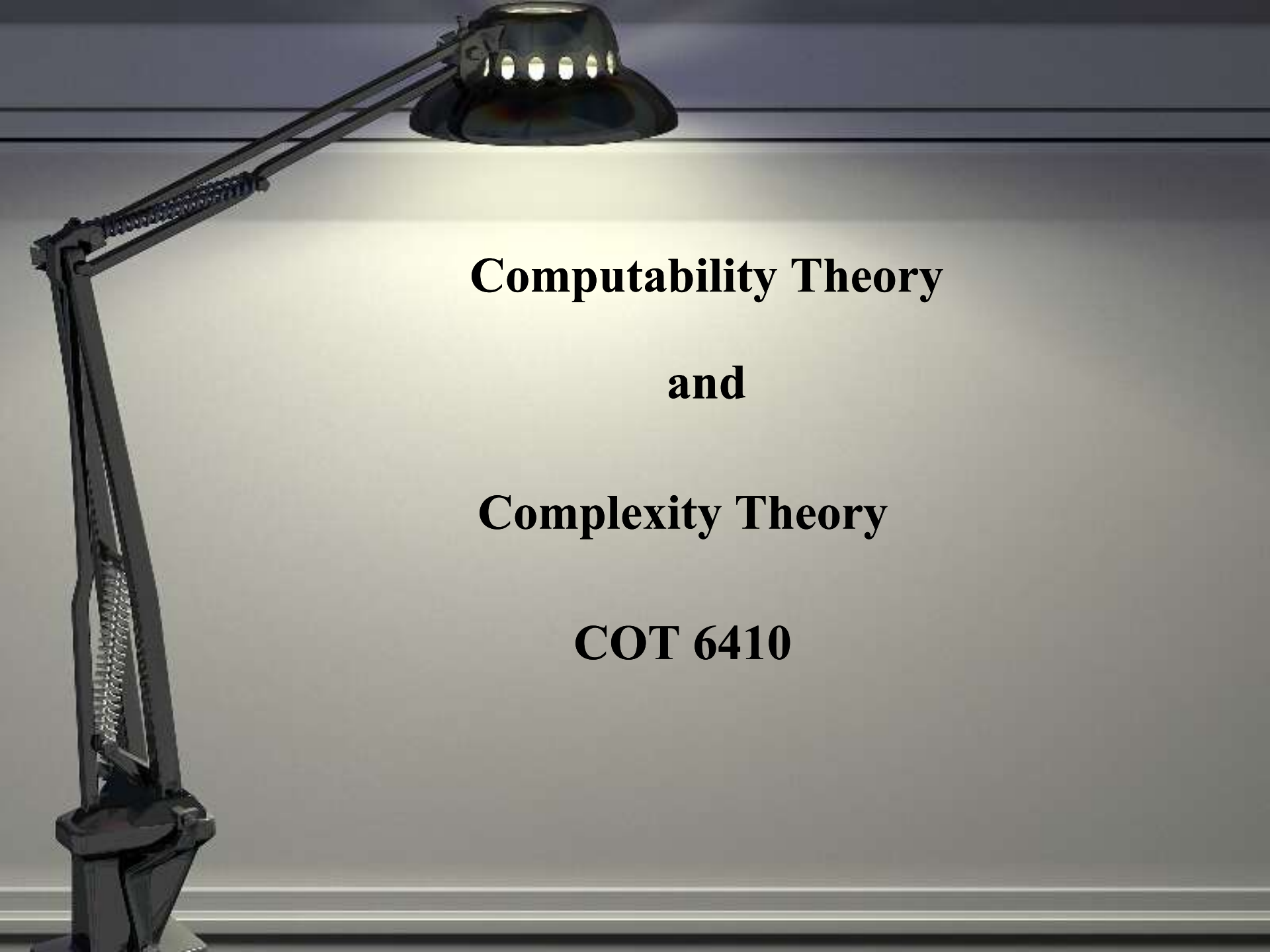
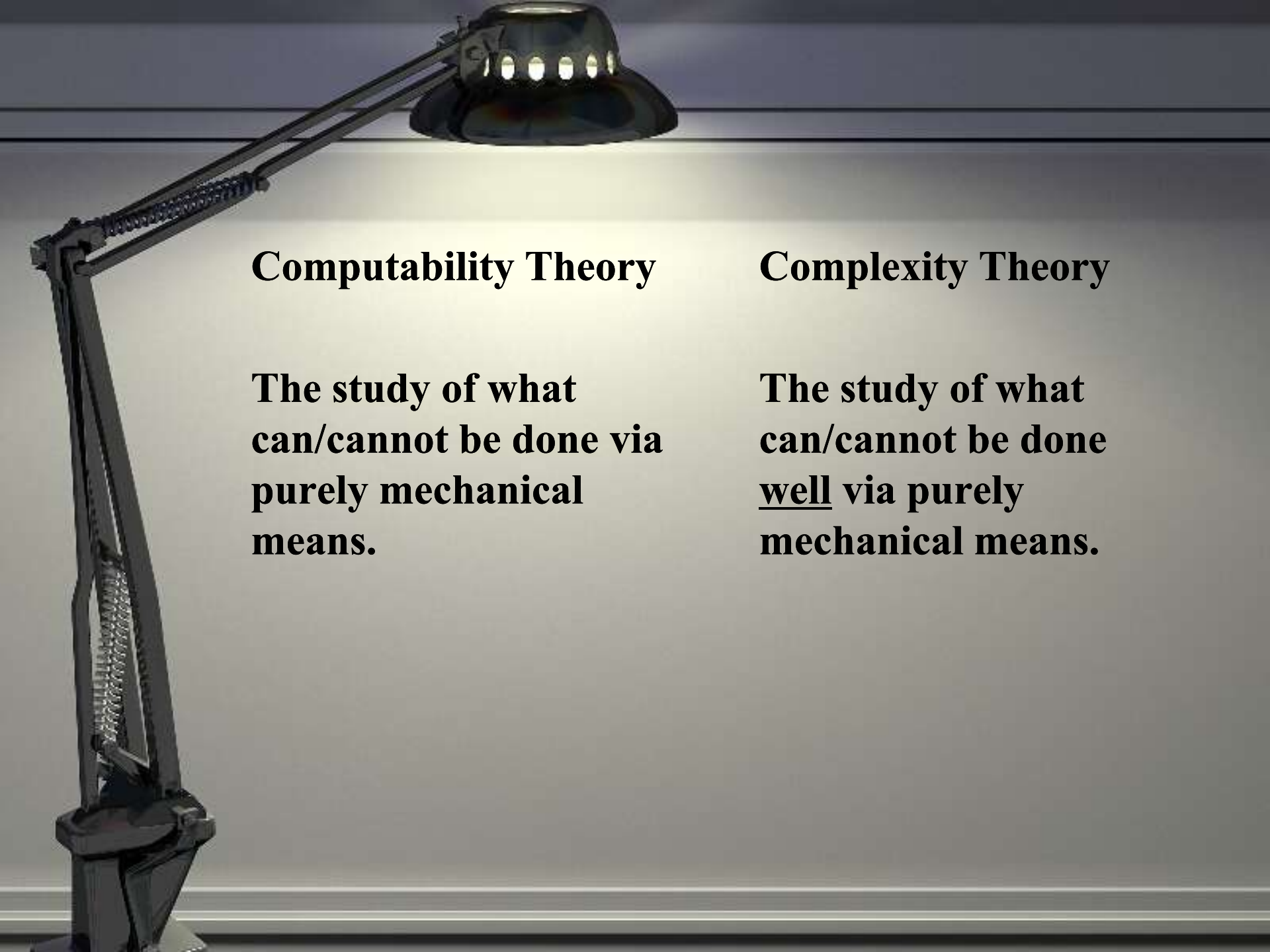

Computability Theory
and
Complexity Theory
COT 6410

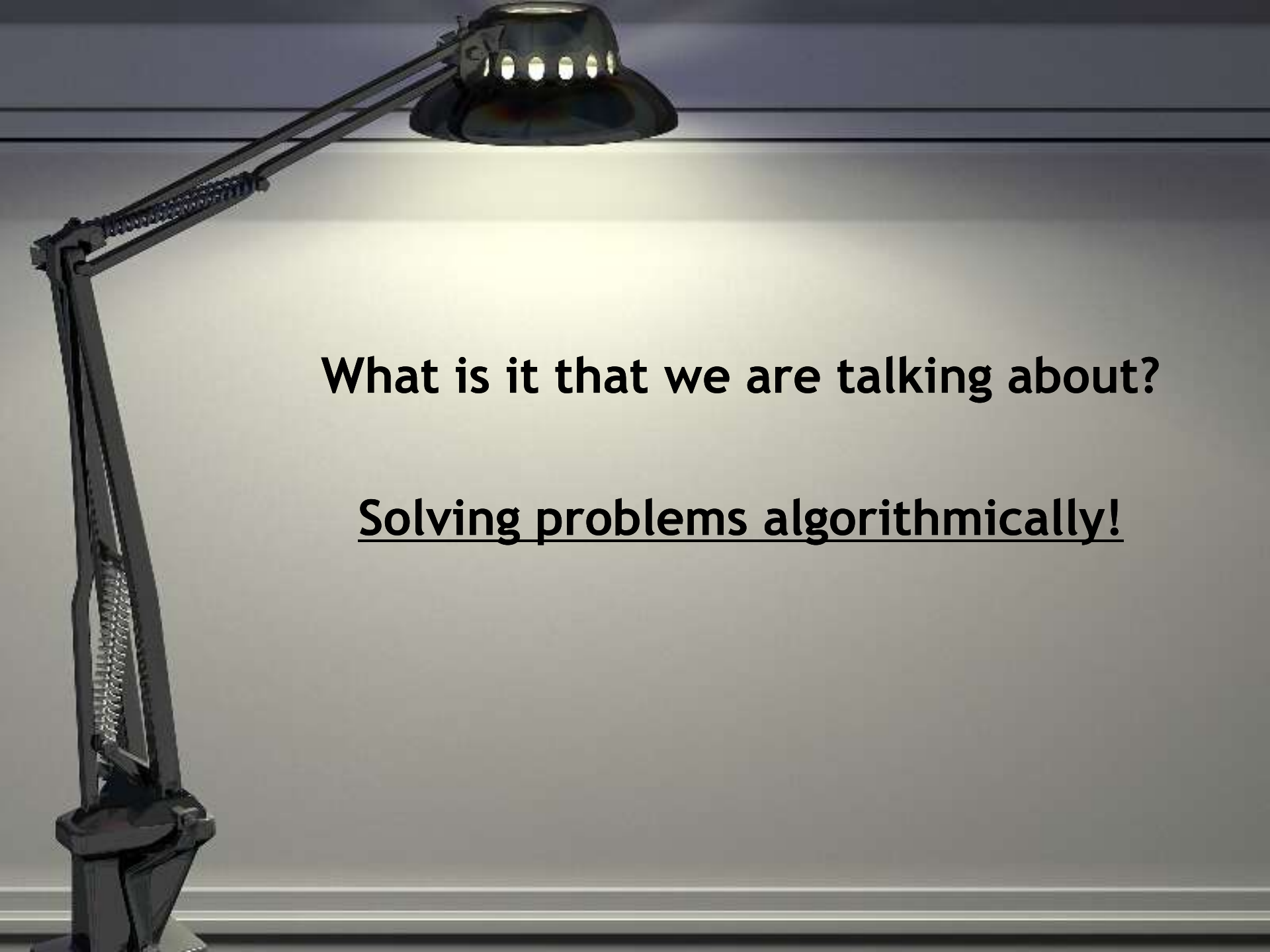


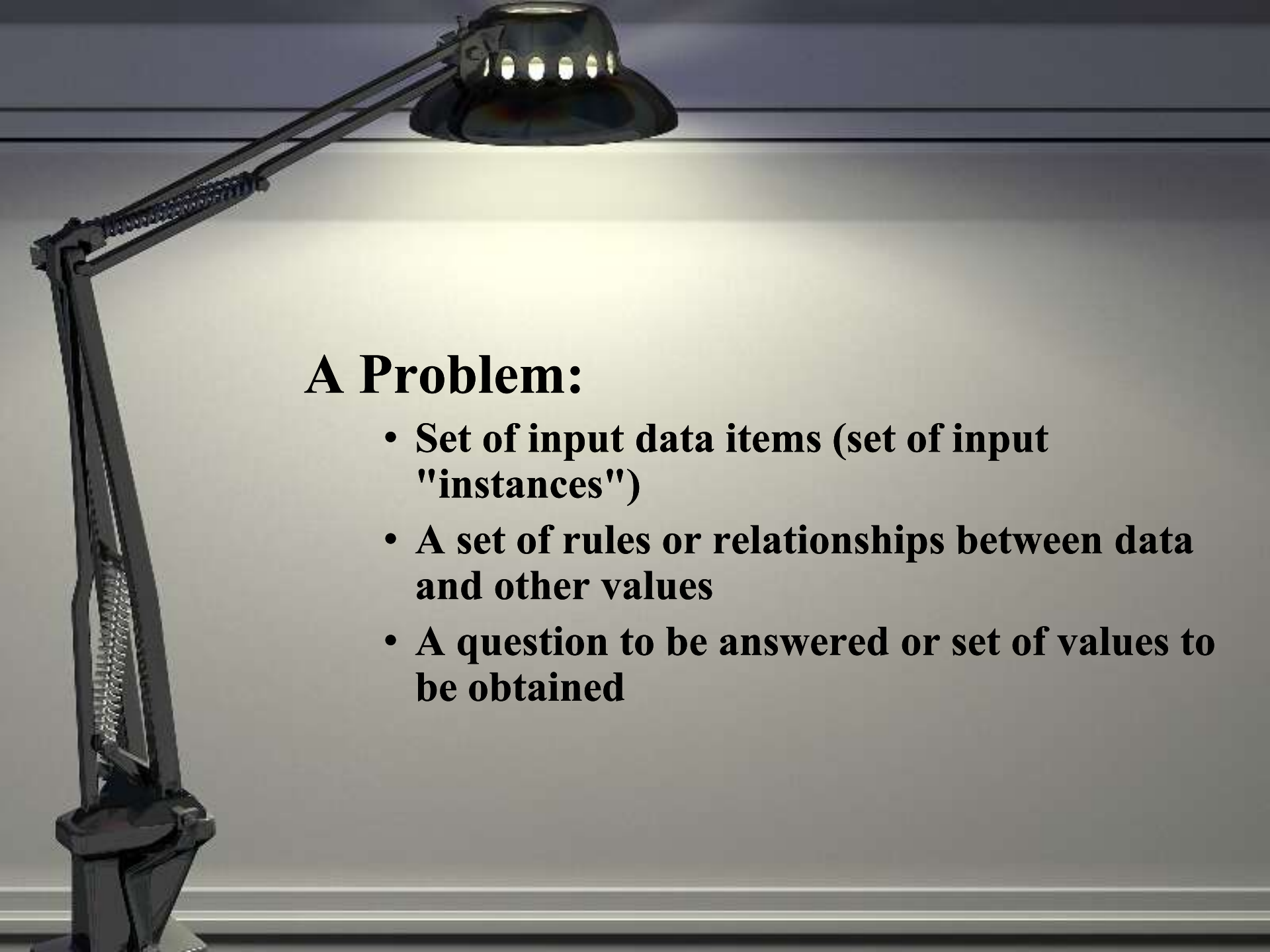
Computability Theory

The study of what can/cannot be done via purely mechanical means.

Complexity Theory

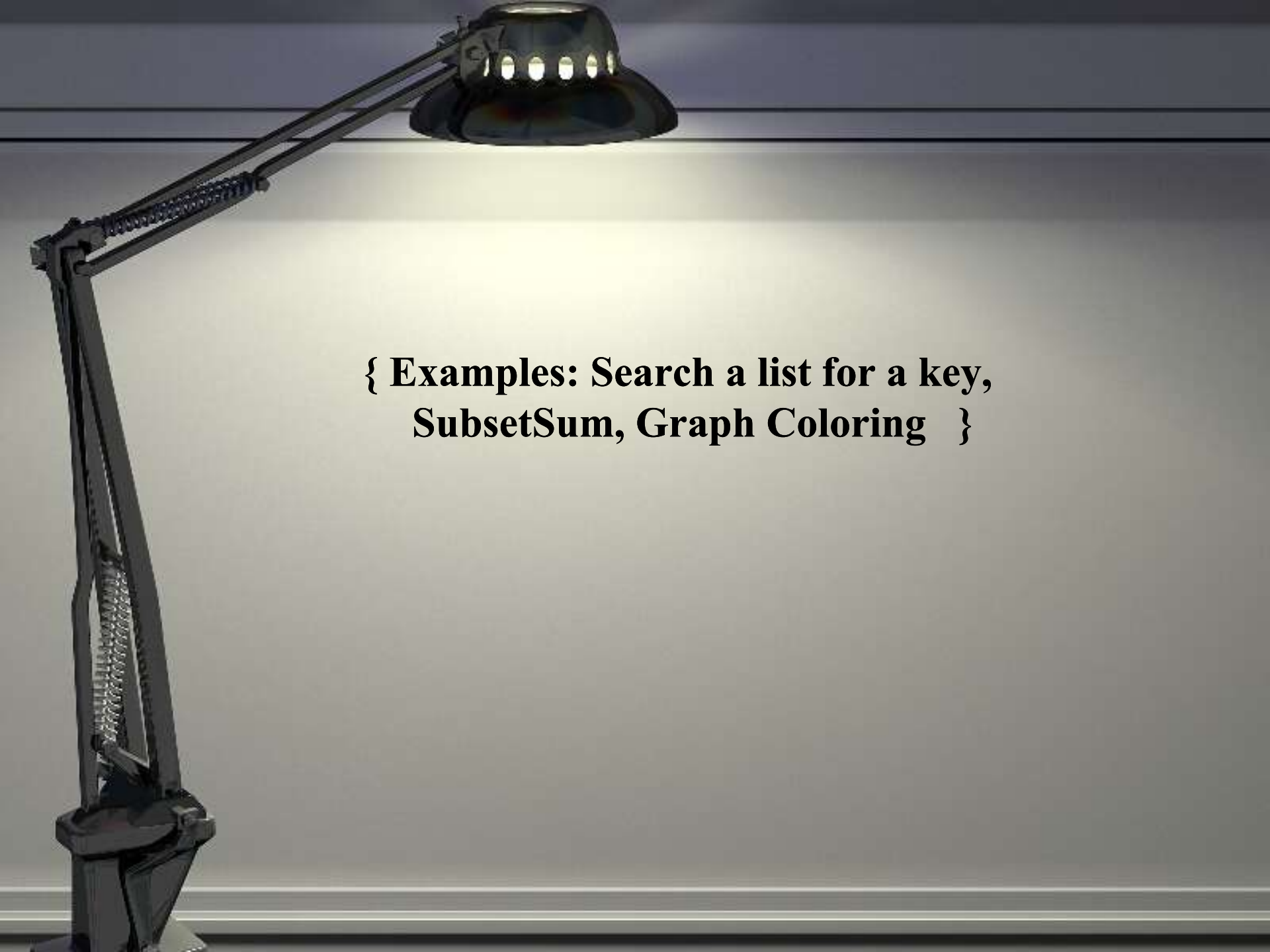
The study of what can/cannot be done <u>well</u> via purely mechanical means.

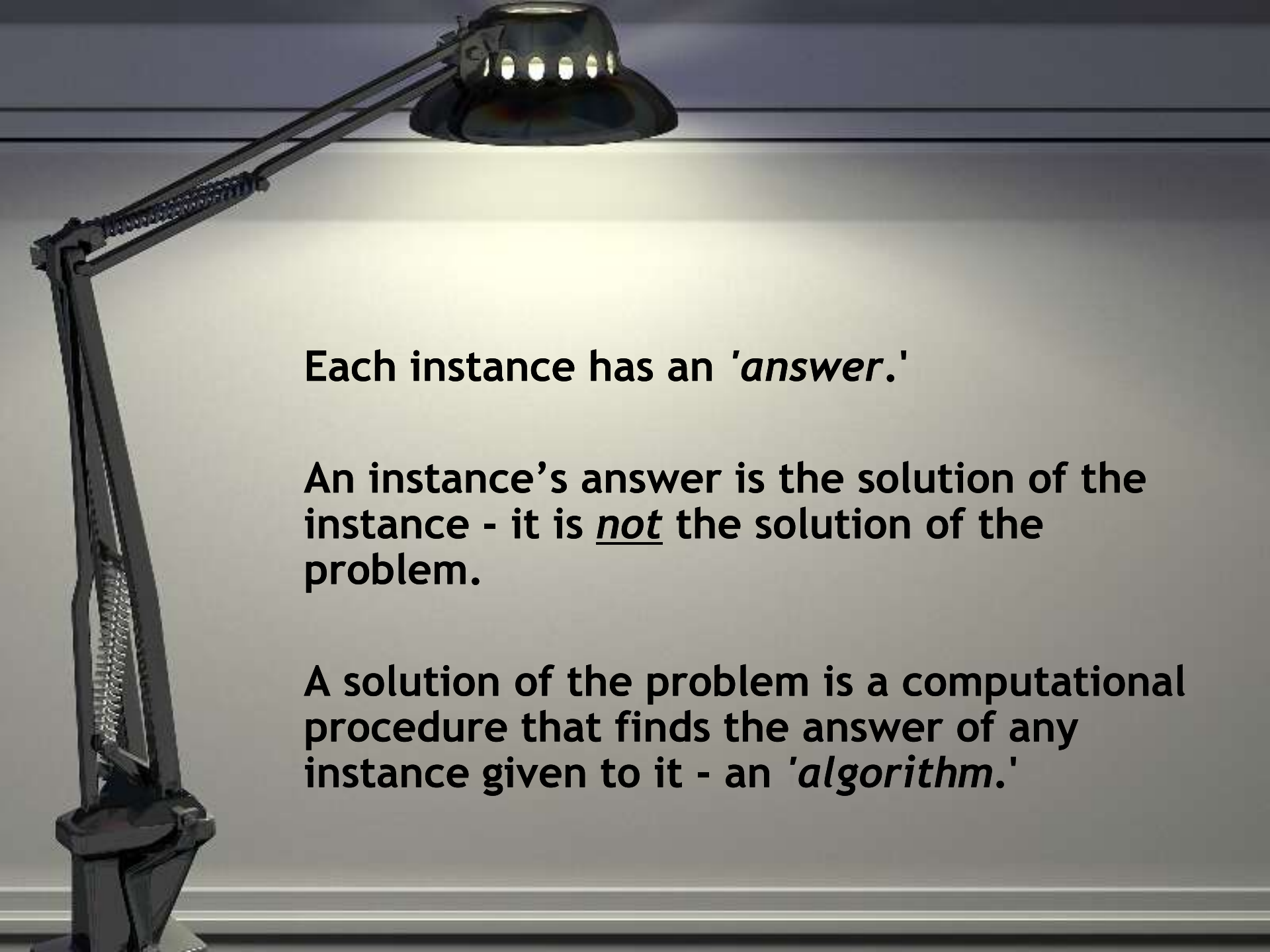

What is it that we are talking about?
Solving problems algorithmically!



A Problem:

- Set of input data items (set of input "instances")
- A set of rules or relationships between data and other values
- A question to be answered or set of values to be obtained

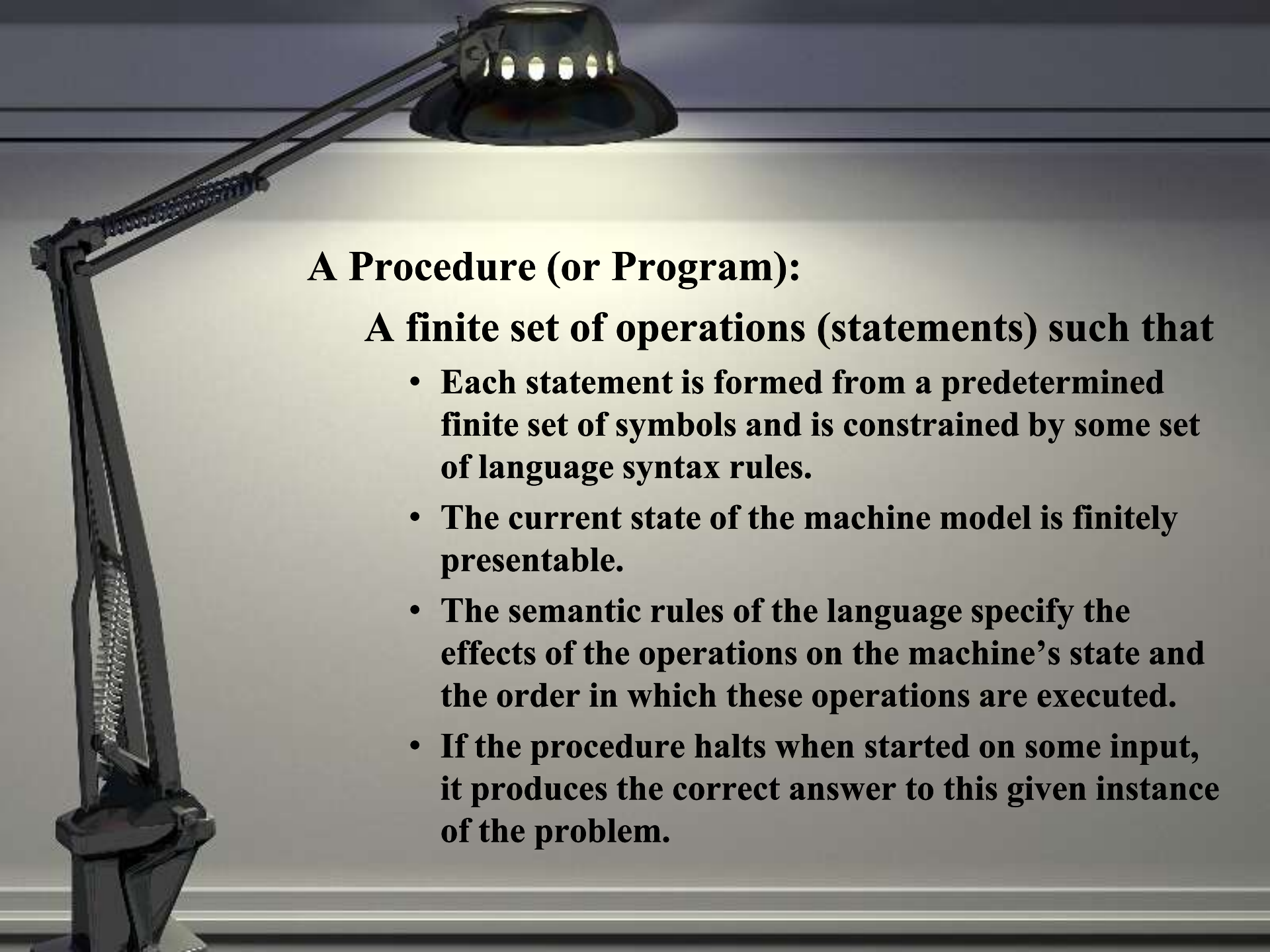

{ Examples: Search a list for a key,
SubsetSum, Graph Coloring }



Each instance has an *'answer.'*

An instance's answer is the solution of the instance - it is ***not*** the solution of the problem.

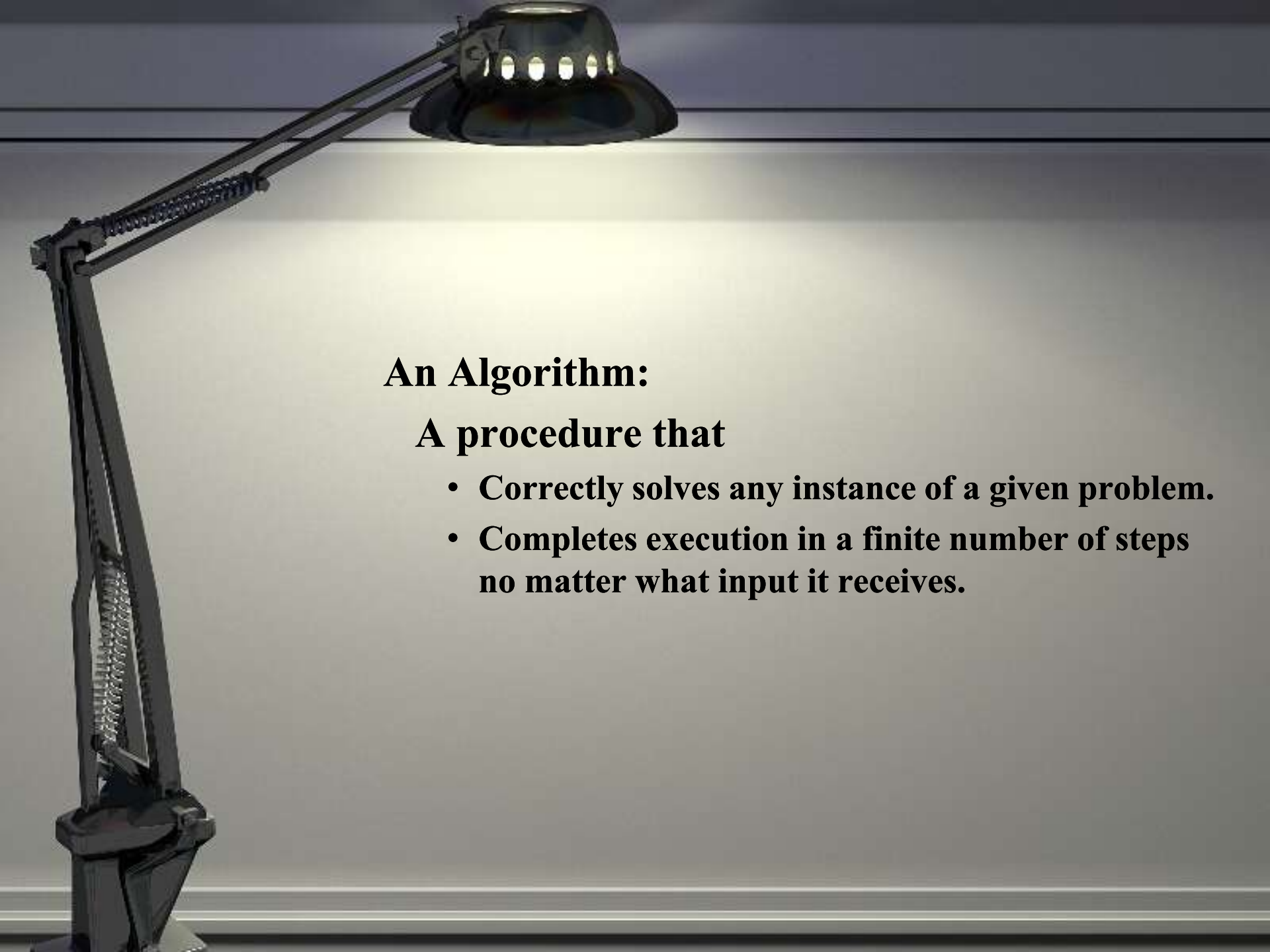
A solution of the problem is a computational procedure that finds the answer of any instance given to it - an *'algorithm.'*



A Procedure (or Program):

A finite set of operations (statements) such that

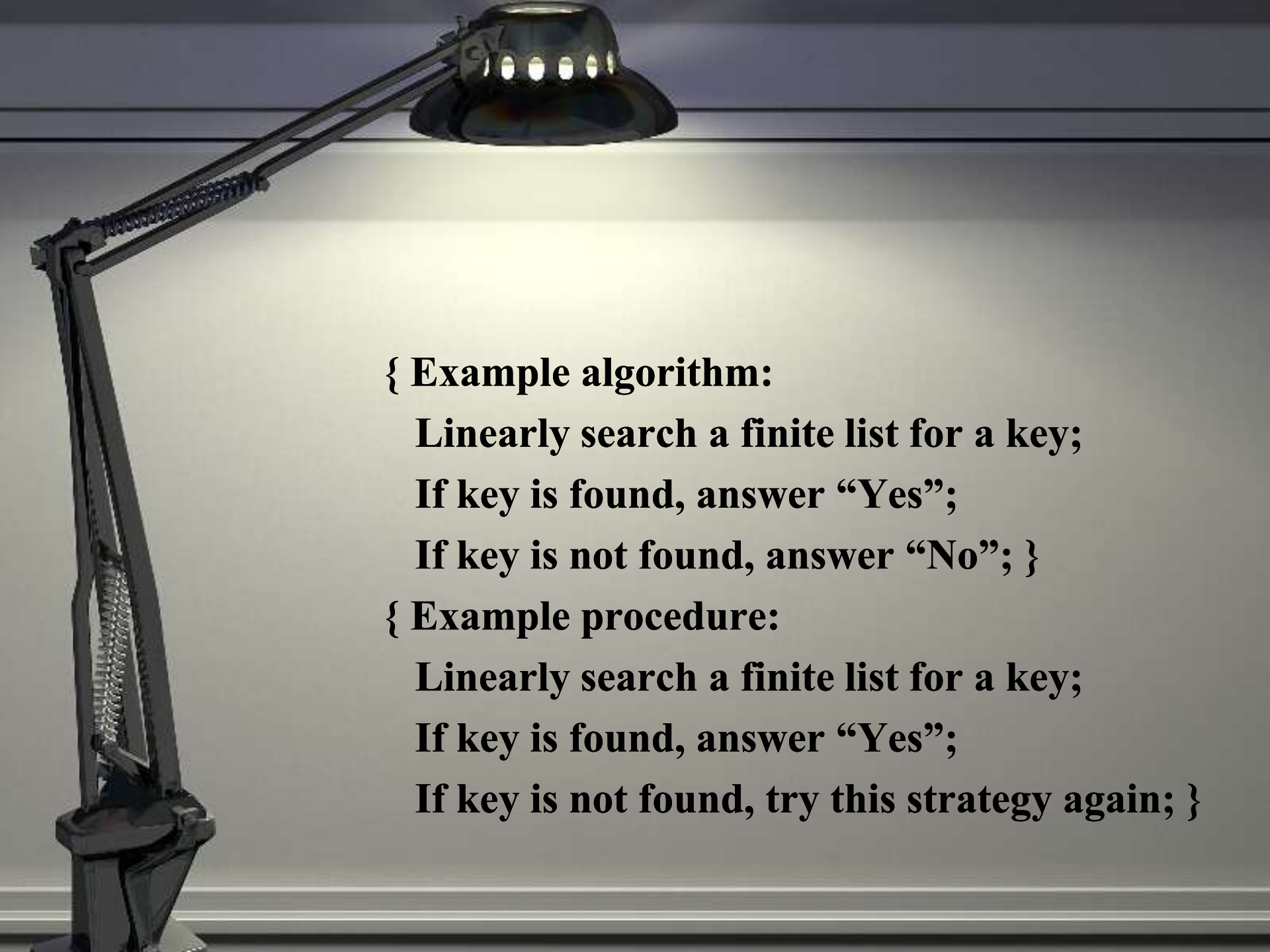
- **Each statement is formed from a predetermined finite set of symbols and is constrained by some set of language syntax rules.**
- **The current state of the machine model is finitely presentable.**
- **The semantic rules of the language specify the effects of the operations on the machine's state and the order in which these operations are executed.**
- **If the procedure halts when started on some input, it produces the correct answer to this given instance of the problem.**



An Algorithm:

A procedure that

- **Correctly solves any instance of a given problem.**
- **Completes execution in a finite number of steps no matter what input it receives.**



{ Example algorithm:

Linearly search a finite list for a key;

If key is found, answer “Yes”;

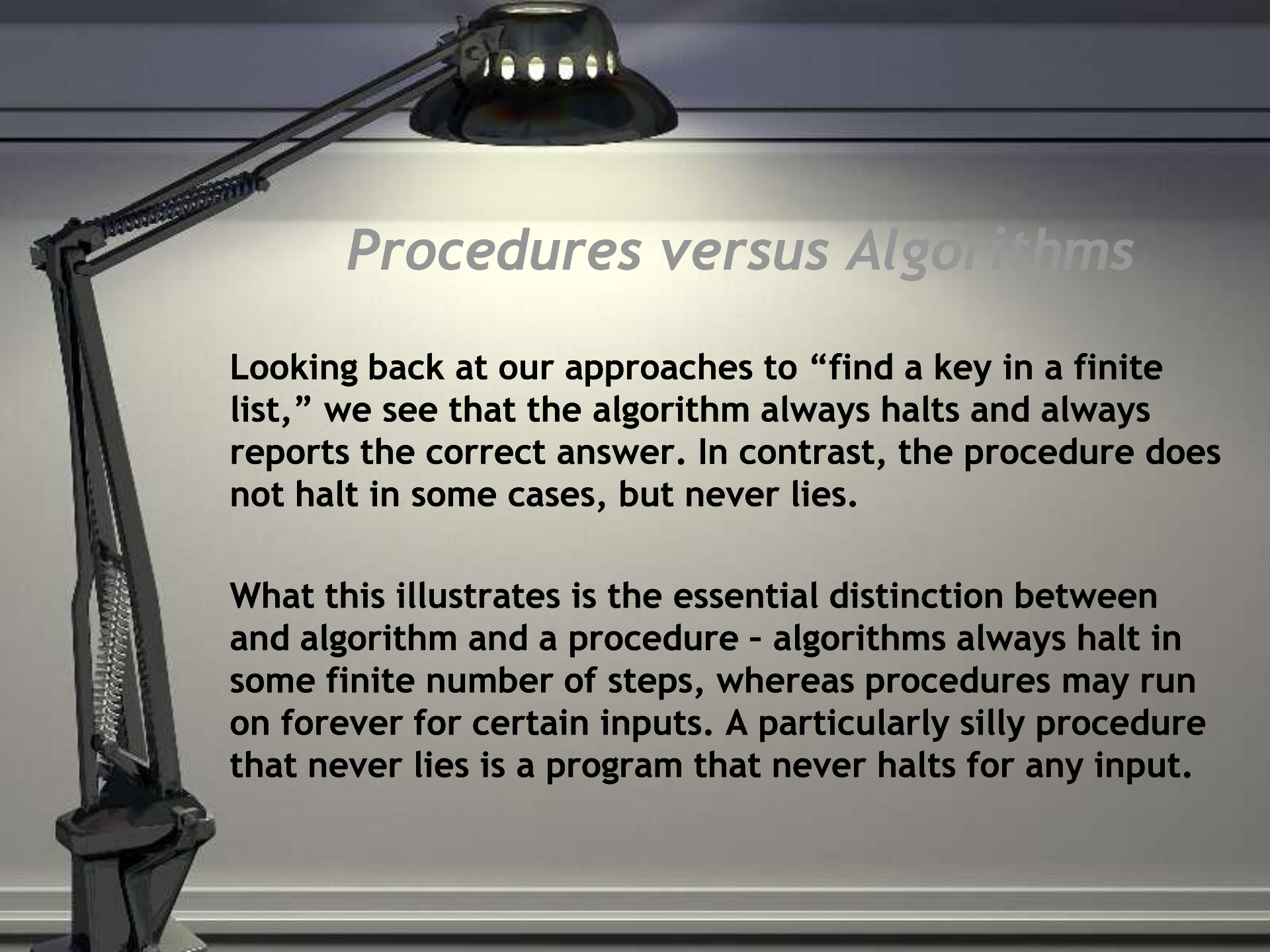
If key is not found, answer “No”; }

{ Example procedure:

Linearly search a finite list for a key;

If key is found, answer “Yes”;

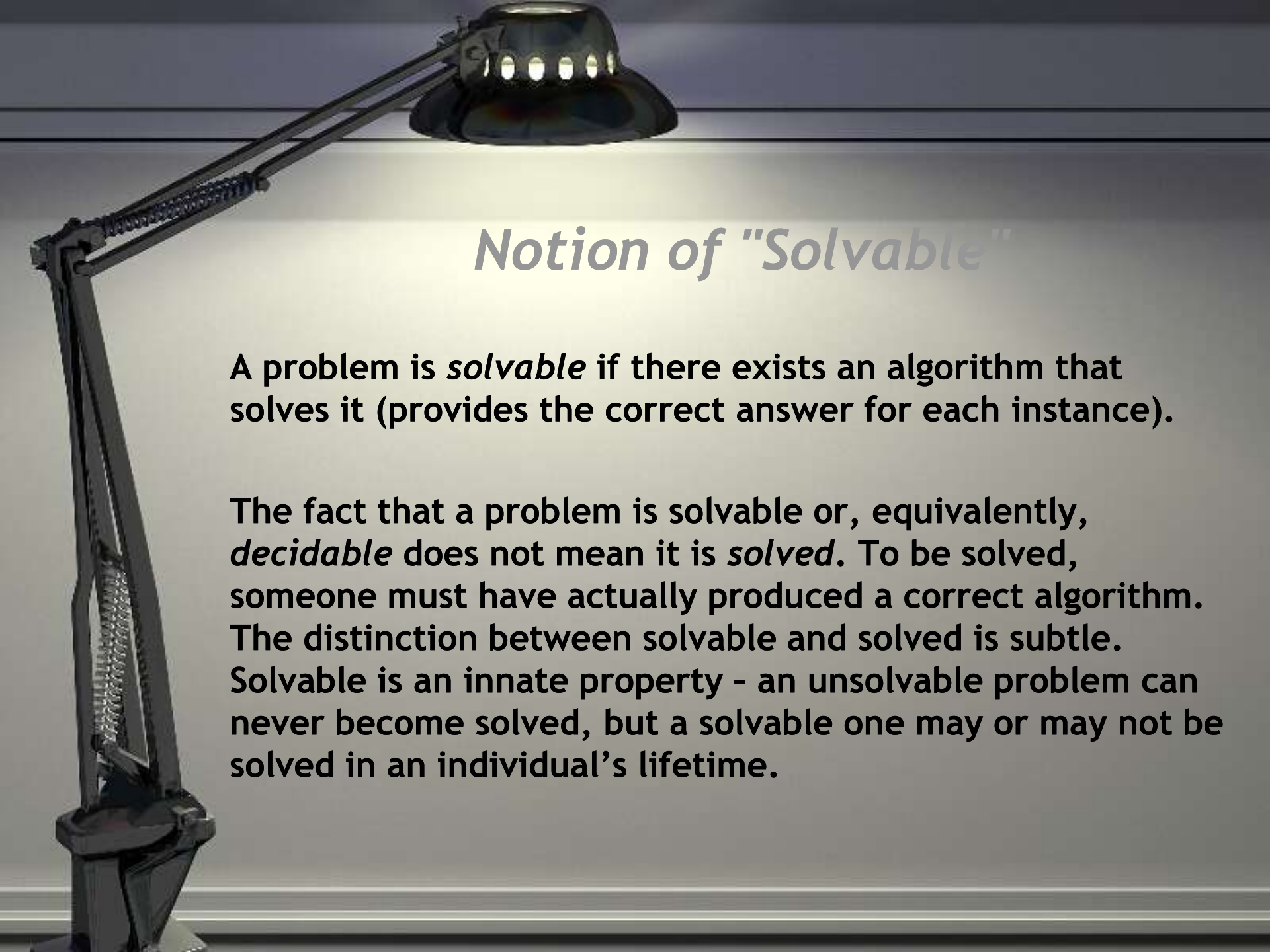
If key is not found, try this strategy again; }



Procedures versus Algorithms

Looking back at our approaches to “find a key in a finite list,” we see that the algorithm always halts and always reports the correct answer. In contrast, the procedure does not halt in some cases, but never lies.

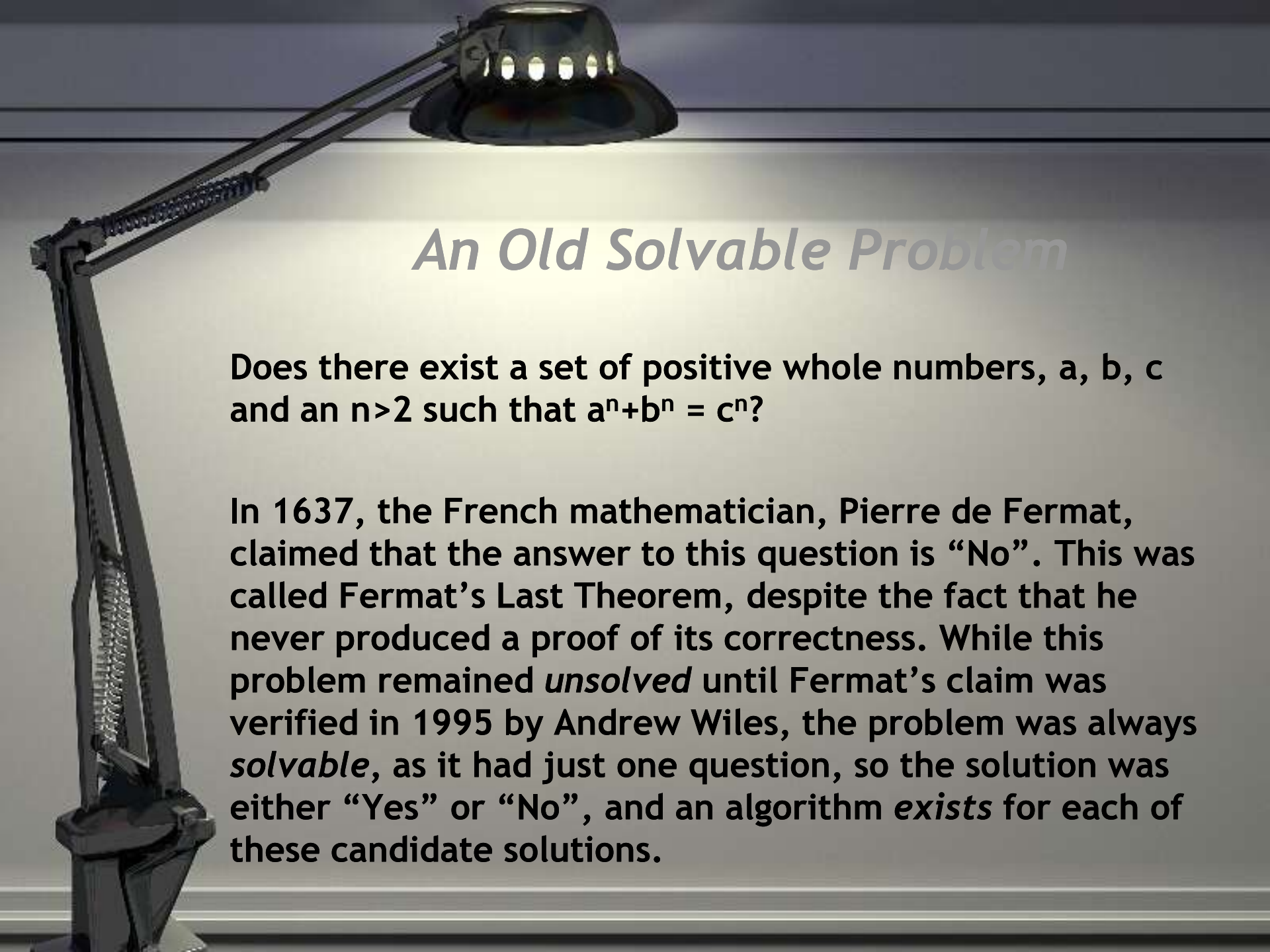
What this illustrates is the essential distinction between and algorithm and a procedure – algorithms always halt in some finite number of steps, whereas procedures may run on forever for certain inputs. A particularly silly procedure that never lies is a program that never halts for any input.



Notion of "Solvable"

A problem is *solvable* if there exists an algorithm that solves it (provides the correct answer for each instance).

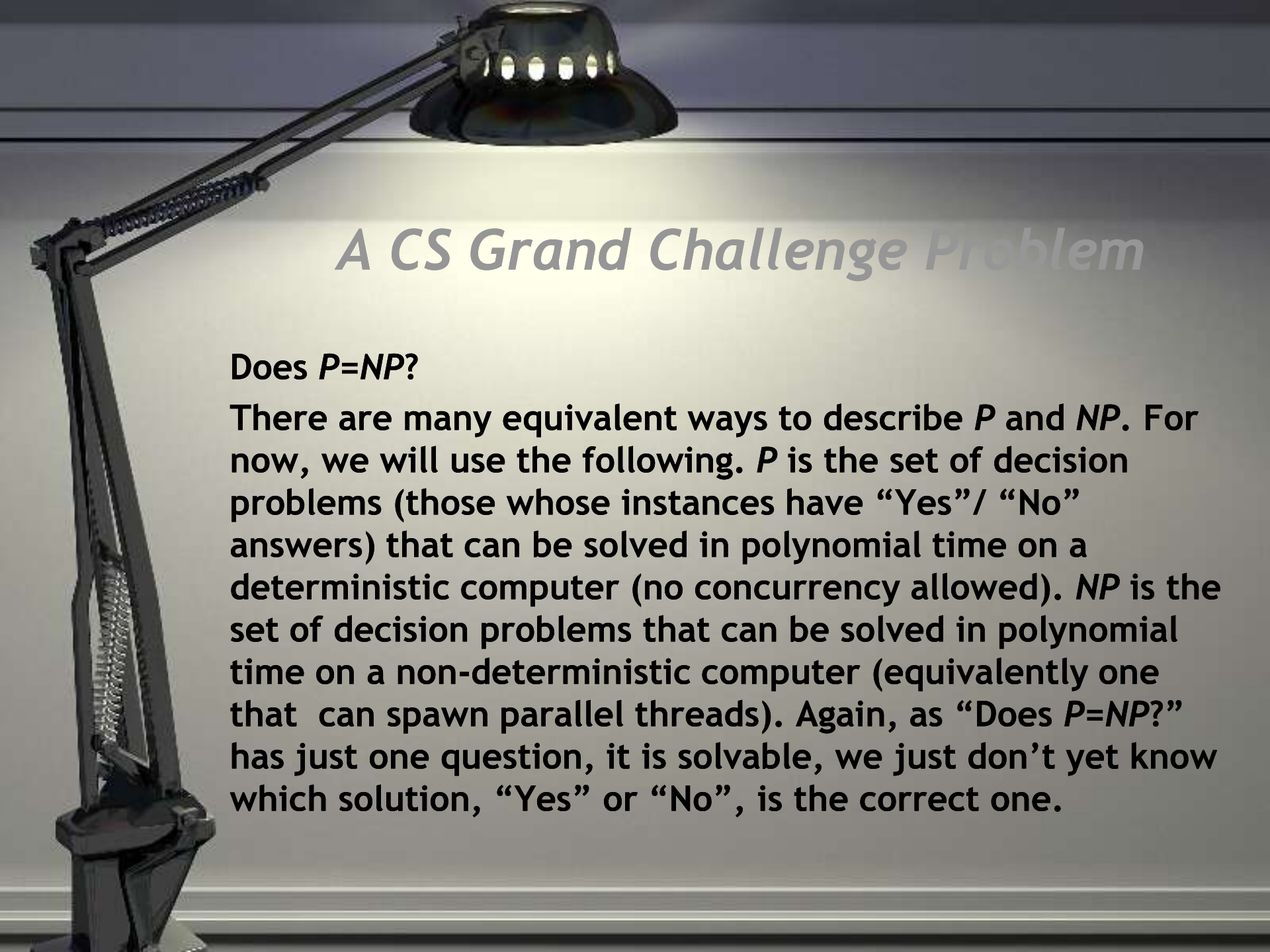
The fact that a problem is solvable or, equivalently, *decidable* does not mean it is *solved*. To be solved, someone must have actually produced a correct algorithm. The distinction between solvable and solved is subtle. Solvable is an innate property – an unsolvable problem can never become solved, but a solvable one may or may not be solved in an individual's lifetime.



An Old Solvable Problem

Does there exist a set of positive whole numbers, a, b, c and an n>2 such that $a^n+b^n = c^n$?

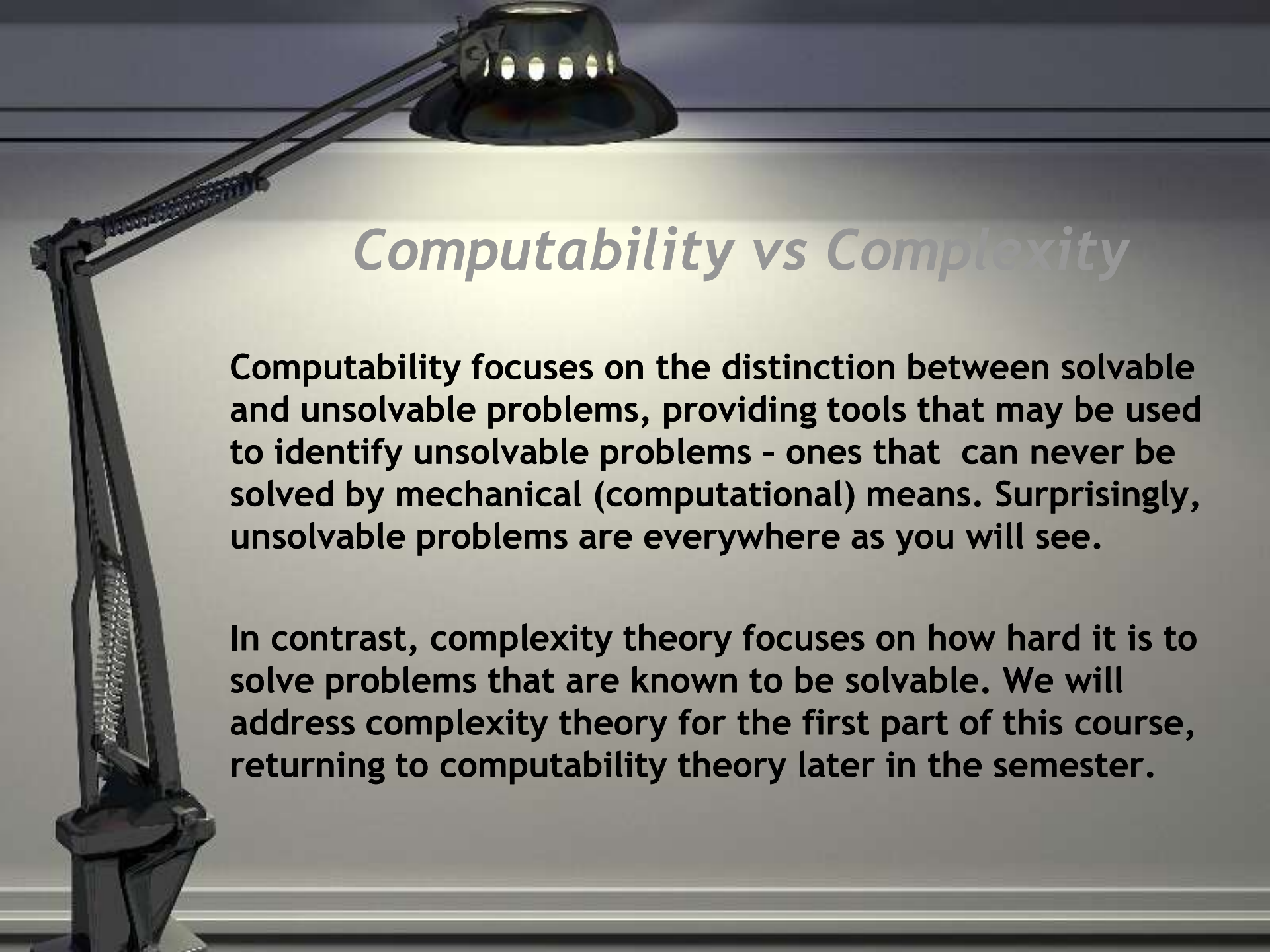
In 1637, the French mathematician, Pierre de Fermat, claimed that the answer to this question is "No". This was called Fermat's Last Theorem, despite the fact that he never produced a proof of its correctness. While this problem remained *unsolved* until Fermat's claim was verified in 1995 by Andrew Wiles, the problem was always *solvable*, as it had just one question, so the solution was either "Yes" or "No", and an algorithm *exists* for each of these candidate solutions.



A CS Grand Challenge Problem

Does *P*=*NP*?

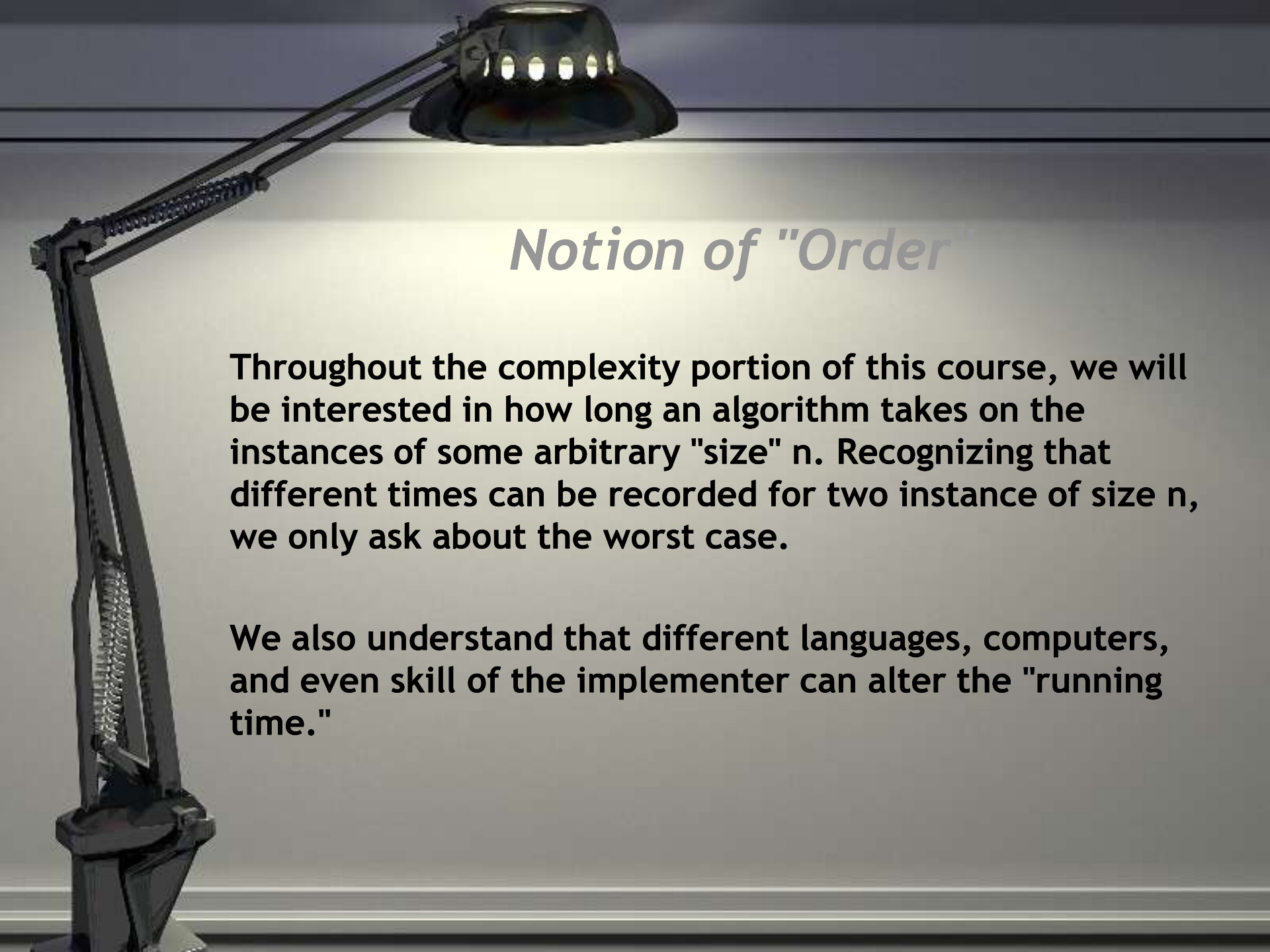
There are many equivalent ways to describe *P* and *NP*. For now, we will use the following. *P* is the set of decision problems (those whose instances have "Yes"/ "No" answers) that can be solved in polynomial time on a deterministic computer (no concurrency allowed). *NP* is the set of decision problems that can be solved in polynomial time on a non-deterministic computer (equivalently one that can spawn parallel threads). Again, as "Does *P*=*NP*?" has just one question, it is solvable, we just don't yet know which solution, "Yes" or "No", is the correct one.



Computability vs Complexity

Computability focuses on the distinction between solvable and unsolvable problems, providing tools that may be used to identify unsolvable problems – ones that can never be solved by mechanical (computational) means. Surprisingly, unsolvable problems are everywhere as you will see.

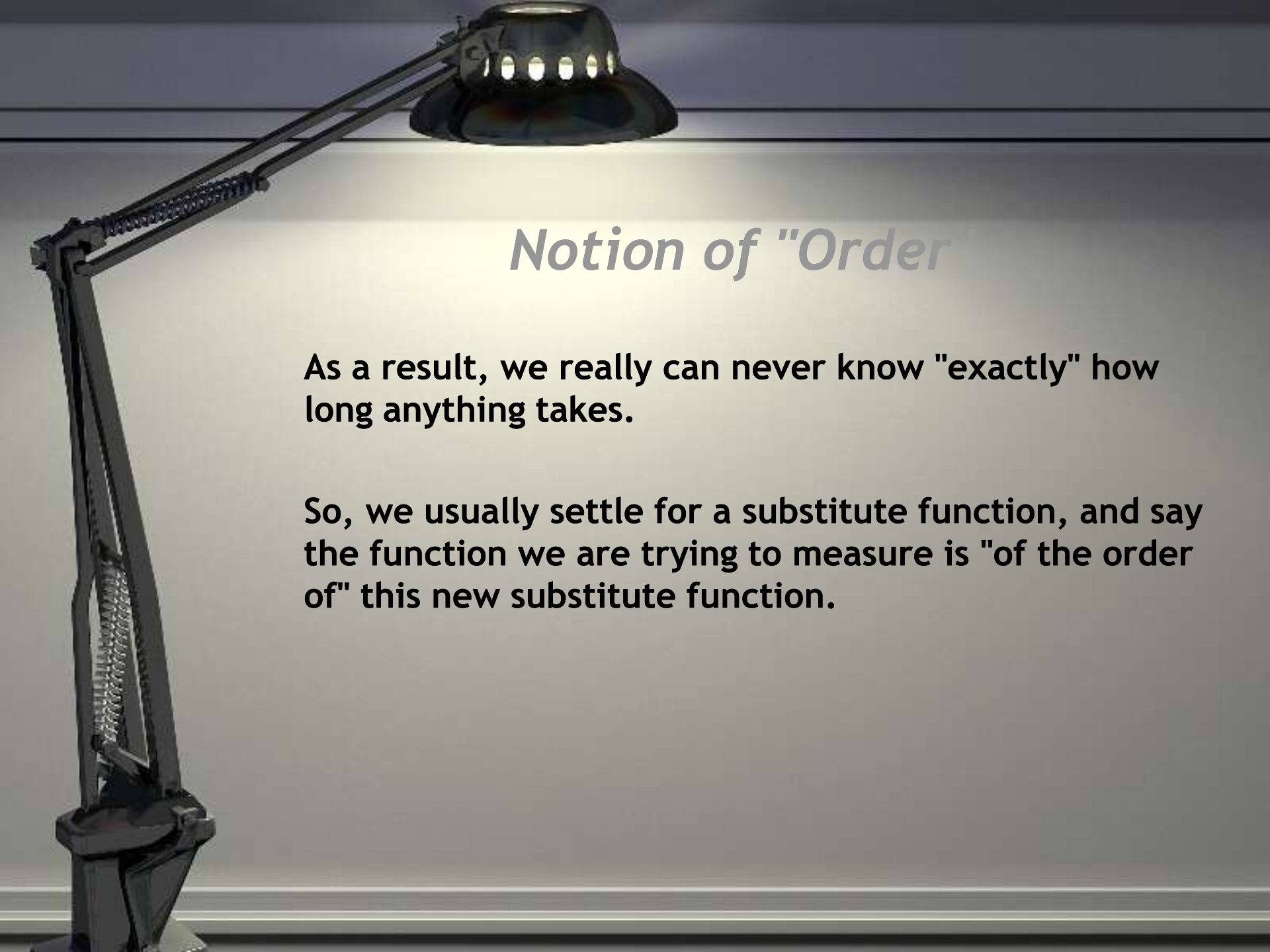
In contrast, complexity theory focuses on how hard it is to solve problems that are known to be solvable. We will address complexity theory for the first part of this course, returning to computability theory later in the semester.



Notion of "Order"

Throughout the complexity portion of this course, we will be interested in how long an algorithm takes on the instances of some arbitrary "size" n. Recognizing that different times can be recorded for two instance of size n, we only ask about the worst case.

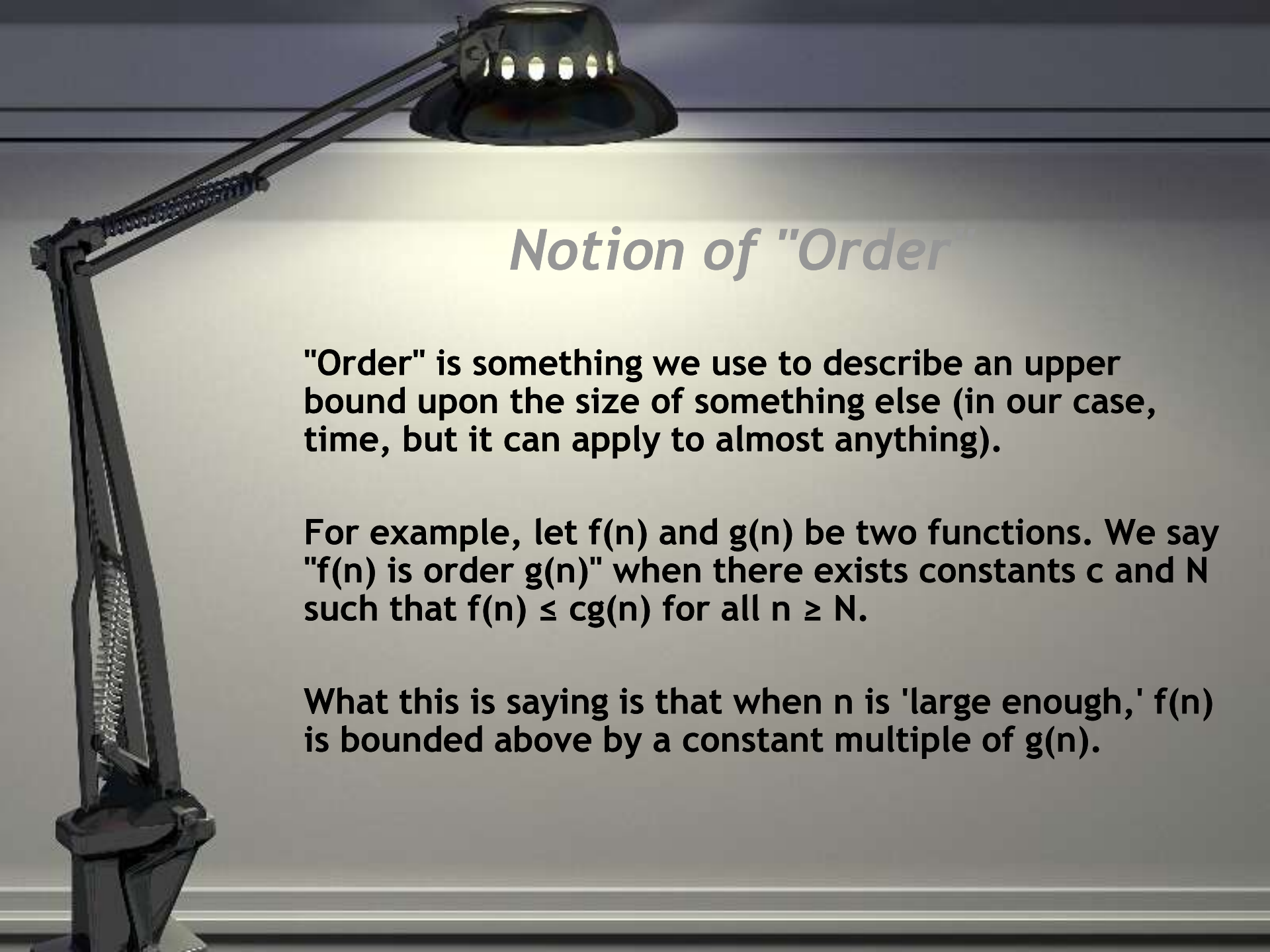
We also understand that different languages, computers, and even skill of the implementer can alter the "running time."



Notion of "Order"

As a result, we really can never know "exactly" how long anything takes.

So, we usually settle for a substitute function, and say the function we are trying to measure is "of the order of" this new substitute function.

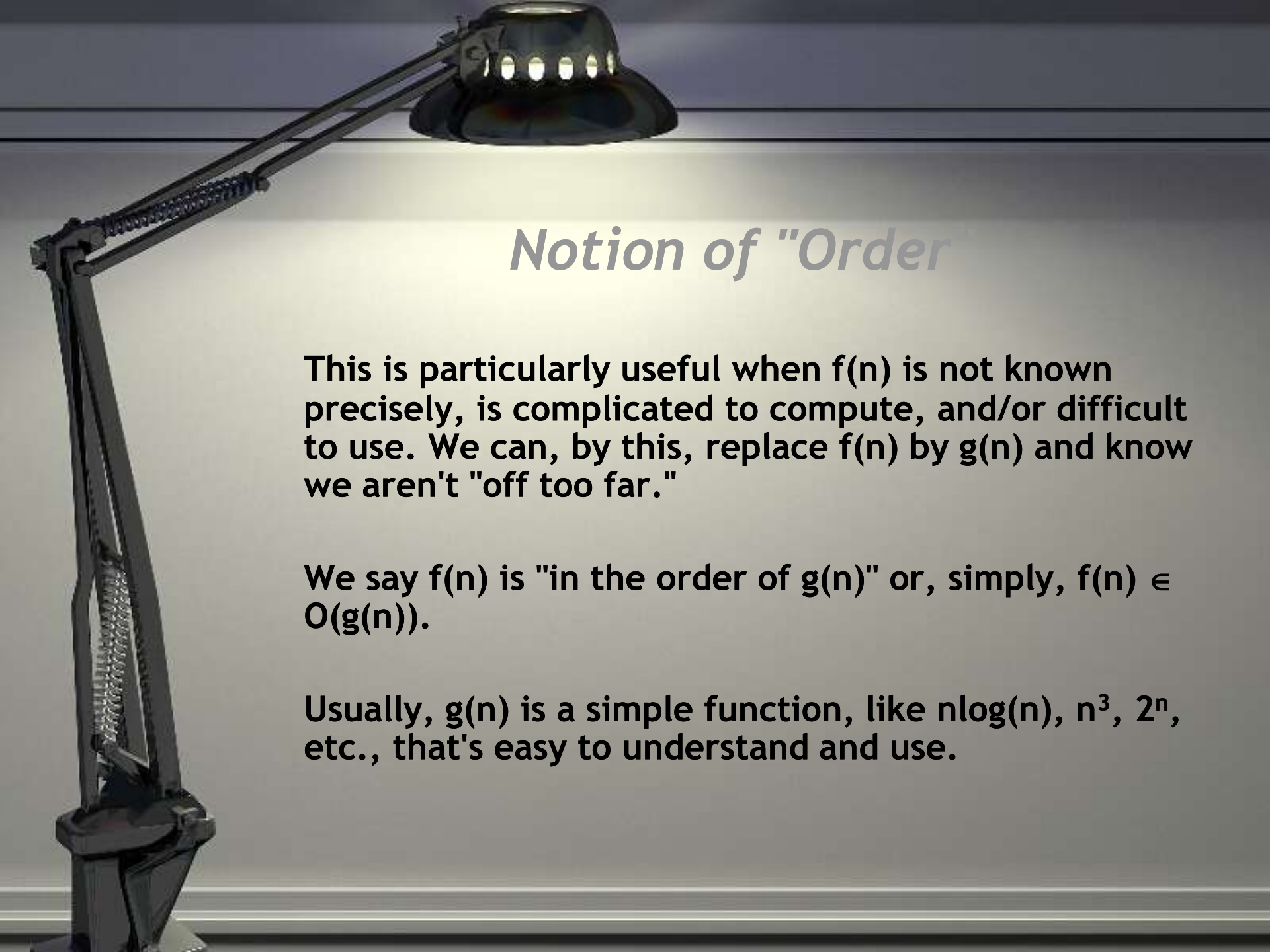


Notion of "Order"

"Order" is something we use to describe an upper bound upon the size of something else (in our case, time, but it can apply to almost anything).

For example, let f(n) and g(n) be two functions. We say "f(n) is order g(n)" when there exists constants c and N such that $f(n) \leq cg(n)$ for all $n \geq N$.

What this is saying is that when n is 'large enough,' f(n) is bounded above by a constant multiple of g(n).

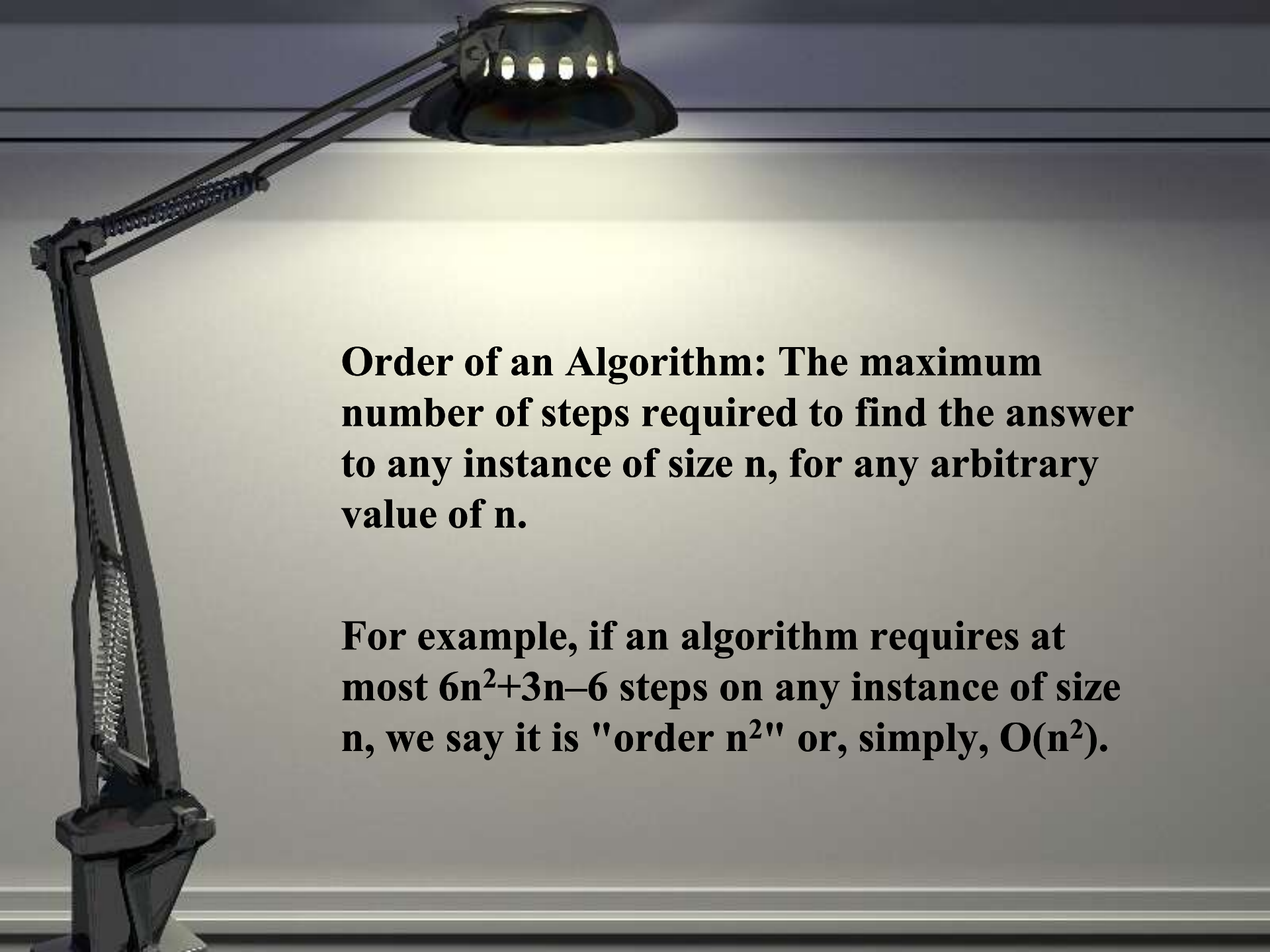


Notion of "Order"

This is particularly useful when f(n) is not known precisely, is complicated to compute, and/or difficult to use. We can, by this, replace f(n) by g(n) and know we aren't "off too far."

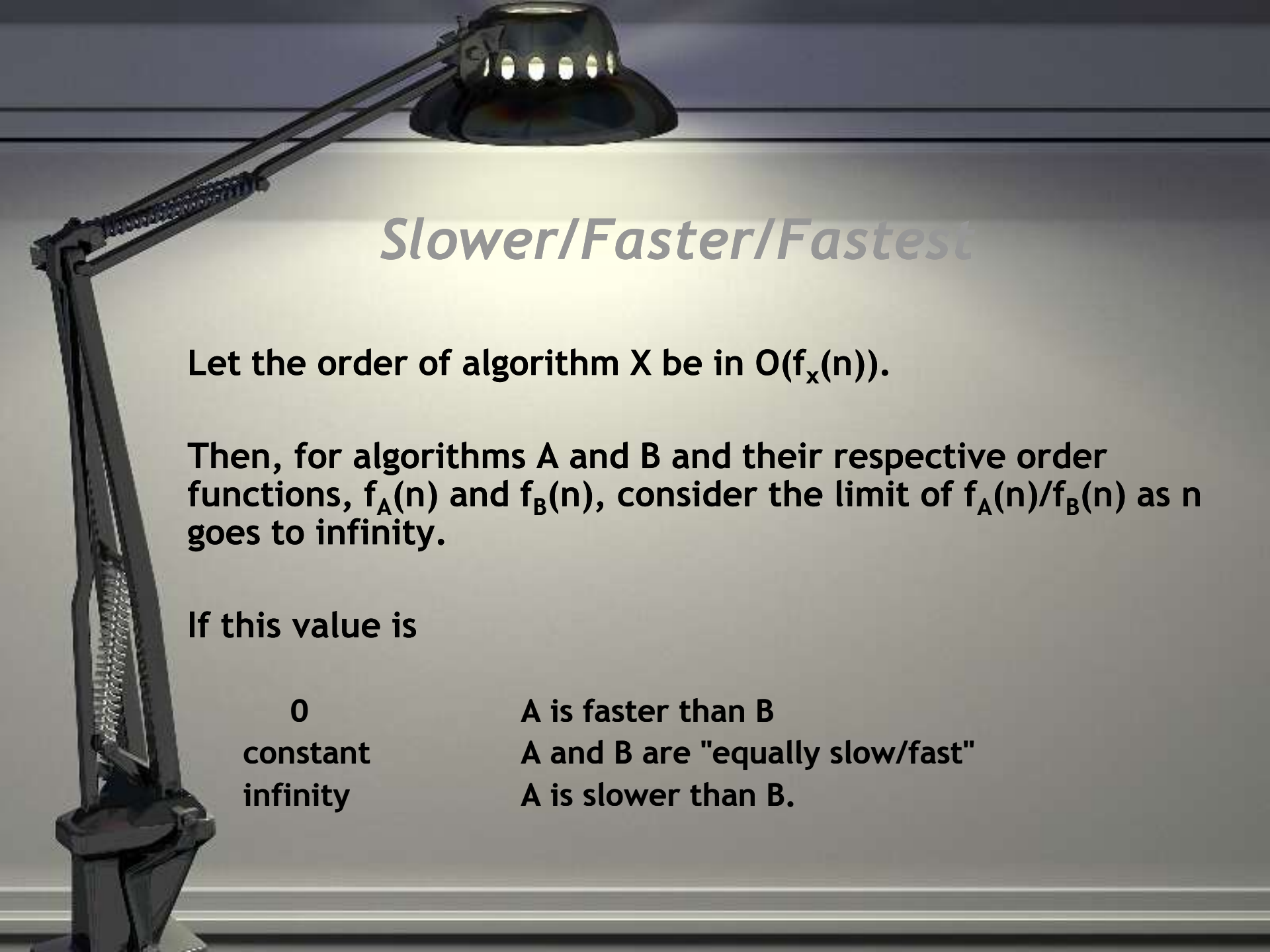
We say f(n) is "in the order of g(n)" or, simply, $f(n) \in O(g(n))$.

Usually, g(n) is a simple function, like nlog(n), n^3, 2^n, etc., that's easy to understand and use.



Order of an Algorithm: The maximum number of steps required to find the answer to any instance of size n, for any arbitrary value of n.

