


Нижегородский государственный университет им. Н.И. Лобачевского

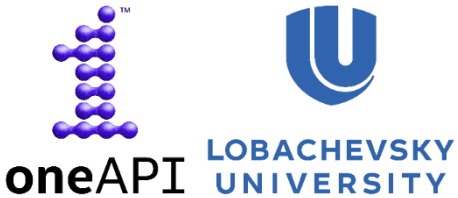
Центр компетенций oneAPI в ННГУ

Институт информационных технологий, математики и механики

Нижегородский государственный университет им. Н.И. Лобачевского
Центр компетенций oneAPI в ННГУ
Институт информационных технологий, математики и механики

Введение в анализ производительности и оптимизацию программ

Приемы оптимизации программ на примере алгоритмов сортировки массива



Панова Е.А., Мееров И.Б.

Содержание

Введение

- Оптимизации бывают *алгоритмические* и *программные*.
- *Алгоритмические* оптимизации заключаются в замене алгоритма на более совершенный.
- *Программные* подразумевают оптимизацию под конкретную архитектуру/класс архитектур за счет учета их особенностей (параллелизм, векторизация и т.п.).
- Основной вклад вносят алгоритмические оптимизации и выбор подходящих структур данных, поэтому сначала следует подобрать оптимальный алгоритм, и лишь затем его оптимизировать под конкретную архитектуру!
- Рассмотрим некоторые алгоритмические оптимизации на примере широко известной и полезной задачи – *сортировки* массива.
- **План:**
  - Повторить (или узнать ☺) теоретические аспекты, касающиеся алгоритмов сортировок.
  - Продемонстрировать общий подход к алгоритмической оптимизации на примере сортировки *вставками* и *быстрой* сортировки.

Постановка задачи

- Дано n элементов (ключей) из множества $a_i \in A, i = 1..n$ (целые или вещественные числа, строки, произвольные объекты, что угодно).
- На этих элементах определено некоторое *отношение порядка* (строгое “<” или нестрогое “≤”).
- Цель: найти такую перестановку индексов $i_1, \dots, i_n$, чтобы выполнялось следующее условие:

$$a_{i_1} \le a_{i_2} \le \cdots \le a_{i_n}$$

- Если данные содержат одинаковые ключи, то таких перестановок может быть несколько.
- Алгоритмы сортировок оцениваются по *времени*, которое требуется для сортировки n ключей, и дополнительным затратам по *памяти*. В теории сортировок оценивают не абсолютное время, а асимптотическое (O-большое).

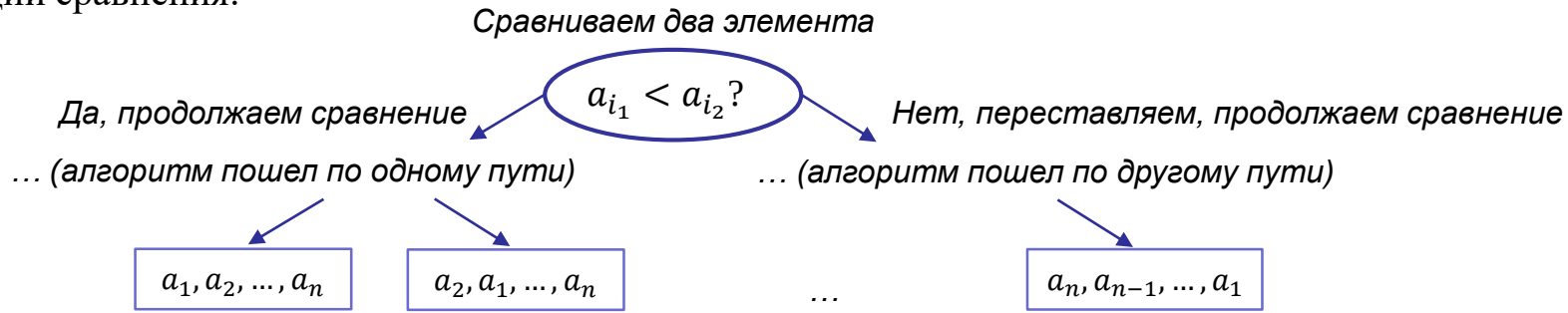
Немного теории

- Сортировки делятся на классы по следующим свойствам:
  - *Устойчивость* – порядок элементов с одинаковыми ключами не изменяется.
    - Имеет смысл, если ключи соотносятся с некоторыми данными (например, идентификатор объекта в базе данных и сам объект). Или для сортировки по нескольким признакам (упорядочив устойчивой сортировкой карты сначала по номиналу, потом по масти, получим, что в рамках каждой масти номинал карт упорядочен).
    - Сортировки пузырьком, вставками, слиянием – устойчивые. Сортировки выбором, быстрая, пирамидальная, Шелла – неустойчивые.
  - *Естественность* – сортировка работает быстрее на частично отсортированных данных.
    - Приведите примеры естественных сортировок.
  - *Вычислительная сложность* – квадратичные сортировки, сортировки за $O(n \log n)$ и т.д.
  - *Дополнительная память* – требуется или нет для хранения промежуточных результатов.
  - …

Немного теории

- Теорема: *минимальная трудоемкость худшего случая для любого алгоритма сортировки, основанного на сравнениях, составляет* $O(n \log n)$.

Доказательство: пусть на вход алгоритма сортировки приходит массив $a_1, a_2, \dots, a_n$.
Любой алгоритм сортировки можно представить в виде бинарного дерева, в вершинах которого операции сравнения:



В конце получаем отсортированную последовательность. Поскольку алгоритм работает на любых данных, то он должен уметь генерировать все перестановки. Всего $n!$ перестановок – листьев дерева. За сложность алгоритма отвечает высота дерева. Минимальная высота бинарного дерева с $n!$ листьями: $\log_2(n!) \approx O(n \log n)$ (формула Стирлинга).

Немного теории

- Существуют сортировки, работающие за *линейное* время.
- Пример 1: *сортировка подсчетом*. Входные данные – целые неотрицательные числа в известном диапазоне $0 \le a_i \le m \ll n$ (m можно найти линейным поиском). Сложность $O(n+m)$.
  - Создадим массив $b_j, j = 1..m$, заполним его нулями. Далее за линейное время выполняем $b_{a_i} \mathrel{+}= 1$, $i = 1..n$. Массив b хранит по индексу a_i число вхождений элемента a_i. В конце остается собрать отсортированный массив a на основе b.
- Пример 2: *поразрядная сортировка*. Входные данные – целые числа, записанные цифрами. Сложность $O(kn)$, где k – число разрядов.
  - Будем сортировать числа по разрядам. Сначала сортируем числа по старшему разряду, затем по следующему, и т.д. до младшего разряда. Сортировать по разрядам можно *устойчивым* вариантом сортировки подсчетом ($m = 9$ в десятичной системе счисления).
  - Можно ли сортировать от младшего разряда к старшему?
  - Можно ли так сортировать целые числа в двоичной системе счисления? Числа с плавающей запятой? Произвольные данные?
- Противоречит ли существование таких сортировок теореме с предыдущего слайда?

Немного теории

- Вывод: подбирать алгоритм сортировки нужно, основываясь на *условии задачи*.
- *Какая цель?* Стоит проанализировать типичные для задачи входные данные (частично отсортированы/мало элементов/известен тип элементов?), требования ко времени работы программы, ограничения по памяти, время на реализацию и т.д.
- Часто достаточно воспользоваться стандартной функцией сортировки (в C++ это std::sort).
- В некоторых случаях время работы сортировки является критичным (например, в системах реального времени). В этом случае стоит задуматься об оптимизации выбранного алгоритма сортировки.
- Далее демонстрируются некоторые, в основном, алгоритмические оптимизации на примере *сортировки вставками* и *быстрой сортировки*.

