Лекция №13
по дисциплине
«ТЕОРИЯ АЛГОРИТМОВ»

СЛОЖНОСТЬ АЛГОРИТМОВ ПРОДОЛЖЕНИЕ

Преподаватель:
Золотоверх Д.О.

АЛГОРИТМЫ ОТЛИЧАЮТСЯ

Некоторые алгоритмы можно выполнить за секунды.

Некоторые алгоритмы не выполнимы за все время жизни вселенной.

Один алгоритм при выполнении задачи может использовать всю доступную память, когда другой – при увеличении количества входных данных использует ее постоянное количество.

ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ

Сложность алгоритма зависит от задачи (от ее размера и природы).

Измеряется в количестве работы выполненной алгоритмом:

- количество циклов работы процессора;
- количество времени;
- количество памяти.

При расчете сложности необходимо учитывать размер задачи.

Способ оценки сложности алгоритма

Нам необходимо найти количество операций O;

Количество операций, как было описано раньше, зависит от сложности (размера) задачи *n*;

Иногда в коде происходят действия, количество которых не зависит от размера выполняемой задачи – константные значения;

Как говорилось ранее, подсчет совершается для худшего исхода (перебор всех значений, последнее место в массиве).

АСИМПТОТИКА

Полный расчет сложности алгоритма может быть очень трудоемким.

Поэтому для его расчета применяется так называемое асимптотическое равенство.

Простыми словами – убираются малозначимые значения.

Когда задача становится очень большой – их значения незначимы.

Например:

$O = n + 4$ стремится к $O = n$

$O = \log_2(3n) + 2$ стремится к $\log_2(n)$

ТИПЫ АЛГОРИТМОВ

Существуют следующие типы сложности:

- O(1) – кол-во операций не растет с задачей;
- O(log n) – рост кол-ва замедляется с ростом задачи
- O(n) – рост кол-ва пропорционален росту задачи
- $O(n^2)$, $O(2^n)$, O(n!) – рост кол-ва ускоряется с простом задачи

График роста сложности алгоритмов от роста задачи

ПРИМЕРЫ АЛГОРИТМОВ

Сложность	Название алгоритма	Пример задачи
O(1)	Постоянный	Адресация, работа с хеш-таблицами, Работа с очередями
O(log n)	Логарифмический	Бинарный поиск (отсортированный список) Алгоритмы типа разделяй и властвуй
O(n)	Линейный	Перебор массива, Адресация связанного списка, Сравнение строк,
O(n log n)	Линейно-арифмический	Сортировки типа Merge Sort, Heap Sort, Quick Sort
$O(n^2)$	Квадратичный	Работа с двумерным массивом, Сортировки типа Bubble Sort, Insertion Sort, Selection Sort
$O(n^3)$	Кубический	Решение уравнений с 3 переменными
$O(k^n)$	Экспоненциальный	Нахождение всех подмножеств
O(n!)	Факториальный	Найти все перестановки заданного набора, Задача коммивояжера

ПРОБЛЕМА СОРТИРОВКИ

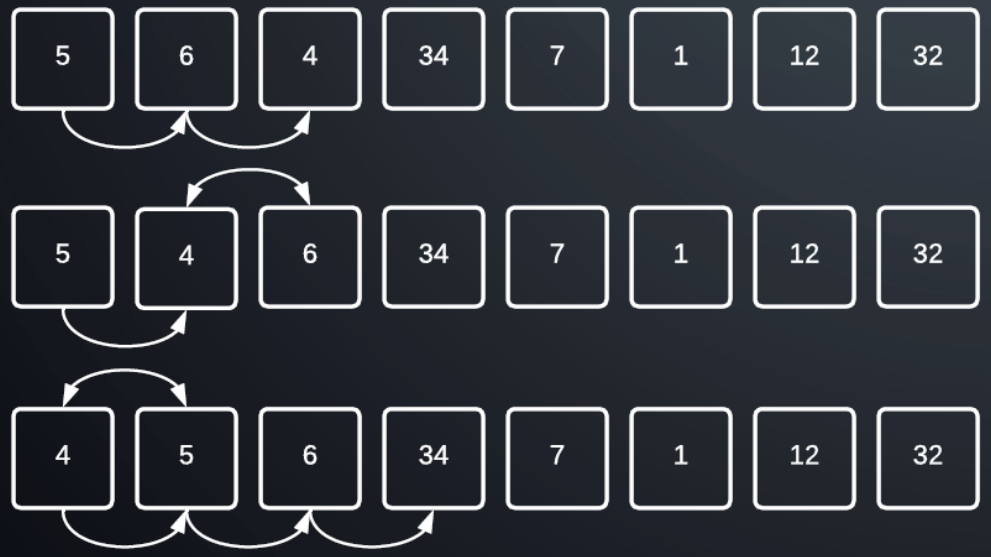
Дан массив целых чисел, необходимо разложить элементы таким образом, чтобы слева были меньше, справа – больше;

Желательно использовать меньше памяти;

Желательно использовать меньше операций.

