Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
ФІОТ
ІСТ

Лабораторна робота №11

по курсу *“*Абстрактні типи даних”

Варіант 3

Виконали ст. групи ІС-31:
Коваль Богдан
Михайлова Софія
Шмигельський Ілля
Перевірив: ст. вик. Дорошенко К.С.

Київ 2024

Комп'ютерний практикум

Тема: Алгоритми сортування. Методи сортування великих обсягів даних

Мета роботи: Дослідження та порівняння алгоритмів сортування.

Завдання

Варіант 3

Щоб запустити в роботу нових роботів необхідно розуміти, який впорядкований

перелік пристроїв існує в невеликій частині даної мережі та в усій мережі (кількість пристроїв

в мережі велика). Але роботи можуть ще пересуватися між мережами, тому склали ще один

відповідний список всіх мереж. Такий список виявився великий і мав багато пристроїв з

однаковою потужністю. Допоможіть скласти фахівцям відповідні списки.

Перелік алгоритмів:

1) сортування злиттям;

2) пірамідальне сортування;

3) плавне сортування;

У варіантах вказані номери алгоритмів із вищевказаного переліку.

Варіанти

1. 1), 2).

2. 1), 3).

3. 2), 3). + +швидке сортування.

Склад звіту практичної роботи

– постановка задачі;

– опис методів сортування: сортування злиттям, пірамідальне сортування, плавне сортування;

– програмна реалізація вирішення задачі на основі свого варіанту методів сортування;

– 3-5 експериментів для оцінки продуктивності реалізацій для обраних алгоритмів, а також для алгоритму швидкого сортування, доречно скористатися створеною програмною реалізацією з попереднього практикуму.

– висновки за результатами експериментів;

висновки про отримання практичного підтвердження або спростування про продуктивність сортування, що було визначено теоретично; для яких даних досліджувані алгоритми підходить якнайкраще, а у яких випадках використовувати його недоцільно;

– відповіді на контрольні питання.

Варіанти завдань.

Варіант 3

Щоб запустити в роботу нових роботів необхідно розуміти, який впорядкований перелік пристроїв існує в невеликій частині даної мережі та в усій мережі (кількість пристроїв в мережі велика). Але роботи можуть ще пересуватися між мережами, тому склали ще один відповідний список всіх мереж. Такий список виявився великий і мав багато пристроїв з однаковою потужністю. Допоможіть скласти фахівцям відповідні списки.

Перелік алгоритмів:

1) сортування злиттям;

2) пірамідальне сортування;

3) плавне сортування;

У варіантах вказані номери алгоритмів із вищевказаного переліку.

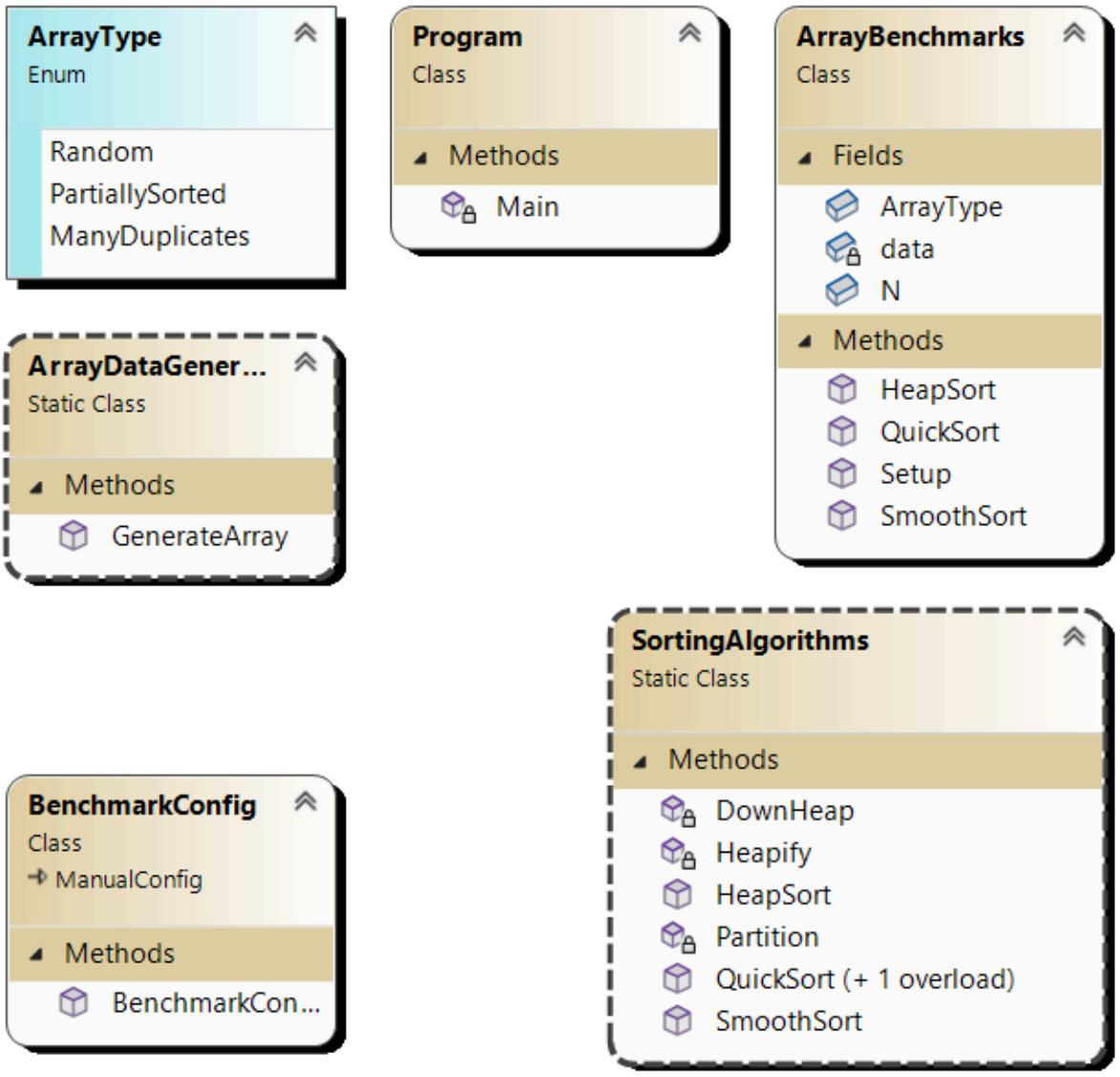
Варіанти

1. 1), 2).

2. 1), 3).

3. 2), 3). + +швидке сортування.

