OiAK - Projekt
Optymalizacja zużycia pamięci RAM w komputerach PC

Autorzy: Filip Przygoński, Agnieszka Jurijków

Politechnika Wrocławska
Wydział Elektroniki
Prowadzący: mgr inż. Tomasz Serafin

1 Wstęp

Nasz projekt miał na celu sprawdzić jak różne języki programowania, struktury danych (tablica, lista, wektor etc.) oraz różne algorytmy sortowania wpływają na użycie pamięci RAM. Przetestowaliśmy również szybkość algorytmów sortowania.

Wykorzystane algorytmy:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort - wersja rekurencyjna
- Quick Sort - wersja iteracyjna
- Top Down Merge Sort (rekurencyjny)
- Bottom Up Merge Sort - (iteracyjny)
- Shell Sort
- Heap Sort
- Standardowy algorytm sortowania danego języka (dalej w sprawozdaniu nazywany LibrarySort)

Wykorzystane języki programowania + struktury:

- **C++** - tablica, vector
- **C#** - tablica, lista
- **Java** - tablica, ArrayList
- **Python** - lista

Pomiary były prowadzone dla 3 różnych zestawów danych (10k, 25k, 50k) wcześniej wylosowanych i takich samych dla każdego języka.
Pomiary czasu były powtarzane 10 razy.
Pomiary RAMu wykonywane były tylko jednokrotnie.

2 O użytych językach

2.1 C++

- kompilowany do kodu natywnego
- bezpośrednie zarządzanie pamięcią
- statycznie typowany

2.2 C#

- kompilowany do kodu bajtowego CLI wykonywanego przez maszynę wirtualną .NET
- zarządzanie pamięcią - garbage collector
- statycznie typowany

2.3 Java

- kompilowana do kodu bajtowego Javy wykonywanego przez maszynę wirtualną JVM
- zarządzanie pamięcią - garbage collector
- statycznie typowana

2.4 Python

- interpretowany
- zarządzanie pamięcią - Python memory manager
- dynamicznie typowany

3 Użyte oprogramowanie

Do napisania programów użyliśmy zintegrowanych środowisk programistycznych:

- Visual Studio 2019 (C++, C#)
- IntelliJ IDEA (Java)
- PyCharm (Python)

Do mierzenia RAMu wykorzystaliśmy program ProcessExplorer.

4 Metody pomiaru czasu

W programach mierzyliśmy jak najdokładniejszą metodą, natomiast dla usystematyzowania wyników wyniki końcowe podajemy w mikrosekundach.

4.1 C++

Do mierzenia czasu w C++ wykorzystaliśmy high_resolution_clock z biblioteki ‘chrono’, który mierzy czas z dokładnością nanosekund.

Listing 1: Przykładowy kod pomiaru, dla przykładu wykorzystano wektor oraz algorytm selection sort

```
#include <chrono>

using namespace std;

static int repNr = 10;
//[...]
void testTime(vector<int> arr)
{
        vector<int> copy(arr);
        auto start = chrono::high_resolution_clock::now();
        auto finish = chrono::high_resolution_clock::now();
        long selectionTime = 0;
        for (int i = 0; i < repNr; i++) {
        copy = arr;
        start = chrono::high_resolution_clock::now();
        Sort::selectionSort(copy);
        finish = chrono::high_resolution_clock::now();
        selectionTime += chrono::duration_cast<chrono::nanoseconds>
        (finish - start).count() / 1000;
        }
        ofstream myfile;
        myfile.open("Vector.txt");
        myfile << selectionTime / repNr << endl;
```

4.2 C#

Do mierzenia czasu w C# wykorzystaliśmy klasę Stopwatch z przestrzeni nazw System.Diagnostics, która jest w stanie mierzyć czas z dokładnością nanosekund.

Listing 2: Przykładowy kod pomiaru, dla przykładu wykorzystano listę oraz algorytm bubble sort

```
using System.Diagnostics
// [...]
static void ListTimeTest(List<int> list)
    {
        int repeats = 10;
        Stopwatch stopwatch = new Stopwatch();
        long nanosecondsPerTick =
        (1000L * 1000L * 1000L) / Stopwatch.Frequency;
        long bubbleTime = 0;
        for (int i = 0; i < repeats; ++i)
        {
            var listCopy = new List<int>(list);
            stopwatch.Start();
            Sorting.BubbleSort(listCopy);
            stopwatch.Stop();
            bubbleTime += stopwatch.ElapsedTicks * nanosecondsPerTick;
        }
        // / 1000 poniewaz wynik jest w nanosekundach
        bubbleTime /= (repeats * 1000);
        Console.WriteLine(bubbleTime);
    }
```

4.3 Java

Do mierzenia czasu w Javie wykorzystaliśmy funkcję System.nanoTime(), mierzącą czas w nanosekundach.

Listing 3: Przykładowy kod pomiaru, dla przykładu wykorzystano tablicę oraz algorytm top down merge sort

```
static int repNr = 10;

static void testTime(int arr[]) {
    int copy[];
    long startTime;
    long topDownMergeSortTime = 0;
        for (int i = 0; i < repNr; i++) {
            copy = arr.clone();
            startTime = System.nanoTime();
            Sort.topDownMergeSort(copy);
            topDownMergeSortTime += System.nanoTime() - startTime;
        }
        try {
            FileWriter myWriter = new FileWriter("JavaArray.txt");
            myWriter.write((double) topDownMergeSortTime / repNr + "\n");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
```

4.4 Python

Do mierzenia czasu w Pythonie wykorzystaliśmy ‘perf_counter_ns’ z modułu ‘time’, który jest w stanie mierzyć czas z dokładnością nanosekund.

Listing 4: Przykładowy kod pomiaru, dla przykładu wykorzystano algorytm insertion sort

```
from time import perf_counter_ns as timer
# [...]
# arr — stworzona wczesniej lista
def time_test(arr):
    repeats = 10
    insertion_time = 0
    for i in range(repeats):
        arr_copy = list.copy(arr)
        start = timer()
        sorting.insertion_sort(arr_copy)
        end = timer()
        insertion_time += end — start
    # / 1000 bo wynik domyslnie jest w nanosekundach
    insertion_time /= (repeats * 1000)
    with open("Python_results.txt", "w") as file_results:
        file_results.write(str(insertion_time) + '\n')
```

5 Metody pomiaru RAMu

RAM mierzyliśmy za pomocą programu ProcessExplorer.
Napisane programy pozwalały nam dokładnie określić moment, w którym odbywało się sortowanie - działo się to po naciśnięciu entera. Uznaliśmy, że najbardziej miarodajne będzie mierzenie RAMu przy osobnym uruchomieniu programu dla każdego algorytmu. Wyniki ustosunkowywaliśmy do wartości początkowej (przed pierwszym naciśnięciem entera) z pierwszego pomiaru dla konkretnej ilości elementów i konkretnej struktury - to znaczy, że jeśli używany przez program RAM się nie zmienił po włączeniu algorytmu, wpisywaliśmy tą wartość początkową. Jeśli używany RAM podniósł się o jakąś wartość - wpisywaliśmy zapisaną początkową wartość powiększoną o zaobserwowany wzrost. Uznaliśmy to za najbardziej miarodajną metodę, ponieważ program przy włączeniu, mimo dokładnie takiej samej struktury używał różnej ilości RAMu.

Listing 5: Przykładowy kod pomiaru w C#, dla przykładu wykorzystano tablicę oraz domyślny algorytm sortowania C#

```
int[] array = loadArrayFromFile("plik");
int[] arrayCopy = new int[array.Length];
Array.Copy(array, arrayCopy, array.Length);
Console.WriteLine("Enter");
Console.ReadLine();
Array.Sort(arrayCopy);
Sorting.QuickSortRecursive(listCopy);
Console.WriteLine("End\n");
Console.ReadLine();
```

6 Wykorzystane algorytmy sortowania przedstawione w języku Python

Źródła algorytmów:

- https://en.wikipedia.org/wiki/Bubble_sort#Optimizing_bubble_sort
- https://en.wikipedia.org/wiki/Heapsort#Pseudocode
- https://en.wikipedia.org/wiki/Insertion_sort#Algorithm
- https://en.wikipedia.org/wiki/Merge_sort#Top-down_implementation
- https://www.geeksforgeeks.org/merge-sort/
- https://en.wikipedia.org/wiki/Merge_sort#Bottom-up_implementation
- https://www.geeksforgeeks.org/iterative-merge-sort/
- Cormen - Introduction to Algorithms - 7.1 - rekurencyjny quicksort
- https://www.geeksforgeeks.org/iterative-quick-sort/
- https://en.wikipedia.org/wiki/Selection_sort#Implementations
- https://en.wikipedia.org/wiki/Shellsort#Pseudocode

