Алгоритмы и анализ сложности

1. Сортировка данных вставками. Пример.

Сортировка вставками – алгоритм, последовательно перемещающий элементы в отсортированную часть массива.

**Принцип работы:**

* Массив условно делится на отсортированную и неотсортированную части
* На каждом шаге берётся первый элемент из неотсортированной части
* Элемент вставляется в нужную позицию в отсортированной части

**Псевдокод:**

insertion\_sort(A):  
 for i = 1 to length(A) - 1  
 key = A[i]  
 j = i - 1  
 while j >= 0 and A[j] > key  
 A[j+1] = A[j]  
 j = j - 1  
 A[j+1] = key

**Характеристики:**

* Временная сложность: O(n²) в худшем и среднем случае, O(n) в лучшем
* Пространственная сложность: O(1)
* Стабильный алгоритм (сохраняет порядок равных элементов)
* Эффективен для небольших массивов и почти отсортированных данных

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

Структуры данных – способы организации данных для эффективного хранения и доступа.

**Основные структуры:**

**Массивы**

* + Непрерывная последовательность элементов одного типа
  + Доступ по индексу: array[i] за O(1)
  + Фиксированный размер (в большинстве языков)

**Связные списки**

* + Цепочка узлов с данными и указателями
  + Доступ: последовательный перебор O(n)
  + Вставка/удаление: O(1) при наличии указателя на узел

struct Node {  
 int data;  
 Node\* next;  
};

**Стеки и очереди**

* + Стек (LIFO): push(), pop(), peek() за O(1)
  + Очередь (FIFO): enqueue(), dequeue(), front() за O(1)

**Деревья**

* + Иерархическая структура с узлами-родителями и потомками
  + Двоичное дерево поиска: операции в среднем за O(log n)

struct TreeNode {  
 int data;  
 TreeNode\* left;  
 TreeNode\* right;  
};

**Хеш-таблицы**

* + Использует хеш-функцию для определения позиции элемента
  + Операции поиска, вставки, удаления: в среднем O(1)
  + Доступ по ключу: hashTable[key] или hashTable.get(key)

3. Сортировка методом «пузырька», разделением.

Сортировка пузырьком

**Принцип:** многократное прохождение по массиву с обменом соседних элементов, если они в неправильном порядке.

**Псевдокод:**

bubble\_sort(A):  
 for i = 0 to length(A) - 1  
 swapped = false  
 for j = 0 to length(A) - i - 1  
 if A[j] > A[j+1]  
 swap(A[j], A[j+1])  
 swapped = true  
 if not swapped  
 break

**Характеристики:**

* Временная сложность: O(n²) в худшем и среднем случае
* Пространственная сложность: O(1)
* Простая реализация, но неэффективен для больших массивов

Быстрая сортировка (разделением)

**Принцип:** "разделяй и властвуй" с выбором опорного элемента и разделением массива.

**Псевдокод:**

quicksort(A, low, high):  
 if low < high  
 pivot\_index = partition(A, low, high)  
 quicksort(A, low, pivot\_index - 1)  
 quicksort(A, pivot\_index + 1, high)  
  
partition(A, low, high):  
 pivot = A[high]  
 i = low - 1  
 for j = low to high - 1  
 if A[j] <= pivot  
 i = i + 1  
 swap(A[i], A[j])  
 swap(A[i + 1], A[high])  
 return i + 1

**Характеристики:**

* Временная сложность: O(n log n) в среднем, O(n²) в худшем случае
* Пространственная сложность: O(log n)
* Один из самых быстрых алгоритмов сортировки на практике

4. Топологическая сортировка отношений.

Топологическая сортировка – упорядочивание вершин ориентированного ациклического графа (DAG) так, чтобы для каждого ребра (u,v) вершина u шла перед v.

**Применение:**

* Планирование задач с зависимостями
* Определение порядка выполнения курсов
* Построение сборки программных модулей

**Алгоритм на основе DFS:**

function topological\_sort(G):  
 L = пустой список  
 S = множество всех вершин без входящих рёбер  
 while S не пусто:  
 выбрать вершину n из S  
 удалить n из S  
 добавить n в конец L  
 for each вершина m с ребром e от n к m:  
 удалить ребро e из графа  
 if m не имеет других входящих рёбер:  
 добавить m в S  
 if граф имеет рёбра:  
 return ошибка (граф имеет цикл)  
 else:  
 return L

**Характеристики:**

* Временная сложность: O(V+E), где V – количество вершин, E – количество рёбер
* Невозможна для графов с циклами
* Результат не всегда уникален (может быть несколько допустимых порядков)

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

Упорядоченный массив – массив с элементами, расположенными в порядке возрастания или убывания.

