Glava 1

Algoritmi i izračunljivost

Računari mogu da obave razne zadatke, mnoge – na zadivljujući način. Zato je interesantno a i veoma važno pitanje šta sve računari *mogu* a i pitanje šta računari *ne mogu* da urade. Da bismo mogli da odgovorimo na ta pitanja, moraćemo da odgovorimo na pitanje šta je to *algoritam*[1]. Neformalno govoreći, algoritam je precizan opis postupka za rešavanje nekog problema u konačnom broju koraka. Svaki računarski *program* je konkretna implementacija nekog algoritma u nekom konkretnom programskom jeziku, na primer, u jeziku Java, C ili C++ i koju računar može da izvrši. Danas granica između algoritama i programa nije kruta jer postoje, na primer, sistemi u kojima opisi algoritama u vidu dijagrama mogu da se neposredno izvršavaju (na primer, Blockly/Scratch dijagrami).

Algoritmi se primenjuju svakodnevno i na svakom koraku. Postoje, na primer, algoritmi za sabiranje i za množenje prirodnih brojeva, za određivanje najmanjeg elementa niza, za uređivanje elemenata po veličini i slično. U okviru razmatranja pojma algoritma i programa, razmatraju se samo postupci kojima se od nekih podataka dobijaju nekakvi rezultati, tj. novi podaci (ne razmatraju se postupci u kojima je potrebno obavljati neku fizičku radnju i slično). Pošto se svi podaci digitalizacijom mogu zapisati brojevima (možda uz neki gubitak), dovoljno je razmatrati algoritme za izračunavanje funkcija čiji su i argumenti i rezultujuće vrednosti prirodni brojevi. Pitanja šta može a šta ne može da uradi računar svode se na pitanja šta se može a šta ne može izračunati algoritamski.

Pitanje šta može da se izračuna algoritamski i od strane mašine, postavljala su se i pre pojave prvih računara. Ljudi su hiljadama godina pravili sprave koje su pomagale u računanju. Jedan od najvećih koraka napravio je Gotfrid Lajbnic još u XVII veku kada je napravio mehaničku mašinu za računanje koja je podržavala sve četiri osnovne računske operacije. Lajbnic je verovao da će biti moguće napraviti mašinski postupak koji će, manipulisanjem simbolima, biti u stanju da dâje odgovor na sva matematička pitanja. Koristeći osnovne operacije izračunavanja, mogu se opisati postupci za rešavanje složenijih matematičkih problema. Takvi postupci postojali su još u vreme starogrčkih matematičara (na primer, Euklidov algoritam za određivanje najvećeg zajedničkog delioca dva broja), pa i pre toga.

1.1 Prirodno-jezički opisi algoritama i izračunavanja

Svaki algoritam sačinjen je od operacija koje treba izvršiti da bi se rešio neki zadatak. To je još uvek daleko od prave definicije. Krenimo sa prirodno-jezičkim opisima algoritama i izračunavanja.

U programiranju (slično kao i u matematici) podaci se predstavljaju *promenljivama*. Međutim, promenljive u programiranju (za razliku od matematike) vremenom mogu da menjaju svoju vrednost (tada kažemo da im se *dodeljuje nova vrednost*). U programiranju, svakoj promenljivoj pridruženo je (jedno, fiksirano) mesto u memoriji i tokom izvršavanja programa promenljiva može da menja svoju vrednost, tj. sadržaj dodeljenog memorijskog prostora. Ako je promenljiva čija je vrednost ulazni parametar označena sa x, a promenljivoj y treba da bude dodeljena vrednost $2x + 3$, onda se to može opisati na sledeći način (gde simbol * označava množenje, a + sabiranje):

[1] Reč „algoritam“ ima koren u imenu persijskog astronoma i matematičara Al-Horezmija (engl. Muhammad ibn Musa al-Khwarizmi). On je 825. godine napisao knjigu, u međuvremenu nesačuvanu u originalu, verovatno pod naslovom „O računanju sa indijskim brojevima“. Ona je u dvanaestom veku prevedena na latinski, bez naslova, ali se na nju obično pozivalo njenim početnim rečima „Algoritmi de numero Indorum“, što je trebalo da znači „Al-Horezmi o indijskim brojevima“ (pri čemu je ime autora latinizovano u „Algoritmi“). Međutim, većina čitalaca je reč „Algoritmi“ shvatala kao množinu od nove, nepoznate reči „algoritam“ koja se vremenom odomaćila sa značenjem „metod za izračunavanje“.

```
dodeli promenljivoj y vrednost 2*x + 3
```

Kao naredni primer, razmotrimo postupak ili algoritam za maksimuma dva data broja. Pretpostavimo da promenljive x i y sadrže dve date brojevne vrednosti, a da promenljiva m treba da dobije vrednost veće od njih. Nakon sledećeg postupka promenljiva m ima vrednost većeg od zadata dva broja:

```
ako je x >= y onda
   dodeli promenljivoj m vrednost x
inače
   dodeli promenljivoj m vrednost y
```

Kao malo komplikovaniji primer razmotrimo stepenovanje. Na primer, n-ti stepen broja x (tj. vrednost x^n) moguće je izračunati uzastopnom primenom množenja: ako se krene od vrednosti 1 i ona se n puta sa pomnoži brojem x, rezultat će biti x^n. Da bi moglo da se osigura da će množenje biti izvršeno tačno n puta, koristi se dodatna promenljiva i koja na početku dobija vrednost 0, a zatim se, prilikom svakog množenja, uvećava sve dok ne dostigne vrednost n. Ovaj postupak možemo predstaviti sledećim opisom:

```
dodeli promenljivoj s vrednost 1
dodeli promenljivoj i vrednost 0
dok je i < n radi sledeće:
   dodeli promenljivoj s vrednost s*x
   dodeli promenljivoj i vrednost i+1
```

Kada se ovaj postupak primeni na vrednosti $x = 3$ i $n = 2$, izvodi se naredni niz koraka:

`s` dobija vrednost `1`	
`i` dobija vrednost `0`	pošto je `i(=0)` manje od `n(=2)`, vrše se dalje operacije
`s` dobija vrednost `s*x = 1*3 = 3`	
`i` dobija vrednost `i+1 = 0+1 = 1`	pošto je `i(=1)` manje od `n(=2)`, vrše se dalje operacije
`s` dobija vrednost `s*x = 3*3 = 9`	
`i` dobija vrednost `i+1 = 1+1 = 2`	pošto `i(=2)` nije manje od `n(=2)`, ne vrše se dalje operacije.

1.2 Programi i izračunavanja na višem programskom jeziku

Na prvim elektronskim računarima moglo je da se programira samo na *mašinski zavisnim programskim jezicima* — na jezicima specifičnim za konkretnu mašinu na kojoj program treba da se izvršava. Polovinom 1950-ih nastali su prvi *jezici višeg nivoa* i oni su drastično olakšali programiranje. Danas se programi obično pišu u *višim programskim jezicima* a zatim prevode na *mašinski jezik* — jezik razumljiv računaru.

Algoritmi i izračunavanja koja su u prethodnom delu opisana na prirodnom jeziku mogu da se pretoče u programe – konkretne implementacije – na višem programskom jeziku kao što je, na primer, C ili C++:

```
y = 2*x + 3;
```

Primetimo da u navedenom kodu, simbol `=` ne označava jednakost nego operaciju dodele. U drugim jezicima ovo izračunavanje opisuje se na slične načine. Naglasimo da navedeno izračunavanje mora da bude deo neke malo šire celine kako bi moglo da se izvrši (na primer, mora da se navede kojoj vrsti brojeva pripadaju `x` i `y`, promenljiva `x` na neki način mora da dobije svoju vrednost, itd). I u naredna dva primera implementacije na jeziku C++ slične su prirodno-jezičkim opisima:

```
if (x >= y) {
   m = x;
}
else {
   m = y;
}
```

```
s = 1;
i = 0;
while (i < n) {
   s = s*x;
   i = i+1;
}
```

1.3 Izračunavanja i programi na mašinski zavisnim jezicima

U prethodnom delu izračunavanja opisana na prirodnom jeziku i na jeziku C/C++ jasna su čoveku, čak i onom koji nije nikad programirao. Međutim, takvi opisi nisu razumljivi računaru. Računar može da „razume“ i izvrši samo programe napisane na mašinskom jeziku. Kao što je rečeno, na prvim elektronskim računarima moglo je da se programira samo na mašinskom jeziku.

