1. Rūšiavimas su įterpimu. Įrodyti algoritmo korektiškumą ir jo sudėtingumą kai duomenys įvedami palankiausiu atveju.
2. Įrodyti algoritmo korektiškumą ir jo sudėtingumą kai duomenys įvedami nepalankiausiu atveju.
3. Algoritmų sudarymo būdai.. Kuo jie skiriasi?. Kokia idėja rūšiavimo algoritmo suliejimo būdu.
4. Įrodykite rūšiavimo algoritmo suliejimo būdu korektiškumą.
5. Rūšiavimo algoritmo suliejimo būdu sudėtingumo įrodymas.
6. Funkcijų augimo asimptotiniai žymėjimai ir jų apibrėžimai.
7. Rekurentinių sąryšių sprendimo būdai (aprašyti idėją). Suformuluoti Pagrindinę teoremą.
8. Dekompozicinių algoritmų sudėtingumo T(n) skaičiavimo formulės struktūra ir sprendimo būdai.
9. Rūšiavimo piramide algoritmo idėja. Kokia duomenų struktūra – „piramidė“? Kaip priklauso piramidės dydis ir aukštis nuo rūšiuojamų duomenų kiekio?
10.Rūšiavimo piramide algoritmo idėja. Kaip atliekamas piramidės sutvarkymas (pateikite algoritmą ir sudėtingumą įrodykite).
11. Greito rūšiavimo algoritmo idėja. Kaip atliekama atraminio elemento paieška?
12. Greito rūšiavimo algoritmo idėja. Įrodykite algoritmo sudėtingumą blogiausiu atveju.
13. ????Greito rūšiavimo algoritmo idėja. Kada greito rūšiavimo algoritmo sudėtingumas ir rūšiavimo piramide sudėtingumo asimptotiniai įverčiai konstantos tikslumu sutampa? Įrodykite kokiam nors atvejui (rūšiavimo piramide sudėtingumo asimptotinio įverčio įrodyti nereikia).!!??
14. Optimalūs rūšiavimo algoritmai naudojantys palyginimus. Šių algoritmų įvertinimo blogiausiam atvejui įrodymas. Įrodyti, kad rūšiavimas piramide ir suliejimo būdu yra asimptotiškai optimalūs algoritmai.
15.Tiesiniai rūšiavimo algoritmai. ????Kodėl jie lenkia asiptotiškai optimalius palyginimo algoritmus.!!??
16. Rūšiavimas skaičiuojant. Jo sudėtingumo įvertinimo įrodymas.
17. Tiesiniai rūšiavimo algoritmai. Kodėl jie lenkia asiptotiškai optimalius palyginimo algoritmus.
18. Kišeninis rūšiavimas. Jo sudėtingumo įvertinimo įvertinimas.
19. Kokia „hešavimo“ paskirtis. Heš lentelės su tiesioginiu adresavimu (suformuluoti uždavinį ir pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti). Aprašyti šio būdo trukumus.
20. Heš funkcijos ir kolizijų sprendimas grandinėlių metodu. Koks privalumas prieš heš lentelės su tiesioginiu adresavimu. Suformuluoti uždavinį, pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti.
21. Paprastas tolygus hešavimas. Suformuluoti teoremas apie paieškos laiko įvertinimą. Įrodyti vieną teoremą.
22. Universalus hešavimas idėja ir pateikti vieną universalaus hešavimo funkcijų klasę. Suformuluoti teoremą apie raktą atitinkančios grandinėlės ilgį.
23. Atviras adresavimas. Suformuluoti uždavinį, pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti. Kada tikslinga naudoti?
24.Tiesinis, kvadratinis, dvigubas, idealus hešavimai.

1. 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 $a`_1 \leq a`_2 \leq \dots \leq a`_n$. 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 $A[i] \leq key$. Todėl algoritmas bus sukamas j = 2, … n kartų, o $t_j = 1$ tad darbo laikas pačiu palankiausiu atveju:

$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = an + b$, čia a ir b – konstantos, priklausančios nuo c_i .Šiuo atveju rūšiavimo laikas $T(n)$ yra tiesinė funkcija, priklausanti nuo n (duomenų kiekio).

