_place*): zaokruživanje decimalnog broja na proslijeđen broj decimala;
- *max*(*iter*): vraća broj maksimalne vrijednosti među proslijeđenim brojevima;
- *abs*(*num*): vraća apsolutni vrijednost broja.

2.10 Uopšteno o modulima

Modul je datoteka koja sadrži Pajtonove definicije i naredbe. Naziv datoteke je naziv modula sa sufiksom *.py*.

Modul može sadržati različite definicije kao što su funkcije, klase i varijable, koje se mogu uvesti i koristiti u drugim Python skriptama ili modulima. To olakšava organiziranje i ponovnu upotrebu kôda u različitim projektima.

Da bismo koristili modul u Pajtonu, prvo ga mora uvesti pomoću naredbe *import*. Npr. ako imamo modul pod nazivom *mojmodul.py*, možemo ga uvesti na sljedeći način:

```
import mojmodul
```

Nakon što smo uvezli modul, možemo pristupiti njegovim funkcijama, klasama i varijablama pomoću notacije tačka. Npr. ako *mojmodul* sadrži funkciju pod nazivom *mojafunkcija*, poziva se ovako:

```
mojmodul.mojafunkcija()
```

Takođe se može koristiti i ključna riječ *from* za uvoz određenih funkcija ili klasa iz modula:

```
from mojmodul import mojafunkcija
```

Ovo će omogućiti direktno korištenje funkcije bez potrebe za tačaka notacijom:

```
mojafunkcija()
```

Pajton dolazi s velikim brojem ugrađenih modula, među kojima izdvajamo *math*, *random*, *datetime*, *os* i *numpy* o kojima će biti riječi u nastavku ove sekcije. Ovi moduli se uključuju direktno sa instalacijom jezika Pajton.

2.10.1 Instaliranje eksternih modula

Da bismo instalirali eksterne module u Pajtonu, koristi se menadžer paketa *pip*, koji je direktno uključen u većinu instalacija pajtona.

Koraci za instaliranje eksternih modula pomoću *pip*-a su dati u nastavku. Otvorimo terminalnu konzolu i upišemo naredbu *pip install naziv_modula*, gdje je *naziv_modula* (obično) odgovara nazivu modula kojeg želimo instalirati. Npr. da bismo instalirali modul *requests*, pišemo:

```
pip install requests
```

Pritiskom enter tastera, instalacija se pokreće; *pip* automatski preuzima modul iz PyPI (eng. *Python Package Index*) i instalirati ga u trenutno pajton okruženje. PyPI je veliki repozitorij softverskih paketa koji su napisani u pajtonu, besplatno dostupni za korištenje. Sadrži više od 300,000 paketa koje možemo instalirati pomoću *pip* menadžera.

Kada proces instalacije modula uspješno završi, on se interno može koristiti pomoću naredbe *import*. Npr. da bismo koristili modul *requests*, pišemo sljedeće:

```
import requests
```

Takođe se možete navesti određena verziju modula za instalaciju dodavanjem “==” iza kojeg slijedi broj verzije nazivu modula. Npr. da bismo instalirali verziju 2.2.1 modula *requests*, pisali bismo:

```
pip install requests==2.2.1
```

2.10.2 Modul Math

Math je matematički modul koji za rad nudi različite matematičke funkcije i konstante. Da bismo koristili matematički modul u svom programu, prvo morate uvesti pomoću naredbe *import math*.

Nakon što se uveze matematički modul, omogućeno je korištenje njegovih funkcija i konstanti u kôdu. Navedimo nekoliko konkretnih primjera.

```
import math
```

```
# Računajmo korjen broja  
x = math.sqrt(25)
```

```

print(x)  # Output: 5.0

# Konstanta Pi
pi = math.pi
print(pi)  # Output: 3.141592653589793

# Računanje sin
sin_angle = math.sin(45)
print(sin_angle)  # Output: 0.7071067811865475

```

Neke od češće korištenih metoda ovog modula su: *sqrt()*, *pow()*, *log()*, *exp()*, *log10()*, *floor()*, *ceil()*, *fabs()*, *gcd()*, *dist()* – Euklidova udaljenost između dvije tacake (predstavljene kao torke, ili liste), *isnan()*, *sin()*, *cos()*, *tan()*, *sinh()*, *asin()*, među ostalim metodama.

U nastavku navodimo jednu od internih implementacija korjene funkcije *sqrt()* broja $x \geq 0$. Ona je implementirana korištenjem algoritma koji se zove *Babilonova metoda*, poznata i pod nazivom Heronova metoda. Algoritam je numerička iterativna metoda koja aproksimira kvadratni korijen broja x uzimajući prosjek prethodne aproksimacije i vrijednosti količnika broja x i vrijednosti dobijene u prethodnoj aproksimaciji. Proces se nastavlja sve dok aproksimacija nije dovoljno precizna. Preciznije, primjeljuje se naredna rekurzija:

$$y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$$

za neki proizvoljan inicijalni korak $y_0 > 0$. Trivijalno, za $x = 0$, vraća se vrijednost 0. Algoritam se izvršava dok se ne postigne unaprijed zadana preciznost ϵ broja x , tj. dok ne bude ispunjeno $|y_{n+1} - \frac{x}{y_n}| \leq \epsilon$.

2.10.3 Modul Random

Modul *random* pruža skup funkcija za generiranje (pseudo-)slučajnih brojeva, korisni u različitim aplikacijama kao što su kriptografija i razvoj igara.

Navedimo neke osnovne funkcije ovog modula.

- *random()*: slučajan nasumični *float* broj između 0 i 1, uključujući 0, ali isključujući 1;
- *randint(a, b)*: generiše nasumični cijeli broj između a i b , uključujući oba broja, a i b .

- *choice(seq)*: vraća slučajan element datog niza *seq*;
- *shuffle(seq)*: nasumično permutuje elemente u datom nizu *seq*;
- *sample(populacija, k)*: Ova funkcija vraća nasumično odabran uzorak od *k* elemenata iz date populacije *populacija* bez zamjene;
- *seed(a=None)*: inicijalizuje generator slučajnih brojeva. Ako argument *a* nije naveden, koristi se sistemsko vrijeme.

Implementacija funkcije *random()* u Pajtonu koristi algoritam generatora pseudo-slučajnih brojeva, koji je deterministički a generiše niz brojeva koji se čini slučajnim. Algoritam koji se koristi u funkciji *random()* je algoritam *Mersenne Twister*, široko korišten za generiranje pseudo-slučajnih brojeva.

Algoritam *Mersenne Twister* generiše niz 32-bitnih cijelih brojeva koristeći specijalni linearni pomaknuti (eng. *shift*) registar povratne sprege, koji koristi operacije isključivanja ili (XOR) za generiranje izlaza. Algoritam koristi veliki prost broj koji se zove *Mersenov* prost ($2^{19937} - 1$) kao modul, koji osigurava da generisani niz ima dug period i dobra statistička svojstva. Inače, algoritam su razvili Makoto Matsumoto i Takuji Nishimura 1997. godine. Drugi generator koji se može koristiti je prosti *linerni kongruentni generator* koji generiše slučajne brojeve po rekurziji

$$x_{n+1} = (ax_n + b) \% m$$

gdje je x_0 početni sid vrijednosti između 0 i $m - 1$.

2.10.4 Modul Time

Modul `time` je ugrađeni modul koji pruža razne funkcije povezane sa radom sa vremenom. Omogućuje izvođenje raznih operacija, kao što su vraćanje trenutnog vremena, uspavlivanje procesa na određeno vrijeme, pretvaranje formata vremena i još mnogo toga.

U nastavku navodimo najčešće korištene funkcije koje pruža ovaj modul:

- *time()*: Vraća trenutno vrijeme u sekundama od epohe (eng. epoch) (računa se kao datum 01.01.1970) kao float tip.
- *time.sleep(secs)*: obustavlja izvršenje trenutne niti na zadani broj sekundi *secs*.

- `time.localtime([secs])`: Pretvara vrijeme u sekundama *secs* od epohe u torku koja predstavlja lokalno vrijeme u obliku: godina, mjesec, dan, sat, minuta, sekunda, radni dan, Julijanski dan, DST.

2.10.5 Modul Os

`Os` modul je ugrađeni modul koji pruža mogućnost korištenja funkcionalnosti zavisne od operativnog sistema kao što je čitanje ili pisanje u sistem datoteka, upravljanje procesima, varijablama okruženja i mnogi drugim sistemskim zadacima.

Nekih od najčešće korištenih funkcija koje pruža ovaj modul su:

- `os.name`: Vraća naziv operativnog sistema (npr. 'posix' za Linux i macOS, 'nt' za Windows).
- `getcwd()`: Vraća trenutni radni direktorij kao u obliku stringa.
- `chdir(path)`: Mijenja trenutni radni direktorij na specificiranu stazu *path*.
- `listdir(path)`: Vraća listu datoteka i direktorija u navedenom direktoriju.
- `mkdir(path, mode=0o777)`: Kreira novi direktorij sa specificiranom putanjom i privilegijama.
- `makedirs(name, mode=0o777, exist_ok=False)`: Kreira novi direktorijum i sve potrebne među-direktorijume sa navedenim nazivom i načinom rada.
- `remove(path)`: Uklanja datoteku sa navedenom putanjom *path*.
- `rmdir(path)`: Uklanja direktorijum na navedenoj putanji.
- `rename(src, dest)`: Mijenja naziv datoteke ili direktorijuma na navedenoj izvornoj stazi *src* sa nazivom na novoj stazi *dest*.
- `path.exists(path)`: Vraća *True* ako navedena putanja postoji, *False* u suprotnom.
- `path.isfile(path)`: Vraća *True* ako se na navedenoj stazi nalazi datoteka, *False* u suprotnom.

- `path.isdir(path)`: Vraća *True* ako je navedena staza direktorij, *False* u suprotnom.

2.11 Izuzeci

Izuzetak predstavlja (nesintaksnu) grešku koja se javlja tokom izvršavanja programa, kada se nešto neočekivano dogodi kao npr. dijeljenje sa nulom, datoteka nije pronađena, ili greška u tipu podatka.

Rad sa izuzecima u programskom jeziku Pajton uključuje identifikovanje i rukovanje sa ovim greškama kako bi se spriječilo da one uzrokuju pad izvršavanja programa. Osnovni načini za rad sa izuzecima su dati u nastavku.

Try-except blok: koristi se za hvatanje i obradu izuzetaka. Kôd unutar bloka *try* se izvršava, a u slučaju se dogodi izuzetak, izvršava se kôd unutar bloka *except*. Evo primjera:

```
try:
    # kod koji moze "baciti" izuzetak
except ExceptionType:
    # kod za obradu izuzetka
```

Višestruki blokovi *except*:

```
try:
    # kod koji moze izbaciti izuzetak
except ValueError:
    # kod za hvatanje izuzetka greske vrijednosti
except TypeError:
    # kod za hvatanje izuzetka greske tipa
except Exception:
    # kod za hvatanje drugih izuzetaka
```

Izbacivanje izuzetaka bez navođenja bloka: Izuzeci se mogu aktivirati i pomoću `raise` naredbe, koristeći inlajn notaciju, kao npr.

```
raise ValueError("Invalid input")
```

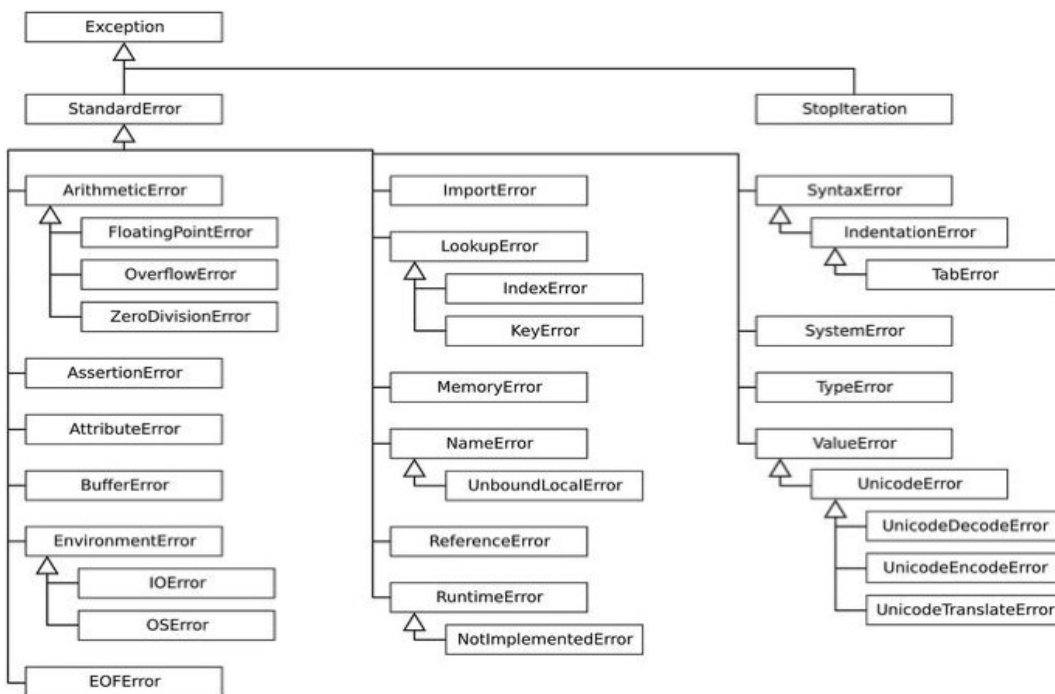
Finally blok: Za izvršavanje koda bez obzira na to da li je izuzetak aktiviran ili ne koristi se *finally* kao u narednom primjeru:

```

try:
    # kod koji može da aktivira izuzetak
finally:
    # kod koji će se uvijek izvršiti, bez obzira na to
    # hoće li se izuzetak desiti

```

Sve vrste klasa izuzetaka, te njihova organizacija u Pajtonu se mogu vidjeti sa Slici 2.10.



Slika 2.10: Klasa izuzetaka u programskom jeziku Pajton.

Postoji još jedan mehanizam koji uslovno aktivira izuzetak, a to je naredba **assert**. Ona provjerava prov da li je uslov tačan (*True*). U slučaju da nije, izbacuje se poseban tip greške **AssertionError** aktivirajući navedenu poruku o grešci, gdje se potom izvršavanje narednih dijelova program zaustavlja. Primjer upotrebe naredbe **assert** je dat sljedećim kôdom

```

x = 5
assert x > 0, "x is not positive"

```


Da zaključimo, rad sa izuzecima je važna praksa u pisanju robusnog koda koji je tolerantan na greške. Izuzeci nam omogućavaju elegantno rješavanje neočekivane situacije koja se javlja pri izvršavanju programa te pruža korisniku korisne povratne informacije.

2.12 Rad sa datotekama

Pomoću svojih ugrađenih funkcija, Pajton omogućuje jednostavan rad sa ulaznim i izlaznim datotekama. U nastavku dajemo pregled najčešće korištenih metoda.

Čitanje iz datoteke. Koristimo funkciju `open()`, koja ima dva argumenta – naziv datoteke (ili putanja do datoteke) i način u kojem se otvara datoteka (npr. za čitanje, pisanje, dodavanje, itd.). Pogledajmo sljedeći primjer:

```
# Otvaranje fajla za čitanje
file = open("datoteka.txt", "r")
# Čitanje sadržaja iz fajla
sadrzaj = file.read()
# Zatvaranje fajla
file.close()
# Ispisivanje sadržaja fajla
print(sadrzaj)
```

Dakle, prvo otvaramo datoteku `datoteka.txt` samo za čitanje. Zatim čitamo sadržaj datoteke koristeći metodu `read()` i čuvamo ga u varijabli `sadrzaj`. Konačno, zatvaramo datoteku pomoću metode `close()` i ispisujemo sadržaj datoteke.

Pisanje u fajl. Za pisanje u datoteku koristimo funkciju `open()`, koja otvara fajl sa određenim nazivom, ali ovaj put sa načinom “w” (eng. write), čime se dopušta pisanje u fajl. Pogledajmo sljedeći primjer.

```
# Otvaranje fajla za pisanje
file = open("datoteka.txt", "w")
# Pisanje teksta u datoteku
file.write("Zdravo!")
# Zatvaranje fajla
file.close()
```

Dodavanje u fajl. Da biste dodali sadržaj u postojeći sadržaj datoteke, otvaramo fajl sa načinom “a” (eng. append), a potom pomoću funkcije `write()` upisujemo novi sadržaj na postojeći.

Napomenimo da je dobra praksa zatvoriti datoteku nakon što završite rad sa njom, kako bismo izbjegli bilo kakve potencijalne probleme sa oštećenjem datoteke ili gubitkom podataka. Praksa je da se rad u fajlovima odvija pod *try-except* blokom. Klase izuzetaka koje se u tim slučajevima mogu koristiti su: `FileNotFoundError` i `OSError`.

2.12.1 Modul Json

Modul `json` se koristi za kodiranje i dekodiranje podataka sa i u JSON format (eng. *Java Script Object Notation*) formatu. JSON je široko rasprostranjen format za razmjenu podataka, ljudima jednostavan za čitanje i pisanje, a mašinama za raščlanjivanje i generisanje.

Modul `json` nudi dvije glavne metode za rad sa JSON podacima:

- `json.dump()` – koristi se za kodiranje pajton objekata u JSON format i njegovo spremanje u objekat datoteka.
- `json.loads()` – koristi se za dekodiranje JSON niza u pajtonov objekat (obično heš mapa).

Pogledajmo i jedan primjer upotrebe `json` modula.

```
import json

# Pajton objekat
objekat = {'ime': 'Mirko', 'godine': 30, 'grad': 'Banja Luka'}
# Enkoduj objekat u JSON string
json_string = json.dumps(objekat)
# Dekodiraj JSON string nazad u pajtonov objekat (dict)
dekodiran_objekat = json.loads(json_string)

print(type(json_string))    # <class 'str'>
print(type(dekodiran_objekat)) # <class 'dict'>
print(dekodiran_objekat)    # {'ime': 'Mirko', 'godine': 30,
                             # 'grad': 'Banja Luka'}
```

Primijetite da funkcija *dumps()* pretvara pajtonov objekat u string (kodiranje), dok funkcija *loads()* radi obrnutu stvar, od stringa pravi pajtonov objekat (enkodiranje).

2.13 Regularni izrazi

U ovoj sekciji izlažemo uopštено o regularnim izrazima. Potom, će biti riječi o pajtonovom modulu **re** specijalno namijenjenog za rad sa regularnim izrazima. Na kraju ćemo dati nekoliko primjera upotrebe ovog modula.

2.13.1 Uopštено o regularnim izrazima

Regularni izraz ili *regex* (šablon ili patern) je niz znakova koji opisuju skupove nizova znakova, na osnovu određenih sintaksnih pravila. U prevodu to je šablon koji opisuje neki skup stringova. Svrha regularnih izraza je opisivanje uzorka za pretraživanje nizova znakova konciznim opisom bez nabiranja svih elemenata (stringova) skupa. Npr. kako koncizno opisati sve binarne nizove pomocu šablona ili kako koncizno opisati skup {gray, grey} (engleski i americki engleski za istu riječ)? Regularne izraze koriste razni uređivači teksta, kao što je filter *grep* u Unix-u, itd.

Osnovni koncepti regularnih izraza su:

- *Alternacija* – izražava se pomoću znaka |, te označava sve alternativne mogućnosti koje se mogu uzeti; npr. regularni izraz $a | b | c$ označava skup riječi {*a*, *b*, *c*}.
- *Zagrada* – označava grupisanje, tj. područje djelovanja. Npr. **gr(a|e)y** označava patern kojim u obuhvaćeni riječi **gray** i **grey**.
- *Specijalni karakteri i skup karaktera*:
 - “.”: sparuje bilo koji karakter samo jednom;
 - []: sparuje bilo koji karakter pod (uglastim) zagradama samo jednom. Npr. ako želimo da označimo skup sva malih slova, odgovarajući patern bi bio $[a - z]$, dok bismo velika označili sa $[A - Z]$, a jedna i druga sa $[a - zA - Z]$
 - \d: sparuje bilo koju cifru samo jednom. Drugi način je koristiti patern sa zagradama: $[0 - 9]$;

- `[^]`: sparuje bilo koji karakter tačno jednom koji pripada komplementu skupa karaktera pod uglastim zagradama;
 - `\w`: sparuje bilo koji karakter riječi (slova, brojevi ili povlaka);
 - `\n`: sparuje novi red;
 - `\s`: sparuje blanko simbol;
 - `\b`: sparuje granicu riječi – položaj između znaka riječi i znaka koji nije riječ, kao što je razmak, interpunkcija oznaka ili početak/kraj reda;
 - `\t`, `\.`, `...`
- *Kvantifikatori* – navode se nakon znaka ili grupe, i oni određuju učestalost pojavljivanja izraza koji prethodi kvantifikatoru:
 - `“?”`: izraz prije pojavljivanja kvantifikatora se pojavljuje 0 ili 1 puta;
 - `“*”`: izraz prije pojavljivanja kvantifikatora se pojavljuje 0 ili više puta;
 - `“+”`: izraz prije pojavljivanja kvantifikatora se pojavljuje 1 ili više puta;
 - `{m,n}`: izraz prije ovog kvantifikatora se pojavljuje od m do n puta (uključujući i granične vrijednosti).

Regularan izraz za registarske tablice automobila su date u formatu tri broja, potom povlaka, pa jedno (veliko) slovo, pa potom tri broja, što odgovara regularnom izrazu:

$$\backslash d\{3\} - [A - Z] - \backslash d\{3\}$$

Regularni izraz za definisanje datum rođenja (sa vodećim nulama za jednocifreni dan i mjesec rođenja; npr. 1 se označava sa 01) izgleda ovako:

$$(\backslash d\{2\})\backslash . (\backslash d\{2\})\backslash . (\backslash d\{4\})\backslash .$$

2.13.2 Modul Re

Modul `re` je ugrađeni modul koji je namijenjen za rad sa regularnim izrazima. Ovaj modul omogućuje programerima da izrade i primjenjuju regularne izraze, te da se efikasno nalaze paterni u tekstu koji se uklapaju u njih.

Neke od metoda modula `re` su date u nastavku.

- Metod *search*(pattern, str): pretraživanje niza znakova za određen *pattern* u tekstu *str*.
- Metod *match*(pattern, str): pronalaženje uzoraka na osnovu obrasca *pattern* iz niza znakova *str*; vraćanje po redoslijedu pojavljivanja uzorka se vrši primjenom funkcije *group*() na objekat vraćen prethodnom funkcijom, (Vrijednost argumenta koji se prosljeđuje označava redni broj uzorka koji se izdvaja.)
- Metod *sub*(pattern, strChange, str): mijenja sva podudaranja obrasca *pattern* sa drugim nizom znakova (*strChange*) u stringu *str*.
- Metod *split*(pattern, str): razdvaja niza znakova stringa *str* na osnovu obrasca *pattern*, te vraća niz razdvojenih (pod)stringova.

