Информатика

Базовые алгоритмы #2

© Марченко Антон Александрович 2016 г.
Абрамский Михаил Михайлович

На прошлой лекции…

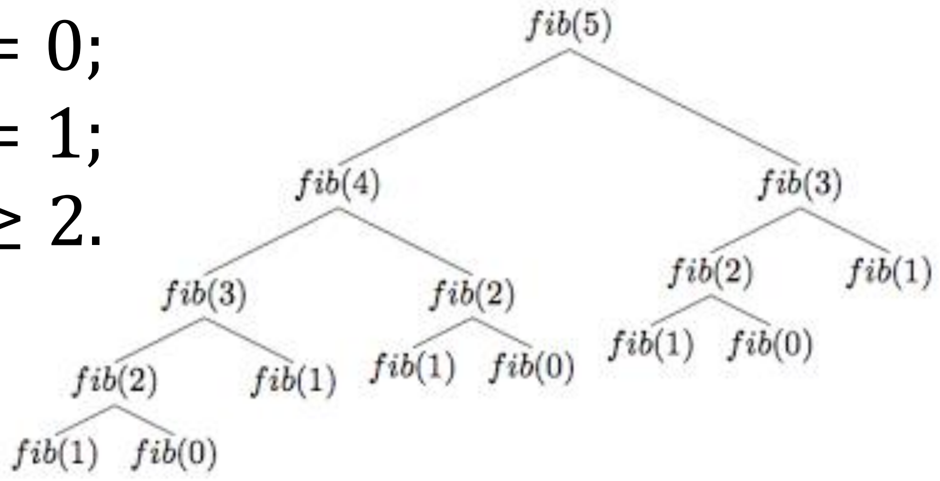
- **Рекурсия, итерация**
- **Декомпозиция «разделяй и властвуй»**

Декомпозиция

Алгоритмы **«разделяй и властвуй»**: *задачу* ***рекурсивно разбивают на несколько*** *более простых* ***подзадач****, и* ***получают решение комбинацией*** *получившихся решений*

Вспомним числа Фибоначчи

$$F_n = \begin{cases} 0, & n = 0; \\ 1, & n = 1; \\ F_{n-1} + F_{n-2}, & n \geq 2. \end{cases}$$



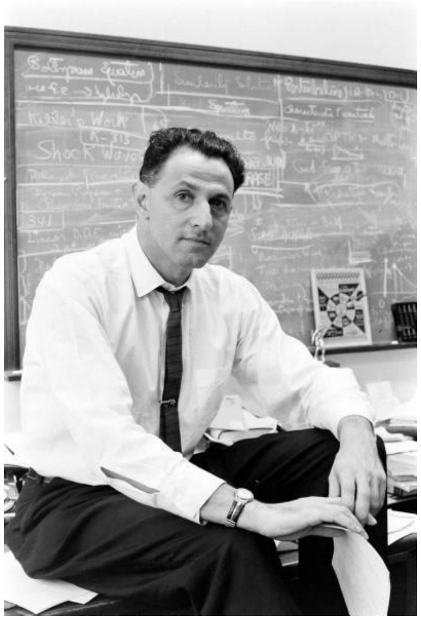
- **Многo повторных вычислений**
- Сложность $F_n \sim \frac{\varphi^n}{\sqrt{5}}$, где $\varphi = \frac{1+\sqrt{5}}{2}$

Развитие идей декомпозиции

- Использовать разбиение на подзадачи
- Получать решение задачи комбинацией решений подзадач
- **Решать каждую подзадачу один раз!**

Динамическое программирование

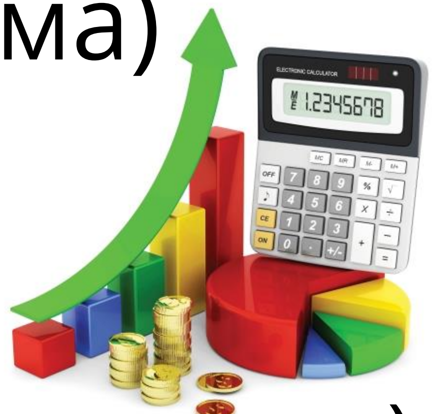
- Теория Ричарда Беллмана (1940-е)
- Bellman equation (уравнение Беллмана)
  - ***решение задач оптимизации с оптимальной подструктурой*** последовательным решением подзадач



  - В дискретном случае – рекуррентное соотношение

Задача оптимизации

- Нахождение экстремума (минимума или максимума) целевой функции
  - Примеры из жизни:
    - минимизация затрат (деньги, время)
    - максимизация прибыли



