Лекция 11

Алгоритмы сортировки и поиска

Мы используем следующую классификацию, упорядоченную по *убыванию* эффективности.

- Константная: $O(1)$
- Логарифмическая: $O(log\ n)$
- Сублинейная: $O(n^d)$ при $d < 1$
- Линейная: $O(n)$
- Линейно-логарифмическая: $O(n \cdot log\ n)$
- Квадратичная: $O(n^2)$
- Экспоненциальная: $O(2^n)$

Сортировка вставками

Сортировка вставками (Insertion Sort) многократно использует вспомогательную функцию `insert`, чтобы обеспечить корректную сортировку $A[0, i]$; в конечном итоге i достигает крайнего справа элемента, сортируя тем самым все множество A целиком.

Сортировка вставками

Наилучший случай: O(n) Средний, наихудший случаи: O(n^2)

```
sort (A)
    for pos = 1 to n-1 do
        insert (A, pos, A[pos])
end

insert (A, pos, value)
    i = pos - 1
    while i >= 0 and A[i] > value do    ❶
        A[i+1] = A[i]
        i = i-1
    A[i+1] = value                      ❷
end
```

❶ Сдвигаем элементы, большие value, вправо.
❷ Вставляем value в корректное местоположение.

На рис. 4.3 показано, как работает сортировка вставками на неупорядоченном множестве A размера $n = 16$. 15 нижних строк на рисунке отображают состояние A после каждого вызова `insert`.

