Лекция 13.
Методы разработки алгоритмов

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: `sillyhat34@gmail.com`

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.

Основные методы разработки алгоритмов

- **Метод грубой силы** (brute force, исчерпывающий поиск — полный перебор)
- **Декомпозиция** (decomposition, «разделяй и властвуй»)
- **Уменьшение размера задачи** («уменьшай и властвуй»)
- **Преобразование** («преобразуй и властвуй»)
- **Жадные алгоритмы** (greedy algorithms)
- **Динамическое программирование** (dynamic programming)
- **Поиск с возвратом** (backtracking)
- **Локальный поиск** (local search)

Метод грубой силы (brute force)

- **Метод грубой силы** (brute force) — решение «в лоб»
- Основан на прямом подходе к решению задачи
- Опирается на определения понятий, используемых в постановке задачи

- **Пример**
- Задача возведения числа *a* в неотрицательную степень *n*
- Алгоритм решения «в лоб»: по определению

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n}$$

Метод грубой силы (brute force)

- Из определения следует простейший алгоритм

```
function pow(a, n)
    pow = a
    for i = 2 to n do
        pow = pow * a
    end for
    return pow
end function
```

$T_{Pow} = O(n)$

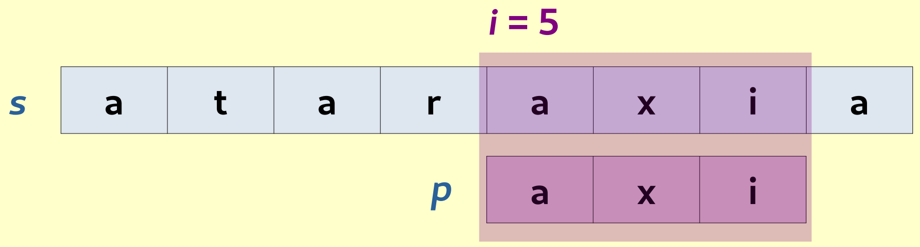
Метод грубой силы (brute force)

- **Примеры алгоритмов, основанных на методе грубой силы:**
  - Умножение матриц по определению — $O(n^2)$
  - Линейный поиск наибольшего/наименьшего элемента в списке
  - Сортировка выбором (selection sort) — $O(n^2)$
  - Пузырьковая сортировка (bubble sort) — $O(n^2)$
  - Поиск подстроки в строке методом грубой силы
  - Поиск перебором пары ближайших точек на плоскости
  - ...

Поиск подстроки в строке (string match)

- Поиск подстроки *p* в строке *s* методом грубой силы:

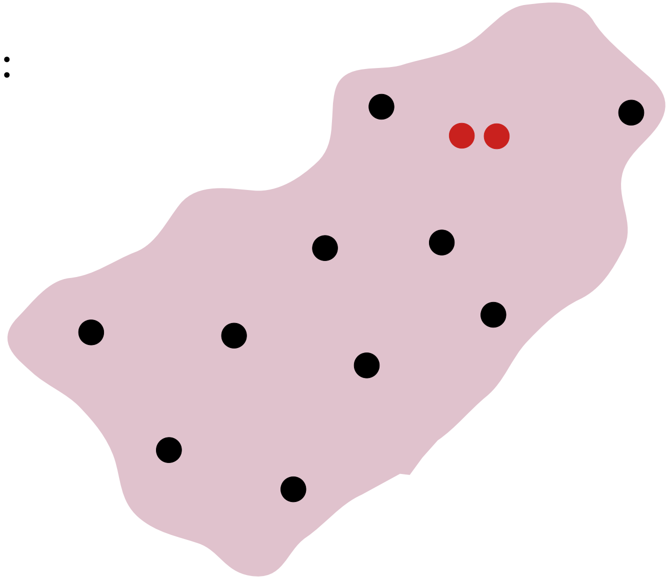
```
function strfind(s, p)
    n = strlen(s)
    m = strlen(p)
    for i = 1 to n - m do
        j = 1
        while j ≤ m and s[i + j - 1] = p[j] do
            j = j + 1
        end while
        if j > m then
            return i
        end if
    end for
end function
```



Поиск пары ближайших точек (closest pair of points problem)

- Во множестве из *n* точек необходимо найти две, расстояние между которыми минимально (точки ближе других друг к другу)
- Координаты всех точек известны: $P_i = (x_i, y_i)$
- Расстояние $d(i, j)$ между парой точек вычисляется как евклидово:

$$d(i,j)=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}$$



Поиск пары ближайших точек (closest pair of points problem)

```
function SearchClosestPoints(x[1..n], y[1..n])
    dmin = ∞
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            d = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2)
            if d < dmin then
                dmin = d
                imin = i
                jmin = j
            end if
        end for
    end for
    return imin, jmin
end function
```

Какова вычислительная сложность алгоритма?

Поиск пары ближайших точек (closest pair of points problem)

```
function SearchClosestPoints(x[1..n], y[1..n])
    dmin = ∞
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            d = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2)
            if d < dmin then
                dmin = d
                imin = i
                jmin = j
            end if
        end for
    end for
    return imin, jmin
end function
```

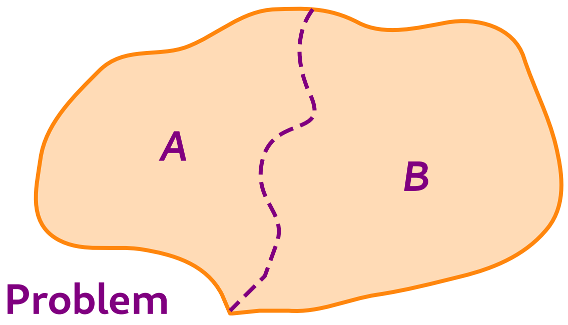
$T_{ClosestPoints} = O(n^2)$

Метод декомпозиции (decomposition)

- **Метод декомпозиции** (decomposition method, «разделяй и властвуй» — «divide and conquer»)

- **Структура алгоритмов, основанных на этом методе:**
  1. Задача разбивается на несколько меньших экземпляров этой же задачи
  2. Решаются сформированные меньшие экземпляры задачи (обычно рекурсивно)
  3. При необходимости решение исходной задачи формируется как комбинация решений меньших экземпляров задачи

Problem = SubProblemA + SubProblemB



Вычисление суммы чисел

- **Задача.** Вычислить сумму чисел $a_0, a_1, \dots, a_{n-1}$
- Алгоритм на основе метода грубой силы (по определению):

```
function sum(a[0..n - 1])
    sum = 0
    for i = 0 to n - 1 do
        sum = sum + a[i]
    end for
    return sum
end function
```

$T_{Sum} = O(n)$

Вычисление суммы чисел

- **Задача.** Вычислить сумму чисел a_0, a_1, …, a_{n-1}
- Алгоритм на основе метода декомпозиции:

$$a_0 + a_2 + \ldots + a_{n-1} = \left(a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}\right) + \left(a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1}\right)$$

