Zapiski za tekmovalno programiranje

ACM priprave na tekmovanja

Različica z dne 12. april 2025

Kazalo

1. Vhod in izhod **5**
- 1.1. Osnovna struktura programa . . . 5
- 1.2. Branje podatkov . . . 6

2. Računske operacije **9**
- 2.1. Seštevanje, odštevanje in množenje . . . 9
  - 2.1.1. Negativna števila . . . 10
- 2.2. Deljenje . . . 10

3. Pogojni stavki **13**
- 3.1. Osnovna struktura pogojnega stavka . . . 13

4. Nizanje pogojev **17**
- 4.1. Logični vezniki . . . 19

5. Zanke **21**
- 5.1. Sintaksa . . . 21
- 5.2. Primeri uporabe . . . 22
  - 5.2.1. Spreminjanje dolžine zanke . . . 22
  - 5.2.2. Branje števil v zanki . . . 22
  - 5.2.3. Zanka z drugačnim korakom in začetkom . . . 23

6. Seznami **25**
- 6.1. Sintaksa . . . 25
  - 6.1.1. Večdimenzionalni seznami . . . 28

7. Nizi in besedilo **29**
- 7.1. Predstavitev znakov . . . 29
- 7.2. Predstavitev nizov . . . 32
- 7.3. Standardne funkcije . . . 35
  - 7.3.1. Primerjava nizov . . . 35
  - 7.3.2. Kopiranje nizov . . . 36
  - 7.3.3. Operacije na prvih m znakih . . . 36

8. Funkcije **37**
- 8.1. Osnovna sintaksa . . . 37
- 8.2. Spremenljivke in funkcije . . . 40
- 8.3. Rekurzija . . . 42

9. Teorija števil **43**
9.1. Evklidov algoritem . . . 43
9.2. Eratostenovo rešeto . . . 44

10. Vhod in izhod: dopolnitev **47**
10.1. Dodatni formatniki za `scanf` . . . 47
10.2. Neznano število podatkov . . . 49
10.3. Uporaba nizov ali datotek za vhodno-izhodni sistem . . . 49

11. Spomin in kazalci **51**
11.1. Računalniški spomin . . . 51
11.2. Kazalci . . . 52
11.3. Kako delujejo seznami . . . 54
11.4. Podajanje po referenci . . . 55

12. Urejanje **59**
12.1. Primerjalna funkcija . . . 60
12.2. Urejanje sestavljenih podatkov . . . 61

13. Hitrost programov in asimptotična notacija **63**
13.1. Merjenje hitrosti programa . . . 63
13.2. Klasifikacija . . . 66

14. Bisekcija **67**
14.1. Osnovna oblika . . . 67
14.2. Implementacija v standardni knjižnici . . . 69
14.3. Bisekcija po odgovoru . . . 69

15. Podatkovne strukture **71**
15.1. Seznami . . . 73
15.1.1. Vektorji . . . 76
15.1.2. Skladi . . . 77
15.1.3. Vrste . . . 80
15.2. Množice . . . 82
15.2.1. Slovarji . . . 84
15.2.2. Neurejene množice in slovarji . . . 87
15.3. Kopice . . . 87
15.4. Primerjava . . . 88

A. Reševanje nalog na papir **91**
A.1. Tekmovanje RTK . . . 91
A.2. Primeri . . . 91
A.2.1. Pribor . . . 92
A.2.2. Palindromski oklepaji . . . 94

1. Vhod in izhod

1.1. Osnovna struktura programa

Programi za svoje delovanje potrebujejo način za komunikacijo z uporabnikom. Kompleksnejši programi v ta namen uporabljajo ekran, miško in tipkovnico, pri tekmovalnem programiranju pa najpogosteje uporabljamo najpreprostejši način za komunikacijo: pisanje in branje s *standardnega vhoda in izhoda*. Običajno to pomeni, da se nam ob zagonu programa odpre okno, kamor lahko pišemo programu in kamor program izpisuje stvari. Ko želimo, da naš program kaj izpiše, uporabimo *funkcijo* `printf`. Poglejmo si enostaven primer.

```
#include<stdio.h>

int main(){
    printf("Hello World!\n");
    return 0;
}
```

Funkciji `printf` v dvojnih narekovajih damo besedilo ali števila, ki jih želimo izpisati. Na koncu tega besedila napišemo `\n`, ki označuje, da mora program na tem mestu iti v novo vrstico. To je pomembno vključiti predvsem, če funkcijo `printf` uporabimo večkrat zaporedoma, saj bi bilo sicer celotno besedilo izpisano v eni vrstici.

Posvetimo se tudi splošni obliki zgornje kode, saj vsebuje ključne elemente, ki jih mora vsebovati vsak program. Prva vrstica, `#include <stdio.h>`, pove programu, da bomo uporabljali funkcije za vhod in izhod, konkretno `printf` in kasneje `scanf`. Če te vrstice nebi napisali, bi ob poskusu izvajanja kode sistem javil napako, in trdil, da funkcije `printf` ne pozna.

Besedilo `int main()` računalniku pove, da bodo sledili zaviti oklepaji (to so oklepaji, ki izgledajo {takole}), znotraj katerih bo glavno telo naše kode. Zaenkrat bomo vso našo kodo napisali med te zavite oklepaje, ko pa bomo spoznali sezname in kasneje funkcije, bomo nekaj kode vnesli tudi drugam. Koda v `main` je organizirana v vrstice, ki se morajo končati s podpičjem `;`. Ko se bo program izvedel, se bodo zaporedoma od zgoraj navzdol izvedle vse vrstice, dokler ne pridemo do zadnje vrstice, ki se mora začeti z ukazom `return`. Temu ukazu sledi številka — ta pove, če se je med izvajanjem programa zgodila kakšna napaka. Če je številka enaka 0, se je program končal brez napak, drugim številkam pa pravimo *kode napake*. Te so uporabne predvsem zato, da

lahko uporabnik programerju le z eno številko pove, kakšna napaka se je v programu zgodila. Mi bomo v prihodnje večinoma pisali programe, katerih uporabniki bomo sami, zato bomo vedno uporabili kodo 0.

V program lahko dodamo *komentarje*. To je besedilo, ki je sicer napisano v kodi programa, a ne vpliva na njegov potek, ker računalnik komentarjev ne izvede. Zaradi tega lahko komentarji vsebujejo tudi besedilo v naravnem jeziku (slovenščini), ki programerjem razlaga pomen kode poleg komentarja. Na voljo imamo dve vrsti komentarjev:

- Če na začetku vrstice napišemo dve poševnici, `//`, s tem dobimo komentar, ki prikriva besedilo v tej vrstici. Če se ta komentar pojavi sredi vrstice, bo prikril vse besedilo od tam naprej do konca vrstice.
- Če v besedilu zapišemo poševnico in zvezdico, `/*`, s tem dobimo komentar, ki prikriva vso besedilo do vključno prve pojavitve nasprotnega simbola, `*/`.

```c
#include<stdio.h>
int main(){ //komentar do konca vrstice
    printf /*komentar znotraj vrstice*/("Zivjo svet!\n");
    /*
      komentar
      čez
      več
      vrstic
    */
    return 0;
}
```

1.2. Branje podatkov

Programu lahko sporočimo različne podatke, program pa mora te podatke nekam shraniti, preden jih lahko obravnava. Mestu, kamor podatke shranimo, pravimo *spremenljivka*, saj lahko te podatke med tekom programa spreminjamo. Vsem spremenljivkam v programu damo ime, s katerim se na njih sklicujemo, ter *podatkovni tip*, ki pove, kakšni podatki so v spremenljivki shranjeni (npr. besedilo, številka, …). V spodnjem primeru ustvarimo eno spremenljivko, ki jo imenujemo `tvoje_ime`, njen tip pa je „besedilo dolžine največ 50“.

```c
#include <stdio.h>

int main(){
    char tvoje_ime[50];
```

```
    printf("Kako ti je ime?\n");
    scanf("%s", tvoje_ime);
    printf("Zivjo, %s!\n", tvoje_ime);
    return 0;
}
```

Tudi `scanf` je funkcija, ki ji podamo dva ali več *parametrov*. Prvi parameter mora vedno biti niz znotraj narekovajev, ki opisuje, kakšnega tipa so podatki, ki naj jih funkcija prebere. Ta opis podamo s *formatnikom*, v zgornjem primeru `%s` računalniku pove, da bo program prebral eno besedo. Preostali parametri povedo, v katero spremenljivko naj funkcija shrani prebrane podatke. Če imamo v nizu več formatnikov, moramo podati eno spremenljivko za vsak formatnik.

Do zdaj smo funkciji `printf` podali samo točno določeno besedilo, ki smo ga želeli izpisati. Izpisujemo pa lahko tudi spremenljivke, kot smo to naredili v tem zadnjem primeru. Znotraj besedila dodamo formatnike na mesta, kjer želimo, da so spremenljivke, potem pa izven narekovajev naštejemo imena spremenljivk, ki jih želimo izpisati.

V zgornjem primeru smo brali in izpisali niz besedila, kar pa je pravzaprav zahtevnejše od branja in pisanja števil. Za delo z nizi potrebujemo kompleksnejše ukaze, ki jih bomo spoznali kasneje, zato se bomo do nadaljnjega omejili na delo s (celimi) števili. Tem pripada tip `int` (angl. *integer*) ter formatnik `%d`, kakor vidimo v naslednjem primeru.

```
#include<stdio.h>

int main(){
    int razred;
    printf("Kateri razred si?\n");
    scanf("%d", &razred);
    printf("%d. razred je najboljši.\n", razred);
    return 0;
}
```

Pri branju števil imamo le eno dodatno zahtevo kot pri branju nizov — pred ime spremenljivke moramo zapisati znak `&`. Razlog za tem bomo spoznali, ko bomo obravnavali kazalce, za sedaj pa to vzemimo kot zahtevo postopka. Pri številih tudi ne povemo direktno največje dolžine, kakor smo to naredili pri nizih, saj je največja velikost določena že vnaprej. Tip `int` lahko shrani pozitivna in negativna števila velikosti največ 2 milijardi.

2. Računske operacije

2.1. Seštevanje, odštevanje in množenje

Najpreprostejše računske operacije na številih, ki jih lahko z računalnikom izračunamo, so seštevanje, odštevanje in množenje. Račune zapisujemo tako kot v šoli, z *operatorji*. Za seštevanje uporabimo `+`, za odštevanje `-` in za množenje `*`. Poglejmo si preprost primer, kjer račun izvedemo kar v funkciji `printf`.

```
#include <stdio.h>

int main(){
    int a = 5, b = 7;
    printf("%d\n", a+b);
    return 0;
}
```

Zgornji program preprosto izpiše številko 12, ki je rezultat računa $5 + 7$. Namesto izpisovanja lahko rezultat tudi shranimo v spremenljivko:

```
#include <stdio.h>

int main(){
    int a, b, vsota, razlika, produkt;
    scanf("%d%d", &a, &b);
    vsota = a+b;
    razlika = a-b;
    produkt = a*b;
    printf("%d\n%d\n%d\n", vsota, razlika, produkt);
    return 0;
}
```

Če v ta program podamo vhod `3 7`, bo izpisal tri števila: 12, -4 in 21. Drugo izpisano število ima pred seboj minus. Če te to preseneča in še ne poznaš negativnih števil, si preberi naslednji razdelek, sicer pa ga lahko izpustiš.

2.1.1. Negativna števila

Na meteorološki postaji Kredarica so leta 2014 izmerili povprečno januarsko temperaturo približno $-5\,^{\circ}\mathrm{C}$, povprečno avgustovsko pa približno $6\,^{\circ}\mathrm{C}$. Med tema meritvama je $11\,^{\circ}\mathrm{C}$ razlike. Pozimi lahko izmerimo temperature manjše od 0. Takšnim številom, kot je -5, rečemo *negativna števila*. Lahko pa jih uporabimo tudi drugje, ne samo pri merjenju temperature. S pozitivnimi števili lahko štejemo od 0 do neskončno $(1, 2, 3, \ldots)$, z negativnimi pa do negativne neskončnosti $(-1, -2, -3, \ldots)$. Tako kot pozitivna števila jih lahko seštevamo in odštevamo:

$$\begin{gathered} 5 - 11 = -6 \\ -6 + 11 = 5 \\ 5 - (-6) = 11 \\ -2 - 1 = -3 \\ -2 - (-1) = -1 \\ -2 + (-1) = -3 \end{gathered}$$

Pri tem se sklicujemo na naslednji pravili, kjer smo z x označili poljubno število:

$$\begin{gathered} -(-x) = x \\ +(-x) = -x \end{gathered}$$

Z negativnimi števili lahko tudi množimo:

$$\begin{gathered} 2 \cdot (-5) = -10 \\ (-5) \cdot (-5) = 25 \end{gathered}$$

Pri tem uporabimo enostavno pravilo: številski del rezultata je enak, kot če bi množili števili brez predznaka `-`. Če smo množili eno pozitivno in negativno število, rezultatu pripišemo še negativni predznak, če pa smo množili dve negativni števili ali dve pozitivni števili, pa tega ne storimo. Računalnik pri računanju ne razlikuje med pozitivnimi in negativnimi števili, računske operacije pišemo enako kot pri pozitivnih.

2.2. Deljenje

Števila lahko tudi delimo, za kar uporabimo znak `/`. Pri tem pa moramo biti pozorni, ker se deljenje v računalniku obnaša drugače kot smo navajeni iz matematike. Običajno pri deljenju dveh števil, recimo 3 in 7, dobimo ulomek:

$$3/7 = \frac{3}{7}.$$

Če pa v C++ program zapišemo `printf("%d\n", 3/7)`, bomo dobili nepričakovan odgovor — 0. To je zato, ker je deljenje v C++ *celoštevilsko*.

Za lažje razumevanje si bomo pomagali s formulo $a = k \cdot b + o$, ki ponazarja deljenje z ostankom. Če želimo število a deliti s številom b, bomo kot odgovor dobili dve števili: celi del, ki smo ga uspešno delili, in ostanek, kjer deljenje ni bilo uspešno. Celi del smo zgoraj označili s k, ostanek pa z o. Ko uporabimo operator `/`, dobimo največje tako število, za katero je produkt rezultata in delitelja (števila na desni strani `/`) manjši ali enak deljencu (številu na levi strani `/`). Z drugimi besedami, dobimo k iz zgornjega zapisa. Če želimo preveriti še vrednost ostanka o, jo lahko izračunamo iz ostalih števil,

$$o = a - k \cdot b,$$

ali pa uporabimo poseben operator `%`, ki mu pravimo *modulo*. Poglejmo si, kako deljenje deluje v praksi.

```c
#include <stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    int k = a / b, o = a % b;
    printf("rezultat deljenja je %d, z ostankom %d\n", k, o);
    printf("%d = %d * %d + %d\n", a, k, b, o);
    return 0;
}
```

Če v zgornji program vpišemo dve števili, bo izpisal rezultat deljenja ter razcep števila po zgornji formuli. Na primer, ob vhodu `25 7` dobimo naslednji izpis:

```
rezultat deljenja je 3, z ostankom 4
25 = 3 * 7 + 4
```

če pa vpišemo `3 7`, dobimo

```
rezultat deljenja je 0, z ostankom 3
3 = 0 * 7 + 3
```

Kaj pa se zgodi, če vpišemo `5 0`? V tem primeru bo program poskusil deliti z 0, kar v matematiki (in v programiranju) ni dovoljeno. Program se bo zato sesul in ne bo izpisal ničesar, odvisno od našega operacijskega sistema pa morda dobimo kakšno sporočilo o napaki. Ko delimo ali računamo ostanek z modulom, in je število na desni strani operatorja spremenljivka, katere vrednosti ne poznamo vnaprej, moramo preveriti, če je slučajno enaka 0. Če je, operacije ne smemo izvesti!

3. Pogojni stavki

3.1. Osnovna struktura pogojnega stavka

Pogosto želimo, da računalnik izvaja drugačno kodo glede na vrednost ene ali več spremenljivk, npr. da nam pokaže drugačno vsebino, če smo napisali pravilno ali napačno geslo, da računalo sešteva, če smo pritisnili gumb za seštevanje, oz. odšteva, če smo pritisnili gumb za odštevanje. Z drugimi besedami, želimo upravljati potek programa (torej izbrati, katera koda naj se izvede) glede na vrednosti spremenljivk. Angleško takemu upravljanju pravimo *control flow*, najpogosteje pa ga izvajamo s t.i. *pogojnimi stavki*. Osnovna struktura je sledeča:

```
1 if (pogoj) {
2     // koda, ki se izvede, ce pogoj velja
3 } else {
4     // koda, ki se izvede, ce pogoj ne velja
5 }
```

Pogoj je nov pojem. Označuje neke vrste račun, katerega rezultat ni število, vendar *logična vrednost*. Tu sta možni vrednosti le dve: pravilno (angl. `true`) in napačno (angl. `false`). Če bo rezultat računa, navedenega v običajnih oklepajih v zgornjem `if` stavku, `true`, se bo izvedla koda znotraj prvih zavitih oklepajev, če pa je rezultat računa `false`, pa se bo izvedla koda v drugih zavitih oklepajih (tistih za besedo `else`). Drugega dela, tj. `else` in oklepaje za njim, ni treba pisati, če tega ne želimo.

Kako pa zapišemo pogoj? Za to uporabimo posebne *logične operatorje*. Pri delu s številkami so nam na voljo naslednji:

- `==`: primerja dve številski vrednosti. Rezultat je `true`, če sta vrednosti enaki.
- `!=`: primerja dve številski vrednosti. Rezultat je `true`, če sta vrednosti različni.
- `<`: primerja dve številski vrednosti. Rezultat je `true`, če je vrednost na levi manjša od vrednosti na desni.
- `>`: deluje podobno kot `<`, le da v drugo smer; rezultat je `true`, če je vrednost na desni manjša od vrednosti na levi.
- `<=`: primerja dve številski vrednosti. Rezultat je `true`, če sta vrednosti enaki, ali če je vrednost na levi manjša od vrednosti na desni.

- >= deluje podobno kot <=, le da v drugo smer.

Poglejmo si primer uporabe pogojnega stavka.

```c
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    if (a == b) {
        printf("Stevili sta enaki.\n");
    }
    if (a != b) {
        printf("Stevili sta razlicni.\n");
    }
    if (a < b) {
        printf("Prvo stevilo je manjse od drugega.\n");
    }
    if (a > b) {
        printf("Prvo stevilo je vecje od drugega.\n");
    }
    if (a <= b) {
        printf("Prvo stevilo je manjse ali enako drugemu.\n");
    }
    if (a >= b) {
        printf("Prvo stevilo je vecje ali enako drugemu.\n");
    }
    return 0;
}
```

Program v zgornjem primeru primerja dve števili z vsemi naštetimi operatorji. Če v program vpišemo npr. števili 3 in 7, vstopimo v drugi, tretji in peti pogojni stavek, zaradi česar se izpišejo naslednje vrstice:

```
Stevili sta razlicni.
Prvo stevilo je manjse od drugega.
Prvo stevilo je manjse ali enako drugemu.
```

Če pa v program dvakrat vnesemo število 12 (ali katerokoli drugo število), pa dobimo naslednji izhod:

```
Stevili sta enaki.
Prvo stevilo je manjse ali enako drugemu.
Prvo stevilo je vecje ali enako drugemu.
```

Pozorni moramo biti, da pri primerjavi enakosti dveh števil napišemo dva enačaja, ==. Če zapišemo le en enačaj, kot v matematiki (torej =), se bo program sicer zagnal, vendar ne bo deloval pravilno. Enojnega enačaja nikoli ne uporabljamo v pogoju `if` stavka!