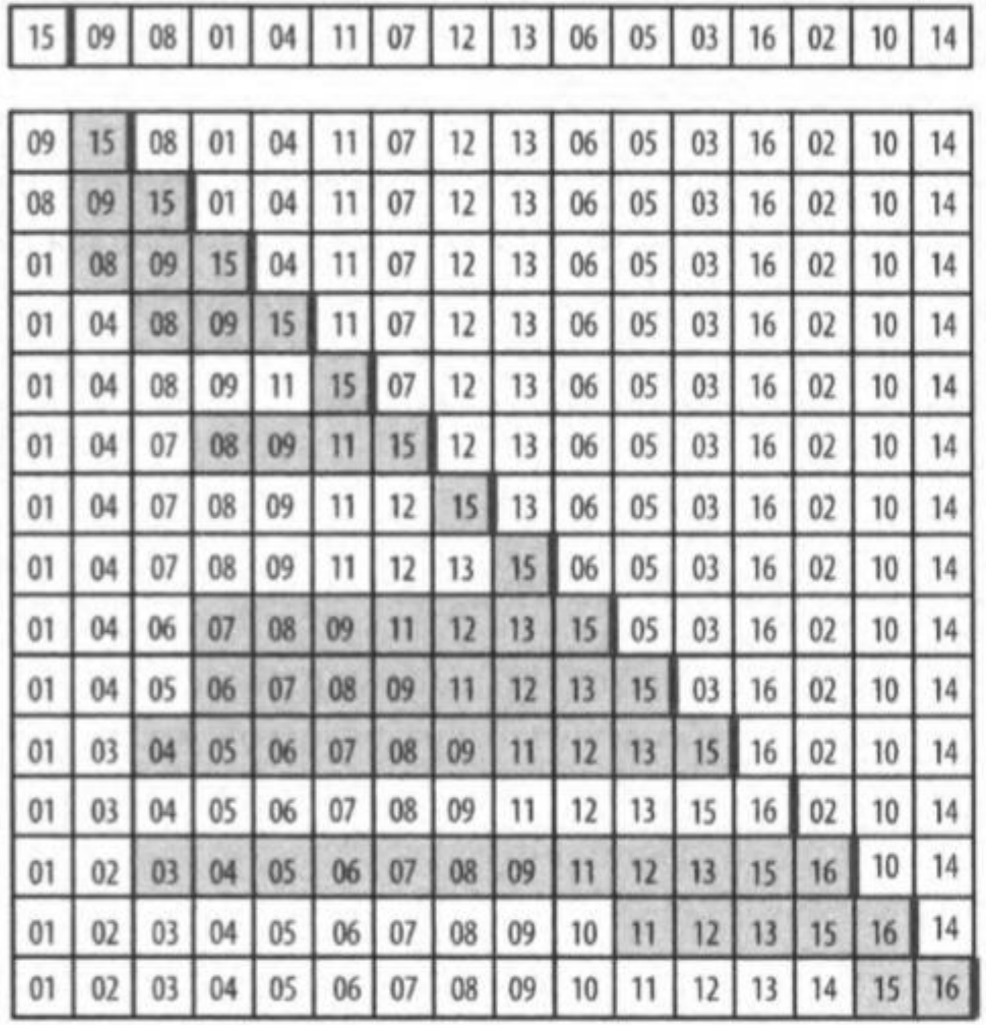

Рис. 4.3. Процесс сортировки вставками небольшого массива

Массив A сортируется без привлечения дополнительной памяти путем увеличения $pos = 1$ до $n - 1$ и вставки элемента $A[pos]$ в его корректную позицию в сортируемом диапазоне $A[0, pos]$, на рисунке ограниченном справа толстой вертикальной линией. Выделенные серым цветом элементы сдвигаются вправо, чтобы освободить место для вставляемого элемента; в целом для данного массива сортировка вставками выполнила 60 обменов соседних элементов (перемещений элементов на одну позицию).

Контекст применения алгоритма

Используйте сортировку вставками, когда у вас есть небольшое количество сортируемых элементов или когда элементы исходного множества уже "почти отсортированы". Критерий "достаточной малости" зависит от конкретной машины и языка программирования. В действительности может играть роль даже тип сравниваемых элементов.

Сортировка выбором

Одна распространенная стратегия сортировки заключается в том, чтобы выбрать наибольшее значение из диапазона $A[0,n)$ и поменять его местами с крайним справа элементом $A[n-1]$. Впоследствии этот процесс повторяется поочередно для каждого меньшего диапазона $A[0,n-1)$ до тех пор, пока массив A не будет отсортирован.

Пример 4.3. Реализация на языке C сортировки выбором

```
static int selectMax(void** ar, int left, int right,
                     int (*cmp)(const void*, const void*))
{
    int maxPos = left;
    int i = left;

    while (++i <= right)
    {
        if (cmp(ar[i], ar[maxPos]) > 0)
        {
            maxPos = i;
        }
    }

    return maxPos;
}
```

```
void sortPointers(void** ar, int n,
                  int (*cmp)(const void*, const void*))
{
    /* Многократный выбор максимального значения в диапазоне ar[0,i]
       и его обмен с перестановкой в корректное местоположение */
    int i;

    for (i = n - 1; i >= 1; i--)
    {
        int maxPos = selectMax(ar, 0, i, cmp);

        if (maxPos != i)
        {

          void* tmp = ar[i];
          ar[i] = ar[maxPos];
          ar[maxPos] = tmp;
        }
    }
}
```

Сортировка выбором является самым медленным из всех алгоритмов сортировки, описанных в этой главе. Она требует квадратичного времени даже в лучшем случае (т.е. когда массив уже отсортирован). Эта сортировка многократно выполняет почти одну и ту же задачу, ничему не обучаясь от одной итерации к следующей. Выбор наибольшего элемента *max* в массиве *A* требует $n-1$ сравнений, а выбор второго по величине элемента *second* выполняется с помощью $n-2$ сравнений — не слишком большой прогресс! Многие из этих сравнений затрачены впустую, например, потому, что если некоторый элемент меньше элемента *second*, то он никак не может быть наибольшим элементом, а потому никак не влияет на вычисление *max*.

Пирамидальная сортировка

Чтобы найти наибольший элемент в неупорядоченном массиве A из n элементов, требуется как минимум $n-1$ сравнений, но, может, есть способ свести к минимуму количество непосредственно сравниваемых элементов? Например, на спортивных турнирах "наилучшую" среди n команд определяют без того, чтобы каждая команда сыграла со всеми $n-1$ соперниками. Одним из самых популярных баскетбольных событий в США является чемпионат, на котором, по сути, за титул чемпиона соревнуются 64 команды колледжей. Команда-чемпион играет с пятью командами для выхода в финал. Так что, чтобы стать чемпионом, команда должна выиграть шесть игр. И не случайно $6 = log(64)$. **Пирамидальная сортировка** (Heap Sort) показывает, как применить такое поведение для сортировки множества элементов.

