## *Лекция 12 Быстрая сортировка*

12.1. ***QuickSort: программа на Си***.

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

```
/* Быстрая сортировка. Предполагается, что left < right */
void QuickSort (int *a, int left, int right) {
    /* comp – компаранд, i, j – значения индексов элементов
     массива a */
   int comp, tmp, i, j;
    /* выбор компаранда */
   i = left; j = right;
   comp = a[(left + right)/2]; //можно и 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);
    /* продолжение сортировки, если не все отсортировано */
   if(left < j)
     QuickSort (*a, left, i - 1);
   if(i < right)
     QuickSort (*a, j + 1, right);
}
```

12.1.2. Программа быстрой сортировки.

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

12.1.3. Цикл **do-while** (или **do**). В отличие от цикла **while** сначала выполняется тело цикла, а потом проверяется условие выхода из цикла. В рассматриваемой программе это **do {**⟨*тело цикла*⟩**} while(i <= j);**

12.1.4. Объяснение работы программы на рисунке: массив **a[]** и его индексы.

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** $j \le right$, так как части разбиения не могут быть пустыми: хотя бы один элемент массива **a[]** (в крайнем случае **a[right]**) содержится в правой части разбиения. Аналогично, в момент окончания работы цикла **do-while** $i \ge left$.

(3) В момент окончания работы цикла **do-while** любой элемент подмассива **a[left..k-1]** не больше любого элемента подмассива **a[k+1.. right]** , где **k** – индекс компаранда, что очевидно.

12.1.5. 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**; (выделены уже сформировавшиеся куски массива **a**). В результате второго прохода цикла **do-while** получим: до перемены мест **3 3 2 (6) 4 {1} 5 7**; а после обмена **3 3 2 1 (4) {6} 5 7**; Теперь массив **a** состоит из двух подмассивов **3 3 2 1 4** и **6 5 7** причем $i < j$ ($i = 5, j = 6$). Теперь нужно применить метод к этим подмассивам (рекурсивные вызовы).

12.1.6. При выборе компаранда можно брать первый элемент, значение которого больше значения следующего элемента. Для результирующих подмассивов из п.°12.1.5 компаранды заключены в квадратные скобки: **3 [3] 2 1 4**; **[6] 5 7**.

12.2. Оценка времени выполнения алгоритма *QuickSort*.

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

***Замечание***. Если $f(n)$ и $g(n)$ – некоторые функции, то запись $g(n) = \Theta(f(n))$ означает, что найдутся такие константы $c_1$, $c_2 > 0$ и такое $n_0$, что для всех $n \geq n_0$ выполняются соотношения $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$. Иными словами при больших $n$ $f(n)$ хорошо описывает поведение $g(n)$. Наше утверждение означает, что неизвестная функция $t_{Part}(n)$ (время построения *Partition*) ведет себя как $c \cdot n$, где $c$ – положительная константа.

12.2.2. Можно доказать, что для алгоритма *QuickSort* максимальное (наихудшее) время выполнения $T_{max}(n) = \Theta(n^2)$. Наихудшее время получается, когда при каждом *Partition* массив длины $n$ разбивается на подмассивы длины 1 и $n - 1$. В самом деле, для $T_{max}(n)$ имеет место соотношение $T_{max}(n) = T_{max}(n - 1) + \Theta(n)$. Очевидно, что $T_{max}(1) = \Theta(1)$. Следовательно,

$$T_{max}(n) = T_{max}(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = n\cdot(n-1)/2 = \Theta(n^2).$$

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

12.2.3. 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. Доказательство использует **теорему о рекуррентных оценках** [1]

12.2.4. 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.