ГЛУПАЯ СОРТИРОВКА

Переставляем элементы, пока не отсортируем весь массив.



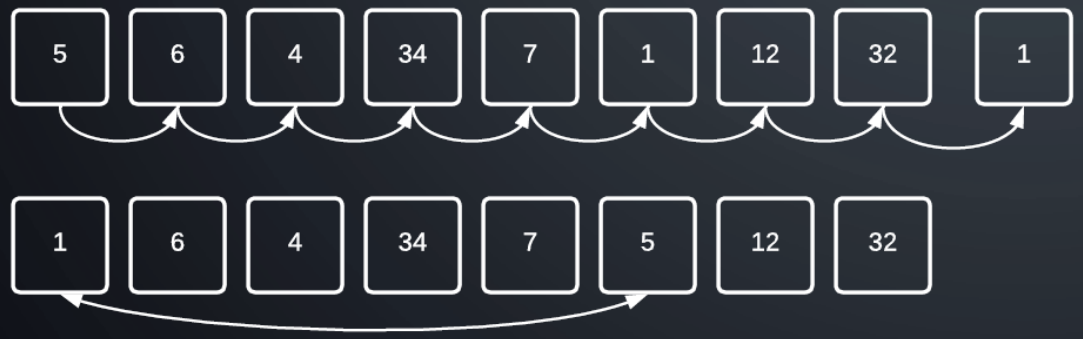
Как следует с названия, не самый лучший способ.

```
void stupidSort(int array[], int arrayLen)
{
    int i, tempBuffer;
    i = 1;
    while (i < arrayLen) {
        if (array[i - 1] > array[i]) {
            tempBuffer = array[i];
            array[i] = array[i - 1];
            array[i - 1] = tempBuffer;

            i = 1;
        } else {
            i++;
        }
    }
}
```

СОРТИРОВКА ВЫБОРОМ

Находим наименьший элемент, меняем его местами с первым элементом.



Таким образом для сортировки массива нам необходимо обойти его 8 раз, при этом придется перебрать все элементы массива.

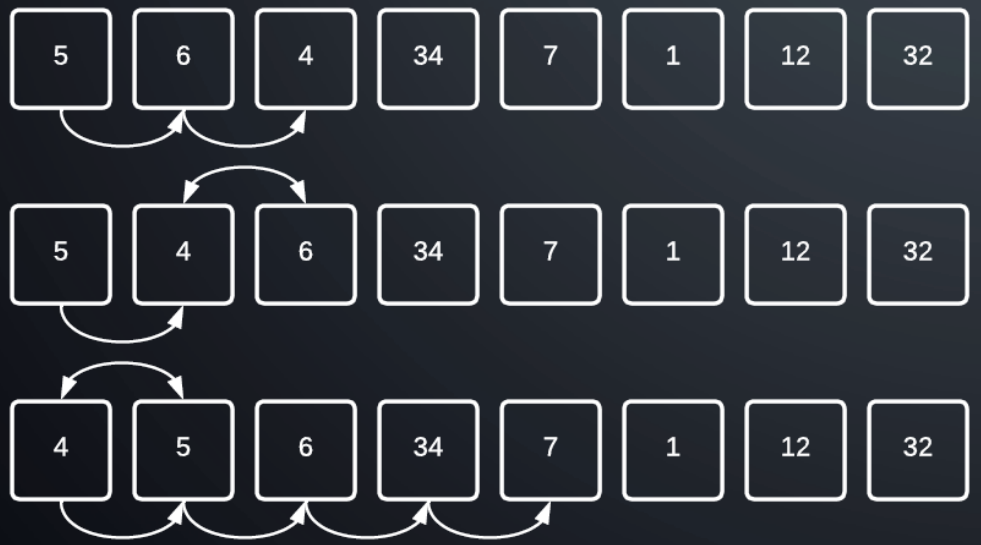
```
void selectionSort(int array[], int arrayLen)
{
    int minElementIndex, tempBuffer;

    for (int i = 0; i < arrayLen - 1; i++) {
        minElementIndex = i;
        for (int j = i + 1; j < arrayLen; j++) {
            if (array[j] <
                  array[minElementIndex]) {
            minElementIndex = j;
            }
        }

        tempBuffer = array[minElementIndex];
        array[minElementIndex] = array[i];
        array[i] = tempBuffer;
    }
}
```

ПУЗЫРЬКОВАЯ СОРТИРОВКА

Смотрим два ближайших элемента, меняем их местами, если нужно.




```
void bubbleSort(
    int array[], int arrayLen) {
    int tempBuffer;
    for (int i = 0;
         i < arrayLen - 1;
         i++) {
        for (int j = 0;
             j < arrayLen - i - 1;
             j++) {
            if (array[j] > array[j + 1]) {
                tempBuffer = array[j];
                array[j] = array[j + 1];
                array[j + 1] = tempBuffer;

            }}}}
```

Несмотря на то, что код проще, все равно нужно 8 раз обойти 8 элементов.

