ОГЛАВЛЕНИЕ

Разбор домашнего задания

Level 1

Найти индекс заданного числа в массиве: `{3, 6, 4, 7, 2, 1, 9}`

Алгоритм на вход должен получать любой массив и одну цифру, индекс которой требуется найти.

Задание считается выполненным если: использован ”Линейный подход” и алгоритм соответствует характеристикам алгоритма и выдает верный результат.

Level 2

Реализовать алгоритм, который будет находить сумму квадратов всех элементов массива! `{3, 6, 4, 7, 2, 1, 9}`

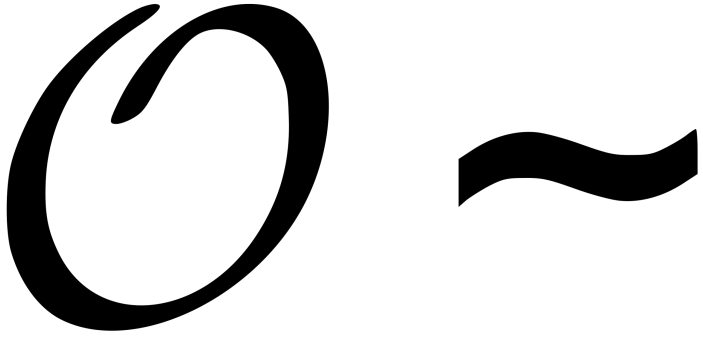
Задание считается выполненным если: использован ”Линейный подход” и алгоритм соответствует характеристикам алгоритма и выдает верный результат.

Асимптотический анализ

В асимптотическом анализе мы оцениваем производительность алгоритма с точки зрения размера входных данных, другими словами мы подразумеваем **анализ времени**, которое потребуется для обработки очень большого набора данных.

⁉️ Имея два алгоритма для задачи, как мы узнаем, какой из них лучше?

Мы рассчитываем, как время “time complexity” и пространство “space complexity”, занимаемое алгоритмом, увеличивается с размером входных данных.



Big(O)

`Big O` нотация нужна для описания сложности алгоритмов. Для этого используется понятие времени.

Прежде чем начать

📌 Концепцию **`Big O`** нужно понимать, чтобы уметь видеть и исправлять неоптимальный код.
📌 Ни один серьезный проект, как и ни одно серьезное собеседование не могут обойтись без вопросов про **`Big O`**
📌 Непонимание **`Big O`** ведет к серьезной потери производительности ваших алгоритмов

Пример

Вам нужно передать файл 500GB другу, который живет за **4000** км от вас.

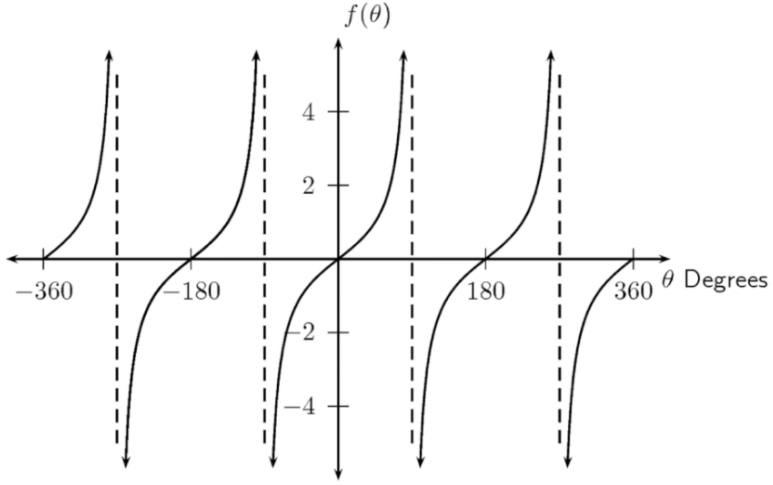
 Интернет	 Самолет

Как это лучше сделать?

