

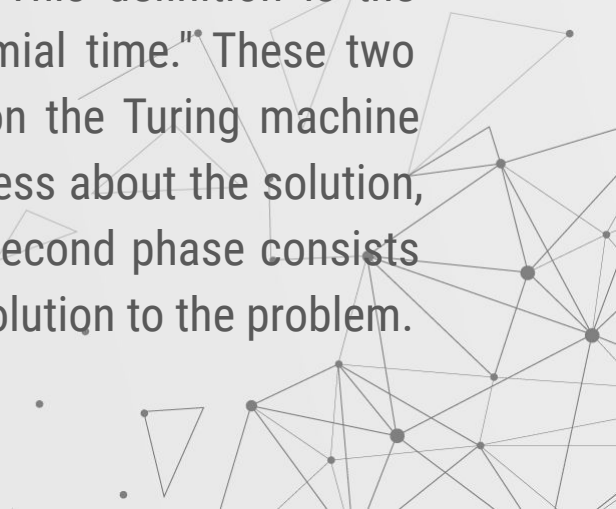
The background features a complex network of thin grey lines and dots, forming a web-like structure. Scattered throughout are various triangles of different sizes and orientations, some with solid black dots at their vertices. The overall aesthetic is minimalist and technical.

NON DETERMINISTIC POLYNOMIAL TIME

NP Hard and NP Complete Problems - Algorithms and Code

In computational complexity theory, NP (nondeterministic polynomial time) is a complexity class used to classify decision problems. NP is the set of decision problems for which the problem instances, where the answer is "yes", have proofs verifiable in polynomial time by a deterministic Turing machine.

An equivalent definition of NP is the set of decision problems *solvable* in polynomial time by a non-deterministic Turing machine. This definition is the basis for the abbreviation NP; "nondeterministic, polynomial time." These two definitions are equivalent because the algorithm based on the Turing machine consists of two phases, the first of which consists of a guess about the solution, which is generated in a non-deterministic way, while the second phase consists of a deterministic algorithm that verifies if the guess is a solution to the problem.

An abstract geometric pattern consisting of numerous small black dots connected by thin, light gray lines. The pattern is concentrated in the bottom right corner of the slide, forming a complex, web-like structure that resembles a network or a molecular model. Some lines form closed polygons, while others are part of larger, more open structures.

Def.

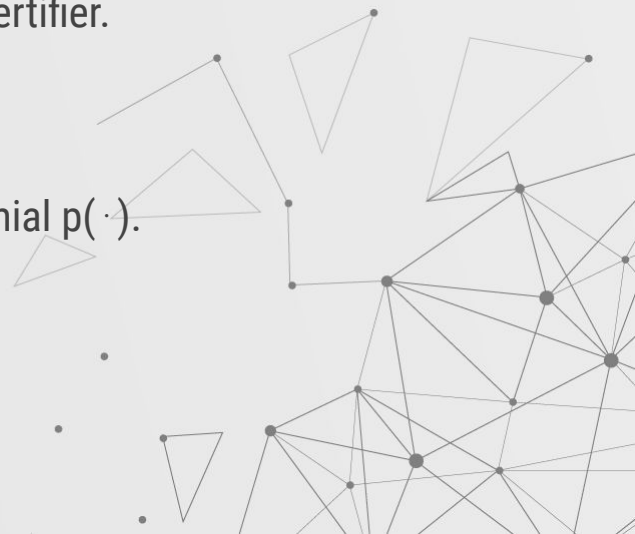
Algorithm $C(s, t)$ is a certifier for problem X if for every string $s : s \in X$ iff there exists a string t such that $C(s, t) = \text{yes}$.


Def.

NP = set of decision problems for which there exists a poly-time certifier.

- $C(s, t)$ is a poly-time algorithm.
- Certificate t is of polynomial size: $|t| \leq p(|s|)$ for some polynomial $p(\cdot)$.

↖
"certificate" or "witness"





“ NP captures vast domains of computational, scientific, and mathematical endeavors, and seems to roughly delimit what mathematicians and scientists have been aspiring to compute feasibly. ”

— Christos Papadimitriou



WHAT IS NOT NP?

