Pozresni algoritmi

December 18, 2024

1 Požrešni algoritmi

Pogosto lahko sestavimo rešitev nekega problema z zaporedjem korakov, pri čemer se na vsakem koraku odločimo za eno izmed več možnih izbir. Pri požrešnem (*greedy*) pristopu reševanje se na vsakem koraku odločimo za izbiro, ki v tistem trenutku izgleda najbolj obetavno. S takim načinom bomo najbrž našli kar spodobno rešitev, pa bo ta tudi optimalna? Odvisno od problema, zato moramo znati razlikovati, kje in zakaj take strategije delujejo in kdaj ne.

Recimo, da želimo na spodnjem zemljevidu priti iz levega zgornjega vogala v desni spodnji vogal s čim manj premiki. Na zemljevidu znaki `.` predstavljajo prosta polja, znaki `#` pa zasedena. Očitno bomo gradili rešitev postopno po premikih. Na vsakem koraku se bomo odločili za eno izmed največ 3 možnih smeri (ne bomo se premikali nazaj, od koder smo prišli). Smiselna mera obetavnosti premika je razdalja sosednjega polja od cilja. Prvo dilemo imamo na polju (3,3), kjer bolje izgleda premik navzdol, kar nas premakne bližje k cilju, kot premik navzgor. Vendar nas to vodi do slabše rešitve zaradi kasnejših komplikacij na poti, ki jih v trenutku požrešne izbire ne upoštevamo. Ni si težko zamisliti tudi primera, kjer taka izbira sploh ne bi vodila do rešitve.

```
.#..#.
.#....
...#..
##.#.#
...#.#
.###..
.#....
...##.
```

V nadaljevanju si bomo ogledali več primerov problemov ter dokazov (ne)pravilnosti požrešnih strategij za njihovo reševanje, s čimer boste razvili nekaj intuicije in zdrave skeptičnosti glede uporabe požrešnih strategij. S požrešnimi strategijami se bomo ponovno srečali tudi kasneje pri algoritmih na grafih (Dijkstra, Prim, Kruskal). Požrešne strategije se običajno dobro obnesejo na enostavnih problemih, medtem ko na kompleksnejših z njimi dobimo neko suboptimalno oz. nepravilno rešitev. Posebej zanimivi pa so primeri, kjer nas v navidez kompleksnih problemih pripeljejo požrešne rešitve do optimalnega rezultata.

[1]:
```
#include <cstdio>
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
```

```
using namespace std;
```

[2]:
```
typedef pair<int,int> PII;
typedef vector<pair<int,int>> VII;
typedef vector<vector<pair<int,int>>> VVP;
```

1.1 Bencinske črpalke

Začnimo s potovalnim problemom polnjenja avta na bencinskih črpalkah (*car fueling*). Z avtom želimo potovati do K kilometrov oddaljenega cilja. Pri tem vemo, da se vzdolž poti nahaja N črpalk, ki so od našega izhodišča oddaljene $0 < x_1 < x_2 < \cdots < x_n < K$ kilometrov. Velikost posode za gorivo oz. doseg našega avta s polnim tankom je T kilometrov (z delno polnim pa sorazmerno manj). Pot bomo začeli s polnim tankom goriva. Je cilj sploh dosegljiv? Kakšno je najmanjše število postankov na črpalkah, da prisemo na cilj?

Primer: $K = 950, T = 400, x = [200, 375, 550, 750, 950]$.

Ugotovitve:

- Na črpalki je vedno smiselno povsem napolniti tank z gorivom. Če ga ne bi napolnili do vrha, bi lahko z bolj polnim tankom opravili enako pot do naslednje črpalke. Morebiten ostanek goriva pa "zlili stran" oz. tam dotočili temu primerno manj.
- Dosegljivost lahko preverimo tako, da tankamo na vsaki črpalki.
- Če je mogoče doseči naslednjo črpalko (ali cilj), lahko preskočimo tankanje na trenutni črpalki. Na naslednji črpalki lahko namreč dotočimo gorivo do nivoja, ki bi ga imeli, če bi natočili gorivo na trenutni.