**Операции:**

**Включение (вставка) элемента**

* + Найти правильную позицию для вставки (бинарным поиском)
  + Сдвинуть все последующие элементы
  + Вставить элемент на нужную позицию
  + Сложность: O(n) из-за сдвига элементов

**Удаление элемента**

* + Найти элемент (бинарным поиском)
  + Сдвинуть все последующие элементы
  + Сложность: O(n) из-за сдвига элементов

**Метод двоичного поиска**

binary\_search(A, target, low, high):  
 while low <= high  
 mid = low + (high - low) / 2  
 if A[mid] == target  
 return mid  
 else if A[mid] < target  
 low = mid + 1  
 else  
 high = mid - 1  
 return -1 // Элемент не найден

* + Сложность: O(log n)
  + Значительно эффективнее линейного поиска O(n)

6. Функция сложности алгоритма. Эффективность алгоритма.

Функция сложности алгоритма – математическая функция, определяющая ресурсы (время, память), необходимые алгоритму в зависимости от размера входных данных.

**Виды сложности:**

* **Временная сложность** – количество операций
* **Пространственная сложность** – объем требуемой памяти

**Асимптотическая нотация:**

* **O (big-O)** – верхняя граница роста функции
* **Ω (big-Omega)** – нижняя граница роста функции
* **Θ (big-Theta)** – точная граница роста функции

**Классы сложности** (от наиболее к наименее эффективным):

* O(1) – константная (поиск в хеш-таблице)
* O(log n) – логарифмическая (бинарный поиск)
* O(n) – линейная (линейный поиск)
* O(n log n) – линеарифмическая (быстрая сортировка)
* O(n²) – квадратичная (сортировка вставками)
* O(2^n) – экспоненциальная (решение задачи коммивояжера)
* O(n!) – факториальная (перебор перестановок)

**Эффективность алгоритма** определяется:

* Временем выполнения
* Использованием памяти
* Простотой реализации
* Масштабируемостью при увеличении объема данных

7. Полиномиальные алгоритмы.

Полиномиальные алгоритмы – алгоритмы с временной сложностью O(n^k), где k – константа.

**Характеристики:**

* Практически реализуемы даже для больших входных данных
* Включают классы P (полиномиально разрешимые) и NP (недетерминированно полиномиальные)
* Противопоставляются экспоненциальным алгоритмам O(a^n), a > 1

**Примеры полиномиальных алгоритмов:**

* Сортировка вставками: O(n²)
* Быстрая сортировка: O(n log n)
* Алгоритм Дейкстры: O((V+E)log V)
* Поиск в глубину (DFS): O(V+E)
* Матричное умножение: O(n³)

8. Эффективные алгоритмы.

Эффективные алгоритмы – алгоритмы с оптимальным использованием ресурсов для решения задач.

**Характеристики:**

* Минимальная временная сложность
* Приемлемая пространственная сложность
* Масштабируемость
* Оптимальное соотношение времени и памяти

**Примеры эффективных алгоритмов:**

* Бинарный поиск: O(log n) вместо линейного O(n)
* Быстрая сортировка: O(n log n) вместо пузырьковой O(n²)
* Алгоритм Дейкстры: O((V+E)log V) вместо полного перебора O(V!)
* Алгоритм Крускала для минимального остовного дерева: O(E log E)
* Хеширование: поиск за O(1) вместо O(n)

9. Способы оценки вычислительной сложности алгоритма.

**Методы оценки:**

**Теоретический анализ**

* + Подсчет операций в псевдокоде
  + Выявление вложенных циклов и рекурсий
  + Определение доминирующих операций

**Асимптотический анализ**

* + Исследование поведения при больших n
  + Использование O, Ω, Θ нотаций
  + Игнорирование констант и младших слагаемых

**Анализ по случаям**

* + Худший случай (upper bound)
  + Средний случай (average case)
  + Лучший случай (lower bound)

**Эмпирическая оценка**

* + Измерение реального времени выполнения
  + Построение графиков зависимости времени от n
  + Сравнение с теоретическими прогнозами

**Амортизационный анализ**

* + Оценка среднего времени операций за длительный период
  + Применяется для структур с "дорогими" операциями (пример: динамические массивы)

**Инструменты анализа:**

* Математическое моделирование
* Рекуррентные соотношения
* Профилирование кода
* Бенчмаркинг