

МИНОБРНАУКИ РОССИИ

*Федеральное государственное бюджетное образовательное учреждение высшего образования*

***«МИРЭА – Российский технологический университет»***

Отчет

Практическая работа №8

Дисциплина Структуры и алгоритмы обработки данных

Тема. Определение эффективного алгоритма сортировки

|  |  |
| --- | --- |
| Выполнил студент | Смольников А.Б. |
|  | Фамилия И.О. |
| Группа | ИКБО-13-21 |
|  | Номер группы |

**Москва 2022**

**Задание 1**

**Вариант №20**

**Оценить зависимость времени выполнения алгоритма простой сортировки на массиве, заполненном случайными числами (средний случай)**

1. Разработать алгоритм сортировки одномерного целочисленного массива A[n] и реализовать его функцией, используя алгоритм согласно варианту, индивидуального задания – *сортировка простого выбора.* Провести тестирование программы на исходном массиве, сформированном вводом с клавиатуры, т.е. доказать ее работоспособность.
2. Разработать функцию заполнения рабочего массива A с использованием генератора псевдослучайных чисел.
3. Провести экспериментальную оценку вычислительной сложности алгоритма, для чего выполнить контрольные прогоны программы для размеров массива n = 100, 1000, 10000, 100000 и 200000 элементов с вычислением времени T(n) выполнения T(n) – (в миллисекундах/секундах). Полученные результаты свести в сводную таблицу Таблица 1.
4. Провести эмпирическую оценку вычислительной сложности алгоритма, определив функцию зависимости времени выполнения алгоритма от размера массива(задачи) и показать ее результат в таблице 1 в столбце **Тэт** =f(C+M).
5. Провести эмпирическую оценку фактического количества операций сравнения Сф и количества операций перемещения Мф. Полученные результаты вставить в сводную таблицу в столбец Тэп=Сф+Мф.

**Разработка алгоритма и его реализация**

Необходимо разработать алгоритм сортировки целочисленного массива методом простого выбора.

Алгоритм: Сортировка методом простого выбора строится на идее сравнивания минимального элемента с элементами, которые стоят после него. Если нашелся элемент, меньше локального минимума, обновим локальный минимум и поменяем местами с элементом бывшего локального минимума.

Внешним циклом пройдем от нулевого до n-1 элемента. В переменную min\_id запишем индекс i. В теле внешнего цикла создадим внутренний цикл от элемента i-1 до n-того элемента. Если элемент массива с индексом j меньше элемента массива с индексом min\_id, обновим min\_id. Поменяем местами arr[min\_id] и arr[i].

Определим «О-большое» данная характеристика определяет временную эффективность алгоритма в зависимости от количества входных данных. О-большое для данной сортировки составит:

Так как в худшем случае придется рассмотреть все элементы в двумерном цикле.

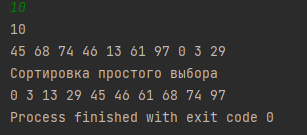
Рассмотрим код функции сортировки:

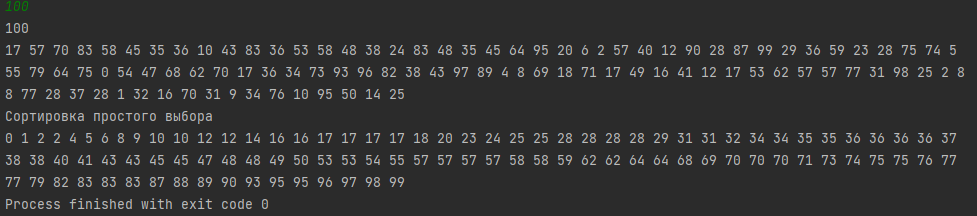
Предусловие – массив целых чисел arr, размерность n

Постусловие – отсортированный массив

1. void selectionSort(int \*arr, int n)  
   {  
    for (int i = 0; i < n - 1; i++)  
    {  
    int min\_id = i;  
     
    for (int j = i + 1; j < n; j++)  
    if (arr[j] < arr[min\_id])  
    min\_id = j;  
     
    int temp = arr[min\_id];  
    arr[min\_id] = arr[i];  
    arr[i] = temp;  
    //if(i%1000==0) cout<<"Selection sort: "<<i<<"/"<<n<<endl;  
    }  
   }

