Deli in vladaj

December 18, 2024

1 Deli in vladaj

Pristop deli in vladaj (*Divide and Conquer*) smo že srečali pri dvojiškem iskanju, hitrem urejanju (*quick sort*) in urejanju z zlivanjem (*merge sort*). Gre za preprosto idejo, da problem razdelimo na več manjših podproblemov, te rešimo rekurzivno po enakem postopku, nato pa združimo dobljene rezultate manjših problemov v rešitev večjega problema. Umetnost pa je v podrobnostih, kako razbiti problem in kako združevati rešitve, da bo celoten postopek res učinkovit.

Oglejmo si to na primeru računanja vsote seznama, ki vsebuje n števil. Seznam razbijemo na levo in desno polovico, rekurzivno izračunamo njuni vsoti, ter ju nato preprosto seštejemo. Enostavno. Kaj pa učinkovito? Navadno seštevanje v zanki ima časovnost zahtevnost $O(n)$. Če smo ustvarjali nove kopije za levo in desno polovico, je ta rešitev pravzaprav slabša, ker ima časovno zahtevnost $O(n \log n)$. Če smo za podsezname uporabljali indekse, pa tudi nismo nič na boljšem. Časovna zahtevnost je še vedno $O(n)$, samo vrstni red seštevanja elementov se je spremenil.

Omenimo nekaj klasičnih primerov algoritmov, ki temeljijo na pristopu deli in vladaj, vendar jih v okviru APS1 ne bomo utegnili obravnavati:

- množenje velikih števil (Karatsuba, FFT)
- množenje matrik (Strassen),
- najbližji par točk v ravnini
- konveksna ovojnica

1.1 Krovni izrek

Običajno razbijemo problem velikosti n na podprobleme velikosti n/b. Rekurzivno moramo rešiti a takih podproblemov. Običajno je $a \leq b$, ni pa nujno. Poleg tega pa za razbitje in združevanje rešitev potrebujemo $f(n)$ operacij:

- dvojiško iskanje: $b = 2, a = 1, f(n) = O(1)$
- quick/merge sort: $b = 2, a = 2, f(n) = O(n)$

Za izračun števila operacij imamo torej rekuzivno formulo $T(n) = aT(n/b) + f(n)$, pri čemer je $T(n) = O(1)$ za dovolj majhen n. Števili a in b sta konstanti, ki nista odvisni od n-ja. Gre za družino rekurzivnih funkcij, za katere nam krovni izrek (tudi mojstrova metoda) v določenih primerih navaja rešitve.

Primera $b = 2, a = 1$ in $b = 2, a = 2$ smo že analizirali. Oglejmo si še primer $b = 2, a = 4$ za npr. $n = 8$.

- Na začetnem (ničtem) nivoju imamo 1 problem velikosti n.
- Na prvem nivoju dobimo a problemov velikosti n/b.

- Na i-tem imamo a^i problemov velikosti n/b^i.

Število nivojev je $\log_b n$, torej je listov tega rekurzivnega drevesa $a^{\log_b n} = n^{\log_b a}$. Eksponent označimo z $c = \log_b a$, ker bo pomemben v nadaljevanju.

Če je funkcija količine dela na posameznem nivoju $f(n)$ dovolj majhna, predstavlja velikost rekurzivnega drevesa glavni del števila izvedenih operacij, čas $f(n)$ pa je zanemarljiv. Če pa je količina dela $f(n)$ dovolj velika funkcija, je glavnina operacij izvedena na začetnem nivoju v korenu, ker problem nato razpade na manjše podprobleme, ki imajo "zanemarljivo" majhno količino dela v primerjavi s korenom.

Za običajni konstanti $b = 2, a = 2$ si oglejmo nekaj primerov funkcije $f(n)$.

- $f(n) = \log n$: V korenu imamo $\log n$ dela, v otrocih ponovno $2 \log n/2 = \log n$, … Skupaj torej $O(n + \log^2 n) = O(n)$, ker prevladuje velikost rekurzivnega drevesa.
- $f(n) = n$: Tega že poznamo. Imamo $\log n$ nivojev in na vsakem nivoju n dela, skupaj torej $O(n \log n)$.
- $f(n) = n^2$: V korenu imamo n^2 dela, v otrocih $2(n/2)^2 = 1/2n^2$, v vnukih $4(n/4)^2 = 1/4n^2$. Vsota je $O(n^2)$, ker prevladuje delo v korenu.

