

# Google Hash Code 2017

UniBG - 08 / 02 / 2017



[seclab.unibg.it](http://seclab.unibg.it)

# Google Hash Code

---

- What is it?
- How does it work?
  - Team composed by 2-4 members
  - It's necessary to register the team and then join the hub
  - Divide tasks between team members (parsing, output, algorithm, ...)
- The perfect solution **is not necessary** (does not exist?)
- You must optimize
  - Find a way to maximize/minimize a goal function
- Several techniques
  - Dynamic Programming with approximations
  - Greedy Algorithms (choose the local optimal choice)



# Hash Code

## Pizza

Practice Problem for Hash Code 2017

# Introduction

Did you know that at any given time, someone is cutting pizza somewhere around the world? The decision about how to cut the pizza sometimes is easy, but sometimes it's *really* hard: you want just the right amount of tomatoes and mushrooms on each slice. If only there was a way to solve this problem using technology...

## Problem description

### Pizza

The pizza is represented as a rectangular, 2-dimensional grid of  $R$  rows and  $C$  columns. The cells within the grid are referenced using a pair of 0-based coordinates  $[r, c]$ , denoting respectively the row and the column of the cell.

Each cell of the pizza contains either:

- mushroom, represented in the input file as  $\mathbf{M}$ ; or
- tomato, represented in the input file as  $\mathbf{T}$

### Slice

A slice of pizza is a rectangular section of the pizza delimited by two rows and two columns, without holes. The slices we want to cut out must contain at least  $L$  cells of each ingredient (that is, at least  $L$  cells of mushroom and at least  $L$  cells of tomato) and at most  $H$  cells of any kind in total - surprising as it is, there is such a thing as too much pizza in one slice.

The slices being cut out cannot overlap. The slices being cut do not need to cover the entire pizza.

### Goal

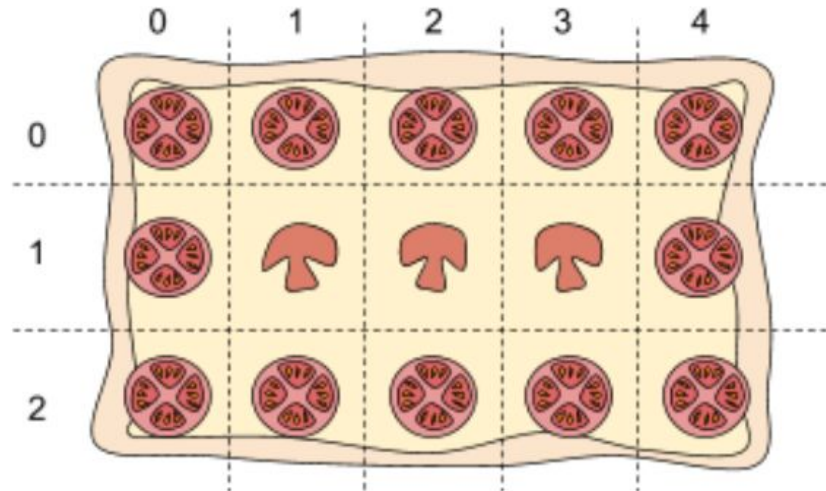
The goal is to cut correct slices out of the pizza maximizing the total number of cells in all slices.

## File format

The file consists of:

- one line containing the following natural numbers separated by single spaces:
  - ***R*** ( $1 \leq R \leq 1000$ ) is the number of rows,
  - ***C*** ( $1 \leq C \leq 1000$ ) is the number of columns,
  - ***L*** ( $1 \leq L \leq 1000$ ) is the minimum number of each ingredient cells in a slice,
  - ***H*** ( $1 \leq H \leq 1000$ ) is the maximum total number of cells of a slice
- ***R*** lines describing the rows of the pizza (one after another). Each of these lines contains ***C*** characters describing the ingredients in the cells of the row (one cell after another). Each character is either 'M' (for mushroom) or 'T' (for tomato).

