Mogućnosti implementacije rekurzivnih algoritama u nastavi programiranja u srednjoj školi

Štruklec, Ivan

Sveučilište u Zagrebu
Prirodoslovno-matematički fakultet
Matematički odsjek

Ivan Štruklec

Mogućnosti implementacije rekurzivnih algoritama u nastavi programiranja u srednjoj školi

Diplomski rad

Voditelj rada:
dr. sc. Goran Igaly

Zagreb, veljača 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____ , predsjednik

2. _____ , član

3. _____ , član

Povjerenstvo je rad ocijenilo ocjenom _____ .

Potpisi članova povjerenstva:

1. _____

2. _____

3. _____

Sadržaj

1. Uvod

1.1. Značaj rekurzije u programiranju

Rekurzija predstavlja ključan koncept u računalstvu i nezaobilaznu tehniku u dizajniranju i analizi algoritama. Ovaj pristup, gdje funkcija poziva samu sebe kako bi riješila manji dio problema, omogućava programerima da pišu jasniji i koncizniji kod za kompleksne zadatke. Rekurzija pruža elegantna rješenja za algoritme čiji su problemi prirodno hijerarhijski strukturirani ili se mogu riješiti ponavljajućim uzorcima, kao što su sortiranje, pretraga, izračun faktorijela ili iteracija kroz strukturalne podatke poput stabala i grafova.

Kroz rekurzivni pristup razvija se duboko razumijevanje kako problem može biti fragmentiran na manje, jednostavnije dijelove. Korištenjem rekurzije, učenici i stručnjaci uče važnost temeljitog razumijevanja problema prije početka programiranja. To ih usmjerava prema analizi slučaja, razvoju osnovnih slučajeva i uvjetima zaustavljanja, koji su ključni elementi u kreiranju rekurzivnih algoritama.

U teoriji algoritama, rekurzija je ključna u strategijama poput "podijeli pa vladaj" gdje kompleksan problem dijelimo na manje potprobleme iste naravi koji su lakši za rješavanje. Algoritmi poput brzog sortiranja (quicksort) i sortiranja spajanjem (Merge sort) temelje se na ovom principu. Također, rekurzija je osnova za brojne tehnike optimizacije poput memoizacije i dinamičkog programiranja, koje povećavaju efikasnost algoritama rješavajući potprobleme samo jednom te čuvajući njihove rezultate za buduću upotrebu.

U kontekstu edukacije, rekurzija nudi važan alat za poučavanje programiranja jer potiče razvoj logičkog mišljenja i razumijevanje kako manji problemi doprinose rješavanju općenitog zadatka. Proces učenja kroz rekurziju učenicima omogućava da analiziraju probleme, identificiraju obrasce i primijene apstrakciju prilikom programiranja. Ove vještine su ključne ne samo za teorijsko razumijevanje informatike već i za praktičnu primjenu u programiranju.

Međutim, rekurzija također donosi izazove, osobito u pogledu analize vremenske složenosti i prostorne efikasnosti. Rekurzivni pozivi zahtijevaju više memorije zbog stoga poziva, a nekontrolirana ili duboka rekurzija može dovesti do preopterećenja stoga ili neefikasnog izvođenja. Stoga je važno ne samo naučiti kako koristiti rekurziju, već i kada je najučinkovitije koristiti, te kako upravljati resursima čiji je kapacitet ograničen. Usporedba rekurzivnih i iterativnih rješenja omogućava učenicima da ocijene i odaberu najprikladniji pristup za određeni problem.

U konačnici, rekurzija je nezamjenjiv alat u arsenalu programera. Precizno i efikasno korištenje rekurzije je pokazatelj profesionalnosti i kompetencije u području računalnih znanosti. Uključivanje rekurzivnih algoritama u programe srednjoškolskog obrazovanja ne samo da obogaćuje nastavni plan već i osposobljava učenike s alatima koji su im potrebni za rješavanje izazovnih zadataka u njihovoj budućoj profesionalnoj karijeri.

1.2. Kratak pregled pristupa učenju programiranja u srednjoj školi

Učenje programiranja u srednjim školama u Hrvatskoj odvija se kroz višeslojni pristup, s ciljem pružanja temeljne digitalne pismenosti, ali i razvijanja naprednih vještina kod učenika koji pokazuju zanimanje za informatiku i računalne znanosti. Nastavni plan i program su usklađeni s nacionalnim okvirnim kurikulumom i standardima Europske unije vezanim za digitalne kompetencije.

Obavezni predmeti:

Informatika kao obavezni predmet u nižim razredima srednje škole usmjerena je na osnove korištenja računala, obrade teksta, tablične proračune, izradu prezentacija, te osnove algoritamskog razmišljanja i programiranja, često u programskom jeziku kao što je Python ili kroz vizualne programske jezike poput Scratcha. Cilj ovog segmenta je pružanje opće informatičke pismenosti svim učenicima.

Izborni predmeti:

Za one učenike koji pokazuju dodatni interes za informatiku, postoje izborni predmeti koji pružaju dublje razumijevanje računalnih znanosti, uključujući programiranje, razvoj web aplikacija, baze podataka i mrežne tehnologije. Kroz ove predmete, učenici istražuju temeljne koncepte struktura podataka, algoritama, objektno orijentiranog programiranja i razvoja softvera.

Strukovne škole:

U okviru strukovnih škola, učenici imaju priliku specijalizirati se kroz praktično orijentirane programe koji uključuju programiranje kao dio obrazovanja za buduće informatičare, tehničare i inženjere. Ovdje se naglasak stavlja na razvoj praktičnih vještina, a programiranje često uključuje rad s konkretnim tehnologijama i projektima koji mogu uključivati web dizajn, programiranje mikro kontrolera, kreiranje mobilnih aplikacija i slično.

Natjecanja i dodatne aktivnosti:

Učenicima u Hrvatskoj su dostupna razna natjecanja iz informatike i programiranja kao što su “Croatian Makers” liga, Državno natjecanje iz informatike i slične aktivnosti koje podupiru razvoj teorijskog i praktičnog znanja.

Ove aktivnosti su važne za motivaciju učenika te kao prilika za primjenu naučenog u konkurencijskom okruženju.

Fokus na probleme i projektne zadatke:

Pristup učenju programiranja kroz probleme i projekte je sve prisutniji u hrvatskim srednjim školama. Kroz rješavanje stvarnih problema i rad na projektima, učenici razvijaju kritičko razmišljanje, sposobnost analize problema i timski rad, što su sve vještine koje su visoko cijenjene u tehnološkom sektoru.

Digitalna transformacija nastave:

Na kraju, važan segment u učenju programiranja jest težnja za stalnim unaprjeđenjem metodologije poučavanja, koja uključuje implementaciju digitalnih alata, platformi za e-učenje te korištenje internetskih resursa za podršku nastavnom procesu.

Učenje programiranja u Hrvatskoj se stoga konstantno prilagođava kako bi se osigurala relevantnost i aktualnost s obzirom na brze promjene koje karakteriziraju područje računalnih znanosti i tehnologije.

1.3. Analiza pojavljivanja pojma rekurzije u kurikulumu nastavnog predmeta Informatika

Odlukom o donošenju kurikuluma za nastavni predmet Informatike za osnovne škole i gimnazije u Republici Hrvatskoj od 12. veljače 2018. godine donesen je kurikulum za nastavni predmet Informatika za osnovne škole i gimnazije u Republici Hrvatskoj.

U ovom kurikulumu se na nekoliko mjesta spominje pojam rekurzije i to u domeni B – Računalno razmišljanje i programiranje.

Osnovna škola

Rekurzija se spominje u ishodu B.8.3 za osmi razred osnovne škole sljedećim izričajem:

"Nakon osme godine učenja predmeta Informatika u domeni Računalno razmišljanje i programiranje učenik prepoznaje i opisuje mogućnost primjene rekurzivnih postupaka pri rješavanju odabranih problema te istražuje daljnje mogućnosti primjene rekurzije."

U razradi ishoda navodi se: "Učenik promatra i opisuje zajednička obilježja nekih rekurzivnih fenomena te poznaje korake rekurzivnoga postupka. Analizira odabrani problem te u njemu identificira osnovi slučaj rekurzije te način rekurzivnoga pozivanja. Pronalazi i predlaže rješenje (grafički, riječima/uputama) odabranoga problema primjenom rekurzivnoga postupka. Učenik istražuje i

predlaže primjere problema pri čijem se rješavanju može primijeniti rekurzivni postupak."

U preporukama za ostvarenje odgojno-obrazovnih ishoda piše:

- "Promatrati neke pokazne grafičke primjere (npr. trokut Sierpinskog, Kochova pahuljica, ...) te diskutirati o njihovim obilježjima. Pokazati različite primjere rekurzivnih fenomena iz svakodnevnoga života te raspravljati o njihovim mogućim zajedničkom obilježjima.
- Koristiti se konkretnim modelima (Matrjoške – ruske lutke, tornjevi Hanoa, primjeri iz stvarnoga života –otoci, jezera, vulkani, dijeljenje stanica...) ili grafičkim modelima (padajući prozori) pri demonstriranju i analizi rekurzivnoga postupka. Opisati i pokazati osnovne korake rekurzivnoga postupka."
- Ovime je dan okvir očekivanja razumijevanja pojma rekurzije na toj razini, odnosno namjera je da učenici shvate da postoje samo ponavljajuće strukture te da u grafičkom okruženju (programski jezik Logo) realiziraju neke od tih struktura.

Srednja škola

S obzirom da se u radu bavimo analizom mogućnosti korištenja rekurzije u nastavi programiranja u srednjoj školi, pogledajmo ishode učenja iz srednje škole koji se bave rekurzijom. U ovom predmetnom kurikulumu četiri su ishoda učenja u kojima se spominje rekurzija. Tri ishoda su gotovo identični dok se četvrti donekle razlikuje. Nažalost, zbog načina označavanja ishoda u kojemu kratka oznaka ishoda nažalost ne znači i njegovu identifikacijsku oznaku, u svrhu snalaženja morat ćemo navesti kompletan (dugačak) naziv programa u kojemu se ishod javlja kako bismo ga jedinstveno odredili.

U srednjoj školi se pojam rekurzije prvi puta spominje u drugom razredu prirodoslovno-matematičke gimnazije (105 sati godišnje):

B.2.2

Nakon druge godine učenja predmeta Informatika u domeni Računalno razmišljanje i programiranje učenik u zadanome problemu uočava manje cjeline, rješava ih te ih potom integrira u jedinstveno rješenje problema.

Razrada ishoda:

Raščlanjuje zadani problem na manje funkcionalne cjeline koje opisuje. Određuje ulazne i izlazne parametre funkcionalnih cjelina. Razlikuje globalne, lokalne i formalne parametre. Funkcionalne cjeline rješava u konkretnome

programskom jeziku ispravno se koristeći programskim funkcijama i integrira ih u cjelovito rješenje problema. Koristi se jednostavnom rekurzivnom funkcijom.

Preporuke za ostvarenje odgojno-obrazovnih ishoda:

- Analiza i rastavljanje većega problema iz stvarnoga života na manje cjeline (poznati matematički ili fizikalni problemi).
- Korištenje kojim vizualnim alatom za prikaz cjelina, odnosa među njima, ulaznih i izlaznih podataka.
- Pohranjivanje svojih funkcija za rješavanje nekoga potproblema u svoju biblioteku funkcija.
- Na jednostavnim primjerima uvesti pojam rekurzivne funkcije. Uvidjeti mogućnosti korištenja rekurzivnim funkcijama, uočiti rekurzivnost u definiciji nekih problema (Fibonaccijevi brojevi).
- Rješavanje problemskoga zadatka samostalno i u timu.

Uz to se pojam rekurzije spominje još u tri ishoda koji su gotovo identični odnosno razlikuju se samo manjim dijelom. Navest ćemo ih u cijelosti, s time da su kosim slovima ("*italic*") označeni dijelovi koji nisu identični u svim ishodima.

4. RAZRED ILI 4. GODINA UČENJA (OPĆE GIMNAZIJE)

B.4.1

Nakon treće godine (pogreška, op. I.Š.) učenja predmeta Informatika u srednjoj školi u domeni Računalno razmišljanje i programiranje učenik rješava problem primjenjujući rekurzivnu funkciju.

Razrada ishoda:

Opisuje osnovne elemente rekurzivnoga postupka. Zapisuje matematički opisanu rekurzivnu funkciju u programskome jeziku. Uočava rekurzivnost u danome problemu, određuje rekurzivnu relaciju i uvjet prekida te realizira rekurzivnu funkciju u programskome jeziku. Procjenjuje efikasnost rekurzivnoga rješenja. Ovisno o problemu odabire rekurzivno odnosno induktivno rješenje.

Preporuke za ostvarenje odgojno-obrazovnih ishoda:

- Učenici pronalaze primjere vizualnih rekurzija poput zrcala koja se ogledaju jedno u drugom.
- Odrediti rekurzivnu relaciju na jednostavnijim problemima kod kojih se lako uočava rekurzivnost, primjerice odrediti zbroj prvih n članova reda: $1 - 2 + 3 - 4 \ldots$
- Vizualizira rekurziju s jednostavnim grafičkim elementima.

- Analizirati neke jednostavne primjere poput Fibonaccijevih brojeva, kamata, zbroja i sl. Skrenuti pozornost na to da u nekim problemima rekurzivni postupci nisu učinkoviti (Fibonaccijevi brojevi).
- Crtanje rekurzivnih crteža (fraktali).

3. RAZRED PRIRODOSLOVNO-MATEMATIČKE GIMNAZIJE (70 SATI GODIŠNJE)

B.3.3

Nakon treće godine učenja predmeta Informatika u srednjoj školi u domeni Računalno razmišljanje i programiranje učenik rješava problem primjenjujući rekurzivnu funkciju.