- *Undecidable problems*

- Given a polynomial with integer coefficients, does it have integer roots
- Hilbert's nth problem
- Impossible to check for all the integers
- Even a non-deterministic TM has to have a finite number of states!
- More on decidability later

- *Tautology*

- A boolean formula that is true for all possible assignments
- Here just one 'verifier' will not work. You have to try all possible values





POLYNOMIAL TRANSFORMATIONS

- Is a linear equation reducible to a quadratic equation?
 - Sure! Let coefficient of the square term be 0
- A problem Q can be reduced to another problem Q' if any instance of Q can be “easily rephrased” as an instance of Q' , the solution to which provides a solution to the instance of Q

Def. Problem X polynomial (Cook) reduces to problem Y if arbitrary instances of problem X can be solved using:


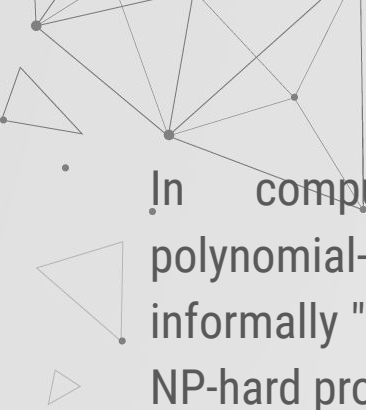
- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y .



NP HARD

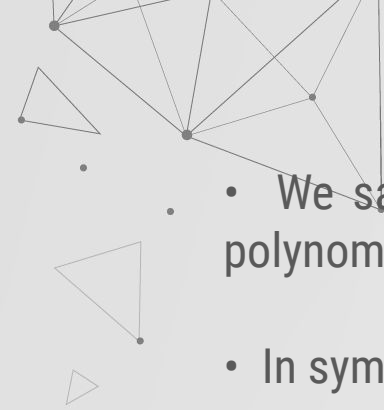

A problem (a language) is said to NP-hard if every problem in NP can be poly time reduced to it.

$$L' \leq_p L \text{ for every } L' \in NP$$



In computational complexity theory, NP-hardness (non-deterministic polynomial-time hardness) is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP". A simple example of an NP-hard problem is the subset sum problem.

A more precise specification is: a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H ; that is, assuming a solution for H takes 1 unit time, H 's solution can be used to solve L in polynomial time. As a consequence, finding a polynomial time algorithm to solve any NP-hard problem would give polynomial time algorithms for all the problems in NP. As it is suspected that $P \neq NP$, it is unlikely that such an algorithm exists.

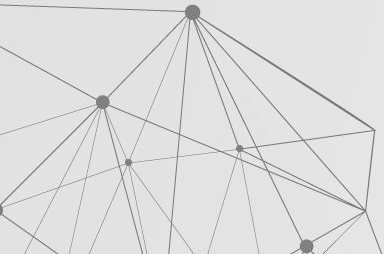
- 
- 
- We say that a decision problem P_i is NP-hard if every problem in NP is polynomial-time reducible to P_i .
 - In symbols, P_i is NP-hard if, for every $P_j \in \text{NP}$, $P_j \text{ poly} \rightarrow P_i$.
 - Note that this doesn't require P_i to be in NP.
 - Highly informally, it means that P_i is 'as hard as' all the problems in NP.
 - If P_i can be solved in polynomial-time, then so can all problems in NP.
 - Equivalently, if any problem in NP is ever proved intractable, then P_i must also be Intractable.

Problem statement :

We have to determine the fastest route for a school bus to start from the depot, visit all the children's homes, get them to school and return back to depot. This is an instance of the classical Travelling Salesman Problem

Problem Description :

A school bus needs to start from the depot early in the morning, pick up all the children from their homes in some order, get them all to school and return to the depot. You know the time it takes to get from depot to each home, from each home to each other home, from each home to the school and from the school to the depot. We want to define the order in which to visit children's homes so as to minimize the total time spent on the route.