Oglejmo si še primer uporabe stavka `else`. Spodnji program bo uporabnika vprašal za PIN, in mu napisal, če je bil PIN pravilen oziroma napačen.

```c
#include <stdio.h>

int main() {
    // Vprasaj uporabnika za PIN, in preveri, ce je enak 42
    int pin;
    scanf("%d", &pin);
    if (pin == 42) {
        printf("PIN je pravilen!");
    } else {
        printf("PIN je napacen!");
    }
    return 0;
}
```

Pogojne stavke lahko tudi gnezdimo, torej vstavimo enega v drugega. Spodnji program od uporabnika sprejme naročilo v restavraciji, kjer ponujajo dve vrsti hrane; juhe in sendviče. Na voljo sta dve vrsti juhe, in dve vrsti sendvičev. Za izbiro kosila uporabnik prvo izbere med juho in sendvičem, nato pa še okus.

```c
#include <stdio.h>
int main() {
    // Uporabnik naj napise 1, ce zeli juho, in 2, ce zeli sendvic.
    int zelja;
    scanf("%d", &zelja);
    if (zelja == 1) {
        // Uporabnik naj napise 1, ce zeli govejo juho,
        // in 2, ce zeli paradiznikovo.
        scanf("%d", zelja);
        if (zelja == 1) {
            printf("Ena goveja juha. Dober tek!\n");
        } else {
            printf("Ena paradiznikova juha. Dober tek!\n");
        }
    } else {
        // Uporabnik naj napise 1, ce zeli sendvic s sunko,
        // in 2, ce zeli vegeterjanski sendvic.
```

```
18          scanf("%d", &zelja);
19          if (zelja == 1) {
20              printf("En sendvic s sunko. Dober tek!\n");
21          } else {
22              printf("En vegeterjanski sendvic. Dober tek!\n");
23          }
24      }
25      return 0;
26  }
```

Pozorni bodimo na postavitev kode. Običajno kodo znotraj zavitih oklepajev `if` stavka pišemo tako, da je poravnana štiri presledke bolj desno od kode zunaj `if` stavka. V nekaterih programskih jezikih je taka poravnava obvezna, v C/C++ pa ne, vendar nezamaknjena koda že v majhnih programih postane popolnoma nepregledna. Branje in popravljanje kode je veliko lažje, če del kode znotraj zavitih oklepajev zamaknemo za štiri presledke. Za to lahko uporabimo tudi tipko Tab, ki se na tipkovnici nahaja levo od tipke Q. Kode kot spodaj nikoli ne pišemo!

```
1   // TAKO NIKOLI NE PIŠEMO!
2   #include <stdio.h>
3   int main() {
4   if (3 > 2) {
5   printf("Velja\n");
6   } else {
7   printf("Ne velja\n");
8   }
9   return 0;
10  }
```

4. Nizanje pogojev

Pogosto se srečamo s problemi, kjer je za rešitev potrebno upoštevati več kot en pogoj. V takih primerih želimo združiti več pogojnih stavkov tako, da se nek del kode izvede, če velja prvi pogoj, drugi del kode pa, če prvi pogoj ne velja, velja pa drugi pogoj. Z gnezdenjem stavkov lahko to v kodo vključimo na naslednji način:

```
if (prvi pogoj) {
    // koda, ki se izvede, če velja prvi pogoj
} else {
    if (drugi pogoj) {
        // koda, ki se izvede, če prvi pogoj ne velja,
        // velja pa drugi pogoj
    } else {
        // koda, ki se izvede, če ne veljata ne prvi ne drugi pogoj
    }
}
```

V takem primeru nam je na voljo bližnjica `else if`. Zgornja koda deluje popolnoma enako kot spodnja:

```
if (prvi pogoj) {
    // koda, ki se izvede, če velja prvi pogoj
} else if (drugi pogoj) {
    // koda, ki se izvede, če prvi pogoj ne velja,
    // velja pa drugi pogoj
} else {
    // koda, ki se izvede, če ne veljata ne prvi ne drugi pogoj
}
```

Prednost te bližnjice je, da je naša koda krajša in bolj razumljiva. Stavke `else if` lahko tudi verižimo; enemu `if` stavku lahko sledi poljubno mnogo stavkov `else if`. Pri tem bo računalnik pogoje preverjal po vrsti. Pri prvem veljavnem pogoju se bo ustavil in izvedel kodo v pripadajočih zavitih oklepajih, za čimer ne bo več preverjal pogojev, temveč bo izvajanje nadaljeval za zaključkom vseh nanizanih stavkov. Kakor nam v osnovnem `if` stavku ni bilo treba pisati dela z `else`, če ga nismo potrebovali, nam ga tudi pri uporabi `else if` ni treba.

Oglejmo si primer uporabe. Naslednji program prebere število in pove, če je večje, manjše ali enako 0.

```c
#include <stdio.h>

int main() {
    int stevilo;
    scanf("%d", &stevilo);
    if (stevilo < 0) {
        printf("Stevilo je manjse od nic.\n");
    } else if (stevilo == 0) {
        printf("Stevilo je enako 0.\n");
    } else if (stevilo > 0) {
        printf("Stevilo je vecje od 0.\n");
    }
    return 0;
}
```

Naslednji primer prikaže, da se izvede samo koda pri prvem veljavnem pogoju, ne glede na to, koliko pogojev za tem je tudi veljavnih. Program izpiše le eno vrstico besedila — `a je 7`, kljub temu, da velja tudi `a > 1`.

```c
#include <stdio.h>

int main() {
    int a = 7;
    if (a < 3) {
        printf("a je manjsi od 3.\n");
    } else if (a == 7) {
        printf("a je 7.\n");
    } else if (a == 4) {
        printf("a je 4.\n");
    } else if (a > 1) {
        printf("a je vecji od 1.\n");
    } else {
        printf("Nic od nastetega ne velja.\n");
    }
    return 0;
}
```

4.1. Logični vezniki

Osnovni operatorji za primerjavo pogosto niso dovolj, da izrazimo željen pogoj. Če na primer želimo pogledati, ali je neko število med dvema drugima, tega ne moramo narediti samo z eno primerjavo. Pogoje združujemo s t.i. *logičnimi vezniki*, ki jim včasih pravimo tudi *logični operatorji*. Poznamo tri osnovne veznike:

- `and` oz. `&&` združi dva pogoja tako, da združeni pogoj velja samo v primeru, da veljata oba hkrati.
- `or` oz. `||` združi dva pogoja tako, da združeni pogoj velja v primeru, da velja katerikoli od dveh, ali da veljata oba.
- `not` oz. `!` sprejme samo en pogoj. Nov pogoj velja samo takrat, ko originalni pogoj ne velja.

Poglejmo si enostaven primer. Če želimo preveriti, ali je dano število med dvema drugima, uporabimo logični veznik `&&`.

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    if (3 < n && n < 9) {
        printf("n je med 3 in 9.\n");
    }
    return 0;
}
```

V matematiki pogosto napišemo dvojno primerjavo $a < b < c$. Če nekaj podobnega napišemo v C++ program, se bo le-ta sicer zagnal, vendar ne bo deloval pravilno. Kaj pričakujemo, da se zgodi, če v tako primerjavo zapišemo `3 < 2 < 1`? Preveri, kaj se dejansko zgodi!

Pri kombiniranju pogojev bodimo previdni glede pravil prednosti. Zanikanje (veznik `not` oz. `!`) ima namreč prednost pred primerjalnimi vezniki (`<`, `==`, …), ter drugima ločnima veznikoma. Če v takem primeru uporabljamo zanikanje, moramo zanikan izraz postaviti v oklepaje.

Za konec si oglejmo še malo bolj kompleksen primer. Napišimo program, ki preveri, ali je uporabnik vpisal prestopno leto. Leto je prestopno, če je deljivo s 4, razen če je hkrati deljivo s 100. Izjema so leta, deljiva s 400, ki so prestopna kljub temu, da so deljiva s 100.

Kako te pogoje zapišemo v program? Opazimo, da so leta, deljiva s 400, prestopna ne glede na druga pogoja. Če leto ni deljivo s 400, potem mora biti deljivo s 4 in ne sme

biti deljivo s 100. Povedano krajše, leto mora biti deljivo s 400, ali pa s 4 in ne hkrati s 100. Tak pogoj lahko zapišemo z logičnimi vezniki.

```c
#include <stdio.h>

int main() {
    int leto;
    scanf("%d", &leto);
    if (leto % 400 == 0 || (leto % 4 == 0 && !(leto % 100 == 0))) {
        printf("Leto je prestopno.\n");
    } else {
        printf("Leto ni prestopno.\n");
    }
    return 0;
}
```

5. Zanke

5.1. Sintaksa

V programiranju pogosto želimo nek del kode ponoviti, zato uporabljamo *zanke*. Poznamo več zank, a najpogosteje uporabljamo zanko `for`. Le-ta ima tri posebne komponente: *začetek*, *pogoj* in *korak*. Poglejmo si preprost primer, ki trikrat izpiše besedo `nekaj`.

```
1  #include <stdio.h>
2
3  int main() {
4      //   začetek;      pogoj;      korak
5      for (int stevec=0; stevec < 3; stevec++) {
6          // med zavitimi oklepaji {} je koda,
7          // ki se izvede vsako iteracijo zanke
8          printf("nekaj\n");
9      }
10     return 0;
11 }
```

Poglejmo si, kako ta program deluje. Podpičja v peti vrstici razdelijo okrogle oklepaje na tri dele: začetek (`int stevec=0`), pogoj (`stevec < 3`) in korak (`stevec++`). Začetek se bo izvedel, ko se ta zanka začne. Vsakič preden se izvede koda v notranjosti zanke, se preveri pogoj. Če pogoj drži, se bo izvedla koda v zanki, sicer pa se bo zanka končala. Korak je podoben začetku, in se izvede na koncu vsake ponovitve zanke.

V zgornjem primeru začetek naredi novo številko `stevec` in jo nastavi na 0. Pogoj preveri, če je `stevec` manjši od 3, korak `stevec++` pa je okrajšava za `stevec = stevec + 1`, torej poveča `stevec` za 1. Program sledi naslednjem postopku:

1. Program se začne in pride do for zanke, najprej se izvede začetek `int stevec=0`.
2. Zdaj se je začela zanka, preveri se pogoj `stevec < 3`. Ker je `stevec` za zdaj še 0, je pogoj izpolnjen. Izvede se vsebina zanke, torej program izpiše `nekaj`. Zdaj smo prišli do konca zanke, izvede se korak, zato se `stevec` poveča na 1, program pa skoči nazaj na začetek zanke.
3. Ker smo na začetku zanke, se preveri pogoj, `stevec < 3`, `stevec` je zdaj 1 in je pogoj še vedno izpolnjen, zato se izvede vsebina zanke. Ko program še enkrat izpiše `nekaj`, izvede korak, `stevec` poveča na 2 in skoči nazaj na začetek.

4. Spet smo na začetku, zato se preveri pogoj, `stevec` je zdaj 2, kar je manjše od 3, zato se `nekaj` izpiše še tretjič. Program potem poveča `stevec` na 3 in skoči nazaj na začetek.

5. Ker smo spet na začetku, se bo še enkrat preveril pogoj, a zdaj je `stevec` enak 3 in 3 ni manjše od 3, zato se zanka konča. Ker je naslednji ukaz `return 0`, se bo program tam končal.

Vredno je omeniti, da je naš števec zavzel vrednosti 0, 1, in 2, kar se morda zdi čudno, glede na to, da bi ponavadi šteli do tri kot $1, 2, 3$. Tovrstno štetje od nič je zelo pogosto v programiranju, in bo prišlo prav kasneje, ko se bomo učili o seznamih.

5.2. Primeri uporabe

5.2.1. Spreminjanje dolžine zanke

Zanka, ki smo jo napisali zgoraj, se bo vedno ponovila trikrat. Kaj pa če hočemo, da se zanka ponovi glede na neko število na vhodu? Seveda je tudi to mogoče in sicer tako, da vstavimo našo spremenljivko v pogoj zanke. Poglejmo si primer.

```c
#include <stdio.h>

int main() {
    int dolzina;
    scanf("%d", &dolzina);
    for (int stevec=0; stevec < dolzina; stevec++) {
        printf("-");
    }
    printf(">\n");
    return 0;
}
```

Zgornji program sprejme število in nariše puščico te dolžine. Tu uporabimo še en trik, in sicer v funkciji `printf` znotraj zanke ne dodamo `\n`, s čimer dosežemo to, da so v izhodu znaki `-` eden zraven drugega v isti vrstici in ne vsak v svoji.

5.2.2. Branje števil v zanki

Ena od moči računalnikov je zelo hitra obdelava velike količine podatkov, računalnik bo na primer zlahka seštel tisoč števil, medtem ko bi bilo to početi na roke precej zamudno. Poglejmo si, kako bi napisali program, ki bi nekaj izračunal z več števili.

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    int vsota = 0;
    for (int i=0; i < n; i++) {
        int sestevanec;
        scanf("%d", &sestevanec);
        vsota += sestevanec;
    }
    printf("%d\n", vsota);
    return 0;
}
```

V zgornjem primeru prvo preberemo število `n`, nato pa ustvarimo spremenljivko `vsota`, ki jo takoj nastavimo na 0. V tej spremenljivki bomo hranili vsoto števil, ki smo jih do sedaj videli na vhodu (razen `n`), na koncu programa bo torej enaka vsoti vseh takih števil.

Opazimo, da v zanki namesto `stevec` sedaj uporabljamo `i`, ki je tradicionalna izbira za spremenljivko v zanki. V notranjosti zanke so zdaj trije ukazi. Najprej naredimo novo spremenljivko, ki jo poimenujemo `sestevanec` in preberemo naslednje število iz vhoda. Nato pa z okrajšavo `vsota += sestevanec` prištejemo spremenljivki `vsota` spremenljivko `sestevanec`. Na daljše bi to lahko napisali kot `vsota = vsota + sestevanec`. V vsaki iteraciji tako prištejemo ravno prebrano število k vsoti, na koncu pa bomo izpisali vsoto vseh.

5.2.3. Zanka z drugačnim korakom in začetkom

Do zdaj so vse naše zanke takole: `for (int i=0; i < 10; i++)`, torej so začele z nič in se nekajkrat ponovile. C++ pa nam dovoli, da lahko z našimi zankami naredimo veliko več. Kot primer si poglejmo zanko, ki izpiše vsa soda števila med 1 in 100.

Pozorno poglejmo števila, ki jih moramo izpisati. Ker 1 ni sodo, bo prvo izpisano število 2. Število 3 prav tako ni sodo, tako da bomo izpisali 4, po tem pa 6, 8, 10 in tako dalje. Vidimo, da vsak korak povečamo izpisano število za 2, napišimo torej program.

```c
#include <stdio.h>

int main() {
    for (int i=2; i<=100; i += 2) {
        printf("%d\n", i);
    }
```

```
7       return 0;
8   }
```

Ker se želena števila začnejo z dva, bomo v začetni del zanke vpisali `int i=2`. Ker želimo izpisati števila med 1 in 100 in ne med 1 in 99, bomo v pogojnem delu uporabili znak manjše ali enako, `i<=100` (pogoj `i<100` ne bi veljal za število 100). Ker želimo povečati naše število za 2 vsak korak, smo v polje za korak napisali `i += 2`. Edina stvar, ki jo naredimo v notranjosti napisane zanke pa je, da izpišemo trenutno vrednost spremenljivke `i`.

6. Seznami

6.1. Sintaksa

Ko si želimo podatke shraniti tako, da jih bomo lahko spreminjali in na koncu nekaj z njimi naredili (npr. da z njimi računamo, jih izpišemo, itd.), za to uporabimo spremenljivke. Vsaka spremenljivka hrani en podatek – vse spremenljivke do sedaj so hranile le eno številko. Pogosto pa si želimo števila shraniti tako, da bomo kasneje lahko dostopali do njih, ampak med pisanjem programa ne vemo točno, s koliko števili bo program moral delati. Problem rešimo s seznami.

Da ustvarimo seznam (angl. *array*), zapišemo tip spremenljivke, ki ga bodo imeli vsi elementi seznama (trenutno poznamo le `int`, a bomo kmalu spoznali tudi druge tipe spremenljivk), nato seznamu damo ime, in na koncu v oglatih oklepajih zapišemo dolžino seznama, takole:

```
1  int seznam_stevil[300];
```

Tukaj smo ustvarili seznam z imenom `seznam_stevil`, ki hrani 300 števil. Če želimo dostopati do elementov seznama, ali nastaviti njihove vrednosti, tudi uporabimo oglate oklepaje, kot v spodnjem primeru.

```
1  #include <stdio.h>
2
3  int seznam_stevil[300];
4
5  int main() {
6      seznam_stevil[3] = 7;
7      seznam_stevil[4] = 9;
8      seznam_stevil[5] = seznam_stevil[3] + seznam_stevil[4];
9      printf("%d\n", seznam_stevil[5]);  // izpiše 16
10     return 0;
11 }
```

Tu smo prvo nastavili tretji element seznama na 7, nato smo nastavili četrti element seznama na 9, in za tem nastavili peti element seznama na njuno vsoto. Ostalih elementov seznama se nismo dotaknili. Tako kot pri spremenljivkah nismo smeli uporabiti

vrednosti spremenljivke, preden smo ji vrednost nastavili, moramo paziti, da vrednosti posamičnega elementa seznama ne uporabljamo, preden je ne nastavimo. Narobe bi bilo na primer izpisati `seznam_stevil[2]` ali to vrednost uporabiti v računu, saj je nismo nikoli nastavili.

Pri dostopanju do elementov seznama moramo paziti tudi na naslednje dejstvo: če imamo seznam dolžine N, potem so elementi tega seznama oštevilčeni s številkami od 0 do $N-1$ vključno, ne pa s številkami od 1 do N, kot bi morda pričakovali. V zgornjem primeru tako lahko zapišemo `seznam_stevil[0]`, `seznam_stevil[1]`, ..., `seznam_stevil[299]`, ne pa tudi `seznam_stevil[300]`. Pravimo, da so seznami *indeksirani* od 0 naprej.

Poglejmo si primer preproste naloge. Na vhodu je podano število N, ki mu sledi N števil. Program mora izpisati ta števila (razen prvega, N), v obratnem vrstnem redu. V ta namen ustvarimo seznam z imenom `seznam`, ki lahko shrani največ 1000 (predpostavimo, da uporabnik ne bo vnesel več kot 1000 števil). Potem s `for` zanko preberemo N števil in vsako shranimo v svoj element seznama, začenši z 0. Na koncu se s še eno `for` zanko sprehodimo skozi $i = 0, 1, \ldots, N-1$. Na vsakem koraku izračunamo indeks (položaj) elementa, ki je i-ti od konca seznama, in ga shranimo v spremenljivko `obratni`. Ta indeks je natanko `N-i-1`, saj mora za `i=0` biti `obratni=N-1`, za `i=1` mora biti `obratni=N-2`, itd. Da se seznam na izhodu izpiše v obratnem vrstnem redu, preprosto izpišemo `seznam[obratni]`.

```c
#include <stdio.h>

int seznam[1000];

int main() {
    int N;
    scanf("%d", &N);

    for (int i = 0; i < N; i++) {
        scanf("%d", &seznam[i]);
    }

    for (int i = 0; i < N; i++) {
        int obratni = N-i-1;
        printf("%d ", seznam[obratni]);
    }
    printf("\n");

    return 0;
}
```

Poglejmo si še malo bolj zanimiv primer. Naša naloga sedaj je, da uredimo seznam N števil ($N \leq 10^6$) po velikosti od najmanjšega do največjega. Podano imamo tudi informacijo, da bodo ta števila velika med vključno 0 in 100. Naloge se lahko lotimo tako, da preštejemo, kolikokrat se neko število pojavi v danem seznamu, nato pa bomo seznam rekonstruirali tako, da bo urejen. Da preštejemo, kolikokrat se kakšno število pojavi, uporabimo nov seznam, kjer indeks pomeni številko, ki jo štejemo, shranjena vrednost pa kolikokrat smo to številko že prešteli.

```c
#include <stdio.h>

// seznam iz vhoda
int seznam[1000003];
// na indeksu j je zapisano, kolikokrat smo že videli
// število j v seznamu
int stetje[101];
// nov, urejen seznam
int novseznam[1000003];

int main() {
    int N;  // velikost seznama
    scanf("%d", &N);
    for (int i = 0; i < N; i++)
        scanf("%d", &seznam[i]);

    // število na mestu i v seznamu je seznam[i]
    // v eni iteraciji zanke vidimo eno število; zato prištejemo
    // 1 na pravo mesto seznama za štetje
    for (int i = 0; i < N; i++)
        stetje[ seznam[i] ] += 1;

    // sedaj rekonstruiramo seznam
    // shranjujemo si indeks prvega elementa v novem seznamu,
    // ki ga še nismo nastavili
    int indeks = 0;
    for (int j = 0; j <= 100; j++) {
        // stetje[j]-krat moramo zapisati j v nov seznam
        for (int k = 0; k < stetje[j]; k++) {
            novseznam[indeks] = j;
            // povečamo indeks, ker smo ga ravno nastavili
            indeks++;
        }
    }
    // izpišemo nov seznam
```

```
36      for (int i = 0; i < N; i++)
37          printf("%d ", novseznam[i]);
38
39      printf("\n");
40      return 0;
41  }
```

Ta algoritem za urejanje je zelo znan; imenuje se urejanje s preštevanjem (angl. *counting sort*). Primeren je, kadar imamo zelo majhen razpon možnih vrednosti števil, kakor smo imeli tu $(0-100)$. Obstajajo tudi drugi algoritmi za urejanje. Nekatere bomo spoznali kasneje.

