AIPS TEORIJA

KAJ JE ALGORITEM?
Algoritem je (neformalno) katerakoli dobro definirala procedura, ki sprejme nek vhod in z izvajanjem računskih korakov pridobi nek izhod - želeno razmerje (rešitev problema).
Je JASEN in NEDVOUMEN mehanični postopek. Nanj lahko gledamo kot na orodje za reševanje specifičnih računskih problemov.

KAJ JE NALOGA?
Naloga ni npr. urejanje seznama števil -> naloga JE SEZNAM, ki ga urejamo, npr. {1,3,2}.

KAJ JE REŠITEV?
Rešitev je logični izhod, ki ga vrne algoritem kot odgovor na nek vhod. Npr. pri problemu iskanja zaporedja dolžine 3 za vhod {1,3,2}, bi rešitve bile:
123,132,231,213,321,312

KAJ JE RAČUNSKI PROBLEM?
Primer bi bil iskanje zaporedja dolžine 3 iz množice {1,3,2}.

Problem predstavlja željeno razmerje med vhodom/izhodom, oziroma med NALOGAMI in njihovimi REŠITVAMI. Je matematični objekt, ki predstavlja zbirko vprašanj katere računalnik mogoče lahko reši. Računski problem ni samo računanje števil, temveč tudi urejanje, ...

CALCULATE – računski problem
COMPUTE – računalniški problem

Poznamo več vrst računskih problemov:

- **ODLOČITVENI** – odgovor vsake instance je lahko DA/NE; npr. if stavki. Problem ponavadi predstavljen z množico instanc, kjer je odgovor pozitiven. Problem iskanja praštevil:
  L = {2,3,5,7,11, ...}
  PROBLEM: X = Y?
  NALOGA: X,Y element N
  REŠITEV: {T,F}

- **PREŠTEVALNI** – rešitev je neko število; preštevalni problem prosi za število rešitev nekega problema.

- **NAŠTEVALNI** – rešitev je seznam; kot rezultat dobimo možne izide.

- **ISKALNI** – rešitev je iskani objekt; iskalni problem lahko predstavimo z relacijo, ki ima vse pare rešitve...? Iskanje problema je lahko tudi računska operacija.

- **OPTIMIZACIJSKI** – iščemo najboljšo rešitev iz množice vseh možnih rešitev. Te probleme dalje ločimo na kriterijske (vrednost rešitve) in konstrukcijske (rešitev želimo konstruirati). Če iščemo najkrajšo rešitev je to minimizacijski problem, če iščemo najdaljšo pa maksimizacijski.

Kako opisujemo, snujemo algoritme?

- Z naravnim jezikom
- S psevdokodo
- S programskim jezikom
- Z diagramom poteka

Lahko jih implementiramo kot navadno LINEARNO ZAPOREDJE UKAZOV ali pa jih delamo REKURZIVNO. Vmes ponavadi pride tudi do razhroščevanja kode (izpis spremenljivk, trace, breakpoints). Pri profiliranju in instrumentaciji kode ugotovimo koliko časa in pomnilnika bodo potrebovali posamezni deli programa.

NARAVNI JEZIK
Je primeren za opis ideje, problem se pojavlja v jasnosti – lahko je nejasen in dvoumen.

PSEVDOKODA
Je nekoliko prenosljiva, imamo poljubno stopnjo natančnosti, uporabljamo lahko matematične formule. Vmešamo lahko programske konstrukte, zato je to lažje prevesti v dejansko kodo.

PROGRAMSKI JEZIK
Predstavlja realnost; žal imamo preveliko izbiro jezikov. Koda je lahko slabše pregledna? Algoritem lahko takoj poženemo.

DIAGRAM POTEKA
Poda grafični, globalni prikaz algoritma, manjkajo mu podrobnosti.

Implementacija algoritmov

LINEARNO
Branje vhoda, izpis izhoda; aritmetične in logične operacije, nizi in tabele, odločitveni (if) stavki, iteracije in zanke.

REKURZIVNO
Rekurzija je funkcija ki kliče sama sebe; pri vračanju se vrne v prejšnje stanje. Je zelo elegantna rešitev za veliko težkih problemov, paziti je treba predvsem na ustavitvene pogoje. Potrebuje svoj sklad, na katerega shranjuje trenutne instance; ta sklad potem koristi ob vračanju.

Linearna rekurzija – vsebuje samo en klic nase
Repna rekurzija – klic nase je zadnja operacija v funkciji
Binarna rekurzija – vsebuje več klicev nase

IZVEDBA ALGORITMA
Zagon programa na računalniku, izvajanje eksperimentov, znanstvena metoda – hipoteza, eksperimentalno ovrednotenje algoritma.

IZVAJANJE PROGRAMA
Prevajanje izvorne kode v strojno kodo; interpretiranje izvorne kode - odvisno od uporabljene tehnologije. Načeloma programski jeziki prevedejo, skriptni pa interpretirajo.

SLED ALGORITMA
Med izvajanjem programa izpisujemo vrednosti spremenljivk, skladov, ... To lahko simuliramo na papir ali pa kar z računalnikom.

Metode razvoja

Predpogoj je dobro razumevanje problema, cilj je opis algoritma.
Paziti moramo da je algoritem IMPLEMENTABILEN (zapisljiv v programsko kodo), da je UČINKOVIT, PREPROST in predvsem PRAVILEN. Slediti moramo naslednjim korakom:

- Definicija problema,
- Izdelava modela
- Specifikacija algoritma
- Design algoritma,
- Preverjanje pravilnosti
- Analiza algoritma
- Implementacija algoritma
- Testiranje programa
- Dokumentacija

Poznamo naslednje metode snovanja algoritmov:

- GROBA SILA — brute force in izčrpno preiskovanje — exhaustive search.
- SESTOPANJE — backtracking
- RAZVEJI IN OMEJI — branch & mound
- POŽREŠNO — greedy; med izvajanjem ubiramo le najboljše poti – zna biti slabo! Zakaj?
- DELI IN VLADAJ — divide & conquer; deljenje velike naloge na manjše dele
- ZMANŠAJ IN VLADAJ — reduce & conquer
- PRETVORI IN VLADAJ — transform & conquer
- DINAMIČNO PROGRAMIRANJE — dynamic programming; drobljenje na manjše probleme, pomnjenje podrešitev
- LINEARNO PROGRAMIRANJE
- REKURZIVNO PROGRAMIRANJE; sestavljanje podrešitev

Pravilnost algoritmov

Najbolj nas zanima, ali algoritem res počne to, kar zanj mislimo da dela ter kakšna je njegova zahtevnost!
Pravilnost algoritmov lahko dokazujemo z intuitivnim razumevanjem, testi, formalnimi dokazi pravilnosti in avtomatskim formalnim preverjanjem.

DELNA PRAVILNOST
Algoritem je delno pravilen, če zanj obstaja možnost, da se zacikla. Če je problem prekompleksen, vrne aproksimacijo, drugače pa vrne pravilno rešitev kadar jo vrne.

