**Вопросы по первому модулю.**

1. Что означают записи “f(n) = Θ(g(n))”, “f(n) = O(g(n))” и “f(n) = Ω(g(n))”?
   1 – ограничена и сверху, и снизу: $\Theta(g(n)) = \{f(n)$ : существуют положительные константы $c_1$, $c_2$ и $n_0$, такиечто$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ для всех $n \ge n_0\}$,

   2 – ограничена сверху: $O(g(n)) = \{f n$ : существуют положительные константы $c$ и $n_0$, такиечто$0 \le f(n) \le cg(n)$ для всех $n \ge n_0\}$,

   3 – ограничена снизу: $\Omega\, g(n) = \{f(n)$ : существуют положительные константы $c$ и $n_0$ такие, что $0 \le cg(n) \le f(n)$ для всех $n \ge n_0\}$
2. Чем плох рекурсивный алгоритм вычисления n-ого числа Фибоначчи?
   Тем, что значения вычисляются по нескольку раз. F(0) = 1; F(1) = 1;F(n) = F(n –1) + F(n –2); Переполнение стека, жрется память и тд.
   Решение – запоминать предыдущее значение.
   Объемдоп. Памяти $\boldsymbol{M(n)=O(n)}$–максимальнаяглубинарекурсии
   Такимобразом, $\boldsymbol{T(n)=\Omega(\varphi^n)}$. $\varphi$=1,6180339887– з о л о т о е с е ч е н и е
   Нерекурсивныйалгоритм:
   Времяработы $\boldsymbol{T(n)=O(n)}$–количество итераций в цикле.
   Объемдоп. Памяти $\boldsymbol{M(n)=O(1)}$.

3. Опишите алгоритм проверки числа на простоту за O(sqrt(n))?
   Будем перебирать все числа от 1 до sqrt($\boldsymbol{n}$), проверяя, делит ли какое-нибудь из них n.
   Объяснение этого принципа. Рассмотрим множители 100. 100 = 1 × 100, 2 × 50, 4 × 25, 5 × 20, 10 × 10, 20 × 5, 25 × 4, 50 × 2, 100 × 1. Обратите внимание, что после пары множителей 10 × 10 все пары множителей повторяются (только множители в этих парах переставлены местами). Поэтому вы можете игнорировать делители числа n большие, чем квадратный корень (n).

4. Опишите алгоритм возведения действительного числа в натуральную степень n за O(log n)?
   Воспользуемся тем, что $\boldsymbol{a^{2^k} = \left(\left(a^2\right)^2\right)^{\cdots 2}}$ ($\boldsymbol{k}$ раз).
   Если $\boldsymbol{n = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}}$, $\boldsymbol{k_1, k_2, \ldots, k_s}$ – **различны, то**
   $$\boldsymbol{a^n = a^{2^{k_1}} a^{2^{k_2}} \cdots a^{2^{k_s}}}.$$
   Как получить разложение $n = 2^{k1} + 2^{k2} + \ldots + 2^{ks}$ ?
   n = 10011010 – в двоичной системе счисления.
   Начнем с младших степеней. result = 1, aInPowerOf2 = a.
   n = 10011~~010~~ – пусть пройдено 3 шага алгоритма. Если следующий бит == 1, то домножим result на aInPowerOf2 = $(a^2)^3$ . Вне зависимости от бита возводим aInPowerOf2 в квадрат
   $a^n = a^{n/2} + a^{n/2}$ – если n четное
   $a^n = a^{n-1} * a$ – если n нечетное.

```
double Power( double a, int n )
{
double result = 1; // Для хранения результата.
double aInPowerOf2 = a; // Текущее значение ((a^2)^2...)^2
while( n > 0 ) {
// Добавляем нужную степень двойки к результату,
// если она есть в разложении n.
```

```
if( n & 1 == 1 ) {
        result *= aInPowerOf2;//сюда заходим только тогда, когда степень двойки достигла того //значения, которое
входит в разложение числа n.
    }
aInPowerOf2 *= aInPowerOf2;//тут копится значение a в нужной степени двойки
n = n >> 1; // Можно писать. }
  return result;
}
```

$T(n)=O(\log n), M(n)=O(1).$

5. Опишите нерекурсивный алгоритм бинарного поиска первого вхождения элемента в массиве.

