Cpp

December 18, 2024

1 C++

V C-ju že znate programirati, C++ pa vam ponuja nekaj dodatnih funkcionalnosti, ki vam bodo olajšale življenje. Večinoma preko svoje standardne knjižnice in raznih sintaktičnih bližnjic. Standardna knjižnica vsebuje številne uporabne podatkovne tipe/strukture (*containters*) in algoritme. Osnovana je bila po knjižnici Standard Template Library (STL), ki se še danes pogosto uporablja kar kot sinonim za C++ standardno knjižnico.

Prav vam bo prišla dokumentacija: - cplusplus - cppreference

1.1 Branje in pisanje

V C-ju ste navajeni branja in pisanja podatkov s knjižnico `stdin.h`in funkcijami kot so `scanf`, `printf`, `gets`, ...

C++ pa v knjižnici `iostream` ponuja t.i. tokove (*stream*). Prav nam bosta prišla `cin` (character input) in `cout` (character output). Podpirata operatorje `>>` oz. `<<`, ki poleg branja/pisanja tudi vrneta isti objekt, da lahko operatorje verižimo.

[1]:
```
#include <iostream>
```

[2]:
```
std::cout << "Zivjo!\n";
```

```
Zivjo!
```

Večina funkcionalnosti, ki jih ponuja C++ v svoji standardni knjižnici, se nahaja v imenskem prostoru `std`, do objektov v njem pa dostopamo z `std::ime`. Malo pisanja si lahko prihranimo z deklaracijo uporabe imenskega prostora `std`.

[2]:
```
using namespace std;
```

[28]:
```
int a,b;
cin >> a >> b;
cout << "vsota = " << a+b << endl;
```

```
 9 10
```

```
vsota = 19
```

1.2 Nizi

C++ ponuja podatkovno strukturo `string`, ki nam olajša delo z nizi v primerjavi s C-jem, kjer smo bili obsojeni na delo s tabelami znakov.

[3]:
```
#include <string>
```

[6]:
```
string ime, priimek;
cin >> ime >> priimek;
string oseba = priimek + " " + ime;
cout << "Pozdravljen, " << oseba << "!" << endl;
cout << "Zacetnice: " << ime[0] << priimek[0] << endl;
cout << "Dolzina imena: " << ime.size() << endl;
```

```
 Tomaz Hocevar
```

```
Pozdravljen, Hocevar Tomaz!
Zacetnice: TH
Dolzina imena: 5
```

1.3 Pari

Pari vrednosti so zelo koristni, da nam ni treba za vsako malenkost ustvarjati novih struktur ali razredov. V jezikih, kot je npr. Python, pa je koncept terk (*tuple*) še bolj prisoten. Paru moramo določiti tudi tipe vsebovanih komponent. Sintaksa z "oklepaji" oz. znaki manjše/večje je podobna tisti, ki ste je navajeni iz Jave. Do obeh elementov dostopamo preko atributov `first` in `second`.

[4]:
```
#include <utility>
```

[8]:
```
pair<int,int> xy;
xy = make_pair(10,12);
cout << xy.first << " " << xy.second << endl;
```

```
10 12
```

1.3.1 Inicializacija s seznamom

Nekatere podatkovne tipe lahko inicializiramo tudi s seznami vrednosti (initializer list). Poleg parov bomo videli primere uporabe tudi kasneje pri drugih strukturah.

[9]:
```
pair<int,int> p = {2,3};
cout << p.first << " " << p.second << endl;
```

```
2 3
```

1.3.2 Avtomatska določitev tipa

Ko začnemo uporabljati gnezdene strukture (npr. par osebe in ocene, pri čemer je oseba prav tako par sestavljen iz imena in priimka), postanejo opisi podatkovnih tipov precej dolgi. Ker zna prevajalnik med prevajanjem ugotoviti, da se nek tip ne ujema, ga lahko tudi kar določi, kar označimo s tipom `auto`.

[10]:
```
pair<pair<string,string>, int> ocena = {{"Tomaz", "Hocevar"}, 10};
auto o = ocena;
cout << o.first.first << endl;
```

```
Tomaz
```

1.4 Seznam

V C-ju smo imeli na voljo tabele fiksne velikosti. Če smo želeli hraniti seznam elementov, ki smo ga podaljševali, pa smo naleteli na manjši problem. Tega nam rešuje poatkovnih tip `vector`, ki predstavlja razširljivo tabelo (*resizable array*), podobno kot ArrayList v Javi. V njem lahko shranjujemo samo elemente enakega tipa, ki ga moramo navesti ob deklaraciji (za razliko od npr. Pythona).

[4]:
```
#include <vector>
```

[4]:
```
vector<int> v;
for (int x=1; x<=1024; x*=2) v.push_back(x);
for (int i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl;
```

```
1 2 4 8 16 32 64 128 256 512 1024
```

1.4.1 Iteratorji

Koncept iteratorjev vam je že poznan iz Jave. V C++ so iteratorji kazalci na elemente v strukturah, s katerimi se lahko premikamo po elementih v tej strukturi. Vsaka struktura ima svoj tip iteratorja (npr. ‘`vector<int>::iterator`) `in ponuja iteratorja na svoj začetek in konec` (`.begin()`in`.end()`‘).

[6]:
```
for (vector<int>::iterator it=v.begin(); it!=v.end(); it++) {
    cout << *it << " ";
}
cout << endl;
```

```
1 2 4 8 16 32 64 128 256 512 1024
```

1.4.2 For each

Iteracija čez vse elemente strukture je zelo pogosta operacija, zato je v številnih programskih jezikih na voljo tudi temu prilagojena sintaksa. V C++ je to `for (tip element : struktura)`.

[17]:
```
vector<pair<int,int>> koordinate = {{2,6},{1,4},{-2,6}};
for (pair<int,int> xy : koordinate) cout << "[" << xy.first << ", " << xy.
 ↪second << "]" << endl;
```

```
[2, 6]
[1, 4]
[-2, 6]
```

1.4.3 Razpakiranje

Dostop do posameznih elementov kompleksnejšega tipa (npr. seznama parov) je lahko kar dolg. To si lahko skrajšamo z uporabo razpakiranja (*structured binding declaration*). Sintaksa je `auto [a, b, ...] = x`.

[18]:
```
for (auto [x, y] : koordinate) cout << "[" << x << ", " << y << "]" << endl;
```

```
[2, 6]
[1, 4]
[-2, 6]
```

1.5 Vrsta, sklad, slovar, množica, povezani seznam, ...

C++ pozna še cel kup drugih podatkovnih tipov, kot so `queue`, `stack`, `map`, `set`, `list` ... Več o njih pa takrat, ko bomo obravnavali abstraktne podatkovne tipe.

[6]:
```
#include <map>
#include <set>
```

[7]:
```
map<string,int> vpisna = {{"Ana", 123}, {"Miha", 456}, {"Tine", 789}};
set<string> prisotni = {"Ana", "Tine"};
for (auto ime : prisotni) cout << vpisna[ime] << endl;
```

```
123
789
```

1.6 Reference

Referenca je drugo ime za isto spremenljivko. Označimo jo s znakom `&`. V spodnjem primeru je `y` referenca na spremenljivko tipa `int`, kar zapišemo kot `int &y`. Referenca ne more biti prazna, inicializirati jo moramo z drugo spremenljivko ob deklaraciji. Deklaracija reference brez inicializacije (`int &y;`) ali inicializacija s konstantno vrednostjo (`int &y = 9;`) nista možni.

[5]:
```
int x = 10;
int &y = x;
cout << x << " " << y << endl;
```

```
10 10
```

Če sedaj spremenimo vrednost spremenljivke `x`, se ta sprememba odraža tudi v spremenljivki `y`, in obratno.

[6]:
```
x = 11;
cout << x << " " << y << endl;
y = 12;
cout << x << " " << y << endl;
```

```
11 11
12 12
```

V C++ se argumenti funkcij *prenašajo po vrednosti*. Funkcija torej prejme kopijo podanega argumenta. V Pythonu ali Javi bi z dodajanjem elementov seznamu, ki ga sprejme funkcija, spremenili tudi zunanji seznam. V C++ temu ni tako.

[7]:
```
void izpisi(vector<int> v) {
    for (int x : v) cout << x << " ";
    cout << endl;
}
```

[8]:
```
auto podvoji(vector<int> v) {
    int n=v.size();
    for (int i=0; i<n; i++) {
        v.push_back(v[i]);
    }
    return v;
}
```

[9]:
```
vector<int> a = {1,2,3,4,5};
vector<int> b = podvoji(a);
izpisi(a);
izpisi(b);
```

```
1 2 3 4 5
1 2 3 4 5 1 2 3 4 5
```

Če želimo, lahko to funkcionalnost dosežemo s *prenosom argumentov po referenci*. Funkcija `podvoji_ref` sprejme argument po referenci, vrednost te spremenljivke spremeni in ne vrne ničesar.

[10]:
```
void podvoji_ref(vector<int> &v) {
    int n=v.size();
    for (int i=0; i<n; i++) {
        v.push_back(v[i]);
    }
}
```

[11]:
```
podvoji_ref(a);
izpisi(a);
```

```
1 2 3 4 5 1 2 3 4 5
```

Prenos po referenci se uporablja za podajanje velikih spremenljivk, da se izognemo ustvarjanju kopije. Druga pogosta uporaba je vračanje več vrednosti iz funkcije preko nastavljanja argumentov, ki so podani po referenci. Slednje smo v C-ju lahko dosegli tako, da smo funkciji podali kazalec na spremenljivko in jo nato spreminjali preko tega kazalca.

V spodnjem primeru funkcija `stat` sprejme seznam celih števil `v` po referenci, da se ne ustvari kopija po nepotrebnem, ker funkcija seznama ne spreminja. Prešteje pozitivna in negativna števila ter rezultate vpiše v argumenta `poz` in `neg`, ki sta prav tako podana po referenci.

[12]:
```
void stat(vector<int> &v, int &poz, int &neg) {
    poz=0;
    neg=0;
    for (int x : v) {
        if (x>0) poz++;
        if (x<0) neg++;
    }
}
```

[13]:
```
vector<int> st = {5, -7, 8, 9, -3, -2, -1, -4};
int poz, neg;
stat(st, poz, neg);
cout << "poz/neg: " << poz << " " << neg << endl;
```

```
poz/neg: 3 5
```

1.7 Algoritmi

Knjižnica `algorithm` ponuje cel kup uporabnih funkcij, kot so `min`, `max`, `min_element`, `swap`, `count`, ...

[13]:
```
#include <algorithm>
```

[14]:
```
cout << min(5,2) << endl;
cout << min({5,3,9}) << endl;

vector<int> v={4,7,1,8};
cout << *min_element(v.begin(), v.end()) << endl;
```

```
2
3
1
```

Verjetno najpogosteje uporabljena funkcija pa je `sort`. Funkcija `sort` sprejme iteratorja na začetek in konec seznama vrednosti, ki jih bo uredila. Tako lahko uporabljamo funkcijo `sort` na različnih strukturah, ki implementirajo pravo vrsto iteratorjev.

[19]:
```
vector<int> sez = {8,41,11,7,2};
sort(sez.begin(), sez.end());
for (int x : sez) cout << x << " ";
cout << endl;
```

```
2 7 8 11 41
```

1.7.1 Anonimne funkcije

Pogosto pišemo kratke funkcije za enkratno uporabo. Zato večina modernih programskih jezikov (Java, Python, C++, ...) pozna anonimne oz. lambda funkcije. Sintaksa v C++ je `[zunanje spremenljivke](argumenti funkcije) { vsebina }`. Če so zunanje spremenljivke prazne, to

pomeni, da lahko vsebina funkcije dostopa samo do svojih argumentov in do globalnih spremenljivk. Vse spremenljivke in argumente lahko funkcija sprejme po vrednosti ali po referenci.

V spodnjem primeru bomo ponovno uredili seznam števil, vendar jih bomo tokrat primerjali po abecedi namesto po vrednosti.

[22]:
```
sort(sez.begin(), sez.end(), [](int a, int b) {
    return to_string(a) < to_string(b);
});
for (int x : sez) cout << x << " ";
cout << endl;
```

```
11 2 41 7 8
```

1.8 Ostalo

C++ seveda ponuja še veliko več. Do tu smo predstavili samo nekaj osnov, ki nam bodo koristile pri reševanju algoritmičnih problemov v nadaljevanju.

Kogar zanima več, si lahko na spletu prebere o temah, kot so: razredi (*classes*), predloge (*templates*), pametni kazalci (*smart pointers*), niti (*threads*), izjeme (*exceptions*), preobremenjevanje operatorjev (*operator overloading*), ...

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.

Dinamicno programiranje

December 18, 2024

1 Dinamično programiranje

Dinamično programiranje je algoritmičen pristop, ki je podoben pristopu deli in vladaj. Tudi pri uporabi dinamičnega programiranja bomo razbili problem na manjše podprobleme, poiskali optimalne rešitve podproblemov in si z njimi pomagali pri rešitvi začetnega problema. Pomembne lastnosti problema, pri katerem si lahko pomagamo z dinamičnim programiranjem so:

- neodvisnost podproblemov: Posamezen podproblem lahko rešujemo neodvisno od drugih podproblemov.
- optimalna podstruktura: Optimalna rešitev problema vsebuje optimalne rešitve podproblemov.
- **prekrivanje/ponavljanje podproblemov**: To je glavna lastnost, ki jo bomo izkoristili za izboljšave in v čemer se pristop razlikuje od tehnike deli in vladaj.

Tehniko lahko enostavno povzamemo z nasvetom "ne računaj enakih stvari večkrat", v praksi pa je kljub temu nekoliko bolj zapleteno - kako to doseči, katere stvari sploh so enake, …

Pristop nima nobene veze z dinamično alokacijo pomnilnika. Poimenoval ga je njen avtor Richard Bellman. "Programiranje" se nanaša na reševanje optimizacijskega problema, poodobno kot matematično programiranje/optimizacija. Pridevnik "dinamično" pa se nanaša na različne podprobleme.

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

1.1 Fibonaccijevo zaporedje

Osnovno idejo dinamičnega programiranja si oglejmo na trivialnem primeru Fibonacijevega zaporedja, ki je definirano rekurzivno kot: $F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$. Zanima nas n-to število v zaporedju. Pri večjih n-jih bodo vrednosti zaporedja precej velike, vendar se s tem ne bomo ukvarjali in bomo zadovoljni z rezultatom, ki je posledica preliva (*overflow*).

