Курс «Алгоритмы и алгоритмические языки»
1 семестр 2017/2018

Лекция 16

Сортировка

- ***Постановка задачи***

  Сортировка – это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, <) по возрастанию или по убыванию.
  Здесь будут рассматриваться целочисленные данные и отношение порядка "<".

- В стандартную библиотеку **`stdlib`** входит функция **`qsort`**:

  ```
  void qsort (void *buf, size_t num, size_t size,
  int(*compare)(const void *, const void *));
  ```

  Функция **`qsort`** сортирует (по возрастанию) массив с указателем **`buf`**, используя алгоритм быстрой сортировки *Ч.Э.Р. Хоара*, который считается одним из лучших алгоритмов сортировки общего назначения.

  Параметр **`num`** задает количество элементов массива **`buf`**, параметр **`size`** – размер (в байтах) элемента массива **`buf`**.

  Параметр **`int(*compare)(const void *, const void *)`** задает правило сравнения элементов массива **`num`**.

Сортировка

- Функция, указатель на которую передается в **`qsort`** в качестве аргумента, соответствующего параметру **`int(*compare)(const void *, const void *)`**, должна иметь описание:

  `int` ***`имя функции`*** **`(const void *`*`arg1`**`, const void *`*`arg2`**`)`**

  и возвращать:

  - целое < 0, если ***`arg1`*** < ***`arg2`***,
  - целое = 0, если ***`arg1`*** = ***`arg2`***
  - целое > 0, если ***`arg1`*** > ***`arg2`***

Сортировка

- ***Простейший алгоритм сортировки***: сведение сортировки к задаче нахождения максимального (минимального) из *n* чисел.

  Нахождение максимума *n* чисел (*n* сравнений):

  Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

- ***Алгоритм сортировки***: находим максимальное из *n* чисел, получаем последний элемент отсортированного массива (*n* сравнений);
  находим максимальное из *n* – 1 оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще *n* – 1 сравнений); и так далее.
- ***Общее количество сравнений***: 1 + 2 + … + *n*-1 + *n* = *n*(*n* - 1)/2.

  Сложность алгоритма $O(n^2)$.

Сортировка

- ***Классификация алгоритмов сортировки***

  Различают *внешнюю* и *внутреннюю* сортировку.

  Рассматривается только *внутренняя сортировка*: сортируемый массив находится в основной памяти компьютера. *Внешняя сортировка* применяется к записям на внешних файлах.

  3 общих метода внутренней сортировки:

  (1) *сортировка обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;

  (2) *сортировка выборкой*: идея описана на предыдущем слайде

  (3) *сортировка вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

Сортировка

- ***Сортировка обменами (пузырьком)***

  Общее количество сравнений (действий): $n(n - 1)/2$, так как внешний цикл выполняется $(n - 1)$ раз, а внутренний – в среднем $n/2$ раза.

```
void bubble_sort (int *a, int n) {
  int i, j, tmp;
  for (j = 1; j < n; ++j)
    for (i = n - 1; i >= j; --i) {
      if (a[i - 1] > a[i]) {
        tmp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = tmp;
      }
    }
}
```

Сортировка

- ***Сортировка вставками***

  Количество сравнений зависит от степени перемешанности массива **a**. Если массив **a** уже отсортирован, количество сравнений равно *n* - 1. Если массив **a** отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2.

```
void insert_sort (int *a, int n) {
  int i, j, tmp;

  for (j = 1; j < n; ++j) {
    tmp = a[j];
    for (i = j - 1; i >= 0 && tmp < a[i]; i--)
      a[i + 1] = a[i];
    a[i + 1] = tmp;
  }
}
```

Дома. Реализуйте один из простых алгоритмов сортировки в общем виде аналогично функции **qsort**.

Сортировка

- ***Оценка сложности алгоритмов сортировки***
- Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.
- Кроме скорости оценивается «естественность» алгоритма сортировки:
  естественным считается алгоритм, который на уже отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.
- Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма.
  Для рекурсивных алгоритмов расход памяти связан с необходимостью сохранять в автоматической памяти (стеке) локальные переменные и параметры.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!).$$

(a) Алгоритм S можно представить в виде двоичного дерева сравнений.
Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений.
Таким образом, дерево сравнений будет иметь не менее n! листьев.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!). \qquad (*)$$

(б) Для высоты h_m двоичного дерева с m листьями имеет место оценка:

$$h_m \geq \log_2 m.$$

Любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h, а у полного двоичного дерева высоты h 2^h листьев.

Применив полученную оценку к дереву сравнений, получим оценку (*)

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq O(n \cdot \log_2(n))$$

- Доказательство.

(2) К $\log_2(n!)$ применим формулу Стирлинга

$$n! = \sqrt{2\pi n} \cdot n^n e^{-n} e^{\vartheta(n)} \qquad (**)$$

$$|\vartheta(n)| \leq \frac{1}{12n}$$

Логарифмируя (**), получаем

$$\log(n!) = \frac{1}{2}\log(2\pi n) + n \cdot \log(n) - n + \vartheta(n)$$

$$\log(n!) \geq O(n \cdot \log(n))$$

Быстрая сортировка

- ***QuickSort*** **– рекурсивная Си-функция следующего вида:**

```
/* Быстрая сортировка. Предполагается, что left<right */
static void QuickSort (int *a, int left, int right) {
  /* comp – компаранд, i, j – значения индексов */

  int comp, tmp, i, j;
  i = left; j = right;
  comp = a[(left + right)/2]; //можно a[left] или a[right]

  /* построение Partition – цикл do-while */
  do {
    while (a[i] < comp && i < right)
      i++;
    while (comp < a[j] && j > left)
      j--;
    if (i <= j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++, j--;
    }
  } while (i <= j);
  ...
}
```