СОРТИРОВКА СЛИЯНИЕМ

Существует более замысловатый, но и более эффективный способ – сортировка слиянием.

Для уменьшения количества операций он использует как метод “разделяй и властвуй”, так и рекурсию.

Но как было сказано – он является сложным, а код алгоритма не интуитивен.

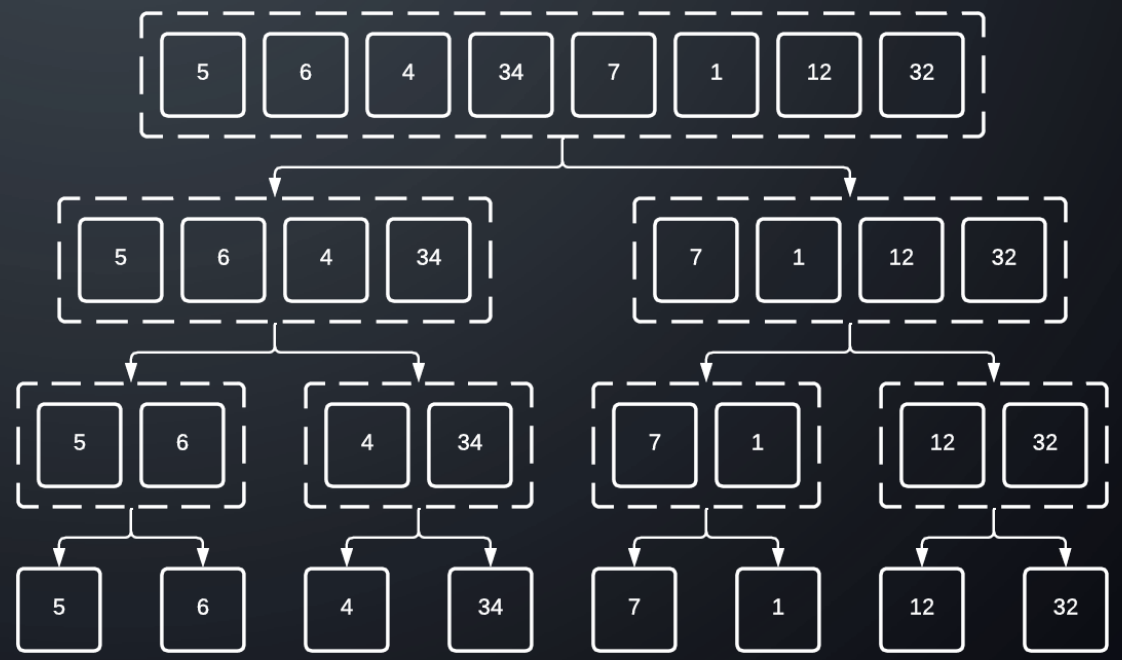
ПЕРВАЯ ЧАСТЬ – РАЗДЕЛЯЙ И ВЛАСТВУЙ

Первая часть алгоритма состоит из разбития массива в массивы меньшего размера.

Здесь и применим принцип «разделяй и властвуй».

В коде это делается не сразу, а по мере поступления проблемы, поэтому нельзя просто заранее поделить массив.

В самом алгоритме первая и вторая часть выполняются вместе.

