# UP ACM

## Solutions to Algolympics 2015 Problems

University of the Philippines
Association for Computing Machinery

ALGOLYMPICS 2015

# Contents

**Introduction** 3

**Problem A: All About The Base** 6
- A.1 Problem . . . 6
- A.2 Slow solution . . . 6
- A.3 Reduction to Pell's equation . . . 6
- A.4 Solution to Pell's equation $x^2 - ny^2 = 1$ . . . 7
- A.5 Enumeration of the solutions . . . 9
- A.6 A logarithmic-time method: matrix exponentiation . . . 11
- A.7 A pure linear recurrence . . . 12
- A.8 Summary . . . 13

**Problem B: Make Gawa This Program** 14
- B.1 Problem . . . 14
- B.2 Solution . . . 14
- B.3 Tips for Efficiency . . . 14
  - B.3.1 Setter's Code . . . 15
  - B.3.2 Discussion of the Code . . . 18
  - B.3.3 Generating Your Own Input . . . 18
  - B.3.4 Common Mistakes . . . 19
- B.4 Summary . . . 20

**Problem C: Rigid Trusses** 21
- C.1 Problem . . . 21
- C.2 Introduction . . . 21
- C.3 Rigidity of a truss . . . 22
- C.4 Graph connectivity . . . 23
- C.5 Summary . . . 25

**Appendices** 26
- C.A Proof of Claim C.5 . . . 26

**Problem D: Slicing Number Cakes** 28

D.1 Problem . . . . . . . . 28
D.2 Backtracking . . . . . . . . 28
D.3 Dynamic programming . . . . . . . . 28
D.4 Greedy approach . . . . . . . . 29
D.5 Even faster approach: suffix arrays and suffix trees . . . . . . . . 29
D.6 Summary . . . . . . . . 30

**Appendices** **31**
D.A Proof of Claim D.7 . . . . . . . . 31

**Problem E: $N$-fruit combo** **34**
E.1 Problem . . . . . . . . 34
E.2 Solution . . . . . . . . 34
E.3 Convex hull . . . . . . . . 35
E.4 Summary . . . . . . . . 36

**Problem F: Alien Defense Deux** **37**
F.1 Problem . . . . . . . . 37
F.2 Gotcha . . . . . . . . 37
F.3 Slow solution . . . . . . . . 37
F.4 Faster but still slow solution: Preprocessing . . . . . . . . 38
F.5 Fast Solution: Binary Search . . . . . . . . 38
F.6 Faster Solution: Dynamic programming . . . . . . . . 39
F.7 Summary . . . . . . . . 39

**Problem G: For Science™!** **40**
G.1 Problem . . . . . . . . 40
G.2 Naïve solution . . . . . . . . 40
G.3 Fast solution . . . . . . . . 40
G.4 Summary . . . . . . . . 41

**Problem H: Algols for Algolympia** **42**
H.1 Problem . . . . . . . . 42
H.2 Slow solutions . . . . . . . . 42
H.3 Polynomial multiplication . . . . . . . . 43
H.4 Semi-fast solution: Karatsuba multiplication . . . . . . . . 43
H.5 Fast solution: Fast Fourier Transform . . . . . . . . 45
H.5.1 Inverse DFT . . . . . . . . 46
H.5.2 Other caveats . . . . . . . . 48
H.6 Summary . . . . . . . . 49

University of the Philippines
Association for Computing Machinery

ALGOLYMPICS 2015

# Introduction

This document contains the solutions to the problems used in Algolympics 2015.

To be able to understand many parts of this document, the reader should be acquainted with the big-O notation[1], and be familiar with time and space complexity of algorithms[2]. Also, we assume that the reader is familiar about basic data structures such as linked lists, stacks, binary trees, etc., and basic algorithms such as fast sorting, binary search, BFS/DFS, etc. If you are not familiar with these, please consult with your coach/professor.[3]

By the way, the code displayed in this document does not use any particular programming language. Instead, it is something called *pseudocode*[4]. The reader is expected to be able to translate pseudocode into their favorite programming language.

Also, the authors of this document are humans, so there are likely to be mistakes and errors. We hope you can figure out the mistakes yourself, and possibly inform us about it. But feel free to ask us if anything is unclear. You may contact one of the authors by sending an email to `kevin.charles.atienza@gmail.com`. Please don't be shy about asking. We're friendly, and happy to help out. :)

We encourage all the participants to read through all of these (even those they have already solved), because we think that they will get something useful out of it.

We hope that by providing these hints and solutions, future participants in Algolympics and other competitive programming contests will have a better chance to score higher next time around, and have a more satisfying/fulfilling experience in the contest.

# Misconceptions

One of the most common mistakes that beginning participants made during the contest is thinking that their programs will be tested manually by the judges, perhaps by running them and trying out a few test cases by hand to see if the program is working/correct. This thinking usually leads to the following wrong assumptions:

- That the participants may add a few "UI Enhancements" to make the job easier for the judges,

[1] Refer to articles on Big-O notation, such as http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o.

[2] More information about time complexity here http://en.wikipedia.org/wiki/Time_complexity. See also http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=complexity1 for a nice introduction.

[3] You can also easily find tutorials for these concepts online by searching with these keywords.

[4] Pseudocode is explained in http://whatis.techtarget.com/definition/pseudocode and also in http://en.wikipedia.org/wiki/Pseudocode

for example printing “Please enter the number of test cases:”, “Please enter five numbers in a single line, separated by spaces:”, or handling inputs in case they go out of bounds specified in the problem.

- That if the participants tested their program by hand using a couple of test cases and it seems to be correct, but the judges reject it, then the mistake must be with the judges.

In reality, the program is not run manually by the judges, but instead fed with a hidden input file prepared before the contest.[5] The output is then compared to the output file prepared by the judge, and it is accepted only if they match exactly. Any mistake, whether major or minor,[6] is not tolerated, e.g. a missing extra line at the end, an extra space at the end of the line, or output such as “Please enter the number of test cases:”, etc. The only time a submission is accepted by the judges is when the program runs within the time limit and without errors, and the output matches the output file exactly, character per character.

This means you have to follow exactly what the output format says. If it says print all numbers in a single line separated by single spaces, then don’t separate them by double spaces, commas, or put them in separate lines. Also, don’t add leading or trailing spaces, and don’t add printouts such as “The output for test case 5 is”, etc. The sample input/output is there for you to test your output format.[7]

Also, one of the mistakes made with Java submissions is that of adding a *package*. Your program should be put into the default package, since your program is compiled with the command `javac [basename].java` and is run with the command `java [basename]`, so if you added a package in your Java program then it will likely get the “Runtime error” verdict.

Also, when constraints (such as $1 \leq N \leq 1000$) are given in the problem statement, then these are *guaranteed* to be true, so you don’t have to check in your program whether they are true. A lot of contestants used up some of their time adding these checks, where they could have used it to ensure that their program is correct. Also, when constraints like these are given, be assured that the input file contains test cases that are at the extreme ends of these bounds. For example, if $1 \leq N \leq 150{,}000$, then you can be sure that many test cases will have $N \approx 150{,}000$. This means that if your solution runs in $O(N^3)$ or $O(N^2)$ time, then it will likely get the verdict “Time limit exceeded”.

[5] But this does not mean that your program should explicitly take its input from a file! You should still take your input from `stdin`, and you should still print your output to `stdout`. The judge runs your program by first compiling it and then running a command like `[yourprogram] < [judgeinput] > [youroutput]`, which redirects standard input to come from the file `[judgeinput]` and standard output to go to the file `[youroutput]`, and finally, comparing `[youroutput]` with `[judgeoutput]`. More about redirection (< and >) in http://ss64.com/nt/syntax-redirection.html, and more about the standard streams (`stdin` and `stdout`) in http://en.wikipedia.org/wiki/Standard_streams

[6] This might seem a bit harsh, but the truth is that it’s better that you are trained now, so that when you join more high-stakes contests such as an ACM-ICPC Regionals, then at least you are less likely to make these mistakes. Following specifications strictly is also very helpful in the future as a software developer when implementing, for example, protocols or file formats.

[7] Also, look out for statements such as “Print an extra line at the end of each test case” or “Separate the output for each test case by a blank line”. These two may look the same, but there is a subtle difference between them!

Also, take note that the input file is prepared specially to include tricky edge cases and a bunch of other things that the contestants might have overlooked. Common mistakes include: integer overflow[8], stack overflow[9], and failing to handle extreme cases/edge cases[10].

Note that no credit is given for “user-friendliness”, “neatness of code”, “modularity”,[11] etc., so don’t focus on these when participating in competitive programming contests[12] such as the Algolympics. Instead, focus on correctness and speed of execution.

[8] See an explanation here: http://www.tech-faq.com/integer-overflow.html

[9] An explanation of *stack overflow* can be found (in stackoverflow.com :D) here http://stackoverflow.com/questions/214741/what-is-a-stack-overflow-error

[10] Edge cases are explained here: http://en.wikipedia.org/wiki/Edge_case

[11] Although modularity somewhat helps to ensure correctness, especially for tricky coding problems such as “Make Gawa This Program”, you should not overdo it.

[12] Read more about competitive programming contests, including some common online contest sites, in http://en.wikipedia.org/wiki/Competitive_programming

# Problem A: All About The Base

**Problem Author:** Jared Guissmo E. Asuncion

**Keywords:** Pell's equation, Precomputation, Modular arithmetic

## A.1 Problem

The problem can be roughly stated as follows: given $d \leq 10^6$, find the $d$th integer $k$ such that the $k$th positive triangle number is a perfect square. The $k$th positive triangle number is $1+2+\cdots+k$, or equivalently $\frac{k(k+1)}{2}$.

## A.2 Slow solution

First, one can try to brute-force the problem: just test every triangle number in turn, and check whether it is a perfect square. However, one will quickly notice that this is infeasible, because square triangular numbers are increasingly rare as our number increases. To illustrate this, note that the 69th solution is the following 53-digit number:

$$1663353610806550606622589363307838843219947442726 6449$$

and the 100,000th answer has 76,555 digits! The one-millionth answer has even more.

This means that we need to find a faster algorithm to compute the answer.

## A.3 Reduction to Pell's equation

Since we found out that brute-force is infeasible, we will try to seek a faster solution. Note that a positive integer $a$ is a solution if only if $\frac{a(a+1)}{2} = b^2$ for some positive integer $b$. Let's try to play around with this:

$$\begin{aligned}
\frac{a(a+1)}{2} &= b^2 \\
4a(a+1) &= 8b^2 \\
4a^2+4a &= 8b^2 \\
4a^2+4a+1 &= 8b^2+1 \\
(2a+1)^2 &= 8b^2+1 \\
(2a+1)^2-8b^2 &= 1 \\
(2a+1)^2-2(2b)^2 &= 1
\end{aligned}$$

Let $x = 2a+1$ and $y = 2b$. Then we are seeking pairs of solutions $(x, y)$ such that $x > 0$ is odd, $y > 0$ is even, and:

$$x^2 - 2y^2 = 1$$

This is a special, well-known kind of equation called **Pell's equation**. Its general form is $x^2 - ny^2 = 1$ (our case is $n = 2$). The solution to this kind of equation is well-known, and we'll describe it here. But first, we note that the conditions "$x$ is odd" and "$y$ is even" is redundant in the case $n = 2$, because all $(x, y)$ solutions to this equation satisfy them. Here's a proof:

*Proof.* Given $x^2 - 2y^2 = 1$, we want to show that $x$ is odd and $y$ is even. The first is easy to show: $x^2 = 2y^2 + 1$, hence $x^2$ is odd.[13] Thus, $x$ is also odd (recall that a number is odd if and only if its square is).

Next, we show that $y$ is even. First, we manipulate $x^2 - 2y^2 = 1$ a bit:

$$\begin{aligned}
2y^2 &= x^2 - 1 \\
2y^2 &= (x-1)(x+1)
\end{aligned}$$

$x$ is odd, so $(x-1)$ and $(x+1)$ are both even. Thus, their product, which is equal to $2y^2$, is a multiple of 4. Now if $y$ were odd, then $2y^2$ would not be a multiple of 4, so it must be that $y$ is even. □

## A.4 Solution to Pell's equation $x^2 - ny^2 = 1$

Let's solve the general Pell's equation $x^2 - ny^2 = 1$. In order not to venture too far, we will only restrict our solution to the case where $n$ is positive and is not a perfect square.[14] Thus, we may assume that $\sqrt{n}$ is an irrational real number.

