

Šolski center Celje  
Srednja šola za kemijo, elektrotehniko in računalništvo

# **PRIMERJAVA SORTIRNIH ALGORITMOV**

Računalništvo

Avtorji:

Jan JURHAR, R-2.b

Jakob MUŽAR, R-2.b

Matevž ZVIR, R-2.b

Mentor:

Bošjan LUBEJ, dipl. inž.

Mestna občina Celje, Mladi za Celje

Celje, 2024

## IZJAVA\*

Mentor Boštjan Lubej v skladu z 20. členom Pravilnika o organizaciji mladinske raziskovalne dejavnosti »Mladi za Celje« Mestne občine Celje, zagotavljam, da je v raziskovalni nalogi z naslovom Primerjava sortirnih algoritmov, katere avtorji so Jan Jurhar, Jakob Mužar in Matevž Zvir:

- besedilo v tiskani in elektronski obliki istovetno,
- pri raziskovanju uporabljeno gradivo navedeno v seznamu uporabljene literature,
- da je za objavo fotografij v nalogi pridobljeno avtorjevo dovoljenje in je hranjeno v šolskem arhivu,
- da sme Osrednja knjižnica Celje objaviti raziskovalno nalogo v polnem besedilu na knjižničnih portalih z navedbo, da je raziskovalna naloga nastala v okviru projekta Mladi za Celje,
- da je raziskovalno nalogo dovoljeno uporabiti za izobraževalne in raziskovalne namene s povzemanjem misli, idej, konceptov oziroma besedil iz naloge ob upoštevanju avtorstva in korektnem citiranju,
- da smo seznanjeni z razpisni pogoji projekta Mladi za Celje.

Celje, 4.4.2025



Podpis mentorja

Podpis odgovorne osebe

\*

## POJASNILO

V skladu z 20. členom Pravilnika raziskovalne dejavnosti »Mladi za Celje« Mestne občine Celje je potrebno podpisano izjavo mentorja (-ice) in odgovorne osebe šole vključiti v izvod za knjižnico, dovoljenje za objavo avtorja (-ice) fotografskega gradiva, katerega ni avtor (-ica) raziskovalne naloge, pa hrani šola v svojem arhivu.

## **Zahvala**

Zahvaljujemo se vsem, ki so kakorkoli pomagali, sodelovali pri izdelavi raziskovalne naloge. Veseli smo bili vsake ideje, nasveta, vzpodbudne besede, navsezadnje tudi kakšne kritike.

Zahvalili bi se predvsem mentorju prof. Boštjanu Lubeju za ves trud, čas, podporo in pomoč pri izdelavi raziskovalne naloge.

## Povzetek

V okviru naše raziskovalne naloge smo se osredotočili na analizo učinkovitosti različnih sortirnih algoritmov, da bi ugotovili, kateri so najprimernejši za uporabo v različnih scenarijih. Predvidevali smo, da bodo algoritmi z višjo kompleksnostjo pogosto hitrejši, obenem pa bodo imeli manjšo porabo pomnilnika, saj bodo bolje izkoriščali razpoložljive vire.

Za ta namen smo testirali šest različnih sortirnih algoritmov: Insertion Sort, Selection Sort, Heap Sort, Merge Sort, Quick Sort in Radix Sort. Vsak izmed teh algoritmov smo preučili tako z vidika časovne zahtevnosti kot tudi porabe pomnilnika, saj so to ključni dejavniki pri izbiri najučinkovitejšega algoritma za določeno nalogo. Testiranje je bilo izvedeno v različnih scenarijih, vključno z obdelavo velikih količin podatkov ter različnimi vrstami vhodnih podatkov, kar omogoča boljšo oceno delovanja algoritmov v praksi.

Poleg tega smo delovanje vizualizirali, kar nam je omogočilo boljše predstav, kako deluje določen algoritem.

**Ključne besede:** sortirni algoritmi, učinkovitost, časovna zahtevnost, poraba pomnilnika, vizualizacija, analiza algoritmov, primerjava algoritmov.

## Abstract

In our research paper, we focused on analyzing the efficiency of various sorting algorithms to determine which are most suitable for different scenarios. We hypothesized that algorithms with higher complexity would often be faster while also requiring less memory, as they would utilize available resources more effectively.

For this purpose, we tested six different sorting algorithms: Insertion Sort, Selection Sort, Heap Sort, Merge Sort, Quick Sort, and Radix Sort. Each of these algorithms was examined in terms of time complexity and memory consumption, as these are key factors in selecting the most efficient algorithm for a given task. The testing was conducted in various scenarios, including processing large datasets and different types of input data, allowing for a more comprehensive evaluation of the algorithms' performance in practice.

Additionally, we visualized the operation of these algorithms, which provided a clearer understanding of how each algorithm functions.

**Keywords:** sorting algorithms, efficiency, time complexity, memory consumption, visualization, algorithm analysis, algorithm comparison.

# KAZALO VSEBINE

1	UVOD .....	1
1.1	Oprelitev problema .....	2
1.2	Hipoteza in cilji .....	2
1.3	Metode raziskovanja .....	3
2	UPORABLJENE TEHNOLOGIJE .....	4
2.1	Microsoft Visual Studio.....	4
2.2	Microsoft Visual Studio Code .....	4
2.3	Python.....	4
2.4	NumPy.....	5
2.5	Matplotlib .....	5
3	NAČRTOVANJE IN IZVEDBA .....	6
4	OPIS SORTIRNIH ALGORITMOV .....	7
4.1	Insertion Sort .....	7
4.2	Selection Sort .....	7
4.3	Quick Sort .....	7
4.4	Merge Sort.....	8
4.5	Heap Sort.....	8
4.6	Radix Sort.....	9
5	PREDSTAVITEV IN REZULTATI TESTIRANJA .....	10
5.1	Predstavitev testiranja .....	10
5.2	Merjenje in analiza .....	10
5.3	Vizualizacija .....	10
5.4	Rezultati testiranja prikazani v tabeli .....	14
5.4.1	Naključni podatki .....	14
5.4.2	Delno sortirani podatki .....	14
5.4.3	Obrnjeno sortirani podatki .....	15
5.4.4	Poraba pomnilnika (obrnjeno sortirani podatki) .....	15
5.5	Ugotovitve .....	16
5.5.1	Časi izvajanja algoritmov .....	16
5.5.2	Vpliv vrste podatkov na čas izvajanja.....	16
5.5.3	Poraba pomnilnika.....	16

5.6	Primerna uporaba .....	17
5.6.1	Insertion Sort .....	17
5.6.2	Selection Sort .....	17
5.6.3	Quick Sort .....	17
5.6.4	Merge Sort.....	17
5.6.5	Heap Sort.....	17
5.6.6	Radix Sort.....	17
6	ANALIZA IN PREVERJANJE ZASTAVLJENIH HIPOTEZ .....	18
7	ZAKLJUČEK.....	19
8	VIRI.....	20
9	PRILOGE .....	21
9.1	Insertion Sort .....	21
9.2	Selection Sort .....	21
9.3	Quick Sort .....	22
9.4	Merge Sort.....	22
9.5	Heap Sort.....	23
9.6	Radix Sort.....	23
9.7	Glavni program .....	24

## KAZALO SLIK

Slika 1: Logo Visual Studio.....	4
Slika 2: Logo Visual Studio Code .....	4
Slika 3: Logo Python.....	4
Slika 4: Logo NumPy .....	5
Slika 5: Logo Matplotlib .....	5
Slika 6: Začetni meni programa za vizualizacijo brez nastavljenih parametrov .....	11
Slika 7: Meni programa za vizualizacijo z parametrom količine testirnih podatkov .....	11
Slika 8: Meni za izbiro sortirnega algoritma .....	12
Slika 9: Gumb za začetek .....	12
Slika 10: Vizualizacija sortirnega algoritma v teku.....	13
Slika 11: Vizualizacija programa po končanem sortiranju.....	13
Slika 12: Tabela čas izvajanja algoritmov na naključnih podatkih .....	14
Slika 13: Tabela čas izvajanja algoritmov na delno sortiranih podatkih .....	14
Slika 14: Tabela čas izvajanja algoritmov na obrnjeno sortiranih podatkih.....	15
Slika 15: Tabela porabe pomnilnika pri izvajanju algoritmov na obrnjeno sortiranih podatkih .....	16

# 1 UVOD

Sortiranje podatkov je ena najpogosteje uporabljenih operacij v računalništvu, saj se pojavlja v številnih aplikacijah, kot so iskanje informacij, obdelava podatkov in optimizacija delovanja sistemov. Zaradi tega je učinkovitost sortirnih algoritmov ključnega pomena, saj lahko izbira prave metode bistveno vpliva na časovno in prostorsko zahtevnost računalniških operacij.