Krovni izrek (*master theorem*) nam poda rešitve rekurzivne enačbe $T(n) = aT(n/b) + f(n)$ pri konstantah $a \geq 1, b > 1$ za tri skupine enačb glede na razmerje med $c = \log_b a$ in funkcijo $f(n)$. Velikostni red funkcije $f(n)$ bomo primerjali z n^c in ločili tri primere.

1. $f(n) = O(n^{c-\epsilon}) \quad \Rightarrow \quad T(n) = \Theta(n^c)$
   Če je čas za združevaje "manjši" od n^c, je velikost rekurzivnega drevesa (n^c) prevladujoča vrednost.

2. $f(n) = \Theta(n^c) \quad \Rightarrow \quad T(n) = \Theta(n^c \log n)$
   Če sta vrednosti "enaki", dobimo dodaten logaritemski faktor. Obstaja še natančnejša formulacija tega primera:
   $f(n) = \Theta(n^c \log^k n), k \geq 0 \quad \Rightarrow \quad T(n) = \Theta(n^c \log^{k+1} n)$

3. $f(n) = \Omega(n^{c+\epsilon}) \quad \Rightarrow \quad T(n) = \Theta(f(n))$
   Če je čas za združevanje "večji", je to prevladujoča vrednost. Ta primer zahteva še dodaten pogoj regularnosti, ki pravi, da je količina dela v vozlišču vsaj tako velika kot količina dela v otrocih (kar je skoraj vedno res): $af(n/b) \leq kf(n)$ za dovolj velike n-je in nek $k < 1$.

Oglejmo si nekaj primerov razpolavljanja z $b = 2$:

- $a = 2, f(n) = 1$ (rekurzivno seštevanje): $c = 1$, velja 1. primer, zato je $T(n) = O(n)$
- $a = 2, f(n) = \log n$: $c = 1$, velja 1. primer, zato je $T(n) = O(n)$
- $a = 3, f(n) = n$ (Karatsuba): $c \approx 1.6$, velja 1. primer, zato je $T(n) = O(n^{1.6})$.
- $a = 1, f(n) = 1$ (dvojiško iskanje): $c = 0$, velja 2. primer, zato je $T(n) = O(\log n)$.
- $a = 2, f(n) = n$ (quick/merge sort): $c = 1$, velja 2. primer, zato je $T(n) = O(n \log n)$.
- $a = 2, f(n) = 2^n$: $c = 1$, velja 3. primer, zato je $T(n) = 2^n$.

Primeri, kjer si ne moremo pomagati s krovnim izrekom:

- $T(n) = 1/2\, T(n/4) + n$: $a < 1$ nima smisla, rešujemo pol problema velikosti $n/4$?
- $T(n) = 2\, T(n/1) + n$: $b = 1$, zato se problem sploh ne zmanjšuje.
- $T(n) = 3\, T(n/2) - n^2$: Delo $f(n)$ ne more biti negativno.
- $T(n) = n/2\, T(n/2) + n$: $a = n/2$ ni konstanta.
- $T(n) = 2\, T(n/2) + n/\log\log n$: Tega ne pokriva noben izmed treh primerov.

Posplošitev krovnega izreka na primere, kjer imamo opravka s podproblemi različnih velikosti (niso vsi enako veliki n/b), je znana kot Akra-Bazzi metoda.

1.2 Primeri nalog

Sedaj smo opremljeni s teorijo pristopa deli in vladaj, ki jo v nadaljevanju poskusimo uporabiti na nekaj primerih.

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

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

1.2.1 Potenciranje

Izračunati želimo potenco x^n. Pri tem predpostavimo, da lahko množimo poljubno velika števila v konstantnem času (kar seveda ni res). Ali pa izračunajmo rešitev po nekem modulu M (v kolobarju ostankov), kjer lahko pri seštevanju in množenju sproti računamo z ostanki.

Če je n sod, bi nam prišla prav potenca $p = x^{n/2}$. Iskani rezultat je ravno p^2. Če pa je n lih, mu odštejemo 1 (in množimo rezultat z n) ter tako pridemo do sodega primera. Obakrat smo s konstantnim številom operacij razpolovili velikost problema, zato je časovna zahtevnost $O(\log n)$, za kar nam niti ni treba komplicirati s krovnim izrekom. Postopek se imenuje **potenciranje s kvadriranjem** (*exponentiation by squaring*).

