Turinys

1.1 Funkcijų augimo asimptotiniai žymėjimai ir jų apibrėžimai

Funkcijų augimo asimptotiniai žymėjimai ir jų apibrėžimai.

Apibrėžimas 1. Sakysime, kad $f(x) = o(g(x))$ ($x \to \infty$)$_5$ jei $\lim_{x \to \infty} f/(x)/g(x)$ egzistuoja ir lygi 0. Funkcija *f* auga daug lėčiau nei *g*. Pavyzdžiui, $x^2=o(x^5)$, $\sin x = o(x)$, $14.709\sqrt{x}=o(x/2 + 7\cos x)$.

Apibrėžimas 2. Sakysime, kad $f(x) = O(g(x))$ ($x \to \infty$), jei C, x_0: $|f(x)| < Cg(x)$ ($x > x_0$). Funkcija f nebedidėja greičiau nei *g*. Rodiklis ($x \to \infty$) paprastai nerašomas. Simbolis *o* griežčiau apibrėžia funkcija, pvz., $1/(1+x^2) = o(1)$ nurodo, jog funkcija ne tik nebus didesnė už 1, bet taip pat artėja į 0, kai $x \to \infty$.

Apibrėžimas 3. Sakysime, kad $f(x) = \Theta(g(x))$ ($x \to \infty$), jei $c_1 \neq 0$, $c_2 \neq 0$, x_0: $c_2 g(x) < f(x)) < c_2 g(x)$ ($x > x_0$) • Funkcija f yra panašaus *g* augimo greičio (įvertinant keletą nežinomų dauginamųjų). Tai griežtesnis nei o ir O apibrėžimas funkcijai f;

Apibrėžimas 4. Sakysime, kad $f(x) \sim g(x)$ ($x \to \infty$), jei $\lim_{x \to \infty} f(x)/g(x) = l$. Tai griežčiausias iš visų minėtų apibrėžimas; pvz., $(3x+1)^4 \sim 81x^4$, $\sin(1/x) \sim 1/x$, $2^x+7 \log x + \cos x \sim 2^x$.

Apibrėžimas 5. Sakysime, kad $f(x) = \Omega(g(x))$ ($X \to \infty$), jei $\varepsilon > 0$: $|f(x)| > \varepsilon g(x)$. Tai atvirkščią reikšmę nei *o* turintis dydis, ir naudojantis funkcija *g(x)* apibrėžiamas apatinis rėžis funkcijai *f(x)*.

Apibrėžimas 6. Funkcija, kuri auga greičiau nei x^a, bet lėčiau nei c^x, $c>1$, vadinama nežymiai eksponentiškai didėjančia. Kitaip, $f(x) = \Omega(x^{a)}$, $a > 0$ ir $f(x) = o((1+\varepsilon)^x)$, $\varepsilon>0$.

Apibrėžimas 7. Funkcija vadinama eksponentiškai didėjančia, jei $C>1: F(X) = \Omega(C^X)$ ir $d: f(x) = o(d^x)$.

1.2 Rekurentinių sąryšių sprendimo būdai. Suformuoluoti Pagrindinę teoremą

Rekurentinių sąryšių sprendimo būdai (aprašyti idėją). Suformuluoti Pagrindinę teoremą.

Trys sprendimo būdai:
1. Pakeitimo
2. Rekursijos medžio
3. Pagrindinis

1)Sprendimas pakeitimo būdų susideda iš dviejų etapų:
1. Daroma prielaida apie sprendinio formą.
2. Matematinės indukcijos būdu nustatomos konstantos ir įrodoma, kad sprendinys teisingas.

2) Rekursijos medžio metodas.

Dažnai nuspėti rekursijos sprendinio pavidalą sunku. Padėti gali rekursijos medis, kurio viršūnėje įrašytas laikas, reikalingas išspręsti atskiram subuždaviniui. Vėliau šie laikai sumuojami ir randamos pilnas sprendimo laikas. Tai dažniausiai taikoma dekompoziciniams algoritmams, kurie sudaromi principu „skladyk ir valdyk“. Jei pavyksta sudaryti rekursijos medį pakankamai tiksliai ir kruopščiai sudėti gautus narius, šis sprendinys gali tapti sprendinio įrodymu.

3) Teorema:

Tegul $a \geq 1$ ir $b \geq 1$ yra konstantos, $f(n)$ – bet kokia funkcija, o $T(n)$ apibrėžiama natūrinių skaičių aibėje rekurentinio sąryšio pagalba

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

Čia n/b suprantam kaip $\lfloor n/b \rfloor$ arba $\lceil n/b \rceil$. Tada asimptotinis elgesys T(n) išreikškiamas taip:

1. Jei $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0$, tada $T(n) = \Theta(n^{\log_b a})$
2. Jei $f(n) = \Theta(n^{\log_b a})$, tai $T(n) = \Theta(n^{\log_b a} \lg n)$
3. Jei $f(n) = \Omega(n^{\log_b a + \varepsilon})$, $\varepsilon > 0$, ir jei $af(n/b) < cf(n)$, kai $c < 1$ ir pakankamai dideliems n, tada $T(n) = \Theta(f(n))$

1.3 Dekompozicinių algoritmų sudėtingumo T(n) skaičiavimo formulės struktūra ir sprendimo būdai

Dekompozicinių algoritmų sudėtingumo T(n) skaičiavimo formulės struktūra ir sprendimo būdai.

Dekompoziciniai algoritmai dažniausiai įgyvendinami rekursyviai, tokio algoritmo darbo laikas:

$$T(n) = \begin{cases} \Theta(1), & n \leq c \\ aT(n/b) + D(n) + C(n) \end{cases} \quad (1)$$

Jei rekursyvios funkcijos kitas narys gali būti gaunamas $x_{n+1} = b_n x_n + c_n$, tuomet sudėtingumui skaičiuoti naudojame formulę:

$$x_n = (\prod_{j=1}^{n-1} b_j)(x_1 + \sum_{i=1}^{n-1} d_i), \text{ čia } d_i = \frac{c_i}{\prod_{j=1}^{i} b_j}.$$

Kadangi dekompozicinių algoritmų rekursyvi funkcija neatitinka šios formos, norėdami naudoti duotąją formulę turime taikyti keitinį $k = \log_b n$, $b^k = n$, ir skaičiuoti ne $T(n)$, o $L(k) = T(n) = T(b^k)$. b – konstanta iš (1) formulės.

1.4 Rūšiavimas su įterpimu (duomenys įvedami palankiausiu atveju)

. Rūšiavimas su įterpimu. Įrodyti algoritmo korektiškumą ir jo sudėtingumą kai duomenys įvedami palankiausiu atveju.

Algoritmas skirtas rūšiavimo problemai spręsti.

Įėjimas: skaičių seka $< a_1, a_2, \dots, a_n >$

Išėjimas: pertvarkymas į $< a`_1, a`_2, \dots, a`_n >$, kad naujos sekos nariai tenkintų sąlygą $a`_1 \le a`_2 \le \dots \le a`_n$. Algoritmo esmė įterpti naują elementą į jau surūšiuotą seką (masyvą). Pvz jei reikia rūšiuoti kortas, tai paėmę pirmąją jau turėsime surūšiuotą aibę. Antrąją kortą talpinsime arba prieš ankstesniąją arba po jos. Kitoms kortoms ieškosime vietos, o radę vietą, kur ji tinka – įterpsime. Tarkime reikia surūšiuoti masyvą A[1..n], turintį n rūšiuojamų narių. lenght[A] – elem. skaičius masyve. Algoritmo pseudo kodas:

```
Insertion_Sort(A)
1. for <- 2 to length[A]
2.    do key <- A[j]
3.     įterpiamas elementas į surūšiuotą seką A[1..j-1]
4.     i <- j-1
5.     while i > 0 ir A[i] > key
6.     do A[i+1] <- A[i]
7.         i <- i-1
8.     A[i+1] <- key
```

Algoritmui ciklo invariantas yra faktas, kad masyvas A[1..n] pozicijose nuo 1 iki j iš pradžių yra nesurūšiuotas, o ciklui pasibaigus jo nariai išdėlioti didėjimo tvarka. Ciklo invariantas turi tenkinti tris savybes:

1. Inicializacijos – t.y ar teisingas prie prieš pirmą ciklo iteraciją.
2. Išsaugojamumą – t.y jei teisinga prieš eilinę ciklo iteraciją tai teisinga ir po jos.
3. Pabaigiamumas – po ciklo pabaigos invariantas leidžia įsitikinti algoritmo teisingumu.

Pirmosios dvi savybės siejasi su matematinės indukcijos metodu. 3 Savybė svarbiausia norint parodyti algoritmo korektiškumą.

