Osnovno urejanje

December 18, 2024

1 Urejanje

V tem poglavju si bomo ogledali različne algoritme urejanja (sortiranja), od povsem neuporabnih, do enostavnih in vse do najbolj naprednih.

Pri urejanju imamo podano neko zaporedje elementov, ki ga želimo preurediti v vrstni red, ki bo ustrezal neki meri urejenosti. Če imamo opravka s števili, nam je že na pogled takoj očitno, kako jih je treba urediti, računalniku pa žal ne. Zato si oglejmo primer s seznamom imen `Tine, Ana, Miha, Mojca`. Imena lahko uredimo po abecedi (`Ana, Miha, Mojca, Tine`), lahko pa jih uredimo po dolžini od krajših proti daljšim (`Ana, Miha, Tine, Mojca`). V tem drugem primeru vrstni red niti ni enolično določen. Enako dober bi bil vrstni red, kjer bi zamenjali Miho in Tineta. Lahko pa imena oseb uredimo glede na njihovo starost in dobimo čisto drugačen vrstni red.

Najprej se bomo posvetili algoritmom, ki temeljijo na medsebojnih *primerjavah* elementov. Tak urejevalni algoritem si lahko za določanje vrstnega reda elementov v urejenem seznamu pomaga samo s primerjavami dveh elementov (npr. A in B), kjer dobi odgovor, ali se mora element A nahajati pred elementom B v iskanem urejenem vrstem redu.

1.1 Neuporabni urejevalni algoritmi

Pri urejanju pravzaprav iščemo neko preureditev elementov seznama, ki bo zadoščala pogojem urejenosti. Zanima nas torej neka permutacija, ki nam da urejen seznam. En zelo neučinkovit način je, da enostavno preverimo vse permutacije. Temu postopku bomo rekli urejanje s permutacijami, znan pa je tudi kot *bogosort*, *permutation sort*, *snail sort*.

Za preverjanje vseh permutacij nam bo prišla prav funkcija za generiranje naslednje permutacije `next_permutation` iz knjižnice `algorithm`. Kasneje pa nam bo za generiranje naključnih permutacij prav prišla funkcija `shuffle` iz iste knjižnice in generator naključnih števil iz knjižnice `random`.

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

[2]:
```
int uredi_perm(vector<string> &sez) {
    vector<int> p;  // permutacija
    for (int i=0;i<sez.size();i++) p.push_back(i);
    int st=0;
```

```
    // preizkusimo vse permuacije
    do {
        st++;
        // iz permutacije sestavimo pripadajoc "urejen" seznam
        vector<string> urejen(sez.size());
        for (int i=0;i<sez.size();i++) {
            urejen[i] = sez[p[i]];
        }
        // preverimo urejenost seznama
        bool je_urejen = true;
        for (int i=0;i+1<sez.size();i++) {
            if (urejen[i] > urejen[i+1]) je_urejen = false;
        }
        // ustavimo iskanje, ce smo nasli resitev
        if (je_urejen) {
            sez = urejen;
            break;
        }
    } while (next_permutation(p.begin(),p.end()));
    return st;
}
```

[3]:
```
vector<string> sez={"Tine", "Ana", "Miha", "Mojca"};
uredi_perm(sez);
for (string ime : sez) cout << ime << endl;
```

```
Ana
Miha
Mojca
Tine
```

Funkcijo `uredi_perm` smo dopolnili tako, da vrača še število obravnavanih permutacij `st`, ki nam bo prišlo prav kasneje. Kako pa deluje `next_permutation`? Permutacije bi lahko generirali rekurzivno, obstaja pa tudi lep iterativen postopek, ki sestavi naslednjo permutacijo. Kogar zanima, si lahko ogleda blog in stran na wikipediji, mi pa nadaljujemo z urejanjem.