U nastavku navodimo jedan primjer primjene modula **re** i funkcije *search*().

```
import re
# definiranje uzorka koji se traži
pattern = r'\b\w+'
# niz znakova za pretraživanje
text = 'Hello, World!'
# pretraživanje po obrascu
match = re.search(pattern, text)
# ispis (prvo) podudaranje
print(match.group())
```

Drugim riječima, odgovara bilo kojem nizu jednog ili više uzastopnih znakova riječi kojima prethodi granica riječi (položaj između znaka riječi i znaka koji nije riječ, kao što je razmak, oznaka interpunkcije ili početak/kraj reda.

2.14 Moduli Sys i Getopt

Pokretanje programa preko terminala. Pokretanje pajton skripte preko terminala se izvršava na sljedeći način:

```
python myscript.py arg1 arg2
```

gdje se komanda *python* (ili *python3*, ako je instalirana pajtonova verzija 3.0 ili viša) postavlja preko varijable okruženja `PYTHONPATH`.

Modul `sys` omogućuje pristup nekim parametrima i funkcijama specifičnim za sistem.

Neke od često korištenih funkcija i atributa `sys` modula su:

- *argv*: Lista argumenata komandne linije proslijeđenih pajtonovoj skripti. Na indeksu 0 se čuva naziv same skripte, dok se na ostalim indeksima čuvaju vrijednosti proslijeđenih parametara.
- *path*: Vraća listu stringova koji specificiraju stazu do pojedinih modula. Inicijalizuje se iz varijable okruženja `PYTHONPATH` i zadate vrijednosti koja zavisi o instalaciji pajtona.
- *exit*([arg]): Izlazak iz pajtonovog interpretera sa opcionim izlaznim kôdom. Ako je dat *arg*, on se koristi kao izlazni kôd; inače je zadani izlazni kod nula.
- *version*: Niz koji sadrži verziju pajtonovog interpretera.
- *platform*: Niz koji identikuje platformu operativnog sustava.

U nastavku slijedi nekoliko programa koji pokazuju rad sa `sys` modulom.

```
import sys
if len(sys.argv) > 1:
    print("Argumenti komandne linije su:")
    for arg in sys.argv[1:]:
        print(arg)
else:
    print("Nema proslijeđenih argumenata komandne linije.")
```

Izvršavanjem prethodnog programa dobijamo izlaz:

Argumenti komandne linije su:

arg1
arg2

Modul `getopt` predstavlja parser argumenata komandne linije. Omogućuje pisanje skripti koje mogu uzimati argumente i opcije iz komandne linije pri izvršavanju programa. Modul `getopt` ima dvije bitne funkcije, `getopt()` i `getopt_long()`, koje se koriste za analiziranje argumenata komandne linije.

Funkcija `getopt()` analizira kratke opcije (opcije od jednog slova) i vraća torku od dva elementa:

- prvi element je lista parova (opcija, vrijednost);
- drugi element je lista argumenata preostalih nakon uklanjanja opcija.

Funkcija `getopt_long()`, sa druge strane, analizira duge opcije (opcije sa više slova) i vraća torku od dva elementa:

- prvi element je lista parova (opcija, vrijednost);
- drugi element je lista argumenata koja preostaje nakon uklanjanja opcija.

Kratke opcije stoje u formatu stringa slova koje odgovaraju opcijama. U slučaju kratkih opcija koje zahtijevaju unos vrijednosti, u nastavku stoji “:”. Npr. ako imamo “hi:o:” koja odgovara kratkim nazivima opcija, to znači da su uz argumente `-i` i `-o` obavezne vrijednosti, dok argument `-h` može da stoji samostalno.

Duge opcije stoje u obliku liste naziva opcija (zapisane pomoću stringa). U slučaju dugih opcija koje zahtijevaju unos vrijednosti, na kraju svake opcije stoji “=”. U nastavku slijedi primjer korišćenja modula `getopt`.

```
import getopt
import sys

def main(argv):
    inputfile = outputfile = ''
    try:
        opts, args = getopt.getopt(argv, "hi:o:", ["ifile=", "ofile="])
    except getopt.GetoptError:
        print('test.py -i <inputfile> -o <outputfile>')
        sys.exit(2)
    for opt, arg in opts:
```

```

    if opt == '-h':
        print('test.py -i <inputfile> -o <outputfile>')
        sys.exit()
    elif opt in ("-i", "--ifile"):
        inputfile = arg
    elif opt in ("-o", "--ofile"):
        outputfile = arg

print('Input file is ', inputfile)
print('Output file is ', outputfile)

if __name__ == "__main__":
    main(sys.argv[1:])

```

U prethodnom primjeru, skripta uzima dva argumenta, *-i* (ili duže - *-ifile*) za ulazni fajl, te *-o* (ili duže - *-ofile*) za izlazni fajl. U slučaju da korisnik ne proslijedi (obavezne) argumente, skripta će ispisati poruku o greški – aktiviraće se izuzetak `GetoptError` koji pripada modulu `getopt`.

Da bi pokrenuli skriptu (naziva `test.py`), na komandnoj liniji unosimo sljedeći kod:

```
python3 test.py -i ulaz.txt -o izlaz.txt
```

Nakon pokretanja skripte, prikazaće se sljedeće:

```

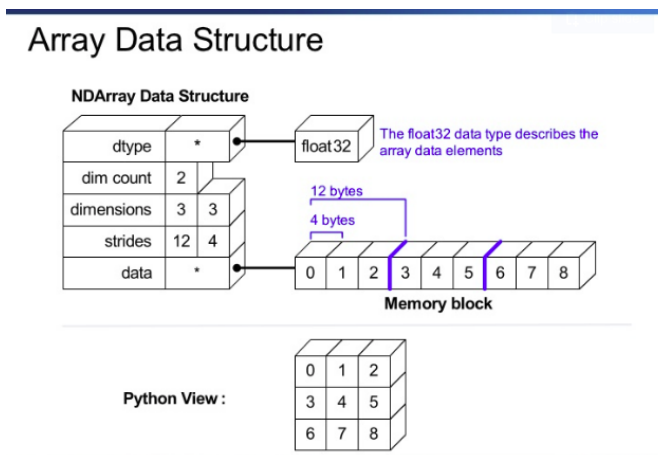
Input file is " ulaz.txt
Output file is " izlaz.txt

```

2.15 Modul Numpy

Ovaj modul je specijalizovan za rad sa nizovima i nizovnim strukturama kao što su matrice, 3D matrice, proizvoljne matrice, velikih dimenzija. Riječ `numpy` predstavlja skraćenicu engleske fraze *numerical python*. Kreirana je 2005. od strane Trvisa Oliphanta, a napisana je u jezicima C/C++. Jedan `numpy` objekat predstavlja neprekidan (statičan) dio u memoriji, a za razliku od liste, ovaj objekat je homogene strukture, vidjeti Sliku 2.11. Zbog svega toga, pogodan je za rad sa velikim tipovima podataka, i brži je za red velicine (oko 50 puta) od tradicionalne liste.

Inicijalizacija objekta.



Slika 2.11: Struktura podataka numpy niza.

```
import numpy as np
ar = np.array([2, 4, 1, 10], dtype="int32") # niz
matrica = np.array([[2, 5 ], [1, 7 ]], dtype="int32") # matrica (2D-niz)
matrica3D = np.array( [[[2, 5 ], [1, 7 ]], [[2, 3], [5, 5]]], \
dtype="int32") # 3D matrica
```

Što se tiče eksplicitno definisanog tipa elemenata objekta numpy (argument *dtype*) mogući su sljedeći tipovi: *i*: integer (i4, i2, itd.), *b*: boolean, *u*: unsigned integer, *f*: float, *M*: datetime, *O*: object, *S*: string.

Osnovne metode.

- Vraćanje elementa na određenoj poziciji (indeksu):
 - 1D niz: [indeks]
 - 2D niz: [indeks1, indeks2]
 - 3D niz: [indeks1, indeks2, indeks3]
- *shape()*: vraćanje dimenzije niza;
- Izrezivanje (*slicing*): [start:end:step]: slično kao i kod listi;
- Kopiranje niza: funkcija *copy()* koja se primjenjuje na objekat koji se kopira (tačka notacija);

- `reshape()`: promjena dimenzije niza, potrebna je kompatibilnost broja elemenata početnog niza sa izmijenjenom dimenzijom;
- `np.concatenate((arr1, arr2), axis =.)`: spajanje dva numpy objekta po x ili y osi (argument `axis` ili 0 ili 1)
- `np.array_split(arr, n, axis)`: dijeljenje niza `arr` na više manjih nizova, gdje je n broj nizova na koje dijelimo niz, dok je `axis` osa po kojoj želimo da vršimo dijeljenje;
- `where(uslov)`: vraća niz indeksa onih elemenata niza koji se uklapaju u `uslov`; npr. uslov može biti: `arr == 4` ili `arr % 2 == 0`;
- `np.sort(arr)`: sortiranje niza `arr`;
- Filtriranje elemenata u nizu se može izvesti na sljedeći način

```
arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]
```

- `numpy.zeros(dim)`, `numpy.ones(dim)`: kreiranje matrice dimenzije `dim` sa svim nulama, odnosno jedinicama.

Memorijska organizacija. Slika 2.11 daje pojednostavljeni prikaz memorijske organizacije numpy objekta u jeziku Pajton. Ovaj objekat pripada klasi `ndarray`, koja ima nekoliko bitnih atributa:

- `dtype`: definiše tip podataka elemenata koji se smještaju u strukturi.
- `dim_count`: definiše dimenzionalnost objekta (2D, 3D, isl.).
- `dimensions`: definiše dimenzije po svakoj od koordinata.
- `strides`: torka koja definiše broj bajtova koje treba pomaknuti u memoriji da bi se pristupilo sljedećem elementu u nizu duž određene dimenzije. Npr. za 2D niz, `stride` torka je definisana sa (`bajt_za_pomicanje_u_smjeru_reda`, `bajt_za_pomicanje_u_smjeru_kolone`).
- `data`: pokazivač na sadržaj niza.

2.16 Modul Ctypes

Modul *ctypes* pruža način za pozivanje funkcija i korištenje tipova podataka u dijeljenim bibliotekama ili bibliotekama dinamičkog povezivanja (DLL) napisanim u jeziku *C* ili drugim jezicima nižeg nivoa unutar pajtonovog koda. Osnovni koraci za korištenje *ctypes* modula za uključivanje *C* programa u pajtonov kod su dati u nastavku.

1. Prvo napisati *C* program koji želimo ugraditi u pajtonov kôd. Ovaj program može da sadrži jednu ili više funkcija koje želimo pozvati unutar kôda pisanog u Pajtonu.
2. Prevesti (kompajlirati) *C* program u dijeljenu biblioteku. Na Linuxu ili macOS sistemima u tu svrhu koristimo *gcc* kompajler. Na sistemu Windows možete koristiti Visual Studio ili MinGW za kompajliranje programa u DLL fajl.
3. Učitati dijeljenu biblioteku u pajtonov kôd. Ovaj proces vršimo uz pomoć modula **ctypes**. Učitavamo .dll fajl pozivanjem funkcije *cdll.LoadLibrary(path)* gdje je *path* proslijeđena staza do dijeljene (dll) biblioteke.
4. Definišimo tipove argumenata i ono što vraćaju funkcije napisane u *C* programu. Prije nego što pozovemo *C* funkciju iz pajton programa pomoću modula **ctypes**, trebaju se navesti tipovi argumenata i tipovi koje funkcije vraćaju. To možemo učiniti postavljanjem *argtypes* i *restype* atributa funkcijskog objekta.
5. Pozivanje *C* funkcije u pajtonovom programu: Na kraju, pozivamo *C* funkciju iz pajtona pomoću funkcijskog objekta koji je kreiran u koraku 4. Vrijednosti argumenata proslijeđujemo kao pajtonove objekte, a modul **ctypes** će ih automatski pretvoriti u odgovarajuće *C* tipove podataka.

U nastavku navodimo čitav proces poziva *C* programa izvršavanjem pajtonovog koda.

```
// dodaj.c
int saberi(int a, int b) {
    return a + b;
}
```

Kompajlirajmo *C* kôd pomoću *gcc* kompajlera:

```
gcc -shared -o lib_dodaj.so dodaj.c
```

Dalje, učitavamo kreirani dijeljeni fajl u pajtonov program pomoću *ctypes* modula:

```
import ctypes
lib_dodaj = ctypes.cdll.LoadLibrary('./lib_dodaj.so')
```

Potom, eksplicitno definišemo tipove argumenata i tip koji vraća odabrana *C* funkcija:

```
saberi_func = lib_dodaj.saberi
saberi_func.argtypes = [ctypes.c_int, ctypes.c_int]
saberi_func.restype = ctypes.c_int
```

Konačno, pozovemo odabranu *C* funkciju uz pomoć kôda:

```
result = saberi_func(2, 3)
print(result)    # Output: 5
```

2.17 Pisanje prilagodjenih modula

Pretpostavimo da želimo kreirati modul koji služi za različite matematičke operacije, kao što su sabiranje, oduzimanje, množenje i dijeljenje. Kako smo pomenuli, modul je skup objekata koji se čuva u datoteku ekstenzije *.py*. Pretpostavimo da želimo kreirati modul *mathOps*. Dakle, kreirajmo datoteku pod nazivom “*mathOps.py*” i tamo definišemo funkcije ovog prilagodjenog modula. Evo primjera kako bi ta datoteka mogla izgledati:

```
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y
```

```
def divide(x, y):  
    if y == 0:  
        raise ValueError("Ne možemo dijeliti sa 0")  
    return x / y
```

Ovaj primjer definiše četiri različite funkcije, od kojih svaka izvodi različite matematičke operacije. Funkcija *add* uzima vrijednost dva argumenta, x i y , i vraća njihov zbir. Funkcija *subtract* vraća razliku dva broja koji se prenose kao argumenti funkcije. Funkcija *multiply* vraća proizvod dva broja. Konačno, funkcija *divide* uzima dva argumenta, i vraća rezultat njihovog dijeljenja, ali takođe javlja grešku ako je nazivnik (vrijednost argumenta y) nula.

Korištenje ovog prilagodjenog modula u drugoj pajtonovoj datoteci se vrši importovanjem modula *mathOps* na sljedeći način:

```
import mathOps  
  
result = mathOps.add(3, 4)  
print(result)    # Output: 7  
  
result = mathOps.divide(10, 2)  
print(result)    # Output: 5.0  
  
result = mathOps.divide(10, 0)  
# Output: ValueError: Ne možemo dijeliti sa 0
```

U ovom primjeru uvozimo modul “*mathOps*” i koristimo njegove funkcije za izvođenje matematičkih operacija. Pozivamo funkciju *add* koja vraća zbir brojeva 3 i 4, a to je 7. Zatim pozivamo funkciju *divide* za dijeljenje 10 sa 2, koja vraća 5,0. Na kraju, ponovno pozivamo funkciju *divide* da bismo podijelili 10 sa 0, što uzrokuje *ValueError* jer dijeljenje sa nulom nije dopušteno.

2.18 Postavljanje korisnički kreirani modul na PyPI repozitorij

PyPI (*Python Package Index*) je centralni repozitorij paketa koji su pisani u programskom jeziku Pajton. PyPI omogućuje programerima lako di-

2.18. POSTAVLJANJE KORISNIČKI KREIRANI MODUL NA PYPI REPOZITORIJ89

stribuiranje korisnički napisanih modula, omogućavajući da dati paket bude korišten široj zajednici korisnika.

Nekoliko koraka treba izvršiti ukoliko želimo postaviti korisnički kreirani modul na PyPI repozitorij. Na zvaničnoj PyPI veb stranici, na linku <https://pypi.org/account/register/>, potrebno je kreirati račun. Nakon što se prijavimo, potrebno je generisati token za prijavu na PyPI. Token je sigurnosni ključ koji omogućuje da se prijavimo i objavimo paket (modul).

Na lokalnom računaru instalirajmo module **setuptools** i **wheel**, u slučaju da nisu instalirani. To se možete učiniti pomoću *pip* naredbe:

```
pip install setuptools wheel
```

Modul **setuptools** omogućuje da upravljamo i distribuiramo naš modul, a modul **wheel** omogućuje generisanje datoteka sa ekstenzijom *.whl* koje se mogu lako instalirati.

Potom, u korijenom direktorijumu našeg modula kreiramo fajl *setup.py*. Fajl *setup.py* definiše informacije o našem modulu kao što su naziv, verzija, autor, opis i druge metapodatke koji će biti prikazani na PyPI stranici modula. Primjer takvog fajla je dat u nastavku.

```
from setuptools import setup

setup(
    name='modul1',
    version='0.0.1',
    author='Mirko Mirkovic',
    description='modul1',
    packages=['modul1'],
    install_requires=[
        'numpy',
        'pandas'
    ]
)
```

Ovaj *setup.py* definiše paket *ex_package* sa verzijom 0.0.1, datim autorom i opisom, a zavisi od modula *numpy* i *pandas*.

Naredni korak je kreiranje distribucijske datoteke (source distribution i wheel distribution) pomoću **setuptools**, izvršavajući naredbu:

```
python setup.py sdist bdist_wheel
```

Dalje instaliramo **twine** paket, koji će pomoći da postavimo naš distribucijski modul na PyPI repozitorijum. Instaliramo ga pomoću naredbe:

```
pip install twine
```

Potom postavimo korisnički modul na PyPI pomoću twine naredbe:

```
twine upload dist/*
```

Izvršavanje prethodne naredba će tražiti da se unese PyPI korisničko ime i lozinka, a zatim će prenijeti paket na PyPI (direktorijum dist).

Nakon objave modula na PyPI, možemo ga instalirati pomoću naredbe:

```
pip install modul1
```

Zadaci

1. Implementirajte igru pogađanja u kojoj korisnik treba da pogodi skriveni broj. Nakon pogađanja, program izvještava korisnika da li je njegov broj velik ili mali (ili je pogođen). Na kraju ispisati broj pokušaja iz kojeg je korisnik pogodio skriveni broj. Računa se jedan pokušaj ukoliko se unese isti broj više puta uzastopno.
2. Napisati program koji računa sljedeću sumu

$$4 \cdot \sum_{k=1}^{10^6} (-1)^{k-1} \frac{1}{2k-1} = 4 \cdot (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 \dots).$$

3. Napisati funkciju koja kombinira dvije liste naizmjenično uzimajući naizmjenično elemente jedne pa druge liste, Npr. za $[a, b, c]$, $[5, 6, 7]$ generišemo listu $[a, 5, b, 6, c, 7]$.
4. Napisati funkciju koja rotira listu za k elemenata. Npr. lista $[1, 2, 3, 4, 5, 6]$ rotirano za $k = 2$ je data sa $[3, 4, 5, 6, 1, 2]$. Riješiti zadatak bez kreiranja kopije liste.
5. Napisati funkciju koja uzima broj i vraća listu njegovih brojeva. Npr. za broj 2342 treba vratiti listu $[2, 3, 4, 2]$.

2.18. POSTAVLJANJE KORISNIČKI KREIRANI MODUL NA PYPI REPOZITORIJI91

6. Napisati funkciju koja uzima listu brojeva, bazu b_1 u kojoj se brojevi liste interpretiraju, te ciljnu bazu b_2 . Kao rezultat vratiti listu brojeva konvertujući svaki od brojeva početne liste u bazu b_2 . Npr. za listu $[21, 10, 01]$ u bazi $b_1 = 3$ pretvaramo u listu brojeva po bazi $b_2 = 10$, dobijajući $[7, 3, 1]$.
7. Napišite funkciju koja množi dvije matrice.
8. Napisati regularni izraz koji prepoznaje registarske tablice. Testirati primjenu uz pomoć modula `re`.
9. Napisati regularni izraz koji prepoznaje imejlove. Testirati primjenu uz pomoć modula `re`.
10. Napisati regularni izraz koji prepoznaje realne brojeve. Testirati primjenu uz pomoć modula `re`.
11. Kreirati datoteku `apsolventi.txt` i popuniti je podacima gdje je u svakom redu data informacija o studentu apsolvantu u formi:

`< Ime_prezime >, < datum_upisa_na_fakultet >, < godine_starosti >`

Datum se zadaje u formi `< broj > . < broj > . < broj >`. Napisati skriptu koja se pokreće sa terminala čiji su argumenti komandne linije:

- `i`: koja zadaje ulazni datoteku (sve zajedno sa putanjom) koja treba da bude učitana u program;
- `o`: koja zadaje izlaznu datoteku (sve zajedno sa putanjom) u kojoj se čuvaju rezultati izvršavanja programa;
- `z`: koji zadaje zadatak koji treba da bude izvršen i obuhvata sljedeći opseg vrijednosti:
 - 1: U izlaznoj datoteci treba da se ispišu svi apsolvanti stariji od 25 godina.
 - 2: U izlaznoj datoteci treba da se ispišu svi apsolvanti koji su upisali fakultet u periodu od 01.01.2014. do 01.01.2015. godine.
 - 3: U izlaznoj datoteci treba da se ispiše sortirani niz (koristeći `merge sort`) studenata po godini života.

- 4: U izlaznoj datoteci treba da se ispiše za svakog studenta koliko godina studira.
- za ostale inpute javiti poruku o neispravnoj opciji (unosu).

Napomena. Studenta učitati uz pomoć regularnih izraza. Dakle, prazni karakter je invarijantna vrijednost što se postiže konstrukcijom odgovarajućeg regularnog izraza pri čitanju redova u datoteci.

12. Kreirati datoteku pretplatnik.txt i popuniti je podacima gdje je u svakom redu data informacija o studentu u formi

< Ime_prezime >, < datum_ugovora >, < cijena_usluge >

Datum se zadaje u formi *< broj > . < broj > . < broj >*, a cijena je decimalni broj. Napisati skriptu koja se pokreće sa terminala sa sljedećim argumentima komandne linije:

- *i*: koja zadaje ulazni datoteku (sve zajedno sa putanjom) koja treba da bude učitana u program;
- *o*: koja zadaje izlaznu datoteku (sve zajedno sa putanjom) koja treba da sačuva rezultate izvršavanja programa;
- *z*: koji zadaje zadatak koji treba da bude izvršen i obuhvata sljedeće vrijednosti:
 - 1: U izlaznoj datoteci treba da se ispišu svi korisnici koji koriste usluge čija je cijena veća od 15,4 KM. Eliminirati duplikate.
 - 2: U izlaznoj datoteci treba da se ispišu svi pretplatnici koji su potpisali ugovor u periodu od 01.01.2022. do 15.01.2023. godine.
 - 3: U izlaznoj datoteci treba da se ispiše zbir svih cijena za usluge koje koristi svaki od pretplatnika
 - 4: U izlaznoj datoteci treba da se ispišu sortirane cijene od najmanje do najveće
 - za ostale inpute javiti poruku o neispravnoj opciji (unosu).

