Algoritam

Treće, korigovano, prošireno i dopunjeno izdanje
BL, 2003

Algoritam

Uvod u programiranje

Namjena i cilj

Priručnik ima dvostuku namjenu. U prve 4 sekcija (do Klasifikacija i tipovi algoritama) date su osnovne definicije i pojmovi i namjenjen je početnicima. Sama sekcija 4 obrađuje elementarne algoritamske strukture reprezentovane u formi dijagrama. Na kraju sekcije 4 imate link koji će Vam omogućiti da pronađete niz primjera jednostavnih programskih struktura i dijagrama toka koji Vas uvode u svijet programiranja.
Slijedi pregled i klasifikacija različitih tipova algoritma kao pokušaj da se na jednom mjestu sagledaju raznovrsne mogućnosti primjene algoritma.

Kasnije sekcije nude ulazak u naprednije načine korištenja algoritma. Clj je da oni koji se bave programiranjem na praktičnom nivou, a nisu stekli predhodnu teoretsku podlogu dobiju uvid u kompleksnije značenje pojma algoritma, da spoznaju mogućnost univerzalnog korištenja algoritma.
U sekciji *U potrazi za definicijom algoritmom* proširena je definicija algoritma sa Tjuringovom mašinom, kao i novim računarskim i algoritamskim paradigmama.

1 Uvod

Porijeklo termina algoritam

Prve matematičke procedure koje bismo mogli nazvati algoritmima sreću se kod starih Grka. Svima su poznati Euklidov algoritam i Eratostenovo sito (250. g. p. n. e.). Dalji razvoj ovakvih metoda seli se na daleki istok u Indiju i Kinu. U Evropu se vraća preko arapske nauke u doba renesanse.

Bio jednom jedan ...

Abu Abdulah Muhamed bin Musa el Horezmi je bio matematičar, astronom i geograf koji je živio u Bagdadu. Bagdad je bio središte carstva, koje se prostiralo od Sjeverne Afrike do Inda. Istovremeno, bio je središte svjetske nauke i u njemu je postojala svojevrsna Akademija nauka poznata kao "Kuća mudrosti" (*Bait al-Hikma*). U njoj su radili i stvarali najpoznatiji i najčuveniji naučnici iz cijelog svijeta. Jedan od njih bio je Ibn Musa Al Horezmi (u standardnoj transkripciji *Mohammed ibn Musa al Khwarizmi*) koji je živio od 780. god do 850. godine. Stvarao je pod uticajem indijskih i grčkih naučnika. Glavno djelo mu je knjiga o algebri *Hisab al-jabr w'al-muqabala*. Iz naziva te knjige izvedena je riječ algebra (*al jabr*).

Monumentalni spomenik El Horezmiju u Hivi

Marka sa likom El Horezmija iz doba Sovjetskog Saveza

Danas Horezmija mnogi svojataju, od Ruske Federacije, preko Iraka i Irana koji se smatraju nasljednicima stare Perzije, do Uzbekistana (koji ga s ponosom veže za grad Hivu, gdje je rođen: Al Horezmi (iz Horezma, drevni grad Hiva (*Khiva*).

Iz netačnog prevoda njegovog djela „Al Horezmi o indijskim brojkama“, koji glasi „Algorithmi de numero indorum“, nastaje latinski korijen riječi algoritam (lat. Algorithmi, -us)

S aspekta moderne matematike problemi koje je rješavao El Horezmi su donekle trivijalni, ali tehnika rješavanja gdje se postavi problem u obliku algebarskih jednačina i primjenom unaprijed zadatih pravila dolazi do rješenja, je suština algoritamskog rješavanja problema, pa se pojam algoritma s pravom veže za ovog matematičara.

2 Algoritam uvodni pojmovi

Bilo kakvo uputstvo, odnosno postupak koji nas postupno, u pojedinačnim koracima vodi do rješavanja nekog problema je algoritam.

Za početak navešćemo primjer, algoritam kojim se u pet koraka opisuje postupak zamjene točka na automobilu:
Primjer1: Zamjena točka
ispitaj ispravnost rezervnog točka,
2. podigni auto,
3. skini točak,
4. postavi rezervni točak,
5. spusti auto

Primjer algoritma je **i bilo koji kulinarski recept**. Da bi kolač uspio neophodno je da se unesu tačni sastojci i obrade tačnim procedurama.

IEEE Standardni rječnik definiše algoritam kao „***Dati skup dobro definisanih pravila ili procesa za rješavanje nekog problema u konačnom broju koraka***“.

2.1 Algoritam i programiranje

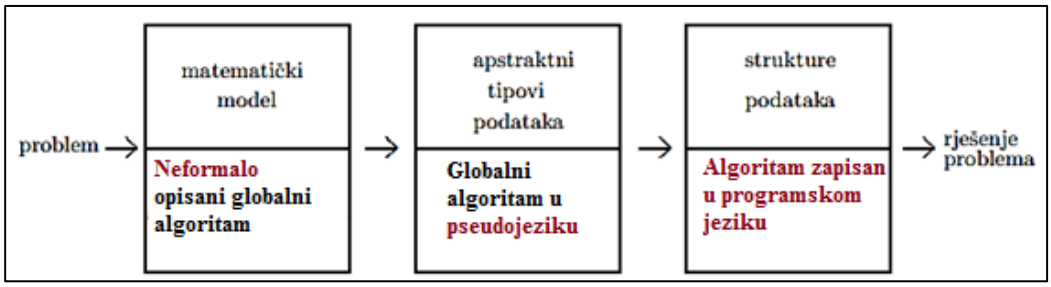
Algoritam je konačan uređen niz precizno formulisanih pravila kojima se rješava jedan ili čitava klasa problema.
Postoji veliki broj različitih matematičkih formalizacija pojma algoritma. Matematički se dokazuje da su sve formalizacije algoritama međusobno ekvivalentne, odnosno **svaki algoritam koji se može predstaviti pomoću jedne od ovih formalizacija, može se predstaviti i pomoću bilo koje druge**. Malo više o ovoj temi može se vidjeti u Dodatku sekcija 10 *U potrazi za definicijom pojma algoritma.*

Svi poslovi koje računar obavlja izvode se postupno, korak po korak, u konačnom vremenu. Svaki korak je jasno preciziran, kao i prelazak na svaki naredni korak. Na kraju postupka, kad računar završi rad (ukoliko se to uopšte desi, jer moguće je i da se rad nikada ne završi), dobije se rezultat.

Program je algoritam zapisan na nekom programskom jeziku, ili kako je to iskazao Wirth jednostavnom formulom:

algoritmi + strukture podataka = programi.

Za razumjevanje algoritma potrebno je (bar dijelom) razjasniti i strukture podataka, jer jedan od osnovnih razloga korištenja algoritma je kreacija programa.



Koraci i način zapisa algoritma i odnos strukture podataka prema formi algoritma u proceduri rješavanja problema

Pojam podatka je bazični pojam u računarstva.
Moglo bi se reći da sve što se nalazi u memoriji je podatak, pa i instrukcije.
Program je uputstvo računaru šta treba da radi.
Izvorni kod programa pisanog na bilo kojem programskom jeziku je samo uređen skup ulaznih podataka drugog programa - kompajlera ili asemblera.

Paralelno sa strukturama podataka egzistira još jedan srodan pojam - tip podataka.
Može se reći da je riječ samo o različitim modelima iste stvari. Dok je struktura podataka, kao model zasnovan na pristupu - skupovima i relacijama, tip podataka se bazira na skupovima i funkcijama.

2.2 Zakonitosti algoritma

Daćemo **obavezujuće osobine (zakonitosti)** koje algoritam treba da ima i tako intuitivni pojam učiniti donekle ekvivalentnim matematičkoj formalizaciji[1].
Svaki dobar algoritam mora da se pridržava osnovnih zakonitosti algoritma:

1. **Posjedovanje ulaza i izlaza.** Za svaki algoritam moraju se definisati ulazi kojih može biti: ni jedan, jedan ili više i izlazi kojih mora biti bar jedan ili više.
2. **Univerzalnost**. To upravo znači da ulazne veličine algoritma mogu uzimati početne vrijednosti različitih skupova podataka, za ma koji izbor vrijednosti ulaznih veličina.
3. **Definisanost-determinisanost (određenost).** Izvršavanje i tumačenje algoritam-skih koraka ne smije zavisiti od volje čovjeka ili mašine. Nakon završetka jednog koraka mora biti definisan prelazak na idući, što omogućava da se algoritam automatski izvršava **(**svi koraci moraju biti jasni i nedvosmisleni).
4. **Konačnost i diskretnost.** Izvršenje algoritma se mora obaviti **u određenom broju koraka**. *Kod složenih algoritama javlja se potreba formalnog dokaza konačnosti algoritma.*

[1] Kasnije ćemo proširiti ovdje date zakonitosti u smislu definicije Markova i Kolmogorova. Ovdje datih 7 zakonitosti su dovoljno dobar osnov.

Algoritam nije prost skup algoritamskih koraka, već je određen i redoslijedom algoritamskih koraka. Proces izvršenja algoritma je diskretan u vremenu što znači da se u svakom vremenskom intervalu izvršava **samo jedan algoritamski korak**. Međutim, u opštem slučaju, kada imamo više izvršilaca (više procesora) postoje i **paralelni algoritmi**, koji omogućuju da se više algoritamskih koraka izvršavaju istovremeno.

5. **Efikasnost** (algoritam se izvršava u razumnom vremenskom intervalu).
6. **Rezultativnost** –usmjerenost. Dobar algoritam je tako definisan da bez obzira na početne (proizvoljne) ulazne vrijednosti, primjena algoritamskih koraka vodi (usmjerava) strogo ka ciljnom rezultatu.
7. **Izvršivost.** Obično se ova osobina definiše onako kako je to uradio D. Knut: *Algoritamski korak je izvršiv ako je čovjek u stanju da ga izvrši za konačno vrijeme (pomoću olovke i papira)*. U formalnoj definiciji algoritma, na osnovu pojma efektivne izvršivosti, daje se opis izračunljivih (algoritamski rješivih) problema.

Za početak ćemo kao ilustraciju dati poznati primjer pravila koje nije izvršivo: *Ako razvoj broja π sadrži 7 uzastopnih devetki saberi sve preostale cifre*. (Pošto je π iracionalan broj ovaj problem je rješiv, ali nije izvršiv.)

Kod algoritamski rješivih zadataka nema neizvršivih koraka.
U procesu programiranja, skup akcija definisan je mogućnostima računara, odnosno naredbama programskog jezika koji se koristi, dok se redoslijed izvršavanja akcija zadaje pomoću algoritamskih (programskih) struktura.

2.2.1 Otkrivanje i definisanje algoritma

Za definisanje algoritma obično se nudi opšta procedura, koju je 1945. definisao mađarski matematičar **Polya**, koja glasi:

1. Shvati problem
2. Napravi plan za rješavanje
3. Provedi plan
4. Provjeri tačnost plana i procijeni njegovu univerzalnost

Polya-in koncept[2] primijenjen na razvijanje programa glasio bi:

1. Shvati problem
2. Smisli kako problem predstaviti algoritamski (otkrij odgovarajući algoritam)
3. Formuliši algoritam (dizajniraj program koristeći algoritam. Treba težiti da instruktivni odnos algoritam : program bude 1:1)
4. Provjeri tačnost i univerzalnost (unesi podatke koji će verifikovati ispravnost programa)

[2] Prevashodno je to koncept rješavanja matematičkih problema, a koristi se kao metod pristupa kako učenicima približiti rješenja matematičkih problema, ali daje mogućnost i da se problem „predstavi i objasni“ mašini-računaru.

Da bismo dali odgovor na pitanje kako napraviti dobar algoritam, trebali bismo prvo definisati pojam dobrog algoritma. Mogli bismo reći da je to onaj algoritam koji zadovoljava tri osnovna uslova:

1. Uvijek daje tačan rezultat
2. Rješava problem u najkraćem mogućem vremenu
3. Razumljiv je ostalima

Ovo su jednostavni, ali kod iole kompleksnijih problema, **vrlo zahtjevni uslovi**.

Postoji i teorijsko ograničenje koje se odnosi na provjeru tačnosti i univerzalnosti. Američki matematičar Rajs (*Henry Gordon Rice;* 1920-2003) je sredinom dvadesetog vijeka dokazao teoremu koja tvrdi da, za bilo koju netrivijalnu osobinu parcijalnih funkcija, nema opšteg i efektivnog metoda koji može da odluči da li postoji algoritam koji izračunava parcijalnu funkciju sa tom osobinom.
Bez dubljeg ulaženja u ovu tematiku, recimo da bi ovdje parcijalnom funkcijom mogli smatrati funkciju čiji egzaktan domen nije poznat, a trivijalna osobina bi bila ona koja važi za sve parcijalne izračunljive funkcije ili nijednu. Krajnje uprošteno, možemo reći da iz ovoga slijedi da ne postoji univerzalan algoritam koji bi služio za rješavanje bilo kog problema.
Za više o ovome vidi npr. https://en.wikipedia.org/wiki/Rice's_theorem

2.2.2 Algoritam i obrada podataka na računaru

Kad se uz algoritam vežemo za obradu podataka na računaru, podrazumijeva se da se podatak prvo učita preko ulazne jedinice, a ispisuje se na izlaznu jedinicu ili se pamti za kasniju upotrebu. Zapamćeni podaci se smatraju dijelom unutrašnjeg stanja sistema.
Za svaki računarski posao algoritam mora biti jasno definisan; naveden na način koji podrazumijeva sve moguće situacije koje se mogu pojaviti.
Znači, svaki uslovni korak se mora sistematično obraditi, slučaj po slučaj; uslov za svaki slučaj mora biti jasan i izračunljiv (matematički definisan).
Pretpostavlja se da su instrukcije navedene jasno, **da počinju *od vrha* i da teku *do dna***[3]. **Ova ideja se formalno opisuje kontrolom toka.**
Ovo je najuobičajeniji koncept u programiranju i opisuje postupke na mehanički način. Kod ovakve formalizacije se unaprijed uzimaju pretpostavke o imperativnom programiranju.

Tehnike programiranja su istovjetne s programskim stilovima i direktno su povezane s pojedinim programskim jezicima. Imperativno programiranje opisuje računanje kao izraze koji mijenjaju stanje programa. Kao što se u govornom jeziku zapovijedni način (ili imperativ) koristi za izražavanje naredbi, tako se imperativni programi[4] *mogu posmatrati kao niz naredbi koje računar treba izvršiti. Jedinstveno za ovaj koncept je* ***operacija dodjeljivanja****, što je davanje vrijednosti promjenljivoj. Ovo proizilazi iz intuitivnog poimanja memorije kao privremenog skladištenja odnosno pamćenja*.

Algoritam je uvijek apstraktniji od načina na koji ga predstavljamo.
Ovo ćemo razjasniti prostom analogijom sa pričom i knjigom.
Priča predstavlja apstraktni pojam, a knjiga je formalna (fizička) manifestacija priče.
Ako knjigu prevedemo na drugi jezik, mijenja se njena predstava, ali priča ostaje ista.
Idući primjer je možda univerzalniji. Uzmimo notni zapis neke melodije. On je isti, ali različito će zvučati u zavisnosti od instrumenta na kojem se izvodi.

O odnosu forme i sadržine mogli bi potrošiti i stotine strana i opet ostati nedorečeni, zato ćemo biti praktični. Prezentovaćemo dva algoritma, koja definišu jedan jednostavan problem.

Primjer 4: Napraviti algoritam koji stepene Celzijusa pretvara u stepene Farenhajta.
Algoritam 1 definisan opisom: pomnoži temperaturu u stepenima Celzijusa sa (9/5) i proizvodu dodaj 32
Algoritam 2 definisan formulom: F=(9/5)*C+32
Zavisno od obrazovanja, upućenosti u problem i slično, pojedincima bi više odgovarao algoritam1, od algoritma 2 i obrnuto: neko voli klavir, a neko harmoniku.

[3] Top-down model je bazični kod imperativnog programiranja. Vidi sekciju Modularnost
[4] Mada je koncept imerativnog programiranja bitno modifikovan, trebalo bi da ga razumijete.

2.2.3 Pojam apstrakcije i apstrahovanja

Definisanje misaonih postupaka je domen logike i ovdje ćemo dati **uprošćene** definicije apstrakcije, konkretizacije i specijalizacije, koje će nam pomoći u shvatanju pojma apstrakcije potrebnog za rad na računaru.
Apstrakcija predstavlja misaoni postupak izdvajanja bitnih svojstava objekta ili pojave (iz stvarnog svijeta), od ostalih svojstava objekta koja nisu bitna.
Predmet apstrakcije: opšte u posebnom i posebno u opštem kao moguća izdvojena cjelina.
Često se umjesto pojma apstrakcija koristi pojam **apstrahovanje**[5] u smislu "izvršiti apstrakciju" kad se želi naglasiti da je apstrakcija proces odbacivanja nebitnih svojstava objekta.
Konkretizacija je proces suprotan apstrakciji. Ona predstavlja određivanje opšteg na osnovu osobina pojedinačnog.
Apstraktan pojam ističe formalnu oznaku predmeta, bez nosioca, npr. vještina, očinstvo, brzina.
Konkretan pojam ističe nosioca, npr. majstor, otac, brz auto.
Specijalizacija je shvatanje posebnog u opštem preko posebnog.
Specijalizacija pojma se sastoji u shvatanju posebnih predmeta iz određene opšte grupe predmeta.

Da bismo radili sa računarom treba da razmišljamo apstraktno.
Uopšteno bi se moglo reći: Softver je računarska apstrakcija nekog dijela realnog svijeta.
Nešto konkretnije mogli bismo reći da rješavanje problema počinje analizom, a nastavlja se apstrahovanjem. U računarstvu, apstrahovanje znači izostavljanje nekih atributa, a specijalizacija dodavanje novih atributa.

Algoritam u sebi nosi univerzalno rješavanje problema, pa je mogućnost apstrakcije i apstrahovanja nešto što se podrazumjeva.

[5] apstrakcija (latinski *abstractio*: izdvajanje, izvlačenje).
Apstrahovanje ili apstrahiranja – (latinski *abs-trahere* = vući van, odvajati, izdvojiti; izdvajanje)

3 Predstavljanje: zapis algoritma

Na osnovu apstraktne prirode algoritma, jasno je da se algoritam može prezentovati na mnoštvo načina. Prije nego što objasnimo načine prezentovanja, još jednom ćemo naglasiti dva osnovna cilja algoritma:

- predstavljanje problema na razumljiv način dekompozicijom problema u algoritamske korake
- mogućnost da se na osnovu algoritma može napisati program

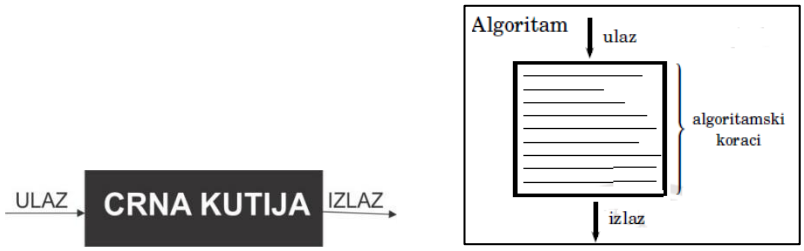
Da bi se ciljevi ostvarili potrebno je da algoritam bude zapisan. Zapis algoritma je moguć na mnogo načina, a obično se koriste tri:

1. skup pravila i rezervisanih riječi iz govornog jezika
2. pseudokod
3. dijagram toka

Kao četvrtu formu zapisa algoritma možemo smatrati program shodno definiciji 3. Mogli bismo ustvrditi da je u slučaju računara program finalna forma algoritma.

3.1 Model crne kutije i lego kockica

Kao uslov da nastavimo sa analizom izrade algoritma potrebno je da definišemo elemente koji čine algoritam.
Daćemo pregled osnovnih elemenata strukture algoritma.
Algoritam definišemo kao niz naredbi.
Da bi olakšali praćenje algoritma podijelićemo nizove na logičke cjeline koje olakšavaju praćenje algoritma.
Algoritam može da se zasniva na principu crnih kutija i lego kockica.



Slika 4 Crna kutija

Crna kutija je sistem koji se može posmatrati u smislu njegovih ulaza i izlaza bez ikakvog znanja o njegovom unutrašnjem radu. Njegova implementacija je "neprozirna" (crna).
Za analizu otvorenog sistema sa tipičnim pristupom "crne kutije", samo ponašanje stimulusa/odgovora će biti uzeto u obzir, da bi se zaključila (nepoznata) kutija.

Uobičajeni prikaz ovog sistema crne kutije je dijagram toka podataka centriran u kutiji.
Definisaćemo kockice kao elementarne logičke strukture. Svaka kockica odgovaraće nekoj logičkoj cjelini.
Potrebno je definisati ulaze i željene izlaze, a unutrašnjost crne kutije shvatiti kao niz naredbi, algoritamskih koraka koji vrši transformaciju ulaza u odgovarajući izlaz.

