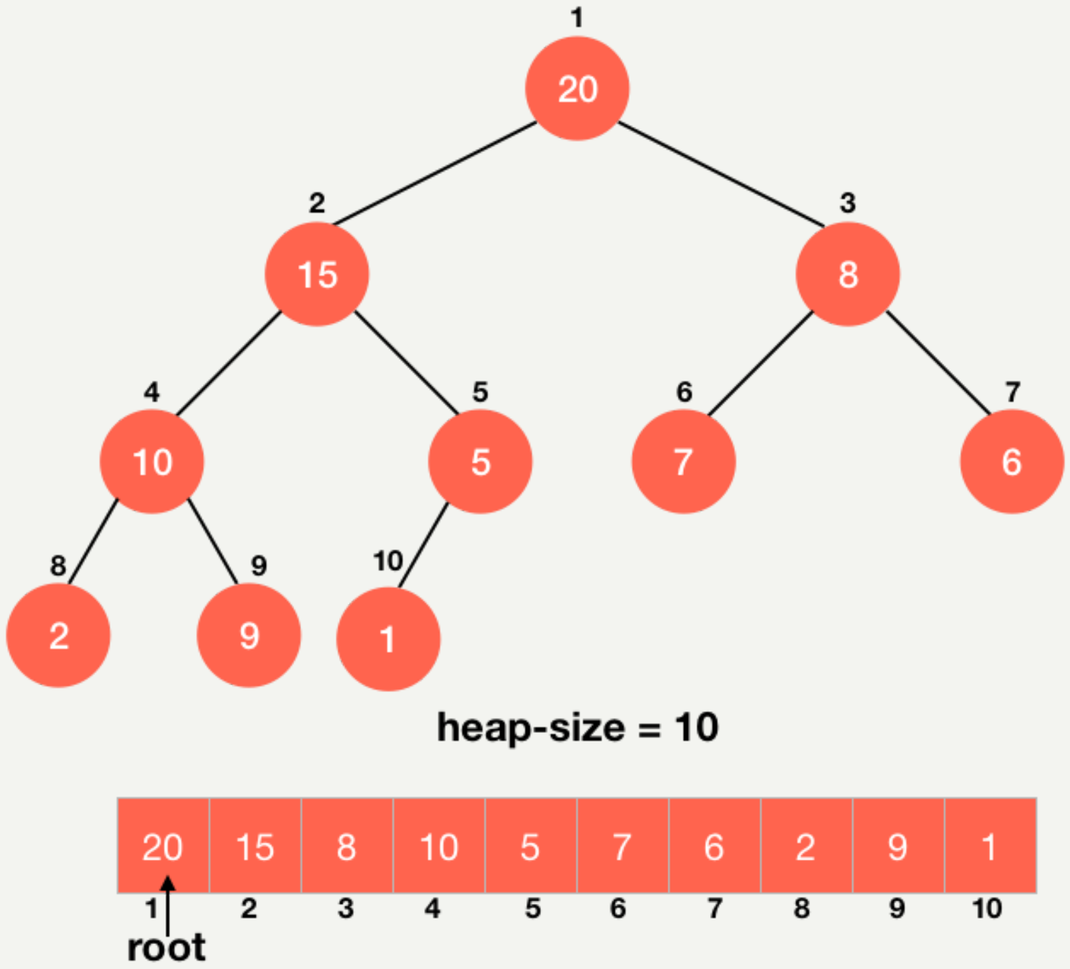
HEAPSORT

(пирамидальная сортировка)

Подготовил: Волков Артемий

ИСТОРИЯ ПИРАМИДАЛЬНОЙ СОРТИРОВКИ



HeapSort является одним из лучших и эффективных методов сортировки в компьютерном программировании и была изобретена Дж. У. Дж. Уильямсом в 1964 году. Сортировка кучей включает создание структуры данных кучи из заданного массива, а затем использование кучи для сортировки массива.

КРАТКАЯ ИСТОРИЯ

ОСНОВНЫЕ ПРИНЦИПЫ РАБОТЫ

Пирамидальная сортировка (или сортировка кучей, HeapSort) — это метод сортировки сравнением, основанный на такой структуре данных как двоичная куча. Она похожа на сортировку выбором, где мы сначала ищем максимальный элемент и помещаем его в конец. Далее мы повторяем ту же операцию для оставшихся элементов.

Двоичная куча — это законченное двоичное дерево, в котором элементы хранятся в особом порядке: значение в родительском узле больше (или меньше) значений в его двух дочерних узлах. Первый вариант называется max-heap, а второй — min-heap. Куча может быть представлена двоичным деревом или массивом.

