Одномерные статические массивы в C++

Часть 2. Алгоритмы обработки одномерных массивов

План лекции

1. Сдвиги

- влево с дополнением нулями, влево циклический;
- вправо с дополнением нулями, вправо циклический;
- циклический сдвиг элементов на k позиций – тривиальный и оптимизированный алгоритмы.

2. Сортировки: глупая, пузырьком, оптимизация пузырька, выбором, вставками, подсчетом.

3. Поиск элемента: линейный, бинарный, левый и правый бинарный поиски.

2 + 3. Сортировка бинарными вставками.

4. Одномерное динамическое программирование: числа Фибоначчи, префиксные суммы, задача о кузнечике, минимальная стоимость пути кузнечика

Задача сдвига элементов массива

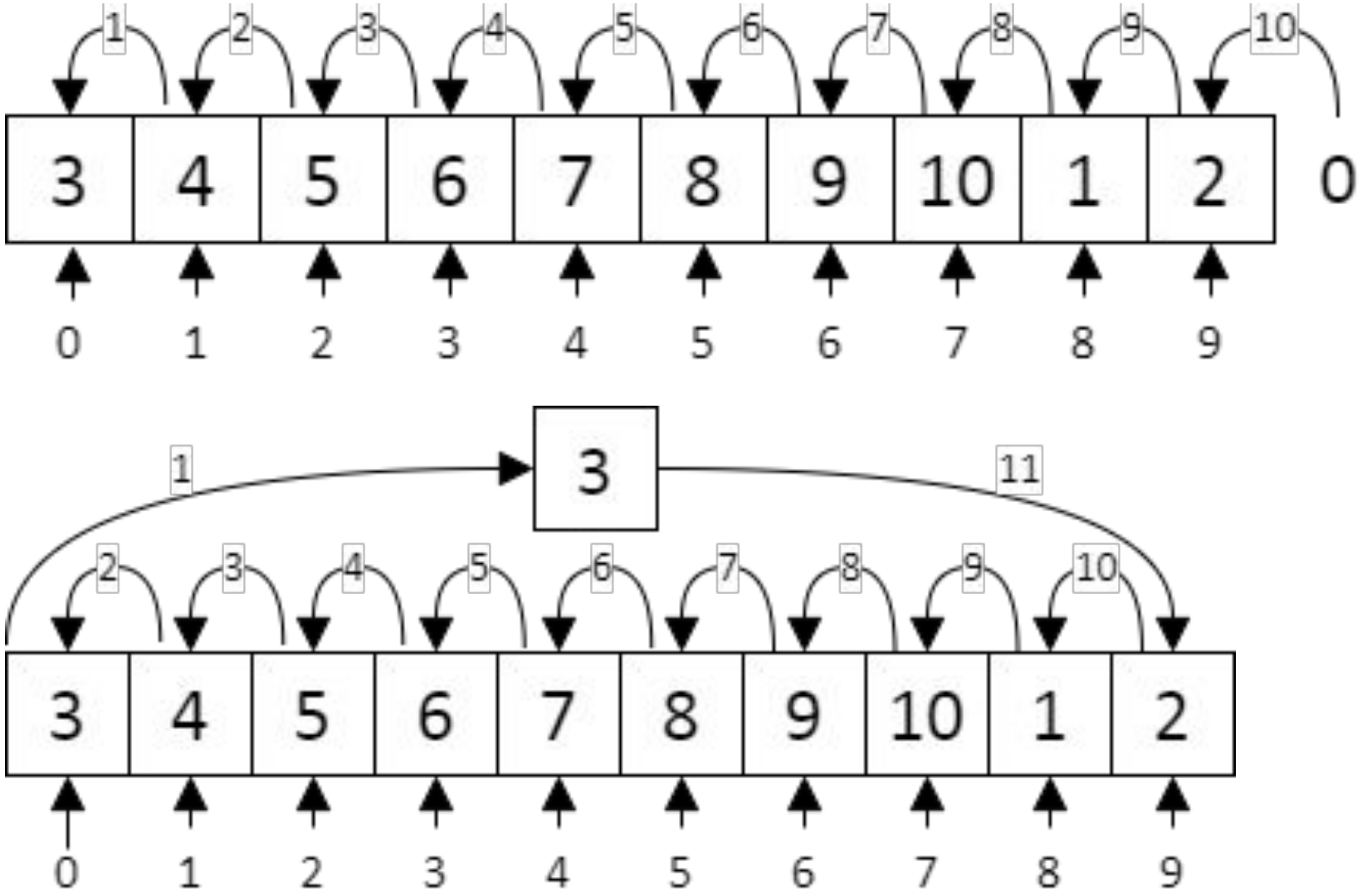
Сдвиг элементов массива – это процесс изменения позиции элементов массива на определенное количество мест в одном направлении.

Применимость алгоритма:

- при «удалении» элемента из начала массива - сдвига влево;
- при «вставке» элемента в середину массива - сдвига вправо;
- при моделировании поведения «циклической очереди» (конвейер на производстве) - циклический сдвиг.

Кроме того, эта техника широко применяется в шифровании данных, задачах обработки изображений, обнаружении образов и других областях программирования.

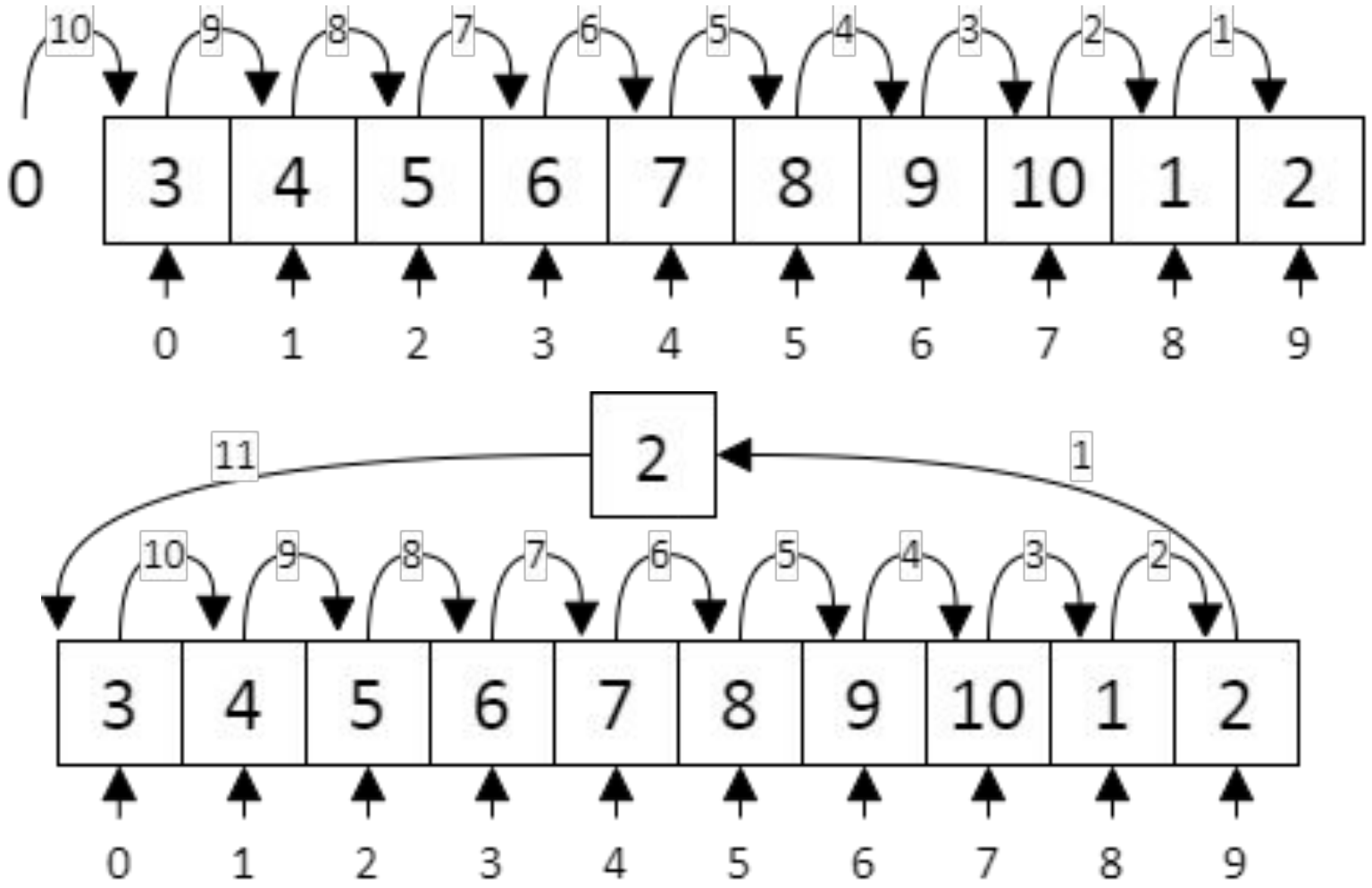
Сдвиг элементов массива влево на 1 ячейку



Реализация сдвига элементов массива влево на 1 ячейку

```
//сдвиг с дополнением
for (int i = 0; i < n - 1; i++)
   mas[i] = mas[i + 1];
mas[n - 1] = 0;

//циклический сдвиг
int first = mas[0];
for (i = 0; i < n - 1; i++)
   mas[i] = mas[i + 1];
mas[n - 1] = first;

//альтернативная реализация
for (i = 1; i < n; i++)
   mas[i - 1] = mas[i];
```

