Rīgas 64. Vidusskola

Datu kārtošanas algoritmu salīdzinošās novērtēšanas izveide daudzveidīgām situācijām.

Zinātniski pētnieciskais darbs datorzinātņu un informātikas sekcijā.

Darba autors: Žans Strojevs 12.DIT

Darba Vadītājs: Edvards Bukovskis

Rīga 2023

Anotācija

Kārtošanas algoritmu snieguma analīzes izstrādāšana. Žans Strojevs, darba vadītājs Rīgas 64. vidusskolas programmēšana II kursa skolotājs Edvards Bukovskis.

Pētnieciskajā darbā apskatīti kartošanas algoritmi un to strādāšanas princips, kā arī tā salīdzinājums programmēšanas gadījumos *Python* programmēšanas valodā.

Darba galvenais uzdevums ir izstrādāt strādājošo kārtošanas algoritmu salīdzināšanu, salīdzināt kārtošanas algoritmu veiktspēju dažādās sarežģītības situācijās, piedāvājot iespēju efektīvāk izmantot datorresursus un optimizēt jaunas programmatūras izstrādi.

Atslēgvārdi: kārtošanas algoritmi, Python, datorresursi.

Annotation

Developing sorting algorithm analysing performance. Žans Strojevs, supervisor and programming II course teacher of Riga 64th secondary school, Edvards Bukovskis.

The research paper examines sorting algorithms and their working principle, as well as its comparison in different programming cases in the Python programming language.

The main task is to develop a working sorting algorithm analysis, to compare the performance of sorting algorithms in different complexity situations, offering the possibility to use computer resources more efficiently and to optimise the development of new software.

Keywords: sorting algorithms, Python, computer resources

Saturs

Ievads

Kārtošanas algoritmi ir pamata rīki datorzinātnē un datu apstrādē. Tiem ir būtiska loma datu efektīvā organizēšanā, padarot to vieglāk meklējamu, atgūstamu un analizējamu informāciju. Kārtošanas algoritmi ir soli pa solim izstrādātas procedūras, kas paredzētas, lai sakārtotu sarakstu ar vienībām vai datu elementiem konkrētā secībā, parasti augošā vai dilstošā secībā, balstoties uz konkrētiem kritērijiem. Šis pamatkoncepts veido dažādu datorzinātnes lietojumu pamatu.

Darba tēma. Kārtošanas algoritmu salīdzinošās novērtēšanas izveide daudzveidīgās situācijās.

Darba mērķis. Izpētīt kārtošanas algoritmu pamatprincipus un veidus, veikt eksperimentālo analīzi, kas ir pielāgots un optimizēts noteiktām situācijām. Salīdzināt algoritmu sniegumu pēc salīdzinošas novērtēšanas jeb *benčmarkinga*. Veikt secinājums par to, kā kartošanas algoritmi veic kārtošanu ar citām datu struktūrām atkarībā no elementu daudzuma.

Darba uzdevumi.

1. Literatūras pārskats par kārtošanas algoritmiem un to izpēte.
2. Situāciju idenfikācija.
3. Izveidot programmatūru, kas ļauj novērot algoritmu darbību.
4. Datu apkopošana un analīze.

Pētāmā problēma. Kārtošanas algoritmu veiktspējas atšķirību izpēte dažādos datu sadalījumos.

Darbā izmantotās metodes. Literatūras analīze, lai iegūtu zināšanas un saprašanu par kārtošanas algoritmiem. Atšķirīgu situāciju veidošana, kas ļauj pētīt algoritmu pilno sniegumu. Programmas izstrādāšana uz *Python* programmēšanas valodā un tai nepieciešamas bibliotēkas. Rezultātu analīze salīdzinājot algoritmus un to veiktspēju situācijas. Publiski pieejami dati, kurus pielietot programmatūras situācijas analizēšanā un testēšanā. Grafiku veidošana, lai vieglāk un effektīvāk salīdzināt algoritmus un izvirzīt secinājumus.

1. Literatūras apraksts

1.1 Datu kārtošanas algoritmi

Kārtošanas algoritmi veido datu organizācijas pamatu datorzinātnē, izmantojot daudzveidīgus principus elementu sistemātiskai pārkārtošanai. Šo algoritmu pamatā ir salīdzināšanas pamatkoncepcija, novērtējot elementu pārus, lai noteiktu to relatīvo secību.[1.] Iterācijas princips ir iestrādāts daudzās kārtošanas metodēs, kur atkārtotie salīdzinājumi un pārkārtošana pakāpeniski sakārto elementus vēlamajā secībā. Turklāt kārtošanas algoritmi bieži izmanto tādas stratēģijas kā dalīšana un iekarošana, sadalot kārtošanas uzdevumu mazākās, pārvaldāmākās apakšproblēmās efektīvai apstrādei. Katrs algoritms ievēro īpašus principus - vai tā būtu burbuļkārtošanas blakus elementu salīdzināšanas vienkāršība, *quick-sort* rekursīvā sadalīšana vai šķiroto sarakstu apvienošana *merge-sort* - pielāgojot savu pieeju, lai panāktu optimālu sakārtošanu, pamatojoties uz pamatprincipiem, ko tās iemieso. Katrs no tiem algoritmiem ietver principu kopumu, no kuriem katrs piedāvā atšķirīgas stratēģijas, lai efektīvi sakārtotu datus dažādos kontekstos.

1.2 Datu kārtošanas algoritmu iedalījums

Kārtošanas algoritmus klasificē, pamatojoties uz dažādiem raksturlielumiem, un katrā kategorijā ir noteiktas īpašas iezīmes un uzvedība. Galvenie klasifikācijas kritēriji bieži ietver to darbības metodi, laika sarežģītību, lieluma sarežģītību un stabilitāti.[2.]