Listing 6: Algorytm Bubble Sort

```
def bubble_sort(arr: list):
    n = len(arr)
    while n > 1:
        new_n = 0
        for i in range(1, n):
            if arr[i - 1] > arr[i]:
                arr[i - 1], arr[i] = arr[i], arr[i - 1]
                new_n = i
        n = new_n
```

Listing 7: Algorytm Heap Sort

```
def heap_sort(arr: list):
    len_arr = len(arr)
    for i in range((len_arr - 2) // 2, -1, -1):
        j = i
        while j != -1:
            j = sorting_helper.heap_sort_heapify_down(arr, j, len_arr)
    for i in range(len_arr - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        j = 0
        while j != -1:
            j = sorting_helper.heap_sort_heapify_down(arr, j, i)
```

Listing 8: Metoda pomocnicza Heap Sort: heapify

```
def heap_sort_heapify_down(arr: list, parent_index: int, len_arr: int) -> int:
    largest_index = parent_index
    left_index = 2 * parent_index + 1
    right_index = 2 * parent_index + 2
    if left_index < len_arr and arr[left_index] > arr[largest_index]:
        largest_index = left_index
    if right_index < len_arr and arr[right_index] > arr[largest_index]:
        largest_index = right_index
    if largest_index != parent_index:
        arr[parent_index], arr[largest_index] = arr[largest_index], arr[parent_index]
        return largest_index
    else:
        return -1
```

Listing 9: Algorytm Insertion Sort

```
def insertion_sort(arr: list):
    for i in range(1, len(arr)):
        x = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > x:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = x
        i += 1
```

Listing 10: Algorytm Top Down Merge Sort

```
def merge_sort_recursive(arr: list):
    len_arr = len(arr)
    if len_arr > 1:
        middle = len_arr // 2
        left_arr = arr[:middle]
        merge_sort_recursive(left_arr)
        right_arr = arr[middle:]
        merge_sort_recursive(right_arr)
        len_left = len(left_arr)
        len_right = len(right_arr)
        sorting_helper.merge_sort_merge(arr, left_arr, right_arr, 0, len_left, len_right
```

Listing 11: Algorytm Bottom Up Merge Sort

```
def merge_sort_iterative(arr: list):
    len_arr = len(arr)
    width = 1
    while width < len_arr:
        left = 0
        while left < len_arr:
            mid = min(left + width - 1, len_arr - 1)
            right = min(left + 2 * width - 1, len_arr - 1)
            len_left = mid - left + 1
            len_right = right - mid
            left_arr = [0] * len_left
            right_arr = [0] * len_right
            for i in range(0, len_left):
                left_arr[i] = arr[left + i]
            for i in range(0, len_right):
                right_arr[i] = arr[mid + i + 1]
            sorting_helper.merge_sort_merge(arr, left_arr, right_arr, left, len_left, le
            left += width * 2
        width *= 2
```

Listing 12: Metoda pomocnicza Merge Sort: merge

```
def merge_sort_merge(arr, left_arr, right_arr, arr_i, len_left, len_right):
    left_i, right_i = 0, 0
    while left_i < len_left and right_i < len_right:
        if left_arr[left_i] < right_arr[right_i]:
            arr[arr_i] = left_arr[left_i]
            left_i += 1
        else:
            arr[arr_i] = right_arr[right_i]
            right_i += 1
        arr_i += 1
    while left_i < len_left:
        arr[arr_i] = left_arr[left_i]
        left_i += 1
        arr_i += 1
    while right_i < len_right:
        arr[arr_i] = right_arr[right_i]
        right_i += 1
        arr_i += 1
```

Listing 13: Algorytm Quick Sort - wersja rekurencyjna

```
def quick_sort_recursive(arr: list, low: int, high: int):
    if low < high:
        pivot = quick_sort_partition(arr, low, high)
        quick_sort_recursive(arr, low, pivot - 1)
        quick_sort_recursive(arr, pivot + 1, high)
```

Listing 14: Algorytm Quick Sort - wersja iteracyjna

```
def quick_sort_iterative(arr: list):
    low = 0
    high = len(arr) - 1
    stack = deque([low, high])
    while stack:
        high = stack.pop()
        low = stack.pop()
        pivot = sorting_helper.quick_sort_partition(arr, low, high) - 1
        if pivot > low:
            stack.append(low)
            stack.append(pivot)
        pivot += 2
        if pivot < high:
            stack.append(pivot)
            stack.append(high)
```

Listing 15: Metoda pomocnicza Quick Sort: Quick Sort partition

```
def quick_sort_partition(arr: list, low: int, high: int) -> int:
    x = arr[high]
    pivot = low - 1
    for i in range(low, high):
        if arr[i] <= x:
            pivot += 1
            arr[pivot], arr[i] = arr[i], arr[pivot]
    pivot += 1
    arr[pivot], arr[high] = arr[high], arr[pivot]
    return pivot
```

Listing 16: Algorytm Selection Sort

```
def selection_sort(arr: list):
    len_arr = len(arr)
    for i in range(len_arr):
        min_index = i
        for j in range(i + 1, len_arr):
            if arr[j] < arr[min_index]:
                min_index = j
        if i != min_index:
            arr[i], arr[min_index] = arr[min_index], arr[i]
```

Listing 17: Algorytm Shell Sort

```
def shell_sort(arr: list):
    len_arr = len(arr)
    gap = len_arr // 2
    while gap > 0:
        for i in range(gap, len_arr):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
```

7 Wyniki pomiarów

		Python	C#		C++		Java	
		Lista	Tablica	Lista	Tablica	Vector	Tablica	ArrayList
10000	Stan początkowy	4480	7820	7900	1536	1588	350300	349356
	BubbleSort	4480	7820	7900	1536	1588	350580	349448
	InsertionSort	4480	7820	7900	1536	1588	350884	350928
	SelectionSort	4480	7820	7900	1536	1588	350636	350456
	QuickSort recursive	4480	7844	7932	1536	1628	351244	353276
	QuickSort iterative	4492	7840	7920	1536	1588	351444	353092
	TopDown MergeSort	4512	7848	7920	1536	1608	351716	350680
	BottomUp MergeSort	4500	7820	7928	1536	1632	351436	353740
	ShellSort	4480	7820	7900	1536	1588	350800	350956
	HeapSort	4480	7820	7900	1536	1588	350672	349436
	LibrarySort	4500	7824	7904	1536	1588	350488	349992

Rysunek 1: Pomiary RAMu dla 10000 liczb

		Python	C#		C++		Java	
		Lista	Tablica	Lista	Tablica	Vector	Tablica	ArrayList
10000	BubbleSort	15790,58	726,52	901,44	246,25	462,17	13,65	618,65
	InsertionSort	7449,86	224,50	302,88	63,18	1653,31	11,76	114,68
	SelectionSort	7476,06	307,70	412,83	124,75	2206,82	48,30	157,40
	QuickSort recursive	43,50	2,45	2,71	1,32	84,53	0,82	2,37
	QuickSort iterative	42,73	2,48	2,81	24,19	34,06	2,36	4,16
	TopDown MergeSort	58,32	2,22	2,29	1,70	12,60	1,00	3,07
	BottomUp MergeSort	107,14	2,07	2,09	8,19	12,14	1,68	5,30
	ShellSort	77,06	3,56	4,48	1,69	18,62	1,29	4,60
	HeapSort	125,06	5,43	6,38	3,69	25,10	1,39	5,87
	LibrarySort	2,72	0,58	0,56	14,62	14,49	1,01	3,14

Rysunek 2: Pomiary czasu dla 10000 liczb

W celu uniknięcia zbyt dużej ilości pomiarów wybraliśmy najbardziej optymalne pod względem użycia RAMu i czasu wykonywania algorytmów struktury (po jednej dla każdego języka) oraz trzy najoptymalniejsze algorytmy. Wybrane przez nas algorytmy i struktury są zaznaczone w powyższych tabelkach na zielono.

