В чем заключается бинарный поиск?

Бинарный поиск – алгоритм поиска в упорядоченном массиве. Он базируется на идее постепенного уменьшения области поиска элемента вдвое; если элемента нет в списке, то длина области поиска постепенно уменьшается до 0, и в этот момент выполнение алгоритма завершается. если это произошло, можно сказать, что в массиве нет искомого элемента, иначе же искомый элемент будет найден в процессе разделения области поиска пополам. Опишем этот алгоритм подробнее:

1) Устанавливаются индексы границ области поиска: изначально индекс левой границы равен 0, правой – индексу последнего элемента массива,
2) Вычисляется индекс середины массива – результат целочисленного деления на 2 суммы индексов левой и правой границы
3) Если искомый элемент равен элементу под индексом середины массива, то элемент найден; индекс возвращается алгоритмом и программа завершает исполнение
4) Если искомый элемент больше элемента под индексом середины массива, то индекс левой границы становится равен индексу середины массива, увеличенному на 1 (то есть левая граница сдвигается на 1 элемент правее серединного элемента)
5) Если искомый элемент меньше элемента под индексом середины массива, то индекс правой границы становится равен индексу середины массива, уменьшенному на 1 (то есть левая граница сдвигается на 1 элемент правее серединного элемента)
6) Если индекс левой границы стал больше индекса правой, то искомого элемента в массиве нет: возвращается число -1 и программа завершает исполнение. Иначе повторяются шаги 2-5.

Определите индексы элементов массива, бинарный поиск которых наиболее продолжителен.

Для примера рассмотрим массив из трех элементов, {1, 2, 3}. Индексы левой и правой границ отметим как l и r, индекс середины как m. Изначально l = 0, r = 2, m = (0+2) div 2 = 1. Если искомый элемент 2, то он уже на первой итерации будет найден. Иначе границы поиска изменятся так, что либо l = 0, r = m – 1 = 0, либо l = m+1 = 2; r = 2. Тогда либо m = (0+0) div 2 = 0, либо m = (2+2) div 2 = 2, то есть уже на второй итерации могут быть найдены элементы 0 и 2.

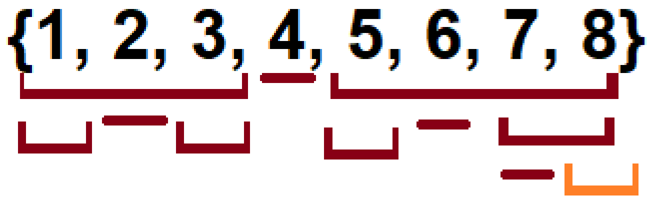
Рассмотрим более объемный пример, массив из семи элементов, {1, 2, 3, 4, 5, 6, 7}. Изначально l = 0, r = 6, m = (0+6) div 2 = 3. Элемент 4 (на 3-ей позиции) будет найден на первой итерации, далее возможно 2 варианта:

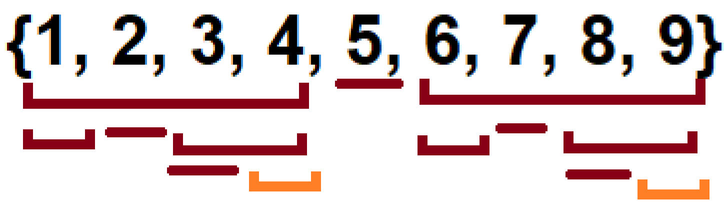
- l = 0; r = m-1 = 2; m = (0+2) div 2 = 1. Тогда элемент 2 (на 1-ой позиции) будет найден на второй итерации; иначе также возможны 2 варианта:
  - l = 0; r = m-1 = 0; m = (0+0) div 2 = 0. Тогда элемент 1 (на 0-ой позиции) будет найден на третьй итерации;
  - l = m+1 = 2; r = 2; m = (2+2) div 2 = 2. Тогда элемент 3 (на 2-ой позиции) будет найден на третьй итерации;