   Шаг. Сравниваем элемент в середине массива (медиану) с заданным элементом. Выбираем нужную половинку массива в зависимости результата сравнения.

   Повторяем этот шаг до тех пор, пока размер массива не уменьшится до 1.

```
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;
}
```

ВАЖНО! В цикле проверка условия **first < last** должна выполняться строго на <, а не на <=.

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?

   Как долго работает функция Add добавления элемента?

   - В лучшем случае = $O(1)$
   - В худшем случае = $O(n)$

Амортизационный анализ отличается от анализа средних величин тем, что в нем не учитывается вероятность.

**Определение**.Пусть S(n) –время выполнения последовательности всех n операций в наихудшем случае.
**Амортизированной стоимостью(временем)** AC(n)называется среднее время, приходящееся на одну операцию *Snn*.

при амортизационном анализе гарантируется **средняя производительность операций в наихудшем случае**.

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

**Определение:**

**Амортизационный анализ** — метод подсчета времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Такой анализ чаще всего используется, чтобы показать, что даже если некоторые из операций последовательности являются дорогостоящими, то при усреднении по всем операциям средняя их стоимость будет небольшой за счёт низкой частоты встречаемости. Подчеркнём, что оценка, даваемая амортизационным анализом, не является вероятностной: это оценка среднего времени выполнения операций для худшего случая.

7. Сколько времени работает линейный поиск в односвязном списке в худшем и в лучшем случае? Сколько времени работает добавление и удаление элемента в середине списка (середина списка неизвестна, есть указатель на начало и конец списка)?

   O(n) в худшем случае, O(1) в лучшем. Добавление и удаление занимает O(1), но сам поиск будет занимать O(n) (пускаем указатели с начала и с конца, когда они станут равны – середина

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 списке затрачивает меньше памяти. Недостатки: большое время добавления элемента если заканчивается память(из-за того, что необходимо 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),O(1), нет лишней памяти, >100Мб. 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.В среднем медленнее, чем на динамическом массиве.

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.

**Сравнение списков с массивами.**
**Недостатки списков**:
- Нет быстрого доступа по индексу.

Расходуется дополнительная память

- Узлы могут располагаться в памяти разреженно, что не позволяет использовать кэширование процессора.

**Преимущества списков перед массивом**:
- Быстрая вставка узла.
- Быстрое удаление узла.

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?