6.1.1. Večdimenzionalni seznami

Videli smo, kako ustvariti seznam števil, kaj pa seznam seznamov? Takemu seznamu pravimo *dvodimenzionalen seznam*, ustvarimo pa ga tako, da napišemo dva zaporedna oglata oklepaja z velikostjo, kot spodaj:

```
1  int seznam_seznamov_stevil[velikost1][velikost2];
```

Dvodimenzionalni seznami so uporabni, kadar moramo podatke predstaviti v tabeli. Do posamičnih elementov dostopamo z dvojnimi oglatimi oklepaji, tako kot pri inicializaciji spremenljivke; `tabela[i][j]`. Tabele si običajno predstavljamo tako, da nam prvi indeks poda zaporedno številko vrstice, drugi pa zaporedno številko stolpca. Sicer pa z njimi delamo enako kot z običajnimi seznami.

7. Nizi in besedilo

Poleg dela s številkami od računalnika pogosto želimo, da nekaj naredi z nizi besedila. Primeri takšnih programov so npr. urejevalniki besedila, ki jih uporabljamo tako za pisanje „enostavnega“ besedila (kode), kot tudi za razna obogatena besedila. Pravzaprav pa skoraj vsak računalniški program dela z besedilom; kadarkoli želimo uporabniku prikazati neke informacije, jih moramo namreč izpisati na zaslon. Ko smo delali s številkami, smo problem izpisovanja prepustili računalniku, ker je kodo za branje in izpisovanje številk k sreči napisal že nekdo drug. Za pisanje splošnih programov pa tovrstno znanje ne bo dovolj, zato si poglejmo osnove dela z besedili.

Pri slovenščini se naučimo, da je besedilo sestavljeno iz več odstavkov, odstavek iz več povedi, poved iz več stavkov, stavek iz več besed, besede pa iz več črk. Pri tem se moramo zavedati, da stavke ločimo z ločili (vejice, pike, klicaji, itd.), besede ločimo s presledki, posamične odstavke pa ločimo z zamikanjem prve povedi v desno. Za predstavitev v računalniku je tak model preveč zakompliciran, zato vzamemo bolj enostavnega. Besedila bomo predstavili z *nizi* (angl. *string*), ki bodo zaporedja več *znakov* (angl. *character*). Vse, kar bi si kadarkoli zaželeli izpisati, bomo proglasili za znak. Tako si bomo vse črke predstavljali kot znak, kjer bomo ločili tudi med velikimi in malimi črkami (saj vendar izgledajo drugače, če jih napišemo), prav tako bomo za znake proglasili tudi ločila, oklepaje in matematične operacije (`+, -, *, /`). Poleg tega bomo za znak proglasili tudi števke od 0 do 9, ker tudi njih izpišemo (večje številke pa so sestavljene iz teh števk, zato ne potrebujemo posebnih znakov za njih).

Nenazadnje bomo ustvarili še nekaj posebnih znakov, ki jih morda nebi pričakovali. Od teh bomo zdaj spoznali tri: znak za presledek, znak za novo vrstico in znak za konec besedila. Znak za presledek bomo uporabili, kjerkoli želimo imeti prostor med dvema besedama (torej presledek). Za razliko od slovenščine tudi presledke obravnavamo, kot da bi bili pravzaprav neke posebne črke. Znak za novo vrstico bomo uporabili tam, kjer želimo, da izpis našega programa skoči vrstico nižje; brez tega znaka nam bo program vse izpisal v eni zelo dolgi vrstici besedila. Ta znak označimo s posebno kodo `\n` (ker se v angleščini ta znak imenuje *new line*), opazimo pa, da ga v funkciji `printf` uporabljamo že od prvega programa, ki smo ga napisali. Uporabili bomo tudi znak za konec besedila, ki ga označimo z `\0`, pogosto pa mu rečemo tudi *NULL*. Več o temu znaku bomo povedali kasneje.

7.1. Predstavitev znakov

Preden si pogledamo nize, moramo razumeti, kako delamo z znaki. V C++-u imamo za to poseben tip `char`, ki nam hrani en znak. Če želimo spremenljivki tipa `char` nastaviti vrednost, moramo želeni znak dati v enojne narekovaje, kot spodaj:

```
1 char crka = 'A';
```

S to kodo ustvarimo spremenljivko tipa `char`, ki hrani vrednost `'A'`, torej znak za veliko črko A. Tako kot števila lahko tudi znake pišemo in beremo; pri tem uporabimo `printf` s formatnikom `%c`, narejen za znake.

```
1 char crka;
2 scanf("%c", &crka);
3 printf("Tvoja crka: %c\n", crka);
```

Znak	ASCII koda
NULL (`\0`)	0
Nova vrstica (`\n`)	10
Presledek (`' '`)	32
`0`	48
`1`	49
`2`	50
⋮	⋮
`9`	57
`A`	65
`B`	66
`C`	67
⋮	⋮
`Z`	90
`a`	97
`b`	98
`c`	99
⋮	⋮
`z`	122

Tabela 7.1.: Del ASCII tabele

Ker so računalniki narejeni za delo s številkami, moramo tudi znake predstaviti kot številke. To dosežemo s t.i. *kodnimi tabelami*, ki vsakemu znaku priredijo eno številko. Najpreprostejša kodna tabela je ASCII, ki lahko zakodira vse črke angleške abecede ter vse ostale zgoraj naštete znake, ne zmore pa zakodirati šumnikov ali večine črk, ki se ne pojavljajo v angleški abecedi. Prav zaradi tega razloga se pri programiranju takih črk izogibamo, kar se le da. ASCII kode nekaterih pogostih znakov so prikazane v tabeli 7.1.

Opazimo lahko, da so števke in črke v tabeli zaporedno; števka `0` ima kodo 48, števka `1` 49, …, črka `A` ima kodo 65, `B` ima kodo 66, itd. Opazimo tudi, da so velike črke od malih ločene, in da imajo male črke večje kode.

Ta dejstva lahko uporabimo v programih tako, da črke preprosto obravnavamo, kot da bi bile številke. Črki `'a'` lahko npr. prištejemo neko številko, in tako dobimo črko, ki je toliko znakov naprej v abecedi; `'a' + 7` je na primer enako `'h'`. Poleg tega lahko znake med sabo primerjamo, kar bomo videli v prvem primeru.

Poglejmo si spodnji primer, kjer je prikazano, kako na enostaven način preverimo, ali je neka črka velika ali majhna. Koda sprva prebere eno črko iz vhoda, nato pa preveri, če je vpisana črka med `A` in `Z`; če ni, potem preverimo še, ali je črka med `a` in `z`.

```c
#include <stdio.h>

int main() {
    char crka;
    scanf("%c", &crka);
    if (crka >= 'A' && crka <= 'Z') {
        printf("To je velika crka\n");
    } else if (crka >= 'a' && crka <= 'z') {
        printf("To je majhna crka\n");
    } else {
        printf("To sploh ni crka!\n");
    }
    return 0;
}
```

Če dobro pogledamo v tabelo, vidimo, da koda 0 ne pripada števki `0`, pač pa znaku za konec besedila. To se morda na prvi pogled zdi nepričakovano, ampak ima svoj smisel; če je števka del besedila, o njej ne razmišljamo kot o številki, temveč pač o nekem znaku, ki ima v drugem kontekstu drugačen pomen. Če želimo to števko pretvoriti v številko, s katero lahko brez skrbi računamo, lahko uporabimo trik, kjer „odštejemo nič“, kot spodaj:

```c
char znak = '7';  // števka 7, napisana kot znak
int stevilka = znak - '0';  // če odštejemo nič, nič ne spremenimo

// NAROBE!
int to_pride_55 = znak - 0;
```

Pri tem triku moramo biti previdni, da odštejemo pravilno ničlo; če odštejemo številko 0, se ne bo nič spremenilo; tako kot pri matematiki namreč odštevanje ničle številke ne

spremeni. Če pa odštejemo *znak* `'0'` (v enojnih narekovajih), pa dejansko odštevamo številko 48, tj. ASCII kodo znaka `'0'`. Praktično uporabo tega trika bomo pokazali v naslednjem razdelku.

7.2. Predstavitev nizov

Niz predstavimo kot seznam znakov, ki ga pišemo podobno kot seznam številk:

```
1 char niz_besedila[300];
```

Ta ukaz pove računalniku, naj ustvari spremenljivko z imenom `niz_besedila`, ki hrani *največ* 300 znakov. V tej spremenljivki bomo hranili naše zaporedje besedila. Če želimo nize brati ali pisati, uporabimo funkciji, ki ju že poznamo, ter formatnik `%s`, tu pa je ena posebnost; za branje nizov pred imenom spremenljivke ne zapišemo znaka `&`, kakor to zapišemo za branje številk ali znakov. Če želimo neko spremenljivko nastaviti na niz, ki ga ne bomo prebrali, jo lahko nastavimo na običajen način z enačajem, ter z dvojnimi narekovaji. Tak način podajanja nizov smo že srečali; namreč vedno, ko uporabimo funkciji `scanf` ali `printf`.

Poglejmo si preprost primer uporabe. Spodnji program prebere uporabnikovo ime in ga pozdravi.

```
1 #include <stdio.h>
2
3 int main() {
4     char uporabnikovo_ime[300];
5     scanf("%s", uporabnikovo_ime);  // NE napišemo znaka &
6     printf("Zivjo %s!\n", uporabnikovo_ime);
7     return 0;
8 }
```

Ko ustvarimo niz, računalniku povemo, kolikšna je njegova najdaljša možna dolžina. Nič pa nam ne preprečuje, da v to spremenljivko shranimo krajši niz. Kako pa potem računalnik ve, kje se naš niz dejansko konča? Pričakujemo namreč, da bomo za zapis kratkega niza uporabili nekaj mest za znake na začetku, potem pa se bo niz nekje končal; kaj je na neuporabljenih mestih zaporedja, nas ne zanima. Ravno iz tega razloga so nizi zgrajeni tako, da imajo na koncu dodaten znak, ki označuje konec besedila. To je znak NULL, ki smo ga omenili na začetku. Ta znak pove računalniku, da se besedilo tu konča in da naj naprej ne gleda. Če vrednost niza nastavimo z enačajem ali niz preberemo s `scanf`, bo računalnik sam poskrbel, da bo ta znak napisan na pravo mesto; če pa z nizi delamo kaj bolj zapletenega, moramo za ta znak skrbeti sami. Zaradi tega znaka je dejansko število vidnih znakov, ki jih lahko shranimo v niz, za eno manjše od predpisane

največje dolžine. Da se tovrstnim problemom izognemo, bomo od sedaj naprej vedno napisali nekaj večjo število za dolžino niza; če pričakujemo, da uporabnik vpiše največ 200 znakov, bomo za velikost niza dejansko napisali 201 (ali celo malo več).

Kot primer, kako uporabimo zgornje dejstvo, si poglejmo spodnji program, ki izračuna dolžino niza.

```c
#include <stdio.h>

int main() {
    char niz[301];
    // uporabnik lahko napise niz dolžine najvec 300
    scanf("%s", niz);

    // da poiščemo dolžino, bomo dejansko poiskali indeks
    // znaka NULL; s tem bomo dobili točno število znakov pred njim
    int i = 0;  // trenutni indeks v nizu
    while (niz[i] != 0) {  // ASCII koda znaka NULL je 0
        i++;
    }
    // na kocnu je i ravno indeks znaka NULL, in s tem enak
    // dolzini niza
    printf("Niz je dolg %d znakov\n", i);
    return 0;
}
```

Ta koda ni težka, vendar jo je pogosto neprijetno pisati, zato imamo boljšo alternativo; če na začetek programa dodamo `#include <string.h>`, lahko uporabljamo funkcijo `strlen`, ki nam ravno tako izračuna dolžino niza:

```c
#include <stdio.h>
#include <string.h>  // strlen

int main() {
    char niz[201];
    scanf("%s", niz);
    printf("Dolzina niza: %d\n", strlen(niz));
    return 0;
}
```

Funkcijo `strlen` uporabljamo v skoraj vsakem programu z nizi, zato je dobro, da se je čim prej navadimo. Poleg tega `strlen` dolžino dejansko izračuna hitreje kakor zgornja

koda.

V naslednjem primeru bomo napisali kodo, ki pretvori besedilo v velike črke. Za to uporabimo eno od lastnosti ASCII tabele, ki smo jo omenili prej; namreč, da so črke napisane zaporedno, in da so velike črke pred malimi.

```c
#include <string.h>
#include <stdio.h>

int main() {
    char niz[201];
    scanf("%s", niz);

    // Zamik med velikimi in majhnimi črkami je enak ne glede na to,
    // katera črka je to. V zanki bomo vsaki majhni črki odšteli
    // zamik, s čimer jo pretvorimo v veliko
    int zamik = 'a' - 'A';
    int dolzina = strlen(niz);
    for (int i = 0; i < dolzina; i++) {
        // če je črka majhna, jo moramo povečati
        // to moramo nujno preveriti, saj drugih znakov ne smemo
        // spremeniti!
        if ('a' <= niz[i] && niz[i] <= 'z') {
            niz[i] = niz[i] - zamik;
        }
    }
    printf("%s\n", niz);
    return 0;
}
```

Za zadnji primer v tem razdelku si poglejmo, kako bi pretvorili številko, zapisano z nizom, v številko, zapisano v spremenljivki tipa `int`. Prej omenjen trik z odštevanjem nič ne bo deloval, ker lahko z njim pretvarjamo le znake; lahko pa številko pretvorimo znak po znak. Pri tem pretvarjanju uporabljamo lastnosti desetiškega zapisa števil; namreč, da zaporedna mesta v zapisu predstavljajo vrednosti, ki se razlikujejo za faktor 10. Ko pretvorimo prvi del besedila, in želimo dopisati še eno številko, moramo že zapisani del „premakniti“ eno mesto v levo, ter premaknjenemu številu prišteti novo številko. Premikanje dosežemo z množenjem z 10.

```c
#include <string.h>
#include <stdio.h>

int main() {
```

```
    char niz[11];  // niz, ki bo hranil številko
    // premisli, zakaj je dolžina 11 že dovolj
    scanf("%s", niz);
    int dolzina = strlen(niz);
    int stevilo = 0;  // začnemo z 0
    for (int i = 0; i < dolzina; i++) {
        stevilo *= 10;
        stevilo += (niz[i] - '0');
        // niz[i] je znak, ki ga moramo pretvoriti v številko, da lahko
        // računamo z njim
    }
    printf("Stevilka je %d\n", stevilo);
    return 0;
}
```

7.3. Standardne funkcije

Izkaže se, da pri delu z nizi pogosto pišemo zelo podobne kose programa, kakor se je zgodilo pri primeru z izračunom dolžine. Namesto da večkrat napišemo skoraj enako kodo, so v knjižnici `string.h` dostopne razne funkcije, ki nam pogosto olajšajo delo.

7.3.1. Primerjava nizov

Za primerjavo dveh nizov ne uporabljamo dvojnega enačaja (==), temveč funkcijo `strcmp`. Funkcijo uporabimo tako, da ji v okrogle oklepaje napišemo dva niza; `strcmp(niz1, niz2)`. Če sta niza enaka, funkcija vrne rezultat `0`, sicer pa vrne drugačen rezultat. Spodnja koda preveri, če je uporabniku ime Filip:

```
char niz[101];
scanf("%s", niz);
if (strcmp(niz, "Filip") == 0) {
    printf("Ime ti je Filip\n");
} else {
    printf("Ni ti ime Filip\n");
}
```

Funkcija nam pravzaprav poda več informacij. Z njo lahko pogledamo, kakšna je *leksikografska ureditev* dveh nizov; preprosto povedano, kateri od nizov bi se, če bi bila oba niza besedi, pojavil prej v slovarju (leksikonu). Če bi se prvi niz pojavil prej, funkcija vrne negativno število. Če bi se drugi niz pojavil prej, pa funkcija vrne pozitivno število.

7.3.2. Kopiranje nizov

Če želimo eno spremenljivko prekopirati v drugo, lahko napišemo `b = a`. Na žalost pa to ne deluje za nize; namesto `niz2 = niz1` moramo napisati `strcpy(niz2, niz1)`. Funkcija `strcpy` prekopira drugi niz v prvega; na koncu bosta oba niza imela enako vsebino.

Če želimo nekemu nizu na konec dodati nek drug niz, lahko za to uporabimo funkcijo `strcat` (*string concatenate*). Ta funkcija prav tako sprejme dva niza; ko jo pokličemo, drug niz kopira na konec prvega.

7.3.3. Operacije na prvih m znakih

Včasih želimo kopirati ali primerjati le del niza. Za to imamo na voljo malce drugačne verzije zgoraj naštetih funkcij; če v imenih teh funkcij za `str` dodamo še `n` (torej `strncmp`, `strncpy`, …), in funkciji kot zadnji argument podamo številko m, bo funkcija svoje delo opravila le na prvih m znakih; `strncmp(niz1, niz2, 3)` bo primerjal le prve tri znake, `strncpy(niz2, niz1, 7)` bo kopiral le prvih sedem znakov, ipd.

8. Funkcije

8.1. Osnovna sintaksa

Ko pišemo daljše programe, se pogosto zgodi, da večkrat uporabimo nek podoben kos kode. Da si v takih situacijah olajšamo življenje, lahko uporabimo *funkcije*. Funkciji podamo nekaj *parametrov* (včasih se uporabi tudi beseda *argumentov*), ona nekaj melje in nam potem *vrne* neko vrednost. Poleg tega, da samo vrne neko vrednost, ima lahko funkcija tudi kakšne stranske učinke. Lahko na primer kaj izpiše z uporabo `printf`, ali pa spremeni vrednost neke spremenljivke. Včasih so parametri ali vrnjena vrednost nepotrebni, zato lahko napišemo tudi funkcije brez parametrov ali funkcije, ki ne vrnejo ničesar.

Najbolj osnovna oblika funkcije je funkcija, ki ne sprejme nobenega parametra in ne vrne ničesar. Če hočemo, da ta funkcija ni neuporabna, bo morala imeti kakšen stranski učinek. V spodnjem primeru funkcija izpiše besedilo.

```c
#include <stdio.h>

void moja_funkcija() {
    // koda, ki se bo izvedla, ko pokličemo funkcijo
    printf("Zdravo\n");
}

int main() {
    moja_funkcija(); // pokličemo funkcijo
    return 0;
}
```

Poglejmo si malo bolj podrobno, kaj posamični deli zgornjega programa naredijo. Pri definiciji funkcije smo prvo napisali `void` — to je posebna oznaka, ki pomeni, da funkcija ne vrne ničesar. Za tem je beseda `moja_funkcija`, ki označuje ime funkcije, ki jo definiramo. Tako kot spremenljivke lahko funkcije poimenujemo, kakor želimo. Za imenom funkcije je običajen oklepaj, ki zaznamuje začetek seznama argumentov. V tem primeru argumentov nimamo, zato seznam takoj končamo z zaklepajem. Na koncu še v zavite oklepaje postavimo kodo, ki jo bo funkcija izvedla.

Če želimo funkciji podati argumente, jih zapišemo kot spremenljivke med oklepaje v definiciji funkcije:

```c
#include <stdio.h>

void plusi(int n) {
    // izpišemo n plusov
    for (int i = 0; i < n; i++) {
        printf("+");
    }
    printf("\n"); // končamo z znakom za novo vrstico
}

int main() {
    plusi(12); // pokličemo funkcijo s parametrom 12
    return 0;
}
```

Zgoraj smo v oklepaje za imenom spremenljivke postavili `int n`. Beseda `int` pove, da je ta parameter celo število, `n` pa je ime za parameter. Ime, ki smo si ga izbrali za parameter, uporabljamo v telesu funkcije, da dostopamo do vrednosti v parametru.

Če je naš parameter seznam, to napišemo kot spodaj.

```c
#include <stdio.h>

void izpisi_prve_tri(int seznam[]) {
    printf("%d %d %d\n", seznam[0], seznam[1], seznam[2]);
}

int main() {
    int stevila[] = {7, 8, 6, 1, 2, 6, 3};
    izpisi_prve_tri(stevila); // izpiše "7 8 6"
    return 0;
}
```

Da funkcija vrne neko vrednost, moramo namesto `void` napisati tip vrednosti, ki jo vrne, nato pa nekje v notranjosti funkcije uporabiti `return`. Poglejmo si enostaven primer, kjer s funkcijo izračunamo vsoto dveh števil.

```c
#include <stdio.h>

int vsota(int a, int b) {
    return a + b;
```

```c
}

int main() {
    int rezultat = vsota(3, 4);
    printf("%d\n", rezultat);
    return 0;
}
```

Kot smo navajeni, se ukazi v telesu funkcije izvajajo po vrsti od zgoraj navzdol. V njej lahko uporabimo vse, kar smo se do zdaj naučili (lahko ustvarjamo nove spremenljivke, uporabimo pogojne stavke, zanke, …). Vredno omembe je, da izvajanje funkcije konča takoj, ko se izvede prvi `return`, tudi če je za tem še nekaj kode.