POPOLNA PRAVILNOST
Za vse vhode se ustavi (se ne zacikla) in vrne pravilno rešitev!
TESTNI PRIMERI

Pravilnost pokažemo z nekimi testnimi primeri, za katere vemo rezultat. Nemogoče je preizkusiti vse možne vhode. Lažje dokažemo nepravilnost kot pravilnost. Veliko robnih primerov za dober nabor testnih primerov.

FORMALNI DOKAZ PRAVILNOSTI
Dokaz osnovan predvsem na matematiki/statistiki. Dokazi lahko postanejo ekstremno dolgi pri kompleksnih algoritmih. **Vsi resnejši algorimi** morajo imeti ta dokaz.

- INDUKCIJA (metoda za dokazovanje neke trditve);
    - OSNOVNI PRIMER: dokažemo pravilnost za majhen primer
    - INDUKTIVNA PREDPOSTAVKA: neka trditev velja za n
    - INDUKTIVNI KORAK: če velja za n, velja tudi za n+1

AVTOMATSKO FORMALNO PREVERJANJE
Temelji na klasičnem formalnem dokazovanju; imamo posebne programske jezike, ki med izvajanjem poskušajo preveriti pravilnost algoritma.

ZANČNA INVARIANTA
Postopek za preverjanje pravilnosti zank. Podobno matematični indukciji, le da se izvaja do terminacije zanke. Ima naslednje korake:

- INICIALIZACIJA
- VZDRŽEVANJE
- TERMINACIJA

Tega baje ne bo na izpitu😊

Model računanja != računski model

Model računanja je neka množica dovoljenih operacij. Vsaka operacija ima neko ceno (ene) izvedbe, te pa so lahko različne. Potrebujemo dovolj preprost model računanja, da je uporaben. Nekateri modeli:

- Turingov stroj
- RAM, PRAM, RASP
- Stroj s števcem, kazalcem
- ...

Zahtevnost algoritmov

Analiza algoritmov napoveduje resource, ki jih bo algoritem potreboval. Večino časa nas zanima procesorski čas. Pred analizo potrebujemo različne modele (model virov tehnologije, model implementacijske tehnologije)... kar je pa zamudno, ker dobimo dolge klobase enačb. Zato stvar poenostavimo z oceno – ASIMPTOTSKO NOTACIJO.

Zahtevnost je odvisna od velikosti naloge – vhoda. Za vse možne naloge pa govorimo o BEST, AVERAGE in WORST case scenarijih. Ponavadi nas najbolj zanima WORST CASE scenario, ker imamo potem garancijo, da algoritem ne bo trajal dalj časa kot v najslabšem primeru, da se izvede.

ASIMPTOTSKA NOTACIJA
Zanima nas predvsem, kakšnega **reda** je naš algoritem. Zanemarimo konstante in obdržimo člen, ki raste najhitreje. Red rasti nam poda preprosto karakterizacijo o učinkovitosti algoritma.

Pri *dovolj velikih* vhodih je pomemben le red rasti (najhitrejši del). Za male vhode je pa ta ocena veljavna le od neke točke dalje, zato to ni najbolj dobro.

$T(n) = 4/3n^3 + 2\sqrt{n}\lg n - 16\lg n$ → $T(n)$ je reda n^3

- O NOTACIJA: zgornja asimptotična meja; čas ne bo naraščal hitreje kot meja, ki jo dodeli O. To preprosto vidimo že z opazovanjem kode; m vgnezdenih zank = $O(n^m)$
- Ω NOTACIJA: spodnja meja; čas ne bo naraščal počasneje kot meja, ki jo določa Ω - za kakršenkoli vhod pri dovolj velikem n algoritem ne more trajati manj časa.
- Θ NOTACIJA: tesna meja; čas ne bo naraščal ne hitreje ne počasneje kot meja, ki jo dodeljuje Θ.
- o NOTACIJA (little-oh): zgornja netesna meja; lahko je precej nad dejansko zahtevnostjo algoritma, ampak še vedno zgornja meja -> primer: zahtevnost algoritma n2 , zgornja netesna meja je n3 ; to pomeni, da bo zadeva definitivno trajala manj kot n3
- ω NOTACIJA: spodnja netesna meja
- NOTACIJA: LIM = 1; kot Θ, le s konstanto?

DOLOČANJE ZAHTEVNOSTI

DOLOČANJE ASIMPTOTIČNE ZAHTEVNOSTI Z LIMITAMI

LIMITA	**0**	**0 ≤ LIM ≤ n**	**0 < LIM < ∞**	**0 ≤ LIM ≤ ∞**	**∞**	**1**
MEJA	Zgornja netesna meja O	Zgornja meja - lahko tesna O	Tesna meja Θ	Spodnja meja Ω	Netesna spodnja meja Ω	Tilda ~

Pri računanju limit za določanje meje se pogosto poslužujemo L'Hopitalovega pravila; če za neko limito velja:

$$\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0 \text{ ali } \infty$$

Potem odvajamo zgornji in spodnji del naše limite ->

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)}$$

(V primeru 0/0, 0 * ∞, ∞/∞, ∞-∞, 1^∞, ∞^0 ipd.)

Abstraktni podatkovni tipi

To so matematični modeli podatkovne strukture, ki opisujejo podatkovni tip z množico vrednosti, operacij in predstavitvijo podatka. Predstavijo tudi obnašanje modela.

MNOŽICA – SET
Matematično gledano je to končna množica, ki vsebuje enolične elemente. Ponavadi nima fiksne dolžine. Za razliko pridobivanja elementov večine ostalih zbirk, množica preverja vsebovanost elementa.

VREČA – BAG
Razširitev množice; dovoljuje ponavljanje elementov. Ne dovoljuje brisanja. Njen namen je sranjevanje elementov skozi katere lahko iteriramo.

SKLAD – STACK
Kot kopica, na vrhu katere lahko dodajamo in jemljemo elemente. Poznamo operacije push in pop! Pojavlja se pri rekurziji, v brskalnikih → ko gremo s strani na stran se dodajajo na stack, ko gremo back se poppajo.

VRSTA – QUEUE
Deluje po principu FIFO. Iz začetka jemljemo elemente, dodajamo jih pa na konec. Primer: vrsta strank na blagajni.

VRSTA Z DVEMA KONCEMA – DEQUEUE
Podobno vrsti, s to razliko da lahko jemljemo in dodajamo elemente s kateregakoli konca. Primer: vrsta strank na blagajni, ki jo ljudje lahko na koncu zapustijo, se ji pridružijo, opravijo svoje ali pa se nekdo vrine na začetek.

VRSTA S PREDNOSTJO – PRIORITY QUEUE
Elemente odstranjujemo z začetka, pri dodajanju elementov pa lahko katerega vrinemo, odvisno od prioritete elementa (ponavadi vrednost elementa samega). V glavnem ven jemljemo elemente z največjo/najmanjšo prioriteto.

ZAPOREDJE – SEQUENCE
Ima naključni dostop! Iz njega lahko dobimo katerikoli element, isto velja za vstavljanje in brisanje.
SLOVAR – DICTIONARY
Namenjeni so hranjenju parov ključ/vrednost. Podobno navadnemu seznamu, kjer element referenciramo po indeksu, le da je indeks tukaj poljuben ključ. Elementi niso urejeni.