Nagrinėsime ar galioja šios 3 sąvybės rūšiavimo su įterpimu atveju:

Inicializacija. Tarkime j = 2, tokiu atveju elementų poaibį A[1..j-1] sudaro tik vienas elementas A[1], saugantis pradinę reikšmę ir be to surūšiuotas (trivialus teiginys). Vadinasi teiginys apie ciklo invarintą prieš pradinę ciklo iteraciją yra teisingas. Išsaugojamumas. Parodysime, kad ciklo invariantas išsaugomas po kiekvienos iteracijos. Neformaliai kalbant išoriniame for cikle įvyksta postūmis A[j-1], A[j-2], A[j-3] ... per vieną poziciją, kol neatsilaisvins vieta elementui A[j]. Tačiau formalus įrodymas bus tik tada gei bus suformuluotas invariantas ciklui while ir įrodytas. Pabaigiamumas. Ciklas for baigiasi, kai j viršija n, t.y j = n+1. Įstatę tai į invarianto formulavimą gausime, kad poaibyje A[1..n] visi elementai yra sutvarkyti, o tai sutampa su masyvu A. Iš čia seka algoritmo korektiškumas.

Pats palankiausias atvejis, kai įvedama jau surūšiuota seka, tuo atveju visuomet bus tenkinama sąlyga $A[i] \le key$. Todėl algoritmas bus sukamas j = 2, … n kartų, o $t_j = 1$ tad darbo laikas pačiu palankiausiu atveju:

1.5 Rūšiavimas su įterpimu (duomenys įvedami nepalankiausiu atveju)

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = an + b$, čia a ir b – konstantos, priklausančios nuo c_i. Šiuo atveju rūšiavimo laikas $T(n)$ yra tiesinė funkcija, priklausanti nuo n (duomenų kiekio).

. Įrodyti algoritmo korektiškumą ir jo sudėtingumą kai duomenys įvedami nepalankiausiu atveju.

Pats blogiausias atvejis, kai seka surūšiuota mažėjimo tvarka. Šiuo atveju $t_j = j$, $\sum_{j=1}^{n} j = \frac{n(n+1)}{2} \Rightarrow \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_7(\frac{n(n-1)}{2}) + c_8(n-1) = an^2 + bn + c$, čia a, b, c - konstantos, priklausančios nuo c_i. Šiuo tai kvadratinė priklausomybė nuo n $T(n^2)$

Rūšiavimo uždavinys suliejimo būdu (RUS):

1. Dalinimas – rūšiuojama seka iš n narių dalinama į du uždavinius po n/2 elementų.
2. Pavergimas – abu gauti uždaviniai sprendžiami RUS būdu.
3. Kombinavimas – Abiejų surūšiuotų sekų sujungimas galutinio sprendinio gavimui.

Procedūra Merge(A, p , q, r), čia A- masyvas, p, q ir r – indeksai ($p \leq q < r$). Šioje procedūroje suprantama, kad A[p..q] ir A[q+1..r] sutvarkyti ir ji atlieka šių sekų suliejimą per $\Theta(n)$ žingsnių.

Merge(A, p , q, r)
1. n_1 <- q-p+1
2. n_2 <- r-q
3. sudaromi masyvai L[1.. n_1+1] ir R[1.. n_2+1]
4. for i <- 1 to n_1
5.   do L[i] <- A[p+i-1]
6. for j <-1 to n_2
7.   do R[j] <- A[q+j]
8. L[n_1+1] <- ∞
9. R[n_2+1] <- ∞
10. i <-1
11. j <-1
12. for k <-p to r
13.   do if L[i] $\leq$ R[j]
14.   then A[k] <- L[i]
15.   i <- i+1
16.   else A[k] <- R[j]
17.   j <- j+1

Bendras RUS algoritmas:

Merge_Sort[A,p,r]
1. if p<r
2.   then q <- L[(p+r)/2]
3.   Merge_Sort(A,p,q)
4.   Merge_Sort(A,q+1,r)
5.   Merge(A,p,q,r)

1.6 Rikiavimo algoritmo suliejimo būdu korektiškumo įrodymas

4. Irodykite rūšiavimo algoritmo suliejimo būdu korektiškuma.

Ciklo for (12-17) invariantas aprašomas taip:

Masyvo dalis A[p..k-1] turi k-p mažiausių surūšiuotų elementų iš masyvų L[1.. n_1+1] ir R[1.. n_2+1]. Be to cikle L[i] ir R[i] yra mažiausi tų masyvų elementai, kurie dar nenukopijuoti į masyvą A.

Inicializacija. Prieš pirmą ciklą k=p, todėl A[p..k-1] – tuščias. Todėl turi p-k = 0 elementų iš L ir R masyvų. Kadangi i = j = 1, tai L[i] ir R[j] yra mažiausi masyvų L ir R elementai, be to nenukopijuoti į masyvą A.

Išsaugojimas. Tarkime, kad L[i] $\leq$ R[j], tada L[i] kopijuojamas i masyva A. Kadangi masyvo dalyje A[p..k-1] yra k-p mažiausių elementų. Po 14 elutes įvykdymo jų bus k-p-1. Padidės ciklo kintamojo k ir i reikšmė, ir ciklo invariantas atsistato. Jei L[i]>R[j] pasikartoja panašūs veiksmai.

Pabaigiamumas. Algoritmas užsibaigia, kai k=r+1, taigi masyvas A[p..k-1] savyje talpina k-p =r-p+1 mažiausių surūšiuotų masyvų L[1.. n_1+1] ir R[1.. n_2+1] elmentų. Suminis masyvų L ir R elementų skaičius yra n_1+ n_2+2 = r-p +3. Visi jie yra nukopijuoti išskyrus du pačius didžiausius.

1.7 Rikiavimo algoritmo suliejimo būdų sudėtingumo įrodymas

. Rūšiavimo algoritmo suliejimo būdu sudėtingumo irodymas.

Tarkime *T(n)* yra algoritmo darbo laikas. Musų uždavinys skaldomas į a uždavinių, kurių dydis yra 1/b pradinio uždavinio dydžio. Jei padalinimas į subuždavinius įvyksta per D(n) laiko, o surinkimas per C(n), tada galime parašyti rekurentinę formulę:

$$T(n) = \begin{cases} \Theta(1), & n \le c \\ aT(n/b) + D(n) + C(n) & \end{cases}$$

RUS atveju:

Dalinimas. Masyvu vidurio radimas atliekamas per fikstuotą laiką $\Theta(1)$, t.y. D(n) = $\Theta(1)$

Pavergimas. Rekursyviai sprendžiami du uždaviniai, kurio kievienas n/2 dydžio. Sprendimo laikas lygus 2T(n/2) (a=2, b=2).

Kombinavimas. Parodėme, kad n elementų sujungimas įvykstą per $\Theta(n)$ laiko, t.y c(n) = $\Theta(n)$

Kadangi $\Theta(n)+\Theta(1)$ yra taip pat tiesinė funkcija pagal n, todėl suma lygi $\Theta(n)$. RUS atveju

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T(n/2) + \Theta(n), & n > 1 \end{cases}$$

Išsprendę šią rekurentinę formulę rekursinio medžio sudarymo būdu, gauname:

$cn(\log_2 n+1) = cn\log_2 n+cn$, nes medį sudaro $\log_2 n+1$ lygių, dalinimų skaičius $\log_2 n$. Iš čia:

T(n) = $\Theta(n\log_2 n)$, $\Theta(n)$ galime atmesti nes n didėja lėčiau nei $n\log_2 n$.

1.8 Rūšiavimo piramide algoritmo idėja. Kaip priklauso piramidės dydis ir aukštis nuo rūšiuojamų duomenų kiekio? (ir 1.9)

. Rūšiavimo piramide algoritmo idėja. Kokia duomenų struktūra – „piramidė“? Kaip priklauso piramidės dydis ir aukštis nuo rūšiuojamų duomenų kiekio?

Pirmaidė – tai duomenų struktūra, objektų masyvas, kuris yra beveik pilnas binarinis medis.

Binarinis medis – vaizduojamas masyvu susitariant, kad po pirmo elemento eina to elemento „vaikai“ , o po jų eina tų vaikų vaikai ir t.t.

Susitarkime, kad procedūra:

Parent(i); return [i/2];grąžins tėvinės piramidės viršūnės indeksą masyve. Left(i); return 2i; Right(i); return 2i+; fręžins vaikų (kairio ir dešinio) indeksus. Šios piramidės ypatumas, kad nereikia papildomos informacijos saugoti atminties. Pakanka atitinkamo masyvo ilgio. Tuo pasinaudojant ir gaunamas Pirmaidės rūšiavimo algoritmo privalumas prieš algoritmą su suliejimu. Laikykime , kad pirmaidė tenkina nedidėjančios savybę , jei A[Parent(i)] $\ge$ A[i], o nemažėjančios, jei A[Parent(i)] $\le$ A[i].

