Napredno urejanje

December 18, 2024

1 Napredno urejanje

Kot smo videli do sedaj, so imeli vsi "naravni" algoritmi za urejanje kvadratno časovno zahtevnost. To pomeni, da imamo resno težavo že, če bi želeli urediti dva milijona prebivalcev Slovenije. Izkaže pa se, da lahko problem urejanja rešimo veliko bolj učinkovito.

[1]:
```
#include <vector>
#include <iostream>
#include <algorithm>
#include <random>
using namespace std;

typedef vector<int> VectorInt;
typedef array<VectorInt,3> VectorInt3;
```

Ker imajo zapiski težave s kompleksnejšimi tipi, bomo uporabljali `VectorInt` kot drugo ime za `vector<int>`. Prav nam bo prišlo pa še nekaj pomožnih funkcij.

Na tem mestu lahko demonstriramo še enostavno uporabo predlog (*template*) v C++. Funkcija `print` bi izgledala skoraj enako, če imamo opravka s seznamom celih števil, decimalnih števil ali pa nizov, razlika bi bila samo v tipu. S spodnjo sintakso povemo prevajalniku, naj naredi kopije funkcije in sicer po potrebi za vse tipe, ki bodo kdaj uporabljali to funkcijo.

[2]:
```
template<typename T>
void print(const vector<T> &s) {
    for (T x : s) cout << x << " ";
    cout << endl;
}
```

[3]:
```
VectorInt concat(VectorInt a, VectorInt b) {
    a.reserve(a.size()+b.size());
    a.insert(a.end(), b.begin(), b.end());
    return a;
}
```

[4]:
```
VectorInt random_numbers(int n, int x=1000000) {
    default_random_engine rnd(123);
    VectorInt v;
    for (int i=0;i<n;i++) v.push_back(rnd()%x);
```

```
    return v;
}
```

1.1 Napredni urejevalni algoritmi

Še vedno se bomo ukvarjali z algoritmi, ki temeljijo na medsebojnih primerjavah elementov. Ogledali si bomo primere algoritmov, ki dosežejo časovno zahtevnost $O(n \log n)$.

1.1.1 Urejanje z zlivanjem (*mergesort*)

Ta algoritem razdeli elemente seznama na prvo in drugo polovico. Rekurzivno uredi vsako polovico na enak način, nato pa združi dva urejena seznama (iz prve in druge polovice) v skupen urejen seznam.

Najprej si oglejmo, kako bi združili dva urejena seznama v enega samega. Na vsakem koraku preverimo najmanjša (prva) elementa v obeh seznamih in v združen seznam dodamo manjšega od njiju ter ga odstranimo iz seznama.

[5]:
```
VectorInt merge(VectorInt a, VectorInt b) {
    int i=0, j=0;
    VectorInt c;
    while (i<a.size() || j<b.size()) {
        if (i<a.size() && j<b.size()) {
            if (a[i]<=b[j]) c.push_back(a[i++]);
            else c.push_back(b[j++]);
        } else if (i<a.size()) c.push_back(a[i++]);
        else c.push_back(b[j++]);
    }
    return c;
}
```

Zlivanje seznamov je sicer poučno, vendar je dovolj pogosto, da je našlo svoje mesto tudi kot funkcija `merge` v knjižnici `algorithms`.

Algoritem je od tu naprej precej enostaven. Seznam razdelimo na pol, rekurzivno uredimo vsako polovico in združimo rezultata.

[6]:
```
VectorInt mergesort(VectorInt sez) {
    int n=sez.size();
    if (n<=1) return sez;
    VectorInt levo(sez.begin(), sez.begin()+n/2);
    VectorInt desno(sez.begin()+n/2, sez.end());
    levo = mergesort(levo);
    desno = mergesort(desno);
    return merge(levo, desno);
}
```

[7]:
```
vector<int> sez = {5,3,4,6,2,7,1};
sez = mergesort(sez);
```

```
print(sez);
```

```
1 2 3 4 5 6 7
```

[8]:
```
vector<int> sez = random_numbers(1000000);
sez = mergesort(sez);
if (is_sorted(sez.begin(), sez.end())) cout << "urejeno" << endl;
```

```
urejeno
```

Ker seznam vsakič razdelimo na pol, bo globina rekurzije $O(\log n)$. Na najglobljem nivoju se bodo združevali pari seznamov dolžine 1, en nivo višje pari seznamov dolžine 2, nato 4, itd. Za združevanjem bomo na posameznem nivoju potrebovali $O(n)$ časa.

