Алгоритмы сортировки

Владимир Борисенко

Мехмат МГУ

vladibor2016@yandex.ru
vladimir_borisen@mail.ru

Задача сортировки

Формулировка. Дан массив длины n элементов некоторого типа, для которого определен линейный порядок (т.е. любые элементы сравнимы между собой). Требуется упорядочить элементы массива по возрастанию:

$$a[0] \leq a[1] \leq \cdots \leq a[n-1].$$

Задача сортировки возникает часто в самых различных ситуациях — например, многие задачи в геометрии, такие, как вычисление выпуклой оболочки множества точек на плоскости, сводятся к сортировке. На решение задач поиска и сортировки компьютер тратит значительную часть своего времени, поэтому очень важно использовать оптимальные алгоритмы для их решения.
Задачам сортировки и поиска посвящен третий том знаменитой книги Д.Кнута "Искусство прогаммирования", своеобразной библии программистов.

Наивные алгоритмы сортировки

Пузырьковая сортировка. Просматриваем массив от начала к концу, если два соседних элемента образуют инверсию, т.е. $a[i] > a[i+1]$, то меняем их местами. Повторяем это до тех пор, пока на очередном проходе инверсий не будет. Вот реализация алгоритма на C++.

```
void bubbleSort(double *a, int n) {
    bool wereInversions = true; // Были инверсии
    while (wereInversions) {
        wereInversion = false;
        for (int i = 0; i < n-1; ++i) {
            if (a[i] > a[i+1]) {
                double tmp = a[i];     // Меняем 2 элемента
                a[i] = a[i+1]; a[i+1] = tmp;
                wereInversion = true; // Запомним, что
            }                         // были инверсии
        }
    }
}
```

Сортировка прямым выбором. Просматривая массив от начала до конца, находим минимальный элемент и ставим его в начало массива. Затем просматриваем элементы массива, начиная со второго элемента, находим среди них минимальный и ставим его на второе место, и так далее.

```
void directSort(double *a, int n) {
    for (int i = 0; i < n-1; ++i) {
        int indMin = i;     // Индекс минимального элемента
        for (int j = i+1; j < n; ++j) {
            if (a[j] < a[indMin])
                indMin = j; // Запомним новый индекс мин. эл-та
        }
        if (indMin != i) {
            double tmp = a[i];      // Ставим элемент a[indMin]
            a[i] = a[indMin]; a[indMin] = tmp;  // на место i
        }
    }
}
```

Время работы наивных алгоритмов сортировки $t = O(n^2)$. Такие алгоритмы можно применять только для совсем небольших массивов.

Оценка снизу числа сравнений в произвольном алгоритме сортировки

Теорема 1. Для любого алгоритма сортировки найдется входной массив, для которого придется выполнить не меньше чем $t = \log_2(n!)$ сравнений.

Теорема 1 дает точную оценку минимального числа сравнений в произвольном алгоритме сортировки (для самого плохого входа). Применив формулу Стирлинга для функции $n!$, можно получить асимптотическую оценку для минимального числа сравнений в алгоритме сортировки.

Следствие 2. Асимптотически минимальное число сравнений в произвольном алгоритме сортировки при $n \to \infty$ эквивалентно

$$t = \log_2(n!) \sim n \log_2 n.$$

Доказательство. Применим формулу Стирлинга.

$$n! \sim n^n e^{-n} \sqrt{2\pi n} \quad \Rightarrow$$
$$\log_2(n!) = n \log_2 n + o(n \log_2 n) \sim n \log_2 n.$$

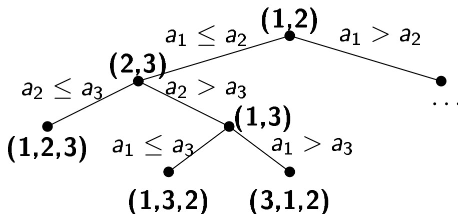
Доказательство оценки снизу для числа сравнений в алгоритме сортировки

Для иллюстрации теоремы 1 покажем, что в оптимальном алгоритме сортировки для массива длины 4 выполняется в худшем случае $t = 5$ сравнений. Действительно, $t \geq \log_2(4!) = \log_2 24 \approx 4.585 > 4$, значит, $t \geq 5$, поскольку t — целое число. Можно указать конкретный алгоритм, который упорядочивает массив длины 4 максимум за 5 сравнений — это *сортировка слиянием*, которая будет рассмотрена ниже.