For example, if an algorithm requires at most $6n^2+3n-6$ steps on any instance of size n, we say it is "order n^2" or, simply, $O(n^2)$.



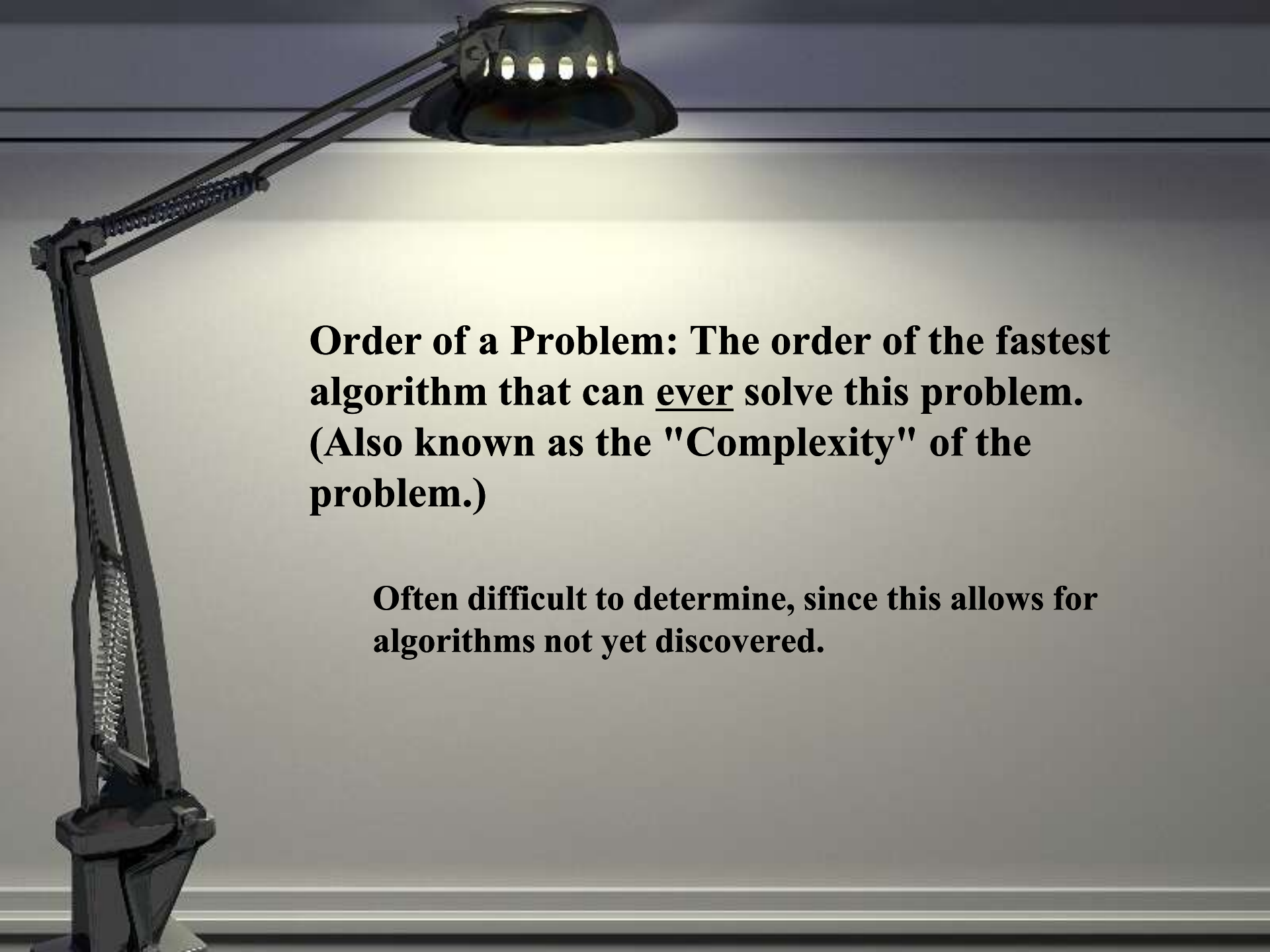
Slower/Faster/Fastest

Let the order of algorithm X be in $O(f_x(n))$.

Then, for algorithms A and B and their respective order functions, $f_A(n)$ and $f_B(n)$, consider the limit of $f_A(n)/f_B(n)$ as n goes to infinity.

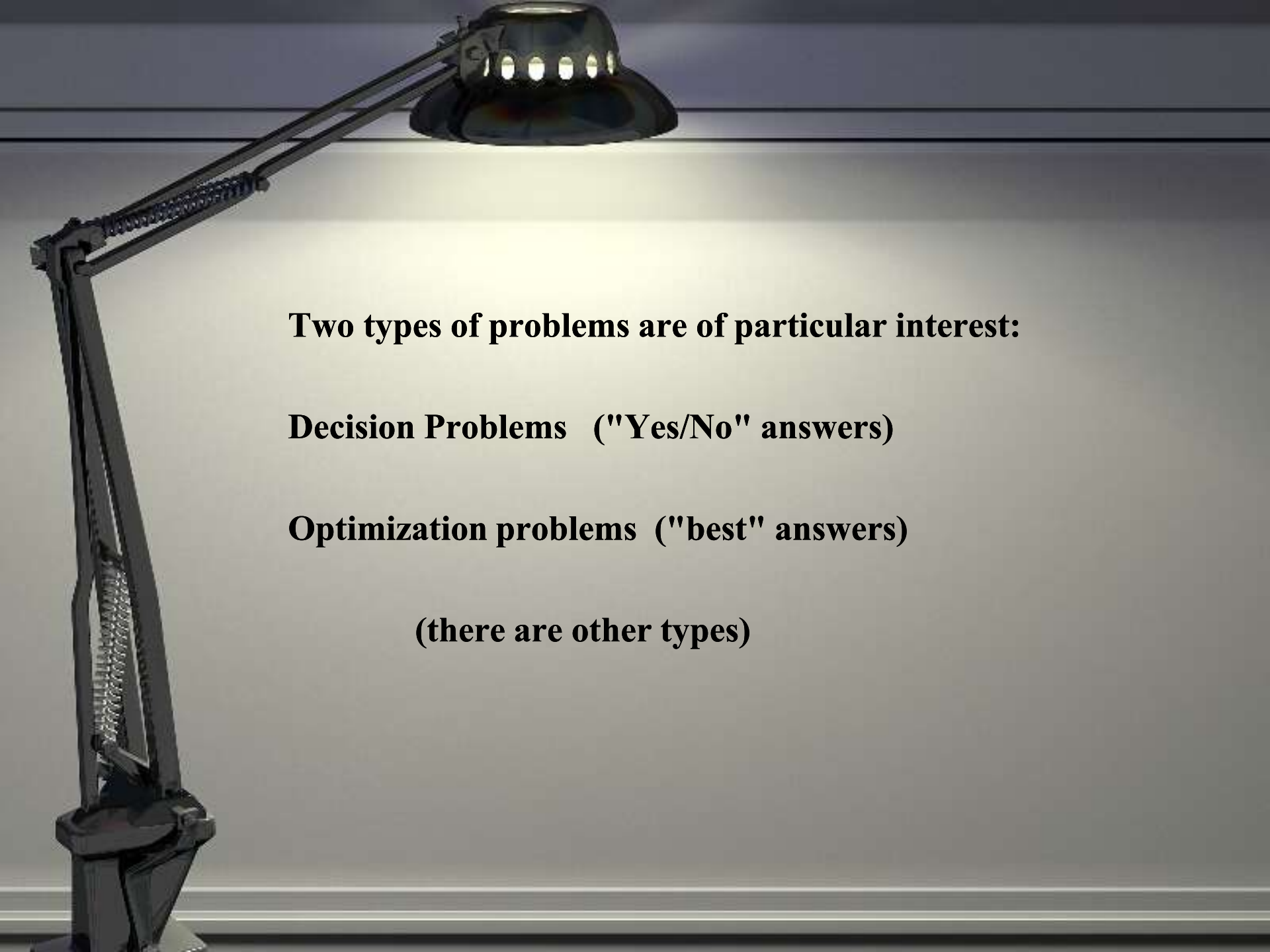
If this value is

0	A is faster than B
constant	A and B are "equally slow/fast"
infinity	A is slower than B.



Order of a Problem: The order of the fastest algorithm that can ever solve this problem. (Also known as the "Complexity" of the problem.)

Often difficult to determine, since this allows for algorithms not yet discovered.

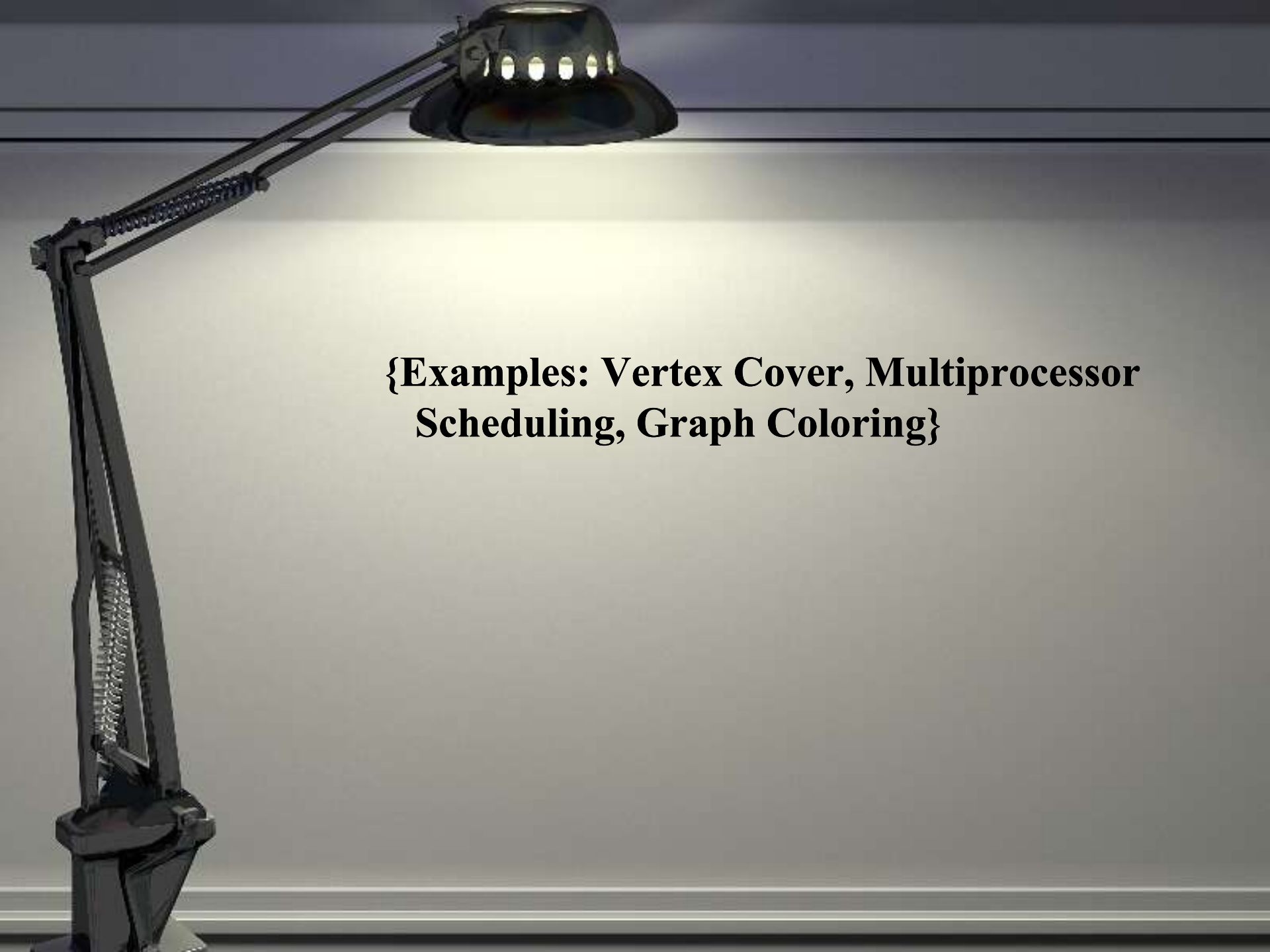


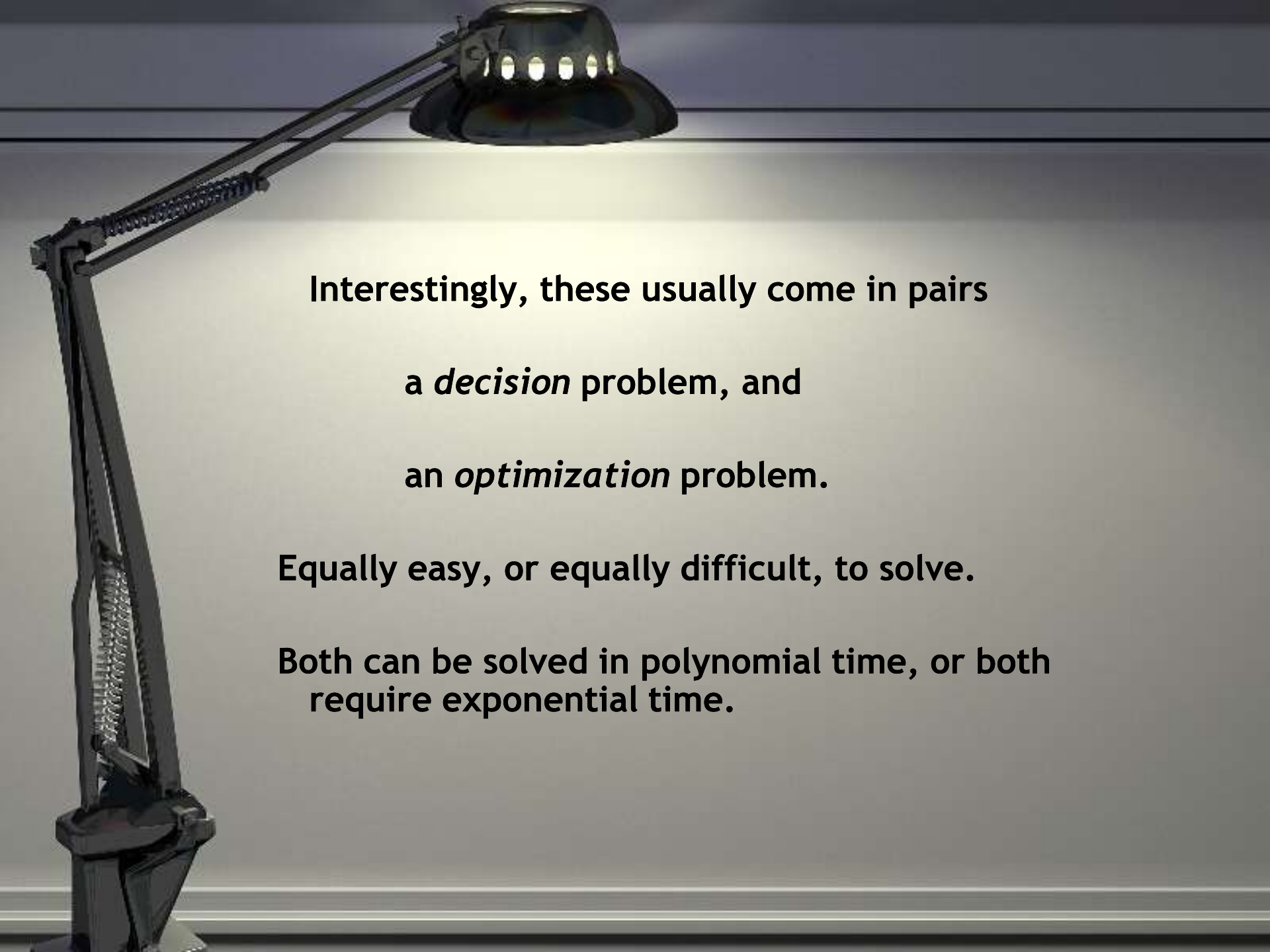
Two types of problems are of particular interest:

Decision Problems ("Yes/No" answers)

Optimization problems ("best" answers)

(there are other types)


{Examples: Vertex Cover, Multiprocessor
Scheduling, Graph Coloring}



Interestingly, these usually come in pairs

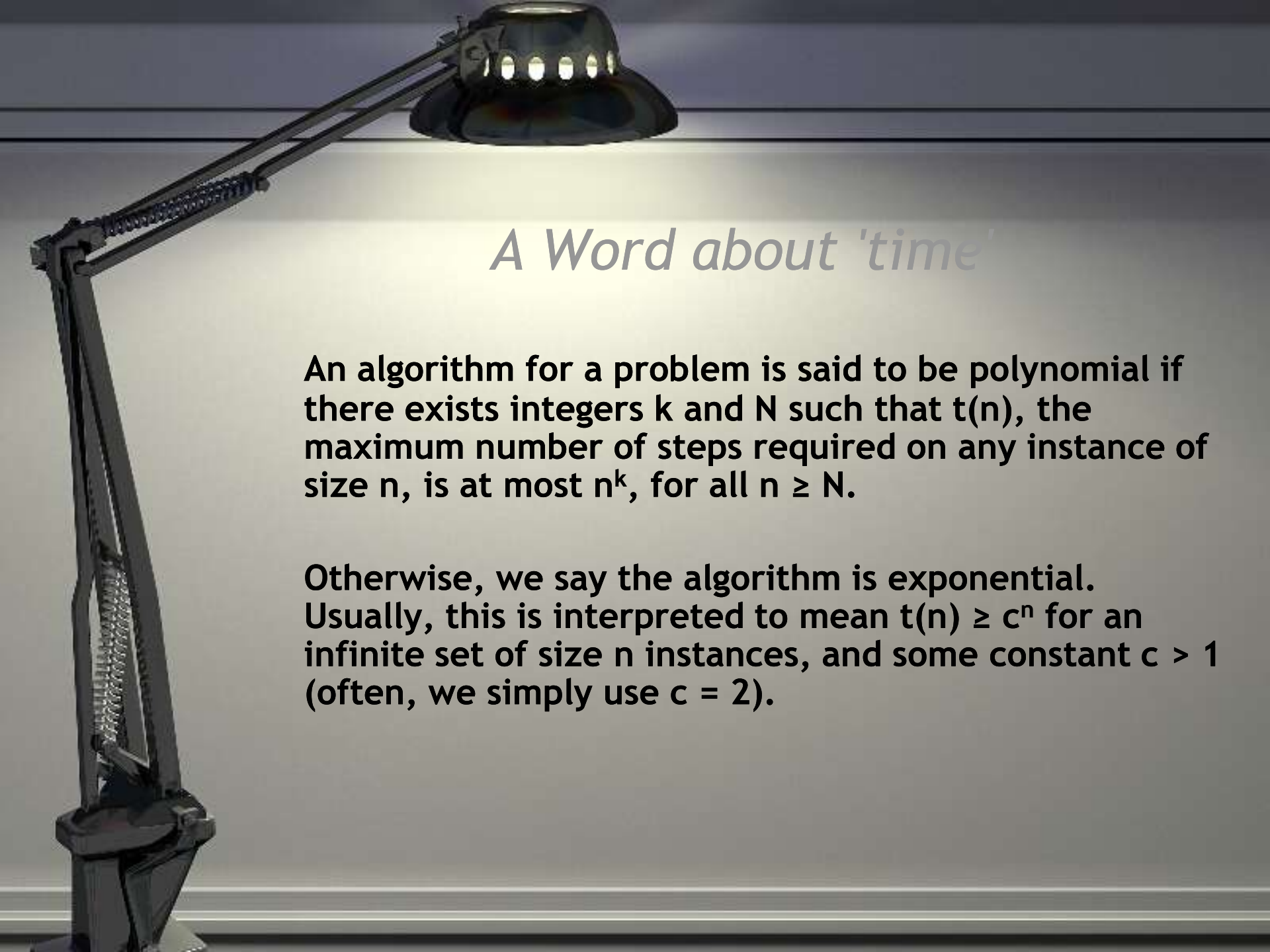
a *decision* problem, and

an *optimization* problem.

Equally easy, or equally difficult, to solve.

Both can be solved in polynomial time, or both require exponential time.

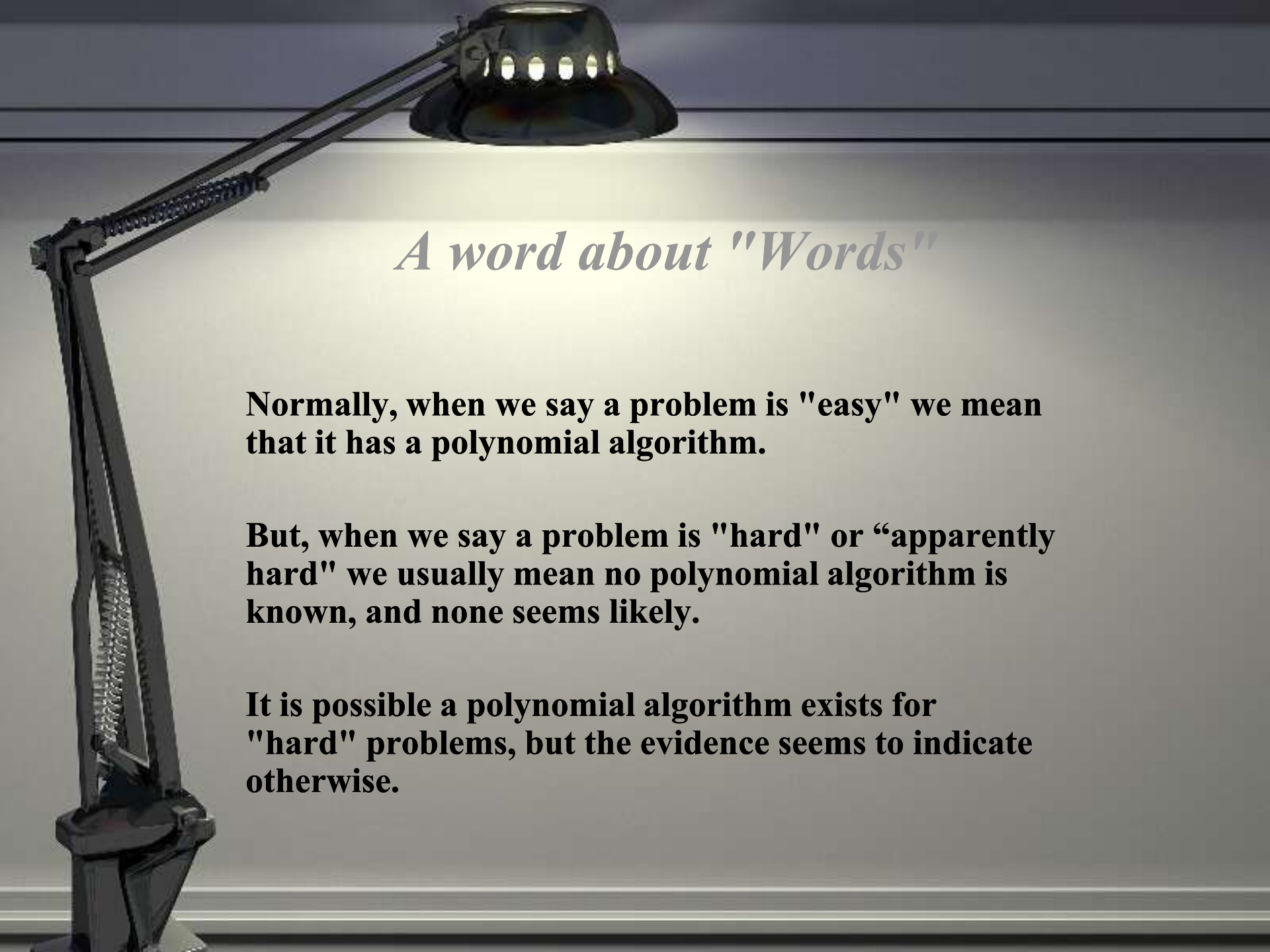

{Example: Vertex Cover,
Multiprocessor Scheduling}



A Word about 'time'

An algorithm for a problem is said to be polynomial if there exists integers k and N such that t(n), the maximum number of steps required on any instance of size n, is at most n^k, for all $n \geq N$.

Otherwise, we say the algorithm is exponential. Usually, this is interpreted to mean $t(n) \geq c^n$ for an infinite set of size n instances, and some constant $c > 1$ (often, we simply use $c = 2$).

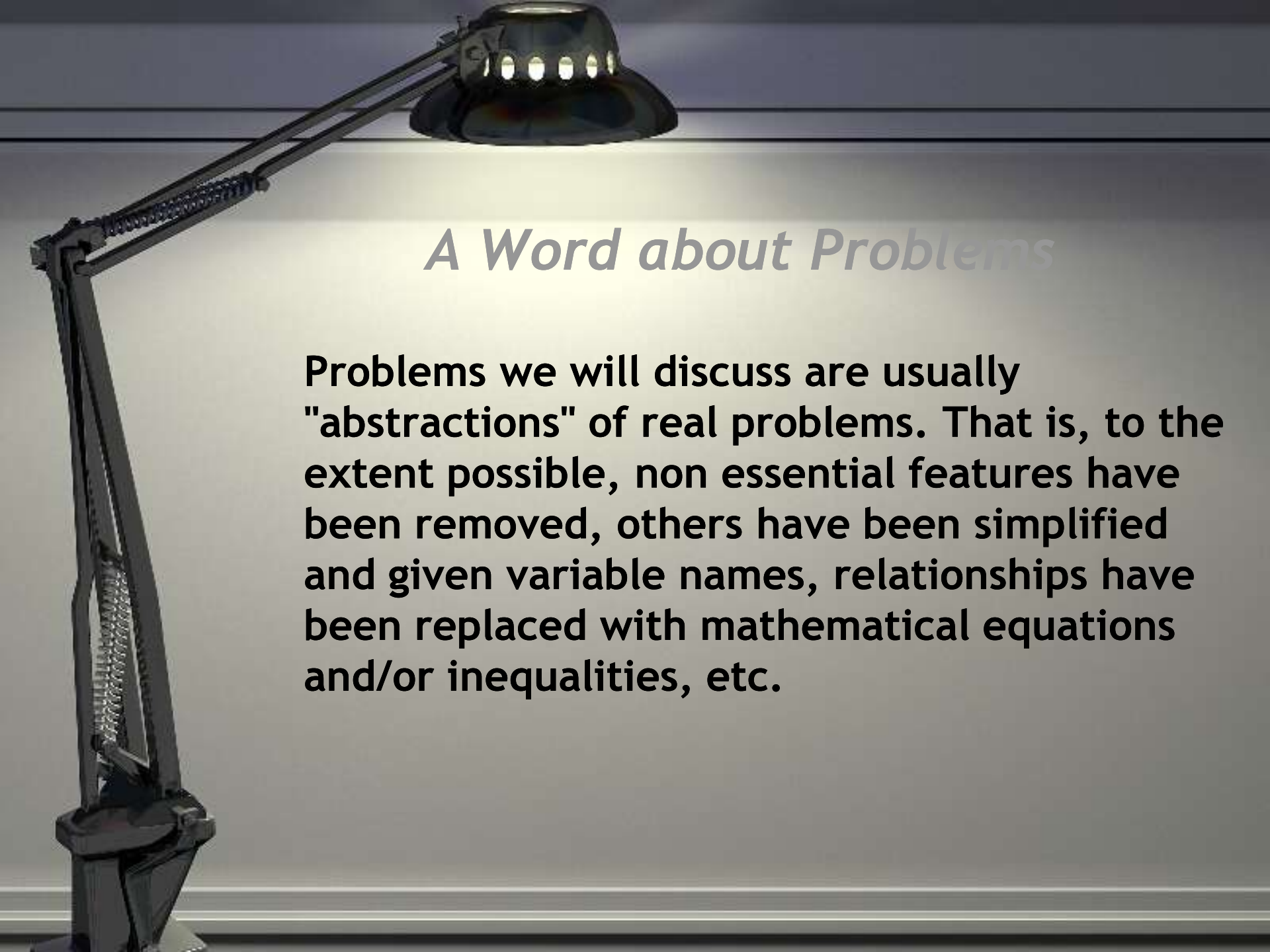


A word about "Words"

Normally, when we say a problem is "easy" we mean that it has a polynomial algorithm.

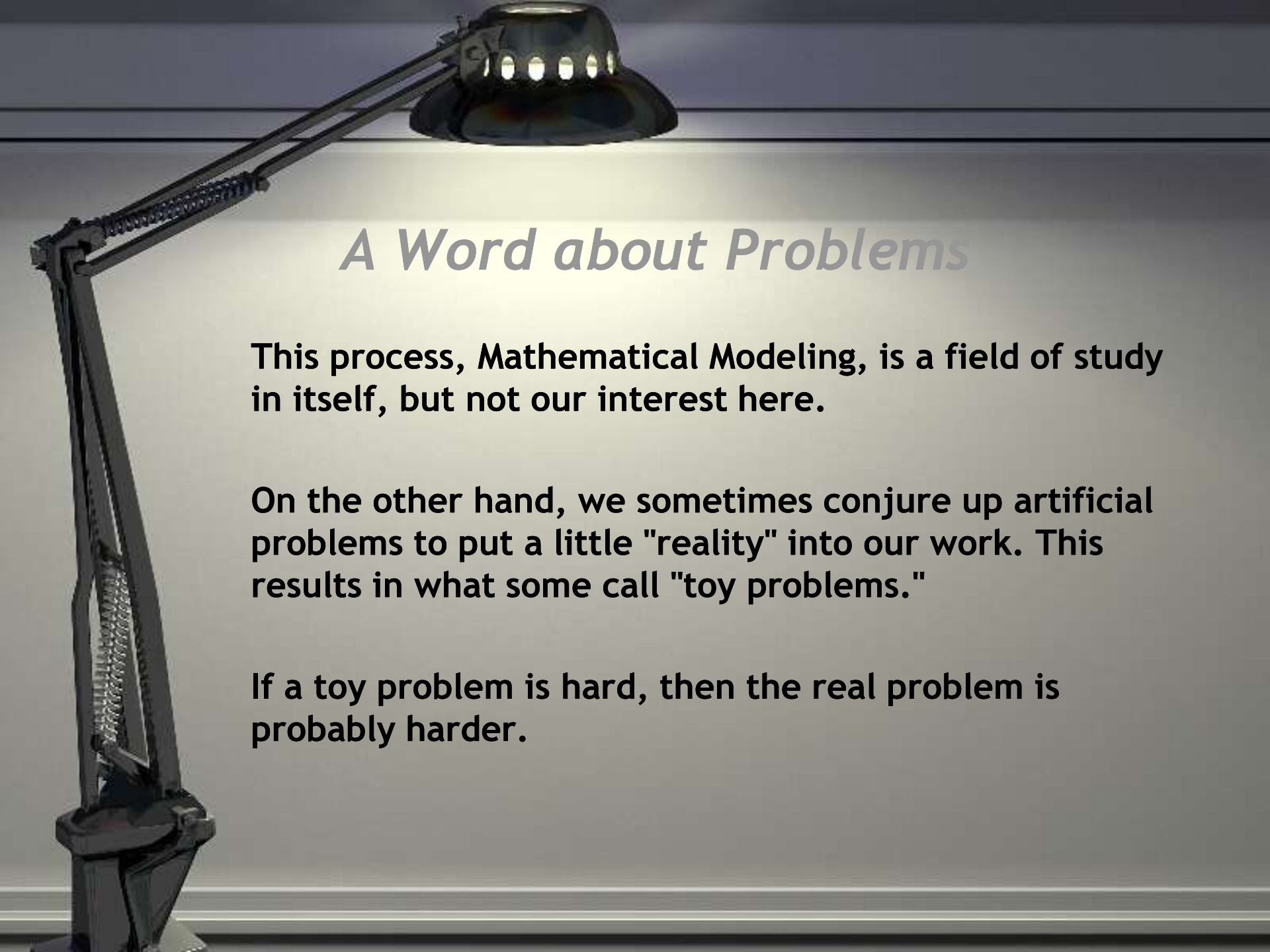
But, when we say a problem is "hard" or “apparently hard" we usually mean no polynomial algorithm is known, and none seems likely.

It is possible a polynomial algorithm exists for "hard" problems, but the evidence seems to indicate otherwise.



A Word about Problems

Problems we will discuss are usually "abstractions" of real problems. That is, to the extent possible, non essential features have been removed, others have been simplified and given variable names, relationships have been replaced with mathematical equations and/or inequalities, etc.

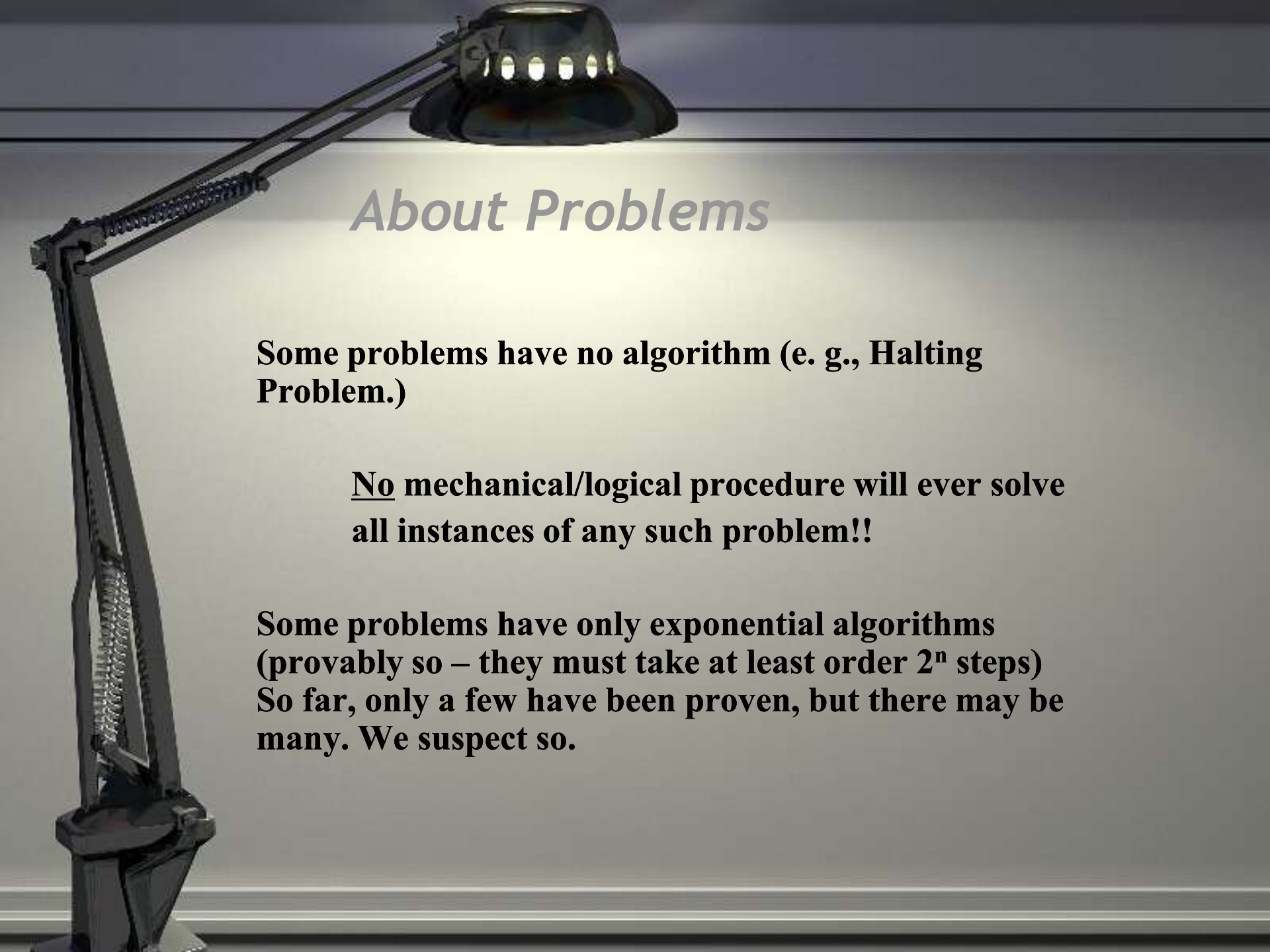


A Word about Problems

This process, Mathematical Modeling, is a field of study in itself, but not our interest here.

On the other hand, we sometimes conjure up artificial problems to put a little "reality" into our work. This results in what some call "toy problems."

If a toy problem is hard, then the real problem is probably harder.

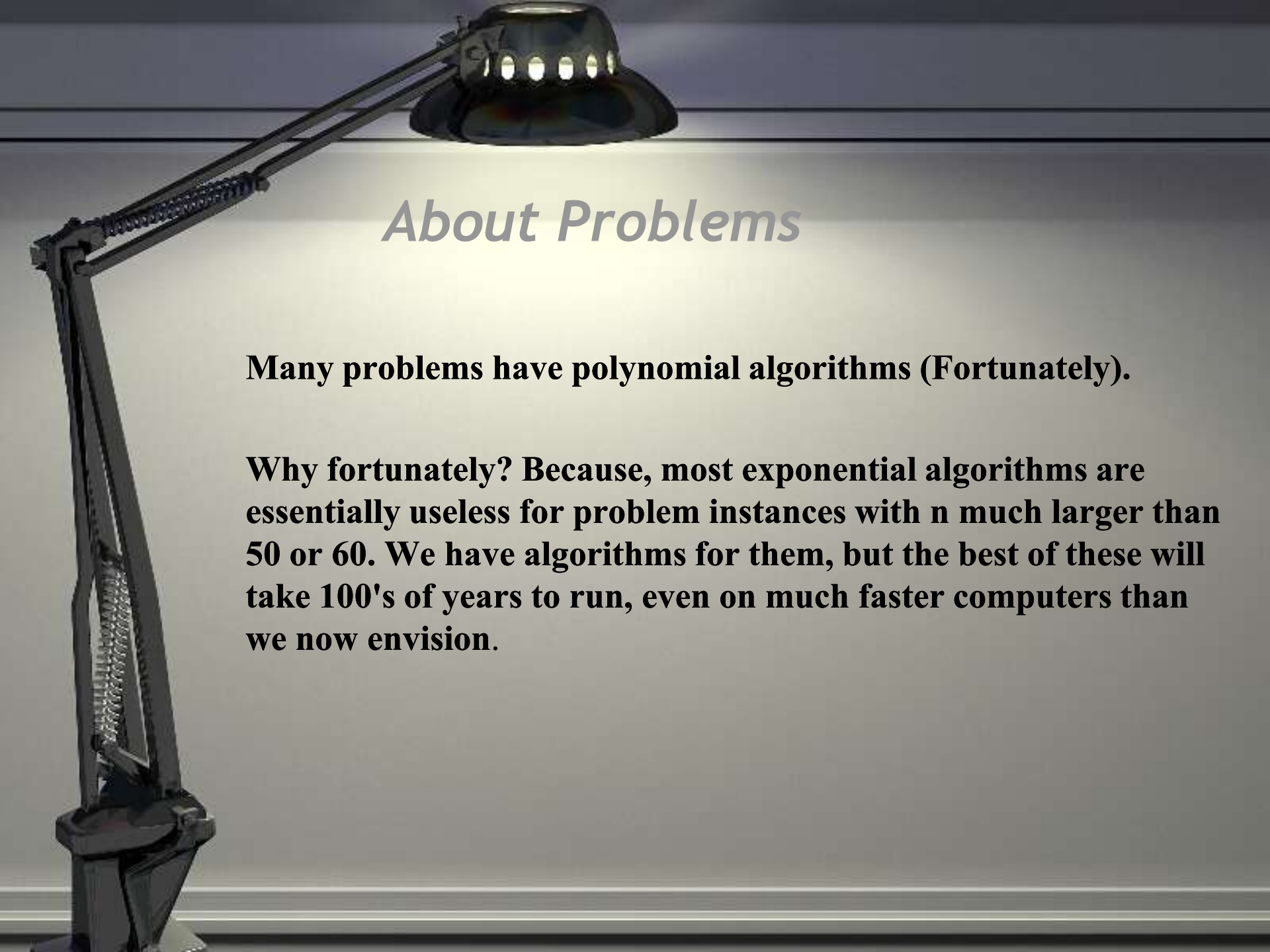


About Problems

Some problems have no algorithm (e. g., Halting Problem.)

No mechanical/logical procedure will ever solve all instances of any such problem!!

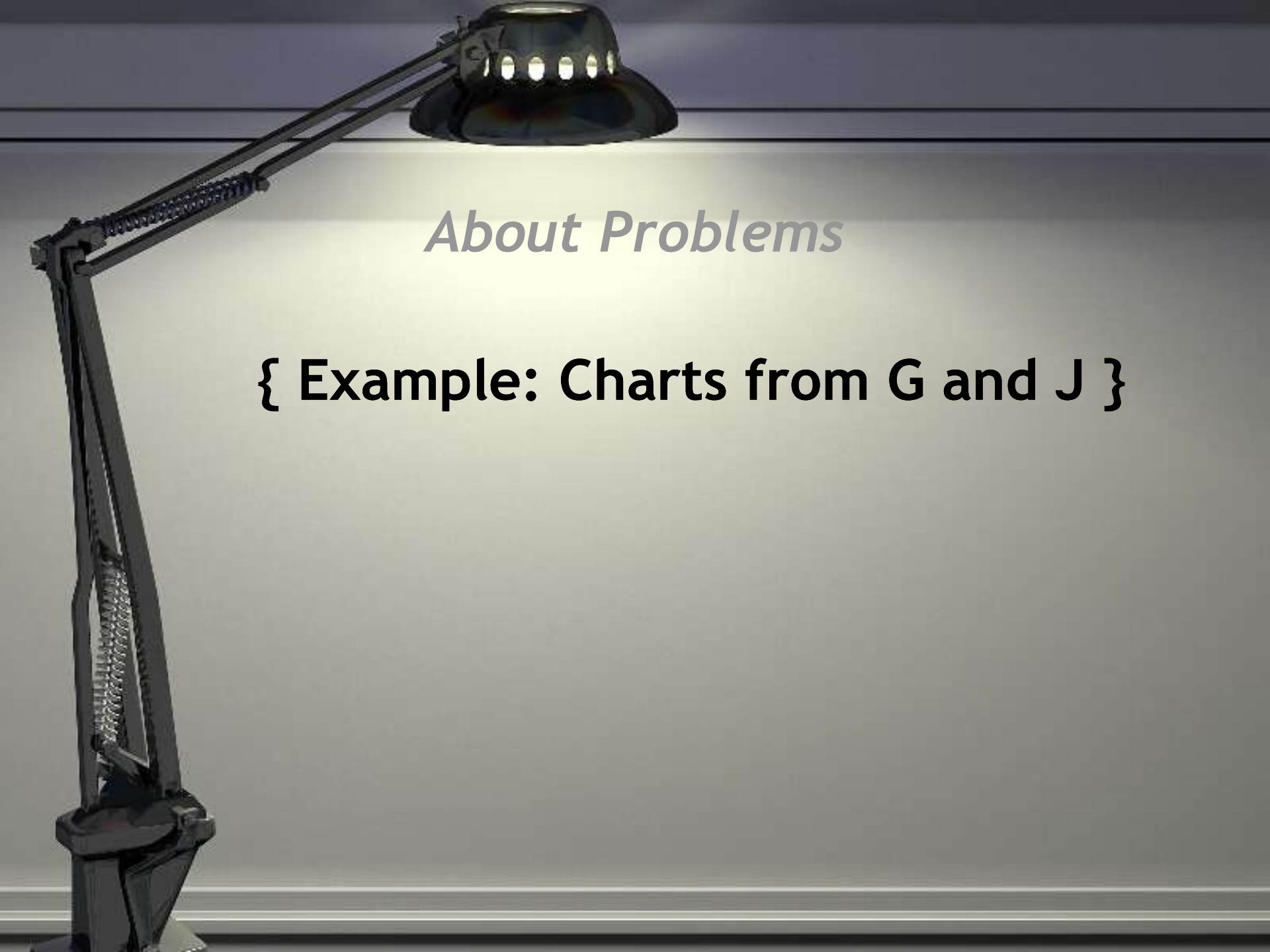
Some problems have only exponential algorithms (provably so – they must take at least order 2^n steps) So far, only a few have been proven, but there may be many. We suspect so.

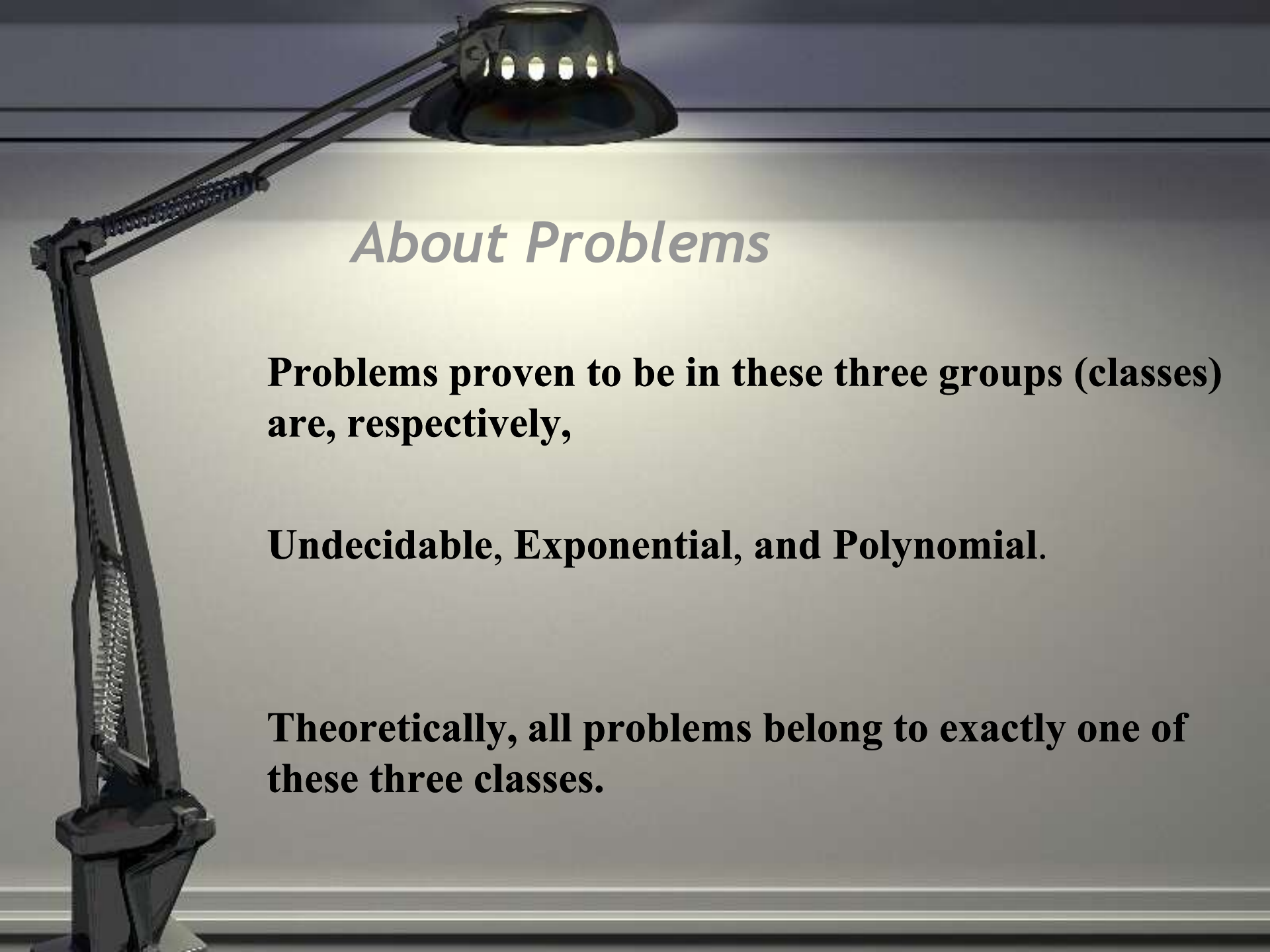


About Problems

Many problems have polynomial algorithms (Fortunately).

Why fortunately? Because, most exponential algorithms are essentially useless for problem instances with n much larger than 50 or 60. We have algorithms for them, but the best of these will take 100's of years to run, even on much faster computers than we now envision.


About Problems
{ Example: Charts from G and J }

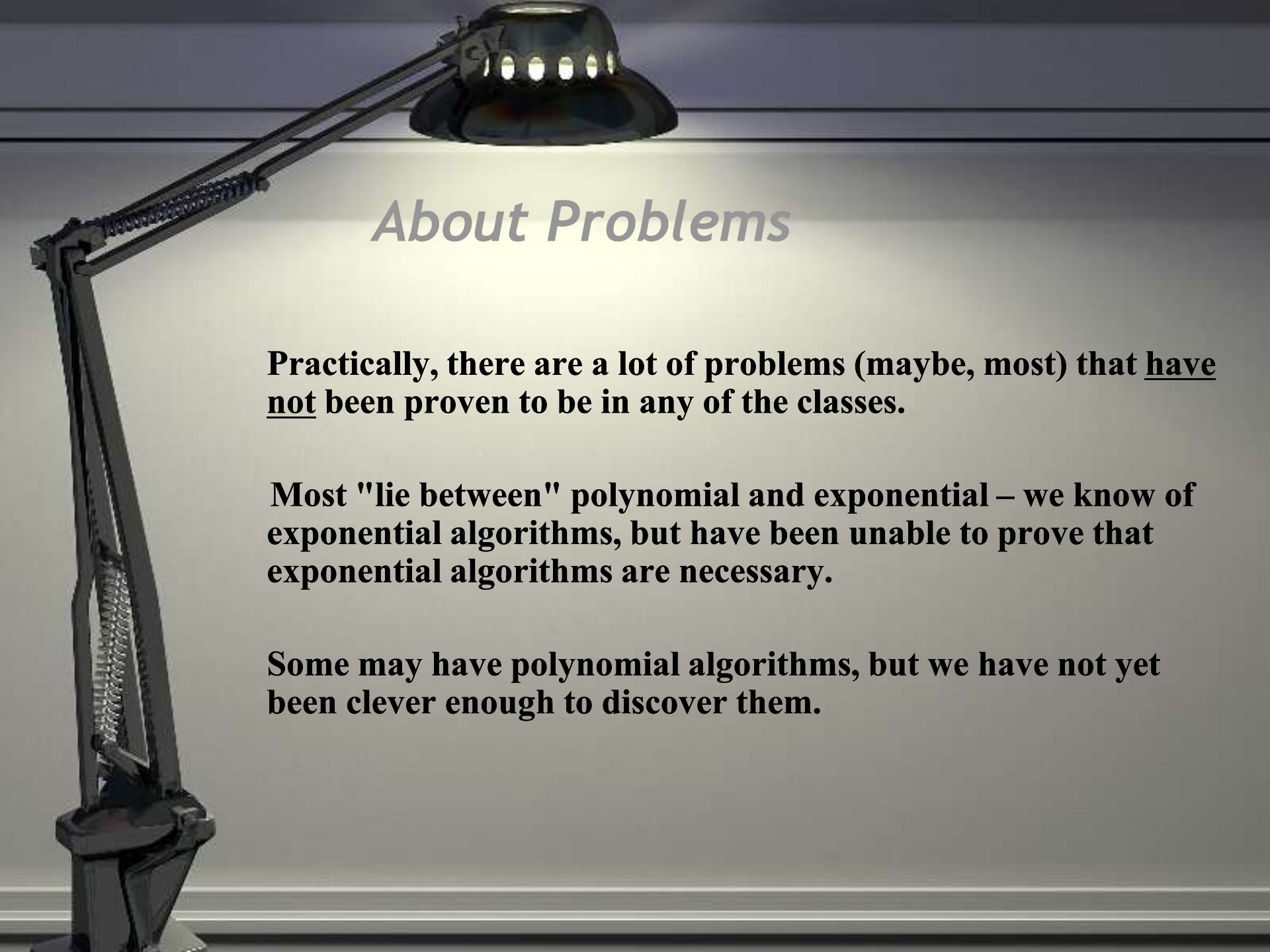


About Problems

Problems proven to be in these three groups (classes) are, respectively,

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes.

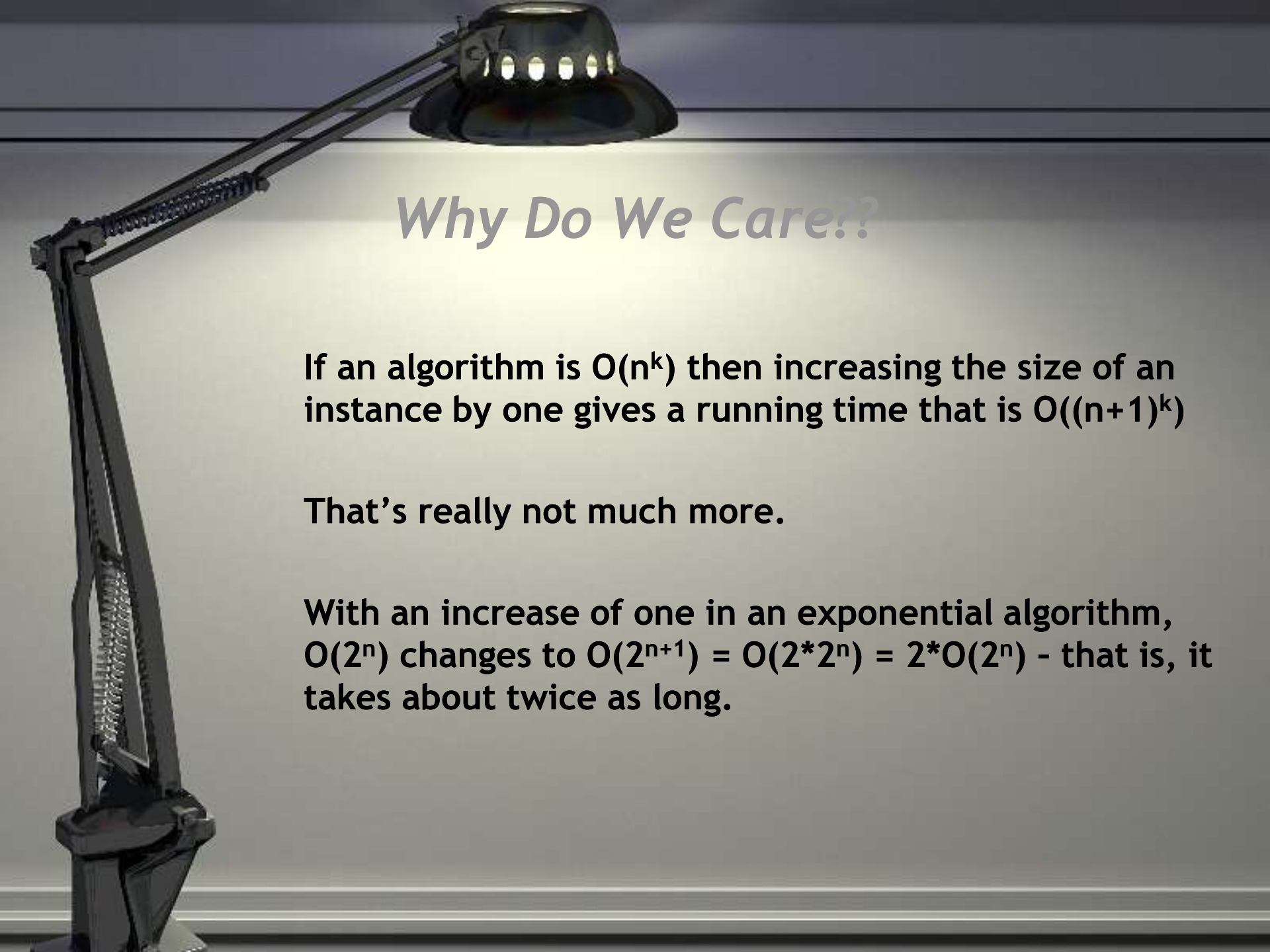


About Problems

Practically, there are a lot of problems (maybe, most) that have not been proven to be in any of the classes.

Most "lie between" polynomial and exponential – we know of exponential algorithms, but have been unable to prove that exponential algorithms are necessary.

Some may have polynomial algorithms, but we have not yet been clever enough to discover them.

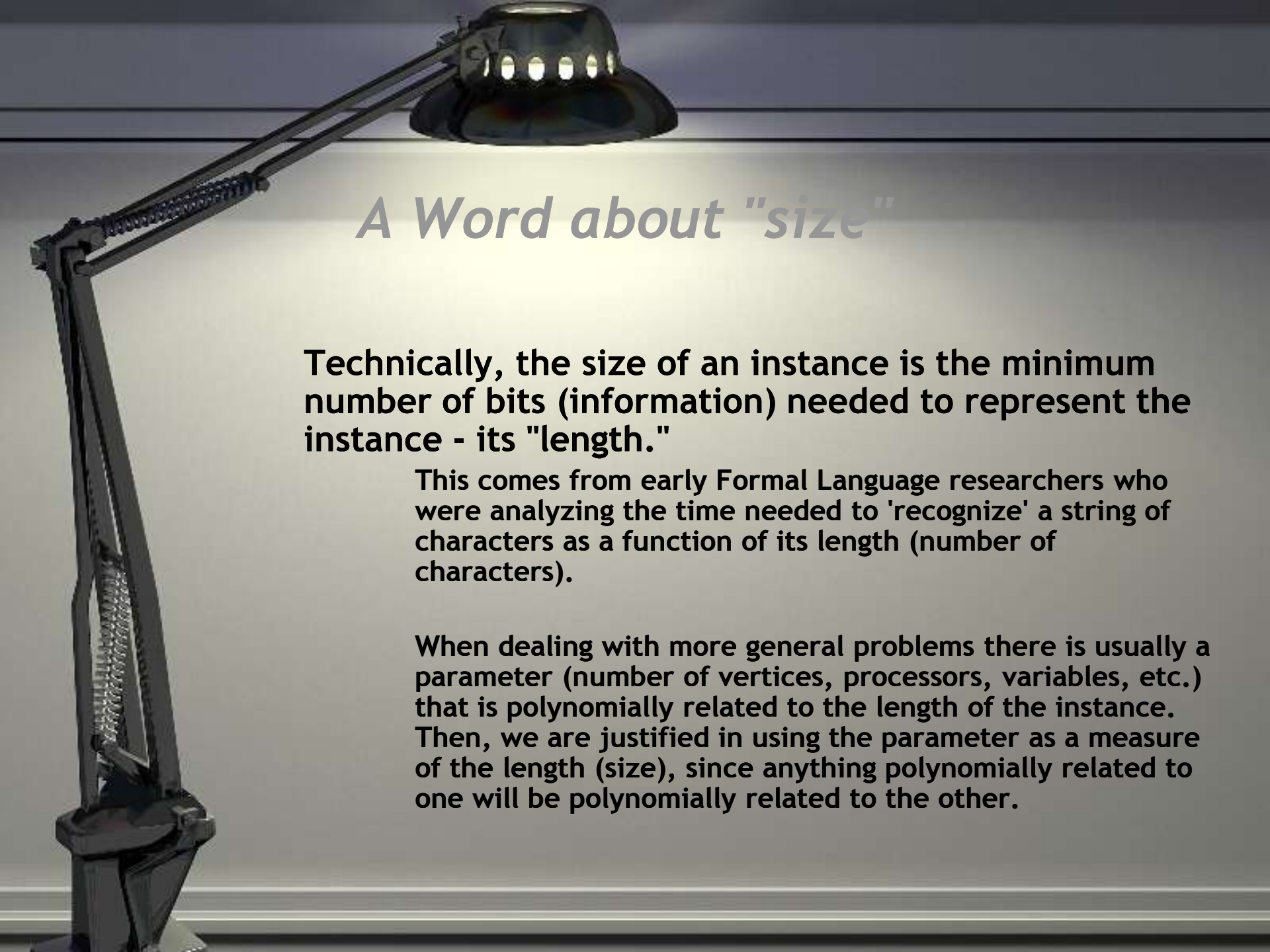


Why Do We Care??

If an algorithm is $O(n^k)$ then increasing the size of an instance by one gives a running time that is $O((n+1)^k)$

That's really not much more.

With an increase of one in an exponential algorithm, $O(2^n)$ changes to $O(2^{n+1}) = O(2*2^n) = 2*O(2^n)$ - that is, it takes about twice as long.

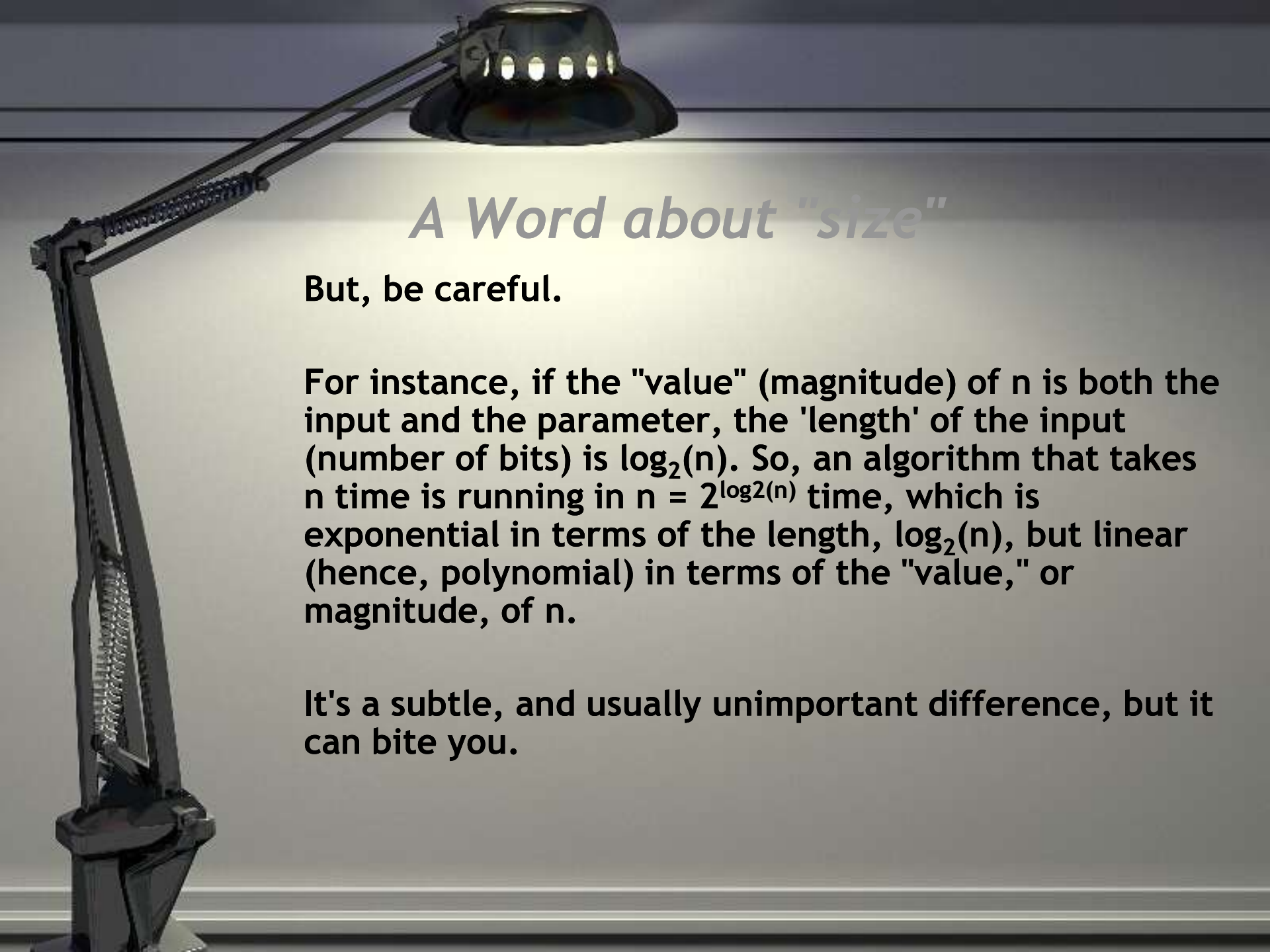


A Word about "size"

Technically, the size of an instance is the minimum number of bits (information) needed to represent the instance - its "length."

This comes from early Formal Language researchers who were analyzing the time needed to 'recognize' a string of characters as a function of its length (number of characters).

When dealing with more general problems there is usually a parameter (number of vertices, processors, variables, etc.) that is polynomially related to the length of the instance. Then, we are justified in using the parameter as a measure of the length (size), since anything polynomially related to one will be polynomially related to the other.

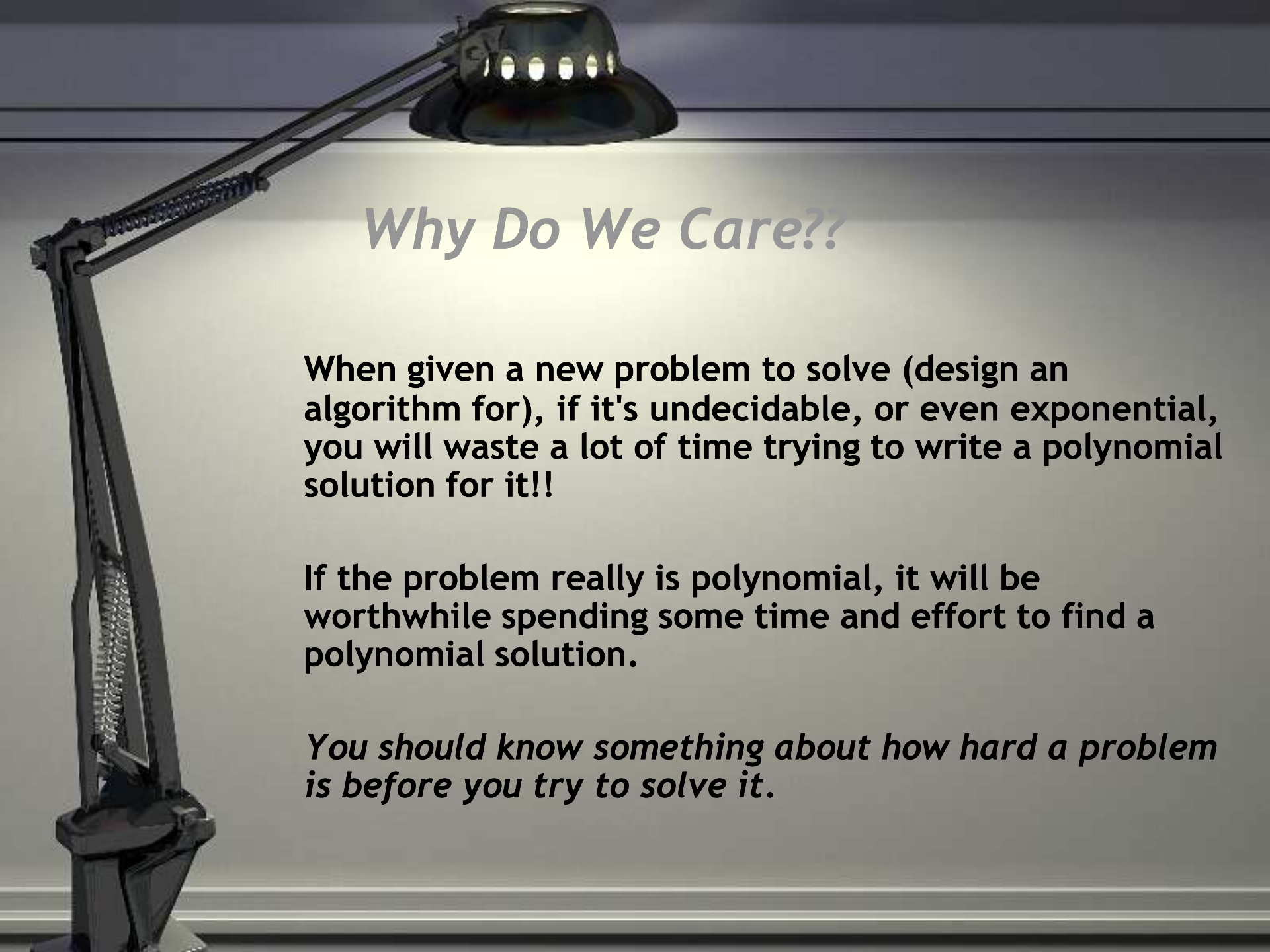


A Word about "size"

But, be careful.

For instance, if the "value" (magnitude) of n is both the input and the parameter, the 'length' of the input (number of bits) is $\log_2(n)$. So, an algorithm that takes n time is running in $n = 2^{\log2(n)}$ time, which is exponential in terms of the length, $\log_2(n)$, but linear (hence, polynomial) in terms of the "value," or magnitude, of n.

It's a subtle, and usually unimportant difference, but it can bite you.



Why Do We Care??

When given a new problem to solve (design an algorithm for), if it's undecidable, or even exponential, you will waste a lot of time trying to write a polynomial solution for it!!

If the problem really is polynomial, it will be worthwhile spending some time and effort to find a polynomial solution.

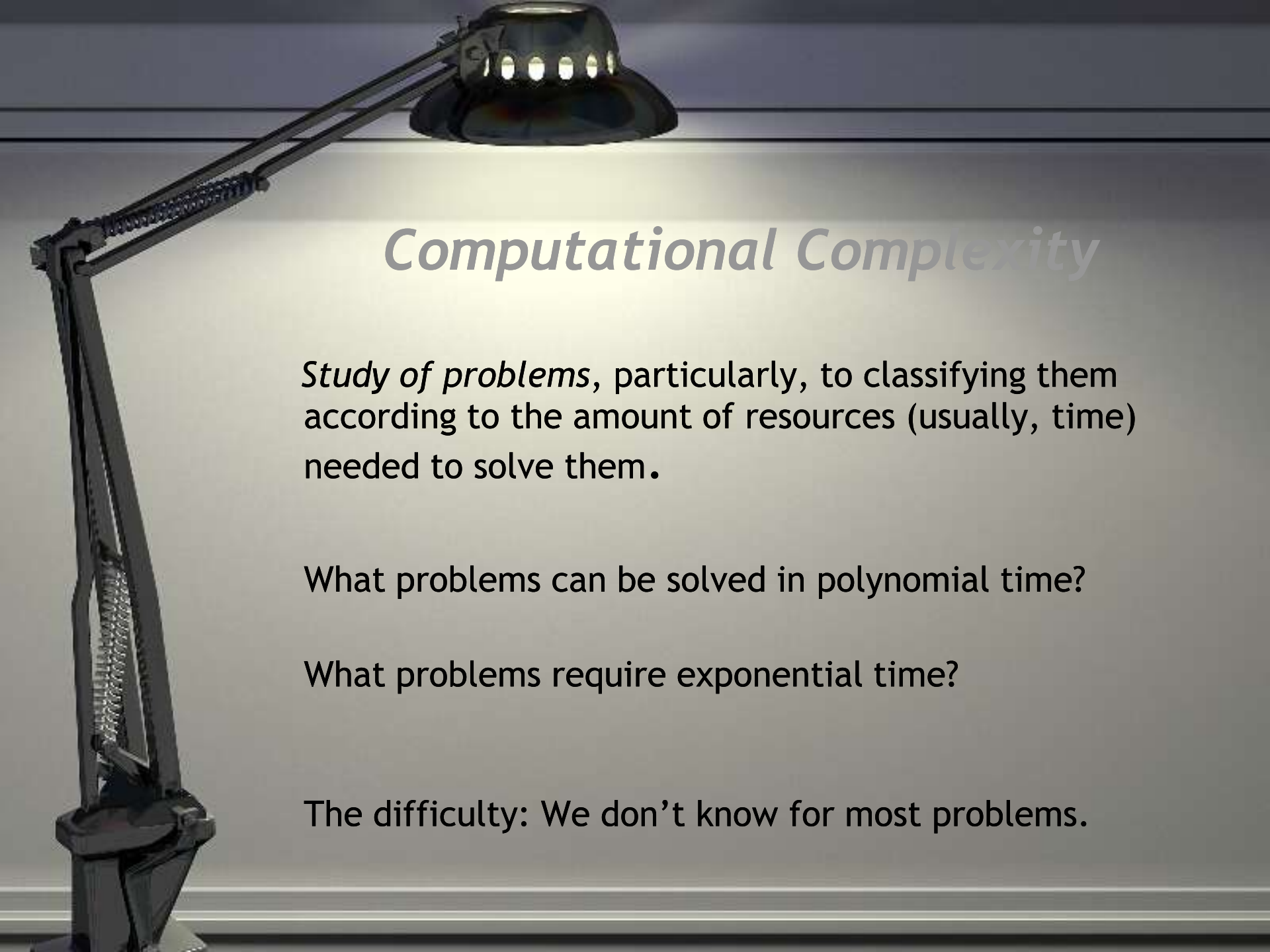
You should know something about how hard a problem is before you try to solve it.

Research Territory

Decidable – vs – Undecidable
(area of Computability Theory)

Exponential – vs – polynomial
(area of Computational Complexity)

Algorithms for any of these
(area of Algorithm Design/Analysis)



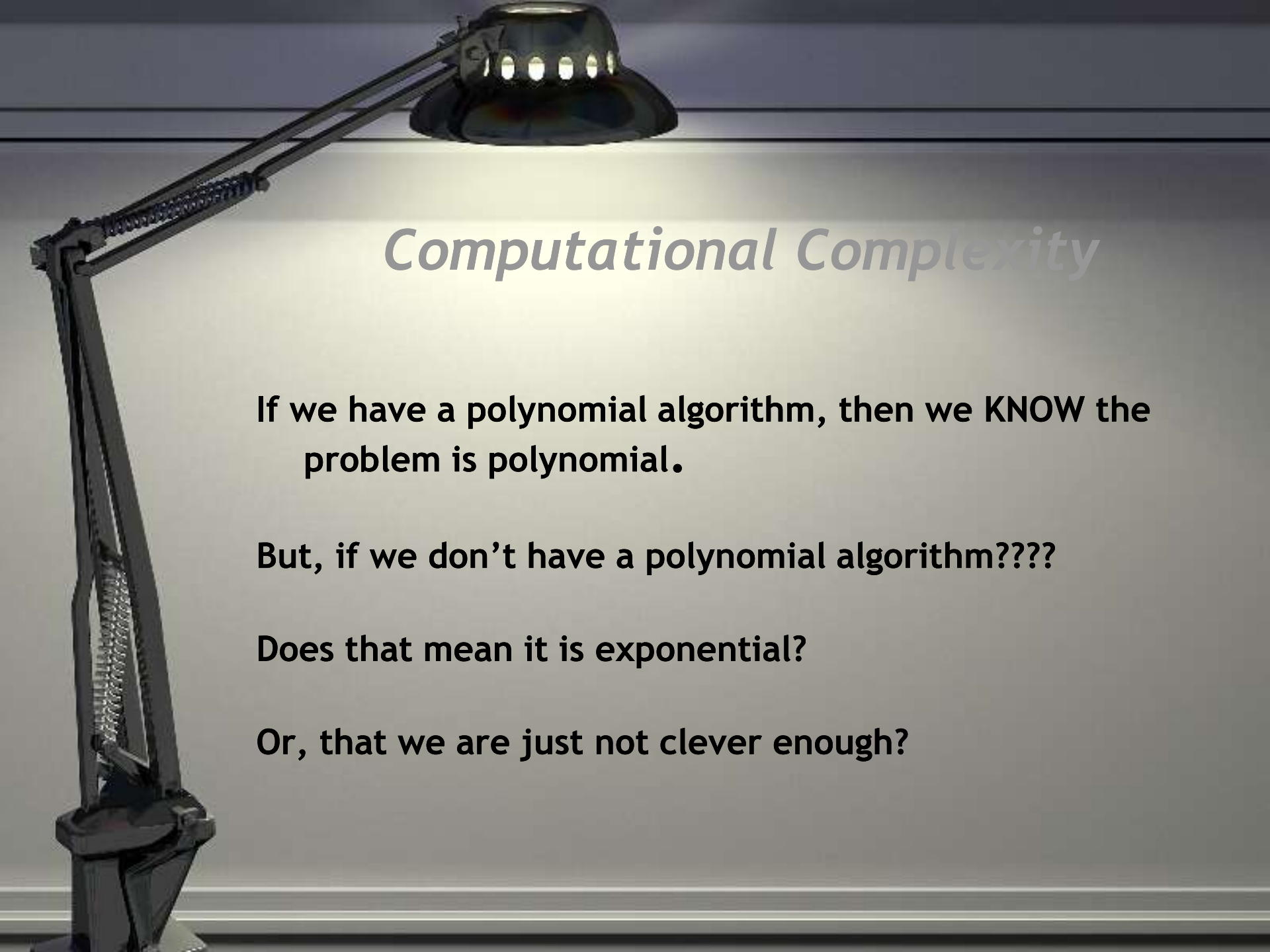
Computational Complexity

Study of problems, particularly, to classifying them according to the amount of resources (usually, time) needed to solve them.

What problems can be solved in polynomial time?

What problems require exponential time?

The difficulty: We don't know for most problems.



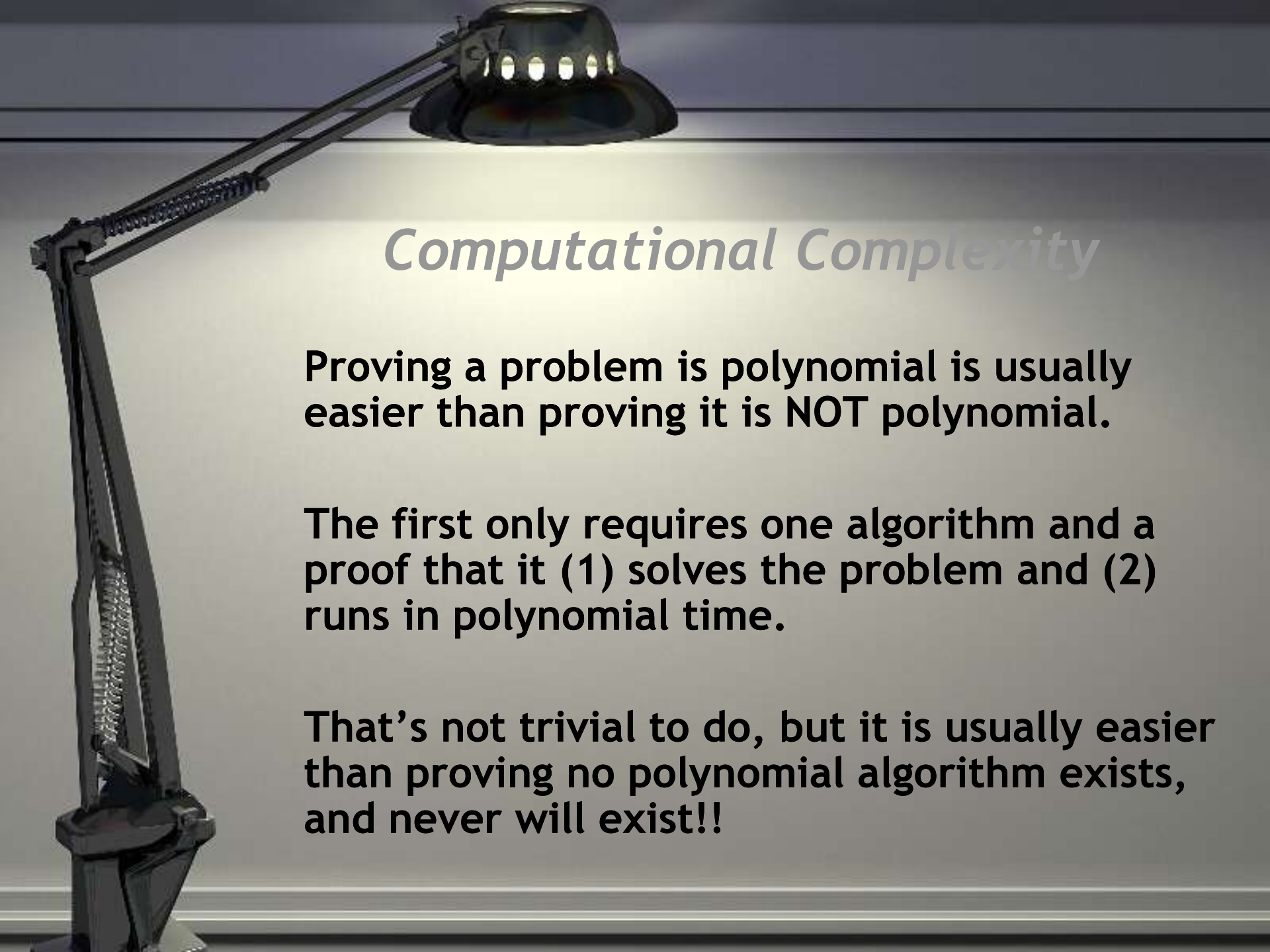
Computational Complexity

If we have a polynomial algorithm, then we KNOW the problem is polynomial.