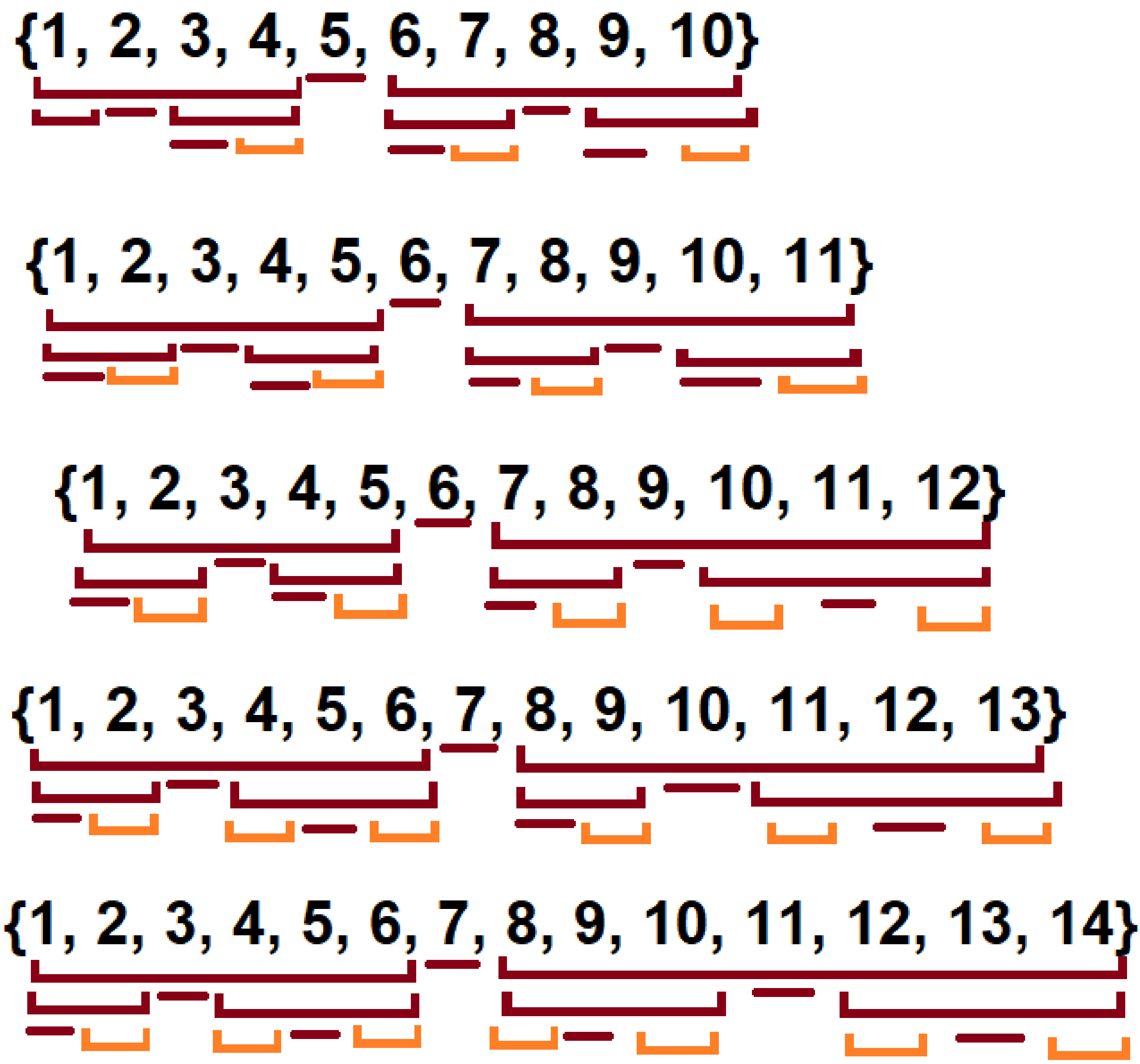
- l = m+1 = 4; r = 6; m = (4+6) div 2 = 5. Тогда элемент 6 (на 5-ой позиции) будет найден на второй итерации; иначе также возможны 2 варианта:
    - l = 4; r = m-1 = 4; m = (4+4) div 2 = 4. Тогда элемент 5 (на 4-ой позиции) будет найден на третьй итерации;
    - l = m+1 = 6; r = 6; m = (6+6) div 2 = 6. Тогда элемент 7 (на 6-ой позиции) будет найден на третьй итерации;

То есть можно увидеть закономерность: в тех случаях, когда массив на каждой итерации делится серединным элементом на 2 равных части, последними будут найдены элементы, начиная с 0-ого и идя по массиву дальше с шагом 2: 0-ой, 2-ой, 4-ый, 6-ой и так далее. В свою очередь, массив будет всегда делиться серединным элементом на равные части в том случае, если его длина равна некоторой степени двойки, уменьшенной на 1: 3, 7, 15 и так далее.