4 + 5 + 1 + 9 + 14 + 11 + 7 =

= (4 + 5 + 1) + (9 + 14 + 11 + 7) =

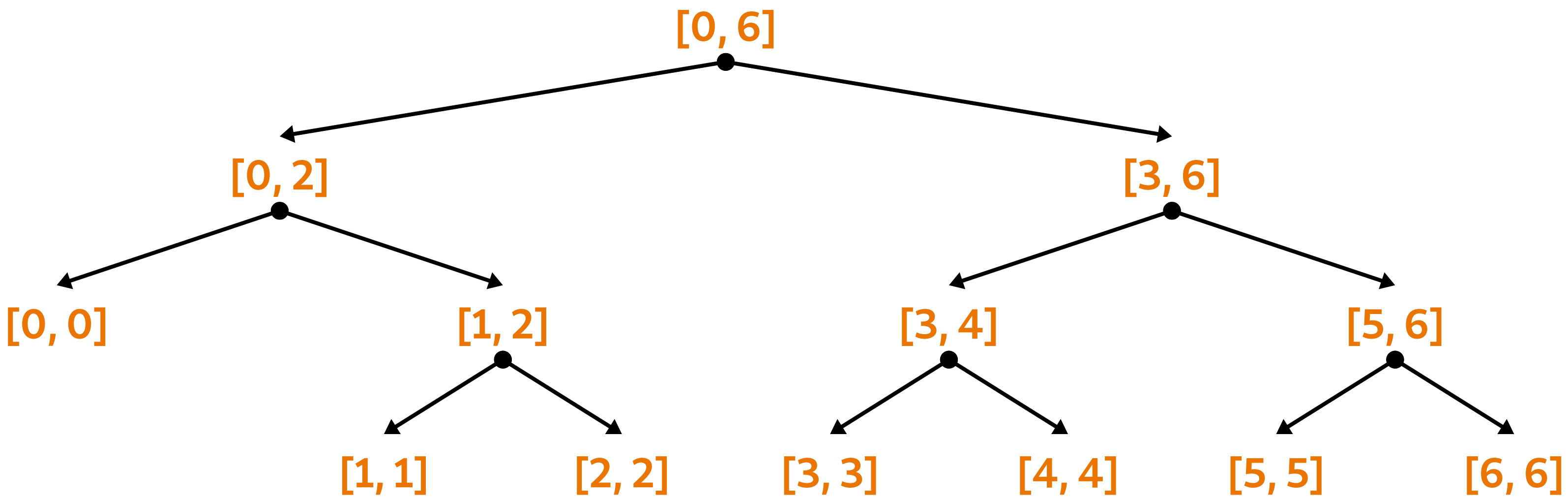
= ((4) + (5 + 1)) + ((9 + 14) + (11 + 7)) = **50**

Вычисление суммы чисел (декомпозиция)

```
int sum(int *a, int l, int r)
{
    int k;
    if (l == r)
        return a[l];
    k = (r - l + 1) / 2;
    return sum(a, l, l + k - 1) + sum(a, l + k, r);
}

int main()
{
    int a[7] = {4, 5, 1, 9, 14, 11, 7};
    int s = sum(a, 0, 6);
    return 0;
}
```

Вычисление суммы чисел (декомпозиция)

Структура рекурсивных вызовов функции sum(0, 6)



4 + 5 + 1 + 9 + 14 + 11 + 7 = (4 + 5 + 1) + (9 + 14 + 11 + 7) =

= ((4) + (5 + 1)) + ((9 + 14) + (11 + 7)) = **50**

Возведение числа *a* в степень *n*

- **Задача.** Возвести число *a* в неотрицательную степень *n*
- **Решение.** Алгоритм на основе метода декомпозиции:

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{n - \lfloor n/2 \rfloor}, & n > 1 \\ a, & n = 1 \end{cases}$$

```
int pow_decomp(int a, int n)
{
    if (n == 1)
        return a;
    int k = n / 2;
    return pow_decomp(a, k) * pow_decomp(a, n - k);
}
```

$T_{Pow} = O(n)$

Метод декомпозиции (decomposition)

- В общем случае исходная задача размера n делится на экземпляры задачи размера b, из которых a требуется решить ($b > 1$, $a \geq 0$)
- Время $T(n)$ работы алгоритма, основанного на методе декомпозиции, равно

$$T(n) = aT(n/b) + f(n), \qquad (*)$$

  где $f(n)$ — функция, учитывающая затраты времени на разделение задачи на экземпляры и комбинирование их решений
- Рекуррентное соотношение (*) — это **обобщённое рекуррентное уравнение декомпозиции** (general divide-and-conquer recurrence)

Основная теорема (master method)

Теорема. Если в обобщённом рекуррентном уравнении декомпозиции $f(n) = \Theta(n^d)$, где $d \geq 0$, то вычислительная сложность алгоритма равна:

$$T(n) = \begin{cases} \Theta(n^d), & \text{если } a < b^d, \\ \Theta(n^d \log n), & \text{если } a = b^d, \\ \Theta(n^{\log_b a}), & \text{если } a > b^d. \end{cases}$$

Анализ алгоритма суммирования *n* чисел

- ***b* = 2** — интервал делится на 2 части
- ***a* = 2** — обе части обрабатываются
- ***f*(*n*) = 1** — трудоёмкость разделения интервала на 2 подмножества и слияние результатов (операция «+») выполняется за время $O(1)$

$$T(n) = 2T(n/2) + 1$$

- Так как $f(n) = 1 = n^0$, следовательно, $\boldsymbol{d = 0}$, тогда согласно теореме сложность алгоритма суммирования *n* чисел

$$T(n) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

Метод декомпозиции (decomposition)

- **Примеры алгоритмов, основанных на методе декомпозиции**
  - Сортировка слиянием (Merge Sort)
  - Быстрая сортировка (Quick Sort)
  - Бинарный поиск (binary search)
  - Обход бинарного дерева (tree traversal)
  - Решение задачи о поиске пары ближайших точек
  - Решение задачи о поиске выпуклой оболочки
  - Умножение матриц алгоритмом Штрассена
  - ...