2.18. POSTAVLJANJE KORISNIČKI KREIRANI MODUL NA PYPI REPOZITORIJ93

Napomena. Pretplatnika učitati uz pomoć regularnih izraza. Npr., smatra se da je unos Marko Markovic, 20.2.2022, 8, 4 isti kao Marko Markovic, 20. 2.2022, 8, 4, itd. Dakle, prazni karakter je invarijantna vrijednost što se postiže konstrukcijom odgovarajućeg regularnog izraza pri čitanju redova u datoteci.

Glava 3

Algoritamske paradigme

Programske paradigme predstavljaju različite pristupe ili stilove programiranja koji definišu način na koji programeri dizajniraju, strukturiraju i organizuju svoj kôd za rješavanje problema. Postoji nekoliko programskih paradigmi, a svaka od njih zahtjeva drugačiji način razmišljanja o problemima i rješenjima programiranja.

Neke od najčešćih programskih paradigmi su date u nastavku.

- *Imperativno programiranje*: Ova paradigma uključuje eksplicitni slijed naredbi koje računar izvršava određenim redoslijedom.
- *Funkcijsko programiranje*: Ova paradigma se fokusira na korištenje funkcija za rješavanje problema.
- *Logičko programiranje*: Ova paradigma uključuje programiranje zasnovano na logici i zaključivanju. Posebno je koristan za probleme koji uključuju pravila i ograničenja.
- *Objektno orijentirano programiranje*: Ova paradigma uključuje kreiranje objekata koji sadrže podatke i metode/funkcije koje rade na tim podacima (mijenjajući stanje objekta).

Različiti programski jezici često podržavaju jednu ili više paradigmi, a neki jezici dozvoljavaju programerima da po potrebi kombinuju više različitih paradigmi. Izbor paradigme programiranja često zavisi od domena problema, složenosti problema i ličnih preferencija i iskustva programera.

Kada je riječ o *algoritamskim paradigmama*, one predstavljaju pristupe ili metode za dizajniranje algoritama koji rješavaju računske probleme. Postoji nekoliko algoritamskih paradigmi; svaka ima svoje prednosti i slabosti.

Evo nekih od najčešćih algoritamskih paradigmi su date u nastavku pomoću kratkog opisa.

- *Rekurzivni algoritmi* – uključuje rješavanje problema dijeljenjem na manje (pod)probleme, koji se potom rekurzivno rješavaju. Prekid rekurzije se dešava dolaskom do baznih slučajeva koji rješavaju trivijalne (ili skoro trivijalne) podprobleme.
- *Algoritmi grube sile* (eng. *brute force*) – uključuje isprobavanje svakog mogućeg rješenja problema dok se ne pronađe ispravno. Obično se upotrebljava samo za male probleme zbog svoje (ne)efikasnosti.
- *Podijeli i vladaj* (eng. *divide-and-conquer*) – uključuje podjelu problema na manje podprobleme, rješavanje svakog podproblema nezavisno, a zatim se rezultati kombinuju da bi se riješio originalni problem. Dakle, koristi se za efikasno rješavanje problema koji se mogu podijeliti na manje, nezavisne dijelove.
- *Pohlepni algoritmi* (eng. *greedy*) – uključuje donošenje lokalno optimalnog izbora u svakom koraku algoritma u nadi da će to dovesti do globalno optimalnog rješenja. Ova paradigma je često korištena za rješavanje problema optimizacije.
- *Algoritmi dinamičkog programiranja* (eng. *dynamic programming*) – uključuje razbijanje problema na manje podprobleme i rješavanje svakog podproblema samo jednom. Kada se podproblem riješi, rezultati podproblema se pohranjuju u odgovarajuću (memorijsku) strukturu. Često se koristi za probleme sa podproblemima koji se preklapaju.
- *Algoritmi vraćanja unazad* (eng. *backtracking*) – uključuje istraživanje svih mogućih rješenja problema postepenom izgradnjom i napuštanjem parcijalnog rješenja ako se utvrdi da je netačno (nedopustivo). Često se koristi za probleme koji se mogu predstaviti putem grafa (stanja).
- *Randomizirani algoritmi* (eng. *randomized*) – uključuje uvođenje slučajnosti u algoritam radi poboljšanja performansi ili smanjenja složenosti. Često se koristi za probleme optimizacije ili probleme koji imaju mnogo mogućih rješenja.

Različite algoritamske paradigme mogu se prilagoditi ili međusobno kombinovati za efikasnije rješavanje složenih problema.

3.1 Rekurzivni algoritmi

Riječ rekurzija potiče od latinske riječi *recurrere*, što doslovno znači vraćanje. U programiranju, funkcija koja u tijelu poziva samu sebe se naziva rekurzivna. Pozivanje iste funkcije unutar same funkcije podrazumijeva sistematično smanjivanje veličine ulaza funkcije koja se poziva, shodno problemu koji se rješava. Smanjivanje dimenzije ulaza odgovara rješavanju odgovarajućih podproblema početnog problema rekurzivnim putem. Da bi se rekurzija izvršavala u konačnom broju koraka, neophodno je definisati prekidne, tzv. *bazne*, slučajeve rekurzije. Bazni slučajevi su bitni iz razloga što su oni okidač prekida rada rekurzije.

Prepoznavanje pogodne rekurzije igra bitnu ulogu u rješavanju problema rekurzivnim načinom. Demonstrirajmo sada rekurziju i rekurzivni način rješavanja problema na jednostavnom problemu računanja n -tog prirodnog broja. Neka $f(n)$ predstavlja n -ti po redoslijedu parni broj u skupu \mathbb{N} . Prvi broj u ovom skupu je 0, pa definišimo $f(0) = 0$. Svaki sljedeći je za dva veći od prethodnog, što je definisano rekurzijom:

$$f(n) = f(n - 1) + 2, n \geq 1, f(0) = 0.$$

Dakle, da bismo izračunali treći parni broj po redu u skupu \mathbb{N} , imamo sljedeći niz jednakosti:

$$f(3) = f(2) + 2 = f(1) + 2 + 2 = f(0) + 2 + 2 + 2 = 0 + 6 = 6.$$

Dakle, poziv $f(3)$ zahtjeva poziv funkcije f sa vrijednosti 2, koja potom zahtjeva poziv funkcije sa vrijednosti (argumenta) 1, sve do trivijalnog koraka, računanja $f(0)$. Potom se, vraćanjem unazad, računa vrijednost $f(1)$, pa se pomoću nje računa vrijednost za $f(2)$, dok ne dobijemo rješenje za $f(3)$. Napomenimo da se u programskom jeziku Pajton, za svaki poziv funkcije f formira (lokalni) stek, koji nakon izlaza iz funkcije (vraćanjem rezultata) biva obrisani.

Napomenimo da se rekurzivne funkcije (tzv. linearne, homogene) kao u prethodnom primjeru mogu izraziti preko eksplicitne funkcije u odnosu na veličinu ulaza problema, pa rekurzivni pozivi za njihovo računanje i nisu potrebni. Jasno se vidi da je gore $f(n) = 2n$.

Rekurzija u jeziku Pajton prethodnog problema je data u nastavku.

```
def even(n):
    if n==0:
        return 0
    return 2 + even(n-1)

n = input("Unesite broj:")
print(even(n))
```

Kompleksnost rekurzije. Jasno se vidi da se program izvršava u linearnom $O(n)$ vremenu.

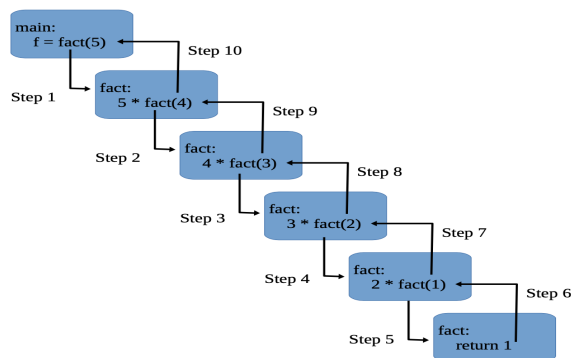
Napišimo sada rekurziju za računanje faktoriijela broja n . Označimo $f(n) = n!$. Iz definicije faktoriijela, imamo:

$$f(n) = n! = n \cdot (n-1) \cdots 2 \cdot 1 = n \cdot (n-1)! = n \cdot f(n-1).$$

Bazni korak rekurzije je $f(1) = 1$. Prema tome, rekurzivna implementacija faktoriijel funkcije izgleda ovako:

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1)
```

Na Slici 3.1, dato je drvo rekurzije prethodnog programa za konkretan slučaj, $n = 5$.



Slika 3.1: Drvo rekurzije.

Kompleksnost rekurzije. Vidi se da je kompleksnost linearna, tj. $O(n)$.

Ako bazni slučaj nije dostignut ili nije definisan, može doći do problema prekoračenja steka (eng. *stack overflow*). Posmatrajmo sljedeći program.

```
def fact(n):  
    # pogrešan bazni slučaj  
    if (n == 100):  
        return 1  
    else:  
        return n*fact(n-1);
```

Ovaj program će izazvati prekoračenje steka u slučaju kada je ulaz $n < 100$; tada bazni slučaj nikada neće biti ispunjen i rekurzija će se pozivati dok god se ne iscrpi memorija steka rezervisana za program.

3.2 Vrste rekurzija

Funkciju `fun` nazivamo *direktnom* rekurzivnom ako ona poziva istu funkciju `fun`. Funkcija `fun` se naziva *indirektnom* rekurzivna ako poziva drugu funkciju koja potom poziva direktno ili indirektno funkciju `fun`. Razlika između direktne i indirektne rekurzije ilustrovana je data u narednom programu.

```
# Direktna rekurzija  
def direktnaRecFunc():  
    # Kod ....  
  
    direktnaRecFunc()  
  
    # Kod...  
  
# Indirektna rekurzija  
def indirektnaRecFun1():  
    # Kod ...  
  
    indirektnaRecFun2()  
  
    # Kod...
```



```
def indirektnaRecFun2():  
  
    # Kod...  
  
    indirektnaRecFun1()  
  
    # Kod...
```

Rekurzivna funkcija se naziva *reptom* rekurzijom ako je rekurzivni poziv posljednji dio koda koju funkcija izvršava. Inače, rekurzivna funkcija se naziva *nereptom*.

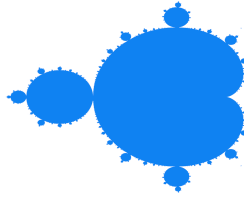
3.2.1 Prednosti i nedostaci rekurzivnog nad iterativnim pristupom

Svaka rekurzija se može napisati na ekvivalentan, iterativni pristup. Iako rekurzija i iteracija u ugnježdenim petljama na prvi pogled djeluju dosta slično postoje značajne, suštinske razlike:

- Rekurzija se prekida kada se dostigne bazni slučaj; iteracija se prekida kada uslov postane netačan.
- Rekurzija se veže za funkcije, dok se iteracija veže za petlje.
- Svaki rekurzivni poziv zahtijeva dodatni prostor koji se alocira u memoriji steka; iteracija ne zahtijeva dodatni memorijski prostor.
- Rekurzija najčešće ima kraći kôd; iteracije u petlji su često obimnijeg kôda.

Rekurzivni program ima veće zahtjeve za prostorom od (ekvivalentnog) iterativnog programa, jer će sve funkcije ostati na steku dok se ne dostigne bazni slučaj. Takođe, zahtjevi za vremenom su veći zbog poziva funkcija i troškova vraćanja rezultata. Dodatno, zbog manje dužine kôda, kodovi znaju biti teški za razumevanje i stoga je potrebno uložiti dodatnu pažnju prilikom pisanja kôda zbog mogućih previda.

Prednost korištenja rekurzija je u čistom i jednostavnom načinu pisanja koda. Neki problemi su inherentno rekurzivni, npr. obilazak stabala, problem



Slika 3.2: Mandelbrotov skup

hanojskih kula, itd. Za takve probleme, poželjno je, a i intuitivnije, pisati rekurzivni kôd. Kako smo već naveli, takve kodove možemo pisati i iterativno uz pomoć strukture podataka steka.

3.2.2 Primjene rekurzivnog pristupa

Rekurzija je moćna tehnika sa raznim primjenama u informatici i programiranju. Neke od uobičajenih primjena rekurzije uključuju:

- *Prelazak stabla i grafa.* Rekurzija se često primjenjuje za prelazak i pretraživanje značajnih struktura podataka kao što su stabla i grafovi. Rekurzivni algoritmi se mogu koristiti za istraživanje svih čvorova ili vrhova stabla ili grafa na sistematičan način.
- *Algoritmi za sortiranje.* Rekurzivni algoritmi se koriste u algoritmima za sortiranje kao što su *quick sort* (brzo sortiranje) i *merge sort* (sortiranje spajanjem). Ovi algoritmi koriste rekurziju da podijele podatke u manje podliste, sortiraju ih, a zatim ponovo spajaju.
- *Generisanje fraktala.* Fraktalni oblici i obrasci mogu se generirati korištenjem rekurzivnih algoritama. Npr. Mandelbrotov skup se generiše uzastopnom primjenom rekurzivne formule na kompleksne brojeve.
- *Algoritmi vraćanja unazad.* Algoritmi vraćanja unazad se koriste za rješavanje problema koji uključuju donošenje niza odluka, gdje svaka odluka zavisi od prethodnih. Ovi algoritmi se mogu implementirati korištenjem rekurzije za istraživanje svih mogućih staza i vraćanje unatrag kada se rješenje ne pronađe.
- *Memoizacija.* Ovo je tehnika koja uključuje pohranjivanje rezultata skupih poziva funkcija i vraćanje keširanih rezultata kada se naiđe

na pozive funkcija istih ulaza. Memoizacija se može implementirati korištenjem rekurzivnih funkcija koje izračunavaju i keširaju rješenja podproblema.

3.2.3 Primjena rekurzivnog pristupa

Pogledajmo naredni zadatak (o rastu populacije zečeva kroz vrijeme).

Primjer 3.1 *Pretpostavimo da se zečevi reprodukuju na sljedeći način: par zečeva se na kraju prvog mjeseca života ne razmnožava. Međutim, na kraju drugog i svakog sljedećeg mjeseca oni reprodukuju novi par zečeva. Postavlja se pitanje koliko će novorođenih parova zečeva biti poslije godinu dana (tj. na kraju 12-og mjeseca)?*

Rješenje. Brojevi parova zečeva čine niz: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Lako zaključavamo da je svaki sljedeći broj u nizu jednak zbiru prethodna dva broja, tj. ako je $f(n)$ broj novorođenih zečeva na kraju n -tog mjeseca, onda je

$$f(n) = f(n-1) + f(n-2), n \geq 2$$

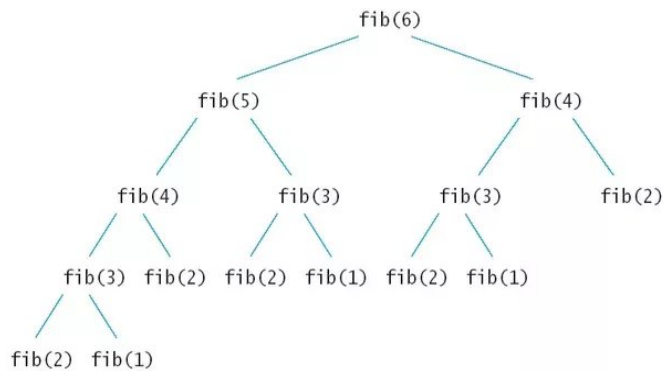
gdje je bazni slučaj dat sa $f(0) = 0, f(1) = 1$.

Ova rekurzija je implementirana sljedećim (Pajton) kodom.

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Dakle, u svakom koraku, ako se ne dođe do trivijalnog koraka, rekurzivni poziv će pozvati dvije nove rekurzije (sa manjim veličinama ulaza). Potom će, vraćanjem rezultata na manjim ulazima, rezultat biti izračunat i za veće ulaze. Rješenje prethodnog zadatka se dobija računajući rekurziju za ulaz $n = 12$.

Kompleksnost algoritma. Primijetimo da se u svakom koraku broj poziva duplira u odnosu na prethodni. Brzina opadanja veličine ulaza opada konstantno u odnosu na n , pa zaključujemo da je kompleksnost izvršavanja rekurzije eksponencijalna, tj. jednaka $O(2^n)$.

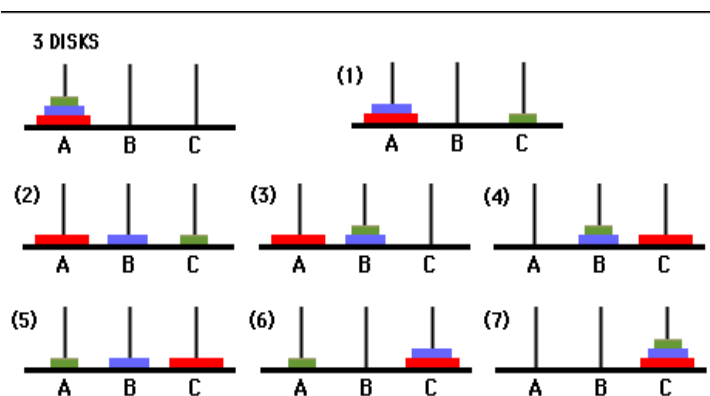
Slika 3.3: Drvo rekurziije za `fib` rekurzivni metod.

Primjer 3.2 Problem Hanojskih kula. *Neka su data tri štapa i n diskova. Zadatak ovog problema je u što manje poteza premjestiti diskove sa jednog štapa na drugi štap. Sljedeća pravila vrijede pri prebacivanju diskova.*

- *Inicijalno, diskovi stoje poredani na jednom štapu (A) jedan na drugom gdje manji diskovi stoje na većim diskovima (pretpostavka je da su diskovi različitih dimenzija).*
- *Potez se sastoji od premještanja jednog diska sa vrha jednog štapa na vrh drugog štapa.*
- *Veći disk ne može nikada da bude stavljen na manji disk u bilo kojoj iteraciji premještanja.*

Rješenje. Situacija sa dva diska i tri štapa je jasna. Pretpostavimo da se dva diska inicijalno nalaze na štapu A i da ih želimo premjestiti na štap C , poštujući uslove zadatka. Označimo ta dva diska crvenom (veći disk) i plavom (manji disk). U prvom korak prebacujemo plavi disk na štap B . Potom crveni disk prebacujemo na štap C . Konačno, plavi disk sa štapa B prebacujemo na štap C , čime je zadatak završen za slučaj sa dva diska ($n = 2$).

Posmatrajmo situaciju sa kulama kao na Slici 3.4 sa tri diska ($n = 3$) i tri štapa (A, B i C).

Slika 3.4: Problem hanojskih kula, $n = 3$ diska.

Prvo se najmanji (zeleni) disk prebacuje na štap C . Drugi korak je prebacivanje plavog (disk srednje veličine) na štap B . Potom, u koraku tri, zeleni disk sa štapa C premiještamo na štap B . Korak četiri je rezervisan za prebacivanje najvećeg diska (sa štapa A) na štap C . Dalje, potom zeleni disk sa B premiještamo na štap A , a plavi na štap C . Konačno, u koraku sedam, zeleni disk sa štapa A prebacujemo na štap C .

Označimo sa $T(n)$ broj koraka koji je potreban da prebacimo n diskova sa štapa A na štap C . Bazni slučaj je $T(1) = 1$. Izvedimo rekursivni korak za ovaj problem. Kako smo pomenuli, ideja rekurzije je svesti početni problem na rješavanje problema manjih dimenzija (podprobleme) koje ponovo rekursivno rješavamo. U ovom slučaju, “hvatamo” se za činjenicu da ukoliko je najveći disk sam na jednom štapu, ostalih $n - 1$ diskova sa drugog štapa možemo nesmetano prebacivati koristeći sva tri štapa (na veliki disk može da se stavi bilo koji od $n - 1$ diskova). Rekursivno prebacimo $n - 1$ diskova sa vrha štapa A na štap C , koristeći štap B . Nakon toga, situacija je sljedeća: (i) najveći disk se nalazi na štapu A ; (ii) ostalih $n - 1$ diskova su složeni na štapu B ; (iii) štap C je prazan. Sljedeći korak je prebacivanje najvećeg diska sa štapa B na štap C . Nakon toga, ponovo rekursivno prebacujemo $n - 1$ diskova sa štapa B na štap C , koristeći štap A . Prema tome, vrijedi rekurzija:

$$T(n) = 2 \cdot T(n - 1) + 1, T(1) = 1.$$

Implementacija rekursivnog pristupa problema Hanojskih kula je data sljedećim kôdom.

```

def TowerOfHanoi(n , izvor, cilj, pomocni):
    if n==1:
        print ("Pomaknite disk 1 sa štapa ", izvor," \
na štap ", cilj)
        return
    TowerOfHanoi(n-1, izvor, pomocni, cilj)
    print ("Pomakni disk ",n," sa štapa ", izvor," na štap \
", cilj) #pomijeranje najvećeg štapa
    TowerOfHanoi(n-1, pomocni, cilj, izvor)

#instanca:
n = 4 #broj diskova
TowerOfHanoi(n, 'A','B','C')

```

Vremenska kompleksnost. Vrlo lako dobijamo eksplicitnu funkciju koja odgovara rekurzivnoj relaciji problema: $T(n) = 2T(n-1) + 1 = 2 \cdot (2 \cdot T(n-2) + 1) + 1 = \dots = 2^{n-2} + 2^{n-1} + \dots + 2 + 1 = 2^n - 1$. Prema tome, $T(n) = O(2^n)$, što je eksponencijalno vrijeme izvođenja.

3.3 Algoritmi grube sile

Prva algoritamska tehnika koja se najčešće koristiti u rješavanju je tehnika grube sile. Ovo je algoritamska paradigma sa kojom se većina čitalaca susretala do sada, mada možda nesvjesno da je riječ o baš toj tehnici rješavanja problema.

Prosto rečeno, paradigma grube sile pokušava sva moguća rješenja problema, a zaustavlja se kada pronađe ono koje je stvarno rješenje. U stvarnom životu, primjer algoritma brutalne sile je uključivanje USB kabela. Prvo pokušamo na jedan način, a ako to ne uspije, isprobamo drugi port. Slično, ako imamo veliki broj ključeva, a nismo sigurni koji od njih otključava vrata, možemo jednostavno isprobati svaki ključ za datu bravu dok jedan ne proradi. Upravo je ovo suština algoritamskog dizajna ove paradigme.

Primjer 3.3 *Odličan primjer algoritma grube sile je pronalaženje najbližeg para tačaka u višedimenzionalnom (ralnom) prostoru. Neka su date n tačaka u \mathbb{R}^m , tj. $S = \{\mathbf{x}_i \mid \mathbf{x}_i \in \mathbb{R}^m, i = 1, \dots, n\}$. Naći dvije tačke $x_i, x_j \in S$ koje su međusobno najbliže među svim ostalim parovima tačaka skupa S .*

Rješenje. Da bismo pronašli odgovor grubom silom, izračunamo udaljenost između svakog pojedinačnog para tačaka, a zatim pratimo minimalnu pronađenu udaljenost. Verzija ovog algoritma je data narednim kôdom.

