Алгоритмы и анализ сложности

1. Сортировка данных вставками. Пример.

Сортировка вставками (Insertion Sort) — алгоритм, при котором элементы входного массива просматриваются один за другим, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

Алгоритм:

1. Начиная со второго элемента (индекс 1), рассматриваем его как "текущий"
2. Сравниваем текущий элемент с предыдущими элементами
3. Если предыдущий элемент больше текущего, перемещаем его вправо
4. Продолжаем, пока не найдем правильную позицию для текущего элемента
5. Вставляем текущий элемент в найденную позицию
6. Повторяем для всех элементов массива

Пример реализации:

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Перемещаем элементы arr[0..i-1], которые больше key,
        // на одну позицию вправо
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Пример сортировки:

Массив: [5, 2, 4, 6, 1, 3]

- Проход 1 (i=1): [5, 2, 4, 6, 1, 3] → [2, 5, 4, 6, 1, 3]
- Проход 2 (i=2): [2, 5, 4, 6, 1, 3] → [2, 4, 5, 6, 1, 3]
- Проход 3 (i=3): [2, 4, 5, 6, 1, 3] → [2, 4, 5, 6, 1, 3] (не меняется)
- Проход 4 (i=4): [2, 4, 5, 6, 1, 3] → [1, 2, 4, 5, 6, 3]
- Проход 5 (i=5): [1, 2, 4, 5, 6, 3] → [1, 2, 3, 4, 5, 6]

Сложность:

- Временная сложность:
    - В лучшем случае: O(n) (когда массив уже отсортирован)
    - В среднем и худшем случае: $O(n^2)$
- Пространственная сложность: O(1) (сортировка выполняется "на месте")

2. Структуры данных: описание, обращение к элементам структуры.

Структура — это пользовательский тип данных, который группирует переменные разных типов под одним именем.

Определение структуры в C/C++:

```
struct Person {
    std::string name;
    int age;
    double height;
};
```

Создание переменных типа структуры:

```
// Объявление и инициализация
Person person1 = {"John", 25, 1.75};

// Отдельное объявление и присваивание
Person person2;
person2.name = "Jane";
person2.age = 30;
person2.height = 1.65;

// Через конструктор (C++)
Person person3{"Alice", 22, 1.70};
```

Обращение к элементам структуры:

1. **Через точку** (для переменных-структур):

```
std::cout << person1.name << " is " << person1.age << " years old."
<< std::endl;
person1.age = 26;  // изменение поля
```

2. **Через стрелку** (для указателей на структуры):

```
Person* personPtr = &person1;
std::cout << personPtr->name << std::endl;
personPtr->age = 27;  // изменение поля через указатель
```

3. **С использованием разыменования указателя**:

```
(*personPtr).name = "John Smith";  // эквивалентно personPtr->name
```

Массивы структур:

```
// Массив структур
Person people[3] = {
    {"John", 25, 1.75},
    {"Jane", 30, 1.65},
    {"Alice", 22, 1.70}
};

// Обращение к элементам массива структур
std::cout << people[1].name << std::endl;  // Jane
```

Структуры как параметры функций:

```
// Передача по значению
void displayPerson(Person p) {
    std::cout << p.name << ", " << p.age << " years" << std::endl;
}

// Передача по ссылке
void incrementAge(Person& p) {
    p.age++;
}

// Передача по указателю
void setName(Person* p, const std::string& newName) {
    p->name = newName;
}
```

Вложенные структуры:

```
struct Address {
    std::string street;
    std::string city;
    std::string country;
};

struct Employee {
    std::string name;
    int id;
    Address address;  // вложенная структура
};

Employee emp = {"John", 12345, {"Main St", "New York", "USA"}};
std::cout << emp.address.city << std::endl;  // New York
```

3. Сортировка методом «пузырька», разделением.

Сортировка пузырьком (Bubble Sort)

Сортировка пузырьком — простой алгоритм сортировки, который многократно проходит по списку, сравнивает соседние элементы и меняет их местами, если они расположены в неправильном порядке.