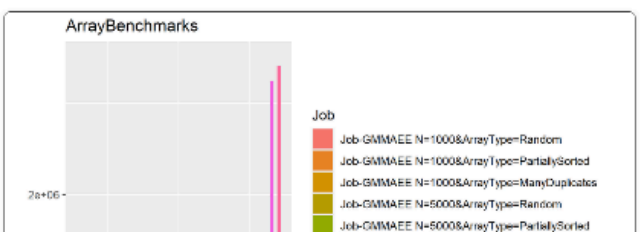
Побудова моделі



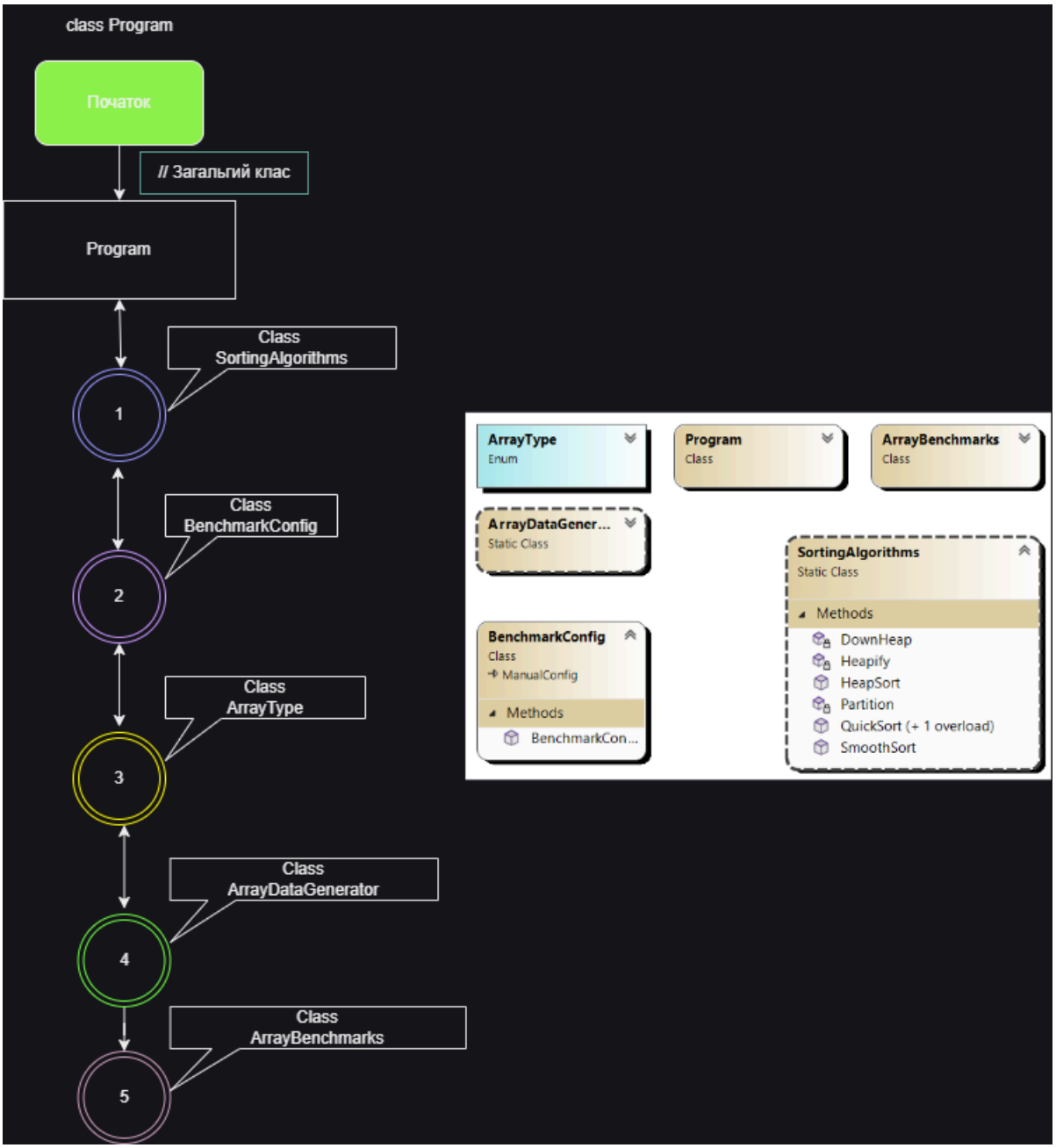
Інтерфейс користувача

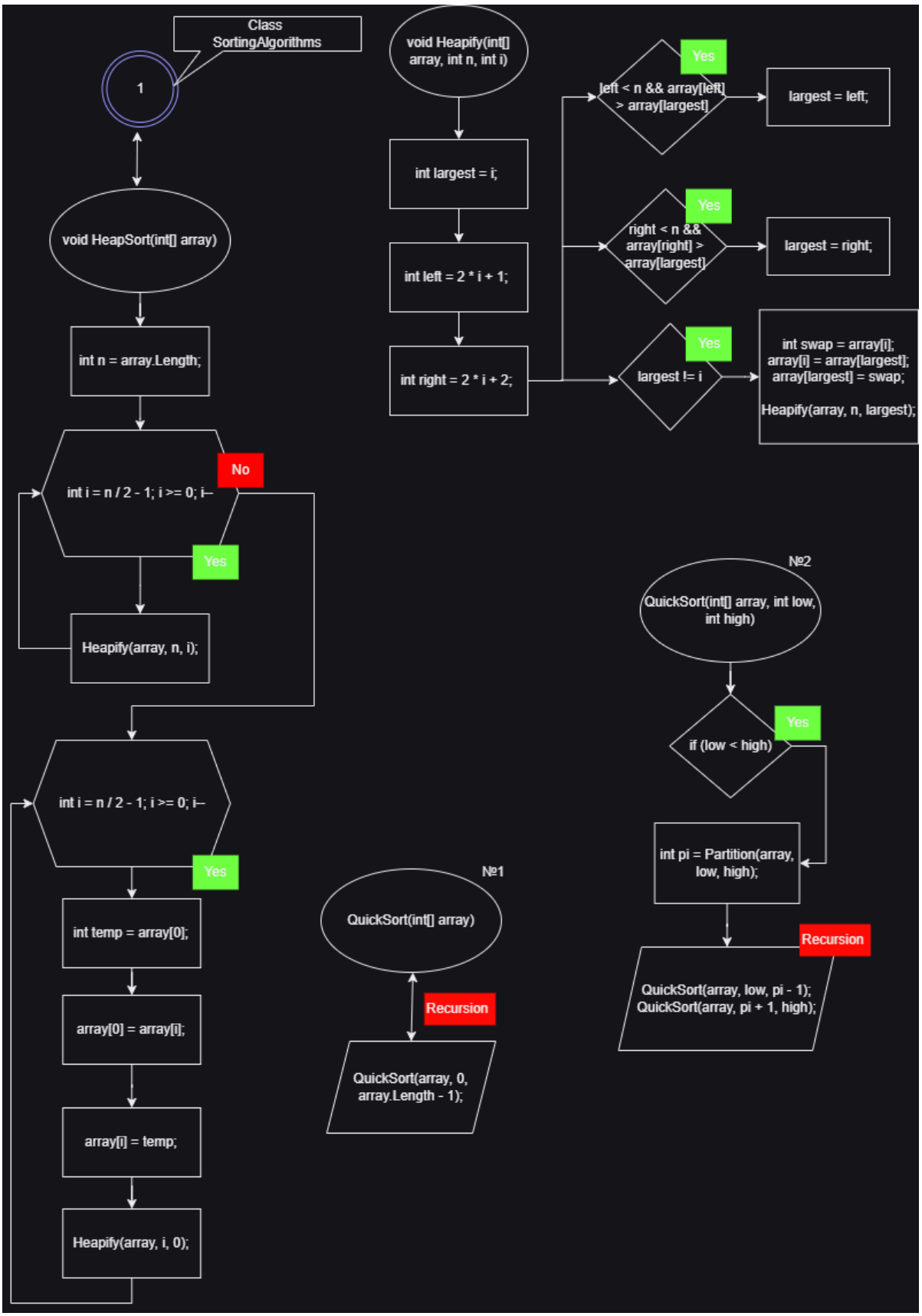
```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.4412/22H2/2022Update)
12th Gen Intel Core i5-12450H, 1 CPU, 12 logical and 8 physical cores
.NET SDK 9.0.100-preview.7.24407.12
  [Host]     : .NET 5.0.17 (5.0.1722.21314), X64 RyuJIT AVX2
  Job-GMMAEE : .NET 5.0.17 (5.0.1722.21314), X64 RyuJIT AVX2

IterationCount=10  RunStrategy=Throughput  WarmupCount=3
```