Оптимальная подструктура

- Частный случай оптимизации
- *Оптимальное решение подзадач* меньшего размера может использоваться *для* получения *оптимального решения* исходной *задачи*

Основная идея

- ***Запоминать решения подзадач* (Мемоизация)**
- *Использовать* известные *результаты вместо повторного* нахождения *решения* подзадач
- ***Жертвуем памятью ради времени***

Числа Фибоначчи

- Запоминаем все известные результаты, используем

```
static int Fib(int n)
{
    int[] f = new int[n+1];
    f[0] = 0; f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}
```

Рекурсия с мемоизацией

- Заводим массив для всех возможных решений подзадач
- Записываем начальные известные решения
- Используем массив в рекурсивных функциях для обращения к известным решениям

Рекурсия с мемоизацией

```
static int Fib(int n)
{
    var cache = new int[n + 1];
    for(int i=0; i<=n;i++)
        if (i < 2) cache[i] = i;
        else cache[i] = -1;
    return FibRec(cache, n);
}
static int FibRec(int[]cache, int n)
{
    if (cache[n]>=0)
        return cache[n];
    cache[n]=FibRec(cache, n - 1) + FibRec(cache, n - 2);
    return cache[n];
}
```

Динамическая мемоизация

- Идея – выделять память по необходимости, запоминать решения по мере появления
- Использование словаря – доступ по ключу к значению за O(1)

Динамическая мемоизация

```
static int Fib(int n)
{
    var cache = new Dictionary<int, int> {{0, 0}, {1, 1}};
    return FibRec(cache, n);
}
static int FibRec(Dictionary<int,int> cache, int n)
{
    if (cache.ContainsKey(n))
        return cache[n];
    cache.Add(n,FibRec(cache, n - 1) + FibRec(cache, n - 2));
    return cache[n];
}
```

Задача: Лесенка

- С вершины лесенки из N ступенек начинает прыгать к основанию мячик.
- Мячик может прыгнуть на следующую ступеньку или через одну.
- Определить количество всевозможных «маршрутов» мячика

Задача «Лесенка»

- На первую ступеньку попадаем одним способом – следующая с вершины
- На вторую – двумя (следующая с первой через одну с вершины)
- На третью – тремя (следующая со второй(2 способа), через одну с первой)
- Итого L(1)=1, L(2)=2, L(N)=L(N-1)+L(N-2)
- **Другая интерпретация чисел Фибоначчи**

Модификация «Лесенки»

- С вершины лесенки из N ступенек начинает прыгать к основанию мячик.
- Мячик может прыгнуть на следующую ступеньку или через две.
- Определить количество всевозможных «маршрутов» мячика

Модифицированная «Лесенка»

$$L(1) = 1, L(2) = 1, L(3) = 2$$
$$L(N) = L(N-1) + L(N-3)$$

```
static int Ladder(int n)
{
    switch (n)
    {
        case 1:
        case 2:
            return 1;
        case 3:
            return 2;
        default:
            return Ladder(n - 1) + Ladder(n - 3);
    }
}
```

Модифицированная «Лесенка»

Динамика с использованием массива

```
static int Ladder(int n)
{
    int[] l = new int[n + 1];
    l[1] = 1;
    l[2] = 1;
    l[3] = 2;
    for (int i = 4; i <= n; i++)
        l[i] = l[i - 1] + l[i - 3];
    return l[n];
}
```

Двумерная динамика

- В прямоугольной таблице $N \times M$ игрок находится в левой верхней клетке.
- Разрешается перемещаться в соседнюю клетку вправо либо вниз.
- В некоторых ячейках есть стены, через которые ходить нельзя.
- Посчитайте, сколько есть способов у игрока попасть в правую нижнюю клетку.

Количество путей в лабиринте

```
var field = new [,]
{
    { 1, 0,-1, 0},
    {-1, 0, 0,-1},
    { 0, 0, 0, 0},
    { 0,-1, 0, 0}
};
```

Количество путей в лабиринте

- Field[0,0]=1
- Пробегаем по всем строкам и столбцам
  - Field[i,j]=0
  - Если i>0 и в Field[i-1,j] не стена, Field[i,j] += Field[i-1,j]
  - Если j>0 и в Field[i,j-1] не стена, Field[i,j] += Field[i,j-1]
- В Field[N,M] ответ

