Fakultet elektrotehnike, strojarstva
i brodogradnje u Splitu

ALGORITMI

Hrvoje Dujmić

ožujak 2012.

ALGORITMI
Hrvoje Dujmić

(3+0+2)
Predavanja
Laboratorijske vježbe

LITERATURA

T.Cormen, C.Leiserson, R.Rivest, C.Stein (CLRS), «Introduction to Algorithms», second edition, third printing, The MIT Press & McGraw-Hill

SADRŽAJ

Analiza algoritama

temelji analize algoritama, najgori mogući slučaj (worst-case), notacija reda veličine (order notation), diskretna matematika (rekurzija, sumacija, asimptotsko ponašanje)

Sortiranje

algoritmi za sortiranje temeljeni na usporedbi (mergesort, quicksort, heapsort), donja granica sortiranja, sortiranje s linearnim vremenom

Graf algoritmi

pretraživanje po širini, najkraći putovi svih parova, Floyd-Warshall algoritam

Dinamičko programiranje

najduža zajednička podsekvenca, množenje lanca matrica

NP potpunost (NP-completeness)

Ostalo

Pohlepni algoritmi (Greedy algorithms) – Huffmanov kod
Brza Fourierova transformacija (Fast Fourier Transform – FFT)
Algoritmi u komunikacijama

1. UVOD

Što je to algoritam?

Algoritam je bilo koja dobro definirana procedura koja pretvara neki ulaz u željeni izlaz.

Algoritam definira, korak po korak, metodu za rješavanje računalnog problema.

Dobro razumijevanje algoritama je ključno za dobro razumijevanje najvažnijeg elementa računalne znanosti: programiranja. Stoga je važno naučiti kako napraviti algoritam i kako razumjeti već napravljene.

Algoritmi su blokovi od kojih se grade veliki sustavi, što podrazumijeva korištenje dodatnih znanja (npr. baze podataka) što se obrađuje u okviru drugih predmeta.

Odakle naziv algoritam?

Naziv algoritam potječe od perzijskog znanstvenika Muhammad ibn Musa al-Khwarizmi (živio od oko 780.-850. godine). Iskrivljavanjem (latinizacijom) njegovog prezimena došlo se do riječi algoritam (al-Khwarizmi je „otac“ i riječi algebra).

Odnos algoritma i programa

Algoritmi nisu programi, nego matematički entiteti i nikada ne bi trebali biti definirani u nekom programskom jeziku. Ukratko: algoritmi su objekti semantike, a programi objekti sintakse.

Cilj je napisati algoritam tako da se može implementirati u bilo kojem programskom jeziku za što koristimo pseudokod.

Zašto proučavamo algoritme i dizajn algoritama?

Postoji više aspekata o kojima moramo voditi računa a koji vode dobrom programu. Dobar dizajn algoritma je jedan od njih (i to jako važan). Da bi dizajnirali dobar algoritam treba biti svjestan ograničenja i svojstva programa kao i računala.

U bilo kojem većem programskom zadatku postoje dva aspekta programiranja: mikro i makro aspekt.

> Makro aspekt odnosi se primjerice na koordiniranje rada većeg broja programera koji rade na istom problemu ili na traženje načina da složeni program zadovolji različite zahtjeve. Makro aspekte programiranja proučava softversko inženjerstvo.
>
> Najveći dio programerskog vremena odnosi se na jednostavne zadatke (ulaz, izlaz, konverzija podataka, kontrola grešaka, generiranje izvještaja i sl.). Međutim, redovito postoji jedan mali ali kritični dio koda na kojem se troši glavnina vremena izvođenja programa. (Obično se kaže da 20% koda zahtijeva 80% vremena izvršavanja

programa). Mikro aspekt programiranja rješava problem kritičnih dijelova koda. U algoritmima se proučavaju mikro problemi i njihova rješenja.

Vrlo važno za uspjeh ukupnog (makro) projekta je da se kritični dio softvera (onih 20%) napiše što je moguće efikasnije. Nažalost, uobičajeni pristup je da se napiše prvo rješenje (algoritam) kojeg se programer sjeti. Nakon toga se nizom finih podešavanja i programerskih trikova algoritam nastoji ubrzati i memorijski zahtjevi smanjiti, a često se kupuju brza (i skupa) računala kako bi se program mogao izvoditi. Ako je algoritam od kojeg smo počeli loš (spor), onda nikakva dodatna podešavanja neće napraviti bitnu promjenu.

Prije implementacije, budite potpuno sigurni da je dizajn (algoritam) dobar.

Dakle, dobar algoritam treba ne samo riješiti problem (točno!) nego ga riješiti i to što je moguće brže, a uz to i sa što je moguće manje zauzimanja memorijskih resursa. Podsjetimo: svaka procedura koja vodi točnom rješenju jeste algoritam, ali svaki algoritam ne mora zadovoljiti naše zahtjeve (npr. za brzinom) pa u tom kontekstu možemo govoriti o boljim i lošijim algoritmima.

Kada bi računala bila beskonačno brza i s beskonačnom memorijom tada bi bolji i lošiji algoritmi bili ravnopravni. Međutim čak i tada bi imali potrebu proučavati algoritme jer bi htjeli znati da li je procedura koju smo napisali algoritam tj. da li vodi željenom izlazu.

Algoritmi su srce bilo koje računalne aplikacije. Oni su «računalni motor» koji pokreće veće softverske sustave. Stoga je razumijevanje algoritama neophodan preduvjet razumijevanju računalne znanosti i dobrog programiranja.

Ovaj predmet bavi se dizajnom dobrih algoritama.

Implementacija algoritama

Jedna od stvari na kojoj ćemo insistirati je proučavanje algoritama kao matematičkih objekata. U skladu s tim ignorirat ćemo elemente kao što su programski jezik, operacijski sustav, memorija, računalo i sl. Tako se možemo koncentrirati na bit algoritma. Međutim, detalji koje ignoriramo mogu biti vrlo važni.

Primjerice, danas procesori uglavnom imaju registre (brzu memoriju – cache memory) u koju pohranjuju često korištene podatke. Naša matematička analiza o toj memoriji ne vodi računa. Npr., tri algoritma za sortiranje koja ćemo spominjati (heapsort, mergesort i quicksort), imaju po matematičkoj analizi isto vrijeme izvršavanja. Međutim, na suvremenim procesorima najbrži se pokazao quicksort jer na najbolji način koristi registre. (Ipak, razlika je relativno mala – 10-20% razlike u vremenu izvršavanja.)

Vrste algoritama

Algoritme možemo na različite načine klasificirati, a obično se dijele po implementaciji, po metodi dizajna te po primjeni.

Klasifikacija po implementaciji
- Rekurzivni/iterativni
- Serijski/paralelni/distribuirani
- Deterministički/nedeterministički
- Točni/aproksimativni

Klasifikacija po dizajnu
- Podijeli pa vladaj
- Dinamičko programiranje
- Pohlepna metoda
- Linearno programiranje
- Redukcija
- Vjerojatnost i heuristika

Klasifikacija po primjeni
- Algoritmi za pretraživanje
- Algoritmi za sortiranje
- Numerički algoritmi
- Graf algoritmi
- String algoritmi
- Kriptografski algoritmi
- Kompresijski algoritmi

2. ANALIZIRANJE ALGORITAMA

Kako bi mogli dizajnirati dobar algoritam moramo se složiti oko toga što znači «dobar». Drugim riječima moramo imati kriterij mjerenja kvalitete algoritma. (Podsjetimo, algoritam po svojoj definiciji mora biti točan.)

Mjera kvalitete algoritma bit će računalni resursi koje algoritam treba. To će uglavnom biti vrijeme izvršavanja, a rjeđe potrebna memorija. U ovisnosti o aplikaciji mogu se mjeriti i druga svojstva: broj pristupa disku, pojasna širina i sl.

2.1. MODEL RAČUNALA

Za razliku od programa koje moraju razumjeti računala, algoritme moraju razumjeti ljudi (programeri). To nam daje određenu slobodu kod opisa algoritma. Primjerice sve nevažne detalje možemo izostaviti (posao programera je da uključi te detalje kod implementacije algoritma).

Želja nam je da algoritam bude što je moguće više neovisan o programskom jeziku, računalu, kompajleru i sl. Međutim, algoritmi se u konačnici ipak implementiraju na računalima pa stoga trebamo definirati jedan opći model računala na koji će se odnositi naši algoritmi. To je model koji bi trebao biti razumna apstrakcija standardnog jednoprocesorskog računala. Taj model zove se Random Access Machine (RAM).

RAM je idealno računalo s beskonačnom Random Access memorijom. Naredbe se izvršavaju jedna po jedna (nema paralelizma). Svaka naredba uključuje temeljne operacije na dvjema veličinama iz memorije (za sada ćemo se zadržati samo na karakterima i cijelim brojevima). Temeljne operacije uključuju primjerice pridruživanje vrijednosti varijabli, računanje osnovnih aritmetički operacija (+,-,*,/) na cijelim brojevima, usporedba (x<5 i sl.), pristup elementima polja (X[10]). Pretpostavljamo da za izvršenje svake temeljne operacija treba isto, konstantno vrijeme.

RAM model je vrlo dobar (i to još od 60-tih godina) za opisivanje računalne snage većine suvremenih (jednoprocesorskih) računala. Ipak, ovaj model ne uključuje neke elemente (npr. registre) o čemu ponekad treba voditi računa. Tako npr. model dozvoljava zbrajanje dva broja u istom, konstantnom vremenu, bez obzira na broj znamenaka. To naravno nije realno.

2.2. PROBLEM 2-D MAKSIMUMA: ALGORITAM GRUBE SILE

Formulacija problema

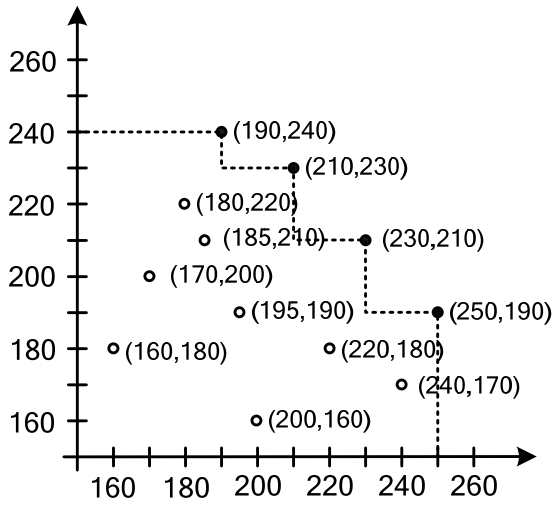
Prije nego damo potrebne definicije, analizirat ćemo jedan jednostavan algoritam koji će pomoći u boljem razumijevanju problema. To je primjer traženja 2-D maksimuma. Zamislite da želite kupiti automobil. Pri tome želite najbrži automobil. Kako naravno nemate previše novaca želite ujedno i najjeftiniji automobil. Ne možete se odlučiti što vam je važnije, brzina ili cijena. Jedino što ste definitivno sigurni je da ne želite automobil ako postoji takav koji je i brži i jeftiniji. Kažemo da brzi i jeftini automobil dominira sporim i skupim (relativno, u odnosu na zadani kriterij). Dakle, imajući listu automobila, mi zapravo tražimo sve one kojima niti jedan drugi automobil ne dominira.

Kako bi se lakše koncentrirali na bit, problem ćemo malo preformulirati. Svakom automobilu pridružimo par vrijednosti (x,y) gdje je x brzina automobila, a y negacija cijene (veći y znači manja cijena). To znači da maksimalna vrijednost odgovara bržem i jeftinijem automobilu.

Problem možemo formalizirati na slijedeći način:

> Neka je točka p u 2-D prostoru definirana svojim (cjelobrojnim) koordinatama p=(p.x, p.y). Kažemo da točka q dominira točkom p ako je $p.x \leq q.x$ i $p.y \leq q.y$. Ako imamo n točaka $P=\{p_1, p_2,...,p_n\}$ u 2-D prostoru za neku točku kažemo da je maksimalna ako ne postoji niti jedna druga točka iz P koja njom dominira.

Cilj nam je iz skupa svih točaka $P=\{p_1, p_2,...,p_n\}$ pronaći sve maksimalne, tj. sve one točke p_i kojima niti jedna druga točka iz P ne dominira.



Slika: 2D maksimum

Algoritam grube sile (Brute Force Algorithm)

Ne razmišljajući previše o efikasnosti zadani 2-D problem riješit ćemo na najjednostavniji mogući način: algoritmom grube sile.

Neka je $P=\{p_1, p_2,...,p_n\}$ zadani skup točaka. Svaku točku p_i testiraj (usporedi) sa svim drugim točkama p_j. Ako niti jedna druga točka ne dominira točkom p_i onda je ona maksimum.

S obzirom da je algoritam vrlo jednostavan implementacija bi bila lagana i na temelju ovog teksta. Ipak, formalizirat ćemo algoritma tako što ćemo ga napisati u pseudokodu.

```
MAX(n, P[1..n])
    for i=1 to n
        maksimum = true                          //P[i] je u startu maksimalan
        for j=1 to n
            if (i!=j) and (P[i].x <= P[j].x) and (P[i].y <= P[j].y)
                maksimum = false                 //P[j] dominira s P[i]
                prekini
        if (maksimum) ispiši P[i]                //niti jedna točka ne dominira P[i]
```

Ne postoje nekakva formalna pravila kako bi trebala izgledati sintaksa pseudokoda. Uglavnom, više detalja ne znači bolji pseudokod. Nevažni detalji se mogu i moraju izostaviti, naročito detalji koji zamagljuju problem i odvraćaju nas od glavne ideje (primjerice definiranje tipa varijable maksimum). Naravno, sve što je bitno mora se uključiti u pseudokod. Posebno treba voditi računa da se struktura vidi potpuno jasno i nedvosmisleno (primjerice, gdje počinje i završava pojedina for petlja). Podsjetimo, algoritme će čitati ljudi, pa razina uključenih detalja ovisi i o ciljanoj skupini ljudi kojima su namijenjeni.

U prethodnom primjeru nismo provjeravali konzistentnost (dosljednost) algoritma kako bi ideju maksimalno pojednostavnili. (Ako imamo dupliciranih točka tada imamo i problem koji u implementaciji moramo riješiti!)

2.3. DEFINIRANJE ALGORITAMA

Definirati neki algoritam ne znači samo napisati pseudokod. Pseudokod je samo jedan, i to ne najznačajniji, dio kod definiranja algoritama. Da bi algoritam bio u potpunosti definiran podrazumijeva se da smo dali:

1. Opis *algoritma i pseudokod.*
2. *Primjer ili dijagram koji će pokazati kako algoritam radi.*
3. *Dokaz (ili indikacija dokaza) da je algoritma ispravan.*
4. *Analizu vremena* izvršavanja algoritma.

Opis algoritma i pseudokod

Za potpunu definiciju algoritma potrebno je dati ne samo pseudokod nego i opis algoritma. Tako je i u ovoj skripti uz svaki algoritam dan opis ideje na kojoj se algoritam temelji. Osim toga, uz pseudokod je dan i kratki komentar pojedinih linija.

Primjer ili dijagram koji će pokazati kako algoritam radi

Ako je to realno, a najčešće jeste, onda se uz algoritam da i primjer ili dijagram koji pokazuje kako algoritam radi. Osnovna ideja algoritma često se najbolje vidi upravo iz primjera. Isto tako, iz primjera se dosta dobro mogu uočiti i eventualne greške u postavci algoritma.

Dokaz (ili indikacija dokaza) da je algoritma ispravan

Uz svaki algoritam potrebno je pokazati da je pseudokod koji smo napisali točan, tj. da vodi željenom rješenju. Kad god napišemo neki algoritam trebamo napisati i argumentaciju kojom potvrđujemo njegovu točnost. Ako se algoritam temelji na nekakvom triku, onda treba opisati zašto trik vodi točnom rješenju. Gore spomenuti primjer vrlo je jednostavan. Točan je jednostavno zato što smo implementirali definiciju: točka je maksimalna ako niti jedna druga točka njom ne dominira.

Analiza vremena izvršavanja algoritma

Glavni cilj matematičke analize bit će mjerenje vremena potrebnog za izvršenje algoritma (ponekad i potrebne memorije). Naravno, stvarno vrijeme izvršenja ovisi o implementaciji (npr. optimiziranje kompajlera). Takve i slične tehnološke aspekte želimo eliminirati iz proračuna. To odmah implicira da ne možemo razlikovati algoritme čija se vremena izvršavanja razlikuju za mali konstantni faktor jer će takvi algoritmi biti brži ili sporiji ovisno o tome kako dobro su iskoristili pojedino računalo ili kompajler. Ali, koliko je to malo? Pokazuje se da, kod proračuna potrebnog vremena, možemo zanemariti sve konstante, a da rezultat koji dobijemo još uvijek bude koristan. Dakle, da pruži realnu i upotrebljivu informaciju o potrebnom vremenu i da usporedba za dva različita algoritma ima smisla.

U našem primjeru možemo mjeriti vrijeme tako da računamo ili broj koraka u pseudokudu koji se izvrše ili koliko je puta pristupljeno elementu P ili broj izvršenih usporedbi.

Vrijeme izvršavanja ovisi o broju ulaznih podataka. Stoga moramo vrijeme izvršavanja definirati kao funkciju broja ulaznih podataka. (Da stvar pojednostavnimo, uzet ćemo da svaki ulazni podatak ima istu, maksimalnu veličinu.)

Različiti ulazi mogu, uz isti algoritam, rezultirati različitim vremenima izvršavanja (npr. u našem primjeru različit može biti broj koraka prije nego iziđemo iz unutrašnje petlje).

Obično se koriste dva kriterija za mjerenje vremena izvršavanja algoritma:

- najgore moguće vrijeme,
- prosječno vrijeme.

Najgore moguće vrijeme je maksimalno moguće vrijeme izvršavanja gledano u odnosu na sve moguće (valjane) ulaze veličine n. Neka je I valjani ulaz, neka je $|I|$ njegova dužina te neka je T(I) vrijeme izvršavanja algoritma uz ulaz I. Vrijedi:

$$T_{najgore\_moguće} = \max_{|I|=n} T(I)$$

Prosječno vrijeme računa se na svim mogućim ulazima veličine n. Za svaki ulaz I neka je p(I) vjerojatnost da će se javiti taj ulaz. Prosječno vrijeme izvršavanja dano je s

$$T_{prosjecno} = \sum_{|I|=n} p(I) \cdot T(I)$$

Gotovo uvijek ćemo raditi s najgorim mogućim vremenom. Razlog je što je prosječno vrijeme redovito vrlo teško izračunati (problem je raspodjela vjerojatnosti ulaza). Osim toga, za većinu algoritama, najgore moguće i prosječno vrijeme razlikuje se samo za konstantu, što u skladu s dogovorom i ovako zanemarujemo. Nadalje, u dosta algoritama najgore moguće vrijeme blisko je prosječnom (primjerice pretraživanje baza gdje često traženi element uopće nije u bazi).

Vrijeme izvršavanja algoritma grube sile za primjer traženja 2D maksimuma

Neka je veličina ulaznog skupa n te neka za vrijeme izvršavanja računamo koliko je puta pristupljeno elementu P. Iz pseudokoda je jasno da imamo dvije petlje: vanjska koja se prolazi n puta te unutrašnja koja se za svaki prolaz vanjske petlje u najgorem slučaju također prolazi n puta. If naredba ima 4 pristupa elementu P a ispis 2 pristupa (P[i].x i P[i].y). U najgorem slučaju svaka točka je maksimalna pa su u svakom prolasku vanjske petlje imamo 2 pristupa zbog ispisa.
Stoga najgore moguće vrijeme izvršavanja možemo izračunati kao dvije ugniježđene sumacije, jedna za i te jedna za j petlju

$$T(n) = \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n} 4 \right)$$

Kako je $\sum_{j=1}^{n} 4 = 4n$ slijedi

$$T(n) = \sum_{i=1}^{n} (2 + 4n) = (2 + 4n)n = 4n^2 + 2n$$

Kako smo već rekli, konstante zanemarujemo. Zanima nas slučaj kada je n velik (kada je n mali tada će svaki algoritam biti dovoljno brz). Kada je n velik tada faktor n^2 postaje bitno veći od faktora n pa ga možemo zanemariti.

Tako analizu vremena izvršavanja algoritma možemo završiti jednostavnom konstatacijom da je najgore moguće vrijeme izvršavanja algoritma grube sile jednako n^2 i to označavamo $\Theta(n^2)$. To se zove asimptotska brzina rasta funkcije.

2.4. SUMACIJE

Analiza vremena izvršavanja redovito uključuje računanje sume. Podsjetimo se nekih osnovni svojstava sumacije kao i iznosa tipičnih suma.

Ako je c konstanta (ne ovisi o indeksu sumacije i)

$$\sum_{i=1}^{n} ca_i = c\sum_{i=1}^{n} a_i \qquad \sum_{i=1}^{n}(a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Aritmetički red: za $n \geq 0$

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

Geometrijski red: neka je $x \neq 1$ konstanta (neovisna o i) te neka je $n \geq 0$

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + x^3 + ... + x^n = \frac{x^{n+1} - 1}{x - 1}$$

ako je $0 < x < 1$ tada je $\Theta(1)$; ako je $x > 1$ tada je $\Theta(x^n)$

Harmonijski red, neka je $n \geq 0$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} \approx \ln n = \Theta(\ln n)$$

Kvadratni red, neka je $n \geq 0$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + ... + n^2 = \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$

2.5. ANALIZIRANJE PETLJI

Promotrimo nešto složeniji primjer nego smo imali u slučaju algoritma grube sile. Na tom primjeru analizirat ćemo vrijeme izvršavanja algoritma sa složenim ugniježđenim petljama. Pri tome ćemo s ". . ." označiti sve eventualne naredbe koje se nalaze unutar petlji jer ćemo te naredbe potpuno ignorirati. Naime te naredbe su potpuno nevažne za analizu.

```
for i=1 to n                          // pretpostavimo da je n veličina ulaznog skupa
    . . .
    for j=1 to 2*i
        . . .
        k=j
        while (k>=0)
            . . .
            k=k-1
```

Kako analizirati algoritam s puno ugniježđenih složenih petlji? Najlakše je tako svaku petlju napišemo kao sumu te potom izračunamo (riješimo) sumu. Označimo s I(), M(), T() vrijeme izvršavanja unutrašnje petlje, srednje petlje i cijelog programa. Da bi konvertirali petlje u sume radimo iznutra prema vani. Promotrimo dakle, najprije unutrašnju petlju. Broj prolazaka kroz unutrašnju petlju ovisi o j, a izvršava se za $k = j, j-1, j-2, ..., 0$. Vrijeme provedeno unutar petlje je konstantno, pa je ukupno vrijeme jednako j+1. Ovaj zaključak možemo formalizirati na slijedeći način:

$$I(j) = \sum_{k=0}^{j} 1 = j + 1$$

Zašto "1" unutar sume? Zato što je vrijeme provedeno unutar petlje konstantno. Zašto brojimo od 0 do j, a ne od j do 0, kako se u petlji uistinu broji? Razlog je u matematičkoj definiciji sume čiji indeks uvijek mora ići od manjeg prema većem, a s obzirom da je zbrajanje komutativno sasvim je svejedno koji je redoslijed zbrajanja (rezultat je uvijek isti).

Razmotrimo sada prolazak kroz srednju petlju. Vrijeme izvršavanja srednje petlje određeno je s i. Koristeći prethodno izračunatu sumu kao i činjenicu da se petlja izvršava za j koji ide od 1 do 2*i, slijedi da je:

$$M(i) = \sum_{j=1}^{2i} I(j) = \sum_{j=1}^{2i} (j+1) = \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1$$

Druga suma iz prethodnog izraza jednaka je 2i. Prvu sumu možemo lako izračunati koristeći izraz za sumu aritmetičkog niza

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Uz malo prilagođavanja indeksa slijedi

$$M(i) = \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 = \frac{2i(2i+1)}{2} + 2i = 2i^2 + 3i$$

Sada možemo izračunati sumu vanjske petlje, a time i cijelog algoritma. Vanjska petlja se izvršava za i koji ide od 1 do n pa slijedi.

$$T(n) = \sum_{i=1}^{n} (2i^2 + 3i) = \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i$$

Druga suma u prethodnom izrazu je aritmetički niz pa je jednaka $\frac{n(n+1)}{2}$. Prva suma je kvadratni niz. Koristeći izraz za sumu kvadratnog niza, slijedi da je ukupno vrijeme izvršenja promatranog algoritma jednako

$$T(n) = \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i = 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2}$$
$$= \frac{4n^3 + 15n^2 + 11n}{6}$$

Kako smo to prethodno rekli, ignoriramo sve osim najbrže rastućeg faktora $4n^3/6$. Osim toga ignoriramo i konstantu pa preostane vrijeme izvršavanja $\Theta(n^3)$.

2.6. RJEŠAVANJE SUMA

Do sada smo spomenuli četiri karakteristična niza (sume) s pripadnim rješenjima. To su aritmetički, geometrijski, harmonijski i kvadratni niz. Ali kako postupiti kada dobijemo sumu čije nam rješenje nije poznato? Ovdje ćemo dati nekoliko ideja koje mogu pomoći u rješavanju takvih suma.

Upotreba grube granice

Ovo je jedan od najjednostavnijih pristupa i obično dobar rezultat daje kada se približavamo asimptotskoj granici. Temelji se na zamjeni svakog izraza iz sume jednostavnom gornjom granicom. Na taj način dobijemo sumu koja je gornja granica točnog rješenja pa će točno rješenje sigurno biti manje od tako izračunate sume. To ujedno znači da je asimptotska granica traženog niza sigurno manja ili jednaka izračunatoj sumi. Najčešće nam nije ni potrebna točnija informacija od te (najčešće nas niti ne zanima točna suma).

Ako promotrimo primjer kvadratnog niza $\sum_{i=1}^{n} i^2$ vidimo da se svaki izraz u sumi sigurno može zamijeniti s najvećim tj. s n^2. Slijedi da je

$$\sum_{i=1}^{n} i^2 \le \sum_{i=1}^{n} n^2 = n^3$$

Dobili smo rezultat koji je asimptotski potpuno jednak točnom rješenju, dakle $\Theta(n^3)$.

Ovaj pristup daje dobar rezultat za relativno spororastuće funkcije (npr. za funkciju oblika i^c gdje je c konstanta). Pristup obično ne daje dobar rezultat za brzorastuće funkcije kao što je npr. 2^i.

Aproksimiranje korištenjem integrala

Integriranje i sumiranje su usko povezani. Integrali se, pod određenim uvjetima, mogu smatrati kontinuiranim oblikom sume.

Neka je f(x) monotono rastuća funkcija (tj. neka funkcija raste kako raste x). Tada vrijedi

$$\int_{0}^{n} f(x)dx \le \sum_{i=1}^{n} f(i) \le \int_{1}^{n+1} f(x)dx$$

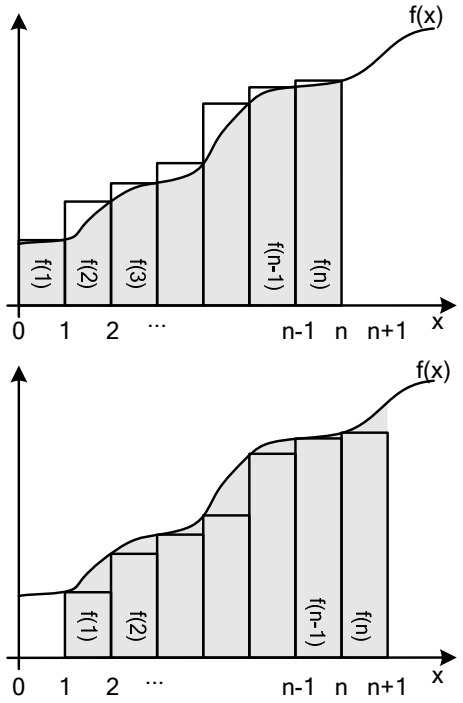
Potvrda da je prethodni izraz točan najbolje se može vidjeti na slici.

Najčešće je vrijeme izvršavanja uistinu monotono rastuća funkcija pa se gornji izraz može gotovo uvijek upotrijebiti za analizu vremena izvršavanja algoritma.

Upotrebom gornje formule možemo izračunati sumu kvadratnog niza. U tom slučaju je $f(x) = x^2$. Zanima nas samo gornja granica pa je

$$\sum_{i=1}^{n} i^2 \le \int_{1}^{n+1} x^2 dx = \frac{x^3}{3}\Big|_{1}^{n+1} = \frac{(n+1)^3}{3} - \frac{1}{3} = \frac{n^3 + 3n^2 + 3n}{3}$$

U ovom slučaju smo također dobili asimptotski rezultat potpuno jednak pravom izrazu, dapače potpuno je jednak i vodeći faktor u sumi tj. $n^3/3$.



Slika: Aproksimiranje sume pomoću integrala

Upotreba indukcije

Ovo je metoda koja je dobra kada pretpostavljamo oblik sume ali nismo sigurni kolike su pojedine konstante. Tako npr. za slučaj kvadratnog niza pretpostavimo da je rješenje oblika

$$\sum_{i=1}^{n} i^2 = an^3 + bn^2 + cn + d$$

Trebamo izračunati a, b, c i d (pretpostavljamo da su konstante tj. da ne ovise o n).

Gornji izraz ćemo dokazati pomoću indukcije te pri tom izračunati a, b, c i d.

Podsjetimo se najprije što je to indukcija i kako se indukcijom dokazuje neka hipoteza. U načelu, dokaz indukcijom je matematički ekvivalent petlji u programiranju. Neka je n cjelobrojna varijabla na kojoj se vrši indukcija. Teorem ili formula koju treba dokazati, a koju zovemo hipoteza je funkcija od n. Hipotezu označimo s HI(n). Dokaz indukcijom sastoji se od dva koraka. U prvom koraku pokažemo da postoji neki dovoljno mali n_0 za koji je hipoteza točna tj. da je točno HI(n_0). Ako je taj uvjet ispunjen prelazimo na drugi korak. Drugi korak kaže da moramo pokazati da je hipoteza točna za sve n-ove veće od n_0.

Dokaži HI(n_0)
for n=n_0+1 to beskonačno
Dokaži HI(n) uz pretpostavku da je HI(n') točno za sve n'<n

Ovim je definirana tzv. jaka indukcija jer pretpostavljamo da je hipoteza točna za sve n'<n. Obično je potrebno da je hipoteza točna samo za prvu nižu vrijednost tj. za n-1. U tom slučaju dokazivanje indukcije možemo ukratko opisati u slijedeća dva koraka:

1. Pokaži da je hipoteza točna za neki mali n (n_0) – to je praktično početni uvjet.
2. Ako je hipoteza točna za neki n-1 pokaži da iz toga slijedi da je hipoteza točna i za n – to je korak indukcije.

Primijenimo sada opisani postupak na primjer kvadratnog niza.

1. Početni uvjet (n=0)

   Uvrštavanjem u izraz kojim smo definirali hipotezu dobivamo

   $$\sum_{i=1}^{n} i^2 = 0 = a0^3 + b0^2 + c0 + d$$

   pa slijedi da je d=0.

2. Korak indukcije

   Pretpostavimo da je n>0 i da je hipoteza (formula) točna za sve n'<n. Na temelju toga trebamo pokazati da je formula točna i za sami n. Struktura dokaza sume pomoću indukcije praktično je uvijek ista. Napišimo prvo sumu koja ide od 1 do n, zatim odvojimo zadnji član u sumi, potom primijenimo pretpostavku da je hipoteza točna za n-1 (tj. da je točno $\sum_{i=1}^{n-1} i^2 = a(n-1)^3 + b(n-1)^2 + c(n-1) + d$). Nakon toga slijedi malo jednostavnog matematičkog kombiniranja, u našem slučaju to izgleda ovako:

   $$\begin{aligned}\sum_{i=1}^{n} i^2 &= \sum_{i=1}^{n-1} i^2 + n^2 \\ &= a(n-1)^3 + b(n-1)^2 + c(n-1) + d + n^2 \\ &= a(n^3 - 3n^2 + 3n - 1) + b(n^2 - 2n + 1) + c(n-1) + d + n^2 \\ &= an^3 + (-3a + b + 1)n^2 + (3a - 2b + c)n + (-a + b - c + d)\end{aligned}$$

   Sada trebamo dokazati da je gornji izraz uvijek (za svaki n) jednak $an^3 + bn^2 + cn + d$. Usporedbom dvaju izraza slijedi da za svaki n (veći od n_0 – u ovom slučaju veći od 0) mora vrijediti:

   $$\begin{aligned}a &= a \\ b &= -3a + b + 1 \\ c &= 3a - 2b + c \\ d &= -a + b - c + d\end{aligned}$$

   Već smo prije vidjeli da je d=0. Iz drugog uvjeta slijedi da je a=1/3. Dalje se lako može izračunati da je b=1/2 a c=1/6. Ovim smo izračunali sumu, te naravno pokazali da je hipoteza točna. Konačnu formulu glasi

   $$\sum_{i=1}^{n} i^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{2n^3 + 3n^2 + n}{6}$$

   Da je naša hipoteza bila pogrešna, tada bismo ili dobili uvjete koji se ne mogu riješiti ili bi dobili rješenja koja su ovisna o n. Ovim postupkom se, za razliku od prethodna dva dobije točno rješenje. Ipak, obično će nas sasvim zadovoljiti i asimptotsko rješenje kakvo dobivamo u prethodna dva postupka.

2.7. PROBLEM 2-D MAKSIMUMA: ALGORITAM PRELAŽENJA RAVNINE

Podsjetimo se problema traženja 2-D maksimuma. Kazali smo da točka q dominira točkom p ako je $p.x \leq q.x$ i $p.y \leq q.y$. Ako imamo n točaka $P=\{p_1, p_2,...,p_n\}$ u 2-D prostoru, kažemo da je točka maksimalna ako ne postoji niti jedna točka iz skupa P koja bi njome dominirala. Problem je pronaći sve maksimalne točke u skupu P.

Problem smo riješili algoritmom grube sile kojim uspoređujemo sve parove točaka pri čemu je vrijeme izvršavanja jednako $\Theta(n^2)$. Postavlja se pitanje postoji li pristup koji bi bio značajno bolji (brži).

Problem s algoritmom grube sile je da ne uključuje nikakvu "pamet" da bi ubrzao rješavanje. Primjerice neka točka p_j dominira točkom p_i. Tada svakom točkom kojom dominira p_i sigurno dominira i p_j pa onda to nije ni potrebno računati. To je posljedica činjenice da je dominacija kao funkcija tranzitivna. Korištenje tranzitivnost ipak ne vodi bitno bržem algoritmu. Što ako su sve točke maksimalne?

