АЛГОРИТМЫ СОРТИРОВКИ

План лекции

- Понятие сортировки
  - Внутренняя и внешняя сортировка
- Алгоритмы сортировки
  - Простые
  - Улучшенные
  - Анализ числа операций
- Нижняя оценка числа операций в алгоритмах сортировки

Задача сортировки

- Значения элементов массива (списка) делятся на ключ и произвольные данные
  struct KeyData { K key; T data; };
- Ключ можно рассматривать как значение функции T -> K, которая вычисляет ключ key на основании (сколь угодно сложного) анализа данных data

- Сортировкой N объектов a[1], ..., a[N] называется нахождение перестановки p1, p2, …, pN, удовлетворяющей условиям
  a[p1].key <= a[p2].key <= ... <= a[pN].key

Устойчивая сортировка

- Сортировка называется устойчивой, если она не меняет порядок объектов с одинаковыми ключами
  $a[i].key = a[j].key$ и $i < j \Rightarrow p_i < p_j$
  где pi и pj – позиции a[i] и a[j] после сортировки

Классификация алгоритмов сортировки

- По объёму данных
  - Внутренние
    - Все данные в памяти
  - Внешние
    - Часть данных на диске

- По алгоритмическому приёму
  - Включение
  - Выбор
  - Обмен
  - Подсчет
  - Разделение
  - Слияние

Сортировка включением

- Разделим элементы массива на неотсортированную часть a[i], … , a[N] и отсортированную часть a[1], … , a[i-1]
- На i-м шаге i-й элемент неотсортированной части вставляется в подходящее место отсортированной части

- Пусть 2 <= i <= N, a[1], … , a[i-1] уже отсортированы
  a[1] <= a[2] <= … <= a[i-1]
- Найдём max j от 1 до i-1 такой, что a[j] <= a[i]
- Сдвинем a[j+1], …, a[i-1] на одно место вправо и переместим запись a[i] в позицию j + 1

Пример

Процесс сортировки включениями покажем на примере последовательности, состоящей из восьми ключей:

i = 1 40 | 51 8 38 90 14 2 63

i = 2 40 51 | 8 38 90 14 2 63

i = 3 8 40 51 | 38 90 14 2 63

i = 4 8 38 40 51 | 90 14 2 63

i = 5 8 38 40 51 90 | 14 2 63

i = 6 8 14 38 40 51 90 | 2 63

i = 7 2 8 14 38 40 51 90 | 63

i = 8 2 8 14 38 40 51 63 90 |

Программа

```
void sort_by_insertion(struct KeyData a[], int N)
{
    int i;
    for (i=1; i < N; i++) {
        struct KeyData x = a[i];
        int j;
        for (j=i-1;j>=0 && x.key<a[j].key; j--)
            a[j+1]= a[j];
        a[j+1]=x;
    }
}
```

Анализ сортировки включением

- Для числа сравнений Ci во внутреннем цикле на i-м шаге внешнего цикла справедливо 1 <= Ci <= i – 1
- Для числа пересылок Mi во внутреннем цикле на i-м шаге внешнего цикла справедливо Mi = Ci + 2
- Всего шагов внешнего цикла N - 1
- Min и max число сравнений и пересылок в худшем и лучшем случаях

$$C_{\min} = N - 1,$$

$$C_{\max} = 1 + 2 + \ldots + N - 1 = \frac{N \cdot (N-1)}{2},$$

$$M_{\min} = 2 \cdot (N - 1),$$

$$M_{\max} = \frac{N \cdot (N-1)}{2} + 2 \cdot (N-1) \approx \frac{N \cdot (N+3)}{2}.$$

Сортировка бинарными включениями

- Если место для a[i] искать методом бинарного поиска в отсортированном массиве, то всего будет произведено ~ N*log2N сравнений
  - Почему?
- Количество пересылок не изменится
  - Почему?

Сортировка выбором

- Разделим элементы массива на неотсортированную часть a[i], ... , a[N] и отсортированную часть a[1], ... , a[i-1]
- На i-м шаге i-й и min элемент неотсортированной части меняются местами и отсортированная часть расширяется на 1 элемент

- Пусть 2 <= i <= N, a[1], ... , a[i-1] уже отсортированы
  a[1] <= a[2] <= ... <= a[i-1]
- Найдём min a[j], j = i … N
- Обменяем местами a[i] и a[j]

Программа

```
void sort_by_selection(struct KeyData key a[], int N)
{
    int  i;
    for( i=0; i < N-1; i++) {
        int j;
        int k=i; /* индекс минимального элемента */
        for (j=i+1; j < N; j++)
            if(a[k].key > a[j].key) k=j;
        if (i!=k) {
            struct KeyData x=a[i];
            a[i]=a[k]; a[k]=x;
        }
    }
}
```

Анализ сортировки выбором

- Число Ci сравнений на i-м шаге внешего цикла есть N-i
  - На первом шаге первый элемент сравнивается с остальными N – 1 элементами
  - На втором шаге число сравнений будет — N - 2 и т. д.
- Поэтому число сравнений есть
  Cmin = Cmax = (N – 1) + (N – 2) + ... + 1 = N * (N - 1)/2
- Max число пересылок Mmax = N -1 так как на каждом проходе выполняется обмен найденного минимального элемента с i-м
  - Чему равно Mmin?

