Лекция 14. Массивы. Алгоритмы

Курс «Программирование»
ВШИСиСТ, МОиАИС, ПрИнф

Щукин Александр Валентинович

Оценка эффективности алгоритмов

- Увеличение эффективности алгоритмов существенно сказывается на результате по сравнению например с аппаратными изменениями
- Любую задачу можно решить разными способами (алгоритмами, методами). А значит актуальным является вопрос сравнения (анализа)

Критерии сравнения

- По эффективности
  - Временной показатель
  - Показатель по памяти. Зависимость требуемой для работы программы памяти от размера (или других показателей) входных данных. Если объем выделяемой памяти = const, то это не представляет интереса (кроме тонкой настройки)
- Другие критерии (часто не количественные)
  - Масштабируемость и модифицируемость
  - Размер (мощность) множества входных данных
  - Легкость отладки (чтения)
  - Другое…

Временной критерий. Вычислительная сложность

- Оценка качества работы алгоритма по времени: в секундах? Проблема: разная среда выполнения, необходима оценка до практической реализации
- Решение – оценка количества действий.
  - = константа, тогда тонкая настройка
  - != константа, если алгоритм итерационный или рекурсивный
- Количество итераций (операций) зависит от входных данных (размера и информации). Нам интересно не само значение, а зависимость (функция, описывающая зависимость) количества итераций от входных данных

Оценка вычислительной сложности

- Входные данные: худший случай (O), лучший случай (o), средний.
- Операции с нотацией O
  - Вложенный вызов: O(F1) * O(F2)
  - Последовательный вызов: O(F1) + O(F2) = O(F1) (если F1 быстрее растет)
- Асимптотическая сложность. Классы алгоритмов P (полиномиально зависит) и NP
- Оценка времени выполнения алгоритма на

	N=10	N=20	N=30	N=40	N=50
N^3	0.001 с	0.008 с	0.027 с	0.064 с	0.125 с
2^N	0.001 с	1.05 с	17.9 мин	1. 29 дней	35.7 лет
3^N	0.059 с	58.1 мин	6.53 лет	$3.86 * 10^5$ лет	$2.28 * 10^{10}$ лет
$N!$	3.63 с	$7.71 * 10^4$ лет	$8.41 * 10^{18}$ лет	$2.59 * 10^{34}$ лет	$9.64 * 10^{50}$ лет

Задача поиска

- Задача поиска заключается в нахождении элемента в группе с заданным значением (заданным значением ключа). Ключ не всегда = значение элемента
- Возможные варианты результата работы:
  - Нашли
    - Ищем первый подходящий элемент = возвращаем его индекс
    - Ищем все подходящие элементы …
  - Не нашли

Линейный поиск

- Дано: одномерный массив int[] фиксированного размера, значение ключа num
- Выход: признак успешности, индекс

```
static bool LinearSearch(int[] arrItems, int num, out int index)
{
    for (index = 0; index < arrItems.GetLength(0); index++)
        if (arrItems[index] == num) return true;

    index = -1;
    return false;
}
```

Линейный поиск, несколько вхождений

- Находим все. Возвращаем массив индексов.

```
··static·int[]·LinearSearchMult(int[]·arrItems,·int·num)
··{
```

- Дополнительный входной аргумент – номер стартового элемента для поиска

```
·static·bool·LinearSearchMult2(int[]·arrItems,·int·num,·int·start,·out·int·index)
·{
```

- Самостоятельное задание

Линейный поиск с барьерным элементом

- Какие операции выполняются на каждой итерации? От какой операции можно отказаться?

```
static bool LinearSearch(int[] arrItems, int num, out int index)
{
    for (index = 0; index < arrItems.GetLength(0); index++)
        if (arrItems[index] == num) return true;

    index = -1;
    return false;
}
```

Линейный поиск с барьерным элементом

- Какие операции выполняются на каждой итерации? От какой операции можно отказаться?

```
static bool LinearSearch(int[] arrItems, int num, out int index)
{
    for (index = 0; index < arrItems.GetLength(0); index++)
        if (arrItems[index] == num) return true;

    index = -1;
    return false;
}
```

Линейный поиск с барьерным элементом

- Отказаться от проверки индекса

```
static bool LinearSearchBar(int[] arrItems, int num, out int index)
{
    int len = arrItems.Length;
    int lastItem = arrItems[len];
    arrItems[len] = num;
    index = 0;

    while (arrItems[index] != num) index++;

    if (index != len)
        return true;
    else if (lastItem == num)
        return true;
    else
    {
        arrItems[len] = lastItem;
        index = -1;
        return false;
    }
}
```