V tej raziskovalni nalogi smo se odločili preučiti različne sortirne algoritme z namenom analize njihove učinkovitosti v različnih pogojih. Osredotočili smo se na klasične algoritme, kot so Insertion Sort, Selection Sort, Heap Sort, Merge Sort, Quick Sort in Radix Sort. Zanimalo nas je, kako se ti algoritmi obnašajo pri različnih vrstah vhodnih podatkov, vključno s popolnoma naključnimi, delno urejenimi in obrnjenimi urejenimi podatki. Prav tako smo želeli preizkusiti njihovo zmogljivost pri različnih velikostih vhodnih podatkov, od 1.000 do 1.000.000 elementov.

Razumevanje učinkovitosti teh algoritmov je ključno za optimizacijo programske opreme in izboljšanje hitrosti procesiranja podatkov. S pomočjo meritev časa izvajanja in porabe pomnilnika smo želeli pridobiti vpogled v prednosti in slabosti posameznih metod ter podati priporočila za njihovo uporabo v praksi. Poleg tega smo primerjali algoritme, implementirane v različnih programskih jezikih, predvsem v Pythonu in C, saj vsak izmed njiju ponuja različne prednosti glede hitrosti izvajanja in porabe sistemskih virov.

Naša raziskava temelji na eksperimentalnem pristopu, kjer smo izvajali številne teste in primerjave. Z uporabo različnih vhodnih podatkov in spremljanjem ključnih metrik smo želeli pridobiti jasen vpogled v to, kateri algoritmi so najučinkovitejši v določenih scenarijih. Na podlagi rezultatov bomo lahko podali smernice za izbiro najboljšega algoritma glede na potrebe uporabnika in lastnosti podatkov, ki jih je treba obdelati.

## 1.1 Opredelitev problema

Vsakdo se je že srečal s potrebo po hitrem in učinkovitem razvrščanju podatkov, bodisi pri upravljanju velikih zbirk podatkov, iskanju informacij ali optimizaciji delovanja računalniških programov. Sortiranje je ključen del številnih algoritmov in podatkovnih struktur, vendar izbira pravega algoritma ni vedno preprosta.

Obstaja več različnih sortirnih algoritmov, ki se razlikujejo po hitrosti, porabi pomnilnika in učinkovitosti glede na vrsto vhodnih podatkov. Nekateri algoritmi so primerni za majhne in že delno urejene nize, drugi pa so optimizirani za velike podatkovne množice. Nepravilna izbira algoritma lahko povzroči nepotrebno porabo virov, upočasnjuje delovanje programov ali celo oteži obdelavo podatkov.

## 1.2 Hipoteza in cilji

Cilj raziskovalne naloge je analizirati učinkovitost različnih sortirnih algoritmov in določiti, kateri so najprimernejši za uporabo v različnih scenarijih. V raziskavi smo si postavili naslednje hipoteze:

H1: Algoritmi z višjo kompleksnostjo bodo hitrejši pri obdelavi večjih količin podatkov, saj bodo bolje izkoriščali razpoložljive računalniške vire in se bolje obvladovali pri obdelavi velikih podatkovnih naborov.

H2: Algoritmi z nizko kompleksnostjo bodo učinkovitejši pri manjših in že delno urejenih podatkih, saj so manj zahtevni glede časa in porabe pomnilnika, kar omogoča hitrejše izvajanje.

H3: Različne vrste vhodnih podatkov, kot so naključno razporejeni, že delno urejeni ali obratno urejeni podatki, bodo vplivale na učinkovitost sortirnih algoritmov, saj so nekateri algoritmi optimizirani za specifične vrste podatkov.

### 1.3 Metode raziskovanja

Za raziskovalno metodo smo uporabili eksperimentalno testiranje različnih sortirnih algoritmov, da bi ocenili njihovo učinkovitost v različnih scenarijih. Pri tem smo se osredotočili na analizo časovne zahtevnosti in porabe pomnilnika za vsak algoritem, pri čemer smo upoštevali različne vrste vhodnih podatkov.

V raziskavi smo testirali šest različnih sortirnih algoritmov: Insertion Sort, Selection Sort, Heap Sort, Merge Sort, Quick Sort in Radix Sort. Testiranja so bila izvedena na različnih velikostih in vrstah podatkovnih nizov, vključno z naključno razporejenimi podatki, že delno urejenimi podatki in obratno urejenimi podatki, da bi bolje razumeli vpliv vhodnih podatkov na delovanje algoritmov.

Vsak algoritem smo testirali večkrat, da smo zagotovili natančne in ponovljive rezultate. Za analizo rezultatov smo merili čas, potreben za izvedbo sortiranja, in porabo pomnilnika v različnih pogojih.

Rezultate testiranj smo uporabili za preverjanje naših hipotez o učinkovitosti različnih algoritmov. Na podlagi teh rezultatov smo ugotovili, kateri algoritmi so najbolj primerni za določene vrste nalog in vhodnih podatkov.

## 2 UPORABLJENE TEHNOLOGIJE

### 2.1 Microsoft Visual Studio

Microsoft Visual Studio je integrirano razvojno okolje, ki omogoča razvijanje aplikacij v različnih programskih jezikih, vključno s Pythonom. Uporabljeno je bilo za implementacijo in testiranje sortirnih algoritmov ter optimizacijo kode. Ponuja orodja za analizo učinkovitosti, kar je bilo ključno pri testiranju algoritmov.

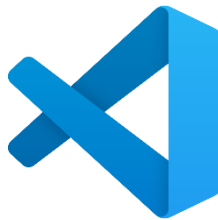


Slika 1: Logo Visual Studio

[https://upload.wikimedia.org/wikipedia/commons/2/2c/Visual\\_Studio\\_Icon\\_2022.svg](https://upload.wikimedia.org/wikipedia/commons/2/2c/Visual_Studio_Icon_2022.svg)

### 2.2 Microsoft Visual Studio Code

VS Code je odprtokodni urejevalnik kode, ki podpira številne programske jezike, vključno s Python. Uporabljali smo vtičnike za Python, ki so omogočili enostavno izvedbo in testiranje algoritmov ter integracijo z matplotlib za vizualizacijo rezultatov.



Slika 2: Logo Visual Studio Code

[https://upload.wikimedia.org/wikipedia/commons/9/9a/Visual\\_Studio\\_Code\\_1.35\\_icon.svg](https://upload.wikimedia.org/wikipedia/commons/9/9a/Visual_Studio_Code_1.35_icon.svg)

### 2.3 Python

Python je bil glavni programski jezik za implementacijo sortirnih algoritmov. Zaradi svoje preprostosti in zmogljivosti omogoča hitro prototipiranje. Za merjenje časa izvajanja smo uporabili time, za vizualizacijo pa matplotlib.



Slika 3: Logo Python

<https://upload.wikimedia.org/wikipedia/commons/c/c3/Python-logo-notext.svg>

## 2.4 NumPy

NumPy je ključna knjižnica za učinkovito manipulacijo z velikimi podatki. Omogoča hitro obdelavo podatkov in je bila uporabljena za manipulacijo podatkovnih nizov pri testiranju algoritmov.

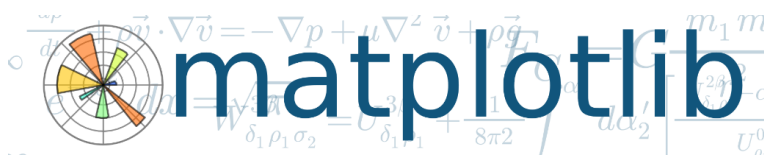


Slika 4: Logo NumPy

[https://upload.wikimedia.org/wikipedia/commons/3/31/NumPy\\_logo\\_2020.svg](https://upload.wikimedia.org/wikipedia/commons/3/31/NumPy_logo_2020.svg)

## 2.5 Matplotlib

Matplotlib je knjižnica za vizualizacijo podatkov, ki je bila uporabljena za prikaz rezultatov algoritmov v grafični obliki, kot so črtni in stolpčni grafi. Pomagala je pri analizi in primerjavi učinkovitosti različnih algoritmov.



