

Элементарные алгоритмы сортировки

Почему так важно?

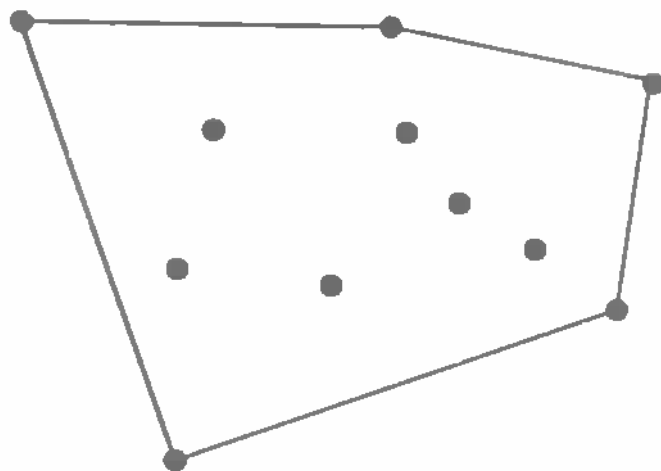
- › **Сортировка** – строительный блок, на котором основаны многие алгоритмы
- › **Сортировка** – породила большинство интересных идей, включая метод «разделяй и властвуй», структуры данных и пр.
- › **Сортировка** – то, на что уходит больше компьютерного времени, чем на остальные задачи
- › **Сортировка** – самая изученная задача в теории вычислительных систем

Сортировка – базовый блок в решении других задач

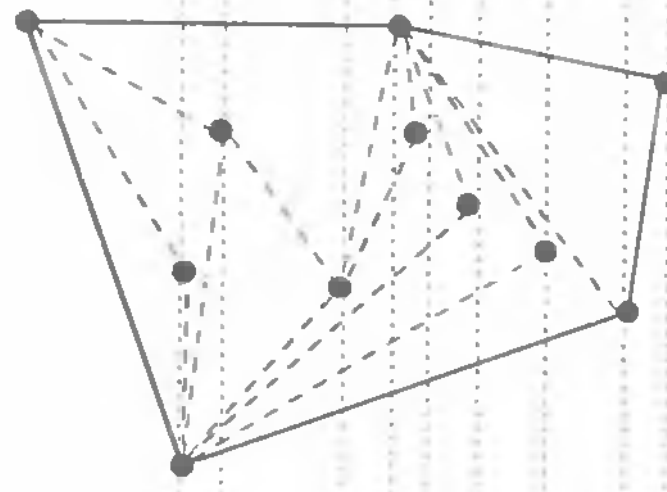
- › **Поиск** – если данные отсортированы, то поиск двоичный, время $O(\log n)$
- › **Поиск ближайшей пары** – как найти в множестве из n чисел два числа с наименьшей разницей? $O(n \log n)$
- › **Определение уникальности элементов** – как определить, имеются ли дубликаты в множестве из n элементов? Частный случай предыдущего
- › **Частотное распределение** – поиск самого часто встречающегося элемента

Сортировка – базовый блок в решении других задач (2)

- › **Выбор элемента** – как найти k -й по величине элемент массива? Если есть сортировка, то линейное время
- › **Выпуклая оболочка** – как определить многоугольник с наименьшей поверхностью?



a)



б)

Сортировка является центральной частью многих алгоритмов.

Сортировка данных должна быть одним из первых шагов, предпринимаемых любым разработчиком алгоритмов с целью повышения эффективности разрабатываемого решения

Задача сортировки

Перестановка элементов оригинальной последовательности для удовлетворения некоторому условию (порядка).

$$A: a_1, a_2, \dots a_n$$

$$\circ: A \rightarrow B$$

$$B: b_1, b_2, \dots b_n$$

$$b_i \circ b_j \quad \forall i, j, i \leq j$$

Сравниваемость элементов

Для любых двух элементов p и q из набора данных должен выполняться ровно один из следующих трех предикатов:

$$p=q, p<q \text{ или } p>q$$

Распространенные сортируемые типы-примитивы:

- › Целые числа
- › Числа с плавающей запятой
- › Символы

<https://wordsonline.ru/dicts/orthography.html>

Сравниваемость элементов (2)

Вопрос упорядочения не прост

- › «Б» больше или меньше «б»?
- › Диакритические знаки – как соотносятся «ò», «ó», «ô» и «õ»?
- › Дифтонги – «а» и «æ»?

Unicode – <http://www.unicode.org/versions/latest>
(на сегодня (21.09.2023) 149813 символов – для представления каждого возможно до 4 байт)

Компаратор – функция сравнения cmp , которая сравнивает элементы p и q и возвращает 0, -, +

Практические аспекты сортировки

- › *Сортировать в возрастающем или убывающем порядке?*
- › *Сортировать только ключ или все поля записи?*
При сортировке набора данных необходимо сохранять целостность сложных записей.
- › *Что делать в случае совпадения ключей?*
Элементы с одинаковыми значениями ключей будут сгруппированы вместе при любом порядке сортировки, но иногда имеет значение порядок размещения их относительно друг друга
- › *Как поступать с нечисловыми данными?*

Устойчивая сортировка

Определение

Говорят, что метод сортировки **устойчив**, если он сохраняет относительный порядок размещения элементов в файле, который содержит дублированные ключи.

Соображения к выбору алгоритма сортировки

- › Если число сортируемых элементов не очень большое, можно воспользоваться простым методом
- › Сложные алгоритмы в общем случае обуславливают непроизводительные затраты ресурсов, а это приводит к тому, что на файлах небольших размеров они работают медленнее элементарных методов сортировки
- › Другими типами файлов, сортировка которых существенно упрощена, являются файлы с почти (или уже) завершенной сортировкой или файлы, которые содержат большое число дублированных ключей

Методы сортировки

- › **Внутренние** – если сортируемый файл полностью помещается в оперативной памяти.
В этом случае доступ к любому элементу не представляет трудностей
- › **Внешние** – если сортируемый файл находится на внешнем носителе: магнитной ленте, диске и пр.
В этом случае возможен только последовательный метод доступа или, по меньшей мере, доступ к блокам больших размеров.

Массивы – Связанные списки

Факторы, влияющие на выбор алгоритмов сортировки

- › *Время выполнения*
- › *Дополнительный объем используемой памяти*
- › *Сложность алгоритма*
- › *Объем кода*
- › *Производительность в разных случаях*
- › *Особенности сортируемого набора*

Фактор времени выполнения

Время исполнения алгоритмов сортировки

$$O(n \log n) - O(n^2)$$

n	$n^2/4$	$n \log n$
10	25	33
100	2 500	664
1 000	250 000	9 965
10 000	25 000 000	132 877
100 000	2 500 000 000	1 660 960

Фактор дополнительного объема используемой памяти

Категории методов:

- › Выполняют сортировку на месте и не нуждаются в дополнительной памяти
- › Используют представление в виде связанного списка, используя указатели или индексы массива – нужна доппамять для их размещения
- › Требуют дополнительной памяти для размещения еще одной копии массива, подвергаемого сортировке.

Фактор сложности алгоритма

Цель: разработка эффективных и недорогих реализаций алгоритмов.

Реализация:

- › Избегать неоправданных расширений внутренних циклов алгоритмов
- › Поиск путей удаления всевозможных команд из внутренних циклов алгоритмов
- › Переключение на более эффективный алгоритм

Критерии выбора алгоритма сортировки

Критерий	Алгоритм сортировки
Только несколько элементов	Сортировка вставками
Элементы уже почти отсортированы	Сортировка вставками
Важна производительность в наихудшем случае	Пирамидальная сортировка
Важна производительность в среднем случае	Быстрая сортировка
Элементы с равномерным распределением	Блочная сортировка
Как можно меньше кода	Сортировка вставками
Требуется устойчивость сортировки	Сортировка слиянием

Пирамидальная сортировка (HeapSort)

Дж. Уильямс 1964

Метод сортировки сравнением, основанный на такой структуре данных как *двоичная куча*.

Двоичная куча — это законченное двоичное дерево, в котором элементы хранятся в особом порядке: значение в родительском узле больше (или меньше) значений в его двух дочерних узлах.