Пирамидальная сортировка

Наилучший, средний, наихудший случаи: $O(n \cdot \log n)$

```
sort (A)
    buildHeap (A)
    for i = n-1 downto 1 do
        swap A[0] with A[i]
        heapify (A, 0, i)
end

buildHeap (A)
    for i = n/2-1 downto 0 do
        heapify (A, i, n)
end

# Рекурсивное обеспечение условия, что A[idx,max)
# является корректной пирамидой
heapify (A, idx, max)
    largest = idx                    ❶
    left = 2*idx + 1
    right = 2*idx + 2
```

```
    if left < max and A[left] > A[idx] then
        largest = left                    ❷
    if right < max and A[right] > A[largest] then
        largest = right                   ❸
    if largest ≠ idx then
        swap A[idx] and A[largest]
        heapify (A, largest, max)
end
```

❶ Предполагается, что родитель A[idx] не меньше любого из потомков.
❸ Левый потомок больше родителя.
❷ Правый потомок больше родителя или левого "брата".

Контекст применения алгоритма

Пирамидальная сортировка не является устойчивой. Она позволяет избежать многих неприятных (почти неловких) ситуаций, которые приводят к плохой производительности быстрой сортировки. Тем не менее в среднем случае быстрая сортировка превосходит пирамидальную.

Анализ алгоритма

Центральной операцией пирамидальной сортировки является `heapify`. В `buildHeap` она вызывается $\lfloor n/2 \rfloor - 1$ раз, а за все время сортировки — $n-1$ раз, в общей сложности $\lfloor 3n/2 \rfloor - 2$ раз. Из-за свойства формы глубина пирамиды всегда равна $\lfloor \log n \rfloor$, где n — количество элементов в пирамиде. Как можно видеть, это рекурсивная операция, в которой выполняется не более чем $\log n$ рекурсивных вызовов до момента полного исправления пирамиды или достижения ее конца. Однако `heapify` может остановиться преждевременно, как только пирамида будет исправлена. Как оказалось, в общей сложности необходимо не более чем $2n$ сравнений [20], так что поведение `buildHeap` описывается как $O(n)$.

Сортировка, основанная на разбиении

Стратегия "разделяй и властвуй" решает задачу, разделяя ее на две независимые подзадачи, каждая из которых размером около половины размера исходной задачи. Эту стратегию можно применить к сортировке следующим образом: найти *медианный* элемент в коллекции A и поменять его со средним элементом A. Теперь будем менять местами элементы из левой половины, большие, чем $A[mid]$, с элементами в правой половине, которые меньше или равны $A[mid]$. Это действие разделяет исходный массив на два отдельных подмассива, которые можно отсортировать рекурсивно и получить отсортированную исходную коллекцию A.

Реализация этого подхода является сложной задачей, потому что не очевидно, как вычислить средний элемент коллекции без предварительной сортировки. Но оказывается, что для разделения A на два подмассива можно использовать любой элемент A. При "мудром" выборе этого элемента оба подмассива будут более или менее одинакового размера, так что можно получить эффективную реализацию данного алгоритма.

Предположим, что имеется функция `partition(A,left,right,pivotIndex)`, которая использует специальное *опорное* значение *pivot* из A (которое представляет собой $A[pivotIndex]$) для того, чтобы модифицировать A, и возвращает местоположение p в A такое, что:

- $A[p] = pivot$;
- все элементы из $A[left, p)$ не превышают опорного значения;
- все элементы из $A[p+1, right]$ больше опорного значения.

Если вам повезет, то по завершении работы функции *partition* размеры этих двух подмассивов будут более или менее близки к половине размера исходной коллекции. В примере 4.5 показана реализация этой функции на языке программирования C.