СОРТИРОВКА ВСТАВКАМИ

Сортировка вставками. Базовая версия

- Сортируем массив a размера n.
- На шаге $i = 1..n-1$:
  - элементы $0..i-1$ уже упорядочены;
  - берем элемент $v = a[i]$, меняем его местами с предыдущим до тех пор, пока подмассив $a[0..i]$ не станет упорядоченным.

```
void insertion_sort(float* a, int n) {
    for (int i = 1; i < n; i++) {
        float v = a[i];

        int j = i - 1;
        while (j >= 0 && a[j] > v) {
            std::swap(a[j], a[j + 1]);
            j--;
        }
    }
}
```

- Алгоритмическая сложность на отсортированном массиве $O(n)$.
- Алгоритмическая сложность в среднем и худшем случае $O(n^2)$.
- Дополнительные затраты по памяти $O(1)$.

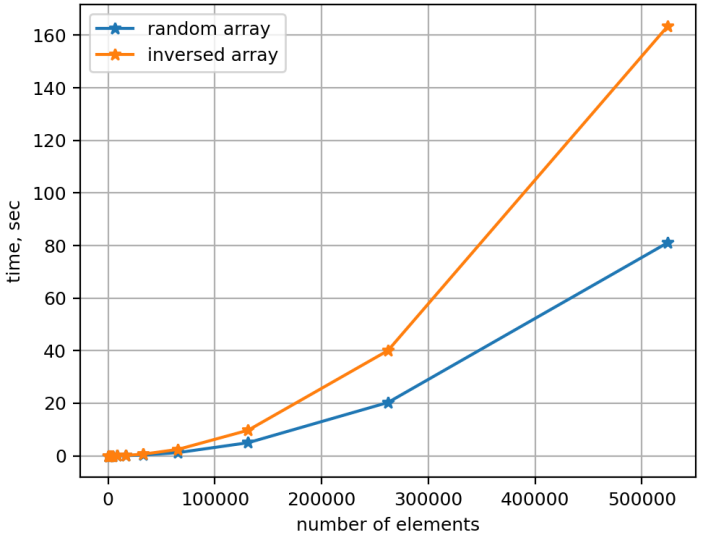
Тестовая инфраструктура

Процессор	Intel Core i7-11800H (Tiger Lake), 2,3 ГГц, Turbo 4,6 ГГц
Операционная система	Windows 10 version 21H2
Компилятор, профилировщик	Intel oneAPI 2023.2: Intel C++ Compiler Classic Intel Advisor

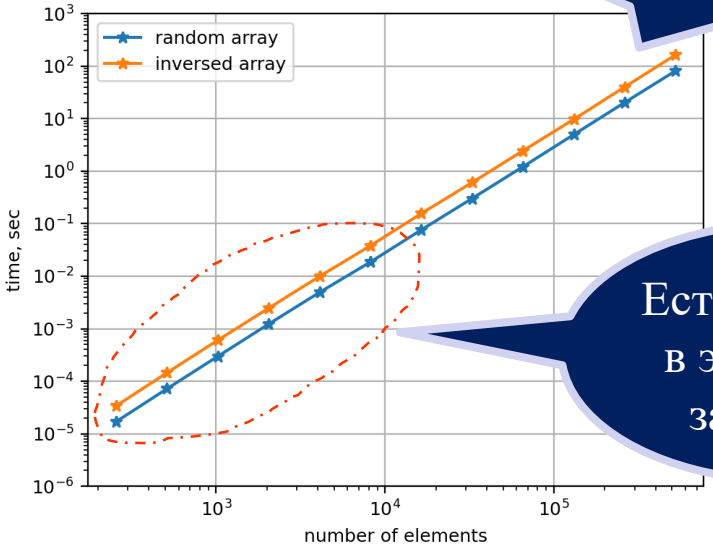
Сортировка вставками. Базовая версия

- Проведем эксперименты на
  - массивах, отсортированных в обратном порядке;
  - массивах, сгенерированных случайным образом.

Как показать, что сложность $O(n^2)$



Время от числа элементов



То же самое в логарифмическом масштабе

Есть ли смысл в этой части запусков?

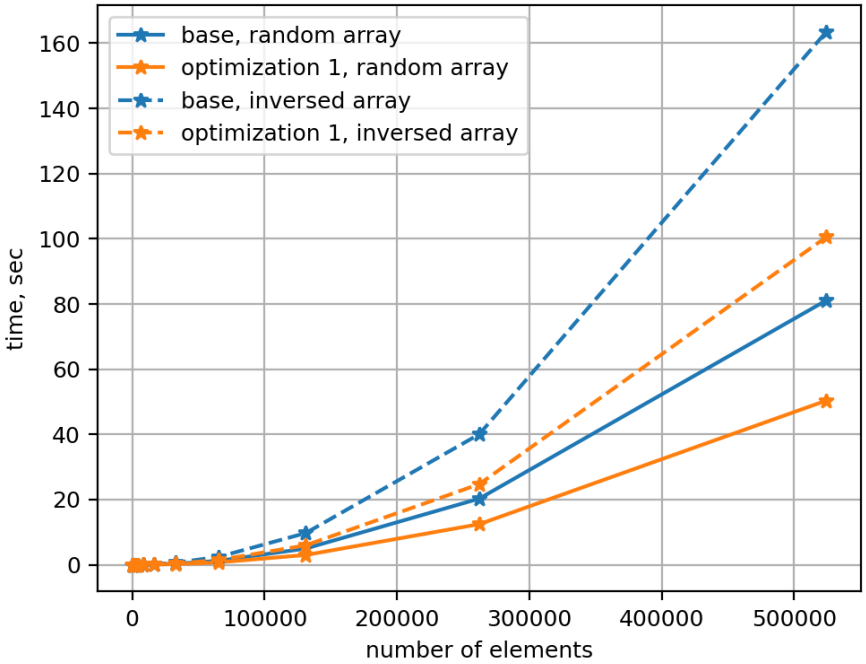
Сортировка вставками. Оптимизация 1

- Попробуем другой вариант алгоритма.
- Сначала найдем позицию опорного элемента, затем поместим его туда, сдвинув поочередно другие элементы вправо.
- Какая теперь алгоритмическая сложность?
- Какой алгоритм должен работать быстрее: этот или предыдущий?

