Бинарный поиск; работа со строками

А. Г. Фенстер, fenster@fenster.name

5 марта 2010 г.

Конспект семинара №4 по программированию для студентов 1 курса ММФ НГУ. Помните, что чтение конспекта не делает посещение соответствующего семинара необязательным! О найденных ошибках и неточностях, пожалуйста, пишите мне по адресу fenster@fenster.name. Также буду рад получить любые комментарии по поводу этого текста.

1 Бинарный поиск

Бинарный поиск применяется для поиска числа в **упорядоченном** (для определённости, по возрастанию) массиве. Если известно, что элементы массива упорядочены, то можно сравнить искомое число с числом в середине массива; возможны три варианта:

1. В середине массива находится искомое число — поиск завершён успешно;
2. Число в середине массива больше искомого — необходимо произвести поиск в левой половине массива;
3. Число в середине массива меньше искомого — необходимо произвести поиск в правой половине массива.

Важно заметить, что при переходе к левой или правой частям массива **не нужно** включать в них середину, потому что мы уже проверили, что в середине массива стоит **не** искомый элемент.

Запишем рекурсивный алгоритм решения задачи. Более того, сделаем это двумя способами: сначала реализуем функцию, которая возвращает индекс найденного элемента (и -1 в случае, если искомый элемент

не найден), а затем сделаем то же самое, но более активно используя указатели. Массив `A` считаем упорядоченным по возрастанию.

```
int binary_search(int *A, int l, int r, int k)
/* ищем в упорядоченном по возрастанию массиве A в диапазоне от l до r число k */
{
    int mid;
    if (l > r)  /* терминальная ветвь рекурсии */
        return (-1);
    mid = (l + r) / 2;
    if (A[mid] == k)
        return mid;
    if (A[mid] > k)
        return binary_search(A, l, mid - 1, k);
    return binary_search(A, mid + 1, r, k);
}
```

Отметим несколько важных моментов.

Во-первых, как уже говорилось чуть выше, при переходе к половине массива средний элемент (с индексом `mid`) не включается в диапазон поиска.

Во-вторых, последний вариант не требует написания отдельного условия, т. к. к последней строке функции мы приходим только в том случае, если не выполнены оба предыдущих условия, следовательно, в конце функции автоматически выполнено `A[mid] < k`.

Наконец, в-третьих, хоть данная рекурсия выглядит чуть сложнее, чем обычная хвостовая рекурсия, она по сути также является хвостовой: во всех случаях рекурсивный вызов будет последним. Следовательно, этот алгоритм можно переписать без рекурсии и дополнительной памяти (это необходимо сделать в одной из задач на практических занятиях).

Пусть теперь наша функция возвращает не индекс, а адрес найденного элемента массива. В случае, если искомый элемент в массиве отсутствует, будем возвращать константу `NULL` — нулевой адрес; гарантируется, что он не совпадает с адресом никакой из ячеек памяти.

```
int *binary_search(int *A, int len, int k)
{
    int mid;
    if (len == 0)
        return NULL;
    mid = len / 2;
    if (A[mid] == k)
        return A + mid;
    if (A[mid] > k)
        return binary_search(A, mid, k);
    return binary_search(A + mid + 1, len % 2 == 0 ? mid - 1 : mid, k);
}
```

В этом варианте мы считаем, что диапазон поиска всегда начинается с нулевого элемента массива. Здесь возникают проблемы при вычислении длины правой части массива: в зависимости от того, нечётное количество элементов в массиве или чётное, длины правой и левой частей могут совпадать или отличаться на единицу. *Условное выражение* `условие ? значение-если-истина : значение-если-ложь` помогает оформить такую логику без лишних инструкций `if`.

Чтобы по адресу некоторого элемента массива узнать его индекс в массиве, необходимо вычесть из этого адреса адрес начала массива, например, так:

```
int *found = binary_search(A, 5, 7);
if (found) /* если не равно нулю.  NULL - ноль, то есть ложь */
{
    printf("Найдено: элемент номер %d\n", found - A);
}
else
{
    printf("Не найдено.\n");
}
```

Конечно, можно объединить присваивание с условием:

```
if (found = binary_search(A, 5, 7))
    ...
```

2 Сложность алгоритмов

Бинарный поиск будет первым алгоритмом, на примере которого мы познакомимся с понятием *временно́й сложности* алгоритма. Оценим количество операций, которое будет выполнено процессором при выполнении этого алгоритма. Видим, что на каждом шаге алгоритма (в каждом вызове рекурсивной функции) выполняются операции, время работы которых не зависит от размера рассматриваемого массива и может быть ограничено сверху числом c. Далее, на каждом шаге от массива длины N мы переходим к массиву длины либо $\frac{N}{2}$, либо $\frac{N}{2} - 1$, то есть в два раза меньшему. Делить массив длины N пополам можно не более чем $[\log_2 N] + 1$ раз ($[\,\cdot\,]$ — целая часть), а значит, общее время работы алгоритма ограничено в худшем случае числом $c \cdot ([\log_2 N] + 1)$, то есть имеет порядок $O(\log_2 N)$.

Легко представить алгоритм, имеющий *линейную* сложность $O(N)$. Таким алгоритмом является, например, поиск минимального элемента

массива: операция сравнения (требующая фиксированного времени) будет выполнена $N-1$ раз, что ограничит общее количество выполняемых операций числом $c \cdot (N-1)$.