[13] Recall that $k$ is odd if it is of the form $2m + 1$.

[14] When $n$ is a perfect square or negative, the solution is easy. We invite the reader to try to solve those cases!

First, consider some solution $(x, y)$ (where $x$ and $y$ are not necessarily positive). Then we have the following:

$$\begin{aligned} x^2 - ny^2 &= 1 \\ (x + y\sqrt{n})(x - y\sqrt{n}) &= 1 \end{aligned}$$

We can associate every solution pair $(x, y)$ with the real number $x + y\sqrt{n}$. Thus, it makes sense to study the following set of numbers (which correspond to solutions to the Pell's equation):

$$\{x + y\sqrt{n} : x, y \in \mathbb{Z}, x^2 - ny^2 = 1\}$$

Let's denote this set by $\mathcal{P}_n$.[15] First, we note the following nice properties of $\mathcal{P}_n$:

1. 1 is an element of $\mathcal{P}_n$.
2. If $r$ is an element of $\mathcal{P}_n$, then $1/r$ is also in $\mathcal{P}_n$.[16]
3. If $r$ and $s$ are elements of $\mathcal{P}_n$, then $rs$ is also an element of $\mathcal{P}_n$.[17]

Thus, we may regard the set $\mathcal{P}_n$ as being a number system on its own, just like $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ or $\mathbb{C}$, with a slightly different set of axioms.[18] Moreover, since we can view $\mathcal{P}_n$ as a subset of the real numbers $\mathbb{R}$, we can put the elements of $\mathcal{P}_n$ in a total order $\leq$.[19]

Now, property 3 is remarkable. It says that products of "solutions" to Pell's equation also yield solutions. For example, in the case $n = 2$, the pairs $(3, 2)$ and $(99, 70)$ are solutions. Thus, we can find a new solution by multiplying $3 + 2\sqrt{2}$ and $99 + 70\sqrt{2}$:

$$\begin{aligned} g(3 + 2\sqrt{2})(99 + 70\sqrt{2}) &= 297 + (2 \cdot 99 + 3 \cdot 70)\sqrt{2} + 140(2) \\ &= 297 + 280 + (198 + 210)\sqrt{2} \\ &= 577 + 408\sqrt{2} \end{aligned}$$

Thus, we have found a new solution pair $(577, 408)$![20] In fact, the pairs don't have to be different: multiplying $3 + 2\sqrt{2}$ with itself gives the solution pair $(17, 12)$.

Now, before we try to characterize the elements of $\mathcal{P}_n$ in a way that is suitable for enumeration, it makes sense to first try to find the "smallest" solution. But we have to define "smallest" unambiguously. We say that $r$ is the **smallest solution** if $r > 1$ and there is no other solution $s$ such

[15] The more standard name for this set is the *set of units of* $\mathbb{Z}[\sqrt{n}]$ which is commonly denoted as $\mathbb{Z}[\sqrt{n}]^\times$, but let's try something different here :)

[16] Hint: show that the reciprocal of $r = x + y\sqrt{n}$ is $x - y\sqrt{n}$, which is a member of $\mathcal{P}_n$.

[17] Hint: show that if $r = x_1 + y_1\sqrt{n}$ and $s = x_2 + y_2\sqrt{n}$, then $rs$ is also of the form $x + y\sqrt{n}$ for some integers $x$ and $y$ with $x^2 - ny^2 = 1$, thus it is a member of $\mathcal{P}_n$.

[18] In standard terms, the three properties described, including the fact that multiplication of nonzero reals are commutative, associative and invertible, simply say that $\mathcal{P}_n$ forms an **abelian group** over real-number multiplication.

[19] A relation $\leq$ on a set $S$ is a **total order** if the following is true for all $a$, $b$, $c$ in $S$: (1) $a \leq b$ and $b \leq a$ imply $a = b$, (2) $a \leq b$ and $b \leq c$ imply $a \leq c$, and (3) $a \leq b$ or $b \leq a$.

[20] You can check that this works by plugging them to the Pell's equation.

that $1 < s < r$.[21]

Now, the smallest solution $r$ is remarkable, because it is possible to show that all solutions are powers of $r$ (including negative powers), and their negatives! We will prove it here:

**Claim A.1.** *Let $r$ be the* smallest solution *in $\mathcal{P}_n$. Then any other member $s$ of $\mathcal{P}_n$ is of the form $\pm r^k$ for some integer $k$.*

*Proof.* The cases $s = \pm 1$ are obviously of the form $\pm r^k$ (for $k = 0$), so assume that $s$ is not 1 or $-1$.

First, we handle the case that $s > 1$. Since $r > 1$, then the sequence $(1, r, r^2, r^3, \ldots)$ goes to infinity, and will eventually exceed $s$. Let $m$ be the point where it first exceeds $s$, i.e. $r^m \leq s < r^{m+1}$. Divide this by $r^m$ to get $1 \leq s/r^m < r$. Now, consider the number $s/r^m$. It is also a member of $\mathcal{P}_n$. Now, assume that it is not 1. Then, it is greater than 1 but smaller than $r$, which contradicts the minimality of $r$. Therefore, $s/r^m$ must be 1, and $s = r^m$. So all $s > 1$ must be of the form $r^k$ for some $k$.

Next, we handle the case $0 < s < 1$. Note that $1/s > 1$, so $1/s = r^m$ for some $m$ (as we have just shown above). Therefore, $s = r^{-m}$, which means that $s$ is also of the form $r^k$ for some $k$.

Finally, we handle the case $s < 0$. Note that $-s > 0$, and we have just shown above that all positive solutions are powers of $r^k$. Therefore, $-s = r^m$ for some $m$, and $s = -r^m$. Thus, $s$ is also of the form $-r^k$ for some $k$. □

This claim is remarkable, since it describes all the elements of $\mathcal{P}_n$ completely: All elements of $\mathcal{P}_n$ can be generated from just the smallest solution $r$ alone!

Finally, we note here without proof that for any $n$, a nontrivial solution always exists, but we won't describe a general method of finding the smallest solution. Instead, we now return to the case $n = 2$, and ask the reader to verify that the solution $3 + 2\sqrt{2}$ is indeed the smallest solution (there are only a few cases to check!).

## A.5 Enumeration of the solutions

Now that we know that all solutions of $x^2 - 2y^2 = 1$ can be generated from the powers of $r = 3 + 2\sqrt{2}$, we can now turn this into an efficient program to enumerate all the solutions of $x^2 - 2y^2 = 1$. We only need positive solutions, so we exclude all negative powers of $r$, and also all the negative ones, i.e. $-r^k$. Thus, the $k$th solution ($k \geq 0$) is $(x_k, y_k)$, where $x_k + y_k\sqrt{2} = (3 + 2\sqrt{2})^k$. We start with

[21] Note that such a definition isn't necessarily well defined. For example, the set of all positive rationals doesn't have a minimum. But one can show that the *smallest solution* is well defined by showing that for any real number $x > 1$, there are only a finite number of elements of $\mathcal{P}_n$ that are $> 1$ and $< x$.

the solution $x_0 + y_0\sqrt{2} = 1 + 0\sqrt{2}$, and describe a way to get the next solution $x_{i+1} + y_{i+1}\sqrt{2}$ from the current solution $x_i + y_i\sqrt{2}$:

$$\begin{aligned} x_{i+1} + y_{i+1}\sqrt{2} &= (x_i + y_i\sqrt{2})(3 + 2\sqrt{2}) \\ &= 3x_i + (3y_i + 2x_i)\sqrt{2} + 2y_i(2) \\ &= (3x_i + 4y_i) + (2x_i + 3y_i)\sqrt{2} \end{aligned}$$

Thus, we have the following recursive relationships:

$$\begin{aligned} x_{i+1} &= 3x_i + 4y_i \\ y_{i+1} &= 2x_i + 3y_i \end{aligned}$$

with base case $(x_0, y_0) = (1, 0)$.

As a final note for this section, remember that we are looking for the indices of the triangular numbers $a_i$s, where $x_i = 2a_i + 1$ (and we might as well substitute $y_i = 2b_i$). Therefore, we can restate the above formula as:

$$\begin{aligned} a_{i+1} &= 3a_i + 4b_i + 1 \\ b_{i+1} &= 2a_i + 3b_i + 1 \end{aligned}$$

with base case $(a_0, b_0) = (0, 0)$.

You can verify that this works:

$$\begin{aligned} (a_1, b_1) &= (3(0) + 4(0) + 1, 2(0) + 3(0) + 1) = (1, 1) \\ (a_2, b_2) &= (3(1) + 4(1) + 1, 2(1) + 3(1) + 1) = (8, 6) \\ (a_3, b_3) &= (3(8) + 4(6) + 1, 2(8) + 3(6) + 1) = (49, 35) \\ (a_4, b_4) &= (3(49) + 4(35) + 1, 2(49) + 3(35) + 1) = (288, 204) \\ (a_5, b_5) &= (3(288) + 4(204) + 1, 2(288) + 3(204) + 1) = (1681, 1189) \\ (a_6, b_6) &= (3(1681) + 4(1189) + 1, 2(1681) + 3(1189) + 1) = (9800, 6930) \\ (a_7, b_7) &= (3(9800) + 4(6930) + 1, 2(9800) + 3(6930) + 1) = (57121, 40391) \\ (a_8, b_8) &= (3(57121) + 4(40391) + 1, 2(57121) + 3(40391) + 1) = (332928, 235416) \\ (a_9, b_9) &= (3(332928) + 4(235416) + 1, 2(332928) + 3(235416) + 1) = (1940449, 1372105) \\ (a_{10}, b_{10}) &= (3(1940449) + 4(1372105) + 1, 2(1940449) + 3(1372105) + 1) = (11309768, 7997214) \\ (a_{11}, b_{11}) &= (3(11309768) + 4(7997214) + 1, 2(11309768) + 3(7997214) + 1) = (65918161, 46611179) \\ &\dots \end{aligned}$$

Note that the problem asks for the solution modulo $10^7 + 6699$, so you have to do all computations modulo $10^7 + 6699$ (strictly speaking this is not necessary, but since the answers grow large very fast, this should be done to avoid time and memory limit verdicts).

Also, note that this solution is much more feasible than the brute-force solution. For example, if you want the one-millionth solution modulo $10^7 + 6699$, then this approach will give you that really fast (like, *milliseconds* fast). However, if you try to give it a test file with a million queries, then you might find that your solution runs really slowly (it might take hours to fully process the million queries). It's easy to get around this though, just compute the first $10^6$ answers before processing the input, so each query is reduced to a simple lookup to the precomputed answers.

## A.6 A logarithmic-time method: matrix exponentiation

Note that the above solution is fast enough to get accepted, but here we will explore further. The good thing is that we will actually arrive at a solution that can handle queries $d \le 10^{18}$, where it is infeasible to calculate the answer in $O(d)$ time even for a single query.

Let's take a look at the recurrence for $a_i$ and $b_i$:

$$\begin{aligned} a_{i+1} &= 3a_i + 4b_i + 1 \\ b_{i+1} &= 2a_i + 3b_i + 1 \end{aligned}$$

Now, consider the following column vectors:

$$\begin{bmatrix} a_{i+1} \\ b_{i+1} \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} a_i \\ b_i \\ 1 \end{bmatrix}$$

Note that the first vector can be obtained by multiplying the second vector with the following matrix (from the left):

$$M = \begin{bmatrix} 3 & 4 & 1 \\ 2 & 3 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, we can obtain these vectors from an initial vector $\begin{bmatrix} a_0 \\ b_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ by $M$ repeatedly. In other words:

$$\begin{bmatrix} a_i \\ b_i \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 4 & 1 \\ 2 & 3 & 1 \\ 0 & 0 & 1 \end{bmatrix}^i \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

This follows because matrix multiplication is associative.