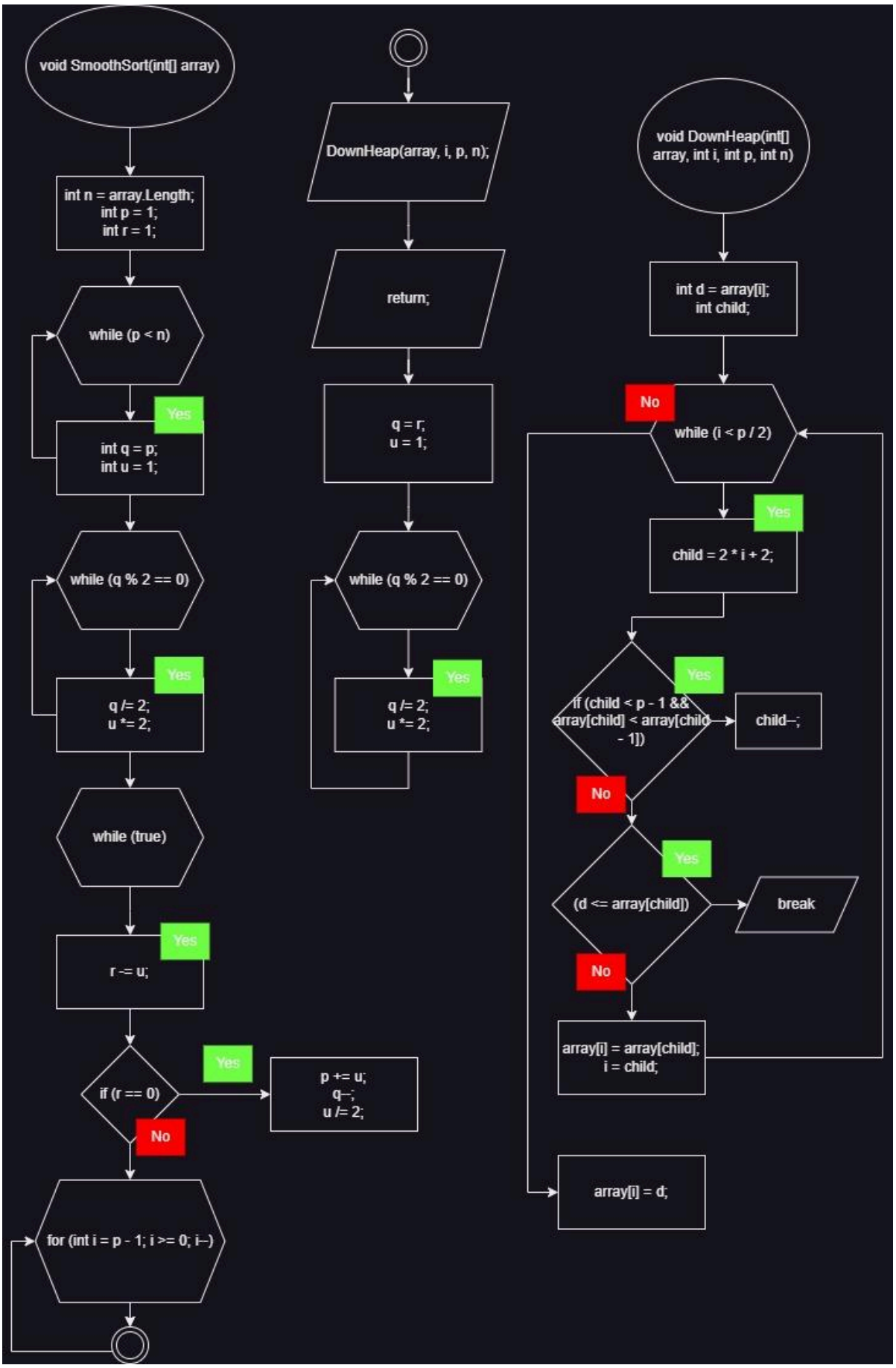
Method	N	ArrayType	Mean	Error	StdDev	Gen0	Gen1	Gen2	Allocated
HeapSort	1000	Random	316.3 ns	19.18 ns	12.68 ns	0.0061	-	-	40 B
SmoothSort	1000	Random	784.0 ns	62.07 ns	41.06 ns	0.0061	-	-	40 B
QuickSort	1000	Random	242.0 ns	10.83 ns	6.44 ns	0.0125	-	-	80 B
HeapSort	1000	PartiallySorted	316.0 ns	30.94 ns	16.18 ns	0.0061	-	-	40 B
SmoothSort	1000	PartiallySorted	802.2 ns	282.98 ns	187.17 ns	0.0061	-	-	40 B
QuickSort	1000	PartiallySorted	552.8 ns	265.39 ns	175.54 ns	0.0125	-	-	80 B
HeapSort	1000	ManyDuplicates	313.8 ns	27.25 ns	14.25 ns	0.0061	-	-	40 B
SmoothSort	1000	ManyDuplicates	984.0 ns	326.43 ns	215.91 ns	0.0061	-	-	40 B
QuickSort	1000	ManyDuplicates	433.8 ns	188.34 ns	98.51 ns	0.0125	-	-	80 B
HeapSort	5000	Random	4,017.5 ns	1,084.17 ns	717.11 ns	0.0293	-	-	200 B
SmoothSort	5000	Random	6,799.8 ns	1,657.70 ns	1,096.47 ns	0.0293	-	-	200 B
QuickSort	5000	Random	5,136.8 ns	1,086.12 ns	718.40 ns	0.0635	-	-	400 B
HeapSort	5000	PartiallySorted	3,916.1 ns	1,282.49 ns	848.29 ns	0.0293	-	-	200 B
SmoothSort	5000	PartiallySorted	6,453.4 ns	2,082.92 ns	1,377.72 ns	0.0293	-	-	200 B
QuickSort	5000	PartiallySorted	4,689.6 ns	1,093.03 ns	722.97 ns	0.0635	-	-	400 B
HeapSort	5000	ManyDuplicates	4,360.2 ns	1,067.67 ns	706.20 ns	0.0293	-	-	200 B
SmoothSort	5000	ManyDuplicates	8,544.9 ns	1,510.13 ns	998.86 ns	0.0293	-	-	200 B
QuickSort	5000	ManyDuplicates	5,428.0 ns	1,685.35 ns	1,114.75 ns	0.0635	-	-	400 B
HeapSort	25000	Random	25,952.7 ns	5,791.75 ns	3,830.88 ns	0.2734	0.2734	0.2734	1000 B
SmoothSort	25000	Random	51,618.4 ns	9,775.90 ns	6,466.15 ns	0.2344	0.2344	0.2344	1000 B
QuickSort	25000	Random	37,573.3 ns	4,612.56 ns	3,050.93 ns	0.5859	0.5859	0.5859	2000 B
HeapSort	25000	PartiallySorted	20,237.4 ns	1,640.11 ns	857.81 ns	0.2734	0.2734	0.2734	1000 B
SmoothSort	25000	PartiallySorted	44,619.2 ns	8,496.47 ns	5,619.89 ns	0.2734	0.2734	0.2734	1000 B
QuickSort	25000	PartiallySorted	25,807.6 ns	6,675.49 ns	4,415.42 ns	0.5859	0.5859	0.5859	2000 B
HeapSort	25000	ManyDuplicates	20,214.7 ns	1,363.60 ns	713.19 ns	0.2734	0.2734	0.2734	1000 B
SmoothSort	25000	ManyDuplicates	49,449.6 ns	7,764.74 ns	4,620.67 ns	0.2734	0.2734	0.2734	1000 B
QuickSort	25000	ManyDuplicates	29,429.5 ns	7,392.39 ns	4,889.61 ns	0.5859	0.5859	0.5859	2000 B
HeapSort	50000	Random	49,740.6 ns	12,885.82 ns	8,523.17 ns	0.5469	0.5469	0.5469	2000 B
SmoothSort	50000	Random	111,601.8 ns	11,385.38 ns	6,775.26 ns	0.5469	0.5469	0.5469	2000 B
