Algoritmi de Sortare

Ce algoritmi sunt mai rapizi si in ce contexte

Baciu Roberto - Balan Teodor - Patru Mihai

11:11PM

x Sort • Radix Sort • Radix Sort• Radix Sort • Radix Sort • Radix Sort • Radix Sort • Radix Sort • Radix Sort • Radix Sort • Radix

Radix Sort (baza 10)

Algoritm

```cpp
void RadixSortBase10(vector<long long> &v, long long n, long long maxim)
{
    vector<long long> output(n);

    long long exp = 1;

    while (maxim / exp > 0)
    {
        vector<long long> count(10, 0);

        for (long long i = 0; i < n; i++)
            count[(v[i] / exp) % 10]++;

        for (long long i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for (long long i = n - 1; i >= 0; i--)
        {
            output[count[(v[i] / exp) % 10] - 1] = v[i];
            count[(v[i] / exp) % 10]--;
        }

        for (long long i = 0; i < n; i++)
            v[i] = output[i];

        exp *= 10;
    }
}
```

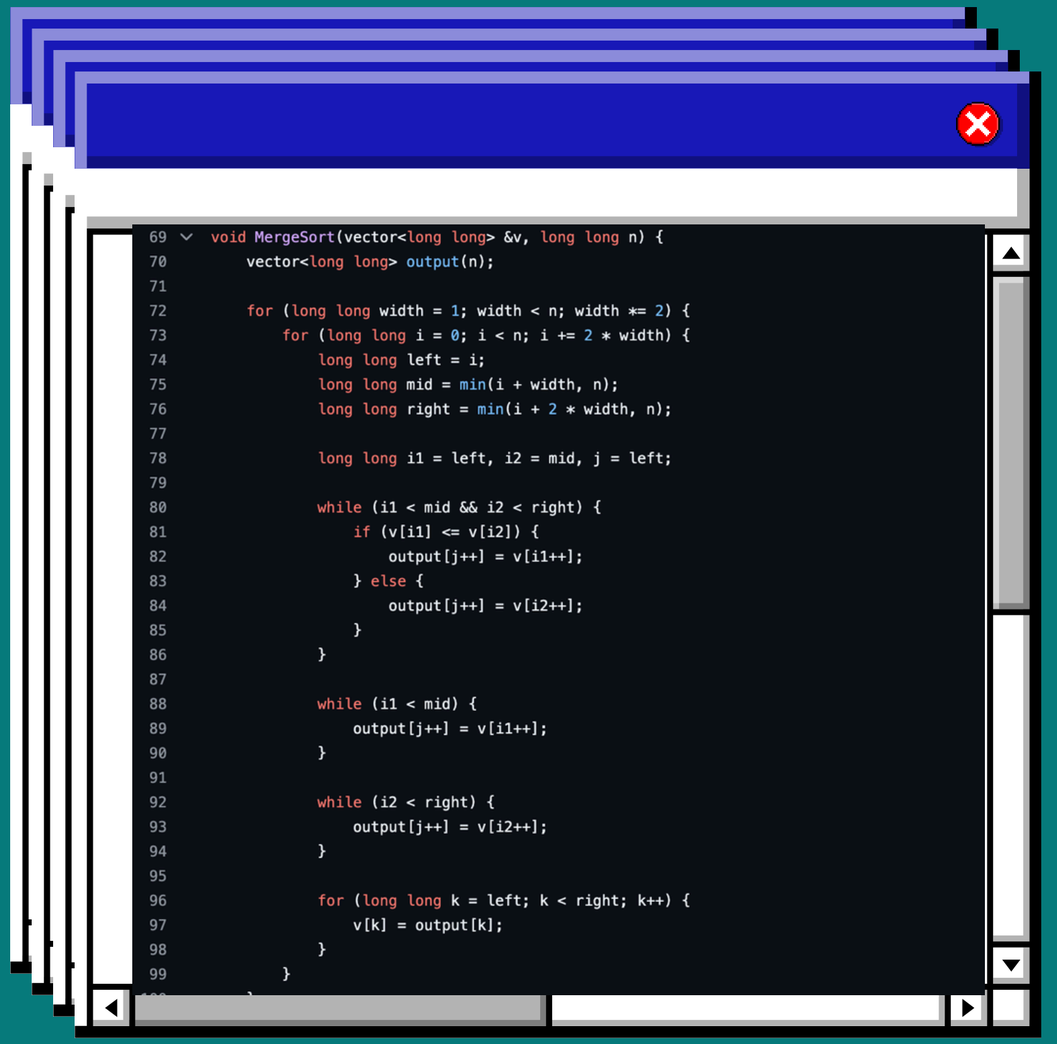
Start

Radix Sort (baza 2^16)

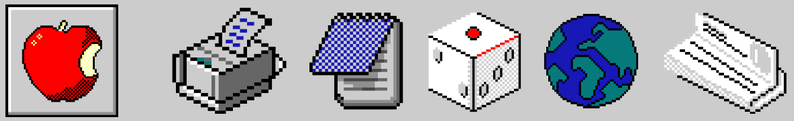
```cpp
void RadixSortBase2pow16(vector<long long> &v, long long n, long long maxim)
{
    vector<long long> output(n);

    const long long BASE = 65536;

    long long exp = 1;

    while (maxim / exp > 0)
    {
        vector<long long> count(BASE, 0);

        for (long long i = 0; i < n; i++)
            count[(v[i] / exp) % BASE]++;

        for (long long i = 1; i < BASE; i++)
            count[i] += count[i - 1];

        for (long long i = n - 1; i >= 0; i--)
        {
            output[count[(v[i] / exp) % BASE] - 1] = v[i];
            count[(v[i] / exp) % BASE]--;
        }

        for (long long i = 0; i < n; i++)
            v[i] = output[i];

        exp *= BASE;
    }
```