V spodnjem bloku kode funkcije vrne logično vrednost `bool`, ki je ena od `true` ali `false`. Funkcija deluje pravilno, saj če bo `crka` enaka `'a'`, `'e'`, `'i'`, `'o'` ali `'u'`, bo funkcija končala znotraj pogojnega stavka, sicer pa bo nadaljevala do `return false` na koncu.

```c
#include <stdio.h>

bool je_samoglasnik(char crka) {
    if (crka == 'a' || crka == 'e'
        || crka == 'i' || crka == 'o'
        || crka == 'u')
    {
        return true;
    }
    return false;
}

int main() {
    char c;
    scanf("%c", &c); // prebere eno črko

    if (je_samoglasnik(c)) {
        printf("To je samoglasnik!\n");
    } else {
        printf("To pa ni samoglasnik.\n");
    }
    return 0;
}
```

8.2. Spremenljivke in funkcije

Spomnimo se, da lahko spremenljivke definiramo na dva načina. Tistim, ki so definirane na vrhu, izven funkcij, pravimo *globalne spremenljivke*, tistim, ki so pa definirane v neki funkciji, pa *lokalne spremenljivke*. Globalne spremenljivke so vidne povsod, torej v vseh funkcijah, medtem ko lahko lokalne spremenljivke uporabljamo samo v funkciji, v kateri smo jih definirali.

```
#include <stdio.h>

// tu so globalne spremenljivke
int g = 12;

void funkcija() {
    // v tej funkciji lahko dostopamo do g
    printf("%d\n", g);
    g += 3;
}

int main() {
    int n = 200;
    funkcija();
    // tu lahko dostopamo do g in n
    printf("%d, %d\n", g, n);
    return 0;
}
```

Seveda, če globalno spremenljivko nekje spremenimo, se bo spremenila tudi za vse druge funkcije. V zgornjem primeru `funkcija` najprej izpiše `g` (12) in ga nato poveča na 15. Ko se `g` izpiše drugič, je zato enak 15.

Če funkciji podamo neko spremenljivko kot parameter in poskušamo to spremenljivko v funkciji spreminjati, se obnaša na dva različna načina, odvisno od tega, ali je spremenljivka seznam ali ne.

Če spremenljivka ni seznam (npr. `int` ali `char`) se za funkcijo ustvari kopija te spremenljivke. Če jo v funkciji spremenimo, to ne bo spremenilo izvirne spremenljivke, saj v funkciji delamo s kopijo. Če pa argument je seznam, bodo spremembe na seznam v funkciji vplivale na izvirni seznam, saj se ta ne bo prekopiral. V spodnjem primeru sta prikazani obe možnosti.

```
#include <stdio.h>

void funkcija(int stevilo, int seznam[]) {
```

```
    stevilo = 123; // ta sprememba ne vpliva na a v main-u
    seznam[0] = 123; // ta sprememba vlpliva na sez v main-u
}

int main() {
    int a = 10;
    int sez[] = {10};
    funkcija(a, sez);
    // a bo še vedno 10, medtem, ko bo seznam enak {123}
    return 0;
}
```

Sprva ne vidimo smisla, za drugačno obravnavo seznamov, a se izkaže, da je lahko ta lastnost tudi uporabna. Če bi želeli napisati funkcijo, ki sprejeme seznam in njegovo dolžino, potem pa obrne vrstni red elementov tega seznama, lahko to napišemo takole:

```
#include <stdio.h>

void obrni(int seznam[], int dolzina) {
    // Obrnemo vrstni red prvih dolzina elementov seznama
    for (int i = 0; i < dolzina / 2; i++) {
        int vmesna = seznam[i];
        seznam[i] = seznam[dolzina-i-1];
        seznam[dolzina-i-1] = vmesna;
    }
}

int main() {
    int n;
    int seznam[1000];

    // preberemo seznam dolžine n
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &seznam[i]);
    }

    obrni(seznam, n);

    // izpišemo obrnjen seznam
    for (int i = 0; i < n; i++) {
        printf("%d\n", seznam[i]);
```

```
27      }
28
29      return 0;
30  }
```

8.3. Rekurzija

Rekurzija je način pisanja programa na način, da funkcija kliče samo sebe. Pogosto lahko zanko nadomestimo z rekurzijo, če nam je tak način programiranja ljubši. Recimo, da želimo sešteti vsa števila med 1 in nekim n. Z zanko bi to naredili tako, da si pripravimo spremenljivko `vsota`, potem pa vanjo prištevamo števila, ki jih vidimo v zanki. Vsoto prvih n števil pa izračuna tudi naslednja funkcija:

```
1  int vsota(int n) {
2      // osnovni pogoj
3      if (n <= 0) {
4          return 0;
5      }
6
7      // korak
8      return n + vsota(n - 1);
9  }
```

Najprej napišemo *osnovni pogoj*, ki v primeru, da je argument `n` manjši ali enak 0, takoj vrne 0. Ta pogoj nujno potrebujemo, saj se bo sicer rekurzija ponavljala v neskončnost. Po osnovnem pogoju zapišemo *korak rekurzije*, ki izračuna `vsota(n)` s pomočjo vrednosti funkcije v nekem manjšem številu, v našem primeru kar `vsota(n - 1)`.

Poglejmo si, kaj se zgodi, če pokličemo `vsota(4)`. Funkcija izračuna $4 +$ vsota(3), ampak da bi izračunali to, moramo prvo izračunati `vsota(3)`. Pri računanju `vsota(3)` izračunamo $3 +$ vsota(2), za kar moramo izračunati `vsota(2)`. Potem podobno nadaljujejo, pri računanju `vsota(2)` računamo `vsota(1)`, pri računanju tega pa izračunamo `vsota(0)`. Tu se rekurzija ustavi, saj pridemo v prvi pogojni stavek, in takoj vrnemo vrednost 0. Potem lahko izračunamo vsota$(1) = 1 + 0$, iz česar dobimo vsota$(2) = 2 + 1$, potem vsota$(3) = 3 + 2$, vsota$(4) = 4 + 6$ in na koncu vsota$(5) = 5 + 10 = 15$.

9. Teorija števil

V programiranju se pogosto pojavijo razne matematične teme, ker nam dovoljujejo, da na smiseln način opišemo razne probleme, s katerimi se srečujemo. Ena od najbolj zanimivih vej matematike je teorija števil. Iz nje izhaja marsikateri algoritem, s katerim si lahko pri programiranju pomagamo; mi si bomo ogledali dva primera, ki izhajata že iz stare Grčije.

9.1. Evklidov algoritem

Evklidov algoritem je postopek, s katerim lahko hitro poiščemo največji skupni delitelj dveh števil. Spomnimo se; največji skupni delitelj števil a in b je *največje* število d, ki deli tako a kot b. V nadaljevanju ga bomo zapisali kot NSD ali $D(a, b)$.

Kot primer, NSD števil 30 in 75 je 15 ker sta tako 30 kot 75 deljiva s 15 (velja namreč $30 = 2 \cdot 15$ in $75 = 5 \cdot 15$), večje število, ki bi delilo oba, pa ne obstaja. Preprost algoritem, ki poišče največji skupni delitelj dveh števil, je sledeč;

```
Z zanko preiščemo vsa števila od d=1 do d=manjsi(a, b)
    preverimo, ali d deli a, ter ali d deli b.
    če deli oba, potem nastavimo NSD = d.
Ko se zanka zaključi, smo našli NSD.
```

Algoritem res pravilno deluje, ker bo zadnje število, ki bo delilo tako a kot b, ravno NSD, zato bo spremenljivka na koncu zanke pravilno nastavljena. Kljub pravilnemu delovanju pa tega algoritma ne želimo uporabljati, ker je zelo počasen. V pomoč nam priskoči drug algoritem, ki ga je opisal starogrški matematik Evklid v svoji knjigi *Elementi*. Ta algoritem bo prav tako vedno dal pravilni rezultat, za razliko od prejšnjega pa bo tudi zelo hiter.

Delovanje algoritma si poglejmo na primeru. Izračunajmo, koliko je $D(54, 42)$. V prvem koraku izračunamo količnik in ostanek pri deljenju 54 s 42:

$$54 = 42 \cdot 1 + 12.$$

Količnika v nadaljevanju ne bomo potrebovali, tako da ga lahko ignoriramo. V drugem koraku prestavimo delitelja na mesto deljenca, ostanek pa na mesto delitelja, ter spet izračunamo količnik in ostanek:

$$42 = 12 \cdot 3 + 6.$$

Postopek bomo ponovili še enkrat. V zadnjem koraku izračunamo količnik in ostanek pri deljenju 12 s 6. Spet smo premaknili delitelja na mesto deljenca ter deljenca na mesto delitelja:

$$12 = 6 \cdot 2 + 0.$$

Ko dobimo ostanek 0, se je algoritem zaključil (v naslednjem koraku bi morali deliti z 0, česar pa ne smemo početi). Takrat je največji skupni delitelj števil zapisan na mestu delitelja (oz. je ostanek, ki smo ga dobili v enem koraku prej).

Na roko lahko preverimo, da je NSD števil 54 in 42 res 6; dobili smo pravi rezultat. Opisan algoritem lahko enostavno zapišemo kot funkcijo v C++:

```
int nsd(int a, int b) {
    int c = a % b;
    while (c != 0) {
        a = b;
        b = c;
        c = a % b;
    }
    return b;
}
```

Če nas namesto največjega skupnega delitelja zanima najmanjši skupni večkratnik, lahko tega izračunamo po formuli:

$$D(a, b) \cdot v(a, b) = a \cdot b.$$

V tej formuli smo označili največji skupni delitelj z D in najmanjši skupni večkratnik z v. Za izračun najmanjšega skupnega večkratnika torej prvo izračunamo največji skupni delitelj s pomočjo Evklidovega algoritma, nato pa uporabimo formulo `v = a / D * b`. Pomembno je, da prvo delimo, sicer bi lahko vmesni rezultat (produkt) postal prevelik za naš številski tip (`int` oziroma `long long`), in povzročil napako.

9.2. Eratostenovo rešeto

Praštevilo je tako število, ki je deljivo le z 1 in samo s seboj. Praštevil je neskončno, najmanjša od njih so $2, 3, 5, 7, 11, 13, \ldots$. V matematiki so praštevila zelo pomembna, med drugim zaradi naslednje trditve, ki se ji reče *osnovni izrek teorije števil*.

Izrek: Vsako število lahko na enoličen način zapišemo kot produkt praštevil.

Izrek nam pove, da lahko poiščemo praštevilski razcep poljubnega števila. Poglejmo si 6552. To število je deljivo z 2, pri deljenju dobimo količnik 3276. Ta količnik lahko spet delimo z 2 in dobimo 1638. Ko še tretjič delimo z 2, dobimo rezultat 819, to število pa ni več deljivo z 2. Lahko ga še dvakrat delimo s 3; sprva dobimo 273, nato 91. Naposled lahko 91 delimo s 7 in dobimo 13. Praštevilski razcep števila 6552 je tako enak

$$6552 = 2^3 \cdot 3^2 \cdot 7 \cdot 13.$$

V primeru smo videli, da se v praštevilskem razcepu lahko kakšno praštevilo ponavlja, kakšno pa se sploh ne pojavi. Če želimo razcep poiskati s programom, ali pa narediti

kaj drugega s praštevili, pa jih moramo prvo poiskati. Spoznali bomo en algoritem za iskanje praštevil, to je Eratostenovo rešeto.

Recimo, da želimo poiskati vsa praštevila do neke zgornje meje N. Sprva si napišemo vsa števila med 2 in N (1 namreč ni praštevilo), kot v spodnji sliki:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 ...

Vemo, da je število 2 praštevilo. Večkratniki 2 zato niso praštevila, ker so deljivi z 2; torej jih prečrtamo.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ ...

Število 3 je praštevilo, torej tudi večkratniki 3 niso praštevila.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ...

Prišli so do števila 4, ki je prečrtano, zato vemo, da ni praštevilo. Ker je deljivo z 2, smo vse večkratnike 4 (ki so tudi večkratniki 2), že prečrtali, torej nam ni treba posebej črtati večkratnikov 4. Število 5 je praštevilo (ker ni prečrtano), torej moremo prečrtati njegove večkratnike.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ...

Število 6 je prečrtano, tako da ga preskočimo.

Postopek bi lahko nadaljevali, dokler ne pridemo do 26 (oz. do želene zgornje meje), vendar nam ni treba. Velja namreč naslednje; če je število M produkt dveh števil, $M = AB$, potem je vsaj eno od teh števil manjše od ali enako veliko kot $\sqrt{M}$. Če bi bili obe števili večji kot $\sqrt{M}$, je njun produkt večji od M, to pa ne mora veljati, ker je njun produkt ravno M. V našem primeru smo torej lahko iskanje ustavili že pri $\sqrt{26}$, kar je nekaj več kot 5, zato moramo preveriti tudi 6, več pa ne.

Možna implementacija algoritma za iskanje praštevil je podana spodaj. Zapisana koda ohranja še dodatno informacijo; za vsako število si zapomnimo enega od njegovih praštevilskih deliteljev.

```
1  int M;  // zgornja meja
2
3  // Tabela, ki hrani trenutno stanje rešeta
4  // če je na nekem mestu zapisana številka 0, to pomeni,
5  // da še nismo našli nobenega praštevilskega delitelja.
6  // Ko nekega delitelja najdemo, ga zapišemo
7  int reseto[M+1];
8
9  void poisci_prastevila() {
10     // V spremenljivki i bomo hranili število, ki ga trenutno
11     // obravnavamo začnemo pri 2, ker niti 0 niti 1 nista praštevili
12     // Iteriramo samo do korena M
13     for (int i = 2; i*i <= M; i++) {
14         // Če je na mestu i ničla, do sedaj nismo našli
```

```
15          // nobenega praštevila, torej mora biti število i
16          // nujno praštevilo
17          if (reseto[i] == 0) {
18              printf("Najdeno praštevilo %d!\n", i);
19
20              // Sedaj "prečrtamo" vse večkratnike števila i
21              for (int j = 2*i; j <= M; j += i) {
22                  // Zabeležimo si, da je i delitelj j
23                  reseto[j] = i;
24              }
25          }
26      }
27  }
```

Ta implementacija nam dovoljuje, da hitro poiščemo nekega delitelja poljubnega števila `n`; preprosto pogledamo v `reseto[n]`. Če najdemo `0`, vemo, da je `n` praštevilo, sicer pa smo našli nekega delitelja.

10. Vhod in izhod: dopolnitev

10.1. Dodatni formatniki za `scanf`

Poleg že znanih formatnikov `%d`, `%lld` in `%s` poznamo tudi druge, ki so lahko uporabni v različnih situacijah. Seznam pogosto uporabnih formatnikov je v tabeli 10.1. V tabeli se pojavi izraz *prazen znak*, angl. *whitespace* — to je znak, ki zasede prostor v spominu (in kateremu pripada ASCII koda), vendar se na zaslonu ne pojavi. Poznamo tri take znake, to se presledek, tabulator `\t` in znak za novo vrstico `\n`. Formatniki `%d`, `%lld` in `%s` preberejo, a ignorirajo, prazne znake, ki se pojavijo pred vsebino, ki jo formatnik dejansko shrani. Če npr. v spodnji primer programa vpišemo vhod

```
Hello

␣␣␣World!
```

bo program prebral le ti dve besedi, in ne praznih znakov med njima.

formatnik	opis
`%%`	en znak `%`
`%d`	število tipa `int`
`%lld`	število tipa `long long`
`%c`	poljuben znak (lahko tudi prazen znak)
`%s`	zaporedje nepraznih znakov
`%[...]`	zaporedje znakov, ki se pojavijo med oglatimi oklepaji
`%[^...]`	zaporedje znakov, ki se ne pojavijo med oglatimi oklepaji

Tabela 10.1.: Različni formatniki za `scanf` in `printf`

```c
#include<stdio.h>

int main(){
    char a[50], b[50];
    scanf("%s", a);
    scanf("%s", b);
    printf("%s %s\n", a, b);
    return 0;
}
```

Zadnja formatnika v tabeli, `%[...]` in `%[^...]` sta bolj zapletena od ostalih. Uporabljamo ju tako, da namesto tropičja v oglate oklepaje zapišemo seznam znakov, ki so dovoljeni v prebranem besedilu. Tako lahko npr. enostavno ločimo besedo, sestavljeno iz črk in številk na niz in na število, ki ga lahko takoj shranimo v `int`:

```
#include <stdio.h>

int main() {
    char a[50];
    int b;
    scanf("%[abcd]%d", a, &b);
    printf("Niz: %s, stevilo: %d\n", a, b);
    return 0;
}
```

Če zgornjemu programu kot vhod podamo `aacbb012`, bo izpisal

```
Niz: aacbb, stevilo: 12
```

Če želimo dovoliti vse male črke abecede, lahko namesto seznama vseh črk zapišemo tudi `%[a-z]`, podobno lahko za velike črke uporabimo `%[A-Z]`, za števke pa `%[0-9]`. Formatnik bo v zadnjem primeru še vedno prebral niz znakov, in ga ne bo avtomatsko pretvoril v `int`. Tudi te obsege lahko kombiniramo, formatnik `%[a-z2-7B]` npr. prebere niz, sestavljen iz malih črk angleške abecede, števk med 2 in 7 ter velike črke B.

Formatnik `%[^...]` deluje podobno kot formatnik `%[...]`, le da prebere vse znake, razen tistih, ki smo jih zapisali namesto tropičja. Ta formatnik je uporaben pri branju niza do konca vrstice, česar s formatnikom `%s` ne moremo narediti, če nimamo podanega števila besed v vrstici. Primer je prikazan v spodnjem programu, kjer uporabimo tudi formatnik `%*c`. Ta deluje enako kot formatnik `%c`, le da znaka, ki ga prebere, nikjer ne shrani, zato zanj tudi ne podamo spremenljivke. Ta formatnik je potreben, ker `%[^\n]` ne prebere znaka za novo vrstico, ki ga „počistimo“ z `%*c`. Če bi želeli prebrati dve zaporedni vrstici in nebi uporabili `%*c`, nebi druga uporaba formatnika `%[^\n]` shranila ničesar, saj bi nemudoma naletela na znak za novo vrstico, in zaključila branje.

```
#include<stdio.h>

int main(){
    char a[50];
    scanf("%[^\n]%*c", a);
    printf("%s\n", a);
    return 0;
}
```

10.2. Neznano število podatkov

Če vemo točno, koliko besed, številk oz. vrstic bomo imeli na vhodu, jih lahko preberemo z zanko. Kaj pa, če ne vemo, koliko podatkov bo na vhodu, kot v nalogi Neznane vsote? Če podatke beremo v neskončni zanki, bomo sicer prebrali vse, a se program ne bo nikoli ustavil. V naslednjem primeru je nakazano, kako ta problem rešimo:

```
#include<stdio.h>

int main(){
    int a;
    while(scanf("%d", &a) != EOF){
        printf("%d\n", a*a); //izpisujemo kvadrate prebranega števila
    }
    return 0;
}
```

Funkcija `scanf` vrne število formatnikov, ki jih je uspešno prebrala (če ji kot parameter npr. podamo samo `%d`, bo vrnila 1, če je uspešno prebrala število, sicer pa 0). Posebno vrednost `EOF` (End of File) pa vrne, če na vhodu ni ničesar več za prebrati. Tedaj se bo zanka ustavila. Če programu vhodne podatke podajamo iz datoteke, se to zgodi avtomatsko ob koncu datoteke, če pa mu podatke podajamo na roko, konec vhoda sporočimo s `Ctrl+D` (Linux in MacOS) ali `Ctrl+Z` (Windows).