Slika 5: Logo Matplotlib

[https://upload.wikimedia.org/wikipedia/en/5/56/Matplotlib\\_logo.svg](https://upload.wikimedia.org/wikipedia/en/5/56/Matplotlib_logo.svg)

### 3 NAČRTOVANJE IN IZVEDBA

Pri izvedbi raziskave o učinkovitosti sortirnih algoritmov smo si zadali visoke cilje, saj smo želeli analizirati in primerjati različne algoritme z vidika časovne zahtevnosti in porabe pomnilnika. Veliko časa smo namenili načrtovanju in pripravi, saj smo se zavedali, da je skrbno načrtovanje ključnega pomena za natančnost in uspešnost testiranja.

Na začetku smo se osredotočili na izbiro sortirnih algoritmov, ki jih bomo preučili. Izbrali smo šest najbolj razširjenih algoritmov: Insertion Sort, Selection Sort, Heap Sort, Merge Sort, Quick Sort in Radix Sort. Te algoritme smo implementirali v programskem jeziku Python, saj omogoča hitro prototipiranje in enostavno testiranje.

Ko smo imeli implementirane algoritme, smo se osredotočili na izvedbo testiranja. Za analizo časovne zahtevnosti smo uporabljali knjižnico `time`, ki nam je omogočila merjenje časa izvajanja za različne velikosti vhodnih podatkov. Testirali smo algoritme na naključnih nizih, urejenih nizih in obrnjenih nizih, da bi preučili, kako algoritmi delujejo v različnih scenarijih.

Za vizualizacijo delovanja algoritmov smo uporabili `matplotlib`. Zasnovali smo animacijo, ki prikazuje, kako se elementi niza premikajo med sortiranjem, kar uporabnikom omogoči, da bolje razumejo, kako vsak algoritem deluje. Ta vizualizacija je bila izdelana v Pythonu in je omogočila lažjo analizo ter primerjavo različnih algoritmov.

Uporabljeni programi in tehnologije so omogočili enostavno izvajanje testov in analiziranje rezultatov. Tako smo lahko hitro preverili, kako se algoritmi obnašajo v različnih pogojih in s tem pridobili natančne podatke o njihovi učinkovitosti.

## 4 OPIS SORTIRNIH ALGORITMOV

### 4.1 Insertion Sort

Insertion Sort deluje tako, da gradi urejeni del seznama postopoma. Na začetku se obravnava le prvi element, saj je sam po sebi urejen. Nato se vsak naslednji element vstavi na ustrezno mesto v že urejenem delu.

Pri vsakem novem elementu se preveri, ali je manjši od prejšnjih. Če je, se premakne nazaj, dokler ne doseže pravilne pozicije. Ta postopek se ponavlja za vse elemente, dokler ni celoten seznam urejen.

Časovna kompleksnost:

Najboljši primer:  $O(n)$  (če so podatki že sortirani)

Povprečni primer:  $O(n^2)$

Najslabši primer:  $O(n^2)$

### 4.2 Selection Sort

Selection Sort deluje tako, da večkrat preleti seznam in poišče najmanjši element, nato pa ga zamenja s prvim neurejenim elementom. Postopek se ponavlja, dokler ni celoten seznam urejen.

Najprej se pregleda celoten seznam in poišče najmanjši element. Ta se nato zamenja z elementom na prvem mestu. Nato se pregleda preostanek seznama in poišče drugi najmanjši element, ki se zamenja z drugim elementom. Tako se nadaljuje, dokler niso vsi elementi na pravih mestih.

Časovna kompleksnost:

Najboljši primer:  $O(n^2)$

Povprečni primer:  $O(n^2)$

Najslabši primer:  $O(n^2)$

### 4.3 Quick Sort

Quick Sort temelji na načelu delitve in urejanja (divide and conquer). Najprej se izbere tako imenovani pivot element, ki služi kot referenčna točka za razdelitev seznama na dva dela.

V levem delu so elementi, ki so manjši od pivota, v desnem delu pa elementi, ki so večji. Nato se ista metoda rekurzivno uporablja na obeh delih, dokler niso vsi deli popolnoma urejeni. Ko so vsi elementi razvrščeni, se deli združijo v končni urejen seznam.

Časovna kompleksnost:

Najboljši primer:  $O(n \log n)$

Povprečni primer:  $O(n \log n)$

Najslabši primer:  $O(n^2)$  (če je pivot izbran slabo)

#### 4.4 Merge Sort

Merge Sort je rekurziven algoritem, ki najprej seznam razdeli na dva enaka dela, dokler ne ostanejo posamezni elementi. Ko so vsi deli razdeljeni, se začne postopek združevanja (merge), pri katerem se elementi primerjajo in razporejajo v urejenem vrstnem redu.

Pri združevanju se vedno primerja najmanjši element iz vsakega dela in se doda v novi seznam. Ta postopek se nadaljuje, dokler ni celoten seznam urejen. Zaradi svoje zanesljivosti in učinkovitosti se Merge Sort pogosto uporablja pri delu z velikimi podatkovnimi nizi.

Časovna kompleksnost:

Najboljši primer:  $O(n \log n)$

Povprečni primer:  $O(n \log n)$

Najslabši primer:  $O(n \log n)$

#### 4.5 Heap Sort

Heap Sort uporablja strukturo binarne kopice (heap), ki zagotavlja, da je največji element vedno na vrhu. Ta algoritem najprej ustvari največjo kopico (max heap), kjer je največji element postavljen na prvo mesto.

Ko je kopica vzpostavljena, se največji element zamenja z zadnjim elementom v seznamu, nato pa se kopica prilagodi in postopek se ponovi. Ta metoda zagotavlja, da se največji elementi postopoma premikajo na svoja pravilna mesta, dokler ni celoten seznam urejen.

Časovna kompleksnost:

Najboljši primer:  $O(n \log n)$

Povprečni primer:  $O(n \log n)$

Najslabši primer:  $O(n \log n)$

## 4.6 Radix Sort

Radix Sort je posebna metoda sortiranja, ki razvršča števila glede na njihove posamezne števke, od najmanjšega do največjega reda. Najprej se vsi elementi razvrstijo glede na enice, nato glede na desetice, stotice itd., dokler ni seznam popolnoma urejen.

Za razvrščanje po posameznih števkih se pogosto uporablja stabilna sortirna metoda, kot je Counting Sort. Ta pristop omogoča hitro in učinkovito razvrščanje velikih podatkovnih nizov, še posebej kadar gre za cela števila s fiksno dolžino.

Časovna kompleksnost:

Najboljši primer:  $O(nk)$

Povprečni primer:  $O(nk)$

Najslabši primer:  $O(nk)$

## 5 PREDSTAVITEV IN REZULTATI TESTIRANJA

### 5.1 Predstavitev testiranja

V okviru raziskave smo izvedli eksperimentalno testiranje šestih različnih sortirnih algoritmov na različnih vrstah vhodnih podatkov. Testiranja so bila izvedena z uporabo programskega jezika Python in knjižnic, kot so NumPy za obdelavo podatkov in Matplotlib za vizualizacijo rezultatov.

Za vsak algoritem smo preučili časovno zahtevnost in porabo pomnilnika pri različnih velikostih podatkovnih nizov. Vhodni podatki so bili izbrani tako, da so odražali različne scenarije, s katerimi se algoritmi lahko srečajo v praksi, kot so naključno razporejeni, že delno urejeni in obratno urejeni podatki.

**Testiranje je bilo izvedeno v naslednjih pogojih:**

**Naključno razporejeni podatki:** Seznami so bili ustvarjeni z naključnim razporejanjem števil, da bi preučili, kako se algoritmi obnašajo v povsem neurejenih podatkih.

**Delno urejeni podatki:** Seznami so bili delno urejeni, kar pomeni, da so bili nekateri elementi že na svojih mestih, da bi preverili, kako algoritmi obvladujejo že delno urejene podatke.

**Obratno urejeni podatki:** V tem scenariju so bili podatki obrnjenega vrstnega reda, da bi ugotovili, kako se algoritmi prilagajajo najbolj neugodnemu primeru.