Поскольку двоичная куча — это законченное двоичное дерево, ее можно легко представить в виде массива, а представление на основе массива является эффективным с точки зрения расхода памяти. Если родительский узел хранится в индексе I, левый дочерний элемент может быть вычислен как 2 I + 1, а правый дочерний элемент — как 2 I + 2 (при условии, что индексирование начинается с 0).

Алгоритм пирамидальной сортировки в порядке по возрастанию:
1) Постройте max-heap из входных данных.
2) На данном этапе самый большой элемент хранится в корне кучи. Замените его на последний элемент кучи, а затем уменьшите ее размер на 1. Наконец, преобразуйте полученное дерево в max-heap с новым корнем.
3) Повторяйте вышеуказанные шаги, пока размер кучи больше 1.

```
    1 usage
    public void sort(int arr[])
    {
        int n = arr.length;

        // Построение кучи (перегруппируем массив)
        for (int i = n / 2 - 1; i ⩾ 0; i--)
            heapify(arr, n, i);
            count++;



        // Один за другим извлекаем элементы из кучи
        for (int i=n-1; i⩾0; i--)
        {
            // Перемещаем текущий корень в конец
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            count++;
            // Вызываем процедуру heapify на уменьшенной куче
            heapify(arr, i,  i: 0);
        }
    }

    // Процедура для преобразования в двоичную кучу поддерева с корневым узлом i, что является
// индексом в arr[]. n - размер кучи
    3 usages
    void heapify(int arr[], int n, int i)
    {
        int largest = i; // Инициализируем наибольший элемент как корень
        int l = 2*i + 1; // левый = 2*i + 1
        int r = 2*i + 2; // правый = 2*i + 2

        // Если левый дочерний элемент больше корня
        if (l < n && arr[l] > arr[largest])
            largest = l;
            count++;

        // Если правый дочерний элемент больше, чем самый большой элемент на данный момент
        if (r < n && arr[r] > arr[largest])
            largest = r;
            count++;
        // Если самый большой элемент не корень
        if (largest ≠ i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
            count++;
            // Рекурсивно преобразуем в двоичную кучу затронутое поддерево
            heapify(arr, n, largest);
        }
    }
```

ОЦЕНКА ВРЕМЕННОЙ СЛОЖНОСТИ

В методе sort, мы используем цикл перегруппирующий массив в кучу, который запускается один раз для создания кучи и имеет время выполнения O(n). Затем, мы вызываем heapify для каждого узла, чтобы поддерживать свойство max-heap всякий раз, когда мы удаляем или вставляем узел в кучу. Поскольку узлов 'n', общее время выполнения алгоритма оказывается равным O (n (log (n)), и мы используем функцию max-heapify для каждого узла.
Математически мы видим, что:
1) Первое удаление узла занимает log(n) времени
2) Второе удаление занимает log(n-1) времени
3) Третье удаление занимает log(n-2) времени
4) И так далее до последнего узла, что займет log(1) времени
Таким образом, суммируя, мы получаем:

log(n) + log(n-1) + log(n-2) + ... + log(1)
так как log(x) + log(y) = log(x * y), мы получаем
=log(n∗(n−1)∗(n−2)∗...∗2∗1)
=log(n!)
При дальнейшем упрощении (используя приближение Стирлинга) log(n!) оказывается
=n∗log(n)−n+O(log(n)) = O(n(log(n))

приближение Стирлинга:

$$\ln \Gamma(n+1) = \ln n! = n \ln n - n + O(\ln n)$$

ИНФОРМАЦИЯ О ВХОДНЫХ ДАННЫХ

Входные данные задаются как (int) (Math.random() * (i+1))
Всего 1000 наборов данных, начиная с 1100 элементов в одном наборе, заканчивая 101000 элементами в одном наборе (в каждом последующем наборе данных на 100 элементов больше чем в предыдущем)

```
for (int j = 0; j < 1000; j++) {
    for (int i = 0; i < (1000 + count); i++) {
        x = String.valueOf( obj: (int) (Math.random()  * (i+1)) + ";");
        fileOutputStream.write(x.getBytes());
    }
    count = count + 100;
}
```

