Методы разработки алгоритмов. Динамическое программирование. «Жадные» алгоритмы (greedy algorithms). Код Хаффмана. Поиск с возвратом (backtracking). Задача о восьми ферзях. Задача о раскраске графа в k цветов. Локальный поиск.

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)*

Динамическое программирование

- **Динамическое программирование** *(dynamic programming)* – метод решения задач *(преимущественно оптимизационных)* путем разбиения их на более простые подзадачи
- Решение задачи идет от простых подзадач к сложным, периодически используя ответы **уже решенных 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

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

Пример задачи для решения динамическим программированием: **Последовательность Фибоначчи**

```
// Функция вычисления n-го числа Фибоначчи
int fibonacci(int n) {
    // Обработка особых случаев
    if (n == 0) return 0;
    if (n == 1) return 1;

    // Выделение памяти для массива
    int* F = (int*)malloc((n + 1) * sizeof(int));
    if (F == NULL) {
        printf("Ошибка выделения памяти\n");
        return -1; // Возвращаем ошибку
    }

    // Базовые случаи
    F[0] = 0;
    F[1] = 1;

    // Вычисление чисел Фибоначчи от 2 до n
    for (int i = 2; i <= n; i++) {
        F[i] = F[i - 1] + F[i - 2];
    }

    int result = F[n]; // Сохраняем результат
    free(F); // Освобождаем память
    return result;
}
```

"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. Имеется неограниченное количество монет номиналом (достоинством) a1 < a2 < … < an
- 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 берется наибольшее возможное количество монет достоинства ai *(от большего к меньшему)*

Код Хаффмана

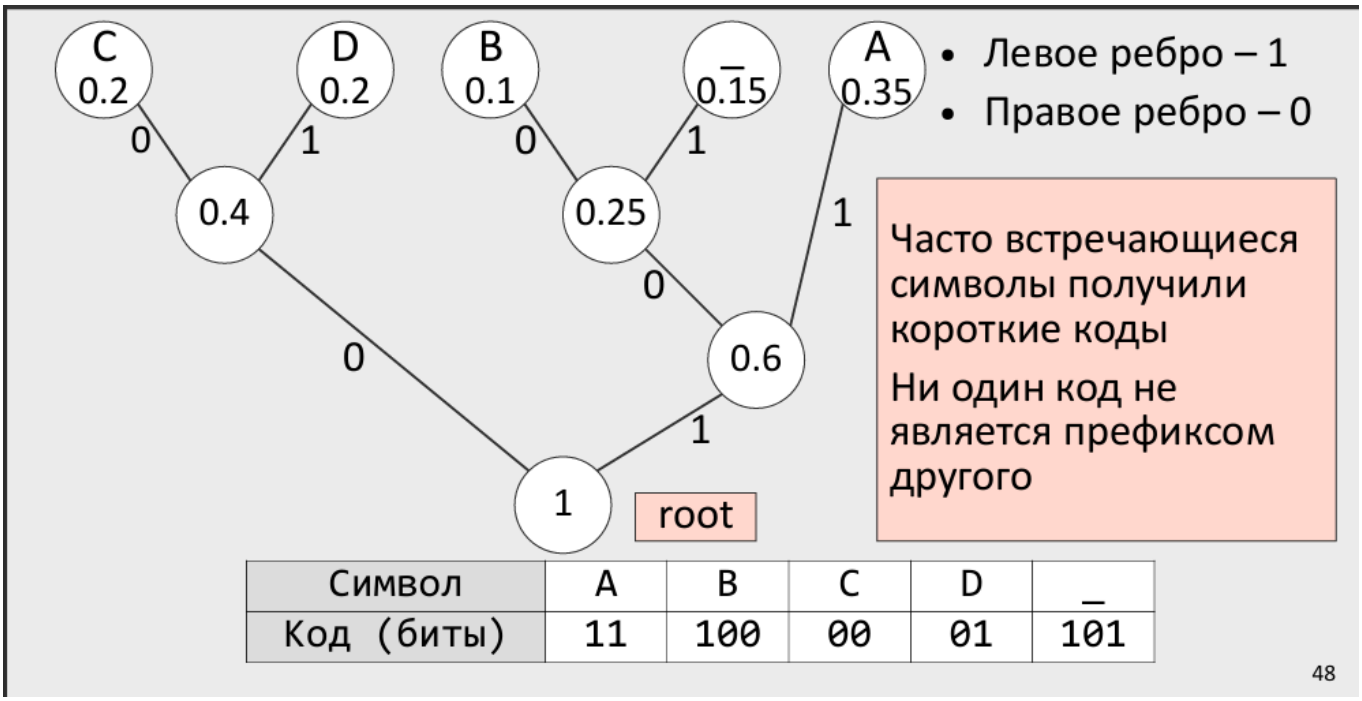
- **Деревья Хаффмана** *(Huffman tree)* и **коды Хаффмана** *(Huffman coding)* используются для сжатия информации путем кодирования часто встречающихся символов короткими последовательностями битов