Пример 4.5. Реализация функции `partition` *для массива* `ar[left,right]` *и указанного опорного элемента*

```
/**
 * Данная функция за линейное время группирует подмассив
 * ar[left,right] вокруг опорного элемента pivot = ar[pivotIndex]
 * путем сохранения pivot в корректной позиции store (которая
 * возвращается данной функцией) и обеспечения выполнения условий:
 * все ar[left,store) < = pivot и все ar[store+1,rigt] > pivot.
*/
int partition(void** ar, int (*cmp)(const void*, const void*),
              int left, int right, int pivotIndex)
{
    int idx, store;
    void* pivot = ar[pivotIndex];

    /* Перемещаем pivot в конец массива */
    void* tmp = ar[right];
    ar[right] = ar[pivotIndex];
    ar[pivotIndex] = tmp;
```

```
    /* Все значения, не превышающие pivot, перемещаются в начало
     * массива, и pivot вставляется сразу после них. */
    store = left;
    for (idx = left; idx < right; idx++)
    {
        if (cmp(ar[idx], pivot) <= 0)
        {
            tmp = ar[idx];
            ar[idx] = ar[store];
            ar[store] = tmp;
            store++;
        }
    }

    tmp = ar[right];
    ar[right] = ar[store];
    ar[store] = tmp;
    return store;
}
```

Алгоритм **быстрой сортировки** (Quicksort), разработанный Ч.Э.Р. Хоаром (C.A.R. Hoare) в 1960 году, выбирает элемент коллекции (иногда случайно, иногда крайний слева, иногда средний) для разбиения массива на два подмассива. Таким образом, быстрая сортировка состоит из двух этапов. Сначала массив разбивается на два, затем рекурсивно сортируется каждый из подмассивов.

Быстрая сортировка

Наилучший и средний случаи: (n·log n), наихудший случай: $O(n^2)$

```
sort (A)
    quicksort (A, 0, n-1)
end

quicksort (A, left, right)
    if left < right then
    pi = partition (A, left, right)
        quicksort (A, left, pi-1)
        quicksort (A, pi+1, right)
end
```

Контекст применения алгоритма

Быстрая сортировка демонстрирует наихудшее квадратичное поведение, если разбиение на каждом шаге рекурсии делит коллекцию из n элементов на "пустое" и "большое" множества, где одно из этих множеств не имеет элементов, а другое содержит $n-1$ элементов. (Обратите внимание, что опорный элемент является крайним из n элементов, поэтому ни один элемент не теряется.)

Реализация алгоритма

Показанная в примере 4.6 реализация быстрой сортировки включает стандартную оптимизацию — сортировку вставками, когда размер подмассива сокращается до предопределенного минимального размера.

Пример 4.6. Реализация быстрой сортировки на языке программирования C

```
/**
 * Сортирует массив ar[left,right] с использованием метода
 * быстрой сортировки. Функция сравнения cmp необходима для
 * корректного сравнения элементов.
*/
void do_qsort(void** ar, int (*cmp)(const void*, const void*),
              int left, int right)
{
    int pivotIndex;

    if (right <= left)
    {
        return;
    }
```

```
    /* Разбиение */
    pivotIndex = selectPivotIndex(ar, left, right);
    pivotIndex = partition(ar, cmp, left, right, pivotIndex);

    if (pivotIndex - 1 - left <= minSize)
    {
        insertion(ar, cmp, left, pivotIndex - 1);
    }
    else
    {
        do_qsort(ar, cmp, left, pivotIndex - 1);
    }

    if (right - pivotIndex - 1 <= minSize)
    {
        insertion(ar, cmp, pivotIndex + 1, right);
    }
    else
    {
        do_qsort(ar, cmp, pivotIndex + 1, right);
    }
}

/** Вызов быстрой сортировки */
void sortPointers(void** vals, int total_elems,
```

```
                int (*cmp)(const void*, const void*))
{
    do_qsort(vals, cmp, 0, total_elems - 1);
}
```

Внешний метод `selectPivotIndex(ar,left,right)` выбирает значение опорного элемента, относительно которого выполняется разбиение массива.