This means that we can calculate $a_d$ (and $b_d$) by calculating the matrix power $M^d$ and multiplying it to our initial vector. This can be done using fast exponentiation methods, which run in $O(\log d)$

time![22] (be careful to do all operations modulo $10^7+6699$, to prevent the numbers from exploding!) For $d$ even up to $10^{18}$, this still runs very fast! (Implement this yourself to verify!)

## A.7 A pure linear recurrence

This section doesn't yield any significant improvement in the running time, but it's nice nonetheless, and the idea in this section may be of use to the reader in other applications.

Since the $a_i$ are all that matter to us, we will try to eliminate the references to $b_i$ completely. From the first equality, we see that $b_i = \frac{a_{i+1}-3a_i-1}{4}$. Substituting this to the second, we get:

$$\begin{aligned}
b_{i+1} &= 2a_i + 3b_i + 1 \\
\frac{a_{i+2} - 3a_{i+1} - 1}{4} &= 2a_i + 3\frac{a_{i+1} - 3a_i - 1}{4} + 1 \\
a_{i+2} - 3a_{i+1} - 1 &= 8a_i + 3(a_{i+1} - 3a_i - 1) + 4 \\
a_{i+2} - 3a_{i+1} - 1 &= 8a_i + 3a_{i+1} - 9a_i - 3 + 4 \\
a_{i+2} - 3a_{i+1} - 1 &= 3a_{i+1} - a_i + 1 \\
a_{i+2} &= 6a_{i+1} - a_i + 2 \\
a_i &= 6a_{i-1} - a_{i-2} + 2
\end{aligned}$$

The last step simply adjusts the indices. Note that this is *almost* a pure linear recurrence, except for the $+2$ blemish at the end. But fortunately there is a way to eliminate it with a simple adjustment of indices:

$$\begin{aligned}
a_i &= 6a_{i-1} - a_{i-2} + 2 \\
a_{i-1} &= 6a_{i-2} - a_{i-3} + 2
\end{aligned}$$

Subtracting the two, we get:

$$\begin{aligned}
a_i - a_{i-1} &= 6a_{i-1} - 7a_{i-2} + a_{i-3} \\
a_i &= 7a_{i-1} - 7a_{i-2} + a_{i-3}
\end{aligned}$$

Now, at the cost of increasing the needed previous terms, we now have a pure linear recurrence involving only $a_i$.

This doesn't have any significant advantage over the previous technique, but it does give a slightly simpler form of matrix to exponentiate:

$$\begin{bmatrix} a_i \\ a_{i-1} \\ a_{i-2} \end{bmatrix} = \begin{bmatrix} 7 & -7 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{i-1} \\ a_{i-2} \\ a_{i-3} \end{bmatrix}$$

[22] See the link http://en.wikipedia.org/wiki/Exponentiation_by_squaring for more details about fast exponentiation.

But the initial vector should be $\begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} 8 \\ 1 \\ 0 \end{bmatrix}$.

## A.8 Summary

Here is the summary of the algorithm:

1. Define $a_0 = b_0 = 0$.
2. For $i$ from 0 to $10^6 - 1$, set $a_{i+1}$ and $b_{i+1}$ as:

$$\begin{aligned} a_{i+1} &= (3a_i + 4b_i + 1) \bmod (10^7 + 6699) \\ b_{i+1} &= (2a_i + 3b_i + 1) \bmod (10^7 + 6699) \end{aligned}$$

3. The answer for a given $d$ in the input should be $a_d$.

After an $O(L)$-time precomputation (assuming $L$ is the maximum input number; in our case $L = 10^6$), each query can be answered in $O(1)$ time!

# Problem B: Make Gawa This Program

**Problem Author:** Jared Guissmo E. Asuncion

**Keywords:** Ad hoc, string processing

## B.1 Problem

Translate a given sentence using the rules described in the problem statement. In other words, *Make salin the pangungusap.*

## B.2 Solution

Follow all instructions carefully and correctly. Nothing more, nothing less.

## B.3 Tips for Efficiency

There are no tricks to make your code significantly faster. This problem is not your code vs the limitations of the computer. This problem is more about your code vs you. For these types of problems, you have to organize your code such that it is readable and easily modifiable, in case you forget to address some parts of the problem (which you most likely will).

At first glance, this problem might look very easy as it actually boils down to reading the instructions and translating this into code. However, as most participants who attempted this problem would recall, it's easier said than done. These types of problems present a subtle challenge in a competitive setting: You must code fast enough so that you have time to solve other more problems but you must also organize your code to some extent just in case you need to modify it later. The second part is clearly as important as the first. Looking at the results of the contest, not one team got it correct during the first attempt. Hence, you have more reason to have decent-looking code to edit once you hear the bad news (of your code not being accepted).

You must also be familiar with the string functions of the programming language of your choice. These functions are not only used in these so-called *coding problems*, but some problems which are algorithmic in nature might even require some processing (see Rigid Trusses).

On subsection B.3.2, we will discuss some ways to organize your code. We will reference the code written on the subsection B.3.1. Take note that this is written in Java. After this, it is advised that you try implementing the problem again without looking at the code. A disclaimer before we begin: the names of the custom functions in this code are too long. The long names are used in order to be more descriptive. In a competition setting, it is not very advisable to have function names this long (unless you want to amuse yourself while coding).

On subsection B.3.3, we will discuss some ways to generate your own test cases. The sample input is not necessarily representative of all the input the judge's input might have. It is just there to illustrate simple cases of the problem. Moreover, generating your own test cases gives you an opportunity to see if your code works correctly and quickly. Making your own test cases is important since this might save from submitting obviously wrong solutions.

All in all, some may feel that this is just a mundane exercise that can be given during an introductory course in computer science. The challenge is to code it fast and get it right on the first try. In most programming contests, there are normally no points given for partially correct answers. Moreover, a problem as easy as this will surely be solved by most teams. Any wrong attempt on this problem will take a toll on your time penalty (that's 20 minutes per wrong submission)! Sometimes, this can greatly affect the rankings since we look at the time penalty if there's a tie in the number of problems solved correctly.

### B.3.1 Setter's Code

```java
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while(sc.hasNext()){
            String s = sc.nextLine();
            String[] str = s.split("␣");
            int len = str.length;
            String SUBJ = str[0];
            String VERB = str[len - 3];
            String NOUN = str[len - 1];
            if(str.length == 6){
                System.out.println(MakeFutureThisVerb(VERB) + "␣" +
                   MakeTagalogThisSubject(SUBJ) + "␣ang␣" + NOUN);
            }else if(str[1].charAt(2) == 'd'){
                System.out.println(MakePastThisVerb(VERB) + "␣" +
                   MakeTagalogThisSubject(SUBJ) + "␣ang␣" + NOUN);
            }else{
                System.out.println(MakePresentThisVerb(VERB) + "␣" +
                   MakeTagalogThisSubject(SUBJ) + "␣ang␣" + NOUN);
            }
        }
    }

    public static String MakeTagalogThisSubject(String SUBJ){
        if(SUBJ.equals("I")) return "ko";
        if(SUBJ.equals("You")) return "mo";
        if(SUBJ.equals("He")) return "niya";
        if(SUBJ.equals("She")) return "niya";
        if(SUBJ.equals("We")) return "natin";
        if(SUBJ.equals("They")) return "nila";
        return "INVALID";
    }

    public static String MakeUlitTheFirstSyllable(String WORD){
        int firstVowel = -1;
        for (int i = 0; i < WORD.length(); i++) {
            if(VowelBa(WORD.charAt(i))){
                firstVowel = i;
                break;
            }
        }
        return WORD.substring(0,firstVowel+1)+WORD;
    }

    public static String MakeFutureThisVerb(String WORD){
        WORD = MakeUlitTheFirstSyllable(WORD);
        int lastVowel = -1;
        for (int i = WORD.length()-1; i >= 0; i--) {
            if(VowelBa(WORD.charAt(i))){
                lastVowel = i;
                break;
            }
        }
        char theVowel = WORD.charAt(lastVowel);
        if(theVowel == ’o’){
            WORD = WORD.substring(0,lastVowel) + "u" + WORD.substring(
                lastVowel+1);
        }
        if(theVowel == ’O’){
            WORD = WORD.substring(0,lastVowel) + "U" + WORD.substring(
                lastVowel+1);
        }
        if(VowelBa(WORD.charAt(WORD.length()-1))){
            return Capitalize(WORD) + "hin";
        }else{
            return Capitalize(WORD) + "in";
        }
    }

    public static String Capitalize(String WORD){
        char firstLetter = WORD.charAt(0);
        if(’a’ <= firstLetter && firstLetter <= ’z’){
            firstLetter = (char)((int)firstLetter - (int)(’a’) + (int)(’A’));
        }
        return firstLetter + WORD.substring(1);
    }

    public static String MakePresentThisVerb(String WORD){
        WORD = MakeUlitTheFirstSyllable(WORD);
        WORD = AddInBeforeFirstVowel(WORD);
        return Capitalize(WORD);
    }

    public static String MakePastThisVerb(String WORD){
        WORD = AddInBeforeFirstVowel(WORD);
        return Capitalize(WORD);
    }
    public static String AddInBeforeFirstVowel(String WORD){
        int firstVowel = -1;
        for (int i = 0; i < WORD.length(); i++) {
            if(VowelBa(WORD.charAt(i))){
                firstVowel = i;
                break;
            }
        }
        return WORD.substring(0,firstVowel)+"in"+WORD.substring(firstVowel);
    }
    public static boolean VowelBa(char C){
        String VOWELS = "AEIOUaeiou";
        for (int i = 0; i < VOWELS.length(); i++) {
            if(C == VOWELS.charAt(i)){
                return true;
            }
        }
        return false;
    }
}
```

### B.3.2 Discussion of the Code

On line 11, we have a trick on how to get the verb. Take note that we start counting from the right. Why? Because the verb can either be in `str[3]` in the case where `"will make"` is used, but in `str[2]` otherwise. This might seem very trivial but you save some lines of code and some potential clutter. You would also feel clever for doing tricks like these. A trick similar of this nature also occurs in lines 13 - 19.

Line 23 defines a rather thoughtless function that just returns a string based on the input. This might seem unnecessary but keeping your code organized like this will definitely be helpful when trying to find typos, especially since this is a problem that relies on a lot of hard-coded strings.

Line 33 defines a function which is used both in the past case and in the future case. Chances are, if you're not using functions, then you're copy-pasting code. Using functions is better since if for example you don't get it right the first try, then you only have to modify it *once*. If you copy-pasted, you might forget to modify the copy-pasted code.

Line 70 shows an illustration of typecasting. Each character in a string can be represented by an integer based on its ASCII code. This converts `firstLetter` into an int so you can perform mathematical operations. This line simply takes the ASCII code of a lowercase character, subtracts the ASCII code of 'a' so that the result will be a code from 0 to 25, then adds the ASCII code of 'A'. Read on ASCII codes and typecasting if you do not understand this paragraph. Note that typecasting looks different in various programming languages. So make sure to check yours.

Line 95 is a simple boolean function which tells if the character given is a vowel or not. Instead of making a lot of if statements, a trick like this would be a very clever way to save time and also maintain code readability, just in case your first few attempts merit a wrong answer.

### B.3.3 Generating Your Own Input

Before submitting your code, make sure you have pushed the limits of your code based on the constraints given. On some problems, you could check if your code still processes boundary cases. Furthermore, on problems like these where one simple mistake could cost you twenty minutes of penalty, generating your own input to test your program is very crucial.

It would help to have a separate file of your team's personal test input on each program. Accumulate your test input as you go along. This problem encourages you to actually do this.

Based on how this particular problem is described, it will be very easy to actually generate some test cases. Think of the judge's input file to be the most sadistic abusive set of input that fits the constraint given in the problem. Don't assume that just because this problem looks like a simple translation program that testing normal words and sentences will do.

The first set of test cases you must have are the sample input. If your output does not match the sample output, then your solution is definitely wrong.