But, if we don't have a polynomial algorithm????

Does that mean it is exponential?

Or, that we are just not clever enough?

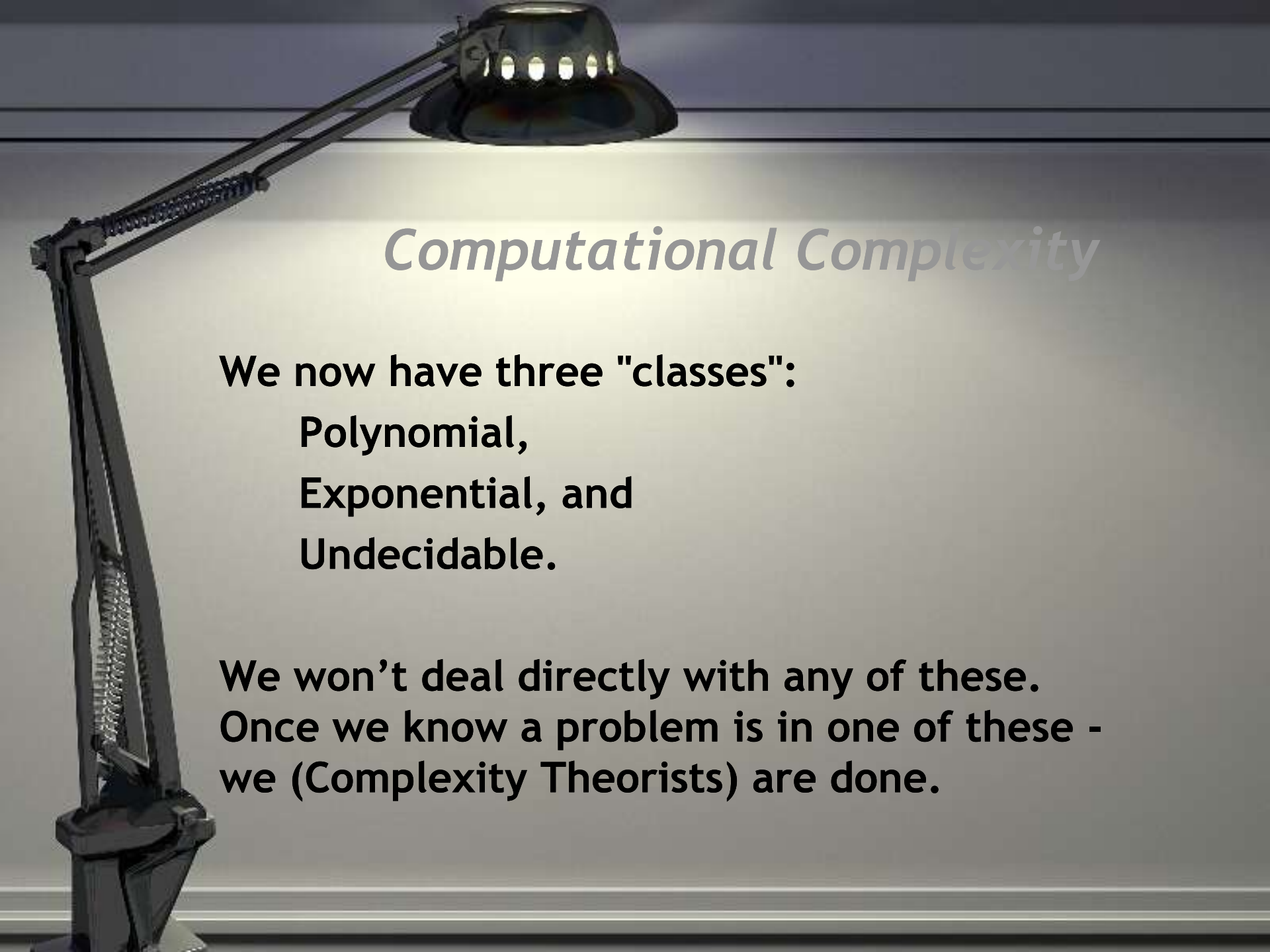


Computational Complexity

Proving a problem is polynomial is usually easier than proving it is NOT polynomial.

The first only requires one algorithm and a proof that it (1) solves the problem and (2) runs in polynomial time.

That's not trivial to do, but it is usually easier than proving no polynomial algorithm exists, and never will exist!!



Computational Complexity

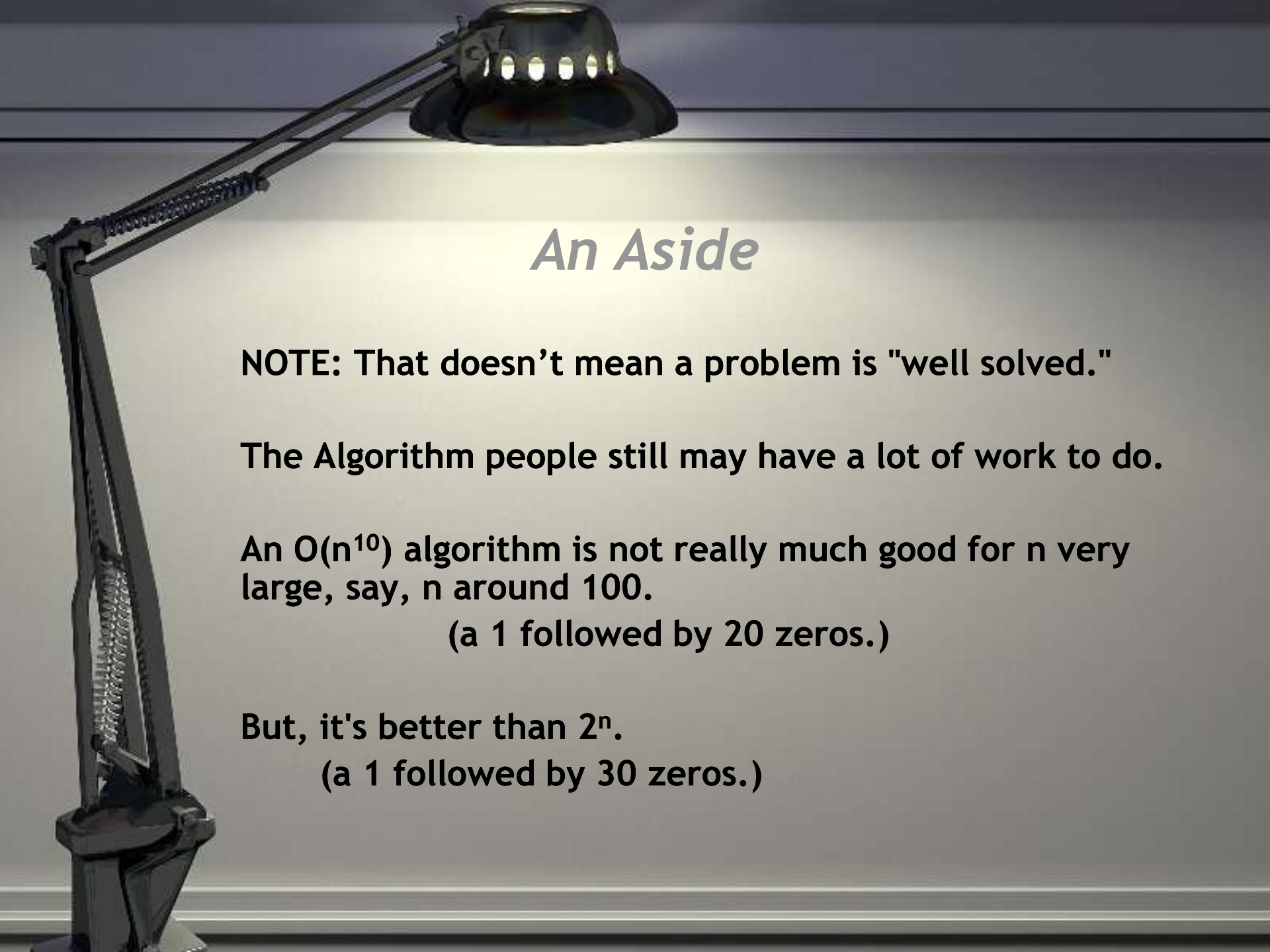
We now have three "classes":

Polynomial,

Exponential, and

Undecidable.

We won't deal directly with any of these. Once we know a problem is in one of these - we (Complexity Theorists) are done.



An Aside

NOTE: That doesn't mean a problem is "well solved."

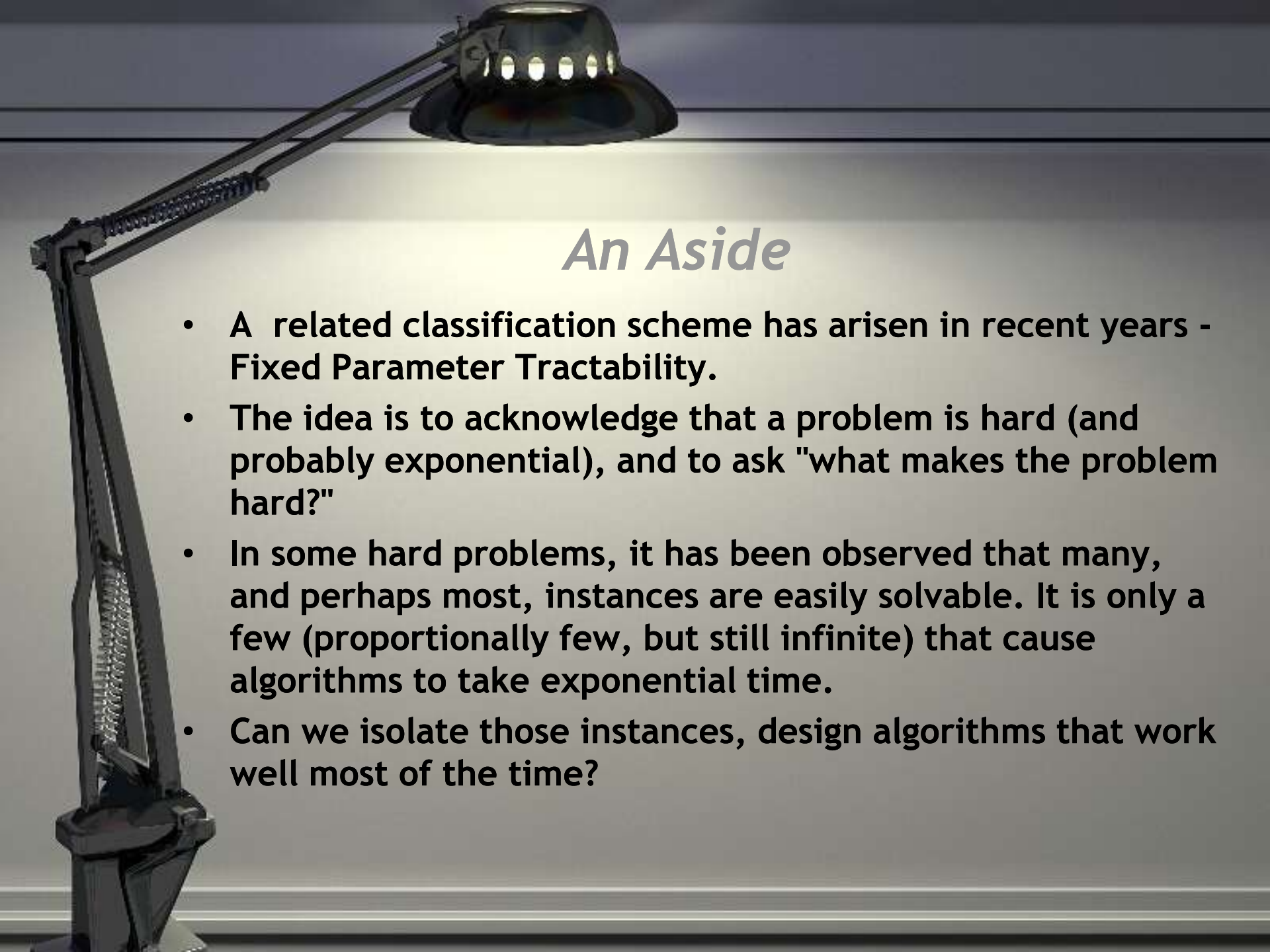
The Algorithm people still may have a lot of work to do.

An $O(n^{10})$ algorithm is not really much good for n very large, say, n around 100.

(a 1 followed by 20 zeros.)

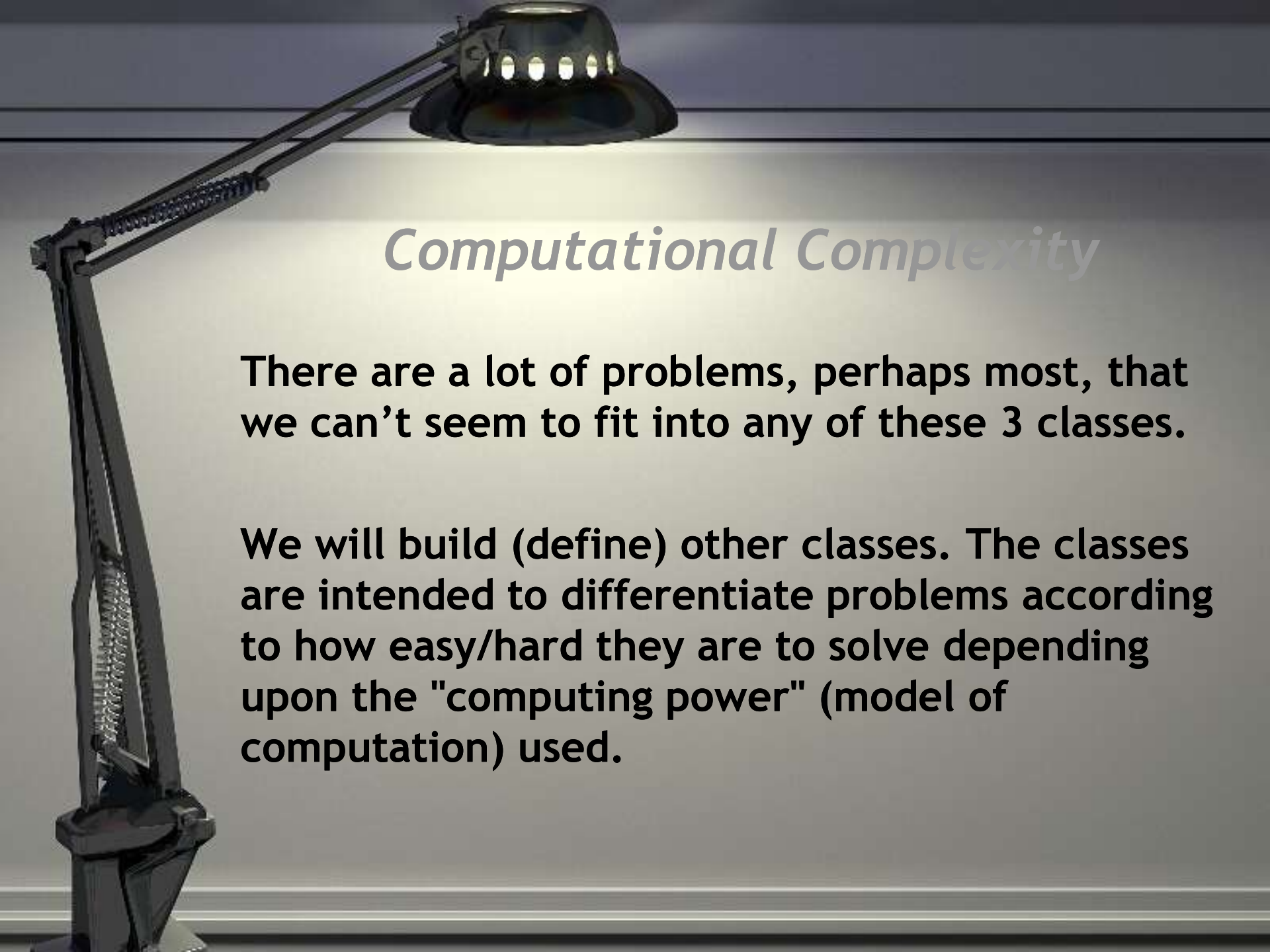
But, it's better than 2^n.

(a 1 followed by 30 zeros.)



An Aside

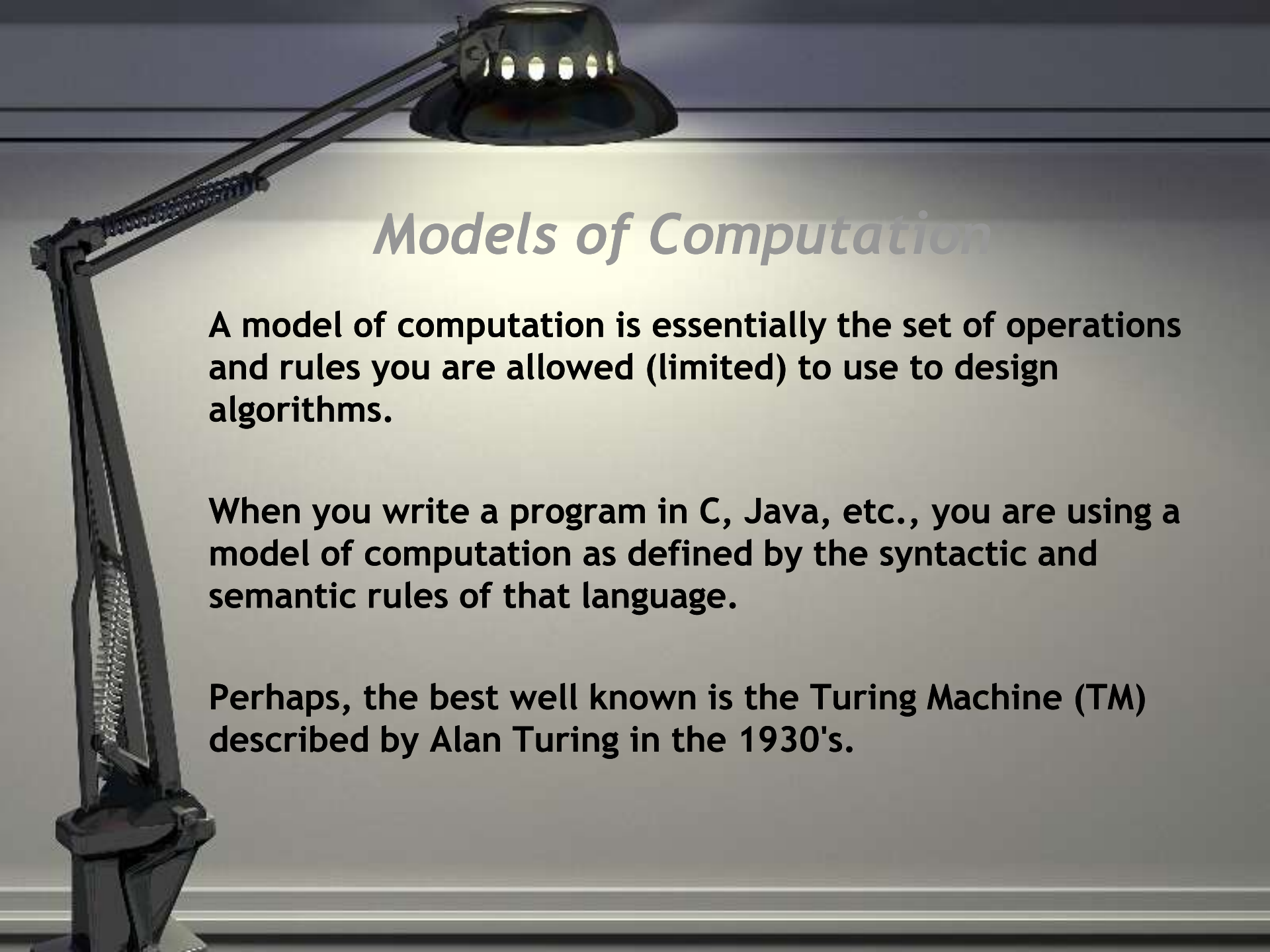
- A related classification scheme has arisen in recent years - Fixed Parameter Tractability.
- The idea is to acknowledge that a problem is hard (and probably exponential), and to ask "what makes the problem hard?"
- In some hard problems, it has been observed that many, and perhaps most, instances are easily solvable. It is only a few (proportionally few, but still infinite) that cause algorithms to take exponential time.
- Can we isolate those instances, design algorithms that work well most of the time?



Computational Complexity

There are a lot of problems, perhaps most, that we can't seem to fit into any of these 3 classes.

We will build (define) other classes. The classes are intended to differentiate problems according to how easy/hard they are to solve depending upon the "computing power" (model of computation) used.

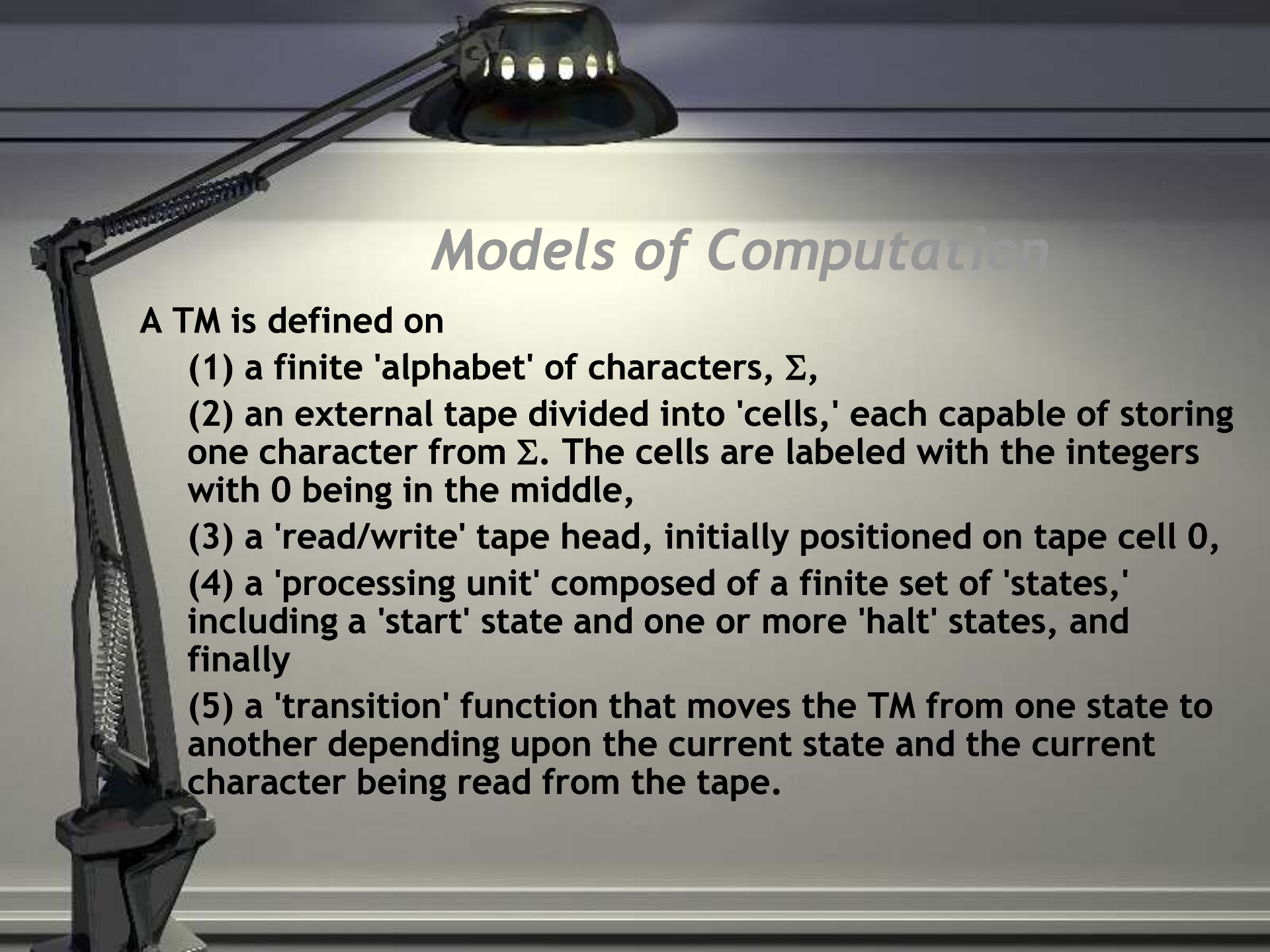


Models of Computation

A model of computation is essentially the set of operations and rules you are allowed (limited) to use to design algorithms.

When you write a program in C, Java, etc., you are using a model of computation as defined by the syntactic and semantic rules of that language.

Perhaps, the best well known is the Turing Machine (TM) described by Alan Turing in the 1930's.



Models of Computation

A TM is defined on

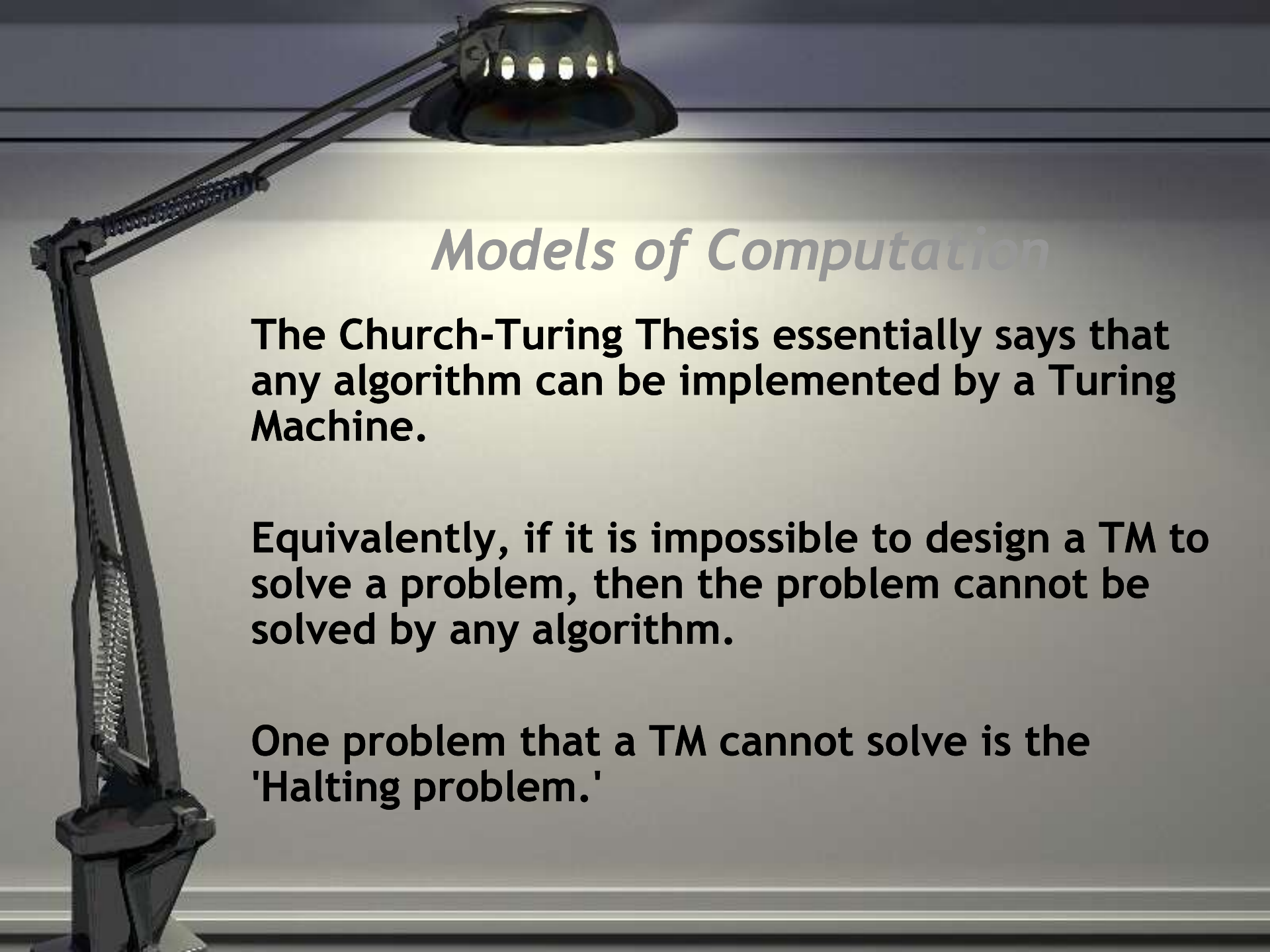
(1) a finite 'alphabet' of characters, Σ,

(2) an external tape divided into 'cells,' each capable of storing one character from Σ. The cells are labeled with the integers with 0 being in the middle,

(3) a 'read/write' tape head, initially positioned on tape cell 0,

(4) a 'processing unit' composed of a finite set of 'states,' including a 'start' state and one or more 'halt' states, and finally

(5) a 'transition' function that moves the TM from one state to another depending upon the current state and the current character being read from the tape.

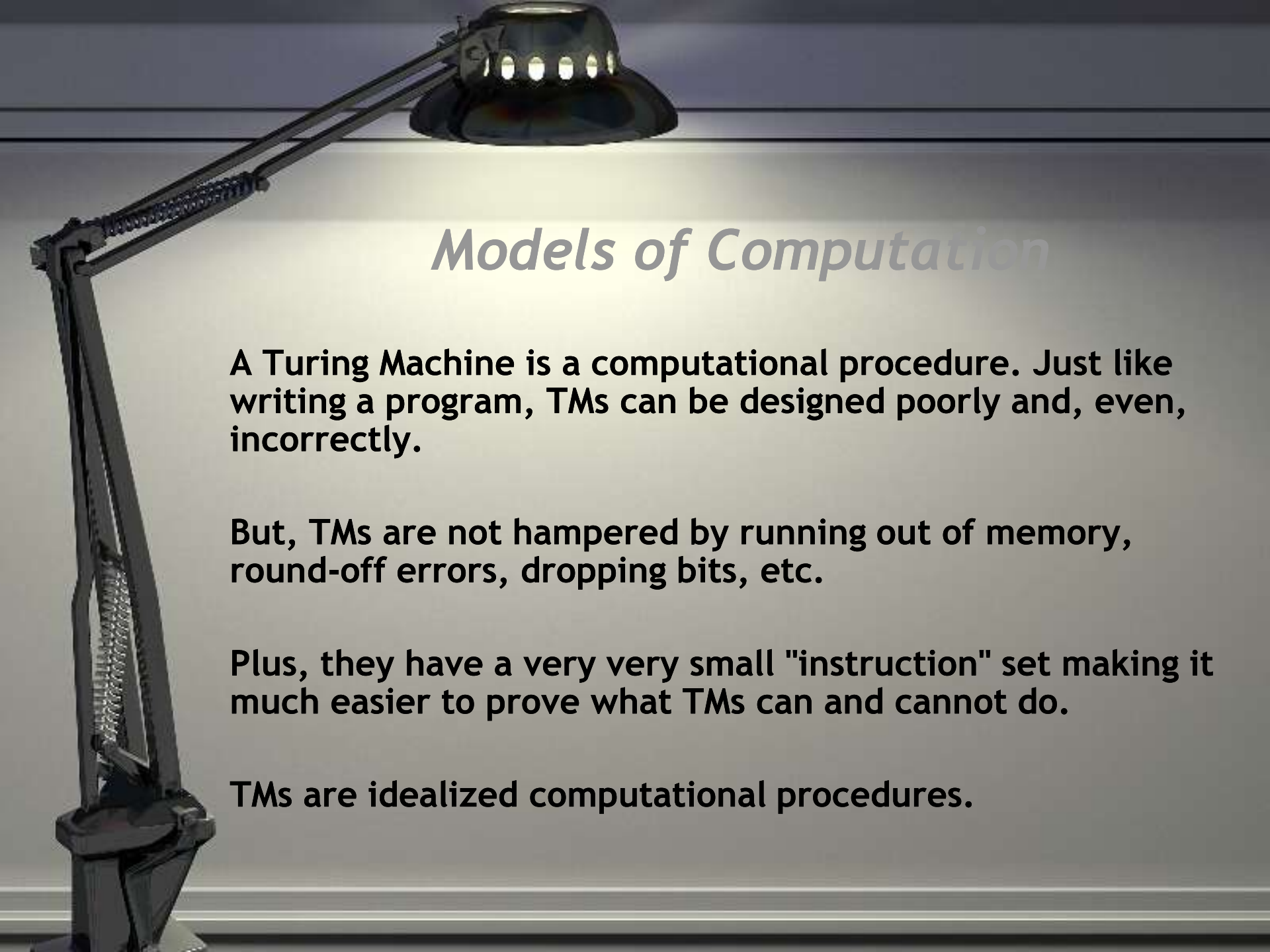


Models of Computation

The Church-Turing Thesis essentially says that any algorithm can be implemented by a Turing Machine.

Equivalently, if it is impossible to design a TM to solve a problem, then the problem cannot be solved by any algorithm.

One problem that a TM cannot solve is the 'Halting problem.'



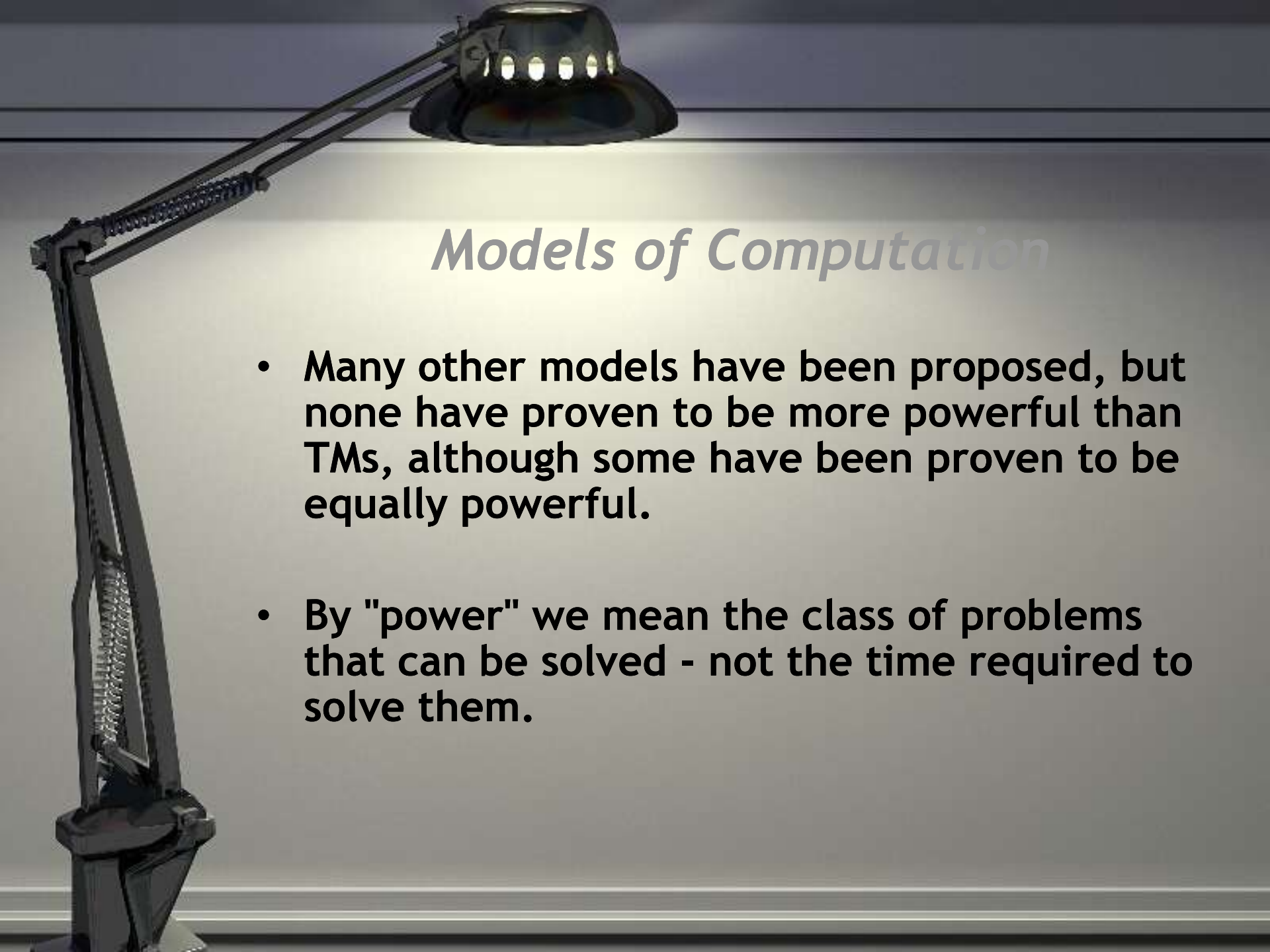
Models of Computation

A Turing Machine is a computational procedure. Just like writing a program, TMs can be designed poorly and, even, incorrectly.

But, TMs are not hampered by running out of memory, round-off errors, dropping bits, etc.

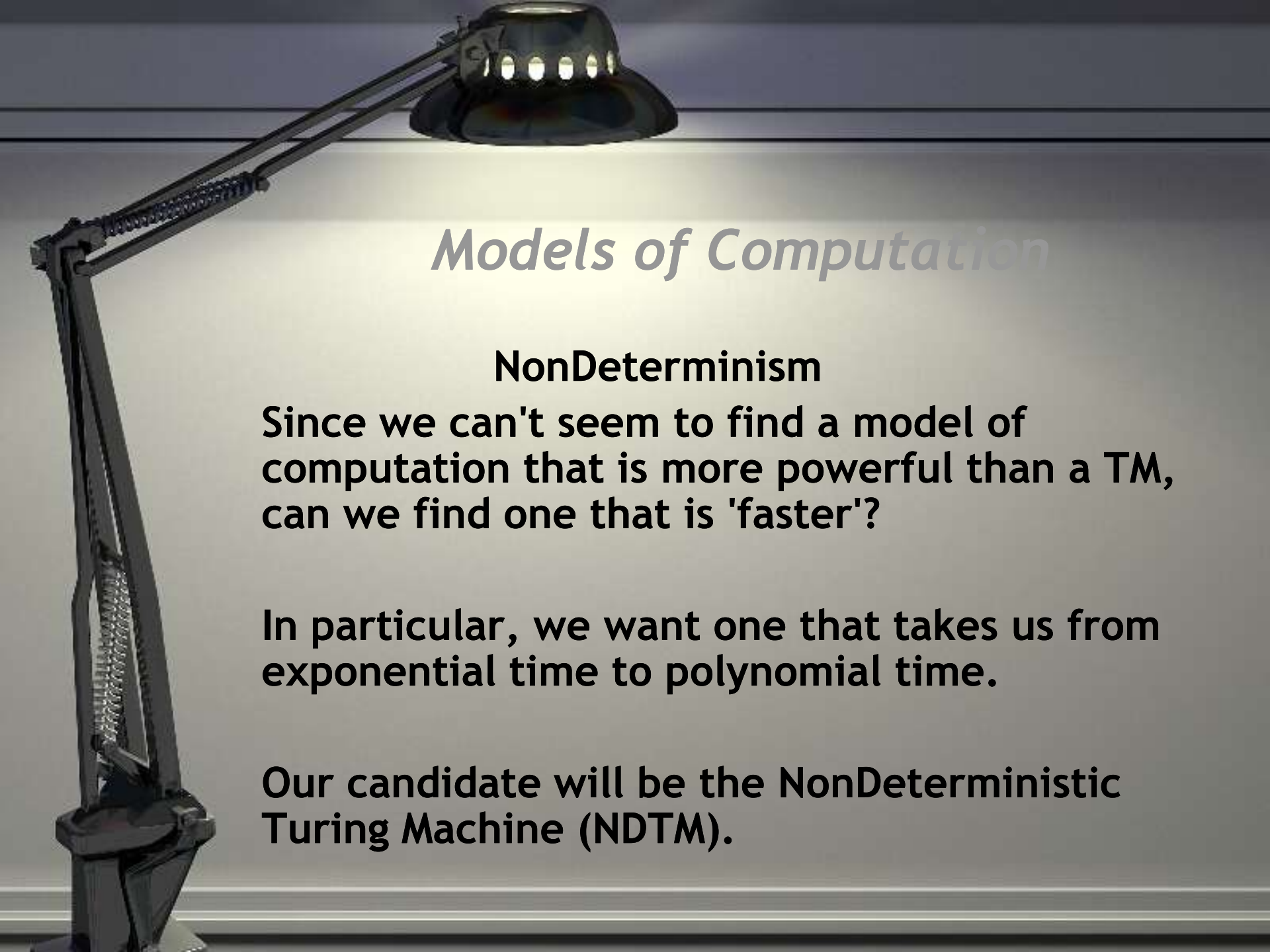
Plus, they have a very very small "instruction" set making it much easier to prove what TMs can and cannot do.

TMs are idealized computational procedures.



Models of Computation

- Many other models have been proposed, but none have proven to be more powerful than TMs, although some have been proven to be equally powerful.
- By "power" we mean the class of problems that can be solved - not the time required to solve them.



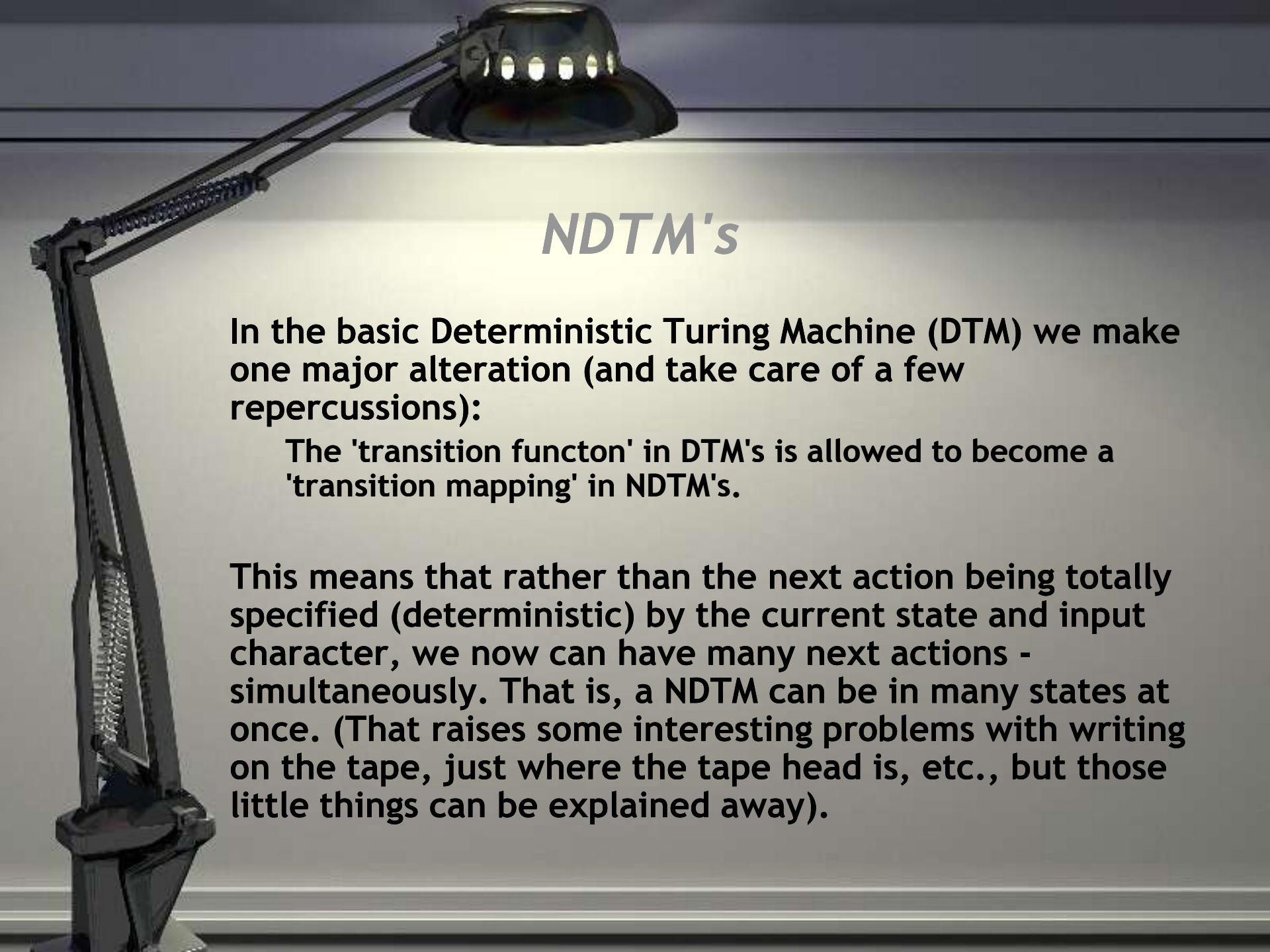
Models of Computation

NonDeterminism

Since we can't seem to find a model of computation that is more powerful than a TM, can we find one that is 'faster'?

In particular, we want one that takes us from exponential time to polynomial time.

Our candidate will be the NonDeterministic Turing Machine (NDTM).

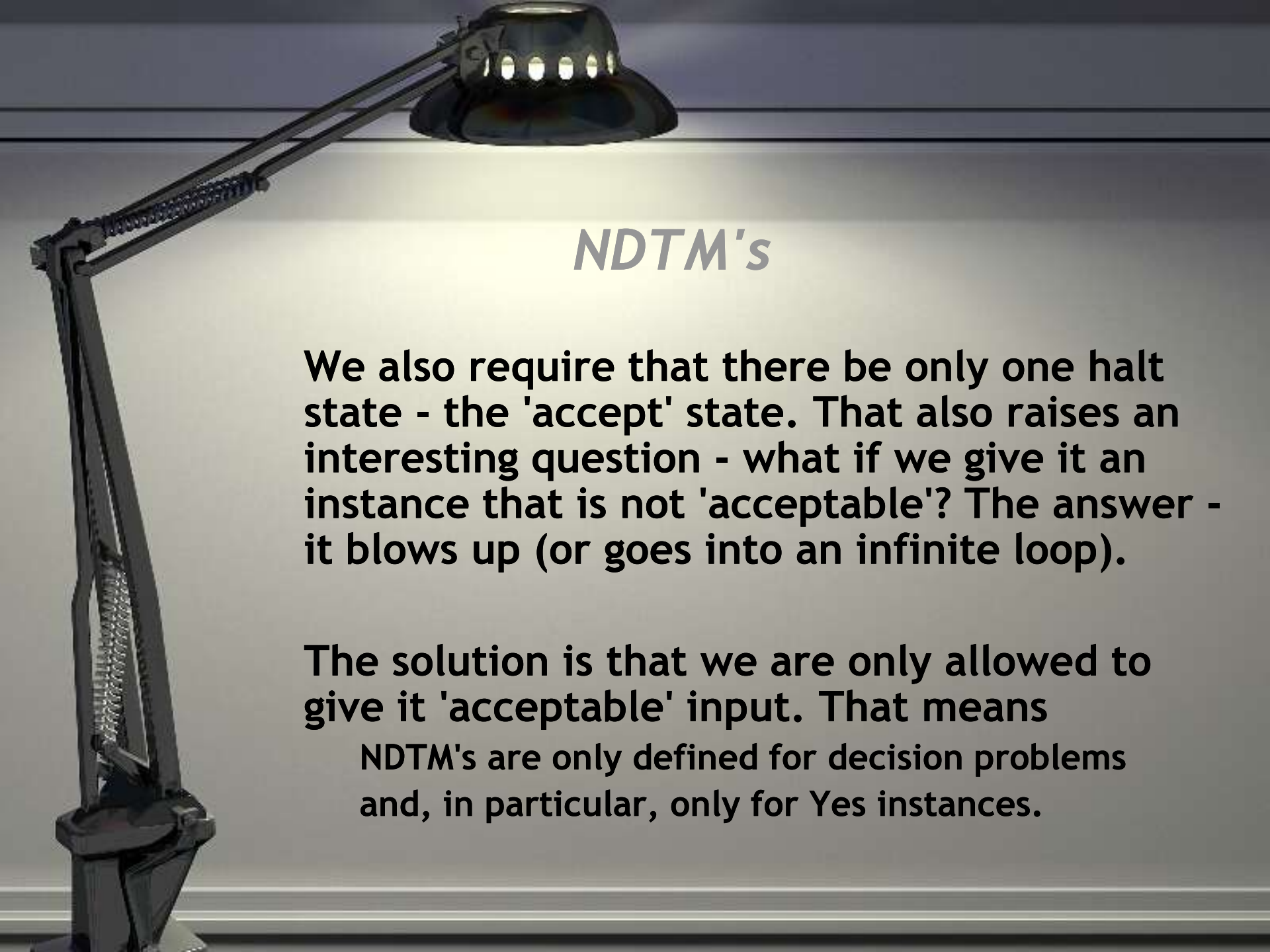


NDTM's

In the basic Deterministic Turing Machine (DTM) we make one major alteration (and take care of a few repercussions):

> The 'transition functon' in DTM's is allowed to become a 'transition mapping' in NDTM's.

This means that rather than the next action being totally specified (deterministic) by the current state and input character, we now can have many next actions - simultaneously. That is, a NDTM can be in many states at once. (That raises some interesting problems with writing on the tape, just where the tape head is, etc., but those little things can be explained away).

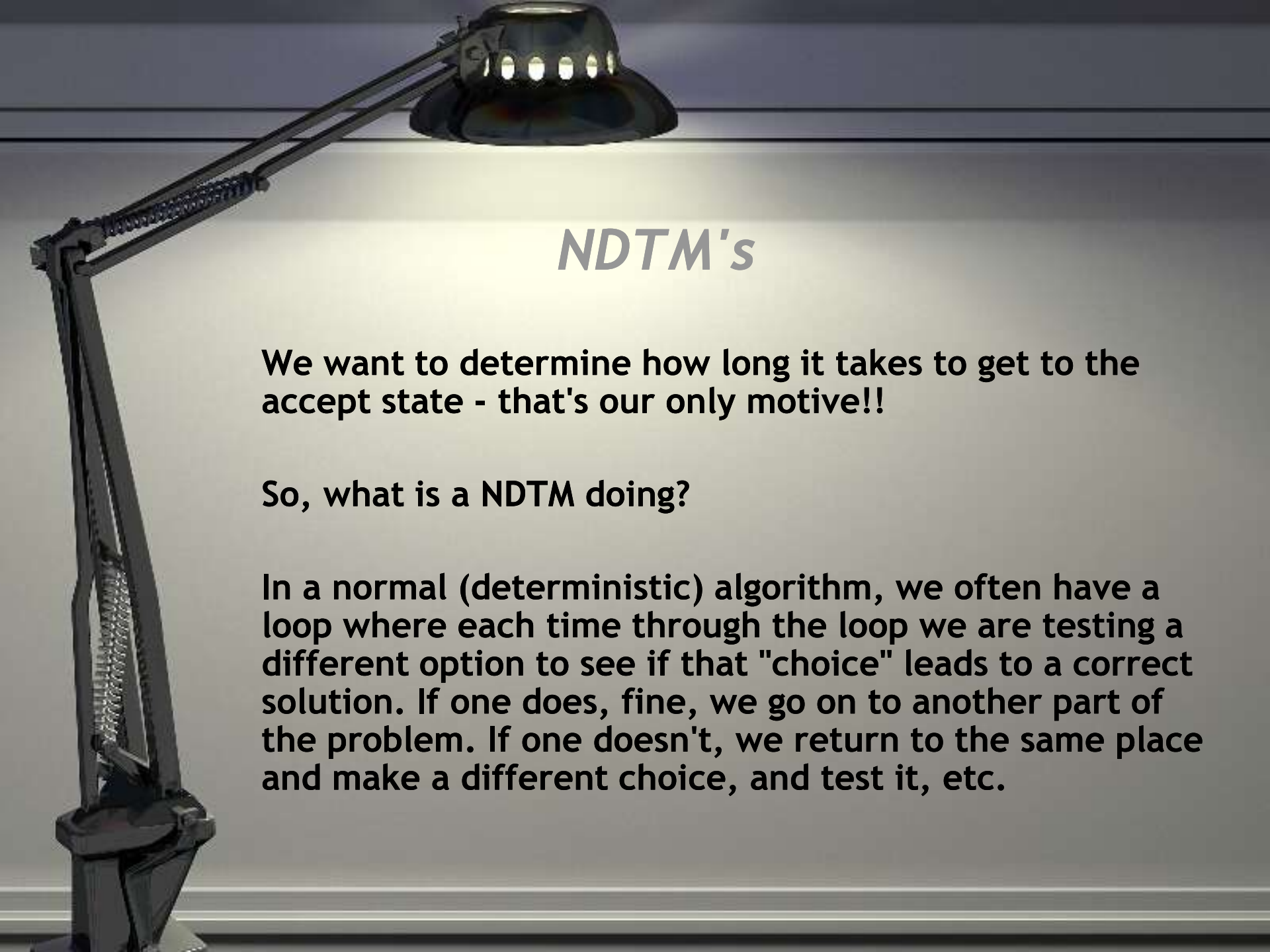


NDTM's

We also require that there be only one halt state - the 'accept' state. That also raises an interesting question - what if we give it an instance that is not 'acceptable'? The answer - it blows up (or goes into an infinite loop).

The solution is that we are only allowed to give it 'acceptable' input. That means

- NDTM's are only defined for decision problems and, in particular, only for Yes instances.

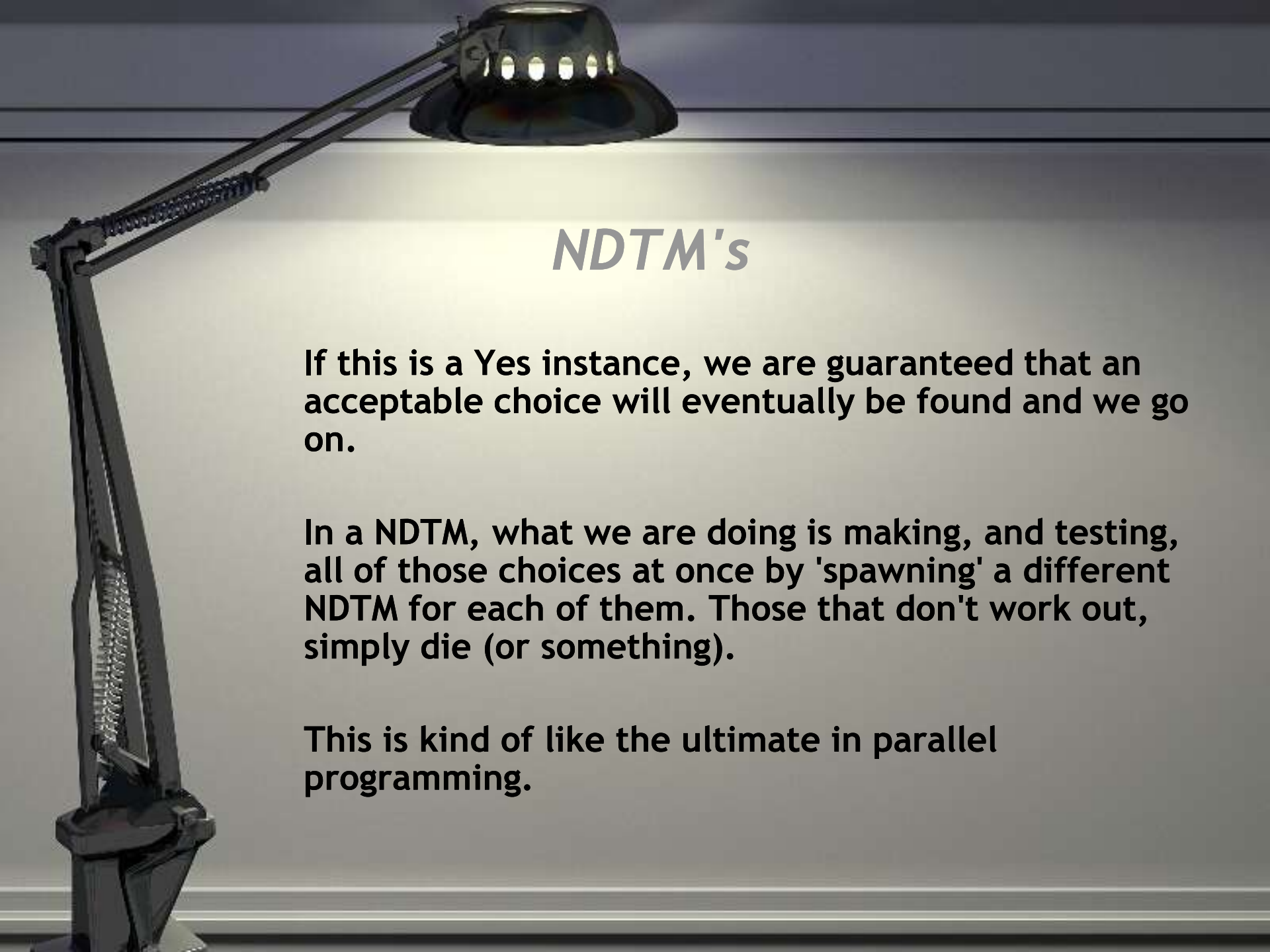


NDTM's

We want to determine how long it takes to get to the accept state - that's our only motive!!

So, what is a NDTM doing?

In a normal (deterministic) algorithm, we often have a loop where each time through the loop we are testing a different option to see if that "choice" leads to a correct solution. If one does, fine, we go on to another part of the problem. If one doesn't, we return to the same place and make a different choice, and test it, etc.

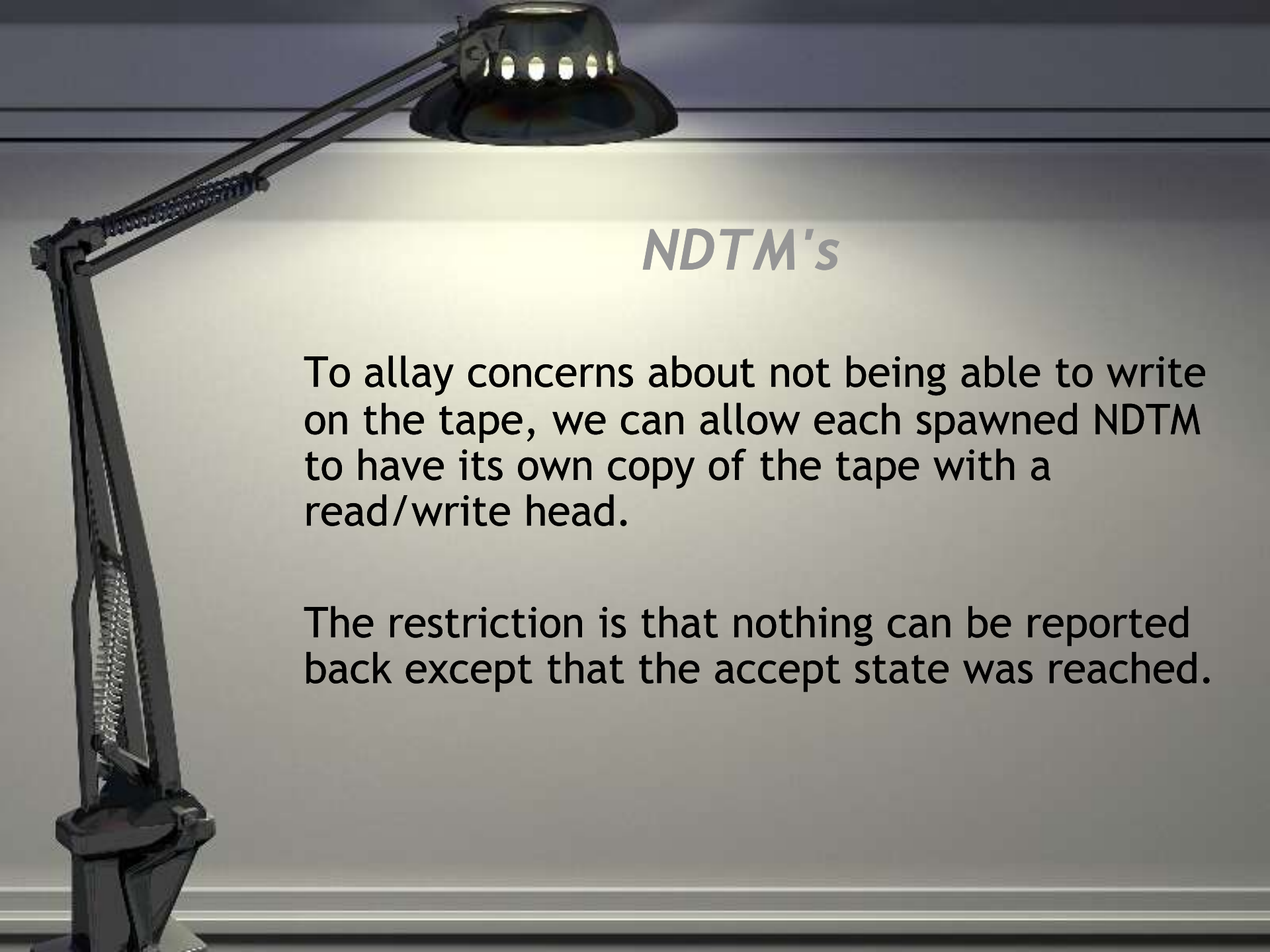


NDTM's

If this is a Yes instance, we are guaranteed that an acceptable choice will eventually be found and we go on.

In a NDTM, what we are doing is making, and testing, all of those choices at once by 'spawning' a different NDTM for each of them. Those that don't work out, simply die (or something).

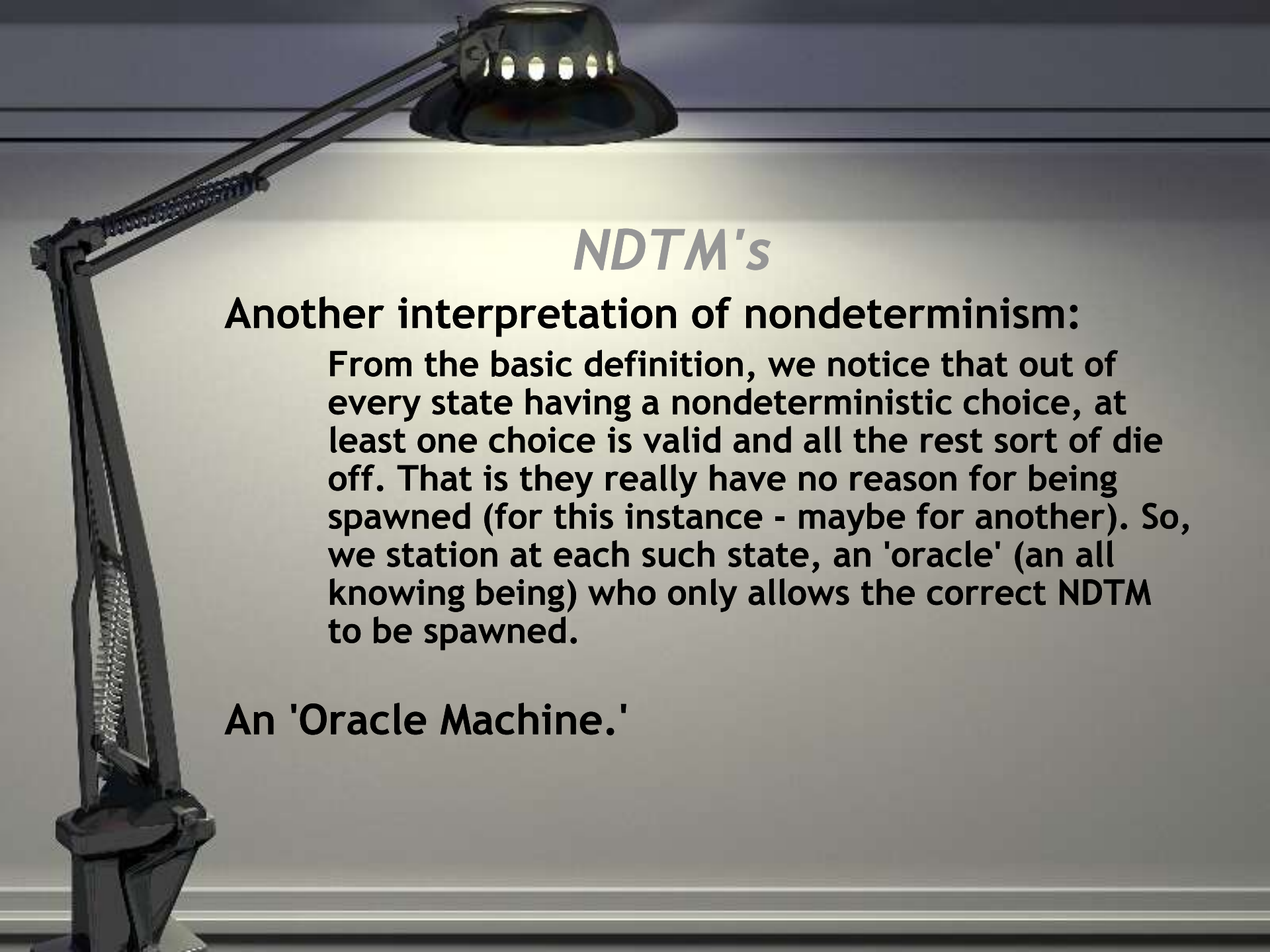
This is kind of like the ultimate in parallel programming.



NDTM's

To allay concerns about not being able to write on the tape, we can allow each spawned NDTM to have its own copy of the tape with a read/write head.

The restriction is that nothing can be reported back except that the accept state was reached.

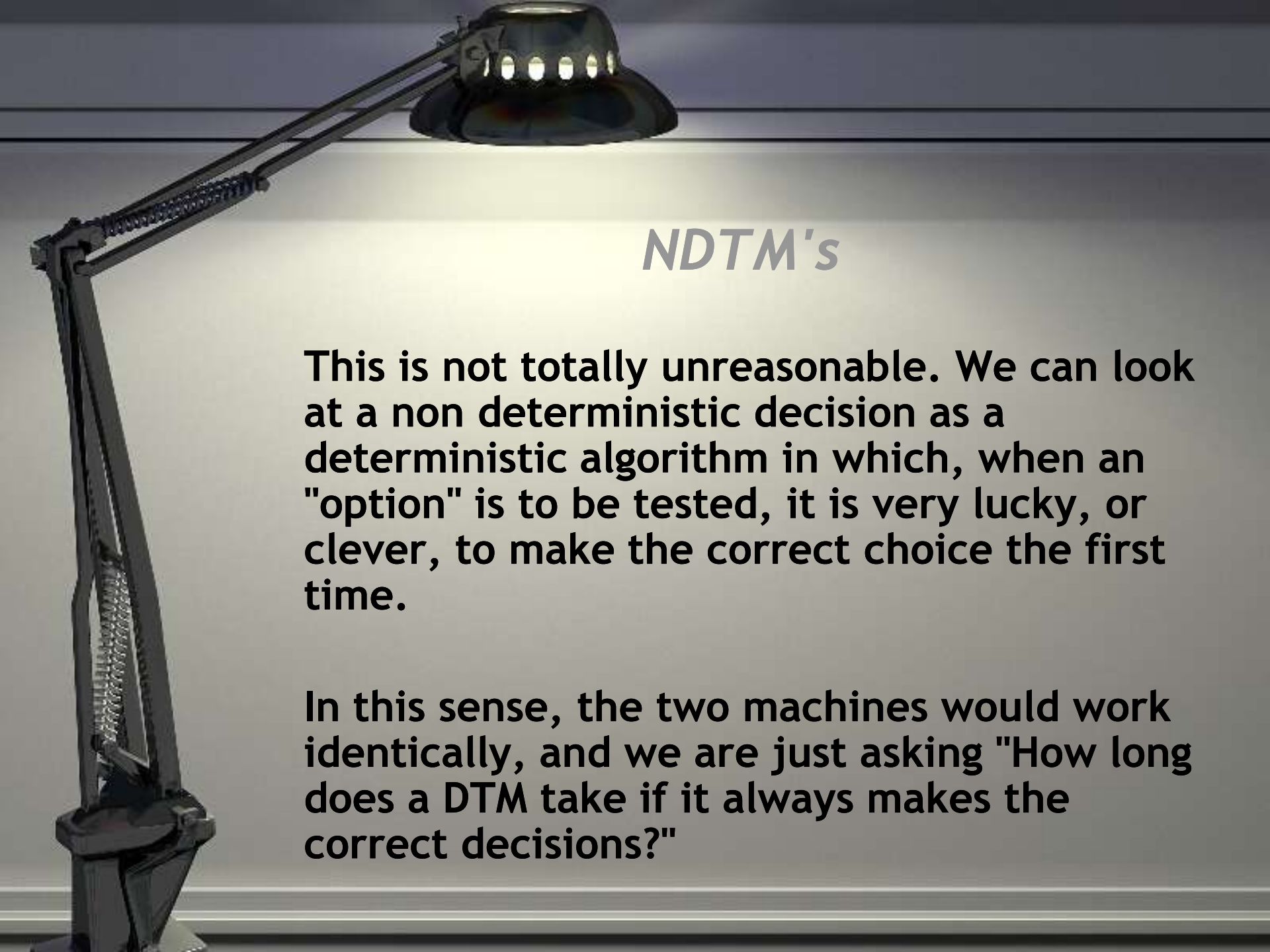


NDTM's

Another interpretation of nondeterminism:

From the basic definition, we notice that out of every state having a nondeterministic choice, at least one choice is valid and all the rest sort of die off. That is they really have no reason for being spawned (for this instance - maybe for another). So, we station at each such state, an 'oracle' (an all knowing being) who only allows the correct NDTM to be spawned.

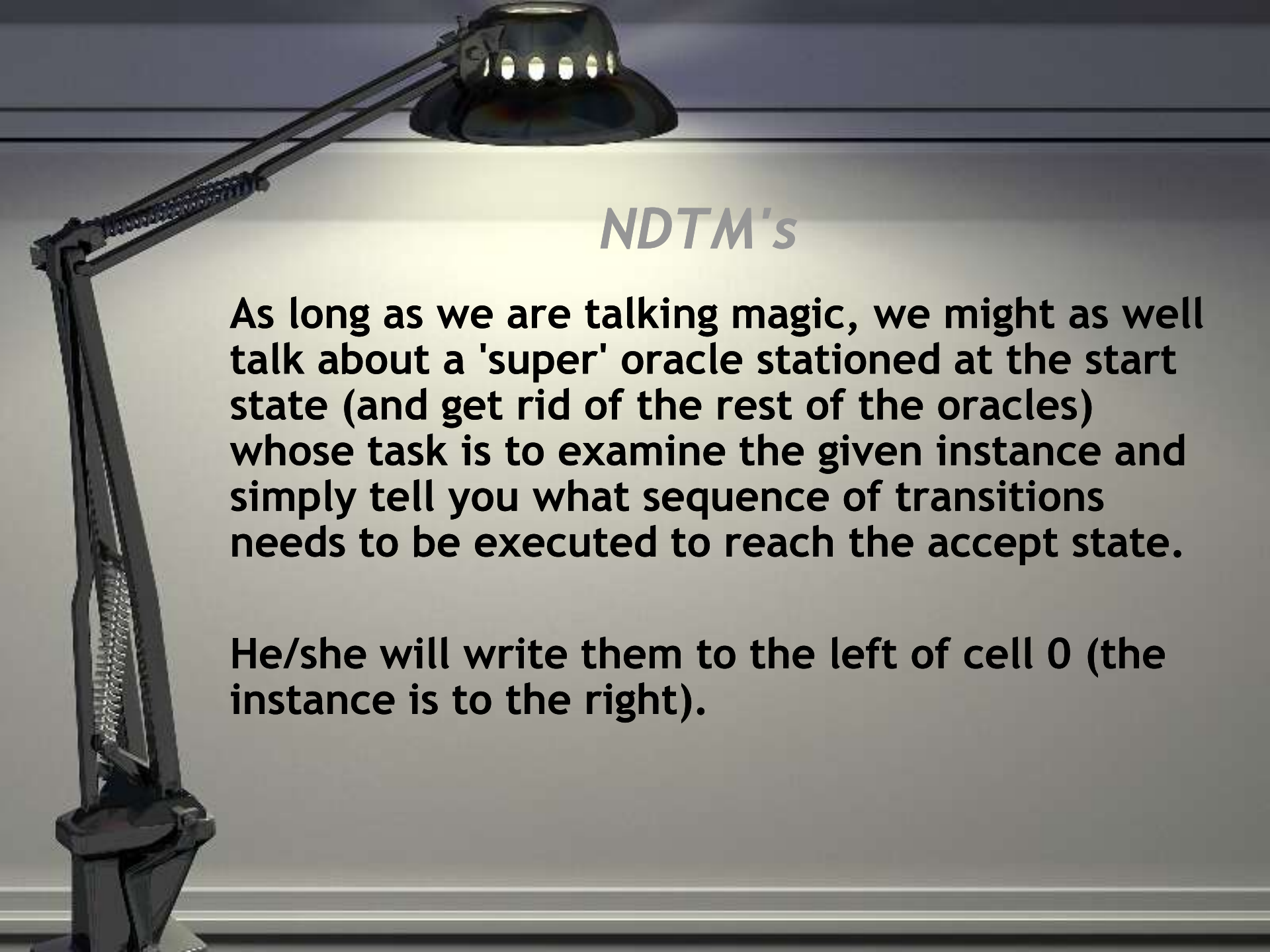
An 'Oracle Machine.'



NDTM's

This is not totally unreasonable. We can look at a non deterministic decision as a deterministic algorithm in which, when an "option" is to be tested, it is very lucky, or clever, to make the correct choice the first time.

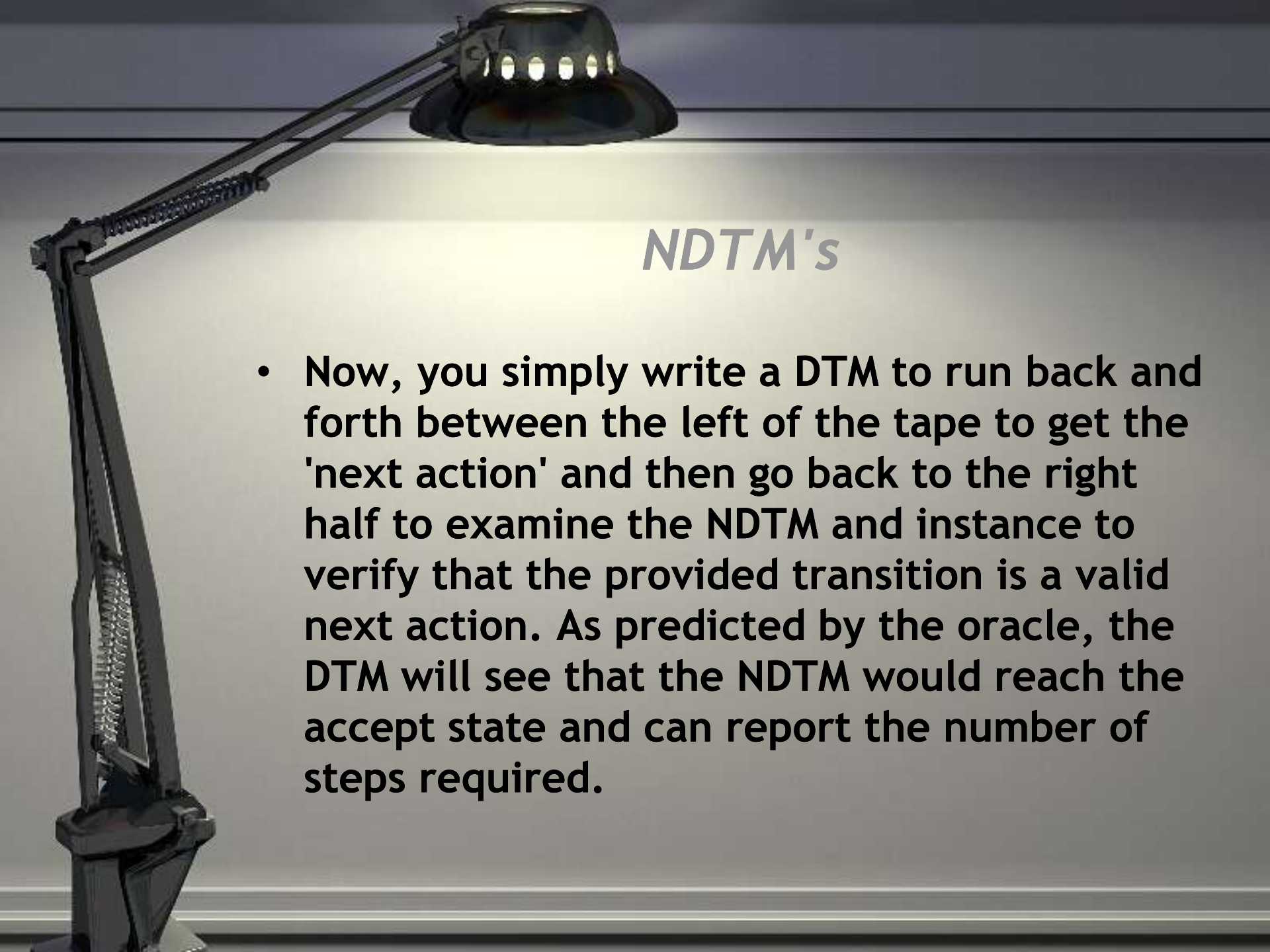
In this sense, the two machines would work identically, and we are just asking "How long does a DTM take if it always makes the correct decisions?"



NDTM's

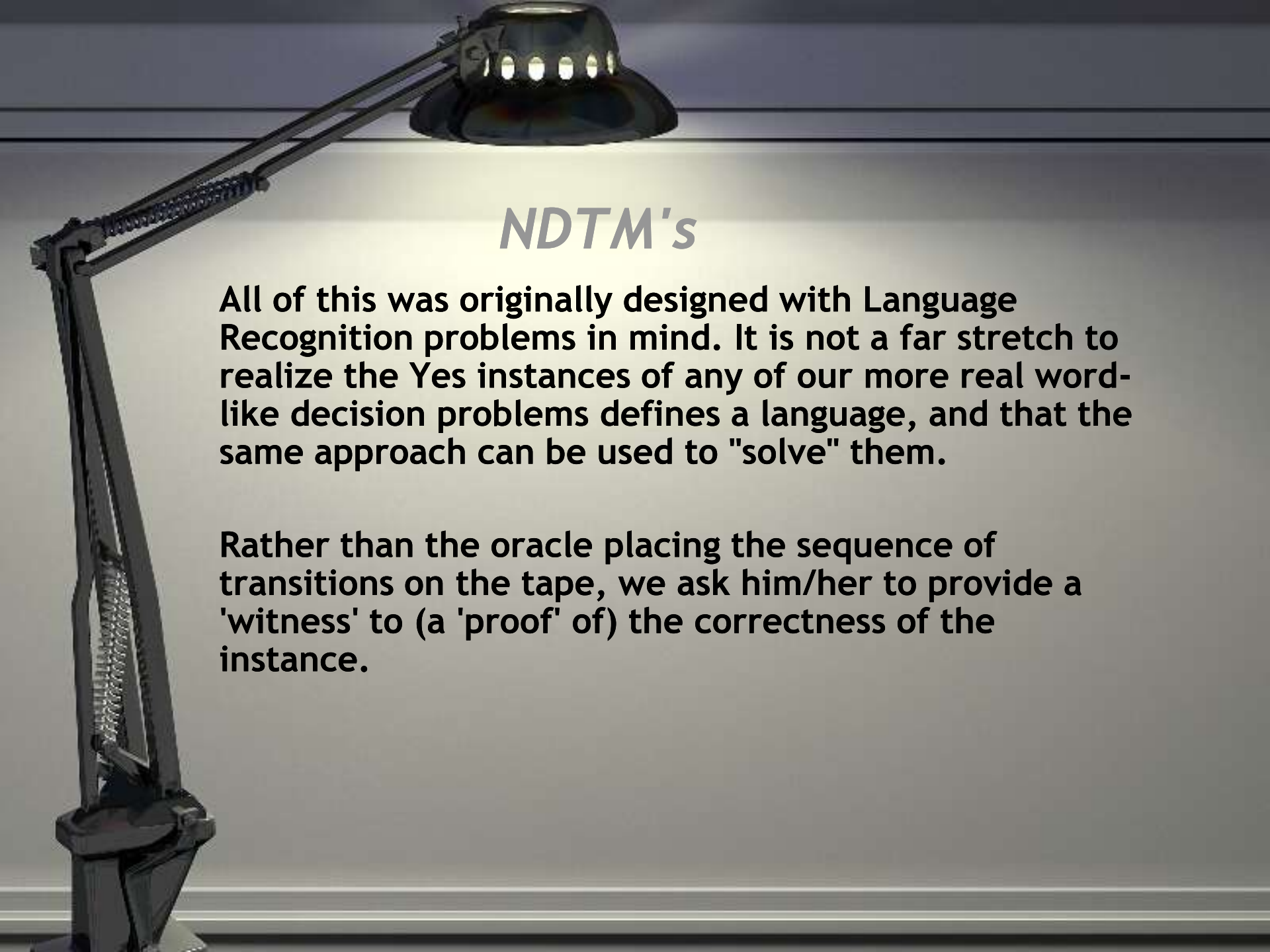
As long as we are talking magic, we might as well talk about a 'super' oracle stationed at the start state (and get rid of the rest of the oracles) whose task is to examine the given instance and simply tell you what sequence of transitions needs to be executed to reach the accept state.