Piramidės aukštis jei joje yra ir n narių yra $\theta(log_2 n) = \theta(\lg\ n)$, nes pilname medyje su n viršūnių yra $2^R = n$, čia R – lygių skaičius -> R = $\lg\ n$, kadangi aukštis suprantamas, kaip lankų skaičius ilgiausiame kelyje (einančiu žemyn) -> kad aukštis yra lygus R-1 = lg(R-1).

1.9.Rūšiavimo piramide algoritmo idėja. Kaip atliekamas piramidės sutvarkymas (pateikite algoritmą ir sudėtingumą įrodykite).

Pirmaidė – tai duomenų struktūra, objektų masyvas, kuris yra beveik pilnas binarinis medis.

Binarinis medis – vaizduojamas masyvu susitariant, kad po pirmo elemento eina to elemento „vaikai“ , o po jų eina tų vaikų vaikai ir t.t.

Piramidės savybės palaikymui naudosime procedūrą MAX_Heapify(A,i):

Max_Heapify(A,i)
1. l <- left(i)
2. r <- Right(i)
3. if l $\le$ heap_size[A] ir A[l] > A[i]
4. then largest <- l
5. else largest <- i
6. if r $\le$ heap_size[A] ir A[r] > A[largest]
7. then largest <- r
8. if largest $\ne$ i
9. then A[i] <--> A[largest]
10. Max_Heapify(A,largest)

Imamas i-tasis viršūnės elementas, o po to tikrinama tvark su jo vaikais, jei su kuriuo nors jis netenkina piramidės (didėjimo ar mažėjimo) sąvybės sukeičiamos vietomis ir pereinama prie tvarkos tvarkymo kitoje piramidės vietoje (kaip tvarkymas vyksta pažiūrėkit scan0028 faile).

Procedūros darbo laikas yra T(n) $\le$ T(2n/3) + O(1), nes po kiekvienos iteracijos reikia nagrinėti nedidesnį nei 2n/3 dydžio piramidę (medį), o kiekvienas palyginus tarp A[i], A[Left(i)] ir A[Right(i)] trunka ne daugiau kaip fiksuotą laiko tarpą O(1).
Kodėl 2n/3 ? Labiausiai skirsis du uždaviniai , jei sudaryta piramidė turi pavidalą:
h + h/2 + 1 = n -> h < 2n/3 (kiek lygyje yra element7, tiek pat, kiek buvo visoje piramidėje iki to lygio piramidėje arba 2 kartus daugiau nei prieš einantį lygį). **Pagrindinė teorema,** duoda sudarytos

rekursinės funkcionalės sprendinį: T(n) = O(lg n)

a=1, b= 3/2 > 1, f(n) = O(1) = O($n^{\log_{\frac{3}{2}} 1}$)

antras atvejis -> T(n) = O($n^{\log_{\frac{3}{2}} 1} \lg n$) = O($\lg n$)

T(n) = T*(n) = O($\lg n$) => T(n) = O($\lg n$)

1.10 Greito rūšiavimo algoritmas (ir 1.11)

Greito rūšiavimo algoritmo idėja. Kaip atliekama atraminio elemento paieška?

Praktikoje dažniausiai naudojamas, nes nors blogiausiu atveju jis dirba O(n^2) n elementų masyvui. Tačiau vidutinis šio algoritmo skaičiavimo laikas O(n lgn). Šis algoritmas sudaromas principu „skaldyk ir valdyk“. Tarkime mums reikia sūrušiuoti masyvą A[p..r]. Tai atliekame, kaip visada trimis etapais:

1. Dalinimas: A[p..r] masyvas dalinamas į A[p..q-1] ir A[q+1..r], kiekvienas A[p..q-1] masyvo elementas neviršyja A[q] elementų, o $\forall$ A[q+1..r] elementas mažesnis už A[q].Indeksas q nustatomas algoritmo vykdymo metu.
2. Pavergimas: masyvai A[p..q-1] ir A[q+1..r] rūšiuojami rekursyviai iškviečiant greitojo rūšiavimo procedūrą.
3. Sujungimas: kadangi rūšiavimas atliekamas tame pačiame A masyve, todėl papildomų veiksmų nereikia.

Quicksort (A, p, q-1)

4. If p < r
5. Then q ← Partition (A,p,r)
6. QuickSort(A,p,q-1)
7. QuickSort(Q,q+1,r)

Viso masyvo rūšiavimo procedūros QuickSort(A,1, length[A]) iškvietimas.

Masyvo dalinimas

Partition (A,p,r)

1. X ← A[r]
2. I ← p-1
3. for j ← p to r-1
4. do if A[j] $\leq$ x
5. then i ← i+1
6. A[i] <--> A[j]
7. A[i+1] <-->A[r]
8. return i+1

Atraminis elementas – paskutinis masyvo elementas. Fiksuojama i-kintamuoju elementų mažiasnių už atraminį elementą pabaiga ir ciklai 3-6 bėgame per visus masyvo A[p..r-1] elementus. Jei kuris iš jų pasirodo mažesnis i – padidinam ir šio indekso elementas sukeičiamas su nagrinėjamu elementu. Pabaigus ciklą masyvo A[r] elementas sukeičiamas su i+1 elementu, t.y. nustatoma tiksli elemento A[r] vieta masyve.

Greito rūšiavimo algoritmo idėja. Įrodykite algoritmo sudėtingumą blogiausiu atveju.

Rūšiavimo greitis QuickSort algoritmo blogiausiu atveju:

T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n) 2ia buvo dalinama į du uždavinius n-1 dydžio ir kitą iš 0 elementų.

Artimetinė progresija O(n^2).

T(n) = T(n-1) + Cn

T(n-1) = T(n-2) + C(n-1) + Cn

T(n) = C $\sum_{i=0}^{n} n$ = (n + 0) $\frac{n}{2}$ = O(n^2)

Geriausias dalinimas, kai uždavinys (masyvas) dalinamas į du vienodus apimties uždavinius t.y $[\frac{n}{2}]$ ir $[\frac{n}{2}] - 1$

T(n) $\leq$ 2T($\frac{n}{2}$) + O(n), šiuo atveju T(n) = O(n lgn). Net asimetrinio padalinimo atveju darbo laikas labiau panašus į geriausią nei į blogiausią: T(n) $\leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + Cn$; T(n) = O(n lgn)

(nepilnas)

1.12 Kada greito rikiavimo algoritmo sudėtingumas ir rikiavimo piramide (Heap sort) sudėtingumo asimptotiniai įverčiai konstantos tikslumu sutampa? Įrodykite kokiam nors atvejui (rūšiavimo piramide sudėtingumo asimptotinio įverčio įrodyti nereikia)

(Iš knygos) (????)

1.13 Optimalūs rūšiavimo algoritmai

Optimalūs rūšiavimo algoritmai naudojantys palyginimus. Sių algoritmų įvertinimo blogiausiam atvejui įrodymas. Įrodyti, kad rūšiavimas piramide ir suliejimo būdu yra asimptotiškai optimalūs algoritmai.

Rūšiavimo naudojant palyginimus darbo laikas. a_i ir a_j galima palyginti šiais ženklais: $<, \leq, =, \geq, >$. Apie $<a_1, a_2, \ldots, a_n>$ elementus nėra žinoma.

Nemažindami bendrumo, tarkime kad visi elementai yra skirtingi $a_i \neq a_j$. Tokiu atveju pakanka vieno palyginimo tipo: $a_i \leq a_j$, norint nustatyti, kuris didesnis.

Teorema 8.1 Blogiausiu atveju bet kokio rūšiavimo algoritmo palyginimo būdu vykdomas atliekamas per $\Omega(n\lg n)$ palyginimų

Įrodymas seka iš to kad rūšiuojant n ilgio masyvą gali būti n! skirtingų perstatymų (keitinių).

Tarkime sprendimų medžio h su l pasiekimų lapų: $n! \leq l \leq 2^h$

$\lg(n!) \leq \lg l \leq 2^h$ => $h \geq \Omega(n\lg)$ įrodyti naudojant stirlingo formulę: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1+\Theta\left(\frac{1}{n}\right))$

Išvada. Rūšiavimas piramide ar suliejimo būdu – asimptotiškai optimalūs rūšiavimo algoritmai.

Įrodymas seka iš T8.1 ir viršutinių algoritmų Q(nlgn).

1.14 Tiesiniai rūšiavimo algoritmai

Tiesiniai rūšiavimo algoritmai. Kodėl jie lenkia asiptotiškai optimalius palyginimo algoritmus.

Rušiavimas skaičiuojant (counting sort)

Apribojimai: Rūšiuojami elementai gali turėti sveikas reikšmes iš intervalo O .. k, čia k-teigiama sveika konstanta.

Jei rūšiuojama n elementų rūšiavimas trunka k=O(n) tada T(n)=O~(n).