10.3. Uporaba nizov ali datotek za vhodno-izhodni sistem

Včasih si lahko pri procesiranju podatkov pomagamo s tem, da dano besedilo prvo shranimo v niz, preden ga pretvorimo v drugo obliko, ali iz njega izluščimo podatke. Pri tem si lahko pomagamo s funkcijama `sscanf` in `ssprintf`, ki delujeta podobno kot poznana `scanf` in `printf`, vendar namesto branja iz oz. pisanja na standardni vhod oz. izhod delujeta z nizi v spominu. Niz, s katerega beremo oz. na katerega pišemo, podamo kot prvi argument v ti funkciji, kakor v spodnjem primeru.

```
#include<stdio.h>

int main(){
    int n;
    char a[10], b;
    char text[]="Slovenska 157 a";
```

```
    sscanf(text, "%s%d %c", a, &n, &b);
    sprintf(text, "%c %d %s", b, n, a);
    printf("%s\n", text);
    return 0;
}
```

Podobno lahko s pomočjo funkcij `fscanf` ter `fprintf` beremo iz in pišemo v datoteke na računalniku. Datoteke predstavimo s posebnim tipom `FILE`, pri katerem moramo poleg imena spremenljivke napisati tudi zvezdico (zakaj je temu tako, spoznamo v enem od kasnejših poglavij). Da se povežemo s pravo datoteko, uporabimo funkcijo `fopen`, ki ji podamo dva argumenta. Prvi argument je ime datoteke, ki jo želimo uporabljati, drug argument pa je možnost odpiranja. Za naše potrebe sta dovolj dve možnosti, `"r"`, ki pove, da bomo datoteko odprli v načinu za branje, ter `"w"`, ki pove, da bomo datoteko odprli v načinu za pisanje. Slednja možnost tudi ustvari datoteko z danim imenom, če ta ne obstaja, oziroma izbriše vso besedilo v tej datoteki, če datoteka že ima vsebino. Primer uporabe funkcij `fscanf` in `fprintf` je prikazan spodaj. Opazimo, da se na koncu programa pojavi tudi funkcija `fclose`. Ta funkcija operacijskemu sistemu pove, da smo z branjem oz. pisanjem zaključili, in da lahko spremembe na tej točki shrani v dejansko datoteko (brez klica te funkcije shranjevanje sprememb ni zagotovljeno). Funkcijo moramo nujno poklicati za vsako datoteko, ki smo jo odprli z `fopen`.

```
#include<stdio.h>

int main(){
    int a;
    FILE *fr, *fw;
    fr = fopen("in.txt", "r");
    fw = fopen("out.txt", "w");
    while(fscanf(fr, "%d", &a) != EOF){
        fprintf(fw, "%d\n", a*a);
    }
    //iz datoteke fr preberemo vsa števila
    // in v fw izpišemo njihove kvadrate
    fclose(fr);
    fclose(fw);
    return 0;
}
```

11. Spomin in kazalci

11.1. Računalniški spomin

Spomin, pomnilnik ali angl. RAM (*Random access memory*) je ključna komponenta v računalniku. Med tekom programa so v njem shranjene vse spremenljivke, ki jih program uporablja. S spremenljivkami lahko naredimo tudi več, kot smo se do sedaj naučili, vendar pa moramo za to razumeti, kako je spomin zgrajen.

Osnovna enota za merjenje količine informacij je *bit*. En bit informacij ustreza odgovoru na eno vprašanje tipa da ali ne – če nam nekdo pove, da so vrata zaprta, nam je podal en bit informacij, ker so lahko vrata bodisi odprta bodisi zaprta. Če imamo v omari dva para hlač, dve majici in dve kapi, lahko opišemo, kako smo oblečeni, s tremi biti informacije — za vsak kos oblačila porabimo en bit.

V računalništvu bite najpogosteje označujemo z ničlami in enicami. Običajno ničla predstavlja odgovor „ne“ na dano vprašanje, enica pa odgovor „da“. Bit pa je zelo majhna količina informacije, zato pogosto govorimo v večjih enotah, kot so *bajti*, *kilobajti*, *megabajti* itd. En bajt ustreza osmim bitom, kilobajt je tisoč bajtov, megabajt je tisoč kilobajtov, gigabajt je tisoč megabajtov, in terabajt je tisoč gigabajtov.

Računalniški spomin je sestavljen iz spominskih celic, ki so dolge en bajt. Vsaka od teh celic ima svoj *naslov* — številko, s katero lahko to celico ločimo od ostalih. Naslovi so zaporedne številke od 0 do velikosti pomnilnika, ki ga imamo nameščenega v računalniku. Celice so naraščajoče urejene po svojih naslovih, tako da je celica številka 150337 sosednja celicama s številkama 150336 in 150338.

Upravljanje z računalniškim spominom je ena od nalog operacijskega sistema. Naši programi operacijski sistem med izvajanjem prosijo za neko količino spomina, operacijski sistem pa določi, katere spominske celice bo program prejel. Te celice tedaj pripadajo programu, dokler se ta ne zaključi, ali dokler tega spomina ne vrne operacijskemu sistemu na drugačen način. Med izvajanjem našega programa praviloma noben drug program nima dostopa do tega dela spomina.

Kako pa program ve, koliko spomina bo potreboval? Da to izračuna, se zanaša na tipe. Vsaka spremenljivka ima tip, vsak tip pa ima fiksno dolžino, ki jo zavzame v spominu. Dolžine pogostih tipov so sledeče:

- `int`: 4 bajti
- `long long`: 8 bajtov
- `char`: 1 bajt
- `bool`: 1 bajt

Spremenljivke, ki v spominu zavzamejo več kot 1 bajt, moramo shraniti v več kot eno spominsko celico. Celice, v katere te vrednosti zapišemo, so v spominu zaporedne; če imamo spremenljivko tipa `int`, bo tako zavzela 4 zaporedne celice.

11.2. Kazalci

V tem razdelku spoznamo kazalce, ki so pomemben koncept v programiranju, pred tem pa podajmo še eno opombo. Razumevanje kazalcev je ključno za razumevanje bolj zapletenih podatkovnih struktur ter nekaterih algoritmov, vendar se lahko z njihovo uporabo hitro zmotimo. Poleg tega je razhroščevanje kode z veliko kazalci pogosto zelo zapleteno, zato se v tekmovalnem programiranju direktne uporabe kazalcev vestno izogibamo. Pogosto jih lahko nadomestimo z indeksi v seznamu ali drugačno strukturo kode, s čimer skoraj vedno pridobimo na razumljivosti kode. Kazalce na tekmovanjih uporabimo le, če je to res nujno!

Ker so spominski naslovi številke, jih lahko shranjujemo, kakor shranjujemo ostale številke. Za to imamo v C++ na voljo poseben tip, ki mu rečemo *kazalec* (angl. *pointer*). Pravzaprav kazalec ni sam svoj tip, ampak razširitev nekega drugega tipa; pravimo, da kazalec *kaže na drug tip*. Da ustvarimo nov kazalec, zapišemo ime tipa, na katerega želimo kazati, nato pa pred ime spremenljivke damo zvezdico `*`. Kazalcu lahko nastavimo vrednost tako, da vanj shranimo naslov neke spremenljivke, ki smo že ustvarili. Do tega naslova dostopamo z operatorjem `&`. Da dostopamo do vrednosti, shranjene v celici, na katero kazalec kaže, uporabimo operator `*` (ki ima drugačen pomen, kot zvezdica v deklaraciji spremenljivke).

```
#include <stdio.h>

int main() {
    int a = 7;
    int b = 3;

    // Ustvarimo kazalec na int, ki kaže na spremenljivko a
    // To pomeni, da bo v kazalcu shranjen naslov prve od štirih
    // celic, ki jih zavzema a.
    int *kazalec = &a;

    // Izpišemo vrednost, na katero kaže kazalec
    printf("%d\n", *kazalec);  // 7

    // Če spremenimo vrednost, na katero kaže kazalec, se spremeni
    // vrednost v spominu, torej tudi spremenljivka, ki je tam
    // shranjena
    *kazalec = 15;
    printf("%d\n", a);  // 15
```

```
20
21      // Če spremenimo lokacijo, kamor kaže kazalec, nismo spremenili
22      // vrednosti, na katero je kazal prej
23      kazalec = &b;
24      printf("%d\n", *kazalec);  // 3
25      printf("%d\n", a);  // 15
26
27      return 0;
28  }
```

Vse, kar smo zgoraj delali s kazalci, je bilo možno (in lažje) narediti tudi z običajnimi spremenljivkami. Kazalci, ki kažejo na spremenljivke, ki tako in tako že obstajajo, so bolj ali manj neuporabni. Kako pa naredimo kazalec, ki kaže na del spomina, v katerem še ni nobene spremenljivke? Če želimo storiti kaj takega, moramo prevzeti odgovornost za upravljanje spomina v našem programu. Do sedaj je za to skrbel prevajalnik, ki je v naš program na pravilna mesta zapisal ukaze, ki si spomin sposojajo od operacijskega sistema, ter ga vračajo, ko ga ne potrebujemo več. Bolj natančno; ko smo deklarirali spremenljivko, je prevajalnik poskrbel, da prosimo za natanko toliko spomina, kolikor ga za to spremenljivko potrebujemo (zato moramo za vsako spremenljivko zapisati tip), ter si njegov naslov zapomnil, ko pa spremenljivke nismo več potrebovali, je prevajalnik poskrbel, da ta del spomina vrnemo operacijskemu sistemu; temu pravimo *sprostitev* (angl. *deallocation*).

Za bolj sofisticirano uporabo spomina moramo ti vlogi prevzeti mi. Za to sta nam na voljo dve funkciji: `malloc` in `free`. Da ju uporabljamo, moramo vključiti knjižnico `stdlib.h`. Oblika funkcij je naslednja:

```
1  void *malloc(size_t size);
2  void free(void *ptr);
```

V obliki je posebnost, ki je še nismo omenili; ni namreč nujno, da ima vsak kazalec tip. Lahko imamo kazalce, ki kažejo na del spomina, mi pa (še) ne vemo, ali pa ni pomembno, kaj je v tistem delu spomina shranjeno. Za take kazalce pravimo, da kažejo na `void`, kar pa ne pomeni, da ne kažejo na nič; na lokaciji v spominu, kamor kažejo, je nekaj shranjeno, mi samo ne vemo, kako naj te podatke interpretiramo.

Funkcija `malloc` sprejme en argument, in vrne kazalec na `void`. Argument je tipa `size_t`, ki je za naše potrebe skoraj enak tipu `unsigned long long`, to je torej številka. Argument pove, koliko bajtov spomina si želimo sposoditi od operacijskega sistema. Funkcija `malloc` nato vrne kazalec na prvi naslov znotraj bloka spomina, ki smo si ga ravno sposodili. Ker operacijski sistem ne ve, kaj bomo v ta spomin shranili, nam `malloc` vrne `void*`, mi pa ga moramo pretvoriti v pravi tip kazalca. To storimo tako, da tik pred klic funkcije v oklepaje zapišemo želeni tip, kakor bomo videli v naslednjem primeru.

Funkcija `free` je ravno nasprotna od `malloc`. Sprejme kazalec, ki ga nam je dal `malloc`, ter sprosti del spomina, na katerega kazalec kaže. Le-ta bo po klicu `free` še vedno obstajal, in bo še vedno kazal na isto mesto. Edina sprememba je, da del spomina, na katerega kaže, ne pripada več našemu programu, in ga ne smemo uporabljati.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *kazalec = (int*) malloc(sizeof(int));
    *kazalec = 3;
    printf("%d\n", *kazalec);  // izpiše 3
    return 0;
}
```

V primeru zgoraj smo uporabili operator `sizeof`, ki nam preprosto pove, koliko bajtov zasede naveden tip.

11.3. Kako delujejo seznami

Nič nas ne omejuje, da od operacijskega sistema zahtevamo zelo velik blok spomina, tudi po več sto tisoč bajtov. Pa imamo lahko kakšen utemeljen razlog, da si toliko spomina izposodimo? Da, ravno to stori prevajalnik, ko ustvarimo seznam. Poglejmo si, kako ustvarimo seznam samo s kazalci.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int dolzina_seznama = 100000;
    int *seznam = (int*)malloc(dolzina_seznama * sizeof(int));

    for (int i = 0; i < dolzina_seznama; i++) {
        scanf("%d", seznam+i);
    }

    // sedaj lahko spremenljivko `seznam` uporabljamo
    // kot katerikoli drugi seznam
    int vsota = 0;
    for (int i = 0; i < dolzina_seznama; i++) {
        vsota += seznam[i];
    }
```

```
18
19      printf("%d\n", vsota);
20
21      // ne smemo pozabiti sprostiti spomina
22      free(seznam);
23
24      return 0;
25  }
```

V zgornjem primeru uporabljamo dva nova operatorja na kazalcih; seštevanje in oglate oklepaje. Če kazalcu `seznam` prištejemo število `i`, dobimo nov kazalec, ki kaže na mesto `seznam + (velikost tipa) * i`, torej na idealno mesto, kamor lahko zapišemo `i`-ti element seznama, če jih zapisujemo enega za drugega. Drug novi operator so oglati oklepaji — ti se obnašajo popolnoma enako kot v seznamih. Oglati oklepaj `seznam[i]` je pravzaprav krajšava za zapis `*(seznam+i)`, torej za dostop do `i`-tega elementa v bloku spomina.

Tudi seznami, kakor smo jih spoznali prej, so dejansko kazalci na blok spomina, le da s tem spominom upravlja prevajalnik. Trik s prištevanjem števila k kazalcu deluje tudi za prištevanje števila k seznamu.

11.4. Podajanje po referenci

Opazimo, da smo operator `&` že srečali, in sicer čisto na začetku. Pri branju številk iz vhoda moramo v `scanf` zapisati ta operator pred imenom spremenljivke. Sedaj razumemo, zakaj je temu tako; `scanf` sprejme kazalce na spremenljivke, ki jih želimo prebrati, ter popravi vrednosti, na katere kažejo kazalci, s prebranimi vrednostmi.

Funkcije, kot smo jih pisali do sedaj, niso sposobne spremeniti vrednosti spremenljivk zunaj funkcije. Spodnji program se na primer ne bo niti prevedel:

```
1   #include <stdio.h>
2
3   void f(int x) {
4       y = 2 * x;
5   }
6
7   int main() {
8       int y;
9       int x = 3;
10      f(x);
11      printf("%d\n", y);
12      return 0;
13  }
```

Naslednji program pa se bo prevedel, vendar bo izhod morda v nasprotju s pričakovanji:

```
#include <stdio.h>

void f(int x) {
    x = 2 * x;
}

int main() {
    int x = 3;
    f(x);
    printf("%d\n", x);  // izpiše 3
    return 0;
}
```

Spremenljivke, ki jih deklariramo v funkciji, to je znotraj telesa funkcije, ali pa v seznamu argumentov, so lokalne na to funkcijo — zunaj nje sploh ne obstajajo. Če želimo, da funkcija popravi neko vrednost, ki jo uporabljamo tudi zunaj funkcije, smo do sedaj lahko to naredili samo tako, da smo spremenljivko naredili globalno, torej dostopno vsem funkcijam v kodi. Kaj pa, če želimo neko spremenljivko na tak način deliti samo med dvema funkcijama?

Da odgovorimo na to vprašanje, moramo razumeti, kako se argumenti podajajo v funkcije. Ko neko funkcijo pokličemo, se argumenti, ki jih funkciji podamo, *prekopirajo* v poseben del spomina, ki ji pripada. Ko smo znotraj ene funkcije, ne poznamo imena spremenljivk v drugih funkcijah; prav tako ne vemo, kje so te spremenljivke shranjene. Nič pa nam ne preprečuje, da spreminjamo spomin, ki našemu programu pripada, pa četudi smo ga rezervirali v drugi funkciji; razen tega, da ne vemo, kateri del spomina je naš, in kateri ni. Lahko si predstavljamo, da smo zabredli v spominsko džunglo, v kateri ne prepoznamo prave poti do spremenljivk, ki jih želimo popraviti. Če pa s seboj prinesemo zemljevid, bomo nenadoma to lahko naredili. Ta zemljevid je kazalec.

Funkcija lahko brez težav sprejme kazalec kot argument. Če to storimo, pravimo, da smo funkciji vrednost podali *po referenci* (angl. *pass by reference*), namesto da bi argument podali običajno, čemur pravimo *podajanje po vrednosti* (angl. *pass by value*). Kazalec, ki smo ga podali, se bo še vedno prekopiral v del spomina, ki pripada funkciji, vrednost, na katerega kazalec kaže, pa bo ostala tam, kjer je. Tako lahko skozi kazalec spremenimo vrednosti spremenljivk zunaj funkcije. Prvi primer od zgoraj bi lahko tako popravili na naslednji način:

```
#include <stdio.h>

```

```c
void f(int vrednost, int *kazalec) {
    *kazalec = 2 * vrednost;
}

int main() {
    int x = 3;
    int y;
    f(x, &y);
    printf("%d\n", y);  // izpiše 6
    return 0;
}
```

12. Urejanje

V programih pogosto želimo nek seznam števil urediti po vrsti. V ta namen lahko napišemo svojo funkcijo, ki implementira enega od znanih algoritmov za urejanje; npr. *bubble sort*, *insertion sort*, *quick sort*, ipd. Ker pa so učinkovite implementacije pogosto komplicirane in se pri pisanju hitro zmotimo, je bolje, da uporabimo funkcije, ravno v ta namen vključene v standardno knjižnjico. Za to bomo potrebovali na začetek programa dodati še dve vrstici:

```
#include <algorithm>
using namespace std;
```

Ukaz `#include` že poznamo, opazimo pa, da tokrat za spremembo nima končnice `.h`. To je zato, ker funkcije, ki smo jih uporabljali do sedaj, izvirajo iz jezika C, tokrat pa potrebujemo funkcijo, napisano posebaj za C++. To razloži tudi drugo vrstico; vse funkcije v standardni knjižnjici v C++ so vključene v imenski prostor `std`. Če jih želimo klicati, moramo pred ime funkcije vedno napisati `std::`, ali pa na začetek programa vključiti vrstico `using namespace std`.

Sedaj lahko uporabimo funkcijo `sort`, ki sprejme dva argumenta; kazalec na začetek in tik za konec predela spomina, ki ga želimo urediti. Poglejmo si enostavni primer.

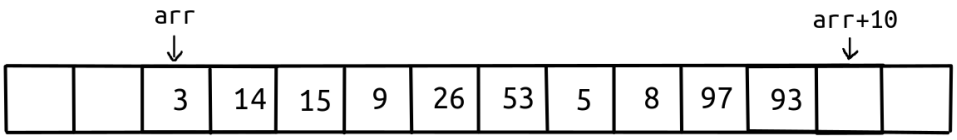
```
#include <algorithm>
#include <stdio.h>
using namespace std;

int arr[1000003];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    sort(arr, arr+n);
    for (int i = 0; i < n; i++) {
        printf("%d\n", arr[i]);
    }
```

```
17      return 0;
18  }
```

Program prebere število n, za njim pa še n števil, jih uredi naraščajoče, in jih izpiše. Funkcijo `sort` smo poklicali tako, da smo kot prvi argument podali seznam `arr` (oz. kazalec na prvi element seznama), kot drugi argument pa kazalec na prvo mesto v spominu, ki ne spada več v naš seznam (oz. prvo mesto v spominu, ki ga ne želimo urediti). Na primer, če želimo urediti seznam `arr` z 10 elementi, moramo podati kazalca `arr` in `arr+10`, kot je prikazano spodaj:



Optimalna časovna zahtevnost algoritma za urejanje je $O(n \log n)$. To v praksi pomeni, da bo urejanje delovalo dovolj hitro za $n \leq 10^6$ podatkov. Če imamo več podatkov kot toliko, bo urejanje trajalo predolgo in naša rešitev ne bo sprejeta.

12.1. Primerjalna funkcija

Če želimo urediti seznam padajoče namesto naraščajoče, lahko seznam prvo uredimo naraščajoče, in ga nato obrnemo. Ker s tem dobimo veliko dodatnega dela, je bolje, da funkciji `sort` podamo lastno primerjalno funkcijo. Le-ta mora sprejeti dva argumenta ter vrniti `bool`, in sicer; če mora biti prvi argument v urejenem seznamu levo od drugega, mora funkcija vrniti `true`, sicer pa `false`. Če ne podamo tretjega argumenta, se `sort` obnaša tako, kot da bi podali naslednjo funkcijo:

```
1   bool compare(int a, int b) {
2       return a < b;
3   }
```

Če želimo urediti seznam padajoče, moramo torej le podati nasprotno funkcijo, kot spodaj:

```
1   #include <algorithm>
2   #include <stdio.h>
3   using namespace std;
4
```

```
5  int arr[1000003];
6
7  int compare_padajoce(int a, int b) {
8      return a > b;
9  }
10
11 int main() {
12     int n;
13     scanf("%d", &n);
14     for (int i = 0; i < n; i++) {
15         scanf("%d", &arr[i]);
16     }
17     sort(arr, arr+n, compare_padajoce);
18     for (int i = 0; i < n; i++) {
19         printf("%d\n", arr[i]);
20     }
21     return 0;
22 }
```

12.2. Urejanje sestavljenih podatkov

Recimo, da imamo v nalogi dana imena tekmovalcev ter točke, ki so jih ti tekmovalci dosegli na tekmovanju, naš cilj pa je, da izpišemo imena tekmovalcev po vrsti glede na doseženo število točk. Če bomo prebrali točke in imena v različna seznama, ter uredili seznam točk, bo seznam imen ostal nespremenjen in ne bomo več vedeli, katero ime pripada katerim točkam.