Your succeeding set of test cases should try to abuse the constraints. Does the problem statement say VERB has at least one vowel and nothing else? Then try a test case whose VERB is exactly one vowel! Does the problem statement have a special instruction if the last vowel is *o* or *u*? Then have a test case with those as the last vowel and (of course) cases such that those aren't the last vowel! The possibilities are endless! Well, not really endless since you have constraints. You have to make sure that your test cases satisfy the constraints of the problem. As a short exercise, identify which of the following are valid inputs to the problem. For the valid inputs, check which constraint (if any) it is trying to abuse.

```
The input is: She makes chorva the chenes.
He made chorva the chenes.
He make ChoRVa the cHenEs.
We makes chorva the chenes
i will make chorva the chenes.
They makes chorva the chenes!
They makes chorvs the chenes!
They makes chorvaloo the chenesloo!
They made ch0rva the chenes.
They made chorva the che nes.
She will make sunog the apoy!
She will make sunooooooooooog the apoy!
They will make pangako the money.
You will make o the x.
You will make x the o.
You will make a the r!
```

By no means does this account for all the "tricks" hidden in the judges input. See if you can find some more! You can also ask a teammate to give you more test cases for practice. Feel free to do this with all the other problems in this set.

A word of caution, however, do not spend all of your time trying to make test cases. You are here to score points and win, not to generate all possible test cases. That's the judge's problem!

### B.3.4 Common Mistakes

After looking at the actual solutions submitted during the contest, we have compiled a list of common mistakes. These include:

- Not taking all lines of input! It should have been clear from the sample I/O that there are multiple lines of input.

- Adding extraneous spaces/lines. In general, judges in algorithmic contests are strict, so you should not print anything aside from what is exactly specified in the problem.
- Being unable to replace the last 'o' or 'O' into a 'u' or 'U'.
- Replacing an 'o' with a 'U', or an 'O' with a 'u'.
- When the first letter of `VERB` is a vowel and you need to take the past tense, being unable to capitalize the 'In' prefix.
- Being unable to conjugate `VERB` correctly if it only contains one vowel, in particular if it consists of just a single vowel such as 'O'.
- Converting the whole `VERB`/`NOUN` into lowercase. You should preserve the capitalization given in the input unless specified otherwise... don't make any unwritten assumptions!
- Placing "in" instead of "hin" or vice versa (usually unknowingly by the contestant).

## B.4 Summary

Read the problem carefully, take note of the constraints, organize your code, address the boundary cases, make your own test cases, check if your solution makes sense and then submit your solution.

# Problem C: Rigid Trusses

**Problem Author:** Karl Ezra S. Pilario

**Keywords:** Ad hoc, Graph traversal, Graph connectivity

## C.1 Problem

Given an $R \times C$ rectangular truss, with some cells possibly having diagonal bars, determine whether it is rigid.

## C.2 Introduction

Before anything else, we note that all test cases are really valid. There are no tricky cases in the input such as the following:

<table>
<tr>
<td>1 3<br>+-+-+-+<br>|   | |<br>+-+-+-+</td>
<td>1 3<br>+-+-+-+-+<br>| | | |<br>+-+-+-+</td>
<td>1 4<br>+-+-+-+-+<br>| | | / |<br>+-+-+-+-+</td>
<td>1 4<br>+-+-+-+ +<br>|/|/|/|/|<br>+-+-+-+-+</td>
<td>1 3<br>+-+-+-+-+<br>|/|/|/| |<br>+-+-+-+-+</td>
</tr>
</table>

The challenge is merely on how to decide rigidity based on the placement and the number of diagonal bars. To determine a pattern, one must inevitably try to come up with one's own test cases on a scratch paper, and check. Here are more examples of trusses. Give a reason why they are rigid or non-rigid.

<table>
<tr>
<td>1 3<br>+-+-+-+<br>| |/|/|<br>+-+-+-+</td>
<td>2 2<br>+-+-+<br>|/| |<br>+-+-+<br>| |/|<br>+-+-+</td>
<td>2 2<br>+-+-+<br>| |/|<br>+-+-+<br>|/| |<br>+-+-+</td>
<td>2 2<br>+-+-+<br>|/|/|<br>+-+-+<br>| |/|<br>+-+-+</td>
<td>2 3<br>+-+-+-+<br>|/| |/|<br>+-+-+-+<br>| |/|/|<br>+-+-+-+</td>
</tr>
<tr>
<td>Case #1: 0</td>
<td>Case #2: 0</td>
<td>Case #3: 0</td>
<td>Case #4: 1</td>
<td>Case #5: 1</td>
</tr>
</table>

In a non-skewed truss, let's call the smallest square (whose edges are parallel to the x and y axes) in the lattice a **cell**. A cell can have at most 1 diagonal bar. Let's call a cell a **bolted** cell if it has a diagonal bar. A bolted cell will always remain a square no matter what you do to the truss. If a truss is non-rigid and if that truss is skewed, each cell can only either remain a square or become a rhombus.

## C.3 Rigidity of a truss

Let's define a cell that will remain a square no matter what you do to the truss a **rigid** cell. Note that bolted cells are rigid, but some non-bolted cells may also be rigid. Let's call those cells **implicitly rigid**. A rectangular truss is **rigid** if every cell is rigid.

When is a particular rectangular truss rigid? A simple idea that may come to mind is that a given truss is rigid if every row and column contains a bolted cell. This makes sense, right? Because if some row or column doesn't have any bolted cell, then we can shear that row/column, showing that the truss is not rigid.

Unfortunately, while this is a necessary condition, it is not sufficient. There are numerous examples above and even one on the problem statement that show it.

Here we describe a important properties of rigid grids and cells.

**Claim C.2.** *No matter what you do to a truss, the "horizontal bars" in each column always remain parallel to each other. The same goes with the "vertical bars" in each row.*

Claim C.2 follows from the fact that opposite sides of a rhombus are parallel. Note that we placed "horizontal" and "vertical" in quotes because they are not necessarily horizontal or vertical when one skews the truss. We define a bar to be **horizontal** if it is a shared edge of two cells from different rows, and **vertical** is defined similarly for columns.

Now, let's label each cell with its (row-number, column-number pair) $(i, j)$, where $1 \leq i \leq R$ and $1 \leq j \leq C$.

**Claim C.3.** *If $(i_1, j_1)$, $(i_2, j_1)$ and $(i_1, j_2)$ are rigid, then so is $(i_2, j_2)$.*

The claim can be easily seen by using claim C.2 repeatedly to deduce that the sides of cell $(i_2, j_2)$ must be perpendicular to each other.

Finally, here's another simple, but very useful claim:

**Claim C.4.** *Given any truss, one can "swap" the configuration of diagonal bars of any two columns without affecting the rigidity/non-rigidity of the truss. The same goes with any two rows.*

Let $T$ be a truss and $T'$ be a truss where some pair of columns (say $i$ and $j$) of $T$ are swapped. To show the claim, one needs to come up with a skewing of $T'$ for every skewing of $T$. This is easy once you realize that the "shape" of every connected set of vertical bars are all the same (because of claim C.2). Thus, given a skewing of $T$, one may always "swap" $i$ and $j$ in their current shapes and they will fit perfectly. A similar argument works for pairs of rows.

A corollary of claim C.4 is that one may rearrange the rows/columns without affecting the rigidity or non-rigidity of the truss.

We now have a new method of determining rigidity. That is: Use C.3 repeatedly to infer cells as (implicitly) rigid or not, until you can infer no more. Then the truss is rigid if and only if all cells are rigid.

Note that this method is better than the old "ensure there are bolted cells in every row and column" method, because it eliminates some false positives such as Case #2 above. But does it all eliminate *all* false positives? Amazingly, yes:

**Claim C.5.** *A truss is rigid if and only if all its cells can be shown rigid by starting with the bolted cells and then repeatedly applying claim C.3.*

One can try to convince oneself that this is true just by pure intuition, but for the truly pedantic, we will provide a proof in the appendices.

One can now transform these claims into an algorithm, and if implemented *correctly* it runs in $O(R^3C^3)$ time.[23] It *might* be possible to optimize your program to squeeze this algorithm into the time limit, but that is not recommended. Instead, we'll show an improved version of the solution that runs in the optimal $O(RC)$ time.

## C.4 Graph connectivity

The previous section described one way of thinking about the problem, and in fact already yields a viable solution to the problem. But there is another perspective that will yield a faster solution. The key is to understand how non-rigidity actually occurs.

Note that every rigid cell forces its row and column to always be *perpendicular* to each other. Furthermore, we see that this a sufficient condition for a cell to be rigid, i.e., a cell is rigid if and only if its row and column are forced to be perpendicular to each other. Thus, it follows that for a given row $i$ and two columns $j_1$ and $j_2$, if there is a rigid cell in $(i, j_1)$ and $(i, j_2)$, then the columns $j_1$ and $j_2$ are forced to be *parallel* to each other. Continuing with this, one can show using the rigid cells that certain sets of rows/columns are *parallel/perpendicular* to each other.

Now, non-rigidity occurs if there is nothing forcing a given row-column pair to be perpendicular to each other. We can model the *parallelity/perpendicularity* relationships of the rows and columns as a *bipartite graph.* We have a node for each row $r_i$ and for each column $c_j$, and we place an edge between $r_i$ and $c_j$ if the cell $(i, j)$ is bolted.

We can now use this representation for the following claim:

[23] We emphasize "correctly" because a naïve implementation will take much slower, possibly $O(R^4C^4)$

**Claim C.6.** *Cell $(i, j)$ is rigid if and only if nodes $r_i$ and $c_j$ are connected.*

The “if” part is easy to prove, by just following the path and determining that row $i$ and column $j$ are indeed perpendicular. The “only if” part can be seen intuitively, but rigorously follows from claim C.5.

As a corollary, we see that a given truss is rigid if and only if our graph is connected! Therefore, testing for rigidity can be reduced to a graph connectivity problem.

As an example, we have produced graph 1 from the given figure:

```
  1 2 3 4 5 6
1 +-+-+-+-+-+
  |/| |/| | |
2 +-+-+-+-+-+
  | | |/|/|/|
3 +-+-+-+-+-+
  | |/| | |/|
4 +-+-+-+-+-+
```

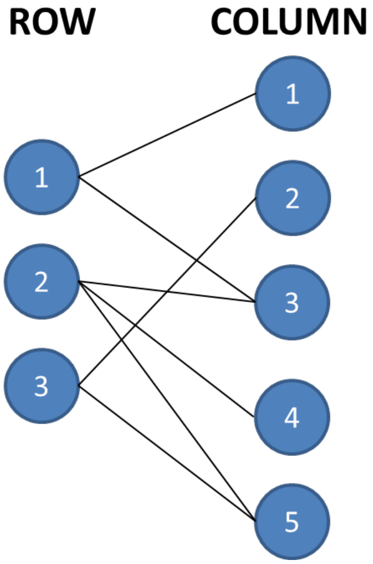

Figure 1: Bipartite Graph from Truss

Because there is a path from each node to every other node, the rotation of a rigid cell will force the rest of the cells to rotate as well. This makes it easy to see that a truss is rigid if its row-column graph is connected. The example below shows a non-rigid truss, whose row-column graph is 2.

```
  1 2 3 4 5 6
1 +-+-+-+-+-+
  |/| |/| |/|
2 +-+-+-+-+-+
  | | | |/| |
3 +-+-+-+-+-+
  |/|/| | |/|
4 +-+-+-+-+-+
```

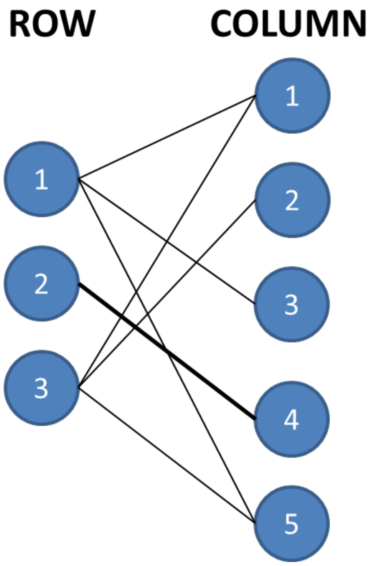

Figure 2: Bipartite Graph from Truss

Note that nodes $r_2$ and $c_4$ are in a different connectd component than the rest of the graph, which indicates that the cell on row 2, column 4 can rotate independently from the other nodes.