Доказательство теоремы 1 основано на рассмотрении *бинарных деревьев*. Можно воспользоваться следующей аналогией. Элементы массива — это баскетбольные команды (в баскетболе нет ничьих), сравнение — это матч между командами, алгоритм сортировки — это расписание турнира, в результате которого команды должны быть упорядочены по их силе. Мы считаем, что упорядочение транзитивно, т.е. если $a \leq b$ и $b \leq c$, то $a \leq c$, т.е. проводить матч между командами a и c уже не обязательно.

Бинарное дерево алгоритма сортировки

Изобразим работу алгоритма сортировки в виде *бинарного дерева*. Вершины дерева соответствуют парам индексов сравниваемых элементов: пара (i, j) соответствует сравнению элементов $a[i]$ и $a[j]$ (матчу команд i и j). Если $a[i] \leq a[j]$, то алгоритм переходит к узлу, являющемуся *левым сыном* данного узла, иначе — к *правому сыну*. Алгоритм заканчивается в *терминальных узлах*, которые соответствуют различным упорядочениям исходного массива. Максимальное число сравнений равно $h - 1$, где h — *высота дерева*.
Для примера рассмотрим дерево алгоритма для турнира трех команд. Сначала играют команды 1 и 2, если $a[1] \leq a[2]$, то переходим вниз по левой ветке, иначе по правой, и т.д. Терминальные вершины соответствуют окончательным расстановкам команд, всего их $3! = 6$. Высота дерева равна 4, максимальное число сравнений равно $4 - 1 = 3$.

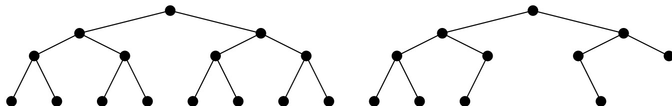


Число терминальных узлов полного дерева

Определение. *Полностью сбалансированным* или *полным* бинарным деревом называется дерево, удовлетворяющее следующим двум свойствам:

- у всех нетерминальных узлов дерева ровно 2 сына;
- длины всех путей от корня к терминальным узлам равны между собой.

На рисунке слева изображено полное дерево высоты 4, справа — дерево, не являющееся полным:

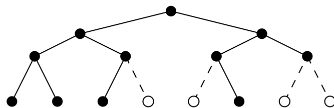


Предложение 3. Число терминальных узлов полностью сбалансированного бинарного дерева высоты h равно 2^{h-1}.

Оценка снизу высоты бинарного дерева

Следствие 4. Пусть T — произвольное бинарное дерево высоты h, имеющее k терминальных узлов. Тогда $h >= \log_2(k) + 1$.

Доказательство. Достроим дерево T до полного дерева T' той же высоты, как показано на рисунке:



При этом число терминальных вершин k' полного дерева может только увеличиться: $k' \geq k$. Для полного дерева по Предложению 3 имеем:

$$k' = 2^{h-1}$$

откуда с учетом неравенства $k' \geq k$ вытекает

$$\log_2 k' = h - 1 \quad \Rightarrow \quad h = \log_2 k' + 1 \geq \log_2 k + 1.$$

Завершение доказательства Теоремы 1

Для завершения доказательства Теоремы 1 рассмотрим произвольный алгоритм сортировки, на вход которого подается массив длины n. Построим бинарное дерево работы этого алгоритма — "турнирное дерево". Поскольку массив можно упорядочить $n!$ способами, у дерева алгоритма число терминальных узлов k должно быть не меньше, чем $n!$. По Следствию 4, высота h дерева оценивается снизу:

$$h \geq \log_2 k + 1.$$

Учитывая неравенство $k \geq n!$ и монотонность функции $\log_2 x$, получаем:

$$h \geq \log_2(n!) + 1.$$

Но время работы алгоритма в худшем случае равно $t = h - 1$. Отсюда следует:

$$t = h - 1 \geq \log_2(n!).$$

Теорема доказана.

Оптимальные алгоритмы сортировки

Итак, согласно Теореме 1, невозможно в общем случае придумать алгоритм, который при больших n работал бы быстрее, чем за $t = n \log_2 n$ шагов. Отметим, что как в теории, так и на практике нас интересует лишь зависимость времени работы от n. Поэтому алгоритм сортировки считается *оптимальным*, если время его работы есть $O(n \log_2 n)$; константа, входящая в определение O-большого, особого значения не имеет.

Существует несколько оптимальных алгоритмов:

- сортировка кучей или пирамидой — HeapSort;
- сортировка слиянием — MergeSort;
- поразрядная сортировка — RadixSort;
- быстрая сортировка — QuickSort (оптимальна лишь в среднем).