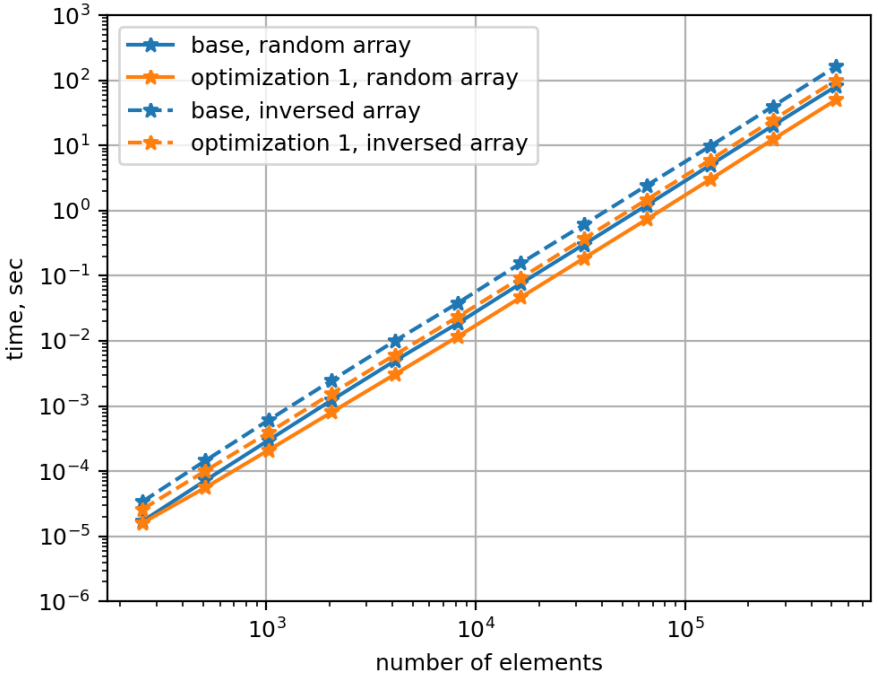
```
void insertion_sort(float* a, int n) {
    for (int i = 1; i < n; i++) {
        float v = a[i];

        // поиск места элемента в подмассиве 0..i-1
        int index = i - 1;
        while (index >= 0 && a[index] > v)
            index--;

        // сдвиг элементов index..i-1 на 1 позицию вправо
        for (int j = i - 1; j > index; j--)
            a[j + 1] = a[j];

        // запись элемента на освобожденную позицию
        a[index + 1] = v;
    }
}
```

Сортировка вставками. Оптимизация 1

- Время уменьшилось на ~40%. Почему?



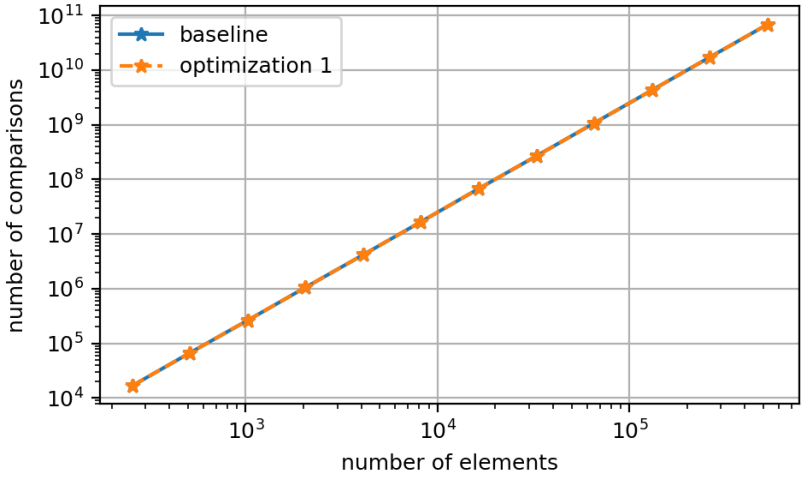
Время от числа элементов

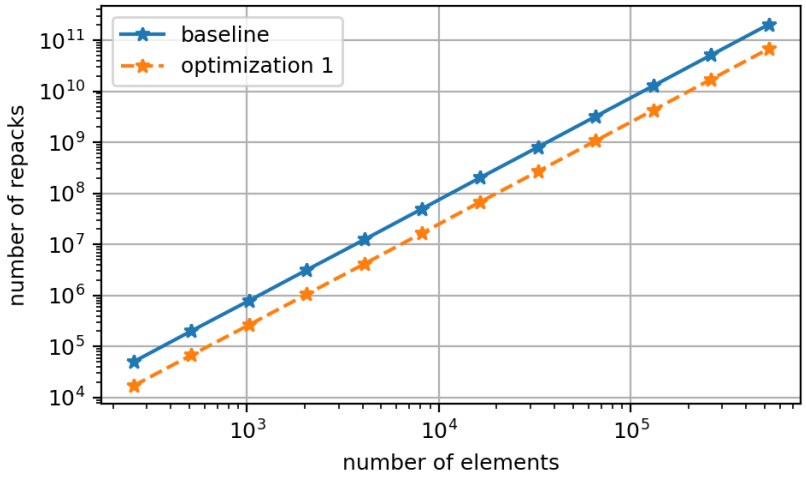


То же самое в логарифмическом масштабе

Сортировка вставками. Оптимизация 1

- Алгоритмическая сложность обоих алгоритмов $O(n^2)$.
- Это можно понять, подсчитав число операций *сравнения* элементов массива.
- Отличие второго алгоритма от первого заключается в количестве *перепаковок* (перемещений элемента в памяти). Для выполнения сдвига одного элемента требуется 1 перепаковка. Сколько перепаковок требуется для выполнения операции swap?





Сортировка вставками. Оптимизация 1

- Заметим, что в новой версии алгоритма все сравнения присутствуют только в первом внутреннем цикле (поиск позиции), а перепаковки только во втором (сдвиг).
- Соберем профиль программы на случайных данных. Замерим время программы в каждом из внутренних циклов и общее время работы.
- Согласно результатам профилировки, первый цикл (сравнения) занимает ~67% от общего времени, второй цикл (перепаковки) ~33%. Время остальных операций менее 0,2%.
- Таким образом, считаем, что все время программы состоит из сравнений и перепаковок: $T_{prog} = T_{comp} + T_{repack}$. Для новой версии алгоритма $T_{comp}^{opt_1} = 0{,}67\ T_{prog}^{opt_1}$, $T_{repack}^{opt_1} = 0{,}33\ T_{prog}^{opt_1}$.
- В первоначальной версии алгоритма было в 3 раза больше перепаковок: $T_{repack}^{base} = 0{,}99\ T_{prog}^{opt_1}$.
  А число сравнений такое же: $T_{comp}^{base} = T_{comp}^{opt_1} = 0{,}67\ T_{prog}^{opt_1}$.
- Следовательно, $T_{prog}^{base} = (0{,}99\ + 0{,}67) T_{prog}^{opt_1} = 1{,}66\ T_{prog}^{opt_1}$, т.е. вторая версия алгоритма должна составлять $1{,}66^{-1} \approx 60\%$ от первой. Это и наблюдается на практике.
- Стоит отметить, что в первой версии алгоритма перепаковки занимали ~60% времени, а сравнения ~40%. Т.е. во второй версии мы существенно уменьшили относительное время перепаковок.

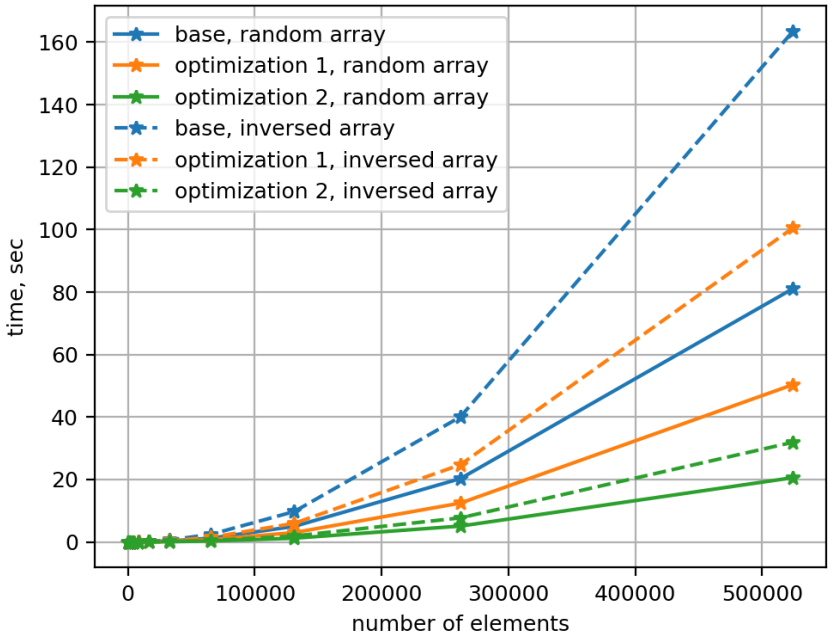
Сортировка вставками. Оптимизация 2

- А что если выполнять поиск позиции не справа налево (уменьшать индекс), а слева направо (увеличивать индекс)?