Количество путей в лабиринте

```
for (int i=0; i<field.GetLength(0); i++)
    for(int j=0; j<field.GetLength(1); j++)
    {
        if (field[i, j] < 0) continue;
        if (i > 0 && field[i-1,j]>0)
            field[i, j] += field[i - 1,j];
        if (j > 0 && field[i, j - 1] > 0)
            field[i, j] += field[i, j - 1];
    }
```

Несколько примеров

- Наибольшая увеличивающаяся последовательность
- Наибольшая общая последовательность
- Расстояние Левенштейна

Расстояние Левенштейна

- Даны две строки S_1 и S_2
- Вычисляем редакционное расстояние $d(S_1, S_2)$
- Операции: Удаление, Вставка, Замена

Расстояние Левенштейна

- Рекуррентная формула

$d(S_1, S_2) = D(M, N)$, где

$$D(i,j) = \begin{cases} 0\ ;\ i = 0, j = 0 \text{ (левый верхний)} \\ i\ ;\ j = 0, i > 0 \text{ (первый столбец)} \\ j\ ;\ i = 0, j > 0 \text{ (первая строка)} \\ \min\begin{Bmatrix} D(i, j-1) \\ D(i-1, j) + 1, \\ D(i-1, j-1) + m\left(S_{1[i]}, S_{2[j]}\right) \end{Bmatrix} ;\ j > 0, i > 0 \end{cases}$$

где $m(a, b) = 0$, если $a = b$ и единица в противном случае

Расстояние Левенштейна

- В $\min\left\{\begin{array}{c} D(i, j-1) \\ D(i-1, j)+1, \\ D(i-1, j-1)+m\left(S_{1[i]}, S_{2[j]}\right) \end{array}\right\}; j>0, i>0$:
  - шаг по i – удаление из S_1
  - шаг по j – вставка в S_1
  - шаг по обоим индексам – замена или отсутствие изменений (в зависимости от m)

Расстояние Левенштейна

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Жадные алгоритмы

- *Принятие локально оптимальных решений на каждом этапе, допуская, что конечное решение окажется оптимальным*
- Можно доказать по инструкции корректность на каждом шаге

Условия применимости

- **Принцип жадного выбора**
  - последовательность локально-оптимальных выборов даёт глобально оптимальное решение
- **Оптимальность для подзадач**
  - оптимальное решение задачи оптимально для всех её подзадач

Greedy vs Dynamic

- **Жадный алгоритм** – нахождение оптимальных решений подзадач с получением глобального оптимума в итоге
  - частный случай динамического программирования
  - *выбираем одно решение, не проверяя другие*
- **Динамическое программирование** – применимо к задачам с перекрывающимися подзадачами и оптимальной подструктурой
  - *выбираем лучшее среди всех решений подзадач*

Задача о размене

Имея неограниченное количество *монет разных номиналов* $a_1 < a_2 < \cdots < a_n$ требуется **выдать сумму** $\boldsymbol{S}$ **наименьшим количеством монет**

Пример: Разменять 27 рублей монетами в 1, 2, 5 и 10 рублей

Задача о размене

- «Жадное» решение:
  - 2 по 10, 1 по 5, 1 по 2
- На каждом шаге берётся наибольшее возможное количество монет достоинства a_n

Оптимальность

- В канонических монетных системах жадный алгоритм для задачи размена всегда даёт оптимальный результат
- Но, в общем случае, жадный алгоритм не всегда даёт оптимальное решение

Контр пример для жадного алгоритма

- Выдать сумму 24 рубля минимальным числом монет достоинством 1, 5 и 7 рублей.
- Жадным алгоритмом:
  3 по 7, 3 по 1 = 6 монет
- Оптимальное решение:
  2 по 7, 2 по 5 = 4 монеты

Задача о рюкзаке

- Задача о рюкзаке – укладка максимального числа ценных вещей в рюкзак при ограниченной вместимости
- Задача о размене – частный случай задачи о рюкзаке

Задача о рюкзаке

- В общей постановке является NP-полной
  - Нет полиномиального алгоритма, решающего её за разумное время
- Нужно выбирать между точными и медленными или быстрыми и приближёнными алгоритмами

Жадный алгоритм
для задачи о рюкзаке

- Сортируем предметы по убыванию стоимости на единицу веса
  - сортировка за $O(N \cdot \log(N))$
- Перебором N предметов наполняем рюкзак
- Не обеспечивает оптимального решения!