Razrada ishoda:

Opisuje osnovne elemente rekurzivnoga postupka. Zapisuje matematički opisanu rekurzivnu funkciju u programskome jeziku. Uočava rekurzivnost u danome problemu, određuje rekurzivnu relaciju i uvjet prekida te realizira rekurzivnu funkciju u programskome jeziku. Procjenjuje efikasnost rekurzivnoga rješenja. Ovisno o problemu odabire rekurzivno odnosno induktivno rješenje.

Preporuke za ostvarenje odgojno-obrazovnih ishoda:

- Učenici pronalaze primjere vizualnih rekurzija poput zrcala koja se ogledaju jedno u drugom.
- Odrediti rekurzivnu relaciju na jednostavnijim problemima kod kojih se lako uočava rekurzivnost, primjerice odrediti zbroj prvih n članova reda: $1 - 2 + 3 - 4. \ldots$
- *Vizualizira rekurziju s jednostavnim grafičkim elementima.*
- Analizirati neke jednostavne primjere poput Fibonaccijevih brojeva, kamata, zbroja i sl. Skrenuti pozornost na to da u nekim problemima rekurzivni postupci nisu učinkoviti (Fibonaccijevi brojevi).
- *Crtanje rekurzivnih crteža (fraktali).*

3. RAZRED PRIRODOSLOVNO-MATEMATIČKE GIMNAZIJE (105 SATI GODIŠNJE)

B.3.3

Nakon treće godine učenja predmeta Informatika u domeni Računalno razmišljanje i programiranje učenik rješava problem primjenjujući rekurzivnu funkciju.

Razrada ishoda:

Opisuje osnovne elemente rekurzivnoga postupka. Zapisuje matematički opisanu rekurzivnu funkciju u programskome jeziku. Uočava rekurzivnost u danome problemu, određuje rekurzivnu relaciju i uvjet prekida te realizira rekurzivnu funkciju u programskom jeziku. Procjenjuje efikasnost rekurzivnoga rješenja. Ovisno o problemu odabire rekurzivno odnosno induktivno rješenje. Uočava sporost rekurzije u nekim vrstama problema te koristi se mogućnostima pohranjivanja međurezultata (primjenjuje tehniku memoizacije).

Preporuke za ostvarenje odgojno-obrazovnih ishoda:

- Učenici pronalaze primjere vizualnih rekurzija poput zrcala koja se ogledaju jedno u drugom.
- Odrediti rekurzivnu relaciju na jednostavnijim problemima kod kojih se lako uočava, primjerice odrediti zbroj prvih n članova reda: $1 - 2 + 3 - 4. \ldots$
- Analizirati neke jednostavne primjere poput Fibonaccijevih brojeva, kamata, zbroja i sl. Skrenuti pozornost na to da u slučaju nekih problema rekurzivni postupci nisu učinkoviti (Fibonaccijevi brojevi).
- *Primijeniti kornjačinu grafiku za crtanje rekurzivnih crteža (fraktali).*
- *Korištenje memoizacije u slučaju »sporih« rekurzija (omotači).*

Ishodi B.4.1 za 4. RAZRED ILI 4. GODINA UČENJA (OPĆE GIMNAZIJE) i B.3.3 za 3. RAZRED PRIRODOSLOVNO-MATEMATIČKE GIMNAZIJE (70 SATI GODIŠNJE) su identični. Potpuno se podudaraju i razrada ishoda i preporuke za ostvarenje odgojno-obrazovnih ishoda. Čak je u prepisivanju napravljena pogreška tako da u ishodu B.4.1 rečenica započinje sa "Nakon treće godine učenja" umjesto "Nakon četvrte godine učenja". Odnosno šteta je što oznake ishoda učenja nemaju ulogu identifikatora što bi umnogome olakšalo usporedivost. Ovako posve isti ishodi imaju različite oznake što otežava snalaženje.

Ishodi B.3.3 za 3. RAZRED PRIRODOSLOVNO-MATEMATIČKE GIMNAZIJE (70 SATI GODIŠNJE) i B.3.3 za 3. RAZRED PRIRODOSLOVNO-MATEMATIČKE GIMNAZIJE (105 SATI GODIŠNJE) se razlikuju utoliko što je u ishodu za program od 105 sati godišnje dodana rečenica *"Uočava sporost rekurzije u nekim vrstama problema te koristi se mogućnostima pohranjivanja međurezultata (primjenjuje tehniku memoizacije)."*

Također se neznatno razlikuju i preporuke za ostvarenje ishoda. U preporuci za 70 sati godišnje navode se preporuke:

"Vizualizira rekurziju s jednostavnim grafičkim elementima." i *"Crtanje rekurzivnih crteža (fraktali)."* U preporuci za 105 sati godišnje navodi se *"Primijeniti*

kornjačinu grafiku za crtanje rekurzivnih crteža (fraktali)." i *"Korištenje memoizacije u slučaju »sporih« rekurzija (omotači)."*

Drugim riječima, ishod B.3.3 za za 3. RAZRED PRIRODOSLOVNO-MATEMATIČKE GIMNAZIJE (105 SATI GODIŠNJE) je od ova tri ishoda najsveobuhvatniji pa ćemo analizu pokrivenosti ishoda primjerima iz rada napraviti na ovom ishodu.

S obzirom da se rad bavi mogućnostima primjene rekurzije u nastavi programiranja, osvrnut ćemo se na mogućnosti ostvarivanja ovih ishoda te na dodirne točke s preporukama za ostvarivanje ishoda. Stoga ponovimo relevantne dijelove ishoda i preporuke s kojima primjeri u ovom radu imaju dodirnih točaka.

B.2.2:

Ishodi

- Raščlanjuje zadani problem na manje funkcionalne cjeline koje opisuje.
- Određuje ulazne i izlazne parametre funkcionalnih cjelina.
- Razlikuje globalne, lokalne i formalne parametre.
- Funkcionalne cjeline rješava u konkretnome programskom jeziku ispravno se koristeći programskim funkcijama i integrira ih u cjelovito rješenje problema.
- Koristi se jednostavnom rekurzivnom funkcijom.

Preporuke

- Na jednostavnim primjerima uvesti pojam rekurzivne funkcije. Uvidjeti mogućnosti korištenja rekurzivnim funkcijama, uočiti rekurzivnost u definiciji nekih problema (Fibonaccijevi brojevi).

B.3.3 i B.4.1:

Ishodi

- Opisuje osnovne elemente rekurzivnoga postupka.
- Zapisuje matematički opisanu rekurzivnu funkciju u programskome jeziku.
- Uočava rekurzivnost u danome problemu, određuje rekurzivnu relaciju i uvjet prekida te realizira rekurzivnu funkciju u programskome jeziku.
- Procjenjuje efikasnost rekurzivnoga rješenja.
- Ovisno o problemu odabire rekurzivno odnosno induktivno rješenje.

Preporuke:

- Odrediti rekurzivnu relaciju na jednostavnijim problemima kod kojih se lako uočava rekurzivnost, primjerice odrediti zbroj prvih n članova reda: $1 - 2 + 3 - 4 \ldots$.
- Vizualizirati rekurziju s jednostavnim grafičkim elementima.
- Analizirati neke jednostavne primjere poput Fibonaccijevih brojeva, kamata, zbroja i sl. Skrenuti pozornost na to da u nekim problemima rekurzivni postupci nisu učinkoviti (Fibonaccijevi brojevi).

2. Algoritam

2.1. Pojam i povijest algoritma

Algoritam je u osnovi skup instrukcija ili definiranih koraka koji se primjenjuju za izvođenje zadatka ili rješavanje problema. Možemo ga smatrati poput recepta u kuhanju gdje svaki korak detaljno opisuje slijed radnji potrebnih za stvaranje jela. U kontekstu računalstva, algoritam predstavlja detaljan proces koji vodi računalo kroz seriju definiranih operacija kako bi stiglo do željenog rezultata.

Povijest algoritama seže daleko u prošlost, puno prije nego što su izumljena moderna računala. Jedan od najstarijih poznatih algoritama je Euklidov algoritam za pronalazak najvećeg zajedničkog djelitelja dva broja, koji datira još iz 300. pr. Kr. To je jasan pokazatelj da su ljudi razvijali sistematizirane metode za rješavanje problema mnogo prije digitalne ere.

U srednjem vijeku, pojam algoritma često se povezivao s matematikom i računanjem. Riječ „algoritam“ dolazi od imena perzijskog matematičara Al-Khwarizmija, koji je napisao "Kitab al-jabr wa'l-muqabala" iz kojeg potječe i riječ „algebra“. Njegovi radovi, koji uključuju metode za rješavanje linearnih i kvadratnih jednadžbi, predstavljaju ranu manifestaciju algoritamskog razmišljanja.

S pojavom računalne znanosti u 20. stoljeću, algoritmi su postali nezaobilazni alat za dizajniranje softverskih programa i rješavanje kompleksnih računskih problema. S vremenom su se razvijali u sve sofisticiranije forme, iz čega su proizašle i specijalizirane grane kao što su algoritmi sortiranja, pretrage, kriptografija, optimizacija i mnogi drugi.

U današnje vrijeme, algoritmi su temelj gotovo svakog aspekta modernog života, od pretraživanja interneta, preko analize velikih podataka, do vođenja autonomnih vozila. Nastava algoritama u školama nije samo pitanje pripreme učenika za tržište rada, već i ključan element u razvoju logičkog razmišljanja koje je primjenjivo u širokom spektru životnih situacija.

Od primitivnih alata za računanje poput abakusa pa sve do složenih računalnih algoritama današnjice, algoritmi su oduvijek bitan element ljudske težnje za redom, razumijevanjem i kontrolom okoline. Svojom svestranom primjenom, algoritmi su postali jedan od temeljnih stupova u izgradnji modernog društva.

2.2. Klasifikacija algoritama prema implementaciji

Kada razmatramo različite vrste algoritama prema njihovoj implementaciji, nailazimo na temeljne kategorije koje određuju način na koji algoritmi rješavaju zadatke. Ova razvrstavanja uzimaju u obzir strukturu algoritma, način na koji

algoritam pristupa problemu, te kako dolazi do konačnog rješenja. Evo glavnih kategorija.

Rekurzivni i iterativni algoritmi

Prva ključna podjela je između rekurzivnih i iterativnih algoritama.

Rekurzivni algoritmi rješavaju problem pozivajući sami sebe kao podfunkciju. Ovi algoritmi se oslanjaju na bazne slučajeve za prekid rekurzije i obično su povezani s elegantnim rješenjima koja su koncizna i često jednostavna za razumijevanje, no mogu voditi do problema sa stogom memorije ukoliko nisu dobro upravljani.

Iterativni algoritmi ponavljaju blokove koda kroz strukture petlji kao što su `for` ili `while` bez pozivanja iste funkcije unutar nje same. Iako su često vezani za veću učinkovitost i manju potrošnju memorije, kodovi mogu biti manje intuitivni i teži za razumijevanje kada se nose s kompleksnijim problemima.

Logički algoritmi

Logički algoritmi koriste formalnu logiku za rješavanje problema. Ovi algoritmi obuhvaćaju sistematizirane postupke dizajnirane da slijede stroga pravila zaključivanja i često su korišteni u programima koji se zasnivaju na umjetnoj inteligenciji.

Serijski i paralelni algoritmi

Serijski algoritmi su oni koji izvode instrukcije jednu za drugom, ili serijalno. Svaki korak mora biti završen prije nego što sljedeći korak počne.

Paralelni algoritmi su dizajnirani za izvršavanje na višeprocesorskim sustavima, gdje više operacija može biti izvedeno istodobno. Paralelizacija može značajno smanjiti vrijeme izvođenja i povećati efikasnost, naročito kod obrade velikih podataka ili kod kompleksnih izračunavanja.

Deterministički i stohastički algoritmi

Deterministički algoritmi daju predvidive, točne i iste rezultate za iste ulazne podatke svaki put kada su pokrenuti. Logički koraci koji se slijede su čvrsto definirani i nema varijacije u razvoju rješenja.

Stohastički algoritmi ili probabilistički algoritmi koriste neformalno elemente slučajnosti ili nasumične odluke u svojem procesu. Ovi algoritmi su korisni kod optimizacijskih problema gdje je teško pronaći apsolutno rješenje, i gdje je cilj pronaći "dobra" ili "dovoljno dobra" rješenja unutar prihvatljivog vremenskog okvira, primjerice, koristeći se genetskim algoritmima ili metodom Monte Carlo.

Svaki od ovih pristupa ima svoju svrhu i optimalno područje upotrebe. Izbor pravog algoritma za određeni problem može imati znatne implikacije na performanse, kvalitetu rješenja i resurse potrebne za izvođenje, što ističe važnost algoritamskog obrazovanja kao osnove za kritičko razumijevanje i implementaciju informatičkih sistema.

2.3. Klasifikacija algoritama prema metodologiji dizajna

U nastavi programiranja, posebno u kontekstu algoritmičkog razmišljanja, važno je podučavati različite strategije dizajniranja algoritama. U ovom radu fokusirat ćemo se na tri popularna pristupa: "podijeli pa vladaj", "pohlepni pristup" i "backtracking", koji su temeljni za razumijevanje kako se složeni problemi mogu rastaviti i riješiti sistematičnim metodama.

Podijeli pa vladaj (“Divide and Conquer“)

"Podijeli pa vladaj" je pristup koji rješava problem tako što ga razbija na manje potprobleme, rješava te potprobleme (često rekurzivno), a zatim kombinira njihova rješenja za formiranje rješenja originalnog problema.