[3]:
```
int crpalke(int K, int T, vector<int> x) {
    x.insert(x.begin(), 0);
    x.insert(x.end(), K);
    int doseg=T, postanki=0;
    for (int i=0;i+1<x.size();i++) {
        int razdalja=x[i+1]-x[i];
        if (doseg<razdalja) { postanki++; doseg=T; }  // po potrebi napolni
        if (doseg<razdalja) return -1;  // tudi s polnim tankom ne gre
        doseg-=razdalja;
    }
    return postanki;
}
```

Da si poenostavimo implementacijo bomo dodali začetek in konec poti kot dve dodatni črpalki. Nato se premikamo med sosednjimi črpalkami v skladu s prejšnjimi ugotovitvami. Preverimo rešitev na začetnem primeru in par drugih posebnih situacijah, kjer ne rabimo dolivati goriva, ga dolivamo povsod ali je nemogoče doseči cilj.

[4]:
```
cout << crpalke(950, 400, {200,375,550,750}) << endl;
cout << crpalke(950, 950, {200,375,550,750}) << endl;
cout << crpalke(950, 200, {200,375,550,750}) << endl;
cout << crpalke(950, 199, {200,375,550,750}) << endl;
```

```
2
0
4
-1
```

1.2 Izbira aktivnosti

Izbira med seboj neodvisnih aktivnosti iz nabora ponujenih (*activity selection*) je klasičen problem. Podanih imamo N aktivnosti, kjer se i-ta aktivnost a_i izvaja v obdobju (s_i, e_i). Izbrati moramo čim večjo podmnožico aktivnosti, za katero velja, da je presek poljubnih dveh aktivnosti prazen (aktivnost se sicer lahko začne v trenutku, ko se prejšnja konča). Ker lahko aktivnosti predstavimo z daljicami, je problem znan tudi kot *interval scheduling*.

Primer: $[(1,3),(2,4),(2,5),(4,5),(4,7),(6,7),(6,8),(7,12),(8,12),(9,10),(9,11),(11,12),(12,13)]$

Hitro pridemo na več idej, kako bi se lahko lotili problema brez preverjanja vseh podmnožic. Katere od njih pa so res pravilne?

- **najzgodnejši začetek** (*earliest start*)

Ne izgubljajmo časa s čakanjem! Razpored aktivnosti lahko sestavljamo po korakih tako, da vsakič dodamo aktivnost, ki se začne prva po zaključku trenutnega razporeda.

Protiprimer: $[(1,6),(2,3),(4,5)]$

- **najkrajši** (*shortest*)

Dolge aktivnosti zasedejo veliko časa, zato začnimo z majhnimi! Razpored sestavljamo tako, da vanj dodajamo aktivnosti od krajših proti večjim. Če za neko aktivnost ni prostora, jo preskočimo.

Protiprimer: $[(4,7),(1,5),(6,10)]$

- **najmanj konflikten** (*fewest conflicts*)

Težave so s konflikti med aktivnostmi, zato začnimo z najmanj konfliktnimi! Izračunajmo konfliktnost vsake aktivnosti in jih po vrsti poskusimo dodajati v razpored. Lahko konfliktnosti izračunamo vnaprej ali jih moramo posodabljati, ko nekatere aktivnosti že dodamo v razpored?

Protiprimer: $[(6,9),(1,3),(4,7),(8,11),(12,14),(2,5),(2,5),(2,5),(10,13),(10,13),(10,13)]$. Prvi interval ima samo dva konflikta, vendar njegova izbira vodi v rešitev s tremi intervali, primer pa lahko rešimo s štirimi.

- **najzgodnejši konec** (*earliest finish*)

Čim prej zaključimo s prvo aktivnostjo, da bomo imeli čim več časa za ostale! Med vsemi aktivnostmi, ki se začnejo ob ali po koncu trenutno zadnje izberemo tisto z najzgodnejšim koncem.

Protiprimer: ?

