МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

Институт информационных технологий

Кафедра информационных технологий и экономической информатики

ОТЧЕТ

по лабораторной работе №1

|  |  |  |  |
| --- | --- | --- | --- |
| Авторы отчета: | подпись | А.А. Аблин  инициалы, фамилия | ПрИ-202  группа |
| подпись | Ю.Д. Аетбаева  инициалы, фамилия | ПрИ-202  группа |
| подпись | А.Д. Григорьева  инициалы, фамилия | ПрИ-202  группа |

|  |  |  |
| --- | --- | --- |
| Отчет защищен: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  дата | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  оценка |

Челябинск, 2024

**Цель работы:** Эмпирический анализ временной сложности алгоритмов. Вводиться понятие временной сложности алгоритма, рассматривается математический аппарат для оценки временной сложности и правила применения этого аппарата.

**Задание:** Для каждого n от 1 до 2000 произведите для пяти запусков замер среднего машинного времени исполнения программ, реализующих нижеуказанные алгоритмы и функции. Изобразите на графике полученные данные, отражающие зависимость среднего времени исполнения от n. Проведите теоретический анализ временной сложности рассматриваемых алгоритмов и сравните эмпирическую и теоретическую временные сложности.

I. Сгенерируйте n-мерный случайный вектор v = [v1, v2, . . ., vn] с неотрицательными элементами. Для полученного вектора v осуществите подсчет функций и реализацию алгоритмов:

1. (Постоянная функция)

2. (Сумма элементов)

3. (Произведение элементов)

4. Полагая, что элементы – коэффициенты многочлена степени , вычислите значение путем прямого (наивного) вычисления для (т.е. оценивая каждый член по одному) и методом Горнера представление полинома: ;

5. Алгоритм сортировки пузырьком (Bubble sort) элементов ;

6. Алгоритм быстрой сортировки (Quick sort) элементов ;

7. Гибридный алгоритм сортировки Timsort элементов ;

8. Алгоритмы возведения в степень.

II. Сгенерируйте случайные матрицы и размером с неотрицательными элементами. Найдите обычное матричное произведение матриц и .

III. На каждого члена команды найти алгоритм не ниже линейного класса сложности и провести с ним эксперимент.

**Стек технологий:**

1. С#
2. WPF
3. Библиотека ScottPlot для отрисовки графиков

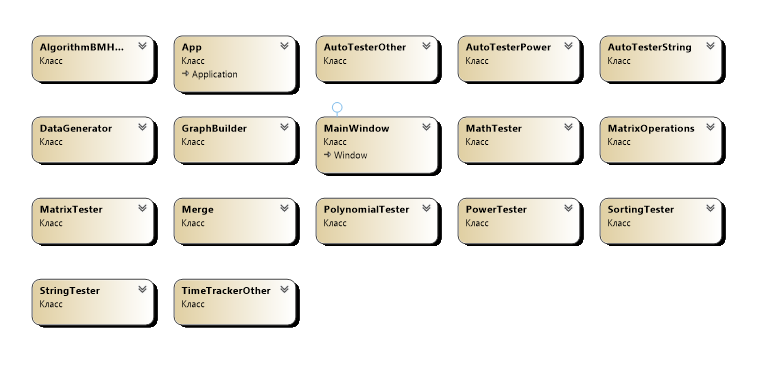
Подробнее о каждом стеке:

1. Объектно-ориентированный язык программирования общего назначения.
2. WPF предоставляет средства для создания визуального интерфейса, включая язык XAML (eXtensible Application Markup Language), элементы управления, привязку данных, макеты, двухмерную и трёхмерную графику, анимацию, стили, шаблоны, документы, текст, мультимедиа и оформление.
3. Библиотека с открытым исходным кодом для .NET. Используется для построения графиков, упрощая интерактивное отображение больших наборов данных.