Given a graph with weighted edges, we need to find the shortest cycle visiting each vertex exactly once. Vertices correspond to homes, the school and the depot. Edges weights correspond to the time to get from one vertex to another one. Some vertices may not be connected by an edge in the general case.

Input Format :

The first line contains two integers n and m – the number of vertices and the number of edges in the graph. The vertices are numbered from 1 through n . Each of the next m lines contains three integers u , v and t representing an edge of the graph. This edge connects vertices u and v , and it takes time t to get from u to v . The edges are bidirectional: you can go both from u to v and from v to u in time t using this edge. No edge connects a vertex to itself. No two vertices are connected by more than one edge.

Constraints. $2 \leq n \leq 17$; $1 \leq m \leq n(n-1) / 2$; $1 \leq u, v \leq n$; $u \neq v$; $1 \leq t \leq 10^9$.

Output Format :

If it is possible to start in some vertex, visit each other vertex exactly once in some order going by edges of the graph and return to the starting vertex, output two lines. On the first line, output the minimum possible time to go through such circular route visiting all vertices exactly once (apart from the first vertex which is visited twice – in the beginning and in the end). On the second line, output the order in which you should visit the vertices to get the minimum possible time on the route.

That is, output the numbers from 1 through n in the order corresponding to visiting the vertices. Don't output the starting vertex second time. However, account for the time to get from the last vertex back to the starting vertex. If there are several solutions, output any one of them. If there is no such circular route, output just -1 on a single line. Note that for $n = 2$ it is considered a correct circular route to go from one vertex to another by an edge and then return back by the same edge.

Example 1 :

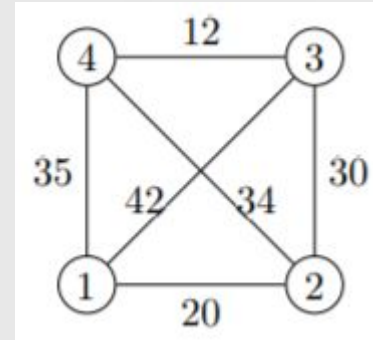
Input:

4 6
1 2 20
1 3 42
1 4 35
2 3 30
2 4 34
3 4 12

Output: 97

1 4 3 2

Explanation:



The suggested route starts in the vertex 1, goes to 4 in 35 minutes, from there to 3 in 12 minutes, from there to 2 in 30 minutes, from there back to 1 in 20 minutes, totalling in $35 + 12 + 30 + 20 = 97$ minutes.

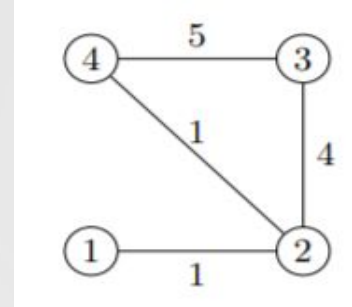
Example 2 :

Input:

4 4
1 2 1
2 3 4
3 4 5
4 2 1

Output: -1

Explanation :



There is no way to visit all the vertices exactly once, as there is only one edge from the vertex 1 (going to 2), so after leaving it we cannot return without visiting 2 twice. Hence solution is -1.

```
from itertools import permutations
from itertools import combinations

INF = 10 ** 9

def read_data():
    n, m = map(int, input().split())
    graph = [[INF] * n for _ in range(n)]
    for _ in range(m):
        u, v, weight = map(int, input().split())
        u -= 1
        v -= 1
        graph[u][v] = graph[v][u] = weight
    return graph

def print_answer(path_weight, path):
    print(path_weight)
    if path_weight == -1:
        return
    print(' '.join(map(str, path)))
```



```
def optimal_path(graph):  
    n = len(graph)  
    C = [[INF for _ in range(n)] for __ in range(1<<n)]  
    backtrack = [[(-1, -1) for _ in range(n)] for __ in range(1<<n)]  
    C[1][0] = 0  
    for size in range(1, n):  
        for S in combinations(range(n), size):  
            S = (0,) + S  
            k = sum([1<<i for i in S])  
            for i in S:  
                if 0 != i:  
                    for j in S:  
                        if j != i:  
                            curr = C[k^1<<i]][j]+graph[i][j]  
                            if curr < C[k][i]:  
                                C[k][i] = curr  
                                backtrack[k][i] = (k^1<<i), j
```




```
if bestAns >= INF:  
    return (-1, [])
```