Algoritam prelaženja ravnine

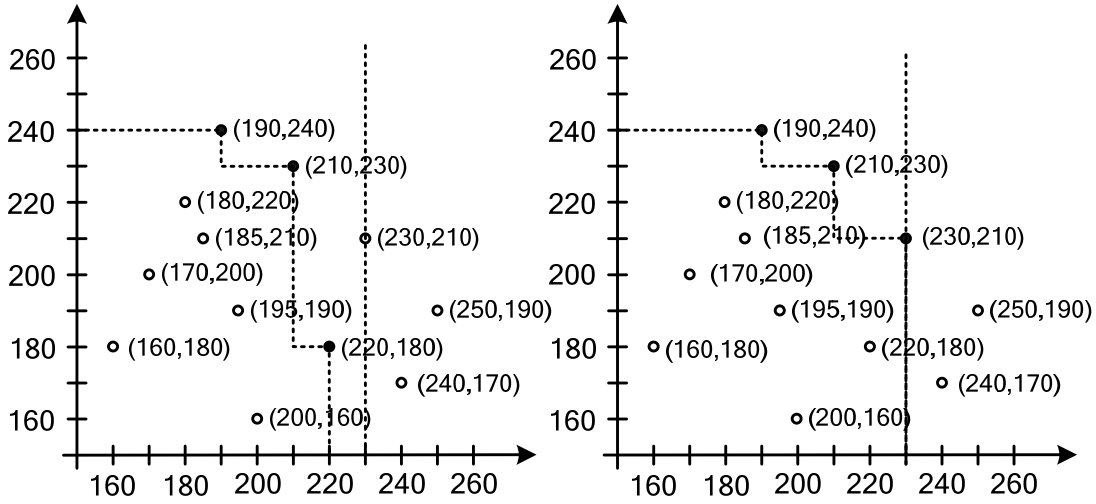
Algoritam prelaženja ravnine je bitno brži od algoritma grube sile. Ideja je slijedeća. Prelazit (prebrisat) ćemo cijelu ravninu vertikalnom linijom i to od lijeva prema desno. Kako prelazimo linijom, stvaramo strukturu koja sadrži maksimalne točke koje leže lijevo od linije kojom prelazimo ravninu. Kada linija kojom prelazimo dosegne krajnju desnu točku iz skupa P, dobili smo skup svih maksimuma. Ovakav pristup rješavanja problema naziva se prelaženje (prebrisavanje) ravnine.

Da bi mogli prelaziti po 2-D ravnini potrebno je točke posložiti po rastućim vrijednostima x komponente. Zbog jednostavnosti, pretpostavit ćemo da ne postoje dvije točke s istim x-om. (Kao i do sada radit ćemo sa što je moguće jednostavnijom verzijom kako razni detalji ne bi zamaglili stvarni problem. Naravno, u samoj implementaciji svi detalji će morati biti uključeni). Nakon što smo posložili točke P možemo liniju pomicati od točke do točke u n diskretnih koraka. Kako dolazimo do pojedine točke tako osvježavamo listu maksimalnih točaka.

Kako posložiti točke po rastućim vrijednostima x komponente? U nastavku ovog predmeta upoznat ćemo nekoliko algoritama kojima se to radi. Za sada je važno da postoji više takvih algoritama i da je asimptotsko vrijeme izvršavanja tih algoritama jednako $\Theta(n \log n)$.

Dakle jedini problem koji je preostao jeste kako osvježit listu maksimalnih točaka. Tvrdimo da je svaka slijedeća točka koja dođe na red pomicanjem linije sigurno maksimalna za trenutni skup prebrisanih točaka. To se lako zaključi iz činjenice da je to točka koja trenutno ima najveću x koordinatu i ne može postojati niti jedna točka koja bi njome dominirala. Međutim, ta točka može dominirati nekima od trenutnih maksimalnih točaka koje se onda

mogu izbrisati iz liste maksimalnih točaka. Točka jednom obrisana jer nije maksimalna, više se nikada neće vratiti na listu maksimalnih točaka, dakle sigurno neće biti maksimum.



Slika: Metoda prelaženja ravnine

Neka je p_i trenutna točka koju promatramo. Točka p_i sigurno ima najveću x koordinatu, pa će dominirati svim onim točkama od kojih ima veću y koordinatu. Tako je naš zadatak pronaći među trenutnim maksimalnim točkama one koje imaju manju ili jednaku y koordinatu i eliminirati ih.

Sada je važno uočiti da su maksimalne točke poredane po rastućim x koordinatama, a da iz toga nužno slijedi da su y koordinate poredane po padajućim vrijednostima. Zašto je to tako? Pretpostavimo suprotno, tj. da imamo dvije maksimalne točke p i q takve da je $p.x \geq q.x$ te također da je $p.y \geq q.y$. U tom slučaju točka p dominira točkom q pa točka q nije maksimum što je u suprotnosti s početnom tvrdnjom.

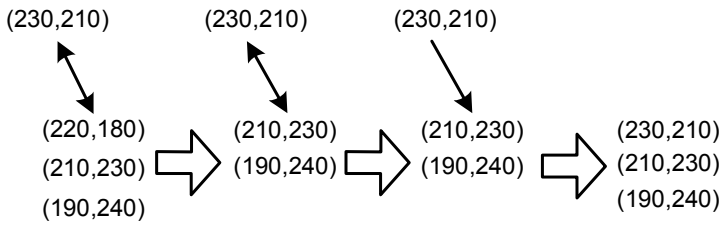
To je korisno svojstvo jer implicira da će sve točke kojima p_i dominira biti na kraju liste maksimalnih točaka. Naš je dakle zadatak samo pronaći onu točku u kojoj se to dešava. Kako pronaći tu točku? Jednostavnim linearnim prolaskom kroz listu. Jedino što moramo paziti da prolazak bude u pravom smjeru. Koji smjer je ispravan? Od lijeva prema desno (tj. u smjeru rasta x-koordinata) sve dok ne naiđemo na prvu točku koja više nije dominantna ili od desna prema lijevo sve dok ne naiđemo na prvu točku koja je još uvijek dominantna. Kad kažemo još uvijek dominantna mislimo nakon dodavanja nove točke tj. nakon dodavanja p_i.

Točan odgovor je od desna prema lijevo. Zašto je to tako? Ako listu prolazimo od desna prema lijevo tada će svaka točka koju prođemo, a čija je y koordinata manja od y koordinate točke p_i biti izbrisana iz liste i nikada više se neće pojaviti u računanju. Međutim, ako listu prolazimo s lijeva na desno, tada ćemo imati točaka koje ćemo prolaziti više puta. U najgorem slučaju (kada su sve točke maksimumi), svaki put ćemo prolaziti preko svih točaka koje su na listi. U tom slučaju to bi bio $\Theta(n^2)$ algoritam, isto kao i u slučaju algoritma grube sile.

Sada možemo napisati pseudokod za algoritam prelaženja ravnine. Kako točke brišemo s kraja liste, a isto tako ih i dodajemo na kraj liste, možemo koristiti stog (stack) za pohranu

maksimalnih točaka. Na kraju stoga nalazi se točka s najvećom x-koordinatom. Sa S označimo stog, pri čemu je zadnji (najviši) element stoga jednak S.top. Pop(S) znači izbacivanje zadnjeg elementa stoga S.

```
MAX2(n, P[1..n])
    sortiraj P po rastućim vrijednostima x-koordinate
    S=empty                                  //inicijaliziraj stog
    for i=1 to n
        while ((S is not empty) and (S.top.y <= P[i].y))
            Pop(S)                           //izbaci točku kojom P[i] dominira
        Push(S,P[i])                         //dodaj P[i] u stog
    ispiši sadržaj stoga S
```



Slika: Sadržaj stoga za zadani primjer

Zašto je algoritam točan? Točnost slijedi iz dane diskusije. Najvažniji je slijedeći zaključak: kada su maksimalne točke poredane po padajućoj x-koordinati onda su nužno poredane po rastućoj y-koordinati. Iz toga slijedi da, idući u stogu po rastućim vrijednostima y-koordinate, čim naiđemo na prvu točku koja ostaje dominantna, da će takve ostati i sve ostale točke.

Analiza vremena trajanja algoritma

Ovo je zanimljiv program za analizu vremena trajanja s obzirom da se mora primijeniti ponešto drugačiji pristup nego smo spominjali u prethodnom poglavlju. Kao prvo, imamo dio pseudokoda koji radi sortiranje n elemenata. U nastavku predavanja vidjet ćemo da je vrijeme izvršavanja algoritama za sortiranje $\Theta(n \log n)$. Koliko je vrijeme izvršavanja drugog dijela pseudokoda? Očito da u tom dijelu imamo dvije ugniježđene petlje. Vanjska petlja se sasvim sigurno izvršava n puta. Unutrašnja while petlja može se, u najgorem slučaju izvršiti n-1 put. To je slučaj kada zadnja dodana točka dominira svim ostalima, a do tada su sve točke bile maksimumi. To na prvi pogled izgleda da imamo $n \cdot (n-1)$ što opet vodi prema ukupnom vremenu izvršavanja od $\Theta(n^2)$.

Međutim, nakon pažljivijeg razmatranja možemo zaključiti da to nije tako. Naime, iako se u najgorem slučaju unutrašnja petlja može izvršiti n-1 put, ukupno se u svim prolascima (tj. u cijelom algoritmu) unutrašnja petlja sigurno neće izvršiti više od n puta. Razlog je vrlo jednostavan. Ukupan broj elemenata koji će tijekom izvršavanja cijelog programa dospjeti u stog je n (naredba Push će se izvršiti n puta). Stoga je nemoguće iz stoga izbaciti (naredba

Pop, unutar while petlje) više od n elemenata, pa je jasno da će se unutrašnja petlja (while naredba) izvršiti maksimalno n puta.

Dakle, ukupan broj iteracija unutrašnje petlje je maksimalno n (ne broj iteracija u svakom prolasku unutrašnje petlje, nego broj iteracija u cijelom programu dakle za sve prolaske vanjske petlje). Nadalje broj prolazaka vanjske petlje je n pa je onda i ukupno vrijeme izvršavanja tog dijela algoritma jednako $\Theta(n)$. Kako je vrijeme sortiranja jednako $\Theta(n \log n)$, ukupno vrijeme izvršavanja cijelog algoritma bit će također $\Theta(n \log n)$.

Usporedba vremena izvršavanja algoritma grube sile i algoritma prelaženja ravnine

Vrijeme izvršavanja algoritma grube sile jednako je $\Theta(n^2)$, a vrijeme izvršavanja algoritma prelaženja ravnine $\Theta(n \log n)$. Po kojoj bazi ćemo računati logaritam u izrazu $n \log n$? Kasnije ćemo vidjeti da baza logaritma nije bitna u asimptotskom slučaju. Stoga ćemo uzeti bazu 2 – to je naime najnepovoljniji slučaj za algoritam prelaženja ravnine (koristit ćemo slijedeću oznaku: $\log_2 n = ldn$).

Kolika je razlika u vremenu izvršavanja ovih dvaju algoritama? Da li je ta razlika uistinu bitna? To ćemo najbolje vidjeti tako da izračunamo omjer vremena izvršavanja dvaju algoritama.

$$\frac{n^2}{nldn} = \frac{n}{ldn}$$

Za male vrijednosti n (recimo manje od 100), razlika vremena izvršavanja algoritma je zanemariva ili vrlo mala ($100/ld100 = 15$). Za malo veći broj ulaza, recimo $n = 1000$, omjer je jednak $n/ldn \approx 1000/10 = 100$. Dakle, vrijeme izvršavanja algoritma prelaženja ravnine je 100 puta kraće od vremena izvršavanja algoritma grube sile. Naravno, ovdje nismo vodili računa o konstantama koje smo, s obzirom da računamo asimptotski slučaj, zanemarili. Kako su oba algoritma vrlo jednostavna, teško je zamisliti da će se konstantni faktori razlikovati za više od 10 puta. Čak i ako je takav kontstantni faktor na štetu algoritma prelaženja ravnine, on će još uvije biti 10 puta brži.

Promotrimo sada još veći ulaz. Neka je $n = 1000000$. U tom slučaju omjer je jednak $n/ldn \approx 1000000/20 = 50000$, što je značajna razlika. Čak i ako uzmemo u obzir konstantni faktor od 10 na štetu algoritma prelaženja ravnine, još uvijek je razlika 5000 puta. Primjerice: ako se algoritam prelaženja ravnine izvršava samo jednu sekundu, algoritam grube sile izvršavat će se skoro sat i pol (odnosno skoro 15 sati ako ne uzmemo u obzir konstantni faktor). Naravno, kako povećavamo broj ulaznih točaka, razlika postaje sve veća.

Iz ovog primjera može se vidjeti važnost asimptotske analize. Takva nam analiza kaže koji je algoritam bolji kada je n jako velik (kako smo već rekli, ako je n relativno mali, tada će svaki algoritam biti dovoljno brz). Obično korisnici povećavaju broj ulaznih podataka sve dok vrijeme izvršavanja ne postane toliko da se više ne može tolerirati. Dobrim dizajnom algoritma omogućuje se korisniku da koristi program s većim broj ulaznih podataka. Naravno, postoje i takvi problemi, koji već u startu imaju toliko ulaznih podataka da se mogu riješiti samo dobro dizajniranim algoritmima.

2.8. ASIMPTOTSKO OZNAČAVANJE (ASIMPTOTSKA NOTACIJA)

Sada ćemo definirati $\Theta(\cdot)$ kojeg smo do sada koristili za označavanje, a bez definicije, te ćemo potom definirati i druga asimptotska označavanja koja koristimo kod izrade i proučavanja algoritama.

Θ-označavanja

Definicija: Za danu funkciju $g(n)$, definiramo $\Theta(g(n))$ kao skup funkcija koje su asimptotski ekvivalentne s $g(n)$:

$$\Theta(g(n)) = \{ f(n) : \text{ postoje pozitivne konstante } c_1, c_2 \text{ i } n_0 \text{ takve da je } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ za sve } n \ge n_0 \}$$

Ovo je dakle, formalna definicija onoga što smo u prethodnim poglavljima radili na način "odbaci sve faktore osim najbrže rastućeg, te ignoriraj sve konstante". Dakle, funkcija $f(n)$ pripada skupu $\Theta(g(n))$ ako postoje takve c_1 i c_2 da $f(n)$ možemo staviti u «sendvič» između $c_1 g(n)$ i $c_2 g(n)$ za dovoljno veliki n.

Kao prvo, možemo uočiti da smo koristili pogrešno označavanje kada bi napisali primjerice $T(n) = \Theta(n^2)$. Naime, na lijevoj strani je funkcija, a na desnoj skup funkcija pa bi ispravno bilo napisati $T(n) \in \Theta(n^2)$. Ipak, takvo, u suštini pogrešno označavanje, uobičajeno je u algoritmima.

Primjer Θ-označavanja $f(n) = 4n^2 + 2n$

Ako se podsjetimo algoritma grube sile tada smo izračunali vrijeme izvršavanja algoritma kao $T(n) = 4n^2 + 2n$ za što smo rekli da je $\Theta(n^2)$. Provjerimo sada da je to uistinu tako. U ovom slučaju je $g(n) = n^2$. Mi trebamo pokazati da je $f(n) = 4n^2 + 2n$ član tog skupa tj. da postoje takve konstante c_1, c_2 i n_0 takve da je

$$0 \le c_1 n^2 \le (4n^2 + 2n) \le c_2 n^2 \text{ za sve } n \ge n_0$$

Ovdje zapravo postoje tri nejednakosti. Prva nejednakost $0 \le c_1 n^2$ je uvijek ispunjena s obzirom da uvijek radimo s pozitivnim n i pozitivnim konstantama. Druga nejednakost je

$$c_1 n^2 \le (4n^2 + 2n)$$

Ako uzmemo da je $c_1 = 4$ tada imamo $4n^2 \le (4n^2 + 2n)$ što je sigurno točno za sve $n \ge 0$. Treća nejednakost je

$$(4n^2 + 2n) \le c_2 n^2$$

Ako uzmemo da je $c_2 = 6$, dobivamo $n^2 \ge n$, što je ispunjeno za $n \ge 1$.

Imamo dakle, dva ograničenja za n, $n \geq 0$ i $n \geq 1$ pa uzimamo strože, dakle $n_0 = 1$. Time smo našli c_1, c_2 i n_0 (c_1=4, c_2=6, n_0=1) te tako formalno dokazali da je $(4n^2 + 2n) \in \Theta(n^2)$.

Intuitivno, ono što želimo kazati s $f(n) \in \Theta(g(n))$ je da su funkcije $f(n)$ i $g(n)$ asimptotski ekvivalentne. To znači da imaju istu brzinu rasta za veliki n. Tako npr. funkcije kao što su n^2, $n^2/6$, $9n^2 + 3n - 5$ i $n(n/2 - 1)$ intuitivno imaju istu brzinu rasta jer, kako n postaje velik, tako najbrže rastući (dominantni) faktor postaje n^2. Dakle, sve te funkcije rastu s kvadratom broja n (naravno za veliki n). Dio izraza u definiciji u kojem tražimo c_1 i c_2 u načelu kaže da konstante nisu važne jer možeš uzeti c_1 i c_2 kakve god hoćeš (ako zadovoljavaju tražene uvjete). Dio definicije u kojem biramo n_0 kaže da nas zanima samo veliki n, budući da uvjete moramo zadovoljiti samo za n-ove veće od n_0. To znači da n_0 može biti konstanta po volji velika.

Primjer Θ-označavanja $f(n) = 8n^2 + 4n - 5$

Promotrimo funkciju $f(n) = 8n^2 + 4n - 5$. Neformalno pravilo kaže da možemo odbaciti sve članove osim vodećeg, te također da možemo odbaciti sve konstante, iz čega slijedi da je $f(n) \in \Theta(n^2)$. Da bi dokazali da ovo neformalno razmatranje uistinu zadovoljava uvjete definicije moramo dokazati dvije stvari. Prvo moramo dokazati da $f(n)$ raste asimptotski brzo barem kao i n^2 (donja granica). Drugo, moramo dokazati da $f(n)$ ne raste asimptotski brže od n^2 (gornja granica).

Donja granica: moramo pokazati da $f(n)$ raste asimptotski brzo barem kao i n^2. Ovo ćemo dokazati onim dijelom definicije koji kaže da postoje pozitivne konstante c_1 i n_0 takve da je $f(n) \geq c_1 n^2$ za sve $n \geq n_0$. Vrijedi:

$$f(n) = 8n^2 + 4n - 5 \geq 8n^2 - 5 = 7n^2 + (n^2 - 5) \geq 7n^2$$

Dakle, za $c_1 = 7$ vrijedi $f(n) \geq 7n^2$. Međutim, gornji izraz je točan samo pod nekim uvjetima. Uvjeti su da je $4n \geq 0$ i $n^2 - 5 \geq 0$. Ta dva uvjeta točna su samo za dovoljno veliki n, konkretno za $n \geq \sqrt{5}$. Dakle, ako izaberemo bilo koji n_0 za koji vrijedi $n_0 \geq \sqrt{5}$ te ako izaberemo $c_1 = 7$, imamo zadovoljen uvjet donje granice.

Gornja granica: moramo pokazati da $f(n)$ ne raste asimptotski brže od n^2. Ovo je određeno onim dijelom definicije koji kaže da postoje pozitivne konstante c_2 i n_0 takve da je $f(n) \leq c_2 n^2$ za sve $n \geq n_0$. Vrijedi:

$$f(n) = 8n^2 + 4n - 5 \leq 8n^2 + 4n \leq 8n^2 + 4n^2 = 12n^2$$

Dakle, za $c_2 = 12$ vrijedi $f(n) \leq c_2 n^2$. Kao i kod proračuna donje granice i ovdje moramo uzeti u obzir uvjete pod kojima je izraz točan. U ovom slučaju to je uvjet $4n \leq 4n^2$ što je točno za $n \geq 1$. Dakle, ako izaberemo $n_0 \geq 1$ i $c_2 = 12$, imamo zadovoljen uvjet gornje granice.

Uvjet koji zadovoljava donju granicu je $n_0 \geq \sqrt{5}$, a gornju $n_0 \geq 1$. Uzimajući u obzir oba zahtjeva slijedi da je $n_0 \geq \sqrt{5}$. Sada, kao zaključak možemo kazati: ako uzmemo da su $c_1 = 7$, $c_2 = 12$ i $n_0 \geq \sqrt{5}$, tada vrijedi

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ za sve } n \geq n_0$$

što je uvjet koji zahtijeva definicija. Dakle, pokazavši da postoje konstante c_1 i c_2 i n_0, dokazali smo da je $f(n) \in \Theta(n^2)$.

Pokažimo sada zašto $f(n)$ asimptotski ne teži u neku drugu funkciju. Pokažimo primjerice da $f(n) \notin \Theta(n)$. Da bi to bilo točno mora biti zadovoljen uvjet i gornje i donje granice. Može se lako pokazati da je donja granica zadovoljena. Međutim, gornja granica nije zadovoljena. Gornja granica zahtijeva da postoje takve pozitivne konstante c_2 i n_0, da je $f(n) \leq c_2 n$ za sve $n \geq n_0$. Iako je intuitivno jasno da ne možemo pronaći toliko veliki c_2 da u asimptotskom slučaju vrijedi $8n^2 + 4n - 5 \leq c_2 n$, ipak ćemo to dokazati i formalno. Pretpostavimo da postoje takvi c_2 i n_0 koji zadovoljavaju traženi uvjet, tada bi on morao biti zadovoljen i kada n teži u beskonačnost. Ako obje strane podijelimo s n slijedi:

$$\lim_{n\to\infty}\left(8n + 4 - \frac{5}{n}\right) \leq c_2$$

Lijeva strana teži u beskonačnost pa nejednakost nikada ne može biti točna, bez obzira koliko veliki c_2 uzeli. Iz toga slijedi da $f(n) \notin \Theta(n)$.

Pokažimo sada da $f(n) \notin \Theta(n^3)$. Ovdje ćemo pokazati da nije zadovoljena donja granica. Donja granica kaže da postoje takve konstante c_1 i n_0 da je $f(n) \geq c_1 n^3$ za sve $n \geq n_0$. Pretpostavimo, dakle, da postoje pozitivne konstante c_1 i n_0 takve da je $8n^2 + 4n - 5 \geq c_1 n^3$ za sve $n \geq n_0$. U tom slučaju nejednakost mora biti zadovoljena i kada n teži u beskonačnost. Ako obje strane nejednakosti podijelimo s n^3 (to je dozvoljeno za $n > 0$, što u našem slučaju jeste zadovoljeno) tada imamo:

$$\lim_{n\to\infty}\left(\frac{8}{n} + \frac{4}{n^2} - \frac{5}{n^3}\right) \geq c_1$$

Lijeva strana teži u 0, pa će nejednakost biti zadovoljena samo ako je $c_1 = 0$. Po definiciji c_1 mora biti pozitivna konstanta pa slijedi da $f(n) \notin \Theta(n^3)$

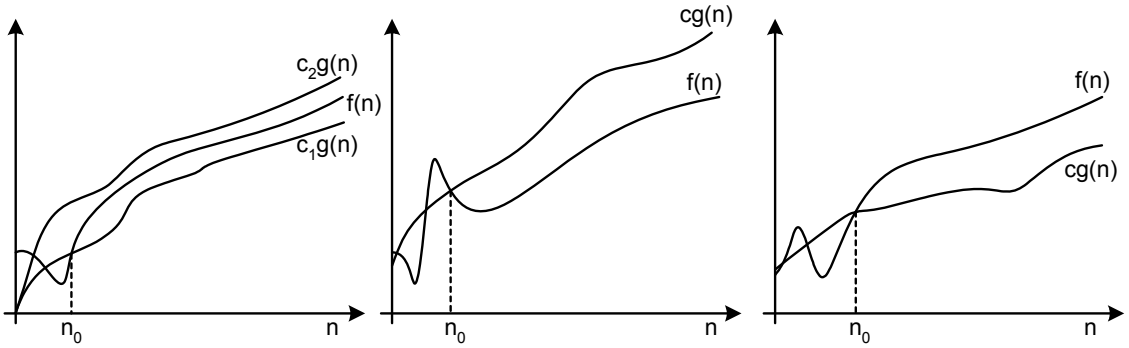
O-označavanje

Θ-označavanjem asimptotski ograničavamo funkciju i od gore i od dolje. Kada nas zanima samo gornja granica tada koristimo O-označavanje (notaciju).

Definicija: Za danu funkciju $g(n)$, definiramo $O(g(n))$ kao skup funkcija:

$$O(g(n)) = \{ f(n) : \text{ postoje pozitivne konstante c i } n_0 \text{ takve da je } 0 \leq f(n) \leq cg(n) \text{ za sve } n \geq n_0 \}$$

Primijetimo da $f(n) \in \Theta(g(n))$ nužno implicira $f(n) \in O(g(n))$, jer je Θ-označavanje strože od O-označavanja. Npr. $f(n) = 8n^2 + 4n - 5 \in \Theta(n^2)$, ali isto tako $f(n) \notin \Theta(n)$ kao i $f(n) \notin \Theta(n^3)$. Što se tiče O-označavanja[1] vrijedi $f(n) \in O(n^2)$ i $f(n) \in O(n^3)$ ali isto tako i $f(n) \notin O(n)$.



$f(n) = \Theta(g(n))$ $f(n) = O(g(n))$ $f(n) = \Omega(g(n))$

Slika: Ilustracija Θ-označavanja, O-označavanja i Ω-označavanja

Ω-označavanja

Slično kao što O-označavanja daje asimptotsku gornju granicu, tako Ω-označavanja daje asimptotsku donju granicu.

Definicija: Za danu funkciju $g(n)$, definiramo $\Omega(g(n))$ kao skup funkcija:

$$\Omega(g(n)) = \{ f(n) : \text{ postoje pozitivne konstante c i } n_0 \text{ takve da je } 0 \le cg(n) \le f(n) \text{ za sve } n \ge n_0 \}$$

Vrijedi da $f(n) = \Theta(g(n))$ nužno implicira $f(n) = \Omega(g(n))$. Ako ponovo promotrimo primjer $f(n) = 8n^2 + 4n - 5$ onda vrijedi $f(n) \in \Omega(n^2)$ i $f(n) \in \Omega(n)$ ali isto tako i $f(n) \notin \Omega(n^3)$.

Da rezimiramo: s O-označavanjem označavamo asimptotsku gornju granicu, a sa Ω-označavanja asimptotsku donju granicu. Može se lako pokazati da $f(n) \in \Theta(g(n))$ vrijedi onda i samo onda ako je $f(n) \in O(g(n))$ i $f(n) \in \Omega(g(n))$.

o-označavanje i ω-označavanje

Spomenimo (bez definicije) još dva označavanja koja koristimo. Asimptotska gornja granica O-označavanja može i ne mora biti asimptotski čvrsta. Granica $2n^2 = O(n^2)$ jeste asimptotski čvrsta, ali granica $2n = O(n^2)$ nije. o-označavanje koristimo za gornju granicu koja nije asimptotski čvrsta. Na primjer, $2n = o(n^2)$, ali $2n^2 \neq o(n^2)$.

[1] Ponekad se u literaturi O-označavanje koristi za definiciju asimptotski čvrste gornje granice, kao što je kod Θ-označavanja granica asimptotski čvrsta. Mi ćemo O-označavanje koristiti, kao što to u literaturi i jeste većinom slučaj, za označavanje asimptotske gornje granice neovisno o tome da li je čvrsta ili ne.

ω-označavanje je prema Ω-označavanja isto što i o-označavanje prema O-označavanje. Na primjer, $2n^2 = \omega(n)$, ali $2n^2 \neq \omega(n^2)$.

Kažemo da je funkcija $f(n)$ asimptotski manja od funkcije $g(n)$ ako je $f(n) = o(g(n))$, te da je funkcija $f(n)$ asimptotski veća od funkcije $g(n)$ ako $f(n) = \omega(g(n))$.

Ograničeno pravilo

U najvećem broju slučajeva može se bitno jednostavnije pokazati da je neka funkcija asimptotski ograničena. Za to nam služi tzv. ograničeno pravilo.

Ograničeno pravilo za Θ-označavanje: Za danu pozitivnu funkciju $f(n)$ i $g(n)$, ako je

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

gdje je c konstanta za koju vrijedi $c > 0$ (striktno pozitivna, ali ne beskonačna), tada vrijedi $f(n) \in \Theta(g(n))$.

Ograničeno pravilo za O-označavanje: Za danu pozitivnu funkciju $f(n)$ i $g(n)$, ako je

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

gdje je c konstanta za koju vrijedi $c \geq 0$ (ne negativna, i ne beskonačna), tada vrijedi $f(n) \in O(g(n))$.

Ograničeno pravilo za Ω-označavanje: Za danu pozitivnu funkciju $f(n)$ i $g(n)$, ako je

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$$

dakle rezultat je striktno pozitivan ili beskonačan, tada vrijedi $f(n) \in \Omega(g(n))$.

Ograničeno pravilo može se primijeniti gotovo uvijek i najčešće je jednostavnije primijeniti nego formalnu definiciju. Izuzetci na koje se ne može primijeniti ograničeno pravilo su uglavnom složene funkcije, kakve se gotovo nikada ne javljaju kod proračuna vremena izvršavanja algoritma (npr. funkcija $f(n) = n^{1+\sin n}$). Kod računanja pomoću ograničenog pravila od koristi može biti L'Hopitalovo pravilo – podsjetite se kako glasi.

Primjer primjene ograničenog pravila: Polinomi

Neka je $f(n) = 3n^3 - 5n^2 - 2n + 9$. Pokažimo primjenom ograničenog pravila da je $f(n) \in \Theta(n^3)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^3 - 5n^2 - 2n + 9}{n^3} = \lim_{n\to\infty} \left(3 - \frac{5}{n} - \frac{2}{n^2} + \frac{9}{n^3} \right) = 3$$

Kako je 3 striktno pozitivna konstanta, primjenom ograničenog pravila slijedi $f(n) \in \Theta(n^3)$. Ovaj zaključak može se lako generalizirati za polinom bilo kojeg stupnja.

Neka je $p(n)$ polinom k-tog stupnja $p(n) = \sum_{i=0}^{k} a_i n^i$ pri čemu je $a_k > 0$, tada vrijedi $p(n) \in \Theta(n^k)$.

Eksponencijalne i logaritamske funkcije

Eksponencijalne i logaritamske funkcije su vrlo važne u analiziranju algoritama pa ćemo se kratko osvrnuti na njihov asimptotski rast, kao i odnos prema asimptotskom rastu polinoma.

Neka su a, b i c pozitivne konstante te $a > 1$. Vrijedi:

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0 \qquad \lim_{n\to\infty} \frac{(\mathrm{ld}\, n)^b}{n^c} = 0$$

Ova dva izraza mogu se lako dokazati primjenom L'Hopitalova pravila. Iz prvog izraza slijedi važan zaključak da polinomi uvijek asimptotski rastu sporije nego eksponencijalna funkcija čija je baza veća od 1. Na primjer:

$$n^{100} \in O(2^n)$$

To je razlog zbog kojeg ćemo uvijek nastojati izbjeći algoritme s eksponencijalnim vremenom izvršavanja. Iz drugog izraza slijedi da logaritamska funkcija raste sporije od bilo kojeg polinoma. Na primjer:

$$(\mathrm{ld}\, n)^{100} \in O(n)$$

To je razlog zbog kojeg ćemo uvijek radije upotrijebiti bilo koliko dodatnih logaritama, ako to znači izbjegavanje dodatne potencije.

Naravno, pri tome treba biti oprezan i realan. Ovi zaključci vrijede u asimptotskom slučaju, dakle za dovoljno veliki n. Primjerice, prvi n za koji je ispunjeno $(\mathrm{ld}\, n)^{100} < n$ je veći od 2^{1000} što je broj ulaznih podataka koji će se rijetko javiti u praksi. Međutim, za dovoljno male potencije logaritma stvar postaje realna. Primjerice $(\mathrm{ld}\, n)^3 < n$ je ispunjeno već za $n > 1000$.

Tipična asimptotska vremena izvršavanja

U algoritmima se uglavnom javlja nekoliko tipičnih asimptotskih vremena izvršavanja. Ovdje je dan kratki komentar o tome što pojedina asimptotska vremena u praksi znače.

$\Theta(1)$ - Konstantno vrijeme. Od ovoga ne može bolje.

$\Theta(\log n)$ - Ovo je vrijeme tipično za efikasne strukture podataka koje rade u jednom pristupu. Također, to je vrijeme koje je potrebno da se pronađe objekt u uređenoj (sortiranoj) listi dužine n (pomoću binarnog pretraživanja).

$\Theta(n)$ - U načelu, ovo je najbrže što neki algoritam može raditi, jer je samo vrijeme učitavanja podataka jednako $\Theta(n)$.

$\Theta(n \log n)$ - Ovo je vrijeme izvršavanja najboljih algoritma za sortiranje (većina problema zahtijeva sortiranje ulaznih podataka).

$\Theta(n^2)$, $\Theta(n^3)$, ... - Polinomsko vrijeme. Ovakvo vrijeme izvršavanja prihvatljivo je ili ako je eksponent mali ili ako broj ulaznih podataka nije prevelik ($n < 1000$).

$\Theta(2^n)$, $\Theta(3^n)$, ... – Eksponencijalno vrijeme. Prihvatljivo jedino ako je broj ulaznih podataka vrlo mali ($n < 50$) ili je to najgore moguće vrijeme koje će se pojaviti vrlo rijetko. Ako je to najgore moguće vrijeme tada bi dobro bilo napraviti analizu prosječnog vremena trajanja.

$\Theta(n!)$, $\Theta(n^n)$ - Prihvatljivo samo za vrlo mali broj ulaznih podataka ($n < 20$).

3. PODIJELI PA VLADAJ, SORTIRANJE SPAJANJEM (MERGESORT)

Podijeli pa vladaj

Ideju "podijeli pa vladaj"[2] u algoritmima interpretiramo na slijedeći način: podijeli ulazne podatke na manje dijelove, riješi problem na svakom od tih manjih dijelova te potom kombiniraj rješenja pojedinih dijelova u jedno zajedničko rješenje. Ali, kad podijelimo ulaz (problem) na manje dijelove, kako riješiti i takav, manji problem? Rješenje je u daljnjoj podjeli, odnosno primjeni pristupa podijeli pa vladaj. Podjelu završavamo kada preostanu tako mali dijelovi (najčešće jedan ili dva podatka) da je rješenje trivijalno.

Dakle, osnovni elementi rješavanja metodom podijeli pa vladaj su.