Временная оценка

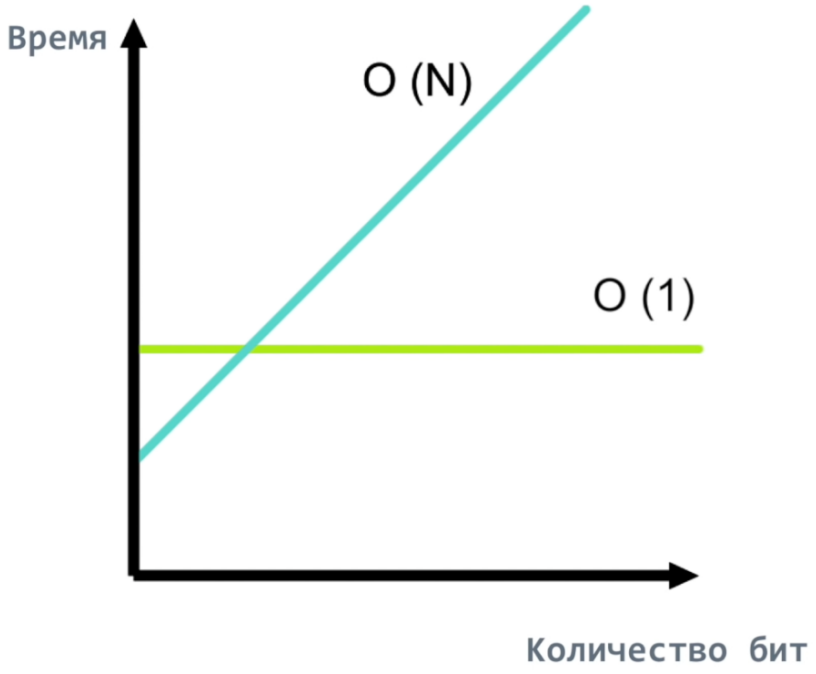
Идея `Big O` показать какое количество шагов нужно сделать чтобы алгоритм закончил свое выполнение!

Big O математическое обозначение для сравнения асимптотического поведения функций.



Другими словами (в рамках computer science) Big O показывает **верхнюю границу** зависимости между **входными параметрами** функции и **количеством операций**, которые выполняет.

Временная оценка



📌 Передача файлов по сети **O(n)**

Чем больше байт, тем дольше передавать.

📌 Перенос носителя в самолете **O(1)**

Размер файла не важен, скорость самолета не изменяется.

Асимптотический анализ в действии

Рассмотрим задачу поиска (поиск заданного элемента) в отсортированном массиве.

 Линейный поиск. Порядок роста — **линейный** `O(n)`

Рекурсивные функции, линейные функции.

 Бинарный поиск. Порядок роста — **логарифмический** `O(log n)`

Компьютер A – константное время 0.2 сек

Компьютер B – константное время 1000 сек

Отбрасывание констант

📌 **`Big O`** описывает только скорость роста

📌 Поэтому мы отбрасываем все константы при оценке сложности

📌 Поэтому алгоритм, описываемый как **`O(2n + n)`** должен описываться как **`O(n)`**

Это процесс упрощения выражения!

(Комбинаторная оптимизация)

Время выполнения линейного поиска в секундах для: **A : 0,2 * n**

Время выполнения бинарного поиска в секундах для: **B : 1000*log(n)**

n	time on A	time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

Причина в том, что **порядок роста** бинарного поиска по размеру входных данных является логарифмическим, а порядок роста линейного поиска — линейным.

Неважная сложность

Как быть со сложностью:	1. N не является константой
O(2n^2 + n + 6)	2. **O(n^2 + n^2) = O(n^2)**
?	3. **n^2 > n** (значительно больше)
	4. Следовательно O(n^2 + n) =

Асимптотический анализ – порядок роста

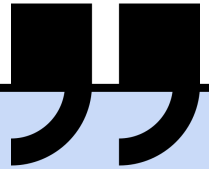
> Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных.
>
> Порядок роста представляется в виде **O-нотации:**
>
> **O(f(x))**, где **f(x)** — формула, выражающая сложность алгоритма.

O(1) – Константный

Порядок роста **O(1)** означает, что вычислительная сложность алгоритма не зависит от размера входных данных.

```
public int getSize(int[] arr) {
 return arr.length;
}
```

O(n) – линейный



Порядок роста **O(n)** означает, что сложность алгоритма **<u>линейно</u>** растет с увеличением входного массива.
Если линейный алгоритм обрабатывает один элемент 1 секунду, то 100 элементов обработается за 100 секунд.