He/she will write them to the left of cell 0 (the instance is to the right).



NDTM's

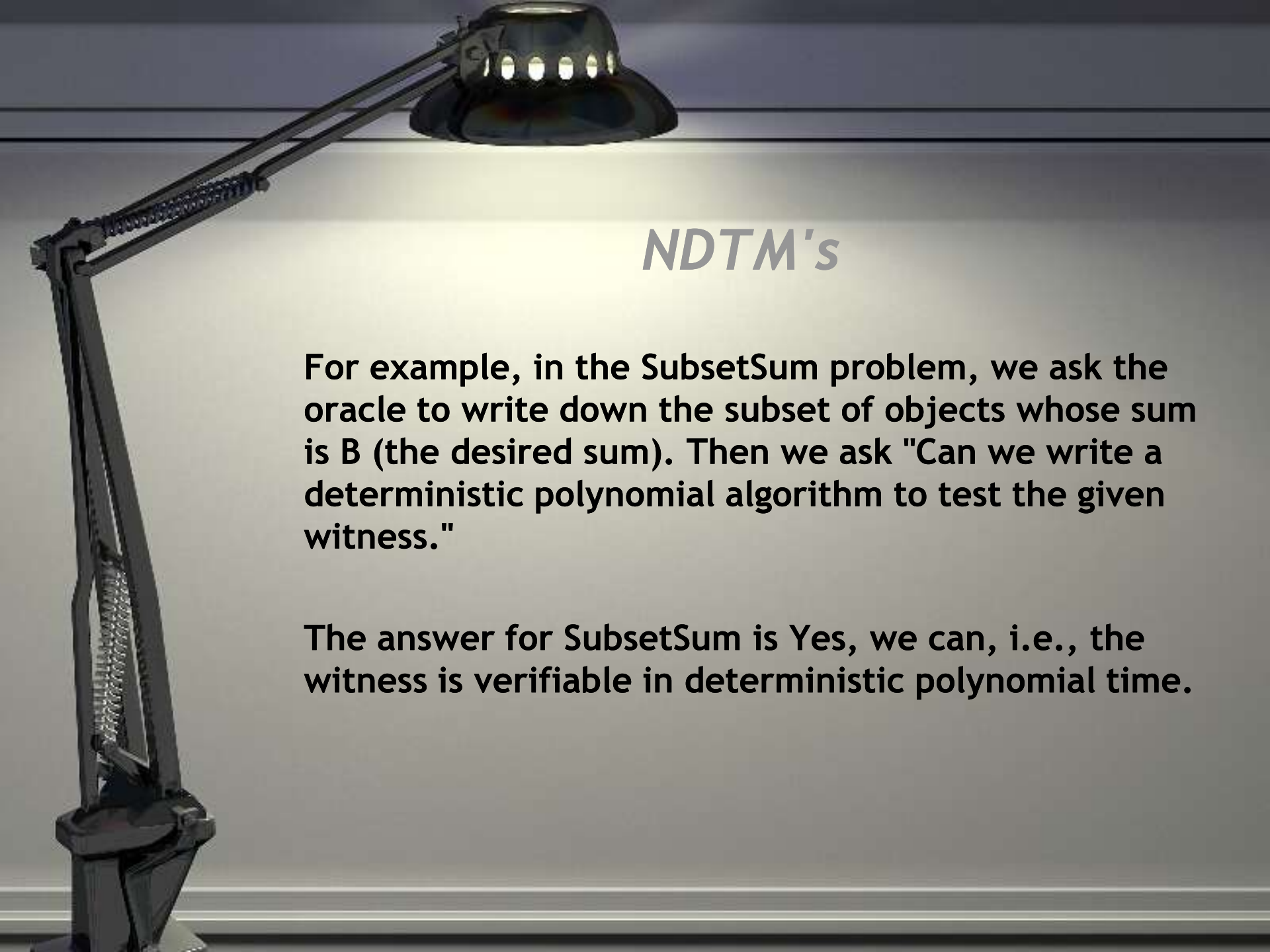
- Now, you simply write a DTM to run back and forth between the left of the tape to get the 'next action' and then go back to the right half to examine the NDTM and instance to verify that the provided transition is a valid next action. As predicted by the oracle, the DTM will see that the NDTM would reach the accept state and can report the number of steps required.



NDTM's

All of this was originally designed with Language Recognition problems in mind. It is not a far stretch to realize the Yes instances of any of our more real word-like decision problems defines a language, and that the same approach can be used to "solve" them.

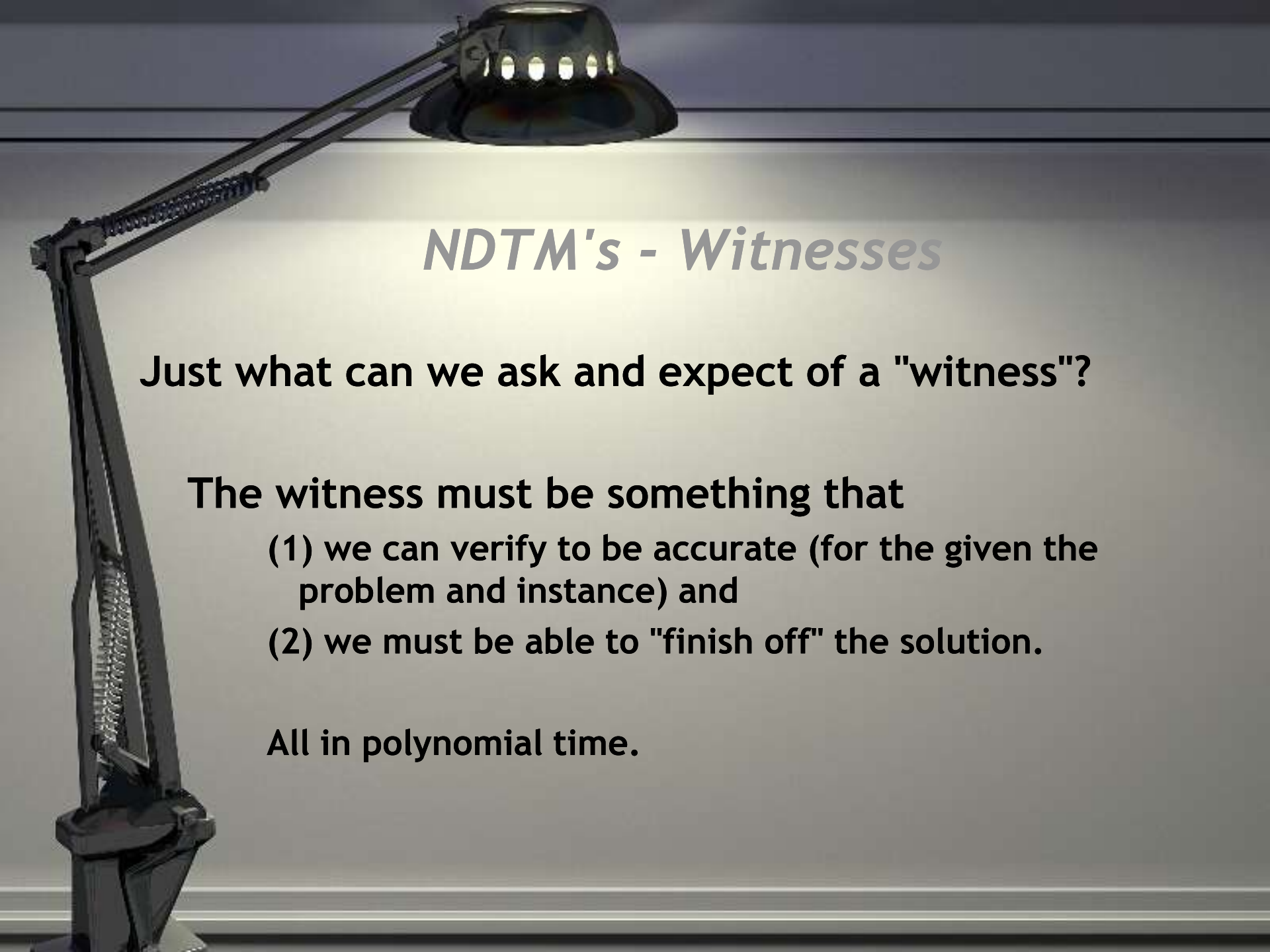
Rather than the oracle placing the sequence of transitions on the tape, we ask him/her to provide a 'witness' to (a 'proof' of) the correctness of the instance.



NDTM's

For example, in the SubsetSum problem, we ask the oracle to write down the subset of objects whose sum is B (the desired sum). Then we ask "Can we write a deterministic polynomial algorithm to test the given witness."

The answer for SubsetSum is Yes, we can, i.e., the witness is verifiable in deterministic polynomial time.



NDTM's - Witnesses

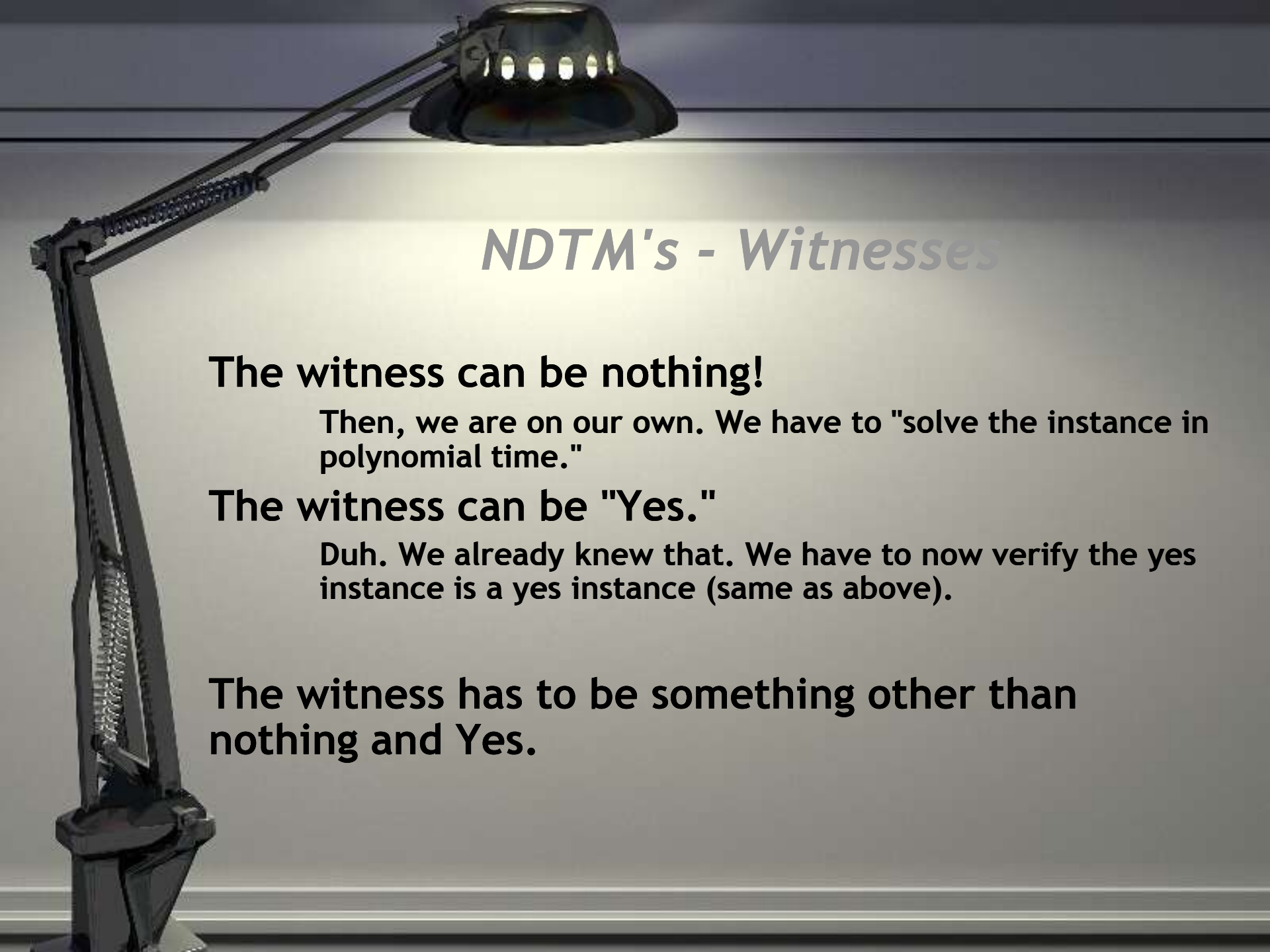
Just what can we ask and expect of a "witness"?

The witness must be something that

(1) we can verify to be accurate (for the given the problem and instance) and

(2) we must be able to "finish off" the solution.

All in polynomial time.



NDTM's - Witnesses

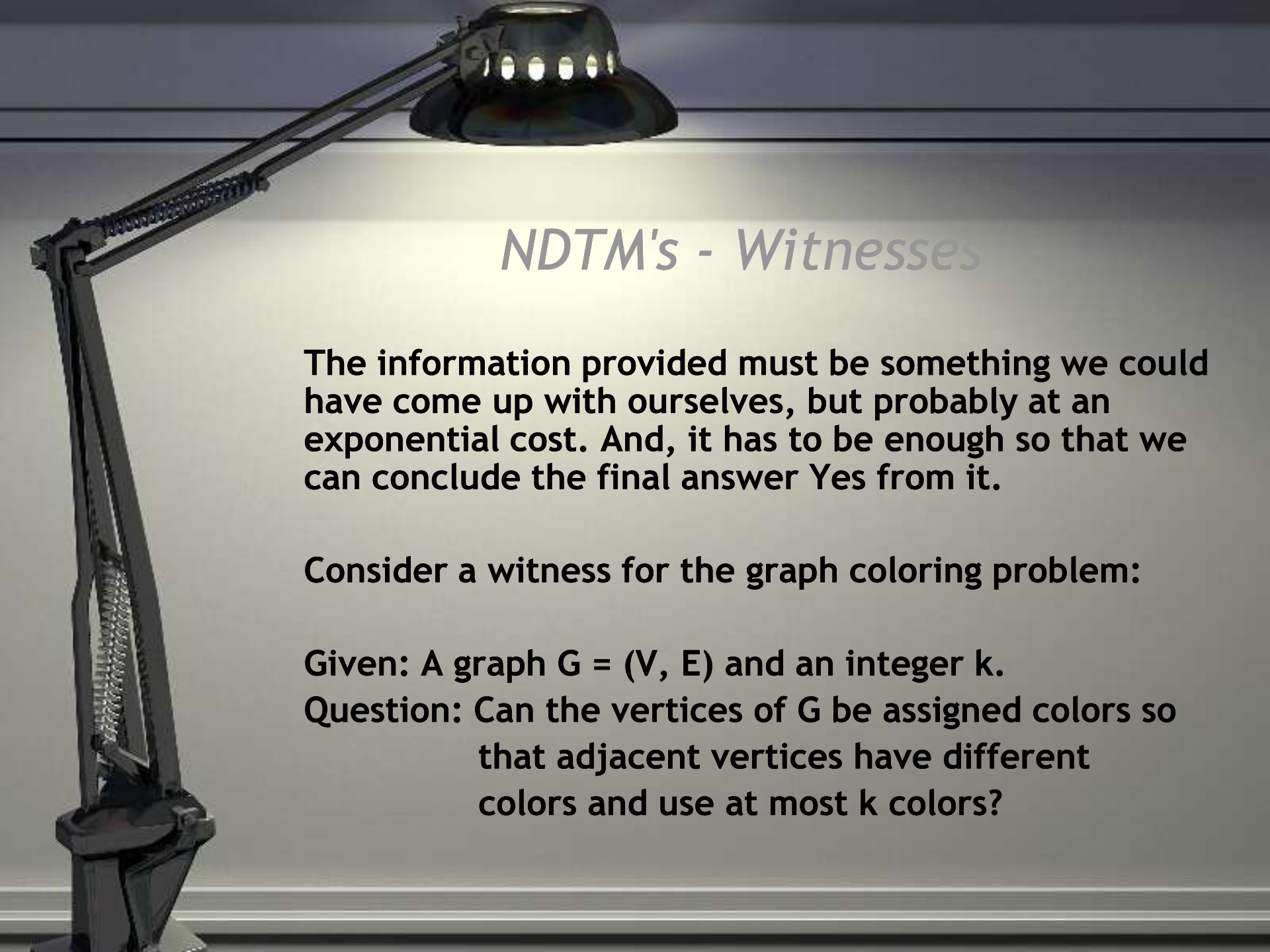
The witness can be nothing!

> **Then, we are on our own. We have to "solve the instance in polynomial time."**

The witness can be "Yes."

> **Duh. We already knew that. We have to now verify the yes instance is a yes instance (same as above).**

The witness has to be something other than nothing and Yes.

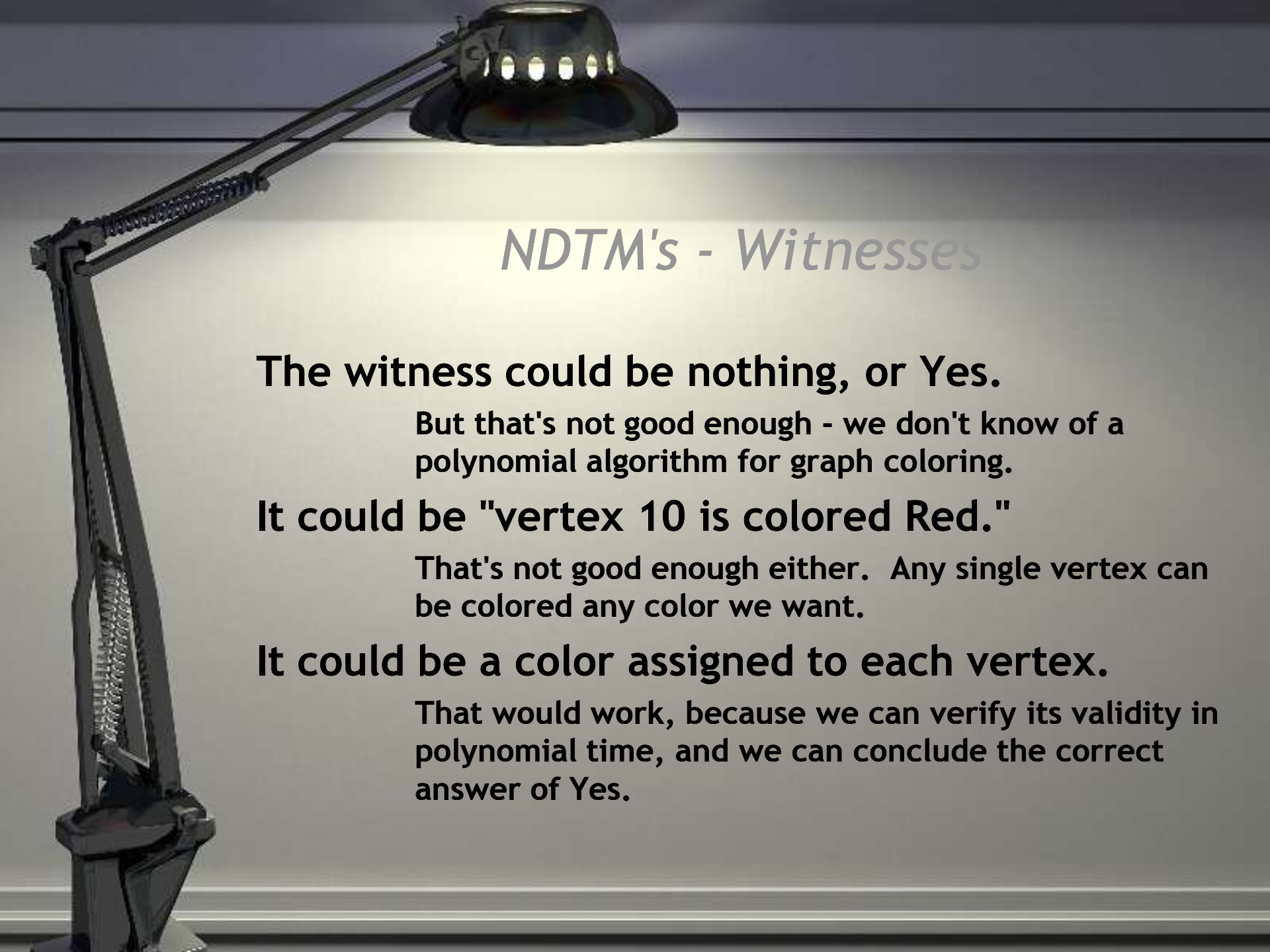


NDTM's - Witnesses

The information provided must be something we could have come up with ourselves, but probably at an exponential cost. And, it has to be enough so that we can conclude the final answer Yes from it.

Consider a witness for the graph coloring problem:

Given: A graph G = (V, E) and an integer k.
Question: Can the vertices of G be assigned colors so that adjacent vertices have different colors and use at most k colors?



NDTM's - Witnesses

The witness could be nothing, or Yes.

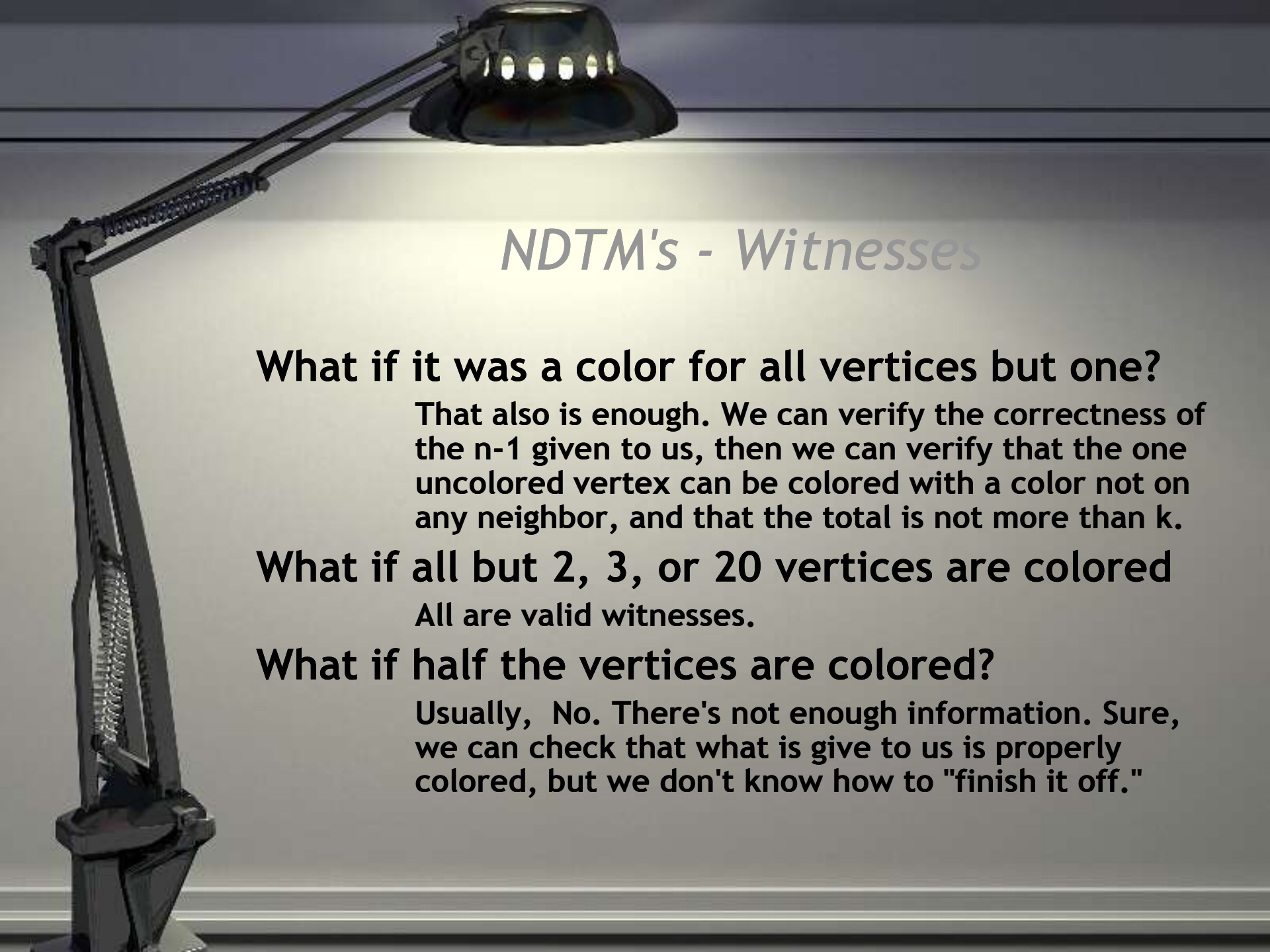
But that's not good enough - we don't know of a polynomial algorithm for graph coloring.

It could be "vertex 10 is colored Red."

That's not good enough either. Any single vertex can be colored any color we want.

It could be a color assigned to each vertex.

That would work, because we can verify its validity in polynomial time, and we can conclude the correct answer of Yes.



NDTM's - Witnesses

What if it was a color for all vertices but one?

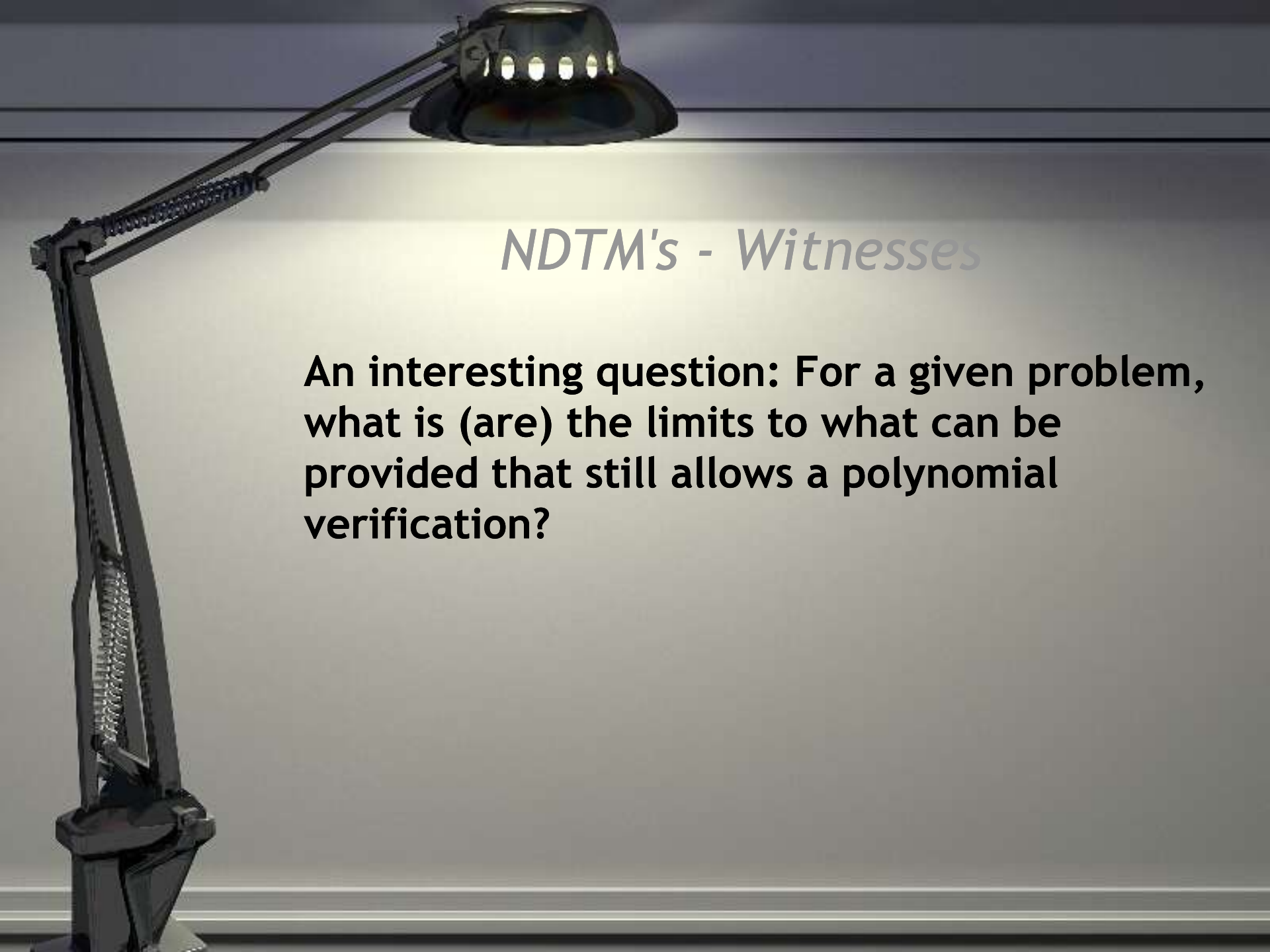
That also is enough. We can verify the correctness of the n-1 given to us, then we can verify that the one uncolored vertex can be colored with a color not on any neighbor, and that the total is not more than k.

What if all but 2, 3, or 20 vertices are colored

All are valid witnesses.

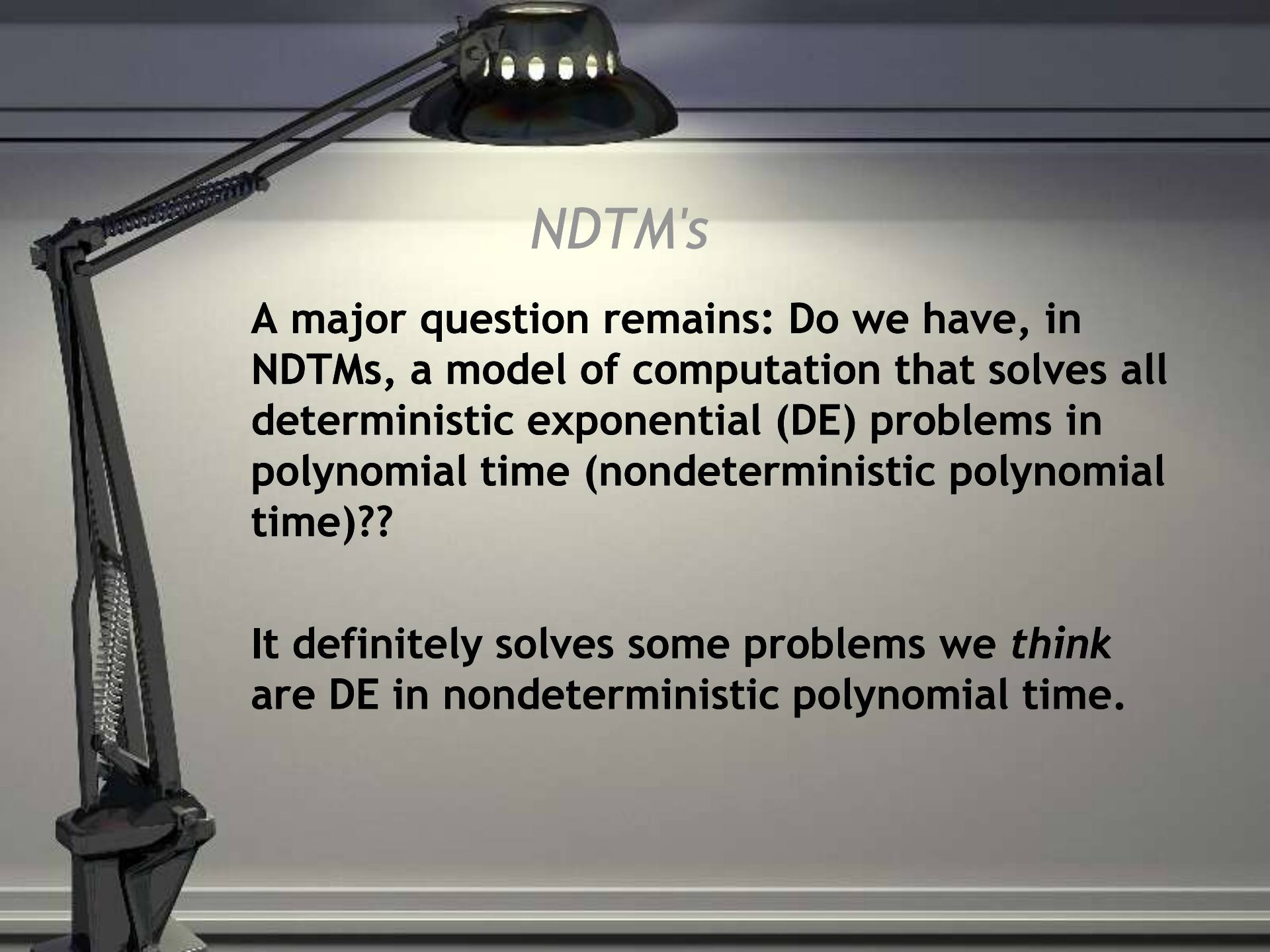
What if half the vertices are colored?

Usually, No. There's not enough information. Sure, we can check that what is give to us is properly colored, but we don't know how to "finish it off."



NDTM's - Witnesses

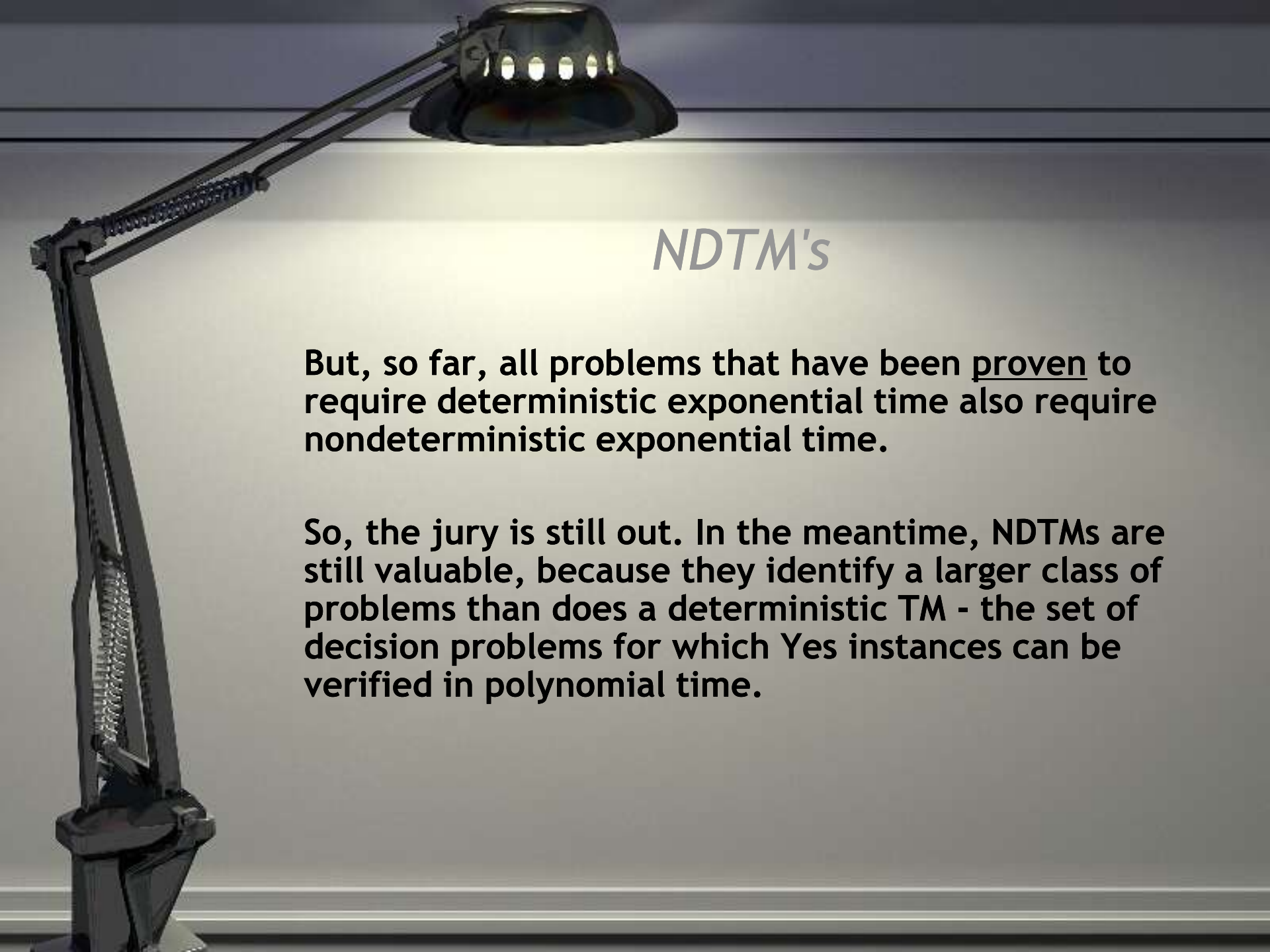
An interesting question: For a given problem, what is (are) the limits to what can be provided that still allows a polynomial verification?



NDTM's

A major question remains: Do we have, in NDTMs, a model of computation that solves all deterministic exponential (DE) problems in polynomial time (nondeterministic polynomial time)??

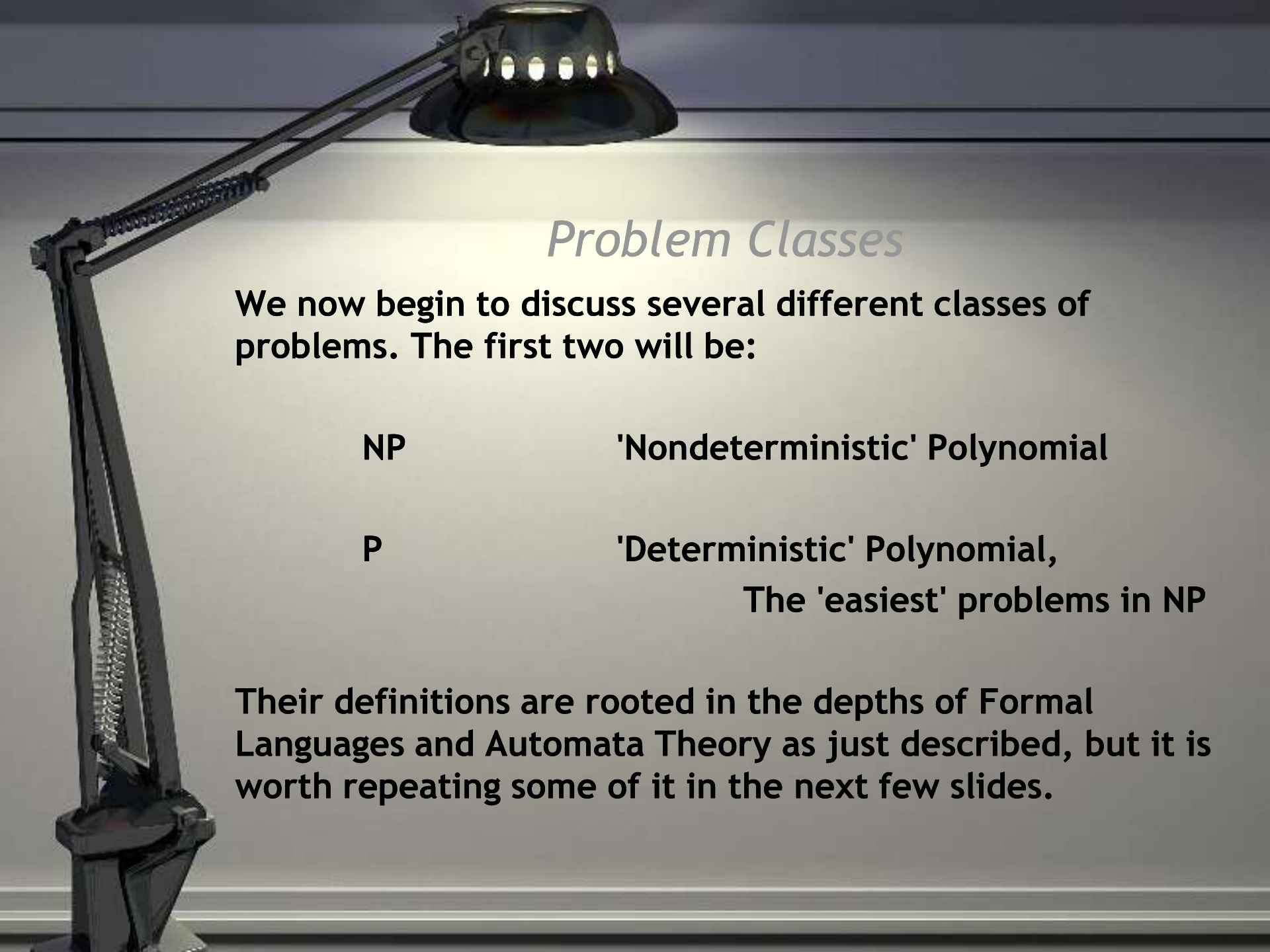
It definitely solves some problems we *think* are DE in nondeterministic polynomial time.



NDTM's

But, so far, all problems that have been proven to require deterministic exponential time also require nondeterministic exponential time.

So, the jury is still out. In the meantime, NDTMs are still valuable, because they identify a larger class of problems than does a deterministic TM - the set of decision problems for which Yes instances can be verified in polynomial time.



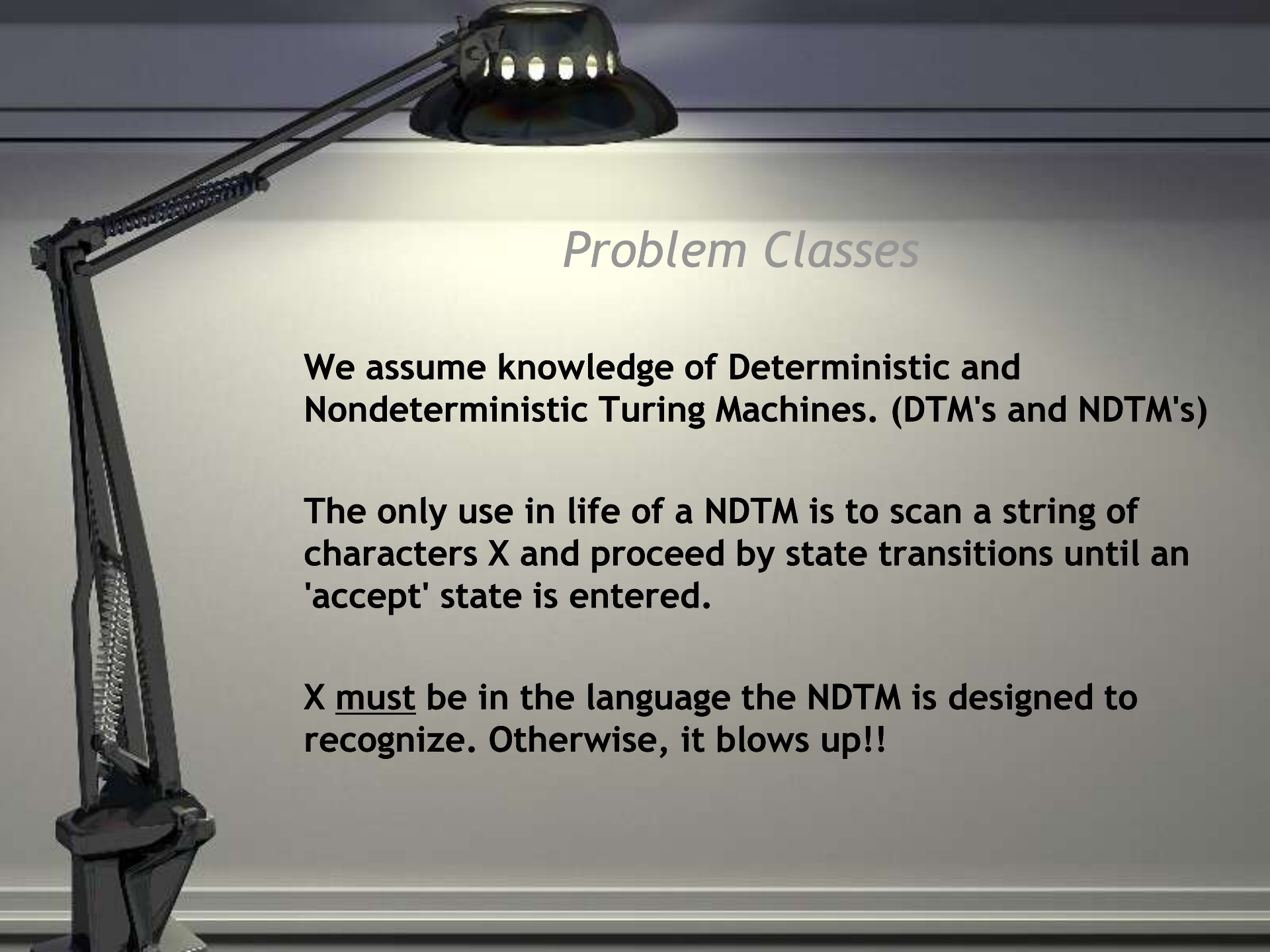
Problem Classes

We now begin to discuss several different classes of problems. The first two will be:

NP **'Nondeterministic' Polynomial**

P **'Deterministic' Polynomial,**
The 'easiest' problems in NP

Their definitions are rooted in the depths of Formal Languages and Automata Theory as just described, but it is worth repeating some of it in the next few slides.

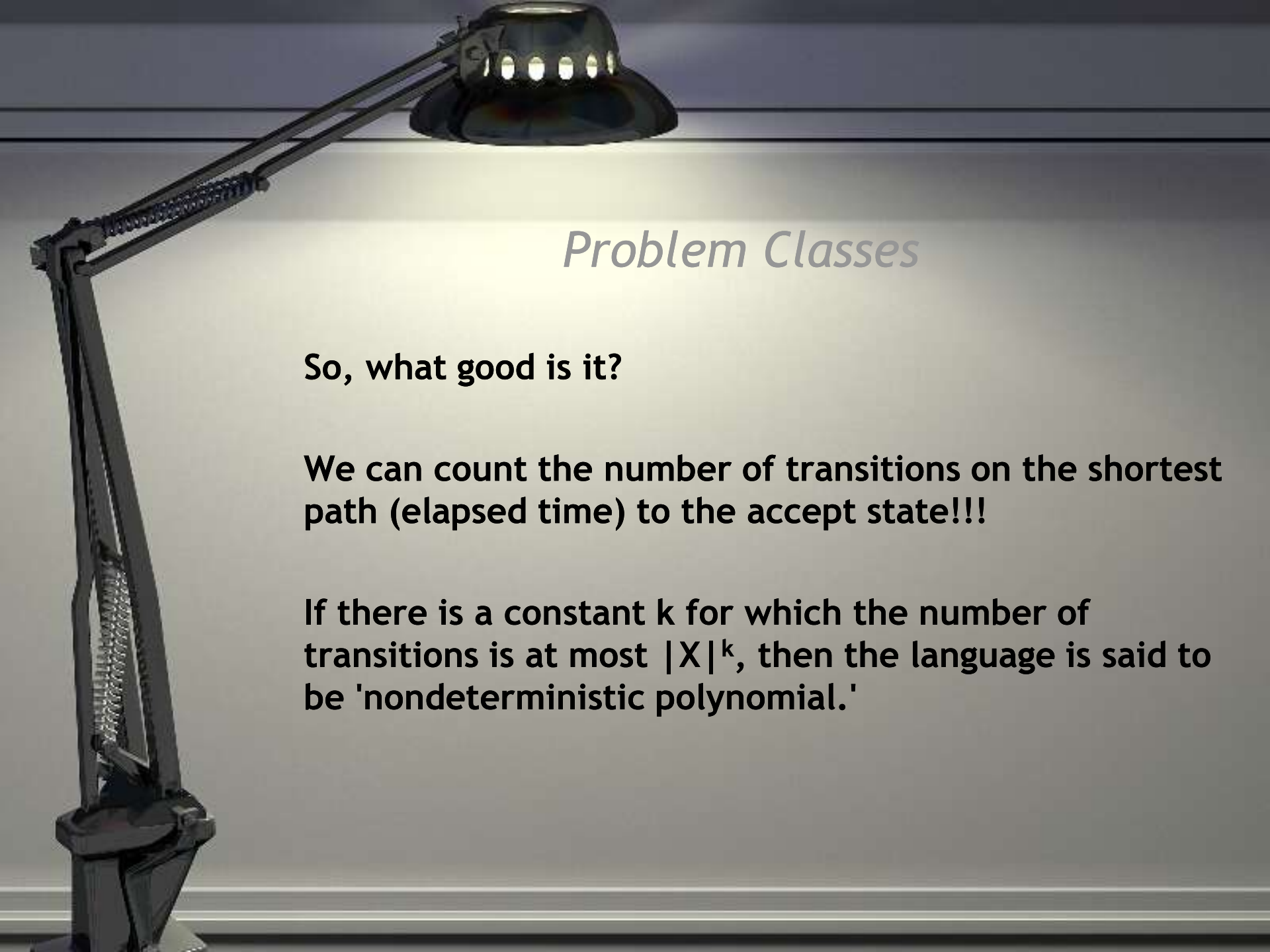


Problem Classes

We assume knowledge of Deterministic and Nondeterministic Turing Machines. (DTM's and NDTM's)

The only use in life of a NDTM is to scan a string of characters X and proceed by state transitions until an 'accept' state is entered.

X must be in the language the NDTM is designed to recognize. Otherwise, it blows up!!

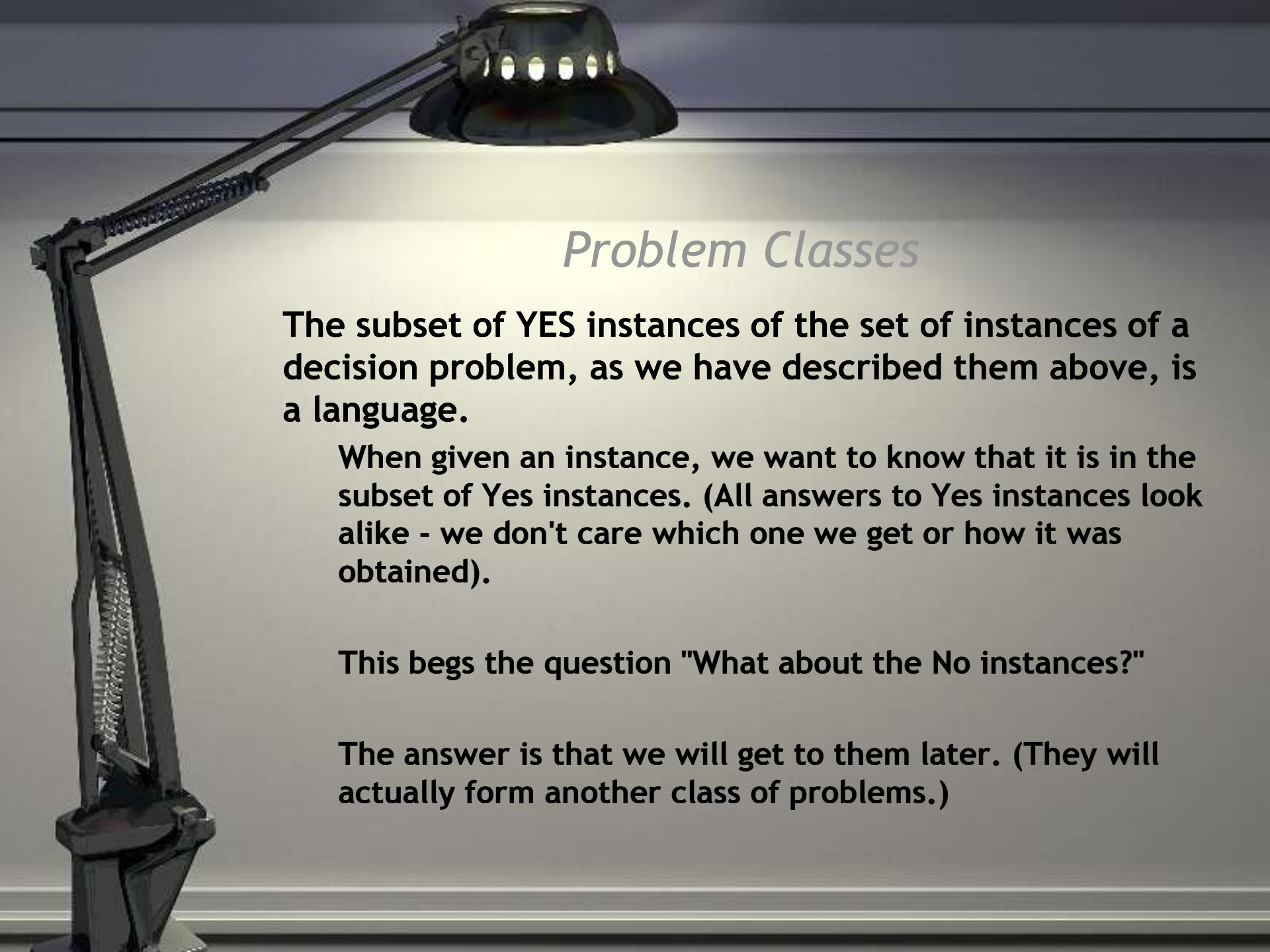


Problem Classes

So, what good is it?

We can count the number of transitions on the shortest path (elapsed time) to the accept state!!!

If there is a constant k for which the number of transitions is at most $|X|^k$, then the language is said to be 'nondeterministic polynomial.'



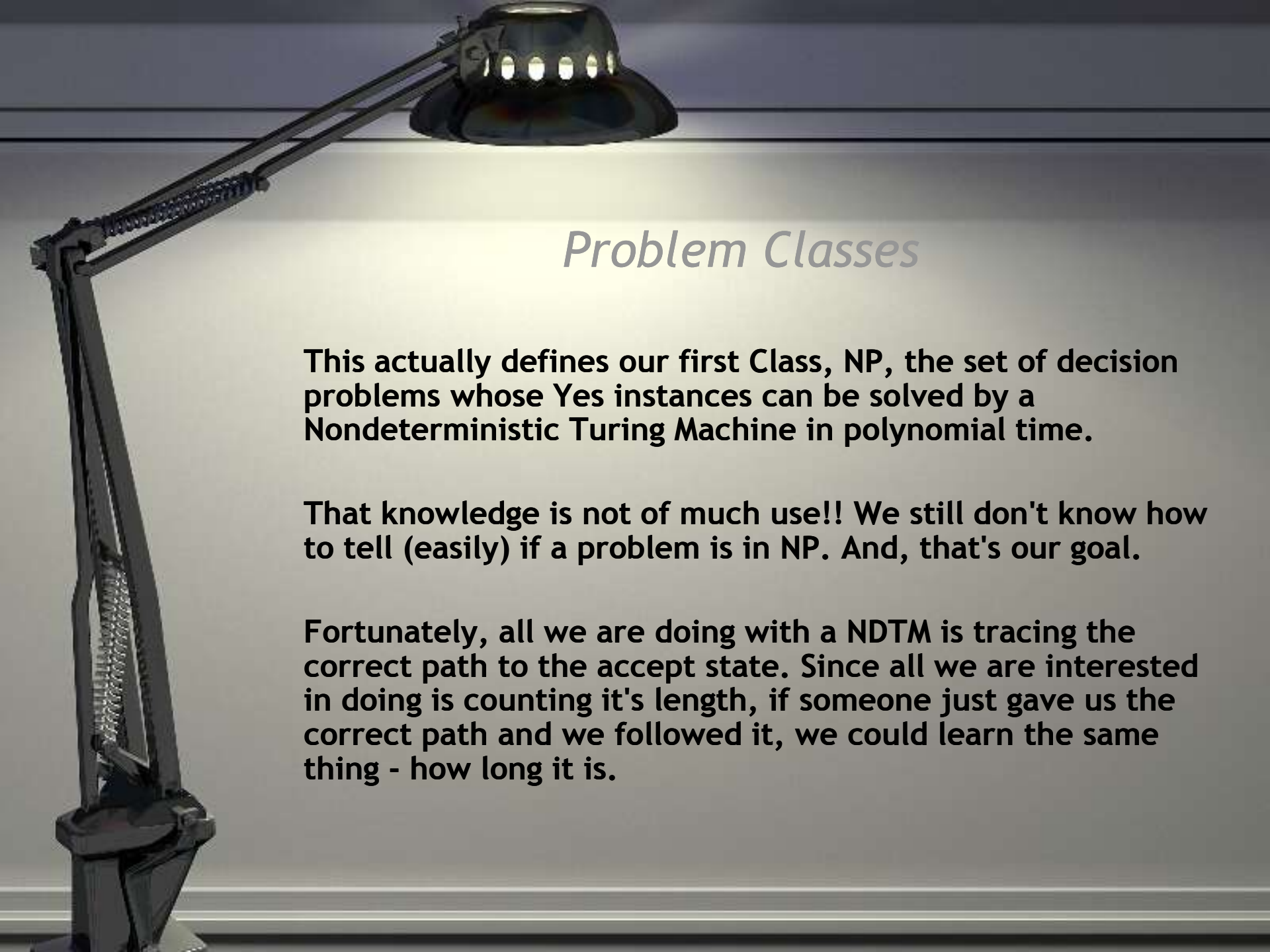
Problem Classes

The subset of YES instances of the set of instances of a decision problem, as we have described them above, is a language.

When given an instance, we want to know that it is in the subset of Yes instances. (All answers to Yes instances look alike - we don't care which one we get or how it was obtained).

This begs the question "What about the No instances?"

The answer is that we will get to them later. (They will actually form another class of problems.)

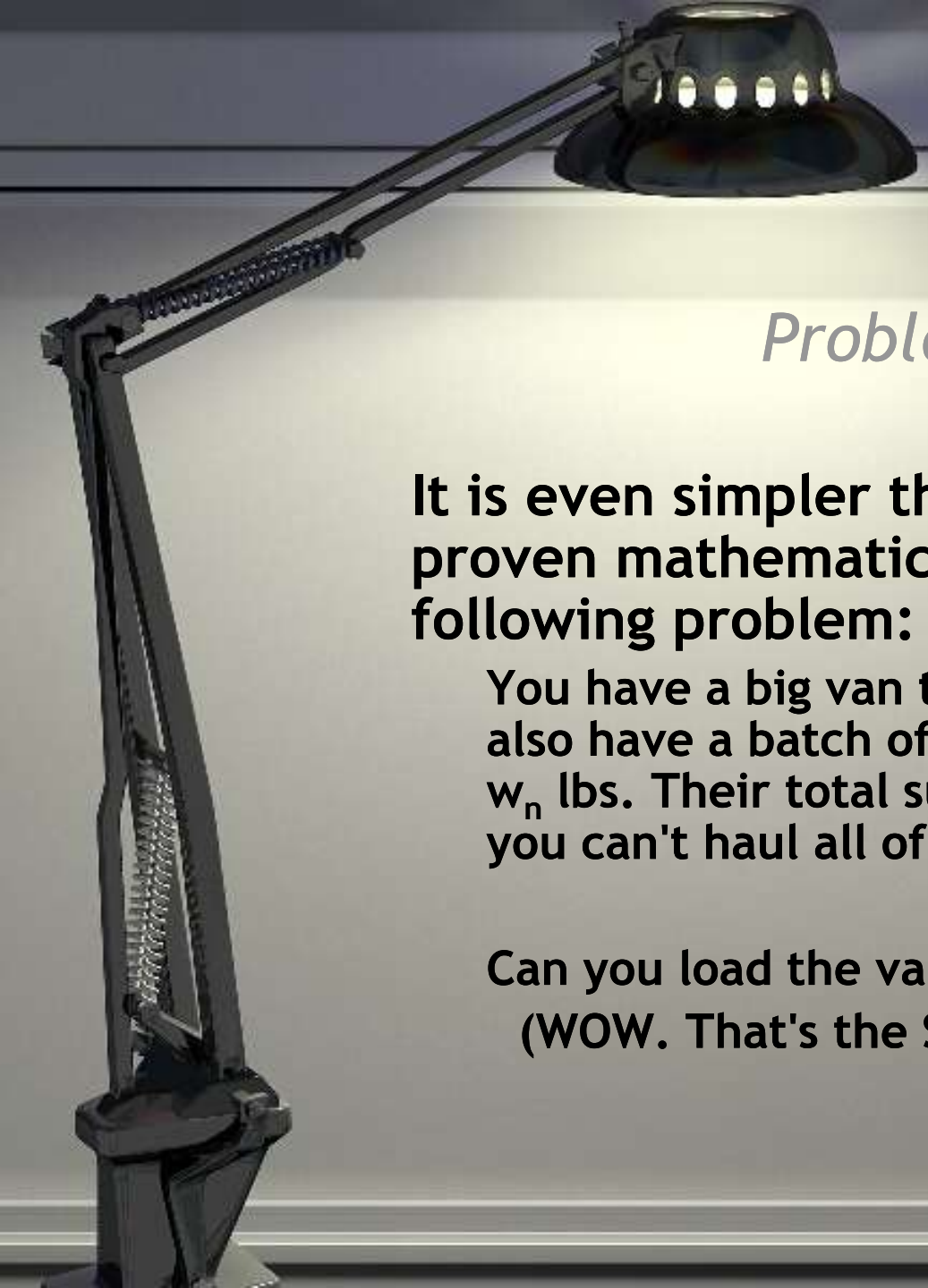


Problem Classes

This actually defines our first Class, NP, the set of decision problems whose Yes instances can be solved by a Nondeterministic Turing Machine in polynomial time.

That knowledge is not of much use!! We still don't know how to tell (easily) if a problem is in NP. And, that's our goal.

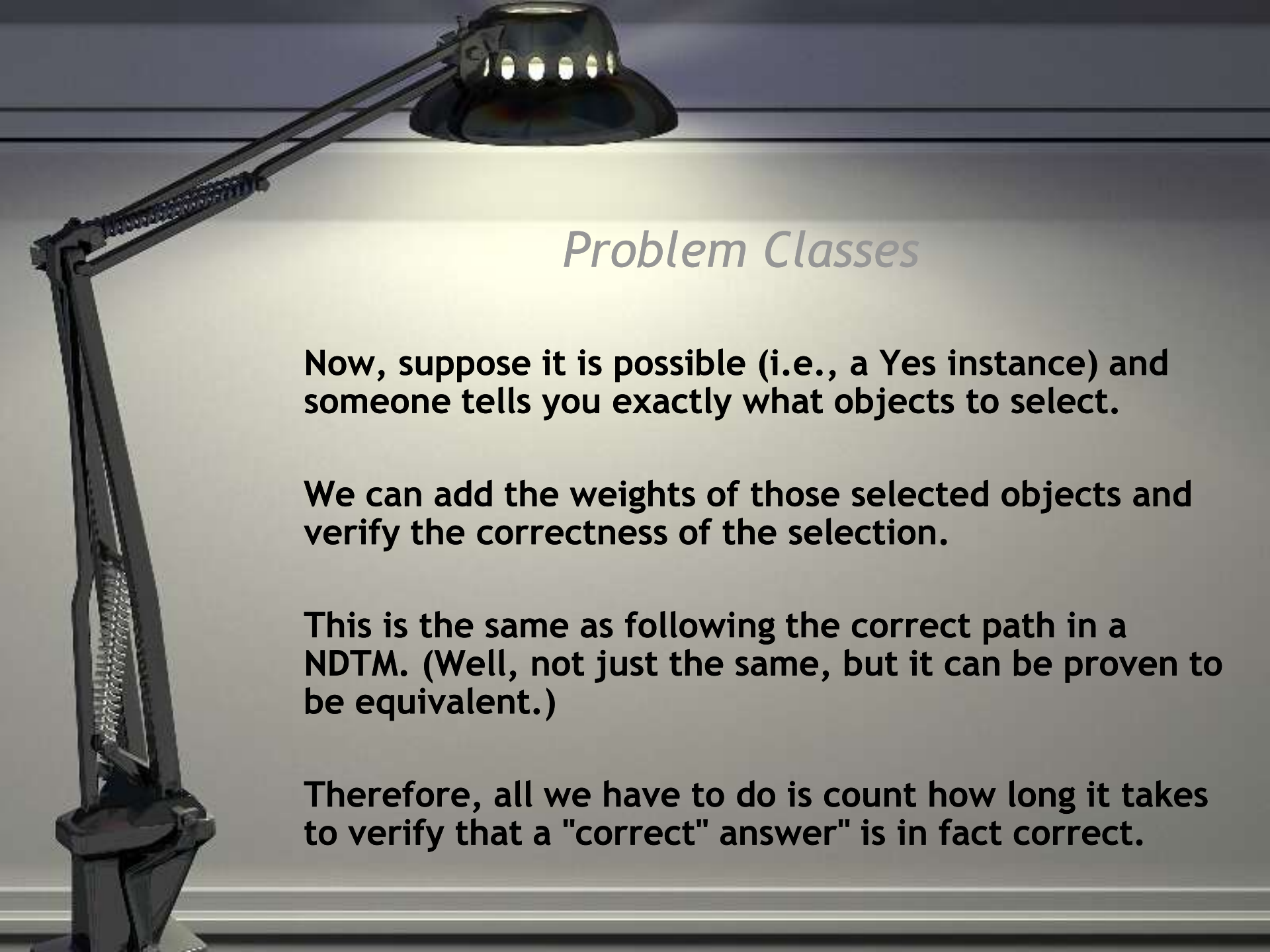
Fortunately, all we are doing with a NDTM is tracing the correct path to the accept state. Since all we are interested in doing is counting it's length, if someone just gave us the correct path and we followed it, we could learn the same thing - how long it is.



Problem Classes

It is even simpler than that (all this has been proven mathematically). Consider the following problem:

> **You have a big van that can carry 10,000 lbs. You also have a batch of objects with weights w_1, w_2, …, w_n lbs. Their total sum is more than 10,000 lbs, so you can't haul all of them.**
>
> **Can you load the van with exactly 10,000 lbs?**
>
> **(WOW. That's the SubsetSum problem.)**



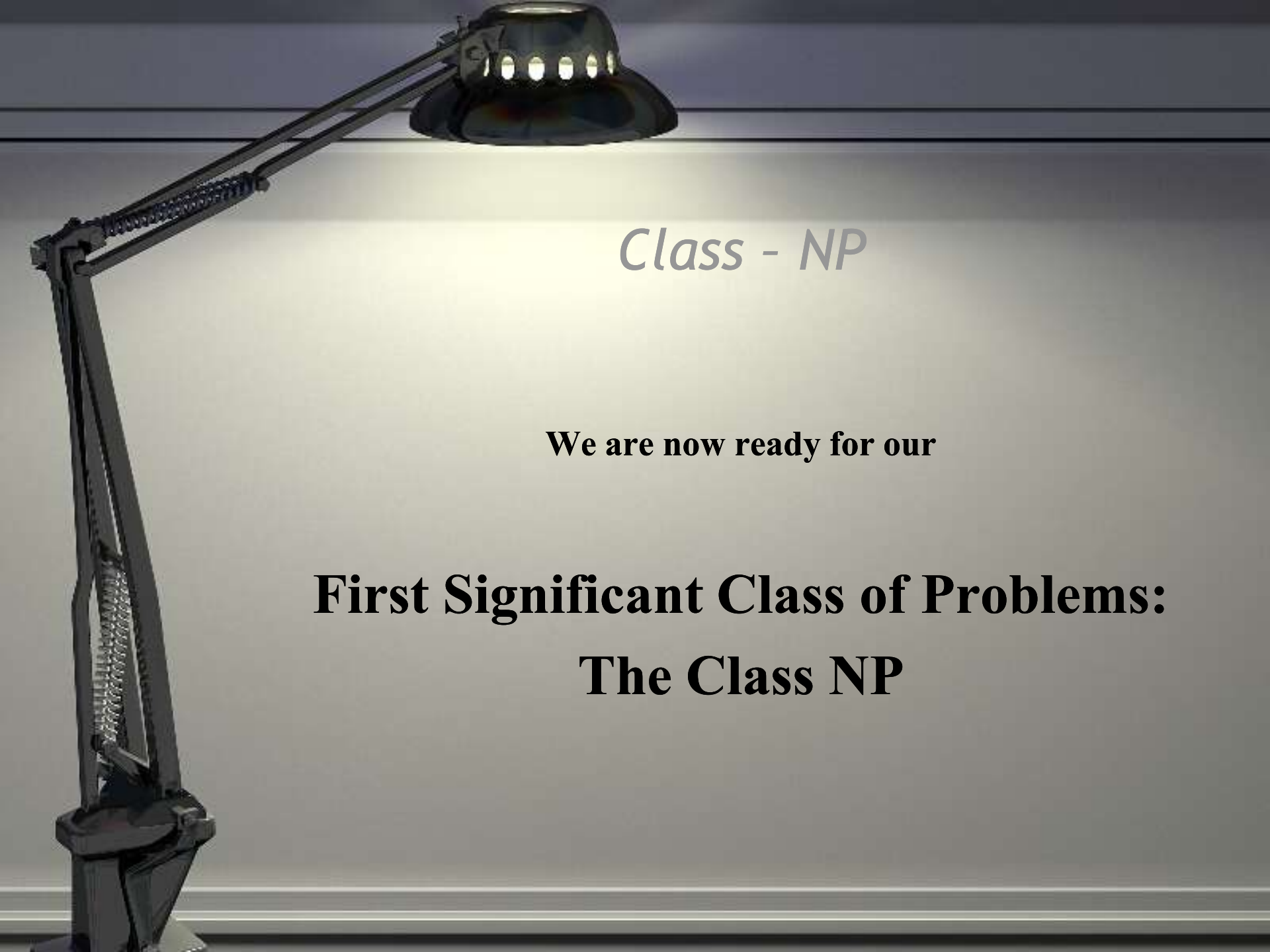
Problem Classes

Now, suppose it is possible (i.e., a Yes instance) and someone tells you exactly what objects to select.

We can add the weights of those selected objects and verify the correctness of the selection.

This is the same as following the correct path in a NDTM. (Well, not just the same, but it can be proven to be equivalent.)

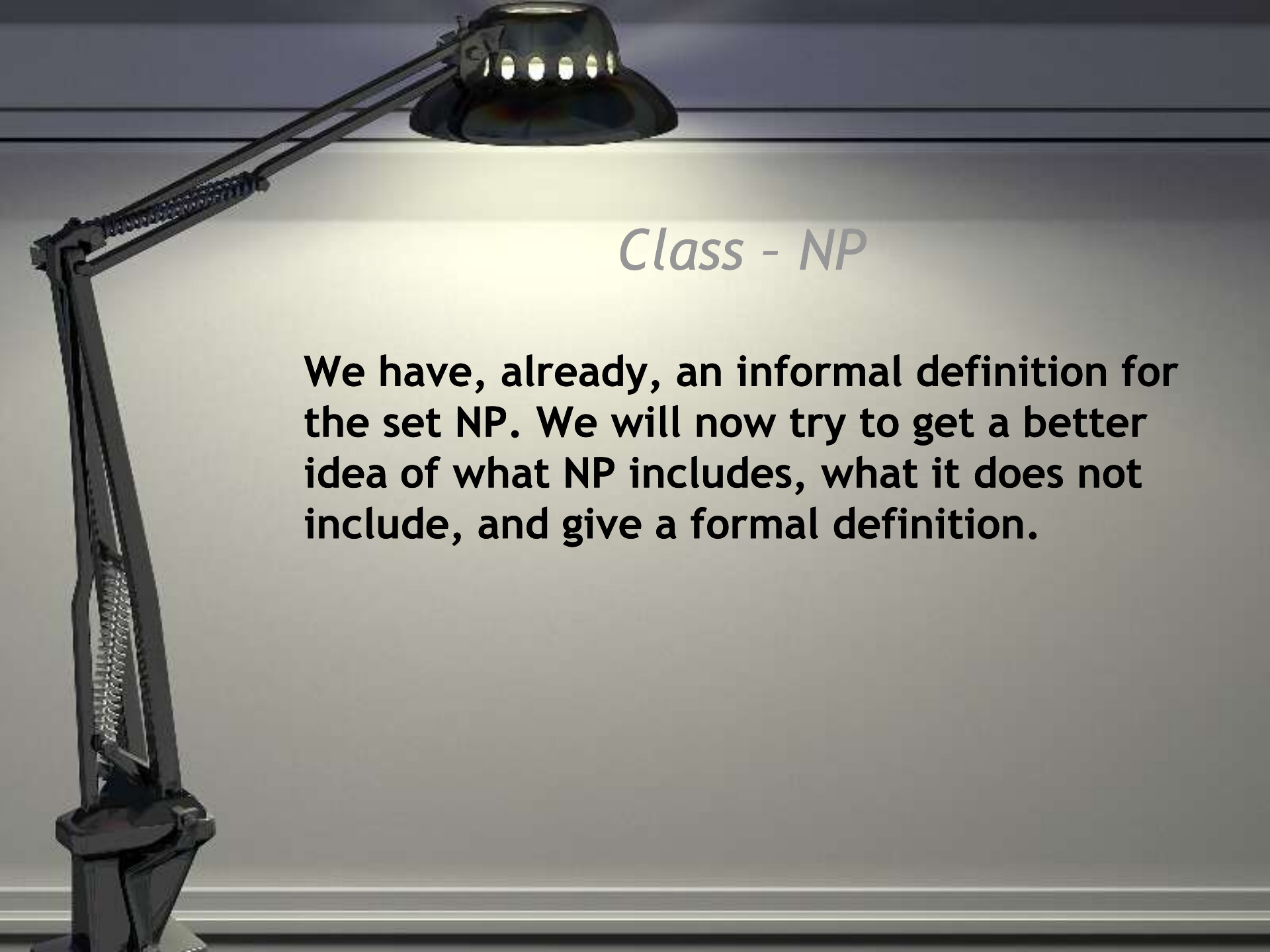
Therefore, all we have to do is count how long it takes to verify that a "correct" answer" is in fact correct.



Class – NP

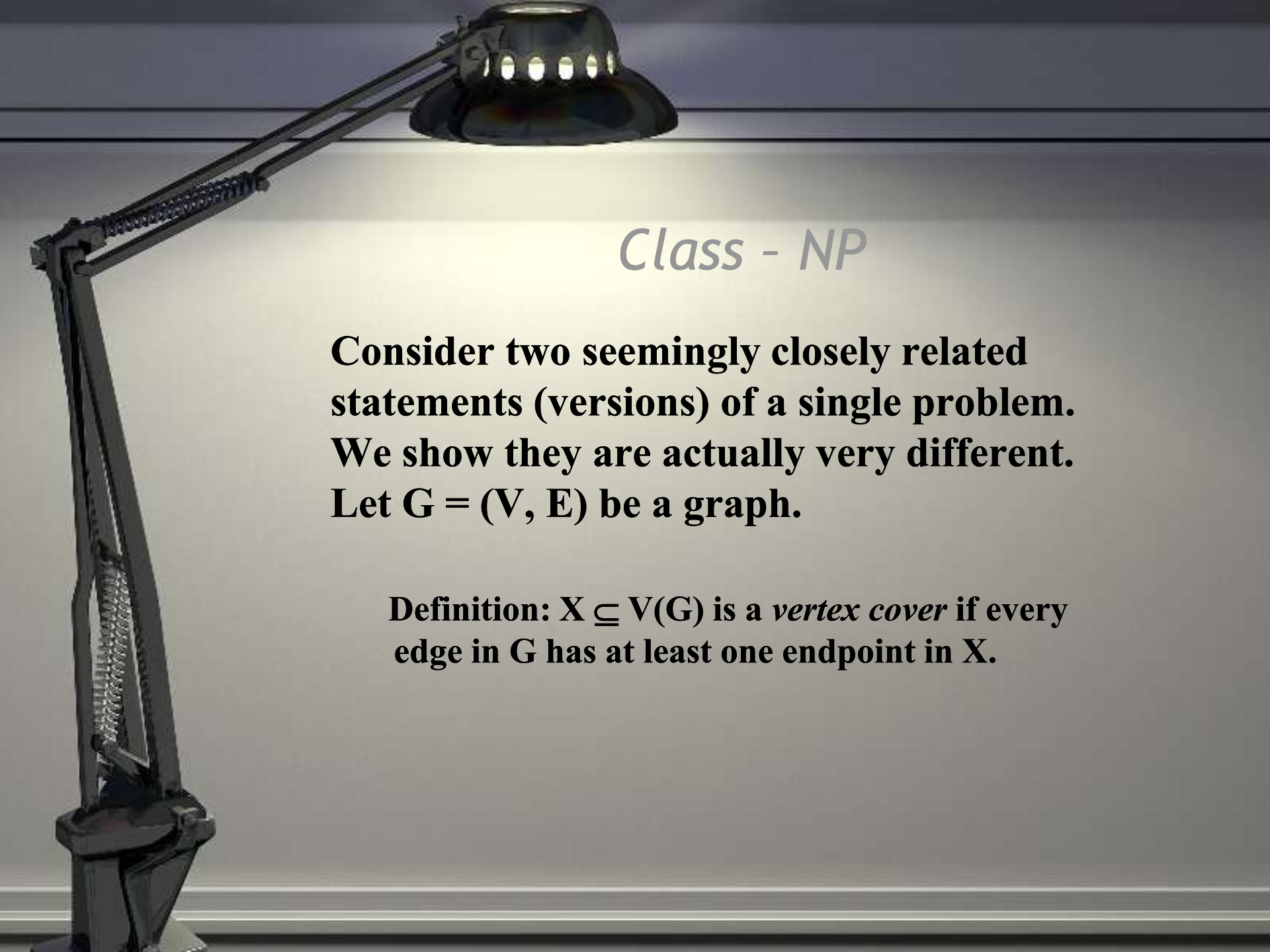
We are now ready for our

First Significant Class of Problems: The Class NP



Class – NP

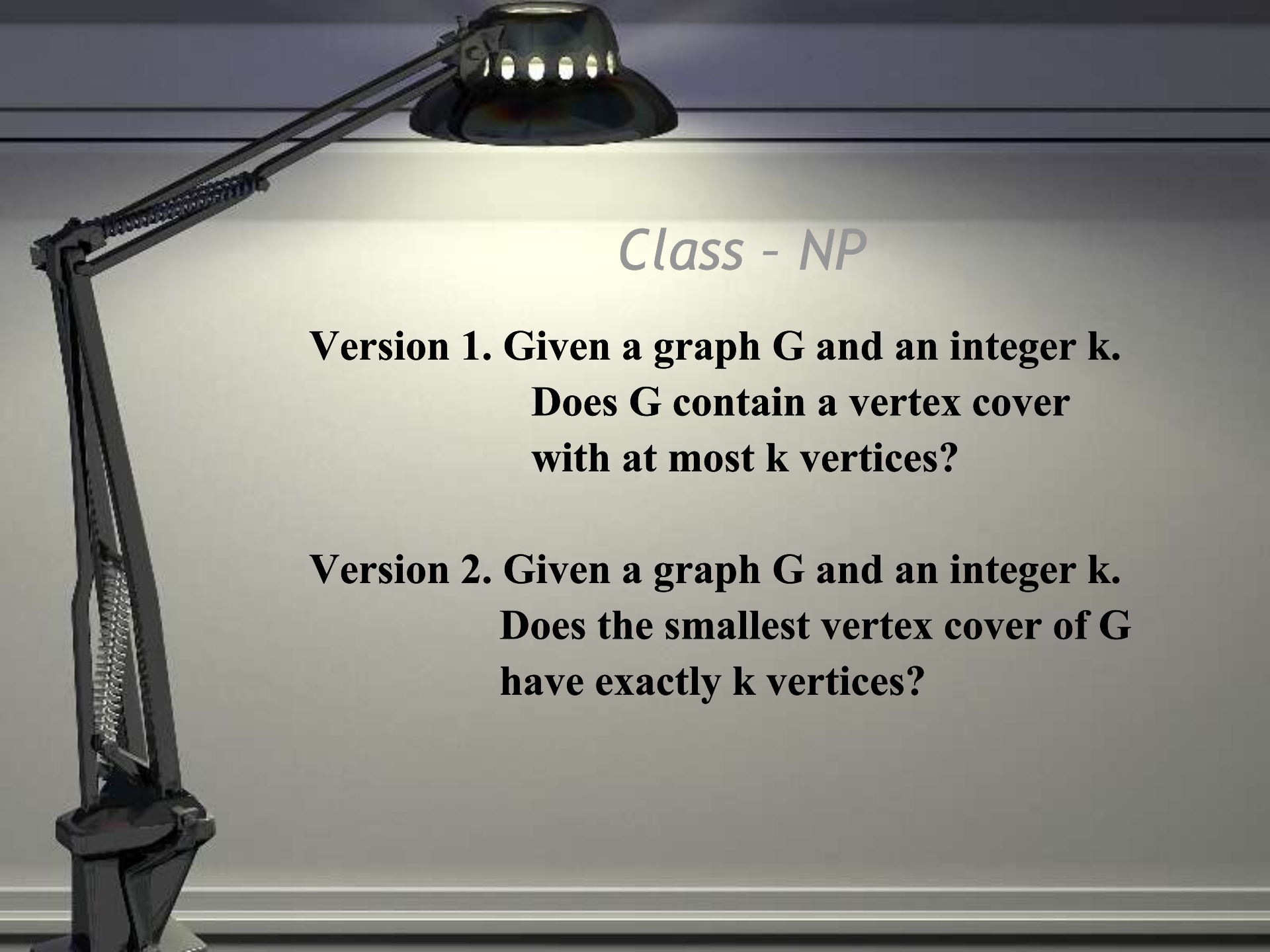
We have, already, an informal definition for the set NP. We will now try to get a better idea of what NP includes, what it does not include, and give a formal definition.