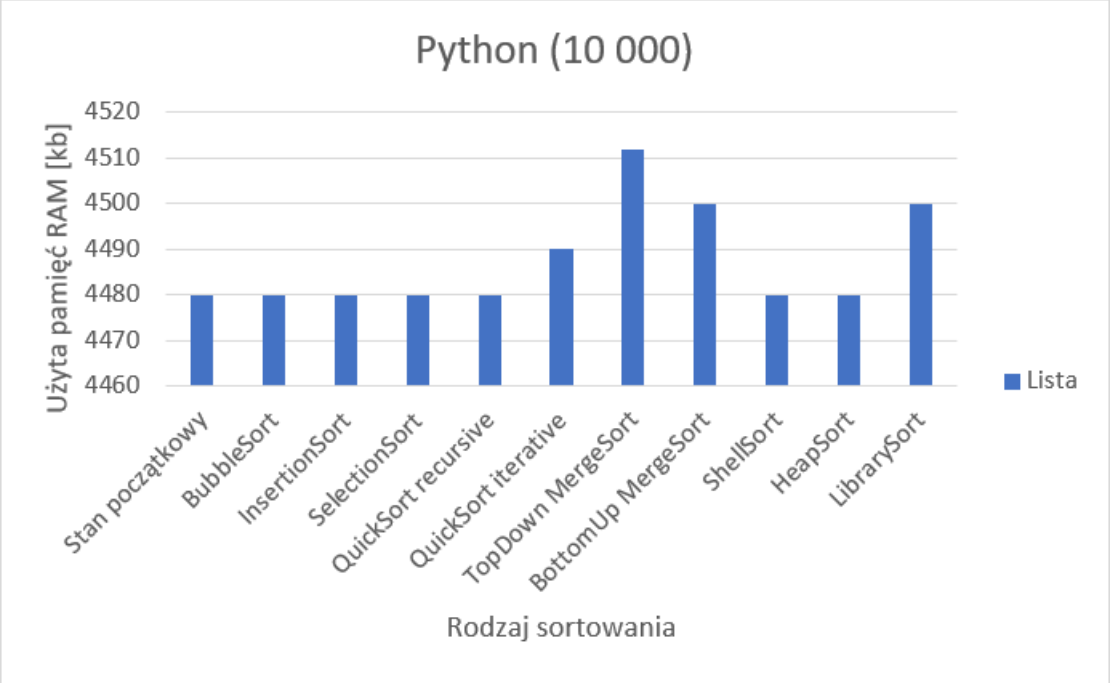
		Python	C#	C++	Java
10000	QuickSort recursive	4480	7844	1536	351244
	TopDown MergeSort	4512	7848	1536	351716
	LibrarySort	4500	7824	1536	350488
25000	QuickSort recursive	4872	9032	1652	352484
	TopDown MergeSort	5004	9048	1652	352936
	LibrarySort	4872	9016	1716	351720
50000	QuickSort recursive	5592	11080	1836	357256
	TopDown MergeSort	5592	11084	2036	357660
	LibrarySort	5588	11080	1836	356420

Rysunek 3: Pomiary RAMu dla wybranych struktur i algorytmów

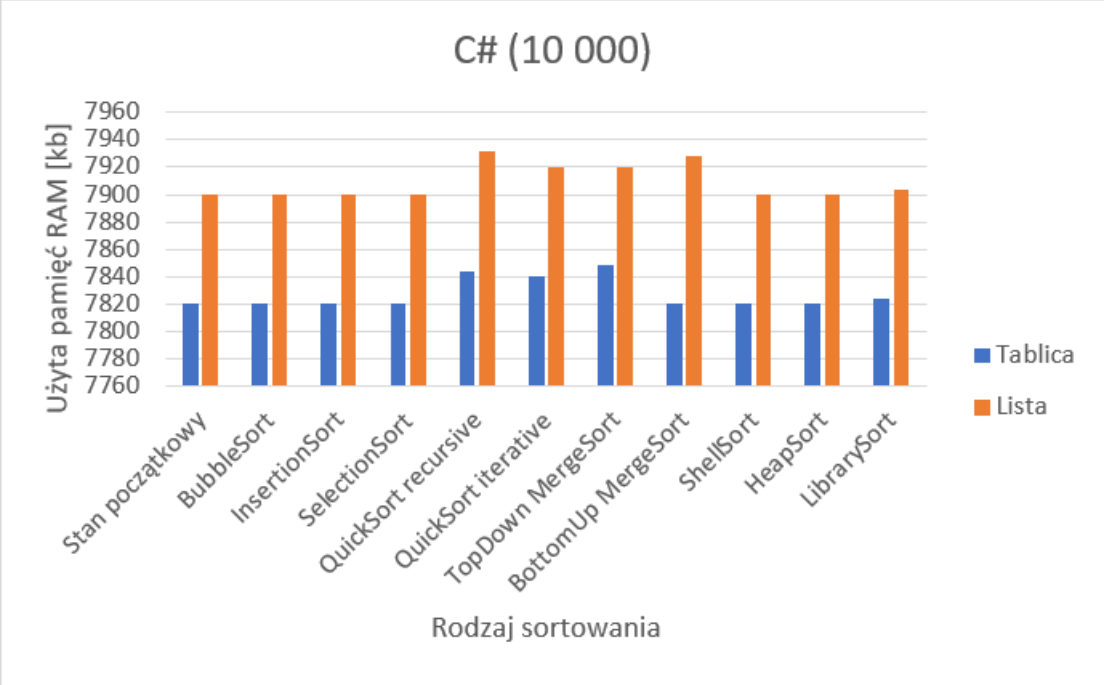
		Python	C#	C++	Java
10000	QuickSort recursive	43,50	2,45	1,32	0,82
	TopDown MergeSort	58,32	2,22	1,70	1,00
	LibrarySort	2,72	0,58	14,62	1,01
25000	QuickSort recursive	126,16	7,99	3,64	2,61
	TopDown MergeSort	174,52	7,41	4,85	3,00
	LibrarySort	7,65	1,73	43,17	3,60
50000	QuickSort recursive	279,55	16,94	8,15	5,33
	TopDown MergeSort	376,68	14,30	8,67	5,86
	LibrarySort	16,42	3,08	93,72	5,81

