3. Сортировка

Рассмотрим множества объектов A и B. Будем считать, что на множестве объектов из B задано отношение линейного порядка $\preccurlyeq$, а элементы множества A занумерованы числами от 1 до n, где n – число элементов в A. *Задача сортировки* заключается в том, чтобы для заданной функции $f: A \to B$ упорядочить элементы из A в соответствие с отношением $\preccurlyeq$. Применительно к числам это может означать их упорядочивание либо по возрастанию, либо по убыванию.

В общем случае, целью сортировки является поиск такой перестановки $\pi = (\pi_1, \ldots, \pi_n)$ элементов из A, для которой выполняются следующие условия:

$$f(a_{\pi_1}) \preccurlyeq f(a_{\pi_2}) \preccurlyeq \cdots \preccurlyeq f(a_{\pi_n}).$$

В дальнейшем будем предполагать, что отношение $\preccurlyeq$ суть отношение линейного порядка «меньше либо равно», и все последующие алгоритмы будут преследовать своей целью построение перестановки π, для которой

$$f(a_{\pi_1}) \leq f(a_{\pi_2}) \leq \cdots \leq f(a_{\pi_n}).$$

Методы сортировки могут быть классифицированы несколькими способами. Различают алгоритмы внешней сортировки и внутренней. Возможно деление на *устойчивые* и *неустойчивые* методы сортировки. Будем говорить, что алгоритм сортировки является *устойчивым*, если в процессе сортировки относительный порядок элементов с одинаковыми ключами не изменяется. Следует отметить, что устойчивые методы сортировки находят широкое применение на практике (в частности, некоторые из строковых алгоритмов используют устойчивую сортировку данных как промежуточный этап). Третья возможная схема классификации методов сортировки – их разделение в зависимости от того, на чем они основаны: на сравнении ключей или на некоторых свойствах ключей. Четвертым способом классификации является разделение алгоритмов сортировки по времени выполнения: квадратичные, логарифмические, линейные, экспоненциальные. Существуют и другие классификации, рассмотрение которых выходит за рамки данного пособия [9].

3.1. Общие сведения. Нижние оценки сложности

Будем говорить, что некоторый алгоритм сортировки упорядочивает элементы *на месте*[*], если для его успешного завершения требуется $O(1)$ единиц памяти. Таким образом, сортировкам «на месте» достаточно памяти, которая была выделена под массив, подлежащий упорядочиванию. Данный класс алгоритмов имеет важное практическое значение, поскольку алгоритмы, ему принадлежащие, могут быть пригодны для сортировки значительно больших объемов данных, нежели алгоритмы, затраты на память у которых составляют по крайней мере $O(n)$.

Независимо от того, какие данные упорядочивает алгоритм, интерес представляет получение оценки нижнего количества операций, необходимых

[*] В англоязычной литературе используется термин *in-place*.

для того, чтобы упорядочить данные в соответствие с заданным отношением порядка. Попробуем получить оценку нижнего числа операций для алгоритмов сортировки, базирующихся на сравнении.

Пусть имеется некий алгоритм $\mathcal{A}$, выполняющий сортировку элементов множества A путем сравнения ключей элементов. Алгоритму $\mathcal{A}$ можно поставить в соответствие *дерево сортировки,* имеющее следующую структуру. Узлы дерева содержат сравнение неких ключей элементов множества A a_i и a_j. Листья дерева представляют собой перестановку исходной последовательности, соответствующую отсортированному порядку. Количество сравнений, которое должен выполнить алгоритм $\mathcal{A}$ в конкретном случае, равняется количеству ребер, ведущих от корня дерева сортировки к некоторому его листу. Лучшему случаю соответствует самый короткий путь от корня к листу, худшему – самый длинный, а для подсчета среднего случая необходимо разделить суммарную длину всех путей от корня к листьям на количество листьев.

Количество возможных листьев дерева сортировки равно $n!$, где n – число элементов сортируемой последовательности. Пусть k – определенный уровень дерева сортировки, считая от корня (уровни нумеруются с нуля). Тогда количество узлов на k-ом уровне равно 2^k. В случае, когда дерево сортировки высоты k является идеально сбалансированным, для количества листьев l справедливо следующее неравенство:

$$n! \leq l \leq 2^k.$$

Логарифмируя данное неравенство и применяя формулу Стирлинга $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, получим следующую оценку для k:

$$k > (n + 0{,}5)\log_2 n - 1{,}4427n.. \qquad (3.1)$$

Таким образом, из неравенства (3.1) можно заключить, что нижняя оценка числа сравнений для алгоритма сортировки $\mathcal{A}$, использующего сравнения, удовлетворяет величине порядка $\Omega(n \log_2 n)$. Отметим, что существует и другое доказательство оценки $\Omega(n \log_2 n)$, базирующееся на том факте, что $n!$ является произведением не менее $n/2$ сомножителей, каждое из которых не превышает $n/2$, т. е. $n! \geq (n/2)^{n/2}$.

Следует отметить, что существует класс алгоритмов, которые работают за линейное время $O(n)$, относительно числа сортируемых элементов. Особенностью таких алгоритмов является использование некоторых особенностей ключа, позволяющих избежать попарного сравнения элементов. В качестве примера рассмотрим алгоритм сортировки подсчетом.