Анализ сортировки выбором

- Число сравнений в методе выбора всегда равно максимальному числу сравнений в методе простых включений
- Число перемещений минимально
- Если вспомнить, что сравниваются ключи, а перемещаются записи целиком, то метод выбора, экономящий число перемещений может на практике оказаться предпочтительней

Пирамидальная сортировка

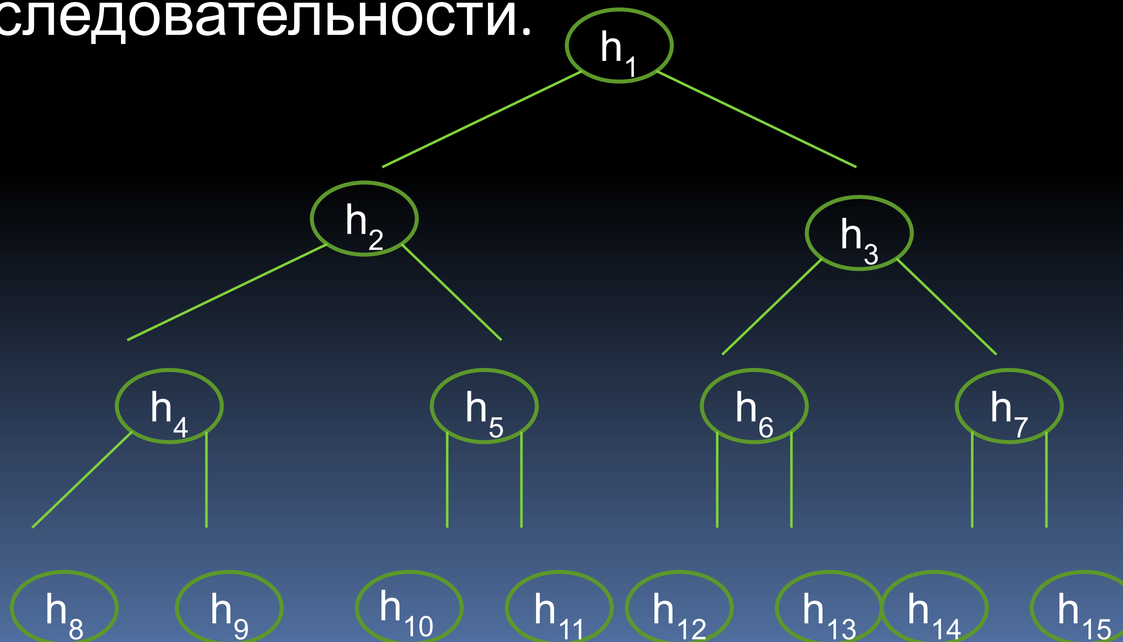
- Линейный поиск min элемента делает сложность всей сортировки выбором квадратичной
- Как найти минимальный элемент быстрее, чем за O(N) операций?
- Williams, James "Algorithm 232 - Heapsort" ACM 1964
  - Не удалось найти первоисточник
- Floyd, Robert W. 1936-2001 "Algorithm 245 - Treesort 3" ACM 1964
- Организовать входные данные в виде так называемой пирамиды упрощающей поиск минимума и поддерживать их в этом виде в процессе сортировки
  - Накапливать информацию о взаимных отношениях элементов

Определение пирамиды

- Пусть дана последовательность h[1], ..., h[n]
- Элемент h[i] образует пирамиду в этой последовательности, если выполнены следующие условия
  - Если $2 * i \le n$, то $h[i] \ge h[2i]$ и h[2i] образует пирамиду
  - Если $2 * i + 1 \le n$, то $h[i] \ge h[2i+1]$ и h[2i+1] образует пирамиду
- Элементы h[n/2+1], ..., h[n] всегда образуют тривиальные пирамиды, поскольку для них приведенные условия имеют ложные посылки
- Последовательность, упорядоченная по убыванию, является полной пирамидой – почему?
- Если элемент h[1] образует пирамиду, то и каждый элемент последовательности образует пирамиду
- В этом случае будем говорить, что вся последовательность является полной пирамидой

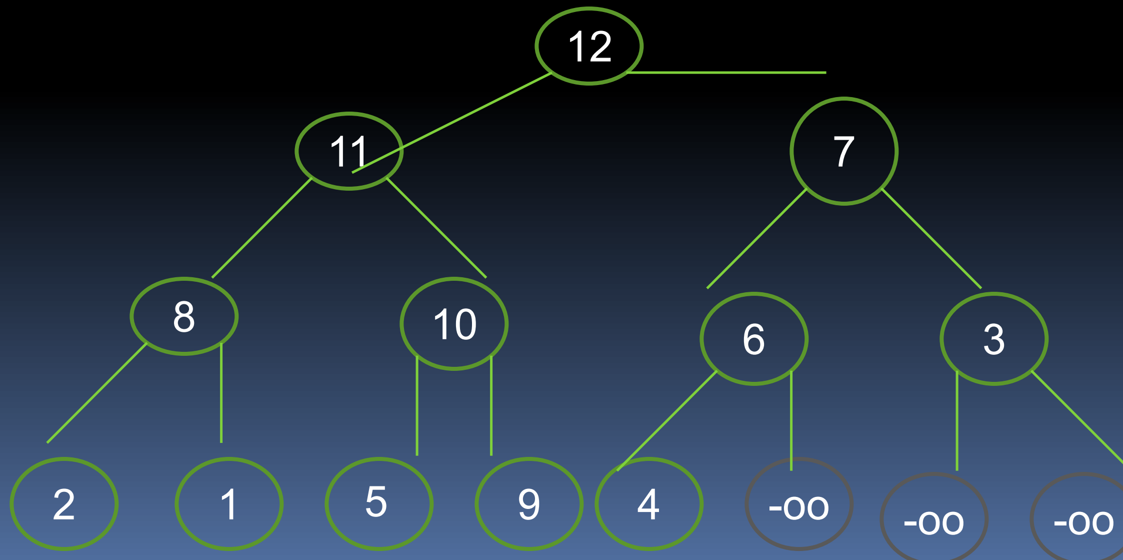
Полная пирамида при n = 15

- Полная пирамида может быть изображена в виде корневого бинарного дерева, в котором элементы h[2i] и h[2i+1] являются сыновьями элемента h[i].
- Элемент в любом узле численно не меньше всех своих потомков, а вершина полной пирамиды h[1] содержит максимальный элемент всей последовательности.