Сдвиг элементов массива вправо на 1 ячейку

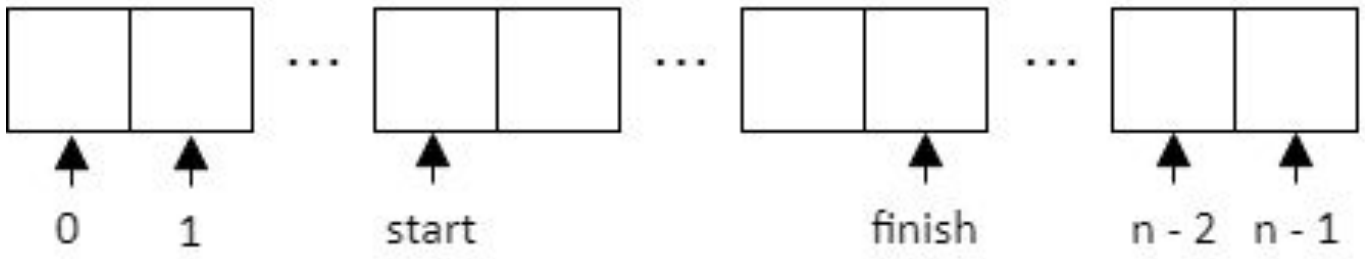


Реализация сдвига элементов массива вправо на 1 ячейку

```
//сдвиг с дополнением
for (i = n - 1; i > 0; i--)
   mas[i] = mas[i - 1];
mas[0] = 0;

//циклический сдвиг
int last = mas[n - 1];
for (i = n - 1; i > 0; i--)
   mas[i] = mas[i - 1];
mas[0] = last;

//альтернативная реализация
for (i = n - 2; i >=0; i--)
   mas[i + 1] = mas[i];
```

Циклический сдвиг между start и finish

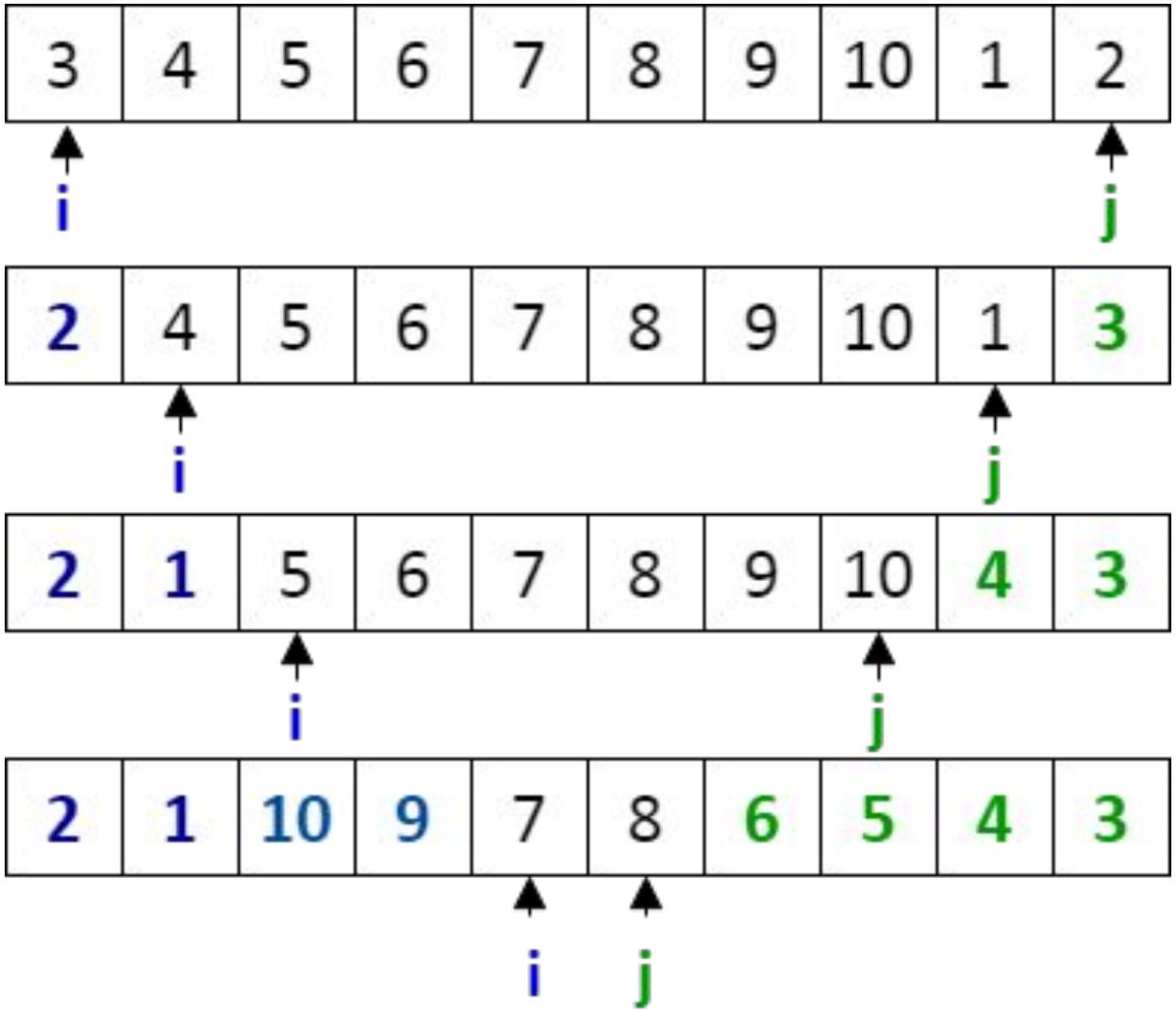
```
//left shift
int first = a[start];
for(int i = start; i < finish; i++){
   int &left = a[i];
   int &right = a[i + 1];
   left = right;
   }
a[finish] = first;

//right shift
int last = a[finish];
for(int i = finish; i > start; i--){
   int &left = a[i - 1];
   int &right = a[i];
   right = left;
   }
a[start] = last;
```



Циклический сдвиг элементов массива влево на ***k* позиций**

```
//тривиальный алгоритм – расход времени
for(int j = 1; j <= k; j++){
   int first = mas[0];
   for(int i = 0; i < n - 1; i++)
      mas[i] = mas[i + 1];
   mas[n - 1] = first;
   }
//тривиальный алгоритм – расход памяти
int temp[k];
for (int i = 0; i < k; i++)
   temp[i] = mas[i];
for (int i = 0; i < n - k; i++)
   mas[i] = mas[i + k];
for (int i = n - k; i < n; i++)
   mas[i] = temp[i – (n – k)];
```

Инвертирование элементов




```
for (int i = 0,
   j = n - 1;
   i < j;
   i++, j--)
    swap(mas[i], mas[j]);

/*
for (int i = 0;
   i < n / 2;
   i++)
    swap(mas[i],
       mas[n - 1 - i]);
*/
```

Оптимизированный алгоритм циклического сдвига на *k* позиций - инвертированием отрезков

Циклический сдвиг вправо:
1) инвертировать начальный отрезок массива длины n-k;
2) инвертировать конечный отрезок длины k;
3) инвертировать весь массив.
Исходный массив:
3, 4, 5, 6, 7, 8, 9, 10, 1, 2 □ k = 3
Сначала инвертируем начало длины 7:
9, 8, 7, 6, 5, 4, 3, 10, 1, 2
Затем инвертируем конец длины 3:
9, 8, 7, 6, 5, 4, 3, 2, 1, 10
На третьем шаге инвертируем весь массив:
10, 1, 2, 3, 4, 5, 6, 7, 8, 9

* для сдвига влево необходимо выполнить обратную последовательность действий

Реализация циклического сдвига инвертированием отрезков

Предположим, что положительные k означают сдвиг вправо, а отрицательные – влево

```
k %= n;
if(k < 0)
    k += n;
for (int i = 0, j = n - 1 - k; i < j; i++, j--)
    swap(mas[i], mas[j]);
for (int i = n - k, j = n - 1; i < j; i++, j--)
    swap(mas[i], mas[j]);
for (int i = 0, j = n - 1; i < j; i++, j--)
    swap(mas[i], mas[j]);
```

Сдвиг орбитами – **самостоятельное изучение O(n)**

http://mech.math.msu.su/~vvb/1course/index.html#CyclicShiftKOrbits

Текущий контроль

Сколько операций присваивания применительно к ячейкам массива необходимо **выполнить** при реализации следующих алгоритмов сдвига для массива размера n?

1) циклический сдвиг на одну ячейку влево;
2) сдвиг с дополнением на одну ячейку вправо;
3) сдвиг с дополнением нулем на две ячейки вправо;
4) циклический сдвиг на k ячеек вправо, тривиальный алгоритм с вложенным циклом;
5) циклический сдвиг на k ячеек влево, тривиальный алгоритм с расходом дополнительной памяти;
6) циклический сдвиг на k ячеек вправо инвертированием отрезков.

Текущий контроль

Сколько операций присваивания применительно к ячейкам массива необходимо **выполнить** при реализации следующих алгоритмов сдвига для массива размера n?