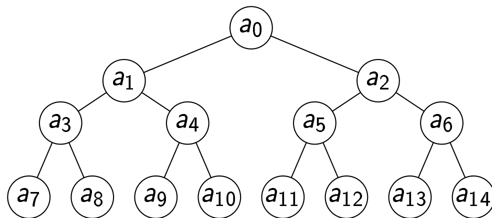
Популярный алгоритм QuickSort, строго говоря, не является оптимальным — он работает в среднем за время $O(n \log_2 n)$, но для любой стратегии выбора стержневого элемента всегда существует вход, на котором алгоритм работает за время $O(n^2)$ — правда, вероятность такого события очень мала.

Сортировка кучей HeapSort

Идея сортировки HeapSort состоит в использовании *бинарной кучи* (binary heap). Кучу также называют *очередью с приоритетами* (priority queue). Куча — это структура данных, в которой в любой момент времени доступен максимальный элемент. Его можно забрать из кучи, после чего структура кучи восстанавливается за время $O(\log_2 n)$, где n — число элементов в куче. Поскольку функция $\log_2 n$ растет очень медленно, можно считать, что операции с кучей выполняются практически мгновенно.
Добавление элемента к куче также выполняется за время $O(\log_2 n)$, поэтому куча из n элементов массива строится за время $O(n \log_2 n)$. Когда куча уже построена, можно забрать из нее максимальный элемент и поставить на последнее место в массиве, после чего куча восстанавливается за время $O(\log_2 n)$. Затем эта процедура повторяется для предпоследнего элемента, и так далее, пока в куче не останется элементов. Всего выполняется $O(n \log_2 n)$ действий, так что и суммарное время построения кучи + упорядочения массива есть $O(n \log_2 n)$.

Реализация бинарной кучи

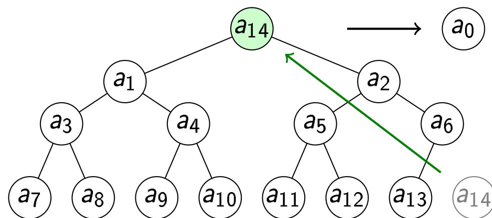
Массив можно наделить структурой бинарного дерева, не используя никаких дополнительных структур данных. Считаем, что элемент a_0 расположен в корне дерева. Сыновьями элемента a_i будут элементы a_{2i+1} (левый сын) и a_{2i+2} (правый сын). В результате все элементы массива будут расположены в вершинах бинарного дерева следующим образом:



Массив будет *кучей* или *пирамидой*, если элементы убывают при движении по дереву сверху вниз: $a_i \geq a_{2i+1}, a_{2i+2}$ — сыновья не превосходят отца.

Восстановление пирамиды при удалении максимального элемента

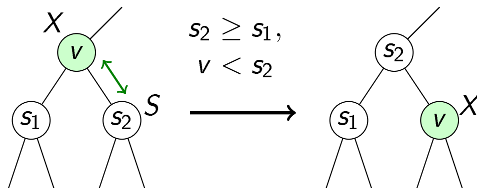
При удалении корня дерева на его место ставится самый нижний правый элемент, при этом размер пирамиды уменьшается на единицу.



Условие пирамиды при этом может нарушаться для корневого элемента, поэтому для него выполняется процедура *просеивания* (или bubleDown — движение пузырька вниз).

Процедура просеивания

Пусть условие пирамиды нарушено для некоторого узла X дерева: узел меньше одного из своих сыновей s_1, s_2. Тогда определяем сына с максимальным значением $S = \max(s_1, s_2)$ и меняем местами значения в узле X и в узле, содержащем старшего сына S. При этом отметка X перемещается вниз по дереву, и мы при необходимости переходим к исправлению нижестоящего узла.



Поскольку высота дерева для кучи не превосходит $\log_2(n) + 2$, то всего в процедуре просеивания выполняется не более $O(\log_2 n)$ шагов.

Реализация процедуры просеивания на C/C++

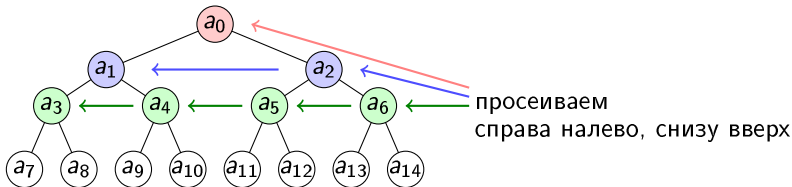
Реализуем функцию просеивания *bubbleDown* на C++. Мы работаем с кучей, состоящей из вещественных чисел. При реализации мы используем вспомогательную функция swap, которая меняет местами 2 элемента.

```
// В массиве a длины n просеиваем элемент с индексом x
void bubbleDown(double* a, int n, int x) {
    while (true) {
        int s1 = 2*x + 1; // Левый сын узла x
        if (s1 >= n)      // Сыновей нет =>
            break;        //     завершить цикл
        int s2 = s1 + 1;  // Правый сын
        int s = s1;       // Старший сын
        if (s2 < n && a[s2] > a[s1])
            s = s2;
        if (a[x] >= a[s]) // Условие пирамиды выполняется =>
            break;        //     завершить цикл
        swap(&(a[x]), &(a[s])); // Меняем значения в узлах x, s
        x = s;            // Перемещаемся вниз по дереву
    }
}
```