Архитектура программы

С помощью встроенного в Visual Studio конструктора классов, были созданы следующие диаграммы: (Рис. 0.1 и 0.2)

|  |
| --- |
|  |
| Рис. 0.1 |



|  |
| --- |
|  |
| Рис. 0.2 |

На них наглядно показана зависимости и взаимодействия между классами программы.

Задание I.

На вход подавались массивы размеров от 1 до N элементов. Замер времени происходил для каждого массива. Подсчеты проводились с помощью параллельных вычислений.

1. (Постоянная функция)

* Временная сложность: O(1)
* N = 50 000
* Количество повторов: 5

Обращение к последнему элементу массива. (Рис. 1.1)

|  |
| --- |
|  |
| Рис. 1.1 Алгоритм постоянной функции |

Код представлен ниже:

**public** **class** **ConstFunction** : BasicOperation

{

**public** **override** **long** **PerformOperation**(**params** **int**[] args)

{

**return** args[args.Length - **1**];

}

}

2. (Сумма элементов)

* Временная сложность: O(n)
* N = 100 000
* Количество повторов: 5

Алгоритм суммирует все элементы массива.

График зависимости времени от размера данных (Рис. 1.2)

|  |
| --- |
|  |
| Рис. 1.2 Алгоритм суммы элементов |

Код алгоритма представлен ниже:

**public** **class** **SumNumbers** : BasicOperation

{

**public** **override** **long** **PerformOperation**(**params** **int**[] args)

{

**int** result = args[**0**];

**for** (**int** i = **1**; i < args.Length; i++)

{

result += args[i];

}

**return** result;

}

}

1. (Произведение элементов)

* Временная сложность: O(n)
* N = 100 000
* Количество повторов: 5

Алгоритм считает произведение всех элементов массива.

График зависимости времени от размера данных (Рис. 1.3)

|  |
| --- |
|  |
| Рис. 1.3 Алгоритм произведения элементов |

Код алгоритма представлен ниже:

**public** **class** **MultiplyNumbers** : BasicOperation

{

**public** **override** **long** **PerformOperation**(**params** **int**[] args)

{

**int** result = args[**0**];

**for** (**int** i = **1**; i < args.Length; i++)

{

result \*= args[i];

}

**return** result;

}

}

* Временная сложность Горнера: O(n)
* Временная сложность прямой: O(n2)
* NГорнера = 5000
* Nпрямой  = 5000
* Количество повторов: 5

Алгоритм Горнера

Алгоритм Горнера - метод для эффективного вычисления значений многочлена в заданной точке.

График зависимости времени от размера данных (Рис. 1.4.1):

|  |
| --- |
|  |
| Рис. 1.4.1 Алгоритм Горнера |

Код алгоритма представлен ниже:

**public** **class** **HornerPolynomial** : PolynomialOperations

{

**public** **override** **double** **Evaluate**(**double** x, **double**[] coefficients)

{

**double** result = **0**;

**int** n = coefficients.Length;

**for** (**int** k = n - **1**; k >= **0**; k--)

{

result = result \* x + coefficients[k];

}

**return** result;

}

}

Прямой (наивный) метод

График зависимости времени выполнения алгоритма Прямым (наивным) методом от размера данных (Рис. 1.4.2):

|  |
| --- |
|  |
| Рис. 1.4.2 Прямой (наивный) метод |

Код алгоритма представлен ниже:

**public** **class** **NaivePolynomial** : PolynomialOperations

{

**public** **override** **double** **Evaluate**(**double** x, **double**[] coefficients)

{

**double** result = **0**;

**int** n = coefficients.Length;

// Формула вычисления P(x) = sum(v\_k \* x^(k-1))

**for** (**int** k = **0**; k < n; k++)

{

result += coefficients[k] \* Math.Pow(x, k);

}

**return** result;

}

}

Изучив два вышеприведенных графика, можно сделать вывод, что метод Горнера намного эффективнее прямого способа (примерно в 7-8 раз).