Procesor je centralni deo računara i on podržava izvesne naredbe, tj. *primitivne instrukcije* koje su hardverski implementirane i koje su sve veoma jednostavne. Postoje, na primer, instrukcije za sabiranje dva broja, za množenje dva broja, instrukcije za poređenje dva broja, instrukcije za konjunkcija bitova, instrukcija skoka na neki korak u programu i slično.

Svaki program na mašinskom jeziku sastavljen je od niza procesorskih primitivnih instrukcija. Naredbe mogu da se izvršavaju redom, jedna za drugom, a kada je potrebno da se neke naredbe ponove veći broj puta ili da se određene naredbe preskoče, koriste se *naredbe skoka*. Procesor ima određeni broj *registara*, memorijskih jedinica u koje može neposredno da upisuje i iz kojih može neposredno da čita podatke. Sve instrukcije na mašinskom jeziku zapisuju se u vidu brojeva, koji su u računaru zapisani u binarnom sistemu (tj. u obliku niza nula i jedinica). Da bi mogao da se izvrši na računaru, program na mašinskom jeziku mora da budu smešten u memoriju, u vidu niza brojeva koji odgovaraju nizu naredbi.

Svi zadaci koje računari izvršavaju svode se na ovakve programe, tj. na nizove primitivnih instrukcija procesora. Kompleksni algoritmi implementirani na ovakvom jeziku mogu se sastojati od ogromnog broja primitivnih instrukcija, te je programiranje na mašinskom jeziku veoma zahtevno i naporno.

Procesori različitih računara mogu da se razlikuju (na primer, po tome koliko registara u procesoru imaju, koje instrukcije može da izvrši njihova aritmetičko-logička jedinica, itd). Razvoj najvećeg broja procesora usmeren je tako da većina savremenih procesora ima veoma slične skupove instrukcija, a isti mašinski programi mogu se koristiti na čitavim familijama procesora.

1.3.1 Programi na asembleru

Asemblerski jezik je jezik koji je veoma blizak mašinskom jeziku računara, ali se, umesto brojevnog zapisa za instrukcija koriste (mnemotehničke, lako pamtljive) simboličke oznake instrukcija (tj. programi se zapisuju u vidu teksta). Da bi ovako napisan program mogao da se izvršava, neophodno je izvršiti njegovo prevođenje na mašinski jezik (tj. zapisati instrukcije u vidu brojeva, i to binarnom azbukom) i uneti na odgovarajuće mesto u memoriji. Ovo prevođenje je trivijalno i jednoznačno i vrše ga programi koji se nazivaju *asembleri*. Korišćenjem asemblera olakšava se programiranje, ali skoro sva zahtevnost programiranja na mašinskom jeziku prisutna je i dalje.

Pretpostavimo da naš hipotetički procesor sadrži, između ostalog, tri registra označena sa `ax`, `bx` i `cx` i još nekoliko izdvojenih bitova (tzv. zastavica). Dalje, pretpostavimo da procesor može da izvršava naredne *aritmetičke instrukcije* (zapisane ovde u asemblerskom obliku):

- Instrukcija `add ax, bx` označava operaciju sabiranja vrednosti brojeva koji se nalaze u registrima `ax` i `bx`, pri čemu se rezultat sabiranja smešta u registar `ax`. Operacija `add` može se primeniti na bilo koja dva registra.
- Instrukcija `mul ax, bx` označava operaciju množenja vrednosti brojeva koji se nalaze u registrima `ax` i `bx`, pri čemu se rezultat množenja smešta u registar `ax`. Operacija `mul` može se primeniti na bilo koja dva registra.
- Instrukcija `cmp ax, bx` označava operaciju poređenja vrednosti brojeva koji se nalaze u registrima `ax` i `bx` i rezultat pamti postavljanjem zastavice u procesoru. Operacija `cmp` se može primeniti na bilo koja dva registra.

Zarad specifikovanja instrukciju na koju se vrši skok, koriste se *labele* – označena mesta u programu. Pretpostavimo da naš procesor može da izvršava, između ostalog, sledeće dve vrste skokova (bezuslovne i uslovne):

- Instrukcija `jmp label`, gde je `label` neka labela u programu, označava bezuslovni skok koji uzrokuje nastavak izvršavanja programa od mesta u programu označenog navedenom labelom.

- Uslovni skokovi prouzrokuju nastavak izvršavanja programa od instrukcije označene navedenom labelom, ali samo ako je neki uslov ispunjen. Ukoliko uslov nije ispunjen, izvršava se naredna instrukcija. U nastavku će se razmatrati samo instrukcija `jge label`, koja uzrokuje uslovni skok na mesto označeno labelom `label` ukoliko je vrednost prethodnog poređenja brojeva bila *veće ili jednako*.

Tokom izvršavanja programa, podaci se nalaze u memoriji i u registrima procesora. S obzirom na to da procesor sve operacije može da izvrši isključivo nad podacima koji se nalaze u njegovim registrima, svaki procesor podržava i *instrukcije prenosa podataka* između memorije i registara procesora (kao i između samih registara). Pretpostavimo da naš procesor podržava sledeću instrukciju ove vrste.

- Instrukcija `mov` označava operaciju prenosa podataka i ima dva parametra — prvi određuje gde se podaci prenose, a drugi koji određuje koji se podaci prenose. Parametar može biti ime registra (što označava da se pristupa podacima u određenom registru), broj u zagradama (što označava da se pristupa podacima u memoriji i to na adresi određenoj brojem u zagradama) ili samo broj (što označava da je podatak baš taj navedeni broj). Na primer, instrukcija `mov ax bx` označava da se sadržaj registra `bx` prepisuje u registar `ax`, instrukcija `mov ax, [10]` označava da se sadržaj iz memorije sa adrese `10` prepisuje u registar `ax`, instrukcija `mov ax, 1` označava da se u registar ax upisuje vrednost `1`, dok instrukcija označava `mov [10], ax` da se sadržaj registra `ax` upisuje u memoriju na adresu `10`.

Opišimo za ovakav procesor izračunavanje vrednosti $2x+3$ (podsetimo se da izračunavanje mora da se razloži na primitivne operacije). Pretpostavimo da se ulazni podatak (broj x) nalazi u glavnoj memoriji i to na adresi `10`, a da rezultat y treba smestiti na adresu `11` (ovo su sasvim proizvoljno odabrane adrese). Izračunavanje se onda može opisati sledećim programom (nizom instrukcija).

```
mov ax, [10]
mov bx, 2
mul ax, bx
mov bx, 3
add ax, bx
mov [11], ax
```

Instrukcija `mov ax, [10]` prepisuje vrednost promenljive x (iz memorije sa adrese `10`) u registar `ax`. Instrukcija `mov bx, 2` upisuje vrednost `2` u registar `bx`. Nakon instrukcije `mul ax, bx` vrši se množenje i registar `ax` sadrži vrednost $2x$. Instrukcija `mov bx, 3` upisuje vrednost `3` u registar `bx`, nakon instrukcije `add ax, bx` se vrši sabiranje i u registru `ax` se nalazi tražena vrednost $2x+3$. Na kraju se ta vrednost instrukcijom `mov [11], ax` upisuje u memoriju na adresu `11`.

Određivanje maksimuma dva broja može se ostvariti na sledeći način. Pretpostavimo da se ulazni podaci nalaze u glavnoj memoriji i to broj x na adresi `10`, broj y na adresi `11`, dok rezultat m treba smestiti na adresu `12`. Program (niz instrukcija) kojima može da se odredi maksimum je sledeći:

```
mov ax, [10]
mov bx, [11]
cmp ax, bx
jge vecix
mov [12], bx
jmp kraj
vecix:
mov[12], ax
kraj:
```

Nakon prenosa vrednosti oba broja u registre procesora (instrukcijama `mov ax, [10]` i `mov bx, [11]`), vrši se njihovo poređenje (instrukcija `cmp ax, bx`). Ukoliko je broj x veći od ili jednak broju y prelazi se na mesto označeno labelom `vecix` (instrukcijom `jge vecix`) i na mesto rezultata upisuje se vrednost promenljive x (instrukcijom `mov[12], ax`). Ukoliko uslov skoka `jge` nije ispunjen (ako x nije veće ili jednako y), na mesto rezultata upisuje se vrednost promenljive y (instrukcijom `mov [12], bx`) i bezuslovno se skače na kraj programa (instrukcijom `jmp kraj`) (da bi se preskočilo izvršavanje instrukcije koja na mesto rezultata upisuje vrednost promenljive x).