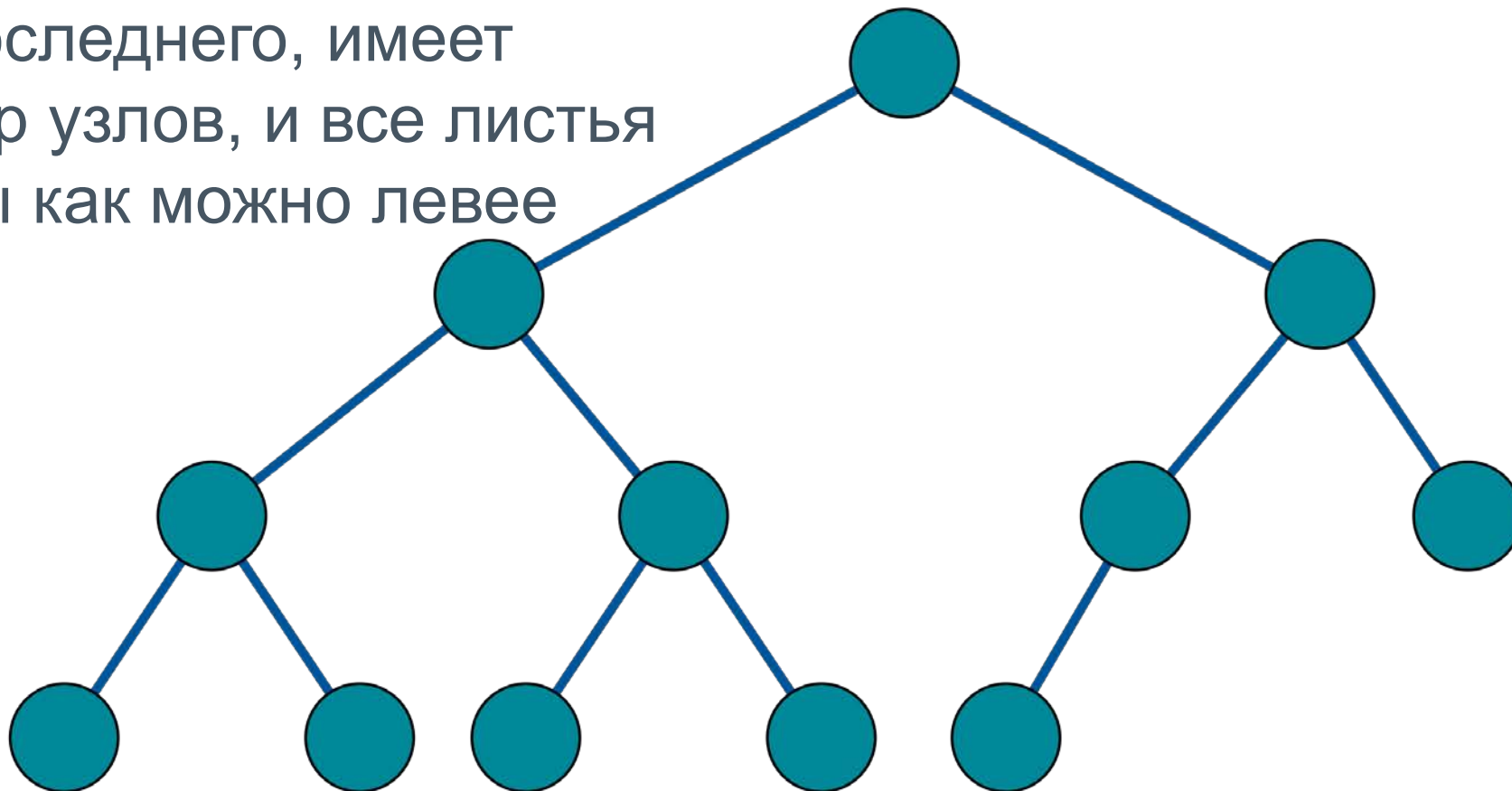
Первый вариант – **max-heap**

Второй вариант – **min-heap**

Куча может быть представлена двоичным деревом или массивом.

Законченное двоичное дерево

Двоичное дерево, в котором каждый уровень, за исключением, возможно, последнего, имеет полный набор узлов, и все листья расположены как можно левее



Почему для двоичной кучи используется представление на основе массива ?

Поскольку двоичная куча — это законченное двоичное дерево, ее можно легко представить в виде массива, а представление на основе массива является эффективным с точки зрения расхода памяти.

Если родительский узел хранится в индексе I , *левый дочерний элемент* может быть вычислен как $2I + 1$, а *правый дочерний элемент* — как $2I + 2$ (при условии, что индексирование начинается с 0).

Почему для двоичной кучи используется представление на основе массива ?

Поскольку двоичная куча — это законченное двоичное дерево, ее можно легко представить в виде массива, а представление на основе массива является эффективным с точки зрения расхода памяти.

Если родительский узел хранится в индексе I , *левый дочерний элемент* может быть вычислен как $2I + 1$, а *правый дочерний элемент* — как $2I + 2$ (при условии, что индексирование начинается с 0).

Алгоритм пирамидальной сортировки (в порядке по возрастанию)

- › Постройте max-heap из входных данных.
- › На данном этапе самый большой элемент хранится в корне кучи. Замените его на последний элемент кучи, а затем уменьшите ее размер на 1. Наконец, преобразуйте полученное дерево в max-heap с новым корнем.
- › Повторяйте вышеуказанные шаги, пока размер кучи больше 1.

Алгоритм пирамидальной сортировки (резюме)

Достоинства

- › Имеет доказанную оценку худшего случая $O(n \cdot \log n)$.
- › Сортирует на месте, то есть требуется всего $O(1)$ дополнительной памяти

Алгоритм пирамидальной сортировки (резюме)

Недостатки

- › Неустойчив.
- › Нет разницы в почти отсортированном или хаотичном массиве
- › На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти
- › Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа.
- › Не распараллеливается

Сортировка выбором

Основная идея

- › В неотсортированном массиве ищется локальный максимум (минимум).
- › Найденный максимум (минимум) меняется местами с последним (первым) элементом в рассматриваемом массиве.
- › Если в массиве остались неотсортированная последовательность, переходим к пункту 1.

π

Сортировка выбором (примеры)

8 9 6 5 4 8 9 <u>1</u> 5 3	ИНДУСТРИА <u>Л</u> ИЗАЦИЯ
1 8 9 6 5 4 8 9 5 <u>3</u>	А ИНДУСТРИЛИЗ <u>А</u> ЦИЯ
1 3 8 9 6 5 <u>4</u> 8 9 5	АА ИН <u>Д</u> УСТРИЛИЗЦИЯ
1 3 4 8 9 6 <u>5</u> 8 9 5	ААД И <u>Н</u> УСТРИЛИЗ <u>Ц</u> ИЯ
1 3 4 5 8 9 6 8 9 <u>5</u>	ААДЗ <u>И</u> НУСТРИЛИЦИЯ
1 3 4 5 5 8 9 <u>6</u> 8 9	ААДЗИ НУСТРИ <u>Л</u> ИЦИЯ
1 3 4 5 5 6 <u>8</u> 9 8 9	ААДЗИИ НУСТР <u>Л</u> ИЦИЯ
1 3 4 5 5 6 8 <u>9</u> 8 9	ААДЗИИИ НУСТР <u>Л</u> ЦИЯ
1 3 4 5 5 6 8 8 <u>9</u> 9	ААДЗИИИИ НУСТР <u>Л</u> ЦЯ
1 3 4 5 5 6 8 8 9 <u>9</u>	ААДЗИИИИЛ <u>Н</u> УСТРЦЯ
1 3 4 5 5 6 8 8 9 9	ААДЗИИИИЛН УСТР <u>Ц</u> Я
	ААДЗИИИИЛНР У <u>С</u> ТЦЯ
	ААДЗИИИИЛНРС У <u>Т</u> ЦЯ
	ААДЗИИИИЛНРСТ У <u>Ц</u> Я

Сортировка выбором (псевдокод)

Для $i = 0, n-2$

$\text{min} = i$

 Для $j = i+1, n-1$

 Если $(A[j] < A[\text{min}])$ $\text{min} = j$

 Если $(\text{min} \neq i)$

 переставить $(A[j], A[\text{min}])$

Сортировка выбором ?????? вопросы ?????

1. Трудоемкость?
2. Модификация?
3. Устойчивость?
4. Затраты памяти?
5. Для каких массивов?

Сортировка выбором ?????? вопросы ?????

