Лабораторная работа № 1, по курсу дискретного анализа: Сортировки за линейное время

Выполнил студент группы М8О-209Б-23 МАИ *Борисов Денис Сергеевич.*

Условие

Требуется разработать программу, осуществляющую ввод пар «ключ-значение», их упорядочивание по возрастанию ключа указанным алгоритмом сортировки за линейное время и вывод отсортированной последовательности.

Вариант задания определяется типом ключа (и соответствующим ему методом сортировки) и типом значения:

Карманная сортировка.

Тип ключа: числа от 0 до $2^{64} - 1$.

Тип значения: строки фиксированной длины 64 символа, во входных данных могут встретиться строки меньшей длины, при этом строка дополняется до 64-х нулевыми символами, которые не выводятся на экран.

Вариант: 8-1

Метод решения

Рассмотрим алгоритм карманной сортировки. Идея, лежащая в основе карманной сортировки, заключается в том, чтобы разбить интервал $[0, 2^{64} - 1]$ на n одинаковых интервалов, или *карманов* (buckets), а затем распределить по этим карманам n входных величин. Поскольку входные числа имеют огромный диапазон, необходимо подобрать такое количество карманов, чтобы оно было оптимальным: не накладывало дополниельной сложности на алгоритм и занимающее оптимальное количество памяти, поэтому опытным путем было принято рещение выбрать 512 карманов для такого диапазона. Такое количество карманов отлично подходило для равномерного рампределения n элементов. Чтобы получить выходную последовательность, нужно просто выполнить сортировку чисел в каждом кармане, а затем последовательно перечислить элементы каждого кармана. Сортировку чисел внутри кармана будем осуществлять с помощью алгоритма вставкой.

Описание программы

```cpp
#include <iostream>
#include <vector>
#include <cstring>
#include <cmath>

// Структура входных данных
struct MapStruct {
    unsigned long long key;
    char value[65];
};

// Сортировка элементов кармана (алгоритм сортировки вставкой)
void insertionSort(std::vector<MapStruct> &bucket) {
    const size_t n = bucket.size();
    for (size_t i = 1; i < n; ++i) {
        MapStruct key = bucket[i];
        int j = i - 1;

        while (j >= 0 && bucket[j].key > key.key) {
            bucket[j + 1] = bucket[j];
            --j;
        }

        bucket[j + 1] = key;
    }
}

// Встраиваемая функция для определения индекса кармана
inline int getBucketIndex(unsigned long long key, int bucketCount) {
    return static_cast<int>((static_cast<double>(key) * (bucketCount - 1)) /
    ↪  0xffffffffffffffffull);
}

//Алгоритм карманной сортировки
void bucketSort(std::vector<MapStruct> &pairs) {
    if (pairs.size() <= 1) return;

    const int bucketCount = 512;
    std::vector<std::vector<MapStruct>> buckets(bucketCount);

    const size_t avgBucketSize = pairs.size() / bucketCount + 1;
    for (auto &bucket: buckets) {
```

```
42          bucket.reserve(avgBucketSize);
43      }
44
45      for (auto &pair: pairs) {
46          int bucketIndex = getBucketIndex(pair.key, bucketCount);
47          buckets[bucketIndex].push_back(std::move(pair));
48      }
49
50      pairs.clear();
51      pairs.reserve(pairs.capacity());
52
53      for (auto &bucket: buckets) {
54          if (bucket.size() > 1) {
55              insertionSort(bucket);
56          }
57
58          pairs.insert(pairs.end(),
59                       std::make_move_iterator(bucket.begin()),
60                       std::make_move_iterator(bucket.end()));
61          std::vector<MapStruct>().swap(bucket);
62      }
63  }
64
65  int main() {
66      std::ios_base::sync_with_stdio(false);
67      std::cin.tie(nullptr);
68
69      std::vector<MapStruct> pairs;
70
71      unsigned long long key;
72      char valueBuffer[65];
73
74      while (std::cin >> key) {
75          char c;
76          while ((c = std::cin.get()) == ' ' || c == '\t');
77
78          std::cin.unget();
79          std::cin.getline(valueBuffer, 65);
80
81          pairs.push_back({key, {0}});
82          std::strncpy(pairs.back().value, valueBuffer, 64);
83          pairs.back().value[64] = '\0';
84      }
85
86      bucketSort(pairs);
```

```
87
88      for (const auto &pair: pairs) {
89          std::cout << pair.key << '\t' << pair.value << '\n';
90      }
91
92      return 0;
93  }
94
```