Back to Main Page

Merge Sort

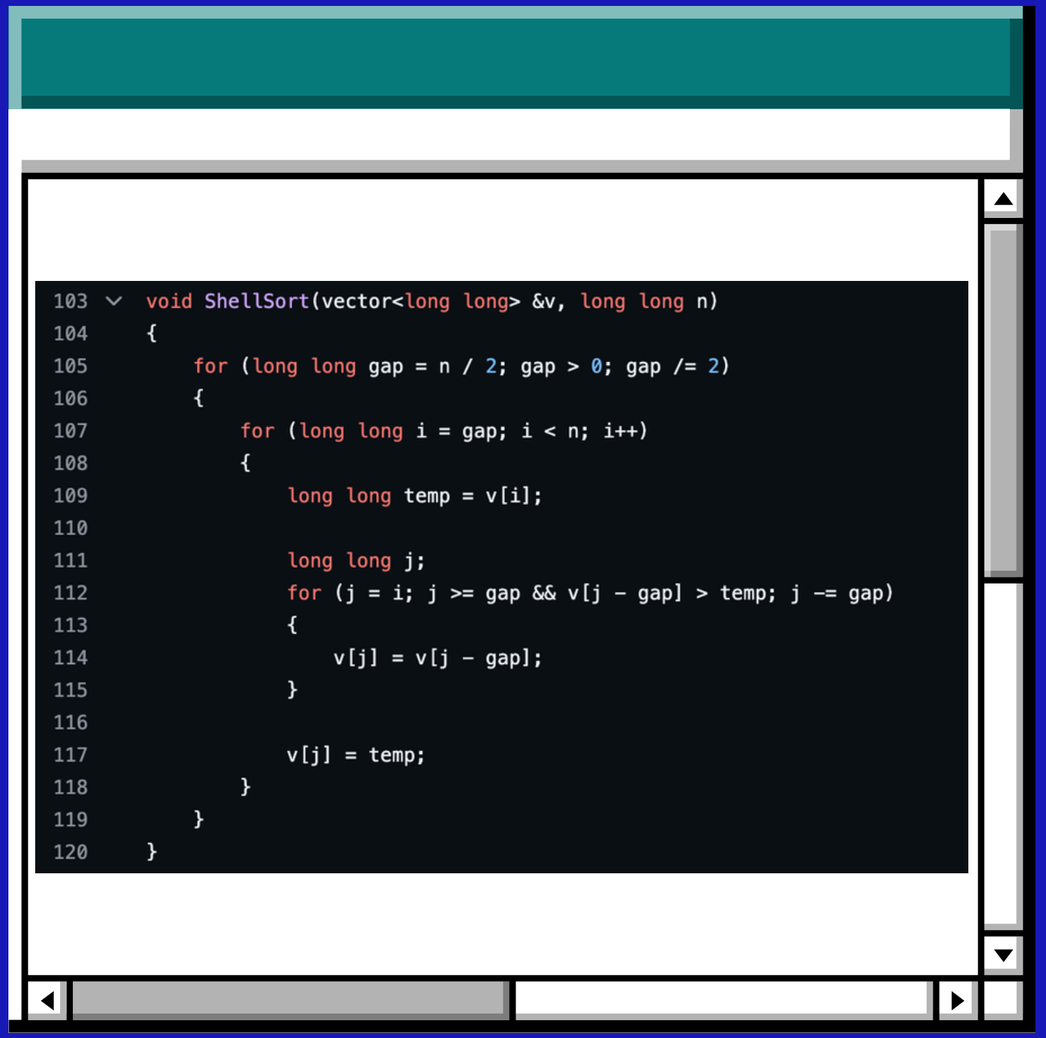


```
void MergeSort(vector<long long> &v, long long n) {
    vector<long long> output(n);

    for (long long width = 1; width < n; width *= 2) {
        for (long long i = 0; i < n; i += 2 * width) {
            long long left = i;
            long long mid = min(i + width, n);
            long long right = min(i + 2 * width, n);

            long long i1 = left, i2 = mid, j = left;

            while (i1 < mid && i2 < right) {
                if (v[i1] <= v[i2]) {
                    output[j++] = v[i1++];
                } else {
                    output[j++] = v[i2++];
                }
            }

            while (i1 < mid) {
                output[j++] = v[i1++];
            }

            while (i2 < right) {
                output[j++] = v[i2++];
            }

            for (long long k = left; k < right; k++) {
                v[k] = output[k];
            }
        }
```



Back to Main Page

Shell Sort



```
103  void ShellSort(vector<long long> &v, long long n)
104  {
105      for (long long gap = n / 2; gap > 0; gap /= 2)
106      {
107          for (long long i = gap; i < n; i++)
108          {
109              long long temp = v[i];
110
111              long long j;
112              for (j = i; j >= gap && v[j - gap] > temp; j -= gap)
113              {
114                  v[j] = v[j - gap];
115              }
116
117              v[j] = temp;
118          }
119      }
120  }
```

Back to Main Page

Shell Sort (Ciura)

```cpp
void ShellSortCiura(vector<long long> &v, long long n)
{
    vector<long long> gaps = {1, 4, 10, 23, 57, 132, 301, 701, 1750};

    while (gaps.back() < n/2.25) {
        gaps.push_back(gaps.back() * 2.25);
    }

    for (int k = gaps.size() - 1; k >= 0; k--)
    {
        long long gap = gaps[k];

        for (long long i = gap; i < n; i++)
        {
            long long temp = v[i];

            long long j;
            for (j = i; j >= gap && v[j - gap] > temp; j -= gap)
            {
                v[j] = v[j - gap];
            }

            v[j] = temp;
        }
    }
}
```