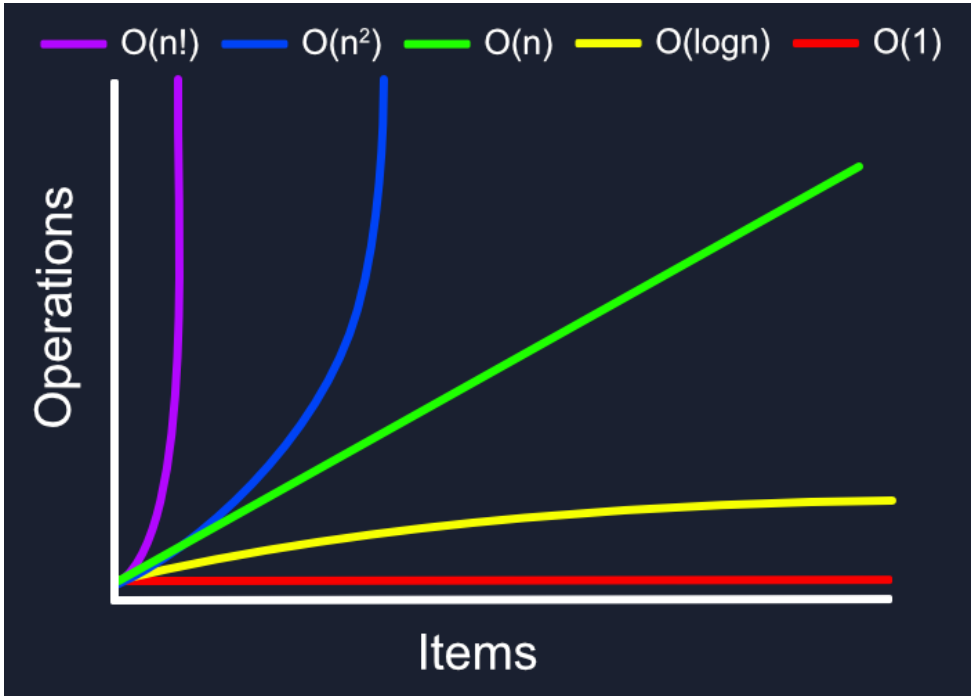
Пример:

```java
public long getSum(int[] arr) {
  long sum = 0;
  for (int i = 0; i < arr.length; i++) {
   sum += i; }
 return sum;
}
```

```javascript
const getSum = (arr) => {
 let sum = 0;
 for (let i = 0; i < arr.length; i++) {
   sum += arr[i];
 }
 return sum;
};
```

`O(log n)` – логарифмический

Порядок роста **`O( log n)`** означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива.
Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность.



O(n^2) – квадратичный

> Время работы алгоритма **O(n^2)** зависит от квадрата размера входного массива.
> Квадратичная сложность — повод задуматься и переписать алгоритм.

👉 Массив из **100** элементов потребует **1 0000** операций,

👉 Массив из миллиона элементов потребует **1 000 000 000 000** (триллион) операций.

👉 Если одна операция занимает миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов **32** года.

👉 Даже если он будет в сто раз быстрее, работа займет 84 дня.

O(n!) – факториальный

Очень медленный алгоритм ~~~ 

Задача коммивояжёра

Задача коммивояжёра (или TSP от англ. *travelling salesman problem*) одна из самых известных задач комбинаторной оптимизации, заключающаяся в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвратом в исходный город.

https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BC%D0%B8%D0%B2%D0%BE%D1%8F%D0%B6%D1%91%D1%80%D0%B0

Наилучший, средний и наихудший случай

Обычно имеется в виду **наихудший случай**, за исключением тех случаев, когда наихудший и средний сильно отличаются.

Например: `ArrayList.add()`
В среднем имеет порядок роста `O(1)`, но иногда может иметь `O(n)`.
В этом случае мы будем указывать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает.

Самое **важное** здесь то, что `O(n)` означает, что алгоритм потребует **не более n шагов**!

Практика

Какой код быстрее?