1. Трудоемкость — $N^2/2$ — операций сравнения, N — операций обмена элементов местами
2. Модификация — минимум/максимум
3. Устойчивость — м.б. любым
4. Затраты памяти — нет дополнительных затрат
5. Для каких массивов — не чувствительна к природе входных данных

Сортировка вставками

Основная идея

Все элементы входной последовательности рассматриваются по одному, каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов

π

Сортировка вставками (примеры)

8 <u>9</u> 6 5 4 8 9 1 5 3	И <u>Н</u> ДУСТРИАЛИЗАЦИЯ
8 9 <u>6</u> 5 4 8 9 1 5 3	ИН <u>Д</u> УСТРИАЛИЗАЦИЯ
6 8 9 <u>5</u> 4 8 9 1 5 3	ДИН <u>У</u> СТРИАЛИЗАЦИЯ
5 6 8 9 <u>4</u> 8 9 1 5 3	ДИНУ <u>С</u> ТРИАЛИЗАЦИЯ
4 5 6 8 9 <u>8</u> 9 1 5 3	ДИНСУ <u>Т</u> РИАЛИЗАЦИЯ
4 5 6 8 8 9 <u>9</u> 1 5 3	ДИНСТУ <u>Р</u> ИАЛИЗАЦИЯ
4 5 6 8 8 9 9 <u>1</u> 5 3	ДИНРСТУ <u>И</u> АЛИЗАЦИЯ
1 4 5 6 8 8 9 9 <u>5</u> 3	ДИИНРСТУ <u>А</u> ЛИЗАЦИЯ
1 4 5 5 6 8 8 9 9 <u>3</u>	АДИИНРСТУ <u>Л</u> ИЗАЦИЯ
1 3 4 5 5 6 8 8 9 9	АДИИЛНРСТУ <u>И</u> ЗАЦИЯ
	АДИИИЛНРСТУ <u>З</u> АЦИЯ
	АДЗИИИЛНРСТУ <u>А</u> ЦИЯ
	ААДЗИИИЛНРСТУ <u>Ц</u> ИЯ

Сортировка вставками (псевдокод)

$A[0] = -\infty$

Для $i = 2, n$

$j = i$

До тех пор, пока $(A[j] < A[j-1])$

 переставить $(A[j], A[j-1])$

$j = j - 1$

Сортировка вставками ?????? вопросы ?????

1. Трудоемкость?
2. Модификация?
3. Устойчивость?
4. Затраты памяти?
5. Для каких массивов?

Сортировка вставками

?????? вопросы ?????

1. Трудоемкость — $N^2/4$ — операций сравнения, $N^2/4$ — операций полубмена элементов местами (перемещений) и в два раза больше операций в наихудшем случае
2. Модификация — использование бинарного поиска
3. Устойчивость — да
4. Затраты памяти — нет дополнительных затрат
5. Для каких массивов — хороша для частично отсортированного массива

π

Пузырьковая сортировка

Основная идея

Проход по файлу с обменом местами соседних элементов, нарушающих заданный порядок

π

Сортировка пузырьком (примеры)

8 9 6 5 4 8 9 1 5 3	И <u>Н</u> ДУСТРИАЛИЗАЦИЯ
8 6 9 5 4 8 9 1 5 3	ИДНУСТРИАЛИЗАЦИЯ
8 6 5 9 4 8 9 1 5 3	ИДНСУТРИАЛИЗАЦИЯ
8 6 5 4 9 8 9 1 5 3	ИДНСТУРИАЛИЗАЦИЯ
8 6 5 4 8 9 9 1 5 3	ИДНСТРУИАЛИЗАЦИЯ
8 6 5 4 8 9 1 9 5 3	ИДНСТРИУАЛИЗАЦИЯ
8 6 5 4 8 9 1 5 9 3	ИДНСТРИАУЛИЗАЦИЯ
8 6 5 4 8 9 1 5 3 9	ИДНСТРИАЛУИЗАЦИЯ
6 8 5 4 8 9 1 5 3 9	ИДНСТРИАЛИУЗАЦИЯ
6 5 8 4 8 9 1 5 3 9	ИДНСТРИАЛИЗУАЦИЯ
6 5 4 8 8 9 1 5 3 9	ИДНСТРИАЛИЗАУИЦЯ
6 5 4 8 8 1 9 5 3 9	ИДНСТРИАЛИЗАИУЦЯ
6 5 4 8 8 1 5 9 3 9	ДИНСРТИАЛИЗАИУЦЯ
6 5 4 8 8 1 5 3 9 9	ДИНСРИТАЛИЗАИУЦЯ

Сортировка пузырьком (псевдокод)

Для $i = 0, n-2$

Для $j = 0, n-i-2$

Если $(A[j] > A[j+1])$

переставить $(A[j], A[j+1])$

Сортировка пузырьком ?????? вопросы ?????

1. Трудоемкость?
2. Модификация?
3. Устойчивость?
4. Затраты памяти?
5. Для каких массивов?

Сортировка пузырьком

?????? вопросы ?????

1. Трудоемкость — $N^2/2$ — операций сравнения, $N^2/2$ — операций обмена как в среднем, так и в наихудшем случаях
2. Модификация — шейкерная, чет-нечетная, расческа
3. Устойчивость — да
4. Затраты памяти — нет дополнительных затрат
5. Для каких массивов — хороша для частично отсортированного массива

Шейкерная сортировка – развитие пузырьковой сортировки

Основная идея

Начало – пузырьковая сортировка с «выдавливанием» максимального значения вправо. Далее – разворот на 180° и «выдавливание» минимального элемента влево. Далее проходы до остановки в середине.

Чет-нечетная сортировка (N.Haberman, 1972)

Основная идея

Модификация пузырьковой сортировки, когда сравниваются элементы массива с четными и нечетными индексами, а на следующем шаге нечетные с четными.

Останов, когда после двух проходов по массиву не переставлено ни одного элемента

Чет-нечетная сортировка (N.Haberman, 1972)

Основная идея (продолжение)

Нечетные итерации

$(a_1, a_2), (a_3, a_4), (a_5, a_6), \dots, (a_{n-1}, a_n)$ при четном n

Четные итерации

$(a_2, a_3), (a_4, a_5), (a_6, a_7), \dots, (a_{n-2}, a_{n-1})$

π

Чет-нечетная сортировка (примеры)

8 9 6 5 4 8 9 1 5 3	И <u>Н</u> ДУСТРИАЛИЗАЦИЯ
8 9 5 6 4 8 1 9 3 5	ИНДУСТИРАЛЗИАЦИЯ
8 5 9 4 6 1 8 3 9 5	ИДНСУИТАРЗЛАИИЦЯ
5 8 4 9 1 6 3 8 5 9	ДИНСИУАТЗРАЛИИЦЯ
5 4 8 1 9 3 6 5 8 9	ДИНИСАУЗТАРИЛИЦЯ
4 5 1 8 3 9 5 6 8 9	ДИИНАСЗУАТИРИЛЦЯ
4 1 5 3 8 5 9 6 8 9	ДИИАНЗСАУИТИРЛЦЯ
1 4 3 5 5 8 6 9 8 9	ДИАИЗНАСИУИТЛРЦЯ
1 3 4 5 5 6 8 8 9 9	ДАИЗИАНИСИУЛТРЦЯ
	АДЗИАИИНИСЛУРТЦЯ
	АДЗАИИИИНЛСРУТЦЯ
	АДАЗИИИИЛНРСТУЦЯ
	ААДЗИИИИЛНРСТУЦЯ

π

Чет-нечетная сортировка (примеры)

8 9 6 5 4 8 9 1 5 3

8 9 5 6 4 8 1 9 3 5

8 5 9 4 6 1 8 3 9 5

5 8 4 9 1 6 3 8 5 9

5 4 8 1 9 3 6 5 8 9

4 5 1 8 3 9 5 6 8 9

4 1 5 3 8 5 9 6 8 9

1 4 3 5 5 8 6 9 8 9

1 3 4 5 5 6 8 8 9 9

1 3 4 5 5 6 8 8 9 9

1 3 5 4 5 6 8 8 9 9

1 5 3 5 4 6 8 9 8 9

1 5 5 3 6 4 9 8 9 8

1 5 5 6 3 9 4 9 8 8

1 5 5 6 9 3 9 4 8 8

1 5 5 9 6 9 3 4 8 8

1 5 9 5 6 9 4 3 8 8

1 5 9 5 6 9 4 8 3 8

Чет-нечетная сортировка (псевдокод)

Для $i = 1, n-1$

Если $(i \% 2 == 1)$ // нечетная итерация

Для $j = 0, n/2 - 3$

сравнить-переставить($A[2*j+1]$, $A[2*j+2]$)

Если $(i \% 2 == 1)$ // сравнение последней пары

сравнить-переставить($A[n-2]$, $A[n-1]$)

Иначе // четная итерация

Для $j = 1, n/2 - 2$

сравнить-переставить($A[2*j]$, $A[2*j+1]$)

Чет-нечетная сортировка ?????? вопросы ?????

1. Трудоемкость?
2. Модификация?
3. Устойчивость?
4. Затраты памяти?
5. Для каких массивов?

Параллельная модификация

№ и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 нечет (1, 2), (3, 4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 чет (2, 3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
3 нечет (1, 2), (3, 4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 чет (2, 3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Чет-нечетная сортировка ?????? вопросы ?????

1. Трудоемкость — $O(N^2)$ — наихудшая сложность, $O(N \log N)$ — средняя сложность, $O(N)$ — наилучший вариант
2. Модификация — параллельная реализация
3. Устойчивость — да
4. Затраты памяти — нет дополнительных затрат
5. Для каких массивов — хороша для частично отсортированного массива

Сортировка расческой (Dobosiewicz, 1980)

Основная идея

Сравнивать не соседние элементы, а на разном расстоянии. Сначала на достаточно большом расстоянии, а по мере упорядочивания сужать расстояние до минимума

Сортировка расческой

Реализация

1. Расстояние между элементами максимальное (n-1)
2. Следующие проходы совершаются с предыдущим шагом, поделенном на фактор уменьшения
3. Последний проход, когда разница индексов 1

Оптимальный фактор уменьшения

$$\frac{1}{1 - e^{-\varphi}} = 1.24733 \dots$$

где φ - золотое сечение

Сортировка расческой ?????? вопросы ?????

