# Osnovno urejanje

Urejanje z izbiranjem: Prostorska O(n), časovna O(n^2).

Urejanje z vztavljanjem: Prostorska O(n), časovna O(n^2)

Mehurčno urejanje:. Prostorska O(n), časovna O(n^2)

# Napredno urejanje

**Urejanje z zlivanjem:** Razdelimo na dva podseznama, ju rekurzivno uredimo in jih nato zlijemo v urejen seznam. Časovna O(nlogn), prostorska lahko tudi O(n)

**Hitro urejanje:** Pivot element, razdelimo seznam na manjše, enake, večje in rekurzivno uredimo. Časovna O(nlogn), lahko tudi O(n^2) - slabi izbiri pivota.

**Heapsort:** S pomočjo kopice, samo popaš.

**Urejanje s štetjem:** Primerno za nabore z majhnimi vrednostmi. Indeksi seznama so števila in vrednosti so količina.

**Radix sort(urejanje s koši):** Sortiraj stringe na prvo črko(vsaka črka svoj koš) in jih nato rekurzivno za drugo itd…Časovna zahtevnost O(n \* max dolzina besede)

# Abstraktni podatkovni tipi

**Polje:** operacije init(c),set(i,x),get(i). Urejen seznam.

**Sklad stack:** LIFO struktura. Operaciji push in pop.

**Seznam:** Hrani seznam elementov, vrstni red vstavljanja pa je odvisen od APT. Operacije add(x), remove(x), size(), traverse.

**Povezan seznam list:** Brisanje in dodajanje O(1), dostop O(n)

**Dinamično polje vector:** Omogoča tudi dajanje elementov na konec seznama. Časovna zahtevnost je O(n) za grajenje dolžine n.

**Vrsta queue, deque:** Podobno kot sklad samo, da je FIFO. Lahko s povezanim seznamom ali pa dinamičnim poljem. Implementacija je lahko s tabelo, pazimo, da jo manjšamo za ½ samo ko je kapaciteta pod ¼.

**Prednostna vrsta priority\_queue:** Elementi urejeni glede na prednost. Push in pop.

**Kopica(dvojiška):** Z njo implementiramo prednostno vrsto. Pravilo je, da so otroci vsi manjši oz. večji od svojih staršev. Predstavimo s tabelo, kjer ima i-to vozlišce otroka na 2i in 2i+1. Starša pa na i/2 navzdol. Element vstavimo na konec tabele in nato popravimo urejenost tako da menjamo s staršem. Brisanje najmanjšega tako, da ga zamenjamo z zadnjim in nato potopimo koren. Čas zahtevnost O(logn).

**Preskočni seznam:** Vstavljaš tako, da naključno generiraš višino in potem po nivojih se sprehajaš to prvega elementa, ki je večji od vstavljenega. O(logn) obe operaciji.

**Množica set:** Operacije insert(x), remove(x), contains(x).

**Slovar map:** Operacije insert(k,v), remove(k), get(k).

**Zgoščena tabela unordered\_map:** Zgoščevalna funkcija preslika vrednosti v indekse tabele. Če pride do trka se ponavadi doda v linked list na indeksu. Ko razmerje med št. elementov in velikostjo tabele preseže 1 se tabela poveča in se vsi elementi znova hashajo. Pričakovana časovna zahtevnost za iskanje je O(1).

# Drevesa

**Poizvedbe na območjih s statičnimi drevesi:** Uporabno za nekatere poizvedbe. Izgradiš drevo, ki ima v vsakem vozlišču izračunano vrednost za območje, ki ga predstavlja in njeni otroci so podobmočja.Obdelave so O(n), poizvedbe so O(logn). :+: Fiksna struktura, učinkovita za statične podatke.-: Neprilagodljivo za dinamične spremembe podatkov.

