Vizualizacija algoritama sortiranja

Cindrić, Mateo

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:059727

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-04-01**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 363

VIZUALIZACIJA ALGORITAMA SORTIRANJA

Mateo Cindrić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 363

VIZUALIZACIJA ALGORITAMA SORTIRANJA

Mateo Cindrić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 363

Pristupnik:	**Mateo Cindrić (0036523282)**
Studij:	Računarstvo
Profil:	Programsko inženjerstvo i informacijski sustavi
Mentorica:	izv. prof. dr. sc. Josipa Pina Milišić

Zadatak: **Vizualizacija algoritama sortiranja**

Opis zadatka:

U suvremenom podučavanju osnova programiranja pojavljuje se potreba za interaktivnim i grafički prilagođenim prikazom procesa sortiranja. Cilj ovog diplomskog rada je kreirati vrijedan obrazovni resurs za studente, nastavnike i sve one koji su zainteresirani za razumijevanje algoritama sortiranja. Vizualizacijom algoritama korisnici mogu steći uvid u njihovu učinkovitost, usporediti različite algoritme, prepoznati njihove prednosti te uočiti nedostatke. Nakon pregleda osnovnih sekvencijalnih algoritama sortiranja, istražit će se mogućnosti korištenja Avalonia UI i React tehnologija za izradu aplikacije.

Rok za predaju rada: 28. lipnja 2024.

Sadržaj

1. Uvod 3

1.1. Povijest algoritama sortiranja 3

2. Analiza algoritama sortiranja 5

2.1. Svojstva algoritama 5

2.1.1. Veliko O notacija 7

2.2. Algoritmi sortiranja 10

2.2.1. Bubble Sort 11

2.2.2. Selection Sort 12

2.2.3. Insertion Sort 14

2.2.4. Merge Sort 16

2.2.5. Quick Sort 19

2.2.6. Heap Sort 22

2.2.7. Counting Sort 28

2.2.8. Radix Sort 32

2.2.9. Bucket Sort 33

3. Implementacija 36

3.1. Vizualizacija podataka 36

3.2. Struktura projekta 39

3.2.1. Komponente 41

3.2.2. Implementacija vizualizacije 45

4. Primjena algoritama sortiranja 53

4.1. Pretraživanje podataka 53

4.2. Baze podataka 54

4.3. Računalna grafika 54
4.4. Prikaz datotečnog sustava 54
4.5. E-trgovine i ostale web stranice 54
4.6. Analiza podataka 55

5. Zaključak 56

Literatura 58

Sažetak 59

Abstract 60

1. Uvod

Sortiranje možemo definirati kao proces razmještanja niza objekata kako bi bili poredani na određen način (ulazno ili silazno) u odnosu na neki njihov atribut [1]. U ovom radu obrađuju se algoritmi sortiranja bazirani na usporedbi te algoritmi sortiranja koji nisu bazirani na usporedbi elemenata. Također se obrađuju načini njihove vizualizacije i opisuje se razvoj aplikacije namjenjene za njihovu vizualizaciju te za njihovo bolje razumijevanje. Kao kratki uvod napravit ćemo pregled povijesti algoritama sortiranja kako bi stekli dojam o njihovom razvoju kroz niz godina. Nakon toga ćemo definirati općenite pojmove vezane za algoritme (vremenska i prostorna složenost) te dati pregled i opis nekih bitnijih algoritama sortiranja. Zatim ćemo napraviti kratak pregled vizualizacije podataka te izabrati način na koji će podaci biti prikazani u aplikaciji za vizualizaciju sortiranja. U istom poglavlju ćemo opisati implementaciju navedene aplikacije pomoću odabrane tehnologije gdje se kreće sa opisom strukture projekta te se nakon toga opisuju bitniji dijelovi koda i korišteni oblikovni obrasci. Za kraj ćemo opisati nekoliko situacija u kojima se primjenjuju opisani algoritmi sortiranja kako bi stekli dojam njihove važnosti u računalnim sustavima.

1.1. Povijest algoritama sortiranja

Kako je navedeno u [2], većina algoritama sortiranja izumljena je u razdoblju od 1954. do 1985. *Radix Sort* je prvi algoritam sortiranja koji je bio korišten komercijalno 1980. Koristio se u stroju za popisivanje čiji je izumitelj **Herman Hollirith**. Nakon toga slijede *Insertion Sort* i *Bubble Sort* u razdoblju od 1945. do 1956.

Do prekretnice dolazimo 1960ih kada je izmišljen *Quick Sort* algoritam. Njegov izumitelj je **Tony Hoare**. To je bio prvi algoritam koji je koristio koncept *podijeli pa vladaj*. Navedeni algoritam je čak i danas jako često korišten u implementacijama raznih funk-

cija za sortiranje u mnogim programskim jezicima.

Nekoliko godina nakon izuma *Quick Sort* algoritma, 1964. **J.W.J. Williams** predstavlja *Heap Sort* algoritam koji na inovativan način koristi strukturu podataka binarnog stabla kako bi sortirao niz podataka.

Razvoj algoritama sortiranja se nije zaustavio nakon izuma *Heap Sort* algoritma. U desetljećima koja su slijedila, istraživači i inženjeri su osmislili mnoge nove i sofisticirane algoritme koji su unaprijedili efikasnost i specifične primjene sortiranja. Algoritmi kao što su *Merge Sort*, *Shell Sort*, *Tim Sort*, i *Intro Sort* su dodali nove tehnike i optimizacije koje su omogućile brže i učinkovitije sortiranje u različitim kontekstima.

Međutim, unatoč ovom kontinuiranom napretku, algoritmi poput *Radix Sort*, *Insertion Sort*, *Bubble Sort*, *Quick Sort*, i *Heap Sort* ostaju ključni temelji u povijesti razvoja algoritama sortiranja. Njihova inovativna rješenja i koncepti oblikovali su put za sve buduće algoritme i još uvijek igraju važnu ulogu u računarstvu danas. Sada kada imamo osnovnu predodžbu o njihovoj povijesti, možemo krenuti sa njihovom analizom.

2. Analiza algoritama sortiranja

Prema [1], algoritam možemo definirati kao niz jednostavnih instrukcija koje je potrebno izvršiti kako bi se rješio neki problem. Algoritam uzima niz podataka kao svoj ulaz te izvršavanjem navedenog niza instrukcija obrađuje ulazne podatke te kreira nama potrebne izlazne podatke. Prema navedenome, algoritam je niz računalnih koraka koji transformiraju ulazne podatke u izlazne. Algoritmi uobičajeno ne ovise o specifičnim ulaznim podacima nego opisuju općenitu proceduru koja transformira ulazne podatke u izlazne za sve moguće slučajeve problema. Za neki problem možemo osmisliti više načina za njegovo rješavanje, odnosno više algoritama. Ti algoritmi će se vjerovatno razlikovati u određenim svojstvima. Uzmimo za primjer dva algoritma koji rješavaju isti problem te su provjereni i znamo da ćemo na kraju izvršavanja algoritama dobiti točan rezultat. Jedan od tih algoritama će možda biti dizajniran sa svhrom brzine te će riješiti problem brže od drugog algoritma, ali postoji mogućnost da će za to vrijeme rješavanja problema koristiti više memorije od drugog algoritma. Ta dva svojstva algoritama zovemo **vremenska složenost** (engl.*time complexity*) i **prostorna složenost** (engl.*space complexity*).

Za početak ćemo opisati navedena svojstva po kojima se algoritmi mogu razlikovati i po kojima je moguće uspoređivati algoritme kako bi mogli odabrati koji algoritam bi nam u određenoj situaciji bio najviše od koristi. Za kraj ćemo detaljno opisati danas najčešće korištene algoritme sortiranja te napraviti analizu njihovih svojstava vremenske i prostorne složenosti.

2.1. Svojstva algoritama

Kako bi razumjeli vremensku i prostornu složenost, za početak je potrebno razumjeti općenitu složenost i notaciju koja se koristi za označavanje složenosti nekog algoritma. Analiza algoritama kojom ćemo se mi baviti je analiza *a priori*. Promatrat ćemo trajanje

izvođenja programa u najlošijem slučaju kao vrijednost funkcije čiji je argument broj ulaznih podataka (veličina ulaznog skupa podataka). Na drugu stranu, postoji statistička analiza *a posteriori* kojom se ovdje nećemo baviti.

Svrha navedene *a priori* analize je predvidjeti prosječno vrijeme izvođenja programa kako bi znali da li kreirani algoritam zadovoljava naše potrebe. Ako ustanovimo da je algoritam spor ili neučinkovit, potrebno je ili prilagoditi se najlošijoj mogućnosti korištenja takvog algoritma ili pronaći tj. kreirati bolji algoritam.

Uvodimo dva pojma:

- vrijeme izvođenja T - vrijeme potrebno za potpuno izvođenje programa
- veličina ulaznog skupa podataka N - broj elemenata u ulaznom skupu podataka

Za primjer označavanja možemo uzeti naš konkretan primjer sortiranja podataka. Ulazni skup podataka može biti niz brojeva koje je potrebno sortirati. Broj elemenata u ulaznom skupu (polju) označavamo sa N, a vrijeme potrebno za sortiranje polja označavamo sa $T(N)$ - funkcija koja ovisi o broju ulaznih elemenata.

Pogledajmo primjer određivanja vrijednosti funkcije T(N). Uzmimo za primjer iteraciju kroz sve elemente ulaznog skupa podataka:

procedure ITERATE($array$)
    for `i in array` **do**
        `<do stuff>`
    end for
end procedure

Za obradu pojedinog elementa u nizu čiji dio koda se nalazi unutar petlje uzimamo konstantno vrijeme. Kako ćemo to ovdje napraviti N puta, za rezultat funkcije možemo postaviti jednadžbu na slijedeći način:

$$T(N) = c * N \tag{2.1}$$

gdje je c konstantno vrijeme potrebno za obradu pojedinog elementa unutar niza.

Vidimo kako dobivamo linearnu funkciju T(N) te zaključujemo kako vrijeme izvođenja algoritma linearno ovisi o broju elemenata u ulaznom skupu podataka. Ako bi na primjer prolazili kroz elemente matrice veličine NxM, u kodu bi koristili jednu petlju za prolaz kroz redove, te jednu unutarnju petlju za prolaz kroz stupce:

```
procedure ITERATE(matrix)
    for row in matrix do
        for column in matrix[row] do
            <do stuff>
        end for
    end for
end procedure
```

Oznake redova (npr. *row* i *column*) bi poprimale vrijednosti od 0 do N i od 0 do M te bi jednadžba za $T(N)$ bila

$$T(N,M) = c * N * c * M = c^2 * N * M \tag{2.2}$$

Navedeni primjeri pojednostavljeno objašnjavaju određivanje vremenske složenosti algoritma ovisno o broju ulaznih elemenata.

Na isti način definiramo funkciju $S(N)$ (engl. Space) koja označava prostornu složenost algoritma. Navedena funkcija nam govori koliko memorije (u proizvoljnoj jedinici kao npr. bit ili bajt) kreirani algoritam zauzima tokom izvođenja u ovisnosti o veličini ulaznog skupa podataka.

Sada kada smo vidjeli na koji način određujemo funkciju vremena izvođenja nekog skupa instrukcija, uvodimo matematičku definiciju veliko O notacije.

2.1.1. Veliko O notacija

U velikom broju slučajeva, vrijeme izvođenja T i prostornu složenost S nemoguće je točno odrediti, odnosno izračunati. S obzirom na to, uvodimo način procjene navedenih svojstava algoritma za veliki broj ulaznih podataka. Za početak, [3] uvodi slijedeću definiciju:

Definicija 1. *Neka su $f, g : \mathbb{N} \to \mathbb{R}^+$ dvije funkcije. Kažemo da je funkcija g* ***asimptotska gornja međa*** *za funkciju f ako postoji $c > 0$ i $n_0 \in \mathbb{N}$ tako da za svaki $n \geq n_0$ vrijedi $f(n) \leq cg(n)$. Oznaka: $f(n) = O(g(n))$.*

Definicija nam govori kako funkcija $f(n)$ za najveću vrijednost može poprimiti konstantni višekratnik funkcije $g(n)$ za dovoljno velike $n \in \mathbb{N}, n \geq n_0$. Zbog toga kažemo da je $g(n)$ gornja granica funkcije $f(n)$. Na taj način možemo koristiti oznaku O notacije kako bi odredili najgoru moguću vremensku te prostornu složenost određenog dijela koda. Tada funkciju $f(n)$ mijenjamo funkcijom $T(n)$ u slučaju vremenske složenosti te funkcijom $S(n)$ u slučaju prostorne složenosti. Uzmimo za primjer funkciju $T(n) = f(n) = n^5 - 3n^3 + n - 100$ te $g(n) = n^5$ te odredimo aproksimaciju vremenske složenosti.

Za sve $n \geq 1$ vrijedi slijedeće:

$$\begin{aligned} f(n) &= n^5 - 3n^3 + n - 100 \\ &\leq n^5 + 3n^3 + n + 100 \\ &\leq n^5 + n^5 + n^5 + n^5 = 4n^5 \end{aligned} \tag{2.3}$$

Iz nejednadžbe 2.3 dobivamo $f(n) \leq 4g(n)$. Iz navedenog, prema definiciji 1 vrijedi $f(n) = O(g(n)) = O(n^5)$. Na prikazani način pomoću O notacije možemo uspoređivati neke algoritme koji imaju istu namjenu kao na primjer, u našem slučaju, sortiranje podataka. [3] također uvodi slijedeće propozicije pomoću kojih možemo na lakši način u određenim situacijama odrediti aproksimaciju vremenske i prostorne složenosti:

Propozicija 1. *Ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$, tada je $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.*

Propozicija 2. *Ako vrijedi $f_1(n) = O(g_1(n))$ i $f_2(n) = O(g_2(n))$, tada je $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.*

Propozicija 3. *Ako je $p : \mathbb{N} \to \mathbb{R}^+$ polinom stupnja k, tada vrijedi $p(n) = O(n^k)$.*

Dokaze navedenih propozicija moguće je pronaći u [3] na stranici 21. Pogledajmo na primjeru kako možemo koristiti navedene propozicije za određivanje približne vremenske složenosti određenog dijela programa:

```
1   int example(int n) {
2     int sum = 0;
3
4     for(int i = 0; i < n; i++) {
5       int j = 1;
6       int inner_sum = 0;
7
8       while(j <= n) {
9         inner_sum += j;
10        j = j * 2;
11      }
12
13      sum = sum + inner_sum * i;
14      i = i + 1;
15    }
16
17    return sum;
18  }
```

U funkciji *example* vanjska petlja pomoću iteratora i prolazi vrijednostima od 0 do n. Na taj način će se vanjska petlja sama po sebi izvršiti u $T(n) = O(n)$ koraka. Nazovimo tu funkciju T_1. U unutarnjoj petlji iterator j poprima vrijednosti $0, 1, 2, 4, ...$ dok ne poprimi vrijednost koja je veća od n. Iz toga zaključujemo kako će se unutarnja petlja izvršiti u $T(n) = O(log_2 n)$ koraka. Za oznaku te funkcije uzimamo T_2. Kako funkcije ovise jedna o drugoj na način da funkciju T_2 ponavljamo T_1 puta, za ukupnu složenost ovakve funkcije dobivamo vrijednost $T_1 \cdot T_2$. Prema propoziciji 2 vrijedi $T_1 \cdot T_2 = O(n \cdot log_2 n)$. Na sličan način možemo koristiti druge propozicije kako bi odredili približnu vremensku ili prostornu složenost određenog dijela koda.

Postoje neki posebni slučajevi vremenskih te prostornih složenosti koje često susrećemo u raznim algoritmima:

Notacija	Primjer
$O(1)$	Funkcija f ne ovisi o ulaznom broju elemenata te ima konstantno vrijeme izvođenja. Za izvršavanje jednostavne operacije kao zbrajanje ili oduzimanje nad podatkom uzimamo konstantno vrijeme.
$O(n)$	Primjer je iteriranje kroz listu jer je potrebno posjetiti svaki element zasebno.
$O(n^2)$	Slično kao u primjeru 2.2
$O(2^n)$	Iteriranje svih mogućih podskupova određenog skupa.
$O(log_2 n)$	Binarno pretraživanje niza. U svakom koraku dio niza koji je potrebno pretražiti kratimo za pola.
$O(nlog_2 n)$	Rekurzivno sortiranje niza prema podskupovima (kasnije ćemo vidjeti algoritam *Merge Sort*).