## Example



```
3 5 1 6  
TTTTT  
TMMMT  
TTTTT
```

3 rows, 5 columns, min 1 of each ingredient per slice, max 6 cells per slice

**Example input file.**

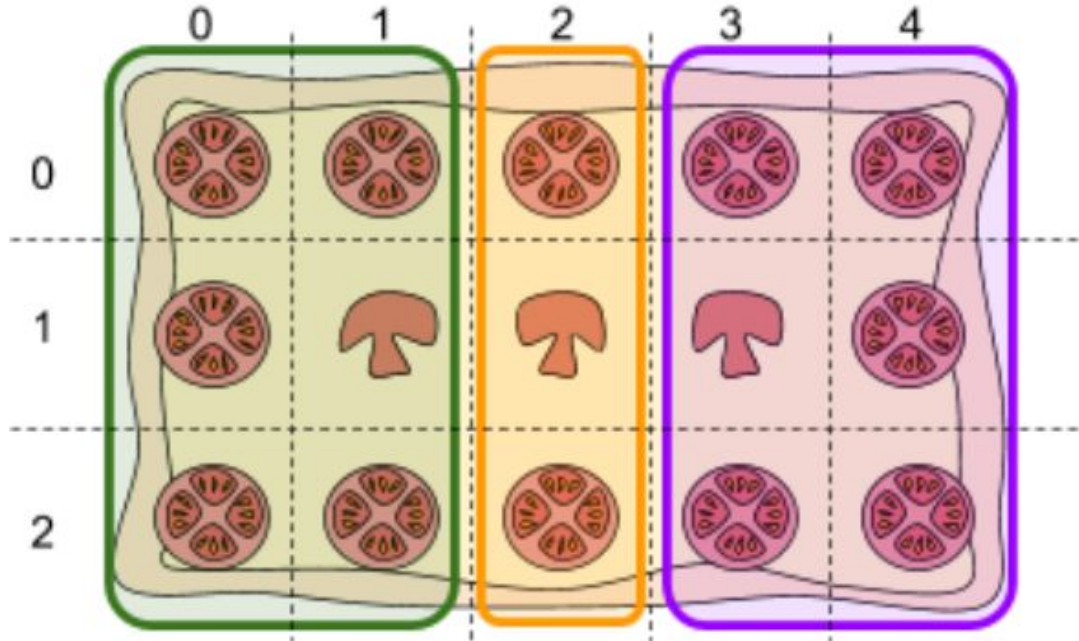
# Submissions

## File format

The file must consist of:

- one line containing a single natural number  $S$  ( $0 \leq S \leq R \times C$ ), representing the total number of slices to be cut,
- $U$  lines describing the slices. Each of these lines must contain the following natural numbers separated by single spaces:
  - $r_1, c_1, r_2, c_2$  ( $0 \leq r_1, r_2 < R, 0 \leq c_1, c_2 < C$ ) describe a slice of pizza delimited by the rows  $r_1$  and  $r_2$  and the columns  $c_1$  and  $c_2$ , including the cells of the delimiting rows and columns. The rows ( $r_1$  and  $r_2$ ) can be given in any order. The columns ( $c_1$  and  $c_2$ ) can be given in any order too.





**Slices described in the example submission file marked in green, orange and purple.**

**Example**

3	3 slices.
0 0 2 1	First slice between rows (0,2) and columns (0,1).
0 2 2 2	Second slice between rows (0,2) and columns (2,2).
0 3 2 4	Third slice between rows (0,2) and columns (3,4).

**Example submission file.**



## Validation

For the solution to be accepted:

- the format of the file must match the description above,
- each cell of the pizza must be included in at most one slice,
- each slice must contain at least  $L$  cells of mushroom,
- each slice must contain at least  $L$  cells of tomato,
- total area of each slice must be at most  $H$

## Scoring

The submission gets a score equal to the total number of cells in all slices.

**Note that there are multiple data sets representing separate instances of the problem. The final score for your team is the sum of your best scores on the individual data sets.**

## Scoring example

The example submission file given above cuts the slices of 6, 3 and 6 cells, earning  $6 + 3 + 6 = 15$  points.