Рассмотрим тестовый пример для 10 и 100 элементов:





Определим зависимость времени от количества входных данных эмпирически. Примерное количество операций, выполняемых С++ за миллисекунду – 1000 000. В действительности эта цифра зависит от характеристик ВМ, типа операций, оптимизации самого компилятора. Тогда рассчитаем примерное время сортировки как:

Рассмотрим эмпирически количество выполняемых операций. В соответствии с кодом задачи, получим:

***Таблица 1. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.3998 | 0.01 | 9900 |
| 1000 | 2.2904 | 1 | 999000 |
| 10000 | 120.383 | 100 | 99990000 |
| 100000 | 12001.5 | 10000 | 9999900000 |
| 1000000 | 1.21896e+06 | 1e+06 | 999999000000 |

Project Path: C:\Users\Алексей\Downloads\UNTITLED.opju
PE Folder: /UNTITLED/Folder1/
Short Name: Graph2

График построен в программе OriginPro 2022. Отчетливо видна квадратичная зависимость всех графиков. Расхождения с теоретическим расчетом времени выполнения пренебрежимо малы.

Теперь рассмотрим работу алгоритма на отсортированном по возрастанию массиве (лучший случай). Данные уже отсортированы, так что массив до и после вызова функции не меняется. Рассмотрим таблицу 1

Сортировка методом простого выбора в худшем, лучшем и среднем случае имеет одинаковую асимптотическую зависимость

В таком случае:

***Таблица 2. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.0129 | 0.01 | 9900 |
| 1000 | 1.1662 | 1 | 999000 |
| 10000 | 115.19 | 100 | 99990000 |
| 100000 | 12198.9 | 10000 | 9999900000 |
| 1000000 | 1.21194e+06 | 1e+06 | 999999000000 |

Составим графики:

Project Path: C:\Users\Алексей\Downloads\UNTITLED.opju
PE Folder: /UNTITLED/Folder1/
Short Name: Graph2

Аналогично с предыдущим пунктом Расхождения с теоретическим расчетом времени выполнения пренебрежимо малы. Графики имеют квадратичную зависимость.

Рассмотрим худший случай:

Сортировка методом простого выбора в худшем, лучшем и среднем случае имеет одинаковую асимптотическую зависимость

В таком случае:

***Таблица 3. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.016 | 0.01 | 9900 |
| 1000 | 1.2151 | 1 | 999000 |
| 10000 | 162.389 | 100 | 99990000 |
| 100000 | 12272.3 | 10000 | 9999900000 |
| 1000000 | 1.32176e+06 | 1e+06 | 999999000000 |

Составим графики:

Project Path: D:\Clion\Siaod8\Siaod8Graph.opju
PE Folder: /Siaod8Graph/Folder1/
Short Name: Graph2

Экспериментально рассмотрев зависимость времени сортировки от расположения элементов массива, можно сделать вывод, что для сортировки простого выбора упорядоченность элементов исходного массива незначительно влияет на время выполнения сортировки.

Проанализируем полученные данные: для всех трех случаев (случайный, лучший, худший) время выполнения имеет строгую квадратичную зависимость, для одинакового количества входных данных время в зависимости от случая почти не изменяется.

**Задание 2. Оценить зависимость времени выполнения алгоритма усовершенствованной сортировки на массиве**

1. Разработать алгоритм усовершенствованной сортировки (задача 2), определенной в варианте, реализовать алгоритм. Сформировать таблицу Таблица 4 результатов сортировки по формату Таблица 1 для массива, заполненного случайными числами. Определить емкостную сложность алгоритма. *Определить асимптотическую сложность алгоритма.*
2. Провести дополнительные прогоны программы для оценки эффективности алгоритмов в наилучшем и наихудшем случаях. Сформировать таблицы 5 и 6.
3. Выполнить анализ полученных результатов по таблицам 4, 5, 6.
4. Определить эффективный из алгоритмов (задача 1) и (задача 2) по временной и емкостной сложности.
5. Представить график зависимости Сф+Мф для анализируемых алгоритмов.

В соответствии с персональным вариантом разработаем алгоритм *сортировки бинарными включениями* (insertionSortBinary).

Общая суть такова: перебираются элементы в неотсортированной части массива, каждый элемент вставляется в отсортированную часть массива на то место, где он должен находиться. В таком случае идея бинарного поиска отлично сработает.