QuickSort	50000	Random	70,073.6 ns	16,797.58 ns	11,110.56 ns	1.1719	1.1719	1.1719	4000 B
HeapSort	50000	PartiallySorted	37,434.1 ns	1,580.00 ns	826.37 ns	0.5469	0.5469	0.5469	2000 B
SmoothSort	50000	PartiallySorted	96,577.6 ns	17,931.55 ns	11,860.61 ns	0.5469	0.5469	0.5469	2000 B
QuickSort	50000	PartiallySorted	57,454.0 ns	15,636.37 ns	10,342.49 ns	1.1719	1.1719	1.1719	4000 B
HeapSort	50000	ManyDuplicates	49,725.6 ns	11,610.96 ns	7,679.93 ns	0.5469	0.5469	0.5469	2000 B
SmoothSort	50000	ManyDuplicates	111,400.4 ns	11,922.00 ns	7,885.67 ns	0.5469	0.5469	0.5469	2000 B
QuickSort	50000	ManyDuplicates	62,795.5 ns	14,468.20 ns	9,569.82 ns	1.1719	1.1719	1.1719	4000 B
HeapSort	100000	Random	108,808.8 ns	32,015.58 ns	21,176.32 ns	1.0938	1.0938	1.0938	4000 B
SmoothSort	100000	Random	159,164.0 ns	1,251.18 ns	654.39 ns	0.9375	0.9375	0.9375	4000 B
QuickSort	100000	Random	150,980.3 ns	34,329.61 ns	22,706.91 ns	2.3438	2.3438	2.3438	8001 B
HeapSort	100000	PartiallySorted	82,984.4 ns	5,836.77 ns	3,052.74 ns	1.0938	1.0938	1.0938	4000 B
SmoothSort	100000	PartiallySorted	209,238.5 ns	31,079.01 ns	20,556.84 ns	1.0938	1.0938	1.0938	4000 B
QuickSort	100000	PartiallySorted	127,004.5 ns	31,980.01 ns	21,152.80 ns	2.3438	2.3438	2.3438	8001 B
HeapSort	100000	ManyDuplicates	110,448.5 ns	29,467.47 ns	19,490.90 ns	1.0938	1.0938	1.0938	4000 B
SmoothSort	100000	ManyDuplicates	165,196.6 ns	8,655.17 ns	4,526.82 ns	0.9375	0.9375	0.9375	4000 B
QuickSort	100000	ManyDuplicates	132,320.2 ns	28,556.77 ns	18,888.54 ns	2.3438	2.3438	2.3438	8001 B
HeapSort	1000000	Random	1,211,481.2 ns	55,167.44 ns	28,853.64 ns	-	-	-	40001 B
SmoothSort	1000000	Random	2,622,555.0 ns	887,653.07 ns	587,127.54 ns	-	-	-	40001 B
QuickSort	1000000	Random	1,607,046.9 ns	345,864.43 ns	228,767.91 ns	5.7143	5.7143	5.7143	80016 B
HeapSort	1000000	PartiallySorted	1,116,392.9 ns	321,499.95 ns	212,652.31 ns	2.0000	2.0000	2.0000	40001 B
SmoothSort	1000000	PartiallySorted	1,726,837.3 ns	46,593.23 ns	24,369.16 ns	-	-	-	40001 B
QuickSort	1000000	PartiallySorted	1,410,681.2 ns	382,604.91 ns	253,069.46 ns	5.0000	5.0000	5.0000	80014 B
HeapSort	1000000	ManyDuplicates	1,488,347.2 ns	434,603.51 ns	287,463.31 ns	2.0000	2.0000	2.0000	40011 B
SmoothSort	1000000	ManyDuplicates	2,702,761.8 ns	1,015,796.29 ns	671,886.35 ns	-	-	-	40001 B
QuickSort	1000000	ManyDuplicates	1,598,111.9 ns	483,337.87 ns	319,698.07 ns	5.7143	5.7143	5.7143	80001 B