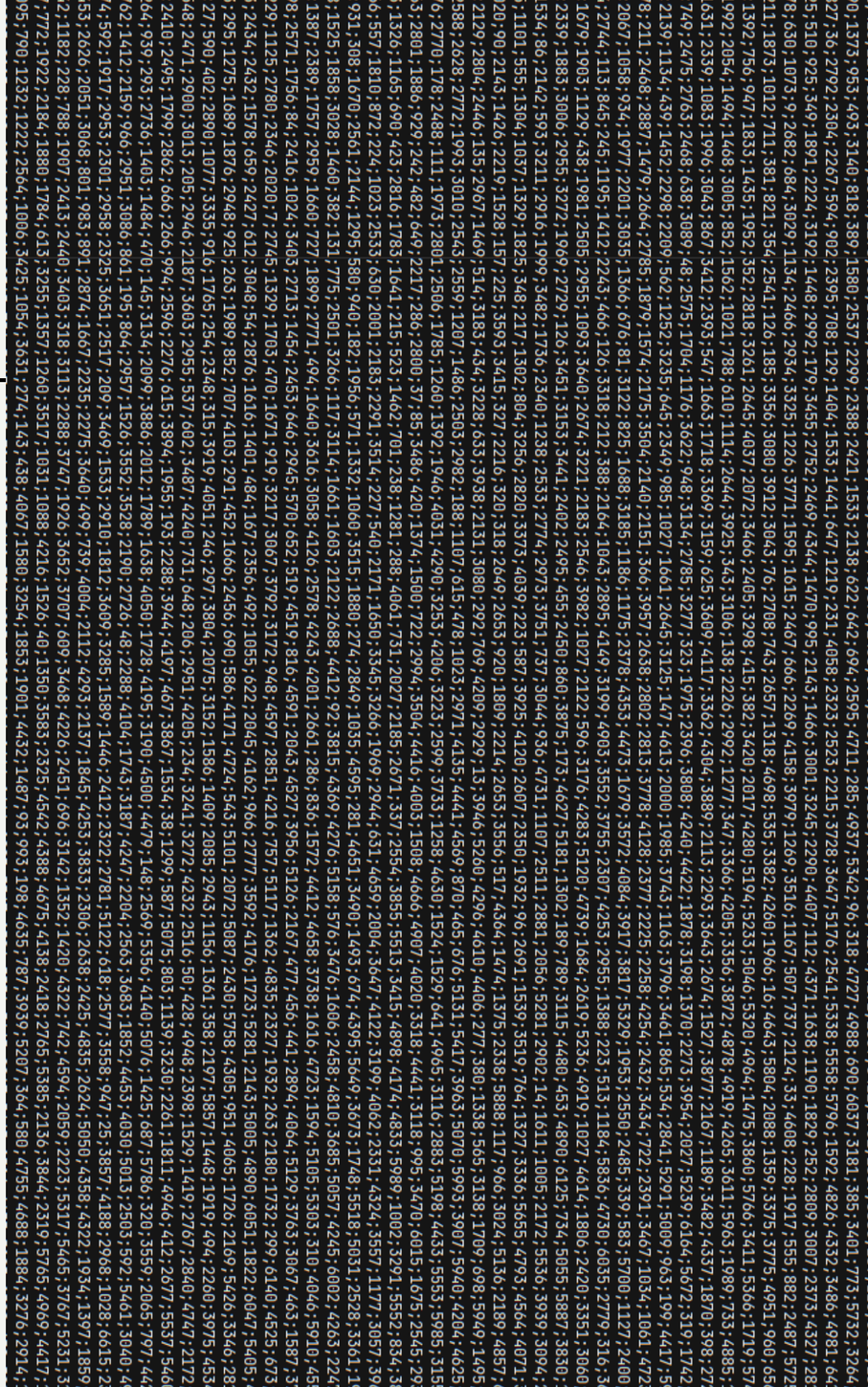


ТАБЛИЦА ПОЛУЧЕННЫХ ЗНАЧЕНИЙ КОЛИЧЕСТВА ИТЕРАЦИЙ И ВРЕМЕНИ РАБОТЫ В ЗАВИСИМОСТИ ОТ РАЗМЕРА ДАННЫХ

количество итераций	время (нс)	длинна набора
32352	820200	1100
68149	183400	1200
107406	151900	1300
150054	159700	1400
196363	174500	1500
246186	189500	1600
299541	197700	1700
356371	212200	1800
416751	234000	1900
480630	232400	2000
548134	240300	2100
619320	287500	2200
694272	269700	2300
772981	285100	2400
855549	296400	2500
941886	306600	2600
1031872	328100	2700
1125783	335800	2800
1223403	303100	2900
1324840	340600	3000
1430226	261400	3100
1539396	224700	3200
1652362	207700	3300
1769208	207500	3400
1889733	216800	3500
2014201	220400	3600
2142261	230200	3700
2274210	245900	3800
2410162	242600	3900
2549922	249100	4000
2693310	337800	4100
2841007	260800	4200
2992563	270800	4300
3148149	336200	4400
3308005	291500	4500
3471978	335800	4600
3639837	340000	4700
3811606	397400	4800
3987588	413500	4900
4167696	484800	5000
4351789	342100	5100
4540092	343400	5200
4732506	346000	5300
4929001	355300	5400
5129559	362300	5500
5334369	370600	5600
5543113	377700	5700
5755962	371600	5800
5972664	375700	5900
6193483	378200	6000