- Podijeli (Podijeli problem na više manjih potproblema),
- Vladaj (Riješi potprobleme daljnjom podjelom. Ako je potproblem dovoljno mali, rješenje je trivijalno ili se može jednostavno riješiti nekom od poznatih metoda.)
- Spoji (Udruži sva rješenja u jedno zajedničko rješenje početnog problema.)

Veliki broj računalnih problema (i ne samo računalnih) može se riješiti primjenom tehnike podijeli pa vladaj. Podijeli pa vladaj algoritmi su tipično rekurzivni, jer dio tehnike "vladaj" uključuje korištenje te iste tehnike na manjim potproblemima.

MergeSort (Sortiranje spajanjem)

Prvi primjer primjene tehnike podijeli pa vladaj koji ćemo obraditi je algoritam za sortiranje MergeSort. To je sasvim sigurno i najpoznatiji takav primjer.

Neka je dan ulazni niz A od n brojeva, koji su pohranjeni u polje $A[1..n]$. Naš cilj je dobiti takvo polje u kojem će n ulaznih brojeva biti posloženo po rastućim vrijednostima. To ćemo, upotrebom tehnike podijeli pa vladaj, napraviti na slijedeći način:

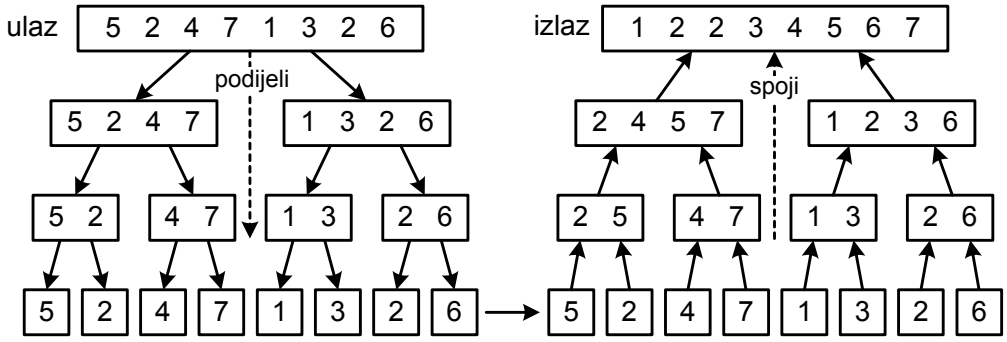
> Podijeli (Podijeli sve ulazne elemente na dvije podsekvence, svaka podsekvenca neka ima približno $n/2$ elemenata.)
>
> Vladaj (Sortiraj svaku od podsekvenci. Sortiranje izvršiti pomoću MergeSort-a rekurzivno na svaku od podsekvenci.)
>
> Spoji (Udruži dvije sortirane podsekvence u jednu zajedničku (sortiranu) listu.)

Proces dijeljenja završava kada podijelimo podsekvence tako da svaka sadrži samo jedan element. Sekvenca koja ima jedan element je trivijalno sortirana. Bitni dio algoritma je korak spajanja kojim se udružuju dvije sortirane liste u jednu, također sortiranu, listu.

---

[2] Naziv ove ideje, pa i sama ideja, potječu još od starih Rimljana koji su postupali na način "Divide et empera", "podijeli pa vladaj", kako bi lakše upravljali svojom velikom zemljom. Engleski naziv za algoritam o kojem govorimo je "Divide and conquer" ("Podijeli pa osvoji", "Podijeli pa pobijedi") što je zapravo parafraza stare rimske ideje koja se sasvim udomaćila u području algoritama. Naime, korektan engleski prijevod za "Divide et empera" bio bi "Divide and rule".

Na slici je prikazana grafička interpretacija algoritma. "Podijeli" dio algoritma prikazan je na lijevom dijelu slike. Desni dio odnosi se "vladaj i spoji" dio algoritma.



Slika: MergeSort: Ilustracija tehnike podijeli pa vladaj

Definirajmo najprije dio algoritma "podijeli", dakle lijevi dio slike. Pretpostavljamo da imamo proceduru koja spaja i da je to procedura Merge (koju ćemo kasnije definirati). Kako se algoritam rekurzivno poziva na podpolja, to ćemo algoritmom prenositi ne samo polje, nego i prvi i zadnji element podpolja koji se sortira. To znači da će poziv MergeSort (A,p,r) sortirati podpolje $A[p..r]$ te vratiti sortirani rezultat u istom podpolju.

Ako je $p = r$ to znači da je u polju samo jedan element koji treba sortirati, što je trivijalno rješivo, pa rezultat možemo odmah vratiti. U drugom slučaju (ako je $p < r$) postoje barem dva elementa što znači da primjenjujemo tehniku podijeli pa vladaj. Trebamo pronaći indeks q, koji se nalazi na sredini između p i r, dakle $q = (p + r)/2$ i to zaokruženo na najbliži niži cijeli broj. Potom dijelimo polje na dva podpolja $A[p..q]$ i $A[q+1..r]$. (Zašto ne bi bilo dobro podijeliti na $A[p..q-1]$ i $A[q..r]$?) Potom pozivamo MergeSort rekurzivno kako bi sortirali svako podpolje. Na kraju pozovemo proceduru, nazvali smo je Merge, koja će spojiti ova dva podpolja u jedno sortirano (pod)polje. Proceduru Merge ćemo kasnije definirati.

```
MergeSort (array A, int p, int r)
    if (p<r)                        //ispunjeno ako imamo barem dva elementa
        q=(p+r)/2
        MergeSort(A, p, q)          //sortiranje A[p..q]
        MergeSort(A, q+1, r)        //sortiranje A[q+1..r]
        Merge(A, p, q, r)           //spajanje
```

Cijelo polje $A = (A[1], A[2], ..., A[n])$ sortiramo pozivom MergeSort(A, 1, length[A]) gdje je $\text{length}[A] = n$.

Spajanje (Merging)

Sada je preostalo još da objasnimo proceduru za spajanje dviju sortiranih lista Merge(A, p, q, r). Pretpostavljamo da su i lijevo ($A[p..q]$) i desno ($A[q+1..r]$) podpolje već sortirani. Ta dva

podpolja spajamo tako da kopiramo elemente u privremeno polje koje ćemo nazvati B. Zbog jednostavnosti pretpostavit ćemo da su indeksi u polju B isti kao i indeksi u polju A tj. $B[p..r]$. Najprije definirajmo indekse i i j, koji pokazuju na trenutne elemente u svakom podpolju. Potom pomaknemo manji od ta dva elementa na slijedeću poziciju u polju B (to mjesto indeksirajmo s k). Slijedi povećavanje odgovarajućeg indeksa (dakle i ili j). Kada potrošimo sve elemente u nekom od podpolja, jednostavno kopiramo preostale elemente iz drugog podpolja u polje B. Na kraju cjelokupno polje B kopiramo u polje A.
Iz ovog opisa vidi se da algoritam MergeSort zahtijeva privremeno polje, što je jedan od njegovih nedostataka.

```
Merge(array A, int p, int q, int r)
    array B[p..r]
    i=k=p                                   //inicijaliziranje indeksa
    j=q+1
    while (i<=q and j<=r)                   //dok oba podpolja nisu prazna
        if (A[i]<=A[j])   B[k++]=A[i++]     //kopiraj iz lijevog podpolja
            else  B[k++]=A[j++]             //kopiraj iz desnog podpolja
    while (i<=q)  B[k++]=A[i++]             //kopiraj sve što je preostalo u B
    while (j<=r)  B[k++]=A[j++]             //kopiraj sve što je preostalo u B
    for i=p to r do A[i]=B[i]               //kopiraj B u A
```

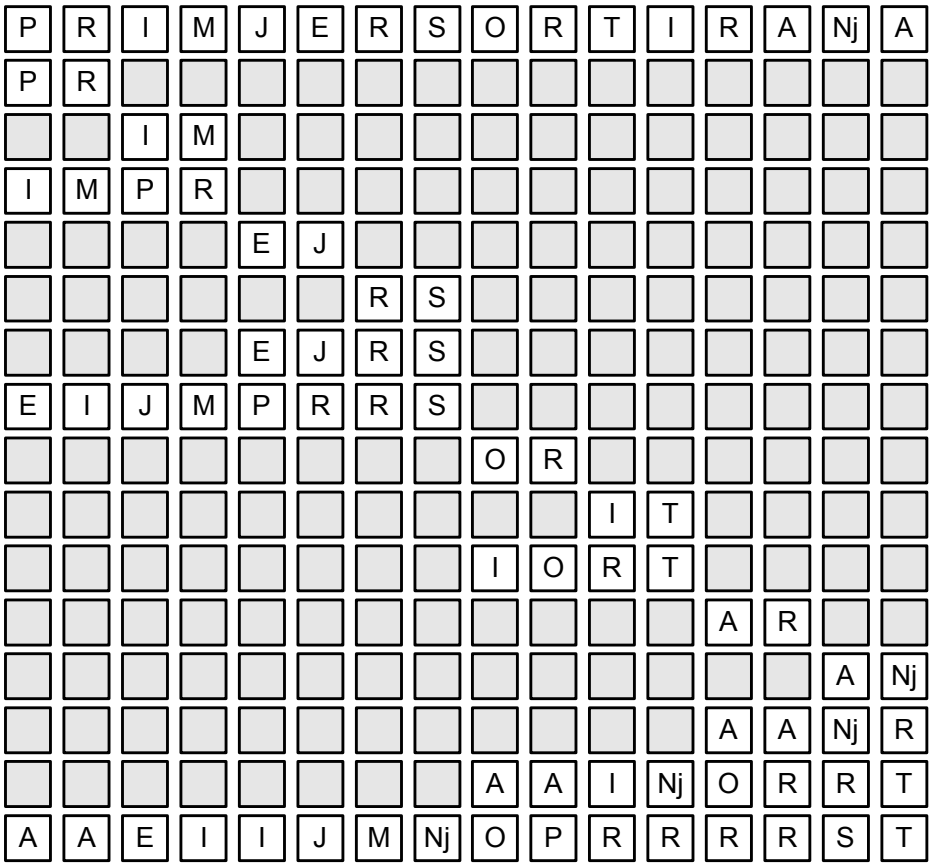
Vrijeme izvršavanja algoritma može se, u odnosu na gore napisani pseudokod, poboljšati primjenom nekoliko «trikova». Naravno, ti «trikovi» ne mijenjaju asimptotsko vrijeme izvršavanja već samo poboljšavaju konstantni faktor. Ako bi poboljšali asimptotsko vrijeme izvršavanja to bi bio potpuno drugi algoritam.

Prvo poboljšanje odnosi se na stalno kopiranje iz A u B i natrag iz B u A. U praksi se to nepotrebno kopiranje rješava korištenjem dvaju polja, jednake dužine. Na neparnoj razini rekurzije podpolja spajamo tako da prebacujemo iz A u B, a na parnoj razini tako da prebacujemo iz B u A. Ako rekurzija ima neparan broj razina tada je na kraju potrebno napraviti jedno dodatno prebacivanje iz B u A, što je još uvijek brže nego da se prebacuje na svakoj razini.

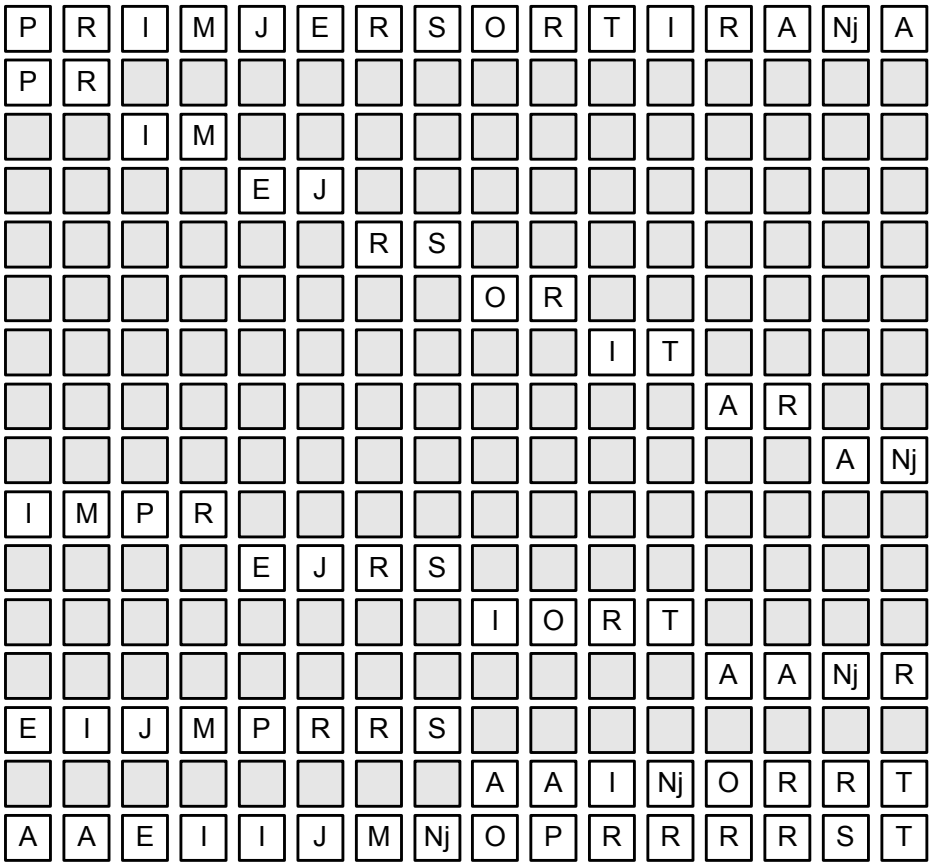
Još jedan od načina na koji se može ubrzati algoritam i poboljšati konstanti faktor je da se dijeljenje ne vrši sve do kraja. Dakle, ne dijelimo sve dok podsekvenca ima jedan element, nego se zaustavimo prije nego podsekvenca postane manja od nekog broja elementa, primjerice 20. Tada na takvoj podsekvenci primijenimo jednostavni $\Theta(n^2)$ algoritam. Često se čak i algoritam grube sile izvršava brže za mali broj podataka u odnosu na MergeSort. Važno je uočiti da se u tom slučaju algoritam grube sile uvijek izvršava na podsekvencama od najviše 20 elementa, pa je stoga vrijeme izvršavanja tog dijela algoritma konstantno, tj. $\Theta(20^2) = \Theta(1)$. Slijedi da uključivanje algoritma grube sile (ili nekog drugog) neće promijeniti ukupno asimptotsko vrijeme izvršavanja algoritma.

Na slici je dan primjer kako se sortira pomoću rekurzivnog MergeSort-a. Može se pokazati da svaki rekurzivni program ima i svoju nerekurzivnu varijantu. Kako bi izgledalo sortiranje

pomoću nerekurzivnog MergeSort-a također je ilustrirano na slici. Pošto je u ovom slučaju broj ulaznih podataka jednak potenciji broja dva, sortiranje se vrši na sličan način i za rekurzivni i za nerekurzivni Mergesort; razlika je "samo" u redoslijedu. Međutim, nerekurzivna i rekurzivna metoda nisu iste, što se najbolje može vidjeti u slučaju kada broj ulaznih podataka nije blizak potenciji broja dva. Naime, kod rekurzivnog algoritma zadnje spajanje je uvijek spajanje dva niza koja su jednako velika ili se razlikuju samo za jedan. Kod nerekurzivnog algoritma to nije slučaj, jer broj elemenata može biti bitno različit.



Slika: Primjer MergeSort-a (rekurzivnog)



Slika: Primjer MergeSort-a (nerekurzivnog)

Analiza vremena izvršavanja algoritma MergeSort

Promotrimo prvo proceduru Merge (A, p, q, r). Neka je $n = r - p + 1$ ukupna dužina i lijevog i desnog podpolja. Algoritam se sastoji od 4 petlje od kojih niti jedna nije ugniježđena. Može se lako vidjeti da će se svaka petlja izvršiti maksimalno n puta. A ako se bolje promotri onda se može vidjeti da će se sve petlje zajedno izvršit n puta, jer svako izvršavanje kopira jedan element u B, a u B ima mjesta samo za n elementa. To znači da je vrijeme izvršavanja Merge algoritma za n elementa jednako $\Theta(n)$. Napišimo to, bez asimptotskog označavanja, jednostavno kao n (poslije ćemo vidjeti zašto).

Kako sada izračunati vrijeme trajanja cijelog algoritma MergeSort? Za to ćemo koristiti rekurziju, tj. funkciju koja je definirana rekurzivno u odnosu na samu sebe. Rekurzija je, za danu vrijednost n, uvijek definirana preko vrijednosti koje su striktno manje od n. Time se izbjegava moguća situacija beskonačne petlje. Nadalje, rekurzija uvijek ima neku početnu vrijednost (npr. za $n = 1$), koja je eksplicitno definirana.

Neka je $T(n)$ najgore moguće vrijeme izvršavanja MergeSort algoritma za polje dužine n. Pri tome je sasvim sve jedno što ćemo računati: broj linija pseudokoda, broj usporedbi, broj pristupa polju, jer će se razlikovati samo u konstanti.

Najprije uočimo da ako polje ima samo jedan element, da će vrijeme izvršavanja biti konstantno. Kako zanemarujemo sve konstante faktore možemo napisati $T(n) = 1$ za $n = 1$. Kada pozivamo MergeSort(A, p, r) za polje gdje je $n > 1$ ($n = r - p + 1$) algoritam prvo

izračuna $q = \lfloor (p + r)/2 \rfloor$. Time smo definirali podpolje $A[p..q]$ koje ima $q - p + 1$ elemenata (može se lako pokazati da je to jednako $\lceil n/2 \rceil$). Stoga drugo podpolje $A[q + 1..r]$ ima $\lfloor n/2 \rfloor$ elemenata. Koliko vremena treba za sortiranje lijevog podpolja? To ne znamo ali, budući je $\lceil n/2 \rceil < n$ za $n > 1$ to vrijeme možemo napisati kao $T(\lceil n/2 \rceil)$. Slično možemo napisati i vrijeme potrebno za sortiranje desnog potpolja kao $T(\lfloor n/2 \rfloor)$. Da bi spojili dva podpolja treba nam, kako smo to već izračunali, n vremena. Na kraju možemo napisati

$$T(n) = \begin{cases} 1 & \text{za } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{za ostale} \end{cases}$$

Napomena: Izraz $\lceil x \rceil$ označava najmanji cijeli broj veći ili jednak x. Izraz $\lfloor x \rfloor$ označava najveći cijeli broj manji ili jednak x.

U nastavku ćemo vidjeti na koje sve načine se rekurzivne formule mogu riješiti.

4. REKURZIJA

Tehnika podijeli pa vladaj je jedna od temeljnih tehnika kojom se dizajniraju algoritmi. Kako pristup podijeli pa vladaj uglavnom vodi prema rješenju koje je rekurzivno, važno nam je imati takav matematički aparat pomoću kojega možemo rješavati rekurziju, bilo egzaktno bilo asimptotski.

MergeSort

Kako smo već vidjeli, rekurzivna formula po kojoj možemo izračunati vrijeme izvršavanja MergeSort je

$$T(n) = \begin{cases} 1 & \text{za } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{za ostale} \end{cases}$$

Dakle, ako je broj ulaznih podataka jednak 1, tada je ukupno vrijeme sortiranja konstantno tj. $\Theta(1)$. Ako je $n > 1$, tada moramo rekurzivno sortirati dva podpolja, jedan veličine $\lceil n/2 \rceil$ i drugi veličine $\lfloor n/2 \rfloor$. Vrijeme izvršavanja nerekurzivnog dijela algoritma je $\Theta(n)$.

Traženje uzorka

Razvijmo sada gornju rekurziju

$$\begin{aligned}
&T(1) = 1 && n = 1 \\
&T(2) = T(1) + T(1) + 2 = 1 + 1 + 2 = 4 && n = 2 \\
&T(3) = T(2) + T(1) + 3 = 4 + 1 + 3 = 8 && n = 3 \\
&T(4) = T(2) + T(2) + 4 = 4 + 4 + 4 = 12 && n = 4 \\
&T(5) = T(3) + T(2) + 5 = 8 + 4 + 5 = 17 && n = 5 \\
&\quad \dots \\
&T(8) = T(4) + T(4) + 8 = 12 + 12 + 8 = 32 && n = 8 \\
&\quad \dots \\
&T(16) = T(8) + T(8) + 16 = 32 + 32 + 16 = 80 && n = 16 \\
&\quad \dots \\
&T(32) = T(16) + T(16) + 32 = 80 + 80 + 32 = 192 && n = 32
\end{aligned}$$

Pravilo po kojem se ponašaju vremena trajanja, na prvi pogled nije lako odrediti. Kako rekurzijom dijelimo broj podataka uvijek s dva, za očekivati je da će za taj slučaj pravilo biti lakše pronaći. Razmotrimo, dakle, omjer $T(n)/n$ za potencije broja dva:

$$T(1)/1 = 1$$
$$T(2)/2 = 2$$
$$T(4)/4 = 3$$
$$T(8)/8 = 4$$
$$T(16)/16 = 5$$
$$T(32)/32 = 6$$

Ovo sugerira da bi za potenciju broja 2 rješenje bilo oblika $T(n)/n = \mathrm{ld}n + 1$, odnosno $T(n) = n \cdot \mathrm{ld}n + n$, što bi značilo da je $\Theta(n \log n)$. To nije dokaz, ali nam daje korisan početni korak. Uočite da unutar $\Theta(\cdot)$ piše $\log\cdot$ umjesto $\mathrm{ld}\cdot$. Razlog je što baza logaritma u asimptotskom slučaju nije važna. Vrijedi

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Kako su a i b konstante to je i $\log_a b$ konstanta. Slijedi da se $\log_b n$ i $\log_a n$ razlikuju samo za konstantu pa se unutar $\Theta(\cdot)$ ne trebamo ni praviti razliku među njima.

Zaokruživanje koje radimo kod rekurzija (tj. $\lceil x \rceil$ i $\lfloor x \rfloor$) neugodni su i nespretni za rad. Stoga, kada god je to razumno, nastojimo pojednostaviti račun jednostavnim izbacivanjem zaokruživanja. Time je rješenje koje se dobije točno samo za dio (ali ipak beskonačan) broja ulaznih podataka. Ako se algoritam ne razlikuje bitno za potencije broja dva u odnosu na ostale, onda će dobiveno rješenje biti dobro za sve n. Ako eliminiramo zaokruživanje, tada je vrijeme trajanja MeregeSort-a jednako

$$T(n) = \begin{cases} 1 & \text{za } n = 1 \\ 2 \cdot T(n/2) + n & \text{za ostale} \end{cases}$$

Provjera pomoću indukcije

Rješenje koje smo našli traženjem uzorka potrebno je dokazati. Za to će nam poslužiti indukcija i to stroga. Naime, ne možemo koristiti uobičajeni dokaz: ako vrijedi za n dokaži da vrijedi za n+1, jer je n ograničen samo na potencije broja 2, što n+1 općenito nije.

Tvrdnja: Za sve $n \geq 1$, gdje je n potencija broja 2, vrijedi $T(n) = n \cdot \mathrm{ld}n + n$.

Dokaz:

Početni uvjet: (n=1) U ovom slučaju je $T(1) = 1 \cdot \mathrm{ld}1 + 1 = 1$ što odgovara definiciji koja kaže da je $T(1) = 1$.

Korak indukcije: Neka je $n > 1$, pretpostavimo da izraz $T(n') = n' \cdot \mathrm{ld}n' + n'$ vrijedi za sve $n' < n$. Želimo dokazati da u tom slučaju formula vrijedi i za n. Da bi to dokazali, izrazit ćemo $T(n)$ preko manjih vrijednosti za što ćemo iskoristiti definiciju

$$T(n) = 2 \cdot T(n/2) + n$$

Sad imamo da je $n/2 < n$, što znači da možemo primijeniti hipotezu (tvrdnju) koja kaže da je $T(n/2) = n/2 \cdot ldn/2 + n/2$. Uvrštavanjem ovog izraza slijedi

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n \\ &= 2(n/2 \cdot ld(n/2) + n/2) + n \\ &= n \cdot ld(n/2) + n + n \\ &= n \cdot (ldn - ld2) + 2 \cdot n \\ &= n \cdot ldn - n + 2 \cdot n \\ &= n \cdot ldn + n \end{aligned}$$

čime smo dokazali da je tvrdnja točna.

Metoda iteracije

Metoda pogađanja rješenja i dokazivanja pomoću indukcije je korisna sve dok je rekurzija toliko jednostavna da se rješenje može lako pogoditi. Ali ako je rekurzija složena, rješenje može biti formula koja nije jednostavna. U tom slučaju može pomoći metoda iteracije. Temeljna ideja je da se rekurzija pretvori u sumu. U načelu, sume je lakše rješavati nego rekurzije.

Počinjemo tako da razvijamo definiciju dok ne vidimo uzorak. Dakle, u definiciji $T(n) = 2 \cdot T(n/2) + n$, zamijenimo $T(n/2)$ s $2 \cdot T(n/4) + n/2$, potom dalje mijenjamo $T(n/4)$ s odgovarajućim izrazom itd. sve dok ne uočimo uzorak.

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + n \\ &= 2 \cdot (2 \cdot T(n/4) + n/2) + n = 4 \cdot T(n/4) + 2 \cdot n \\ &= 4 \cdot (2 \cdot T(n/8) + n/4) + 2 \cdot n = 8 \cdot T(n/8) + 3 \cdot n \\ &= 8 \cdot (2 \cdot T(n/16) + n/8) + 3 \cdot n = 16 \cdot T(n/16) + 4 \cdot n \\ &= \ldots \end{aligned}$$

Već sada se može uočiti uzorak

$$T(n) = 2^k \cdot T(n/2^k) + k \cdot n$$

Sada smo napravili važan korak, ali još uvijek nemamo rješenje. Za rješenje je potrebno eliminirati $T(\cdot)$ s desne strane izraza. Kako to napraviti? Znamo da je $T(1) = 1$. Stoga ćemo odabrati takav k da izraz $n/2^k$ bude uvijek jednak 1. Iz $n/2^k = 1$ slijedi da je $n = 2^k$, odnosno $k = ldn$. Ako to uvrstimo u gornji izraz, slijedi

$$\begin{aligned} T(n) &= 2^{ldn} \cdot T(n/2^{ldn}) + (ldn) \cdot n \\ &= n \cdot T(n/n) + n \cdot ldn \\ &= n \cdot T(1) + n \cdot ldn \\ &= n + n \cdot ldn \end{aligned}$$

Dakle, došli smo do istog rješenja. Na prvi pogled našli smo rješenje samo za jedan slučaj. Međutim, to nije točno jer smo s $n = 2^k$ našli rješenje za sve slučajeve kada je n potencija broja 2, a i ovom slučaju može se pokazati da je to dobro rješenje i za sve n.

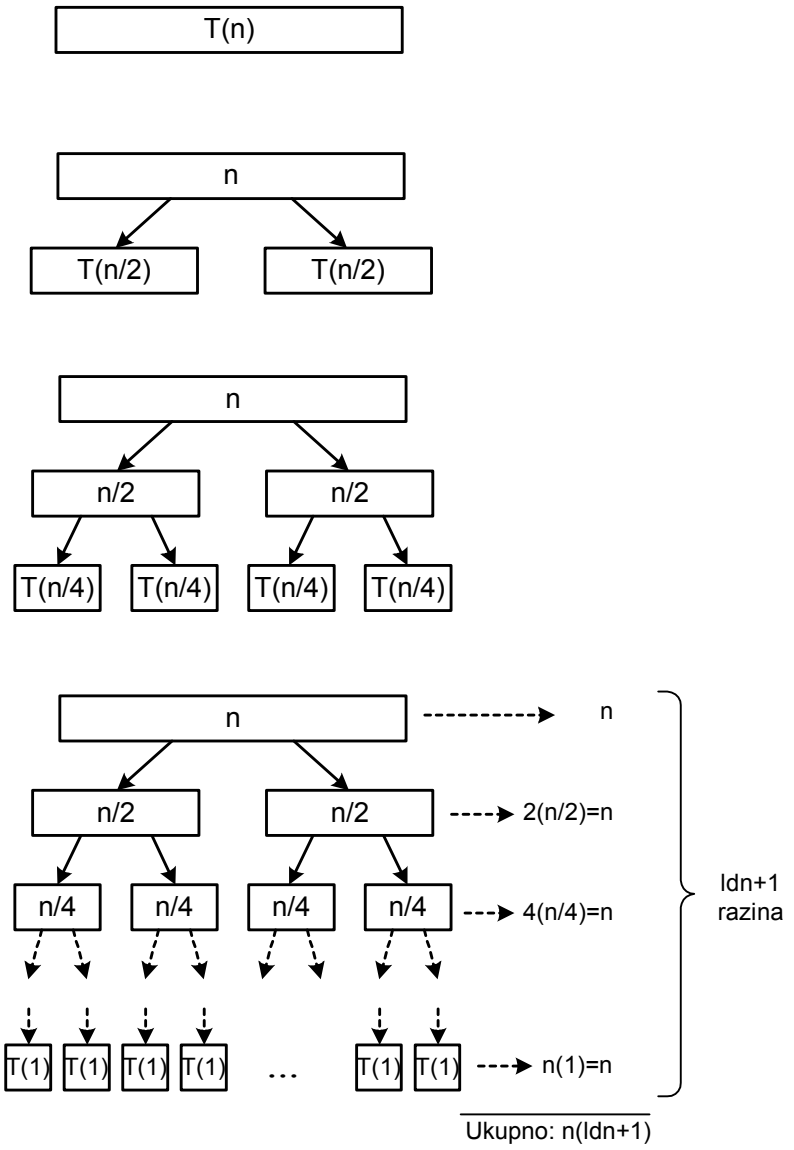
Metoda rekurzivnog stabla

Metoda rekurzivnog stabla naročito je pogodna za opisivanje vremena trajanja algoritama podijeli pa vladaj. Osim toga, to je zgodan načina da se vizualizira problem. Naime, vrlo je lako u radu sa simbolima izgubiti iz vida što se zapravo dešava. Ovom metodom se bilo koja rekurzija može prikazati kao stablo.

Promotrimo ponovo primjer MergeSort-a i pojednostavljene formule (bez zaokruživanja)

$$T(n) = \begin{cases} 1 & \text{za } n = 1 \\ 2 \cdot T(n/2) + n & \text{za ostale} \end{cases}$$

Podsjetimo da je vrijeme koje potrebno da bi se udružile dva polja dužine n/2 u jedno polje dužine n, jednako $\Theta(n)$ što smo jednostavno pisali kao n.



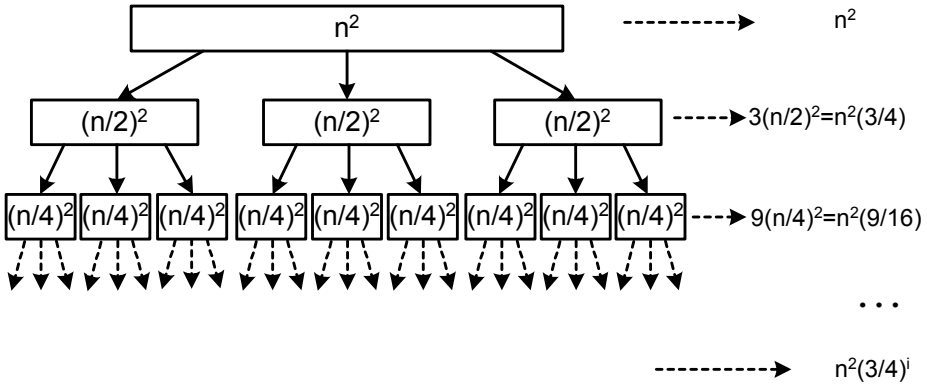
Slika: Rekurzivno stablo za MergeSort algoritam

Vrijeme izvršavanja na najvišoj razini je n. Na drugoj razini imamo dva udruživanja, od kojih svaki treba vrijeme n/2, što je ukupno $2(n/2) = n$. Na trećoj razini imamo 4 udruživanja s vremenom trajanja n/4 što je ukupno $4(n/4) = n$. Slično je i za sve ostale razine. Kako razina imamo točno $ldn + 1$ $(0, 1, 2, ..., ldn)$, a svaka razina doprinosi vremenu izvršavanja s n, to je ukupno vrijeme izvršavanja jednako $n(ldn + 1)$. Tako smo dobili isto rješenje kao i metodom iteracije, ali na način koji je vizualno jasniji.

Primijenimo sada metodu rekurzivnog stabla i na drugi primjer. Neka je zadana rekurzija

$$T(n) = \begin{cases} 1 & \text{za } n = 1 \\ 3 \cdot T(n/2) + n^2 & \text{za ostale} \end{cases}$$

Nacrtat ćemo odmah konačno rekurzivno stablo.



Slika: Rekurzivno stablo – drugi primjer

Ovo rekurzivno stablo vodi prema sumi

$$T(n) = n^2 \sum_{i=0}^{?} \left(\frac{3}{4}\right)^i$$

Gornju granicu u sumi još nismo odredili. Ako želimo samo asimptotsko rješenje onda je nije niti potrebno odrediti. Naime, suma u prethodnom izrazu je geometrijski red s bazom $3/4$ (dakle manje od 1), što znači da suma konvergira prema konstanti različitoj od nula. Slijedi da je vrijeme izvršavanja $\Theta(n^2)$.

Da bi dobili točan rezultat potrebno je odrediti gornju granicu sume, odnosno broj razina. Na svakoj razini broj podataka dijelimo sa dva (n, n/2, n/4, ...) pa će na i-toj razini broj podataka biti $n/2^i$. Rekurzivno stablo završi kada smo dostigli granični uvjet, tj. kada je broj podataka na razini jednak 1. Slijedi da je $n/2^i = 1$, tj. $i = ldn$. To je ujedno i gornja granica sume. Uočite da je ukupan broj razina jednak $ldn + 1$ (0, 1, 2, ..., ldn). Sada možemo izračunati sumu

$$T(n) = n^2 \sum_{i=0}^{\mathrm{ld}\, n} \left(\frac{3}{4}\right)^i = n^2 \frac{\left(\frac{3}{4}\right)^{\mathrm{ld}\, n+1} - 1}{\frac{3}{4} - 1} = 4 \cdot n^2 \cdot \left(1 - \left(\frac{3}{4}\right)^{\mathrm{ld}\, n+1}\right)$$

$$= 4 \cdot n^2 \cdot \left(1 - \frac{3}{4} \cdot \left(\frac{3}{4}\right)^{\mathrm{ld}\, n}\right) = 4 \cdot n^2 \cdot \left(1 - \frac{3}{4} \cdot (n)^{\mathrm{ld}\frac{3}{4}}\right)$$

$$= 4 \cdot n^2 \cdot \left(1 - \frac{3}{4} \cdot (n)^{\mathrm{ld}3 - \mathrm{ld}4}\right) = 4 \cdot n^2 \cdot \left(1 - \frac{3}{4} \cdot (n)^{\mathrm{ld}3 - 2}\right)$$

$$= 4 \cdot n^2 \cdot \left(1 - \frac{3}{4} \cdot \frac{(n)^{\mathrm{ld}3}}{n^2}\right)$$

$$= 4 \cdot n^2 - 3 \cdot n^{\mathrm{ld}3}$$

(Upotrijebili smo formulu $x^{\log_a y} = y^{\log_a x}$.)

