# Algoritmi de Sortare Implementați în Python

## Contents



# 1 Sortări de Bază

## 1.1 Insertion Sort

Insertion Sort este un algoritm simplu de sortare care construiește lista finală sortată câte un element pe rând. Este mult mai puțin eficient pe liste mari decât algoritmii de sortare mai avansați, cum ar fi QuickSort, HeapSort sau merge sort.

# 2 Sortări Avansate

## 2.1 Heap Sort

Heap Sort este un algoritm de sortare bazat pe comparație. HeapSort poate fi văzut ca o variantă îmbunătățită a sortării prin selecție: la fel ca sortarea prin selecție, HeapSort împarte intrarea sa într-o regiune sortată și una nesortată, și își micșorează iterativ regiunea nesortată extrăgând elementul cel mai mare și mutându-l în regiunea sortată.

## 2.2 Merge Sort

Merge Sort este un algoritm eficient, de uz general, bazat pe comparații. Este un exemplu de algoritm Divide and Conquer. Este un algoritm stabil, ceea ce înseamnă că implementarea păstrează ordinea de intrare a elementelor egale în ieșirea sortată.

## 2.3 Radix Sort

Radix Sort este un algoritm de sortare necomparativ. Evită compararea prin crearea și distribuirea elementelor în compartimente în funcție de radixul lor. Pentru elementele cu mai mult de o cifră numerică, acest proces de compartimentare se repetă pentru fiecare cifră, în timp ce se păstrează ordinea de la trecerea anterioară, până când toate cifrele au fost procesate.

## 2.4 Quick Sort

Quick Sort este un algoritm de sortare eficient dezvoltat de Tony Hoare. În medie, face $\Theta(n \log n)$ comparații pentru a sorta n elemente, în cel mai rău caz, face $\Theta(n^2)$ comparații. QuickSort este de obicei mai rapid în practică decât alți algoritmi $\Theta(n \log n)$.

### 2.4.1 Quick Sort cu Pivot la Mijloc

QuickSort cu Pivot la Mijloc alege elementul din mijlocul partiției curente ca pivot. Elementul pivot este mutat la începutul partiției, apoi doi pointeri, left și right, sunt folosiți pentru a parcurge partiția. Algoritmul se apelează recursiv pe subpartițiile din stânga și din dreapta pivotului. Scopul este de a echilibra partițiile, presupunând că elementul din mijloc are șanse mari să fie aproape de mediana partiției.

### 2.4.2 Quick Sort cu Pivot Media a 3

QuickSort cu Pivot Media a 3 alege mediana dintre primul, ultimul și elementul din mijloc al partiției curente ca pivot. Se calculează mediana dintre cele trei elemente, iar mediana este mutată la începutul partiției. Această strategie reduce probabilitatea de a alege un pivot extrem, ceea ce ar duce la partiții dezechilibrate și performanțe slabe.

### 2.4.3 Quick Sort cu Pivot Media a 5

QuickSort cu Pivot Media a 5 extinde abordarea medianei la cinci elemente. Se alege mediana dintre primul, ultimul, elementul din mijloc și încă două elemente mijlocii ale partiției curente. Se calculează mediana dintre cele cinci elemente, mediana este mutată la începutul partiției, se efectuează partiționarea și se fac apeluri recursive. Această metodă îmbunătățește și mai mult robustețea alegerii pivotului, cu un cost computațional ușor mai mare pentru calcularea medianei.

### 2.4.4 Quick Sort cu Primul Pivot

QuickSort cu Primul Pivot alege primul element al partiției curente ca pivot. Primul element este considerat pivot, se efectuează partiționarea folosind aceeași logică de bază (parcurgere cu doi pointeri și schimb de elemente), și se fac apeluri recursive. Aceasta este cea mai simplă strategie de pivotare, dar este susceptibilă la performanțe slabe pe date deja sortate sau aproape sortate.