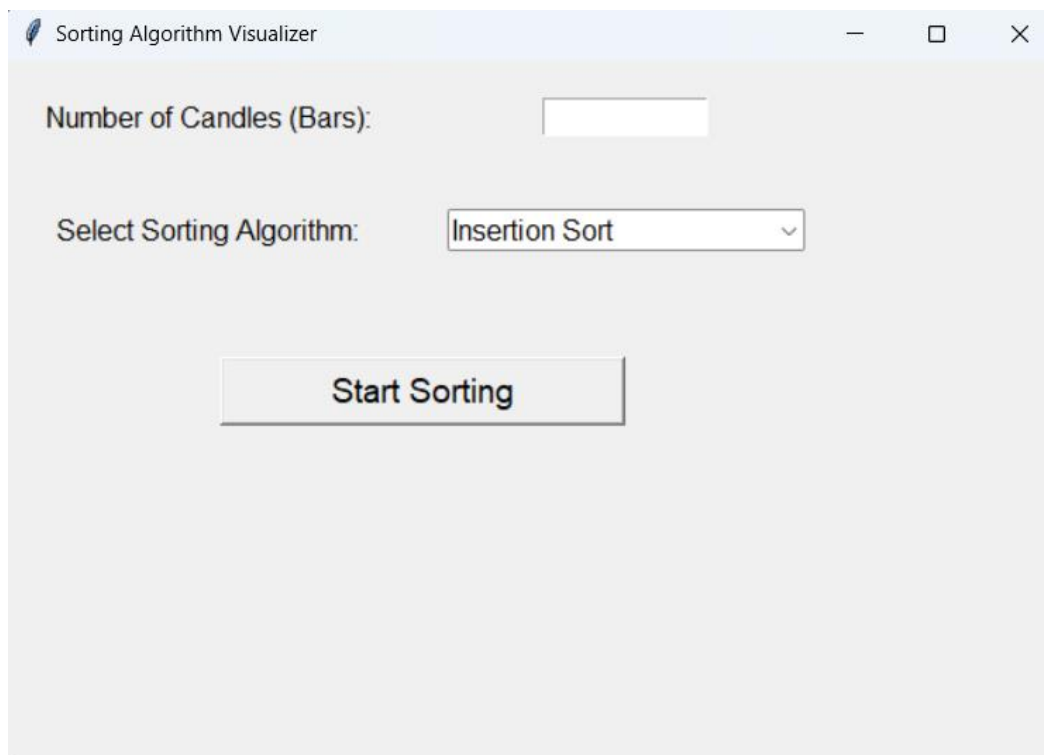
### 5.2 Merjenje in analiza

Za merjenje časovne zahtevnosti smo uporabili knjižnico time, ki je omogočila merjenje časa, ki je potreben za izvedbo sortiranja. Vsak algoritem smo testirali na različnih velikostih podatkovnih nizov (od 100 do 1.000.000 elementov), pri čemer smo izvedli več ponovitev, da smo zagotovili natančne in ponovljive rezultate.

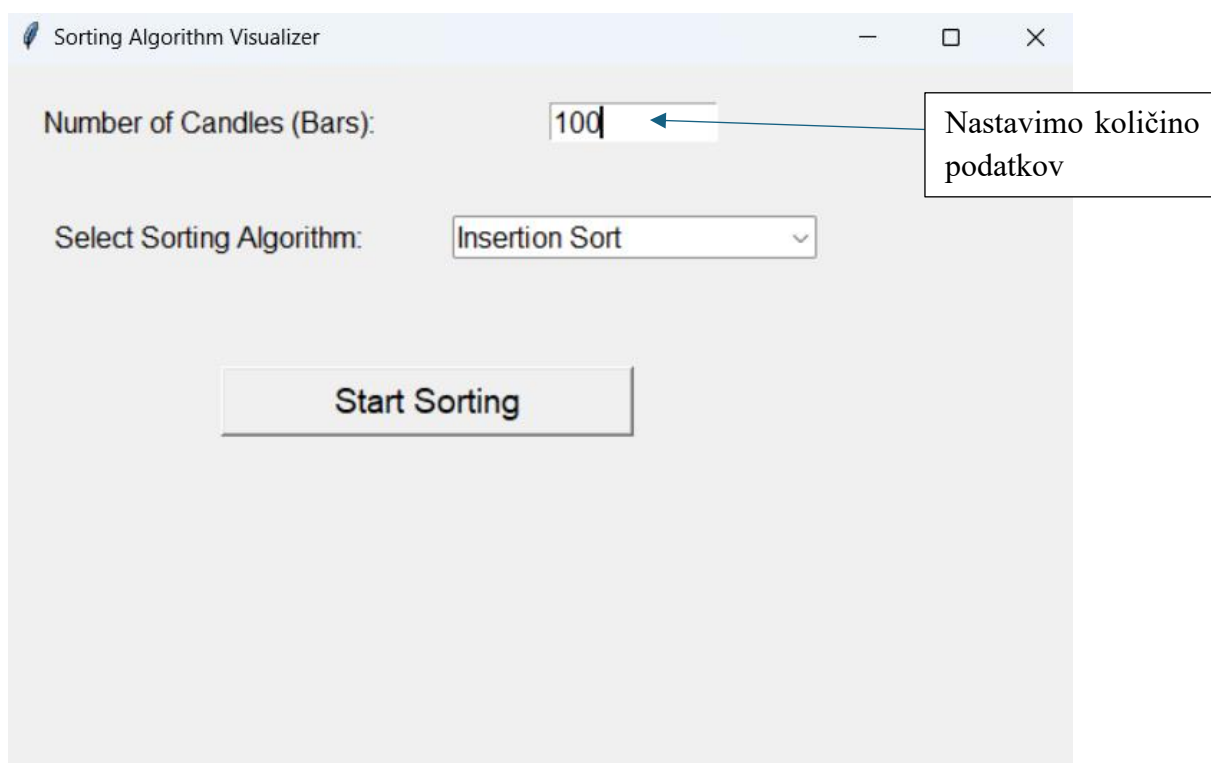
Poleg tega smo analizirali porabo pomnilnika, pri čemer smo za vsak algoritem spremljali, koliko pomnilnika je bil porabljen v različnih fazah izvajanja. To nam je omogočilo, da smo primerjali učinkovitost algoritmov tudi glede na porabo virov.

### 5.3 Vizualizacija

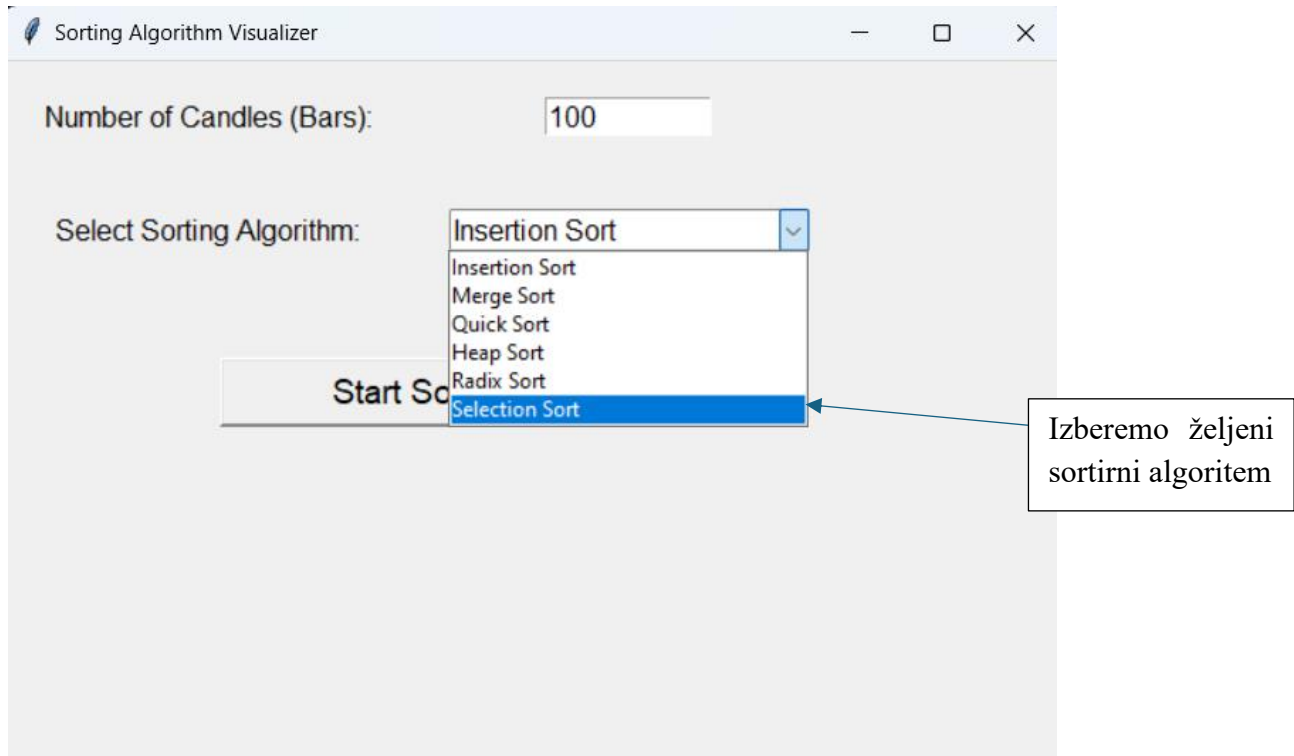
Za boljšo predstavo o tem, kako vsak algoritem deluje, smo izdelali animacijo s pomočjo knjižnice Matplotlib. Vizualizacija prikazuje korake, ki jih algoritmi izvajajo pri sortiranju, in omogoča boljši vpogled v dinamiko premikanja elementov med postopkom.