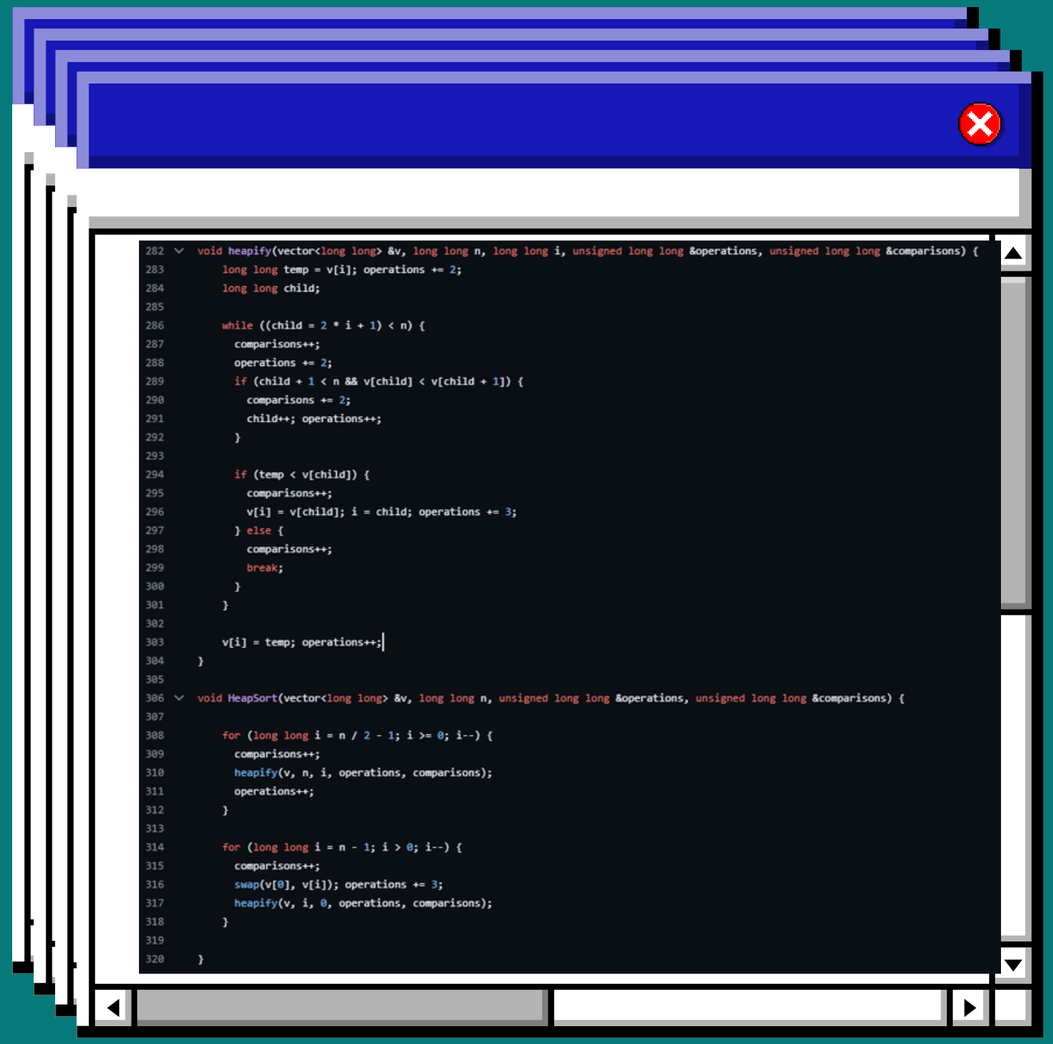
Back to Main Page

Count Sort

```cpp
void CountSort(vector<long long> &v, long long n, long long maxim)
{
    vector<long long> count(maxim + 1, 0);
    vector<long long> output(n);

    for (long long i = 0; i < n; i++)
        count[v[i]]++;

    for (long long i = 1; i <= maxim; i++)
        count[i] += count[i - 1];

    for (long long i = n - 1; i >= 0; i--)
    {
        output[count[v[i]] - 1] = v[i];
        count[v[i]]--;
    }

    for (long long i = 0; i < n; i++)
        v[i] = output[i];
}
```

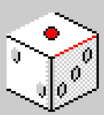
Back to Main Page

Heap Sort



```cpp
void heapify(vector<long long> &v, long long n, long long i, unsigned long long &operations, unsigned long long &comparisons) {
    long long temp = v[i]; operations += 2;
    long long child;

    while ((child = 2 * i + 1) < n) {
      comparisons++;
      operations += 2;
      if (child + 1 < n && v[child] < v[child + 1]) {
        comparisons += 2;
        child++; operations++;
      }

      if (temp < v[child]) {
        comparisons++;
        v[i] = v[child]; i = child; operations += 3;
      } else {
        comparisons++;
        break;
      }
    }

    v[i] = temp; operations++;
}

void HeapSort(vector<long long> &v, long long n, unsigned long long &operations, unsigned long long &comparisons) {

    for (long long i = n / 2 - 1; i >= 0; i--) {
      comparisons++;
      heapify(v, n, i, operations, comparisons);
      operations++;
    }

    for (long long i = n - 1; i > 0; i--) {
      comparisons++;
      swap(v[0], v[i]); operations += 3;
      heapify(v, i, 0, operations, comparisons);
    }

}
```



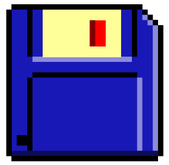



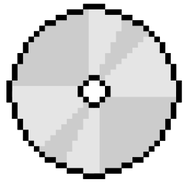
Back to Main Page

Quick Sort

Trei tipuri de Quick Sort:

- Mediana de mijloc
- Mediana din 3
- Mediana din 5



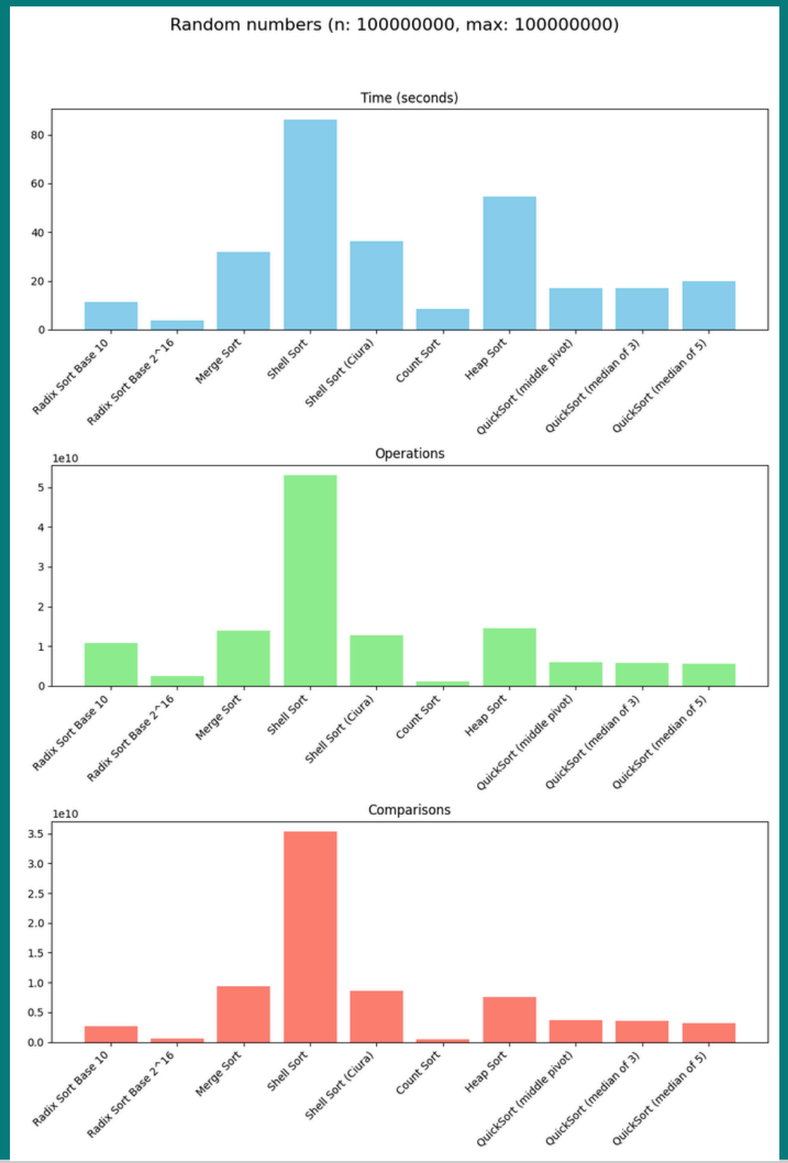


```cpp
void QuickSortMiddle(vector<long long> &v, int left, int right)
{
    if(left < right)
    {
        int m = (left + right) / 2;
        int aux = v[left];
        v[left] = v[m];
        v[m] = aux;
        int i = left , j = right, d = 0;
        while(i < j)
        {
            if(v[i] > v[j])
            {
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
                d = 1 - d;
            }
            i += d;
            j -= 1 - d;
        }
        QuickSortMiddle(v, left , i - 1);
        QuickSortMiddle(v, i + 1 , right);
    }
}
```

```cpp
void QuickSortMedian3(vector<long long> &v, int left, int right)
{
    if(left < right)
    {
        int m = (left + right) / 2;

        if(v[left] > v[m]) swap(v[left], v[m]);
        if(v[m] > v[right]) swap(v[m], v[right]);
        if(v[left] > v[m]) swap(v[left], v[m]);

        swap(v[left], v[m]);

        int i = left , j = right, d = 0;
        while(i < j)
        {
            if(v[i] > v[j])
            {
                swap(v[i], v[j]);
                d = 1 - d;
            }
            i += d;
            j -= 1 - d;
        }
        QuickSortMedian3(v, left , i - 1);
        QuickSortMedian3(v, i + 1 , right);
    }
}
```

```cpp
int medianOfFive(vector<long long>& v, int idx1, int idx2, int idx3, int idx4, int idx5) {
    int arr[5] = {idx1, idx2, idx3, idx4, idx5};
    for (int i = 1; i < 5; i++) {
        int j = i;
        while (j > 0 && v[arr[j-1]] > v[arr[j]]) {
            swap(arr[j-1], arr[j]);
            j--;
        }
    }
    return arr[2];
}

void QuickSortMedian5(vector<long long>& v, int low, int high, unsigned long long &operations, unsigned long long &comparisons) {
    operations++;
    comparisons++;
    if(low < high) {
        int idx1 = low;
        int idx2 = low + (high - low) / 4;
        int idx3 = low + (high - low) / 2;
        int idx4 = low + 3 * (high - low) / 4;
        int idx5 = high;
        int medianIdx = medianOfFive(v, idx1, idx2, idx3, idx4, idx5); operations++;
        long long pivot = v[medianIdx]; operations++;

        int i = low, j = high; operations += 2;
        while(i <= j) {
            comparisons++;
            while(v[i] < pivot) { comparisons++; i++; operations++; }
            while(v[j] > pivot) { comparisons++; j--; operations++; }
            comparisons++;
            if(i <= j) {
                swap(v[i], v[j]); operations += 3;
                i++; j--; operations += 2;
            }
        }
        comparisons++; if(low < j) QuickSortMedian5(v, low, j, operations, comparisons);
        comparisons++; if(i < high) QuickSortMedian5(v, i, high, operations, comparisons);
    }
}
```