Časovna zahtevnost (najslabša, povprečna, najboljša): $O(n \log n)$, $O(n \log n)$, $O(n \log n)$.

Prostorska zahtevnost je odvisna od implementacije. Zgornja ima prostorsko zahtevnost $O(n \log n)$, ker na vsakem nivoju rekurzije obstaja ena kopija vsakega elementa. To lahko enostavno izboljšamo, če ne ustvarjamo novih seznamov (ampak uporabljamo indekse za določitev podseznamov), za vse korake zlivanja pa uporabimo isto pomožno tabelo velikosti $O(n)$. Omenimo, da je možno tudi urejanje z zlivanjem izvesti povsem na mestu brez dodatnega pomnilnika, vendar je to že bolj zakomplicirano.

1.1.2 Hitro urejanje (*quicksort*)

Algoritem hitrega urejanja se loti urejanja tako, da razdeli elemente seznama na majhne in velike. Majhni bodo na začetku seznama, veliki pa na koncu. Seznam majhnih in velikih pa lahko vsakega zase rekurzivno uredimo na enak način. S tem smo v posameznem koraku opravili samo manjši del urejanja: elemente smo razdelili na majhne in velike. Če to ponovimo rekurzivno, pa bomo na koncu uspešno uredili seznam.

Kako naj razdelimo (*partition*) seznam na majhne in velike elemente? Idealno bi bilo, če bi jih lahko razbili na enako veliki skupini, vendar to izgleda kot ravno tako težek problem. Izbrali bomo enostavnejšo strategijo. Iz seznama, ki ga urejamo, si izberimo neko (naključno) število (*pivot*). Lahko je to kar prvi element. Elemente, ki so manjši, bomo razglasili za majhne, tiste, ki so večji, pa za velike. Imamo pa še tretjo skupino, in to so elementi, ki so enaki pivotu.

[9]:
```
VectorInt3 partition(VectorInt sez) {
    int pivot = sez[0];
    VectorInt majhni, enaki, veliki;
    for (int i=0; i<sez.size(); i++) {
        if (sez[i]<pivot) majhni.push_back(sez[i]);
        else if (sez[i]>pivot) veliki.push_back(sez[i]);
        else enaki.push_back(sez[i]);
    }
    VectorInt3 p = {majhni, enaki, veliki};
    return p;
}
```

[10]:
```
VectorInt quicksort(VectorInt sez) {
    if (sez.size()<=1) return sez;
    auto [majhni, enaki, veliki] = partition(sez);
    VectorInt urejeni_majhni = quicksort(majhni);
    VectorInt urejeni_veliki = quicksort(veliki);
    return concat(concat(urejeni_majhni, enaki), urejeni_veliki);
}
```

[11]:
```
vector<int> sez = {5,3,4,6,2,7,1};
sez = quicksort(sez);
print(sez);
```

```
1 2 3 4 5 6 7
```

Razmislimo, kako učinkovit je ta postopek? Recimo, da imamo srečo, in izbiramo elemente tako, da seznam vedno razpade na dve enako veliki skupini majhnih in velikih. V tem primeru bomo imeli $O(\log n)$ nivojev rekurzije. Na vsakem nivoju pa se bomo ukvarjali z $O(n)$ elementi. Na prvem nivoju z eno skupino n elementov, na drugem nivoju z dve skupinama velikosti $n/2$ itd. S posemezno skupino nimamo prav veliko dela, v enem prehodu jih razdelimo med manjše in večje. Skupaj bomo torej naredili $O(n \log n)$ operacij.

[7]:
```
vector<int> sez = random_numbers(1000000);
sez = quicksort(sez);
if (is_sorted(sez.begin(), sez.end())) cout << "urejeno" << endl;
```

```
urejeno
```

Izkaže se, da naša predpostavka, da bomo imeli vedno srečo pri izbiri delilnega elementa, ni tako slaba. Tudi pri naključnem izbiranju, bosta velikosti seznamov malih in velikih elementov v nekem smiselnem razmerju. Če bi bilo razmerje vedno npr. 1:2 (namesto 1:1), to še vedno vodi do enake časovne zahtevnosti. Tako je pričakovana (povprečna) časovna zahtevnost enaka tisti v najboljšem primeru.

Časovna zahtevnost (najslabša, povprečna, najboljša): $O(n^2)$, $O(n \log n)$, $O(n \log n)$.