Pod elementarnom algoritamskom strukturom smatraćemo niz naredbi koji definiše jedinstven rezultat, koji se koristi kao međurezultat: kao jedan od rezultata između početka i kraja procesa.

Pravljenje algoritma možemo shvatiti kao slaganje algoritamske strukture, koja se sastoji od niza elementarnih kockica.
U slučaju da je problem jednostavan, jedna kockica, može obuhvatati više elementarnih struktura. U slučaju da je problem kompleksan, jedna elementarna struktura se može podijeliti na više cjelina.
Ako niste shvatili: i sama predstava algoritma (osnovni podaci, pseudokod, dijagram toka, opis, ili nešto treće) je određena stepenom složenosti problema.

Mi ovdje koristimo algoritam kao prvi-uvodni korak u programiranje. Kao polazni i osnovni model koristi se princip crne kutije (*Black box*) On je najčešći i najstarijji, ali samo jedan od modela programiranja.

Kod tzv. reaktivnih programa (agenata) koristi se tzv. princip zelenog kaktusa, gdje programi primaju poruke iz okruženja i reaguju na njih. Ovakav princip interakcije je obavezan kod programa kod kojih je bitna pouzdanost i brzina rada kao što su programi za nadzor i konrolu, gdje na promjene ulaza mora da se reaguje hitno i odmah; npr. programi za kontrolu nuklearnih centrala, upravljanje ventilacijom u rudnicima, protupožarni nadzor i slično.

Koriste se i neki drugi principi, npr. princip prozirne čaše (*White glass)* gdje korisnik vidi programski kod i može ga mijenjati; npr kod Linux operativnog sistema.

3.2 Zapis algoritma skupom pravila i rezervisanim riječima

Kad ljudi jedan drugom nešto objašnjavaju koriste govor (odnosno govorni jezik). Za predstavljanje algoritma potrebna nam je nekakva forma slična jeziku. Osnovna mana takve prezentacije je mogućnost da pojedine riječi budu pogrešno shvaćene zbog višeznačnosti koja karakteriše sve govorne jezike.
Da bismo kod zapisa algoritma izbjegli problem višeznačnosti trebamo uvesti skup rezervisanih riječi i skup pravila.

Rezervisana riječ (identifikator) je riječ u programskom jeziku koja ima fiksirano značenje i ne može biti predefinisana od strane programera, što znači da programer ne može uzimati imena funkcija, labela, slogova i sl. iz skupa rezervisanih riječi. Neke od rezervisanih riječi u programskim jezicima su **if**, **then**, **else**, **integer**, **real**, **true**, **false**. Blizak pojam rezervisanoj riječi je ključna riječ.

Razlika između rezervisanih riječi i ključnih riječi dolazi do izražaja prilikom prevođenja programa, u što ovdje nećemo ulaziti. U većini modernih programskih jezika skup ključnih riječi je podskup skupa rezervisanih riječi. U jezicima *C* i *Python* ovi se skupovi poklapaju, a u jeziku *Java* je skup ključnih riječi pravi podskup skupa rezervisanih riječi. Takav tretman ključnih riječi olakšava prevođenje programa, jer ključne riječi ne mogu biti pobrkane sa identifikatorima koje je uveo programer. U nekim starijim jezicima (kao što je napr. *FORTRAN*), ključne riječi nisu rezervisane riječi pa u programu mogu biti korištene i kao identifikatori koje uvodi programer. To otežava prevođenje programa.

Rezervisane i ključne riječi imaju **sintaksu** i **semantiku**. Sintaksa predstavlja pravila pisanja riječi i rečenica, a semantika (značenje) riječi zadaje se u opisu programskog jezika i određuje način upotrebe riječi.

Osnovni (**prosti**, **primarni**) **tipovi podataka** se definišu ključnim riječima. Klasični osnovni tipovi podataka mogu uključiti karaktere (**character**, **char**), cijele brojeve (**integer**, **int**, **short**, **long**, **byte**), realne brojeve različitih preciznosti (**float**, **double**, **real**, **double precision**), realne brojeve fiksirane preciznosti (**fixed**) koja može biti unaprijed izabrana. Osnovni tipovi služe kao dijelovi od kojih se prave složeni tipovi za rad sa podacima u programskom jeziku, kao na primjer nizovi, slogovi, liste.

Daćemo osnovna pravila i simbole koji se mogu koristiti kao univerzalni jezik za zapis algoritma[6]:

1. Pravilo dodjeljivanja : `<promjenljiva> := <izraz>`
Vrijednost izraza koja se izračunava se dodjeljuje varijabli sa lijeve strane simbola dodjeljivanja := . Umjesto := mogu se koristiti simboli = , ← , ⇐.

[6] ovdje smo **namjerno** umjesto uobičajenih engleskih termina dali „naše“ sa namjerom da pokažemo da se kao ključne riječi koriste riječi iz govornog jezika

2. Pravilo uslovnog grananja : `Ako <uslov> tada <pravilo>`

3. Pravilo grananja : `Ako <uslov> tada <pravilo1>`
`inače <pravilo2>`

4. Pravilo bezuslovnog grananja (skoka) : `Skoči na <oznaka pravila>`

5. Pravilo za prekid izvršenja algoritma : `Kraj (Stop)`

6. Pravilo za ulaz podataka : `Ulaz (<lista ulaznih varijabli>)`

7. Pravilo za izlaz podataka : `Izlaz (<lista izlaznih varijabli>`

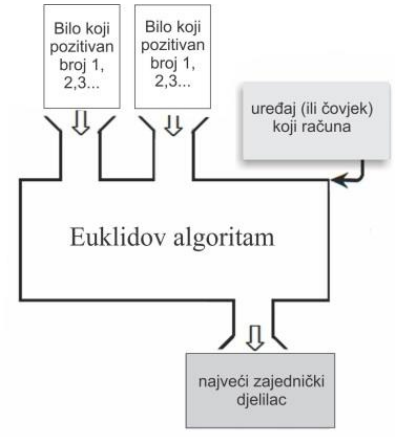
8. Pravilo ciklusa : `Ponavljaj`
```
<pravilo 1>
<pravilo 2>
. . .
<pravilo n>
              Sve dok se <uslov> ne ispuni
```

Algoritamska pravila se standardno obilježavaju brojevima, tako da se u „nultom“ redu napiše ime algoritma, a potom se izvršavaju redoslijedom (1,2...) dok se ne dođe do pravila za grananje ili prekida.

Primjer 5. Euklidov algoritam za nalaženje najvećeg zajedničkog djelioca. Na Slici 2 je data grafička ilustracija koja vas podsjeća na cilj zadatka, a u listi zapis algoritma koji koristi ključne riječi.

Algoritamski zapis:

```
E0. Euklid
E1. Ulaz (m,n)
E2. r:= m MOD n {ostatak dijeljenja}
E3. Ako r=0 tada predji na E7
E4. m:=n
E5. n:=r
E6. Predji na E2
E7. nzd:=n
E8. Izlaz (nzd)
E9. Kraj
```



Slika 2 Euklidov algoritam

3.3 Pseudokod

Programski jezik čini skup osnovnih oblika zajedno sa skupom pravila koja definišu kako se osnovni oblici mogu koristiti. Ako se vratimo pojmu algoritma jasno je kako se on može predstaviti: **napišete program**. Da bi to uspješno uradili neophodno je da se upoznate sa kompletnim instrukcionim setom, da ovladate tehnikama kreacije složenih tipova. **Prosti tipovi su osnovni blokovi za izgradnju struktura podataka.** Tako, kao što smo već rekli, strukture kao što su klase i nizovi zovemo složeni tipovi.

Umjesto formalne, stroge strukture programskog jezika, često se koristi **manje formalan sistem notacije za predstavljanje algoritma**, poznat kao pseudokod.
Osnovna svrha pseudokoda je da omogući da lakše slijedimo put od ideje do konačne realizacije.
Jedan od načina predstavljanja pseudokodom je izostavljanje pravila iz naredbi koja prate svaki formalni programski jezik.
Ovaj način se koristi kada znamo koji ćemo programski jezik upotrijebiti, pa je tako struktura kojom opisujemo algoritam velikim dijelom nalik, ali je manje formalna od programskog jezika.

Ako želimo zapis za predstavljanje algoritma nezavisan o programskom jeziku, tada se obično služimo principom kutija i slagalica, gdje na intuitivan način predstavljamo pojedine dijelove algoritma.

Pseudokod možemo definisati kao intuitivni način predstavljanja algoritma kod kojeg ipak postoje određena pravila zapisa. Poželjno bi bilo da formalizujemo oznake za određene logičke (semantičke) strukture. Da bi bilo još čitljivije, pseudokod pišemo sa komentarima koji dodatno opisuju pojedine strukture.
Bitno je da se ne izgubi **osnovna namjena pseudokoda**: on **treba da posluži kao pomoć za predstavljanje ideja**.

Euklidov algoritam pseudokodom možemo predstaviti kao na Tabeli l.

1: procedure Euclid(a,b)	Pronađi NZD
2: r ← a mod b	
3: while r≠ 0 do	tačno kad je r=0
4: a←b	
5: b←r	
6: r←a mod b	
7: end while	
8: return b	b je NZD
9: end procedure	

Tabela 1 Euklidov algoritam predstavljen pseudokodom (sa komentarima)

4 Organigram: Dijagram toka

Dijagram toka ("***flowchart***"), poznat i kao **organigram**[7], možemo shvatiti kao specijalnu formu pseudokoda.
Zbog svoje jednostavnosti ovo je načešće korištena forma predstavljanja algoritma.

Dijagram toka je simbolički predstavljen algoritam. Može se reći: **dijagram toka je grafički prikaz algoritma**. Sastoji se od niza simbola povezanih strelicama (spojnim putevima) koji definišu tok i smjer realizacije programa.
Dijagram toka prikazuje korake procesa u logičkom nizu. Dijagram toka je grafička prezentacija koja ukazuje na algoritam. Programeri ga često koriste kao alat za rješavanje problema. Dijagram toka koristi različite obrasce za ilustraciju operacija i procesa u programu.
Grafički prikaz je jednostavan, pregledan, lako se pronalaze greške. Nadalje, problem se može jednostavno analizirati, uporediti s nekim drugim problemom, što će nam skratiti vrijeme pronalaženja rješenja.

Dijagram toka je slikovni prikaz algoritma. Dijagrami toka, uz pomoć različitih grafičkih obrazaca, lakši su za razumijevanje i jednostavniji za korištenje. Osnovni elementi (simboli) i izgled jednostavnog dijegrama toka su prikazani na Slici 3.

Početak programa

Ulaz

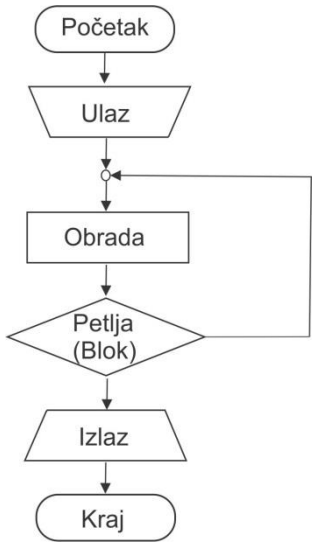
Čvorište

Obrada

Petlje ili blokovi

Izlaz

Kraj programa



Slika 3 Dijagram toka

[7] Pojam organigram, uopšte, odgovara grafičkom prikazu neke organizacije pa se često koristi za prikaz hijerarhiju pozicija zaposlenih unutar organizacije, pa je možada bolje koristiti dijagram toka da se izbjegne zabuna.

Postoje različite konvencije, ali pri izradi dijagrama toka se uglavnom koriste četiri simbola:
1. Oval
Kraj ili početak prilikom kreiranja dijagrama toka
2. Pravougaonik
Korak u procesu izrade dijagrama toka
Pravougaonik je glavni simbol koji predstavlja bilo koji korak u procesu koji želite predstaviti.
3. Strelica
Označava usmjereni tok
Strelica se koristi za vođenje i praćenje putanje dijagrama toka.
4. Romb-Dijamant
Romb simbolizira da je potrebno donijeti odluka kako da se krene dalje. Ovo može biti binarni, (DA/NE) izbor ili složenija odluka s višestrukim izborima.
Sa ova četiri osnovna simbola, vjerovatno imate sve što vam je potrebno da započnete kreirati svoj vlastiti dijagram toka.

4.1 Lista podataka

Daćemo definiciju liste podataka kao osnovne strukture ulaznih i izlaznih podataka.
Lista je konačan skup od n (n>0) **elemenata**, koje možemo označiti sa a_1, a_2, ..., a_n, čija je osnovna strukturna osobina da su **linearno uređeni**.

Ako je n>0 onda je element a_1 prvi element u listi, a_n je posljednji element u listi, a za a_k, gdje je kažemo da je k-ti element, pri čemu se ispred njega nalazi a_{k-1} a iza njega a_{K+1} element, pri čemu je $1<k<n$.

U opštem slučaju svi elementi liste su istog tipa.

Postoje različiti načini predstavljanja liste.
Najčešće se koristi prikaz liste sa zagradama, gdje se elementi liste odvajaju zarezima.
Npr.:
- lista brojeva B=(1, 7, 13, 2, 7, 65, 35),
- lista slova S=('A,F,G,L,B,M,Q,S')
- lista imena I=('Pero, Violeta, Jovo, Simo, Tomo')
- lista listi L=(('A, B, D, F'), ('G, H, J'))

Prvi element liste se često naziva glava liste (head). Obično se smatra da je to prvi element na lijevom kraju liste. Kada se iz zadate liste odstrani glava liste, dobijamo listu koja se zove rep (tail) liste.
Primjer: Neka je data lista S=(’C’, ’A’, ’T’, ’S’), glava liste S je element ’C’, a rep liste je lista (’A’, ’T’, ’S’).

Da bi matematički pojam liste pretvorili u apstraktni tip podataka, potrebno je definisati operacije koje se obavljaju na listama.
Tako možemo definisati listu čiji su elementi slova, brojevi, slogovi, liste itd.

Osnovne operacije za rad na listama mogu se podjeliti u tri grupe:
- operacije kreiranja liste (konstruktorske operacije),
- operacije postavljanja upita nad listom (predikatske operacije),
- operacije selekcije dijelova liste (selektorske operacije).

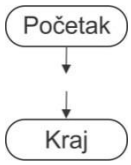
Npr.:
- niz brojeva B=(1, 7, 13, 2, 7, 65, 35),
- niz slova S=(’A,F,G,L,B,M,Q,S’)
- niz imena I=(’Pero, Violeta, Jovo, Simo, Tomo’)
- višedimenzionalni niz L=((’A, B, D, F’), (’G, H, J’))

4.2 Početak i kraj

Prva algoritamska struktura je najprostija, označava početak i kraj algoritma (programa).

Početak i kraj možemo shvatiti kao specijalni slučaj: niz od jedne naredbe.

Na početku i na kraju zapisa svakog algoritma nalaze se naredbe koje označavaju prekide: start i stop. Ovo su polazni i završni koraci svakog algoritma i svi ostali koraci se nalaze između njih.

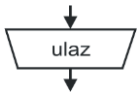


Slika 5 Simboli za početak i kraj algoritma

Pseudokod označavanje ovih naredbi je obično početak (**start)**, odnosno kraj (**end)**, a grafički simboli kod dijagrama toka su po ISO notaciji standardno kao na Slici 5.

4.2.1 Definisanje ulaza i lista ulaznih promjenljivih

Nakon što se definiše početak potrebno je definisati ulazne promjenljive.
Definisanje ulaza, na osnovu kojih će se graditi struktura algoritma, umnogome određuje problem.



Slika 6 Simbol za ulazne promjenjve

U zavisnosti od koncepta jezika i predstave (dijagram toka, pseudokod ili nešto treće) zavisiće način deklaracije ulaznih promjenljivih. Simbol koji se koristi za prikaz ulaznih promjenjivih predstavlja se trapezom koji sadrži identifikatore (listu imena ulaznih varijabli), kao na Slici 6.

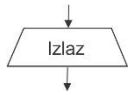
U svakom slučaju, da bismo lakše mogli da slijedimo tok algoritma potrebno je da ulaznim varijablama damo neka imena. Obično se ta imena nazivaju **identifikatori**. Imena bi trebalo da nas asociraju na sadržaj varijabli. U složenijim slučajevima, kad varijablama pridružujemo uobičajena matematička imena (x,y...) poželjno je da se koriste komentari, kojima se opisuje namjena promjenljivih.

Osim pridruživanja imena po potrebi se definiše i tip varijable. To se obično obavlja u formi: ime = tip (varijable)
Npr. `Prezime=Character(20)` značiće da smo za promjenljivu Prezime predvidjeli znakovni tip maksimalne dužine 20 karaktera.

Definisanje ulaznih promjenljivih je jedan od najvažnijih koraka pri izradi algoritma. Ovdje treba naći pravu mjeru. Analiza potrebnih ulaza je praktično analiza problema, ali traženje svih finesa može da uzrokuje zastoj. Pravu mjeru je nemoguće preporučiti. Definisati potrebne ulaze značilo bi definisati listu ulaznih promjenljivih i njihovu strukturu (ime, tip i **domen** ulaznih i izlaznih varijabli).

4.2.2 Definisanje izlaza

Pod izlaznom promjenljivom ćemo smatrati izlaz koji nastaje obradom ulaznih promjenljivih. Izlaz može da se koristi za prikaz rezultata (na ekranu, štampaču) ili kao međurezultat, koji omogućava dalje računanje.

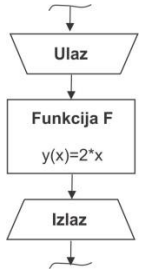


Slika 7 Izlaz

Simbol koji se koristi za prikaz izlaznih promjenjivih predstavlja se ekvivalentno ulazu: trapezom koji sadrži identifikatore (listu imena izlaznih varijabli), kao na Slici 7.

Izraz koji definiše izlaznu promjenljivu može biti naredba ili složena naredba (blok naredbi). Standardno, izlaz je definisan listom izlaza, koja odgovara listi ulaza.

Primjer 6



Slika 8 Računanje izlaza

Za Y=f(X)=2*X ulaz je neka veličina X a funkcija na izlazu daje izračunatu vrijednost za taj X.
Za ulaz X=2 izlaz će biti Y=4.
za ulaz 3, Y će biti 6
za ulaz 4 izlaz će biti 8 itd..

Ovaj primjer nema praktičnu vrijednost osim pokazne: možemo vidjeti predstavljanje funkcije i kreiranje izlaza na osnovu poznatog ulaza.

Za razliku od **ulaza** koji se definiše **deklarativno**, vrijednost **izlaza je izvedena veličina**. Vrijednost izlaza (njegova veličina, tip i domen) definisana je ulazima i izrazima koji ga čine. To znači da treba biti pažljiv i voditi računa da operacije u korištenim izrazima budu definisane.

4.3 Elementarne algoritamske strukture

Pomoću algoritamskih struktura se zadaje redoslijed izvršavanja akcija. Postoje tri elementarne algoritamske strukture:

- linijska,
- razgranata i
- ciklična.

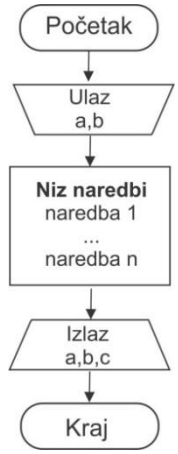
4.3.1 Linijska struktura i definicija sekvence

Linijska struktura je osnovna logička struktura i odgovara izvršavanju niza naredbi. To je elementarna struktura kod koje se svaki algoritamski korak izvršava tačno jednom.

Ilustrovano pseudokodom uopšteno bi bilo:

Akcija1
Akcija2
.
.
.
AkcijaN

Postoji samo jedna grana izvršavanja.



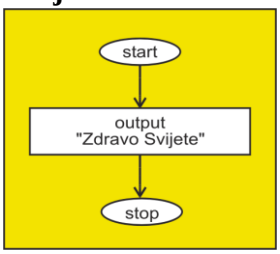
Slika 9 Dijagram toka sa linijskom strukturom

Linijsku strukturu čini **niz akcija** (naredbi) jedna iza druge, **koji čini sekvencu** i predstavlja osnovnu strukturu algoritma.
Akcija se definiše izrazom iz seta naredbi koje je moguće izvesti.