S obzirom da je $\mathrm{ld}3 = 1.58$ slijedi da je vrijeme izvršavanja $T(n)$ asimptotski jednako $\Theta(n^2)$.

Master Theorem

Master theorem daje vrlo jednostavnu metodu ("recept") kojim se mogu riješiti rekurzije oblika

$$T(n) = aT(n/b) + f(n)$$

gdje su $a \geq 1$ i $b > 1$ konstante, a $f(n)$ je asimptotski pozitivna funkcija. Glavni teorem ima tri slučaja pomoću kojih možemo odrediti rješenja velikog broja rekurzija i to vrlo lako. Opći oblik rekurzije $T(n) = aT(n/b) + f(n)$ opisuje vrijeme izvršavanja algoritma koji dijeli problem veličine n na a podproblema, svaki veličine n/b, gdje su n i b pozitivne konstante, a podprobleme rješava rekurzivno i to za vrijeme $T(n/b)$. Vrijeme koje je potrebno za podjelu i udruživanje opisano je funkcijom $f(n)$. Npr. rekurzija MergeSort-a ima $a = 2$, $b = 2$ i $f(n) = \Theta(n) = n$.

Napomena: I ovdje smo zanemarili zaokruživanje, što znači da će dobiveni rezultat vrijediti samo za n koji je potencija b, tj. kada je n/b cijeli broj. Međutim, može se pokazati da je asimptotsko ponašanje rekurzije kada se koristi zaokruživanje isto kao i bez zaokruživanja. Početni uvjet tj. $T(1)$, može biti bilo koja konstanta.

Teorem

Neka su $a \geq 1$ i $b > 1$ konstante, neka je $f(n)$ funkcija, te neka je $T(n)$ rekurzija definirana na nenegativnim cijelim brojevima

$$T(n) = aT(n/b) + f(n)$$

Tada $T(n)$ možemo asimptotski ograničiti na slijedeći način

1. Ako je $f(n) = O(n^{\log_b a - \varepsilon})$ za neku konstantu $\varepsilon > 0$, tada je $T(n) = \Theta(n^{\log_b a})$.
2. Ako je $f(n) = \Theta(n^{\log_b a})$, tada je $T(n) = \Theta(n^{\log_b a} \mathrm{ld}\, n)$.

3. Ako je $f(n)=\Omega(n^{\log_b a+\varepsilon})$ za konstantu $\varepsilon>0$, i ako je $af(n/b)\le cf(n)$ za konstantu $c<1$ i sve dovoljno velike n-ove, tada je $T(n)=\Theta(f(n))$.

Pojednostavljeni Master teorem

Neka su $a\ge 1$ i $b>1$ konstante, neka je $T(n)$ rekurzija definirana na nenegativnim cijelim brojevima

$$T(n)=aT(n/b)+n^k$$

Tada $T(n)$ možemo asimptotski ograničiti na slijedeći način

1. Ako je $a>b^k$, tada je $T(n)=\Theta(n^{\log_b a})$.
2. Ako je $a=b^k$, tada je $T(n)=\Theta(n^k \operatorname{ld} n)$.
3. Ako je $a<b^k$, tada je $T(n)=\Theta(n^k)$.

Iskoristimo Pojednostavljeni teorem za rješavanje MergeSort rekurzije za koji imamo da je $a=2$, $b=2$ i $k=1$. Vrijedi $a=b^k$ $(2=2^1)$ što znači da primjenjujemo 2. slučaj Pojednostavljenog teorema. Slijedi da je $T(n)=\Theta(n \operatorname{ld} n)$. Uočite da nismo dobili točno nego asimptotsko rješenje.

Razmotrimo sada rekurziju $T(n)=3T(n/2)+n^2$ gdje imamo $a=3$, $b=2$ i $k=2$. Kako je $a<b^k$ $(3<2^2)$ slijedi da primjenjujemo 3. slučaj pa vrijedi $T(n)=\Theta(n^2)$.

I na kraju promotrimo i rekurziju $T(n)=4T(n/3)+n$ gdje je $a=4$, $b=3$ i $k=1$. Vrijedi $a>b^k$ $(4>3^1)$ pa primjenjujemo 1. slučaj odakle slijedi $T(n)=\Theta(n^{\log_3 4})\approx\Theta(n^{1.26})$. Iako, na prvi pogled, izgleda čudno imati necjelobrojni eksponent, to je dosta čest slučaj kod algoritama koje se temelje na tehnici podijeli pa vladaj.

Postoji dosta rekurzija koje se ne mogu riješiti pomoću Pojednostavljenog Master teorema pa čak niti pomoću Master teorema, npr.

$$T(n)=\begin{cases}1 & \text{za } n=1\\ 2T(n/2)+n\log n & \text{za ostale}\end{cases}$$

Rješenje ove rekurzije je $T(n)=\Theta(n\log^2 n)$, ali do rješenja ne možemo doći pomoću Master teorema. Međutim, rješenje možemo naći primjerice pomoću iterativne metode.

5. HEAP I HEAPSORT

Tree sort - Stablasto sortiranje

Prema Informatičkom rječniku (Miroslav Kiš) heap znači

1. dio memorije koji se koristi za pohranu podataka, a pristup je memoriji slučajan
2. binarno stablo u kojem je vrijednost u svakom čvoru veća ili jednaka vrijednostima djece tog čvora

U nekoliko slijedećih sati predmet proučavanja bit će algoritmi za sortiranje. I do sada smo obradili jedan od algoritama sortiranja (MergeSort). Razlog tako intenzivnog proučavanja algoritma za sortiranje je dvostruk.

Kao prvo, sortiranje je važan problem jer se sortiranje javlja u većini velikih softverskih sustava, bilo eksplicitno bilo implicitno. Stoga je dizajniranje efikasnih algoritama za sortiranje važno za ukupnu efikasnost takvih sustava.

Drugi razlog je pedagoški. Postoji puno algoritama za sortiranje, neki su brži neki sporiji, neki imaju željena svojstva neki nemaju. Stoga su zgodan način da se na jednom istom problemu, koji je vrlo jasan, prezentiraju različiti koncepti i ideje koje se koriste u algoritmima.

U problemu sortiranja imamo polje $A[1..n]$ od n brojeva, koje trebamo presložiti po veličini. U općem slučaju, A je polje koje sadrži blokove podataka. Iz tih blokova podataka biramo jedan podatak koji je vrijednost ključa (key value), na temelju kojeg će elementi biti sortirani. Vrijednost ključa ne mora nužno biti broj. To može biti bilo koji objekt koji je iz potpuno uređene domene. Potpuna uređenost znači da za bilo koja dva elementa x i y iz te domene vrijedi ili $x < y$ ili $x = y$ ili $x > y$.

Spori algoritmi za sortiranje

Postoji puno dobro-poznatih sporih algoritama za sortiranje.

BubbleSort (sortiranje u valovima, zamjenjivačko sortiranje /Kiš/): Prelazi preko polja. Kada god naiđeš na dva uzastopna elementa koji nisu posloženi po željenom redu, zamijeni ih. Ponavljaj postupak sve dok svi uzastopni elementi nisu posloženi po redu.

InsertionSort (sortiranje umetanjem): Sortirani elementi stavljaju se u novo polje, nazovimo ga A'. Novo polje je, dakle, u svakom trenutku sortirano. Trenutni element koji sortiramo $A[i]$, stavljamo na odgovarajuće mjesto u novom polju A', na način da sve veće elemente pomaknemo u desno za jedno mjesto čime smo napravili mjesto za novi element. Postupak ponavljamo dok u novo polje ne složimo sve elemente originalnog polja.

SelectionSort (sortiranje odabirom): Pronađi najmanji element i zamijeni ga s elementom koji je na prvom mjestu. Potom u ostatku pronađi najmanji element i zamjeni ga s drugim. Postupak ponavljaj dok ne dođeš do zadnjeg elementa.

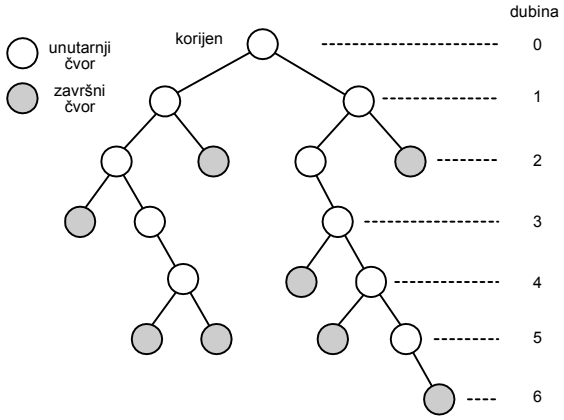
Općenito, imamo podpolje $A[1..i-1]$ koje sadrži $i-1$ najmanjih elementa i to sortiranih. Treba pronaći najmanji element u podpolju $A[i..n]$ i zamijeniti ga s elementom $A[i]$.

Ovo su sve algoritmi koje je vrlo lako implementirati, ali je njihovo izvršavanje (najgore vrijeme izvršavanja) jednako $\Theta(n^2)$. Već smo vidjeli da postoji algoritam, MergeSort, s $\Theta(n \log n)$. Sada ćemo vidjeti još dva takva algoritma, HeapSort i QuickSort.

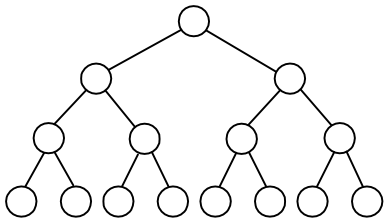
Heap struktura podataka

HeapSort algoritam se temelji na strukturi podataka koja se zove heap. Heap je struktura podataka koju možemo predstaviti kao skoro potpuno binarno stablo. Koncept heap strukture podataka objasnit ćemo preko binarnog stabla, a poslije ćemo vidjeti da se heap struktura podataka može pohraniti i u obliku polja.

Pod binarnim stablom mislimo na strukturu podataka koja je ili prazna ili se sastoji od tri dijela: ishodišni čvor (korijen - root node), lijevo podstablo (subtree) i desno podstablo. Lijevo i desno podstablo su i sami binarna stabla, a nazivaju se lijevo i desno dijete (left child, right child). Ako su i lijevo i desno dijete nekog čvora prazni, tada se taj čvor naziva završni čvor (leaf node). Čvor koji nije završni naziva se unutarnji čvor (internal node). Općenito, svaki čvor osim završnog ima dijete, te svaki čvor osim korijena ima roditelja.



Slika: Binarno stablo



Slika: Potpuno binarno stablo

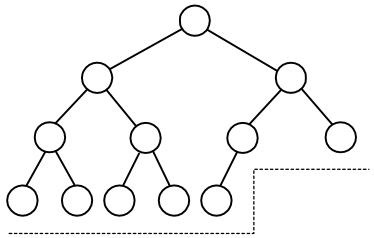
Dubina (depth) čvora u binarnom stablu je njegova udaljenost od korijena (ishodišta). Korijen ima dubinu 0, njegovo dijete 1, dijete njegovog djeteta 2, itd. Visina (height) binarnog stabla jednaka je njegovoj maksimalnoj dubini. Za binarno stablo kažemo da je potpuno (complete) ako svi unutarnji čvorovi imaju dvoje (neprazne) djece i ako svi završni čvorovi imaju istu dubinu. Važna činjenica potpunog binarnog stabla je da potpuno binarno stablo visine h ima ukupno n čvorova gdje je

$$n = 1 + 2^1 + 2^2 + \ldots + 2^h = \sum_{i=0}^{h} 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$$

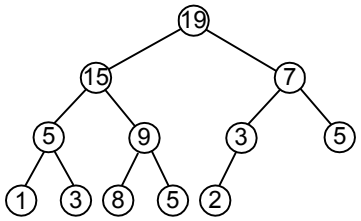
Ako sada iz jednadžbe $n = 2^{h+1} - 1$ izrazimo h kao funkciju n slijedi da je

$$h = \mathrm{ld}(n+1) - 1 = \Theta(\log n)$$

Heap je takva struktura podataka koje se može predstaviti lijevim potpunim binarnim stablom (left-complete binary tree). To znači da su sve razine stabla pune, osim donje razine, koja se puni od lijeva prema desno. Osim toga, u heap strukturi podataka ključevi su pohranjeni u tzv. heap rasporedu. To znači da za svaki čvor u, koji nije korijen, vrijedi $\text{key}(\text{Parent}(u)) \geq \text{key}(u)$, dakle ključ roditelja je uvijek veći ili jednak ključevima djece. To implicira da, idući od završnog čvora prema korijenu, ključevi se javljaju u rastućem (ne striktno rastućem, nego samo rastućem) redoslijedu. To isto tako implicira da je korijen nužno najveći element.



Slika: Lijevo potpuno binarno stablo



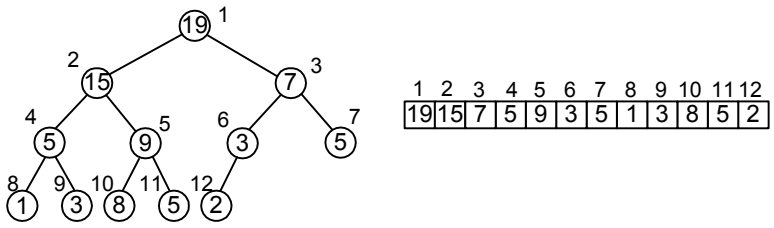
Slika: Heap struktura (raspored)

Pohrana heap strukture podataka kao polja

Jedno od dobrih svojstava heap strukture je da se može pohraniti kao polje, dakle bez upotrebe pokazivača (pointer-a) kao što bi to bilo normalno potrebno za pohranu binarnog stabla. Razlog zašto je to moguće je u činjenici da je heap struktura binarno stablo koje je lijevo potpuno.

Uzmimo da heap strukturu smještamo u polje $A[1..n]$. Općenito nećemo koristiti cijelo polje $A[1..n]$, jer ne moraju svi ključevi biti dio trenutne heap strukture. Iz tog razloga definiramo varijablu $m \leq n$ u koju smještamo informaciju o trenutnom broju elementa koji su aktivno pohranjen u heap strukturi. Stoga je heap struktura zapravo pohranjena u polju $A[1..m]$.

Heap struktura se pohranjuje u polju jednostavnim upisivanjem razinu po razinu. Budući je binarno stablo lijevo potpuno, točno znamo koliko elementa ima u svakoj razini. Korijen ima 1 čvor, slijedeća razina 2, zatim 4, zatim 8, itd. Samo u zadnjoj razini može broj elemenata biti različit od potencije broja 2. Ali u tom slučaju možemo upotrijebiti varijablu m da definiramo koji je element zadnji.



Slika: Pohranjivanje heap strukture kao binarnog stabla i kao polja

Može se lako pokazati da pristup elementima u heap strukturi zahtijeva jednostavne aritmetičke operacije na elementima polja, npr.:

*Left(i): vraća 2*i*

*Right(i): vraća 2*i+1*
Parent(i): vraća $\lfloor i/2 \rfloor$
IsLeaf(i): vraća Left(i)>m (tj, ako lijevo dijete i-tog čvora nije u stablu)
IsRoot(i): vraća i==1

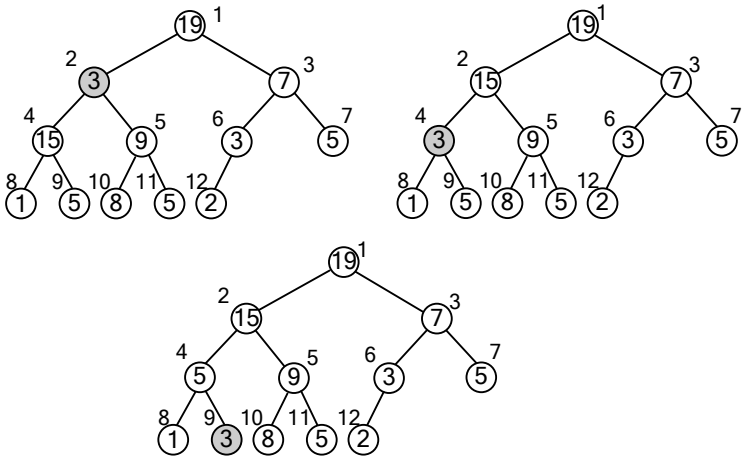
Većina procesora ove operacije može izvršiti u jednom ciklusu (primjerice 2*i je obično binarno pomicanje i za jedno mjesto u lijevo).

Održavanje heap rasporeda

U načelu imamo jednu operaciju koja održava heap raspored. Zovemo je Heapify (ponekad se zove i sifting down – prosijavanje). Ideja je da imamo element heap strukture za koji sumnjamo da nije u valjanom heap rasporedu, a pretpostavljamo da su svi elementi kojima je taj element korijen u valjanom heap rasporedu. Ako je taj element manji od pripadne djece (ili samo jednog djeteta), to znači da ne zadovoljava heap raspored. Da to ispravimo, element "prosijavamo" prema dolje tako da ga zamjenjujemo s jednim od djece. S kojim od djece? Da bi zadržali heap strukturu trebamo ga zamijeniti s većim od dvoje djece. Ovaj postupak nastavljamo rekurzivno sve dok element nije ili veći od oboje djece ili dok ne postane završni čvor.

Pseudokod je slijedeći. Neka je dana heap struktura smještena u polje A, neka je i indeks elementa kojeg želimo smjestiti na odgovarajuće mjesto, te neka je m veličina trenutno aktivne heap strukture. Element $A[max]$ se postavlja na maksimum od $A[i]$ i njegove dvoje djece. Ako je $max \neq i$ tada zamijeni $A[i]$ i $A[max]$, te potom nastavi dalje rekurzivno s $A[max]$.

```
Heapify(array A, int i, int m)
    l=Left(i)                          //lijevo dijete (2*i)
    r=Right(i)                         //desno dijete (2*i+1)
    max=i
    if (l<=m and A[l]>A[max]) max=l    //lijevo dijete postoji i veće je
    if (r<=m and A[r]>A[max]) max=r    //desno dijete postoji i veće je
    if (max!=i)                        //ako je neko dijete veće
        swap A[i] with A[max]          //zamijeni element s većim djetetom
        Heapify(A, max, m)             //rekurzivno nastavi
```



Slika: Heapify

Ovdje je dan vizualni prikaz načina rada Heapify u obliku stabla, a ne polja, jer je stablo najprirodniji način vizualizacije heap strukture. Ali slično se način rada može pokazati i na polju.

Rekurzivna implementacija Heapify nije najefikasnija. Procedura se može napisati i iterativno. Mi smo koristili rekurziju jer je za većinu algoritma na stablu rekurzija najprirodniji način primjene.

HeapSort algoritam sastoji se od dva temeljna dijela. Prvi je izgradnja heap strukture, a drugi je izvlačenje maksimalnih elemenata (jedan po jedan) iz te strukture. Vidjet ćemo kako nam Heapify pomaže da napravimo i jedno i drugo.

Ali da najprije vidimo koliko je vrijeme izvršavanja Heapify? Uočimo da prije svakog novog poziva Heapify treba uvijek isto, konstanto vrijeme. Stoga na svakoj razini (jer jedno izvršavanje Heapify odgovara jednoj razini) trebamo vrijeme od $O(1)$. Budući imamo ukupno $\Theta(\log n)$ razina u stablu, slijedi da je vrijeme izvršavanja za Heapify jednako $O(\log n)$. Dakle, ne $\Theta(\log n)$ jer ako, primjerice, radimo Heapify na završnom čvoru, tada je potrebno vrijeme jednako $\Theta(1)$.

Izgradnja heap strukture

Heapify možemo iskoristiti za izgradnju heap strukture na slijedeći način. Počinjemo od "heap strukture" u kojoj elementi nisu u heap strukturi. Neka je to jednostavno polje koje trebamo sortirati. Heap strukturu gradimo tako da počnemo od završnog čvora i potom pokrećemo Heapify na svakom čvoru. Zašto ne možemo početi od korijena ili nekog drugog unutarnjeg čvora? Zato što je po definiciji Heapify potrebno da svi elementi ispod trenutnog budu u heap strukturi. Zapravo čak i možemo početi od unutarnjeg čvora, ali samo od čvora koji je najbliži završnom čvoru, jer je po definiciji završni čvor sam za sebe u heap strukturi. Time je zapravo i algoritam značajno efikasniji. U tom slučaju (kada počinjemo od čvora koji je najbliži završnom čvoru) počinjemo zapravo od pozicije $\lfloor n/2 \rfloor$. (Zašto?)

Budući da radimo s cijelim poljem, parametar m za Heapify, koji se odnosi na trenutnu veličinu heap strukture, bit će jednak n, tj. veličini polja kojeg sortiramo i to u svim pozivima funkcije.

```
BuildHeap (int n, array A[1..n])
        for i=n/2 to 1
                Heapify (A, i, n)
```

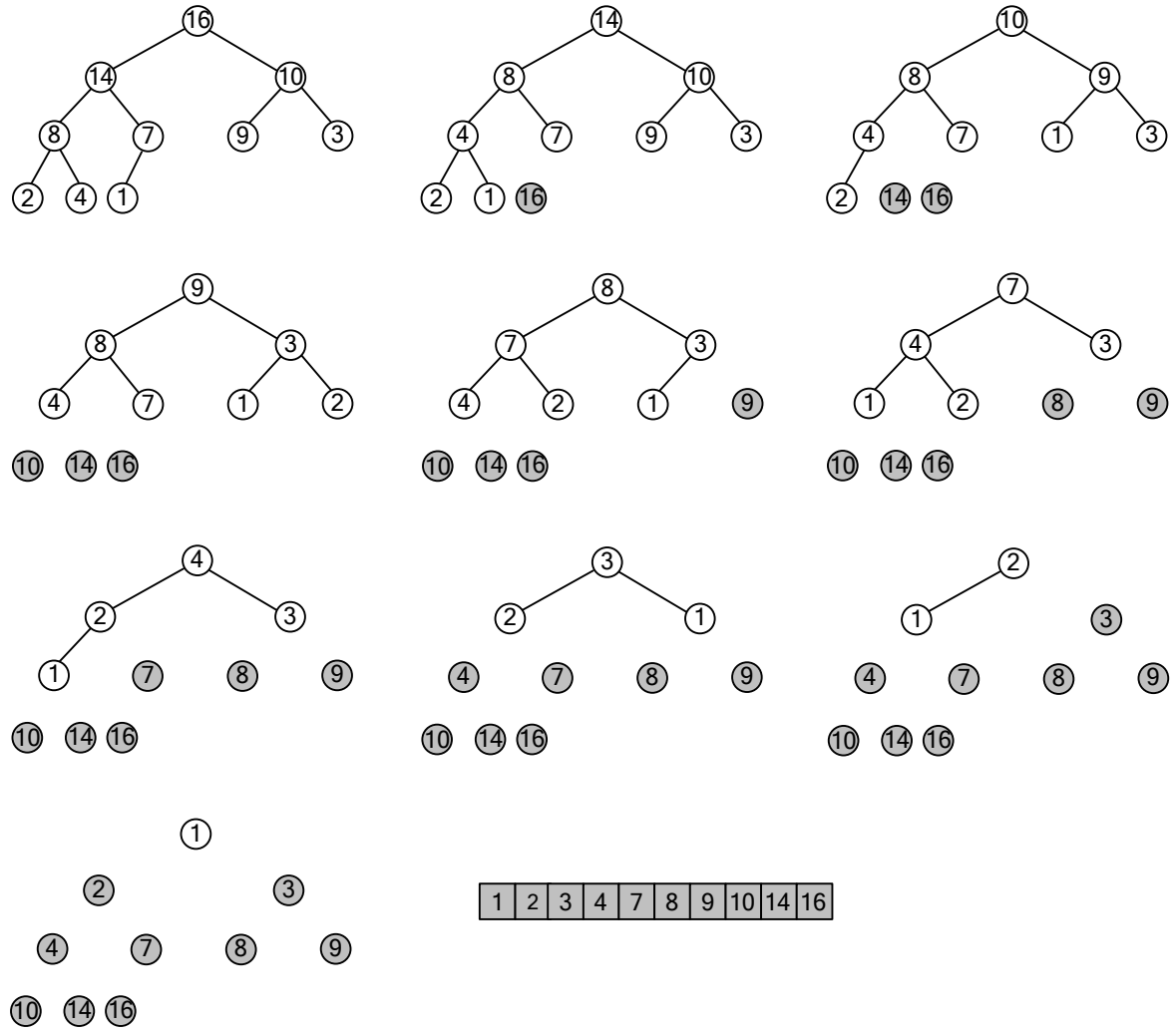
Svaki poziv Heapify traži $O(\log n)$ vremena, a poziv funkcije vršimo n/2 puta. Slijedi da je ukupno vrijeme izvršavanja BuildHeap jednako $O(n \log n)$. Kasnije ćemo vidjeti da je to vrijeme bitno brže od $\Theta(n \log n)$.

HeapSort

Sada možemo dati i HeapSort algoritam. Ideja je da svaki put uzmemo maksimalnu vrijednost iz heap strukture. U prvom koraku to je element koji je na vrhu strukture, dakle korijen. Kada uzmemo taj element (korijen) ostane praznina u stablu. Da to ispravimo zamijenimo ga s zadnjim završnim čvorom u stablu, dakle s $A[m]$. Sada je, naravno, heap struktura narušena pa jednostavno primijenimo Heapify na korijen čime smo opet dobili stablo s heap strukturom.

```
HeapSort (int n, array A[1..n])
        BuildHeap (n,A)             //izgradi heap strukturu
        m=n                         //inicijalno heap struktura sadrži sve elemente
        while (m>=2)
                swap A[1] with A[m] //izvuci m-ti najveći
                m=m-1               //izbaci A[m] (koji je već složen) iz heap struk.
                Heapify (A,1,m)     //složi element koji nije u heap strukturi
```

Vrijem izvršavanja HeapSort algoritma ima dva dijela: BuildHeap i while petlju. BuildHeap se izvršava s $O(n \log n)$. Heapify pozivamo ukupno n-1 put. Kako Heapify treba $O(\log n)$ vremena slijedi da je ukupno vrijeme izvršavanja while petlje jednako $O((n-1)\log n) = O(n \log n)$. Uzimajući u obzir vrijeme izvršavanja while petlje i BuilHeap slijedi da je ukupno vrijeme izvršavanje HeapSort algoritma jednako $O(n \log n)$.



Slika: Heapsort

Komentar na vrijeme izvršavanja HeapSort algoritma

Vidjeli smo da je vrijeme izvršavanja Heapify jednako $O(\log n)$ jer heap struktura ima $O(\log n)$ razina, a svaki element se pomiče za jednu razinu u konstantnom vremenu. Iz toga slijede dvije činjenice

1) Da nam za izgraditi heap strukturu (BuildHeap) treba $O(n \log n)$ vremena jer Heapify pozivamo približno n/2 puta (tj. za svaki od unutarnjih čvorova).

2) Da nam treba $O(n \log n)$ vremena da izvučemo maksimalne elemente, jer ih imamo n a za svaki treba pozvati Heapify koji ima vrijeme izvršavanja $O(\log n)$. Rezultat je da je vrijeme izvršavanja HeapSort algoritma jednako $O(n \log n)$.

Time smo odredili samo gornju granicu. Da li je time vrijeme izvršavanja čvrsto omeđeno, tj. da li je vrijeme izvršavanja $\Theta(n \log n)$? Odgovor je da. Ipak može se pokazati da prvi dio analize koji se odnosi na izgradnju heap strukture (BuildHeap) ne daje Θ asimptotsko vrijeme izvršavanja. Može se pokazati da je vrijeme izvršavanja jednako $\Theta(n)$. U kontekstu

HeapSort algoritma to nije ni važno jer je u svakom slučaju dominantno vrijeme izvršavanja (koje proizlazi iz faze izvlačenja maksimalnih vrijednosti) opet $\Theta(n \log n)$.

Ipak, ima situacija u kojima je činjenica da je vrijeme izgradnje heap strukture $\Theta(n)$, a ne $\Theta(n \log n)$, važna. Često nije ni potrebno sortirati sve elemente, nego primjerice samo elemente koji su manji od nekog zadanog kriterija. U tom slučaju zanimljivo je što prije izgraditi heap strukturu, jer i ovako nećemo iz te strukture izvlačiti sve elemente nego samo neke. U tom kontekstu važna je brzina izgradnje heap strukture.

6. QUICKSORT

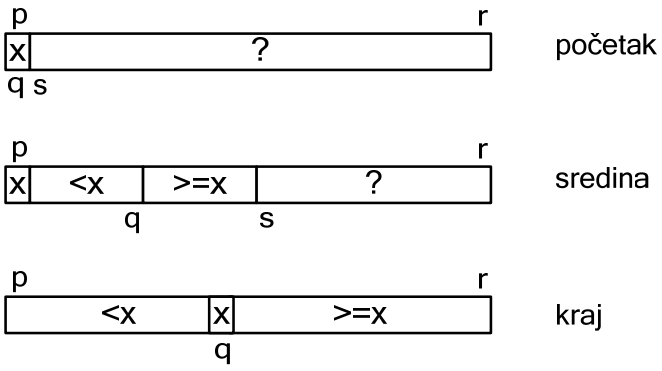
Slijedeći algoritam za sortiranje koji ćemo raditi je QuickSort. To je algoritma za sortiranje koji se danas vrlo često koristi. Iako ima isto očekivano (asimptotsko) vrijeme izvršavanja kao MergeSort i HeapSort, ipak je nešto brži na većini suvremenih računala, jer bolje iskorištava njihove karakteristike. Tako je i dobio ime. Ono što je posebno zanimljivo, to je algoritam koje se temelji na slučajnom odabiru (randomized algorithm). Pokazat ćemo da je njegovo najgore vrijeme izvršavanja jednako $\Theta(n^2)$, ali i da je očekivano vrijeme izvršavanja $\Theta(n \log n)$. To očekivano vrijeme pojavljuje se s velikom vjerojatnošću.

Algoritam dijeljenja

Prije nego obradimo QuickSort, obradit ćemo jedan njegov aspekt, a to je dijeljenje. Neka nam je dano polje $A[p..r]$ i jedna točka x iz polja A. Algoritam dijeljenja (partitioning algorithm) dijeli polje A u tri podpolja: $A[p..q-1]$ čiji su elementi manji (ili jednaki) x, $A[q] = x$ i $A[q+1..r]$ čiji su elementi veći ili jednaki x. Pretpostavimo da je x prvi element polja, tj. da je $x = A[p]$. U stvarnosti, x može biti bilo koji element polja. U tom slučaju, da bi mogli primijeniti algoritam koji smo razvili pod pretpostavkom da je x prvi element polja, jednostavno zamijenimo taj element s $A[p]$ prije pozivanja algoritma.

Da bi realizirali algoritam dijeljenja uvedimo još indeks s te pretpostavimo da polje dijelimo na 4 dijela:

1) $A[p] = x$ sadrži element x,
2) $A[p+1..q]$ sadrži elemente manje od x,
3) $A[q+1..s-1]$ sadrži elemente koji su veći ili jednaki x,
4) $A[s..r]$ sadrži elemente čija vrijednost trenutno nije poznata.

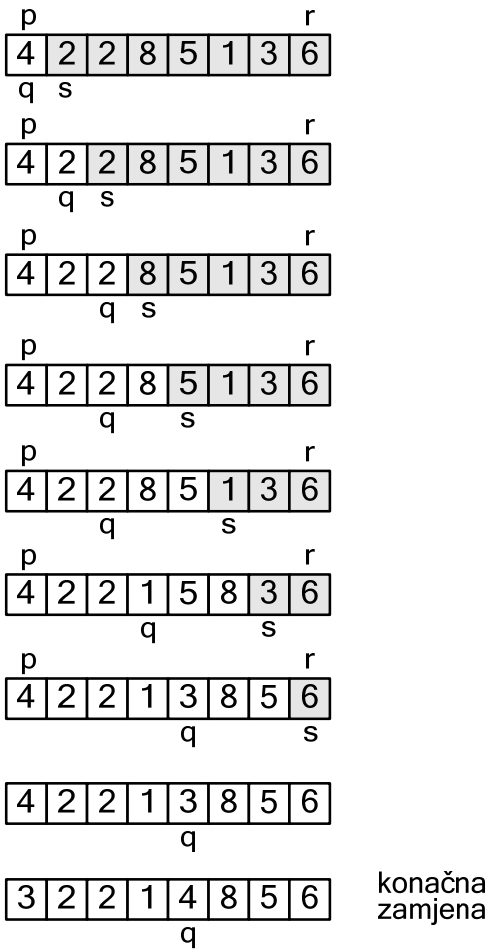


Slika: Ilustracija algoritma dijeljenja

Algoritam počinje postavljanjem $q = p$ i $s = p+1$. Sa svakim korakom algoritma ispitujemo vrijednost $A[s]$ u odnosu na x. Ako je $A[s] \geq x$, tada jednostavno povećamo s. Ako je

$A[s] < x$, povećamo q, zamijenimo $A[s]$ s $A[q]$, te zatim povećamo s. U oba slučaja zadržana su svojstva podpolja definirana kroz četiri, gore navedene, točke. Prvi slučaj je očigledan, a za drugi je potrebna jednostavna analiza. Algoritam završi kada je $s = r$, tj. kada su svi elementi obrađeni. Na kraju je potrebno zamijeniti $A[p]$ (dakle x) s $A[q]$ i vratiti vrijednost q. Zamjenom $A[p]$ s $A[q]$ stavljamo $A[p]$ na pripadajuće mjesto. Vraćamo vrijednost q (to je pozicija elementa x) jer je to pozicija polja koji ga dijeli na dva dijela: indeksi manji od q sadržavaju vrijednosti manje od x, dok indeksi veći i jednaki q sadržavaju vrijednosti veće ili jednake x. Slijedi odgovarajući pseudokod

```
Partition (int p, int r, array A)
    x = A[p]                          //x je jednak prvom elementu
    q = p
    for s=p+1 to r do
        if (A[s]<x)
            q=q+1
            swap A[q] with A[s]
    swap A[p] with A[q]               //stavi element x na odgovarajuće mjesto
    return q                          //vrati poziciju elementa x
```



Slika: Primjer algoritma dijeljenja

QuickSort i slučajno odabiranje

Kod proučavanja algoritama najčešće se traži najgore moguće vrijeme izvršavanja. Ipak, ponekad je važno pronaći prosječno vrijeme izvršavanja, posebno ako se prosječno i najgore moguće vrijeme bitno razlikuju. Upravo to je slučaj s QuickSort algoritmom. Kod njega je najgore moguće vrijeme $\Theta(n^2)$, a prosječno (očekivano) vrijeme izvršavanja $\Theta(n \log n)$.