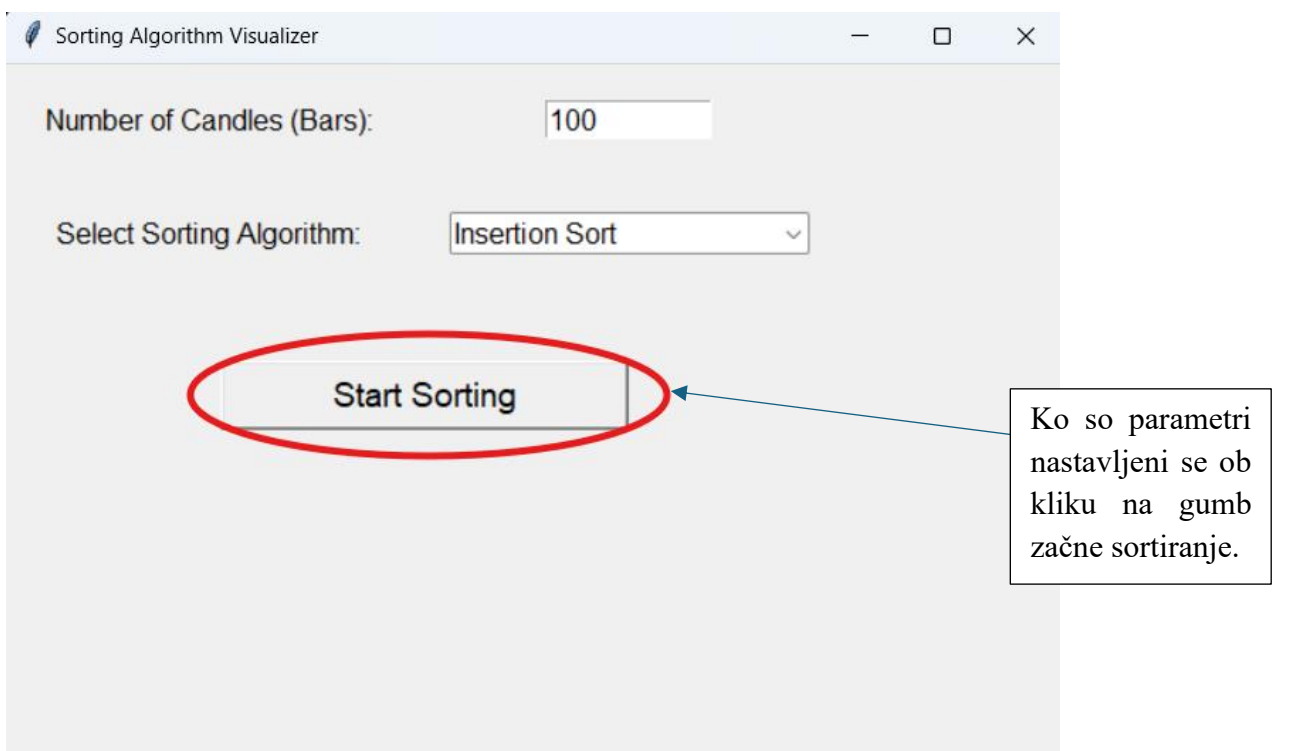
Slika 6: Začetni meni programa za vizualizacijo brez nastavljenih parametrov



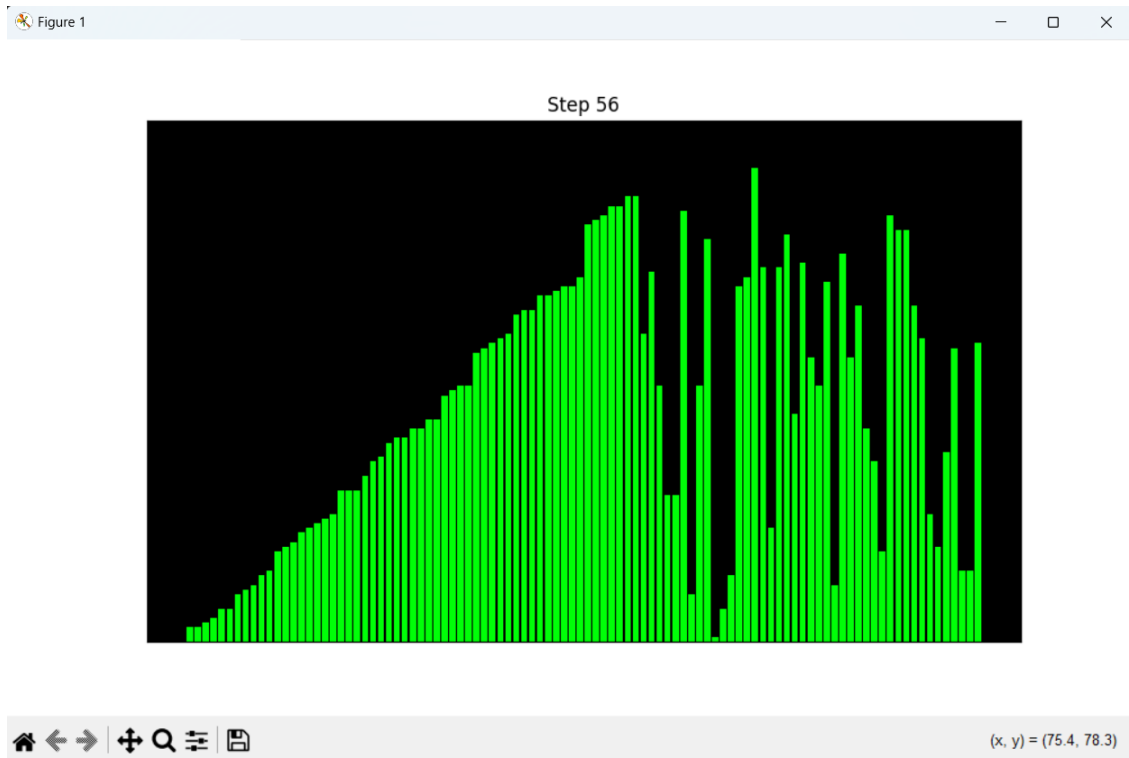
Slika 7: Meni programa za vizualizacijo z parametrom količine testirnih podatkov



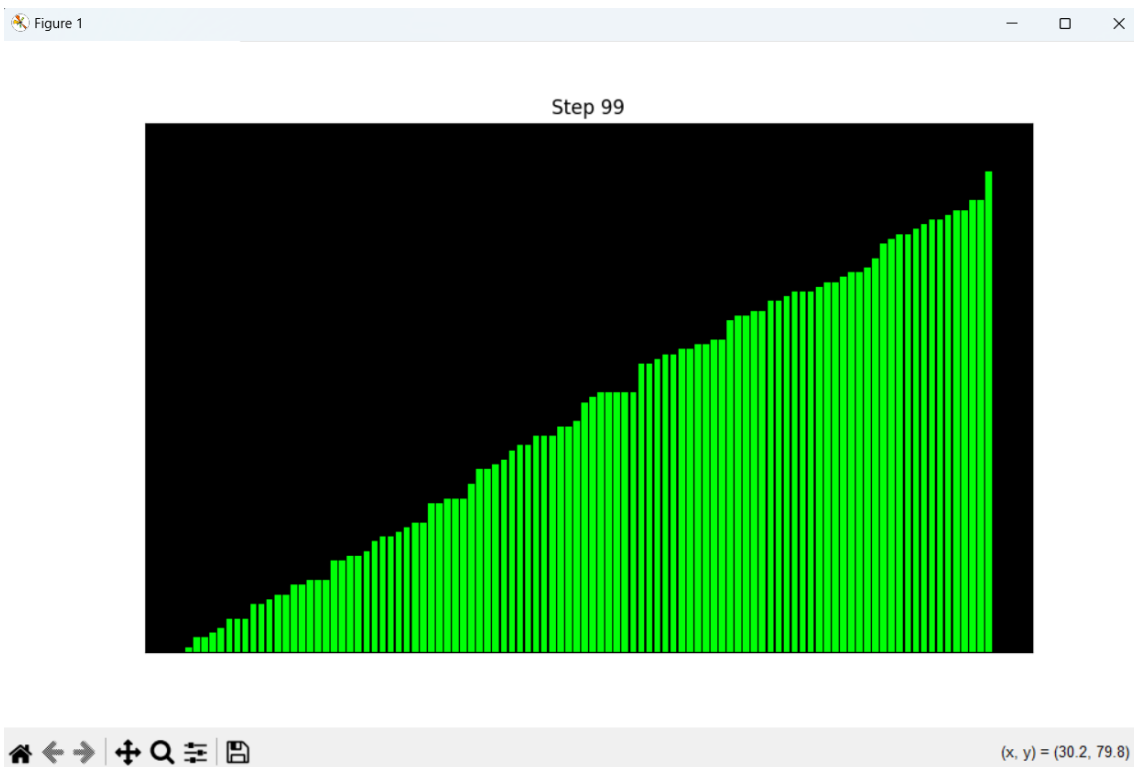
Slika 8: Meni za izbiro sortirnega algoritma