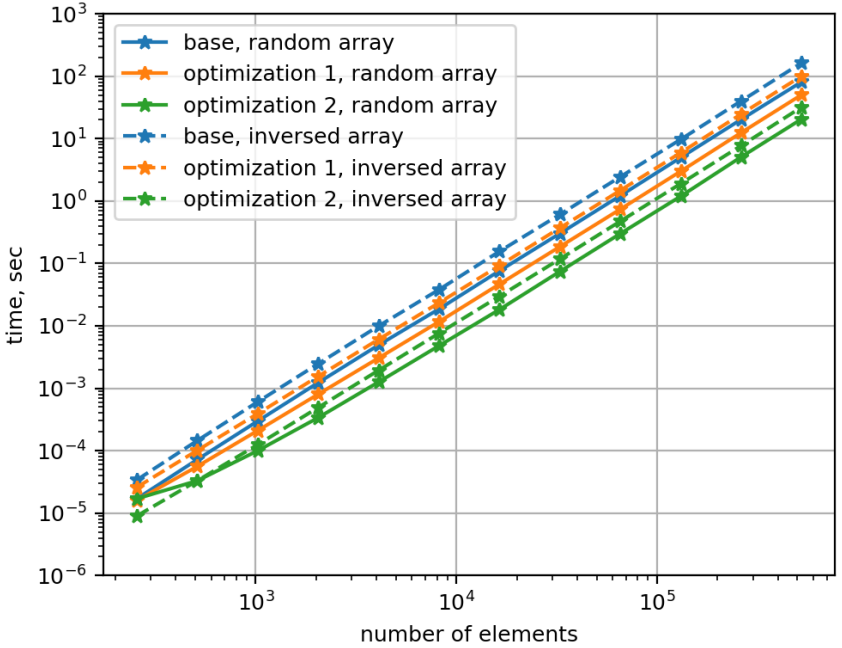
```
void insertion_sort(float* a, int n) {
    for (int i = 1; i < n; i++) {
        float v = a[i];

        // поиск места элемента в подмассиве 0..i-1
        int index = 0;
        while (index <= i - 1 && a[index] <= v)
            index++;   // поиск слева направо

        // сдвиг элементов index..i-1 на 1 позицию вправо
        for (int j = i - 1; j > index; j--)
            a[j + 1] = a[j];

        // запись элемента на освобожденную позицию
        a[index + 1] = v;
    }
}
```

Сортировка вставками. Оптимизация 2

- Время уменьшилось на ~60% относительно предыдущей версии. Почему?



Время от числа элементов



То же самое в логарифмическом масштабе

Сортировка вставками. Оптимизация 2

- Компилятор теперь распознал линейный поиск и оптимизировал его (задействовал векторные вычисления).
- Понять это помог профилировщик.

Предыдущая версия: линейный поиск не распознан

Function Call Sites and Loops	Type	Why No Vectorization?	Vectorized Loop	
			Vect...	Gain...
[loop in sort at insert_v1.cpp:11]	Scalar	loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria		

Новая версия: цикл оптимизирован

Function Call Sites and Loops	Type	Why No Vectorization?	Vectorized Loops			
			Vector ...	Efficiency	Gain...	VL ...
[loop in sort at insert_v2.cpp:13]	Scalar	vectorization possible but seems inefficient. Use vector always dir ...				
[loop in sort at insert_v2.cpp:11]	Vectorized (Bod...	1 loop with multiple exits cannot be vectorized unless it meets ...	AVX512	~53%	4,23x	8
[loop in sort at insert_v2.cpp:11]	Vectorized (Body)		AVX512			8
[loop in sort at insert_v2.cpp:11]	Peeled	loop with multiple exits cannot be vectorized unless it meets searc ...				
[loop in sort at insert_v2.cpp:11]	Remainder	loop with multiple exits cannot be vectorized unless it meets searc ...				

Сортировка вставками. Оптимизация 2

- Как подсчитать число сравнений и перепаковок для векторного кода?
- Можно попробовать оценить следующим образом.
- Длина векторного регистра 8 элементов float (предыдущий слайд), выполняется векторное сравнение => сравнений приблизительно в 8 раз меньше.
- Стоит отметить, что сравнение двух векторов реализуется в ассемблерном коде через побитовые операции (маски).
- Число перепаковок не изменилось, загрузками данных в векторный регистр пренебрегаем.
- Используя предыдущие результаты, получаем, что $T_{comp}^{opt_2} = \frac{1}{8} \cdot 0{,}67\, T_{prog}^{opt_1} \approx 0{,}08\, T_{prog}^{opt_1}$, $T_{repack}^{opt_2} = T_{repack}^{opt_1} = 0{,}33\, T_{prog}^{opt_1}$.
- Таким образом, $T_{prog}^{opt_2} = 0{,}41\, T_{prog}^{opt_1}$, т.е. выигрыш составляет ~60%.
- Это подтверждается экспериментами и результатами профилировки.

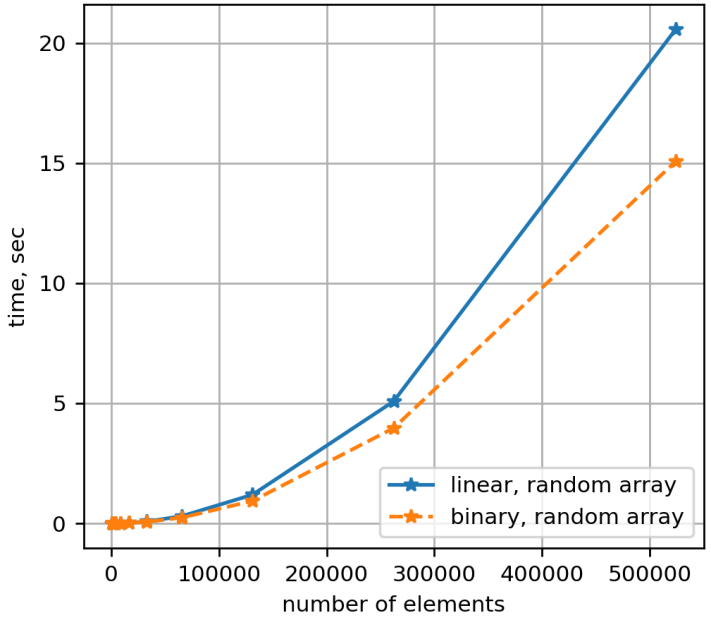
Сортировка вставками. Оптимизация 3

- Давайте заменим линейный поиск бинарным.
- Сложность сортировки не изменится, но время работы может уменьшиться.
- Воспользуемся стандартной реализацией бинарного поиска во избежание ошибок.