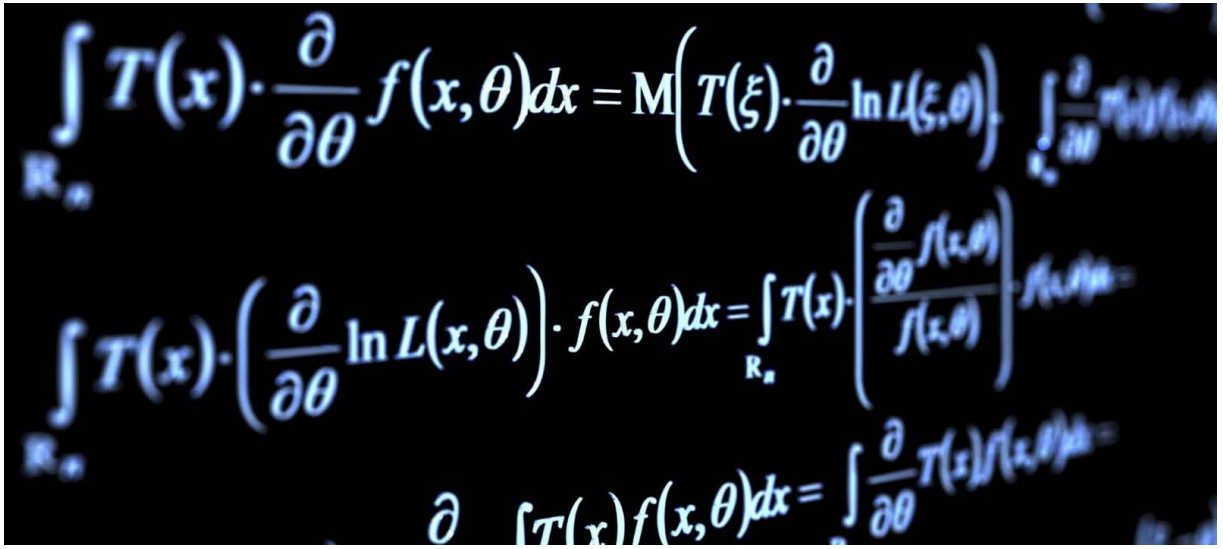
```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
  if (x < min) min = x;
  if (x > max) max = x;
}
```

```
int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;
for (int x : array) {
  if (x < min) min = x;
}
for (int x : array) {
  if (x > max) max = x;
}
```

Ответ:

Упростите выражения

1	**O**(n^2 + n)	
2	**O**(n + log n)	
3	**O**(5 * 2^n + 10 * n^2)	
4	**O**(n^2 + B)	



Найдите Big O

В функции есть два последовательных цикла for, каждый из которых проходит массив длиной N

Исходный код

```
void foo(int[] array) {
  int sum = 0;
  int product = 1;
  for (int i = 0; i < array.length; i++)
    sum += array[i];
  for (int i = 0; i < array.length; i++)
    product *= array[i];
  println(sum + ", " + product);
}
```

Псевдо код

```
foo(array) {
  for // N раз
  for // N раз
}
```

Ответ:

Найдите Big O

В функции есть цикл в цикле, каждый из них проходит массив длиной N

Исходный код

```
void printPairs(int[] arr) {
  for (int i = 0; i < arr.length; i++)
    for (int j = 0; j < arr.length; j++)
      println(array[i] + "," + array[j]);
}
```

Псевдо код

```
printPairs(arr) {
  for   // N раз
    for // N раз
}
```

Ответ:

Что мы в итоге измеряем и всегда ли это работает?

При измерении сложности алгоритмов и структур данных мы обычно говорим о двух вещах: количество операций, требуемых для завершения работы (вычислительная сложность), и объем ресурсов, в частности, памяти, который необходим алгоритму (пространственная сложность).

📌 Алгоритм, который выполняется в десять раз быстрее, но использует в десять раз больше места, может вполне подходить для серверной машины с большим объемом памяти.

➡️ Но на встроенных системах, где количество памяти ограничено, такой алгоритм использовать нельзя.

Асимптотический анализ не идеален, но это лучший доступный способ анализа алгоритмов.

- два алгоритма сортировки, которые занимают на машине 1000 nLogn и 2 nLogn
- мы не можем судить, какой из них лучше, поскольку мы игнорируем константы
- таким образом, вы можете в конечном итоге выбрать алгоритм, который асимптотически медленнее, но быстрее для вашего программного обеспечения.

Асимптотический анализ – основное, что надо запомнить!

1. Скорость алгоритма измеряется не в секундах, а в приросте количества операций.

2. Быстро возрастает время работы алгоритма в зависимости от увеличения объема входящих данных.

3. Время работы алгоритма выражается при помощи нотации большого «O».

4. Алгоритм со скоростью O(log n) быстрее, чем со скоростью O(n), но он становится намного быстрее по мере увеличения списка элементов.