Пример полной пирамиды при n = 12

- Если число элементов в полной пирамиде не равно 2^k – 1, самый нижний уровень дерева будет неполным: недостающих сыновей можно достроить, добавив в пирамиду несколько заключительных «минимальных» элементов «-оо», не нарушающих условия пирамиды



Макет пирамидальной сортировки

- Подготовка к сортировке
  - Входная неупорядоченная последовательность перестраивается в пирамиду
- Сортировка
  - Массив делится на две части
    - Неупорядоченное начало массива
    - Упорядоченный конец массива
  - Пока не упорядочим весь массив
    - Меняем местами h[1] и последний элемент неупорядоченной части
    - Восстанавливаем пирамиду начиная с h[1]

Макет пирамидальной сортировки

- Основой алгоритм является процедура просеивания, так перестраивающая h[i+1], ..., h[n], образующие пирамиду, чтобы пирамиду образовывал и элемент h[i]

- На каждой итерации цикла наибольший из трех элементов h[i], h[2*i] и h[2*i+1] путем обмена ставим в корень h[i] текущего поддерева
  - Выполнили первую часть условия пирамиды для h[i]
- Если при этом изменяется корень левого или правого поддерева, то просеивание продолжается для него
  - Начинаем выполнять вторую часть условия пирамиды для h[i]

Просеивание

```
Просеять (h, i, n)
    пока 2 * i < n /* цикл просеивания h[i]  в поддеревья*/
        hL = h[2*i];
        hR = 2 * i + 1 < n ? h[2*i+1] : -oo;
        если h[i] > hL и h[i] > hR то конец просеивания
        если hL > hR
        то  /* просеять в левое п/дерево */
            обменять h[i] и h[2*i];
            i = 2 * i;
        иначе если hR > -oo
        то   /* просеять в правое п/дерево, если есть*/
            обменять h[i] и h[2*i+1];
            i =2 * i + 1;
    конец цикла
конец
```

Построение пирамиды

	h1	h2	h3	h4	h5	h6	h7	h8	h9	h10
Шаг 1, i=5:	52	81	42	23	11	76	91	63	37	20
Шаг 2, i=4 :	52	81	42	23	20	76	91	63	37	11
Шаг 3, i=3 :	52	81	42	63	20	76	91	23	37	11
Шаг 4, i=2 :	52	81	91	63	20	76	42	23	37	11
Шаг 5, i=1 :	52	81	91	63	20	76	42	23	37	11
Выход:	91	81	76	63	20	52	42	23	37	11

11 81 63 37 — обмен просеивание

11 76 52 — обмен просеивание

23 52 — обмен просеивание

42 52 — обмен просеивание

11 — обмен просеивание

20 37 11 23 42 52 63 76 81 91 — обмен просеивание

i=4: 37 20 11 23 42 52 63 76 81 91

23 20 11 37 42 52 63 76 81 91 — обмен просеивание

i=3: 23 20 11 37 42 52 63 76 81 91

11 20 23 37 42 52 63 76 81 91 — обмен просеивание

i=2: 20 11 23 37 42 52 63 76 81 91 — обмен

11 20 23 37 42 52 63 76 81 91

Алгоритм пирамидальной сортировки

- Шаг 1
  - Построение пирамиды
    ```
    i = n / 2;
    пока i >= 1
            Просеять(h, i, n);
            i = i - 1;
    конец
    ```
- Шаг 2
  - Сортировка на пирамиде:
    ```
    i = n;
    пока i > 1
            Обмен (h, 1, i);
            i = i - 1;
            Просеять(h, 1, i);
    конец
    ```

Просеивание

```
void Sift(struct KeyData a[], int i, int n)
{
	int l;
	i++;
	while ((l=2*i)<= n) {
		int r = (l+1 <= n)? l+1 : i;
		if (a[i-1].key>=a[l-1] .key&&a[i-1] .key>=a[r-1] .key)
			return;
		int k = a[l-1] .key>= a[r-1] .key ? l : r;
		struct KeyData t  = a[i-1];  // C99
		a[i-1] = a[k-1];
		a[k-1] = t;
		i = k;
	}
}
```

Пирамидальная сортировка

```
void heap_sort(struct KeyData a[], int N)
{
    int i;
    /* строим полную пирамиду */
    for (i = N/2; i >= 0; i--) Sift (a, i, N);
    /* сортируем */
    for (i = N-1; i > 0; i--) {
        struct KeyData t = a[0];
        a[0] = a[i];
        a[i] = t;
        Sift (a, 0, i); /* восстанавливаем пирамиду */
    }
}
```

Анализ пирамидальной сортировки

- Число итераций цикла в процедуре просеивания не превосходит высоты пирамиды
- Высота полного бинарного дерева из N узлов, каковым является пирамида, равна [log2 N]
- Просеивание имеет логарифмическую сложность
- Итоговая сложность пирамидальной сортировки ~N * log2 N
- Наилучший случай – обратное упорядочение входной последовательности
  - Почему?

- Понятие сортировки
  - Внутренняя и внешняя сортировка
- Алгоритмы сортировки
  - Простые
    - Включением, выбором, пузырёк
  - Улучшенные
    - Пирамидальная, быстрая
  - Анализ числа операций
- Нижняя оценка числа операций в алгоритмах сортировки

Сортировка простым обменом (пузырёк)

- На первом шаге сравним последний и предпоследний элементы, если они не упорядочены, поменяем их местами.
- Далее проделаем то же со вторым и третьим элементами от конца массива, третьим и четвертым и т. д. до первого и второго с начала массива.
- При выполнении этой последовательности операций меньшие элементы в каждой паре продвинутся влево, наименьший займет первое место в массиве.
- Повторим этот же процесс от N-го до 2-го элемента, потом от N-го до 3-го и т. д.