Class – NP

Consider two seemingly closely related statements (versions) of a single problem. We show they are actually very different. Let G = (V, E) be a graph.

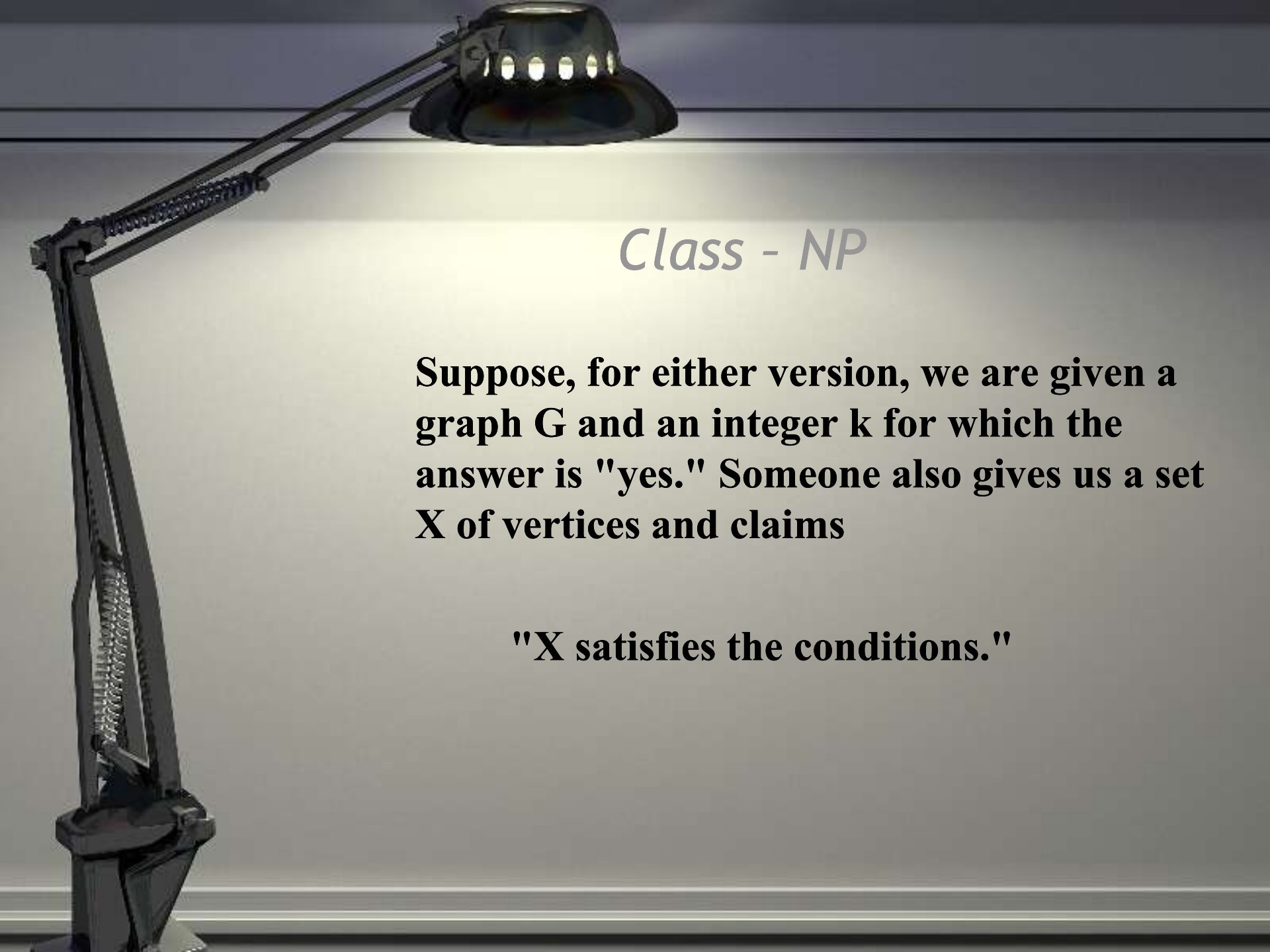
Definition: $X \subseteq V(G)$ is a *vertex cover* if every edge in G has at least one endpoint in X.



Class – NP

Version 1. Given a graph G and an integer k.
Does G contain a vertex cover
with at most k vertices?

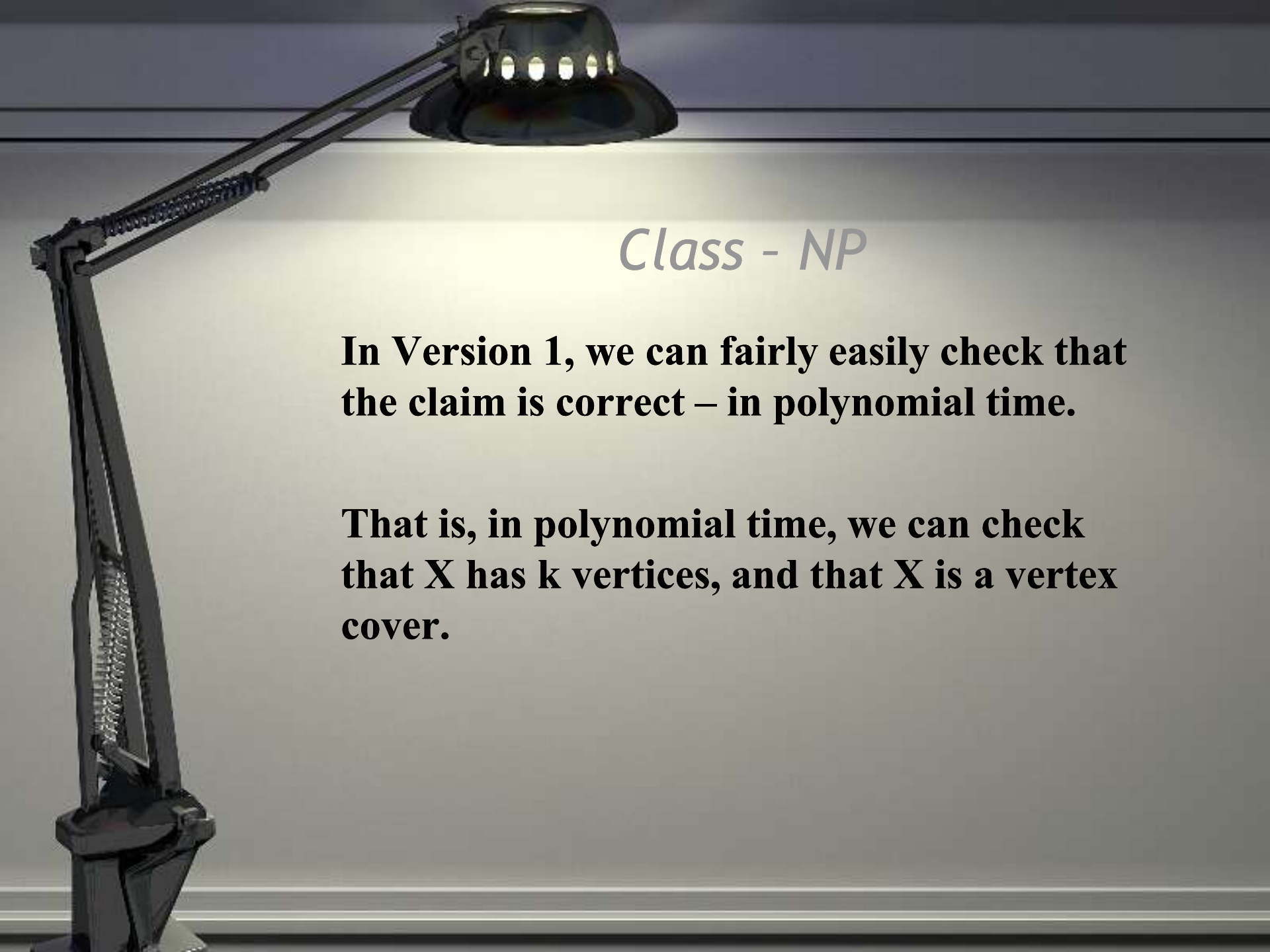
Version 2. Given a graph G and an integer k.
Does the smallest vertex cover of G
have exactly k vertices?



Class – NP

Suppose, for either version, we are given a graph G and an integer k for which the answer is "yes." Someone also gives us a set X of vertices and claims

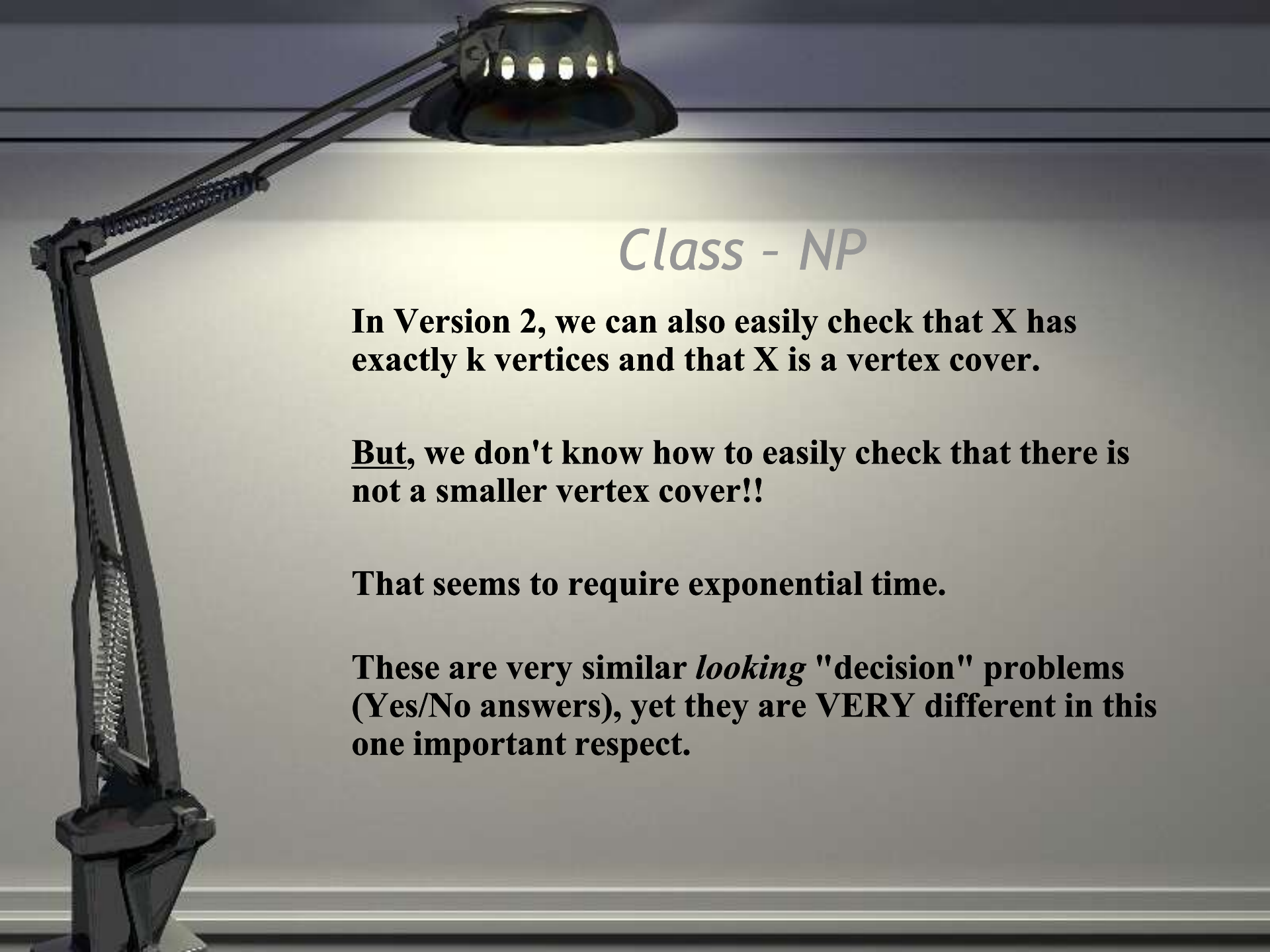
"X satisfies the conditions."



Class – NP

In Version 1, we can fairly easily check that the claim is correct – in polynomial time.

That is, in polynomial time, we can check that X has k vertices, and that X is a vertex cover.



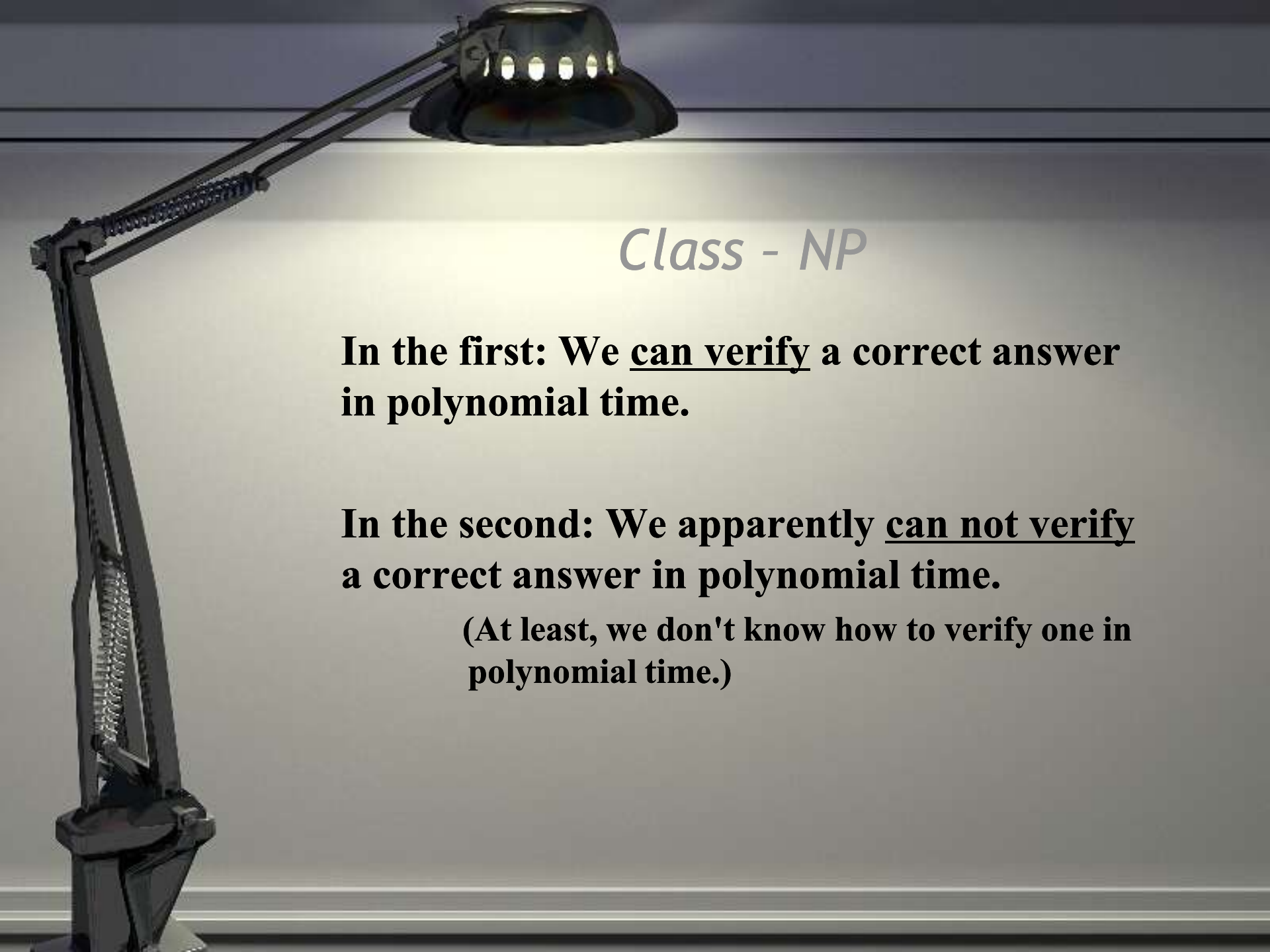
Class – NP

In Version 2, we can also easily check that X has exactly k vertices and that X is a vertex cover.

<u>But</u>, we don't know how to easily check that there is not a smaller vertex cover!!

That seems to require exponential time.

These are very similar *looking* "decision" problems (Yes/No answers), yet they are VERY different in this one important respect.

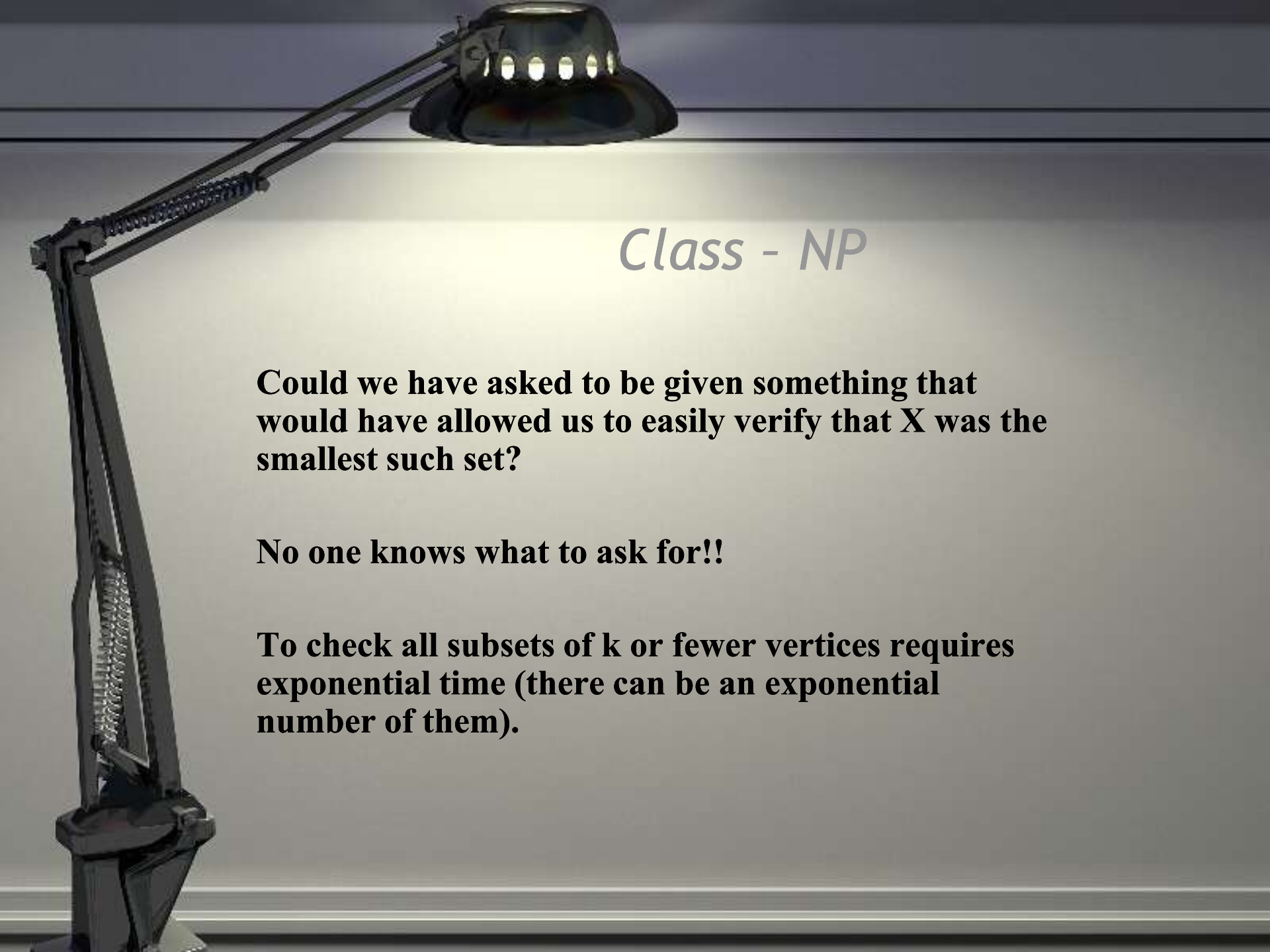


Class – NP

In the first: We <u>can verify</u> a correct answer in polynomial time.

In the second: We apparently <u>can not verify</u> a correct answer in polynomial time.

(At least, we don't know how to verify one in polynomial time.)

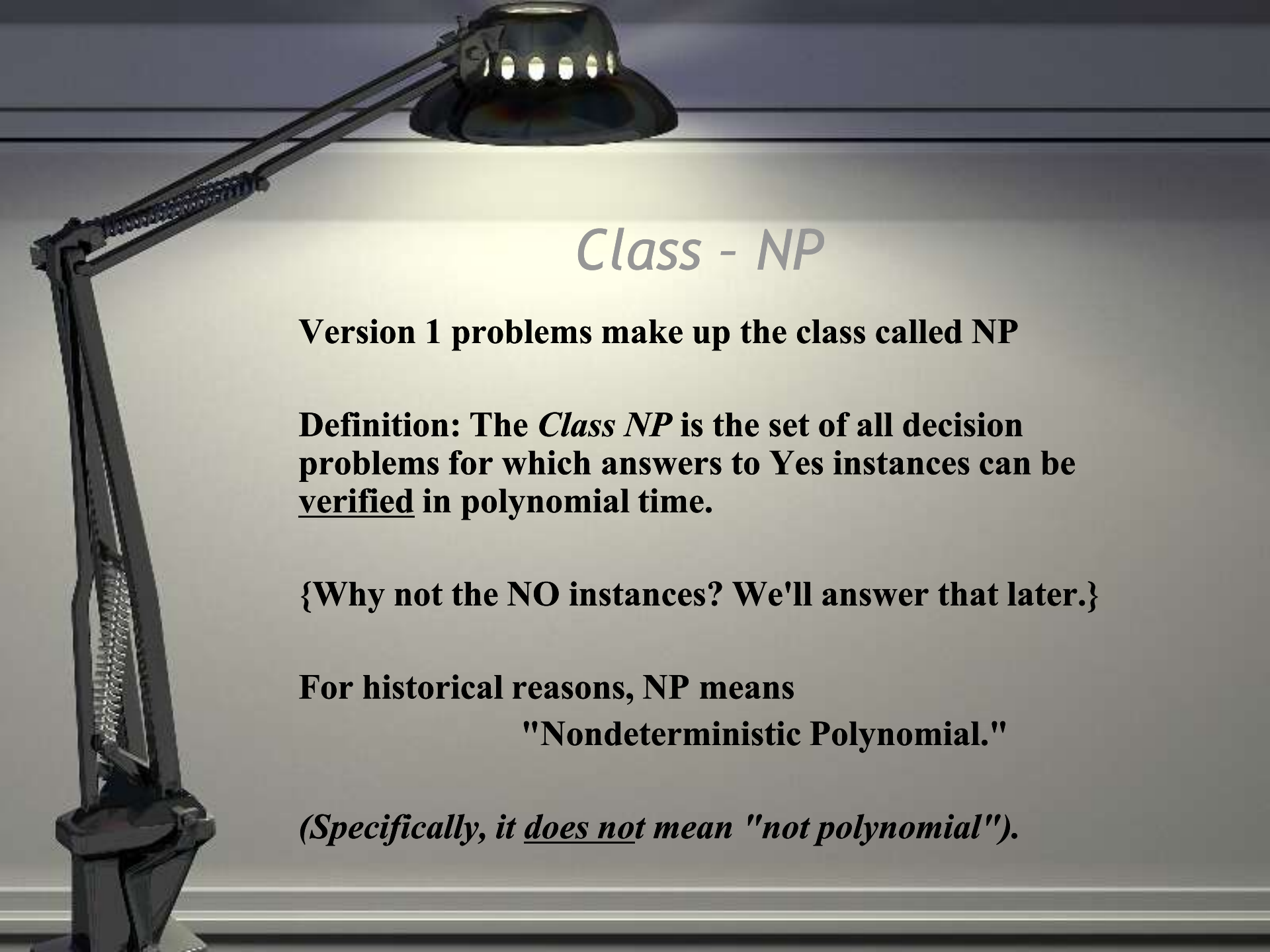


Class – NP

Could we have asked to be given something that would have allowed us to easily verify that X was the smallest such set?

No one knows what to ask for!!

To check all subsets of k or fewer vertices requires exponential time (there can be an exponential number of them).



Class – NP

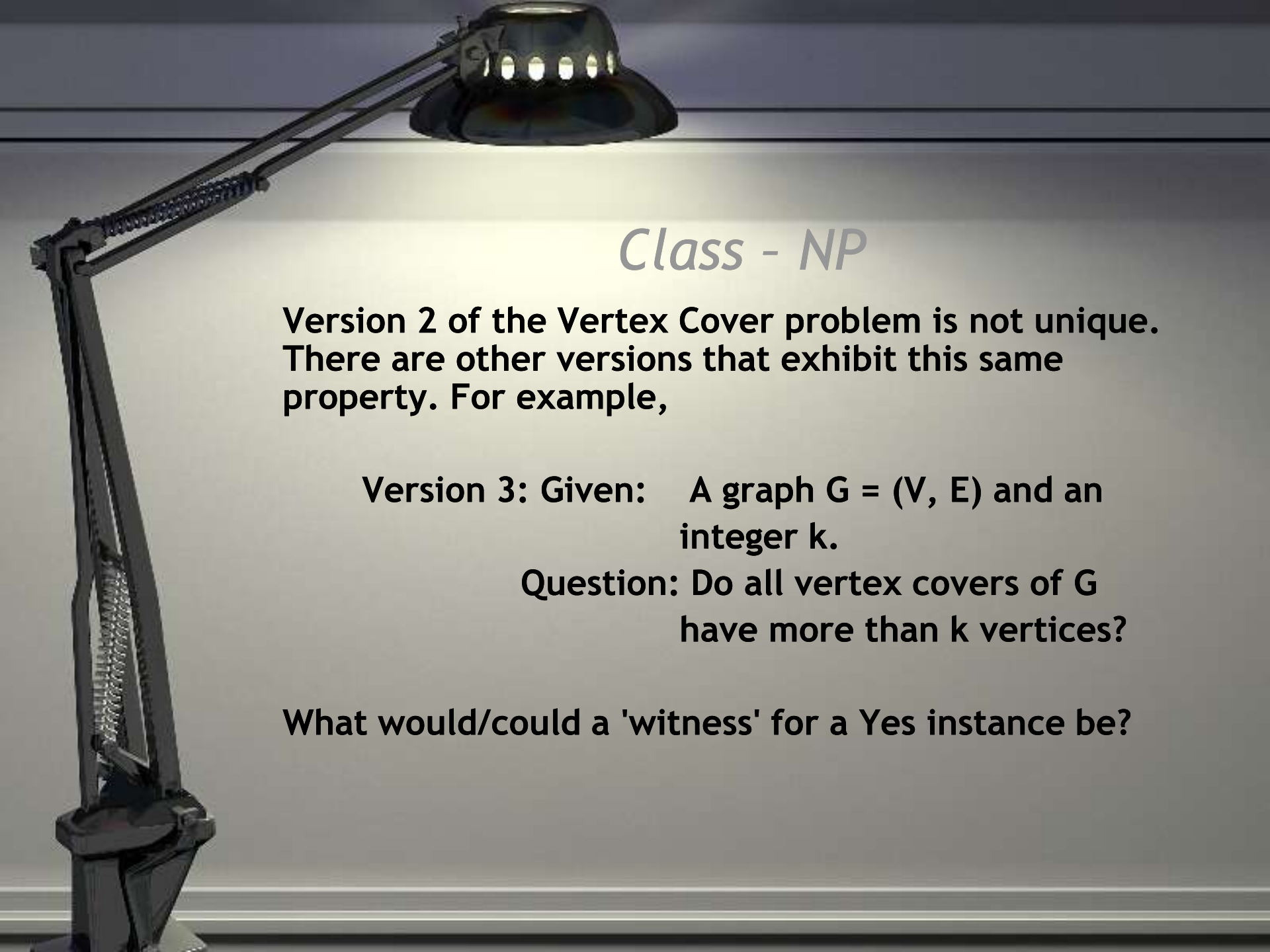
Version 1 problems make up the class called NP

Definition: The *Class NP* is the set of all decision problems for which answers to Yes instances can be verified in polynomial time.

{Why not the NO instances? We'll answer that later.}

For historical reasons, NP means
"Nondeterministic Polynomial."

(Specifically, it does not mean "not polynomial").



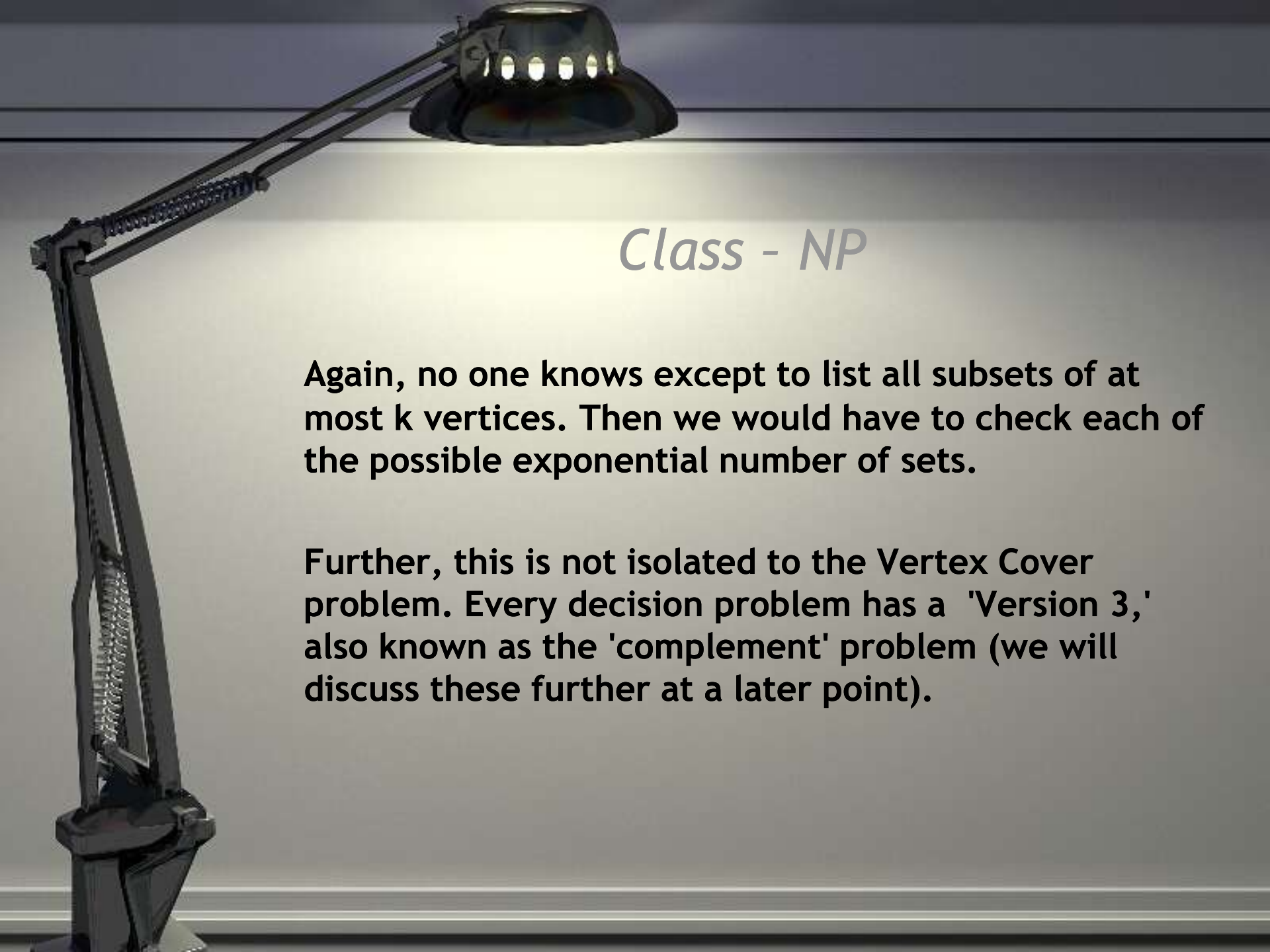
Class – NP

Version 2 of the Vertex Cover problem is not unique. There are other versions that exhibit this same property. For example,

Version 3: Given: A graph G = (V, E) and an integer k.

Question: Do all vertex covers of G have more than k vertices?

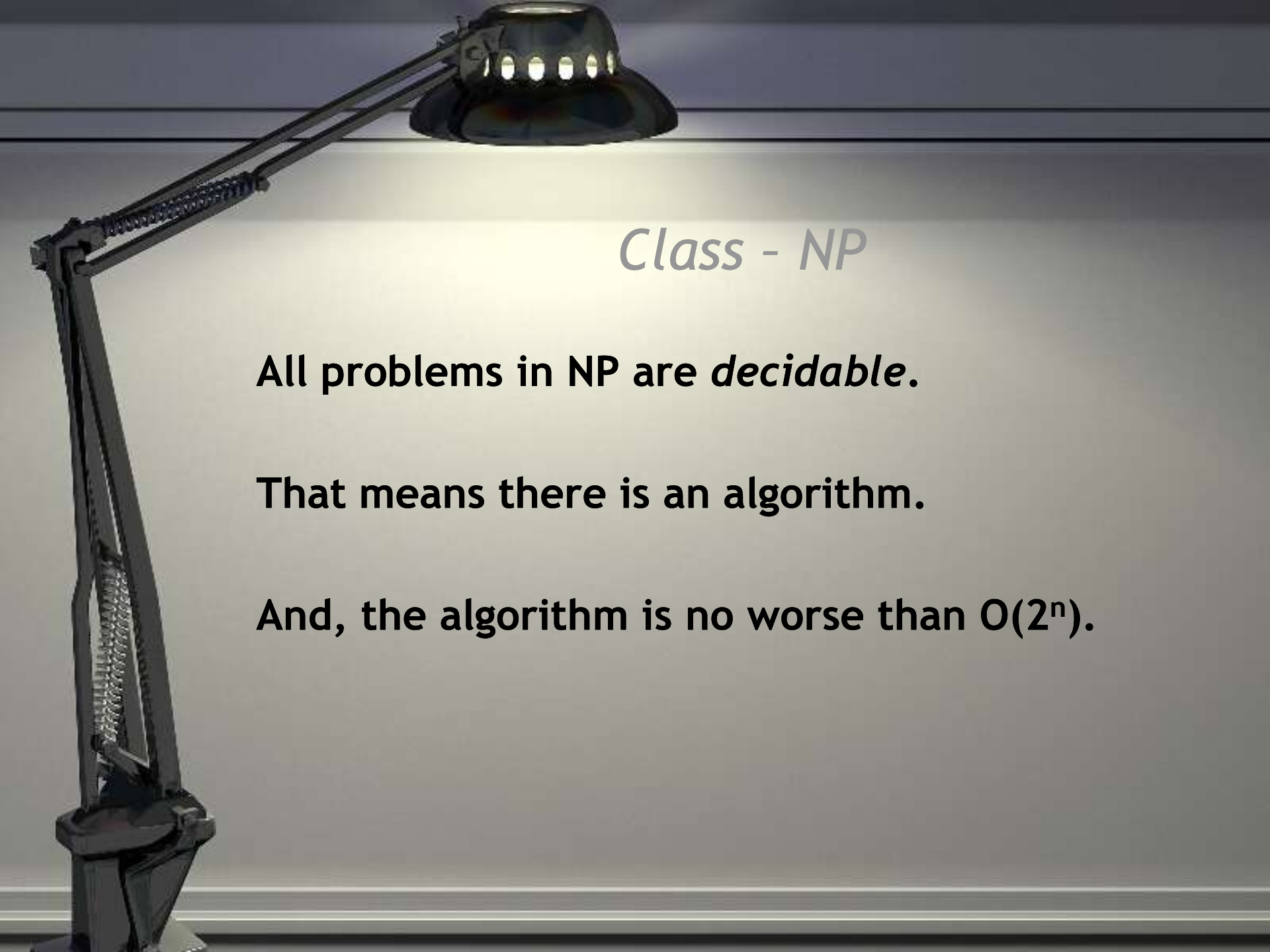
What would/could a 'witness' for a Yes instance be?



Class – NP

Again, no one knows except to list all subsets of at most k vertices. Then we would have to check each of the possible exponential number of sets.

Further, this is not isolated to the Vertex Cover problem. Every decision problem has a 'Version 3,' also known as the 'complement' problem (we will discuss these further at a later point).

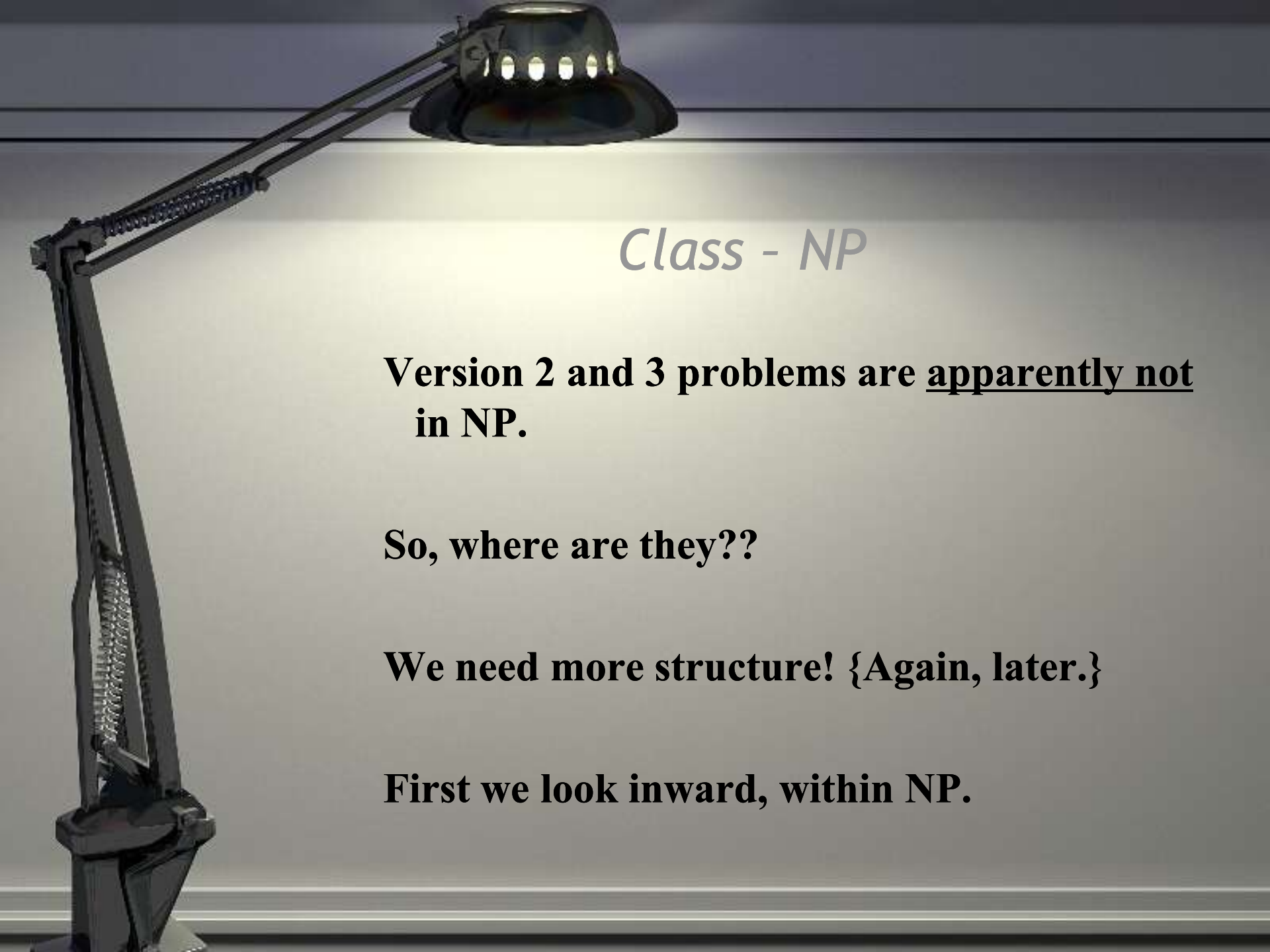


Class – NP

All problems in NP are *decidable*.

That means there is an algorithm.

And, the algorithm is no worse than $O(2^n)$.



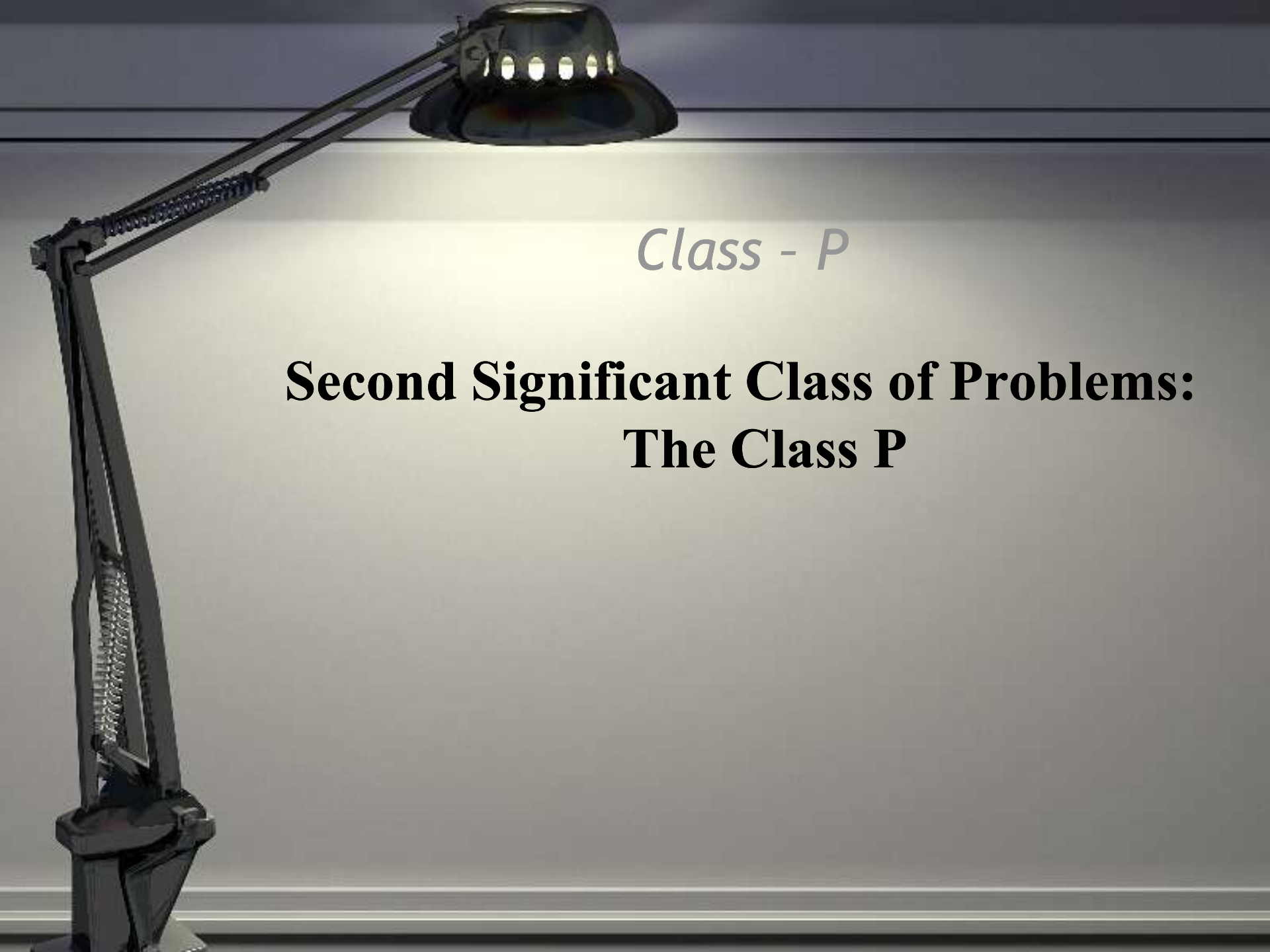
Class – NP

Version 2 and 3 problems are <u>apparently not</u> in NP.

So, where are they??

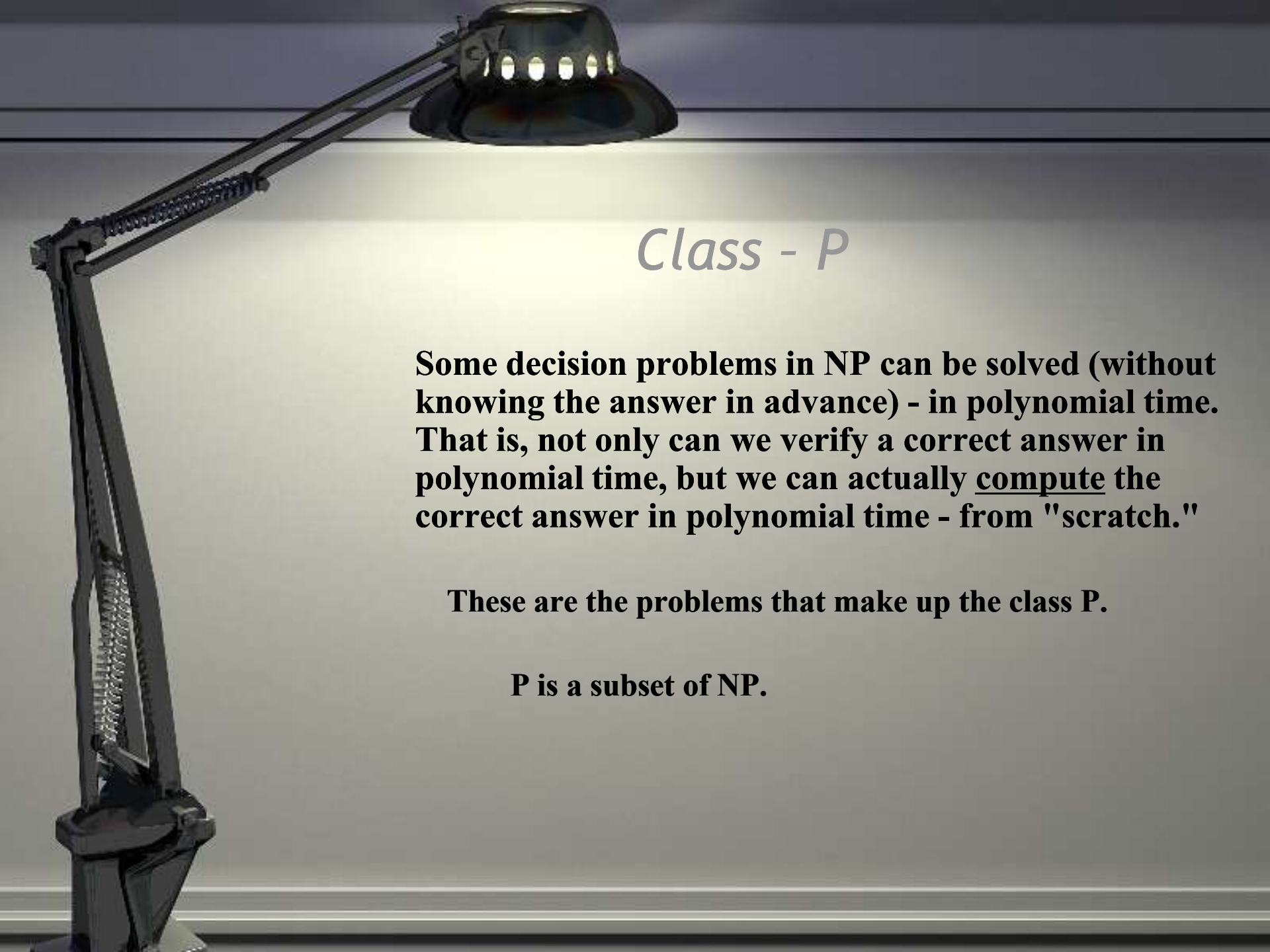
We need more structure! {Again, later.}

First we look inward, within NP.



Class - P

Second Significant Class of Problems: The Class P

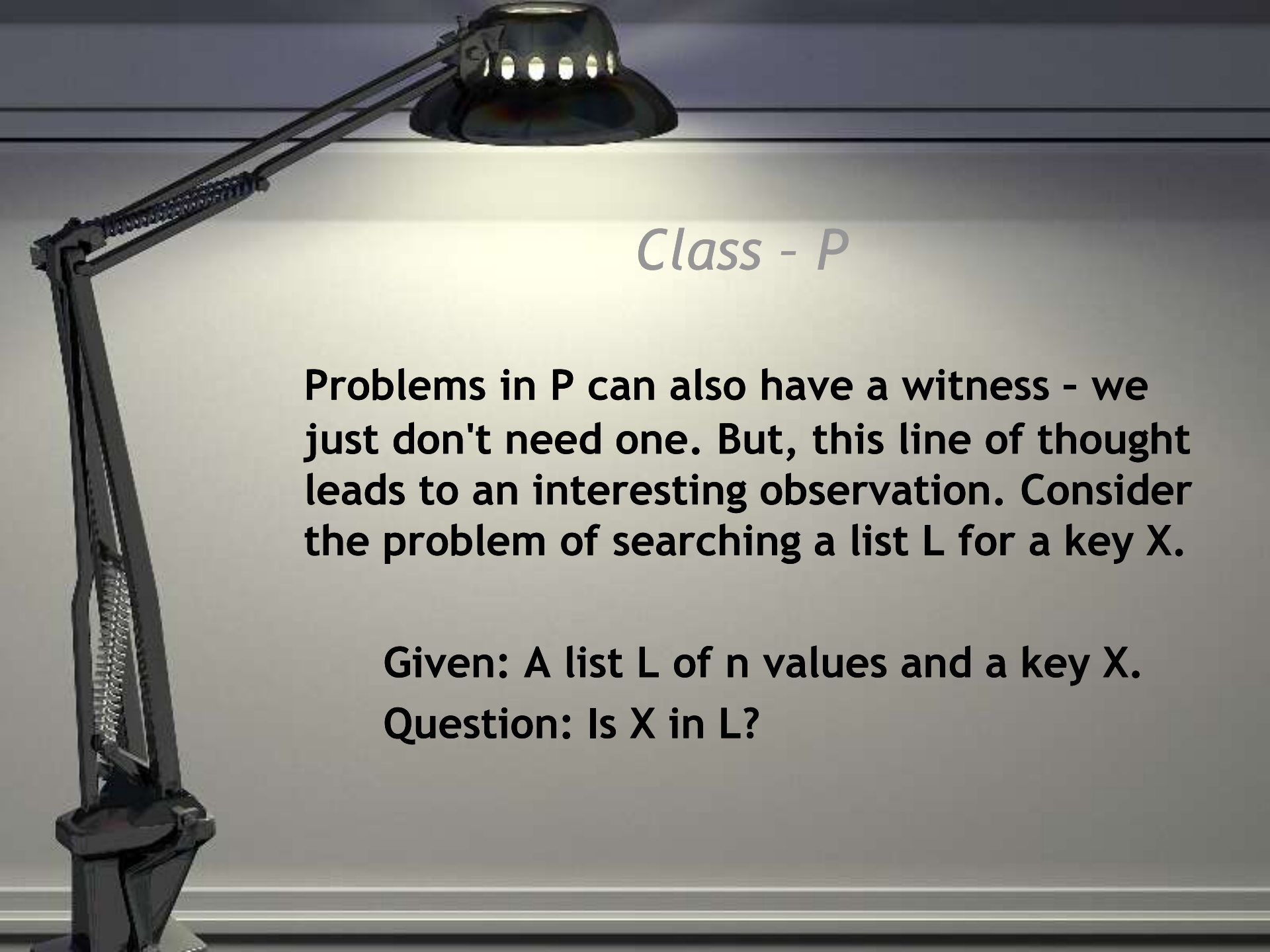


Class – P

Some decision problems in NP can be solved (without knowing the answer in advance) - in polynomial time. That is, not only can we verify a correct answer in polynomial time, but we can actually compute the correct answer in polynomial time - from "scratch."

These are the problems that make up the class P.

P is a subset of NP.

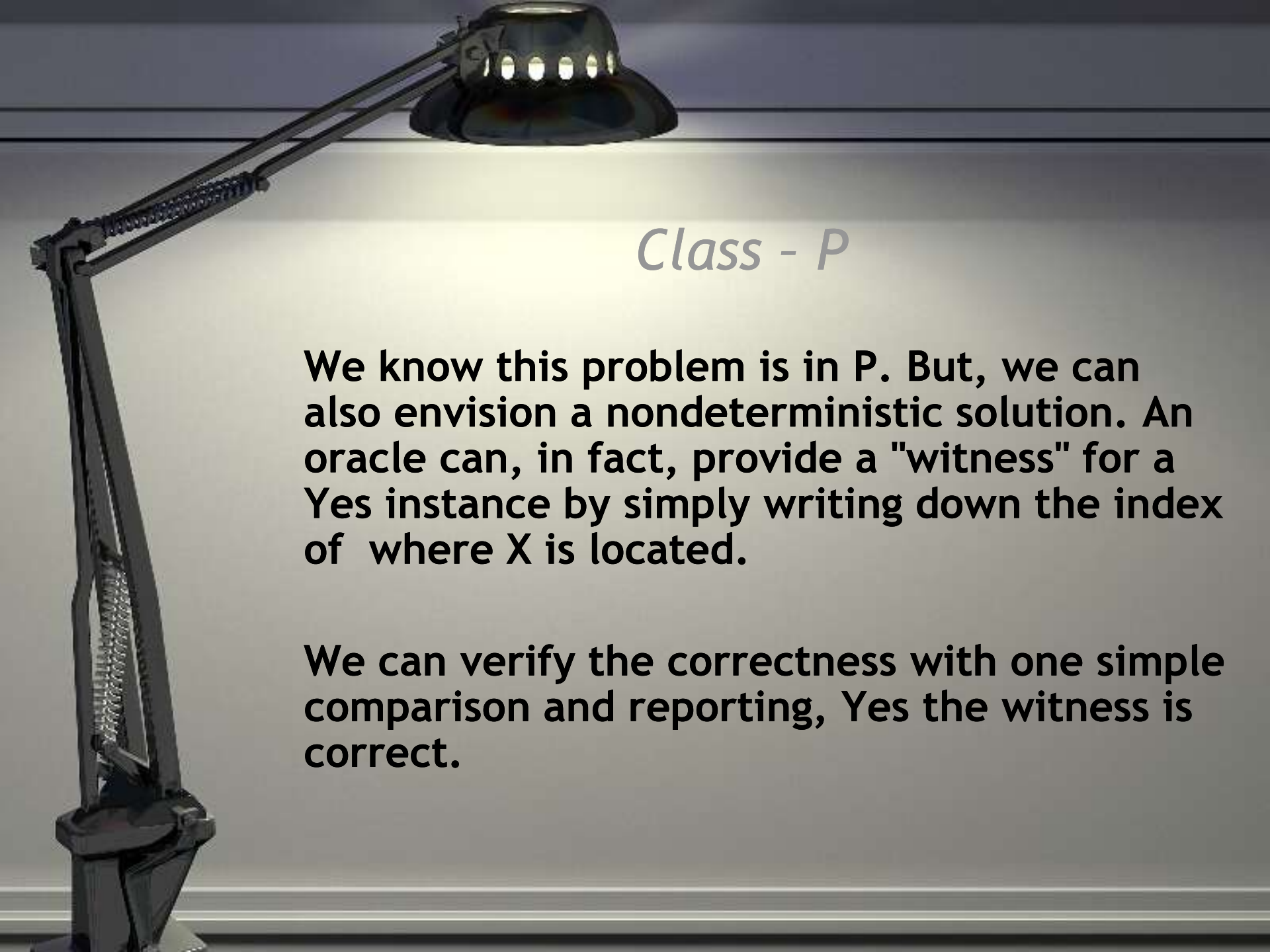


Class – P

Problems in P can also have a witness – we just don't need one. But, this line of thought leads to an interesting observation. Consider the problem of searching a list L for a key X.

Given: A list L of n values and a key X.

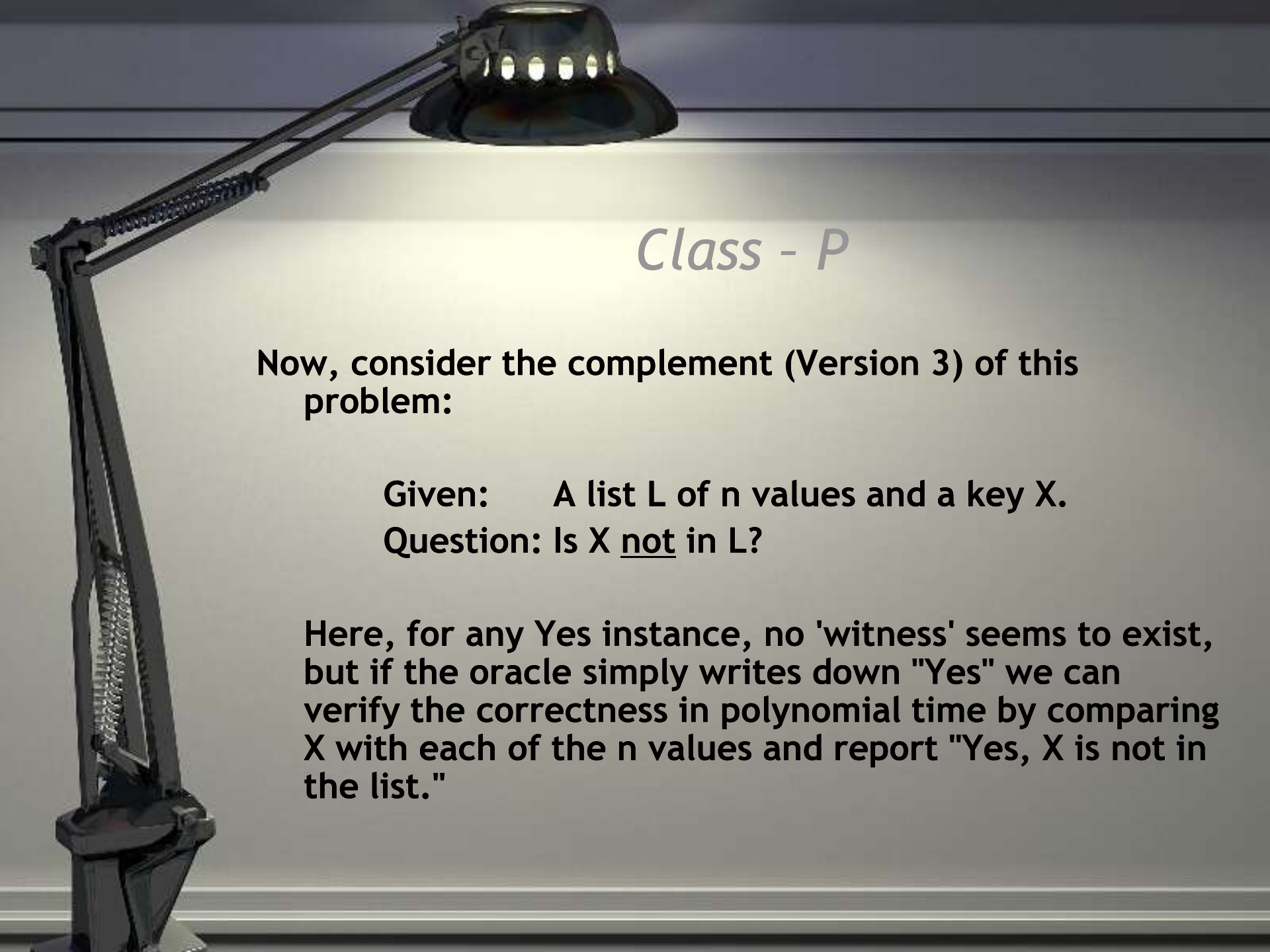
Question: Is X in L?



Class – P

We know this problem is in P. But, we can also envision a nondeterministic solution. An oracle can, in fact, provide a "witness" for a Yes instance by simply writing down the index of where X is located.

We can verify the correctness with one simple comparison and reporting, Yes the witness is correct.



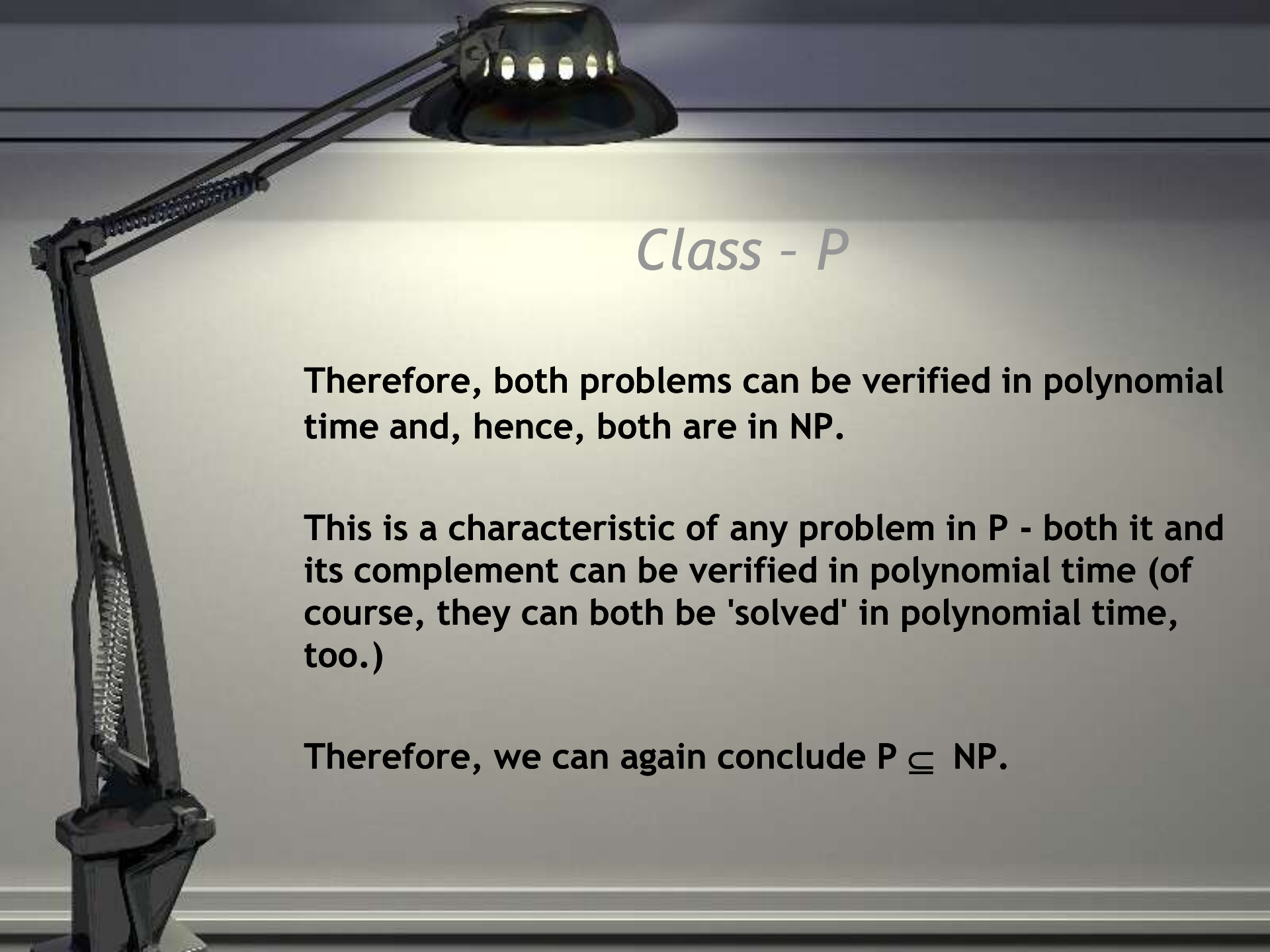
Class – P

Now, consider the complement (Version 3) of this problem:

Given: A list L of n values and a key X.

Question: Is X not in L?

Here, for any Yes instance, no 'witness' seems to exist, but if the oracle simply writes down "Yes" we can verify the correctness in polynomial time by comparing X with each of the n values and report "Yes, X is not in the list."

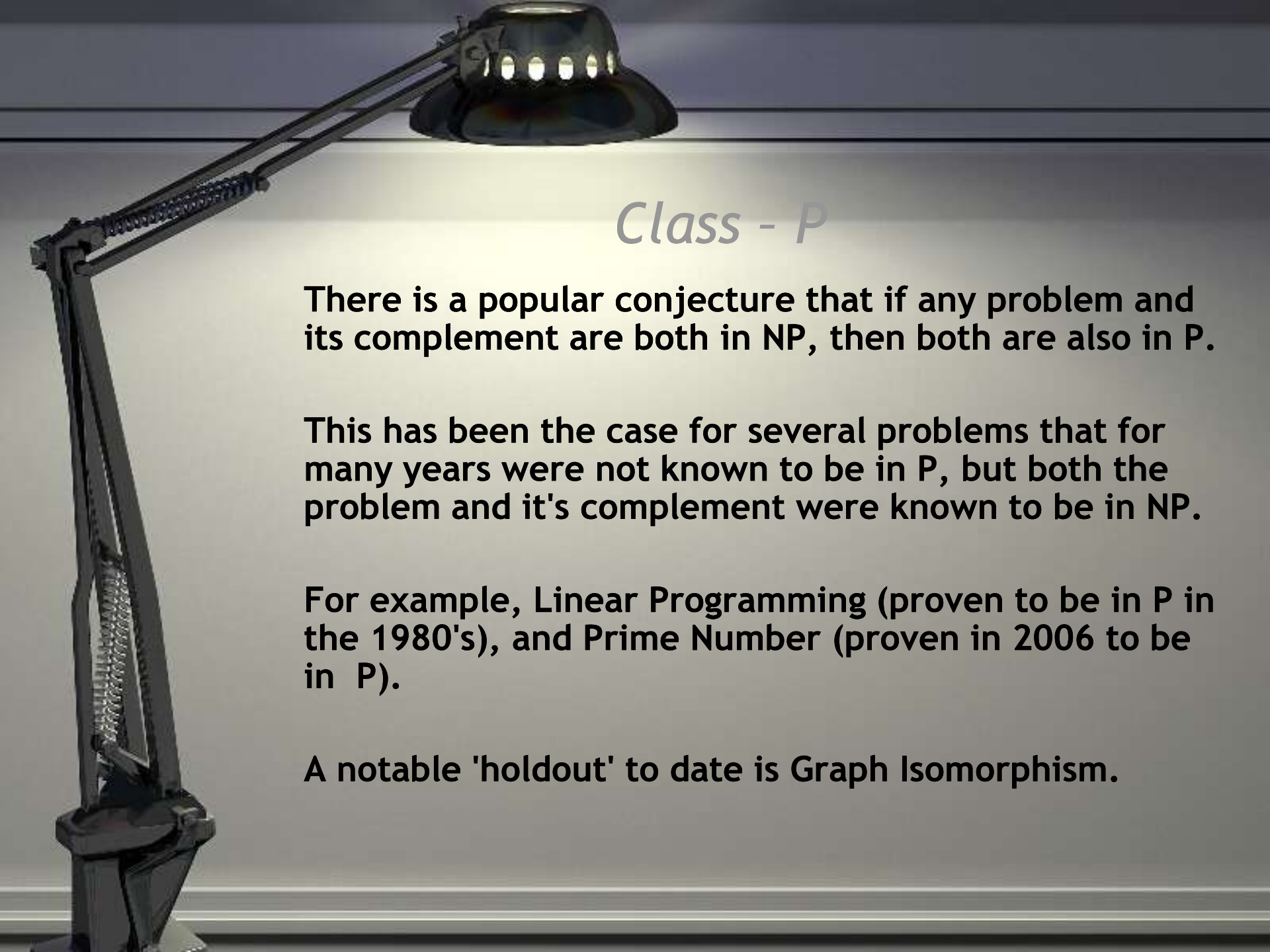


Class – P

Therefore, both problems can be verified in polynomial time and, hence, both are in NP.

This is a characteristic of any problem in P - both it and its complement can be verified in polynomial time (of course, they can both be 'solved' in polynomial time, too.)

Therefore, we can again conclude $P \subseteq NP$.



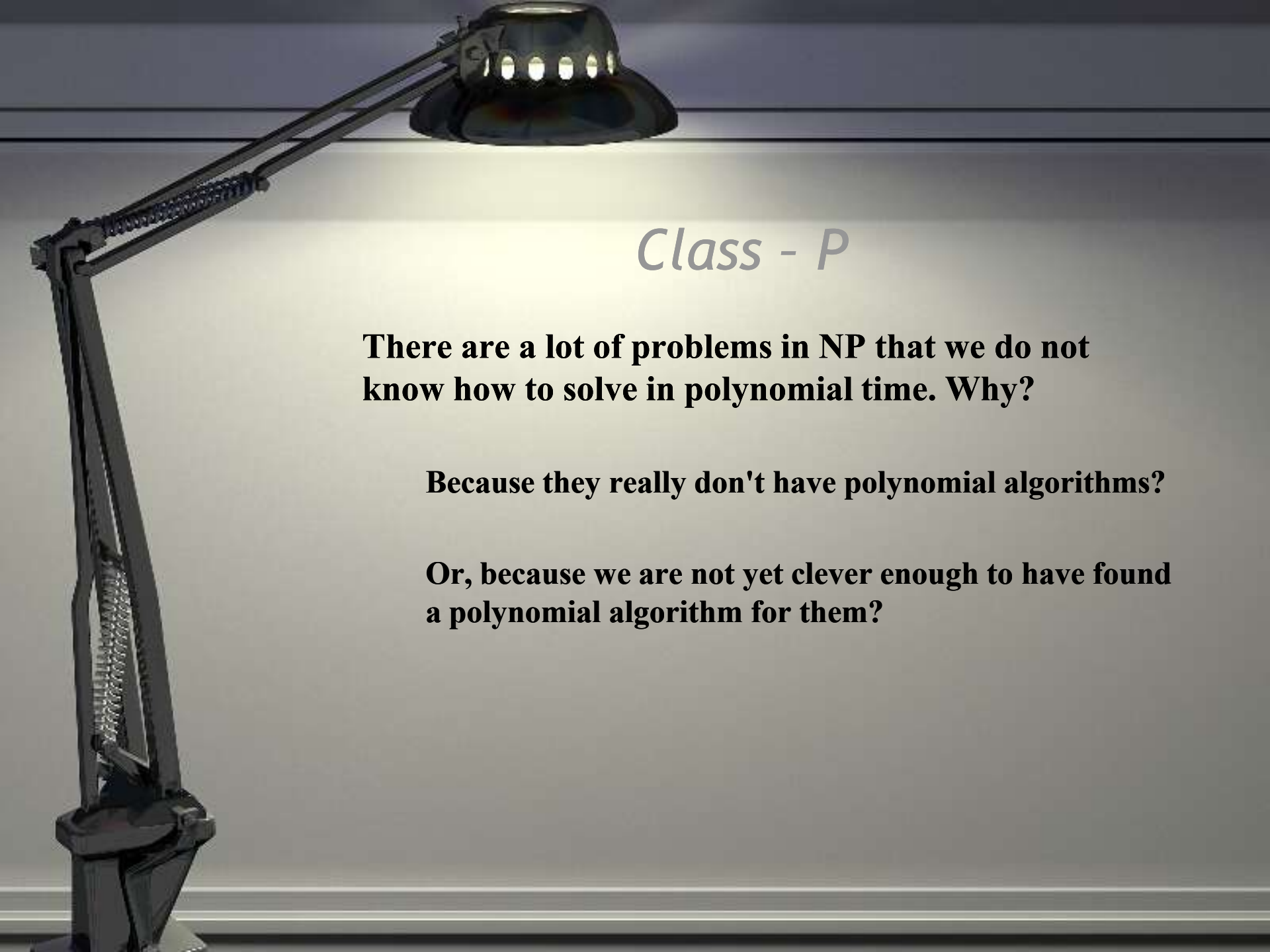
Class - P

There is a popular conjecture that if any problem and its complement are both in NP, then both are also in P.

This has been the case for several problems that for many years were not known to be in P, but both the problem and it's complement were known to be in NP.

For example, Linear Programming (proven to be in P in the 1980's), and Prime Number (proven in 2006 to be in P).

A notable 'holdout' to date is Graph Isomorphism.

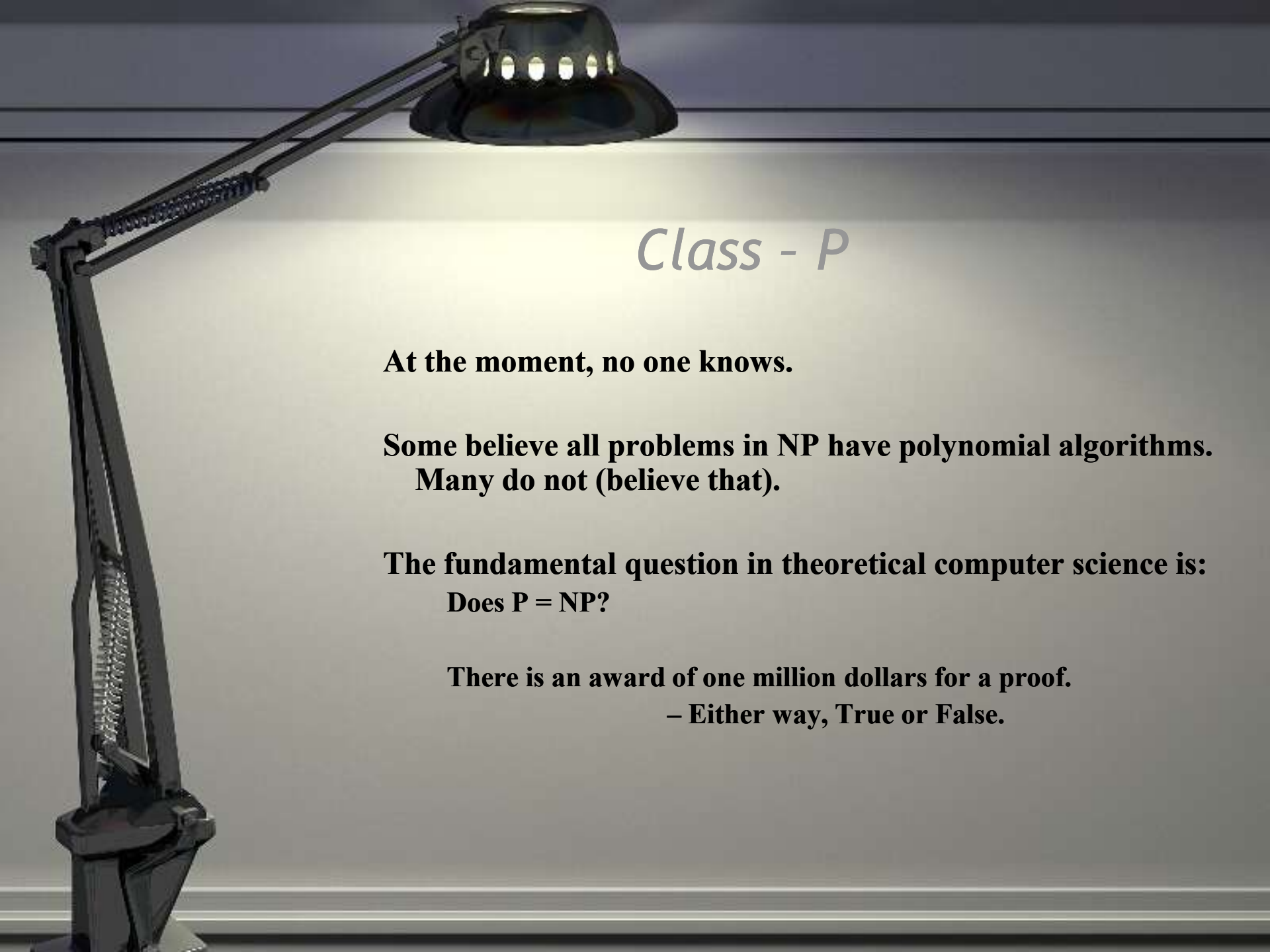


Class – P

There are a lot of problems in NP that we do not know how to solve in polynomial time. Why?

Because they really don't have polynomial algorithms?

Or, because we are not yet clever enough to have found a polynomial algorithm for them?



Class – P

At the moment, no one knows.

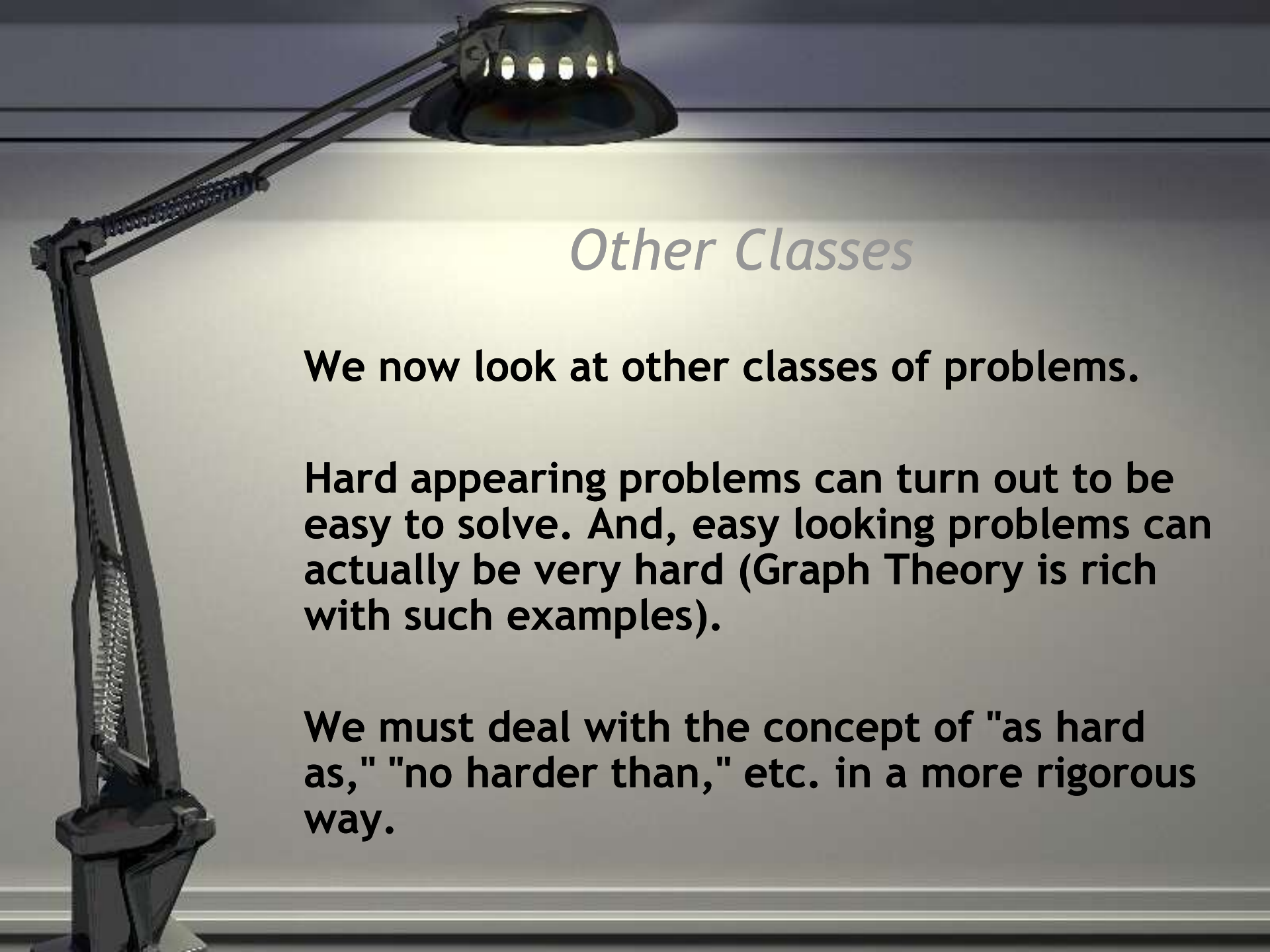
Some believe all problems in NP have polynomial algorithms.
Many do not (believe that).