To izgleda obetavno. Dokažimo, da je pravilno. Recimo, da obstaja boljša optimalna rešitev, ki se na začetku strinja s požrešno, pri i-ti aktivnosti v izbranem razporedu pa pride prvič do razlike. Optimalna izbere aktivnost o, požrešna pa p. Ker požrešna vedno izbere aktivnost z najzgodnejšim koncem, velja $e_p <= e_o$. Zato se aktivnost p ne more pojaviti kje kasneje v predpostavljeni optimalni razporeditvi. Obe aktivnosti nista konfliktni s predhodnimi. Če v optimalnem razporedu zamenjamo aktivnost o z p, bo preostanek razporeda ostal veljaven, rešitev pa ne bo nič slabša.

Tako smo podaljšali del optimalne rešitve, ki se se strinja s požrešno, ne da bi jo kako poslabšali. Če ta razmislek ponovimo večkrat, bomo predpostavljeno optimalno rešitev lahko predelali v požrešno brez poslabšanja rezultata. Tudi požrešna rešitev nas torej pripelje do optimalnega rezultata.

[5]:
```
VII aktivnosti(VII a) {
    VII razpored;
    int konec=0;
    while (1) {
        int j=-1;
        for (int i=0;i<a.size();i++) {
            if (konec<=a[i].first) {  // relevanten?
                if (j==-1 || a[i].second<a[j].second) j=i;  // boljsi?
            }
        }
        if (j==-1) break;
        razpored.push_back(a[j]);
        konec=a[j].second;
        a.erase(a.begin()+j);
    }
    return razpored;
}
```

[6]:
```
VII a = {{1,3}, {2,4}, {2,5}, {4,5}, {4,7}, {6,7}, {6,8}, {7,12}, {8,12},␣
 ↪{9,10}, {9,11}, {11,12}, {12,13}};
VII r = aktivnosti(a);
printf("%d:",(int)r.size());
for (auto [s,e] : r) printf(" (%d,%d)",s,e);
printf("\n");
```

```
6: (1,3) (4,5) (6,7) (9,10) (11,12) (12,13)
```

Lahko to naredimo bolj učinkovito? Aktivnosti uredimo po njihovih koncih in jih izbiramo po vrsti, če se začetek ne seka s koncem trenutno zadnje aktivnosti. Časovna zahtevnost je tako samo $O(n \log n)$. Gre še hitreje? Če so vrednosti majhna cela števila, bi lahko uporabili urejanje s štetjem.

[7]:
```
bool cmpSecond(pair<int,int> a, pair<int,int> b) {
    return a.second < b.second;
}
```

[8]:
```
VII aktivnosti(VII a) {
    sort(a.begin(), a.end(), cmpSecond);
    VII razpored;
    int konec=0;
    for (auto [s,e] : a) {
        if (konec<=s) {
            razpored.push_back({s,e});
            konec = e;
```

```
        }
    }
    return razpored;
}
```

[9]:
```
VII a = {{1,3}, {2,4}, {2,5}, {4,5}, {4,7}, {6,7}, {6,8}, {7,12}, {8,12},␣
 ↪{9,10}, {9,11}, {11,12}, {12,13}};
VII r = aktivnosti(a);
printf("%d:",(int)r.size());
for (auto [s,e] : r) printf(" (%d,%d)",s,e);
printf("\n");
```

```
6: (1,3) (4,5) (6,7) (9,10) (11,12) (12,13)
```

Kaj pa utežena različica problema, kjer ima vsaka aktivnost poleg začetka in konca tudi svojo pomembnost in želimo namesto števila aktivnosti v razporedu maksimizirati vsoto pomembnosti? To se izkaže za malo težji problem, h kateremu se bomo vrnili kasneje pri tehniki dinamičnega programiranja.

1.3 Rezervacije učilnic

Pri problemu rezervacije učilnic (*classroom scheduling, interval partitioning*) moramo na fakulteti izvesti N predavanj, kjer posamezno predavanje poteka v časovnem intervalu (s_i, e_i). Kakšno je najmanjše število predavalnic, ki jih potrebujemo, da bomo lahko izvedli vsa predavanja?