Namesto sistematičnega preverjanja vseh možnih permutacij, bi lahko generirali naključne permutacije. Če je naš naključni generator pošten, bomo zagotovo nekoč našli pravo permutacijo (povsem slučajno). Tokrat bomo preurejali kar vhodni seznam brez uporabe pomožne permutacije.

[4]:
```
int uredi_rand(vector<string> &sez) {
    default_random_engine rnd;  // generator nakljucnih stevil
    int st=0;
    while (1) {
        st++;
        // nakljucno premesamo seznam
        shuffle(sez.begin(),sez.end(),rnd);
        // preverimo urejenost seznama
        bool je_urejen = true;
```

```
        for (int i=0;i+1<sez.size();i++) {
            if (sez[i] > sez[i+1]) je_urejen = false;
        }
        if (je_urejen) break;
    }
    return st;
}
```

[5]:
```
vector<string> sez={"Tine", "Ana", "Miha", "Mojca"};
uredi_rand(sez);
for (string ime : sez) cout << ime << endl;
```

```
Ana
Miha
Mojca
Tine
```

Razmislite, kako bi napisali svojo funkcijo **shuffle**, ki bo naključno premešala seznam. Idealno bi bilo, če so vse permutacije enako verjetne.

Kateri izmed zgornjih algoritmov je boljši - deterministični ali naključni? V najslabšem primeru se nam lahko zgodi, da bo imel naključni algoritem res nesrečo in zelo dolgo ne bo odkril pravega vrstnega reda. Ali pa bo ravno obratno in ga bo uganil zelo hitro. Kaj pa v povprečju? Tega se lahko lotimo eksperimentalno in preštejemo število permutacij, ki jih oba algoritma obravnavata. Če imamo n imen, je vseh možnih permutacij $n!$ (n fakulteta). Poskusimo z $n = 7$ in naredimo 100 poskusov urejanja naključno premešanega seznama z obema algoritmoma.

[15]:
```
vector<string> sez={"Tine", "Ana", "Miha", "Mojca", "Joze", "Katja", "Vid"};
default_random_engine rnd(123);
int st_p=0, st_r=0;
int k=100;
for (int it=0; it<k; it++) {
    shuffle(sez.begin(), sez.end(), rnd);
    vector<string> sez1 = sez, sez2 = sez;  // kopiji seznama za urejanje
    st_p += uredi_perm(sez1);
    st_r += uredi_rand(sez2);
    assert(sez1==sez2);
}
cout << "deterministicni: " << (double)st_p/k << endl;
cout << "nakljucni:       " << (double)st_r/k << endl;
```

```
deterministicni: 2671.32
nakljucni:       4929.98
```

Zanimivo, deterministični se v povprečju izkaže za boljšega. Vseh permutacij je $7! = 5040$. Deterministični po v povprečju našel pravo permutacijo nekje na polovici, kakšne prej, kakšne pa kasneje. Naključni pa jih obravnava dvakrat več, približno toliko, kolikor je vseh permutacij. Zakaj je temu tako? Razmislite, koliko metov kocke potrebujete v povprečju, da boste vrgli 6 pik. Če je x pričakovano število metov, velja $x = 1 + \frac{1}{6} \cdot 0 + \frac{5}{6} \cdot x$, torej $x = 6$. Tu imamo opravka z $n!$-strano kocko. Do rezultata bi se lahko torej dokopali tudi analitično namesto eksperimentalno.

1.2 Osnovni urejevalni algoritmi

Oglejmo si nekatere osnovne urejevalne algoritme, ki bodo služili tudi kot primeri za vajo prej obravnavanih konceptov računske zahtevnosti. Pri urejevalnih algoritmih se včasih posebej obravnava računsko zahtevnost glede na število narejenih primerjav med elementi. To je še posebej smiselno, če je primerjava netrivialna. Mi se bomo omejili na primerjanje enostavnih tipov, in bomo ocenjevali časovno zahtevnost glede na število osnovnih operacij.

Veliko bomo izpisovali vsebino seznamov, zato si pripravimo pomožno funkcijo.