Шаг 1.

- Создается n одноузловых деревьев
- В каждом узле записан символ алфавита и вероятность его появления в тексте

Шаг 2.

- Находим два дерева с наименьшими вероятностями и делаем их левым и правым поддеревьями нового дерева – создаем 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



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

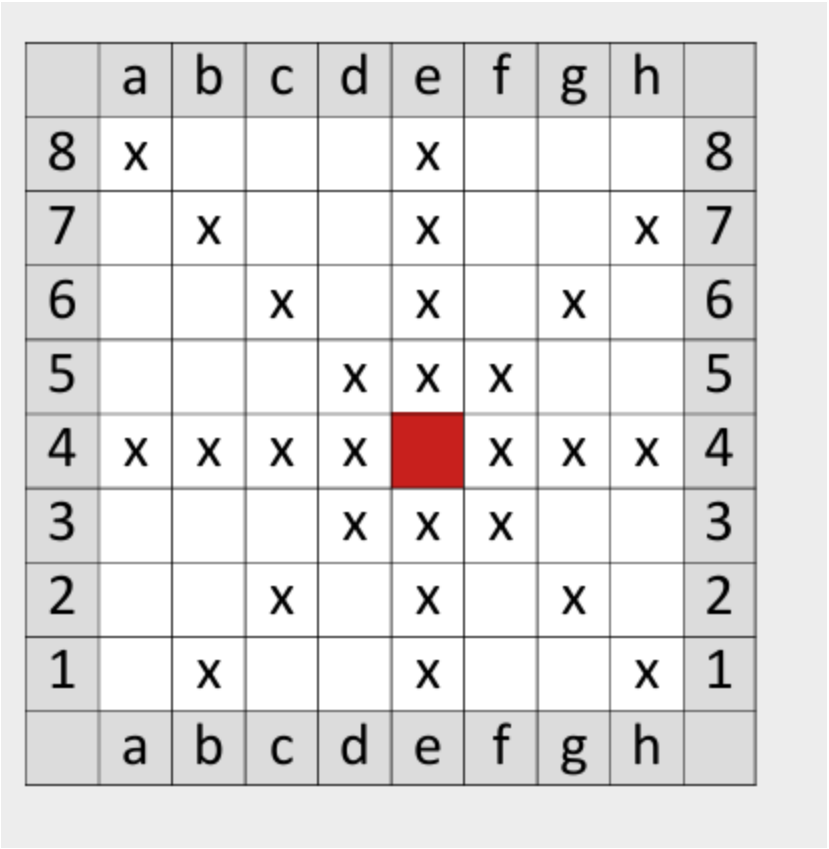
Поиск с возвратом *(backtracking).*

- **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

Задача о восьми ферзях

- **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

- Число возможных решений на 64 клеточной доске: 92



	a	b	c	d	e	f	g	h	
8	x				x				8
7		x			x			x	7
6			x		x		x		6
5				x	x	x			5
4	x	x	x	x		x	x	x	4
3				x	x	x			3
2			x		x		x		2
1		x			x			x	1
	a	b	c	d	e	f	g	h	

```c
// Размер шахматной доски (8x8)
enum { N = 8 };

// Шахматная доска (0 - пустая клетка, 1 - стоит ферзь)
int board[N][N];

int main()
{
    int i, j;
    // Инициализация доски - заполнение нулями
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            board[i][j] = 0;

    // Запуск алгоритма backtracking с первой строки (row = 0)
    backtrack(0);

    return 0;
}

// Рекурсивная функция для поиска всех возможных расстановок ферзей
void backtrack(int row)
```

```
{
    int col;

    // Базовый случай: если дошли до последней строки
    if (row >= N) {
        // Найдено решение - печатаем доску
        print_board();
        return;
    }

    // Перебираем все колонки в текущей строке
    for (col = 0; col < N; col++) {
        // Проверяем, можно ли поставить ферзя в (row, col)
        if (is_correct_order(row, col))
        {
            // Ставим ферзя
            board[row][col] = 1;

            // Рекурсивно вызываем для следующей строки
            backtrack(row + 1);

            // Откатываем изменения (backtracking)
            board[row][col] = 0;
        }
    }
}

// Функция проверки корректности позиции для нового ферзя
int is_correct_order(int row, int col)
{
    // Для первой строки всегда можно поставить ферзя
    if (row == 0)
        return 1;