4.3.2 Algoritam Zdravo Svijete

Kombinujući proste algoritamske strukture u jednu linijsku u prilici smo da napravimo algoritam koji se koristi kao početni tzv. **Zdravo svijete** (***Hello World***) algoritam, kojim se **tradicionalno** započinje svaki praktičan rad u nekom programskom jeziku

Primjer 7



Mogući algoritam za ispis
u pseudo-jeziku:

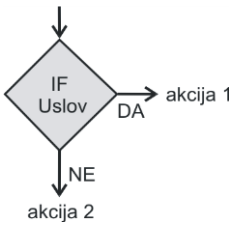
input: /*nema podataka sa ulaza*/
output: poruka /*Zdravo Svijete*/
{
output:" Zdravo Svijete!"
}

Slika 10 Dijagram Hello World

Zasad ćemo ostati na intuitivnoj jasnoći ovog primjera.

4.3.3 Razgranata struktura: uslovno grananje

Razgranata struktura će se pojaviti kod algoritma koji ima blok odlučivanja. Razgranate strukture **dopuštaju donošenje odluke**. To su strukture koje nude mogućnost izbora jedne od dvije (ili više) ponuđenih vrijednosti na temelju zadovoljenog uslova.



Slika 11 Dijagram IF strukture

Pseudokod za 2R (dvostruku razgranatu) strukturu:

```
If (uslov)
  Then
     Else (akcija2)
Endif
```

Koristimo ključne riječi **if**, **then** i **else** da bismo naznačili podstrukture u strukturi. U opštem slučaju to je elementarna razgranata struktura sa n grana kod koje će se akcija na jednoj od grana izvršiti jednom, dok se akcije na ostalih n-1 grana neće izvršiti nijednom. *Možemo, (a ne moramo) da koristimo zagrade da bismo naznačili granice podstruktura. To zavisi od sintakse i određene semantike pseudokoda.*

Primjer 8

Tipičan primjer ovakve strukture definiše algoritam rečenicom:
Ako promet robom raste, naručuje se uobičajena količina proizvoda, inače, uskladištenu robu ponuditi po nižoj cijeni.

Ako koristimo pseudokod to je već napisana 2R razgranata struktura:

```
If (uslov1) Then (akcija1)
                Else (akcija2)
```

gdje uslov1 možemo definisati kao poređenje vrijednosti prometa u određenom periodu sa referentnim prometom, akciju1 kao naručivanje uobičajene količine, a akciju2 kao umanjenje cijena u odnosu na deklarisane.
Zbog preglednosti se preporučuje nazubljena struktura notacije pseudokoda, koja omogućava da se lakše prati koja akcija odgovara kojem uslovu.

Pseudokod za 3R (trostruku razgranatu) strukturu:

```
If uslov1 Then
      If uslov2  Then
           Grupa1 akcija
          Else
             Grupa2 akcija
       Endif
          Else
             Grupa3 akcija
Endif
```

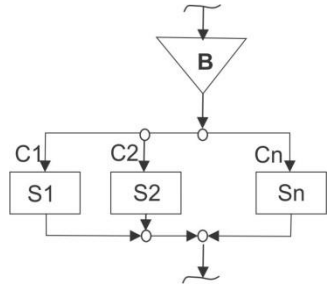
Pravilno postavljanje pitanja omogućava jednoznačne odgovore, što je osnov uspješnog algoritma, odnosno programiranja. Ovdje je bitno primijetiti da odgovor ne mora biti jedan. Uslov je **logički**, a tačan je ili netačan u zavisnosti od toga da li je neka relacija ili logička formula zadovoljena.

4.3.4 Višestruko uslovno grananje

Uopšteno uslovno grananje naziva se **prekidačka/switch naredba grananja**, ili višestruko uslovno grananje.
Iako se SWITCH izraz koristi znatno rjeđe nego **if** izraz, vrlo je koristan za grananja u više grana.
Switch izraz omogućuje procjenu uslova i na osnovu te vrijednosti skok na neko mjesto unutar switch izraza. Ova je struktura pogodna u slučaju kada je potrebno više puta ispitivati istinitost izraza, a na osnovu jednog argumenta.

Pseudokod ove strukture je:

```
switch izraz
    case test_izraz1
        naredbe..
    case test_izraz2
        naredbe2...
    otherwise
        naredbe3
 end
```



Slika 12 Dijagram switch/case strukture

Ovdje izraz mora biti skalar ili znakovna varijabla. Ukoliko je **test_izraz1** (B kod dijagrama toka) **istinit** izvršavaju se **naredbe1** (kod dijagrama toka S1 definisane labelom c1**)**, a ukoliko taj izraz nije istinit prelazi se na testiranje izraza **test_izraz2** i ako je on **istinit** izvršavaju se **naredbe2**, a ako ne (dakle nije istinit **ni** test_izraz1 **ni** test_izraz2) izvršavaju se **naredbe3.**

4.3.5 Ciklične strukture: Ponavljanje akcija u petlji

Ciklični algoritam će se pojaviti kada treba isti posao uraditi više puta. Da bismo olakšali kreiranje algoritma, bez pisanja linijskih struktura koje se neprekidno ponavljaju, kreiramo programsku petlju. Programske petlje se koriste u slučajevima kad ne želimo pisati linijske naredbe više puta, liniju po liniju.

Postoji više varijanti ciklične srtukture a dvije osnovne podjele su:

- Iteracija se vrši dok se ne zadovolji određen uslov
- Broj iteracija (ponavljanja) je unaprijed poznat

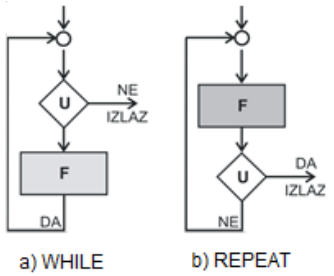
4.3.6 Uslovne ciklične strukture

Ciklične strukture su predstavljene skupom instrukcija koje se ponavljaju u petlji, a među njima se nalazi i instrukcija koja označava sve instrukcije iz petlje.
Forma petlje ima tri komponente:

- inicijalizacija početnog stanja
- testiranje trenutnog stanja sa traženim stanjem
- modifikovanje trenutnog stanja prema traženom

Postoje dva tipa uslovnih cikličnih struktura koje predstavljamo petljama (tzv. **pretest loop** i **posttest loop**), a u pseudokodu ih predstavljamo sljedećim formama:

- **REPEAT (aktivnost) UNTIL (uslov)**
- **WHILE (uslov) DO (aktivnost)**



Dijagram toka ovih struktura na Slici 13 omogućava da ih lakše shvatimo.
Izraz F predstavlja niz naredbi (akciju) koja se interaktivno mijenja, a U uslov koji treba da se zadovolji.
Vidimo da uslov može da se ispituje na početku ili na kraju ciklusa.

Slika 13 Dijagram While i Repeat strukture

Pogledajte rješenje istog problema korišćenjem različitih struktura dato u Tabeli2

`i ← 0`
`while (i!=2) {print(i); i ← i+1}`
 	
`0`
`1`
`i`
`2`
	`i ← 0`
	`repeat {if (i==2) break; print(i);`
	`i ← i+1}`
	
	`0`
	`1`
	`i`
	`2`

Tabela 2

Razlika između ove dvije varijante je što kod provjere istinitosti uslova na kraju petlje program mora proći bar jednom kroz petlju.
Kod provjere istinitosti na ulazu u petlju postoji mogućnost da se ne izvrši ni jedno ponavljanje. Razlog je jednostavan, ako uslov nije ispunjen odmah se "preskače" na sljedeću liniju izvan petlje. Za ovu vrstu petlje u većini programa se koristi ključna riječ "WHILE" prilagođena sintaksi programskog jezika.
Kod provjere istinitosti na kraju petlje, ne postoji mogućnost da se ne izvrši ni jedno ponavljanje. Razlog je jednostavan, provjera se vrši u posljednjem redu bloka kôda, te se mora proći barem jedanput kroz čitav blok. Za ovu vrstu petlje u većini programa se koristi ključna riječ "DO..WHILE" prilagođena sintaksi programskog jezika.

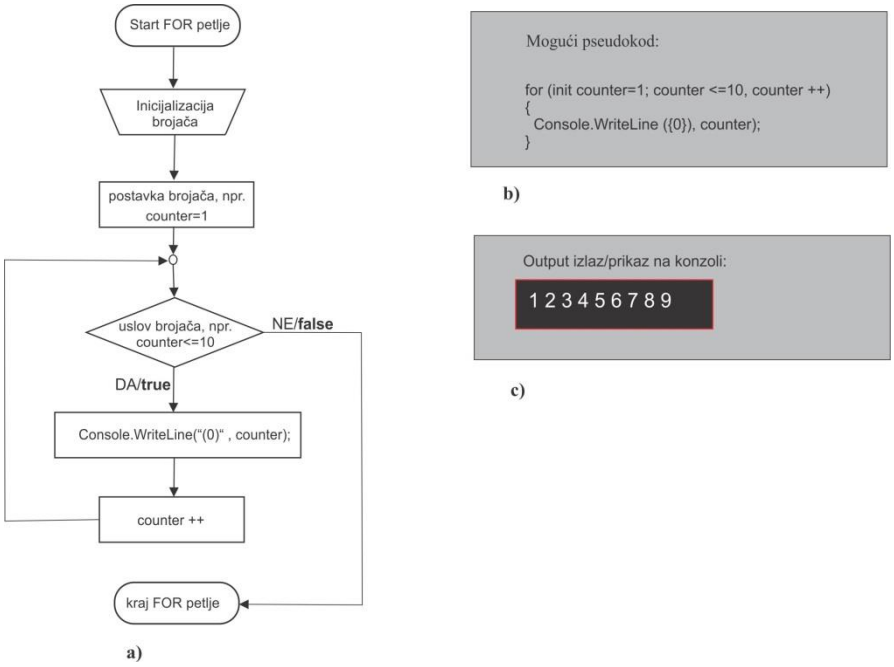
4.3.7 For/Next petlja

Kad je broj iteracija (ponavljanja) je unaprijed poznat obično se koriste FOR petlje omogućavaju da se grupa naredbi ponavlja unaprijed određeni broj puta. Predstavlja specijalni slučaj ciklične strukture. Unutar sebe ima ugrađen brojač koji inicijalnu vrijednost automatski mijenja (povećava: inkrementira, ili smanjuje: dekrementira). Obavljanje automatske promjene zavisi od sintakse jezika i naknadno ćemo ga obraditi na primjerima.

Opšti oblik pseudokoda FOR petlje je:

```
FOR (inicijalizacija; uslov nastavljanja; promjena vrijednosti)
      izraz
   end
```

Naredbe između **for** i **end** izvršavaju se jednom za svaki prolaz. Uslov nastavljanja mora biti logički izraz, dok inicijalizacija i promjena vrijednosti mogu biti bilo kakvi izrazi.



Slika 14 Primjer FOR/NEXT strukture

Primjer 8. Na Slici 14 a) data je jedna od mogućih varijanti predstavljanja FOR/NEXT strukture dijagramom toka, sa pripadajućim pseudokodom 14 b) i prikazom izlaza-rezultata na konzoli Slika 14 c).

Primjer 9

Primjer za ispis parnih brojeva izmedu 2 i 20:

```
FOR (N = 2; N <= 20; N = N + 2){
        System.out.println( N );
    }
```

4.4 Algoritam za sekvencijalno pretraživanje

Najjednostavniji primjer algoritma koji koristi listu je jednostavna iterativna struktura za sekvencijalno pretraživanje.

Cilj nam je pretražiti listu (pojmova, imena) u potrazi za određenim ciljem (target). Algoritam nam treba dati odgovor da li je traženi cilj na listi ili ne. Pri tom, pretpostavljamo da je lista složena nekim redom (npr. abecednim).
Logično je zadatak riješiti tako da članove liste, redom, poredimo sa vrijednosti cilja sve dok član sa liste ne postane jednak traženoj vrijednosti ili po abecednoj vrijednosti veći od ciljnog člana. U tom slučaju zaustavljamo pretraživanje i objavljujemo rezultat pretraživanja.

Za najednostavnji slučaj algoritam bi mogli predstaviti kodom:

```
Procedure Trazi(Lista, CiljnaVrijednost)
If (Lista prazna)
   Then  javi gresku (Rezultat pretrazivanje neispravno jer nema elemenata u listi)
     Else  (Odaberi prvu vrijednost u Listi kao TestVrijednost;

While (CiljnaVrijednost>TestVrijednost pa ima elemenata u Listi koje treba ispitati)
   Do (Odaberi slijedeću vrijednost sa Liste kao TestVrijednost.);
      If (CiljnaVrijednost=TestVrijednost)
         Then (Rezultat pretrazivanja pozitivan)
         Else (Rezultat pretrazivanja negativan)
 ) end if
```

Zbog jednostavnosti ovakav algoritam je primjeren pretraživanju malih listi. U ovom algoritmu iterativna struktura predstavljena je formom petlje kod koje se skup instrukcija (tijelo petlje) ponavlja u skladu sa postavljenim uslovima.
Razmotrimo utjecaj brzine pretrage na veću listu.
U takvim slučajevima bi koristili nešto drukčiji algoritam poznat kao binarna pretraga, metod koji koristi djeljenje liste u cilju ubrzanja rezultata. I ovdje bi koristili djeljenje liste i pivot metod. Evo mogućeg pseudokoda za jedan takav algoritam:

```
Procedure Trazi (lista, CiljnaVrijednost)
If (Lista prazna)
Then javi gresku (Rezultat pretrazivanje neispravno jer nema elemenata u listi)
Else
   Odaberi srednju vrijednost iz liste kao TestUlaz;
        Uporedi TestUlaz i CiljnuVrijednost, izvedi odgovarajuci skup instrukcija,
        zavisno o rezultatu poredjenja (case);
    Case 1: CiljnaVrijednost=TestUlaz     Javi rezultat da je ime na listi
    Case 2: CiljnaVrijednost<TestUlaz     Ponovi za imena  koja iznad TestUlaza
    Case 3: CiljnaVrijednost>TestUlaz     Ponovi za imena koja  ispod TestUlaza
) end if
```

4.5 Modularnost i podalgoritmi

Modularnost ćemo definisati kao programsku tehniku izrade softvera od više nezavisnih dijelova. Modularnost se primjenjuje kako bi se omogućila višestruka upotreba softverskih komponenti, odnosno tzv. **ponovna upotreba koda** (*code reuse*). Kompleksne probleme je teško „riješiti odjednom", a pogotovo to teško može uraditi jedan čovjek u kratkom vremenskom roku. Zbog toga se formiranje algoritma često obavlja na način da se on podijeli na više podalgoritama.
Svaki od podalgoritama odgovara složenoj proceduri ili potprogramu, i može se tretirati kao jedna cjelina, kao algoritam čiji početni uslovi su definisani u drugom (glavnom) algoritmu.
Struktura glavnog algoritma i podalgoritma treba da bude takva da je moguće jednostavno i jednoznačno uspostaviti vezu između matematičkih i logičkih modela koji im odgovaraju.
Između algoritma i podalgoritma postoji veza koja se ogleda u listi ulazno-izlaznih parametara. Lista izlaznih parametara podalgoritma mora da zadovoljava ulaznu listu algoritma i obrnuto.
Podalgoritam se može smatrati formom algoritma, pa i on nastaje kombinovanjem tri osnovne algoritamske strukture (linijske, razgranate i cikličke).
Modularni način programiranja podrazumijeva rastavljanje programa na dijelove (module) koji čine posebne autonomne cjeline. Podalgoritmi nam služe za razvoj (projektovanje) programskih modula.
Pošto algoritme želimo koristiti pri programiranju (kodovanju) daćemo primjer koji objašnjava princip modularnog programiranja.
Na primjer, kod rješavanja matematičkih zadataka napravićemo posebne module za različite matematičke operacije: sabiranje, oduzimanje, množenje i dijeljenje, te ih prilikom rješavanja zahtjevnijih zadataka pozivati po potrebi.

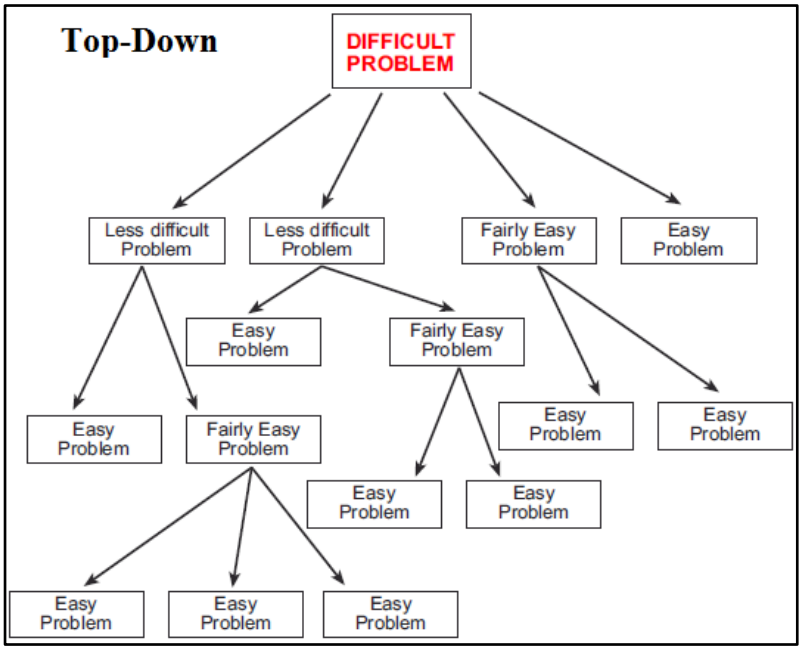
Moduli se mogu uključivati i isključivati od strane jednog ili više korisnika, a obzirom na mjesto gdje se nalaze dijelimo ih na:
- interne (nalaze se u programu)
- eksterne (nalaze se izvan programa i izvršavaju se pozivom)

Algoritmi su dizajnirani pomoću dva pristupa koji su top-down i bottom-up pristup.

U pristupu odozgo prema dolje, kompleksni modul je podijeljen na podmodule. S druge strane, pristup odozdo prema gore počinje s elementarnim modulima, a zatim ih dalje kombinira. Svrha algoritma je upravljanje podacima sadržanim u strukturi podataka. Drugim riječima, algoritam se koristi za izvođenje operacija na podacima unutar struktura podataka.

4.5.1 Top-Down postupak odozgo prema dolje

Projektovanje odozgo prema dolje je proces korak po korak koji počinje najopštijom funkcijom, razlaže tu funkciju u podfunkcije i ponavlja taj proces za svaku podfunkciju sve dok one ne budu dovoljno male i jednostavne da se mogu kodirati u stvarne programske instrukcije.



Ilustracija Top-Down postupka rješavanja problema

To je neformalna strategija projektovanja za **dekompoziciju** (rasčlanjivanje) kompleksnog problema na manje module, i time jasnije, djelove.
Ovaj pristup je pogodan za projektovanje manjih aplikativnih programa, jer je za velike projekte suviše neformalan.
Rješenju se približavamo postepeno, rješavanjem niza potproblema; zadatak rastavljamo na manje cjeline, od kojih je svaka lakše rješava od samog problema.
Top-Down je postupak, kod kojeg primijenjena metodologija ide od opšteg ka specifičnom: analiza odozgo-naniže.
Ponekad se ovakvi postupci mogu sresti i pod nazivom TDD (Top-Down-Design), ili čak reverzni inžinjering.
pristup odozgo prema dolje je prikladniji kada se softver treba dizajnirati od nule, a vrlo specifični detalji su nepoznati.
Razbijanjem većeg problema na manje fragmente, pristup odozgo prema dolje minimizira komplikacije koje obično nastaju prilikom dizajniranja algoritama. Nadalje, u ovom pristupu, svaka funkcija u kodu je jedinstvena i radi nezavisno od drugih funkcija.

4.5.2 Bottom-Up postupak odozdo prema gore

Predstavlja metodu suprotnu Top-Down postupku.
Metod dolaska do rješenja od dna do vrha. Kreće se od rješavanja najjednostavnijih problema ka sve složenijim, koji se posle "oslanjaju" na rješenja onih jednostavnijih.
Primjer bi bile slagalice (pazle), znamo finalnu formu i na osnovu komadića gradimo strukturu.
Analiza i projektovanje kod **Bottom-Up postupaka kreću od specifičnog ka opštem**.
U ovoj metodi, svaki modul se gradi i testira na individualnom nivou (testiranje jedinica) prije nego što se integrira u izgradnju konkretnog rješenja. Jedinično testiranje se izvodi korištenjem specifičnih funkcija niskog nivoa.

