Лекция 1.2
Алгоритмы сортировки

Сортировка

- сортировка — важная задача и сама по себе;
- ключ сортировки – это информация, которая сопоставляется с сортируемыми элементами и которая определяет порядок расположения элементов;

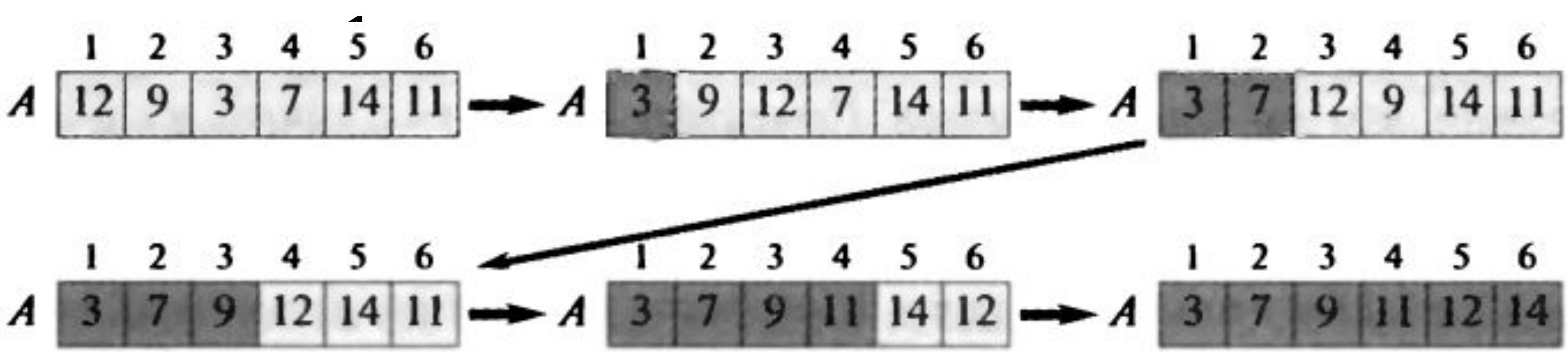
- **Задача**: разместить элементы в порядке возрастания

Рассмотрим четыре алгоритма сортировки массива:

- все они имеют время работы в худшем случае либо $\Theta(n^2)$, либо $\Theta(n\log_2 n)$;
- если требуется выполнить лишь один или несколько поисков, то лучше остановиться на линейном поиске;
- если нужно выполнять поиск много раз, то имеет смысл сначала отсортировать массив, а затем применять бинарный поиск.

Сортировка выбором

- Проходим по всему массиву, находим наименьший элемент и меняем этот элемент местами с первым элементом.
- Вновь проходим по массиву, начиная со второго элемента, находим наименьший элемент среди оставшихся и меняем этот элемент со вторым элементом массива.
- То же самое выполняется для третьего элемента и т.д.
- После того, как нужный элемент поставлен в по



Процедура Selection-Sort(A,n).

Вход:

- A – сортируемый массив.
- n – количество сортируемых элементов в массиве A.

Результат: элементы массива A отсортированы в неубывающем порядке.

Шаги процедуры:

1. Для i = 1 до n-1:
   A. Установить значение переменной smallest равным i.
   B. Для j = i+1 до n:
      i. Если A[j] < A[smallest] присваиваем переменной smallest значение j.
   C. Обменять A[i] ↔ A[smallest].

- Поиск наименьшего элемента в подмассиве A[i..*n*] представляет собой вариант линейного поиска.
- Наличие «вложенного цикла».
- Доказательство корректности можно провести с помощью двух инвариантов (по одному на каждый цикл)

а) «В начале каждой итерации цикла на шаге 1 подмассив A[1..i-1] содержит i-1 наименьших элементов массива в отсортированном порядке».

б) «В начале каждой итерации цикла на шаге 1В элемент A[smallest] представляет собой наименьший элемент в подмассиве A[i..j-1]».

Время работы сортировки выбором

1) На i-м шаге внешнего цикла внутренний цикл выполняется *n*-i раз.

2) Общее количество итераций внутреннего цикла равно сумме по всем итерациям внешнего цикла:

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

3) Следовательно, время работы сортировки выбором равно $\Theta(n^2)$ во всех случаях (если итерации выполняются за постоянное время).