Kako uredimo oba seznama hkrati? Najbolj enostavna možnost je uporaba `struct`, ki pa ga še ne poznamo. Namesto tega si lahko pripravimo seznam indeksov, ki na začetku na i-tem mestu hrani številko i. Potem sestavimo funkcijo `compare` tako, da sprejme dva indeksa, ter ju uredi glede na vrednosti v tabeli s točkami na pripadajočih indeksih. Urejamo pa ne tabele s točkami, temveč novo tabelo indeksov. Na ta način se tabeli s točkami in z imeni ne bosta spreminjali, in bodo točke pripadale imenu na istem indeksu.

Primer implementacije opisane rešitve je prikazan spodaj.

```
1  #include <algorithm>
2  #include <stdio.h>
3  using namespace std;
4
5  int tocke[100003];
6  char imena[100003][30];
```

```
7  int idxs[100003];
8
9  int compare(int i, int j) {
10     return tocke[i] > tocke[j];
11 }
12
13 int main() {
14     int n;
15     scanf("%d", &n);
16     for (int i = 0; i < n; i++) {
17         scanf("%s%d", imena[i], &tocke[i]);
18     }
19     // pripravimo tabelo indeksov
20     for (int i = 0; i < n; i++)
21         idxs[i] = i;
22
23     sort(idxs, idxs+n, compare);
24
25     for (int i = 0; i < n; i++) {
26         int idx = idxs[i];
27         printf("%s\n", imena[idx]);
28     }
29     return 0;
30 }
```

Če programu podamo spodnji vhod,

```
5
France 37
Gregor 34
Julija 38
Matija 29
Urska 8
```

nam bo izpisal imena, urejena padajoče po številu točk:

```
Julija
France
Gregor
Matija
Urska
```

13. Hitrost programov in asimptotična notacija

13.1. Merjenje hitrosti programa

Pogosto obstaja več možnosti, kako se lahko lotimo reševanja danega problema. Če želimo najti najmanjši element v seznamu, lahko na primer pregledamo celoten seznam in si beležimo najmanjšega, ki smo ga našli do sedaj, lahko pa celoten seznam uredimo po vrsti in nato izberemo prvi element. Pričakujemo lahko, da se bodo različni algoritmi za reševanje istega problema razlikovali tudi po tem, kako hitro problem rešijo. Kako pa v računalništvu izmerimo hitrost? Če delamo samo na enem računalniku, lahko izmerimo, konkretno koliko časa je program potreboval, da je zaključil z delovanjem. Na ta način lahko na primerih demonstriramo, da je nek algoritem boljši od drugega; ko pa želimo naše rezultate deliti in primerjati z drugimi, pa se ne moramo zanašati, da bodo imeli enako močen računalnik kot mi, in da bodo njihovi testni primeri primerljivo zahtevni z našimi. Dejansko so težave pri tem še hujše; na hitrost delovanja našega programa ne vpliva samo strojna oprema računalnika (torej, kakšen procesor ima, koliko ima spomina itd.), temveč tudi ostali programi, ki jih imamo hkrati odprte. Če se želimo pogovarjati o hitrosti algoritmov, potrebujemo bolj abstraktno orodje. Na pomoč pride asimptotična zahtevnost.

Da določimo hitrost našega programa, moramo prvo določiti, katere spremenljivke vplivajo na čas delovanja, ter kako je čas od njih odvisen. Rezultat take analize zapišemo kot izraz v oklepaje, pred katere zapišemo veliko črko O, takole: $O(\ldots)$. Poglejmo si primer.

```
1  int arr[100002];
2
3  int poisci_najmanjsega(int n) {
4      // Poišči najmanjše število v seznamu, dolgemu n
5      int min_idx = 0;
6      for (int i = 0; i < n; i++) {
7          if (arr[min_idx] > arr[i]) {
8              min_idx = i;
9          }
10     }
11     return arr[min_idx];
12 }
```

Funkcija `poisci_najmanjsega` sprejme število n, ki pove dolžino seznama `arr`. Po seznamu se nato enkrat sprehodi, in si ob tem beleži indeks najmanjšega elementa, ki ga je do sedaj našla. Razmislimo, katere vse različne operacije zgornji program opravi.

- Večkrat med programom nastavimo neki spremenljivki novo vrednost.
- V vsaki iteraciji zanke prištejemo 1 spremenljivki `i`.
- Poleg tega v vsaki iteraciji zanke tudi primerjamo `i` z `n`,
- dvakrat dostopamo do nekega elementa v seznamu,
- ter ju primerjamo.
- Na koncu še enkrat dostopamo do elementa v seznamu, ter ga vrnemo.

Vse naštete operacije same po sebi *trajajo* $O(1)$ časa. To pomeni, da se vedno izvajajo enako hitro, neodvisno od parametrov, ki jim podamo (npr. dve števili bomo vedno enako hitro sešteli, ne glede na to, ali seštevamo $5 + 7$ ali $123456 + 984621$). Rečemo tudi, da porabijo *konstantno mnogo* časa.

Kolikokrat pa izvedemo te operacije? Analizirajmo najslabši primer za naš program; če je seznam `arr` padajoče urejen. Tedaj bomo v vsaki iteraciji zanke enkrat primerjali `i < n`, dvakrat dostopali do vseh elementov podanega seznama, enkrat primerjali `arr[min_idx] > arr[i]`, enkrat nastavili `min_idx`, ter enkrat povečali `i`. Zunaj zanke bomo nastavili `min_idx` ter `i` na začetni vrednosti, ter še enkrat dostopali do elementa v seznamu. Zanka se vedno izvaja za natanko `n` iteracij; vedno vsak element pregledamo enkrat. Torej je celotna časovna zahtevnost našega programa $O(3 + 6n)$. Ker pa za velike n del zunaj zanke hitro postane nepomemben, ga ignoriramo, in zanemarimo 3 v časovni zahtevnosti. Poleg tega ignoriramo tudi faktor 6 pred členom n – ker tako in tako ne moramo vedeti, kako hitre so operacije v zanki v primerjavi druga z drugo, te konstante ne moramo natančno določiti. Končna časovna zahtevnost našega programa je torej $O(n)$.

To je tudi najboljši možni algoritem za iskanje najmanjšega elementa v seznamu. Če bi nek algoritem namreč deloval v hitrejšem času kot $O(n)$, bi moral nekatera mesta v seznamu izpustiti; če tedaj algoritmu podamo seznam, ki ima najmanjši element ravno na takem mestu, ga algoritem ne bo našel, in bo podal napačen odgovor.

Pomembna opazka je, da hitrost našega algoritma ni odvisna od velikosti števil v seznamu, temveč le od velikosti seznama. Naslednji program prav tako poišče najmanjše število v seznamu, vendar je konkretno počasnejši od zgornjega:

```
1  int arr[100002];
2  int poisci_najmanjsega_slabsi(int n, int m) {
3      // Poišči najmanjše število v seznamu, dolgemu n,
4      // kjer je največje število veliko največ m
5      for (int zelja = 0; zelja <= m; zelja++) {
```

```
6           // zelja nam pove, kateri element si v tej iteraciji želimo
7           // najti. Pogledati moramo še, da ta element dejansko je v
8           // seznamu; ko pa najdemo enega, bo to najmanjši
9           // (ker zelja v vsaki iteraciji narasca)
10          bool je_v_seznamu = false;
11          for (int i = 0; i < n; i++) {
12              if (arr[i] == zelja)
13                  je_v_seznamu = true;
14          }
15          if (je_v_seznamu)
16              return zelja;
17      }
18  }
```

V tem programu imamo dve zanki; ena se sprehaja po vseh možnih vrednosti števil v seznamu, druga pa preverja, če je ta element dejansko v seznamu. Vsakič, ko se zunanja zanka izvede enkrat, se notranja izvede `n`-krat (v najslabšem primeru), zunanja zanka pa se izvede `(m+1)`-krat. Torej je zahtevnost $O((m+1) \cdot n)$, oziroma $O(mn+n)$. Člen n v vsoti pa je v vseh primerih manjši od člena mn ali njemu enako velik, zato ga kot prej izpustimo. Končna časovna zahtevnost drugega algoritma je torej $O(mn)$.

Spremenljivka tipa `int` lahko hrani števila, velika do približno dve milijardi – v najslabšem primeru je m torej približno $2 \cdot 10^9$. Če prvi algoritem na nekem računalniku potrebuje eno sekundo, da se konča, bi drugi algoritem v najslabšem primeru na istem računalniku potreboval več kot šestdeset let.

Poglejmo si še nekaj primerov. Naslednji program za vsako število v seznamu `arr` poišče število števil desno od njega, ki so večja. Notranja zanka se v prvi iteraciji izvede $(n-1)$-krat, v drugi iteraciji $(n-2)$-krat, v tretji $(n-3)$-krat, itd. V zadnji iteraciji se sploh ne izvede. Skupaj se koda znotraj notranje zanke ponovi tolikokrat:

$$(n-1) + (n-2) + \ldots + 1 + 0 = \frac{n(n-1)}{2}$$

Spet ignoriramo konstanto $\frac{1}{2}$ ter člen samo z n, in pridemo do zahtevnosti $O(n^2)$.

```
for (int i = 0; i < n; i++) {
    int stevilo = 0;
    for (int j = i+1; j < n; j++) {
        if (arr[j] > arr[i]) {
            stevilo++;
        }
    }
    printf("%d\n", stevilo);
}
```

Spodnji program preveri, če je število `n` praštevilo. Program ima eno zanko, ki se sprehaja toliko časa, dokler kvadrat spremenljivke i ne postane večji od n, oz. dokler je $i \leq \sqrt{n}$. Zanka se torej izvede v $O(\sqrt{n})$.

```
bool preklicano = false;
for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
        printf("%d ni prastevilo\n", n);
        preklicano = true;
        break;
    }
}
if (!preklicano)
    printf("%d je prastevilo\n", n);
```

13.2. Klasifikacija

Računalnik lahko v eni sekundi opravi približno 10^7 operacij. Da določimo, kako dober algoritem potrebujemo za rešitev neke naloge, lahko preverimo omejitve vhodnih podatkov. Spodnja tabela prikazuje nekaj pogostih časovnih zahtevnosti, ter pripadajoče največje omejitve. Z uporabo te tabele lahko vnaprej določimo, kakšno največjo časovno zahtevnost mora imeti naš program, da reši določeno nalogo.

Zahtevnost	Omejitev za n	Ime zahtevnosti
$O(1)$	brez	*konstantna*
$O(\log n)$	zelo visoka	*logaritemska*
$O(\sqrt{n})$	10^{14}	*korenska*
$O(n)$	10^7	*linearna*
$O(n \log n)$	10^6	
$O(n^2)$	10^4	*kvadratna*
$O(n^3)$	300	*kubična*
$O(2^n)$	20	*eksponentna*

14. Bisekcija

14.1. Osnovna oblika

Recimo, da imamo dano število k, za katerega želimo čim hitreje preveriti, če se pojavlja v seznamu `arr` dolžine n. Običajen algoritem za tako iskanje preveri vsako število v seznamu, dokler števila k ne najde, oziroma dokler ne pride do konca seznama. V najslabšem primeru, če števila k ni v seznamu, ali če se pojavi le na zadnjem mestu, bo algoritem porabil $O(n)$ operacij za tako iskanje. To je najbolje možno, če število iščemo le enkrat, če pa v seznamu iščemo veliko različnih števil, lahko uporabimo spodnji postopek, da iskanje pohitrimo.

Najprej bomo seznam uredili od najmanjšega do največjega števila z enim od hitrih, t.j. $O(n \log n)$ algoritmov za urejanje. V urejenem seznamu lahko iskanje izvajamo na pametnejši način, ki mu pravimo *bisekcija* ali *binarno iskanje*. Prvo bomo pogledali število na sredini seznama. Če je to število večje od k, vemo, da se k bodisi nahaja na levi polovici seznama, bodisi ga na seznamu sploh ni – če bi se namreč k pojavil na desni polovici seznama, potem seznam nebi bil urejen! Podobno, če je element na sredini seznama manjši od k, se k nahaja v desni polovici seznama, ali pa ga v seznamu ni. S tem enim pogledom smo zmanjšali število možnih elementov, ki jih moramo pregledati, iz n na $\frac{n}{2}$. Če postopek ponovimo na polovičnemu seznamu, torej ga spet razdelimo na polovico, nato dobljeno četrtino še enkrat razdelimo na polovico, in tako dalje, potem smo pogledali le $\log n$ elementov, da smo našli število k, oziroma ugotovili, da ga v seznamu ni.

Iskanje lahko implementiramo, kot je napisano spodaj:

```
1  // dan imamo seznam arr dolžine n
2  // seznam prvo uredimo
3  sort(arr, arr+n);
4
5  // iz standardnega vhoda q-krat preberemo iskano število k,
6  // ter izpišemo "DA", če je število v seznamu, in "NE", če števila
7  // ni v seznamu
8  for (int i = 0; i < q; i++) {
9      int k;
10     scanf("%d", &k);
11
12     // hranimo si dva indeksa; levi in desni rob podseznama, na katerem
13     // smo prepričani, da se število k nahaja (če se sploh nahaja na
```

```
14      // seznamu). Na začetku nimamo nobenih informacij, zato nastavimo
15      // levi rob na 0 in desni rob na n-1.
16      int levo = 0;
17      int desno = n-1;
18
19      bool nasel = false;
20      while (levo <= desno) {
21          // Izračunamo indeks srednjega elementa na trenutnem podseznamu
22          // in preverimo, če je večji ali manjši od k
23          int sredina = (levo + desno) / 2;
24          if (arr[sredina] == k) {
25              // Našli smo k! Iskanje lahko takoj zaključimo.
26              nasel = true;
27              break;
28          }
29          // Posodobimo eno od mej podseznama glede na vrednost
30          // arr[sredina]
31          if (arr[sredina] > k) desno = sredina-1;
32          else levo = sredina+1;
33      }
34
35      if (nasel) printf("DA\n");
36      else printf("NE\n");
37  }
```

V postopku definiramo spremenljivki `levo` in `desno`, ki označujeta levi in desni rob dela seznama, ki ga moramo pregledati. V vsaki iteraciji izračunamo sredino seznama, in preverimo, če smo slučajno naleteli na število k. Če smo, lahko takoj končamo, sicer pa glede na vrednost `arr[sredina]` primerno spremenimo levi oziroma desni rob iskanja. V primeru `arr[sredina] > k` vemo, da je število na sredini trenutnega podseznama preveliko, torej se k pojavlja na levi polovici, vemo pa tudi, da se ne pojavi na indeksu `sredina`, zato lahko nastavimo novo desno mejo na `sredina-1`. Podobno v drugem primeru, če je `arr[sredina] < k`, nastavimo novo levo mejo na `sredina+1`.

Če bi si shranili tudi indeks, kjer smo našli število k, bi lahko s pomočjo tega indeksa izračunali, koliko elementov v seznamu je večjih in koliko elementov seznama je manjših od k. Pri tem moramo paziti, da bisekcija vrne enega od indeksov elementov, ki so enaki k; če se to število v seznamu pojavi več kot enkrat, ni zagotovljeno, da bodo dobili prvo pojavitev (ali zadnjo, ali sredinsko).

Zgornji program ima časovno zahtevnost $O(n \log n + q \log n)$, ker za iskanje vsakega elementa potrebujemo $\log n$ korakov. Običajen postopek bi za isto delo potreboval $O(qn)$ operacij, kar je v primeru, da sta tako n in q velika, hudo prepočasno.

14.2. Implementacija v standardni knjižnici

Bisekcija je kratek algoritem, ki ga lahko hitro spišemo (v zgornjem bloku kode na primer porabi le 12 vrstic kode, če ne štejemo komentarjev in praznih vrstic), včasih pa je spisati cel algoritem zamudno, hkrati pa se lahko hitro zmotimo. K sreči se lahko v veliko primerih izognemo pisanju algoritma, in namesto tega uporabimo implementacijo iz standardne knjižnice C++. Konkretno sta nam na voljo funkciji `lower_bound` in `upper_bound`, ki jima podamo kazalca na začetek in konec seznama ter iskano število, funkciji pa potem na seznamu poiščeta dano število. Obe imata časovno zahtevnost $O(\log n)$, prav tako pa za obe potrebujemo vključiti knjižnico `<algorithm>`.

Funkcija `lower_bound` vrne kazalec na prvi element seznama, ki je večji ali enak iskanemu številu, funkcija `upper_bound` pa vrne kazalec na prvi element, ki je strogo večji kot iskano število. Če funkciji ne najdeta takšnih števil, namesto tega vrneta kazalec na konec seznama. Poglejmo si, kako bi zgornji primer rešili z uporabo funkcije `lower_bound`.

```
1  // seznam mora biti nujno urejen
2  sort(arr, arr+n);
3
4  for (int i = 0; i < q; i++) {
5      int k;
6      scanf("%d", &k);
7
8      // Funkciji lower_bound podamo kazalec na začetek seznama,
9      // kazalec na mesto tik za koncem seznama in iskano število
10     int *kazalec = lower_bound(arr, arr+n, k);
11
12     // Ta korak sicer ni potreben, a je lažje razmišljati z
13     // indeksi kot s kazalci.
14     int idx = kazalec - arr;
15
16     // arr[idx] je prvo število na seznamu, ki je večje ali enako
17     // številu k; če je enako k, potem se ta pojavi, sicer pa se ne
18     if (arr[idx] == k) printf("DA\n");
19     else printf("NE\n");
20 }
```

14.3. Bisekcija po odgovoru

Namesto iskanja števila v seznamu nam bisekcija lahko poda tudi argument funkcije, pri katerem funkcija da določeno vrednost. To metodo reševanja imenujemo *bisekcija po*

odgovoru oziroma angleško *binary search the answer*.

Če želimo poiskati `x`, pri katerem je `f(x) == k`, lahko v zgornji implementaciji bisekcije nadomestimo primerjavo elementa seznama s klicem funkcije, in bo postopek deloval enako. Paziti moramo, da to deluje le, če je funkcija `f` naraščajoča, torej če za vsaki možni vrednosti `x` in `y`, kjer je `x <= y`, velja `f(x) <= f(y)`.

15. Podatkovne strukture

Pogosto si moramo za rešitev različnih problemov podatke organizirati drugače, kot da jih le shranimo v običajen seznam. Z uvedbo posebnih *podatkovnih struktur* lahko hitreje odgovarjamo na različna vprašanja o shranjenih informacijah, kar lahko uporabimo kot sestavno komponento v raznih algoritmih. Poglejmo si primer. Recimo, da imamo dan seznam števil dolžine n, nato pa bi radi q-krat ugotovili, če je neko število v teh seznamu.

```c
#include <stdio.h>

int arr[1000];

int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    for (int j = 0; j < q; j++) {
        int k;
        scanf("%d", &k);
        bool najdeno = false;
        for (int i = 0; i < n; i++) {
            if (arr[i] == k) {
                printf("Stevilo %d se pojavi v seznamu.\n", k);
                najdeno = true;
                break;
            }
        }
        if (!najdeno) {
            printf("Stevila %d ni v seznamu.\n", k);
        }
    }
    return 0;
}
```

Če si podatke le shranimo v seznam, kot zgoraj, potem bomo v najslabšem primeru za vsako od poizvedb morali preiskati celoten seznam, torej je naša rešitev $O(nq)$. Za hitrejšo rešitev bi lahko seznam na začetku uredili, nato pa pojavitve števil k iskali z bisekcijo:

```
#include <stdio.h>
#include <algorithm>
using namespace std;

int arr[1000];

int main() {
    int n, q;
    scanf("%d%d", &n, &q);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    sort(arr, arr+n);

    for (int j = 0; j < q; j++) {
        int k;
        scanf("%d", &k);
        int idx = lower_bound(arr, arr+n, k) - arr;
        if (arr[idx] == k)
            printf("Stevilo %d se pojavi v seznamu.\n", k);
        else
            printf("Stevila %d ni v seznamu.\n", k);
    }

    return 0;
}
```

Takšna rešitev ima časovno zahtevnost $O((n+q)\log n)$, saj za urejanje porabimo $O(n \log n)$ časa, vsaka izmed q poizvedb pa se izvede v $O(\log n)$ korakih. Povedano drugače smo v zgornjem primeru opazili naslednje dejstvo: Operacija „preveri, ali se dan podatek nahaja v podatkovni strukturi“ ima zahtevnost $O(n)$ v sezamu in $O(\log n)$ v urejenem seznamu. Kaj pa lahko povemo o drugih operacijah, ki bi si jih želeli uporabljati v algoritmih?