POLJE
To je podatkovna struktura, ki ima vnaprej točno določeno kapaciteto. Ob kreiranju se v pomnilniku zasede nek prostor, kamor se nato podatki pripisujejo. Če želimo večjo tabelo moramo narediti novo in elemente prepisati vanjo (current_size * 2). Izkoriščenost je potem vprašljiva. Praznjenje polje zmanjša za 1/3.

KAPACITETA – koliko elementov je lahko v polju
VELIKOST – koliko elementov dejansko vsebuje polje
IZKORIŠČENOST – velikost/kapaciteta?

STATIČNO POLJE
Ne more spreminjati velikosti. Elemente vstavljamo od desne proti levi in jih zamikamo za ena v desno. Brisanje in vstavljanje imata zahtevnost O(1), premiki pa O(n).

DINAMIČNO POLJE
Dinamično lahko dodeljujemo in odvzemamo prostor. Operaciji add in delete imata O(n) zahtevnost. Operacija resize naredi enkrat večjo kapaciteto, zmanjša se pa samo za 1/3.
AMORTIZIRANA ZAHTEVNOST – več operacij združimo v eno in dobimo boljše rezultate...?

Povezani seznami

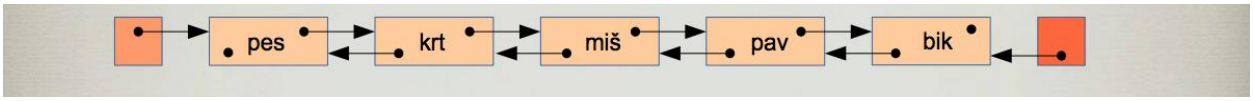
Temeljijo na kazalčni podatkovni strukturi. Kazalec pove, kje se nek objekt nahaja. Vrednost kazalca je naslov. Porabijo malo več prostora zaradi kazalcev. To je dinamična podatkovna struktura (spremenljiva velikost).

Kazalec je bolj splošen od reference in z njim lahko tudi računamo, kazalec na kazalec... Kaže na lokacijo v pomnilniku!
Referenca je bolj preprosta in nima kaj dosti operacij. Referenca kaže na dejansko vrednost?

VOZLIŠČE vsebuje en element (podatek) in kazalec. Vozlišča so med seboj povezana s kazalci. Lahko jih je več, odvisno od strukture.

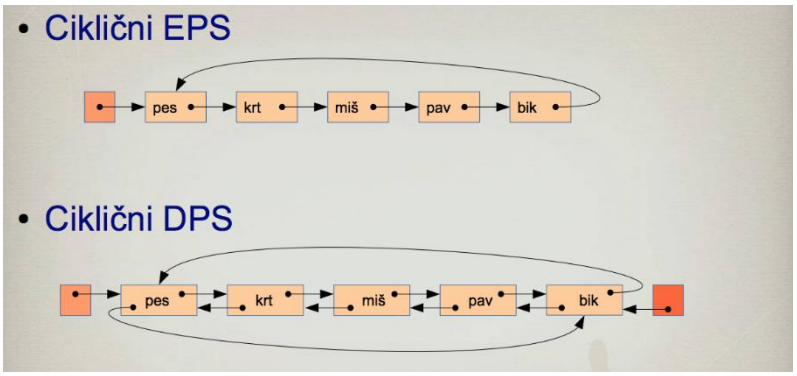
ENOJNO POVEZANI SEZNAM
Povezave le v eno smer; prejšnji element povezan na naslednjega. Sprehajanje po tem seznamu je zamudno, ker moramo čez vse prejšnje elemente. Pri odstranitvi moramo prevezati kazalec. Pri dodajanju novih objektov je zadeva podobna.
Tale seznam lahko uporabljamo tudi kot sklad. Če mu dodamo atribut 'last' dobimo vrsto.

DVOJNO POVEZANI SEZNAM
Vsak element ima povezavo na naslednika in predhodnika. Tako se hitreje pomikamo po elementih; z miši nazaj na krta → en korak, namesto da bi šli spet od začetka.



Gre za opcijsko sproščanje pomnilnika in prevezovanje kazalcev...?

CIKLIČNO POVEZANI SEZNAM
Takole povezan seznam je CIRCULAR, drugače pa je OPEN ali LINEAR.

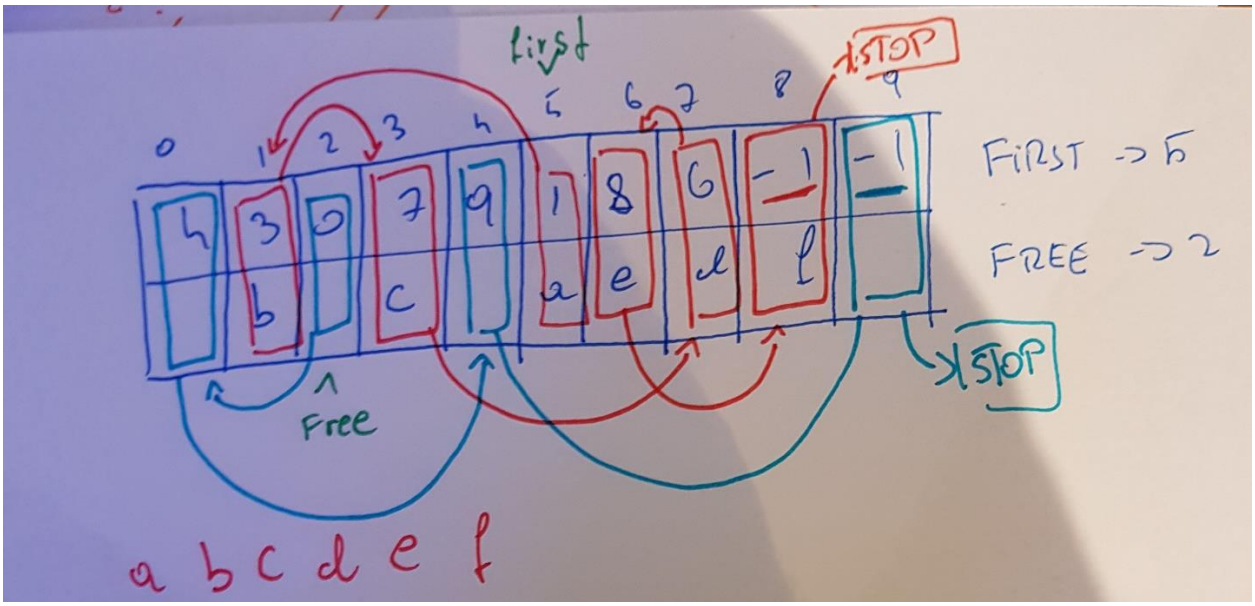


Paziti moramo, da v tem seznamu ne pride do kakšnega null pointer exceptiona; zato dodamo čuvaja! Je kot nekakšen dummy data na začetku, da ko npr. poppamo prazen list ne dobimo napake.

