Osnovno urejanje

Urejanje z izbiranjem: Iz seznama izbiranje min element, odstranili in postavili na 1. mesto urejenega seznama, ki ga tako gradimo -> dokler nam ne zmanjka vhodnega seznama. Urejanje na mestu – swap 2 elementov, Space O(n), time O(n^2).

Urejanje z vstavljanjem: Po vrsti bomo jemali elemente iz vhodnega zaporedja in vsakega posebej vstavili v novo nastajajoče urejeno zaporedje. Na vsakem koraku imamo urejeno zaporedje na prvih i-1 mestih, v preostanku pa je še neurejeno. 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 –> dokler nam ne zmanjka vhodnega seznama. Space O(n), time O(n^2) – best O(n)

Mehurčno urejanje: samo z zamenjavami sosednjih elementov. Pare bomo pregledovali po vrsti. Ko pridemo do konca seznama, pa se bomo vrnili nazaj na začetek. Prehod brez zamenjav -> urejeno!. Space O(n), time O(n^2) – best O(n)

Napredno urejanje

Urejanje z zlivanjem: Razdelimo na dva podseznama, ju rekurzivno uredimo in jih nato zlijemo v urejen seznam. time O(nlogn), space O(n) – >O(nlogn)

Hitro urejanje: Pivot element, razdelimo seznam na manjše, enake, večje in rekur. uredimo. time O(nlogn) -> O(n^2) – slab izbor pivota, space O(n) -> O(nlogn)

Heapsort: S pomočjo kopice – najprej jo narediš, nato pa popaš vsak element.

Urejanje s štetjem: Primerno za nabore z majhnimi vrednostmi. Indeksi seznama so števila in vrednosti so količina. Time O(n), space odvisno od max elementa

Bucket sort(urejanje s koši): razdeli elemente v koše glede na vrednost. Npr. 2 koša: [min,med], [med + 1, max]

Radix sort(korensko): Sortiraj stringe na prvo črko(vsaka črka svoj koš) in jih nato rekurzivno za drugo itd…Time O(nd), space O(nda), d=max besede, a=size(abeceda)

Bisekcija: lower_bound in upper_bound

Abstraktni podatkovni tipi

Polje: operacije init(c),set(i,x),get(i). navaden array

Sklad stack: LIFO struktura. Operaciji push in pop.

Seznam-list: 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: Vsakič, ko dosežemo kapaciteto, alociramo novo, dvakrat večjo tabelo - Resizeable array. Časovna zahtevnost je O(n) za grajenje dolžine n.

Vrsta queue, deque: Podobno kot sklad samo, da je FIFO. Lahko s povezanim seznamom (kazalec na začetek in konec) 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. Implementacija je možna z:

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

Preskočni seznam: Vstavljaš tako, da naključno generiraš višino in potem po nivojih se sprehajaš do prvega elementa >= vstavljenega-> linked list. O(logn) obe operaciji.

Množica set: Operacije insert(x), remove(x), contains(x). time O(logn) – skip list

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

Zgoščena tabela unordered_map: Zgoščevalna (hash) funkcija preslika vrednosti v indekse tabele – h(x)=x modH. Če pride do trka se ponavadi doda v linked list na indeksu. (a=n/H > 1)? -> se tabela poveča in se vsi elementi znova hashajo. time O(1+a), če je a konst. O(1) –> worse O(n)

Drevesa

Premi (pre-order) obhod obišče vozlišče -> rekurzivno po vrsti vsa poddrevesa.

Vmesni (in-order) obhod obišče levo poddrevo -> vozlišče -> desno poddrevo (v primeru dvojiškega drevesa). Če gre za iskalno drevo, nam vmesni obhod vrne urejeno zaporedje.

Obratni (post-order) obhod obišče rekurzivno vsa poddrevesa in šele nato vozlišče.

Nivojski (level-order) obhod obišče vozlišča po nivojih, koren-> otroke -> otroci itd.

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), višini levega in desnega se razlikujeta za max 1. 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, O(logn).-: 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, vsi listi na isti globini, iskanje ključev znotraj vozlišča z bisekcijo. -: Kompl. implementacija v primerjavi s preprostimi drevesi.