[2]:
```
int fib(int n) {
    if (n<=1) return n;
    return fib(n-1)+fib(n-2);
}
```

[3]:
```
for (int n=0;n<10;n++) {
    cout << n << ": " << fib(n) << endl;
}
```

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
```

Vrednosti izgledajo pravilne. Hitro pa ugotovimo, da na ta način ne bomo mogli računati vrednosti že za malo večje n-je. Težava je v eksponentni velikosti drevesa rekurzivnih klicev. Listov tega drevesa, kjer je rezultat funkcije 1, je natanko F_n. Poleg tega pa imamo še liste z vrednostjo 0 in vsa notranja vozlišča. Skratka, ogromno število vozlišč oz. klicev funkcije.

[4]:
```
//cout << fib(100) << endl;  // prepocasi
```

Opazimo lahko, da se bo funkcija izvedla večkrat z istim argumentom n. Če se nismo kje zmotili, bi moral imeti vsak tak klic funkcije tudi enak rezultat. Rezultat si lahko ob prvem klicu funkcije shranimo, v kasnejših klicih pa ga samo vrnemo. To je pristop **od zgoraj navzdol** (*top-down*), ki je znan tudi pod imenom **memoizacija** (*memoization*, brez "r"). Funkcija se bo torej za vsak možen argument izvedla natanko enkrat, ob ostalih klicih pa bo takoj vrnila vrednost, česar niti ne bomo šteli kot klic funkcije. Število klicev funkcije bo torej $O(n)$, čas izvedbe posameznega klica funkcije pa $O(1)$. Rešitev ima časovno in prostorsko zahtevnost $O(n)$.

Za ugotavljanje, ali je bil nek podproblem že rešen ali ne, lahko v tem primeru izkoristimo kar vrednost 0, saj bomo kot izračunane rezultate vpisovali samo večja števila. V splošnem pa bi lahko imeli eno tabelo, ki bi nam povedala, ali je bil nek podproblem že rešen, ter drugo tabelo, ki bi hranila dejanske rezultate. Zaradi enostavnosti bomo uporabili dovolj veliko fiksno tabelo dovolj. Namesto tega bi lahko uporabili katerokoli implementacijo slovarja, ki bi imel kot ključ argumente, ki predstavljajo opis podproblema, za pripadajočo vrednost pa njegovo rešitev.

[5]:
```
const int N=10000;
int memo[N+1];  // memoizacijska tabela
```

[6]:
```
int fib2(int n) {
    if (n<=1) return n;
    if (memo[n]!=0) return memo[n];
    memo[n]=fib2(n-1)+fib2(n-2);
    return memo[n];
}
```

Če smo malo bolj sistematični, lahko rešujemo podprobleme v takem vrstnem redu, da imamo rešitve manjših podproblemov vedno že rešene, ko jih potrebujemo. Podprobleme bomo torej reševali od manjših proti večjim, kar v tem primeru pomeni od manjših proti večjim n-jem. Takemu

reševanju rečemo **od spodaj navzgor** (*bottom-up*). Časovna in prostorska zahtevnost sta enaki kot v prejšnjem primeru, le da sta še bolj očitni.

[7]:
```
int fib3[N+1];
fib3[0]=0;
fib3[1]=1;
for (int n=2;n<=N;n++) fib3[n]=fib3[n-1]+fib3[n-2];
cout << fib3[100] << endl;  // overflow
cout << fib3[10] << endl;
```

```
-980107325
55
```

Zaradi sistematičnosti pa smo lahko malo bolj prostorsko učinkoviti. Vedno namreč potrebujemo rezultate samo zadnjih dveh izračunanih problemov. Tako lahko prostorsko zahtevnost zmanjšamo na $O(1)$.

[8]:
```
int fib4(int n) {
    int f2=0, f1=1;
    for (int i=2;i<=n;i++) {
        int fi=f1+f2;
        f2=f1;
        f1=fi;
    }
    return f1;
}
```

[9]:
```
cout << fib4(10) << endl;
```

```
55
```

1.2 Žabji skoki

Vzdolž potoka gleda iz vode n skal na koordinatah $x_1 < x_2 < ... < x_n$. Žabec sedi na prvi skali in bi rad z zaporedjem skokov po skalah prispel do zadnje skale. V enem skoku lahko skoči najmanj a in največ b enot daleč v smeri proti cilju. Kakšno je najmanjše število skokov, ki jih potrebuje za to?

Če je $a = 0$, smo že v poglavju o požrešnih algoritmih na podobnem problemu ugotovili, da lahko z vsakim skokom skoči do najbolj oddaljene skale, ki jo še doseže, in bo s tem minimiziral število svojih skokov. Vpeljava spodnje meje dolžine skoka pa problem zakomplicira.

Če razmišljamo rekurzivno, se bo žabec v prvem skoku premaknil na neko skalo x_i, ki je oddaljena med a in b od skale x_1. Če take skale sploh ni, pot do cilja ne obstaja. Za to je porabil en skok, nato pa se mora v čim manjšem številu skokov premakniti s skale x_i do cilja. Definirajmo podproblem $f(i)$ kot najmanjše število skokov, ki ga žabec potrebuje, da pride na cilj z i-te skale:

- $f(n) = 0$
- $f(i) = \min_{j>i:\ a \le x_j - x_i \le b} (1 + f(j))$

Očitno bo prišlo do ponavljanja podproblemov. Do neke skale lahko žabec pride na več načinov,

ampak za optimalno pot od tam do cilja je povsem nepomembno, kako je do tja prišel. Pomembno je samo, na kateri skali se nahaja. Zato si lahko rešitev shranimo in jo kasneje po potrebi uporabimo, ne da bi jo računali ponovno. Lahko pa bi probleme reševali tudi sistematično po principu od spodaj navzgor, kar v tem primeru pomeni od skal bližje cilju proti tistim bližje začetku.

Rešiti moramo $O(n)$ podproblemov, za rešitev vsakega od njih pa moramo preveriti $O(n)$ možnosti za naslednji skok. Časovna zahtevnost je $O(n^2)$, prostorska pa $O(n)$.

[10]:
```
const int inf=1e9;
int a=3, b=4;
int mem_jump[1000];
```

[11]:
```
int jump(int i, vector<int> &x) {
    int n=x.size();
    if (i==n-1) return 0;
    if (mem_jump[i]!=0) return mem_jump[i];
    int best=inf;
    for (int j=i+1;j<n;j++) {
        int d=x[j]-x[i];
        if (a<=d && d<=b) best=min(best, 1+jump(j,x));
    }
    mem_jump[i]=best;
    return best;
}
```

[12]:
```
vector<int> x = {0,3,4,6,10};
cout << jump(0,x) << endl;
```

```
3
```

1.3 Rezanje palice

Pri problemu rezanja palice (*rod cutting*) imamo podano palico dolžine n, ki jo želimo razrezati na manjše kose in te kose prodati posamično za čim večjo skupno ceno. Dolžina palice in dolžine kosov morajo biti celoštevilske. Podano imamo tabelo cen c, v kateri nam i-to število c_i pove, za kakšno ceno bomo lahko prodali palico dolžine i. Daljši kot je kos, za večjo ceno ga bomo lahko prodali: veljalo bo $c_i \leq c_{i+1}$. Kakšen je največji možen izkupiček od prodaje razrezane palice?

Oglejmo si primer s spodnjo tabelo cen:

i

1

2

3

4

5

6

7

8

c_i

2

5

6

9

15

16

17

20

Naj bo dolžina palice $n = 8$:

- Če razrežemo palico na kose dolžine 1, bomo zanjo dobili $n \cdot c_1 = 16$.
- Če pustimo palico celo, dobimo zanjo $c_8 = 20$.
- Če jo razrežemo na dva kose dolžin 2 in 6, pa bomo dobili $c_2 + c_6 = 21$.
- Če jo razrežemo na dva kose dolžin 1, 2 in 5, bomo dobili $c_1 + c_2 + c_5 = 22$.

Rekurzivni razmislek o zaslužku $f(n)$ pri optimalnem rezanju palice dolžine n nam pove, da bomo morali izbrati dolžino prvega reza. Če je palica dolžine n, si moramo izbrati enega od rezov dolžine $x \leq n$ (s čimer zaslužimo c_x) ter optimalno zrezati preostanek palice dolžine $n - x$. Ker ne vemo, katera dolžina reza bo najboljša, rekurzivno preverimo vse. Uporabimo tokrat pristop od spodaj navzgor in izračunajmo zaslužke za vedno daljše palice: $f(n) = \max_{x \leq n} f(n - x) + c_x$.

[13]:
```
vector<int> c = {0,2,5,6,9,15,16,17,20};
int N=8;
int f[1000];
f[0]=0;
for (int n=1;n<=N;n++) {
    f[n]=0;
    for (int x=1;x<=n;x++) {
        f[n]=max(f[n], f[n-x]+c[x]);
    }
}
cout << f[N] << endl;
```

22

Časovna zahtevnost algoritma je $O(n^2)$, prostorska pa $O(n)$.

Razmislimo še o **rekonstrukciji** rešitve. Katere reze je treba narediti, da dosežemo optimalno ceno? Za vsak podproblem poiščemo potezo, ki je vodila do optimalnega rezultata. Druga možnost pa je, da si že ob reševanju podproblema shranimo optimalno potezo: npr. v dodatni tabeli $g(n)$ bi lahko hranili x, pri katerem funkcija $f(n)$ doseže svoj maksimum.

[14]:
```
int n=N;
while (n>0) {
    for (int x=1;x<=n;x++) {
        if (f[n]==f[n-x]+c[x]) {
            cout << x << ": " << c[x] << endl;
            n-=x;
            break;
        }
    }
}
```

```
1: 2
2: 5
5: 15
```

1.4 Pot v mreži

V labirintu višine h in širine w oz. tabeli znakov '.', ki predstavljajo prosto polje in '#', ki predstavlajo blokirano polje, nas zanima, na koliko načinov lahko pridemo iz levega-zgornjega kota v desni-spodnji kot, pri čemer se lahko premikamo samo desno in navzdol. V spodnjem primeru obstajajo tri take poti.

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

Rekurzivno bi problem reševali tako, da bi se s trenutne celice poskusili premakniti desno in navzdol (če sta oba premika sploh možna) in sešteli možne poti do cilja iz nove lokacije (sosednje celice). Dosedanji problemi so imel eno-dimenzionalen opis podproblema, kjer smo podproblem opisali z eno spremenljivko. Tokrat pa podproblem opišemo z dvema dimenzijama - vrstico in stolpcem celice. Če je polje zasedeno ali se nahaja izven mreže, je število poti do cilja enako 0, sicer pa velja $f(i,j) = f(i+1,j) + f(i,j+1)$. Robni pogoj v desnem-spodnjem kotu je $f(h-1,w-1) = 0$.

Podprobleme lahko rešujemo sistematično po vrsticah od spodaj navzgor in znotraj vrstice od desne proti levi. Tako imamo potrebne rešitve podproblemov vsakič že na voljo. Časovna in prostorska zahtevnost sta $O(hw)$.

[2]:
```
vector<string> lab = {".#....",
                      "....#.",
                      ".#..#.",
                      "......"};
int h=lab.size(), w=lab[0].size();
int f[10][10];
memset(f,0,sizeof(f));
for (int i=h-1;i>=0;i--) {
    for (int j=w-1;j>=0;j--) {
        if (i==h-1 && j==w-1) f[i][j]=1;
        else if (lab[i][j]=='#') f[i][j]=0;
```

```
        else f[i][j]=f[i+1][j]+f[i][j+1];
    }
}
cout << f[0][0] << endl;
```

```
4
```

Prostorsko zahtevnost bi lahko izboljšali na $O(w)$, ker pri računanju vrednosti $f(i,*)$ potrebujemo samo že izračunane rezultate desno v isti vrstici $f(i,*)$ in eno vrstico nižje $f(i+1,*)$.

1.5 Najdaljše skupno podzaporedje

Pri problemu najdaljšega skupnega podzaporedja (*longest common subsequence, LCS*) nizov S in T (dolžine n in m), iščemo najdaljši niz $\mathrm{LCS}(S,T)$, ki se pojavi kot podzaporedje (ne nujno podniz) v S in v T. Oglejmo si primer $S = A\bar{B}C\bar{B}\bar{B}D\bar{A}B$ in $T = \bar{B}DC\bar{B}\bar{B}B\bar{A}$, kjer je eno izmed najdaljših skupnih podzaporedij $\mathrm{LCS}(S,T) = BCBA$ dolžine 4.

Drugačen pogled na isti problem je poravnava obeh nizov, da se pri tem čim več znakov ujema.

```
AB CB DAB
 BDCBB A
```

Rekurzivni razmislek je sledeč:

- Če se oba niza začneta z enakim znakom, je ta znak lahko začetek LCS-ja, preostanek pa je LCS za en znak krajših nizov.
- Če se niza razlikujeta v prvem znaku, potem vsaj en od teh dveh znakov ne bo del LCS-ja. Preizkusimo obe možnosti in rešimo problem z nizoma, kjer je en malo krajši.

Naj bo $\mathrm{LCS}(i,j)$ najdaljši skupni podniz nizov $S_iS_{i+1}\ldots S_{n-1}$ in $T_jT_{j+1}\ldots T_{m-1}$:

$$\mathrm{LCS}(i,j) = \max\begin{cases} 1+\mathrm{LCS}(i+1,j+1) & \text{če } S_i = T_j \\ \mathrm{LCS}(i+1,j) \\ \mathrm{LCS}(i,j+1) \end{cases}$$

Robni primeri pa so $\mathrm{LCS}(n,*) = 0$ in $\mathrm{LCS}(*,m) = 0$.

Problem lahko rešujemo sistematično od večjih proti manjšim i-jem in enako za j. Rešujemo torej probleme z vedno daljšimi priponami nizov S in T. S tem pravzaprav izpolnjujemo 2D tabelo od desnega spodnjega kota proti levemu zgornjemu, tako da izberemo večjo od spodnje in desne celice. Če sta začetna znaka enaka, pa upoštevamo še diagonalen rezultat povečan za 1. Dokažemo lahko tudi, da bo ta diagonalna poteza vedno optimalna, če je na voljo.

[18]:
```
string LCS(string s, string t) {
    int n=s.size(), m=t.size();
    int lcs[n+1][m+1];  // dodatna vrstica in stolpec nicel
    memset(lcs,0,sizeof(lcs));
    for (int i=n-1;i>=0;i--) {
        for (int j=m-1;j>=0;j--) {
            lcs[i][j]=max(lcs[i+1][j], lcs[i][j+1]);
            if (s[i]==t[j]) lcs[i][j]=max(lcs[i][j], 1+lcs[i+1][j+1]);
        }
```

```
    }
    // izpis izracunane tabele
    for (int i=0;i<n;i++) {
        for (int j=0;j<m;j++) {
            cout << lcs[i][j] << '\t';
        }
        cout << endl;
    }
    // rekonstrukcija
    string l="";
    int i=0, j=0;
    while (i<n && j<m) {
        if (lcs[i][j]==lcs[i+1][j]) i++;
        else if (lcs[i][j]==lcs[i][j+1]) j++;
        else { l+=s[i]; i++; j++; }
    }
    return l;
}
```

[19]:
```
string l = LCS("ABCBDAB", "BDCBBA");
cout << "LCS = " << l << endl;
```

```
4       3       3       3       2       1
4       3       3       3       2       1
3       3       3       2       2       1
3       2       2       2       2       1
2       2       1       1       1       1
1       1       1       1       1       1
1       1       1       1       1       0
LCS = BCBA
```

Časovna in prostorska zahtevnost sta $O(nm)$. Problem lahko rešujemo tudi v obratni smeri od konca proti začetkom nizov, kjer se vprašamo, kaj se bo zgodilo z zadnjima znakoma obeh nizov (namesto prvima), kar boste pogosto videli v drugih virih.

Kako pa bi problem rešili za tri nize? $\mathrm{LCS}(S,T,U)$ namreč ni enak $\mathrm{LCS}(\mathrm{LCS}(S,T),U)$! Stanje bi opisali s trojico indeksov $\mathrm{LCS}(i,j,k)$ in obravnavali primere podobno kot za dva niza. Če velja $S_i = T_j = U_k$, je ta znak lahko del LCS-ja, sicer pa vsaj en izmed njih ne bo in lahko enega od nizov skrajšamo.

Soroden problem je iskanje najdaljšega skupnega podniza (ne podzaporedja; *longest common substring*), kjer mora biti pojavitev podniza strnjena v obeh nizih. Ta problem ima drugačne in bolj učinkovite rešitve.

1.6 Nahrbtnik

Problem nahrbtnika (*knapsack, backpack*) je še en klasičen primer uporabe dinamičnega programiranja. Podan imamo nabor n predmetov, za katere poznamo njihove teže t_i in vrednosti v_i (oboje so cela števila). Izbrali bi radi neko podmnožico S teh predmetov, ki bo imela čim večjo vrednost ($\sum_{j\in S} v_j$) in jih bomo lahko spravili v nahrbtnik z nosilnostjo T ($\sum_{j\in S} t_j \leq T$). Problemu se

natančneje reče 0-1 nahrbtnik, ker vsak predmet vzamemo v celoti ali pa ga pustimo, ne moremo pa vzeti samo dela predmeta.

V rekurzivni rešitvi bi se lahko za vsak predmet odličili, ali ga bomo vzeli ali ne. Če ga vzamemo, imamo za preostale predmeta na voljo nekoliko manjšo nosilnost. Podproblem torej opišemo z dvema atributoma.

- Nabor predmetov, za katere se moramo še odločiti, kaj bomo z njimi. Če smo sistematični, se lahko o vključenosti predmetov odločamo po vrsti od prvega do zadnjega.
- Nosilnost nahrbtnika, ki je na voljo za preostale predmete.

Naj bo $f(i, x)$ največja vrednost, ki jo lahko dobimo v nahrbtniku z nosilnostjo x, če lahko vanj dodajamo predmete $i, i+1, \ldots, n$. Obravnavamo dva primera, glede na (ne)uporabo i-tega predmeta. Robni primer je $f(n, *) = 0$ (če nam zmanjka predmetov, lahko dobimo samo vrednost 0).

$$f(i,x) = \max \begin{cases} f(i+1,x) & \text{ne uporabimo i-tega predmeta} \\ f(i+1,x-t_i)+v_i & \text{če je } t_i \leq x\text{, lahko uporabimo i-ti predmet} \end{cases}$$

Časovna zahtevnost je $O(nT)$. Če nimamo meje za T, vemo, da teža predmetov ne bo presegla $\sum t_i$. Ta rešitev z dinamičnim programiranjem izkorišča majhne celoštevilske teže predmetov in nosilnost nahrbtnika. Če bi bile teže in vrednosti neka realna števila, postane problem izrazito težji (NP-težek). V tem primeru imajo različne kombinacije predmetov različne teže in vrednosti, zato se nam podproblemi ne bi ponavljali. V primeru celih števil pa so bile te vrednosti samo z omejenega intervala celih števil. Čeprav obstaja $O(2^n)$ podmnožic, je na razpolago samo $O(T)$ različnih nosilnosti nahrbtnika.

[8]:
```
const int n = 4;
const int nosilnost = 40;
vector<int> teza     = {30,10,40,20};
vector<int> vrednost = {10,20,30,40};

int f[n+1][nosilnost+1];
memset(f,0,sizeof(f));
for (int i=n-1;i>=0;i--) {
    for (int x=0;x<=nosilnost;x++) {
        f[i][x] = f[i+1][x];  // ne uporabimo i-tega predmeta
        if (teza[i]<=x) {  // poskusimo uporabiti i-ti predmet
            f[i][x] = max(f[i][x], vrednost[i]+f[i+1][x-teza[i]]);
        }
    }
}
cout << f[0][nosilnost] << endl;
```