Что для случаев, когда длина массива не равна такому числу? Рассмотрим случаи с прибавлением к «эталонной» длине массива одного, двух, трех, четырех элементов. Подробное расписывание этих случаев будет слишком объемным, а потому обозначим графически. Серединный элемент на каждой итерации будет обозначать подчеркиванием, а подмассивы, на которые он разделяет исходный – горизонтальными квадратными скобками. Чем позднее происходит итерация, тем ниже нарисованы подчеркивания и скобки. Каждая итерация в какой-то момент закончится тем, что будет выделен подмассив из одного элемента: элемент в этом подмассиве на последней итерации и будет являться тем, чей поиск наиболее продолжителен. Выделим их цветом.







Можно заметить, что количество таких элементов возрастает каждый раз на единицу. Если «идеальное» количество элементов, не больше заданного размера n, можно вычислить как 2^floor($\log_2$(n+1)) – 1, то количество элементов с максимальной продолжительностью поиска равно k = 2 – 2^floor($\log_2$(n+1)) + 1, если не рассматривать «идеальный» случай выше. И в этом случае, увы, нельзя выделить единую последовательность с четким шагом: расстояние между такими элементами меняется и в пределах одного массива. Можно описать общий принцип: для n элементов массив делится на parts_amount = 2^level частей по тому же принципу, по которому делит массив алгоритм бинарного поиска; за level примем «глубину» спуска, количество раз, когда понадобится разбить подмассив надвое, level = ceil($\log_2$(n+1)). Если в получившейся части нет элементов вообще, то она опускается: если в части один элемент, то он и будет одним из элементов с максимальной продолжительностью поиска. Можно перебрать все получившиеся части, перебирая числовые значения от 0 до parts_amount и, работая с этими числами как с двоичными, находить нужное число алгоритмом, похожим на бинарный поиск: i-тый разряд, равный 1, свидетельствует, что на i-той итерации нужно сдвинуть левую границу области поиска, а i-тый разряд, равный 0 – о том, что нужно сдвинуть правую границу.

```c
#include <stdbool.h>
#include <stdio.h>

int logarithm (int base, int num, bool is_ceil) {
    int cur_num = 1;
    int cur_result = 0;
    while (cur_num < num) {
        cur_num *= base;
        cur_result++;
    }
    if (!is_ceil) {
        return cur_result - (cur_num > num);
    } else {
        return cur_result;
    }
}

int degree (int base, int d) {
    int result = 1;
    for (int i = 0; i < d; i++) {
        result *= base;
    }
    return result;
}

int get_cur_index(int size, int num, int level) {
    int left = 0;
    int right = size - 1;
    while (level > 0 && (right >= left)) {
        int middle = (left + right) / 2;
        if (num >> (level - 1) & 1) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
        level--;
    }
    return (right >= left) ? right : -1;
}

int main () {
    for (int n = 8; n <= 14; n++) {
        printf("n = %d;\t indexes: ", n);
```

```
        int level = logarithm(2, n+1, false);
        int parts_amount = degree(2, level);

        for (int cur = 0; cur < parts_amount; cur++) {
            int cur_index = get_cur_index(n, cur, level);
            if (cur_index != -1) {
                printf("%d, ", get_cur_index(n, cur, level));
            }
        }
        printf("\b\b \n");
    }
}
```

```
"C:\Users\sovac\Desktop\АСД\АСД 4 си\index_search.exe"
n = 8;   indexes: 7
n = 9;   indexes: 3, 8
n = 10;  indexes: 3, 6, 9
n = 11;  indexes: 1, 4, 7, 10
n = 12;  indexes: 1, 4, 7, 9, 11
n = 13;  indexes: 1, 3, 5, 8, 10, 12
n = 14;  indexes: 1, 3, 5, 7, 9, 11, 13

Process finished with exit code 0
```

Разработайте и реализуйте итеративный и рекурсивный алгоритмы бинарного поиска?