Procesori skoro uvek podržavaju instrukcije kojima se izračunavaju zbir i proizvod dva cela broja, ali stepenovanje obično nije podržano kao primitivna instrukcija. Izračunavanje vrednosti x^n može se ostvariti na sledeći način. Pretpostavimo da se ulazni podaci nalaze u glavnoj memoriji i to broj x na adresi `10`, a broj n na adresi `11`, i da konačan rezultat treba da bude smešten u memoriju i to na adresu `12`. Pretpostavimo da će pomoćne promenljive s i i koje se koriste u postupku biti smeštene sve vreme u procesoru, i to promenljiva s u registru `ax`, a promenljiva i u registru `bx`. Pošto postoji još samo jedan registar (`cx`), u njega će naizmenično biti smeštane vrednosti promenljivih n i x, kao i konstanta 1 koja se sabira sa promenljivom i. Niz instrukcija koji implementira algoritam za stepenovanje opisan u prethodnom delu (i koji odgovara opisu iz poglavlja i) je sledeći:

```
  mov ax, 1
  mov bx, 0
petlja:
  mov cx, [11]
  cmp bx, cx
  jge kraj
  mov cx, [10]
  mul ax, cx
  mov cx, 1
  add bx, cx
  jmp petlja
kraj:
  mov [12], ax
```

1.3.2 Mašinski jezik

Fon Nojmanova arhitektura podrazumeva da se i sam program (niz instrukcija) nalazi u glavnoj memoriji prilikom njegovog izvršavanja. Potrebno je svaki program (poput tri navedena) predstaviti nizom nula i jedinica, na način „razumljiv“ procesoru — na mašinskim jeziku. Na primer, moguće je da su binarni kodovi za instrukcije uvedeni na sledeći način:

`mov`	`001`
`add`	`010`
`mul`	`011`
`cmp`	`100`
`jge`	`101`
`jmp`	`110`

Takođe, pošto neke instrukcije primaju podatke različite vrste (neposredno navedeni brojevi, registri, apsolutne memorijske adrese), uvedeni su posebni kodovi za svaki od različitih vidova adresiranja. Na primer:

neposredno	`00`
registarsko	`01`
apsolutno	`10`

Pretpostavimo da registar `ax` ima oznaku `00`, registar `bx` ima oznaku `01`, a registar `cx` oznaku `10`. Pretpostavimo i da su sve adrese osmobitne. Pod navedenim pretpostavkama, instrukcija `mov [10], ax` se, u ovom hipotetičkom mašinskom jeziku, može kodirati kao `001 10 01 00010000 00`. Kôd `001` dolazi od instrukcije `mov`, zatim slede `10` i `01` koji ukazuju da prvi argument predstavlja memorijsku adresu, a drugi oznaku registra, za čim sledi memorijska adresa $(10)_{16}$ binarno kodirana sa `00010000` i na kraju oznaka `00` registra `ax`. Na sličan način, celokupan prikazani mašinski kôd navedenog asemblerskog programa koji izračunava $2x + 3$ je moguće binarno kodirati kao:

```
001 01 10 00 00010000   // mov ax, [10]
001 01 00 01 00000010   // mov bx, 2
011 00 01               // mul ax, bx
001 01 00 01 00000011   // mov bx, 3
010 00 01               // add ax, bx
001 10 01 00010001 00   // mov [11], ax
```

Između prikazanog asemblerskog i mašinskog programa postoji veoma direktna i jednoznačna korespondencija (u oba smera) tj. na osnovu datog mašinskog koda moguće je jednoznačno rekonstruisati asemblerski kôd.

Specifični hardver koji čini kontrolnu jedinicu procesora dekodira jednu po jednu instrukciju i izvršava akciju zadatu tom instrukcijom. Kod realnih procesora, broj instrukcija i načini adresiranja su bogatiji a prilikom pisanja programa potrebno je uzeti u obzir mnoge aspekte na koje se u navedenim jednostavnim primerima nije obraćala pažnja. Ipak, mašinske i asemblerske instrukcije stvarnih procesora veoma su slične hipotetičkim instrukcijama navedenim ovde za ilustraciju.

1.4 Zasnivanje pojma algoritma

Kroz nekoliko navedenih primera ilustrovali smo pojam algoritma, ali da bi se pitanje šta se uopšte može izračunati algoritamski i mnoga slična mogla precizno razmatrati, neophodno je najpre definisati (matematički rigorozno) šta je to algoritam. Iako su razni algoritmi razvijani vekovima, do početka dvadesetog veka retko se postavljalo pitanje kako formalno zasnovati pojam algoritma i koji su sve problemi mašinski, algoritamski rešivi. Prve precizne definicija algoritma ipak su prethodile prvim računarima i date su početkom dvadesetog veka, u jeku reforme i novog utemeljivanja matematike. Jedno od fundamentalnih pitanja postavljenih tada bilo je pitanje da li postoji algoritam kojim se (pojednostavljeno rečeno) mogu dokazati sve matematičke teoreme.

Algoritam možemo opisati kao bilo koji postupak koji može da se izvrši na računaru, preciznije na njegovom procesoru (ne razmatramo postupke koji uključuju periferije: emitovanje zvuka ili slike, štampanje i slično). Time dolazimo do pitanja šta očekujemo od jednog računara ili procesora da može da izvrši kao svoje osnovne operacije. Međutim, nisu svi procesori isti, pa se ovaj pristup komplikuje - možemo onda da razmatrmo izračunavanja koja mogu da se izvrše da proizvoljnom računaru. S jedne strane, želimo da naš pojam algoritma pokriva sve što može da se uradi na savremenom računaru, a s druge – da taj pojam bude što jednostavniji, kako bi bio podesniji za formalna razmatranja.

Očigledno je da očekujemo da sabiranje brojeva može i treba da se smatra algoritamskim postupkom i da (proizvoljni) procesor može da ga izvršava. Ne samo da je sabiranje očigledno operacija potrebna u mnogim problemima, nego je ova operacija jasno opisiva. Zaista, algoritam za sabiranje brojeva uči se još u prvom razredu osnovne škole. Dakle, od (svakog) procesora očekujemo da ume da sabira brojeve. Slično, kako je množenje brojeva važno i precizno opisivo, od (svakog) procesora očekujemo da ume da množi brojeve. Međutim, pitanje je da li od procesora očekujemo da ume da neposredno računa, na primer, determinantu matrice. Odgovor je da ne mora, pošto se to računanje može svesti na množenja i sabiranja (naravno, u tome pretpostavljamo da procesor može da se instruira i da ponavlja neke radnje pod nekim uslovima). Slično, kao što smo videli u prethodnom delu, nije neophodno da procesor kao osnovnu operaciju ima stepenovanje prirodnih brojeva, jer se ono može svesti na množenje. Zapravo, uviđamo da onda od procesora ne treba da očekujem da ume neposredno ni da množi, jer se množenje može svesti na sabiranje. Zaista, proizvod prirodnih brojeva x i y jednaj je zbiru $x + x + \ldots + x$ u kojem se vrednost x pojavljuje y puta. Ali, onda možemo ići i još dalje: nije nam potrebno ni sabiranje sve dok naš procesor ume da nekoj vrednosti dodaje vrednost 1 pod nekim uslovom. Zaista, zbir prirodnih brojeva x i y jednak je zbiru $x + 1 + \ldots + 1$ u kojem se vrednost 1 pojavljuje y puta. Takvim razmatranjem dolazimo do opisa veoma jednostavnog procesora koji će poslužiti za definisanje pojma algoritma.

1.4.1 UR mašine

UR mašina je apstraktna mašina koja ne postoji u fizičkom obliku ali predstavlja matematičku idealizaciju računara i omogućava zasnivanje pojma algoritma[2]: reći ćemo da je algoritam onaj i samo onaj postupak koji može da se opiše kao program za UR mašinu.