# 3 Shell Sort

Shell Sort este o generalizare a sortării prin inserție care permite schimbul de elemente aflate la distanță. Ideea este de a sorta elementele aflate la distanță unul de altul, apoi de a micșora treptat decalajul dintre elementele de comparat.

## 3.1 Shell Sort simplu

Shell sort simplu începe prin sortarea perechilor de elemente aflate la distanță unul de altul, apoi reduce treptat decalajul.

### 3.2 Shell Sort Hibbard

Shell Sort Hibbard folosește secvența Hibbard pentru a defini decalajele. Secvența Hibbard este o secvență de numere dată de formula $h_k = 2^k - 1$, unde $k$ este un întreg pozitiv. Această secvență generează decalaje precum 1, 3, 7, 15, 31 și așa mai departe.

### 3.3 Shell Sort Sedgewick

Shell Sort Sedgewick este o altă variantă a algoritmului Shell Sort, dar folosește o secvență diferită de decalaje, cunoscută sub numele de secvența Sedgewick. Secvența Sedgewick este definită de următoarele formule:

- $h_k = 4^k + 3 \cdot 2^{(k-1)} + 1$ pentru $k$ par
- $h_k = 9 \cdot 2^k - 9 \cdot 2^{(k/2)} + 1$ pentru $k$ impar

Această secvență produce decalaje precum 1, 5, 19, 41, 109, și așa mai departe. S-a demonstrat empiric că secvența Sedgewick are performanțe mai bune în practică decât alte secvențe, cum ar fi secvența Hibbard sau decalajul simplu de n // 2.

## 4 Tim Sort

Timsort este un algoritm de sortare hibrid derivat din merge sort și insertion sort, proiectat pentru a funcționa bine pe multe tipuri de date reale. Implementarea a fost creată de Tim Peters în 2002 pentru a fi utilizată în limbajul de programare Python.

## 5 Analiza Comparativă a Algoritmilor de Sortare

Această secțiune prezintă o analiză detaliată a performanței diferiților algoritmi de sortare pe diverse seturi de date. Analiza se bazează pe timpii medii de execuție măsurați pentru fiecare algoritm pe diferite seturi de date, fiecare având o dimensiune specifică.

### 5.1 Metodologie

Timpii de execuție au fost măsurați pentru fiecare algoritm pe un set de patru seturi de date sintetice. Seturile de date au fost generate cu dimensiuni variind. Mai exact, avem:

- Setul de date 1: n = 1
- Setul de date 2: n = 2
- Setul de date 3: n = 3
- Setul de date 4: n = 4

Pentru fiecare set de date, au fost generate mai multe instanțe, iar timpul mediu de execuție a fost calculat. Algoritmii de sortare analizați includ:

- Shell Sort (cu variantele Hibbard și Sedgewick)
- Tim Sort (cu dimensiunile de rulare 32, 128 și 512)
- Built-in Tim Sort (implementarea Python)
- Merge Sort
- Heap Sort
- Radix Sort (cu bazele 10, 16 și 16-bit)

Seturile de date sintetice au fost concepute pentru a reprezenta diferite scenarii de intrare, inclusiv:

- Seturi de date cu valori într-un interval mic

- Seturi de date cu puține valori unice
- Seturi de date cu numere mari
- Seturi de date sortate crescător
- Seturi de date sortate descrescător
- Seturi de date aproape sortate
- Seturi de date cu valori alternante mari și mici
- Seturi de date cu porțiuni inversate
- Seturi de date cu valori aleatorii

## 5.2 Rezultate Experimentale

### 5.2.1 Seturi de Date cu Interval Mic

Figura 1 și Tabelul 1 prezintă rezultatele pentru seturile de date cu valori într-un interval mic.