# Dynamic Programming

- a gentle introduction -

# The Fibonacci Sequence

**1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377...**

$$1+1=2$$

$$1+2=3$$

$$2+3=5$$

$$3+5=8$$

$$5+8=13$$

$$8+13=21$$

$$13+21=34$$

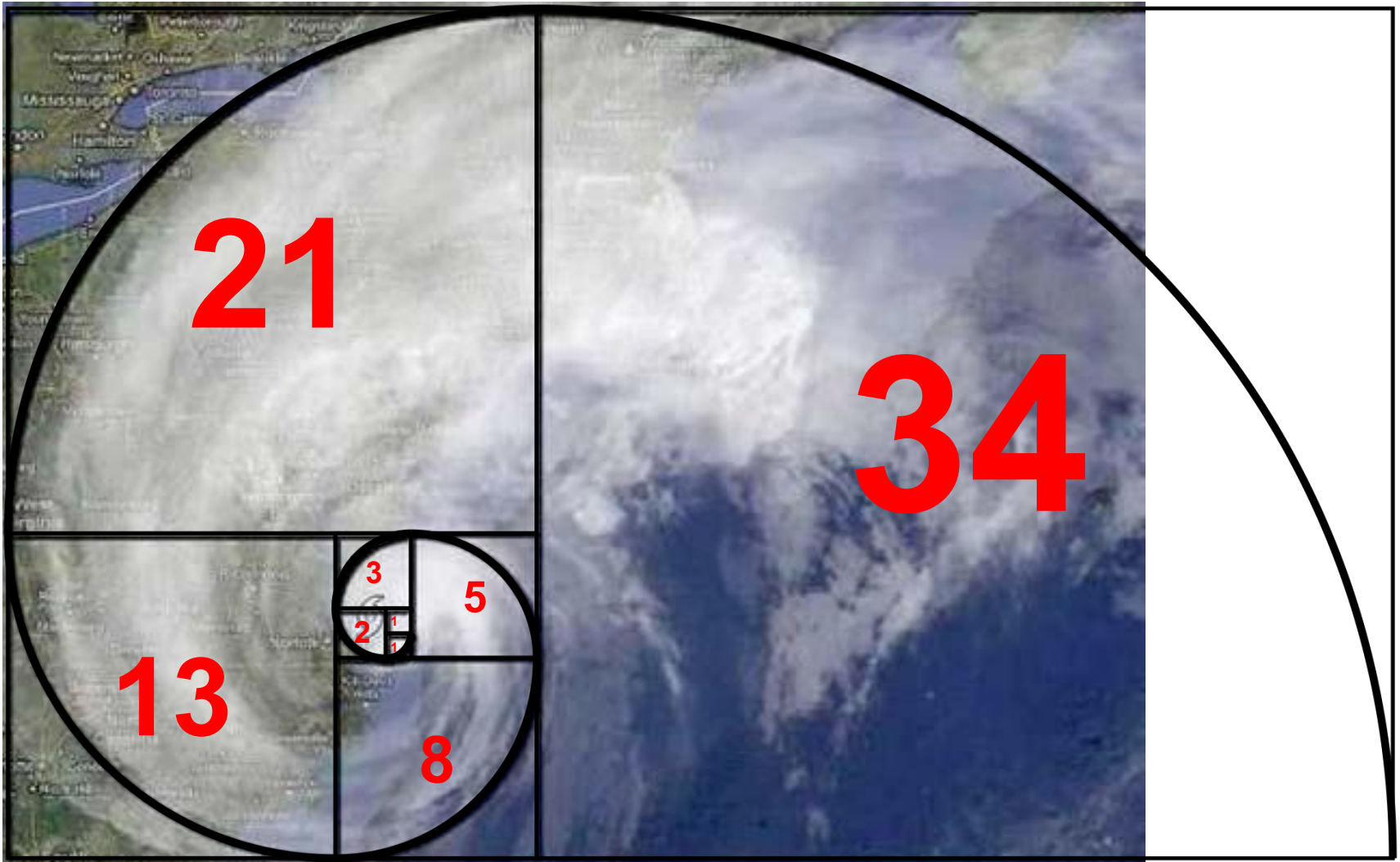
$$21+34=55$$

$$34+55=89$$

$$55+89=144$$

$$89+144=233$$

$$144+233=377$$



**Problem:**

**Compute Fibonacci for  
N OVER 9000!**

# static int fibonacci\_1(int n)

---

```
1.     static int fibonacci_1(int n) {
2.         System.out.println("computing " + n);
3.         if (n <= 2)
4.             return 1;
5.         else
6.             return fibonacci_1(n - 1) + fibonacci_1(n - 2);
7.     }
```



# main

---

```
1.     public static void main(String[] args) {
2.         Scanner sc = new Scanner(System.in);
3.         int n = sc.nextInt();
4.         System.out.println(fibonacci_1(n));
5.     }
```