UR mašina raspolaže beskonačnim skupom memorijskih lokacija – *registara*, koji služe kao prostor za čuvanje (brojevnih) podataka. Registri su označeni prirodnim brojevima: $, 2, 3, \ldots$. Svaki od njih u svakom trenutku sadrži neki prirodan broj. Stanje registara u nekom trenutku zovemo *konfiguracija*. Sadržaj k-tog registra označava se sa r_k, kao što je to ilustrovano na sledećoj slici:

r_1	r_2	r_3	$\ldots$

U prethodnom delu teksta, naslutili smo da je idealizovanom računaru potreban i dovoljan mali broj operacija. Tako je dizajnirana i UR mašina – ona podržava:

- operaciju upisivanja nule u memorijsku lokaciju;
- operaciju dodavanja jedinice na sadržaj memorijske lokacije;
- kopiranje sadržaja sa jedne u drugu lokaciju;

[2]Prvi opis dat je u jednom radu Šeperdsona i Sturdžisa (engl. Sheperdson and Sturgis) iz 1963. godine.

- operaciju skoka na određeno mesto u programu ukoliko je sadržaj dve lokacije jednak.

Zapis i kratak opis URM instrukcija (naredbi) dati su u tabeli 1.1.[3] U tabeli, na primer, $r_m := 0$ označava da se vrednost 0 upisuje u m-ti registar, a $r_m := r_m + 1$ označava da se sadržaj m-tog registra uvećava za jedan.

oznaka	naziv	efekat
$Z(m)$	nula–instrukcija	$r_m := 0$
$S(m)$	instrukcija sledbenik	$r_m := r_m + 1$
$T(m,n)$	instrukcija prenosa	$r_n := r_m$
$J(m,n,p)$	instrukcija skoka	ako je $r_m = r_n$, idi na p-tu; inače idi na sledeću instrukciju

Tabela 1.1: Tabela URM instrukcija

URM program P je konačan numerisan niz URM instrukcija. Instrukcije se izršavaju redom (počevši od prve), osim u slučaju instrukcije skoka. Izvršavanje programa se zaustavlja onda kada ne postoji instrukcija koju treba izvršiti (kada se dođe do kraja programa ili kada se naiđe na skok na instrukciju koja ne postoji u numerisanom nizu instrukcija).

Početnu konfiguraciju čini niz prirodnih brojeva $a_1, a_2, \ldots$ koji su upisani u registre od početnog redom. Ako je funkcija koju treba izračunati $f(x_1, x_2, \ldots, x_n)$, onda se podrazumeva da su vrednosti $x_1, x_2, \ldots, x_n$ redom smeštene u prvih n registara. Početne vrednosti u registrima iza njih nisu poznate unapred. Podrazumeva se i da, na kraju rada programa, rezultat treba da bude smešten u prvi registar.

Ako URM program P za početnu konfiguraciju $a_1, a_2, \ldots, a_n$ ne staje sa radom, onda pišemo $P(a_1, a_2, \ldots, a_n) \uparrow$. Ako program staje sa radom i u prvom registru je, kao rezultat, vrednost b, onda pišemo $P(a_1, a_2, \ldots, a_n) \downarrow b$.

Kažemo da URM program izračunava funkciju $f : \mathbb{N}^n \to \mathbb{N}$ ako za svaku n-torku argumenata $a_1, a_2, \ldots, a_n$ za koju je funkcija f definisana i važi $f(a_1, a_2, \ldots, a_n) = b$ istovremeno važi i $P(a_1, a_2, \ldots, a_n) \downarrow b$. Funkcija je URM-*izračunljiva* ako postoji URM program koji je izračunava.

Primer 1.1. *Neka je funkcija f definisana na sledeći način: $f(x, y) = x + y$. Vrednost funkcije f može se izračunati za sve vrednosti argumenata x i y* UR *mašinom. Ideja algoritma za izračunavanje vrednosti $x + y$ je da se vrednosti x doda vrednost 1, y puta, jer važi:*

$$x + y = x + \underbrace{1 + 1 + \cdots + 1}_{y}$$

Odgovarajući URM *program podrazumeva sledeću početnu konfiguraciju:*

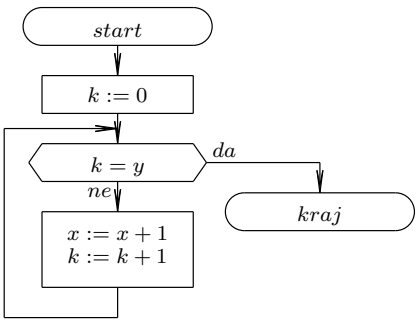
x	y	$\ldots$	$\ldots$

i sledeću konfiguraciju u toku rada programa:

$x+k$	y	k	$\ldots$

gde je $k \in \{0, 1, \ldots, y\}$.

Algoritam se može zapisati u vidu dijagrama toka i u vidu URM *programa kao u nastavku:*



[3] Oznake URM instrukcija potiču od naziva ovih instrukcija na engleskom jeziku (*zero instruction, succesor instruction, transfer instruction* i *jump instruction*).

1. $Z(3)$
2. $J(3,2,100)$
3. $S(1)$
4. $S(3)$
5. $J(1,1,2)$

Prekid rada programa realizovan je skokom na nepostojeću instrukciju 100[4]*. Bezuslovni skok je realizovan naredbom oblika* $J(1,1,\ldots)$ *— poređenje registra sa samim sobom uvek garantuje jednakost te se skok vrši uvek.*

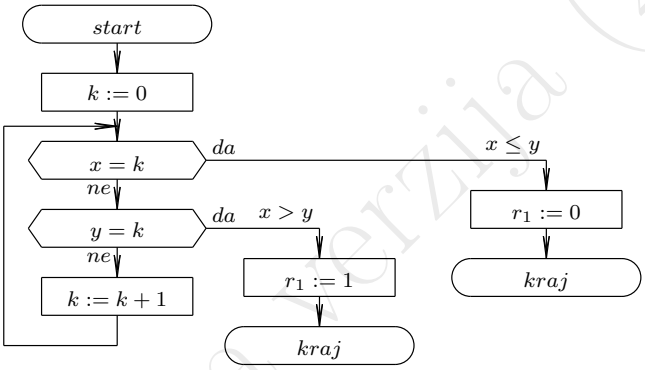
Primer 1.2. *Neka je funkcija* f *definisana na sledeći način:*

$$f(x,y) = \begin{cases} 0 & , \text{ ako } x \leq y \\ 1 & , \text{ inače} \end{cases}$$

URM *program koji je računa koristi sledeću konfiguraciju u toku rada:*

x	y	k	$\ldots$

gde k *dobija redom vrednosti* $0, 1, 2\ldots$ *sve dok ne dostigne vrednost* x *ili vrednost* y*. Prva dostignuta vrednost je broj ne veći od onog drugog. U skladu sa tim zaključkom i definicijom funkcije* f*, izračunata vrednost je* 0 *ili* 1*.*



1. $Z(3)$ $\quad k = 0$
2. $J(1,3,6)$ $\quad x = k?$
3. $J(2,3,8)$ $\quad y = k?$
4. $S(3)$ $\quad k := k + 1$
5. $J(1,1,2)$
6. $Z(1)$ $\quad r_1 := 0$
7. $J(1,1,100)$ $\quad kraj$
8. $Z(1)$
9. $S(1)$ $\quad r_1 := 1$

Navedena dva primera ilustruju dva izračunavanja nad prirodnim brojevima. Možemo se sada zapitati da li se nekako mogu vršiti i izračunavanja koja se odnose na brojeve koji nisu prirodni, pa ni celi, posebno imajući u vidu da neki brojevi nemaju konačan decimalni zapis, na primer, broj $\sqrt{2}$. Iako taj broj nije ceo, možemo, na primer, zahtevati da se izračuna njegova k-ta cifra. Pokazaćemo, bez upuštanja u detalje, da to jeste izračunljivo. Koristeći osobinu $n = [\sqrt{x}] \Leftrightarrow n^2 \leq x < (n+1)^2$, može se pokazati da je izračunljiva naredna funkcija:

$$f(x) = [\sqrt{x}]$$

Pošto je izračunljiva funkcija f, očigledno je izračunljiva i funkcija $g(k) = [10^k \cdot \sqrt{2}]$. Poslednja cifra vrednosti $g(k)$ (što je takođe izračunljivo) jednaka je k-toj cifri broja $\sqrt{2}$. Ovo ilustruje način na koji možemo vršiti izračunavanja koja se ne odnose na prirodne brojeve.

Primer 1.3. *Neka je funkcija* f *definisana na sledeći način:*

$$f(x) = \begin{cases} 0 & , \text{ ako je } x = 0 \\ nedefinisano & , \text{ inače} \end{cases}$$

[4]Broj 100 je odabran proizvoljno kao broj sigurno veći od broja instrukcija u programu. I u programima koji slede, uniformnosti radi, za prekid programa će se takođe koristiti skok na instrukciju 100.