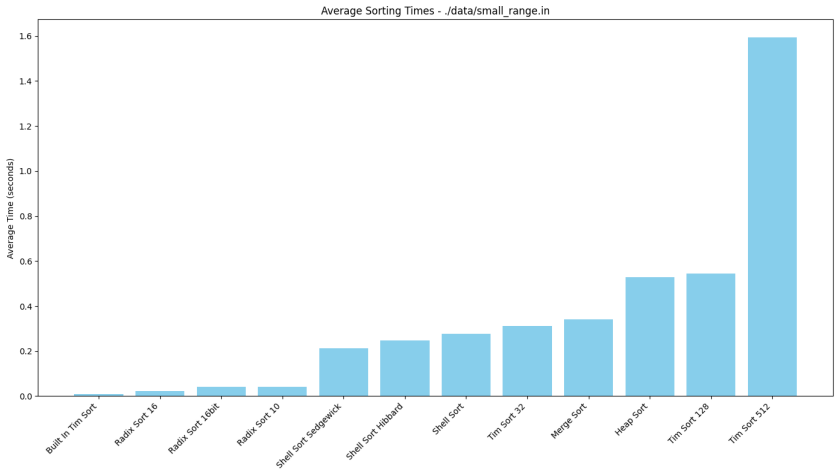


Figure 1: Timpii medii de sortare pentru seturile de date cu interval mic.

| Algoritm | Setul de date 1 (n=1) | Setul de date 2 (n=2) | Setul de date 3 (n=3) | Setul de date 4 (n=4) |
|---|---|---|---|---|
| Shell Sort | 0.0014 | 0.0181 | 0.1639 | 0.9212 |
| Shell Sort Hibbard | 0.0014 | 0.0152 | 0.1507 | 0.8271 |
| Shell Sort Sedgewick | 0.0029 | 0.0237 | 0.1255 | 0.7036 |
| Tim Sort 32 | 0.0018 | 0.0224 | 0.1947 | 1.0265 |
| Tim Sort 128 | 0.0093 | 0.0370 | 0.3435 | 1.7904 |
| Tim Sort 512 | 0.0247 | 0.1042 | 1.0612 | 5.1847 |
| Built In Tim Sort | 0.0001 | 0.0006 | 0.0065 | 0.0336 |
| Merge Sort | 0.0027 | 0.0224 | 0.2102 | 1.1304 |
| Heap Sort | 0.0033 | 0.0285 | 0.3177 | 1.7683 |
| Radix Sort 10 | 0.0005 | 0.0026 | 0.0333 | 0.1322 |
| Radix Sort 16 | 0.0003 | 0.0016 | 0.0198 | 0.0707 |
| Radix Sort 16bit | 0.0199 | 0.0194 | 0.0298 | 0.0936 |

Table 1: Timpii medii de sortare (în secunde) pentru seturile de date cu interval mic.

**Observații:**

- Built-in Tim Sort are performanțe semnificativ mai bune decât ceilalți algoritmi pe toate seturile de date.

- Radix Sort 16 este al doilea cel mai rapid algoritm pentru seturile de date 1, 2 și 3.
- Shell Sort Sedgewick are performanțe mai bune decât celelalte variante de Shell Sort.
- Tim Sort cu dimensiuni de rulare mai mari (128, 512) devine mai lent odată cu creșterea dimensiunii setului de date.

### 5.2.2 Seturi de Date cu Puține Valori Unice

Figura 2 și Tabelul 2 prezintă rezultatele pentru seturile de date cu puține valori unice.