Анализ алгоритма

Удивительно, но при использовании в качестве опорного случайного элемента быстрая сортировка демонстрирует производительность для среднего случая, которая обычно превосходит производительность всех прочих алгоритмов сортировки. Кроме того, имеется целый ряд исследований по усовершенствованию и оптимизации быстрой сортировки, которые позволяют достичь эффективности, превышающей эффективность любого алгоритма сортировки.

В идеальном случае `partition` делит исходный массив пополам, и быстрая сортировка демонстрирует производительность $O(n \cdot log\, n)$. На практике быстрая сортировка вполне эффективна при выборе опорного элемента случайным образом.

В наихудшем случае в качестве опорного элемента выбирается наибольший или наименьший элемент массива. Когда это происходит, быстрая сортировка делает проход по всем элементам массива (за линейное время) для того, чтобы отсортировать единственный элемент в массиве. Если этот процесс повторится $n-1$ раз, это приведет к наихудшему поведению данного метода сортировки — $O(n^2)$.

Последовательный поиск

Последовательный поиск (Sequential Search), называемый также линейным поиском, является самым простым из всех алгоритмов поиска. Это метод поиска одного значения *t* в коллекции *C* "в лоб". Он находит *t*, начиная с первого элемента коллекции и исследуя каждый последующий элемент до тех пор, пока не просмотрит всю коллекцию или пока соответствующий элемент не будет найден.

Для работы этого алгоритма должен иметься способ получения каждого элемента из коллекции, в которой выполняется поиск; порядок извлечения значения не имеет. Часто элементы коллекции *C* могут быть доступны только через *итератор*, который предназначен только для чтения и получает каждый элемент из *C*, как, например, курсор базы данных в ответ на SQL-запрос. В книге рассмотрены оба режима доступа.

Входные и выходные данные алгоритма

Входные данные представляют собой непустую коллекцию C из $n > 0$ элементов и целевое значение t, которое мы ищем. Поиск возвращает `true`, если коллекция C содержит t, и `false` в противном случае.

Последовательный поиск

Наилучший случай: $O(1)$; средний и наихудший случаи: $O(n)$

```
search (A,t)
    for i=0 to n-1 do                ❶
        if A[i] = t then
            return true
    return false
end

search (C,t)
    iter = C.begin()
    while iter ≠ C.end() do          ❷
        e = iter.next()              ❸
        if e = t then
            return true
    return false
end
```

❶ Поочередный доступ к каждому элементу от позиции 0 до n-1.
❷ Итерации продолжаются до полного исчерпания элементов.
❸ Все элементы последовательно извлекаются из итератора.

Контекст применения алгоритма

Зачастую требуется найти элемент в коллекции, которая может быть не упорядочена. Без дополнительных знаний об информации, хранящейся в коллекции, последовательный поиск выполняет работу методом "грубой силы". Это единственный алгоритм поиска, который можно использовать, если коллекция доступна только через итератор.

Если коллекция неупорядоченная и хранится в виде связанного списка, то вставка элемента является операцией с константным временем выполнения (он просто добавляется в конец списка). Частые вставки в коллекцию на основе массива требуют управления динамической памятью, которое либо предоставляется базовым языком программирования, либо требует определенного внимания со стороны программиста. В обоих случаях ожидаемое время поиска элемента равно $O(n)$; таким образом, удаление элемента требует по крайней мере времени $O(n)$.

Последовательный поиск накладывает наименьшее количество ограничений на типы искомых элементов. Единственным требованием является наличие функции проверки совпадения элементов, необходимой для выяснения, соответствует ли искомый элемент элементу коллекции; часто эта функция возлагается на сами элементы.

Реализация алгоритма

Обычно реализация последовательного поиска тривиальна. В примере 5.1 показана реализация последовательного поиска на языке программирования Python.