Ključne značajke:

- Problemi se dijele na manje verzije istog problema sve dok se ne dođe do potproblema koji su dovoljno mali da se rješavaju direktno.
- Mehanizam rada obuhvaća tri koraka: podijeli, osvoji i kombiniraj.
- Efikasnost ovog pristupa proizlazi iz sposobnosti da se problemi rješavaju paralelno i učinkovito.
- Primjeri uključuju algoritme za brzo sortiranje (quicksort) i sortiranje spajanjem (mergesort).

Pohlepni pristup (“Greedy algorithm“)

Pohlepni algoritmi rješavaju probleme tako što u svakom koraku biraju lokalno optimalno rješenje u nadi da će konačno rješenje biti globalno optimalno. Ova metoda obično se primjenjuje u problemima optimizacije.

Ključne značajke:

- Jednostavna za razumijevanje i implementirati.
- Algoritam ne preispituje svoje odluke, već se fokusira na trenutno najbolje rješenje.
- Može biti vrlo efikasan, ali ne garantira uvijek globalno optimalno rješenje.
- Primjeri primjene su algoritam za problem najkraćeg razapinjajućeg stabla težinskog grafa (Primov i Kruskalov algoritam) i algoritam za problem raspodjele kovanica.

Unatražni algoritam ("backtracking")

Unatražni algoritam podrazumijeva tehniku rješavanja problema koja se koristi kada se traži više rješenja (kao što su sve permutacije, kombinacije ili raspon rješenja). Kada backtracking algoritam pronađe da trenutno rješenje nije valjano ili da ne vodi prema cilju, on se "vraća unatrag" kako bi isprobao drugi put.

Ključne značajke:

- Koristi se za probleme gdje ima puno mogućih rješenja i kada treba istražiti sve potencijalne opcije.
- Algoritam se vraća unatrag kad utvrdi da ne postoji mogućnost daljnjeg nastavka, poništavajući posljednje korake i pokušavajući alternativni put.
- Može biti vremenski zahtjevan jer zahtijeva ispitivanje svih mogućih konfiguracija, ali je vrlo koristan za kompleksne probleme za koje se ne zna efikasnija metoda, kao što je problem trgovačkog putnika (Travelling Salesman Problem, TSP).
- Primjeri uključuju rješavanje problema osam kraljica, prolazak kroz labirint, ili pronalazak svih mogućih rješenja za sudoku.

Svaki od ovih pristupa ima svoje specifične prednosti i primjene, izbor pristupa ovisi o prirodi problema i željenim performansama algoritma. Poticati učenike prepoznavati koji se pristup najbolje primjenjuje na određenu situaciju je vitalno za njihovo dublje razumijevanje računarskih znanosti i programiranja.

2.4. Klasifikacija algoritama prema složenosti

Analiza složenosti algoritma ključna je komponenta informacijskih znanosti jer pruža uvid u to koliko će brzo ili polako algoritam izvesti zadatak s obzirom na veličinu ulaznih podataka. Vremenska složenost algoritma opisuje vezu između broja operacija potrebnih za izvođenje algoritma i veličine ulaza n.

Za opisivanje “najgoreg slučaja“ izvršavanja algoritama, koristi se poseban zapis, tzv. “O notacija“, gdje ispred zagrade stoji veliko slovo O a unutar zagrade je navedena konkretna klasa složenosti. Evo najčešćih klasifikacija složenosti algoritama:

Linearna $O(n)$

Algoritmi s linearnom složenost u obavljaju broj operacija koji je direktno proporcionalan veličini ulaznih podataka. Primjer takvog algoritma može biti jednostavna linearna pretraga, gdje algoritam mora pregledati svaki element ulazne liste kako bi našao željeni element.

Logaritamska $O(\log(n))$

Algoritmi s logaritamskom složenost smanjuju skup podataka koji se obrađuje u svakom koraku, često dijeljenjem skupa na pola. Klasični primjer je binarna pretraga, koja nalazi element u sortiranom nizu tako što na svakom koraku uspoređuje srednji element s ciljanim vrijednošću te odbacuje polovicu elemenata.

Polinomijalna

Unutar polinomijalne složenosti postoji nekoliko potkategorija, uključujući:

- Linearna $O(n)$
- Kvadratna $O(n^2)$
  Primjer algoritma s takvom složenošću je sortiranje zamjenom susjednih elemenata (Bubble sort), gdje se za svaki od n elemenata može izvršiti najviše n usporedbi.
- Kubična $O(n^3)$
  Primjer može biti algoritam za množenje matrica ili neki problemski algoritmi sa trostrukim petljama.

Eksponencijalna $O(2^n)$

Ovi algoritmi pokazuju rast broja operacija koji raste eksponencijalno u odnosu na veličinu ulaza. Primjeri uključuju razne algoritme koji izvršavaju rekurzivno pretraživanje svih mogućih rješenja, kao što je problem trgovačkog putnika riješen metodom "probaj sve mogućnosti". Eksponencijalni algoritmi postaju izuzetno neefikasni s povećanjem veličine ulaznih podataka.

Faktorijalna $O(n!)$

Faktorijalna složenost je još jedan primjer algoritama s visokom složenošću, gdje broj operacija raste sa faktorijelom veličine ulaznih podataka. Problem trgovačkog putnika kada se rješava putem enumeracije svih permutacija također pada pod ovu kategoriju.

Razumijevanje i pravilna klasifikacija algoritama prema njihovoj vremenskoj složenosti omogućava programerima i inženjerima da biraju ili dizajniraju algoritme koji su najprikladniji za dani problem, uzimajući u obzir ograničenja resursa i zahtjeve izvođenja. Odabir pravog algoritma za zadani problem može dramatično utjecati na performanse aplikacije u stvarnom svijetu, što čini ovu temu izuzetno važnom u edukaciji i praksi računalstva.

3. Rekurzivni algoritmi

3.1. Definicija rekurzije i rekurzivnog algoritma

Rekurzija je proces u kojem funkcija ili procedura u računalnom programu poziva sebe direktno ili indirektno. Ovo je jedan od temeljnih koncepata u računalstvu, posebno u teoriji algoritama, koji se koristi za rješavanje problema koje je moguće podijeliti u manje, slične potprobleme. Rekurzivni algoritmi omogućavaju programerima i dizajnerima algoritama da izraze rješenje kompleksnog problema na jednostavan i elegantan način.

3.2. Osnovne komponente rekurzije

Osnovne komponente rekurzivne definicije uključuju:

Osnovni slučaj (ili slučajevi)

Osnovni slučaj je ključan element rekurzivnog algoritma koji omogućava zaustavljanje rekurzije. To je specifičan uvjet koji ne uključuje rekurzivni poziv i koji rješava problem bez daljnjeg raspoređivanja na manje potprobleme. Bez osnovnog slučaja, rekurzivni algoritam bi nastavio s pozivanjem samoga sebe u nedogled, što bi uzrokovalo pogrešku prekoračenja stoga poziva (engl. stack overflow error).

Rekurzijsko pravilo (ili pravila)

Rekurzijsko pravilo definira kako se problem raspoređuje na manje dijelove koji se dalje rješavaju rekurzivnim pozivima. Ideja je smanjiti veličinu problema svakim rekurzivnim pozivom, sve dok se ne dosegne osnovni slučaj.

Smjer rekurzije

Smjer rekurzije odnosi se na redoslijed kojim se rekurzivni pozivi odvijaju i rješavaju, te kako se kombiniraju kako bi se došlo do konačnog rješenja problema. U rekurzivnim algoritmima, ovaj proces može biti ili “prema dolje” (od početnog problema prema osnovnim slučajevima) ili “prema gore” (od osnovnih slučajeva vraćajući se do početnog problema).

Stog poziva (Call Stack)

Iako stog poziva nije formalno dio rekurzije, bitno ga je spomenuti. Kad se tijekom izvođenja programa pozove funkcija, informacije o tom pozivu, uključujući parametre i lokalne varijable, stavljaju se na stog poziva. Ako ta funkcija pozove samu sebe (ili neku drugu funkciju), na stog se dodaje novi skup informacija. Svaki put kad se dostigne osnovni slučaj, funkcija završava svoje izvođenje te se njezin skup informacija uklanja sa stoga, vraćajući kontrolu pozivu koji je iznad nje na stogu.

Stog poziva kritičan je za rekurziju jer omogućuje funkciji da "zapamti" gdje se nalazila prije nego što je pozvala samu sebe. To omogućava funkciji da se nastavi izvršavati sa točke zaustavljanja jednom kad se riješe svi rekurzivni pozivi.

3.3. Vrste rekurzija i njihove ključne značajke

Rekurzija je široko korištena u programiranju i može se kategorizirati na različite načine temeljem njene strukture i načina na koji funkcija pristupa rješavanju problema. Ključne značajke različitih vrsta rekurzija pomažu u razumijevanju različitih scenarija u kojima se određene vrste rekurzija mogu primijeniti.

Linearna rekurzija

Kod linearne rekurzije, svaki rekurzivni poziv generira najviše jedan daljnji rekurzivni poziv. Ovo je jednostavan oblik rekurzije koji se lako prati i analizira. Primjeri uključuju izračunavanje faktorijela ili rekurzivnu implementaciju pronalaska n-tog člana Fibonaccijevog niza.

Ključne značajke:

- Jednostavna za razumijevanje i implementaciju.
- Osnovni slučaj dobro je definiran.
- Stog poziva linearno raste s brojem rekurzivnih poziva.

Višestruka rekurzija

Kod višestruke rekurzije, jedan poziv može generirati više sljedećih rekurzivnih poziva. Česti primjeri uključuju različite algoritme sortiranja poput quicksort, gdje svaki poziv dovodi do rekurzivnog pozivanja.

Ključne značajke:

- Kompleksnija struktura stoga poziva zbog razgranatosti.
- Može biti zahtjevna za analizu vremenske složenosti.
- Osnovni slučajevi moraju biti pažljivo definirani kako bi se izbjegla prekomjerna rekurzija.

Repna rekurzija

Repna rekurzija događa se kada je rekurzivni poziv posljednja operacija koju funkcija izvršava prije nego što vrati vrijednost. Repna rekurzija je posebno zanimljiva jer se može optimizirati u tzv. repni poziv, što znači da se stog poziva ne širi dalje.

Ključne značajke:

- S obzirom da je rekurzivni poziv posljednja operacija, može se optimizirati kako bi se koristio manji broj podataka koje je potrebno pohranjivati na stogu.
- Pomaže u smanjenju rizika od prekoračenja stoga.
- Osnovni slučaj mora biti dobro definiran kako bi se osiguralo pravilno zaustavljanje.

Indirektna rekurzija

Indirektna rekurzija se pojavljuje kada funkcija A poziva funkciju B, koja zatim poziva funkciju A (mogu postojati i duži lanci pozivanja). Ovaj oblik rekurzije može biti nešto teže pratiti zbog njene cikličke naravi.

Ključne značajke:

- Zahtijeva pažljivo planiranje za izbjegavanje beskonačnih petlji.
- Može biti teže razumjeti i analizirati ovu vrstu rekurzije, posebno za početnike.
- Osnovni slučajevi su ključni za svaku funkciju unutar lanca poziva.

Složena (ili ugniježđena) rekurzija

Složena rekurzija nastaje kada rekurzivni pozivi nisu direktni niti linearni, već rekurzivne funkcije pozivaju jedna drugu u složenom uzorku. Primjer može biti rekurzivno računanje određenih matematičkih funkcija koje zahtijevaju višestruke rekurzije.

Ključne značajke:

- Može dovesti do brze ekspanzije stoga poziva.
- Ovu vrstu rekurzije teže je pratiti i optimizirati.
- Precizno definiranje osnovnih slučajeva od ključne je važnosti.
- Svaka od ovih vrsta rekurzija ima posebnu primjenjivost ovisno o problemu koji se rješava. Poznavanje različitih vrsta rekurzija omogućava programerima da biraju pravi alat za određeni problem te razvijaju učinkovite algoritme.

4. Primjeri iz udžbenika informatike u srednjim školama

Rekurzivni algoritmi predstavljaju ključni alat u disciplinama računalstva, pružajući elegantna i efikasna rješenja za širok spektar problema. U nastavku slijede nekoliko osnovnih rekurzivnih primjera u Pythonu iz srednjoškolskih udžbenika informatike.

4.1. Rekurzivno izračunavanje faktorijela broja n

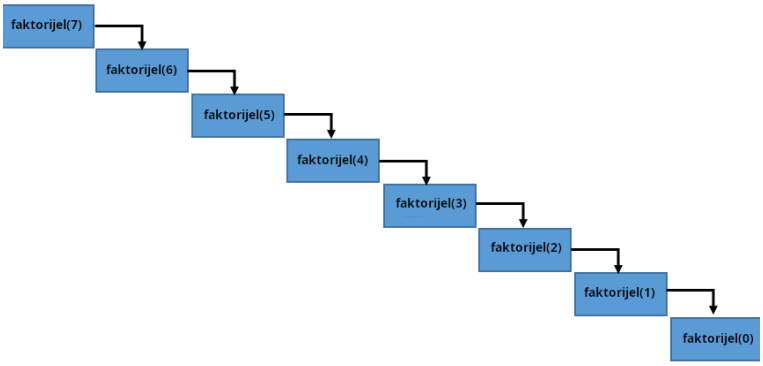
Ovaj primjer iz udžbenika "#mojportal8 – udžbenik iz informatike u osmom. razred osnovne škole: Babić, Bubica, Leko, Dimovski, Stančić, Mihočka, Ružić, Vejnović" pokazuje jedan linearan rekurzivni algoritam za računanje faktorijela zadanog broja n.