**AVL drevo:** Struktura isto kot dvojiško iskalno drevo(torej levo večji, desno manjši). V vsakem vozlišču izračunamo višino desnega poddrevesa - višina levega. Če ima A faktor uravnoteženosti 2 in je drevo Z višje od Y potem naredimo en rotateLeft. Če pa je Y pregloboko pa prvo naredimo rotateRight z B in Y nato pa še rotateLeft z A in Y. +:Uravnoteženo drevo, zagotavlja logaritemski čas operacij.-: Dodatni stroški pri vzdrževanju uravnoteženosti.

**Binary Search Tree (BSTree**):+: Preprosta implementacija, učinkovito iskanje.-: Neuravnoteženo -> neoptimalne zmogljivosti pri nekaterih operacijah.

**Rdeče-črno drevo**:+: Uravnoteženo drevo, manj stroga pravila kot AVL drevo.-: Lahko ima večjo globino kot AVL drevo.

**2-3 drevo**:+: Podpora večjemu številu otrok na vozlišče.-: Manj učinkovito v primerjavi s specializiranimi drevesi za določene primere.

**B drevo**:+: Podpora večjemu št. otrok na vozlišče, uporabljeno za indeksiranje v podatkovnih bazah.-: Kompl. implementacija v primerjavi s preprostimi drevesi.

**Lomljeno drevo** (Splay Tree):+: Poudarek na lokalnosti za izboljšano učinkovitost dostopa do "vročih" elementov.-: Morda ni najbolj primerno za enakomerno porazdeljene poizvedbe.

**Naključno uravnoteženo drevo**:+: Naključno uravnoteženo, dobra zmogljivost v povprečju.-: Težko predvidljivo vedenje v najslabšem primeru.

# Požrešni algoritmi

Tukaj vedno vzamemo prvo lokalno optimalno rešitev in tako zmanjšamo problem. Zaplete se pri dokazovanju pravilnosti:

Lahko dokažemo, da je pri vsakem koraku požrešna rešitev tako dobra kot optimalna. Lahko se optimalna in naša razlikujeta prvič na i-tem mestu in dokažemo, da če zamenjamo z našo se optimalna ne bo poslabšala.

# Grafi

**Topološko urejanje:** Smiselno samo v acikličnih usmerjenih grafih. Vodimo seznam z vhodnimi stopnjami in posebej sklad z vhodnimi stopnjami 0. Povrsti jih odstranjujemo in zraven updatamo seznam s stopnjami.

**Kritična pot:** Najdaljša usmerjena pot v uteženem acikličnem grafu. Računamo d(x) kar je najdaljša pot iz posameznega vozlišča.

Torej za vsako vozlišče je to maksimum razdalj naslednikov + utež. Torej imamo topološko urejen seznam in gremo v obratnem vrstnem redu od tistih brez naslednikov da tistih brez predhodnikov in hranimo vozlišče z maksimalnim d(x).

Slika, ki vsebuje besede besedilo, posnetek zaslona, pisava, številka

Opis je samodejno ustvarjen

# Najkrajše poti

**Neuteženi grafi:** Samo BFS naredis z dolzino poti

**Uteženi grafi:** nenegativnimi utežmi

**Dijkstrov algoritem:** Računa najkrajšo razdaljo od izhodiščnega vozlišča do vseh drugih. Vzdržujemo tabelo potencialnih razdalj od vozlišča A(izhodiščno) do okolice(torej vozlišč sosednjih že obravnavanim). Izberemo vozlišče z najmanjšo tako potencialno razdaljo in potem posodobimo njene sosede z min(trenutna poten., d(x) + utez). Zakaj je izbira najmanjše potencialne razdalje dobra: ker da bi obstajala krajša alternativna pot bi morali iz nekega drugega vozlišča se odpraviti po njej in bi bila potem to naša najkrajša razdalja. Zahtevnost je O(n^2+e)=O(n^2) za implementacijo z manualnim iskanjem najmanjše razdalje. Če iščemo s priority queue je potem O(nlogn+elogn)=O(elogn). Ta implementacija ima smisel samo če je graf redek(e =/= n^2). Algoritem lahko še izboljšamo s predalčkanjem, če so uteži majhne(tabelo z indeksi kot razdaljami). Zahtevnost je O(e + n\*max utez).

