Программирование и основы алгоритмизации

Лекция 3

Основные алгоритмы.

Поиск и сортировка

Алгоритмы и структуры данных

- Исследование алгоритмов и структур данных является одной из основ программирования, а также богатым полем элегантных технологий и сложных математических изысканий.
- Хороший алгоритм или структура данных могут позволить решить в течение нескольких секунд проблему, которая без них решалась бы годы
- В таких специальных областях, как графика, базы данных, синтаксический разбор, цифровой анализ и моделирование, возможность решения задачи целиком и полностью зависит от наличия специальных алгоритмов и структур данных
- Если вы разрабатываете программы в новых для вас областях программирования, то вы должны выяснить, какие наработки уже существуют, иначе вы потратите свое время впустую в попытках плохо сделать то, что уже кем-то было сделано хорошо

Структуры данных

- Массив – одномерный или двумерный
- Структура или объединение в С
- Списки
  - Последовательные (массивы)
  - Связанные (на основе указателей и динамического распределения памяти)
    - Стеки
    - Очереди
    - Иерархические структуры (деревья)
    - Сети
- Хэш-таблицы

Алгоритм Евклида нахождения наибольшего общего делителя двух чисел

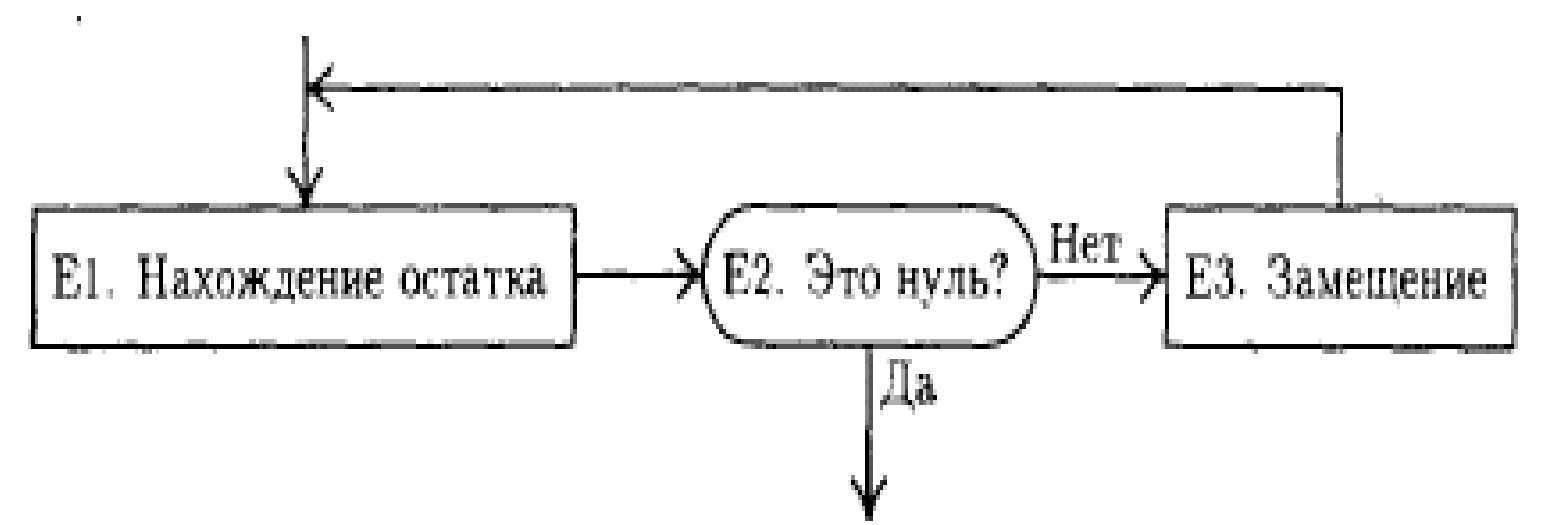
Алгоритм Е (*Алгоритм Евклида*). Даны два целых положительных числа m и n. Требуется найти их *наибольший общий делитель*, т. е. наибольшее целое положительное число, которое нацело делит оба числа m и n.

Е1. [Нахождение остатка.] Разделим m на n, и пусть остаток от деления будет равен r (где $0 \leq r < n$).

Е2. [Сравнение с нулем.] Если $r = 0$, то выполнение алгоритма прекращается; n — искомое значение.

Е3. [Замещение.] Присвоить $m \leftarrow n$, $n \leftarrow r$ и вернуться к шагу Е1. ∎

Блок-схема алгоритма Евклида



Так не пишите. Это не современная блок-схема и не по ГОСТу

К алгоритму можно добавить:

E0. [Гарантировать, что $m \geq n$.] Если $m < n$, то выполнить взаимный обмен $m \leftrightarrow n$.

Простейший алгоритм поиска слова в массиве (последовательный)

```
char *flab[] = {
    "actually",
    "just",
    "quite",
    "really",
    NULL
};
```

```
/* lookup: последовательный поиск слова в массиве */
int lookup(char *word, char *array[])
{
    int i;
    for (i = 0; array[i] != NULL; i++)
        if (strcmp(word, array[i]) == 0)
            return i;
    return -1;
}
```

Функция возвращает индекс слова

Конец массива обозначается значением NULL.
Можно использовать для организации цикла известное количество элементов в массиве