Сортировка слиянием (Merge Sort)

```
function MergeSort(a[0, n - 1])
    if n > 1 then
        k = n / 2
        Copy(a[0, k - 1], b[0, k - 1])
        Copy(a[k, n - 1], c[k, n - 1])
        MergeSort(b)
        MergeSort(c)
        Merge(b, c, a)
    end if
end function
```

Сортировка слиянием (Merge Sort)

```
function Merge(b[p], c[q], a[n])
    i = j = k = 0
    while i < p and j < q do
        if b[i] ≤ c[j] then
            a[k] = b[i]
            i = i + 1
        else
            a[k] = c[j]
            j = j + 1
        end if
        k = k + 1
    end while
```

```
    if i = p then
        Copy(c[j, q - 1], a[k, p + q - 1])
    else
        Copy(c[j, q - 1], a[k, p + q - 1])
    end if
end function
```

Анализ алгоритма сортировки слиянием

- ***b* = 2** — массив делится на 2 части
- ***a* = 2** — обе части обрабатываются
- ***f*(*n*) = T_{Merge}(*n*)** — количество сравнений в процессе слияния массивов
- В худшем случае $T_{Merge}(n) = n - 1$

$$T(n) = 2T(n/2) + n - 1$$

- Так как $f(n) = n - 1 = \Theta(n)$, следовательно, ***d* = 1**, тогда, согласно теореме

$$T(n) = \Theta(n^d \log n) = \Theta(n \log n)$$

Теоретический минимум вычислительной сложности алгоритмов сортировки, основанных на сравнениях, есть

$$\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$$

Динамическое программирование

- **Динамическое программирование** (dynamic programming) — метод решения задач (преимущественно оптимизационных) путём разбиения их на более простые подзадачи
- Решение задачи идёт от простых подзадач к сложным, периодически **используя ответы уже решённых подзадач** (как правило, через рекуррентные соотношения)
- Основная идея — **запоминать решения встречающихся подзадач на случай, если та же подзадача встретится вновь**
- Теория динамического программирования разработана Р. Беллманом в 1940 — 50-х годах

Последовательность Фибоначчи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$F(n) = F(n-1) + F(n-2), \qquad \text{при } n > 1$$

$$F(0) = 0, F(1) = 1$$

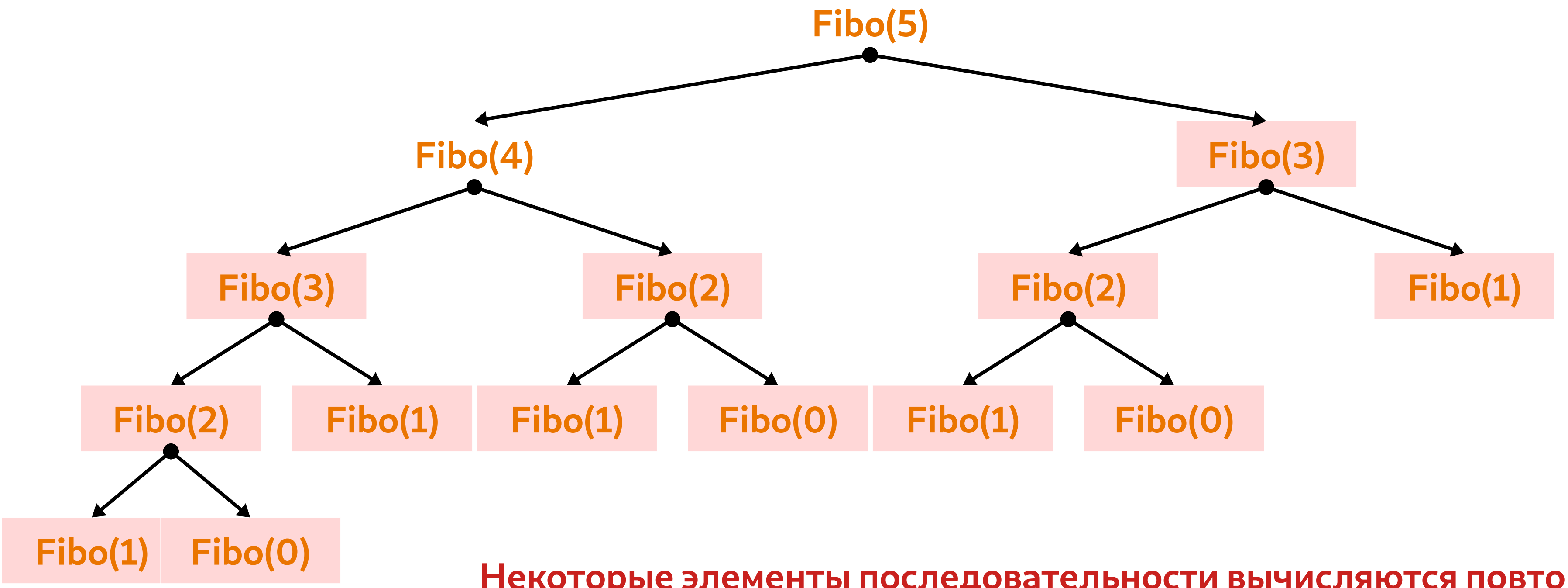
- **Задача.** Вычислить *n*-й элемент $F(n)$ последовательности Фибоначчи

Последовательность Фибоначчи

```
function Fibo(n)
    if n ≤ 1 then
        return n
    end if
    return Fibo(n - 1) + Fibo(n - 2)
end function
```

Последовательность Фибоначчи

Структура рекурсивных вызовов функции Fibo(5)



Некоторые элементы последовательности вычисляются повторно

Последовательность Фибоначчи

```
function Fibo(n)
    F[0] = 0
    F[1] = 1
    for i = 2 to n do
        F[i] = F[i - 1] + F[i - 2]
    end for
    return F[n]
end function
```

В динамическом программировании используются таблицы, в которых сохраняются решения подзадач (жертвуем памятью ради времени)

«Жадные» алгоритмы (greedy algorithms)

- **«Жадный» алгоритм** (greedy algorithm) — это алгоритм, принимающий на каждом шаге локально-оптимальное решение

- Предполагается, что конечное решение окажется оптимальным

- **Примеры «жадных» алгоритмов:**
  - Алгоритм Дейкстры
  - Алгоритм Прима
  - Алгоритм Крускала
  - Алгоритм Хаффмана (кодирование)
  - ...

Задача о размене (change-making problem)

- **Задача.** Имеется неограниченное количество монет номиналом (достоинством) $a_1 < a_2 < \ldots < a_n$
- **Требуется** выдать сумму S наименьшим числом монет

- **Пример:**
  Имеются монеты достоинством 1, 2, 5 и 10 рублей
  Выдать сумму S = 27 рублей

- **«Жадное» решение:** 2 монеты по 10 рублей, 1 по 5, 1 по 2

- На каждом шаге берётся наибольшее возможное количество монет достоинства a_i (от большего к меньшему)

Задача о размене (change-making problem)

- **Задача.** Имеется неограниченное количество монет номиналом (достоинством) $a_1 < a_2 < \ldots < a_n$
- **Требуется** выдать сумму S наименьшим числом монет

- **Пример:**

  Имеются монеты достоинством 1, 5 и 7 рублей

  Выдать сумму S = 24 рублей

- **Решение «жадным» алгоритмом:** 3 по 7, 3 по 1 = 6 монет
- **Оптимальное решение:** 2 по 7, 2 по 5 = 4 монеты

Код Хаффмана

- **Деревья Хаффмана** (Huffman tree) и **коды Хаффмана** (Huffman coding) используются для сжатия информации путём кодирования часто встречающихся символов короткими последовательностями битов

- Алгоритм предложен Д. А. Хаффманом в 1952 году (США, MIT)