Prostorska zahtevnost je odvisna od implementacije. Zgornja koda zaradi preglednosti porabi $O(n \log n)$ prostora. Postopek pa lahko implementiramo tudi na mestu s prestavljanjem elementov znotraj seznama, kar zmanjša prostorsko zahtevnost na $O(n)$. V sledečem primeru bomo za pivot izbrali zadnji element, nato pa preuredili preostale tako, da bodo na začetku manjši elementi, njihovo število pa bomo hranili v spremenljivki m. Funkcija `quicksort2` uredi seznam med indeksoma i in j, vključno z i-tim in brez j-tega.

[12]:
```
void quicksort2(VectorInt &sez, int i, int j) {
    if (j-i<=1) return;
    int m=0, pivot=sez[j-1];
    for (int k=i;k<j;k++) {
        if (sez[k]<pivot) {
            swap(sez[i+m],sez[k]);
            m++;
        }
```

```
    }
    swap(sez[i+m], sez[j-1]);
    quicksort2(sez, i, i+m);
    quicksort2(sez, i+m+1, j);
}
```

[13]:
```
vector<int> sez = {5,3,4,6,2,7,1};
quicksort2(sez, 0, sez.size());
print(sez);
```

```
1 2 3 4 5 6 7
```

Pozor: zgornja implementacija ima resno težavo v določenem primeru. Če so vsa števila enaka, bo namreč časovna zahtevnost $O(n^2)$. Kako bi lahko odpravili?

Če primerjamo algoritma *mergesort* in *quicksort*, prvi razdeli elemente na leve in desne in večino dela z zlivanjem naredi po zaključku rekurzivnega urejanja, drugi pa jih razdeli na majhne in velike, kar zahteva večino dela z razdelitvijo pred rekurzivnim urejanjem manjših delov.

1.1.3 Urejanje s kopico (*heapsort*)

Urejanje s kopico je pravzaprav izboljšava navadnega urejanja z izbiranjem (*selection sort*). Namesto, da bi vsakič znova iskali najmanjši element med še neurejenimi, lahko ta korak pohitrimo. To dosežemo tako, da hranimo neurejene elemente v posebni podatkovni strukturi, ki nam omogoča učinkovito iskanje in odstranjevanje najmanjšega elementa v njej. Točno temu je namenjena kopica (*heap*). Več o tem kdaj drugič.

1.2 Praksa

Kateri algoritmi pa se uporabljajo v praksi, npr. v standardnih knjižnicah programskih jezikov, kot so C, C++, Java, Python, itd. Običajno gre za neke kombinacije pristopov, saj se različni algoritmi obnesejo različno dobro na manjših ali večjih primerih.

- C ponuja funkcijo `qsort`, kjer je že iz imena očitno, da gre za quicksort.
- C++ uporablja t.i. *introsort*, ki je pravzaprav quicksort v kombinaciji še z dvema drugima algoritmoma. Če med urejanjem velikost seznama pade pod neko mejo, se uporabi navaden insertion sort. Če rekurzija preseže neko vnaprej definirano globino, pa se od tam naprej uporabi heapsort.
- Python uporablja *timsort*, ki je kombinacija mergesorta in insertion sorta.
- Java uporablja različne pristope za urejanje primitivnih tipov in za urejanje drugih objektov. Za prve uporablja različico quicksorta, za druge pa različico timsorta.

1.3 Urejanje brez primerjav

Do sedaj smo urejali elemente v okviru zelo splošnih omejitev, ki nam omogočajo samo primerjave med pari elementov. Včasih pa lahko izkoristimo tudi kakšno drugo lastnost podatkov, ki jih urejamo.

1.3.1 Urejanje s štetjem (*counting sort*)

Recimo, da moramo uredi seznam števil, ki predstavljajo poštne številke. Ne glede na to, kako dolg bo seznam, je nabor različnih poštnih številk precej majhen. Tako lahko za vsako poštno številko preštejemo, kolikokrat se pojavi v seznamu, in jo na koncu temu primerno večkrat vnesemo v urejen seznam.

