Upotreba gotovih algoritama sortiranja

Pri rešavanju problema često se kao jedan od koraka javlja sortiranje elemenata niza. U takvim zadacima uglavnom nije potrebno dublje poznavanje algoritama za sortiranje, jer može da se upotrebi već napisan algoritam sortiranja, koji je deo neke standardne biblioteke za dati jezik. Na žalost, standardne biblioteke za različite jezike se manje ili više razlikuju, kako po sadržaju, tako i po načinu pozivanja algoritama. Zbog toga primena gotovih bibliotečkih algoritama suštinski spada u poznavanje progrmskog jezika, a ne u znanje algoritmike. Kako se ovaj kurs bavi algoritmima a ne programskim jezicima, ova tema strogo uzevši izlazi van granica kursa. Odlučili smo se da temu ipak uvrstimo u kurs, i to iz sledećih razloga:

- Tema ima veliki praktičan značaj, a često je nedovoljno obrađena pri učenju datog programskog jezika. Ovo je razumljivo, jer programski jezici danas imaju ogromne biblioteke koje se ne mogu detaljno ispredavati. Mogućnosti biblioteke se uglavnom otkrivaju onda kada vam zatrebaju, to jest pri rešavanju problema iz date oblasti, ili na specijalizovanim kursevima koji se tom oblašću bave.
- Postoje sumnje oko toga da li pri bavljenju algoritmima i njihovom izučavanju treba koristiti gotova rešenja. Ovo je mesto i prilika da da damo kratak i jasan odgovor: **treba!** Pri rešavanju algoritamskih (i drugih) problema treba pre svega biti racionalan i truditi se da se do rešenja dođe sa manje uloženog truda i vremena. **Ako gotova rešenja mogu da pomognu u tome, njihova upotreba je legitimna i poželjna!** Ovo jednako važi za za takmičenja u programiranju i profesionalnu programsersku praksu (industrija softvera nikada ne bi doživela ovako buran razvoj da je svaki proizvođač morao sam da napiše sve što koristi). Ovo svakako ne sprečava sve one koji žele da uvežbaju pisanje nekog algoritma da i ovakve zadatke rečavaju “ručno”, što takođe ima svoj pun smisao.

Može se postaviti pitanje zbog čega se onda uče detalji naprednih algoritama sortiranja (posebna lekcija u ovom kursu), kada postoje gotova rešenja. **Ima situacija kada gotova rešenja nisu dovoljno optimizovana, ili prosto usled nekih specifičnosti ne mogu bez izmene da se primene na dati zadatak.** Tada, naravno, dolazi do izražaja dublje poznavanje materije. Uostalom, sva gotova rešenja takođe je morao neko da napiše.

Upotreba gotovih algoritama sortiranja biće ilustrovana primerima na jeziku c++. Nadamo se da će to biti dovoljno da podstakne sve one koji programiranju na drugim jezicima da potraže slična rešenja u bibliotekama tih jezika (ili ako za dati jezik nema takve podrške, da eventualno razmisle i o promeni jezika).

Jezik C++ raspolaže vrlo moćnom i obimnom bibliotekom koja se zove STL (standard template library). Ovde se ne možemo baviti opisivanjem svega onoga što STL nudi čak ni samo

u oblasti algoritmike, ali svakako želimo da podstaknemo i ohrabrimo detaljnije upoznavanje sa izvanrednim mogućnostima ove biblioteke.

Za upotrebu različitih delova STL-a potrebno je uključiti različite hedere (zaglavlja). Heder „algorithm“ omogućava upotrebu sortiranja, kao i mnogih drugih algoritama i naprednih struktura podataka.

Sortiranje prostih tipova

a) U najjednostavnijem obliku, funkcija sort ima dva argumenta. Prvi argument je adresa početnog elementa niza, a drugi je adresa **iza** poslednjeg elementa. Elementi niza mogu da budu bilo kojeg tipa za koji je definisan operator < (manje je od). Na primer, ako niz ima n elemenata, treba pisati **std::sort(&a[0], &a[n]);** ili kraće **std::sort(a, a+n);**

b) Ukoliko želimo da sortirano u nerastući (opadajući) poredak, za celobrojni niz treba pisati **std::sort(a, a+n, std::greater<int>());** Naravno, ako su elementi niza drugog tipa, njihov tip treba da stoji umesto reči int.

c) Ukoliko nam je potreban neki specijalan poredak, možemo napisati svoju funkciju za poređenje elemenata niza, takozvanim komparatorom. Fukcija kao argumente prima dve vrednosti (istog tipa kao što su elementi niza), a vraća vrednost tipa **bool**. Funkcija treba da vrati **true** ako prvi argument u uređenom nizu treba da se nalazi pre drugog, a **false** u suprotnom. Napisanu funkciju navodimo kao treći argument pri pozivu algoritma za sortiranje.