Код Хаффмана

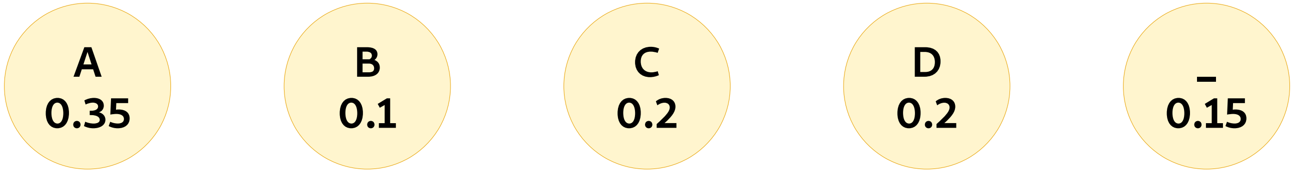
- **Задано** множество символов и известны вероятности их появления в тексте (файле)
  - $A = \{a_1, a_2, \ldots, a_n\}$ — множество символов (алфавит)
  - $P = \{p_1, p_2, \ldots, p_n\}$ — вероятности появления символов
- **Требуется** каждому символу сопоставить код — последовательность битов (codeword)
  - $C(A, P) = \{c_1, c_2, \ldots, c_n\}$

Пример:

Символ	A	B	C	D	_
Код (биты)	11	100	00	01	101

Код Хаффмана

- **Шаг 1.** Создаётся *n* одноузловых деревьев
- В каждом узле записан символ алфавита и вероятность его появления в тексте

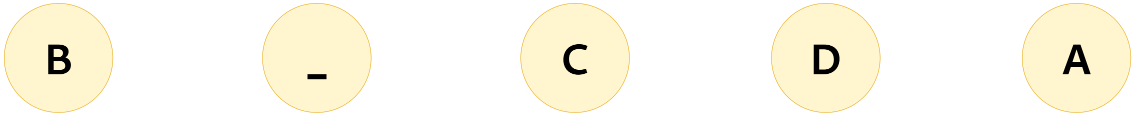


Код Хаффмана

- **Шаг 2.** Находим два дерева с наименьшими вероятностями и делаем их левым и правым поддеревьями нового дерева — создаём родительский узел
- В созданном узле записываем сумму вероятностей поддеревьев
- Повторяем шаг 2, пока не получим одно дерево

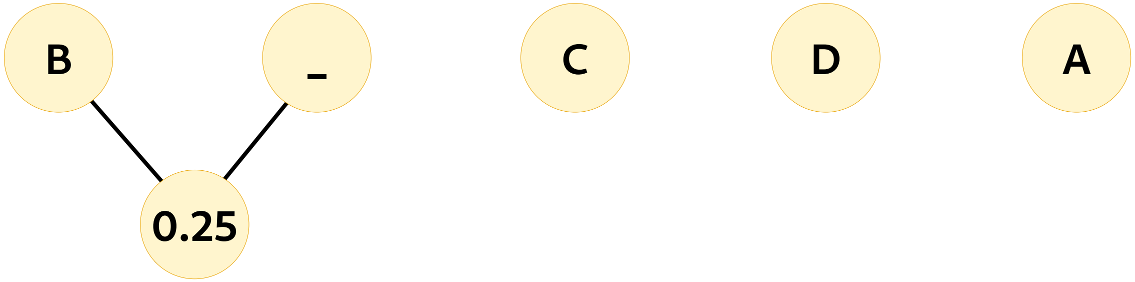
На каждом шаге осуществляется «жадный» выбор — выбираем два узла с наименьшими вероятностями

Код Хаффмана



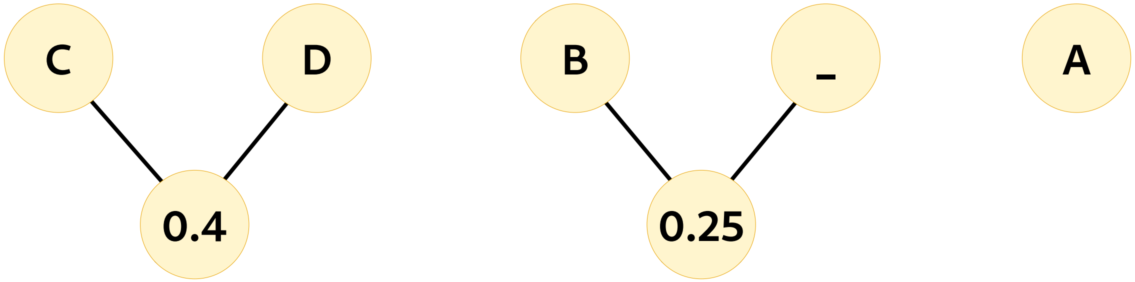
Символ	B	_	C	D	A
Вероятность	0.1	0.15	0.2	0.2	0.35

Код Хаффмана



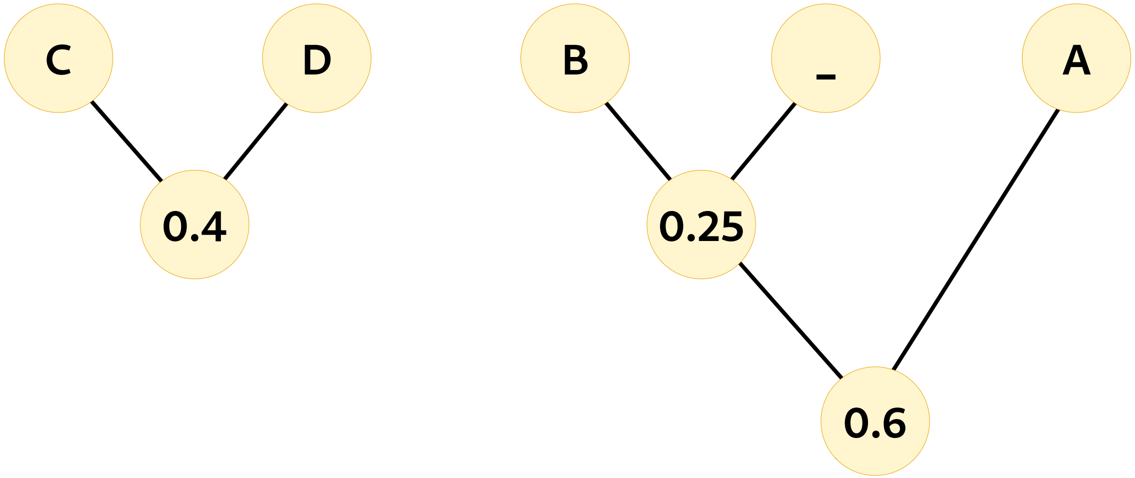
Символ	B	_	C	D	A
Вероятность	0.25		0.2	0.2	0.35