Counting_sort(A,B,k)

1. for i ← O to k
2. do C[i]←O
3. for j←1 to lenght[A]
4. do C[A[j]]←C[A[j]]+1
5. C[i] išsaugoma kiekis elementų lygių i.
6. for i ←1 to k
7. do C[i]←C[i]+C[i+1]
8. C[i]-išsaugoma elementų skaičių neviršinančių i.
9. for j←lenght[A] downto 1
10. do B[C[A[j]]]←A[j]
11. C[A[j]]←C[A[j]]-1

Pvz.:

1) A: $2^1 5^2 3^3 0^4 2^5 3^6 0^7 3^8$
   C: $2^0 0^1 2^2 3^3 0^4 1^5$ (3,4 eil)

2) C: $2^0 2^1 4^2 7^3 7^4 8^5$ (6,7 eil)

3) A: 2 5 3 0 2 3 0 3
   B: 1 2 3 4 5 6 3 7 8
   C: 2 2 4 6 7 8

Sudėtingumas, jei h=(n), T(n)=O~(k+1)+O~(n)+O~(k)+O~(n)=O~(n+k)=O~(n).

Pozicinis rūšiavimas (radix_sort)
1. for i←1 to d
2. do rūšiavimas pagal masyvo A i-taji skaitmenį
čia d-pozicijų skaičius
Sudėtingumas O~(d(n+k)), kai rūšiuojama n d pozicijų skaičiai su k galimų reikšmių, jei stabilus rušiavimas šio algoritmo atliekamas per O~(n+k).

1.15 Kišeninis rūšiavimas

<u>. Kišeninis rūšiavimas. Jo sudėtingumo įvertinimo įvertinimas.</u>
Kišeninis rūšiavimas (bucket_sort)
Apribojimas: rūšiuojami skaičiai intervale (0,1) ir beto tolygiai pasiskirstę.
bucket_sort(A)
1. n←lenght[A]
2. for i←1 to n
3. do įrašyti į sąrašą B[nA[i]]
4. for i ← to n-1
5. do rūšiavimas sąrašo B[i] sąrašo B[i] su įterpimu O(n^2)
6. sujungimas sąrašų B[0],B[1].....B[n-1]

Idėja sudėti masyvo A elementus į n sąrašų taip, kad elementai iš B[i] sąrašo būtų mažesni už kiekvieną B[j] sąrašo elementus, jei tik i<j. Kadangi skaičiai tolygiai pasiskirstę tai tikėtina kad tie sąrašai bus „maždaug"vienodi ir po to atlikus B[0]...B[n-1] sąrašų rūšiavimų suliejimas tiesiog nuoseklus surašymas. Tarkime, kad n_i-elem. Skaičius i-tajam sąraše. 1-6 išskyrus 5 vykdoma per O(n) laikų, o 5 per O(n^2), nes rūšiavimas atliekamas su įterpimu. $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$ Vidurkis

$$E(T(n)) = E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) = \Theta(n)$$

1.16 Hešavimas. Tiesioginis adresavimas.

<u>. Kokia „hešavimo" paskirtis. Heš lentelės su tiesioginiu adresavimu (suformuluoti uždavinį ir pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti). Aprašyti šio būdo trukumus.</u>

Heš- lenteles – tai efektyvus būdas i duomenų struktūras realizuojant žodynus t.y. dinaminiu aibių realizavimą su trimis operacijomis įterpimas, paieška ir pašalinimas.
Šios operacijos blogiausiu atveju realizuojamos per *O(n)* laiką, bet praktikoje, tai atliekama vidutiniškai *O(1)*.
Lentelės su tiesiogine adresacija:
Uždavinys: Tarkime, turime dinaminių elementų aibę, kiekvienas kurių turi raktą iš aibės *U={0,1,...,m-1}*, čia m nėra labai didelis. Be to, tarkime kad bet kurie du elementai neturi vienodų raktų.
Šiam uždaviniui realizuoti naudojamas masyvas (arba lentelė su tiesiogine adresacija), kurį pažymėsime kaip *T[0..m-1]*, kurio kiekviena pozicija ar ląstelė, atitinka raktą iš raktų erdvės *U*.
Paieška struktūroje:
Direct_Address_Search (T,k) *k-raktas*
1. return T[k]
Įterpimas i duomenų struktūrą:
Direct_Address_Insert (T,x) *x-rodyklė*
1. T[key[x]]←x
Pašalinimas:
Direct_Address_Delete (T,x)
1. T[key[x]]←nil
Visi veiksmai atliekami per *O(1)*.
Haš – lentelės:

Tiesioginės adresacijos trūkumas – efektyvi kai raktų erdvė *U* nedidelė, priešingu atveju *T* – masyvas tampa labai dideliu ir neracionaliai naudojamas, jei *k* - yra mažas lyginant su |*U*| (galimu raktų skaičiumi). Kitas blogas dalykas, kad raktai skirtingiems elementams turi nesikartoti. Taigi reikia rasti duomenų saugojimo struktūra, kurios saugojimui reikėtų *Θ(|k|)* ir pagrindiniai veiksmai su struktūra būtų atliekami vidutiniškai per *O(1)*laiko. (Pastaba: tiesioginiu atveju tai buvo blogiausiam atvejui).

Tam tikslui naudojama heš - funkcija *h*, kuri elementų raktus atvaizduoja į masyvą *T[0..m-1]* elementų indeksų aibę:

h : U→{0,1,...,m-1}

Sakysime, kad elementas su raktu k hešuojamas į poziciją *h(k)*. Dydis *h(k)* – vadinamas heš – reikšme rodyklei *k*.

Blogiausiu atveju, hešavimas su grandinėlėmis gali elgtis ypač nepalankiai, t.y. kai visi elementai hešuojami į vieną (tą pačią) grandinėlę. Tokiu atveju paieška užtrunka *Θ(n)* laiko ir plius laikas skirtas raktų heš reikšmių skaičiavimui.

Heš – f-jų *h* parinkimas turi būti toks, kad ši f-ja, kiek galima tolygiau paskirstytų elementus visoms masyvo *T* grandinėlėms.

1.17 Paprastas tolygus hešavimas.

Paprastas tolygus hešavimas. Suformuluoti teoremas apie paieškos laiko įvertinimą. Įrodyti vieną teoremą.

Paprastu tolygiu hešavimu vadinsime h funkciją, kuri paskirsto n elementų į m pozicijų po n_i reikšmių i – tajai pozicijai, kad $E[n_i] = \alpha = \frac{n}{m}$, kur $n = \sum_{i=0}^{m-1} n_i$.

Laikysime, kad h(k) reikšmė randama per Q(A) laiko tarpą. Paieška elemento sąraše T[h(k)] priklauso nuo jo ilgio $n_{h(k)}$. Jei neskaičiuosime kiek trunka rasti h funkcijos reikšmę, rasime vidutinį laiką palyginimų, kuriuos reikia atlikti vykdant paiešką.

Galimos dvi situacijos:

1. kai pieška nesėkminga
2. kai paieška sėkminga

Teoremos:

a) Heš lentelėje su kolizijų sprendimu grandinėlių pagalba matematinis vidurkis nesėkmingos paieškos laiko paprasto tolygaus hešavimo atveju yra $\theta(1+\alpha)$.

Įrodymas: Kiekvienas raktas k su vienoda tikimybe tolygaus hešavimo atveju gali būti patalpintas į vieną iš m pozicijų. Nesėkmingos paieškos atveju reikia atlikti paiešką iki galo sąraše T[h(k)], kurio ilgio $n_{h(k)}$ vidurkis $E[n_{h(k)}] = \alpha$. Tokiu būdu vidutiniškai tikrinamų elementų skaičius yra lygus α ir reikalingas laikas h(k) reikšmės radimui. Todėl bendros nesėkmingos paieškos laikas yra lygus $\theta(1+\alpha)$.

b) Heš lentelėje su kolizijų sprendimu grandinėlių pagalba matematinis vidurkis sėkmingos paieškos laiko paprasto tolygaus hešavimo atveju yra $\theta(1+\alpha)$.

1.18 Atviras adresavimas

Atviras adresavimas. Suformuluoti uždavinį, pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti. Kada tikslinga naudoti?

Šiuo atveju visi elementai saugomi heš lentelėje išvengiant rodyklių, t.y. bet kokia pozicija saugo elementą ar reikšme nil.

Šiuo atveju

h : U(raktų erdvė) x {0,1,…,m-1}(hešavimo bandymai)->{0,1,…,m-1}(heš reikšmė)

Beto reikalaujame, kad bet kookiu k raktų seka „bandymų“ < h(k,0),...,h(k,m-1)> yra aibės <0,1…,m-1> keitinys, kad būtų galima peržiūrėti visus heš lentelės elementus.