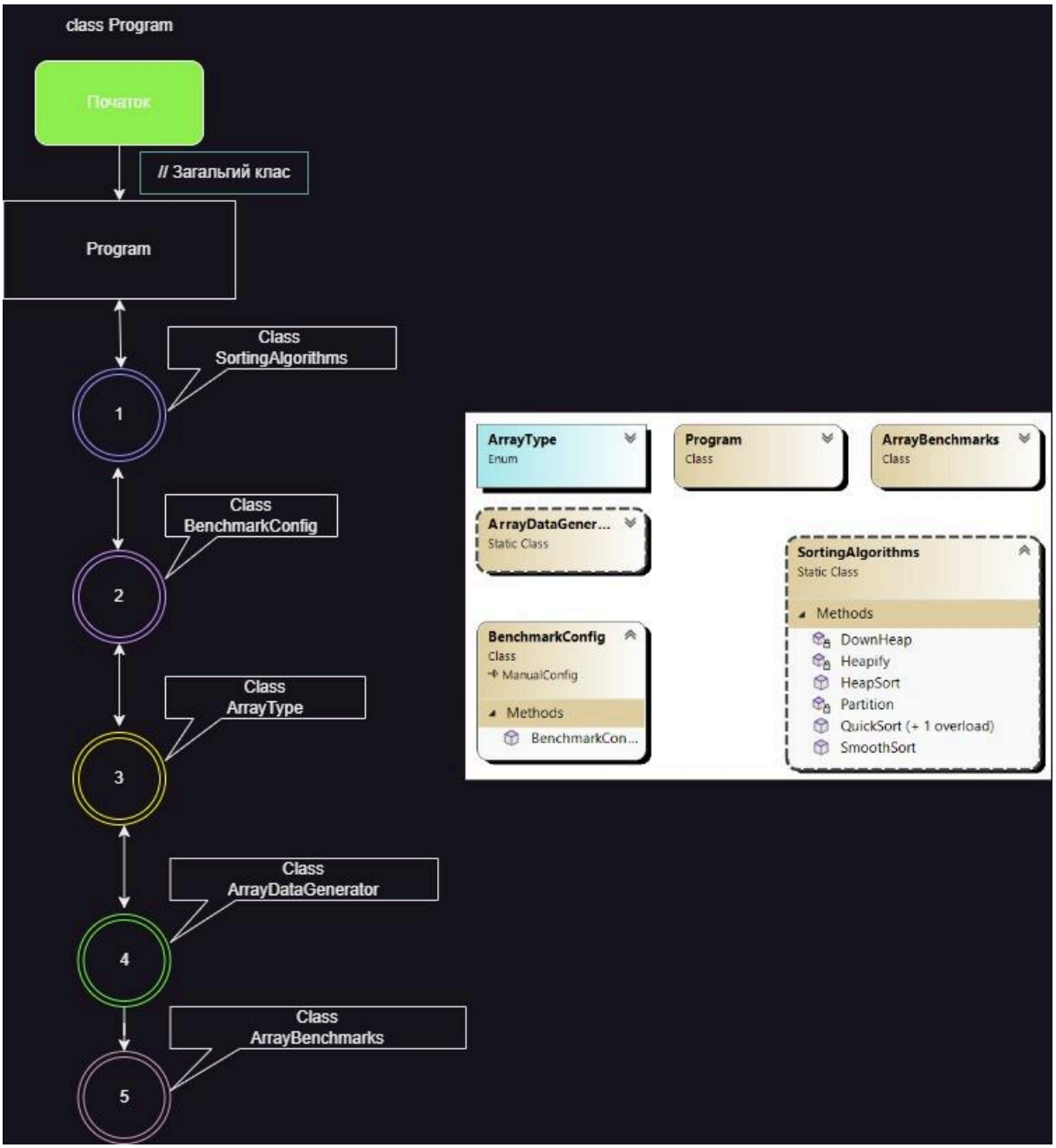
Розробка алгоритму



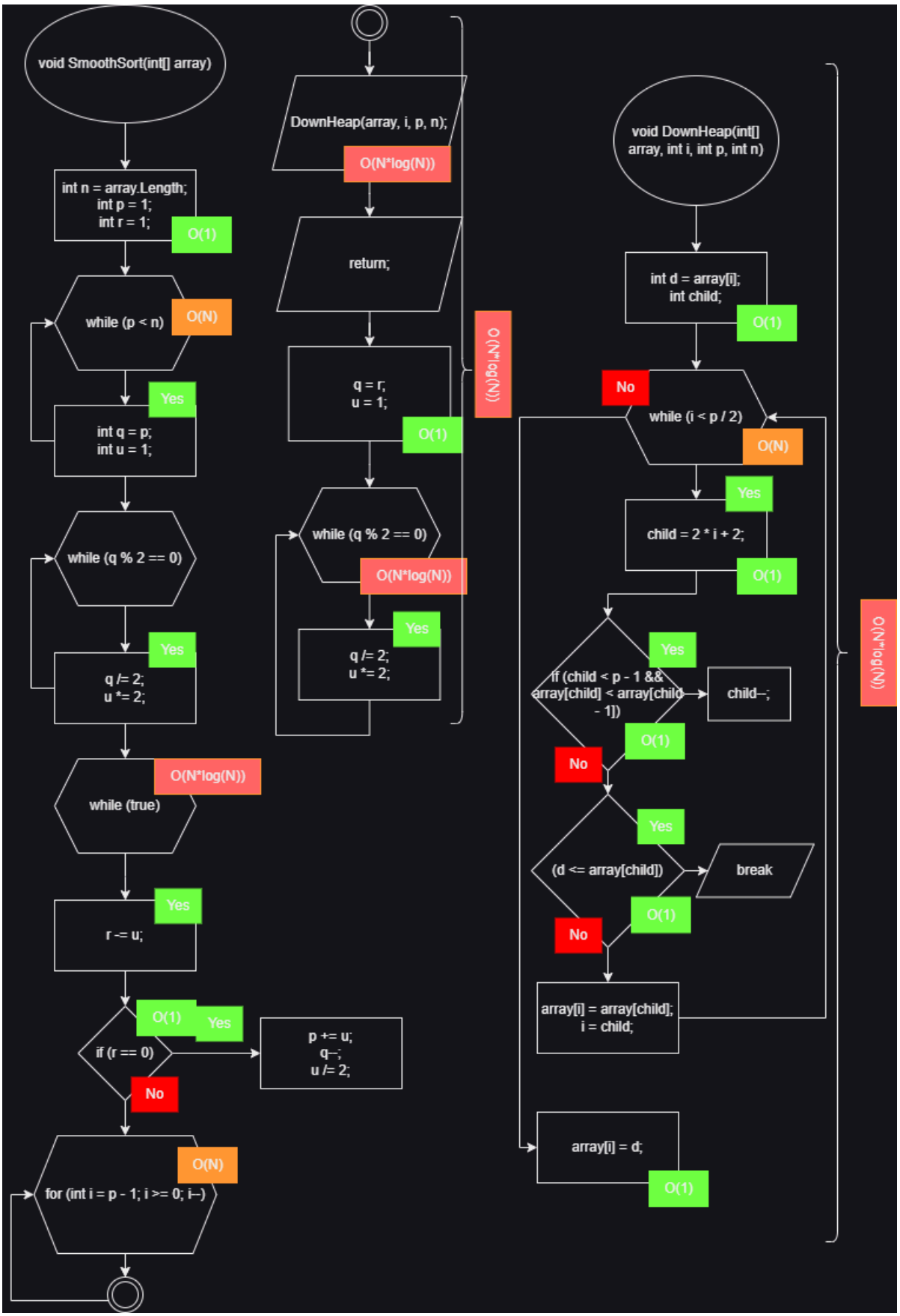

Class SortingAlgorithms
1
void HeapSort(int[] array)
int n = array.Length;
int i = n / 2 - 1; i >= 0; i--
No
Yes
Heapify(array, n, i);
int i = n / 2 - 1; i >= 0; i--
Yes
int temp = array[0];
array[0] = array[i];
array[i] = temp;
Heapify(array, i, 0);
void Heapify(int[] array, int n, int i)
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;
left < n && array[left] > array[largest]
Yes
largest = left;
right < n && array[right] > array[largest]
Yes
largest = right;
largest != i
Yes
int swap = array[i];
array[i] = array[largest];
array[largest] = swap;
Heapify(array, n, largest);
№1
QuickSort(int[] array)
Recursion
QuickSort(array, 0, array.Length - 1);
№2
QuickSort(int[] array, int low, int high)
if (low < high)
Yes
int pi = Partition(array, low, high);
Recursion
QuickSort(array, low, pi - 1);
QuickSort(array, pi + 1, high);


void SmoothSort(int[] array)
int n = array.Length;
int p = 1;
int r = 1;
while (p < n)
Yes
int q = p;
int u = 1;
while (q % 2 == 0)
Yes
q /= 2;
u *= 2;
while (true)
Yes
r -= u;
if (r == 0)
Yes
p += u;
q--;
u /= 2;
No
for (int i = p - 1; i >= 0; i--)
DownHeap(array, i, p, n);
return;
q = r;
u = 1;
while (q % 2 == 0)
Yes
q /= 2;
u *= 2;
void DownHeap(int[] array, int i, int p, int n)