4.5.3 Ključne razlike između pristupa Top-Down i Botum-Up

1. Pristup odozgo prema dolje dekomponuje veliki zadatak na manje podzadatke, dok pristup odozdo prema gore prvo bira da direktno riješi različite fundamentalne dijelove zadatka, a zatim ih kombinuje u cijeli program.
2. Svaki podmodul se posebno obrađuje u pristupu odozgo prema dolje. Za razliku od toga, pristup odozdo prema gore implementira koncept skrivanja informacija ispitivanjem podataka koji se inkapsuliraju.
3. Različiti moduli u pristupu odozgo prema dolje ne zahtijevaju mnogo komunikacije. Naprotiv, pristup odozdo prema gore treba interakciju između odvojenih osnovnih modula da bi se kasnije kombinovali.
4. Pristup odozgo prema dolje može proizvesti redundantnost dok pristup odozdo prema gore ne uključuje suvišne informacije.
5. Proceduralni programski jezici kao što su Fortran, COBOL i C slijede pristup odozgo prema dolje. Nasuprot tome, **objektno orijentisani programski jezici kao što su C++, Java, C#, Perl, Python pridržavaju se pristupa odozdo prema gore.**
6. Pristup odozdo prema gore se prethodno koristi u testiranju. Suprotno tome, pristup odozgo prema dolje se koristi u dokumentaciji modula, kreiranju test slučajeva, otklanjanju grešaka, itd.

Kretanje odozdo prema gore je način da se izbjegne rekurzija, štedeći trošak memorije koji rekurzija ima kada izgradi stek poziva. Jednostavno rečeno, algoritam odozdo prema gore "počinje od početka", dok rekurzivni algoritam često "počinje od kraja i radi unazad".

4.6 Logičke igre, pitalice i algoritmi

Ukoliko algoritam shvatite kao neku strogu formu i u njegovom rješavanju ne pronađete ni malo užitka, Vi ćete možda da se bavite programiranjem, ali teško da ćete biti programer.
Logika i invencija, mogu da se uvježbaju (TEŠKO I DONEKLE) pa primjeri koji slijede, mogu da posluže kao test da li imate "štofa" za programera. Rješenja ne morate pronaći, ali ih trebate shvatiti. U protivnom razmislite o dizajnu, ili e-poslovanju, možda je to Vaš fah u informatičkoj branši.

4.6.1 Example1: Da li se trkate?

Problem
Prije trke A, B, C i D dali su svoje prognoze o rezultatu trke:
A je predvidio da će pobjediti B
B je predvidio da će D biti posljednji
C je predvidio da će A biti treći
D je predvidio da je A dobro procijenio.
Samo je jedna prognoza bila tačna i nju je predvidio pobjednik. Kako je završila trka?
Rješenje
A nije pobijedio jer je prognozirao da će pobjediti B
B nije pobjednik jer bi A imao tačnu prognozu
C je mogući pobjednik; A je u tom slučaju treći, B je posljednji (da bi prognoza B bila pogrešna), a D drugi, dakle rješenje je dobijeno **metodom eliminacije**: CDAB.
Komentar
Najjednostavniji metod je algoritam iscrpne pretrage. Rješenje je relativno jednostavno no ako od njega pokušate napraviti metodu i prevesti ga u univerzalni algoritam sigurno će Vam trebati vremena.

4.6.2 Example2: Svirate li klavir?

Problem
Osoba A treba pogoditi koliko imaju godina djeca osobe B.
Da bi joj olakšala posao, osoba B kaže osobi A da ima troje djece i da je proizvod njihovih godina 36.

Nakon razmišljanja, osoba A, inače jako dobar programer, kaže da joj je potrebna bar još jedna dodatna informacija.

Nakon toga joj osoba B uzvrati da može pretpostaviti da joj je poznat i zbir godina njene djece.

Kako je A rekla da joj na temelju dobijenih informacija niko ne može dati tačan odgovor, čak ni kad bi se tačno znalo koliko iznosi zbir njihovih godina.

Osoba B je tada rekla neka onda vodi računa o tome da joj najstariji sin svira klavir. I to je bilo dovoljno da se da odgovor.

Rješenje
Zadovoljna dobijenim informacijama, A izvijesti osobu B da zna kako ima dvogodišnje blizance i devetogodišnjeg sina koji svira klavir!?
Analiza
Kako je osoba A pronašla tačno rješenje, ako je druga informacija bila nepotpuna (nije rekla koliki je tačno zbir godina), a treća se čini besmislenom.

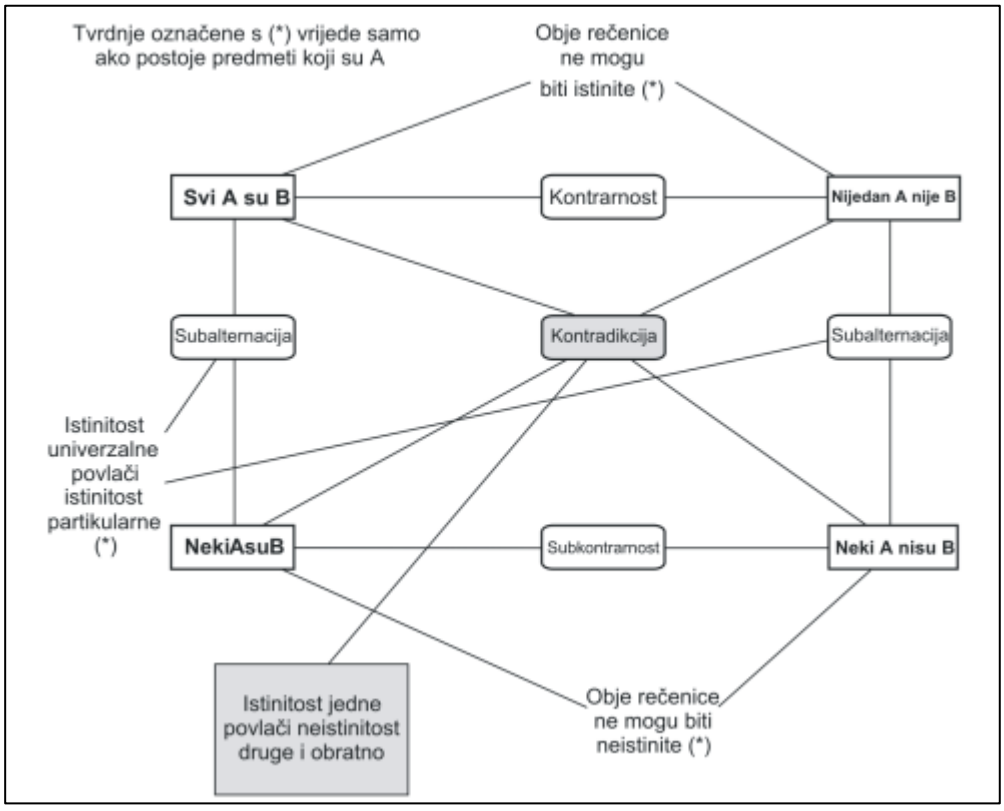
U trenutku kada je B rekla da je proizvod godina njene djece 36, A je napravila listu svih mogućih kombinacija:	Nakon pravljenja LISTE 1 uz pretpostavku da je poznata i suma njihovih godina lista bi poprimila slijedeći oblik:
LISTA 1	LISTA 2
(1 1 36)	(1 1 36 38)
(1 2 18)	(1 2 18 21)
(1 3 12)	(1 3 12 16)
(1 4 9)	(1 4 9 14)
(1 6 6)	(1 6 6 13)
(2 2 9)	(2 2 9 13)
(2 3 6)	(2 3 6 11)
(3 3 4)	(3 3 4 10)

Iz oblika liste, jasan je i smisao odgovora osobe A, jer dvije liste imaju isti iznos za sumu godina. Iako treća informacija zvuči sasvim besmisleno posmatramo li zadatak kao problem rješavanja tri jednačine sa tri nepoznate, informacija osobe B, rješenju konkretnog problema daje smisao, jer razdvaja liste sa istim odgovorima.
Ostaju nam kombinacije 1 6 6 i 2 2 9.
Pošto postoji samo jedna najveća godina (najstariji) to nam ostaje 2 2 9 kao rješenje.

Međutim ovo rješenje krije u sebi dvije netačnosti:
Prvo I blizanci nisu rođeni u baš isto vrijeme, pa je jedan stariji
Drugo: najstarije dijete ne mora imati veći broj godina, moglo se roditi jedno dijete, i onda 10 mjeseci kasnije drugo dijete. Tokom dva mjeseca u godini oni će imat isti broj godina.

4.6.3 Četiri Aristotelova oblika

Aristotel navodi četiri faktora koje treba uzeti u obzir kad se analizira bilo kakva promjena. Bez potrebe za filozofijom, ili matematikom[8], zadovoljite se čistom logikom[9]. Aristotel je proučavao četiri kvantifikatora: *svi, neki, nijedan* i *nisu svi*. Analizirajte te faktore u poznatom kvadrat suprotnosti:



Logički kvadrat

Proslijedite čuveni logički kvadrat i provjerite pitanje istinitosti rečenica koje se nalaze u uglovima uparenim međusobno po svim smjerovima.
Ovdje su prisutne tzv. razgovorna podrazumjevanja (implikature).
Izraz *Neke A su B* će se najčešće shvatiti da neke A jesu, a neke nisu B. Međutim taj izraz uopšte nema to značenje. Nigdje nije rečeno da B postoji, drugo šta ako je B veće od A, ili ako su sve A sadržane u B.
Tek s pojavom moderne logike predikata omogućen je pristup šemama zaključivanja koje obuhvaćaju sve moguće zaključke koji se odnose na četiri kvantifikatora koje je Aristotel proučavao.

[8] a to je u staro vrijeme bilo isto
[9] a to je i u staro i u ovo vrijeme bilo isto: filozofije i matematike nema bez logike, pa ni shvatanja algoritma bez apstrakcije

Umjesto opisivanja šta riječi poput *svaki* označavaju, standardni je postupak u modernoj logici prikazati jedinstvene istinosne uslove za *rečenice* koje počinju sa *svaki*. DAKLE nema univerzalnog rješenja (bez obzira na autoritet Aristotela).

Ovu temu završićemo sa čuvenim paradoksom kojim je Bernad Rasel početkom 20. vijeka pokazao da matematička logika vodi u kontradiktornosti:

U nekom selu su svi ljudi obrijani. Postoji berberin koji brije u selu sve ljude koji se ne briju sami, i nikoga više. Takvo selo ne može postojati jer se postavlja sljedeće pitanje: ***Ko onda brije berberina****?*
Ako se brije sam, onda se ne brije, ali ako se ne brije sam, onda se brije. Paradoks zapravo samo predstavlja dokaz da nema takvog berberina, ili drugim riječima, da je uslov nekonsistentan.
Ovo je najpoznatiji od paradoksa koji pokazuje kontradiktornost u osnovama teorije skupova. Neke klase imaju same sebe kao članove: klasa svih apstraktnih objekata je, na primjer, apstraktni objekat. Kada razmotrimo klasu svih klasa koje nemaju sebe kao članove dolazimo do pitanja: Da li je ta klasa svoj član? Ako jeste, onda nije, a ako nije, onda jeste.

4.6.4 Pisanje programa i lucida intervala

Ovi primjeri možda nisu karakteristični problemi koje ćete sretati pri programiranju. Oni potenciraju neke karakteristike sličnih problema koje je teško do kraja razumijeti i analizirati.
Univerzalan i sistematičan put do rješenja je rijedak.
Česta je priča da se do rješenja dolazi kad se najmanje očekuje, u šetnji, u snu i slično. U tim pričama je uvijek prisutan naporan i dugotrajan rad, gdje je bljesak ideje samo vrh ledenog brijega.
Većina programera prolazi kroz iskustvo koje se često opisuje kao trenutni blijesak u kome kome se pojavljuje rješenje nekog problema, koji se dugo činio nerješivim.
Psiholozi tumače pojavu kao djelovanje nesvjesnog, koje je tokom vremena savladalo problem i prebacilo rješenje u područje svjesnog. Naglasimo da ovo **tokom vremena** obično znači **dugotrajno**.

4.7 Link Primjeri izrade jednostavnih dijagrama toka

Ako ovaj priručnik koristite kao udžbenik, ili pomoćno nastavno sredstvo nudimo Vam mogućnost da na adresi:
https://sveznadar.info/diag/Algoritmi.htm
pronađete stotinjak primjera izrade dijagrama.

Primjeri i vježba su osnov razumijevanja algoritamskih struktura.

5 Klasifikacija i tipovi algoritama

Prema implementaciji algoritme dijeli na

- rekurzivne ili iterativne,
- logičke,
- serijske ili paralelne i
- determinističke ili stohastičke.

Rekurzivni algoritmi pozivaju sami sebe dok ne dođu do rješenja, odnosno, od većeg problema stvaraju manje i jednostavnije potprobleme i tako rješavaju glavni.
Nasuprot rekurzivnima, iterativni koriste petlje i nitove za rješavanje danih zadataka. Rekurzivni algoritmi mogu se konvertirati u iterativne i obrnuto, a ovisno o prirodi problema moguće je odabrati koji će se od dva načina rješavanja koristiti.
Svaki problem koji se može rešiti rekurzivno može se riješiti i iterativno, kao i obrnuto.

Logički algoritmi polaze od pretpostavke da se algoritam može promatrati kao kontrolisanu logičku dedukciju. Logička komponenta predstavlja činjenice koje se mogu koristiti u rješavanju problema, dok kontrola predstavlja način na koji se dedukcija primjenjuje na činjenice.

Serijski algoritmi nose takav naziv jer se pri obradi koristi jedan procesor (ranije su računari bili uglavnom takvi) pa se radnje izvršenja algoritma obavljaju jedna po jedna – serijski. S druge strane su paralelni koji koriste mogućnosti više procesora u računaru tako što probleme dijele na manje potprobleme od kojih svaki obrađuje jedan procesor.

Deterministički algoritmi rješavaju probleme unaprijed zadanim procesima dok stohastički do rješenja dolaze slučajnim izborom u svakom koraku procesa.

5.1 Klasifikacije algoritama prema metodologiji dizajna

Drugi način klasifikacije algoritama je **prema metodologiji dizajna**, gdje svaka kategorija sadrži dodatne različite tipove algoritama. Postoji više tipova algoritama. Daćemo kratak pregled 10 najvažnijih tipova algoritma prema ovoj klasifikaciji.
Poslije ove klasifikacije shvatićete da u osnovi klasifikacije nemaju baš previše smisla, jedan tip algoritma u sebe uključuje drugi.

5.1.1 Algoritam pretraživanja (Searching Algorithm)

Algoritam pretraživanja (**Searching Algorithm**): Algoritmi pretraživanja su oni koji 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 treba naći.
Najjednostavnije je za pretragu koristiti algoritam iscrpne pretrage.

5.1.2 Algoritam iscrpne pretrage (Brute Force)

Alogoritam iscrpne pretrege **(Brute Force)** algoritam, algoritam zasnovan na gruboj sili. Vrši se pretraga svih mogućih stanja koja se nalaze u sistemu. Tehnika rješavanja problema i algoritamska paradigma koja se sastoji od sistematskog nabrajanja svih mogućih kandidata za rješenje i provjere da li svaki kandidat zadovoljava iskaz problema. To je najjednostavniji pristup problemu.
Jednostavne metode rješavanja problema koje se oslanjaju na čistu računarsku snagu i isprobavanje svake mogućnosti umjesto naprednih tehnika za poboljšanje efikasnosti.
Klasičan primjer u informatici je problem trgovačkog putnika (TSP). Pretpostavimo da prodavac treba da poseti 10 gradova širom zemlje. Kako odrediti redoslijed kojim se ti gradovi trebaju posjetiti tako da se ukupna pređena udaljenost svede na minimum?
Rješenje korištenjm grube sile je jednostavno izračunati ukupnu udaljenost za svaku moguću rutu, a zatim odabrati najkraću. Ovo nije naročito efikasno jer je moguće eliminisati mnoge moguće rute pomoću pametnih algoritama.

Vremenska složenost grube sile je O(mn), što se ponekad piše kao O(n*m) . Dakle, ako bismo tražili niz od "n" znakova u nizu od "m" znakova koristeći grubu silu, trebalo bi nam n * m pokušaja.

U želji da se brute-force pretraga primjeni na određenu klasu problema, moraju se provesti četiri funckije, prvo-***first***, sljedeće-***next***, validno-***valid***, i izlaz-***output***.
Ova procedura treba da uzme kao parametar P u konkretnom slučaju problema koje treba rješiti, a trebalo bi da uradi sljedeće:

1. first (P): generiše rješenje kandidata za prvo P.
2. next (P, c): generisati sljedeći kandidat za P nakon trenutne c.
3. valid (P, c): provjerite da li kandidat c je rešenje za P.
4. output (P, c): koristiti rešenje c od P kao pogodno za primjenu

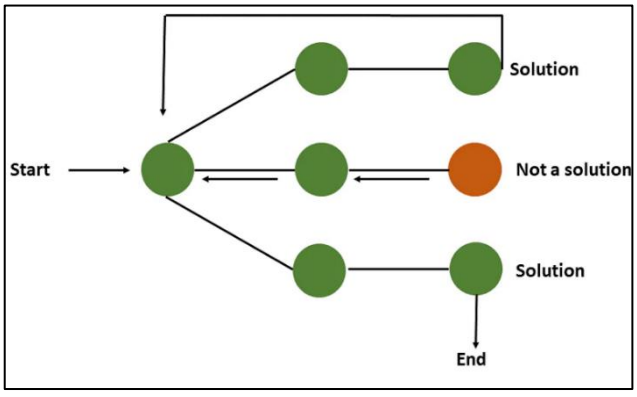
Postupak mora reći kada više ne postoje kandidati za primjer P, nakon sadašnjeg c. Pogodan način za to je da se vrati na "nultu kandidata", neki konvencionalni podatak vrijednost i Λ da se razlikuje od stvarnog kandidata. Isto tako prvo treba da vrati Λ ako uopšte nema kandidata za primjer P.

Brute-force metoda se može opisati psudokodom na ilustraciji ispod:

```
c ← first(P)
while' c ≠ Λ do
 if' valid(P,c) then output(P, c)
 c ← next(P,c)
end while
```

5.1.4 Algoritam vraćanja unazad (Backtracking Algorithm)

Algoritam vraćanja unazad (**Backtracking Algorithm**): Algoritam vraćanja unatrag u osnovi gradi rješenje pretragom među svim mogućim rješenjima i u osnovi je jedna od varijanti iscrpne pretrage.



Koristeći ovaj algoritam nastavljamo graditi rješenje prema kriterijima. Kad god rješenje ne uspije, vraćamo se do tačke prekida-kvara i gradimo na sljedećem rješenju i nastavljamo ovaj proces dok ne pronađemo rješenje, ili dok se ne potraže sva moguća rješenja (rješenje je i da nema rješenja).

U principu algoritam vraćanja unazada rješava tri vrste problema:

1. Problem odlučivanja (Decision Problem) – tražimo izvodljivo rješenje.
2. Problem optimizacije (Optimization Problem)– tražimo najbolje rješenje.
3. Problem nabrajanja (Enumeration Problem) – nalazimo sva izvodljiva rješenja.

5.1.5 Algoritam sortiranja (Sorting Algorithm)

Algoritam sortiranja (**Sorting Algorithm**): Sortiranje je sređivanje grupe podataka na određeni način prema zahtjevu. Algoritmi koji pomažu u obavljanju ove funkcije nazivaju se algoritmi sortiranja. Generalno, algoritmi za sortiranje se koriste za sortiranje grupa podataka na rastući ili opadajući način.
Niz sortiramo tako što tražimo najvećeg u nizu, postavljamo ga na prvo mjesto u isti takav, prazan niz. Zatim tražimo sljedećeg po veličini i tako redom dok ne dođemo do kraja.
Prostorna složenost ovog algoritma je **O(2n)** jer je potreban još jedan niz isti kao i originalan. Vremenska složenost ovog algoritma je **$O(n^2)$** jer se algoritam odvija u potpunom dvostrukom ciklusu.

Primjeri (sortiranje nizova od n članova):

- **Metoda zamjene**

Niz sortiramo tako što uspoređujemo susjedne članove niza. Pri tome vršimo zamjenu članova niza ako je prvi veći. Prolazeći tako kroz niz, najveći član se postavlja na posljednje mjesto. U sljedećem prolazu ponavljamo postupak, ali sada ne moramo ići do kraja niza jer je posljednji član na svom mjestu. I tako dok ne ostanu dva člana.
Prostorna složenost ovog algoritma je **O(n)** jer nije potreban dodatan niz kao u prethodnom slučaju.
Vremenska složenost ovog algoritma je **$O((n-1)^2/2)$**. I dalje je potreban dvostruki ciklus ali za jedan prolaz manje od broja elemenata u nizu, a u unutrašnjem ciklusu se broj obrada smanjuje od n-1 do (prosječno n/2)
Metoda se može malo poboljšati ako provjeravamo da li je bilo zamjene u pojedinom prolazu. Ako nije, nema potrebe dalje raditi jer to znači da je niz sortiran. Ovo ne utiče na složenost algoritma jer je to specifičan slučaj.