Veiksmai:

Hash_insert

1.i<-0
2. repeat j <-h(k,i)
3. if T[j] = nil (or T[j] = deleted)
4 then T[j] <- k
5. return j
6. else i<- i+1
7. Until i = m
8. error “Lentelė papildyta“

Hash_search(T,k)

1.i<-0

2.repeat j<-h(k,i)
3. if T[j] = k
4. then return j
5. i<-i+1
6.until T[j]=nil ar i=m
7.retun nil

Pašalinimas yra sudėtingas, n es nepakanka pažymėti elementų reikšmę nil, nes sugadinsime paiešką. Tai gaima apeiti žymint tokių elementų specialiu simboliu „deleted". Reikia modifikuoti has_insert procedūrą. Paieškos procedūros keisti nereikia. Tačiau šiuo atveju paieška nustoja priklausyti nuo užpildymo koeficiento alfa, todėl šis metodas netiakomas kai su duomenų struktūra reikia atlikinėti pašalinimo veiksmus

Tolimesnė analizė remiasi prilaidą apie tolygų hešavimą, t.y. kad bet kokokiam raktui bandymų seka vienodai galima iš visų m! variantų.

Tai sunku užtikrinti, bet naudojamos pakankamai geros aproksimacijos, kaip pvz dvigubas hešavimas.

2.1 Binariniai paieškos medžiai

Binariniai paieškos medžiai

Kiekvienas elementas turi tris laukus: `left, right` ir p, t. y. rodykles į vaikus ir tėvinį elementą.

Binarinis paieškos medis turi tenkinti savybę:

- Jei x – binarinio paieškos medžio viršūnė (mazgas), o y – kairės šakos mazgas, tada $key[x] \geq key[y]$;
- Jei x – binarinio paieškos medžio mazgas, o y – dešinės šakos mazgas, tada $key[x] \leq key[y]$.

Binarinio medžio savybė:

- Išvesti binarinio paieškos medžio raktus surikiuota tvarka, kuris vadinimas centriniu (simetriniu) medžio T apėjimu. Procedūra INORDER_TREE_WALK($root[T]$).

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

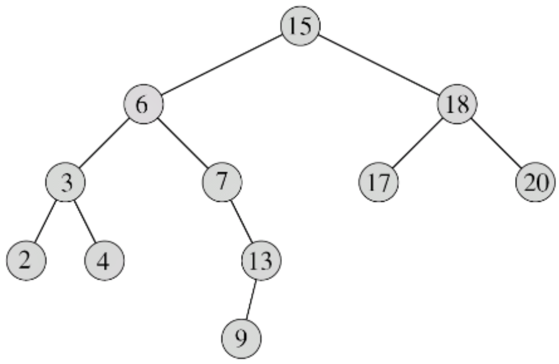
Galimi ir kiti medžio apėjimo būdai: tiesiogine tvarka, atvirkštine tvarka.

Minėtoje procedūroje INORDER_TREE_WALK keičiasi 3 eilutės vieta. Pirmuoju atveju ji tampa 2 eilute, o antruoju atveju 4 eilute.

Veiksmai su binariniu paieškos medžiu

- Paieška
- Minimumas ir maksimumas
- Prieš ir po einantys elementai (raktų atžvilgiu)

Paieška



Sudėtingumas $T(n)=O(h)$, čia h – medžio aukštis.

Procedūra paieškai binariniame medyje atlikti

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else  return TREE-SEARCH(right[x], k)
```

Minimumo ir maksimumo paieška

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
        do x ← left[x]
return x
```

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
        do x ← right[x]
return x
```

Prieš ir po einantys elementai

Tarkime, kad visi medžio mazgai, $x_1, x_2, \ldots, x_n$ surikiuoti didėjimo tvarka $\langle x'_1, x'_2, \ldots, x'_n \rangle$, taip kad $key[x'_1] < key[x'_2] < \ldots < key[x'_n]$.

Ieškant prieš einantį elementą, reikia rasti elementą y elementui x, tokį kad $key[x'_{i-1}] < key[x'_i]$, kai $y = x'_{i-1}$ $x = x'_i$.

Ieškant po einantį elementą, reikia rasti elementą y elementui x, tokį kad $key[x'_{i+1}] > key[x'_i]$, kai $y = x'_{i+1}$ $x = x'_i$.

Po einančio elemento paieškos algoritmas

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
   then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
     do x ← y
        y ← p[y]
return y
```

Įterpimas

Į medį T įterpiame elementą z. Pradinės reikšmės: $key[z] = v$, $left[z] = nil$ ir $right[z] = nil$.

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
       if key[z] < key[x]
          then x ← left[x]
          else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                    ▷ Tree T was empty
   else if key[z] < key[y]
          then left[y] ← z
          else right[y] ← z
```

Šalinimas

Iš medžio T šaliname elementą z.

Pradinės reikšmės: $key[z]=v$, $left[z]=nil$ ir $right[z]=nil$.

Galimos trys situacijos:

- z – medžio lapas;
- z – neturi vieno iš vaikų;
- z – turi abu vaikus.

```
TREE-DELETE(T, z)
if left[z] = NIL or right[z] = NIL
   then y ← z
   else  y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
   then x ← left[y]
   else  x ← right[y]
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else  if y = left[p[y]]
            then left[p[y]] ← x
            else  right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

Visos operacijos priklauso nuo medžio aukščio h.

- Blogiausias atvejis, kai medis išsigimsta į sąrašą.
- Geriausias, kai medis yra artimas pilnam binariniam medžiui, t. y. $h=O(\log_2 n)$.

Kai duomenų imtis randomizuojama galima pasiekti, kad sudaromo medžio vidutinis aukštis elgtųsi kaip geriausiu atveju (nenaudojama šalinimo veiksmų)

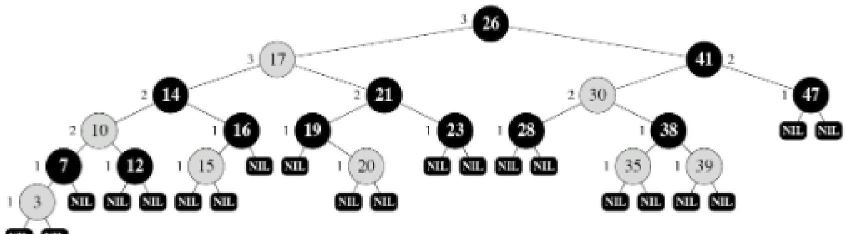
(Jeigu netingit gilintis, varykit čia, rasit pavyzdžių ir geriau suprasit klausimą ☺)

2.2 Raudonai-juodi binariniai paieškos medžiai

Šiuo atveju įvesime kiekvienam mazgui papildomą spalvą, t. y. $color[x]$.

Papildomi reikalavimai mazgams:

1. Kiekvienas mazgas raudonas arba juodas;
2. Medžio šaknis juodos spalvos;
3. Medžio lapai (*nil*) juodi;
4. Raudono mazgo vaikai yra juodi.
5. Kiekvieno mazgo visi keliai iki jo lapų turi vienodą kiekį juodų mazgų.



(a)

Įrodyti teoremą apie jų aukštį

Lema 13.1. Raudonai juodo medžio su n vidinių mazgų aukštis nedidesnis kaip $2\log_2(n+1)$.

Įrodymas.

Pažymėkime $bh(x)$ juodų mazgų skaičių nuo mazgo x (neįskaičiuojant jį) iki lapų.

Pirmiausia parodysime, kad kiekvienas mazgas keliuose iki lapų turi nemažiau $2^{bh(x)}-1$ vidinių mazgų.

Taikysime matematinę indukciją.

Kai aukštis x mazgo medyje yra lygus 0, jis turi būti lapas ($nil[T]$). Todėl jo medžio dalis, kurio jis yra šaknis turi savyje 0 vidinių mazgų, o $bh(x)=0$, taigi $2^{bh(x)}-1=2^0-1=0$.

(Lema – prielaida)

Tarkime medžio vidinis x mazgas turi du vaikus. Šiuo atveju, kiekvienas vaikas turi juodų mazgų aukštį $bh(x)$, jei jis raudonas arba $bh(x)-1$, jei jis juodas.

Remiantis indukcija kiekvienas vaikas turi nemažiau $2^{bh(x)}-1$ arba $2^{bh(x)-1}-1$ vidinių mazgų.

Mazgas x turi mažiausiai (įskaitant ir jį patį)

$$2^{bh(x)-1}-1+2^{bh(x)-1}-1+1=2^{bh(x)}-1$$

vidinių mazgų.

Pažymėkime h medžio aukštį. Remiantis 4 sąlyga nemažiau pusė kelio nuo šaknies iki lapų turi būti juodi mazgai, t. y. $\frac{h}{2}$ mazgų, nes po raudono būtinai eina juodas mazgas.

Visas medis turi $n \geq 2^{bh(root[T])}-1 \geq 2^{\frac{h}{2}}-1$ vidinių mazgų.

$n+1 \geq 2^{\frac{h}{2}}$,

$\log_2(n+1) \geq \log_2 2^{\frac{h}{2}} = \frac{h}{2}$,

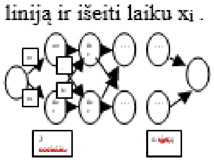
$h \leq 2\log_2(n+1)$.