```
60
```

Grafi

December 18, 2024

1 Grafi

Graf G je abstraktni podatkovni tip, ki ga sestavljata množica **vozlišč** (*nodes, vertices, points*) V in množica **povezav** (*edges, links*) E, ki predstavljajo relacije med pari vozlišč. Vozliščema, ki sestavljata povezavo, rečemo **krajišči** (*endpoints*). Vozlišča in povezave lahko hranijo tudi kakšne dodatne lastnosti.

Običajne operacije, ki jih želimo izvajati na grafu so:

- dodajanje/odstranjevanje vozlišča/povezave
- nastavljanje/ugotavljanje lastnosti vozlišča/povezave
- ugotavljanje sosednosti dveh vozlišč
- iskanje vseh sosednjih vozlišč
- …

Kadar z grafom modeliramo nek resničen pojav ali proces, namesto grafa pogosto uporabimo izraz *omrežje (network)*. Grafe lahko uporabimo za modeliranje številnih procesov, kot so razna družbena ali komunikacijska omrežja, omrežja soavtorstev ali celo biološka omrežja, ki modelirajo razne kemijske procese. Mi pa se bomo ukvarjali samo s strukturami brez njihovega ozadja, torej z grafi.

1.1 Terminologija

Glavni lastnosti grafa sta število vozlišč $n = |V|$ in število povezav $e = |E|$ (za število povezav bomo včasih uporabljali tudi m).

Poznamo več vrst grafov glede na njihove lastnosti:

- **Neusmerjeni** (*undirected*) grafi vsebujejo same neusmerjene povezave, ki predstavljajo simetrične relacije, kjer vrstni red krajišč ni pomemben, npr. med dvema bratoma. **Usmerjeni** (*directed*) grafi (*digraphs*) pa so sestavljeni iz usmerjenih povezav, ki predstavljajo asimetrično relacijo, npr. od otroka k staršu. Te običajno ponazorimo z puščicami.
- Glede na lastnost povezav ločimo med **neuteženimi** (*unweighted*) in **uteženimi** (*weighted*) grafi. V neuteženih grafih so vse povezave enakovredne, v uteženih pa vsaki povezavi priredimo neko numerično vrednost, ki ji rečemo utež, in lahko predstavlja npr. dolžino, ceno, …
- **Enostavni** (*simple*) grafi ne vsebujejo **zank** (*loop*), ki povezujejo vozlišče s samim seboj, in **vzporednih povezav** (*multiple/parallel edges*) med istimi pari vozlišč.
- Glede na prisotnost ciklov v grafih poznamo **aciklične** (*acyclic*) in **ciklične** (*cyclic*) grafe.
- Grafe precej grobo ločujemo tudi po razmerju med številom povezav in številom vozlišč. V **gostih** (*dense*) grafih je število vozlišč velikostnega reda, ki je blizu maksimalnemu številu

možnih povezav, $e = O(n^2)$. V **redkih** (*sparse*) grafih pa je število povezav linearno odvisno od števila vozlišč $e = O(n)$.

Oglejmo si še nekaj drugih terminov povezanih z grafi:

- Tako kot pri drevesih, tudi v grafih poznamo **stopnjo** (*degree*) vozlišča, ki je enaka številu povezav, ki vključujejo to vozlišče. Če govorimo o stopnji grafa (kar bomo označevali z d), pa mislimo največjo stopnjo njegovega vozlišča. V usmerjenih grafih ločujemo **vhodno** in **izhodno** stopnjo (*indegree/outdegree*), ki sta število povezav, ki kažejo v vozlišče oz. izven njega.
- Dve vozlišči sta **sosednji** (*adjacent*) oz. **soseda**, če ju povezuje katera izmed povezav v grafu. Množici sosednjih vozlišč izbranega vozlišča rečemo tudi soseščina (*neighbourhood*).

Poleg že omenjenih splošnih vrst grafov, poznamo tudi več razredov grafov, ki imajo podobne strukturne lastnosti. Poznamo:

- **drevesa** (*trees*), ki so v kontekstu novih terminov pravzaprav aciklični povezani neusmerjeni graf
- **polne grafe** (*complete graph*), ki vsebujejo vse možne povezave
- **regularne grafe** (*regular graph*), v katerih imajo vsa vozlišča enako stopnjo
- **dvodelne grafe** (*bipartite graph*), ki so sestavljeni iz dveh skupin vozlišč, povezave pa potekajo samo med obema skupinama
- …

Na grafih nas pogosto zanimajo premiki med sosednjimi vozlišči:

- **Sprehod** (*walk*) je poljubno zaporedje vozlišč, med katerimi se premikamo po povezavah v grafu. Če obstaja sprehod med dvema vozliščema, bomo rekli, da sta **povezani**. Spomnimo se, da če sta povezani neposredno z eno samo povezavo, jima rečemo tudi sosednji.
- **Obhod** (*closed walk*) je sprehod, ki se začne in konča v istem vozlišču.
- **Steza** (*trail*) je sprehod brez ponovljenih povezav.
- **Pot** (*path*) je sprehod brez ponovljenih vozlišč. Uporablja se nekoliko nekonsistentno, npr. za sprehod. V nekaterih primerih pa je to celo nepomembno - najkrajša pot v pozitivno uteženem grafu bo zagotovo pot in ne sprehod, kjer bi se kaj ponavljalo.
- **Cikel** (*cycle*) je obhod brez ponovljenih vmesnih vozlišč (z izjemo začetnega in končnega, ki sta enaka).
- V angleščini se pojavlja tudi termin *tour*, ki pa nima poenotene definicije (npr. knight's tour, Euler tour). Običajno pomeni, da zaporedje premikov obišče celoten graf (npr. vsa vozlišča, vse povezave) ob možnih dodatnih omejitvah (npr. vsako povezavo samo enkrat, vrne se na izhodišče).

1.2 Predstavitve

Strukturo grafa, ki jo definirajo vozlišča in povezave, moramo nekako predstaviti oz. shraniti, da bomo lahko na njej izvajali kakšne izračune. Glede na funkcionalnost, ki jo potrebujemo, poznamo tri pogoste načine predstavitve grafov. Če je treba, pa si lahko pomagamo kar z več različnimi predstavitvami sočasno.

[1]:
```
#include <iostream>
#include <fstream>
#include <vector>
```

```
#include <queue>
#include <algorithm>
using namespace std;

typedef pair<int,int> PII;
typedef vector<int> VI;
typedef vector<pair<int,int>> VII;
typedef vector<vector<int>> VVI;
```

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

- **Seznam povezav** (*edge list*) je najbolj enostavna predstavitev. Vse povezave v grafu preprosto shranimo v seznam. Ta predstavitev bo primerna, če želimo obravnavati vse povezave ne glede na vrstni red.

[3]:
```
VII read_graph(string fname, int &n, int &m) {
    ifstream fin(fname);
    fin >> n >> m;
    vector<PII> povezave;
    for (int i=0;i<m;i++) {
        int a,b;
        fin >> a >> b;
        povezave.push_back({a,b});
    }
    fin.close();
    return povezave;
}
```

[4]:
```
int n,m;
vector<PII> povezave = read_graph("graph.txt",n,m);
for (auto [a,b] : povezave) cout << '(' << a << ',' << b << ')' << ' ';
cout << endl;
```

```
(0,1) (0,4) (1,3) (1,4) (1,5) (1,7) (2,3) (2,5) (4,5) (6,7)
```

- **Seznam sosedov** (*adjacency list*) hrani za vsako vozlišče seznam njegovih sosedov. Kadar se premikamo po grafih od enega vozlišča k drugemu, nam to pride zelo prav.

[5]:
```
VVI adjacency_list(VII &edge_list, int n, bool dir=false) {
    vector<VI> adj(n);
    for (auto [a,b] : edge_list) {
        adj[a].push_back(b);
        if (!dir) adj[b].push_back(a);
    }
    return adj;
```

```
}
```

[6]:
```
vector<VI> sosedi = adjacency_list(povezave, n);
for (int i=0;i<n;i++) {
    cout << i << ": ";
    print(sosedi[i]);
}
```

```
0: 1 4
1: 0 3 4 5 7
2: 3 5
3: 1 2
4: 0 1 5
5: 1 2 4
6: 7
7: 1 6
```

- **Matrika sosednosti** (*adjacency matrix*) je namenjena učinkovitemu preverjanju sosednosti dveh vozlišč. Sestavimo namreč matriko M, kjer na mestu $M_{x,y}$ hranimo informacijo o prisotnosti ali teži povezave med vozliščema x in y.

[7]:
```
VVI adjacency_matrix(VII &edge_list, int n) {
    vector<VI> mat(n, vector<int>(n));
    for (auto [a,b] : edge_list) {
        mat[a][b] = 1;
        mat[b][a] = 1;
    }
    return mat;
}
```

[8]:
```
vector<VI> sosednost = adjacency_matrix(povezave, n);
for (int i=0;i<n;i++) {
    print(sosednost[i]);
}
```

```
0 1 0 0 1 0 0 0
1 0 0 1 1 1 0 1
0 0 0 1 0 1 0 0
0 1 1 0 0 0 0 0
1 1 0 0 0 1 0 0
0 1 1 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 1 0
```

Predstavitev s seznami povezav ali sosedov bi lahko nadgradili z uporabo množic. Namesto v seznamu hranimo povezave ali sosede v množicah, ki so implementirane z razpršeno tabelo ali kakšno uravnotežene drevesno strukturo.

Omenjene predstavitve imajo svoje prednosti in slabosti. Primerjajmo jih med seboj glede na prostorsko zahtevnost in časovne zahtevnosti nekaterih operacij na enostavnih grafih.

seznam povezav

seznam sosedov

matrika sosednosti

Prostorska zahtevnost

$O(e)$

$O(n + e)$

$O(n^2)$

Dodajanje povezave

$O(1)$

$O(1)$

$O(1)$

Brisanje povezave

$O(e)$

$O(n)$

$O(1)$

Dodajanje vozlišča

$O(1)$

$O(1)$

$O(n^2)$

Brisanje vozlišča

$O(e)$

$O(e)$

$O(n^2)$

Sosednost vozlišč

$O(e)$

$O(n)$

$O(1)$

1.3 Preiskovanje grafov

Preiskovanje grafa (*graph traversal/search*) je sistematičen postopek, ki obišče vsa vozlišča grafa v nekem vrstnem redu. Poznamo dve pogosti vrsti preiskovanj.

1.3.1 Preiskovanje v širino (*breadth-first search, BFS*)

Preiskovanje v širino preiskuje vozlišča podobno kot nivojski obhod v drevesih, le da se izogiba povezavam, ki vodijo do že obiskanih vozlišč. Najprej obišče začetno vozlišče, nato njegove sosede, njihove sosede, itd.

[9]:
```
void BFS(int x, vector<VI> &adj, vector<int> &vis, vector<int> &seq) {
    queue<int> q;
    q.push(x); vis[x]=1;
    while (!q.empty()) {
        x=q.front(); q.pop();
        seq.push_back(x);
        for (int y : adj[x]) if (vis[y]==0) {
            q.push(y); vis[y]=1;
        }
    }
}
```

[10]:
```
vector<int> visB(n), seqB;
BFS(0,sosedi,visB,seqB);
print(seqB);
```

```
0 1 4 3 5 7 2 6
```

Iskanje v širino ima to lepo lastnost, da obiskuje vozlišča po nivojih od bližjih proti bolj oddaljenim. Z minimalno prilagoditvijo ga lahko uporabimo za računanje *najkrajših poti* iz začetnega vozlišča do vseh ostalih vozlišč v neuteženem grafu, kjer je dolžina poti definirana s številom povezav na njej!

1.3.2 Preiskovanje v globino (*depth-first search, DFS*)

Preiskovanje v globino je podobno prememu obhodu v drevesu, ki se izogiba povezam do že obiskanih vozlišč. Najprej obišče začetno vozlišče. Nato izvede preiskovanje v globino na prvem otroku. Ko se to zaključi in če drugi otrok še ni bil obiskan, izvede preiskovanje v globino še iz drugega otroka itd.

[11]:
```
void DFS(int x, vector<VI> &adj, vector<int> &vis, vector<int> &seq) {
    seq.push_back(x);
    vis[x]=1;
    for (int y : adj[x]) if (vis[y]==0) {
        DFS(y, adj, vis, seq);
    }
}
```

[12]:
```
vector<int> visD(n), seqD;
DFS(0,sosedi,visD,seqD);
print(seqD);
```

```
0 1 3 2 5 4 7 6
```

Oba opisana postopka obiščeta samo del grafa, ki je dosegljiv iz začetnega vozlišča. Tej množici vozlišč v neusmerjenem grafu, ki so vsa povezana med seboj, rečemo **povezana komponenta** grafa (*connected component*). Za iskanje povezanih komponent lahko uporabimo kateregakoli od omenjenih postopkov za preiskovanje.

Prostorska zahtevnost obeh preiskovanj je $O(n)$. Časovno zahtevnost bi lahko ocenili z $O(n^2)$, vendar smo lahko bolj natančni z $O(e)$, ker bomo vsako povezavo obravnavali največ dvakrat (enkrat iz vsakega krajišča).

Drevo preiskovanja v globino Tudi iskanje v globino ima svoje lepe lastnosti. Prva je jedrnatost. Druga pa je v strukturi povezav, ki jih postopek obišče med preiskovanjem. Prehojene povezave bodo imele obliko drevesa (to sicer velja tudi za iskanje v širino). Poleg tega pa bodo vse ostale povezave v grafu vedno povezovale vozlišča z nekim svojim prednikom (*back-edge*) ali potomcem (*forward-edge*) v drevesu. Nemogoče je, da bi obstajala povezava med dvema poddrevesoma (*cross-edge*). Razmislite, zakaj je temu tako. To lastnost izkoriščajo pomembni algoritmi za iskanje *mostov*, *prereznih vozlišč* in *močno povezanih komponent*. Razmislite tudi, kakšne povezave lahko nastopajo v drevesu preiskovanja v globino na usmerjenem grafu.

1.4 Detekcija ciklov

Podan imamo graf, za katerega ne vemo, ali vsebuje kakšen cikel ali ne. Ugotovili bi radi prisotnost cikla in tudi našli konkreten primer cikla v grafu. Problem se nekoliko razlikuje med neusmerjenimi in usmerjenimi grafi. Če bi vsako neusmerjeno povezavo modelirali z dvema nasproti usmerjenima, bi vsaka povezava predstavljala cikel, česar nočemo.

Oglejmo si najprej primer neusmerjenega grafa. Pri razmisleku nam bo prav prišlo drevo preiskovanja v globino. Cikel bo v tem drevesu izgledal tako, da bo obstajala povezava med dvema vozliščema, ki imata relacijo prednik-potomec. To povezavo bomo pri preiskovanu v globino našli takrat, ko bomo obravnavali neko vozlišče x in našli povezavo do nekega že obiskanega prednika y. Vozlišča na poti od x proti y bodo formirala cikel, ker med njima obstaja pot po drevesu poleg tega pa še novo odkrita direktna povezava. Prav nam bo prišlo, če bi drevo preiskovanja v globino hranili v obliki tabele staršev za vsako vozlišče. Če je ta vrednost nenastavljena (npr. -1), je vozlišče še neobiskano, koren pa naj ima za starša kar samega sebe. Tako lahko za izgradnjo cikla preprosto sledimo tem starševskim povezavam od x do y.

[13]:
```
int cycle(int x, vector<VI> &adj, vector<int> &par, vector<int> &cyc) {
    if (par[x]==-1) par[x]=x;
    for (int y : adj[x]) if (y!=par[x]) {
        if (par[y]!=-1) {  // cikel
            for (int z=x; z!=y; z=par[z]) cyc.push_back(z);
            cyc.push_back(y);
            return 1;
        }
        par[y]=x;
        if (cycle(y,adj,par,cyc)) return 1;
    }
    return 0;
}
```

[14]:
```
vector<int> vis(n), par(n,-1), cyc;
cout << cycle(0,sosedi,par,cyc) << endl;
print(cyc);
```