The fundamental question in theoretical computer science is:

Does P = NP?

There is an award of one million dollars for a proof.

– Either way, True or False.

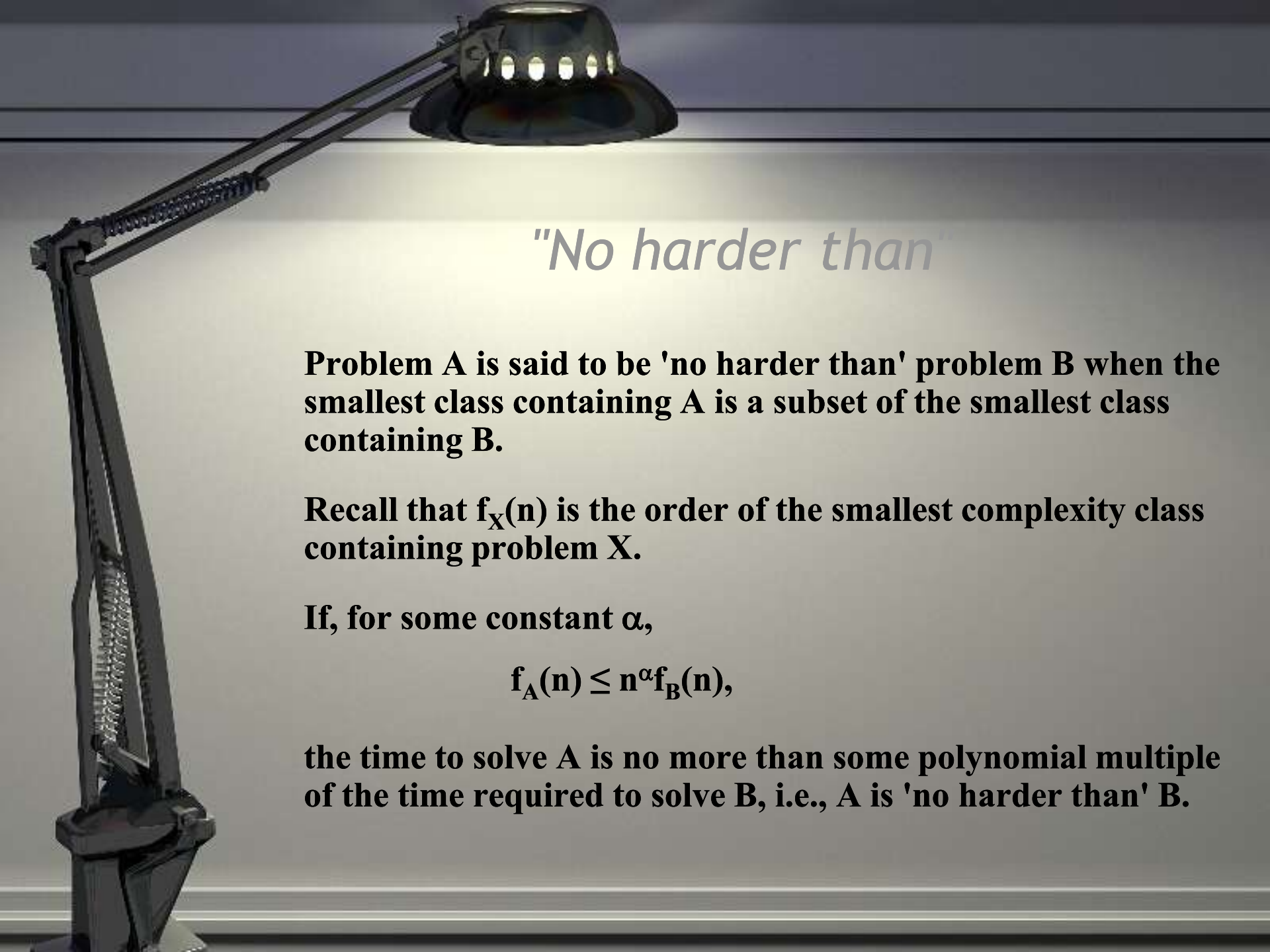


Other Classes

We now look at other classes of problems.

Hard appearing problems can turn out to be easy to solve. And, easy looking problems can actually be very hard (Graph Theory is rich with such examples).

We must deal with the concept of "as hard as," "no harder than," etc. in a more rigorous way.



"No harder than"

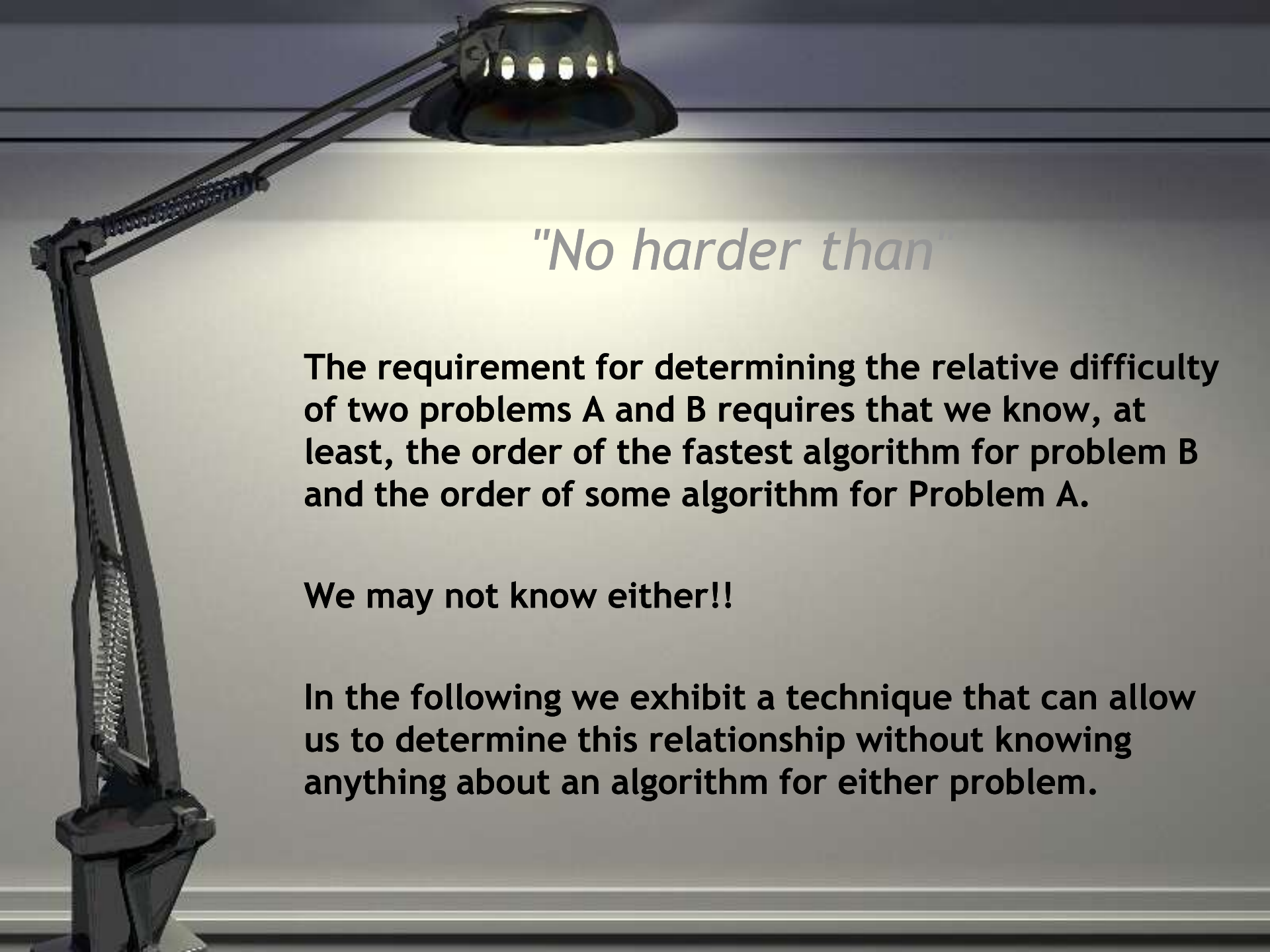
Problem A is said to be 'no harder than' problem B when the smallest class containing A is a subset of the smallest class containing B.

Recall that $f_X(n)$ is the order of the smallest complexity class containing problem X.

If, for some constant α,

$$f_A(n) \leq n^{\alpha} f_B(n),$$

the time to solve A is no more than some polynomial multiple of the time required to solve B, i.e., A is 'no harder than' B.

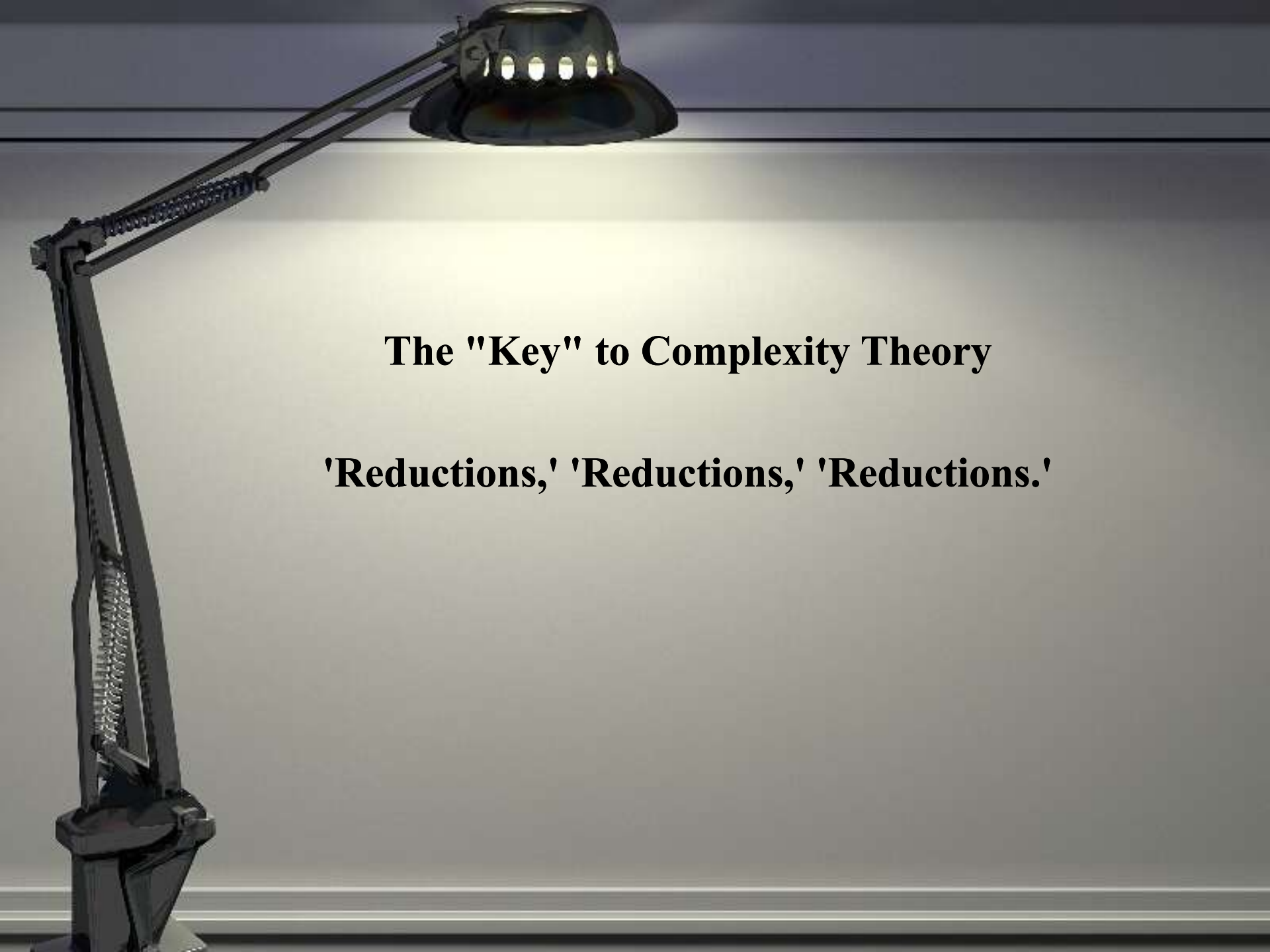


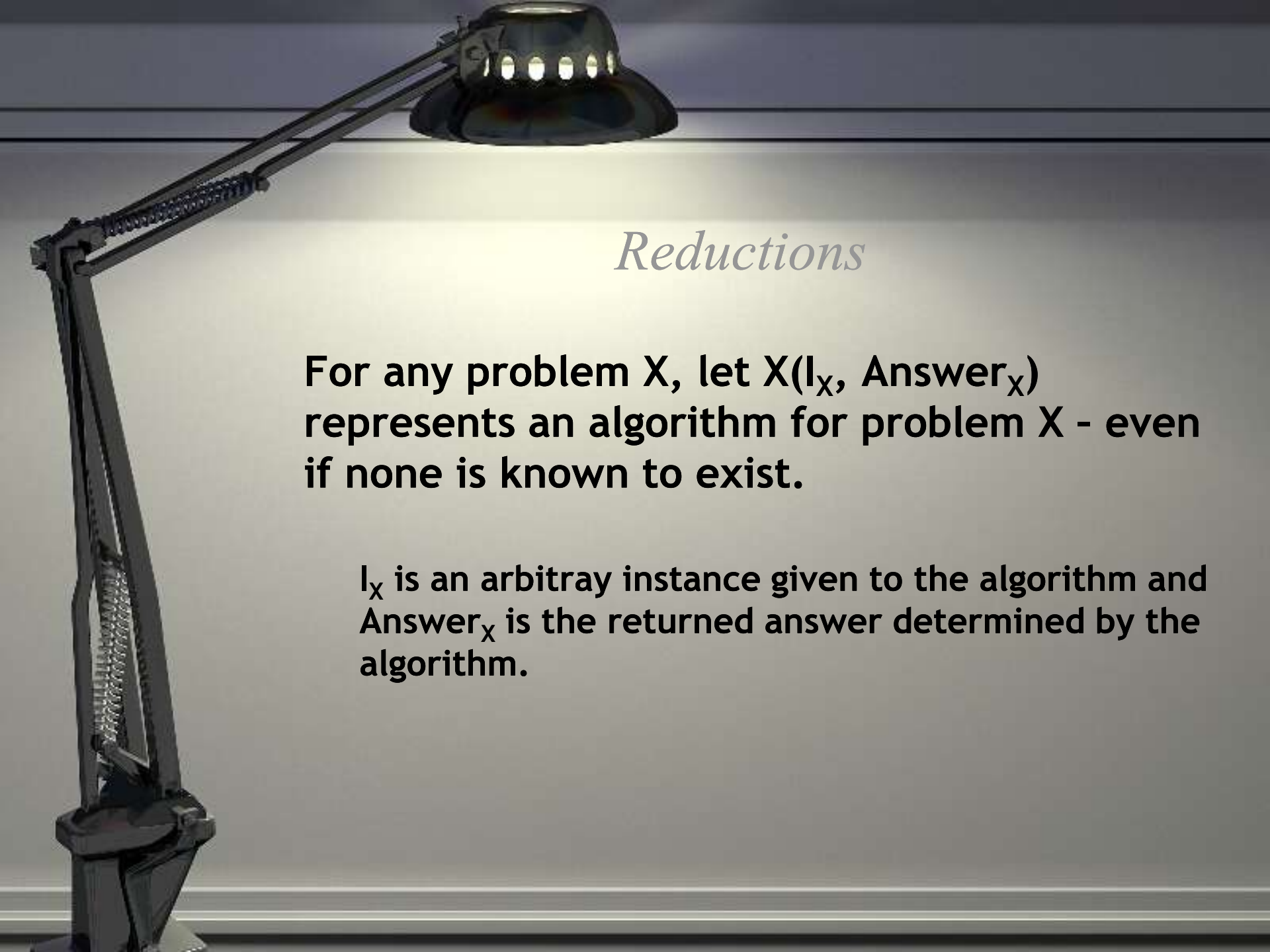
"No harder than"

The requirement for determining the relative difficulty of two problems A and B requires that we know, at least, the order of the fastest algorithm for problem B and the order of some algorithm for Problem A.

We may not know either!!

In the following we exhibit a technique that can allow us to determine this relationship without knowing anything about an algorithm for either problem.

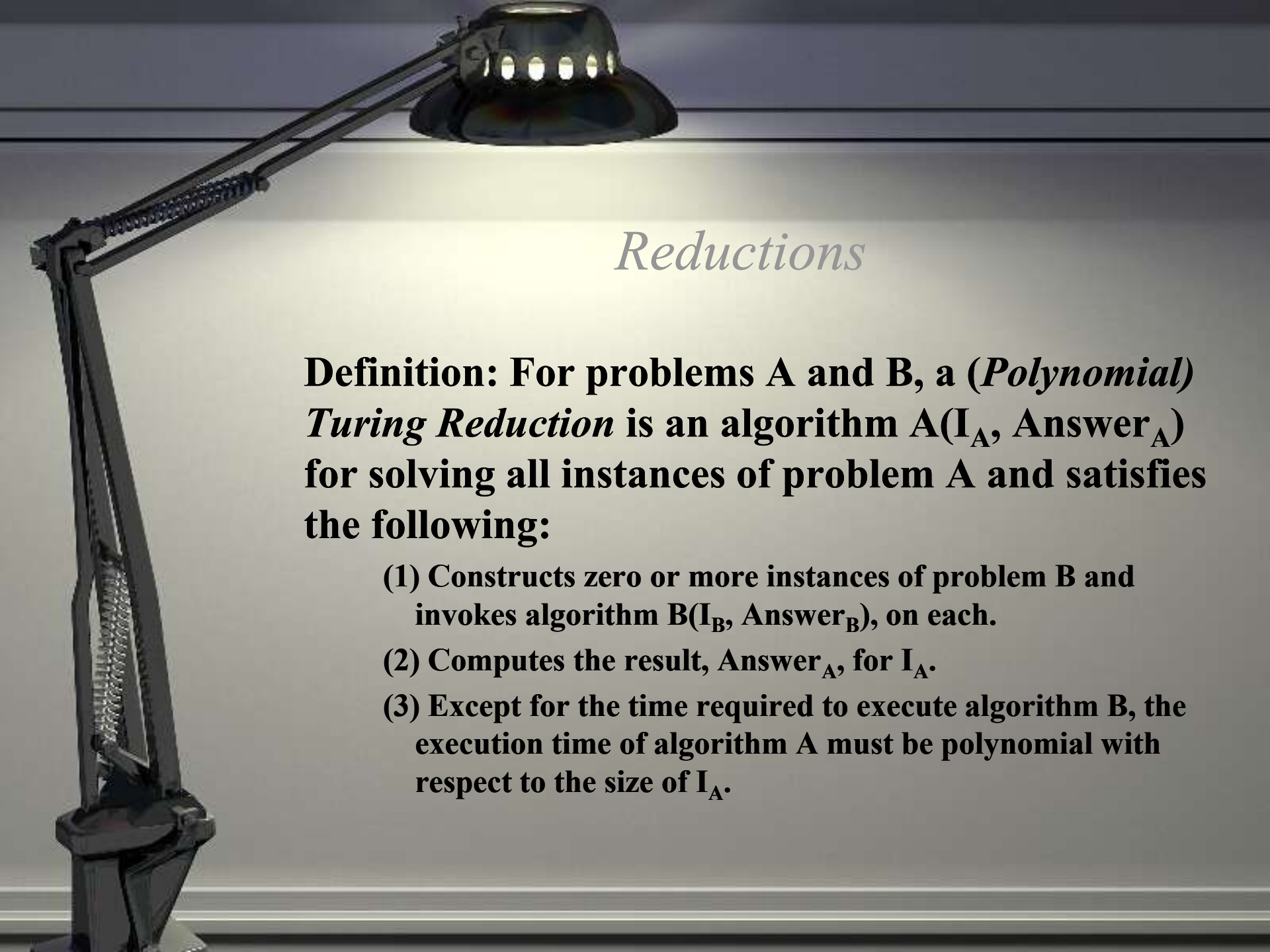

The "Key" to Complexity Theory
'Reductions,' 'Reductions,' 'Reductions.'



Reductions

For any problem X, let $X(I_X, Answer_X)$ represents an algorithm for problem X – even if none is known to exist.

- **I_X is an arbitray instance given to the algorithm and $Answer_X$ is the returned answer determined by the algorithm.**



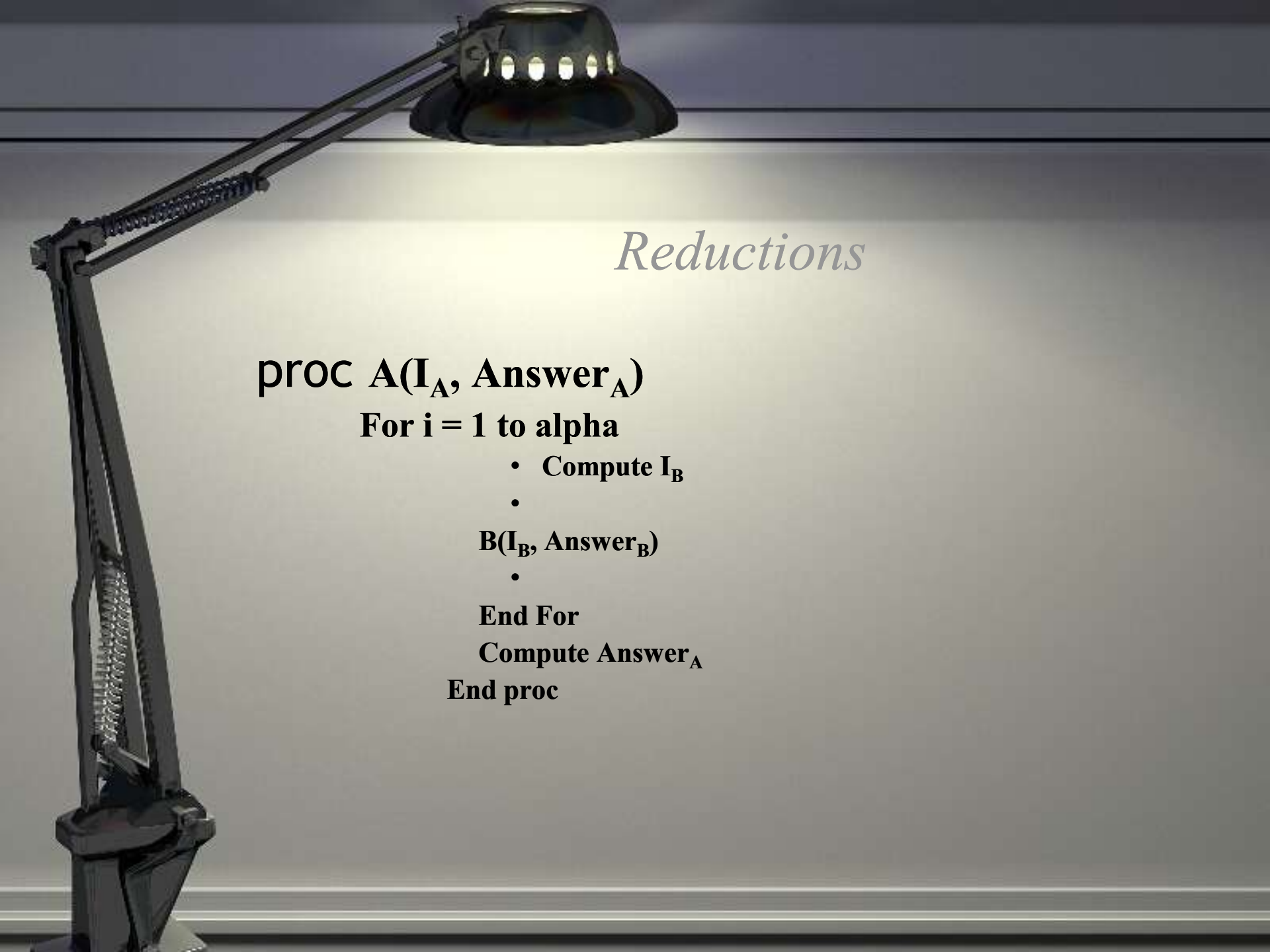
Reductions

Definition: For problems A and B, a (*Polynomial) Turing Reduction* is an algorithm $A(I_A, Answer_A)$ for solving all instances of problem A and satisfies the following:

(1) Constructs zero or more instances of problem B and invokes algorithm $B(I_B, Answer_B)$, on each.

(2) Computes the result, $Answer_A$, for I_A.

(3) Except for the time required to execute algorithm B, the execution time of algorithm A must be polynomial with respect to the size of I_A.



Reductions

proc $A(I_A, Answer_A)$

For i = 1 to alpha

- **Compute I_B**
-

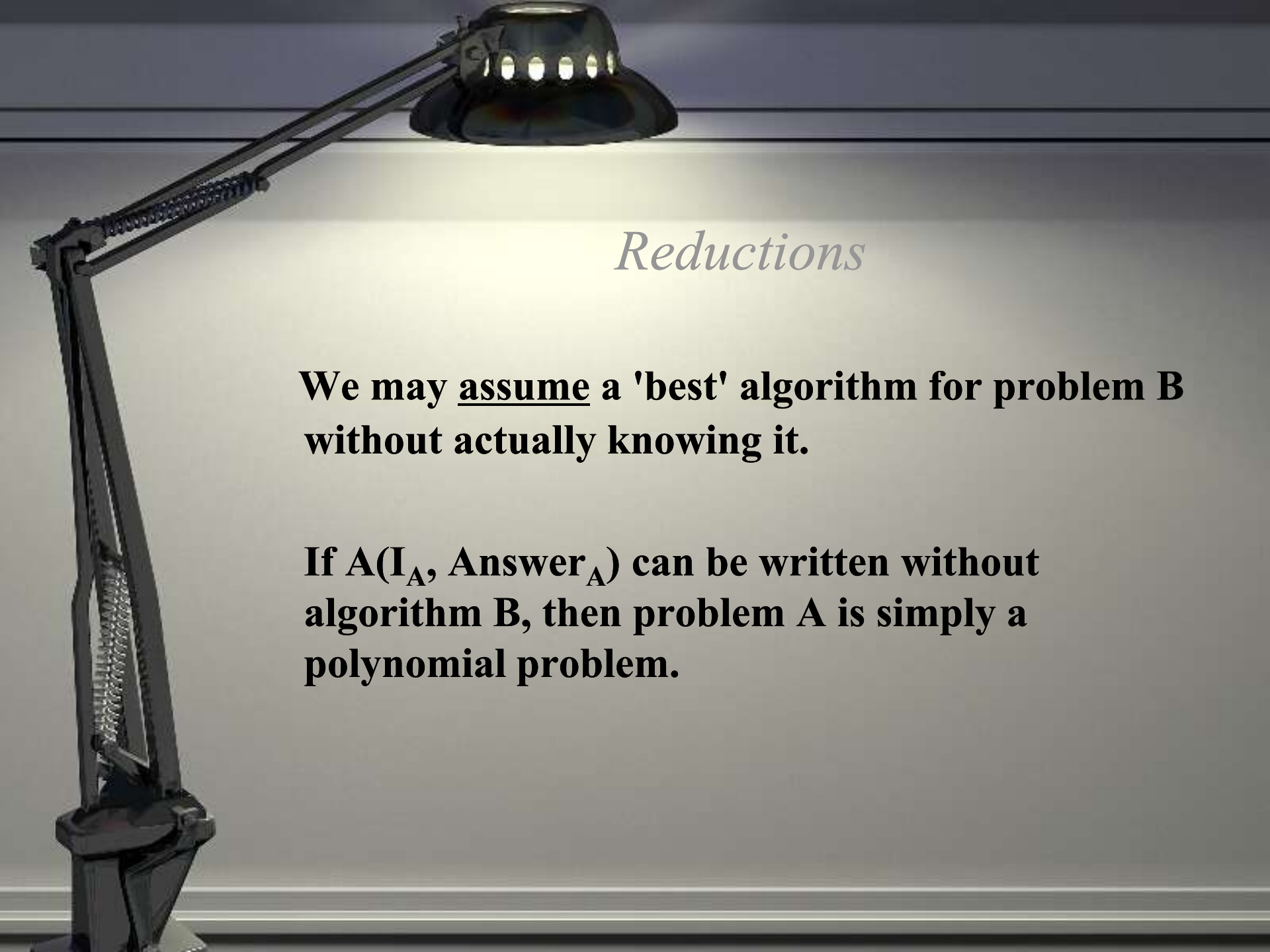
$B(I_B, Answer_B)$

-

End For

Compute $Answer_A$

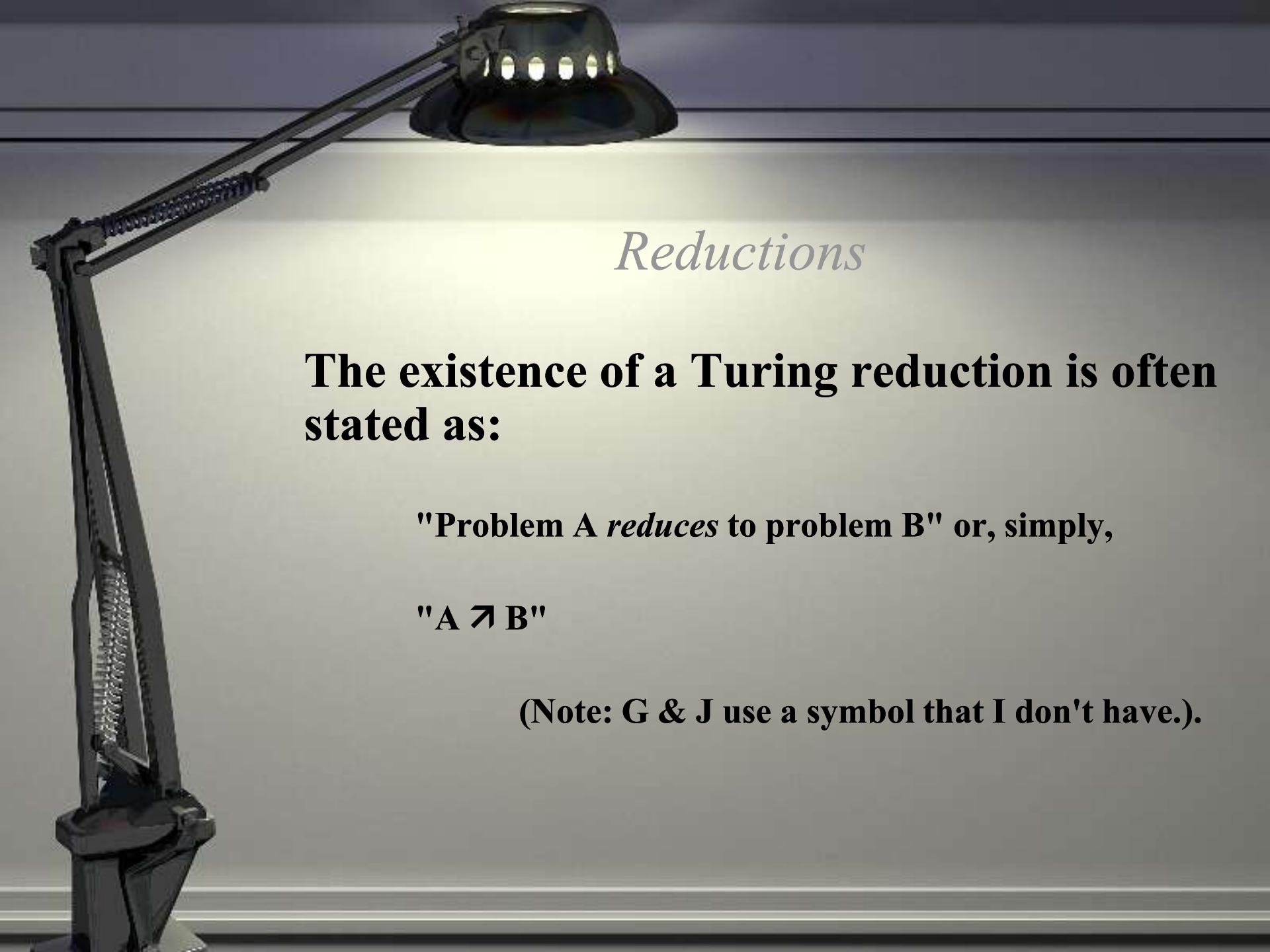
End proc



Reductions

We may <u>assume</u> a 'best' algorithm for problem B without actually knowing it.

If $A(I_A, Answer_A)$ can be written without algorithm B, then problem A is simply a polynomial problem.



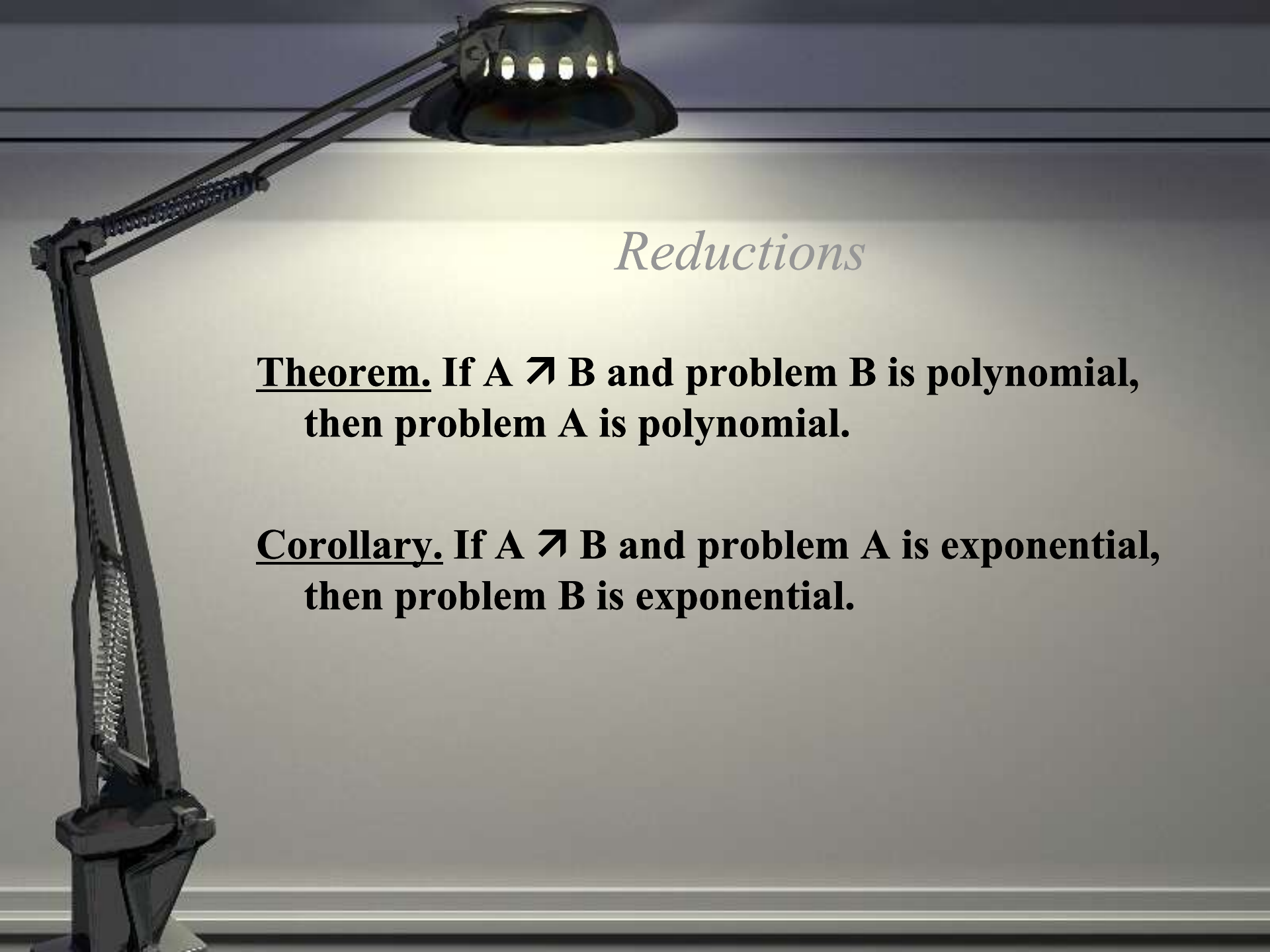
Reductions

The existence of a Turing reduction is often stated as:

"Problem A *reduces* to problem B" or, simply,

"A ↗ B"

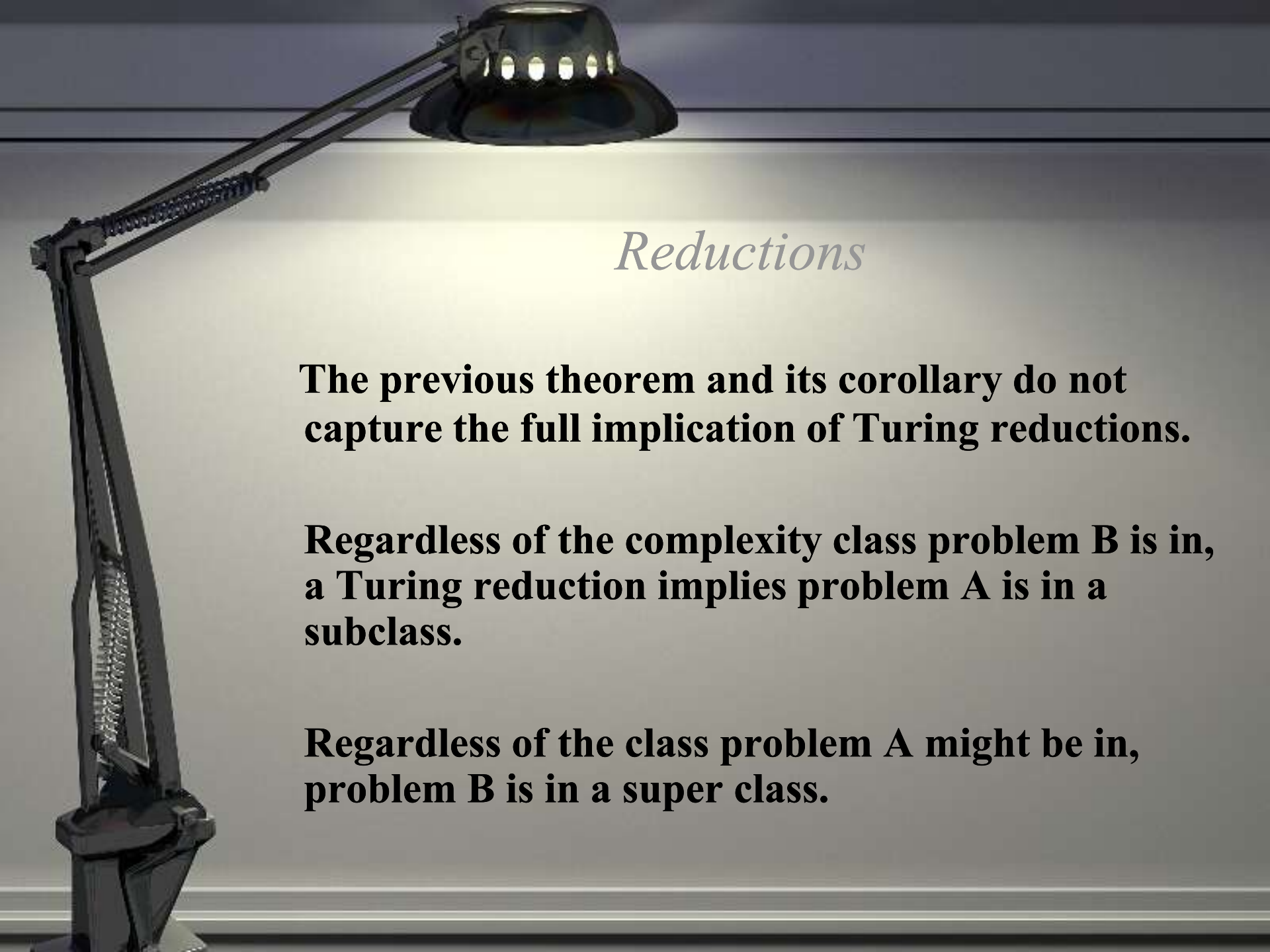
(Note: G & J use a symbol that I don't have.).



Reductions

<u>Theorem.</u> If A ↗ B and problem B is polynomial, then problem A is polynomial.

<u>Corollary.</u> If A ↗ B and problem A is exponential, then problem B is exponential.

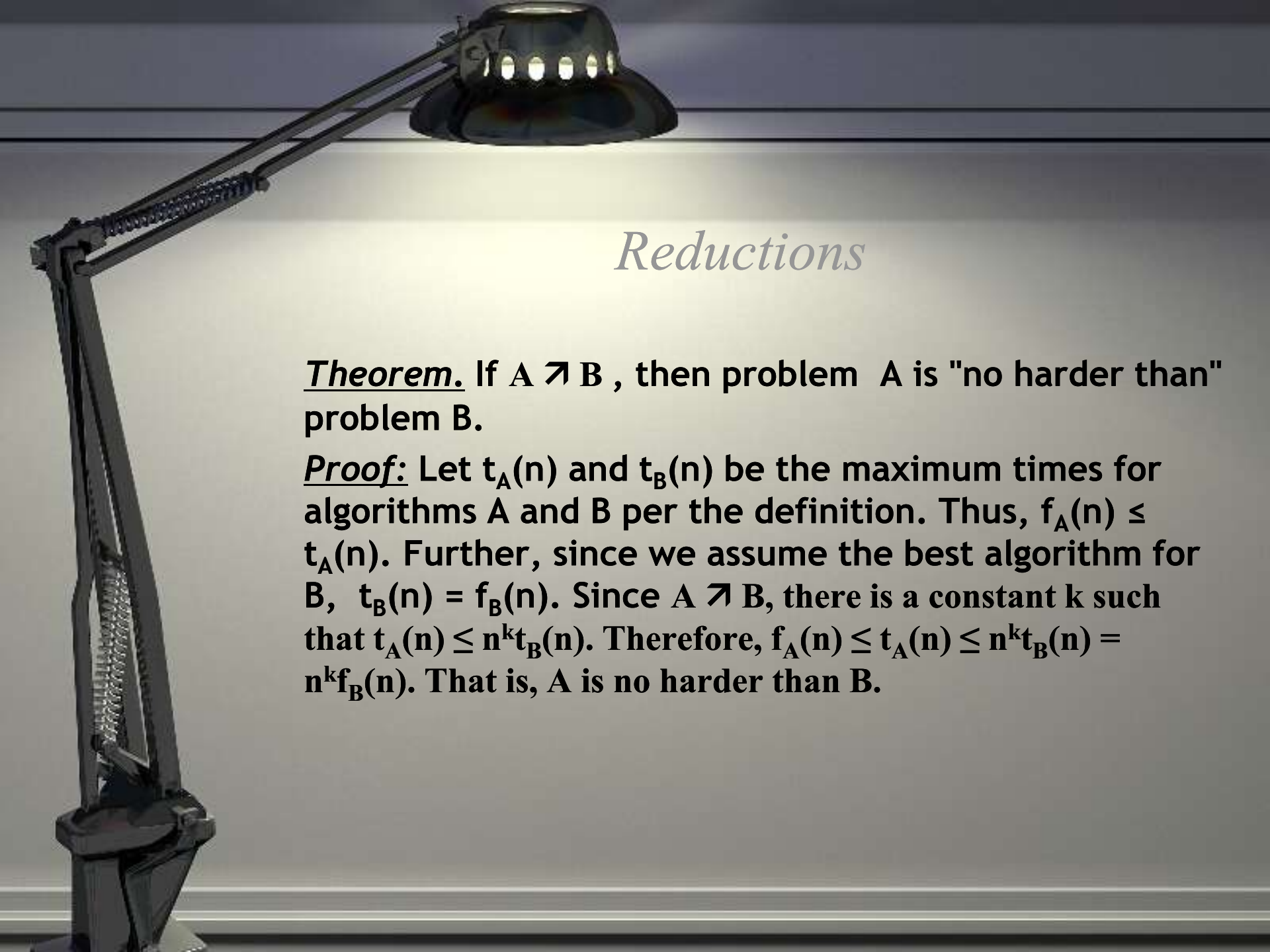


Reductions

The previous theorem and its corollary do not capture the full implication of Turing reductions.

Regardless of the complexity class problem B is in, a Turing reduction implies problem A is in a subclass.

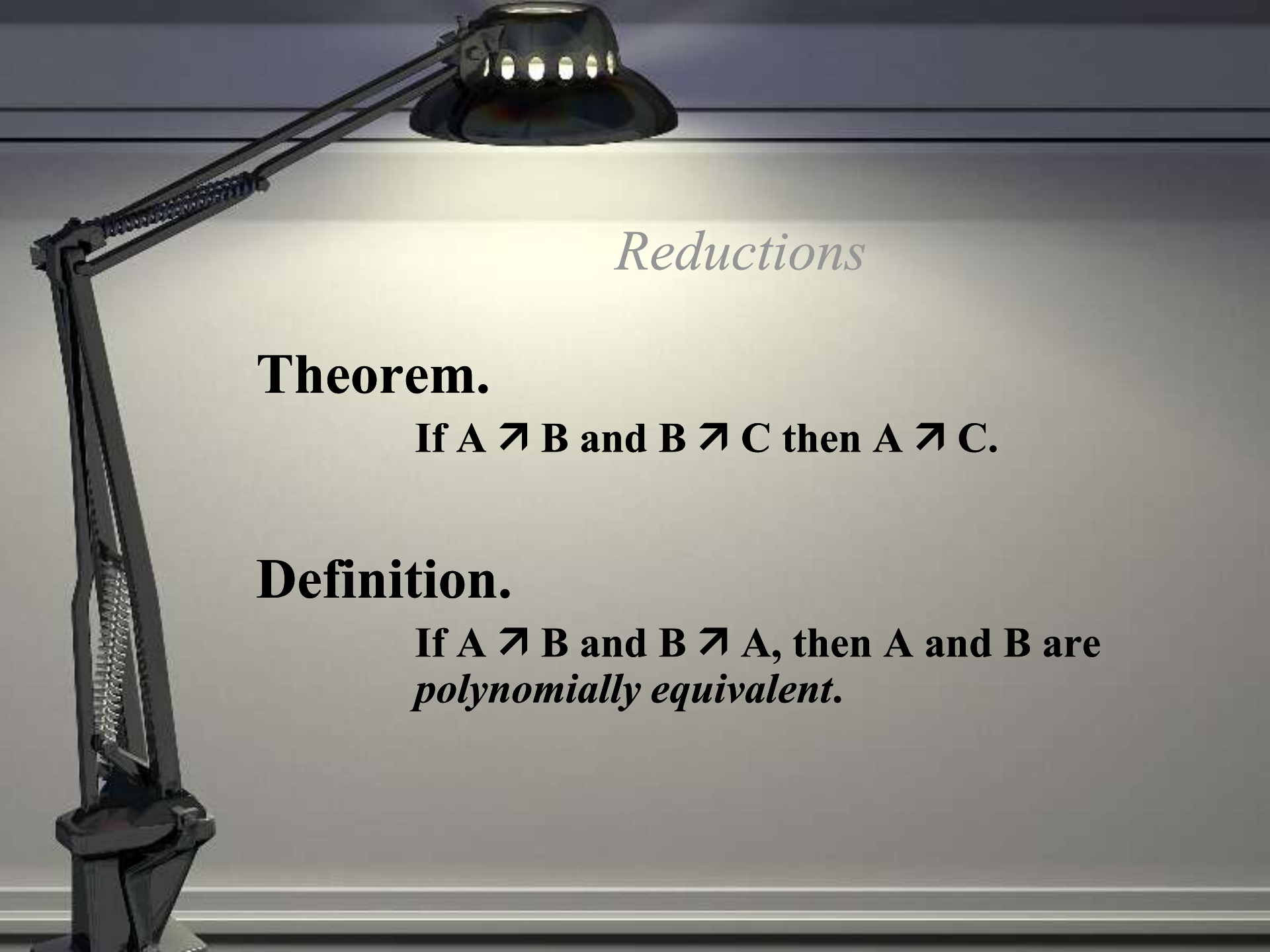
Regardless of the class problem A might be in, problem B is in a super class.



Reductions

Theorem. If $A \nearrow B$, then problem A is "no harder than" problem B.

Proof: Let $t_A(n)$ and $t_B(n)$ be the maximum times for algorithms A and B per the definition. Thus, $f_A(n) \leq t_A(n)$. Further, since we assume the best algorithm for B, $t_B(n) = f_B(n)$. Since $A \nearrow B$, there is a constant k such that $t_A(n) \leq n^k t_B(n)$. Therefore, $f_A(n) \leq t_A(n) \leq n^k t_B(n) = n^k f_B(n)$. That is, A is no harder than B.



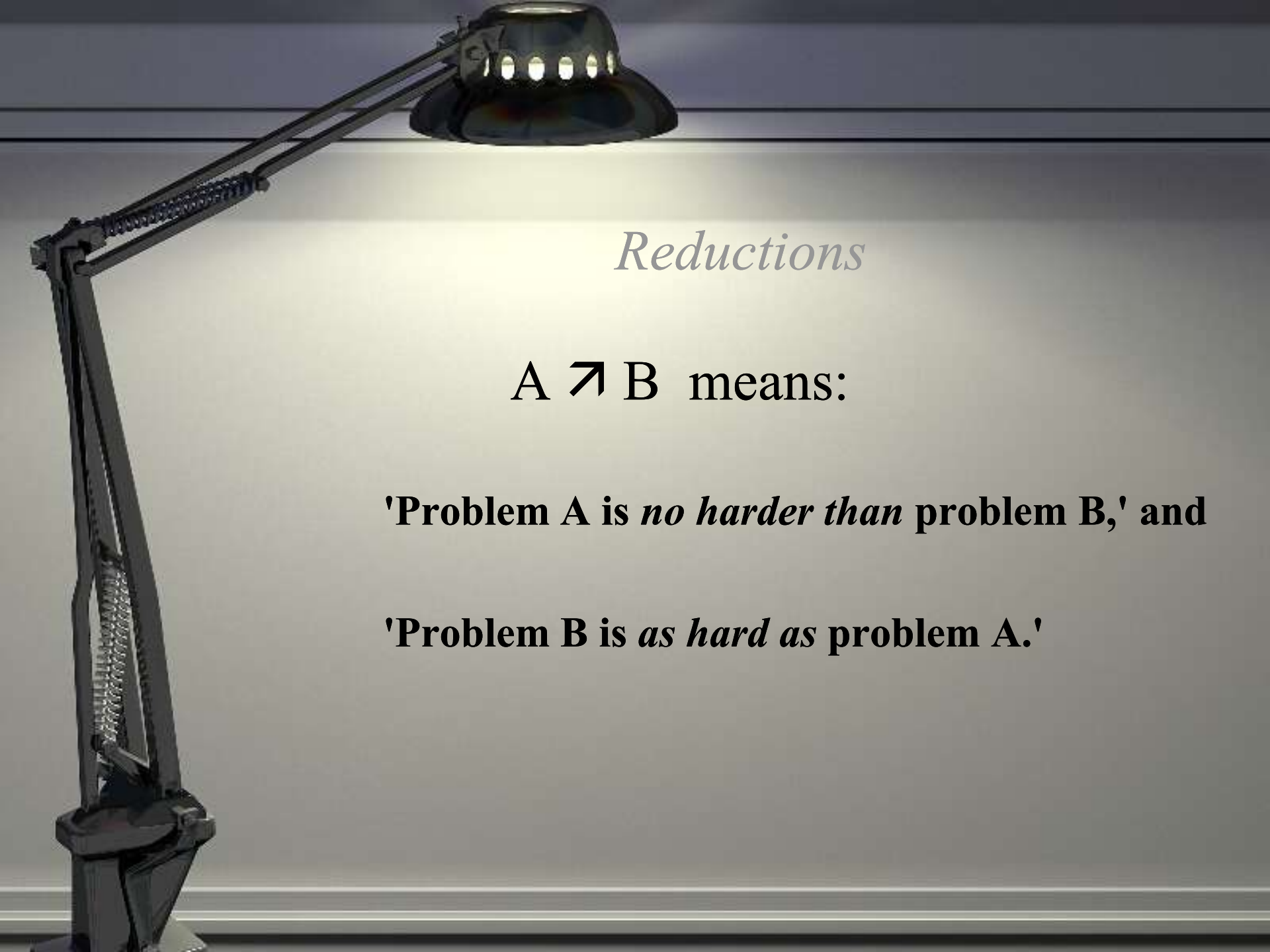
Reductions

Theorem.

If A ↗ B and B ↗ C then A ↗ C.

Definition.

If A ↗ B and B ↗ A, then A and B are *polynomially equivalent*.

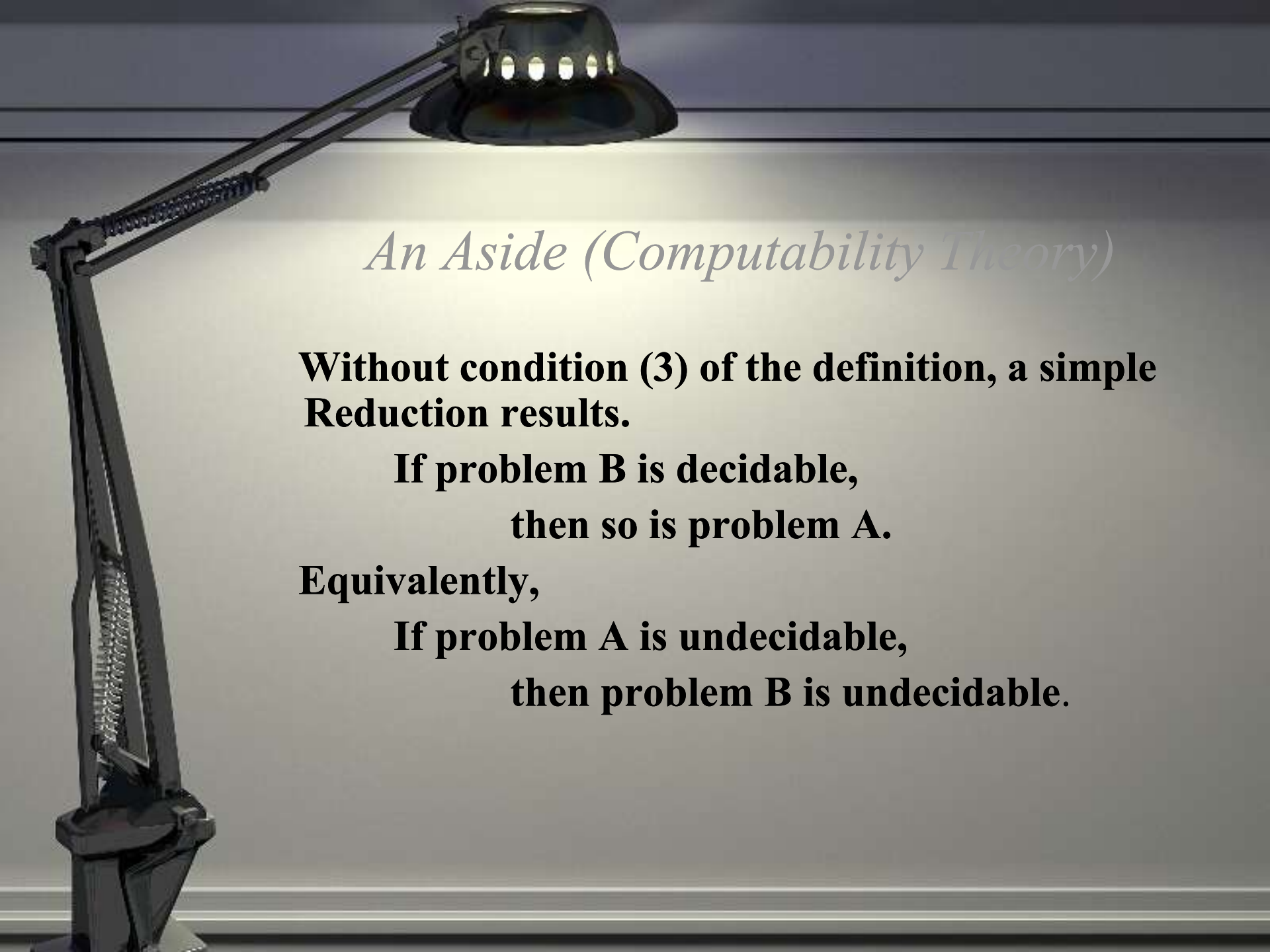


Reductions

A ↗ B means:

'Problem A is *no harder than* problem B,' and

'Problem B is *as hard as* problem A.'



An Aside (Computability Theory)

Without condition (3) of the definition, a simple Reduction results.

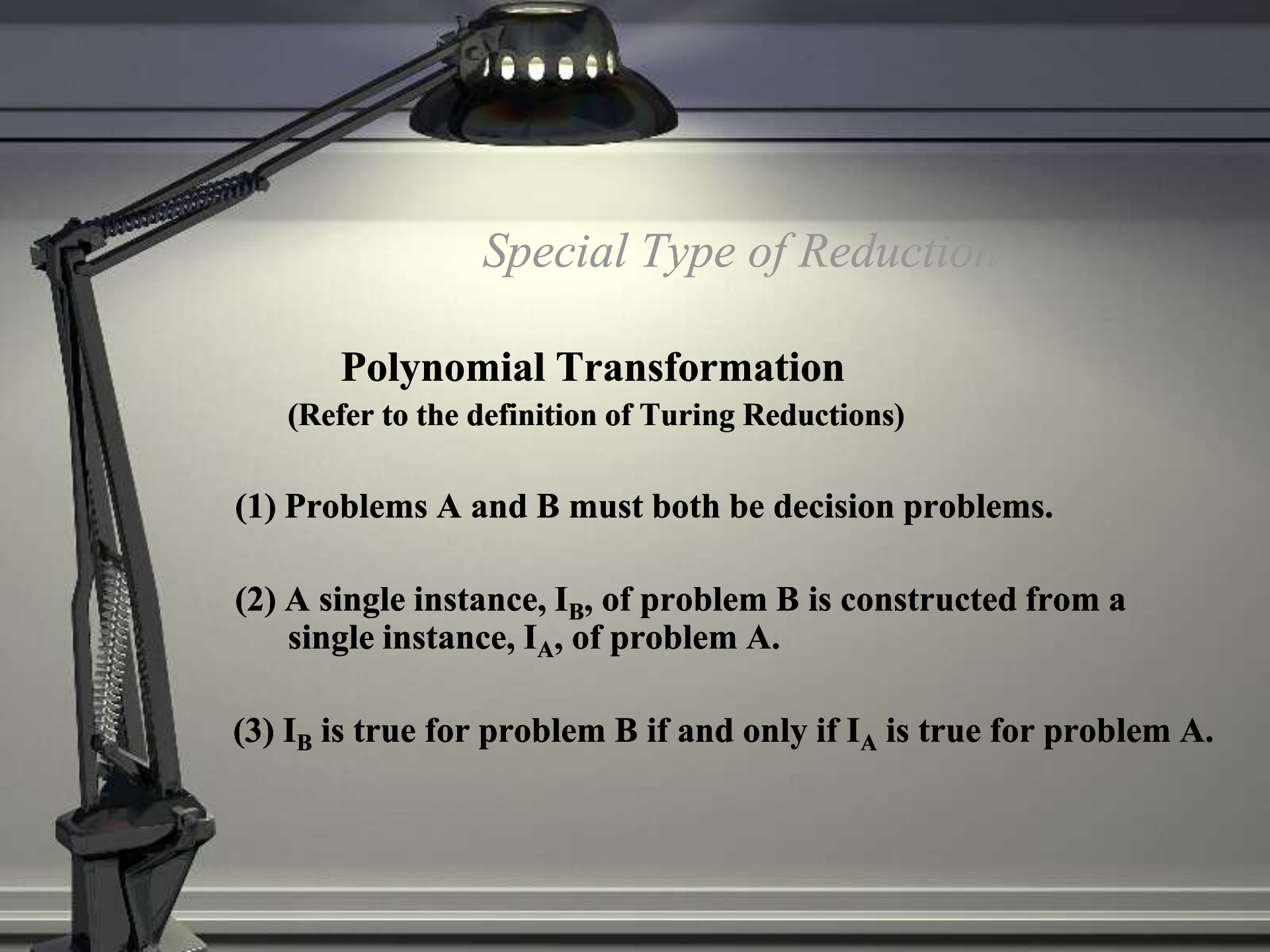
If problem B is decidable,

then so is problem A.

Equivalently,

If problem A is undecidable,

then problem B is undecidable.



Special Type of Reduction

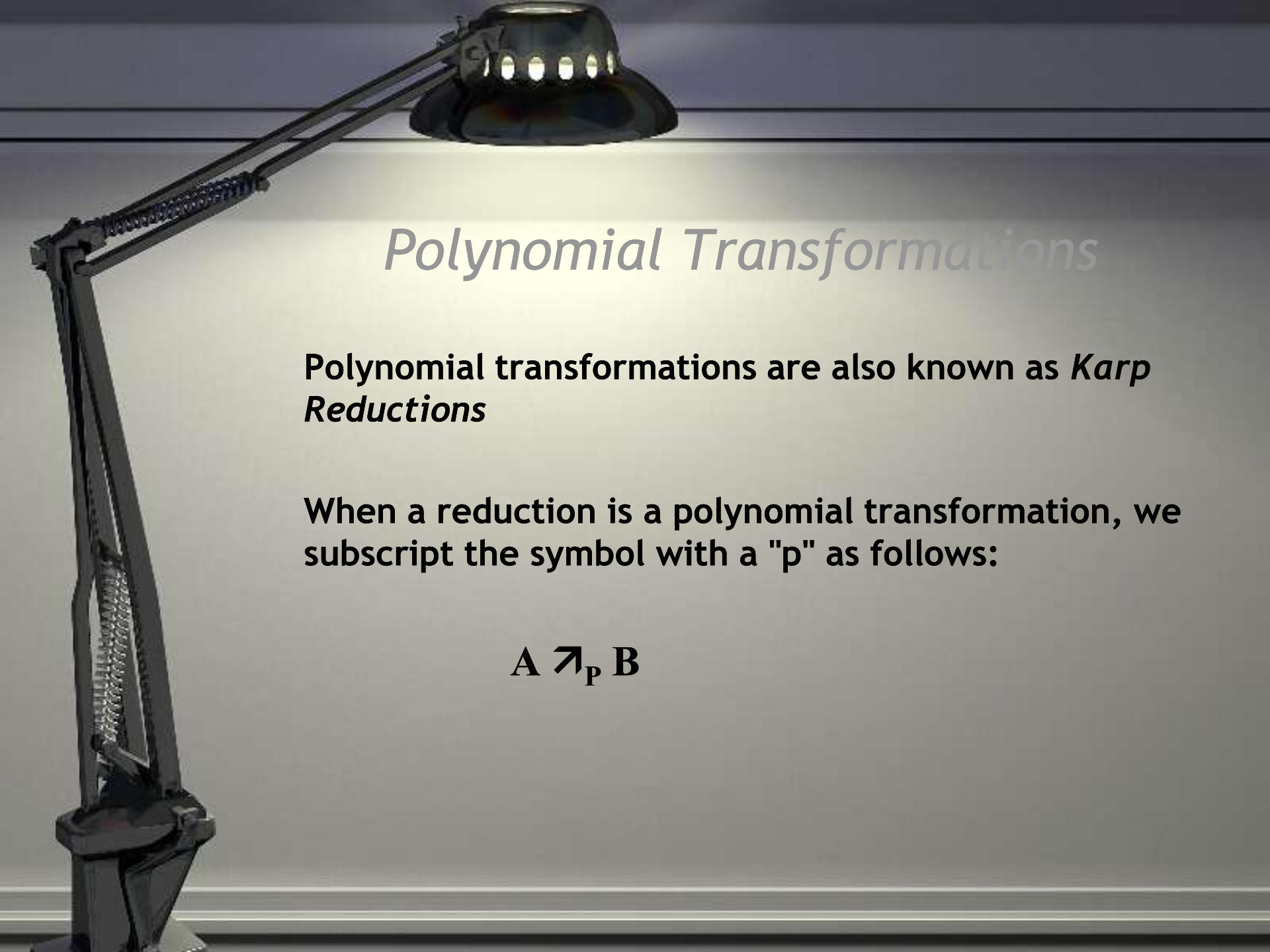
Polynomial Transformation

(Refer to the definition of Turing Reductions)

(1) Problems A and B must both be decision problems.

(2) A single instance, I_B, of problem B is constructed from a single instance, I_A, of problem A.

(3) I_B is true for problem B if and only if I_A is true for problem A.

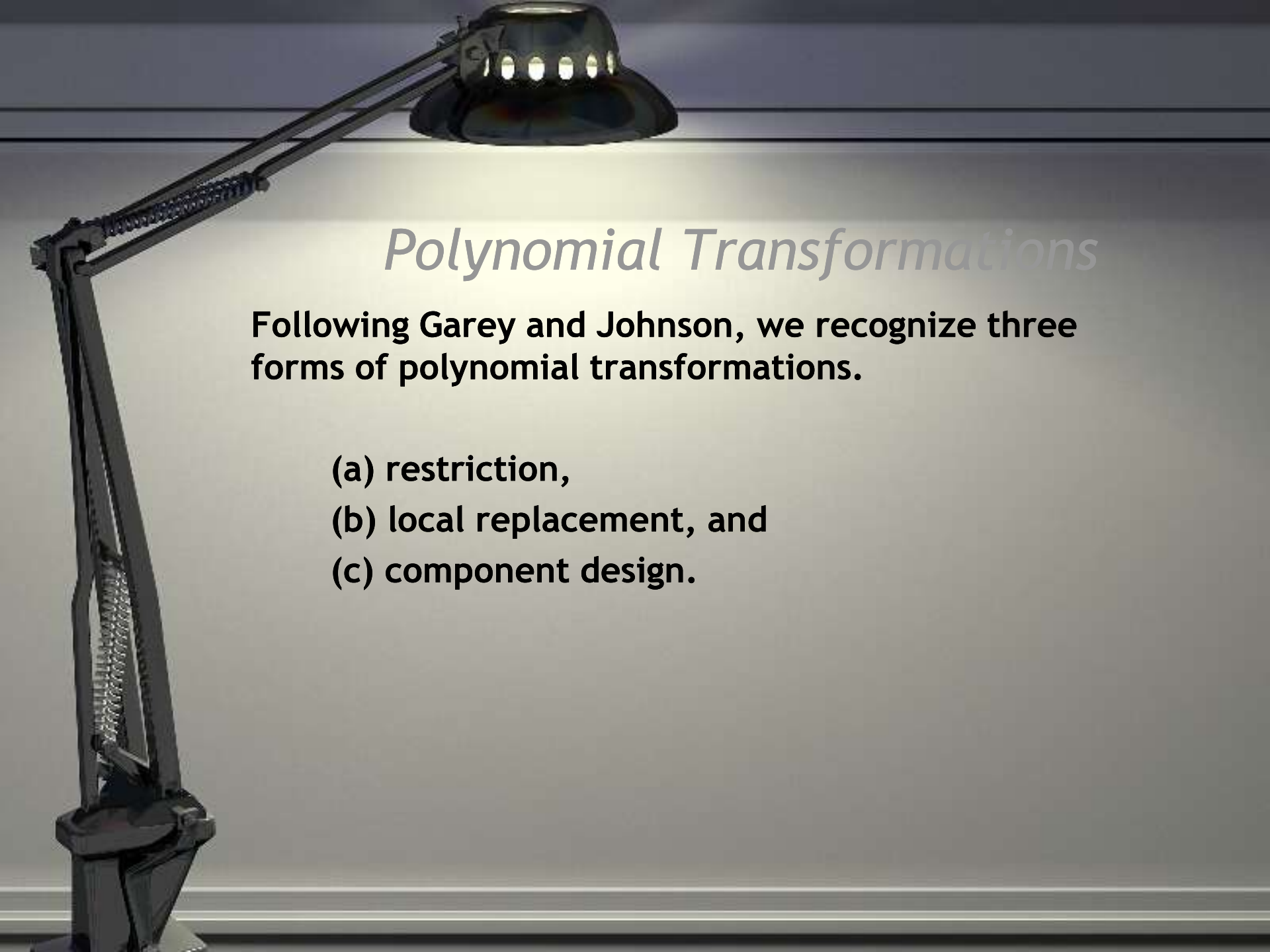


Polynomial Transformations

Polynomial transformations are also known as *Karp Reductions*

When a reduction is a polynomial transformation, we subscript the symbol with a "p" as follows:

$$A \nearrow_P B$$



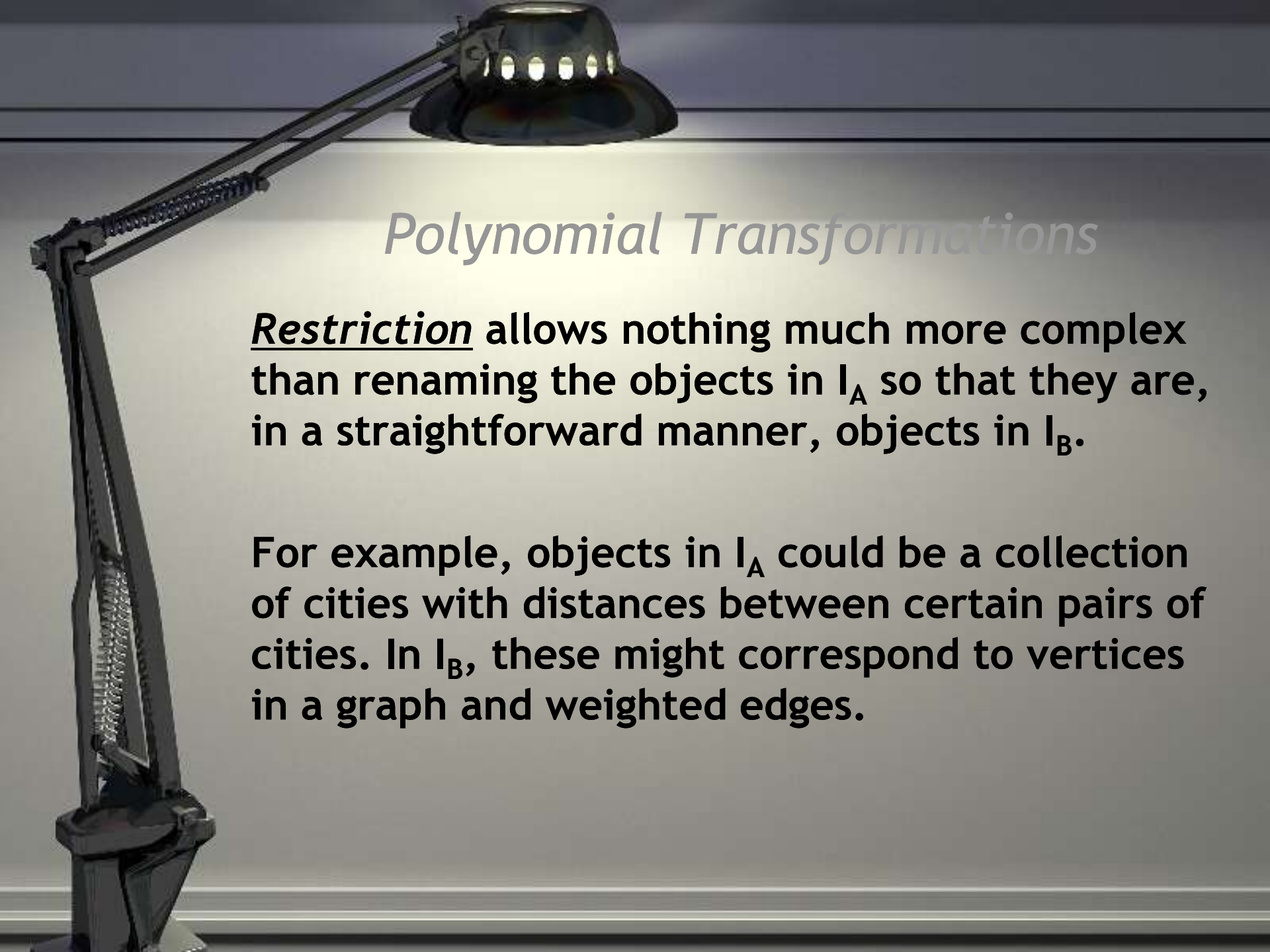
Polynomial Transformations

Following Garey and Johnson, we recognize three forms of polynomial transformations.

(a) restriction,

(b) local replacement, and

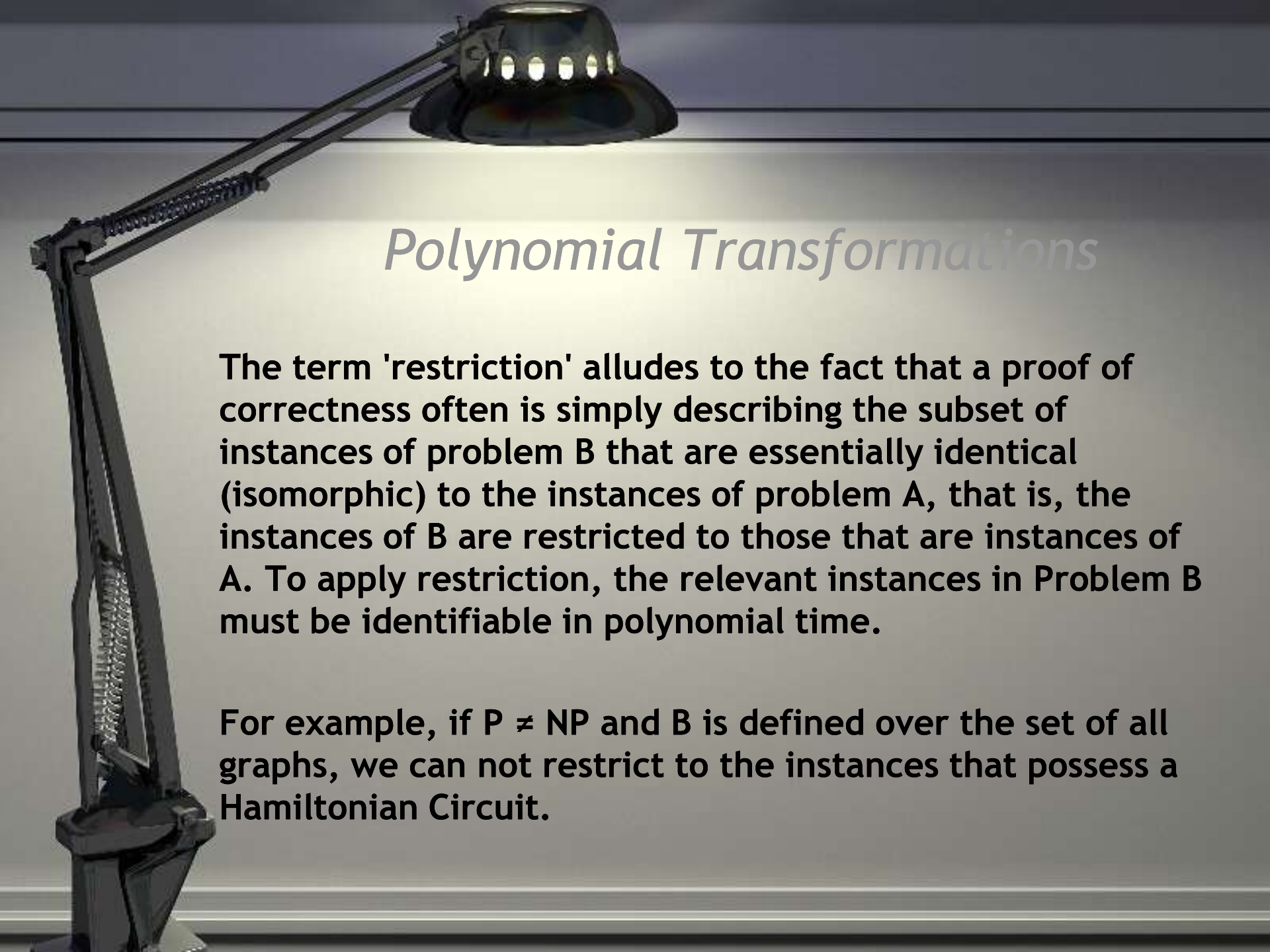
(c) component design.



Polynomial Transformations

Restriction **allows nothing much more complex than renaming the objects in I_A so that they are, in a straightforward manner, objects in I_B.**

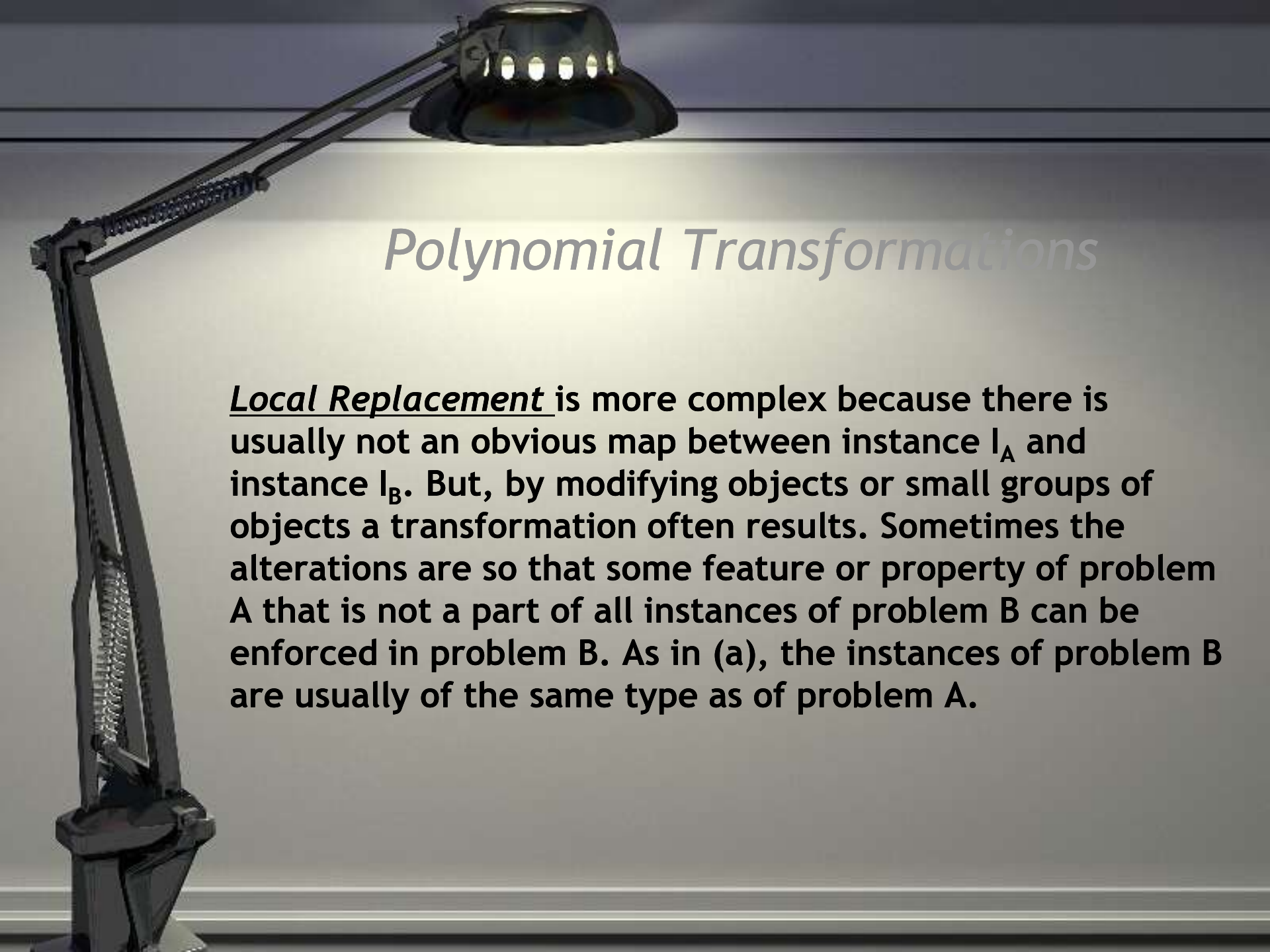
For example, objects in I_A could be a collection of cities with distances between certain pairs of cities. In I_B, these might correspond to vertices in a graph and weighted edges.