Сложность и эффективность алгоритма

- Рассмотренный алгоритм не эффективен в случае большого массива (работает долго)
- Для оценки эффективности алгоритма введено понятие сложности (вычислительной сложности) алгоритма
- Для рассмотренного алгоритма обозначается как O(N). Означает, что время выполнения алгоритма прямо пропорционально значению N – длине массива, хотя более точно будет O(N*L), где L – средняя длина слов в массиве
- Для случая последовательного поиска слова в двумерном массиве (матрице) сложность равна O(N*M*L), где N и M – размерности массива
- При разработке программ, работающих с большими объемами данных разработка (или выбор) эффективных алгоритмов является основной задачей

- Анализ алгоритмов
  - Получение количественных характеристик алгоритма
- Теория алгоритмов
  - Рассматриваются вопросы существования и не существования алгоритмов для вычисления определенных величин
  - Берет начало с машины Тьюринга
  - Тесно связана с теорией автоматов и комбинаторикой (дискретной математикой)
- Численные методы (вычислительная математика)
  - Алгоритмы приближенного решения на компьютере вычислительных задач, для которых не существует точных алгоритмов (интегрирование, решение дифференциальных и трансцендентных уравнений)

Двоичный поиск в упорядоченном массиве

Найти заданное число 90 в массиве чисел

1, 3, 39, 41, ***56***, 89, 90,99, 100, 101 — смотрим число посередине,

90>56 — во второй половине,

89, 90,***99***, 100, 101 — см. среднее число

90<99 — в первой половине

89, ***90*** — см. среднее число

Найдено

Двоичный поиск в упорядоченном массиве (2)

```
int binary(int c[], int n, int val)
{                          // Массив, размерность и искомое значение
int a,b,m;                                  // Левая, правая границы и
for(a=0,b=n-1; a <= b;)                                    // середина
    {
    m = (a + b)/2;                          // Середина интервала
    if (c[m] == val)                       // Значение найдено -
    return(m);                               // вернуть индекс
    if (c[m] > val)
        b = m-1;                           // Выбрать левую половину
    else
        a = m+1;                           // Выбрать правую половину
    }
return(-1);                                 // Значение не найдено
}
```

Задание на дом – переделать этот алгоритм к нашему случаю поиска слова в упорядоченном массиве, показать на лабе

Двоичный поиск в упорядоченном массиве (3). Рекурсивная функция

```
int search (int& x[], int l, int u, int keys)
    //предполагается, что массив x отсортирован
    //ищется x[m] равный keys,
    //если l<=m<=u, то возвращается m,
    //иначе возвращается -1
    {
    int m=(l+u)/2;
    if (l>u) return (-1);
    if (x[m]==keys) return (m);
    if (keys<x[m]) return (search(x,l,m,keys));
    else return (search(x,m,u,keys));
    }
```

Эффективность этого алгоритма

- Сложность этого алгоритма $O(\log_2 N)$
- Но необходимо предварительно отсортировать массив

Сравнение сложности последовательного и бинарного поиска

n	**$\log_2 n$**
1	0
16	4
256	8
4096	12
65536	16
1048476	20

Сортировка методом пузырька

1, 41, 3, 56, 90,99, 100, 89,, 39, 101
3, 41 89, 100 просмотр пар и
39, 100 перестановка если a[i]<a[i-1]
1, 3, 41, 56, 90, 99, 89, 39, 100, 101
89, 99 еще
39, 99
1, 3. 41. 56. 90, 89, 39, 99, 100, 101
89, 90 еще
39, 90
1, 3, 41, 56, 89, 39, 90, 99, 100, 101
39, 89 еще
1, 3, 41, 56, 39, 89, 90, 99, 100, 101
39, 56 еще
1, 3, 41, 39, 56, 89, 90, 99, 100, 101
39, 41 еще
1, 3, 39, 41, 56, 89, 90, 99, 100, 101 еще раз, и нет перестановок
– признак окончания цикла

Сортировка методом пузырька (2)

```
void sort(int A[], int n)
{
int i,found;                          // Количество сравнений
    do {                              // Повторять просмотр...
    found =0;
    for (i=0; i<n-1; i++)
        if (A[i] > A[i+1])            // Сравнить соседей
        {                             // Переставить соседей
        int cc;
        cc = A[i]; A[i]=A[i+1]; A[i+1]=cc;
        found++;
        }
    } while(found !=0);               //...пока есть перестановки
}
```

Верхняя граница сложности $O((N-1)^2)$

Алгоритм быстрой сортировки

```
//-------"Быстрая" сортировка
void sort(int in[], int a, int b)
{
int i,j,mode;
if (a>=b) return;
for (i=a, j=b, mode=1; i < j; mode >0 ? j-- : i++)
    if (in[i] > in[j])
    {
    int c;
    c = in[i]; in[i] = in[j]; in[j]=c;
    mode = -mode;
    }
sort(in,a,i-1);
sort(in,i+1,b);
}
```

Алгоритм быстрой сортировки (2)

1, 41, 3, 56, 90,99, 100, 89,, 39, 101 i=0, j=9
1<101 перестановки нет, j=9-1=8
1<39 перестановки нет, j=8-1=7
……
1 остается на своем месте, sort(in, 0, -1), sort(in, 1, 9) рекурсия
41<101 перестановки нет, j=9-1=8
41>39 перестановка i=1+1=2
1, **39**, 3, 56, 90,99, 100, 89,, **41**, 101
3<41 перестановки нет, i=2+1=3
56>41 перестановка j=8-1=7
1, 39, 3, **41**, 90,99, 100, 89,, **56**, 101
41<89 перестановки нет
…….
41 остается на месте sort(in, 0, 2), sort(in, 4, 9)
Sort(in, 0, 2) обеспечивает сортировку 1-ой части 1, 39, 3
Sort(in, 4, 9) обеспечивает сортировку второй части
90,99, 100, 89,, 56, 101