Having reduced the problem to a simple question of connectivity, we now have an algorithm that checks whether a given truss is rigid in $O(RC)$ time: use any standard search such as breadth-first search (BFS) or depth-first search (DFS)!

## C.5 Summary

Here is the summary of the algorithm:

1. Create a graph with $R + C$ nodes and labels $r_1, r_2, \ldots, r_R$ and $c_1, c_2, \ldots, c_C$, such that there is an edge between $r_i$ and $c_j$ if and only if there is a diagonal bar in cell $(i, j)$.

2. Use BFS/DFS to determine whether the graph is connected.
3. If the graph is connected, output 1, otherwise output 0.

This runs in $O(RC)$ time!

# Appendix

## C.A Proof of Claim C.5

In this appendix we prove claim C.5.

*Proof.* Suppose that we have a truss such that not all cells can be demonstrated rigid by repeatedly applying claim C.3. Thus, we only need to show that this truss is not rigid.

First, apply claim C.3 repeatedly, and place diagonal bars on all rigid cells, including the implicitly rigid ones. It can be easily seen that this does not affect the rigidity/non-rigidity of the truss—we are simply making implicit rigidity explicit. By our assumption, not all cells are bolted.

Let $(i, j)$ be any bolted cell (if there is no such cell, then the truss is automatically non-rigid). Next, collect all bolted cells which share the same column or row as $(i, j)$. Then collect all bolted cells sharing some row/column with any of those cells, and so on, until you can collect no more. Let's say all the collected cells form the set $S$.

Using claim C.4, rearrange the rows and columns so that the cells of $S$ occupy the leftmost and bottommost columns and rows. Thus, the resulting truss may look like the following:

```
+-+-+-+-+-+-+-+
| | | | |?|?|?|
+-+-+-+-+-+-+-+
| | | | |?|?|?|
+-+-+-+-+-+-+-+
|/|/|/|/| | | |
+-+-+-+-+-+-+-+
|/|/|/|/| | | |
+-+-+-+-+-+-+-+
|/|/|/|/| | | |
+-+-+-+-+-+-+-+
```

Note that we have placed '?'s in the upper-right portion of the truss because we don't know whether they are bolted or not, but we don't need to know that to proceed.

By assumption, the cells in $S$ do not span all rows and columns, thus there is at least one row or column not containing any cell in $S$. However, we know that the '?' portion of the truss is nonempty, otherwise there will be at least one row or column not containing any bolted cell and the truss is automatically non-rigid. Thus, there is at least one '?' cell.

Now, we do not know whether these '?' cells are rigid, but there's nothing wrong with assuming that they do (non-rigid cells can always pretend they are rigid). To finally show that the truss is not rigid, take the rectangle of the '?' cells collectively, and rotate them as one, without affecting the bolted cells at the lower-left. The blank cells in the upper-left and lower-right will go along with the rotation, similar to the rotation you do to show that the grid in Case #3 above is non-rigid. This means that the whole truss is not rigid, thus proving the claim. □

# Problem D: Slicing Number Cakes

**Problem Author:** Karl Ezra S. Pilario

**Keywords:** Backtracking, Dynamic programming, Greedy algorithms, Arbitrary-precision arithmetic, Suffix array, Suffix tree

## D.1 Problem

Given an integer cake with $d$ digits, you may slice this cake in $d-1$ positions between digits, forming a partition of the digits. The satisfaction that can be obtained from a partition is the sum of the resulting numbers (leading zeroes allowed). Now, given an integer $N$ and the number of slices $S$, find the maximum possible satisfaction that can be obtained by slicing $N$ exactly $S$ times.

## D.2 Backtracking

First, note that the number of digits of $N$ is small enough to allow for backtracking approaches to pass: Simply try out all possible partitions and find the maximum sum possible. The largest number of partitions one needs to generate is $\binom{18}{9} = 48620$.

## D.3 Dynamic programming

One can improve the backtracking approach by noticing that the best partition for a certain substring of the integer and a given number of slices does not depend on how the rest of the digits have been partitioned. This makes it amenable to *dynamic programming*.

Let us assume that $N$ has $n$ digits $d_{n-1}d_{n-2}\cdots d_1d_0$, and for $a \geq b$, define $N_{a,b}$ as the integer $d_ad_{a-1}\dots d_b$.

Next, let $B(i,j)$ be the best partition of the integer $N_{n-1,j}$ by doing exactly $i$ slices (for $j=n$, let $B(i,j)=0$ if $i=0$, and $-\infty$ otherwise). Note that the answer is simply $B(S,0)$.

Now, how does one calculate $B(i,j)$? Well, if $i=0$ (which means no slices), then the answer is trivially $N_{n-1,j}$, the integer itself. Now, what if $i>0$? Then we can try to select the last partition first. Let $k$ be the first digit of the last partition ($n > k \geq j$). Then the best partition in this case is the sum of $N_{k,j}$ and the best partition of the rest of the digits, $N_{n-1,k+1}$, with $i-1$ slices. But the latter quantity is simply $B(i-1,k+1)$ by definition. Therefore, $B(i,j)$ is simply the maximum $N_{k,j}+B(i-1,k+1)$ among all possible $k$ ($n \geq k \geq j$). Thus, $B$ has the following recursive relationship:

$$B(i,j) = \begin{cases} 0 & \text{if } j = n \text{ and } i = 0 \\ -\infty & \text{if } j = n \text{ and } i > 0 \\ N_{n-1,j} & \text{if } j < n \text{ and } i = 0 \\ \max\limits_{n>k\geq j} (N_{k,j} + B(i-1, k+1)) & \text{if } j < n \text{ and } i > 0 \end{cases}$$

If one stores the $B$ values in a table, where $0 \leq i \leq S$ and $0 \leq j \leq n$, and then build the table entry by entry, then each particular entry $B(i,j)$ in the table can be calculated in $O(n)$ time in the worst case. Therefore, this dynamic programming approach takes $O(n \cdot nS) = O(n^2 S)$ time,[24] where $n$ is the number of digits. This is better than the naïve backtracking approach whose running time is exponential in $n$.

## D.4 Greedy approach

Note that $O(n^2 S)$ is very efficient, but there is still a better way to calculate the answer! It relies on a certain property of the optimal partition.

First, let's define a **big piece** as a piece of the partition which has at least two digits. The single-digit pieces are called **small pieces**. Then we have the following claim:

**Claim D.7.** *There is an optimal partition with at most one big piece.*

This can be seen by some as intuitive, but a rigorous proof will be given in the appendices. The nice thing is that this claim gives us a faster algorithm, which is *greedy* in nature.

Since we can restrict ourselves to partitions with at most one big piece, it makes sense to try to make this big piece as large as possible. In fact, one can rigorously show that this is true. This is a simple proof which we'll leave to the reader.

Now, if we need to perform $S$ slices, then there are $S+1$ possible big pieces to check:

$$N_{n-1,S}, N_{n-2,S-1}, N_{n-3,S-2}, \ldots, N_{n-S-1,0}$$

The answer is thus the largest among these, plus the rest of the digits. If implemented correctly, this should run in $O(S)$ time, which is very fast!

## D.5 Even faster approach: suffix arrays and suffix trees

We have already provided feasible solutions above, but of course we are interested in even faster solutions than that. In particular, we will consider $N$'s that cannot fit in a 64-bit integer, so we represent $N$ as a string of digits, and also we can no longer assume that arithmetic can be done in $O(1)$ time. Removing this assumption, the greedy algorithm above actually runs in $O(nS)$ time, since we need $O(n)$ time to compare two large numbers, and we need $S$ comparisons to find the largest among $S+1$ strings. The final part of

[24] if implemented correctly, for example by precomputing the $N_{a,b}$'s, etc.

adding the remaining digits can be done in $O(n)$ time by first adding the digits themselves first to obtain a number that is $O(\log n)$ in size, and then adding it to our big piece with $O(n)$ digits. Addition can be done in linear time in the number of digits, so this just takes $O(n)$ time.

Thus, the bulk of the running time is in finding the largest subinteger with $n - S$ digits. Somewhat nicely, we can view this as a *string problem*: Given a string (of digits) of length $n$, find the lexicographically highest substring of length $n - S$. Such a problem can be solved using a well-known data structure called a suffix array.

A **suffix array** of a given string $N$ of length $n$ is simply all the $n + 1$ suffixes of $N$ in sorted order. Now, for compactness, a suffix array is usually represented as simply the list of $n + 1$ *indices* which point to the positions of each suffix, rather than the suffixes themselves. Using a suffix array, one can easily find the largest subinteger with $n - S$ digits in $O(n)$ time, by simply traversing the suffix array backwards and stopping at the first encountered suffix whose length is *at least* $n - S$. Then the substring we are looking for is simply the length $n - S$ prefix of this suffix.

Now, how does one construct the suffix array of the string $N$? The naïve way is to simply use a comparison sort on the suffixes. Unfortunately, comparison sorts always take $\Omega(n \log n)$ comparisons, and each comparison takes $O(n)$ time. Therefore, building the suffix array this way takes $O(n^2 \log n)$, which is bad.

Fortunately, there are standard fast ways of constructing the suffix array. The easiest to implement is called the **prefix-doubling algorithm**. A detailed explanation is given in the link `http://www.mi.fu-berlin.de/wiki/pub/ABI/RnaSeqP4/suffix-array.pdf`. It runs in $O(n \log^2 n)$ time if one uses a comparison sort at each pass, or $O(n \log n)$ if one uses specialized integer sorting methods such as bucket sort.[25] $O(n \log^2 n)$ is good—this means that we can solve the problem in less than a second even when there are up to 100,000 digits!

However, there are still faster ways to compute the suffix array. There is something called the **skew algorithm** that can construct the suffix array in $O(n)$ time! As a final note, there is a completely different $O(n)$ time algorithm to construct the suffix array, by first constructing something called the **suffix tree** in linear time (for example, using **Ukkonen's algorithm**), and then constructing the suffix array from the tree in $O(n)$ time.

## D.6 Summary

Here is the summary of the algorithm:

1. Let $n$ be the number of digits of $N$.
2. Find the largest contiguous subinteger of $N$ with length $n - S$.
3. The answer is that subinteger plus all the remaining digits of $N$.

This runs in $O(S)$ time, assuming constant-time arithmetic.

[25] However, implementing bucket sort for this is not recommended because it's easy to make mistakes, and one's implementation is not necessarily faster than a reasonably optimized comparison sort, as is the case with language-builtin sorts.

# Appendix

## D.A Proof of Claim D.7

In this appendix we prove claim D.7. We begin with a series of useful lemmas:

**Lemma D.1.** *The following statements are true:*

1. *If $N$ has no leading zeroes, then there is an optimal partition such that all big pieces do not have leading zeroes.*
2. *If $N$ has a leading zero, then there is an optimal partition such that at most one big piece has a leading zero, and that big piece, if it exists, necessarily contains the leftmost digit of $N$.*

*Proof.* Assume we have a partition of $N$, and let $V$ be the first big piece with a leading zero which also does not contain the leftmost digit of $N$ (if no such $V$ exists, then the lemma is satisfied and we are done). Let $V'$ be the big piece immediately before $V$ (if there are no other big pieces before $V$, let $V'$ be the small piece containing the leftmost digit). Then one can shorten $V$ by slicing its leading 0 off of it, and then lengthen $V'$ by merging the small slice immediately right of it (or the leading 0 from $V$ if they are adjacent). Note that by doing so, the number of slices stays the same, so the result is still a valid partition. Also, note that the value of $V$ does not decrease, but $V'$ possibly increases, thus the new partition is at least as good as our original partition. So if our original partition were optimal in the first place, then the resulting partition is optimal too.