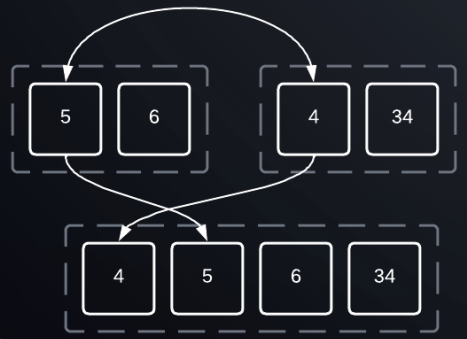


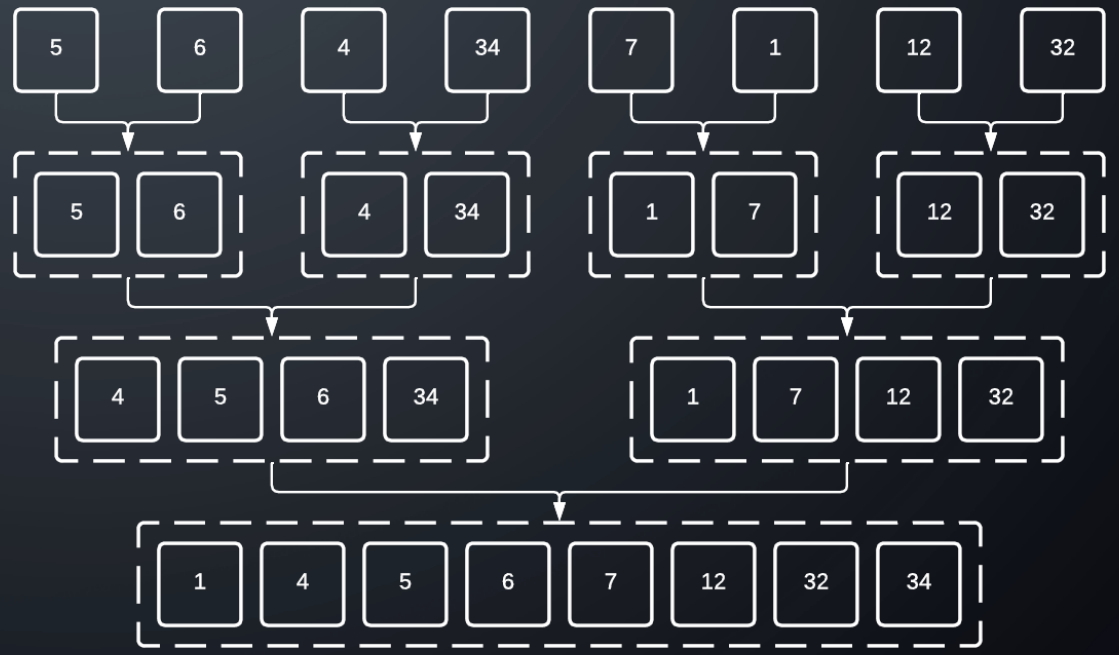
ВТОРАЯ ЧАСТЬ – СРАВНЕНИЕ И СОРТИРОВКА

Вторая часть – операции обратные первой, с сортировкой массивов поменьше.

После деление массивов в первой части, начинается их обратная сборка.

Сборка сопровождается поэтапным сравнением элементов:





ОЦЕНКА СЛОЖНОСТИ ГЛУПОЙ СОРТИРОВКИ

При запуске совершается 4 операции: выделение памяти с присвоением стартовых значений.

В цикле совершается как минимум одно действие – сравнение.

В случае необходимости перестановки – 3, иначе – 1;

В нашем примере было совершено 108 операций.

Но в худшем случае необходимо перебрать перебор 8 раз, поэтому имеем сложность:

$O = n^3$

ОЦЕНКА СЛОЖНОСТИ СОРТИРОВКИ ВЫБОРОМ

Характерным для данной сортировки является то, что мы постоянно ищем наименьшее число (обходим массив).

Такой обход повторяется столько раз, сколько элементов в массиве (8);

В данном примере, учитывая подготовки и операции в циклах, а также операции с перестановкой – их было совершено 66.

Имея вложенный цикл имеем квадратичную сложность:

$O = n^2$

ОЦЕНКА СЛОЖНОСТИ ПУЗЫРЬКОВОЙ СОРТИРОВКИ

Пузырьковая сортировка в исполнении немного проще, так как мы избавились от некоторого количества переменных.

Также в первом цикле пропал инкремент – операция которая могла в худшем случае выполнятся 8 раз.

В данном примере программа совершила 59 операций.

Но, как и сортировка выбором, мы имеем дело с вложенным цикл, а поэтому сложность также квадратичная:

$O = n^2$

ОЦЕНКА СЛОЖНОСТИ СОРТИРОВКИ ВСТАВКОЙ

Алгоритм достаточно сложен в своей реализации, поэтому нет смысла считать всё количество совершенных операций.

Но теоретически, можно предположить, что первая часть – разделение, как и двоичный поиск – имеет сложность $O = \log n$.

Вторая часть состоит из сборки – операции обратной двоичному разделению, а также со сравнения элементов каждого подмассива $O = n \log n$;

Применив асимптотику, сложность такого алгоритма равна:

$O = n \log n$

НО ЕСТЬ ОДНО НО

Посчитав операции, в реализации сортировки слиянием можно прийти к совершенно обратному результату:

Количество совершенных операций – **209**.

Это в два раза хуже, чем в глупой сортировки (108).

Проблема сложных, но эффективных алгоритмов в том, что они «разгоняются».

Они становятся «лучше» только когда проблема является «большой».

НО ЕСТЬ ОДНО НО

СПАСИБО ЗА ВНИМАНИЕ!