V nadaljevanju bomo za različne podatkovne strukture obravnavali časovno zahtevnost naslednjih operacij:

(O1) dodaj podatek v podatkovno strukturo,

(O2) odstrani podatek iz podatkovne strukture,

(O3) preveri, če je podatek v podatkovni strukturi.

Število podatkov, shranjenih v podatkovni strukturi, bomo označili z n.

15.1. Seznami

Najbolj osnovna podatkovna struktura je običajen seznam. V računalniškem spominu je predstavljen z enim velikih blokom spomina, ki si ga lahko predstavljamo kot n spremenljivk istega tipa, položenih eno za drugo. Ko deklariramo seznam, moramo nujno podati njegovo dolžino, da lahko prevajalnik rezervira blok primerne velikosti. Kakor smo povedali v uvodnem razdelku poglavja, za operacijo (O3) potrebujemo $O(n)$ časa. Pri določanju časovne zahtevnosti za operaciji (O1) in (O2) pa naletimo na težavo. Seznami namreč ne podpirajo dodajanja ali odstranjevanja podatkov — število elementov seznama je fiksno in določeno v kodi. Lahko pa se pretvarjamo, da seznam podpira ti operaciji tako, da si zapomnimo trenutno število podatkov v seznamu:

```
// V spremenljivki arr hranimo elemente seznama (in nekaj praznega
// prostora), v spremenljivki velikost pa število teh elementov.
// Začnemo s praznim seznamom
int arr[1000];
int velikost = 0;

// Funkcija doda element k na i-to mesto našega seznama
void dodaj_element(int k, int i) {
    // Zapisati želimo arr[i] = k, a tega še ne smemo narediti,
    // ker bomo s tem prepisali element, ki je tam trenutno shranjen.
    // Zato moramo prvo prekopirati vse elemente na eno mesto desno.
    for (int j = velikost; j > i; j--) {
        arr[j] = arr[j-1];
    }
    arr[i] = k;
    velikost++;  // dodali smo element, zato je naš seznam za 1 daljši
}

// Funkcija odstrani element na indeksu i iz našega seznama
void odstrani_element(int i) {
    // Če želimo odstraniti i-ti element, moramo poskrbeti, da bodo
    // vsi kasnejši elementi na pravem mestu, torej da bo prvi "prazen"
    // prostor na mestu arr[velikost], in ne prej
    for (int j = i; j < velikost; j++) {
        arr[j] = arr[j+1];
    }
```

```
    velikost--;
}
```

Iz zgornje kode je jasno razvidno, da imata tako operaciji (O2) kot (O3) časovno zahtevnost $O(n)$, saj moramo pri obeh v najslabšem primeru premakniti vse elemente. Opazimo pa lahko tudi, da potrebujeta operaciji „dodaj na konec seznama“ in „odstrani iz konca seznama“ le konstantno mnogo časa, saj se v obeh primerih zanka takoj konča. To bo pomembno kasneje.

Druga težava z zgornjo implementacijo pa je ta, da naš seznam lahko shrani največ 1000 elementov. Seveda bi lahko pri deklaraciji `int arr[1000]` le povečali številko tako, da bi bila dovolj velika za naše potrebe, vendar tega včasih ne želimo narediti (na primer če imamo veliko majhnih seznamov in nekaj velikih, a ne vemo v naprej, kateri od teh seznamov bodo veliki). Lahko bi si namesto fiksnega seznama `arr` hranili kazalec na del spomina, kjer je seznam shranjen, ter širino tega dela spomina, ter ga po potrebi povečali, kot spodaj:

```
int *arr = nullptr;
int kapaciteta = 0;  // število elementov, ki jih lahko shranimo v arr
int velikost = 0;  // število elementov, trenutno shranjenih v seznamu

void dodaj_element(int k, int i) {
    if (velikost == kapaciteta) {
        // Ne moremo dodati novega elementa, saj imamo rezerviranega
        // premalo spomina.
        // Na začetku si torej sposodimo več spomina, in v nov spomin
        // prekopiramo stare podatke
        int *nov_arr = (int*) malloc((kapaciteta + 1) * sizeof(int));
        for (int j = 0; j < velikost; j++) {
            nov_arr[j] = arr[j];
        }
        kapaciteta++;

        // Starega bloka spomina ne potrebujemo več, zato ga vrnemo
        // operacijskemu sistemu
        if (arr != nullptr) free(arr);
        arr = nov_arr;
    }
    // Nadaljujemo tako kot prej
    for (int j = velikost; j > i; j--) {
        arr[j] = arr[j-1];
    }
    arr[i] = k;
```

```
    velikost++;
}
```

Funkcija `odstrani_element` je v tem primeru enaka kot prej. Napisali smo *dinamični seznam*, tj. seznam, ki prilagaja svojo hitrost glede na število shranjenih elementov. S tem pa smo izgubili zgoraj omenjeno lastnost, da je dodajanje na oz. brisanje iz konca seznama $O(1)$, saj bo tekom zgornjega programa velikost vedno enaka kapaciteti, zato se bo prvi pogojni stavek v funkciji vedno aktiviral. To lastnost pa lahko še vedno v neki obliki obdržimo, če naredimo majhno spremembo:

```
void dodaj_element(int k, int i) {
    if (velikost == kapaciteta) {
        // Ne moremo dodati novega elementa, saj imamo rezerviranega
        // premalo spomina.
        // Na začetku si torej sposodimo več spomina, in v nov spomin
        // prekopiramo stare podatke
        int nova_kapaciteta = 2*kapaciteta;
        if (nova_kapaciteta == 0) nova_kapaciteta = 1;
        int *nov_arr = malloc(nova_kapaciteta * sizeof(int));
        for (int j = 0; j < velikost; j++) {
            nov_arr[j] = arr[j];
        }
        kapaciteta = nova_kapaciteta;

        // Starega bloka spomina ne potrebujemo več, zato ga vrnemo
        // operacijskemu sistemu
        if (arr != nullptr) free(arr);
        arr = nov_arr;
    }
    // Nadaljujemo tako kot prej
    for (int j = velikost; j > i; j--) {
        arr[j] = arr[j-1];
    }
    arr[i] = k;
    velikost++;
}
```

Ko `arr` povečamo, torej njegovo kapaciteto podvojimo. Kaj pa smo s tem dejansko dosegli? Ko v seznam dodamo prvi element, ne potrebujemo ničesar kopirati. Ko dodamo drugi element, moramo prekopirati prvega. Ko dodamo tretji element, moramo prekopirati prvega in drugega. Pri četrtem elementu pa nam ni treba ničesar kopirati,

saj je `arr` že dovolj velik, da lahko četrti element takoj shranimo vanj. Za peti dodan element bomo morali spet prekopirati prve štiri, za šesti, sedmi in osmi element pa nam ne bo treba ničesar kopirati. Zapišimo še nekaj členov tega zaporedja (koliko elementov moramo prekopirati, ko dodamo nov element v seznam):

$$0, 1, 2, 0, 4, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 16, \ldots$$

Z nekaj razmisleka vidimo, da je vsota prvih n členov tega zaporedja enaka $2n$, če je n ravno potenca števila 2, oziroma n v vseh ostalih primerih. V povprečju bomo torej za dodajanje enega elementa na konec seznama porabili le $O(1)$ časa. Pravimo, da je ta operacija *asimptotično konstantna* — v povprečju potrebujemo konstanten čas, da jo izvedemo, vsake toliko pa bo operacija trajala dlje.

15.1.1. Vektorji

K sreči ne potrebujemo vsakič znova pisati svojih dinamičnih seznamov, saj so že vključeni v standardni knjižnici C++ pod imenom `vector`. Poglejmo si, kako delujejo.

```
#include <stdio.h>
#include <vector>
#include <algorithm>  // sort
using namespace std;  // nujno za vector

int main() {
    vector<int> seznam;

    for (int i = 0; i < 1000; i++) {
        seznam.push_back(i);
    }

    for (int i = 0; i < seznam.size(); i++) {
        if (seznam[i] > 500) {
            seznam[i] = 7;
        }
    }

    seznam.insert(seznam.begin() + 852, 17);
    printf("%d\n", seznam[852]);
    seznam.erase(seznam.begin() + 852);
    printf("%d\n", seznam[852]);

    sort(seznam.begin(), seznam.end());
    printf("%d\n", seznam[852]);

```

```
27      seznam.clear();
28      return 0;
29  }
```

Ko deklariramo vektor, v trikotnih oklepajih (znaka za manjše oz. večje) dopišemo še tip podatkov, ki jih bomo v vektor shranjevali. V tem primeru uporabimo kar `int`, lahko pa bi uporabili kakršenkoli drugi tip, na primer `char`, `long long`*, `vector<vector<int>>` itd. Potem nam je na voljo več metod, ki jih uporabimo tako, da napišemo ime metode za imenom spremenljivke in piko:

- metoda `size()` vrne trenutno velikost vektorja,
- metoda `push_back(x)` doda `x` na konec vektorja v $O(1)$,
- metoda `begin()` vrne iterator na začetek vektorja,
- metoda `end()` vrne iterator tik čez konec vektorja,
- metoda `insert(it, x)` doda `x` tik pred mesto `it` v $O(n)$,
- metoda `erase(it)` izbriše element na mestu `it` v $O(n)$,
- metoda `clear()` izbriše vse elemente vektorja.

Iterator je poseben tip podatkov, ki si ga lahko predstavljamo kot kazalec (ker tudi podpira običajne operacije kazalcev). Iterator dobimo iz metod `begin()` in `end()`, moramo pa ga podati kot argument metodama `insert()` in `erase()`. Iteratorjem lahko prištevamo številke, s čimer lahko dostopamo do poljubnega elementa vektorja (ali pa to naredimo tudi z oglatimi oklepaji).

15.1.2. Skladi

Sklad je podatkovna struktura, ki podpira le dve operaciji: „dodaj element na vrh sklada“ in „odvzemi element z vrha sklada“. Predstavljamo si, da elemente zlagamo na kup, iz katerega lahko jemljemo le vrhnji element, ker bi se kup sicer prevrnil.

```
1   #include <stdio.h>
2   #include <stack>
3   using namespace std;
4
5   int main() {
6       stack<int> sklad;
7
8       // Za dodajanje na sklad uporabimo metodo push()
9       sklad.push(3);
```

```
    sklad.push(7);
    sklad.push(9);
    sklad.push(4);

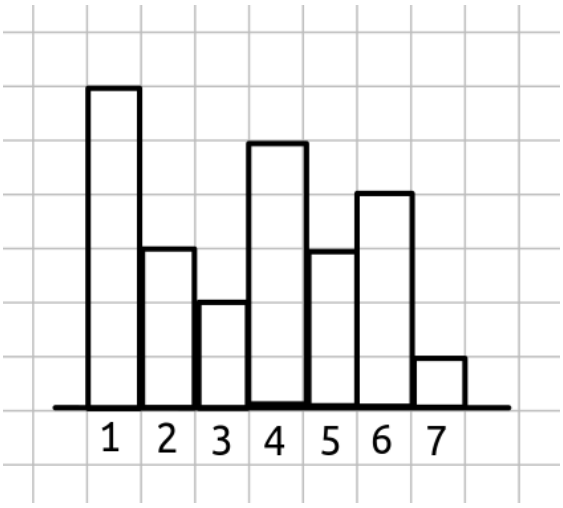
    // Z metodo top() lahko pogledamo vrhnji element sklada
    printf("%d\n", sklad.top());

    // Za odstranjevanje s sklada uporabimo metodo pop()
    sklad.pop();
    sklad.pop();

    printf("%d\n", sklad.top());
    return 0;
}
```

Poleg metod `push()` in `pop()` za dodajanje oz. odstranjevanje elementov imamo na voljo tudi metodi `top()`, ki pogleda, kaj je shranjeno na vrhu sklada, in `size()`, ki vrne število elementov na skladu. Vse metode imajo časovno zahtevnost $O(1)$.

Poglejmo si primer naloge, kjer v rešitvi uporabimo sklad. Zadan nam je naslednji problem: Dan je seznam števil, ki opisuje višino stolpnic, postavljenih v vrsto ena za drugo. Za vsako stolpnico ugotovi, katera je njej najbližja stolpnica na levi, ki je višja od nje. Za tretjo stolpnico na spodnji sliki je to stolpnica številka 2, za četrto stolpnico pa je odgovor enak 1, saj sta stolpnici 2 in 3 nižji od stolpnice 4.




```
#include <stdio.h>

```

```
int arr[1000];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        bool najdena = false;
        for (int j = i-1; j >= 0; j--) {
            if (arr[j] > arr[i]) {
                printf("%d\n", j);
                najdena = true;
                break;
            }
        }
        if (!najdena) {
            printf("Ni visje stolpnice na levi.\n");
        }
    }
    return 0;
}
```

Zgornji program bo rešil dan problem na očiten način v $O(n^2)$. Obstaja pa tudi linearna rešitev, ki se poslužuje dveh skladov. Osnovna ideja je, da si hranimo padajoče zaporedje stolpnic, ki so kandidati za najbližjo višjo stolpnico na levi. Ko pridemo do stolpnice, ki je nižja od najnižjega kandidata, lahko za njo takoj zatrdimo, da je najnižji kandidat ravno iskani odgovor. Če pa je najnižji kandidat višji od stolpnice, ki jo trenutno obravnavamo, vemo, da ne bo ta kandidat nikoli več najbližja višja stolpnica, saj je trenutna stolpnica hkrati višja od kandidata in bližje preostalim stolpnicam. Torej lahko kandidate, ki so nižji od trenutne stolpnice, kar odstranimo iz sklada.

V rešitvi imamo dve gnezdeni zanki (`while` znotraj `for`). Zakaj je potem še vedno linearna? Enostaven argument je, da bomo vsako stolpnico enkrat dodali v sklad, in jo zato največ enkrat odstranili iz sklada, torej bomo v najslabšem primeru opravili $2n$ operacij na skladu.

```
#include <stdio.h>
#include <stack>
using namespace std;

stack<int> sklad, indeksi;

int main() {
```

```
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        int t;
        scanf("%d", &t);
        while (sklad.size() > 0 && sklad.top() < t) {
            sklad.pop();
            indeksi.pop();
        }
        if (sklad.size() == 0) {
            printf("Ni visje stolpnice na levi.\n");
        } else {
            printf("%d\n", indeksi.top());
        }
        sklad.push(t);
        indeksi.push(i);
    }
    return 0;
}
```

15.1.3. Vrste

Skladom sorodne podatkovne strukture so *vrste*. Vrsta podpira le dodajanje elementov na konec in odvzemanje elementov z začetka. Tako v nasprotju v skladom pri vrstah elemente odstranjujemo v istem vrstnem redu, v katerem smo jih dodali.

V C++-u imamo na voljo podatkovni tip `queue`, ki podaja implementacijo vrste. Za dodajanje elementov v vrsto uporabljamo metodo `push()`, za odvzemanje metodo `pop()`, za dostop do prvega elementa pa metodo `front()`. Vse te operacije imajo časovno zahtevnost $O(1)$.

```
#include <stdio.h>
#include <queue>
using namespace std;

int main() {
    queue<int> vrsta;

    // Za dodajanje v vrsto uporabimo metodo push()
    vrsta.push(3);
    vrsta.push(7);
    vrsta.push(9);
```

```
    vrsta.push(4);

    // Z metodo front() lahko pogledamo prvi element vrste
    printf("%d\n", vrsta.front());  // izpiše 3

    // Za odstranjevanje iz vrste uporabimo metodo pop()
    vrsta.pop();
    vrsta.pop();

    printf("%d\n", vrsta.top());  // izpiše 9
    return 0;
}
```

Razmislimo, kako bi napisali svojo implementacijo vrste. Lahko bi uporabili razširljiv seznam iz začetka poglavja, kjer bi za dodajanje uporabili funkcijo `dodaj_element` in ji vedno podali `i = velikost-1`, za odstranjevanje elementov pa bi uporabili funkcijo `odstrani_element`, ki bi ji vedno podali argument `i = 0`. Problem je, da bi potem za odstranjevanje elementov vedno potrebovali $O(n)$ časa, saj bi vedno morali prekopirati celoten seznam.

V spodnji kodi se poslužimo naslednje ideje: Namesto, da se vrsta vedno začne na indeksu 0 tabele `arr`, bomo pustili, da se tudi indeks začetka spreminja. Potem lahko element vrste „odstranimo“ preprosto tako, da povečamo ta indeks; podatek bo sicer še vedno shranjen v spominu, a bomo mi vedeli, da ga moramo ignorirati.

```
int *arr = nullptr;
int velikost = 0;
int kapaciteta = 0;
int zacetek = 0;

int prvi_element() {
    return arr[zacetek];
}

// V vrsto vedno dodajamo elemente na konec
void dodaj_element(int k) {
    if (velikost == kapaciteta) {
        int nova_kapaciteta = 2*kapaciteta;
        if (nova_kapaciteta == 0) nova_kapaciteta = 1;
        int *nov_arr = malloc(nova_kapaciteta * sizeof(int));

        // Prekopiramo seznam na novo lokacijo, pri čemer lahko
        // nastavimo nov začetek na 0
```

```
        for (int j = 0; j < velikost; j++) {
            nov_arr[j] = arr[(zacetek+j)%kapaciteta];
        }
        kapaciteta = nova_kapaciteta;
        zacetek = 0;

        if (arr != nullptr) free(arr);
        arr = nov_arr;
    }

    // Na kateri indeks dodamo element?
    int i = (zacetek + velikost) % kapaciteta;
    arr[i] = k;
    velikost++;
}

// Iz vrste vedno odstranimo prvi element
void odstrani_element() {
    zacetek++;
    velikost--;
}
```

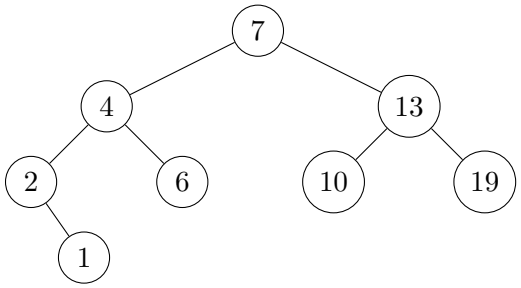
15.2. Množice

Vrnimo se k osrednjemu vprašanju, hitrosti operacij (O1), (O2) in (O3). V sezamu, vrsti in skladu smo podatke imeli organizirano zaporedoma, eden za drugim (kakor so v seznamu shranjeni v spominu), in s tem uspeli doseči konstantno časovno zahtevnost za dodajanje in odstranjevanje elementov, operacija (O3) pa še vedno traja $O(n)$, ker moramo pogledati vsak element posebej.

Če si podatke organiziramo v drevesno strukturo, lahko dosežemo hitrost $O(\log n)$ na vseh treh operacijah hkrati. Na primer, če želimo shraniti številke $1, 2, 4, 6, 7, 10, 13, 19$, jih lahko organiziramo kakor v spodnji sliki, tako da vsaka črta na levo pomeni manjšo številko, črta na desno pa pomeni večjo številko:

Potem lahko preverimo, če je število k vsebovano v naši strukturi tako, da začnemo na vrhu (v sedmici), in se prestavimo levo, če je $k < 7$, oziroma desno, če je $k > 7$. Potem postopek ponavljamo, dokler lahko, oziroma dokler ne najdemo števila k. V najslabšem primeru preverimo toliko števil, kolikor je vrstic v drevesu (na zgornji sliki 4). Če števila pametno razporedimo, lahko poskrbimo, da bo višina drevesa približno $\log n$.

Tudi operaciji dodajanja in odvzemanja števil iz take strukture lahko implementiramo tako, da bosta imeli časovno zahtevnost $O(\log n)$, vendar sta bolj kompliciran, zato se vanju ne bomo podrobno spuščali.