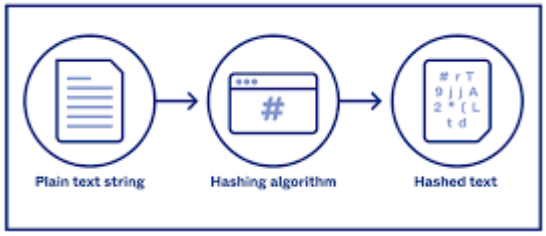
- **Quick sort**

Poenta ove metode je da niz podijelimo u dva podniza, lijevi i desni od nekog izabranog elementa u nizu. Pri tome se u lijevom nizu nalaze elementi koji su manji od izabranog a u desnom oni koji su veći od izabranog. Sada svaki od podnizova nastavljamo obrađivati na potpuno isti način. Kako smo problem sveli na jednostavniji, a dalji postupak je isti, postupak odrađujemo re-kurzijom.
Prostorna složenost ovog algoritma zavisi od verzije (dijeljenje u mjestu ili ne-što drugo), načina izbora pivota i rasporeda članova niza. Za verziju gdje se niz dijeli ***u mjestu*** najgori slučaj je **O(n logn)** dok je u najboljem slučaju **$O(\log^2 n)$**.
Vremenska složenost ovog algoritma je **O(n logn)**. Mada je za najgori slučaj, kada je podjela nizova na 1 član i n-1 član pri svakom dijeljenju, kada je vremenska složenost **$O(n^2)$**.
Kada su nizovi sa velikim brojem članova, a vrijeme nije kritična stvar u izvršavanju, tada Quick sort daje najbolje rezultate od svih algoritama sa složenošću **O(n logn).**

5.1.6 Hashing algoritam

Hashing algoritam: Hashing algoritmi rade slično algoritmu pretraživanja. Ali oni sadrže indeks sa ID ključem. Kod heširanja, ključ se dodjeljuje određenim podacima.

Heš funkcije se prvenstveno koriste za generisanje fiksne dužine izlaznih podataka koji se ponašaju kao reference na originalne podatke. Jedna praktična primjena je struktura podataka koja se zove heš tabela u kojoj su podaci smešteni asocijativno. Linearna pretraga imena osobe u listi traje duže kako se dužina liste povećava, ali heširana vrijednost može biti skladištiti referencu na originalni podatak, pa tako dobijamo konstantno vrijeme za pretragu.



Primjena haš algoritma na kriptovanje

Postoje desetine različitih algoritama za heširanje i svi rade malo drugačije. Ali u svaki od njih se upisuju podaci, a program ih mijenja u drugačiji oblik.
Često se koriste gotove funkcije pogotovu ako želimo da kriptujemo podatke.

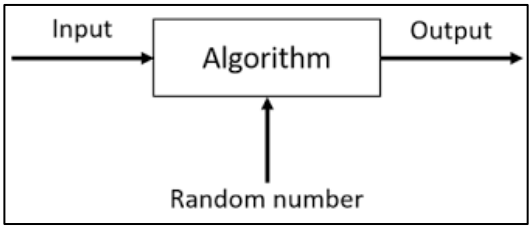
5.1.7 Pohlepan algoritam (Greedy Algorithm)

Pohlepan algoritam (**Greedy Algorithm**): U ovom tipu algoritma rješenje se gradi dio po dio. Rješenje sljedećeg dijela je izgrađeno na osnovu neposredne koristi od sljedećeg dijela. Jedno rješenje koje daje najveću korist biće izabrano kao rješenje za sljedeći dio. Ovaj algoritam slijedi heuristiku rješavanja problema donošenja lokalno optimalnog izbora u svakoj fazi. Ovaj algoritam bira najbolje izvodljivo rješenje u tom trenutku bez obzira na posljedice. Pohlepna metoda kaže da bi problem trebalo riješiti u fazama u kojima se smatra da je svaki unos izvediv.
Pohlepni algoritam najbolje je primijeniti kada je potrebno rješenje u stvarnom vremenu, a približni odgovori su "dovoljno dobri".
Pohlepni algoritmi su uglavnom jednostavnog oblika. Koriste se većinom za rješavanje problema optimizacije, kao na primjer nalaženja minimalnog razapinjućeg stabla grafa, nalaženja najkraćeg puta u grafu te nalaženja najboljeg redoslijeda izvođenja zadanih poslova.
Pohlepni algoritam nigdje ne koristi funkciju cilja, već samo funkciju izbora. Da bi rješenje nađeno na ovakav način bilo i optimalno rješenje, potrebno je dokazati da pohlepni algoritam korektno rješava problem optimizacije.

Za mnoge probleme pohlepni algoritam ne uspije proizvesti optimalno rješenje, također, događa se da proizvede jedinstveno najgore moguće rješenje, ali su značajno brži od drugih algoritama.

5.1.8 Slučajni/Nasumični algoritam (Randomized Algorithm)

Randomizirani algoritam (**Randomized Algorithm**): U randomiziranom algoritmu koristimo slučajan broj tako da daje trenutnu korist.
Nasumični algoritam je algoritam koji koristi određeni stepen slučajnosti kao dio svoje logike ili procedure.



Slučajni broj pomaže u odlučivanju o očekivanom ishodu.
Na primjer, u nasumičnom brzom sortiranju (Randomized Quick Sort) koristimo nasumični broj za izbor sljedećeg pivota (ili nasumično miješamo niz). Obično se ova slučajnost koristi za smanjenje vremenske složenosti ili kompleksnosti prostora u drugim standardnim algoritmima.

5.1.9 Rekurzivni algoritam (Recursive Algorithm)

Rekurzivni algoritam (**Recursive Algorithm**): Rekurzivni algoritam je zasnovan na rekurziji.
Rekurzija je metoda rješavanja računskog problema gdje rješenje zavisi o rješenjima manjih instanci istog problema. Rekurzija rješava takve rekurzivne probleme korištenjem funkcija koje se same pozivaju iz vlastitog koda.
Problem se dijeli na nekoliko poddijelova i poziva istu funkciju iznova i iznova sve dok ne pronađe zadovoljavajuće rješenje.
Generalno, ako se problem može riješiti korištenjem rješenja manjih verzija istog problema, a manje verzije se svode na lako rješive slučajeve, tada se može koristiti rekurzivni algoritam za rješavanje tog problema.
Do rješenja se dolazi u postupku “od opšteg ka pojedinačnom”:
Problem se najpre podijeli u više manjih potproblema
Zatim se nezavisno rješava svaki od tih potproblema
Kombinovanjem njihovih rješenja se dolazi do rješenja polaznog problema

Programi bazirani na rekurzivnim algoritmima zahtevaju više memorije i računanja u poređenju sa iterativnim algoritmima, ali su jednostavniji i u mnogim slučajevima prirodan način razmišljanja o problemu.

Ideja je rekurzivnih funkcija svaki uzastopni rekurzivni poziv dovesti bliže slučaju koji je lako rješiv.
Slučaj koji je lako rješiv bazni je slučaj koji mora imati svaka rekurzivna funkcija te još mora imati i opšti-generalni slučaj.
Dakle, u rekurzivnim funkcijama nema iteracija, već se ponavljanje može postići tako da funkcija poziva samu sebe dok ne dođe do baznog slučaja koji omogućuje završetak ponavljanja. Rekurzivna funkcija mora imati završetak, a kada nema, nerješiva je jer se izvodi beskonačno.

Rekurzivna funkcija je funkcija koja poziva samu sebe tokom izvršavanja. To omogućava da se funkcija ponavlja više puta, vraćajući rezultat na kraju svakog ponavljanja.

Daćemo jedan primjer rekurzivne funkcije.
Primjer Funkcija koja vrši stepenovanje realnog broja *x* prirodnim eksponentom *n*.

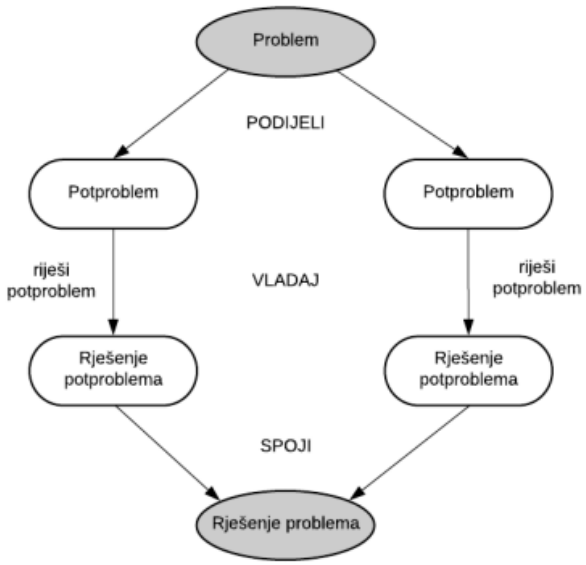
```
function xnan (x:real; n:integer):real;
begin
   if n=1 then
      xnan:=x
   else
      xnan:=xnan(x,n-1)*x;
end;
```

Rekurzivne funkcije su uobičajene u računarstvu zato što dozvoljavaju programerima da pišu efikasne programe koristeći minimalnu količinu koda. Nedostatak je taj da one mogu uzrokovati beskonačne petlje i druge neočekivane rezultate ako nisu pažljivo napisane. Ako odgovarajući slučajevi nisu uključeni u funkciju da zaustave izvršavanje, rekurzija će se ponavljati zauvijek, uzrokujući da program padne, ili što je još gore, da obori čitav računarski sistem.

5.1.10 Algoritam Podijeli i vladaj (Divide and Conquer)

Algoritam Podijeli i vladaj (**Divide and Conquer Algorithm**): Ovaj algoritam razbija problem na podprobleme, rješava jedan podproblem i spaja rješenja kako bi se dobilo konačno rješenje.
Sastoji se od sljedeća tri koraka:

- Podijelite
- Riješiti
- Kombinujete



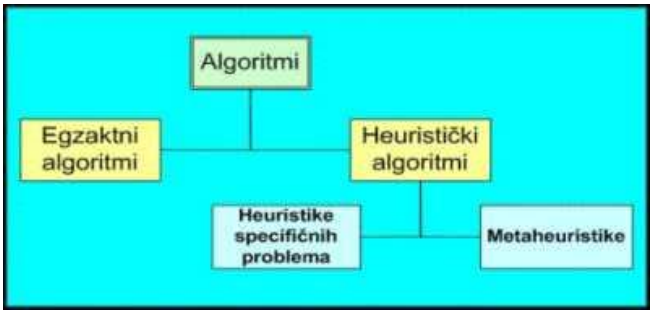
Algoritam zavadi pa vladaj rekurzivno razlaže problem na dva ili više podproblema istog ili srodnog tipa, sve dok ovi ne postanu dovoljno jednostavni da se mogu direktno riješiti.

Standardni algoritmi koji koriste ovu metdologiju su:

- Binarno pretraživanje je algoritam za pretraživanje. U svakom koraku, algoritam upoređuje ulazni element x sa vrijednošću srednjeg elementa u nizu. Ako se vrijednosti podudaraju, vratite indeks sredine. Inače, ako je x manji od srednjeg elementa, tada se algoritam ponavlja za lijevu stranu srednjeg elementa, inače se ponavlja za desnu stranu srednjeg elementa.
- Quicksort je algoritam za sortiranje. Algoritam bira stožerni element, preuređuje elemente niza na takav način da se svi elementi manji od izabranog pivot elementa pomiču na lijevu stranu pivota, a svi veći elementi na desnu. Konačno, algoritam rekurzivno sortira podnise lijevo i desno od pivot elementa.
- Sortiranje spajanjem je takođe algoritam za sortiranje. Algoritam dijeli niz na dvije polovine, rekurzivno ih sortira i na kraju spaja dvije sortirane polovine.

5.2 Heuristički algoritmi

Kao posbna klasa mogu se izdvojiti algoritmi zasnovani na heuristici. Oni bi obuhvatali randomizirane algoritme i različite metode pretrage.
Heuristički se sakupljaju iskustva (engl. experience-based learning) i stvara se (unutar varijable vremena) **zdravorazumsko rješenje: common sense**.
Heurističko mišljenje se ponekad opisuje frazom "umjetnost saznanja (ili otkrivanja)". Često se spominje i u kontekstu poboljšanja rješenja problema,odnosno lakšeg nalaženja rješenja.
Glavna heuristička novost sistemskog pristupa je svođenje sistema (odn. redukcija) na dinamiku, za razliku od dekontruktivističkog pristupa u klasičnom smislu (analitičkom) pristupu.
Problem, odnosno sistem se rješava heurističkom redukcijom (sastavnica, elemenata ili entiteta), koristeći dinamiku koja daje različite prioritete i važnost pojedinim članovima.
Najbliže heurističkom je intuitivno mišljenje, dok bi analitička analiza po (zadanom) obrascu više odgovrala algoritmu. Međutim korištenje heuristike može da se obavlja po propisanim metodama i tada govorimo o metaheuristici.
Heuristički algoritmi su algoritmi nastali eksperimentisanjem u svrhu dobijanja zadovoljavajućeg rješenja. Bitno svojstvo heurističkih algoritama je da mogu približno (**dovoljno dobro**) rješiti probleme eksponencijalne složenosti.



Heurističke algoritme možemo podjeliti na algoritme koji rješavaju heuristike specifičnih problema i metaheuristike.

Heuristike specifičnih problema (**Problem specific heuristics**) su heuristički algoritmi namijenjeni za rješavanje tačno određenog problema. U takve heuristike ubrajamo i funkcije procjene, tj. funkcije heuristike. Funkcije heuristike su sastavni dijelovi svih metaheurističkih algoritama.

Metaheuristički algoritmi obuhvataju širok skup algoritama namijenjenih optimizacijskim problemima. Oni su se pokazali izuzetno dobri u rješavanju NP-teških problema: problema nerješivih egzaktnim algoritmima u stvarnom vremenu.

Glavna snaga heurističkih algoritama je da smanjuju prostor pretrage koristeći neke saznanja, te time znatno ubrzavaju proces pronalaženja rešenja. No, to znači i da heuristički algoritmi moraju uvijek dati optimalno rješenje, a ponekad njihovo izvođenje može trajati duže od algoritama koji koriste egzaktne metode rešavanja.

Dva su osnovna uslova koje algoritam mora zadovoljiti ako želimo da pronađemo globalni optimum:

- uslov stabilnosti (stabilizaciju u globalnom optimumu)
- uslov oslobađanja iz lokalnog optimuma

Heuristički metod postiže taj zadatak koristeći stabla pretrage u prostoru stanja. Međutim, umjesto da generiše sve grane, heuristika bira grane koje će vjerovatnije doći do rješenja od ostalih. Selektivna je u svakoj tački odluke, birajući grane koje će prije dovesti do rješenja. Svaka slejdeća iteracija zavisi od prethodnog koraka i na taj način heuristička pretraga uči koje puteve da koristi a koje da odbacuje mjereći koliko je blizu je trenutna iteracija rješenju. Samim tim neke mogućnosti se neće generisati jer će biti izmjereno da je manje moguće da dođu do rješenja.

5.2.1 Kombinatorička eksplozija heuristika i vještačka inteligencija

Važna klasa problema na koje se intenzivno primjenjuju metode heuristike i umjetne inteligencije su problemi opterećeni tzv."**kombinatoričkom eksplozijom**".

To su problemi za koje je poznat egzaktni matematički algoritam, ponekad je čak i jednostavan, ali bi njegova primjena, zahtijevala neostvarivo mnogo vremena.

Primjer kombinatoričke eksplozije je igranje šaha. Nije posebno teško napisati na nekom programskom jeziku algoritam koji bi pretraživanjem svih varijanata do kraja partije egzaktno odredio najbolji potez, ali realizacija takvog algoritma nije moguća u stvarnosti, zbog golemog broja varijanata koje bi trebalo istražiti, koje mnogostruko nadmašuju mogućnosti bilo kojeg računara.
Šah po broju podataka i mogućih varijanata ne predstavlja (teoretski) veliki problem postoji samo 64 polja i 32 figure.
Problemi koji se pojavljuju u drugim djelatnostima, npr. u automatskom projektovanju, često su veći za mnogo redova veličine. U literaturi se navode primjeri koji po formulaciji izgledaju prilično bezazleno, ali bi za njihovo egzaktno rješavanje na nekom budućem računaru mnogo djelotvornijem od današnjih trebalo mnogostruko više vremena od sadašnje starosti svemira. Naravno da se mora odustati rješavanja takvog problema pretraživanjem svih mogućnosti.
Ako opet uzmemo primjer iz šaha, program će odrediti potez koji samo slučajno može biti egzaktno najbolji, ali je najbolji koji se može odrediti na raspoloživom kompjteru odabranim programom u raspoloživom vremenu. *Ni šahovski velemajstor ne može garantovati da je njegov potez apsolutno najbolji – osim u slučajevima kad je rješenje jednostavno – na primjer kad je moguć forsirani matni napad ili pat te često u završnici kad je jako reduciran broj figura na ploči.*

5.3 Grafovski algoritmi

U matematici i računarstvu teorija grafova[10] se bavi proučavanjem grafova, koji su matematičke strukture za modeliranje parova odnosa između objekata.

Graf je vrsta strukture podataka, tačnije apstraktan tip podataka, koji se sastoji od skupa čvorova i skupa grana, koje predstavljaju odnose (veze) između čvorova. Graf kao struktura podataka direktno potiče od matematičkog koncepta grafa.
Graf je uređena trojka G= (V;E;F) gdje je F funkcija koja svakoj grani E pridružuje 2-člani skup čvorova V.

Mnoge se pojave algoritamski modeliraju grafovima koji se sastoje od čvorova i grana (veza među čvorovima). U praksi, za svaki čvor i granu vezujemo neke podatke sa kojima želimo da manipulišemo.
Na primjer, čvorovi mogu predstavljati ljude iz neke skupine, a grane (edge) parove prijatelja.

U računarstvu se često dijagram toka nekog algoritma prikazuje grafom kojem su čvorovi naredbe (instrukcije), a lukovi iz jedne u drugu naredbu su grane.
Grafovski algoritmi su od velikog značaja u računarstvu. Tipične operacije povezane sa grafovima su **nalaženje puta** između dva čvora, za šta se na primer koriste pretraga grafa u dubinu i pretraga grafa u širinu, i **nalaženje najkraćeg puta** od jednog do drugog čvora.

Proučavanje algoritama koji rješavaju probleme upotrebom grafova predstavlja značajan dio informatičke nauke.

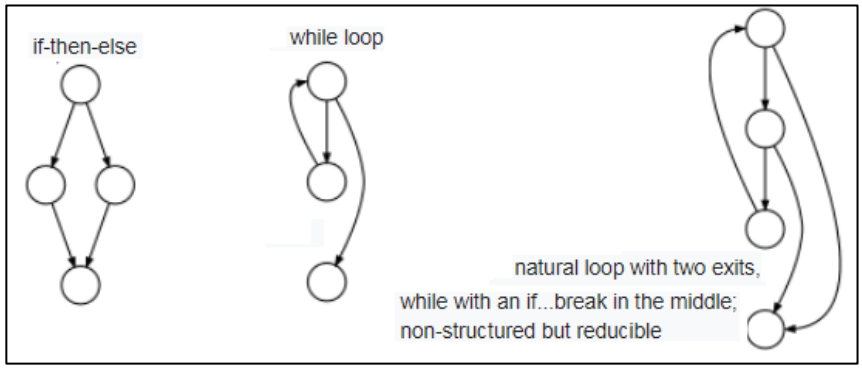
Ako želite nastavili sa analizom heurističkih algoritama potrebno je da se upoznate sa teorijom grafova i proučite kako se obavlja pretraga u prostoru stanja.

Jednako tako grafovima se prezentuju i razne računarske strukture podataka, umrežavanje i paralelizam računara i njihov sekvencijalni rad, evolucijska ili porodična stabla u biologiji itd.
Graf može predstavljati električnu mrežu čiji su vrhovi električke komponente, a grane električne veze. Putne- cestovne, željezničke, avio veze itd. daljnji su primjeri modela s grafovima.

[10] Grafovi su jedan od primarnih predmeta studija u diskretnoj matematici i ovdje će se pokušati samo krajnje pojednostavljeno uspostaviti veza između dijagrama toka kao jednostavne predstave algoritma i grafa toka kao kompleksne matematičke predstave

5.3.1 Graf kontrole podataka

Graf kontrole podataka (Control-flow graph) poznat i kao graf toka programa je kvazi-digraf[11] čiji čvorovi odgovaraju naredbama, njihovim dijelovima ili pak grupama.
Graf toka je struktura slična blok dijagramu algoritma (organigramu) koja pokazuje redosljed izvršavanja njegovih naredbi. Takođe, ulazne i izlazne grane grafa toka programa obično se vezuju za samo jedan čvor, tj. ulazna grana ne izlazi ni iz jednog čvora niti izlazna grana ulazi u neki čvor. Kaže se da one ulaze iz okoline odnosno izlaze u nju.
Za razliku od blok dijagrama algoritma, graf toka programa nema čvorova tipa START i STOP, a takođe se ne pravi razlika između operacija obrade i raznih operacija ulaza-izlaza.