    int i, j;

    /* Проверка вертикали сверху */
    for (i = row; i >= 0; i--) {
        if (board[i][col] != 0)
            return 0;
    }

    /* Проверить левую и правую диагонали... */
    // (эта часть кода не реализована полностью в примере)
```

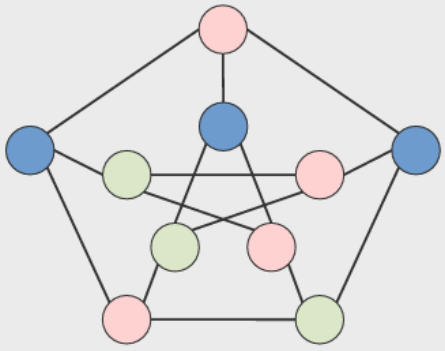
```
    return 1;
}
```

Результат :

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



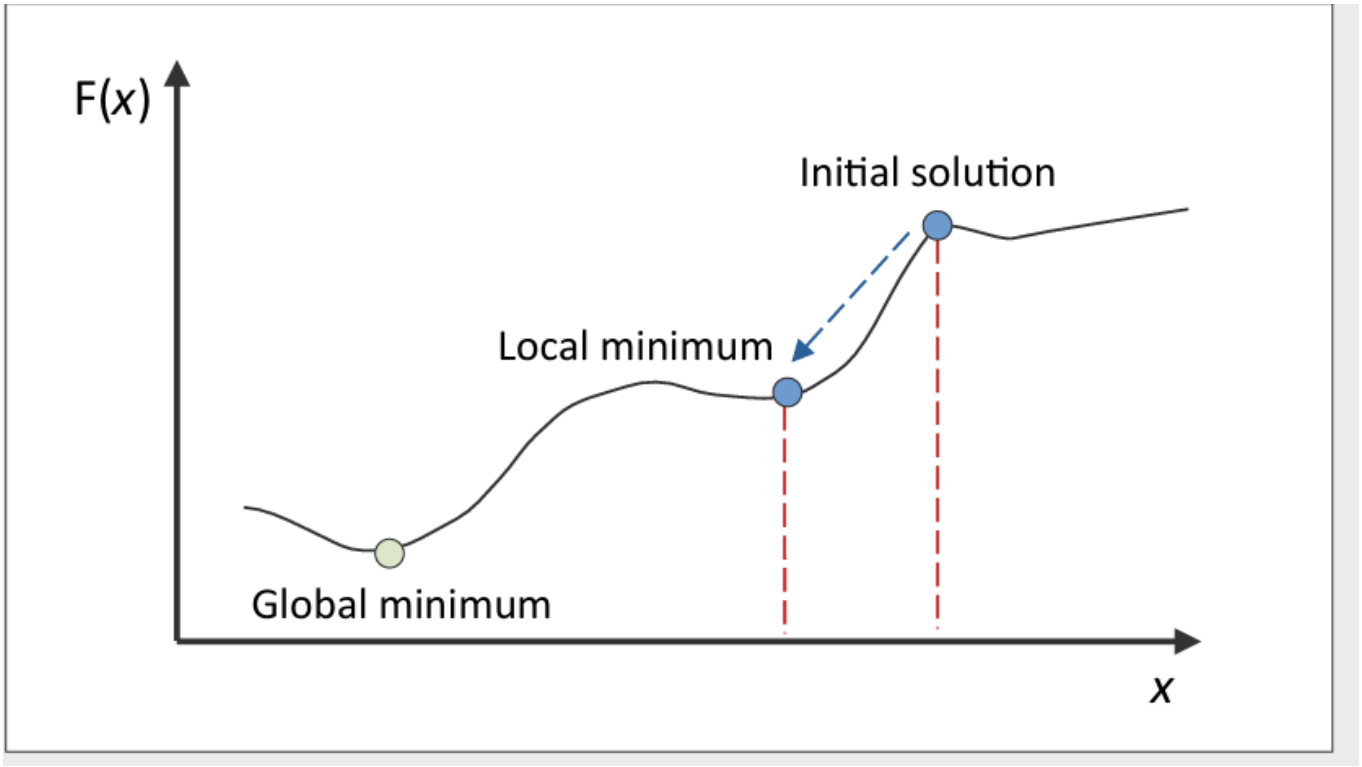
Пример раскраски 10 вершин графа в 3 цвета

```c
// Функция backtracking (поиск с возвратом) для раскраски графа
// v - текущая вершина, которую нужно раскрасить (нумерация с 1)
void backtracking(int v)
{
    // Базовый случай: если раскрасили все вершины
    if (v > nvertices) {
        // Найдено решение - печатаем раскраску
        print_colors();
        // Выходим из программы после первого найденного решения
        exit(0);
    }

    // Перебираем все возможные цвета для текущей вершины
    for (int i = 0; i < ncolors; i++) {
        // Пробуем раскрасить вершину v в цвет i
        color[v - 1] = i;  // v-1 потому что массив индексируется с 0

        // Проверяем, допустима ли такая раскраска
        if (is_correct_color(v))
            // Если да, рекурсивно переходим к следующей вершине
            backtracking(v + 1);

        // Откатываем изменения (backtracking)
        color[v - 1] = -1;  // -1 может означать "не раскрашено"
    }
}

int main()
{
    // Запускаем алгоритм с первой вершины (нумерация с 1)
    backtracking(1);
}

/*
 * Функция проверки корректности раскраски для вершины v
 * Проверяет, что цвет вершины v не совпадает с цветами смежных вершин
 */
int is_correct_color(int v)
{
    int i;
    // Проверяем все вершины графа
    for (i = 0; i < nvertices; i++) {
        // Если есть ребро между v и i+1 (i+1 потому что v нумеруется с 1)
        if (graph_get_edge(g, v, i + 1) > 0)
            // И если цвета совпадают
            if (colors[v - 1] == colors[i])
```

```
                // Раскраска некорректна
                return 0;
    }
    // Если не нашли конфликтов - раскраска корректна
    return 1;
}
```