Это медленный алгоритм:

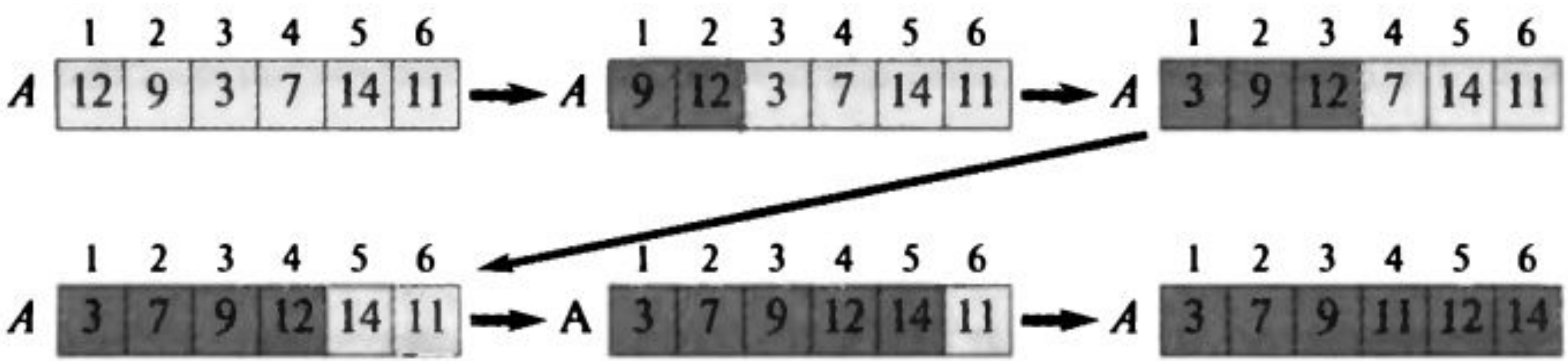
- Время $\Theta(n^2)$ обусловлено сравнениями элементов на каждой итерации.

- Количество обменов элементов массива равно только $\Theta(n)$.

Сортировка вставкой

- Сортировка ведется так, что элементы в первых i позициях — это те же элементы, которые были изначально в первых i позициях, но теперь отсортированные в правильном порядке (по возрастанию).
- Чтобы определить, куда надо вставить элемент, первоначально находившийся в A[i], сортировка вставкой проходит по подмассиву A[1..i-1] справа налево, начиная с элемента A[i-1], и переносит каждый элемент, больший, чем A[i] на одну позицию вправо.

• При обнаружении элемента, который не превышает A[i], или перемещения до левого конца массива, элемент, изначально находившийся в A[i], переносится в его новую позицию в массиве.



Процедура Insertion-Sort(A,n).

Вход и результат: те же, что и в Selection-Sort.

Шаги процедуры:

1. Для i = 1 до n-1:
   A. Установить переменную key равной A[i], а переменной j присвоить значение i-1.
   B. Пока j > 0 и A[j] > key, выполнять следующее:
      i. Присвоить A [j+1] значение A [j].
      ii. Уменьшить j на единицу (присвоить переменной j значение j-1).
   C. Присвоить A[j + 1] значение key.

Время работы сортировки вставкой

- Количество итераций внутреннего цикла зависит как от индекса i внешнего цикла, так и от значений элементов массива.
- *Наилучший случай*: массив A уже отсортирован (внутренний цикл выполняет нуль итераций). Тогда итерации внешнего цикла выполняются *n*-1 раз и процедура занимает время Θ(*n*) (считаем, что каждая итерация внешнего цикла выполняется за постоянное время).

- *Наихудший случай*: массив А отсортирован в обратном порядке (внутренний цикл делает максимально возможное количество итераций). Тогда внешний цикл каждый раз выполняет итерации внутреннего цикла i-1 раз. Итог тот же, как и при сортировке выбором: время работы Θ(n^2).
- В *среднем* каждый элемент будет больше около половины предшествующих ему элементов и меньше тоже около половины этих элементов, что сократит время работы по сравнению с наихудшим случаем в два раза, т.е. время работы останется Θ (n^2).
- Сортировка вставкой может перемещать элементы до Θ(n^2) раз.
- Сортировка вставкой лучше, если массив почти отсортирован.

Сортировка слиянием

Парадигма *«разделяй и властвуй»*

1) *Разделение*. Задача разбивается на несколько подзадач, которые представляют собой меньшие экземпляры той же самой задачи.
2) *Властвование*. Рекурсивно решаются подзадачи. Если они достаточно малы, они решаются как базовый случай.
3) *Объединение*. Решения подзадач объединяются в решение исходной задачи.

Разделяем сортируемый подмассив путем нахождения значения q посредине между p и r: $q = \lfloor (p+r)/2 \rfloor$

Рекурсивно сортируем элементы в каждой половине подмассива, созданной на шаге разделения (от p до q и от q+1 до r).

Объединение отсортированных элементов в промежутках от p до q и от q+1 до r так, чтобы элементы в промежутке от p-го до r-го были отсортированы.