Polynomial Transformations

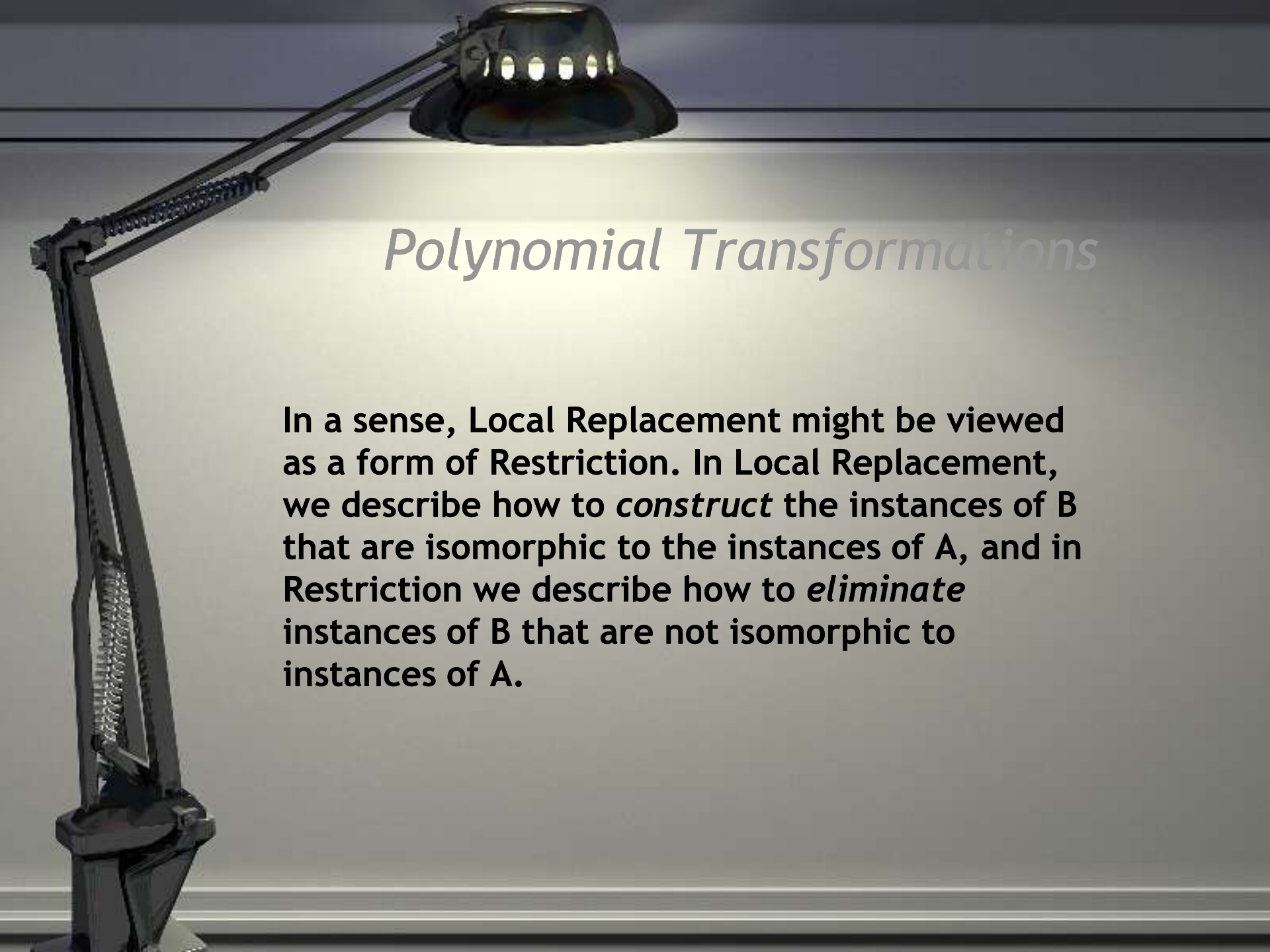
The term 'restriction' alludes to the fact that a proof of correctness often is simply describing the subset of instances of problem B that are essentially identical (isomorphic) to the instances of problem A, that is, the instances of B are restricted to those that are instances of A. To apply restriction, the relevant instances in Problem B must be identifiable in polynomial time.

For example, if P ≠ NP and B is defined over the set of all graphs, we can not restrict to the instances that possess a Hamiltonian Circuit.



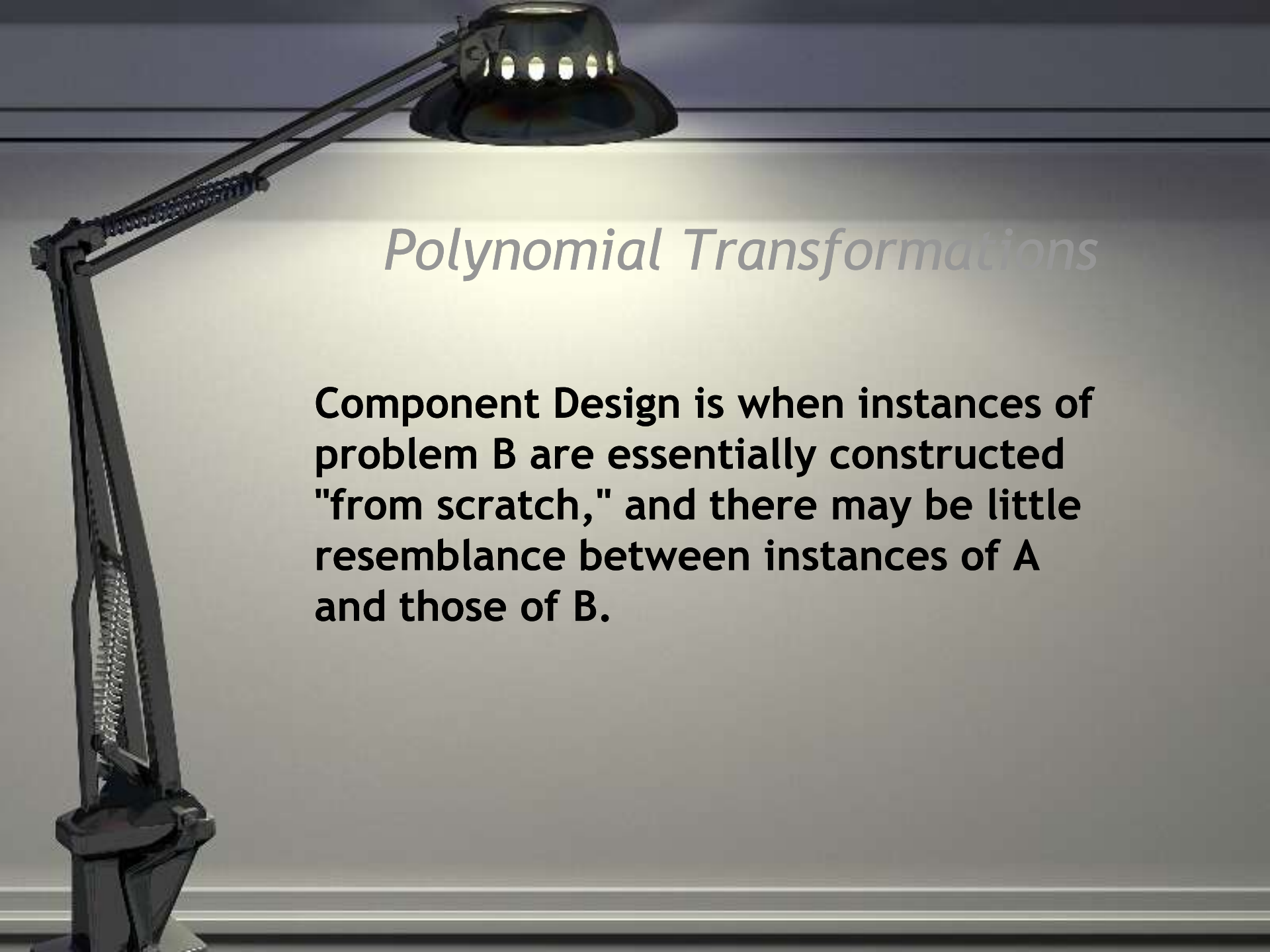
Polynomial Transformations

Local Replacement **is more complex because there is usually not an obvious map between instance I_A and instance I_B. But, by modifying objects or small groups of objects a transformation often results. Sometimes the alterations are so that some feature or property of problem A that is not a part of all instances of problem B can be enforced in problem B. As in (a), the instances of problem B are usually of the same type as of problem A.**



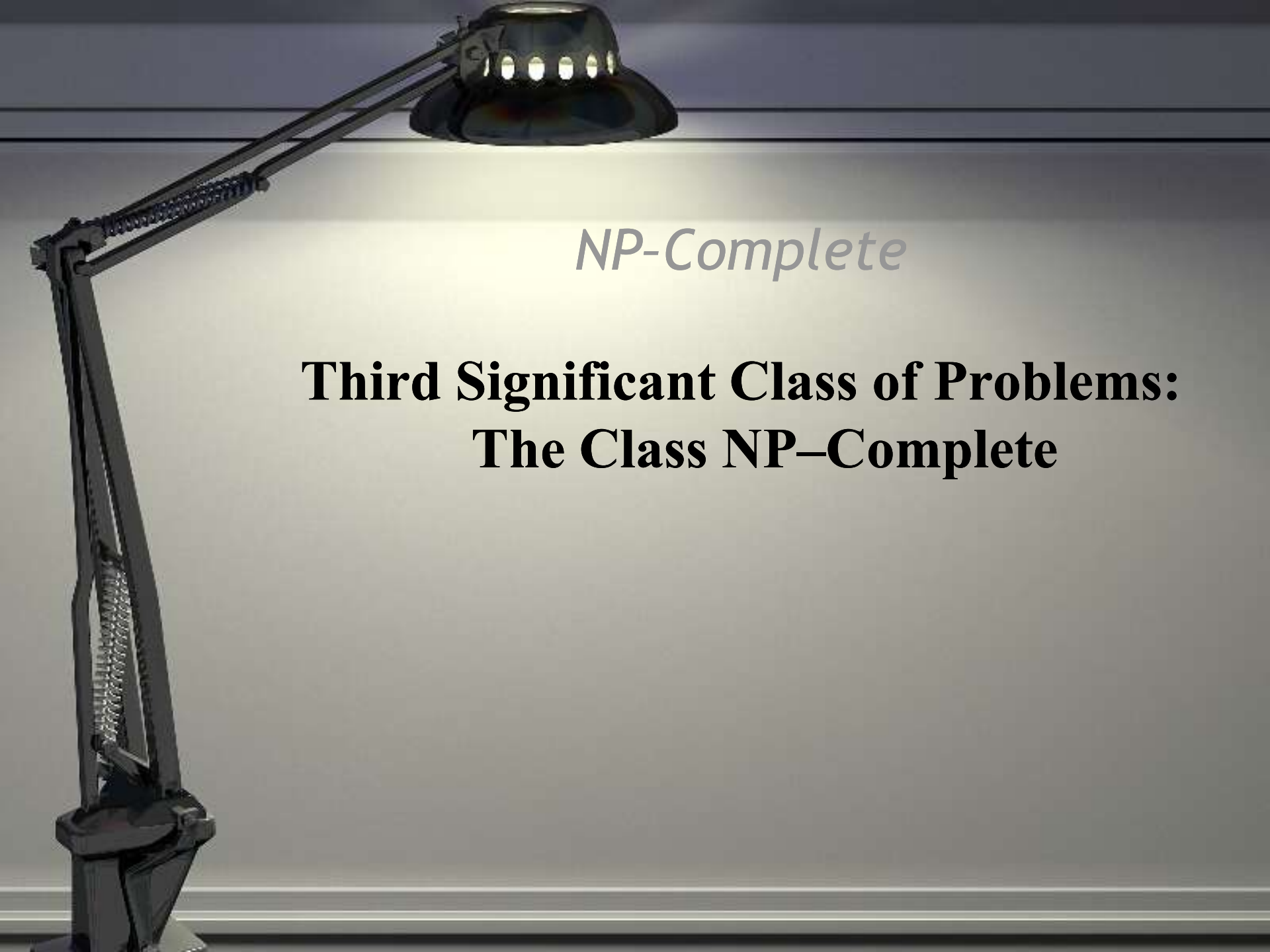
Polynomial Transformations

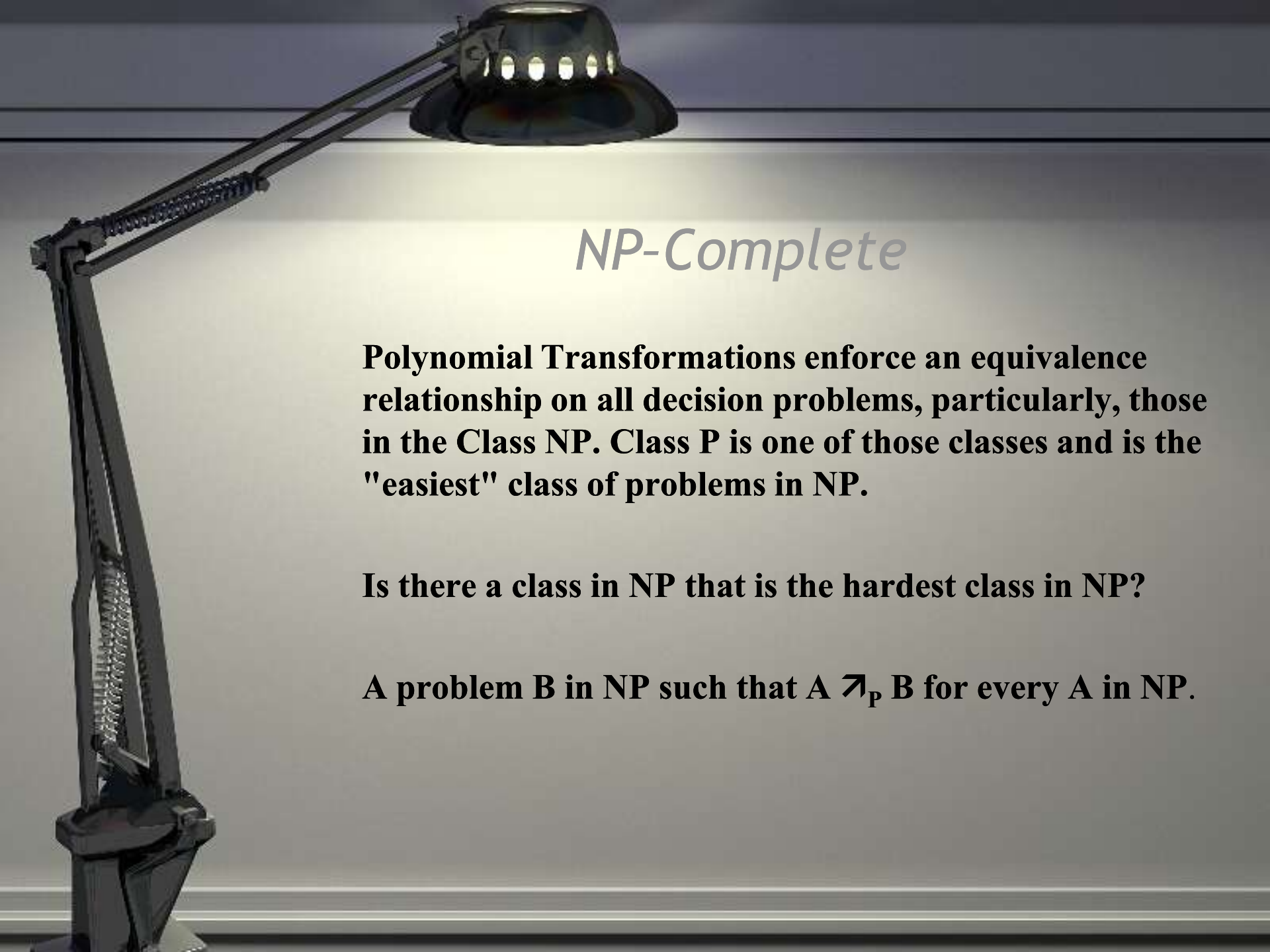
In a sense, Local Replacement might be viewed as a form of Restriction. In Local Replacement, we describe how to *construct* the instances of B that are isomorphic to the instances of A, and in Restriction we describe how to *eliminate* instances of B that are not isomorphic to instances of A.



Polynomial Transformations

Component Design is when instances of problem B are essentially constructed "from scratch," and there may be little resemblance between instances of A and those of B.


NP-Complete
Third Significant Class of Problems:
The Class NP–Complete

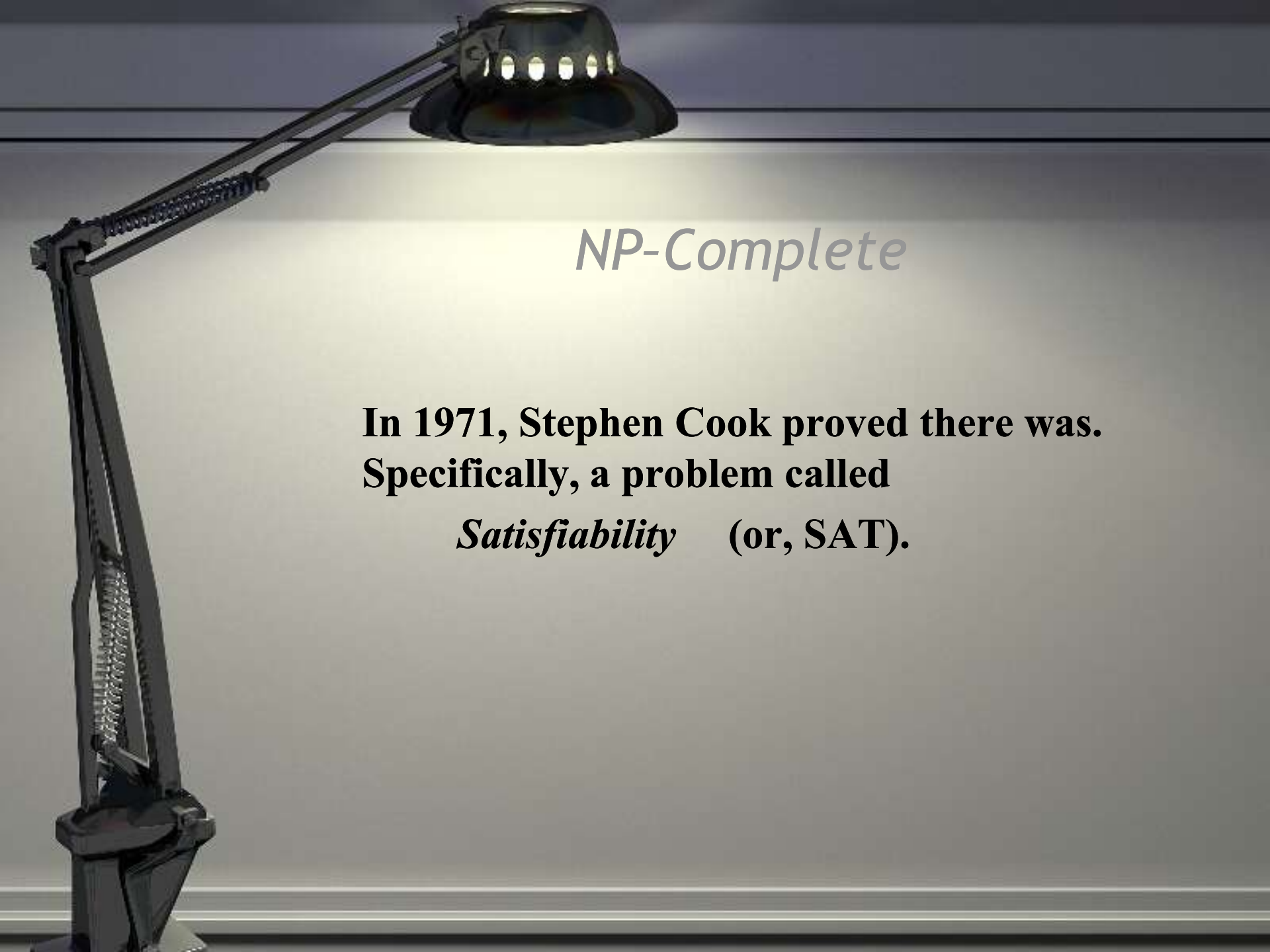


NP-Complete

Polynomial Transformations enforce an equivalence relationship on all decision problems, particularly, those in the Class NP. Class P is one of those classes and is the "easiest" class of problems in NP.

Is there a class in NP that is the hardest class in NP?

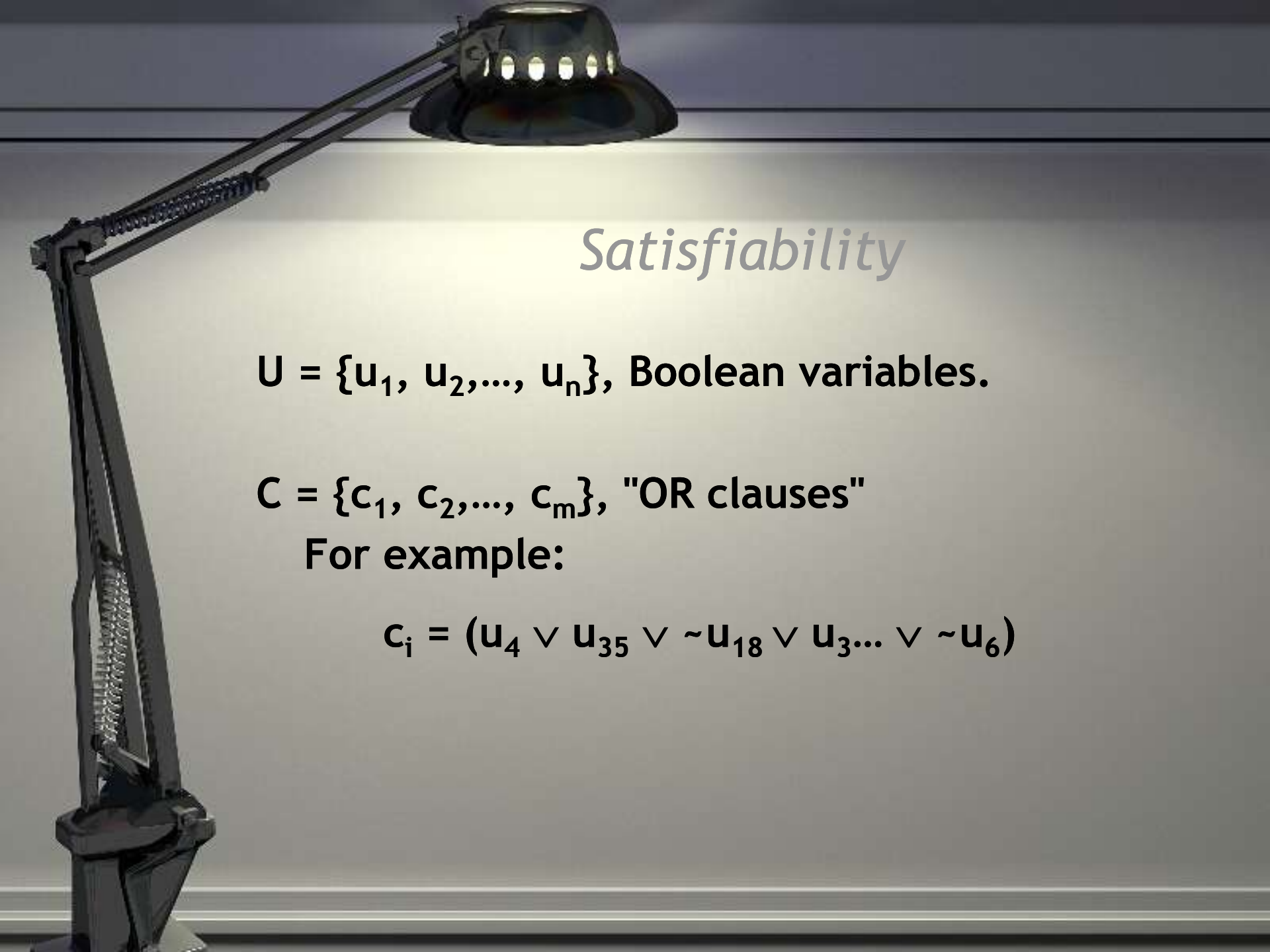
A problem B in NP such that A $\nearrow_P$ B for every A in NP.



NP-Complete

In 1971, Stephen Cook proved there was.
Specifically, a problem called

Satisfiability **(or, SAT).**



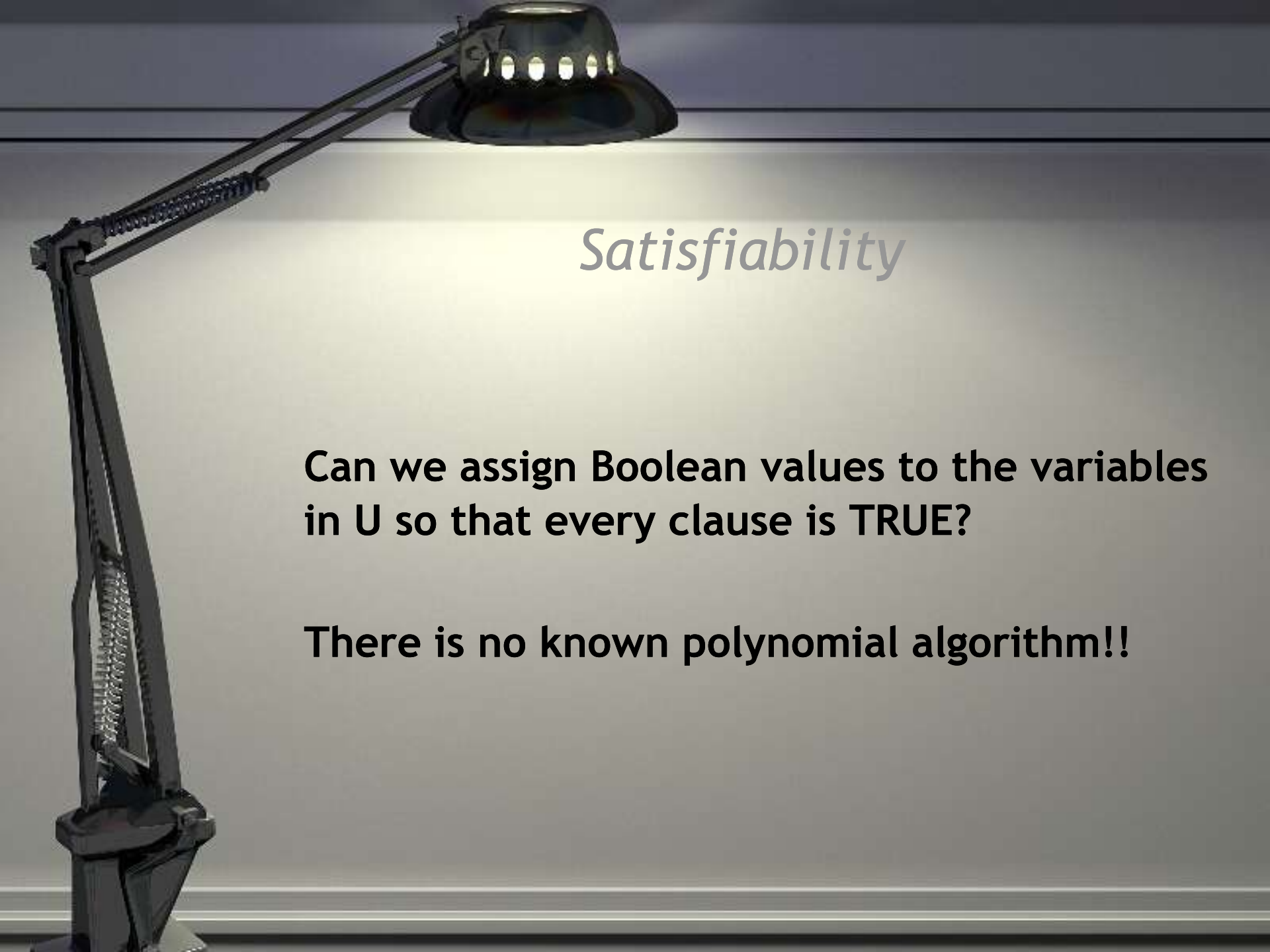
Satisfiability

$U = \{u_1, u_2, \ldots, u_n\}$, Boolean variables.

$C = \{c_1, c_2, \ldots, c_m\}$, "OR clauses"

For example:

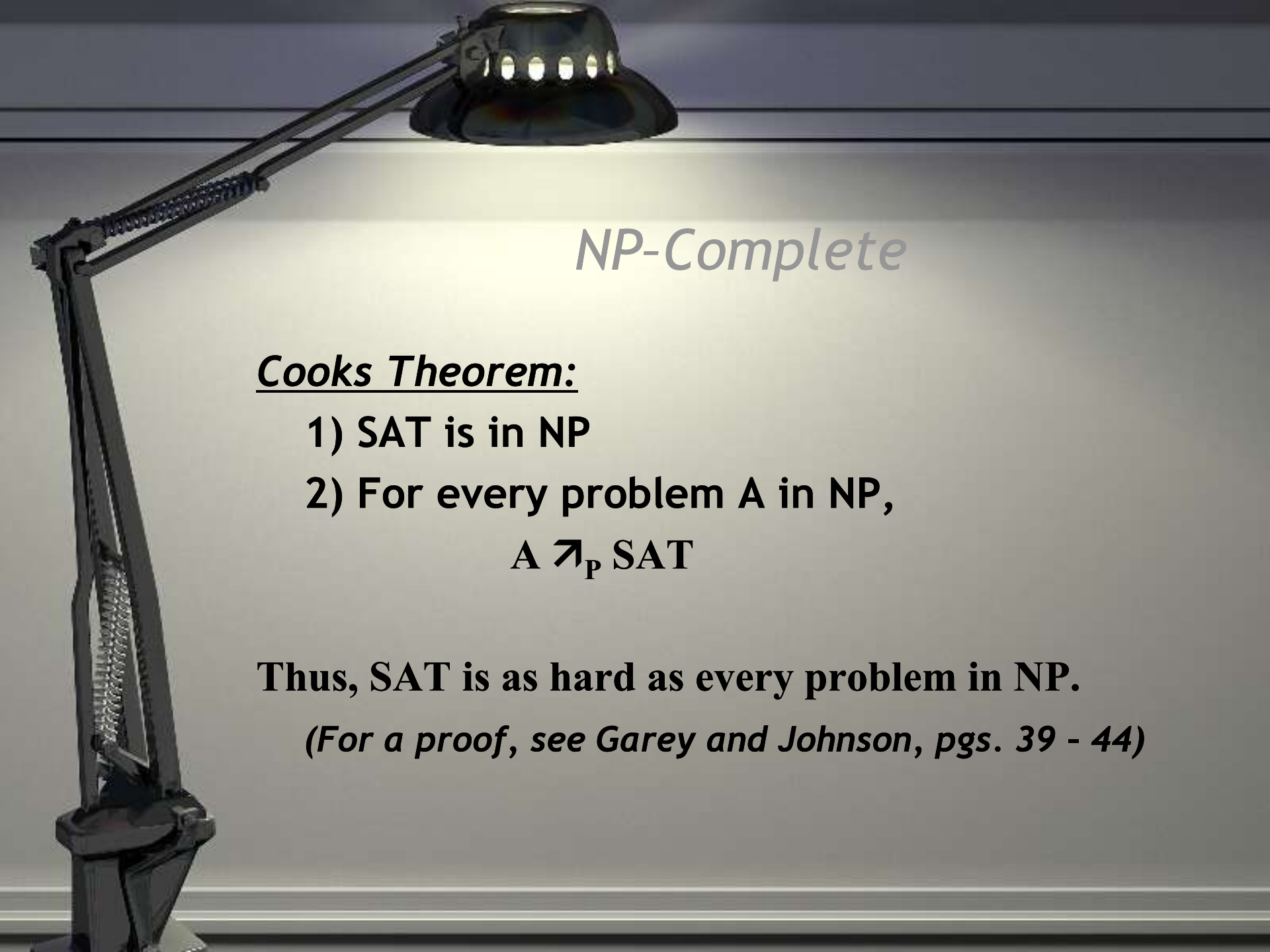
$$c_i = (u_4 \vee u_{35} \vee \sim u_{18} \vee u_3 \ldots \vee \sim u_6)$$



Satisfiability

Can we assign Boolean values to the variables in U so that every clause is TRUE?

There is no known polynomial algorithm!!



NP-Complete

Cooks Theorem:

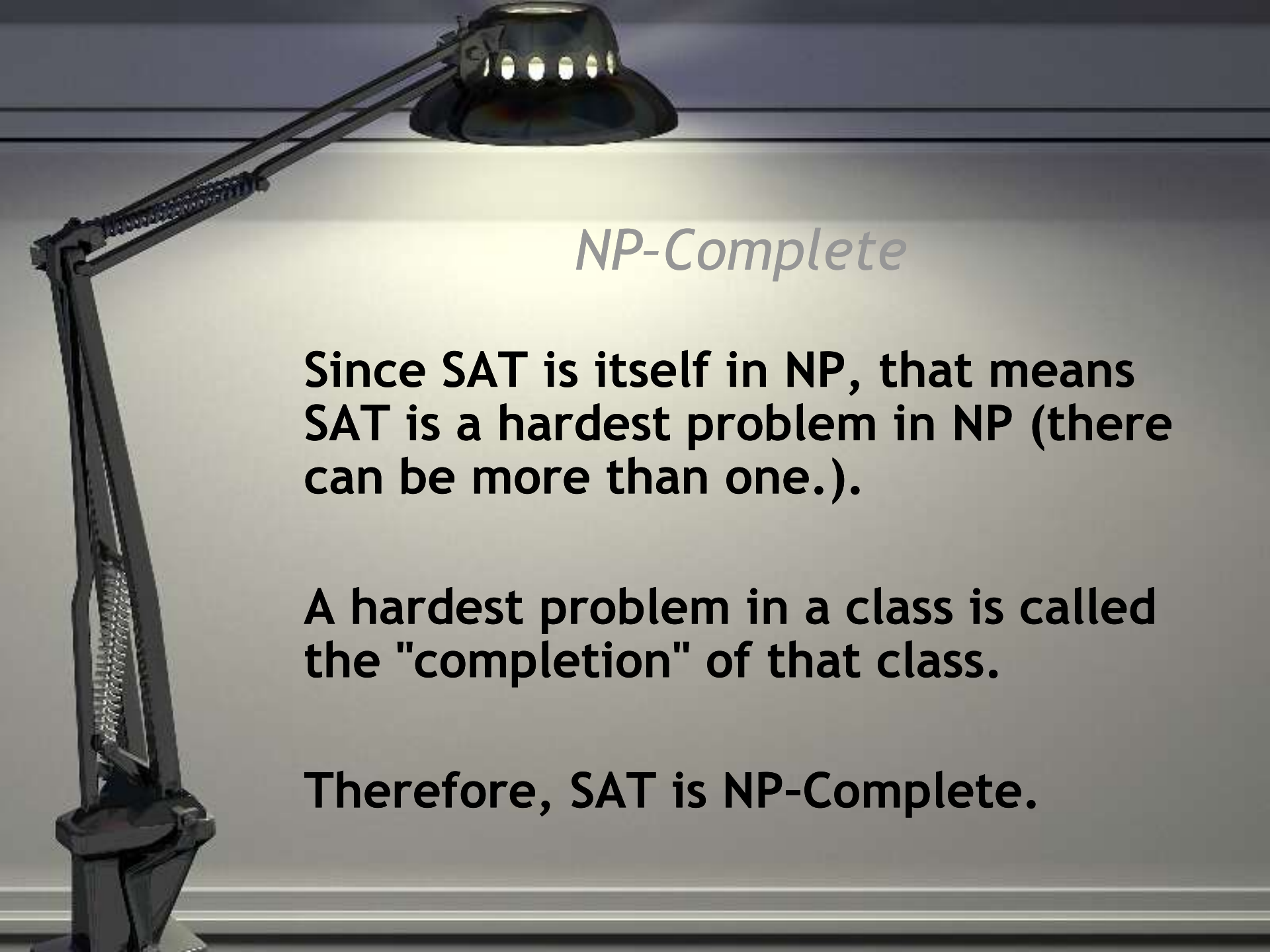
1) SAT is in NP

2) For every problem A in NP,

$$A \nearrow_P SAT$$

Thus, SAT is as hard as every problem in NP.

(For a proof, see Garey and Johnson, pgs. 39 - 44)

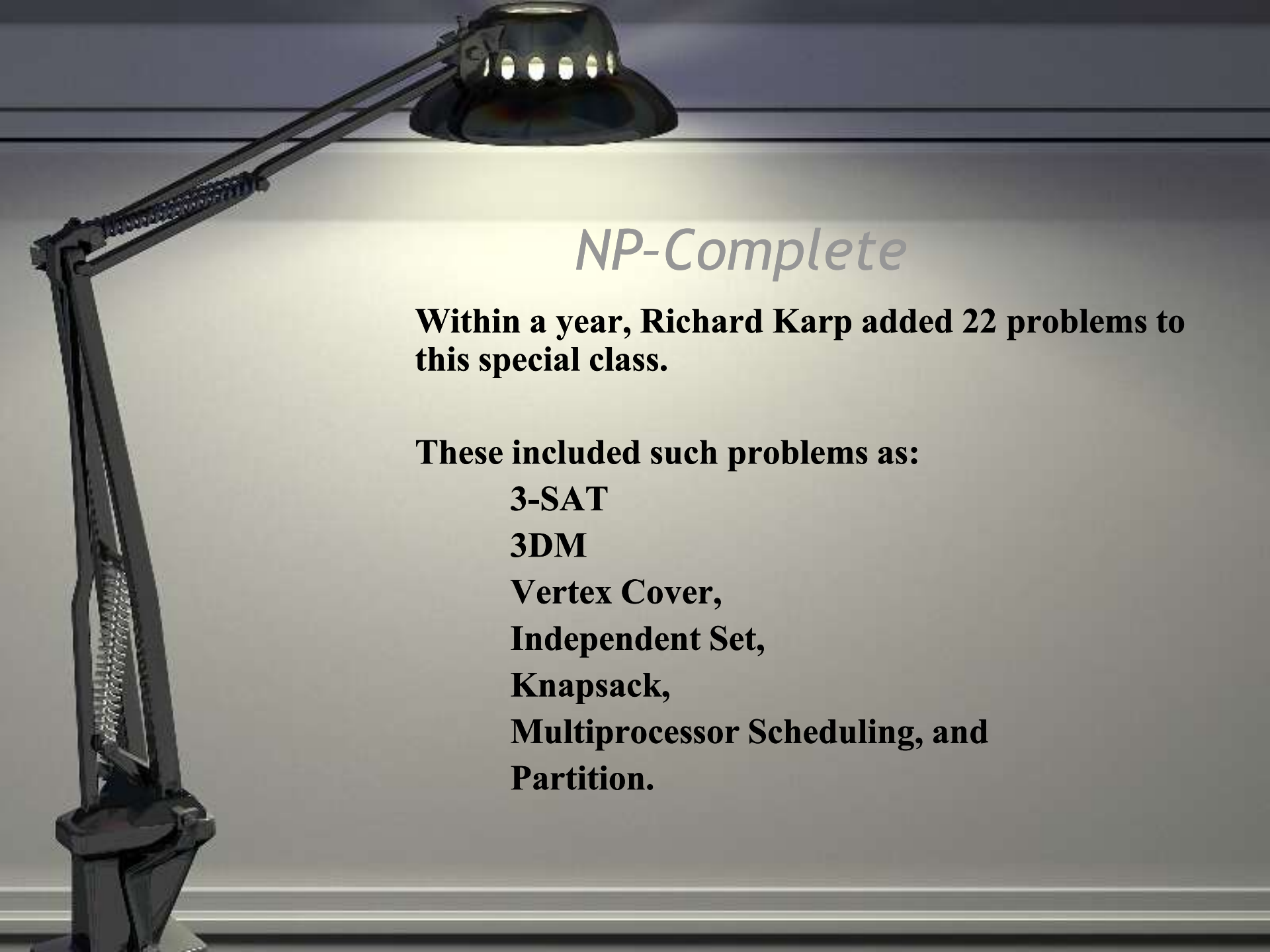


NP-Complete

Since SAT is itself in NP, that means SAT is a hardest problem in NP (there can be more than one.).

A hardest problem in a class is called the "completion" of that class.

Therefore, SAT is NP-Complete.

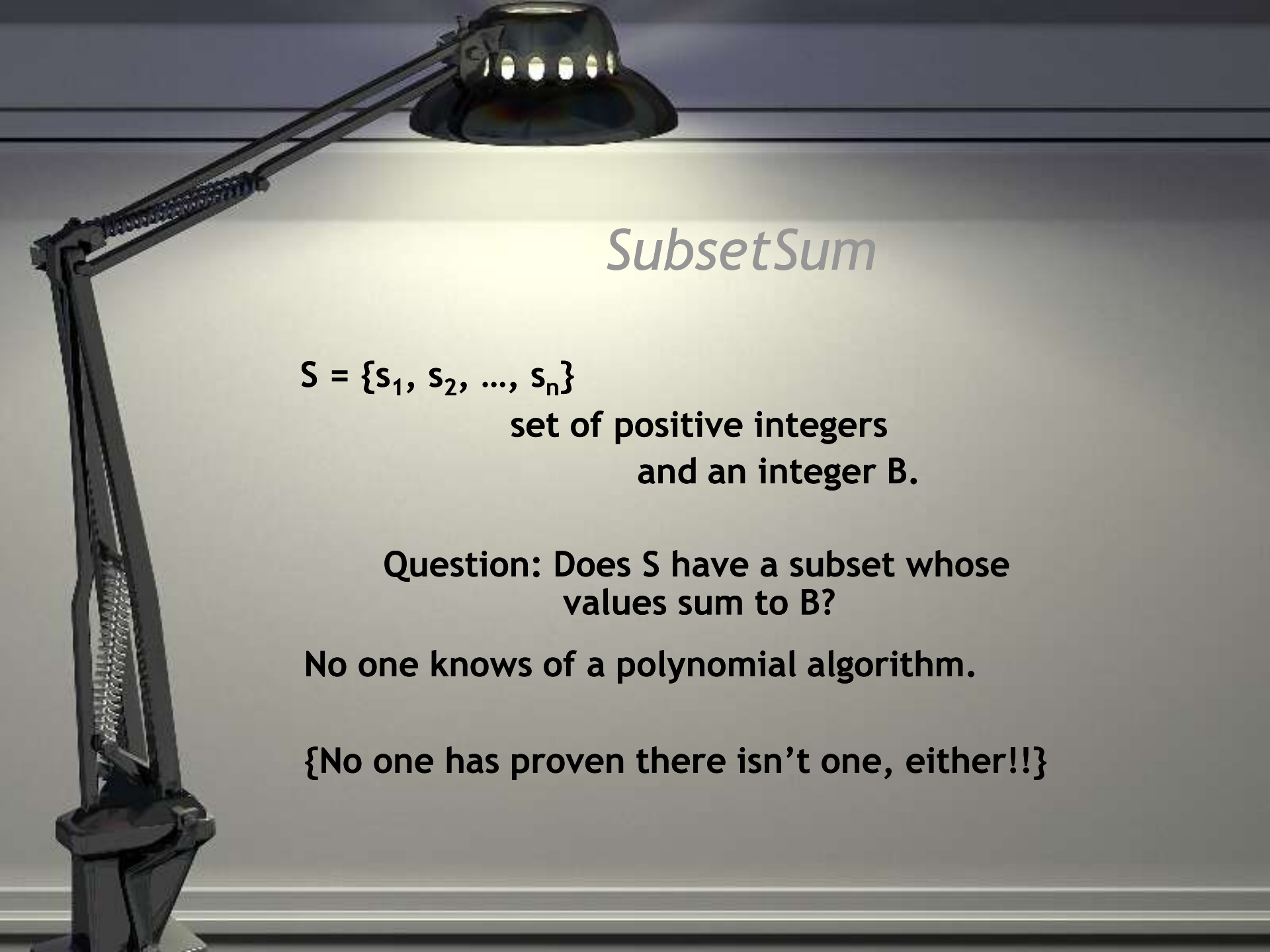


NP-Complete

Within a year, Richard Karp added 22 problems to this special class.

These included such problems as:

- **3-SAT**
- **3DM**
- **Vertex Cover,**
- **Independent Set,**
- **Knapsack,**
- **Multiprocessor Scheduling, and**
- **Partition.**



SubsetSum

$S = \{s_1, s_2, \ldots, s_n\}$

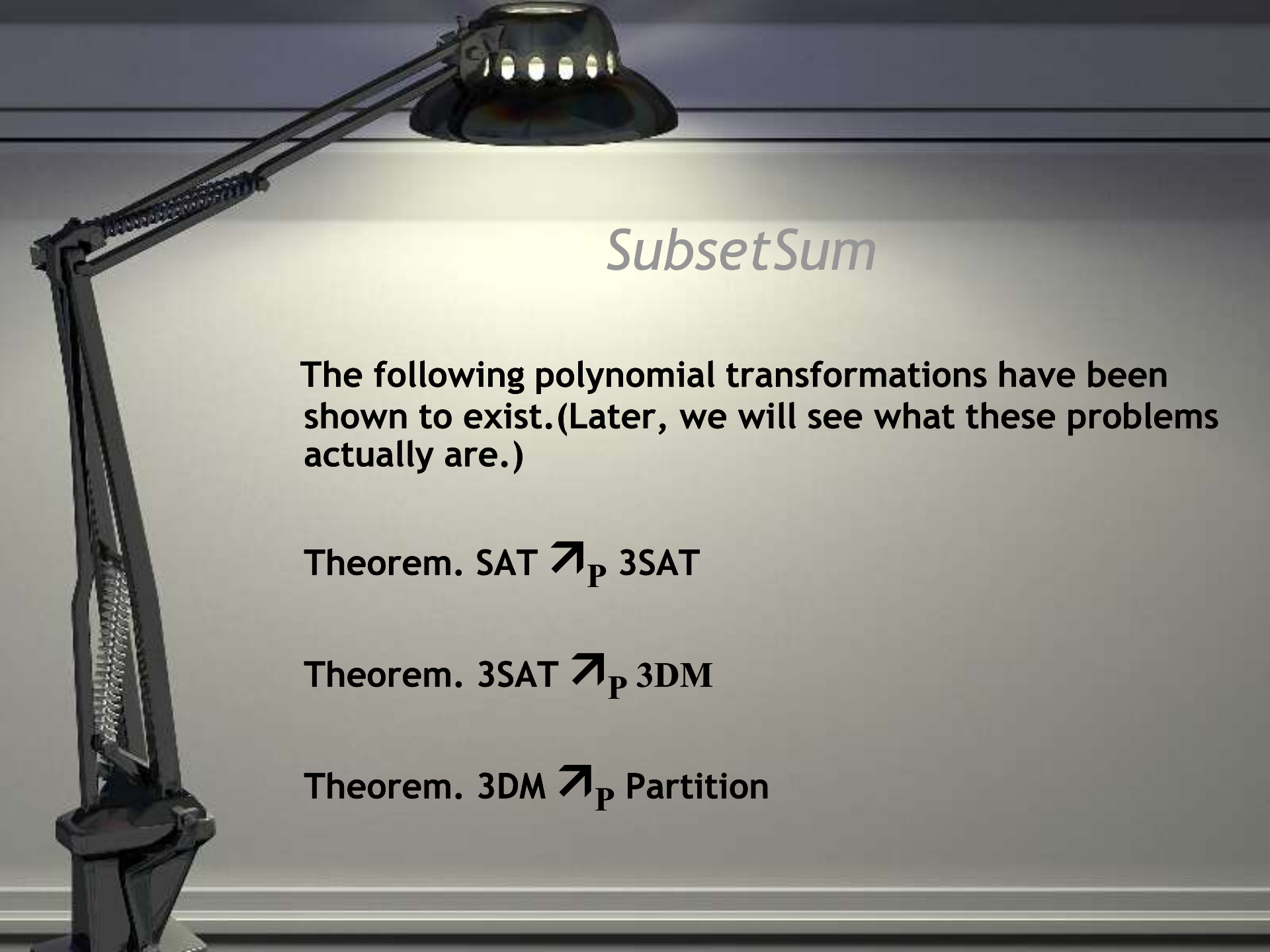
set of positive integers

and an integer B.

Question: Does S have a subset whose values sum to B?

No one knows of a polynomial algorithm.

{No one has proven there isn't one, either!!}



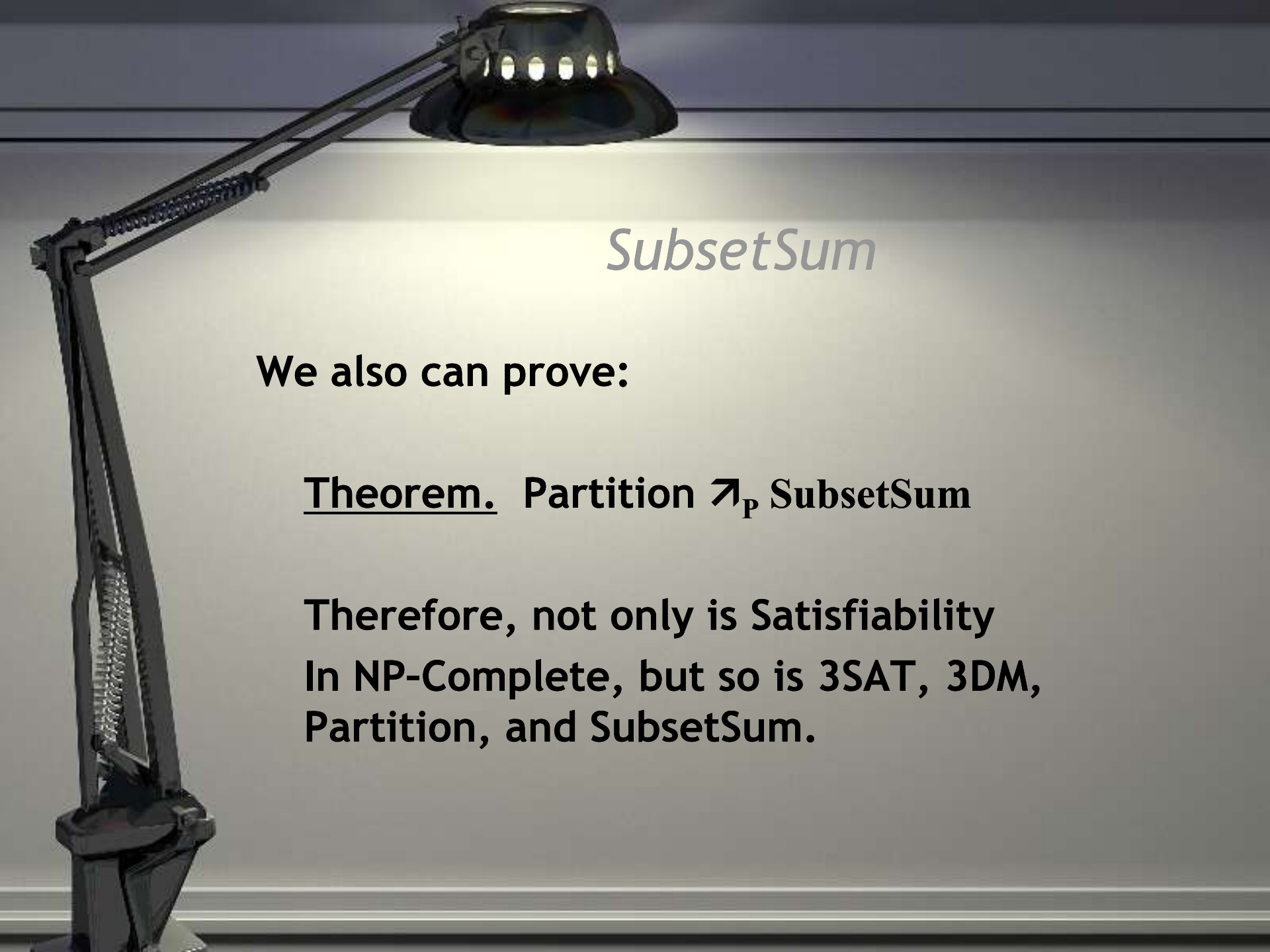
SubsetSum

The following polynomial transformations have been shown to exist.(Later, we will see what these problems actually are.)

Theorem. SAT $\nearrow_P$ 3SAT

Theorem. 3SAT $\nearrow_P$ 3DM

Theorem. 3DM $\nearrow_P$ Partition

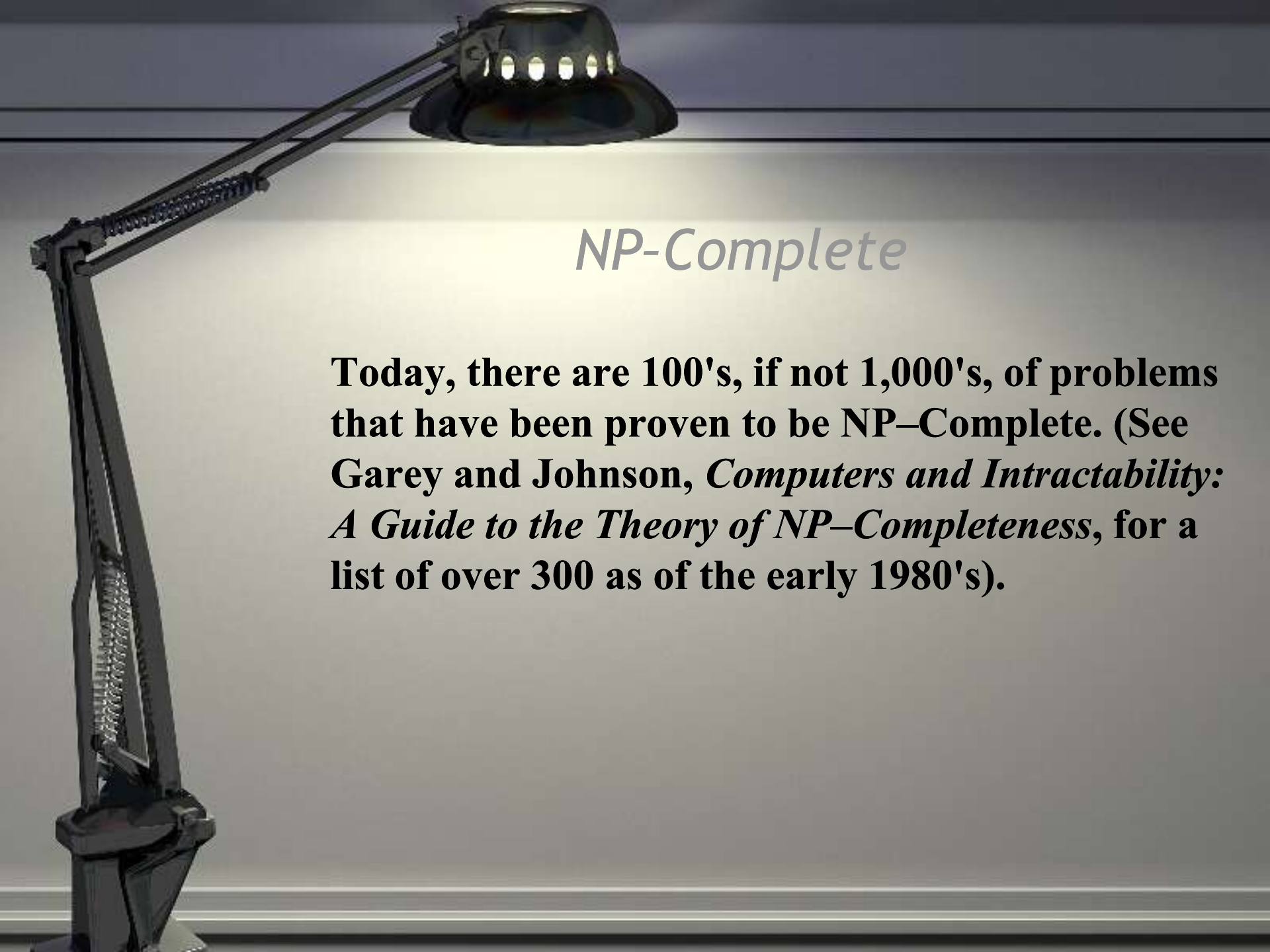


SubsetSum

We also can prove:

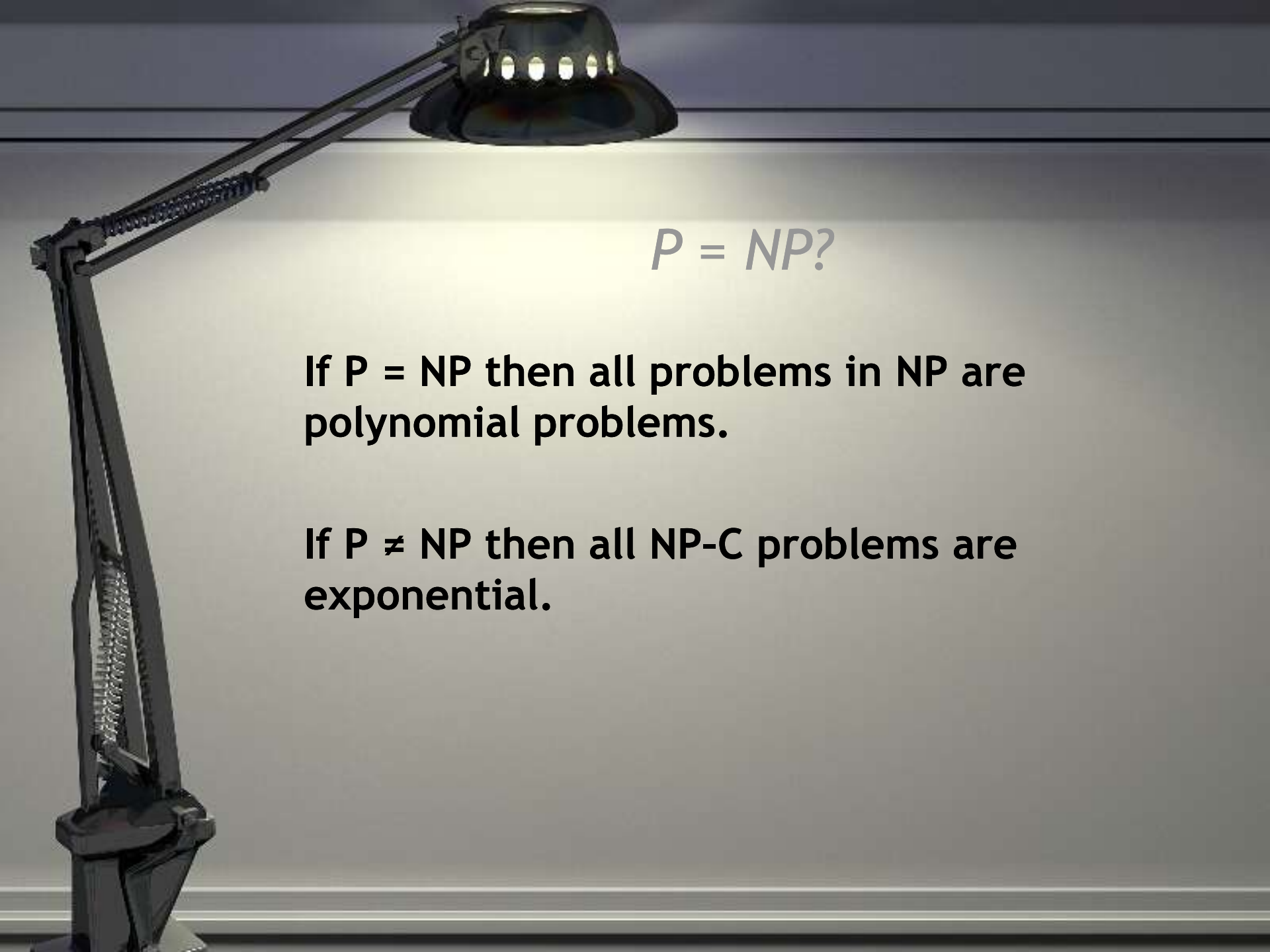
Theorem. **Partition $\nearrow_P$ SubsetSum**

Therefore, not only is Satisfiability In NP-Complete, but so is 3SAT, 3DM, Partition, and SubsetSum.



NP-Complete

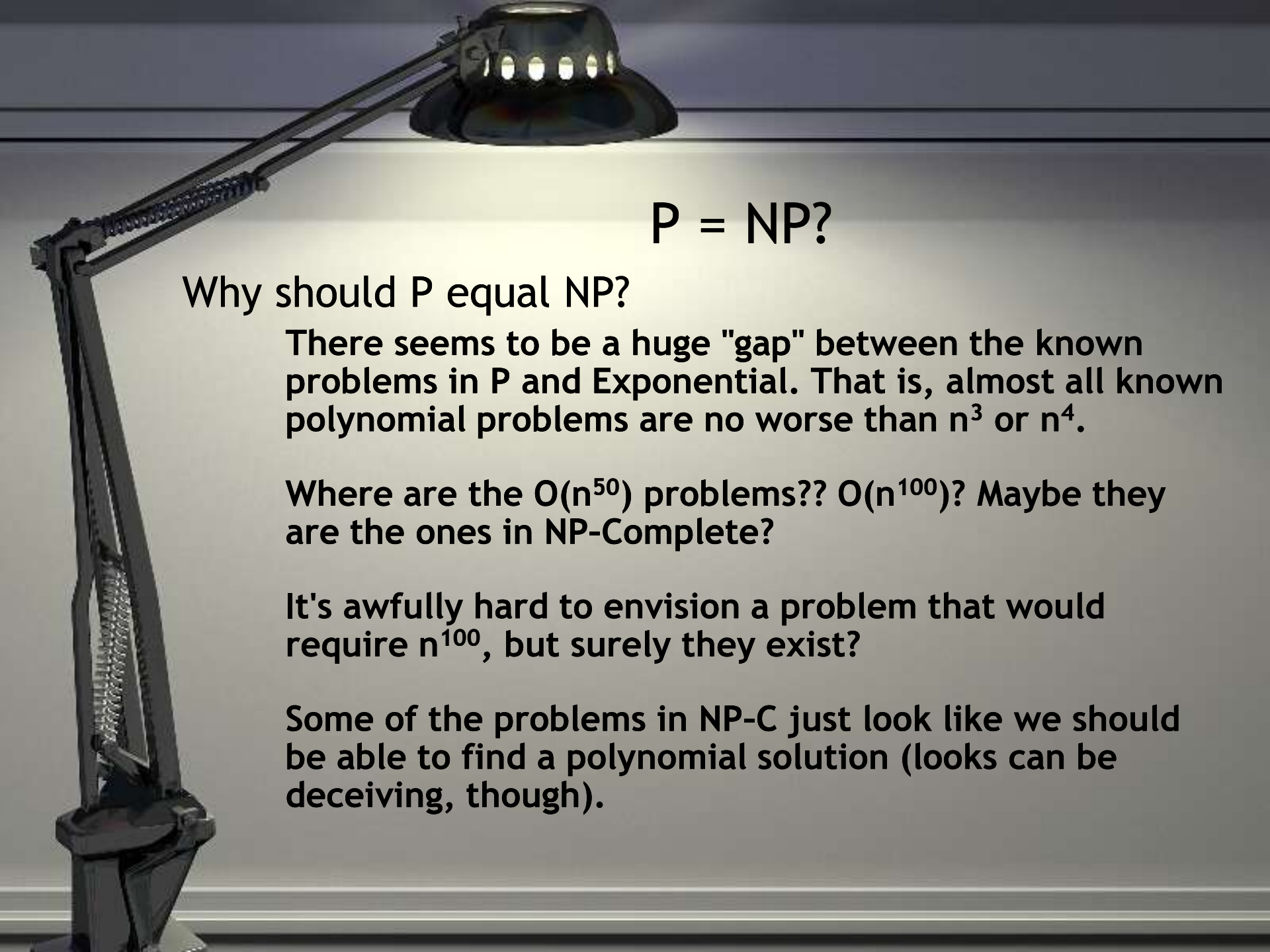
Today, there are 100's, if not 1,000's, of problems that have been proven to be NP–Complete. (See Garey and Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness*, for a list of over 300 as of the early 1980's).



P = NP?

If P = NP then all problems in NP are polynomial problems.

If P ≠ NP then all NP-C problems are exponential.



P = NP?

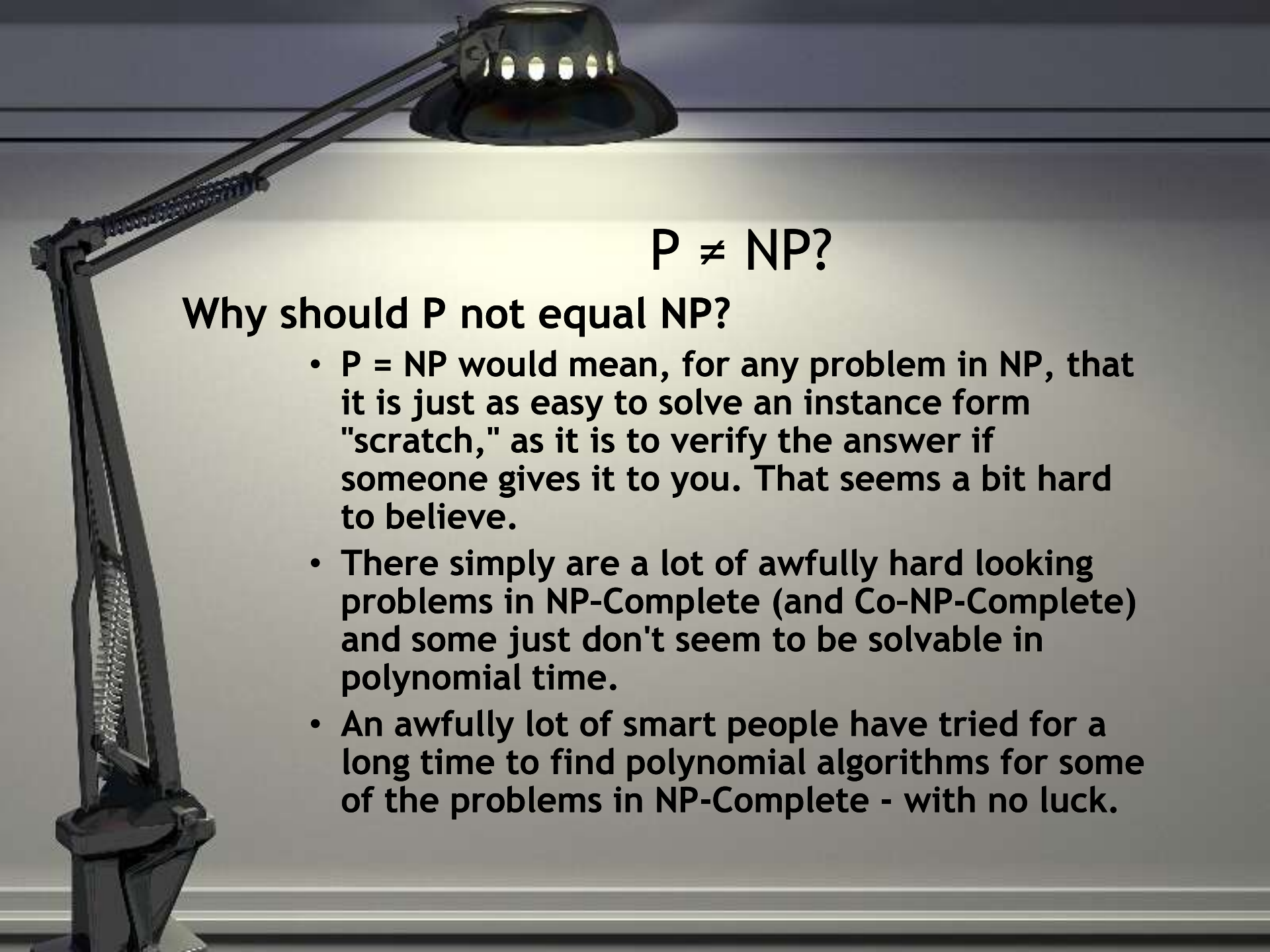
Why should P equal NP?

There seems to be a huge "gap" between the known problems in P and Exponential. That is, almost all known polynomial problems are no worse than n^3 or n^4.

Where are the $O(n^{50})$ problems?? $O(n^{100})$? Maybe they are the ones in NP-Complete?

It's awfully hard to envision a problem that would require n^{100}, but surely they exist?

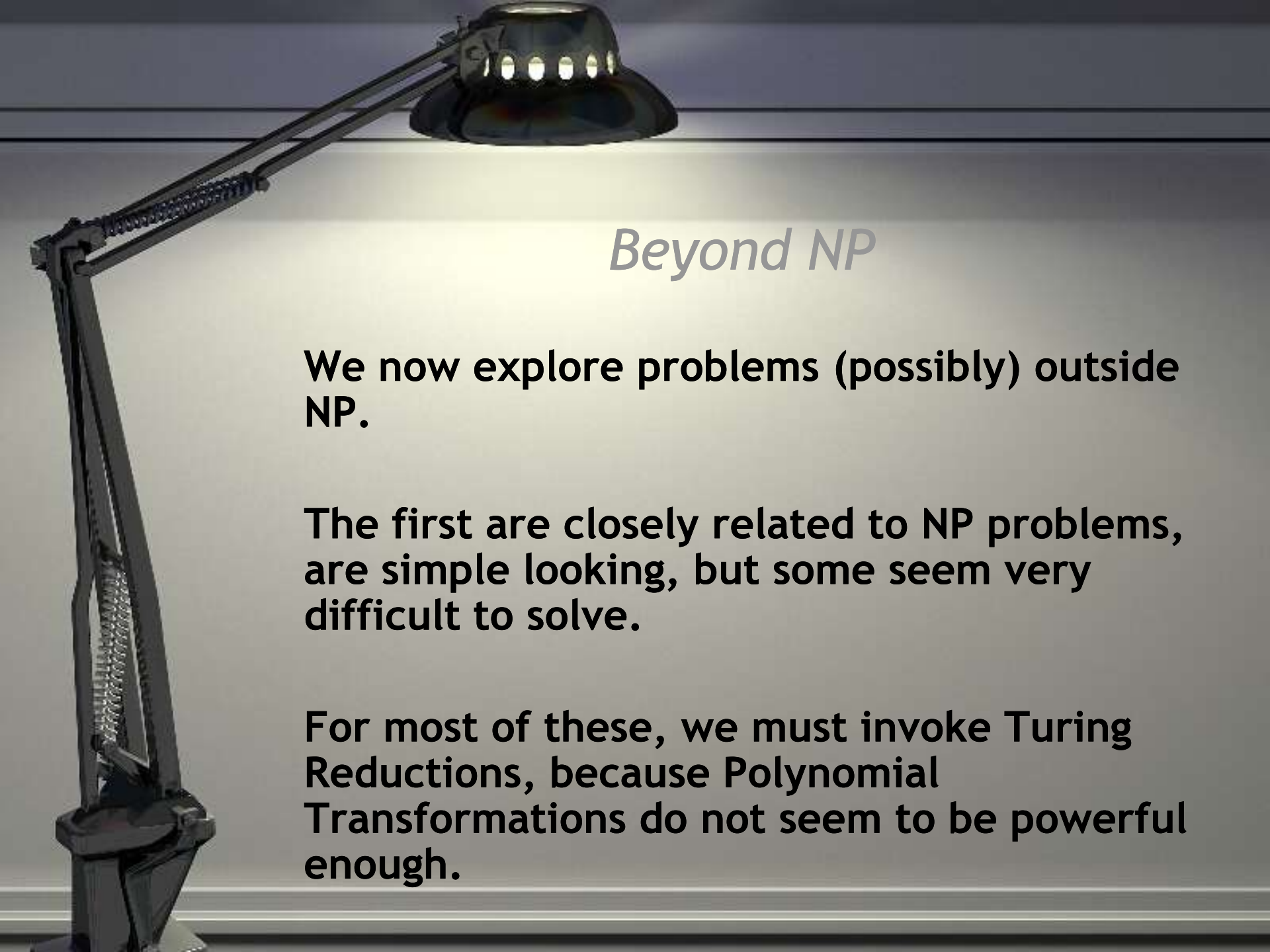
Some of the problems in NP-C just look like we should be able to find a polynomial solution (looks can be deceiving, though).



P ≠ NP?

Why should P not equal NP?

- P = NP would mean, for any problem in NP, that it is just as easy to solve an instance form "scratch," as it is to verify the answer if someone gives it to you. That seems a bit hard to believe.
- There simply are a lot of awfully hard looking problems in NP-Complete (and Co-NP-Complete) and some just don't seem to be solvable in polynomial time.
- An awfully lot of smart people have tried for a long time to find polynomial algorithms for some of the problems in NP-Complete - with no luck.

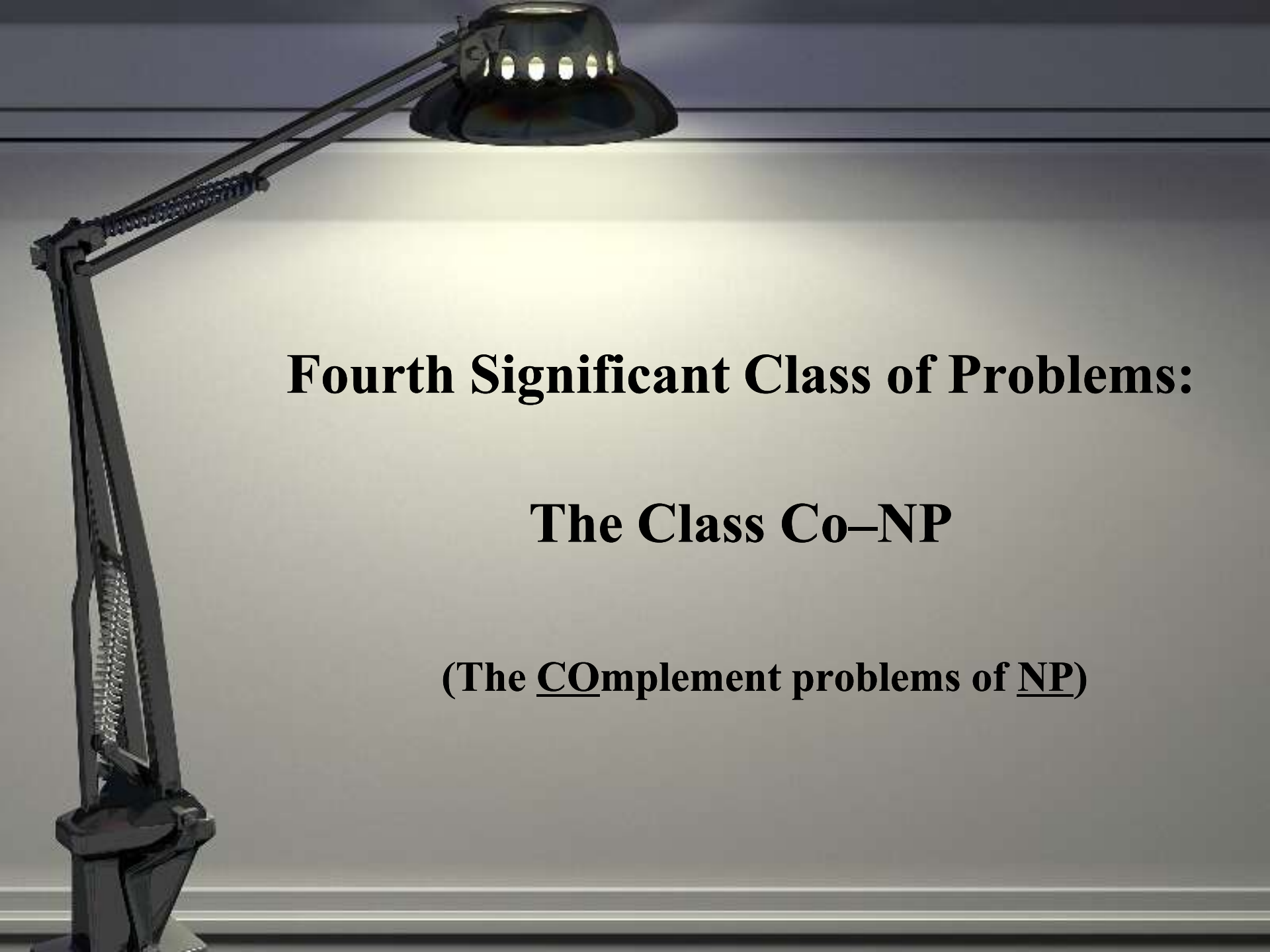


Beyond NP

We now explore problems (possibly) outside NP.

The first are closely related to NP problems, are simple looking, but some seem very difficult to solve.

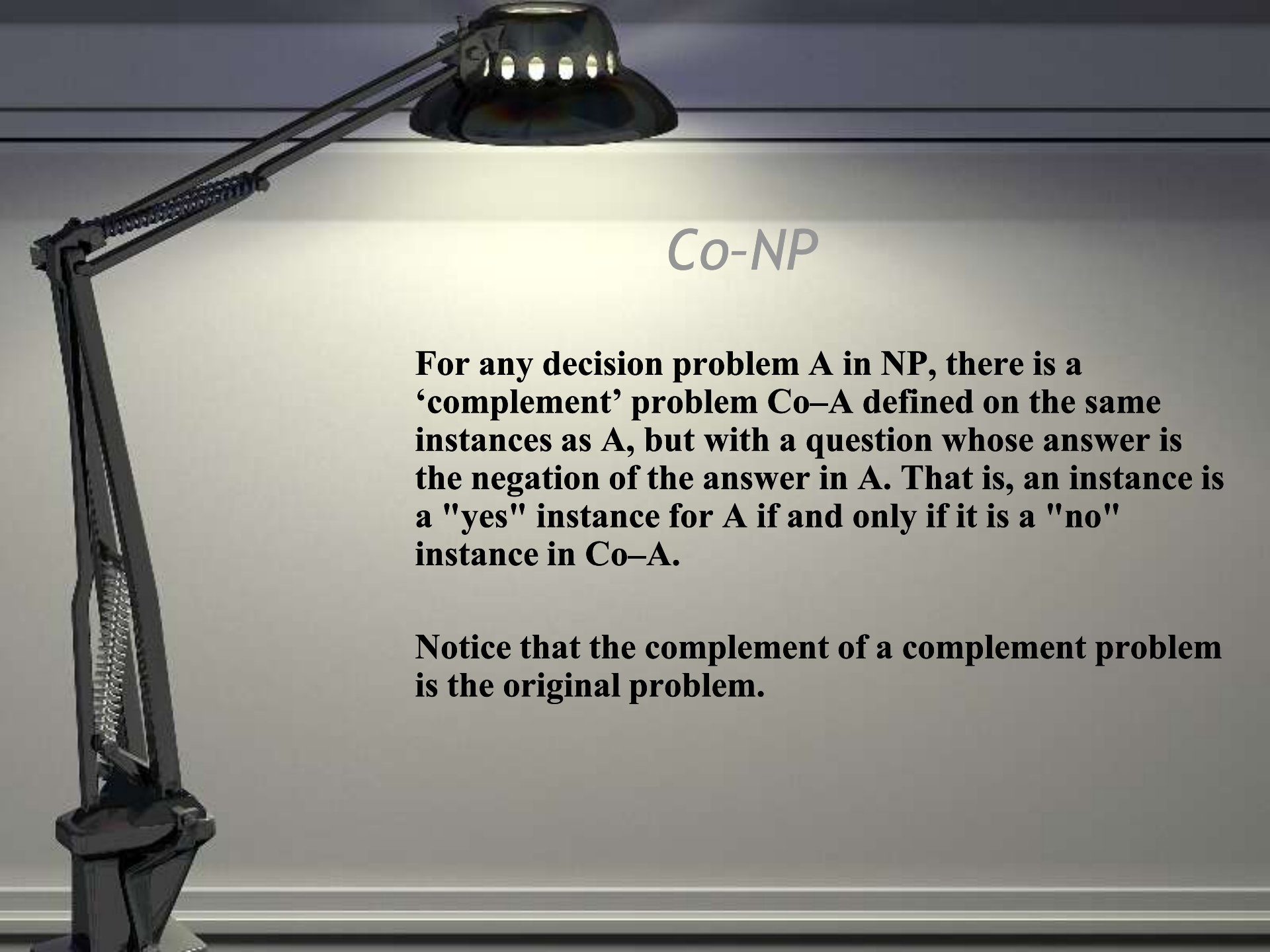
For most of these, we must invoke Turing Reductions, because Polynomial Transformations do not seem to be powerful enough.