1) циклический сдвиг на одну ячейку влево; **n + 1**
2) сдвиг с дополнением на одну ячейку вправо; **n**
3) сдвиг с дополнением нулем на две ячейки вправо; **n**
4) циклический сдвиг на k ячеек вправо, тривиальный алгоритм с вложенным циклом; **k * (n + 1)**
5) циклический сдвиг на k ячеек влево, тривиальный алгоритм с расходом дополнительной памяти; **n + k**
6) циклический сдвиг на k ячеек вправо инвертированием отрезков. **2 * n**

Что будет выведено на экран в результате работы программы?

```
const int n = 10;
int mas[n] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (int i = 0; i + 2 < n; i++)
   mas[i] = mas[i + 2];
for (int i = 0; i < n; i++)
   cout << mas[i] << ' ';
```

а) 3 2 5 4 7 6 9 8 9 10

б) 2 1 4 3 6 5 8 7 10 9

в) 3 4 5 6 7 8 9 10 0 0

г) 3 4 5 6 7 8 9 10 9 10

д) 1 4 3 6 5 8 7 10 9 10

Что будет выведено на экран в результате работы программы?

```
const int n = 10;
int mas[n] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (int i = 0; i + 2 < n; i++)
   mas[i] = mas[i + 2];
for (int i = 0; i < n; i++)
   cout << mas[i] << ' ';
```

а) 3 2 5 4 7 6 9 8 9 10

б) 2 1 4 3 6 5 8 7 10 9

в) 3 4 5 6 7 8 9 10 0 0

г) 3 4 5 6 7 8 9 10 9 10

д) 1 4 3 6 5 8 7 10 9 10

Что будет выведено на экран в результате работы программы?

```
const int n = 10;
int mas[n] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (int i = 0; i + 2 < n; i += 2)
   mas[i] = mas[i + 2];
for (int i = 0; i < n; i++)
   cout << mas[i] << ' ';
```

а) 3 2 5 4 7 6 9 8 9 10

б) 2 1 4 3 6 5 8 7 10 9

в) 3 4 5 6 7 8 9 10 0 0

г) 3 4 5 6 7 8 9 10 9 10

д) 1 4 3 6 5 8 7 10 9 10

Что будет выведено на экран в результате работы программы?

```
const int n = 10;
int mas[n] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
for (int i = 0; i + 2 < n; i += 2)
   mas[i] = mas[i + 2];
for (int i = 0; i < n; i++)
   cout << mas[i] << ' ';
```

а) 3 2 5 4 7 6 9 8 9 10

б) 2 1 4 3 6 5 8 7 10 9

в) 3 4 5 6 7 8 9 10 0 0

г) 3 4 5 6 7 8 9 10 9 10

д) 1 4 3 6 5 8 7 10 9 10

Задача сортировки массива

Сортировка — это перестановка элементов исходного массива таким образом, чтобы каждый элемент был не меньше предыдущего в случае сортировки по возрастанию или не больше предыдущего в случае сортировки по убыванию.

Если в исходном массиве был повторяющийся элемент, в отсортированном массиве этот элемент будет повторяться столько же раз.

Отношение порядка элементов должно обладать следующими свойствами:

- любой элемент x не больше себя самого (рефлексивность);
- если элемент x не больше y и y не больше x, то x = y (антисимметричность);
- если элемент x не больше y и y не больше z, то x не больше z (транзитивность).

Задав порядок, который обладает этими свойствами, можем по нему произвести сортировку массива.

Примеры отсортированных последовательностей

Алфавит - образует собой порядок букв. Мы говорим, что буква *x* не больше *y*, если *x* идет не позже *y* в алфавите. Условия 1–3 в этом случае выполняются.

Список фамилий обычно приводится в так называемом лексикографическом порядке: слова, начинающиеся с одной буквы, дальше сравниваются по следующим буквам, пока не дойдем до разных букв. И уже по ним упорядочиваем слова. Например, фамилии *Аношин* и *Антошин* имеют одинаковое начало *Ан*. Однако на 3-й позиции в первой фамилии стоит буква *о*, а во второй фамилии — буква *т*, и поскольку в алфавите *о* идет раньше *т*, *Аношин* меньше в нашем порядке, чем *Антошин*.

Рейтинг текущей успеваемости студентов по дисциплине «Основы алгоритмизации и программирования» для определения номера индивидуального варианта для предстоящей лабораторной работы – образует собой порядок целых чисел.

Текущий контроль

Как может выглядеть отсортированный массив [3, 6, 1, 9]?

а) [9, 6, 3, 1]

б) [6, 3, 9, 1]

в) [1, 3, 6, 9]

г) [3, 1, 9, 6]

Текущий контроль

Как может выглядеть отсортированный массив [3, 6, 1, 9]?

а) [9, 6, 3, 1]

б) [6, 3, 9, 1]

в) [1, 3, 6, 9]

г) [3, 1, 9, 6]

Глупая (обезьянья) сортировка

Носит теоретический характер.

Генерируются все возможные перестановки входных данных, пока не получатся отсортированная последовательность.

Например, для элементов массива: 5, 4, 3, 2, 1 будем проверять отсортированность по возрастанию следующих последовательностей:

5, 4, 3, 2, 1

5, 4, 3, 1, 2

5, 4, 2, 3, 1

5, 4, 2, 1, 3

5, 4, 1, 3, 2

5, 4, 1, 2, 3

…

Всего $n!$ = 120 последовательностей, для каждой по n проверок.

Сортировка пузырьком

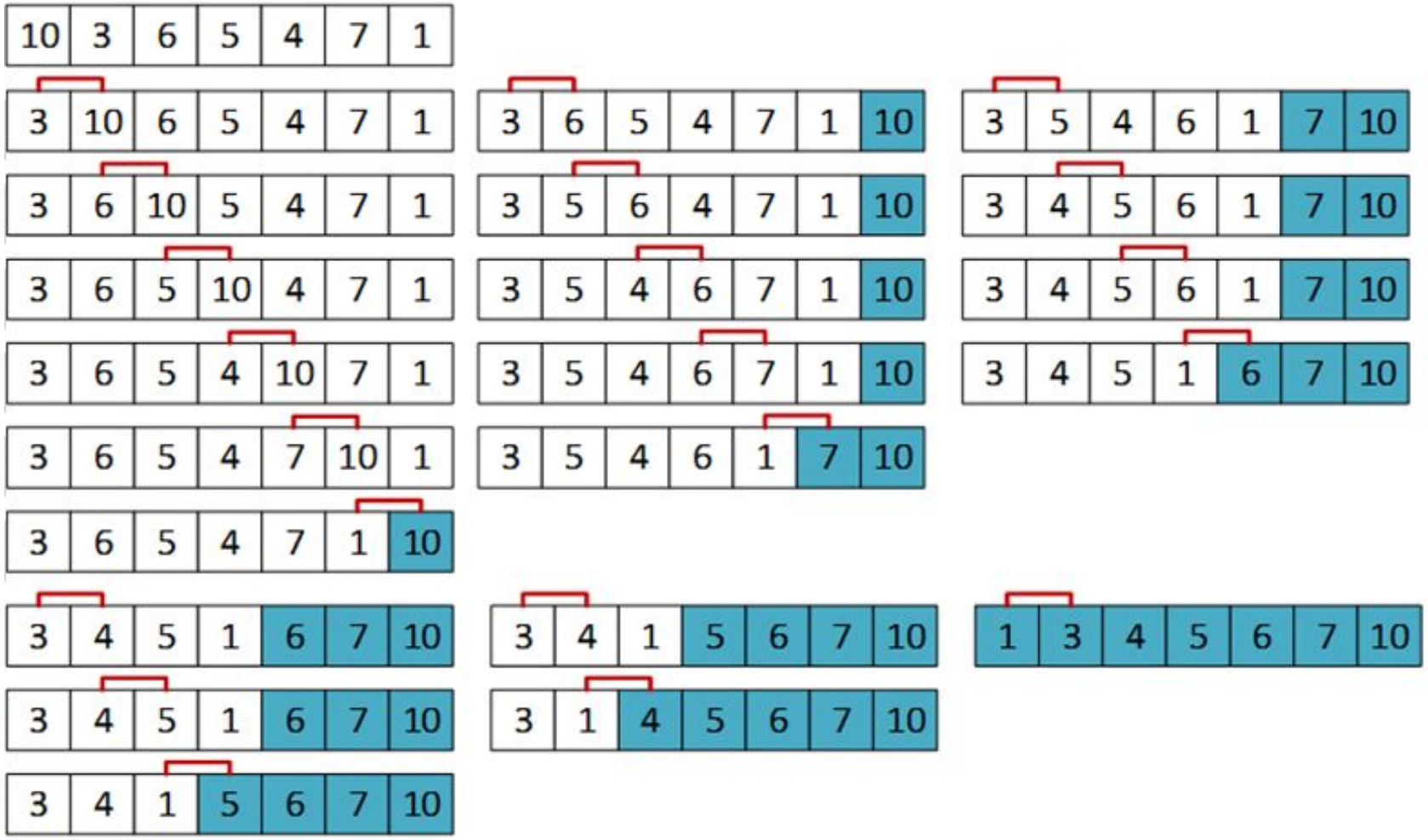
Идея сортировки заключается в последовательном сравнении значений соседних элементов. Если текущий элемент больше следующего, меняем их местами (для сортировки по возрастанию).

В лучшем случае, когда массив полностью отсортирован, мы совершим одну итерацию и алгоритм остановится. Временная сложность O(n)

В худшем случае, когда массив будет отсортирован в обратном порядке, придется выполнять порядка n итераций Временная сложность O(n^2)

Сортировка пузырьком

Происходит последовательное сравнение значений соседних элементов. Если текущий элемент больше (меньше) следующего, меняем их местами.



Сортировка пузырьком. Оптимизация 1

Для каждой итерации **фиксируется** последний элемент, участвующий в обмене. На очередном проходе этот элемент и все последующие в сравнении участвовать не будут.

```
for (int i = 1; i < n; i++)
    for (int j = 0; j < n - i; j++)
        if (mas[j] > mas[j + 1])
            swap(mas[j], mas[j + 1]);
```

Сортировка пузырьком. Оптимизация 2

Для каждой итерации проводится проверка выполнения хотя бы одной перестановки. Если **перестановок не было**, то массив уже упорядочен.