Алгоритм:

1. Сравниваем соседние элементы
2. Если они находятся в неправильном порядке (левый больше правого), меняем их местами
3. Повторяем для всех пар соседних элементов
4. После первого прохода самый большой элемент окажется в конце
5. Повторяем для оставшихся n-1, n-2, ... элементов

Пример реализации:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Меняем элементы местами
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Оптимизированная версия:

```
void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = false;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                swapped = true;
            }
        }
        // Если за проход не было обменов, массив отсортирован
        if (!swapped)
            break;
    }
}
```

Сложность:

- Временная сложность: $O(n^2)$ в худшем и среднем случае, $O(n)$ в лучшем случае (оптимизированная версия)
- Пространственная сложность: $O(1)$

Сортировка разделением (Quicksort)

Быстрая сортировка (Quicksort) — эффективный алгоритм сортировки, использующий стратегию "разделяй и властвуй".

Алгоритм:

1. Выбираем опорный элемент из массива (обычно последний)
2. Перераспределяем элементы так, чтобы:
    - Элементы меньше опорного перемещаются влево от него
    - Элементы больше опорного перемещаются вправо от него
3. Рекурсивно применяем шаги 1-2 к подмассивам слева и справа от опорного элемента

Пример реализации:

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // выбираем последний элемент как опорный
    int i = low - 1;        // индекс меньшего элемента

    for (int j = low; j < high; j++) {
        // Если текущий элемент меньше или равен опорному
        if (arr[j] <= pivot) {
            i++;
            // Меняем arr[i] и arr[j] местами
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Меняем arr[i+1] и arr[high] (опорный элемент) местами
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;  // возвращаем позицию опорного элемента
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi - индекс опорного элемента
        int pi = partition(arr, low, high);

        // Рекурсивная сортировка элементов до и после опорного
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Пример сортировки:

Массив: [10, 7, 8, 9, 1, 5]

1. Опорный элемент: 5
    - Разделение: [1, 5, 8, 9, 10, 7]
    - Опорный элемент теперь на позиции 1
2. Рекурсивно для левой части [1]:
    - Уже отсортирован
3. Рекурсивно для правой части [8, 9, 10, 7]:
    - Опорный элемент: 7
    - Разделение: [7, 9, 10, 8]
    - Опорный элемент теперь на позиции 0
4. Рекурсивно для [9, 10, 8]:
    - Опорный элемент: 8
    - Разделение: [8, 10, 9]
    - Опорный элемент теперь на позиции 0

5. Рекурсивно для [10, 9]:
   - Опорный элемент: 9
   - Разделение: [9, 10]
   - Опорный элемент теперь на позиции 0
6. Рекурсивно для [10]:
   - Уже отсортирован

Итоговый отсортированный массив: [1, 5, 7, 8, 9, 10]

Сложность:

- Временная сложность:
  - В среднем: O(n log n)
  - В худшем случае: $O(n^2)$ (если опорный элемент всегда крайний)
- Пространственная сложность: O(log n) из-за рекурсии

4. Топологическая сортировка отношений.

Топологическая сортировка — это линейное упорядочивание вершин ориентированного ациклического графа (DAG) таким образом, что для каждого направленного ребра (u,v), вершина u идет раньше вершины v в упорядочивании.

Применение:

- Упорядочивание задач с зависимостями (например, расписание курсов)
- Определение порядка сборки программы (зависимости между модулями)
- Определение критического пути в планировании проекта

Алгоритм (основанный на DFS):

1. Создаем временную метку для каждой вершины (не посещена, в процессе обработки, обработана)
2. Для каждой не посещенной вершины выполняем DFS
3. В процессе DFS:
   - Помечаем текущую вершину как "в процессе обработки"
   - Рекурсивно обрабатываем всех непосещенных соседей
   - Помечаем текущую вершину как "обработана" и добавляем в результат
4. В конце переворачиваем результат