Te sezname lahko še kako drugače prikažemo; npr. s polji.

SEZNAM V POLJU

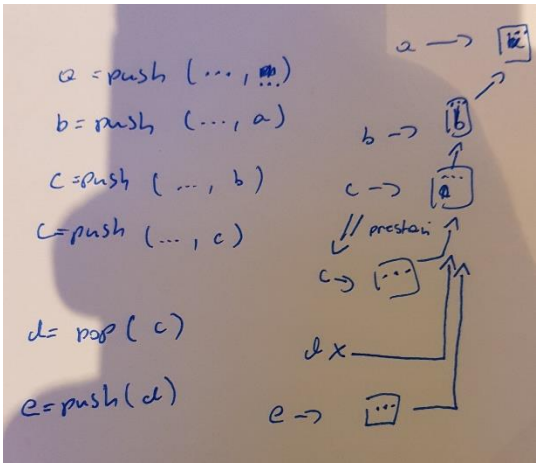
Polje next ima indekse naslednji elementov, polje item pa dejanske elemente. To pomeni hitrejše dostopanje do elementov, omejitev je statičnost polj (→ združevanje polj?).



First kaže na prvo celico. Vrednost zgornjega polja kaže naslednjo celico itd. Pri -1 je konec. Za vrednosti pogledamo trenutni indeks spodnjega polja.

PERSISTENCA

Pomeni pregled za nazaj. Ko nekaj dodamo, se ustvari referenca na tisto točko.



Implicitne in eksplicitne podatkovne strukture

IMPLICITNE
Porabijo manj prostora, ker hranijo le vrednosti objektov. Objekti shranjeni tako, da lahko na njih gledamo, kot da so v drugi strukturi (drevo v seznamu, sklad...).

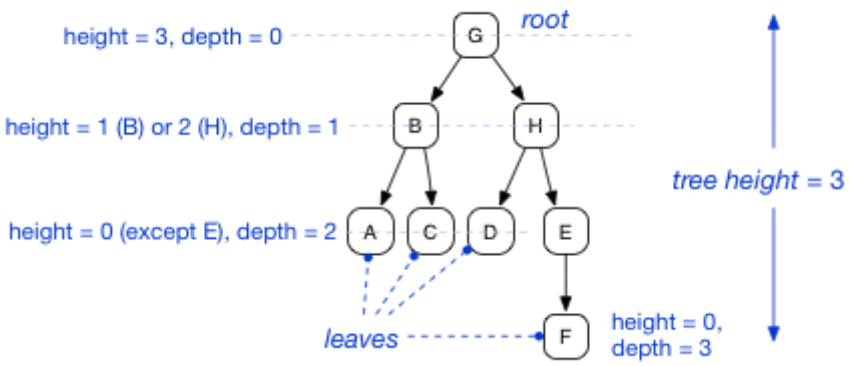
EKSPLICITNE
Potrebujejo veliko več prostora za shranjevanje kazalcev, ki so potrebni da zadeva sploh stoji.

Drevesa

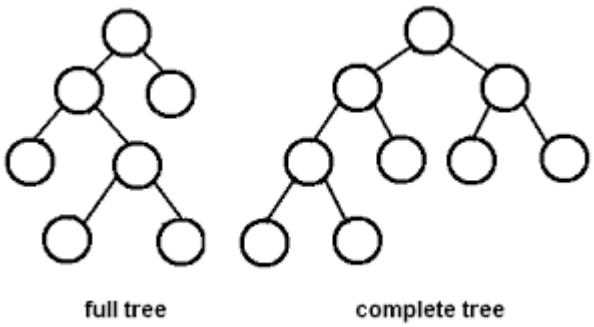
Nekaj osnovnih pojmov:

DREVO S KORENOM – UKORENINJENO DREVO
Notranje vozlišče, starš, otrok, prednik, potomec...?

KOREN – najvišji element v drevesni hierarhiji
VOZLIŠČE – element v drevesu
POVEZAVA – povezava med dvema vozliščema v drevesu
LIST – končno vozlišče
POT – povezava od izvora do cilja. Število poti je enako številu povezav.
PODDREVO – vozlišče in vsi njegovi potomci. Tudi LIST je poddrevo! Koren je poddrevo in poddrevo je drevo.
GOZD – množica dreves
UREJENO DREVO – lahko je urejeno glede na vrstni red otrok ali glede na urejenost starš/otrok
VIŠINA VOZLIŠČA – največje število povezav od vozlišča do najbolj oddaljenega lista
GLOBINA VOZLIŠČA – število povezav od korena do izbranega vozlišča. Nivo so vozlišča na isti globini.



STOPNJA VOZLIŠČA – število otrok oziroma poddreves
STOPNJA DREVESA – največja stopnja vozlišča; dvojiško, trojiško, d-tiško...
POLNO VOZLIŠČE – stopnja vozlišča je enaka stopnji drevesa
POLNO DREVO – vsa notranja vozlišča so polna (vsa vozlišča imajo enako število potomcev)
POPOLNO DREVO – polno drevo s tem, da so vsi listi na istem nivoju?
CELOVITO DREVO – vsi nivoji so polni, razen morda zadnjega, vsi listi na zadnjem nivoju so levo...?



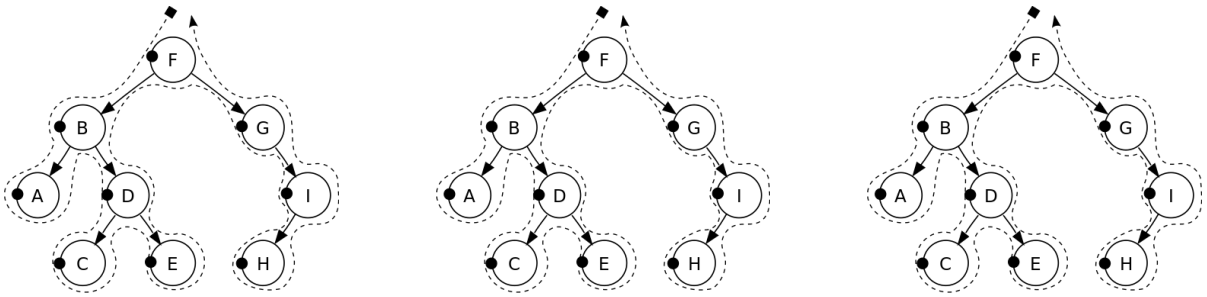
OSNOVNI ALGORITMI

- Štetje vozlišč
- Štetje listov
- Štetje notranjih vozlišč
- Globina vozlišča
- Višina vozlišča
- Stopnja drevesa
- Vsota stopenj vozlišč

DREVESNI OBHODI

Gre za sistematične obiske vseh vozlišč drevesa. Poznamo več vrst obhodov:

- Premi obhod (preorder); rekurzivno in v globino, izpisujemo od korena dalje
- Obratni obhod (postorder); rekurzivno in v globino, od korena do otrok
- Vmesni obhod (inorder); samo za dvojiška – najprej levo poddrevo v postorder, nato koren, nato desno poddrevo
- Obhod po nivojih (level order); iterativno po nivojih (v širino?)