Nju izračunava naredni program (nedefinisanost funkcije f postiže se time što se program izvršava beskonačno izuzev ako je vrednost argumenta jednaka 0):

1. $Z(2)$
2. $J(1, 2, 100)$
3. $J(1, 1, 2)$

1.4.2 Enumeracija URM programa

Važno pitanje je koliko uopšte ima različitih URM programa. Očigledno je da ih ima beskonačno, ali postoje razne vrste beskonačnosti. Za dva skupa kaže se da *imaju istu kardinalnost* ako i samo ako je između njih moguće uspostaviti bijektivno preslikavanje. Za skupove koji imaju istu kardinalnost kao skup prirodnih brojeva kaže se da su *prebrojivi*. Dakle, neki skup je prebrojiv ako i samo ako je njegove elemente moguće poređati u niz (niz odgovara bijekciji sa skupom prirodnih brojeva). Za skupove koji su ili konačni ili prebrojivi, kaže se da su *najviše prebrojivi*. Skup realnih brojeva nije prebrojiv, a postoje i mnogi skupovi koji imaju mnogo veće kardinalnosti.

Primer 1.4. *Skupovi parnih i neparnih brojeva imaju istu kardinalnost jer je između njih moguće uspostaviti bijektivno preslikavanje $f(n) = n + 1$. Štaviše, oba ova skupa su prebrojiva, tj. imaju istu kardinalnost kao i skup prirodnih brojeva, iako su njegovi pravi podskupovi (na primer, funkcija $f(n) = 2 \cdot n$ uspostavlja bijekciju između skupa prirodnih i skupa parnih brojeva).*

Primetimo da se u zapisu URM programa koriste samo sledeći simboli: Z, S, T, J, (,), , i deset cifara za zapis brojeva. Zato se može napisati samo konačno mnogo programa koji imaju ukupno k simbola, $k = 1, 2, 3, ...$ (zapravo, za neke dužine, kao što su 1 ili 2 taj broj je jednak nuli). Dalje, zbog toga možemo u niz poređati sve URM programe: najpre sve one dužine 1 (recimo po alfabetskom redu), pa onda dužine 2, pa dužine 3, itd. Odatle sledi naredna teorema.

Teorema 1.1. *Različitih* URM *programa ima prebrojivo mnogo*

Na osnovu navedene teoreme, moguće je uspostaviti bijekciju između skupa svih URM programa i skupa prirodnih brojeva. Drugim rečima, može se definisati pravilo koje svakom URM programu dodeljuje jedinstven prirodan broj i koje svakom prirodnom broju dodeljuje jedinstven URM program. Zbog toga, za fiksirano dodeljivanje brojeva programima, možemo da govorimo o prvom, drugom, trećem, ..., stotom URM programu. Za bilo koju vrednost i, smatramo da je funkcija koju izračunava program P_i algoritamski izračunljiva. I obratno, ako je neka funkcija algoritamski izračunljiva onda mora da postoji broj i, takav da program P_i izračunava tu funkciju. Na sličan način može se pokazati da i na bilo kom programskom jeziku ima prebrojivo mnogo programa.

1.4.3 Drugi pristupi zasnivanju pojma algoritma

U prethodnom tekstu, pojam izračunljivosti uveden je na bazi UR *mašina (*URM*)*. Taj formalizam je elegantan, intuitivan i blizak savremenom programiranju, ali nije jedini. Posebno tokom 1920-ih i 1930-ih godina više velikih matematičara bavilo se zasnivanjem pojma algoritma i izračunljivosti i oni su razvili više raznorodnih formalizama. Najznačajniji među njima verovatno su: *Tjuringove mašine* (Tjuring), *rekurzivne funkcije* (Gedel[5] i Klini[6]) i *λ-račun* (Čerč[7]). *Blok dijagrami* (kaže se i *algoritamske šeme*, *dijagrami toka*, tj. *tokovnici*), kao oni prikazani u primerima 1.1 i 1.2, mogu se smatrati poluformalnim načinom opisa algoritama.

Ako se neka funkcija može izračunati u nekom od navedenih formalizama (tj. ako se u tom formalizmu za sve moguće argumente mogu izračunati vrednosti funkcije), onda kažemo da je ona *izračunljiva* u tom formalizmu. Za sistem izračunavanja koji je dovoljno moćan da izvrši sva izračunavanja koja može da izvrši Tjuringova mašina kaže se da je *Tjuring potpun* (engl. *Turing complete*).

Iako su pomenuti formalizmi međusobno veoma različiti, može se rigorozno dokazati da su klase izračunljivih funkcija identične za sve njih, tj. svi oni formalizuju isti pojam algoritma i izračunljivosti. Drugim rečima, svi navedeni formalizmi su Tjuring potpuni i ekvivalentni. Zbog toga se, umesto pojmova poput, na primer, Tjuring-izračunljiva ili URM-izračunljiva funkcija (i sličnih) može koristiti samo termin *izračunljiva funkcija*. Konačno, možemo reći da je algoritam svaki postupak kojim se izračunava neka izračunljiva funkcija.

[5] Kurt Gödel (1906-1978), austrijsko-američki matematičar.
[6] Stephen Kleene (1909-1994), američki matematičar.
[7] Alonzo Church (1903-1995), američki matematičar.

1.4.4 Savremeni računari i izračunljivost

Sada, kada nam je poznat formalni pojam algoritma, možemo da se zapitamo da li savremeni računari mogu da izvrše svaki algoritam, što bi bilo očekivano. Savremeni računari (tj. savremeni procesori koji su centralni deo računara) očigledno mogu da izvrše sve instrukcije koje ima UR mašina, pa mogu da izračunaju sve funkcije koje mogu da izračunaju nabrojani formalizmi za izračunavanje, tj. sve izračunljive funkcije. Međutim, ne mogu da izračunaju ništa što ne mogu ti formalizmi! Štaviše, striktno govoreći, njihova moć je i manja: svi navedeni formalizmi za izračunavanje podrazumevaju, pojednostavljeno rečeno, beskonačnu raspoloživu memoriju. Zbog toga savremeni računari (koji raspolažu konačnom memorijom) nisu Tjuring potpuni, mada mogu da izrše sve URM programe koji ne zahtevaju beskonačnu memoriju.

Savremeni računar ima procesor koji sadrži ugrađene komponente za sabiranje, oduzimanje, množenje, deljenje, poređenje celih brojeva i brojeva u pokretnom zarezu. Takvim operacijama odgovaraju instrukcije procesora, a on ima još i instrukcije koje vrše skok na određenu instrukciju u programu. Dakle, procesor savremenog računara nije suštinski mnogo bogatiji od UR mašine. Zapravo, mogao bi da se napravi procesor (i računar oko njega) koji može da izvršava samo URM instrukcije. To bi bilo dovoljno da računar može da radi bilo šta što može koristeći neki savremeni procesor. Takav URM procesor bio bi sigurno znatno jeftiniji (jer je znatno jednostavniji), ali bi ipak bio nepraktičan i neefikasan jer bi se, na primer, operacije množenja velikih brojeva svodile na ogroman broj dodavanja vrednosti 1.

Sledeće važno pitanje je da li nekakvi drugačiji računari mogu da izračunaju nešto što ne može UR mašina. Na primer, kvantni računari imaju bitno drugačiju arhitekturu od sada dominantnih računara. Oni neka izračunavanja mogu da izvrše znatno brže nego obični računari, ali i dalje ne mogu da izračunaju ništa što ne mogu i obični računari. Trenutno ne postoji računar niti arhitektura za (fizički, ne apstraktni) računar koja omogućava izračunavanje koje ne omogućava UR mašina.