Sada kada razumijemo osnovne pojmove prema kojima možemo uspoređivati algoritme sortiranja, napraviti ćemo pregled nekih najčešće korištenih algoritama sortiranja. Nakon što objasnimo način rada svakog algoritma, odrediti ćemo njegovu vremensku i prostornu složenost koristeći definicije i propozicije definirane u ovom odjeljku.

2.2. Algoritmi sortiranja

Kroz povijest, ljudi su oduvijek imali potrebu za organizacijom i sortiranjem stvari oko sebe. Neki primjeri takvih potreba su sortiranje knjiga u knjižnici, organizacija predmeta u trgovini ili dućanu, izrada rasporeda sjedenja po abecedi i slično.

Postepenim napretkom tehnologije, počela se javljati potreba za organizacijom i sortiranjem podataka u računalu te se na taj način uvode algoritmi sortiranja kao predodređeni postupci koji nam omogućuju organizaciju podataka prema određenim svojstvima. Takvi algoritmi su se s vremenom razvijali te se danas nalazimo u situaciji gdje postoji velik broj navedenih algoritama od kojih svaki sadrži svoje prednosti i mane. Povećanjem broja podataka te brzine i memorije računala jaljva se potreba za učinkovitim metodama sortiranja radi optimalne analize i manipulacije podataka.

U ovom poglavlju ćemo dati pregled nekih danas najčešće korištenih algoritama sortiranja te ćemo istaknuti njihova svojstva poput vremenske i prostorne složenosti kako

bi ih mogli usporediti sa ostalima te ćemo dati neke primjere situacija u kojima bi bilo primjereno koristiti navedene algoritme. Opisat ćemo neke algoritme bazirane na usporedbi te neke algoritme koji nisu bazirani na usporedbi

2.2.1. Bubble Sort

Bubble Sort algoritam radi na principu uzastopne zamjene elemenata do trenutka u kojem cijeli niz nije sortiran. Krećemo od početka niza te kontinuirano uspoređujemo susjedne elemente i zamjenjujemo ih u slučaju krivog redoslijeda. Uzmimo za primjer sortiranje niza brojeva od najmanjeg prema najvećem. U tom slučaju, najveći elementi će se pomicati prema kraju niza te će situacija biti obrnuta od algoritma *Selection Sort* u kojem se najmanji elementi pomiču na početak niza. Možemo naslutiti kako ovakav algoritam nije prikladan za velik niz podataka, budući da je potrebno mnogo puta prolaziti niz te uzastopno mijenjati elemente koji se nalaze na krivom mjestu.

Pseudo-kod *Bubble Sort* algoritma:

Algorithm 1 Bubble Sort

```
1:  function BUBBLESORT(A)                          ▷ Where A - array
2:      n = array.length
3:      for i = 0 to n − 1 do
4:          for j = 0 to n − i − 1 do
5:              if array[j] > array[j + 1] then
6:                  Swap(A, j, j + 1)
7:              end if
8:          end for
9:      end for
10: end function
```

U vanjskoj petlji prolazimo po danom nizu od početka do kraja. U unutarnjoj petlji pokušavamo smjestiti najveći pronađeni element na zadnje mjesto u trenutnom podnizu (iteriramo do vrijednosti $n - i - 1$). Na početku će taj podniz biti jednak cijelom nizu, u drugoj iteraciji cijelom nizu bez zadnjeg elementa itd.

Svojstva *Bubble Sort* algoritma:

Svojstvo	Vrijednost
Vremenska složenost	$O(n^2)$
Prostorna složenost	$O(1)$

Za vremensku složenost vrijedi slijedeće:

U vanjskoj petlji, iterator i poprima vrijednosti od 0 do n što znači da se kod unutar vanjske petlje uvijek izvodi n puta. U unutarnjoj petlji, iterator j poprima vrijednosti od 0 do $n - i - 1$. To znači da će se unutarnja petlja izvoditi na slijedeći način:

- Kada vrijedi $i = 0$, petlja se izvodi $n - 1$ puta.
- Kada vrijedi $i = 1$, petlja se izvodi $n - 2$ puta.
- ...
- Kada vrijedi $i = n - 1$, petlja se izvodi 0 puta.

Na taj način dobivamo slijedeću sumu:

$$\sum_{i=0}^{n-1}(n-1-i) = (n-1) + (n-2) + (n-3) + \cdots + 1 + 0 \tag{2.4}$$

Koristeći formulu za sumu prirodnih brojeva dobivamo slijedeće:

$$\sum_{i=0}^{n-1}(n-1-i) = \frac{n \cdot (n-1)}{2} \tag{2.5}$$

Prema dobivenoj formuli za vrijeme izvođenja $T(n)$ i definiciji 1 vrijedi slijedeće:

$$T(n) = O(n^2) \tag{2.6}$$

Možemo zaključiti kako *Bubble Sort* algoritam sam po sebi nije pretežito koristan kada radimo sa nizovima brojeva velike duljine. Algoritam kao takav je vrlo jednostavan te se najčeše koristi kao početni primjer algoritma sortiranja kako bi mogli lakše razumjeti potrebne instrukcije za kreiranje sortiranog niza.

2.2.2. Selection Sort

Selection Sort algoritam je vrlo jednostavan algoritam sortiranja koji radi na principu konstantne zamjene trenutnog elementa u iteraciji sa najmanjim elementom u ostatku

niza. Započinjemo od prvog elementa u nizu, pronalazimo najmanji element u ostatku niza te ga mijenjamo sa prvim elementom. Zatim počinjemo od drugog elementa te ponavljamo postupak i tako cijelo vrijeme do zadnjeg člana niza.

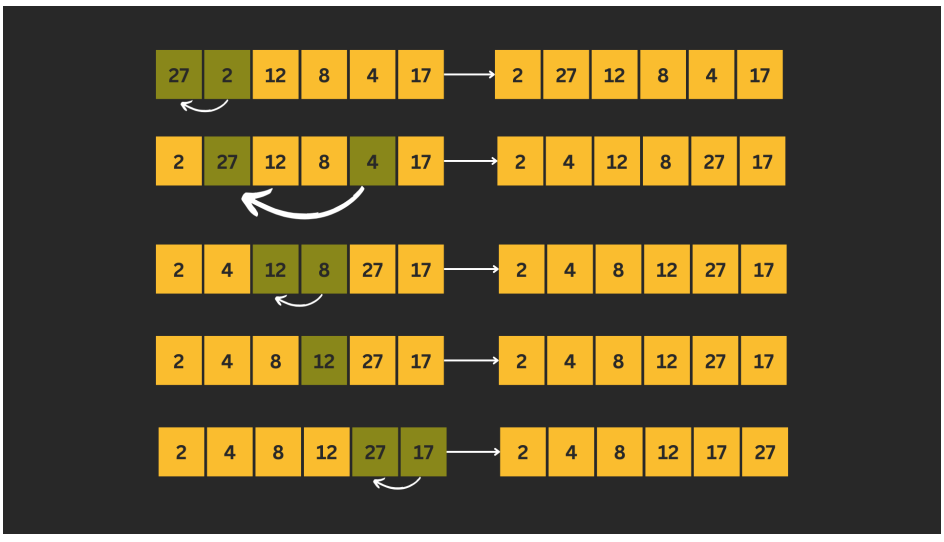
Pseudo-kod *Selection Sort* algoritma:

Algorithm 2 Selection Sort

```
1:  function SELECTIONSORT(A)                          ▷ Where A - array
2:      n = array.length
3:      for i = 0 to n − 1 do
4:          lowestIndex = i
5:          for j = i + 1 to n do
6:              if array[j] < array[lowestIndex] then
7:                  lowestIndex = j
8:              end if
9:          end for
10:         swap(A, i, lowestIndex)          ▷ Swaps elements at given positions
11:     end for
12: end function
```

U svakoj iteraciji započinjemo s pretpostavkom da je najmanji član trenutnog podniza početni, odnosno element na poziciji i. Spremamo ga kao $lowestIndex$ te prolaskom od trenutnog člana do zadnjeg pronalazimo najmanji član podniza usporedbom sa trenutnim. Nakon prolaska kroz ostatak niza, zamjenjujemo trenutni element na poziciji i sa zaključenim elementom na poziciji $lowestIndex$.

Ilustraciju koraka opisanog algoritma možemo vidjeti na sljedećoj slici:



Slika 2.1. *Selection Sort* algoritam

Svojstva *Selection Sort* algoritma:

Svojstvo	Vrijednost
Vremenska složenost	$O(n^2)$
Prostorna složenost	$O(1)$

Za vremensku složenost dobivamo vrijednost na sličan način kao i u *Bubble Sort* algoritmu:

Iterator i unutar vanjske petlje poprima vrijednosti od 0 do $n-1$ što znači da se kod vanjske petlje izvodi n puta. U unutarnjoj petlji, iterator j poprima vrijednosti od $i+1$ do n. Na taj način dobivamo slijedeću sumu:

$$\sum_{i=0}^{n-1}(n-(i+1)) = \sum_{i=0}^{n-1}(n-i-1) = (n-1)+(n-2)+(n-3)+\cdots+1+0 \tag{2.7}$$

Na isti način kao i kod *Bubble Sort* algoritma (2.5) vrijedi:

$$T(n) = O(n^2) \tag{2.8}$$

2.2.3. Insertion Sort

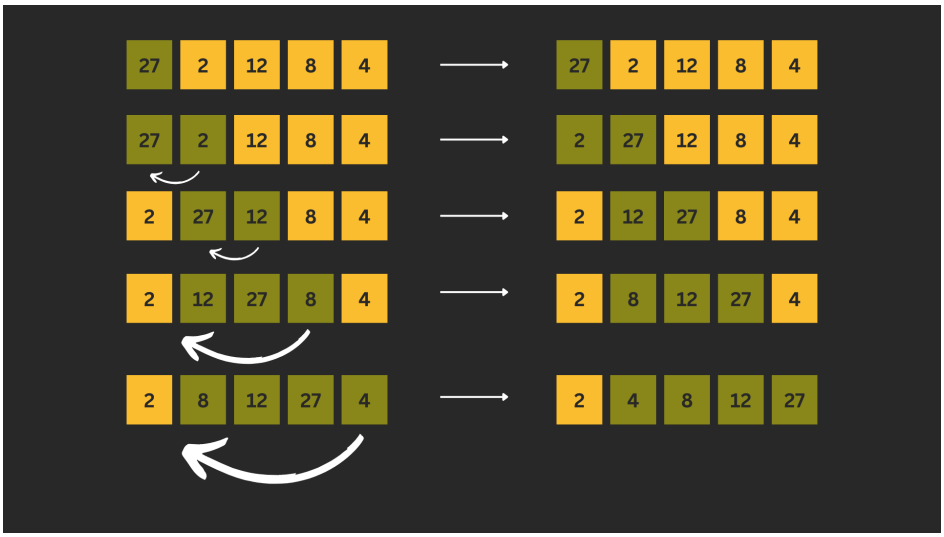
Insertion Sort algoritam gradi sortirani niz s desne strane niza prema lijevoj. Elementi s lijeve strane od trenutnog elementa u iteraciji su uvijek sortirani. Takav algoritam u određenim slučajevima radi bolje od prijašnjih algoritama. Na primjer, ako je niz gotovo sortiran, *Insertion Sort* radi puno brže zbog toga što je potrebno odraditi samo nekoliko iteracija.

Pseudo-kod *Insertion Sort* algoritma:

Algorithm 3 Insertion Sort

```
1:  function INSERTIONSORT(A)                          ▷ Where A - array
2:      n = array.length
3:      for i = 0 to n − 1 do
4:          for j = i + 1 to 0 do
5:              if array[j − 1] > array[j] then
6:                  Swap(A, j − 1, j)
7:              end if
8:          end for
9:      end for
10: end function
```

Ilustracija *Insertion Sort* algoritma:



Slika 2.2. Insertion Sort

Svojstva *Insertion Sort* algoritma:

Svojstvo	Vrijednost
Vremenska složenost	$O(n^2)$
Prostorna složenost	$O(1)$

Kao i u prijašnjim algoritmima, nije potrebno alocirati novu memoriju te stoga dobivamo $O(1)$ prostornu složenost. Vremenska složenost algoritma u najgorem slučaju kada su elementi poredani u obrnutom redoslijedu je $O(n^2)$, kao i kod prijašnja dva algoritma. U najboljem slučaju, svi elementi su sortirani te se samo jednom prolazi kroz elemente niza te dobivamo vremensku složenost $O(n)$.

Insertion Sort je jednostavan i intuitivan algoritam koji je učinkovit za male i za gotovo

sortirane nizove. Kako mu je vremenska složenost ista kao i prijašnjim algoritmima, na isti način zaključujemo da algoritam nije učinkovit na većim nizovima u kojima su elementi nepoznatog poretka.

2.2.4. Merge Sort

Merge Sort algoritam je rekurzivan algoritam koji radi na principu *podijeli pa vladaj*. To znači da algoritam dijeli početni problem na sve manje i manje probleme te rješenja takvih podproblema koristi za rješavanje početnog problema.

Algoritam je baziran na operaciji *merge* koja spaja dva sortirana niza brojeva u veći sortirani niz brojeva.

Pseudo-kod *Merge Sort* algoritma:

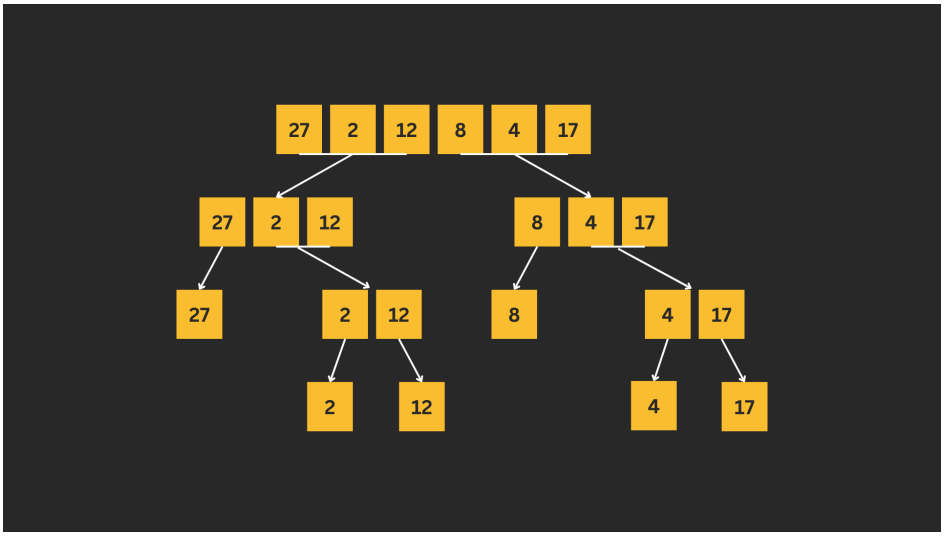
Algorithm 4 Merge Sort algoritam

```
1:  function MERGE(L, R)                          ▷ Where L - left array, R - right array
2:      result = empty list
3:      while L is not empty and R is not empty do
4:          if first(L) < first(R) then
5:              append first(L) to result
6:              L = L without first(L)
7:          else
8:              append first(R) to result
9:              R = R without first(R)
10:         end if
11:     end while
12:     insert leftover elements from L to result
13:     insert leftover elements from R to result
14: end function
15: function MERGESORT(A)                                  ▷ Where A - array
16:     n = array.length
17:     if n == 1 then                                     ▷ Preostao jedinični element
18:         return A
19:     end if
20:     middle = floor(n/2)                                ▷ Indeks srednjeg elementa niza A
21:     left = slice(A, 0, middle)                         ▷ Svi elementi lijevo od srednjeg indeksa
22:     right = slice(A, middle, n)                        ▷ Svi elementi desno od srednjeg indeksa
23:     leftSorted = MergeSort(left)
24:     rightSorted = MergeSort(right)
25:     return merge(leftSorted, rightSorted)
26: end function
```

Algoritam je rekurzivan i možemo ga podijeliti u dvije faze: dijeljenje i spajanje. U

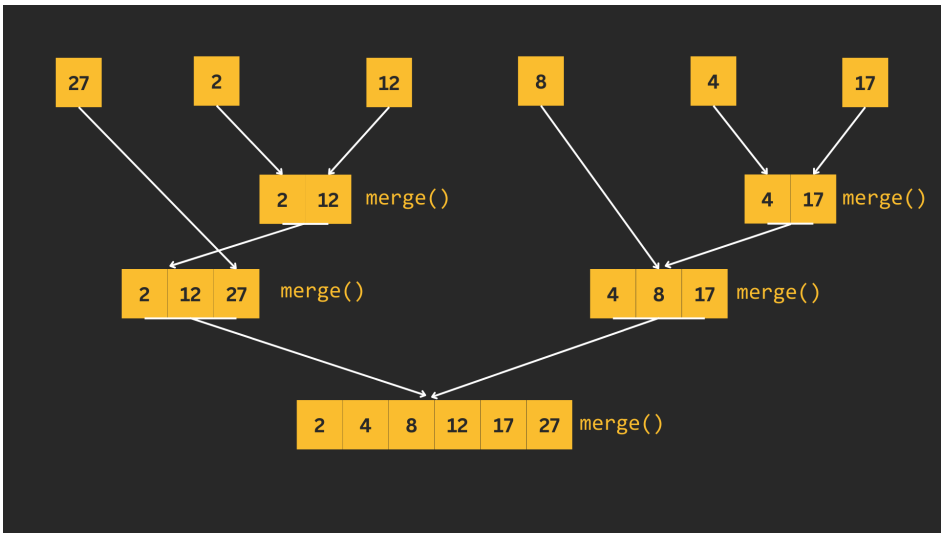
fazi dijeljenja, počinjemo sa početnim nizom A te ga dijelimo na dva dijela u odnosu na srednji element niza. Rekurzivno pozivamo funkciju $MergeSort()$ na lijevom i desnom podnizu te spremamo vrijednosti u varijable $leftSorted$ i $rightSorted$. Tako znamo da će te varijable sigurno sadržavati sortirane podnizove $left$ i $right$. Nakon toga operacijom $merge()$ spajamo dva podniza u jedan niz početne duljine.