INDEKS ELEMENTOV V DREVESU

ZA OTROKE – stopnja * trenutno vozlišče + število otroka (1 = 2i + 1 | 2 = 2i +2)
ZA STARŠE – trenutno vozlišče – 1 / stopnja drevesa ($p = \lfloor (i-1) / 2 \rfloor$)

PREDSTAVITEV DREVES
Drevesa lahko predstavljamo na različne načine.

S KAZALCI
- za otroke imamo zaporedje kazalcev,
- imamo kazalec na prvega otroka in potem vsak otrok kazalec na sorojenca,
- kazalec na starša (dobro za neurejena drevesa; se ne uporablja samostojno... hitrejše skakanje po drevesu)

S POLJEM
V polju imamo elemente in last index. Računamo indekse otrok, ki nas zanimajo. To je uporabno pri predstavitvi celovitih / popolnih dreves, ker seznam nima lukenj in neizkoriščenega prostora.

Kopica
Kopica je celovito dvojiško drevo z učinkovito predstavo v polju, delno urejenostjo vozlišč; velja urejenost med staršem in otroci.

Poznamo:
- MIN KOPICO; ključ starša <= ključ otroka
- MAX KOPICO; ključ starša >= ključ otroka

LASTNOSTI KOPICE
Koren vsebuje vedno največji/najmanjši element. Vsako poddrevo je kopica. Učinkovito jih lahko predstavljamo v polju:

- OTROKA: L = 2i+1, R = 2i+2
- STARŠ: P = $\lfloor (i-1) / 2 \rfloor$ (floor)
- NOTRANJA VOZLIŠČA: prvih n/2 elementov
- LISTI: zadnjih $\lceil n/2 \rceil$ elementov (ceil)

OPERACIJE NAD KOPICAMI

- DVIGANJE ELEMENTA – SIFT UP; ko smo 1 element stran od kopice. Menjamo otroka s starši dokler ni ok.
- VSTAVLJANJE – ENQUEUE: Element dodamo na konec kopice, velikost povečamo za ena in ustrezno premikamo nov element z 'sift up'
- UGREZANJE ELEMENTA – SIFT DOWN; starš na indeksu kvari urejenost; zamenjamo z večjim/manjšim otrokom
- ODVZEMANJE SPREDAJ – DEQUEUE; koren damo na stran in zadnjega vržemo na vrh, nato sift up/sift down za ok kopico

GRADNJA KOPICE Z DVIGANJEM
Kopico gradimo z dodajanjem elementov, ni potrebno poznavanje celotnega zaporedja. Ideja je, da je prazna kopica tudi kopica.

GRADNJA KOPICE Z UGREZANJEM
Poznati moramo vse elemente. Ideja je, da so listi kopica in imamo potem ugrezanje notranjih vozlišč – obhod po višini.

OSTALE OPERACIJE

operacija (max kopica)	zahtevnost
siftUp	$O(\lg n)$
siftDown	$O(\lg n)$
enqueue	$O(\lg n)$
dequeue	$O(\lg n)$
gradnja z dvigovanjem	$O(n \lg n)$
gradnja z spuščanjem	$\Theta(n)$
maksimum	$\Theta(1)$
drugi največji	$\Theta(1)$
iskanje elementa	$\Theta(n)$
večanje ključa elementa	$O(\lg n)$
zmanjševanje ključa elementa	$O(\lg n)$
odstranjevanje poljubnega elementa	$O(\lg n)$

Urejanje

Urejanje se uporablja praktično povsod, zato je pomembno da imamo algoritme, ki to počnejo kar se da hitro. Pri sortirnih algoritmih se vhodna instanca imenuje INSTANCA SORTIRNEGA PROBLEMA. Paziti moramo na omejitve, kako urejene so številke,... Tudi nepravilni algoritmi so lahko uporabni, če imajo omejeno možnost napake.

Urejamo lahko tako nestrukturirane (števila, nizi) kot strukturirane (zapisi, objekti, ključi...) podatke. Sortirn algoritm s primerjanjem, ki bi bili hitrejši od (n * log n) ne obstajajo, ker je št. Listov v optimalnem drevesu za sortiranje (n * log n).

Algoritmi, ki ne delujejo s primerjanjem, dosegajo tudi konstantne čase – imajo pa druge omejitve.

NAVADNA UREJANJA
Urejanje z izbiranjem (selection sort), vstavljanjem (insertion sort) in zamenjavami (bubble sort).

SELECTION SORT
Vzamemo en element in vse na njegovi levi, ki so manjši/večji prestavimo za ena v desno.

INSERTION SORT
Začnemo na začetku seznama. Vzamemo prvi element, ki ni v delno sortiranem delu in ga premikamo po seznamu v levo, dokler ne naletimo na nek večji/manjši element. Vsak tako premaknjen element je potem del delno sortiranega dela in ga ni treba posebej premikati.

BUBBLE SORT
Primerjava 2 elementov po seznamu navzgor. Če je tisti z nižjim indeksom večji od tistega z višjim, se zamenjata. S tem spravimo velike elemente proti koncu seznama – sortirani del na vrhu seznama. Algoritem ima kvadratno kompleksnost.

NAPREDNA UREJANJA
Urejanje s kopico (heap sort), zlivanjem (merge sort), hitro urejanje (quick sort)

MERGE SORT
Problem se najprej rekurzivno razbije do konca, potem pa se ti manjši seznami sproti združujejo skupaj da dosežemo urejenost. Razbijanje se splača ustaviti nekje pri dolžini 7 elementov in nadaljevati z navadnim vstavljanjem. Posebaj urejene sezname zlivamo v enega v O(n) času, če:

- Nastavimo pointer na začetek podseznamov
- Vzamemo najmanjšo vrednost na katero kaže kazalec in jo premaknemo za ena v desno
- Ponavljamo dokler nimamo vseh elementov

Je stabilen, naredi pa nov seznam.