```
import math
def min_dist_par(S):
    min_dist = float('-inf')
    point1 = None
    point2 = None
    for p1 in S:
        for p2 in S:
            if p1 != p2:
                distance = math.dist(p1, p2)
                if distance < min_dist:
                    min_dist = distance
                    point1 = p1
                    point2 = p2
```

Kompleksnost algoritma. Kako je $|S| = n$, lako se primijeti da je broj parova tačaka jednak $O(n^2)$. Dalje, računanje (euklidove) udaljenosti između dvije m -dimenzionalne tačke se izvršava u $O(m)$ vremenu. Prema tome, ukupno vrijeme izvršavanja je $O(n^2) \cdot O(m) = O(n^2 \cdot m) = O(n^2)$, jer je m konstantno.

Primjer 3.4 *Riješimo problem ruksaka uz pomoć algoritma brutalne sile. Ovaj problem je definisan na sljedeći način: Neka je dato n proizvoda i svaki do njih ima svoju težinu w_i i cijenu (vrijednost) $p_i, i = 1, \dots, n$. Ruksak ima kapacitet $C > 0$. Koji od proizvoda staviti u ruksak, tako da cijena bude najveća, ali da kapacitet ruksaka ne bude narušen.*

Rješenje. Posmatrajmo primjer instance problema.

proizvod	težina	cijena
1	2	20
2	5	30
3	10	50
4	5	10

Neka je kapacitet ruksaka $C = 16$.

Enumeracijom prostora mogućih rješenja, dobijamo tabelu.

Rješenja (proizvodi u ruksaku)	Ukupna težina	Ukupna cijena
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	15	80
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	nedopustivo
{1, 2, 4}	12	60
{1, 3, 4}	17	nedopustivo
{2, 3, 4}	20	nedopustivo
{1, 2, 3, 4}	22	nedopustivo

Iz prethodne tabele, lako se vidi da je najbolje odabrati proizvod 2 i 3 i staviti ih u ruksak. Pri tome, njihova cijena je 80. Primijetimo da rješenje koje uzima proizvode 1, 2 i 3 u ruksak narušava njegov kapacitet (težina je 17, što je veće od 16).

Jasno je da je skup rješenja problema ruksaka poistovijećen sa pretraživanjem svih podskupova skupa $[n]$. Prema tome, kompleksnost ovakvog pristupa je eksponencijalna, tj. $O(2^n)$. Pristup je dat sljedećim kodom.

```
best_fitness: float = 0.0

#instance example:
N = 4
w = [2, 5, 10, 5]
p = [20, 30, 50, 10]
C = 16
# helper function
def objective(solution):

    if len(solution) == 0:
        return float('-inf')
    fitness: float = 0.0
    weight: float = 0.0
```



```

        for i in solution:
            fitness += p[i]
            weight += w[i]

    if weight > C:
        return float('-inf')
    else:
        return fitness

# brute force recursion
def knapsack_brute_force(solution):
    global best_fitness

    if objective(solution) > best_fitness:
        best_fitness = objective(solution)

    for i in range(N):
        if len(solution) >= 1:
            if i > max(solution) : # symmetry breaking
                #print(solution + [i])
                knapsack_brute_force(solution + [i])
        else:
            knapsack_brute_force(solution + [i])

knapsack_brute_force([])
print(best_fitness) # 80.0

```

Kako primjećujemo na osn ovu prethodne (rekurzivne) implementacije, svaki od rekurzivnih poziva, kojih je tačno 2^n , poziva funkciju `objective`. Preciznije, kompleksnost pristupa brutalne sile je $O(2^n)$. Napomenimo da postoje dosta efikasnij pristupi za rješavanje problema ruksaka. Među njima, izdvajamo metod dinamičkog programiranja, o kojem će biti više riječi u nastavku.

3.4 Algoritmi pretraživanja

Algoritmi pretraživanja su dizajnirani sa zadatkom provjeravanja da li se element nalazi (ili da se vrati) u strukturi podataka u kojoj je on pohranjen; takođe, da se vrati poruka ako takav element ne postoji.

Ovi algoritmi se generalno dijele u dvije kategorije na osnovu načina pretraživanja:

- *Sekvencijalna pretraga*: U ovom slučaju se lista ili niz prelazi uzastopno i svaki element se provjerava; primjer ovakve pretrage je *linearna pretraga*.
- *Intervalna pretraga*: Ovi algoritmi su dizajnirani za pretraživanje u sortiranim strukturama podataka; mnogo su efikasniji od linearne pretrage jer dijele prostor pretraživanja na dijelove, pokušavajući na taj način eliminisati veliki dio prostora za pretragu svakom iteracijom. Primjer takve pretrage je *binarna pretraga*.

3.4.1 Linearna pretraga

Ovo je sekvencijalna pretraga, koja pretražuje (nizovne) strukture tako što prolazi kroz svaki njene element dok se ne pronađe željeni element, ili se ne dođe do kraja skupa podataka strukture (tj. svi elementi strukture su posjećeni pretragom). Iterativna implementacija linearne pretrage je data sljedećim kodom.

```
def search(arr, x):  
    for i in range(len(arr)):  
        if (arr[i] == x):  
            return i  
    return -1  
  
arr = [2, 3, 1, 11, 10, 20]  
x = 10  
  
res = search(arr, x)  
if(res == -1):  
    print("Element nije prisutan")
```

```

else:
    print("Element je prisutan pod indeksom ", res)

```

Kompleksnost algoritma. Lako se vidi da je kompleksnost linearne pretrage upravo linearna, $O(n)$, gdje je n veličina (nizovne) strukture.

Implementacija rekurzivnog pristupa za linearnu pretragu je dat narednim kôdom.

```

def linear_search(arr, x, n):
    if n >= len(arr):
        return -1

    if arr[n] == x:
        return n

    return linear_search(arr, x, n+1)

#poziv
res = linear_search(arr,x,0)

```

Vremenska složenost je $O(n)$ dok se za rekurzivni pristup izdvaja i pomoćni prostor kompleksnosti $O(n)$, za alokaciju prostora steka tokom izvođenja rekurzivnih poziva.

Linearna pretraga nije efikasna za nizove velikih dimenzija i koristi se samo kada radimo sa malim skupom podataka, te kada želimo da algoritam držimo što jednostavnijim.

3.4.2 Binarna pretraga

Za upotrebu binarnog pretraživanja u bilo kojoj strukturi podataka, struktura podataka mora imati sljedeća svojstva: (i) treba biti sortirana; (ii) pristup bilo kojem elementu strukture podataka se odvija u konstantnom vremenu.

Ideja binarne pretrage se sastoji u eksploatisanju svojstva da je niz sortiran. Pretpostavimo bez smanjenja opštosti da je niz sortiran u rastućem poretku. To znači da u slučaju da se na indeksu i nalazi element veći od traženog elementa x , x se (potencijalno) nalazi lijevo od indeksa i , te indekse

$i + 1, \dots$ isključujemo dalje iz razmatranja. U svakoj iteraciji, biramo indeks i kao središnji element niza, koji se potom poredi sa x . U zavisnosti od rezultata, imamo tri mogućnosti: (i) element na poziciji i je upravo x , čime prekidamo pretragu vraćajući dati indeks; (ii) element na poziciji i je veći od x , čime pretragu usmjeravamo na elemente niza lijevo od pozicije i ; (iii) inače, pretraga se seli na desni dio niza, tj. elemente čiji je indeks veći od i . Za bazne slučajeve uzimamo kada je niz veličine 0 ili 1. U prvom slučaju vraćamo *False*, dok u drugom slučaju vraćamo rezultat poređenja elementa niza sa x .

Implementacija rekurzivnog pristupa binarne pretrage je data sljedećim kôdom.

```
def binary_search(arr, x):

    if len(arr) == 0:
        return False

    if len(arr) == 1:
        return True if arr[0] == x else False

    mid = len(arr) // 2
    if arr[mid] == x:
        return True
    elif arr[mid] < x:
        return binary_search(arr[mid+1:], x)
    else:
        return binary_search(arr[:mid], x)

# pozivanje programa:
arr = [2, 8, 10, 12, 15]
x = 2
pos = binary_search(arr, x)
print(pos)
```

Kompleksnost algoritma. Broj interacija u rekurziji koju zadovoljava binarna pretraga je $T(n) = T(n/2) + c$, gdje je $c = O(1)$. na osnovu Master teoreme, kompleksnost algoritma je jednaka $O(\log n)$.

Binarnu pretragu koristimo u situacijama kada pretražujemo veliki skupa podataka, jer ima vremensku složenost od $O(\log n)$, dakle, mnogo brže od

linearne pretrage. Pri tom skup podataka treba biti sortiran. Bitno je i da podaci u nizu nemaju složenu strukturu ili odnose između sebe, jer to narušava efikasnost pretrage te kompleksnost rješavanja ne mora više ne bude logaritamska.

3.4.3 Pretraga preskakanjem

Kod ove pretrage se takođe podrazumijeva da je niz sortiran. Osnovna ideja je sistematskom provjerom razumnog broja elemenata niza, koristeći preskakanje nekih elemenata umjesto pretraživanja svih elementa ustanoviti uži interval u kojem se dati element x traži.

Pretpostavimo da imamo niz niz veličine n i dužinu bloka (elemenata) veličine m koji koristimo kao korak preskakanja. Pretraga prvo pretražuje elemente $niz[0], niz[m], niz[2m] \dots niz[k \cdot m]$ i tako dalje. Kada pronađemo interval (indekse) sa svojstvom $niz[k \cdot m] < x < niz[(k + 1) \cdot m]$, izvodimo operaciju linearne pretrage elementa x na intervalu dužine m .

Razmotrimo sljedeći $niz = [0, 1, 1, 2, 3, 5, 8, 12, 24, 37, 55, 82, 140, 433, 567, 622]$. Preskakanjem pronalazimo vrijednost $x = 55$, za veličinu koraka preskakanja $m = 4$. Sljedeće iteracije se izvršavaju.

- Skok sa indeksa 0 na indeks 4;
- Skok sa indeksa 4 na indeks 8;
- Skok sa indeksa 8 na indeks 12;
- Pošto je element na indeksu 12 veći od 55, skočit ćemo korak unazad da bismo došli do indeksa 8;
- Izvršite linearnu pretragu iz indeksa 8 da bismo našli element 55.

Kompleksnost algoritma. U najgorem slučaju, moramo uraditi n/m skokova, a ako je posljednja provjerena vrijednost veća od elementa koji se traži, vršimo $m - 1$ poređenja više za linearnu pretragu. Stoga je ukupan broj poređenja u najgorem slučaju jednak $((n/m) + m - 1)$. Vrijednost ove funkcije je minimalna kada je $m = \sqrt{n}$, što i predstavlja najbolju vrijednost za preskakanje. Dakle, u najboljem slučaju, kompleksnost algoritma je $O(\sqrt{n})$.

Relurzivna implementacija pretage preskakanjem je data sljedećim kodom u jeziku Pajton.

```

def jump_search(niz, x, jump, k):

    if k * jump >= len(niz): #empty array
        return False

    if k * jump < len(niz) and (k+1) * jump >= len(niz):
        found = linear_search(niz[k*jump:], x, 0)
        return found
    #rekurzivni korak: oba kraja intervala su elementi niza:
    if niz[ k * jump ] <= x and x <= niz[ (k+1) * jump ]:
        found = linear_search(niz[k*jump: ((k+1)*jump + 1)], x, 0)
        return found
    else:
        return jump_search(niz, x, jump, k+1) # idemo na naredni interval

#instanca:
niz = [2, 10, 21, 33, 38, 41, 45, 52, 57, 70, 75]
jump_step = 3
x= 57
# poziv metoda:
found = jump_search(niz, 57, jump_step, 0)
print(found)

```

3.4.4 Interpolacijska pretraga

Interpolacijska pretraga je poboljšanje binarnog pretraživanja gdje pretpostavljamo da su vrijednosti u sortiranom nizu uniformno (ravnomjerno) raspoređene. Interpolacija konstruiše nove tačke unutar opsega diskretnog skupa strukture podataka. Kao što znamo, binarno pretraživanje uvijek polazi sa provjerom srednjeg elementa niza. Sa druge strane, interpolacijska pretraga može tražiti na različitim mjestima u strukturi. Npr. ako je vrijednost traženog elementa bliža posljednjem elementu, interpolacijska pretraga će vjerovatno započeti pretragu više prema krajnjoj strani strukture.

Za pronalazak pozicije *pos* u nizu *niz*, koristi se sljedeća ideja: vraćamo višu vrijednost *pos* ako je element koji se traži bliži kraju niza, tj. *niz[n - 1]*. Manja vrijednost za *pos* se vraća kada je vrijednost bliže početku niza, tj. elementu *niz[0]*.

Poziciju pos računamo po sljedećoj formuli:

$$pos = \left\lfloor \frac{(n-1)}{niz[n-1] - niz[0]} \cdot (x - niz[0]) \right\rfloor \quad (3.1)$$

Algoritam se sastoji od sljedećih koraka:

1. Izračunamo vrijednost pos koristeći formulu (3.1).
2. Ako se vrijednost podudara sa traženom, vraćamo indeks/vrijednost *True* i izlazimo iz rekurzije.
3. Ako je traženi element manji od $niz[pos]$, interpoliramo poziciju pos lijevog podniza. U suprotnom, interpoliramo poziciju pos desnog podniza.
4. Ponavljati korake 1-3, sve dok se ne pronađe podudaranje ili dok podniz ne postane prazan.

Implementacija rekurzivnog pristupa interpolacijske pretrage je data narednim kôdom.

```
def interpolation_search(niz, l, r, x):

    if (lo <= hi and x >= niz[l] and x <= niz[r]):
        pos = l + ((r - l) // (niz[r] - niz[l])) * \
            (x - niz[l])

        if niz[pos] == x:
            return pos

        #desni podniz
        if niz[pos] < x:
            return interpolation_search(niz, pos + 1, r, x)

        #lijevi podniz
        if niz[pos] > x:
            return interpolation_search(niz, l, pos - 1, x)

    return -1
```

Kompleksnost algoritma. Algoritam u najgorem slučaju postiže vrijeme od $O(n)$. Može se pokazati da je očekivano vrijeme izvršavanja ovog algoritma mnogo bolje, i iznosi $O(\log \log n)$.

3.4.5 Eksponencijalna pretraga

Naziv ovog algoritma potiče iz načina biranja sljedećeg elementa u pretrazi.

Eksponencijalna pretraga uključuje dva koraka. Prvo, pronadimo interval u kojem se element potencijalno nalazi (ako se nalazi) koristeći eksponencijalne korake. Potom, na datom intervalu primijenimo binarnu pretragu. Za poziciju *pos* uzimamo 0, 2, 4, ... dok ne nađemo indeks *i* tako da je traženi element *x* veći ili jednak od elementa na poziciji 2^{i-1} a manji ili jednak od elementa na poziciji 2^i niza koji se pretražuje.

Implementacija ove pretrage je data narednim kodom koji je napisan u jeziku pajton.

```
def exponential_search(niz, x, l):  
  
    if len(niz) <= 1:  
        if x in niz:  
            return l + int(x in niz)  
        else:  
            return -1  
  
    if l == 0:  
        pos = 2  
    else:  
        pos = l * 2  
  
    if pos >= len(niz):  
        ind_found = binary_search(niz[l:], x)  
        # ako x ne postoji u sufiksu  
        if ind_found == -1:  
            return -1  
        else:  
            return l + binary_search(niz[l:], x)  
    else:  
        if niz[l] <= x and x <= niz[pos] :  
            return l + binary_search(niz[l: pos+1], x)  
        else:  
            return exponential_search(niz, x, pos)
```



```
# instance:
niz = [2, 10, 22, 38, 44, 51, 100, 220, 550, 880]
x = 10
# poziv funkcije:
print(exponential_search(niz, x, 0))
```

Kompleksnost algoritma. Ovaj algoritam pripada klasi $O(\log n)$.

Eksponencijalna pretraga je posebno korisna za pretraživanja u kojima je veličina niza ogromna. Ova pretraga se pokazala efikasnijom od binarnog pretraživanja za ograničene nizove (maksimalni element je unaprijed ograničen nekom konstantom), a takođe i kada je element koji se traži bliži prvom elementu.

3.5 Algoritamska paradigma podijeli pa zavlada

Ideja ovog (rekurzivnog) pristupa je sljedeća:

- Podijelimo problem na nezavisne podprobleme efikasno;
- Riješimo podprobleme rekurzivno;
- Kombinujemo rješenja podproblema na efikasan način, tako da rezultat ostane validan i za veći podproblem.

Pseudokod date paradigme je dat Algoritmom 5.

Dobro poznati primjer gdje se ova algoritamska paradigma primjenjuje je sortiranje spajanjem, gdje se sortiranje n brojeva odvija u $n \log n$ vremenu.

Ideja sortiranja niza je sljedeća:

- Podijeliti niz na dva (podjednaka), disjunktna, dijela – lijevi i desni;
- Sortirati rekurzivno oba dijela niza;
- Spojiti dva sortirana niza u linearnom vremenu.

Izvorni kod za sortiranje spajanjem u programskom jeziku Pajton je dat u nastavku.

Algoritam 5 Pseudokod algoritamske paradigme Zavadi pa vladaj.

```

1: procedure D&C( $n$ : veličina ulaza)
2:   if  $n \leq n_0$  then
3:     Riješiti problem bez dodatnog dijeljenja
4:   else
5:     Podijeliti problem na  $r$  podproblema svaki veličine  $\approx n/k$ ;
6:     for svaki od  $r$  podproblema do
7:       Pozovemo D&C ( $n/k$ )
8:     end for
9:     Kombinujemo  $r$  rezultujućih rješenja podproblema da bi se dobilo
       rješenje početnog problema
10:   end if
11: end procedure

```

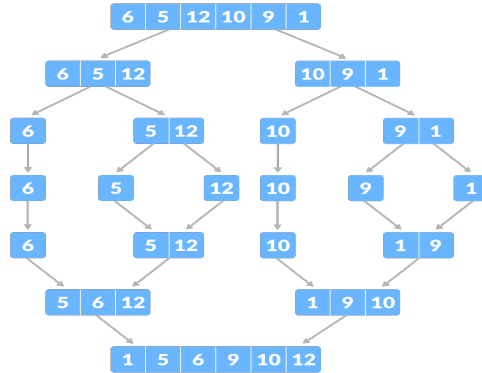
```

def mergeSort(niz):

    if len(niz) <= 1:
        return niz
    else:
        ##partition:
        nizLeftSort = niz[0 : (len(niz)//2) ]
        nizRightSort = niz[ (len(niz) // 2) :]
        mergeSort(nizLeftSort)
        mergeSort(nizRightSort)
        ## combine
        niz.clear()
        indexL = indexR = 0
        while(indexL < len(nizLeftSort) and indexR < len(nizRightSort)):
            if(nizLeftSort[ indexL ] >= nizRightSort[ indexR ]):
                niz.append(nizLeftSort[indexL])
                indexL = indexL + 1
            else:
                niz.append(nizRightSort[indexR])
                indexR = indexR + 1

        if(indexL < len(nizLeftSort)):
            niz += nizLeftSort[indexL : ]

```



Slika 3.5: Sortiranje spajanjem pomoću podijeli pa zavladaaj paradigme: koraci

```

elif (indexR < len(nizRightSort)):
    niz += nizRightSort[indexR : ]

return niz

```

Glavna `while`-petlja u rekurziji radi spajanje dva sortirana dijela niza (lijevi i desni). Nezavisna dva (pod)niza `nizLeftSort` i `nizRightSort` se rekurzivno sortiraju, a potom spajaju (u linearnom vremenu).

Primjer 3.5 Množenje cijelih brojeva. Za množenje dva n -cifrena broja, metodom koji se uči u nižim razredima osnovne škole, množeći cifru po cifru, zahtijeva se vrijeme izvršavanja od $O(n^2)$. Konstruisati algoritam zasnovan na paradigmi podijeli pa zavladaaj za problem množenje dva broja.

Rješenje. Jednostavnosti radi, smatramo da su u ulazu binarni brojevi sa istim brojem cifara. Pokušajmo sa alternativnom metodom koja će voditi paradigmi zavadi pa vladaaj. Napišimo svaki od brojeva u obliku: $x = x_1 2^{n/2} + x_2$, $y = y_1 2^{n/2} + y_2$. Tada je

$$xy = (x_1 2^{n/2} + x_2) \cdot (y_1 2^{n/2} + y_2) = x_1 y_1 2^n + (x_1 y_2 + y_1 x_2) 2^{n/2} + x_2 y_2.$$

Dakle, da bismo izračunali proizvod dva (binarna) broja sa n cifara, treba da izračunamo proizvod $q = 4$ broja sa po $n/2$ cifara. Stoga, kompleksnost u ovom slučaju je i dalje $O(n^{\log_2 4}) = O(n^2)$. Da bismo ubrzali algoritam, uradimo sljedeće operacije u datom redoslijedu:

- Izračunajmo x_1y_1 i x_2y_2 .
- Izračunajmo $(x_1 + x_2) \cdot (y_1 + y_2) = x_1y_1 + x_1y_2 + x_2y_1 + x_2y_2$.
- Iz prethodnog izraza izvucimo vrijednost $x_1y_2 + x_2y_1$.

Iz master teoreme, slijedi da je sada kompleksnost algoritma jednaka $O(n^{\log_2 3})$.

3.6 Algoritamska paradigma vraćanja unazad

Vraćanje unazad je algoritamska tehnika za rekurzivno rješavanje problema postepenom izgradnjom rješenja, dio po dio, uklanjajući ona (parcijalna) rješenja koja ne zadovoljavaju ograničenja problema, momentalno nakon dodavanja nove komponente (odnosi se na vrijeme koje je proteklo do dostizanja bilo kojeg nivoa stabla pretraživanja). Za ovu paradigmu se može reći da je ona poboljšanje pristupa grube sile.

U osnovi, traži se rješenje problema među svim dostupnim opcijama. U početku krećemo od jedne moguće opcije i ako se problem može riješiti tom opcijom, vraćamo dato rješenje. U suprotnom vraćamo se unazad, birajući drugu opciju među preostalim opcijama. Slučaj u kojem niti jedna od opcija ne daje rješenje rezultira scenarijem da se neće naći nikakvo rješenje za određeni problem. Vraćanje unazad predstavlja rekurzivni pristup jer se proces pronalaska rješenja iz različitih dostupnih opcija rekurzivno ponavlja, sve dok se ne nađe rješenje problema ili dok se dođe do konačnog stanja.

Tehnika vraćanja unazad u svakom koraku eliminiše one izbore koji ne mogu dati rješenje i nastavlja sa onim izborima koji imaju potencijal da odvedu do rješenja. Algoritmi vraćanja unatrag su generalno eksponencijalni i vremenski i prostorno te se ne preporučuju za rješavanje teških problema.

Postavlja se pitanje kako prepoznati da se neki problem može riješiti pomoću ove algoritamske paradigme. Dakle, osnovna ideja je da se rješenja problema mogu konstruisati inkrementalno, dodavanjem jedne komponente za drugom u svakom koraku u postojeće (parcijalno) rješenje, dok se ne dobije rješenje problema ili ne utvrdi da se tim izborom komponente, nikada neće dobiti validno rješenje (nakon čega se pretraga vraća nazad, pretražujući druge opcije). Prevedeno na jezik rješavanja podproblema, algoritamska paradigma vraćanja unazad rješava sve podprobleme jedan za drugim, kako bi došla do najboljeg mogućeg rješenja.

Pseudokod paradigme vraćanja unazad je dat Algoritmom 6.

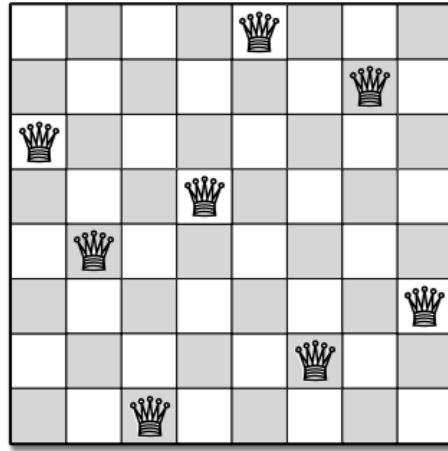
Algoritam 6 Paradigma vraćanja unazad.