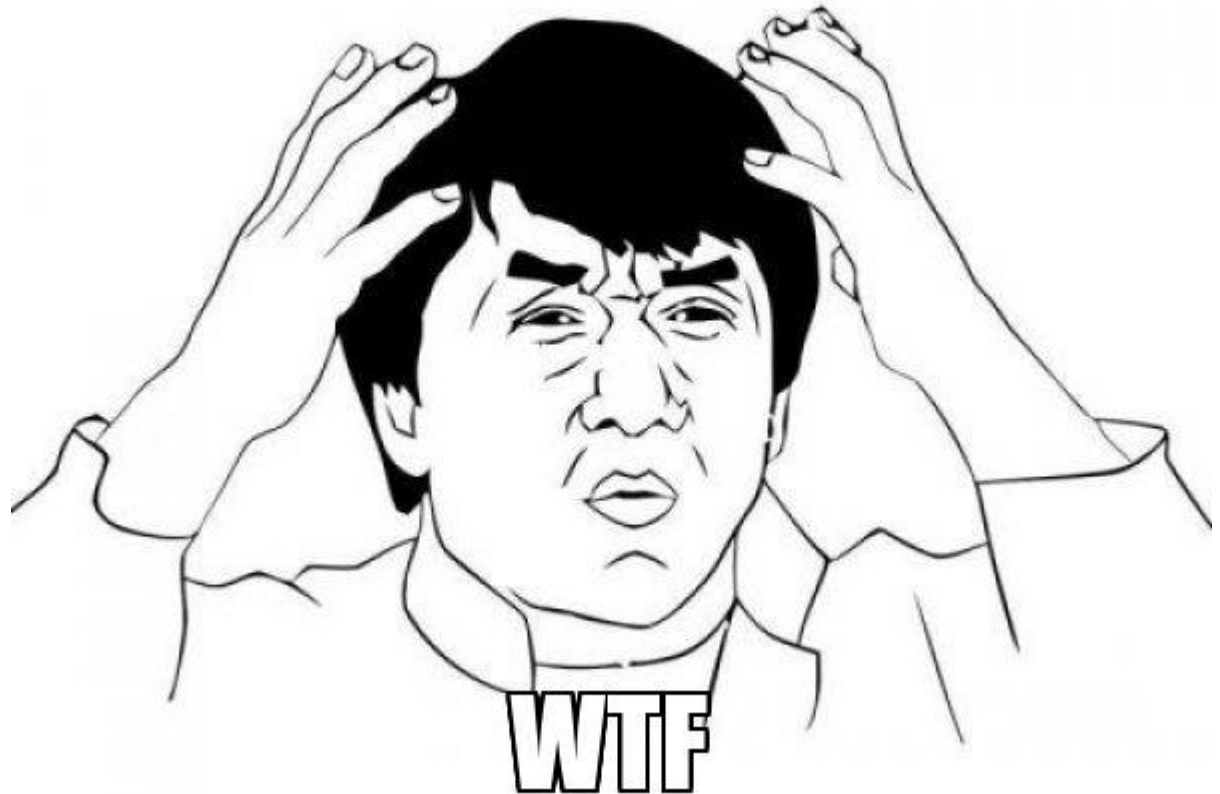
# Ok, let's try with 6

---

6

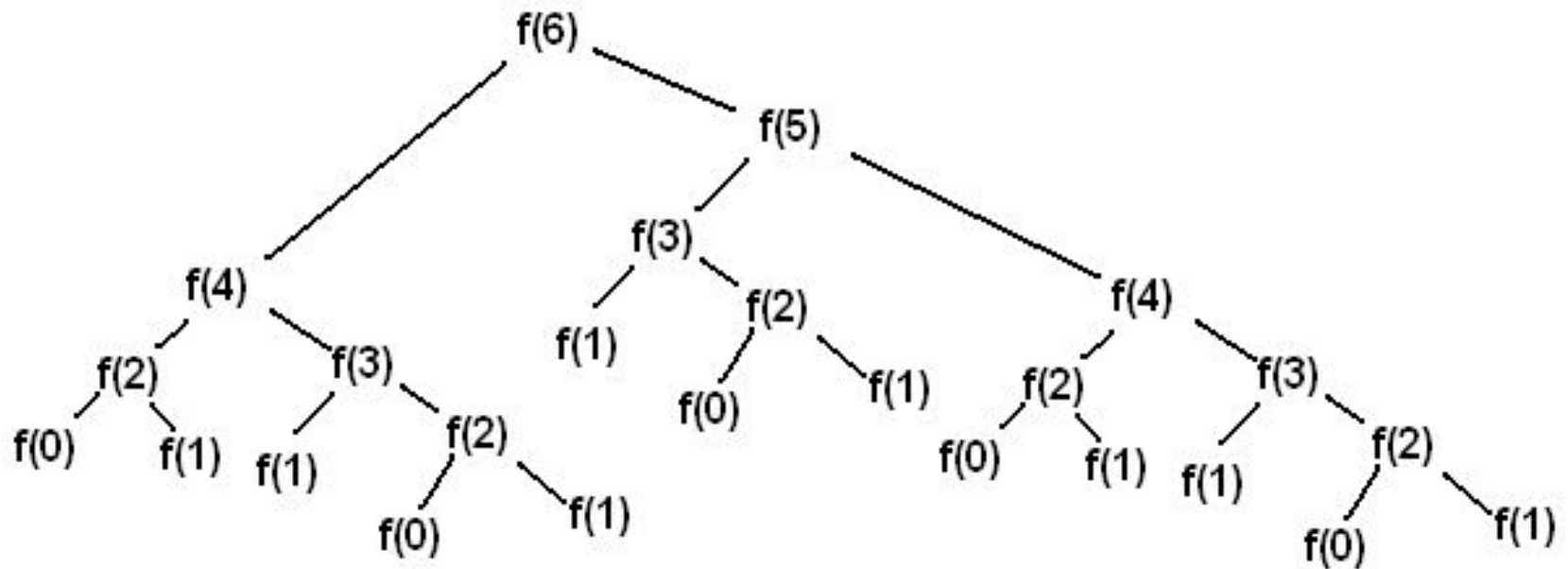
computing 6  
computing 5  
computing 4  
computing 3  
computing 2  
computing 1  
**computing 2**  
**computing 3**  
**computing 2**  
**computing 1**  
**computing 4**  
**computing 3**  
**computing 2**  
**computing 1**  
**computing 2**