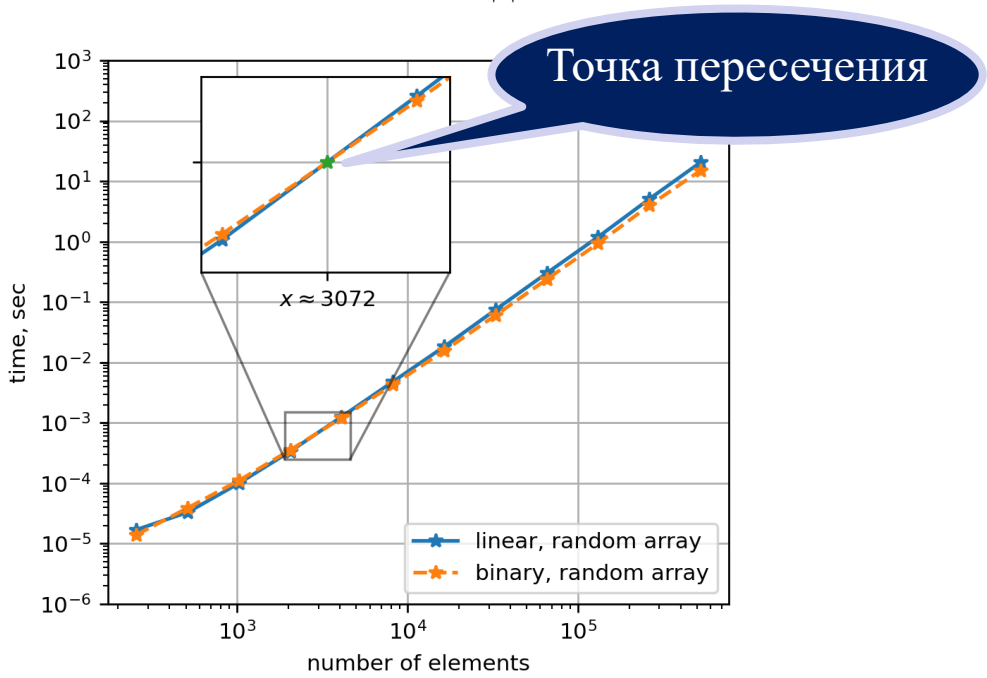
```
void insertion_sort(float* a, int n) {
    for (int i = 1; i < n; i++) {
        float v = a[i];

        // бинарный поиск
        int index = std::upper_bound(a, a + i, v) - a;

        // сдвиг элементов index..i-1 на 1 позицию вправо
        for (int j = i - 1; j > index; j--)
            a[j + 1] = a[j];

        // запись элемента на освобожденную позицию
        a[index + 1] = v;
    }
}
```

Сортировка вставками. Оптимизация 3

- Время уменьшилось на 20-30% на случайных массивах большой длины.



Время от числа элементов



То же самое в логарифмическом масштабе

Сортировка вставками. Оптимизация 4

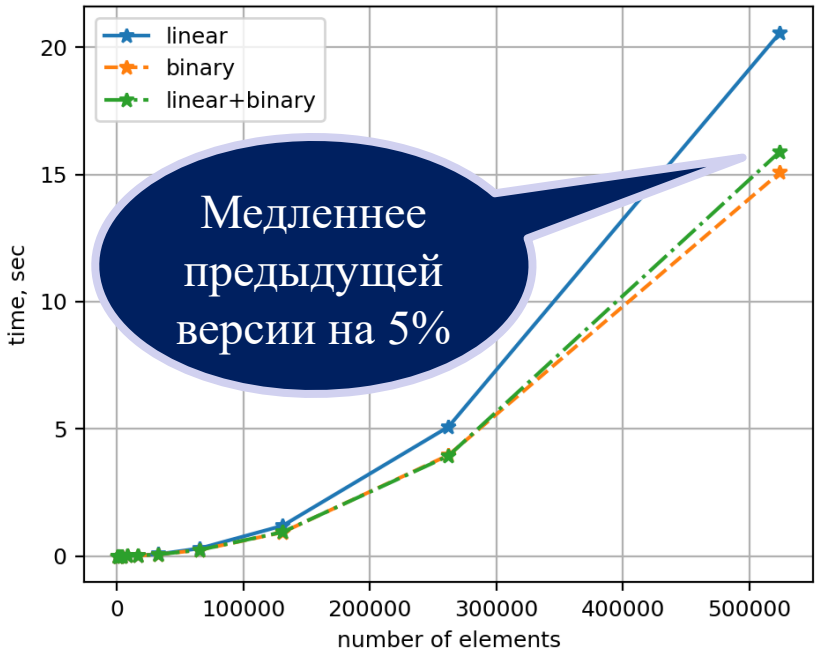
- Будем выполнять линейный поиск на небольших массивах и бинарный поиск на больших массивах.

Точка пересечения графиков с предыдущего слайда

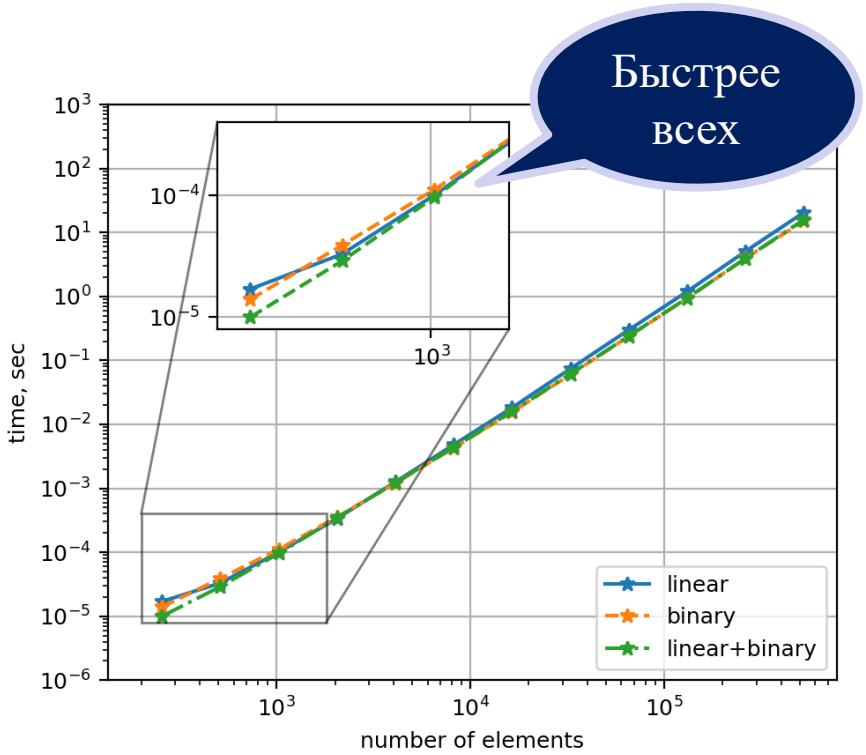
```
void insertion_sort(float* a, int n) {
    for (int i = 1; i < n; i++) {
        float v = a[i];

        // поиск
        const int threshold = 3072;
        int index = 0;
        if (n >= threshold)
            index = std::upper_bound(a, a + i, v) - a;
        else while (index <= i - 1 && a[index] < v)
            index++;

        // сдвиг элементов index..i-1 на 1 позицию вправо
        for (int j = i - 1; j > index; j--)
            a[j + 1] = a[j];

        // запись элемента на освобожденную позицию
        a[index + 1] = v;
    }
}
```

Сортировка вставками. Оптимизация 4

- Сработала ли оптимизация?



Время от числа элементов



То же самое в логарифмическом масштабе

Сортировка вставками. Выводы. Вопросы для обсуждения

- Удалось ускорить работу сортировки вставками на 80%.
- Есть ли смысл в оптимизации 4 (комбинация линейного и бинарного поиска)?
- Есть ли смысл в оптимизации 3 (замена линейного поиска на бинарный)?
- Насколько часто на практике требуется оптимизировать сортировку вставками?
- Нужно ли проверять корректность работы программы после каждой новой оптимизации? Как это сделать применительно к сортировкам?