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

1.2.1 Urejanje z izbiranjem (*selection sort*)

Gre za najbolj enostavno strategijo, ki jo običajno izberejo ljudje. Iz seznama, ki ga želimo urediti, bomo izbrali najmanjši element, ga odstranili in ga postavili na prvo mesto urejenega seznama, ki ga tako gradimo. To ponavljamo, dokler nam ne zmanjka vhodnega seznama, pri tem pa smo po vrsti od najmanjšega do največjega elementa zgradili urejen seznam.

Urejanje na mestu Hitro lahko ugotovimo, da nam ni treba vzdrževati dveh seznamov, ampak lahko na podoben način prerazporedimo elemente kar v vhodnem seznamu. Temu rečemo urejanje na mestu. Najmanjši element zamenjamo s prvim in ga tako premaknemo na prvo mesto. Ponovimo postopek samo s seznamom od drugega mesta naprej itd.

Vzdržujemo invarianto, da je v i-tem koraku na začetku seznama postavljenih prvih i elementov urejenega zaporedja, preostali elementi pa so še neurejeni. V vsakem koraku urejeni del povečamo za en element, ki ga postavimo na indeks i.

[3]:
```
void selection_sort(vector<int> &s) {
    int n=s.size();
    print(s);
    for (int i=0; i<n; i++) {  // iscemo i-ti najmanjsi element
        int m=i;  // indeks najmanjsega elementa med neurejenimi
        for (int j=i+1; j<n; j++) {
            if (s[j]<s[m]) m=j;
        }
        swap(s[i], s[m]);
        print(s);
    }
}
```

[4]:
```
vector<int> sez = {7,2,5,1,2,9,3};
selection_sort(sez);
```

```
7 2 5 1 2 9 3
1 2 5 7 2 9 3
1 2 5 7 2 9 3
```

```
1 2 2 7 5 9 3
1 2 2 3 5 9 7
1 2 2 3 5 9 7
1 2 2 3 5 7 9
1 2 2 3 5 7 9
```

Pri prostorski zahtevnosti lahko opazujemo celotno porabo prostora, ki je $O(n)$, ali pa samo količino dodatnega prostora (poleg vhodnih podatkov), ki je $O(1)$. V nadaljevanju se bomo držali prve interpretacije.

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

Stabilnost Zanimivo vprašanje je, ali algoritem ohranja vrstni red enakih elementov, kar imenujemo stabilnost. To postane smiselno v primeru urejanja npr. imen oseb po njihovi starosti. Kakšen bo vrsti red Ane in Jana, če sta enako stara? Bo tak, kot je bil v vhodnem seznamu, ali se lahko zgodi, da ju algoritem premeša?

Urejanje z izbiranje je v zgornji obliki *nestabilen* algoritem, ker lahko pri zamenjavi najmanjšega elementa (na indeksu m) z elementom, ki mu je v napoto (na mestu i), pokvarimo ta vrstni red.

Stabilnost lahko vedno dosežemo s tem, da vhodni seznam elementov x_i zamenjamo s seznamom parov (x_i, i), ki vključujejo še indeks, in uredimo tega. Pri primerjavi parov pride najprej do primerjave prvega dela para, v primeru enakosti pa se primerja drugi del.

1.2.2 Urejanje z vstavljanjem (*insertion sort*)

Tudi tu postopoma gradimo vedno večje urejeno zaporedje. Namesto, da bi iskali element, ki paše na naslednje mesto (kot smo to počeli pri urejanju z izbiranjem), bomo naslednji element postavili na pravo mesto. Po vrsti bomo jemali elemente iz vhodnega zaporedja in vsakega posebej vstavili v novo nastajajoče urejeno zaporedje.

Tako kot prej, lahko tudi to izvedemo na mestu. Na vsakem koraku imamo urejeno zaporedje na prvih $i-1$ mestih, v preostanku pa je še neurejeno vhodno zaporedje. V tem stanju bomo i-ti element vstavili na pravo mesto tako, da bomo konec urejenega zaporedja, ki je večji od i-tega elementa, zamaknili in naredili prostor zanj.