Реализация:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

class Graph {
private:
    int V;   // количество вершин
```

```
    vector<vector<int>> adj;  // список смежности

    // DFS для топологической сортировки
    void topologicalSortUtil(int v, vector<bool>& visited,
stack<int>& Stack) {
        // Помечаем текущую вершину как посещенную
        visited[v] = true;

        // Рекурсивно обходим все соседние вершины
        for (int i : adj[v]) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, Stack);
            }
        }

        // Помещаем текущую вершину в стек результатов
        Stack.push(v);
    }

public:
    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    // Добавление ребра в граф
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    // Топологическая сортировка
    void topologicalSort() {
        stack<int> Stack;
        vector<bool> visited(V, false);

        // Вызываем рекурсивную функцию для всех непосещенных вершин
        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, Stack);
            }
        }

        // Выводим содержимое стека
        cout << "Topological Sort: ";
        while (!Stack.empty()) {
            cout << Stack.top() << " ";
            Stack.pop();
        }
        cout << endl;
    }
};

int main() {
    // Пример графа
    Graph g(6);
```

```
    g.addEdge(5, 2);
    g.addEdge(5, 0);
    g.addEdge(4, 0);
    g.addEdge(4, 1);
    g.addEdge(2, 3);
    g.addEdge(3, 1);

    g.topologicalSort();

    return 0;
}
```

Алгоритм Кана (альтернативный подход):

1. Вычисляем входящую степень для каждой вершины
2. Помещаем все вершины с входящей степенью 0 в очередь
3. Пока очередь не пуста:
    - Извлекаем вершину из очереди и добавляем в результат
    - Уменьшаем входящую степень для всех соседей
    - Если входящая степень соседа стала 0, добавляем его в очередь

Реализация алгоритма Кана:

```
void topologicalSortKahn() {
    vector<int> inDegree(V, 0);

    // Вычисляем входящие степени всех вершин
    for (int u = 0; u < V; u++) {
        for (int v : adj[u]) {
            inDegree[v]++;
        }
    }

    // Создаем очередь и добавляем все вершины с входящей степенью 0
    queue<int> q;
    for (int i = 0; i < V; i++) {
        if (inDegree[i] == 0) {
            q.push(i);
        }
    }

    // Счетчик обработанных вершин
    int count = 0;
    vector<int> topOrder;

    // Обрабатываем вершины в очереди
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        topOrder.push_back(u);

        // Для всех соседних вершин уменьшаем входящую степень на 1
        for (int v : adj[u]) {
            if (--inDegree[v] == 0) {
                q.push(v);
            }
        }
        count++;
    }

    // Проверка на цикл
    if (count != V) {
        cout << "Graph contains a cycle!" << endl;
        return;
    }

    // Вывод результата
    cout << "Topological Sort: ";
    for (int i : topOrder) {
        cout << i << " ";
    }
    cout << endl;
}
```

Сложность:

- Временная сложность: O(V + E), где V - количество вершин, E - количество ребер
- Пространственная сложность: O(V)

5. Упорядоченный массив: включение, удаление элементов, метод двоичного поиска.

Упорядоченный массив — массив, элементы которого расположены в порядке возрастания (или убывания). Поддержание упорядоченного массива позволяет использовать эффективные алгоритмы поиска, такие как двоичный поиск.

Включение (вставка) элемента

При вставке нового элемента в упорядоченный массив необходимо:

1. Найти правильную позицию для нового элемента
2. Сдвинуть элементы, чтобы освободить место
3. Вставить новый элемент на нужное место

```
bool insertSorted(int arr[], int& n, int capacity, int value) {
    // Проверка на переполнение
    if (n >= capacity) {
        return false;
    }