Пусть множество A состоит из n чисел от 0 до $k - 1$. Заведем массив подсчетов C из k элементов. Индекс массива подсчетов соответствует одному из элементов множества A. Перед началом сортировки обнулим массив C. Далее просмотрим элементы множества A и подсчитаем число различных элементов в нем. В силу того, что элементы множества A ограниченны, результат можно занести в массив C. После сбора необходимой статистики, построим отсортиро-

ванную последовательность, воспользовавшись информацией из массива C. Приведенный алгоритм иллюстрирует следующая процедура (3.1.1):

Процедура 3.1.1. Counting Sort

```
01:  Counting Sort Impl(A)
02:        C[i] = 0, i∈[0, k)
03:        foreach a in A do
04:              C[a] = C[a] + 1
05:        index = 1
06:        for key = 0 to k – 1 do
07:              for count = 0 to C[key] do
08:                    A[index] = key
09:                    index = index + 1
```

Проведем анализ алгоритма (3.1.1). Обнуление массива C требует k итераций, перебор всех элементов множества A требует n итераций. Операторы вложенного цикла в сумме выполнятся ровно n раз. Таким образом, время работы сортировки подсчетом можно оценить как $O(n + k)$. Следовательно, сортировка подсчетом выполняется за линейное, относительно числа элементов исходного массива, время. Узким местом приведенного алгоритма является наличие дополнительного массива C. Если, например, сортируется массив из 100 чисел, каждое из которых принимает значение от 0 до $2^{32} - 1$, то потребуется дополнительный массив из 2^{32} элементов. В этом случае ни о каком преимуществе по скорости (а тем более по используемой памяти) перед традиционными алгоритмами сортировки говорить не приходится.

3.2. Внутренняя сортировка

Под внутренней сортировкой будем понимать те алгоритмы сортировки или их реализации, используемые для упорядочивания данных, которые полностью помещаются в оперативную память ЭВМ в виде массива. Выбор массива в качестве структуры данных для хранения сортируемых элементов является не случайным, так как это единственная «родная» структура данных большинства ЭВМ, предоставляющая произвольный способ доступа к ячейкам массива. В таком случае затратами на доступ к элементам массива можно пренебречь и считать, что они выполняются за учетное время $O(1)$.

Ниже будут рассмотрены классические методы сортировки, которые условно могут быть разделены на квадратичные, логарифмические и линейные. Данная классификация обусловлена как временем выполнения алгоритмов сортировки, так и тем, что она является наиболее распространенной при выборе методов, используемых для упорядочивания данных.

3.2.1. Квадратичная сортировка

Методы квадратичной сортировки выполняют сортировку данных за $O(n^2)$ шагов и требуют $O(1)$ дополнительной памяти. Перед непосредственным изучением методов сортировки рассмотрим алгоритм, который является интуитивно понятным и базируется на том факте, что элементы исходного множества

должны быть упорядочены по не убыванию ключей. Как следствие, для достижения цели может быть использована процедура нахождения минимального элемента в массиве (процедура 3.2.1):

Процедура 3.2.1. Basic Quadratic Sort

```
01:  Basic Quadratic Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              {* f(A[outer]) is min from f(A[outer]), …, f(A[n]) *}
04:              for inner = outer + 1 to n do
05:                    if f(A[inner]) < f(A[outer]) then
06:                          swap(inner, outer)
```

Количество итераций, совершаемое приведенным алгоритмом пропорционально n^2, поскольку $(n-1)+(n-2)+\cdots+1=\frac{(n-1)^2+(n-1)}{2}$. Далее будут рассмотрены такие классические методы сортировки, как сортировка «пузырьком», выбором, вставками, и модификация сортировки вставками, предложенная Д. Л. Шеллом.

Сортировка пузырьком

Сортировка пузырьком является одним из простейших методов устойчивой сортировки, идея которого базируется на следующем наблюдении. Предположим, что элементы сортируемого массива A расположены вертикально. Тогда элемент с наименьшим значением ключа должен всплыть вверх, за ним всплывает следующий наименьший элемент и т. д. (процедура 3.2.2):

Процедура 3.2.2. Bubble Sort

```
01:  Bubble Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              for inner = n downto outer + 1 do
04:                    if f(A[inner]) < f(A[inner - 1]) then
05:                          swap(inner, inner - 1)
```

Проанализируем время работы алгоритма. Соответствующее рекуррентное соотношение будет иметь вид

$$\begin{cases} T(1)=0, \\ T(n)=T(n-1)+O(n),\ n>1. \end{cases} \quad (3.2)$$

Решением данного соотношения является функция $T(n)=O(n^2)$, поэтому сложность алгоритма пузырьковой сортировки есть $O(n^2)$, где n – число элементов в A.

Время, затрачиваемое сортировкой «пузырьком», может быть улучшено на частично упорядоченных массивах. Предположим, что после выполнения внутреннего цикла не произошло ни одного обмена. Это означает, что часть массива, которая была подвергнута анализу внутренним циклом, является отсортированной, и дальнейшую работу можно прекратить (процедура 3.2.3).

Следует обратить внимание на то, что в худшем случае приведенная оптимизация не даст выигрыша и сложность алгоритма пузырька по-прежнему составит $O(n^2)$ операций.

Процедура 3.2.3. Improved Bubble Sort

```
01:  Improved Bubble Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              swapped = false
04:              for inner = n downto outer + 1 do
05:                    if f(A[inner]) < f(A[inner - 1]) then
06:                          swap(inner, inner - 1)
07:                          swapped = true
08:              if not swapped then return
```

Сортировка выбором

Сортировка выбором, как и алгоритм пузырька, является простейшим методом упорядочивания данных и базируется на идее базового алгоритма сортировки, приведенного выше (процедура 3.2.1). Разница между алгоритмами заключается в моменте времени, в который происходит обмен элементов. Алгоритм сортировки выбором приведен в процедуре 3.2.4:

Процедура 3.2.4. Selection Sort

```
01:  Selection Sort Impl(A[1, n])
02:        for outer = 1 to n - 1 do
03:              index = outer
04:              for inner = outer + 1 downto n do
05:                    if f(A[inner]) < f(A[index]) then
06:                          index = inner
07:              if (index <> outer) then
08:                    swap(index, outer)
```

Рекуррентное соотношение для времени работы сортировки выбором не приводится по причине точного совпадения с выражением (3.2). Следовательно, временная сложность рассмотренного метода составляет $O(n^2)$ операций. Следует отметить, что сортировка выбором не принадлежит к классу устойчивых сортировок.