БЫСТРАЯ СОРТИРОВКА

Быстрая сортировка. Алгоритм. Базовая версия

- Классическое описание алгоритма быстрой сортировки (сортировки Хоара):

1. Выбирается опорный элемент среди элементов массива.
2. Элементы, меньшие опорного, помещаются в левую часть массива; большие – в правую (процедура partition).
3. Сортировка вызывается рекурсивно для левой и правой половины массива.

```
void quick_sort(float* a, int left, int right)
{
    // выход из рекурсии, если массив размера 1
    if (right <= left) return;

    // делим массив на 2 части
    int i = partition(a, left, right);

    // сортируем левый подмассив
    quick_sort(a, left, i-1);
    // сортируем правый подмассив
    quick_sort(a, i+1, right);
}
```

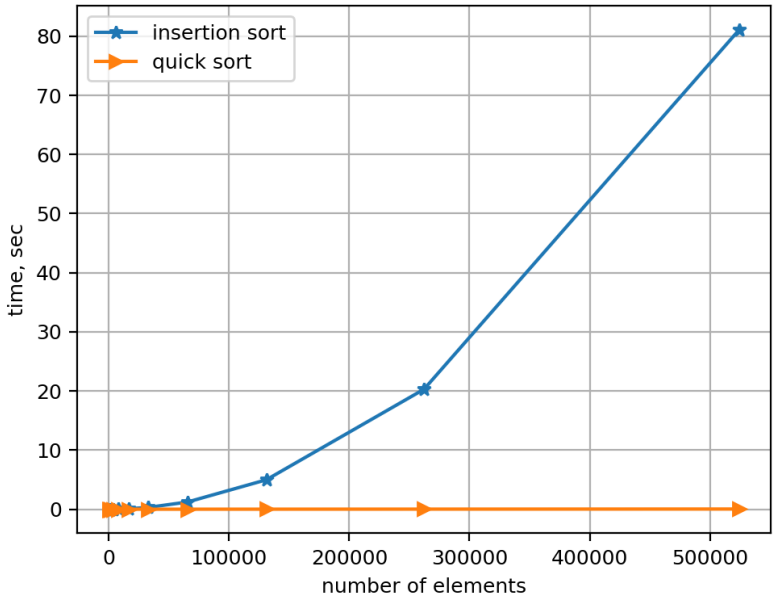
Быстрая сортировка. Алгоритм. Процедура partition

- Какова алгоритмическая сложность процедуры partition?
- Какова алгоритмическая сложность быстрой сортировки в лучшем случае? В худшем? В среднем?
- Когда достигается наихудший случай? Приведите примеры входных данных.
- Как будет вести себя сортировка на отсортированных данных?

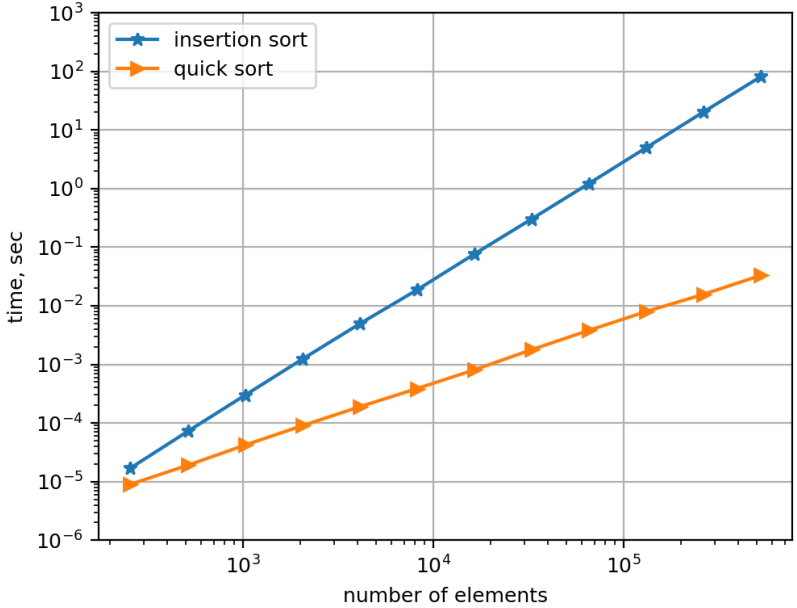
```
int partition(float* a, int left, int right) {
    const float pivot = a[right];  // опорный элемент
    int i = left - 1;              // левый индекс
    int j = right;                 // правый индекс

    while (true) {
        // двигаем левый индекс, пока не найдем a[i] >= pivot
        while (a[++i] < pivot);
        // двигаем правый индекс, пока не найдем a[j] <= pivot
        while (pivot < a[--j]) if (j == left) break;
        if (i >= j) break;
        std::swap(a[i], a[j]);  // меняем элементы местами
    }

    // ставим опорный элемент на место
    std::swap(a[i], a[right]);
    return i;
}
```

Быстрая сортировка. Базовая версия

- Время на случайном массиве значительно уменьшилось относительно сортировки вставками.
- Функция $n \log n$ близка к n (логарифм растет медленно).



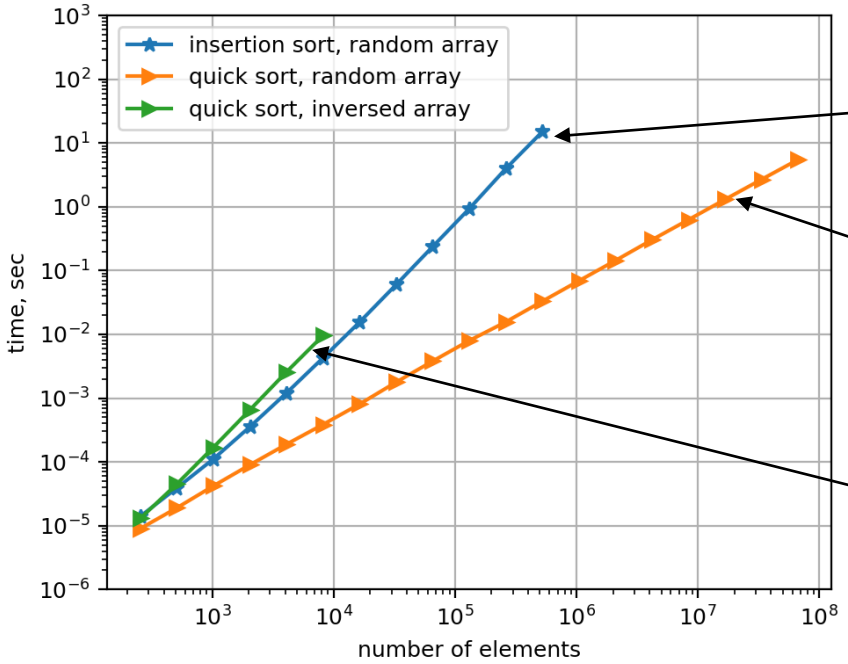
Время от числа элементов



То же самое в логарифмическом масштабе

Быстрая сортировка. Базовая версия

- На инвертированном массиве удалось посчитать лишь до размера 8192 элементов.
- На больших массивах программа падает. Почему?