Prezentirat ćemo QuickSort algoritam kao algoritam koji se temelji na slučajnom odabiru (i to na tzv. Las Vegas tipu algoritma). To je algoritam koji uvijek daje točan rezultat, ali je vrijeme izvršavanja slučajna varijabla. Osim tih, postoji i tzv. Monte Carlo algoritmi koji mogu dati pogrešan rezultat, ali se vjerojatnost pojavljivanja pogrešnog rezultata može učiniti po volji malom. QuickSort algoritam ne mora se implementirati tako da se temelji na slučajnom odabiru, ali je rješenje sa slučajnim odabirom bolje.

QuickSort algoritam

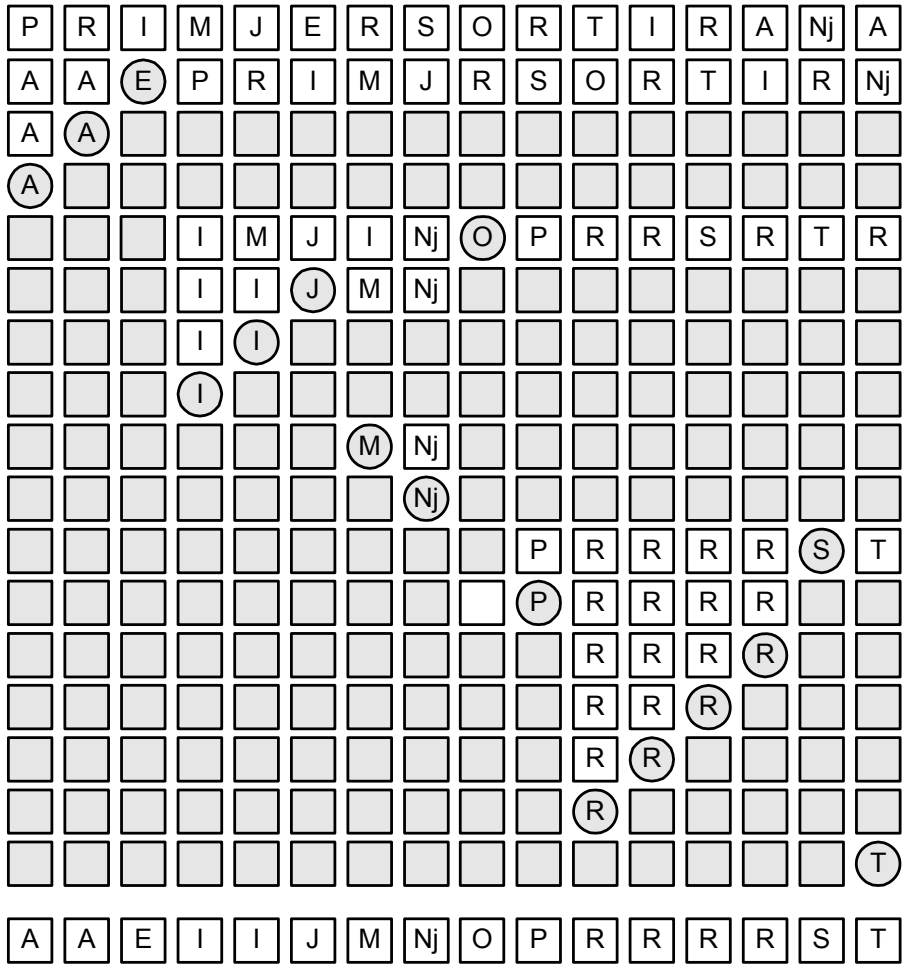
QuickSort algoritam se temelji na tehnici podijeli pa vladaj. Za razliku od MergeSort-a gdje se većina posla obavlja nakon rekurzivnog poziva, u QuickSort-u većina sa posla obavlja prije rekurzivnog poziva.

Neka je $A[p..r]$ (pod)polje koje treba sortirati. (To znači da će prvi, inicijalni, poziv odnositi na $A[1..n]$.) QuickSort algoritam ima slijedeće korake:

1. Ako polje ima 0 ili 1 element, onda je povratak (return).
2. Izaberi slučajni element x iz polja.
3. Podijeli polje na tri podpolja s elementima $A[p..q-1] < x$, $A[q] = x$ i $A[q+1..r] \geq x$.
4. Rekurzivno sortiraj $A[p..q-1]$ i $A[q+1..r]$.

Inicijalni poziv QuickSort-a je QuickSort(1, n, A). Procedura Partition, kako smo to definirali, uzima da je x jednak prvom elementu. Mi želimo da nam x bude slučajan element, što znači da moramo slučajno odabrati neki indeks i iz $[p..r]$, te potom zamijeniti $A[i]$ s $A[p]$.

```
QuickSort (int p, int r, array A)
    if (r <= p) return                  //0 ili jedan element, return
    i = slučajni indeks iz [p..r]       //odabiranje slučajnog indeksa (elementa)
    zamijeni A[i] s A[p]                //stavljanje x na odgovarajuće mjesto
    q = Partition (p, r, A)             //podjela A oko elementa x
    QuickSort (p, q-1, A)               // sortiranje A[p..q-1]
    QuickSort (q+1, r, A)               //sortiranje A[q+1..r]
```



Slika: Ilustracija sortiranja QuickSort-om

Analiza najgoreg mogućeg vremena izvršavanja

Točnost QuickSort algoritma je prilično jasna. Međutim, analiza vremena izvršavanja nije tako očigledna. Može se lako pokazati da vrijeme izvršavanja bitno ovisi o načinu na koji se bira element x. Ako je položaj elementa x u konačno sortiranom polju takav da je pri kraju ili pri početku polja, vrijeme izvršavanja će biti loše. Međutim, ako je položaj negdje blizu sredine, tada će i vrijeme izvršavanja biti dobro.

Analizu ćemo početi razmatranjem najgoreg mogućeg vremena izvršavanja. Kako je QuickSort rekurzivni program, koristit ćemo rekurziju za njegovo računanje. Za razliku od MergeSort-a gdje smo u svakom koraku znali broj elemenata u rekurziji, ovdje to ne znamo. Naime taj broj ovisi o odabranom elementu x.

Pretpostavimo da sortiramo polje veličine n, $A[1..n]$, te nadalje pretpostavimo da je element x na q-tom mjestu u zadanom polju. Može se lako vidjeti da dio koji se nalazi ispred rekurzivnog poziva (uključujući i proceduru Partition) ima vrijeme izvršavanja $\Theta(n)$. Imamo dva rekurzivna poziva. Prvi se odnosi na podpolje $A[1..q-1]$ koji ima $q-1$ element, a drugi na podpolje $A[q+1..n]$ koji ima $n-(q+1)+1=n-q$ elemenata. Slijedi da je

$$T(n)=T(q-1)+T(n-q)+n$$

Prethodni izraz ovisi o q. Da nađemo najgore moguće vrijeme moramo maksimizirati preko svih mogućih vrijednosti q. Znamo da je $T(0)=T(1)=\Theta(1)$ pa slijedi

$$T(n)=\begin{cases}1 & n\le 1\\ \max\limits_{1\le q\le n}(T(q-1)+T(n-q)+n) & \text{za ostale}\end{cases}$$

Rekurzije ovakvog tipa (koje uključuju maksimume i minimume) vrlo su nezgodne za rješavanje. Cilj je odrediti takav q koji daje maksimalnu vrijednost vremena izvršavanja. Za to nam može poslužiti opće pravilo koje kaže da je najgore moguće vrijeme uglavnom na ekstremnim vrijednostima ili na sredini. U tom smislu možemo pokušati s vrijednostima $q=1,\ q=n$ i $q=n/2$ odakle se može dobiti da je najgore moguće vrijeme izvršavanja za slučaj obiju ekstremnih vrijednosti. (Do istog zaključka može se doći i matematičkom analizom koristeći derivacije.) Ako sada razvijemo rekurziju za $q=1$ slijedi

$$\begin{aligned}T(n)&\le T(0)+T(n-1)+n=1+T(n-1)+n=\\&=T(n-1)+(n+1)\\&=T(n-2)+n+(n+1)\\&=T(n-3)+(n-1)+n+(n+1)\\&=T(n-4)+(n-2)+(n-1)+n+(n+1)\\&=...\\&=T(n-k)+\sum_{i=-1}^{k-2}(n-i)\end{aligned}$$

Kako je $T(1)=1$, uzmemo da je $n-k=1$, pa dalje slijedi:

$$\begin{aligned}T(n)&\le T(1)+\sum_{i=-1}^{n-3}(n-i)\\&=1+(3+4+5+...+(n-1)+n+(n+1))\\&\le\sum_{i=1}^{n+1}i=\frac{(n+1)(n+2)}{2}=O(n^2)\end{aligned}$$

Podrobnijom analizom može se pokazati da je najgore moguće vrijeme izvršavanja ne samo $O(n^2)$ nego i $\Theta(n^2)$.

Analiza prosječnog vremena izvršavanja

Pokažimo sada da je prosječno vrijeme izvršavanja QuickSort algoritma jednako $\Theta(n\log n)$. Očito je da vrijeme izvršavanja ovisi o načinu na koji se bira element x. Vrijeme izvršavanja ne ovisi o razdiobi ulaznih vrijednosti (vrijednosti koje treba sortirati). To je važna činjenica jer znači da je analiza ista za sve moguće ulaze. (Napomenimo da u većini slučajeva kod računanja prosječnog vremena izvršavanja treba uključiti i neke pretpostavke o razdiobi ulaza, što ovdje nije slučaj.) U ovom slučaju, prosječna vrijednost se računa preko svih mogućih slučajnih izbora elementa x.

Neka je $T(n)$ prosječno vrijeme izvršavanja QuickSort algoritma za n ulaza. Problem ćemo malo pojednostavniti na način da uzmemo da su svi ulazni elementi različiti. To znači da

algoritam može na n slučajnih načina odabrati element x i to s vjerojatnošću $1/n$. Sada rekurziju možemo modificirati tako da je prosječno vrijeme izvršavanja jednako

$$T(n)=\begin{cases}1 & n\le 1\\ \frac{1}{n}\sum_{q=1}^{n}\left(T(q-1)+T(n-q)+n\right) & \text{za ostale}\end{cases}$$

Ovo nije standardni oblik rekurzije, pa ne možemo primijeniti Master teorem. Najlakše će biti primijeniti konstruktivnu indukciju kako bi riješili ovu rekurziju. Željeli bismo pokazati da je vrijeme izvršavanja $\Theta(n\log n)$ pa stoga pokušajmo dokazati da postoji konstanta c takva da je $T(n)\le c\cdot n\cdot \ln n$ za sve $n\ge 2$.

Dokaz možemo napraviti konstruktivnom indukcijom. Za bazu, $n=2$ imamo

$$T(2)=\frac{1}{2}\sum_{q=1}^{2}\left(T(q-1)+T(2-q)+2\right)$$

$$=\frac{1}{2}\left(T(0)+T(1)+2+T(1)+T(0)+2\right)=\frac{8}{2}=4$$

Željeli bismo da je to najviše $c2\ln 2$ pa slijedi da je $c\ge 4/(2\ln 2)\approx 2.88$

Za korak indukcije pretpostavimo da je $n\ge 3$, te da je hipoteza indukcije da za svaki $n'<n$, imamo $T(n')\le cn'\ln n'$. Želimo dokazati da je to točno i za $T(n)$. Iz definicije za $T(n)$ slijedi

$$T(n)=\frac{1}{n}\sum_{q=1}^{n}\left(T(q-1)+T(n-q)+n\right)$$

$$=\frac{1}{n}\sum_{q=1}^{n}\left(T(q-1)+T(n-q)\right)+n$$

Uočite da ako sumu podijelimo na dvije sume, obje će sadržavati iste vrijednosti $(T(0)+T(1)+\ldots+T(n-1))$, samo što jedna broji prema gore a druga prema dolje. Stoga te dvije sume možemo zamijeniti s $2\sum_{q=0}^{n-1}T(q)$. $T(0)$ i $T(1)$ ne možemo računati po formuli pa ćemo te dvije vrijednosti posebno računati. Ako uvrstimo gornju supstituciju i primijenimo hipotezu indukcije na ostatak sume (što možemo jer je $q<n$) imamo

$$T(n)=\frac{2}{n}\left(\sum_{q=0}^{n-1}\left(T(q)\right)\right)+n$$

$$=\frac{2}{n}\left(T(0)+T(1)+\sum_{q=2}^{n-1}T(q)\right)+n$$

$$\le\frac{2}{n}\left(1+1+\sum_{q=2}^{n-1}cq\ln q\right)+n$$

$$=\frac{2c}{n}\left(\sum_{q=2}^{n-1}q\ln q\right)+n+\frac{4}{n}$$

Može se pokazati da za sumu u gornjem izrazu vrijedi

$$S(n)=\sum_{q=2}^{n-1} q\ln q=\left(\frac{n^2}{2}\ln n-\frac{n^2}{4}\right)-(2\ln 2-1)$$
$$\le \frac{n^2}{2}\ln n-\frac{n^2}{4}$$

Dalje slijedi da je

$$T(n)\le \frac{2c}{n}\left(\frac{n^2}{2}\ln n-\frac{n^2}{4}\right)+n+\frac{4}{n}$$
$$=cn\ln n-\frac{cn}{2}+n+\frac{4}{n}$$
$$=cn\ln n+n\left(1-\frac{c}{2}\right)+\frac{4}{n}$$

Da bi završili dokaz moramo pokazati da postoji takav c za koji će gornji izraz biti najviše $cn\ln n$. To će biti ispunjeno ako je $n\left(1-\frac{c}{2}\right)+\frac{4}{n}\le 0$. Lako se vidi da je uz $c=3$ (uvažavajući da je $n\ge 3$) ta nejednakost ispunjena jer je

$$n\left(1-\frac{3}{2}\right)+\frac{4}{n}=-\frac{n}{2}+\frac{4}{n}=\frac{8-n^2}{2n}$$

što je za $n\ge 3$ sigurno manje od 0, što smo i željeli pokazati. Iz uvjeta za bazu dobili smo da je $c\ge 2.88$, što znači da možemo uzeti $c=3$ jer su time zadovoljeni svi uvjeti.

Zaključak

Iako je najgore moguće vrijeme izvršavanja QuickSort algoritma $\Theta(n^2)$, prosječno vrijeme izvršavanja je $\Theta(n\log n)$. Iako nismo to izvodili, može se pokazati da se najgore moguće vrijeme javi vrlo rijetko. Za velike vrijednosti n, vrijeme izvršavanja je $\Theta(n\log n)$ s vrlo velikom vjerojatnošću. Da bi dobili vrijeme izvršavanja $\Theta(n^2)$ moramo izabrati loš x gotovo u svakom koraku algoritma. Loš izbor elementa x je rijedak, a posebno je onda rijetko da se više puta uzastopno izabere loš element x.

Da li se može napraviti takav QuickSort algoritam da vrijeme izvršavanja bude uvijek $\Theta(n\log n)$ tako što bi svaki put za element x birali srednji element? Može, samo što se na takav način dobije algoritam koji je spor pa se i ne koristi.

QuickSort je danas najpopularniji algoritam za sortiranje jer je njegovo izvršavanja na suvremenim računalima nešto bolje od drugih algoritama za sortiranje. Razlog je u činjenici da, za razliku od HeapSort-a koji pravi velike skokove po polju, QuickSort uglavnom pristupa elementima koji su blizu jedan drugoga. Nešto je bolji od MergeSort-a (koji također uglavnom pristupa bliskim elementima) jer se većina usporedbi radi s istim elementom (element x) koji se onda može pohraniti u registru. MergeSort uvijek uspoređuje dva (različita) elementa polja. Učinkovitost QuickSort algoritma može se još povećati na način da se za velika polja koristi rekurzija, a kada jednom veličina polja padne na neku malu vrijednost (npr. 20 elemenata) onda se koristi neki jednostavni iterativni algoritam kao što je npr. sortiranje odabirom.

7. DONJA GRANICA VREMENA IZVRŠAVANJA ALGORITAMA ZA SORTIRANJE

Za algoritme koji ne trebaju dodatnu memoriju (dodatno polje) za sortiranje, kažemo da sortiraju u mjestu (in-place). Za algoritam koji iste elemente nakon sortiranja zadržava u istom relativnom odnosu, kažemo da je stabilan (stable). Vidjet ćemo kakvi su algoritmi koje smo do sada obradili s obzirom na stabilnost i sortiranje u mjestu.

Spori algoritmi /BubbleSort (sortiranje u valovima), InsertionSort (sortiranje umetanjem), SelectionSort (sortiranje odabirom)/ - To su algoritmi s $\Theta(n^2)$. Sortiranje vrše u mjestu. BubbleSort i InsertionSort mogu se implementirati kao stabilni algoritmi, a SelectionSort ne može (bez značajnih modifikacija).

MergeSort – MergeSort je stabilan algoritam s vremenom izvršavanja od $\Theta(n \log n)$. Loša strana je što MergeSort nije algoritam koji sortira u mjestu jer zahtijeva dodatnu memoriju.

QuickSort – Općenito se smatra najbržim od brzih algoritama. Algoritam ima očekivano vrijeme izvršavanja od $O(n \log n)$, a najgore vrijeme izvršavanja od $O(n^2)$. Vjerojatnost da se algoritam izvršava sporije od $O(n \log n)$ je izuzetno mala za velike n (uz uvjet da se element x bira slučajno). Obično kažemo da QuickSort algoritam sortira u mjestu, iako zapravo zahtijeva stack veličine $O(\log n)$ kako bi pratio izvršavanje rekurzije. QuickSort algoritam nije stabilan.

HeapSort – Temelji se na heap strukturi podataka. Vrijeme izvršavanja je $\Theta(n \log n)$. Elementi se dodaju u heap strukturu u vremenu $O(\log n)$, a najveći element može se dobiti u $O(\log n)$ vremenu. HeapSort je vrlo zanimljiv kada ne trebamo sortirati sve elemente. Primjerice ako nam treba samo k najvećih elemenata, onda HeapSort algoritam to radi u $O(n + k \log n)$ vremenu. To je algoritam koji sortira u mjestu, ali nije stabilan.

Donja granica za algoritme sortiranja koji se temelje na usporedbi

Može li se sortirati brže od $O(n \log n)$? Pokazat ćemo da, ako se algoritam temelji samo na usporedbi, onda se takvim algoritmom ne može sortirati brže od $\Omega(n \log n)$.
Svi, gore spomenuti algoritmi, temelje se na usporedbi (comparison-based sorthing algorithms). Praktično svi algoritmi za sortiranje opće namjene temelje se na usporedbi. To ne isključuje mogućnost i bržeg slaganja, ali pod nekim drugim, posebnim uvjetima.

Mi ćemo pokazati da bilo koji algoritam koji se temelji na usporedbi, da bi sortirao niz od n brojeva treba najmanje $\Omega(n \log n)$ usporedbi u najgorem slučaju.

Može se lako pokazati da je neki konkretni algoritam brz (samo treba analizirati algoritam). Iz toga slijedi da se lako može pokazati i da algoritam za sortiranje može biti barem jednak nekoj brzini. Međutim, na prvi pogled ne izgleda da je uopće moguće pokazati da se problem sortiranja ne može riješiti brže od nekog vremena. Naime, to se odnosi na sve algoritme (uključujući i one koji još nisu napisani), pa pokazati da niti jedan neće biti brži od nekog vremena nije lako. Kako ćemo to vidjeti u nastavku, kada se ograničimo na algoritme koji se temelje na usporedbi, onda ćemo moći matematičkom analizom, pronaći tu donju granicu.

Stablo odlučivanja

Da bi mogli dokazati postojanje donje granice, moramo imati način na koji ćemo modelirati bilo koji algoritam koji se temelji na usporedbi. To radimo pomoću apstraktnog modela kojeg nazivamo stablo odlučivanja (decision tree).

Kod algoritama koji se temelje na usporedbi, jedini način na koji se može izvršiti sortiranje je usporedba. Neka je zadan ulazni niz $(a_1, a_2, \ldots, a_n)$. To znači da se za dva elementa, a_i i a_j, njihov relativni položaj može odrediti isključivo kao rezultat usporedbi, kao što su $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$ i $a_i > a_j$.

Stablo odlučivanja je način matematičkog predstavljanja algoritma za sortiranje (za fiksnu vrijednost n). Svaki čvor u stablu odlučivanja predstavlja usporedbu u algoritmu (npr. elementa a_4 i a_7). Dvije grane predstavljaju dva moguća rezultata, npr. lijeva grana (to znači i lijevo podstablo) sastoji se od svih preostalih usporedbi napravljenih pod pretpostavkom da je $a_4 \leq a_7$, a desna grana od svih preostalih usporedbi za slučaj kada je $a_4 > a_7$.

Ako znamo vrijednost n, tada su sve akcije algoritma za sortiranje u potpunosti određene rezultatima usporedbi. Te akcije uključuju npr. pomicanje elementa po polju, kopiranje elementa na neko drugo mjesto u memoriji i sl. Na kraju rezultat mora biti polje čiji su elementi presloženi u rastućem (ili padajućem) redoslijedu. Da budemo malo jasniji, uzmimo da je $n = 3$, te izgradimo stablo odlučivanja za SelectionSort. U našem slučaju SelectionSort će vršiti sortiranje na slijedeći način: najprije će naći najmanji element cijele liste, te ga zamijeni s prvim elementom. Potom će naći manji od preostala dva elementa, te ga zamijeniti s drugim elementom. U nastavku je dano stablo odlučivanja i u tekstualnom i u grafičkom obliku. Prva usporedba je između a_1 i a_2. Mogući rezultati su:

$a_1 \leq a_2$ - U tom slučaju je a_1 trenutni minimum. Slijedeće što treba napraviti je usporediti a_1 s a_3 za što može vrijediti

$a_1 \leq a_3$ - Sada znamo da je a_1 minimum svih elemenata, što znači da ostaje na svom originalnom, prvom mjestu. Sada moramo usporediti a_2 s a_3. Mogući rezultati su:

$a_2 \leq a_3$ - Konačni izlaz je (a_1, a_2, a_3).

$a_2 > a_3$ - Znači da treba zamijeniti a_2 s a_3 pa je konačni izlaz (a_1, a_3, a_2).

$a_1 > a_3$ - Sada znamo da je a_3 minimum svih elemenata, što znači da ga trebamo zamijeniti s a_1. Nakon toga moramo usporediti a_2 s a_1 (koji je sada na trećem mjestu u polju). Mogući rezultati su:

$a_2 \le a_1$ - Konačni izlaz je (a_3, a_2, a_1).

$a_2 > a_1$ - Znači da treba zamijeniti a_2 s a_1 pa je konačni izlaz (a_3, a_1, a_2).

$a_1 > a_2$ - U tom slučaju je a_2 trenutni minimum. Slijedeće što treba napraviti je usporediti a_2 s a_3 za što može vrijediti

$a_2 \le a_3$ - Sada znamo da je a_2 minimum svih elemenata, što znači da mijenjamo a_2 s a_1. Sada moramo usporediti a_1 s a_3. Mogući rezultati su:

$a_1 \le a_3$ - Konačni izlaz je (a_2, a_1, a_3).

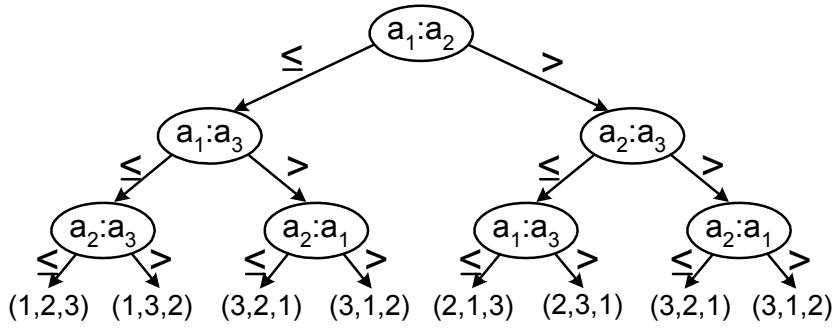
$a_1 > a_3$ - Znači da treba zamijeniti a_1 s a_3 pa je konačni izlaz (a_2, a_3, a_1).

$a_2 > a_3$ - Sada znamo da je a_3 minimum svih elemenata, što znači da mijenjamo a_3 s a_1. Sada moramo usporediti a_2 s a_1 (koji je sada na trećem mjestu u polju). Mogući rezultati su:

$a_2 \le a_1$ - Konačni izlaz je (a_3, a_2, a_1).

$a_2 > a_1$ - Znači da treba zamijeniti a_1 s a_2 pa je konačni izlaz (a_3, a_1, a_2).

Konačno stablo odlučivanja prikazano ja na slici. Uočite da postoje neki čvorovi koje nikako ne možemo doseći. Primjerice, da dosegnemo krajnju desnu granu, mora biti ispunjeno $a_1 > a_2$ i $a_2 > a_1$, što istovremeno nikako ne može biti točno. (Zašto postoje neke grane koje se nikada neće doseći?)



Slika: Stablo odlučivanja za SelectionSort s tri elementa

Kao što se može vidjeti ih prethodnog razmatranja, razvijanje stabla odlučivanja za složenije algoritme (npr. HeapSort) ili velike n-ove, biti će dugačko, ali i jednostavno u smislu da je svaki korak jasan.

Upotreba stabla odlučivanja za analiziranje sortiranja

Razmotrimo općeniti (bilo koji) algoritam za sortiranje. Neka je $T(n)$ maksimalni broj usporedbi koji algoritam radi za ulaz veličine n. Uočite da vrijeme izvršavanja algoritma, u

najgorem slučaju, mora biti barem $T(n)$. Algoritam definira stablo odlučivanja. Uočite i da je visina stabla odlučivanja točno jednaka $T(n)$.

Bilo koje stablo visine $T(n)$ ima najviše $2^{T(n)}$ grana (krajnjih). To znači da algoritam za sortiranja može razlikovati najviše $2^{T(n)}$ različitih konačnih akcija (ne rezultata, jer mogućih rezultata ima manje). Označimo s $A(n)$ tu vrijednost, dakle najveći broj različitih konačnih akcija. Svaka akcija predstavlja jedan put (način) na koji se može, permutirajući originalni ulaz, doći do sortiranog izlaza.

Koliko mogućih izlaza (rezultata) algoritam za sortiranje mora razlikovati. Ako se ulaz sastoji od n brojeva, tada imamo ukupno $n!$ mogućih izlaza (permutacija). Za svaku permutaciju algoritam mora imati različiti put, dakle različitu akciju. Slijedi da mora biti $A(n) \geq n!$. $A(n)$ općenito nije jednak $n!$, jer mnogi algoritmi sadrže redundantne grane, grane koje se nikada ne mogu doseći.

Kako je $A(n) \leq 2^{T(n)}$ imamo da je $2^{T(n)} \geq n!$ odakle slijedi

$$T(n) \geq \mathrm{ld}(n!)$$

Sada možemo upotrijebiti Stirling-ovu aproksimaciju za $n!$

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Slijedi da je

$$T(n) \geq \mathrm{ld}\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$$
$$= \mathrm{ld}\left(\sqrt{2\pi n}\right) + n\,\mathrm{ld}\,n - n\,\mathrm{ld}\,e \in \Omega(n \log n)$$

Zaključak: Bilo koji algoritam za sortiranje koji se temelji na usporedbi ima (u najboljem slučaju) najgore vrijeme izvršavanja od $\Omega(n \log n)$.

Ovaj zaključak može se generalizirati na način da se pokaže da je i prosječno vrijeme za sortiranje također jednako $\Omega(n \log n)$.

7.1. SORTIRANJE S LINEARNIM VREMENOM

Vidjeli smo da nije moguće sortirati brže od $\Omega(n \log n)$ ali pod uvjetom da radimo usporedbu elemenata. Taj se zaključak temeljio na činjenici da bilo koji algoritam za sortiranje koji se temelji na usporedbi možemo prikazati kao stablo odlučivanja. Stablo odlučivanja mora imati $n!$ grana, pa stoga visina stabla odlučivanja mora biti barem $\mathrm{ld}(n!)$, a vrijedi $\mathrm{ld}(n!) \in \Omega(n \log n)$.

Možemo zaključiti da ako želimo sortirati brže od $n \log n$, tada to ne možemo napraviti koristeći samo usporedbu. Da li je moguće sortirati bez usporedbe? Da, ali samo uz određene, i to vrlo ograničavajuće, uvjete. Prezentirat ćemo dva algoritma koji u specijalnim slučajevima mogu sortirati brže, bez usporedbe elemenata.

Sortiranje brojenjem (CountingSort)

CountingSort pretpostavlja da su svi ulazi cijeli brojevi u rasponu od 1 do k. Ovaj algoritam sortira u vremenu $\Theta(n+k)$. Ako nam je k poznat i ako iznosi $\Theta(n)$, tada je vrijeme sortiranja algoritma jednako $\Theta(n)$.

Temeljna ideja je da se za svaki element u polju odredi njegov položaj u konačno sortiranom polju. Položaj (engl. rank) je broj elementa u polju koji su manji ili jednaki od danog elementa. Ako znamo položaj svakog elementa, onda možemo sortirati jednostavnim kopiranjem elementa na odgovarajuće mjesto u finalnom, sortiranom polju. Pitanje je kako naći položaj elementa a bez da ga uspoređujemo s drugim elementima?

CountingSort koristi tri polja. Kao i obično $A[1..n]$ je ulazno polje. Iako obično mislimo o A kao o listi brojeva, to je zapravo lista zapisa (records), pri čemu je ključ neka vrijednost na temelju koje je lista sortirana.

Kako smo već rekli, koristimo tri polja

$A[1..n]$ - Sadrži inicijalni, ulazni niz. $A[j]$ je zapis. $A[j].key$ je cjelobrojni ključ na temelju čije vrijednosti ćemo vršiti sortiranje.

$B[1..n]$ - Polje zapisa koje sadržava sortirane vrijednosti.

$R[1..k]$- Polje cijelih brojeva. $R[x]$ je položaj x-a u A, gdje je $x \in [1..k]$.

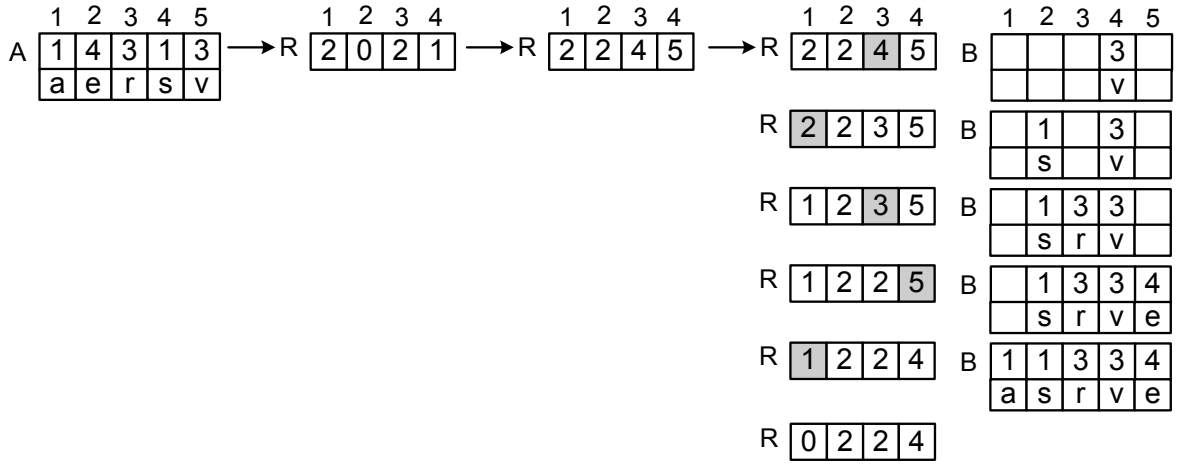
Algoritam je vrlo jednostavan. Najprije odredimo R. To radimo u dva koraka. Prvo postavimo $R[x]$ da bude jednak broju elemenata $A[j]$ čiji je ključ jednak x-u. To možemo napraviti tako da postavimo R u nulu, te potom za svaki j, od 1 do n, povećamo $R[A[j].key]$ za jedan. Primjerice, ako je $A[j].key = 6$, znači da ćemo povećati 6-ti element polja R za jedan, a to signalizira da imamo još jednu šesticu. Da bi odredili broj elemenata koji su manji ili jednaki x, zamijenimo $R[x]$ sa sumom svih elemenata u podpolju $R[1..x]$.

Sada $R[x]$ sadrži položaj x-a. To znači da ako je $x = A[j].key$ tada konačni položaj $A[j]$ treba biti na položaju $R[x]$. Stoga stavljamo $B[R[x]] = A[j]$. Uočite da ovim kopiramo cijeli zapis, a ne samo ključ. Treba biti pažljiv, ako imamo dupliciranih vrijednosti, da ne prebrišu same sebe u polju B.

```
CountingSort (int n, int k, array A, array B)
    for x = 1 to k do R[x]=0                  //inicijaliziranje R
    for j = 1 to n do R[A[j].key]++           //R[x] dodamo 1 ako je A[j].key==x
    for x = 2 to k do R[x] += R[x-1]          //R[x] postaje položaj x-a
    for j = n to 1 do                         //pomicanje elementa iz A u B
        x = A[j].key
```

```
B[R[x]] = A[j]
R[x]--                                  //da bi riješili problem duplih
```

CountingSort algoritam ima ukupno 4 (neugniježđene) petlje koje se izvršavaju k puta, n puta, $k-1$ puta i n puta. Dakle, ukupno vrijeme izvršavanja je $\Theta(n+k)$. Ako je $k=O(n)$, tada je ukupno vrijeme izvršavanja jednako $\Theta(n)$.



Slika: Primjer sortiranja CountingSort-om

CountingSort očito nije algoritam koji sortira u mjestu jer treba dva dodatna polja. Da li je ovo stabilan algoritam?