1. Трудоемкость — $O(N^2)$ — наихудшее время, $O(N)$ — лучшее время, $\Omega(N^2/2^p)$ — среднее время
2. Модификация — параллельная реализация
3. Устойчивость — да
4. Затраты памяти — нет дополнительных затрат
5. Для каких массивов — ????

Сортировка Шелла (*Donald L. Shell, 1959*)

Основная идея



Модифицировать сортировку вставками, применив идею сортировки расческой

Алгоритм сортировки Шелла

Каждый проход в алгоритме характеризуется смещением h_i , таким, что сортируются элементы отстающие друг от друга на h_i позиций.

h -упорядоченные файлы

i начинается с t и заканчивается 0 по определенному правилу $h_0=1$

Алгоритм сортировки Шелла

Шаг 0. $i = t$

Шаг 1. Разобьем массив на списки элементов, отстающих друг от друга на h_i . Таких списков будет h_i

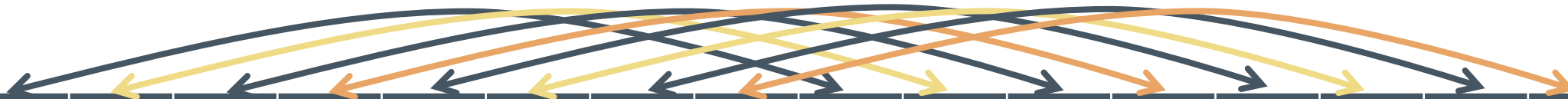
Шаг 2. Отсортируем элементы каждого списка сортировкой вставки

Шаг 3. Объединим списки обратно в массив. Уменьшим i . Если i не отрицательно – вернемся к шагу 1.

π

Пример

8 групп по 2 элемента



12	8	14	6	4	9	1	8	13	5	11	3	18	3	10	9
12	5	11	3	4	3	1	8	13	8	14	6	18	9	10	9
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
4	3	1	3	12	5	10	6	13	8	11	8	18	9	14	9
1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
4	3	1	3	12	5	10	6	13	8	11	8	18	9	14	9
1	3	4	3	10	5	11	6	12	8	13	8	14	9	18	9
1	3	3	4	5	6	8	8	9	9	10	11	12	13	14	18

// Последовательный алгоритм сортировки Шелла

```
ShellSort ( double A[], int n ){  
    int incr = n/2;  
    while( incr > 0 ) {  
        for ( int i=incr+1; i<n; i++ ) {  
            j = i-incr;  
            while ( j > 0 )  
                if ( A[j] > A[j+incr] ){  
                    swap(A[j], A[j+incr]);  
                    j = j - incr;  
                }  
            else j = 0;  
        }  
        incr = incr/2;  
    }  
}
```


π

Какова оптимальная последовательность?

Классика – последовательность шагов Шелла

Первый интервал между элементами равен длине массива. На каждом следующем шаге интервал уменьшается вдвое. Вычислительная сложность в худшем случае $O(n^2)$

1 2 4 8 16 32 64 128 256 512 1024 2048 ...

Другие последовательности

Последовательность Хиббарда $(2^k - 1) - O(n^{3/2})$

1 3 7 15 31 63 127 255 511 1023

Последовательность Седжвика – $O(n^{4/3})$

1 5 19 41 109 209 505 929 2161 3905

Последовательность Пратта $(2^i 3^j) - O(n \log^2 n)$

1 2 3 4 6 9 8 12 18 27 16 24 36 54 81....

Последовательность Циура – $O(n \log n)$

1 4 10 23 57 132 301 701 1750

π

Направления для творчества 😊

$$h_i = ah_{i-1} + kh_{i-2}$$

Оптимальные пары {a,k}?

a=3, k=1/3

a=4, k=1/4

a=4, k=-1/5

π

Эмпирические исследования

N	O	K	G	S	P	I
12500	16	6	6	5	6	6
25000	37	13	11	12	15	10
50000	102	31	30	27	38	26
100000	303	77	60	63	81	58
200000	817	178	137	139	180	126

Ключи

O 1 2 4 8 16 32 64 128 256 512 1024 2048 ...

K 1 4 13 40 121 364 1093 3280 9841 ...

G 1 2 4 10 23 52 113 249 548 1207 2655 5843 ...

S 1 8 23 77 281 1073 4193 16577 ...

P 1 7 8 49 56 64 343 392 448 512 2401 2744 ...

I 1 5 19 41 109 209 505 929 2161 3905 ...

Сортировка других типов данных

Массивы

Списки

Сортировка индексов и указателей

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
static char buf[100000];
static int cnt = 0;
int operator<(const Item& a, const Item& b)
    { return strcmp(a.str, b.str) < 0; }
void show(const Item& x)
    { cout << x.str << " "; }
int scan(Item& x)
    { int flag = (cin >> (x.str = &buf[cnt])) != 0;
      cnt += strlen(x.str)+1;
      return flag;
    }
```

Сортировка по индексам

Работая с индексами, а не с самими записями, можно упорядочить массив одновременно по нескольким ключам

0	10	9	Wilson	63
1	4	2	Johnson	86
2	5	1	Jones	87
3	6	0	Smith	90
4	8	4	Washington	84
5	7	8	Thompson	65
6	2	3	Brown	82
7	3	10	Jackson	61
8	9	6	White	76
9	0	5	Adams	86
10	1	7	Black	71

Данные могут быть

1. фамилии студентов и их оценками,
2. результат индексной сортировки по именам
3. результат индексной сортировки по убыванию оценок.

Сортировка по указателям

Пример:

qsort из стандартной библиотеки C

она представляет собой сортировку по указателям, которая принимает функцию сравнения в качестве аргумента

В обычных приложениях указатели используются для доступа к записям, которые могут иметь несколько ключей.

Причины использования индексов или указателей

Не затрагивать данные, подвергаемые сортировке

Можно «сортировать» файл даже в том случае, когда к нему разрешен доступ «только для чтения»

Можно избежать расходов на перемещения записей целиком

Достигается значительная экономия, если записи большие (а ключи маленькие)

Сортировка связанных списков

***Программы должны менять только
связи в узлах и не должны менять
ключей или какой-либо другой
информации***

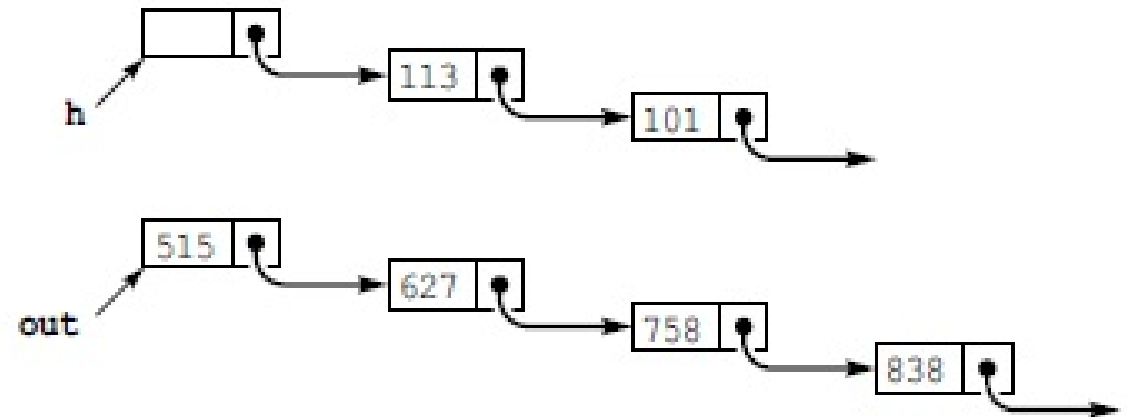
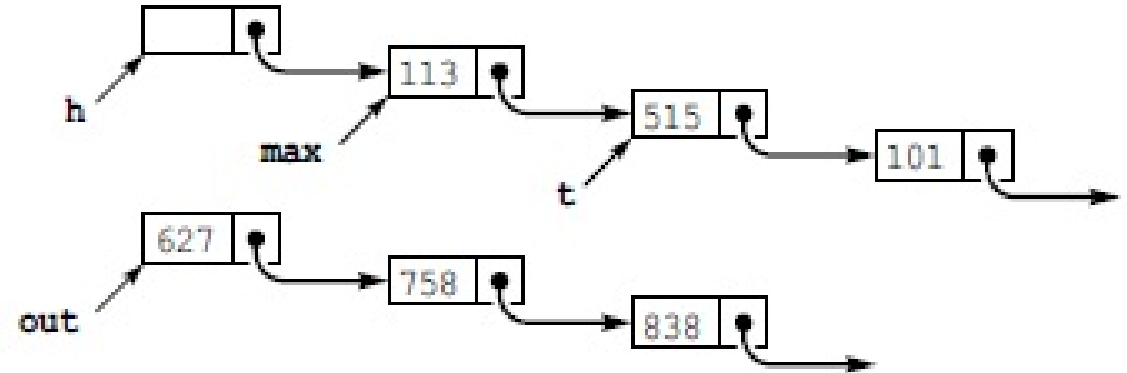
Сортировка выбором для связанных списков

Имеется:

1. Входной список
(хранятся исходные
данные)

2. Выходной список
(фиксируются результаты
сортировки)

Входной список просматривается с целью обнаружения max элемента, который потом удаляется из списка и помещается в начало выходного списка



Сортировка выбором для связанных списков