(Jeigu norit pasigilinti)

2.3 Dinaminis programavimas: konvejeris

2. Dinaminis programavimas: Konvejeris (15.1)

Yra 2 gamybos linijos. Kiekviena iš linijų turi n stočių pažymėtų j=1,2,…n. Turime įėjimo laiką e_i kiekvienai važiuoklei, kad įeiti į liniją ir išeiti laiku x_i .



a_{ij} – sugaištama laiko;
t_{ij} – iš i-tojo konvejerio ir j-osios vietos į priešingą konvejerį;

f_i [j] – laikas i-tojo konvejerio iki j-tosios pozicijos
tada f_1 [j]= min {f_1 [j-1] +a_{1j}, f_2 [j-1]+t_{2j} +a_{1j}} j>1 ;
f_2 [j]= min {f_2 [j-1] +a_{2j}, f_1 [j-1]+t_{1j} +a_{2j}} j>1.

Laikas sugaištamas iki pirmo konvejerio: f_i[1]=e_i + a_{i1}, j=1
Sudarom 2 masyvus: f_i(j), kur i=1,2; e(j) – į tą vietą iš kurios pozicijos patekti.
Tada Bendras optimalus spendinys – f* = min {f_1[n], f_2[n]}

2.4 Dinaminis programavimas: matricų daugybos tvarka

3. **Dinaminis programavimas: Marticų daugybos tvarka (15.2)**

Turim seką $A_1, A_2, \ldots A_n$ n matricų, kurias norėsime sudauginti ir norime įvertinti produktą $A_1A_2 \ldots A_n$
Pvz.: jei turim matricų grandinę A1,A2,A3,A4, tai produktui A1A2A3A4 skliaustai gali būti sudėti keliais būdais, pora paminėsiu: (((A1 A2) A3) A4). ((A1 (A2 A3)) A4) ir t.t. Mūsų tikslas yra sudėti skliaustus taip, kad veiksmų skaičius būtų minimalus.

```
MATRIX-MULTIPLY(A, B)
if columns[A] ≠ rows[B]
then error "incompatible dimensions"
else for i ← 1 to rows[A]
do for j ← 1 to columns[B]
do C[i, j] ← 0
for k ← 1 to columns[A]
do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
return C
```

Galima sudauginti matricas A ir B tik jei jos yra suderinamos: stulpelius skaičius A matricoje turi būti lygus B eilučių skaičiuj. Jei A yra p x q matrica ir B yra q x r matrica, tada rezultate gausim C matricą p x r.
Tarkim turim trijų matricų grandinę A1,A2,A3. Atitinkamai jų dydžiai yra 10×100, 100×5, ir 5×50. Jei dauginsim pagal pirmą skliaustų sudėjimo variantą ((A1 A2) A3), tai atliksim $10 \cdot 100 \cdot 5 = 5000$ skaliarinių daugybų apskaičiuoti 10×5 matricos produktą A1 A2, plius kitus $10 \cdot 5 \cdot 50 = 2500$ skaliarinės daugybos, kad sudauginti šią matricą iš A3, bendroj sumoj 7500 skaliarinių daugybų. Jei dauginsim pagal (A1 (A2 A3)), tai atliksim $100 \cdot 5 \cdot 50 = 25{,}000$ skaliarinių daugybų suskaičiuoti 100×50 matricos produktą A2 A3, plius kitas $10 \cdot 100 \cdot 50 = 50{,}000$ skaliarinių daugybų iš A1, sumoj gaunam 75,000 skaliarinių daugybų. Taigi, skaičiuojant pirmu būdu gaunam rezultatą 10 kartų greičiau.

Tačiau tikrinimas visų galimų skliaustų sudėjimo variantų neduos efektyvaus algoritmo. Pažymim skaičių alternatyvių variantų skliaustų sudėjimo n sekų kaip P(n). Tada gaunam rekurentinę formulę:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \,., \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \,.. \end{cases}$$

panašios formulės augimas yra $\Omega(4n/n3/2)$
Rekursyvinio apibrėžimo minimaliam laikui skliaustų sudėjimo produktui Ai Ai+1 Aj yra:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \,., \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \,.. \end{cases}$$

The m[i, j] values give the costs of optimal solutions to subproblems

2.5 Dinaminio programavimo elementai

1. **Dinaminis programavimo elementai (15.3 skyreliai: įvadas, optimali substruktūra, pagalbinių uždavinių perdengimas)**

Įvadas:
Kada galima taikyti dinaminį programavimą?
Požymiai: 1) buvimas optimalios substruktūros ir perdengiančios pagalbinės programos.
Optimali substruktūra:
Optimali substruktūra atsiranda tuo atveju, jei jos optimalus sprendinys apima (turi) pagalbinių uždavinų optimalius sprendinius. Jei uždavinyje surandama optinalios struktūros buvimas, tai svarus argumentas, kad uždavinys gali būti sprendžiamas kaip din. prog. uždavinys. Din. prog. optimalus sprendinys formuojamas iš optimalių sprendinių. Konvejaryje kelias iki j-tosios darbo vietos bus optimalus, jei optimalus bus kelias iki j-1 vietos. Matricų daugybos atveju (skliaustelių dėliojimas) parodytų, kad optimali substruktūra $A_i\ A_{i+1} \ldots A_j$ dalinasi į dvi sekas. Tarp matricų A_k ir A_{k+1} ir bus optimalai padalinta jei sekos $A_i\ A_{i+1}\ A_k$ ir $A_{k+1}\ A_{k+2} \ldots A_j$ bus optimalios, t.y. bus sudėti skliausteliai tokiu būdu: $(A_i\ A_{i+1} \ldots A_{k-1}\ A_k)(A_{k+1}\ A_{k+2} \ldots A_j)$
Optimalios struktūros pasirinkimas atliekamas pagal schemą:
1)Pirmame etape reikia parodyti, kad sprendimo procese reikia atlikti pasirinkimą.
2)Antrame etape laikome, kad pasirinktam uzdaviniui daromas pasirinkimas vedantis į optimalų sprendinį.
3)Nustatoma kokie pagalbiniai uždaviniai gaunami ir kaip geriausiai charakerizuoti gaunamų uždavinių erdvę.
4)Parodyti, kad sprendžiant pagalbinius uždavinius, gaunamus ieškant optimalaus sprendinio, jie patys taip pat turi būti optimalūs.
Pagalbinių uždavinių perdengimas:
Din. Prog. atveju pagalbinių uždavinių erdvė turi būti „nedidelė". t.y. kad rekursyvinio sprendimo algoritme pasikartotų tie patys uždaviniai. Kaip taisyklė, pagalbinių uždavinių kiekis yra polinominė įeinančių duomenų kiekio f-ja. Dalinimo uždaviniuose atsiranda visai nauji uždaviniai, kas yra skirtumas tarp dinaminio programavimo, kuris gali pirma išspręsti mažesnius uždavinius, jų rezultatus išsaugoti, o poto naudoti kitų sprendime.

2.6 Godūs algoritmai

4. Godūs algoritmai: Procesų pasirinkimo uždavinys (16.1)

Godūs algoritmai – tai greitesni ir efektyvesni algoritmai lyginant su dinaminio programavimo uždaviniais. Jų esmė: kiekvienu momentu daromas sprendimas, kuris tuo momentu yra geriausias.
Turim aibę procesų:
$S=\{a_1, a_2,\ldots,a_n\}$
Procesai naudoja vieną ir tą patį resursą laike $0 \leq s_i < f_i < \infty$
Reikia rasti max poaibį aibės S, kurio procesų darbo laikai persidengia t.y. $[s_i, f_i) \cap [s_j, f_j) = 0$
kiekvienam a_i, $a_j \in s'$, $i \neq j$,
***i* 1 2 3 4 5 6 7 8 9 10 11**
si 1 3 0 5 3 5 6 8 8 2 12
fi 4 5 6 7 8 9 10 11 12 13 14

$\{a_1,a_4,a_8,a_{11}\}$
$\{a_{21},a_4,a_9,a_{11}\}$

Pirma sudarysime optimalią struktūrą:
Tarkim, kad $Sij = \{ak \in S : fi \leq sk < fk \leq sj\}$, kuri apima visus procesus, kurie gali spėti baigtis tarp procesų a_i ir a_j (a_i – pabaigos , a_j - pradžios). Papildom fiksuotais procesais a_0 ir a_{n+1} , kur a_0 pabaiga $f_0 = 0$, o a_{n+1} pradžia
$S_{n+1} = \infty$. Tada $S = S_{0,n+1}$. Beto laikome, kad $f_0 \leq f_1 \leq \ldots \leq f_n \leq f_{n+1}$ tada $Sij=\varnothing$, jei $i>=j$.
Sakykime turime netuščią uždavinį S_{ij} ir tarkim, kad jos sprendinyje bus proc. a_k
Procesas a_k generuoja du papildomus uždavinius S_{ik} ir S_{kj}. Tai S_{ij} sprendinys bus dydžio $|S_{ik}| + |S_{kj}| +1$. Pažymėkime uždavinio S_{ij} optimalų sprendinį A_{ij}, kuriame yra procesas a_k Todėl sub. uždavinių S_{ik} opt. sprendinys A_{ik} , o S_{kj} – A_{kj} taip pat turi būti optimalūs. Todėl optimalus sprendinys turintis savyje procesą a_k bus toks: $A_{ij} = A_{ik} \cup A_{kj} \cup \{a_k\}$
Rekursyvus sprendimas:
Antrame etape nustatom reikšmes, atitinkančias optimaliam sprendiniui. Tarkim c[i,j] – kiekis procesų maksimaliam poaibyje uždavinio S_{ij} $c[i,j] = 0$, kai $S_{ij} = 0$, kai $i \geq j$. Iš optimalaus sprendinio struktūros seka, kad $c[i,j] = c[i,k] + c[k,j] +1$ kai sprendinyje yra procesas a_k.