- i-й проход по массиву приводит к «всплыванию» наименьшего элемента из входной последовательности на i-е место в готовую последовательность.

Пример

$i = 0$	40	51	8	38	90	14	2	63
$i = 1$	2	40	51	8	38	90	14	63
$i = 2$	2	8	40	51	14	38	90	63
$i = 3$	2	8	14	40	51	38	63	90
$i = 4$	2	8	14	38	40	51	63	90
$i = 5$	2	8	14	38	40	51	63	90
$i = 6$	2	8	14	38	40	51	63	90
$i = 7$	2	8	14	38	40	51	63	90

Алгоритм (метод пузырька)

```
для i от 2 до N с шагом 1
    // проход от конца массива к началу
    для j от N до i с шагом -1
        если A[j–1] > A[j]
        то Обмен(A, j, j–1)
    конец для
конец для
```

Анализ

- Поскольку число сравнений C_i на i-м шаге внешнего цикла равно N-i

  $C_{min} = C_{max} = C = (N - 1) + (N - 2) + \ldots + 1 =$
  $= N \cdot (N - 1)/2$

- Минимальное число пересылок $M_{min} = 0$
- Максимальное $M_{max} = C$

- В каких случаях достигаются M_{max} и M_{min}?

Улучшение метода пузырька

- Последние проходы сортировки простым обменом работают «вхолостую», если элементы уже упорядочены
  - Запомним, производился ли на очередном проходе обмен. Если ни одного обмена не было, то алгоритм может закончить работу.
- Один неправильно расположенный «пузырек» на «тяжелом» конце почти отсортированного массива «всплывет» на место за один проход

- Неправильно расположенный «камень» на «легком» конце «опустится» на правильное место за N-1 проход

Шейкер-сортировка (алгоритм)

```
left = 1        // левая граница несортированной части
right = N       // правая граница несортированной части
упор = ложь     // истинна, если массив упорядочен
пока не упор
    упор = истина
    i = left
    //Проход по массиву от начала к концу:
    пока i < right
        если A[i] > A[i + 1] то
            Обмен (A, i, i+1)
            упор = ложь
        конец если
        i = i + 1
    конец пока
    // >=1 элемент встал на своё место справа
    right = right – 1
// см. след. слайд
```

Шейкер-сортировка (продолжение)

```
    если не упор то // TODO – rewrite
        упор = истина
        i = right
        пока i > left
            если A[i] < A[i – 1] то
                Обмен (i, i–1)
                упор = ложь
            конец если
            i = i – 1
        конец пока
    конец если
    // >= 1 элемент встал на своё место слева
    left = left + 1
конец пока // не упор
```

Программа

```
void shaker_sort (struct KeyData A [], int N)
{
    int left = 0, right = N-1;
    int  sorted = 0;
    while (!sorted && left < right) {
        int i;
        sorted = 1;
        for (i = left;  i < right; ++i) {
            if (A[i+1] < A[i]) {
                struct KeyData x = A[i+1];
                A[i+1] = A[i];
                A[i] = x;
                sorted = 0;
            }
        }
        right -= 1;
```

Продолжение программы

```
        if (!sorted) for (i = left+1; i<=right; i++) {
            if (A[i-1] > A[i]) {
                struct KeyData x = A [i-1];
                A[i-1] = A[i];
                A[i] = x;
            }
        }
        left += 1;
    } // while
} // shaker sort
```

Анализ

- Cmin= N –1, Cmax = O(N*N)
- Число пересылок такое же как для сортировки пузырьком
  - Каждый обмен соседних элементов уменьшает число инверсий (пар элементов, нарушающих порядок) в массиве на 1
  - Любой алгоритм, основанный на обмене пар соседних элементов, делает столько пересылок, сколько в массиве инверсий
- Сортировка обменом и ее улучшенная сортировка хуже, чем сортировка включениями и выбором по числу пересылок
  - Почему?
- Шейкер-сортировку выгодно использовать тогда, когда массив почти упорядочен
- Шейкер (англ. "to shake" -- трясти) – это устройство для приготовления жидких смесей

Сортировка подсчётом

- Для каждого элемента подсчитаем число элементов, которые меньше его
- Это число (+1) есть его позиция элемента в отсортированной последовательности
  - При условии, что все элементы различны
- Наивная реализация записывает отсортированные элементы в новый массив

Алгоритм (на одном массиве)

```
i = 1;  // номер 1-го элемента в несортированной части массива
пока i < N
      r = 1; // число элементов в массиве, меньших A[i]
      цикл по j от 1 до N с шагом 1 выполнять
            если A[i] > A[j]
            то r = r + 1;
      конец цикла
      если r >= i         // i-й элемент стоит на своем месте,
      то i = i + 1        // увеличить сортированную часть на 1 элемент
      иначе
            // вычислить позицию, куда нужно поставить i-й элемент:
            пока A[r] == A[i]
                  r = r+1
            конец пока
            // поменять его местами с тем элементом,
            // который находится в этой позиции
            Обмен (A, r, i)
      конец
конец пока
```

Сортировка с разделением

- Быстрая сортировка Ч. Э. Р. Хоар
  - В время парктики Московском Государственном Университете им. Ломоносова
  - Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". Comm. ACM 4 (7): 321
- Рекурсивная сортировка обменом пар несоседних элементов

Сортировка разделением, идея алгоритма

- Пусть a[m] произвольный пилотируемый элемент
- Разделим элементы массива относительно a[m] так, что все элементы слева от a[m] не превосходят всех элементов справа от a[m]
  для всех i, j
  если 1 <= i <= m и m < j <= N
  то a[i] <= a[j]
  - Как это сделать?
- Отсортируем любым методом левую часть, не затрагивая элементы правой части
- Отсортируем любым методом правую часть, не затрагивая элементы левой части
- В результате упорядочится весь массив

Сортировка разделением (макет)

```
СортировкаРазделением (a, l, r)
    выбрать пилотируемый элемент m
    разделить a[l],…, a[r] относительно a[m]
    // части длины 0 и 1 не сортируем
    если l < m то
        СортировкаРазделением (a, l, m) // XXX
    иначе если m + 1 < r   то
        СортировкаРазделением (a, m + 1, r)
    конец если
конец
```

Анализ макета

- Массив упорядочивается "сам собой" по мере упорядочения его частей?!
  - Рекурсия приводит к сортировке массива длины 1, которая не требует ни сравнений, ни пересылок
  - Объединение левой и правой части после их сортировки не требуется

- Сортировка происходит на этапе разделения массива относительно пилотируемого элемента

- В классической версии алгоритма в качестве m выбирается произвольный элемент сортируемой последовательности
  - Первый, последний, расположенный в середине или иначе.