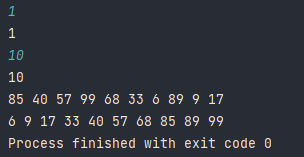
Реализуем алгоритм на C++:

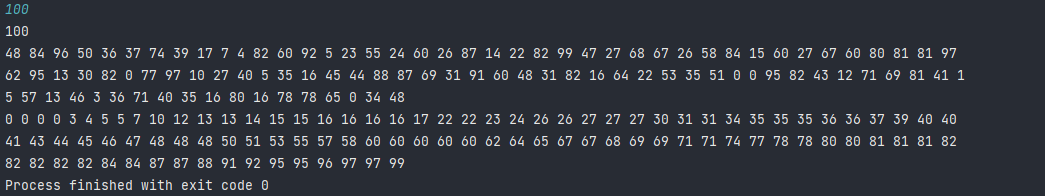
Предусловие – массив целых чисел arr, размерность n

Постусловие – отсортированный массив

1. void InsertionSortBinary(int \*arr, int n){
2. int current;
3. int left, right, mid;
4. for (int i = 0; i < n; i++){
5. current = arr[i];
6. left = 0;
7. right = i;
8. while (left < right)
9. {
10. mid = (left + right) / 2;
11. if (arr[mid] <= current){
12. left = mid + 1;
13. }else{
14. right = mid;
15. }
16. }
17. for (int j = i; j > left; j--){
18. arr[j] = arr[j - 1];
19. }
21. arr[left] = current;
22. }
23. }

Рассмотрим тестовый пример для 10 и 100 элементов:





Определим зависимость времени от количества входных данных эмпирически. Примерное количество операций, выполняемых С++ за миллисекунду – 1 000 000. В действительности эта цифра зависит от характеристик ВМ, типа операций, оптимизации самого компилятора. Тогда рассчитаем примерное время сортировки как:

Рассмотрим эмпирически количество выполняемых операций. В соответствии с кодом задачи, получим:

Таким образом, асимптотическая сложность бинарной сортировки вставками:

Анализ емкостной сложности:

Для данного алгоритма сортировки требуется только постоянное количество дополнительного пространства. Мы сортируем данный массив, сдвигая и вставляя элементы.

Следовательно, пространственная сложность этого алгоритма равна O (1), если мы используем итеративный двоичный поиск.

***Таблица 4. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.0118 | 0,00651 | 6011 |
| 1000 | 0.5231 | 0,517 | 512408 |
| 10000 | 42.9441 | 50,197 | 50147104 |
| 100000 | 5693.61 | 5002,201 | 5001701293 |
| 1000000 | 650432 | 500024,31 | 5,00019E+11 |

Project Path: D:\Clion\Siaod8\Siaod8Graph.opju
PE Folder: /Siaod8Graph/Folder1/
Short Name: Graph2

Отчетливо заметна квадратичная зависимость роста времени от количества входных данных.

Рассмотрим наилучший случай (исходный массив отсортирован по возрастанию):

В таком случае бинарный поиск будет сильно упрощен, асимптотическая зависимость:

***Таблица 5. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время, ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.2337 | 0,00046 | 461 |
| 1000 | 0.2256 | 0,0069 | 6908 |
| 10000 | 0.5468 | 0,092 | 92104 |
| 100000 | 6.623 | 1,151 | 1151293 |
| 1000000 | 71.3634 | 13,81 | 13815511 |

Project Path: D:\Clion\Siaod8\Siaod8Graph.opju
PE Folder: /Siaod8Graph/Folder1/
Short Name: Graph5Project Path: D:\Clion\Siaod8\Siaod8Graph.opju
PE Folder: /Siaod8Graph/Folder1/
Short Name: Graph6

Из двух построенных графиков можно увидеть, что экспериментальное время так же соответствует функции O(n ln(n)). Имеются расхождения с теоретически вычисленными значениями, они вызваны особенностями выполнения программного кода на данной ВМ.