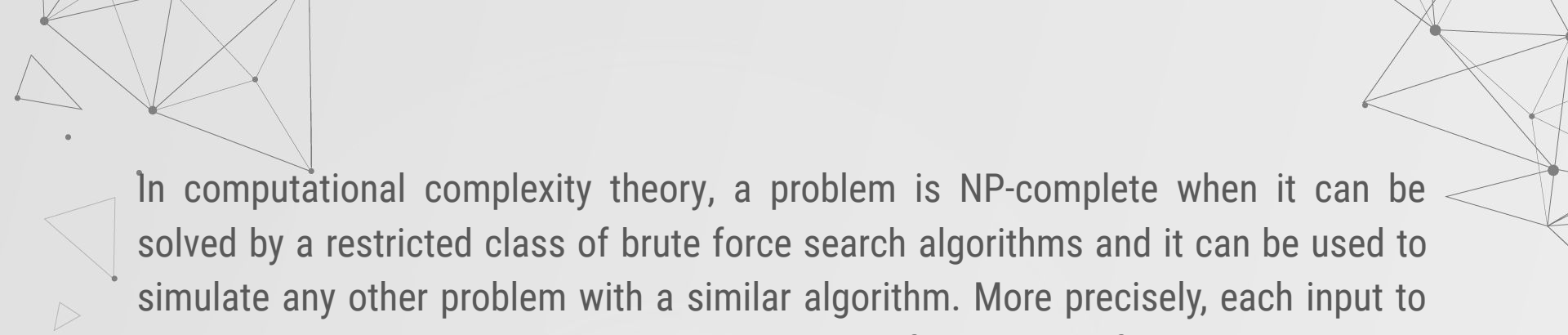
```
bestPath = []  
currIndex1 = (1<<n)-1  
while -1 != currIndex1:  
    bestPath.insert(0, currIndex2+1)  
    currIndex1, currIndex2 = backtrack[currIndex1][currIndex2]  
return (bestAns, bestPath)
```

```
if __name__ == '__main__':  
    print_answer(*optimal_path(read_data()))
```

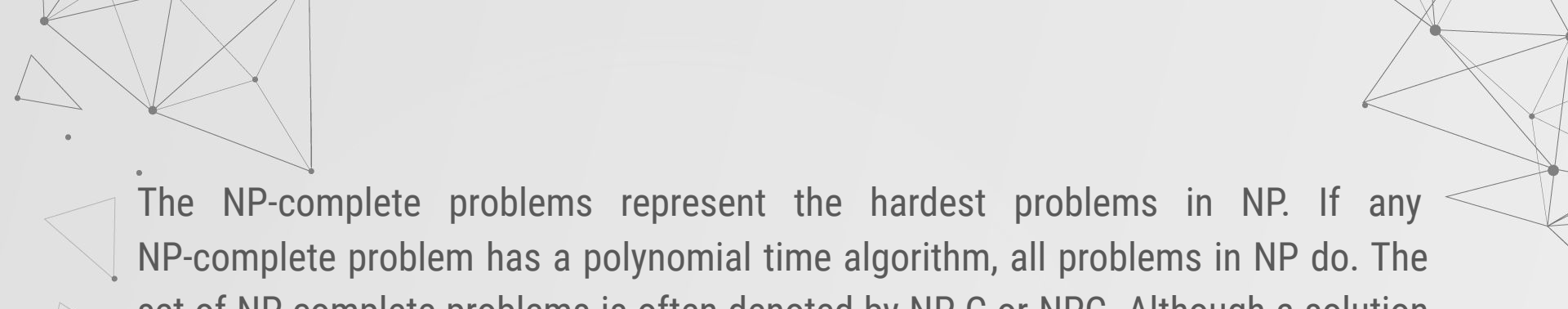


NP COMPLETE


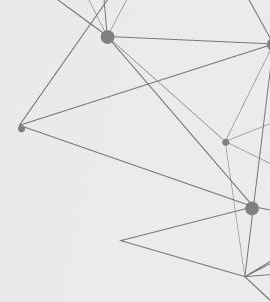
A problem $Y \in \text{NP}$ with the property that for every problem $X \in \text{NP}$, $X \leq_P Y$