Faktorijel broja n, označen kao $n!$, je produkt svih prirodnih brojeva manjih ili jednakih od n. Formalno, faktorijel se definira funkcijom:

$$n! = \begin{cases} 1 & , \quad n = 0 \\ n \cdot (n-1), & n \geq 1 \end{cases}$$

Ova definicija odmah upućuje na prirodnu rekurzivnu strukturu faktorijela: da bi izračunali $7!$, prvo izračunamo $6!$, zatim $5!$, i tako dalje, sve dok ne dođemo do $0!$ *(Slika 1).* Ovdje se radi o jednostavnoj linearnoj rekurziji.

$$\begin{aligned} 7! &= 7 \cdot 6! \\ &= 7 \cdot 6 \cdot 5! \\ &= 7 \cdot 6 \cdot 5 \cdot 4! \\ &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3! \\ &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! \\ &= 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\ &= 5040 \end{aligned}$$



Slika 1: Rekurzivno računanje n!

Ovako bi izgledala implementacija u Pythonu:

```
def faktorijel(n):
    if n == 0:
        return 1
    else:
        return n * faktorijel(n-1)
print(faktorijel(7))  # Ispisat će se 5040
```

Ovdje, funkcija *faktorijel* koja prima broj n za argument, poziva samu sebe sa smanjenim argumentom dok ne dođe do osnovnog slučaja kada je n jednako 0. U tom trenutku, funkcija vraća 1, što zaustavlja daljnju rekurziju.

4.2. Rekurzivno izračunavanje n-tog Fibonaccijevog broja

Ovaj primjer iz udžbenika "Informatika 8: Deljač, Gregurić, Hajdinjak, Počuča, Rakić, Svetličić" pokazuje jedan linearan rekurzivni algoritam za računanje n-tog Fibonaccijevog broja.

Fibonaccijevi brojevi čine niz u kojem je svaki broj zbroj prethodna dva, osim prva dva broja (nulti i prvi Fibonaccijev broj) koji su definirani kao 0 i 1. Niz počinje s $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ i tako dalje.

Matematički, n-ti Fibonaccijev broj $F(n)$ se može izračunati prema funkciji:

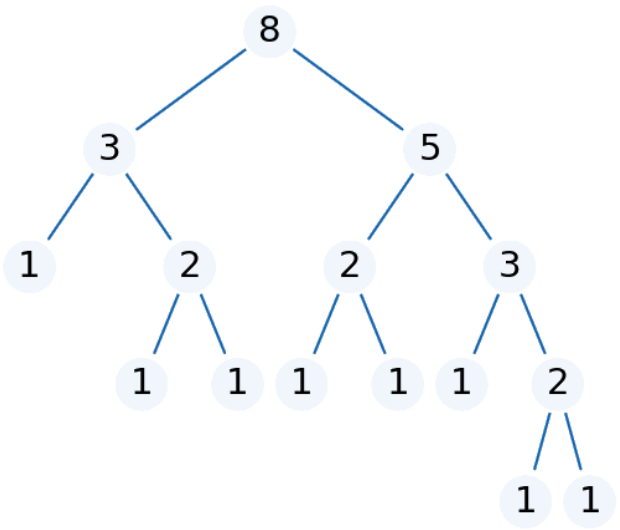
$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-2) + F(n-1), & n > 1 \end{cases}$$

I ovdje se vidi rekurzivna struktura. Na primjer, da bi izračunali šesti Fibbonacijev broj $F(6)$ potrebno je prije izračunati zbroj $F(4) + F(5)$. No da bi izračunali $F(4)$ potrebno je prije izračunati $F(2) + F(3)$ i da bi izračunali $F(5)$ potrebno je prije izračunati $F(3) + F(4)$.

Nastavljamo postupak sve dok ne dođemo do $F(0)$. Ako grafički prikažemo ove izračune, dobivamo binarno stablo *(Slika 2)*.

Postupak izračuna 6-tog Fibonaccijevog broja možemo prikazati i ovako:

$$
\begin{aligned}
F(6) &= F(4) + F(5) \\
&= F(2) + F(3) + F(3) + F(4) \\
&= F(0) + F(1) + F(1) + F(2) + F(1) + F(2) + F(2) + F(3) \\
&= 0 + 1 + 1 + F(0) + F(1) + 1 + F(0) + F(1) + F(0) + F(1) + F(1) + F(2) \\
&= 0 + 1 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 + 1 + F(0) + F(1) \\
&= 0 + 1 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 + 1 + 0 + 1 \\
&= 8
\end{aligned}
$$



Slika 2: Rekurzivno računanje šestog Fibonaccijevog broja

Ovako bi izgledala implementacija u Pythonu:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return(fibonacci(n-2) + fibonacci(n-1))
```

Funkcija *fibonacci* za argument prima prirodan broj n. Ukoliko je $n = 0$ funkcija vraća 0, ukoliko je $n = 1$ vraća 1, za sve ostale vrijednosti vraća zbroj rekurzivnih poziva *fibonacci(n-2)* i *fibonacci(n-1)*.

Rezultat pozivanja funkcije print(*fibonacci(6)*):

```
8
```

Rezultat pozivanja funkcije print(*fibonacci(20)*):

```
6765
```

Uočimo da ovaj algoritam nije efikasan. Naime, u prvom primjeru $F(3)$ se čak tri puta računa. Postupkom memoizacije, odnosno pamćenjem već izračunatih Fibonaccijevih brojeva, uvelike se ubrzava vrijeme izvođenja algoritma. U ovom radu nećemo obrađivati taj pristup.

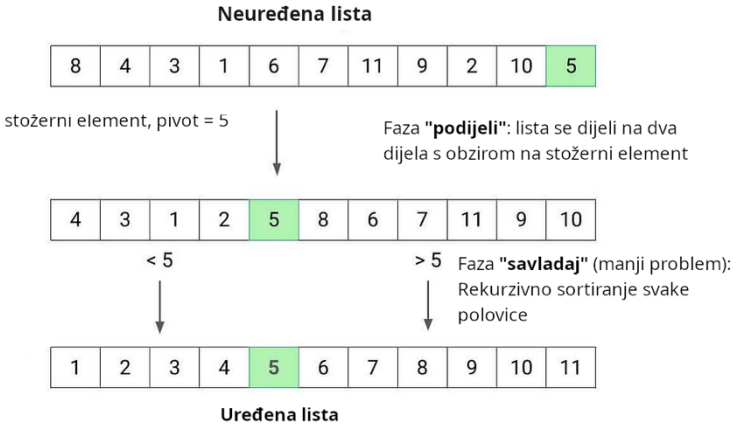
4.3. Quicksort

Ovaj primjer iz udžbenika “Napredno rješavanje problema programiranjem u Pythonu - udžbenik za 3. razred prirodoslovno-matematičke gimnazije: Budin, Brođanac, Perić“ i iz udžbenika “Svijet informatike 3 – udžbenik informatike u trećem razredu gimnazija: Šafar Đerki, Leventić, Ivanović-Ižaković, Stjepanek, Tomić“ pokazuje jedan višestruki rekurzivni algoritam sortiranje liste brojeva.

Jedan od najbržih dostupnih algoritama za sortiranje, quicksort koristi pristup "podijeli pa vladaj" kako bi se brzo sortirali podaci. Bit te strategije je da se problem podijeli na više jednostavnijih potproblema, čija se rješenja kombiniraju kako bismo dobili rješenje polaznog problema.

Osnovna zamisao je podijeliti listu elemenata npr. listu brojeva na dvije podliste elemenata: prva podlista čine svi elementi koji su manji od odabranog elementa niza (stožerni element ili pivot), a drugu grupu svi elementi koji su veći ili jednaki od odabranog elementa. Tako se odabrani element postavlja između podlista, na svoje mjesto, i tijekom daljnjeg sortiranja niza neće biti potrebe za njegovim premještanjem. Također, svi elementi prve podliste ostat će lijevo od odabranog

elementa, a svi elementi druge podliste ostat će desno od odabranog elementa. Svaka od podlista se zasebno sortira primjenom istog postupka rekurzivno ponovljenog. Primjer takvog sortiranja na neuređenoj listi brojeva možemo vidjeti na idućoj slici *(Slika 4)*, gdje za pivot izabiramo posljednji element liste.



Slika 3: Grafički prikaz quicksort algoritma

Prilikom dijeljenja niza, cilj je da grupe sadrže otprilike jednak broj elemenata, i upravo na taj način se postiže veća brzina ovog algoritma za sortiranje. Odabir pivota u značajnoj mjeri utječe na brzinu algoritma no mi ćemo ovdje za pivot uvijek birati zadnji element liste.

Ovako bi izgledala implementacija u Pythonu:

```
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivot = lista[-1]
        manji = []
        veci = []
        for i in range(len(lista) - 1):
            if lista[i] < pivot:
                manji.append(lista[i])
            else:
                veci.append(lista[i])
        return quicksort(manji) + [pivot] + quicksort(veci)
```

Rezultat pozivanja funkcije *quicksort([8, 4, 3, 1, 6, 7, 11, 9, 2, 10, 5])*:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Ovdje funkcija *quicksort* provjerava je li lista koju prima kao argument prazna ili sadrži samo jedan element. Ako je to slučaj, vraća se kao već sortirana.

- Ako lista ima više od jednog elementa, posljednji element liste biramo za pivot. Ovo je referentna točka za dijeljenje liste na manje elemente (manji) i veće elemente (veci).
- Koristeći petlju *for*, program prolazi kroz sve elemente liste osim zadnjeg (pivota). Ako je trenutni element manji od pivota, dodaje se u listu *manji*. Ako je veći ili jednak, dodaje se u listu *veci*.
- Funkcija *quicksort* zatim se rekurzivno poziva na listama *manji* i *veci*, koje se sortiraju zasebno.
- Naposljetku se spoji sortirana lista *manji*, lista koja sadrži samo pivot *[pivot*], i sortirana lista *veci* u jednu listu koja predstavlja konačni sortirani niz.
- Rekurzivni pozivi funkcije nastavljaju podijeliti liste sve manje i manje dok svaka lista ne sadrži najviše jedan element, na kojem se osnovni slučaj funkcije i dalje primjenjuje, rezultirajući u sortiranim listama koje se spajaju nazad uz stablo rekurzivnih poziva.

Ovaj kod je jednostavan i čitljiv, s jasnim logičkim koracima koji bi trebali biti razumljivi učenicima srednje škole koji su upoznati s osnovama Pythona i konceptom rekurzije.

4.4. Hanojski tornjevi

Ovaj primjer je iz udžbenika “Informatika 8 - udžbenik iz informatike za 8. razred osnovne škole: Kniewald, Galešav, Sokol, Kager, Vlahović, Purgar“ i iz udžbenika “Informatika 3 - udžbenik iz informatike za 3. razred prirodoslovno-matematičkih gimnazija: Dmitrović, Grabusin, Bujanović, Miletić, Kager“ pokazuje jedan višestruki rekurzivni algoritam za rješavanje problema Hanojskih tornjeva.

Hanojski tornjevi je zagonetka koja se sastoji od 3 tornja (štapa), na početnom se nalazi niz od n diskova, poredanih po veličini, na dnu je najveći, na vrhu najmanji *(Slika 4)*. Cilj je prebaciti sve diskove na ciljni toranj koristeći pomoćni toranj, tako da oni zadrže originalni poredak, koristeći 2 jednostavna pravila:

1. može se prebacivati samo jedan disk
2. ne smije se staviti veći disk na manji

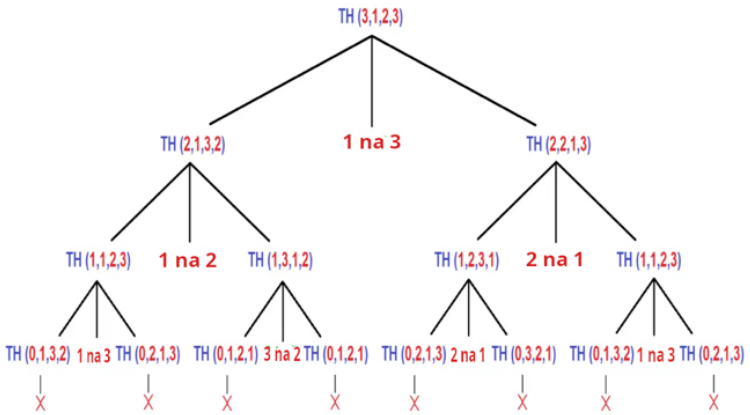
Slika 4: Hanojski tornjevi

Princip algoritma za rješavanje problema hanojskih tornjeva je jednostavan te u nastavku možemo vidjeti rekurzivnu prirodu rješenja. Najjednostavniji slučaj bio bi kada bi početni toranj sadržavao samo jedan disk gdje bi jednostavno prebacili taj jedan disk na ciljni toranj.

Označimo početni toranj sa 1, ciljni toranj sa 2 i pomoćni sa 3. Da bi pomakli n diskova na 2, moramo prvo pomaknuti $n-1$ diskova na 3. Tek onda možemo pomaknuti n-ti disk na 2 te pomaknuti preostala $n-1$ diska sa 3 na 2. I ovdje se vidi rekurzivna struktura:

- ako je $n = 1$ pomakni toranj sa početnog tornja na ciljni toranj (1 na 2)
- inače:
  pomakni $n-1$ toranj sa početnog na pomoćni (1 na 3)
  pomakni n-ti toranj sa početnog na ciljni (1 na 2)
  pomakni $n-1$ toranj sa pomoćnog na ciljni (3 na 2)

Ako grafički prikažemo ove korake u primjeru sa tri diska, dobivamo stablo *(Slika 5).*



Slika 5: Grafički prikaz rješavaja problema Hanojskih tornjeva

Ovako bi izgledala implementacija u Pythonu:

```
def hanojski_tornjevi(n, pocetni, ciljni, pomocni):
    if n == 1:
        print(f"Pomakni disk 1 sa tornja {pocetni} na toranj {ciljni}.")
    else:
        # Pomakni n-1 diskova sa pocetnog na pomocni toranj
        hanojski_tornjevi(n-1, pocetni, pomocni, ciljni)

        # Pomakni preostali n-ti disk na ciljni toranj
        print(f"Pomakni disk {n} sa tornja {pocetni} na toranj {ciljni}.")