1. Алгоритм сортировки пузырьком (Bubble sort) элементов

* Временная сложность: O(n2)
* N = 10 000
* Количество повторений: 5

Псевдокод алгоритма:

function **bubbleSort**(a):

**for** i = **0** to n - **2**

**for** j = **0** to n - **2**

**if** a[j] > a[j + **1**]

swap(a[j], a[j + **1**])

Алгоритм последовательно сравнивает значения соседних элементов и меняет значения местами, если предыдущее оказывается больше последующего. Сортировка пузырьком востребована в компьютерной графике благодаря своей способности обнаруживать очень маленькую ошибку в отсортированных массивах

График зависимости времени выполнения алгоритма от размера данных (Рис. 1.5):

|  |
| --- |
|  |
| Рис. 1.5 Bubble sort |

Код алгоритма представлен ниже:

**public** **class** **BubbleSort** : SortingAlgorithm

{

**public** **override** **void** **Sort**(**int**[] array)

{

**int** temp;

**for** (**int** j = **0**; j < array.Length - **1**; j++)

{

**for** (**int** i = **0**; i < array.Length - j - **1**; i++)

{

**if** (array[i] > array[i + **1**])

{

(array[i], array[i + **1**]) = (array[i + **1**], array[i]);

}

}

}

}

}

1. Алгоритм быстрой сортировки (Quick Sort) элементов

* Временная сложность: O(n\*log n)
* N = 20 000
* Количество повторений: 5

Псевдокод алгоритма:

algorithm **quicksort**(A, low, high) **is**

**if** low < high then

p:= partition(A, low, high)

quicksort(A, low, p - **1**)

quicksort(A, p, high)

Отличие алгоритма Quick Sort от других алгоритмов состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы. Этот алгоритм широко используется благодаря своей эффективности. Также, как и сортировка пузырьком, популярен в компьютерной графике.

График зависимости времени выполнения быстрой сортировки от объёма данных (Рис. 1.6):

|  |
| --- |
|  |
| Рис. 1.6 Quick Sort |

Код алгоритма представлен ниже:

**private** **void** **Quicksort**(**int**[] array, **int** leftIndex, **int** rightIndex)

{

**var** i = leftIndex;

**var** j = rightIndex;

**var** pivot = array[leftIndex];

**while** (i <= j)

{

**while** (array[i] < pivot)

{

i++;

}

**while** (array[j] > pivot)

{

j--;

}

**if** (i <= j)

{

(array[i], array[j]) = (array[j], array[i]);

i++;

j--;

}

}

**if** (leftIndex < j)

Quicksort(array, leftIndex, j);

**if** (i < rightIndex)

Quicksort(array, i, rightIndex);

}

1. Гибридный алгоритм сортировки Timsort элементов

* Временная сложность: O(n\*log n)
* N = 10 000
* Количество повторений: 5

Метод сортировки Timsort - это гибридный алгоритм, который сочетает в себе элементы сортировки вставками и сортировки слиянием. Алгоритм построен на идеи, что сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. Следуя этой логике, он разбивает входной массив на подмассивы и сортирует их, используя сортировку вставками. После сортировки вставками объединяет подмассивы попарно сортировкой слиянием и возвращает отсортированный массив.

Timsort используется в качестве сортировки по умолчанию в Python, Java и на платформе Android.

График зависимости времени выполнения Timsort от объема данных (Рис. 1.7):

|  |
| --- |
|  |
| Рис. 1.7 Timsort |

Код алгоритма представлен ниже:

**public** **class** **TimSort** : SortingAlgorithm

{

// Сортировка TimSort

**private** **static** **int** MIN\_MERGE = **32**;

**public** **override** **void** **Sort**(**int**[] array)

{

// Выполняем сортировку вставками на подмассивах размером MIN\_MERGE

InsertionSort insertionSort = **new** InsertionSort();

**for** (**int** i = **0**; i < array.Length; i += MIN\_MERGE)

{

insertionSort.Sort(array);

}

// Выполняем слияние отсортированных подмассивов

Merge merge = **new** Merge();