Дневник отладки

В ходе выполнения работы не составило сложности разобраться с самим алгоритмом сортировки, однако проблемы начались на этапе оптимизации. Изначально мое решение использовало болше памяти, чем требовалось в условии. Мне пришлось применить move семантику, вручную выделять память для ограничения ее использования и переиспользовать уже имеющиеся ссылки, вместо того чтобы создавать новые. Кроме того, после решения проблем связанных с памятью, мне пришлось решать проблему связанную со временем выполнения программы. Детально изучив алгоритм, я понял что чем больше карманов я создам, тем быстрее они будут отсортированы, из за меньше числа элементов в них. Опытным путем я заметил что увеличение карманов с 256 до 512 дало значительный прирост в происводительности и незначительное увеличение накладных расходов памяти.

Тест производительности

Оценка сложности алгоритма

Выполним оценку сложности алгоритма сортировки и докажем ее линейную сложность. Чтобы оценить время работы алгоритма, заметим, что в наихудшем случае для распределения всех элементов по карманам, требуется время $O(n)$. Остается просуммировать полное время, которое потребуется для n вызовов алгоритма сортировки методом вставок. Таким образом суммарная сложность имеет вид:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right).$$

Однако такая сложность все еще не похожа на линейную. Воспользуемся формулой математического ожидания для обеих частей уравнения. В результате получим следующий вид:

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)\right] = \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}\left[O\left(n_i^2\right)\right] = \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}\left[n_i^2\right]\right)$$

Мы утверждаем, что для всех $i = 0, 1, \ldots, n-1$

$$\mathrm{E}\left[n_i^2\right] = 2 - 1/n.$$

Таким образом, каждому i-му карману соответствует одна и та же величина $\mathrm{E}\left[n_i^2\right]$, поскольку все элементы входного массива могут попасть в любой карман с равной вероятностью. Значит итоговая сложность алгоритма преобретает вид:

$$\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n).$$

Оценка производительности

Для практического подтверждения теоретических выводов о сложности алгоритма был проведен ряд экспериментов с различными размерами входных данных. Результаты сравнения реализации карманной сортировки и стандартной сортировкой `std::sort` представлены в таблице:

Таблица 1: Время работы алгоритмов сортировки (мс)

Размер данных	**Bucket Sort (мс)**	**std::sort (мс)**
1000	0.392	0.167
5000	1.000	1.005
10000	1.874	2.141
50000	15.281	12.445
100000	46.021	25.876
500000	885.351	148.282
1000000	3410.600	306.171

На основании полученных данных был построен график зависимости времени работы от размера входных данных. Для лучшей визуализации асимптотики алгоритмов представим те же данные в логарифмической шкале:

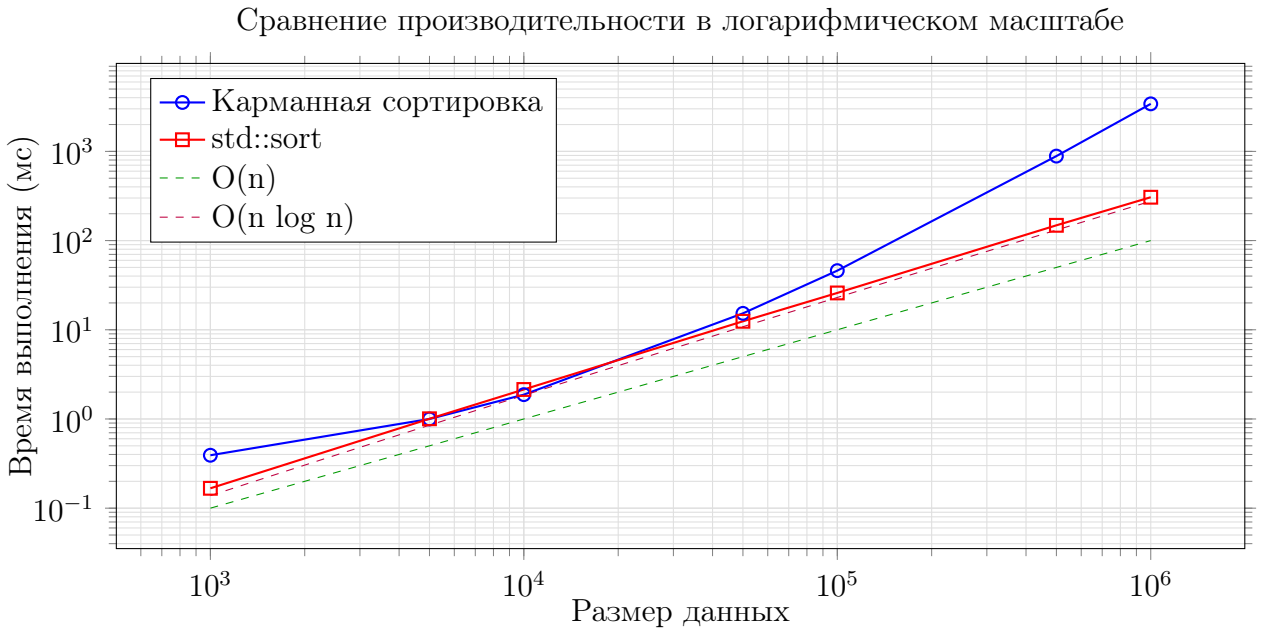

Рис. 1: Анализ асимптотической сложности алгоритмов

Анализ результатов

Анализ полученных результатов показывает, что:

1. На малых размерах данных (до 10000 элементов) реализованная карманная сортировка и `std::sort` демонстрируют схожую производительность, причем на некоторых наборах данных (5000-10000 элементов) карманная сортировка даже немного опережает стандартную.

2. На средних объемах данных (10000-100000 элементов) наблюдается постепенное увеличение разрыва в производительности в пользу `std::sort`.

3. При больших объемах данных (500000-1000000 элементов) разница становится существенной, что может быть связано с:

   - Большими накладными расходами на создание и инициализацию карманов
   - Неравномерным распределением элементов по карманам
   - Более эффективной реализацией `std::sort`

4. График в логарифмических координатах показывает, что асимптотическая сложность реализованной карманной сортировки на практике не всегда соответствует теоретической $O(n)$, из за того, что необходимо обязательно соблюсти условие равномерного распределения

5. На графике отношения времен хорошо видно, что эффективность карманной сортировки относительно `std::sort` снижается по мере увеличения размера данных.

Можно заключить, что теоретическая линейная сложность карманной сортировки проявляется в полной мере только при равномерном распределении входных данных. В реальных условиях производительность алгоритма существенно зависит от распределения входных данных и особенностей их обработки, что объясняет наблюдаемое отклонение от теоретической оценки производительности.

В итоге математическое ожидание времени работы алгоритма карманной сортировки в целом линейно зависит от количества входных элементов.

Выводы

В результате выполнения лабораторной работы, я изучил алгоритм карманной сортировки и реализовал его для поставленой задачи, провел анализ этой сортировки и сравнил ее со стандартной встроенной реализации.

Анализ показал, что для линейной сложности алгоритма карманной сортировки строго необходимо выполнение условия равномерного распределения элементов по карманам. На практике сортировка кажется достаточно бесполезной, из за строго соблюдения условия, однако в реальной жизни данные не всегда смогут равномерно распределиться. Если же условие равномерного распределения не выполняется, то сложность сортировки будет стремиться к квадратичной, что делает ее менее эффективной.