Pēc darbības veida kārtošanas algoritmus kopumā iedala uz salīdzināšanu balstītos vai nesalīdzināšanas algoritmos. Uz salīdzināšanu balstīti algoritmi, piemēram, burbuļkārtošana šķirošana vai ātrā kārtošana(*quick-sort*), izmanto pāru salīdzinājumus, lai pārkārtotu elementus.[2.] Algoritmi, kas nav balstīti uz salīdzināšanu, piemēram, skaitīšanas kārtošana vai *radix* kārtošana, šķiro elementus, pamatojoties uz to īpašajām īpatnībām bez tiešiem salīdzinājumiem.[2.]

Laika sarežģītības klasifikācija ietver algoritmu iedalīšanu dažādās efektivitātes klasēs, pamatojoties uz to veiktspēju, izmantojot dažādus ievades lielumus.

Lieluma sarežģītība(*Space complexity)* koncentrējas uz šķirošanas algoritmu atmiņas izmantošanu. Tā novērtē, cik efektīvi algoritms izmanto atmiņas resursus, šķirojot datus. Analīzējot izšķir divus galvenos algoritmu veidus. *In-place* algoritmi: Vietas šķirošanas algoritmi pārkārto elementus, izmantojot tikai konstantu papildu atmiņas apjomu. Šie algoritmi ir izdevīgi, ja atmiņas resursi ir ierobežoti vai atmiņas izmantošanas optimizācija ir prioritāte.[3.]

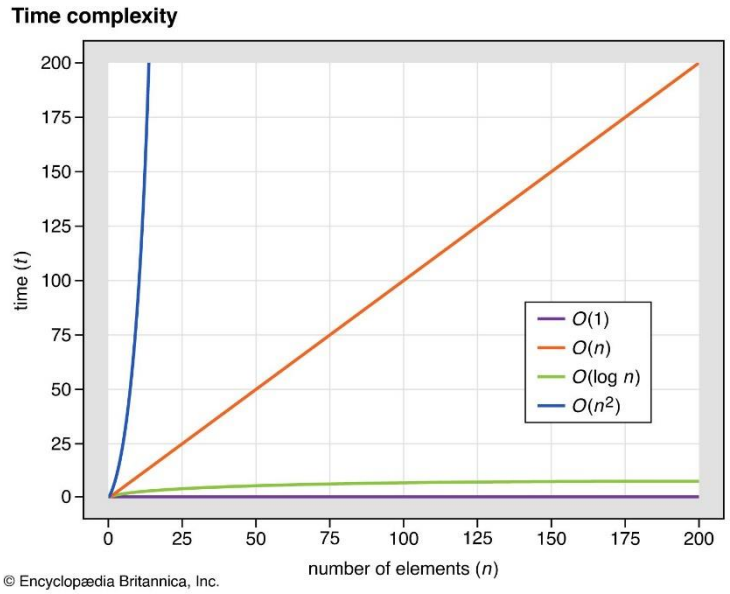
Algoritmi, kas nav *in-place* algoritmi: Algoritmi, kas nav *in-place* algoritmi, šķirošanai prasa papildu vietu atmiņā. Lai gan šie algoritmi var būt mazāk atmiņas ziņā efektīvi, tie var piedāvāt priekšrocības citos aspektos, piemēram, ātrumā.[3.]

Stabilitātes klasifikācijā novērtē, vai vienādi elementi pēc šķirošanas saglabā savu relatīvo secību. Šī daudzdimensiju klasifikācijas sistēma ļauj iegūt aptverošu izpratni par šķirošanas algoritmiem, palīdzot lietotājiem izvēlēties piemērotāko algoritmu konkrētiem datiem un veiktspējas prasībām.

1.3 Laika sarežģītība

Kārtošanas algoritmu novērtēšanā galvenais rādītājs ir laika sarežģītība(*Time complexity*), kas atspoguļo, kā algoritma izpildei nepieciešamais skaitļošanas laiks mainās atkarībā no ievades lieluma. [4.] Izpratne par algoritma laika sarežģītību ļauj programmētājiem izvēlēties savām vajadzībām piemērotāko algoritmu, jo ātrs algoritms, kas ir pietiekami labs, bieži vien ir labāks par lēnu algoritmu, kas darbojas labāk pēc citiem rādītājiem. [5.]

Lai novērtētu algoritma izpildei nepieciešamās skaitļošanas operācijas, tiek izmantoti matemātiskie modeļi. Algoritma faktiskais izpildes laiks var svārstīties atkarībā no tā konkrētā lietojuma, piemēram, meklējot 100 vai 1 miljonu ierakstu masīvā. Tāpēc datorzinātnieki definē laika sarežģītību attiecībā pret ievades lielumu, ar kuru algoritms strādā. Laika sarežģītība, ko attēlo kā T(n), ir saistīta ar mainīgo n, kas apzīmē ievades mērogu. Lai raksturotu T(n), tiek lietots *Big-O* apzīmējums, kas ilustrē veidu un ātrumu, kādā funkcija paplašinās, palielinoties tās elementu skaitam, piemēram O(n^2).



(1. attēls. Grafiski attēlota laika sarežģītība. Redzams kā atkarībā no n mainās funckijas laika pieaugums. Pieejams: https://www.britannica.com/science/time-complexity)

Konstanta laika sarežģītība, apzīmēta ar O(1), apzīmē algoritmu, kas konsekventi izpilda fiksētu operāciju skaitu neatkarīgi no elementu daudzuma. Piemēram, saraksta garuma noteikšanas algoritms var veikt vienu operāciju, lai iegūtu pēdējā elementa indeksu. O(1) nenozīmē vienīgo operāciju, bet garantē stabilu, nemainīgu operāciju skaitu.