Процедура Merge-Sort(A,p,r).

Вход: A – массив, p, r – начальный и конечный индексы подмассива A.

Результат: элементы подмассива A[p..r] отсортированы в неубывающем порядке.

Шаги процедуры:

1. Если p≥r, подмассив A[p..r] содержит не более одного элемента, так что он автоматически является отсортированным. Выполняем возврат из процедуры без каких-либо действий.

2. В противном случае выполняем следующие действия: $q = \lfloor (p+r)/2 \rfloor$

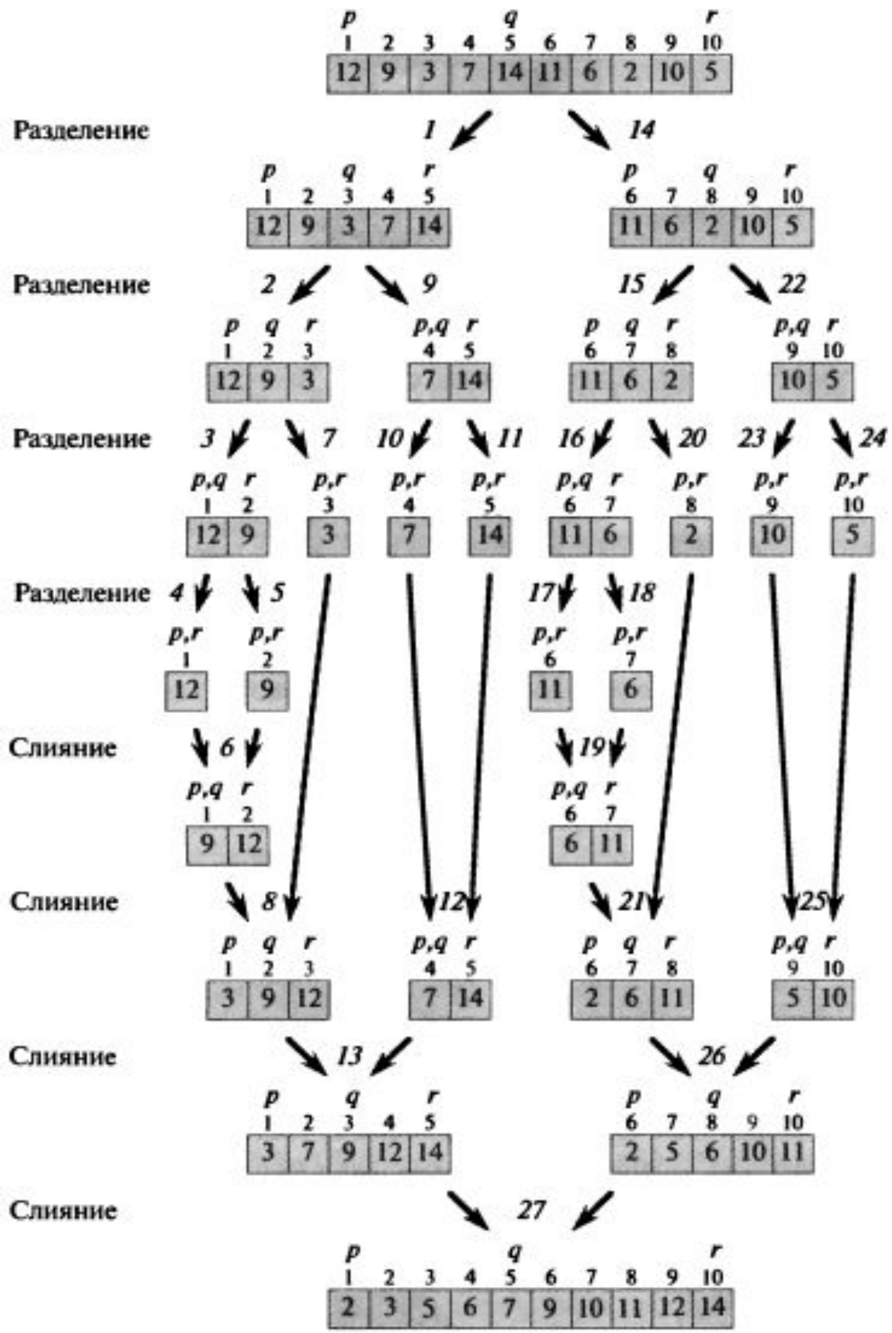
   A. Установить

   B. Рекурсивно вызвать Merge-Sort(A,p,q).

   C. Рекурсивно вызвать Merge-Sort(A,q+1,r).

   D. Вызвать Merge(A,p,q,r).

Пример:
Merge-Sort(A,1,10)



Слияние не может осуществляться без привлечения дополнительной памяти.