Vzdržujemo invarianto, da je v i-tem koraku urejenih prvih i elementov (kar ni nujno tudi prvih i elementov končnega urejenega seznama). V vsakem koraku povečamo dolžino urejenega dela z vstavljanjem naslednjega elementa v seznamu.

[7]:
```
void insertion_sort(vector<int> &s) {
    int n=s.size();
    print(s);
    for (int i=1; i<n; i++) {
        int x=s[i];
        int j=i-1;
        while (j>=0 && s[j]>x) {
            s[j+1]=s[j];
            j--;
        }
        s[j+1]=x;
```

```
        print(s);
    }
}
```

[8]:
```
vector<int> sez = {7,2,5,1,2,9,3};
insertion_sort(sez);
```

```
7 2 5 1 2 9 3
2 7 5 1 2 9 3
2 5 7 1 2 9 3
1 2 5 7 2 9 3
1 2 5 7 2 9 3
1 2 2 5 7 9 3
1 2 2 5 7 9 3
1 2 2 3 5 7 9
```

Prostorska zahtevnost: $O(n)$

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

1.2.3 Mehurčno urejanje (*bubble sort*)

V tem algoritmu bomo zaporedje uredili samo z zamenjavami sosednjih elementov, zato je včasih imenovano tudi *urejanje z zamenjavami*. Pravzaprav je ideja zelo preprosta: dokler obstaja kakšen par, ki je narobe urejen, ga najdemo in zamenjamo. Kljub temu bomo malo bolj sistematični. Pare sosednjih elementov bomo pregledovali po vrsti. Ko pridemo do konca seznama, pa se bomo vrnili nazaj na začetek. Če kdaj naredimo celoten prehod, ne da bi naredili kakšno zamenjavo, lahko zaključimo.

[9]:
```
void bubble_sort(vector<int> &s) {
    int n=s.size();
    print(s);
    bool change = true;
    while (change) {
        change = false;
        for (int i=0;i+1<n;i++) {
            if (s[i]>s[i+1]) {
                swap(s[i],s[i+1]);
                change = true;
            }
        }
        print(s);
    }
}
```

[14]:
```
vector<int> sez = {7,2,5,1,2,9,3};
bubble_sort(sez);
```

```
7 2 5 1 2 9 3
2 5 1 2 7 3 9
2 1 2 5 3 7 9
```

```
1 2 2 3 5 7 9
1 2 2 3 5 7 9
```

Pravilnost tega postopka že ni več tako očitna, kot v prejšnjih primerih. Bomo res vedno prišli do urejenega seznama, ali se lahko algoritem zatakne v kakšnem neurejenem stanju? In koliko prehodov potrebuje v najslabšem primeru?

Opazimo lahko, da algoritem v prvem prehodu z zamenjavami premakne na konec največji element, nato drugega največjega na predzadnje mesto itd. Med tem premikanjem pa poskrbi za še malo sprotnega urejanja preostalih elementov. Sedaj je jasno, da je algoritem pravilen in da potrebuje največ $n-1$ prehodov. Če jih naredimo n, pa tudi ne bo škode. Sedaj ga lahko še nekoliko skrajšamo, da je res enostaven, čeprav malo manj učinkovit.

[15]:
```
void bubble_sort_n(vector<int> &s) {
    int n=s.size();
    for (int it=0;it<n;it++) {
        for (int i=0;i+1<n;i++) {
            if (s[i]>s[i+1]) swap(s[i],s[i+1]);
        }
    }
    print(s);
}
```

[16]:
```
vector<int> sez = {7,2,5,1,2,9,3};
bubble_sort_n(sez);
```

```
1 2 2 3 5 7 9
```

Oglejmo si še računske zahtevnosti prve različice algoritma, ki zaključi, čim je rezultat urejen.

Prostorska zahtevnost: $O(n)$

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