```
link listselection(link h)
{
    node dummy(0); link head = &dummy, out = 0;
    head->next = h;
    while (head->next != 0)
    {
        link max = findmax(head), t = max->next;
        max->next = t->next;
        t->next = out; out = t;
    }
    return out;
}
```

Метод распределяющего подсчета

Повышение эффективности сортировки за счет использования специфических ключей

Задача

Требуется выполнить сортировку файла из N элементов, ключи которых принимают различные значения в диапазоне от 0 до $N - 1$.

```
for (i = 0; i < N; i++) b[key(a[i])] = a[i];
```

π

Метод распределяющего подсчета

```
for (i = 0; i < N; i++)  
    b[cnt[a[i]]++] = a[i];
```

Обобщение

Отсортировать файл, состоящий из N элементов, ключи которого принимают целые значения в диапазоне от 0 до $M - 1$

Эффективен для не очень больших M

π

Описание метода

Исходная последовательность
из n структур

хранится в массиве A ,

отсортированная — в массиве B того же размера.

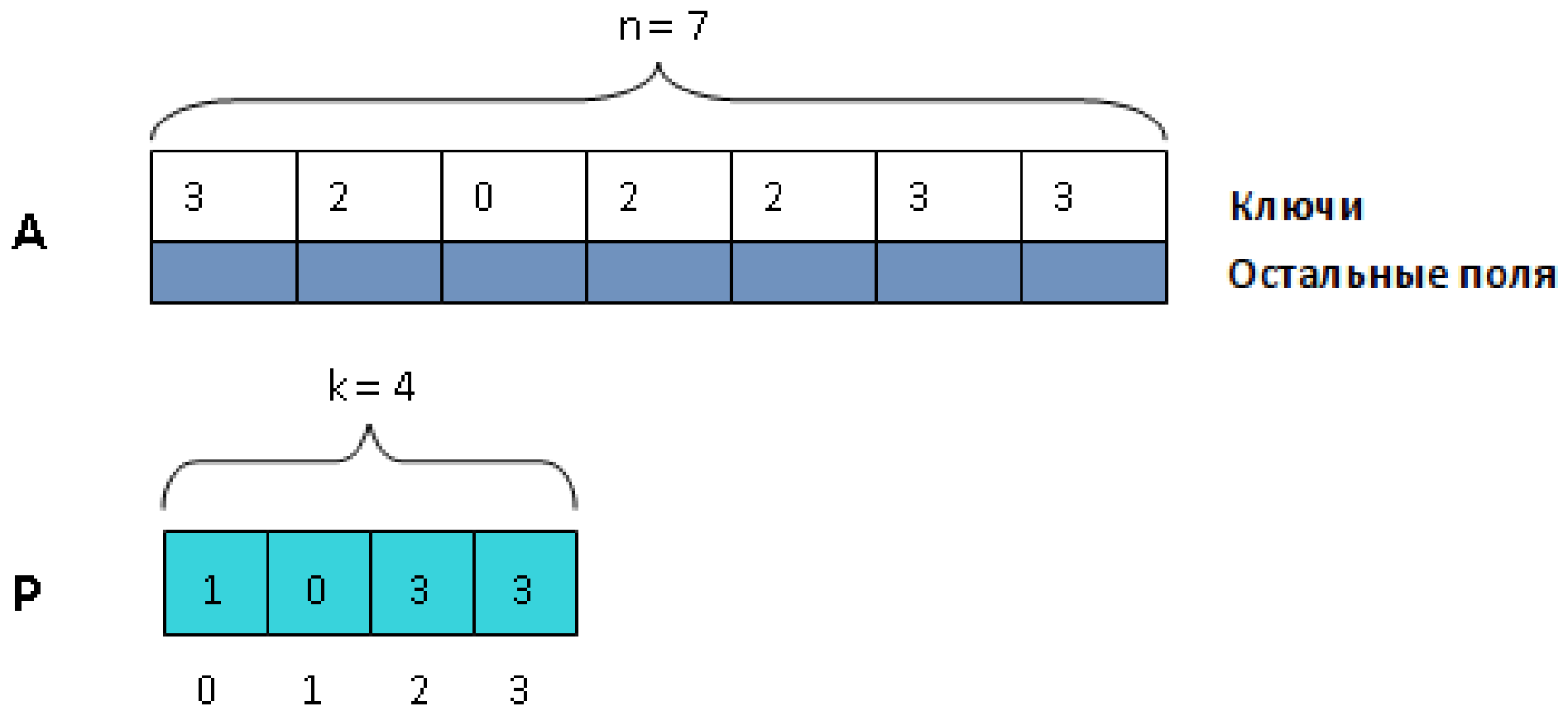
Кроме того, используется вспомогательный массив P
с индексами от 0 до $k-1$

Идея метода

1. Производим предварительный подсчет количества элементов с различными ключами в исходном массиве и разделении результирующего массива на части соответствующей длины (будем называть их блоками)
2. При повторном проходе исходного массива каждый его элемент копируется в специально отведенный его ключу блок, в первую свободную ячейку. Это осуществляется с помощью массива индексов P , в котором хранятся индексы начала блоков для различных ключей.
 $P[key]$ индекс в результирующем массиве, соответствующий первому элементу блока для ключа key

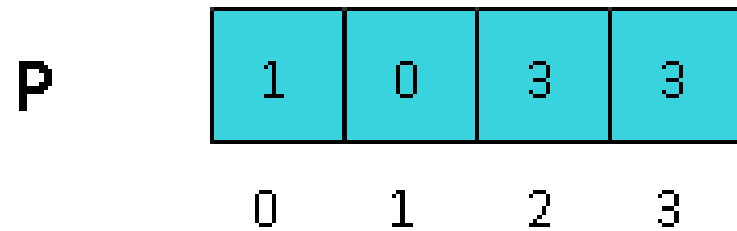
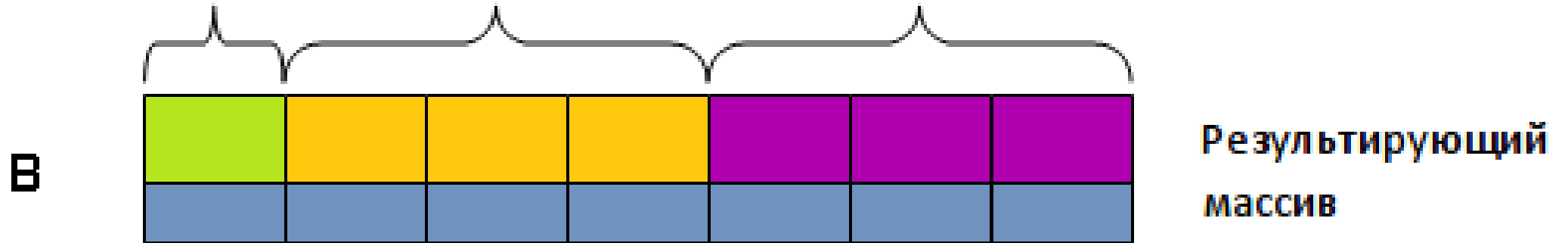
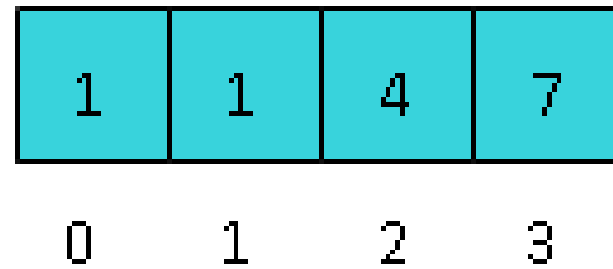
π

Первый шаг



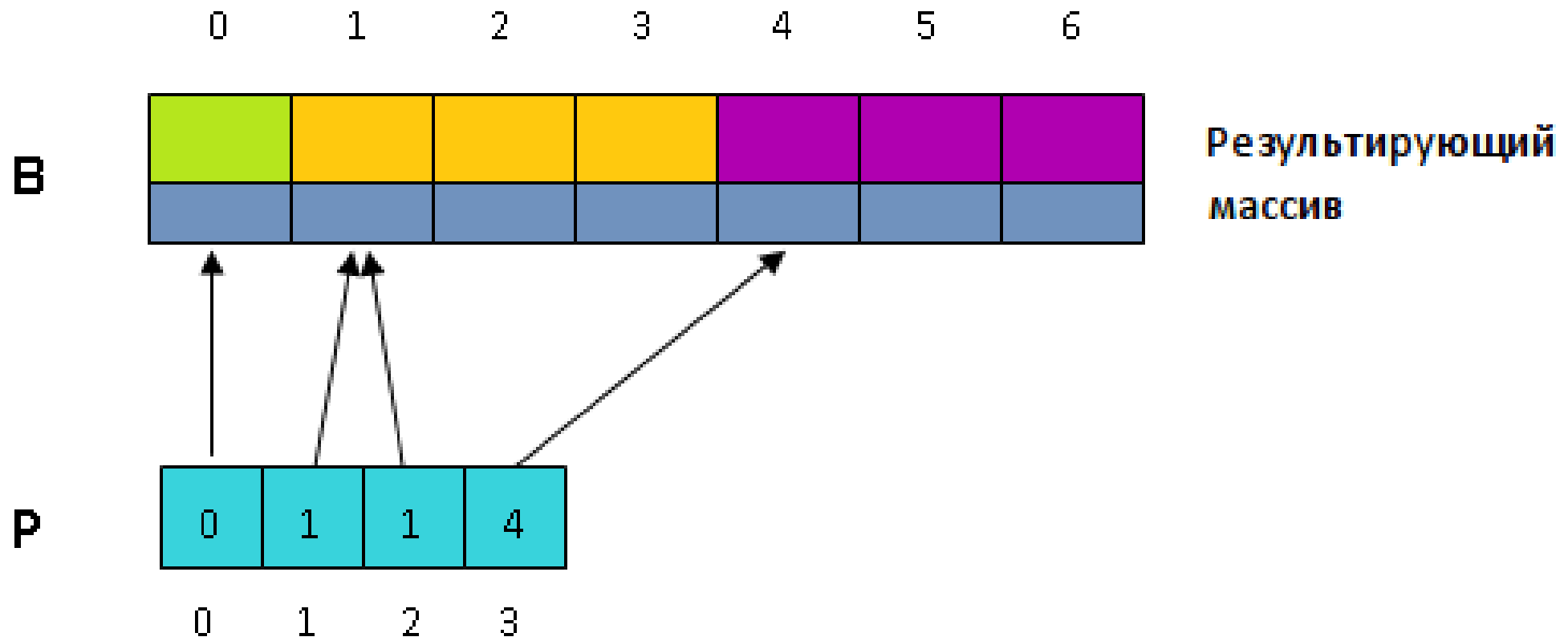
π

Второй шаг

**Р**

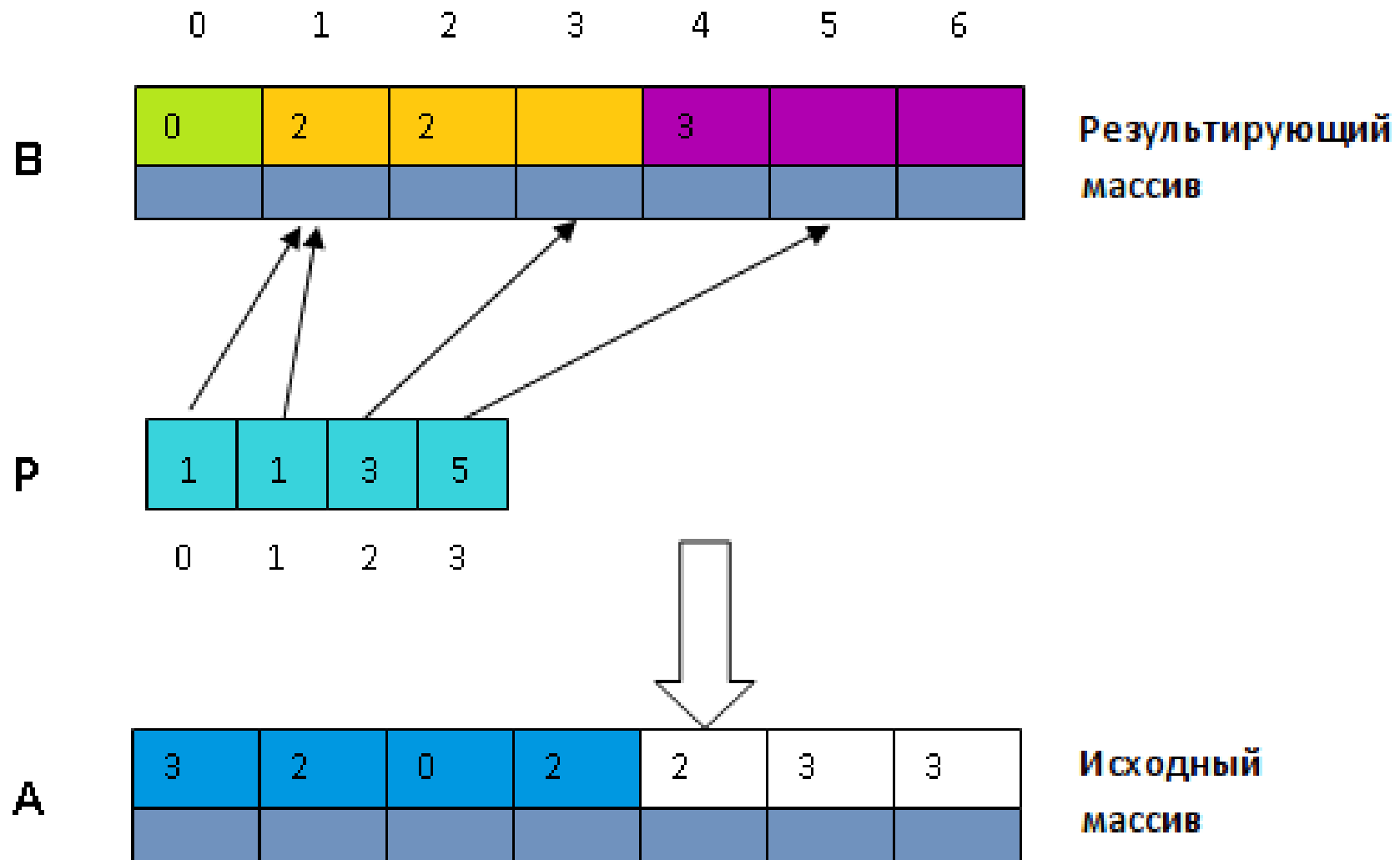
π

Третий шаг



π

Четвертый шаг



Псевдокод