```
1: procedure BACKTRACK( $x$ )
2:   if  $x$  nije rješenje then
3:     return false
4:   end if
5:   if  $x$  je novo rješenje then
6:     dodaj  $x$  u listu rješenja
7:   end if
8:   for svaku moguću opciju  $e$  do
9:     BACKTRACK(proširi  $x$  sa  $e$ )
10:  end for
11: end procedure
```

Napomenimo da je implementacija ovog algoritma vezana za konstruisanje stabla odluke, koje se češće naziva *stablo stanja* (eng. *state-space tree*). U okviru ovog kursa ne ulazimo u ovu notaciju, nego bez previše formalizma pokušavamo objasniti rad osnovnih algoritamskih paradigmi i način rješavanja problema pomoću svake od njih.

Primjer 3.6 Problem N kraljica. *Neka je data šahovska ploča dimenzije $N \times N$. Smjestiti N kraljica na šahovsku ploču tako da se nikoje dvije kraljice međusobno ne napadaju. Korisiti algoritam vraćanja unazad.*

Rješenje. Pogledajmo problem 8 kraljica, te jedno njeno rješenje dato na sljedećoj slici.

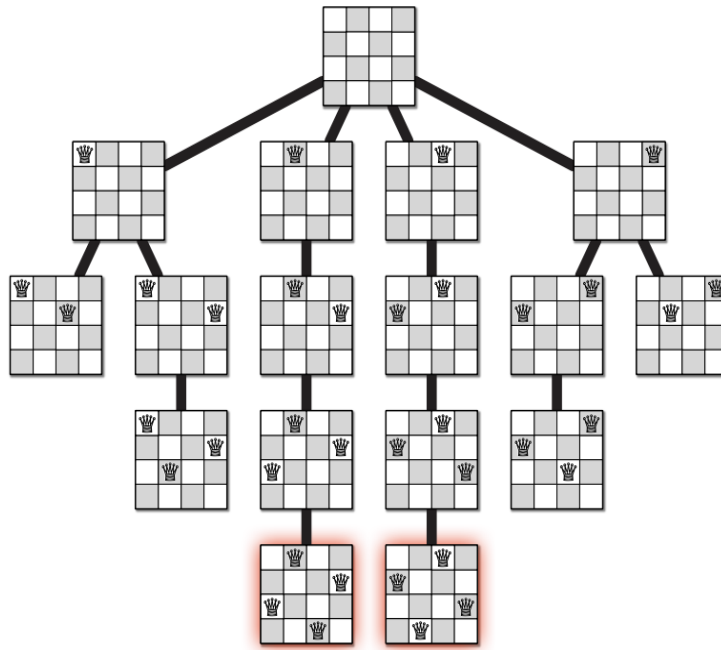


Jasno je da se po jedna kraljica smiješta u svakom redu šahovske ploče. Pozicija kraljice u i -tom redu je određena brojem kolone u kojoj je ta kraljica postavljena. U prevodu, svako rješenje koje odgovara poziciji kraljica na ploči možemo kodirati nizom *niz* dužine N , gdje se na poziciji i nalazi broj kolone u i -tom redu u kojoj je smještena kraljica. Konkretno, rješenje (pozicije kraljica) na prethodnoj slici kodirano je nizom $[5, 7, 1, 4, 2, 8, 6, 3]$.

Algoritam vraćanja unazad kreće od inicijalno prazne ploče i u svakom koraku nastoji dodati kraljicu u sljedeći (prazni) red. Dakle, kreće se od praznog rješenja, u koji se dodaje novi element koji odgovara poziciji kraljice sljedećeg (nepopunjenog) reda. Stoga, pri dodavanju nove kraljice u rješenje *niz*, ispitujemo da li pozicije kraljica u (produženom) rješenju zadovoljavaju uslove zadatka (ne postoje dvije kraljice koje napadaju jedna drugu). U slučaju da postoji konflikt između nekih kraljica, algoritam se vraća unazad na rješenje *niz* prije dodavanja posljednje dodane kraljice, i razmatra nove pozicije za dodavanje kraljice u tom redu – tj. za proširivanje trenutnog rješenja. Ovaj proces se ponavlja sve dok se ne nađe prvo rješenje dužine N koje zadovoljava uslove zadatka (tj. popune se svi redovi na ploči sa kraljicama koje se međusobno ne napadaju), ili se ustanovi da takvo rješenje ne postoji.

Implementacija u Pajtonu koja rješava ovaj problem je data narednim kôdom.

```
def NQueens(niz, N):
    if len(niz) == N: # kompletno rješenje
```



Slika 3.6: Primjer stabla stanja formiranog algoritmom vraćanja unazad za problem 4 kraljice

```

        return niz
    else:
        for i in range(N):
            niz = niz + [i]
            if not valid(niz): #backtrack ako nije
                NQueens(niz + [i])

N = 8
sol = NQueens([], N)
if sol != None:
    print("Rješenje je: ", sol)
else:
    print("Nema rješenja") #npr. za N=3

```

Napomenimo da ovdje nismo prikazali implementaciju funkcije *valid()*, koja vraća *True* ako je pozicija kraljice validna, u protivnom vraća *False*. Ovo ostavljamo čitaocu kao zadatak za vježbanje.

Primjer 3.7 *Problem sume 4 kvadrata. Lagranžova teorema o 4 kvadrata kaže da se svaki prirodni broj može napisati kao zbir četiri broja koji su kvadrati nekih brojeva. Neka je dat broj n . Naći četiri kvadratna broja koji u zbiru daju broj n pomoću algoritma vraćanja unazad.*

Npr. za $n = 2000$, vrijedi $1764 + 196 + 36 + 4 = 2000$.

Rješenje. Rješenje problema predstavlja četvorka $sol = (a, b, c, d)$ prirodnih brojeva, tako da je $a^2 + b^2 + c^2 + d^2 = n$. Ideja rekurzije je odabrati svaki od brojeva jedan po jedan, dok ne dobijemo četvorku koja odgovara uslovima zadatka. Svaki put kada dodamo novi broj x u rješenje sol , naredni rekurivni poziv ispituje trenutno zbir kvadrata brojeva u proširenom rješenju sol . Ako ono prelazi n , algoritam vrši vraćanje unazad i bira novog kandidata za proširivanje rješenja.

Implementacija algoritma vraćanja unazad za ovaj problem je dat narednim kôdom.

```
from math import sqrt, pow

def FourSquareSum(sol):
    sum = 0
    for s in sol:
        sum += pow(s, 2)
    return int(sum)

def FourSumSquares(sol, n):
    if len(sol) == 4:
        if FourSquareSum(sol) == n:
            print(sol)
    else:
        for i in range(1, int(sqrt(n))+2):
            solI = sol + [i]
            sum_sol_i = FourSquareSum(solI)
            if sum_sol_i <= n-(4-len(solI)): # else backtrack
                FourSumSquares(solI, n)

# poziv metode
FourSumSquares([], 20)
```


Kao što primijećujemo u prethodnom kodu, ako je sol kardinalnosti 4, i ako je zbir kvadrata njegovih elemenata jednak n , ispisujemo sol kao rješenje. U protivnom, ako je $|sol| < 4$, (parcijalno) rješenje sol proširujemo ubacujući naredni broj (varijabla $i \in \{1, \dots, \lfloor \sqrt{n} \rfloor + 1\}$), dobijajući niz $solI$. Ako je zbir kvadrata elemenata iz sol veći od $n - (4 - |solI|)$, ovakva ekstenzija nikada neće voditi ka rješenju koje zadovoljava uslove zadatka, pa pretraga ide unazad, rekurzivno, provjeravajući druge kandidate za proširenje trenutnog rješenja sol . Na početku, rješenje sol je predstavljeno prazanim nizom.

3.7 Pohlepni algoritmi

Pohlepni algoritam je pristup rješavanju problema odabirom najbolje dostupne opcije u datom trenutku za trenutno rješenje u koristeći neki (pohlepni, intuitivni) kriterijum. Pristup ne brine da li će lokalno najbolji odabir uticati ili ne na optimalni rezultat. Algoritam nikada ne poništava raniju odluku čak i u slučaju da je izbor pogrešan. Upravo zbog toga se ovom algoritamskom paradigmom generalno ne može garantovati nalazak optimalnog rješenja. Dakle, uvijek se traži najbolji lokalni izbor očekujući da će ta odluka voditi najboljem globalnom rezultatu. U principu, ova paradigma pripada pristupu *odozgo prema dolje* (eng. *top-down approach*).

U neformalnom smislu, pohlepni algoritam je algoritam koji počinje jednostavnim, nekompletnim, tj. parcijalnim rješenjem (teškog) problema, a zatim iterativno traži najbolji način za poboljšanje rješenja proširujući ga iterativno, dok ono ne postane kompletno (neproširivo). Pseudokod ove algoritamske paradigme je dat Algoritmom 7.

Ulaz u algoritam su problem \mathcal{P} , instanca problema, te pohlepna funkcija g . Algoritam kreće od nekog, obično praznog, parcijalnog rješenja koje se kroz iteracije nastoji proširiti najboljim mogućim odlukama (komponentama). U svakoj iteraciji izvršava se sljedeće:

- Za trenutno rješenje x , izračunavaju se komponente koje dopustivo (u odnosu na ograničenja problema) proširuju x ;
- Među svim takvim komponentama, bira se ono koje (naizgled) najviše doprinosi kvalitetu krajnjeg rješenja; ova odluka se donosi na osnovu pohepne funkcije g , koja se još naziva i *heuristička* funkcija.
- Proširimo rješenje sa najboljom komponentom rješenja.

Ovaj proces se izvršava sve dok je skup komponenti koje dopustivo proširuju trenutno rješenje neprazan. Nakon prekida, vraćamo dobijeno rješenje.

Algoritam 7 Shema paradigme pohlepni algoritama.

```

1: Ulaz: instanca nekog problema  $\mathcal{P}$ , pohlepna funkcija  $g$ 
2: Izlaz: (aproksimativno) rješenje instance problema  $\mathcal{P}$ 
3:  $x \leftarrow$  inicijalno parcijalno rješenje problema  $\mathcal{P}$ 
4:  $\mathcal{C} \leftarrow$  odredi komponente rješenja koje dopustivo proširuju rješenje  $x$ 
5: while  $\mathcal{C} \neq \emptyset$  do
6:    $e \leftarrow$  odaberi lokalno najbolju komponentu iz  $\mathcal{C}$  u odnosu na pohlepnu
     funkciju  $g$ 
7:   Proširi  $x$  sa  $e$ 
8:    $\mathcal{C} \leftarrow$  odredi komponente rješenja u odnosu na prošireno rješenje  $x$ 
9: end while
10: return  $x$ 

```

Dakle, da bimo primijenili pohlepni pristup u rješavanju problema, potrebno je definisati: (i) skup komponenti rješenja; (ii) strukturu parcijalnog rješenja; (iii) pohlepnu funkciju koja računa kvalitet komponente koja može proširiti postojeće parcijalno rješenje.

Ovu paradigmu koristimo kada je potrebno dobiti rješenje razumnog kvaliteta u relativno kratkom vremenskom intervalu, na jeftin način (kodirajući relativno brzo). Ovo je često jedna od prvih tehnika koja se primjenjuje u rješavanju teških problema. Ovi algoritmi se takođe koriste u svrhu dobijanja početnog rješenja koje se zatim poboljšava naprednijim tehnikama. U osnovi, ova paradigma nam nudi dobru polaznu osnovu da “opipamo” problem, da se osjeti koliko je on težak, te da se dobije na uvid koliko heurističke, konstruktibilne (aproksimativne) metode mogu biti efikasne za rješavanje takvih problema.

U nastavku navodimo nekoliko primjena pohlepni algoritama u rješavanju raznih tipova (kombinatornih) problema.

3.7.1 Problem razmjene novca

Primjer 3.8 *Imamo X KM-ova. Na raspolaganju su apoeni od x_1, x_2, \dots, x_k KM-ova, negoraničene količine. Pretpostavimo da su $X, x_i, i = 1, \dots, n$ prirodni brojevi.*

Kako razmijeniti X KM-ova tako da dobijemo što manji broj novčanica? Rješiti zadatak pomoću paradigme pohlepkih algoritama.

Rješenje. Definišimo rješenje problema i njegovu strukturu. Bilo koja k -torka (p_1, p_2, \dots, p_k) , gdje je p_i broj apoena novčanice od x_i KM-ova tako da je $\sum_i p_i = X$ (razmijenjena je količina od X KM-ova) predstavlja (dopustivo) rješenje. Parcijalno rješenje problema je bilo koja s -torka (p_1, p_2, \dots, p_s) za koju je $\sum_i p_i \leq X, s \leq k$. Ovakvo rješenje se može nadopuniti do dopustovog.

Konstruišimo sada pohlepnu funkciju. Ideja je da prvo razmatramo novčanice najvećeg mogućeg apoena koji je manji od X . Uzmemo takvih novčanica koliko možemo, ali da ne pređemo vrijednost od X . Od preostalog iznosa X' koji treba da bude razmijenjen, uzmemo onoliko novčanica maksimalnog apoena, koji je manji od X' , koliko god je to moguće. Ovaj proces nastavimo, dok ne dobijemo dopustivo rješenje.

Prema tome, prvo sortiramo vrijednosti apoena od najvećeg do najmanjeg. Pretpostavimo bez smanjenja opštosti da je početni redoslijed apoena x_1, \dots, x_k dat u sortiranom poretku. Pohlepna funkcija od trenutnog parcijalnog rješenja $s_{\text{partial}} = (p_1, \dots, p_s), s \leq k$ nalazi najmanji indeks apoena i koji je veći od s , a da je pri tome $x_i \leq X' = X - \sum_i^s p_i x_i$. Tada je novo prošireno parcijalno rješenje jednako

$$s'_{\text{partial}} = (p_1, \dots, p_s, 0, \dots, 0, \underbrace{\lfloor X'/x_i \rfloor}_{p_i}).$$

Algoritam se završava kada razmotrimo sve apoene, ili prije toga, kada algoritam nađe rješenje. Kôd algoritma je dat u nastavku.

```
def razmijeni(P, X):
    rjesenje = []
    i = 0
    usitnjeno = 0
    brojNovcanica = 0
    sortApoeni = sort(P) # sort apoene opadajuće
    while i < n and usitnjeno < X :
        k = (X - usitnjeno) // P[i] #br. novcanica sortApoeni[i]
        if k > 0:
            usitnjeno = usitnjeno + k * sortApoeni[i]
            brojNovcanica = brojNovcanica + k
```

```

        i = i + 1
    if usitnjeno == X:
        return brojNovcanica
    else:
        return -1 # ne može se usitniti

```

Kompleksnost algoritma. Sortiranje algoritma zahtjeva vrijeme od $O(n \log n)$. Dalje, glavna petlja se izvršava u $O(n)$ vremenu, odakle slijedi da je ukupno vrijeme izvršavanja algoritma jednako $O(n \log n)$, što je, svakako, polinomijsko. Naopmenimo da ovakvom strategijom garantujemo nalazak optimalnog rješenja problema (ili utvrđujemo da rješenje ne postoji).

3.7.2 Dijeljeni problem ruksaka

Primjer 3.9 *Ovaj problem je relaksirana verzija problema ruksaka, obrađen u prethodnim sekcijama. Odluka da li uzeti (cio) proizvod ili ne i staviti u ruksak se relaksira na to da se bilo koji (razlomljeni) dio proizvoda može uzeti i staviti u ruksak. Zamislamo da je u pitanju proizvod kao što je šećer ili ulje koji (teorijski) može da se dijeli na bilo koji razlomljeni (realni) dio.*

Zadatak je odabrati proizvode (dijelove proizvoda) i staviti u ruksak, tako da je profit maksimalan, ali da se poštuje ograničenje za kapacitet ruksaka. Koristiti paradigmu pohlepni algoritama.

Rješenje. Posmatrajmo primjer instance:

$$n = 3, w = [10, 20, 30], p = [60, 100, 120],$$

a kapacitet $C = 50$. Rješenje ove konkretne instance je uzeti prvi i drugi proizvod u potpunosti (svih 10 i 20(kg)), dok uzimamo $2/3$ trećeg proizvoda (20). Kapacitet ruksaka je zadovoljen, a vrijednost proizvoda u ruksaku je jednaka $60 + 100 + 2/3 \cdot 120 = 240$.

Rješenje i struktura problema. Rješenje $c = \{c_1, \dots, c_n\}$, gdje je $0 \leq c_i \leq 1$ dio uzetog proizvoda i koji je stavljen u ruksak, a pri tome je zadovoljen uslov o nepremašivanju kapaciteta ruksaka u vrijednosti C .

Parcijalno rješenje i komponente rješenja. Parcijalno rješenje $c = \{c_{i_1}, \dots, c_{i_k}\}$, pri čemu je $i_s \in [n]$, $i_s \neq i_r$, $s \neq r$, $k \leq n$, a kapacitet ruksaka je zadovoljen.

Komponente rješenja u odnosu na parcijalno rješenje s su svi oni proizvodi $C_s = [n] \setminus \{i_1, \dots, i_k\}$ koji se trenutno ne nalaze u ruksaku. Proširenje parcijalnog rješenja podrazumijeva operaciju dodavanja količine $c_r > 0$ onog proizvoda $r \in C_s$, tako da proširivanjem rješenja s ovom komponentom ne narušavamo kapacitet ruksaka.

Pohlepna funkcija. Razmotrimo sljedeću razumnu strategiju. Favorizujemo proizvode koji imaju veću cijenu po (jednici) težine, tj. kojima je količnik $\frac{c_i}{w_i}$ veći u odnosu na ostale. Sortirajmo proizvode po ovom kriterijumu. Neka je, bez smanjenja opštosti, sortirani redosljed proizvoda po ovom kriterijumu dat sa (indeksima) $1, 2, \dots, n$.

Dodajmo dio c_1 proizvoda indekiranim sa 1 koliko god je to moguće (maksimalno 1) u odnosu na kapacitet ruksaka. Ako je $c_1 = 0$, to znači da je kapacitet ruksaka zadovoljen, pretraga se završava. Inače, nastavimo istom strategijom, ažurirajući preostalo stanje ruksaka koji treba da se popuni, a zatim posmatrajući naredni proizvod (sa indeksom 2). Potom određujemo maksimalni udio proizvoda $c_2 \geq 0$ koji (dopustivo) proširuje trenutno parcijalno rješenje, itd. Postavimo da smo istom strategijom generisali rješenje $s = \{c_1, \dots, c_k\}$. Slično, posmatramo naredni proizvod sa indeksom $i = k + 1$, te izračunamo udio tog proizvoda c_i kojeg ćemo ubaciti u ruksak, tj. proširiti rješenje s . Ako je $c_i = 0$, pretraga se završava. U protivnom, dodajemo komponentu u trenutno parcijalno rješenje, uz ažuriranje stanja o kapacitetu ruksaka koji je preostao za popunjavanje. Pretraga traje sve dok ne popunimo ruksak ili ne razmotrimo sve proizvode.

Implementacija algoritma je data naredim kôdom.