  - Выбор m существенно влияет на число сравнений и пересылок – обсудим далее

Разделение массива

- Пусть x = a[m] – значение пилотируемого элемента
- Сортируемую часть массива разделим на три участка
  - a[l] ... a[i-1]   a[k] <= x
  - a[j+1] ... a[r]   a[k] >= x
  - a[i] ... a[j]   неизвестно
- На кажом шаге уменьшаем часть, где неизвестно отношение между a[k] и x

Процесс разделения

```
i = l;  j = r;
пока i < j
     пока a[i]  < x
           i = i + 1; /*  в конце a[i] >=  x */
     конец
     пока x < a[j]
           j = j – 1; /* в конце aj <=  x */
     конец
     если i  >= j  то
           выйти из цикла
     конец если
     обмен( a, i, j )
     i = i + 1; /* расширить левую часть */
     j = j – 1; /* расширить правую часть */
конец пока
```

Комментарии

- Циклы по встречным индексам переносят из средней части в левую или правую элементы, строго меньшие или большие x, которые могут быть добавлены в эти части без перестановки
- После их выполнения процесс разделения либо заканчивается (если $i >= j$), либо пара a_i и a_j образует инверсию
- В последнем случае их следует обменять и включить в левую и правую части
- Здесь происходят упорядочивающие обмены с уменьшением числа инверсий в последовательности

Комментарии

- Проверка того, что бегущие индексы не выходят за границы l...r не нужна
  - На первом проходе выход за границы невозможен, так как в массиве есть сам элемент x и оба цикла остановятся на нем

- В конце же первого прохода происходит обмен элементов и обе части становятся не пусты, что гарантирует остановку циклов по встречным индексам в пределах интервала l...r и на следующих проходах

Комментарии

Цикл оканчивается при $i \geq j$.

Однако нам еще надо определить медиану.

Определенные границы левой части – $l...i - 1$,
правой – $j + 1...r$, однако интервал $j + 1...i - 1$ может быть не вырожден и заполнен элементами x (почему?).

Эти элементы останутся на своих местах в процессе сортировки (почему?), поэтому их можно исключить из левой и правой частей.

Окончательно границами левой части можно считать $l...j$,
а правой – $i...r$.

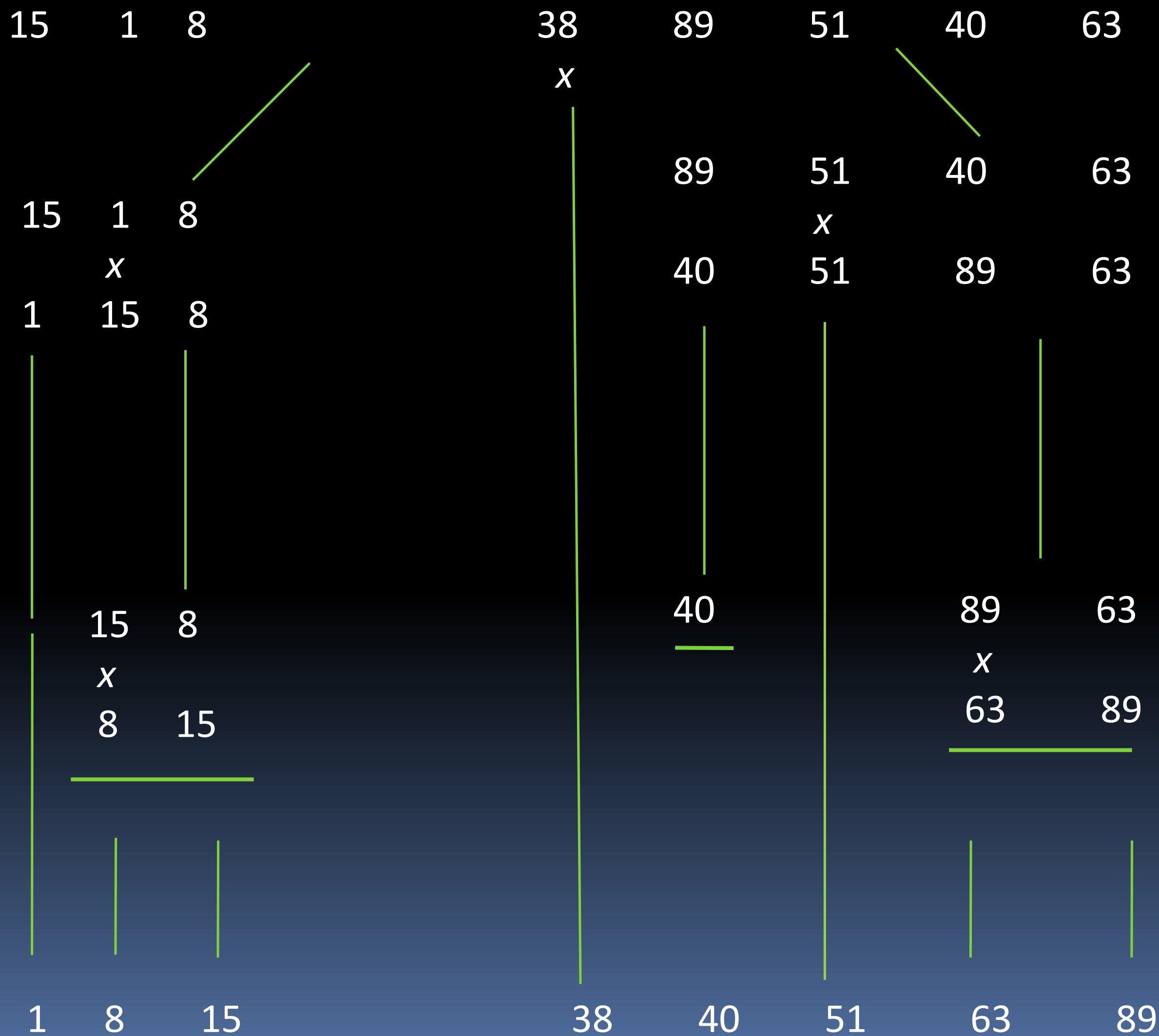
Процесс разделения, пример

```
 Разделение:   40     51     8     38     89     1     15    63
    Начало:    i                   x                         j
Первый проход:i                                         j
Сближение

Обмен
             15     51     8     38     89     1     40     63
Второй проход:       i                                j
Сближение

Обмен
                i                              j
                15      1      8     38     89     51    40     63
Третий проход:                 i                j
Сближение                            ij
Обмен
                15     1       8     38     89     51    40     63
Четвертый проход:              j                i
Сближение
```