```
function complexCountingSort(A: int[n], B: int[n]):  
    for i = 0 to k - 1  
        P[i] = 0;  
    for i = 0 to length[A] - 1  
        P[A[i].key] = P[A[i].key] + 1;  
    carry = 0;  
    for i = 0 to k - 1  
        temporary = P[i];  
        P[i] = carry;  
        carry = carry + temporary;  
    for i = 0 to length[A] - 1  
        B[P[A[i].key]] = A[i];  
        P[A[i].key] = P[A[i].key] + 1;
```

Анализ

Трудоемкость - $\Theta(n+k)$

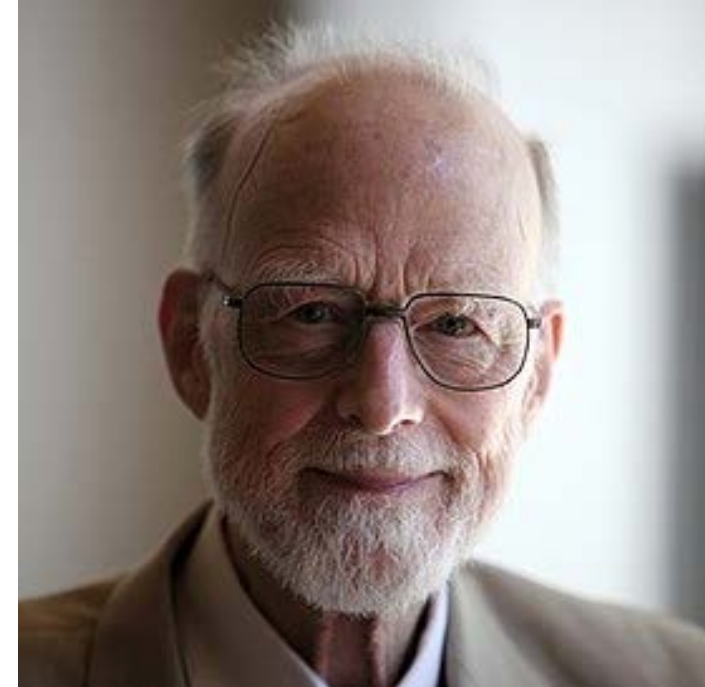
$$\Theta(n)$$

$$k = O(n)$$

Дополнительная память - $\Theta(n+k)$

Быстрая сортировка (C.A.R. Hoare, 1960)

Основная идея **QuickSort**



«разделяй и властвуй»

Делим сортируемый массив на две части, а затем сортируем эти части независимо друг от друга.

И так рекурсивно

Интерпретация QuickSort в терминах принципа «разделяй и властвуй»

- › **Разделять:** Разделить n исходных элементов по трем спискам, обозначенным как *smallList*, *equalList*, *bigList*, где все элементы в *smallList* меньше, чем все элементы в *equalList*, все элементы в *equalList* имеют одинаковые/близкие значения, и все элементы в *equalList* меньше, чем все элементы в *bigList*
- › **Властвовать:** Рекурсивно отсортировать *smallList* и *bigList*
- › **Совместить:** «Склеить», сцепить *smallList* и *bigList*

Общий механизм сортировки

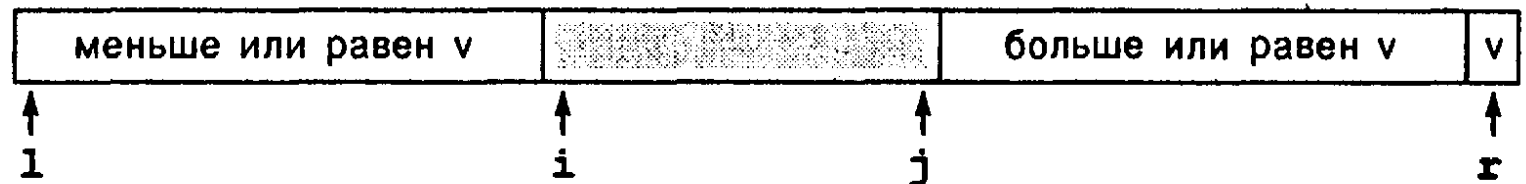
- › Выбрать элемент из массива – опорный/разделяющий элемент, пивот
- › *Разбиение*: перераспределение элементов в массиве таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.
- › Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента.
Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы

Псевдокод (пример)