Ilustracije izgleda GKP koji predstavljaju programske sekvence nekog programa

Kod grafa kontrole podataka svaki čvor grafa predstavlja bazični blok, tj. dio koda bez ikakvih skokova ili ciljeva skokova; ciljevi skokova predstavljaju početke blokova a skokovi predstavljaju krajeve blokova.
Usmjerene ivice (grane) se koriste za reprezentaciju skokova u kontroli toka.
Kod većine reprezentacija, postoje dva specijalna bloka: ulazni blok, kroz koji kontrola ulazi u graf, i izlazni blok, kroz koji kontrole napuštaju graf.
Zbog načina kreiranja, u GKP-u, svaka ivica A→B ima svojstvo da:
izlazni stepen(A) > 1 ili izlazni stepen(B) > 1 (ili oba).

GKP se dakle može dobiti, bar konceptualno, od punog grafa nekog programa (tj. grafa kod koga svaki čvor predstavlja individualnu instrukciju) primjenom redukcije ivice za svaku ivicu za koju ne važi prethodno tvrđenje tj. redukcija svake ivice čiji početak ima tačno jedan izlaz i čiji kraj ima tačno jedan ulaz.
Ovakav algoritam baziran na redukciji **nema** praktičnih primena, osim kao vizuelno pomagalo za razumjevanje GKP konstrukcije, jer se GKP može mnogo efikasnije konstruisati direktno iz programa skenirajući bazične blokove.

[11] Orijentisani graf koji opisuje neku strukturu podataka,

6 Algoritmi strukturno i objektno programiranje

Strukturirano programiranje je jedan od načina pokušaj da se u postupak razvoja programa uvede red.
Za osnov strukturiranog programiranja smatra se **strukturna teorema** koju su 1966 prezentovali Boehm-a i Jacopinija 1966. godine. 1968. Dajkstra (E.W. Dijkstra) objavio čuvenu raspravu „Bezuslovni skok je štetan“ "Goto Statement Considered Harmful", koja se smatra prekretnicom i začetkom modernog koncepta (strukturnog) programiranja. U raspravi je kao glavni „krivac“ neupravljivosti složenih softverskih paketa naveo bezuslovne programske skokove koji su korišteni u to doba glavnim programskim jezicima, fortranu i kobolu.
Metode i tehnike koje čine strukturirano programiranje intenzivno su se razvijale i u narednih desetak do petnaest godina. U poslednjih desetak godina (praktično u ovom mileniju) strukturirano programiranje je izgubilo primat i kao opšteprihvaćena programska paradigma se koriste jezici bazirani na objektno orijentisanom programiranju. Međutim, **algoritamske upravljačke strukture koje se koriste u objektnom programiranju (sekvence, selekcije i ciklusi) su iste one koje su definisane u strukturiranom** pristupu. Pod strukturiranim programiranjem podrazumeva se skup tehnika za razvoj programskih modula koje koriste strogo definisane grupe upravljačkih struktura i struktura podataka.

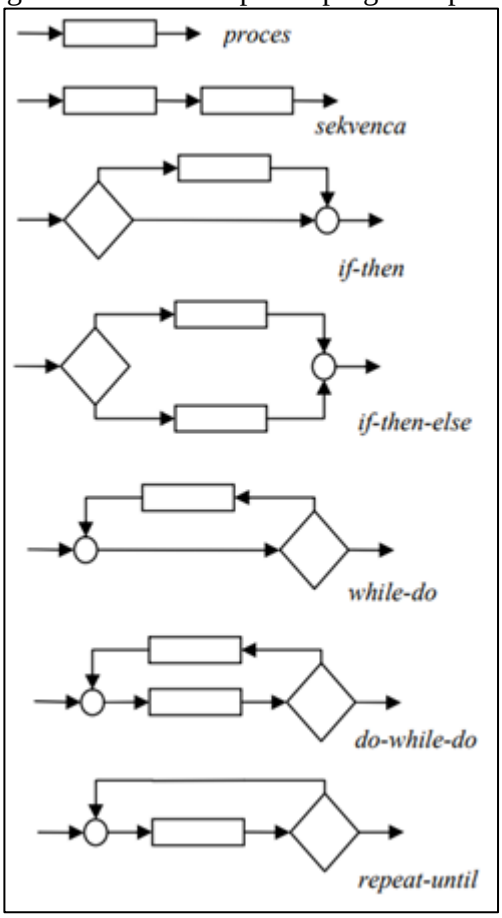
6.1 Proceduralno programiranje pravilni i prosti programi

Proceduralno programiranje je programska paradigma, izvedena iz imperativnog programiranja, zasnovana na konceptu poziva procedure. Procedure sadrže niz računskih koraka koje treba izvršiti.
Teorija procedurnog programiranja zasnovana je na pojmu pravilnog programa.
Pravilan program je program čiji graf toka zadovoljava sljedeća tri uslova:
1. Postoji tačno jedna ulazna grana.
2. Postoji tačno jedna izlazna grana.
3. Kroz svaki čvor prolazi najmanje jedan put od ulazne do izlazne grane (ovaj uslov sprečava postojanje beskonačnih ciklusa i izolovanih grupa čvorova).

Pravilni programi nisu neka posebna klasa programa. Naprotiv oni programi koji ne zadovoljavaju da budu pravilan program predstavljaju posebne, najčešće besmislene, slučajeve.
Prost program je program kojem odgovara graf toka takav da nijedan od njegovih pravilnih potprograma nema više nego jedan čvor (a radi se o procesu, što je lako zaključiti). Prosti programi su od ključne važnosti za strukturiranu analizu jer imaju tu osobinu da se ne mogu dekomponovati na dijelove koji su za sebe pravilni programi, sa izuzetkom čvorova-procesa. Značaj prostih programa leži u činjenici da su upravljačke strukture (naredbe) procedurnih programskih jezika uglavnom prosti programi.

6.2 Baza strukturiranih programa

Skup prostih programa čijom se superpozicijom može realizovati bilo koji pravilan program naziva se baza strukturiranih programa.
Baze strukturiranih programa svode se na proste programe prikazane na slici.

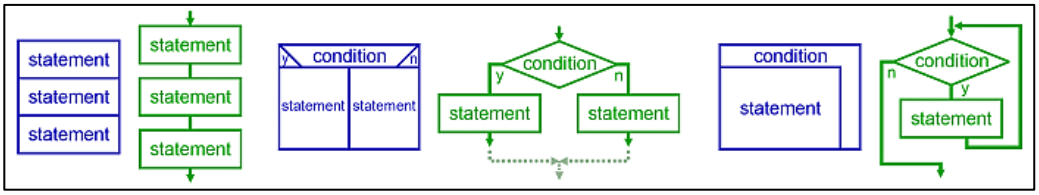


Prosti programi koji čine bazu strukturnih programa

Sa ovim strukturama smo se već upoznali. Primjetimo da nazivi osnovnih prostih programa potiču iz paskalske terminologije.
Ovi osnovni prosti programi, u formi naredbi, sreću se u svim programskim jezicima nastalim posle paskala.
Kada (ako) postoje dodatne upravljačke strukture u strukturiranim programskim jezicima one su uvedene iz praktičnih razloga, da bi programski kôd bio kraći i razumljiviji.

6.3 Strukturna teorema

Suština strukturne teoreme je u tome da se svaki pravilan program može realizovati superpozicijom tri prosta programa:
Svaki pravilan program može se transformisati u ekvivalentan, formalno strukturiran program uz korišćenje tri osnovne upravljačke strukture: sekvence, selekcije i ciklusa.



Grafički prikaz tri osnovne strukture uzorka teoreme strukturiranog programa

Strukturna teorema ima bar dve važne posledice:
- za programsku realizaciju pravilnih programa nisu potrebne naredbe skok
- osnovni prosti programi tipa sekvence, selekcije if-then-else i ciklusa while-do čine bazu strukturiranih programa.

Pošto sada raspolažemo bar jednom bazom, moguće je ekvivalentnim transformacijama osnovnih prostih programa izgraditi i druge baze.

Ako se ima u vidu da se prost program
if(p) A else B
može transformisati u ekvivalentnu sekvencu oblika
{
if(p) A
if(!p) B
}
koja sadrži samo sekvencu i if-then sledi zaključak da:
Prosti programi tipa sekvence, selekcije *if-then* i ciklusa *while-do* takođe čine bazu strukutiranih programa.
Na sličan način može se ciklus *while-do* zameniti u bazi ciklusom *repeat-until* itd.

Prema ovome iz strukturne teoreme se može zaključiti:

- svaki algoritam može se prikazati na programskom jeziku čije upravljačke strukture sadrže bar jednu bazu strukturiranih programa, i
- svaki algoritam se može prikazati na programskom jeziku koji sadrži bar sekvencu i ciklus while-do (alternativno: sekvencu i if-then).

7 U potrazi za definicijom pojma algoritma

Šta je algoritam? Odgovor na ovo fundamentalno pitanje nije samo od teorijskog interesa. Problem razumijevanja pojma algoritma je veoma važan za *specifikaciju*, *validaciju* i *verifikaciju* softverskih i hardverskih sistema.

Dušan T. Malbaški - *Algoritmi i strukture podataka*:
Jedan od najosnovnijih pojmova u računarstvu jeste pojam algoritma, do te mere fundamentalan da bi, možda, najbolji odgovor na pitanje "šta je to?" bio "**zna se**!". Zato ne treba da čudi to što neki veoma istaknuti teoretičari, kao što je Uspenski, smatraju da algoritme ni ne treba definisati dovođenjem u vezu sa drugim jezičkim simbolima, nego ih valja smatrati polaznim, i da je svaki takav pokušaj definisanja, u stvari, samo opisivanje. Drugi pak stoje na stanovištu da algoritam ipak treba definisati, ali samo u logičkom, a ne u formalno-matematičkom smislu. U svakom slučaju, sam pojam algoritma je toliko bazičan da stoji tvrdnja da je "*najveće otkriće u oblasti nauke o algoritmima otkriće samog algoritma*".

7.1 Definicija algoritma po Kolmogorovu

Problem apsolutne definicije algoritma je 1953. god. razmatrao sovjetski matematičar Andrej Kolmogorov. Svoje intuitivne ideje o algoritmima je izrazio na sljedeći način:

- Algoritamski proces se dijeli na korake čija je kompleksnost unaprijed ograničena, tj. granice su nezavisne od ulaza i trenutnog stanja izračunavanja.
- Svaki korak se sastoji od direktne i neposredne transformacije trenutnog stanja.
- Ova transformacija se primjenjuje samo na aktivni dio stanja i ne mijenja ostali dio stanja.
- Veličina aktivnog dijela je unaprijed ograničena.
- Proces traje sve dok ili sljedeći korak ne bude nemoguć ili dok signal kaže da je rješenje dostignuto.

Kao dodatak ovim intuitivnim idejama, Kolmogorov je sa svojim studentom Vladimirom Uspenskim dao nacrt novog modela računanja. Model mašine Kolmogorova može se posmatrati kao generalizacija modela Tjuringove mašine gdje je traka usmjereni graf. Čvorovi grafa odgovaraju Tjuringovim kvadratima; svaki čvor ima boju izabranu iz fiksne konačne palete boja čvorova; jedan od čvorova je trenutni centar izračunavanja. Svaka grana ima boju izabranu iz fiksne konačne palete boja grana; različite grane iz istog čvora imaju različite boje. Nasuprot Tjuringovoj traci čija topologija je fiksna, Kolmogorovljeva “traka“ je rekonfigurabilna.

7.2 Parcijalna i totalna korektnost algoritma

U teorijskom računarstvu, za jedan algoritam se tvrdi da je k**orektan** kada se kaže da je korektan u odnosu na *specifikaciju.*
U računarstvu, **formalne specifikacije** su matematički zasnovane tehnike čija je svrha da pomognu pri implementaciji sistema i softvera. One se koriste za opisivanje sistema, za analizu njegovog ponašanja i kao pomoć pri njegovom dizajniranju tako što *verifikuju* ključne osobine sistema koje su od interesa. To se radi pomoću rigoroznih i efikasnih alata za rasuđivanje.Ove specifikacije su *formalne* u smislu da imaju sintaksu, njihova semantika se nalazi unutar jednog domena i mogu se koristiti da bi se došlo do korisnih informacija.
U ovom kontekstu, **formalna verifikacija** označava čin dokazivanja ili opovrgavanja korektnosti algoritama koji podržavaju neki sistem, a u vezi sa određenom *formalnom specifikacijom* ili osobinom, koristeći formalne metode matematike.
Formalna verifikacija može biti od pomoći u dokazivanju korektnosti hardverskih sistema kao što su napr. kombinaciona kola ili digitalna kola sa internom memorijom, ali takođe i softvera izraženog u njegovom izvornom kodu.
Verifikacija ovih sistema se radi davanjem formalnog dokaza na apstraktnom matematičkom modelu sistema, pri čemu je odnos između matematičkog modela i prirode sistema poznat iz konstrukcije sistema. Primjeri matematičkih objekata koji se često koriste za modelovanje sistema su: mašine konačnih stanja, vektorski adicioni sistemi, hibridni automati, procesna algebra, formalna semantika programskih jezika kao što je operaciona semantika, aksiomatska semantika i Hoareova logika.
Funkcionalna korektnost odnosi se na ulazno-izlazno ponašanje algoritma (tj. za svaki ulaz algoritam proizvodi očekivani izlaz).
Postoji razlika između **totalne korektnosti**, gdje se dodatno zahtijeva da se algoritam završava (terminira), i **parcijalne korektnosti**, gdje se jednostavno traži da *ako* je jedan odgovor vraćen on bude korektan.
Hoareova logika je specifični formalni sistem za rigorozno rasuđivanje o korektnosti računarskih programa. Ona koristi aksiomatske tehnike za definisanje semantike programskih jezika i diskutovanje o korektnosti programa.

7.2 Vremenska i prostorna složenost (kompleksnost) algoritma

U teoriji složenosti se proučava problematika složenosti (kompleksnosti) algoritma, u smislu zauzimanja resursa, a to su prostor (količina zauzete memorije) i vrijeme (količina potrošenog procesorskog vremena). Složenost je funkcija veličine ulaznih podataka. Algoritmi se prave za rješenje opšteg problema, bez obzira na veličinu ulaza, ali sa druge strane razne ulazne veličine uzrokuju da programi troše razne količine resursa.

Vremenska složenost algoritma se iskazuje kao broj elementarnih koraka za obavljanje algoritma, što zavisi od veličine ulaza, a koja može biti izražena napr.u bitovima. Ako kažemo da je algoritam (A) uređivanja niza od n elemenata problem vremenske složenosti n^2, znači da dvostruko veći broj elemenata zahtijeva četiri puta više vremena za uređivanje. Ako je, pak drugi algoritam (B) malo pažljivije napisan i brži je dvostruko, on će raditi dvostruko brže za bilo koju veličinu niza. Međutim, ako se programer potrudi i osmisli suštinski drugačiji algoritam (C) za uređivanje, on stvarno može biti reda složenosti $n \cdot log(n)$. Algoritmi (A) i (B) su iste složenosti, jer se u tzv. **notaciji sa velikim O** obilježavaju sa $O(n^2)$, a u govoru se zovu „algoritmi kvadratne složenosti“, dok je algoritam (C) „algoritam složenosti $n \cdot log(n)$“. Zaključak: algoritam (C) je najbolji sa stanovišta korišćenja vremena za velike skupove ulaznih podataka.

Prostorna složenost se odnosi na funkciju zavisnosti zauzimanja memorijskog prostora u zavisnosti od veličine ulaza. Dešava se da je pronalaženje algoritma manje vremenske složenosti povezano sa povećanjem prostorne složenosti. Obično se algoritmi dijele na one logaritamske složenosti, linearne, kvadratne i uopšteno polinomijalne složenosti, kao i one najzahtjevnije, eksponencijalne složenosti.

7.3 Pojam algoritamskog sistema

Pojam algoritma može shvatiti kao niz pravila na osnovu kojih se vrši proces transformacije ulaznih podataka u traženi rezultat, gdje se proces transformacije obavlja se mehanički.

Da bi se proces obavio potreban je “nekakav” izvršioc i taj izvršioc koji može *mehanički, samo na osnovu algoritma*, a bez ikakvog uticaja korisnika kome je rezultat potreban, bez ikakve kreativnosti, da produkuje traženi rezultat.

Očigledno da je potrebna nekakav dogovoreni skup pravila koji omogućava komunikaciju: jezik, kojim se algoritam zadaje, a koji može da razumije i izvrši izvršilac.

Taj jezik, nema neku unapred zadatu, formu. Algoritam se može zadati u vidu uređenog skupa naredbi-rečenica (npr. pseudokod), ali i grafički u formi blok dijagrama, ili kao sekvenca iskaza na formalnom jeziku (posljednjih godina to može biti i govorni), programskom jeziku, pa i kao kolekcija crteža. Bilo kako, ali tako da mehanički izvršilac bude u stanju da, na osnovu opisa i samo na osnovu njega, proizvedi rezultat.

Ovdje se javlja bar dva seta problema:

- Koji su dopušteni ulazni podaci i šta su zadovoljavajući traženi rezultati
- Koji su elementi jezika kojima se obavlja transformacija i kako se definiše alfabet jezika kojim se obavljaju transformacije.

Nema (zasad) univerzalnog odgovora kako da se realizuje algoritam, odnosno tj. kako izgleda postupak transformacije ulaznih podatak u izlazne i kakav je to jezik koji to omogućava.
Ponuđeno je nekoliko algoritamskih sistema koji definišu postupak transformacije ulazne riječi u izlaznu.
Najčešće korišteni formalizmi su:

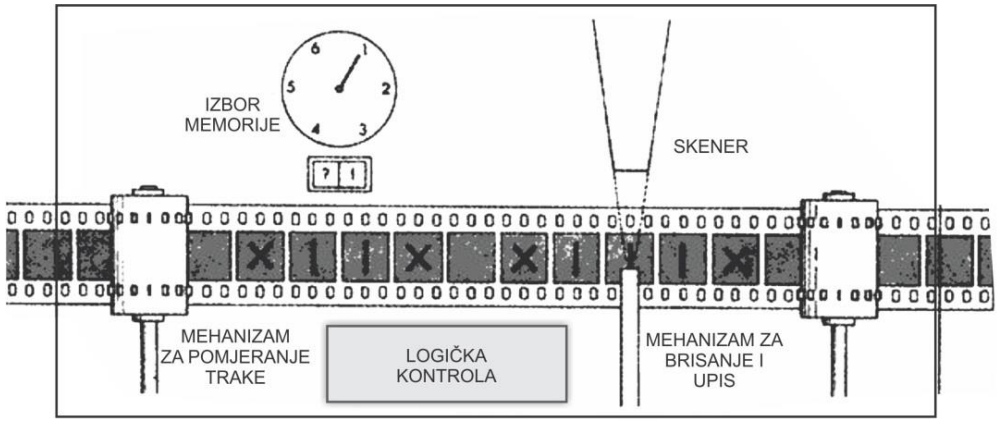
- Tjuringova mašina i
- Normalni algoritmi Markova

7.3.1 Tjuringova mašina

Tjuringova mašina[12] (nazvana po britanskom matematičaru Alanu Tjuringu; (*Alan Mathison Turing* 1912.-1954.) je *apstraktna mašina* koja manipuliše simbolima na traci prema nekoj tablici pravila; da budemo precizniji, radi se o *matematičkom modelu računanja* koji definiše takav uređaj. Uprkos jednostavnosti modela, za svaki dati računarski algoritam je moguće konstruisati Tjuringovu mašinu koja je u stanju da simulira logiku tog algoritma.

Tjuringova mašina je **izvršilac algoritma**. Algoritamski sistem pridržavajući se propisanih pravila-kodeksa, a na osnovu ulaznog i izlaznog alfabeta realizuje alfabetski operator, tj. daje traženi izlaz, i to postupajući *mehanički*.
Mašina radi na beskonačnoj memorijskoj traci podijeljenoj na *ćelije*. Mašina pozicionira svoju *glavu* iznad ćelije i "čita" (skenira) simbol koji se tu nalazi. Zatim prema simbolu i njegovom trenutnom mjestu u *konačnoj tabeli* korisnički specificiranih instrukcija mašina (i) upisuje simbol (tj. cifru ili slovo iz konačnog alfabeta) u ćeliju (neki modeli dozvoljavaju brisanje simbola i/ili neupisivanje), onda (ii) pomjera traku za jednu ćeliju lijevo ili desno (neki modeli ne dozvoljavaju pomjeranje, neki modeli pomjeraju glavu), potom (iii) (kako je određeno posmatranim simbolom i mjestom u tabeli mašine) ili nastavlja sa narednom instrukcijom ili zaustavlja računanje.