Lineārā laika sarežģītība, apzīmēta ar O(n), atspoguļo algoritma izpildes ilgumu, kas lineāri mainās, pieaugot "n". Piemēram, 1000 ierakstu saraksta pārmeklēšanai vajadzētu aizņemt aptuveni desmitkārt vairāk laika nekā 100 ierakstu saraksta pārmeklēšanai, līdzīgi kā 100 ierakstu saraksta pārmeklēšana aizņem aptuveni desmitkārt vairāk laika nekā saraksta ar 10 ierakstiem pārmeklēšana.

O(log n) laika sarežģītība apzīmē algoritma laika pieaugumu, kas atbilst "n" logaritmam. Tādos gadījumos kā binārā meklēšana sakārtotā sarakstā, meklēšana ietver atkārtotu saraksta

sadalīšanu uz pusēm, līdz tiek atrasts vajadzīgais elements. Nepieciešamās dalīšanas mērogo ar "n" logaritmu 2. bāzē, nevis tieši proporcionāli "n". Algoritmi ar O(log n) uzrāda lēnākus pieauguma tempus salīdzinājumā ar lineāro laiku (O(n)), kā rezultātā laika sarežģītība ir mazāka.

Konkrēta algoritma sarežģītība var atšķirties atkarībā no problēmas specifikas, un to var analizēt attiecībā uz labākā, sliktākā un vidējā gadījuma grūtību. Piemēram, *quick-sort* parasti uzrāda vidējo laika sarežģītību O(n log n), bet tā sliktākā gadījuma scenārijs var sasniegt O(n2).

Sort type	Average time	Worst case	Storage space	Stability
Simple sort	O（n^2）	O（n^2）	O（1）	Stable
Quick sort	O（$n\log_2 n$）	O（n^2）	O（$n\log_2 n$）	Unstable
Select sort	O（n^2）	O（n^2）	O（1）	Unstable
Heap sort	O（$n\log_2 n$）	O（$n\log_2 n$）	O（1）	Unstable
Merge sort	O（$n\log_2 n$）	O（$n\log_2 n$）	O（n）	Stable
Radix sort	O（d（n+r））	O（d（n+r））	O（r）	Stable

(2. attēls. Tabula ar *Big-O* notāciju vidējā un sliktākajā gadījumā 6 kārtošanas algoritmiem, atmiņas daudzumu un stabilitāti. Adaptēts no *Analysis and Research of Sorting Algorithm in Data Structure Based on C Language, 2020*)

1.4 Lieluma sarežģītība

Lieluma sarežģītība datu kārtošanas algoritmos attiecas uz atmiņas apjomu, kas algoritmam nepieciešams, lai atrisinātu kārtošanas problēmu attiecībā uz tā ievades lielumu. Kārtošanas algoritmu lieluma sarežģītība atšķiras, un tas var ietekmēt to piemērotību dažādās situācijās, jo īpaši, ja runa ir par lielām datu kopām vai atmiņas ziņā ierobežotām vidēm. [6.]

Vietas jeb *in-place* kārtošanas algoritmi:

Šādi algoritmi kā *Quicksort* un *Heap Sort* tiek uzskatīti par *in-place* jeb vietas algoritmiem, jo tie šķiro elementus pašā masīvā, neprasot papildu atmiņu proporcionāli ievades lielumam. To telpas sarežģītība parasti ir O(1) vai konstanta telpa. [7.]

Nevietējas jeb *not in-place* kārtošanas algoritmi:

Algoritmi, piemēram, apvienošanas kārtošana(*Merge sort*) un ievietošanas kārtošana(*Insertion sort*), bieži prasa papildu atmiņu pagaidu glabāšanai to darbības laikā. *Merge Sort* darbības laikā alogritmam vajag papildus atmiņu, kas ir vienāda ar ieejas masīva lielumu, kā rezultātā sliktākajā gadījumā telpas sarežģītība ir O(n), kur 'n' ir šķirojamo elementu skaits. Ievietošanas kārtošana, lai gan tā neprasa daudz papildus atmiņas, dažos variantos atkarībā no implementācijas var būt no O(1) līdz O(n) vietas sarežģītība.

Ārējie kārtošanas algoritmi:

Šie algoritmi ir paredzēti liela datu apjoma kārtošanai, kas nevar pilnībā ietilpt galvenajā atmiņā. Tādi algoritmi kā ārējā apvienošanas šķirošana vai daudzfāžu apvienošanas kārtošana ietver datu lasīšanu un rakstīšanu no/uz ārējo atmiņu, piemēram, cietajiem diskiem. Tiem parasti ir vietas sarežģītība, kas ir atkarīga no ārējās atmiņas lieluma un datu pārsūtīšanai izmantotā bloka lieluma.

1.5 Adaptīvie un neadaptīvie algoritmi

Pielāgojamība ir svarīga kārtošanas algoritmu īpašība, un tā būtiski ietekmē to veiktspēju, jo īpaši, strādājot ar daļēji sakārtotiem datiem. Adaptīvs kārtošanas algoritms ir algoritms, kas var izmantot esošo kārtību ievades datos, lai uzlabotu tā efektivitāti. Tas kļūst ātrāks, ja to piemēro daļēji sakārtotiem vai gandrīz sakārtotiem datiem, salīdzinot ar pilnīgi nejaušiem datiem.[8.]

