1.20 Сортировка кучей

"Простая" реализация:

```
template<typename T>
void heapify(std::vector<T>& v, size_t n, size_t i) {
    size_t largest = i;
    size_t l = 2 * i + 1;
    size_t r = 2 * i + 2;

    if (l < n && v[l] > v[largest])
        largest = l;
    if (r < n && v[r] > v[largest])
        largest = r;

    if (largest != i) {
        std::swap(v[i], v[largest]);
        heapify(v, n, largest);
    }
}

template<typename T>
void simple_heapsort(std::vector<T>& v) {
    size_t n = v.size();
    for (int i = v.size() / 2 - 1; i >= 0; --i)
        heapify(v, n, i);

    for (int i = n - 1; i >= 0; --i) {
        std::swap(v[0], v[i]);
        heapify(v, i, 0);
    }
```

Функция **simple_heapsort** вызывает **heapify** O(n) раз с минимальным индексом *i = 0*. Функция **heapify** каждый раз рекурсивно вызывается с увеличенным в около 2 раз индексом *i*, который не может превосходить *n*. Значит, асимптотика **heapify** ~ O(log(n)), а итоговая асимптотика сортировки O(n * log(n)).

Оптимизиация

В качестве оптимизированной версии предлагается использовать функцию сортировки кучей из стандатрной библиотеки C++:

```
template<typename T>
void std_heapsort(std::vector<T>& v) {
   std::make_heap(v.begin(), v.end());
   std::sort_heap(v.begin(), v.end());
}
```

Опираясь на стандарт языка, заметим, что **std::make_heap** ~ O(n) (https://en.cppreference.com/w/cpp/algorithm/make_heap (https://en.cppreference.com/w/cpp/algorithm/make_heap)), а **std::sort_heap** ~ O(n * log(n)) (https://en.cppreference.com/w/cpp/algorithm/sort_heap (https://en.cppreference.com/w/cpp/algorithm/sort_heap)).

Тестирование

Тестирование предлагается проводить на сгенерированных случайным образом одномерных массивах разных размеров и типов данных. Были выбраны три типа данных:

- int - ожидается наибольшая скорость сравнения двух элементов
- float - те же 4 байта, но уже сравнения между числами с плавающей точкой
- double - ожидаются наибольшие затраты времени

Временем выполнения (в мс) будем считать среднее по 5 запускам с предварительным "разогревом".

Simple/Std	**5e5**	**7.5e5**	**1e6**	**2.5e6**	**5e6**
int	91 / 79	144 / 126	206 / 170	624 / 527	1497 / 2020
float	94 / 81	151 / 130	216 / 181	649 / 552	6440 / 5643
double	119 / 87	190 / 145	272 / 240	855 / 667	8171 / 6612

Выводы

Из проведенных экспериментов можно сделать вывод, что реализация из STL превосходит кастомную реализацию. Можно предположить следующие причины: использование концепции итераторов, хвостовая рекурсия, компиляция intrinsic инструкций, меньшее количество ветвлений (кэш миссы, несправедливый рандом, нечестный планировщик, ...).

В конечном итоге можно прийти к мнению, что не надо изобретать велосипед, а использовать по максимуму стандартную библиотеку, поскольку она протестирована временем, выигрывает по перформансу и написана гораздо более умными людьми.