We can repeat this process until all big pieces do not have leading zeroes (except possibly a big piece containing the leftmost digit of $N$). It should be easy to see that this process halts after a finite number of steps, and at that point we end up with an optimal partition satisfying the lemma. □

In the following lemma, we handle the case where $N$ has a leading zero.[26]

**Lemma D.2.** *If $N$ has a leading zero, and if there is an optimal partition such that the leftmost digit is part of a big piece, then at least one of the following cases holds:*

1. *There is an optimal partition with at most one big piece.*
2. *There is an optimal partition in which all big pieces do not have leading zeroes.*

*Proof.* Assume that we have an optimal partition of $N$ such that the leftmost digit is part of a big piece. Furthermore, using the proof of Lemma D.1, we transform this partition so that all the other big pieces do not have leading zeroes.

Now, we can lengthen the leftmost big piece by merging it to the piece immediately right of it, and then shorten it one digit left to detach the leading zero (i.e. *shift* the big piece one place to the right). Note that

[26] The problem guarantees that $N$ doesn't have leading zeroes, but we're handling it here for the sake of completeness.

this does not change the number of slices. However, since we only detached a zero, this cannot possibly decrease the sum. We repeat this process until one of the following happens:

1. **The leading digit of the big piece becomes nonzero.** At this point, we end up with an optimal partition in which all big pieces do not have leading zeroes, so case 2 holds.
2. **The big piece cannot shift anymore, i.e. it has reached the rightmost digit.** In this case, the big piece has traveled all the way across all digits of $N$, effectively merging with all the other big pieces along the way. Thus, we end up with a partition with just one big piece, so case 1 holds.

□

Lemmas D.1 and D.2 together handle the cases where some big piece has a leading zero. Thus, we may restrict ourselves to cases where all big pieces do not have leading zeroes.

**Lemma D.3.** *In any optimal partition where all big pieces do not have leading zeroes, it is always true that the number of digits of any big piece is greater than the number of digits of each big piece before it.*

*Proof.* Suppose on the contrary that there is a pair of consecutive big pieces $V'$ and $V$ such that the first one has at least as many digits as the second one. Then one can lengthen $V'$ one digit right, and then shorten $V$ one digit left (in effect, at least scaling $V'$ by 10 and removing the most significant digit of $V$). This is guaranteed to increase the sum because the worst case that could happen is that $V'$ and $V$ have the same number of digits, $V'$ is of the form $1000\ldots$ and $V$ is of the form $9999\ldots$, and even then, this process still increases the sum.[27] Thus, this contradicts optimality. □

The following lemma constitutes the final piece of the puzzle.

**Lemma D.4.** *If there is an optimal partition such that that all big pieces do not have leading zeroes, then there is also an optimal partition such that there is at most one big piece.*

*Proof.* Suppose we have an optimal partition where big pieces do not have leading zeroes. By lemma D.3, there is also an optimal partition where the number of digits of each subsequent big piece increases.

Now, suppose that our partition has at least two big pieces. Let $V'$ and $V$ be the first two. By assumption, $V'$ has fewer digits than $V$. Suppose that $V'$ has $a$ digits and $V$ has $b$ digits, and $2 \leq a < b$. Then we do the following process $b-1$ times:

*Lengthen $V'$ one digit right, and then shorten $V$ one digit left.*

This effectively reduces $V$ into a small piece.

Now, we claim that after this $(b-1)$-step process, the sum has increased. This is because, like in the proof of lemma D.3, the worst case that could happen is that $V'$ is of the form $1000\ldots$ and $V$ is of the form $9999\ldots$, and even then, one can show that this process still increases the sum, if only by a little bit. Thus, this contradicts optimality. □

[27] We invite the reader to prove by themselves why this is indeed the worst possible.

We are now ready to prove claim D.7, by piecing together all the lemmas we have just proven:

*Proof.* First, assume that $N$ has no leading zeroes. then by Lemma D.1, we know that there is an optimal partition in which all big pieces do not have leading zeroes. Thus by Lemma D.4, there is an optimal partition with at most one big piece.

Next, assume that $N$ has leading zeroes. Then by using both Lemmas D.1 and D.2, we see that there are two cases:

1. *There is an optimal partition with at most one big piece.* Then we are done, since this is exactly what we want to prove.
2. *There is an optimal partition in which all big pieces do not have leading zeroes.* In this case, again by Lemma D.4, there is an optimal partition with at most one big piece.

In any case, we have shown that there is always an optimal partition with at most one big piece. □

# Problem E: $N$-fruit combo

**Problem Author:** Karl Ezra S. Pilario

**Keywords:** Point-line distance, Convex hull

## E.1 Problem

Given a set of $N$ points, does there exist a line such that each of the points is at most 3 units away from that line?

## E.2 Solution

Before anything else, if you've been playing Fruit Ninja, note first that the definition of a straight slice in this problem is different from that in the real game. In the game, the straight slice is actually a line segment, and slices are not done instantaneously. In this problem, aside from slices being instantaneously done regardless of length, you will realize that you should make the straight slice as long as possible so as to include the most number of strawberries all the time.

When the problem said that the straight slice "infinitely extends beyond the screen," it means that even if the gesture is physically done only within the rectangular region $0 \leq X \leq 200$ and $0 \leq Y \leq 150$, the produced line goes beyond that region. The following test case shows why this is important.

**Input**

```
1
4
1 2
2 1
0 11
11 0
```

**Output**

```
Case #1: 1
```

The naïve solution is to do some form of regression. The equation of the best-fit line is calculated, after which, the perpendicular distance between all strawberries and the best-fit line is checked to be within 3 units. If at least 1 strawberry is farther than 3 units from the line, the answer is 0. This solution will most likely yield a Wrong Answer verdict to one (or both) of the following the test cases:

**Input**

```
2
3
10 20
10 30
10 40
10
0 6
10 0
20 0
30 0
40 0
50 0
60 0
70 0
80 0
90 6
```

**Output**

```
Case #1: 1
Case #2: 1
```

For this problem, brute-force solutions are accepted, because the test cases are small ($N \leq 10$). You need not to determine the equation of the correct straight slice, because multiple correct straight slices exist. The brute-force solution assumes that there exists some solution line that is parallel to one of the lines determined by two of the points.[28]

1. For every pair of points in the set, consider the line $L$ determined by these two points.
2. Get the farthest points from line $L$ in the set in both directions, and let it be point $P_1$ and $P_2$. Let $D_1$ and $D_2$ be the perpendicular distances of these points from $L$, respectively.
3. If $D_1 + D_2 \leq 6$, then output the answer 1. Otherwise, if no such line $L$ exists such that $D_1 + D_2 \leq 6$, output 0.

The solution above posits that a correct straight slice exists midway between points $P_1$ and $P_2$, and is parallel to line $L$. This slice ensures all the other points to be within the 6-unit width strip from $P_1$ to $P_2$ parallel to $L$.

This algorithm takes $O(N^3)$ time to finish. Lastly, avoid floating-point operations! The inputs are all integers for a reason. In comparing distances such as $\sqrt{x^2 + y^2} \leq D$, use $x^2 + y^2 \leq D^2$ instead. Otherwise, you may risk getting Wrong Answer verdicts due to rounding errors.

## E.3 Convex hull

We now try to seek faster solutions.

[28] This can be proved by starting at a solution line, and then continuously moving it so that it becomes parallel to one such line. Rigorously proving that such movement is possible is a bit involved, so we will defer to intuition here.

The first observation is that we don't need to test all candidates $L$. With a bit of intuition and imagination, one can see that the only $L$'s we need to check are those which are parallel to some side of the convex hull of the $N$ points.[29] Thus, if one calculates the convex hull first (in which there are standard optimal $O(N \log N)$ time algorithms for, such as Graham's scan), then the algorithm takes $O(N^2)$ time (since there are at most $N$ sides in the convex hull).

One can improve this algorithm further by noting that the farthest point from any line is also in the convex hull! Furthermore, as you check each side of the convex hull in turn in one direction, for example counterclockwise, the sequence of farthest points also go counterclockwise, so one can just use a pointers that goes around the convex hull and stays at every farthest point, so that the second step reduces to $O(N)$! This means that the bottleneck is now the calculation of the convex hull, so the algorithm runs in $O(N \log N)$.

## E.4 Summary

Here is the summary of the algorithm:

1. For every pair of points in the set, consider the line $L$ determined by these two points.
2. Get the farthest points from line $L$ in the set in both directions, and let it be point $P_1$ and $P_2$. Let $D_1$ and $D_2$ be the perpendicular distances of these points from $L$, respectively.
3. If $D_1 + D_2 \leq 6$, then output the answer 1. Otherwise, if no such line $L$ exists such that $D_1 + D_2 \leq 6$, output 0.

This runs in $O(N^3)$ time.

[29] The **convex hull** of a set of points is the smallest convex set that contains that set of points (a set is **convex** if for every two points in the set, the line segment connecting them is also in the set). In the case of a finite set of points, the convex hull is always a polygon.

# Problem F: Alien Defense Deux

**Problem Authors:** Kevin Charles V. Atienza, Tim Joseph F. Dumol (Story-writer)

**Problem Tester:** John Eddie R. Ayson

**Keywords:** Dynamic programming, Binary Search

## F.1 Problem

Given an $M \times N$ grid, where each cell is either a ‘#’ and ‘.’, how many square-shaped regions of each size are there that do not contain ‘#’s?

## F.2 Gotcha

Before we proceed with describing the solution, we would like to point out a mistake made by at least one contestant. The mistake was to print only the number of valid square-shaped regions with sizes $1 \times 1$, $2 \times 2$ and $3 \times 3$. The problem statement doesn’t say anything about checking only squares of size at most $3 \times 3$. Most likely, they inferred it from the sample I/O, which only prints for squares up to $3 \times 3$. The lesson here is, as usual, to read the problem statement carefully, and don’t overgeneralize from the sample I/O, which are intended only to illustrate.

## F.3 Slow solution

The naïve way to do this is to simply check each square whether it contains a ‘#’ or not, and tally them for each size. However, this approach is slow. To see this, note that processing a square of size $S$ takes $S^2$ steps, and there are $(M - S + 1)(N - S + 1)$ squares of size $S$. Therefore, processing all squares of size $S$ takes $S^2(M - S + 1)(N - S + 1)$ steps, and processing all squares takes:

$$\sum_{S=1}^{\min(M,N)} S^2(M - S + 1)(N - S + 1) \approx \frac{(M + N)^5}{960}$$

For $M = N = 900$ (the maximum allowed in the problem), this is approximately 11 trillion operations! Assuming that the computer can process 500 million operations per second, this means that this method will take approximately ten hours per test case!

## F.4 Faster but still slow solution: Preprocessing

One can improve the above approach by preprocessing the grid so that one can quickly compute whether a square contains a ‘#’. Indeed, if $C(i,j)$ is the number of ‘#’s in the subgrid with corners $(1,1)$ and $(i,j)$, then the number of ‘#’s in the subgrid with corners $(i_1,j_1)$ and $(i_2,j_2)$ (assuming $i_1 \le i_2$ and $j_1 \le j_2$) is $C(i_2,j_2) - C(i_2,j_1-1) - C(i_1-1,j_2) + C(i_1-1,j_1-1)$. The values $C(i,j)$ can be precalculated in $O(MN)$ time using the recurrence:

$$C(i,j) = \begin{cases} C(i,j-1) + C(i-1,j) - C(i-1,j-1) + 1 & \text{if the } i^{\text{th}} \text{ cell is a ‘\#’} \\ C(i,j-1) + C(i-1,j) - C(i-1,j-1) & \text{if the } i^{\text{th}} \text{ cell is a ‘.’} \end{cases}$$

This means that a square of any size can be processed in $O(1)$ time by using the table $C(i,j)$ to check if there is any ‘#’ in that square. Thus the total number of steps is just the number of squares:

$$\sum_{S=1}^{\min(M,N)} (M-S+1)(N-S+1) \approx \frac{(M+N)^3}{24}$$

For $M = N = 900$, and assuming 500 million operations per second again, this will take around 0.5 seconds per test case. However, since there are at most 12 cases, this might take around 6 seconds to process the whole input, which still exceeds the time limit.