Neadaptīvie algoritmi negūst nekādu labumu no iepriekš pastāvošās datu kārtības. Tie saglabā nemainīgu veiktspējas līmeni neatkarīgi no tā, vai dati jau ir daļēji sakārtoti vai pilnīgi nesakārtoti.[8.]

Kārtošanas algoritma pielāgojamība var būt vērtīga iezīme reālos lietojumos. Piemēram, aplūkojot gadījumu, kad bieži ir nepieciešams šķirot sarakstu ar skaitļiem, kas lielākoties sakārtoti augošā secībā ar dažiem nejauši izvietotiem elementiem. Adaptīvais algoritms var izmantot jau sašķiroto daļu priekšrocības, tādējādi šādās situācijās nodrošinot ātrāku šķirošanu.

2. Datu kārtošanas salīdzinošās novērtēšanas izveide

Datu kārtošanas salīdzinošās novērtēšanas izveide ietver standarta izveidi, ar kuru var salīdzināt dažādu datu kārtošanas algoritmus. Šis process parasti ietver galveno rādītāju, piemēram, datu apstrādes ātruma, glabāšanas efektivitātes, uzticamības, mērogojamības un drošības, noteikšanu. Kad salīdzinošās novērtēšanas kritēriji ir noteikti, dažādus datu pārvaldības risinājumus vai stratēģijas var novērtēt, salīdzinot ar šiem kritērijiem, lai noteiktu to efektivitāti un lietderību. Salīdzinošā novērtēšana ļauj noteikt jomas, kurās nepieciešami uzlabojumi. Galu galā datu kārtošanas salīdzinošās novērtēšanas izveide ļauj optimizēt datu infrastruktūru, uzlabot datu kvalitāti un pieejamību un uzlabot vispārējo efektivitāti.

Autors izvēlējās galveno radītāju kā izpildes ātrums, jo tas ir galvenais rādītājs, kas ļauj notiekt algoritma efektivitāti un noderīgumu. Izpildes ātrums tiks salīdzināts starp izmantotiem kārtošanas algoritmiem, lai noteiktu efektīvāko algoritmu. Autors izmantos salīdzinošai analīzei *Quick-sort*, *Merge-sort* un *Heap-sort*.

Datu kārtošanas algoritmu funckijas izpildes ātrums ir atkarīgs no datora uz kā algoritms tiek ieslēgts. Autora datora specifikācija. Procesors: *AMD Ryzen 3 1300x 3.8GHz*. Operatīvā atmiņa: *Patriot Viper Elite Grey CL16 2666MHz*.

Salīdzinošās analīzes izveidei un datu salīdzināšanai autors izmantos *Python* ar bibliotēkām un koda redaktoru *Visual Studio Code*. Katrai salīdzinošās novērtēšanas situācijai tika izmantoti autora izvēlētie kārtošanas algoritmu implementācijas *Python* programmēšanas valodā.

2.1 Datu kārtošanas algoritma salīdzinošās novērtēšanas situāciju apraksts

1. situācija: kārtošanas algoritmu analīze, kur katram algoritmam ir jāsašķiro skaitļi no 1-10000.

Šajā situācijā autora mērķis ir analizēt dažādu šķirošanas algoritmu veiktspēju, ja uzdevums ir šķirot skaitļu kopu no 1 līdz 10000. Salīdzinot dažādu šķirošanas algoritmu izpildes laikus, autors cenšas noteikt, kurš algoritms šajā konkrētajā datu kopā darbojas

efektīvāk. Tāda tipa analīze var palīdzēt izvēlēties piemērotāko šķirošanas algoritmu skaitlisko datu kārtošanai dažādās lietojumprogrammās, tostarp datu apstrādei, datubāzu pārvaldībai un skaitļošanas algoritmiem.

2. situācija: kārtošanas algoritmu analīze, kurā algoritmi šķiro nejauši ģenerētas skaņas paraugu kopas.

Šajā situācijā autors plāno novērtēt dažādu kārtošanas algoritmu efektivitāti, piemērojot tos nejauši ģenerētām skaņas paraugu kopām. Kārtojot skaņas paraugus, kurus var attēlot kā skaitliskus datus, autors vēlas simulēt reālus scenārijus, kuros kārtošanas algoritmus izmanto audioapstrādē, ciparu signālu apstrādē vai mūzikas analīzes lietojumos. Šī analīze sniegs ieskatu par to, kā šķirošanas algoritmi darbojas, apstrādājot netradicionālus datu tipus veselajos skaitļos, un var palīdzēt izvēlēties piemērotus algoritmus uzdevumiem.

3.situācija: kārtošanas algoritmu analīze, kurā algoritmiem būs jākārto nejauši ģenerētas *hash values* jeb hašvērtības.

Šajā situācijā autors plāno novērtēt dažādu kārtošanas algoritmu veiktspēju, kārtojot nejauši ģenerētas *hash* vērtības. Hešvērtības parasti tiek attēlotas kā virknes vai skaitliskas vērtības, kas iegūtas no hešfunkcijām, un tās parasti izmanto datu struktūrās, piemēram, heš tabulās vai kriptogrāfijas lietojumos. Kārtojot heša vērtības, autors cenšas saprast, kā šķirošanas algoritmi apstrādā datus ar unikālām īpašībām, piemēram, neregulāru sadalījumu un augstu sadursmju skaitu. Šī analīze sniegs ieskatu par šķirošanas algoritmu efektivitāti un lietderību situācijās, kas saistīti ar hešētiem datiem.