1) Пусть n_1 = q-p+1 — количество элементов в A[p..q], а n_2 = r-q — количество элементов в A[q+1..r]. Создадим временные массивы B с n_1 элементами и C с n_2 элементами и скопируем элементы из A[p..q], не нарушая их порядок, в массив B, а элементы из A[q+1..r] — в массив C.

2) Копируем элементы массивов B и C обратно в подмассив A[p..r], последовательно сравнивая элементы из массивов B и C и копируя минимальный из

3) Чтобы не проверять каждый раз, не исчерпался ли полностью один из массивов, разместим в правом конце массивов B и C дополнительный элемент, который заведомо больше любого другого элемента, т.е. что-то типа ограничителя. Когда все элементы из массивов B и C скопированы обратно в исходный массив, в них в качестве наименьших элементов остаются ограничители, которые не попадают в исходный массив.

Процедура Merge(A,p,q,r).

Вход: A – массив, p, q, r – индексы в массиве A. Подмассивы A[p..q] и A[q+1..r] считаются уже отсортированными.

Результат: отсортированный подмассив A[p..r], содержащий все элементы, изначально находившиеся в подмассивах A[p..q] и A[q+1..r].

Шаги процедуры:

1. Установить n_1 равным q-p+1, а n_2 — равным r-q.
2. B[1..n_1 +1] и C[1..n_2+1] представляют собой новые массивы.
3. Скопировать A[p..q] в B[1..n_1], а A[q+1..r] — в C [1..n_2].

4. Установить B[n_1 +1] и C[n_2+1] равными ∞.
5. Установить i и j равными 1.
6. Для k = p до r:
A. Если B[i] ≤ C[j], установить A[k] равным
B[i] и увеличить i на 1.
B. В противном случае (B[i] > C[j])
установить A[k] равным C[j] и
увеличить j на 1.

Время работы: Θ(n)

Время работы сортировки слиянием

- Для простоты положим, что размер массива n представляет собой степень 2, так что каждый раз, когда мы делим массив пополам, размеры подмассивов равны.
- Время сортировки $T(n)$ состоит из трех компонентов:

1) Разделение занимает константное время, поскольку состоит только в вычислении индекса q.

2) Властвование состоит из двух рекурсивных вызовов для подмассивов, каждый размером $n/2$ элементов, что занимает время $2T(n/2)$.

3) Объединение результатов двух рекурсивных вызовов с помощью слияния отсортированных подмассивов выполняется за время $\Theta(n)$.

$$\boxed{T(n)=2T(n/2)+\Theta(n)}$$

Результат решения этого рекуррентного уравнения:

$T(n)$ имеет вид $\Theta(n\log_2 n)$.

Сравнение алгоритмов сортировки

Плюсы сортировки слиянием:

-- С точки зрения времени работы сортировка слиянием [$\Theta(n\log_2 n)$] однозначно выгодна по сравнению с наихудшим временем работы $\Theta(n^2)$ у алгоритмов сортировки выбором и сортировки вставкой.

Минусы сортировки слиянием:

-- Требуется дополнительная память: сортировка делает полные копии всего входного массива. Если вопрос использования памяти приоритетен, использовать сортировку слиянием нельзя.