**Рассмотрим наихудший случай для сортировки бинарным выбором.**

В таком случае:

***Таблица 5. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время, ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.3 | 0.1 | 10000 |
| 1000 | 1.3102 | 1 | 1000000 |
| 10000 | 113.551 | 100 | 100000000 |
| 100000 | 12646 | 10000 | 10000000000 |
| 1000000 | 1.34937e+06 | 1e+06 | 1000000000000 |

Project Path: D:\Clion\Siaod8\Siaod8Graph.opju
PE Folder: /Siaod8Graph/Folder1/
Short Name: Graph5Project Path: D:\Clion\Siaod8\Siaod8Graph.opju
PE Folder: /Siaod8Graph/Folder1/
Short Name: Graph7

На построенных графиках отчетливо видна квадратичная зависимость. Расхождения связаны с особенностями выполнения программного кода на данной ВМ.

**Задание 3. Оценить зависимость времени выполнения алгоритма быстрой сортировки на массиве**

1. Разработать алгоритм усовершенствованной сортировки (задача 3), определенной в варианте, реализовать алгоритм. Сформировать таблицу Таблица 4 результатов сортировки по формату Таблица 1 для массива, заполненного случайными числами. Определить емкостную сложность алгоритма. *Определить асимптотическую сложность алгоритма.*
2. Провести дополнительные прогоны программы для оценки эффективности алгоритмов в наилучшем и наихудшем случаях. Сформировать таблицы 5 и 6.
3. Выполнить анализ полученных результатов по таблицам 4, 5, 6.
4. Определить эффективный из алгоритмов (задача 1) и (задача 2) по временной и емкостной сложности.
5. Представить график зависимости Сф+Мф для анализируемых алгоритмов.

В соответствии с персональным вариантом разработаем алгоритм *сортировки слиянием* (merge sort).

Общая суть такова: исходный массив рекурсивно разделяется на правую и левую часть, которые потом сливаются в один отсортированный

Реализуем алгоритм на C++:

Предусловие – массив целых чисел arr, размерность n

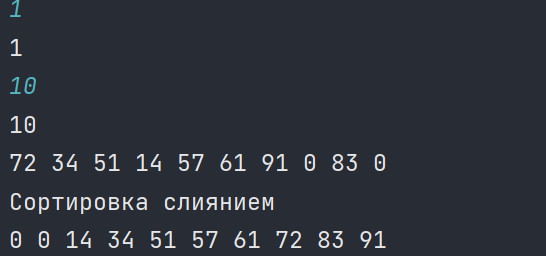
Постусловие – отсортированный массив

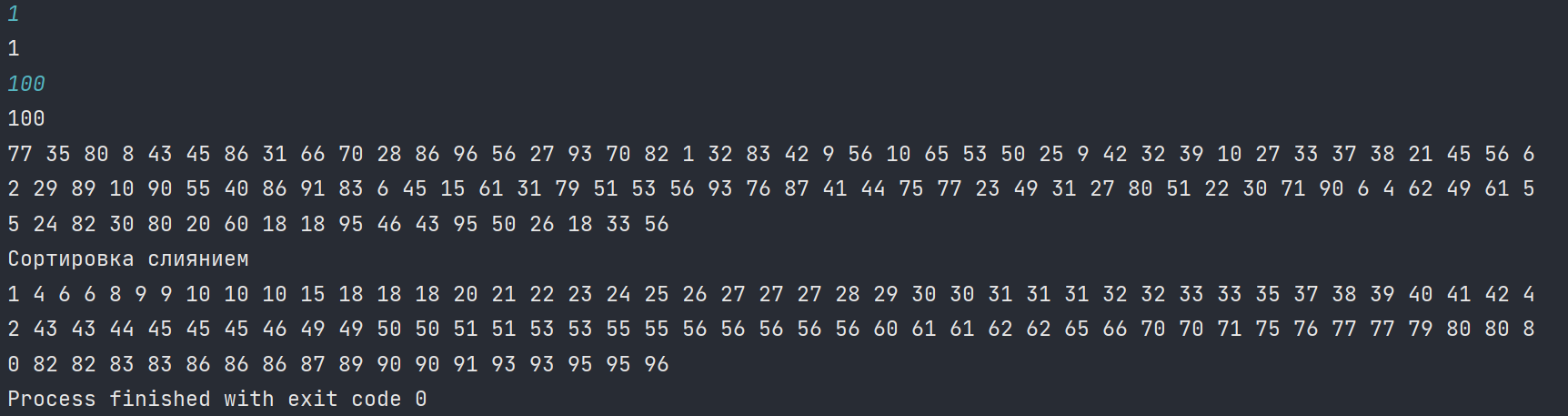
1. void merge(int \*array, int const left, int const mid, int const right)
2. {
3. int const subArrayOne = mid - left + 1;
4. int const subArrayTwo = right - mid;
6. auto \*leftArray = new int[subArrayOne],
7. \*rightArray = new int[subArrayTwo];
9. for (auto i = 0; i < subArrayOne; i++)
10. leftArray[i] = array[left + i];
11. for (auto j = 0; j < subArrayTwo; j++)
12. rightArray[j] = array[mid + 1 + j];
14. int indexOfSubArrayOne = 0, indexOfSubArrayTwo = 0;
15. int indexOfMergedArray = left;