2.2 Kārtošanas algoritmu darbības implementācija

Kārtošanas algoritmu darbības implementācija ietver katra izvēlētā kārtošanas algoritma realizācija salīdzinošā novertēšanā. *Quick-sort* algoritms kārto masīvu, izvēloties balsta elementu un sadalot masīvu divos apakšmasīvos. Pēc tam tas rekursīvi sakārto katru apakšmasīvu. Funkcija quicksort_wrapper uzsāk šo procesu, izmantojot nepieciešamos parametrus.

```python
def quicksort(arr, low, high):
    while low < high:
        pivot_index = partition(arr, low, high)
        quicksort(arr, low, pivot_index)
        low = pivot_index + 1


def partition(arr, low, high):
    pivot = arr[low]
    left = low
    for i in range(low + 1, high):
        if arr[i] < pivot:
            left += 1
            arr[i], arr[left] = arr[left], arr[i]
    arr[low], arr[left] = arr[left], arr[low]
    return left


def quicksort_wrapper(arr):
    quicksort(arr, 0, len(arr))
```

(3. attēls. *Quick-sort* darbības implementācija Python programmēšanas valodā.)

Funkcija merge_sort īsteno apvienošanas kārtošanas algoritmu *Merge-sort*. Funkcija rekursīvi sadala masīvu divās daļās, līdz katrā daļā ir tikai viens elements. Pēc tam otra funckija apvieno sašķiroto pusi, lai iegūtu galīgo sakārtoto masīvu. Apvienošanas funkciju izmanto, lai divus sakārtotus apakšmalu masīvus apvienotu vienā.

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    sorted_arr = merge(left_half, right_half)

    return sorted_arr
```

(4. attēls. *Merge-sort* masīvā sadlīšanas darbības implementācija.)

```python
def merge(left, right):
    result = []
    left_index, right_index = 0, 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1

    result.extend(left[left_index:])
    result.extend(right[right_index:])

    return result
```

(5. attēls. *Merge-sort* masīvā apvienošabas darbības implementācija.)

Heap-sort kārtošanas algoritms. Funkcija *Heapify* no dotā masīva izveido maksimālo kaudzi. Funkcija heap_sort ievieš kaudzes kārtošanas algoritmu, kas elementu kārtošanai izmanto kaudzes datu struktūru. Vispirms tā izveido maksimālo kaudzi no masīva un pēc tam atkārtoti izgūst maksimālo elementu no kaudzes un novieto to masīva galā.

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[l] > arr[largest]:
        largest = l

    if r < n and arr[r] > arr[largest]:
        largest = r


    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)
```

(6. attēls. Funkcijas *Heapify* darbības implementācija.)

```python
def heap_sort(arr):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

(6. attēls. Funkcijas *Heap-sort* darbības implementācija.)

2.3 Datu kārtošanas algoritmu salīdzinošās novērtēšanas izveide katrai situācijai

Pirmā situācijās kods ietver četras darbības: kārtošanas algoritmu darbības implementācija, datu ģenerēšanas algoritms, salīdzinošās analīzes izpilde un rezultāti.

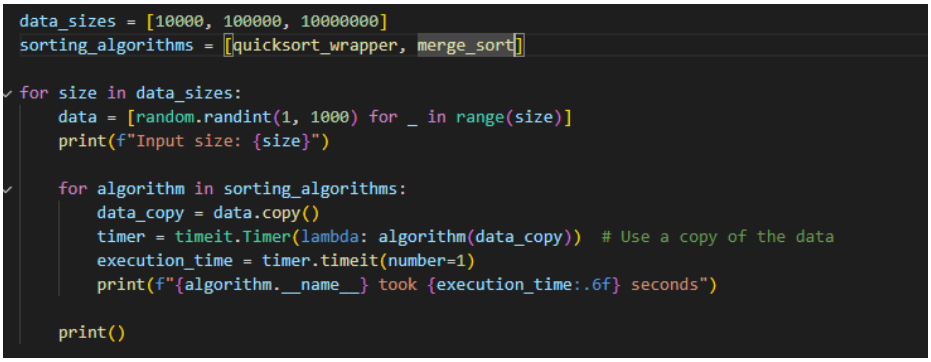
Sākotnējais kods iterē sarakstu ar nosaukumu data_sizes, kurā, domājams, ir veseli skaitļi, kas apzīmē datu masīvu izmērus. Katram izmēram, kas norādīts data_sizes, kods ģenerē nejaušu šāda izmēra datu masīvu. Kad tiek ģenerēts katrs masīvs, masīva lielums tiek izdrukāts konsoles ekrānā, lai tas būtu redzams. Ģenerētajos datu masīvos ir veselu skaitļu vērtības no 1 līdz 10000. Šie masīvi kalpo kā ieejas dati turpmākajiem šķirošanas algoritmiem. Pēc katra datu masīva ģenerēšanas kods turpina iterēt pa sarakstu ar nosaukumu sorting_algorithms, kurā, visticamāk, ir dažādi vērtējamie šķirošanas algoritmi. Katram sarakstā iekļautajam šķirošanas algoritmam:

Kods piemēro algoritmu pašreizējam datu kopumam.

Tiek mērīts kārtošanas algoritma izpildes laiks dotajam datu kopumam.

Algoritma nosaukums kopā ar atbilstošo izpildes laiku tiek izdrukāts konsolē.

Šo procesu atkārto katram lielumam, kas norādīts data_sizes, un katram kārtošanas algoritmam, kas uzskaitīts sorting_algorithms kopā. Tādējādi izvades rezultāti sniedz ieskatu par katra kārtošanas algoritma veiktspēju ar citiem ievades datu lielumiem.