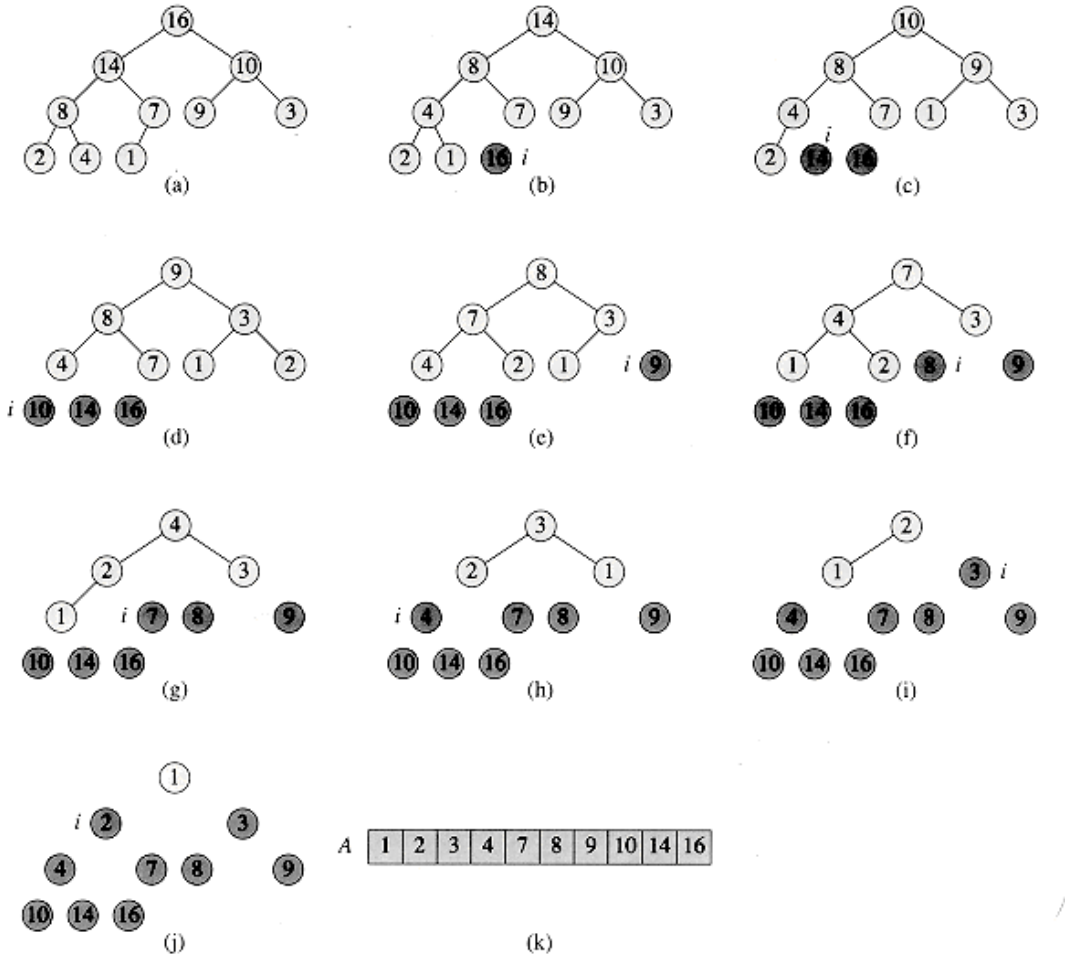
QUICK SORT
Urejanje glede na pivot. Če ga slabo izberemo je zahtevnost lahko $O(n^2)$. Naključni pivot pomeni swap levega elementa z naključnim v seznamu, potem pa obdržimo levi element kot pivot.

Načeloma veliko hitrejši sortirni algoritem od ostalih, worst case scenario je $O(n^2)$, average case $O(n * \lg n)$. Tudi tu lahko rekurzijo prej končamo in nadaljujemo z insertion sortom.

Več pivotov pride prav v praksi zaradi boljše izkoriščenosti pomnilnika.

HEAP SORT

Imamo dve verziji. Prva ustvari novo kopico vsakič ko poppamo element...? Druga pa gradi kopico na levi strani seznama in ureja desni del.



LOČITEV ALGORITMOV SORTIRANJA PO NAMENU

- STABILNOST; insertion, bubble, merge
- MAJHNA VELIKOST; insertion sort
- VELIKA VELIKOST; quick, merge, heap
- REKURZIJA – nikoli do konca!; merge, quick → do konca insertion.

Elemente ponavadi hranimo v notranjem pomnilniku; če je elementov preveč jih shranimo v datoteke. Quick sort ima s tem težave zaradi zaporednega zapisa, merge sort pa zaradi svoje narave to metodo zelo ceni.

STABILNOST UREJANJA

Pomeni, da ohranjamo prvotni vrstni red elementov pri enakih elementih?

Algoritmi brez primerjanja

Primerjanje vzame svoj delež časovne zahtevnosti. Če poznamo podatke in vemo omejitve lahko naredimo hitrejši algoritem – lahko dosežemo tudi linearno zahtevnost, za ceno drugih napak.

UREJANJE S ŠTETJEM

Potrebujemo dva dodatna seznama velikosti toliko, kot je maks vrednost. Poštejemo koliko je katerih elementov in iz tega poračunamo indekse. V števno polje prištevamo kolikokrat se pojavi kakšna vrednost – index števnega polja je enak vrednosti, katere pojavitve štejemo. Potem gremo linearno od zadaj naprej in v izhodno polje dodajamo X videnih elementov.

Če zadnji korak izvajamo od desne proti levi, je algoritem stabilen, drugače pa ne. Zakaj?

KORENSKO UREJANJE
Najprej sortiramo po enicah, nato deseticah, nato stoticah... Znebimo se gromozanskih seznamov pri večjih cifrah.

UREJANJE S KOŠI
Elemente mečemo v koše.

- Velikost posameznega koša je celoten razpon, št. različnih vrednosti ki se pojavi -> v: ((max - min +1) / št. košev)
- Indeks koša kamor element spada: i(x) = floor((x - min) / v)

Vrsta urejanja	Zahtevnost	Razno
Navadno vstavljanje	$O(n^2)$, best: $O(n)$	stabilno
Navadno izbiranje	$\Theta(n^2)$	
Navadne zamenjave	$\Theta(n^2)$	stabilno
Urejanje s kopico	$\Theta(n \log n)$	
Urejanje z zlivanjem	$\Theta(n \log n)$	stabilno, ni *in-place*, dodatni prostor
Hitro urejanje	$O(n^2)$, avg: $\Theta(n \log n)$	randomizacija, dodatni prostor
Urejanje s štetjem	$O(n + m)$	stabilno, končna množica
Korensko urejanje	$O(d(n + m))$	stabilno, končna množica
Urejanje s koši	$O(n^2)$, avg: $\Theta(n)$	stabilno?, enakomerno

---

Grafi

To so elementi povezani med seboj – zelo podobni drevesom, le da nimajo korenov. Poznamo usmerjene in neusmerjene grafe, z uteženimi ali neuteženimi potmi.

NEUSMERJENI GRAF
Je podatkovna struktura G = (V,E); podamo vozlišča in povezave. Pomeni da smer povezave iz enega v drugo vozlišče ni pomembna, po vsaki povezavi med dvema vozliščema se da priti v obe vozlišči.

? vozlišča in povezave – velikost grafa – označeni graf – uteženi graf (omrežje) – sosednost ● enak z enakim – incidenca ● vozlišča s povezavami – stopnja vozlišča ?

USMERJENI GRAF
Povezave so usmerjene. Tu imamo vhodno in izhodno stopnjo vozlišča – pomenita koliko povezav gre v ali iz posameznega vozlišča.

Usmerjeni graf je šibko povezan, če he ustrezen neusmerjeni graf povezan – povezave naredimo obojestranske.
Graf je povezan, če za vsak par vozlišč (u,v) obstaja pot u → v ali u ← v. Krepko povezan graf ima vozlišča z obojestranskimi povezavami?