```
1
5 2 3 1
```

V usmerjenem grafu je situacija nekoliko drugačna. Povezave na ciklu morajo kazati v isto smer. Če ponovno razmislimo o situaciji na drevesu preiskovanja v globino, bo cikel tudi tu nastal s povezavo od nekega vozlišča x do njegovega prednika y. Povezave iz vozlišča x do nekega drugega dela drevesa, ki je že bil obiskan, ne vzpostavijo cikla zaradi usmerjenosti. Poleg obiskanosti vozlišč bomo hranili še informacijo o vozliščih na poti od korena do trenutnega vozlišča. S tem lahko učinkovito ugotovimo, ali je vozlišče prednik x-a. Pri sestavljanju cikla bomo zaradi premikanja proti prednikom cikel sestavili v obratnem vrstnem redu.

[15]:
```
int cycleDir(int x, vector<VI> &adj, vector<int> &par, vector<int> &path,␣
 ↪vector<int> &cyc) {
    if (par[x]==-1) par[x]=x;
    path[x]=1;
    for (int y : adj[x]) if (y!=par[x]) {
        if (path[y]) {  // prednik (cikel)
            for (int z=x; z!=y; z=par[z]) cyc.push_back(z);
            cyc.push_back(y);
            reverse(cyc.begin(), cyc.end());
            return 1;
        }
        if (par[y]==-1) {  // neobiskano
            par[y]=x;
            if (cycleDir(y,adj,par,path,cyc)) return 1;
        }
    }
    path[x]=0;
    return 0;
}
```

Za testiranje si bomo izposodili spodnji usmerjeni graf z dodatno povezavo $5 \to 4$, da ustvarimo cikel. Paziti moramo tudi na to, od kod začnemo iskanje. Če cikel ni dosegljiv iz začetnega vozlišča, ga ne bomo našli. V tem primeru bi morali začeti iskanje na novo iz nekega neobiskanega vozlišča, dokler niso obiskana vsa in šele takrat lahko zagotovimo, da cikla ni.

[16]:
```
povezave = read_graph("directed.txt",n,m);
povezave.push_back({5,4});
vector<VI> sosediDir = adjacency_list(povezave, n, true);
```

[17]:
```
vector<int> visDir(n), parDir(n,-1), path(n), cycDir;
cout << cycleDir(2,sosediDir,parDir,path,cycDir) << endl;
print(cycDir);
```

```
1
```

```
1 5 4 0
```

1.5 Topološko urejanje

Naj usmerjeni graf predstavlja medsebojne odvisnosti izvedbe opravil. Vozlišča ustrezajo opravilom, povezava $x \to y$ pa pomeni, da je treba opravilo x izvesti pred opravilom y. V kakšnem vrstnem redu naj izvajamo opravila, da bomo lahko izvedli vsa oz. je to sploh mogoče?

Topološki vrstni red vozlišč v usmerjenem grafu je tak vrstni red, da vse povezave v grafu kažejo od zgodnejšega proti kasnejšemu vozlišču v topološkem vrstnem redu. Topološki vrstni red ni enoličen. Za zgornji primer bi bil možen topološki vrstni red npr. `[4,0,2,3,1,6,5]`. Ker v grafu nastopa povezava $0 \to 5$, se v topološkem vrstnem redu 0 pojavi pred 5. Preverimo lahko, da to velja za vse povezave.

[18]:
```
povezave = read_graph("directed.txt",n,m);
sosedi = adjacency_list(povezave, n, true);
for (int i=0;i<n;i++) {
    cout << i << ": ";
    print(sosedi[i]);
}
```

```
0: 1 3 5
1: 5
2: 3
3: 1 6
4: 0 3
5:
6:
```

Razmislimo o algoritmu za izgradnjo topološkega vrstnega reda. Vozlišča brez predhodnikov lahko postavimo na začetek topološkega vrstnega reda. Če je takih vozlišč več, njihov medsebojni vrstni red ni pomemben. Za povezave, ki izhajajo iz njih, je torej poskrbljeno. Zato lahko ta vozlišča in njihove povezave odstranimo iz grafa ter ponovimo postopek z morebitnimi novimi vozlišči brez predhodnikov. Postopek se ne zaključi, če topološki vrstni red ne obstaja zaradi prisotnosti cikla v grafu. **Usmerjeni aciklični grafi** (*directed acyclic graph - DAG*) so svoj razred grafov, ki jih je mogoče topološko urediti.

Kako naj opisani postopek učinkovito implementiramo? Odstraniti moramo n vozlišč in na vsakem koraku iščemo med preostalimi vozlišči kakšnega z vhodno stopnjo 0. Direktna implementacija takega postopka bo imela kvadratno časovno zahtevnost. To pa lahko izboljšamo v vodenjem seznama vozlišč z vhodno stopnjo 0. Vsakič, ko odstranimo vozlišče in njegove izhodne povezave, dodamo v seznam morebitna novo nastala začetna vozlišča. Tako dobimo algoritem s časovno zahtevnostjo $O(n+e)$. Običajno je število povezav vsaj tolikšno kot število vozlišč, zato lahko brez večje škode poenostavimo na $O(e)$.

[19]:
```
VI toposort(vector<VI> &sosedi, int n) {
    vector<int> indeg(n);
    for (int x=0;x<n;x++) {
        for (int y : sosedi[x]) indeg[y]++;
    }
```

```
    queue<int> q;
    for (int x=0;x<n;x++) {
        if (indeg[x]==0) q.push(x);
    }
    vector<int> seq;
    while (!q.empty()) {
        int x=q.front(); q.pop();
        seq.push_back(x);
        for (int y : sosedi[x]) {
            indeg[y]--;
            if (indeg[y]==0) q.push(y);
        }
    }
    return seq;
}
```

[20]:
```
vector<int> topo = toposort(sosedi, n);
print(topo);
```

```
2 4 0 3 1 6 5
```

1.6 Kritična pot

Potek izvajanja projekta lahko modeliramo z mejniki in aktivnostmi, ki doprinesejo k izpolnjevanju teh mejnikov. Mejnike predstavimo z vozlišči, aktivnosti pa s povezavami v usmerjenem grafu. Ko so končane vse potrebne aktivnosti, je mejnik dosežen. Poleg tega poznamo čas $w(x, y)$ za izvedbo določene aktivnosti med mejnikoma x in y. Očitno mora biti graf acikličen. Kakšen je najkrajši čas za izvedbo projekta ob "neomejeni" količini resursov, pri čemer lahko vsako aktivnost izvaja ena oseba, vendar imamo na voljo poljubno število oseb? Ta čas predstavlja najdaljša pot v uteženem usmerjenem acikličnem grafu, ki ji rečemo tudi kritična pot.

Kako pa jo izračunamo? Vozlišča naprej topološko uredimo v linearnem času. Nato pa lahko računamo najdaljše poti $d(x)$, ki se začnejo v v posameznem vozlišču x, v obratnem topološkem vrstnem redu. Če vozlišče nima naslednikov, je $d(x) = 0$. Sicer pa velja $d(x) = \max_{y:\, x<y \wedge (x,y)\in E} (w(x, y) + d(y))$.

Opravka imamo z uteženim grafom, ki ga moramo najprej prebrati. V seznamu sosedov bomo poleg sosednjega vozlišča hranili še težo povezave, ki vodi do njega.

[21]:
```
ifstream fin("critical.txt");
fin >> n >> m;
vector<VI> adj(n);
vector<VII> adjw(n);
for (int i=0;i<m;i++) {
    int a,b,c;
    fin >> a >> b >> c;
    adj[a].push_back(b);
    adjw[a].push_back({b,c});
}
```

```
fin.close();
```

Algoritem za izračun topološkega vrstnega reda že imamo, samo obrnemo ga.

[22]:
```
vector<int> ord = toposort(adj, n);
reverse(ord.begin(), ord.end());
```

V tem obratnem topološkem vrstem redu lahko izračunamo dolžino najdaljše poti iz vsakega vozlišča, saj bo vsaka vrednost odvisna samo od naslednikov, za katere imamo rezultat že izračunan. Zapomnimo si tudi vozlišče z največjim rezultatom, ki je začetek najdaljše poti.

[23]:
```
vector<int> d(n);
int start = ord[0];
for (int x : ord) {
    for (auto [y,w] : adjw[x]) {
        d[x] = max(d[x], w+d[y]);
    }
    if (d[x]>d[start]) start=x;
}
cout << "dolzina = " << d[start] << endl;
```

```
dolzina = 10
```

Izračunane vrednosti so dovolj, da lahko pot tudi rekonstruiramo. Iz trenutnega vozlišča nadaljujemo tam, kjer je izračunana najdaljša pot ravno za dolžino povezave krajša. Druga možnost bi bila, da si pri računanju najdaljših poti za vsako vozlišče poleg razdalje shranjujemo tudi naslednje vozlišče, ki je vodilo do te maksimalne vrednosti.

[25]:
```
cout << start;
int x=start;
while (d[x]!=0) {
    for (auto [y,w] : adjw[x]) {
        if (d[x]==w+d[y]) {
            cout << " " << y;
            x = y;
            break;
        }
    }
}
cout << endl;
```

```
4 6 0 5 2
```

1.7 Eulerjev obhod

Dobro znan problem na neusmerjenih grafih je iskanje Eulerjevega obhod (*Eulerian tour/cycle/circuit*). Pri tem iščemo obhod, ki obišče vse povezave v grafu (vsako povezavo natanko enkrat, vozlišča pa morda tudi večkrat). Podoben problem je iskanje Eulerjevega sprehoda (*Eulerian trail/path/walk*). Pravzaprav iščemo stezo (sprehod brez ponovljenih povezav vendar morda s

ponovljenimi vozlišči), ki obišče vse povezave v grafu. Za razliko od obhoda pa se lahko začne in konča na različnih mestih.

S tem problemov ste se najbrž že srečali pri risanju oblik z eno potezo (npr. odprtega pisma/ovojnice). Euler pa pri problemu sedmih mostov v Königsbergu (danes Kaliningrad). Zanimalo ga je, kako bi lahko na sprehodu prehodil vsak most natanko enkrat.

Eulerjev izrek pravi, da v povezanem grafu obstaja Eulerjev obhod natanko takrat, ko so vsa vozlišča sode stopnje. Eulerjev sprehod pa natanko takrat, ko so vsa vozlišča sode stopnje razen morda točno dveh vozlišč, kjer se začne in konča. Dokažimo to trditev za primer obhoda (za sprehod velja podobno).

- Recimo, da obstaja Eulerjev obhod. Potem ta obhod na prehodu skozi vsako vozlišče zmanjša stopnjo tega vozlišča za 2. Če sproti odstranjujemo prehojene povezave, imajo na koncu vsa vozlišča stopnjo 0. Torej morajo biti na začetku vsa sode stopnje.
- Obratna smer je bolj kompleksna in jo lahko dokažemo kar s konstrukcijo Eulerjevega obhoda na povezanem grafu z vozlišči sodih stopenj. Začnemo v poljubnem vozlišču x in sledimo povezavam, dokler se ne vrnemo v začetno vozlišče x. Pri tem se ne moremo zatakniti v nekem drugem vozlišču y, ker bi že porabili vse njegove povezave. V vsakem prehodu skozi vozlišče namreč porabimo dve povezavi - če je na voljo vsaj ena za vstop, bo tudi druga za izstop, ker so vsa vozlišča sode stopnje. Morda pa smo se vrnili v začetno vozlišče, pri tem pa še nismo obiskali vseh povezav. Postopek ponovimo na enem od že obiskanih vozlišč, ki ima še kakšne neobiskane povezave. Od tam na enak način zgradimo obhod in ga združimo s prejšnjim. To ponavljamo dokler niso obiskane vse povezave. To je Hierholzerjev algoritem, ki ga lahko implementiramo v linearnem času.

Najkrajse poti

December 18, 2024

1 Najkrajše poti

Klasičen problem na grafih je iskanje najkrajših poti. Zanima nas na primer najkrajša pot med parom vozlišč A in B (*single-pair shortest path*). Naj bo ta najkrajša pot sestavljena iz vozlišč $A, ...X, B$, kjer je X predzadnje vozlišče na poti. V tem primeru mora biti tudi pot od A do X najkrajša, sicer bi lahko pot od A do B izboljšali. Pri iskanju najkrajše poti od A do B posledično izračunamo tudi najkrajše poti do ostalih vozlišč na tej poti.

Če bomo že morali izračunati najkrajše poti iz A do več drugih vozlišč, pa jih lahko izračunamo iz začetnega vozlišča kar do vseh (*single-source shortest path*). Opazimo tudi, da bodo te najkrajše poti v grafu formirale **drevo najkrajših poti**. Vsako vozlišče bo imelo namreč enega optimalnega predhodnika/starša na najkrajši poti (npr. X bo predhodnik B-ja). Koren drevesa pa bo seveda v vozlišču A.

Za problem iskanja najkrajših poti med vsemi pari točk, lahko N-krat poženemo algoritem za iskanje drevesa najkrajših poti iz posameznega začetnega vozlišča. Obstajajo pa tudi drugi algoritmi, ki si namenjeni prav iskanju poti med vsemi pari točk. Tak primer je *Floyd-Warshall*-ov algoritem, ki ga tu ne bomo obravnavali.

Ukvarjali se bomo predvsem z neusmerjenimi grafi. V usmerjenih grafih je situacija namreč podobna in lahko uporabimo enake razmisleke.

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

typedef pair<int,int> PII;
typedef vector<int> VI;
typedef vector<pair<int,int>> VII;
typedef vector<vector<int>> VVI;
```

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

1.1 Neuteženi grafi

V neuteženih grafih ni potrebe po kompliciranju, saj že poznamo metodo **iskanja v širino (BFS)**, ki obiskuje vozlišča od bližnjih proti bolj oddaljenim glede na število povezav. Potrebuje je malenkostno dopolnitev, da bo poleg obiskovanja vozlišč beležila še dolžine poti in prednike vozlišč v drevesu najkrajših poti.

[3]:
```
ifstream fin("graph.txt");
int n,m;
fin >> n >> m;
vector<vector<int>> sosedi(n);
for (int i=0;i<m;i++) {
    int a,b;
    fin >> a >> b;
    sosedi[a].push_back(b);
    sosedi[b].push_back(a);
}
```

[3]:
```
void BFS_distance(vector<VI> &adj, int start, vector<int> &dist, vector<int>␣
 ↪&prev) {
    int n=adj.size();
    dist=vector<int>(n,-1); prev=vector<int>(n);
    vector<int> vis(n);
    queue<int> q;
    q.push(start); vis[start]=1;
    dist[start]=0; prev[start]=-1;
    while (!q.empty()) {
        int x=q.front(); q.pop();
        for (int y : adj[x]) {
            if (!vis[y]) {
                q.push(y); vis[y]=1;
                dist[y]=dist[x]+1; prev[y]=x;  // distance, previous node
            }
        }
    }
}
```

[5]:
```
vector<int> dist, prev;
BFS_distance(sosedi,0,dist,prev);
print(dist);
print(prev);
```

```
0 1 3 2 1 2 3 2
-1 0 3 1 0 1 7 1
```

1.2 Uteženi grafi

V uteženih grafih pa je situacija malo bolj zapletena. Omejili se bomo na grafe s **pozitivnimi (nenegativnimi) utežmi**, s kakršnimi imamo večinoma opravka v praksi, kasneje pa se bomo

vrnili še k negativnim utežem. Utež (ceno, dolžino) povezave med vozliščema X in Y bomo označili z $w(X,Y)$.