```
data_sizes = [10000, 100000, 10000000]
sorting_algorithms = [quicksort_wrapper, merge_sort]

for size in data_sizes:
    data = [random.randint(1, 1000) for _ in range(size)]
    print(f"Input size: {size}")

    for algorithm in sorting_algorithms:
        data_copy = data.copy()
        timer = timeit.Timer(lambda: algorithm(data_copy))  # Use a copy of the data
        execution_time = timer.timeit(number=1)
        print(f"{algorithm.__name__} took {execution_time:.6f} seconds")

    print()
```

(7. attēls. Algoritms, kas veic datu ģenerāciju un kārtošanas algoritmu iterāciju un izpildes laika mērīšana.)

Pēc mērijumu beigšanas kods ģenerē diagrammu (skatīt pielikumā – 1.), kurā salīdzina trīs šķirošanas algoritmu (*Quick-sort, Merge-Sort, Heap-Sort*) izpildes laikus dažādiem ievades lielumiem. Labākai vizualizācijai katra algoritma izpildes laiks ir attēlots logaritmiskā skalā atkarībā no ieejas lieluma. Diagrammā ir iestrādātas katra datu punkta atzīmes, leģenda algoritmu identificēšanai un tīkla līnijas skaidrības nolūkam. Visbeidzot, grafiks tiek parādīts, izmantojot plt.show().

Otrā situācijās kods ietver četras darbības: kārtošanas algoritmu implementācija, skaņas ģenerēšanas algoritms, algoritmu izpildes laika mērīšana un rezultātu attēlošana grafiski.

Sākumā kods iniciē skaņas viļņu veidošanos. Tas tiek panākts, izmantojot NumPy funkciju random.randint, lai izveidotu nejaušības masīvu, kas satur skaņas paraugus. Šie paraugi kalpo par pamatu dzirdamiem skaņas viļņiem. Šajā procesā galvenā loma ir funkcijai generate_sound. Tā pārveido sakārtotos masīvus dzirdamos skaņas viļņos, katram paraugam piešķirot noteiktu frekvenci. Tas ļauj datus attēlot dzirdamā formā. Pēc tam gan sākotnējie, gan sakārtotie skaņas viļņi tiek saglabāti kā *WAV* faili, izmantojot bibliotēku "skaņas faili". Tas ļauj saglabāt skaņas datus turpmākai analīzei vai atskaņošanai. Sakārtoto skaņas viļņu vizuālo attēlojumu no oriģinālajiem viļņiem veic Matplotlib (skatīt pielikumā – 2.). Tas palīdz salīdzināt un analizēt dažādu kārtošanas algoritmu ietekmi uz skaņas datiem. Katrā vizualizācijas grafikā ir attēlots oriģinālais skaņas vilnis kopā ar kārtošanas algoritma ģenerētajiem sakārtotiem skaņas viļņiem, norādot to izpildes laiku. Lai precīzi novērtētu kārtošanas algoritmu veiktspēju, ir rūpīgi izmērīti izpildes laiki kārtošanai, izmantojot *Merge-sort*, *Quick-sort* un *Heap-sort*. Tas tiek panākts, izmantojot timeit.default_timer funkciju, kas nodrošina precīzus laika mērījumus. Lai saglabātu oriģinālo un sašķiroto skaņas viļņu ierakstu, tiek saglabāti *WAV* faili, nodrošinot skaņas datu attēlojumu dažādos kārtošanas procesa posmos.

```python
def generate_sound(sorted_array):
    duration = 0.05
    samples_per_tone = int(duration * 44100)

    full_sound = []
    for value in sorted_array:
        frequency = 100 + value
        samples = np.arange(samples_per_tone)

        waveform = np.sin(2 * np.pi * frequency * samples / 44100)
        full_sound.extend(waveform)

    return np.array(full_sound)
```

(8. attēls. Algoritms, kas veic skaņas ģenerāciju.)

Trešās situācijās kods ietver piecas darbības: datu ģenerēšana, failu ievades/izvades funkcijas, kārtošanas algoritmu implementācija, algoritmu izpildes laika mērīšana un rezultātu attēlošana grafiski.

Kods izmanto funkciju generate_random_hash, lai izveidotu vārdnīcas struktūru. Šī struktūra ir datu kopa, kas satur nejauši ģenerētus burtu un ciparu atslēgas, no kurām katra ir saistīta ar nejauši piešķirtu veselā skaitļa vērtību. Datu kopas lielumu, ko norāda parametrs num_keys, var mainīt, nodrošinot testēšanas mērogojamību. Pēc ģenerēšanas programmatūra izmanto failu apstrādes funkcijas save_data_to_file un load_data_from_file, lai pārvaldītu datu saglabāšanu. Pirms kārtošanas tas saglabā nešķirotos datus *JSON* failos, nodrošinot atkārtojamību un atvieglojot turpmāku analīzi. Tas ļauj efektīvi uzglabāt un piekļūt datu kopām. Salīdzinošās novērtēšanas process. Kods iterē, izmantojot iepriekš definētu datu izmēru sarakstu, data_sizes, kas ietver dažādus lielumus, piemēram, 1000, 10000, 100000, 1000000. Katram datu lielumam tas ģenerē atbilstošu datu kopu, izmantojot generate_random_hash. Nesortificētās datu kopas tiek saglabātas *JSON* failos, lai tās varētu izmantot nākotnē, tādējādi nodrošinot iespēju atkārtot eksperimentus. Katrs kārtošanas algoritms tiek izpildīts datu kopai, un izpildes laiks tiek mērīts, izmantojot timeit.Timer klasi. Tas nodrošina precīzus laika mērījumus, samazinot ārējo faktoru ietekmi. Katra kārtošanas algoritma izpildes laiki tiek reģistrēti un saglabāti analīzei, kas atvieglo ieskatu algoritmu efektivitātē. Pēc kārtošanas procesa pabeigšanas skripts izmanto Matplotlib bibliotēku, lai vizualizētu katra algoritma izpildes laiku, salīdzinot to ar ievades lielumu. Iegūtais grafiks sniedz algoritmu veiktspējas grafisku attēlojumu.