8



# Successione di Fibonacci - Stack delle chiamate

---



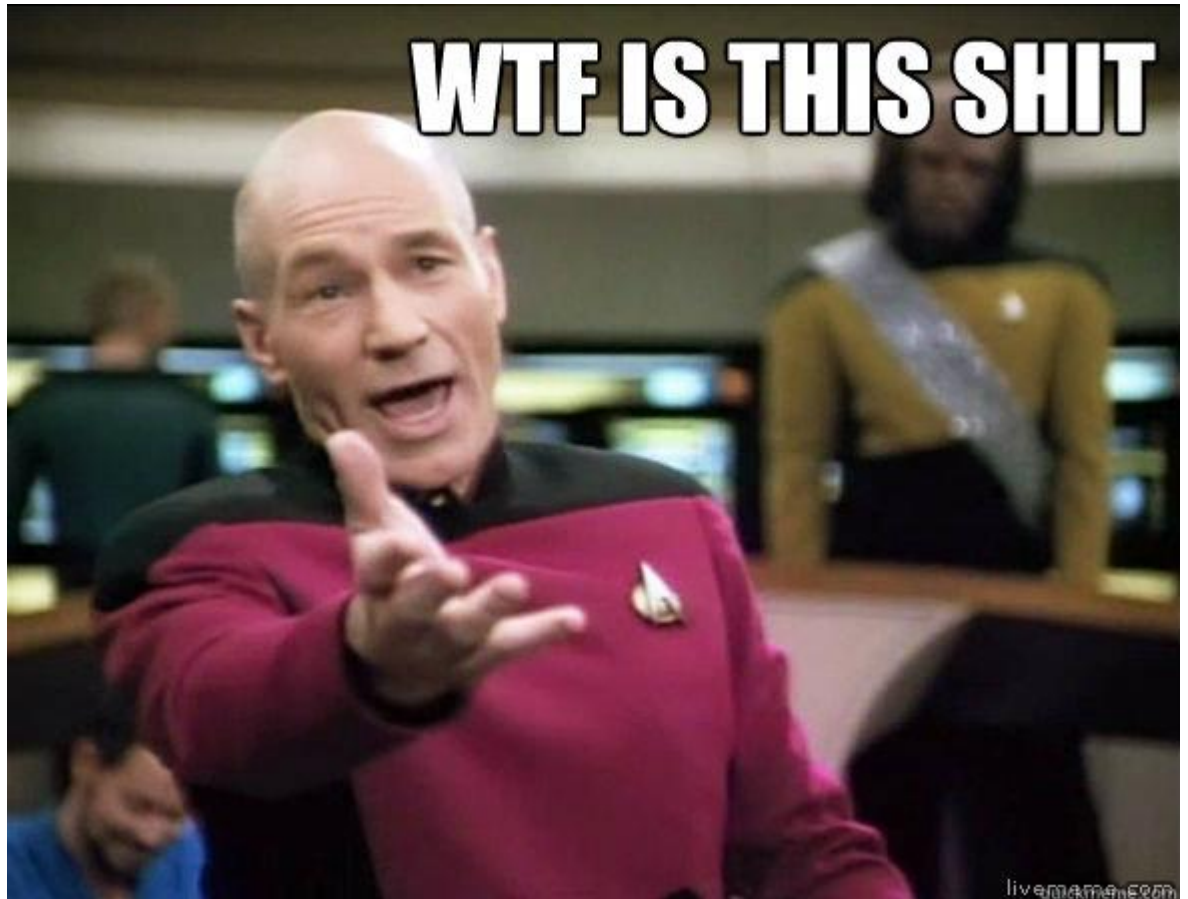
# 6 works, let's try 100!

---

```
100
computing 100
computing 99
computing 98
...
...
...
computing 99
...
...
computing 98
...
...
...
```

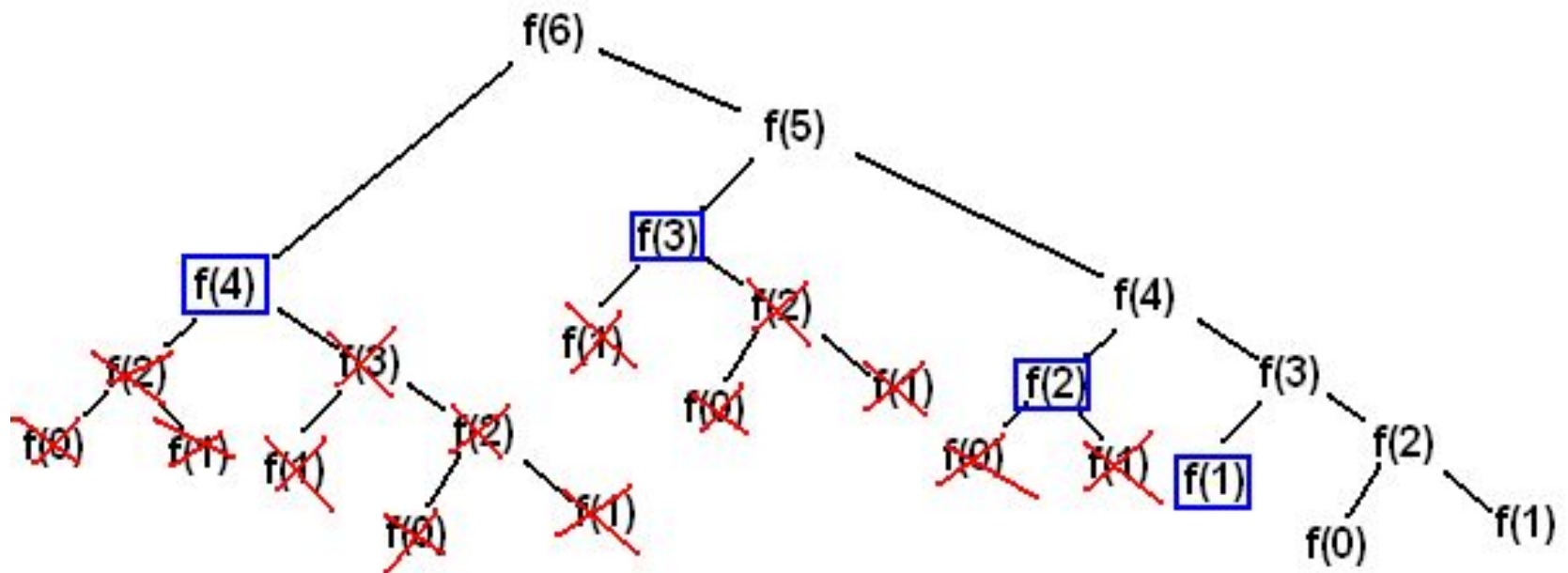
**5 minutes later**

**Not yet finished**



# Successione di Fibonacci - Dynamic Programming

---



# static int fibonacci\_2(int n)

---

```
1.     static Map<Integer, Integer> cache_2 = new HashMap<Integer, Integer>();
2.
3.     static Integer fibonacci_2(int n) {
4.         if (cache_2.containsKey(n))
5.             return cache_2.get(n);
6.
7.         System.out.println("computing " + n);
8.         int result;
9.
10.        if (n <= 2)
11.            result = 1;
12.        else
13.            result = fibonacci_2(n - 1) + fibonacci_2(n - 2);
14.
15.        cache_2.put(n, result);
16.        return result;
17.    }
```