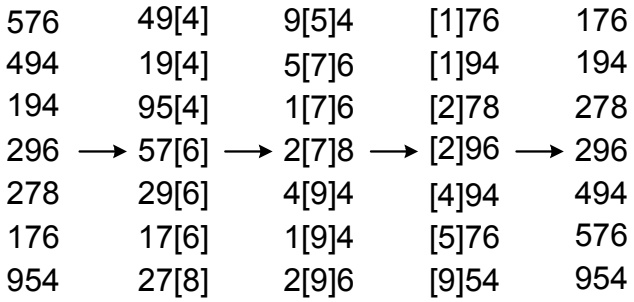
Korijensko sortiranje (RadixSort)

Glavni nedostatak CountingSort algoritma je što je realno primjenjiv (zbog ograničene memorije) samo za relativno male cijele brojeve. Ako su nam cijeli brojevi od 1 do 100 milijuna, nije realno da alociramo memoriju od 100 milijuna elemenata samo da bi sortirali polje. RadixSort rješava taj problem na način da sortira brojeve znamenku po znamenku (digit po digit). Što je znamenka odnosno digit ovisi o konkretnoj implementaciji. Primjerice, često se sortira po bajtovima.

Ideja je opet vrlo jednostavna. Neka se naša lista koju želimo sortirati sastoji od n brojeva od kojih svaki ima d znamenki (decimalnih znamenki ili po bilo kojoj bazi). Pretpostavimo također da imamo stabilan algoritam za sortiranje, kao što je CountingSort. Da bi sortirali brojeve jednostavno sortiramo od najmanje značajne znamenke (digita) prema značajnijoj. Kako je algoritam za sortiranje stabilan, znamo da će sortirani bojevi zadržati pravilan redoslijed kada nakon sortiranja najmanje važne znamenke, sortiramo i ostale.

Kao i obično, neka je $A[1..n]$ polje koje trebamo sortirati, neka je d broj znamenki u A. Nećemo posebno razmatrati kako A podijeliti na znamenke (digite). Samo spomenimo da se to može napraviti manipulacijom na bitima (pomicanje, maskiranje) ili pristupanjem elementu bajt po bajt.

```
RadixSort (int n, int d, array A)
    for i = 1 to d do
        sortiraj A (stabilno) po i-tom najmanje značajnom bitu
```



Slika: Primjer sortiranja RadixSort-om

Očito je vrijeme izvršavanja RadixSort-a jednako $\Theta(d \cdot (n + k))$, gdje je d broj znamenki, n je dužina polja kojeg sortiramo, a k je broj vrijednosti koje znamenka može imati. k je obično konstanta, pa će vrijeme izvršavanja algoritma biti $\Theta(dn)$ ako je $k \in O(n)$.

8. ALGORITMI TEMELJENI NA GRAFOVIMA (GRAPH ALGORITHMS)

Veliki broj problema najlakše je i najprirodnije formulirati u obliku grafova. To se javlja svaki put kada imamo niz objekata i veza (odnosa, interakcije) između parova objekata.

Takav primjer je karta sa svim letovima u Europi. Nekoga može zanimati: Koji je najbrži način da se dođe od Splita od Bratislave? Drugoga može zanimati koji je najjeftiniji način? Za odgovor na ta pitanja moramo imati samo informacije o vezama (avionskim linijama) među objektima, ali ne i o samim objektima (gradovima). (Ovo je transportna mreža, nama su važne i komunikacijske mreže).

Drugi tipičan primjer su veze na tiskanim pločicama. Najjednostavnije pitanje koje možemo postaviti je: Da li su svi element povezani? Nešto složenije pitanje je: Da li će tiskana pločica raditi kako je predviđeno? Za odgovor na prvo pitanje potrebne su nam samo informacije o svojstvima veza, a odgovor na drugo pitanje zahtijeva informacije o vezama, i o svojstvima objekata.

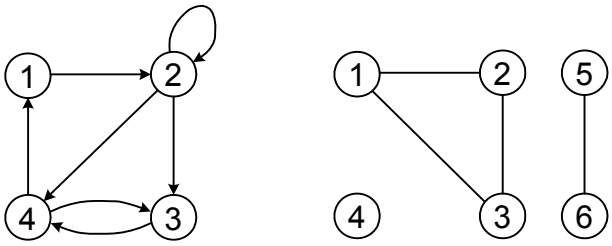
Treći primjer je planiranje poslova kada jedan posao ovisi o drugome pa nas zanima koji posao treba biti obavljen prije ostalih (ili prije nekog drugog). Uglavnom će nas zanimati odgovor na pitanje: Kada pojedini posao (zadatak) treba obaviti?

Graf je matematički objekt kojim možemo modelirati takve situacije. Teorija grafova je grana matematike (kombinatorike) koja se intenzivno proučava već više stotina godina. Ipak, sa stajališta algoritama, primjena grafova je relativno nova. Iako su neki temeljni algoritmi prilično stari, najveći dio algoritma temeljenih na grafovima otkriven je u zadnjih 20-tak godina. Algoritmi temeljeni na grafovima su trenutno jedno od najprodornijih i najzanimljivijih područja u računalnoj znanosti.

8.1. DEFINIRANJE POJMOVA

Graf je skup **čvorova** (nodes, vertices (jednina je vertex)) povezanih **vezama** (edges). Razlikujemo **neusmjereni graf** (undirected graph) i **usmjereni (orijentirani) graf** (directed graph, digraph). Neusmjereni graf najčešće nazivamo samo **graf**.

Usmjereni graf $G=(V,E)$ sastoji se od konačnog skupa čvorova V i binarne relacije E koja je definirana na skupu V. Binarna relacija znači da je to skup uređenih parova iz V. Uočite da je kod usmjerenog grafa (za razliku od neusmjerenog) dozvoljena **petlja** koja se zatvara sama u sebe (self-loop), kao što je slučaj na čvoru 2 prikazanom na slici. Višestruke veze nisu dozvoljene (vodite računa da veze (v,u) i (u,v) nisu iste). Na slici je prikazan usmjereni graf $G=(V,E)$ čiji je skup čvorova $V=\{1,2,3,4\}$, a skup svih veza $E=\{(1,2),(2,2),(2,3),(2,4),(3,4),(4,3),(4,1)\}$.



Slika: Usmjereni i neusmjereni graf

Neusmjereni graf (ili samo **graf**) $G=(V,E)$ ima čvorove čiji parovi nisu uređeni pa su veze (v,u) i (u,v) iste. Kod neusmjerenih grafova petlje koje se zatvaraju same u sebe nisu dozvoljene. Na slici je prikazan neusmjereni graf s čvorovima $V=\{1,2,3,4,5,6\}$ i vezama $E=\{(1,2),(1,3),(2,3),(5,6)\}$.

Za čvor v kažemo da je **susjedni** (adjacent) čvoru u, ako u grafu (usmjerenom ili neusmjerenom) $G=(V,E)$ postoji veza (u,v). Ako je graf neusmjeren, tada je relacija susjedstva simetrična.

Ako je (u,v) veza u usmjerenom grafu $G=(V,E)$, tada kažemo da veza (u,v) **izlazi iz** (incident from, leaves) čvora u, odnosno da **ulazi u** (incident to, enters) čvor v. Tako na slici veze koje izlaze iz čvora 2 su $(2,2)$, $(2,4)$ i $(2,3)$. Veze koje ulaze u čvor 2 su $(1,2)$ i $(2,2)$. Ako je (u,v) veza u neusmjerenom grafu $G=(V,E)$, tada kažemo da veza (u,v) **upada na** (incident on) čvorove u i v. Tako veze koje upadaju na čvor 2 su $(1,2)$ i $(2,3)$. Uočite da su usmjereni i neusmjereni grafovi matematički različiti objekti (ali ipak slični).

Za usmjereni ili neusmjereni graf kažemo da je **otežan**, ako je njegovim vezama pridružena numerička vrijednost (težina). Značenje težine ovisno je o konkretnoj primjeni, npr. to može biti udaljenost, kapacitet, cijena i sl.

Kod usmjerenog grafa, broj veza koje izlaze iz čvora naziva se **izlazni stupanj** (out-degree) čvora, a broj veza koje ulaze u čvor **ulazni stupanj** (in-degree) čvora. Suma izlaznog i ulaznog stupnja čvora, kod usmjerenog grafa, naziva se **stupanj** čvora. Na slici je za čvor 2 ulazni stupanj 2, izlazni stupanj je 3, a stupanj 5. Ako je graf neusmjeren, onda govorimo samo o **stupnju** (degree) čvora. Čvor čiji je stupanj 0 je **izolirani čvor** (isolated). Na slici je za čvor 2 stupanj 2, a čvor 4 je izolirani čvor.

Kod usmjerenog grafa, svaka veza povećava ulazni stupanj nekog čvora za jedan, te također izlazni stupanj tog istog ili nekog drugog čvora za jedan. Dakle, za usmjereni graf $G=(V,E)$ vrijedi

$$\sum_{v\in V}\text{ulazni\_stupanj}(v)=\sum_{v\in V}\text{izlazni\_stupanj}(v)=|E|$$

gdje je $|E|$ kardinalni broj (broj elemenata u skupu E).

Kod neusmjerenog grafa, svaka veza doprinosi povećanju stupnja dva različita čvora pa slijedi

$$\sum_{v \in V} \text{stupanj}(v) = 2|E|$$

Put (putanja, staza (path)) duljine k, iz čvora u prema čvoru u', u usmjerenom grafu $G = (V, E)$, je niz čvorova $(v_0, v_1, v_2, \ldots, v_k)$ takvih da je $u = v_0$ i $u' = v_k$ te $(v_{i-1}, v_i) \in E$ za $i = 1,2,\ldots k$. **Duljina puta** (length) jednaka je broju veza na putu. Kažemo da je čvor u' **dohvatljiv** (reachable) iz čvora u ako postoji put iz u prema u'. (Uočite da je svaki čvor dohvatljiv iz samog sebe i to putem čija je duljina 0). Put je **jednostavan** (simple) ako su svi čvorovi na putu različiti (osim eventualno prvog i zadnjeg). Na slici je put $(1,2,3,4)$ jednostavan, dok put $(2,3,4,3)$ nije jednostavan.

Ciklus (cycle) u usmjerenom grafu je put koji sadrži barem jednu vezu i za kojeg vrijedi $v_0 = v_k$. Ciklus je jednostavan ako su čvorovi $v_1, v_2, \ldots, v_k$ različiti. Petlju koja se zatvara sama u sebe (kao za čvor 2 na slici) smatramo jednostavnim ciklusom duljine 1.

Kod neusmjerenog grafa, put i ciklus definiramo na isti način kao i za usmjereni uz još jedan dodatni zahtjev za ciklus. Naime kod neusmjerenog grafa imamo još i uvjet da je $k \geq 3$, što znači da ciklus mora "posjetiti" barem tri različita čvora. Na taj način se eliminira trivijalni ciklus (u, u', u) gdje se pomičemo naprijed-nazad po istoj vezi. Na slici imamo put $(1,2,3,1)$ koji je ciklus.

Graf koji nema niti jedan ciklus zovemo **necikličan** (acyclic).

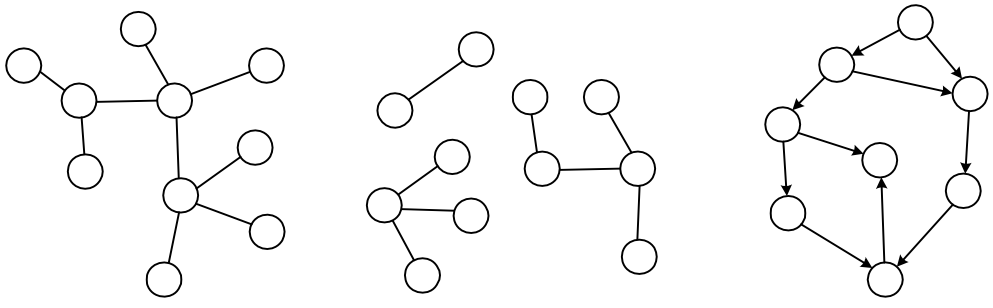
Često nas zanimaju dvije posebne klase ciklusa. Jedan je **Hamilton-ov ciklus** kod kojeg treba posjetiti svaki čvor u grafu točno jednom. **Euler-ov ciklus** je ciklus kod kojeg treba posjetiti svaku vezu u grafu točno jednom. (Postoje i varijante kod kojih govorimo ne o ciklusu nego o putu.)

Neusmjereni graf je **povezan** (connected) ako se svaki čvor može doseći iz svakog drugog čvora. Neciklični povezani graf zove se **slobodno stablo** (free tree) ili često samo **stablo** (tree). Izraz "slobodno" koristimo da se naglasi da stablo nema korijen, kao što je to uobičajeno kod strukture podataka

Povezane komponente grafa su oni čvorovi za koje vrijedi relacija povezanosti. Tako na slici imamo tri povezane komponente $(1,2,3)$, (4) i $(5,6)$. Povezani graf ima samo jednu povezanu komponentu. Ako se neciklični graf sastoji od više slobodnih stabala, zove se **šuma**.

Usmjereni graf je **strogo povezan** ako se bilo koja dva čvora mogu doseći jedan iz drugoga. (Postoji i slaba povezanost o kojoj nećemo govoriti). Slično kao i kod neusmjerenog grafa, i ovdje postoje **strogo povezane komponente**.

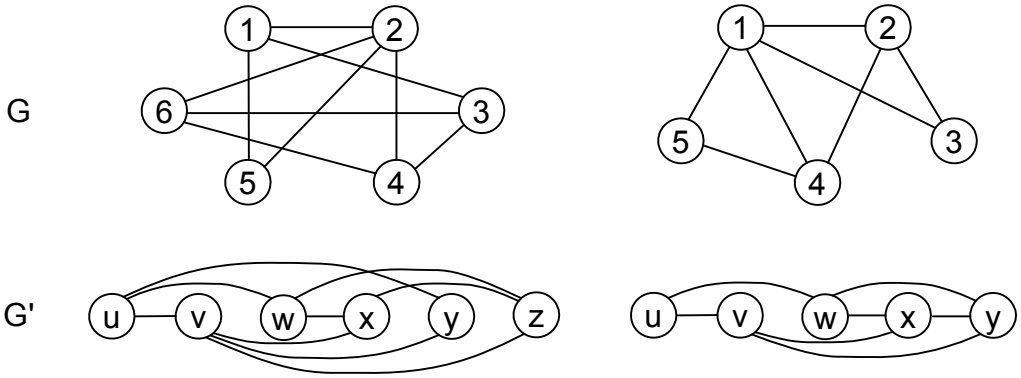
Za usmjereni neciklični graf često koristimo skraćenicu **dag** (Directed Acyclic Graph).



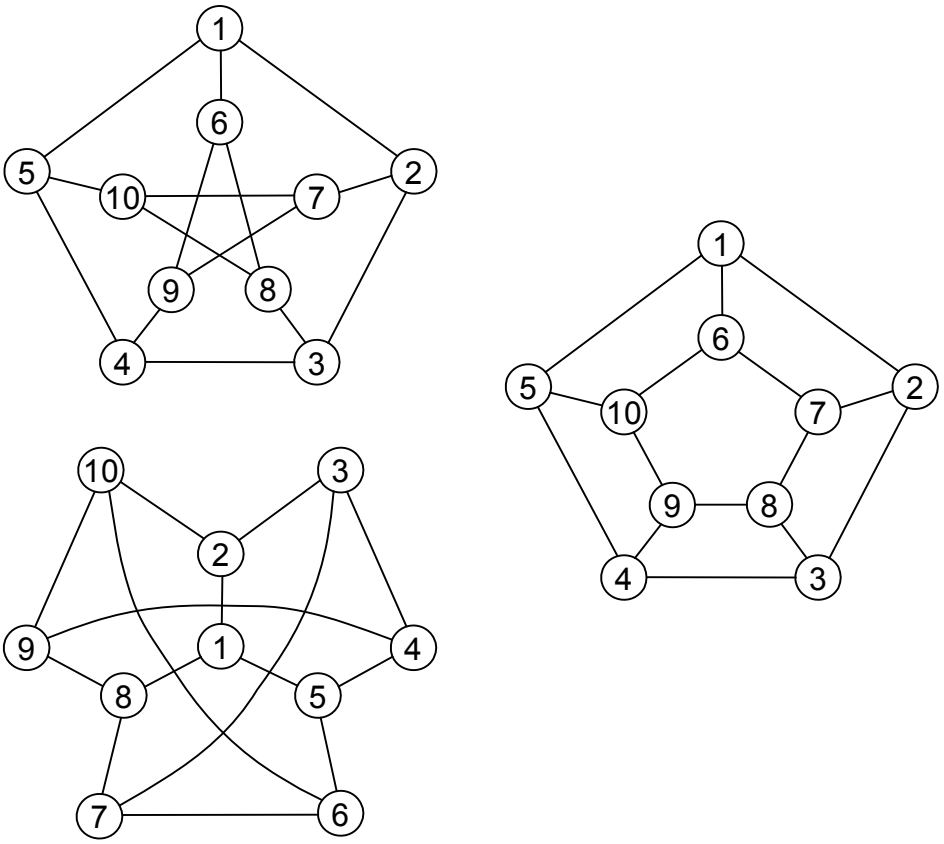
Slika: Slobodno stablo, šuma i dag

Za dva grafa $G=(V,E)$ i $G'=(V',E')$ kažemo da su **izomorfna** (isomorphic) ako postoji funkcija bijekcije $f: V \to V'$ takva da je $(u,v)\in E$ onda i samo onda ako je $(f(u),f(v))\in E'$. Izomorfni grafovi su u osnovi isti, osim što čvorovi imaju različite nazive.

Odrediti da li su grafovi izomorfni često puta nije lako. Na slici su prikazana dva para grafova G i G'. Skup čvorova, za lijevi par, jednak je $V=\{1,2,3,4,5,6\}$ i $V'=\{u,v,w,x,y,z\}$. Preslikavanje (tj. tražena bijektivna funkcija) iz V u V' je $f(1)=u$, $f(2)=v$, $f(3)=w$, $f(4)=x$, $f(5)=y$ i $f(6)=z$ pa su to izomorfni grafovi. Desni par grafova nije izomorfan. Iako oba grafa imaju po 5 čvorova i 7 veza, gornji graf ima čvor čiji je stupanj 4, dok donji graf nema takav čvor.



Slika: Parovi grafova

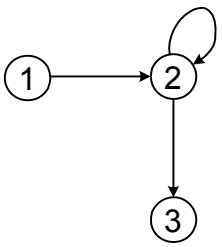


Slika: Grafovi (Da li su izomorfni? Koja je funkcija preslikavanja?)

Kažemo da je graf $G' = (V', E')$ **podgraf** (subgraph) grafa $G = (V, E)$ ako $V' \subseteq V$ i $E' \subseteq E$. Za dani $V' \subseteq V$, podgraf grafa G **uzrokovan** (induced) s V' je graf $G' = (V', E')$ za koji vrijedi

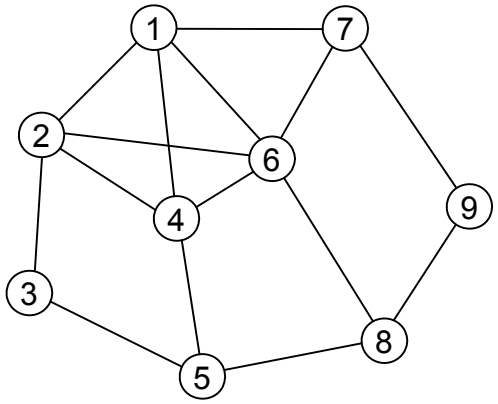
$$E' = \{(u, v) \in E : u, v \in V'\}$$

Drugim riječima, trebamo uzeti sve veze iz G koje povezuju parove čvorova u V'. Npr. na slici koja pokazuje usmjereni i neusmjereni graf, podgraf uzrokovan s čvorovima $\{1,2,3\}$ je prikazan na slijedećoj slici i ima skup veza $\{(1,2),(2,2),(2,3)\}$.



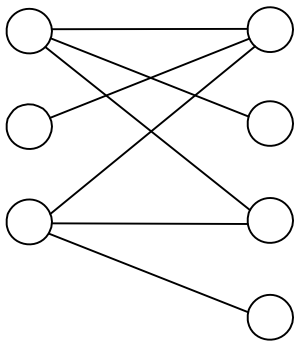
Slika: Graf

Nekoliko vrsta grafova dobili su posebna imena. Neusmjereni graf koji ima maksimalni mogući broj veza (tj. kod kojeg je svaki par čvorova susjedan) naziva se **potpuni** (complete) graf. Potpuni graf se obično označava sa slovom K (npr. K_5 je potpuni graf s 5 čvorova). Za dani graf G, kažemo da podskup čvorova $V' \subseteq V$ formira **kliku** (clique) ako je podgraf uzrokovan s V' potpun. Podskup čvorova V' formira **neovisni** (independent) skup ako podgraf uzrokovan s V' nema veza. Npr. podskup $\{1,2,4,6\}$ formira kliku, a $\{3,4,7,8\}$ neovisni skup.



Slika: Klika i neovisni graf

Dvodijelan (bipartitan (bipartite)) graf je neusmjereni graf $G = (V, E)$ kod kojeg se skup čvorova V može podijeliti na dva skupa V_1 i V_2 takva da $(u, v) \in E$ implicira ili $u \in V_1$ i $v \in V_2$ ili $u \in V_2$ i $v \in V_1$. Drugim riječima, sve veze idu između dva skupa V_1 i V_2.



Slika: Dvodijelni graf

Komplement (complement) grafa $G = (V, E)$, koji se označava s $\overline{G}$, je graf na istom skupu čvorova, ali kod kojeg su veze komplementi veza na grafu G. Drugim riječima, kada od potpunog grafa oduzmemo veze grafa G, preostale veze (zajedno s čvorovima) tvore komplement grafa. (Nađite komplement za prethodne grafove.)

Inverz (reversal) usmjerenog grafa je graf na istom skupu čvorova, ali s vezama čiji je smjer invertiran. Inverz usmjerenog grafa označava se s G^R. (Ponekad se naziva i **transponirani** (transpose) graf i označava s G^T.

Graf je **planaran** (planar) ako se može nacrtati na **ravnini** (plane) tako da ne postoje dvije veze koje prelaze jedna preko druge (koje se križaju). Planarni grafovi su specijalni slučaj grafova koji imaju veliko praktično značenje. Javljaju se kod zemljopisnih informacijskih sustava (GIS) (podjela područja na manja podpodručja), kod tiskanih pločica (gdje se žice ne smiju sjeći), kod modeliranja kompleksnih površina (primjerice nizom trokuta) i sl.

Vrijeme izvršavanja algoritama koji se temelje na grafovima ovisi o veličini grafa. S $|V|$ označavamo broj čvorova, a s $|E|$ broj veza. Za usmjereni graf vrijedi $|E| \le |V|^2 = O(|V|^2)$. Za neusmjereni graf vrijedi $|E| \le \binom{|V|}{2} = \frac{|V| \cdot (|V|-1)}{2} = O(|V|^2)$. Podsjetimo se da graf (i usmjereni i neusmjereni) može imati 0 veza. (Često puta je zanimljivo naći najmanji broj veza tako da graf bude povezan.)

Za graf kod kojeg je $|E| << |V|^2$ kažemo da je **rijedak** (sparse). Primjerice, važna klasa planarnih grafova je $|E| = O(|V|)$. Općenito su vrlo veliki grafovi uglavnom rijetki, jer u protivnom je vrijeme izvršavanja algoritama koji se temelje na njima neprihvatljivo veliko.

8.2. PRIKAZ GRAFOVA

Postoje dva standardna načina da se prikaže graf $G = (V, E)$: kao matrica susjedstva (adjecency matrix) i kao lista susjedstva (adjecency list).

Neka je $G = (V, E)$ usmjereni graf s $|V|$ čvorova i $|E|$ veza. Pretpostavit ćemo da su čvorovi grafa G indeksirani $\{1,2,...,|V|\}$.

Prikaz grafa pomoću matrice susjedstva

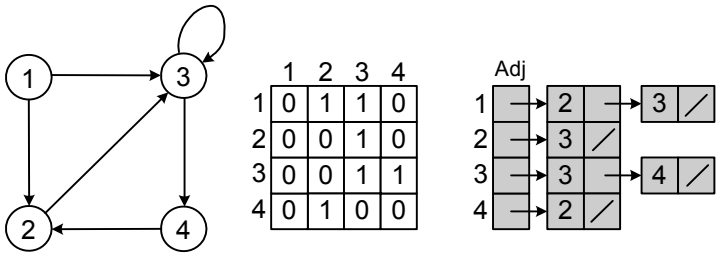
Prikaz grafa (usmjerenog) G pomoću matrice susjedstva sastoji se od $|V| \times |V|$ matrice A za koju vrijedi

$$A[v, w] = \begin{cases} 1 & \text{za } (v, w) \in E \\ 0 & \text{za ostale} \end{cases}$$

Ako je usmjereni graf otežan, u matrici ne moraju biti jedinice nego težine.

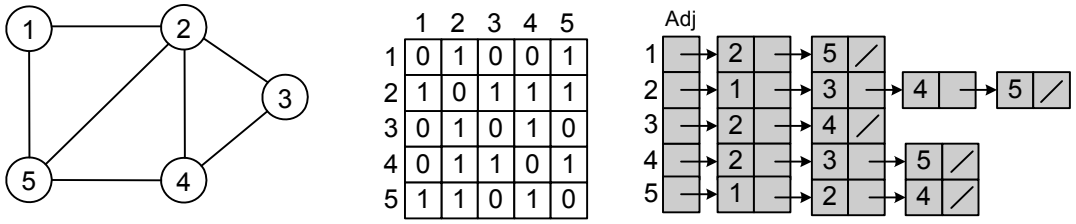
Prikaz grafa pomoću liste susjedstva

Prikaz grafa (usmjerenog) pomoću liste susjedstva sastoji se od polja Adj koje sadrži $|V|$ lista, jednu za svaki čvor. Za svaki $v \in V$, lista susjedstva $Adj[v]$ sadrži sve čvorove w takve da je veza $(v, w) \in E$. Drugim riječima, $Adj[v]$ sadrži sve čvorove susjedne čvoru v (to mogu biti i pointeri na čvorove). Ako su veze otežane, te težine mogu se također pohraniti u listi.



Slika: Prikaz usmjerenog grafa pomoću matrice susjedstva i liste susjedstva

Neusmjereni graf se može prikazati na potpuno isti način kao i usmjereni, s time da se svaka veza pohrani dva puta. Primjerice, neusmjerenu vezu (v, w) prikazujemo kao dvije usmjerene veze (v, w) i (w, v). Pri tome treba biti pažljiv, jer su u tom slučaju dvije jedinice u matrici susjedstva u stvarnosti samo jedna veza.



Slika: Prikaz neusmjerenog grafa pomoću matrice susjedstva i liste susjedstva

Matrica susjedstva za pohranu treba $\Theta(|V|^2)$ memorijskih jedinica. Lista susjedstva treba $\Theta(|V| + |E|)$. Za rijetke grafove lista susjedstva očito treba manje memorije nego matrica susjedstva. S druge strane za guste (dense) grafove bit će prikladniji prikaz preko matrice susjedstva. Matrica susjedstva je pogodan izbor i kada trebamo brzi odgovor na pitanje da li između dva čvora postoji veza.

8.3. PRETRAŽIVANJE PO ŠIRINI

Najkraći putovi

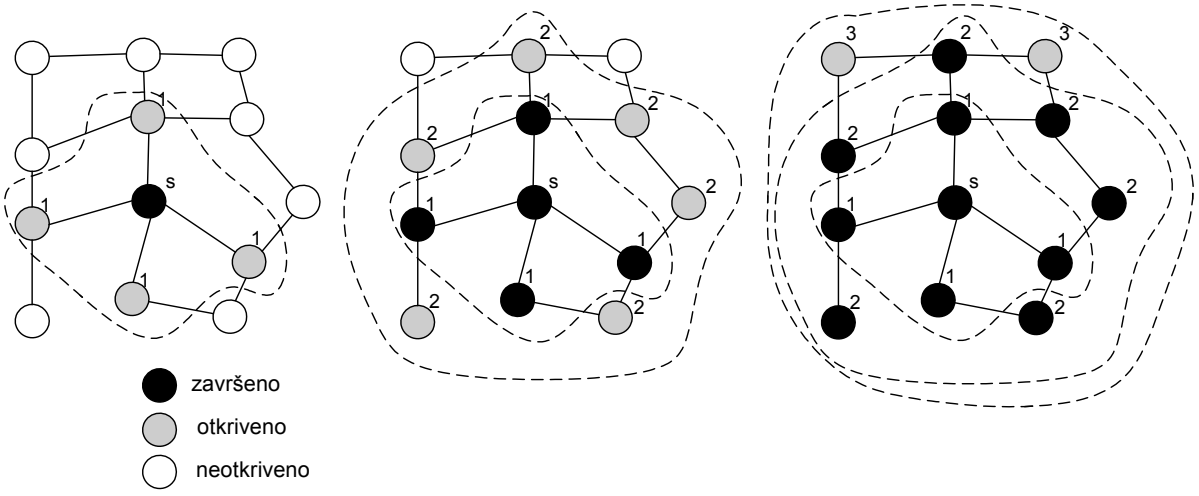
Razmotrimo slijedeći problem (koji će nas dovesti do prvog algoritma temeljenog na grafovima). Neka je dan neusmjereni graf $G = (V, E)$ i početni čvor (source vertex) $s \in V$. Dužina puta u grafu (koji nije otežan) je broj veza na putu. Mi želimo pronaći najkraći put od čvora s do svakog drugog čvora u G. Ako imamo dva (ili više) najkraćih putova iste dužine, onda jedan od njih možemo odabrati na slučajan način.

Konačni rezultat prikazat ćemo na slijedeći način. Za svaki čvor $v \in V$, pohranit ćemo distancu $d[v]$, tj. dužinu najkraćeg puta od s do v. (Uočite da je $d[s] = 0$.) Također, pohranit ćemo prethodni pointer (predecessor pointer, parent pointer) $\pi[v]$ koji pokazuje na prvi čvor duž najkraćeg puta unazad (od v prema s). Neka je $\pi[s] = \text{NULL}$.

Iako možda nije očigledno, taj jedan prethodni pointer, dovoljan je da se rekonstruira najkraću put do bilo kojeg čvora. Zašto?

Očito je da postoji strategija koja se temelji na gruboj sili pomoću koje možemo odrediti najkraće putove. Možemo isprobati sve (jednostavne) putove iz s u svaki od čvorova, te potom za svaki čvor uzeti onaj najkraći. Međutim, takvih (jednostavnih) putova u grafu može biti $|V-1|!$ (Kada?) pa takva strategija sasvim sigurno nije dobra.

Razmotrimo sada strategiju koja je bitno učinkovitija. Krenimo iz početnog čvora s. Očito je da će distanca do svih susjednih čvorova biti 1. Označimo sve te (susjedne) čvorove s njihovom distancom, u ovom slučaju je to 1. Promotrimo dalje sve susjede tih susjeda koji nemaju već određenu distancu. Oni će imati distancu 2 (od čvora s). Dalje promatramo susjede od susjeda od susjeda i tako dalje sve dok ne posjetimo sve susjede.



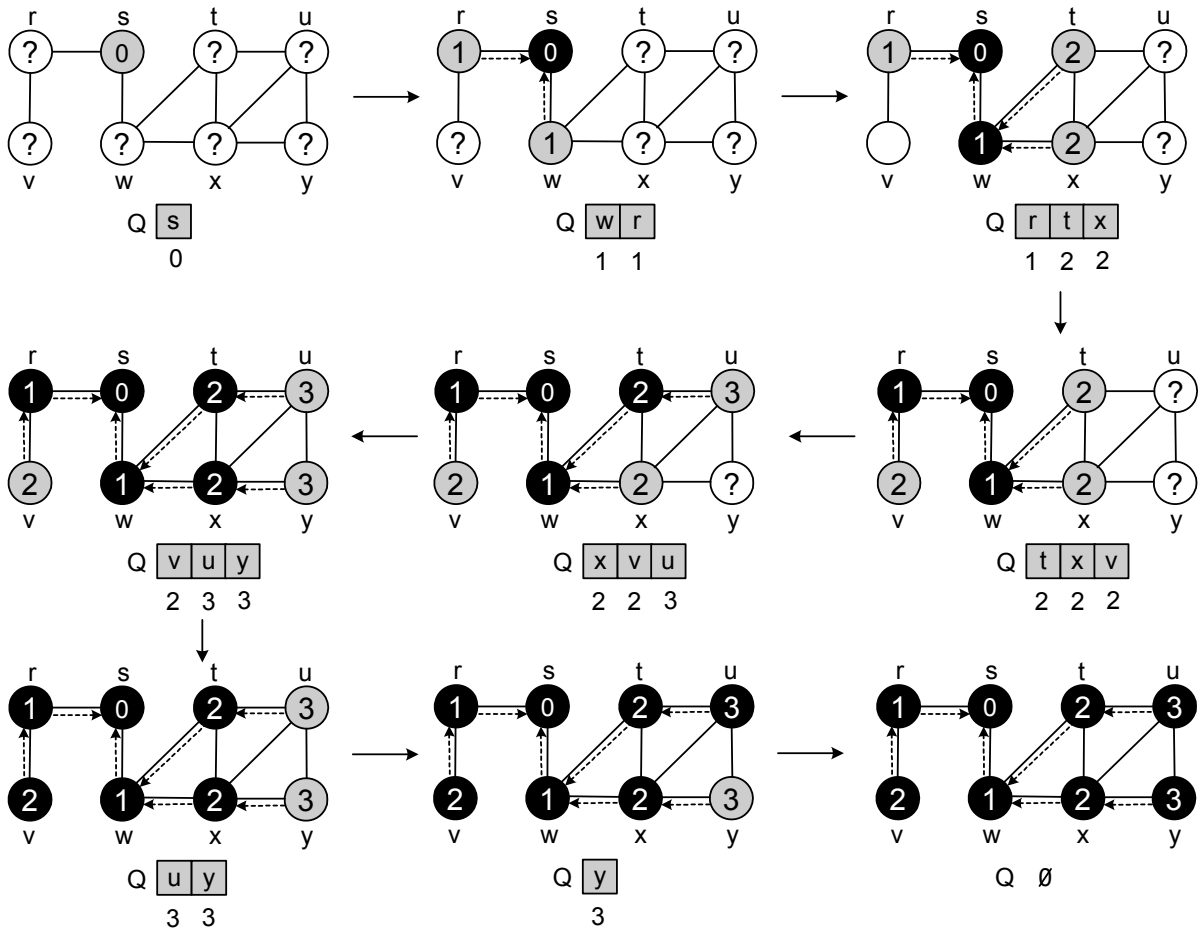
Slika: Najkraći putovi metodom pretraživanja po širini

Pretraživanje po širini

Za dani graf $G = (V, E)$, pretraživanje po širini (breadth-first search, BFS) počinje na početnom čvoru s te traži sve čvorove dohvatljive iz s. Pretraživanje po širini dolazi do (otkriva) čvorova po rastućim vrijednostima distance, što znači da takav pristup možemo koristiti za traženje najkraćih putova. U svakom trenutku postoji granica koja obuhvaća sve čvorove koji su otkriveni. Pretraživanje po širini dobilo je ime upravo zbog toga što čvorove posjećuje (obrađuje) duž cijele granice odnosno po cijeloj širini granice.
Inicijalno su svi čvorovi (osim izvorišnog čvora) obojani bijelo, što znači da još nisu otkriveni. Kada se čvor otkrije on postaje dio granice te se oboji u sivo. Kada se sivi čvor obradi on postaje crn.