1.4.5 Saremeni programski jezici i izračunljivost

Na *savremenim programskim jezicima* takođe se mogu implementirati svi algoritmi, te ih je moguće pridružiti navedenoj listi pristupa zasnivanja algoritama. Zaista, za veliku većinu savremenih programskih jezika važi da su Tjuring-kompletni. Značajna razlika je u tome što nabrojani formalizmi teže da budu što jednostavniji tj. da koriste što manji broj operacija i što jednostavnije modele mašina (a u cilju jednostavnije formalne analize), dok savremeni programski jezici teže da budu što udobniji za programiranje te uključuju veliki broj operacija (koje, sa teorijskog stanovišta, nisu neophodne jer se mogu definisati preko malog broja osnovnih operacija). [8]

1.4.6 Čerč-Tjuringova teza

Pominjane formalne definicije izračunljivosti i algoritama su ekvivalentne i to je moguće rigorozno dokazati. Veoma važno pitanje je i koliko formalne definicije algoritama uspevaju da pokriju naš intuitivni pojam algoritma, tj. da li čovek zaista može efektivno izvršiti sva izračunavanja definisana nekom od formalizacija izračunljivosti i, obratno, da li sva izračunavanja koja čovek intuitivno ume da izvrši zaista mogu da budu opisana korišćenjem bilo kog od precizno definisanih formalizama izračunavanja. Očigledno je odgovor na prvo pitanje potvrdan (jer čovek može da simulira rad jednostavnih mašina za izračunavanje), ali oko drugog pitanja postoji doza rezerve. *Čerč-Tjuringova teza*[9] tvrdi da je odgovor na oba navedena pitanja potvrdan.

Čerč-Tjuringova teza: *Klasa intuitivno izračunljivih funkcija identična je sa klasom formalno izračunljivih funkcija.*

Ovo tvrđenje je hipoteza, a ne teorema i ne može biti formalno dokazano. Naime, ono govori o intuitivnom pojmu algoritma, čija svojstva ne mogu biti formalno, matematički ispitana (jer formalno, matematički ne mogu biti opisani ni procesi koji se odvijaju u ljudskom mozgu). U korist ove teze govori činjenica da do sada nije pronađen nijedan primer intuitivno, efektivno izvodivog postupka izračunavanja koji nije moguće formalizovati u okviru nabrojanih formalnih sistema izračunavanja. Ovim dolazimo do uverenja da je skup funkcija koje može da izračuna čovek jednak skupu funkcija koje može da izvrši bilo koji savremeni računar i jednak je skupu funkcija koje može da izvrši UR mašina (ovde pretpostavljamo da za izračunavanja nije potrebno beskonačno memorijskih

[8] Naredba skoka u programima može da dovodi do nečitljivih i nerazumljivih (tzv. špageti) programa. Dokazano je da ona i nije neophodna i da je dovoljno da programski jezik podržava samo sekvencijalno nizanje naredbi, naredbu izbora (if-then-else) i barem jednu vrstu petlje (na primer, do-while). Ovo tvrdenje je poznato kao teorema o strukturnom programiranju, Korado Bema (nem. Corrado Böhm) i Duzepea Jakopinija (it. Giuseppe Jacopini) iz 1966. godine.

[9] Ovu tezu, svaki za svoj formalizam, formulisali su nezavisno Čerč i Tjuring.

lokacija). U ovoj vezi, naravno, potpuno zanemarujemo pitanje dužine trajanja izračunavanja koje vrši čovek i koje vrše mašine.

1.4.7 Neizračunljivost i neodlučivost

Algoritmima se, za zadate argumente, izračunavaju neke vrednosti, tj. algortimima se opisuju izračunljive funkcije. Veliko pitanje je da li postoje funkcije nad brojevima koje su jasno definisane i totalne (definisane za sve vrednosti argumenata) a nisu izračunljive. Drugim rečima, postoje li problemi koji ne mogu da se reše algoritamski. Razmotrimo narednih nekoliko problema.

1. Već pomenuto pitanje da li postoji algoritam kojim se (pojednostavljeno rečeno) mogu dokazati sve matematičke teoreme može se takođe može svesti na izračunavanje neke funkcije. Ovo pitanje može se formulisati i kao pitanje da li postoji algoritam koji za proizvoljni zadati skup aksioma i zadato tvrđenje proverava da li je tvrđenje posledica aksioma. Ovaj problem, poznat pod imenom „Entscheidungsproblem“, postavio je David Hilbert 1928. godine. Ipak, 1930-ih, rezultatima Čerča, Tjuringa i Gedela pokazano je da ovakav postupak ne može da postoji. Međutim, pre ovog znamenitog rezultata trebalo je opisati šta se uopšte može smatrati algoritmom.

2. Neka su data dva konačna skupa reči. Pitanje je da li je moguće nadovezati nekoliko reči prvog skupa i, nezavisno, nekoliko reči drugog skupa tako da se dobije ista reč. Na primer, za skupove $\{a, ab, bba\}$ i $\{baa, aa, bb\}$, jedno rešenje je $bba \cdot ab \cdot bba \cdot a = bb \cdot aa \cdot bb \cdot baa$. Za skupove $\{ab, bba\}$ i $\{aa, bb\}$ rešenje ne postoji, jer se nadovezivanjem reči prvog skupa uvek dobija reč čija su poslednja dva slova različita, dok se nadovezivanjem reči drugog skupa uvek dobija reč čija su poslednja dva slova ista. Zadatak je konstruisati opšti algoritam koji za proizvoljna dva zadata skupa reči određuje da li tražena nadovezivanja postoje.[10]

3. Diofantska jednačina je jednačina oblika $p(x_1, \ldots, x_n) = 0$, gde je p polinom sa celobrojnim koeficijentima. Zadatak je konstruisati opšti algoritam kojim se određuje da li proizvoljna zadata diofantska jednačina ima racionalnih rešenja.[11]

4. Zadatak je konstruisati opšti algoritam koji proverava da li se proizvoljni zadati program P zaustavlja za date ulazne parametre.[12]

Za sva četiri navedena problema pokazano je da su *algoritamski nerešivi* ili *neodlučivi* (tj. ne postoji izračunljiva funkcija koja ih rešava). Ovo ne znači da nije moguće rešiti pojedine instance problema[13], već samo da ne postoji jedinstven, opšti postupak koji bi mogao da reši proizvoljnu instancu problema. Pored nabrojanih, ima još mnogo važnih neodlučivih problema. Poznavanje neodlučivih problema u informatici je veoma važno i može se uporediti sa poznavanjem činjenice iz fizike i mašinstva da nije moguće napraviti *perpetuum mobile* – mašinu koja proizvodi jednako ili više energije nego što je njoj predato. Kao što je besmisleno pokušavati napraviti perpetuum mobile, tako je besmisleno pokušavati napraviti program koji rešava neki od nerešivih problema.

U nastavku će precizno, korišćenjem formalizma UR mašina, biti opisan četvrti od navedenih problema, tj. *halting problem*, izuzetno važan za programiranje.

1.5 Zaustavljanje programa i halting problem

Pitanje zaustavljanja računarskih programa je jedno od najznačajnijih pitanja računarstva i programiranja. Često je veoma važno da li se neki program zaustavlja za neku ulaznu vrednost, da li se zaustavlja za ijednu ulaznu vrednost i slično. Za mnoge konkretne programe i za mnoge konkretne ulazne vrednosti, na ovo pitanje može se odgovoriti. No, nije očigledno da li postoji opšti postupak kojim se za proizvoljni dati program i proizvoljne vrednosti ulaznih argumenata može proveriti da li se program zaustavlja ako se pokrene sa tim argumentima.

Iako se problem ispitivanja zaustavljanja može razmatrati i za programe u savremenim programskim jezicima, u nastavku ćemo razmotriti formulaciju halting problema za URM programe:

[10] Ovaj problem se naziva *Post's correspondence problem*, jer ga je postavio i rešio Emil Post 1946. godine.

[11] Ovaj problem je 10. Hilbertov problem izložen 1900. godine kao deo skupa problema koje „matematičari XIX veka ostavljaju u amanet matematičarima XX veka“. Problem je rešio Matijaševič 1970-ih godina.

[12] Ovaj problem rešio je Alan Tjuring 1936. godine.

[13] Instanca ili *primerak problema* je jedan konkretan zadatak koji ima isti oblik kao i opšti problem. Na primer, za prvi u navedenom spisku problema, jedna instanca je zadatak ispitivanja da li je moguće nadovezati nekoliko reči skupa $\{ab, bba\}$ i, nezavisno, nekoliko reči skupa $\{aa, bb\}$ tako da se dobije ista reč.