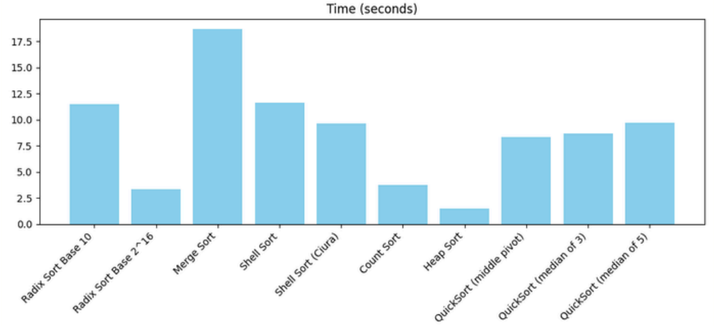
Test Case 1

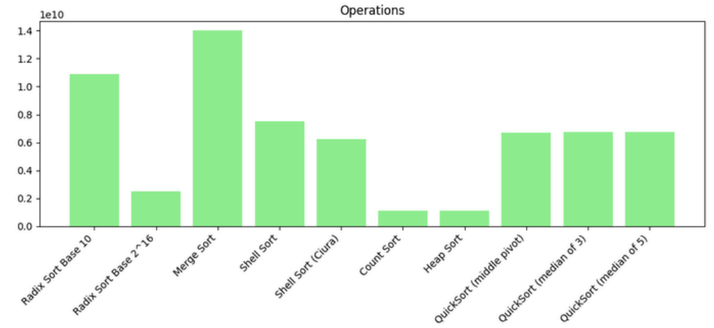


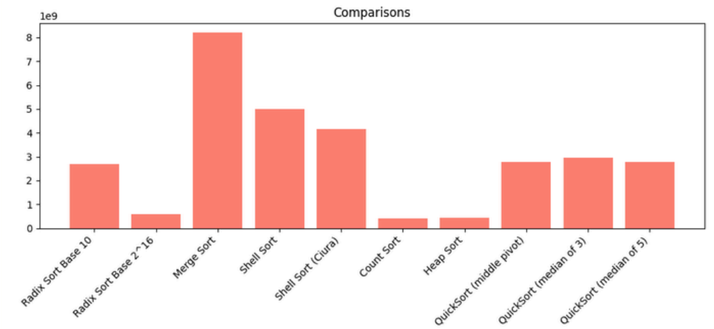
Back to Main Page

Test Case 2

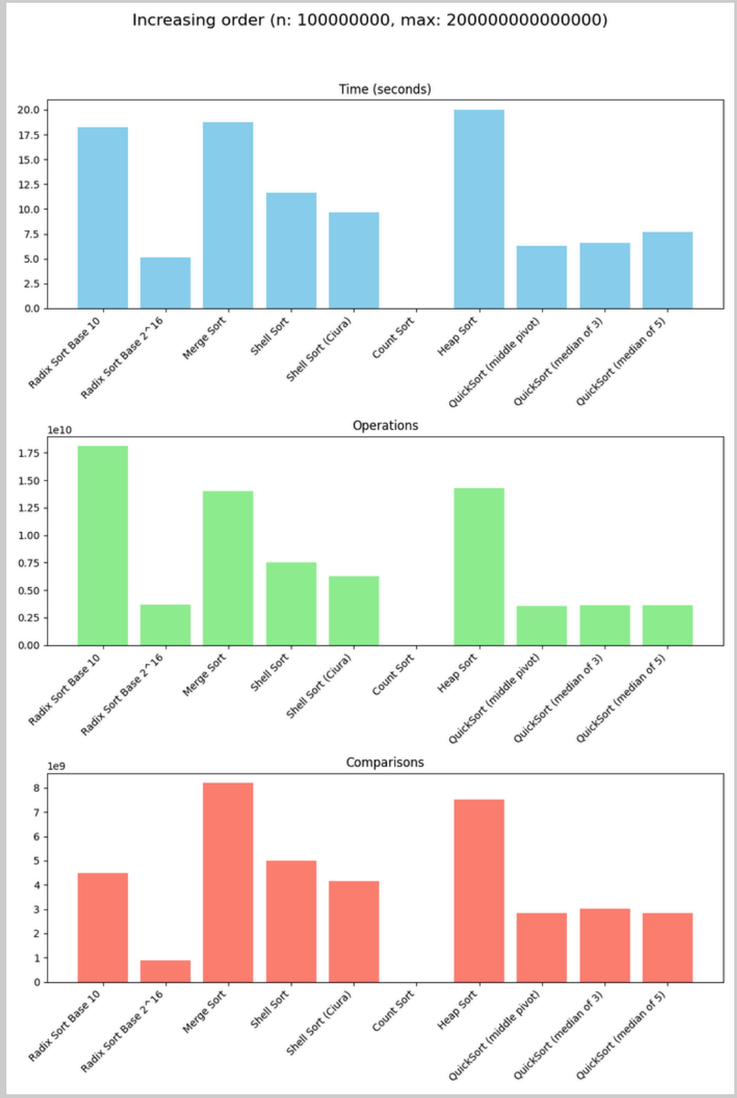
One repeated number (n: 100000000, max: 100000000)