[12] Postoji niz varanti Tjuringove mašine od kojih se svaka može da se matematički opiše na više načina. Postoje dva osnovna tipa: tzv. specijalnu Tjuringovu mašinu (sa jednom trakom) i univerzalna Tjuringova mašina. Ovjde ćemo (dijelom) prfedstaviti Specijalnu Tjuringovu mašinu.



Slika 16 Model Tjuringove mašine
obrađeno prema http://stackoverflow.com/questions/236000/whats-a-turing-machine

Turingova mašina je uređena sedmorka (Q, S, T, b, q_0, F, δ), pri čemu je:

- *S konačan skup znakova s kojima mašina radi (abeceda trake),*
- *T konačan skup znakova koji se mogu naći na traci prije početka rada mašine (ulazna abeceda); $T \subset S$,*
- *$b \in S \setminus T$ tzv. "prazan" simbol; oznaka da je polje prazno,*
- *$q_0 \in Q$ početno stanje rada mašine,*
- *$F \subseteq Q$ skup završnih stanja i*
- *$\delta : Q \times S \rightarrow Q \times S \times \{S, L, D\}$, gdje S označava "stani", L "lijevo", a D "desno".*

Mašina ima beskonačnu traku (što bi bio ekvivalent za hard disk) i glavu koja se kreće po toj traci za jedno mjesto ulijevo ili udesno, te čita podatke s trake i zapisuje nove. Stanja predstavljaju radnu memoriju i ima ih proizvoljno, ali konačno mnogo. Njih određujemo prilikom sastavljanja mašine, dok se po traci možemo kretati koliko god želimo.

Na početku rada mašine na traci se nalazi zapis opisan ulaznom abecedom koji je analogon ulaznim podacima programa.

Turingova mašina nalazi se u stanju *q* na nekoj poziciji na traci odakle je pročitao znak *s*. zavisno od ta dva parametra mašina prelazi u novo stanje *q′*, zapisuje novi znak *s′* na traku na poziciju na kojoj se nalazi, te pomiče glavu za jedno mjesto ulijevo ili udesno ili ostane stajati na istom mjestu, tj. obavi pomak iz skupa $\{S, L, D\}$.

Jedan korak Turingove mašine može se predstaviti petorkom (q, s, q', s', m), pri čemu $q, q' \in Q, s, s' \in S, m \in \{S, L, D\}$.

Proces rada Turingove mašine u potpunosti je odreden konačnim skupom petorki opisanog oblika.

Definisano je da mašina staje s radom kada dođe u završno stanje i napravi pomak "*S* ", tj. kad funkcija δ izvrši preslikavanje $(qi, x) \rightarrow (qp, y, S)$, pri čemu $x, y \in S$, $qi \in Q$, $qp \in F$.

Tjuring je, inače, izvorno ovaj model nazvao *a-mašina* (automatska mašina). Sa ovim modelom on je bio u stanju da odgovori negativno na dva pitanja:

1. Da li može da postoji mašina koja može da odredi da li je bilo koja proizvoljna mašina na svojoj traci "cirkularna" (tj. zaustavlja se, ili ne može da nastavi svoj zadatak računanja); slično,
2. Da li može da postoji mašina koja može da odredi da li bilo koja proizvoljna mašina na svojoj traci ikada štampa dati simbol.

Tako je Tjuring dajući matematički opis vrlo jednostavnog uređaja sposobnog za proizvoljna izračunavanja, bio u stanju da dokaže osobine računanja uopšte – i posebno, *neizračunljivost* tzv. Hilbertovog "problema odlučivanja".
Generalno problem (ne)odlučivosti sastoji se u traženju odgovora na pitanje:
Postoji li efektivna metoda (algoritam) kojom bi se za danu matematičku tvrdnju moglo ustanoviti je li istinita ili nije u svim logičkim strukturama datog aksiomatskog sistema?
Problem zaustavljanja, poznat kao Halting problem[13] je problem odluke koji se neformalno može iskazati na sljedeći način:
Za dani opis programa i konačnog ulaza, odluči može li program da se završi ili se izvršuje unedogled, za dani ulaz.
Radi se o pitanju „da li se može unapred zaključiti hoće li proizvoljna Tjuringova mašina za zadatu ulaznu reč završiti rad“, tj. hoće li upravljački blok doći u neko od stanja iz skupa F?
Problem odluke je skup prirodnih brojeva - "problem" je odlučiti je li istaknuti broj element skupa.
Odgovor na pitanje je negativan!

Halting problem spada u klasu neodlučivih problema.

U *teoriji izračunljivosti*[14], problem zaustavljanja je problem određivanja, iz opisa proizvoljnog računarskog programa i njegovog ulaza, da li će se taj program zaustaviti ili će nastaviti da se izvršava u beskraj.

Alan Tjuring je dokazao da opšti algoritam koji bi riješio halting problem za *svaki* mogući par program-ulaz, ne može postojati.

[13] Možda je zanimljivo: Često se kaže da je Turing iskazao i dokazao teorem o zaustavljanju u 'On Computable Numbers', ali ovo nije strogo istnito U nijednom od svojih radova Turing nije koristio riječ "zaustavljanje" ili "terminacija". Problem je zaustavljanja tako imenovan od strane Martina Davisa. Formalno Tjuring nije ni koristio, pa ni rješio problem zaustavljanja, ali je rješenje proizašlo iz njegove mašine.
[14] Teorija izračunljivosti se bavi modelima i oblicima izračunavanja, posebno izračunavanjima uz pomoć apstraktnih matematičkih mašina, analizom algoritama i problemima kompleksnosti algoritama, te formalnim gramatikama i formalnim jezicima.

Ključni dio njegovog dokaza bila je matematička definicija računara i programa, koja je postala poznata kao **Tjuringova mašina**; halting problem je *nerješiv* na Tjuringovim mašinama. To je bio jedan od prvih primjera *problema rješivosti*.

Ideja Turingovog dokaza kreće od osnovnih elemenata:
ulaz, program (skup pravila koji djeluju na ulaz i vraćaju izlaz) i izlaz. Turing je pretpostavio da je mogućce napisati program HALT koji, za svaki zadani program i pripadajući ulaz, može odrediti da li će se izvodenje tog programa zaustaviti ili ne:

$$\text{HALT(program, ulaz)} = \begin{cases} \text{izlaz DA}, & \text{ako se program zaustavi} \\ \text{izlaz NE}, & \text{inače} \end{cases}.$$

Ako takav program HALT postoji, može se napisati novi program OPPOSITE koji analizira programe koji za ulaz imaju sami sebe, , tj. koji se zaustavlja ako i samo ako se zadani program ne zaustavlja:

$$\text{OPPOSITE(program)} = \begin{cases} \text{zaustavlja se}, & \text{ako HALT(program, program) vrati NE} \\ \text{beskonačna petlja}, & \text{inače} \end{cases}$$

Šta se događa ako se program OPPOSITE pokuša pokrenuti na samom sebi?
Ako se zaustavi, onda mora upasti u beskonačnu petlju, a ako upadne u beskonačnu petlju, onda se mora zaustaviti (*sjećate li se onog brice*?)
Iz ove kontradikcije slijedi da program HALT ne može postojati, dakle problem zaustavljanja je neodlučiv.

Pošto ne postoji opšte rješenje problema zaustavljanja, potvrda totalne korektnosti mora ležati znatno "dublje". Radi se o tzv. *dokazu završavanja* (terminacionom dokazu) koji je tip matematičkog dokaza, a koji igra ključnu ulogu u *formalnoj verifikaciji* pošto totalna korektnost algoritma zavisi od terminacije.

Na primjer, za sukcesivno pretraživanje kroz cijele brojeve 1, 2, 3, … da bismo vidjeli hoćemo li naići na primjer nekog fenomena — recimo na neparan savršen broj — dosta je lako napisati parcijalno korektan program (koristeći dugo dijeljenje sa dva da bismo provjerili da li je *n* savršen ili nije). Ali da bismo rekli da je ovaj program totalno korektan trebalo bi potvrditi nešto što trenutno nije poznato u teoriji brojeva.
Dokaz bi trebao da bude matematički dokaz, uz pretpostavku da su i algoritam i specifikacija dati formalno.
Posebno se ne očekuje da bude potvrde korektnosti za dati program koji implementira algoritam na datoj mašini. To bi uključivalo takva razmatranja kao što su ograničenja računarske memorije.

Smatrajući da Turingova mašina ispunjava tačno one uslove za koje se smatra da ih intuitivni pojam algoritma treba imati, Church je iznio tezu[15] da se svaki algoritam može realizovati nekom Turingovom mašinom. Teza poznata je pod Churchova teza (ili Turing-Churchova) i do danas nije opovrgnuta i opšte je prihvaćena.
Teza tvrdi da je bilo koji izračun koji je uopšte moguć, moguće napraviti algoritmom koji se može izvršiti na računaru, uz dostatne vremenske i prostorne resurse. Teza ne može biti matematički dokazana, te se stoga ponekad predlaže kao fizikalni zakon ili kao definicija.
Neformalno, Church-Turingova teza iskazuje da se intuitivna predstva algoritma može precizirati, te da računari mogu izvršavati te algoritme. Nadalje, računar teoretski može izvršavati bilo koji algoritam. Drugim riječima, svi uobičajeni računari su međusobno ekvivalentni u terminima teoretske računske moći, i stoga nije moguće izgraditi uređaj za računanje koji će biti moćniji od najjednostavnijeg računara (Turingove mašine). Valja uočiti da ova formulacija moći zanemaruje praktične faktore kao što su brzina ili memorijski kapacitet - razmatra se sve što je teoretski moguće, uz dano neograničeno vrijeme i memoriju.

Digitalni računar se na apstraktnom nivou se obično prikazuje kao sistem sastavljen od procesora, memorije i ulazno-izlaznih uređaja. Procesor iz memorije pribavlja naredbe i podatke nad kojima se vrši obrada u skladu sa značenjem naredbi, a dobijeni rezultati se vraćaju u memoriju. Preko U/I uređaja podaci koji će biti obrađeni se unose u memoriju, odnosno iz memorije se preuzimaju rezultati obrade i prikazuju na odgovarajući način. Komunikacija dijelova računara se obavlja preko magistrala.

Tjuringova mašina je preteča ovakvog modela računara, pri čemu su neka svojstva idealizovana: memorija - potencijalno beskonačna u svakom koraku izračunavanja Tjuringove mašine zauzet je samo konačan broj memorijskih registara, **ne postoji ograničenje koliki je taj konačan broj registara** svakom koraku izračunavanja moguće je i zahtjevati novi, do tada neiskorišteni memorijski registar i svaki takav zahtjev će biti ispunjen.

Tjuringova kompletnost je sposobnost nekog sistema instrukcija da simulira Tjuringovu mašinu.
Programski jezik koji je Tjuring-kompletan je teorijski u stanju da izrazi sve zadatke koji se mogu izvršavati na računarima; gotovo svi programski jezici koji se danas koriste su Tjuring-kompletni.

[15] Vremenska tačnost ove tvrdnje nije baš korektna. Prvo je Čerč (u Americi) iznio teoretske postavke, Tjuring, bez da je bio upoznat sa njima, (u Engleskoj) je gotovo paralelno prezentovao koncept svoje mašine, pa se upoznao sa Čerčovim radom, iznio neke primjedbe, Čerč proširio svoju analizu...
Tezu je prvi predložio Stephen C. Kleene 1943., ali je nazvana po Churchu i Turingu.

7.3.2 Normalni algoritmi Markova

U teorijskom računarstvu, **Markovljev (ili normalni) algoritam** je sistem za ***ponovno pisanje stringova*** (*String Rewriting System*, skraćeno **SRS)** koji koristi pravila slična gramatičkim, da operiše nad stringovima simbola.
U matematici, računarstvu i logici, pojam **rewriting** pokriva širok opseg potencijalno nedeterminističkih) metoda zamjenjivanja pod-članova neke formule sa drugim članovima.
Ostali nazivi su **rewrite uređaji (**engl. *engines*) ili **redukcioni sistemi**. U svojoj najosnovnijoj formi, oni se sastoje od skupa objekata, plus relacije pomoću kojih se transformišu ti objekti. Za Markovljeve algoritme je dokazano da su ekvivalentni Tjuringovim mašinama, što znači da su pogodni kao opšti model računanja i mogu predstavljati bilo koji matematički izraz pomoću svoje jednostavne notacije. Markovljevi algoritmi su dobili ime po sovjetskom matematičaru Andreju Markovu.

Prvi korak na putu ka kanoničkom obliku algoritma je standardizacija alfabetskog operatora G.
Markov je uočio da se transformacija ulazne riječi iz skupa X* u riječ iz Y*, a posredstvom G, može realizovati u etapama primjenom samo dvije vrste elementarnih operatora:
1. operatora obrade čiji je zadatak da tekući oblik riječi preradi u sljedeći koji vodi (ili ne vodi) konačnom rješenju i
2. upravljačkih operatora što, na osnovu osobina tekuće reči, odlučuju o tome koji će operator obrade biti odabran za narednu transformaciju.

Pod normalnim algoritmom Markova podrazumjeva **se strogo uređeni niz operatora smjene i odluka o njihovoj primjeni**.
Suština definicije je u sintagmi "strogo uređeni niz" jer različiti nizovi smjena mogu da vode ka različitim rezultatima.
Ako se na riječ xyzyx primjene redom smjene x→a, x→b dobija se riječ ayzyb;
ako se redosljed smjena promeni u x →b , x →a rezultat je byzya.

Značaj normalnih algoritama je njihova univerzalnost koja proističe iz tzv. principa normalizacije koji glasi:
Za svaki algoritam može se konstruisati funkcionalno ekvivalentan normalni algoritam koji je nedokaziv zbog neodređenosti pojma "svaki algoritam".

Dakle svaki algoritam može se predstaviti kao normalni algoritam Markova iz čega neposredno slijedi da su Tjuringove mašine i normalni algoritmi Markova međusobno ekvivalentni načini za zadavanje istog pojma - algoritma.
Treba napomenuti da za ovo tvrđenje postoji i formalni dokaz.
Postoji i programski jezik koji se naziva **Refal** i koji je zasnovan na Markovljevim algoritmima.

7.3.3 λ – račun

λ – račun mogli bismo nazvati i *najmanjim univerzalnim programskim jezikom*.

λ – račun se sastoji od jednog transformacionog pravila (supstitucije varijabli) i jedne sheme za definiciju funkcija. Uveo ga je tridesetih godina dvadesetog vijeka američki matematičar Alonzo Čerč (*Alonzo Church*; 1903.-1995.) kao način formalizovanja koncepta efektivne izračunljivosti. λ – račun je univerzalan u smislu da se bilo koja izračunljiva funkcija može izraziti i procijeniti pomoću njegovog formalizma.

Međutim, λ – račun naglašava upotrebu transformacionih pravila i ne uzima u obzir koja ih stvarna mašina implementira. To je pristup koji je više povezan sa softverom nego sa hardverom.

Centralni koncept u λ – računu je "izraz". "Ime", takođe nazvano i "varijabla", je identifikator koji, za ovu svrhu, može biti bilo koje slovo a, b, c, . . .
Izraz je definisan rekurzivno na sljedeći način:

<izraz> : = <ime> | <funkcija> | <aplikacija>
<funkcija> : = λ <ime> . <izraz>
<aplikacija> : = <izraz> <izraz>

Izraz može biti u zagradama zbog jasnoće, to jest, ako je E izraz, (E) je isti izraz. Jedine ključne riječi koje se koriste u jeziku su λ i tačka. Da bi se izbjeglo nagomilavanje izraza sa zagradama, usvojena je konvencija da aplikacija funkcije asocira s lijeva, to jest, izraz

$E_1E_2E_3 \ldots E_n$
se evaluira na sljedeći način:
(. . . ((E1E2)E3) . . . En)

Kao što se može vidjeti iz definicije gornjih λ izraza, jedan identifikator je λ izraz. Primjer jedne funkcije izgleda ovako:

$$\lambda x.x$$

Ovaj izraz definiše funkciju identiteta. Ime nakon λ je identifikator argumenta te funkcije. Izraz nakon tačke (u ovom slučaju jedno *x*) se naziva "tijelo" definicije.
Funkcije se mogu primijeniti na izraze. Primjer jedne primjene (aplikacije) je

$$(\lambda x.x)\ y$$

Ovo je funkcija identiteta primijenjena na y. Zagrade se koriste zbog jasnoće i izbjegavanja dvosmislenosti. Primjena funkcije se vrši supstituisanjem vrijednosti argumenta *x* (u ovom slučaju *y*) u tijelu definicije funkcije, tj.

$$(\lambda x.x)\ y = [y/x](x) = y$$

U ovoj transformaciji notacija [*y/x*] se koristi da naznači da su sva pojavljivanja *x* supstituisana sa *y* u izrazu na desnoj strani.
Imena argumenata u definicijama funkcija nemaju sama po sebi nikakvo značenje. Ona su samo "čuvari mjesta" (engl. place holders), to jest, ona se koriste da naznače kako treba preraspodijeliti argumente funkcije kada se ona evaluira. Dakle

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

i tako dalje. Koristimo simbol "≡ " da naznačimo da kada je $A \equiv B$, A je samo sinonim za B.

8 Nove algoritamske paradigme i pravci u računarstvu

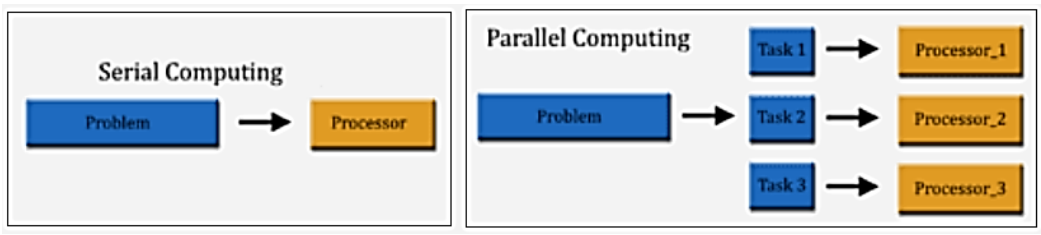
8.1 Paralelizam i konkurentnost

Termin **paralelizam** odnosi se na tehnike koje čine program bržim tako što se nekoliko izračunavanja izvršava paralelno. Ovo zahtijeva hardver sa više procesnih jedinica. U mnogim slučajevima pod-izračunavanja imaju istu strukturu, ali to nije neophodno. Grafička izračunavanja na GPU su paralelizam.
Ključni problem paralelizma je redukovanje zavisnosti po podacima da bi se mogla izvoditi izračunavanja na nezavisnim jedinicama sa minimalnom komunikacijom između njih. U tom cilju mogla bi čak biti i prednost izvršavati isto računanje dvaput na različitim jedinicama.

Termin **konkurentnost** (istovremenost – concurrency) odnosi se na tehnike koje čine program upotrebljivijim. Konkurentnost može biti implementirana i mnogo se koristi na jednoj procesnoj jedinici, mada može imati koristi od više procesnih jedinica što se tiče brzine. Ako se neki operativni sistem naziva multi-tasking operativni sistem, to je sinonim za podržavanje konkurentnosti. Ako možemo otvoriti više dokumenata istovremeno u tabovima našeg internet pretraživača, otvarati menije i izvoditi još više operacija, to je konkurentnost.
Ako pokrenemo distribuirana mrežna izračunavanja u pozadini dok radimo sa interaktivnim aplikacijama, to je konkurentnost. Sa druge strane, dijeljenje zadatka na pakete koji se mogu obrađivati preko distribuiranih mrežnih klijenata, to je paralelizam.

8.1.1 Razlika između sekvencijalnog i paralelnog računarstva, Sequential and Parallel Computing

Sekvencijalno računanje, također poznato kao serijsko računanje, odnosi se na korištenje jednog procesora za izvršavanje programa koji je razbijen na niz diskretnih instrukcija, od kojih se svaka izvršava jedna za drugom bez preklapanja u bilo kojem trenutku.

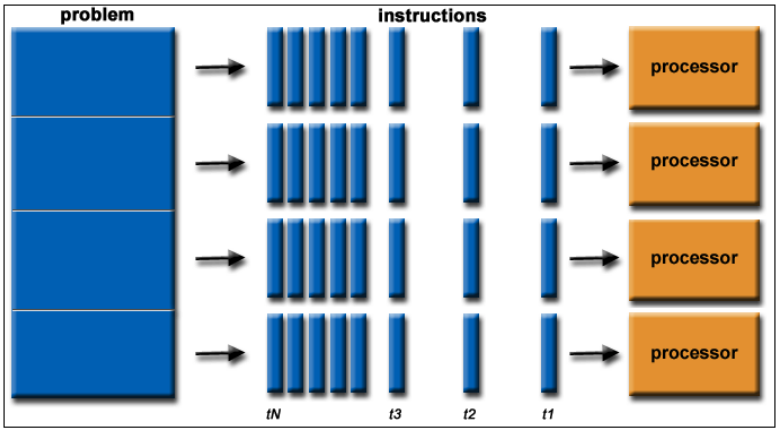


Softver je tradicionalno programiran sekvencijalno, što omogućava jednostavniji pristup, ali je značajno ograničen brzinom procesora i njegovom sposobnošću da izvrši svaku seriju instrukcija.