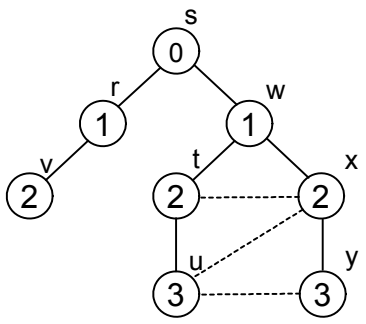
```
BFS (G,s)                    //definiramo algoritam BFS na grafu G i izvorišnom čvoru s
    int distanca[1..size(V)]          //distance čvora
    int boja[1..size(V)]              //boje čvora
    čvor prethodni[1..size(V)]        //prethodni pointeri
    queue Q=empty                     //FIFO queue
    za svaki u iz V                   //inicijalizacija
        boja[u] = bijela
        distanca[u] = INF
        prethodni[u] = NULL
    boja[s] = siva                    //definiranje izvora
    distanca[s] = 0
    enqueue(Q,s)                      //stavi izvor u queue
    while (Q is nonempty)
        u = dequeue (Q)               //u je slijedeći čvor kojeg ćemo posjetiti
        za svaki v iz Adj[u]
            if (boja[v] == bijela)        //ako susjed još nije otkriven
                boja[v] = siva
                distanca[v] = distanca[u] + 1
                prethodni[v] = u
                enqueue(Q,v)
        boja[u] = crna                            //završena je obrada u
```

Pretraživanje po širini koristi queue, dakle FIFO (first-in first-out) listu, u kojoj se elementi brišu istim redoslijedom kako se i unose (enqueue je ekvivalent s insert, dequeue s delete).



Slika: Primjer pretraživanja po širini

Prethodni pointeri definiraju invertirano stablo koje zovemo BFS stablo za graf G. Postoji puno potencijalnih BFS stabala za dani graf, ovisno o tome gdje pretraživanje počinje i po kojem redoslijedu se čvorovi stavljaju u queue. Veze koje se nalaze u BFS stablu nazivaju se veze stabla (tree edges), a preostale veze unakrsne veze (cross edges). Kod neusmjerenog grafa, unakrsne veze su uvijek između dva čvora koja se razlikuju za najviše jednu razinu.



Slika: BFS stablo

Analiza vremena izvršavanja

Najprije uočimo da dio algoritma kojim vršimo inicijalizaciju treba $\Theta(|V|)$ vremena. Nakon inicijalizacije niti jedan čvor se više neće obojiti u bijelo što znači da će se svaki čvor ubaciti u queue (i izbaciti iz queue) samo jednom (zbog *if (boja[v] == bijela)*). Kako niti jedan čvor ne posjećujemo više od jednom, slijedi da će broj prolazaka kroz while petlju biti najviše $|V|$ (odnosno točno $|V|$ ako je svaki čvor dohvatljiv iz ishodišnog čvora). Odavde slijedi da se svaka lista susjedstva prelazi najviše jednom. Budući da je suma dužina svih lista susjedstva $\Theta(|E|)$, slijedi da je ukupno vrijeme provedeno u prelaženju lista susjedstva jednako $O(|E|)$.

Ako uzmemo u obzir i vrijeme inicijalizacije slijedi da je vrijeme izvršavanja BFS jednako $O(|V|+|E|)$. To je vrijeme koje je proporcionalno s veličinom prikaza grafa preko liste susjedstva.

8.4. NAJKRAĆI PUTOVI SVIH PAROVA (ALL-PAIRS SHORTEST PATHS)

Već smo vidjeli kako odrediti najkraći put počevši iz definiranog ishodišnog čvora i to pod pretpostavkom da veze nisu otežane. Sada ćemo generalizirati taj problem. Naći ćemo najkraće putove iz svakog čvora u grafu, dakle pronaći ćemo najkraći put za svaki par čvorova. Osim toga problem ćemo generalizirati na način da ćemo dozvoliti da veze budu otežane.

Neka je $G=(V,E)$ usmjereni, otežani graf. Ako je $(u,v)\in E$ veza u grafu G, tada težinu te veze označavamo s $W(u,v)$. (Kako smo to već vidjeli, težina može biti primjerice udaljenost između gradova, cijena prijevoza i sl.) Pretpostavit ćemo da je težina pozitivna vrijednost, iako algoritam koji će biti opisan može primati u specijalnim slučajevima i negativne vrijednosti. Za dani put $P=(v_0,v_1,...v_k)$, cijena puta (cost) je suma otežanih veza

$$\cos t(\pi)=W(v_0,v_1)+W(v_1,v_2)+...+W(v_{k-1},v_k)=\sum_{i=1}^{k}W(v_{i-1},v_i)$$

Podsjetimo se da se izraz duljina (length) koristi kada govorimo o broju veza na putu). Udaljenost (distance) između dva čvora jednaka je minimalnoj cijeni (po svim putovima koji povezuju ta dva čvora).

Mi ćemo promatrati problem određivanja najmanje cijene (udaljenosti) između svih parova čvorova u otežanom usmjerenom grafu. Opisat ćemo dva algoritma. Prvi je $\Theta(|V|^4)$, a drugi

$\Theta\left(|V|^3\right)$. Oba algoritma temelje se na tehnici dizajniranja algoritama koju do sada nismo spominjali: to je dinamičko programiranje (dynamic programming). Drugi algoritam ($\Theta\left(|V|^3\right)$) zove se Floyd-Warshall algoritam.

Kod oba ova algoritma pretpostavit ćemo da je usmjereni graf prikazan pomoću matrice susjedstva, a ne pomoću uobičajenije liste susjedstva. Podsjetimo se da je lista susjedstva obično bolja za rijetke grafove (a veliki grafovi teže da budu rijetki). Međutim, u ovom slučaju problem vrlo brzo vodi prema gustom grafu. (Potrebno je pohraniti informaciju o udaljenosti između svakog para čvorova, a osim toga obično se skoro svaki čvor može doseći iz skoro svih drugih čvorova.) Stoga, budući je graf i ovako najčešće gust (misli se na graf udaljenosti), nije problem koristiti matricu susjedstva umjesto liste susjedstva.

Neka je $G = (V, E, W)$ ulazni usmjereni graf s čvorovima, vezama i težinom veza. Težina veza može biti pozitivna, nula ili negativna, ali uz uvjet da ne postoji takav ciklus čija je ukupna težina negativna. Zašto bi nam negativna težina ciklusa stvorila problem?

Format ulaza

Ulaz nam je matrica težina veza. Dimenzija matrice je $|V| \times |V|$, a temelji se (jednaka je) na vrijednostima težina usmjerenog grafa. Označimo s $w_{i,j}$ vrijednosti ulazne matrice W:

$$w_{i,j} = \begin{cases} 0 & \text{ako je } i = j \\ W(i,j) & \text{ako je } i \neq j \text{ i } (i,j) \in E \\ +\infty & \text{ako je } i \neq j \text{ i } (i,j) \notin E \end{cases}$$

Stavljanje beskonačne vrijednosti ($w_{i,j} = \infty$) tamo gdje nema veza intuitivno znači da ne postoji direktna veza između ta dva čvora, pa je stoga i cijena direktnog prelaska iz jednog čvora u drugi beskonačna. Nulu ($w_{i,j} = 0$) stavljamo po dijagonali jer, opet intuitivno, cijena putovanja iz jednog čvora u taj isti čvor jednaka je nuli. Uočite da kod usmjerenih grafova možemo imati veze iz jednog čvora u taj isti čvor. U tom slučaju $W(i,i)$ općenito ne mora biti jednak 0. Negativno ne može biti (u tom slučaju bi imali negativni ciklus, za što smo rekli da ne smije postojati). Ako je takva veza pozitivna, nama je nekorisna. Naime, takav put sigurno nikada nećemo izabrati, jer povećava cijenu, a ne vodi nas tamo gdje nismo bili.

Format izlaza

Izlaz je $|V| \times |V|$ matrica udaljenosti $D = d_{i,j}$, gdje je $d_{i,j} = \delta(i,j)$ udaljenost (cijena najkraćeg puta) iz čvora i u čvor j. Također ćemo promatrati obnavljanje (rekonstrukciju) (recovering) najkraćeg puta. Da bi to mogli napraviti definirat ćemo pomoćnu matricu $pred[i,j]$. Vrijednost $pred[i,j]$ sadržavat će čvor koji je negdje duž najkraćeg puta između čvorova i i j. Ako najkraći put između i i j ne prolazi niti jednim drugim čvorom, tada vrijedi $pred[i,j] = \text{NULL}$. Vidjet ćemo kasnije da se korištenjem tih vrijednosti može rekonstruirati najkraći put u $\Theta(|V|)$ vremenu.

Dinamičko programiranje za računanje najkraćeg puta

Dinamičkim programiranjem tipično rješavamo optimizacijske probleme. To je tehnika koja ima neke sličnosti s tehnikom podijeli pa vladaj, jer dijeli probleme na manje potprobleme koji se onda rješavaju rekurzivno. Temeljni elementi koji karakteriziraju dinamičko programiranje su

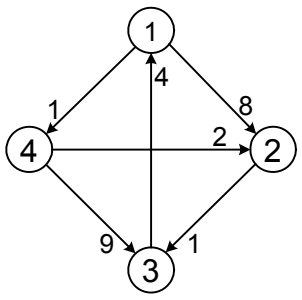
Rastavi problem na manje potprobleme.
Svaki od potproblema riješi optimalno.
Kombiniraj rješenja malih potproblema da bi riješili veće potprobleme, i eventualno došli do rješenja cijelog problema.

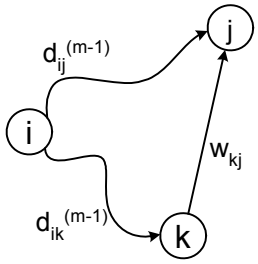
Pitanje je kako, na smislen način, rastaviti problem najkraćeg puta na potprobleme. Postoji jedan način koji je vrlo prirodan, ali ne vodi najboljem rješenju. Drugi pristup (Floyd-Warshall algoritam) nije tako očit, ali vodi boljem rješenju. Mi ćemo obraditi oba ova načina.

Formulacija duljine i udaljenosti puta

Usredotočit ćemo se na računanje cijene najkraćeg puta, a ne samog puta. Opišimo ukratko postupak prirodne podjele našeg problema na potprobleme. Željeli bismo pronaći takav parametar koji će ograničiti računanje cijene najkraćeg puta. U početku bi rezultat trebao biti grub (približan). Povećavanjem parametra, procjena najmanje cijene (udaljenosti) trebala bi težiti točnoj vrijednosti. Prirodni način da se to napravi je da se ograniči broj veza za koje je dozvoljeno da budu u najkraćem putu.

Za $0 \le |E| \le |V| - 1$, definirajmo $d_{ij}^{(m)}$ - najmanju cijenu od čvora i do čvora j koja sadrži najviše m veza. Neka je $D^{(m)}$ matrica čiji su koeficijenti upravo te vrijednosti. Ideja je da prvo izračunamo $D^{(0)}$, potom $D^{(1)}$ pa sve do $D^{(|V|-1)}$. Budući da znamo da niti jedan put s najmanjom cijenom neće imati više od $|V| - 1$ veza (u suprotnom bi više puta prolazio kroz isti čvor), znamo da je $D^{(|V|-1)}$ konačna matrica udaljenosti.





$d_{1,3}^{(1)} = \text{INF}$ (nema puta)

$d_{1,3}^{(2)} = 9$ (1,2,3)

$d_{1,3}^{(3)} = 4$ (1,4,2,3)

Pitanje je kako računati ove matrice udaljenosti? Mogli bi početi s putom koji sadrži 0 veza tj. s $D^{(0)}$. Međutim, to je trivijalan slučaj. Osim toga vrlo je lako krenuti i s putom koji sadrži 1 vezu ($D^{(1)}$). Vrijedi $D^{(1)} = W$, jer sve veze u usmjerenom grafu možemo smatrati putovima duljine 1. Kako nam je W poznat (jer je to ulaz), možemo lako početi s $D^{(1)}$. Stoga za naš osnovni slučaj vrijedi početni uvjet

$$d_{ij}^{(1)} = w_{ij}$$

Sad, kada smo definirali početni uvjet, trebamo induktivno dalje računati matrice. Drugim riječima trebamo pokazati tvrdnju da se $D^{(m)}$ može izračunati iz $D^{(m-1)}$ za svaki $m \geq 2$. Razmotrimo kako računati $d_{ij}^{(m)}$. To je najmanja cijena (udaljenost) puta od i do j koristeći najviše m veza. Postoje dva slučaja:

1. Ako put s najmanjom cijenom koristi manje od m veza, tada je cijena jednaka $d_{i,j}^{(m-1)}$.
2. Ako put s najmanjom cijenom koristi točno m veza, tada taj put koristi $m-1$ vezu da dođe od čvora i do nekog čvora k, te potom slijedi jedna jedina veza (k, j) težine w_{kj} kako bi se došlo do čvora j. Put od i do k, po načelu optimalnosti, treba biti s najmanjom cijenom, pa je cijena rezultirajućeg puta jednaka $d_{ik}^{(m-1)} + w_{kj}$. Naravno, mi ne znamo koji je k u pitanju, pa do njega dolazimo minimiziranjem po svim mogućim čvorovima.

Ovo razmatranje sugerira slijedeći izraz

$$d_{ij}^{(m)} = \min\begin{cases} d_{ij}^{(m-1)} \\ \min\limits_{1 \leq k \leq |V|} \left(d_{ik}^{(m-1)} + w_{kj} \right) \end{cases}$$

Dva dijela ovog izraza odnose se na dva spomenuta slučaja. U drugom slučaju, razmatramo sve čvorove k. Kada izaberemo čvor k, onda promatramo put s najmanjom cijenom od i do k (za koji treba m-1 veza) na koji dodajemo cijenu veze (s jednom vezom) od k do j.
Formulu možemo pojednostavniti tako što ćemo uzeti u obzir da je $w_{jj} = 0$, pa imamo da je $d_{ij}^{(m-1)} = d_{ij}^{(m-1)} + w_{jj}$. To se javlja u drugom slučaju kada je $k = j$, pa je prvi slučaj redundantan. Iz ovog razmatranja slijedi

$$d_{ij}^{(m)} = \min_{1 \leq k \leq |V|} \left(d_{ik}^{(m-1)} + w_{kj} \right)$$

Sada se postavlja pitanje kako implementirati ovaj izraz, odnosno ovo razmišljanje. Jedan način je rekurzivna procedura. Da bi izračunali najkraći put od i do j, poziv treba biti Dist(n-1,i,j)

```
Dist (int m, int i, int j)
    if (m==1) return W[i,j]                    //slučaj jedne veze
    najbolji = INF
    for k = 1 to n do                          //n je ukupan broj čvorova
```

```
            najbolji = min (najbolji, Dist(m-1, i, k) + w[k,j])
        return najbolji
```

Nažalost, ovo je algoritam koji je izuzetno spor. Neka je $T(m,|V|)$ vrijeme izvršavanja ovog algoritma za graf koji ima $|V|$ čvorova, a čiji je prvi argument m (podsjetimo se – to znači naći put s najmanjom cijenom koji ima najviše m veza). Algoritam radi $|V|$ poziva na samog sebe s prvim argumentom $m-1$. Kada je $m=1$, rekurzija završava, pa u tom trenutku imamo $T(1,|V|)=1$. U svim drugim slučajevima radimo $|V|$ rekurzivnih poziva $T(m-1,|V|)$. To nam daje rekurziju

$$T(m,|V|)=\begin{cases}1 & \text{ako je } m=1\\ |V|T(m-1,|V|)+1 & \text{za ostale}\end{cases}$$

Nas zanima vrijeme izvršavanja algoritma kada je prvi argument $|V|-1$, tj. tražimo $T(|V|-1,|V|)$. Iz gornje rekurzije može se vidjeti vrlo lako da je rezultat $O(|V|^{|V|})$, što je izuzetno velika vrijednost. Naime, ako malo pažljivije promotrimo algoritam, vidjet ćemo da se isprobavaju svi mogući putovi od i do j. Broj takvih putova je eksponencijalno velik.

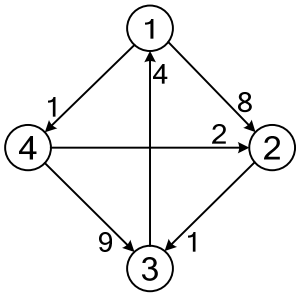
Kako bi ovaj postupak mogli ubrzati. Odgovor je u korištenju pretraživanja tablice (table look-up). To je ključ dinamičkog programiranja. Uočite da postoji samo $O(|V|^3)$ različitih mogućih vrijednosti $d_{ij}^{(m)}$ koje moramo izračunati. Kada jednom izračunamo neku od tih vrijednosti, pohranjujemo je u tablicu. Nakon toga, ako nam ponovo zatreba ta vrijednost, jednostavno je potražimo u tablici.

Glavna procedura NajkraciPut(n,W) treba broj čvorova $|V|$ (n) i matricu težina veza W. Matrica $D^{(m)}$ je pohranjena kao $D[m]$ za $1 \le m \le n-1$. Za svaki m, $D[m]$ je dvodimenzionalna matrica, iz čega slijedi da je D trodimenzionalna matrica. $D^{(1)}$ inicijaliziramo tako što ga izjednačimo s W. Nakon toga svaki poziv ProduzeniPut() računa $D^{(m)}$ iz $D^{(m-1)}$.

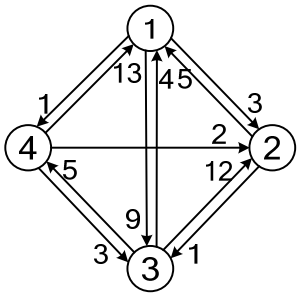
```
NajkraciPut (int n, int W[1..n, 1..n]
        array D[1..n-1][1..n, 1..n]
        kopiraj W u D[1]                                    //inicijaliziranje D[1]
        for m = 2 to n-1 do
                D[m] = ProduzeniPut(n, D[m-1], W)           //računanje D[m] iz D[m-1]
        return D[n-1]

ProduzeniPut(int n, int d[1..n, 1..n], int W[1..n, 1..n])
        matrix dd[1..n, 1..n] = d[1..n, 1..n]               //kopiraj d u privremenu matricu
        for i= 1 to n do
                for j = 1 to n do
                        for k = 1 to n do
                                dd[i,j] = min(dd[i,j], d[i,k] + W[k,j])
```

return dd *//vrati matricu cijena*

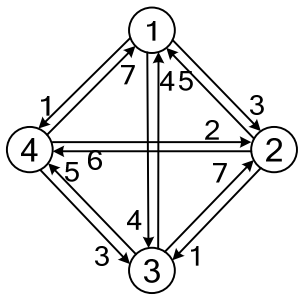
Procedura ProduzeniPut() sastoji se od 3 ugniježđene petlje koje idu od jedan do n, pa je vrijeme izvršavanja jednako $\Theta(n^3)$. Ta procedura se poziva $n-2$ puta, pa je ukupno vrijeme izvršavanja jednako $\Theta(n^4)$. Ako se vratimo na prije korištene oznake to bi bilo $\Theta(|V|^4)$.



$$D^{(1)} = \begin{bmatrix} 0 & 8 & INF & 1 \\ INF & 0 & 1 & INF \\ 4 & INF & 0 & INF \\ INF & 2 & 9 & 0 \end{bmatrix} = W$$



$$D^{(2)} = \begin{bmatrix} 0 & 3 & 9 & 1 \\ 5 & 0 & 1 & INF \\ 4 & 12 & 0 & 5 \\ 13 & 2 & 3 & 0 \end{bmatrix}$$



$$D^{(3)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

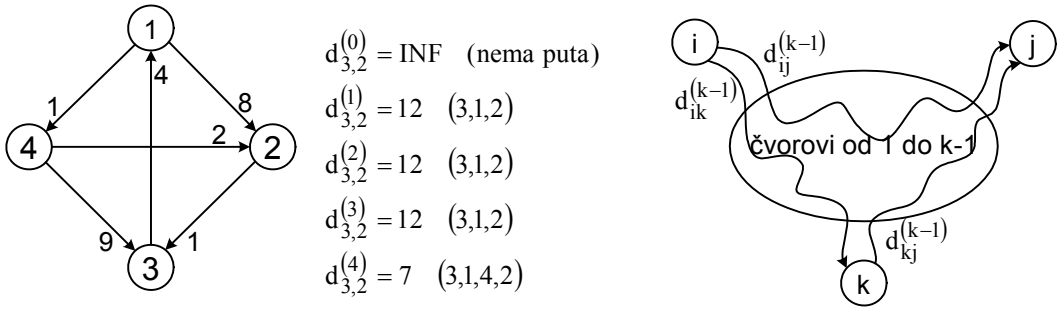
Slika: Primjer traženja najkraćeg puta

8.5. FLOYD-WARSHALL ALGORITAM

Floyd-Warshall-ovim algoritmom računamo najmanju cijenu između svih parova čvorova u usmjerenom grafu. Floyd-Warshall-ovim algoritmom to je moguće napraviti u $\Theta\left(|V|^3\right)$ vremenu. Podsjetimo se, u prethodnom smo primjeru vidjeli kako se to može napraviti u $\Theta\left(|V|^4\right)$ vremenu. Floyd-Warshall algoritam, u odnosu ne prethodno formulirani, uvodi drugačiju formulaciju problema najkraćeg puta. I upravo ta drugačija formulacija, rezultirala je boljim (bržim) algoritmom. (Iako na prvi pogled ta formulacija može izgledati neprirodna.)

Kao i u prethodnom primjeru, računat ćemo niz matrica čije su vrijednosti $d_{ij}^{(k)}$. Samo što ćemo u ovom slučaju promijeniti značenje tih vrijednosti. (Ponekad se Floyd-ov i Warshall-ov algoritam promatraju posebno. Floyd-ov algoritam je zapravo poopćenje Warshall-ovog algoritma, ali u načelu su to dva ista algoritma.)

Za dani put $p = (v_1, v_2, ..., v_{m-1}, v_m)$ kažemo da su čvorovi $v_2, v_3 ..., v_{m-1}$ prijelazni čvorovi (međučvorovi) (intermediate vertices). Uočite da put koji se sastoji od jedne veze nema prijelaznih čvorova. Definiramo $d_{ij}^{(k)}$ kao najkraći put iz i u j takav da je bilo koji prijelazni čvor na putu izabran iz skupa $\{1,2,\ldots,k\}$. Drugim riječima, put između i i j ima samo jednu vezu (i, j), ili su prijelazni čvorovi koje prijeđe na tom putu isključivo iz $\{1,2,\ldots,k\}$. Pri tome put može posjetiti bilo koji podskup tih čvorova i u bilo kojem redoslijedu. Uočite da smo u definiciji koju smo prije koristili imali $d_{ij}^{(m)}$, gdje nam indeks m ograničava broj veza na putu. Kod Floyd-Warshall-ovog algoritma imamo $d_{ij}^{(k)}$, gdje nam indeks k ograničava skup čvorova (a ne samo broj) kroz koje put može proći.



Slika: Ilustracija Floyd-Warshall-ovog algoritma

Kako izračunati matricu $d_{ij}^{(k)}$ ako znamo matricu $d_{ij}^{(k-1)}$? Kao i prije, imamo dva osnovna slučaja, ovisno o načinu na koji možemo doći iz čvora i u čvor j. (Naravno, uz pretpostavku da smo prijelazne čvorove izabrali iz skupa $\{1,2,\ldots,k\}$.

1. Slučaj kada uopće ne idemo preko čvora k. U tom slučaju najkraći put iz i u j koristi samo prijelazne čvorove iz skupa $\{1,2,\ldots,k-1\}$ pa je stoga najmanja cijena (udaljenost) puta $d_{ij}^{(k-1)}$.
2. Slučaj kada idemo preko čvora k. Najprije uočimo da put s najmanjom cijenom ne prelazi preko istog čvora dva puta pa možemo pretpostaviti da preko čvora k prelazimo točno jednom. (Naravno, ovo vrijedi uz pretpostavku da nema kruga čija bi cijena bila negativna.) Drugim riječima, idemo od čvora i do k, te potom od k do j. Da bi ukupan put imao najmanju cijenu, potrebno je izabrati put s najmanjom cijenom od i do k i put s najmanjom cijenom od k do j. Svaki od tih putova koristi prijelazne čvorove iz skupa $\{1,2,\ldots,k-1\}$ pa je ukupna cijena puta jednaka $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Ovo razmatranje sugerira slijedeću rekurziju za računanje $d_{ij}^{(k)}$

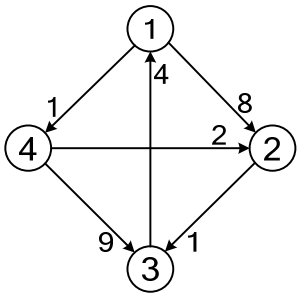
$$d_{ij}^{(0)} = w_{ij}$$
$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \quad \text{za } k \geq 1$$

Konačan odgovor (najmanju cijenu između svih parova čvorova) dobijemo za $d_{ij}^{(n)}$ jer se time prijelazni čvorovi mogu birati iz skupa svih mogućih čvorova. Kao i u prethodnom primjeru, i ovdje možemo napisati rekurzivni algoritam koji bi računao $d_{ij}^{(k)}$, ali bi opet bio neprihvatljivo spor. Umjesto rekurzivnog algoritma, koristit ćemo algoritam koji će pohranjivati vrijednosti u tablicu, te ih po potrebi koristiti (table look-up). U algoritmu ćemo koristiti i prethodni pointer (predecessor pointer) $\text{pred}[i, j]$ kako bi rekonstruirali konačni najkraći put.

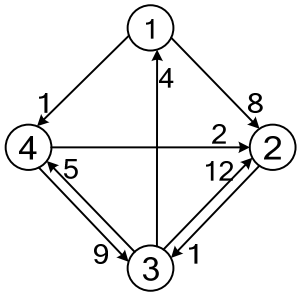
```
Floyd-Warshall (int n, int W[1..n, 1..n]
    array d[1..n, 1..n]
    for i = 1 to n do                                   //inicijalizacija
        for j = 1 to n do
            d[i,j] = W[i,j]
            pred[i,j]=NULL
    for k = 1 to n do                                   //korištenje prethodnih rezultata
        for i = 1 to n do                               // od i
            for j = 1 to n do                           // do j
                if (d[i,k] + d[k,j]) < d[i,j]           //ako je zadovoljeno - mijenjaj
                    d[i,j] = d[i,k] + d[k,j]            //nova vrijednost
                    pred[i,j] = k                       //novi put prolazi preko k
    return d                                            //vrati matricu konačnih cijena
```

Algoritam ima tri ugniježđene petlje od 1 do n, pa je očito da je vrijeme izvršavanja algoritma $\Theta(n^3)$. Memorijski prostor koji ovaj algoritam treba je $\Theta(n^2)$. (Podsjetimo se da je n broj

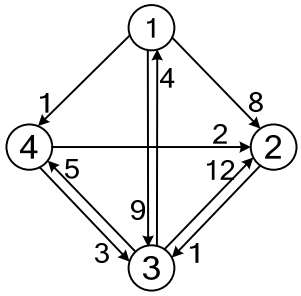
čvorova, tj. $n = |V|$). Uočite da nismo koristili referencu na (k). Drugim riječima, nemamo $d(i, j, k)$ (gdje k nije čvor već broj dozvoljenih čvorova) nego $d(i, j)$. Zašto to ne utječe na točnost algoritma?



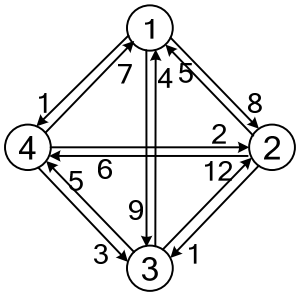
$$D^{(0)} = \begin{bmatrix} 0 & 8 & INF & 1 \\ INF & 0 & 1 & INF \\ 4 & INF & 0 & INF \\ INF & 2 & 9 & 0 \end{bmatrix} = W$$



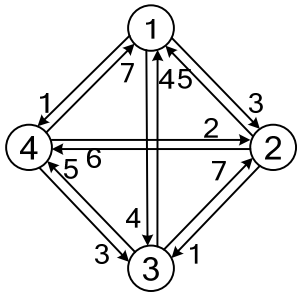
$$D^{(1)} = \begin{bmatrix} 0 & 8 & INF & 1 \\ INF & 0 & 1 & INF \\ 4 & 12 & 0 & 5 \\ INF & 2 & 9 & 0 \end{bmatrix}$$



$$D^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ INF & 0 & 1 & INF \\ 4 & 12 & 0 & 5 \\ INF & 2 & 3 & 0 \end{bmatrix}$$



$$D^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$



$$D^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Slika: Primjer Floyd-Warshall-ovog algoritma

Izvlačenje (rekonstrukcija) najkraćeg puta

Da bi izvukli konačni, najkraći put, služi nam prethodni pointer $pred[i, j]$. Ideja je slijedeća: svaki put kada otkrijemo da najkraći put od i do j prolazi preko prijelaznog čvora k, postavimo $pred[i, j] = k$. Ako najkraći put ne prolazi preko niti jednog prijelaznog čvora, tada je $pred[i, j] = NULLl$. Da bi našli najkraći put od i do j iskoristimo $pred[i, j]$. Ako je vrijednost $pred[i, j]$ jednaka NULL, tada je najkraći put jednostavno veza (i, j). U ostalim slučajevima rekurzivno računamo najkraći put od i do $pred[i, j]$ (npr. od i do k) i najkraći put od $pred[i, j]$ do j (npr. od k do j).

```
NajkraćiPut(i,j)
        if pred[i,j] = NULL                 //put je samo jedna veza
                ispisi(i,j)
        else                                //put prolazi kroz pred[i,j]
                NajkraćiPut(i,pred[i,j])    //ispisi put od i do pred[i,j]
                NajkraćiPut(pred[i,j],j)    //ispisi put od pred[i,j] do j
```

9. NAJDUŽA ZAJEDNIČKA PODSEKVENCA (LONGEST COMMON SUBSEQUENCE)

Nizovi (strings)

Jedno od značajnih područja primjene algoritama su nizovi karaktera gdje postoji cijeli niz zanimljivih problema. Primjerice, tu je problem traženja određenog niza u nekom tekstu (kao što se traži u editorima (priređivačima) teksta). Ponekad je zanimljivo ne samo naći točan tekst, već tekst koji je više ili manje sličan traženom. To je primjerice korisno kod istraživanja u genetici. Genetski kod je pohranjen kao dugačka DNA molekula. Kako su male promjene u DNA molekuli vrlo česte, to je slučaj da je DNA molekula potpuno ista, vrlo rijedak. Stoga je zanimljivo pronaći sličnost između nizova koji nisu potpuno isti. Naravno u tom nam je slučaju potrebna neka mjera sličnosti. Jedna takva mjera se temelji na minimalnom broju potrebnih umetanja, brisanja i premještanja karaktera potrebnih da jedan niz pretvorimo u drugi. Mjera sličnosti može biti i najduža zajednička podsekvenca. Upravo to je problematika koju ćemo obraditi.

Najduža zajednička podsekvenca

Neka su dane dvije sekvence $X = (x_1, x_2, \ldots, x_m)$ i $Z = (z_1, z_2, \ldots, z_k)$. Kažemo da je Z podsekvenca od X ako postoji striktno rastuća sekvenca k indeksa $(i_1, i_2, \ldots, i_k)$ $(1 \le i_1 < i_2 < \ldots < i_k)$ takva da je $Z = (x_{i1}, x_{i2}, \ldots, x_{ik})$. Na primjer, neka je $X = (\text{PONAVLJANJE})$ i neka je $Z = (\text{POVLJA})$, tada je Z podsekvenca od X.

Za dana dva niza X i Y, najduža zajednička podsekvenca od X i Y je najduža sekvenca Z koja je podeskvenca i od X i od Y.

Neka je npr. $X = (\text{PONAVLJANJE})$, a $Y = (\text{NAVIJANJE})$. U tom slučaju, najduža zajednička podsekvenca je $Z = (\text{NAVJANJE})$.

Problem najduže zajedničke podsekvence je slijedeći. Za dane dvije sekvence $X = (x_1, x_2, \ldots, x_m)$ i $Y = (y_1, y_2, \ldots, y_n)$ odredi najdužu zajedničku podsekvencu. Uočite da rješenje nije uvijek jedinstveno. Pronađite takav primjer!

Rješenje pomoću dinamičkog programiranja

Najjednostavnije rješenje problema najduže zajedničke podsekvence, je algoritam grube sile. Dakle, uzmi sve moguće podsekvence iz jednog niza, i ispitaj da li je ta podeskvenca ujedno i podsekvenca u drugom nizu. Naravno, takav pristup će rezultirati s algoritmom koji ima vrijeme izvršavanja neprihvatljivo veliko.

Umjesto toga dat ćemo rješenje koje se temelji na dinamičkom programiranju. To znači da trebamo naš problem podijeliti (razbiti) na manje dijelove. Postoji puno načina da se to napravi, ali se može pokazati da je dovoljno razmotriti sve parove prefiksa (prefix, prefixes).

Prefiks sekvence je početni niz vrijednosti. Za sekvencu $X = (x_1, x_2, \ldots, x_m)$, prefiks je $X_i = (x_1, x_2, \ldots, x_i)$ uz $i \le m$. (X_0 je prazna sekvenca.)

Ideja je da se izračuna najduža zajednička podsekvenca za svaki mogući par prefiksa. Neka je $c[i, j]$ dužina najduža zajedničke podsekvence od X_i i Y_j. (Nas u konačnici zanima $c[m, n]$, jer će to biti najduža zajednička posekvenca cijelog niza. Ideja je da izračunamo $c[i, j]$ pretpostavljajući da već znamo vrijednost $c[i', j']$ za $i' \le i$ i $j' \le j$ (ali jednakost ne smije biti u oba slučaja). Da bi primijenili dinamičko programiranje treba nam početni uvjet i način na koji ćemo iskoristiti prethodne rezultate.