int d = array[i];
int child;
No
while (i < p / 2)
Yes
child = 2 * i + 2;
if (child < p - 1 && array[child] < array[child - 1])
Yes
child--;
No
(d <= array[child])
Yes
break
No
array[i] = array[child];
i = child;
array[i] = d;

Аналіз алгоритму та його складності



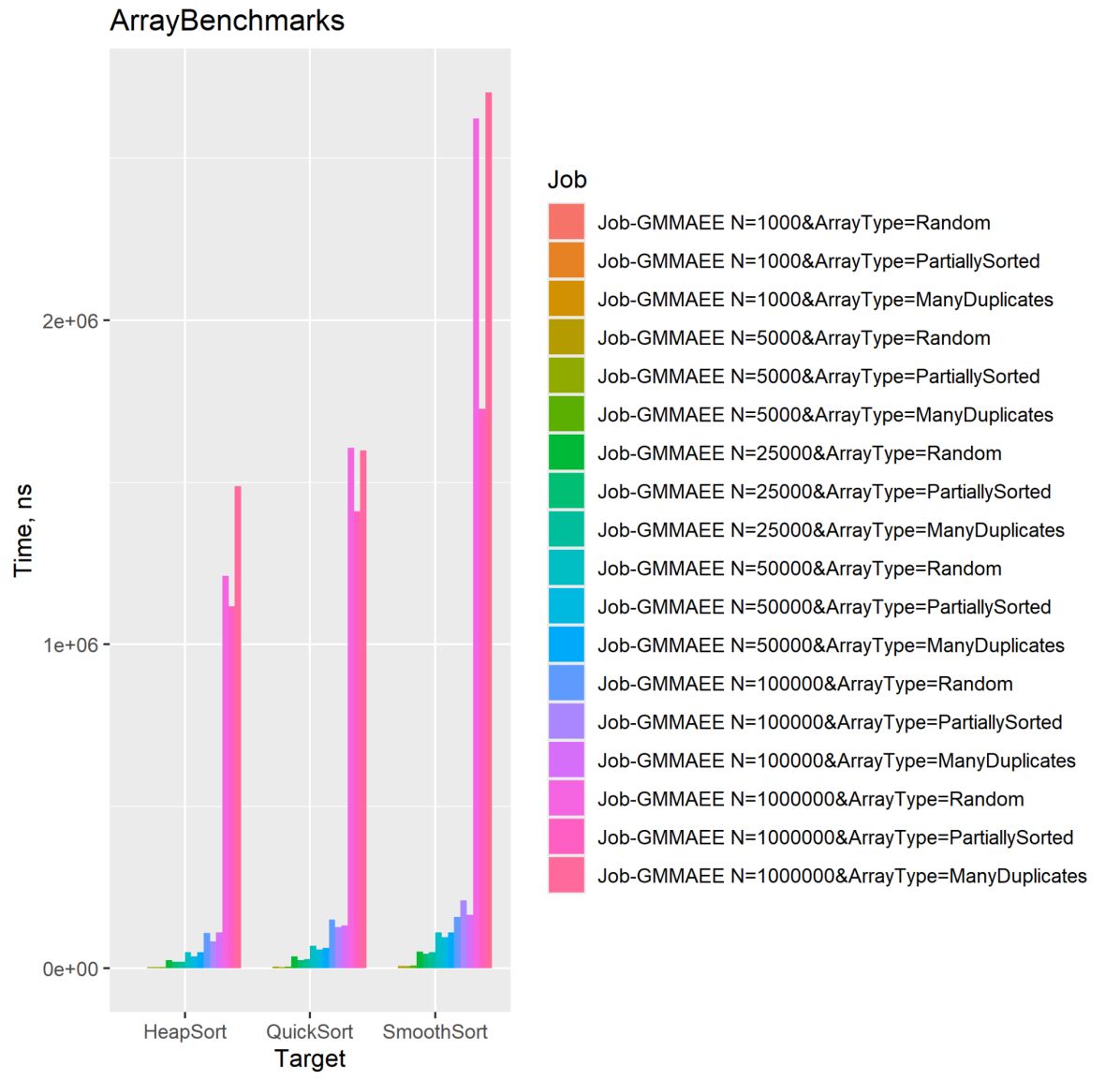

Class SortingAlgorithms
1
void HeapSort(int[] array)
int n = array.Length;
O(1)
int i = n / 2 - 1; i >= 0; i--
No
O(N)
Yes
Heapify(array, n, i);
int i = n / 2 - 1; i >= 0; i--
O(N)
Yes
int temp = array[0];
O(1)
array[0] = array[i];
O(1)
array[i] = temp;
O(1)
Heapify(array, i, 0);
O(1)
O(N*log(N))
void Heapify(int[] array, int n, int i)
int largest = i;
O(1)
int left = 2 * i + 1;
O(1)
int right = 2 * i + 2;
O(1)
left < n && array[left] > array[largest]
Yes
O(1)
largest = left;
right < n && array[right] > array[largest]
Yes
O(1)
largest = right;
largest != i
Yes
O(1)
int swap = array[i]; array[i] = array[largest]; array[largest] = swap;
Heapify(array, n, largest)
O(N*log(N))
№1
QuickSort(int[] array)
Recursion
O(N*log(N))
QuickSort(array, 0, array.Length - 1);
O(N*log(N))
№2
QuickSort(int[] array, int low, int high)
if (low < high)
Yes
O(1)
int pi = Partition(array, low, high);
Recursion
QuickSort(array, low, pi - 1);
QuickSort(array, pi + 1, high);
O(N*log(N))
O(N*log(N))