Быстрая сортировка

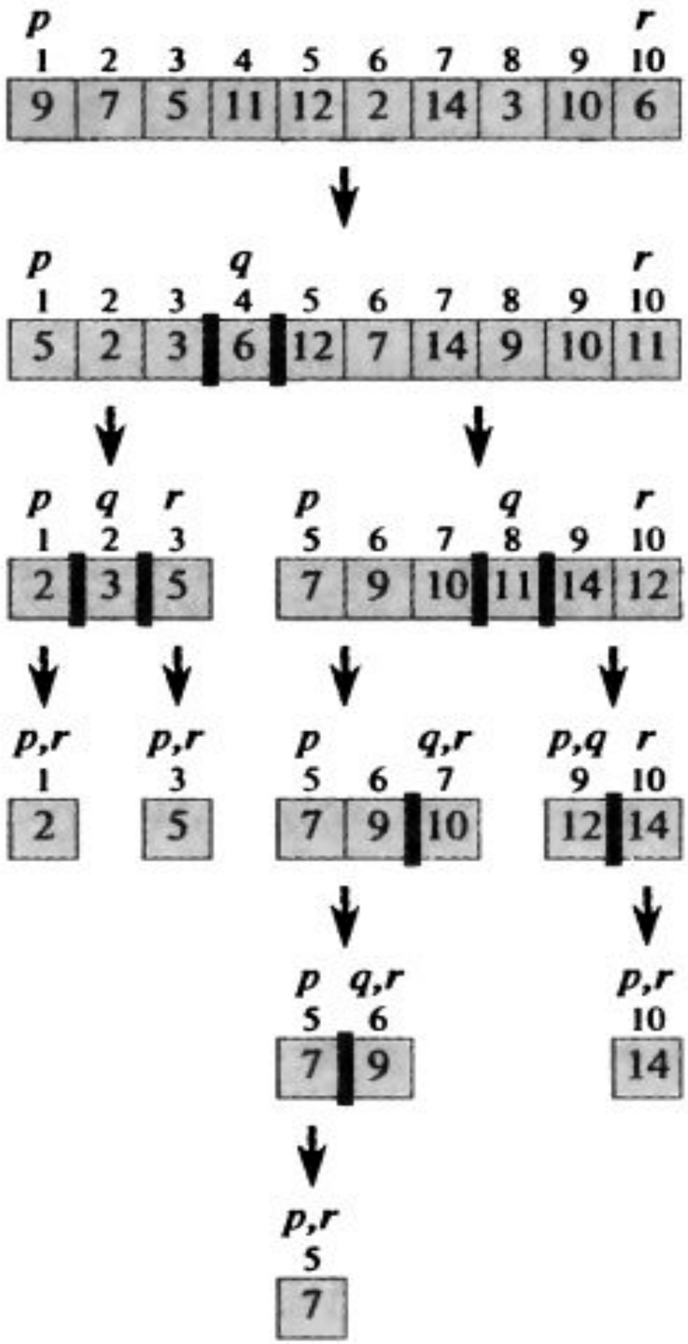
Как и в сортировке слиянием, используется парадигма «разделяй и властвуй».

Существенные отличия:

а) Быстрая сортировка работает "на месте", без привлечения дополнительной памяти;

б) Асимптотическое время работы быстрой сортировки для среднего случая отличается от времени работы для наихудшего случая;

в) Хороший постоянный множитель (лучше, чем у сортировки слиянием), так что на практике чаще всего предпочтение отдается быстрой сортировке.



Выберем один элемент и назовем его опорным. Поместим все элементы, меньшие опорного, слева, а элементы, большие опорного, справа от этого элемента. Если опорный элемент находится в позиции q, то далее рекурсивно сортируются элементы в положениях с p до q-1 и с q+1 до r. Эта рекурсия и дает полностью отсортированный массив.
На примере в качестве опорного элемента используется последний элемент каждого подмассива.

Самое нижнее значение в каждой позиции массива показывает, какой элемент будет находиться в этой позиции по завершении сортировки.

Процедура Quicksort(A,p,r).

Вход и результат: те же, что и у процедуры Merge-Sort.

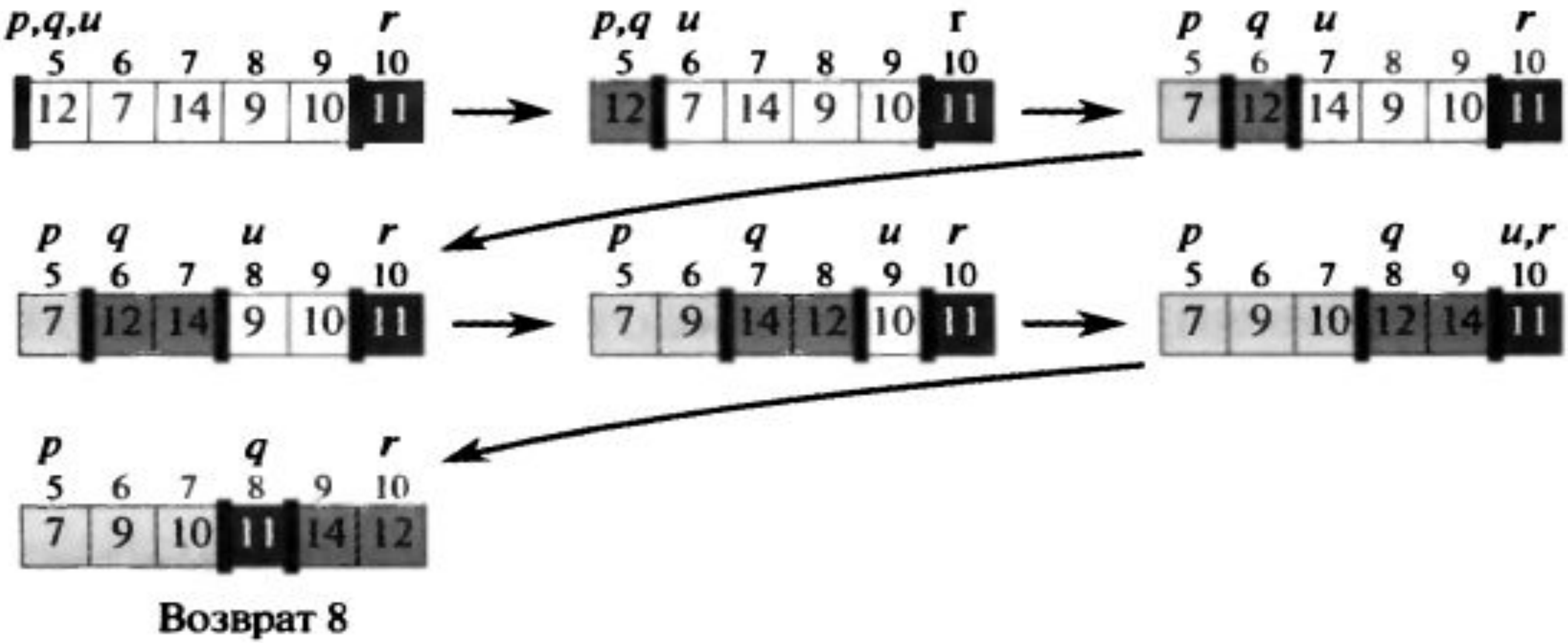
Шаги процедуры:

1. Если p ≥ r, просто выйти из процедуры, не выполняя никаких действий.
2. В противном случае выполнить следующее:
   A. Вызвать Partition(A,p,r) и установить значение q равным результату вызова.
   B. Рекурсивно вызвать Quicksort(A,p,q-1).
   C. Рекурсивно вызвать Quicksort(A,q+1,r).

- Базовый случай осуществляется, когда сортируемый подмассив содержит менее двух элементов.

- Процедура Partition(A,p,r) разбивает подмассив A[p..r] и возвращает индекс q позиции, в которую помещается опорный элемент.

Процедура разбиения



- Выбираем в подмассиве A[p..r] крайний справа элемент A[r] в качестве опорного.
- Затем мы проходим через подмассив слева направо, сравнивая каждый элемент с опорным.

Процедура Partition(A,p,r).

Вход: тот же, что и для Merge-Sort.

Результат: перестановка элементов A[p..r], такая, что каждый элемент в A[p..q-1] не превышает A[q], а каждый элемент в A[q+1..r] больше A[q]. Возвращает значение индекса q.

Шаги процедуры:

1. Установить q равным p.
2. Для u = p до r-1:
   A. Если A[u] ≤ A[r], обменять A[q] с A[u], а затем увеличить q на 1.
3. Обменять A[q] и A[r], а затем вернуть q.

Время работы быстрой сортировки

- Выполняется по одному сравнению каждого элемента с опорным и не более одного обмена для каждого элемента, так что **время работы процедуры разбиения** с n-элементным подмассивом равно $\Theta(n)$.
- В **наихудшем случае** размеры разделов являются несбалансированными. Например, если массив изначально отсортирован, мы всякий раз будем разбивать массив A[p..r] на подмассивы A[p..r-1] и A[r]. Тогда для времени сортировки подмассива из n элементов получаем рекуррентное соотношение $T(n)=T(n-1)+ \Theta(n)$. Оказывается, что в этом случае $T(n)$ имеет вид $\Theta(n^2)$

- В **наилучшем случае**, если всякий раз каждый из подмассивов будет иметь размер $n/2$, то рекуррентное соотношение для времени работы такое же, как для сортировки слиянием: $T(n)=2T(n/2)+\Theta(n)$, а значит, $T(n)$ имеет вид $\Theta(n\log_2 n)$.
- Если элементы входного массива располагаются **в случайном порядке**, то в среднем получаем разделения, достаточно близкие к разбиениям пополам, так что быстрая сортировка имеет при этом время работы $\Theta(n\log_2 n)$.

- Чтобы повысить шансы на получение хороших разбиений, можно выбирать опорные элементы случайным образом.
- Следует также оценить, сколько раз процедура Quicksort обменивает элементы. Наибольшее количество обменов осуществляется, когда n четно и входной массив имеет вид n, n-2, n-4,...,4,2,1,3,5,..., n-3, n-1. В этом случае выполняется $n^2/4$ обменов, и асимптотическое время работы алгоритма соответствует наихудшему случаю $\Theta(n^2)$.

Резюме

Алгоритмы поиска

Алгоритм	Время работы в наихудшем случае	Время работы в наилучшем случае	Требует ли отсортированного входного массива
Линейный поиск	$\Theta(n)$	$\Theta(1)$	Нет
Бинарный поиск	$\Theta(\lg n)$	$\Theta(1)$	Да

Алгоритмы сортировки

Алгоритм сортировки	Время работы в наихудшем случае	Время работы в наилучшем случае	Обменов в наихудшем случае	Выполняется ли сортировка на месте
Выбором	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Да
Вставкой	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Да
Слиянием	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	Нет
Быстрая	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n^2)$	Да

Можно ли превзойти время сортировки $\Theta(n\log_2 n)$?