Построение пирамиды

```
void swap(double* a, double* b) { // Обмен двух элементов
    double tmp = *a; *a = *b; *b = tmp;
}
```

Построение кучи выполняется с помощью той же самой функции просеивания. Первоначально нам дан массив, в котором условие пирамиды выполняется только для тех элементов, у которых нет сыновей. Применяем процедуру просеивания для второго снизу слоя пирамиды, при этом двигаемся справа налево. В результате условие пирамиды будет выполнено для двух нижних слоев. Повторяем этот шаг для третьего снизу слоя и так далее, пока не дойдем до вершины пирамиды. После просеивания корня дерева пирамида будет построена.



Реализация функции построения пирамиды на C/C++

```
void makePyramid(double* a, int n) {
    int k = n/2;
    assert(2*k + 1 >= n); // Утв: у узла k нет сыновей
    while (k > 0) {
        // Инвариант: a[k], a[k+1], ..., a[n-1] -- пирамида
        --k;
        bubbleDown(a, n, k);
    }
}
```

Теперь мы можем написать функцию сортировки кучей HeapSort. На первом этапе с помощью функции *makePyramid* строится пирамида, после чего на втором этапе выполняется собственно сортировка.

Реализация функции heapSort на C/C++

Поскольку максимальный элемент пирамиды находится в начале массива, то начальный элемент надо поставить на последнее место. Меняем местами начальный и последний элементы массива, отсекаем от пирамиды последний элемент и восстанавливаем структуру пирамиды, просеивая элемент в корне дерева. Повторяем этот процесс, пока в пирамиде содержится больше одного элемента. Поскольку число повторений равно $n - 1$, а функция просеивания выполняется за $\log_2 n$ шагов, то общее время сортировки равно $O(n \log_2 n)$.

```
void heapSort(double* a, int n) {
    makePyramid(a, n);
    int k = n;
    while (k > 1) {
        // Инвариант: a[0], a[1], ..., a[k-1] -- пирамида
        // a[0], ..., a[k-1] <= a[k] <= a[k+1] <= ... <= a[n-1]
        --k;
        swap(&(a[0]), &(a[k]));
        bubbleDown(a, k, 0);
    }
}
```

Сортировка слиянием Merge Sort

Сортировка слиянием Merge Sort была предложена Джоном фон Нейманом в 1945 г. Алгоритм работает за время $O(n \log_2 n)$. Единственным недостатком классического варианта этого алгоритма является то, что он требует дополнительной памяти размера $O(n)$. Однако достоинства этого алгоритма перекрывают его недостаток. Среди оптимальных алгоритмов сортировки он является единственным *стабильным*. (Стабильным является также алгоритм Radix Sort, но он сортирует только ключи специального вида.)

Определение

Алгоритм сортировки называется *стабильным*, если он сохраняет взаимный порядок равных элементов.

К примеру, пусть мы сортируем таблицу студентов по их именам, при этом в таблице, помимо имен, содержится и другая информация (номер группы, оценки экзаменов и т.п.). Пусть таблица содержит однофамильцев. Тогда после применения стабильного алгоритма сортировки относительный порядок однофамильцев сохранится.

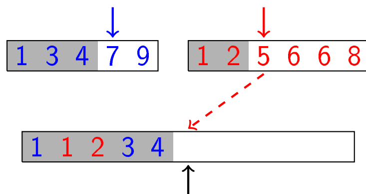
Процедура слияния массивов Merge

Вторым достоинством алгоритма Merge Sort является то, что он применим не только к массивам, но и к спискам — последовательным структурам данных, которые не обеспечивают прямого доступа к произвольным элементам.

Основная идея алгоритма состоит в применении процедуры слияния массивов *merge*, которая сливает два упорядоченных массива a и b в один упорядоченный массив c. Вначале устанавливаем указатели в начала обоих массивов. Затем сравниваем текущие элементы массивов a и b, меньший из них копируется в выходной массив c. Указатель в том массиве, из которого скопирован элемент, сдвигается вправо на одну позицию. Это повторяется до тех пор, пока указатель в одном из массивов не дойдет до конца, после чего остается только переписать оставшиеся элементы второго массива в выходной массив c.