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

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

$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_7(\frac{n(n-1)}{2}) + c_8(n-1) = an^2 + bn + c$, čia a, b, c - konstantos, priklausančios nuo c_i.Šiuo tai kvadratinė priklausomybė nuo n $T(n^2)$

3. Algoritmų sudarymo būdai.. Kuo jie skiriasi?. Kokia idėja rūšiavimo algoritmo suliejimo būdu.

Du būdai:

1. Nuosekliai didėjantis (inkrementinis) būdas. Pirma rūšiuojama A[1..j-1], o poto A[1..j].
2. Dekompozicinis(„skaldyk ir valdyk“). Rūšiavimo uždavinio atvejo pavyksta sumažinti laiką reikalingą blogiausiam įvykiui surūšiuoti. Dauguma tokių algoritmų turi rekursyvinę struktūrą. Dalinimo metodas – sudėtinga užduotis dalinama į keletą paprastesnių tačiau panašių į pradinę užduotį, tik su mažesniu duomenų kiekiu. Po to šių uždavinių rezultatai kombinuojami norint gauti pradinio uždavinio sprendinį. Dekompozicinio metodo schema susideda iš 3 etapų:

1. Dalinimas – į kelis uždavinius.
2. Pavergimas – rekursyvinis sprendimas. Kai uždavinys tampa mažas sprendžiama tiesiogiai.
3. Kombinavimas – sudaromas sprendinys iš pagalbinių uždavinių sprendinių suliejimo.