Второй вариант – изменить размер массива, добавив еще один элемент в конец

Оценка сложности линейного поиска

- Худший случай O(N)
- Лучший случай O(1)
- Средний случай
  - **n**-размер входных данных
  - **m**-число групп
  - **Pi**-вероятность того, что входные данные принадлежат группе с номером i
  - **ti**-время, необходимое алгоритму для обработки данных с номером i

$$A(n) = \sum_{i=1}^{m} p_i t_i \qquad A(n) = \frac{1}{m} \sum_{i=1}^{m} t_i$$

Двоичный поиск

- Как ускорить алгоритм поиска? Выполнить предобработку данных. Например, отсортировать.
- В упорядоченном массиве не нужно делать последовательный перебор элементов.

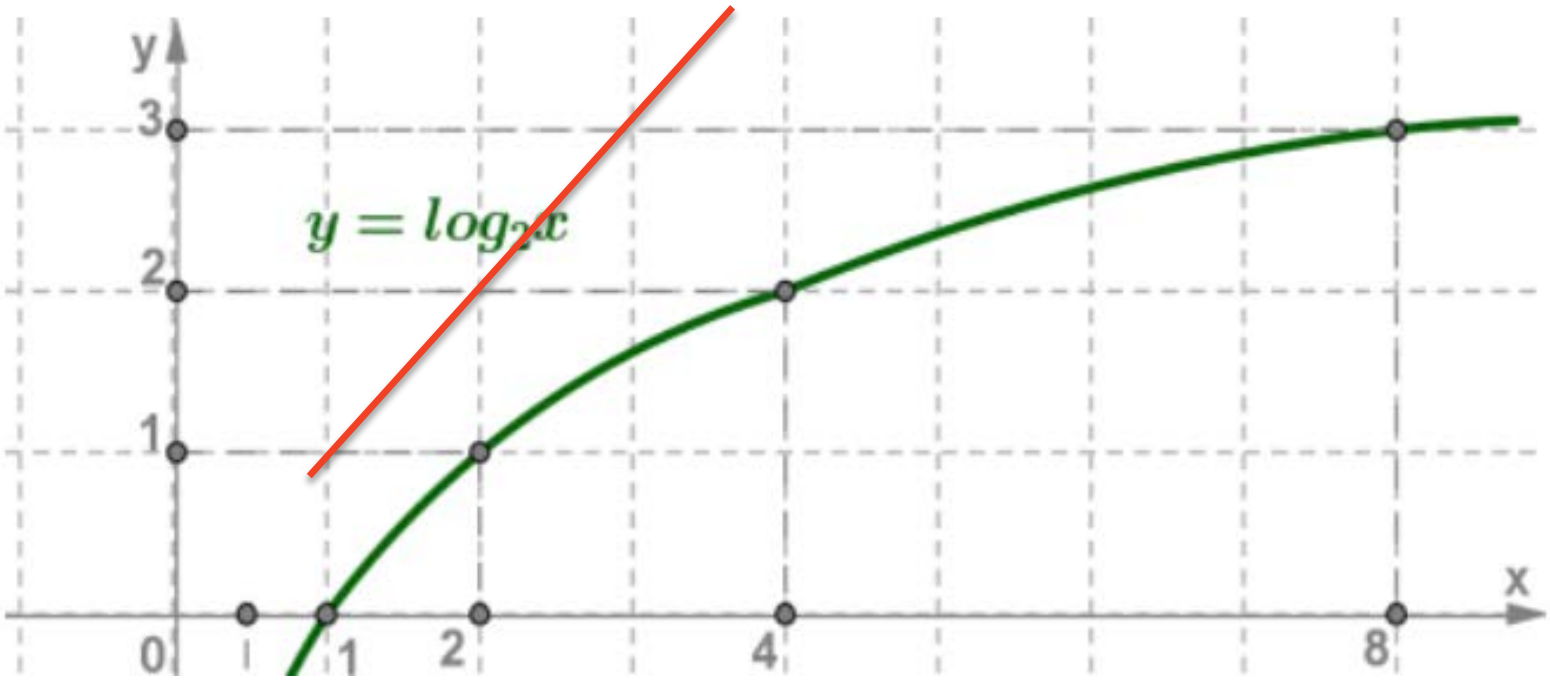
Двоичный поиск

```
static bool BinarySearch(int[] arrItems, int num, out int index)
{
    int left, right, mid;
    left = 0; right = arrItems.GetUpperBound(0);

    while (left <= right)
    {
        mid = (left + right) / 2;
        if (arrItems[mid] == num)
        { index = mid; return true; }
        else if (arrItems[mid] < num)
            right = mid - 1;
        else
            left = mid + 1;
    }
    index = -1;
    return false;
}
```