[4]:
```
ifstream fin("weighted.txt");
int n,m;
fin >> n >> m;
vector<VII> adjw(n);
for (int i=0;i<m;i++) {
    int a,b,w;
    fin >> a >> b >> w;
    adjw[a].push_back({b,w});
    adjw[b].push_back({a,w});
}
```

1.2.1 Dijkstrov algoritem

Tako kot smo v neuteženem primeru z iskanjem v širino računali najkrajše poti od bližnjih proti bolj oddaljenim vozliščem, bomo to storili tudi tu. Najbližje vozlišče je kar izhodiščno, $d(A) = 0$. Naslednje najbližje vozlišče pa bo eno od njegovih sosedov. Ker povezave niso negativne, je nemogoče, da bi dosegli manjšo razdaljo po kakšni poti z več povezavami. Tem neizračunanim sosedom do sedaj izračunanih vozlišč bomo rekli okolica. To so vozlišča, ki še niso izračunana in so iz že izračunanih dosegljiva po eni povezavi. Za vsako od njih bomo hranili potencialno najkrajšo pot $p(Y)$: kakšna bi bila razdalja, če bi se do njega premaknili z enega izmed že izračunanih vozlišč. Če iz okolice izberemo vozlišče X s trenutno najmanjšo potencialno dolžino $p(X)$, bo to zagotovo dejanska najmanjša dolžina poti do tega vozlišča ($d(X) = p(X)$). Zaradi odsotnosti negativnih povezav, bi bila katerakoli druga pot od že izračunanih vozlišč do X sestavljena iz več povezav in zato daljša. Množico že izračunanih vozlišč smo torej povečali z novim vozliščem X. Poskrbeti moramo še za posodobitev okolice. Vse sosede Y vozlišča X dodamo v okolico, če so že v njej, pa zgolj posodobimo njihovo potencialno oddaljenost z $p(Y) = \min(p(Y), d(X) + c(X,Y))$. Postopek ponavljamo, dokler nimamo izračunanih najkrajših poti do vseh vozlišč.

V postopku imamo opravka s tremi skupinami vozlišč. V prvi skupini so tista, za katera imamo že izračunane najkrajše poti. V drugi skupini so vozlišča iz okolice, ki imajo samo potencialne dolžine. Tretja skupina pa so še povsem neobiskana vozlišča. Pri implementaciji bomo vse te informacije hranili v tabeli potencialnih razdalj. Razdalja -1 bo označevala še neobiskano vozlišče iz tretje skupine, -2 pa že izračunano iz prve.

[5]:
```
void Dijkstra(vector<VII> &adjw, int start, vector<int> &dist, vector<int>␣
 ↪&prev) {
    int n=adjw.size();
    dist=vector<int>(n,-1); prev=vector<int>(n,-1);
    vector<int> p(n,-1);  // provisional distance (-1=unvisited, -2=done)
    p[start]=0;
    while (1) {
        int x=-1;  // smallest provisional
        for (int i=0;i<n;i++) if (p[i]>=0) {
            if (x==-1 || p[i]<p[x]) x=i;
        }
```

```
        if (x==-1) break;
        dist[x]=p[x]; p[x]=-2;
        for (auto [y,w] : adjw[x]) {  // update neighbors
            int d=dist[x]+w;
            if (p[y]==-1 || (p[y]>=0 && d<p[y])) {
                p[y]=d; prev[y]=x;
            }
        }
    }
}
```

[11]:
```
vector<int> dist, prev;
Dijkstra(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 4 11 17 9 22 7 8 11
-1 0 4 2 7 3 0 6 7
```

Prostorska zahtevnost algoritma je $O(n)$. Časovna zahtevnost pa je odvisna od iskanja najmanje potencialne razdalje ($O(n^2)$) in posodabljanja sosedov ($O(e)$). Ker je $e = O(n^2)$, je časovna zahtevnost take implementacije algoritma $O(n^2)$.

Razmislimo o izboljšavi. Težavno je iskanje vozlišča z najmanjšo potencialno razdaljo. Hkrati pa moramo biti sposobni posodabljati potencialne razdalje sosedov. Vozlišča iz okolice s potencialnimi razdaljami bi lahko hranili v uravnoteženem iskalnem drevesu. Tako lahko v času $O(\log n)$ poiščemo najmanjšega in spremenimo potencialno razdaljo vozlišča. Časovna zahtevnost bi bila $O(n \log n + e \log n) = O(e \log n)$.

Iskanje najmanjšega elementa je namen prioritetne vrste, zato je to v praksi pogostejši način implementacije, ki je tudi preprostejši in zato bolj učinkovit. Če za prioritetno vrsto uporabimo dvojiško kopico, mora ta omogočati tudi spremembo prioritete. Pravzaprav gre samo za zmanjšanje prioritete v minimalni dvojiški kopici, kar lahko dosežemo v logaritemskem času. Tudi ta rešitev ima časovno zahtevnost $O(e \log n)$.

V spodnji implementaciji pa bomo malo "goljufali" in se izognili spreminjanju prioritet. Pri posodabljanju bomo v prioritetno vrsto samo vstavili novo manjšo vrednost, stare pa ne bomo izbrisali. Nova vrednost bo prišla iz vrsto prej, zato lahko stare neveljavne vrednosti, ki pridejo iz vrste nekoč kasneje, enostavno ignoriramo. V tabeli razdalj `dist` bomo hranili razdalje do vseh vozlišč (nekatere so pravilne, druge zgolj potencialne). Vozlišča, katerih razdalje so zgolj potencialne, bomo hranili v prioritetni vrsti. Ko pride vozlišče iz prioritetne vrste, vemo, da je njegova razdalja pravilna in posodobimo sosede. V prioritetni vrsti je lahko $O(e)$ elementov, zato je taka tudi prostorska zahtevnost. Časovna zahtevnost pa je $O(e \log e) = O(e \log n^2) = O(e \cdot 2 \log n) = O(e \log n)$. Goljufija torej ni bila prav huda.

Vso to kompliciranje pa ima smisel samo, če je graf dovolj redek. Če je graf gost in vsebuje skoraj vse možne povezave ($e \approx n^2$), je časovna zahtevnost $O(e \log n)$ pravzaprav $O(n^2 \log n)$, kar je slabše od $O(n^2)$, s čimer smo začeli.

[6]:
```
void Dijkstra_PQ(vector<VII> &adjw, int start, vector<int> &dist, vector<int>␣
↪&prev) {
```

```
    int n=adjw.size();
    dist=vector<int>(n,-1); prev=vector<int>(n,-1);
    priority_queue<PII, vector<PII>, greater<PII>> pq;  // (distance, node)
    dist[start]=0; pq.push({0,start});
    while (!pq.empty()) {
        auto [d,x]=pq.top(); pq.pop();
        if (dist[x]!=d) continue;  // ignore old values
        for (auto [y,w] : adjw[x]) {  // update neighbors
            int d=dist[x]+w;
            if (dist[y]==-1 || d<dist[y]) {
                dist[y]=d; prev[y]=x;
                pq.push({d,y});
            }
        }
    }
}
```

[17]:
```
vector<int> dist, prev;
Dijkstra_PQ(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 4 11 17 9 22 7 8 11
-1 0 4 2 7 3 0 6 7
```

Algoritem lahko v nekaterih primerih še izboljšamo. Pogosto so uteži relativno majhna cela števila. Naj bo c največja utež v grafu. Največja oddaljenost vozlišča v grafu bo tako $(n-1)c$. Namesto v prioritetni vrsti lahko vozlišča s potencialnimi razdaljami hranimo "popredalčkana" v tabeli, ki na mestu i hrani seznam vozlišč na razdalji i. Temu rečemo tudi vrsta z vedri (*bucket queue*). Podobno kot prej ne spreminjamo vrednosti, ampak dodajamo nove in po potrebi ignoriramo stare. Prostorska in časovna zahtevnost take rešitve sta $O(e+nc)$.

[7]:
```
void Dijkstra_BQ(vector<VII> &adjw, int start, vector<int> &dist, vector<int>␣
 ↪&prev) {
    int n=adjw.size();
    dist=vector<int>(n,-1); prev=vector<int>(n,-1);
    int c=0;  // maximum weight
    for (int x=0;x<n;x++) for (auto [y,w] : adjw[x]) c=max(c, w);
    int maxd=(n-1)*c;
    vector<VI> bq(maxd+1);  // bucket queue
    dist[start]=0; bq[0].push_back(start);
    for (int d=0;d<=maxd;d++) {
        for (int x : bq[d]) {
            if (dist[x]!=d) continue;  // ignore old values
            for (auto [y,w] : adjw[x]) {  // update neighbors
                int d=dist[x]+w;
                if (dist[y]==-1 || d<dist[y]) {
                    dist[y]=d; prev[y]=x;
                    bq[d].push_back(y);
```

```
                }
            }
        }
    }
}
```

[8]:
```
vector<int> dist, prev;
Dijkstra_BQ(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 4 11 17 9 22 7 8 11
-1 0 4 2 7 3 0 6 7
```

1.2.2 Negativne uteži

Do sedaj smo se omejili na pozitivne oz. nenegativne uteži. Negativne uteži imajo smisel samo na usmerjenih grafih. Sicer bi se lahko sprehajali tja in nazaj po isti negativni povezavi in imeli vedno krajšo pot.

Kje pa pride do težave na usmerjenih grafih? Naša predpostavka, da ima vozlišče v okolici z najmanjšo potencialno razdaljo prav tako tudi dejansko razdaljo, ni več resnična. To lahko demonstriramo s spodnjim primerom.

[10]:
```
// (0,1,2), (0,2,3), (2,1,-2)
vector<VII> adjw = {{{1,2},{2,3}},{},{{1,-2}}};
vector<int> dist, prev;
Dijkstra(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 2 3
-1 0 0
```

Situacija je lahko še slabša. V usmerjenem grafu se lahko pojavi negativen cikel (cikel z negativno vsoto uteži). V takem primeru koncept najkrajših poti tudi nima smisla, ker lahko krožimo po ciklu in s tem poljubno krajšamo svojo pot.

Obstajajo algoritmi, ki uspešno rešujejo probleme najkrajših poti tudi v prisotnosti negativnih povezav in zaznavajo prisotnost negativnih ciklov. Klasičen primer je Bellman-Fordov algoritem, ki ga boste obravnavali kasneje.

1.3 Primeri

Grafi so zelo pogost način modeliranja relacij, iskanje najkrajših poti pa eden najobičajnejših problemov na njih. V nadaljevanju si bomo ogledali nekaj primerov sorodnih problemov.

1.3.1 Najširša pot

Recimo, da z grafom modeliramo cestno omrežje. Povezave predstavljajo dvosmerne ceste, vozlišča pa križišča. Uteži povezav ustrezajo širini ceste. Kakšna je največja širina vozila, ki se lahko pripelje od vozlišča A do B?

Gre za problem iskanja najširše poti (*widest path, maximum capacity path*). Pri njem iščemo pot od A do B, za katero bo veljalo, da je najmanjša utež na poti čim večja. Za primerjavo nas je v klasičnem problemu najkrajših poti zanimala tista pot, kjer je bila vsota uteži čim manjša. Vsoto smo torej zamenjali z minimumom, minimizacijo pa z maksimizacijo.

Uporabimo lahko povsem enak razmislek kot pri Dijkstrovem algoritmu. Poti do vozlišče bomo računali od širših proti ožjim. Najširša pot (∞ širine) vodi do začetnega vozlišča. Na vsakem koraku bomo med izračunana vozlišča dodali vozlišče iz okolice, do katerega vodi trenutno najširša potencialna pot. To ima zagotovo pravo vrednost, saj bi kakršnakoli druga pot obiskala več povezav, kar širine poti ne more povečati, temveč jo kvečjemu zmanjša.

1.3.2 Najdaljša pot

Kaj pa, če nas namesto najkrajše poti zanima najdaljša? Trivialno, uteži negiramo in je problem rešen. Žal ne, ker s tem dobimo graf z negativnimi cikli. Pravzaprav je koncept najdaljše poti slabo definiran - lahko bi se sprehajali sem in tja po isti povezavi in poljubno podaljšali pot.

V primeru najdaljše poti nas zanimajo poti brez ponovljenih vozlišč. Pri najkrajših poteh je bilo to samoumevno, saj od večkratnega obiskovanja vozlišč ni nobene koristi ampak samo škoda. Izkaže se, da gre za težek problem, ki spada v razred NP-polnih (*NP-complete*) problemov. Več o tem pa pri predmetu Izračunljivost in računska zahtevnost.

Izjema so usmerjeni aciklični grafi (DAG), ki ne vsebujejo ciklov. Tam smo že rešili prav ta problem, le da smo mu rekli kritična pot.

1.3.3 15 Puzzle

Verjetno poznate drsno sestavljanko prikazano na spodnji sliki. Igra se na mreži velikosti 4x4, kjer se na vsakem polju nahaja ploščica z enim izmed števil od 1 do 15. Vsako število se pojavi enkrat, eno polje pa je prazno. Zanima nas, kako naj s premiki ploščic na prazno sosednje polje uredimo števila po velikosti (po vrsticah od zgoraj navzdol in znotraj vrstice od leve proti desni). Še bolje, izračunajmo najmanjše potrebno število potez.

V tem primeru nimamo opravka z **grafom stanj**. Vsako stanje sestavljanke ustreza nekemu vozlišču. Za izračun najmanjšega števila potez bomo uporabili iskanje v širino (BFS). Seveda ne bomo vnaprej zgradili celotnega grafa, ker bi bil ta prevelik, ampak ga bomo odkrivali sproti. Rečemo, da bo graf predstavljen *implicitno* s stanji sestavljanke. Za vsako stanje oz. vozlišče znamo namreč izračunati njegove sosede. Pri tem upamo, da bomo dosegli rešitev dovolj zgodaj, preden bomo preiskali prevelik del grafa.

Obstajajo tudi izboljšave tega osnovnega preiskovanja, ki z uporabo hevristik usmerjajo iskanje proti delom grafa, v katerih je bolj verjetno, da bomo našli rešitev. Primer nadgradnje iskanja najkrajših poti z uporabi hevristik je algoritem $A*$.

[14]:
```
int puzzle15(VVI start, vector<VVI> &seq) {
    map<VVI, int> dist;
    map<VVI, VVI> prev;
    queue<VVI> q;
    q.push(start); dist[start]=0;
    VVI goal = {{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,0}};
    while (!q.empty()) {
```

```
        VVI state=q.front(); q.pop();
        if (state == goal) break;
        // next states
        for (int i=0;i<4;i++) for (int j=0;j<4;j++) if (state[i][j]==0) {  //␣
↪find empty cell
            for (auto [di,dj] : VII{{0,1},{0,-1},{1,0},{-1,0}}) {  // possible␣
↪moves
                int i2=i+di, j2=j+dj;
                if (i2<0 || i2>=4 || j2<0 || j2>=4) continue;
                VVI state2=state;  // adjacent state
                swap(state2[i][j], state2[i2][j2]);
                if (dist.count(state2)==0) {  // new?
                    dist[state2] = dist[state]+1;
                    prev[state2] = state;
                    q.push(state2);
                }
            }
        }
    }
    // reconstruct sequence of states
    VVI state=goal;
    seq.push_back(state);
    while (state!=start) {
        state = prev[state];
        seq.push_back(state);
    }
    reverse(seq.begin(),seq.end());
    return dist[goal];
}
```

[15]:
```
VVI state = {{5, 0, 2, 3},
             {6, 1, 7, 4},
             {9, 10,11,8},
             {13,14,15,12}};
vector<VVI> seq;
cout << puzzle15(state, seq) << endl;
for (VVI state : seq) {
    cout << endl;
    for (VI row : state) print(row);
}
```

```
9

5 0 2 3
6 1 7 4
9 10 11 8
13 14 15 12
```

```
5 1 2 3
6 0 7 4
9 10 11 8
13 14 15 12

5 1 2 3
0 6 7 4
9 10 11 8
13 14 15 12

0 1 2 3
5 6 7 4
9 10 11 8
13 14 15 12

1 0 2 3
5 6 7 4
9 10 11 8
13 14 15 12

1 2 0 3
5 6 7 4
9 10 11 8
13 14 15 12

1 2 3 0
5 6 7 4
9 10 11 8
13 14 15 12

1 2 3 4
5 6 7 0
9 10 11 8
13 14 15 12

1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
```

Za konec zgolj kot zanimivost omenimo še reševanje **Rubikove kocke**. Iskanje najkrajših poti je še vedno predmet algoritmičnega raziskovanja. S precej računske moči so nedavno dokazali, da je mogoče vsako stanje Rubikove kocke rešiti v največ 20 potezah oz. 26 potezah (če je ena poteza

rotacija ploskve samo za 90° in ne 180°).