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 ($p \le q < r$). 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. n1 <- q-p+1
2. n2 <- r-q
3. sudaromi masyvai L[1.. n1+1] ir R[1.. n2+1]
4. for i <- 1 to n1
5.     do L[i] <- A[p+i-1]
6. for j <-1 to n2
7.     do R[j] <- A[q+j]
8. L[n1+1] <- ∞
9. R[n2+1] <- ∞
10. i <-1
11. j <-1
12. for k <-p to r
13.      do if L[i] ≤ 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)
```

4. Įrodykite rūšiavimo algoritmo suliejimo būdu korektiškumą.

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 $L[i] \le R[j]$, 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.

5. Rūšiavimo algoritmo suliejimo būdu sudėtingumo įrodymas.

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 $n\log_2 n$.

6. Funkcijų augimo asimptotiniai žymėjimai ir jų apibrėžimai.

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

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

Apibrėžimas 3. Sakysime, kad f(x) = Θ(g(x)) (x->∞), jei $c_l \neq 0$, $c_2 \neq 0$, 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)~g{x)* (x->∞), jei $lim_{x->\infty} f(x)/g(x) = l$. Tai griežčiausias iš visų minėtų apibrėžimas; pvz., $(3x+l)^4 \sim 81x^4$, $\sin(l/x)\sim l/x$, $2^x+7\log x + \cos x\sim 2^x$.

Apibrėžimas 5. Sakysime, kad f(x) = Ω(g(x)) (x->∞), jei ε> 0 : $|f(x)|> \varepsilon g(x)$. Tai atvirkščią reikšmę nei *o* turintis dydis, ir naudojantis funkcija *g(x)* apibrėžiamas apatinis r ė ž i s funkcijai f*{x)*.

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

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

7. 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 spręndinys 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 ≥ 1 ir b ≥ 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 $f(n) = O(n^{\log_b a-\varepsilon})$, ε >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 $f(n) = \Omega(n^{\log_b a+\varepsilon})$, ε >0, ir jei *af(n/b) < cf(n),* kai c <1 ir pakankamai dideliems n, tada $T(n) = \Theta(f(n))$

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

Dekompoziciniai algoritmai dažniausiai įgyvendinami rekursyviai, tokio algoritmo darbo laikas:

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

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

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

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.

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+; frąžins 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)] $\geq$ A[i], o nemažėjančios, jei A[Parent(i)] $\leq$ 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).

10.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 $\leq$ heap_size[A] ir A[l] > A[i]
4. then largest <- l
5. else largest <- i
6. if r $\leq$ heap_size[A] ir A[r] > A[largest]
7. then largest <- r
8. if largest $\neq$ 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) $\leq$ 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($n^{\log_{\frac{3}{2}} 1} lgn$) = O($lgn$)
T(n) = T*(n) = O(lgn) => T(n) = O(lgn)

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.

12. 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)

13. ???!!Greito rūšiavimo algoritmo idėja. Kada greito rūšiavimo algoritmo sudėtingumas ir rūšiavimo piramide sudėtingumo asimptotiniai įverčiai konstantos tikslumu sutampa? Įrodykite kokiam nors atvejui (rūšiavimo piramide sudėtingumo asimptotinio įverčio įrodyti nereikia).!!??

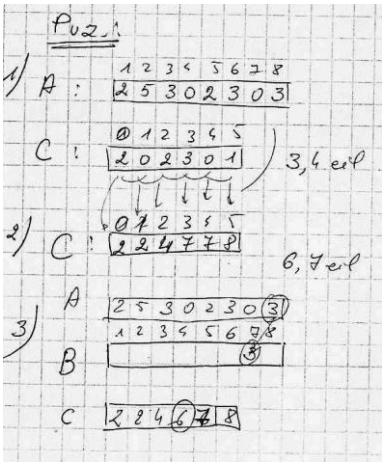
14. Optimalūs rūšiavimo algoritmai naudojantys palyginimus. Šių algoritmų įvertinimo blogiausiam atvejui įrodymas. Įrodyti, kad rūšiavimas piramide ir suliejimo būdu yra asimptotiškai optimalūs algoritmai.
Rūšiavimo naudojant palyginimus darbo laikas. a_i ir a_j galima palyginti šiais ženklais: <,≤,=,≥,>. Apie $<a_1,a_2,....,a_n>$ 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 Ω(nlgn) 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! ≤ l≤ 2^h
lg(n!)≤ lgl≤2^h => h≥ Ω(nlg) įrodyti naudojant stirlingo formulę: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1+\Theta\left(\frac{1}{n}\right)\right)$
Išvada. Rūšiavimas piramide ar suliejimo būdu – asimptotiškai optimalūs rūšiavimo algoritmai.
Įrodymas seka iš T8.1 ir viršutinių algoritmų Q(nlgn).

15.Tiesiniai rūšiavimo algoritmai. ???!Kodėl jie lenkia asiptotiškai optimalius palyginimo algoritmus.!!??
Rūšiavimas skaičiuojant(counting_short), pozicinis rūšiavimas(radix_short), kišeninis rūšiavimas(bucket_short)

16. Rūšiavimas skaičiuojant. Jo sudėtingumo įvertinimo įrodymas.
Apribojimai: Rūšiuojant elementai gali turėti sveikas reikšmes iš intervalo 0...k, čia k – reigiama sveika konstanta.
Jei rūšiuojama n elementų rušiavimas trunka, kai k=Q(n) tada T(n)=Θ(n)
Counting_sort(A,B,k)
1. for i <- 0 to k
2. do C[i] <- 0
3. for j <- 0 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 kiekis elementų skaičiai 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.:



Sudėtingumas, jei h=Q(n), T(n)= Θ(k+1)+ Θ(n)+ Θ(k)+ Θ(n)=Q(n+k)=Q(n)

17. 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).

18. Kišeninis rūšiavimas. Jo sudėtingumo įvertinimo įvertinimas.

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