# What about 100 now?

---

100

computing 100

computing 99

computing 98

computing 97

computing 96

computing 95

...

computing 5

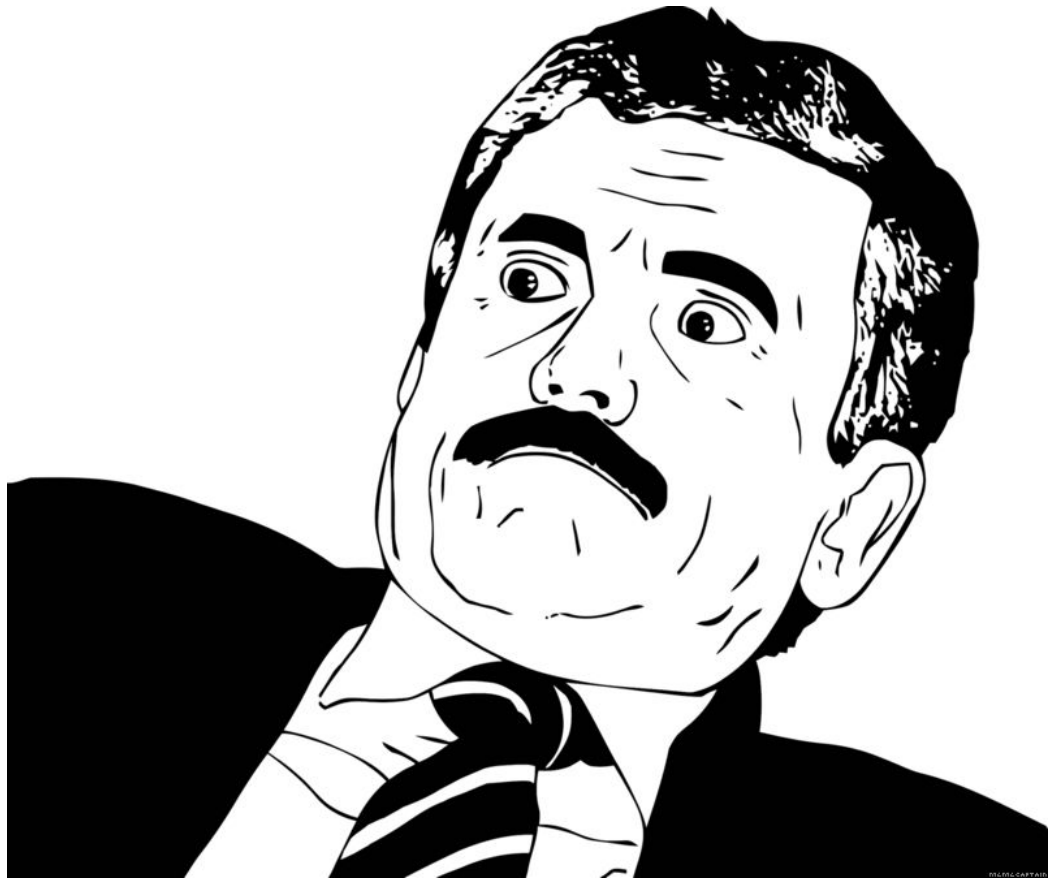
computing 4

computing 3

computing 2

computing 1

**-980107325**



java.math

## Class BigInteger

java.lang.Object

java.lang.Number

java.math.BigInteger

### All Implemented Interfaces:

Serializable, Comparable<BigInteger>

```
public class BigInteger  
extends Number  
implements Comparable<BigInteger>
```

**Immutable arbitrary-precision integers.** All operations behave as if BigInteger were represented in two's-complement notation (like Java's primitive integer types). BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.



