Imię i Nazwisko	Kierunek	Rok studiów i grupa
Patryk Kozłowski	Informatyka Techniczna	1 rok grupa 2
Data zajęć	Numer i temat sprawozdania	
18.11.2024	8. Złożoność obliczeniowa	

Wprowadzenie teoretyczne

Złożoność obliczeniowa algorytmów jest kluczowym pojęciem w informatyce, pozwalającym określić efektywność algorytmu pod względem czasu wykonania oraz zużycia pamięci. Analiza złożoności umożliwia porównanie różnych algorytmów pod kątem ich wydajności i doboru odpowiedniego rozwiązania do danego problemu. Złożoność obliczeniowa dzieli się na dwa główne typy:

- **Złożoność czasową**: określa liczbę operacji wykonywanych przez algorytm w zależności od rozmiaru danych wejściowych. Wyrażana jest za pomocą notacji „O”, np. O(n)O(n), O(n2)O(n2), gdzie nn to liczba elementów.
- **Złożoność pamięciową**: dotyczy ilości pamięci potrzebnej do przechowywania danych podczas działania algorytmu. Może przybierać wartości stałe O(1)O(1) lub zależne od rozmiaru danych O(n)O(n).

Do wyznaczania złożoności rekurencyjnych algorytmów stosuje się metodę drzewa rekurencji, która polega na dekompozycji problemu na mniejsze podproblemy i analizie pracy wykonywanej na każdym poziomie. Przykładem może być równanie T(n)=T(25n)+T(35n)+cnT(n)=T(52n)+T(53n)+cn, które rozwiązuje się, analizując kolejne poziomy drzewa aż do osiągnięcia problemu bazowego.

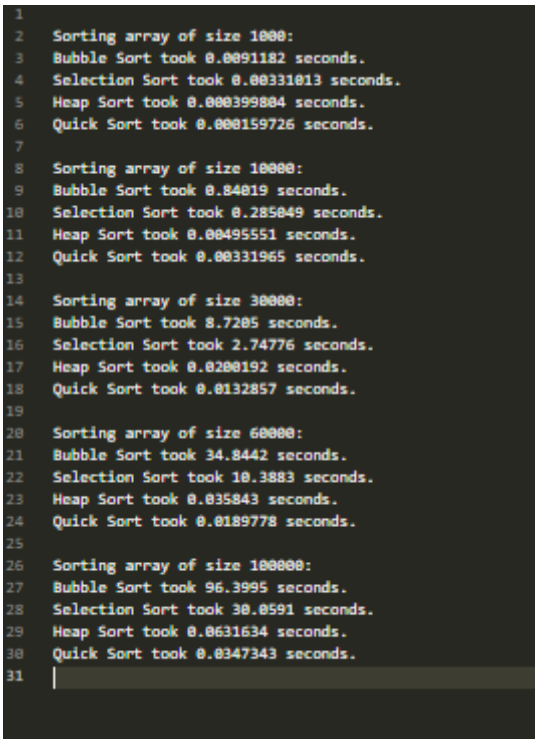
Zadanie 1

KOD Z ALGORYTMAMI W C++:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>
#include <chrono>
#include <fstream>
#include <thread>

void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void selectionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        std::swap(arr[i], arr[minIndex]);
    }
}

void heapify(std::vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        std::swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                std::swap(arr[i], arr[j]);
            }
        }
        std::swap(arr[i + 1], arr[high]);
        int pi = i + 1;