Tamo gdje jednoprocesorske mašine koriste sekvencijalne strukture podataka, strukture podataka za paralelna računarska okruženja su istovremene.

8.1.2 Razlika između paralelne obrade i paralelnog računanja, Parallel Processing and Parallel Computing

Paralelna obrada je metoda u računarstvu u kojoj se odvojeni dijelovi cjelokupnog složenog zadatka razbijaju i pokreću istovremeno na više CPU-a, čime se smanjuje količina vremena za obradu.



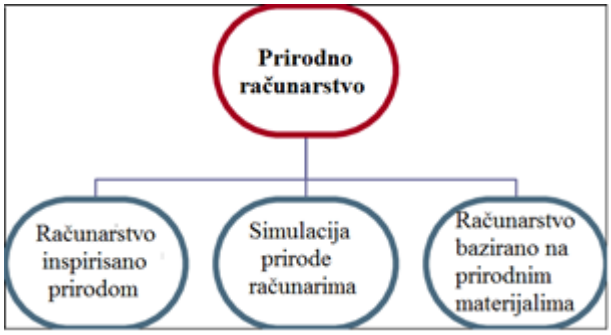
Podjelu i dodjeljivanje svakog zadatka različitom procesoru obično izvode programeri uz pomoć softverskih alata za paralelnu obradu, koji će također raditi na ponovnom sastavljanju i čitanju podataka nakon što svaki procesor riješi svoju specifičnu jednačinu.
Ovaj proces se ostvaruje ili preko računarske mreže ili preko računara sa dva ili više procesora.

Paralelna obrada i paralelno računanje se javljaju u tandemu, stoga se **termini često koriste naizmjenično**; međutim, kada se paralelna obrada odnosi na broj jezgara i CPU-a koji rade paralelno u računaru, paralelno računanje se odnosi na način na koji se softver ponaša da bi se optimizovao za to stanje.

8.2 Prirodno računarstvo Natural Computing

Prirodno računarstvo, koje se naziva i prirodno računanje (**natural computation**), je termin koji opisuje tri srodne oblasti:

- računarstvo inspirisano prirodnim pojavama,
- simulaciju procesa koji se odvijaju u prirodi i
- računarstvo koje za osnovu ima neke biološke materijale.



Računarske paradigme koje proučava prirodno računarstvo su apstrahovane od prirodnih pojava koje su raznolike kao što su samoreplikacija, funkcionisanje mozga, darvinistička evolucija, grupno ponašanje, imunološki sistem, definišući svojstva i razvoj životnih oblika.
Prirodna izračunavanja pokušavaju da imitiraju izračunavanja onako kako se ona odvijaju u prirodi:

- Izučavanjem modela izračunavanja i računarskih paradigmi sa kojima je priroda eksperimentisala milijardama godina
- njihovim implementiranjem u izračunavanjima koja se obavljaju u laboratorijskim uslovima (in vitro), ili
- njihovim implementiranjem u izračunavanjima koja se vrše u informatici (in info), simboličkim izrazima, i eventualno implementiranim u silicijumskim medijima.

Prirodna izračunavanja obuhvataju četiri osnovne oblasti:

1. neuronska izračunavanja, koja su inspirisana međusobno povezanim neuronskim strukturama u mozgu i nervnom sistemu
2. evoluciona izračunavanja, koja koriste koncepte mutacije, rekombinacije gena i prirodne selekcije iz biologije
3. kvantna izračunavanja, koja su zasnovana na kvantnoj fizici i koriste kvantni paralelizam
4. molekularna izračunavanja, koja su zasnovana na paradigmama iz molekularne biologije

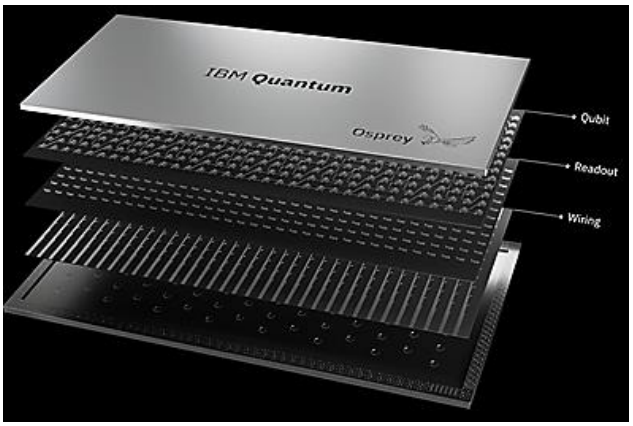
8.2.1 Kvantno računanje

Kvantno računarstvo proučava teorijske računarske sisteme (**kvantne računare**) koji direktno koriste kvantno-mehaničke fenomene, kao što su *superpozicija* i *uplitanje*, da bi izvodili operacije nad podacima. Kvantni računari se razlikuju od binarnih digitalnih elektronskih računara zasnovanih na tranzistorima. Dok obično digitalno računanje zahtijeva da podaci budu kodirani u binarnim ciframa (bitovima), od kojih je svaki uvijek u jednom od dva definitna stanja (0 ili 1), kvantno računanje koristi kvantne bitove (qubit), koji mogu biti u superpoziciji stanja. **Kvantna Tjuringova mašina** je teorijski model takvog računara, i poznat je takođe kao univerzalni kvantni računar. Kvantni računari dijele teorijske sličnosti sa *nedeterminističkim i probabilističkim računarima*.

Kvantni kompjuteri poštuju Church-Turingovu tezu. To znači da kvantni računari ne pružaju nikakve dodatne prednosti u odnosu na klasične računare u smislu izračunljivosti, ali kvantni algoritmi za određene probleme imaju znatno manju vremensku složenost od odgovarajućih poznatih klasičnih algoritama. Vjeruje se da kvantni računari mogu brzo riješiti određene probleme koje nijedan klasični računar ne bi mogao riješiti u bilo kojem izvodljivom vremenu – što je poznato kao "kvantna nadmoć".

Feynman (1982) je postavio hipotezu da je nemoguće simulirati fenomene kvantne fizike klasičnim računarom bez eksponencijalnog smanjenja brzine efikasnosti simulacije. On je sugerisao da bi korišćenjem kvantnog računara koji radi prema zakonima kvantne fizike bila moguća efektivna simulacija. Drugim rečima, Feynman je prvi predložio kvantni računar koji bi imao eksponencijalno veću efikasnost računanja od bilo kog klasičnog računara.

Postoje kvantni algoritmi, kao što je Simonov algoritam, koji se izvršavaju brže nego bilo koji mogući probabilistički klasični algoritam.



U novebru 2022 IBM je predstavio svoj najnoviji kvantni računar Osprey, koji ima 433 kubita i najavio 4000-plus kubita do 2025

Danas je razvoj stvarnih kvantnih računara još uvijek u začetku. Vrše se eksperimenti u kojima se kvantne računarske operacije izvršavaju na vrlo malom broju kvantnih bita.

Praktična i teorijska istraživanja se nastavljaju, i mnoge nacionalne vlade i vojne agencije finansiraju kvantno-računarska istraživanja u naporu da razviju kvantne računare za civilne, poslovne, trgovinske, ekološke i sigurnosne svrhe, kao što je *kriptoanaliza*.

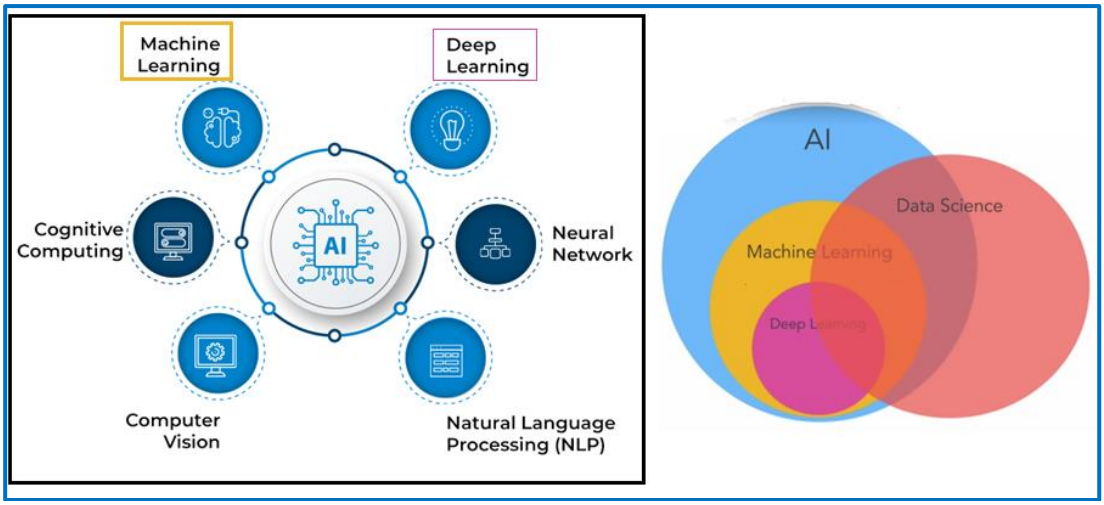
8.2.2 Molekularno računanje

Sljedeća velika promjena u računarskim naukama i informacionoj tehnologiji će doći od oponašanja tehnika pomoću kojih biološki organizmi procesiraju informacije. Da bi se to postiglo računarski naučnici će morati da sarađuju sa ekspertima za teme koje se obično ne povezuju sa njihovim poljem rada, uključujući organsku hemiju, molekularnu biologiju, bioinženjering i pametne materijale.
Molekularno računarstvo je interdisciplinarno polje. Ono obuhvata sve od apstraktnih principa do izgradnje stvarnih sistema. Ova oblast uključuje upotrebu proteina i drugih molekula za procesiranje informacija, molekularno prepoznavanje, računanje u nelinearnim medijima, računare zasnovane na fizičkim reakciono-difuzionim sistemima koji se nalaze u hemijskim medijima, DNK računarstvo (DNA computing), bioelektroniku i proteinski zasnovane optičke računare i biosenzore.
Svi današnji digitalni računari, kao i kvantni računari su u svojoj osnovi Univerzalne Tjuringove Mašine (UTM), tačnije oni vrše simulaciju Tjuringove mašine na ograničen, deterministički način. Stvaranje mašine koja bi radila po nedeterminističkom principu je smatrano nemogućim.
2004. godine dogodio se prvi proboj. Ehud Shapiro sa izraelskog Instituta Weizmann, sa grupom saradnika radio je na konstruisanju programabilnog molekularnog sistema za molekularni računar koji bi umesto poluprovodničkih čipova bio sastavljen od enzima i DNK molekula. U takvom uređaju bi DNK molekul snabdjevao računar podacima i osiguravao svu energiju koja je računaru potrebna.
Od 2012. godine DNK se uspješno koristi za skladištenje podataka. Naučnici su u više navrata demonstrirali kako se raznim metodama u DNK mogu čuvati digitalni podaci poput JPG slika, muzike, knjiga…

U DNK računarstvu, informacije su predstavljene korištenjem genetske abecede od četiri znaka (A [adenin], G [gvanin], C [citozin] i T [timin]), umjesto binarnog sistema (1 i 0) koji koriste tradicionalni kompjuteri.
Ovo je ostvarivo jer se kratki molekuli DNK bilo koje proizvoljne sekvence mogu sintetizirati po narudžbini. Unos algoritma je stoga predstavljen (u najjednostavnijem slučaju) molekulama DNK sa specifičnim sekvencama, instrukcije se izvode laboratorijskim operacijama na molekulima (kao što je njihovo sortiranje prema dužini ili sjeckanje niti koje sadrže određenu podsekvenciju), a rezultat se definiše kao neko svojstvo konačnog skupa molekula (kao što je prisustvo ili odsustvo određene sekvence).

8.3 Vještačka inteligencija i samoučeći algoritmi

Izdvojićemo jednu, još uvijek novu, ali u praksi korištenu klasu: samoučeće algoritme, poznate i kao Automatski obučeni algoritmi.
Samoučeći algoritmi koriste tehnike i tehnologiju vještačke inteligencije (AI), koja je i sama još uvijek u fazi razvoja. Danas se različite metode vještačke inteligencije primjenjuju, a očekuju se i dalja poboljšanja.
U suštini, samoučeći mašinski algoritam je dizajniran da sam poboljša svoje performanse. Ovakve algoritme obično koristi specijalna klasa programa poznata pod imenom agenti koji stupju u interakciju sa svojim okruženjem tako što opažaju okolinu putem senzora, a zatim djeluje kroz aktuatore ili efektore.

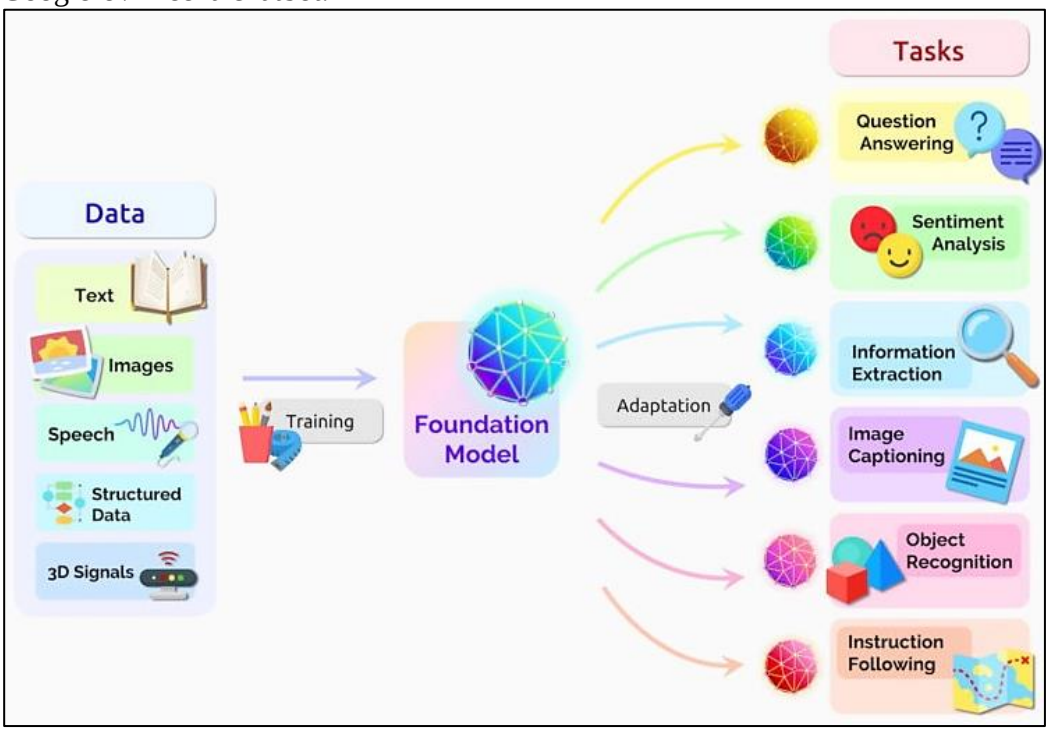


Mašinsko učenje je dio AI, a sve više potaje i dio "realnog svijeta"

Koncept različitih tipova algoritama za samoučenje je baziran na razvoju sistema dubokog učenja koji samostalno uči kako da popuni praznine. Duboko učenje (Deep learning) je dio metoda mašinskog učenja (ML: machine learning) zasnovanih na vještačkim neuronskim mrežama sa učenjem reprezentacije. Učenje može biti nadzirano, polunadzirano ili bez nadzora.

Najpopularniji model gdje su implementirani samoučeći algoritmi je **Transformers**.
Ova arhitektura se pokazala vrlo uspješnom u obradi prirodnog jezika. Transformersima nisu potrebni uređeni i označeni podaci. Ovi sistemi su obučeni na velikim korpusima nestrukturiranog teksta kao što su postovi na Wikipediji, blogovi na društvenim mrežama i slično. Ali ove vrste algoritama za samoučenje su još uvijek vrlo daleko od stvarnog razumijevanja ljudskog jezika.

Transformeri su postali veoma poznati i osnov su za gotovo sve najsavremenije jezičke sisteme, uključujući Facebookovu Robertu, Google-ov BERT, OpenAI-jev GPT2 i Google-ov Meena chatbot.



Transformeri, koji se ponekad nazivaju osnovnim modelima (*Foundation model*)**, već se koriste sa mnogim izvorima podataka za mnoštvo aplikacija, izvor: https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/**

ML algoritam je procedura. Izvršava se na skupu podataka da biste kreirali izlaz. Programeri mogu opisati ML algoritam u koristeći pseudokod ili neki matematički zapis prilagođen arhitekturi platforme gdje će da se implementira.
Možete koristiti popularne programske jezike (Python, Scala, R programming, Java and JavaScript, C++, Golang, Lisp…) za kodiranje algoritma i analizirati njegovu realizaciju.

ML model je rezultat koji proizvodi ML algoritam. Sadrži detalje o "učenju" algoritma. Priroda ML algoritma određuje vrstu detalja sadržanih u modelu, npr.: ML model može sadržavati pravila i vrijednosti koje je algoritam „naučio“. ML model može uključivati strukture podataka potrebne za kreiranje predviđanja. Može sadržavati proceduru za predviđanje iz ulaznih varijabli. U nekim slučajevima, ML model može čak sadržavati cijeli skup podataka za obuku.
Postoje određene sličnosti između statističkih modela učenja i modela mašinskog učenja. Pojednostavljeno, kod ML učenja to je kombinacija ststistike, heuristike i vještačke inteligencije.

Ovi algoritmi čitaju ulazne skupove podataka i analiziraju ih. Na osnovu ove analize, ovi algoritmi predviđaju izlazne vrijednosti. ML algoritmi drže ova predviđanja unutar prihvatljivog raspona vrijednosti i tokom vremena dodaju nove podatke algoritmima ML-a (sami sebi).
ML algoritmi „uče“ iz ovih novih skupova podataka i optimizuju svoje performanse. Mašinsko učenje je važna sposobnost unutar superskupa umjetne inteligencije (AI. Organizacije ih često koriste u kombinaciji s drugim AI sposobnostima kao što su kompjuterski vid i obrada prirodnog jezika (NLP).

Sem problema vezanih za poznavanje tehnike programiranja i matematike, kod ovog tipa algoritma se javlja i etički problemi.
Prije svega:
Ko je taj koji određuje funkciju cilja, naročito kod nenadziranog učenja?
Ko kontroliše upotrebu prikupljenih podataka?
…
U krajnjoj (vjerovatno dalekoj) instanci da li će računar upravljan ovakvim algoritmom služiti čovjeku, ili će čovjek postati dodatak računaru?

Izvori i reference

https://rivaltimes.com/ibm-presents-osprey-its-quantum-processor-with-more-than-400-qubits/s [1, 2023]

Dušan T. Malbaški, Algoritmi i strukture podataka, Univerzitet Educons Fakultet informacionih tehnologija, https://educons.edu.rs/wp-content/uploads/2015/11/ malbaski-algoritmi-i-strukture-podataka.pdf [12, 2022]

Marić, Lucija, Problem (ne)odlučivosti, Master's thesis, 2022, Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, https://urn.nsk.hr/urn:nbn:hr:217:653786 [12, 2022]

Milan Balać, Rekurzivni algoritmi, Završni rad Sveučilište J.J. Strossmayera u Osijeku, Filozofski fakultet, Preddiplomski studij Informatologije, [11, 2022]

https://repozitorij.ffos.hr/islandora/object/ffos%3A1475/datastream/PDF/view[11, 2022]

https://www.finsliqblog.com/ai-and-machine-learning/what-are-the-types-of-self-learning-algorithms/ [11, 2022]

https://www.devteam.space/blog/what-are-machine-learning-algorithms/ [11, 2022]

Dejan Živković, Uvod u algoritme i strukture podataka, Univerzitet Singidunum, Beograd, 2010

G. Polya, How To Solve It A New Aspect of Mathematical Method, Stanford University; https://pdfcoffee.com/qdownload/polya-howtosolveitpdf-pdf-free.html [11, 2017]

Robert Manger, Miljenko Marušić, Struktura podataka i algoritmi, skripta, Drugo izdanje, Zagreb, 2003.

Simbolička logika, -priručnik-, dr Berislav Žaranić, ww.vusst./hrčlogika [11, 2017]

https://razno.sveznadar.info/ [8, 2020]

Mala kompjuteska enciklopedija

MKE

Banja Luka, 2023

Ovaj priručnik ima isključivo edukativnu namjenu