```
def fraction(i):
    return P[i] / W[i]

def fraction_knapsack():
    products = [i for i in range(len(W))] #list of prods
    products.sort(key = fraction, reverse=True)
    #print(products)
    current_C = C
    sol = []
    parts = []
```

```

    for i in products:

        part = min(current_C / W[i], 1)

        if part > 0:
            sol.append(i)
            parts.append(part)
            current_C -= part * W[i]

        else: # knapsack filled
            return sol, parts

    return sol, parts
#instanca:
W = [10, 20, 30]
P = [60, 100, 120]
C = 50
sol, parts = fraction_knapsack()
print(sol, " parts: ", parts)

```

Kompleksnost algoritma. Najviše vremena je utrošeno na sortiranje proizvoda, što iznosi $O(n \log n)$ vremena. Dalje, u glavnoj petlji, koja se vrti najviše n puta, svaka iteracija se izvršava u konstantnom vremenu. Dakle, vrijeme izvršavanja petlje je $O(n)$. Prema tome, algoritam se izvršava u $O(n \log n)$ vremenu.

Napomenimo da ovaj algoritam takođe garantuje nalazak optimalnog rješenja.

Primjer 3.10 Problem pokrivanja skupa. Neka je dat skup X i podskup njegove familije podskupova $\mathcal{F} \subseteq P(x)$, pri čemu je $\cup_{f \in \mathcal{F}} f = X$.

Naći podskup $S \subseteq \mathcal{F}$ minimalne kardinalnosti $|S|$, tako da je $\cup_{s \in S} s = X$. Problem riješiti uz pomoć paradigme pohlepnih algoritama.

Rješenje. Neka je data instanca problema $X = \{1, 2, 3, 4, 5\}$, te $\mathcal{F} = \{\{1, 2, 3\}, \{1, 4\}, \{2, 4, 5\}, \{2, 3\}, \{4, 5\}, \{1, 5\}\}$. Rješenje problema je dato sa $S = \{\{1, 2, 3\}, \{2, 4, 5\}\}$. Njegova kardinalnost je $|S| = 2$.

Ovaj problem ima konkretnu praktičnu primjenu u odabiru minimalnog broja ljudi u komisiju, a da komisija bude kompetentna u svakoj od (traženih) oblasti. Konstruišimo sada algoritam.

Rješenje i struktura rješenja. Skup (skupova) $S \subseteq \mathcal{F}$ je rješenje problema ako unijom elemenata svih skupova iz S dobijamo skup X .

Parcijalno rješenje i komponenta rješenja. Pod parcijalnim rješenjem ovog problema podrazumijevamo bilo koji (pod)skup $S \subseteq \mathcal{F}$ koji ne mora obavezno zadovoljavati uslov o pokrivanju skupa X . Komponenta rješenja S je bilo koji skup iz $C \in \mathcal{F} \setminus S$, dok proširenje rješenja S komponentom rješenja C podrazumijeva operaciju unije, tj. $S' = S \cup \{C\}$.

Pohlepna funkcija. Neka je S parcijalno rješenje i neka ono pokriva $X'_S \subseteq X$ elemenata. Intuitivno, bolja opcija podrazumijeva biranje komponente rješenja C koja u presjeku sa skupom $X \setminus X'_S$ ima više elemanta nego neki drugi C' sa manjim takvim brojem. Razlog je taj što nastojimo pokriti što više skupa X u što manjem broju koraka pohlepnog algoritma. U simboličnom zapisu, pohlepna funkcija ima sljedeći oblik

$$g(S, C) = |C \cap (X \setminus X'_S)|,$$

dok biramo element C^* tako da

$$C^* \leftarrow \operatorname{argmax}\{g(s, C') \mid C' \in \mathcal{F} \setminus S\}$$

kao element koji će proširiti parcijalno rješenje S u datoj iteraciji.

Algoritam u svakom koraku bira skup $C^* \in \mathcal{F}$ u odnosu na pohlepnu funkciju g i dodaje ga u parcijalno rješenje dok god rješenje ne pokrije čitav skup X , nakon čega se algoritam prekida. Napomenimo da se umjesto kompletnih skupova koji se čuvaju u rješenju S , mogu čuvati pozicije (indeksi) skupova iz \mathcal{F} koji pripadaju rješenju S .

Implementacija algoritma je data u nastavku sljedećim kôdom.

```
def g(X_S, C):
    return len((X.difference(X_S)).intersection(C))
def cover_X():
    X_S = set()
    S = [] # solution

    while(X_S != X):
        C_star = istar = None
        g_best = 0

        for i in range(len(F)): # iteration best-next
```

```

        if i not in S:
            g_i = g(X_S, F[i])
            if g_best < g_i:
                g_best = g_i
                i_star = i

        X_S = X_S.union(F[i_star]) #update cover
        S.append(i_star)
    return S

#instanca:
X = {1, 2, 3, 4, 5}
F = [{1, 2, 3}, {1, 4}, {2, 4, 5}, {2, 3}, {4, 5}, {1, 5}]
sol = []
C = 50
sol = cover_X()
print(sol)

```

Kompleksnost algoritma. Glavna `while`-petlja se izvršava u najviše n iteracija. Dalje, U unutrašnjoj `for`-petlji, koja se izvršava $|\mathcal{F}|$ puta, najskuplja operacija je pozivanje `g` funkcije, koja se izvršava u najviše linearnom vremenu, tj. pripada $O(n)$. Dakle, kompleksnost čitave `for` petlje je $O(n \cdot |\mathcal{F}|)$. Dakle, kompleksnost algoritma je $O(n^2 \cdot |\mathcal{F}|)$. Prema tome, efikasnost algoritma najviše zavisi od veličine skupa \mathcal{F} .

Napomenimo da ovako dizajniran algoritam ne garantuje nalazak optimalnog rješenja kao u prethodna dva problema.

Uzmimo npr. instancu $X = \{1, 2, 3, 4, 5, 6\}$, te

$$\mathcal{F} = \{\{1, 2\}, \{3, 4\}, \{4, 5\}, \{3, 6\}, \{5\}, \{6\}\}.$$

Prva iteracija u algoritmu će uzeti, recimo, prvi skup $\{1, 2\}$ u parcijalno rješenje. Dalje, naredna iteracija uzima skup $\{3, 4\}$. Nakon toga, bira se $\{4, 5\}$, pa $\{6\}$. Dakle, konačno rješenje je kardinalnosti 4. Međutim, to očigledno nije optimalno rješenje. Optimalno rješenje je veličine 3 i dato je sa $\{\{1, 2\}, \{3, 6\}, \{4, 5\}\}$.

Primjer 3.11 *Problem sekvencijalnog raspoređivanja poslova.* *Dat je skup poslova, gdje svaki posao ima dozvoljeno krajnje vrijeme završetka*

(eng. *deadline*) i odgovarajući profit, ako posao bude završen prije tog vremena. Pretpostavimo da je za izvršavanje svakog posla potrebno identično (jedinično) vrijeme. Startno vrijeme za izvršavanje svih poslova je isto za sve poslove (pretpostavljamo da kreće od 0).

Maksimizovati ukupni profit, ako samo jedan posao može da bude izvršen u svakoj jedinici vremena. Riješiti problem pomoću paradigme pohlepnih algoritma.

Rješenje. Posmatrajmo primjer instance problema, data sljedećom tabelom.

Posao (ID)	Krajnji rok (Dedlajn)	Profit
1	4	20
2	1	10
3	1	40
4	1	30

Otimalan odabir poslova u slučaju ove instance je: $\{3, 1\}$, sa profitom od 60.

Posmatrajmo narednu instancu problema.

Posao (ID)	Krajnji rok (Dedlajn)	Profit
1	2	100
2	1	19
3	2	27
4	1	25
5	3	15

Rješenje za ovu instancu je odabir poslova iz skupa $\{3, 1, 5\}$ redom kako su i navedeni. Profit ovog rješenja je 142.

Prva ideja za rješavanje ovog problema je koristiti potpunu enumeraciju: generisati sve podskupove skupa poslova te za svaki od njih provjeriti dopustivost i na taj način pratiti maksimalni profit. Međutim, razmotrimo tehniku pohlepnih algoritama.

Rješenje i struktura rješenja. Svaki podskup I_s skupa svih poslova $I = \{1, \dots, n\}$ koji zadovoljava uslove zadatka (da se svi mogu izvršiti u okviru njihovog krajnjeg vremena izvršavanja, te nijedan se ne izvršava paralelno sa drugim), predstavlja rješenje problema. Dalje, struktura rješenja može da bude predstavljena strukturom podataka koja odgovara skupu.

Parcijalno rješenje i njegovo proširenje. Parcijalno rješenje je bilo koji podskup skupa poslova koji zadovoljava uslove zadatka. Ostatak poslova, koji može da se doda postojećem skupu poslova, a pri tome da ne naruši uslove zadatka, pripada skupu komponenti rješenja koje proširuju dato (parcijalno) rješenje.

Pohlepna funkcija. Za parcijalno rješenje I_s , odabрати komponentu rješenja (posao) $i \in I \setminus I_s$ sa maksimalnim profitom koji proširuje I_s na dopustiv način (uslovi zadatka su ispunjeni). Taj posao se, pri tome, smiješta u prvi slobodan (jedinični) interval u kojem se izvršava.

Algoritam. Sortirajmo poslove u odnosu na njihov profit (opadajuće). Bez smanjenja opštosti, pretpostavimo da su oni dati u poretку $\{1, 2, \dots, n\}$. U provoj iteraciji dodajemo posao 1 maksimalnog profita u parcijalno rješenje I_s , koje je inicijalno prazno. Dalje, pretpostavimo da smo u i -toj iteraciji generisali parcijalno rješenje I_s . Tada, u narednoj iteraciji razmatramo posao indeksiran sa $i + 1$. Moguća su dva scenarija.

- Posao sa indeksom $i + 1$ ne može da proširi trenutno parcijalno rješenje I_s ; u tom slučaju idemo na naredni posao ($i + 2$);
- Posao sa indeksom $i + 1$ može da bude dodan u I_s . U tom slučaju, nalazimo prvi slobodan jedinični interval u kojem taj posao može da bude izvršen, u odnosu na već raspoređene poslove u trenutnom rješenju I_s .

Algoritam se izvršava sve dok se ne iteriše kroz sve poslove, nakon čega se vraća konačno rješenje. Implementacija algoritma je data sljedećim kôdom.

```
def ProfitSort(i):
    return Profit[i]

# možemo li dodati i-ti proizvod ili ne (-1) u I_s
def find_next_interval(I_s_covered, i):

    for index, covered in enumerate(I_s_covered):
        if index >= Deadline[i]:
            return -1
        if covered == False:
            return index

    return -1
```

```

def sequetial_scheduling():

    I_s_covered = [False] * max(Deadline)
    I_s = []
    #sortiranje jobs u odnosu na Profit (preprocessing):
    Jobs.sort(key=ProfitSort, reverse=True)

    for i in Jobs:

        interval = find_next_interval(I_s_covered, i)
        if interval >= 0: #nadjena (dopustiv) interval:
            I_s_covered[interval] = True #zauzet
            I_s.append(i)

    return I_s, sum([ Profit[I_s[i]] for i in range(len(I_s)) ])

#instanca:
num_jobs = 5
Jobs = [i for i in range(num_jobs)]
Deadline = [2, 1, 2, 1, 3]
Profit = [100, 19, 27, 25, 15]
# izvršavanje algoritma:
I_s, profit = sequetial_scheduling()
print(I_s, " profit: ", profit)

```

Kompleksnost algoritma. Soritranje poslova se odvija u $O(n \log n)$. Dalje, glavna while-petlja se sastoji od n iteracija. U svakoj iteraciji, najskuplja operacija je pozivanje funkcije `find_next_interval` koja se izvršava u $O(n)$ vremenskoj kompleksnosti. Prema tome, glavna petlja se izvršava u $O(n^2)$. Zaključujemo da se algoritam izvršava u $O(n^2)$ vremenu.

Napomenimo da se može pokazati da ovaj pohlepni algoritam garantuje nalazak optimalnog rješenja.

3.8 Dinamičko programiranje

Dinamičko programiranje je tehnika u programiranju koja pomaže u efikasnom rješavanju klase problema koji se mogu podijeliti na podprobleme

koji se preklapaju (eng. *overlapping subproblems*) i za koje se može definisati *svojstvo optimalne podstrukture* (eng. *optimal substructure property*). Pod preklapajućim podproblemima podrazumijevamo one podprobleme čija se rješenja potrebna više puta da bi se dobilo rješenje početnog problema. Pod optimalnim svojstvom podstrukture podrazumijevamo da kombinovanjem optimalnih rješenja podproblema dobijamo optimalno rješenja datog problema.

Ako se bilo koji problem može podijeliti na podprobleme, koji su zatim rekurzivno dijele na manje podprobleme, te ako postoji preklapanje među njima, rješenja podproblema ima smisla čuvati (u memoriji) za buduću upotrebu. Ovaj proces se naziva *memoizacija*. Memoizacija poboljšava efikasnost rekurzivnog pristupa na taj način da onemogućuje ponovna evaluacija istog podproblema već se to rješenje jednostavno pročita iz određene strukture podataka. Međutim, ovaj pristup zahtjeva dodatnu memoriju koja čuva rješenja podproblema.

Da bismo primijenili dinamičko programiranje u rješavanju određenog problema, potrebno je voditi brigu o sljedećim stvarima.

- Raščlaniti problem na manje probleme; to uključuje izgradnju adekvatne *rekurzije*, tj. matematičkog modela rješenja;
- Čuvati rješenja podproblema u pogodno odabranoj strukturi podataka – *memoizacija* (pristup odozgo).

Princip dinamičkog programiranja može se implementirati i pristupom odozdo prema gore (eng. *bottom-up approach*), i tada se naziva *tabuliranje* (eng. *tabulation*). Tabuliranjem čuvamo rezultate podproblema u tabelu, potom koristimo te rezultate za rješavanje većih podproblema dok se ne riješi cijeli problem. Dakle, za razliku od pristupa odozgo prema dole, gdje rekurzivno formulišemo rješenje problema u smislu njegovih podproblema, treba da preformulišemo problem na način da se prvo riješe podproblemi, a potom koriste izračunata rješenja za nadogradnju i dolazak do rješenja većih podproblema.

Tabuliranje se obično implementira iterativno i pogodno je za probleme za koje je skup ulaza velik.

Pogledajmo sljedeće primjere izračunavanja n -tog fibonačijevog broja u svrhu razumijevanja razlike između memoizacije i tabuliranja.

```
def fib(n, cache={}): # memoizacija
    if n in cache:
```

```

        return cache[n]
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        res = fib(n-1) + fib(n-2)
        cache[n] = res
        return res

def fib(n): #tabuliranje
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        tab = [0] * (n + 1) #init
        tab[0] = 0
        tab[1] = 1
        for i in range(2, n+1):
            tab[i] = tab[i-1] + tab[i-2]
        return tab[n]

```

U implementaciji memoizacije koristimo objekat rječnika `cache` za čuvanje rezultata poziva funkcija, a rekurzija se koristi za izračunavanje rezultata.

U implementaciji tabele koristimo niz nazvan `tab` za čuvanje rezultata podproblema, a koristimo iterativnu implementaciju za izračunavanje rezultata. Obje implementacije vraćaju isti rezultat, ali su zasnovane na drugačijoj implementaciji.

Da zaključimo, princip dinamičkog programiranja ima smisla koristiti gdje god prepoznamo da rekurzivni pristup ima ponovljive pozive za iste ulaze, koji se optimizuju uz pomoć ove algoritamske paradigme.

3.8.1 Rekurzija vs. Dinamičko programiranje

Dinamičko programiranje se uglavnom primjenjuje na rekurzivne algoritme. U osnovi, velika većina problema optimizacije u rješavanju zahtijeva definisanje rekurzije, dok se za njenu optimizaciju koristi dinamičko programiranje.

5	x	x	x
2	4	x	x
7	9	2	x
7	7	6	7

Slika 3.7: Primjer putanje pijuna. Ukupna težina označenog puta je 25.

Ne mogu se svi problemi koji koriste rekurziju koristiti i dinamičko programiranje. Osim ako ne postoji prisustvo podproblema koji se preklapaju, kao u problemu računanja fibonačijevog broja, rekurzija može doći do (jednako efikasnog) rješenja problema pomoću pristupa zavadi pa vladaj. U ovome nalazimo razlog zašto rekurzivni algoritam koji realizuje sortiranje spajanjem ne može koristiti dinamičko programiranje, jer se podproblemi ni na koji način ne preklapaju (dijeljenje niza na dva disjunktne podniza).

3.8.2 Primjene dinamičkog programiranja

Primjer 3.12 *Neka je data kvadratna matrica $A = (a_{ij})$ prirodnih brojeva. Pijun se nalazi u gornjem lijevom uglu. On se može kretati jedno polje dolje ili jedno polje dijagonalno dolje-desno u odnosu na trenutni položaj.*

Kako izabrati putanju hoda pijuna po ploči, od početnog položaja do (bilo koje pozicije) donjeg reda tako da suma brojeva na putu bude maksimalna.

Rješenje. Posmatrajmo jedan primjer validnog puta pijuna na slici 3.7.

Memoizaciju izvršavamo matricom M : na polju (i, j) pamtimo vrijednost najboljeg puta od pozicije $(0, 0)$ (gornji lijevi ugao) do pozicije (i, j) . Pokušajmo rekurzivno odrediti vrijednost $A(i, j)$ u zavisnosti od odgovarajućih podproblema. Moguća su dva scenarija, pijun je došao iz pozicije $(i - 1, j - 1)$ u poziciju (i, j) ili iz pozicije $(i - 1, j)$ u poziciju (i, j) . Prema tome, vrijedi rekurzija:

$$A(i, j) = \max\{A(i - 1, j - 1) + a_{ij}, A(i - 1, j) + a_{ij}\}. \quad (3.2)$$

Bazni slučajevi rekurzije su dati sa

$$A(0, 0) = a_{00}, A(0, j) = 0, j > 1. \quad (3.3)$$

Rješenje problema nalazimo sljedećom formulom:

$$\max_{j=0,\dots,n-1} A(n-1, j) \quad (3.4)$$

Implementacija ovog rješenja je data sljedećim kodom (pristup odozgo prema dolje).

```

M = None #matrica sa svim -1
def init(n):

    M = [ [-1] * n for _ in range(n) ]

    M[0][0] = A[0][0] #bazni slučaj
    for j in range(n):
        M[0][j] = 0

    def solve(i, j):
        if i == 0:
            return M[0][0]

        if M[i][j] != -1: #Podproblem vec izračunat
            return M[i][j]

        best = rec(i-1, j)
        if j > 0:
            best = max(best, solve(i-1, j-1))
        M[i][j] = best + A[i, j]
        return M[i][j]

#instanca
A= [[4 5 7 0]
     [2 3 1 3]
     [1 1 3 1]
     [2 3 3 3]
    ]
init(len(A))
sum_path = 0
for j in range(len(A)):

```

```

init(len(A))
sum_path = max(sum_path, solve(len(A)-1, j))

```

Kompleksnost algoritma. Rekurzija `solve` se poziva n puta. Dalje, svaki poziv funkcije se izvršava u $O(n^2)$ vremenskoj kompleksnosti. Prema tome, ukupna kompleksnost algoritma je $O(n^3)$.

Primjer 3.13 *Rješiti problem ruksaka pomoću dinamičkog programiranja.*

Rješenje. *Podproblemi.* Riješimo problem ruksaka kapaciteta $j \leq C$ birajući prvih $i \leq n$ proizvoda, tj. one proizvode indekirane sa indeksima $1, \dots, i$. Memoriramo rješenje matricom DP , konkretno sa $DP[i][j]$. Dakle, podproblem je definisan parom brojeva (i, j) , $i \in \{0, 1, \dots, n\}$, $j \in \{0, \dots, C\}$.

Rekurzija. Posmatrajmo fiksirani podproblem dat parom indeksa (i, j) i nađimo (manje) podprobleme koji su relevantni za rješavanje njega samog. Vrijedi:

- Ako i -ti proizvod ne učestvuje u (optimalnom) rješenju podproblema (i, j) , tada je $DP[i][j] = DP[i-1][j]$
- Inače, posmatrajmo podprobleme koji uzimaju u razmatranje prvih $k \in \{1, \dots, i-1\}$ proizvoda. Prvo, podproblemi kojima je kapacitet veći od C nisu relevantni za rješavanje podproblema (i, j) . Posmatrajmo neki podproblem (k, l) , $k < i$, $0 \leq l < j$. Ako možemo dopustivo dodati i -ti proizvod u rješenje ovog podproblema, dobijamo potencijalo rješenje većeg podproblema (i, j) . Dakle, ako je $l + w_i \leq j$, potencijalno rješenje podproblema je dato sa vrijednošću $D[k][l] + p_i$. Prema tome, imamo rekurziju:

$$DP[i][j] = \max\{D[i-1][j], \max\{DP[k][j - w_i] + p_i \mid j - w_i \geq 0 \wedge k = 1, \dots, i-1, j \leq C\}\}.$$

Bazni slučajevi. Ako je kapacitet ruksaka $C = 0$, onda je i cijena takvog rješenja jednaka 0, za sve razmatrane proizvode (u ruksak koji ne može da stane ništa, ne stavimo proizvode). Dakle, $DP[i][0] = 0$. Takođe, u ruksak bilo kog kapaciteta $j \leq C$, ukoliko ne postoje proizvodi koji se stavljaju u ruksak, trivijalno ne staju proizvodi, pa je $DP[0][j] = 0$, $0 \leq j \leq C$.

Rješenje proizvoda se memoriše u strukturi DP na poziciji indeksa (n, C) .

Implementacija paradigme dinamičkog programiranja za rješavanje problema ruksaka je data narednim kôdom.