Početni uvjet: $c[i,0] = c[0, j] = 0$, dakle ako je bilo koji niz prazan, tada je najduža zajednička podsekvenca prazna.

Zadnji karakter je isti. Neka je $x_i = y_j$. Primjerice, neka je $X_i = (\text{ABCA})$ i neka je $Y_j = (\text{DACA})$. Budući da oba završavaju s A, tvrdimo da najduža zajednička podsekvenca također mora zvršiti u A. (Za najdužu zajednički podsekvencu koristit ćemo oznaku LCS (Longest Common Subsequence) kao što je to uobičajeno u literaturi.) Budući je A dio LCS, ukupni LCS možemo naći tako da izbacimo A iz obje sekvence i onda tražimo LCS od $X_{i-1} = (\text{ABC})$ i $Y_{j-1} = (\text{DAC})$. U ovom slučaju LCS od X_{i-1} i Y_{j-1} je (AC) i tome trebamo dodati (na kraj) A što onda daje ukupni LCS da je jednak (ACA).

Možemo zaključiti: ako je $x_i = y_j$, slijedi da je $c[i, j] = c[i-1, j-1] + 1$

Zadnji karakter nije isti. Neka je $x_i \ne y_j$. U tom slučaju x_i i y_j ne mogu biti oba u LCS (jer moraju biti zadnji karakter u LCS). Slijedi da ili x_i nije dio LCS, ili y_j nije dio LCS (moguće je i da oba nisu dio LCS).

U prvom slučaju je LCS od X_i i Y_j jednak LCS od X_{i-1} i Y_j, što je $c[i-1, j]$. U drugom slučaju je LCS od X_i i Y_j jednak LCS od X_i i Y_{j-1}, što je $c[i, j-1]$. Mi ne znamo unaprijed koji je slučaj u pitanju, stoga probamo oba i uzmemo onaj koji rezultira dužim LCS.

Možemo zaključiti: ako je $x_i \ne y_j$, slijedi da je $c[i, j] = \max(c[i-1, j], c[i, j-1])$.

Sada možemo dati ukupni zaključak o dužini LCS:

$$c[i, j] = \begin{cases} 0 & \text{ako } i = 0 \text{ ili } j = 0 \\ c[i-1, j-1] + 1 & \text{ako } i, j > 0 \text{ i } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{ako } i, j > 0 \text{ i } x_i \ne y_j \end{cases}$$

Sada nam preostaje samo da primijenimo zaključak. Implementacija zaključka daje samo odgovor na pitanje koliko je duga najduža zajednička sekvenca. Stoga ćemo koristiti i polje b[0..m, 0..n] u koje ćemo pohranjivati informacije neophodne za rekonstrukciju same najduže zajedničke sekvence. U to polje pohranjivat ćemo informaciju da li je pomak *GOREiLIJEVO, GORE ili LIJEVO*. (To možemo smatrati prethodnim pointerom.)

LCS(char x[1..m], char y[1..n])

```
int c[0..m, 0..n]
for i = 0 to m do
        c[i,0] = 0                          //inicijalizacija stupca polja c
        b[i,0] = 0                          //inicijalizacija stupca polja b
for j = 0 to n do
        c[0,j] = 0                          //inicijalizacija retka polja c
        b[0,j] = 0                          //inicijalizacija retka polja b
for i = 1 to m do
        for j = 1 to n do
                if (x[i] == y[j])
                        c[i,j] = c[i-1, j-1] + 1
                        b[i,j] = GOREiLIJEVO
                else if (c[i-1,j] >= c[i,j-1]       //ako X[i] nije u LCS
                        c[i,j] = c[i-1, j]
                        b[i,j] = GORE
                else                                //ako Y[j] nije u LCS
                        c[i,j] = c[i, j-1]
                        b[i,j] = LIJEVO
return c[m,n]
```

Ovo je rješenje koje koristi dinamičko programiranje. Koristeći isti zaključak može se lako napraviti i rekurzivni algoritam (ali s eksponencijalnim vremenom izvršavanja).

Dinamičko programiranje, kao i metoda podijeli pa vladaj, rješava problem kombinirajući rješenja potproblema. Kao što smo već vidjeli, algoritmi temeljeni na tehnici podijeli pa vladaj dijele problem na neovisne potprobleme, rekurzivno riješe te potprobleme, te potom udruže rješenja potproblema kako bi riješili glavni problem. Suprotno tome, dinamičko programiranje koristimo kada potproblemi nisu neovisni (dakle, kada potproblemi dijele potpotprobleme). U tom slučaju algoritmi temeljeni na tehnici podijeli pa vladaj rade više posla nego je potrebno, rješavajući više puta zajedničke potpotprobleme. Algoritmi temeljeni na dinamičkom programiranju, riješe svaki potpotproblem samo jednom te njegovo rješenje pohrane. Time se rješenje može, kada zatreba, koristiti bez da se svaki put ponovo računa.

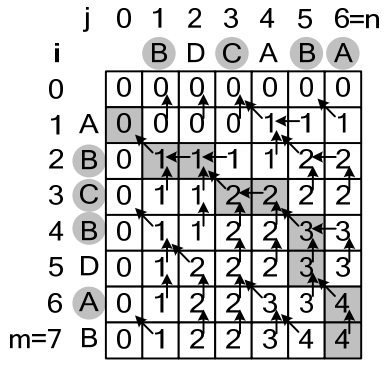
Dinamičko programiranje se tipično koristi za probleme optimiziranja. To su problemi kod kojih postoji puno mogućih rješenja, a cilj je pronaći ono rješenje koje je optimalno (koje ima najveću ili najmanju vrijednost).

	j	0	1	2	3	4	5	6=n
i			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
m=7	B	0	1	2	2	3	4	4

tablica dužina LCS

X=ABCBDAB
Y=BDCABA

LCS=BCBA



s uključenim prethodnim pointerom

Slika: Primjer računanja najduže zajedničke podsekvence

Vrijeme izvršavanja algoritma

Vrijeme izvršavanja algoritma je očito $O(mn)$ jer imamo dvije ugniježđene petlje s m i n iteracija. Unutar tih petlji vrijeme izvršavanja je konstantno. Memorija koju algoritam koristi je također $O(mn)$.

Izvlačenje najduže zajedničke sekvence

Samim algoritmom u načelu dobijemo samo dužinu LCS. Da bi izvukli najdužu zajedničku sekvencu trebaju nam prethodni pointeri (pohranjeni u polju b[0..m, 0..n]). $b[i, j] = \text{GOREiLIJEVO}$ znači da $X[i]$ i $Y[j]$ zajedno formiraju zadnji karakter u LCS. Uzmemo, dakle taj zajednički karakter i, i nastavimo gore i lijevo tj. $b[i-1, j-1]$. Ako je $b[i, j] = \text{GORE}$, tada znamo da $X[i]$ nije u LCS pa ga preskačemo i idemo gore tj. $b[i-1, j]$. Slično, ako je $b[i, j] = \text{LIJEVO}$, tada znamo da $Y[j]$ nije u LCS pa ga preskačemo i idemo lijevo tj. $b[i, j-1]$. Slijedeći povratni pointer, ispisujemo karakter svaki put kada se pomaknemo dijagonalno, čime dobijemo konačnu najdužu zajedničku podsekvencu.

```
IzvlačenjeLCS(char x[1..m], char y[1..n], int b[0..m, 0..n])
        LCS = prazan niz
        i = m
        j = n
        while (i != 0 && j != 0)
                switch b[i,j]
                case GOREiLIJEVO
                        dodaj x[i] u LCS                    //isto je ako dodamo y[j]
                        i--
                        j--
                        break
                case GORE
                        i--
                        break
                case LIJEVO
                        j--
                        break
        return LCS
```

10. MNOŽENJE LANCA MATRICA

Problem množenja lanca matrica je tipičan predstavnik velikog broja problema čiji je cilj odgovor na pitanje koji je optimalan redoslijed izvođenja niza operacija. (Ta je klasa problema prisutna primjerice kod optimiziranja koda ili optimiziranja upita kod baza podataka.) Problem množenja matrica rješavat ćemo dinamičkim programiranjem.
Pretpostavimo da želimo pomnožiti niz matrica

$$A_1, A_2, ... A_n$$

Množenje matrica je operacija koja je asocijativna, ali nije komutativna. To znači da u niz matrica možemo staviti zagrade kako god želimo, ali ne smijemo mijenjati redoslijed samih matrica. Također, ako želimo pomnožiti dvije matrice, tada su njihove dimenzije ograničene. Primjerice, ako matricu A s dimenzijama $p \times q$ pomnožimo s matricom B dimenzija $q \times r$, rezultat je nova matrica C čije su dimenzije $p \times r$. Drugim riječima, broj stupaca matrice A mora odgovarati broju redaka matrice B. Konkretno, za $1 \le i \le p$ i $1 \le j \le r$, vrijedi

$$C[i, j] = \sum_{k=1}^{q} A[i, k] B[k, j]$$

Uočite da postoji ukupno $p \cdot r$ elemenata u C i da svaki treba $O(q)$ vremena da bi ga izračunali. Stoga je ukupno vrijeme (tj. broj množenja) potrebno da se pomnože ove dvije matrice jednako $p \cdot r \cdot q$.

Iako bilo koji raspored zagrada vodi točnom rješenju, broj potrebnih operacija se razlikuje. Neka imamo tri matrice: A_1 dimenzije 5×4, A_2 dimenzije 4×6 i A_3 dimenzije 6×2. Mogući rasporedi zagrada i pripadni broj operacija su

$$\text{br\_množenja}[(A_1 A_2) A_3] = (5 \cdot 4 \cdot 6) + (5 \cdot 6 \cdot 2) = 180$$
$$\text{br\_množenja}[A_1 (A_2 A_3)] = (4 \cdot 6 \cdot 2) + (5 \cdot 4 \cdot 2) = 88$$

I na ovom jednostavno primjeru, imamo veliku razliku u vremenu izvođenja. Razlika može biti još veća kod množenja većeg broja matrica.

Problem množenja niza matrica možemo definirati na slijedeći način: Za dani niz matrica $A_1, A_2, ... A_n$ i pripadnih dimenzija $p_0, p_1, ... p_n$ gdje A_i ima dimenziju $p_{i-1} \times p_i$, odredi redoslijed množenja koji minimizira broj operacija. (Uočite da ne tražimo algoritam koji će množiti matrice, nego algoritam koji će naći najbolji redoslijed množenja.)

Rješenje problema pomoću dinamičkog programiranja

Želimo podijeliti problem na potprobleme, čija se rješenja mogu kombinirati kako bi se dobilo rješenje cijelog problema. S $A_{i..j}$ označit ćemo umnožak matrica od A_i do A_j. Može se lako vidjeti da je $A_{i..j}$ matrica dimenzije $p_{i-1} \times p_j$.

Pretpostavimo da smo podijelili cijeli problem na način da množimo samo dvije matrice. Dakle, za bilo koji k za koji vrijedi $1 \le k \le n - 1$ imamo

$$A_{1..n} = A_{1..k} A_{k+1..n}$$

Sada je problem određivanja optimalnog redoslijeda sveden na dva pitanja (problema): kako odrediti gdje razbiti niz matrica (koji je k?) i kako podijeliti dva novodobivena niza matrica $A_{1..k}$ i $A_{k+1..n}$. Drugi problem možemo riješiti primjenjujući rekurzivno istu logiku. Prvi problem možemo riješiti tako da ispitamo sve moguće vrijednosti k.

Rješenja problema pohranit ćemo u tablicu. Za $1 \le i \le j \le n$, neka je $m[i, j]$ minimalni broj množenja potreban da se izračuna $A_{i..j}$.

Treba nam početi uvjet i logika napredovanja. Optimalni broj množenja možemo objasniti slijedećom rekurzivnom logikom. Kao početni uvjet razmotrimo slučaj kada je $i = j$. U tom slučaju niz se sastoji samo od jedne matrice, pa je broj množenja jednak 0. Slijedi da je $m[i,i] = 0$. Ako je $i < j$, tada tražimo umnožak $A_{i..j}$. Taj problem možemo riješiti promatrajući svaki k iz $i \le k < j$ (dakle promatrajući svaki umnožak $A_{i..k}$ s $A_{k+1..j}$.

Optimalno vrijeme za izračunati $A_{i..k}$ je $m[i,k]$, a optimalno vrijeme za izračunati $A_{k+1..j}$ je $m[k+1, j]$. Pretpostavljamo da smo te vrijednosti već izračunali i da smo iz pohranili u neko polje. Budući da je $A_{i..k}$ matrica dimenzija $p_{i-1} \times p_k$, a $A_{k+1..j}$ matrica dimenzija $p_k \times p_j$, vrijeme potrebno za njihovo množenje je $p_{i-1} \cdot p_k \cdot p_j$. Iz toga proizlazi slijedeće rekurzivno pravilo za računanje $m[i, j]$.

$$m[i,i] = 0$$
$$m[i,j] = \min_{i \le k < j} \left(m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \right) \qquad \text{za } i < j$$

Ovo rekurzivno pravilo sada treba pretvoriti u algoritam. Kod računanja $m[i, j]$ trebaju nam vrijednosti $m[i,k]$ i $m[k+1, j]$ za k koji je između i i j. To znači da računanje moramo organizirati prema broju matrica u podnizu. Neka je $L = j - i + 1$ dužina podniza (broj matrica) koji množimo. Podniz dužine 1 ($m[i,i]$) je trivijalan. Nakon toga gradimo rješenje računajući podnizove dužine $2,3,4,...,n$. Konačni odgovor koji nas zanima je $m[1,n]$. Kod petlji treba biti pažljiv. S obzirom da podniz duljine L počinje iz pozicije i, slijedi da je $j = i + L - 1$. Kako je $j \le n$, to znači da je $i + L - 1 \le n$ ili $i \le n - L + 1$. Stoga se petlja po i, kako bi j zadržala u granicama, vrti od 1 do $n - L + 1$.

```
NizMatrica(array p[1..n], int n)
    array s[1..n-1, 2..n]
    for i = 1 to n do m[i,i] = 0                //inicijalizacija tj. dužina podniza = 1
    for L = 2 to n do                           //dužina podniza
        for i = 1 to n-L+1 do                   //početak podniza
            j = i + L –1                        //kraj podniza
            m[i,j] = INFINITY
            for k = i to j-1 do
                q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
                if (q < m[i,j] )
```

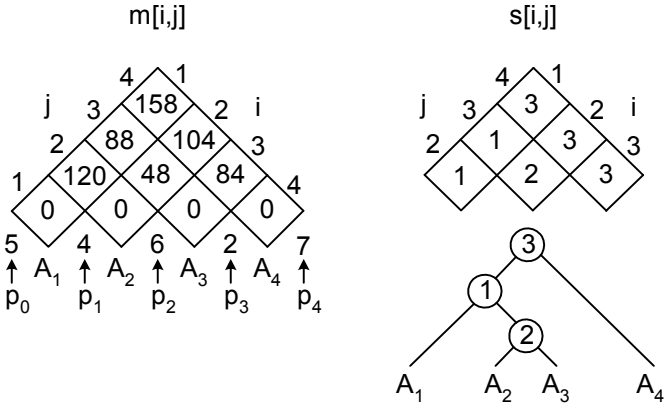
```
                    m[i,j] = q
                    s[i,j] = k
    return m[1,n]
    return s
```

Polje s[i,j] služi za izvlačenje stvarne sekvence (rasporeda množenja). Vrijeme izvršavanja ove procedure jednako je $O(n^3)$. (Izračunajte to!)

Izvlačenje konačne sekvence

Osnovna ideja je da pratimo najbolje podjele. Drugim riječima, kada pronađemo koja je to vrijednost k koja vodi minimalnom $m[i,j]$, onda to zapišimo u posebno polje (s[i,j]). Primjerice, pretpostavimo da je $s[i,j] = k$. To znači da je najbolje podijeliti podniz $A_{i..j}$ tako da najprije pomnožimo podniz $A_{i..k}$, potom podniz $A_{k+1..j}$, te na kraju pomnožimo dvije tako dobivene matrice. Jasno je da nam $s[i,j]$ kaže koje množenje radimo zadnje.
Pretpostavimo da su matrice pohranjene u polju matrica $A[1..n]$ te da je $s[i,j]$ globalna za proceduru Mnozenje.

```
Mnozenje(i, j)
    if (i < j)
        k = s[i, j]
        X = Mnozenje(i, k)
        Y = Mnozenje(k+1, j)
        return X*Y                          //množenje matrica X i Y
    else
        retururn A[i]
```



Slika: Primjer množenja niza matrica

U primjeru je dano množenje niza matrica $A_1\ (5\times 4)$ s $A_2\ (4\times 6)$ s $A_3\ (6\times 2)$ s $A_4\ (2\times 7)$. Optimalni raspored množenja je $((A_1(A_2A_3))A_4)$.

11. NP-POTPUNOST

Kada pronađemo rješenje nekog problema (algoritam) onda često postavimo sebi pitanje da li je to naše rješenje najbolje moguće. Posebno ako je rješenje sporo, odnosno ako je vrijeme izvršavanja eksponencijalno, npr. 3^n, $n^{\sqrt{n}}$, $n!$. Drugim riječima, da li postoje takvi problemi koje, ma kako ih dobro riješili, ne možemo riješiti brže od eksponencijalnog vremena.

Kako su se pronalazila efikasna rješenja mnogih problema, tako je rasla i lista problema koji se ne mogu efikasno riješiti. To su bili problemi za koje je najbolje poznato rješenje imalo eksponencijalno vrijeme izvršavanja. Ljudi su se počeli pitati da li postoji neka metoda koja još nije poznata, ili su to jednostavno teški problemi za koje ne postoji brže rješenje od eksponencijalnog.

Krajem 60-tih i početkom 70-tih znanstvenici su došli do jednog važnog otkrića. Pokazali su da su mnogi od tih teških problema povezani u smislu da, ako možeš riješiti jedan od njih u polinomskom vremenu, onda možeš i sve druge riješiti u polinomskom vremenu.

Polinomsko vrijeme

Moramo pronaći neki način da razlikujemo efikasno rješive probleme od neefikasno rješivih.

Algoritam s polinomskim vremenom (polynomial time algorithm) je bilo koji algoritam sa svojstvom da je vrijeme izvršavanja $O(n^k)$, gdje je k konstanta neovisna o n. Ako za neki problem postoji algoritam koji ga rješava u polinomskom vremenu, tada za taj problem kažemo da je rješiv u polinomskom vremenu (solvable in polynomial time).

Uočite da neke funkcije koje na prvi pogled nisu polinomske zapravo jesu. Primjerice, vrijeme izvršavanja $O(n \log n)$ ne izgleda kao polinomsko, ali jeste jer je omeđeno od gore s $O(n^2)$. Neke funkcije koje izgledaju kao polinomske to zapravo nisu. Primjerice, vrijeme izvršavanja $O(n^k)$ nije polinomsko vrijeme ako je parametar k funkcija od n.

Problemi odlučivanja

Mnogi problemi uključuju neki oblik optimizacije: pronađi najkraći put, minimalnu cijenu i sl. Većina problema koje ćemo diskutirati biti će, iz praktičnih razloga, problemi odlučivanja. Problem odlučivanja (decision problem), je takav problem kod kojeg je izlaz "da" ili "ne" (odnosno "0" ili "1", "istina" ili "laž" i sl.).

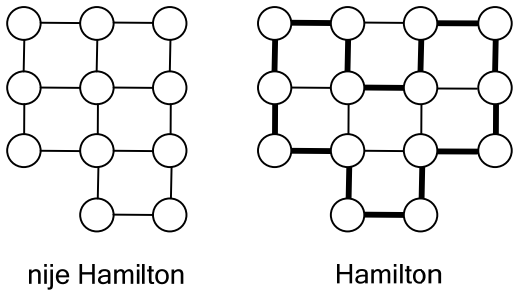
Uočite da gornje ograničenje (promatranje binarnih probleme) nije bitno ograničenje. Većina problema može se svesti na probleme s binarnim odlučivanjem. Primjerice, umjesto da postavimo pitanje: koliki je, za dani graf, najkraći put između čvorova, možemo postaviti pitanje: da li za dani graf postoji najkraći put između čvorova duljine k. Ako znamo odgovor na ovaj problem s binarnim odlučivanjem, tada možemo naći i najkraći put između čvorova. To možemo napraviti pokušavajući sve moguće vrijednosti k.

Definirat ćemo P kao skup svih problema odlučivanja koji se mogu riješiti u polinomskom vremenu.

NP i verifikacija polinomskim vremenom (NP and Polynomial Time Verification)

Prije govora o klasi NP-problema, važno je uvesti pojam verifikacije. Primjerice, mnogi problemi mogu biti vrlo teški za riješiti, ali imaju svojstva da je lako provjeriti (verificirati) da li je rješenje točno.

Razmotrimo slijedeći problem koji se zove problem neusmjerenog Hamilton-ovog ciklusa (Undirected Hamiltonian Cycle Problem (UHC)). Za dani neusmjereni graf G, da li G ima ciklus koji posjeti svaki čvor točno jednom ?



Slika: Neusmjereni Hamilton ciklus

Ako graf uistinu ima Hamilton-ov ciklus, i ako ste ga pronašli, tada je vrlo lako uvjeriti nekoga da je to točno. Dovoljno je kazati čemu je Hamilton-ov ciklus jednak (primjerice $(v_5, v_6, v_2, ..., v_9)$). Provjera je vrlo jednostavna jer treba vidjeti samo da li je to ciklus i da li posjeti svaki čvor točno jednom. Drugim riječima, iako ne znamo efikasan način za riješiti problem Hamilton-ovog ciklusa, znamo efikasan način da verificiramo da je dani graf Hamilton. Dani ciklus naziva se certifikat (certificate). Općenito certifikat je ona informacija koja nam omogućava da verificiramo (provjerimo) rješenje problema. Ako je moguće verificirati točnost certifikata u polinomskom vremenu, tada kažemo da je problem verificabilan (provjerljiv) u polinomskom vremenu (polynomial time verifiable).

Nisu svi problemi takvi da je rješenje lako verificirati. Razmotrimo problem određivanja da li graf ima točno jedan Hamilton-ov ciklus. Lako je nekoga uvjeriti da postoji barem jedan ciklus, ali vrlo teško da je taj jedan ujedno i jedini.

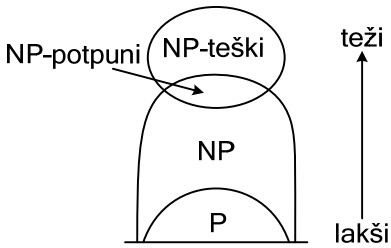
Definirat ćemo NP kao skup svih problema odlučivanja koji se mogu verificirati u polinomskom vremenu.

Polinomsko vrijeme verificiranja i polinomsko vrijeme rješavanja dva su bitno različita pojma. Uočite da vrijedi $P \subseteq NP$. Drugim riječima, ako možemo riješiti problem u polinomskom vremenu onda sasvim sigurno možemo i provjeriti da li je rješenje točno u polinomskom vremenu.

Međutim, nije poznato da li vrijedi $P = NP$ i prilično se razumnim čini da to nije tako. Drugim riječima, znati verificirati točnost odgovora u polinomskom vremenu, neće puno pomoći kod traženja rješenje. Većina stručnjaka misli da vrijedi $P \neq NP$, ali nitko nije dao dokaz da je to tako. (NP potječe od izraza nondeterministic polynomial time.)

NP-potpunost (NP-completeness)

Nećemo dati formalnu definiciju NP-potpunosti jer bi to izašlo iz okvira ovog predmeta. Jednostavno ćemo za NP-potpune probleme reći da su to problemi, iz skupa svih NP problema, koje je najteže riješiti. Postoje i problemi koje je još teže riješiti, a koji nisu u klasi NP problema. Takvi problemi zovu se NP-teški (NP-hard) problemi.



Slika: Odnos između P, NP, NP-potpunih i NP-teških problema

Pitanje je kako možemo pitanje težine problema matematički formulirati. Za to nam je potreban koncept redukcije (reduction).

Redukcija

Pretpostavimo da imamo dva problema, A i B. Znamo (ili barem jako vjerujemo) da je nemoguće riješiti problem A u polinomskom vremenu. Želimo dokazati da B ne možemo riješiti u polinomskom vremenu. Kako to možemo napraviti?
Želimo pokazati da

$$A \notin P \quad \Rightarrow \quad B \notin P$$

Da bi to dokazali, dokažimo inverznu tvrdnju

$$B \in P \quad \Rightarrow \quad A \in P$$

Drugim riječima, da bi pokazali da B nije rješiv u polinomskom vremenu, mi ćemo pretpostaviti da postoji takav algoritam koji rješava B u polinomskom vremenu. Ako, polazeći od te pretpostavke, može pokazati da A može biti riješen u polinomskom vremenu, dobili smo kontradikciju. Naime, za A znamo (ili sumnjamo) da se ne može riješiti u polinomskom vremenu.

Potom treba izvesti kontradikciju pokazujući da A može biti riješen u polinomskom vremenu.

Kako to napraviti? Pretpostavimo da imamo potprogram kojim možemo riješiti problem B u polinomskom vremenu. Tada je sve što trebamo napraviti pokazati da možemo upotrijebiti taj potprogram da riješimo problem A (u polinomskom vremenu). Time smo reducirali problem A na problem B. (Uočite da pretpostavljeni potprogram zapravo ne postoji.) Mi znamo (ili jako vjerujemo) da A ne može biti riješen u polinomskom vremenu. To znači da mi u načelu

pokazujemo da takav potprogram ne postoji, što implicira da B ne može biti riješen u polinomskom vremenu.

Promotrimo jedan primjer, kako bi ovo razmatranje bilo jasnije. Vrijedi (ili barem mislimo da vrijed) da je pitanje da li neusmjereni graf ima Hamilton-ov ciklus, NP-potpuni problem. To znači da (za sada) ne postoji algoritam koji bi problem rješavao u polinomskom vremenu, te osim toga većina stručnjaka vjeruje da takav algoritam niti ne postoji.

Pretpostavimo da vam šef kaže da želi polinomsko rješenje problema Hamilton-ovog ciklusa za usmjereni graf (DHC). Nakon kratkog razmišljanja uvjerite se da taj zahtjev nije razuman. Naime, dodavanje smjerova u veze neće napraviti problem ništa lakšim od varijante s neusmjerenim grafovima (UHC). Međutim vaš šef smatra da je varijanta sa smjerovima lakša. Kako ga uvjeriti da nije u pravu?
Pretpostavimo da postoji efikasno (dakle u polinomskom vremenu) rješenje DHC problema, te pokažimo da iz toga slijedi da postoji i rješenje UHC problema u polinomskom vremenu. Dakle, upotrijebit ćemo algoritam za DHC (koji zapravo još nismo niti napisali) kao potprogram za rješenje UHC. Kako se i šef slaže da UHC ne može biti riješen u polinomskom vremenu, slijedilo bi da DHC nikako ne može biti u polinomskom vremenu. Slijedi da je postavljeni zadatak nerazumno težak, te da do sada za takav problem nije nađeno efikasno rješenje.

Evo kako to možemo pokazati. Za dani neusmjereni graf G, kreiraj usmjereni graf G' tako da svaku neusmjerenu vezu $\{u, v\}$, zamijenimo s dvije usmjerene veze $\{u, v\}$ i $\{v, u\}$. Sada je svaki jednostavni put u G ujedno i jednostavni put u G', i obrnuto. Stoga G ima Hamilton-ov ciklus ako i samo ako ga ima i G'. Dakle, algoritam za rješavanje problema neusmjerenog Hamilton-ovog ciklusa je slijedeći. Uzmi neusmjereni graf G, pretvori ga u ekvivalentni usmjereni graf G' (dupliciranjem veza), te pozovi (pretpostavljeni) algoritam za usmjereni Hamilton-ov ciklus. Koji god odgovor taj algoritam dao, to je ujedno odgovor i na početno pitanje. Kako nemamo polinomski algoritam za DHC, slijedi da ga neće biti niti za UHC. Uočite da nismo riješili niti UHC niti DHC problem. Samo smo pokazali kako pretvoriti rješenje DHC u rješenje UHC. To se zove redukcija.

Za dana dva problema A i B, kažemo da se A polinomski reducira (sažima) (polynomially reducible) u B, ako, dani polinomski potprogram za B, možemo iskoristiti da riješimo A u polinomskom vremenu. To pišemo kao

$$A \leq_P B$$

Koncept redukcije primijenit ćemo na probleme za koje jako vjerujemo da ne postoji rješenje u polinomskom vremenu.

Neki važne činjenice o redukciji:

1. Ako je $A \leq_P B$ i $B \in P$ tada $A \in P$.
2. Ako je $A \leq_P B$ i $A \notin P$ tada $B \notin P$.
3. Ako je $A \leq_P B$ i $B \leq_P C$ tada $A \leq_P C$. (tranzitivnost)

Prije smo definirali NP-potpune probleme kao najteže NP probleme. Dajmo sada nešto formalniju definiciju NP-potpunosti (u odnosu na reducibilnost).

B je NP-potpun problem ako

1. $B \in NP$ i
2. Svaki NP problem A možemo reducirati u B ($A \leq_P B$) u polinomskom vremenu.

Stoga, ako možeš riješiti B u polinomskom vremenu, tada možeš riješiti i A u polinomskom vremenu. Kako je A NP, možeš riješiti svaki problem iz NP u polinomskom vremenu. Drugim riječima, ako bilo koji problem koji je NP potpuno možeš riješiti u polinomskom vremenu, tada i svaki problem iz NP možeš riješiti u polinomskom vremenu.

Uočite da je smjer u kojem se radi redukcija nije uobičajen. U normalnoj redukciji, želimo reducirati problem koji ne znamo kako riješiti u neki koji znamo kako riješiti. Ali kod NP-potpunih problema, mi ne znamo kako riješiti nijedan problem (zapravo želimo pokazati da efikasno rješenje uopće ne postoji). Stoga je smjer redukcije zapravo suprotan. Mi reduciramo poznati problem u problem za koji želimo pokazati da je NP-potpun.

Dakle, uvijek reduciramo poznati NP-potpuni problem u problem za koji želimo pokazati da je NP-potpun. Uočite još i da redukcijom uopće ne želimo riješiti problem. Jednostavno želimo napraviti jedan problem sličan nekom drugom. Reduciranjem možemo ići toliko daleko da kažemo da zapravo postoji samo jedan NP-potpun problem.

Primjer: Hamilton-ov put i Hamilton-ov ciklus

Već smo vidjeli problem Hamilton-ovog ciklusa (HC): da li, za dani graf, postoji ciklus koji posjeti svaki čvor točno jednom? Promotrimo sada problem Hamiltonovog puta (HP): da li, za dani graf, postoji jednostavni put koji posjeti svaki čvor točno jednom.

Pretpostavimo da znamo da je HC problem NP-potpun, te da želimo pokazati da je i HP problem NP-potpun. Kako to napraviti? Podsjetimo se, želimo pokazati da se poznati NP-potpuni problem može reducirati na naš problem, dakle da je $HC \leq_P HP$. Drugim riječima, pretpostavimo da imamo potprogram koji rješava HP u polinomskom vremenu. Kako možemo to iskoristiti da riješimo HC u polinomskom vremenu.

Prvi pokušaj (koji ne radi) bio bi slijedeći. Pretpostavimo da smo primijenili potprogram HamPath (koji rješava HP problem u polinomskom vremenu) i da je odgovor ne tj. da nema rješenja. U tom slučaju je i rješenje početnog pitanja (tj. postojanje ciklusa) također ne. (Sasvim je jasno da, ako u nekom grafu nema puta, onda nema niti ciklusa. Nasuprot tomu, postoje grafovi koji imaju Hamilton-ov put ali nemaju Hamilton-ov ciklus.) Međutim što ako je rješenje potprograma HamPath da, dakle ako put postoji. Tada ciklus može i ne mora postojati, pa jednoznačno rješenje našeg problema ne postoji.

Drugi pristup (isto bez točnog rješenja) bio bi slijedeći. Možemo konvertirati ciklus u put brisanjem bilo koje veze u ciklusu. Pretpostavimo da graf G ima Hamilton-ov ciklus. Tada taj ciklus počinje u nekom čvoru u, posjeti sve čvorove u grafu, dođe do zadnjeg čvora v i ponovno se vrati u čvor u. Dakle, sigurno postoji veza {u,v} u grafu. Izbrišimo sada tu vezu čime Hamilton-ov ciklus postaje Hamilton-ov put. Primijenimo, na tako dobiveni graf, potprogram HamPath. Kako znati koju vezu izbrisati?. Ne znamo, pa ćemo pokušati sa svim

vezama. Ako HamPath potprogram kaže da za svaku izbrisanu vezu, tada je i odgovor na naše početno pitanje da.

Međutim i ovdje postoji problem. Naša namjera je da Hamilton-ov put počne u u i završi u v. Ali kada pozivamo potprogram HamPath, nemamo načina da natjeramo potprogram da ispuni taj uvjet. Mi ne možemo modificirati potprogram (Podsjetimo se, potprogram zapravo niti ne postoji). Možemo ga samo pozvati i provjeriti odgovor.

Kako dakle natjerati HamPath potprogram da krene iz u završi u v. Za to trebamo malo modificirati algoritam. Osim što ćemo izbrisati vezu od u do v, dodat ćemo dodatni čvor x povezan samo na u i dodatni čvor y povezan samo na v. Kako ti čvorovi imaju stupanj jedan, ako Hamilton-ov put postoji, on mora početi u x i završiti u y. Sada možemo dati algoritam koji će raditi.

Za dani graf G za koji želimo odrediti da li ima Hamilton-ov ciklus, idemo od veze do veze i to jednu po jednu. Za svaku vezu {u,v} (nadajući se da će to biti zadnja veza u Hamilton-ovom ciklusu) napravimo novi graf brišući tu vezu i dodajući čvor x povezan samo na u i čvor y povezan samo na y. Neka se rezultirajući čvor zove G'. Sada primijenimo potprogram HamPath da vidimo da li G' ima Hamilton-ov put. Ako ima, tada mora početi iz x u y i završiti iz v u y (ili obrnuto). Tada znamo da originalni graf ima Hamilton-ov ciklus. Ako za sve rubove, ne postoji Hamilton-ov put, tada originalni graf nema niti Hamilton-ov ciklus.

```
bool HamCycle (graph G)
    za svaki rub {u,v} iz G
        kopiraj G u novi graf G'
        izbriši rub {u,v} iz G'
        dodaj novi čvor x i y u G'
        ako (Hampath(G') vrati true
    return false
```

Slijedi da, kako je HC NP-potpun problem, to znači da neće biti moguće naći niti efikasno rješenje HP problema.