Halting problem: *Da li postoji* URM *program koji na ulazu dobija drugi* URM *program P i neki broj x i ispituje da li se program P zaustavlja za ulazni parametar x?*

Problem prethodne formulacije je činjenica da traženi URM program mora na ulazu da prihvati kao svoj argument drugi URM program, što je naizgled nemoguće, s obzirom na to da URM programi kao argumente mogu da imaju samo prirodne brojeve. Ipak, ovo se jednostavno razrešava zahvaljujući tome što je svakom URM programu P moguće dodeliti jedinstveni prirodan broj n koji ga identifikuje, i obratno, svakom broju n može se dodeliti program P_n (kao što je opisano u poglavlju 1.4.2). Imajući ovo u vidu, dolazi se do teoreme o halting problemu za URM.

Teorema 1.2 (Neodlučivost halting problema)**.** *Neka je funkcija h definisana na sledeći način:*

$$h(x,y) = \begin{cases} 1, & \text{ako se program } P_x \text{ zaustavlja za ulaz } y \\ 0, & \text{inače.} \end{cases}$$

Ne postoji program koji izračunava funkciju h, tj. ne postoji program koji za proizvoljne zadate vrednosti x i y može da proveri da li se program P_x zaustavlja za ulazni argument y.

Dokaz: Pretpostavimo da postoji program H koji izračunava funkciju h. Onda se jednostavno može konstruisati i program H' sa jednim argumentom x, koji vraća rezultat isti rezultat kao i program $H(x,x)$, tj. koji vraća rezultat 1 (tj. upisuje ga u prvi registar) ako se program P_x zaustavlja za ulaz x, a rezultat 0 ako se program P_x ne zaustavlja za ulaz x. Dalje, postoji i program Q (dobijen kombinovanjem programa H' i programa iz primera 1.3) koji za argument x vraća rezultat 0 ako se P_x ne zaustavlja za x (tj. ako je $h(x,x) = 0$), a izvršava beskonačnu petlju ako se P_x zaustavlja za x (tj. ako je $h(x,x) = 1$). Za program Q važi:

$$Q(x) \downarrow 0 \text{ ako je } P_x(x) \uparrow$$
$$Q(x) \uparrow \text{ ako je } P_x(x) \downarrow$$

Ako postoji takav program Q, onda se i on nalazi u nizu svih programa tj. postoji redni broj k koji ga jedinstveno identifikuje, pa važi:

$$P_k(x) \downarrow 0 \text{ ako je } P_x(x) \uparrow$$
$$P_k(x) \uparrow \text{ ako je } P_x(x) \downarrow$$

No, ako je x jednako upravo k, pokazuje se da je definicija ponašanja programa Q (tj. programa P_k) kontradiktorna: program Q (tj. program P_k) za ulaznu vrednost k vraća 0 ako se P_k ne zaustavlja za k, a izvršava beskonačnu petlju ako se P_k zaustavlja za k:

$$P_k(k) \downarrow 0 \text{ ako je } P_k(k) \uparrow$$
$$P_k(k) \uparrow \text{ ako je } P_k(k) \downarrow$$

Dakle, polazna pretpostavka je bila pogrešna i ne postoji program H, tj. funkcija h nije izračunljiva. Pošto funkcija h, karakteristična funkcija halting problema, nije izračunljiva, halting problem nije odlučiv. □

Neodlučivost halting problema ne odnosi se samo na URM programe, već i na bilo koji savremeni, Tjuring-kompletan programski jezik (uključujući C, C++, Java, itd).

Halting problem je neodlučiv, tj. ne postoji opšti postupak kojim se za proizvoljni zadati program može utvrditi da li se on zaustavlja za zadate vrednosti argumenata. Ipak, za mnoge konkretne programe, može se utvrditi da li se zaustavljaju ili ne. Kako ne postoji opšti postupak koji bi se primenio na sve programe, zaustavljanje svakog programa mora se ispitivati zasebno i koristeći specifičnosti tog programa.

U programima u kojima su petlje jedine naredbe koje mogu dovesti do nezaustavljanja potrebno je dokazati zaustavljanje svake pojedinačne petlje. Ovo se obično radi tako što se definiše dobro zasnovana relacija[14] takva

[14] Za relaciju $\succ$ se kaže da je *dobro zasnovana* (engl. well founded) ako ne postoji beskonačan opadajući lanac elemenata $a_1 \succ a_2 \succ \ldots$.

da su susedna stanja kroz koje se prolazi tokom izvršavanja petlje međusobno u relaciji. Kod elementarnih algoritama ovo se obično radi tako što se izvrši neko preslikavanje skupa stanja u skup prirodnih brojeva i pokaže da se svako susedno stanje preslikava u manji prirodan broj.[15] Pošto je relacija $>$ na skupu prirodnih brojeva dobro zasnovana, i ovako definisana relacija na skupu stanja biće dobro zasnovana.

Primer 1.5. *Algoritam koji vrši množenje uzastopnim sabiranjem (poglavlje 1.3.1) se zaustavlja. Zaista, u svakom koraku petlje vrednost $n - i$ je prirodan broj. Ova vrednost opada kroz svaki korak petlje (jer se n ne menja, a i raste), pa u jednom trenutku mora da dostigne vrednost* 0.

Primer 1.6. *Ukoliko se ne zna gornje ograničenje za polaznu vrednost n, nije poznato da li se naredna petlja uvek zaustavlja:*

```
while (n > 1) {
   if (n % 2)
      n = 3*n + 1;
   else
      n = n/2;
}
```

Opšte uverenje je da se funkcija zaustavlja za svaku ulaznu vrednost n (to tvrdi još uvek nepotvrđena Kolacova (Collatz) hipoteza *iz 1937). Navedeni primer pokazuje kako pitanje zaustavljanja čak i za neke veoma jednostavne programe može da bude ekstremno komplikovano.*

Naravno, ukoliko je poznata širina podatka `unsigned int`*, i ukoliko se testiranjem za sve moguće ulazne vrednosti pokaže da se* `f` *zaustavlja, to bi dalo odgovor na pitanje u specijalnom slučaju.*

1.6 Algoritmika i vremenska i prostorna složenost izračunavanja

Prvo pitanje koje se postavlja kada je potrebno izračunati neku funkciju (tj. napisati neki program) je da li je ta funkcija izračunljiva (tj. da li uopšte postoji neki program koji je izračunava). Ukoliko takav program postoji, sledeće pitanje je koliko izvršavanje tog program zahteva vremena i prostora (memorije). Najčešće se složenost algoritma određuje tako da ukazuje na to koliko on može utrošiti vremena i prostora u najgorem slučaju. Ponekad je moguće izračunati i prosečnu složenost algoritma — prosečnu za sve moguće vrednosti argumenata.

Primer 1.7. *Razmotrimo problem izračunavanja vrednosti x^n (za nenegativnu vrednost n). U poglavlju 1.2 već smo videli programski kôd koji izračunava traženu vrednost:*

```
s = 1;
i = 0;
while (i < n) {
   s = s*x;
   i = i+1;
}
```

Nije teško videti da će naredni kôd da izvrši upravo n množenja. Možemo se zapitati da li se tražena vrednost može izračunati i efikasnije. Primetimo sledeće: ako je vrednost n parna, onda je $x^n = (x^2)^{n/2}$ pa, na primer, ako je n jednako 6*, vrednost u izračunavanju može da se koristi veza $x^6 = (x^2)^3$. U izračunavanju vrednosti x^2 koristi se jedno množenje, a onda u izračunavanju $(x^2)^3$ još dva, ukupno tri, što je znatno nego šest koliko bi koristio početni način. Na ovoj ideji zasniva se sledeći kôd za izračunavanja vrednosti x^n:*

```
s = 1;
while (n > 0) {
   if (n % 2 == 0) {
      x = x*x;
      n = n/2;
   } else {
      s = s*x;
      n = n-1;
   }
```

[15] Smatramo da i nula pripada skupu prirodnih brojeva.

```
}
```

Može se pokazati da navedeni kôd u izračunavanju x^{16} koristi samo četiri množenja, a u izračunavanju x^{1024} samo deset. Generalno, u izračunavanju x^n koristi oko $\log_2 n$ množenja, što je za velike vrednosti n mnogo bolje nego polazni, naivni algoritam.

Navedeni primer ilustruje teme kojima se bavi oblast informatike koja se zove *algoritmika*: analizom vremenske i prostorne složenosti programa, kao i dizajnom što efikasnijih algoritama koji rešavaju neki problem.