V primerjavi s prej obravnavanim problemom izbire aktivnosti, smo morali tam izbrati čim več aktivnosti, ki jih lahko izvedemo z eno predavalnico. V tem primeru pa moramo izvesti vse, pri čemer nas zanima, najmanj koliko predavalnic potrebujemo.

Spodnji primer prikazuje razpored predavanj s štirimi predavalnicami, mogoče pa jih je izvesti tudi samo s tremi.

```
P1: (4,10), (12,15)
P2: (0,3), (4,7), (8,11)
P3: (0,7), (10,15)
P4: (0,3), (8,11), (12,15)
```

Če v kakšnem trenutku sočasno poteka več predavanj, bomo zagotovo potrebovali vsaj toliko predavalnic. Največjemu številu sočasnih predavanj bomo rekli globina (*depth*), ki predstavlja spodnjo mejo rešitve. Je to spodnjo mejo vedno mogoče doseči, ali kdaj potrebujemo več predavalnic? Če se razporejanja lotimo slabo, jih bomo seveda potrebovali več; kaj pa če jih razporedimo optimalno?

S požrešnim algoritmom bomo predavanja po vrsti glede na njihov začetek razporejali v predavalnice. Na vsakem koraku preverimo, ali je kakšna od predavalnic že prosta in lahko vanjo dodelimo trenutno predavanje. Če je takih več, izberemo katerokoli. Če take predavalnice ni, odpremo/dodamo novo predavalnico (začnemo z 0 predavalnicami) in v njo dodelimo novo predavanje.

Dokažimo, da prej opisani postopek doseže ravno globino množice predavanj, ki je spodnja meja rešitve. Recimo, da postopek potrebuje d predavalnic. Do tega pride, ko želimo nekam razporediti predavanje i z začetkom ob času $t = s_i$, vendar so vse ostale predavalnice še zasedene. To pomeni,

da imamo $d-1$ predavanj, ki se zaključijo po času t. Vsa predavanja, ki potekajo v njih, so se začela prej ali takrat kot i-to, ker jih dodajamo po vrsti. Torej so vsi njihovi začetki manjši ali enaki t. V trenutku $t+\epsilon$ torej poteka sočasno d predavanj. Če je požrešen postopek uporabil d predavalnic, je to zato, ker nekje sočasno poteka d predavanj in torej doseže spodnjo mejo.

[10]:
```
VVP predavalnice(VII predavanja) {
    sort(predavanja.begin(), predavanja.end());
    VVP urnik;
    for (auto p : predavanja) {
        auto [s,e] = p;
        int x=-1;
        for (int i=0;i<urnik.size();i++) {
            if (urnik[i].back().second<=s) { x=i; break; }
        }
        if (x==-1) urnik.push_back({p});  // odpremo novo
        else urnik[x].push_back(p);
    }
    return urnik;
}
```

[11]:
```
VII predavanja = {{4,10}, {12,15}, {0,3}, {4,7}, {8,11}, {0,7}, {10,15}, {0,3},␣
↪{8,11}, {12,15}};
VVP urnik = predavalnice(predavanja);
for (auto ucilnica : urnik) {
    for (auto [s,e] : ucilnica) printf(" (%d,%d)",s,e);
    printf("\n");
}
```

```
 (0,3) (4,7) (8,11) (12,15)
 (0,3) (4,10) (10,15)
 (0,7) (8,11) (12,15)
```

Dokazali smo, da je rešitev pravilna. Razmislimo še o njeni učinkovitosti. Razporediti moramo N predavanj enega za drugim. Pri tem pa vsakič preverimo vse odprte predavalnice. Lahko se nam zgodi, da bo vsako predavanje v svoji predavalnici, zato jih je na vsakem koraku treba preveriti $O(N)$. Časovna zahtevnost zgornje implementacije je torej $O(N^2)$.