Двоичный поиск. Оценка сложности

- Количество бинарных разбиений O(log2 N)
- Бинарный поиск
  - + Ассимптотика лучше
  - - Нужно сортировать
- Линейный поиск
  - - Ассимптотика хуже
  - + Не нужно сортировать
- Размер массива. Нужно ли в будущем изменять данные. Сколько раз надо выполнять поиск.

Графики



Выборка

- Выборка – поиск элемента из множества, обладающего некоторым специальным свойствам, при этом его значение (ключ) не известно
- Выборка – поиск минимального, поиск максимального, поиск K-ого по величине и т.п.

Поиск максимального

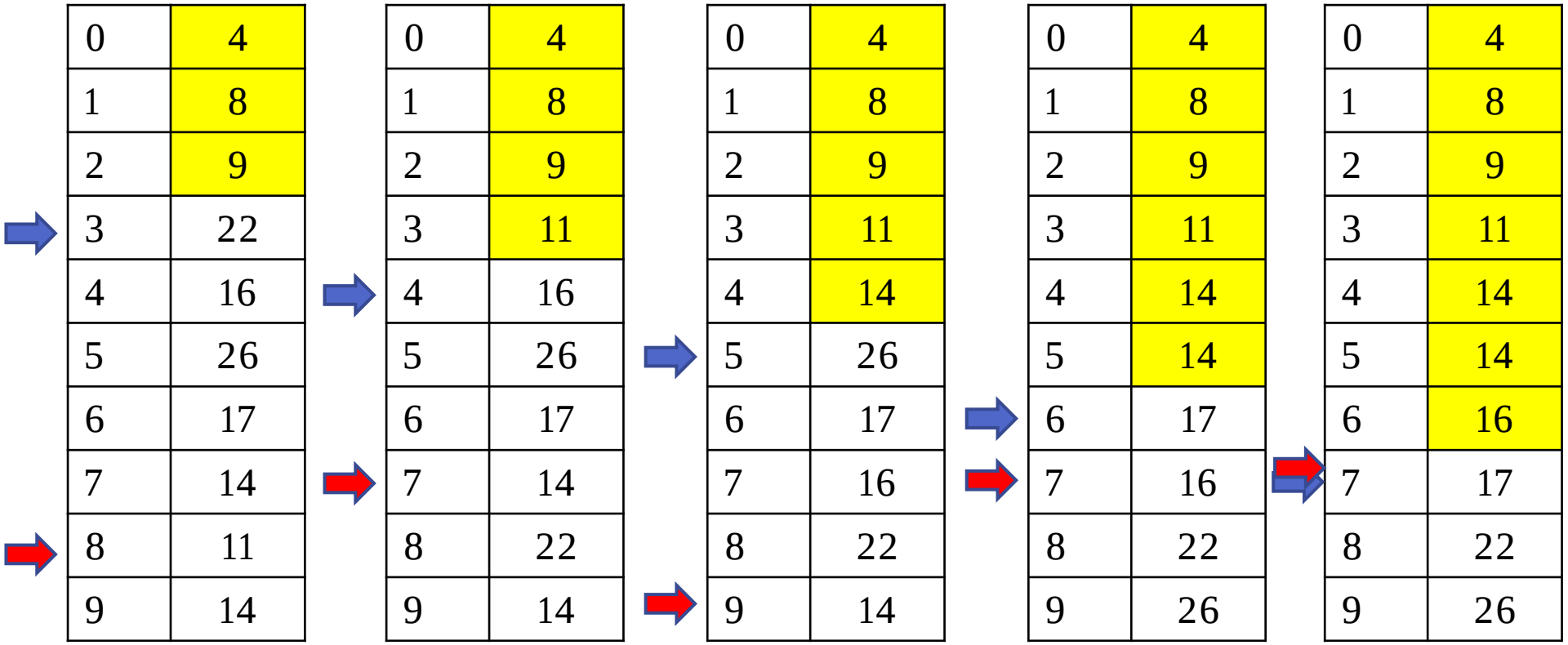
```
static int GetMax(int[] arrItems)
{
    int max = arrItems[0];
    int index = 0;

    for(int j = 1; j < arrItems.Length; j++)
        if (arrItems[j] > max)
        {
            max = arrItems[j];
            index = j;
        }
    return index;
}
```