## F.5 Fast Solution: Binary Search

The solution above can be improved further with the following observation: If there is a square of size $S$ with bottom-right corner $(i,j)$ not containing a ‘#’, then there is a square of any size $T < S$ with the same bottom-right corner! This means that for a given bottom-right corner, we can just find the largest square with that corner not containing any ‘#’. If $F(S)$ is the number of size-$S$ squares not containing a ‘#’ and $L(S)$ is the number of bottom-right corners whose largest square without a ‘#’ has size $S$, then we have the following relationship:

$$F(S) = L(S) + L(S+1) + L(S+2) + \cdots$$

Since $S$ is at most $\min(M,N)$, all the $F$’s can be calculated from the $L$’s in $O(\min(M,N)^2)$ time!

Now, given a bottom-right corner, what is the largest square not containing a ‘#’? By the same observation above, we can use binary search to find it! Thus, finding the largest square per corner takes $O(\log \min(M,N))$ time, and the final complexity is $O(MN \min(M,N))$. This passes the time limit!

## F.6 Faster Solution: Dynamic programming

In fact, there is an even faster and simpler solution than the above! Let $G(i,j)$ be the size of the largest square with bottom-right corner $(i,j)$ not containing a '#'. Assume that cell $(i,j)$ contains a '.' (otherwise, we immediately know that $G(i,j) = 0$). Obviously, if $G(i,j) = S$, then $G(i-1,j)$, $G(i-1,j-1)$ and $G(i,j-1)$ are all at least $S-1$ (why?). Furthermore, if $G(i-1,j)$, $G(i-1,j-1)$ and $G(i,j-1)$ are all at least $S$, then $G(i,j)$ is at least $S+1$ (why?). Thus, combining all this, we have the following cool observation (when cell $(i,j)$ is a '.'):

$$G(i,j) = \min(G(i-1,j), G(i-1,j-1), G(i,j-1)) + 1$$

Thus, the table of $G$ values can be constructed in $O(MN)$ time, and using this table, we can easily and quickly calculate $L$ (and thus $F$). Thus, the overall algorithm runs in $O(MN)$! (Note that we don't have to precalculate the table $C(i,j)$ anymore!)

As a final optimization, we note that there is a way to calculate the $F$ values from the $L$ values in $O(\min(M,N))$ time instead of $O(\min(M,N)^2)$. Though it doesn't affect the final time complexity, it still might reduce the running time of your program somewhat. We leave it to the reader to discover this method.

## F.7 Summary

Here is the summary of the algorithm:

1. Create a table $G(i,j)$ and build it with the following:

$$G(i,j) = \begin{cases} \min(G(i-1,j), G(i-1,j-1), G(i,j-1)) + 1 & \text{if the } i^{\text{th}} \text{ cell is a '.'} \\ 0 & \text{if the } i^{\text{th}} \text{ cell is a '\#'} \end{cases}$$

2. Build the array $L(S)$, $1 \le S \le \min(M,N)$, where $L(S)$ is the number of times $S$ appears in the table $G$
3. Build the array $F(S)$, $1 \le S \le \min(M,N)$, where $F(S) = L(S) + L(S+1) + L(S+2) + \cdots$
4. The array $F$ contains the answers.

This runs in $O(MN)$ time.

University of the Philippines
Association for Computing Machinery

ALGOLYMPICS 2015

# Problem G: For Science™!

**Problem Authors:** Kevin Charles V. Atienza, Payton Robin O. Yao (Story-writer)

**Problem Tester:** John Eddie R. Ayson

**Keywords:** Set (data structure), Tree traversal, Heavy-light decomposition

## G.1 Problem

The problem can also be stated as follows: You are given a rooted tree with $N$ nodes, and there is a number on every node. Now, for each node $i$, you need to output the sum of the *distinct* numbers in the subtree rooted at $i$.

## G.2 Naïve solution

One can simply implement a solution naïvely by simply collecting all numbers in a subtree using a set, and adding the numbers in the set. However, each node can potentially have $O(N)$ number of descendants, so this algorithm runs in $O(N^2)$ in the worst case (assuming you are using a set with an $O(1)$ amortized lookup and insert, like one implemented with a *hash table*).

Because $N$ can be as large as 100,000, one should also take care to implement the algorithm iteratively, to avoid the risk of *stack overflow*.

## G.3 Fast solution

Amazingly, the approach above can be made significantly faster with a simple modification. For the current node, suppose you have already computed the sets of distinct numbers for every child tree. Take the largest set, and then insert all the other numbers to it, including the number on the current node itself.

The advantage of this is that you only have to iterate the nodes of the smaller trees, saving potentially a lot of computations per node. In fact, one can prove that this runs in $O(N \log N)$ time (assuming $O(1)$ set operations again), by observing that any particular number on a node will only be iterated over on an ancestor if it belongs to a non-largest subtree of that ancestor. Since such a non-largest subtree is at most half the size of the tree, and you can only halve the size of a tree $O(\log N)$ times, this means that each node will only be iterated over $O(\log N)$ times!

## G.4 Summary

Here is the summary of the algorithm:

1. Preprocess the tree. Use an iterative implementation to prevent stack overflow.
2. Traverse the tree bottom-up (iteratively), calculating for every node $x$ the sets $S[x]$ of distinct numbers and their sums $T[x]$. For the current node $x$:
   (a) Find the child $y$ with the largest $S[y]$, and set $S[x] := S[y]$ and $T[x] := T[y]$.
   (b) Insert the number on $x$ to the set $S[x]$ (update $T[x]$ if necessary).
   (c) Insert the values from every other subtree to $S[x]$, updating $T[x]$ if necessary.
3. Output the sums for each node in the correct order.

The time complexity is $O(N \log N)$ assuming $O(1)$ set operations.

# Problem H: Algols for Algolympia

**Problem Authors:** Kevin Charles V. Atienza, Alvin John M. Burgos (Story-writer)

**Problem Tester:** John Eddie R. Ayson

**Keywords:** Modular Exponentiation, Fast Fourier Transform, Modular Arithmetic

## H.1 Problem

Given a sequence of $N$ numbers $A_1 \dots A_N$, find the product of all pairwise sums, i.e. the following:

$$\prod_{1 \le i < j \le N} (A_i + A_j)$$

Give your answer modulo $10^9 + 7$.

## H.2 Slow solutions

It's easy to write a brute-force solution for this problem that simply computes the sums of all pairs and multiplies them all (reducing intermediate values modulo $10^9+7$ every time). However, since $N$ can be up to 200,000 and there are $\Theta(N^2)$ pairs, this won't pass the time limit.

Here's another way to do it. We can define the arrays $[a_0, a_1, a_2, \dots]$ and $[b_0, b_1, b_2, \dots]$, where $a_v$ is the number of times $v$ appears in the sequence $A_i$, and $b_v$ is the number of times that $v$ appears as $A_i + A_j$ for $1 \le i < j \le N$. The $a_v$ values can be computed quickly from the $A_i$ in $O(N)$, and the $b_v$'s can be calculated from th $a_v$'s using the following relationship:

$$b_v = \begin{cases} \frac{1}{2}[a_0a_v + a_1a_{v-1} + a_2a_{v-2} + \cdots + a_{v-1}a_1 + a_va_0] & \text{if } v \text{ is odd} \\ \frac{1}{2}[(a_0a_v + a_1a_{v-1} + a_2a_{v-2} + \cdots + a_{v-1}a_1 + a_va_0) - a_{v/2}] & \text{if } v \text{ is even} \end{cases}$$

The $\frac{1}{2}$ factor is to account for the fact that the formula originally counts pairs $A_i + A_j$ where $i > j$, and the $-a_{v/2}$ in the even case is for the cases where $i = j$.

From this, the answer can be calculated as:

$$\left(\prod_{0 \le v \le 2M} v^{b_v}\right) \bmod (10^9 + 7)$$

Each term in this product can be computed quickly using *modular exponentiation*, and the bottleneck is computing the $b_v$ values themselves. Unfortunately, computing each $b_v$ by this formula takes $O(v)$, so computing all $b_v$s takes time:

$$\sum_{v=1}^{2M} O(v) = O\left(\frac{2M(2M+1)}{2}\right) = O(M^2)$$

where $M$ is the largest $A_i$. Since $M$ can be up to 200,000, this still won't pass the time limit.

## H.3 Polynomial multiplication

The above approach with $a_v$'s and $b_v$'s can be made significantly faster with a few modifications. First, define a new array $[c_0, c_1, c_2, \ldots]$ where $c_v$ is defined as:

$$c_v = a_0 a_v + a_1 a_{v-1} + a_2 a_{v-2} + \cdots + a_{v-1} a_1 + a_v a_0$$

This means that $b_v$ can be calculated from the $a_v$ and $c_v$ in $O(M)$ with the following:

$$b_v = \begin{cases} \frac{1}{2} c_v & \text{if } v \text{ is odd} \\ \frac{1}{2}[c_v - a_{v/2}] & \text{if } v \text{ is even} \end{cases}$$

Now, the trouble is that $c_v$ is still hard to calculate from the $a_v$, so we haven't really made any improvements yet. However, notice that the formula for $c_v$ in terms of $a_v$ is a convolution! In fact, if we define two polynomials $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots$ and $C(x) = c_0 + c_1 x + c_2 x^2 + \cdots$, then we have the following:

$$C(x) = A(x)^2$$

This means that $C(x)$ can be calculated by squaring a polynomial, and there are well-known ways of doing that!

## H.4 Semi-fast solution: Karatsuba multiplication

Multiplying two degree-$M$ polynomials $A(x)$ and $B(x)$ naïvely takes $O(M^2)$ time, but there is a well-known way to do it much faster.

To do this, first assume that $M$ is odd (if it is even, then you can add a dummy leading term with coefficient zero) so that the polynomials have an even number of terms. Then, assuming that the first and second halves of $A(x)$ are $A_1(x)$ and $A_2(x)$, respectively, and the first and second halves of $B(x)$ are $B_1(x)$ and $B_2(x)$, respectively, so that:

$$A(x) = A_1(x)x^{(M+1)/2} + A_2(x)$$
$$B(x) = B_1(x)x^{(M+1)/2} + B_2(x)$$

then the product $A(x)B(x)$ is the following:

$$A(x)B(x) = C_1(x)x^{M+1} + C_2(x)x^{(M+1)/2} + C_3(x)$$

where

$$C_1(x) = A_1(x)B_1(x)$$
$$C_2(x) = A_1(x)B_2(x) + A_2(x)B_1(x)$$
$$C_3(x) = A_2(x)B_2(x)$$

The $C_i(x)$'s can be calculated by recursively calling our multiplication method four times, and the additions at the end can be done in $O(M)$, so that if the running time is $T(M)$, then we have the following relation:

$$T(M) = 4T(M/2) + O(M)$$

By the master theorem[30], $T(M) = O(M^2)$, so there is still no improvement yet. However, there is a way to reduce the number of multiplications from four to three, by noticing the following:

$$C_2(x) = (A_1(x) + A_2(x))(B_1(x) + B_2(x)) - C_1(x) - C_3(x)$$

By computing $C_1(x)$ and $C_3(x)$ first and then using the results to calculate $C_2(x)$, we only need three multiplications (at the cost of more additions, but that's okay). This means that the running time now follows the new relation:

$$T(M) = 3T(M/2) + O(M)$$

By the master theorem again, $T(M) = O(M^{\log_2 3}) = O(M^{1.585})$, which is an improvement! This algorithm is called the *Karatsuba algorithm.*

However, $O(M^{1.585})$ for the current bounds for $M$ is still too slow, unless your implementation is really, *really* good. Luckily, there is still a faster way to multiply two polynomials, which we will describe in the next part.

[30] The **master theorem** solves recurrence equations of this kind, which is useful because these forms arise frequently in algorithmic analysis. More about it here: http://en.wikipedia.org/wiki/Master_theorem

## H.5 Fast solution: Fast Fourier Transform

A faster way to multiply two polynomials arises by transforming the polynomials into a representation in which multiplication is easy. We'll describe such a representation here.