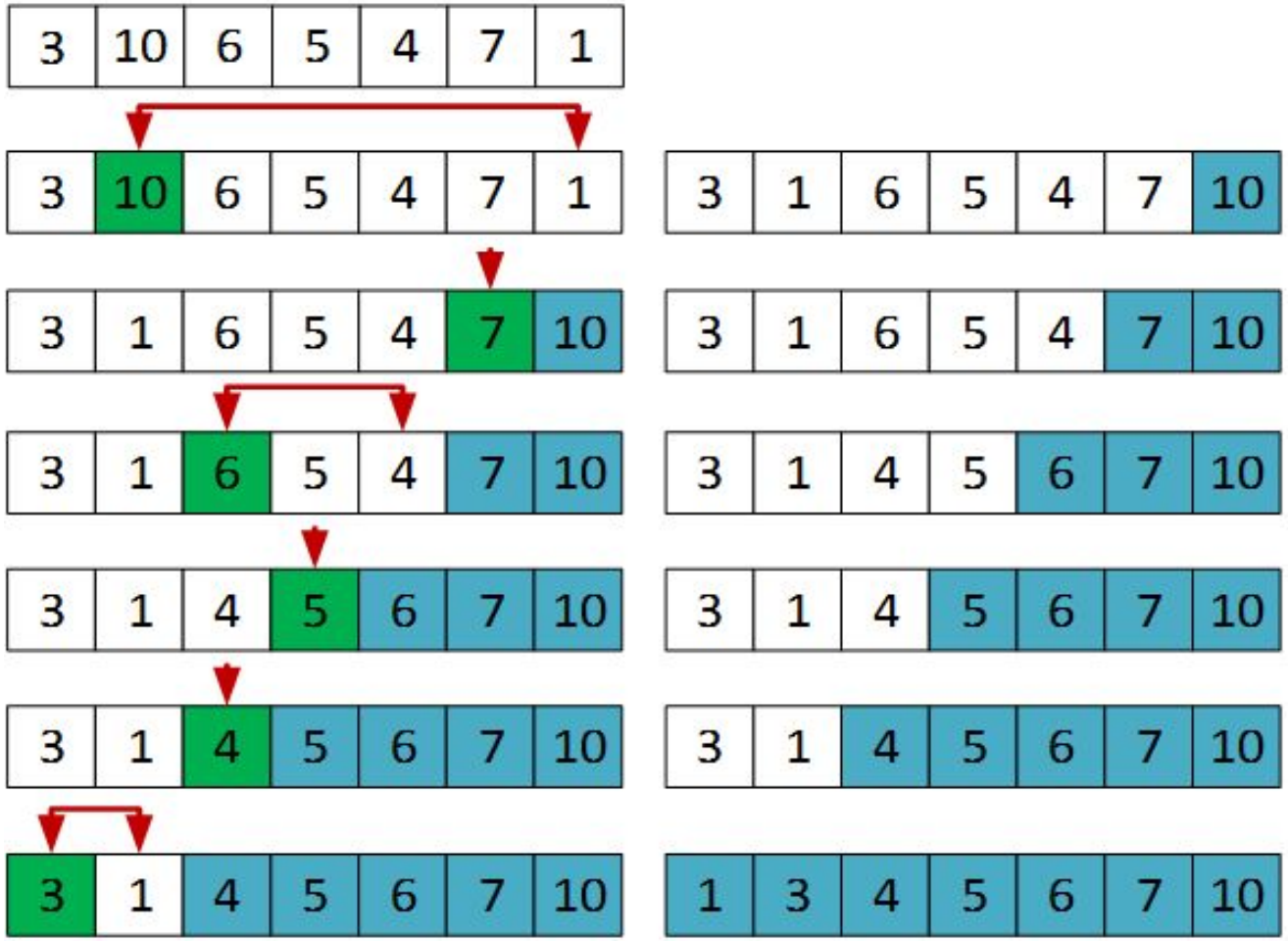
```
bool isSorted = false;
for (int i = 1; !isSorted && i < n; i++){
    isSorted = true;
    for (int j = 0; j < n - i; j++)
        if (mas[j] > mas[j + 1]){
            swap(mas[j], mas[j + 1]);
            isSorted = false;
            }
    }
```

В лучшем случае, когда массив полностью отсортирован, выполняется один проход и алгоритм остановится. Временная сложность O(n).

В худшем случае, когда массив будет отсортирован в обратном порядке, выполняется порядка n проходов. Временная сложность O(n^2).

Сортировка выбором

На каждой итерации во всей последовательности неотсортированных данных выбирается максимальный элемент и обменивается значениями с последним неотсортированным элементом.



Сортировка выбором

```
for (int j = 1; j < n; j++){
    int iMax = 0;
    for (int i = 1; i < n - j + 1; i++)
        if (mas[i] > mas[iMax])
            iMax = i;
    if (iMax != n - j)
        swap(mas[n - j], mas[iMax]);
    }
```

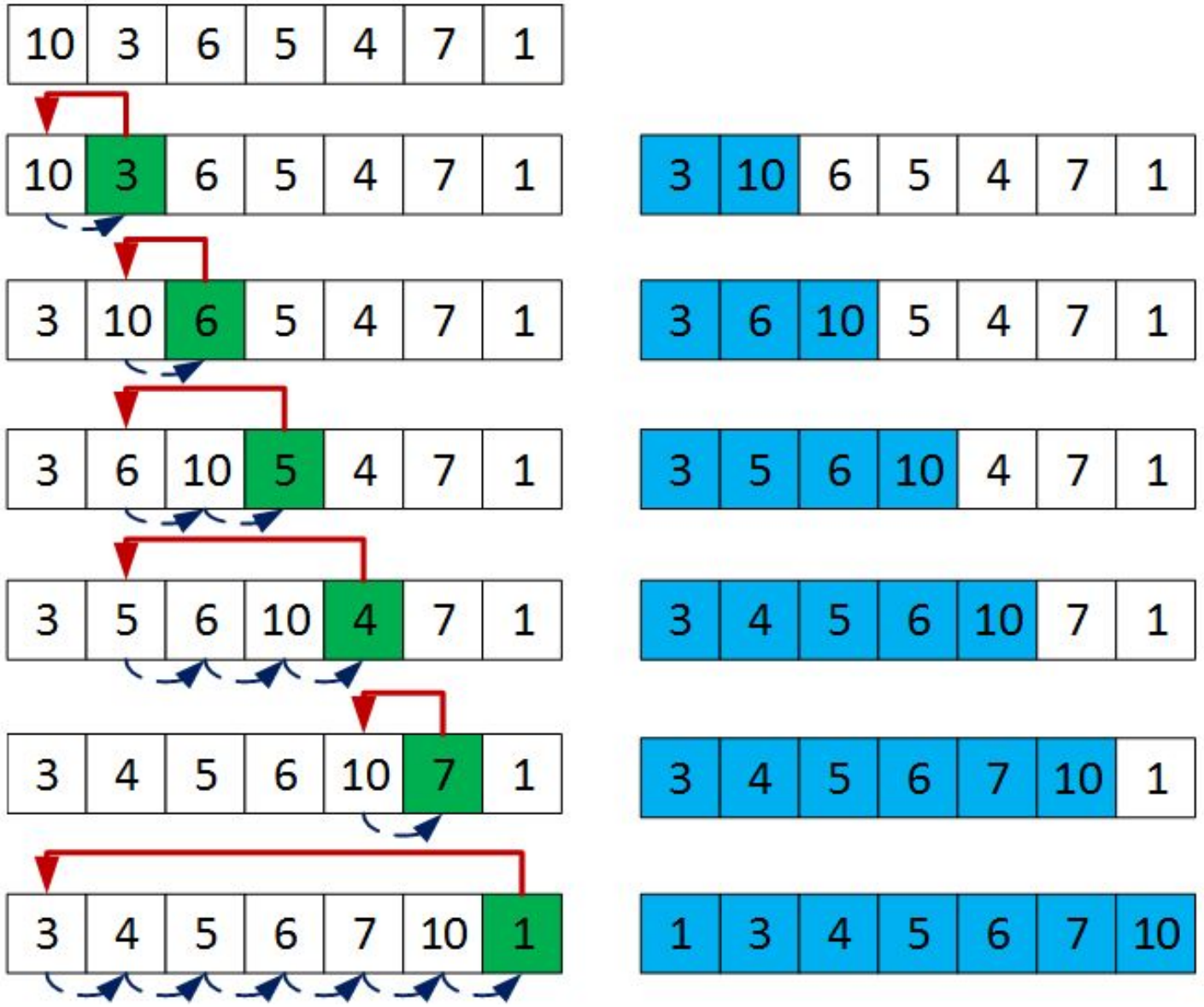
Временная сложность

T(n) = O((n-1)+(n-2)+...+1) = O((n-1)*n/2) = O(n^2) в худшем и лучшем случаях.

Сортировка вставками

- Принимается что для элемента mas_i все значения, которые стоят до позиции *i* являются упорядоченными.

Происходит сдвиг вправо элементов, стоящих перед *i* до тех пор, пока не найдутся такая позиция, в которую можно вставить mas_i с сохранением упорядоченности.