    // Находим позицию для вставки
    int i;
    for (i = n - 1; (i >= 0 && arr[i] > value); i--) {
        arr[i + 1] = arr[i];  // Сдвигаем элементы вправо
    }

    arr[i + 1] = value;  // Вставляем элемент
    n++;  // Увеличиваем размер массива

    return true;
}
```

Сложность:

- Временная сложность: O(n) в худшем случае, когда новый элемент должен быть вставлен в начало массива
- Пространственная сложность: O(1)

Удаление элемента

При удалении элемента из упорядоченного массива:

1. Находим элемент, который нужно удалить
2. Сдвигаем все элементы правее его на одну позицию влево
3. Уменьшаем размер массива

```
bool deleteSorted(int arr[], int& n, int value) {
    // Находим индекс элемента (можно использовать двоичный поиск)
    int pos = binarySearch(arr, 0, n - 1, value);

    // Если элемент не найден
    if (pos == -1) {
        return false;
    }

    // Сдвигаем элементы влево
    for (int i = pos; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }

    n--;  // Уменьшаем размер массива
    return true;
}
```

Сложность:

- Временная сложность: O(log n) для поиска + O(n) для сдвига = O(n) в худшем случае
- Пространственная сложность: O(1)

Двоичный поиск

Двоичный поиск — алгоритм поиска элемента в отсортированном массиве путем деления интервала поиска пополам на каждом шаге.

Алгоритм:

1. Задаем левую (left) и правую (right) границы поиска
2. Пока left <= right:
    - Находим средний элемент: mid = (left + right) / 2
    - Если arr[mid] равен искомому значению, возвращаем mid
    - Если arr[mid] > значения, ищем в левой половине: right = mid - 1
    - Если arr[mid] < значения, ищем в правой половине: left = mid + 1
3. Если цикл закончился, элемент не найден, возвращаем -1

Реализация (итеративная):

```
int binarySearch(int arr[], int left, int right, int value) {
    while (left <= right) {
        int mid = left + (right - left) / 2;  // Избегаем
переполнения

        // Проверяем средний элемент
        if (arr[mid] == value) {
            return mid;  // Найден
        }

        // Если значение больше, игнорируем левую половину
        if (arr[mid] < value) {
            left = mid + 1;
        }
        // Если значение меньше, игнорируем правую половину
        else {
            right = mid - 1;
        }
    }

    // Элемент не найден
    return -1;
}
```

Реализация (рекурсивная):

```
int binarySearchRecursive(int arr[], int left, int right, int value)
{
    if (right >= left) {
        int mid = left + (right - left) / 2;

        // Если элемент находится в середине
        if (arr[mid] == value) {
            return mid;
        }

        // Если элемент меньше среднего, ищем в левой половине
        if (arr[mid] > value) {
            return binarySearchRecursive(arr, left, mid - 1, value);
        }

        // Иначе ищем в правой половине
        return binarySearchRecursive(arr, mid + 1, right, value);
    }