[3]:
```
int potenca(int x, int n) {
    if (n==0) return 1;
    if (n%2==0) {
        //return potenca(x, n/2) * potenca(x, n/2);  // narobe! ... O(n)
        int p = potenca(x, n/2);
        return p*p;
    } else {
        return x*potenca(x, n-1);
    }
}
```

[4]:
```
cout << potenca(2,10) << endl;
```

```
1024
```

Pozorni moramo biti, da ne računamo vrednosti `potenca(x, n/2)` dvakrat. V tem primeru bi bila časovna zahtevnost $O(n)$, kar ni nič boljše od zaporednega množenja. Vrednost izračunamo enkrat in jo nato kvadriramo.

1.2.2 Enakomerno razbitje seznama

Podan imamo seznam n števil $a_1, a_2, \ldots, a_n$ z vsoto $V = \sum_1^n a_i$, ki ga želimo razbiti na k strnjenih podseznamov (ki so lahko tudi prazni). Želimo si, da je razbitje tako, da so si vsote podseznamov čim bolj podobne. Idealno bi bilo, če bi imel vsak podseznam vsoto V/k, vendar to ni vedno mogoče. Odločili smo se, da bomo to dosegli tako, da bomo zahtevali, da je največja vsota v posameznega podseznama čim manjša (minimiziramo maksimalno vsoto). Kakšno je optimalno razbitje?

Za primer vzemimo seznam `12, 8, 3, 5, 4, 13, 5, 3, 7` in $k = 3$. Vsota je 60, zato bi bilo idealno, če bi naredili skupine po 20. Prva dva elementa se ravno seštejeta v 20, zato bi ju bilo smiselno dati v svojo skupino. Potem nam ostane še dilema glede meje med drugo in tretjo skupino, kjer lahko preizkusimo obe meji okoli vsote 20. Smo s tem požrešnim razmislekom prišli do optimalne rešitve? Nismo. Prvo skupino se splača podaljšati, da pride do lepše delitve med drugo in tretjo. Optimalno razbitje je `(12, 8, 3), (5, 4, 13), (5, 3, 7)`, kjer so vsote 23, 22 in 15.

Pogosto so odločitveni problemi lažji od optimizacijskih. Je neka konkretna meja v sprejemljiva? Ali obstaja razbitje na k kosov, katerih vsota ne presega v? Večja kot je meja za vsoto, lažji je problem. Če obstaja razbitje z mejno vsoto v, obstaja tudi pri meji $v+1$ (veljavno je isto razbitje). In obratno, če pri meji v ne obstaja, potem ne obstaja tudi pri $v-1$. Iščemo mejo med situacijama, kjer razbijte še obstaja in kjer ne. To lahko poiščemo z dvojiškim iskanjem. Pravzaprav delamo dvojiško iskanje po možnih rešitvah v in preverjamo, ali so sprejemljive.

Kako ugotovimo, ali obstaja veljavno razbitje pri neki mejni vsoti v? Poiskali bomo razbitje s čim manj kosi, ki ne presegajo vsote v (vedno lahko dodamo kakšnega praznega, da jih bo točno k). Tega se lahko lotimo na požrešen način. Prvi kos naj bo največja predpona seznama, ki še ne preseže vsote v. To bo vedno vodilo do neke optimalne rešitve. Recimo, da ne bi, in bi moral biti prvi kos krajši (daljši očitno ne more biti). Potem bi lahko v tej predpostavljeni optimalni rešitvi premaknili nekaj elementov iz drugega kosa v prvega. Vemo, da je v prvem kosu še prostor, z zmanjševanjem drugega kosa pa tudi ne pokvarimo rešitve. Požrešno strategijo lahko torej uporabimo za določanje vsakega kosa znova. Če s tem nismo presegli k kosov, je mejna vsota v sprejemljiva, sicer pa ne.

Razmislimo še o časovni zahtevnosti opisanega postopka. Za dvojiško iskanje meje v bomo potrebovali $O(\log V)$ korakov. Za določanje sprejemljivosti posamezne meje pa $O(n)$. To je skupaj $O(n \log V)$.