# Vpeta drevesa

**Disjunktne množice:** Operacije add,find,union. Find vrne koren množice. Združevanje delamo po velikosti, torej pridružimo manjšo množico k večji. Pri iskanju pa uporabljamo stiskanje poti torej, ko enkrat izvemo koren, vozlišče kar direktno na koren povežemo. Čas zaht. je skoraj konst. O(e\*alpha(n)) – O(mlog\*n).

**Minimalno vpeto drevo:** Min. prerezna povezava je vedno del vpetega drevesa. Katerokoli drugo povezavo med komponentami -> vedno slabša ali enaka rešitev.

**Primov algoritem:** Izbiramo minimalne prerezne povezave iz množice že obravnavanih vozlišč v množico še ne obravnavanih. Lahko pa pospešimo to tako, da gledamo imamo množico potencialnih razdalj do že obravnavanih in vzamemo najmanjšo. Potem pa samo posodobimo nove. To lahko naredimo isto kot pri Dijkstri s pq. Stare vrednosti v pq pa samo ignoriramo. Čas zahtevnost brez optimizacije je O(n^2+e)=O(n^2)(e povezav in iščemo minimalno) z pa je O(elogn)

**Kruskalov algoritem:** Začne z množico vozlišč in dodaja povezave od manjšim proti večjim. Zraven samo pazi, da ne dodaja povezav, ki so znotraj iste komponente. Za urejanje rabimo O(e log n), za vsako pa moramo gledati ali so krajišča znotraj iste komponente. Torej je O(e\*n). Uporaba disjunksnte množice pa je O(e log e + ealpha(n))=O(elogn).

# Deli in vladaj

**Krovni izrek:** Problem velikost n razbijemo na b problemov velikost n/b in obravnavamo jih a. Torej imamo rekurzivno formulo T(n)=aT(n/b)+f(n) kjer je f(n) dodatno delo, ki ga opravimo na vsakem nivoju. Število nivojev je log\_b(n) => listov n^(log\_b(a)). Označimo c = log\_b(a).

# Dinamično programiranje

To je isto kot deli in vladaj samo, da se podproblemi ponavljajo. Potem jih lahko rešimo top-down z memoizacijo ali pa bottom-up z račun posameznih podprobl.

# Računska geometrija

***Razdalje***: **med točkama** - pitagora, **točka in premica** - projekcija točke na premico (skalarni prod.) in potem izračunamo razdaljo med točko in projekcijo, **točka in daljica** - izračunamo razdaljo z nosilko in potem gledamo ali je projekcija izven daljice, **dve daljici** - preizkusimo vsa 4 krajišča.

***Presečišča****:* **točka in premica** - vektorski produkt (A-p0) in smernega vektorja mora biti 0, **točka in daljica** - preverimo za nosilko potem pa še za pravokotnik krajišč, **dve premici** - rešimo enačbo p0 + t\*s1= p1+k\*s2 za obe koordinati, **premica in daljica** - presečišče nosilk in pogledamo če je znotraj daljice, **dve daljici** - prvo pogledamo a se nosilki sekata, potem pa gledamo ali sta krajišči AB na drugi strani nosilke CD. Orientacijo gledamo z vektorskim produktom.

**Površina večkotnika:** Razdelimo na trikotnike in računamo po formuli p = ½ |AB x AC|. Za nekonveksne večkotnike pa uporabimo formulo.

**Vsebovanost točke:** Za trikotnik samo gledamo ali je ista orientacija za vse daljice torej vektorski produkti ABxAT, BCxBT in CAxCT morajo imeti isti predznak. To tudi deluje za konveksni večkotnik. Za nekonveksne pa uporabljamo raycasting. Torej iz poljubne smeri streljamo žarek v točko T in štejemo kolikokrat smo prečkali daljice. Za liho št. prečkanj je točka znotraj. Žarek pa lahko seka oglišča -> moramo pogledati kje sta sosednji oglišči, da vidimo ali je žarek šel v notranjost.