S obzirom da se sve ove operacije obavljaju rekurzivno, teško je zamisliti redoslijed operacija i na koji način algoritam radi. Pogledajmo stoga slijedeće dvije ilustracije:



Slika 2.3. *Merge Sort* dijeljenje

Na slici 2.3. možemo vidjeti kako se početni niz od šest elemenata dijeli sve dok ne dođemo do podniza duljine jedan. Redoslijed operacija je takav zbog toga što se niz prvo dijeli na manje dijelove rekurzivnim pozivom funkcije $MergeSort()$ na linijama 23 i 24 te se operacija ponavlja sve dok ne podijelimo niz na podnizove duljine jedan. Tada prema liniji 18 vraćamo takav podniz kao povratnu vrijednost funkcije $MergeSort()$. Nakon toga slijedi faza spajanja koju možemo vidjeti na slijedećoj ilustraciji:



Slika 2.4. *Merge Sort* spajanje

Slika 2.4. ilustrira operaciju koju obavlja funkcija *merge*() koju pozivamo na liniji 25 Funkcija uzima podnizove L (engl.*left*) i R (engl.*right*) te ih spaja u jedan niz koji sadrži elemente u sortiranom redoslijedu te vraća taj niz kao svoju povratnu vrijednost.

Svojstva *Merge Sort* algoritma:

Svojstvo	Vrijednost
Vremenska složenost	$O(nlogn)$
Prostorna složenost	$O(n)$

Kao što vidimo, prednost ovog algoritma je njegova vremenska složenost $O(nlogn)$.

Analiza vremenske složenosti

Algoritam dijelimo u tri koraka:

- **Korak podjele**: $O(1)$ - pronalazimo sredinu trenutnog podskupa niza podataka
- **Korak rekurzivnog poziva**: svaki rekurzivan poziv u radi sa jednom polovicom podijeljenog niza podataka - nizovi veličina $\frac{n}{2}$, $\frac{n}{4}$ itd.
- **Korak spajanja podniza**: $O(n)$ za dva podniza veličina $\frac{n}{2}$.

Ovime dobivamo slijedeću rekurzivnu formulu za vremensku složenost algoritma:

$$T(n) = 2 \cdot T(\frac{n}{2}) + O(n) \tag{2.9}$$

Kako se svakom podjelom veličina niza smanjuje za pola, dubina rekurzivnog problema je $log_2 n$. Nakon toga se nalazimo u osnovnom slučaju gdje je veličina trenutnog niza jednaka jedan odnosno postoji samo jedan element u takvom podnizu. Kako se u svakom rekurzivnom pozivu također poziva i *merge()* funkcija čija je složenost $O(n)$, dobivamo slijedeću pojednostavljenu formulu:

$$T(n) = O(log_2(n)) \cdot O(n) \tag{2.10}$$

Prema propoziciji 2 vrijedi slijedeće:

$$O(log_2(n)) \cdot O(n) = O(log_2(n) \cdot n) \tag{2.11}$$

Tako dobivamo vremensku složenost algoritma:

$$T(n) = O(n \cdot log_2(n)) \tag{2.12}$$

Analiza prostorne složenosti

Algoritam ima prostornu složenost $O(n)$ zbog funkcije $merge()$. U navedenoj funkciji potrebno je svaki put alocirati niz duljine $L.length + R.length$. Kako se radi o podnizovima početnog niza, na kraju ćemo alocirati ukupno n novih mjesta za elemente.

2.2.5. Quick Sort

Quick Sort algoritam kao i *Merge Sort* algoritam radi na principu *podijeli pa vladaj*. Prednost ovog algoritma nad drugim sličnim algoritmima je ta što algoritam ne zauzima novu memoriju tijekom izvršavanja (ako izuzmemo korištenje stoga sustava) nego mijenja početni predani niz podataka. Za takav algoritam kažemo da radi *na mjestu* (engl.*in place*).

Algoritam radi na način da particionira početni niz podataka u dva podniza podataka, ovisno o jednom odabranom elementu koji nazivamo *pivot* element. Takve podnizove podataka kao i u *Merge Sort* algoritmu ponovno rekurzivno particioniramo i sortiramo oko novog pivot elementa. Kako je navedeno u [1] na stranici 288, jedina mana ovakvog pristupa je potreba za pažljivim odabirom pivot elementa.

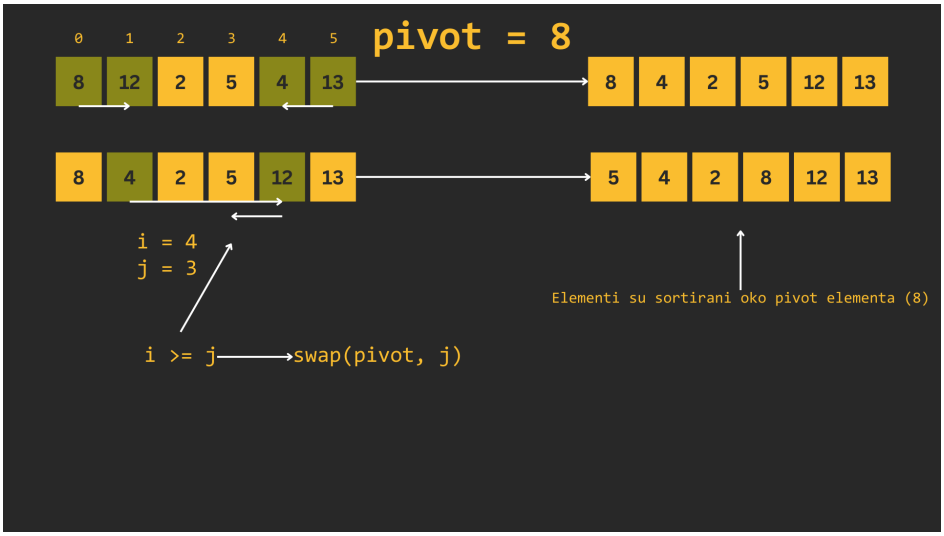
Pogledajmo za početak pseudo kod algoritma:

Algorithm 5 Quick Sort algoritam

```
1:  function PARTITION(A, lo, hi)
2:      i = lo + 1
3:      j = hi
4:      pivot = A[lo]
5:      while true do
6:          while A[i] < pivot do          ▷ look for next element greater than pivot
7:              i = i + 1
8:              if i = hi then
9:                  break
10:             end if
11:         end while
12:         while A[j] > pivot do          ▷ look for next element lower than pivot
13:             j = j − 1
14:             if j = lo then
15:                 break
16:             end if
17:         end while
18:         if i ≥ j then
19:             break
20:         end if
21:         swap(A, i, j)
22:     end while
23:     swap(A, lo, j)
24:     return j
25: end function
26: function QUICKSORT(A, lo, hi)
27:     if hi ≤ lo then
28:         return
29:     end if
30:     p = partition(A, lo, hi)
31:     QuickSort(A, lo, p − 1)
32:     QuickSort(A, p + 1, hi)
33: end function
```

Vidimo kako algoritam radi uzastopnim i rekurzivnim particioniranjem početnog niza. Za razliku od *Merge Sort* algoritma možemo uočiti kako ne dolazimo prvo do podnizova duljine jednog elementa kako bi napravili neku zamjenu ili spajanje nizova. U ovom slučaju vidimo kako prvo pozivamo funkciju $partition()$ prije rekurzivnog poziva funkcije $QuickSort()$. Funkcija $partition()$ particionira početni niz oko prvog elementa u nizu. Takav postupak se ponavlja cijelo vrijeme, sve dok ne dođemo do najmanjih podnizova veličine jedan.

Kako se funkcija *partition*() može na prvu činiti nejasnom, pogledajmo na primjeru jednostavnog niza na koji način funkcija particionira elemente oko odabranog *pivot* elementa:



Slika 2.5. Quick Sort particioniranje

Na slici 2.5. uzimamo prvi element 8 kao pivot element. Pomoću indeksa i krećemo se prema desnoj strani niza, dok se pomoću indeksa j krećemo od zadnjeg indeksa prema lijevoj strani niza. Koristeći indeks i, tražimo prvi element koji je veći od pivot elementa. Na isti način, pomoću indeksa j tražimo prvi element koji je manji od pivot elementa koji je u ovom slučaju 8. U prvoj iteraciji, pronašli smo broj 12 pomoću indeksa i (linija broj 6 u pseudo kodu) te broj 4 pomoću indeksa j (linija broj 12 u pseudo kodu).

Cilj ovakvog postupka je pronaći sve elemente koji su veći od pivot elementa i prebaciti ih na desnu stranu niza (isto vrijedi i za manje elemente na lijevu stranu niza). Postupak se ponavlja sve dok postoje elementi koji su manji/veći od pivot elementa.

U ovom slučaju, već u drugoj iteraciji dolazimo do situacije gdje ne postoje elementi koje treba prebaciti. Taj slučaj nalazimo na liniji broj 18 Indeksi i i j se u tom slučaju *mimoilaze* te tako sigurno znamo da su svi elementi razvrstani na dobru stranu u odnosu na pivot element.

Jedino što preostaje jest prebaciti pivot element na odgovarajuće mjesto u nizu. To radimo na liniji broj 21 u pseudo kodu. Navedeno možemo vidjeti na slici 2.5. u drugoj iteraciji. Element na indeksu $j = 3$ (5) zamjenjujemo sa pivot elementom te niz postaje

particioniran. Na lijevoj strani pivot elementa nalazimo sve elemente koji su manji od njega te na desnoj strani nalazimo sve elemente koji su veći od njega.

Takav postupak konstantno ponavljamo pomoću rekurzivnog poziva *QuickSort*() funkcije, predavajući kao parametre niz te granične vrijednosti za lijevu i za desnu stranu početnog niza. Možemo zaključiti kako će algoritam uspješno sortirati početni niz ponavljanjem navedenog postupka particioniranja.

Svojstva *Quick Sort* algoritma:

Svojstvo	Vrijednost
Vremenska složenost	$O(nlog_2n)$
Prostorna složenost	$O(1)$

Vremensku složenost dobivamo na identičan način kao i u analizi prostorne složenosti *Merge Sort* algoritma 2.2.4. Što se tiče prostorne složenosti, ako izuzmemo zauzimanje memorije stoga sustava rekurzivnim pozivima funkcije *QuickSort*(), dobivamo prostornu složenost $O(1)$ kako ne zauzimamo novu memoriju nego manipuliramo nizom podataka na mjestu. Ako uzmemo u obzir zauzimanje memorije stoga sustava, dobivamo prostornu složenost $O(n)$, kao i u slučaju *Merge Sort* algoritma.

2.2.6. Heap Sort

Kako bi razumjeli način na koji radi *Heap Sort* algoritam, prvo trebamo znati na koji način radi struktura podataka *prioritetni red* (engl.*priority queue*) te razumjeti njenu implementaciju pomoću strukture podataka *gomila* (engl.*heap*).

Prioritetni Red

Kako je navedeno u [1], prioritetni red je tip podatka koji podržava dvije operacije: *brisanje prioritetnog elementa* i *unos elementa*. Navedenu strukturu podataka koristimo u situacijama gdje je potrebno obaviti neku operaciju nad elementom koji trenutno ima najveći prioritet te ga nakon toga izbaciti iz reda. Unos elemenata u takav red također mora biti dopušten. Navedeni postupak se zatim ponavlja sve dok postoje elementi u redu. Za implementaciju ovakvog tipa podatka možemo koristiti osnovne strukture podataka poput niza ili povezane liste. Niz u tom slučaju može biti uređen na način da

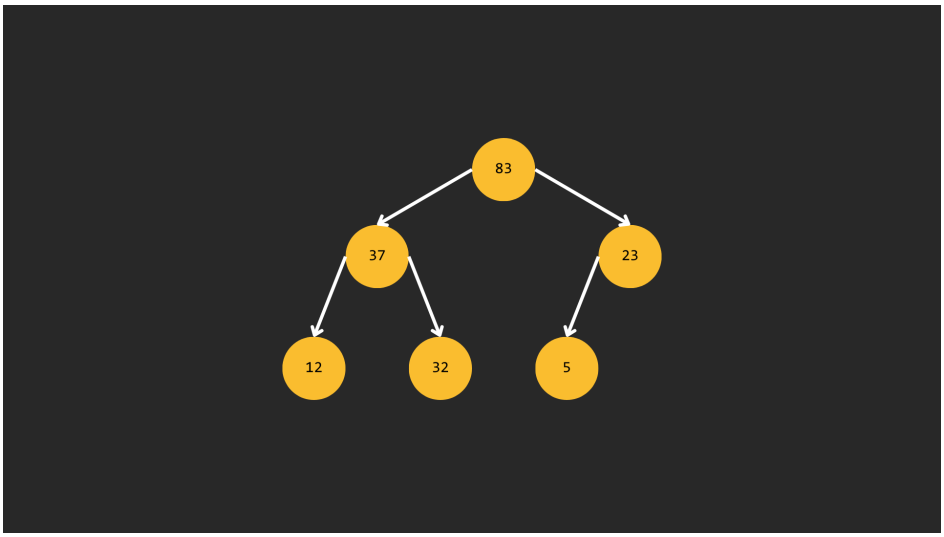
su svi elementi u njemu sortirani prema prioritetu ili neuređen. Kada bi implementirali ovakvu strukturu podataka pomoću navedenih opcija, dobili bi slijedeće vremenske složenosti za potrebne operacije:

Struktura podataka	*unos*	*brisanje prioritetnog elementa*
Uređen niz	$O(n)$	$O(1)$
Neuređen niz	$O(1)$	$O(n)$
Povezana lista	$O(1)$	$O(n)$

Ako bi koristili uređen niz, potrebno je pretražiti niz podataka te u najgorem slučaju proći kroz cijeli niz i staviti novi element na zadnje mjesto te zbog toga dobivamo složenost operacije *unos* $O(n)$. Operacija *briasnje prioritetnog elementa* sastoji se od smanjivanja veličine niza za jedan te na taj način dobivamo konstantnu složenost. Pri korištenju neuređenog niza dobivamo obrnutu situaciju: element jednostavno unosimo na kraj niza ali za brisanje prioritetnog elementa potrebno je pronaći element te ga izbrisati i reorganizirati niz kako ne bi postojalo prazno mjesto. Što se tiče povezane liste, element se jednostavno unosi dodavanjem novog čvora na kraj liste ali je također za brisanje elementa potrebno pretražiti listu te izbrisati čvor koji sadrži prioritetni element. Vidimo kako ove strukture podataka pružaju jednostavan način za implementaciju prioritetnog reda, ali u najgorem slučaju nam pružaju linearno vrijeme za bilo koju od dvije potrebne operacije. Međutim, struktura podataka *gomila* podržava obje operacije u $O(log_2 n)$ vremenu te je prikladna struktura podataka za implementaciju prioritetnog reda.