Rysunek 4: Pomiary czasu dla wybranych struktur i algorytmów

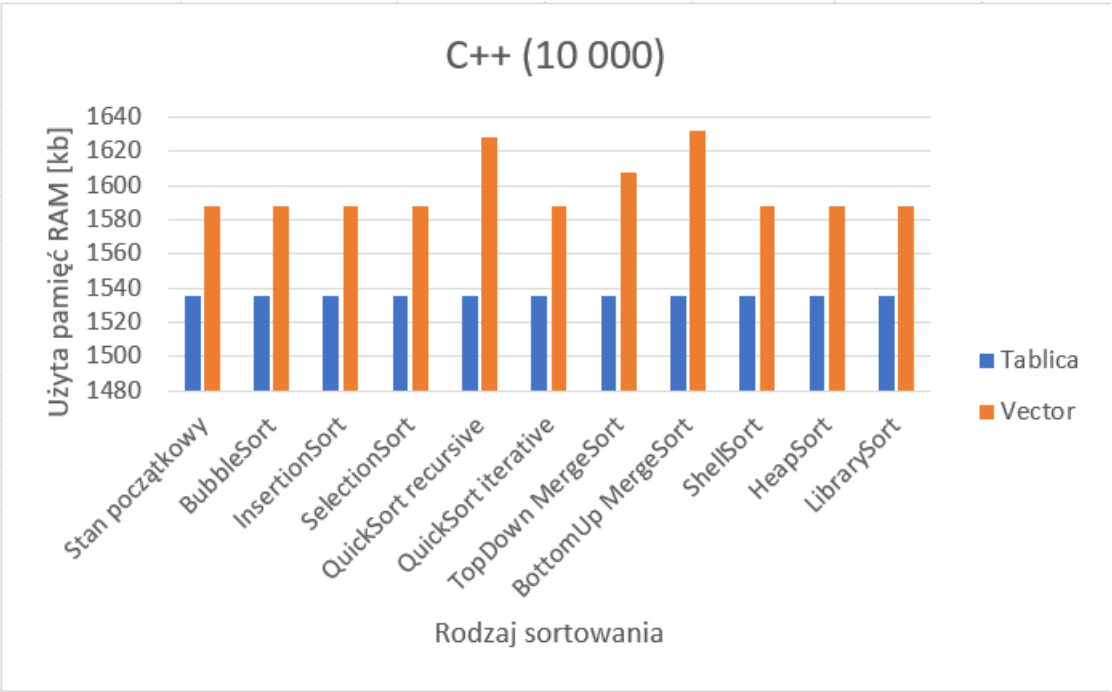
7.1 Pomiary RAMu dla 10000 liczb



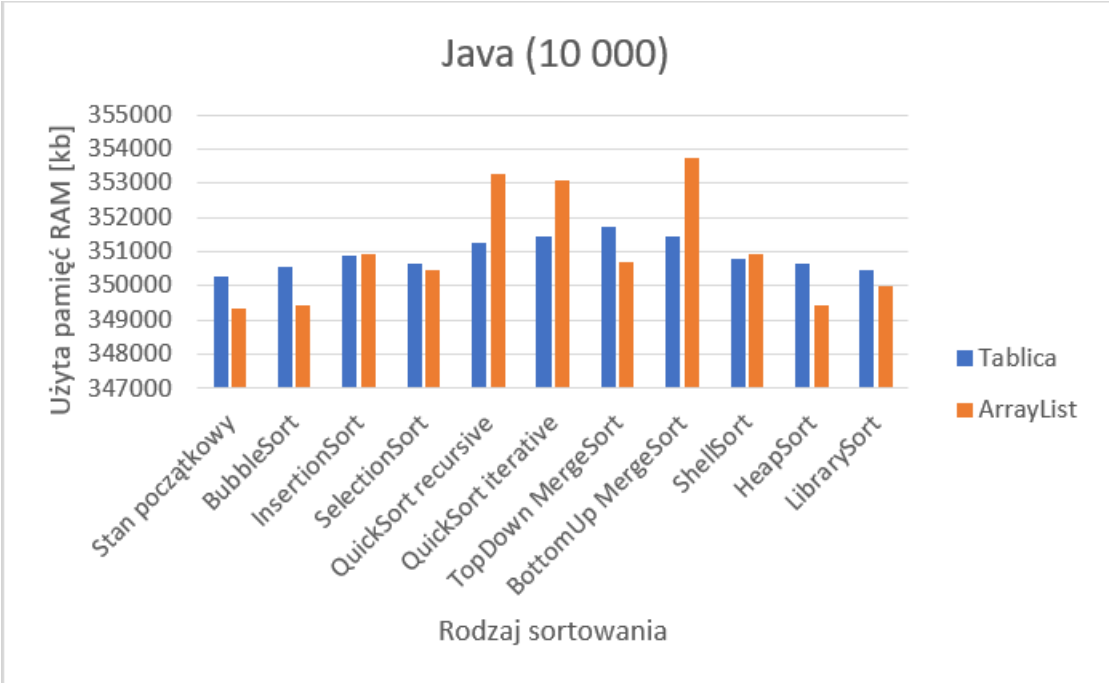
Rysunek 5: Python RAM



Rysunek 6: C# RAM

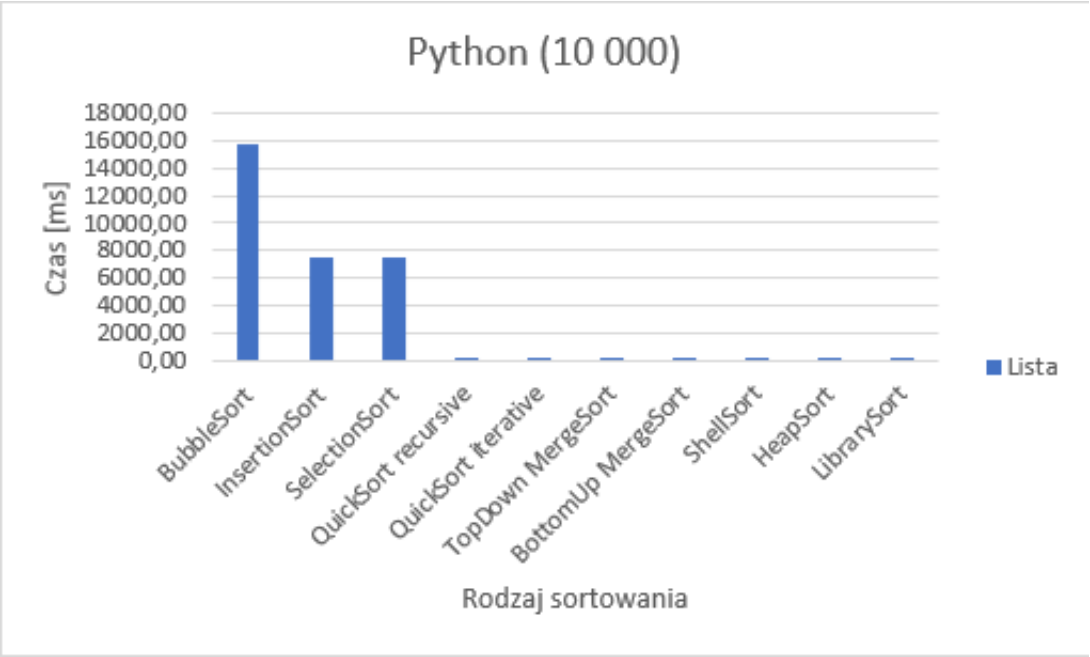


Rysunek 7: C++ RAM