Takšno drevesno podatkovno strukturo v jeziku C++ uporabljajo *množice*. Vsi elementi v množici so različni — tudi če isti element dodamo večkrat, bo shranjena le ena ponovitev. Množice definiramo tako kot vektorje, na voljo pa so nam naslednje metode:

- `size()` vrne število elementov v množici,
- `clear()` izbriše vse elemente iz možice,
- `insert(x)` vstavi `x` v množico,
- `count(x)` vrne 1, če je `x` v množici, in 0 sicer,
- `erase(x)` izbriše `x` iz množice.

Poglejmo si primer uporabe.

```
#include <stdio.h>
#include <set>
using namespace std;

int main() {
    set<int> mnozica;

    mnozica.insert(1);
    mnozica.insert(2);
    mnozica.insert(2);
    mnozica.insert(4);
    mnozica.insert(6);
    mnozica.insert(6);
    mnozica.insert(6);
    mnozica.insert(7);
    mnozica.insert(10);
    mnozica.insert(10);
    mnozica.insert(13);
    mnozica.insert(19);

```

```
21      printf("V mnozici je %d elementov.\n", mnozica.size());
22
23      if (mnozica.count(10)) {
24          printf("10 je v mnozici.\n");
25      } else {
26          printf("10 ni v mnozici.\n");
27      }
28
29      mnozica.erase(10);
30
31      if (mnozica.count(10)) {
32          printf("10 je v mnozici.\n");
33      } else {
34          printf("10 ni v mnozici.\n");
35      }
36
37      return 0;
38  }
```

Če želimo pregledati vse elemente množice, tega žal ne moremo narediti z zanko kot pri seznamih, saj ne moremo dostopati do posamičnih elementov — vprašanje „Kaj je i-ti element?“ sploh nima smisla pri množicah, saj elementi niso postavljeni zaporedno. Še vedno pa lahko pregledamo vse elemente v množici z zanko naslednje oblike:

```
1   for (auto it : mnozica) {
2       printf("%d\n", *it);
3   }
```

V tem primeru je `it` iterator na elemente množice in ima tip `set<int>::iterator`. Da nebi toliko pisali, lahko uporabimo tudi `auto`.

15.2.1. Slovarji

Včasih želimo podatke obravnavati v parih (ključ, vrednost), in se hitro sklicati na vrednost pri danem ključu. To lahko dosežemo tako, da ustvarimo množico oblike

```
1   set<pair<tip_kljuca, tip_vrednosti>> mnozica;
```

kjer podamo tip za ključ in vrednosti. Potem do ključa v paru dostopamo s `par.first`, do vrednoti pa s `par.second`.

V C++ pa nam je na voljo podatkovna struktura, ki to naredi namesto nas: *slovar*. Ko ga ustvarimo, moramo podati tip za ključe in tip za vrednosti v slovarju, potem pa lahko do shranjenih vrednosti dostopamo tako, da zapišemo ključ v oglatih oklepajih.

```
#include <stdio.h>
#include <map>
using namespace std;

int main() {

    map<char, int> slovar;

    // Dodamo nekaj vrednosti
    slovar['a'] = 3;
    slovar['b'] = 6;
    slovar['g'] = 7;
    slovar['8'] = 8;

    // Vse shranjene podatke lahko pregledamo s for-each zanko
    for (auto p : slovar) {
        // tip spremenljivke p je v tem primeru pair<char, int>
        printf("%c: %d\n", p.first, p.second);
    }

    // Do podatkov lahko dostopamo z oglatimi oklepaji
    printf("Vrednost pri 'g': %d\n", slovar['g']);

    // Preverimo, če ima nek ključ shranjeno vrednost
    if (slovar.count('c')) {
        printf("'c' je v slovarju.\n");
    } else {
        printf("'c' ni v slovarju.\n");
    }

    // Podatke lahko odstranjujemo po ključih
    slovar.erase('b');

    // Na voljo sta nam tudi metodi lower_bound in upper_bound, ki
    // poiščeta prvi ključ v slovarju, večji ali enak (oziroma večji)
    // danemu iskanemu ključu.
    // Funkciji vrneta iterator, ki kaže na par (ključ, vrednost),
    // oziroma na slovar.end(), če ne najdeta ničesar.
    map<char, int>::iterator it = slovar.lower_bound('c');
```

```
40      printf(
41          "Najmanjsi kljuc, vecji od 'c': %c. Vrednost = %d\n",
42          (*it).first,
43          (*it).second);
44
45      return 0;
46  }
```

Slovarje lahko uporabimo tudi kot razpršene tabele, torej kot sezname, kjer so indeksi lahko poljubno veliki ali majhni, in kjer nekateri indeksi manjkajo. Običajnega seznama tu ne moremo uporabiti, saj bi lahko potrebovali zelo veliko tabelo — če so indeksi med -10^9 in 10^9, moramo imeti tabelo z več kot dvema milijardama elementov, ki bi zavzela več gigabajtov spomina. Velikost slovarja pa bo odvisna le od števila elementov, ki jih dejansko shranimo.

```
1   #include <stdio.h>
2   #include <map>
3   using namespace std;
4
5   int main() {
6       map<int, int> arr;
7
8       // Preberemo podatke
9       int n;
10      scanf("%d", &n);
11      for (int i = 0; i < n; i++) {
12          int x, y;
13          scanf("%d%d", &x, &y);
14          arr[x] = y;  // O(log n) namesto O(1) pri seznamu
15      }
16
17      // Sedaj imamo podatke shranjene, in lahko z njimi naredimo,
18      // kar algoritem zahteva.
19      // Za demonstracijo jih samo izpišemo, kjer vidimo tudi nekaj
20      // nove sintakse.
21      for (auto[x, y] : arr) {
22          printf("%d: %d\n", x, y);
23      }
24
25      return 0;
26  }
```

15.2.2. Neurejene množice in slovarji

Množice in slovarji, kot smo jih spoznali zgoraj, si elemente na pameten način shranijo urejene po vrsti, s čimer dosežejo časovno zahtevnost $O(\log n)$ za operacije (O1), (O2) in (O3). Obstajata pa tudi neurejeni različici teh struktur, ki podpirata iste operacije (z izjemo `lower_bound` in `upper_bound`), a imata časovno zahtevnost $O(1)$ za naštete operacije. V delovanje teh struktur se ne bomo spuščali, lahko pa seveda uporabljamo C++ implementaciji teh struktur.

```
#include <stdio.h>
#include <unordered_set>
#include <unordered_map>
using namespace std;

int main() {
    unordered_set<int> mnozica;
    unordered_map<char, int> slovar;

    for (int i = 0; i < 20; i++) {
        mnozica.insert(2*i+7);
        slovar[i] = 2*i+7;
    }

    slovar.erase(0);
    mnozica.erase(7);

    for (int i = 0; i < 20; i++) {
        if (mnozica.count(i)) {
            printf("%d\n", slovar[i]);
        }
    }

    return 0;
}
```

15.3. Kopice

Kopica ali *prioritetna vrsta* deluje podobno kot običajna vrsta, le da namesto prvo dodanega elementa vedno odstranimo največji element iz kopice. Na voljo so nam podobne

operacije kot pri vrstah: `push(x)` doda `x` na kopico, `pop()` odstrani največji element iz kopice, `top()` pa vrne največji element v kopici. Prvi dve našteti metodi imata časovno zahtevnost $O(\log n)$, zadnja pa $O(1)$. Da to dosežemo, tudi pri kopici elemente hranimo v drevesni strukturi, ki pa je malo drugačna kot struktura pri množicah.

```
#include <stdio.h>
#include <queue>  // priority_queue se najde v <queue>
using namespace std;

int main() {
    priority_queue<int> vrsta;

    vrsta.push(3);
    vrsta.push(7);
    vrsta.push(5);

    while (vrsta.size() > 0) {
        // v prioritetni vrsti uporabimo .top() namesto .front()
        printf("%d\n", vrsta.top());
        vrsta.pop();
    }

    return 0;
}
```

Če želimo, da namesto največjega elementa delamo z najmanjšim elementom na kopici, ji moramo ob definiciji dodati še nekaj posebnih argumentov:

```
priority_queue<int, vector<int>, greater<int>> vrsta;
```

15.4. Primerjava

Spoznane strukture lahko primerjamo glede na to, katere operacije podpirajo, in kako hitre so te operacije. V spodnji tabeli je prikazana časovna zahtevnost pogostih operacij na različnih strukturah. Operacija „dodaj“ pomeni, da v strukturo dodamo nek element, „poišči“ pa da preverimo, ali se dani element nahaja v naši strukturi. Ostaneta še operaciji „odstrani“ in „odstrani x“. Prva od njih iz strukture odstrani *nek* element — kateri element bo odstranjen je odvisno od podatkovne strukture (npr. pri skladu bo to vrhnji element, pri vrsti pa prvi element). Operacija „odstrani x“ pa poišče dan element

x v strukturi in ga odstrani, če je prisoten. Če podatkovna struktura ne podpira neke operacije, je to v tabeli označeno z X.

Operacija	Seznam	Vektor	Sklad	Vrsta	Množica	N. množica	Kopica
dodaj	X	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$
odstrani	X	X	$O(1)$	$O(1)$	X	X	$O(\log n)$
odstrani x	X	$O(n)$	X	X	$O(\log n)$	$O(1)$	X
poišči	$O(n)$	$O(n)$	X	X	$O(\log n)$	$O(1)$	X

A. Reševanje nalog na papir

A.1. Tekmovanje RTK

Na šolskem nivoju tekmovanja RTK ter v nekaterih skupinah tekmovanja na državnem nivoju se programerske naloge rešuje „na papir.“ To ne pomeni, da moraš rešitve dejansko z roko napisati na list papirja, temveč le, da oddaje ne bodo avtomatsko ocenjene, temveč jih bo pregledal eden od popravljavcev, rezultat pa dobiš šele po koncu tekmovanja. Pišeš lahko bodisi na računalnik, bodisi na list papirja.

Ker tvojega programa ne bo nihče prevedel ali pognal, so sprejemljive tudi rešitve, ki niso popolnoma sintaktično pravilne — za napake kot manjkajoče podpičje na koncu vrstice, napačen formatnik v `printf`, tipkarski škrati ipd. se praviloma odšteje le nekaj točk, kljub temu da tak program na običajnem tekmovanju nebi dosegel nobene točke. Med drugim to tudi pomeni, da lahko oddajaš delne rešitve. Če nalogo na primer razdeliš na dva manjša problema, enega od katerih znaš rešiti, drugega pa ne, lahko zapišeš rešitev prvega podproblema in opišeš, kako bi jo lahko kombiniral z rešitvijo drugega problema. Za tako oddajo seveda ne dobiš vseh točk, če pa je rešen del problema dovolj pomemben, pa ti popravljavci lahko dodelijo delne točke.

Ker programa ne bo pregledal računalnik, pač pa človek, moraš svojo rešitev napisati tako, da bo bralcu jasno, kaj tvoja rešitev počne. To med drugim pomeni, da uporabljaš deskriptivna imena spremenljivk, da v komentarjih programa razložiš, kakšno idejo tvoja rešitev uporablja, ter da na kratko utemeljiš, kaj naredi posamezen blok kode. Koda mora biti tudi lepo poravnana, da bo bralec lahko že s pogledom videl, kako je program strukturiran.

Včasih navodila naloge dopuščajo pisanje rešitev v psevdokodi. To pomeni, da lahko rešitev zapišeš v besedah, a jo strukturiraš kot da bi bila koda — torej tako, da lahko vsakdo, ki je vsaj malo vešč v programiranju, tvoj zapis enostavno prevede v svoj najljubši programski jezik. Ko pišeš v psevdokodi, uporabljaš standardne strukturne elemente programov (`if` stavke, `for` zanke itd.), vendar jih zapišeš v slovenščini, ne kot kodo.

A.2. Primeri

Spodaj je nanizanih nekaj primerov rešitev za naloge iz šolskega tekmovanja MladiRTK 2024, ki so dostopne na tej povezavi. Vse naloge so vredne 25 točk.

A.2.1. Pribor

Kot prvi primer si poglejmo program spodaj. Iz opisa je razvidno, da je tekmovalec prebral nalogo, in da ve, kateri funkciji se v C++ uporabljata za branje in pisanje iz standardnega vhoda in izhoda, vendar je zapisana rešitev vredna 0 točk. Tekmovalec ni nakazal, kako bi program dejansko napisal, temveč je le v svojih besedah zapisal besedilo naloge.

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);  // s scanfom preberemo število n
    // potem bom nkrat prebral število in ga shranil v int x.
    // Vsakič ko preberem večje število od velikosti trenutnega kupa
    // bom začel nov kup, in nanj zlagal žlice tako dolgo, dokler ne
    // dobim x, ki je večji od najmanjše žlice na kupu, takrat pa spet
    // začnem nov kup.
    // To ponavljam, dokler ne pridem do n, ko končam program in s
    // printf izpišem rešitev
    return 0;
}
```

Primerjajmo to z rešitvijo, ki je vredna vseh 25 točk. Ne samo, da spodnja koda pravilno reši problem, to dejstvo je tekmovalec tudi utemeljil. Koda je berljiva, poleg tega pa je na vsakem koraku utemeljeno, kakšno vlogo ima vsak del kode.

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    // Ideja: hranimo si velikost prejšnje žlice in trenutno velikost
    // kupa. Ko preberemo manjšo žlico od trenutne, bomo izpisali
    // velikost kupa in začeli nov kup s to žlico.

    int prejsnja;  // velikost prejšnje žlice
    int velikost = 1;  // število žlic v trenutnem kupu
    // Prvo žlico obravnavamo posebej, in jo bomo vedno dali na
    // začetni kup, zato lahko začnemo z velikostjo 1, in beremo
    // direktno v prejsnja
```

```
    scanf("%d", &prejsnja);

    // V zanki beremo velikosti. Ker smo eno velikost že prebrali,
    // iteriramo le do n-1
    for (int i = 0; i < n-1; i++) {
        int trenutna;
        scanf("%d", &trenutna);
        if (trenutna > prejsnja) {
            // začnemo nov kup, na katerega že položimo trenutno
            // žlico
            printf("%d\n", velikost);
            velikost = 1;
        } else {
            // žlico lahko damo na isti kup
            velikost++;
        }
        prejsnja = trenutna;
    }

    // zadnjega kupa še nismo izpisali, ker nismo nanj nikoli
    // poskusili položiti prevelike žlice
    printf("%d\n", velikost);
    return 0;
}
```

Spodnja rešitev namesto programa poda opis rešitve. Naloga je zahtevala kodo ali psevdokodo, zato za opisno rešitev odbijemo nekaj točk, a je iz spodnjega opisa jasno, da tekmovalec razume, kako bi kodo rešitve dejansko zapisal, zato še vedno prejme delne točke. Žal pa so tudi v opisu napake, za katere dodatno odbijemo nekaj točk. Tekmovalec je števila na vhodu shranil v spomin, za kar odbijemo 2 točki, kakor za to nalogo priporočajo nasveti za popravljanje, poleg tega pa je tekmovalec pozabil posebej obravnavati prvo prebrano število, za kar odbijemo 4 točke. Opisan postopek tudi ne omeni, da moramo na koncu še izpisati višino zadnjega stolpa, izpis namreč zahteva le ob obravnavi prevelike žlice. Ta rešitev je tako vredna 10 točk.

```
Prvo preberemo število n, in nato z zanko preberemo še preostala
števila na vhodu, ki si jih shranimo v seznam.
Ustvarimo še eno pomožno spremenljivko, ki predstavlja velikost
trenutnega stolpa žlic. Ker na začetku nimamo žlic, to spremenljivko
nastavimo na 0.

Potem v zanki pregledujemo seznam števil na vhodu. Vsakič, ko zanka
```

```
pride v glavni del, preverimo, če je trenutna žlica manjša ali enaka
prejšnji žlici. Če je, potem žlico damo na trenutni kup, torej
povečamo pomožno spremenljivko za 1, če pa ni, potem pa izpišemo
velikost trenutnega stolpa. Potem pomožno spremenljivko spet nastavimo
na 0 in na stolp damo to žlico, torej povečamo spremenljivko za 1.
```

Poglejmo si še eno rešitev v psevdokodi. Če bi spodnje besedilo prepisali v C++, bi bila rešitev pravilna, razen tega, da bi tekmovalec za primerjavo moral uporabiti <= namesto <, saj v navodilih piše, da lahko zložimo dve enako veliki žlici eno nad drugo. V navodilih za popravljanje je navedeno, da za tako napako odbijemo 5 točk. Načeloma bi bila potem na tekmovanju taka rešitev vredna 20 točk, a bo popravljavec po lastni presoji lahko dodatno odbil točke zaradi stila pisanja. Vse spremenljivke v kodi so enočrkovne, prav tako pa noben del rešitve ni utemeljen. Če bi imel tekmovalec v kodi napako ali bi kakšen del rešitve napisal z nejasno oznako, bi lahko zaradi pomanjkanja razumevanja popravljavec taki rešitvi dal mnogo manj točk, saj nebi bilo razvidno, da tekmovalec nalogo sploh razume.

```
preberemo n
preberemo x
nastavimo v=1
nastavimo p=x

ponavljamo (n-1)-krat
    preberemo x
    če je x < p
        v=v+1
        p=x

    če ni
        izpišemo v
        v=1
        p=x

izpišemo v
```

A.2.2. Palindromski oklepaji

Ključna opazka pri reševanju naloge je, da niz ne mora biti hkrati zrcalen in palindrom. Če je niz palindrom, mora namreč prvi znak biti enak zadnjemu, če pa je zrcalen, pa mora biti prvi znak zrcalna slika zadnjega. Navodila popravljanja za to opazko predvidijo 6

točk, tudi če rešitev ne reši naloge. Nadaljnje reševanje lahko razdelimo na dva neodvisna dela — preverjanje, ali je niz palindrom, in preverjanje, ali je niz zrcalen.

Spodnji program preveri le, če je niz palindrom, tekmovalec pa je dodal še opombo, da je niz lihe dolžine vedno bodisi palindrom bodisi navaden. Takšna rešitev bi na tekmovanju dobila 13 točk.

```c
#include <stdio.h>

// Predpostavimo, da je niz dolg največ 1000 znakov
char niz[1001];
char obrnjen[1001];

int main() {
    scanf("%[^\n]", niz);

    // Opazimo, da niz ne more biti hkrati palindrom in zrcalen,
    // saj bi potem prvi znak moral biti hkrati enak in različen
    // od zadnjega znaka. Zato lahko le posebej preverimo, če je
    // niz palindrom ali če je zrcalen

    // Za preverjanje palindromskosti bomo obrnili niz, potem
    // pa ga primerjali z originalnim
    int dolzina = strlen(niz);
    for (int i = 0; i < dolzina; i++) {
        obrnjen[i] = niz[dolzina-i-1];
    }

    if (strcmp(niz, obrnjen) == 0) {
        printf("palindrom\n");
    }

    // ne znam preveriti, če je niz zrcalen, kratek razmislek pa
    // pove, da niz lihe dolžine ne more biti nikoli zrcalen, saj
    // znak na sredini ne mora biti enak svoji zrcalni sliki.
    if (dolzina % 2 == 1) {
        printf("navaden\n");
    } else {
        printf("zrcalen izraz\n");
    }
    return 0;
}
```

V spodnji rešitvi tekmovalec pravilno določi, če je niz palindrom in če je zrcalen, a manjka razmislek o tem, da niz ne more biti hkrati palindrom in zrcalen, za kar odbijemo nekaj točk. Poleg tega je tekmovalec pozabil prebrati niz in namesto `zrcalen izraz` izpisal le `zrcalen`, za kar dodatno odbijemo dve točki. Tekmovalec prejme 20 točk.

```c
#include <stdio.h>

char niz[300];
char obrnjen[300], zrcalen[300];

int main() {
    int dolzina = strlen(niz);

    // Prvo izračunamo obrnjen in zrcalen niz
    for (int i = 0; i < dolzina; i++) {
        obrnjen[i] = niz[dolzina-i-1];

        // tudi pri zrcaljenju se niz obrne, zato lahko gledamo
        // obrnjen
        if (obrnjen[i] == '(') zrcalen[i] = ')';
        if (obrnjen[i] == ')') zrcalen[i] = '(';
        if (obrnjen[i] == '[') zrcalen[i] = ']';
        if (obrnjen[i] == ']') zrcalen[i] = '[';
    }

    // preverimo, če se niz ujema z obrnjenim in zrcalnim
    if (strcmp(niz, obrnjen) == 0 && strcmp(niz, zrcalen)) {
        printf("palindrom in zrcalen\n");
    } else if (strcmp(niz, obrnjen) == 0) {
        pritnf("palindrom\n");
    } else if (strcmp(niz, zrcalen) == 0) {
        printf("zrcalen\n");
    } else {
        printf("navaden\n");
    }

    return 0;
}
```