Time complexity на простых примерах

Временная сложность

Я кому то из вас дал **ручку!** 

У меня есть несколько способов найти мою ручку.

- ~~Я иду и спрашиваю у первого человека в классе, есть ли у него ручка. А дальше мне самому лениво идти к следующему и я просто начинаю спрашивать этого человека о других людях в классе, есть ли у них эта ручка? и так далее.~~

- ~~Я иду и спрашиваю каждого ученика по отдельности.~~

- ~~Я делю класс на две группы и спрашиваю: «Где моя ручка?» Затем я беру эту половину, делю ее на две части и спрашиваю снова, и так далее. Пока у меня не останется один ученик, у которого есть моя ручка.~~

- Точно, я вспомнил кому я ее дал.

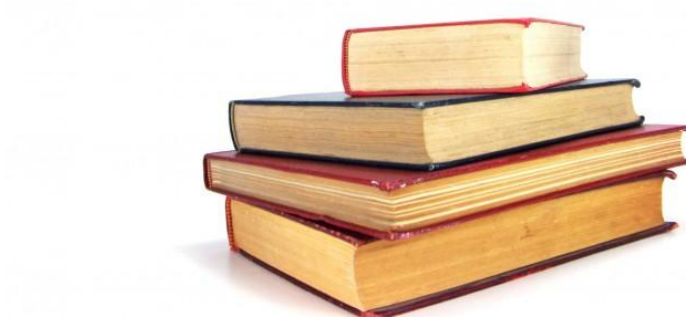
Литература

Для изучения алгоритмов рекомендую книгу **"Introduction to Algorithms"** Т. Кормена, Ч. Лейзерсона, R. Rivest и C. Stein.

Это классический учебник, который предоставляет глубокое понимание ключевых концепций и методов алгоритмов.

"Introduction to Algorithms" можно найти в интернете в формате электронной книги или купить в книжном магазине. Вот некоторые ссылки для покупки книги:

- Amazon: https://www.amazon.com/Introduction-Algorithms-Thomas-H-Cormen/dp/0262033844
- Barnes & Noble: https://www.barnesandnoble.com/w/introduction-to-algorithms-thomas-h-cormen/1100687969
- Books-A-Million: https://www.booksamillion.com/p/Introduction-Algorithms/Thomas-H-Cormen/Q3931717706740?id=7718718907746



Упражнения

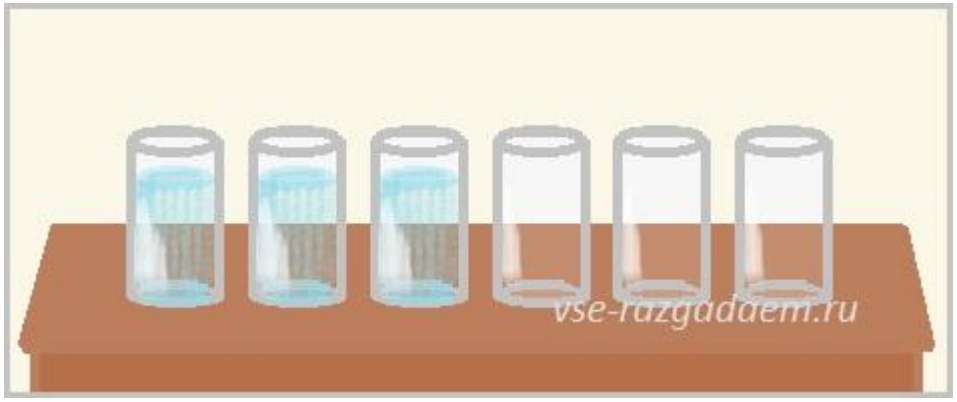
1. **O(3n + 2)**
2. **O(2^n + n^2 + n)**
3. **O(nlogn + n)**
4. **O(n^3 + 10n^2)**
5. **O(logn + log(logn) + log(loglogn))**
6. **O(n^2 + 2^n)**
7. **O(nlogn + n^2)**
8. **O(sqrt(n) + logn)**
9. **O(n^3 + nlogn)**
10. **O(1 + 1 + 1 + 1)**