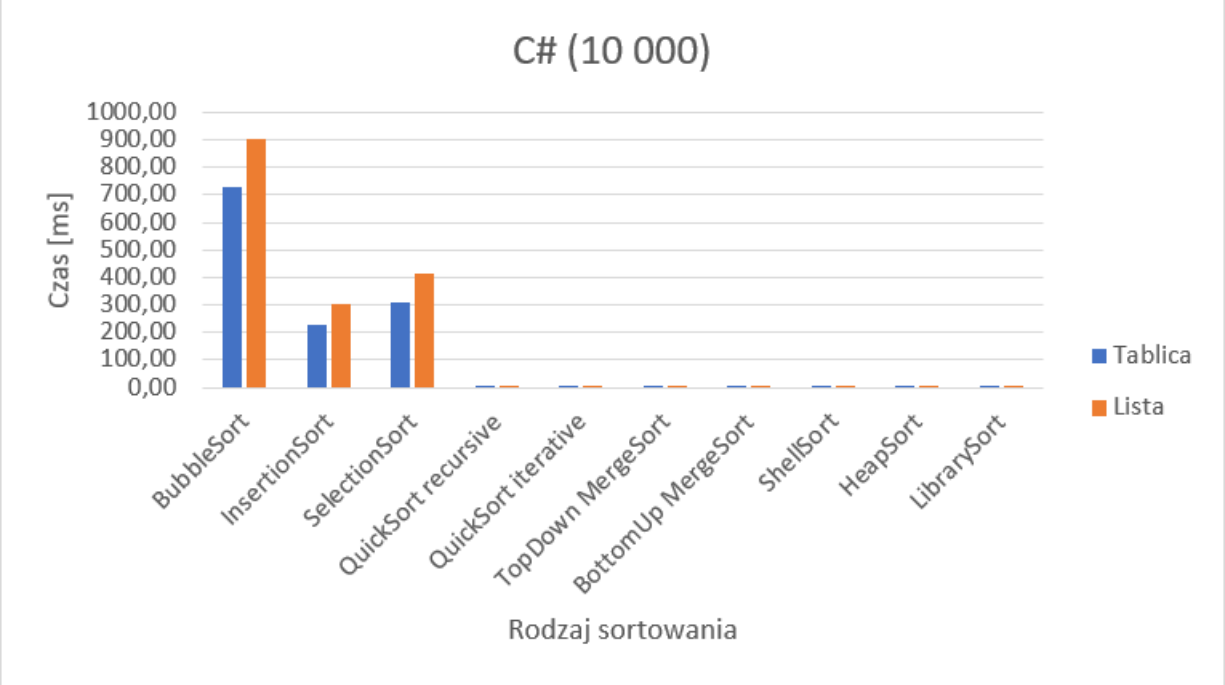


Rysunek 8: Java RAM

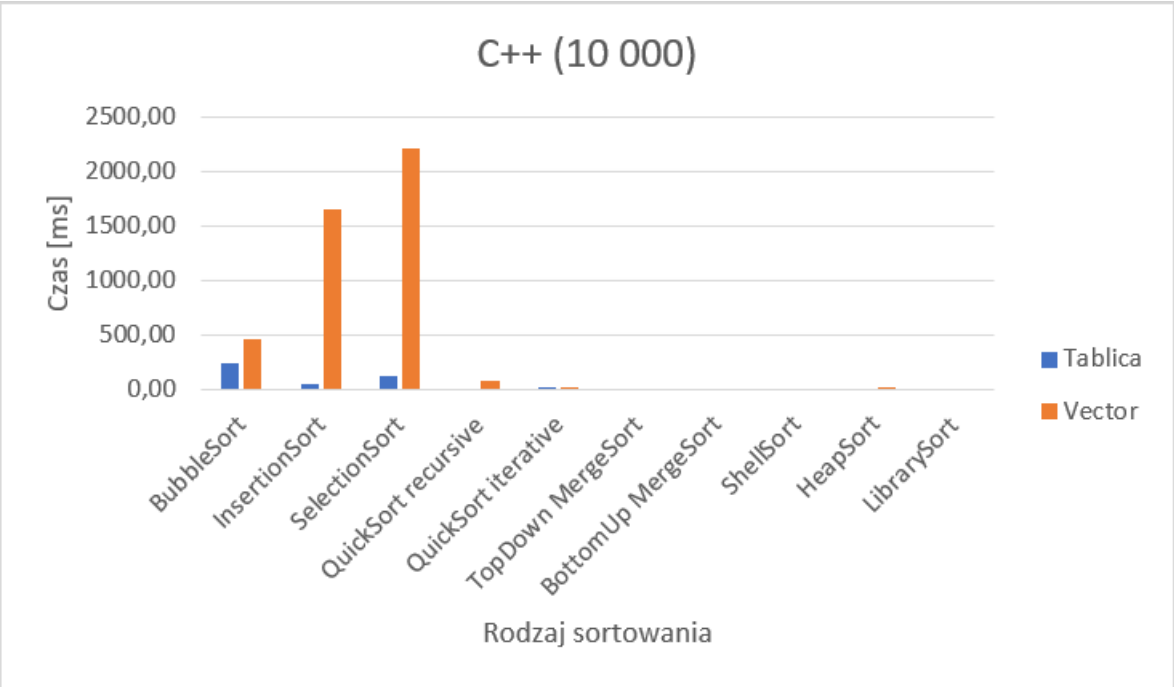
7.2 Pomiary czasu dla 10000 liczb



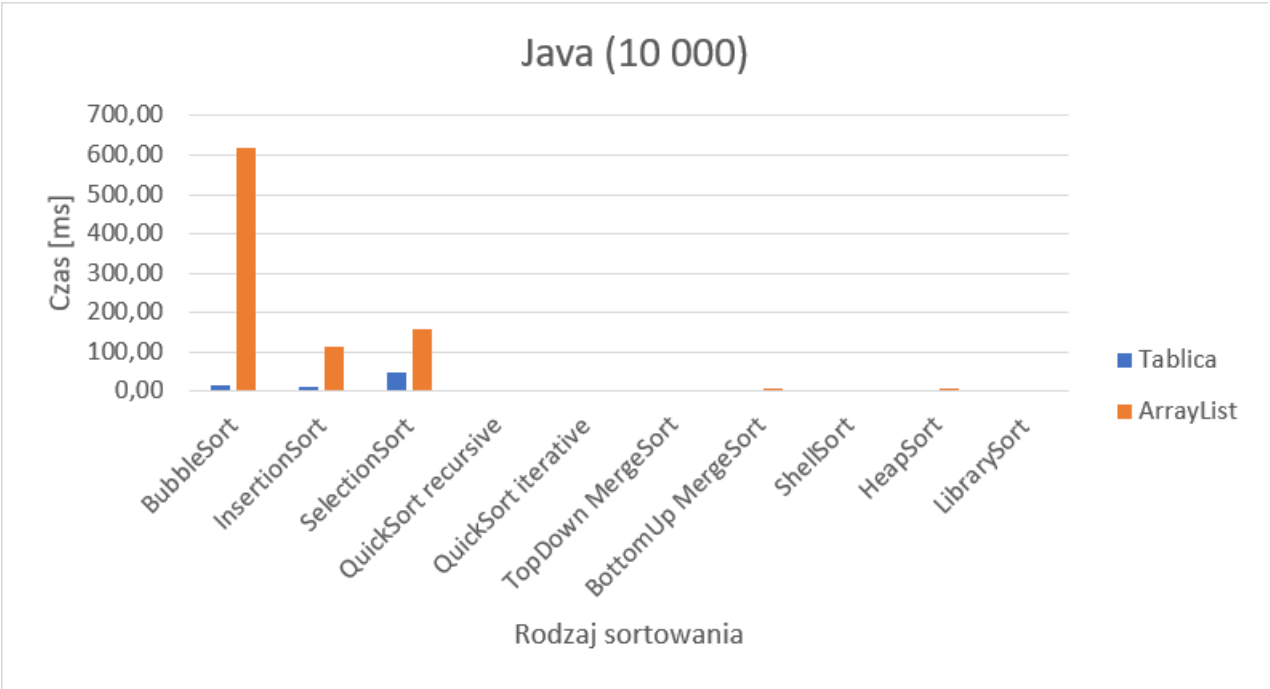
Rysunek 9: Python czas



Rysunek 10: C# czas

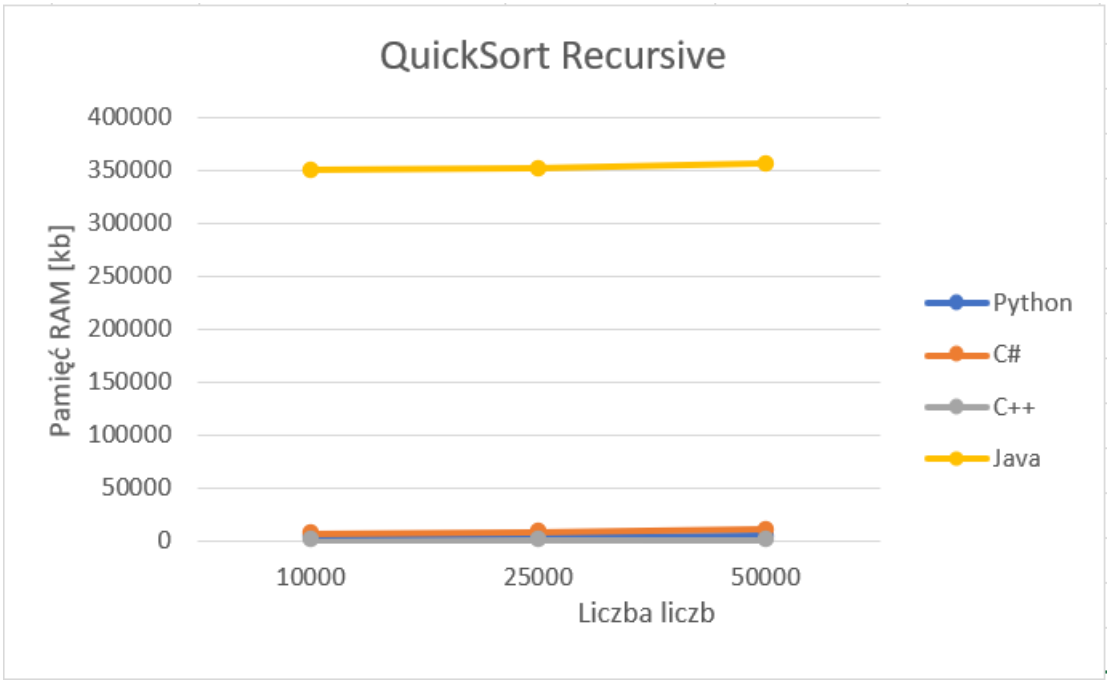


Rysunek 11: C++ czas

