Вопросы по первому модулю.

1. Что означают записи “f(n) = Θ(g(n))”, “f(n) = O(g(n))” и “f(n) = Ω(g(n))”?

Определение. Для функции g(n) записи Θ(g(n)), O(g(n)) и Ω(g(n)) означают следующие множества функций:

$\Theta(g(n)) = \{f(n)$: существуют положительные константы c_1, c_2 и n_0, такие что $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всех $n \ge n_0\}$,

$O(g(n)) = \{f(n)$: существуют положительные константы c и n_0, такие что $0 \le f(n) \le cg(n)$ для всех $n \ge n_0\}$,

$\Omega(g(n)) = \{f(n)$: существуют положительные константы c и n_0, такие что $0 \le cg(n) \le f(n)$ для всех $n \ge n_0\}$.

Обозначение. Вместо записи «$T(n) \in \Theta(g(n))$» часто используют запись «$T(n) = \Theta(g(n))$».

2. Чем плох рекурсивный алгоритм вычисления n-ого числа Фибоначчи?
   А) Скорость — золотое сечение в степени N (1.618…), потому что многие действия делаем 2 раза
   Б) Много памяти, т.к. на каждый рекурсивный вызов выделяется дополнительная память под стек функции

3. Опишите алгоритм проверки числа на простоту за O(sqrt(n))?

```
bool IsPrime( int n )
{
    if( n == 1 ) {
        return false;
    }
    for( int i = 2; i * i <= n; ++i ) {
        if( n % i == 0 ) {
            return false;
        }
    }
    return true;
}
```

4. Опишите алгоритм возведения действительного числа в натуральную степень n за O(log n)?
5. Опишите нерекурсивный алгоритм бинарного поиска первого вхождения элемента в массиве.

```
// Бинарный поиск без рекурсии.
int BinarySearch( double* arr, int count, double element )
{
    int first = 0;
    int last = count; // Элемент в last не учитывается.
    while( first < last ) {
        int mid = ( first + last ) / 2;
        if( element <= arr[mid] )
            last = mid;
        else
            first = mid + 1;
    }
    // Все элементы слева от first строго больше искомого.
    return ( first == count || arr[first] != element ) ? -1 : first;
}
```

Сложность — log(n), память O(1)

6. Какова амортизированная стоимость операции Add в реализации динамического массива с удвоением буфера? Можно ли увеличивать буфер в 1.5 раза? Как это скажется на оценке?

Утверждение. Пусть в реализации функции *grow()* буфер удваивается. Тогда амортизированная стоимость функции *Add* составляет $O(1)$.

Доказательство. Рассмотрим последовательность из n операций *Add*.
Обозначим $P(k)$ - время выполнения *Add* в случае, когда *RealSize* $= k$.

- $P(k) \le c_1 k$, если $k = 2^m$.
- $P(k) \le c_2$, если $k \ne 2^m$.

$S(n) = \sum_{k=0}^{n-1} P(k) \le c_1 \sum_{m:2^m<n} 2^m + c_2 \sum_{k:k\ne 2^m} 1 \le 2c_1 n + c_2 n = (2c_1 + c_2)n.$

Амортизированное время $AC(n) = {}^{S(n)}/_{n} \le 2c_1 + c_2 = O(1)$.

Можно, на оценке никак не скажется (поменяется коэффициент при c1)

7. Сколько времени работает линейный поиск в односвязном списке в худшем

(O(n)) и в лучшем **(O(1))** случае? Сколько времени работает добавление и удаление элемента в середине списка (середина списка неизвестна, есть указатель на начало и конец списка) **(O(n/2)) = (O(n))** ?

8. Назовите преимущества и недостатки реализации очереди с помощью динамического массива.

Достоинства: можно не заботится, о том, что закончится память, можно быстро найти нужный элемент (например бинарным поиском если очередь отсортирована. В отличии от очереди реализованной на затрачивает меньше памяти.
Недостатки: большое время добавления элемента, если заканчивается память(из-за того, что необходима скопировать весь массив в новый буфер.

9. Назовите преимущества и недостатки реализации стека с помощью односвязного списка.

Достоинства: Добавление элемента всегда работает за одно и тоже время (т.к. нет необходимости компилировать весь стек, если вдруг кончится память),
Недостатки: Возможность перемещаться по стеку лишь в одном направлении, что затруднит поиск необходимого элемента, элементы списка могут располагаться в памяти разреженно, что оказывает негативный эффект на кэширование процессора.

10. Назовите преимущества и недостатки реализации дека с помощью динамического массива.

Достоинства: Занимает меньше памяти, чем реализация дека с помощью списка
Недостатки: Сложнее добавлять новые элементы если реализовывать не списком.