Сортировка вставками

```
for(int i = 1, j = i; i < n; i++){
    int temp = mas[i];
    for(j = i; j > 0 && mas[j - 1] > temp; j--)
       mas[j] = mas[j - 1];
     mas[j] = temp;
    }
```

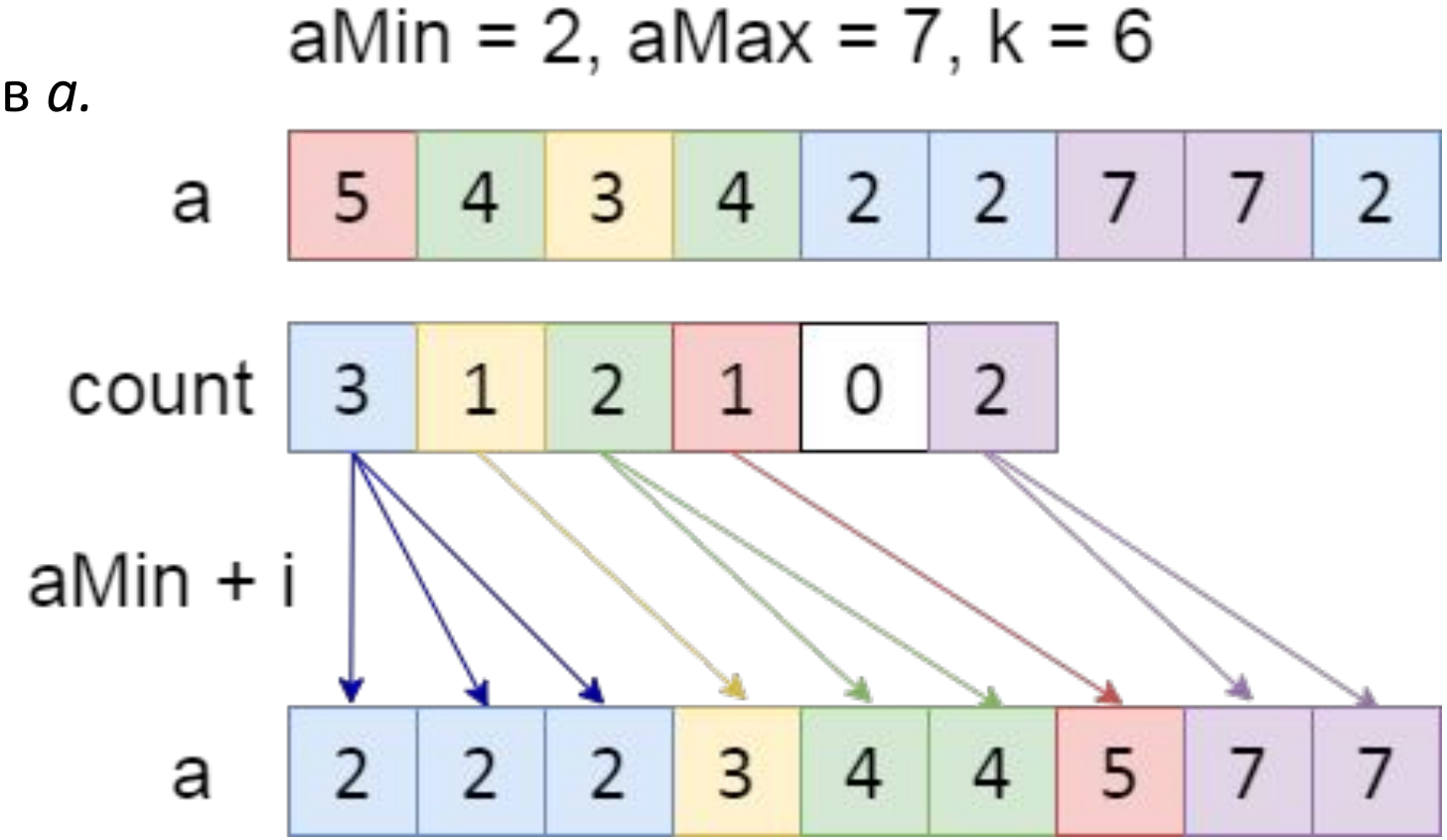
В лучшем случае, когда массив полностью отсортирован, выполняется один проход и алгоритм остановится. Временная сложность O(n).
В худшем случае, когда массив будет отсортирован в обратном порядке, выполняется порядка n проходов. Временная сложность O(n^2).

Сортировка подсчетом

Устойчива, сложность O(n + k), требуется O(k) дополнительной памяти для массива счета, k = max(a) – min(a) + 1.

Алгоритм применятся, когда k сопоставимо с размером массива.

1. Считаем сколько раз в массиве встречается каждое значение и записываем в массив счета.
2. Перезаписываем массив *a.*



Сортировка подсчетом

```
int aMax = a[0], aMin = a[0];
for(int i = 1; i < n; i++){
   if(a[i] > aMax) aMax = a[i];
   if(a[i] < aMin) aMin = a[i];
  }
int k = aMax - aMin + 1;
int count[k]{};
for(int i = 0; i < n; i++)
   count[a[i] - aMin]++;
int indA = 0;
for(int i = 0; i < k; i++){
   int iCount = count[i];
   for(int j = 0; j < iCount; j++)
      a[indA++] = aMin + i;
   }
```

Текущий контроль

В отсортированном по возрастанию массиве взяли последний (наибольший) элемент и поставили в начало массива. Мы хотим отсортировать полученный массив одним из изученных алгоритмов (оптимальные версии алгоритмов). Выберите все верные утверждения:

а) число итераций по массиву при сортировке пузырьком равно 1;

б) число итераций по массиву при сортировке пузырьком равно 2;

в) число итераций по массиву при сортировке пузырьком равно n – 1;

г) число итераций по массиву при сортировке выбором равно 1;

д) число итераций по массиву при сортировке выбором равно 2;

е) число итераций по массиву при сортировке выбором равно n – 1;

ж) число итераций по массиву при сортировке вставками равно 1;

з) число итераций по массиву при сортировке вставками равно 2;

и) число итераций по массиву при сортировке вставками равно n – 1.

Текущий контроль

В отсортированном по возрастанию массиве взяли последний (наибольший) элемент и поставили в начало массива. Мы хотим отсортировать полученный массив одним из изученных алгоритмов (оптимальные версии алгоритмов). Выберите все верные утверждения:

а) число итераций по массиву при сортировке пузырьком равно 1;

б) число итераций по массиву при сортировке пузырьком равно 2;

в) число итераций по массиву при сортировке пузырьком равно n – 1;

г) число итераций по массиву при сортировке выбором равно 1;

д) число итераций по массиву при сортировке выбором равно 2;

е) число итераций по массиву при сортировке выбором равно n – 1;

ж) число итераций по массиву при сортировке вставками равно 1;

з) число итераций по массиву при сортировке вставками равно 2;

и) число итераций по массиву при сортировке вставками равно n – 1.

Текущий контроль

Пусть дан массив, упорядоченный по убыванию. Мы хотим отсортировать его по возрастанию. Выберите все верные утверждения.

а) число итераций по массиву при сортировке пузырьком равно 1;

б) число итераций по массиву при сортировке пузырьком равно n – 1;

в) число итераций по массиву при сортировке выбором равно 1;

г) число итераций по массиву при сортировке выбором равно n – 1;

д) число итераций по массиву при сортировке вставками равно 1;

е) число итераций по массиву при сортировке вставками равно n – 1.

Текущий контроль

Пусть дан массив, упорядоченный по убыванию. Мы хотим отсортировать его по возрастанию. Выберите все верные утверждения.

а) число итераций по массиву при сортировке пузырьком равно 1;

б) число итераций по массиву при сортировке пузырьком равно n – 1;

в) число итераций по массиву при сортировке выбором равно 1;

г) число итераций по массиву при сортировке выбором равно n – 1;

д) число итераций по массиву при сортировке вставками равно 1;

е) число итераций по массиву при сортировке вставками **равно n – 1.**

Текущий контроль

Для каких массивов целесообразно применить сортировку подсчетом

а) {1, 2, 4, 3, 6, 5, 7, 8, 9, 10};

б) {-1, 3, 2, -1, 2, 2, 3, -1, -1, 2};

в) {100000, 1000, 1000000, 1000, 2000, 2000, 2000, 1000, 1000, 10000};

г) {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘l’, ‘o’, ’l’, ‘h’, ‘o’};

д) {0.5, 1, 0.5, 0,025, 0.5, 1, 1, 0,025, 1, 1};

е) {101, 105, 104, 105, 107, 100, 107, 101, 104, 105}.

Текущий контроль

Для каких массивов целесообразно применить сортировку подсчетом

а) {1, 2, 4, 3, 6, 5, 7, 8, 9, 10};

б) {-1, 3, 2, -1, 2, 2, 3, -1, -1, 2};

в) {100000, 1000, 1000000, 1000, 2000, 2000, 2000, 1000, 1000, 10000};

г) {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘l’, ‘o’, ’l’, ‘h’, ‘o’};

д) {0.5, 1, 0.5, 0,025, 0.5, 1, 1, 0,025, 1, 1};

е) {101, 105, 104, 105, 107, 100, 107, 101, 104, 105}.

Задача поиска элемента в массиве

Предположим, имеется некоторый набор данных, например, список оценок в зачетке по экзаменам за последние две сессии:

4, 4, 5, 5, 5, 3, 4, 4, 5, 5.

Для начисления повышенной стипендии необходимо выяснить, есть ли среди этих оценок хотя бы одна оценка «удовлетворительно».

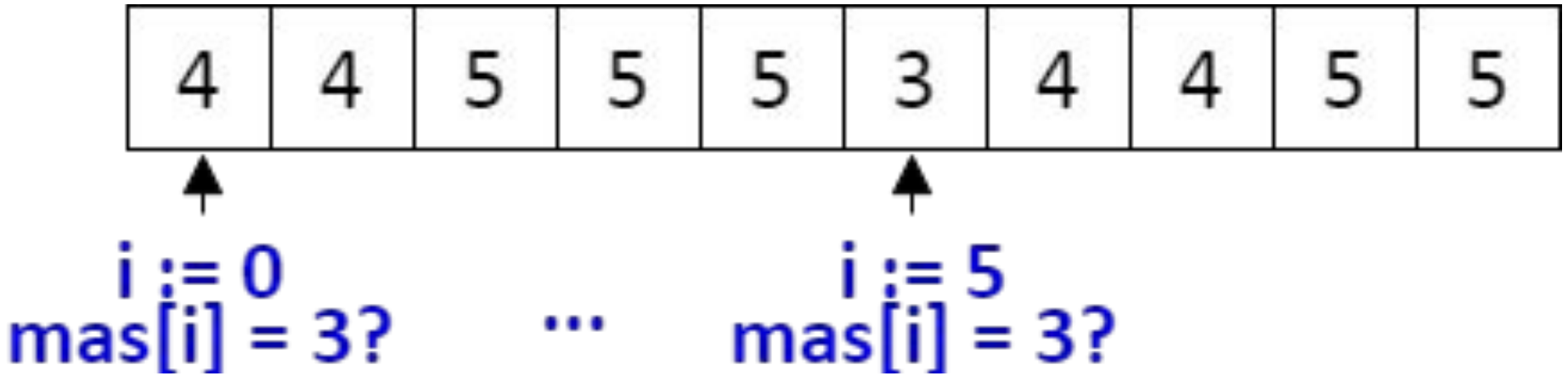
Базовая формулировка задачи поиска элемента в массиве:

осуществить поиск заданного элемента key в массиве длины n. В случае присутствия элемента key вывести его индекс в массиве, в противном случае вывести соответствующее сообщение.

