Zapiski s priprav

Hitrost programov in asimptotična notacija

ACM priprave na tekmovanja

Različica z dne 5. december 2024

Kazalo

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.

```
int arr[100002];

int poisci_najmanjsega(int n) {
    // Poišči najmanjše število v seznamu, dolgemu n
    int min_idx = 0;
    for (int i = 0; i < n; i++) {
        if (arr[min_idx] > arr[i]) {
            min_idx = i;
        }
    }
    return arr[min_idx];
}
```

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:

```
int arr[100002];
int poisci_najmanjsega_slabsi(int n, int m) {
    // Poišči najmanjše število v seznamu, dolgemu n,
    // kjer je največje število veliko največ m
    for (int zelja = 0; zelja <= m; zelja++) {
        // zelja nam pove, kateri element si v tej iteraciji želimo
        // najti. Pogledati moramo še, da ta element dejansko je v
        // seznamu; ko pa najdemo enega, bo to najmanjši
        // (ker zelja v vsaki iteraciji narasca)
        bool je_v_seznamu = false;
        for (int i = 0; i < n; i++) {
            if (arr[i] == zelja)
                je_v_seznamu = true;
        }
```

```
        if (je_v_seznamu)
            return zelja;
    }
}
```

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++) {
```

```
3       if (n % i == 0) {
4           printf("%d ni prastevilo\n", n);
5           preklicano = true;
6           break;
7       }
8   }
9   if (!preklicano)
10      printf("%d je prastevilo\n", n);
```

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*