```

def knapsack(n, C, W, p):
    DP = [[0 for x in range(C + 1)] for x in range(n + 1)]

    # tabuliranje
    for i in range(n + 1):
        for j in range(C + 1):
            if i == 0 or j == 0:
                DP[i][j] = 0
            elif W[i-1] <= j:
                DP[i][j] = max(p[i-1]
                               + DP[i-1][j-W[i-1]],
                               DP[i-1][j])
            else:
                DP[i][j] = DP[i-1][j]

    return K[n][C]

#instanca:
p = [60, 100, 120]
W = [10, 20, 30]
C = 50
n = len(profit)
#poziv funkcije
print knapSack(n, C, W, p, n)

```

Kompleksnost algoritma. Najskuplji dio algoritma su dvije ugnježdene for petlje, koje izvršavaju $O(n \cdot C)$ iteracija. Svaka od iteracija se izvršava u konstantnom vremenu, pa je i ukupna kompleksnost algoritma jednaka $O(n \cdot C)$.

Navedimo primjer jedne instance i DP tabelu koja se kreira rekursivno.

Neka je $n = 4$, te neka je kapacitet ruksaka $C = 5$, dok je $W = (2, 3, 4, 5)$, a $P = (3, 4, 5, 6)$. U prvom koraku ($i = 0$ i $j = 0$) se popunjavaju nule u prvoj vrsti i prvoj koloni matrice DP , kako je prikazano u sljedećoj tabeli:

$j \rightarrow$	Proizvod	0	1	2	3	4	5
$i = 0$		0	0	0	0	0	0
$i = 1$	$w_1 = 2$ $p_1 = 3$	0					
$i = 2$	$w_2 = 3$ $p_2 = 4$	0					
$i = 3$	$w_3 = 4$ $p_3 = 5$	0					
$i = 4$	$w_4 = 5$ $p_4 = 6$	0					

Dalje, za $i = 1$, imao sljedeće izračunavanje:

- proizvod 1 ima težinu 2 i vrijednost 3, pa u drugoj vrsti matrice DP ($i = 1$) sve do trećeg elementa ($j = 2$) vrijednosti su 0, dok je krenuvši sa tim elementom, svakoj poziciji dodijeljena vrijednost 3 ($=p_1$).

Dakle, popunjavamo drugu vrstu kao u sljedećoj tabeli:

$j \rightarrow$	Proizvod	0	1	2	3	4	5
$i = 0$		0	0	0	0	0	0
$i = 1$	$w_1 = 2$ $p_1 = 3$	0	0	3	3	3	3
$i = 2$	$w_2 = 3$ $p_2 = 4$	0					
$i = 3$	$w_3 = 4$ $p_3 = 5$	0					
$i = 4$	$w_4 = 5$ $p_4 = 6$	0					

Dalje, za treću vrstu ($i = 2$) imamo sljedeću tabelu:

$j \rightarrow$	Proizvod	0	1	2	3	4	5
$i = 0$		0	0	0	0	0	0
$i = 1$	$w_1 = 2$ $p_1 = 3$	0	0	3	3	3	3
$i = 2$	$w_2 = 3$ $p_2 = 4$	0	0	3	4	4	7
$i = 3$	$w_3 = 4$ $p_3 = 5$	0					
$i = 4$	$w_4 = 5$ $p_4 = 6$	0					

Diskutujemo slučaj $DP[2][4] = 4$. Ako rješenje podproblema ne uzima proizvod 2, $DP[1][4] = 3$ je potencijalno rješenje. U protivnom, ako se uzme, moguće rješenje razmatranog problema je $4 + DP[2][4 - 4] = 4 + 0 = 4$. Kako je $\max\{3, 4\} = 4$, slijedi da je $DP[2][4] = 4$.

Narednom iteracijom, popunjavamo četvrtu vrstu ($i = 3$) u prethodnoj tabeli, pa imamo:

$j \rightarrow$	Proizvod	0	1	2	3	4	5
$i = 0$		0	0	0	0	0	0
$i = 1$	$w_1 = 2 \quad p_1 = 3$	0	0	3	3	3	3
$i = 2$	$w_2 = 3 \quad p_2 = 4$	0	0	3	4	4	7
$i = 3$	$w_3 = 4 \quad p_3 = 5$	0	0	3	4	5	7
$i = 4$	$w_4 = 5 \quad p_4 = 6$	0					

Diskutujemo slučaj $DP[3][4] = 5$. Ako rješenje podproblema ne uzima proizvod 3, $DP[2][4] = 3$ je potencijalno rješenje ovog podproblema. Dalje, ako uzima, onda je rješenje $DP[2][4-4] + 5 = 0 + 5 = 5$ potencijalni kandidat za razmatrani podproblem. Kako je $\max\{4, 5\} = 5$, slijedi zaključak.

Konačno, posljednjom iteracijom popunjavamo petu vrstu ($i = 4$) u prethodnoj tabeli, pa imamo:

$j \rightarrow$	Proizvod	0	1	2	3	4	5
$i = 0$		0	0	0	0	0	0
$i = 1$	$w_1 = 2 \quad p_1 = 3$	0	0	3	3	3	3
$i = 2$	$w_2 = 3 \quad p_2 = 4$	0	0	3	4	4	7
$i = 3$	$w_3 = 4 \quad p_3 = 5$	0	0	3	4	5	7
$i = 4$	$w_4 = 5 \quad p_4 = 6$	0	0	3	4	5	7

Diskutujemo slučaj $DP[4][5] = 7$. Ako rješenje podproblema ne uzima proizvod 4, $DP[3][5] = 7$ je potencijalno rješenje ovog podproblema. Dalje, ako uzima proizvod 4, onda je i rješenje $DP[4][5-5] + 6 = 0 + 6 = 6$ potencijalni kandidat za razmatrani podproblem. Kako je $\max\{6, 7\} = 7$, slijedi zaključak.

Definicija 3.1 String s je podniz stringa s_1 akko se s može dobiti brisanjem karaktera iz stringa s_1 .

Npr. **acd** je podniz stringa **abbbcddef**.

Prefiks stringa s dužine k je podstring stringa s koji se sastoji od prvih k karaktera. Npr. prefiks dužine tri stringa **abbbcddef** je string **abb**.

Primjer 3.14 Problem najdužeg zajedničkog podniza (eng. the longest common subsequence problem – LCSP). Neka su u ulazu data dva stringa s_1 i s_2 .

Naći najduži mogući string s koji je podniz oba ulazna stringa.

Za instancu $s_1 = \text{a a t d c c d d c}$ i $s_2 = \text{a g c c d d t a}$, rješenje je string $s = \text{a c c d d}$.

Rješenje. Definišimo podproblem indukovano parom (i, j) , $0 \leq i \leq |s_1|$, $0 \leq j \leq |s_2|$ za dva stringa: prvi predstavlja prefiks dužine i stringa s_1 , a drugi predstavlja prefiks dužine j stringa s_2 . Čuvajmo rješenje svakog podproblema indukovano sa (i, j) u matrici DP na poziciji (i, j) , tj. $DP[i][j]$.

Rekurzija. Neka je dat podproblem (i, j) . Nađimo strategiju razbijanja problema na manje podprobleme i sve podprobleme na osnovu kojih se može konstruisati rješenje podproblema (i, j) . Posmatrajmo karaktere na pozicijama i i j , prvog i drugog ulaznog stringa, redom. Moguća su dva slučaja.

- $s_1[i-1] = s_2[j-1]$: U ovom slučaju, vrijedi rekurzija $DP[i][j] = 1 + DP[i-1][j-1]$, jer se optimalno rješenje podproblema $(i-1, j-1)$ tada može proširiti za 1 (karakter $s_1[i-1]$) tako da je novonastalo rješenje optimalno za podproblem (i, j) .
- $s_1[i-1] \neq s_2[j-1]$: U ovom slučaju, sigurno karakter predstavljen parom karaktera $(s_1[i-1], s_2[j-1])$ ne doprinosi optimalnom rješenju podproblema (i, j) . Dakle, za određivanje optimuma podproblema (i, j) relevantna su dva podproblema: $(i-1, j)$ i $(i, j-1)$, nastala izbacivanjem krajnjih karaktera prvo u prvom, pa onda u drugom prefiksu. Duže rješenja ta dva podproblema će činiti i optimalno rješenje za podproblem (i, j) . Dakle, u ovom slučaju vrijedi rekurzija:

$$DP[i][j] = \max\{DP[i-1][j], DP[i][j-1]\}$$

Rješenje problema nalazimo u DP na poziciji $(|s_1|, |s_2|)$.

Bazni slučajevi. Ako je jedan od ulaznih stringova prazan, rješenje je jednako 0, tj. $DP[i][0] = 0$, $0 \leq i \leq |s_1|$, $DP[0][j] = 0$, $0 \leq j \leq |s_2|$.

Implementacija paradigme dinamičkog programiranja za rješavanje LCSP je data narednim kôdom.

```
def LCS(s1, s2):

    if len(s1)==0 or len(s2)==0:
        return 0

    DP = [ [0]*(len(s2)+1) for _ in range(len(s1)+1)]
```

```

for i in range(1, len(s1)+1):
    for j in range(1, len(s2)+1):
        if s1[i] == s2[j]:
            DP[i][j] = DP[i-1][j-1] + 1:
        else:
            DP[i][j] = max(DP[i-1][j], DP[i][j-1])

return DP[len(s1)][len(s2)]

#instanca:
s1 = "abcdcdcdaa"
s2 = "abbaabccdd"
lcs = LCS(s1, s2)
print("Duzina LCS-a je: ", lcs)

```

Primijetimo da implementacija koristi princip odozdo-nagore.

Kompleksnost algoritma. Dvije for-petlje izvršavaju ukupno $O(|s_1| \cdot |s_2|)$ iteracija. Svaka od iteracija se izvršava u konstantnom vremenu, pa je shodno tome ukupno vrijeme izvršavanja algoritma jednako $O(|s_1| \cdot |s_2|)$. Ako je $n = \max\{|s_1|, |s_2|\}$, dobijamo kvadratnu $O(n^2)$ kompleksnost.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

Slika 3.8: DP matrica za dva stringa $s_1 = \text{MZJAWXU}$ i $s_2 = \text{XMJYAUZ}$. Rješenje je dužine 4 (pogledati donji desni ugao matrice).

Primjer 3.15 Problem sume podskupa. Neka je u ulazu dat niz niz (od n) nenegativnih cijelih brojeva i broj sum.

Da li postoji podniz datog niza niz čija je suma elemenata jednaka vrijednosti sum.

Rješenje. Definišimo podproblem indukovani parom (i, j) , $0 \leq i \leq n, 0 \leq j \leq \text{sum}$ koji razmatra prvih i brojeva niza niz i za njih vraća odgovor (*True* ili *False*) da li se među njima nalaze brojevi čiji je zbir jednak j . Čuvajmo rješenje ovakvog podproblema u matrici DP , na poziciji (i, j) , tj. $DP[i][j]$.

Rekurzija. Analizirajmo podprobleme koji su relevantni za rješavanje podproblema (i, j) . Razlikujemo dva slučaja:

- i -ti element niza niz može biti dio rješenja. Tada se podproblem (i, j) redukuje na podproblem $(i - 1, j - \text{niz}[i - 1])$ – pod uslovom da je $j - \text{niz}[i - 1] \geq 0$ – i podproblem $(i - 1, j)$

$$DP[i, j] = DP[i - 1, j] \vee DP[i - 1][j - \text{niz}[i - 1]].$$

- i -ti element niza niz nije dio rješenja. Tada se podproblem (i, j) redukuje na podproblem $(i - 1, j)$, a rekurzija data sa

$$DP[i][j] = DP[i - 1][j].$$

Bazni slučajevi. $DP[i][0] = \text{True}, i \geq 0$, i $DP[0][j] = \text{False}$, za $j \geq 1$.

Implementacija paradigme dinamičkog programiranja za rješavanje problema sume podskupa je data narednim kôdom.

```
def sum_array(niz, suma):
    #tabulation:
    DP = [[False] * (suma + 1) for _ in range(len(niz) + 1) ]
    for i in range(len(niz)+1):
        DP[i][0] = True

    for i in range(1, len(niz)+1):
        for j in range(1, suma+1):
            if j >= niz[i-1]:
                DP[i][j] = DP[i-1][j-niz[i-1]] or DP[i-1][j]
            else:
                DP[i][j] = DP[i-1][j]

    #instanca:
    niz = [3, 34, 4, 12, 5, 2]
    suma = 9
    postoji = sum_array(niz, suma)
```

```
print("Postoji podniz sa sumom" if postoji else
      "Ne postoji podniz sa sumom")
```

Primijetimo da smo dinamičko programiranje implementirali pomoću pristupa *odozdo prema gore*.

Implementirajmo sada dinamičko programiranje za ovaj problem pristupom odozgo prema dolje.

```
def init(niz, sum):
    global DP
    DP = [[-1] * (sum + 1) for _ in range(len(niz) + 1) ]

def sum_array_top_down(niz, sum, i, j, DP = [ ]):

    if DP[i][j] != -1:
        return DP[i][j]

    if j == 0:
        DP[i][0] = True
        return True
    if i == 0:
        DP[0][j] == False
        return False
    #recursion:
    if j >= niz[i-1]:
        DP[i][j] = sum_array_top_down(niz, sum, i-1, j, DP)
        or sum_array_top_down(niz, sum, i, j-niz[i-1], DP)
        return DP[i][j]
    else:
        DP[i][j] = sum_array_top_down(niz, sum, i-1, j, DP)
        return DP[i][j]
```

Kompleksnost algoritma. Broj iteracija koje se izvršavaju je jednak $O(n \cdot sum)$. Svaka iteracija se izvršava u konstantnom vremenu, pa je i ukupna kompleksnost algoritma jednaka $O(n \cdot sum)$. Dakle, ako je vrijednost *sum* ogromna, ekponencijalno velika u odnosu na veličinu niza, ovakav algoritam će biti neefikasan.

Primjer 3.16 Problem rezanja štapa. *Dat je štap dužine n i lista $price$ koja odgovara cijenama štapa dužine i , za sve $1 \leq i \leq n$.*

Pronaći optimalan način da se isiječe štap na manje štapove da bi se maksimizovao ukupan profit.

Rješenje. Posmatrajmo primjer jedne instance problema:

$$price = [1, 5, 8, 9, 10, 17, 17, 20].$$

Neka je štap dužine $d = 4$. Rješenje konkretne instance je rezati štap na dva dijela dužine od po 2. Zarada je u tom slučaju jednaka $5 + 5 = 10$.

Podprobleme datog problema indukujemo indeksima i , $0 \leq i \leq n$, gdje vrijednost indeksa i označava instancu problem rezanja štapa dužine i . Pretpostavimo da vrijednost optimalnog rezanja (tj. optimalno rješenje) instance dužine i čuvamo u nizovnoj strukturi DP na poziciji i , tj. $DP[i]$.

Rekurzija. Podproblem indukovani vrijednošću i možemo riješiti rješavajući podprobleme koji su indukovani sa k , $0 \leq k < i$. To znači da štap dužine i dijelimo na dva dijela, dužine k (sa cijenom $price[k-1]$) i štap dužine $(i-k)$, koji opet razmatramo rekurzivno.

Rekurzija u ovom slučaju izgleda ovako:

$$DP[i] = \max\{DP[i-k] + price[k-1] \mid 1 \leq k \leq i\}.$$

Za bazne slučajeve je $DP[0] = 0$, dok je $DP[1] = price[0]$. Rješenje inicijalnog problema se nalazi u $DP[n]$.

Implementacija paradigme dinamičkog programiranja za rješavanje problema rezanja štapa je data narednim kôdom.

```
def stick_cut(d, price):

    DP = [-1] * (d+1)    #init
    DP[0] = 0
    DP[1] = price[0]

    for i in range(2, d+1):
        for k in range(1, i+1):
            if DP[i] < DP[i-k] + price[k-1]:
                DP[i] = DP[i-k] + price[k-1]
```



```

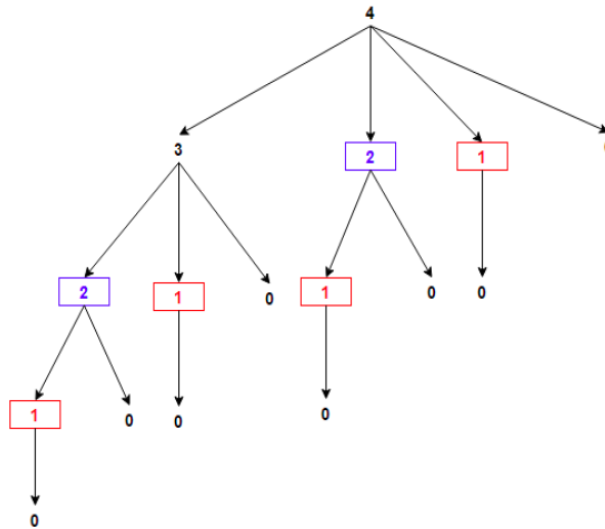
    return DP[d]

#instanca
d = 4
price = [1, 5, 8, 9, 10, 17, 17, 20]
max_profit = stick_cut(d, price) #poziv:
print(max_profit)

```

Kompleksnost algoritma. Dvije for petlje izvršavaju ukupno najviše $O(d^2)$. Kako se svaka iteracija izvršava u konstantnom vremenu, zaključujemo da je kompleksnost algoritma jednaka $O(d^2)$.

Napomena. Pogledajmo drvo koje se kreira algoritmom grube sile na slici 3.9 pri rješavanju problema rezanja štapa za $d = 4$. Ovaj pristup enumeriše sva moguća rješenja za rezanje štapa dužine $d = 4$. Prvo uzima cio štap u razmatranje (i njegovu cijenu), zatim otkida dio štapa dužine 1, dok se ostatak štapa (dužine 3) rješava (dijeli) rekursivno itd. Kompletно rješenje dobijamo sa listovima stabla (označeni sa 0).



Slika 3.9: Drvo grananja paradigme grube sile za rješavanje problema rezanja štapa.

Npr. put koji posjećuje čvorove sa oznakama $4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$, odgovara rješenju u kojem se štap sječe na 4 manja štapa dužine 1 (cijene $4 \times price[0]$). Slično, za put koji posjećuje čvorove sa oznakama $4 \rightarrow 3 \rightarrow 1 \rightarrow 0$, odgovara rješenju u kojem se štap sječe na dva manja štapa dužine 1, te jedan dužine 2 (cijene $2 \times price[0] + price[1]$).

Jasno se vidi da se podproblemi isprepliću, tj. da se, npr. rješavanje problema štapa dužine dva računa više (dva) puta. U slučaju dinamičkog programiranja implementiran pristupom odozgo prema dolje, stablo će biti značajno redukovano, jer neće postojati rekalkulacije istih podproblema (dakle, nema identičnih kopija podstabala u različitim regionima DP stabla).

Zadaci

1. Riješiti problem trgovačkog putnika pomoću algoritamske paradigme grube sile.
2. Riješiti problem ruksaka pomoću algoritamske paradigme grube sile.
3. Dati su brojevi $0 \leq M < N \leq 1,000,000$. Odrediti najmanji broj sastavljen samo od neparnih cifara, koji pri dijeljenju sa N daje ostatak M . Ukoliko traženi broj ne postoji, vratiti 1.
4. Neka su dati prirodni brojevi n , koji označava broj promjenljivih, i s , koji označava sumu promjenljivih. Uz pomoć rekurzivnih algoritama napisati program koji pronalazi sva moguća rješenja za promjenljive kojih ima n i čija suma treba biti jednaka s .

Instanca. $n = 5, s = 4$; *Rješenje.* 70.

Napomena: instanca predstavlja jednačinu $x_1 + x_2 + x_3 + x_4 + x_5 = 4, x_i \in \mathbb{N}$. Sva moguća rješenja su: $(1\ 1\ 1\ 1\ 0), (1\ 0\ 1\ 1\ 1), (0\ 1\ 1\ 1\ 1), \dots, (2\ 1\ 0\ 0\ 1), \dots, (2\ 2\ 0\ 0\ 0)$ itd.

5. Pronaći podskup elemenata niza tako da je proizvod elemenata u podskupu minimalan. Koristiti paradigmu pohlepnih algoritama.
Instanca. niz = $[-1, -1, -2, 4, 3]$; *Rješenje.* -24 ($= (-2) \cdot (-1) \cdot (-1) \cdot 4 \cdot 3$).
6. Svaki pozitivan razlomak možemo predstaviti predstaviti kao zbir jedinstvenih jediničnih razlomaka. Razlomak je jedinični razlomak ako je

brojnik 1, a nazivnik pozitivan cijeli broj; npr. $1/3$ je jedinični razlomak. Npr. $2/3 = 1/2 + 1/6$, $3/7 = 1/3 + 1/11 + 1/231$. Uz pomoć paradigme pohlepni algoritama, za proizvoljan razlomak, naći jedinične razlomke koje u zbiru daju taj razlomak.

7. Riješiti problem nalaska najdužeg zajedničkog podniza dva stringa pomoću dinamičkog programiranja pristupom odozgo prema dolje.
8. Neka je dat niz brojeva u ulazu. Konstruisati algoritam dinamičkog programiranja koji daje odgovor na pitanje li postoji particionisanje niza na dva (disjunktna) skupa tako da je zbir u obe particije jednak.
9. Implementirati metod dinamičkog programiranja za problem rezanja štapa koristeći pristup odozgo prema dolje.
10. Neka su u ulazu data dva stringa s_1 i s_2 . Uz pomoć dinamičkog programiranja, pronaći najkraći string s tako da su stringovi s_1 i s_2 podnizovi stringa s .
11. Data je riječ i riječnik (skup riječi). Ispitati da li se riječ može podijeliti na segmente (podstringove) tako da oni pripadaju riječniku.

Instanca.

`dict = {this, th, is, famous, Word, break, b, r, e, a, k, br, bre, brea, ak, problem}`

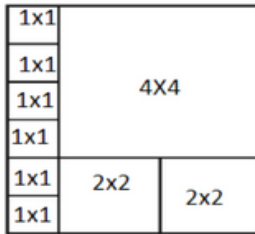
te

`word = Wordbreakproblem.`

Rješenje. `Word break problem`, kao jedno od rješenja.

12. Neka je dat pravougaonik dimenzija $N \times M$. Pretpostavimo da imamo na raspolaganju beskonačan broj pločica formata $2^i \times 2^i$, $i = 0, 1, 2, 3, \dots$. Napisati program koji rekurzivno pronalazi minimalan broj pločica koji je potreban da bi se popločao pravougaonik datih dimenzija.

Instanca. $N = 6$, $M = 5$; *Rješenje.* 9 (pločica) – vidjeti narednu sliku.

Slika 3.10: Optimalno pokrivanje ploče 6×5

13. Napisati program koji generiše sve mogućnosti da se stavi $+$ ili $-$ ili ništa između brojeva $1, 2, \dots, 9$ (ovim redom) tako da se kao rezultat izvršenja operacija dobije vrijednost 100. Npr. jedno takvo rješenje je $1 + 2 + 3 - 4 + 5 + 6 + 78 + 9 = 100$.
14. Za dva stringa, napisati program koji ispisuje najkraći niz umetanja i brisanja karaktera u prvi string čijom se primjenom dobija drugi string.

Glava 4

Primjena algoritamskih paradigmi na probleme iz aritmetike

U ovoj glavi se bavimo rješavanjem nekih interesantnih problema iz aritmetike primijenjujući algoritamske paradigme koje smo obradili u prethodnoj glavi.

4.1 Eratostenovo sito

Definicija 4.1 Broj $n \in \mathbb{N}$ je prost akko je djeljiv isključivo sa 1 i sa samim sobom.

Npr. lista prvih 5 prostih brojeva se: 2, 3, 5, 7, i 11. Broj 6 nije prost, jer je djeljiv (pored 1 i 6) sa 2 i sa 3.

Postavimo sljedeći zadatak. *Ispitati koliko se prostih brojeva može pronaći, u nekom razumnom vremenu, među prvih N prirodnih brojeva.*

Paradigmom brutalne sile, ovo bi se moglo riješiti sljedećim rezonovanjem: ispitajmo svaki od (prirodnih) brojeva u intervalu od 2 do N da li je prost ili ne, sljedećom jednostavnom procedurom:

```
def prost(n):  
  
    for i in range(2, n):
```

```

    if n % i == 0:
        return False
    return True

```

Kompleksnost ove procedure je $O(n) = O(N)$ i nju pozivamo $N - 1$ puta. Dalje, kompleksnost ovog algoritma bi bila $O(N^2)$. Možemo li bolje od ovoga?

Iskoristimo ideju da se za neki prost broj p , niti jedan od brojeva $k \cdot p$, $k = 1, \dots, \lceil N/p \rceil$ nije prost, osim za $k = 1$. Dakle, ove brojeve možemo direktno profiltrirati iz skupa $\{2, \dots, N\}$. Zatim, isto se uradi i za naredni prost broj $p' > p$ u nizu preostalih brojeva, i tako progresivno, sve dok ne dođemo do posljednjeg broja N . Dakle, u svakoj iteraciji algoritma, kroz “sito” prolaze (filtriraju se) brojevi koji nisu prosti, izvršavajući provjeru da li je razmatrani broj prost ili ne u konstantnom vremenu. Brojevi koji preostanu u “situ” su upravo svi prosti brojevi koji su manji (ili jednaki) N .

Implementacija algoritma je data narednim kôdom.