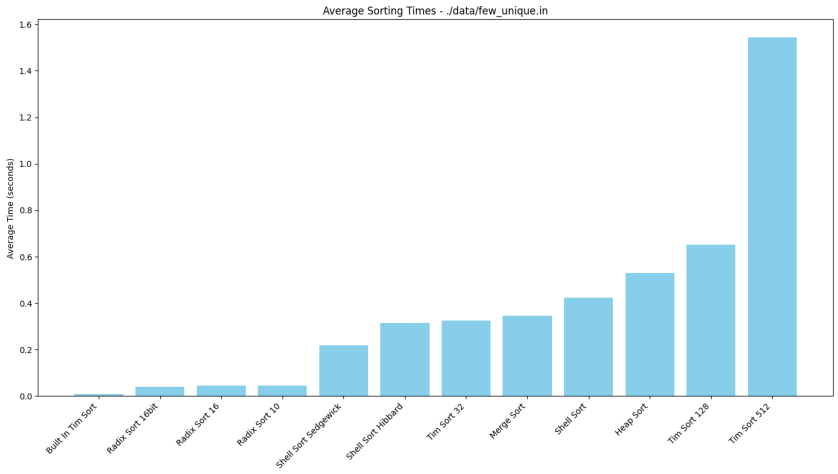


Figure 2: Timpii medii de sortare pentru seturile de date cu puține valori unice.

| Algoritm | Setul de date 1 (n=1) | Setul de date 2 (n=2) | Setul de date 3 (n=3) | Setul de date 4 (n=4) |
|---|---|---|---|---|
| Shell Sort | 0.0012 | 0.0372 | 0.1444 | 0.7729 |
| Shell Sort Hibbard | 0.0010 | 0.0268 | 0.1380 | 0.8348 |
| Shell Sort Sedgewick | 0.0023 | 0.0148 | 0.1063 | 0.7494 |
| Tim Sort 32 | 0.0031 | 0.0253 | 0.1800 | 1.0891 |
| Tim Sort 128 | 0.0065 | 0.0332 | 0.3137 | 2.2580 |
| Tim Sort 512 | 0.0211 | 0.0912 | 0.9067 | 5.1553 |
| Built In Tim Sort | 0.0001 | 0.0005 | 0.0045 | 0.0324 |
| Merge Sort | 0.0034 | 0.0237 | 0.2095 | 1.1478 |
| Heap Sort | 0.0040 | 0.0255 | 0.2763 | 1.8131 |
| Radix Sort 10 | 0.0006 | 0.0025 | 0.0317 | 0.1509 |
| Radix Sort 16 | 0.0007 | 0.0033 | 0.0356 | 0.1425 |
| Radix Sort 16bit | 0.0164 | 0.0181 | 0.0245 | 0.0652 |

Table 2: Timpii medii de sortare (în secunde) pentru seturile de date cu puține valori unice.

**Observații:**

- Similar cu seturile de date cu interval mic, Built-in Tim Sort excelează.
- Radix Sort 16 are performanțe bune pentru seturile de date 1 și 2.
- Shell Sort Sedgewick este din nou cea mai bună variantă a Shell Sort.

### 5.2.3 Seturi de Date cu Numere Mari

Figura 3 și Tabelul 3 prezintă rezultatele pentru seturile de date cu numere mari.
**Observații:**

- Built-in Tim Sort rămâne cel mai rapid.
- Radix Sort 16bit are performanțe surprinzător de bune pentru seturile de date 3 și 4.

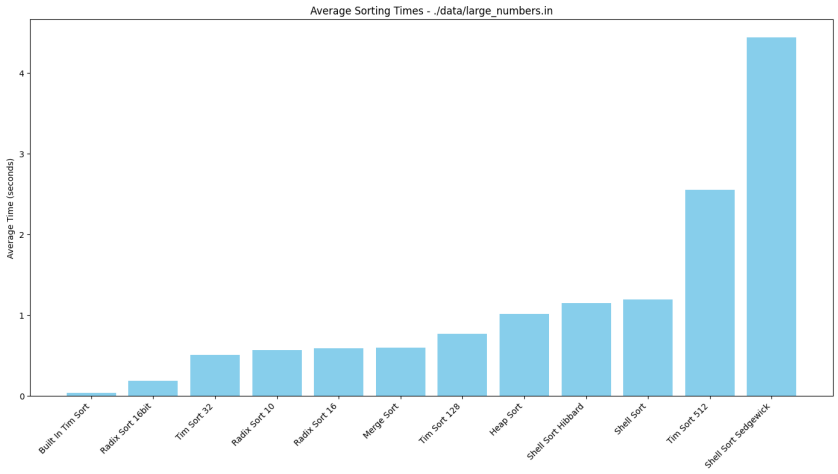


Figure 3: Timpii medii de sortare pentru seturile de date cu numere mari.