[15]:
```
void counting_sort(VectorInt &sez) {
    int m = *max_element(sez.begin(), sez.end());
    VectorInt f(m+1);
    for (int x : sez) f[x]++;
    int i=0;
    for (int x=0; x<=m; x++) {
        for (int r=0;r<f[x];r++) sez[i++]=x;
    }
}
```

[16]:
```
vector<int> sez = {1000,2000,2000,4000,2000,1000};
counting_sort(sez);
print(sez);
```

```
1000 1000 2000 2000 2000 4000
```

Časovna zahtevnost je linearna, torej $O(n+m)$, kjer je m največja možna vrednost. Ne pozabite na člen m, saj ustvarjanje tabele in iteracija čez njo ni zastonj, sploh če je števil malo, njihov razpon pa velik. Neugodna je prostorska zahtevnost, ki je odvisna od največjega elementa. Kaj pa, če vrednosti niso prikladno majhna cela števila? To težavo bomo rešili, ko se bomo pogovarjali o slovarjih.

1.3.2 Urejanje s koši (*bucket/bin sort*)

Urejanje s koši (ali vedri) je zelo splošna tehnika, iz katere izhaja veliko različnih algoritmov. Osnovna ideja algoritma je, da razdeli elemente seznama v koše glede na njihovo vrednost. Med koši obstaja urejenost od košev z manjšimi elementi proti tistim z večjimi. Pri tem se zanaša na enakomerno razporejenost elementov po koših. Vsak koš lahko nato uredimo s poljubnim urejevalnim algoritmom, ali pa rekurzivno uporabimo enak postopek razdeljevanja elementov znotraj koša.

Na primer, če uporabljamo dva koša, kjer prvi vsebuje elemente z vrednostmi z območja $[\min, \mathrm{med}]$, drugi pa $[\mathrm{med}+1, \max]$ in uporabimo rekurzivno strategijo, dobimo nekaj podobnega algoritmu *quicksort*, kjer je kot pivot (namesto nekega elementa iz seznama) izbrana srednja vrednost $\mathrm{med} = (\min + \max)/2$ med najmanjšo (min) in največjo (max) vrednostjo iz seznama.

Korensko urejanje (*radix sort*) Kot primer urejanja s koši si oglejmo še korensko urejanje. V tem algoritmu razporejamo elemente v koše glede na števke v primeru števil ali črke v primeru nizov. Obstaja več različic, mi si bomo ogledali urejanje od bolj pomembnih proti manj pomembnim znakom (MSD - Most Significant Digit) in sicer na primeru urejanja nizov po abecedi.

Nize lahko razdelimo v koše glede na njihovo prvo črko, nato pa posamezen koš uredimo po enakem postopku, le da nize sedaj delimo v koše glede na drugo črko itd. Ko so koši urejeni, rezultate enos-

tavno zložimo skupaj. Vse kar potrebujemo je tabela košev `buckets`, ki bo na mestu `buckets[c]` hranila seznam nizov, ki imajo na trenutno relevantnem mestu črko `c`. Relevantno mesto bomo hranili v argumentu `r` in ga povečevali v rekurzivnih klicih. Dodatno pa hranimo še koš prekratkih besed (`done`), ki sploh nimajo `r`-te črke.

[8]:
```
void radix_sort(vector<string> &sez, int r=0) {
    if (sez.size()<=1) return;
    vector<string> buckets['z'-'a'+1], done;
    for (string x : sez) {
        if (r>=x.size()) {
            done.push_back(x);
        } else {
            int b = x[r]-'a';
            buckets[b].push_back(x);
        }
    }
    int i=0;
    for (string s : done) sez[i++] = s;
    for (int b=0; b<='z'-'a'; b++) {
        radix_sort(buckets[b], r+1);
        for (string s : buckets[b]) sez[i++] = s;
    }
}
```

[9]:
```
vector<string> sez = {"bab","a","a","aabab","aa","ba","z","az"};
radix_sort(sez);
print(sez);
```

```
a a aa aabab az ba bab z
```

Časovna zahtevnost zgornjega algoritma je $O(nd)$, če je d največja dolžina niza. Enako velja za prostorsko zahtevnost, saj na vsakem izmed d nivojev hranimo v koših vseh n elementov. Upoštevati pa moramo tudi prazne koše, ki zasedajo prostor. Teh je lahko precej. Zato je boljša ocena prostorske zahtevnosti $O(nda)$, kjer je a velikost abecede (če je konstantna, to lahko zanemarimo). V vsakem izmed $O(nd)$ klicev funkcije namreč alociramo a košev.