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*.

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): $O(n^2)$, $O(n^2)$, $O(n)$

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 $U' < U$.

Racunska geometrija

December 18, 2024

1 Računska geometrija

Računska geometrije je področje algoritmov in podatkovnih struktur, ki se ukvarja z učinkovitim reševanjem geometrijskih problemov. Ti vključujejo delo s točkami, daljicami, večkotniki in drugimi geometrijskimi objekti ter relacijami med njimi, kot sta npr. razdalja ali vsebovanost. Računska geometrija je očiten del računalniške grafike, vida, robotike. Manj očitno pa je prisotna tudi v številnih drugih problemih, ki dopuščajo geometrijsko formulacijo.

Omejili se bomo na reševanje problemov v ravnini, saj se problemi v višjih dimenzijah običajno dodatno zakomplicirajo. Poleg tega je reševanje ravninskih problemov enostavno za vizualizacijo. Kljub temu pa moramo biti pri reševanju pozorni na številne *posebne primere*, kot so kolinearne točke, sovpadanje točk, vzporedne daljice, … Pomembna ovira je tudi *računska natančnost*. Če imamo opravka s celoštevilskimi objekti, želimo rešiti problem z uporabo celih števil, da ne vpeljemo računske napake, ki bi lahko povzročila povsem napačen rezultat.

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

typedef pair<int,int> PII;
typedef vector<PII> VII;
```

[2]:
```
template<class A, class B>
ostream& operator<<(ostream& os, pair<A,B> &p) {
    os << "(" << p.first << ", " << p.second << ")";
    return os;
}
```

[3]:
```
template<typename T>
void print(const vector<T> &sez) {
    for (T x : sez) cout << x << " ";
    cout << endl;
}
```

1.1 Razdalje in presečišča

Razdalje in presečišča so najbolj osnovni koncepti, ki jih moramo obvladati. Ne vključujejo kakšnih novih algoritmičnih prijemov, vendar služijo kot ponovitev geometrije in linearne algebre. Za naštete probleme seveda obstaja več formul. Ogledali si bomo najbolj enostavne, ki jih lahko izpeljemo brez prepisovanja iz kakšnega učbenika. Glede na predstavitev premic imamo različne pristope. Predpostavili bomo, da imamo premico P predstavljeno s točko P_0 in smernim vektorjem V_P.

Razdalje:

- Točki S in T: Pitagorov izrek poznamo še iz osnovne šole.
- Točka S in premica P: Izračunamo projekcijo S' točke S na premico P in izračunamo razdaljo med S in projekcijo S'. **Projekcijo** točke na premico izračunamo s pomočjo **skalarnega produkta**: $\text{proj}_b\, a = \frac{a \cdot b}{||b||^2} b$. Če delamo projekcijo na enotski smerni vektor premice, je dolžina projekcije kar enaka skalarnemu produktu.
- Točka S in daljica AB: Izračunamo projekcijo točke na nosilko (premico) daljice. Če je projekcija izven daljice, bo najkrajša razdalja do krajišča A ali B, sicer pa do projekcije S'.
- Daljici AB in CD: Predpostavimo, da se daljici ne sekata, sicer je odgovor 0. Najkrajša razdalja med daljicama bo enaka razdalji med enim izmed krajišč in drugo daljico. Izberemo najmanjšo izmed štirih možnosti.

Presečišča:

- Točka S in premica P: Če je vektorski produkt vektorja P_0S in smernega vektorja premice P enak 0, leži točka na premici.
- Točka S in daljica AB: Preverimo, ali točka leži na nosilki daljice in znotraj očrtanega pravokotnika (*bounding box*).
- Premici P in R: Če sta premici vzporedni, imamo neskončno ali nobenega presečišča. Sicer rešimo sistem enačb $P_0 + aV_P = R_0 + bV_R$ za obe koordinati.
- Premica in daljica: Izračunamo presečišče premice in nosilke daljice ter preverimo, ali leži presečišče na daljici.
- Daljici AB in CD: Ugotoviti moramo, ali se daljici sploh sekata, nato pa uporabimo rešitev za izračun presečišča nosilk obeh daljic. Daljici se sekata natanko takrat, ko sta krajišči prve daljice A in B na nasprotnih straneh nosilke daljice CD in obratno. **Stran/smer** ugotovimo s pomočjo **vektorskega produkta**. Točka A je na levi strani vektorja CD (v pozitivni smeri oz. nasprotni smeri urinega kazalca), če je vektorski produkt $CD \times CA$ pozitiven (na drugi strani bi bil negativen). Posebej pozorni moramo biti na primere, ko se daljici dotikata, kjer je vektorski produkt lahko 0.

1.2 Površina večkotnika

Začnimo s trikotnikom ABC. Če imamo podane koordinate oglišč, si lahko izberemo oglišče A za izhodišče in izračunamo polovico absolutne vrednost vektorskega produkta $p = \frac{1}{2}|AB \times AC|$.

Konveksen večkotnik lahko enostavno razbijemo na trikotnike in uporabimo prejšnji rezultat.

Na težave naletimo pri večkotnikih, ki niso nujno konveksni. Uporabimo formulo s predznačenimi vsotami trapezov $p = |\sum_{i=1}^{n} \frac{1}{2}(y_i + y_{i+1})(x_i - x_{i+1})|$. Predpostavimo lahko, da se večkotnik v celoti nahaja nad x-osjo (formula deluje tudi brez te predpostavke). Postavimo se nekam na x-os in opazujmo ozek vertikalen stolpec. Vsakič, ko bomo pri obhodu večkotnika prečkali stolpec

v desno stran, bomo območje pod njim odšteli, pri prehodu v levo pa prišteli. Marsikaj se bo izničilo in ostala bodo samo območja, ki imajo nad seboj liho število prečkanj (ta se izmenjujejo v levo in desno), kar je ravno notranjost večkotnika. Če naredimo obhod v drugo smer, bo rezultat negativen, po absolutni vrednosti pa enak.

Omenimo še, da deluje enak argument, če si izberemo poljubno izhodišče (npr. (0,0)) in seštevamo predznačene površine trikotnikov, ki jih z izbranim izhodiščem formirajo stranice na robu večkotnika.

1.3 Vsebovanost točke

Začnimo z najenostavnejšim primerom točke T, ki se nahaja v trikotniku ABC ali pač ne. Točka se nahaja v trikotniku, če se pri obhodu trikotnika ves čas nahaja na isti strani, kar preverimo z vektorskim produktom. Ali je to pozitivna ali negativna stran, je odvisno od smeri obhoda. Sledeči vektorski produkti morajo imeti enak predznak: $AB \times AT$, $BC \times BT$ in $CA \times CT$. Pozorni moramo biti, kaj problem zahteva v primeru, da se točka nahaja točno na robu trikotnika.

Naslednji primer je vsebovanost točke v konveksnem večkotniku. Enostavno ga lahko razbijemo na trikotnike (ki imajo skupno izbrano oglišče) in prevedemo problem na vsebovanost točke v trikotniku. Deluje pa tudi prej omenjeni pristop z lokacijo točke na isti strani obhoda večkotnika.

Kako pa rešimo problem za poljuben večkotnik (*point in polygon*), ki ni nujno konveksen? V tem primeru uporabimo tehniko metanja žarka (*ray casting*). Če sledimo poltraku iz točke T v poljubno smer, se ob vsakem križanju z robom večkotnika spremeni lokacija znotraj/zunaj. Če je število križanj liho, je točka znotraj večkotnika, sicer je izven. Pomembna podrobnost je, kaj se zgodi, če žarek seka večkotnik v enem od oglišč. Sprememba je namreč odvisna od sosednjih oglišč. Če več sosednjih oglišč leži na žarku, nas zanima prvo oglišče, ki ne. Če sta obe na isti strani žarka, ni spremembe, sicer pa je. Prikladna izbira smeri je npr. $z = (-1, 0)$. Lahko pa se tej komplikaciji izognemo s tako izbiro smeri (naključno), da do tega ne pride.

V spodnji implementaciji bomo predpostavili, da se točka ne nahaja na robu večkotnika. Če to ni res, bi lahko to posebej preverili. Pretvarjali se bomo, da ima točka za ϵ večjo y koordinato. To ne spremeni rešitve, vendar poenostavi razmislek, ker so vsa oglišča nad ali pod njo, ne pa na isti višini (oglišča z enako višino bodo obravnavana kot nižja). Problem bi z malo več truda lahko rešili tudi v celih številih, vendar zaradi preglednosti ne bomo dodatno kompliciral.

[4]:
```
#define OPERATOR_SUBTRACT operator- // workaround for a bug of cling
```

[5]:
```
PII OPERATOR_SUBTRACT(PII a, PII b) {
    return {a.first-b.first, a.second-b.second};
}
```

[6]:
```
int point_in_polygon(vector<PII> poly, PII t) {
    int n=poly.size(), cnt=0;
    auto [x,y] = t;
    for (int i=0;i<n;i++) {
        int j=(i+1)%n;
        if ((poly[i].second<=y) != (poly[j].second<=y)) {  // stranica seka␣
 ↪vodoravno premico
            PII s = poly[j]-poly[i];  // vektor stranice: i -> i+1
```

```
            PII v = t-poly[i];  // vektor do tocke: i -> t
            double k = (double)v.second/s.second;
            double xp = poly[i].first + k*s.first;  // presecisce z vodoravno␣
 ↪premico
            if (xp < x) cnt++;
        }
    }
    return cnt%2;
}
```

[7]:
```
vector<PII> poly = {{0,0}, {1,1}, {3,1}, {4,2}, {5,1}, {6,2}, {7,0}, {8,1},␣
 ↪{9,0}, {10,1}, {10,3}, {0,3}};
cout << point_in_polygon(poly, {9,1}) << endl;
cout << point_in_polygon(poly, {9,2}) << endl;
cout << point_in_polygon(poly, {6,1}) << endl;
cout << point_in_polygon(poly, {5,0}) << endl;
```

```
1
1
0
0
```

1.4 Konveksna ovojnica

Konveksna ovojnica/ogrinjača/lupina (*convex hull*) množice točk v ravnini je najmanjša konveksna množica, ki vsebuje vse podane točke. Običajno nas zanima rob konveksne ovojnice, ki je najkrajša sklenjena črta, ki vsebuje vse točke. Včasih tudi lomljeni črti, ki predstavlja rob konveksne ovojnice, rečemo kar konveksna ovojnica. Predstavljamo si jo lahko kot elastiko, ki se skrči okoli množice točk.

Iščemo ekstremne (robne) točke na robu ovojnice, ki jo definirajo. V primeru več kolinearnih točk na robu, je stvar definicije problema, ali želimo poročati samo oglišča ali tudi točke vzdolž stranic konveksne ovojnice. V nadaljevanju se bomo omejili na primere, kjer ni treh kolinearnih točk.

Ogledali si bomo par najbolj klasičnih algoritmov, obstaja pa jih še veliko več. Problem seveda postane težji, če ga rešujemo v treh ali še več dimenzijah.

[8]:
```
vector<PII> points = {{4,0}, {2,3}, {5,2}, {6,1}, {8,4}, {6,6}, {5,4}, {4,5},␣
 ↪{2,6}, {1,1}, {1,5}, {3,2}};
```

1.4.1 Identifikacija stranic

Rob konveksne ovojnice je sestavljen iz daljic med pari točk. Če lahko za posamezen par točk oz. daljico med njima ugotovimo, ali je del konveksne ovojnice, lahko zgradimo konveksno ovojnico. Daljica je del roba konveksne ovojnice, če se vse ostale točke nahajajo na isti strani (recimo na levi/pozitivni). Časovna zahtevnost takega postopka je $O(n^3)$, kjer je n število točk.

[9]:
```
int cross(PII u, PII v) {
    return u.first*v.second - u.second*v.first;
```

```
}
```

[10]:
```
int n=points.size();
for (int i=0;i<n;i++) {
    for (int j=0;j<n;j++) if (i!=j) {  // vektor daljice i-j
        PII d=points[j]-points[i];
        int ok=1;
        for (int k=0;k<n;k++) if (k!=i && k!=j) {
            PII v=points[k]-points[i];
            if (cross(d,v)<0) ok=0;
        }
        if (ok) cout << char('A'+i) << " " << char('A'+j) << endl;
    }
}
```

```
A D
D E
E F
F I
I K
J A
K J
```

Stranice seveda niso izpisane v vrstnem redu, kot si sledijo na konveksni ovojnici, vendar bi jih lahko uredili, če bi bilo treba.

1.4.2 Zavijanje darila

Pri iskanju konveksne ovojnice smo lahko bolj učinkoviti. Naraven pristop zavijanja darila (*gift wrapping, Jarvis march*) začne z izbiro točke, ki je gotovo del konveksne ovojnice. V ta namen lahko izberemo npr. najbolj levo točko A (najnižjo med najbolj levimi) in raztegnemo ovojni papir navzgor. Papir ovijamo v smeri urinega kazalca dokler se ne dotakne naslednje točke. Postopek ovijanja ponavljamo, dokler ne pridemo do začetne točke.

Naslednjo točko, ki se jo dotakne papir pri ovijanju, lahko poiščemo na več načinov. Ker so med gradnjo konveksne ovojnice vedno vse točke na isti strani zadnje točke A (del neke polravnine skozi A), lahko med njimi poiščemo najbolj levo z uporabo vektorskega produkta $AC \times AB$ za primerjavo, ali je točka C bolj levo (oz. v nasprotni smeri urinega kazalca) od točke B.

[11]:
```
VII gift_wrapping(VII points) {
    int n=points.size();
    PII start=*min_element(points.begin(), points.end());
    vector<PII> hull;
    PII a=start;
    while (a!=start || hull.empty()) {
        hull.push_back(a);
        PII b = (a!=points[0])?points[0]:points[1];  // katerakoli tocka, ki ni␣
↪a
        for (PII c : points) if (c!=a) {
```

```
            PII ac=c-a, ab=b-a;  // vektorja AC, AB
            if (cross(ab,ac)>0) b=c;
        }
        a = b;
    }
    return hull;
}
```

[12]:
```
auto hull = gift_wrapping(points);
print(hull);
```