U programu koji sledi upotrebljene su sve pomenute varijante.

```
#include <iostream>   // input/output
#include <algorithm>  // sort
#include <functional> // greater

const int n = 8;
int a[n] = {5, 3, 4, 1, 3, 7, 2, 2};

void Output()
{
   for (int i = 0; i < n; i++) std::cout << a[i] << " ";
   std::cout << std::endl;
}

bool Order1(int a, int b) { return a > b; }

bool Order2(int a, int b)
{
```

```
    int ra = a % 2;
    int rb = b % 2;
    if (ra != rb) return ra < rb; // ako su brojevi razlicite parnosti, paran ide pre
    else return a < b; // ako su iste parnosti, manji ide pre
}

int main()
{
    // Rastuci poredak
    std::sort(a, a+n);
    Output();

    // Opadajuci poredak
    std::sort(a, a+n, std::greater<int>());
    Output();

    // Opadajuci poredak sa svojim komparatorom
    std::sort(a, a+n, Order1);
    Output();

    // Svi parni u rastucem poretku, pa svi neparni u rastucem poretku
    std::sort(a, a+n, Order2);
    Output();
    return 0;
}
```

Da bi algoritam sortiranja ispravno radio, bitno je da komparatorska funkcija f implementira jednu relaciju na skupu elemenata niza, koja je:

- anti-refleksivna, to jest takva da je za svako x f(x, x) netačno;
- anti-simetrična (kad god je f(x, y) tačno, tada mora biti f(y, x) netačno);
- i tranzitivna (ako je tačno f(x, y) i tačno f(y, z), onda je tačno i f(x, z)).

Slobodnije rečeno, ovo otprilike znači da pri pisanju komparatora treba koristiti stroge (<, >), a ne nestroge (≤, ≥) nejednakosti.

Sortiranje struktura

Neka je, na primer, potrebno da se tačke u ravni sortiraju sleva na desno. To znači da sve promene pozicija u nizu x koordinata moraju biti izvršene i na nizu y koordinata. Najjednostavniji način da ovo postignemo je da tačke, odnosno njihove koordinate držimo u nizu struktura umesto u dva odvojena niza. U tom slučaju se gotov algoritam sortiranja koristi na praktično isti način. Ovde su korišćena dva komparatora. Prvi od njih za tačke sa istim x

koordinatama ne određuje koja od njih dolazi pre, pa je u okviru iste x koordinate redosled neodređen (slučajan). Drugi komparator precizira da pri istoj x koordinati pre dolazi ona tačka koja ima manju y koordinatu.

```
#include <stdio.h>   // input/output
#include <algorithm>  // sort

struct Point { double x; double y; };
const int n = 8;
Point a[n] = {{2, 3}, {1, 5}, {2, 1}, {7, -3}, {4, 5}, {2, 4}, {1, 3}, {1, 7}};

void Output()
{
   for (int i = 0; i < n; i++)
      printf("(%0.3f, %0.3f)\n", a[i].x, a[i].y);
}

bool PointsOrderLeftToRight(const Point& a, const Point& b) { return a.x < b.x; }

bool PointsOrderFull(const Point& a, const Point& b)
{
   return (a.x < b.x) || ((a.x == b.x) && (a.y < b.y));
}

int main()
{
   std::sort(a, a+n, PointsOrderLeftToRight);
   Output();
   printf("\n");

   std::sort(a, a+n, PointsOrderFull);
   Output();
   return 0;
}
```

Program ispisuje:

```
(1.000, 5.000)
(1.000, 3.000)
(1.000, 7.000)
(2.000, 3.000)
(2.000, 1.000)
(2.000, 4.000)
(4.000, 5.000)
(7.000, -3.000)
```

(1.000, 3.000)
(1.000, 5.000)
(1.000, 7.000)
(2.000, 1.000)
(2.000, 3.000)
(2.000, 4.000)
(4.000, 5.000)
(7.000, -3.000)