НЕТ

Если единственный способ определения порядка размещения элементов – это их сравнение

Сортировки сравнением (определяет порядок сортировки только путем сравнения пар элементов)

$\Omega(n\log_2 n)$

экзистенциальная нижняя граница (т.к. существуют такие входные данные)

ДА

Если имеется дополнительная информация о сортируемых элементах

$\Omega(n)$

универсальная нижняя граница (т.к. применима ко всем входным данным)

Простая сортировка за время Θ(n)

- Предположим, что каждый ключ сортировки является либо единицей, либо двойкой.

- Пройдем по всем элементам и подсчитаем, сколько среди них единиц (k).

- Установим значение 1 в первых k позициях массива и значение 2 в остальных n-k позициях.

Процедура Really-Simple-Sort(A,n).

Вход:

- A – массив, все элементы которого имеют значения 1 или 2,
- n – количество сортируемых элементов A.

Результат: элементы A отсортированы в неубывающем порядке.

Шаги процедуры:

1. Установить k равным нулю.
2. Для i = 1 до n:
   A. Если A[i] = 1, увеличить k на единицу.
3. Для i = 1 до k:
   A. Установить A[i] равным 1.
4. Для i = k + 1 до n:
   A. Установить A[i] равным 2.

- Алгоритм никогда не сравнивает два элемента массива один с другим: он сравнивает каждый элемент массива со значением 1, но не с другим элементом массива.

- Процедура выполняется за время $\Theta(n)$, т.к. первый цикл выполняет n итераций, как и два последних цикла вместе.

- Т.о. если есть дополнительная информация об элементах массива, можно превзойти алгоритмы сортировки сравнением.

Сортировка подсчетом

- Обобщение на случай m различных возможных значений ключей сортировки, которые являются, скажем, целыми числами от 0 до m-1.
- *Идея*: если мы знаем, что у k элементов ключи сортировки равны x, а у l элементов ключи сортировки меньше x, то элементы с ключами сортировки, равными x, в отсортированном массиве должны занимать позиции от l+1 до l+k.

- *Надо*: для каждого возможного значения ключа сортировки вычислить, у какого количества элементов ключи сортировки меньше этого значения (значение l) и сколько имеется элементов с данным значением ключа сортировки (значение k).

Разобъем на подзадачи

1) Вычислим, у какого количества элементов ключи сортировки равны заданному значению

Процедура Count-Keys-Equal(A,n,m).

Вход:

- A – массив целых чисел в диапазоне от 0 до m-1,
- n – количество элементов в массиве A,
- m – определяет диапазон значений в массиве A.

Выход: массив equal[0..m-1], такой, что equal[j] содержит количество элементов массива A, равных j, для j = 0,1,2,...,m-1.

Шаги процедуры:

1. Пусть equal[0..m-1] представляет собой новый массив.

2. Установить все значения массива equal равными нулю.
3. Для i=1 до n:
    A. Установить значение переменной key равным A[i].
    B. Увеличить equal[key] на единицу.
4. Вернуть массив equal.

Время работы: $\Theta(n+m)$

2) Выясним, у какого количества элементов ключи сортировки меньше каждого возможного значения

Процедура Count-Keys-Less(equal,m).

Вход:

- equal – массив, возвращаемый вызовом процедуры Count-Keys-Equal,
- m – определяет диапазон индексов массива equal – от 0 до m-1.

Выход: масив less[0..m-1], такой, что для j = 0,1,2,...,m-1 элемент less[j] содержит сумму equal[0] + equal[1] + … + equal[j-1].

Шаги процедуры:

1. Пусть less[0..m-1] представляет собой новый массив.
2. Установить less[0] равным нулю.
3. Для j = 1 до m-1:
   A. Установить less[j] равным less[j-1] + equal[j-1].
4. Вернуть массив less.

Время работы: Θ(m)

3) Создадим отсортированный массив путем перемещения элементов из массива A в массив B так, чтобы они в конечном итоге оказались в массиве B в отсортированном порядке

Процедура Rearrange(A,less,n,m).

Вход:

- A – массив целых чисел в диапазоне от 0 до m-1,
- less – массив, возвращаемый процедурой Count-Keys-Less,
- n – количество элементов в массиве A,
- m – определяет диапазон значений элементов в массиве A.

Выход: массив B, содержащий элементы массива A в отсортированном порядке.