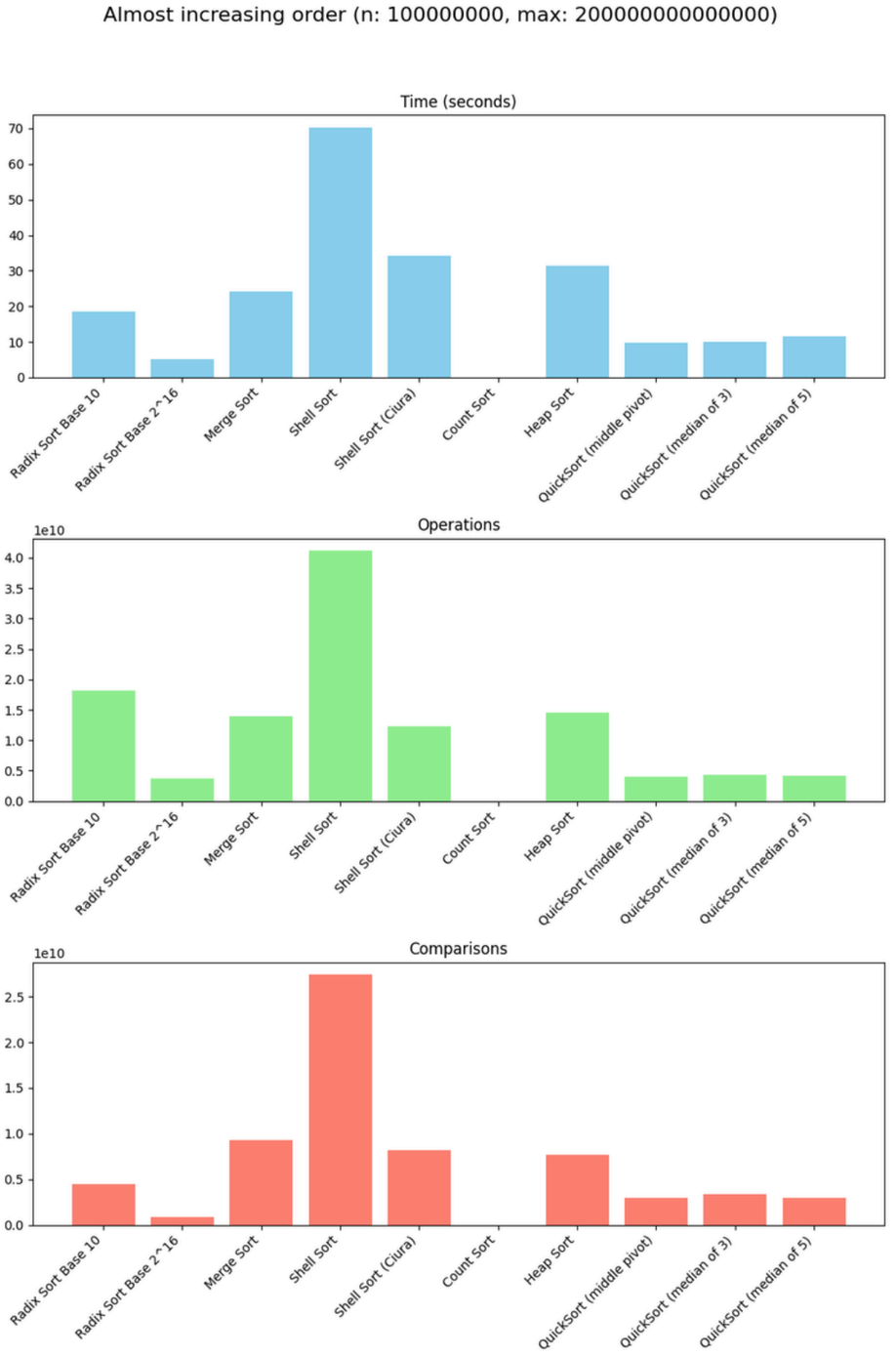




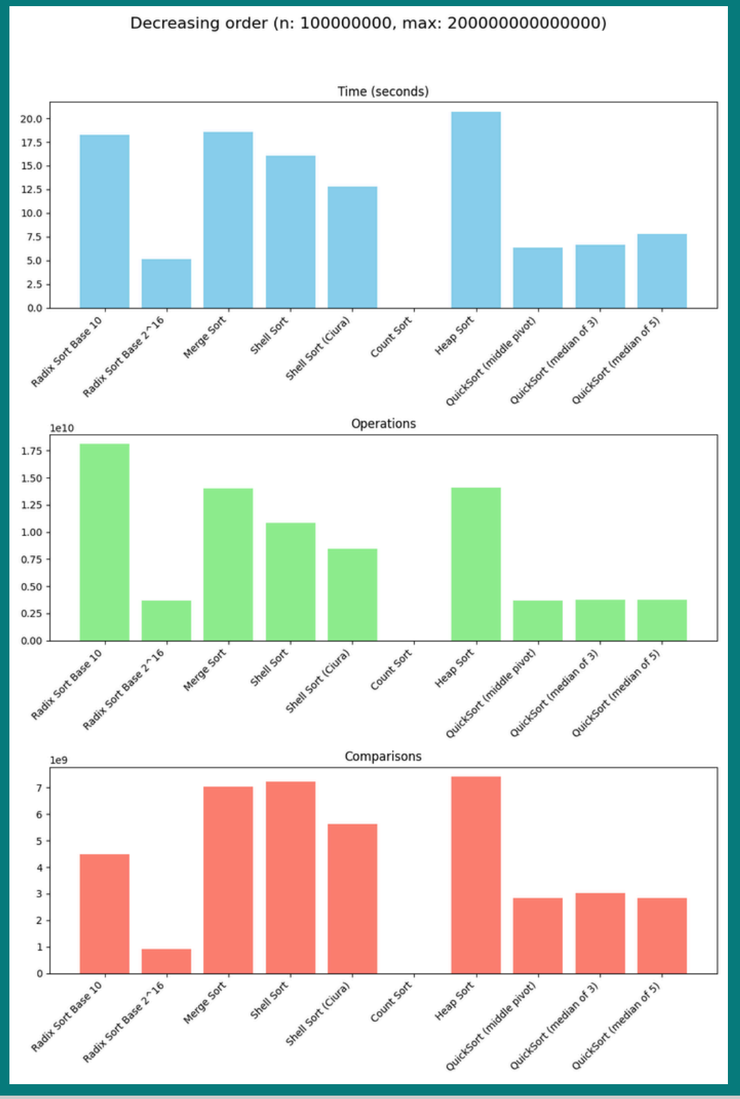
Test Case 3


Increasing order (n: 100000000, max: 200000000000000)
Time (seconds)
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Operations
1e10
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Comparisons
1e9
8
7
6
5
4
3
2
1
0