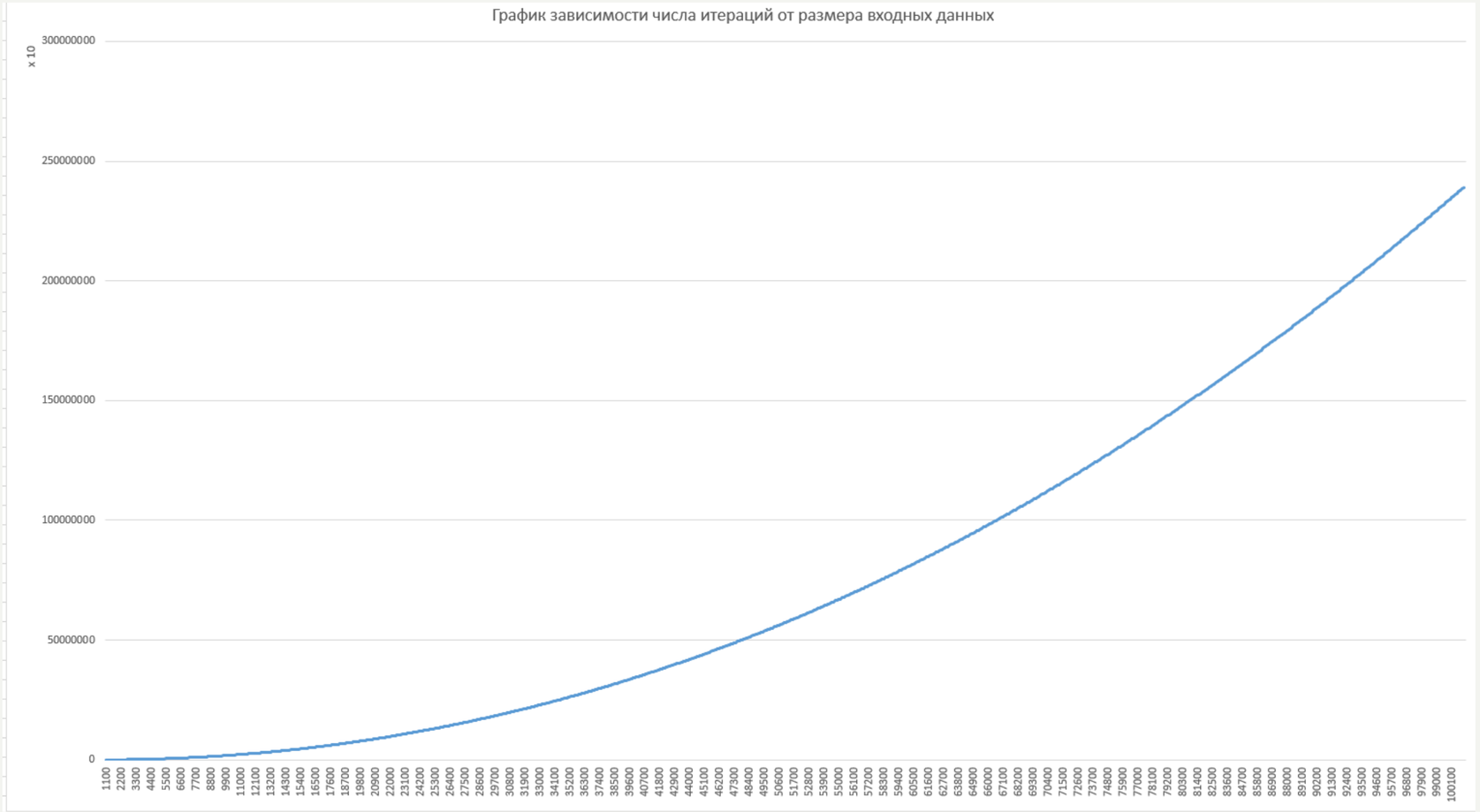


График зависимости количества итераций от объема входных данных