Процедура слияния массивов Merge

Рисунок иллюстрирует процедуру слияния двух массивов:



Число сравнений в процедуре слияния двух массивов равно минимуму из длин массивов, число копирований равно сумме длин. Таким образом, время выполнения процедуры слияния равно $O(n)$, где n — сумма длин массивов.

Реализация функции merge на C/C++

```
void merge(
    const double* a, int n, const double* b, int m, double* c
) {
    int i = 0, j = 0, k = 0;
    while (i < n && j < m) {
        if (a[i] <= b[j]) {
            c[k] = a[i]; ++i;
        } else {
            c[k] = b[j]; ++j;
        }
        ++k;
    }
    while (i < n) {
        c[k] = a[i]; ++i; ++k;
    }
    while (j < m) {
        c[k] = b[j]; ++j; ++k;
    }
}
```

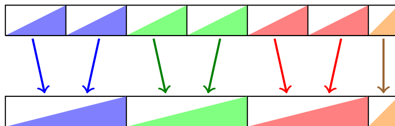
Восходящая и нисходящая схемы реализации Merge Sort

Существуют две схемы реализации сортировки слиянием: рекурсивная, или нисходящая, и восходящая. Обе схемы требуют вспомогательной памяти размера n (хотя существуют реализации, в которых требуется вдвое меньший объем памяти).

В *нисходящей схеме* массив делится на две части и к ним рекурсивно применяется алгоритм сортировки. Упорядоченные половины массива сливаются в один отсортированный массив во вспомогательной памяти, который затем копируется в исходный массив.

В *восходящей схеме* массив разбивается на подмассивы первоначально длины 1. Соседние подмассивы объединяются в пары и к каждой паре применяется процедура слияния. В результате получаются упорядоченные подмассивы вдвое большой длины. Это повторяется в цикле до тех пор, пока в результате слияния не останется только 1 подмассив. Такая схема называется *двусторонней*, поскольку на нечетных шагах результат получается во вспомогательной памяти, на четных — в исходном массиве. Если общее число шагов нечетно, то в конце требуется скопировать результат из вспомогательной памяти в исходный массив.

Глубина рекурсии в нисходящей схеме, так же как и число шагов в восходящей схеме, равна $O(\log_2 n)$. На каждом шаге выполняется $O(n)$ операций, поэтому полное время работы алгоритма есть $O(n \log_2 n)$. Восходящая схема предпочтительнее, т.к. она не нагружает стек и не тратится времени на вызовы функции. Рассмотрим реализацию восходящей схемы. Ее идея иллюстрируется следующим рисунком:



Здесь изображен один шаг восходящего алгоритма. Массив-источник разбит на упорядоченные подмассивы одинаковой длины (кроме последнего), они объединены в пары. Подмассивы в каждой паре сливаются в один подмассив вдвое большей длины. Если количество подмассивов нечетно, то последний подмассив просто копируется в выходной массив.

Реализация восходящей схемы Merge Sort на C/C++

```
void mergeSort(double* a, int n) {
    if (n <= 1) return;
    double* b = new double[n]; // Вспомогательная память
    double *src = a, *dst = b; // Источник, получатель
    int len = 1;      // Длина подмассивов
    while (len < n) { // Пока не весь массив упорядочен
        int i = 0;    // Индекс начала текущей пары подмассивов
        while (i < n-len) { // Цикл для каждой пары подмассивов
            int len2 = len; // длина второго подмассива пары
            if (i + len + len2 > n)
                len2 = n - (i + len);
            merge(          // Сливаем пару подмассивов
                src + i, len,
                src + i + len, len2,
                dst + i
            );
            i += len + len2; // Переходим к следующей паре
        }
```

```
        // Если число подмассивов нечетно, то копируем
        if (i < n)    // последний подмассив в выходной массив
            copyArray(src+i, n-i, dst+i);
        len *= 2;     // Удваиваем длину подмассивов

        // Меняем местами массив-источник и получатель
        double *tmp = src; src = dst; dst = tmp;
    }
    if (src != a)        // Если число шагов нечетно, то копируем
        copyArray(b, n, a); // вспомогательный массив в основной
    delete[] b;          // Освобождаем вспомогательную память
}
```

Реализация, помимо уже написанной функции *merge*, использует также вспомогательную функцию $copyArray(a, n, b)$, которая копирует массив a длины n в массив b.

```
void copyArray(const double* a, int n, double* b) {
    for (int i = 0; i < n; ++i)
        b[i] = a[i];
}
```