Сортировка вставками

Еще одним из методов квадратичной устойчивой сортировки является метод вставок, идея которого заключается в следующем. На каждой итерации алгоритм ставит текущий элемент в требуемую позицию (процедура 3.2.5):

Процедура 3.2.5. Insertion Sort

```
01:  Insertion Sort Impl(A[1, n])
02:        for outer = 2 to n do
03:              current = A[outer]
04:              if f(current) < f(A[outer – 1]) then
05:                    inner = outer
06:                    while inner > 0 do
07:                          if f(current) < f(A[inner - 1]) then
08:                                A[inner] = A[inner - 1]
09:                                inner = inner – 1
09:                          else break
10:                    A[inner] = current
```

Проведем анализ времени работы алгоритма (3.2.5). Основное внимание уделим операторам внутреннего цикла. Если исходный массив уже отсортирован по возрастанию, внутренний цикл не выполнится ни разу. Если же исходный массив изначально отсортирован по убыванию, внутренний цикл на каждой итерации главного цикла будет выполняться i раз, где i – счетчик главного цикла. Следовательно, полное число итераций, которое совершит внутренний цикл есть $\frac{(n-1)n}{2} = O(n^2)$ раз.

Проанализируем поведение алгоритма в среднем случае. Пусть текущим является i-ый элемент. В результате работы внутреннего цикла этот элемент может остаться на месте, а может занять одну из позиций в отсортированной левой части. Следовательно, i-ый элемент может занять одну из i позиций, которые будем полагать равновероятными. Если элемент остается на месте, будет выполнено ноль итераций внутреннего цикла, если сдвигается на одну позицию влево – одна итерация, и т. д. Значит, среднее число итераций для элемента на i-ой позиции равно $\frac{1}{i}(0 + 1 + 2 + .. + (i-1)) = \frac{i-1}{2}$. Просуммируем полученное выражение по всем i:

$$T(n) = \sum_{i=2}^{n} \frac{i-1}{2} = O(n^2). \qquad (3.3)$$

Итак, время работы алгоритма сортировки вставками на множестве из n элементов в лучшем случае есть $O(n)$, а в среднем и худшем случае – $O(n^2)$. В среднем случае выполняется примерно в два раза меньше операций, нежели в худшем.

Следует отметить тесную связь между числом инверсий[*] k в массиве A и числом операций, которое совершает алгоритм сортировки вставками. Так как названная сортировка меняет только соседние элементы, то количество инверсий в массиве будет уменьшаться на единицу на каждой итерации алгоритма. Можно заключить, что сложность алгоритма сортировки вставками составит $O(n + k)$ шагов. Данная оценка совпадает с полученной ранее оценкой для худшего и среднего случая, так как в массиве, упорядоченном в обратном порядке, есть $\frac{n(n-1)}{2}$ инверсий, что соответствует $O(n^2)$.

Сортировка бинарными вставками

Сортировка вставками допускает модификацию, идея которой базируется на том факте, что вставка следующего элемента осуществляется в отсортированную часть массива. Как следствие, поиск позиции, в которую будет вставляться текущий элемент, можно осуществить, используя правосторонний двоичный поиск (процедура 3.2.6)[**]:

---

[*] *Инверсией* в массиве $A[n]$ называется такая пара индексов i и j $(1 \le i < j \le n)$, для которой выполнено $f(A_i) > f(A_j)$.

[**] Двоичный поиск и его модификации будут рассмотрены в разделе 4.

Процедура 3.2.6. Binary Insertion Sort

```
01:  Binary Insertion Sort Impl(A[1, n])
02:        for outer = 2 to n do
03:              current = A[outer]
04:              if f(current) < f(A[outer – 1]) then
05:                    index = Get Index(A[1, outer – 1], f(current))
06:                    for inner = outer downto index + 1
07:                          A[inner] = A[inner - 1]
08:                    A[index] = current
```

Следует отметить, что алгоритм бинарной вставки не позволяет улучшить оценку $O(n)$ для числа перемещений элементов, несмотря на то, что число сравнений уменьшается до $O(\log n)$. Поэтому в худшем случае сортировка бинарными вставками будет по-прежнему работать за $O(n^2)$ шагов.

Модификация Шелла

Д. Л. Шелл в 1959 г. предложил следующую модификацию алгоритма сортировки вставками [10]. В отличие от алгоритма (3.2.5), Шеллом было предложено рассматривать не только элементы, расположенные в соседних позициях, но и элементы, отстоящие на некотором расстоянии h друг от друга. Далее алгоритм упорядочивает элементы, расположенные на некотором расстоянии меньшем h и т. д. Завершается сортировка Шелла упорядочиванием элементов, отстоящих друг от друга на расстоянии $h = 1$, то есть обычной сортировкой вставками. Последовательность шагов, которая была предложена Шеллом, имеет вид:

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, \frac{n}{2^k}, 1. \tag{3.4}$$

Сортировка методом Шелла, соответствующая набору шагов (3.4) приведена ниже (процедура 3.2.7):

Процедура 3.2.7. Shell Sort

```
01:  Shell Sort Impl(A[1, n])
02:        scale = 1
03:        while (n shr scale > 0) do {* shift n by scale bits to the right *}
04:              offset = n >> scale
05:              for outer = offset + 1 to n do
06:                   inner = outer
07:                   while inner > 0 do
08:                         if f(A[inner]) < f(A[inner - offset]) then
09:                               swap(inner, inner – offset)
10:                               inner = inner – offset
11:                         else break
12:              scale = scale + 1
```

Время работы процедуры (3.2.7) в худшем случае составляет $O(n^2)$ операций, что соответствует массиву данных, в котором половина элементов с меньшими значениями находится в четных позициях, а половина элементов с большими значениями – в нечетных позициях.