The top corners of the slide feature abstract geometric diagrams. The top-left diagram consists of several interconnected triangles and lines, with some vertices marked by small black dots. The top-right diagram is a more complex network of lines and dots, resembling a graph or a complex polygonal structure. The text is centered on the page, overlaid on a light gray background.

In computational complexity theory, a problem is NP-complete when it can be solved by a restricted class of brute force search algorithms and it can be used to simulate any other problem with a similar algorithm. More precisely, each input to the problem should be associated with a set of solutions of polynomial length, whose validity can be tested quickly (in polynomial time), such that the output for any input is "yes" if the solution set is non-empty and "no" if it is empty. The complexity class of problems of this form is called NP, an abbreviation for "nondeterministic polynomial time". A problem is said to be NP-hard if everything in NP can be transformed in polynomial time into it, and a problem is NP-complete if it is both in NP and NP-hard.

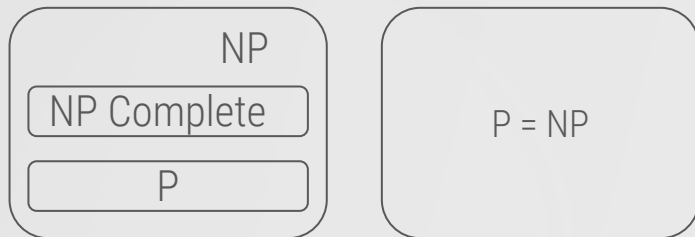


The NP-complete problems represent the hardest problems in NP. If any NP-complete problem has a polynomial time algorithm, all problems in NP do. The set of NP-complete problems is often denoted by NP-C or NPC. Although a solution to an NP-complete problem can be *verified* "quickly", there is no known way to *find* a solution quickly. That is, the time required to solve the problem using any currently known algorithm increases rapidly as the size of the problem grows. As a consequence, determining whether it is possible to solve these problems quickly, called the P versus NP problem, is one of the fundamental unsolved problems in computer science today.

- 
- 
- We say that a decision problem P_i is NP-complete if
 - it is NP-hard and
 - it is also in the class NP itself.
 - In symbols, P_i is NP-complete if P_i is NP-hard and $P_i \in \text{NP}$
 - Highly informally, it means that P_i is one of the hardest problems in NP.
 - So the NP-complete problems form a set of problems that may or may not be intractable but, whether intractable or not, are all, in some sense, of equivalent complexity.
 - If anyone ever shows that an NP-complete problem is tractable, then
 - every NP-complete problem is also tractable
 - indeed, every problem in NP is tractable and so $P = \text{NP}$.

and so $P = \text{NP}$.

- If anyone ever shows that an NP-complete problem is intractable, then
 - every NP-complete problem is also intractableand, of course, $P \neq NP$.
- So there are two possibilities:



We don't know which of these is the case.

- But this gives Computer Scientists a clear line of attack. It makes sense to focus efforts on the NP-complete problems: they all stand or fall together.

- So these sound like very significant problems in our theory. But how would you show that a decision problem is NP-complete?

- How to show a problem P_i is NP-complete (Method 1, from the definition)

- First, confirm that P_i is a decision problem.

- Then show P_i is in NP.

- Then show that P_i is NP-hard by showing that every problem P_j in NP is polynomial-time reducible to P_i .

- * You wouldn't do this one by one!

- * You would try to make a general argument.



The “first” NP-complete problem

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be “reduced” to the problem of determining whether a given propositional formula is a tautology. Here “reduced” means, roughly speaking, that the first problem can be solved deterministically in polynomial time provided an oracle is available for solving the second. From this notion of reducible, polynomial degrees of difficulty are defined, and it is shown that the problem of determining tautologyhood has the same polynomial degree as the problem of determining whether the first of two given graphs is isomorphic to a subgraph of the second. Other examples are discussed. A method of measuring the complexity of proof procedures for the predicate calculus is introduced and discussed.