| Algoritm | Setul de date 1 (n=1) | Setul de date 2 (n=2) | Setul de date 3 (n=3) | Setul de date 4 (n=4) |
|---|---|---|---|---|
| Shell Sort | 0.0021 | 0.0298 | 0.2254 | 1.2585 |
| Shell Sort Hibbard | 0.0016 | 0.0292 | 0.1701 | 1.3414 |
| Shell Sort Sedgewick | 0.0033 | 0.0690 | 0.1971 | 0.7729 |
| Tim Sort 32 | 0.0025 | 0.0245 | 0.4490 | 1.0558 |
| Tim Sort 128 | 0.0049 | 0.0882 | 0.7712 | 2.2195 |
| Tim Sort 512 | 0.0201 | 0.2979 | 1.7610 | 5.2152 |
| Built In Tim Sort | 0.0001 | 0.0007 | 0.0191 | 0.0324 |
| Merge Sort | 0.0039 | 0.0261 | 0.2268 | 1.1578 |
| Heap Sort | 0.0046 | 0.0362 | 0.3547 | 1.8131 |
| Radix Sort 10 | 0.0009 | 0.0110 | 0.1047 | 0.1509 |
| Radix Sort 16 | 0.0010 | 0.0159 | 0.0861 | 0.1425 |
| Radix Sort 16bit | 0.0157 | 0.0485 | 0.0652 | 0.0526 |

Table 3: Timpii medii de sortare (în secunde) pentru seturile de date cu numere mari.

- Shell Sort Sedgewick are performanțe mai bune decât celelalte variante de Shell Sort pentru seturile de date 3 și 4.

#### 5.2.4 Alte Tipuri de Seturi de Date

Rezultatele pentru alte tipuri de seturi de date (sortate crescător, descrescător, aproape sortate, alternante, cu porțiuni inversate și aleatoare) sunt prezentate în graficele corespunzătoare (Figurile 4 - 9). În general, Built-in Tim Sort are performanțe consistente bune în toate scenariile. Radix Sort are performanțe bune pe anumite tipuri de date, dar performanța sa poate varia în funcție de distribuția datelor. Algoritmii de sortare bazată pe comparație (Shell Sort, Merge Sort, Heap Sort) au performanțe relativ similare, cu mici variații în funcție de caracteristicile specifice ale datelor de intrare.

### 5.3 Concluzii

Analiza experimentală evidențiază următoarele observații cheie:

- Built-in Tim Sort (implementarea Python) are în mod constant cele mai bune performanțe în majoritatea scenariilor, demonstrând eficiența optimizărilor sale pentru datele din lumea reală.
- Radix Sort poate fi competitiv pentru anumite tipuri de date (de exemplu, interval mic, puține valori unice, numere mari), dar performanța sa este foarte dependentă de distribuția datelor și de baza utilizată.
- Variantele Shell Sort (în special Sedgewick) oferă îmbunătățiri față de implementarea simplă, dar sunt în general depășite de Built-in Tim Sort și Radix Sort în scenariile optime ale acestora.