Slika 9: Gumb za začetek



Slika 10: Vizualizacija sortirnega algoritma v teku



Slika 11: Vizualizacija programa po končanem sortiranju

## 5.4 Rezultati testiranja prikazani v tabeli

### 5.4.1 Naključni podatki

Quick Sort in Merge Sort sta se izkazala za najhitrejša algoritma z najnižjimi časi izvajanja, tako pri manjših kot večjih podatkih. Za 1000 elementov so bili časi izvajanja le 0,0041 sekunde pri Quick Sortu in 0,0055 sekunde pri Merge Sortu. Ko smo prešli na večje podatke (1000000 elementov), so časi ostali razmeroma nizki, s Quick Sortom, ki je dosegel čas 10,1081 sekunde, in Merge Sortom 10,8624 sekunde.

Heap Sort je bil še vedno hiter, vendar so bili njegovi časi izvajanja nekoliko višji kot pri Quick Sortu in Merge Sortu. Za 1000000 elementov je dosegel čas 22,0834 sekunde. Insertion Sort in Selection Sort sta bila izrazito počasna, še posebej pri večjih podatkih. Za 1000000 elementov sta bila časi izvajanja za oba algoritma precej visoka, kar pomeni, da sta za takšne podatke manj primerna.

Naključni podatki				
	Čas [s]			
Velikost	1000	10000	100000	1000000
Insertion sort	0,0948	9,6011	/	/
Selection sort	0,1175	11,9441	/	/
Heap sort	0,0092	0,1337	1,1798	22,0834
Merge sort	0,0055	0,0791	0,9289	10,8624
Quick sort	0,0041	0,0666	0,8241	10,1081
Radix sort	0,0075	0,0912	0,9562	11,1188

Slika 12: Tabela čas izvajanja algoritmov na naključnih podatkih

### 5.4.2 Delno sortirani podatki

Quick Sort in Merge Sort sta še vedno pokazala najboljše rezultate, vendar so bili časi izvajanja nekoliko višji kot pri naključnih podatkih. Na primer, pri 1000000 elementih je Quick Sort dosegel čas 13,2196 sekunde, medtem ko je Merge Sort potreboval 9,4461 sekunde. Heap Sort je ponovno bil učinkovit, čeprav so se časi nekoliko povečali v primerjavi z naključnimi podatki. Insertion Sort in Selection Sort sta bila še vedno počasna, vendar so bili časi nekaj nižji kot pri naključnih podatkih, saj so bili podatki delno sortirani, kar je pripomoglo k izboljšanju njihove učinkovitosti.

Delno sortirani podatki				
	Čas [s]			
Velikost	1000	10000	100000	1000000
Insertion sort	0,0504	4,9706	/	/
Selection sort	0,1206	11,6727	/	/
Heap sort	0,0095	0,1471	1,8307	22,1125
Merge sort	0,0054	0,0652	0,7938	9,4461
Quick sort	0,0047	0,0774	1,0441	13,2196
Radix sort	0,0075	0,0792	1,1285	11,9662

Slika 13: Tabela čas izvajanja algoritmov na delno sortiranih podatkih

### 5.4.3 Obrnjeno sortirani podatki

Pri obrnjeno sortiranih podatkih so rezultati deloma sovpadali z rezultati za delno sortirane podatke. Quick Sort in Merge Sort sta še vedno najhitrejša, čeprav so bili časi nekoliko višji. Quick Sort je pri 1000000 elementih dosegel čas 10,7967 sekunde, medtem ko je Merge Sort dosegel čas 9,7606 sekunde. Heap Sort je bil nekoliko počasnejši kot pri drugih vrstah podatkov, vendar je bil še vedno primeren za večje podatke. Insertion Sort in Selection Sort sta bili zopet počasna, kar kaže, da te vrste podatkov niso primerne za te algoritme.

Obrnjeno sortirani podatki				
	Čas [s]			
Velikost	1000	10000	100000	1000000
Insertion sort	0,2119	21,735	/	/
Selection sort	0,1219	11,985	/	/
Heap sort	0,0104	0,1415	1,8321	22,4312
Merge sort	0,0051	0,0722	0,8148	9,7606
Quick sort	0,0056	0,0767	0,9683	10,7967
Radix sort	0,0057	0,0856	1,1282	13,2505

Slika 14: Tabela čas izvajanja algoritmov na obrnjeno sortiranih podatkih

### 5.4.4 Poraba pomnilnika (obrnjeno sortirani podatki)

Pri obdelavi obrnjeno sortiranih podatkov se poraba pomnilnika med algoritmi ni dramatično spreminjala, vendar so obstajale nekatere razlike, zlasti pri večjih podatkih. Razčlenimo porabo pomnilnika glede na velikost podatkov:

Quick Sort je imel najvišjo porabo pomnilnika pri večjih podatkih. Pri 1000 elementih je bila poraba pomnilnika 117,19 MB, vendar je pri 1000000 elementih narasla na 148,55 MB, kar pomeni, da se je poraba pomnilnika s povečanjem velikosti podatkov precej povečala. To je posledica njegovih lastnosti, kjer uporablja rekurzivne klice in potrebne dodatne pomnilniške alokacije.

Merge Sort je prav tako pokazal rast porabe pomnilnika z večjo velikostjo podatkov. Za 1000 elementov je poraba znašala 117,25 MB, pri 1000000 elementih pa je narasla na 139,94 MB. Podobno kot Quick Sort, Merge Sort za obdelavo večjih podatkov zahteva dodatni pomnilnik zaradi rekurzije in kopiranja pod nizov.

Heap Sort je pokazal nekoliko manjšo rast porabe pomnilnika v primerjavi z Quick Sort in Merge Sort. Za 1000 elementov je bila poraba pomnilnika 117,19 MB, pri 1000000 elementih pa je dosegla 124,94 MB. Heap Sort običajno uporablja le konstantno dodatno pomnilniško enoto, saj ne uporablja rekurzije ali dodatnih kopij podatkov, kar pomeni, da je bolj učinkovit z vidika pomnilnika.

Insertion Sort in Selection Sort sta imela najmanjšo porabo pomnilnika, ki se je gibala okoli 117 MB za vse velikosti podatkov. To je posledica dejstva, da ti algoritmi delujejo in manipulirajo s podatki znotraj obstoječe strukture (brez dodatnih alokacij pomnilnika), zato njihova poraba pomnilnika ostaja skoraj konstantna ne glede na velikost podatkov.

Radix Sort je imel porabo pomnilnika, ki se je nekoliko povečala z večanjem velikosti podatkov, vendar ni dosegla nivoja Quick Sorta ali Merge Sorta. Za 1000 elementov je bila poraba 118,66 MB, pri 1000000 elementih pa je narasla na 136,41 MB.