Помимо последовательности шагов (3.4), предложенной непосредственно Шеллом, существует множество других, которые дают лучшие оценки для худшего случая. Например, последовательность Д. Кнута [9]

$$1,4,13,40,\dots,\frac{3^k-1}{2},\frac{3^k-1}{2}\le\left\lceil\frac{n}{3}\right\rceil \tag{3.5}$$

и последовательность Р. Седжвика [11]

$$1,5,19,41,109,\dots,h_i=\begin{cases}9\cdot 2^i-9\cdot 2^{i/2}+1, & i-\text{четно},\\ 8\cdot 2^i-6\cdot 2^{\lceil i/2\rceil)}+1, & i-\text{нечетно},\end{cases}\quad 3\cdot h_i\le n \tag{3.6}$$

В случае последовательности Кнута алгоритм потребует $O(n\sqrt{n})$ операций, а для последовательности Седжвика – $O(n^{4/3})$ операций в худшем случае.

Следует отметить, что сортировка Шелла работает с любой убывающей последовательностью шагов, заканчивающейся единицей, так как после упорядочивания с шагом h_i элементы массива, упорядоченные на итерации $i-1$ с шагом h_{i-1} ($h_i < h_{i-1}$), по-прежнему остаются отсортированными [9].

3.2.2. Логарифмическая сортировка

Ниже будут рассмотрены алгоритмы сортировки, базирующиеся на сравнении ключей, время выполнения которых в среднем случае требует $O(n\log n)$ операций. Данная оценка является асимптотически наилучшей (как это было отмечено в разд. 3.1), поэтому названные алгоритмы получили широкое распространение на практике для сортировки больших объемов данных. Несмотря на оптимальную оценку для среднего случая, некоторые из логарифмических алгоритмов сортировки требуют порядка $O(n^2)$ операций в худшем случае.

Следует отметить, что как алгоритм быстрой сортировки, так и алгоритмы слияния и пирамидальной сортировки, имеет смысл применять лишь для упорядочивания множеств, число элементов в которых не превосходит $16*$. Для сортировки же небольших участков массива обычно применяется либо сортировка вставками, либо ее модификация – сортировка Шелла.

Быстрая сортировка

Алгоритм *быстрой сортировки* был предложен Ч. Хоаром в 1962 г. [9]. Идея алгоритма заключается в следующем. Вначале выбирается элемент множества A, называемый *осевым*, после чего происходит переупорядочивание элементов массива относительно ключа осевого элемента так, чтобы все элементы, расположенные левее осевого, были меньше его, а элементы, расположенные правее осевого, были больше его. В левой и правой частях (относительно осевого элемента) массива элементы остаются неупорядоченными. Следовательно, алгоритм быстрой сортировки требуется применить к левой и правой частям массива, т. е. вызвать рекурсивно (процедура 3.2.8):

---

[*] Выбор числа 16 обусловлен практическими результатами. В зависимости от специфики задачи данное значение может быть уменьшено, например, до 8 либо 4.

Процедура 3.2.8. Quick Sort

```
01:  return Quick Sort Impl(A[1, n], 1, n)
02:
03:  Quick Sort Impl(A, lo, hi)
04:        if lo < hi then
05:              i = lo, j = hi
06:              pivot = f(A[lo + ⌊(hi - lo)/2⌋])
07:              do
08:                    while f(A[i]) < pivot do i = i + 1
09:                    while pivot < f(A[j]) do j = j - 1
10:                    if i ≤ j then
11:                          swap(i, j)
12:                          i = i + 1
13:                          j = j - 1
14:              while i ≤ j
15:              if i < hi then Quick Sort Impl(A, i, hi)
16:              if lo < j then Quick Sort Impl(A, lo, j)
```

В силу того, что алгоритм быстрой сортировки имеет важное практическое значение, приведем детальный анализ времени его выполнения для среднего, лучшего и худшего случаев.

Начнем со среднего случая. Пусть $A(n)$ – количество сравнений в среднем для массива длины n. Так как быстрая сортировка является рекурсивным алгоритмом, найдем рекуррентное соотношение для $A(n)$. Очевидно, что $A(0) = A(1) = 0$. В общем случае осевой элемент может быть выбран произвольно. Пусть позиция осевого элемента равняется p. Тогда рекуррентное соотношение будет иметь вид:

$$\begin{cases} A(n) = 0,\ n < 2 \\ A(n) = A(n-1) + A(n-p) + (n-1),\ n \geq 2. \end{cases}$$

Будем считать, что значение p с равной вероятностью распределено от 1 до n. Таким образом

$$A(n) = (n-1) + \frac{1}{n}\sum_{i=1}^{n}\big(A(i-1) + A(n-i)\big).$$

В записанной сумме аргумент первого слагаемого пробегает значения от 0 до $n-1$, а аргумент второго слагаемого – те же значения, но в обратном порядке. Следовательно,

$$A(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} A(i).$$

Из выражения для $A(n) \cdot n - A(n-1) \cdot (n-1)$ получим, что

$$A(n) = \frac{(n+1)}{n}A(n-1) + \frac{2n-2}{n}.$$

Сделав замену $D(n) = \frac{A(n)}{n+1}$, получим следующее рекуррентное соотношение для $D(n)$:

$$D(n) = D(n-1) + 2\frac{n-1}{n(n+1)}.$$

Путем применения метода итераций к выражению для $D(n)$ можно показать, что замкнутый вид для $D(n)$ будет выражаться формулой

$$D(n) = 2\sum_{k=1}^{n}\frac{k-1}{k(k+1)} = 2\left(H_{n+1} + \frac{1}{n+1} - 2\right), \quad (3.7)$$

Из (3.7) следует выражение для $A(n)$

$$A(n) = (n+1)D(n) \approx 2(n+1)\ln(n+1) - (4-2\gamma)n \approx$$
$$\approx 2(n+1)\ln(n+1) = O(n\log n),$$

которое говорит о том, что быстрая сортировка имеет трудоемкость $O(n\log n)$ в среднем случае.