Другие жадные алгоритмы

- **Алгоритм Хаффмана**
  префиксное кодирование алфавита оптимально, с минимальной избыточностью
- **Алгоритм Крускала**
  поиск минимального остовного дерева в графе
- **Алгоритм Прима**
  поиск минимального остовного дерева в связном графе

Метод состояний

- Решение еще более узких задач
  - потоковая обработка (преобразование входного потока)
  - реактивное поведение (реакция на входы)
- Обработчик – конечный автомат
- Фиксировано число состояний

Описание конечного автомата

- Фиксируем набор состояний
- Описываем правила смены состояния в зависимости от входа
- Описываем правила записи символа по состоянию и входу

Формальное описание

Автомат без выхода:

$A = \langle X, S, s_0, F, \delta \rangle$, *где*
X — входной алфавит
S — множество состояний, $s_0 \in S$ — начальное
F — множество финальных состояний
$\delta: S \times (X \cup \{\varepsilon\}) \to S$ — функция переходов
ε — пустой вход

Формальное описание

Автомат с выходом:

классический автомат Мили (Mealy machine)

$A = \langle X, S, Y, s_0, \delta, \lambda \rangle$, *где*

X – входной алфавит

Y – выходной алфавит

S – множество состояний, $s_0 \in S$ – начальное

$\delta: S \times X \to S$ – функция переходов

$\lambda: S \times X \to Y$ – функция выходов

Автомат и Машина Тьюринга

- Управляющее устройство машины Тьюринга – конечный автомат
  - работает согласно правилам перехода
  - изменяет состояния в зависимости от текущего состояния и входа
  - записывает в ячейку ленты значение

Что есть состояние?

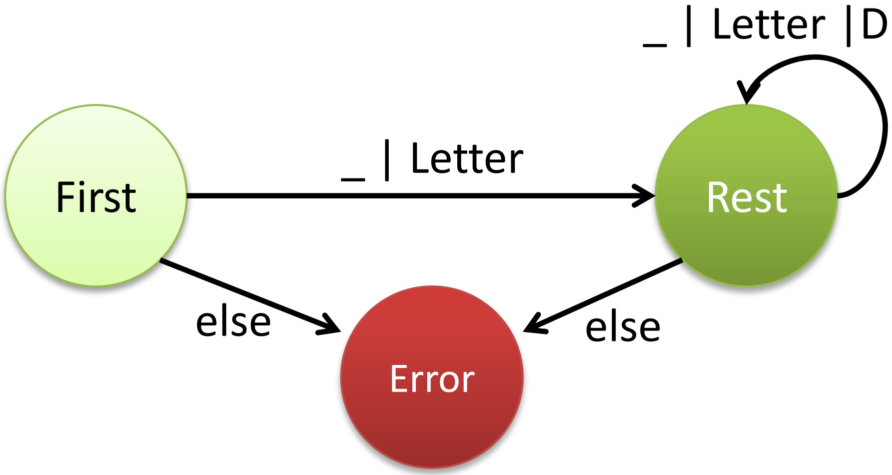
- Память о проделанной работе, закодированная в число (или в значение перечислимого типа)
- Состояния в реальном мире
  - светофор, кодовый замок

Кванторные задачи

- Тоже содержат состояния – флажки
- Состояний всего 2:
  - «нет/есть» для ∃
  - «все/не все» для ∀

Задачи проверки входа

- Проверка корректности записи
- Пример – проверка идентификатора



Проверка идентификатора

```
enum State { First, Rest, Error }
static void Main()
{
    var state = State.First;
    string str = Console.ReadLine();
    foreach (char t in str)
    {
        switch (state)
        {
            case State.First:
                state = t == '_' || char.IsLetter(t) ? State.Rest : State.Error;
                break;
            case State.Rest:
                state = t == '_' || char.IsLetterOrDigit(t) ? State.Rest : State.Error;
                break;
        }
        if (state == State.Error) break;
    }
    Console.WriteLine(state == State.Error ? "Всё плохо" : "Всё хорошо");
}
```

Альтернатива

- Регулярные выражения - специальный язык
- Предназначен для обработки строк
- Если интересно, почитайте про регулярные выражения и Regex в C#

```
static void Main()
{
    string str = Console.ReadLine();
    bool result = Regex.IsMatch(str, "^[_a-zA-Z][_a-zA-z0-9]*$");
    Console.WriteLine(result  ? "Всё хорошо":"Всё плохо");
}
```

Регулярные выражения

- Позволяют сделать то, что можно сделать конечным автоматом
- Ограничены в применении
  - нельзя проверить, что во входной строке одинаковое количество нулей и единиц
    - хотя алгоритм вы легко и быстро напишете

Вопросы?

e-mail: marchenko@it.kfu.ru