    // Элемент не найден
    return -1;
}
```

Сложность:

- Временная сложность: O(log n)
- Пространственная сложность: O(1) для итеративной версии, O(log n) для рекурсивной (из-за стека вызовов)

6. Функция сложности алгоритма. Эффективность алгоритма.

Функция сложности алгоритма

Функция сложности алгоритма — это математическая функция, которая выражает зависимость количества операций (или времени выполнения) алгоритма от размера входных данных. Обычно обозначается как T(n), где n — размер входных данных.

Асимптотическая сложность

Для оценки эффективности алгоритмов используют асимптотическую нотацию, которая позволяет абстрагироваться от деталей реализации и сосредоточиться на росте функции сложности при увеличении размера входных данных.

Основные асимптотические обозначения:

1. **O-нотация (верхняя граница)**:
   - $f(n) = O(g(n))$, если существуют константы $c > 0$ и n_0, такие что $f(n) \leq c \cdot g(n)$ для всех $n \geq n_0$
   - Описывает наихудший случай или максимальную сложность
2. **Ω-нотация (нижняя граница)**:
   - $f(n) = \Omega(g(n))$, если существуют константы $c > 0$ и n_0, такие что $f(n) \geq c \cdot g(n)$ для всех $n \geq n_0$
   - Описывает наилучший случай или минимальную сложность
3. **Θ-нотация (тесная граница)**:
   - $f(n) = \Theta(g(n))$, если $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$
   - Описывает точный порядок роста сложности

Распространенные классы сложности

1. **O(1)** — Константная сложность:
   - Время выполнения не зависит от размера входных данных
   - Примеры: доступ к элементу массива по индексу, математические операции
2. **O(log n)** — Логарифмическая сложность:
   - Алгоритмы, которые делят задачу на части
   - Примеры: двоичный поиск, сбалансированные деревья поиска
3. **O(n)** — Линейная сложность:
   - Время выполнения прямо пропорционально размеру входных данных

   - Примеры: линейный поиск, обход массива
4. **O(n log n)** — Линеарифмически-логарифмическая сложность:
   - Примеры: эффективные алгоритмы сортировки (быстрая сортировка, сортировка слиянием)
5. **O(n^2)** — Квадратичная сложность:
   - Примеры: простые алгоритмы сортировки (пузырьком, вставками)
6. **O(2^n)** — Экспоненциальная сложность:
   - Время выполнения удваивается с увеличением n на 1
   - Примеры: перебор всех подмножеств, решение задачи о рюкзаке методом перебора
7. **O(n!)** — Факториальная сложность:
   - Примеры: перестановки, задача коммивояжера методом перебора

Эффективность алгоритма

Эффективность алгоритма определяется несколькими факторами:

1. **Временная сложность**: количество операций или времени, необходимых для выполнения алгоритма в зависимости от размера входных данных
2. **Пространственная сложность**: объем памяти, необходимый для выполнения алгоритма
3. **Простота реализации**: сложность кода и возможность ошибок
4. **Скрытые константы и накладные расходы**: факторы, которые не учитываются в асимптотической нотации

Примеры анализа сложности алгоритмов

Линейный поиск:

```
int linearSearch(int arr[], int n, int value) {
    for (int i = 0; i < n; i++) {  // O(n) операций
        if (arr[i] == value) {
            return i;
        }
    }
    return -1;
}
```

- Временная сложность: O(n)
- Пространственная сложность: O(1)

Двоичный поиск:

```
int binarySearch(int arr[], int left, int right, int value) {
    while (left <= right) {  // O(log n) итераций
        int mid = left + (right - left) / 2;
        if (arr[mid] == value) return mid;
        if (arr[mid] < value) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

- Временная сложность: O(log n)
- Пространственная сложность: O(1)

Сортировка пузырьком:

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {       // O(n) итераций
        for (int j = 0; j < n-i-1; j++) { // O(n) итераций
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

- Временная сложность: $O(n^2)$
- Пространственная сложность: O(1)

7. Полиномиальные алгоритмы.

Полиномиальные алгоритмы — алгоритмы, временная сложность которых ограничена полиномиальной функцией от размера входных данных. Математически это можно выразить как $O(n^k)$, где k — некоторая константа.

Классы полиномиальных алгоритмов:

1. **O(1)** — Константная сложность
2. **O(log n)** — Логарифмическая сложность
3. **O(n)** — Линейная сложность
4. **O(n log n)** — Линеарифмически-логарифмическая сложность
5. **$O(n^2)$** — Квадратичная сложность
6. **$O(n^3)$** — Кубическая сложность
7. И т.д. для больших степеней k

Значение полиномиальных алгоритмов

Полиномиальные алгоритмы играют важную роль в теории сложности вычислений и практической информатике:

1. **Класс P**: Множество задач, которые можно решить за полиномиальное время. Эти задачи считаются "эффективно решаемыми".
2. **P vs NP**: Одна из важнейших открытых проблем в информатике — вопрос о том, может ли любая задача, решение которой можно проверить за полиномиальное время (класс NP), быть решена за полиномиальное время (класс P).