Gomila

Gomila je **potpuno** binarno stablo (stablo u kojem je svaka razina osim zadnje potpuno popunjena) u kojem se čvorovi mogu uspoređivati određenom relacijom i u kojem svaki čvor zadovoljava navedenu relaciju s obzirom na svoje dječje čvorove. Gomilu s relacijom **veći od** zovemo *max heap* dok gomilu s relacijom **manji od** zovemo *min heap*.

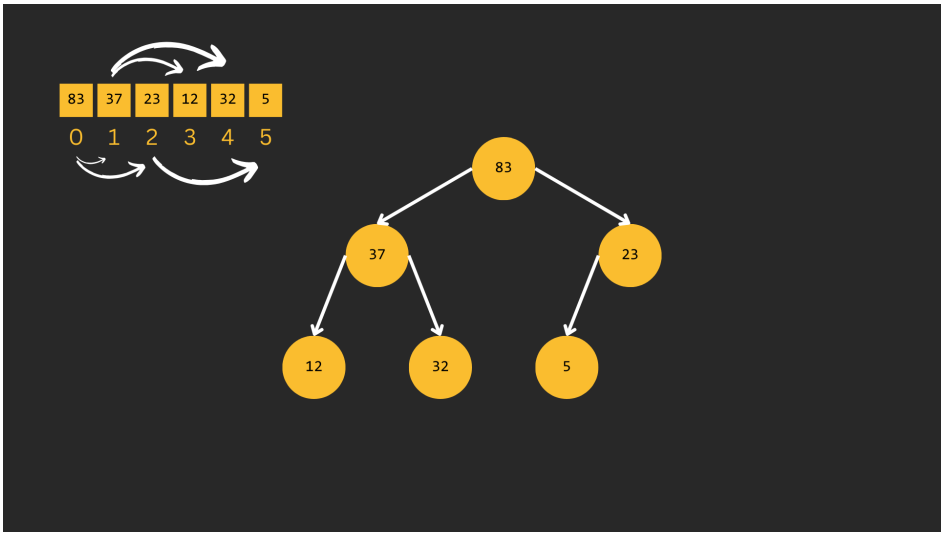


Slika 2.6. *max heap*

Binarno stablo moguće je implementirati pomoću običnog niza podataka. Korijenski čvor nalazi se na poziciji 0 te vrijede slijedeće formule za dječje čvorove L (engl.*left*) i R (engl.*right*) bilo kojeg čvora stabla:

- $L = 2 \cdot i + 1$
- $R = 2 \cdot i + 2$

i je u ovom slučaju indeks trenutnog čvora na kojem se nalazimo. Na primjer, za korijenski čvor vrijedi $L = 2 \cdot 0 + 1 = 1$ i $R = 2 \cdot 0 + 2 = 2$. Pogledajmo reprezentaciju pomoću niza podataka vezanu za ilustraciju 2.6.:



Slika 2.7. *max heap* pomoću niza podataka

Također, ako bismo se željeli vratiti na roditelja pojedinog čvora, vrijedi slijedeća for-

mula: $P = \left\lfloor \frac{i}{2} \right\rfloor$ gdje je i indeks čvora na kojem smo trenutno pozicionirani.

Kako bi prioritetni red implementirali pomoću gomile, moramo biti u mogućnosti podržati operacije *unos* i *brisanje prioritetnog elementa*.

Unos elementa radi tako da ubacimo element na kraj niza te restrukturiramo cijeli niz kako bi se zadovoljilo pravilo gomile. Algoritmi potrebni za restrukturiranje objašnjeni su u [1] na stranici 316. Kod implementacije *heap sort* algoritma opisat ćemo jedan od tih algoritama.

Brisanje prioritetnog elementa radi na slijedeći način: najveći element po definiciji gomile nalazi se u korijenskom čvoru. Njega mijenjamo sa zadnjim elementom niza gomile, smanjujemo veličinu niza za jedan te ponovno organiziramo niz kako bi vrijedilo pravilo gomile.

Obje operacije zahtjevaju $O(log_2 n)$ vremena te stoga čine optimalnu implementaciju strukture podataka prioritetni red.

Algoritam Heap Sort

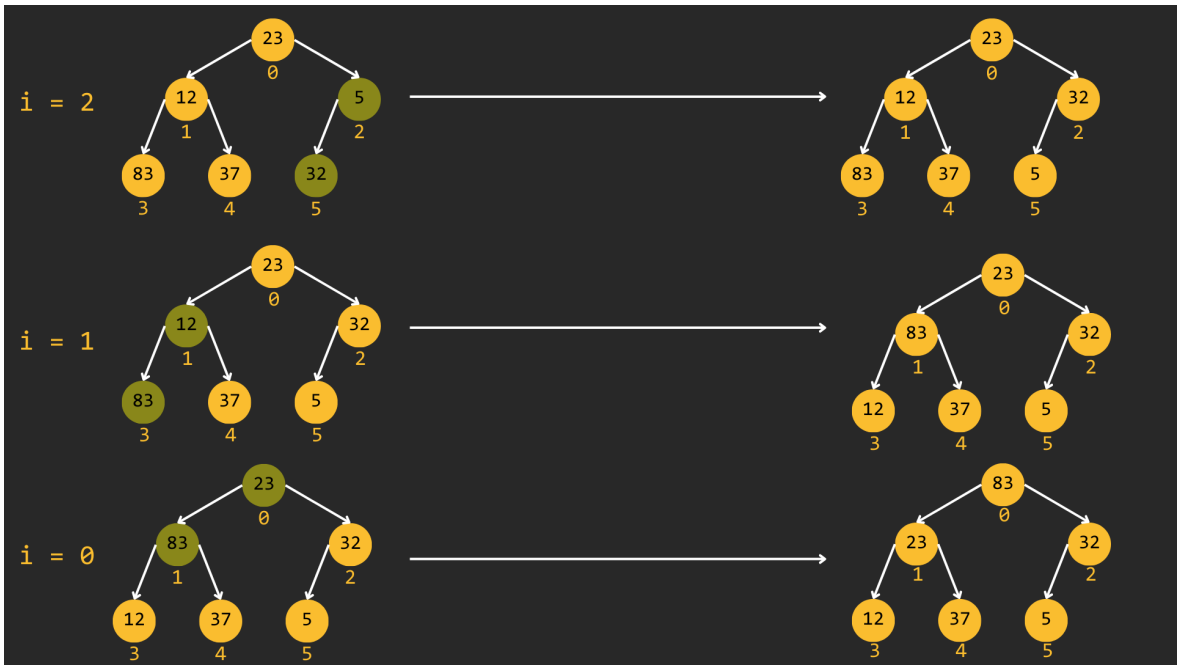
Prema opisanim postupcima operacija *unos elementa* i *brisanje prioritetnog elementa* prioritetnog reda možemo već donekle zamisliti na koji način ova struktura podataka omogućuje sortiranje niza brojeva. Algoritam se sastoji od dvije faze: *konstrukcija gomile* i *sortiranje prema dolje*. Korištenjem *max heap* strukture podataka, početni niz elemenata možemo restrukturirati u gomilu (faza **konstrukcije gomile**) te kontinuirano primjenjivati operaciju *brisanje prioritetnog elementa* kako bi najveći element stavljali na kraj niza i u konačnici dobili potpuno sortiran niz podataka (faza **sortiranja prema dolje**).

Pogledajmo za početak pseudo kod algoritma:

Algorithm 6 Heap Sort algoritam

```
1: function HEAPIFY(A, i, N)
2:     l = 2 · i + 1
3:     r = 2 · i + 2
4:     largest = i
5:     if l < N and A[l] > A[largest] then
6:         largest = l
7:     end if
8:     if r < N and A[r] > A[largest] then
9:         largest = r
10:    end if
11:    if largest ≠ i then
12:        swap(A, i, largest)
13:        heapify(A, largest, N)
14:    end if
15: end function
16: function HEAPSORT(A)
17:    N = A.length
18:    for i = ⌊i/2⌋ − 1; i ≥ 0; i = i − 1 do
19:        heapify(A, i, N)
20:    end for
21:    for i = N − 1; i > 0; i = i − 1 do
22:        swap(A, 0, i)
23:        heapify(A, 0, i)
24:    end for
25: end function
```

Uzmimo za primjer brojeve iz ilustracije 2.7. ali promijenjenog redoslijeda, na primjer 23, 12, 5, 83, 37, 32. Pogledajmo prvo fazu izgradnje gomile:



Slika 2.8. Izgradnja gomile početnog niza

Krećemo od linije 18 sa izgradnjom gomile od početnog niza te želimo izgraditi *max heap*. Počinjemo od indeksa $i = \left\lfloor \frac{i}{2} \right\rfloor - 1$ te se krećemo prema korijenu stabla, odnosno prema početku niza sve do indeksa 0. Krećemo od ovog čvora zbog toga što se na tom indeksu nalazi zadnji čvor stabla koji nije list. Kako smo već spomenuli, gomila je **potpuno** binarno stablo što znači da se radi o stablu kojem je svaka razina osim zadnje potpuno popunjena. Čvorovi koji se nalaze na indeksima $\left\lfloor \frac{i}{2} \right\rfloor \ldots n - 1$ su listovi stabla. Za njih vrijedi *pravilo gomile* zbog toga što oni nisu roditelji niti jednom drugom čvoru. Zbog toga ne postoji potreba za ispravljanjem tog dijela stabla. Na drugu stranu, čvorove koji posjeduju djecu potrebo je ispraviti u slučaju da za njih ne vrijedi pravilo *max heap* gomile (sva djeca čvora sadrže vrijednost manju od vrijednosti tog čvora). To radimo pomoću funkcije *heapify*. Funkcija kao parametre uzima niz podataka, indeks koji je potrebno provjeriti i ispraviti te broj elemenata u nizu podataka. Premisa funkcije je jednostavna: provjeri da li ijedno dijete trenutnog čvora sadrži vrijednost veću od njegove vrijednosti te ako je to slučaj, zamijeni ih i napravi isto sa zamijenjenim djetetom zbog moguće promjene stanja sa dječjim čvorovima tog čvora (rekurzivan poziv). Navedeni postupak, kao što možemo vidjeti na slici 2.8., ponavljamo sve do korijena stabla kako bi osigurali svojstvo gomile za sve potrebne čvorove stabla.

Nakon faze kreiranja gomile, slijedi faza sortiranja prema dolje. Kako smo funkciju

heapify već objasnili, postupak sortiranja postaje trivijalan. U petlji krećemo od zadnjeg elementa niza te se krećemo prema korijenu stabla. Mijenjamo trenutni element na kojem se nalazimo (koji će cijelo vrijeme predstavljati zadnji element gomile) sa korijenom (čija će vrijednost po definiciji *max heap* gomile uvijek biti najveća u cijelom stablu). Nakon zamjene elemenata, virtualno smanjujemo veličinu gomile tako što na liniji 23 pri pozivu funkcije *heapify* umjesto veličine niza A predajemo trenutni indeks i na kojem se nalazimo. Prema opisanom postupku, kontinuirano stavljamo najveći element na kraj niza, kratimo ukupnu duljinu niza za 1 te popravljamo gomilu od korijena stabla. Provedbom navedenog postupka dobivamo sortiran niz od najmanjeg elementa prema najvećem.

Svojstva *Heap Sort* algoritma:

Svojstvo	Vrijednost
Vremenska složenost	$O(nlog_2n)$
Prostorna složenost	$O(1)$

Kako je vremenska složenost *heapify* metode jednaka $O(log_2n)$, te taj postupak ponavljamo n puta, dobivamo slijedeće:

$$T(n) = O(n) \cdot O(log_2n) \tag{2.13}$$

Prema 2 vrijedi:

$$O(n) \cdot O(log_2n) = O(n \cdot log_2n) \tag{2.14}$$

Uvrštavanjem u 2.13 dobivamo:

$$T(n) = O(n \cdot log_2n) \tag{2.15}$$

2.2.7. Counting Sort

Counting Sort je algoritam sortiranja koji nije baziran na direktnoj usporedbi brojeva nego se temelji na brojanju ponavljanja pojedinih elemenata iz ulaznog niza. Ideja al-

goritma je prebrojati koliko se puta pojedini element ponavlja u ulaznom nizu te zatim prema tim informacijama kreirati novi niz i postaviti prebrojane vrijednosti u njega na ispravne pozicije. Algoritam je efikasan kada se u ulaznom nizu nalaze vrijednosti u ograničenom rasponu.

Pseudo kod *Counting Sort* algoritma:

Algorithm 7 Counting Sort algoritam

```
1:  function CountingSort(A)
2:      Find the maximum value m in the array A
3:      Initialize array C of length m + 1 with all zeros
4:      Initialize array B of length A.length
5:      for each element a in A do
6:          C[a] ← C[a] + 1
7:      end for
8:      for i from 1 to m do
9:          C[i] ← C[i] + C[i − 1]
10:     end for
11:     for i from A.length down to 1 do
12:         B[C[A[i]] − 1] ← A[i]
13:         C[A[i]] ← C[A[i]] − 1
14:     end for
15:     return B
16: end function
```

Pogledajmo detaljno na koji način funkcionira algoritam. Za početak inicijaliziramo dva nova niza - B i C. Niz B ćemo koristiti za kreiranje novog niza u kojem će se nalaziti ispravno sortirane vrijednosti iz niza A. Pronalazimo vrijednost m - maksimalna vrijednost koja se nalazi u nizu A. Koristimo dobivenu vrijednost kako bi mogli prebrojati sve moguće vrijednosti koje se nalaze u nizu A. Niz C ćemo koristiti za brojanje pojedinih vrijednosti iz niza A te ga stoga inicijaliziramo sa veličinom $m + 1$ i postavljamo sve brojače na nulu. Koristimo $m + 1$ zbog toga što se pretpostavlja da je prvi mogući indeks niza jednak nuli.

Nakon toga u prvoj petlji prolazimo kroz niz A i pamtimo brojeve ponavljanja pojedinih vrijednosti iz niza u niz C. Zatim, u drugoj petlji ažuriramo niz C kako bi svaki element niza sadržavao zbroj prethodnih brojača (na poziciji 3 u nizu nalazit će se informacija o broju elemenata iz niza A koji su manji ili jednaki broju 3). Ovime osiguravamo ispravan način rada algoritma te sposobnost algoritma da postavi vrijednosti iz niza A na

ispravno mjesto u nizu B.

Za kraj, u zadnjoj petlji postavljamo vrijednosti iz niza A na ispravno mjesto pomoću niza C. Iteriramo unazad kroz početni niz A kako bi osigurali **stabilnost** algoritma (u slučaju običnih vrijednosti nema neke razlike ali je ključno ako bi elementi sadržavali neke druge informacije te ako bi trebali ostati u svojem relativnom redoslijedu kao i u nizu A). Prvo postavljamo $A[i]$ na točno mjesto u nizu B. Objašnjenje izraza $B[C[A[i]]-1] \leftarrow A[i]$:

- $A[i]$ je trenutni element iz ulaznog niza
- $C[A[i]]$ sadrži broj elemenata koji su manji ili jednaki $A[i]$
- $C[A[i]]-1$ je vrijednost točnog indeksa u nizu B (počinjemo od 0) gdje je potrebno postaviti $A[i]$

Nakon postavljanja vrijednosti u niz B, smanjujemo brojač $C[A[j]]$ za jedan. Na taj način će slijedeći element koji će biti potrebno postaviti na isto mjesto biti postavljen jedno mjesto prije trenutno postavljenog elementa.