Пример быстрой сортировки



Программа

```
void quicksort(struct KeyData a[], int l, int r)
{
    int i = l, j = r;
    struct KeyData x = a[(l+r)/2];
    do {
        while (a[i] < x) ++i;
        while (x < a[j]) --j;
        if (i <= j) {
            struct KeyData w = a[i];
            a[i] = a[j] ;
            a[j] = w;
        }
    } while (i<j);
    if (l<j) quicksort (a, l, j);
    if (i<r) quicksort (a, i, r);
}
```

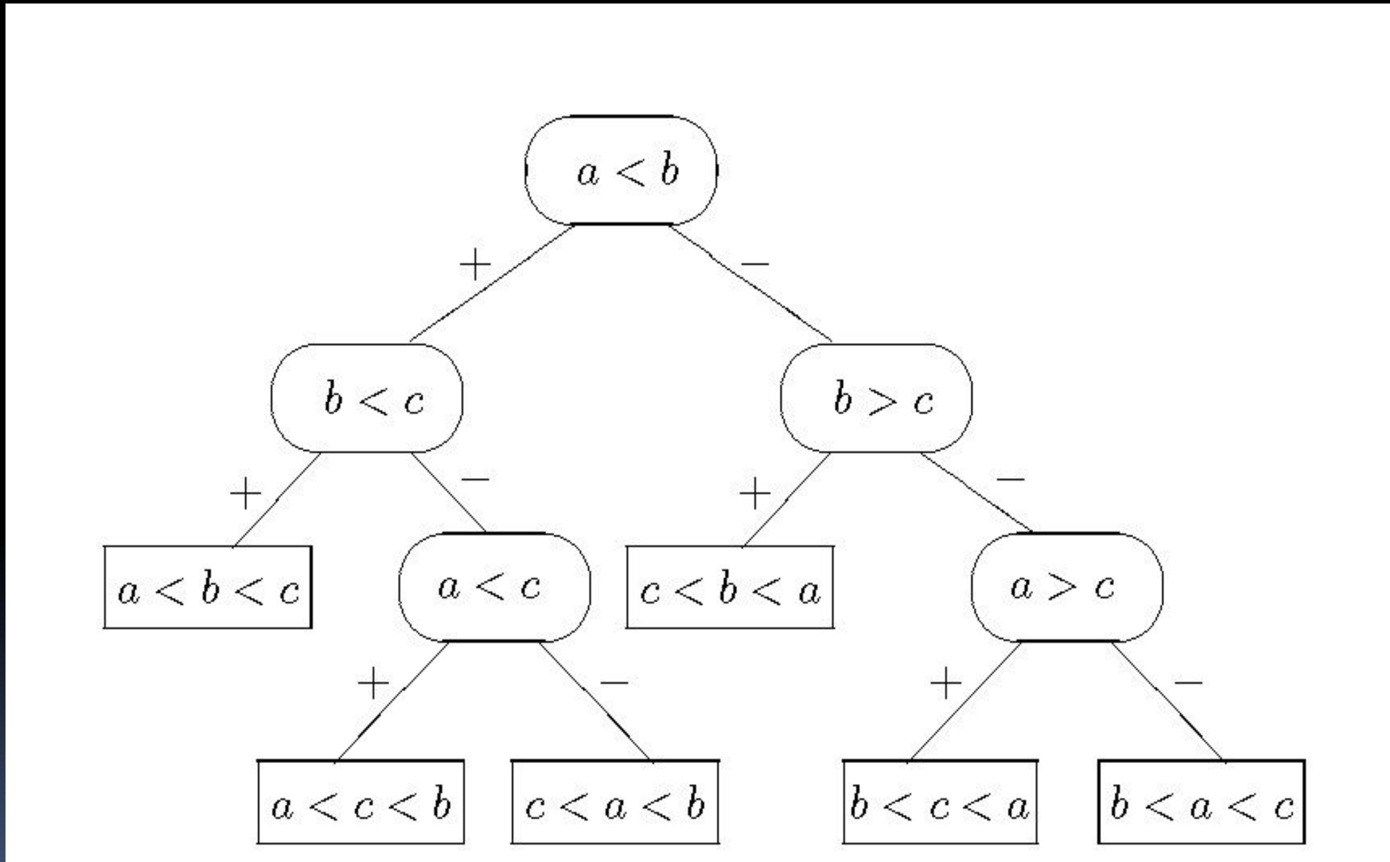
Анализ

- Число сравнений и пересылок зависит от выбора пилотируемого элемента
- Если брать пилотируемый элемент равным медиане массива, то
  - Cmin = O(N*log(N)), Cmax = O(N*log(N))
  - Mmin = 0, Mmax = O(N*log(N))
- В худшем случае
  - Cmin = O(N*N) Cmax = O(N*N)
  - Mmin = 0, Mmax = O(N*N)