11. Опишите подход динамического программирования для вычисления рекуррентных функций двух аргументов: $F(x, y) = G(F(x - 1, y), F(x, y - 1))$. Как оптимизировать использование дополнительной памяти?

<u>Пример.</u> Вычисление рекуррентных функций нескольких аргументов.

$$F(x, y) = 3 \cdot F(x-1, y) - 2 \cdot F^2(x, y-1),$$
$$F(x, 0) = x, F(0, y) = 0.$$

Вычисление $F(x, y)$ сводится к вычислению двух $F(\cdot,\cdot)$ от меньших аргументов.

Есть перекрывающиеся подзадачи.

$F(x-1, y-1)$ в рекурсивном решении вычисляется дважды.

$F(x-2, y-1)$ в рекурсивном решении вычисляется три раза.

$F(x-n, y-m)$ в рекурсивном решении вычисляется C_{n+m}^{n} раз.

Снова будем использовать кэширование – сохранять результаты.

Вычисления будем выполнять от меньших аргументов к большим.

```
// Вычисление рекуррентного выражения от двух переменных.
int F( int x, int y )
{
    vector<vector<int>> values( x + 1 );
    for( int i = 0; i <= x; ++i ) {
        values[i].resize( y + 1 );
        values[i][0] = i; // F( x, 0 ) = x;
    }
    for( int i = 1; i <= y; ++i ) {
        values[0][i] = 0; // F( 0, y ) = 0;
    }
    // Вычисляем по столбцам для каждого x.
    for( int i = 0; i <= x; ++i ) {
        for( int j = 0; j <= y; ++j ) {
            values[i][j] = 3 * values[i - 1][j] -
```

При вычисление рекуррентной функции $F(x, y)$ можно было не хранить значения на всех рядах.
Для вычисления очередного ряда достаточно иметь значения предыдущего ряда.

Важная оптимизация ДП: Запоминать только те значения, которые будут использоваться для последующих вычислений.

Для вычисления числа Фибоначчи F_i также достаточно хранить лишь два предыдущих значения: F_{i-1} и F_{i-2}.

12. Вычисление наибольшей общей подпоследовательности.

Будем решать задачу нахождения наибольшей общей подпоследовательности с помощью ДП.

Сведем задачу к подзадачам меньшего размера:

$f(n_1, n_2)$ - длина наибольшей общей подпоследовательности строк $s_1[0..n_1]$, $s_2[0..n_2]$.

$$f(n_1,n_2) = \begin{cases} 0, & n_1 = 0 \vee n_2 = 0 \\ f(n_1-1,n_2-1)+1, & s[n_1] = s[n_2] \\ max(f(n_1-1,n_2), f(n_1,n_2-1)), & s[n_1] \neq s[n_2] \end{cases}$$

		A	B	C	A	B
	0	0	0	0	0	0
D	0	←↑ 0	←↑ 0	←↑ 0	←↑ 0	←↑ 0
C	0	←↑ 0	←↑ 0	↖ 1	← 1	← 1
B	0	←↑ 0	↖ 1	←↑ 1	←↑ 1	↖ 2
A	0	↖ 1	←↑ 1	←↑ 1	↖ 2	←↑ 2

13. Вычисление редакторского расстояния (расстояния Левенштейна).

$$D(i,j) = min\begin{cases} D(i-1,j-1) + m(S[i],T[j]) \\ 1 + D(i-1,j) \\ D(i,j-1) + 1 \end{cases}$$

Где m(S[i], T[j]) – 0, если символы S[i] и T[j] совпадают,
– 1, иначе.

- Первое выражение соответствует замене i-го символа первой строки на j-ый символ второй строки.
- Второе выражение соответствует удалению i-го символа первой строки и получению из S[0..i-1] строки T[0..j].
- Третье выражение соответствует получению из строки S[0..i] строки T[0..j-1] и добавлению T[j].

		А	Р	Е	С	Т	А	Н	Т
	0	1	2	3	4	5	6	7	8
Д	1	1	2	3	4	5	6	7	8
А	2	1	2	3	4	5	5	6	7
Г	3	2	2	3	4	5	6	6	7
Е	4	3	3	2	3	4	5	6	7
С	5	4	4	3	2	3	4	5	6
Т	6	5	5	4	3	2	3	4	5
А	7	6	6	5	4	3	2	3	4
Н	8	7	7	6	5	4	3	2	3

14. Жадный алгоритм в решении задачи размена монет. Пример, когда жадный алгоритм дает неверное решение.
15. Жадный алгоритм в решении задачи “Покрытие отрезками”.
16. Жадный алгоритм в решении задачи о рюкзаке.