Линейный поиск

В повседневной жизни мы часто неявно пользуемся самым простым способом поиска - линейным поиском.

Начиная с первого по счету элемента массива, поочередно сравниваем текущий элемент массива с нужным нам элементом key, пока не найдем его.



Реализация алгоритма линейного поиска

```
int key = 3;
for (int i = 0; i < n; i++)
   if(mas[i] == key){
      cout << i + 1 ;
      break;
      }
```

Преимущества линейного поиска:

- прост в понимании и реализации;
- способен работы на неотсортированных массивах.

Недостатки линейного поиска:

- не самый эффективный - имеет временную сложность O(n).

Задача поиска позиции вставки. Линейный поиск

Дан массив целых чисел, упорядоченных по возрастанию и целое число *key*. Известно, что в массиве нет элементов, равных по значению *key*. Найдите позицию (индекс), на которую нужно вставить элемент *key* (подвинув элементы, стоящие справа), чтобы сохранить упорядоченность массива.

```
int i = 0;
while (i < n && mas[i] < key)
    i++;
cout << i;
```

Существует ли алгоритм с меньшей временной сложностью?

Да, если данные упорядочены.

Задача. Ваш друг загадал число от 1 до 100. Необходимо, задавая вопрос «Загаданное число больше key?» и получая на него ответы Да или Нет, угадать это число.

Получая ответ на вопрос, мы сразу можем отсечь числа из диапазона от 1 до *key* в случае ответа Да и числа из диапазона от *key + 1* до 100 в противном случае.

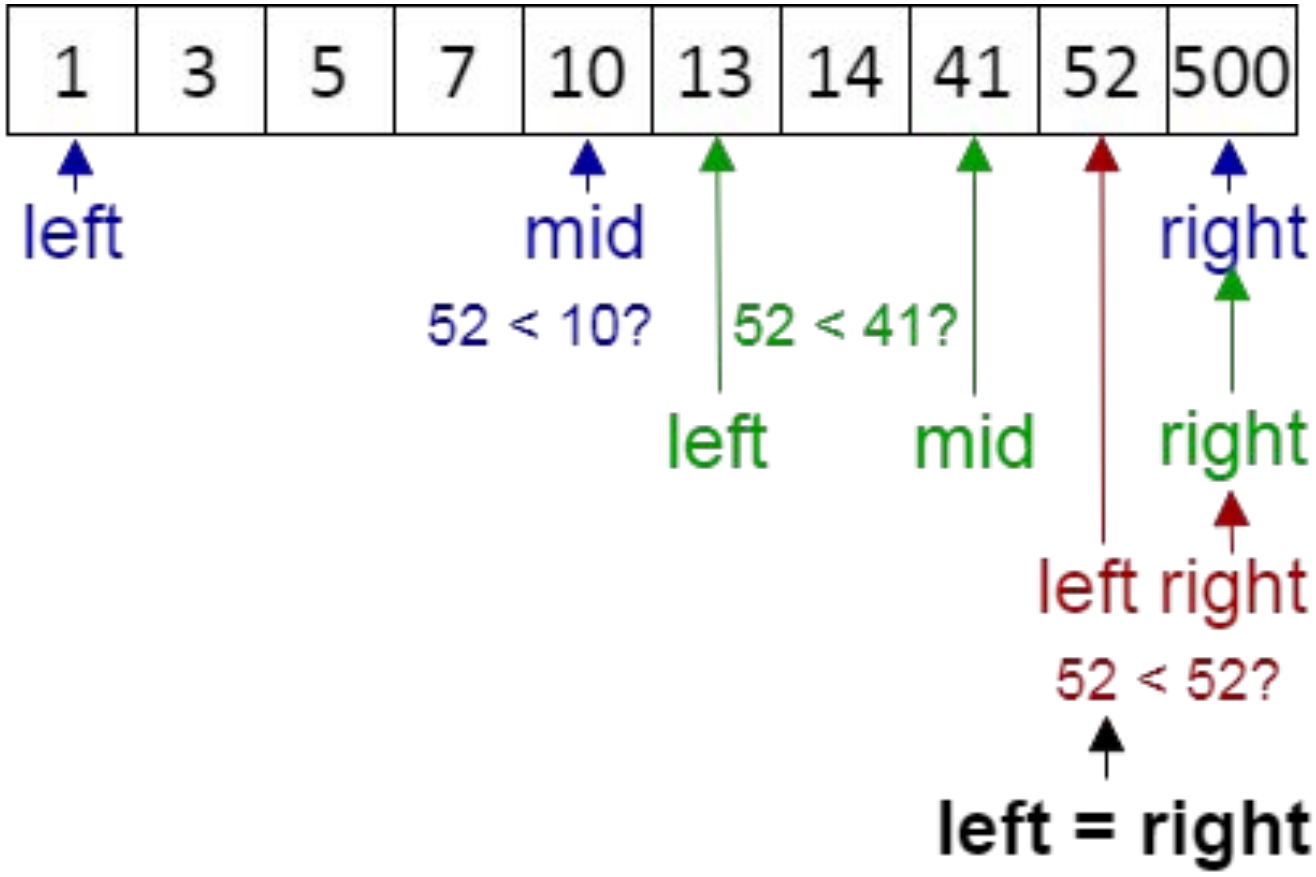
Рекуррентно отсекая от массива части, мы получим массив длины 1 — это и будет загаданное число.

Идея **бинарного поиска** состоит в подборе такого *key*, чтобы каждый раз отсекалась половина оставшегося массива.

Бинарный поиск

Бинарный поиск использует дробление массива на половины для каждой новой итерации. Изначально искомое значение сравнивается со средним элементом в массиве. Если значения не равны, то он отбрасывает ту часть массива, в которой целевое значение гарантированно не может находиться.

Временная сложность O(logN)



Бинарный поиск

```
int left = 0, right = n - 1, mid = 0;
while (left < right){
   mid = left + (right - left) / 2;
   if (mas[mid] < key)
      left = mid + 1;
   else
      right = mid;
   }
if (mas[right] != key)
    cout << -1;
else
   cout << right;
```

Задача поиска позиции вставки. Бинарный поиск

Дан массив целых чисел, упорядоченных по возрастанию и целое число *key.* Известно, что в массиве нет элементов, равных по значению *key*. Найдите позицию (индекс), на которую нужно вставить элемент *key* (подвинув элементы, стоящие справа), чтобы сохранить упорядоченность массива.

```
int left = 0, right = n, mid = 0;
while (left < right){
    mid = left + (right - left) / 2;
   if (mas[mid] < key)
      left = mid + 1;
    else
      right = mid;
   }
if (key < mas[left])
   cout << left;
else
   cout << right;
```

Единственное пропущенное число. Задача

Дан упорядоченный целочисленный массив длины N, содержащий уникальные значения в диапазоне от 0 до N включительно. Найдите единственное пропущенное число из диапазона 0-N.

```
int left = 0, right = n, mid = 0;
while (left < right){
   mid = left + (right - left) / 2;
   if (mas[mid] == mid)
      left = mid + 1;
   else
      right = mid;
   }
cout << left;
```

Левый и правый бинарный поиск

Если отсортированный массив имеет повторяющиеся значения, то может возникнуть необходимость найти минимальный и/или максимальный индекс вхождения key.

Например:

mas = {1, 1, 3, 3, 3, 3, 3, 3, 6, 6}, key = 3

mid = 0 + (9 - 0) / 2 = 4

Необходимо изменить алгоритм так, чтобы он не “отрезал” куски массива, которые содержат key.

Для минимального индекса key запустим левый бинарный поиск.

Для максимального индекса key запустим правый бинарный поиск.

Левый бинарный поиск

```
int left = 0, right = n - 1;
while (left + 1 < right){ //пока не соседние элементы!
    int mid = left + (right - left) / 2;
    if (mas[mid] < key)
        left = mid; //не отсекаем mid!
    else
            right = mid; /*может быть равно key, но нас
интересует только левая граница*/
    }
int leftKey = -1;
if (mas[left] == key)
    leftKey = left;
else
    if(mas[right] == key)
        leftKey = right;
```

Правый бинарный поиск

```
int left = 0, right = n - 1;
while (left + 1 < right){
   int mid = left + (right - left) / 2;
   if (mas[mid] > key)
      right = mid;
   else
      left = mid;
   }
int rightKey = -1;
if (mas[right] == key)
   rightKey = right;
else
   if(mas[left] == key)
      rightKey = left;
```

Текущий контроль

Выберите верные утверждения:

а) линейный поиск числа 8 в списке [1, 4, 2, 9, 8, 17, -5] выведет число 4;

б) если искомого элемента в списке нет, линейный поиск с циклом for не запустится;

в) линейный поиск возвращает индекс первого (если считать слева) вхождения элемента x в список;

г) линейный поиск работает только на отсортированных массивах.

Текущий контроль

Выберите верные утверждения:

а) линейный поиск числа 8 в списке [1, 4, 2, 9, 8, 17, -5] выведет число 4;

б) если искомого элемента в списке нет, линейный поиск с циклом for не запустится;

в) линейный поиск возвращает индекс первого (если считать слева) вхождения элемента x в список;

г) линейный поиск работает только на отсортированных массивах.