Код Хаффмана



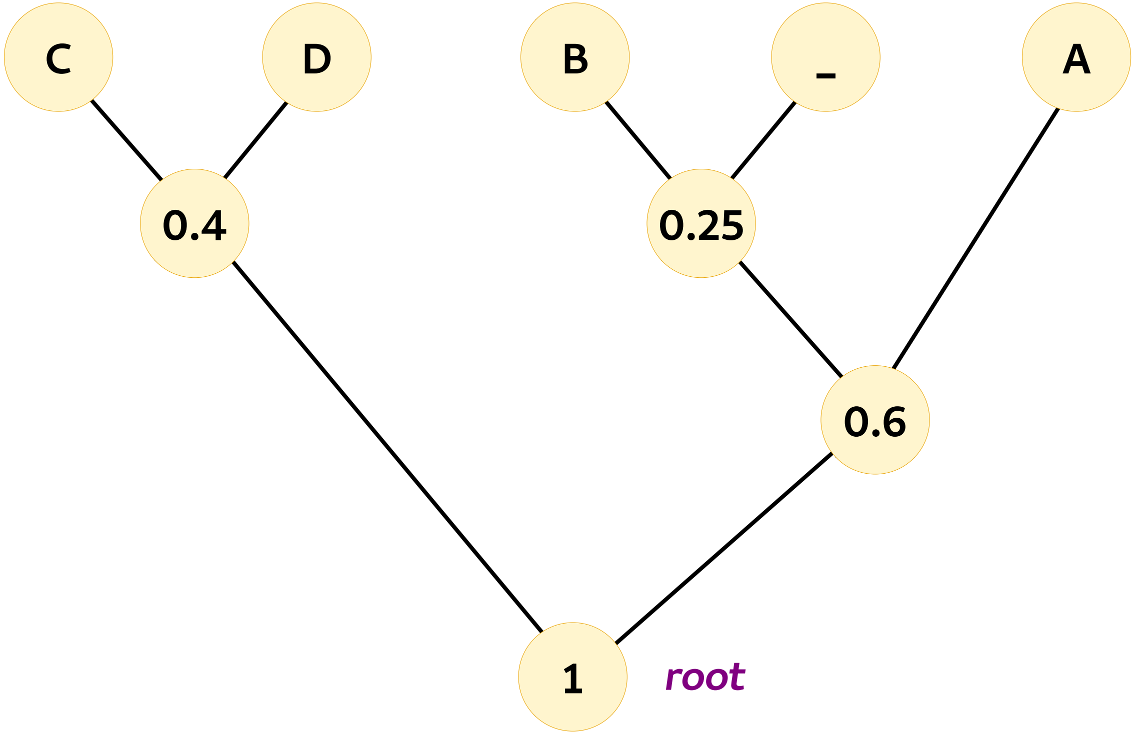
Символ	C	D	B	_	A
Вероятность	0.4		0.25		0.35

Код Хаффмана

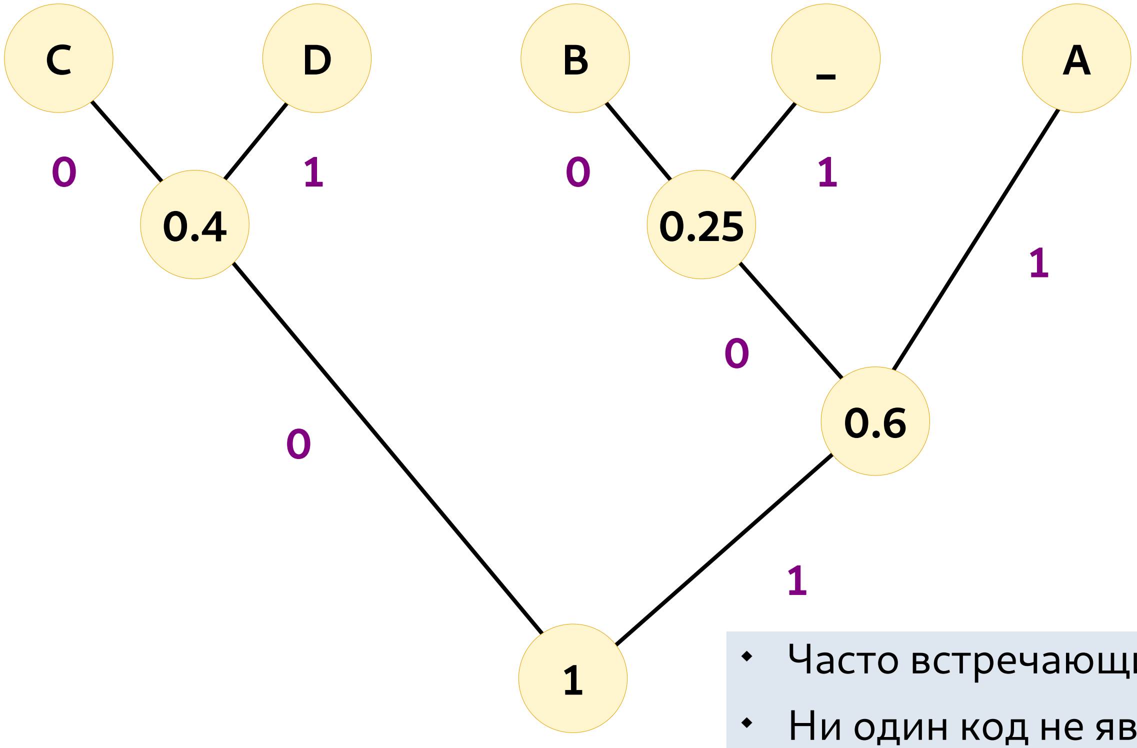


Символ	C	D	B	_	A
Вероятность	0.4		0.6		

Код Хаффмана



Код Хаффмана



- Левое ребро — 1
- Правое ребро — 0

Символ	Код
A	11
B	100
C	00
D	01
_	101

- Часто встречающиеся символы получили короткие коды
- Ни один код не является префиксом другого

Поиск с возвратом (backtracking)

- **Поиск с возвратом** (backtracking) — это метод решения задач, в которых необходим полный перебор всех возможных вариантов в некотором множестве *M*
- «Построить все возможные варианты…», «Сколько существует способов…», «Есть ли способ…»
- Термин *backtracking* введён в 1950 г. Д. Г. Лемером (D. H. Lehmer)
- **Примеры задач:**
  - Задача коммивояжёра
  - Подбор пароля
  - Задача о восьми ферзях
  - Задача об упаковке рюкзака
  - Раскраска графа
  - …

Поиск выхода из лабиринта

```
6 9
1 1
1 1 1 1 1 1 1 1 1
1 0 0 0 0 0 1 1 1
1 0 0 1 1 0 0 0 1
1 1 0 0 0 0 1 0 1
1 0 0 1 0 1 1 0 1
1 1 1 1 0 1 1 1 1
```

6 строк, 9 столбцов

Старт: строка 1, столбец 1

- 1 — стена
- 0 — проход
- Разрешено ходить влево, вправо, вверх и вниз

Поиск выхода из лабиринта

```
int main(int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, “Usage: maze <maze-file>\n”);
        exit(1);
    }

    load_maze(argv[1]);
    print_maze();
    backtrack(startrow, startcol);

    printf(“Exit not found\n”);

    return 0;
}
```