Оценим количество сравнений для алгоритма быстрой сортировки в лучшем и худшем случаях. Очевидно, что наименьшее число сравнений будет достигаться, если осевой элемент оказывается ровно в середине сортируемого участка. Таким образом, $B(n) = (n-1) + 2B(n/2)$. Решением данного рекуррентного соотношения, является функция $B(n) = O(n\log n)$.

Наихудший случай для числа сравнений получается тогда, когда осевой элемент разбивает массив на максимально неравные участки, т. е. когда осевой элемент оказывается либо минимальным, либо максимальным элементом сортируемого массива. В этом случае имеет место следующее рекуррентное соотношение $W(n) = (n-1) + W(n-1) + T(1)$. Решением данного рекуррентного соотношения, является функция $W(n) = O(n^2)$. Таким образом, в худшем случае алгоритм быстрой сортировки не является быстрым. Его трудоемкость сравнима с трудоемкостью квадратичных алгоритмов.

Помимо выбора среднего элемента в качестве осевого (т. е. элемента, находящегося в средней позиции), существует более эффективный подход, базирующийся на выборе «медианы из трех». Суть метода заключается в том, чтобы произвольным образом выбирать три элемента из сортируемого участка и в качестве осевого выбирать медиану среди выбранных элементов. Сам же Ч. Хоар предлагает выбирать осевой элемент случайно.

Одним из основных приложений алгоритма быстрой сортировки является нахождение k-ой порядковой статистики множества A, т. е. такого элемента из A, который после упорядочивания A будет находиться на k-ой позиции. К частным случаям рассматриваемой задачи можно отнести поиск минимального и максимального элементов, который в обоих случаях выполняется за $O(n)$ операций. В общем случае поиск k-го по величине элемента можно выполнить предварительно отсортировав массив A и затем взяв элемент, находящийся на

k-ой позиции. Тогда алгоритм затратит как в худшем, в среднем, так и в лучшем случаях $O(n \log n)$ шагов. Существует алгоритм, который в среднем случае выполняется за $O(n)$ шагов и идея которого базируется на идее алгоритма быстрой сортировки (процедура 3.2.9).

Процедура 3.2.9. Select

```
01:  Select Impl(A[1, n], k)
02:        pivot = f(A[random(1, n)])
03:        A1, A2 = []
04:        for i = 1 to n do
05:              if f(A[i]) < pivot then add A[i] into A1
06:              if f(A[j]) > pivot then add A[i] into A2
07:        if k ≤ |A1| then
08:              return Select Impl(A1, k)
09:        if k > n - |A2| then
10:              return Select Impl(A2, k – (n - |A2|))
11:        return pivot
```

Следует отметить, что существует усовершенствованная версия рассмотренного алгоритма, которая гарантирует время поиска $O(n)$ в худшем случае. Детали алгоритма приводятся в [4, 12].

В заключение рассмотрения алгоритма быстрой сортировки скажем о том, что существует нерекурсивная реализация алгоритма (3.2.8), базирующаяся на АТД «Стек». Такая реализация алгоритма находит применение на практике, поскольку позволяет избежать накладных расходов, связанных с рекурсивными вызовами. Помимо этого, алгоритм быстрой сортировки может быть реализован на многопроцессорной машине, т. к. рекурсивные вызовы сортируют не пересекающиеся между собой части исходного массива A.

Сортировка слиянием

Алгоритм *сортировки слиянием* основан на стратегии декомпозиции и использует технику «разделяй и властвуй». Впервые алгоритм сортировки слиянием был предложен Дж. фон Нейманом в 1945 г. [9]. Идея его состоит в следующем. Первый этап заключается в разбиении массива на две равные части. На втором этапе алгоритм рекурсивно применяется к каждой из полученных частей. Последний, третий этап алгоритма, сливает отсортированные на предыдущем шаге части в один результирующий массив (отсюда название «сортировка слиянием»). Следует отметить, что процедура слияния будет линейна по времени выполнения, т. е. требовать $O(n)$ операций (процедура 3.2.10):

Процедура 3.2.10. Merge Sort

```
01:  Merge Sort Impl(A)
02:        if |A| > 1 then
03:              L = A[1, [n / 2]]
04:              R = A[[n / 2] + 1, n]
05:              Merge Sort Impl(L)
06:              Merge Sort Impl(R)
07:              Merge(A, L, R)
```

```
08:  Merge(A, L, R)
09:        a = 1, l = 1, r = 1
10:        while l ≤ |L| and r ≤ |R| do
11:              if f(L[l]) < f(R[r]) then
12:                    A[a] = L[l], l = l + 1
13:              else
14:                    A[a] = R[r], r = r + 1
15:              a = a + 1
16:        A[a, n] = L[l, |L|]
17:        A[a, n] = R[r, |R|]
```

Проанализируем время работы приведенного алгоритма. Пусть длина исходного массива равна n. Как было отмечено выше, слияние двух отсортированных частей требует времени, которое линейно зависит от n. С учетом правила разделения на подзадачи, получаем следующее рекуррентное соотношение для времени работы алгоритма:

$$\begin{cases} T(1) = 0, \\ T(n) = 2T(n/2) + O(n), \ n > 1. \end{cases} \tag{3.8}$$

Решением данного соотношения является функция $T(n) = O(n \log_2 n)$, поэтому сложность алгоритма сортировки слиянием есть $O(n \log_2 n)$, где n – число элементов в A. В худшем случае алгоритм сортировки требует также $O(n \log_2 n)$ шагов. Делаем вывод, что сортировка слиянием является асимптотически лучше алгоритма быстрой сортировки. Следует отметить, что сортировка слиянием принадлежит к классу устойчивых алгоритмов сортировки, а также может быть реализована на многопроцессорной машине, поскольку рекурсивные вызовы сортируют не пересекающиеся между собой части L и R исходного массива A.