Kako bi lahko to izboljšali? Najbolj problematičen del je iskanje proste predavalnice. Predavalnice bi lahko hranili urejene v prioritetni vrsti glede na čas zaključka zadnjega predavanja. Namesto v poljubno prosto predavalnico, bomo razporedili predavanje v tisto, ki je že najdlje prosta oz. se je čim bolj zgodaj sprostila. Če ta ni primerna, ne bo tudi nobena druga. Če uporabimo dvojiško kopico, je časovna zahtevnost $O(N \log N)$.

[12]:
```
VVP predavalnice(VII predavanja) {
    sort(predavanja.begin(), predavanja.end());
    VVP urnik;
    priority_queue<PII, VII, greater<PII>> pq;  // min-heap
    pq.push({predavanja.back().second, -1});  // dummy
    for (auto p : predavanja) {
```

```
        auto [s,e] = p;
        auto [konec, x] = pq.top();
        if (konec<=s) {
            pq.pop(); pq.push({e,x});
            urnik[x].push_back(p);
        } else {
            pq.push({e, urnik.size()});
            urnik.push_back({p});
        }
    }
    return urnik;
}
```

[13]:
```
VII predavanja = {{4,10}, {12,15}, {0,3}, {4,7}, {8,11}, {0,7}, {10,15}, {0,3},␣
 ↪{8,11}, {12,15}};
VVP urnik = predavalnice(predavanja);
for (auto ucilnica : urnik) {
    for (auto [s,e] : ucilnica) printf(" (%d,%d)",s,e);
    printf("\n");
}
```

```
 (0,3) (4,7) (8,11) (12,15)
 (0,3) (4,10) (10,15)
 (0,7) (8,11) (12,15)
```

1.4 Datoteke na traku

Pred časom trdih diskov so se podatki hranili na trakovih. Slaba stran trakov je, da je treba za dostop do podatka na mestu x prevrteti celoten trak od začetka do tega mesta. Oglejmo si problem optimalnega shranjevanja datotek na traku (*storing files on tape*). Podanih imamo N datotek, ki so opisane s pari števil $d_i = (s_i, f_i)$, kjer s_i velikost datoteke, f_i pa pogostost dostopa do nje. Ceno zapisa datotek na trak bomo ocenili z $\sum_i x_i f_i$, kjer je x_i začetno mesto zapisa datoteke. Pri tem se zapisi datotek seveda ne smejo prekrivati. Kakšen je optimalen razpored datotek in z njim povezana minimalna cena?

Primer: $d = [(60, 5), (27, 3), (1, 20), (32, 4)]$

Ugotovitve:

- Datoteke je smiselno zapisovati v strnjenem zaporedju, saj morebiten prazen prostor med njimi samo škodi.
- Ni enostavno.

Preizkusimo najprej obnašanje problema na manjših različicah. S tem dobimo tudi občutek za glavne ovire pri reševanju. Pripravimo si funkcijo za ocenjevanje razporeda in preizkusimo različne permutacije.

[14]:
```
int score(vector<pair<int,int>> d) {
    int x=0, sc=0;
    for (auto [s,f] : d) { sc+=x*f; x+=s; }
```

```
    return sc;
}
```

[15]:
```
VII d = {{60,5}, {27,3}, {1,20}, {32,4}};
cout << score(d) << endl;
sort(d.begin(),d.end());
do {
    cout << score(d) << ' ';
} while (next_permutation(d.begin(),d.end()));
```

```
2272
415 495 403 448 540 528 952 1032 1588 2783 2227 2863 1039 1084 1576 2771 2279
2816 1735 1723 2272 2908 2359 2896
```

Problem deluje zapleteno, zato najprej rešimo poenostavljene različice.

Recimo, da so vse datoteke enako dolge, npr. $s_i = 1$. Intuitivno bi rekli, da morajo biti bolj pogosto dostopane datoteke na začetku, da bo dostop do njih hiter. Naj bosta sosednji datoteki i in j, pred njima pa se nahaja x datotek. K ceni prispevata $xf_i + (x+1)f_j$. Če ju med seboj zamenjamo, bosta prispevali $xf_j + (x+1)f_i$, kar je sprememba za $f_i - f_j$. Če je negativna (kar zmanjša ceno), ko je $f_i < f_j$, ju je smiselno zamenjati, da bo bolj pogosto dostopana datoteka pred manj dostopano. S tem lahko utemeljimo, da je optimalen naraščajoč vrstni red po pogostosti dostopa.