Загальний графік оцінки складності

[Params(1_000, 5_000, 25_000, 50_000, 100_000, 1_000_000)]

[Params(ArrayType.Random, ArrayType.PartiallySorted, ArrayType.ManyDuplicates)]



BenchmarkDotNet v0.13.12

Порівняльний висновок :

В ході лабораторної роботи ми дослідили методи сортування великих обсягів даних для трьох типів вхідних даних: хаотичних, частково впорядкованих та з багатьма повторами. Аналізувалися такі алгоритми як пірамідальне сортування, плавне сортування та швидке сортування. На основі проведених досліджень можна зробити висновок про

ефективність кожного з алгоритмів у різних умовах вхідних даних. Результати дослідження допоможуть вибрати найбільш оптимальний метод сортування для конкретної задачі та масштабу даних.

Реалізація алгоритму:

```
using System;

public static class SortingAlgorithms
{
    public static void HeapSort(int[] array)
    {
    int n = array.Length;

    for (int i = n / 2 - 1; i >= 0; i--)
        Heapify(array, n, i);

    for (int i = n - 1; i > 0; i--)
    {
        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;

        Heapify(array, i, 0);
    }
    }

    private static void Heapify(int[] array, int n, int i)
    {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && array[left] > array[largest])
```

```
        largest = left;

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

    if (largest != i)
    {
        int swap = array[i];
        array[i] = array[largest];
        array[largest] = swap;

        Heapify(array, n, largest);
    }
    }

    public static void QuickSort(int[] array)
    {
    QuickSort(array, 0, array.Length - 1);
    }

    private static void QuickSort(int[] array, int low, int high)
    {
    if (low < high)
    {
        int pivot = Partition(array, low, high);
        QuickSort(array, low, pivot - 1);
        QuickSort(array, pivot + 1, high);
    }
    }

    private static int Partition(int[] array, int low, int high)
    {
    int pivot = array[high];
    int i = low - 1;
```

```
    for (int j = low; j < high; j++)
    {
        if (array[j] < pivot)
        {
            i++;
            Swap(ref array[i], ref array[j]);
        }
    }
    Swap(ref array[i + 1], ref array[high]);
    return i + 1;
    }

    private static void Swap(ref int a, ref int b)
    {
    int temp = a;
    a = b;
    b = temp;
    }

    public static void SmoothSort(int[] array)
    {
    int q, r, p, b, c, r1, b1, c1;

    bool IsAscending(int A, int B)
    {
        return A < B;
    }

    void UP(ref int IA, ref int IB, ref int temp)
    {
        temp = IA;
        IA += IB + 1;
        IB = temp;
    }
```

```
void DOWN(ref int IA, ref int IB, ref int temp)
{
    temp = IB;
    IB = IA - IB - 1;
    IA = temp;
}

void Sift(ref int[] A)
{
    int r0, r2, temp = 0;
    int t;
    r0 = r1;
    t = A[r0];

    while (b1 >= 3)
    {
        r2 = r1 - b1 + c1;

        if (!IsAscending(A[r1 - 1], A[r2]))
        {
            r2 = r1 - 1;
            DOWN(ref b1, ref c1, ref temp);
        }

        if (IsAscending(A[r2], t))
        {
            b1 = 1;
        }
        else
        {
            A[r1] = A[r2];
            r1 = r2;
            DOWN(ref b1, ref c1, ref temp);
```

```
            }
        }

        if (Convert.ToBoolean(r1 - r0))
            A[r1] = t;
    }

    void Trinkle(ref int[] A)
    {
        int p1, r2, r3, r0, temp = 0;
        int t;
        p1 = p;
        b1 = b;
        c1 = c;
        r0 = r1;
        t = A[r0];

        while (p1 > 0)
        {
            while ((p1 & 1) == 0)
            {
                p1 >>= 1;
                UP(ref b1, ref c1, ref temp);
            }

            r3 = r1 - b1;

            if ((p1 == 1) || IsAscending(A[r3], t))
            {
                p1 = 0;
            }
            else
            {
                --p1;
```

```
            if (b1 == 1)
            {
                A[r1] = A[r3];
                r1 = r3;
            }
            else
            {
                if (b1 >= 3)
                {
                    r2 = r1 - b1 + c1;

                    if (!IsAscending(A[r1 - 1], A[r2]))
                    {
                        r2 = r1 - 1;
                        DOWN(ref b1, ref c1, ref temp);
                        p1 <<= 1;
                    }
                    if (IsAscending(A[r2], A[r3]))
                    {
                        A[r1] = A[r3]; r1 = r3;
                    }
                    else
                    {
                        A[r1] = A[r2];
                        r1 = r2;
                        DOWN(ref b1, ref c1, ref temp);
                        p1 = 0;
                    }
                }
            }
        }
    }
```

```
        if (Convert.ToBoolean(r0 - r1))
            A[r1] = t;

        Sift(ref A);
    }

    void SemiTrinkle(ref int[] A)
    {
        int T;
        r1 = r - c;

        if (!IsAscending(A[r1], A[r]))
        {
            T = A[r];
            A[r] = A[r1];
            A[r1] = T;
            Trinkle(ref A);
        }
    }

    int temp = 0;
    q = 1;
    r = 0;
    p = 1;
    b = 1;
    c = 1;

    while (q < array.Length)
    {
        r1 = r;
        if ((p & 7) == 3)
        {
            b1 = b;
            c1 = c;
```

```
            Sift(ref array);
            p = (p + 1) >> 2;
            UP(ref b, ref c, ref temp);
            UP(ref b, ref c, ref temp);
        }
        else if ((p & 3) == 1)
        {
            if (q + c < array.Length)
            {
                b1 = b;
                c1 = c;
                Sift(ref array);
            }
            else
            {
                Trinkle(ref array);
            }

            DOWN(ref b, ref c, ref temp);
            p <<= 1;

            while (b > 1)
            {
                DOWN(ref b, ref c, ref temp);
                p <<= 1;
            }

            ++p;
        }

        ++q;
        ++r;
    }
```

```
    r1 = r;
    Trinkle(ref array);

    while (q > 1)
    {
        --q;

        if (b == 1)
        {
            --r;
            --p;

            while ((p & 1) == 0)
            {
                p >>= 1;
                UP(ref b, ref c, ref temp);
            }
        }
        else
        {
            if (b >= 3)
            {
                --p;
                r = r - b + c;
                if (p > 0)
                    SemiTrinkle(ref array);

                DOWN(ref b, ref c, ref temp);
                p = (p << 1) + 1;
                r = r + c;
                SemiTrinkle(ref array);
                DOWN(ref b, ref c, ref temp);
                p = (p << 1) + 1;
            }
```

```
            }
        }
        }
}
```