```
(1, 1) (1, 5) (2, 6) (6, 6) (8, 4) (6, 1) (4, 0)
```

Časovno zahtevnost algoritma lahko analiziramo v odvisnosti od velikosti rezultata (*output-sensitive*) - število točk h na konveksni ovojnici. V tem primeru je časovna zahtevnost $O(hn)$. Če pa velikosti rezultata ne upoštevamo, so lahko v najslabšem primeru vse točke na robu ovojnice, zato je časovna zahtevnost $O(n^2)$.

1.4.3 Grahamov pregled

Konveksno ovojnico točk v ravnini lahko poiščemo bolj učinkovito kot v kvadratnem času in sicer z uporabo Grahamovega pregleda (*Graham scan*). Ponovno si izberimo neko ekstremno točko T, ki je zagotovo del konveksne ovojnice (npr. najnižjo med najbolj levimi točkami). Uredimo preostale točke glede na kote vektorjev iz točke T (od tistih, ki kažejo navzdol, proti vodoravnim in tistim, ki kažejo navzgor). Naj bo ta urejen seznam točk $P_1, P_2, \ldots$ Če jih povežemo, dobimo ovojnico, ki vsebuje vse točke (kar v ogliščih), vendar ni konveksna. Vemo tudi, da bo konveksna ovojnica neka podmnožica tega seznama točk. Vrstni red točk je že pravilen, samo izbrati moramo prave.

Algoritem gradi konveksno ovojnico postopno z dodajanjem novih točk v izbranem vrstnem redu po kotih. Po vsaki dodani točki popravi konveksnost zgrajene ovojnice, če je nova točka podrla konveksnost z obratom v napačno smer. To naredi z odstranjevanjem točk s konca zgrajene ovojnice, dokler zaključek ovojnice z novo točko ni konveksen.

Grahamov pregled pravzaprav gradi vedno večjo konveksno ovojnico z dodajanjem posameznih točk po kotih. V i-tem koraku doda točko P_i in iz konveksne ovojnice točk $T, P_1, P_2, \ldots, P_{i-1}$ izračuna konveksno ovojnico točk $T, P_1, P_2, \ldots, P_i$.

Časovna zahtevnost algoritma je zaradi urejanja $O(n \log n)$. Preostanek algoritma vključuje dodajanje in odstranjevanje točk z ovojnice, vendar je vsaka točka lahko dodana in odstranjena kvečjemu enkrat. Zato je ta del algoritma linearen v odvisnosti od števila točk.

[13]:
```
VII graham_scan(VII points) {
        int n=points.size();
    PII t=*min_element(points.begin(), points.end());

    vector<pair<double,PII>> angles;
    for (PII p : points) if (p!=t) {
        PII v = p-t;
        angles.push_back({atan2(v.second, v.first), p});
    }
```

```
    sort(angles.begin(), angles.end());

    vector<PII> hull = {t};  // stack
    for (auto [_,c] : angles) {
        while (hull.size()>=2) {  // restore convexity
            PII a=hull[hull.size()-2], b=hull[hull.size()-1];
            PII ab=b-a, ac=c-a;
            if (cross(ab,ac)>0) break;
            hull.pop_back();
        }
        hull.push_back(c);
    }
    return hull;
}
```

[14]:
```
auto hull2 = graham_scan(points);
print(hull2);
```

```
(1, 1) (4, 0) (6, 1) (8, 4) (6, 6) (2, 6) (1, 5)
```

Racunska zahtevnost

December 18, 2024

1 Računska zahtevnost

Poskusimo odgovorit na par vprašanj, ki si jih lahko zastavimo v zvezi s prejšnjimi urejevalnimi algoritmi.

- Kateri algoritmi so dobri in kateri slabi?
- Kateri algoritem je najboljši oz. kateri izmed dveh je boljši?
- Kako merimo učinkovitost oz. računsko zahtevnost algoritma?

Za algoritma s permuacijami lahko brez škode rečemo, da sta slaba. Poznamo precej hitrejše postopke urejanja, ki niso bistveno kompleksnejši (morda celo enostavnejši). Za ostale osnovne algoritme urejanja pa že ni povsem jasnega odgovora. Poznamo namreč učinkovitejše vendar tudi kompleksnejše algoritme. Tudi osnovni pristopi so lahko povsem primerni.

Pri iskanju najboljšega algoritma naletimo na podobno dilemo. Poleg tega ni jasno, na kakšnih podatkih želimo, da je algoritem najboljši - povsem naključnih, kakšnih posebnih, kako velikih?

To nas pripelje do tretjega vprašanja, kako sploh merimo učinkovitost algoritma?

- Lahko merimo **čas izvajanja**, vendar je te čase problematično primerjati na različnih računalnikih.
- Lahko merimo **število operacij**, ki jih potrebuje algoritem. Dogovoriti pa se moramo, *katere operacije* bomo šteli (primerjave, aritmetične, logične, pomnilniške, …)
- Dogovoriti se moramo, kakšen **primer podatkov** bomo obravnavali (najboljšem, najslabšem, povprečnem).
- Dogovoriti se moramo o **velikosti primerov**, s katerimi imamo opravka. En algoritem je lahko boljši za manjše primere, drugi pa se izkaže pri večjih.

Kot bomo videli v nadaljevanju, običajno ocenjujemo asimptotično zgornjo mejo števila operacij v najslabšem primeru.

Računska zahtevnost (kompleksnost) je količina virov, ki jih potrebuje algoritem za rešitev problema dane velikosti. Pri virih se običajno osredotočamo na čas in prostor, zato govorimo o **časovni** in **prostorski zahtevnosti**.

Ker imamo lahko različne podatke enake velikosti, moramo definirati, ali gre za **najboljšo**, **najslabšo** ali **povprečno** računsko zahtevnost. Običajno se osredotočamo na najslabšo (*worst-case*), če ni določeno drugače.

Natančno količino virov je pogosto težko izračunati, poleg tega pa ni pretirano praktično uporabna. Na računalniku z malenkost drugačno arhitekturo je že lahko drugačna. Poleg tega pa nas za majhne probleme običajno ne zanima, ker je takrat preglednost bolj pomembna od učinkovitosti. Zato se

običajno ukvarjamo z **asimptotično zahtevnostjo**, ki opisuje porabo virov algoritma pri zelo velikih problemih. Pri tem pogosto ocenjujemo neko mejo asimptotične zahtevnosti. Najpogosteje ocenjujemo **zgornjo mejo**, za kar se uporablja **notacija z velikim O-jem** (*Big O notation*). Rečemo, da ima funkcija $f(n)$ kompleksnost reda $g(n)$, kar zapišemo kot $O(g(n))$ ali $f(n) \in O(g(n))$ ali celo kar $f(n) = O(g(n))$ (čeprav ne gre za enakost). Formalno to pomeni:

$\exists k > 0\ \exists n_0\ \forall n > n_0 \colon f(n) \leq k\, g(n)$

ali enakovredno z limitami

$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

Poleg zgornje meje asimptotične zahtevnosti (veliki O) poznamo še notacije za druge meje (velika omega - Ω, velika theta - Θ, ...). Več o njih pa pri drugih algoritmičnih predmetih. Omenjene definicije lahko posplošimo tudi na funkcije z več spremenljivkami, če opazujemo časovno zahtevnost algoritma v odvisnosti od več parametrov velikosti problema.

> Najpogostejši primer je analiza zgornje meje asimptotične računske zahtevnosti v najslabšem primeru. S tem postavimo pesimistično oceno za najbolj neugoden primer velikih podatkov. Kadar govorimo o *časovni zahtevnosti*, običajno mislimo kar zgornjo mejo asimptotične časovne zahtevnosti v najslabšem primeru, če seveda ni pojasnjeno drugače.

Recimo, da smo izračunali čas izvajanja oz. število operacij za rešitev problema velikost n s funkcijo $f(n) = \frac{1}{2}(n-1)(n+2)\log n + \sqrt{n}$. Če izraz razširimo, dobimo $f(n) = \frac{1}{2}n^2 \log n + \frac{1}{2}n \log n - \log n + \sqrt{n}$. Časovno zahtevnost takega algoritma bi lahko ocenili kot $O(2n^3)$, kar je sicer pravilno, vendar precej nenatančna meja. Boljša ocena časovne zahtevnosti bi bila $O(n^2 \log n)$. Vsi ostali členi so namreč zanemarljivi v primerjavi z $n^2 \log n$, ko gre n proti neskončnosti (za potrebe zgornje meje bi jih lahko nadomestili z $n^2 \log n$), konstantni člen pred njim pa po definiciji ni relevanten. Primeren (ne pa edini) izbor konstant v zgornji definiciji bi bil npr. $n_0 = 2$ in $k = 3$, ker so vsi trije pozitivni členi manši ali enaki $n^2 \log n$ pri $n >= 2$. V praksi to pomeni, da:

- pri vsoti obdržimo samo najhitreje rastoči člen,
- pri produktu pa lahko zanemarimo konstantne faktorje.

Tipične časovne zahtevnosti so:

- $O(1)$, konstantna (neodvisna od velikosti problema n)
- $O(\log n)$, logaritemska
- $O(\sqrt{n})$, korenska
- $O(n)$, linearna
- $O(n \log n)$, loglinearna, linearitmična
- $O(n \log^c n)$ za konstanto $c > 0$, npr. $O(n \log^2 n)$ kvazilinearna
- $O(n^2)$, kvadratna
- $O(n^3)$, kubična
- $O(n^c)$ za konstanto $c > 0$, npr. $O(n^5)$, polinomska
- $O(c^n)$ za konstanto $c > 1$, npr. $O(2^n)$, eksponentna

Kako velike probleme lahko rešujemo z algoritmi določene časovne zahtevnosti, npr. $O(n^2)$? Ker ta sintaksa skriva konstantni faktor, tega ne moremo reči natančno. Dobra praktična ocena pa je, da lahko na tipičnem osebnem računalniku trenutno izvedemo približno 10^8 osnovnih operacij na sekundo.

1.0.1 Primeri

Oglejmo si nekaj primerov funkcij, ki predstavljajo računske zahtevnosti, in jih ocenimo z notacijo z velikim O-jem.

- $f_1(n) = 100 + 2n + 3n^2 = O(n^3)$ (ali $O(n^4 \log n)$, kar je sicer pravilna, vendar slaba meja)
- $f_2(n) = 3n\cos(2\pi n) + \frac{n}{\log n} + 2n = O(n)$
- $f_3(n) = 1 + n\log n + n^{1.5} = O(n^{1.5})$ (da logaritem raste počasneje kot koren, se lahko prepričate z uporabo l'Hôpitalovega pravila za izračun $\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$)

Funkcija lahko vsebuje vsote kakšnih vrst.

- $f(n) = \sum_{k=1}^{n} n/k = n\sum_{k=1}^{n} 1/k = O(n\log n)$ (Harmonična vrsta)

Pogoste so tudi rekurzivne funkcije.

- $f(n) = n + f(n/2) = n + n/2 + n/4 + ... < 2n = O(n)$

Lahko imamo funkcije več spremenljivk.

- $f(n,m) = an^2 + n\sqrt{m} + b\log m = O(n^2 + n\sqrt{m})$ (a in b sta konstanti)

Parametriziran algoritem Načrtujemo algoritem, v katerem bomo problem velikosti n enakomerno razbili na skupine velikosti k, ki jih bo torej n/k. Izračunali smo, da lahko problem za posamezno skupino rešimo z algoritmom s korensko časovno zahtevnostjo (v odvisnosti od velikosti skupine), časovna zahtevnost postopka združevanja rezultatov več skupin pa je kubična (v odvisnosti od števila skupin). Kako naj izberemo parameter k, da bo časovna zahtevnost algoritma čim boljša?

$f(n;k) = n/k \cdot O(\sqrt{k}) + O((n/k)^3)$. Oglejmo si ekstremne primere. Pri $k = 1$ dobimo $f(n) = n + n^3 = O(n^3)$, pri $k = n$ pa $f(n) = \sqrt{n} + 1 = O(\sqrt{n}) = O(n^{0.5})$. V prvem primeru je večji drugi člen, v drugem primeru pa prvi člen. Želimo, da noben od njiju ne dominira, torej naj bosta enaka. Iz enačbe $n\sqrt{k}/k = (n/k)^3$ lahko določimo $k = n^{4/5}$ in $f(n) = O(n^{2/5}) = O(n^{0.4})$.

Analiza programa Ocenimo časovno zahtevnost spodnjega programa.

```
for (int x = 1; x <= n; x *= 2) {
  for (int i = 0; i < x; i++) {
    for (int j = 0; j < n; j += 2) {
       // konstantno število operacij
    }
    for (int j = 1; j < n; j *= 2) {
      // konstantno število operacij
    }
  }
}
```

Na for zanke se bomo sklicevali kar s prva, druga, tretja in četrta, kot se pojavijo v programu. Določimo, največ kolikokrat se izvede katera od njih: prva $\log n$-krat, druga: n-krat, tretja: $n/2$-krat in četrta: $\log n$-krat. Tretja in četrta se izvedeta zaporedno, pri čemer dominira tretja. Časovno zahtevnost lahko zato ocenimo z $O(\log n \cdot n \cdot (n/2 + \log n)) = O(n^2 \log n)$.

Pri ocenjevanju časovne zahtevnosti pa smo lahko bolj natančni. Število ponovitev druge zanke je namreč odvisno od trenutne iteracije prve zanke (v prejšnjem odstavku smo vzeli kar najbolj pesimistično oceno). Število izvedb druge zanke bo $1 + 2 + 4 + 8 + ... + n = O(n)$, za vsako od teh ponovitev pa tretja zanka prispeva še $O(n)$ operacij. Bolj natančna ocena časovne zahtevnosti je torej $O(n^2)$.

Uvod

December 18, 2024

1 Algoritmi in podatkovne strukture 1

Namen predmeta APS1 je naučiti udeležence *algoritmičnega razmišljanja*. Ukvarjali se bomo s *pravilnostjo* in *učinkovitostjo* algoritmov. Spoznali bomo več osnovnih algoritmov in z njimi povezanih podatkovnih struktur, ki bodo predstavljali našo osnovno orodjarno. Poleg tega bodo služili kot primeri, na katerih se bomo učili načrtovanja ter analiziranja algoritmov in podatkovnih struktur. S konkretnimi implementacijami zasnovanih idej pa bomo utrjevali in poglabljali znanje programiranja.

Algoritem - Postopek za reševanje določenega problema, ki ga reši v končnem številu korakov in je nedvoumno opisan s končnim številom ukazov, izvedljivih mehanično brez rabe uma in običajno podanih v obliki psevdokode, diagrama poteka ali v nekem programskem jeziku.

Primeri: iskanje največjega elementa v seznamu, Evklidov algoritem, Kruskalov algoritem, …

Včasih med algoritme štejemo tudi nekoliko bolj dvoumne postopke opisane v naravnem jeziku, kot so npr. recepti, razni birokratski postopi (npr. postopek za prijavo začasnega prebivališča) itd.

Podatkovna struktura - Način organizacije podatkov, ki omogoča učinkovito izvajanje operacij na njih.

Primeri: seznam, množica, uravnoteženo iskalno drevo, …

Algoritmi in podatkovne strukture se močno prepletajo. Algoritmi (postopki) v svojih korakih potrebujejo učinkovito organizirane podatke (podatkovne strukture). In obratno. Podatkovne strukture potrebujejo določene postopke (algoritme), da vzdržujejo podatke organizirane in omogočajo učinkovite operacije. Glede na delež enega ali drugega, govorimo o algoritmu ali podatkovni strukturi.

1.1 Izvajanje predmeta

Predmet se izvaja v tedenskih sklopih s predavanji, ki jim sledijo vaje. Na predavanjih so bomo ogledali teorijo in kakšen praktičen primer rešili tudi skupaj. Na vajah boste vsak teden samostojno (ob pomoči asistentov) reševali programersko nalogo v programskem jeziku C++, ki bo vključevala nek algoritmičen problem povezan z vsebino predavanj. Za domačo naloga vas bo čakal podoben problem. Ob koncu sklopa se bodo vaše rešitve avtomatsko testirale. Za opravljeno nalogo mora vaša rešitev uspešno prestati vse primere. Prejeli boste povratno informacijo o številu uspešno prestalih testov, ne pa o njihovi vsebini. Kdor bo želel svojo rešitev popraviti, bo imel za to čas do konca naslednjega sklopa.

1.2 Delovno okolje

Vaše programe bomo ocenjevali na Ubuntu 22.04 s prevajalnikom GCC 11 in s standardom C++20. Če je vaš oddani program resitev.cpp, ga bomo prevedli in pognali z

```
g++ -std=c++20 -o program resitev.cpp
./program < test.in > test.res
```

Če si boste pripravljali virtualko z Ubuntu 22.04, morate dodatno namestiti build-essential paket, s katerim dobite C++ prevajalnik.