Analizirajmo vremensku i prostornu složenost ovog algoritma. Što se tiče prostorne složenosti, na početku radimo inicijalizaciju dva niza, jedan duljine n te drugi duljine $m+1$ gdje je m maksimalna vrijednost pronađena u nizu A. Time dobivamo slijedeću prostornu složenost:

$$S(n) = O(n) + O(M+1) \tag{2.16}$$

Prema 3 vrijedi:

$$O(m+1) = O(m) \tag{2.17}$$

Prema 1 vrijedi:

$$O(n) + O(m) = O(n + m) \tag{2.18}$$

Uvrštavanjem ovih vrijednosti u početnu jednadžbu dobivamo slijedeću prostornu složenost:

$$S(n) = O(n + m) \tag{2.19}$$

S druge strane, za vremensku složenost možemo pogledati tri petlje koje koristimo za sortiranje početnog niza A. Imamo slijedeće korake:

- Inicijalizacija niza B: $O(n)$
- Inicijalizacija niza C: $O(m + 1) \approx O(m)$
- Brojanje pojavljivanja elemenata iz A: $O(n)$
- Ažuriranje niza C: $O(m)$
- Izgradnja niza B: $O(n)$

Dobivamo slijedeći zbroj:

$$T(n) = O(n) + O(m) + O(n) + O(m) + O(n) \tag{2.20}$$

Prema propozicijama 1 i 3 dobivamo:

$$T(n) = O(n + k) \tag{2.21}$$

Vidimo kako je dobiveni algoritam vrlo efikasan u slučajevima gdje ulazni niz sadrži vrijednosti iz ograničenog raspona. Na primjer, ovakav algoritam ne bi bio primjenjiv ako bi ulazni niz sadržavao realne brojeve ili velike brojeve koji bi zahtjevali alokaciju i inicijalizaciju niza C velike duljine. U slučaju da ulazni niz sadrži podatke realnog tipa ili podatke koji imaju vrlo širok raspon vrijednosti, preporučuje se korištenje drugih

algoritama za sortiranje koji su općenitiji i efikasniji u takvim situacijama.

2.2.8. Radix Sort

Radix Sort je algoritam baziran na primjeni *Counting Sort* algoritma više puta kako bi sortirao brojčane elemente po znamenkama ili znakovne nizove po znakovima. Algoritam radi na način da broj razdvoji na znamenke, sortira te brojeve po svakoj znamenci primjenjujuću *Counting Sort* te tako postepeno gradi sortirani niz.

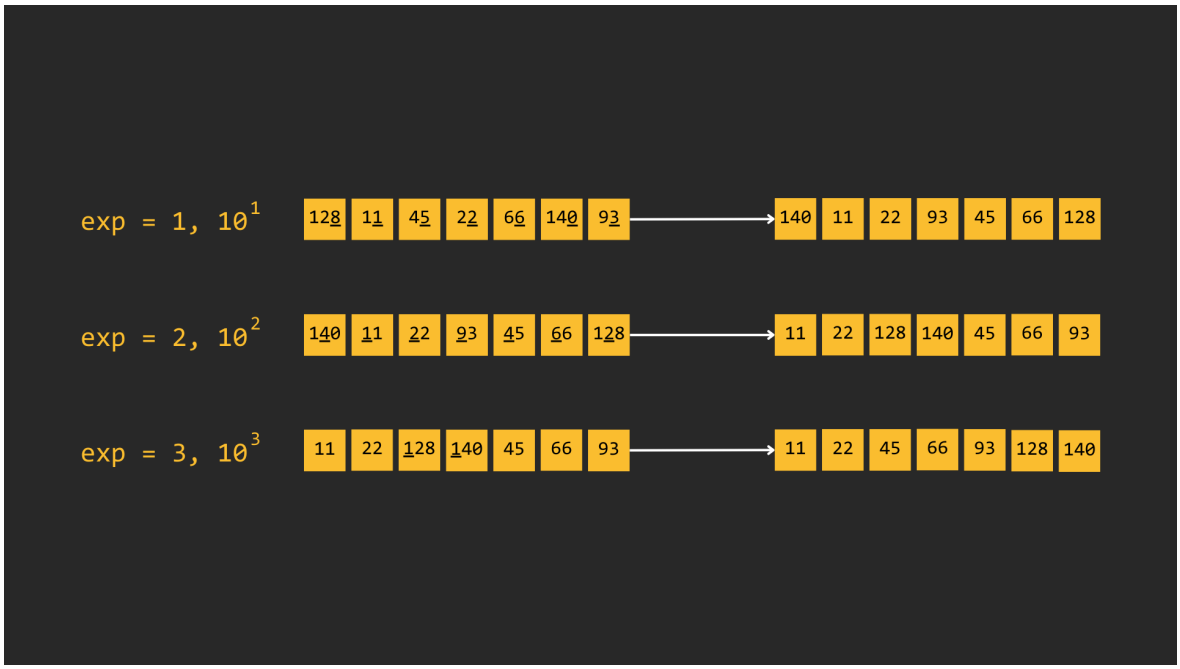
Pogledajmo pseudo kod algoritma.

Algorithm 8 Radix Sort algoritam

```
1: function RadixSort(A)
2:     Find the maximum value max in the array A
3:     for exp from 1 to digits count in max do
4:         Apply counting sort on A by using exponent exp
5:     end for
6: end function
```

Vidimo kako je algoritam sam po sebi vrlo jednostavan. *Counting Sort* algoritam bi trebao biti modificiran kako bi funkcija mogla primiti parametar exp kako bi algoritam znao sortirati niz prema pojedinoj znamenci. Kao što je navedeno u 2.2.7., *Counting Sort* broji pojavljivanja pojedinog elementa u predanom nizu. Za *Radix Sort*, *Counting Sort* treba gledati samo jednu znamenku te će niz C uvijek biti duljine 10 (znamenke 0-9).

Navedeni postupak se ponavlja onoliko puta koliko znamenaka ima maksimalni pronađeni broj u ulaznom nizu A. Na taj način osiguravamo sortiranje prema svim znamenkama broja s desna na lijevo (exp kreće od 1).



Slika 2.9. Radix Sort

Označimo sa d broj znamenki maksimalnog broja iz ulaznog niza. Za vremensku i prostornu složenost dobivamo slične vrijednosti kao i u *Counting Sort* algoritmu ($m = 10$):

Svojstvo	Vrijednost
Vremenska složenost	$O(d \cdot (n + m))$
Prostorna složenost	$O(n + m)$

Kao što vidimo, algoritam je vrlo povezan sa *Counting Sort* algoritam te je također koristan u istim slučajevima kao i *Counting Sort*, ali i u slučajevima gdje je potrebno sortirati znakovne nizove čija se duljina nalazi u određenom rasponu. Na drugu stranu, algoritam je manje koristan kada se koriste realni brojevi te je u takvim slučajevima potrebno koristiti druge algoritme opće namjene poput *Merge Sort* ili *Heap Sort*.

2.2.9. Bucket Sort

Bucket Sort je algoritam sortiranja koji također nije baziran na direktnoj usporedbi elemenata ulaznog. Algoritam radi tako da sprema elemente u određene spremnike te zatim sortira elemente unutar svakog spremnika korištenjem bilo kojeg općeg algoritma sortiranja. Nakon navedenog postupka, svi elementi se spajaju nazad u jedan niz. Kako bi ovakav algoritam bio učinkovit, postoji preduvjet uniformne distribucije elemenata. Ako

elementi u nizu nisu jednoliko raspoređeni ili ako postoje velike među njihovim vrijednostima, sortiranje pojedinog spremnika može trajati puno duže nego što je zamišljeno.

Pseudo kod algoritma:

Algorithm 9 Bucket Sort algorithm

```
1:  function BucketSort(A)
2:      Initialize n empty buckets
3:      Find max from A
4:      for i from 0 to n − 1 do
5:          Create empty bucket[i]
6:      end for
7:      for each i in A do
8:          index ← ⌊n × i/(max+1)⌋
9:          Insert A[i] into bucket[index]
10:     end for
11:     for i from 0 to n − 1 do
12:         Sort elements in bucket[i]
13:     end for
14:     Merge sorted elements
15:     k ← 0
16:     for i from 0 to n − 1 do
17:         for each j in bucket[i] do
18:             A[k] ← bucket[i][j]
19:             k ← k + 1
20:         end for
21:     end for
22: end function
```

Započinjemo inicijalizacijom praznih spremnika. Broj spremnika ovisi o situaciji te se ovdje pretpostavlja da je poznato na koji način će biti raspoređeni spremnici kako bi se elementima omogućila uniformna raspodjela. Za broj spremnika uzimamo n. Nakon toga se svaki element sprema u odgovarajući spremnik čiji indeks dobivamo množenjem broja spremnika i odgovarajućim indeksom podijeljenim sa najvećim elementom u nizu. Prolazimo kroz svaki spremnik te ga sortiramo proizvoljnim algoritmom poput *Insertion Sort* ili *Quick Sort*. Nakon što je svaki spremnik sortiram, prolazimo kroz sve spremnike te jedan za drugim elemente spremamo u konačni niz koji vraćamo korisniku.

Uzmimo za primjer brojeve 73, 15, 42, 87, 23, 36 i 50 te uzmimo 10 kao n odnosno broj praznih spremnika na početku algoritma. U prvom koraku stvaramo 10 praznih spremnika. Prolazimo kroz niz te raspodjeljujemo elemente u spremnike. Uzmimo za primjer

broj 87. Dijelimo ga sa 88 te dobivamo ≈ 0.97. To množimo sa 10 te uzimamo cijeli dio i time dobivamo indeks 9 što znači da 87 spremamo u zadnji spremnik (spremnik 10). Na isti način raspodjeljujemo sve ostale elemente koji će u ovom slučaju biti svaki u svom spremniku te neće ni postojati potreba za sortiranjem svakog spremnika. To općenito nije slučaj pa zato postoji potreba za sortiranjem svakog spremnika. Na kraju prolazimo kroz sve spremnike i dodajemo elemente jedan po jedan u završni niz. Tako dobivamo sortiran niz $[15, 23, 36, 42, 50, 73, 87]$.

Analizirajmo vremensku i prostornu složenost algoritma. Za vremensku složenost imamo tri koraka:

- Raspodjela elemenata u spremnike: $O(n)$
- Sortiranje elemenata u svakom spremniku: $O(n^2)$ u najgorem slučaju (npr. *Insertion Sort*)
- Spajanje spremnika u konačni niz: $O(n)$

Zbrajanjem ovih vrijednosti dobivamo slijedeću vremensku složenost:

$$T(n) = O(n) + O(n^2) + O(n) = O(n^2) \tag{2.22}$$

Za prostornu složenost dobivamo $O(n + k)$ gdje je k broj elemenata u svakom spremniku.

Svojstva *Bucket Sort* algoritma.

Svojstvo	Vrijednost
Vremenska složenost	$O(n^2)$
Prostorna složenost	$O(n + k)$

3. Implementacija

U sklopu ovog rada potrebno je implementirati aplikaciju koja će nam služiti za vizualizaciju gore obrađenih algoritama sortiranja. Aplikacija bi trebala prikazivati način odabira elemenata koji se trenutno uspoređuju te njihovu zamjenu u slučaju promjene redoslijeda elemenata. Također je potrebno naznačiti na kojoj liniji koda se trenutno nalazimo kako bi korisnik bio u mogućnosti točno razumjeti kako algoritam radi te kako dolazi do sortiranja ukupnog niza podataka. Za implementaciju takve aplikacije odabrano je korištenje WEB tehnologija, odnosno izrada aplikacije u obliku WEB stranice. Točnije, koristit će se biblioteke *React* za izradu sučelja aplikacije i *MaterialUI* za korištenje jednostavnih React komponenata. Odabrani jezik implementacije je Typescript.

Za početak ćemo reći nešto općenito o vizualizaciji podataka te ćemo odlučiti na koji način bi mogli prikazati nama potrebne podatke kako bi korisnik mogao što jasnije pratiti tok podataka u odabranom algoritmu sortiranja. Nakon toga ćemo opisati strukturu projekta i detaljnije objasniti odabrane tehnologije za izradu aplikacije. Uzet ćemo u obzir odabranu metodu vizualizacije te ćemo kreirati *React* komponente za prikaz podataka. Nakon toga ćemo razmotriti način na koji bi trebali strukturirati kod kako bi mogli na više neovisnih načina prikazivati vizualizaciju podataka pojedinog algoritma sortiranja. Za kraj, biti će dan opis bitnijih dijelova koda koji omogućuju rad aplikacije.

3.1. Vizualizacija podataka

Kako je navedeno u [4], vizualizaciju podataka definiramo kao grafičku reprezentaciju informacija i podataka. Često je potrebno podatke u tekstualnom obliku na neki način grafički prikazati kako bi dobili jasniji dojam o podacima te o njihovoj interpretaciji. U takvim slučajevima koristimo vizualne elemente poput grafova, grafikona, dijagrama i tablica kako bi tim podacima pridonijeli neku grafičku vrijednost te kako bi lakše mogli

vizualizirati ili čak uspoređivati njihove vrijednosti. Također, na taj način je moguće razumjeti i primjetiti neke uzorke ili ekstreme među podacima. Važnost vizualizacije podataka leži u tome da znatno pomaže ljudima razumjeti određen skup podataka.

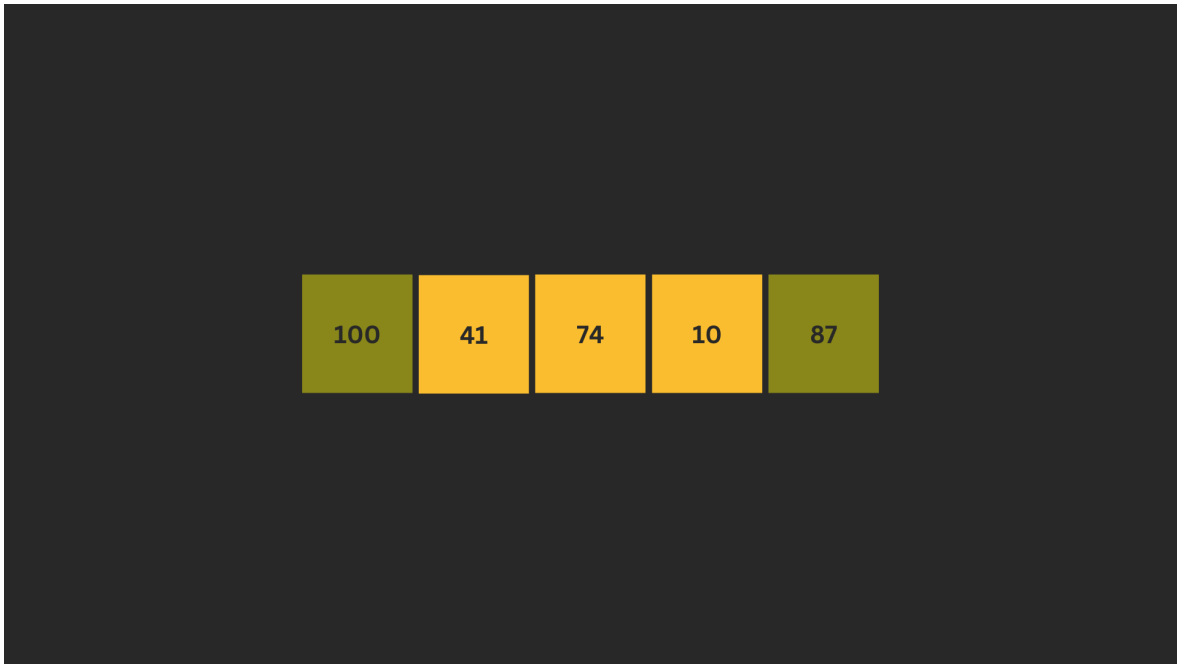
U našem slučaju, imamo slijedeće zahtjeve:

- Mogućnost vizualizacije brojeva iz niza podataka
- Mogućnost uposredbe navedenih brojeva
- Mogućnost označavanja pojedinačnih vrijednosti
- Mogućnost vizualizacije zamjene elemenata u nizu

Za navedene zahtjeve, postoje slijedeće mogućnosti prikaza podataka:

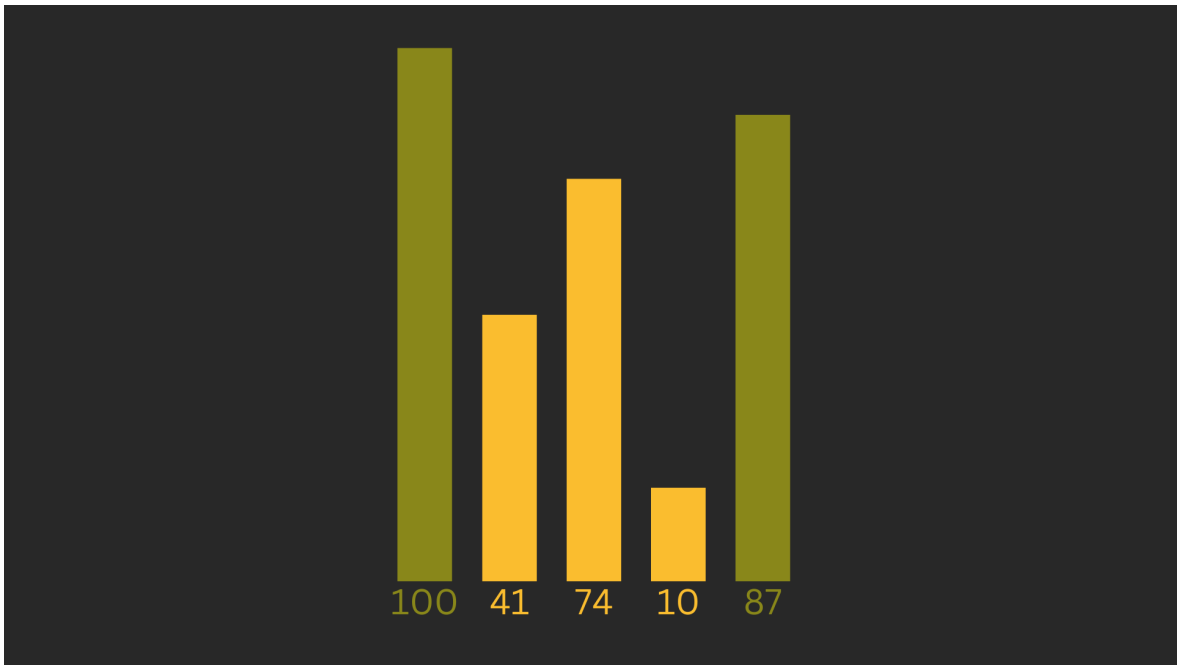
- Prikaz pojedinih elemenata jedan do drugog unutar određenog oblika (kvadrat ili krug)
- Korištenje stupčastog grafikona (jedan element odgovara jednom stupcu u grafikonu)

Za prvi način, radi se o prikazu sličnom kao u prijašnjim slikama u poglavlju 2.:



Slika 3.1. Kvadratni prikaz elemenata

Za drugu mogućnost prikaza podataka, radi se o slijedećem načinu vizualizacije:



Slika 3.2. Stupčasti prikaz elemenata

Obje metode vizualizacije su validne za naše navedene zahtjeve. Postoji jedino razlika u drugom zahtjevu za našu aplikaciju (*Mogućnost uposredbe navedenih brojeva*). Iako tehnički možemo usporediti trenutno označene brojeve ako koristimo prvi način vizualizacije elemenata (kvadratni prikaz), njihova vrijednost nije vizualno reprezentirana kao što je slučaj u stupčastom prikazu. Ako koristimo stupčasti prikaz, ispunjeni su svi uvjeti koje naša aplikacija treba ispunjavati i također možemo lakše vizualizirati o kojem se elementu trenutno radi. Na taj način možemo pratiti veličinu trenutnog stupca te tako **vizualiziramo** vrijednost podatka umjesto da razmišljamo o brojci koja se nalazi unutar kvadrata.

Korisnik u tom slučaju jasnije percipirati vrijednost pojedinog podatka uspoređujući veličinu stupca.

Prema navedenome, u implementaciji ćemo koristiti **stupčasti grafikon** za prikaz elemenata niza čije ćemo sortiranje vizualizirati. Svaki element će biti predstavljen svojim stupcem te će njegova vrijednost biti reprezentirana veličinom (točnije visinom) stupca.

3.2. Struktura projekta

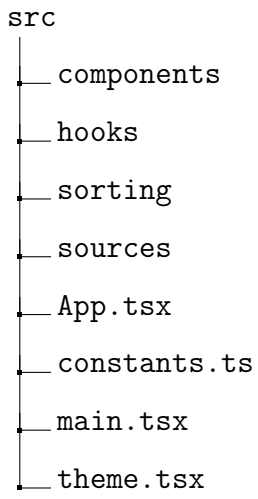
Što se tiče osnovne strukture projekta, koristimo dvije primarne tehnologije - NodeJS i ViteJS. NodeJS je odabran zbog toga što omogućava jednostavno korištenje tehnologija poput *Typescript* i *React* što znatno olakšava razvoj složenih korisničkih sučelja. Prednost korištenja ViteJS alata nalazi se u tome što nam alat pruža razne napredne mogućnosti za brži razvoj i optimizaciju aplikacije. Jedna od tih mogućnosti je učitavanje modula uživo (engl.*hot reload*) što nam omogućuje znatno brži razvoj sučelja u odnosu na tradicionalni pristup gdje bi prvo trebali promjeniti određeni dio koda ili komponente te nakon toga izgraditi cijeli projekt i provjeriti napravljene promjene.

Koristimo slijedeću strukturu projekta:

```
visort
├── src
├── public
├── index.html
├── package.json
├── tsconfig.json
└── vite.config.ts
```

U *src* direktoriju nalazimo implementacijski kod aplikacije. *public* direktorij sadrži statičke dijelove aplikacije (npr. slike ili font datoteke). *tsconfig.json* datoteka definira opcije koje su potrebne Typescript kompilatoru. *vite.config.ts* datoteka sadrži konfiguraciju ViteJS razvojnog poslužitelja te definira njegove dodatke (engl.*plugins*) koje je moguće dodavati po potrebi.

Sada ćemo dati kratak pregled strukture implementacijskog dijela projekta te ćemo objasniti u koje dijelove je projekt podijeljen. Nakon toga ćemo neke dijelove detaljno opisati kako bi mogli razumjeti na koji način je dizajnirana i implementirana aplikacija za vizualizaciju algoritama sortiranja.



components direktorij sadrži kreirane *React* komponente naše aplikacije. U *React* biblioteci, komponente predstavljaju način za odvajanje funkcionalnosti te za kreiranje ponovno iskoristivog dijela koda čija je svrha izgraditi određeni dio korisničkog sučelja. U našoj aplikaciji možemo zamisliti nekoliko odvojenih dijelova koji služe za prikaz određenog dijela aplikacije: kontrole vizualizatora, stupčasti grafikon, stupac grafikona i preglednik izvornog koda algoritma. Navedene komponente će biti detaljno opisane u slijedećem odjeljku.

U *hooks* direktoriju nalazimo kreirane *React Hooks*. Kako je navedeno u [5], *React Hookovi* nam omogućuje korištenje raznih *React* značajki u komponentama. U našem slučaju, služit će nam prvenstveno za organizaciju podataka te za odvajanje stanja komponente od korisničkog sučelja komponente.

sorting direktorij sadrži glavni dio aplikacije koji se bavi vizualizacijom koraka algoritama sortiranja. U njemu ćemo pronaći klase koje pružaju mogućnost za označavanje stupaca u grafikonu i trenutne linije koda na kojoj se nalazimo.

sources direktorij sadrži definicije pseudo koda svakog algoritma poput *Bubble Sort* ili *Insertion Sort*. Odabrani jezik za reprezentaciju pseudo koda je *Javascript*. Svaki pseudo kod spremljen je u varijablu tipa *string*.

Što se tiče ostalih datoteka, *App.tsx* sadrži korijenski dio korisničkog sučelja. Sastavljena je od kreiranih komponenti iz *components* direktorija te koristi jedini kreirani *hook* **useVisualizer**. Datoteka *constants.ts* sadrži konstantne vrijednosti aplikacije poput minimalnog i maksimalnog broja elemenata niza čije će vrijednosti biti porebno sortirati.

U datoteci *main.tsx* nalazimo glavnu korijensku komponentu čiji će sadržaj biti prikazan u datoteci *index.html* nakon što izgradimo projekt. Za kraj, datoteka *theme.tsx* sadrži definiciju *Material UI* teme. Odabrana tema za korisničko sučelje aplikacije je gruvbox. Tema je vrlo jednostavna te nam omogućuje vizualno distinktivno označavanje određenih stupaca grafikona te trenutne linije koda na kojoj se nalazimo.

3.2.1. Komponente

U ovom odjeljku ukratko ćemo opisati kreirane komponente koje nam služe za definiciju korisničkog sučelja.

BarComponent

BarComponent komponenta prestavlja pogled na jedinični stupac u stupčastom grafikonu koji će reprezentirati elemente generiranog niza čiji ćemo sadržaj sortirati.

```
1  interface BarComponentProps {
2    bar: BarData,
3    delay: number,
4  }
5
6  export const BarComponent = styled('div')(function (props: BarComponentProps) {
7    return {
8        backgroundColor: props.bar.isHighlighted ? theme.palette.primary.main :
             theme.palette.success.main,
9        height: `${props.bar.value}%`,
10       '&:hover': {
11           transitionDuration: '100ms',
12           filter: `drop-shadow(0 0 10px ${props.bar.isHighlighted ?
                 theme.palette.primary.main : theme.palette.success.main});`
13       },
14       margin: '1px',
15       flexGrow: 1,
16       transitionDuration: `${props.delay / 2}ms`,
17   }
18 });
```

Komponenta je kreirana kao *styled* komponenta. Kao svojstva, prima objekt koji sadrži vrijednosti *bar* i *delay*. *bar* sadrži svojstva pojedinog stupca - njegovu visinu te vrijednost koja označava da li je trenutno stupac označen ili nije. *delay* sadrži vrijednost koja predstavlja odgodu u milisekundama koju je potrebno uzeti u obzir u slučaju promjene određenog svojstva nad trenutnim stupcem. Naime, kako želimo imati podršku

za promjenu brzine vizualizacije, potrebno je također uskladiti odgodu animacije tranzicije između stanja pojedinih stupaca. Vidimo kako je na liniji 8 pozadinska boja stupca odabrana prema svojstvu *isHighlighted*. Također na liniji 9 možemo vidjeti kako je visina stupca određena pomoću svojstva *value* is *bar* objekta.

SourceCodeComponent

SourceCodeComponent komponentu koristimo za prikaz pseudo koda algoritama, čije se vrijednosti nalaze u direktoriju *sources* kako je navedeno u prijašnjem odjeljku.

```
1  export type SourceCodeComponentProps = {
2    highlightedLines: Array<number>,
3    sortingAlgorithm: SortingAlgorithm,
4  }
5
6  export function SourceCodeComponent(props: SourceCodeComponentProps) {
7    const algSource = SourceCodeProvider.getAlgorithmSource(props.sortingAlgorithm)
8    return (
9        <SyntaxHighlighter
10           language="javascript"
11           style={gruvboxDark}
12           wrapLines={true}
13           showLineNumbers={true}
14           lineProps={function (lineNumber) {
15               const style: any = {}
16               for (const hl of props.highlightedLines) {
17                   if (hl == lineNumber) {
18                       style.backgroundColor = theme.palette.secondary.dark
19                       break;
20                   }
21               }
22               return { style: style }
23           }}
24       >
25           {algSource}
26       </SyntaxHighlighter>
27   );
28 };
```

Kao što vidimo na liniji broj 7, koristi se **singleton** klasa **SourceCodeProvider** za dohvat pseudo koda algoritma prema trenuto odabranom algoritmu u aplikaciji. Također vidimo kako je jedno svojstvo koje komponenta prima *highlightedLines*. Ono predstavlja niz linija čiji je sadržaj potrebno naznačiti. Za prikaz pseudo koda algoritama (za čiji je sadržaj odabran jezik *Javascript*) koristimo vanjsku biblioteku react-syntax-highlighter.

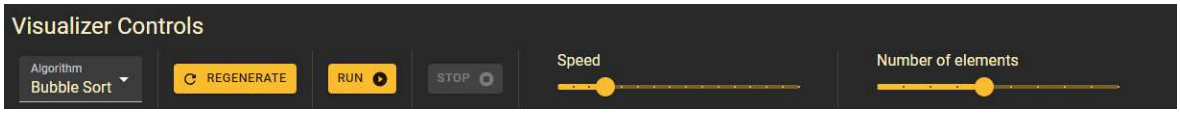
Komponenta je podešena tako da koristi označavanje sintakse Javascript jezika. Na liniji broj 14 možemo vidjeti dio koda koji prolazi kroz predani niz brojeva linija te označava liniju sa pozadinskom bojom u slučaju da se njen broj nalazi u predanom nizu *highlightedLines*.

```
1  function bubbleSort(array) {
2      const n = array.length;
3
4      for (var i = 0; i < n; i++) {
5          for (var j = 0; j < (n - i - 1); j++) {
6              if (arr[j] > arr[j + 1]) {
7                  var temp = arr[j]
8                  arr[j] = arr[j + 1]
9                  arr[j + 1] = temp
10             }
11         }
12     }
13
14     return array;
15 }
```

Slika 3.3. Komponenta za prikaz pseudo koda algoritma

VisualizerControlsComponent

VisualizerControlsComponent komponentu koristimo za izdvajanje svih kontrola pomoću kojih možemo utjecati na ponašanje vizualizatora algoritama. Komponenta sadrži sve kontrole kao na primjer kombinatorni okvir za odabir algoritma, klizač za podešavanje brzine kojom će vizualizator raditi te gumbe za ponovno generiranje niza podataka i za pokretanje/zaustavljanje vizualizatora.



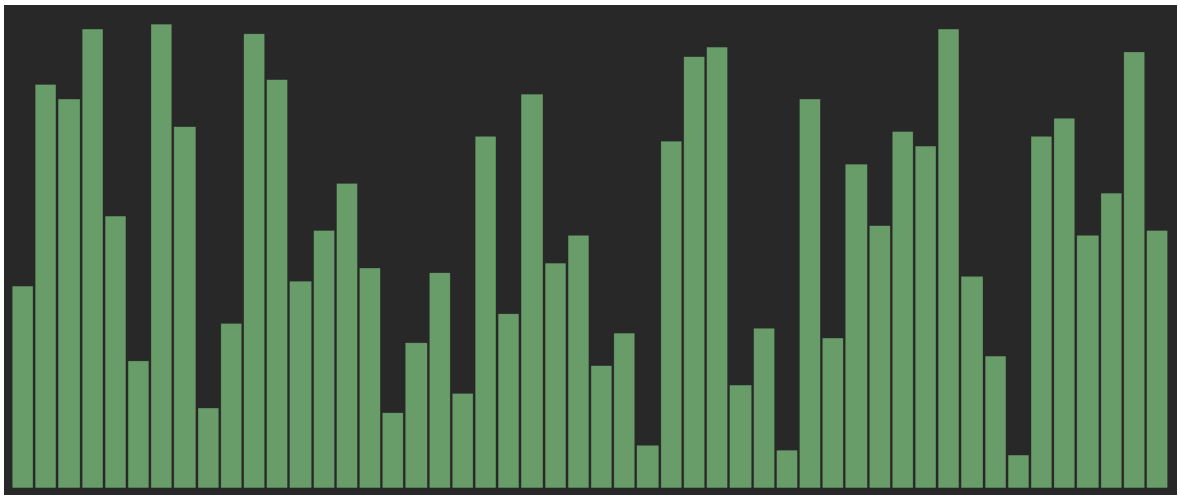
Slika 3.4. Kontrole vizualizatora

VisualizerFrameComponent

VisualizerFrameComponent komponenta predstavlja ranije spomenuti grafikon stupaca. Komponenta sama po sebi ne prima podatke kao ulaz nego predstavlja roditeljsku komponentu svih stupaca koji će biti prikazani pomoću ranije objašnjene **BarComponent** komponente. Komponenta radi pomoću *flexbox* HTML sustava za raspored elemenata.