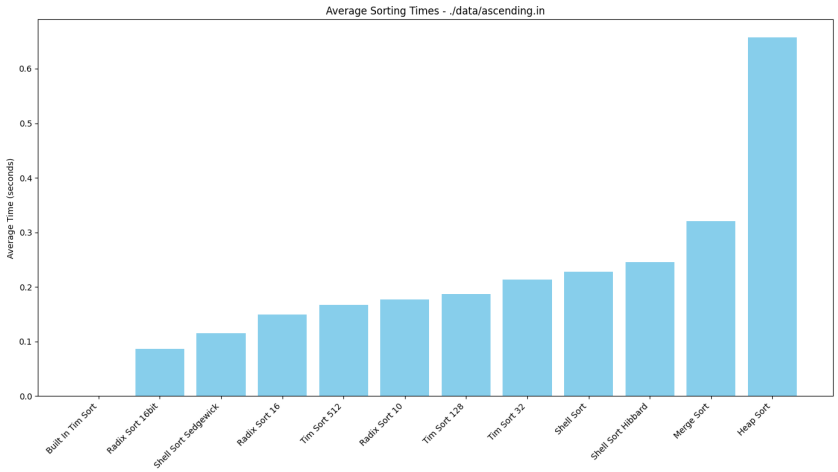


Figure 4: Timpii medii de sortare pentru seturile de date sortate crescător.

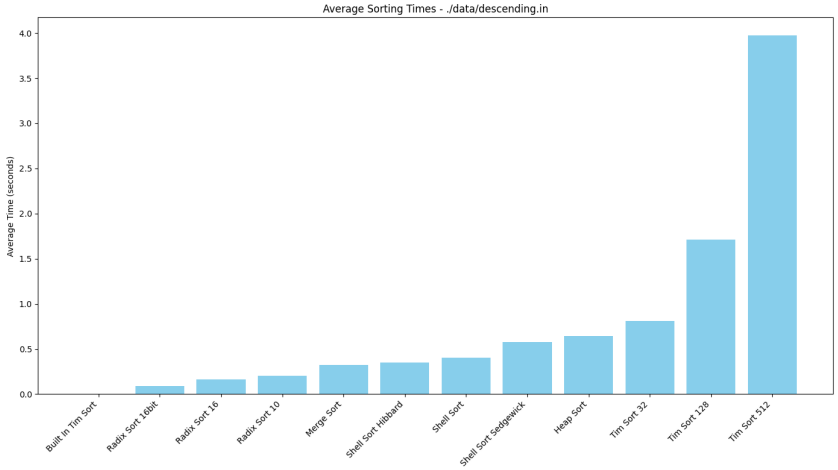


Figure 5: Timpii medii de sortare pentru seturile de date sortate descrescător.

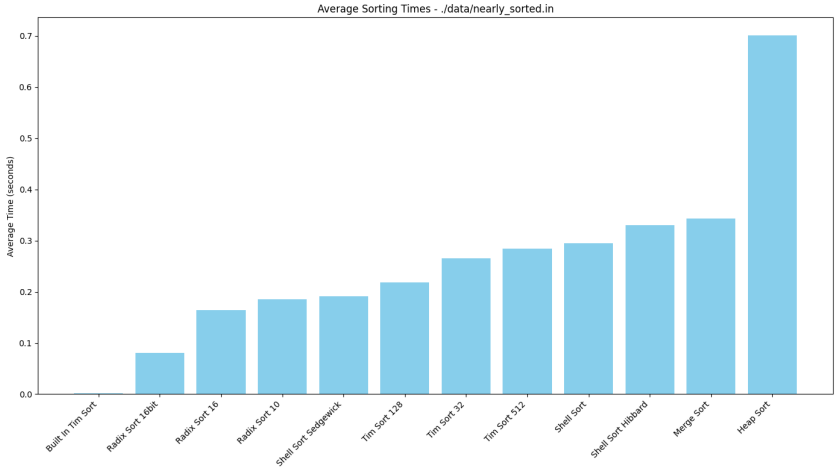


Figure 6: Timpii medii de sortare pentru seturile de date aproape sortate.

- Algoritmii de sortare bazată pe comparație (Merge Sort, Heap Sort) au performanțe relativ similare, cu mici variații în funcție de caracteristicile datelor de intrare.