```
sudo apt install build-essential
```

1.3 Ocenjevanje

Za pristop k izpitu morate imeti uspešno opravljeno sprotno delo, kar pomeni povsem pravilno rešenih vsaj 50% tedenskih nalog. Skoraj pravilna rešitev je še vedno nepravilna in je zato ne upoštevamo. Oceno predmeta prejmete na izpitu, ki ga rešujete na papir.

Če odkrijemo dve ali več prepisanih rešitev, vsem udeleženim ne priznamo sprotnega dela in tako v tekočem študijskem letu ne morete opravljati izpita. Zato ne objavljajte svojih rešitev. Posebej drzne kršitve bomo prijavili disciplinski komisiji.

1.4 Literatura

Algoritmi in podatkovne strukture:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.
- Sedgewick R. & Wayne K. (2011). Algorithms fourth edition. Addison-Wesley.
- Aho A. V. Hopcroft J. E. & Ullman J. D. (1983). Data structures and algorithms. Addison-Wesley.
- Kononenko, I., Robnik Šikonja, M., & Bosnić, Z. (2008). Programiranje in algoritmi. Fakulteta za računalništvo in informatiko.

Programiranje v C++:

- cplusplus, cppreference
- Modern C++ for C Programmers
- Stroustrup, B. (2013). The C++ Programming Language.-4th. Addison-Wesley.

1.5 Dodatne vaje

Na spletu je kup strani, ki omogočajo reševanje programerskih in algoritmičnih nalog (po tematikah) s preverjanjem pravilnost: Codeforces, SPOJ, LeetCode, HackerEarth, HackerRank, ...

Vpeta drevesa

December 18, 2024

1 Disjunktne množice

Čeprav poglavje obljublja delo z vpetimi drevesi, se bomo najprej posvetili neki drugi podatkovni strukturi, ki nam bo kasneje prišla prav. Prav pa nam pride v številnih aplikacijah, kjer imamo opravka z združevanjem množic ali kakšnih drugih ekvivalenčnih razredov objektov.

Podatkovna struktura disjunktnih množic (*disjoint-set*) hrani množico disjunktnih množic (ali razbitje množice na podmnožice) in omogoča naslednje operacije:

- `add(x)`: Doda novo množico $\{x\}$ z enim samim elementom.
- `find(x)`: Najde množico, ki ji pripada element x.
- `union(x,y)`: Združi množici elementov x in y.

Poleg disjunktnih množic se za to podatkovno strukturo uporablja tudi izraz **union-find**. Pogosto se problemi začnejo s množicami posameznih elementov, ki jih nato združujemo z uporabo funkcij `union` in `find`, zato se bomo omejili na ta primer. Dopolnitev razvitih rešitev s funkcijo `add` za dodajanje novega elementa je enostavna.

Posamezne množice bomo predstavili z drevesi. Koren drevesa pa bo predstavnik posamezne množice. Funkcija `find(x)` bo torej morala poiskati in vrniti koren drevesa, funkcija `union(x,y)` pa združiti dve drevesi v eno. Koren drevesa z elementom x lahko pripnemo kot otroka korenu drevesa z elementom y. Združevanje je torej učinkovito, vendar lahko s takimi združevanji nastanejo zelo izrojena drevesa, zato je časovna zahtevnost operacije `find` linearna.

Ker imamo opravka z dvema funkcijama, pri analizi učinkovitosti običajno opazujemo zaporedje $n-1$ združevanj (kar postopoma združi vseh n posameznih elementov v eno samo množico), med tem pa izvedemo še $m \geq n$ klicev funkcije find.

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

typedef pair<int,int> PII;
typedef vector<int> VI;
typedef vector<pair<int,int>> VII;
typedef vector<vector<int>> VVI;
```

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

1.0.1 Združevanje po velikosti

Prva izboljšava temelji na pametnejšem združevanju. Pri združitvi dveh dreves je smiselno manjšega pripeti k večjemu. Velikost drevesa lahko merimo po število vozlišč (*union by size*) ali po oceni višine (*union by rank*). Osredotočili se bomo na prvo možnost, ker dobimo z drugo enake rezultate.

Ob vsaki združitvi se višina drevesa lahko poveča za največ 1 (če združujemo enako globoki drevesi). Pri združevanju postane vozlišče manjšega drevesa del vsaj dvakrat večjega združenega drevesa. Zato lahko vsako vozlišče nastopa v največ $O(\log n)$ združevanjih (sicer bi moralo imeti združeno drevo več kot n vozlišč, kar ni mogoče). Časovna zahtevnost operacije join je $O(1)$, find pa $O(\log n)$.

1.0.2 Stiskanje poti

Druga možna izboljšava temelji na iskanju korena drevesa (find). Če smo že prehodili dolgo pot, da smo našli koren, bi lahko vsa vozlišča na poti tudi povezali direktno nanj, da nam kasneje ne bo treba tega početi ponovno.

Če imamo opravka samo z operacijami find (brez združevanj), je amortizirana časovna zahtevnost operacije find $O(1)$ (v zaporedju $m \geq n$ find-ov). V zaporedju operacij find bomo vsako vozlišče pri iskanju korena prehodili enkrat (morda jih bomo prehodili cel kup že v prvi operaciji in kasneje nobenega, ali pa v vsaki operaciji nekaj, skupaj pa ravno vse).

Če upoštevamo še združevanja, je amortizirana analiza nekoliko kompleksnejša. Povejmo samo, da je časovna zahtevnost postopnega združevanja vseh elementov v eno množico ($n-1$ operacij join) z $m \geq n$ vmesnimi operacijami find enaka $O(m \log n)$. Amortizirana zahtevnost operacije find je torej $O(\log n)$. S strategijo združevanja po velikosti smo dosegli enako zahtevnost, ki pa ni bila amortizirana.

1.0.3 Skupna rešitev

Obe izboljšavi lahko tudi združimo, saj ne vplivata ena na drugo. Združevanje po velikosti skrajša poti, ki jih stiskanje poti kasneje še dodatno skrajša. Stiskanje poti ne spremeni velikosti drevesa, temveč ga zgolj preuredi, zato ne vpliva na združevanje po velikosti.

Rezultat je podatkovna struktura s skoraj konstantnimi amortiziranimi časovnimi zahtevnostmi posameznih operacij. Časovna zahtevnost je $O(m \log^* n)$, še tesnejša meja pa je $O(m\alpha(n))$. Obe funkciji (iterirani logaritem in inverzna Ackermannova funkcija) rasteta izjemno počasi in sta praktično konstantni za vse razumne vrednost, npr. $n = 2^{65536} \approx 10^{20000}, \log_2^*(n) = 5, \alpha(n) = 4$. Amortizirana časovna zahtevnost posamezne operacije v procesu združevanja posameznih elementov v eno končno množico je torej praktično konstantna!

[3]:
```
class DisjointSet {  // Union-Find
public:
    vector<int> parent, size;
```

```
    DisjointSet(int n) {
        parent = vector<int>(n);
        size = vector<int>(n);
        for (int i=0;i<n;i++) {  // individual sets
            parent[i] = i;
            size[i] = 1;
        }
    }

    int root(int x) {  // find
        if (parent[x]==x) return x;  // reached the root
        int r = root(parent[x]);
        parent[x] = r;  // path compression
        return r;
    }

    void join(int x, int y) {  // union by size
        x=root(x); y=root(y);  // replace by roots
        if (x==y) return;
        if (size[x]>size[y]) swap(x,y);  // make x smaller
        parent[x] = y;  // attach to larger root
        size[y] += size[x];
    }
};
```

[4]:
```
DisjointSet ds(10);
ds.join(3,4); ds.join(5,7); ds.join(0,3); ds.join(8,9); ds.join(7,9);
cout << (ds.root(3) == ds.root(7)) << endl;
cout << (ds.root(5) == ds.root(8)) << endl;
```

```
0
1
```

2 Minimalno vpeto drevo

Vpeto drevo (*spanning tree*) grafa G je drevo T, ki vključuje vsa vozlišča grafa G in podmnožico njegovih povezav. **Minimalno vpeto drevo** (*minimum spanning tree, MST*) je tisto vpeto drevo, ki ima najmanjšo vsoto uteži povezav. Če imamo opravka z več komponentami, govorimo o minimalnem povezanem gozdu. Tam za vsako povezano komponento ločeno poiščemo minimalno vpeto drevo.

Vpeto drevo lahko enostavno poiščemo s preiskovanjem v širino ali globino iz poljubnega vozlišča. Kako pa poiščemo minimalno vpeto drevo?

[5]:
```
ifstream fin("mst.txt");
int n,m;
fin >> n >> m;
vector<VI> edges;
```

```
vector<VII> adj(n);
for (int i=0;i<m;i++) {
    int a,b,w;
    fin >> a >> b >> w;
    edges.push_back({a,b,w});
    adj[a].push_back({b,w});
    adj[b].push_back({a,w});
}
```

2.0.1 Prerezna lastnost

Razbitju vozlišč grafa na dve disjunktni množici pravimo **prerez grafa** (*cut*). Povezavam s krajišči v različnih delih razbitja pa **prerezne povezave** (*cut-edge, cut-set*).

Prerezna lastnost (*cut property*) pravi, da je najmanjša prerezna povezava vedno del nekega minimalnega vpetega drevesa (ne glede na izbrani prerez). Naj bo e najmanjša prerezna povezava v razbitju vozlišč na množici A in $B = V - A$. Recimo, da ta povezava ni del nobenega minimalnega vpetega drevesa. Potem mora v minimalnem vpetem drevesu obstajati neka druga povezava e' med A in B. Vemo, da je $w(e) \leq w(e')$. Povezavo e' lahko zamenjamo z e in pri tem ohranimo ali zmanjšamo vsoto povezav v vpetem drevesu.

2.1 Prim

Primov algoritem je požrešen algoritem, ki gradi minimalno vpeto drevo s širjenjem od izhodiščenega vozlišča navzven proti sosedom. Za izhodišče lahko uporabimo poljubno vozlišče, saj morajo biti vsa del minimalnega vpetega drevesa. Oglejmo si prerez grafa na množico A, ki vključuje vsa vozlišča do sedaj zgrajenega drevesa in množico B, ki vsebuje preostala. Iz prerezne lastnosti sledi, da je najmanjša povezava med A in B del nekega minimalnega vpetega drevesa. Zato jo lahko dodamo v drevo in ponovimo enak razmislek.

Analizirajmo časovno zahtevnost takega postopka. V drevo moramo dodati n vozlišč, vsakič pa moramo obravnavati m povezav, da najdemo najmanjšo med že dodanimi vozlišči in preostankom. Časovna zahtevnost bi bila $O(nm)$.

Lahko pa jo izboljšamo. Za vsako še nedodano vozlišče bomo vzdrževali njegovo razdaljo do že zgrajenega drevesa. Na začetku so vse te razdalje enake ∞, razen za začetno vozlišče, ki ima razdaljo 0. Na vsakem koraku poiščemo vozlišče z najmanjšo razdaljo, ga dodamo v drevo in posodobimo razdalje do drevesa vseh njegovih sosedov. Vse skupaj bomo obravnavali $O(m)$ povezav. Na vsakem koraku dodajanja novega vozlišča v drevo pa bomo iskali vozlišče z najmanjšo razdaljo do drevesa. Časovna zahtevnost je $O(n^2 + m) = O(n^2)$.

Namesto večkratnega iskanja vozlišča z najmanjšo razdaljo lahko hranimo vozlišča v prioritetni vrsti podobno kot v Dijkstrovem algoritmu. Posodobljene razdalje dodajamo v vrsto, če dobimo iz vrste kakšno staro vrednost, pa jo ignoriramo. Časovna zahtevnost take implementacije je $O(m \log n)$.

[6]:
```
int Prim(int n, vector<VII> &adj, vector<PII> &mst) {
    vector<int> dist(n,-1);  // distance from the tree
    vector<int> done(n), parent(n);
    int cost=0;
    priority_queue<PII, vector<PII>, greater<PII>> pq;
```

```
    dist[0]=0; pq.push({0,0});
    while (!pq.empty()) {
        auto [d,x]=pq.top(); pq.pop();
        if (done[x]) continue;  // ignore old items in queue
        cost+=d;
        done[x]=1;
        for (auto [y,w] : adj[x]) if (!done[y]) {  // update unfinished␣
 ↪neighbors
            if (dist[y]==-1 || w<dist[y]) {  // new or smaller distance
                dist[y]=w; pq.push({w,y});
                parent[y]=x;
            }
        }
    }
    for (int x=1;x<n;x++) {  // skip root
        mst.push_back({x,parent[x]});
    }
    return cost;
}
```

[7]:
```
vector<PII> mst;
cout << Prim(n, adj, mst) << endl;
for (PII edge : mst) cout << edge.first << " " << edge.second << endl;
```

```
37
1 0
2 1
3 2
4 3
5 2
6 5
7 6
8 2
```

2.2 Kruskal

Kruskalov algoritem je prav tako požrešne narave. Začne z množico vozlišč in dodaja povezave od manjših proti večjim povezavam glede na uteži. Pravzaprav postopoma pretvarja gozd z več manjšimi drevesi v eno veliko drevo. Vsako povezavo (x, y) doda, če njena vključitev ne ustvari cikla. Povedano drugače, vozlišči x in y ne smeta pripadati istemu drevesu oz. povezani komponenti.

Vodi ta postopek res do optimalne rešitve? Tudi tu si lahko pomagamo s prerezno lastnostjo. Recimo, da smo že sestavili nek gozd in želimo dodati povezavo (x, y). Naj bo drevo z vozliščem x množica A, vsa ostala vozlišča pa množica B. Povezava (x, y) je globalno najcenejša nedodana povezava in zato tudi najcenejša povezava med množicama A in B. Torej jo lahko gotovo dodamo v vpeto drevo in pri tem ne bomo zgrešili optimalne rešitve.

Za začetek moramo povezave urediti po velikosti, kar zahteva $O(m \log m)$ časa. Nato pa obravnavamo po vrsti vseh m povezav in za vsako preverjamo, ali sta krajišči del iste povezane kom-

ponente. Povezano komponento lahko vsakič znova določimo z uporabo preiskovanja v širino ali globino, ki ima časovno zahtevnost $O(m)$. Časovna zahtevnost celega postopka bi bila $O(m \log m + mm) = O(m^2)$.

Lahko pa uporabimo podatkovno strukturo disjunktnih množic, ki predstavljajo povezane komponente. Posamezna vozlišča združujemo v povezane komponente, da dobimo na koncu eno samo komponento, ki je minimalno vpeto drevo. Operacije v strukturi disjunktnih množic so praktično konstantne in zanemerljive v primerjavi z začetnim urejanjem povezav. Časovna zahtevnost je $O(m \log m + m\alpha(n)) = O(m \log m) = O(m \log n)$.

[8]:
```
bool cmpW(VI e1, VI e2) { return e1[2] < e2[2]; }
```

[9]:
```
int Kruskal(int n, vector<VI> &edges, vector<PII> &mst) {
    sort(edges.begin(), edges.end(), cmpW);  // sort by weights
    DisjointSet ds(n);
    int cost=0;
    for (VI e : edges) {
        int a=e[0], b=e[1], w=e[2];
        if (ds.root(a)==ds.root(b)) continue;  // same component?
        ds.join(a,b);
        cost+=w;
        mst.push_back({a,b});
    }
    return cost;
}
```

[10]:
```
vector<PII> mst;
cout << Kruskal(n, edges, mst) << endl;
for (PII edge : mst) cout << edge.first << " " << edge.second << endl;
```

```
37
7 6
8 2
6 5
0 1
2 5
2 3
0 7
3 4
```

2.3 Steinerjevo drevo v grafu

V problemu minimalnega vpetega drevesa smo morali poiskati podmnožico povezav z najmanjšo vsoto, ki med seboj povezujejo vsa vozlišča grafa v obliki drevesa. Problem lahko posplošimo tako, da zahtevamo, da je med seboj povezana samo neka izbrana podmnožica vozlišč (ki jim rečemo terminali, njihovo število pa bomo označili s t), vključuje pa lahko tudi druga vozlišča

- $t = n$: Če so vsa vozlišča terminali, imamo opravka s problemom minimalnega vpetega drevesa.
- $t = 2$: Če moramo povezati samo dve vozlišči, imamo opravka s problemom najkrajše poti.

- V splošnem se temu problemu reče Steinerjevo drevo v grafu. Vozliščem, ki so del rešitve (drevesa), čeprav niso terminali, pa Steinerjeve točke.

Problem Steinerjevega drevesa spada med težke probleme, za katere ne poznamo algoritmov s polinomsko zahtevnostjo v odvisnosti od števila terminalov t. Soroden geometrijski problem Steinerjevega drevesa v ravnini, kjer želimo povezati t točk z ravnimi črtami, pri čemer lahko dodajamo vmesne točke/križišča, je prav tako težek.