2.7 Amortizacinė algoritmų analizė. (17sk. žinoti metodų idėjas ir pateikti pavyzdžius) (bus papildyta vėliau)

2.8 Paieška į plotį

8.Paieška į plotį (22.2 be teiginių įrodymo)

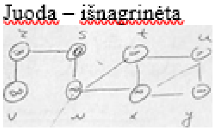
Šio algoritmo tikslas rasti visas pasiekiamas viršūnes iš s (s priklauso V) viršūnės ir kartu rasti atstumą bei trumpiausią kelią (tarsi reiktų susumuoti visus svorius, o šiuo atveju w = 1 kiekvienai (u, v), priklausančiai E). Tikslas: suformuoti medį į plotį, t.y. pirma randamos visos viršūnės, pasiekiamos per atstumą k, ir tik po to per atstumą k + 1.

```
BFS(G, s)
1  for each vertex u ∊ V [G] - {s}
2      do color[u] ← WHITE
3         d[u] ← ∞
4         π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← Ø
9  ENQUEUE(Q, s)
10 while Q ≠ Ø
11     do u ← DEQUEUE(Q)
12        for each v ∊ Adj[u]
13            do if color[v] = WHITE
14                then color[v] ← GRAY
15                     d[v] ← d[u] + 1
16                     π[v] ← u
17                     ENQUEUE(Q, v)
18        color[u] ← BLACK
```

Balta – neaplankyta

Pilka – viršūnė kuria reikia ištyrinėti eilėje (Q)

Juoda – išnagrinėta

 2 schema

Q={s} atstumas iki s

Analizė

BFS

Enq- 0(1)

Deq- 0(1)

Idėti reikia tik po vieną kartą –baltas

Viršūnės, kurios iškarto perdažomos, o kai išimamos, tai taip pat po vieną kartą ir perdažomas juodai.

Inicializacija O(V) eilutėse 12-18 dirba Θ(E), nes ilgis visų yra |E|. Gauname, kad darbo laikas BFS O(V+E) yra tiesinė priklausomybė nuo grafo G dydžio.

2.9 Paieška į gylį

9.Paieška į gylį (22.3 be teiginių įrodymo)

```
DFS(G)
1.     for ∀ u ∈ V[G]
2.     do color[u] ← balta
3.     µ[u] ← nil
4.     time ← 0
5.     for ∀ u ∈ V[G]
6.     do if color[u] = balta
7.     then DFS_Visit(u)

DFS_Visit(u)
1.     color[u] ← pilka
2.     time ← time + 1
3.     d[u] ← time
4.     for ∀ v ∈ Adj[u]
5.     doifcolor[v] ← balta
6.     then µ[v] ← u
7.                DFS_Visit(v)
8.     color[u] ← juoda
9.f[u] ← time ← time + 1
```

Sudėtingumas

```
DFS(G)
for each vertex u ∈ V[G]
      do color[u] ← WHITE
         π[u] ← NIL
time ← 0
```

$\Theta(V)$

```
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)
```

Vykdoma $|V|$ kartų.

Briaunų klasifikacija

Išskirsime keturias briaunų tipus paieškoje į gylį:

1. Medžių briaunos (tree edges) – briaunos priklausančios paieškos įgylį medžiui G_π .
2. Grįžimo briaunos (back edges) briaunos jungiančios viršūnes su jų protėviais paieškos įgylį medyje.
3. Tiesioginės briaunos (forward edges) – briaunos nesančios medžio briaunomis jungiančios viršūnes su jos palikuonimis paieškos įgylį medyje.
4. „*Kertančios*“ briaunos (cross edges) – likusios briaunos, kurios jungia viršūnes kurios nėra protėviu viena kitai arba skirtingų medžių viršūnes.

22.10 teorema. Paieškoje į gylį neorentuotame grafe G betkuri briauna yra paieškos į gylį medžio briauna arba grįžimo briauna.

2.10 Topologinis rūšiavimas

10. **Topologinis rūšiavimas (22.4 be teiginių įrodymo)**

Topologinis rūšiavimas gali būti pritaikytas necikliniam orientuotam grafui. Grafo G=(V,E) topologinis rūšiavimas yra tiesinis visų jo viršūnių surūšiavimas taip, kad jei grafas G turi briauną (u,v), tai u(viršūnė) surūšiuotame sąraše eina prieš v. Taigi, topologinį rūšiavimą galima suprasti kaip grafo G viršūnių išdėliojimą pagal horizontalią liniją taip, kad visos briaunos būtų nukreiptos ir kairės į dešinę.
Supaprastintai tokio rūšiavimo algoritmas atrodytų taip:
1)Paieškos gilyn algoritmu apskaičiuojame pabaigos laikus f[v] kiekvienai viršūnei v
2)Kai kiekviena viršūnė baigta, ją dedame į susieto sąrašo priekį.
3)Grąžiname viršūnių sąrašą
Topologinės paieškos sudėtingumas Θ (V+E). Taip yra, nes topologinio rūšiavimo algoritmas remiasi paieškos gilyn algoritmu (sudėtingumas Θ(V+E)) ir dar užima O(1) tam, kad įdėti reikiamą viršūnę į sąrašo priekį.
Pagrindinis topologinio rūšiavimo pritaikymas yra parodymas kažkokių įvykių sekos, kurie yra aprašyti grafu, tai yra, parodyti kas po ko eina.

2.11 Minimalūs padengiantys medžiai

12. Minimalūs padengiantys medžiai (23.1 be teiginių įrodymo)

Turim g=(v,E)- jungų neorentuotą grafą su w:E→R svorio f-ja. Tegul A ⊂ E, kuri priklauso kokiam nors min padengt medžiui grafo G, o (s, v-s)-pjuvis G, suderintas su A, o (u, w) lengva briauna, priklausanti pjuviui (s, v-s), tada (u,v) yra saugi briauna A.
Minimal padeng medį galima rasti 2 algoritmais: Kruskalo arba Primos. Abu ya godūs. Jų darbo laikas O(E*lgV). Naudojant fibonatines piramides, galima pasiekti, kad Prima algoritmo darbo laikas būtų O(E+lgV), kai |B|<<|E|.

Bendras algoritmas:

Generic_MST (G,w)
1. A <- Ø – min. pag. medžio briaunos
2. while A nėra min. pag. medis
3. do Rasti saugią briauną (u,v) medžiui A
4. A <- A ∪{(u,w)}
5. Return A

Teorema 23.1

Turim G=(V,E), jungų neorientuotą grafą su w: E->R svorio f-ja. Tegul A ⊆ E, kuris priklauso kokiam nors min. padeng. medžiui grafo G, o (S,V-S) – pjūvis G, suderintas su A, o (u,w) lengva briauna, priklausanti pjūviui (S,V-S). Tada (u,v) yra saugi briauna A.

2.12 Kruskalo algoritmas

13. Kruskalo algoritmas. (23.2)

```
MST-KRUSKAL(G, w)
1 A ← Ø
2 for each vertex v _ V[G]
3 do MAKE-SET(v)
4 sort the edges of E into nondecreasing order by weight w
5 for each edge (u, v) _ E, taken in nondecreasing order by weight
6 do if FIND-SET(u) ≠ FIND-SET(v)
7 then A ← A _ {(u, v)}
8 UNION(u, v)
9 return A
```

1-3 eilutės pažymi aibę A kaip tuščią aibę ir sukuria |V| medžius, kurie talpina po vieną viršūnę kiekvienas. Briauna esanti E yra surūšiuojama į nemažėjančią tvarką svoriu, kuris yra 4 eilutėje. For ciklas 5-8 eilutėse tikrina, kiekvienai briaunai (u,v) ar galiniai taškai u ir v priklauso tam pačiam medžiui. Jei priklauso, tai briauna (u,v) negali būti įtrauktą į mišką nesukuriant ciklo ir briauna yra atmetama. Kitu atveju dvi viršūnės priklauso skirtingiems medžiams. Šiuo atveju briauna (u,v) yra įtraukiama į A 7 eilutėje ir viršūnės dviejuose medžiuose yra sujungiamos 8 eilutėje.