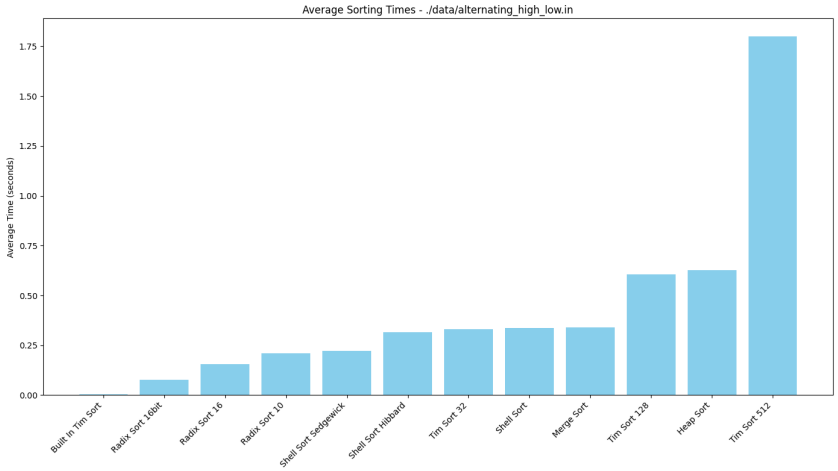


Figure 7: Timpii medii de sortare pentru seturile de date cu valori alternante mari și mici.

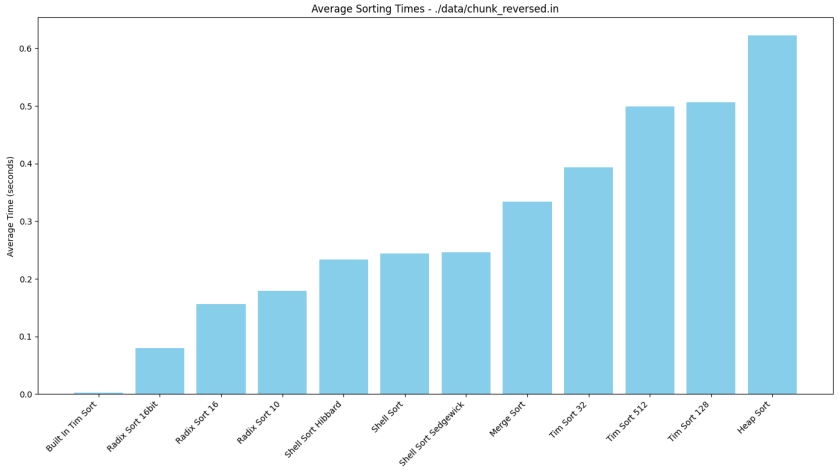


Figure 8: Timpii medii de sortare pentru seturile de date cu porțiuni inversate.

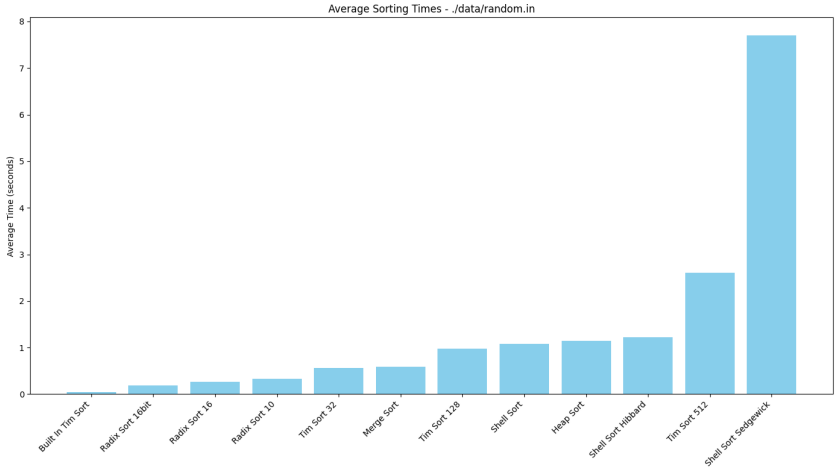


Figure 9: Timpii medii de sortare pentru seturile de date aleatoare.