**Konveksna ovojnica:** to je konveksni večkotnik, ki ima kot oglišča točke množice in vsebuje vse preostale.

**Identifikacija stranic:** Za vsak par točk lahko preverimo ali so vse ostale točke na isti strani daljice, torej bo ta daljica del ovojnice. Zahtevnost O(n^3).

**Zavijanje darila:** Začnemo v krajni levi točki in potem se vrtimo v smeri urinega kazalca in najdemo prvo sosedo. Da preverjamo, da je to res prva lahko gledamo vektorski produkt AB x AC kjer je B trenutna kandidatka za prvo v smeri urinega kazalca, C pa je neka poljubna točka. Zahtev. je O(n^2) oz. O(velik. ovojnice \* n).

**Grahamov pregled:** Izberemo ekstremno točko T in uredimo preostale točke po kotu glede na T. Potem jih po vrsti dodajamo v ovojnico in hkrati popravljamo konveksnost. Torej če je A predzadnja dodana točka, B zadnja dodana točka in C točka, ki jo obravnavamo, dokler je ABxAC < 0 mečemo točko B iz seznama. Zaradi urejanje po kotih je O(nlogn).

**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;}

**bool** cmpSecond(pair**<int**,**int>** a, pair**<int**,**int>** b) {

**return** a.second **<** b.second;}

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;}

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;}

**bool** cmpRatio(pair<int,int> a, **pair**<int,int> b) {**return** a.first\*b.second< b.first\*a.second;}

**int** trak(vector<pair<int,int>> d) {

sort(d.begin(), d.end(), cmpRatio);

**return** score(d);}

**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;}

**int** zamuda(VII o) {

sort(o.begin(),o.end(),cmpSecond);

**return** late(o);}

**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;} } }

**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); } }

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;}

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;}

»dolzina« = d[start] "kriticna pot = " 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**; } } }

**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* } } } }

**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});

//predalčkanje: **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]) {////

**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}); } } } }

**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; *// 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); *//smaller x*

parent[x] **=** y; *// attach to larger root*

size[y] **+=** size[x]; } };

**int** Prim(**int** n, vector**<**VII**>** **&**adj, vector**<**PII**>** **&**mst) {

vector**<int>** dist(n,-1); *//distance from 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;}

**int** Kruskal(**int** n, vector**<**VI**>** **&**edges, vector**<**PII**>** **&**mst) {

sort(edges.begin(), edges.end(), cmpW); *// sort by weights {* ***return*** *e1[2]* ***<*** *e2[2]; }*

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;}

**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;

**while** (lo**+**1**<**hi) {

**int** lim**=**(lo**+**hi)**/**2; **int** sum**=**0, chunks**=**1;

**for** (**int** x **:** a) {

**if** (sum**+**x**<=**lim) sum**+=**x; *// isti*

**else** { chunks**++**; sum**=**x; } *// nov*

} **if** (chunks**<=**k) hi**=**lim;

**else** lo**=**lim;

} **return** hi; }

**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;}

vector**<int>** c **=** {0,2,5,6,9,15,16,17,20}; //sekanje palice

**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]); }}

**int** n **=** 4; **int** nosilnost **=** 40; //nahrbtnik 0-1

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*

**if** (teza[i]**<=**x) { *// poskusimo uporabiti*

f[i][x] **=** max(f[i][x], vrednost[i]**+**f[i**+**1][x**-**teza[i]]); } }}

cout **<<** f[0][nosilnost] **<<** endl;

PII OPERATOR\_SUBTRACT(PII a, PII b) { //točka v liku

**return** {a.first**-**b.first, a.second**-**b.second};}

**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]; *// v stranice*

PII v **=** t**-**poly[i]; *// vektor do tocke:*

**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;}