Сортировка методом выбора

Массив делится на две части: отсортированную и нет. В начале работы алгоритма отсортированная часть пустая. Выполняются следующие операции:

1. Текущим устанавливается первый элемент в неотсортированной части.
2. Выполняется поиск минимального элемента в неотсортированной части.
3. Найденный минимальный элемент меняется местами с текущим элементом.
4. Неотсортированная часть массива уменьшается на один элемент. Если количество оставшихся элементов больше одного, то происходит переход на шаг 1. В противном случае выполняется выход из алгоритма.

Сортировка методом выбора



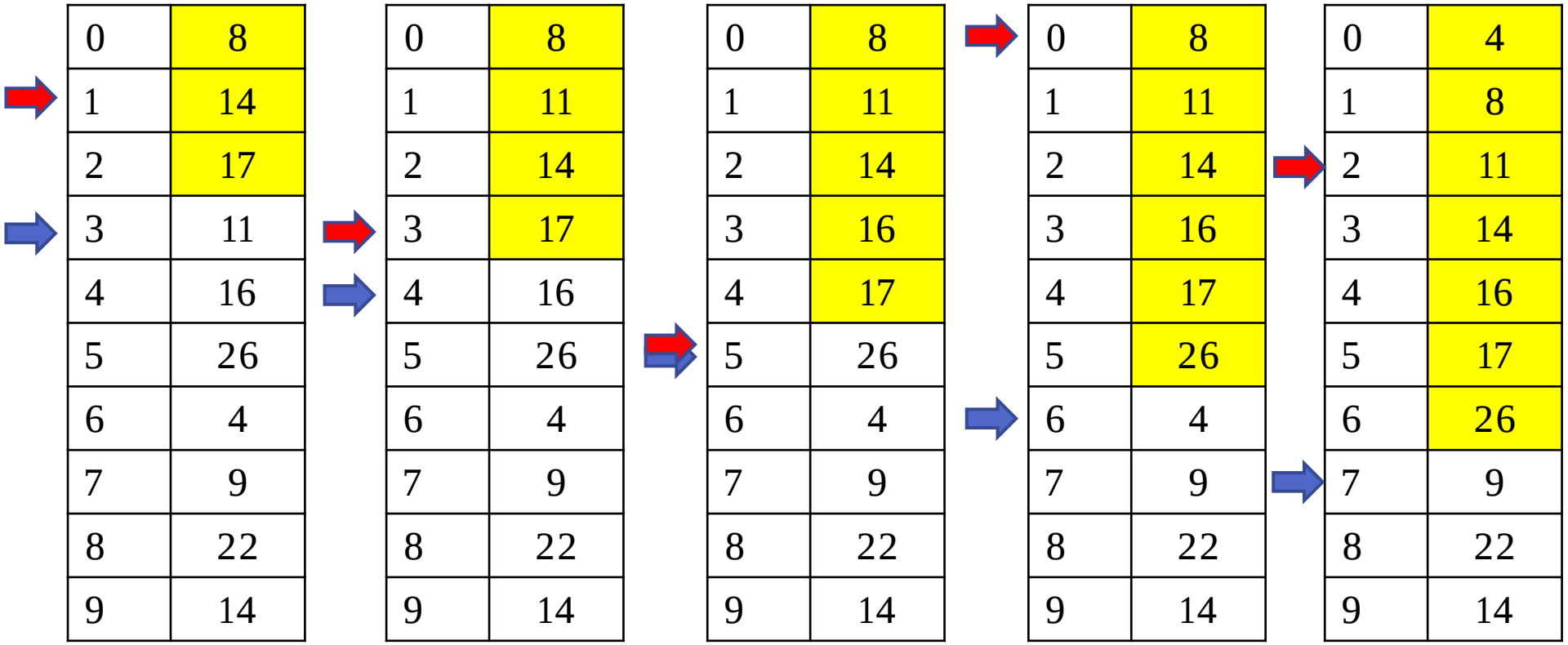
Оценка временной сложности -

Сортировка методом прямой вставки

Массив делится на две части: отсортированную и нет. В начале работы алгоритма отсортированная часть состоит из одного элемента. Выполняются следующие операции:

1. Текущим устанавливаем первый элемент из неотсортированной части.
2. В цикле определяется, в какое место в отсортированной части его необходимо вставить. При этом все элементы, перед которыми вставляется новое значение, последовательно сдвигаются на одну позицию вниз.
3. Неотсортированная часть массива уменьшается на один элемент. Если количество оставшихся элементов больше нуля, то происходит переход на шаг 1. В противном случае выполняется выход из алгоритма.

Сортировка методом прямой вставки



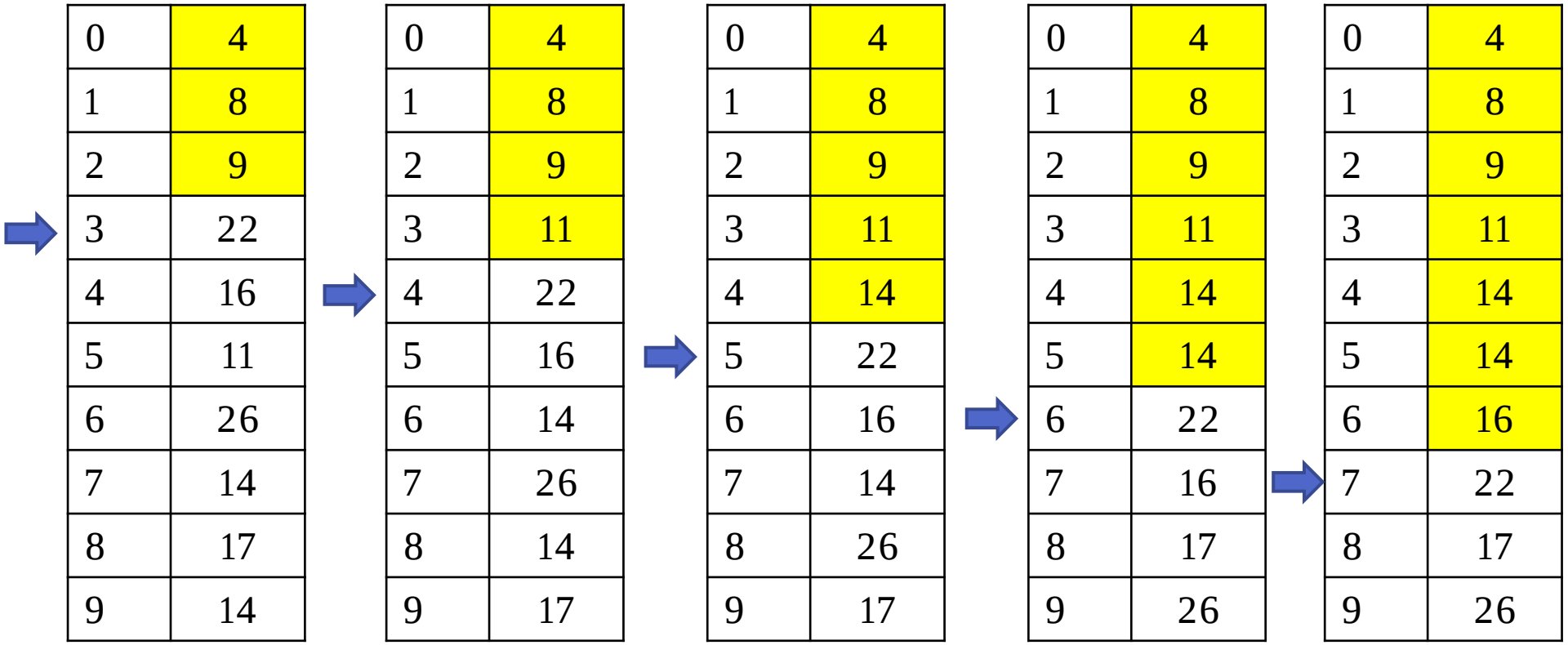
Оценка временной сложности -

Сортировка методом пузырька

Массив делится на две части: отсортированную и нет. В начале работы алгоритма отсортированная часть пустая. Выполняются следующие операции:

1. В цикле выполняется проход по элементам массива от первого элемента до предпоследнего в неотсортированной части. Текущим устанавливается самый первый элемент.
2. В процессе обхода элементов выполняется сравнение текущего элемента и следующего. Если текущий элемент больше следующего, то выполняется их перестановка.
3. Текущим становится следующий элемент массива.
4. После завершения цикла неотсортированная часть массива уменьшается снизу на один элемент. Если количество оставшихся элементов больше одного, то происходит переход на шаг 1.