```python
def generate_random_hash(length=5, num_keys=100):
    hash = {}
    for _ in range(num_keys):
        key = ''.join(random.choices(string.ascii_letters, k=length))
        value = random.randint(1, 10000)
        hash_value = hashlib.sha1(key.encode()).hexdigest()  # Using SHA-1 hashing
        hash[hash_value] = value
    return hash
```

(9. attēls. Algoritms, kas veic hash ģenerēšanu vārdnīcas struktūrā.)

2.4 Datu kārtošanas algoritmu veikumu salīdzināšana

Lai salīdzinātu datu sniegumus katrai situācijai, autors iterēja katras situācijas salīdzinošo novērtēšanu trīs reizes. Snieguma rezultāti tiks apkopoti tabulā un datu lielums, ko katram algoritmam vajag sakārtot būs, 1000, 10000, 100000 un 1000000, jo mazākos datu izmēros algoritmi kārto ar minimālu atšķirību ātrumā.

Pirmās situācijas benčmarkings parādīja to, ka *Heap-sort*, sakārto datus lēnāk. *Quick-sort* mazākos datu apjomos pārspēj vidēji par 35%, tomēr ar datu izmēru 1000000 *Merge-sort* sakārto datus vidēji par 16% atrāks.

Kārtošanas algoritms	Vidējais rezultāts(1000)	Vidējais rezultāts(10000)	Vidējais rezultāts(100000)	Vidējais rezultāts(1000000)
Quick-sort	0,008809	0,086559333	1,111799	19,66236067
Merge-sort	0,012523	0,130704	1,531262333	16,56435
Heap-sort	0,027832667	0,339038	4,658350667	53,77177867

(10.attēls. Kārtošanas algoritmu vidējie rezultāti sekundēs pēc kārtošanas beigšanas.)

Otrās situācijas benčmarkings tiks veikts ar fiksētu datu daudzumu 10000. Pēc kārtošanas *Quick-sort* ir ātrākais starp izvēlētiem algoritmiem. Līdzīgi kā pirmajā situācijā *Heap-sort* ir lēnākais starp algoritmiem. *Quick-sort* ir videji par 25% ātrāks nekā *Merge-sort* un par 66% nekā *Heap-sort*.

Kārtošanas algoritms	Vidējie laiki(1000)
Quick-sort	0,144312667
Merge-sort	0,191457333
Heap-sort	0,430908

(11.attēls. Kārtošanas algoritmu vidējie laiki izpildot otrās situācijas kārtošanu sekundēs.)

Trešās situācijas benčmarkings līdzināsies pirmajam, kur datu izmērs būs 1000, 10000, 100000 un 1000000. *Quick-sort* ir ātrākais starp izvēlētiem kārtošanas algoritmiem katrā datu izmērā. *Heap-sort* ir lēnakais. *Quick-sort* ir par 33% ātrāks nekā *Merge-sort* kārtojot *hash* tipa datus.

Kārtošana	Vidējais rezultāts(1000)	Vidējais rezultāts(10000)	Vidējais rezultāts(10000	Vidējais rezultāts(1000000)
Quick-sor	0,00794603	0,0846987	1,028071933	12,94157947
Merge-so	0,012261963	0,125102727	1,5974085	17,34963336
Heap-sort	0,025194663	0,33246563	4,547014297	55,808493

(12. attēls Kārtošanas algoritmu vidējie laiki izpildot trešās situācijas kārtošanu sekundēs.)

Kopumā analizējot kārtošanas algoritmu sniegumus. *Quick-sort* bija ātrākais, tomēr kārtojot veselos skaitļus lielajā apjomā *Merge-sort* ātruma ziņā ir ātrāks nekā *Quick-sort*. Tomēr kārtojot *hash* datu tipu ar lielu apjomu *Quick-sort* pārspēj arī *Merge-sort*, jo kodā ģenerēto datu kopu veido SHA-1 heši, kas parasti ir vienmērīgi sadalīti. *Quick-sort* labi darbojās ar vienmērīgi sadalītiem datiem, pateicoties tās sadalīšanas procesam, kas sadala datus aptuveni vienāda lieluma nodalījumos. Tas ir pierādījums tam, ka atkarībā no datu tipa, ne vienmēr var balstīties uz laiku sarežģītību, izmantojot kārtošanas algoritmu.

Secinājumi

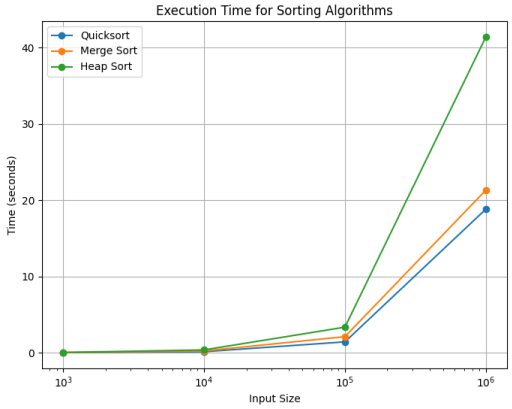