Примеры полиномиальных алгоритмов:

Линейный поиск (O(n)):

```
int linearSearch(int arr[], int n, int value) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == value) return i;
    }
    return -1;
}
```

Сортировка методом вставок ($O(n^2)$):

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Алгоритм Флойда-Уоршелла ($O(n^3)$):

```
void floydWarshall(int graph[][V]) {
    int dist[V][V];

    // Инициализация матрицы расстояний
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Обновление матрицы расстояний
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}
```

Эффективность полиномиальных алгоритмов

Хотя все полиномиальные алгоритмы считаются "эффективными" с точки зрения теории сложности, их практическая эффективность может значительно различаться:

1. **Малые степени полинома**: Алгоритмы с временной сложностью O(n) или O(n log n) обычно работают быстро даже на больших наборах данных.
2. **Большие степени полинома**: Алгоритмы с временной сложностью $O(n^3)$ или выше могут быть слишком медленными для практического применения на больших наборах данных.

Противопоставление неполиномиальным алгоритмам

Алгоритмы с неполиномиальной сложностью (например, $O(2^n)$ или O(n!)) растут настолько быстро с увеличением n, что становятся непрактичными для всех, кроме самых маленьких входных наборов данных:

- Для n = 10: 2^{10} = 1,024 (тысяча операций)
- Для n = 20: 2^{20} = 1,048,576 (миллион операций)
- Для n = 30: 2^{30} = 1,073,741,824 (миллиард операций)
- Для n = 100: $2^{100} \approx 10^{30}$ (астрономическое число)

Применение полиномиальных алгоритмов

Полиномиальные алгоритмы широко используются в различных областях:

1. **Поиск и сортировка**: линейный поиск, двоичный поиск, сортировка вставками, быстрая сортировка

2. **Теория графов**: обход в ширину, обход в глубину, алгоритм Дейкстры, алгоритм Флойда-Уоршелла
3. **Обработка строк**: наивный поиск подстроки, алгоритм Кнута-Морриса-Пратта
4. **Линейное программирование**: симплекс-метод, алгоритм Кармаркара
5. **Динамическое программирование**: задача о рюкзаке с псевдополиномиальным решением, задача о самой длинной общей подпоследовательности

8. Эффективные алгоритмы.

Эффективные алгоритмы — это алгоритмы, которые решают задачи за разумное время с использованием разумного количества ресурсов. В теории сложности алгоритмов эффективными обычно считаются алгоритмы полиномиальной сложности.

Характеристики эффективных алгоритмов:

1. **Оптимальная временная сложность**: минимальное количество операций для решения задачи
2. **Оптимальная пространственная сложность**: экономное использование памяти
3. **Масштабируемость**: способность эффективно работать при увеличении размера входных данных
4. **Устойчивость**: стабильность работы при различных входных данных

Примеры эффективных алгоритмов:

1. Двоичный поиск (O(log n)):

```
int binarySearch(int arr[], int left, int right, int value) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == value) return mid;
        if (arr[mid] < value) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

2. Быстрая сортировка (O(n log n) в среднем):

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

3. Динамическое программирование (пример - задача о рюкзаке):

```
int knapsack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];

    // Заполняем таблицу DP снизу вверх
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]],
dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }

    return dp[n][W];
}
```

4. Алгоритм Дейкстры (поиск кратчайших путей):

```cpp
void dijkstra(vector<vector<pair<int, int>>> graph, int src, int V) {
    priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
    vector<int> dist(V, INT_MAX);