**for** (**int** size = MIN\_MERGE; size < array.Length; size \*= **2**)

{

**for** (**int** left = **0**; left < array.Length; left += **2** \* size)

{

**int** mid = left + size - **1**;

**int** right = Math.Min(left + **2** \* size - **1**, array.Length - **1**);

**if** (mid < right)

{

merge.MergeSort(array, left, mid, right);

}

}

}

}

}

**if** (leftIndex < j)

Quicksort(array, leftIndex, j);

**if** (i < rightIndex)

Quicksort(array, i, rightIndex);

}

1. Алгоритмы возведения в степень
2. Классическое возведение в степень

* N = 10 000
* Основание степени: 2

|  |
| --- |
|  |
| Рис. 1.8.1 Классическое возведение в степень |

Код алгоритма представлен ниже:

**public** **class** **SimplePow** : PowerOperation

{

// Линейное возведение в степень

**public** **override** (**long** result, **int** steps) CalculatePower(**int** x, **int** n)

{

**long** result = **1**;

**int** steps = **0**;

**for** (**int** i = **0**; i < n; i++)

{

result \*= x;

steps++;

}

**return** (result, steps);

}

}

1. Рекурсивный метод возведения в степень

* N = 10 000
* Основание степени: 2

|  |
| --- |
|  |
| Рис. 1.8.2 Рекурсивный метод возведения в степень |

Код алгоритма представлен ниже:

**public** **class** **RecPow** : PowerOperation

{

// Метод для рекурсивного возведения в степень с подсчетом шагов

**public** **override** (**long** result, **int** steps) CalculatePower(**int** x, **int** n)

{

**int** steps = **0**;

**return** (RecursivePower(x, n, **ref** steps), steps);

}

**private** **long** **RecursivePower**(**int** x, **int** n, **ref** **int** steps)

{

steps++;

**if** (n == **0**)

**return** **1**;

**long** half = RecursivePower(x, n / **2**, **ref** steps);

**if** (n % **2** == **0**)

**return** half \* half;

**else**

**return** half \* half \* x;

}

}

1. Быстрое возведение в степень

* N = 10 000
* Основание степени: 2

|  |
| --- |
|  |
| Рис. 1.8.3 Быстрое возведение в степень |

Код алгоритма представлен ниже:

**public** **class** **QuickPowFaster** : PowerOperation

{

// Оптимизированное быстрое возведение в степень с подсчетом шагов

**public** **override** (**long** result, **int** steps) CalculatePower(**int** x, **int** n)

{

**long** result = **1**;

**int** steps = **0**;

**if** (n % **2** == **1**)

{

result = x;

}

**while** (n > **0**)

{

steps++;

n /= **2**;

x \*= x;

**if** (n % **2** == **1**)

{

result \*= x;

}

}

**return** (result, steps);

}

}

1. Быстрое классическое возведение в степень

* N = 10 000
* Основание степени: 2

|  |
| --- |
|  |
| Рис. 1.8.3 Быстрое классическое возведение в степень |

Код алгоритма представлен ниже:

**public** **class** **QuickPowClassic** : PowerOperation

{

// Быстрое возведение в степень с подсчетом шагов

**public** **override** (**long** result, **int** steps) CalculatePower(**int** x, **int** n)

{

**long** result = **1**;

**int** steps = **0**;

**while** (n > **0**)

{

steps++;

**if** (n % **2** == **1**)

{

result \*= x;

}

x \*= x;

n /= **2**;

}

**return** (result, steps);

}

}

Задание II

Сгенерируйте случайные матрицы A и B размером n x n с неотрицательными

элементами. Найдите обычное матричное произведение матриц A и B.

* Временная сложность: O(n2)
* N = 500
* Количество повторов: 5

Генерировались матрицы размером и считались с помощью скалярного произведения.