Сортировка методом пузырька



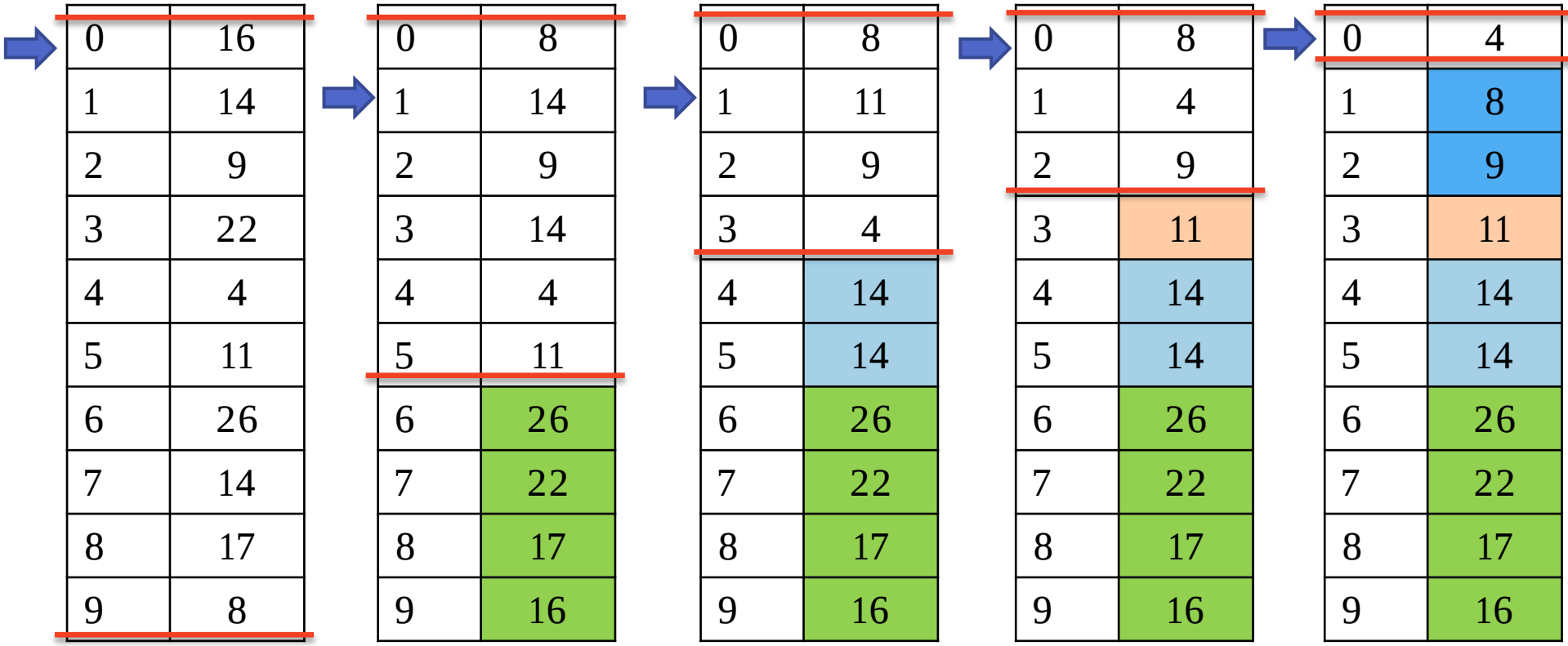
Оценка временной сложности -

Быстрая сортировка (сортировка Хоара)

Это рекурсивный алгоритм.

1. Выбрать из массива опорный элемент с которым на следующем шаге будет происходить сравнение.
2. Сравнить все элементы массива с опорным и переставить их так, чтобы элементы меньше опорного были выше него, а остальные элементы были ниже него
3. Для части массива выше опорного и части массива ниже опорного рекурсивно повторить действия с п. 1

Быстрая сортировка (сортировка Хоара)



Оценка временной сложности -