Pētījuma mērķi un iepriekš minētie uzdevumi autoram izdevās veiksmīgi izpildīt. Autors izpētīja kārtošanas algoritmu sadalījumu, un kā tos salīdzina izmantojot laika sarežģītību. Autors secināja to, ka ikdienā biežāk lietotie algoritmi ne vienmēr atbilst standartam un tie mēdz būt atkarīgi no datu tipiem. Darba autors veiksmīgi izprata, kā tiek salīdzināti kārtošanas algoritmi, kā strāda kārtošanas algoritmi un kā no kārtošanas algoritmiem iegūt sakārtoto datu struktūru. Darba autors veiksmīgi saprata, kādu galveno rādītāju ņemt vērā taisot benčmarkingu.

Praktiskajā daļā autors definēja trīs situācijās, kuros pētīs datu kārtošanas algoritmu veiktspēju. Implementēja katra izvēlētā kārtošanas algoritma darbību. Izveidoja katrai situācijai atbilstošo benčmarkingu un izveidoja iespēju nolasīt rezultātus caur tabulu. Iterējot katru benčmarkingu trīs reizes autors ieguva rezultātus, ka *Quick-sort* pārspēj gan *Merge-sort*, gan *Heap-sort. Quick-sort* kārtošanas izpildīšanas laiks vidēji ir par 30% ātrāks nekā citiem algoritmiem. No salīdzinošās novērtēšanas rezultātiem autors secināja to, ka algoritmu veiktspēju ir jāanalizē ņemot vēra dažādus datu tipus un datu lielumus.

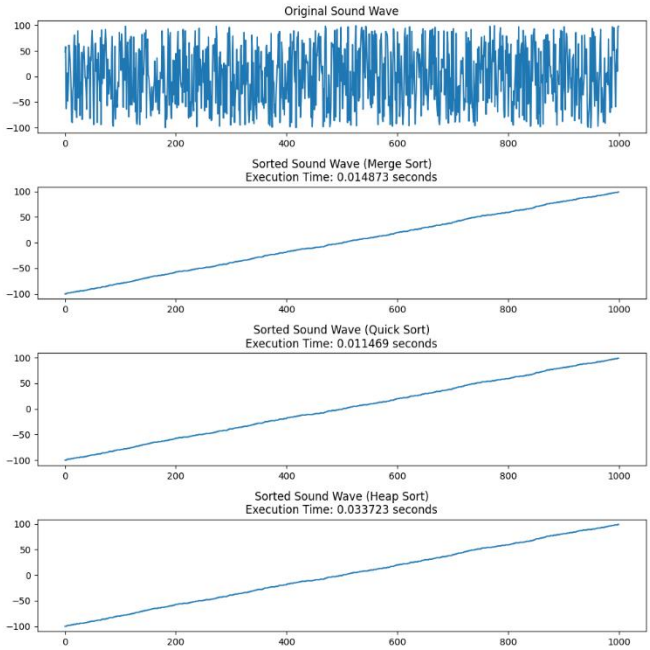
Izmantotie literatūras avoti

1. Britannica, Stephen Eldridge, (2023) sorting algorithm. Pieejams: https://www.britannica.com/technology/sorting-algorithm (Skatīts 20.10.2023)
2. Tomass H. Kormens, Čārlzs Ē. Leizersons, Ronalds L. Rivests, Klifords S. Steins *Introduction To Algorithms* Kembridža, Masačūsetsa: *The MIT Press*, 1990, 1132 lpp. (Skatīts 21.10.2023)
3. Princeton University, Sanjeev Arora and Boaz Barak, (2007) Computational Complexity: A Modern approach. Pieejams: https://theory.cs.princeton.edu/complexity/book.pdf (Skatīts 22.10.2023)
4. Southern Illinois University at Carbondale, Austin Mohr, Quantum Computing in Complexity Theory and Theory of Computation. Pieejams: http://www.austinmohr.com/Work_files/complexity.pdf (Skatīts 22.10.2023)
5. Britannica, Stephen Eldridge, (2023) time complexity. Pieejams: https://www.britannica.com/science/time-complexity (Skatīts 23.10.2023)
6. Veis Kuo, Mings J. Zuo *Optimal reliability modeling principles and applications* Ņūdžersija, Amerikas Savienotās Valstis: *John Wiley & Sons, Inc.,* (2002), 560 lpp. Pieejams: https://books.google.lv/books?id=vdZ4Bm-LnHMC&pg=PA62&redir_esc=y#v=onepage&q&f=false (Skatīts 23.10.2023)
7. Geeksforgeeks - In-Place Algorithm Pieejams: https://www.geeksforgeeks.org/in-place-algorithm/ (Skatīts 23.10.2023)
8. Geeksforgeeks - Adaptive and Non-adaptive Sorting Algorithms. Pieejams: https://www.geeksforgeeks.org/adaptive-and-non-adaptive-sorting-algorithms/ (Skatīts 23.10.2023)

Pielikumi



(1. Attēls. Trešās situācijas rezultātu grafiks.)



(2. Attēls. Otrās situācijas sākotnējas skaņas vilnis un rezultātu grafiks.)

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

ode\extensions\ms-python.python-2023.6.1\pythonFiles\lib\p
ntegers.py'
Input size: 1000
quicksort_wrapper darbība aizņēma 0.006260 sekundes
merge_sort darbība aizņēma 0.011090 sekundes
heap_sort darbība aizņēma 0.016428 sekundes

Input size: 10000
quicksort_wrapper darbība aizņēma 0.070159 sekundes
merge_sort darbība aizņēma 0.131646 sekundes
heap_sort darbība aizņēma 0.216670 sekundes

Input size: 100000
quicksort_wrapper darbība aizņēma 0.866171 sekundes
merge_sort darbība aizņēma 1.560629 sekundes
```

(3. Attēls. Pirmās situācijas precīzo kārtošanas rezultātu izvade terminālī.)