Итеративный алгоритм бинарный поиск реализуем так же, как в лабораторной работе, с небольшими изменениями (возврат индекса сразу после того, как он был найден, а не сохранение его в результирующую переменную + измеренные имена переменных). Функции передаются указатель на массив arra, его размер size и искомый элемент value. Вначале задаются переменные индексов левой и правой границ, left = 0 и right = size-1. Запускается цикл, продолжающий выполнение, пока область поиска не пуста, то есть left <= right. В теле цикла для left и right находится серединный индекс middle = (left + right) div 2. Если элемент под индексом middle, array[middle], равен value, функция возвращает middle; если value больше array[middle], значение left перезаписывается как middle + 1; если value меньше array[middle], значение rigth перезаписывается как middle – 1. Выход из цикла означает, что ни на родной из итераций серединный элемент не был равен value, и элемента value в массиве нет вообще – тогда функция возвращает -1.

```
int iterativeBinarySearch(int *array, int size, int value) {
    int left = 0;
    int right = size-1;
```

```
    while (left <= right) {
        int middle = (left + right)/2;
        if (array[middle] == value) {
            return middle;
        } else if (value > array[middle]) {
            left = middle + 1;
        } else {
            right = middle - 1;
        }
    }
    return -1;
}
```

При реализации рекурсивного алгоритма бинарного поиска используются примерно те же смысловые блоки, однако за них отвечают другие синтаксические конструкции. Переменные индексов левой и правой границ, left и right не создаются в начале функции, а передаются ей как аргументы, вместо размера массива; также передаются указатель на массив array и искомый элемент value. Условие, отвечающее за то, что область поиска не пуста, помещено не в цикл, а в условную конструкцию: если оказывается, что left > right, то функция возвращает -1 – знак того, что элемента нет в массиве. Иначе для переданных функции left и right находится серединный индекс middle (также путем деления суммы left и right на 2). Если элемент под индексом middle, array[middle], равен value, функция возвращает middle; если value больше array[middle], возвращается значение функции рекурсивного бинарного поиска для массива array, левой границы middle + 1, правой границы right и искомого элемента value; если value меньше array[middle], возвращается значение функции рекурсивного бинарного поиска для массива array, левой границы left, правой границы middle - 1 и искомого элемента value.

Также можно создать для рекурсивного алгоритма бинарного поиска функцию-обертку, чтобы привычно передавать ей размер массива size вместо индексов left и right. Тогда в теле функции-обертки эти индексы будут вычисляться как left=0; right = size-1, и с этими значениями будет вызываться основная функция рекурсивного алгоритма.

```
int recursiveBinarySearch_(int *array, int left, int right, int
value) {
    if (left > right) {
        return -1;
    } else {
        int middle = (left + right)/2;
        if (array[middle] == value) {
            return middle;
        } else if (value > array[middle]) {
            return recursiveBinarySearch_(array, middle + 1,
```

```
right, value);
        } else {
            return recursiveBinarySearch_(array, left, middle - 1,
value);
        }
    }
}
```

```
int recursiveBinarySearch(int *array, int size, int value) {
    return recursiveBinarySearch_(array, 0, size-1, value);
}
```

У тебя есть функция, в которую передаётся переменная _X, в функции есть одинарный цикл. Данная переменная перед заходом в данную функцию обрабатывается во вложенном цикле. Из вложенного цикла передается переменная в функцию и возвращается результат во вложенный цикл. Какова сложно алгоритма? Вопрос понятен ?

Если это значит, что алгоритм имеет вид

```
function (x) {
        for (<перебор диапазона, размер которого прямо пропорционален x>) {
                x = <какие-либо операции над x>;
        }
        return x
}

main () {
        _X = <произвольное значение>
        for (<перебор диапазона, размер которого прямо пропорционален _X>) {
                function(_X) //модификация таким образом переменной _X или
любых других переменных
        }
}
```

...То алгоритм будет иметь сложность O(_X^{2}).