        # Pomakni n-1 diskova sa pomocnog na ciljni toranj
        hanojski_tornjevi(n-1, pomocni, ciljni, pocetni)

# Broj diskova
n = 3

# Pozivanje funkcije
hanojski_tornjevi(n, '1', '2', '3')
```

Rezultat pozivanja funkcije *hanojski_tornjevi(3, '1', '2', '3')*:

```
Pomakni disk 1 sa tornja 1 na toranj 2.
Pomakni disk 2 sa tornja 1 na toranj 3.
Pomakni disk 1 sa tornja 2 na toranj 3.
Pomakni disk 3 sa tornja 1 na toranj 2.
Pomakni disk 1 sa tornja 3 na toranj 1.
Pomakni disk 2 sa tornja 3 na toranj 2.
Pomakni disk 1 sa tornja 1 na toranj 2.
```

Rezultat pozivanja funkcije *hanojski_tornjevi(4, '1', '2', '3')*:

Pomakni disk 1 sa tornja 1 na toranj 3.

Pomakni disk 2 sa tornja 1 na toranj 2.

Pomakni disk 1 sa tornja 3 na toranj 2.

Pomakni disk 3 sa tornja 1 na toranj 3.

Pomakni disk 1 sa tornja 2 na toranj 1.

Pomakni disk 2 sa tornja 2 na toranj 3.

Pomakni disk 1 sa tornja 1 na toranj 3.

Pomakni disk 4 sa tornja 1 na toranj 2.

Pomakni disk 1 sa tornja 3 na toranj 2.

Pomakni disk 2 sa tornja 3 na toranj 1.

Pomakni disk 1 sa tornja 2 na toranj 1.

Pomakni disk 3 sa tornja 3 na toranj 2.

Pomakni disk 1 sa tornja 1 na toranj 3.

Pomakni disk 2 sa tornja 1 na toranj 2.

Pomakni disk 1 sa tornja 3 na toranj 2.

U ovom Python programu definirana je funkcija *hanojski_tornjevi* koja rješava problem Hanojskih tornjeva rekurzivnim pristupom. Funkcija prima četiri argumenta: broj diskova n, oznaku početnog tornja, oznaku ciljnog tornja i oznaku pomoćnog tornja. Oznake tornjeva su zadane kao '1', '2' i '3'.

Ako je na početnom tornju samo jedan disk ($n = 1$), jednostavno se taj disk premješta na ciljni toranj. Ako ima više od jednog diska, problem se rješava rekurzivno prvo premještanjem $n - 1$ diska na pomoćni toranj, zatim premještanjem n-tog diska na ciljni toranj, i naposljetku premještanjem $n - 1$ diska s pomoćnog tornja na ciljni toranj.

Ovaj postupak ponavlja se sve dok se svi diskovi ne premjeste sa početnog tornja na ciljni, prateći pravila da niti jedan veći disk ne može biti iznad manjeg diska. Svaki put kad se obavi pomak diska, program ispisuje pomak, što olakšava praćenje koraka u izvršenju algoritma.

5. Traženje elementa u listi

5.1. Opis problema

Traženje elementa u listi može se napraviti pomoću jednostavne rekurzivne pretrage. Primjer takvog algoritma bio bi linearna rekurzivna pretraga, gdje funkcija poziva sebe sve dok ne pronađe element ili dosegne kraj liste.

Primjerice, na velikoj polici knjiga čiji se naslovi nalaze samo na prednjoj korici knjige, želimo pronaći knjigu s određenim naslovom. To možemo napraviti tako da prvu knjigu uzmemo sa police i usporedimo naslov knjige s traženim naslovom. Ako je to ta knjiga traženje je završilo, ako nije isti postupak traženja nastavljamo nad preostalim knjigama na polici.

Ovaj algoritam za rekurzivno traženje elementa u listi temelji se na principu dijeljenja problema na manje dijelove sve dok se ne dosegne najjednostavniji mogući slučaj, tj osnovni slučaj. Ideja je provjeriti odgovara li trenutni element traženom elementu, a ako ne odgovara, nastaviti pretragu manjim dijelom liste, isključujući već provjereni element.

U osnovnom slučaju provjeravamo je li prvi element u listi, traženi element. Ako je, pretraga se završava i vraća se uspješan rezultat. Drugi osnovni slučaj jest kada se lista svede na praznu listu, što znači da traženi element nije pronađen u listi, što također završava pretragu s neuspjelim rezultatom.

Ako osnovni slučaj nije zadovoljen, algoritam se nastavlja rekurzivno. Pretraga isključuje prvi element iz daljnjeg razmatranja i nastavlja pretraživati ostatak liste. To se postiže pozivom iste pretrage, ali na preostalom dijelu liste, koji ne uključuje trenutno provjereni element.

Procesom rekurzije, lista se postupno smanjuje dok se ne pronađe element ili dok lista u potpunosti ne postane prazna. Svaki rekurzivni poziv funkcije nastavlja isti postupak, smanjujući problem i približavajući se osnovnom slučaju. Na taj način, rekurzivna pretraga pomaže u strukturiranju problema na način koji omogućuje jednostavno i elegantno rješenje.

5.2. Kod u Pythonu

Evo kako bi implementacija linearnog rekurzivnog pretraživanja mogla izgledati u Pythonu:

```
def rekurzivno_pretrazivanje(lista, trazeni_element):
    # Ako je lista prazna, element se ne nalazi u listi
    if not lista:
        return False

    # Provjera je li trenutni element (prvi element liste) onaj koji tražimo
    if lista[0] == trazeni_element:
        return True
    else:
        # Rekurzivnim pozivom tražimo element u ostatku liste (bez prvog elementa)
        return rekurzivno_pretrazivanje(lista[1:], trazeni_element)

# Lista u kojoj tražimo element
lista = [1, 3, 5, 7, 9, 11]

# Element koji tražimo
trazeni_element = 7

# Pozivanje funkcije
pronasao = rekurzivno_pretrazivanje(lista, trazeni_element)
print(f'Element {"pronađen" if pronasao else "nije pronađen"} u listi.')
```

Objašnjenje koda:

Ovaj kod definira funkciju *rekurzivno_pretrazivanje* koja prima listu brojeva *lista* i traženi element *trazeni_element* koji treba pronaći.

- Prvi korak u funkciji je provjera je li *lista* prazna. Ako je *lista* prazna, funkcija vraća *False*, što označava da, *trazeni_element* nije pronađen (to je implicitni osnovni slučaj).

- Zatim se provjerava je li prvi element liste jednak *trazeni_element*. Ako jest, funkcija vraća *True*, označavajući da je *trazeni_element* pronađen.
- Ako prvi element nije onaj koji tražimo, funkcija se rekurzivno poziva na preostalom dijelu liste - to se postiže uklanjanjem prvog elementa liste kroz *lista[1:].*
- Na kraju, ako funkcija dođe do prazne liste, to znači da je prošla kroz sve elemente i nije pronašla *trazeni_element*, ona vraća *False*.
- Van funkcije, pozivamo *rekurzivno*_pretrazivanje s konkretnom listom *lista* i elementom *trazeni_element* koji tražimo, te ispisujemo poruku da li je element pronađen ili nije.

5.3. Pokrivenost kurikuluma

Ovaj primjer je primjeren učenicima 2. razreda srednje škole pokrivajući ishod B.2.2. Nastavnik će na ovom jednostavnim primjeru uvesti pojam rekurzivne funkcije te će učenici moći uvidjeti mogućnosti korištenja rekurzivnih funkcija i uočiti rekurzivnost u definiciji problema.

Učenici će moći raščlaniti zadani problem na manje funkcionalne cjeline, odrediti ulazne i izlazne parametre, razlikovati globalne, lokalne i formalne parametre, ispravno koristiti programske funkcije i integrirati u cjelovito rješenje problema te koristiti jednostavnu rekurzivnu funkciju.

5.4. Vremenska složenost

Vremenska složenost ove rekurzivne pretrage je $O(n)$, gdje je n broj elemenata u listi. Ovo je zato jer u najgorem slučaju, ako je traženi element na kraju liste ili ga uopće nema, potrebno proći kroz svaki element točno jednom. To znači da se vrijeme potrebno za izvršenje algoritma povećava linearno s povećanjem broja elemenata u listi.

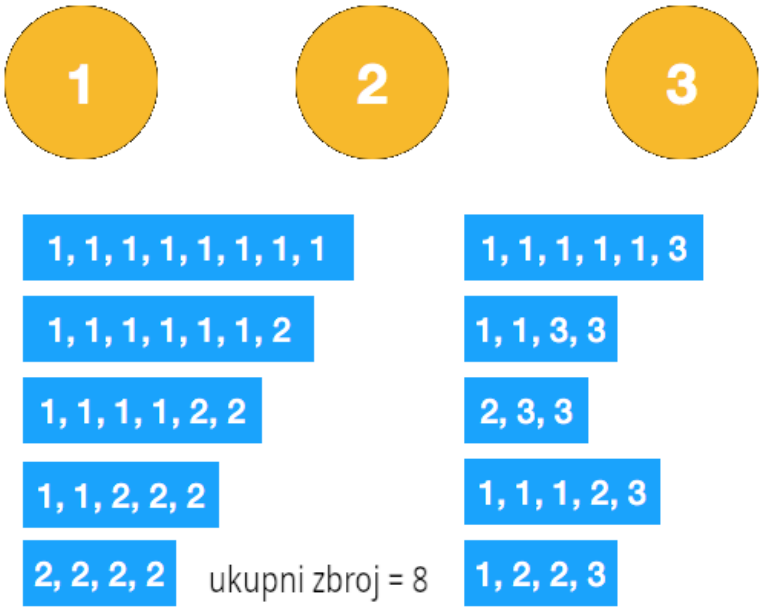
Ovo je primjer linearnog rekurzivnog pretraživanja koje nije najučinkovitije jer u najgorem slučaju prolazi kroz cijelu listu. Za velike skupove podataka gdje performanse postaju važne, koristile bi se naprednije tehnike pretrage, poput binarne pretrage, ali binarna pretraga zahtijeva sortiranu listu.

6. Problem raspodjele kovanica

6.1. Opis problema

Problem raspodjele kovanica može se referirati na različite probleme, ali jedan od klasičnih je problem razmjene novca, gdje je zadatak pronaći sve načine kako rasporediti određeni iznos novca koristeći dane denominacije kovanica. Ovo je standardni problem kombinatorike koji se može riješiti koristeći rekurziju za generiranje svih mogućih kombinacija.

Na idućoj slici *(Slika 6)* možemo vidjeti raspodjelu kovanicama sa denominacijama 1, 2 i 3 eura tako da u zbroju čine 8 eura. Za dani primjer postoji ukupno 10 različitih raspodjela kovanica.



Slika 6: Primjer raspodjele kovanica

Algoritam započinje s ciljem pronalaska svih mogućih načina na koje se može kombinirati određeni set kovanica kako bi se dobila suma koja odgovara predviđenom iznosu. Proces se odvija kroz seriju rekurzivnih poziva gdje svaki poziv predstavlja jednu razinu dublje u potrazi za pravilnom kombinacijom.

Pri svakom rekurzivnom pozivu, algoritam provjerava postoji li kombinacija kovanica koja točno odgovara traženom iznosu. Ako trenutna kombinacija kovanica točno odgovara iznosu, algoritam bilježi to kao uspješno rješenje. Ako kombinacija premašuje iznos, znači da ta kombinacija nije ispravna i algoritam je odbacuje, to je osnovni slučaj rekurzije.

Svaki put algoritam uzima jednu kovanicu te pokušava izgraditi valjanu kombinaciju smanjujući ukupni iznos za vrijednost te kovanice. To znači da

algoritam "razmišlja" unaprijed: "Ako uzmem ovu kovanicu, koliko mi onda ostaje iznosa da popunim s preostalim kovanicama?" Rekurzija omogućuje algoritmu da istraži različite putanje prema rješenju, krećući se naprijed kad dodaje nove kovanice i povratak unazad kad naiđe na slijepu ulicu, odnosno kada shvati da trenutni izbor kovanica ne vodi prema rješenju.

Algoritam nastavlja s procesom sve dok se sve kovanice ne isprobaju u svim mogućim kombinacijama. Kada se završe svi mogući pokušaji kombiniranja, algoritam završava i pruža ukupan broj kombinacija koje odgovaraju početnom iznosu. Složenost ovog pristupa raste s povećanjem broja različitih kovanica i iznosa koji se traži, zbog kombinatorne prirode problema.

6.2. Kod u Pythonu

Evo jednostavnog primjera takvog rekurzivnog programa u Pythonu:

```
def pronadi_kombinacije(iznos, kovanice, trenutna_kombinacija=[]):
    if iznos == 0:
        print(trenutna_kombinacija)
        return 1  # Pronašli smo jednu valjanu kombinaciju
    if iznos < 0:
        return 0  # Trenutna kombinacija prelazi zadani iznos


    broj_kombinacija = 0
    for i in range(len(kovanice)):
        # Dodajemo kovanice jednu po jednu i ponavljamo postupak
        broj_kombinacija += pronadi_kombinacije(iznos - kovanice[i], kovanice[i:], trenutna_kombinacija + [kovanice[i]])
    return broj_kombinacija
```

Rezultat pozivanja funkcije *pronadi_kombinacije(8, [1, 2, 3])*:

```
[1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 2]
[1, 1, 1, 1, 1, 3]
[1, 1, 1, 1, 2, 2]
```

[1, 1, 1, 2, 3]

[1, 1, 2, 2, 2]

[1, 1, 3, 3]

[1, 2, 2, 3]

[2, 2, 2, 2]

[2, 3, 3]

Ukupno mogućih kombinacija: 10

Rezultat pozivanja funkcije *pronadi_kombinacije(10, [3, 5, 7, 8])*:

[3, 7]

[5, 5]

Ukupno mogućih kombinacija: 2

Svi prethodni primjeri su se mogli riješiti i brojanjem “na prste“. U nastavku slijedi primjer sa velikim brojem kovanica gdje nećemo ispisati sve kombinacije već samo ukupan broj kombinacija.

Rezultat pozivanja funkcije *pronadi_kombinacije(100, [3, 5, 7, 8, 9, 10, 12, 13, 15])*:

Ukupno mogućih kombinacija: 19745

Objašnjenje koda:

Ovaj kod definira funkciju *pronadi_kombinacije* koja pronalazi sve moguće načine za raspodjelu zadanog iznosa *iznos* pomoću dostupnih kovanica *kovanice*. Evo koraka kako ova funkcija radi:

- Osnovni slučajevi: Ako je *iznos* jednak 0, to znači da smo pronašli valjanu kombinaciju kovanica *trenutna_kombinacija* koja odgovara zadanom iznosu, pa ispisujemo tu kombinaciju i vraćamo 1. Ako je iznos manji od 0, trenutna kombinacija prelazi zadani iznos i nije valjana, pa vraćamo 0 jer nije pronađena valjana kombinacija.
- Rekurzija: Ako nismo dosegli osnovni slučaj, iniciramo varijablu *broj_kombinacija* s 0 koja će brojiti sve valjane kombinacije. Prolazimo kroz sve dostupne kovanice i za svaku od njih, rekurzivno pozivamo *pronadi_kombinacije* smanjenim iznosom za trenutnu kovanicu i

ažuriranom listom kovanica koja ne uključuje kovanice veće vrijednosti od one koju trenutno razmatramo (to osigurava da nećemo generirati kombinacije koje sadrže iste kovanice u različitom redoslijedu).
- Za svaku iteraciju, sve kombinacije koji dolaze iz rekurzivnih poziva se dodaju na ukupan broj kombinacija *broj_kombinacija.*
- Na kraju, funkcija vraća ukupan broj valjanih kombinacija *broj_kombinacija* kovanica koje odgovaraju zadanom iznosu.

6.3. Pokrivenost kurikuluma

Ovaj primjer je primjeren učenicima 3. razreda srednje škole pokrivajući ishod B.3.3. Nastavnik će na ovom jednostavnim primjeru uvesti pojam rekurzivne funkcije te će učenici moći uvidjeti mogućnosti korištenja rekurzivnih funkcija i uočiti rekurzivnost u definiciji problema.

Učenici će moći uočiti rekurzivnost u definiciji problema, moći opisati osnovne elemente rekurzivnoga postupka, zapisati matematički opisanu rekurzivnu funkciju u programskome jeziku, uočiti rekurzivnost u danome problemu, odrediti rekurzivnu relaciju i uvjet prekida te realizirati rekurzivnu funkciju u programskome jeziku, procijeniti efikasnost rekurzivnoga rješenja i ovisno o problemu odabrati rekurzivno odnosno induktivno rješenje.

6.4. Vremenska složenost

Određivanje vremenske složenosti ove rekurzivne funkcije nije posve jednostavno jer ovisi o broju denominacija kovanica i o veličini iznosa. U najgorem slučaju, ako imamo samo najmanju kovanicu kao opciju, rekurzivna funkcija će biti pozvana za svaki pojedinačni "cent" sve do nule, što znači da će biti $O(iznos)$ poziva funkcija. Međutim, kako uključujemo više denominacija, broj poziva funkcije eksponencijalno raste zbog kombinatoričke prirode problema. Zbog toga bi složenost ovog algoritma mogla biti blizu eksponencijalnoj, posebice $O(m^n)$, gdje je m iznos, a n broj kovanica.

7. Sortiranje spajanjem

7.1. Opis problema

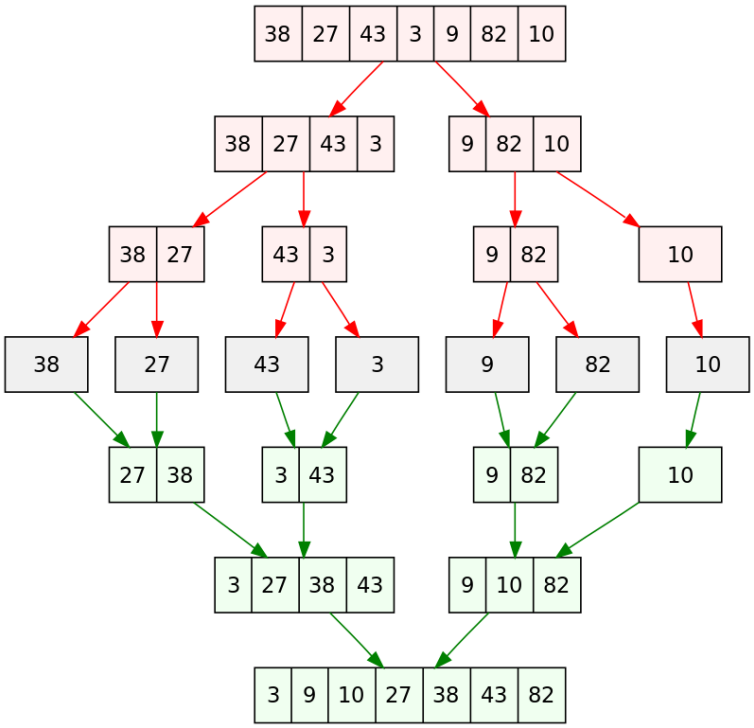
Sortiranje spajanjem, poznato i kao mergesort, je algoritam za sortiranje koji koristi pristup “podijeli pa vladaj“. Njegova osnovna ideja je da se veliki problem (u ovom slučaju, nesortirani niz ili lista) podijeli na manje, lakše rješive probleme, koji se zatim rješavaju neovisno jedan od drugoga , nakon čega se rješenja manjih problema (sortirane kraće liste) kombiniraju za konačno rješenje.

Kako funkcionira mergesort:

1. Podjela: ulazni niz dijeli se na dva niza približno jednake veličine.
2. Rekurzivno sortiranje: svaki od tih nizova zatim se rekurzivno sortira koristeći isti algoritam sortiranja spajanjem.
3. Spajanje: konačno, dva sortirana niza spajaju se u jedan sortirani niz.

Postupak spajanja:

Prilikom spajanja dva niza, uspoređuje se njihovi elementi jedan po jedan, pri čemu se najmanji elementi (prema sortiranju) prvo premještaju u izlazni niz. Ovaj postupak nastavlja se sve dok se svi elementi oba niza ne premjeste u izlazni, sada sortirani niz *(Slika 7)*.



Slika 7: Grafički prikaz mergesort algoritma

7.2. Kod u Pythonu

Evo jednostavnog primjera takvog rekurzivnog programa u Pythonu:

```
def merge_sort(lista):
    if len(lista) > 1:
        sredina = len(lista) // 2
        lijeva_polovica = lista[:sredina]
        desna_polovica = lista[sredina:]
        merge_sort(lijeva_polovica)
        merge_sort(desna_polovica)

        i = j = k = 0
        while i < len(lijeva_polovica) and j < len(desna_polovica):
            if lijeva_polovica[i] < desna_polovica[j]:
                lista[k] = lijeva_polovica[i]
                i += 1
            else:
                lista[k] = desna_polovica[j]
                j += 1
            k += 1

        while i < len(lijeva_polovica):
            lista[k] = lijeva_polovica[i]
            i += 1
            k += 1
        while j < len(desna_polovica):
            lista[k] = desna_polovica[j]
            j += 1
            k += 1
    return lista
```

Rezultat pozivanja funkcije *print(merge_sort(nesortirana_lista*(*[38, 27, 43, 3, 9, 82, 10]))*):

```
[3, 9, 10, 27, 38, 43, 82]
```

Rezultat pozivanja funkcije *print(merge_sort(nesortirana_lista([54, 26, 93, 17, 77, 31, 44, 55, 20]))*):

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Objašnjenje koda:

Funkcija *merge_sort je* rekurzivna funkcija za uzlazno sortiranje ulazne liste. Evo koraka po kojima funkcija radi:

- Osnovni slučaj rekurzije: Ako je lista duljine 1, ona je već sortirana (to je osnovni slučaj), i funkcija se ne poziva dalje.
- Podjela liste: lista se dijeli na dvije polovice - *lijeva_polovica i* desna*_polovica*, upotrebom sredine liste.
- Rekurzivni pozivi: merge*_sort* se rekurzivno poziva na oba dijela liste (lijeva*_polovica* i desna*_polovica*), izvodeći daljnju podjelu sve dok se ne dosegne osnovni slučaj.
- Spajanje polovica: nakon što se polovice rekurzivno sortiraju, one se moraju spojiti natrag u jednu sortiranu listu. To se čini prelaskom kroz elemente obiju polovica i dodavanjem manjeg elementa u rezultantnu listu.
- Kopiranje preostalih elemenata: Eventualno preostali elementi u jednoj od polovica koji nisu do sada dodani u rezultantnu listu moraju biti kopirani.
- Vraćanje rezultata: Na kraju, kada se sve polovice spoje, funkcija vraća sortiranu listu *lista*.

7.3. Pokrivenost kurikuluma

Ovaj primjer je također primjeren učenicima 3. razreda srednje škole pokrivajući ishod B.3.3.

7.4. Vremenska složenost

Algoritam Merge Sort ima vremensku složenost $O(n\log(n))$, što ga čini vrlo efikasnim za sortiranje velikih nizova ili listi. Međutim, Merge sort zahtijeva dodatni prostor za pohranu privremenih polovica niza tijekom sortiranja.

Ipak, ovaj algoritam je zbog svoje vremenske složenosti efikasan zbog načina na koji razdvaja problem na manje dijelove, što ga čini dobrim primjerom pristupa "podijeli pa vladaj".

Zbog svoje jasne logike podjele na manje, lakše rješive probleme,, Merge sort se često koristi kao uvodni primjer rekurzivnih algoritama u edukativne svrhe te ima važno mjesto u osnovama informatičkog obrazovanja i algoritamskog treninga.

8. Traženje najkraćeg puta u grafu

8.1. Definicija težinskog grafa

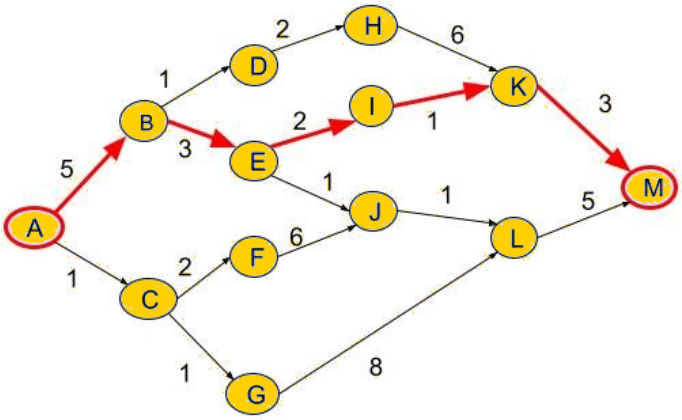
Težinski graf je graf u kojem je svakom bridu (spoju između dva čvora) pridružena brojčana vrijednost, odnosno težina. Te težine mogu predstavljati razne veličine kao što su udaljenost, vrijeme, trošak, kapacitet ili bilo koju drugu veličinu koja se može iskazati brojem. U kontekstu teorije grafova i računalnih znanosti, ove težine igraju ključnu ulogu u određivanju najkraćih puteva, minimalnih razapinjućih stabala, optimizaciji mreža i u primjenama gdje je potrebno pronaći rješenje prema nekom kriteriju optimalnosti.

Primjer težinskog grafa u svakodnevnom životu mogao bi biti karta grada, gdje čvorovi predstavljaju križanja ili specifičnu lokaciju, a težine bridova mogu predstavljati udaljenost između tih križanja ili predviđeno vrijeme putovanja od jedne do druge točke.

U formalnom matematičkom smislu, težinski graf G može se definirati kao trojka (V, E, w):

- V je skup čvorova ili vrhova grafa.
- E je skup bridova, gdje svaki brid povezuje dva čvora.
- w je funkcija težine koja svakom bridu E dodjeljuje težinsku vrijednost, često označenu kao $w: E \rightarrow \mathbb{R}$, gdje je $\mathbb{R}$ skup realnih brojeva.

Težinski graf može biti usmjeren ili neusmjeren. U usmjerenom grafu, svaki brid ima smjer i težina ovog brida može biti različita ovisno o smjeru. Na primjer, put od točke A do točke B može imati drugačiju težinu od puta od točke B do točke A, ili čak može biti da ne postoji takav put, što možemo vidjeti u aplikacijama poput navigacije gdje je put uzbrdo drugačiji od puta nizbrdo ili gdje je put jednosmjeran *(Slika 8)*. U neusmjerenom grafu, bridovi nemaju smjer i težina je jednaka neovisno o smjeru u kojem se putuje. Mi ćemo dalje razmatrati samo neusmjerene grafove.



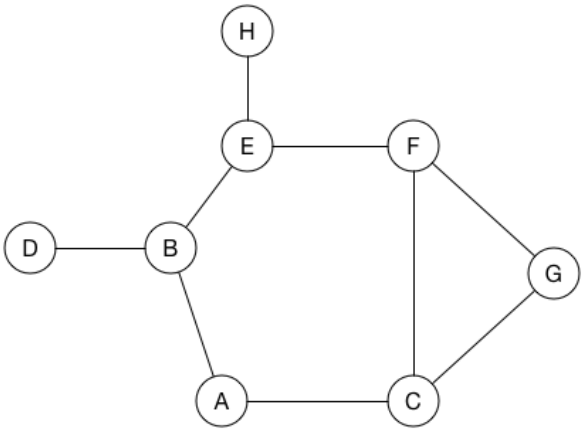
Slika 8: Primjer usmjerenog težinskog grafa

8.2. Opis problema

Traženje najkraćeg puta u grafu je složen problem i postoji puno različitih metoda za njegovo rješavanje, ovisno o prirodi grafa i dodatnim ograničenjima koja mogu postojati. Za jednostavne grafove *(Slika 9)* u kojima tražimo najkraći put od jednog čvora do drugog, možemo koristiti rekurzivnu pretragu u dubinu (depth-first search, DFS).

Pretraživanje u dubinu metoda je obilaska čvorova u kojoj prelazimo s jednog čvora na drugi dok god nalazimo nove, neposjećene čvorove. U slučaju da iz trenutnog čvora ne možemo obići neki novi čvor, vraćamo se na čvor iz kojeg smo došli, pokušavamo ići na neposjećene čvorove („vraćanje unatrag“). Pretraživanje završava kad se posjete svi čvorovi do kojih postoji put iz prvog čvora.

Ako promotrimo *sliku 9*, i vidimo da je najkraći put između čvora A i čvora F upravo put $A \rightarrow C \rightarrow F$.



Slika 9: Primjer neusmjerenog grafa bez težina

8.3. Kod u Pythonu

U segmentu koji slijedi nalazi se Python prikaz algoritma koji koristi DFS za pronalazak puta s minimalnim brojem bridova u neusmjerenom grafu, ne uzimajući u obzir težine.