```
void quicksort(a: T[n], int l, int r)
    if l < r
        int q = partition(a, l, r)
        quicksort(a, l, q)
        quicksort(a, q+1, r)
```

Стратегия разбиения

- › Разделяющий элемент занимает окончательную позицию
- › Начинаем просмотр с левого конца массива до тех пор, пока не будет найден превосходящий опорный элемент.
- › Затем начинаем просмотр с правого конца массива, пока не отыщется меньший опорного элемент
- › Оба элемента, на которых прерван просмотр, меняются местами
- › Продолжаем дальше, пока слева от левого указателя не осталось элементов больше разделяющего, а справа от правого указателя не осталось элементов меньше разделяющего



Псевдокод разбиения

```
int partition(a: T[n], int l, int r)
    T v = ?????????
    int i = l
    int j = r
    while (i<=j)
        while (a[i] < v) i++
        while (a[j] > v) j--
        if(i>=j) break
        swap(a[i++], a[j--])
    return j
```

Проблемы процедуры разбиения

- › Не очевидно, что индексы i, j не выходят за границы промежутка $[l, r]$ в процессе работы
- › Важно, что в качестве граничного значения выбирается $a[l]$, а не $a[r]$. Может оказаться, что $a[r]$ самый большой элемент массива, и в конце выполнения процедуры будет $i=j=r$ и возвращать j будет нельзя – процедура зациклится.

Выбор опорного элемента

- › Первый/последний элемент (первые версии)
- › Средний элемент
- › Случайный элемент
- › Медиана (первый, средний, последний)
- › Разбиение Ломута
- › Разбиение Хоара

Быстрая сортировка

?????? вопросы ?????

1. Трудоемкость —
 $O(n^2)$ — наихудшее время, маловероятно
 $O(n \log n)$ — лучшее, среднее время
 $O(n \log n)$ — в среднем обменов
2. Модификация — множество реализаций
3. Устойчивость — нет
4. Затраты памяти — $O(\log n)$ — стек вызовов
5. Для каких массивов — ????

Модификации

Нерекурсивная реализация

```
void quicksort(a: T[n], int l, int r)
    stack< pair<int,int> > s
    s.push(l, r)
    while (s.isNotEmpty)
        (l, r) = s.pop()
        if (r ≤ l)
            continue
        int i = partition(a, l, r)
        if (i - l > r - i)
            s.push(l, i - 1)
            s.push(i + 1, r)
        else
            s.push(i + 1, r)
            s.push(l, i - 1)
```

Модификации

Улучшенная быстрая сортировка