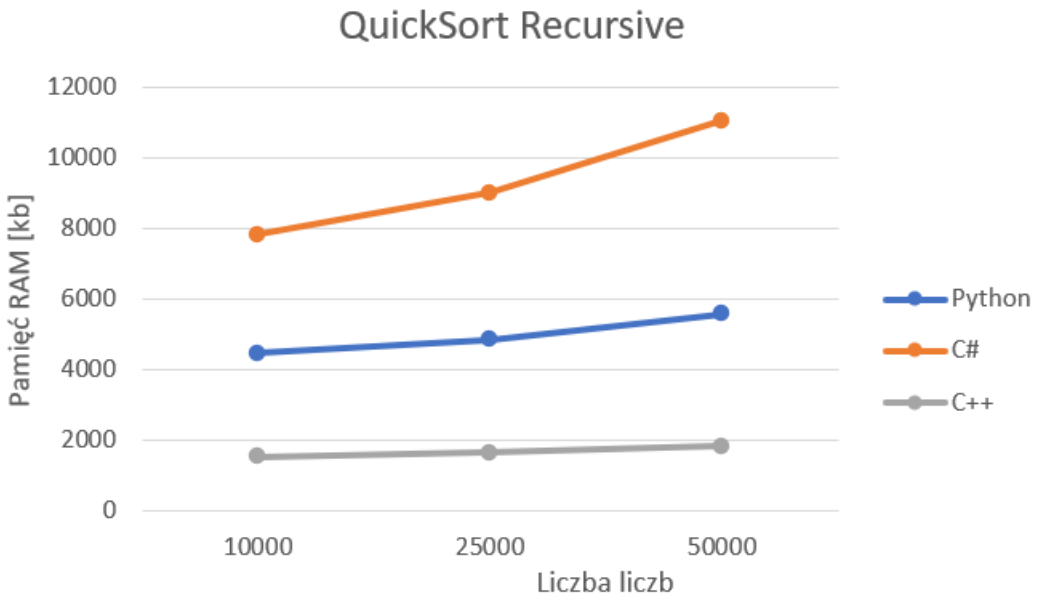


Rysunek 12: Java czas

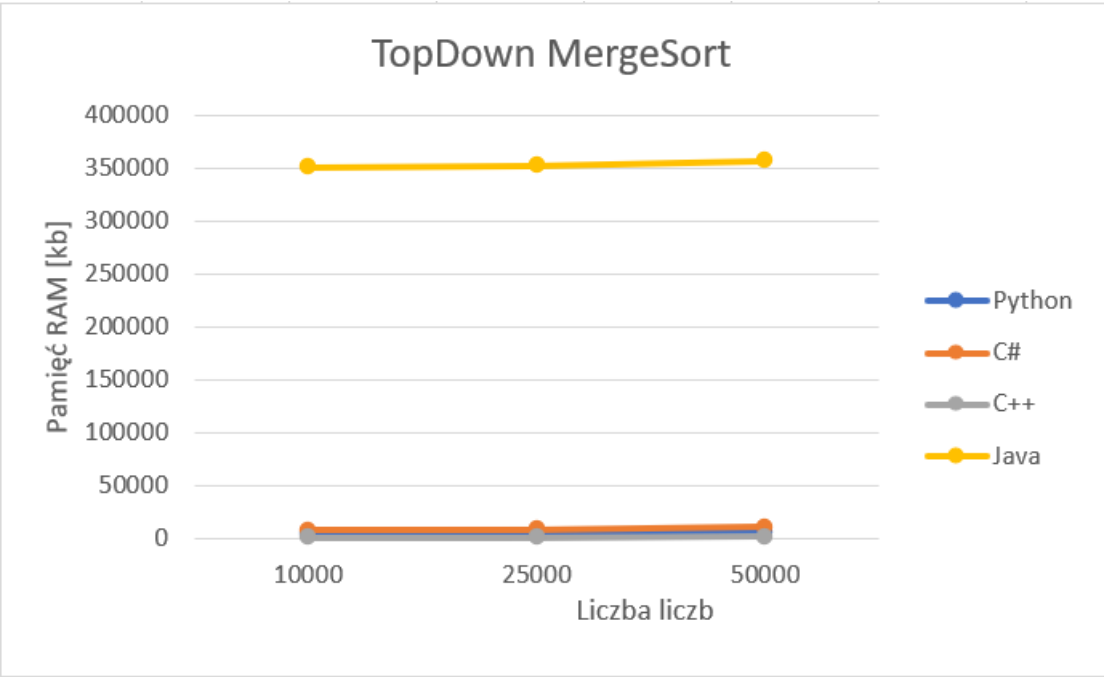
7.3 Pomiary RAMu dla wybranych algorytmów i struktur



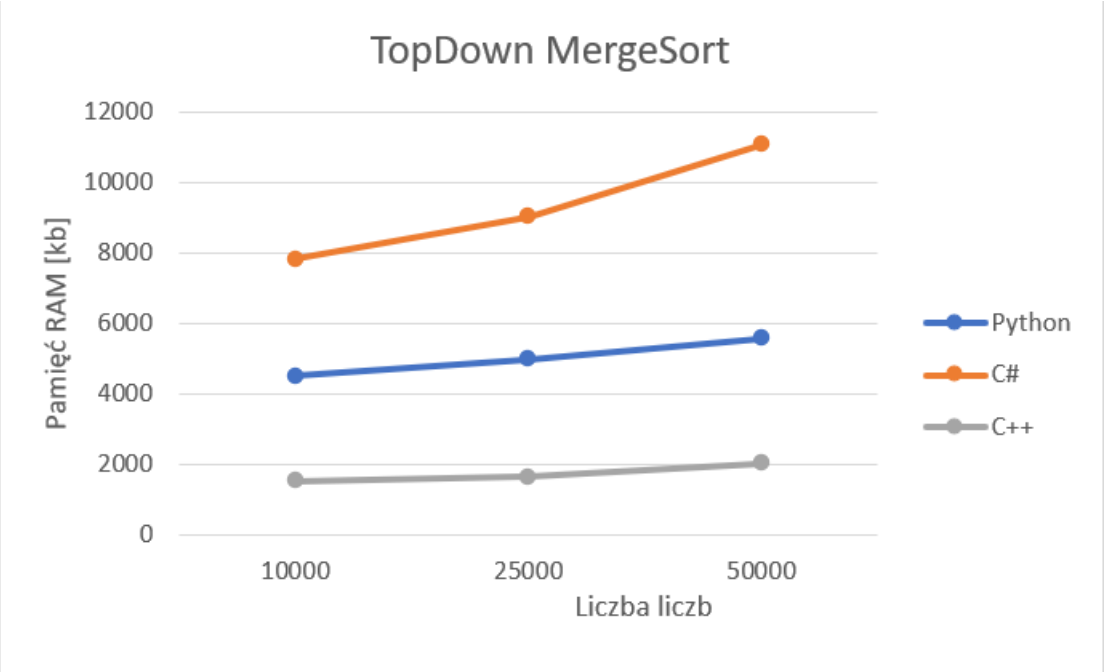
Rysunek 13: QuickSort RAM



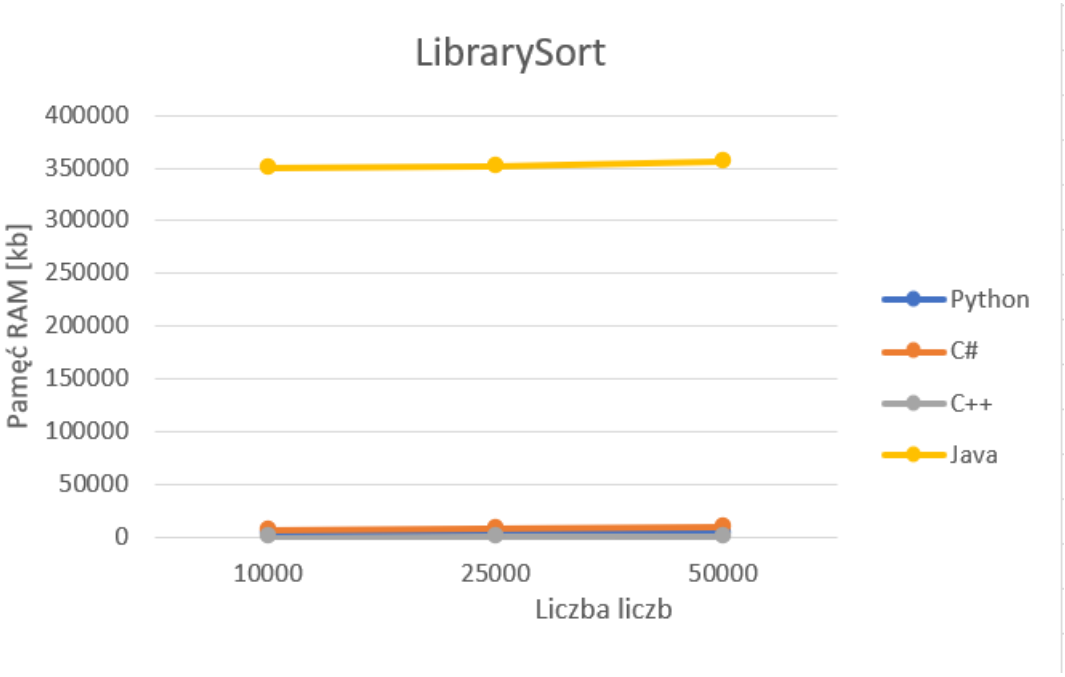
Rysunek 14: QuickSort RAM (bez Javy)