Поиск выхода из лабиринта

```
void backtrack(int row, int col)
{
    int drow[4] = {-1, 0, 1, 0}, dcol[4] = {0, 1, 0, -1};
    int nextrow, nextcol, i;
    maze[row][col] = 2;                // Встали на новую позицию
    if (row == 0 || col == 0 || row == (nrows - 1) || col == (ncols - 1)) {
        print_maze();                  // Нашли выход
        exit(0);
    }
    for (i = 0; i < 4; i++) {
        nextrow = row + drow[i];
        nextcol = col + dcol[i];
        if (maze[nextrow][nextcol] == 0)    // Есть проход?
            backtrack(nextrow, nextcol);
    }
    maze[row * ncols + col] = 0;
}
```

Поиск выхода из лабиринта

```
1 1 1 1 1 1 1 1 1
1 2 2 2 2 2 1 1 1
1 0 0 1 1 2 0 0 1
1 1 0 0 2 2 1 0 1
1 0 0 1 2 1 1 0 1
1 1 1 1 2 1 1 1 1
```

Общая структура метода поиска с возвратом

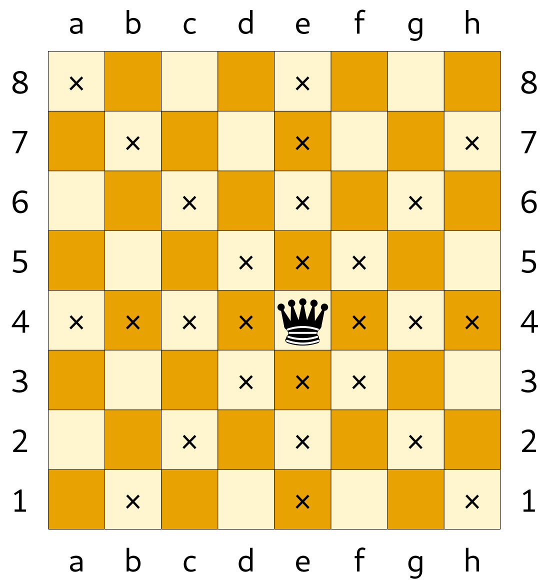
```
int main()
{
    backtracking(1);
}
```

```
void backtracking(int step)
{
    save_variant();
    if (is_solution_found()) {
        print_solution();
        exit(0);
    }
    for (i = 0; i < nvariants; i++) {
        if (is_correct_variant())
            backtracking(step + 1);
    }
    delete_variant();
}
```

Задача о восьми ферзях (eight queens puzzle)

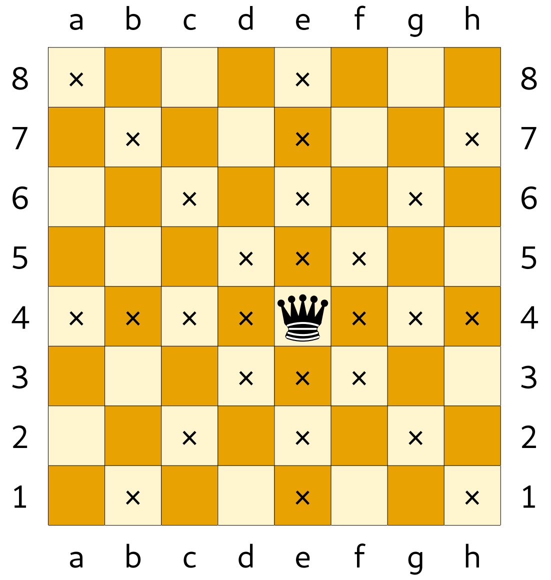
- **Классическая формулировка**
  Расставить на стандартной 64-клеточной шахматной доске 8 ферзей (королев) так, чтобы ни один из них не находился под боем другого

- **Альтернативная формулировка**
  Заполнить матрицу размером 8×8 нулями и единицами таким образом, чтобы сумма всех элементов матрицы была равна 8, при этом сумма элементов ни в одном столбце, строке или диагональном ряду матрицы не превышала единицы



Задача о восьми ферзях (eight queens puzzle)

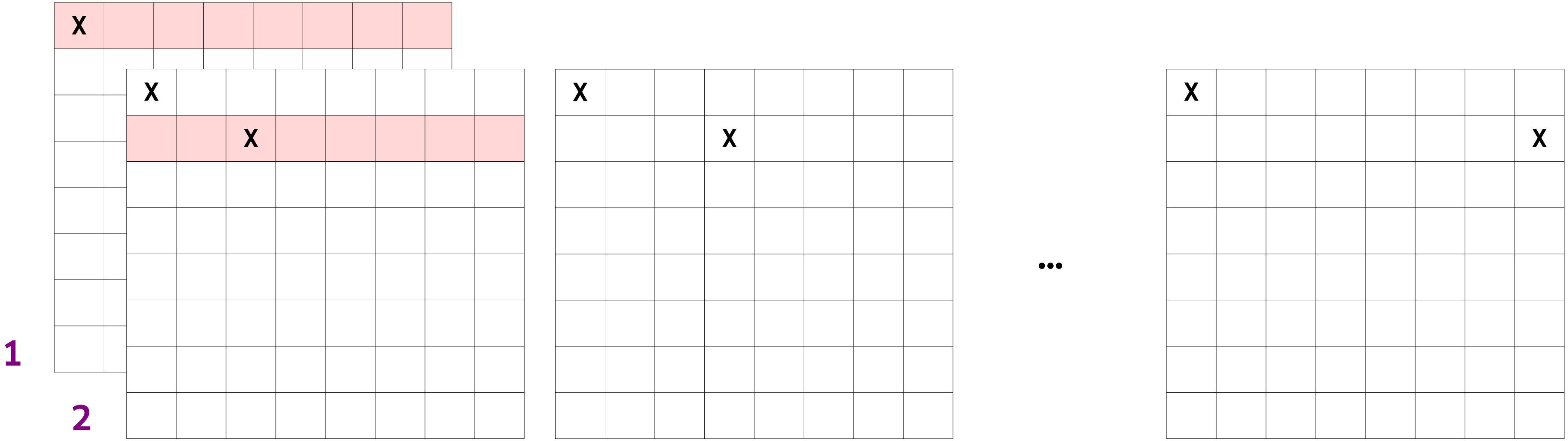
- Задача впервые была решена в 1850 г. Карлом Фридрихом Гаубом (Carl Friedrich Gaub)
- Число возможных решений на 64-клеточной доске: **92**