Алгоритм, выполняющий поиск количества пар элементов массива, нарушающих порядок (таких, что $i < j$, но $A[i] > A[j]$ — такие пары называются *инверсиями*), может быть реализован при помощи двух вложенных циклов `for` и будет иметь *квадратичную* сложность $O(N^2)$: в простой реализации

```
for (i = 0; i < N - 1; i++)
    for (j = i + 1, j < N; j++)
        if (A[i] > A[j])
            count++;
```

условие `if` проверяется

$$(N-1)+(N-2)+\ldots+1 = \frac{1+(N-1)}{2} \cdot (N-1) = \frac{N^2}{2} - \frac{N}{2} = O(N^2) \text{ раз.}$$

В дальнейшем мы будем пытаться оценивать сложность любого из рассматриваемых нами алгоритмов.

3 Работа со строками

В языке C нет специального типа «строка» (как `string` в паскале). Строки представляются в виде массивов символов, т. е. элементов типа `char` или `unsigned char`. Для хранения строки из N символов требуется $N+1$ элемент массива: в последнем (N-ом) элементе должен быть записан символ с кодом 0.

Строка может быть инициализирована как массив (перечислением элементов) или при помощи двойных кавычек. Следующие четыре записи эквивалентны и приводят к появлению в памяти массива, изображённого на рисунке ниже:

```
char s[] = { 'v', 'a', 's', 'y', 'a', 0 };
char s[6] = { 'v', 'a', 's', 'y', 'a', 0 };
```

```
char s[] = "vasya";
char s[6] = "vasya";
```

'v'	'a'	's'	'y'	'a'	0

Все стандартные функции, принимающие на вход строку, останавливаются, «увидев» символ с кодом 0, в том числе функция `printf` (для вывода строки используйте `%s`). Собственно, единственный способ узнать длину строки — пройти по ней циклом и найти в ней элемент с нулевым значением. Примерно так работает функция `strlen`, вычисляющая длину строки (для использования подключите `string.h`):

```
int strlen(char *s)
{
    int i;
    for (i = 0; s[i]; i++)  /* пока s[i] не равно нулю */
        ;                   /* пустое тело цикла */
    return i;
}
```

На самом деле функция `strlen` возвращает не `int`, а `size_t` — специальный тип для хранения размеров объектов в памяти, обычно соответствующий беззнаковому длинному целому. На машинах с 64-битной архитектурой диапазон значений типа `size_t` обычно больше диапазона значений `int`, однако для нас в настоящий момент это не так важно.

Другой вариант реализации `strlen`:

```
int strlen(char *s)
{
    char *p = s;
    while (*p)
        p++;
    return (p - s);
}
```

Цикл `while` можно (возможно, с некоторой потерей читаемости) записать даже проще:

```
    while (*p++)
        ; /* пустое тело цикла */
```

Выражение `*p++` вычисляется следующим образом: к `p` применяется операция разыменования и её результат становится значением всего выражения, а `p` затем увеличивается на единицу. Это свойство **постфиксного** оператора `++` (который ставится после переменной). Префиксный оператор `++` (перед переменной) вернёт новое (уже увеличенное) значение. В этом случае `p` уйдёт на одну клетку правее, так что результатом функции будет `p - s - 1`.

Для копирования одной строки в другую можно использовать функцию `strcpy`, которая реализована примерно так:

```
char *strcpy(char *destination, char *source)
/* согласно стандарту, возвращает адрес начала строки, в которую копируем */
{
    int i;
```

```
    for (i = 0; source[i]; i++)
        destination[i] = source[i];
    destination[i] = 0;
    return destination;
}
```

Обратите внимание на необходимость отдельно поставить в конец строки `destination` символ с кодом 0.

Вариант `strcpy`, использующий указатели, можно назвать классическим: этот пример можно встретить чуть ли не в любой книге по языку C.

```
char *strcpy(char *destination, char *source)
{
    char *p = destination; /* сохранили, чтобы потом вернуть */
    while (*destination++ = *source++)
        ; /* пустое тело цикла */
    return p;
}
```

В условии цикла `while` выполняются следующие операции:

1. Значение ячейки памяти, на которую указывает `destination`, становится равным значению ячейки памяти, на которую указывает `source`;
2. Результатом выражения становится присвоенное значение;
3. Оба указателя сдвигаются вправо на одну клетку каждого из массивов.

Если результатом выражения является 0 (а это происходит при присваивании последнего, завершающего символа с кодом 0), цикл останавливается.

Важно понимать, что функция `strcpy` не может узнать, сколько места есть в массиве `destination`. Позаботиться о том, чтобы места хватило — забота программиста. Существует более безопасная функция `strncpy`, которая принимает третий параметр — ограничение на количество символов для копирования:

```
char *strncpy(char *destination, char *source, int n);
```

но здесь тоже нужно быть внимательным: если длина строки `source` превышает `n`, строка `destination` не будет завершена нулём — это нужно будет проверять самостоятельно.

Задача. Реализуйте функцию, преобразующую строку, состоящую из цифр, в число (по аналогии с функцией `read_integer` с предыдущих занятий, но цифры берутся из `char *`).

4 Проверочное задание («пятиминутка»)

Тема: строки (см. задание).