Курс «Алгоритмы и алгоритмические языки»

Лекция 16

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

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

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).

Отладка программ

- Все программы содержат ошибки, отладка – это процесс поиска и удаления некоторых ошибок
- Существуют другие методы обнаружения ошибок (тестирование, верификация, статические и динамические анализаторы кода), но их применение не гарантирует отсутствия ошибок
- Для отладки используют инструменты, позволяющие получить информацию о поведении программы на некоторых входных данных, не изменяя ее поведения
- Простейший метод: *отладочная печать*

```
int *a; int n;
n = read_array (a);
debug_array (a, n);
```

```
static void debug_array (int *a, int n) {
  fprintf (stderr, "Array (%d)", n);
  for (int i = 0; i < n; i++)
    fprintf (stderr, "%d ", a[i]);
  fprintf (stderr, "\n");
}
```

- Отладочная печать может контролироваться макросом (**NDEBUG**)

Отладка программ: отладчики

- Отладчик – основной инструмент отладки программы
- Отладчик позволяет
  - запустить программу для заданных входных данных
  - останавливать выполнение по достижении заданных точек программы безусловно или при выполнении некоторого условия на значения переменных
  - останавливать выполнение, когда некоторая переменная изменяет свое значение
  - выполнить текущую строку исходного кода программы и снова остановить выполнение
  - посмотреть/изменить значения переменных, памяти
  - посмотреть текущий стек вызовов
- Необходимое условие для отладки на уровне исходного кода: наличие в исполняемом файле программы *отладочной информации* (связи между командами процессора и строками исходного кода программы, связь между адресами и переменными и т.д.)

Отладка программ: отладчик **gdb**

- Компиляция с отладочной информацией: **`gcc -g`**
- Некоторые команды **gdb**
  - **`gdb <file> --args <args>`** – загрузить программу с заданными параметрами командной строки
  - **`run/continue`** – запустить/продолжить выполнение
  - **`break <function name/file:line number>`** – завести безусловную *точку останова*
  - **`cond <bp#> condition`** – задать условие остановки выполнения для некоторой точки останова
  - **`watch <variable/address>`** – задать *точку наблюдения* (остановка выполнения при изменении значения переменной или памяти по адресу)
  - **`next/step`** – выполнить текущую строку исходного кода программы без захода/с заходом в вызываемые функции
  - **`print <var>/set <var> = expression`** – посмотреть /изменить текущие значения переменных, памяти
  - **`bt`** – посмотреть текущий стек вызовов
- Среда Code::Blocks поддерживает **gdb** в своем интерфейсе

Отладка программ: примеры команд gdb

- Установка точек останова

```
b fancy_abort
b 7199
b sel-sched.c:7199
cond 2 insn.u.fld.rt_int == 112
cond 3 x_rtl->emit.x_cur_insn_uid == 1396
```

- Просмотр и изменение значений переменных

```
p orig_ops.u.expr.history_of_changes.base
p bb->index
set sched_verbose=5
call debug_vinsn (0x4744540)
```

- Установка точек наблюдения

```
wa can_issue_more
wa ((basic_block) 0x7ffff58b5680)->preds.base.prefix.num
```