Test Case 4

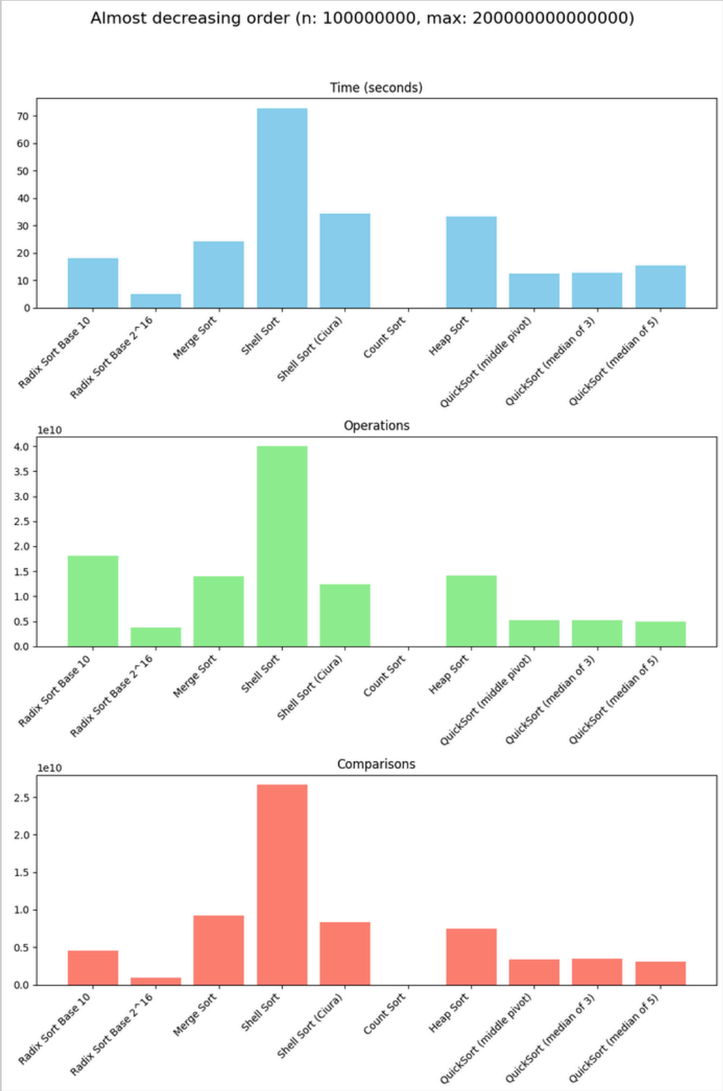

Almost increasing order (n: 100000000, max: 200000000000000)
Time (seconds)
70
60
50
40
30
20
10
0
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Operations
1e10
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Comparisons
1e10
2.5
2.0
1.5
1.0
0.5
0.0
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)

Test Case 5

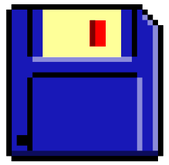


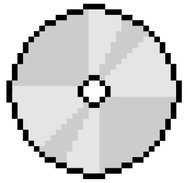
Back to Agenda Page

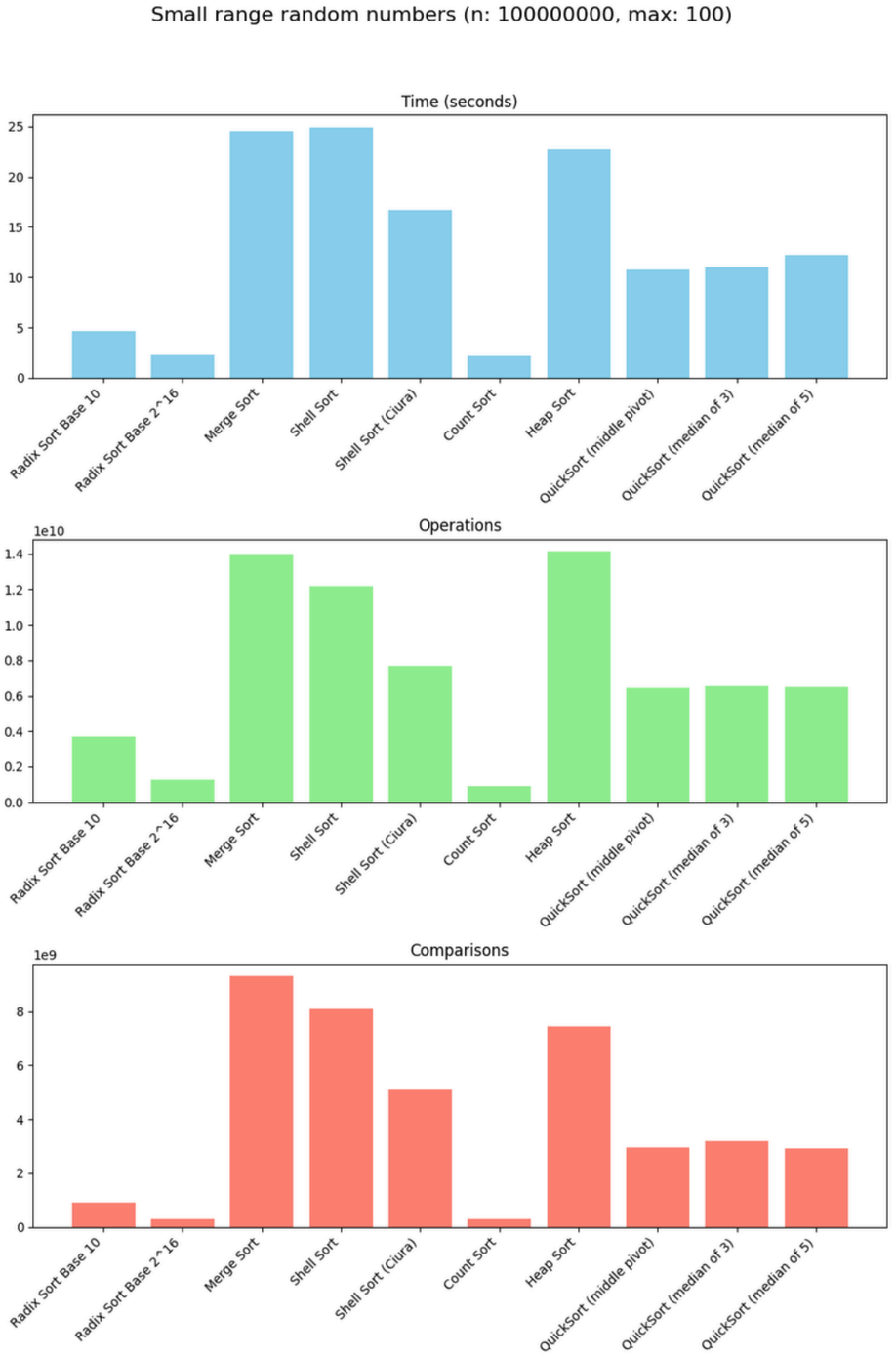
Test Case 6


Almost decreasing order (n: 100000000, max: 200000000000000)
Time (seconds)
0
10
20
30
40
50
60
70
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Operations
1e10
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Comparisons
1e10
0.0
0.5
1.0
1.5
2.0
2.5
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)