Локальный поиск

- **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)*




```
// Целевая функция (пример: минимизация суммы квадратов)
double F(double x) {
    return x * x;  // Простая параболическая функция для демонстрации
}
```

```
// Генерация начального решения
double InitialSolution() {
    srand(time(NULL));  // Инициализация генератора случайных чисел
    return (double)(rand() % 200 - 100) / 10.0;  // Возвращаем случайное число
от -10.0 до 10.0
}

// Генерация нового решения на основе текущего
double GenerateNewSolution(double current) {
    // Добавляем небольшое случайное изменение к текущему решению
    return current + (double)(rand() % 21 - 10) / 10.0;  // Изменение от -1.0
до +1.0
}

// Условие выхода из цикла
int ExitCondition(double x, double x_prime, int iterations) {
    // Можно использовать различные условия:
    // 1. Максимальное число итераций
    // 2. Незначительное улучшение решения
    // 3. Достижение целевого значения
    return (iterations >= 100) || (fabs(F(x_prime) - F(x)) < 0.0001;
}

// Основная функция локального поиска
double LocalSearch() {
    double x = InitialSolution();      // Начальное решение
    double x_prime;                   // Новое решение-кандидат
    int iteration = 0;                // Счетчик итераций

    printf("Начальное решение: x = %.2f, F(x) = %.2f\n", x, F(x));

    do {
        x_prime = GenerateNewSolution(x);  // Генерируем новое решение

        // Если новое решение лучше текущего
        if (F(x_prime) < F(x)) {
            x = x_prime;  // Принимаем новое решение
            printf("Улучшение: x = %.2f, F(x) = %.2f (Итерация %d)\n",
                   x, F(x), iteration);
        }

        iteration++;
    } while (!ExitCondition(x, x_prime, iteration));

    printf("Финальное решение: x = %.2f, F(x) = %.2f\n", x, F(x));
```

```
    return x;
}
```