Пример 5.1. Последовательный поиск на языке Python

```
def sequentialSearch(collection, t):
    for e in collection :
        if e == t:
            return True
    return False
```

Таблица 5.1. Производительность последовательного поиска (в секундах)

n	*p* = 1,0	*p* = 0,5	*p* = 0,25	*p* = 0,0
4096	0,0057	0,0087	0,0101	0,0116
8192	0,0114	0,0173	0,0202	0,0232
16384	0,0229	0,0347	0,0405	0,0464
32768	0,0462	0,0697	0,0812	0,0926
65536	0,0927	0,1391	0,1620	0,1853
131072	0,1860	0,2786	0,3245	0,3705

Бинарный поиск

Бинарный (двоичный) поиск обеспечивает лучшую производительность, чем последовательный поиск, поскольку работает с коллекцией, элементы которой уже отсортированы. Бинарный поиск многократно делит отсортированную коллекцию пополам, пока не будет найден искомый элемент или пока не будет установлено, что элемент в коллекции отсутствует.

Входные и выходные данные алгоритма

Входными данными для бинарного поиска является индексируемая коллекция A, элементы которой полностью упорядочены, а это означает, что для двух индексов, i и j, $A[i] < A[j]$ только тогда, когда $i < j$. Мы строим структуру данных, которая хранит элементы (или указатели на элементы) и сохраняет порядок ключей. Выходные данные бинарного поиска — `true` или `false`.

Контекст применения алгоритма

Для поиска по упорядоченной коллекции в худшем случае необходимо логарифмическое количество проб.

Бинарный поиск поддерживают различные типы структур данных. Если коллекция никогда не изменяется, элементы следует поместить в массив (это позволяет легко перемещаться по коллекции). Однако, если необходимо добавить или удалить элементы из коллекции, этот подход становится неудачным и требует дополнительных расходов. Есть несколько структур данных, которые мы можем использовать в этом случае; одной из наиболее известных является бинарное дерево поиска, описываемое далее в этой главе.

Бинарный поиск

Наилучший случай: $O(1)$; средний и наихудший случаи: $O(\log n)$

```
search (A,t)
    low = 0
    high = n-1
    while low ≤ high do                    ❶
        mid = (low + high)/2               ❷
        if t < A[mid] then
            high = mid - 1
        else if t > A[mid] then
            low = mid + 1
        else
            return true
    return false                           ❸
end
```

❶ Повторяется, пока есть диапазон для поиска.

❷ Средняя точка вычисляется с помощью целочисленной арифметики.

❸ Далее, в разделе "Вариации алгоритма", обсуждается поддержка операции "поиск или вставка" на основе конечного значения mid в этой точке.

Реализация алгоритма

Для упорядоченной коллекции элементов, представленной в виде массива, в примере 5.3 приведен код Java параметризованной реализации бинарного поиска для произвольного базового типа T. Java предоставляет интерфейс `java.util.Comparable<T>`, содержащий метод `compareTo`. Любой класс, который корректно реализует этот интерфейс, гарантирует нестрогое упорядочение его экземпляров.

Пример 5.3. Реализация бинарного поиска на языке Java

```
/**
* Бинарный поиск в предварительно отсортированном массиве
* параметризованного типа.
*
* @param T Элементы коллекции имеют данный тип.
* Тип T должен реализовывать интерфейс Comparable.
*/
public class BinarySearch<T extends Comparable<T>>
{
```

```
    /** Поиск ненулевого элемента; возврат true в случае успеха. */
    public boolean search(T[] collection, T target)
    {
        if (target == null)
        {
            return false;
        }

        int low = 0, high = collection.length - 1;

        while (low <= high)
        {
            int mid = (low + high) / 2;
            int rc = target.compareTo(collection[mid]);

            if (rc < 0)        // Искомый элемент < collection[i]
            {
                high = mid - 1;
            }
            else if (rc > 0)   // Искомый элемент > collection[i]
            {
                low = mid + 1;
            }
            else               // Искомый элемент найден
            {
                return true;
            }
        }

        return false;
    }
}
```

Анализ алгоритма

Двоичный поиск на каждой итерации уменьшает размер задачи примерно в два раза. Максимальное количество делений пополам для коллекции размером n равно $\lfloor log\ n \rfloor+1$. Если одной операции достаточно, чтобы определить, равны ли два элемента, меньше или больше один другого (это делает возможным применение интерфейса `Comparable`), то достаточно выполнить только $\lfloor log\ n \rfloor+1$ сравнений. Таким образом, алгоритм можно классифицировать как имеющий производительность $O(log\ n)$.