```
def dfs_najkraci_put(graf, pocetak, kraj, put=None):
    if put is None:
        put = []
    put = put + [pocetak]
    if pocetak == kraj:
        return put
    najkraci_put = None
    for susjed in graf[pocetak]:
        if susjed not in put:
            novi_put = dfs_najkraci_put(graf, susjed, kraj, put)
            if novi_put:
                if najkraci_put is None or len(novi_put) < len(najkraci_put):
                    najkraci_put = novi_put
    return najkraci_put
# Grafički prikazivanje grafa preko rječnika
graf = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B', 'F', 'H'],
    'F': ['C', 'E', 'G'],
    'G': ['C', 'F'],
    'H': ['E']
}
```

```
# Pronađi najkraći put između čvorova 'A' i 'F' koristeći DFS pretragu
najkraci_put = dfs_najkraci_put(graf, 'A', 'F')
print(najkraci_put)
```

Rezultat izvođenja gornjeg koda:

```
['A', 'C', 'F']
```

Objašnjenje koda:

Ulazni argumenti funkcije *dfs_najkraci_put* uključuju:

- *graf*: Ovo je rječnik koji predstavlja graf u kojem svaki ključ predstavlja čvor, a vrijednost ključa je lista susjednih čvorova.
- *pocetak*: Ovo je ključ koji označava početni čvor od kojeg želimo pronaći najkraći put.
- *kraj*: Ovo je ključ koji označava ciljni čvor do kojeg želimo pronaći najkraći put.
- *put*: Ovo je argument koji nam omogućuje praćenje trenutnog puta tijekom rekurzivnih poziva. U početku, može biti postavljen na None.

Ovi argumenti omogućuju funkciji da izvršava rekurzivnu pretragu u grafu kako bi pronašla najkraći put od početnog do ciljnog čvora.

Algoritam započinje traženje najkraćeg puta od početnog čvora prema ciljnom čvoru.

- Osnovni slučaj rekurzije događa se kada trenutni čvor postane jednak ciljnom čvoru. Tada se vraća put koji sadrži samo taj čvor.
- U rekurzivnom koraku, algoritam ispituje sve susjedne čvorove trenutnog čvora. Ako susjedni čvor nije već uključen u trenutni put, rekurzivno se poziva *dfs_najkraci_put* s tim susjednim čvorom kao početnim čvorom. Ovaj proces se ponavlja sve dok ne pronađemo ciljni čvor, nakon čega se vraća put koji predstavlja najkraći put od početnog do ciljnog čvora.

Na taj način, DFS koristi rekurziju za istraživanje svih mogućih puteva u grafu kako bi pronašao najkraći put od početnog do ciljnog čvora.

8.4. Pokrivenost kurikuluma

Ovaj primjer je primjeren učenicima 3. razreda srednje škole pokrivajući ishod B.3.3. Nastavnik će na ovom primjeru odrediti rekurzivnu relaciju i vizualizirati rekurziju s jednostavnim grafičkim elementima.

Učenici će moći opisati osnovne elemente rekurzivnoga postupka, zapisati matematički opisanu rekurzivnu funkciju u programskome jeziku, uočiti rekurzivnost u danome problemu, odrediti rekurzivnu relaciju i uvjet prekida te realizirati rekurzivnu funkciju u programskome jeziku, procijeniti efikasnost rekurzivnoga rješenja i ovisno o problemu odabrati rekurzivno odnosno induktivno rješenje.

8.5. Vremenska složenost

Vremenska složenost algoritma Depth-First Search (DFS) u neusmjerenim grafovima može se analizirati kroz prizmu dva bitna faktora: broj čvorova V i broj bridova E. DFS algoritam slijedi granu grafa do kraja prije nego što se vrati unatrag i pokuša alternativne puteve, što znači da će eventualno posjetiti svaki čvor i pregledati svaki brid.

Za gornji rekurzivni DFS algoritam, možemo konkretno navesti sljedeće:

- Posjeta svim čvorovima: algoritam će posjetiti svaki čvor u grafu barem jednom kako bi utvrdio postoji li put do ciljnog čvora ili ne. Ako uzmemo u obzir da svaki čvor može imati više susjeda, ali svaki susjed će biti pregledan samo jedanput, to daje složenost $O(V)$.
- Iteracija kroz sve bridove: za svaki čvor, DFS može potencijalno pregledati sve bridove koji se protežu od tog čvora. U neusmjerenom grafu svaki neusmjereni brid povezuje dva čvora i može biti pregledan iz oba čvora, zbog čega je ukupni broj pregleda bridova $2E$. Međutim, u analizi složenosti, konstantni faktor se zanemaruje, pa se složenost bridova opet izražava kao $O(E)$.

Ukupna vremenska složenost DFS algoritma za neusmjereni graf dakle iznosi $O(V + E)$. U praksi, to znači da će u najgorim slučajevima gdje je potrebno posjetiti svaki čvor i pregledati svaki brid za pronalazak puta, broj operacija biti proporcionalan zbroju broja čvorova i bridova u grafu.

S obzirom da ovaj algoritam pretražuje prvi put koji zatekne do ciljnog čvora, što nije nužno put s najmanjom ukupnom težinom jer ne uzima u obzir težine bridova. Za pravi algoritam traženja najkraćeg puta s obzirom na težine (kao što je Dijkstrin algoritam ili A* algoritam), potrebno je koristiti drugačije algoritme optimizirane za rad s težinama.

9. Problem trgovačkog putnika

U literaturi postoje brojne formulacije problema trgovačkog putnika. Njegov osnovni oblik se najčešće formulira na dva načina:

- Trgovački putnik ima zadan skup gradova od kojih svaki mora posjetiti točno jedanput i vratiti se u početni grad. Udaljenosti među gradovima su poznate. Pitanje je kojim redoslijedom bi trebao obilaziti gradove da ukupna duljina puta bude minimalna.
- U težinskom grafu pronaći Hamiltonov ciklus minimalne težine.

9.1. Hamiltonov ciklus i problem trgovačkog putnika

Hamiltonov ciklus i problem trgovačkog putnika (TSP - Travelling Salesman Problem) blisko su povezani u teoriji grafova te predstavljaju dva fundamentalna koncepta u računalstvu i kombinatorici.

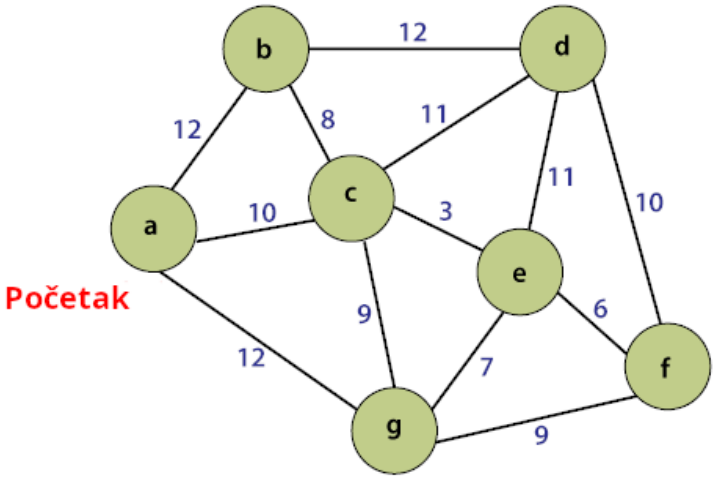
Hamiltonov ciklus je vrsta puta u grafu koji počinje i završava u istom čvoru te prolazi kroz svaki preostali čvor točno jednom. Drugim riječima, Hamiltonov ciklus mora proći kroz sve čvorove u grafu bez ponavljanja i vratiti se u početni čvor.

Pojam je dobio ime po irskom matematičaru Sir Williamu Rowanu Hamiltonu (1805.-1865.), koji je proučavao ovaj problem.

Problem trgovačkog putnika također uključuje pronalaženje puta koji posjeti svaki čvor u grafu točno jednom i vraća se u početni čvor, no sa dodatnim zahtjevom da ukupna težina (koja može predstavljati udaljenost, trošak, vrijeme ili neku drugu mjerljivu veličinu) puta bude što je moguće manja. Drugim riječima, TSP traži optimalni Hamiltonov ciklus, odnosno Hamiltonov ciklus s najmanjom mogućom težinom.

Poveznica između ova dva koncepta je u tome što svaki problem trgovačkog putnika uključuje potragu za Hamiltonovim ciklusom, ali s dodatnim ograničenjem optimizacije težine. Dok je identificiranje bilo kojeg Hamiltonovog ciklusa u grafu sam po sebi izazovno, TSP dodaje još jednu razinu složenosti tražeći najefikasniji takav ciklus.

Zbog svoje složenosti i praktične primjene u planiranju ruta, logistici i mnogim drugim područjima, TSP je jedan od najistraživanijih problema u optimizaciji i računalnoj znanosti.



Slika 10: Primjer neusmjerenog težinskog grafa

Na slici (*Slika 10)* možemo vidjeti primjer grafa s 7 čvorova, označeni sa a, b, c, d, e, f i g. Neki čvorovi su povezani bridom a neki nisu i svakom bridu je dodijeljena težina. Želimo pronaći Hamiltonov ciklus s najmanjom mogućom težinom. Bilo bi korisno zapisati sve te težine u tablicu *(Slika 11).*

	a	b	c	d	e	f	g
a	0	12	10	0	0	0	12
b	12	0	8	12	0	0	0
c	10	8	0	11	3	0	9
d	0	12	11	0	11	10	0
e	0	0	3	11	0	6	7
f	0	0	0	10	6	0	9
g	12	0	9	0	7	9	0

Slika 11: Tablica težina bridova

Ako svaki čvor slijedno označimo sa $0, 1, 2, 3, 4, 5$ i 6, ova tablica se može prevesti u matricu težina u kojoj broj u i-tom retku i j-tom stupcu predstavlja težinu brida koji povezuje i-ti i j-ti čvor.

Naime, svi Hamiltonovi ciklusi i njihove ukupne duljine u ovom grafu su:

$0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 0$, Ukupna duljina ciklusa: 69
$0 \to 1 \to 2 \to 3 \to 5 \to 4 \to 6 \to 0$, Ukupna duljina ciklusa: 66
$0 \to 1 \to 2 \to 4 \to 3 \to 5 \to 6 \to 0$, Ukupna duljina ciklusa: 65
$0 \to 1 \to 3 \to 2 \to 4 \to 5 \to 6 \to 0$, Ukupna duljina ciklusa: 65
$0 \to 1 \to 3 \to 4 \to 5 \to 6 \to 2 \to 0$, Ukupna duljina ciklusa: 69
$0 \to 1 \to 3 \to 5 \to 4 \to 2 \to 6 \to 0$, Ukupna duljina ciklusa: 64
$0 \to 1 \to 3 \to 5 \to 4 \to 6 \to 2 \to 0$, Ukupna duljina ciklusa: 66
$0 \to 1 \to 3 \to 5 \to 6 \to 4 \to 2 \to 0$, Ukupna duljina ciklusa: 63
$0 \to 2 \to 1 \to 3 \to 4 \to 5 \to 6 \to 0$, Ukupna duljina ciklusa: 68
$0 \to 2 \to 1 \to 3 \to 5 \to 4 \to 6 \to 0$, Ukupna duljina ciklusa: 65
$0 \to 2 \to 4 \to 6 \to 5 \to 3 \to 1 \to 0$, Ukupna duljina ciklusa: 63
$0 \to 2 \to 6 \to 4 \to 5 \to 3 \to 1 \to 0$, Ukupna duljina ciklusa: 66
$0 \to 2 \to 6 \to 5 \to 4 \to 3 \to 1 \to 0$, Ukupna duljina ciklusa: 69
$0 \to 6 \to 2 \to 4 \to 5 \to 3 \to 1 \to 0$, Ukupna duljina ciklusa: 64
$0 \to 6 \to 4 \to 5 \to 3 \to 1 \to 2 \to 0$, Ukupna duljina ciklusa: 65
$0 \to 6 \to 4 \to 5 \to 3 \to 2 \to 1 \to 0$, Ukupna duljina ciklusa: 66
$0 \to 6 \to 5 \to 3 \to 4 \to 2 \to 1 \to 0$, Ukupna duljina ciklusa: 65
$0 \to 6 \to 5 \to 4 \to 2 \to 3 \to 1 \to 0$, Ukupna duljina ciklusa: 65
$0 \to 6 \to 5 \to 4 \to 3 \to 1 \to 2 \to 0$, Ukupna duljina ciklusa: 68
$0 \to 6 \to 5 \to 4 \to 3 \to 2 \to 1 \to 0$, Ukupna duljina ciklusa: 69

Mi želimo odrediti onaj Hamiltonov ciklus s najmanjom ukupnom duljinom.

9.2. Kod u Pythonu