1.7 Pregled

- Svi podaci koje se mogu obrađivati na računaru predstavljeni su brojevima. Računar svaku obradu podataka svodi na niz instrukcija koje vrše izračunavanje nad brojevima (ili vrše skok na neku instrukciju u nizu).
- Za računar je, suštinski, dovoljno da može da obavi svega nekoliko, veoma jednostavnih instrukcija.
- Neformalno, algoritam je precizan opis postupka za rešavanje nekog problema u konačnom broju koraka. Računarski program je konkretna implementacija nekog algoritma u nekom konkretnom programskom jeziku.
- Formalno, algoritam je svaki postupak koji se može opisati na mašini kao što je URM.
- Savemeni programski jezici mogu da služe i kao formalizam za opisivanje pojma algoritma.
- Bilo koji savremeni računar može da uradi ono i samo ono što može UR. Preciznije - može i manje, jer ima samo konačno mnogo memorijskih lokacija na raspolaganju.
- Bilo koji savremeni računar može da uradi ono i samo ono što može da izračuna i čovek – to govori Čerč-Tjuringova teza (koja se ne dokazuje). U ovoj vezi zanemaruje se brzina izvršavanja pojedinačnih operacija.
- Postoje totalne, precizno definisane funkcije nad brojevima koje se mogu i one koje se ne mogu izračunati algoritamski, na računaru.
- Mnogi važni problemi ne mogu se rešiti algoritamski. Jedan od njih je da ne postoji program koji za proizvoljni zadati program može da utvrdi da li taj program staje sa radom za neki zadati argument.
- Algoritmika je oblast informatike koja se bavi analizom vremenske i prostorne složenosti programa, kao i dizajnom što efikasnijih algoritama koji rešavaju neki problem.

Pitanja i zadaci za vežbu

Pitanje 1.1. *Po kome je termin* algoritam *dobio ime?*

Pitanje 1.2. *Šta je to algoritam (formalno i neformalno)? Navesti nekoliko formalizma za opisivanje algoritama. Kakva je veza između formalnog i neformalnog pojma algoritma. Šta tvrdi Čerč-Tjuringova teza? Da li se ona može dokazati?*

Pitanje 1.3. *Da li postoji algoritam koji opisuje neku funkciju iz skupa prirodnih brojeva u skup prirodnih brojeva i koji može da se isprogramira u programskom jeziku C i izvrši na savremenom računaru, a ne može na Tjuringovoj mašini?*

Pitanje 1.4. *Da li je svaka* URM *izračunljiva funkcija intuitivno izračunljiva? Da li je svaka intuitivno izračunljiva funkcija* URM *izračunljiva?*

Pitanje 1.5. *U čemu je ključna razlika između* URM *mašine i bilo kog stvarnog računara?*

Pitanje 1.6. *Opisati efekat* URM *naredbe $J(m, n, p)$.*

Pitanje 1.7. *Da li se nekim* URM *programom može izračunati hiljadita cifra broja 2^{1000} ?*

Pitanje 1.8. *Da li postoji* URM *program koji izračunava broj $\sqrt{2}$? Da li postoji* URM *program koji izračunava n-tu decimalnu cifru broja $\sqrt{2}$, gde je n zadati prirodan broj?*

Pitanje 1.9. *Da li se nekim* URM *programom može izračunati hiljadita decimalna cifra broja* π *?*

Pitanje 1.10. *Koliko ima racionalnih brojeva? Koliko ima kompleksnih brojeva? Koliko ima različitih programa za Tjuringovu mašinu? Koliko ima različitih programa u programskom jeziku C?*

Pitanje 1.11. *Koliko elemenata ima unija konačno mnogo konačnih skupova? Koliko elemenata ima unija prebrojivo mnogo konačnih skupova? Koliko elemenata ima unija konačno mnogo prebrojivih skupova? Koliko elemenata ima unija prebrojivo mnogo prebrojivih skupova?*

Pitanje 1.12. *Koliko ima različitih* URM *programa? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa prirodnih brojeva? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa realnih brojeva? Kakva je kardinalnost skupa* URM *programa u odnosu na kardinalnost skupa programa na jeziku C?*

Pitanje 1.13. *Da li se svakom* URM *programu može pridružiti jedinstven prirodan broj (različit za svaki program)? Da li se svakom prirodnom broju može pridružiti jedinstven* URM *program (različit za svaki broj)?*

Pitanje 1.14. *Da li se svakom* URM *programu može pridružiti jedinstven realan broj (različit za svaki program)? Da li se svakom realnom broju može pridružiti jedinstven* URM *program (različit za svaki broj)?*

Pitanje 1.15. *Kako se naziva problem ispitivanja zaustavljanja programa? Kako glasi* halting *problem? Da li je on odlučiv ili nije? Ko je to dokazao?*

Pitanje 1.16.

1. *Da li postoji algoritam koji za drugi zadati* URM *program utvrđuje da li se zaustavlja ili ne?*
2. *Da li postoji algoritam koji za drugi zadati* URM *utvrđuje da li se zaustavlja posle 100 koraka?*
3. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program proverava da li radi beskonačno?*
4. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program proverava da li vraća vrednost 1?*
5. *Da li je moguće napisati* URM *program kojim se ispituje da li data izračunljiva funkcija (ona za koju postoji* URM *program)* f *zadovoljava da je* $f(0) = 0$*?*
6. *Da li je moguće napisati* URM *program koji za drugi zadati* URM *program ispituje da li izračunava vrednost* 2012 *i zašto?*

Pitanje 1.17. *Na primeru korena uporedite URM sa savremenim asemberlskim jezicima. Da li URM ima neke prednosti?*

Zadatak 1.1. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = xy$. √

Zadatak 1.2. *Napisati* URM *program koji izračunava funkciju* $f(x) = 2^x$.

Zadatak 1.3. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = x^y$.

Zadatak 1.4. *Napisati* URM *program koji izračunava funkciju:*

$$f(x,y) = \begin{cases} 1 & , \text{ako } x \geq y \\ 0 & , \text{inače} \end{cases}$$

Zadatak 1.5. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} x - y & , \text{ako } x \geq y \\ 0 & , \text{inače} \end{cases}$$

√

Zadatak 1.6. *Napisati* URM *program koji izračunava funkciju:*

$$f(x) = \begin{cases} x/3 & , \text{ako } 3|x \\ nedefinisano & , \text{inače} \end{cases}$$

Zadatak 1.7. *Napisati* URM *program koji izračunava funkciju* $f(x) = x!$.

Zadatak 1.8. *Napisati* URM *program koji izračunava funkciju* $f(x) = \left[\frac{2x}{3}\right]$.

Zadatak 1.9. *Napisati* URM *program koji broj* 1331 *smešta u prvi registar.*

Zadatak 1.10. *Napisati* URM *program koji izračunava funkciju* $f(x) = 1000 \cdot x$.

Zadatak 1.11. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = 2x + y$.

Zadatak 1.12. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = min(x, y)$, *odnosno:*

$$f(x,y) = \begin{cases} x & , \text{ ako } x \leq y \\ y & , \text{ inače} \end{cases}$$

Zadatak 1.13. *Napisati* URM *program koji izračunava funkciju* $f(x, y) = 2^{(x+y)}$

Zadatak 1.14. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} 1 & , \text{ ako } x|y \\ 0 & , \text{ inače} \end{cases}$$

Zadatak 1.15. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y) = \begin{cases} \left[\frac{y}{x}\right] & , \text{ ako } x \neq 0 \\ nedefinisano & , \text{ inače} \end{cases}$$

Zadatak 1.16. *Napisati* URM *program koji izračunavaju sledeću funkciju:*

$$f(x,y) = \begin{cases} 2x & , x < y \\ x - y & , x \geq y \end{cases}$$

Zadatak 1.17. *Napisati* URM *program koji izračunava funkciju:*

$$f(x,y) = \begin{cases} x/3 & , 3|x \\ y^2 & , inace \end{cases}$$

Zadatak 1.18. *Napisati* URM *program koji izracunava funkciju* $f(x, y, z) = x + y + z$

Zadatak 1.19. *Napisati* URM *program koji izracunava funkciju* $f(x, y, z) = min(x, y, z)$.

Zadatak 1.20. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y,z) = \begin{cases} \left[\frac{y}{3}\right] & , \text{ ako } 2|z \\ x + 1 & , \text{ inače} \end{cases}$$

Zadatak 1.21. *Napisati* URM *program koji izračunava funkciju*

$$f(x,y,z) = \begin{cases} 1, & \text{ako je } x + y > z \\ 2, & \text{inače} \end{cases}$$