    dist[src] = 0;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (auto& neighbor : graph[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}
```

Принципы разработки эффективных алгоритмов:

1. Разделяй и властвуй

Разбиение задачи на подзадачи, решение их независимо и объединение результатов.

- Примеры: быстрая сортировка, сортировка слиянием, двоичный поиск

2. Динамическое программирование

Решение сложных задач путем разбиения их на более простые подзадачи и сохранения результатов подзадач для избежания повторных вычислений.

- Примеры: задача о рюкзаке, нахождение наибольшей общей подпоследовательности

3. Жадные алгоритмы

Выбор локально оптимального решения на каждом шаге с надеждой, что это приведет к глобально оптимальному решению.

- Примеры: алгоритм Дейкстры, алгоритм Прима, алгоритм Крускала

4. Уменьшение константных множителей

Оптимизация алгоритма для снижения скрытых констант в асимптотической сложности.

- Примеры: оптимизация кода, предварительная обработка данных

5. Использование подходящих структур данных

Выбор структур данных, которые наилучшим образом соответствуют операциям, выполняемым алгоритмом.

- Примеры: хеш-таблицы для быстрого поиска, кучи для приоритетных очередей

Показатели эффективности:

1. **Временная сложность**: количество операций или время выполнения
2. **Пространственная сложность**: использование памяти
3. **Скрытые константы**: факторы, не отражаемые в асимптотической нотации
4. **Локальность данных**: эффективность использования кеша
5. **Параллелизм**: возможность параллельного выполнения

Оптимизация алгоритмов:

1. Алгоритмическая оптимизация

Изменение подхода к решению задачи для достижения лучшей асимптотической сложности.

- Пример: замена линейного поиска (O(n)) на двоичный поиск (O(log n))

2. Оптимизация реализации

Улучшение конкретной реализации алгоритма без изменения его асимптотической сложности.

- Примеры: развертывание циклов, минимизация операций ввода-вывода

3. Оптимизация данных

Использование подходящего представления данных.

- Примеры: упорядочивание данных, использование сжатия данных

Эффективность в реальном мире:

Асимптотическая нотация не всегда отражает реальную производительность:

- Алгоритм с худшей асимптотикой может быть быстрее на практике для малых наборов данных
- Скрытые константы и накладные расходы могут значительно влиять на производительность
- Кеширование, локальность данных и другие аспекты архитектуры компьютера играют важную роль

9. Способы оценки вычислительной сложности алгоритма.

Оценка вычислительной сложности алгоритма — процесс определения количества ресурсов (обычно времени и памяти), необходимых алгоритму в зависимости от размера входных данных.

1. Асимптотический анализ

Асимптотический анализ фокусируется на росте функции сложности при увеличении размера входных данных, абстрагируясь от констант и членов более низкого порядка.

Основные обозначения:

1. **O-нотация (верхняя граница)**:
   - f(n) = O(g(n)) означает, что f(n) растет не быстрее, чем g(n)
   - Формально: $\exists$ c > 0, n_0 > 0 такие, что f(n) ≤ c·g(n) для всех n ≥ n_0
2. **Ω-нотация (нижняя граница)**:
   - f(n) = Ω(g(n)) означает, что f(n) растет не медленнее, чем g(n)
   - Формально: $\exists$ c > 0, n_0 > 0 такие, что f(n) ≥ c·g(n) для всех n ≥ n_0
3. **Θ-нотация (точная граница)**:
   - f(n) = Θ(g(n)) означает, что f(n) растет асимптотически так же быстро, как g(n)
   - Формально: f(n) = O(g(n)) и f(n) = Ω(g(n))

Пример:

```
int sum(int arr[], int n) {
    int total = 0;  // 1 операция (присваивание)
    for (int i = 0; i < n; i++) {  // n+1 сравнений, n присваиваний
        total += arr[i];  // n операций (сложение и присваивание)
    }
    return total;  // 1 операция (возврат)
}
```

Всего: 1 + (n+1) + n + n + 1 = 3n + 3 операций Асимптотическая сложность: O(n)

2. Амортизационный анализ

Амортизационный анализ учитывает последовательность операций и распределяет стоимость дорогих операций на все операции в последовательности.

Методы амортизационного анализа:

1. **Метод агрегирования**: распределение общей стоимости операций по всем операциям
2. **Метод потенциалов**: введение "потенциальной энергии" для учета будущих дорогих операций

3. **Метод бухгалтерского учета**: распределение "кредитов" на будущие операции

Пример: Динамический массив