Сортировка вставками слишком медленная, смогли посчитать только 2^{19} элементов

Быстрая сортировка на случайном массиве показывает отличные результаты, смогли посчитать за меньшее время 2^{26} элементов

На отсортированном массиве быстрая сортировка падает, если элементов больше 2^{13}

Быстрая сортировка. Базовая версия

- В наихудшем случае имеется $O(n)$ рекурсивных вызовов. Таким образом, базовая версия сортировки требует $O(n)$ дополнительной памяти.
- Для организации рекурсии используется ограниченный по размеру программный стек. На массивах больше некоторого порога происходит переполнение стека (stack overflow).
- Выбор другого опорного элемента решит проблему с упорядоченным массивом, т.к. в среднем требуется $O(\log n)$ рекурсивных вызовов.
- Однако вероятность возникновения наихудшего случая всегда ненулевая. Следовательно, нужно решать проблему с памятью.

Быстрая сортировка. Метод оконечной рекурсии

- Можно избавиться от второго рекурсивного вызова, заменив его циклом (*метод оконечной рекурсии*, *tail recursion*).

```
void quick_sort(float* a, int left, int right) {
    while (left < right) {
        int i = partition(a, left, right);
        quick_sort(a, left, i-1);
        left = i+1;
    }
}
```

- Это дает некоторый прирост производительности за счет уменьшения затрат на загрузку/выгрузку параметров при рекурсивном вызове.
- Уменьшились ли затраты по памяти в среднем случае?
- Какие затраты по памяти для отсортированного в прямом порядке массива? В обратном?
- Решает ли метод оконечной рекурсии проблему с памятью?
- Как исправить код так, чтобы сортировка стабильно работала на массивах больших размеров, отсортированных как в прямом, так и в обратном порядке?

Быстрая сортировка. Метод оконечной рекурсии

- Давайте делать рекурсивный вызов только для подмассива меньшего размера.
- Покажите, что теперь дополнительные затраты по памяти как в худшем, так и в среднем случае, равны $O(\log n)$.
- Будут ли они критичными для современных архитектур?

```
void quick_sort(float* a, int left, int right)
{
    while (left < right) {
        int i = partition(a, left, right);
        if (i - left < right - i) {
            quick_sort(a, left, i-1);
            left = i+1;
        }
        else {
            quick_sort(a, i+1, right);
            right = i-1;
        }
    }
}
```

Быстрая сортировка. Метод оконечной рекурсии

- А что если оставить классическую рекурсивную версию с двумя вызовами, но обрабатывать меньшую часть первой?
- Оценка по памяти в среднем случае остается прежней – $O(\log n)$.
- В наихудшем случае оценка по памяти $O(n)$.
- Однако данная программа успешно отработала на всевозможных данных, в том числе на больших отсортированных массивах.
- Как такое могло произойти?

```
void quick_sort(float* a, int left, int right)
{
    if (right <= left) return;
    int i = partition(a, left, right);
    if (i - left < right - i) {
        quick_sort(a, left, i-1);
        quick_sort(a, i+1, right);
    }
    else {
        quick_sort(a, i+1, right);
        quick_sort(a, left, i-1);
    }
}
```

Быстрая сортировка. Метод оконечной рекурсии

- Современные компиляторы часто могут сами превратить два рекурсивных вызова в один (для оптимизации по скорости).

```
__declspec(noinline) void sort(float* a, int left, int right) {
    if (right <= left) return;
    [loop in sort at quick_v0.cpp:26]
        Scalar loop. Not vectorized: loop control variable was not identified.
        No loop transformations applied
    int i = partition(a, left, right);
    sort(a, left, i-1);
    sort(a, i+1, right);
}
```

Профилировщик показывает цикл в базовой рекурсивной версии

- Таким образом, чтобы избежать проблем с памятью, в нашем случае достаточно обрабатывать меньший подмассив в первую очередь.

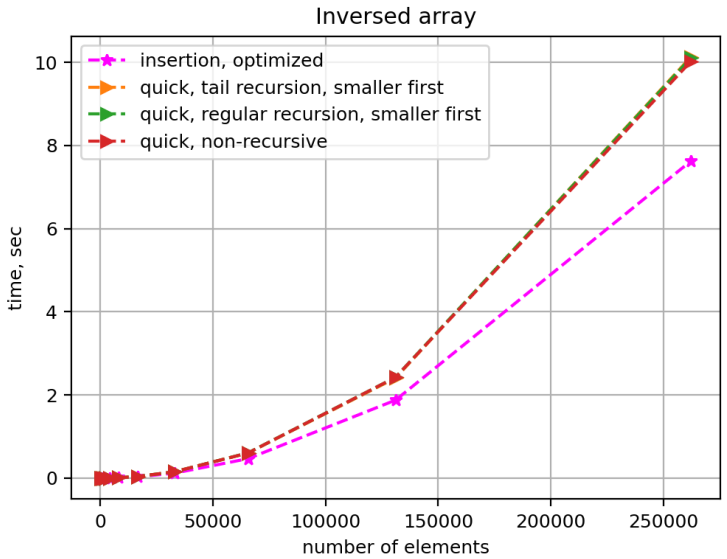
Быстрая сортировка. Нерекурсивная версия

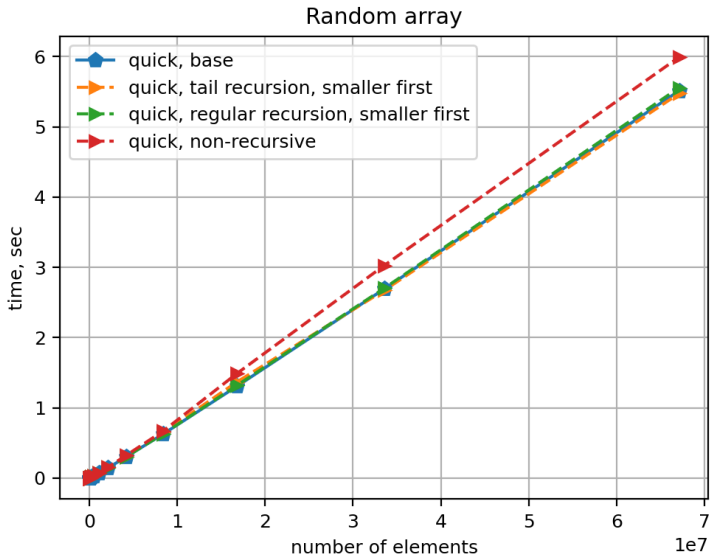
- Давайте попробуем вообще избавиться от рекурсии (но не от дополнительных затрат по памяти). Иногда это дает значительный прирост производительности.
- Любую рекурсию всегда можно переписать в виде цикла.
- Для этого воспользуемся структурой данных «стек», который будет имитировать программный стек.
- Сначала кладем в стек большую половину, чтобы обработать ее позже.

```
void quick_sort(float* a, int left, int right) {
    std::stack<std::pair<int, int>> stack;
    stack.push({ left, right });

    while (!stack.empty()) {
        auto borders = stack.top();
        stack.pop();
        left = borders.first;
        right = borders.second;
        if (left >= right) continue;

        int i = partition(a, left, right);
        if (i - left < right - i) {
            stack.push({ i+1, right });
            stack.push({ left, i-1 });
        }
        else {
            stack.push({ left, i-1 });
            stack.push({ i+1, right });
        }
    }
}
```