Текущий контроль

Выберите верные утверждения:

а) линейный поиск элемента x = 3 в списке [1, 5, 4, 12, 3, 6] с циклом for произведет четыре операции сравнения на равенство (операция ==);

б) число операций сравнения на равенство при линейном поиске циклом for не может быть больше длины массива;

в) число операций сравнения на равенство при линейном поиске циклом for не может быть меньше одного;

г) число операций сравнения на равенство при линейном поиске циклом for не может быть больше n-1, где n — длина массива.

Текущий контроль

Выберите верные утверждения:

а) линейный поиск элемента x = 3 в списке [1, 5, 4, 12, 3, 6] с циклом for произведет четыре операции сравнения на равенство (операция ==);

б) число операций сравнения на равенство при линейном поиске циклом for не может быть больше длины массива;

в) число операций сравнения на равенство при линейном поиске циклом for не может быть меньше одного;

г) число операций сравнения на равенство при линейном поиске циклом for не может быть больше n-1, где n — длина массива.

Текущий контроль

Пусть мы запустили левый и правый бинарные поиски элемента key на заданном массиве mas размера n. Они вернули индексы leftKey и rightKey. Чему равно количество элементов в массиве mas, не равных key?

а) rightKey – leftKey;

б) leftKey;

в) n - rightKey – 1;

г) n - rightKey - 1 + leftKey.

Текущий контроль

Пусть мы запустили левый и правый бинарные поиски элемента key на заданном массиве mas размера n. Они вернули индексы leftKey и rightKey. Чему равно количество элементов в массиве mas, не равных key?

а) rightKey – leftKey;

б) leftKey;

в) n - rightKey – 1;

г) n - rightKey - 1 + leftKey.

Текущий контроль

Необходимо совершить M поисков элементов в неотсортированном массиве длины N. Какой будет временная сложность программы?

а) O(N*log(M));

б) O(M*N);

в) O(M + N);

г) O(M*log(N)).

Текущий контроль

Необходимо совершить M поисков элементов в неотсортированном массиве длины N. Какой будет временная сложность программы?

а) O(N*log(M));

б) O(M*N);

в) O(M + N);

г) O(M*log(N)).

Текущий контроль

Нам нужно совершить M поисков элементов в отсортированном массиве длины N. Какой будет минимальная временная сложность программы?

а) O(N*log(M));

б) O(log(N));

в) O(M*log(N));

г) O(M + log(N)).

Текущий контроль

Нам нужно совершить M поисков элементов в отсортированном массиве длины N. Какой будет минимальная временная сложность программы?

а) O(N*log(M));

б) O(log(N));

в) O(M*log(N));

г) O(M + log(N)).

Текущий контроль

Какой из поисков может работать с отсортированным массивом?

а) линейный;

б) бинарный;

в) ни один из них;

г) оба.

Текущий контроль

Какой из поисков может работать с отсортированным массивом?

а) линейный;

б) бинарный;

в) ни один из них;

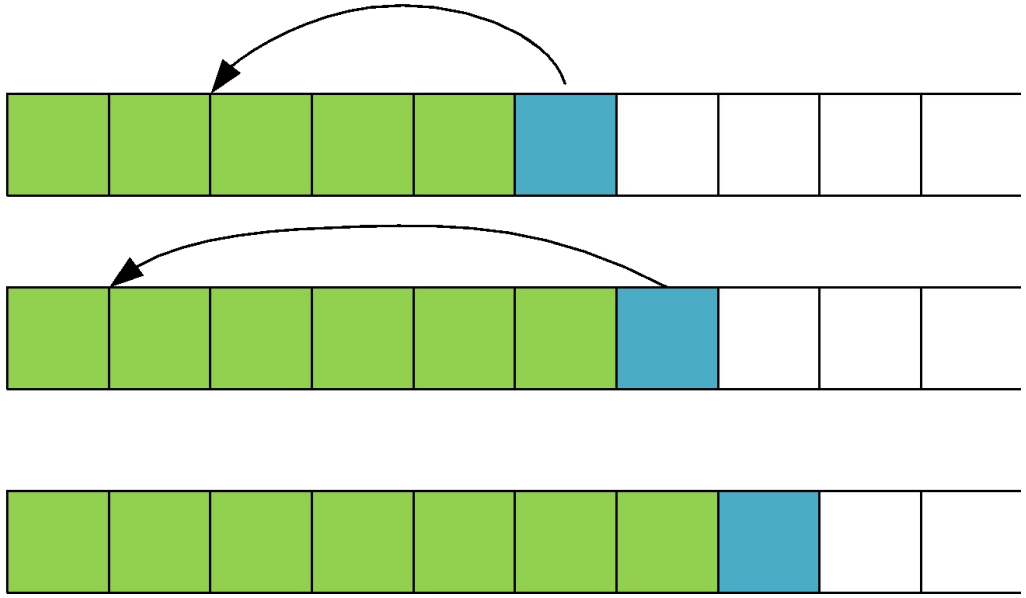
г) оба.

Сортировка бинарными вставками

Устойчивая, Сложность O(n^2)

Метод учитывает, что готовая последовательность, в которую нужно включить новый элемент уже упорядочена.

Место включения элемента можно найти значительно быстрее, применив бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения.



Сортировка бинарными вставками. Реализация

```
for (int i = 1; i < n; i++){
   if (mas[i - 1] > mas[i]) {
      int cur = mas[i];
      int left = 0, right = i - 1;
      do {
         int mid = left + (right - left) / 2;
         if (mas[mid] < cur)
            left = mid + 1;
         else
            right = mid - 1;
         }
      while (left <= right);
      for (int j = i - 1; j >= left; j--)
         mas[j + 1] = mas[j];
      mas[left] = cur;
      }
   }
```

Одномерное динамическое программирование

Метод динамического программирования – это подход к решению сложных задач путем разбиения их на более мелкие части и запоминания результатов решения этих частей.

1 + 2 = ?

1 + 2 + 3 = ?

1 + 2 + 3 + 4 = ?

…

Числа Фибоначчи

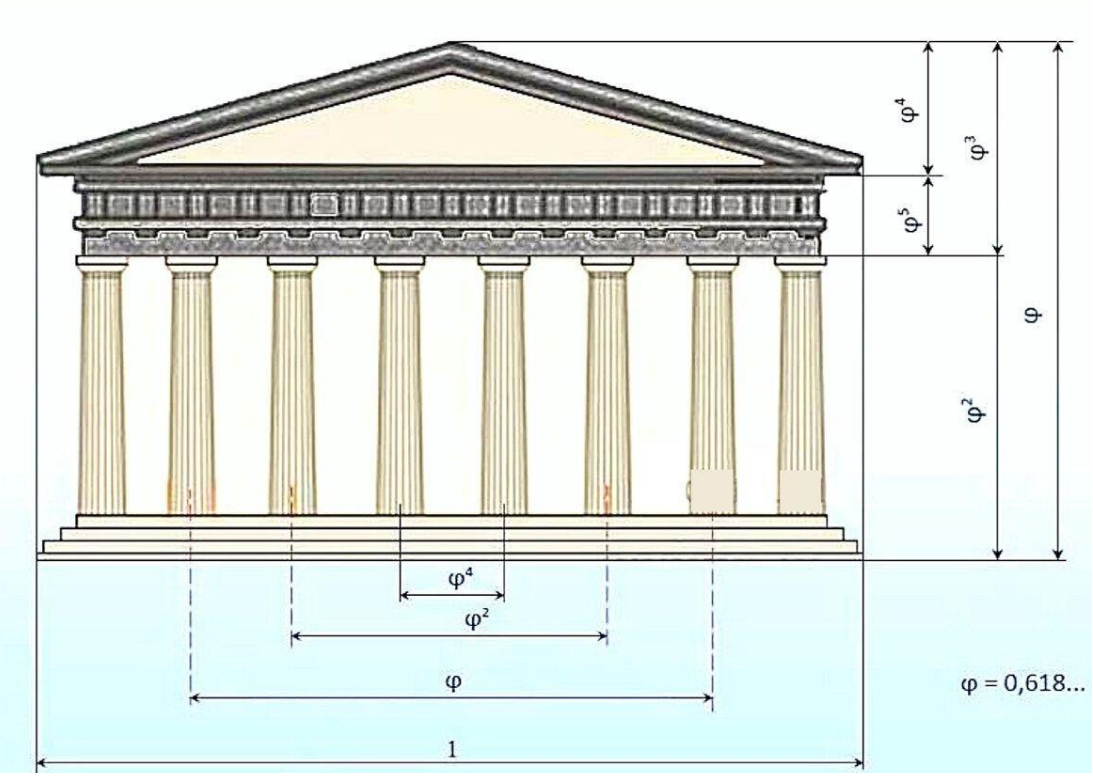
Числа Фибоначчи - числовая последовательность, которая задается следующим образом: первые два элемента: 0 и 1, а далее каждый элемент является суммой двух предыдущих.