Сортировка слиянием, рассмотренная в том виде, в каком она приведена в процедуре (3.2.10) обладает одним серьезным недостатком, который заключается в выделении дополнительной памяти под массивы L и R, в результате чего затраты по памяти составляют $O(n)$ байт. Существует модификация алгоритма сортировки слиянием, сортирующая элементы массива на месте и базирующаяся на так называемой технике *sqrt-декомпозиции*. Существуют и более сложные методы, описание которых приводится в [9]. Общей идей, объединяющей все существующие подходы, является разделение отсортированных частей массива A на некоторое количество блоков $K_1, K_2, \dots K_m$, желательно одинакового размера, после чего происходит слияние соседних блоков между собой, возможно, с некоторым предварительным числом подготовительных операций, общее число которых не превышает $O(n)$.

Пирамидальная сортировка

Пирамидальная сортировка базируется на АТД «Очередь с приоритетом», реализованной посредством структуры данных бинарная куча, рассмотренной ранее в разд. 2.4. Таким образом, алгоритм пирамидальной сортировки может выглядеть следующим образом (процедура 3.2.11):

Процедура 3.2.11. Heap Sort Base

```
01:  Heap Sort Base Impl(A)
02:        priorityQueue queue(n, f)
03:        foreach a in A do
04:              queue.push(a)
05:        for i = 1 to n do
06:              A[i] = queue.top()
07:              queue.pop()
```

Трудоемкость представленного алгоритма есть $O(n \log n)$ в худшем случае, поэтому пирамидальная сортировка, как и алгоритм сортировки слиянием, считается асимптотически лучше алгоритма быстрой сортировки. Недостатком процедуры (3.2.11) является использование АТД «Очередь с приоритетами», в связи с чем затраты на память составят $O(n)$, где n – число сортируемых элементов. Можно попытаться свести затраты памяти к $O(1)$, но для этого следует отойти от использования очереди и реализовать бинарную кучу непосредственно на массиве A (процедура 3.2.12):

Процедура 3.2.12. Heap Sort

```
01:  Heap Sort Impl(A)
02:        for i = ⌊n / 2⌋ downto 1 do
03:              Push Down(A, i, n)
04:        for i = n downto 1 do
05:              swap(A[1], A[i])
06:              Push Down(A, 1, i – 1)
```

Следует отметить, что в реализации процедуры «проталкивания вниз» следует учесть то, что наверху должен оказаться элемент с наибольшим ключом, что в дальнейшем позволяет его ставить в корректную позицию. Важным замечанием является то, что в общем случае пирамидальная сортировка не принадлежит к классу устойчивых алгоритмов. Если же такое поведение потребуется, то функция f получения ключа должна быть изменена таким образом, чтобы учитывать начальную позицию элементов в исходном массиве.

Несмотря на то, что пирамидальная сортировка является асимптотически оптимальной, на практике рассмотренный алгоритм не получил широкого распространения в силу большой константы, скрытой в асимптотике O.

3.2.3. Линейная сортировка

Линейная сортировка выполняет упорядочивание множества A за фиксированное число просмотров всех элементов из A. Особенностью методов, требующих линейное число операций, является использование некоторых внутренних характеристик ключей, которые позволяют избавиться от операции сравнения. Единственным ограничением, которое накладывается на структуру ключей, является их конечное множество. Таким образом, ниже будут рассмотрены алгоритмы, трудоемкость которых есть $O(n)$ в худшем случае, где n – число сортируемых объектов.

Блочная сортировка

Блочная сортировка (или *карманная*) – алгоритм линейной сортировки, предназначенный для упорядочивания таких наборов данных, ключи которых принадлежат множеству B с заведомо известным числом элементов. В этом случае ожидаемое время работы алгоритма есть $O(n)$, т. е. линейно зависит от числа элементов в заданном наборе.

Рассмотрим основные принципы работы блочной сортировки. Оригинальная версия алгоритма базируется на том факте, что ключи элементов сортируемого множества являются вещественными числами, равномерно распределенными на интервале $[0,..,1)$. Далее алгоритм распределяет элементы множества A по карманам, каждый из которых представляет собой АТД «Список» либо другой АТД, в который вставка элементов осуществляется за $O(1)$. Далее элементы каждого из карманов сортируются каким-либо алгоритмом сортировки (например, сортировкой вставками). После того, как карманы отсортированы, происходит их слияние в результирующий массив. Предположив, что количество карманов равно n и функция распределения по карманам выглядит как $d(a) = \lfloor n \cdot f(a) \rfloor$, можно предложить следующую реализацию карманной сортировки (процедура 3.2.13):

Процедура 3.2.13. Bucket Sort

```
01:  Bucket Sort Impl(A)
02:        bucket[i].clear(), i∈[0, n)
03:        foreach a in A do
04:              bucket[i].push(⌊n·f(a)⌋)
05:        for i = 0 to n – 1 do
06:              sort(bucket[i])
07:        A = bucket[0] ∪ bucket[1] ∪ ... ∪ bucket[n – 1]
```

Проанализируем время работы приведенного алгоритма. Соответствующее рекуррентное соотношение будет иметь вид

$$T(n) = \sum_{i=0}^{n-1} O(n_i^2) + O(n), \tag{3.9}$$

где n_i – число элементов в кармане i.

В случае равномерного распределения ключей решением полученного рекуррентного соотношения является функция $T(n) = O(n)$ [4, 7]. Если же ключи элементов распределены не равномерно, то в худшем случае трудоемкость приведенного алгоритма будет составлять $O(n^2)$.

Следует отметить, что если вставка в карман осуществляется в конец списка и используемый алгоритм сортировки карманов является устойчивым, то и вся блочная сортировка в целом также будет принадлежать к классу устойчивых алгоритмов сортировки.