2.13 Prima algoritmas

14. Prima algoritmas (23.2)

```
MST-PRIM(G, w, r)
1 for each u _ V [G]
2 do key[u] ← ∞
3 π[u] ← NIL
4 key[r] ← 0
5 Q ← V [G]
6 while Q ≠ Ø
7 do u ← EXTRACT-MIN(Q)
8 for each v _ Adj[u]
9 do if v _ Q and w(u, v) < key[v]
10 then π[v] ← u
11 key[v] ← w(u, v)
```

1-5 eilutėse nustatomas raktas kiekvienos viršūnės iki begalybės, kiekvienos viršūnės tėvas nustatomas į NIL ir vykdom min-pirmenybės eilę Q, kas patalpinti visas viršūnes. 6-11 eilutėse kiekvienos iteracijos while ciklo:
1. 1. A = {(v, π[v]) : v _ V - {r} - Q}.
2. Viršūnės jau patalpintos į min apimties medį yra tos, kurios yra V-Q.
3. Visoms viršūnėms v _ Q, if π[v] ≠ NIL, then key[v] < ∞ and key[v] yra svoris lengvos briaunos (v, π[v]) sujungiant v su keliom viršūnėm jau padėtom į min apimties medį.

2.14 Trumpiausi keliai iš vienos viršūnės. Belmano-Fordo algoritmas

15. Trumpiausi keliai iš vienos viršūnės. Belmano-Fordo algoritmas (24 skyreliai: įvadas, optimali uždavinio apie trumpiausią kelią struktūra; 24.1 be teiginių įrodymo)

Turim G = (V, E) orientuotą grafą su svorių f-ja ω:E→R

Kelio p = <v_0, v_1, v_2, …v_p> yra $w(p) = \sum_{i=1}^{k} w(vi-1, vi)$

Trumpiausias kelias iš u į v yra $\delta(u,v) = \begin{cases} \min\{\omega(p) : u \xrightarrow{p} v\} \\ \infty \end{cases}$

Optimali struktūra uždavinio apie trumpiausią kelią.
Jei p=<v1, …, vp> - trumpiausias kelias iš v1 į vp grafe G=(V, E). $\omega : E \to R$ o pij = <vi, vi+1, …, vj> - dalinis kelias (kelio fragmentas) kelio iš v1 į vp. Tada pij – trumpiausias kelias iš vi į vj-1
Trumpiausiam kelyje ciklai negalimi. Jei yra ciklas teigiamas trumpiausiam kelyje, jį pašalinus rasime dar trumpesnį kelią. Nulinio svorio ciklus galime ignoruoti, o neigiamo svorio ciklai iš karto duoda, kad trumpiausio kelio svoris yra -∞. Todėl reikalaujama, kad neigiamais svoriais ciklų nebūtų.

<u>Pagalbinės funkcijos:</u>

Initialize_single_source(G, s)

```
1. for ∀v∈V[G]
2.   do d[v] ← ∞
3.      Π[v] ← nil
4. d[s]=0
```

Relax(u, v, ω)

```
1. if d[v]>d[u]+ ω(u, v)
2.   then d[v] ← d[u]+ ω(u, v)
3.        Π[v] ← u
```

Universalus algoritmas(su neigiamais svoriais) Belmano-Fordo
Turim orientuotą grafą G=(V, E) ir ω: E → R
Algoritmas grąžins "true" jei nebus rasta neigiamo svorio ciklų.
Sudėtingumas O(VE)
Belmano_Ford(G, s) (G – grafas, s – pradinė viršūnė)

```
1.Initialize_Single_Source(G, s)
2.  For i ← 1 to |V[G]|-1
3.    do for ∀(u,v)∈E[G]
4.       do Relax(u, v, ω)
5.  for ∀(u,v)∈E[G]
6. do if d[v] > d[u] + ω(u,v)
7. then return "false"
8. return "true"
```

<u>Trumpiausias kelias iš vienos viršūnės orientuotuose neecikliniose grafuose</u>

Dog_Shortest_Paths(G, w, s) G – grafas, w – s – pradinė viršūnė

```
1. Topologinis rūšiavimas G grafo
2.Initialize_Single_Source(G, s)
3. for (kiekvienai u viršūnei iš topologinio rūšiavimo )
4.do for ∀v∈Adj[u]
5.  do Relax(u, v, w)
Sudėtingumas O(V+E)
```

2.15 Trumpiausi keliai iš vienos viršūnės orientuotame necikliniame grafe.

Trumpiausi keliai iš vienos viršūnės orentuotame necikliniame grafe

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    do for each vertex v ∈ Adj[u]
        do RELAX(u, v, w)
```

Sudėtingumas $\Theta(V+E)$.

Pavyzdys:

16. Deiksra algoritmas (24.3 be teiginių įrodymo)

Deiksra algoritmas – turi G=(V,E) orentuotą grafą ir ω:E→R, t y ω(u, v)>0 su visais u, v $\in$ E.

```
Deisktra(G, ω, s)
1. Initiolize_Single_Source(G, s)
2.     S← Ø
3.       Q← V≠ Θ
4.       while Q≠ Ø
5.   do u← Extract_Min(Q)
6.          S← S U {u}
7.  for su visais v← Adj[u]
8.     do Relax(u, v, w)
```

čia S – aibė viršunių, kuriomis rasti svoriai(galutiniai).

Sudėtingumas $O(V^2+E)= O(V^2)$, bet tai priklauso nuo eiles Q reiksmes.

2.16 Trumpiausių kelių paieška tarp visų viršūnių taikant dinaminį programavimą

$$w_{ij} = \begin{cases} 0 & \text{if } i = j , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E . \end{cases}$$

Spręskime, kaip dinaminio programavimo uždavinį.

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j , \\ \infty & \text{if } i \neq j . \end{cases}$$

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1\le k\le n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} . \end{aligned}$$

$W = (w_{ij})$

$L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, $m = 1, 2, \ldots, n-1$

$L^{(m)} = (l_{ij}^{(m)})$

```
EXTEND-SHORTEST-PATHS(L, W)
n ← rows[L]
let L′ = (l′_ij) be an n × n matrix
for i ← 1 to n
     do for j ← 1 to n
            do l′_ij ← ∞
               for k ← 1 to n
                   do l′_ij ← min(l′_ij, l_ik + w_kj)
return L′
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

$$\begin{aligned} l^{(m-1)} &\rightarrow a\,, \\ w &\rightarrow b\,, \\ l^{(m)} &\rightarrow c\,, \\ \min &\rightarrow +\,, \\ + &\rightarrow \cdot \end{aligned}$$

```
MATRIX-MULTIPLY(A, B)
n ← rows[A]
let C be an n × n matrix
for i ← 1 to n
     do for j ← 1 to n
            do c_ij ← 0
               for k ← 1 to n
                   do c_ij ← c_ij + a_ik · b_kj
return C
```

$$\begin{aligned} L^{(1)} &= L^{(0)} \cdot W &= W\,, \\ L^{(2)} &= L^{(1)} \cdot W &= W^2\,, \\ L^{(3)} &= L^{(2)} \cdot W &= W^3\,, \\ &\vdots \\ L^{(n-1)} &= L^{(n-2)} \cdot W &= W^{n-1}\,. \end{aligned}$$

Sudėtingumas $\Theta(n^4)$

2.17 Fordo-Fulkensono metodas. Liekamasis tinklas. Srautą didinantis kelias

Fordo-Fulkersono metodas

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p
3      do augment flow f along p
4  return f
```

Plačiau

2.18 NP-pilnumas bei P ir NP uždavinių klasės

NP–pilnumas bei P ir NP uždavinių klasės

P uždavinių klasė – uždaviniai, kurie sprendžiami per polonominę laiko trukmę $O(n^k)$, čia k – konstanta, o n – įvedamų duomenų kiekis.

NP uždavinių klasė – uždaviniai, kurie pasiduoda *patikrinimui* per polonominę laiko trukmę. Tai yra jei, kokiu nors būdu buvo gautas sprendinio *sertifikatas*, tai jo teisingumą galima patikrinti per polonominę laiko trukmę tokio sprendinio korektiškumą.

$P \subseteq NP$, nes P – uždavinio sprendinys gaunamas per polinominį laiką, net ir neturint sprendinio sertifikato.

Uždavinys yra NP pilnas, jei jis priklauso NP uždavinių klasei ir toks pat *sudėtingas* kaip ir bet kuris kitas NP uždavinys.

Taigi jei egzistuoja bent vienam NP pilnam uždaviniui polinominis sprendimo algoritmas, tai bet kuriam kitam šios klasės uždaviniui egzistuoja toks algoritmas.