Шаги процедуры:

1. Пусть B[1..n] и next[0..m-1] – новые массивы.
2. Для j = 0 до m-1:
   A. Установить next[j] равным less[j] + 1.
3. Для i = 1 до n:
   A. Установить значение key равным A[i].
   B. Установить значение index равным next[key].
   C. Установить B[index] равным A[i].
   D. Увеличить значение next[key] на единицу.

Время работы: Θ ($m+n$)

- Вспомогательный массив next[j] указывает индекс элемента в массиве B, в который должен быть помещен очередной элемент массива A с ключом j. Этот индекс первоначально равен next[j] = less[j] +1 и с каждым найденным элементом с ключом j должен быть увеличен на 1.
- Цикл на шаге 2 выполняется за время Θ(m), а цикл на шаге 3 — за время Θ(n). Следовательно, процедура Rearrange имеет время работы Θ($m+n$).

4) Собираем все три процедуры вместе для создания окончательной процедуры сортировки подсчетом

Процедура Counting-Sort(A,n,m).

Вход:

- A – массив целых чисел в диапазоне от 0 до m-1,
- n – количество элементов в массиве A,
- m – определяет диапазон значений в массиве A.

Выход: массив B, содержащий элементы массива A в отсортированном порядке.

Шаги процедуры:

1. Вызвать процедуру Count-Keys-Equal(A,n,m) и сохранить ее результат как массив equal.
2. Вызвать процедуру Count-Keys-Less(equal,m) и сохранить ее результат как массив less.
3. Вызвать процедуру Rearrange(A,less,n,m) и сохранить ее результат как массив B.
4. Вернуть массив B.

Время работы сортировки подсчетом

- Исходя из времени работы процедур Count-Keys-Equal ($\Theta(m+n)$), Count-Keys-Less ($\Theta(m)$) и Rearrange ($\Theta(m+n)$), получаем, что процедура Counting-Sort выполняется за время $\Theta(m+n)$, или просто $\Theta(n)$, если m представляет собой константу.
- Сортировка подсчетом превосходит нижнюю границу $\Omega(n\log_2 n)$ сортировки сравнением, потому что она никогда не сравнивает ключи сортировки один с другим.

- Ключи сортировки используются для *индексирования массивов,* что вполне реально, когда ключи сортировки являются небольшими целыми значениями.

- Если ключи сортировки представляют собой действительные числа или, например, строки символов, то использовать сортировку подсчетом *нельзя*.

Устойчивость сортировки

Сортировка подсчетом имеет еще одно важное свойство. Она является устойчивой: элементы с одним и тем же ключом сортировки оказываются в выходном массиве в том же порядке, что и во входном. Другими словами, устойчивая сортировка, встречая два элемента с равными ключами, разрешает неоднозначность, помещая в выходной массив первым тот элемент, который появляется первым во входном массиве.

Поразрядная сортировка

- Используется сортировка подсчетом и ее свойство устойчивости.
- Предполагается, что каждый ключ сортировки можно рассматривать как d-значное число, каждая цифра которого находится в диапазоне от 0 до *m*-1.
- Поочередно используется устойчивая сортировка (например, сортировка подсчетом) для каждой цифры *справа налево*. Порядок сортировки цифр или символов <u>действительно важен</u>.

Пример поразрядной сортировки

Нужно отсортировать по алфавиту и по возрастанию двухсимвольные коды <F6, E5, R6, X6, X2, T5, F2, T3>.

1) Сортируем подсчетом по правому символу: <X2, F2, T3, E5, T5, F6, R6, X6>. В силу устойчивости после сортировки X2 продолжает находиться перед F2.

2) Сортируем результат подсчетом по левому символу и получим то, что и требуется: <E5, F2, F6, R6, T3, T5, X2, X6>.

Если начать сортировку слева направо, то после сортировки подсчетом по левому символу получили бы <E5, F6, F2, R6, T5, T3, X6, X2>, а затем после сортировки подсчетом по правому символу получили бы неверный результат <F2, X2, T3, E5, T5, F6, R6, X6>.

Время работы поразрядной сортировки

- Если в качестве устойчивой применяется сортировка подсчетом, то время сортировки по одной цифре составляет $\Theta(m+n)$, а время сортировки по всем d цифрам — $\Theta(d(m+n))$.
- Если m является константой, то время работы поразрядной сортировки становится равным $\Theta(dn)$.
- Если d также представляет собой константу, то время работы поразрядной сортировки превращается в просто $\Theta(n)$.
- Причина: поразрядная сортировка никогда не сравнивает два ключа сортировки один с другим, а использует отдельные цифры для индексирования массивов в сортировке подсчетом.