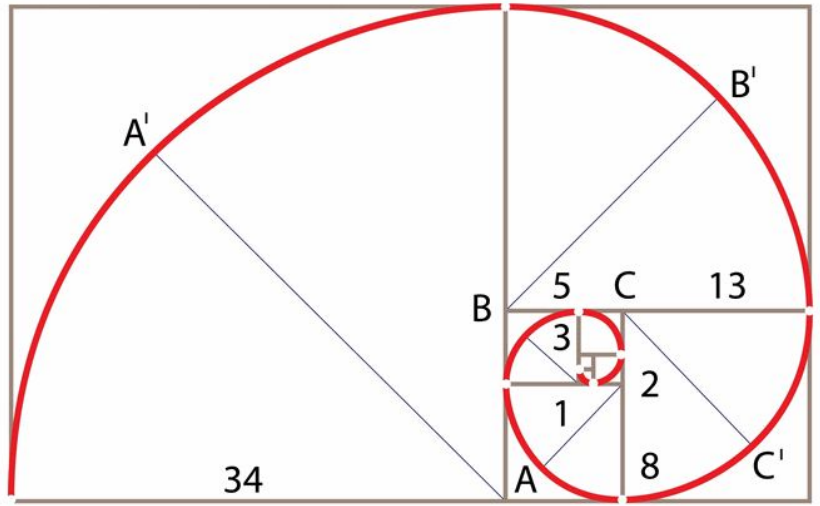
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

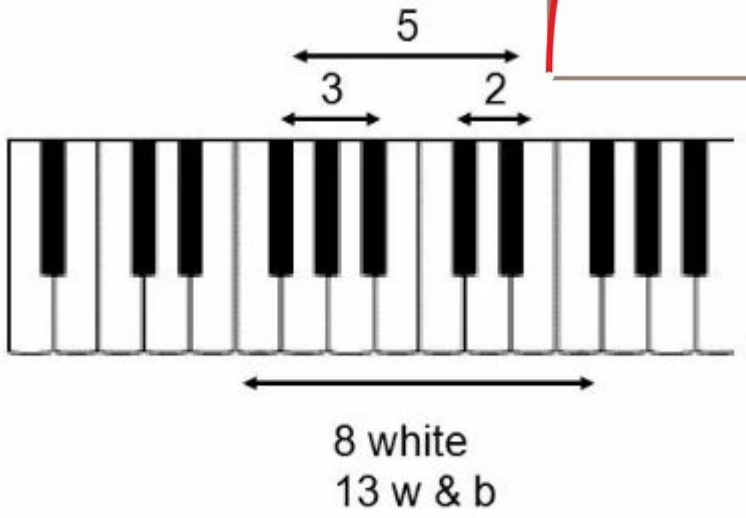
«Человек посадил пару кроликов в загон, окруженный со всех сторон стеной. Сколько пар кроликов за год может произвести на свет эта пара, если известно, что каждый месяц, начиная со второго, каждая пара кроликов производит на свет одну пару?»

Числа Фибоначчи в архитектуре, музыке, поэзии

А.С. Пушкина с этой точки зрения показал, что размеры стихов распределены весьма неравномерно; оказалось, что Пушкин явно предпочитает размеры в 5, 8, 13, 21 и 34 строк (числа Фибоначчи).







Вычисление последовательности Фибоначчи с помощью динамического программирования

Реализуйте код, который делает m запросов чисел Фибоначчи по введенному n (для n < 100).

```
const unsigned int n = 101;
unsigned fib[n];
fib[0] = 0,
fib[1] = 1;
for(unsigned i = 2; i < n; i++)
   fib[i] = fib[i - 1] + fib[i - 2];
int m, nFib;
cin >> m;
while (m--){
   cin >> nFib;
   cout << fib[nFib] << '\n';
   }
```

Вычисление префиксных сумм

Дан массив размера n. Необходимо посчитать сумму чисел в массиве, предшествующие i-му элементу.

Обозначим за f(i) значение суммы на префиксе.

f(0) = mas[0].

Для вычисления последующих значений f(i) потребуется сумма f(i−1) на префиксе длины i−1. То есть:

f(i) = f(i − 1) + mas[i], при i ≥ 1

```
int pref[n]{mas[0]};
for(int i = 1; i < n; i++)
   pref[i] = pref [i - 1] + mas[i];
```

Задача о кузнечике

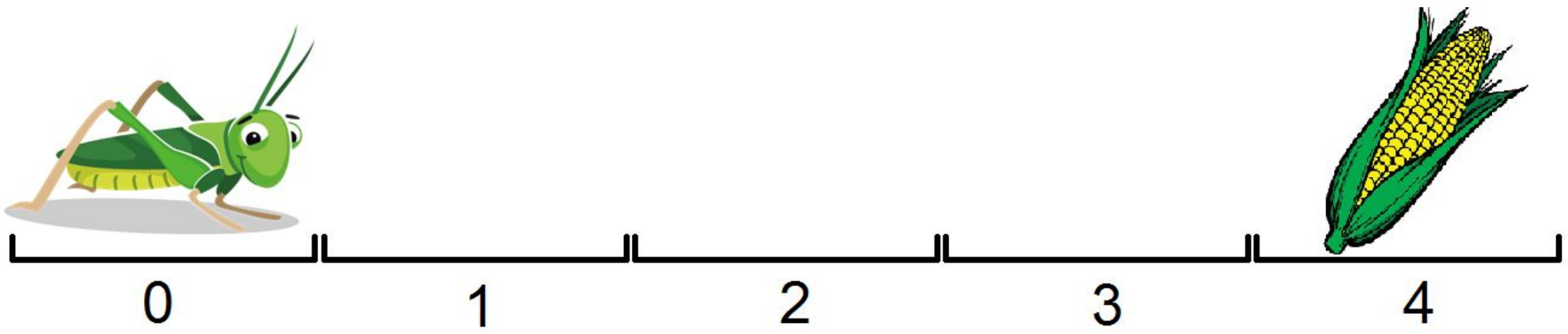
На числовой прямой сидит кузнечик, который может прыгать вправо на одну или две единицы. Первоначально кузнечик находится в точке с координатой 0. Сколько существует способов для кузнечика добраться до координаты 4?

0->1->2->3->4 0->1->2->40->1->3->40->2->3->40->2->4

Задача о кузнечике

1. Изначально кузнечик находится в нулевой координате. Сколько у него способов добраться до координаты 1? **1 способ.**
2. Сколько у него способов добраться до координаты 2? В координату 2 он может попасть как из 1, так и из 0. То есть **2 способа**.
3. В координату 3 он может попасть из 1 и 2. Но мы помним, что в координату 2 можно попасть двумя способами. В итоге, получаем 1 + 2 = **3 способа**.

То есть число способов попасть кузнечику в конкретную точку — это сумма способов попасть в координату, меньшую на 1, и в координату, меньшую на 2.



Задача о кузнечике

На числовой прямой сидит кузнечик, который может прыгать вправо на одну или две единицы. Первоначально кузнечик находится в точке с координатой 0. Сколько существует способов для кузнечика добраться до координаты n?

```
int k[n + 1]{};
k[0] = k[1] = 1;
for (int i = 2; i < n + 1; i++)
   k[i] = k[i - 1] + k[i - 2];
cout << k[n];
```

Не напоминает числа Фибоначчи?

Сложность такого алгоритма O(n), поскольку он сводит решение задачи к заполнению одномерного массива длины n+1.

Задача о кузнечике 2

Пусть теперь кузнечик способен прыгать вправо на расстояние 1, 2 и 3. Мы хотим определить количество различных маршрутов кузнечика, приводящих его в определенную точку на прямой.

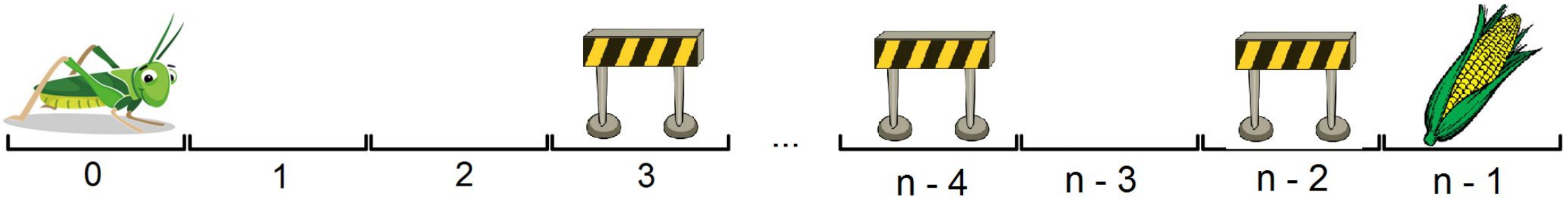
Теперь в конкретную точку i можно попасть из точек с координатами i - 1, i - 2, i - 3.

```
int k[n + 1]{};
k[0] = k[1] = 1;
k[2] = k[0] + k[1];
for (int i = 3; i < n + 1; i++)
   k[i] = k[i - 1] + k[i - 2] + k[i – 3];
cout << k[n];
```

Задача о кузнечике с препятствиями

На числовой прямой сидит кузнечик, который может прыгать вправо на одну или две единицы. Первоначально кузнечик находится в точке с координатой 0. Некоторые из клеток помечены как занятые и на них прыгать нельзя (массив obstacle типа bool). Сколько существует способов для кузнечика добраться до координаты n?

```
bool obstacle[n] = {/**/};
int k[n]{obstacle[0], obstacle[1] + obstacle[0]};
for (int i = 2; i < n; i++)
   k[i] = obstacle[i - 1] * k[i - 1]
        + obstacle[i - 2] * k[i - 2];
cout << k[n - 1];
```



Минимальная стоимость пути кузнечика

Введем стоимости для каждого прыжка.

Пусть стоимости будут различны для каждой точки и хранятся в массиве cost, таким образом стоимость прыжка в точку i определяется значением cost[i]. Найдите минимальную стоимость, за которую кузнечик может добраться из точки 0 до n, где n — длина массива cost.

```
int k[n]{};
k[0] = cost[0];
k[1] = cost[1];
for (int i = 2; i < n; i++)
  k[i] = min(k[i - 1], k[i - 2]) + cost[i];
cout << k[n - 1];
```

Текущий контроль

Какие из кузнечиков, приведенных ниже, способны добраться до точки с координатой 13?

а) кузнечик, прыгающий на 3 и 4;

б) кузнечик, прыгающий на 3 и 6;

в) кузнечик, прыгающий на 4;

г) кузнечик, прыгающий на 1 и 5.

Текущий контроль

Какие из кузнечиков, приведенных ниже, способны добраться до точки с координатой 13?

а) кузнечик, прыгающий на 3 и 4;

б) кузнечик, прыгающий на 3 и 6;

в) кузнечик, прыгающий на 4;

г) кузнечик, прыгающий на 1 и 5.