18. while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
19. if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
20. array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
21. indexOfSubArrayOne++;
22. }
23. else {
24. array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
25. indexOfSubArrayTwo++;
26. }
27. indexOfMergedArray++;
28. }
30. while (indexOfSubArrayOne < subArrayOne) {
31. array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
32. indexOfSubArrayOne++;
33. indexOfMergedArray++;
34. }
36. while (indexOfSubArrayTwo < subArrayTwo) {
37. array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
38. indexOfSubArrayTwo++;
39. indexOfMergedArray++;
40. }
41. delete[] leftArray;
42. delete[] rightArray;
43. }

46. void mergeSort(int \*array, int const begin, int const end)
47. {
48. if (begin >= end)
49. return;
51. auto mid = begin + (end - begin) / 2;
52. mergeSort(array, begin, mid);
53. mergeSort(array, mid + 1, end);
54. merge(array, begin, mid, end);
55. }

 Рассмотрим тестовый пример для 10 и 100 элементов:





Определим зависимость времени от количества входных данных эмпирически. Примерное количество операций, выполняемых С++ за миллисекунду – 1 000 000. В действительности эта цифра зависит от характеристик ВМ, типа операций, оптимизации самого компилятора. Тогда рассчитаем примерное время сортировки как:

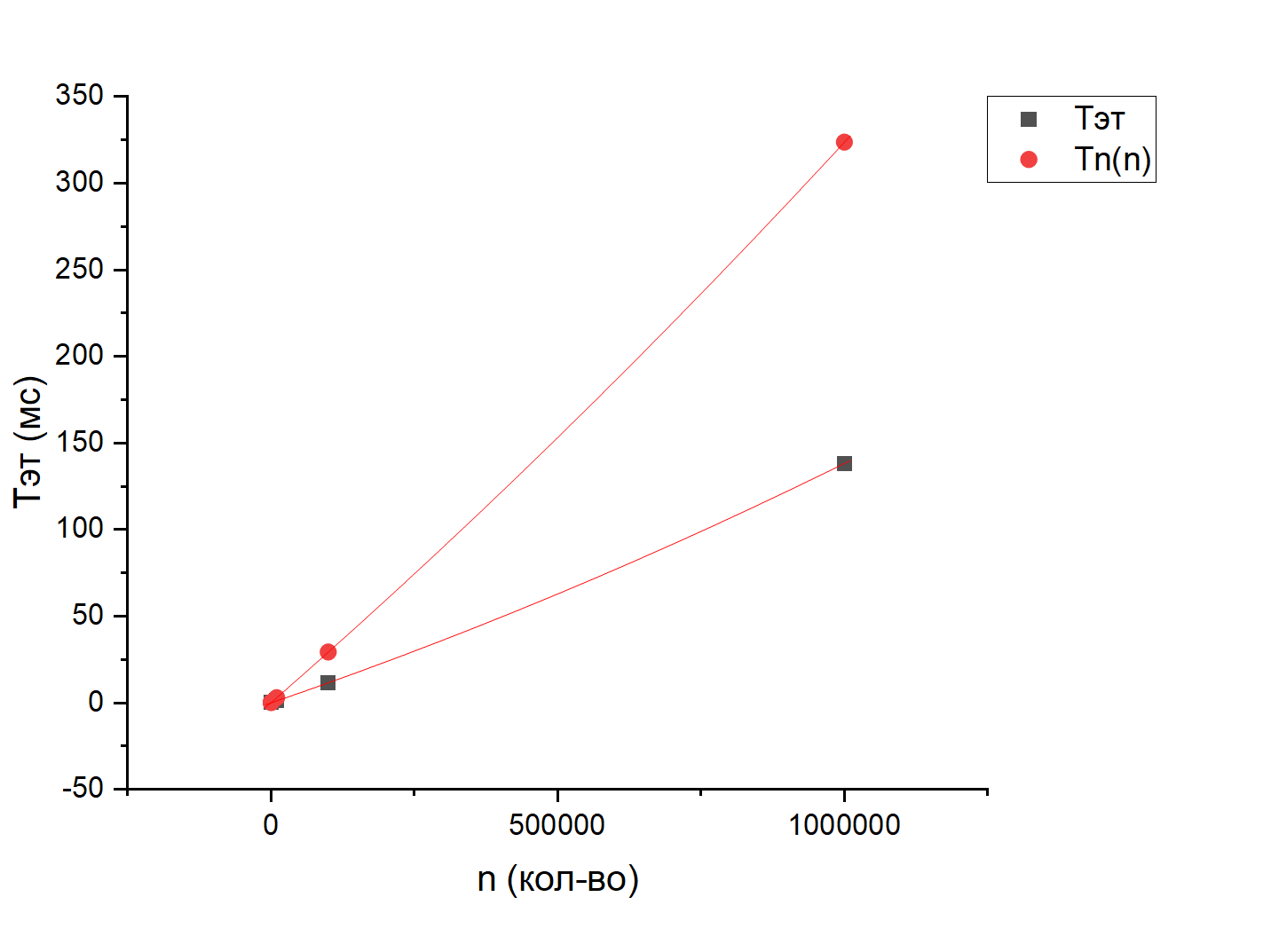
Рассмотрим эмпирически количество выполняемых операций. В соответствии с кодом задачи, получим:

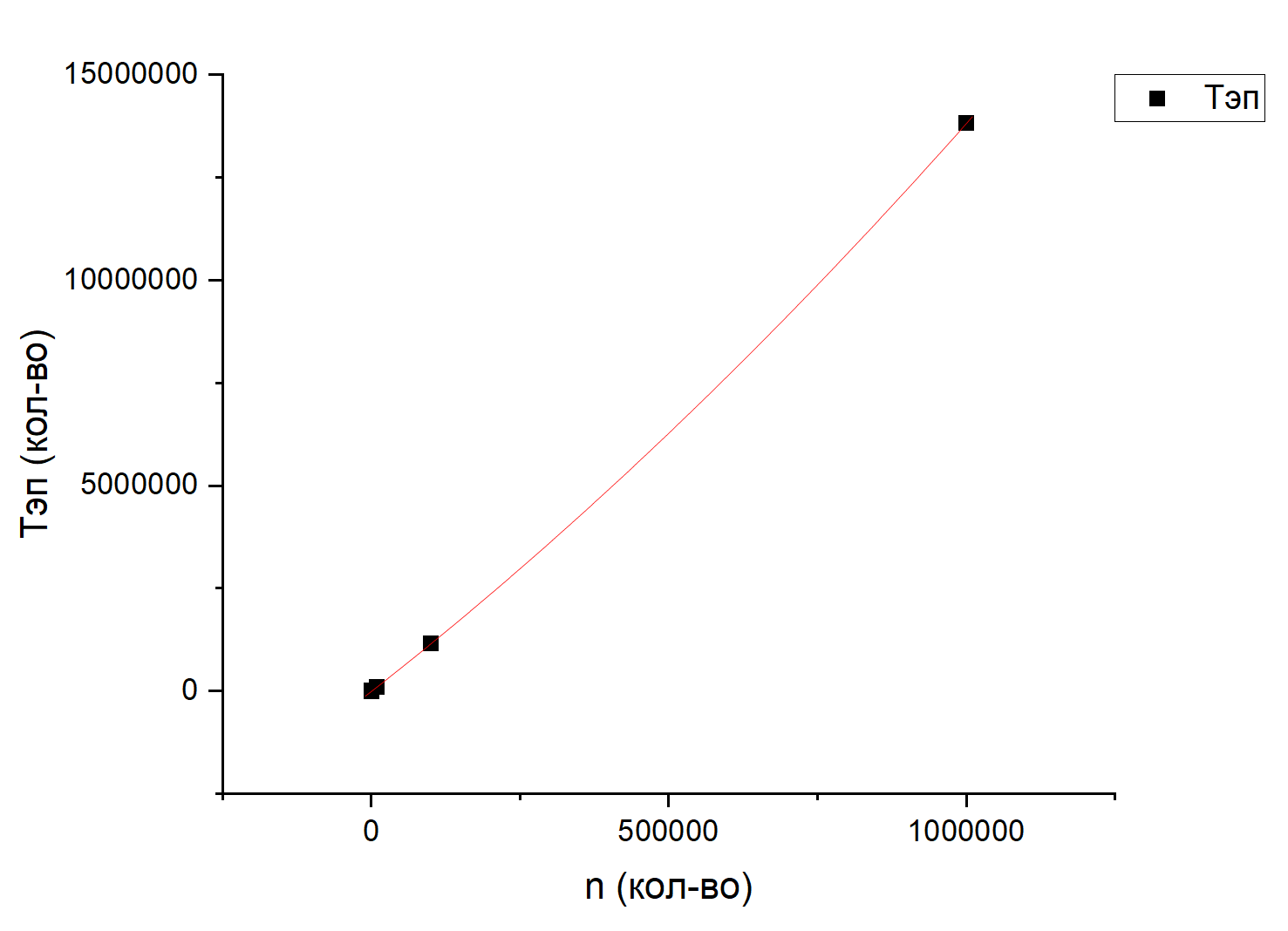
Таким образом, асимптотическая сложность бинарной сортировки вставками:

Анализ емкостной сложности:

***Таблица 4. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.0642 | 0,004605 | 461 |
| 1000 | 0.2882 | 0,069078 | 6908 |
| 10000 | 2.8971 | 0,921034 | 92104 |
| 100000 | 29.2487 | 11,51293 | 1151293 |
| 1000000 | 323.621 | 138,1551 | 13815511 |



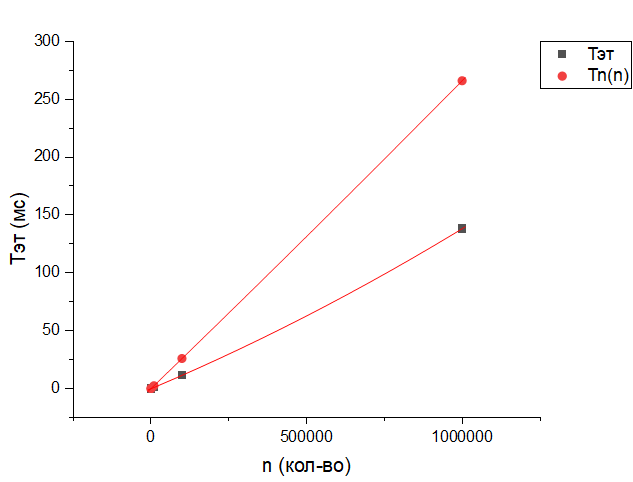
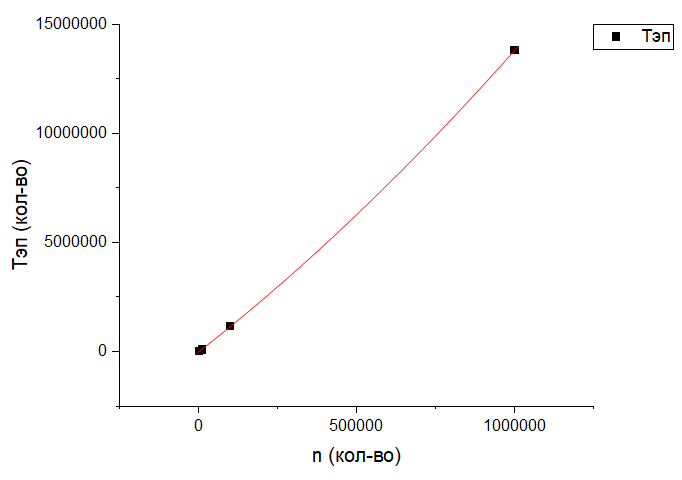


Отчетливо виден рост медленнее квадратичной зависимости.