Быстрая сортировка

- ***QuickSort* – рекурсивная Си-функция следующего вида**:

```
static void QuickSort (int *a, int left, int right) {
  ...

  /* продолжение сортировки, если не все отсортировано */
  if (left < j)
    QuickSort (a, left, j);
  if (i < right)
    QuickSort (a, i, right);
}
```

- Программа быстрой сортировки.

```
void qsort (int *a, int n) {
  QuickSort (a, 0, n - 1);
}
```

- Нужно, чтобы значение компаранда было таким, чтобы он попал в середину результирующей последовательности. Мы пытаемся угадать, какой из элементов массива имеет такое значение. Чем лучше мы угадаем, тем быстрее выполнится алгоритм

Быстрая сортировка

- Покажем, что цикл **`do-while`** действительно строит нужное нам разбиение массива **`a[]`**.

  (1) В процессе работы цикла индексы **`i`** и **`j`** не выходят за пределы отрезка [**`left, right`**], так как в циклах **`while`** выполняются соответствующие проверки.

  (2) В момент окончания работы цикла
  `do-while j ≤ right`,
  так как части разбиения не могут быть пустыми: хотя бы один элемент массива **`a[]`** (в крайнем случае **`a[right]`**) содержится в правой части разбиения.

  (3) Аналогично, в момент окончания работы цикла
  `do-while i ≥ left`.

  (4) В момент окончания работы цикла **`do-while`** любой элемент подмассива **`a[left..j]`** не больше любого элемента подмассива **`a[i..right]`**, что очевидно.

Быстрая сортировка

- Работа цикла **do-while** на примере: **5 3 2 6 4 1 3 7**.
  - Пусть в качестве первого компаранда выбран первый элемент массива – **5** (**a[left]**).
  - Во время первого прохода цикла **do-while** после выполнения обоих циклов **while** получим:

    (5) 3 2 6 4 1 {3} 7;

    (в круглых скобках элемент с индексом **i**,
    в фигурных – элемент с индексом **j**).
  - Поскольку **i < j**, элементы, выделенные скобками, нужно поменять местами (оператор **if**):

    3 (3) 2 6 4 {1} 5 7;
  - В результате второго прохода цикла **do-while** получим:
    до обмена **3 3 2 (6) 4 {1} 5 7;**
    после обмена **3 3 2 1 ({4}) 6 5 7;**
  - Третий проход лишь увеличивает **i**.
  - Теперь массив **a** состоит из двух подмассивов
    3 3 2 1 4 и **6 5 7**
    причем **i = 5**, **j = 4**.
    и нужно рекурсивно применить метод к этим подмассивам.

Быстрая сортировка

- При выборе компаранда можно брать первый элемент, значение которого больше значения следующего элемента. Для результирующих подмассивов из примера компаранды заключены в квадратные скобки:

  3 [3] 2 1 4; **[6] 5 7.**

- Если $f(n)$ и $g(n)$ – некоторые функции, то запись $g(n) = \Theta(f(n))$ означает, что найдутся такие константы $c1, c2 > 0$ и такое n_0, что для всех $n \geq n_0$ выполняются соотношения

  $$0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n).$$

  т.е. при больших n
  $f(n)$ хорошо описывает поведение $g(n)$.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(1) Время выполнения цикла **`do-while`**
$\Theta(n)$, где $n = right - left + 1$.

(2) для алгоритма **`QuickSort`** максимальное (наихудшее) время выполнения $T_{max}(n) = \Theta(n^2)$.

Наихудшее время: при каждом *Partition* массив длины n разбивается на подмассивы длины 1 и $n - 1$.

(2Д) Для $T_{max}(n)$ имеет место соотношение
$T_{max}(n) = T_{max}(n - 1) + \Theta(n)$.
Очевидно, что $T_{max}(1) = \Theta(1)$.
Следовательно,

$$T_{max}(n) = T_{max}(n - 1) + \Theta(n) =$$

$$\sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \quad n \cdot (n - 1)/2 = \Theta(n^2).$$

(3) Если исходный массив **`a`** отсортирован в порядке убывания, время его сортировки в порядке возрастания с помощью алгоритма **`QuickSort`** будет $\Theta(n^2)$.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

(4) Минимальное и среднее время выполнения алгоритма *QuickSort*

$$T_{mean}(n) = \Theta(n \cdot \log n)$$

с разными константами: чем ближе разбиение на подмассивы к сбалансированному, тем константы меньше.

(4Д) Доказательство использует теорему о рекуррентных оценках [1]

(5) Рекуррентное соотношение для минимального (наилучшего) времени сортировки $T_{min}(n)$ имеет вид

$$T_{min}(n) = 2 \cdot T_{min}(n/2) + \Theta(n),$$

так как минимальное время получается тогда, когда на каждом шаге удается выбрать компаранд, который делит массив на два подмассива одинаковой длины $\lceil n/2 \rceil$.

Применяя ту же теорему, получаем $T_{min}(n) = \Theta(n \cdot \log n)$.

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. ISBN 5-900916-37-5, с. 66 – 73.

Быстрая сортировка

- Оценка времени выполнения алгоритма **`QuickSort`**.

  (6) Рекуррентное соотношение для $T(n)$ в общем случае, когда на каждом шаге массив делится в отношении $q:(n-q)$, причем q равновероятно распределено между 1 и n, также можно решить и установить, что $T(n) = \Theta(n \cdot \log n)$ (та же книга, с.160 – 164).