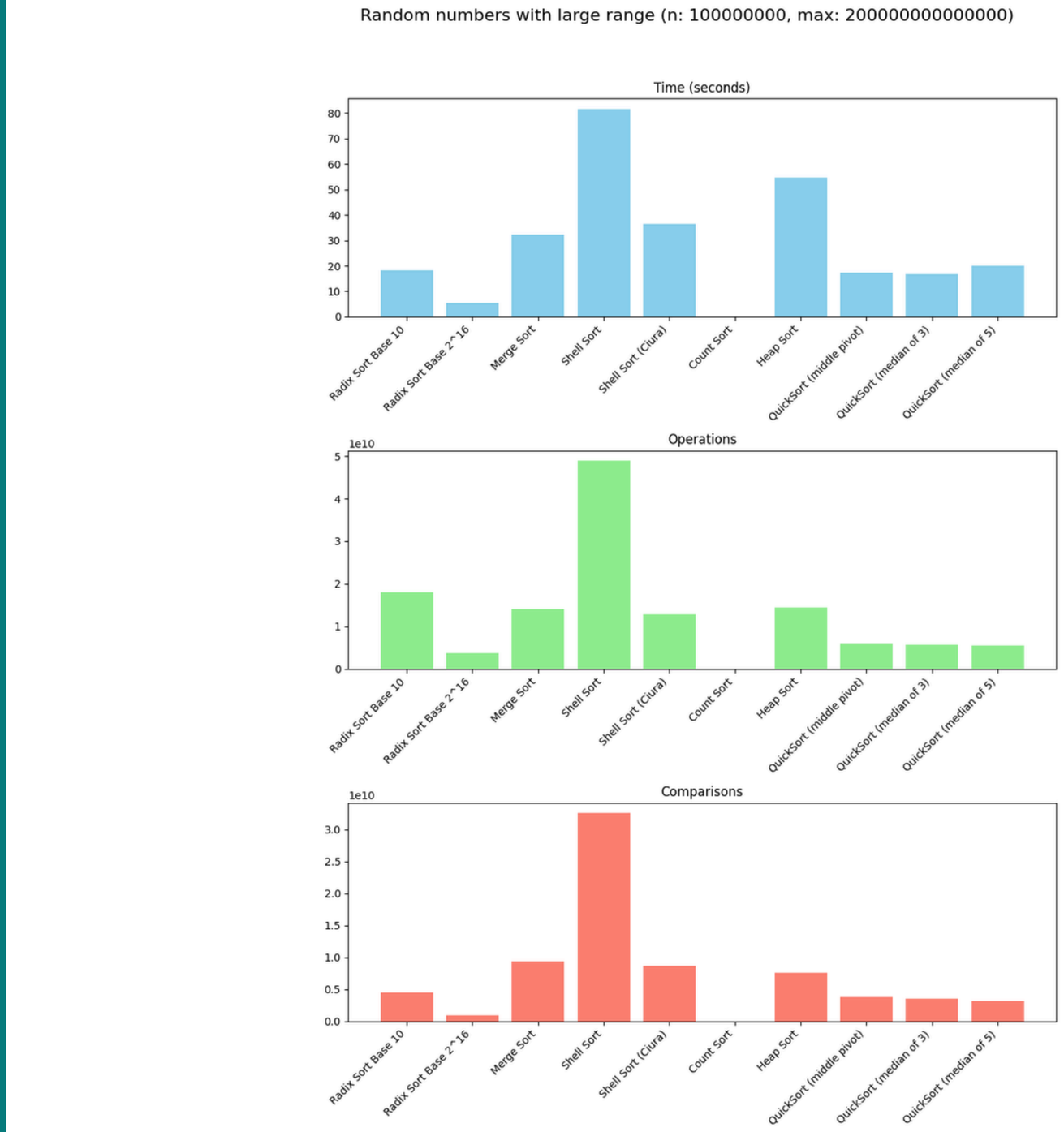
Test Case 7



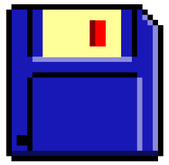


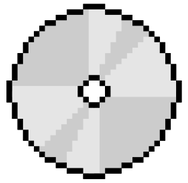

Small range random numbers (n: 100000000, max: 100)
Time (seconds)
0
5
10
15
20
25
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Operations
1e10
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Comparisons
1e9
0
2
4
6
8

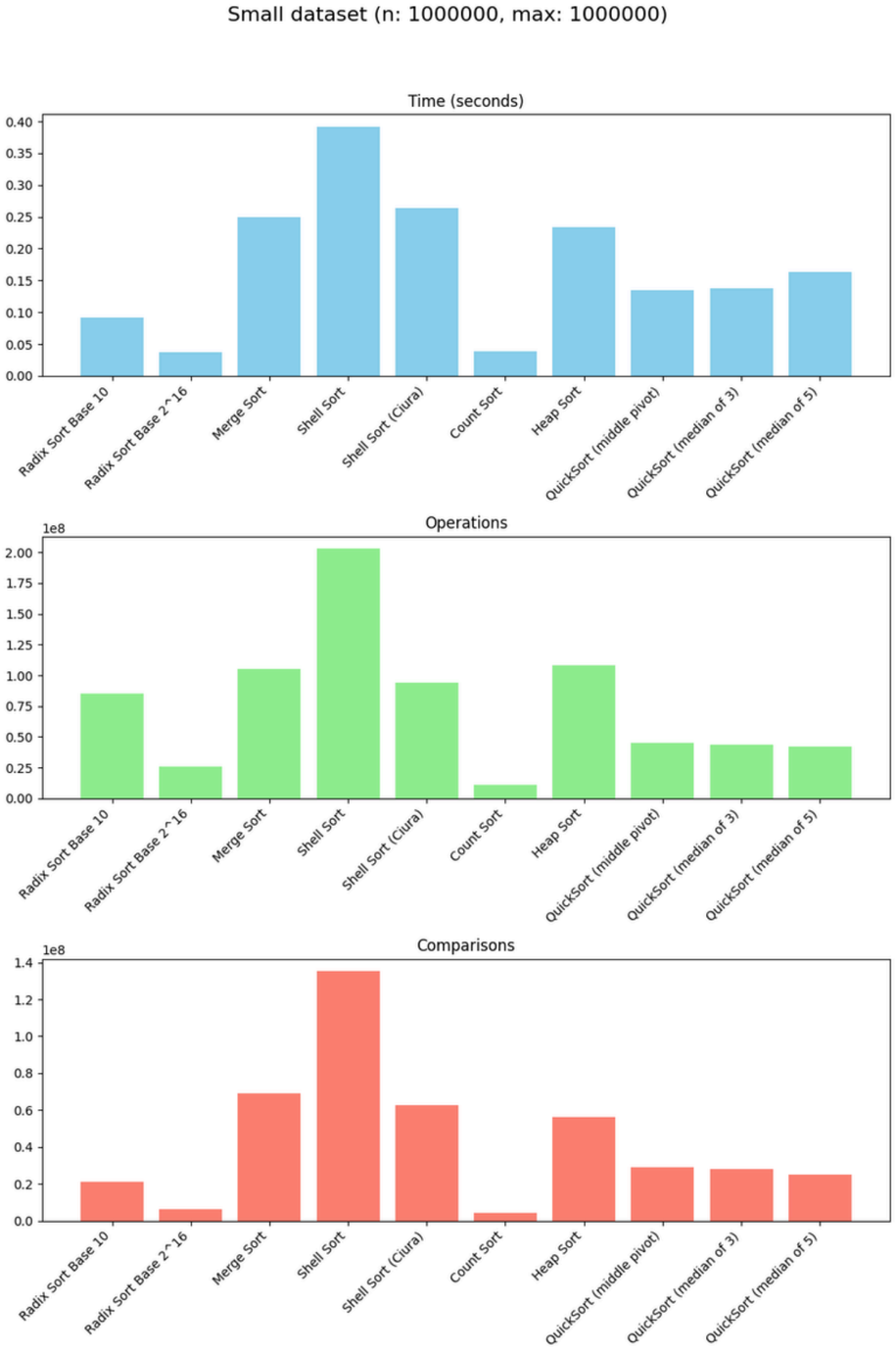
Test Case 8


Random numbers with large range (n: 100000000, max: 200000000000000)
Time (seconds)
Operations
1e10
Comparisons
1e10
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)