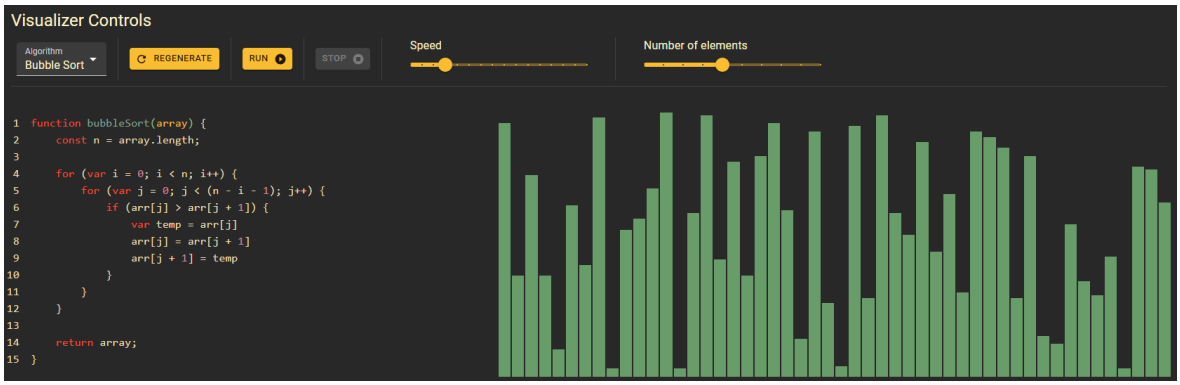
```
export const VisualizerFrameComponent = styled('div')({
  display: 'flex',
  flexDirection: 'row',
  justifyContent: 'flex-start',
  alignItems: 'flex-end',
  height: '100%',
  padding: '10px'
})
```

Koristimo standardni način rasporeda elemenata pomoću definicije roditeljskog *div* elementa čije svojstvo *display* poprima vrijednost *flex*. Unutarnje elemente usklađujemo na kraj komponente pomoću *alignItems* svojstva. Na taj način će se stupci nalaziti na dnu roditeljskog *div* elementa umjesto na početku.



Slika 3.5. Komponenta stupčastog grafikona

Sve navedene komponente čine korisničko sučelje naše aplikacije. U *App.tsx* datoteci koristimo sve navedene komponente te ih povezujemo pomoću *Material UI* biblioteke te *Grid* komponente unutar navedene biblioteke. Također u aplikaciji koristimo **useVisualizer** *hook* za odvajanje ukupnog stanja aplikacije od definicije korisničkog sučelja. Kada povežemo sve objašnjene komponente naše aplikacije, dobivamo slijedeći izgled:

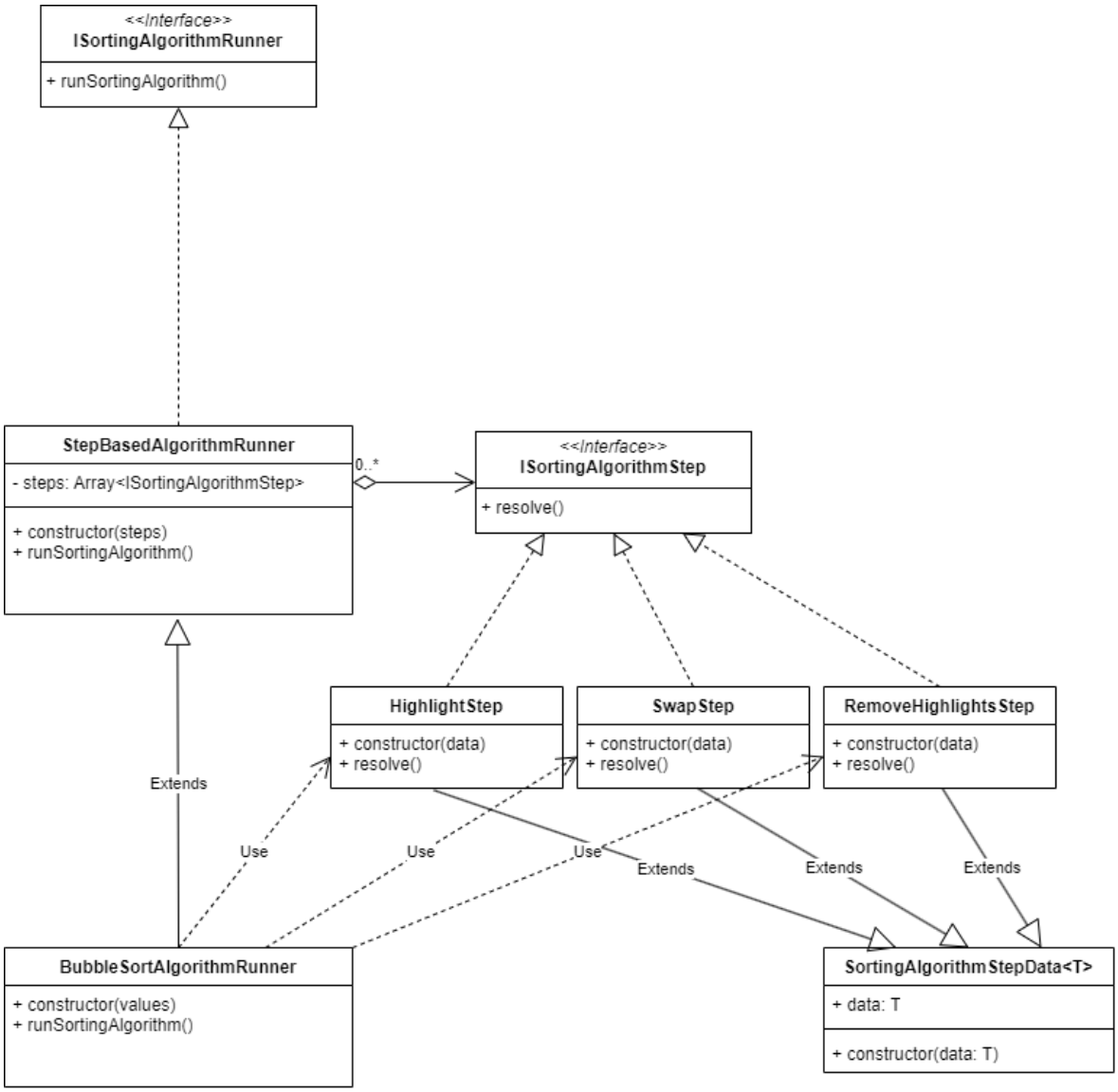


Slika 3.6. Korisničko sučelje aplikacije

3.2.2. Implementacija vizualizacije

Za implementaciju vizualizacije koristit ćemo dva oblikovna obrasca: *Strategija* (engl.*Strategy*) i *Okvirna Metoda* (engl.*Template Method*). Kako je navedeno u [6], *Strategija* je ponašajni obrazac koji nam omogućuje definiciju obitelji algoritama za koje možemo kreirati zasebne klase koje će potom biti zamjenjive. Na drugu stranu, *Okvirna Metoda* je također ponašajni obrazac u kojem je definiran zajednički algoritam za sve klase koje naslijede klasu koja sadrži okvirnu metodu, ali dopušta tim klasama da utječu na pojedine korake algoritma bez mijenjanja njegove implementacije. U našem slučaju, potrebno je kreirati obitelj postupaka koji će vizualizirati trenutno odabrani algoritam sortiranja nad trenutno generiranim nizom podataka. Tako ćemo definirati sučelje čiju će metodu morati implementirati sve klase koje trebaju biti u mogućnosti vizualizirati pojedini algoritam sortiranja. Također ćemo omogućiti izvođenje ovakve vizualizacije pomoću jedne zajedničke klase koja će biti bazna klasa mnogim algoritmima. Naime, postupak vizualizacije moguće je zamisliti kao niz koraka koje je potrebno obaviti, jedan za drugim, kako bi korisniku predočili način rada pojedinog algoritma. Prema tome ćemo definirati jednu klasu koja će u svojem konstruktoru kao parametar primiti niz definiranih koraka koje je potrebno obaviti nad ulaznim podacima kako bi se vizualiziralo sortiranje podataka. Ta klasa će imati definiranu okvirnu metodu koja će izvoditi dobivene korake jedan po jedan te pauzirati izvođenje slijedećeg koraka za određeno odstupanje koje će biti izračunato na temelju odabrane brzine izvođenja algoritma sortiranja. Svaka klasa koja će naslijediti navedenu klasu okvirne metode, treba u svome konstruktoru generirati niz potrebnih koraka te proslijediti te korake roditeljskoj klasi okvirne metode. Prvo ćemo pogledati UML dijagram koji ćemo bazirati na implementaciju vizualizacije *Bubble Sort*

algoritma pomoću navedenih klasa. Nakon toga ćemo objasniti svaku klasu detaljno te proći kroz proces vizualizacije algoritma *Bubble Sort*.



Slika 3.7. UML dijagram klasa potrebnih za vizualizaciju

Krećemo od sučelja *ISortingAlgorithmRunner*. Sučelje definira metodu *runSortingAlgorithm()* koja će se izvoditi asinkrono. Takav pristup nam je potreban zbog toga što želimo korisniku omogućiti kontinuiranu interakciju sa stranicom. Korisnik mora na primjer biti u mogućnosti zaustaviti vizualizaciju ako mu ista nije više potrebna. Kada bi pokrenuli vizualizaciju sinkrono, korisnik više nebi bio u mogućnosti upravljati stranicom sve dok vizualizacija ne završi. Nakon toga, primjećujemo klasu *StepBasedAlgorithmRunner*. Klasa predstavlja objekt koji će biti u mogućnosti izvoditi vizualizaciju prema definiranom nizu koraka. Klasa radi na slijedeći način: u svojem konstruktoru prima

niz koraka koji implementiraju sučelje *ISortingAlgorithmStep* te u nadjačanoj metodi *runSortingAlgorithm()* prolazi kroz isti niz koraka te pokreće njihovu metodu *resolve()* u kojoj će svaki korak izvršiti potrebne akcije kako bi se simulirao algoritam sortiranja te kako bi se vizualno prikazalo trenutno stanje niza. Vidimo na dijagramu neke primjere takvih koraka koje koristi *Bubble Sort* algoritam. *HighlightStep* korak označava potrebne elemente u nizu te također označava potrebnu liniju pseudo koda. *SwapStep* zamijenjuje dva elementa. *RemoveHighlightsStep* uklanja oznaku sa trenutno označenih elemenata te također briše oznaku sa trenutno označene linije pseudo koda. Svi navedeni koraci naslijeđuju istu klasu *SortingAlgorithmStepData* koja nam služi za jednostavnije upravljanje potrebnim podacima za svaki tip koraka. Kako je navedena klasa generičkog tipa, svaki korak koji ju naslijeđuje može definirati koje podatke mu je potrebno poslati za njegovo uspješno izvršavanje. Parametri potrebni za sve korake se predaju istima u konstruktoru pri njihovoj inicijalizaciji. Na primjer, koraku *SwapStep* potrebno je proslijediti dva indeksa trenutnog niza podataka koje je potrebno zamijeniti. Kao što vidimo, *BubbleSortAlgorithmRunner* naslijeđuje klasu *StepBasedAlgorithmRunner* te radi na principu generiranja navedenih koraka. Prema naznačenome, *BubbleSortAlgorithmRunner* koristi korake *HighlightStep*, *SwapStep* i *RemoveHighlightsStep*. To znači da je *Bubble Sort* algoritam moguće vizualno prikazati pomoću samo tri tipa koraka: korak označavanja, korak zamjene elemenata i korak micanja trenutnih oznaka.

Možemo se zapitati zašto je uopće potrebno odvajati sučelje *ISortingAlgorithmRunner* od *StepBasedAlgorithmRunner* ako je moguće na taj način implementirati vizualizaciju algoritama sortiranja? Razlog tomu jest taj što generiranje koraka na ovaj način može loše utjecati na performanse aplikacije u određenim slučajevima. U konkretnoj implementaciji *Merge Sort* algoritma za prikazivanje promjena koristimo korak zamjene ukupnog niza podataka. Iako se takav korak u *Merge Sort* vizualizaciji ne izvodi često, možemo zamisliti ogromnu količinu podataka koja bi bila generirana ako bi se takav korak ponavljao velik broj puta. Iako je u našem slučaju moguće na ovaj način vizualizirati velik broj postupaka sortiranja, kada bi se našli u situaciji u kojoj generiranje koraka ne bi predstavljalo optimalno rješenje, bilo bi potrebno napisati novu implementaciju vizualizacije nevezanu za prikazivanje pojedinih koraka algoritma. Iz tog razloga odvajamo metodu *runSortingAlgorithm()* u sučelje *ISortingAlgorithmRunner* kako bi imali mogućnost dodati sasvim novu implementaciju vizualizacije ako se ukaže potreba za istom.

Sada kada bolje razumijemo strukturu koda korištenu za implementaciju vizualizacije, možemo pogledati ključne dijelove koda koji omogućavaju ovakav proces vizualizacije:

```
interface ISortingAlgorithmRunner {
  runSortingAlgorithm(
      bars: Array<BarData>,
      setBars: React.Dispatch<React.SetStateAction<Array<BarData>>>,
      setHighlightedLine: React.Dispatch<React.SetStateAction<number>>,
      setIsSuccessSnackbarOpen: React.Dispatch<React.SetStateAction<boolean>>,
      isRunningRef: React.MutableRefObject<boolean>,
      calculatedDelay: number,
  ): Promise<void>
}
```

Metoda *runSortingAlgorithm()* za parametre prima slijedeće vrijednosti:

- **bars** - trenutni generirani stupci unutar stupčastog grafikona
- **setBars** - funkcija pomoću koje *React* mijenja niz stupaca te pokreće ponovno prikazivanje HTML elemenata zahvaćenih promjenom
- **setHighlightedLine** - funkcija za promjenu trenutno označene linije koda
- **setIsSuccessSnackbarOpen** - funkcija koja omogućuje prikaz poruke korisniku o uspješnom sortiranju niza
- **isRunningRef** - referenca na *boolovu* vrijednost koja nam govori da li je vizualizacija trenutno pokrenuta
- **calculatedDelay** - izračunata vrijednost kašnjenja kojom je potrebno odvojiti prikazivanje koraka

Kako je objašnjeno u prethodnom dijelu, sučelje *ISortingAlgorithmRunner* mora biti u mogućnosti omogućiti klasama koje ga implementiraju potpunu kontrolu nad vizualizacijom algoritma sortiranja. Iz tog razloga funkcija prima velik broj parametara pomoću kojih je moguće promijeniti bilo koji korisničkog sučelja kako bi korisniku što točnije bilo prikazano trenutno stanje u kojem se nalazi algoritam sortiranja. Sada možemo pogledati na koji način *StepBasedAlgorithmRunner* koristi ove parametre kako bi pomoću primljenih koraka vizualizirao ponašanje algoritma sortiranja:

```
export class StepBasedSortingAlgorithmRunner {
  async runSortingAlgorithm(
      bars: Array<BarData>,
      setBars: React.Dispatch<React.SetStateAction<Array<BarData>>>,
      setHighlightedLine: React.Dispatch<React.SetStateAction<number>>,
      setIsSuccessSnackbarOpen: React.Dispatch<React.SetStateAction<boolean>>,
      isRunningRef: React.MutableRefObject<boolean>,
      calculatedDelay: number,
  ): Promise<void> {
      isRunningRef.current = true
      let stopped = false
      const initialData = JSON.parse(JSON.stringify(bars))
      const currentData = JSON.parse(JSON.stringify(initialData))
      const ctx: SortingContext = {
          currentData
      }

      for (const step of this.steps) {
          // check if user stopped execution
          if (!isRunningRef.current) {
              // stop running
              stopped = true
              break
          }

          // resolve the step
          step.resolve(
              ctx,
              setBars,
              setHighlightedLine
          )

          // sleep so the user can see the change
          await sleep(calculatedDelay)
      }

      // if the algorithm was stopped - set the bars to the initial array
      if (stopped) {
          setBars(initialData)
      } else {
          setIsSuccessSnackbarOpen(true)
      }

      setHighlightedLine(0)
      isRunningRef.current = false
  }
}
```

Konstruktor u ovom slučaju nije prikazan zbog toga što je trivijalan i dodjeljuje vrijednost primljenih koraka putem parametra klasnoj varijabli *steps*. Na liniji 10 postavljamo vrijednost referenci *isRunningRef* u *true*. Time naznačujemo kako počinjemo sa izvođenjem vizualizacije. Na liniji 14 Kreiramo kontekst sortiranja koji u našem slučaju sadrži samo trenutnu vrijednost niza podataka čijim se sortiranjem bavimo. Počinjemo sa izvođenjem dobivenih koraka na liniji 18. Kako je cijelo vrijeme potrebno provjeravati dali je korisnik zaustavio izvođenje vizualizacije, prije obrade trenutnog koraka provjeravamo referencu *isRunningRef* te u slučaju da je njena vrijednost postavljena u *false* prekidamo izvođenje koraka. U slučaju da je vrijednost reference i dalje *true* nastavljamo sa izvođenjem. Na liniji 27 izvodimo trenutni korak pozivanjem *resolve()* metode. Svaki korak mora biti u mogućnosti manipulirati trenutnim nizom stupaca te trenutno označenom linijom te stoga kao parametre prima funkcije *setBars* i *setHighlightedLine*. Kao prvi parametar metode proslijeđujemo ranije spomenuti kontekst izvođenja vizualizacije. Pomoću njega svaki korak ima pristup trenutnim podacima, odnosno stupcima grafikona. Na taj način korak može pregledati trenutno stanje, izmjeniti ga, te iskoristiti prikupljene informacije kako bi funkcijama *setBars* i *setHighlightedLine* proslijedio točne podatke. Nakon što korak završi sa izvršavanjem svoje metode, pomoću *calculatedDelay* parametra pozivamo *sleep()* funkciju koja će pauzirati izvođenje trenute funkcije za *calculatedDelay* milisekundi.