График зависимости времени от размера данных (Рис. 2.1):

|  |
| --- |
|  |
| Рис. 2.1 Произведение матриц |

Код алгоритма:

**public** **class** **MatrixOperations**

{

// Метод для умножения двух матриц

**public** **int** **MatrixMultiply**(**int**[,] matrix1, **int**[,] matrix2)

{

**int** rows = matrix1.GetLength(**0**);

**int** cols = matrix2.GetLength(**1**);

**int** innerDim = matrix1.GetLength(**1**);

**int**[,] resultMatrix = **new** **int**[rows, cols];

**int** scalarResult = **0**;

// Умножаем матрицы и сохраняем в resultMatrix

**for** (**int** i = **0**; i < rows; i++)

{

**for** (**int** j = **0**; j < cols; j++)

{

**int** sum = **0**;

**for** (**int** k = **0**; k < innerDim; k++)

{

sum += matrix1[i, k] \* matrix2[k, j];

}

resultMatrix[i, j] = sum;

scalarResult += resultMatrix[i, j];

}

}

**return** scalarResult;

}

}

Задание III

**1. Tree Sort** - алгоритм сортировки, который основан на использовании двоичных деревьев поиска.

* Временная сложность: O(n\*log n)
* N = 10 000
* Количество повторов: 5

Основная идея заключается в следующем:

Создание дерева: алгоритм начинает с создания двоичного дерева поиска. Каждый элемент массива вставляется в дерево. При вставке элемента сравнивается его значение с текущим узлом: если значение меньше, элемент помещается в левое поддерево, если больше — в правое.

Обход дерева: после того как все элементы были вставлены, осуществляется обход дерева в симметричном порядке. При таком обходе элементы выводятся в отсортированном порядке.

Сортировка таким методом оптимальна для получения данных путем чтения из потока (например, с файла, консоли и тд).

График зависимости времени выполнения Tree sort массива от объема данных (Рис. 3.1):

|  |
| --- |
|  |
| Рис. 3.1 Tree Sort |

Код алгоритма:

//рекурсивное добавление узла в дерево

**public** **void** **Insert**(TreeNode node)

{

**if** (node.Data < Data)

{

**if** (Left == **null**)

{

Left = node;

}

**else**

{

Left.Insert(node);

}

}

**else**

{

**if** (Right == **null**)

{

Right = node;

}

**else**

{

Right.Insert(node);

}

}

}

//преобразование дерева в отсортированный массив

**public** **int**[] **Transform**(List<**int**> elements = **null**)

{

**if** (elements == **null**)

{

elements = **new** List<**int**>();

}

**if** (Left != **null**)

{

Left.Transform(elements);

}

elements.Add(Data);

**if** (Right != **null**)

{

Right.Transform(elements);

}

**return** elements.ToArray();

}

}

Часть кода №1

**public** **override** **void** **Sort**(**int**[] array)

{

**var** treeNode = **new** TreeNode(array[**0**]);

**for** (**int** i = **1**; i < array.Length; i++)

{

treeNode.Insert(**new** TreeNode(array[i]));

}

**int**[] sorted\_array = treeNode.Transform();

}

Часть кода №2

**2. Pancake Sort.**

* Временная сложность: O(n2)
* N = 10 000
* Количество повторов: 5

Основные этапы работы Pancake Sort:

1. Поиск максимального элемента: Алгоритм начинает с поиска наибольшего элемента в текущем неотсортированном массиве.
2. Переворот: если максимальный элемент не находится на первой позиции, он переводится на первую позицию с помощью переворота всего подмассива от начала до его текущей позиции.
3. Переворот для окончательной сортировки: затем максимальный элемент переворачивается еще раз, чтобы поместить его в конец массива (в отсортированную часть).
4. Повторение: Эти операции повторяются для оставшейся части массива (исключая уже отсортированные элементы), пока весь массив не будет отсортирован.

Блинная сортировка интересна с теоретической стороны и показывает, как задачи работы с объектами решаются через ограниченный набор операций.