Lomljeno drevo (Splay Tree):+: Poudarek na lokalnosti za izboljšano učinkovitost dostopa do pogosto uporabljenih elementov, dostopane elemente pomika h korenu s pametno uporabo dvojnih rotacij.-: Morda ni najbolj primerno za enakomerno porazdeljene poizvedbe.

Naključno uravnoteženo drevo:+: Naključno uravnoteženo, dobra zmogljivost v povprečju, uporablja random prioritete volišč, iskalno drevo urejeno v kopico glede na prioriteto elem. O(logn).-: 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. **Izbira aktivnosti:** 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. 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. Prišli smo torej do protislovja in zato ne obstaja rešitev, ki bi bila boljša od požrešne. **Minimizacija zamude:** earliest deadline. Naj bodo opravila urejena naraščajoče po rokih, torej d1<=d2…<=dn. 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 (dj>di); sicer bi bila ta rešitev enaka požrešni. Nova zamuda (z') vseh drugih opravil razen i in j se 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-dj=fi-dj<=fi-di=zi (končata se ob enakem času; j ima manjši rok). Vemo torej, z'k=zk (vsak k, ki ni i ali j), z'i<=zi. Iz tega sledi, da je Z'=max z'k<= max zk=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 obstaja boljša rešitev. **Datoteke na traku:** n datotek (s velikost, f pogostost dostopa). Obravnavajmo sedaj splošen primer, kjer opazujemo možna vrstna reda dveh sosednjih datotek na traku. Ceni dostopa sta xfi+(x+si)fj in xfj+(x+sj)fi, 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 si/fi. Torej jih lahko v rešitev požrešno zložimo po vrsti od tistih z nižjim proti tistim z višjim razmerjem. si/fi <=sj/fj

Grafi

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.

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.

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.

	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	$O(e)$	$O(n)$	$O(1)$

Preiskovanje v širino (BFS) 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. -> računanje najkrajših poti.

Preiskovanje v globino (DFS) 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. +: Jedrnatost.
Time O(n^2), vendar smo lahko bolj natančni z O(e), ker bomo vsako povezavo obravnavali največ dvakrat (enkrat iz vsakega krajišča). Space O(n^2).

Topološko urejanje: 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**.** Smiselno samo v acikličnih usmerjenih grafih. Vodenje seznama vozlišč z vhodno stopnjo 0. Vsakič ko odstranimo vozlišče in njegove izhodne povezave, dodamo v seznam morebitna novo nastala začetna vozlišča. Time O(n+e)->O(e).

Kritična pot: Najdaljša usmerjena pot v uteženem acikličnem grafu. Vozlišča naprej topološko uredimo v linearnem času. Nato pa lahko računamo najdaljše poti d(x), ki se začnejo v posameznem vozlišču x, v obratnem topološkem vrstnem redu. Če vozlišče nima naslednikov, je d(x)=0, sicer pa: $d(x) = \max_{y:\, x<y \wedge (x,y)\in E} (w(x,y) + d(y))$.
Torej za vsako vozlišče je to maksimum razdalj naslednikov + utež. 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.

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(neizračunana vozlišča in so iz že izračunanih dosegljiva po eni povezavi). Izberemo vozlišče z najmanjšo tako potencialno razdaljo in potem posodobimo potencialne razdalje njenih sosedov z min(trenutna potencialna, d(x) + utez). Potencialna razdalja: -1=neobiskano, -2=že izračunano. 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), če je e=n^2 -> O(n^2logn). Algoritem lahko še izboljšamo s predalčkanjem (bucket queue), če so uteži majhne(tabelo z indeksi kot razdaljami). Zahtevnost je O(e + n*max utez).
Najširša pot: iščemo pot od A do B, maksimiziramo min utež na poti. Vsoto smo torej zamenjali z min, minimizacijo pa z maksimizacijo. Poti do vozlišče bomo računali od širših proti ožjim. Najširša pot (inf) vodi do začetnega vozlišča.