Recimo, da imajo vse datoteke točno en dostop, torej $f_i = 1$. Intuitivno bi želeli imeti kratke datoteke na začetku, saj naredijo manj "škode" kot daljše. S podobnim argumentom o zamenjavi lahko dokažemo, da morajo biti datoteke na traku urejene naraščajoče po velikosti. Če primerjamo oba možna vrstna reda dveh sosednjih datotek i in j sta njuna doprinosa k ceni $x + (x + s_i)$ in $x + (x + s_j)$. Na ceno ostalih njuna medsebojna zamenjava nima vpliva. Vrstni red, kjer je i pred j, je torej boljši, če je $s_i < s_j$.

Obravnavajmo sedaj splošen primer, kjer opazujemo možna vrstna reda dveh sosednjih datotek na traku. Ceni dostopa sta $xf_i + (x + s_i)f_j$ in $xf_j + (x + s_j)f_i$, če bi bila datoteka j pred i. Hitro lahko izračunamo, kdaj je cena ureditve i pred j manjša od obratne. Tako pridemo do zaključka, da morajo biti v optimalnem vrstnem redu datoteke urejene naraščajoče glede na razmerje med velikostjo in pogostostjo dostopa s_i/f_j. Torej jih lahko v rešitev požrešno zložimo po vrsti od tistih z nižjim proti tistim z višjim razmerjem.

$$\begin{aligned} xf_i + (x + s_i)f_j &\leq xf_j + (x + s_j)f_i \\ s_i f_j &\leq s_j f_i \\ s_i/f_i &\leq s_j/f_j \end{aligned}$$

[16]:
```
bool cmpRatio(pair<int,int> a, pair<int,int> b) {
    return a.first*b.second < b.first*a.second;  // a.first/a.second < b.first/
 ↪b.second ... racunska napaka?
}
```

[17]:
```
int trak(vector<pair<int,int>> d) {
    sort(d.begin(), d.end(), cmpRatio);
```

```
    return score(d);
}
```

[18]:
```
cout << trak(d) << endl;
```

```
403
```

1.5 Minimizacija zamude

Pri razvrščanju z minimizacijo največje zamude (*minimum lateness scheduling*) imamo opravka z N opravili, ki jih moramo izvesti na enem računalniku. Vsako opravilo je opisano s parom $o_i = (t_i, d_i)$, ki predstavlja čas izvajanja in rok, do katerega mora biti opravilo zaključeno. Če je s_i čas začetka izvajanja, se konča ob času $f_i = s_i + t_i$. Zamuda opravila je $z_i = \max(0, f_i - d_i)$. Cilj razvrščanja opravil je *minimizirati največjo zamudo* opravila v razporedu. Minimiziramo torej $Z = \max z_i$.

Primer: $o = [(2,5), (1,2), (3,6), (2,7)]$

Očitno ni koristi od tega, da bi imel urnik kakšne proste luknje. Z odstranitvijo lukenj zagotovo ne moremo poslabšati urnika oz. maksimalne zamude, morda pa ga izboljšamo. Odločiti se moramo samo za vrstni red opravil. Namesto preverjanja vseh permutacij, se ponovno ponuja nekaj požrešnih strategij.

[19]:
```
int late(VII o) {
    int Z=0,now=0;
    for (auto [t,d] : o) {
        now+=t;
        int z=max(0, now-d);
        if (z>Z) Z=z;
    }
    return Z;
}
```

[20]:
```
VII o = {{2,5}, {1,2}, {3,6}, {2,7}};
sort(o.begin(),o.end());
do {
    cout << late(o) << ' ';
} while (next_permutation(o.begin(),o.end()));
```

```
2 1 2 3 1 3 2 1 3 6 4 6 2 3 3 6 4 6 2 3 4 6 4 6
```

- **najkrajši čas izvajanja** (*shortest processing time*)

Kratka opravila izvedemo prej, da ne bodo zamujala zaradi dolgih opravil! Kaj pa, če imajo dolga opravila kratke roke in obratno?