Throughout this paper, a set of strings means a set of strings on some fixed, large, finite alphabet Σ . This alphabet is large enough to include symbols for all sets described here. All Turing machines are deterministic recognition devices, unless the contrary is explicitly stated.

1. Tautologies and Polynomial Reducibility.

Let us fix a formalism for the propositional calculus in which formulas are written as strings on Σ . Since we will require infinitely many proposition symbols (atoms), each such symbol will consist of a member of Σ followed by a number in binary notation to distinguish that symbol. Thus a formula of length n can only have about $n/\log n$ distinct function and predicate symbols. The logical connectives are \wedge (and), \vee (or), and \neg (not).

The set of tautologies (denoted by $\{ \text{tautologies} \}$) is a

certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will give evidence that (tautologies) is a difficult set to recognize, since many apparently difficult problems can be reduced to determining tautologyhood. By reduced we mean, roughly speaking, that if tautologyhood could be decided instantly (by an “oracle”) then these problems could be decided in polynomial time. In order to make this notion precise, we introduce query machines, which are like Turing machines with oracles in [1].

A query machine is a multitape Turing machine with a distinguished tape called the query tape, and three distinguished states called the query state, yes state, and no state, respectively. If M is a query machine and T is a set of strings, then a T -computation of M is a computation of M in which initially M is in the initial state and has an input string w on its input tape, and each time M assumes the query state there is a string u on the query tape, and the next state M assumes is the yes state if $u \in T$ and the no state if $u \notin T$. We think of an “oracle”, which knows T , placing M in the yes state or no state.

Definition

A set S of strings is P-reducible (P for polynomial) to a set T of strings iff there is some query machine M and a polynomial $Q(n)$ such that for each input string w , the T -computation of M with input w halts within $Q(|w|)$ steps ($|w|$ is the length of w), and ends in an accepting state iff $w \in S$.

It is not hard to see that P -reducibility is a transitive relation. Thus the relation E on

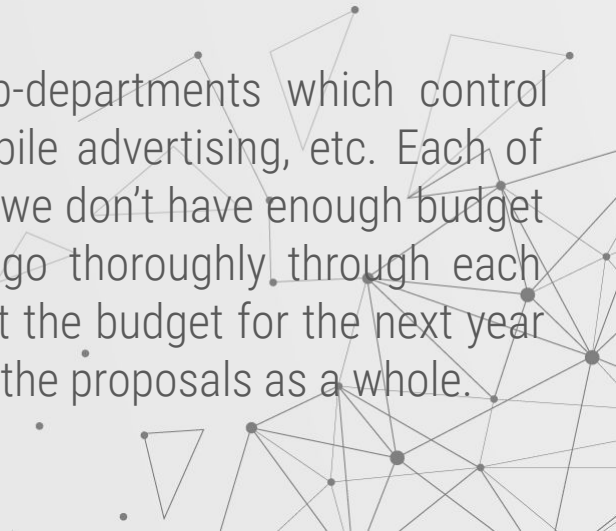
Theorem. [Cook 1971, Levin 1973] $\text{SAT} \in \text{NP-complete}$.

Advertisement Budget Allocation :

Problem Introduction :

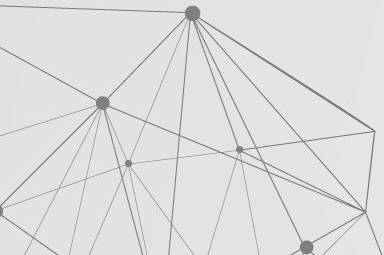
Suppose we are working for a big company that uses advertising to promote itself. We will have to determine whether it is possible to allocate advertising budget and satisfy all the constraints. We will have to reduce this problem to a particular type of Integer Linear Programming problem. Then we will design and implement an efficient algorithm to reduce this type of Integer Linear Programming to SAT.