Рассмотрим наилучший случай (исходный массив отсортирован по возрастанию):

***Таблица 5. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время, ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.0483 | 0,004605 | 461 |
| 1000 | 0.243 | 0,069078 | 6908 |
| 10000 | 2.4951 | 0,921034 | 92104 |
| 100000 | 26.0095 | 11,51293 | 1151293 |
| 1000000 | 265.986 | 138,1551 | 13815511 |

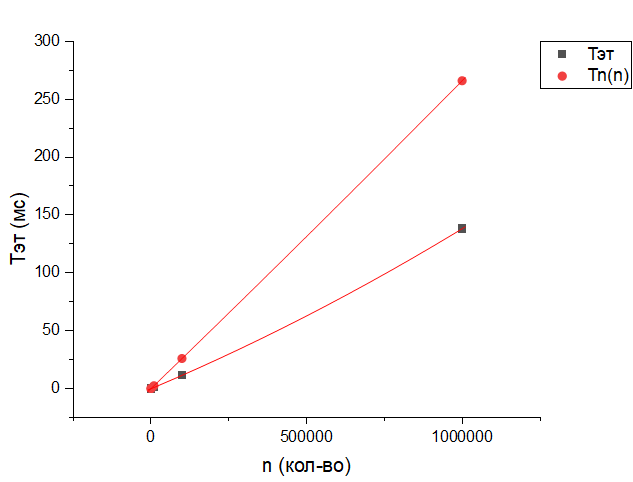
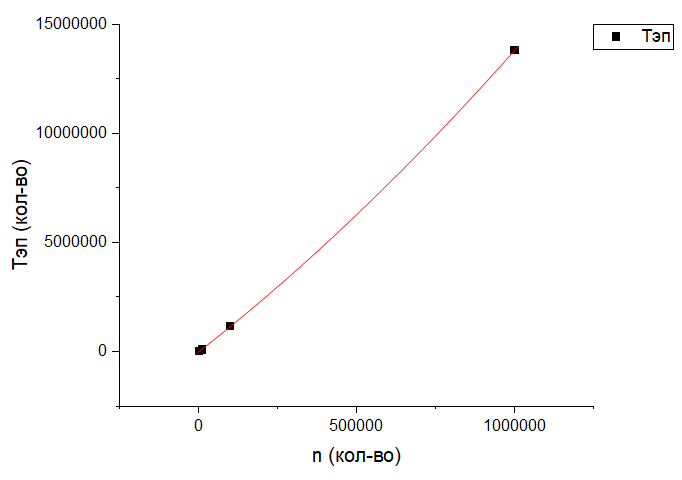


Из двух построенных графиков можно увидеть, что экспериментальное время так же соответствует функции O(n ln(n)). Имеются расхождения с теоретически вычисленными значениями, они вызваны особенностями выполнения программного кода на данной ВМ.

**Рассмотрим наихудший случай для сортировки бинарным выбором.**

***Таблица 5. Сводная таблица результатов***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время, ms** | **Тэт=f(C+M)-функция, ms** | **Т**эп**=Cф+Mф- количество** |
| 100 | 0.0384 | 0,004605 | 461 |
| 1000 | 0.2702 | 0,069078 | 6908 |
| 10000 | 2.6514 | 0,921034 | 92104 |
| 100000 | 26.3227 | 11,51293 | 1151293 |
| 1000000 | 265.291 | 138,1551 | 13815511 |



На построенных графиках отчетливо видна квадратичная зависимость. Расхождения связаны с особенностями выполнения программного кода на данной ВМ.

**Выводы**

Определим более эффективный алгоритм из всех предложенных:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название алгоритма | Асимптотическая сложность алгоритма | | | |
| Наихудший случай | Наилучший случай | Средний случай | Емкостная сложность |
| Простого выбора | O(n^2) | O(n^2) | O(n^2) | O(1) |
| Сортировка бинарными включениями | O(n^2) | O(nlogn) | O(n^2) | O(1) |
| Сортировка слиянием | O(nlogn) | O(nlogn) | O(nlogn) | O(n) |

Таким образом, на большом массиве данных лучше всего себя показывает сортировка слиянием, однако для чрезвычайно больших объемов данных следует использовать внешние сортировки (k-way merge, polyphase sort). Если брать в учет труд программиста, то для большинства мелких задач будут удобны простые алгоритмы сортировок, а в погоне за производительностью следует выбрать оптимизированные сортировки, временная сложность которых описывается O(nlogn) или еще оптимальнее.