PREDSTAVITEV GRAFA
Grafe lahko predstavimo na številne načine:
- SEZNAM SOSEDOV; za vsako vozlišče seznam sosedov – O(n+m)
- MATRIKA SOSEDNOSTI; za vsak par vozlišč imamo št povezav (0,1), če imamo uteži pa v matriko zapišemo utež – $O(n^2)$
- MATRIKA RAZDALJ; uporablja se za omrežja, za vsak par vozlišl imamo utež povezave med njima. Če ni povezave zapišemo neskončno. Neusmerjen graf ima simetrično matriko.
- INCIDENČNA MATRIKA; za vsak par imamo vozlišče in povezavo

Algoritmi na grafih
Nekaj uporabnih pojmov:

SPREHOD - WALK
Zaporedje povezav oz. vozlišč, katere začetek naslednje povezave je enak koncu prejšne. Poznamo odprt in zaprt sprehod.

STEZA - TRAIL
Tu vsaka povezava nastopa samo enkrat.

POT - PATH
Tu vsako vozlišče nastopa le enkrat.

CIKEL
To je zaprta pot.

Tu obstajajo neki algoritmi, kjer lahko preprosto množimo matrike za rezultat.
- Število sprehodov,
- Dosegljivost
- Število trikotnikov

Obhodi grafov

ISKANJE V GLOBINO - DEPTH FIRST SEARCH - DFS
Tu je načelo poguma. Gremo naprej, če lahko. Je rekurzivna metoda in lahko pridemo zelo daleč od začetka. Je pa problem, da dobimo velik stack rekurzije.

Gozd iskanja v globino sestoji iz dreves iskanja v globino. DFS(v) obišče vsa iz v dosegljiva vozlišča.

Vrstni red obiskovanja je lahko vstopni - ko vozlišče prvič obiščemo, ter izstopni, ko ga zadnjič obiščemo.

ISKANJE V ŠIRINO - BREADTH-FIRST SEARCH
Tu velja načelo previdnosti. Raziščemo svojo okolico in potem gremo nižje. Ta je dosti bolj zamuden, ampak lahko najde optimalno pot do neke točke.

Gozd iskanja v širino sestoji iz dreves iskanja v širino. BFS(v) obišče vsa iz v dosegljiva vozlišča. Vrstni red je tu samo vrstni red dodajanja v vrsto.

S tem se da ugotavljati ali je vozlišče sploh dosegljivo. Ugotovimo lahko tudi ali je graf usmerjen ali ne, cikličen ali ne.

NAJKRAJŠA POT
Z BFS lahko najdemo najkrajšo pot, ker gremo po nivojih. DFS lahko najde tisto čisto na koncu, katero po "sreči" najde.

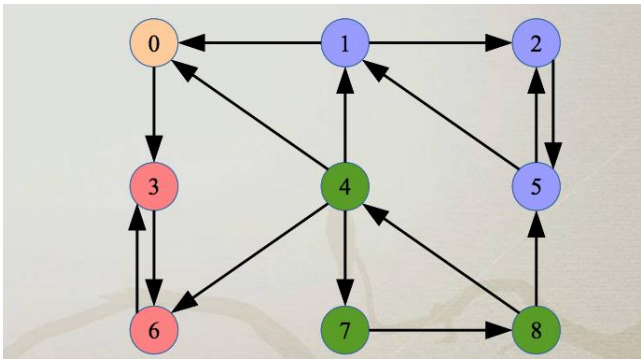
TOPOLOŠKO UREJANJE
Tu želimo vozlišča grafa prikazati tako kot so po vrstnem redu. Torej jih uredimo in povežemo kot so. Če vsebujejo kakšne cikle, grafa ne moremo topološko urediti.

Dobimo ga tako, da izpišemo DFS, izstopni vrstni red in tega obrnemo okrog – ali pa z odstranjevanjem vozlišč. Vozlišče, ki nima vstopnih povezav, je lahko na prvem mestu.
Jerman: Pol jih moreš odstranit, zraven pa še njihove incidenčne povezave. Naprej pa neb več vedu…

POVEZANOST
Graf je povezan, če med vsakim parom vozlišč obstaja pot. Usmerjeni grafi so šibko povezani.

POVEZANE KOMPONENTE
Komponenta je največji povezani podgraf

KREPKO POVEZANE KOMPONENTE SCC - STRONGLY CONNECTED COMPONENTS
Graf lahko razdelimo na največje krepko povezane podgrafe. To naredimo z Kosaraju-ovim in Tarjan-ovim algoritmom.



KOSARAJEOV ALGORITEM
Zapišemo topološki vrstni red transponiramo graf (obrnemo povezave), izvedemo DFS nad tem transponiranim grafom. Tam kjer se DFS konča in vzame novo točko za začetek je ena komponenta.

TARJANOV ALGORITEM

Metode snovanja algoritmov

To je sistematičen pristop, neko ogrodje/orodje k snovanju algoritma za reševanje danega problema. Obstajajo različne vrste pristopov. Posamezne algoritme lahko zasnujemo na več različnih načinov - iterativno, rekurzivno, … Problem moramo najprej jasno razumeti!

DIREKTEN NAPAD NA PROBLEM

- Groba sila, izčrpno preiskovanje
- Sestopanje, razveji & omeji

DEKOMPOZICIJA PROBLEMA (NA PODPROBLEME)

- Dinamično programiranje, požrešna metoda
- Deli & vladaj, zmanjšaj & vladaj

PREVEDBA PROBLEMA

- Prevedi & vladaj; prevedemo problem v nekaj, kar že znamo rešiti.

UPORABA NAKLJUČJA

- Randomizacija; npr. quick sort z random pivotom.

Optimizacijski problemi

Obstaja več rešitev. Izbrati pa moramo tisto, ki je najboljša - ponavadi tista najkrajša (najhitrejša). Omeniti moramo naslednje:

- NALOGA - kako so matematično opisani vhodni podatki?
- REŠITEV IN DOPUSTNA REŠITEV; kako so opisani izhodni podatki? Morebitne dodatne omejitve. Rešitev, ki zadošča vsem omejitvam je dopustna.
- CILJ IN KRITERIJSKA FUNKCIJA; funkcija nad rešitvijo, ki vrne vrednost rešitve. Minimizacija in maksimizacija.
- OPTIMALNA REŠITEV; to je dopustna rešitev, ki optimizira kriterijsko funkcijo.

Par zanimivih problemov je množenje s seštevanjem, potenciranje in iskanje podniza.
Brute force iskanje podniza vzame lahko do $O(n^2)$ časa. Lahko pa zadevo rešimo veliko bolje.

ALGORITEM RABIN-KARP

Za iskan substring izračuna hash, potem pa linearno potegne substring iz glavega in izračunava hashe. Če se hasha ujemata, je treba še linearno čez, ker je lahko prišlo do kolizije hash funkcij. Potem je potrebno izbrati še dovolj pametno hash funkcijo.

GROBA SILA - BRUTE FORCE

Direkten napad brez uporabe bližnjic in razmišljanja. Navadno temelji na definiciji problema, iz tega pa poskušamo zapisati kodo. Pogledamo problem in po definiciji naredimo kodo. Se držimo definicije, čeprav gremo lahko v eksponentno zahtevnost.