Быстрая сортировка. Исследование производительности

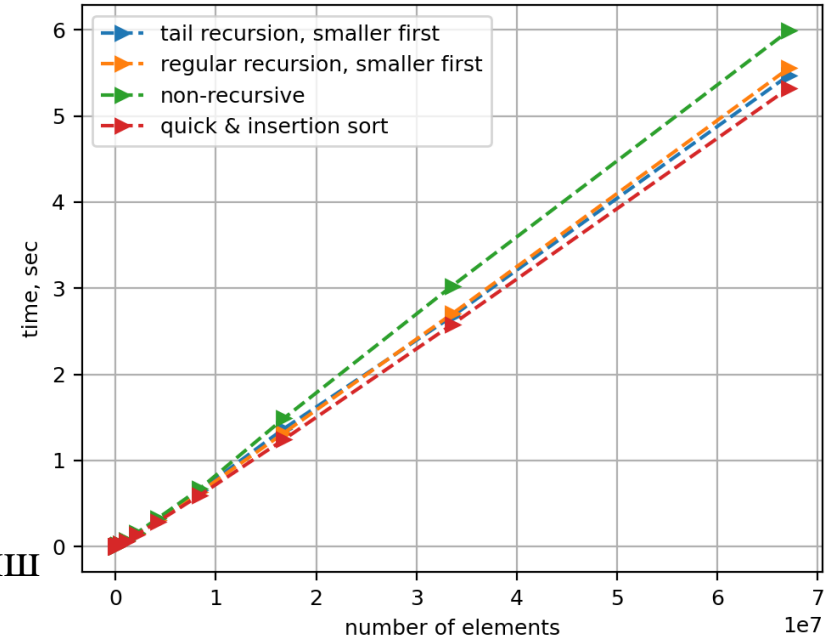
- На инвертированном массиве оптимизированная версия сортировки вставками работает быстрее быстрой сортировки. Но асимптотика в обоих случаях $O(n^2)$.
- Нерекурсивная версия быстрой сортировки работает медленнее всех. Сгенерированный компилятором код оказался сложнее.





Быстрая сортировка. Обработка небольших массивов

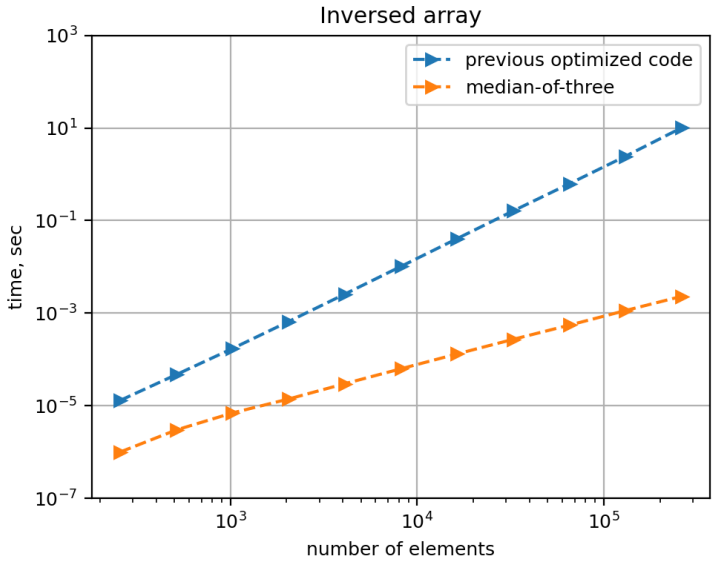
- Рекурсивно обрабатывать подмассивы небольших размеров не имеет смысла, зато присутствуют дополнительные затраты на рекурсивные вызовы.
- Давайте на небольших массивах подключать сортировку вставками.
- Эксперименты показали, что наилучший размер, с которого стоит подключать сортировку вставками – 16 элементов. На другом программно-аппаратном окружении оптимальным может быть другое значение.
- Это дало ускорение 4% на случайном массиве. На других архитектурах выигрыш может быть больше.



Быстрая сортировка. Выбор опорного элемента

- Нетривиальный выбор опорного элемента позволяет значительно уменьшить вероятность возникновения наихудшего случая, когда алгоритмическая сложность сортировки достигает $O(n^2)$.
- Пример – медиана из трех элементов (начало, конец, середина).
- Это полностью решает проблему с отсортированным или частично отсортированным массивом.

```
int mid = (left + right) / 2;
if (a[mid] < a[left])
    std::swap(a[left], a[mid]);
if (a[right] < a[left])
    std::swap(a[left], a[right]);
if (a[right] < a[mid])
    std::swap(a[right], a[mid]);
const float pivot = a[mid];
```



Быстрая сортировка. Выбор опорного элемента

- С практической точки зрения, лучший вариант входных данных – случайный массив (т.е. когда все перестановки элементов равновероятны).
- Если выбирать опорный элемент случайным образом (*рандомизированная сортировка*), то сортировка не будет зависеть от распределения входных данных.
- Можно выбирать медиану из трех случайных элементов.
- Чем сложнее метод выбора опорного элемента, тем больше константа у сортировки.
- Если алгоритм генерации случайных чисел и его параметры известны, то всегда можно подобрать такие входные данные для рандомизированной сортировки, которые дают наихудший случай времени.

Быстрая сортировка. Выводы. Вопросы для обсуждения

- Мы воспользовались более эффективным алгоритмом сортировки по сравнению с сортировкой вставками, чем значительно уменьшили время работы в среднем.
- Однако в наихудшем случае время работы быстрой сортировки больше на 25%, чем сортировки вставками. Тем не менее, при правильном выборе опорного элемента вероятность появления наихудшего случая крайне мала.
- Потребовалось решить проблему с памятью, возникающую при большом количестве рекурсивных вызовов.
- Известно, что почти всегда реализация метода std::sort в стандартной библиотеки C++ использует быструю сортировку. Однако в стандарте зафиксировано, что метод std::sort должен иметь сложность $O(n \log n)$ для любых входных данных. Как это можно обеспечить?
- Можно ли использовать линейную поразрядную сортировку для сортировки вещественных чисел? Произвольных данных? Если да, то почему не она используется в стандарте?

Заключение

- Первый шаг любой оптимизации программы… понять, а действительно ли нужно оптимизировать ☺
- Второй шаг оптимизации программы – определить, можно ли усовершенствовать алгоритм или заменить его на более эффективный с меньшей асимптотикой (по возможности). Только после этого имеет смысл приступать к другим оптимизациям.
- С другой стороны, может случиться так, что более «долгий» алгоритм работает быстрее на некоторых архитектурах (например, лучше параллелится).
- Не существует «лекарства от всех болезней». При выборе алгоритма обычно приходится балансировать между временем работы, дополнительной памятью, другими свойствами. Так, заменив сортировку вставками на эффективную быструю сортировку, мы увеличили затраты по памяти ($O(\log n)$ на рекурсивные вызовы).
- Тем не менее, нам без потерь удалось ускорить базовый алгоритм сортировки вставками на 80%, а также свести вероятность возникновения наихудшего случая в быстрой сортировке к минимуму. Таким образом, пытаться все же стоит ☺

Литература

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to algorithms. Fourth Edition. – MIT press, 2022.
2. Sedgewick R. Algorithms in c++, parts 1-4: fundamentals, data structure, sorting, searching. – Pearson Education, 1998.

Авторский коллектив

- Мееров Иосиф Борисович, к.т.н., доцент, зам. зав. каф. МОСТ
- Сысоев Александр Владимирович, к.т.н., доцент каф. МОСТ
- Линев Алексей Владимирович, зав. лаб. интернета вещей, каф. ПРИН
- Волокитин Валентин Дмитриевич, программист лаборатории СТиВВ, каф. МОСТ
- Козинов Евгений Александрович, к.т.н., преподаватель каф. МОСТ
- Панова Елена Анатольевна, инженер лаборатории СТиВВ, каф. МОСТ

Контакты

Нижегородский государственный университет

http://www.unn.ru

Центр компетенций oneAPI в ННГУ

http://hpc-education.unn.ru/ru/центр-компетенций-oneapi-в-ннгу

Институт информационных технологий, математики и механики

http://www.itmm.unn.ru