Protiprimer: $[(1, 100), (10, 10)]$

- **najkrajši prosti čas** (*smallest slack*)

Poleg časa izvajanja t_i moramo upoštevati tudi rok opravila d_i. Opravila izvajamo glede na naraščajoč prosti čas oz. "manevrski prostor" $d_i - t_i$!

Protiprimer: $[(1, 2), (10, 10)]$

- **najzgodnejši rok** (*earliest deadline*)

Opravila izvajamo samo glede na rok za zaključek opravila d_i!

Protiprimer: ?

Izgleda ok, pa je res optimalno? Naj bodo opravila urejena naraščajoče po rokih, torej $d_1 \leq d_2 \leq ... \leq d_N$. Recimo, da obstaja neka optimalna rešitev, ki je boljša od požrešne. V njej se zagotovo pojavljata dve sosednji opravili j in i, kjer ima prvo kasnejši rok od drugega ($d_j > d_i$); sicer bi bila ta rešitev enaka požrešni. Ob njuni zamenjavi se nova zamuda (z') vseh drugih opravil razen i in j ne spremeni. Zamuda opravila i se kvečjemu zmanjša, ker se opravilo po zamenjavi zaključi prej. Če opravilo j po zamenjavi ne zamuja, ni problema. Recimo torej, da zamuja in sicer za $z'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i = z_i$ (končata se ob enakem času; j ima manjši rok).

Vemo torej, $z'_k = z_k \quad \forall k \notin \{i, j\}, \quad z'_i \leq z_i, \quad z'_j \leq z_i$. Iz tega sledi, da je $Z' = \max z'_k \leq \max z_k = Z$. Če ti dve opravili zamenjamo med seboj, ne bomo povečali največje zamude. Če to ponavljamo, bomo prišli do lepo urejene požrešne rešitve, ne da bi povečali zamudo, kar pa je v protislovju s predpostavko, da požrešna rešitev ni optimalna. Skonstruiramo lahko namreč požrešno rešitev, ki je tako dobra ali celo boljša od predpostavljene optimalne.

[21]:
```
int zamuda(VII o) {
    sort(o.begin(),o.end(),cmpSecond);
    return late(o);
}
```

[22]:
```
cout << zamuda(o) << endl;
```

```
1
```

1.6 Dokazovanje pravilnosti

Za dokazovanje pravilnosti požrešnih strategij smo uporabiljali naslednje pogosto uporabljene vrst argumentov, ki pa seveda niso edini.

Prednost (*stay ahead*) Dokažemo, da je po vsakem koraku rešitev požrešne strategije vsaj tako dobra kot katerakoli druga. Kot primer smo obravnavali bencinske črpalke.

Zamenjava (*exchange argument*) Dokažemo, da lahko z določenimi spremembami pretvorimo predpostavljeno boljšo rešitev v tako, ki bi jo našla tudi požrešna metoda, pri tem pa ne poslabšamo njene kvalitete. Pravilnost požrešnega algoritma smo dokazali s protislovjem po naslednjem principu:

1. Predpostavimo, da obstaja optimalna rešitev, ki je boljša od požrešne rešitve. Med njimi izberemo tisto, ki se čim bolj strinja s požrešno. Torej ima mesto i, kjer se prvič razlikuje od požrešne, čim večje. Lahko bi ji rekli najbolj ekstremen protiprimer (največji, najmanjši, najkasnejši, ...) Cilj je pokazati, da obstaja še ekstremnejši, ki pa je vsaj tako dober, če ne boljši.
2. Argumentiramo, da bi lahko na tem mestu izbrali tudi požrešno potezo in zato rešitev ne bi bila nič slabša, morda pa celo boljša.