1.4 Dvojiško iskanje (*binary search*)

Zakaj bi sploh želeli urejati sezname? Zato, da lahko v njih učinkovito iščemo stvari. To pa počnemo z dvojiškim iskanje (bisekcijo). Ko iščemo neko vrednost v urejenem seznamu, jo lahko primerjamo z nekim elementom in če je iskana vrednost manjša od izbranega elementa, moramo nadaljevati na levi strani, sicer pa na desni. Če vedno izberemo srednji element, bomo velikost seznama na vsakem koraku prepolovili in tako potrebovali $O(\log n)$ korakov, da najdemo element oz. ugotovimo, da ga ni v urejenem seznamu.

Ideja je zavajujoče enostavna in pogosto vodi do nepravilnih rešitev. Oglejmo si eno tako.

[4]:
```
bool bisekcija_narobe(VectorInt sez, int x) {
    // nastavimo levo in desno mejo
```

```
    int levo=0, desno=sez.size()-1;
    while (1) {
        // primerjamo s srednjim elementom
        int i = (levo+desno)/2;
        // popravimo meje
        if (x < sez[i]) levo = i-1;
        else desno = i+1;
        // ce smo nasli element, ali so se meje prekrizale, ustavimo iskanje
        if (sez[i] == x || desno < levo) break;
    }
    // ce so meje smiselne, smo ga nasli, sicer ga ni
    return levo <= desno;
}
```

S to rešitvijo je narobe cel kup stvari:

- Popravljanje mej bi moralo biti ravno obratno. Če je iskani element manjši od srednjega, moramo premakniti desno mejo in obratno.
- Iskanje v praznem seznamu se sesuje, ker se vedno izvede vsaj ena iteracija iskanja.
- Največjega elementa ne bomo nikoli našli, ker se takrat, ko ga najdemo, tudi prekrižajo meje. To pa je naše merilo, ali smo našli element ali ne.
- Časovna zahtevnost ni $O(\log n)$, ampak $O(n)$ zaradi kopiranja seznama, ko pokličemo funkcijo.

[5]:
```
bool bisekcija(VectorInt &sez, int x) {
    int levo=0, desno=(int)sez.size()-1;
    while (levo<=desno) {
        int i = (levo+desno)/2;
        if (sez[i] == x) return true;
        else if (x < sez[i]) desno = i-1;
        else levo = i+1;
    }
    return false;
}
```

Oglejmo si malo težjo različice naloge. V urejenem seznamu bomo iskali mesto, kamor bi morali vanj vstaviti nek nov element, da se bo ohranjala urejenost. Če obstaja več takih mest, ker imamo več enakih števil, ga želimo vstaviti na najmanjše mesto. Npr. v seznam `{2,3,7,7,8,10,10,10}` bi število 7 želeli vstaviti na indeks 2.

Pri implementaciji bisekcije in tudi drugih algoritmov moramo biti bolj sistematični, da se izognemo napakam. To storimo tako, da v iteracijah vzdržujemo neke lastnosti, ki jim rečemo *invariante*. V našem primeru imamo v urejenem seznamu nekaj števil, ki so manjša, nato pa števila, ki so večja ali enaka `{<, <, >=, >=, >=, >=, >=, >=}`. Iščemo mejo med tema dvema območjema. Uporabljali bomo indeksa `loin hi`, kjer bo prvi ves čas kazal na neko manjše, drugi pa na večje ali enako število. Za inicializacijo teh dveh kazalcev, si lahko predstavljamo, da imamo pred seznamom na indeksu -1 vrednost $-\infty$, za njim pa ∞. Nato ju bomo v več korakih bisekcije bližali in ko bosta sosednja, smo našli iskano mejo, ki je takrat shranjena v `hi`.

[6]:
```
int lokacija(VectorInt &sez, int x) {
    int lo=-1, hi=sez.size();
    while (hi-lo>1) {
        int i = (lo+hi)/2;
        if (sez[i] < x) lo = i;
        else hi = i;
    }
    return hi;
}
```

[7]:
```
vector<int> sez = {2,3,7,7,8,10,10,10};
cout << lokacija(sez, 7) << endl;
```

```
2
```

Sedaj, ko to znamo, povejmo še, da C++ to funkcionalnost že ponuja v knjižnici `algorithm` s funkcijo `lower_bound`, ki vrne iterator na iskano meso. Funkcija `upper_bound` pa bi med enakovrednimi mesti za vstavljanje vrnila največje.

[8]:
```
vector<int> sez = {2,3,7,7,8,10,10,10};
cout << lower_bound(sez.begin(), sez.end(), 7) - sez.begin() << endl;
```

```
2
```

K dvojiškemu iskanju se bomo ponovno vrnili, ko se bomo pogovarjali o tehniki *deli in vladaj*.