```
// Добавление элемента в конец динамического массива
void add(int value) {
    if (size == capacity) {
        // Если массив заполнен, увеличиваем его вдвое
        capacity *= 2;
        int* newArray = new int[capacity];
        for (int i = 0; i < size; i++) {
            newArray[i] = array[i];
        }
        delete[] array;
        array = newArray;
    }
    array[size++] = value;
}
```

Хотя в худшем случае операция add имеет сложность O(n) (когда требуется перевыделение памяти), амортизированная сложность составляет O(1), поскольку перевыделение происходит редко.

3. Вероятностный анализ

Вероятностный анализ оценивает ожидаемое время выполнения алгоритма при случайном распределении входных данных.

Пример: Быстрая сортировка

- Худший случай: $O(n^2)$
- Средний случай: O(n log n)
- Вероятность худшего случая очень мала при случайном выборе опорного элемента

4. Практические методы оценки

1. Подсчет операций

Подсчитывают основные операции в алгоритме и выражают их как функцию от размера входных данных.

```
// Подсчет количества сравнений в сортировке пузырьком
int bubbleSort(int arr[], int n) {
    int comparisons = 0;
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            comparisons++;  // Увеличиваем счетчик сравнений
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
            }
        }
    }
    return comparisons;  // Общее количество сравнений
}
```

2. Измерение времени выполнения

Измеряют фактическое время выполнения алгоритма для различных размеров входных данных.

```
#include <chrono>

void measureSortingAlgorithm(void (*sortFunc)(int[], int), int arr[],
int n) {
    // Копируем массив для сортировки
    int* copy = new int[n];
    for (int i = 0; i < n; i++) {
        copy[i] = arr[i];
    }

    // Замеряем время выполнения
    auto start = std::chrono::high_resolution_clock::now();
    sortFunc(copy, n);
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Time: " << elapsed.count() << " seconds" <<
std::endl;

    delete[] copy;
}
```

3. Профилирование

Использование профилировщиков для детального анализа производительности алгоритма:

- Определение "горячих точек" (частей кода, которые выполняются долго)
- Анализ использования памяти
- Выявление узких мест производительности

5. Анализ худшего, среднего и лучшего случаев

Худший случай

- Наибольшее время выполнения для входных данных заданного размера
- Обеспечивает верхнюю границу производительности
- Пример: сортировка пузырьком для обратно отсортированного массива — $O(n^2)$

Средний случай

- Ожидаемое время выполнения при случайном распределении входных данных
- Требует знания или предположения о распределении входных данных
- Пример: среднее время быстрой сортировки — O(n log n)

Лучший случай

- Наименьшее время выполнения для входных данных заданного размера
- Обычно наименее информативен, но иногда полезен
- Пример: сортировка пузырьком для уже отсортированного массива — O(n)

6. Оценка пространственной сложности

Помимо временной сложности, важно оценивать использование памяти:

1. Постоянная память (O(1))

Алгоритм использует фиксированное количество памяти, независимо от размера входных данных.

```
int findMax(int arr[], int n) {
    int max = arr[0];  // Одна дополнительная переменная
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

2. Линейная память (O(n))

Использование памяти пропорционально размеру входных данных.

```
int* createCopy(int arr[], int n) {
    int* copy = new int[n];  // Память пропорциональна n
    for (int i = 0; i < n; i++) {
        copy[i] = arr[i];
    }
    return copy;
}
```

3. Дополнительная память vs общая память

- **Дополнительная память**: память, используемая в дополнение к входным данным
- **Общая память**: общее количество используемой памяти, включая входные данные

7. Комбинированный анализ

В сложных алгоритмах может потребоваться комбинация различных методов анализа:

- Асимптотический анализ для общей структуры
- Амортизационный анализ для операций в последовательности
- Вероятностный анализ для алгоритмов с случайным поведением
- Практические измерения для конкретных реализаций