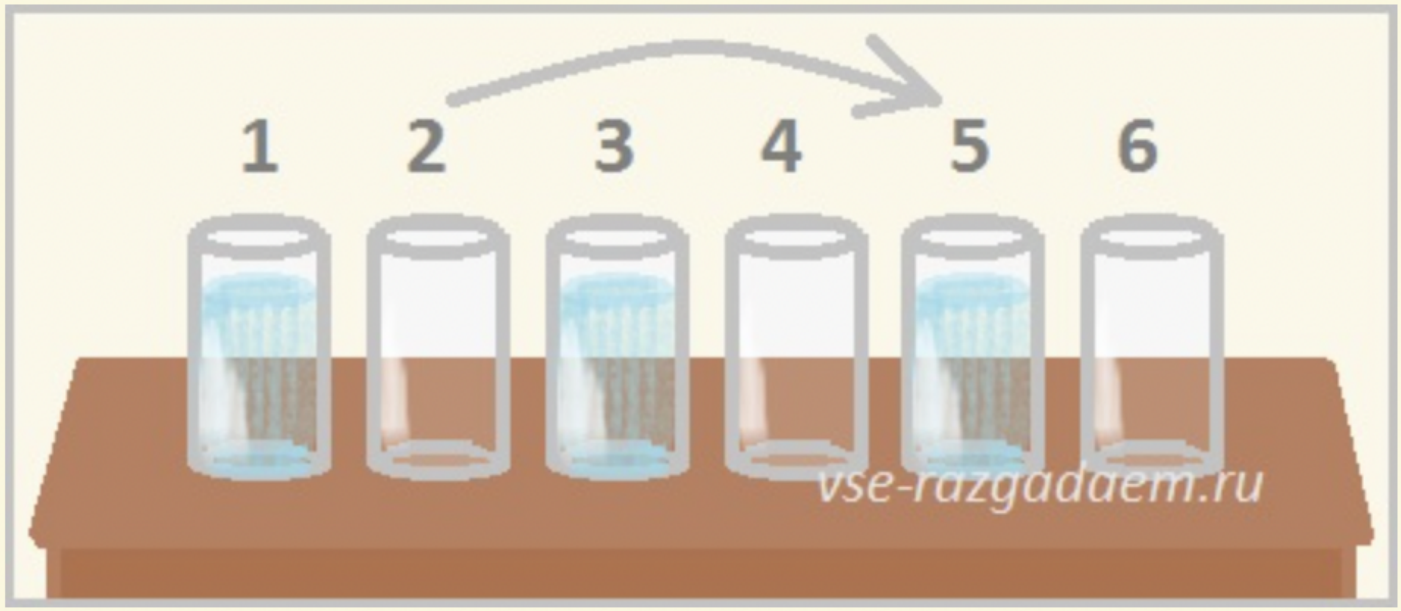
1. O(log n + B)	
2. O(n^3 + 10^n)	
3. O(log + 2 +n^2 +B)	
4. O(n^logn + n^n)	
5. O((logn+logn)+5+(n+n)+B+n^6)	

1. O(n + 2^logn)	
2. O(2n ^ 2n)	
3. O(n^3 + n + x)	
4. O(2n^ 2 + logn * 2n^ 2)	
5. O(2^n + 2^n)	

Поиск алгоритма решения

На столе стоят шесть стаканов. Три из них наполнены водой, три – пустые. Как сделать так, чтобы пустые и полные стаканы чередовались? Важное условие: брать в руки можно только один стакан (совершать действие можно только с одним стаканом).





Перелив воду из второго стакана в пятый и вернув его на место, вы сможете сделать так, чтобы пустые и полные стаканы чередовались.

Домашнее задание

Level 1

Какова временная сложность?

```
void test1(int n)
{
   if (n==1)
      return;
   for (int i=1; i<=n; i++)
       for (int j=1; j<=n; j++)
           System.out.println("*");
           break;
}
```

```
void test2(int n)
{
   int a = 0;
   for (i = 0; i < n; i++)
      for (j = n; j > i; j--)
          a = a + i + j;
}
```

Level 2

Какова временная сложность?

```
void test3(int n)
{
   int i, j, a = 0;
   for (i = n/2; i <=n; i++)
      for (j = 2; j <=n; j=j*2)
          a=a+n/2;
}
```

```
*void test4(int n)
{
   int a = 0, i = n;
   while (i > 0) {
      a += i;
      i /= 2;
   }
}
```

+ Читать личетаруру 📚