```
def tsp(graf, trenutni, posjeceni, n, brojac, duljina, min_duljina, put):
    if brojac == n and graf[trenutni][0]:
        if duljina + graf[trenutni][0] < min_duljina[0]:
            min_duljina[0] = duljina + graf[trenutni][0]
            put[0] = list(posjeceni)
        return
    for cvor in range(n):
        if (graf[trenutni][cvor] and not posjeceni[cvor]):
            posjeceni[cvor] = True
```

```
            tsp(graf, cvor, posjeceni, n, brojac + 1, duljina + graf[trenutni][cvor],
min_duljina, put)
            posjeceni[cvor] = False


# Težinski graf
graf = [
    [0, 12, 10, 0, 0, 0, 12],
    [12, 0, 8, 12, 0, 0, 0],
    [10, 8, 0, 11, 3, 0, 9],
    [0, 12, 11, 0, 11, 10, 0],
    [0, 0, 3, 11, 0, 6, 7],
    [0, 0, 0, 10, 6, 0, 9],
    [12, 0, 9, 0, 7, 9, 10],
]

n = len(graf)
posjeceni = [False] * n
posjeceni[0] = True
min_duljina = [float('inf')]
put = []
tsp(graf, 0, posjeceni, n, 1, 0, min_duljina, put)

print("Najkraći trgovački put je:", min_duljina])
print("Hamiltonov ciklus s duljinom trajanja", min_duljina[0], "je:", put)
```

Rezultat izvođenja gornjeg koda:

```
Najkraći trgovački put je: 63
Hamiltonov ciklus s duljinom trajanja 63 je: [0, 1, 3, 5, 6, 4, 2, 0]
```

Objašnjenje koda:

Funkcija *tsp* koristi rekurziju kako bi pronašla najkraći trgovački put u danom grafu. Evo objašnjenja koda koristeći izraze osnovni slučaj i rekurzija:

Osnovni slučaj:

- U osnovnom slučaju, funkcija *tsp* provjerava je li trgovački put završen (brojač *n*) i da li postoji brid koji spaja trenutni čvor s početnim čvorom (*graf[trenutni][0]*).
- Ako su ova dva uvjeta zadovoljena, tada se provjerava duljina puta. Ako je trenutna duljina puta manja od *min_duljina*, tada se ažurira *min_duljina* i *put* s trenutnim posjećenim čvorovima.

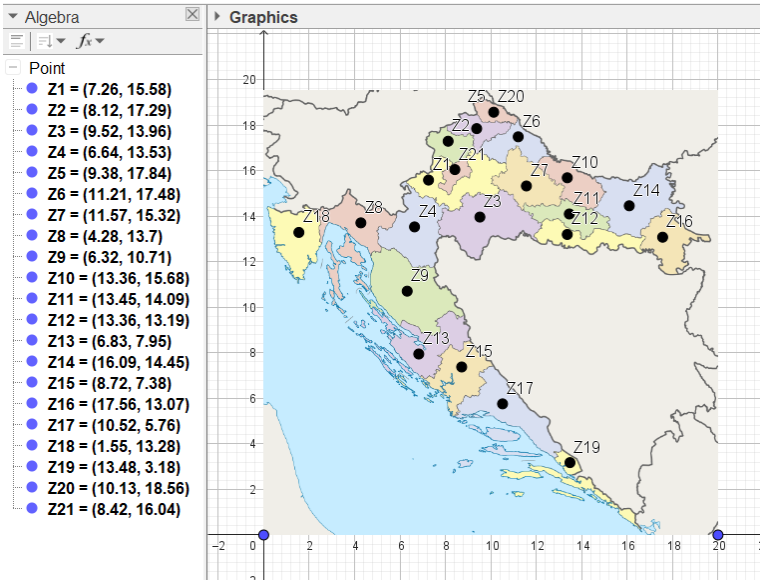
Rekurzija:

- U rekurzivnom koraku, funkcija *tsp* prolazi kroz sve susjedne gradove trenutnog čvora.
- Za svaki čvor koji može biti posjećen (*cvor[trenutni][grad] > 0*) i koji nije već posjećen (*not posjeceni[cvor]*), funkcija rekuzivno poziva samu sebe s novim posjećenim čvorom, ažuriranom duljinom puta, te smanjenim brojačem gradova koji je još potrebno posjetiti.
- Nakon što funkcija rekurzivno posjeti sve moguće čvorove, vraća se na prethodni čvor i postupak se ponavlja za sve prethodne čvorove.

Konačni ispis prikazuje duljinu najkraćeg trgovačkog puta i pripadni Hamiltonov ciklus.

Primjer iz svakodnevnog života:

Na slici (*Slika 12)* je dan primjer kod kojega je u svakoj hrvatskoj županiji odabrana jedna točka i njezine koordinate. Udaljenosti pojedinih vrhova mogu se izračunati kao euklidske udaljenosti točaka u ravnini. Na ovom primjeru se učenicima može pokazati kako već s relativno malim brojem vrhova, u ovom slučaju zadatak postaje vrlo složen. Primjer je također pogodan jer je povezan sa stvarnim problemom, relativno je lako pripremiti ulazne podatke, a ima i poveznicu s matematikom. Također, potrebno je malo doraditi gornji programski kod i napraviti programski modul koji na temelju koordinata točaka generira matricu međusobnih udaljenosti potrebnu za implementaciju gore opisanog algoritma.



Slika 12: Karta Hrvatske

Podaci:

Z1 = (7.26, 15.58); Z2 = (8.12, 17.29); Z3 = (9.52, 13.96); Z4 = (6.64, 13.53); Z5 = (9.38, 17.84); Z6 = (11.21, 17.48); Z7 = (11.57, 15.32); Z8 = (4.28, 13.7); Z9 = (6.32, 10.71); Z10 = (13.36, 15.68); Z11 = (13.45, 14.09); Z12 = (13.36, 13.19); Z13 = (6.83, 7.95); Z14 = (16.09, 14.45); Z15 = (8.72, 7.38); Z16 = (17.56, 13.07); Z17 = (10.52, 5.76); Z18 = (1.55, 13.28); Z19 = (13.48, 3.18); Z20 = (10.13, 18.56); Z21 = (8.42, 16.04)

9.3. Pokrivenost kurikuluma

Ovaj primjer je primjeren učenicima 4. razreda srednje škole pokrivajući ishod B.4.1. Nastavnik će na ovom primjeru odrediti rekurzivnu relaciju i vizualizirati rekurziju s jednostavnim grafičkim elementima.

Učenici će moći opisati osnovne elemente rekurzivnoga postupka, zapisati matematički opisanu rekurzivnu funkciju u programskome jeziku, uočiti rekurzivnost u danome problemu, odrediti rekurzivnu relaciju i uvjet prekida te realizirati rekurzivnu funkciju u programskome jeziku, procijeniti efikasnost rekurzivnoga rješenja i ovisno o problemu odabrati rekurzivno odnosno induktivno rješenje.

9.4. Vremenska složenost

Ovaj se pristup ne bi smatrao učinkovitim za veće instance TSP-a zbog naglog porasta broja putova koje je potrebno provjeriti. Međutim, za didaktičke svrhe i učenje osnovnih koncepta rekurzije, ovaj kod je primjeren i u skladu je s razinom srednjoškolskog obrazovanja u Hrvatskoj.

Bibliografija

[1] Donald Knuth: Art of Computer Programming, The: Combinatorial Algorithms, Volume 4A, Addison-Wesley Professional; 2011.

[2] Donald Knuth: Art of Computer Programming, The: Combinatorial Algorithms, Volume 4B, Addison-Wesley Professional, 2022.

[3] Milan Balać: Rekurzivni algoritmi, https://dabar.srce.hr/islandora/object/ffos\%3A147

[4] Ivan Milišić: Algoritmi u nastavi informatike u gimnazijama, https://repozitorij.pmf.unizg.hr/en/islandora/object/pmf\%3A4759/datastream/PDF/view

[5] Robert Manger, Miljenko Marušić: Strukture podataka i algoritmi, http://lnr.irb.hr/soya/nastava/spa-skripta.pdf

[6] Ivana Oreški: Rekurzije, https://www.mathos.unios.hr/~mdjumic/uploads/diplomski/ORE02.pdf

[7] Marko Grdičak: Koncept rekurzije u nastavi informatike u osnovnoj i srednjim školama, https://zir.nsk.hr/islandora/object/pmf\%3A3242/datastream/PDF/view

[8] Filip Vuković: Algoritmi najkraćeg puta na grafovima i njihova primjena u cestovnoj navigaciji, https://www.mathos.unios.hr/~mdjumic/uploads/diplomski/VUK39.pdf

[9] Salim Žunić: Pregled algoritama za traženje najkraćeg puta u grafu, https://dabar.srce.hr/islandora/object/fpz:1323

[10] Velga Bosančić, Anka Golemac, Tanja Vojković: Kako pomoći trgovačkom putniku, https://hrcak.srce.hr/file/148131

[11] Odluka o donošenju kurikuluma za nastavni predmet Informatike za osnovne škole i gimnazije u Republici Hrvatskoj, Ministarstvo znanosti i obrazovanja Republike Hrvatske, 2018., https://narodne-novine.nn.hr/clanci/sluzbeni/2018_03_22_436.html

Sažetak

U ovom radu koji obrađuje implementaciju rekurzivnih algoritama u nastavi programiranja, posebno u kontekstu srednjoškolskog obrazovanja, razmotrili smo kako rekurzija kao koncept može obogatiti razumijevanje algoritama kod učenika. Kroz različite primjere, od sortiranja spajanjem, preko algoritama za traženje elementa u listi, do složenijih problema kao što su raspodjela kovanica, traženje najkraćeg puta u grafu i problem trgovačkog putnika, rekurzija se manifestira kao snažan alat za razvoj dubokih analitičkih vještina.

Prednosti korištenja rekurzivnih algoritama u nastavi su višestruke; pružaju temeljit pristup razumijevanju problem rastavljanja na manje potprobleme, osnažuju konceptualno razmišljanje i nude sofisticiran pristup programiranju kroz čist i efektivan kod. Funkcionalno razumijevanje rekurzije omogućava učenicima da intuitivno rješavaju kompleksne probleme koristeći pristupe poput "podijeli pa vladaj", "pohlepni pristup" i "unatražna pretraga".

Kroz ovaj rad nastojimo pokazati da je rekurzija ne samo primjerena, već i ključna u suvremenoj nastavi informatike, pod uvjetom da se izazovi pažljivo navode i premoste putem primjerenih lekcija koje stimuliraju intelektualnu znatiželju i razvoj učenika. Budućnost podučavanja informatike leži u osnaživanju učenika da koriste ove koncepte ne samo kao alate za rješavanje zadanih problema, već kao dio temeljnog razmišljanja unutar šireg spektra znanosti i tehnologije.

Summary

In this work, which deals with the implementation of recursive algorithms in programming education, especially in the context of high school education, we have considered how recursion as a concept can enrich students' understanding of algorithms. Through various examples, from merge sort, to algorithms for searching for an element in a list, to more complex problems such as coin distribution, finding the shortest path in a graph, and the traveling salesman problem, recursion manifests itself as a powerful tool for developing deep analytical skills.

The advantages of using recursive algorithms in education are manifold; they provide a thorough approach to understanding the process of breaking a problem down into smaller sub-problems, strengthen conceptual thinking, and offer a sophisticated approach to programming through clean and efficient code. A functional understanding of recursion enables students to intuitively solve complex problems using approaches such as "divide and conquer," "greedy approach," and "backtracking."

Through this work, we seek to demonstrate that recursion is not only appropriate but also crucial in modern computer science education, provided that challenges are carefully addressed and overcome through suitable lessons that stimulate intellectual curiosity and student development. The future of computer science education lies in empowering students to use these concepts not only as tools for solving given problems, but also as part of fundamental thinking within a broader spectrum of science and technology.

Životopis

Rođen sam u Zagrebu 1984. godine. Osnovnu školu Mato Lovrak sam pohađao u Zagrebu gdje kasnije upisujem i srednju školu Prva tehnička škola Nikola Tesla. Godine 2018. upisujem Prirodoslovno-matematički fakultet, preddiplomski studij matematike i računalstva; smjer nastavnički kojeg završavam 2021. godine. U rujnu iste godine upisujem Diplomski studij matematike i informatike; smjer: nastavnički. Tijekom studija radio sam pretežno kao predavač informatike u Pučkom otvorenom učilištu Zagreb. Od 2022. godine radim u školi za Cestovni promet u Zagrebu kao nastavnik matematike i računalstva.