Test Case 9






Small dataset (n: 1000000, max: 1000000)
Time (seconds)
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Radix Sort Base 10
Radix Sort Base 2^16
Merge Sort
Shell Sort
Shell Sort (Ciura)
Count Sort
Heap Sort
QuickSort (middle pivot)
QuickSort (median of 3)
QuickSort (median of 5)
Operations
1e8
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Comparisons
1e8
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0

Concluzii

- Count Sort — clar campion când domeniul valorilor e mic (de ex. maxim ⩽100): număr minim de operații și zero comparații. În rest, când k (range-ul) crește, viteza lui scade rapid și devine ineficient față de celelalte.
- Radix Sort — în majoritatea scenariilor (numere cu range mare, random, sortate/pre-sortate) iese pe primul loc dintre algoritmii lin–log sau liniari, datorită faptului că nu face comparații și rulează în O(n·d).
- Baza 2^{16} este puțin mai rapidă decât baza 10 (mai puține treceri), dar costă spațiu auxiliar mai mare pentru buckets.
- Merge Sort — constanță absolută: același număr de comparații/operații indiferent de ordinea intrării, dar, din cauza copierii în buffer, nu poate ține pasul cu Radix/Count la n foarte mare.
- Heap Sort — la fel stabil pe orice tip de date, dar cu overhead ridicat la menținerea heap-ului; e tot timpul mai lent decât Merge Sort (și de 2–3× mai lent decât QuickSort median-of-5).
- Shell Sort (Ciura) — reduce masiv costul față de Shell original, dar în continuare mult peste n·logn; poate fi util când nu doriți memorie auxiliară, însă rar e alegerea optimă pentru n mari.
- QuickSort pivot fix (mijloc) — sensibil la intrări aproape sortate, degradează către $O(n^2)$; de aceea îl vedem foarte rapid pe random, dar foarte lent pe increasing/decreasing.
- QuickSort median-of-3 și median-of-5 — cele mai echilibrate variante de QuickSort:
- Supraviețuiesc bine la vectori deja sortați (evită cazurile degenerative).
- Au comparări puțin mai multe față de pivotul fix, dar operații generale mai puține decât Merge/Heap.
- Sensibilitate la ordine
- Sortările comparativ-bazate (Quick, Merge, Heap) rămân în jurul lui c·n·logn de comparații, cu variații mici.
- Algoritmii non-comparativi (Count, Radix) nu simt ordinea inițială, doar dimensiunea și gama valorilor.
- Dataset mic ($n \approx 10^6$)
- Overhead-ul algoritmilor lin–log (Merge, Heap, Quick) e mult mai vizibil: la valori mici, avantajul Count/Radix dispare parțial (număr de treceri multiplu din n) și scapă în fața QuickSort median-of-5.

[Back to Main Page](#)

In concluzie...

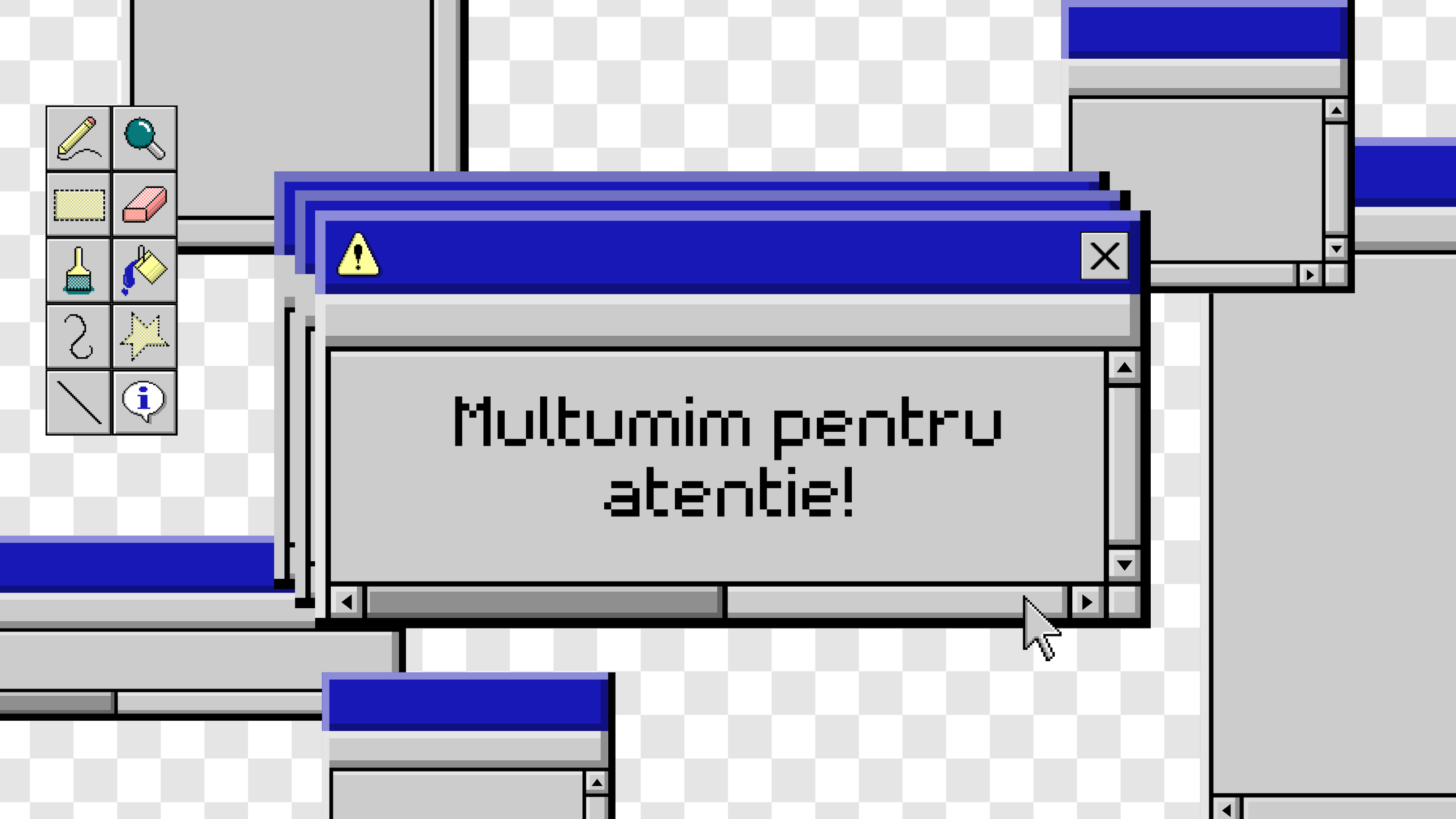
1. Dacă ai range foarte mic → Count Sort.

2. Dacă ai range mediu-mare → Radix Sort (preferabil cu bază 2^6).

3. Dacă nu poți folosi memorie auxiliară mare → QuickSort median-of-5 sau Shell Ciura.

4. Când ai nevoie de predictibilitate totală (date critice) → Merge Sort sau Heap Sort.

Back to Agenda Page


Multumim pentru
atentie!