Obrnjeno sortirani podatki				
	Pomnilnik [MB]			
Velikost	1000	10000	100000	1000000
Insertion sort	117,09	117,18	/	/
Selection sort	116,78	117,08	/	/
Heap sort	117,19	117,20	118,24	124,94
Merge sort	117,25	117,45	119,23	139,94
Quick sort	117,19	117,81	121,35	148,55
Radix sort	118,66	118,86	121,00	136,41

Slika 15: Tabela porabe pomnilnika pri izvajanju algoritmov na obrnjeno sortiranih podatkih

## 5.5 Ugotovitve

### 5.5.1 Časi izvajanja algoritmov

Quick Sort in Merge Sort sta se izkazala za najhitrejša algoritma, tudi pri večjih podatkih. Pri 1000 elementih so časi znašali 0,0041 sekunde za Quick Sort in 0,0055 sekunde za Merge Sort, medtem ko pri 1000000 elementih Quick Sort potrebuje 10,1081 sekunde, Merge Sort pa 10,8624 sekunde. Heap Sort je bil nekoliko počasnejši (22,0834 sekunde za 1000000 elementov), medtem ko sta Insertion Sort in Selection Sort pri večjih podatkih pokazala zelo visoke čase, kar pomeni, da nista primerna za obdelavo večjih podatkov.

### 5.5.2 Vpliv vrste podatkov na čas izvajanja

Pri delno sortiranih podatkih so Quick Sort in Merge Sort še vedno najbolj učinkovita, vendar so časi nekoliko višji kot pri naključnih podatkih. Heap Sort je bil učinkovit, čeprav so se časi nekoliko povečali. Insertion Sort in Selection Sort sta bila še vedno počasna, vendar so se časi izboljšali zaradi delno sortiranih podatkov.

Pri obrnjeno sortiranih podatkih so bili časi za Quick Sort in Merge Sort nekoliko višji kot pri drugih vrstah podatkov, vendar so ostali najnižji. Heap Sort je bil nekoliko počasnejši, vendar še vedno primeren za večje podatke. Insertion Sort in Selection Sort sta bila zopet počasna.

### 5.5.3 Poraba pomnilnika

Pri obrnjeno sortiranih podatkih se je poraba pomnilnika povečala predvsem pri Quick Sortu in Merge Sortu, zlasti pri večjih podatkih. Quick Sort je pri 1000000 elementih dosegel 148,55 MB, Merge Sort pa 139,94 MB. Heap Sort je imel manjšo rast porabe pomnilnika, saj je narasla le na 124,94 MB pri 1000000 elementih. Insertion Sort in Selection Sort sta imela najmanjšo porabo pomnilnika, ki je ostala skoraj konstantna pri okoli 117 MB. Radix Sort je imel nekoliko višjo porabo pomnilnika, vendar ni dosegel ravni Quick Sorta ali Merge Sorta.

## 5.6 Primerna uporaba

Ko izbirate algoritem za sortiranje, je pomembno upoštevati velikost seznama, vrsto podatkov in porabo pomnilnika.

### 5.6.1 Insertion Sort

Odlična izbira za majhne sezname ali skoraj že urejene podatke. Algoritem deluje hitro pri manjših seznamih, saj pri skoraj urejenih podatkih zahteva zelo malo premikov. Je enostaven za implementacijo in ne potrebuje dodatnega pomnilnika, vendar pri večjih seznamih postane neučinkovit zaradi časovne kompleksnosti  $O(n^2)$ .

### 5.6.2 Selection Sort

Primeren za majhne sezname, kjer ni potrebe po dodatnem pomnilniku. Izbira najmanjši element in ga premesti na pravo mesto, vendar je zaradi časovne kompleksnosti  $O(n^2)$  neprimeren za večje količine podatkov. Kljub temu je preprost in zahteva minimalen pomnilnik.

### 5.6.3 Quick Sort

Ena izmed najbolj učinkovitih možnosti za večje in naključno razporejene sezname. Deluje hitro v večini primerov z  $O(n \log n)$  časovno kompleksnostjo, vendar se lahko upočasni pri že urejenih podatkih. Optimizacija pivota preprečuje upočasnitev v najslabših primerih.

### 5.6.4 Merge Sort

Izjemno stabilen algoritem, primeren za velike sezname. Ima konstanten čas delovanja  $O(n \log n)$  v vseh primerih, kar ga naredi zelo predvidljivega, vendar zahteva dodatni pomnilnik. Je priporočljiv za obsežna sortiranja in zunanje podatke, ki ne morejo biti shranjeni v pomnilniku.

### 5.6.5 Heap Sort

Koristen, kadar potrebujete stabilno časovno kompleksnost  $O(n \log n)$  in minimalno porabo pomnilnika. Primeren je za večje sezname, če hitrost ni ključna kot pri Quick Sort. Uporablja malo dodatnega pomnilnika, vendar ni najhitrejši med algoritmi.

### 5.6.6 Radix Sort

Izjemno hiter pri številčnih podatkih, še posebej pri omejenem razponu števil. Ne temelji na primerjavah, kar omogoča linearno časovno kompleksnost  $O(n)$  v določenih primerih. Ni primeren za nestrukturirane ali besedilne podatke, saj deluje le na številkah.

Za izbiro algoritma morate upoštevati velikost in vrsto podatkov ter potrebo po stabilnosti, minimalnem pomnilniku ali hitrem izvajanju.

## 6 ANALIZA IN PREVERJANJE ZASTAVLJENIH HIPOTEZ

**H1: Algoritmi z višjo kompleksnostjo bodo hitrejši pri obdelavi večjih količin podatkov, saj bodo bolje izkoriščali razpoložljive računalniške vire in se bolje obvladovali pri obdelavi velikih podatkovnih naborov.**

Ocenjeno: Zavrnjeno.

Razlaga: Ta hipoteza ni nujno pravilna. Algoritmi z višjo kompleksnostjo (npr.  $O(n^2)$  in višji) imajo lahko slabšo zmogljivost pri večjih količinah podatkov, saj zahtevajo več časa in virov. Običajno niso primerni za velike podatkovne nabore, saj se časovna zahtevnost povečuje s številom elementov. Čeprav se lahko optimizirajo z večjedrnimi procesorji, večja kompleksnost ne pomeni nujno hitrejšega delovanja. Algoritmi z nižjo kompleksnostjo, kot so  $O(n \log n)$  (npr. quicksort, mergesort), so pogosto učinkovitejši.

**H2: Algoritmi z nizko kompleksnostjo bodo učinkovitejši pri manjših in že delno urejenih podatkih, saj so manj zahtevni glede časa in porabe pomnilnika, kar omogoča hitrejše izvajanje.**

Ocenjeno: Potrjeno.

Razlaga: Ta hipoteza je smiselna. Algoritmi z nizko kompleksnostjo, kot so tisti, ki delujejo v  $O(n)$  ali  $O(\log n)$  časovnih okvirih (npr. insertion sort pri manjših, že delno urejenih podatkih), so pogosto učinkovitejši pri takih primerih. Preprosti algoritmi, kot je insertion sort, so hitrejši za manjše podatkovne nize, saj imajo manjšo potrebo po pomnilniku in nimajo visoke kompleksnosti algoritmov za večje podatke.

**H3: Različne vrste vhodnih podatkov, kot so naključno razporejeni, že delno urejeni ali obratno urejeni podatki, bodo vplivale na učinkovitost sortirnih algoritmov, saj so nekateri algoritmi optimizirani za specifične vrste podatkov.**

Ocenjeno: Potrjeno.

Razlaga: Ta hipoteza je resnična. Sortirni algoritmi imajo različne zmogljivosti glede na vhodne podatke. Na primer:

Quicksort je učinkovit pri naključnih podatkih, a lahko postane neučinkovit ( $O(n^2)$ ) pri že urejenih podatkih.

Merge sort in heapsort sta stabilna, ne glede na vhodne podatke, vendar porabita več pomnilnika.

Insertion sort je zelo učinkovit pri že delno urejenih podatkih, saj ima skoraj linearno časovno kompleksnost.

## 7 ZAKLJUČEK

V raziskovalni nalogi smo preučili učinkovitost različnih sortirnih algoritmov z vidika časovne zahtevnosti in porabe pomnilnika, ter izvedli obsežno testiranje na različnih vrstah vhodnih podatkov. Rezultati so pokazali, da so Quick Sort in Merge Sort najboljši algoritmi za obdelavo večjih podatkovnih nizov, saj sta se izkazala za najhitrejša, z minimalno časovno zahtevnostjo v povprečnih primerih. Heap Sort, čeprav nekoliko počasnejši, ostaja učinkovit, zlasti za večje podatke, medtem ko sta Insertion Sort in Selection Sort pokazala znatno počasnost pri večjih naborih podatkov in sta zato manj primerna za večje naloge.