Na liniji 39, u slučaju da je korisnik prekinuo izvođenje vizualizacije, postavljamo stupce u stupčastom grafikonu na njihove inicijalne vrijednosti kako bi korisnik mogao ponovno pokrenuti vizualizaciju. U slučaju da smo uspješno proveli sve korake, na liniji 41 omogućujemo prikazivanje poruke o uspješnosti izvođenja algoritma sortiranja korisniku.

Na kraju cijelog procesa, brišemo označenu liniju ako je ista preostala te postavljamo referencu vrijednosti o izvođenju procesa vizualizacije u *false*.

Za kraj, možemo pogledati na koji način se generiraju koraci za izvođenje *Bubble Sort* algoritma:

```
export class BubbleSortAlgorithmRunner extends StepBasedSortingAlgorithmRunner {
  constructor(bars: Array<BarData>) {
      let steps: Array<ISortingAlgorithmStep> = []

      const n = bars.length;

      for (var i = 0; i < n; i++) {
          for (var j = 0; j < (n - i - 1); j++) {
              steps.push(
                  new HighlightStep({
                      indexes: [j, j + 1],
                      line: 6
                  })
              )
              if (bars[j].value > bars[j + 1].value) {
                  steps.push(
                      new HighlightStep({
                          indexes: [j, j + 1],
                          line: 7
                      })
                  )
                  var temp = bars[j]
                  steps.push(
                      new HighlightStep({
                          indexes: [j, j + 1],
                          line: 8
                      })
                  )
                  bars[j] = bars[j + 1]
                  steps.push(
                      new SwapStep({
                          first: j,
                          second: j + 1,
                          line: 9
                      })
                  )
                  bars[j + 1] = temp
              }
              steps.push(
                  new RemoveHighlightsStep({
                      indexes: [j, j + 1]
                  })
              )
          }
      }

      super(steps)
  }
}
```

Krećemo sa praznim nizom koraka na liniji 3. Možemo primjetiti kako zapravo izvršavamo *Bubble Sort* algoritam sa dobivenim vrijednostima stupaca koje se nalaze unutar varijable *bars*. Kao što smo već napomenuli u poglavlju o vizualizaciji podataka, kako bi vizualizirali proces sortiranja potrebno je naznačiti svaku usporedbu elemenata te njihovu zamjenu (uz naravno označavanje trenutne linije koda). Na liniji 9 generiramo ranije opisani *HighlightStep* sa indeksima elemenata koje je potrebno naznačiti te sa brojem linije pseudo koda koju je također potrebno istaknuti. Time označavamo liniju broj 15 - usporedbu elemenata na indeksima j i $j + 1$. Nakon toga slijedi označavanje sa različitim linijama koda - koraci na linijama 22 i 29. Kada dođemo do koraka zamjene, generiramo *SwapStep* kojem predajemo podatke o elementima koje je potrebno zamijeniti. Nakon što smo završili sa označavanjem elemenata (neovisno o tome da li smo ih zamijenili), generiramo korak *RemoveHighlightsStep* koji briše oznaku sa elemenata j i $j + 1$. Navedeni korak također briše oznaku sa trenutno označene linije koda.

Ovdje smo detaljno opisali generiranje koraka algoritma sortiranja *Bubble Sort*. U implementaciji, svi ostali algoritmi prate isti pristup izvođenja sortiranja nad predanim nizom podataka te generiranja potrebnih koraka vizualizacije.

Prema opisanoj implementaciji vizualizacije dobivamo vrlo modularnu implementaciju u kojoj možemo na dva načina vizualizirati željene algoritme sortiranja:

- nasljeđivanjem klase *StepBasedAlgorithmRunner* te generiranjem potrebnih koraka za manipulaciju podataka uz izvođenje algoritma sortiranja
- implementacijom sučelja *ISortingAlgorithmRunner* te vizualizacijom za vrijeme izvođenja algoritma sortiranja

4. Primjena algoritama sortiranja

Sortiranje je jedna od temeljnih operacija u računalnim sustavima koja se koristi za organizaciju podataka. Efikasni algoritmi sortiranja omogućuju brži pristup i analizu podataka, što je ključno u mnogim aplikacijama. Uvođenjem algoritama sortiranja, mogu se značajno poboljšati performanse sustava koji obrađuju velike količine podataka. Algoritmi opisani u poglavlju 2. nalaze primjenu u različitim područjima, od baza podataka i računalne grafike do e-trgovina i analize podataka. Cilj ovog poglavlja je istražiti različite primjene algoritama sortiranja i pokazati kako njihova primjena omogućuje efikasniju organizaciju i obradu informacija. Spomenut ćemo specifične primjere i scenarije u kojima sortiranje igra ključnu ulogu u optimizaciji performansi i poboljšanju korisničkog iskustva.

4.1. Pretraživanje podataka

Jedna od najvažnijih primjena sortiranja je u pretraživanju podataka. Sortirani podaci omogućuju korištenje učinkovitijih metoda pretraživanja poput binarnog pretraživanja, koje značajno smanjuje vrijeme potrebno za pronalaženje određenog elementa u nizu ($O(log_2 n)$). Na primjer, u telefonskim imenicima ili bazama podataka kontakata, imena su često sortirana abecedno kako bi se omogućilo brzo pronalaženje određenog kontakta.

Primjerice, u bibliotekama digitalnih knjiga kao što su *Kindle* ili *Google Books*, knjige su sortirane po autoru ili naslovu kako bi se korisnicima omogućilo brzo pretraživanje i pristup željenim materijalima. Ovo sortiranje je posebno korisno kada korisnici traže specifične naslove među velikim brojem dostupnih knjiga.

4.2. Baze podataka

U bazama podataka, sortiranje igra ključnu ulogu u optimizaciji upita. U određenim tablicama i relacijama nad kojima je potrebno omogućiti brzu pretragu podataka rade se **indeksi tablica**. Indeksi su sortirani po određenom ključu (pojedini atribut tablice) te omogućuju brzi dohvat zapisa i poboljšavaju performanse pretraživanja. Kada korisnik izvrši upit, baza podataka može brzo pronaći relevantne zapise koristeći sortirane indekse, čime se smanjuje vrijeme potrebno za vraćanje informacije korisniku. Ova sposobnost brzog pretraživanja omogućuje znatno bolje korisničko iskustvo.

4.3. Računalna grafika

U računalnoj grafici, algoritmi sortiranja koriste se za različite tehnike prikazivanja objekata. Primjer je **Painterov** algoritam, koji sortira objekte po udaljenosti od kamere i crta ih od najudaljenijeg do najbližeg. Ovaj pristup osigurava da su svi objekti ispravno prikazani, bez obzira na redoslijed njihovog prikazivanja. Primjer upotrebe Painterovog algoritma može se vidjeti u video igrama i simulacijama, gdje je potrebno ispravno prikazati slojevite objekte. Na primjer, u igri s trodimenzionalnom grafikom, zgrade, drveće i likovi moraju biti prikazani u pravilnom redoslijedu kako bi scena izgledala realistično.

4.4. Prikaz datotečnog sustava

Sortiranje je korisno za organizaciju datoteka i mapa na disku. Mnogi operacijski sustavi koriste sortiranje kako bi prikazali datoteke u abecednom ili kronološkom redu. Ovo korisnicima olakšava pronalaženje i upravljanje datotekama. U aplikacijama kao što su *Windows Explorer* ili *macOS Finder*, datoteke i mape se automatski sortiraju po različitim kriterijima kao što su ime, posljednji datum izmjene ili veličina. Time se omogućuje korisnicima da brzo pronađu potrebne datoteke, što je posebno korisno u poslovnim okruženjima gdje je upravljanje velikim količinama podataka svakodnevna potreba.

4.5. E-trgovine i ostale web stranice

U e-trgovinama, proizvodi se često sortiraju po cijeni, popularnosti ili ocjenama korisnika. Sustavi za preporuku koriste algoritme sortiranja kako bi korisnicima prikazali

relevantne proizvode ili sadržaje. Na primjer, prilikom pretraživanja proizvoda na web stranici, rezultati su često sortirani tako da prikazuju najpopularnije ili najbolje ocijenjene proizvode na vrhu.

Za primjer možemo uzeti web stranice poput *Netflix* ili *Spotify*, gdje se sadržaji kao što su filmovi, serije ili pjesme sortiraju prema preferencijama korisnika. Takvi sustavi koriste algoritme sortiranja i preporučivanja kako bi korisnicima pružili personalizirani sadržaj, što povećava njihovu angažiranost i zadovoljstvo.

4.6. Analiza podataka

U analizi velikih skupova podataka (engl.*Big Data*), sortiranje je ključno za filtriranje i organizaciju podataka prije obrade i analize. Sortirani podaci omogućuju efikasniju primjenu statističkih i analitičkih metoda, te brže generiranje izvještaja i vizualizacija.

Uzmimo za primjer financijsku industriju. U njoj velike količine transakcijskih podataka trebaju biti sortirane i analizirane kako bi se otkrile prijevare ili trendovi na tržištu. Algoritmi sortiranja omogućuju analitičarima brzu obradu ovakvih podataka te donošenje informirane odluke. Slično tome, u zdravstvu, podaci o pacijentima mogu biti sortirani kako bi se brzo identificirali obrasci koji ukazuju na zdravstvene rizike.

5. Zaključak

Algoritmi sortiranja predstavljaju temeljne komponente računalnih znanosti i informatike, koji omogućuju učinkovitu organizaciju i obradu podataka. Razumijevanje i primjena ovih algoritama ključna je za optimizaciju performansi sustava u različitim područjima, uključujući baze podataka, računalnu grafiku, e-trgovinu i analitiku podataka. Kako se svi obrađeni algoritmi razlikuju u svojoj vremenskoj i prostornoj složenosti oni pronalaze svoju primjenu u različitim područjima i situacijama. Za kraj dajemo njihovu tabličnu usporedbu zajedno sa primjerom situacije u kojoj bi njihova primjena bila adekvatna.

Algoritam	$T(n)$	$S(n)$	Primjena
Bubble Sort	$O(n^2)$	$O(1)$	općeniti problemi sa manjim brojem niza podataka; kada nije poželjna alokacija memorije
Selection Sort	$O(n^2)$	$O(1)$	problemi u kojima postoje često ponovljeni ulazni podaci; kada nije poželjna alokacija memorije
Insertion Sort	$O(n^2)$	$O(1)$	slučajevi u kojima je potrebno održati relativni poredak; kada nije poželjna alokacija memorije
Merge Sort	$O(nlog_2n)$	$O(n)$	općeniti problemi sa velikim brojem ulaznih podataka
Quick Sort	$O(nlog_2n)$	$O(1)$	općeniti problemi sa velikim brojem ulaznih podataka
Heap Sort	$O(nlog_2n)$	$O(1)$	općeniti problemi sa velikim brojem ulaznih podataka
Counting Sort	$O(n+k)$	$O(n+k)$	kada su ulazne vrijednosti u ograničenom rasponu
Radix Sort	$O(d \cdot (n+k))$	$O(n+k)$	sortiranje objekata prema ključu čija je abeceda konstantne veličine (brojevi, riječi)
Bucket Sort	$O(n^2)$	$O(n+k)$	ulazne vrijednosti su uniformno distribuirane

Nadalje, vizualizacija algoritama sortiranja igra značajnu ulogu u obrazovanju i istraživanju, omogućujući jasnije razumijevanje njihovih postupaka. Vizualizacije pružaju intuitivan uvid u rad algoritama, olakšavajući učenje i analizu njihovih performansi. Kroz grafičke prikaze, složeni procesi sortiranja postaju pristupačniji, čime se poboljšava sposobnost studenata i istraživača da prepoznaju prednosti i nedostatke različitih

pristupa. U budućnosti, daljnje istraživanje i razvoj novih algoritama sortiranja, kao i poboljšanje postojećih, bit će od ključne važnosti za suočavanje s izazovima obrade sve većih količina podataka. Vizualizacijski alati će i dalje igrati ključnu ulogu u edukaciji, pomažući novim generacijama informatičara da steknu dublje razumijevanje algoritama sortiranja i njihovih primjena. Zaključno, algoritmi sortiranja i njihova vizualizacija ne samo da poboljšavaju učinkovitost računalnih sustava, već i obogaćuju obrazovni proces, omogućujući bolje razumijevanje i inovacije u računalnim znanostima.

Literatura

[1] R. Sedgewick i K. Wayne, *Algorithms*. Addison-Wesley, 2011.

[2] P. Kadam i S. Kadam, "Root to fruit (2): An evolutionary approach for sorting algorithms", *Oriental Journal of Computer Science and Technology*, sv. 7, br. 3, str. 369–376, 2014.

[3] M. Vuković, *Složenost algoritama*. Sveučilište u Zagrebu, 2019.

[4] Tableau, https://www.tableau.com/learn/articles/data-visualization, [mrežno; stranica posjećena: travanj 2024.].

[5] React Documentation, https://react.dev/learn, [mrežno; stranica posjećena: travanj 2024.].

[6] A. Shvets, *Dive Into Design Patterns*. Refactoring.Guru, 2019.

Sažetak

Vizualizacija algoritama sortiranja

Mateo Cindrić

U ovom radu daje se pregled algoritama sortiranja poput *Insertion Sort*, *Merge Sort*, *Heap Sort*, *Radix Sort* i dr. Prikazuje se analiza njihovih vremenskih i prostornih složenosti te se opisuju načini njihove primjene u stvarnom životu. Također se naglašava značaj vizualizacije algoritama sortiranja kao obrazovnog i istraživačkog alata. Vizualizacija pomaže u boljem razumijevanju rada algoritama, olakšavajući analizu njihovih performansi i primjenu u praktičnim scenarijima. Prezentira se implementacija aplikacije koja pomaže u vizualizaciji obrađenih algoritama sortiranja. Rad zaključuje da algoritmi sortiranja i njihova vizualizacija ne samo da poboljšavaju učinkovitost računalnih sustava, već i obogaćuju proces obrazovanja, omogućujući novim generacijama stjecanje dubljeg razumijevanja i inovativnih ideja u ovom području.

Ključne riječi: algoritmi sortiranja; vizualizacija; primjene algoritama sortiranja

Abstract

Visualization of sorting algorithms

Mateo Cindrić

This paper provides an overview of sorting algorithms such as *Insertion Sort*, *Merge Sort*, *Heap Sort*, *Radix Sort* and others. An analysis of their time and space complexities is presented, and ways of their application in real life are described. The importance of visualizing sorting algorithms as an educational and research tool is also emphasized. Visualization helps in better understanding the operation of algorithms, facilitating the analysis of their performance and application in practical scenarios. The implementation of the application which helps in the visualization of sorting algorithms is presented. The paper concludes that sorting algorithms and their visualization not only improve the efficiency of computer systems, they also enrich the education process, enabling new generations to gain a deeper understanding and innovative ideas in this field.

Keywords: sorting algorithms; visualization; applications of sorting algorithms