График зависимости времени от размера данных (Рис. 3.2):

|  |
| --- |
|  |
| Рис. 3.2 Pancake sort |

Код алгоритма:

**public** **override** **void** **Sort**(**int**[] array)

{

**int** n = array.Length;

**for** (**int** curr\_size = n; curr\_size > **1**; curr\_size--)

{

**int** maxIndex = FindMax(array, curr\_size);

**if** (maxIndex != curr\_size - **1**)

{

**if** (maxIndex != **0**)

{

Flip(array, maxIndex);

}

Flip(array, curr\_size - **1**);

}

}

}

// Метод для переворота массива от 0 до k

**private** **void** **Flip**(**int**[] array, **int** k)

{

**int** start = **0**;

**while** (start < k)

{

(array[start], array[k]) = (array[k], array[start]);

start++;

k--;

}

}

// Метод для поиска индекса максимального элемента в массиве[0..n-1]

**private** **int** **FindMax**(**int**[] array, **int** n)

{

**int** maxIndex = **0**;

**for** (**int** i = **1**; i < n; i++)

{

**if** (array[i] > array[maxIndex])

{

maxIndex = i;

}

}

**return** maxIndex;

}

**3. Алгоритм Бойера-Мура-Хорспула**

* Длина строки = 5000
* Длина искомой подстроки = 100

Алгоритм для эффективного поиска подстроки в строке. Он представляет собой модифицированный вариант классического алгоритма Бойера-Мура.

Общий смысл работы алгоритма:

1. Алгоритм начинает проверку шаблона с самого правого символа, сравнивая его с символом в строке.
2. Если все символы совпадают, поиск завершается с нахождением подстроки. Если возникает несовпадение, алгоритм использует ранее подготовленные таблицы для определения наименьшего сдвига шаблона.
3. Относительно текущего положения символа в строке, шаблон сдвигается вправо, в зависимости от значения, найденного в таблицах.
4. Процесс повторяется до тех пор, пока не будет найдено совпадение или не будут проверены все возможные позиции строки.

График зависимости количества шагов от длины строки (Рис. 3.2).

|  |
| --- |
|  |
| Рис. 3.2 Алгоритм Бойера-Мура-Хорспула |

Код алгоритма:

// Метод для создания таблицы смещений

**private** **static** Dictionary<**char**, **int**> GetTableOffset(**string** pattern)

{

Dictionary<**char**, **int**> table = **new** Dictionary<**char**, **int**>();

**string** alf = **string**.Empty;

**for** (**int** i = pattern.Length - **2**; i > -**1**; i--)

{

**if** (!alf.Contains(pattern[i]))

{

table[pattern[i]] = pattern.Length - i - **1**;

alf += pattern[i];

}

}

**if** (!alf.Contains(pattern[pattern.Length - **1**]))

{

table[pattern[pattern.Length - **1**]] = pattern.Length;

}

// Добавляем символ '\*' если он ещё не был добавлен

**if** (!table.ContainsKey('\*'))

{

table.Add('\*', pattern.Length);

}

**return** table;

}

Часть кода №1

// Метод для поиска индексов

**public** (List<**int**>, **int**) GetIndexes(**string** text, **string** pattern)

{

List<**int**> result = **new** List<**int**>();

**int** textLen = text.Length;

**int** patternLen = pattern.Length;

**int** stepCount = **0**; // Счётчик шагов

**int** currentPosition = **0**;

**while** (currentPosition <= textLen - patternLen)

{

**int** indexPattern = patternLen - **1**;

**while** (indexPattern >= **0** && text[currentPosition + indexPattern] == pattern[indexPattern])

{

stepCount++;

indexPattern--;

}

**if** (indexPattern < **0**)

{

result.Add(currentPosition);

currentPosition++;

}

**else**

{

stepCount++;

**if** (\_tableOffset.ContainsKey(text[currentPosition + patternLen - **1**]))

{

currentPosition += \_tableOffset[text[currentPosition + patternLen - **1**]];

}

**else**

{

currentPosition += pattern.Length;

}

}

}

**return** (result, stepCount);

}

Часть кода №2