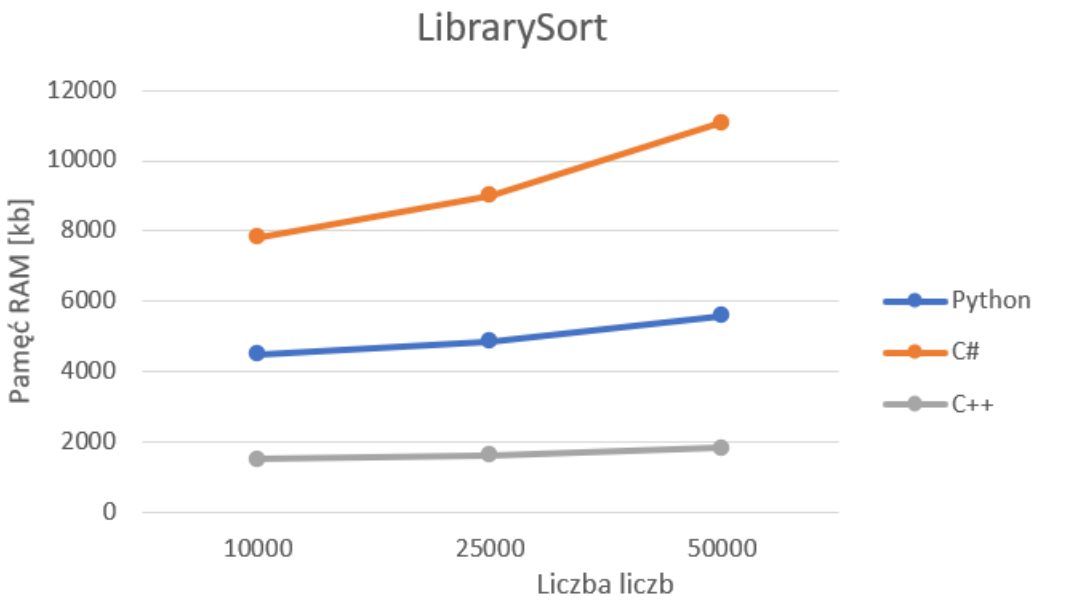
Rysunek 15: TopDown MergeSort RAM



Rysunek 16: TopDown MergeSort RAM (bez Javy)

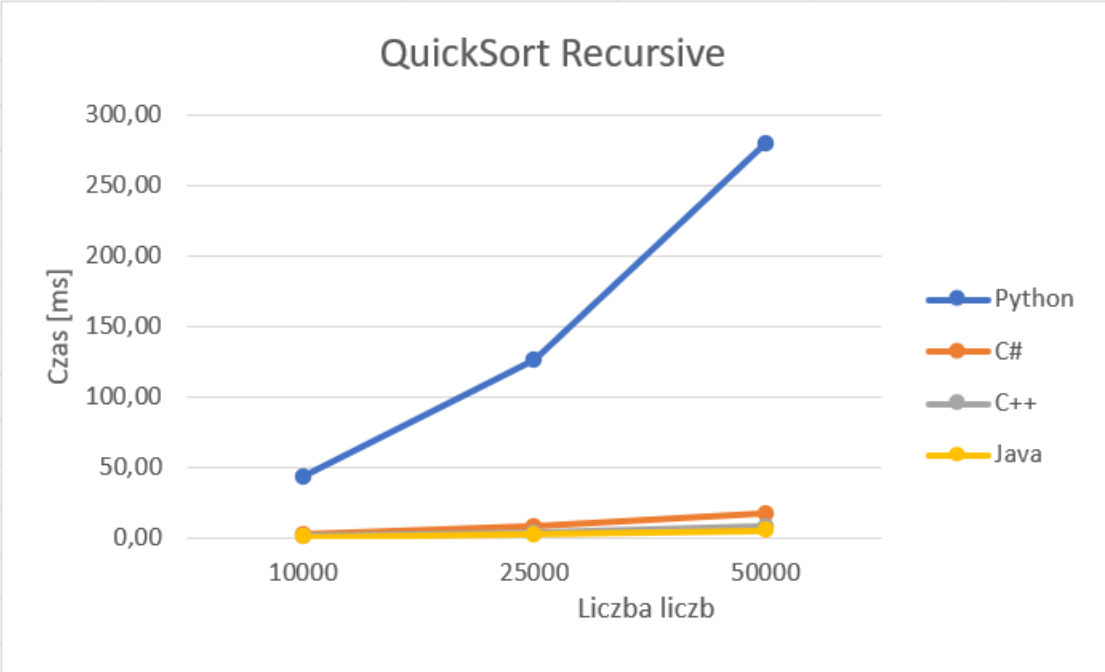


Rysunek 17: LibrarySort RAM

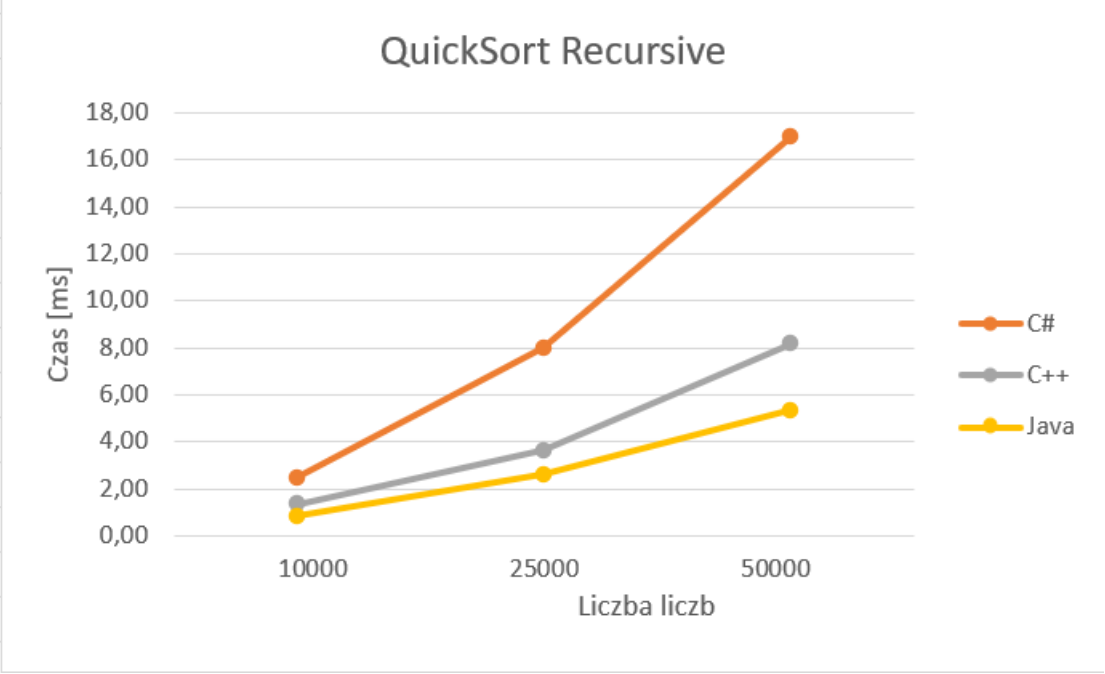


Rysunek 18: LibrarySort RAM (bez Javy)

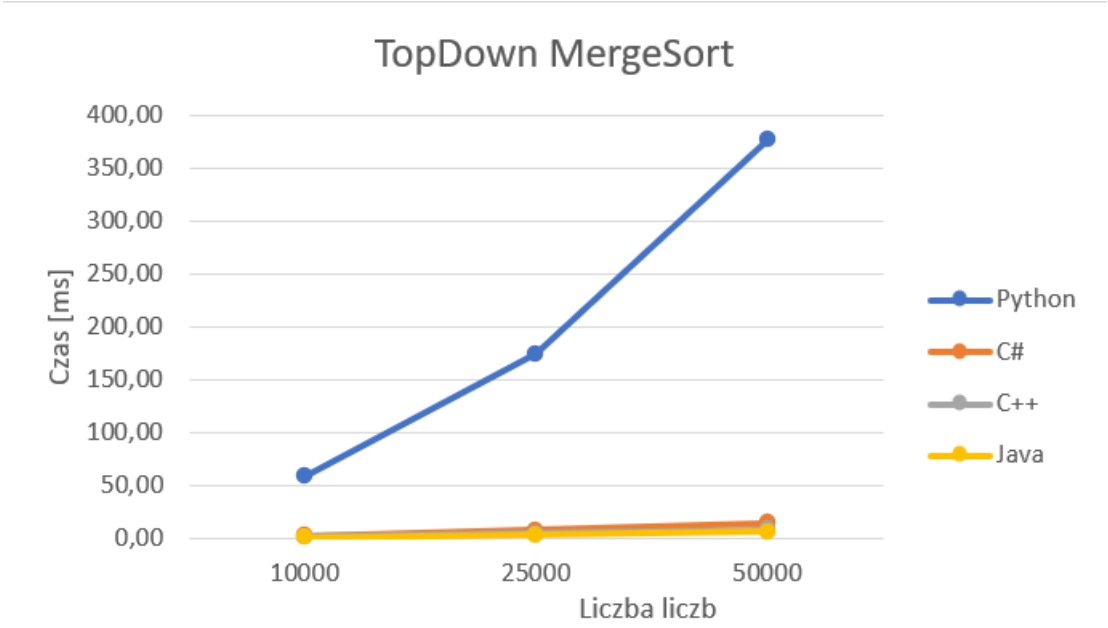
7.4 Pomiary czasu dla wybranych algorytmów i struktur