Пример. 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] -
                2 * values[i][j - 1] * values[i][j - 1];
        }
    }
    return value[x][y];
}
```

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 $F_{i-2}$.

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?

# 12. Вычисление наибольшей общей подпоследовательности.

# Наибольшая общая подпоследовательность

Последовательность X является подпоследовательностью Y, если из Y можно удалить несколько элементов так, что получится последовательность X.

Задача. Наибольшая общая подпоследовательность (англ. longest common subsequence, LCS) – задача поиска последовательности, которая является подпоследовательностью нескольких последовательностей (обычно двух).

Элементами подпоследовательности могут быть числа, символы...

X = ABCAB,
Y = DCBA,
LCS( X, Y ) = BA, CA, CB.

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 |

В каждой ячейке – длина наибольшей общей подстроки соответствующих начал строк.

$f(n_1, n_2)$ – длина НОП.

Как восстановить саму подпоследовательность?

Можно хранить в каждой ячейке таблицы «направление» перехода. Переход по диагонали означает, что очередной символ присутствует в обоих последовательностях.

Начинаем проход от правого нижнего угла.
Идем по стрелкам, на каждый переход по диагонали добавляем символ в начало строящейся подпоследовательности.

«Направления» можно не хранить, а вычислять по значениям в таблице.

| | | 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 |

Желтым выделены ячейки, лежащие на лучшем пути.
Переход по диагонали соответствует совпадению символа.
Возможно существованию несколько лучших путей – по одному на каждую Наибольшую Общую Подпоследовательность.

## 13. Вычисление редакторского расстояния (расстояния Левенштейна).

**Расстояние Левенштейна** (также **редакторское расстояние** или **дистанция редактирования**) между двумя строками – это минимальное количество следующих операций:

- вставки одного символа,
- удаления одного символа,
- замены одного символа на другой,

необходимых для превращения одной строки в другую.

Впервые задачу упомянул в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей 0-1.

Расстояние Левенштейна и его обобщения активно применяется:

- для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи).
- для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы.
- в биоинформатике для сравнения генов, хромосом и белков.

Будем вычислять расстояние Левенштейна с помощью Динамического программирования.

*D(i, j)=количество операций, необходимых для приведения одной подстроки S[0..i] к другой T[0..j].*

Краевые значения:

D(i, 0) = i для всех i в диапазоне [0, |S|].

D(0, j) = j для всех j в диапазоне [0, |T|].

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

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].

Пример.

| | | М | Е | Т | Р | О |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| Л | 1 | 1 | 2 | 3 | 4 | 5 |
| Е | 2 | 2 | 1 | 2 | 3 | 4 |
| Т | 3 | 3 | 2 | 1 | 2 | 3 |
| О | 4 | 4 | 3 | 2 | 2 | 2 |

**14. Жадный алгоритм в решении задачи размена монет. Пример, когда жадный алгоритм дает неверное решение.**

## Жадные алгоритмы. Размен монет.

ТЕХНОПАРК

Монетная система некоторого государства состоит из монет достоинством $a_1 = 1 < a_2 < \cdots < a_n$ . Требуется выдать сумму S наименьшим возможным количеством монет.

Жадный алгоритм:
Берём наибольшее возможное количество монет наибольшего достоинства $a_n$: $x_n = \lfloor S/a_n \rfloor$.

Далее аналогично получаем, сколько нужно монет меньшего номинала для выдачи остатка.


- Не всегда даёт оптимальное решение.
  Например. Монеты в 1, 5 и 7коп. Сумма 24.
  Жадный алгоритм разменивает так:
  7к. — 3 шт., 1к. — 3 шт.
  Правильное решение:
  7к. — 2 шт., 5к. — 2 шт.
- Тем не менее, на всех *реальных* монетных системах жадный алгоритм даёт правильный ответ.

**15. Жадный алгоритм в решении задачи “Покрытие отрезками”.**

## Жадные алгоритмы. Покрытие отрезками.


Задача. Дано множество отрезков $[a_i, b_i]$, покрывающее отрезок $[0, X]$. Найти наименьшее подпокрытие, т.е. минимальный набор отрезков, по-прежнему покрывающий отрезок $[0, X]$.

Жадное решение.

Упорядочим набор отрезков по возрастанию левого конца $a_i$.

- Шаг 1. Среди отрезков, содержащих 0, найдем такой, у которого наибольший правый конец $b_i$. Обозначим этот отрезок $[\widetilde{a_1}, \widetilde{b_1}]$.
- Шаг 2. Среди отрезков, содержащих $\widetilde{b_1}$, найдем такой, у которого наибольший правый конец $b_i$. Обозначим этот отрезок $[\widetilde{a_2}, \widetilde{b_2}]$.
- И так далее.

## 16. Жадный алгоритм в решении задачи о рюкзаке.

## Жадные алгоритмы. Задача о рюкзаке.


Жадный алгоритм.

Предметы сортируются по убыванию стоимости единицы каждого (по отношению цены к весу).

Шаг 1. Помещаем в рюкзак первый предмет из отсортированного массива, который поместится в рюкзак.

Шаг 2. Помещаем в рюкзак первый из оставшихся предметов отсортированного массива, который поместится в рюкзак.

И т.д., пока в рюкзаке остается место или все оставшиеся предметы оказались тяжелее.