# static BigInteger fibonacci\_3(int n)

---

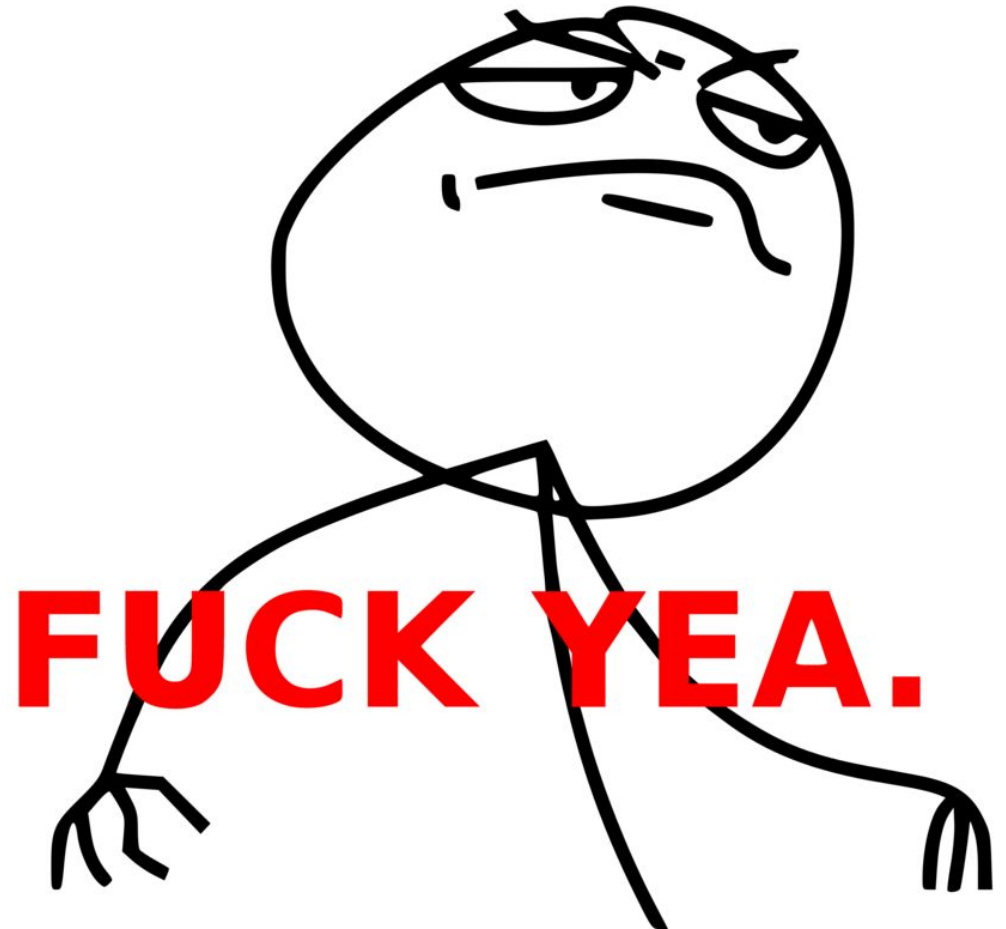
```
1.  static Map<Integer, BigInteger> cache_3 = new HashMap<>();
2.
3.  static BigInteger fibonacci_3(int n) {
4.      if (cache_3.containsKey(n))
5.          return cache_3.get(n);
6.
7.      System.out.println("computing " + n);
8.      BigInteger result;
9.
10.     if (n <= 2)
11.         result = BigInteger.ONE;
12.     else
13.         result = fibonacci_3(n - 1).add(fibonacci_3(n - 2));
14.
15.     cache_3.put(n, result);
16.     return result;
17. }
```

# Let's make fibonacci great again!

----

```
100  
computing 100  
computing 99  
computing 98  
computing 97  
computing 96  
computing 95
```

```
...  
computing 5  
computing 4  
computing 3  
computing 2  
computing 1  
354224848179261915075
```



# GO BIG!

---

1000

computing 1000

computing 999

computing 998

computing 997

computing 996

computing 995

...

computing 5

computing 4

computing 3

computing 2

computing 1

43466557686937456435688527675040625802564660517371780402481729089536  
55541794905189040387984007925516929592259308032263477520968962323987  
33224711616429964409065331879382989696499285160037044761377951668492  
28875





**SONO  
VERAMENTE**

**EUFORICO!**



# GO BIIIIIIIIIIIIIIIG!

---

```
10000
computing 10000
computing 9999
computing 9998
computing 9997
computing 9996
computing 9995
...
computing 4471
```

```
Exception in thread "main"
  java.lang.StackOverflowError
  ...
  at com.company.Main.fibonacci_3
```

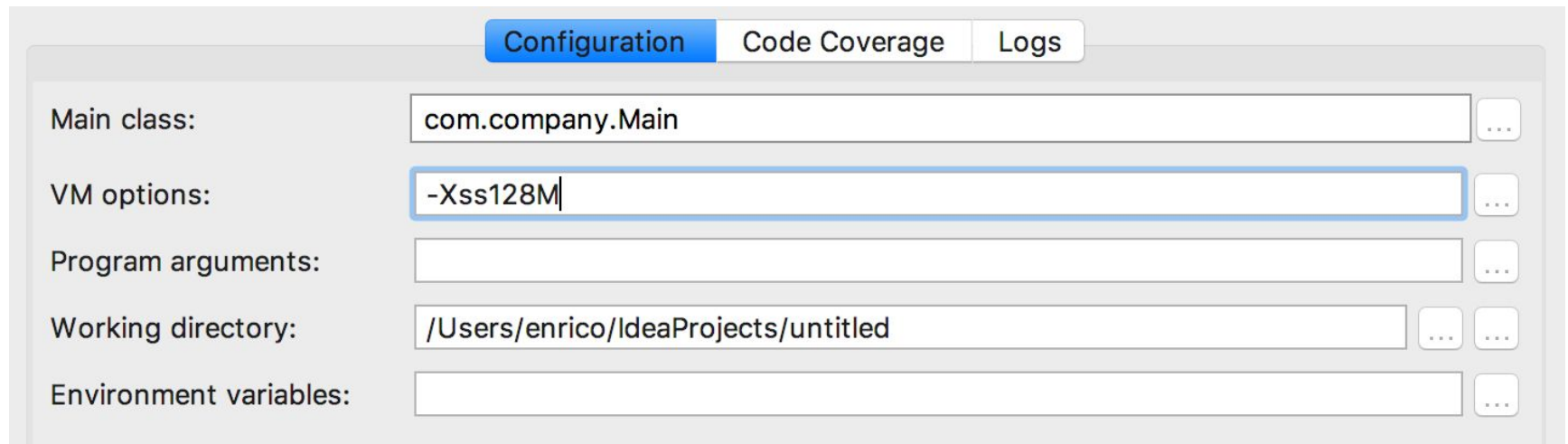


**ONE DOES NOT SIMPLY**

**ACCEPT A STACK OVERFLOW EXCEPTION**

# Stack Overflow? Let's increase the Stack Size!

---



The image shows a configuration window with three tabs: "Configuration" (selected), "Code Coverage", and "Logs". The "Configuration" tab contains several fields:

- Main class:** `com.company.Main`
- VM options:** `-Xss128M` (highlighted with a blue border)
- Program arguments:** (empty)
- Working directory:** `/Users/enrico/IdeaProjects/untitled`
- Environment variables:** (empty)

Each field has a text input box and a small "..." button to its right.

just leave a bit of memory for the system,

But don't fear to go to gigabytes!

GO B|||||G!111

---

10000

computing 10000

computing 9999

computing 9998

computing 9997

computing 9996

computing 9995

...

computing 5

computing 4

computing 3

computing 2

computing 1

# GO GGG!!111

---

3364476487643178326662161200510754331030214846068006390656476997468008144216666236815559551  
3633734025582065332680836159373734790483865268263040892463056431887354544369559827491606602  
0998841839338646527313000888302692356736131351175792974378544137521305205043477016022647583  
1890652789085515436615958298727968298751063120057542878345321551510387081829896979161312785  
6265033195487140214287532698187962046936097879900350962302291026368131493195275630227837628  
4415403605844025721143349611800230912082870460889239623288354615057765832712525460935911282  
0392528539343462090424524892940390170623388899108584106518317336043747073790855263176432573  
3993712871937587746897479926305837065742830161637408969178426378624212835258112820516370298  
0893320999057079200643674262023897831114700540749984592503606335609338838319233867830561364  
3535189213327973290813373264265263398976392272340788292817795358057099369104917547080893184  
1056146322338217465637321248226383092103297701648054726243842374862411453093812206564914032  
7510866433945175121615265453613331113140424368548051067658434935238369596534280717687753283  
4823434555736671973139274627362910821067928078471803532913117677892465908993863545932789452  
3777674406192240337638674004021330343297496902028328145933418826817683893072003634795623117  
1031012919531697946076327375892535307725523759437884345040677155557790564504430166401194625  
8097221672975861502696844314695203461493229110597067624326851599283470989128470674086200858  
7135016260312071903172086094081298321581077282076353186624611278245537208532365305775956430  
0725177443150515396009051686032203491632226408852488524331580515348496224348482993809050704  
8348244932745373262456775587908918719080366205800959474315005240253270974699531877072437682  
5907419939632265984147498193609285223945039707165443156421328157688908058783183404917434556  
2705202235648464951961124602683139709750693826487066132645076650746115126775227486215986425  
3071129844118262266105716351506926002986170494542504749137811515413994155067125627119713325  
2763631939606902895650288268608362241082050562430701794976171121233066073310059947366875



**IT'S OVER 9000!!!**

# static BigInteger fibonacci\_4(int n)

---

```
1.  static Map<Integer, BigInteger> cache_4 = new HashMap<>();
2.
3.  static BigInteger fibonacci_4(int n) {
4.
5.      Function<Integer, BigInteger> fn = new Function<>() {
6.          @Override
7.          public BigInteger apply(Integer x) {
8.              if (x <= 2)
9.                  return BigInteger.ONE;
10.             else
11.                 return fibonacci_4(x - 1).add(fibonacci_4(x - 2));
12.             }
13.         };
14.
15.     return cache_4.computeIfAbsent(n, fn);
16. }
```



# aaaand it's done !

of course one should not use recursion for Fibonacci, but a loop... anyway...

```
1.  static BigInteger fibonacci_5(int n) {  
2.      BigInteger a = BigInteger.ONE, b = BigInteger.ONE, next;  
3.      for (int i = 2; i < n; i++) { next = a.add(b); a = b; b = next; }  
4.      return b;  
5.  }
```

# What about libraries?

---

- The first rule of Java Club is: **use maven**
- The second rule of Java Club is: **USE MAVEN**

Useful Java Libraries:

- **Fastutil** <https://github.com/vigna/fastutil>
- **ND4J** <http://nd4j.org/>
- **Guava** <https://github.com/google/guava>

**AUTOMATE**



**ALL THE THINGS**

# How do I learn?

---



<https://hackerrank.com>



[seclab.unibg.it](http://seclab.unibg.it)

one more thing ...



# Hash Code

## Pizza

Practice Problem for Hash Code 2017



## Team score

Data Set	Best submission
Big	N/A
Example	N/A
Medium	N/A
Small	N/A
<b>Overall score</b>	<b>0</b>

## New submission

The round is in progress. You can make a new submission.

[START A NEW SUBMISSION](#)

# Unibg Seclab - Practice problem internal competition

---

The team that submits the highest scores for the practice problem gets free pizza during the competition.





[seclab.unibg.it](http://seclab.unibg.it)