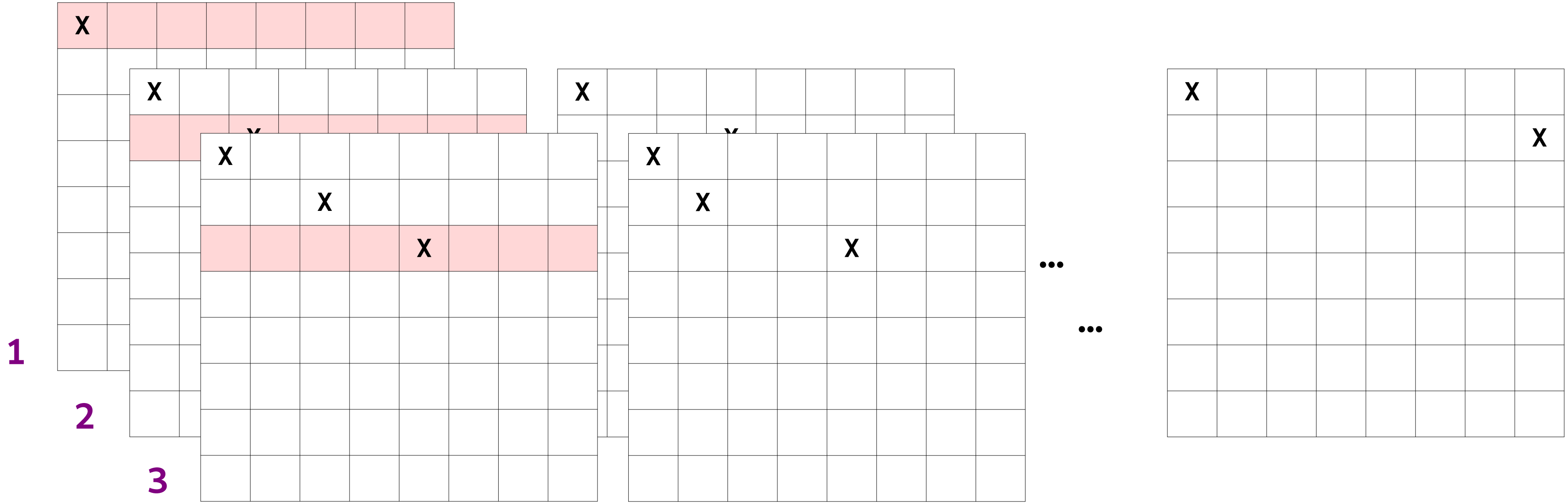
Задача о восьми ферзях (eight queens puzzle)

	X							
1								

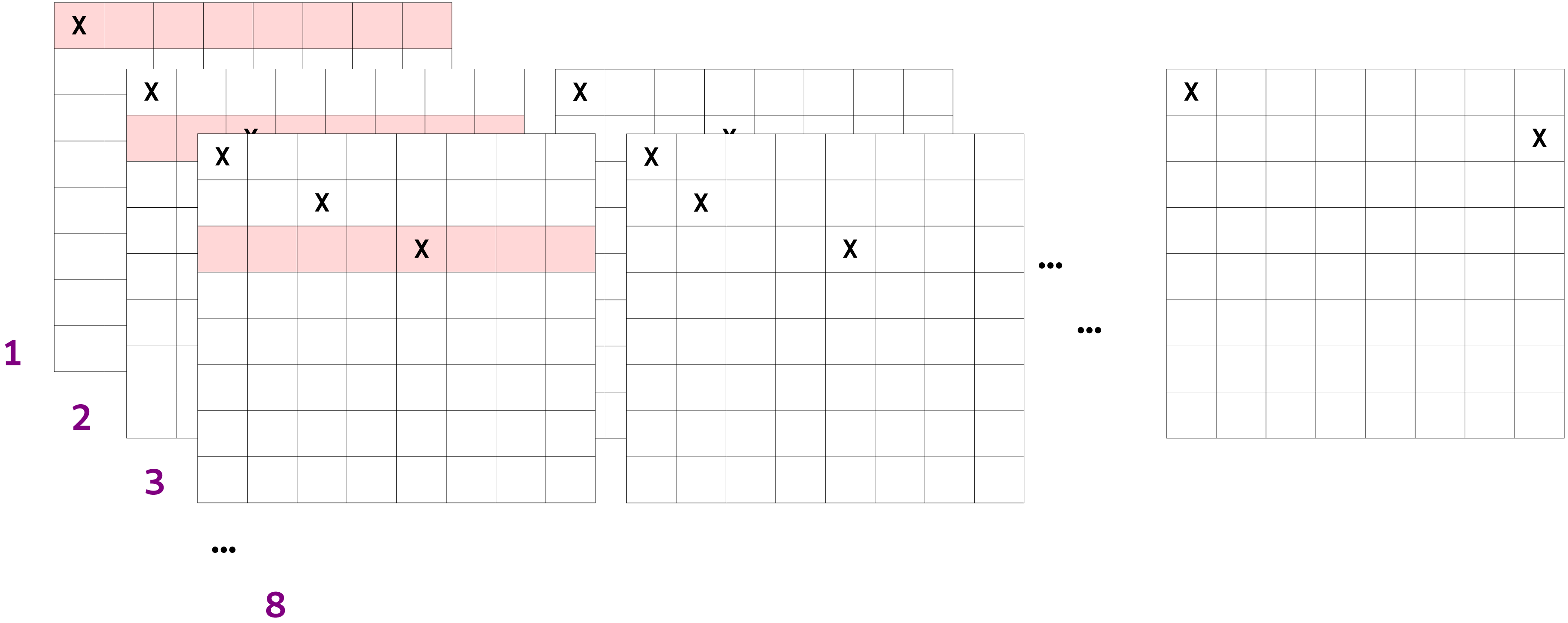
Задача о восьми ферзях (eight queens puzzle)



Задача о восьми ферзях (eight queens puzzle)



Задача о восьми ферзях (eight queens puzzle)



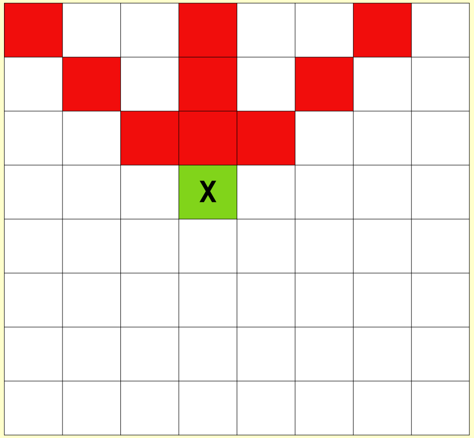
Задача о восьми ферзях (eight queens puzzle)

```c
enum { N = 8 };

int board[N][N];

int main()
{
    int i, j;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            board[i][j] = 0;
    backtrack(0);
    return 0;
}
```

```c
void backtrack(int row)
{
    int col;
    if (row >= N)
        print_board();
    for (col = 0; col < N; col++) {
        if (is_correct_order(row, col)) {
            board[row][col] = 1;  // Ставим ферзя
            backtracking(row + 1);
            board[row][col] = 0;  // Откат
        }
    }
}
```

Задача о восьми ферзях (eight queens puzzle)

```c
int is_correct_order(int row, int col)
{
    if (row == 0)
        return 1;
    int i, j;
    /* Проверка позиций сверху */
    for (i = row; i >= 0; i--) {
        if (board[i][col] != 0)
            return 0;
    }
    /* Проверить левую и правую диагонали... */
    return 1;
}
```