Одной из модификаций блочной сортировки является *сортировка подсчетом*. Простейшая версия сортировки подсчетом была рассмотрена ранее в разд. 3.1, процедура (3.1.1). Ниже приводится более совершенный алгоритм

сортировки подсчетом, который не учитывает тот факт, что сортируемые элементы суть числа, а базируется лишь на том, что ключи сортируемых элементов принадлежат множеству $[0,\ k)$ (процедура 3.2.14):

Процедура 3.2.14. Counting Sort

```
01:  Counting Sort Impl(A)
02:        C[i] = 0, i∈[0, k)
03:        foreach a in A do
04:              C[f(a)] = C[f(a)] + 1
05:        for key = 1 to k – 1 do
06:              C[key] = C[key] + C[key – 1]
07:        for i = n downto 1 do
08:              key = f(A[i])
09:              B[C[key]] = A[i]
10:              C[key] = C[key] – 1
11:        A = B
```

Время работы процедуры (3.2.14), как и процедуры (3.1.1), составляет $O(n + k)$ шагов. Обратим внимание на то, что заключительный проход по элементам массива A осуществляется в обратном порядке, что позволяет добиться устойчивого поведения алгоритма.

Поразрядная сортировка

Будем рассматривать множество элементов A и такое множество ключей B, каждый из которых может быть представлен в виде

$$b = b_d b_{d-1} \dots b_1, b \in B. \tag{3.10}$$

Будем говорить, что ключ элемента $f(a) = b$ состоит из d разрядов, причем значение в разряде i $(1 \le i \le d)$ ключа b может быть получено как $f_i(a)$. Также будем считать, что на множестве элементов B_i, соответствующего значениям, которые могут встречаться на позиции i-го разряда, задано отношение линейного порядка «меньше либо равно» и в каждом из множеств B_i существует нулевой элемент.

Поразрядная сортировка (или *цифровая*) – алгоритм сортировки, предназначенный для упорядочивания наборов данных, ключи которых принадлежат множеству B с описанными выше свойствами (процедура 3.2.15):

Процедура 3.2.15. Radix Sort

```
01:  Radix Sort Impl(A)
02:        for i = 1 to d do
03:              sort A using f_i:A → B_i key-function
```

Время работы поразрядной сортировки зависит от времени выполнения сортировки массива A по одному из ключей. Так, если вспомогательная процедура сортировки работает за линейное время $O(n)$, то приведенный алгоритм поразрядной сортировки будет иметь трудоемкость $O(d \cdot n)$.

Поразрядную сортировку принято разделять на два класса: сортировка от младших разрядов к старшим (LSD-сортировка) и сортировка от старших разрядов к младшим (MSD-сортировка)[*].

LSD-сортировка

Основным применением алгоритма LSD-сортировки является упорядочивание данных, длина ключей которых является одинаковой. Если же длины ключей оказались различны, то их всегда можно выровнять путем добавления нулевых элементов на позиции старших разрядов.

Как было отмечено выше, LSD-сортировка проводит упорядочивание данных, начиная от младших разрядов к старшим и, таким образом, придерживается следующего линейного порядка:

$$\forall b, c \in B | b \leq c \Leftrightarrow b_i = c_i \forall i = \overline{1, d} \vee \exists i : b_j = c_j, j < i \wedge b_i < c_i.$$

Если в качестве вспомогательной процедуры использовать сортировку подсчетом, то можно предложить следующую реализацию алгоритма (процедура 3.2.16):

Процедура 3.2.16. LSD Radix Sort

```
01:  LSD Radix Sort Impl(A)
02:         for i = 1 to d do
03:               Counting Sort(A, f_i)
```

В силу того, что сортировка подсчетом в худшем случае работает за $O(n + k)$ шагов, трудоемкость алгоритма LSD-сортировки есть $O(d \cdot (n + k))$.

Следует отметить, что использование LSD-сортировки оправдано в том случае, если число разрядов является не достаточно большим. Так, если требуется отсортировать n $(n \geq 10^5)$ 32-битных чисел, то предпочтительнее будет использование быстрой сортировки либо другой, работающей за время $O(n \log n)$ и базирующейся на непосредственном сравнений ключей. В том числе, некоторые из рассмотренных ранее логарифмических алгоритмов сортировки не требуют дополнительных затрат памяти, что также является несомненным преимуществом перед поразрядной сортировкой.

MSD-сортировка

Основным применением алгоритма MSD-сортировки является упорядочивание данных, длина ключей которых может быть различной. При этом для выравнивания длин ключей нулевые элементы добавляются на позиции младших разрядов, в отличие от LSD-сортировки. Таким образом, рассматриваемый алгоритм базируется на следующем линейном отношении порядка:

$$\forall b, c \in B | b \leq c \Leftrightarrow b_i = c_i \forall i = \overline{1, d} \vee \exists i : b_j = c_j, j > i \wedge b_i < c_i.$$

В этом случае будем говорить, что элемент b *лексикографически меньше либо равен* элементу c. Соответственно порядок данных, построенный с использованием данного отношения, будем называть *лексикографическим*.

[*] Термин LSD происходит от английского *least significant digit*. Термин MSD происходит от английского *most significant digit.*

Заметим, что в процессе поразрядной MSD-сортировки элементы исходного множества будут разбиваться на блоки, каждый из которых – последовательность соседних элементов с одинаковым префиксом ключей. Поэтому при переходе к упорядочиванию по следующему разряду, блоки не будут менять свои места, а будут лишь перемещаться элементы внутри каждого из блоков. Если в качестве вспомогательной процедуры использовать сортировку подсчетом, то можно предложить следующую рекурсивную реализацию алгоритма (процедура 3.2.17):

Процедура 3.2.17. MSD Radix Sort

```
01:  return MSD Radix Sort Impl(A, 1, n, d)
02:  MSD Radix Sort Impl(A, lo, hi, d)
03:        if (lo < hi) then
04:              C[i] = 0, i∈[0, k)
05:              foreach a in A[lo, hi] do
06:                    C[f_d(a)] = C[f_d(a)] + 1
07:              for key = 1 to k – 1 do
08:                    C[key] = C[key] + C[key – 1]
09:              T = C
10:              for i = hi downto lo do
11:                    key = f_d(A[i]),
12:                    B[C[key]] = A[i]
13:                    C[key] = C[key] – 1
14:              A[lo, hi] = B
15:              for key = 0 to k – 1 do
16:                    MSD Radix Sort Impl(A, lo, lo + T[key] - 1, d - 1)
17:                    lo = lo + T[key]
```

Трудоемкость алгоритма MSD-сортировки, как и LSD-алгоритма, есть $O(d \cdot (n + k))$, поскольку сортировка подсчетом линейна относительно числа элементов и количества ключей k.