3. Našli smo protislovje, ker smo lahko skonstruirali rešitev, ki se ujema s požrešno na prvih i mestih in je enako dobra ali celo boljša od predpostavljene "optimalne", hkrati pa se od nje razlikuje kasneje. Predpostavljena optimalna rešitev torej ni bila najbolj ekstremna.
4. Predpostavka, da obstaja drugačna rešitev, ki je boljša od požrešne, torej ne drži in je požrešna rešitev zato optimalna.

Kot primer smo obravnavali izbiro aktivnosti, datoteke na traku in minimizacijo zamude.

Struktura (*structural argument*) Dokažemo neko strukturno lastnost (vrednost) optimalne rešitve, ki predstavlja mejo in dokažemo, da jo požrešna rešitev res doseže. Kot primer smo obravnavali rezervacijo učilnic.

1.7 Menjava kovancev

V blagajni imamo kovance (in bankovce) različnih vrednosti v evrih: 1, 2, 5, 10, 20, 50, 100, 200 in 500 €. Predpostavimo, da je blagajna dobro založena z vsemi vrednostmi. Blagajniki se običajno poslužujejo požrešne strategije za vrnitev določene vrednosti X s čim manjšim številom kovancev. Uporabijo največji kovanec, ki ne presega vrednosti X in nato ponovijo postopek na zmanjšani vrednosti.

Ali s tem za vsako možno vrednost X res uporabijo najmanjše število kovancev? Izkaže se, da v primeru evrskih kovancev to drži.

Ali to velja za poljuben nabor vrednosti kovancev? Hitro najdemo protiprimer, npr. plačilo 6 s kovanci $[1, 3, 4]$. Požrešna metoda bi uporabila tri kovance ($6 = 4 + 1 + 1$), optimalna pa zgolj dva ($6 = 3 + 3$).

Kako bi lahko dokazali, da za podan nabor kovancev požrešna strategija deluje za poljubno vrednost, ki jo moramo sestaviti? Tega se bomo lotili tako, da bomo preverili pravilnost požrešne strategije do neke meje in dokazali, da če deluje do tja, bo delovala tudi za vse večje. Za velike vrednosti bo optimalna rešitev izbirala največje kovance, kar pa je enako kot pri požrešni rešitvi, torej mora priti do razlike med njima pri neki manjši vrednosti.

Najprej moramo znati izračunati optimalno število kovancev za menjavo neke vrednotsi X, da lahko primerjamo, ali to število požrešna strategija doseže. To lahko naredimo s preverjanjem vseh možnih kombinacij, ali pa malo učinkoviteje, kot se bomo naučili v poglavju o dinamičnem programiranju. Predpostavimo torej, da imamo postopek, ki zna izračunati optimalno število kovancev za menjavo dane vrednosti.

Naj bodo kovanci urejeni po velikosti $a_1 < a_2 < ... < a_n$. Naj bo S najmanjši protiprimer, kjer požrešna strategija ne najde optimalne rešitve. Razmislimo, kaj lahko povemo o optimalni strategiji pri menjavi vrednosti S.

- Optimalna rešitev ne uporabi a_n. Če bi ga uporabila optimalna, bi ga tudi požrešna, zato bi se na prvem koraku rešitvi strinjali in bi moral obstajati manjši protiprimer $S - a_n$.
- Optimalna rešitev uporabi kovanec a_i manj kot a_n-krat. Če bi optimalna rešitev vzela kovanec a_i kar a_n-krat (ali še večkrat), bi bilo bolje vzeti kovanec a_n zgolj a_i-krat, s čimer dosežemo enako vrednosti.

Iz tega sledi, da je največja vrednost, ki jo optimalna strategija lahko zamenja (in se pri tem razlikuje od požrešne) enaka $U = (a_1 + ... + a_{n-1})(a_n - 1)$. Vemo torej, da mora biti najmanjši protiprimer $S \leq U$. Če preverimo rešitve do U in ne najdemo razlike, bo to veljalo tudi za vsa

večja števila. Meja U je za nabor vrednosti v evrih dovolj nizka. Obstajajo pa tudi boljše (in bolj zapletene) zgornje meje $U' < U$.