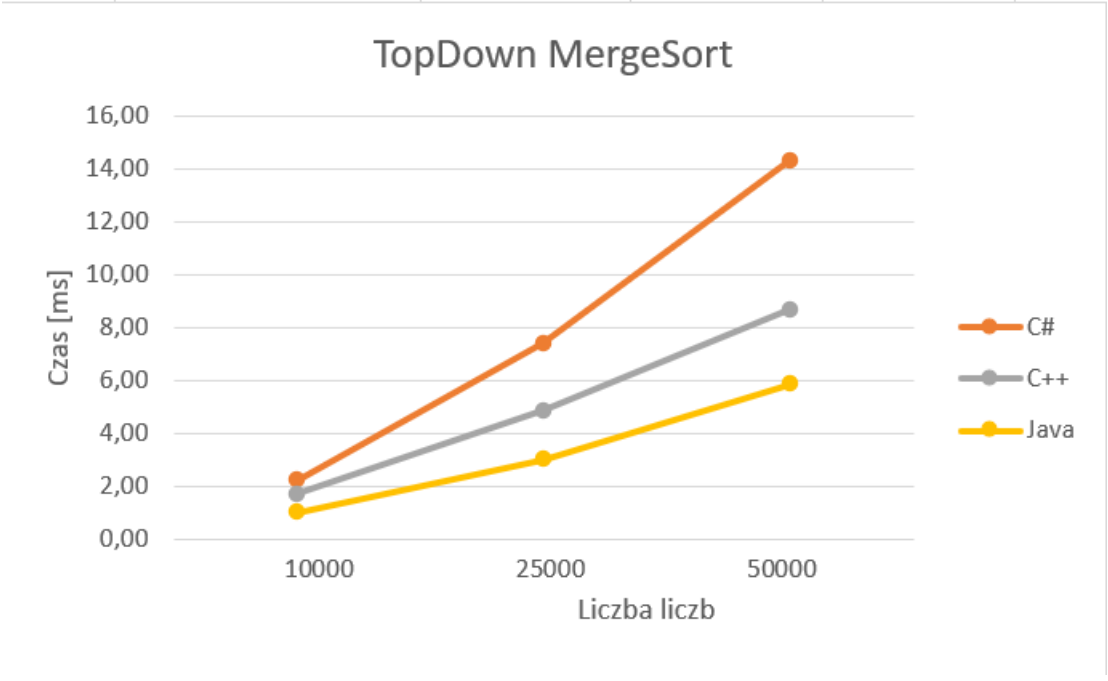
Rysunek 19: QuickSort czas



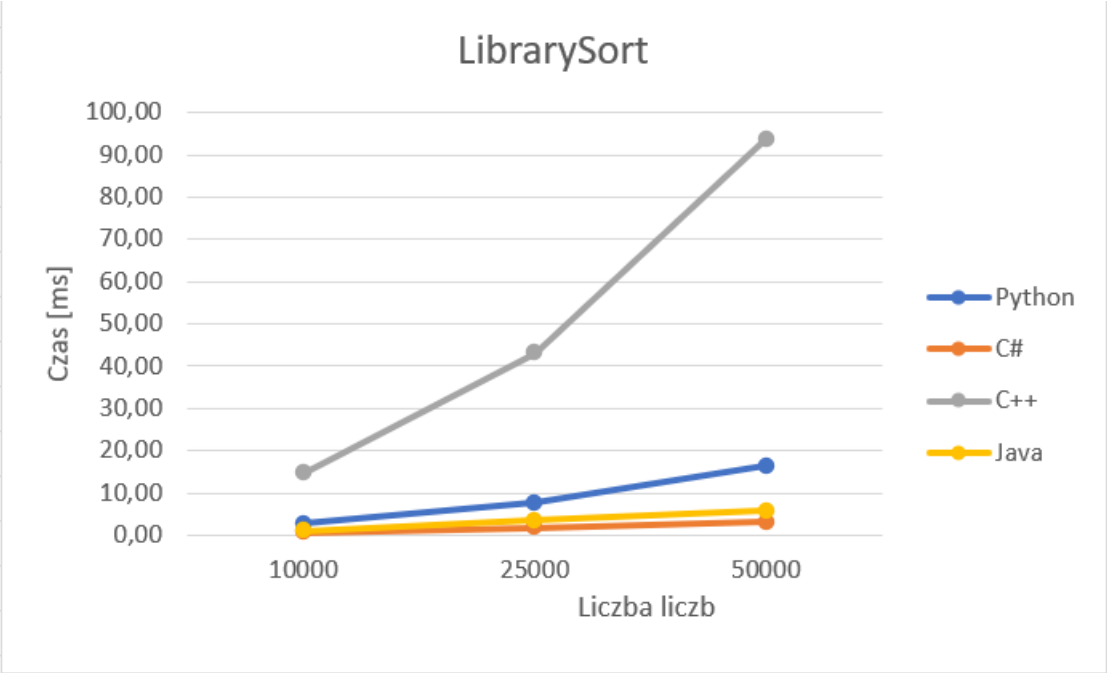
Rysunek 20: QuickSort czas (bez Pythona)



Rysunek 21: TopDown MergeSort czas



Rysunek 22: TopDown MergeSort czas (bez Pythona)



Rysunek 23: LibrarySort czas

8 Wnioski

Zaimplementowane algorytmy sortowania zazwyczaj miały minimalny wpływ na używany RAM (w skrajnych przypadkach rzędu kilkuset kb). Dużą różnicę w użyciu RAMu widać jednak często przy implementowanych strukturach. Zazwyczaj najlepszymi opcjami okazywały się zwykłe tablice, czyli najprostsze struktury dostępne w tych językach (poza Pythonem, w którym nie ma odpowiednika tablicy - jest lista). Okazywały się one również lepsze pod względem szybkości ich sortowania.

Pod względem użytego RAMu zdecydowanie najbardziej efektywnym językiem jest C++. Wynika to z tego, że C++ jest kompilowany do kodu natywnego, w przeciwieństwie do innych testowanych języków, które są uruchamiane przez maszynę wirtualną lub interpretowane.

Mało efektywny okazał się C#, ponieważ musi być on wykonywany poprzez maszynę wirtualną .NET. Podobną bolączke posiada Java która jest najmniej efektywnym pod tym względem językiem, używa około 200 razy więcej RAMu niż C++. Wczytuje ona sporo klas ze swojej standardowej biblioteki mimo że mogą być niepotrzebne oraz alokuje dużo pamięci dla programu z wyprzedzeniem. Java wypada bardzo słabo przy prostych programach (takich jak ten przez nas napisany), jednak przy bardziej złożonych projektach może okazać się porównywalnie dobrą opcją jak pozostałe języki, szczególnie dzięki jej wynikom czasowym.

Dla dwóch wybranych algorytmów (QuickSort rekurencyjny i Top-Down MergeSort) najlepszą okazuje się Java, chociaż nie ma takiej dużej różnicy między nią a C++ i C# jak w przypadku RAMu - jest ona szybsza o około pół milisekundy od C++ i o nieco ponad jedną milisekundę od C#. Mało efektywnym okazał się Python, w którym algorytmy wykonywały się około 20 razy dłużej niż w c#, który miał najgorsze czasy spośród trzech pozostałych języków. Wynika to z faktu, że użyta struktura - Pythonowa lista nie jest dobrą strukturą do sortowania. Najszybszym spośród wszystkich algorytmem okazał się podstawowy algorytm sortowania w C#.

Częstotliwość mierzenia RAMu była niska, przez co prawdopodobnie nie uzyskaliśmy najwiarygodniejszych wyników. Dodatkowo w Javie wyniki tak bardzo się wahały, że jedynym wnioskiem jaki z tych pomiarów możemy wyciągnąć jest fakt, że Java używa o wiele więcej RAMu od innych języków. Z uzyskanych pomiarów również nie wynika, która ze struktór jest bardziej efektywna w użyciu RAMu.

9 Źródła

- C++:
  https://en.wikipedia.org/wiki/C%2B%2B
- C#:
  https://en.wikipedia.org/wiki/C_Sharp_(programming_language)
- Java:
  https://en.wikipedia.org/wiki/Java_(programming_language)
- Python:
  https://en.wikipedia.org/wiki/Python_(programming_language)
- Dokumentacja C++:
  https://en.cppreference.com/
- Dokumentacja Java:
  https://docs.oracle.com/en/java/javase/14/docs/api/index.html
- Dokumentacja C#:
  https://docs.microsoft.com/pl-pl/dotnet/csharp/
- Dokumentacja Python:
  https://docs.python.org/3/
- Algorytmy sortowania:
  Introduction to algorithms, CLRS
  https://www.geeksforgeeks.org/sorting-algorithms/