```
const int M = 10
void quicksort(a: T[n], int l, int r)
    if (r - l ≤ M)
        insertion(a, l, r)
    return
    int med = median(a[l], a[(l + r) / 2], a[r])
    swap(a[med], a[(l + r) / 2])
    int i = partition(a, l, r)
    quicksort(a, l, i)
    quicksort(a, i + 1, r)
```


Сравнительный анализ

N	Метод Шелла	Быстрая сортировка			Быстрая сортировка с вычислением медианы из трех элементов		
		M=0	M=10	M=20	M=0	M=10	M=20
12500	6	2	2	2	3	2	3
25000	10	5	5	5	5	4	6
50000	26	11	10	10	12	9	14
100000	58	24	22	22	25	20	28
200000	126	53	48	50	52	44	54
400000	278	116	105	110	114	97	118
800000	616	255	231	241	252	213	258

Достоинства и недостатки

1. Один из самых быстродействующих (на практике) из алгоритмов внутренней сортировки общего назначения.
2. Алгоритм очень короткий: запомнив основные моменты, его легко написать «из головы», невелика константа при $n \log n$
3. Требуется мало дополнительной памяти (от $O(1)$ до $O(n)$)
4. Хорошо сочетается с механизмами кэширования и виртуальной памяти

Достоинства и недостатки

5. Допускает естественное распараллеливание
6. Допускает эффективную модификацию для сортировки по нескольким ключам: благодаря тому, что в процессе разделения автоматически выделяется отрезок элементов, равных опорному, этот отрезок можно сразу же сортировать по следующему ключу
7. Работает на связных списках и других структурах с последовательным доступом, допускающих эффективный проход как от начала к концу, так и от конца к началу

Достоинства и **недостатки**

1. Сильно деградирует по скорости (до $O(n^2)$) в худшем или близком к нему случае, что может случиться при неудачных входных данных.
2. Прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека, так как в худшем случае ей может потребоваться сделать $O(n)$ вложенных рекурсивных вызовов
3. Неустойчив

Параллельный вариант

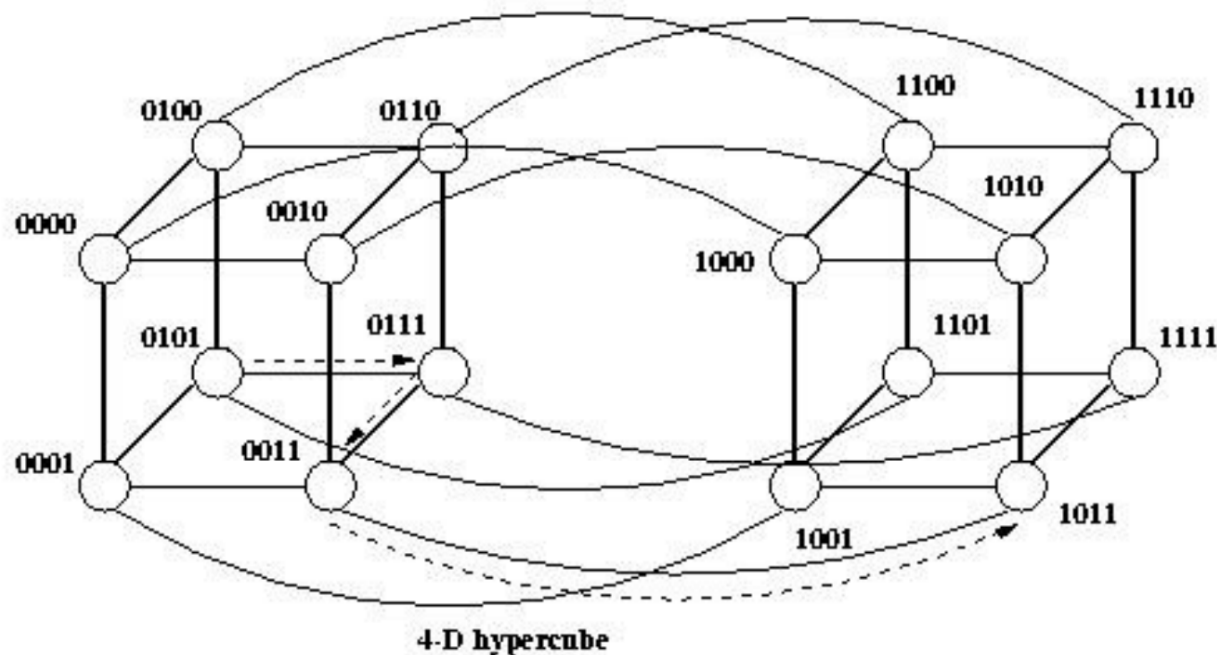
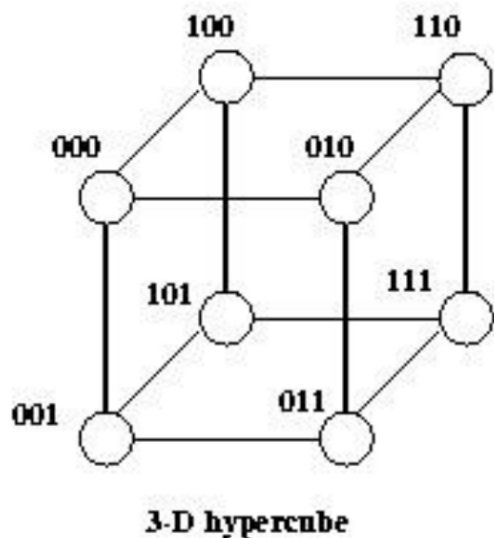
1. Выбор опорного элемента
2. Разбиение набора на каждом узле
3. Узлы разбиваются на “верхние” и “нижние”
 1. “Верхние” отправляют “нижним” значения, меньше опорного элемента
 2. “Все, что больше опорного элемента, “нижние” отправляют “верхним”
4. Повторение процесса в каждой группе до предела.
5. Последовательная сортировка значений на каждом узле

Параллельный вариант (проблемы)

1. Проблемы с балансировкой нагрузки
2. Сложность выбора подходящего опорного элемента

HyperQuicksort – реализация быстрой сортировки на гиперкубе

1. Изначально предполагается, что n элементов равномерно распределены по 2^d вершинам гиперкуба так, что каждая вершина содержит $N=n/2^d$ элементов



HyperQuicksort

1. Сортировка элементов на каждом узле
2. Выбор в качестве опорного элемента чьей-либо медианы
3. Пункты 2-3 предыдущего варианта
4. Сортировка элементов на каждом узле
5. Рекурсивное повторение процедуры
6. Последовательное соединение

HyperQuicksort

ожидаемое время выполнения

$$\Theta\left(N \log N + \frac{d(d+1)}{2} + dN\right).$$

Выражение $N \log N$ представляет время первого шага, второе слагаемое – передачу данных, выражение dN – время, необходимое для обмена и объединения множеств элементов

Introsort

Использует быструю сортировку и переключается на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень

Parallel Sorting by Regular Sampling (PSRS) 1992

Преимущества по сравнению с быстрой сортировкой:

- › сохраняет размер списка более сбалансированным на протяжении всего процесса
- › избегает повторных перестановок ключей

(P процессоров, набор из n элементов)

Parallel Sorting by Regular Sampling (PSRS) 1992

1. Сортировка элементов на каждом узле
2. Выбор на каждом узле ключевых элементов $(0, n/P^2, 2n/P^2, \dots, (P-1)n/P^2)$
3. Сбор элементов на одном узле X , их сортировка.
4. Выбор на X $P-1$ опорных элементов, их рассылка остальным узлам.
5. Каждый узел делит свои элементы на P частей. Узел i оставляет себе i -ую часть, а j -ую – отправляет узлу j .
6. Каждый узел соединяет полученные части в отсортированный набор.
7. Последовательное соединение частей

Parallel Sorting by Regular Sampling (PSRS)

ПРИМЕР

[15,46,48,93,39,6,72,91,14,36,69,40,89,61,97,12,21,54,53,97,84,58,32,27,33,72,20] – 27 элементов, 3 процессора

Описание этапа	1 процессор	2 процессор	3 процессор
Разделение между процессорами	15 46 48 93 39 6 72 91 14	36 69 40 89 61 97 12 21 54	53 97 84 58 32 27 33 72 20
После сортировки частей	6 14 15 39 46 48 72 91 93	12 21 36 40 54 61 69 89 97	20 27 32 33 53 58 72 84 97
Выбор элементов	6 14 15 39 46 48 72 91 93	12 21 36 40 54 61 69 89 97	20 27 32 33 53 58 72 84 97

Описание этапа	Данные
Выбранные элементы	6 39 72 12 40 69 20 33 72
После сортировки	6 12 20 33 39 40 69 72 72
Выбор элементов	6 12 20 33 39 40 69 72 72
Разделители	33 69

Parallel Sorting by Regular Sampling (PSRS)

ПРИМЕР

[15,46,48,93,39,6,72,91,14,36,69,40,89,61,97,12,21,54,53,97,84,58,32,27,33,72,20] – 27 элементов, 3 процессора

Описание этапа		1 процессор			2 процессор			3 процессор	
После сортировки частей	6 14 15	39 46 48	72 91 93	12 21	36 40 54 61 69	89 97	20 27 32 33	53 58	72 84 97
После обмена данными	6 14 15	12 21	20 27 32 33	39 46 48	36 40 54 61 69	53 58	72 91 93	89 97	72 84 97
После слияния	6 12 14	15 20 21	27 32 33	36 39 40	46 48 53	54 58 61 69	72 72	84 89 91	93 97 97

$$O\left(\frac{n \log(n/p)}{p}\right)$$

Сортировка слиянием (*John von Neumann, 1945*)

Основная идея **MergeSort**



«разделяй и властвуй»

Можно рассматривать дополнение быстрой сортировки в том, что состоит из двух рекурсивных вызовов с последующей процедурой слияния

Интерпретация MergeSort в терминах принципа «разделяй и властвуй»

- › **Разделять:** Разделим неупорядоченную n -элементную последовательность на две неупорядоченные подпоследовательности (ПП), причем каждая содержит $n/2$ элементов.
- › **Властвовать:** Рекурсивно сортируем каждую из двух ПП. Если ПП содержит только один элемент, то эту ПП не нужно рекурсивно сортировать, т.к. один элемент уже является отсортированным.
- › **Совместить:** Объединить две отсортированные ПП с помощью *слияния* их в отсортированный результат.

Сортировка слиянием

```
MergeSort (A[a0, a1, ... an-1] ) :  
  if (n==0)  return A  
  B = MergeSort ([a0, ..., an/2])  
  C = MergeSort ([an/2+1, ..., an-1])  
  return merge (B, C)
```

Сортировка слиянием

Узкое место – слияние наборов

$a[0], \dots, a[N-1]$ и $b[0], \dots, b[M-1] \rightarrow c[0], \dots, c[N+M-1]$

Стратегия: последовательно выбирать для c наименьший оставшийся элемент из a и b .

Двухпутевое слияние

```
for (int i = 0, j = 0, k = 0; k < N+M; k++)  
{  
    if (i == N) { c[k] = b[j++]; continue; }  
    if (j == M) { c[k] = a[i++]; continue; }  
    c[k] = (a[i] < b[j]) ? a[i++] : b[j++];  
}
```

[illegible]

Сортировка слиянием – итеративный алгоритм

```
function mergeSortIterative(a : int[n]):  
    for i = 1 to n, i *= 2  
        for j = 0 to n - i, j += 2 * i  
            merge(a, j, j + i, min(j + 2 * i, n))
```

Достоинства и недостатки

1. Работает на структурах данных последовательного доступа
2. Хорошо сочетается с подкачкой и кэшированием памяти
3. Возможно распараллеливание
4. Не подвержена «деградации», как быстрая сортировка
5. Устойчивая
6. *Возможность сортировки данных, расположенных на периферийных устройствах и не вмещающихся в оперативную память*

Достоинства и **недостатки**

1. На «почти отсортированных» массивах работает столь же долго, как на хаотичных.
2. Требуется дополнительной памяти по размеру исходного массива

Сортировка слиянием

?????? вопросы ?????

1. Трудоемкость —
 $O(n \log n)$ — в любом случае
2. Модификация — множество реализаций
3. Устойчивость — да
4. Затраты памяти — $O(n)$ — для массива, $O(1)$ — для списка
5. Для каких массивов — ????

Сравнительный анализ

N	Быстрая	сверху вниз			снизу вверх	
		стандарт	отсечение файлов небольшог о размера	с отсечением и без копирования массива	стандарт	с отсечением файлов небольшого размера
12500	2	5	4	4	5	4
25000	5	12	8	8	11	9
50000	11	23	20	17	26	23
100000	24	53	43	37	59	53
200000	52	111	92	78	127	110
400000	109	237	198	168	267	232
800000	241	524	426	358	568	496

Вопросы к размышлению

1. Усовершенствование базового алгоритма
2. Параллельная реализация
3. Проблемы при работе со связными списками
4. Алгоритмы слияния

Сравнение времени работы mergeSort и mergeSortParallel