Pri testiranju na različnih vrstah vhodnih podatkov (naključno razporejeni, delno sortirani in obrnjeni podatki) so se rezultati obdržali, saj sta Quick Sort in Merge Sort obdržala svojo učinkovitost v vseh scenarijih, medtem ko sta Insertion Sort in Selection Sort ostala počasna, ne glede na vhodne podatke. Analiza porabe pomnilnika je razkrila, da Quick Sort in Merge Sort zahtevata več pomnilnika pri večjih podatkih, medtem ko Heap Sort ohranja nizko porabo pomnilnika.

Na podlagi teh ugotovitev lahko zaključimo, da je izbira sortirnega algoritma odvisna od vrste podatkov, velikosti niza in pomnilniških zahtev. Quick Sort in Merge Sort sta priporočljiva za obdelavo večjih naborov podatkov, medtem ko sta Insertion Sort in Selection Sort primerna za manjše ali skoraj že sortirane sezname. V prihodnosti bi bilo koristno raziskati možnosti optimizacije teh algoritmov ter preučiti nove pristope za obvladovanje pomnilniških omejitev pri obdelavi zelo velikih podatkovnih nizov.

Ta raziskava je ponudila dragocene vpoglede v delovanje različnih sortirnih algoritmov, kar bo v pomoč pri izbiri najbolj primerne algoritma v različnih realnih scenarijih, kjer so ključni čas in učinkovitost uporabe virov.

## 8 VIRI

- [1] Selection Sort. (brez datuma). Pridobljeno iz YouTube:  
<https://www.youtube.com/watch?v=EwjnF7rFLNs>, nazadnje obiskano 5.3.2025, ob 15:00
- [2] Insertion Sort. (brez datuma). Pridobljeno iz YouTube:  
<https://www.youtube.com/watch?v=8mJ-OhcfpYg>, nazadnje obiskano 5.3.2025, ob 16:00
- [3] Quick Sort. (brez datuma). Pridobljeno iz YouTube:  
<https://www.youtube.com/watch?v=Vtckgz38QHs>, nazadnje obiskano 5.3.2025, ob 16:00
- [4] Merge Sort. (brez datuma). Pridobljeno iz YouTube:  
<https://www.youtube.com/watch?v=3j0SWDX4AtU>, nazadnje obiskano 5.3.2025, ob 17:00
- [5] Heap Sort. (brez datuma). Pridobljeno iz YouTube:  
<https://www.youtube.com/watch?v=H5kAcmGOn4Q>, nazadnje obiskano 5.3.2025, ob 16:00
- [6] Radix Sort. (brez datuma). Pridobljeno iz YouTube:  
<https://www.youtube.com/watch?v=mVRHvZF8xtg&t=183s>, nazadnje obiskano 5.3.2025, ob 15:00
- [7] Matplotlib Documentation. (brez datuma). Pridobljeno iz Matplotlib:  
<https://matplotlib.org/stable/index.html>, nazadnje obiskano 5.3.2025, ob 16:00
- [8] NumPy Documentation. (brez datuma). Pridobljeno iz NumPy:  
<https://numpy.org/doc/>, nazadnje obiskano 5.3.2025 ob 16:00

## 9 PRILOGE

### 9.1 Insertion Sort

```
def insertion_sort(arr):
    arr = arr.copy()
    for n in range(1, len(arr)):
        temp = arr[n]
        temp_index = n

        for j in range(n - 1, -1, -1):
            if arr[j] > temp:
                arr[j + 1] = arr[j]
                temp_index = j
            else:
                break

        arr[temp_index] = temp

    return arr
```

### 9.2 Selection Sort

```
def selection_sort(arr):
    arr = arr.copy()
    for n in range(len(arr)):
        min_index = n

        for i in range(n + 1, len(arr)):
            if arr[i] < arr[min_index]:
                min_index = i

        if min_index != n:
            arr[min_index], arr[n] = arr[n], arr[min_index]

    return arr
```

### 9.3 Quick Sort

```
def quick_sort(data, low, high):
    if low < high:
        pivot_index = partition(data, low, high)
        quick_sort(data, low, pivot_index - 1)
        quick_sort(data, pivot_index + 1, high)
    return data

def partition(data, low, high):
    pivot_index = random.randint(low, high)
    data[pivot_index], data[high] = data[high], data[pivot_index]
    pivot = data[high]
    i = low - 1

    for j in range(low, high):
        if data[j] < pivot:
            i += 1
            data[i], data[j] = data[j], data[i]

    data[i + 1], data[high] = data[high], data[i + 1]
    return i + 1
```

### 9.4 Merge Sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

    return arr
```

## 9.5 Heap Sort

```
def heap_sort(arr):
    def heapify(arr, n, i):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2

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

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

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    n = len(arr)

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

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

    return arr
```

## 9.6 Radix Sort

```
def radix_sort(arr):

    def counting_sort(arr, exp):
        n = len(arr)
        output = np.zeros(n, dtype=int)
        count = np.zeros(10, dtype=int)

        for i in range(n):
            index = (arr[i] // exp) % 10
            count[index] += 1

        for i in range(1, 10):
            count[i] += count[i - 1]

        for i in range(n - 1, -1, -1):
            index = (arr[i] // exp) % 10
            output[count[index] - 1] = arr[i]
            count[index] -= 1

        for i in range(n):
            arr[i] = output[i]

    return arr

max_val = np.max(arr)
exp = 1
while max_val // exp > 0:
    counting_sort(arr, exp)
    exp *= 10
return arr
```

## 9.7 Glavni program

```
def visualize_sorting(frame_data, ax):
    frame_num, arr_state = frame_data
    ax.clear()
    ax.bar(range(len(arr_state)), arr_state, color='#00ff08')
    ax.set_facecolor('black')
    ax.set_title(f"Step {frame_num + 1}", color='black')
    ax.set_xlabel("Index", color='white')
    ax.set_ylabel("Value", color='white')
    ax.set_ylim(0, max(arr_state) + 10)
    ax.tick_params(axis='x', colors='white')
    ax.tick_params(axis='y', colors='white')

def animate_algorithm(process, fig, ax):
    ani = FuncAnimation(fig, visualize_sorting, fargs=(ax,), frames=enumerate(process), interval=200, repeat=False, save_count=len(process))
    plt.show(block=True)
    return ani
```

```
def run_gui():
    def on_start_button_click():
        num_candles = int(entry_num_candles.get())
        unsorted_data = [random.randint(1, 100) for _ in range(num_candles)]
        selected_algorithm = algorithm_var.get()
        if selected_algorithm == "Insertion Sort":
            process = insertion_sort_process(unsorted_data)
        elif selected_algorithm == "Heap Sort":
            process = heap_sort_process(unsorted_data)
        elif selected_algorithm == "Merge Sort":
            process = merge_sort_process(unsorted_data)
        elif selected_algorithm == "Quick Sort":
            process = quick_sort_process(unsorted_data)
        elif selected_algorithm == "Radix Sort":
            process = radix_sort_process(unsorted_data)
        elif selected_algorithm == "Selection Sort":
            process = selection_sort_process(unsorted_data)
        fig, ax = plt.subplots(figsize=(10, 6))
        animate_algorithm(process, fig, ax)
```

```
root = tk.Tk()
root.title("Sorting Algorithm Visualizer")
root.geometry("600x400")
label_num_candles = tk.Label(root, text="Number of Candles (Bars):", font=("Arial", 12))
label_num_candles.grid(row=0, column=0, padx=20, pady=20)
entry_num_candles = tk.Entry(root, font=("Arial", 12), width=10)
entry_num_candles.grid(row=0, column=1, padx=20, pady=20)
label_algorithm = tk.Label(root, text="Select Sorting Algorithm:", font=("Arial", 12))
label_algorithm.grid(row=1, column=0, padx=20, pady=20)
algorithm_var = tk.StringVar()
algorithm_var.set("Insertion Sort")
algorithm_dropdown = ttk.Combobox(root, textvariable=algorithm_var, font=("Arial", 12), width=20)
algorithm_dropdown['values'] = ("Insertion Sort", "Merge Sort", "Quick Sort", "Heap Sort", "Radix Sort", "Selection Sort")
algorithm_dropdown.grid(row=1, column=1, padx=20, pady=20)
start_button = tk.Button(root, text="Start Sorting", command=on_start_button_click, font=("Arial", 14), width=20)
start_button.grid(row=2, column=0, columnspan=2, padx=20, pady=40)
root.mainloop()

run_gui()
```