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: je tisto vpeto drevo, ki ima najmanjšo vsoto uteži povezav. Razbitju vozlišč grafa na dve disjunktni množici pravimo **prerez grafa**. Povezavam s krajišči v različnih delih razbitja pa **prerezne povezave**. Min. prerezna povezava je vedno del vpetega drevesa. Katerokoli drugo povezavo med komponentami -> vedno slabša ali enaka rešitev.
Primov algoritem: požrešen algoritem, ki gradi minimalno vpeto drevo s širjenjem od izhodiščenega vozlišča navzven proti sosedom. 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 glede na uteži. Zraven samo pazi, da ne dodaja povezav, ki ustvarijo cikla. Za urejanje rabimo O(eloge), za vsako pa moramo gledati ali so krajišča znotraj iste komponente. Torej je O(eloge +en) = O(en). 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).
Enakomerno razbitje seznama: imamo seznam n števil z vsoto V, ki ga želimo razbiti na k strnjenih podseznamov. 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. Za dvojiško iskanje meje v bomo potrebovali O(logV) korakov. Za določanje sprejemljivosti posamezne meje pa O(n). To je skupaj O(nlogV).

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 računanjem posameznih podproblemov.
Žabji skoki: n skal na x1 < … < xn koordinatah. Žabec lahko skoči min a in max b enot. Najmanj koliko skokov potrebuje? 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. Če razmišljamo rekurzivno, se bo žabec v prvem skoku premaknil na neko skalo xi, ki je oddaljena med a in b od skale x1. 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 xi 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<xj-xi<b](1+f(j)). 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).
Rezanje palice: palica dolžine n, želimo razrezati na manjše kose in jih prodati posamično za max ceno. 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<=n (s čimer zaslužimo cx) 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<=n]f(n-x)+cx. Time O(n^2), space O(n). Rekonstrukcija rešitve: 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 bi lahko hranili x, pri katerem funkcija f(n) doseže svoj max.
Pot v mreži: labirint h * w. Koliko poti obstaja iz levo zgoraj v desno spodaj? Rekurzivno bi problem reševali tako, da bi se s trenutne celice poskusili premakniti desno in navzdol (če sta oba premika sploh možna – če ni je 0) in sešteli možne poti do cilja iz nove lokacije (sosednje celice). F(i,j)=f(i+1,j)+f(i,j+1); 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). Prostorsko bi lahko izboljšali O(w), ker pri računanju f(i,*) potrebujemo le f(i,*+1) in f(i+1,*).
Najdaljše skupno podzaporedje: niz S in T niz LCS(S,T). LCS(i,j)=max{1+LCS(i+1,j+1) //Si=Ti, LCS(i+1,j), LCS(i,j+1)}. LCS(n,*)=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. Time in space O(nm).
Nahrbtnik: n predmetov, teže ti in vrednost vi. Želimo maksimizirati vrednost, tako da skupna teža ne bo presegla nosilnosti T. V rekurzivni rešitvi bi se lahko za vsak predmet odločili, ali ga bomo vzeli ali ne. Če ga vzamemo, imamo za preostale predmeta na voljo nekoliko manjšo nosilnost. F(i,x)=max{f(i+1,x) // ne uporabimo, f(i+1, x-ti) + vi // uporabimo i-ti predmet}. Časovna zahtevnost je O(nT). Če nimamo meje za T, vemo, da teža predmetov ne bo presegla sum(ti). Ta rešitev z dinamičnim programiranjem izkorišča majhne celoštevilske teže predmetov in nosilnost nahrbtnika. 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.

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 (min je potem razdalja do A ali B, če ne pa projekcija), **dve daljici** - preizkusimo vsa 4 krajišča (krajišče do presečišča).
Presečišča: **točka in premica** - vektorski produkt vektorja p0A in smernega vektorja premice p 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 se točka pri obhodu nahaja na isti strani -> 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. h=št. točk na konv. ovojnici. Več algoritmov:
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 (ekstremna točka T) 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. Time O(hn) -> worse O(n^2).
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).