[5]:
```
int partition(vector<int> a, int k) {
    int total=0, largest=0;
    for (int x : a) {
        total+=x;
        largest = max(largest, x);
    }
    int lo=largest-1, hi=total;  // lo=infeasible, hi=feasible
    while (lo+1<hi) {
        int lim=(lo+hi)/2;
        int sum=0, chunks=1;
        for (int x : a) {
            if (sum+x<=lim) sum+=x;  // extend last chunk
            else { chunks++; sum=x; }  // start new chunk
        }
        if (chunks<=k) hi=lim;
```

```
        else lo=lim;
    }
    return hi;
}
```

[6]:
```
vector<int> a={12,8,3,5,4,13,5,3,7};
cout << partition(a, 3) << endl;
for (int k=1;k<=a.size();k++) {
    cout << k << ": " << partition(a, k) << endl;
}
```

```
23
1: 60
2: 32
3: 23
4: 17
5: 15
6: 13
7: 13
8: 13
9: 13
```

1.2.3 K-ti element

V problemu izbire k-tega elementa (*selection problem*) imamo podan (neurejen) seznam n števil $a_1, a_2, \ldots, a_n$. Zanima nas, katero število je k-to po velikosti oz. bi bilo na k-tem mestu, če bi seznam uredili.

Seznam lahko uredimo in preverimo, kateri element konča na k-tem mestu. Časovna zahtevnost je odvisna od časa urejanja in je v splošnem $O(n \log n)$. Smo lahko kaj bolj učinkoviti? Vsakakor moramo preveriti vseh n elementov, morda pa lahko izboljšamo faktor $\log n$.

Ker bomo uporabili podoben prostop kot pri hitrem urejanju (*quick sort*), se algoritmu, ki ga bomo opisali, reče hitro izbiranje (*quick select*). Izbrali bomo delilni element (*pivot*) in razdelili števila na manjša (ali enaka) in večja. Naj bo manjših števil m. Če je $k <= m$, moramo k-tega iskati med manjšimi. Sicer pa moramo med večjimi poiskati $(k-m)$-tega.

Ob predpostavki, da nam seznami razpadajo na prbližno enako velike skupine, bo pričakovana časovna zahtevnost $O(n + n/2 + n/4 + ...) = O(n)$. S tem se strinja tudi krovni izrek pri $b = 2, a = 1, f(n) = n$ (3. primer).

V C++ je ta funkcionalnost že na voljo kot funkcija `nth_element` iz knjižnice `algorithm`, ki delno uredi seznam tako, da je n-ti element na pravem mestu, pred njim so samo manjši ali enaki elementi, za njim pa večji ali enaki.

[7]:
```
vector<int> v = {3,5,2,8,1,10,2,3,8,5,1};
nth_element(v.begin(), v.begin()+4, v.end());
print(v);
sort(v.begin(),v.end());
print(v);
```

```
2 1 1 2 3 5 3 5 8 10 8
1 1 2 2 3 3 5 5 8 8 10
```

1.2.4 Štetje inverzij

V seznamu n števil $a_1, a_2, \ldots, a_n$ je inverzija par indeksov i in j ($i < j$), za kateri velja, da sta pripadajoči števili v seznamu narobe urejeni ($a_i > a_j$). Zanima nas, koliko inverzij vsebuje podani seznam? Seveda lahko preverimo vse pare indeksov, vendar ima to kvadratno časovno zahtevnost.

Prilagodili bomo algoritem urejanja z zlivanjem (merge sort). Poleg urejanja podseznamov naj funkcija izračuna še število inverzij v njem (pred urejanjem). Recimo, da smo seznam razbili na levo in desno polovico, ter rekurzivno rešili manjša problema. S tem smo dobili število inverzij v levi polovici in urejeno levo polovico, ter enako za desno polovico. Urejeni polovici znamo zliti v urejeno celoto. Kaj pa inverzije?

Inverzije v levi in desni polovici seštejemo, vendar nam manjkajo še tiste inverzije, kjer je eno število v levi, drugo pa v desni polovici. Za vsako število x iz leve polovice bomo izračunali število inverzij, v katerih nastopa - koliko je v desni polovici manjših števil od x? To lahko učinkovito izračunamo med zlivanjem obeh polovic. Recimo, da smo že zlili l števil iz leve polovice in d iz desne ter je naslednje na vrsti število x iz leve polovice. Pred njim je v zlitem urejenem seznamu že d manjših števil iz desne polovice, s katerimi je formiral inverzije in jih prištejemo k rezultatu.