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```cpp
void measureSortingTime(void (*sortFunction)(std::vector<int>&), std::vector<int> arr, const std::string& sortName, std::fstream& file) {
    auto start = std::chrono::high_resolution_clock::now();
    sortFunction(arr);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;

    file << sortName << " took " << elapsed.count() << " seconds.\n";
}

void printProgressBar(int progress, int total) {
    int barWidth = 70;
    float progressRatio = static_cast<float>(progress) / total;
    int pos = barWidth * progressRatio;

    std::cout << "[";
    for (int i = 0; i < barWidth; ++i) {
        if (i < pos) std::cout << "=";
        else if (i == pos) std::cout << ">";
        else std::cout << " ";
    }
    std::cout << "] " << int(progressRatio * 100.0) << " %\r";
    std::cout.flush();
}

int main() {
    srand(time(NULL));

    std::fstream file;
    file.open("results.txt", std::ios::out);
    std::vector<int> sizes = {1000, 10000, 30000, 60000, 100000};

    int totalTasks = sizes.size() * 4; // 4 sorting algorithms
    int currentTask = 0;

    for (int size : sizes) {
        std::vector<int> arr(size);
        for (int i = 0; i < size; i++) {
            arr[i] = rand() % 100000 + 1;
        }

        file << "\nSorting array of size " << size << ":\n";

        measureSortingTime(bubbleSort, arr, "Bubble Sort", file);
        currentTask++;
        printProgressBar(currentTask, totalTasks);

        measureSortingTime(selectionSort, arr, "Selection Sort", file);
        currentTask++;
        printProgressBar(currentTask, totalTasks);

        auto heapArr = arr;
        auto start = std::chrono::high_resolution_clock::now();
        heapSort(heapArr);
        auto end = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = end - start;
        file << "Heap Sort took " << elapsed.count() << " seconds.\n";
        currentTask++;
        printProgressBar(currentTask, totalTasks);

        auto quickArr = arr;
        start = std::chrono::high_resolution_clock::now();
        quickSort(quickArr, 0, quickArr.size() - 1);
        end = std::chrono::high_resolution_clock::now();
        elapsed = end - start;
        file << "Quick Sort took " << elapsed.count() << " seconds.\n";
        currentTask++;
        printProgressBar(currentTask, totalTasks);
    }

    std::cout << std::endl;
    return 0;
}
```

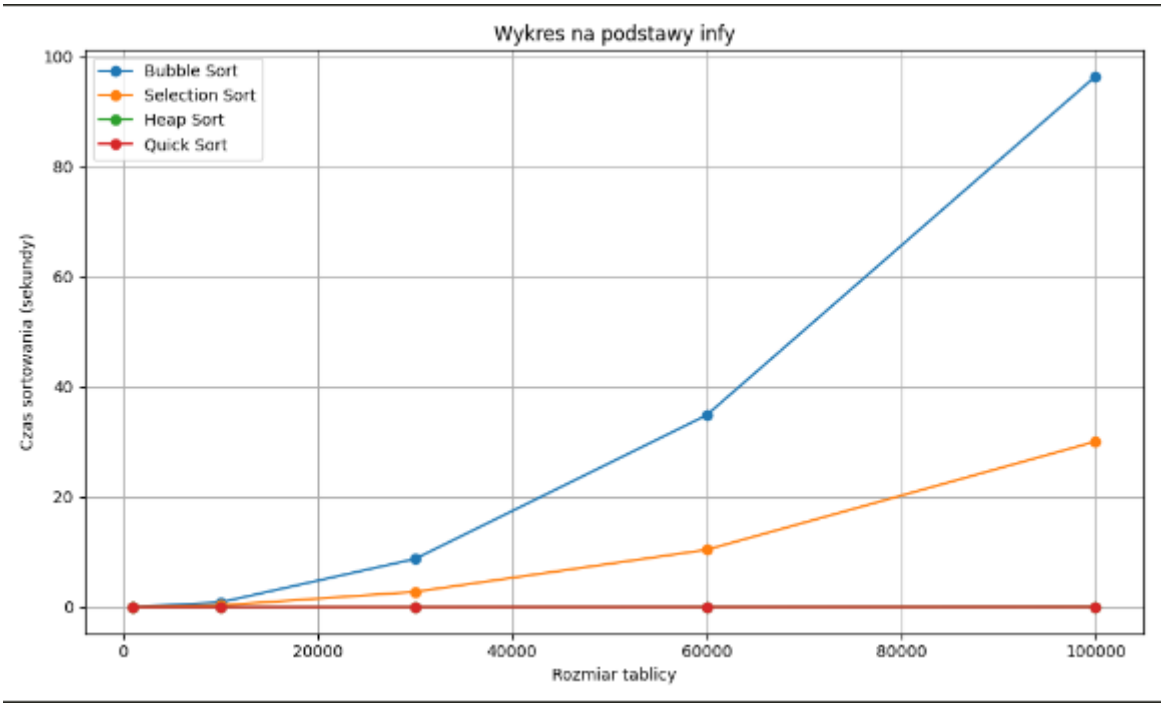
WYNIKI ZAPISANE W PLIKU TXT:



```
Sorting array of size 1000:
Bubble Sort took 0.0091182 seconds.
Selection Sort took 0.00331013 seconds.
Heap Sort took 0.000399804 seconds.
Quick Sort took 0.000159726 seconds.

Sorting array of size 10000:
Bubble Sort took 0.84019 seconds.
Selection Sort took 0.285049 seconds.
Heap Sort took 0.00495551 seconds.
Quick Sort took 0.00331965 seconds.

Sorting array of size 30000:
Bubble Sort took 8.7205 seconds.
Selection Sort took 2.74776 seconds.
Heap Sort took 0.0200192 seconds.
Quick Sort took 0.0132857 seconds.

Sorting array of size 60000:
Bubble Sort took 34.8442 seconds.
Selection Sort took 10.3883 seconds.
Heap Sort took 0.035843 seconds.
Quick Sort took 0.0189778 seconds.

Sorting array of size 100000:
Bubble Sort took 96.3995 seconds.
Selection Sort took 30.0591 seconds.
Heap Sort took 0.0631634 seconds.
Quick Sort took 0.0347343 seconds.
```

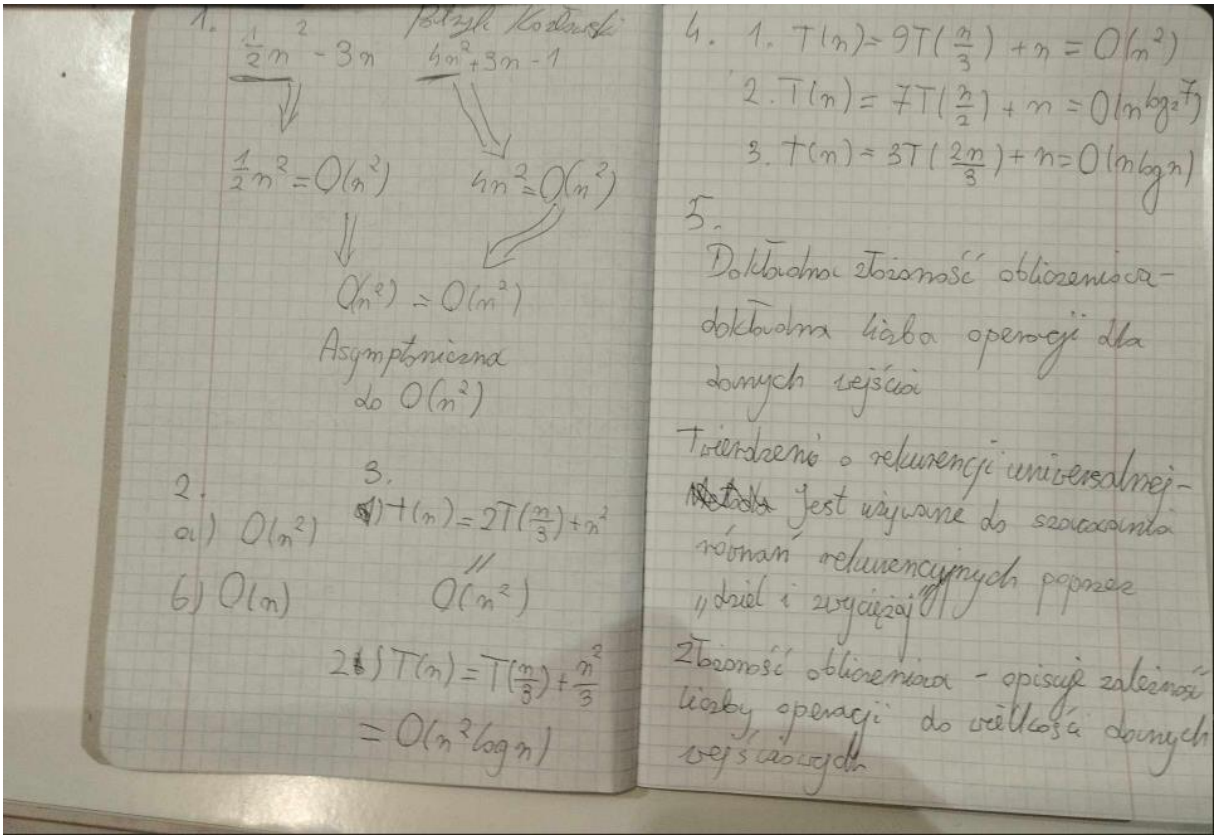
MATPLOTLIB W PYTHONIE BO WRAPPER W C++ NIE DZIAŁA:

```python
import matplotlib
matplotlib.use("Agg")

import matplotlib.pyplot as plt
import re

file_path = "results.txt"
with open(file_path, "r") as file:
    data = file.readlines()

sizes = []
bubble_sort_times = []
selection_sort_times = []
heap_sort_times = []
quick_sort_times = []

for line in data:
    if "Sorting array of size" in line:
        size = int(re.search(r'\d+', line).group())
        sizes.append(size)
    elif "Bubble Sort" in line:
        bubble_sort_times.append(float(re.search(r'\d+\.\d+', line).group()))
    elif "Selection Sort" in line:
        selection_sort_times.append(float(re.search(r'\d+\.\d+', line).group()))
    elif "Heap Sort" in line:
        heap_sort_times.append(float(re.search(r'\d+\.\d+', line).group()))
    elif "Quick Sort" in line:
        quick_sort_times.append(float(re.search(r'\d+\.\d+', line).group()))
plt.figure(figsize=(10, 6))
plt.plot(sizes, bubble_sort_times, label="Bubble Sort", marker='o')
plt.plot(sizes, selection_sort_times, label="Selection Sort", marker='o')
plt.plot(sizes, heap_sort_times, label="Heap Sort", marker='o')
plt.plot(sizes, quick_sort_times, label="Quick Sort", marker='o')

plt.title("Czas sortowania dla różnych algorytmów")
plt.xlabel("Rozmiar tablicy")
plt.ylabel("Czas sortowania (sekundy)")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.title("Wykres na podstawy infy")

plt.savefig("wykres.png")
```



Wykres do zadania 1

Zadanie 2



Wnioski

1. Złożoność obliczeniowa pozwala na ocenę efektywności algorytmów pod kątem czasu i pamięci. Znajomość tych parametrów umożliwia optymalny wybór metody rozwiązania problemu.
2. Rekurencyjne algorytmy, takie jak przedstawiony Algorytm A, wymagają szczegółowej analizy za pomocą równań rekurencyjnych, które określają liczbę operacji na kolejnych poziomach drzewa rekurencji.
3. Złożoność czasowa algorytmów sortowania, np. bąbelkowego O(n2)O(n2) i quicksort O(nlogn)O(nlogn), pokazuje różnice w wydajności, co ma kluczowe znaczenie przy pracy z dużymi zbiorami danych.
4. Asymptotyczna analiza funkcji pozwala na dokładne oszacowanie złożoności, co jest istotne w przypadku optymalizacji algorytmów i podejmowania decyzji projektowych dotyczących struktury kodu.
5. Dążenie do minimalizacji złożoności obliczeniowej jest ważnym aspektem tworzenia wydajnego oprogramowania, zwłaszcza w systemach, gdzie zasoby obliczeniowe są ograniczone.