Контрольні запитання

1. Переваги та недоліки швидкого сортування Швидке сортування (QuickSort) — це алгоритм сортування на основі розділення (partitioning), де масив ділиться на дві частини за допомогою "опорного" елемента (pivot), потім ці частини сортуються рекурсивно.

Переваги:

- **Швидкість**: У середньому, часова складність O(n log n), що робить QuickSort одним з найшвидших алгоритмів для сортування.
- **Низькі вимоги до пам'яті**: Потребує невеликої додаткової пам'яті, особливо при нерекурсивних реалізаціях.
- **Гнучкість**: Підходить для різних типів даних і може бути адаптований для використання з різними стратегіями вибору опорного елемента.
- **Простота реалізації**: Досить простий у реалізації, особливо при використанні технік, як-от метод Хоара.

Недоліки:

- **Нестійкість**: Швидке сортування може змінювати порядок рівних елементів.
- **Чутливість до вибору "опорного" елемента**: Невдалий вибір опорного елемента може призвести до значного зниження продуктивності (до $O(n^2)$).
- **Рекурсивна глибина**: Якщо не використовується оптимізація, надто глибока рекурсія може призвести до переповнення стеку.

2. Переваги та недоліки сортування злиттям. Найпоширеніші модифікації цього методу сортування Сортування злиттям (MergeSort) — це рекурсивний алгоритм, який розділяє масив на дві частини, сортує їх, а потім зливає у відсортованому порядку.

Переваги:

- **Стійкість**: Зберігає порядок рівних елементів.
- **Постійна продуктивність**: Часова складність O(n log n) навіть у найгіршому випадку.
- **Добре для великих наборів даних**: MergeSort ефективний для великих масивів і може бути адаптований для сортування на диску (наприклад, при обробці великих файлів).
- **Паралелізм**: MergeSort легко паралелізувати, оскільки операції розділення та злиття можуть виконуватися одночасно.

Недоліки:

- **Високі вимоги до пам'яті**: Потребує додаткової пам'яті для злиття, зазвичай O(n).
- **Більш складна реалізація**: Порівняно зі швидким сортуванням, реалізація може бути складнішою через операції злиття.

Найпоширеніші модифікації:

- **Сортування злиттям з рекурсивним розділенням**: Стандартна версія, що використовує рекурсивне розділення масиву.
- **Нерекурсивне сортування злиттям**: Виконується в циклі, поетапно зливаючи масиви різної довжини.
- **Тимчасовий масив злиття (Merge In-Place)**: Зменшує додаткові вимоги до пам'яті шляхом використання спеціальних методів злиття.

3. Переваги та недоліки пірамідального сортування Пірамідальне сортування (HeapSort) використовує структуру даних "піраміда" (або "куча"), яка є спеціальним типом двійкового дерева, де батьківський вузол завжди більший (або менший) за дітей.

Переваги:

- **Постійна продуктивність**: Часова складність O(n log n) у всіх випадках.

- **Низькі вимоги до додаткової пам'яті**: Потребує лише O(1) додаткової пам'яті.
- **Гнучкість**: Пірамідальне сортування використовується для різних типів даних і підходить для реалізації пріоритетних черг.

Недоліки:

- **Нестійкість**: Пірамідальне сортування може змінювати порядок рівних елементів.
- **Потенційно повільніші операції**: Оскільки кожне видалення піраміди включає відновлення її структури, це може призвести до додаткових операцій.

4. Переваги та недоліки плавного сортування Плавне сортування (SmoothSort) — це варіант пірамідального сортування, який забезпечує O(n log n) у середньому та O(n) у найкращому випадку.

Переваги:

- **Ефективність**: Може бути швидшим, ніж звичайне пірамідальне сортування, особливо при майже відсортованих масивах.
- **Низькі вимоги до пам'яті**: Також потребує мінімум додаткової пам'яті.
- **Постійна продуктивність**: Забезпечує часову складність O(n log n).

Недоліки:

- **Складність реалізації**: Реалізація складніша порівняно з іншими методами сортування.
- **Нестійкість**: Порядок рівних елементів не гарантовано зберігається.

5. Порівняльна характеристика швидкого сортування та сортування злиттям Швидке сортування (QuickSort):

- Часова складність: O(n log n) у середньому, але може бути $O(n^2)$ у найгіршому випадку.
- Просторова складність: Низька, але може бути рекурсивна глибина.
- Стійкість: Нестійкий.

- Найкраще працює: Для невеликих масивів, випадкових даних, коли важливіша швидкість, а не стійкість.

Сортування злиттям (MergeSort):

- Часова складність: Постійно O(n log n), навіть у найгіршому випадку.
- Просторова складність: Вимагає додаткової пам'яті.
- Стійкість: Стійкий.
- Найкраще працює: Для великих масивів, коли потрібна стійкість, або для паралельних обчислень.

6. Порівняльна характеристика пірамідального та плавного сортування
Пірамідальне сортування (HeapSort):

- Часова складність: Постійно O(n log n).
- Просторова складність: Низька, потребує лише O(1) додаткової пам'яті.
- Стійкість: Нестійкий.
- Найкраще працює: Для великих масивів, коли потрібна ефективність і низькі вимоги до пам'яті.

Плавне сортування (SmoothSort):

- Часова складність: O(n log n), але може бути O(n) при майже відсортованих даних.
- Просторова складність: Низька, подібна до пірамідального сортування.
- Стійкість: Нестійкий.
- Найкраще працює: Для великих масивів, коли дані можуть бути майже відсортовані.

Висновок: У цій лабораторній роботі було проведено дослідження різних алгоритмів сортування, зокрема тих, які добре підходять для сортування великих обсягів даних. Основною метою було порівняти ефективність алгоритмів і визначити, в яких ситуаціях кожен з них є найбільш підходящим. Ці алгоритми відомі своєю здатністю ефективно сортувати великі обсяги даних, хоча вони відрізняються

підходами та особливостями реалізації. Загалом, ця лабораторна робота надала розуміння різних методів сортування великих обсягів даних та їх порівняння. Це допоможе у виборі оптимального алгоритму для різних завдань у практичному застосуванні.