A polynomial is usually represented by its $M+1$ coefficients, because a polynomial is uniquely defined by those coefficients. However, currently our best way to multiply two polynomials in that representation runs in $O(M^{1.585})$.[31]

A polynomial $P(x)$ of degree $M$ is also uniquely defined by its value on $M+1$ fixed distinct $x$ arguments. Furthermore, given $V$ distinct values for $V$ distinct $x$ arguments, there is a unique polynomial of degree $< V$ having all those values! For example, the unique polynomial $P(x)$ of degree less than three such that $P(0) = 0$, $P(1) = 0$ and $P(2) = 2$ is $P(x) = x^2 - x$. Therefore, we can represent our polynomials as a list of $V$ values at $V$ distinct points. This is called the **point-value representation** of a polynomial. Now, the nice thing about this representation is that addition and multiplication are very easy to carry out: just add or multiply the corresponding values! This means that this representation would be very useful for our purposes.

However, our input polynomials are given in the coefficient form, and we also want the answer in the coefficient form (because we need the coefficients!). Thus, we need to find a way to transform our polynomials into the point-value representation, and vice versa, quickly. General ways of transforming a polynomial to and from the point-value representation (repeated *Horner evaluation* and *Lagrange polynomials*, respectively) take $O(M^2)$ time, so we need to find something else.

The trick is to choose the $x$ values such that computing this representation can be done quickly. It turns out that such a fast method exists when the $x$'s are chosen to be the $V$th roots of unity. This is called the **discrete Fourier transform** (DFT) of the coefficients of the polynomial.[32] Our transform will exploit some properties of such roots of unity.

First, let us assume that $V$ is a power of two—if not, we can simply pad a few leading zero coefficients, and this will only at most double the degree of the polynomial—so that $V = 2^k$, and that the degree of the polynomial $P(x)$ we want to transform is less than $V$ (so that the polynomial is uniquely defined). Thus, we wish to evaluate $P(x)$ at the $2^k$th roots of unity.

Let $\omega$ be any *primitive* $2^k$th root of unity (for example, one can choose $\omega = \cos\frac{2\pi}{2^k} + i\sin\frac{2\pi}{2^k}$). Then some important properties of $\omega$ are the following:

1. All the $2^k$th roots of unity can be expressed as powers of $\omega$, i.e., they are precisely the set:

$$\{1, \omega, \omega^2, \omega^3, \dots, \omega^{2^k-1}\} = \{\omega^i : 0 \le i < 2^k\}$$

2. $\omega^{2^{k-1}} = -1$ and $\omega^{2^k} = 1$.

[31] There are in fact faster ways to multiply polynomials in this representation (such as the Toom-Cook algorithm), but as far as I know, there isn't one that is relatively simple and at the same time matches the efficiency of the algorithm that will be described here.

[32] We say the "DFT of the coefficients of the polynomial" instead of the "DFT of the polynomial" because the DFT is actually a transform that takes a vector (list of values) to another vector, but we won't describe DFT much more than what is needed for our purposes.

3. $\omega^2$ is a primitive $2^{k-1}$th root of unity, thus $\{1, \omega^2, \omega^4, \omega^6, \dots, \omega^{2^k-2}\} = \{\omega^{2i} : 0 \le i < 2^{k-1}\}$ is precisely the set of $2^{k-1}$th roots unity.

Now, let $E(x)$ and $O(x)$ be the polynomial obtained by taking the coefficients of the even-exponent and odd-exponent terms of $P(x)$, respectively, so that:

$$P(x) = E(x^2) + xO(x^2)$$

and let's say that $i$ is some number from 0 to $2^{k-1} - 1$. Then observe the following:

$$P(\omega^i) = E(\omega^{2i}) + \omega^i O(\omega^{2i})$$

and

$$\begin{aligned}
P(\omega^{i+2^{k-1}}) &= E(\omega^{2i+2^k}) + \omega^{i+2^{k-1}} O(\omega^{2i+2^k}) \\
&= E(\omega^{2i}\omega^{2^k}) + \omega^i \omega^{2^{k-1}} O(\omega^{2i}\omega^{2^k}) \\
&= E(\omega^{2i}(1)) + \omega^i(-1)O(\omega^{2i}(1)) \\
&= E(\omega^{2i}) - \omega^i O(\omega^{2i})
\end{aligned}$$

This means that we can evaluate $P(\omega^i)$ and $P(\omega^{i+2^{k-1}})$ if we know the values of $E(\omega^{2i})$ and $O(\omega^{2i})$. Thus we can finish our task quickly if we know the values of $E(x)$ and $O(x)$ for all $x \in \{\omega^{2i} : 0 \le i < 2^{k-1}\}$. But this means precisely that we want the DFT of the coefficients of $E$ and $O$! Since $E$ and $O$ are degree $2^{k-1}$ polynomials, we can simply solve these two DFT problems with the same method recursively, and then finish calculating the DFT of the coefficients of $P$ in $O(V)$ time. This method is called the **fast Fourier transform** (FFT) method.

To calculate how fast this is, note that we recursively call our method two times on inputs half the size, and then combine the results in linear time, so if $T(V)$ is the running time, then:

$$T(V) = 2T(V/2) + O(V)$$

By the master theorem, this means that $T(V) = O(V \log V)$, which is very nice!

### H.5.1 Inverse DFT

Now that we know how to compute the DFT quickly, we also need to know how to compute the inverse quickly, so that we can recover the coefficients of the product. Luckily, the DFT is really nice because the inverse transform of the DFT can be expressed in terms of the DFT itself!

To see how, note that for any primitive $V$th root of unity $\omega$ (where $V$ is not necessarily a power of two), we have the following fact:

$$1+\omega^i+\omega^{2i}+\omega^{3i}+\cdots+\omega^{(V-1)i}=\begin{cases}V & \text{if } V \text{ divides } i\\ 0 & \text{otherwise}\end{cases}$$

Note that this statement holds for all integers $i$, including the negative integers. This fact can be easily proven as follows:

*Proof.* When $V$ divides $i$, we have $\omega^i = 1$, so the sum reduces to $1+1+\cdots+1 = V$.

Next, we show that the sum is 0 when $V$ does not divide $i$. Since $V$ does not divide $i$, we know that $\omega^i \neq 1$ (why?). Now, the roots of the polynomial $x^V - 1$ are exactly the $V$th roots of unity, which include $\omega^i$. However, we can factorize $x^V - 1$ as:

$$x^V - 1 = (x-1)(1+x+x^2+\cdots+x^{V-1})$$

Now, since $\omega^i$ is a root and $(\omega^i - 1) \neq 0$, this means that $\omega^i$ is a root of the remaining term, $(1+x+x^2+\cdots+x^{V-1})$. In other words:

$$1+\omega^i+\omega^{2i}+\omega^{3i}+\cdots+\omega^{(V-1)i}=0$$

This proves the claim. □

Now, this fact allows us to formulate the inverse DFT in terms of the DFT. We know that the $i$th term of the DFT is the following (assuming that $P(x) = p_0 + p_1x + \cdots + p_{V-1}x^{V-1}$):

$$q_i = p_0 + p_1\omega^i + p_2\omega^{2i} + \cdots + p_{V-1}\omega^{(V-1)i}$$

Now, consider the following value:

$$q_0 + q_1\omega^{-i} + q_2\omega^{-2i} + \cdots + q_{V-1}\omega^{-(V-1)i}$$

This is almost the DFT of the values $\{q_i\}$, except that we are using the primitive root $\omega^{-1}$ instead of $\omega$. Let's manipulate it further:

$$\begin{aligned}
&= q_0 + q_1\omega^{-i} + q_2\omega^{-2i} + \cdots + q_{V-1}\omega^{-(V-1)i} \\
&= \sum_{j=0}^{V-1} q_j\omega^{-ij} \\
&= \sum_{j=0}^{V-1} \left( \sum_{k=0}^{V-1} p_k\omega^{jk} \right) \omega^{-ij} \\
&= \sum_{j=0}^{V-1} \sum_{k=0}^{V-1} p_k\omega^{j(k-i)} \\
&= \sum_{k=0}^{V-1} \sum_{j=0}^{V-1} p_k\omega^{j(k-i)} \\
&= \sum_{k=0}^{V-1} p_k \sum_{j=0}^{V-1} \omega^{j(k-i)}
\end{aligned}$$

Now, consider the inner sum $\sum_{j=0}^{V-1} \omega^{j(k-i)} = 1 + \omega^{k-i} + \omega^{2(k-i)} + \cdots + \omega^{(V-1)(k-i)}$. From what we just proved above, we know that this is equal to $V$ if $k-i$ is divisible by $V$ and equal to 0 otherwise. But since $0 \le k, i < V$, the latter only happens if $k = i$. Thus, the inner sum is almost always zero, except when $k = i$ in which case it is $V$. In Iverson bracket notation[33], this is just $V \cdot [k = i]$. Thus, we can now continue:

$$\begin{aligned}
&= \sum_{k=0}^{V-1} p_k \sum_{j=0}^{V-1} \omega^{j(k-i)} \\
&= \sum_{k=0}^{V-1} p_k (V \cdot [k = i]) \\
&= Vp_i
\end{aligned}$$

Therefore, the "DFT" of the $\{q_i\}$ values is just $V$ times the original $\{p_i\}$ values! This means that the inverse DFT can be done by simply finding the DFT of the values (but using the inverse of the primitive root used in the original DFT) and then dividing them all by $V$! So if one uses the FFT method again, this runs in $O(V \log V)$ time too!

### H.5.2 Other caveats

The above techniques now describe a way to multiply two polynomials of degree $M$ in $O(M \log M)$ running time (get the DFTs of the input, multiply point-wise, then get the inverse DFT). However, the output

[33] The *Iverson bracket* notation gives 1 if the condition inside the bracket is true, and 0 otherwise. To learn more, see http://en.wikipedia.org/wiki/Iverson_bracket.

polynomial has degree $2M$, so the number of points that one must use for the DFT must be greater than $2M$. This means that $V$ must be a power of two that is greater than $2M$. By choosing $V$ to be the least such power, we still guarantee an $O(M \log M)$ running time.

There is another potential problem with using the FFT method when multiplying two polynomials, namely that of floating point rounding error. Since all the computations are done with floating point numbers, it is possible that rounding error occurs. Since we need the exact value of the coefficients for our algorithm, having rounding error will almost surely give you a wrong answer.

Luckily, there is a way to calculate the product of two polynomials *exactly* by doing the DFT on the finite field $\mathbb{Z}/p\mathbb{Z}$, and using a modular $V$th root of unity from that field.[34] This way, we avoid using floating point numbers altogether. The $p$ must be chosen carefully though:

- A $V$th root of unity only exists modulo $p$ iff $p$ is of the form $Vm + 1$ for some integer $m$. This means that the prime must be of this form. Since we usually choose $V$ to be a power of two, this usually means that the prime is of the form $2^k m + 1$.
- Since the coefficients are reduced modulo $p$, the value $p$ must be chosen so that it exceeds the maximum coefficient in the product. Alternatively, one can compute the DFT on finite fields for multiple primes, and then reconstruct the final answer using the *Chinese remainder theorem*.

## H.6 Summary

Here is the summary of the algorithm:

1. Let $M = \max_i A_i$.
2. Compute the array $[a_0, a_1, a_2, \ldots, a_M]$ where $a_v$ is the number of times $v$ appears in the array $A_i$.
3. Let $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_M x^M$.
4. Let $C(x) = A(x)^2$. Calculate $C(x)$ using the Fast Fourier Transform (FFT) method to get an $O(M \log M)$ running time.
5. Let $c_v$ be the coefficient of $x^v$ in $C(x)$, for $0 \le v \le 2M$.
6. For $0 \le v \le 2M$, let $b_v$ be the following:
$$b_v = \begin{cases} \frac{1}{2} c_v & \text{if } v \text{ is odd} \\ \frac{1}{2}[c_v - a_{v/2}] & \text{if } v \text{ is even} \end{cases}$$
7. The answer is then $\displaystyle\prod_{1 \le v \le 2M} v^{b_v}$, modulo $10^9 + 7$.

The time complexity of this algorithm is $O(M \log M)$.

[34] This is called the **number-theoretic transform** (NTT). To learn more, see http://en.wikipedia.org/wiki/Discrete_Fourier_transform_%28general%29#Number-theoretic_transform.