Необходимо отметить, что алгоритм MSD-сортировки принадлежит к классу устойчивых алгоритмов, а также может быть реализован на многопроцессорной машине, т. к. рекурсивные вызовы сортируют не пересекающиеся между собой блоки исходного массива A.

3.3. Внешняя сортировка

Под *внешней сортировкой* будем понимать те алгоритмы сортировки или их реализации, использующиеся для упорядочивания данных, которые не помещаются в оперативную память ЭВМ в виде массива. Таким образом, алгоритмы внешней сортировки применяются для упорядочивания данных, хранящихся в файлах достаточно большого размера.

Любой из алгоритмов внешней сортировки базируется на алгоритмах внутренней сортировки. Отдельные части массива данных сортируются в оперативной памяти и с помощью специального алгоритма сцепляются в один массив, упорядоченный по ключу.

Наиболее популярным алгоритмом внешней сортировки является сортировка слиянием и ее модификации применительно к упорядочиванию файлов.

3.4. Задачи для самостоятельного решения

1. Рассмотрим перестановку P натуральных чисел от 1 до n и некоторый массив $A[n]$, на элементах которого задано отношение линейного порядка. Разработайте алгоритм, который упорядочит элементы массива A таким образом, что после применения перестановки P к массиву A k раз, элементы массива A будут идти в неубывающем порядке.

2. Рассмотрим n отрезков на прямой, каждый из которых характеризуется двумя координатами x_1 и x_2, обозначающими, соответственно, начало и конец отрезка. Разработайте алгоритм, который позволит найти объединение и пересечение всех отрезков.

3. Рассмотрим массив A из n $(1 \le n \le 10^6)$ натуральных чисел, не превышающих $50'000$. К массиву A применяется M операций, каждая из которых состоит из двух фаз: изменение элемента, находящегося в позиции i; возврат числа инверсий в массиве A. Разработайте алгоритм, который позволит эффективно отвечать на поступающие запросы. Рассмотрите случаи, когда запросы поступают в режиме *on-line* и *off-line* соответственно.

4. Рассмотрим два текста A и B, содержащие, соответственно, n и m символов. Будем предполагать, что каждый из символов текста встречается в нем ровно один раз. Будем говорить, что текст X (возможно, пустой) называется *общим подтекстом* текстов A и B, если X может быть получен вычеркиванием некоторых символов как из A, так и из B. *Наибольшим общим подтекстом* двух текстов называется их общий подтекст, имеющий максимально возможную длину. Требуется определить длину наибольшего общего подтекста заданных текстов A и B.

5. В нулевой момент времени мастеру одновременно поступает n работ, пронумерованных от 1 до n. Для каждой работы i заранее известна следующая информация: время выполнения работы t_i, кратное суткам; штраф p_i за каждые сутки ожидания работы i до момента начала ее выполнения. Одновременно может выполняться только одна работа, и, если мастер приступает к выполнению некоторой работы, то он продолжает выполнять ее, пока не закончит. Таким образом, суммарный штраф, который надо будет уплатить, равен сумме $\sum p_i \cdot s_i$, где s_i – время начала выполнения работы i. Требуется найти такой порядок выполнения работ, при котором штраф будет минимальным.

6. Рассмотрим n программ, которые нужно протестировать. Известно, что для тестирования программы с номером i требуется t_i дней. С другой стороны, тестирование программы с номером i должно быть завершено через d_i дней, начиная с сегодняшнего дня. Например, если тестирование надо завершить послезавтра, то $d_i = 2$. Разработайте алгоритм, который отвергнет минимальное число программ, чтобы не затянуть сроки тестирования оставшихся проектов, а также составит одну из возможных последовательностей их выполнения.

7. Рассмотрим n детей, для каждого из которых известно время, требуемое для того, чтобы уложить ребенка спать, и время, которое он будет находиться во сне. Задача состоит в том, чтобы уложить всех детей спать, причем

делать это разрешается строго последовательно. Разработайте алгоритм, который определит, в каком порядке должны будут засыпать дети, при этом максимизировав время между моментом, когда заснул последний, и проснулся первый из уже заснувших.

8. Разработайте алгоритм, который отсортирует элементы заданного стека S в неубывающем порядке. Предполагается, что детали реализации стека S неизвестны и доступны лишь операции, определяемые АТД «Стек».

9. Разработайте алгоритм, который отсортирует элементы заданного стека S в неубывающем порядке, используя операцию реверса, суть которой заключается в том, чтобы взять фиксированное число элементов с вершины стека и записать их в тот же стек в обратном порядке.

10. На прямой линии определенным образом расположено n отрезков. Из них нужно выбрать наибольшее количество непересекающихся между собой. Модифицируйте полученный алгоритм для задачи, в которой каждому из отрезков задан вес и требуется выбрать набор попарно непересекающихся отрезков с максимальным суммарным весом.

11. Рассмотрим массив A из n натуральных чисел. Обозначим через S следующую сумму: $\sum|A_i - A_{i+1}|$, где $1 \leq i < n$. Требуется найти такую перестановку элементов массива A, при которой сумма S была бы минимальной. Рассмотрите случай, когда разрешается переставлять все элементы массива A, за исключением элемента, находящегося в заданной позиции p $(1 \leq p \leq n)$.