Нижняя граница числа сравнений в сортировке

- Для сортировки массива из N элементов необходимо O(N * log2 N) сравнений элементов N --> oo
- Обоснование
  - Для заданной последовательности из N элементов может быть построено N! перестановок
  - Для нахождения перестановки, сортирующей массив, требуется выполнить не менее log2(N!) условных операторов
    - В противном случае для разных массивов будет найдена одна и та же перестановка

Дерево решений для массива из 3 элементов



Для задачи сортировки в дереве решений N! листьев

Значит, высота дерева решений ≥ log2(N)

Из неравенства

$$N! \geq N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot [\frac{N}{2}] \geq (\frac{N}{2})^{\frac{N}{2}}$$

следует заключение теоремы, так как

$$log_2 N! \geq (\frac{N}{2}) \cdot log_2 \frac{N}{2} \geq (\frac{N}{4}) \cdot log_2 N \quad (\text{при } N \geq 4).$$

Число сравнений, которые необходимо сделать, чтобы получить любую из перестановок примера, не меньше 2 (log2(6) ≈ 2.5).

Заключение

- Понятие сортировки
  - Внутренняя и внешняя сортировка
- Алгоритмы сортировки
  - Простые
    - Включением, выбором, пузырёк
  - Улучшенные
    - Пирамидальная, быстрая
  - Анализ числа операций
- Нижняя оценка числа операций в алгоритмах сортировки

Анализ, продолжение

Просуммируем всевозможные варианты выбора медианы и разделим эту сумму на *N*, в результате получим ожидаемое число обменов:

$$M = \frac{1}{N} \cdot \left[\sum_{i=1}^{N} (x-1) \cdot \frac{(N-x+1)}{N} \right] = \frac{N}{6} - \frac{1}{6 \cdot N}.$$

Ожидаемое число обменов равно приблизительно $N/6$.

В лучшем случае каждое разделение разбивает массив на две равные части, а число проходов, необходимых для сортировки, равно $log_2 N$.

Тогда общее число сравнений равно $N \log_2 N$.

Анализ, продолжение

Однако в худшем случае сортировка становится «медленной».

Например, когда в качестве пилотируемого элемента всегда выбирается наибольшее значение. Тогда в результате разбиения в левой части оказывается *N* - 1 элемент, т. е. массив разбивается на подмассивы из одного элемента и

из *N* - 1 элемента.

В этом случае вместо $\log_2 N$ разбиений необходимо сделать ~ *N* разбиений.

В результате в худшем случае оценка оказывается ~ N^2, что гораздо хуже пирамидальной сортировки.

Алгоритм

- Условно разделить массив А на отсортированную и несортированную части. К сортированной части сначала относится только первый элемент.
- цикл по i от 2 до N с шагом 1 выполнять
- // i – номер первого элемента в несортированной части массива
- x = A[i];
- j = i – 1;
- // Все элементы из отсортированной части, большие,
- // чем x, сдвинуть на одну позицию вправо:
- пока j>0 и A[j]>x выполнять
- A[j+1] := A[j];
- j = j – 1;
- конец пока
- // Элемент x поставить на свое место по порядку:
- A[j+1] = x;
- конец цикла

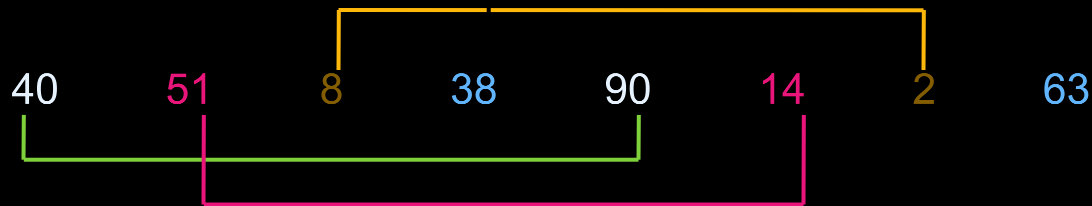
Сортировка включениями с убывающим шагом. Метод Шелла

Хоар, Флойд, Шелл: для алгоритмов сортировки, перемещающих в последовательности запись вправо или влево только на одну позицию, среднее время работы будет в лучшем случае пропорционально N^2.

Хотелось бы, чтобы записи перемещались «большими скачками, а не короткими шажками».

Д. Шелл предложил в 1959 г. метод, названный сортировкой *с убывающим шагом.*

Пример работы сортировки Шелла



Полученные 4 последовательности отсортируем на месте независимо друг от друга методом простых включений.

Этот процесс называется *4-сортировкой.*

Пример работы сортировки Шелла, продолжение

В результате 4-сортировки получим последовательность:

На следующем шаге элементы, отстоящие друг от друга на две позиции, объединяются в подпоследовательности и сортируются простыми вставками независимо друг от друга.

Этот процесс называется *2-сортировкой.*

После 2-сортировки получим последовательность:

2 14 8 38 40 51 90 63

Ее сортируют методом простых вставок.

К последнему шагу элементы довольно хорошо упорядочены,

поэтому требуется мало перемещений.

Данный процесс называется *1-сортировкой.*

Выбор шага в сортировке Шелла

В сортировке методом Шелла можно использовать любую убывающую последовательность шагов

$$h_t,\ h_{t-1},\ \ldots,\ h_1$$

Чтобы выбрать некоторую хорошую последовательность шагов сортировки, нужно проанализировать время работы как функцию от этих шагов.