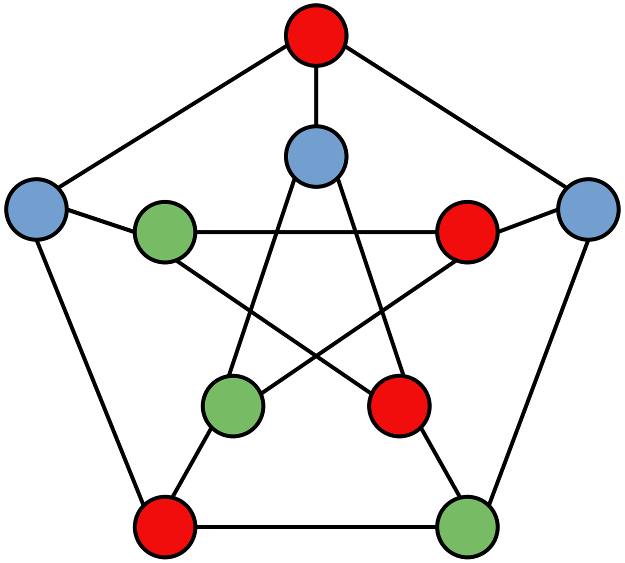
Задача о восьми ферзях (eight queens puzzle)

```
1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

Задача о раскраске графа в *k* цветов

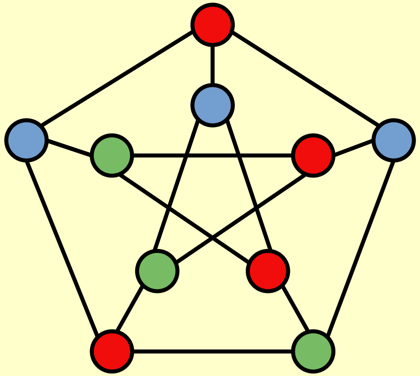
- Имеется граф $G = (V, E)$, состоящий из n вершин
- Каждую вершину нужно раскрасить в один из k цветов так, чтобы смежные вершины были раскрашены в разные цвета

Пример раскраски 10 вершин графа в 3 цвета



Задача о раскраске графа в *k* цветов

```
int main()
{
    backtracking(1);
}
```



```
void backtracking(int v)
{
    if (v > nvertices) {
        print_colors();
        exit(0);
    }
    for (i = 0; i < ncolors; i++) {
        color[v - 1] = i;
        if (is_correct_color(v))
            backtracking(v + 1);
        color[v - 1] = -1;
    }
}
```

Задача о раскраске графа в *k* цветов

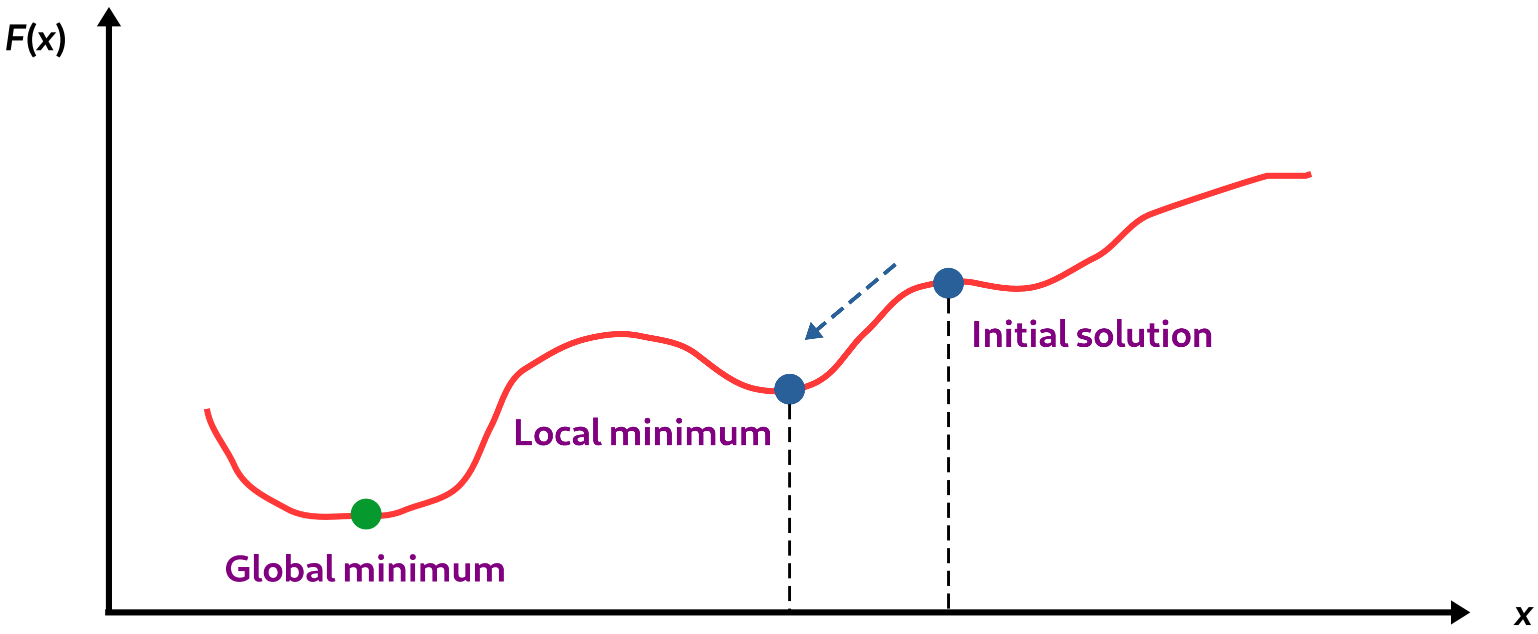
```c
/*
 * Граф представлен матрицей смежности (см. graph.h, graph.c)
 */

int is_correct_color(int v)
{
    int i;
    for (i = 0; i < nvertices; i++) {
        if (graph_get_edge(g, v, i + 1) > 0)
            if (colors[v - 1] == colors[i])
                return 0;
    }
    return 1;
}
```

Локальный поиск (local search)

- **Локальный поиск** (local search) — это метод приближённого решения оптимизационных задач
- Жертвуется точность решения для сокращения времени работы алгоритма

- **Примеры методов локального поиска:**
  - Имитация отжига (simulated annealing)
  - Генетические алгоритмы (genetic algorithms)
  - Поиск с запретами (tabu search)
  - …

Локальный поиск (local search)



Локальный поиск (local search)

```
function LocalSearch()
    x = InitialSolution()

    repeat
        x’ = GenerateNewSolution(x)
        if F(x’) < F(x) then
            x = x’
        end if
    while not ExitCondition(x, x’)

    return x
end function
```

Домашнее чтение

- Прочитать в [Levitin] раздел о методах разработки алгоритмов
- Оценить трудоёмкость алгоритма быстрой сортировки с использованием обобщённого рекуррентного уравнения декомпозиции [Levitin, С. 174]

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: `sillyhat34@gmail.com`

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.