»Je pa fajn, ker je algoritem ponavadi dost berljiv in ni treba kakih komentarjev devat not.« - Tomaž Jerman, 2018

IZČRPNO PREISKOVANJE - EXHAUSTIVE SEARCH

To je podvrsta grobe sile. Uporabljena za reševanje kombinatoričnih problemov, torej pregleduje vse možnosti. S tem lahko preverimo veliko rešitev, ampak ni ravno optimalno.

SESTOPANJE

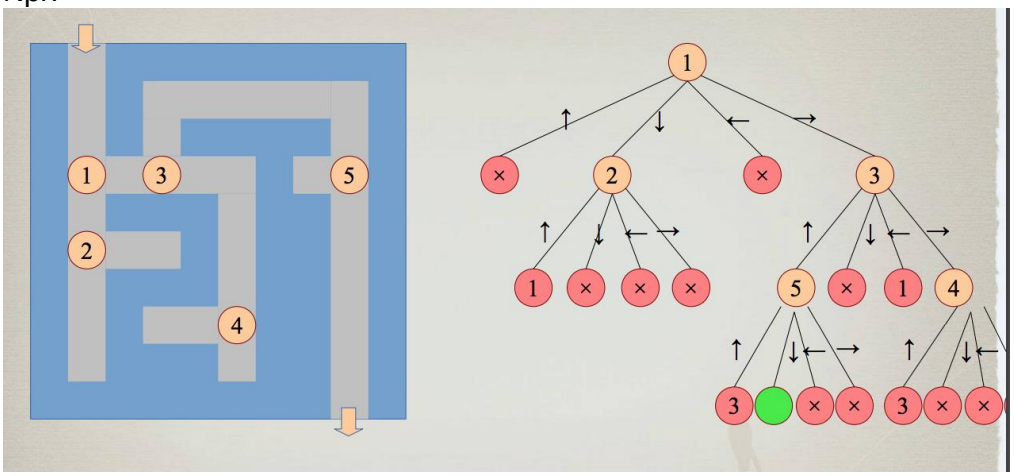
Rešitev generiramo po komponentah. S tem generiramo delne rešitve. Na vsakem koraku generiramo eno komponento, kjer upoštevamo vse možne izbire. Ko nimamo več nobene izbire, potem sestopimo - se vrnemo v predhodno stanje (korak - rekurzija se malo vrne). Ko pa zgeneriramo zadnjo komponento, se vse skupaj vrne kot rešitev.

Odločitveno drevo predstavlja sled sestopanja. Vozlišča so stanja, povezave pa prehodi.

FUN FACT: Za računalnik je pojem “veliko število” relativno glede na njegovo arhitekturo - kar je še pod overflow-om.

SESTOPANJE Z REZANJEM
Rezanje drevesa (pruning) - lahko režemo ali ne. Če ne režemo, potem se to sestopanje premakne v izčrpno preiskovanje. Če pa režemo, pa upoštevamo samo dopustne izbire. Glede na omejitve naloge problema in glede na trenutno delno rešitev. Tukaj je težje oceniti časovno zahtevnost. V praksi se izkažejo kot dovolj dobri.

Npr:



Zanimiva problema sta tudi

Vozliščno pokritje - izbrano je vsaj eno krajišče vsake povezave
Najmanjše vozliščno pokritje - najmanjša podmnožica vozlišč, ki je vozliščno pokritje

---

Deli in vladaj

Ideja tega pristopa je ta, da problem razdelimo na več manjših problemov, dokler ne dobimo nekega obvladljivega problema. Ko pridemo do majhnih problemov z razbitjem, te rešimo in jih potem nazaj lepimo v celoto - rešitev originalnega problema. To je rekurzivna metoda.

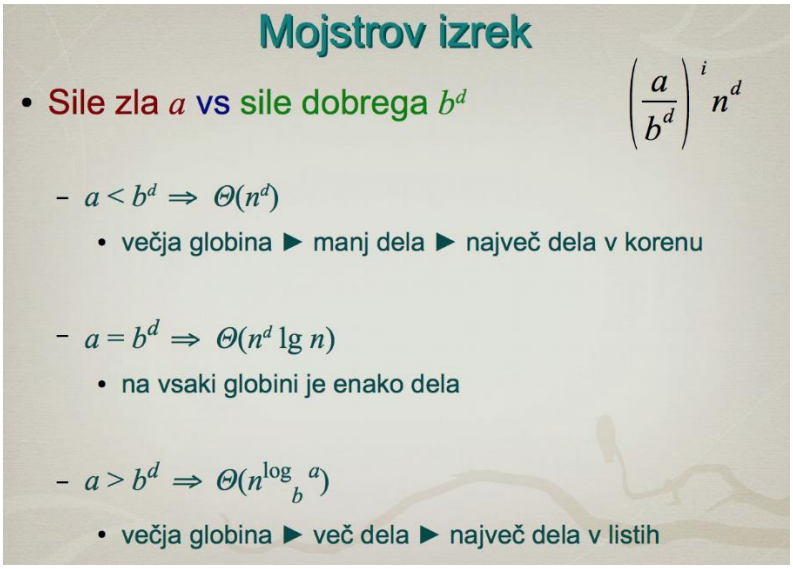
Rekurzivne algoritme lahko pogosto opišemo z rekurzivno enačbo oz. recurrence. Nato z matematičnimi orodji rešimo to rekurenco in s tem opišemo izvajanje algoritma za problem dolžine n. To z matematiko rešimo in dobimo neko zahtevnost algoritma.

KORAKI
- Delitev naloge – divide; razbij eno nalogo na več podnalog.
- Reši manjše naloge – conquer; podnalogo razbij naprej, ali pa reši, če je dovolj majhna
- Združevanje rešitev; združi podrešitve v rešitev prvotnega problema.

MASTER THEOREM
Uporaben pri iskanju časovne zahtevnosti divide and conquer algoritma. Če je rekurzivna enačba v naslednji obliki:

$$T(n)=\begin{cases}\Theta(n^d) & a<b^d\\ \Theta(n^d \lg n) & a=b^d\\ \Theta(n^{\log_b a}) & a>b^d\end{cases}$$

$$T(n)=a\cdot T(\frac{n}{b})+c\,n^d$$



Požrešni algoritmi

Izmed vseh algoritmov imamo neko zelo veliko podmnožico problemov, ki so NP popolni, kar pomeni, da še nihče ni odkril, kakšnega polinomskega algoritma (če ta sploh obstaja). Dokazati, da ne obstaja, pa tudi ne moremo, oziroma bi to bilo izjemno težko.

V takšnih primerih dobro delujejo požrešni algoritmi, je pa res da dobimo lokalni optimum ne pa globalnega.

Razveji in omeji

Tukaj problem rešujemo tako, da hranimo možne rešitve in ocenjujemo njihovo obetavnost. Zavržemo kandidate, ki ne morejo pripeljati do boljših rešitev. (KAKO TO VEMO!?)
Je izboljšano sestopanje. Za razliko od sestopanja se razlikuje ker ne išče po vejah rešitev, kjer ni boljših rešitev od že dobrih…