12.2.5. Рекуррентное соотношение для ($n$) в общем случае, когда на каждом шаге массив делится в отношении $q$:($n - q$), причем $q$ с вероятностью 2/ принимает значение 1 и с вероятностями 1/$n$ значения 2, …, $n - 1$, имеет вид

$$T(n) = \frac{1}{n}\left( T(1) + T(n-1) + \sum_{q=1}^{n-1} (T(q) + T(n-q)) \right) + \Theta(n).$$

Достаточно сложные рассуждения позволяют решить это соотношение и установить, что $T(n) = \Theta(n \cdot \log n)$ (та же книга, с.160 – 164).

12.2.6. Более того, упомянутые методы позволяют доказать, что не существует алгоритма сортировки массива из $n$ элементов,

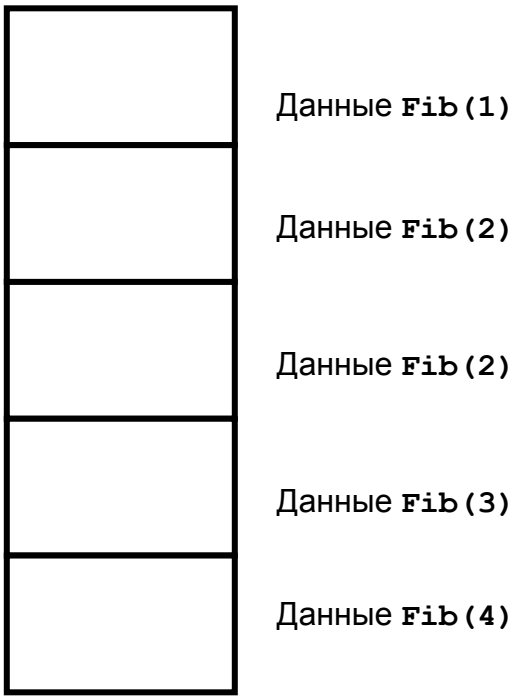
12.3. ***Как в системе программирования Си выполняются рекурсивные функции?***

12.3.1. В Си разрешается, чтобы функция вызывала сама себя. Такая функция называется рекурсивной. *QuickSort* – пример рекурсивной функции. Более простой пример – рекурсивная функция, вычисляющая числа Фибоначчи.

12.3.2. Числа Фибоначчи возникли в решении задачи о кроликах, предложенном в XIII веке Леонардо из Пизы, известным как Фибоначчи. **Задача**: пара новорожденных кроликов помещена на остров. Каждый месяц любая пара дает приплод – также пару кроликов. Пара начинает давать приплод в возрасте двух месяцев. Сколько кроликов будет на острове через $n$ месяцев? В конце первого и второго месяцев на острове будет одна пара кроликов: $f_1 = 1, f_2 = 1$. В конце третьего месяца родится новая пара, так что $f_3 = f_2 + f_1$. По индукции можно доказать, что для $n \geq 3$ $f_n = f_{n-1} + f_{n-2}$. Рекурсивная программа:

```
/* Вычисление n-го числа Фибоначчи */
int Fib(int n) {
   if (n < 1)
      return (0) // неверные начальные данные
   if ((n == 1) || (n == 2))
      return 1;
   else
      return (Fib(n - 1) + Fib (n - 2));
}
```

12.3.3. Рекурсивная функция выполняется сложнее, чем не рекурсивная. Пусть, например, $n$ = 4. Вызов **`Fib(4)`** приведет к вызову **`Fib(3)`** и **`Fib(2)`**, вызов **`Fib(3)`** – к вызову **`Fib(2)`** и **`Fib(1)`**. При каждом вызове будет выполняться *одна и та же* функция, но над разными данными (копий кода функции не создается). Данные размещаются в стеке как показано на рисунке.




12.3.4. Не рекурсивная программа (в большинстве случаев рекурсию нетрудно заменить итерацией).

```
int Fbn(int n) {
   if ((n == 1) || (n == 2))
     return 1;
   else {
     g = h = 1;
     for(k = 2; k < n; k++) {
          Fb = g + h;
          h = g;
          g = Fb;
     }
     return Fb;
   }
}
```