The marketing department of our big company has many sub-departments which control advertising on TV, radio, web search, contextual advertising, mobile advertising, etc. Each of them has prepared their advertising campaign plan, and of course we don't have enough budget to cover all of their proposals. We don't have enough time to go thoroughly through each sub-department's proposals and cut them, because we need to set the budget for the next year tomorrow. We decide that we will either approve or decline each of the proposals as a whole.



There is a bunch of constraints we face. All of these constraints can be rewritten as an Integer Linear Programming: for each subdepartment i , denote by x_i boolean variable that corresponds to whether we will accept or decline the proposal of that sub-department. Then each constraint can be written as a linear inequality.

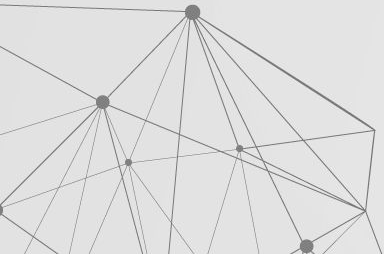
For example, $\sum_{i=1}^n \text{spend}_i \cdot x_i \leq \text{Total Budget}$ is the inequality to ensure your total budget is enough to accept all the selected proposals. And $\sum_{i=1}^n \text{spend}_i \cdot x_i \leq 10 \cdot \text{mobile}$ corresponds to the fact that mobile advertisement budget is at least 10% of the total spending. You will be given the final Integer Linear Programming problem in the input, and you will need to reduce it to SAT. It is guaranteed that there will be at most 3 different variables with non-zero coefficients in each inequality of this Integer Linear Programming problem.



Input Format :

The first line contains two integers n and m – the number of inequalities and the number of variables. The next n lines contain the description of $n \times m$ matrix A with coefficients of inequalities (each of the n lines contains m integers, and at most 3 of them are non-zero), and the last line contains the description of the vector b (n integers) for the system of inequalities $Ax \leq b$. You need to determine whether there exists a binary vector x satisfying all those inequalities.

Constraints - $1 \leq n, m \leq 500$; $-100 \leq A_{ij} \leq 100$; $-1\,000\,000 \leq b_i \leq 1\,000\,000$



Output Format :

We need to output a boolean formula in the CNF form in a specific format. If it is possible to accept some of the proposals and decline all the others while satisfying all the constraints, the formula must be satisfiable. Otherwise, the formula must be unsatisfiable. The number of variables in the formula must not exceed 3000, and the number of clauses must not exceed 5000. On the first line, output integers C and V – the number of clauses in the formula and the number of variables respectively. On each of the next C lines, output a description of a single clause. Each clause has a form $(x4 \text{ OR } x1 \text{ OR } x8)$. For a clause with k terms (in the example, $k = 3$ for $x4$, $x1$ and $x8$), output first those k terms and then number 0 in the end (in the example, output “4 - 1 8 0”). Output each term as integer number. Output variables $x1, x2, \dots, xV$ as numbers $1, 2, \dots, V$ respectively. Output negations of variables $x1, x2, \dots, xV$ as numbers $-1, -2, \dots, -V$ respectively. Each number other than the last one in each line must be a non-zero integer between $-V$ and V where V is the total number of variables specified in the first line of the output. We will have to ensure that $1 \leq C \leq 5000$ and $1 \leq V \leq 3000$.



Example 1 :

Input:


2 3
5 2 3
-1 -1 -1
6 -2

Output:

1 1 1 -1 0

Explanation:

Here we have two inequalities: $5 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 \leq 6$ and $(-1) \cdot x_1 + (-1) \cdot x_2 + (-1) \cdot x_3 \leq -2$. The binary vector $x_1 = 0, x_2 = 1, x_3 = 1$ satisfies all the inequalities, so we need to output a satisfiable formula. The formula in the output uses just one variable x_1 and one clause, and the only clause is $(x_1 \text{ OR } x_1)$ which is, of course, satisfiable: for any value of x_1 , the boolean value of the formula is true. Note that you could output another satisfiable formula, like x_1 or $(x_1 \text{ OR } x_2 \text{ OR } x_3) \text{ AND } (x_1 \text{ OR } x_2)$, or one of many others. You do not have to make sure that the formula you output is satisfied only by the binary vectors which satisfy the given system of inequalities, but the intended solution ensures that.