```

def sieve(N):

    sieve = [True] * (N+1)
    sieve[0] = sieve[1] = False

    p = 2
    while p <= N:

        if sieve[p]:

            index = 2 * p
            while index <= N:
                sieve[index] = False
                index += p

            p = p + 1

    for i in range(N+1):
        if sieve[i]: #broj je prost
            print(i)
#poziv funkcije:

```

```
n = 1000
sieve(n)
```

Kompleksnost algoritma. Može se pokazati da je kompleksnost prethodnog algoritma jednaka $O(n \log \log n)$.

Demonstrirajmo prvih nekoliko iteracija ovog algoritma u tabelarnom zapisu, za $N = 50$.

U početnoj iteraciji, inicira se niz (tabela) brojeva gdje su svi brojevi označeni kao prosti (na poziciji koja odgovara broju se čuva vrijednost *True*).

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Dalje, u prvoj iteraciji, nalazimo se na poziciji 2, gdje elemente na pozicijama $k \cdot 2, k = 2, \dots, 50$ eliminišemo kroz sito (kao oni koji nisu prosti, dodijeljujući im vrijednost *False*). Vizuelno, preostali brojevi za ispitivanje su oni neoznačeni:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

Dalje, u drugoj iteraciji, prelazimo na poziciju 3. Kako je ona označena sa *True* (prost broj), elemente na pozicijama $k \cdot 3, k = 2, \dots, 16$ eliminišemo kroz sito (jer nisu prosti). Preostali brojevi za ispitivanje su oni neoznačeni u narednoj tabeli.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50

U narednoj iteraciji prelazimo na poziciji 4. Kako to nije prost broj (dakle filtriran prethodno), prelazimo na naredni broj (5), koji je prost, nakon čega ponavljamo operacije opisane prethodnim iteracijama.

4.2 Faktorizacija broja

Fundamentalna teorema aritmetike nam kaže da se svaki broj $n \in \mathbb{N}$ može na jedinstven način faktorisati na proizvod prostih činilaca, do na redoslijed faktora. Dakle, svaki n se može zapisati u formi

$$n = \prod_{i=1}^k p_i^{n_i}$$

za neke $k, n_1, \dots, n_i, \dots, n_k \in \mathbb{N}$.

Npr. broj 120 se može faktorisati kao $2^3 \cdot 3^1 \cdot 5^1$.

Sa stanovišta izračunavanja, postavlja se pitanje kako problem izračunavanja faktorizacije broja izvršiti efikasno. U tu svrhu, iskoristimo znanje o Eratostenovom situ.

Prvo primijenimo paradigmu podijeli pa zavladaaj. Dake, za broj n , nađimo najmanji prost broj p koji dijeli n . Tada vrijedi $n = p \cdot n/p$. Dalje, rekursivno primijenjujemo istu akciju za broj n/p sve dok je $n > 1$. Bazni slučaj je kada je broj n prost, i pri tome se kao takav i vraća, prekidajući momentalno rad rekurzije.

Dakle, potrebno je generisati strukturu podataka tako da na poziciji i čuva najmanji prost faktor broja i ; ako je broj prost, čuvamo vrijednost 0.

Implementacija ovakve (nizovne) strukture je data narednim kôdom.

```
def sieve_adaptation(n):
```

```

sieve_fact = [0] * (n+1)
sieve_fact[0] = sieve_fact[1] = 0

p = 2
while p <= n:

    if sieve_fact[p] == 0:

        index = 2 * p
        while index <= n:
            sieve_fact[index] = p
            index += p

        p = p + 1

    return sieve_fact

def factorization(n):

    factorize_min_p = sieve_fact(n)
    F = []
    while True:
        i = factorize_min_p[n]
        if i == 0:
            F.append(n)
            return F
        else:
            n = n // i
            F.append(i)
    return F

#ulaz:
n = 120
F = factorization(n)
print("Lista faktora je: ", F)

```

Kompleksnost algoritma. Najskuplja operacija u algoritmu je adaptacija algoritma Eratostenovog sita (funkcija `sieve_adaptation`) i ona se izvršava u $O(n \log \log n)$ vremenu. Glavna `while`-petlja se izvršava u linearnom $O(n)$

vremenu. Dakle, čitav algoritma se izvršava u $O(n \log \log n)$ vremenu.

4.3 Izračunavanje binomnih koeficijenata

Binomni koeficijenti igraju bitnu ulogu u kombinatorici. Npr. broj načina na koji se k ljudi može izabrati iz skupa od n ljudi je predstavljen binomnim koeficijentom $\binom{n}{k}$.

Definicija 4.2 Binomni koeficijent $\binom{n}{k}$ gdje je $n, k \in \mathbb{N}, k > 0$ je dat sa

$$\binom{n}{k} = \frac{n(n-1) \cdots (n-k+1)}{k!}$$

Definišemo $\binom{n}{0} := 1$. Binomni koeficijent $\binom{n}{1} = n$, dok je $\binom{n}{n} = 1$ za sve $n \in \mathbb{N}$. Takođe, ako je $n < k$, vrijedi $\binom{n}{k} = 0$.

Lako se može pokazati da vrijedi Paskalova jednakost:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

što nam daje rekurziju za računanje binomnih koeficijenata. Dakle, ako označimo $\text{binom}(n, k) = \binom{n}{k}$, vrijedi:

$$\text{binom}(n, k) = \text{binom}(n-1, k-1) + \text{binom}(n-1, k).$$

Implementacija rekurzivnog pristupa je data sljedećim kôdom.

```
def binom(n, k):
    if k == 0:
        return 1
    if n < k:
        return 0
    return binom(n-1, k-1) + binom(n-1, k)
```

Kompleksnost algoritma. Faktor grananja rekurzije je 2. To znači da na svakoj narednoj dubini rekurzije imamo (najviše) duplo više rekurzivnih poziva nego je to slučaj na dubini prije. Kako je dubina rekurzije maksimalno n (vrijednost prvog ili drugog atributa se smanjuje za 1 u narednom pozivu rekurzije), zaključujemo da je maksimalni broj poziva rekurzije reda veličine

$O(2^n)$, što nam daje eksponencijalnu kompleksnost.

Pokušajmo optimizovati ovu naivnu implementaciju rekurzije. Jasno se vidi da se većina podproblema rješava više puta iz nule. Zbog toga upotrebimo princip memoizacije. Struktura u koju čuvamo rješenja je matrica $B[i][j] = \binom{i}{j}$. Bazni slučaj je dat sa $B(n, 0) = 1 = B(n, n)$. Takođe, ako je $i < j$, imamo $B[i][j] = 0$. Implementacija memoizacije za izračunavanje binomnog koeficijenta $\binom{n}{k}$ je data narednim kôdom.

```
def initialization(n, k):
    Binom = [[-1] * (k+1) for _ in range(n+1)]
    return Binom

def binom_memoization(i, j, Binom):

    if Binom[i][j] != -1:
        return Binom[i][j]
    if j == 0:
        Binom[i][j] = 1
        return 1
    if i < j:
        Binom[i][j] = 0
        return 0
    Binom[i][j] = binom_memoization(i-1, j-1, Binom)
                  + binom_memoization(i-1, j, Binom)
    return Binom[i][j]

#poziv metode:
n = 10
k = 4
Binom = initialization(n, k)

n_over_k = binom_memoization(n, k, Binom)
print(n_over_k)
```

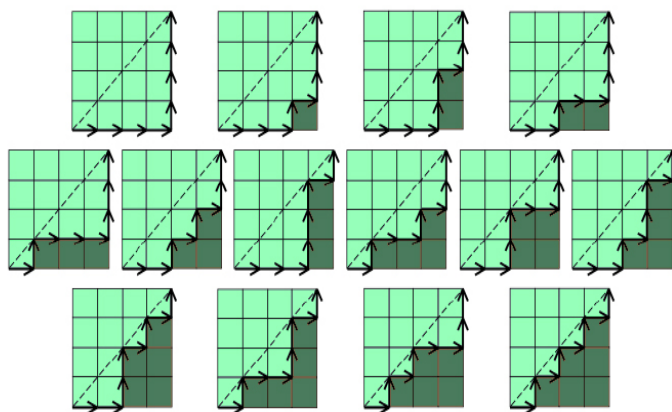
Kompleksnost algoritma. Kreiranje strukture se izvršava u $O(n \cdot k)$ vremenskoj kompleksnosti. Dalje, broj podproblema koji se rješavaju je jednak

$(n+1) \cdot (k+1)$. U svakom rekurzivnom pozivu, izvršava se konstantan broj operacija. Sveukupno, kompleksnost algoritma je jednaka $O(n \cdot k)$.

4.4 Izračunavanje Katalanovih brojeva

Katalanovi brojevi se pojavljuju u mnogim interesantnim (kombinatornim) problemima. Neki od tih problema su sljedeći.

- Broj različitih puteva sa $2n$ koraka na pravougaonoj mreži $n \times n$ koji polaze od donjeg lijevog ugla mreže, tj. koordinate $(n-1, 0)$, pa do gornjeg desnog ugla $(0, n-1)$ uz dodatan uslov da put ne presijeca glavnu dijagonalu (već je može samo eventualno dodirivati) je predstavljen ovim brojevima.



- Broj permutacija skupa $\{1, \dots, n\}$ koji ne sadrže patern 123 (ili bilo koji drugi, dužine 3) su predstavljeni Katalanovim brojevima. Npr. za $n = 3$, sljedeće permutacije zadovoljavaju uslove: 132, 213, 231, 312, 321. Dakle, ima ih ukupno 5.

Prvih nekoliko Katalanovih brojeva su: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ... Može se pokazati da za Katalanove brojeve vrijedi sljedeća rekurzija

$$C(0) = 1, C(n+1) = \sum_{i=0}^n C(i) \cdot C(n-i), n \geq 0.$$

Primijenimo metod brutalne sile (rekurzivno) pri računanju n -tog Katalanovog broja. Implementacija ovog pristupa je data sljedećim kôdom.

```

def catalan(n):
    if n == 0:
        return 1
    catalan_n = 0
    for i in range(n):
        catalan_n += catalan(i) * catalan(n-i-1)
    return catalan_n

```

Kompleksnost algoritma. Faktor grananja rekurzije je (najviše) n . Dubina rekurzije je $n + 1$. Dakle, kompleksnost ovog algoritma je $O(n^n)$, dakle eksponencijalna.

Pokušajmo optimizovati ovaj (direktni) rekurzivni pristup. Iskoristimo princip memoizacije, zbog toga što se svaki podproblem izražunava iznova više puta. U nizovnoj strukturi $Cat(i)$ čuvajmo vrijednost i -tog Katalanovog broja prvi put kada se izračuna. Vrijedi svojstvo optimalne podstrukture: $Cat(n + 1) = \sum_{i=0}^n Cat(i) \cdot Cat(n - i)$. Bazni slučaj je dat sa $Cat(0) = 1$.

Implementacija memoizacije je data sljedećim kôdom.

```

def initialization(n):
    Cat = [ -1 for _ in range(n+1) ]
    return Cat

def c(i, Cat):
    if Cat[i] != -1:
        return Cat[i]
    if i == 0:
        Cat[i] = 1
        return 1

    catalan_n = 0
    for k in range(i):
        catalan_n += catalan_memoization(k, Cat) *
                     catalan_memoization(i-k-1, Cat)

    Cat[i] = catalan_n
    return Cat[i]

```

```

#poziv metode:
n = 10
Cat = initialization(n)

cat_n = catalan_n(n, Cat)
print(cat_n)

```

Kompleksnost algoritma. Dubina rekurzije je (najviše) $n + 1$. U svakoj rekurziji, broj iteracija koji se izvršava je linearne kompleksnosti $O(n)$. Prema tome, kompleksnost algoritma je $O(n) \cdot O(n) = O(n^2)$.

4.5 Još neki zadaci

Posmatrajmo konstrukciju niza brojeva opisanu sljedećom (iterativnom) procedurom.

- Neka je dat niz brojeva:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

- Izbrišemo svaki drugi broj iz skupa, čime dobijamo niz brojeva:

1, 3, 5, 7, 9, 11, 13, 15, 17, 19, ...

- Iz prethodnog niza potom izbrišemo svaki treći broj, čime dobijamo novi niz brojeva: 1, 3, 7, 9, 13, 15, 19, ...
- Nastavljamo proceduru brisanja brojeva iz prethodnog niza tako što brišemo svaki četvrti broj u nizu itd.

U ulazu se zadaje broj n . Ispitati da li je on *srećan*, tj. da li neće biti obrisani prethodnom (iterativnom) konstrukcijom.

Rješenje. U prvoj iteraciji ($i = 1$) će biti obrisani svi brojevi na parnoj poziciji u nizu, tj. oni koji su djeljivi sa $i + 1 = 2$. Ako n nije djeljiv sa

2, preživjeće u ovoj iteraciji, te će se u novom (filtriranom) nizu pojaviti na poziciji $\lceil n/2 \rceil$. Dalje, u narednoj iteraciji ($i = 2$) provjerimo da li je $\lceil n/2 \rceil < i+1 = 3$. Ako je to slučaj, broj n nikad neće biti obrisani u narednim iteracijama, pa prekidamo pretragu, vraćajući rezultat *True*. Inače, brišemo sve elemente novog niza čija je pozicija djeljiva sa $3 = i + 1$. Ako je $\lceil n/2 \rceil$ djeljiv sa 3, n će biti obrisani (prekidamo pretragu, vraćajući rezultat *False*). Inače, broj n preživljava u ovoj iteraciji, te će se nakon operacije brisanja, pojaviti na poziciji $\lceil n/2 \rceil - \lceil n/2/(i+1) \rceil = \lceil n/(i+1) \rceil = \lceil n/3 \rceil$ novonastalog niza. U narednoj iteraciji ($i = 3$), sličnim rezonovanjem prvo provjeravamo da li je $\lceil n/3 \rceil < i+1 = 4$, pa vraćamo rezultat *True*, ukoliko je to slučaj. Potom, provjeravamo da li je $\lceil n/3 \rceil$ djeljiv sa 4. Ako jeste, prekidamo pretragu i vraćamo rezultat *False*. U protivnom, iz trenutnog niza brišemo sve elemente čija je pozicija djeljiva sa 4, pa će se broj n naći na poziciji $\lceil n/4 \rceil$ novonastalog niza. Ovaj proces ponavljamo, za svaku iteraciju, do prekida (u najgorem slučaju gornja granica broja iteracija je n).

Implementacija algoritma je data narednim kôdom.

```
from math import ceil
def lucky_number(n):

    for i in range(1, n): #iteracije
        if i+1 > ceil(n/i):
            return True
        if ceil(n/i) % (i+1) == 0:
            return False
    return True

#poziv funkcije
print(lucky_number(7)) #True
```

Definicija 4.3 Hornerov metod za računanje vrijednosti polinoma.

Neka je dat polinom $c_n x^n + c_{n-1} x^{n-1} + \dots + c_1 x + c_0$ kao i tačka $x \in \mathbb{R}$. Naći vrijednost polinoma u tački x . Napomenimo da je ulaz vezan za polinom dat u obliku niza *coef* gdje *coef*[*i*] predstavlja koeficijent uz monom x^{n-i} , $i \in \{0, \dots, n\}$.

Rješenje. Pogledajmo primjer sljedeće instance: *coef* = [1, -3, 2, -1], $x = 3$. U ovom slučaju, riječ je o polinomu $P(x) = x^3 - 3x^2 + 2x - 1$. Izlaz je

$$P(3) = 5.$$

Naivnim metodom, dakle računanjem vrijednosti x^n običnom petljom se izvršava se u linearnoj $O(n)$ kompleksnosti. Kako imamo n takvih vrijednosti za računanje (koji se potom sabiraju), ukupna kompleksnost ovakvog (naivnog) pristupa je $O(n^2)$.

Hornerovim metodom, računanje vrijednosti polinoma se može izvršiti u linearnoj $O(n)$ kompleksnosti. Ideju algoritma demonstrirajmo na primjeru polinoma iz prethodnog primjera, koji se može napisati kao:

$$((x - 3)x + 2)x - 1.$$

Dakle, u tom slučaju, evaluacija ovako napisanog polinoma bi išla: prvo krećemo od koeficijenta $c_3 = 1$ kojeg množimo sa $x = 3$ – pri tome dobijamo (kumulativnu) vrijednost 3. Dalje, trenutnoj vrijednosti dodajemo vrijednost narednog koeficijenta $c_2 = -3$, te dobijamo vrijednost 0, koju množimo sa $x = 3$, dobijajući opet 0. Ponavljamo korak sa narednim koeficijentom $c_1 = 2$, čime dobijamo vrijednost 2, koju množimo sa $x = 3$, dobijajući 6. Nadalje, posljednji koeficijent $c_0 = -1$ jednostavno dodamo kumulativnoj vrijednosti, odakle dobijamo konačan rezultat, a to je 5.

Gledajući uopšteno, vrijedi:

$$c_n x^n + c_{n-1} x^{n-1} + \dots + c_1 x + c_0 = ((\dots ((c_n x + c_{n-1})x + c_{n-2})x + \dots c_1)x + c_0)$$

Trivijalni slučaj je $n = 0$, kada je $P(x) = c_0$. U tom slučaju, jednostavno vratimo vrijednost c_0 .

Implementacija algoritma Hornerovog metoda je data narednim kôdom.

```
def horner_metod(coef, n, x):

    if n == 0:
        return coef[0]

    # inicijalizacija
    res = coef[n] * x
    for i in range(1, n):
        res = res + coef[i]
        if i < n-1:
            res *= x
```

```

    return res

# poziv funkcije
# Evaluacija:  $x^3 - 3x^2 + 2x - 1$ ,  $x = 3$ 
coef = [1, -3, 2, -1]
x = 3
n = len(coef)
print(horner_metod(coef, n, x))

```

Kompleksnost algoritma. Jasno je da je najintenzivniji dio izvršavanja operacija smješten u `for`-petlju. Svaka iteracija se izvršava u konstantnom vremenu, pa zaključujemo da se algoritam izvršava u linearnom vremenu, tj. $O(n)$.

Zadaci

1. Broj je poluprost (eng. *semiprime*) ako se može predstaviti kao proizvod dva (ne obavezno različita) prosta broja. Npr. 4, 6, 9, 10, ... su neki od poluprostih brojeva. Konstruisati (efikasan) algoritam koji ispituje da li je broj poluprost.
2. Broj je savršen akko je jednak zbiru svojih djelilaca (ne računajući njega samog). Npr. 6 i 28 su savršeni jer je $6 = 1 + 2 + 3$; $28 = 1 + 2 + 4 + 7 + 14$. Napisati program koji ispituje da li je broj savršen korištenjem Eratostenovog sita. Kolika je vremenska složenost algoritma? Uporediti ovaj algoritam sa algoritamskom paradigmom brutalne sile.
3. Broj je ružan akko su mu jedini prosti faktori 2 ili 3 ili 5. Niz prvih ružnih brojeva je dat sa 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... (1 je po konvenciji uključen). U ulazu je dat broj n . Izračunati n -ti ružni broj. Koristiti paradigmu dinamičkog programiranja.
4. Sfenički brojevi su pozitivni cijeli brojevi koji se dobijaju kao proizvod tačno 3 različita prosta faktora. Npr. 30, 42, 66, 70, 78, 102, 105, 110, 114, ... su neki od takvih brojeva. U ulazu je dat broj n . Ispitati da li je on sfenički? *Primjer.* $30 = 2 \cdot 3 \cdot 5$ jeste sfenički, dok $60 = 2^2 \cdot 3 \cdot 5$ nije sfenički.

5. Za broj se kaže da je slobodan od kvadrata ako ga nijedan prosti faktor ne dijeli više od jednom, tj. najveći stepen prostog faktora koji dijeli n je jedan. Prvih nekoliko slobodnih kvadrata brojeva su

1, 2, 3, 5, 6, 7, 10, 11, 13, 14, 15, 17, 19, 21.

Na ulazu je dat broj n . Ispitati da li je on broj slobodnog kvadrata.

6. Neka je dat broj n . Ispitati da li je to *lažni* broj ili ne. Pod lažnim brojem podrazumijevamo složeni broj, čiji je zbir cifara jednak zbiru cifara njegovih različitih prostih faktora.
7. Dat je skup od n elemenata, pronaći broj načina da se on particioniše (Belovi brojevi). Implementirati efikasnu proceduru računanja ovakvih brojeva u zavisnosti od n .

Bibliografija

- [1] T.Cormen, C. Leiserson, R. Rives, Introduction to algorithms, MIT Press, Cambridge, 2001.
- [2] Ognjanovic, Zoran, and Nenad Krdzavac. "Uvod u teorijsko racunarstvo." Fakultet organizacionih nauka, Beograd (2004).
- [3] Živković, Dejan. Uvod u algoritme i strukture podataka. Univerzitet Singidunum, 2010.
- [4] <https://www.geeksforgeeks.org>
- [5] https://adriann.github.io/programming_problems.html
- [6] <http://people.seas.harvard.edu/~cs125/fall14/lec6.pdf>
- [7] Miodrag Živković. Algoritmi. Matematički fakultet, Beograd, 2000
- [8] Steven S. Skiena. The Algorithm Design Manual, Springer
- [9] Jeff Erickson. Algorithms, 2023