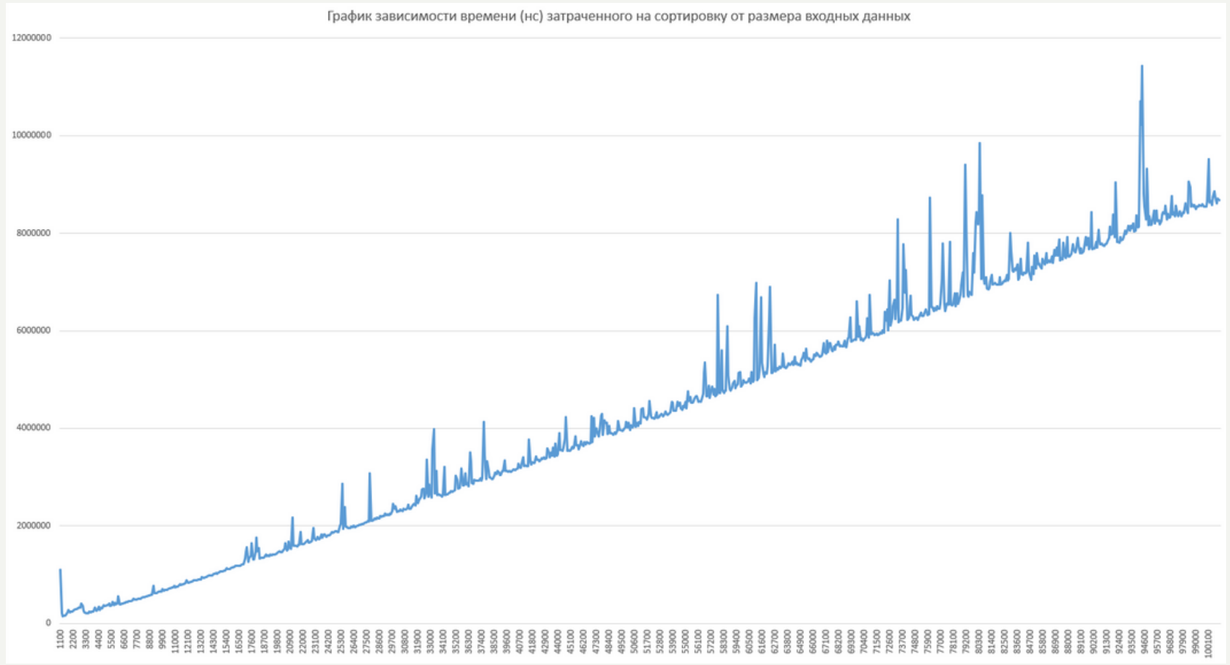


График зависимости времени (нс) от объема входных данных (на практике)