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

19. Kokia „hešavimo“ paskirtis. Heš lentelės su tiesioginiu adresavimu (suformuluoti uždavinį ir pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti). Aprašyti šio būdo trukumus.

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.

20. Heš funkcijos ir kolizijų sprendimas grandinėlių metodu. Koks privalumas prieš heš lentelės su tiesioginiu adresavimu. Suformuluoti uždavinį, pateikti veiksmų algoritmus ir jų sudėtingumo įvertinimus pagrįsti.

Heš-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|.(galimų raktų skaičius). Kitas blogas dalykas, kad raktai skirtingiems elementams turi nesikartoti . taigi reikia rasti duomenų saugojimo struktūrą, kurios saugojimui reikėtų O~(|K|) ir pagrindiniai veiksmai su struktūra būtų atliekami vidutiniškai per O(1) laiko.(Pastaba: tiesiog. Adres. Atveju tai buvo blogiausiam atvejui).
Tam tikslui naudojama heš-f-ja h, kuri elementų raktus atvaizduoja i masyvo T[0 ... m-1] elementų indeksų aibę:
h:U→{0,1,...,m-1}. Sakysime, kad elem. Su raktu k hešuojamas į (poziciją h(k);dydis h(k)-vadinamas heš-reikšme raktui k. Šis būdas, turi silpną vietą h-f-ja gali atvaizduoti keletos elementų raktus į vieną heš reikšmę, t.y. h1!=h2, h(k1)=h(k2). Ši situacija vadinama kolizija, kurios sprendimui yra „gerų“ metodų. Šių kolizijų negalima išvenkti, todėl, kad |U|>m, todėl turi egzistuoti raktų aibės, kurie turi tą pačią hes-reikšmę.

Kolizijų sprendimas grandinėlių pagalba

Stand. Veiksmų realizacija:
1. Chained_hash_insert(T,x)
   Įterpti į sąrašo T[h(key[x])] priekį
2. Chained_hash_search(T,k)
   Elemento su raktu k paieška sąraše T(h(k))
3. Chained_hash_delete (T,x)
   Pašalinimas x iš sąrašo T[h(key[x])]

Blogiausias laikas įterpimui yra O(1). Pašalinimas ir paieška vyksta per laiką proporcingą atitinkamo sąrašo ilgiui. Tačiau jei turime abipusį sąrašą pašalinimas galimas per O(1).

21. 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 $\theta(1+\alpha)$.

22. Universalus hešavimas idėja ir pateikti vieną universalaus hešavimo funkcijų klasę. Suformuluoti teoremą apie raktą atitinkančios grandinėlės ilgį.

Idėja*:*

Kaikurie hešavimo būdai yra jautrūs hešuojamų raktų imčiai ir gali atsitikti taip, kad paieška truks $\theta(n)$ laiko, nes visi elementai sukris į vieną grandinėlę. Todėl yra tikslinga pasirinkti atsitiktinę heš funkciją – toks būdas vadinamas universaliu hešavimu. Pagrindinė idėja universalaus hešavimo yra atsitiktinai pasirinkti hešavimo funkciją iš atrinktos jų aibės (klasės), t.y. kad tie patys blogi duomenys neduos blogų rezultatų ir garantuos vidutiniškai gerą paieškos laiką.

Universalaus hešavimo funkcijų klasė:

Minėtas savybes turi tokia heš funkcijų aibė (klasė):

$b \epsilon \{0,1,\dots,p-1\} = Z_p$, $a \epsilon \{1,2,\dots,p-1\} = Z_p^*$,

čia p – pakankamai didelis pirminis skaičius.

$h_{a,b}(k) = ((ak+b) mod\ p) mod\ m$

$H_{p,m} = \{h_{a,b} |\ a \epsilon Z_p^*, b \epsilon Z_p\}$

$h_{a,b}$ funkcija pasižymi savybe, kad m yra nebūtinai pirminis skaičius, bet gali būti bet koks. Aibė $H_{p,m}$ yra universali heš funkcijų klasė.

Teorema:

Tegul h heš funkcija, išrinkta iš universalios funkcijų aibės, naudojama hešuoti n raktų į m dydžio heš lentelių, o kolizijų sprendimui naudojama grandinėlių metodas. Jei rakto k nėra lentelėje, tai matematinis vidurkis $E[n_{h(k)}]$ sąrašo ilgio, į kurį patenka hešuojamas raktas k, neviršija α. Jei raktas k yra lentelėje, tai sąrašo ilgio matematinis vidurkis $E[n_{h(k)}]$ kuriame randasi raktas k, neviršija 1+α.

23. 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.

24.Tiesinis, kvadratinis, dvigubas, idealus hešavimai.

Tiesinis hešavimo bandymas
Tegul h`:U->{0,1,..,m-1}, naudosime kaip pagalbinę haš f-ją.
Tiesinis hešavimo bandymas naudoja heš f-ją:
h(k,i) = (h`(k)+i)mod m, čia i {0,1,…,m-1)
Kvadratinis hešavimo bandymas
H(k,i) = (h`(k)+$c_1$i+$c_2i^2$)mod m, čia h` - pagalbinė heš f-ja, c_1,c_2 != 0 – pagalbinės konstantos.
Dirba geriau nei tiesinis, bet reikia parinkti m, c_1, c_2, kad būtų perrenkamos visos pozicijos.
Dvigubas hešavimas
h(k,i) = (h_1(k)+ih_2(k))mod m, čia h_1, h_2 – pagalbinės heš f-jos.
Pvz.:
h_1(k) = h mod m
h_2(k) = h_1 + (k mod m`), čia m` = ne daug mažesnis nei m.
Dvigubas hešavimas lenkia tiesinį ir kvad. heš bandymus sekų kiekiu (0(m^2), kai tuo tarpu nienti tik 0(m).
Idealus hešavimas
Idealiu hešavimu laikome metodą, krusi blogiausiu atveju atliks paiešką per O(1) kreipinių į atmintį.
Šis metodas primena hešavimą kai kolizijos pašalinamos grandinėlių pagalba. Šis metodas remiasi tuo, kad pasinaudojus universaliu hešqavimu grandinėlės dar kartą yra hešuojamos.
Pirminis hešavimas turi heš f-ją.
h(k) = (ak+b)mod p) mod m
Antrinis
h_j(k) = ((a_jk + b_j) mod p) mod m_j
Parinkus a, b, a_j, b_j, m, m_j, galima gauti lentelę, kuri neturi nei vienos kolizijos, tai garantuoja paiešką per fiksuotą lauką.
Kad garantuoti kolizijų nebuvima, antrame lygyje, reikalaujama, kad m_j heš lentelės S_j būtų kvadratas n_j raktų patekusių į poziciją j pirmojo hešavimo atveju.
Jei pasirinkote teisingos pirminio hešavimo f-ja atrinktas poreikis suvedamas į O(n).