До сих пор не удалось найти наилучшую возможную последовательность шагов h_t, h_{t-1}, ..., h_1 для больших *N.*

Выявлен примечательный факт, что элементы последовательностей приращений не должны быть кратны друг другу.

Это позволяет на каждом проходе сортировки перемешивать цепочки, которые ранее никак не взаимодействовали.

Желательно, чтобы взаимодействие между разными цепочками происходило как можно чаще.

Кнут:

..., 121, 40, 13, 4, 1, где $h_{k+1} = 3 \cdot h_k + 1$, $h_1 = 1$

..., 31, 15, 7, 3, 1, где $h_{k+1} = 2 \cdot h_k + 1$, $h_1 = 1$

Анализ эффективности метода

Утверждение

Если k-отсортированная последовательность i-сортируется (k > i), то она остается k-отсортированной.

→ С каждым следующим шагом сортировки с убывающим приращением количество отсортированных элементов в последовательности возрастает.

Для последовательности шагов $2^k + 1, \ldots, 9, 5, 3, 1$

количество пересылок пропорционально $N^{1.27}$,

для последовательности $2^k - 1, \ldots, 15, 7, 3, 1$ — $N^{1.26}$,

для последовательности $(3^k - 1)/2, \ldots, 40, 13, 4, 1$ — $N^{1.25}$

Общая оценка: величина порядка $N^{3/2}$

Алгоритм

```
процедура Вставка(b, h)
 // b — номер первого элемента последовательности
 // h – величина шага
начало процедуры
   // Пусть i – номер первого элемента в несортированной части массива
   i = b + h;
   пока i ≤ N выполнять
      x= A[i];
      j = i - h;
      пока j ≥ b и A[j]>x выполнять
      // Все элементы из отсортированной части, большие
      // x, сдвинуть на величину шага h вправо,
         A[j+h] = A[j];
     j = j - h;
      конец пока
      // Элемент x поставить на свое место по порядку:
      A[j+h] = x;
      i = i + h;
   конец пока
конец процедуры
```

Алгоритм, продолжение

```
Основная программа:
// Выбор начального шага:
h = 1;
пока h < N/3 выполнять
   h = 3*h + 1;
конец пока
// Сортировка:
пока h ≥ 1 выполнять
   цикл по i от 1 до h с шагом 1 выполнять
      Вставка (i, h);
   конец цикла
   h = (h – 1) / 3;
конец пока
```

Сортировка простым выбором

Методы сортировки посредством выбора основаны на идее многократного выбора.

На *i*-м шаге выбирается наименьший элемент из входной последовательности a_i, ..., a_n и меняется местами с a_i-м.

Таким образом, после шага *i* на первом месте во входной последовательности будет находиться наименьший элемент.

Затем этот элемент перемещается из входной в готовую последовательность.

Процесс выбора наименьшего элемента из входной последовательности повторяется до тех пор, пока в ней останется только один элемент.

Пример

Проиллюстрируем этот метод на той же последовательности

|40 51 8 38 90 14 2 63.

На первом шаге находим наименьший элемент 2, обмениваем его с первым элементом 40 и перемещаем в готовую последовательность:

2 | 51 8 38 90 14 40 63

2 8 | 51 38 90 14 40 63

2 8 14 | 38 90 51 40 63

2 8 14 38 | 90 51 40 63

2 8 14 38 40 | 51 90 63

2 8 14 38 40 51 | 90 63

2 8 14 38 40 51 63 | 90

Обсуждение

Данный метод в некотором смысле противоположен сортировке простыми включениями.

При сортировке простым выбором рассматриваются все элементы входной последовательности и для фиксированного места из нее выбирается наименьший элемент.

При этом не возникает необходимости "сдвига" участка массива, поскольку выбранный элемент вставляется всегда в конец готовой последовательности. Вытесняемый же элемент достаточно переставить на освободившееся место в несортированной входной части.

Алгоритм

Условно разделить массив А на отсортированную и несортированную части. Сначала весь массив — это несортированная часть.

цикл по i **от** 1 **до** N–1 **с шагом** 1 **выполнять**

// i – номер первого элемента в несортированной части массива

`r = i;`

// Найти минимальный элемент в несортированной части массива:

цикл по j **от `i+1`** **до** N **с шагом** 1 **выполнять**

если `A[j] < A[r]` то `r = j;`

конец цикла

// Найденный минимальный элемент поменять местами с

// первым элементом несортированной части:

если $i \neq r$ **то** Обмен (i, r);

// Он будет последним элементом новой сортированной

// части массива А.

конец цикла

Пирамидальная сортировка

- На каждом шаге сортировки первый элемент массива, минимальный элемент пирамиды, переносится в начало готовой последовательности путем обмена с последним элементом пирамиды, занимающим его место.
- Затем остаток входной последовательности вновь перестраивается в пирамиду, обеспечивая корректность следующего шага.
- В начале i-го шага элементы h[1], .., ai, по предположению, хранят входную последовательность как пирамиду, а ai+1, .., aN – упорядоченную по возрастанию готовую последовательность (изначально пустую).
- На i-м шаге текущий максимальный элемент пирамиды a1 обменивается с ai, становясь началом новой готовой последовательности, где он будет новым минимальным элементом. Входная последовательность (пирамида) при этом претерпевает два изменения:
- — она теряет последний элемент, что не нарушает условий пирамиды ни в одном узле;
- — ее первый элемент становится произвольным, что может нарушать условие пирамиды только в первом узле.

Сортировка, продолжение

Таким образом, для новой входной последовательности

$a_1, \ldots, a_{i-1}$

условия пирамиды выполнены для всех элементов, кроме первого.

Применение процедуры просеивания к a_1 восстанавливает полную пирамиду в $a_1, \ldots, a_{i-1}$, что обеспечивает условия осуществимости следующего шага.