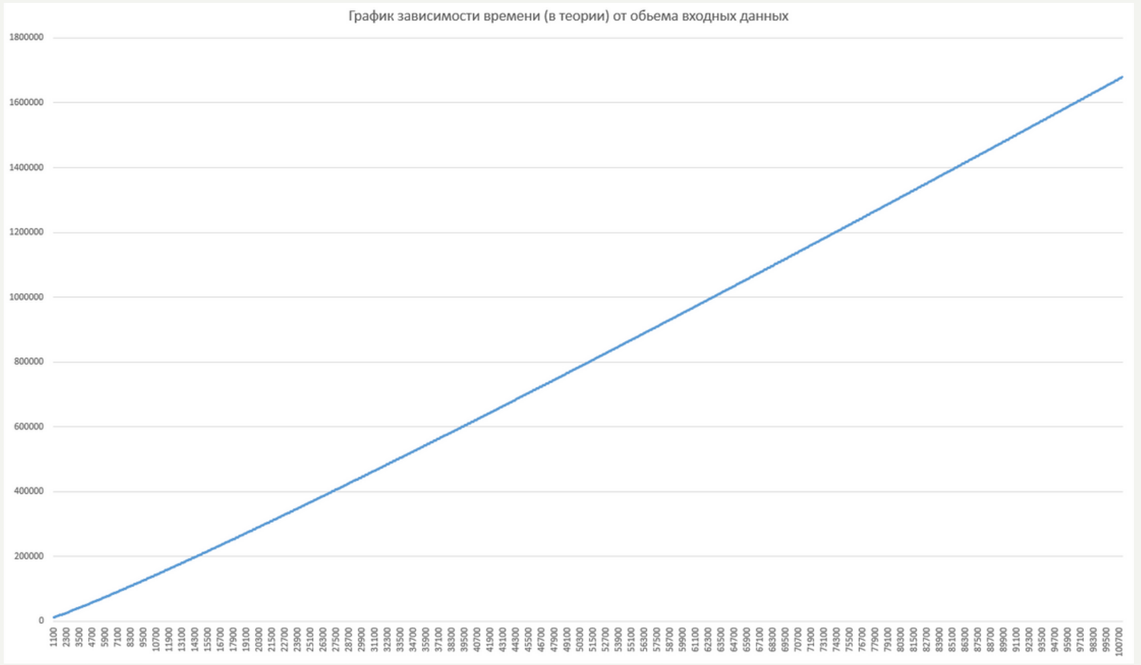


График зависимости времени от объема входных данных (в теории)

ПОЧЕМУ ГРАФИКИ НА ПРАКТИКЕ И В ТЕОРИИ РАЗЛИЧАЮТСЯ?

Теоретически скорость пирамидальной сортировки определяется количеством сравнений и перестановок, необходимых для сортировки входных данных, что обычно выражается как временная сложность O(n*log(n)) . Эта временная сложность основана на предположении, что все операции занимают постоянное время, независимо от размера входных данных.

Однако на практике на скорость пирамидальной сортировки может влиять ряд факторов, не учитываемых при теоретическом анализе. Некоторые из этих факторов включают:

Эффекты кэширования. Если данные не хранятся в кеше, то время, необходимое для их извлечения из памяти, может быть намного больше, чем время, необходимое для выполнения сравнения или подкачки. В результате фактическое время работы пирамидальной сортировки может быть больше, чем предсказывает теоретический анализ.

Детали реализации. Фактическая реализация пирамидальной сортировки также может существенно повлиять на ее производительность. Например, способ построения кучи или порядок, в котором элементы меняются местами, могут повлиять на количество требуемых сравнений и свапов, что, в свою очередь, может повлиять на время выполнения.

Аппаратные ограничения. На производительность пирамидальной сортировки также может влиять аппаратное обеспечение, на котором она выполняется. Например, если процессор имеет небольшой кэш, то производительность пирамидальной сортировки может быть ниже ожидаемой.

Характеристики входных данных. Производительность пирамидальной сортировки также может зависеть от характеристик входных данных. Например, если данные уже частично отсортированы или содержат много дубликатов, то количество требуемых сравнений и перестановок может быть меньше, чем предсказано теоретическим анализом, что приводит к ускорению времени выполнения.

Следовательно, хотя теоретическая временная сложность heapsort составляет O(n*log(n)), на ее фактическую производительность может повлиять ряд практических факторов. На практике фактическое время выполнения пирамидальной сортировки может быть медленнее или быстрее, чем время в теории, в зависимости от реализации и характеристик входных данных и оборудования.

ПЛЮСЫ И МИНУСЫ HEAPSORT

Плюсы сортировки кучей:

1) Он имеет временную сложность O(n log n), что делает его эффективным для сортировки больших наборов данных.
2) Это алгоритм сортировки на месте, что означает, что для выполнения операции сортировки не требуется дополнительная память.
3) Он стабилен, то есть сохраняет порядок одинаковых элементов во входных данных.
5) Его относительно легко реализовать и понять.

Минусы сортировки кучей:

1) Он имеет большой постоянный коэффициент, что означает, что это может быть не самый быстрый алгоритм сортировки для небольших наборов данных.
2) Это не дружественный к кэшу алгоритм, а это означает, что он может страдать от низкой производительности при использовании с большими наборами данных, которые не помещаются в кэш-память.

ПРИМЕНИМОСТЬ HEAPSORT

Эта сортировка особенно полезна для сортировки больших наборов данных, где память является ограничением. Свойство сортировки на месте делает его популярным выбором для встраиваемых систем и других сред с ограниченным объемом памяти. Кроме того, его свойство стабильности делает его подходящим для приложений, где важно сохранять порядок одинаковых элементов, например, в запросах к базе данных. Однако низкая производительность кэш-памяти может ограничить его полезность в некоторых приложениях, где часто осуществляется доступ к данным из памяти. В целом, сортировка кучей — хороший выбор для сортировки больших наборов данных с ограниченным объемом памяти, где важна стабильность, а первоначальный порядок данных не имеет значения.