Example 2 :

Input:

2 1
1
-1
0 -1

Output:

2 1
1 0
-1 0

Explanation:

There are two inequalities $x_1 \leq 0$ and $-x_1 \leq -1 \Rightarrow x_1 \geq 1$, but x_1 cannot be less than 0 and more than 1 simultaneously, so we need to output an unsatisfiable formula. The formula in the output has 2 clauses with one variable, it is $(x_1) \text{ AND } (x_1)$. Note that we could output another formula, like $(x_1 \text{ OR } x_2) \text{ AND } (x_1) \text{ AND } (x_2)$, or one of many other unsatisfiable formulas.

CODE :

```
from sys import stdin

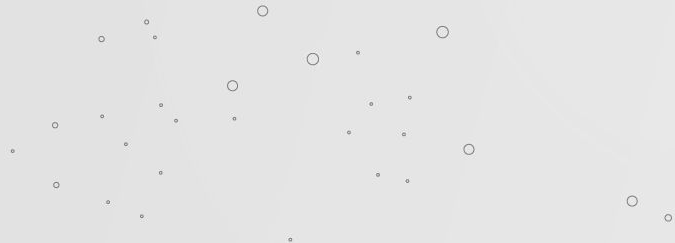
n, m = list(map(int, stdin.readline().split()))

A = []

for i in range(n):
    A += [list(map(int, stdin.readline().split()))]

b = list(map(int, stdin.readline().split()))

one_var = [[0], [1]]
two_var = [[0,0], [0,1], [1,0], [1,1]]
three_var = [[0,0,0], [0,0,1], [0,1,0], [1,0,0], [0,1,1], [1,1,0], [1,0,1], [1,1,1]]
```



```
def printEquisatisfiableSatFormula():
    clauses = []
    for i in range(n):
        check_line(A[i], b[i], clauses)

    var_list = count_var(clauses)
    if len(clauses):
        print(str(len(clauses)) + ' ' + str(len(var_list)))
        for cl in clauses:
            text=''
            for num in cl:
                if num<0:
                    text += '-' + str(var_list.index(abs(num)) + 1) + ' '
                else:
                    text += str(var_list.index(num) + 1) + ' '
            print(text + '0')
        else:
            print('1 1')
            print('1 -1 0')
```



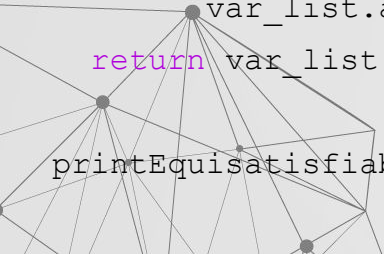
```
def check_line(line, value, clauses):
    coe_index= []
    for i in range(m):
        if line[i] != 0:
            coe_index.append(i)
    if len(coe_index) == 1:
        compare_result(line, coe_index, one_var, value, clauses)
    elif len(coe_index) == 2:
        compare_result(line, coe_index, two_var, value, clauses)
    elif len(coe_index) == 3:
        compare_result(line, coe_index, three_var, value, clauses)

def compare_result(line, coe_index, test_case, value, clauses):
    for test in test_case:
        result=0
        for i in range(len(coe_index)):
            result += line[coe_index[i]]*test[i]
        if result > value:
            add_boolean(coe_index, test, clauses)
```

```
def add_boolean(coe_index, test, clauses):
    cl = []
    for i in range(len(test)):
        if test[i] == 0:
            cl.append(coe_index[i] + 1)
        else:
            cl.append(-coe_index[i] - 1)
    clauses.append(cl)
```

```
def count_var(clauses):
    var_list = []
    for cl in clauses:
        for v in cl:
            if abs(v) not in var_list:
                var_list.append(abs(v))
    return var_list
```

```
printEquisatisfiableSatFormula()
```



THANK YOU



VIDHI KAPOOR
J021



KARTIKAY LADDHA
J025