Fourth Significant Class of Problems:

The Class Co–NP

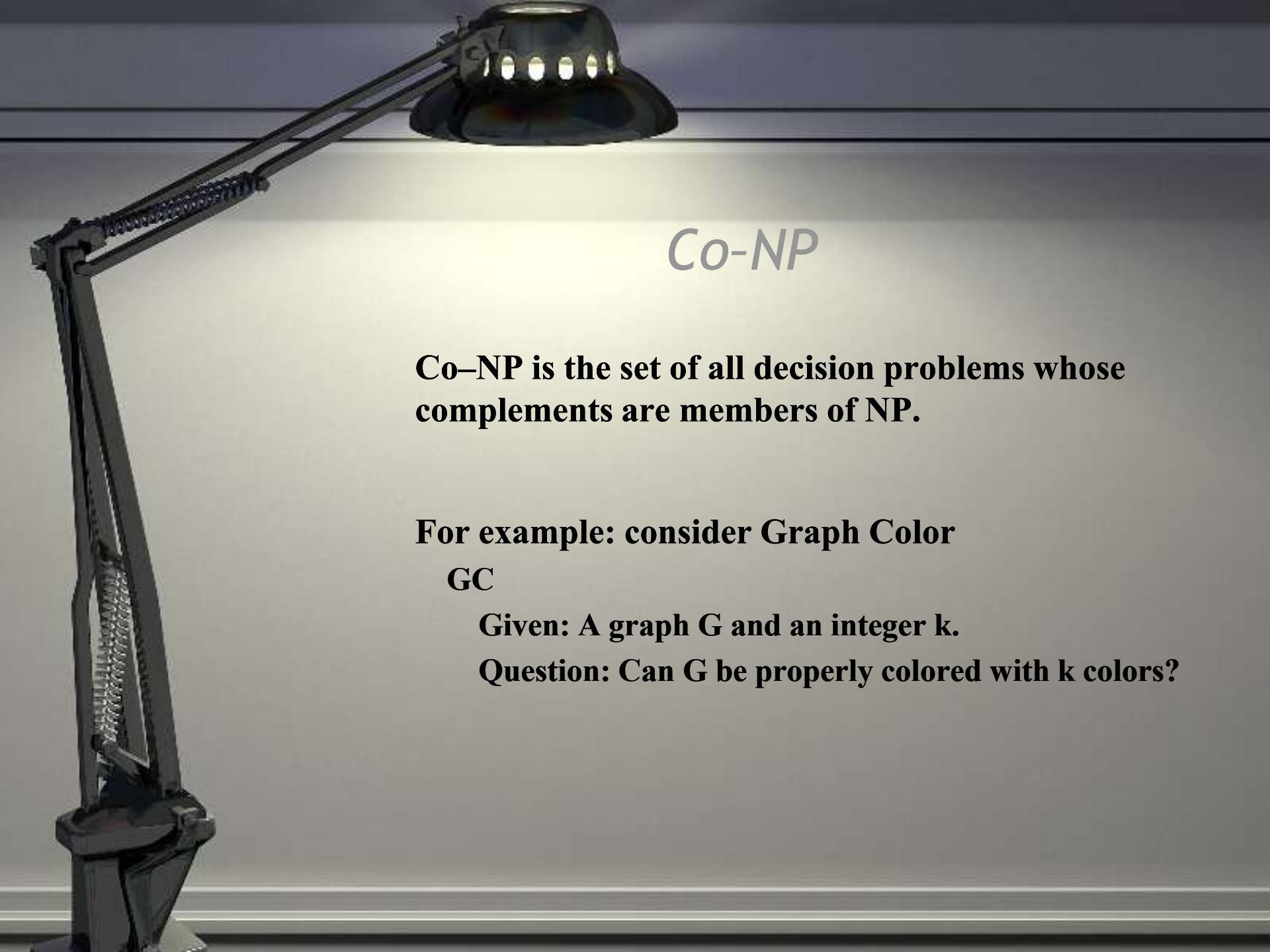
(The COmplement problems of **NP)**



Co-NP

For any decision problem A in NP, there is a ‘complement’ problem Co–A defined on the same instances as A, but with a question whose answer is the negation of the answer in A. That is, an instance is a "yes" instance for A if and only if it is a "no" instance in Co–A.

Notice that the complement of a complement problem is the original problem.



Co-NP

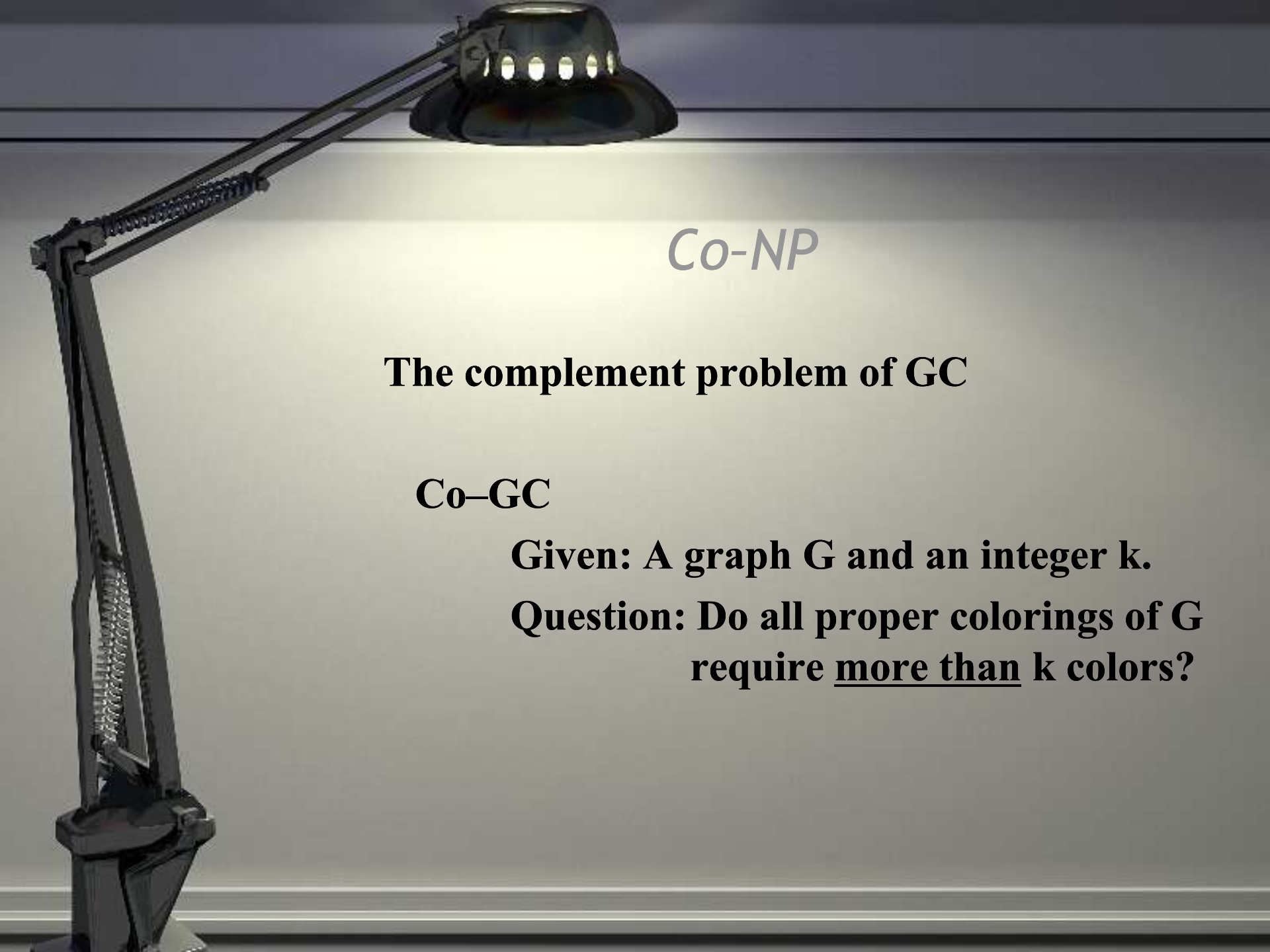
Co–NP is the set of all decision problems whose complements are members of NP.

For example: consider Graph Color

GC

Given: A graph G and an integer k.

Question: Can G be properly colored with k colors?



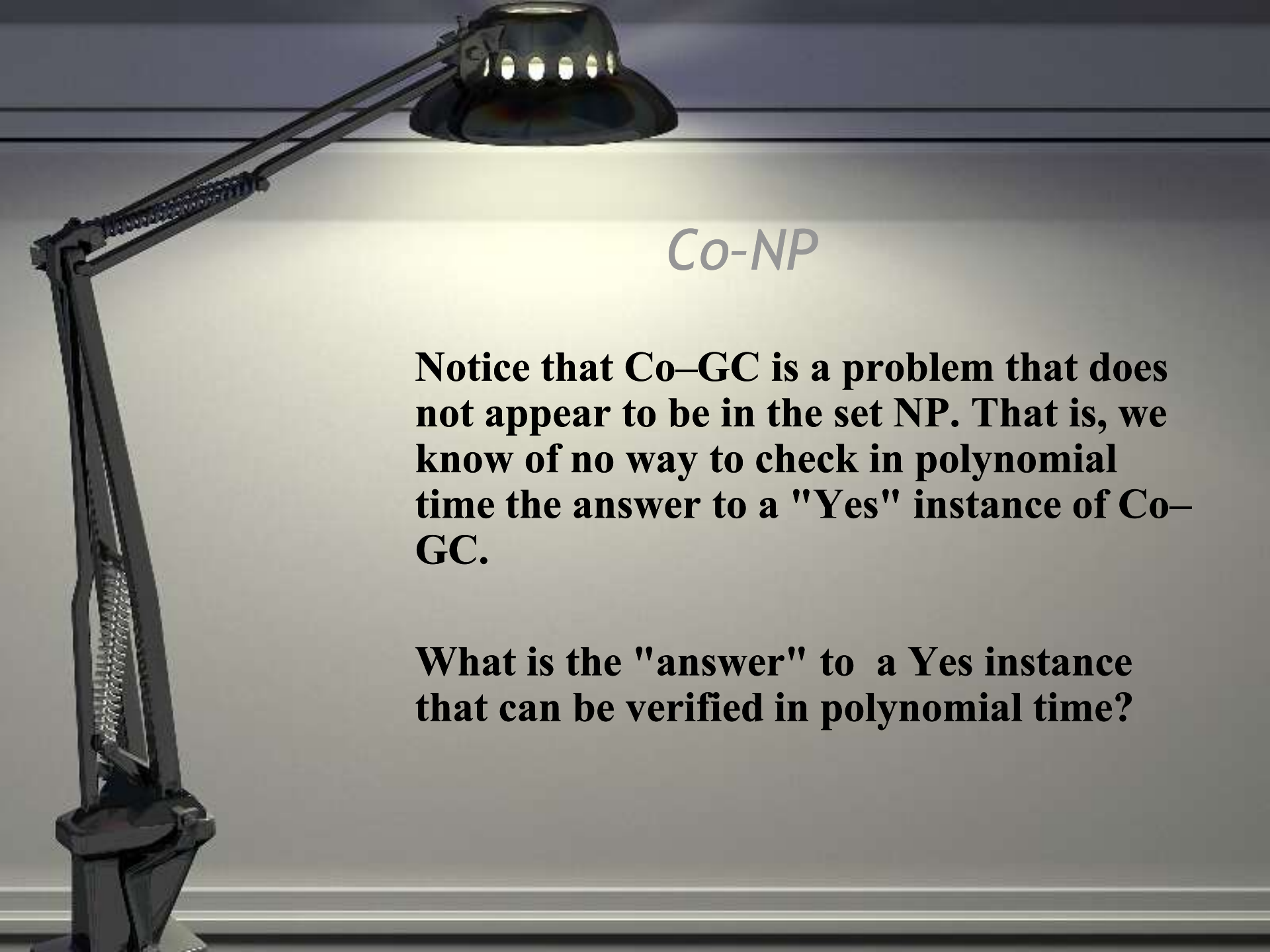
Co-NP

The complement problem of GC

Co–GC

Given: A graph G and an integer k.

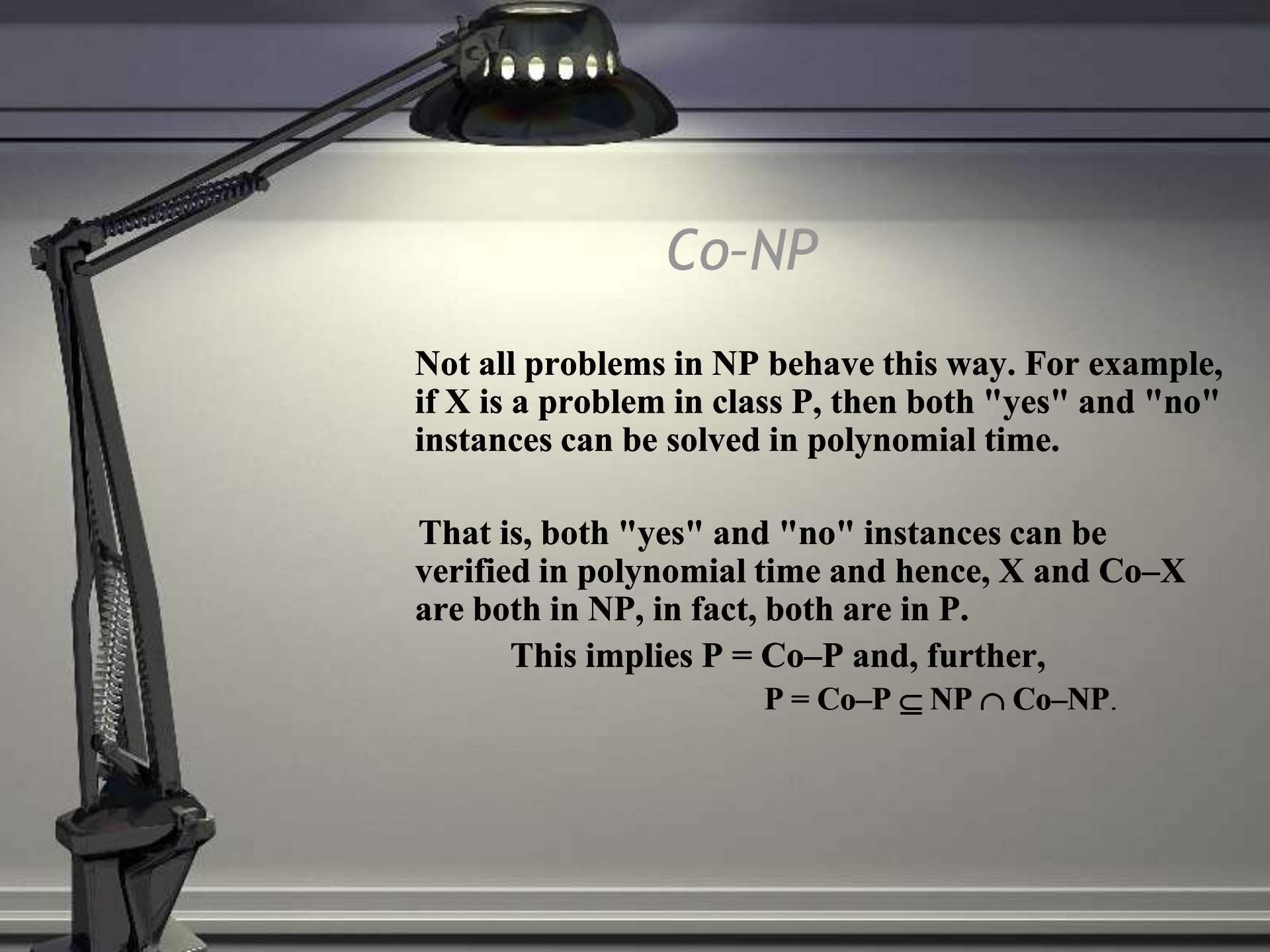
Question: Do all proper colorings of G require <u>more than</u> k colors?



Co-NP

Notice that Co–GC is a problem that does not appear to be in the set NP. That is, we know of no way to check in polynomial time the answer to a "Yes" instance of Co–GC.

What is the "answer" to a Yes instance that can be verified in polynomial time?



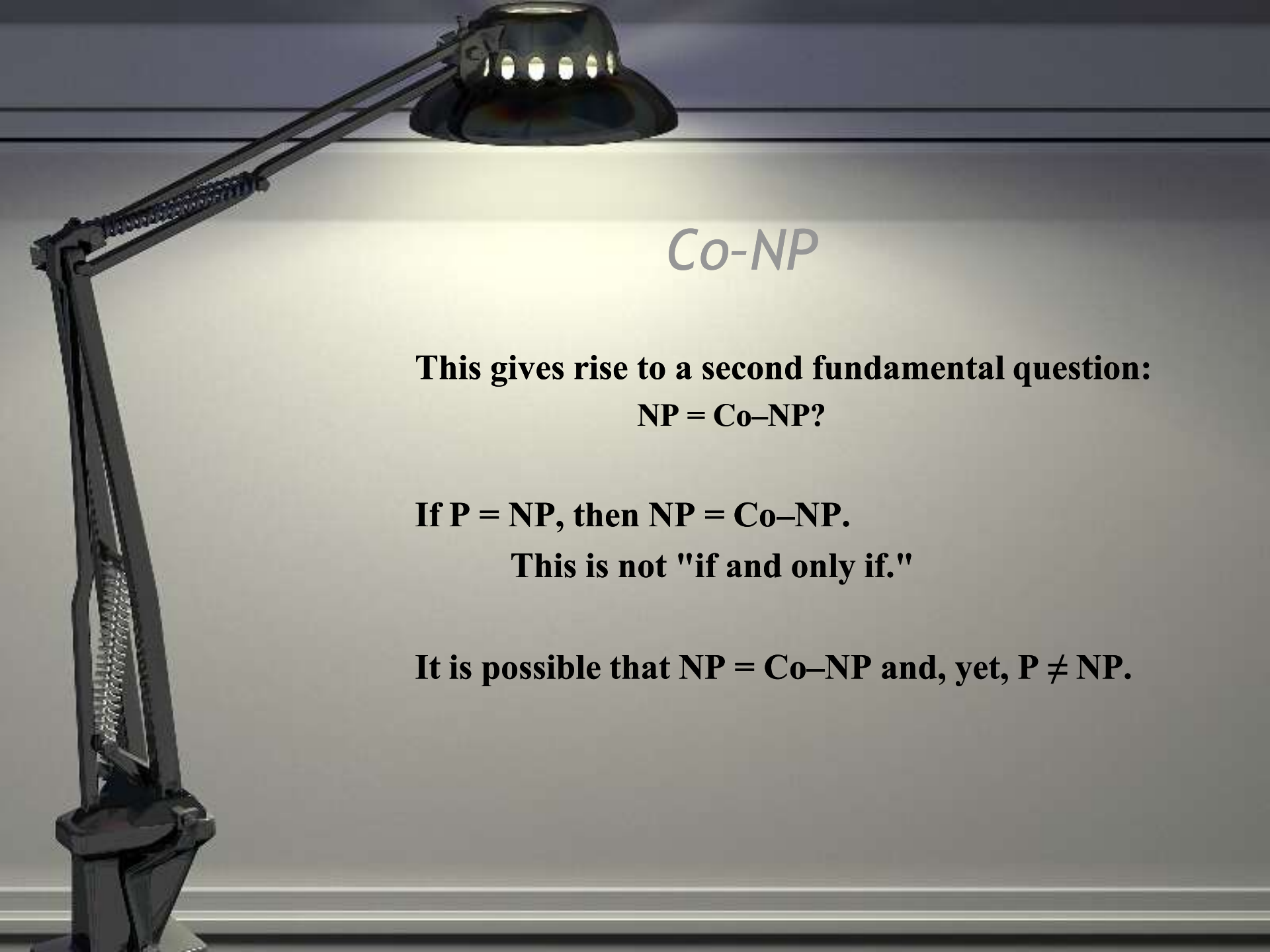
Co-NP

Not all problems in NP behave this way. For example, if X is a problem in class P, then both "yes" and "no" instances can be solved in polynomial time.

That is, both "yes" and "no" instances can be verified in polynomial time and hence, X and Co–X are both in NP, in fact, both are in P.

This implies P = Co–P and, further,

$$P = \text{Co–}P \subseteq NP \cap \text{Co–}NP.$$



Co-NP

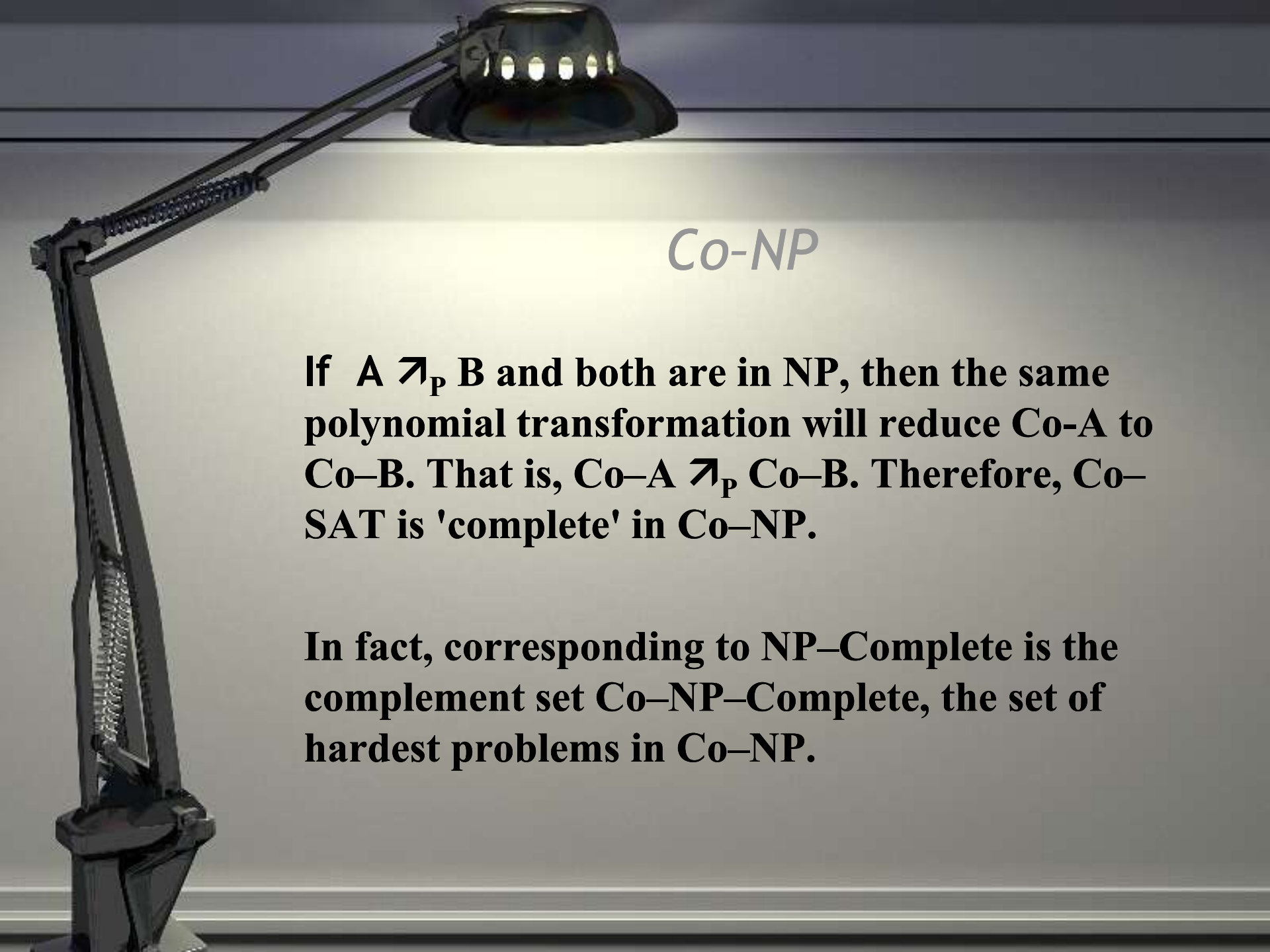
This gives rise to a second fundamental question:

NP = Co–NP?

If P = NP, then NP = Co–NP.

This is not "if and only if."

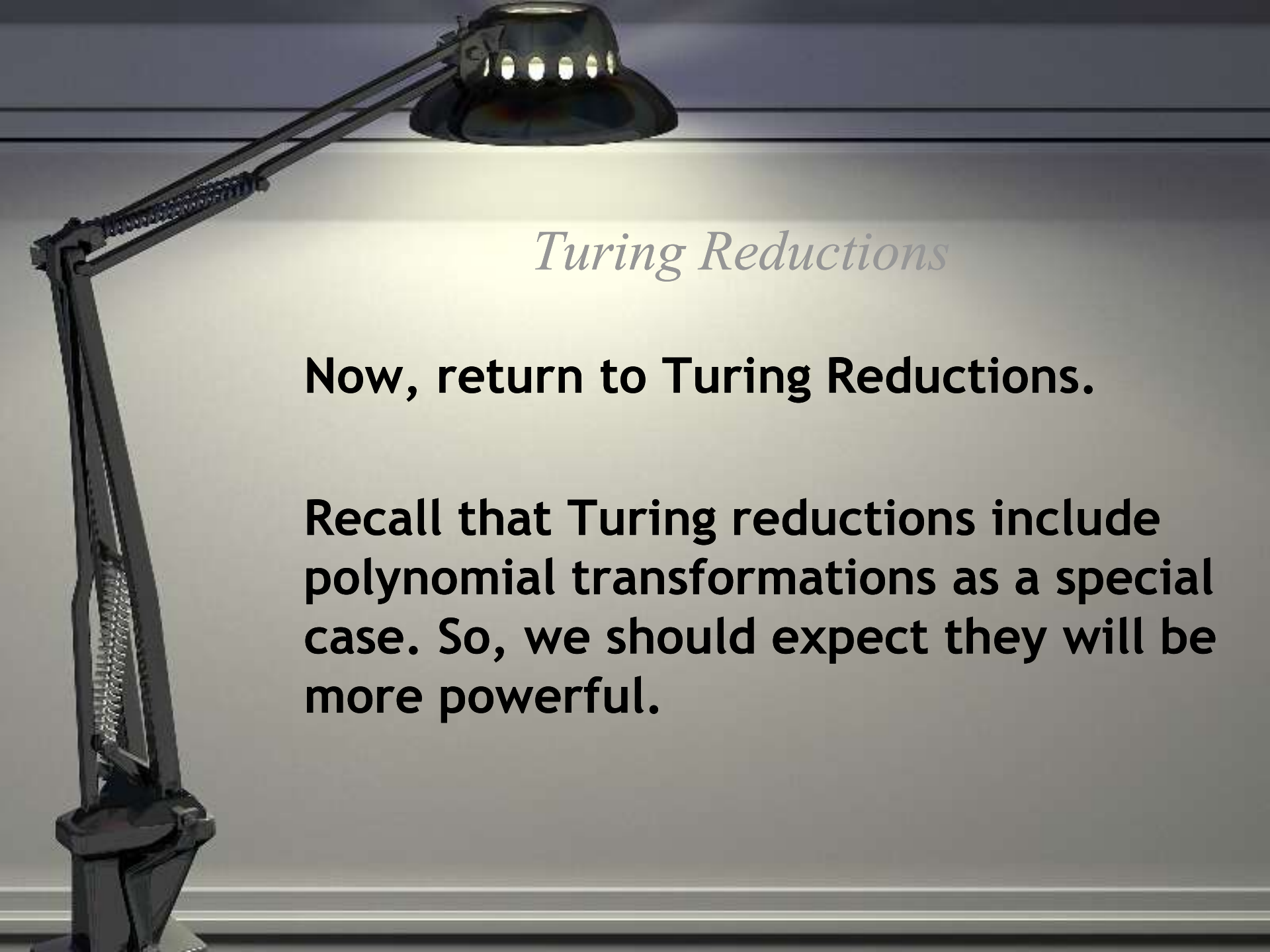
It is possible that NP = Co–NP and, yet, P ≠ NP.



Co-NP

If A $\nearrow_P$ B and both are in NP, then the same polynomial transformation will reduce Co-A to Co–B. That is, Co–A $\nearrow_P$ Co–B. Therefore, Co–SAT is 'complete' in Co–NP.

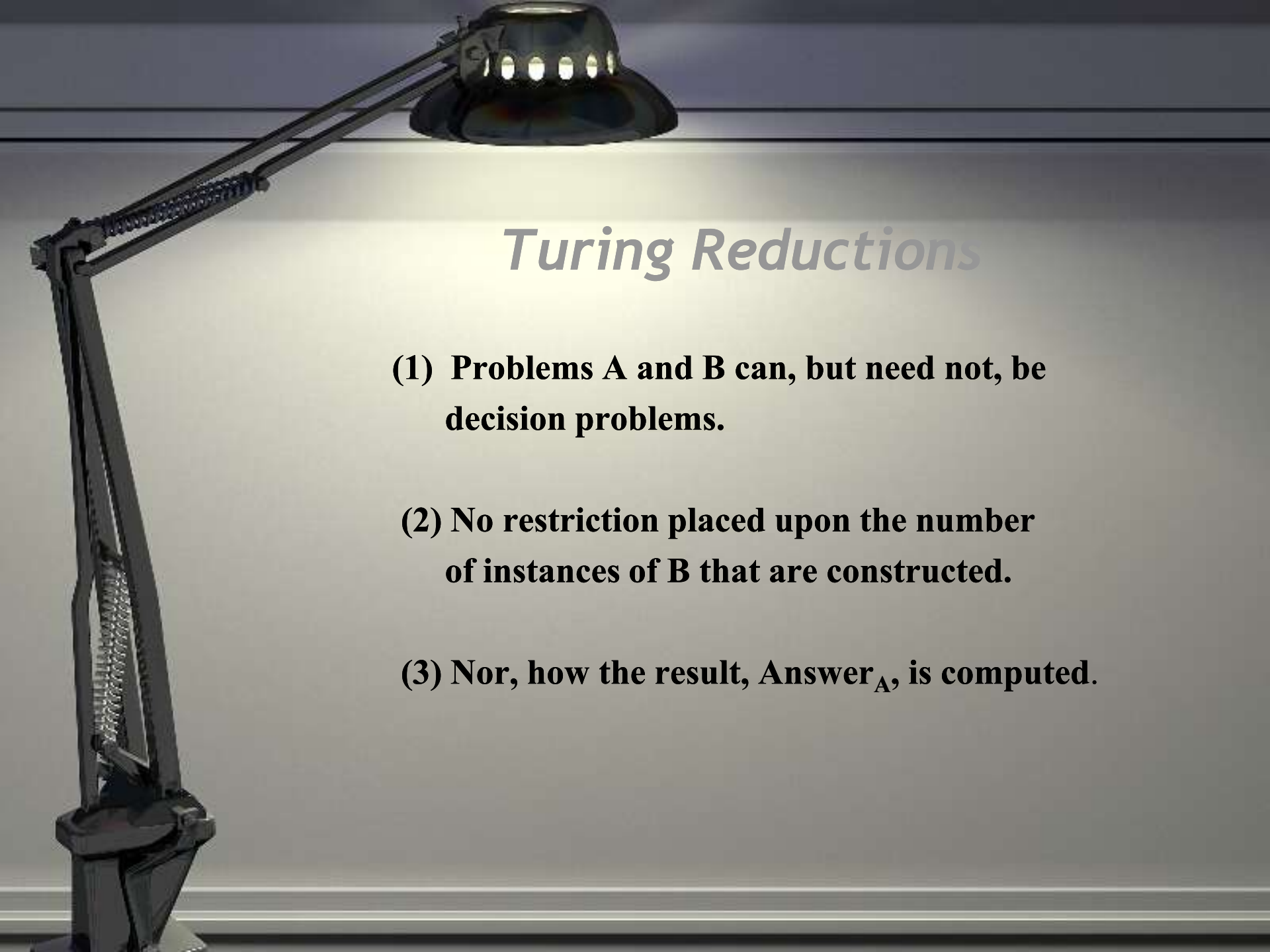
In fact, corresponding to NP–Complete is the complement set Co–NP–Complete, the set of hardest problems in Co–NP.



Turing Reductions

Now, return to Turing Reductions.

Recall that Turing reductions include polynomial transformations as a special case. So, we should expect they will be more powerful.

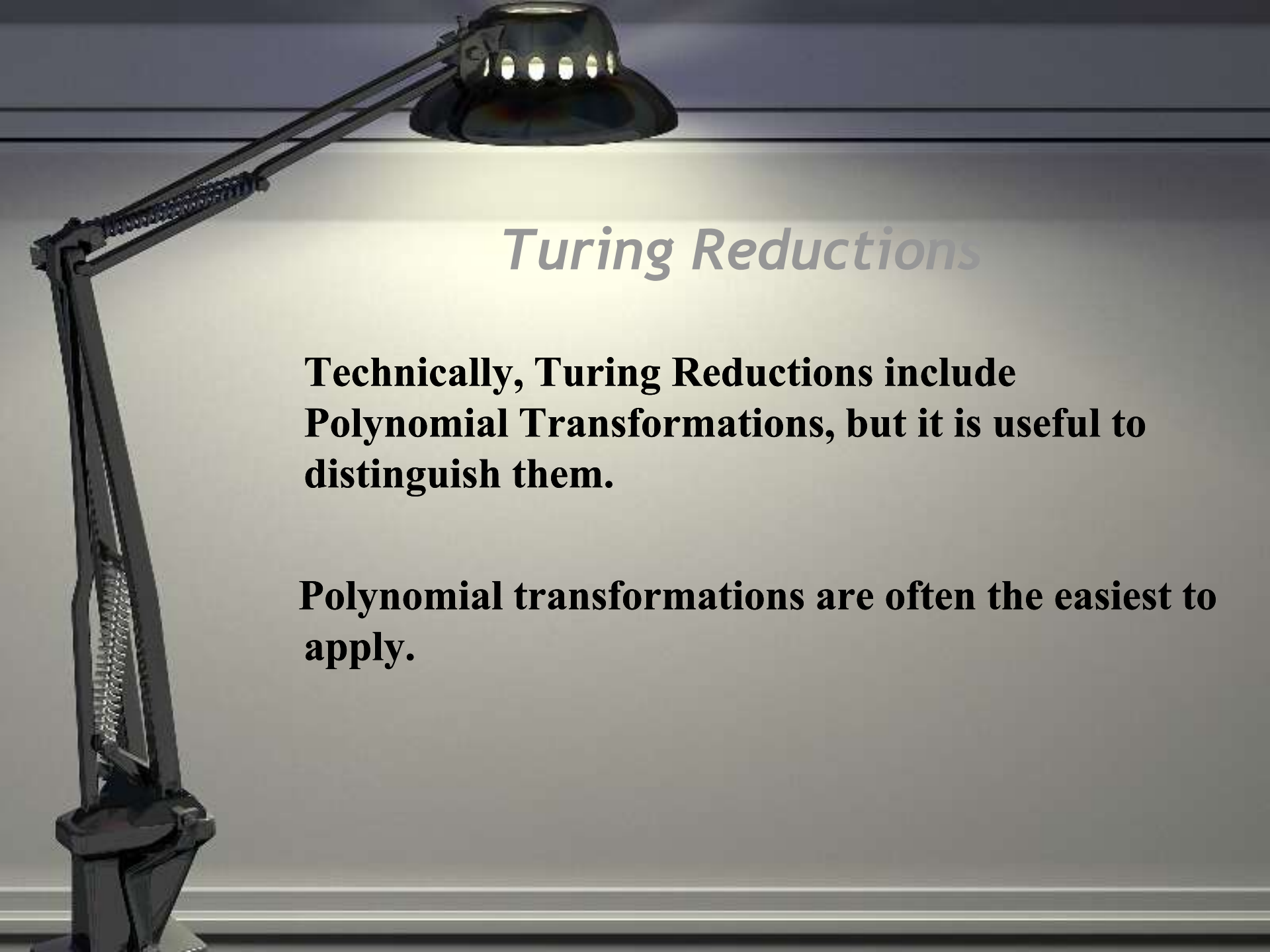


Turing Reductions

(1) Problems A and B can, but need not, be decision problems.

(2) No restriction placed upon the number of instances of B that are constructed.

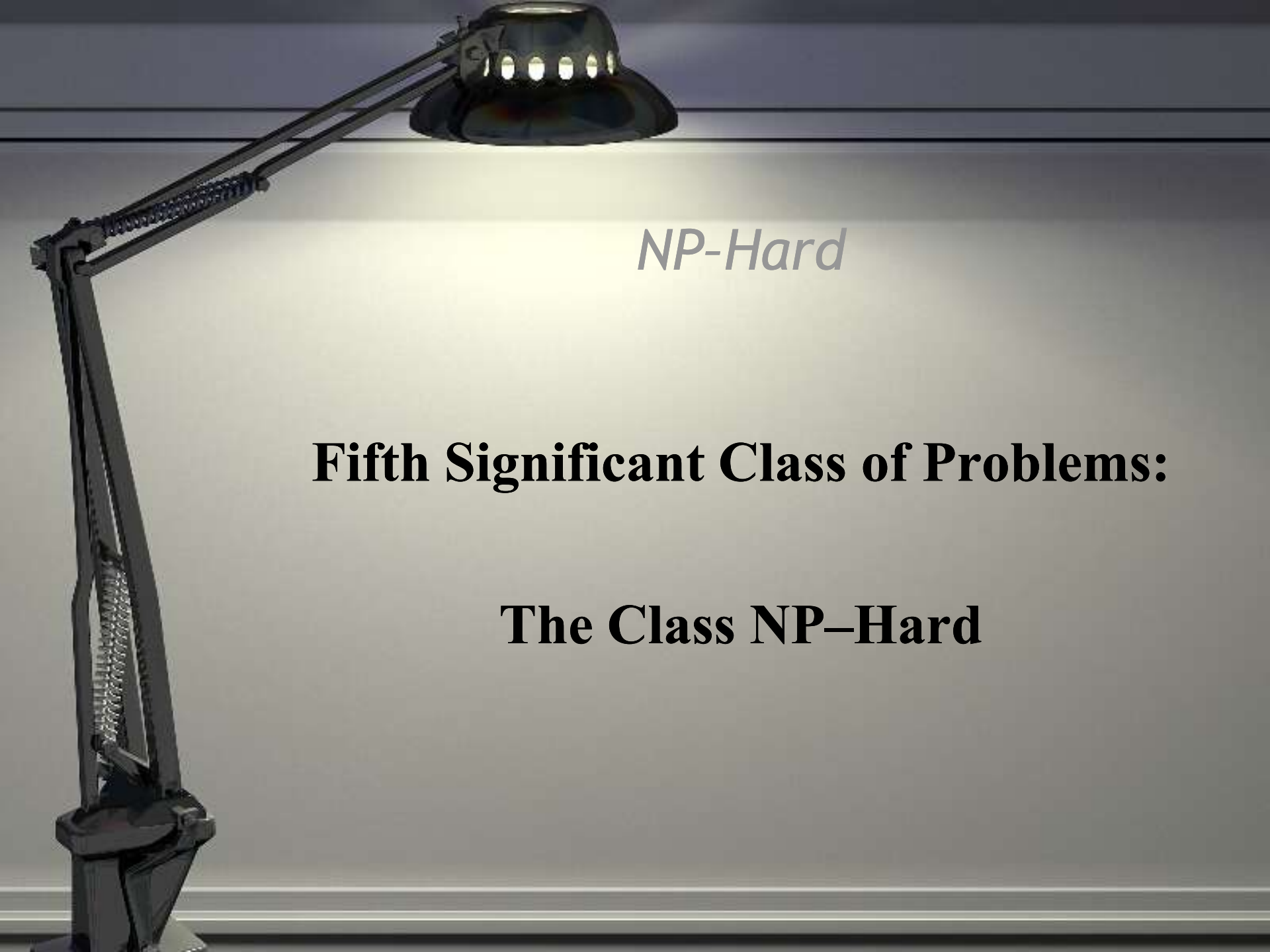
(3) Nor, how the result, Answer_A, is computed.

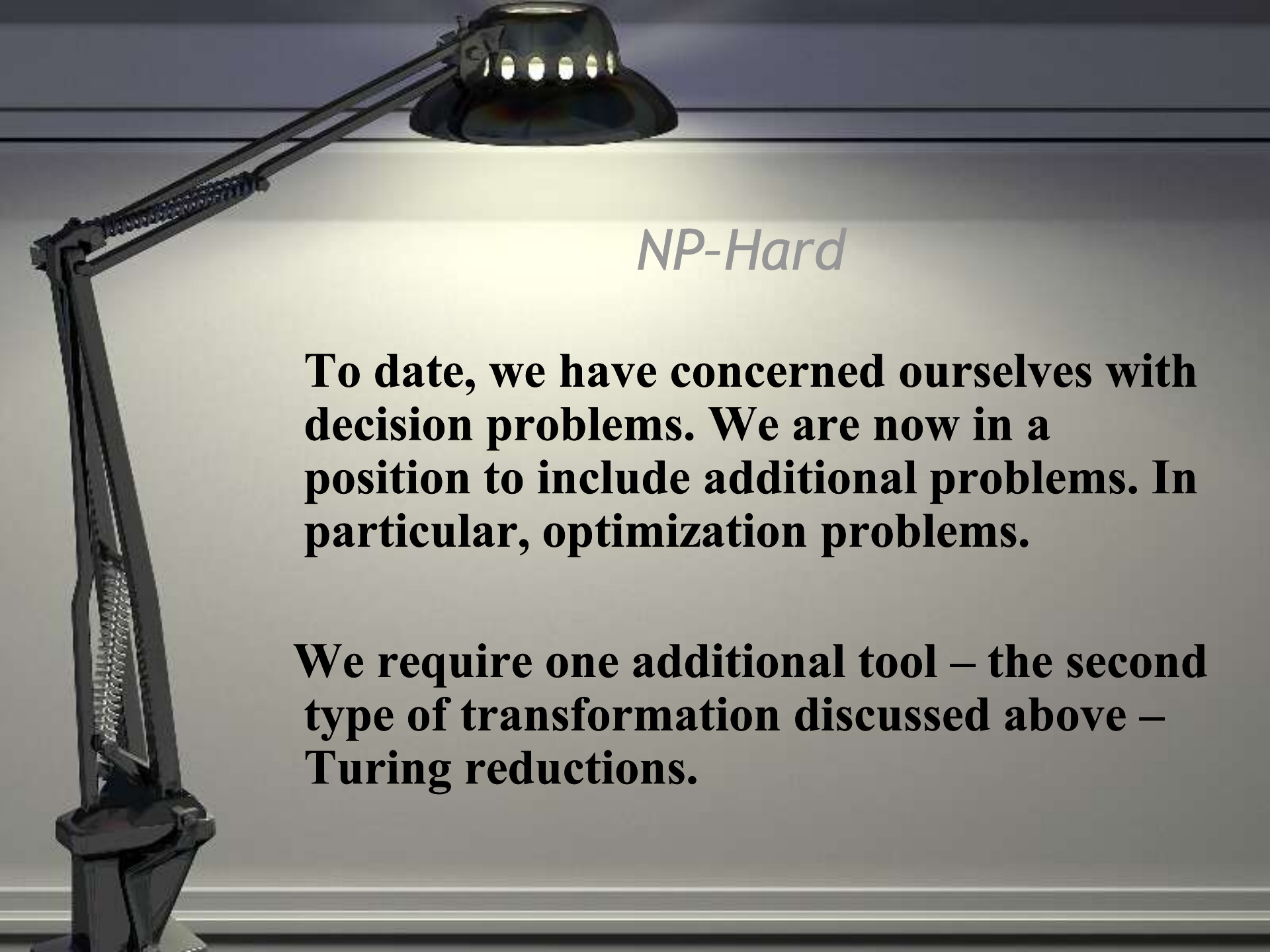


Turing Reductions

Technically, Turing Reductions include Polynomial Transformations, but it is useful to distinguish them.

Polynomial transformations are often the easiest to apply.

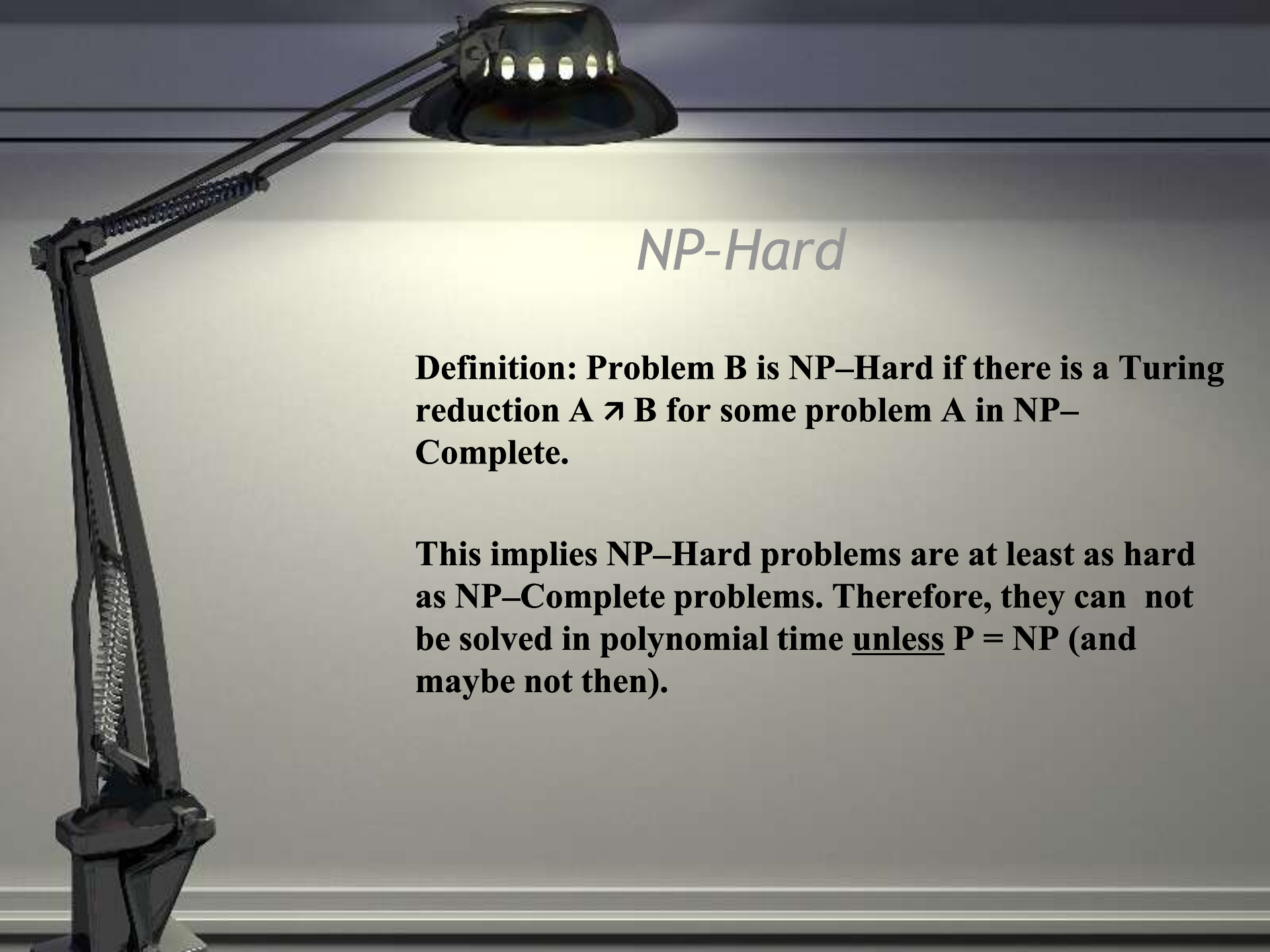

NP-Hard
Fifth Significant Class of Problems:
The Class NP–Hard



NP-Hard

To date, we have concerned ourselves with decision problems. We are now in a position to include additional problems. In particular, optimization problems.

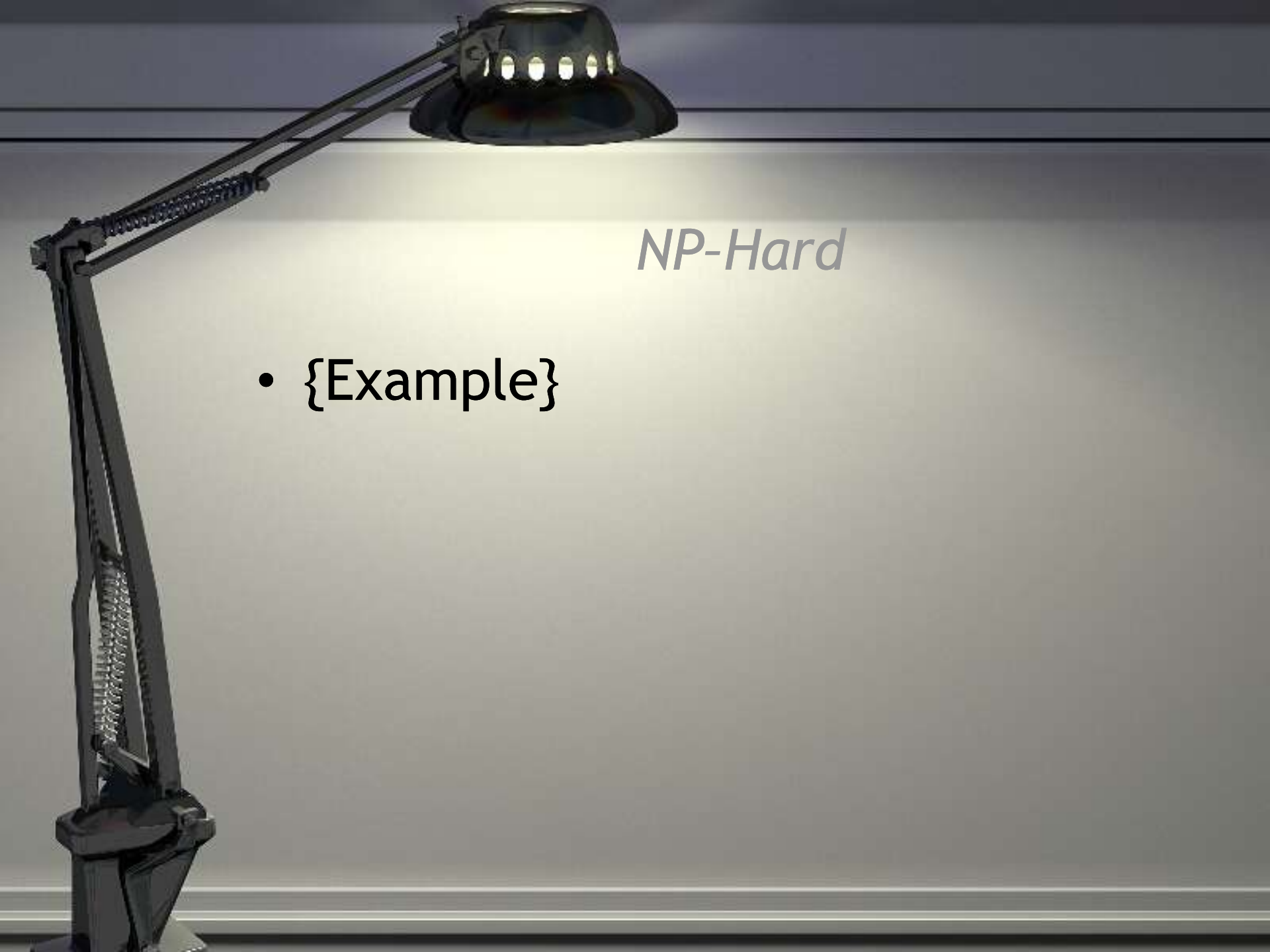
We require one additional tool – the second type of transformation discussed above – Turing reductions.



NP-Hard

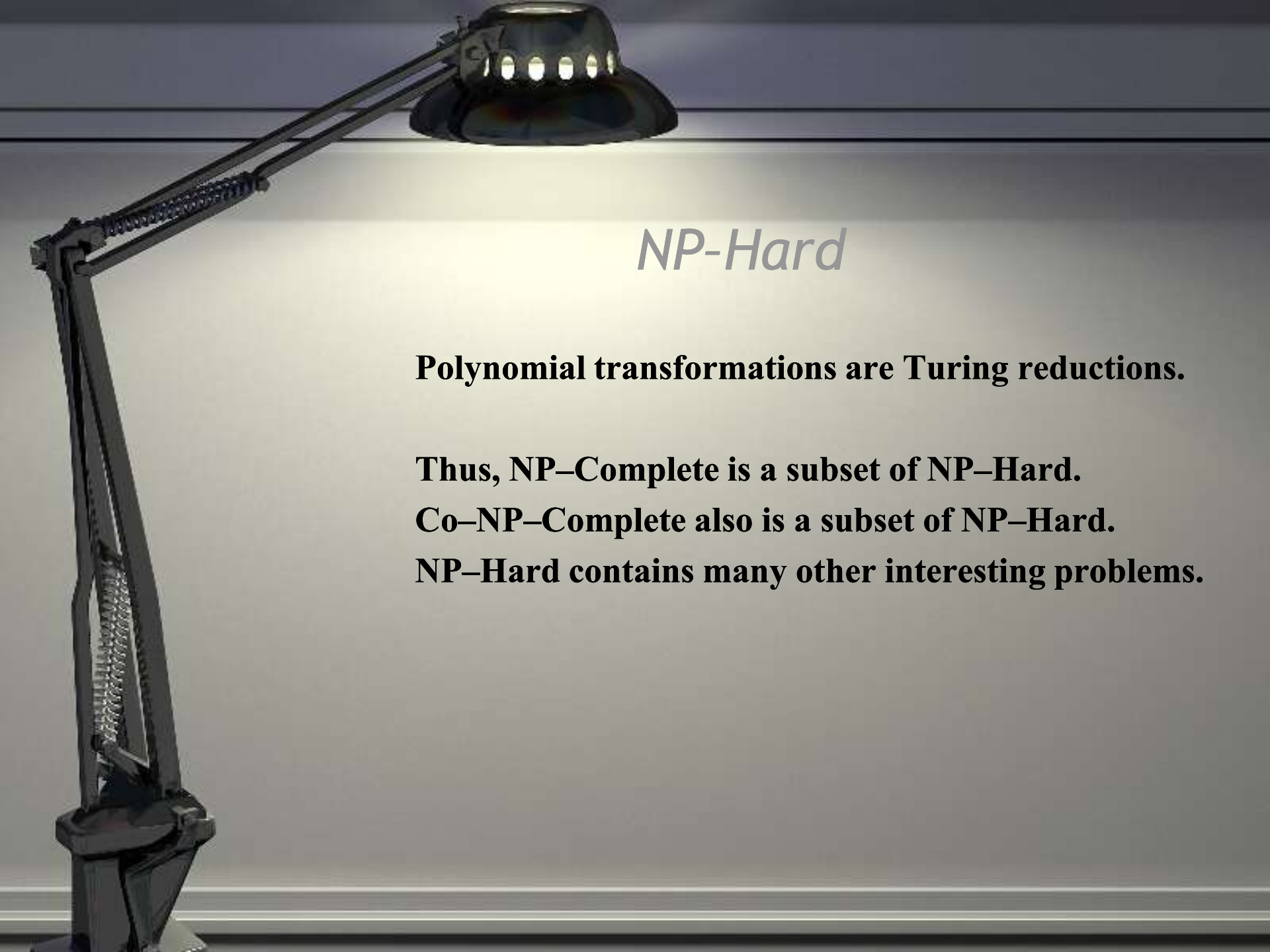
Definition: Problem B is NP–Hard if there is a Turing reduction A ↗ B for some problem A in NP–Complete.

This implies NP–Hard problems are at least as hard as NP–Complete problems. Therefore, they can not be solved in polynomial time unless P = NP (and maybe not then).



NP-Hard

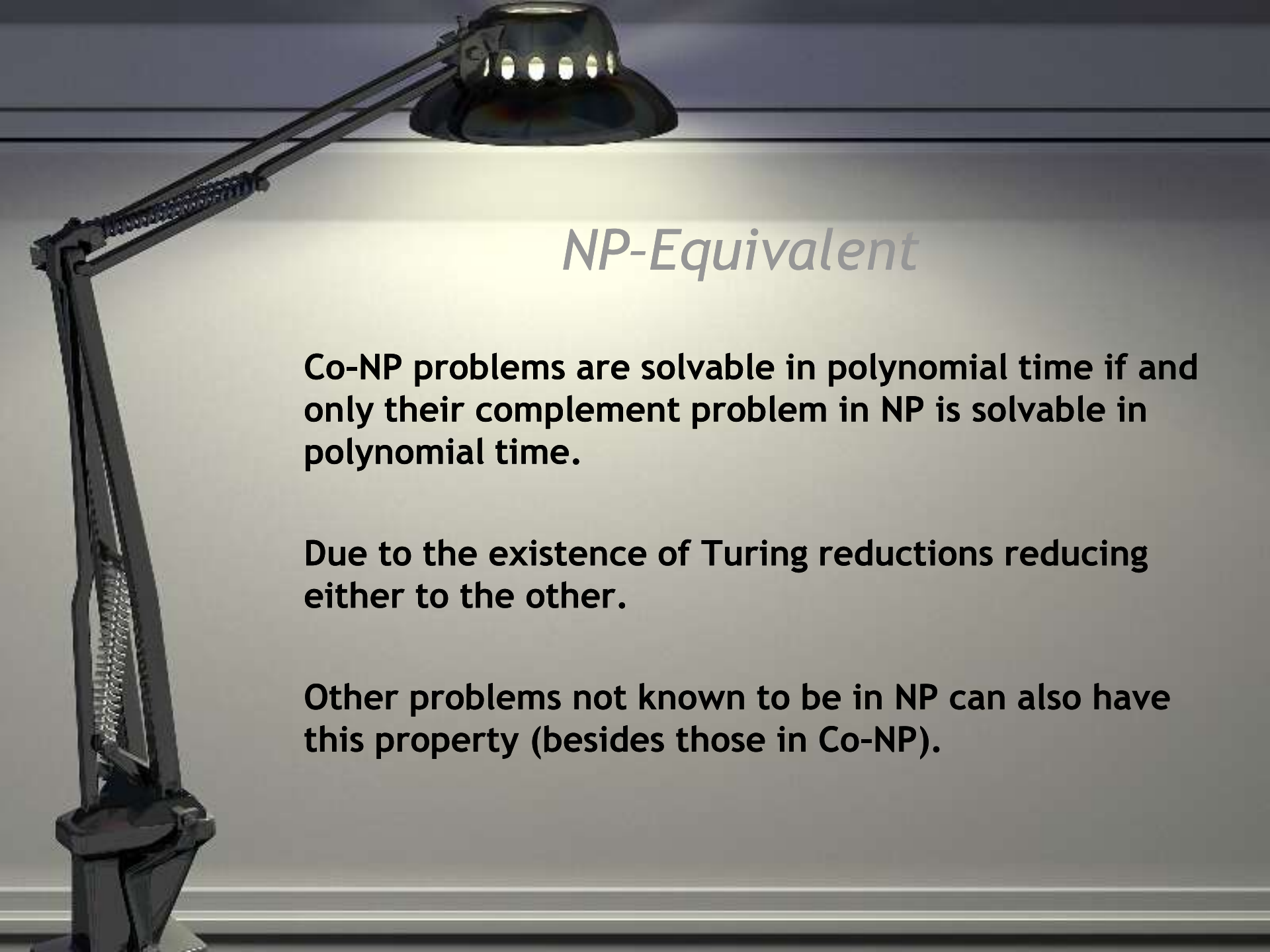
- {Example}



NP-Hard

Polynomial transformations are Turing reductions.

Thus, NP–Complete is a subset of NP–Hard.
Co–NP–Complete also is a subset of NP–Hard.
NP–Hard contains many other interesting problems.

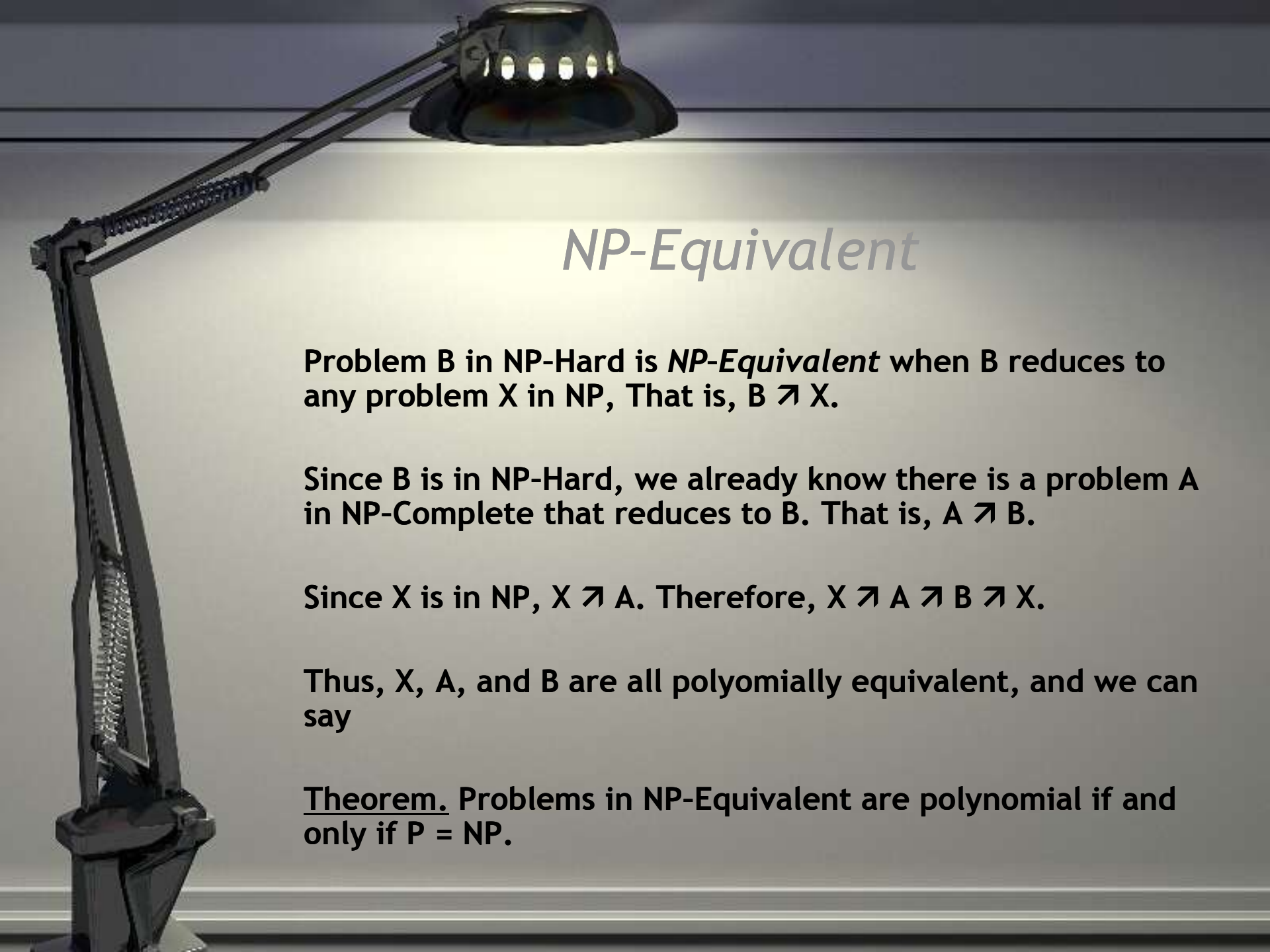


NP-Equivalent

Co-NP problems are solvable in polynomial time if and only their complement problem in NP is solvable in polynomial time.

Due to the existence of Turing reductions reducing either to the other.

Other problems not known to be in NP can also have this property (besides those in Co-NP).



NP-Equivalent

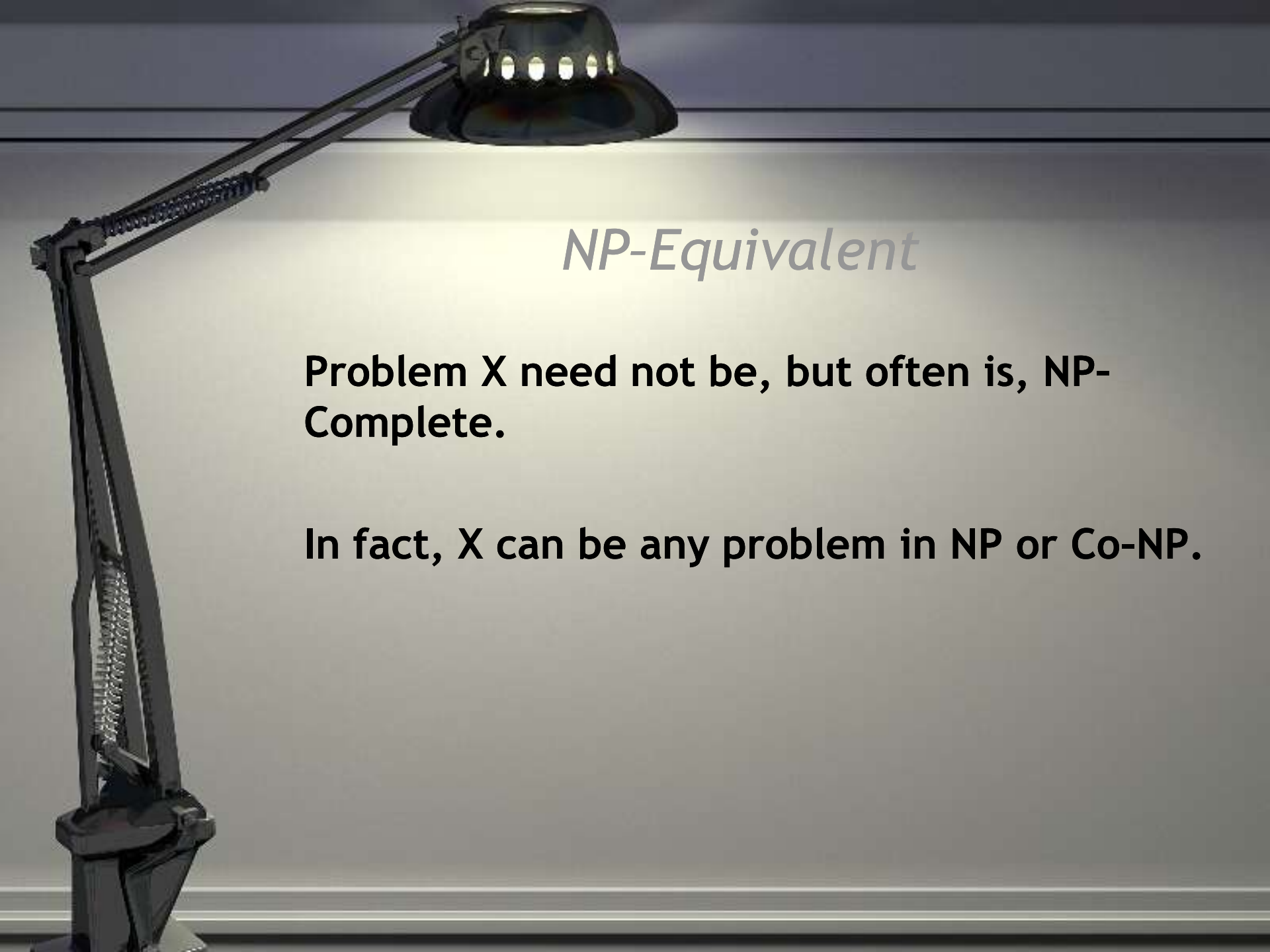
Problem B in NP-Hard is *NP-Equivalent* when B reduces to any problem X in NP, That is, B ↗ X.

Since B is in NP-Hard, we already know there is a problem A in NP-Complete that reduces to B. That is, A ↗ B.

Since X is in NP, X ↗ A. Therefore, X ↗ A ↗ B ↗ X.

Thus, X, A, and B are all polyomially equivalent, and we can say

<u>Theorem.</u> Problems in NP-Equivalent are polynomial if and only if P = NP.

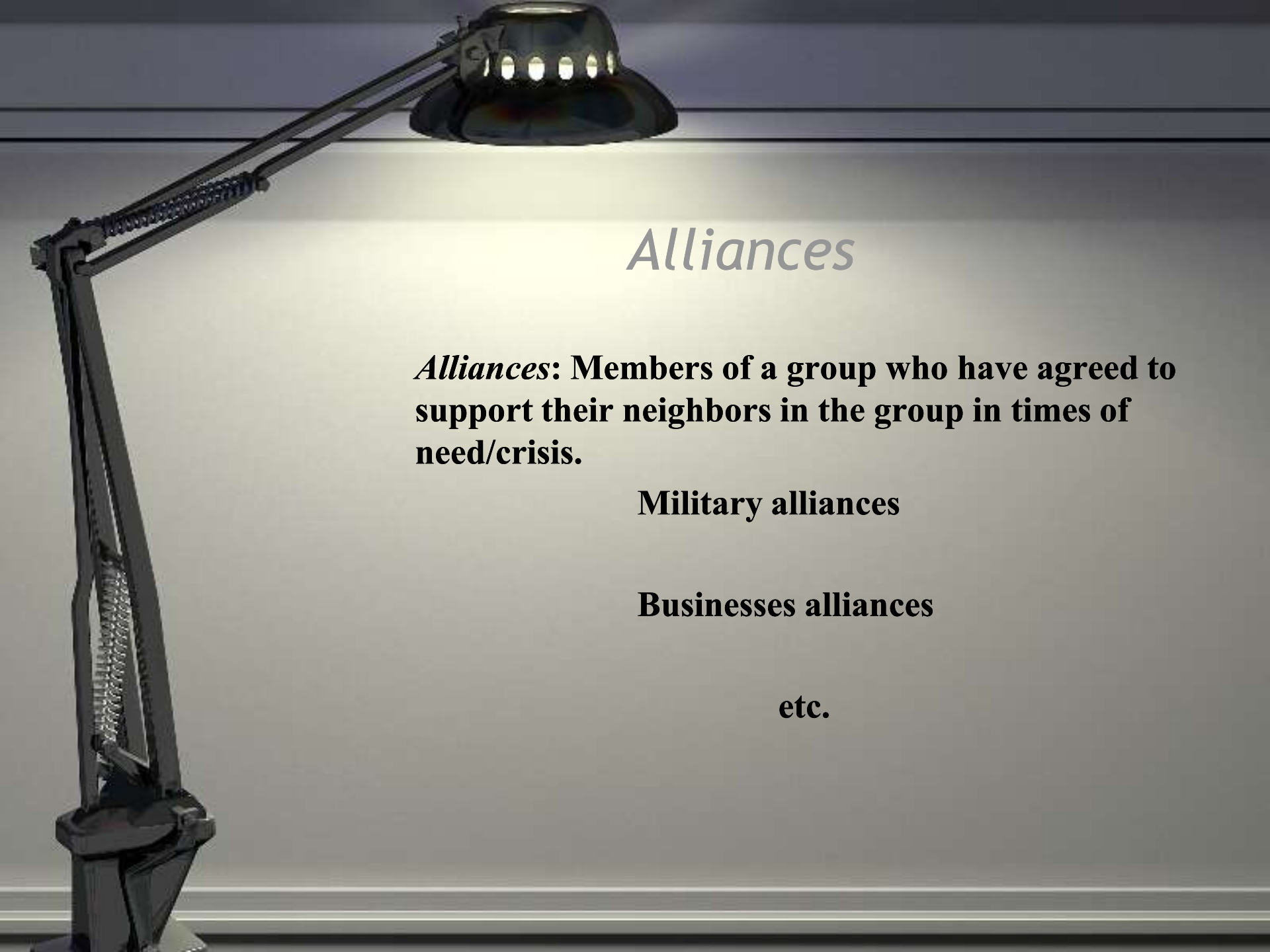


NP-Equivalent

Problem X need not be, but often is, NP-Complete.

In fact, X can be any problem in NP or Co-NP.


Case Studies
Alliances and Secure Sets



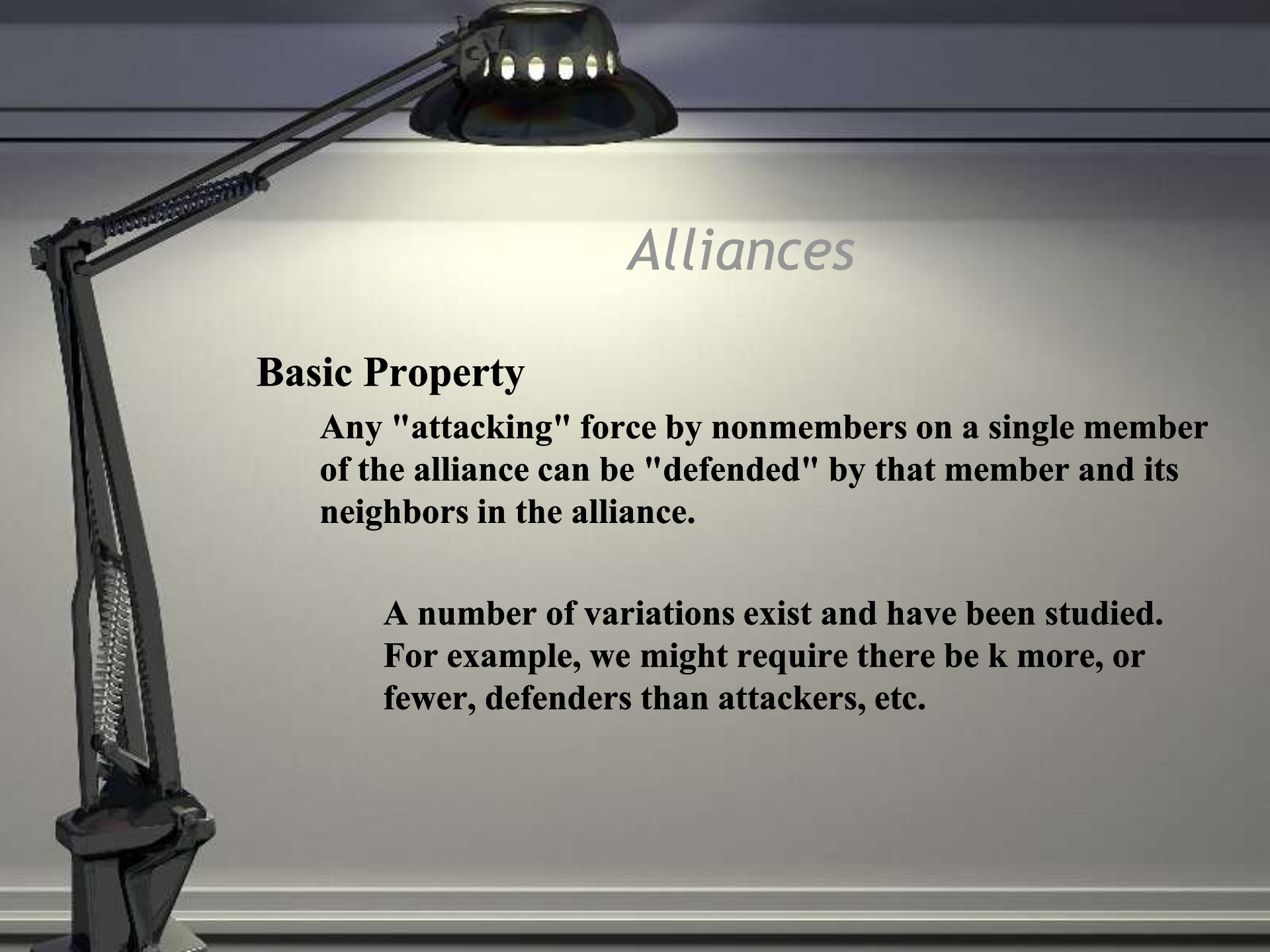
Alliances

***Alliances*: Members of a group who have agreed to support their neighbors in the group in times of need/crisis.**

Military alliances

Businesses alliances

etc.

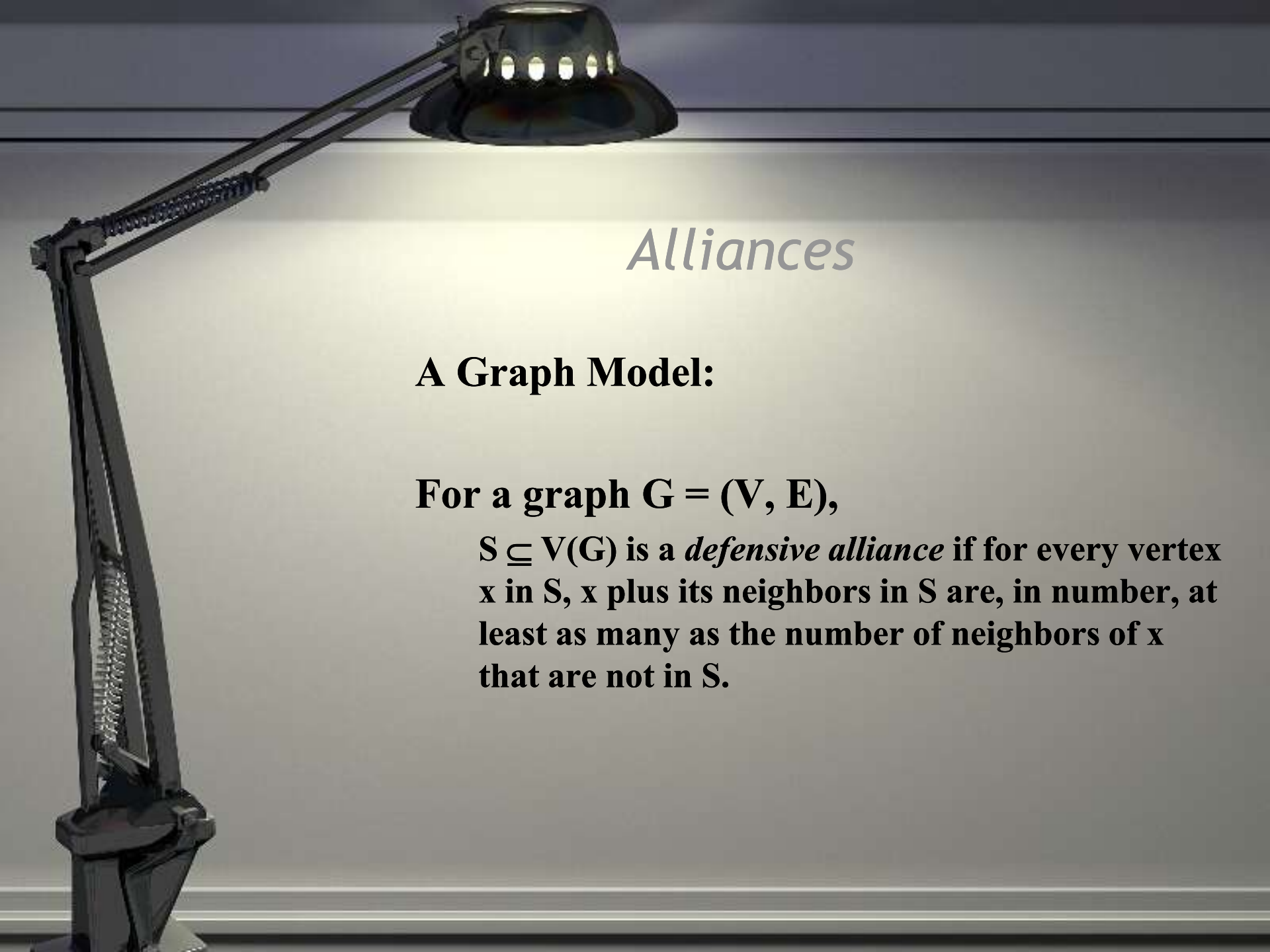


Alliances

Basic Property

Any "attacking" force by nonmembers on a single member of the alliance can be "defended" by that member and its neighbors in the alliance.

A number of variations exist and have been studied. For example, we might require there be k more, or fewer, defenders than attackers, etc.

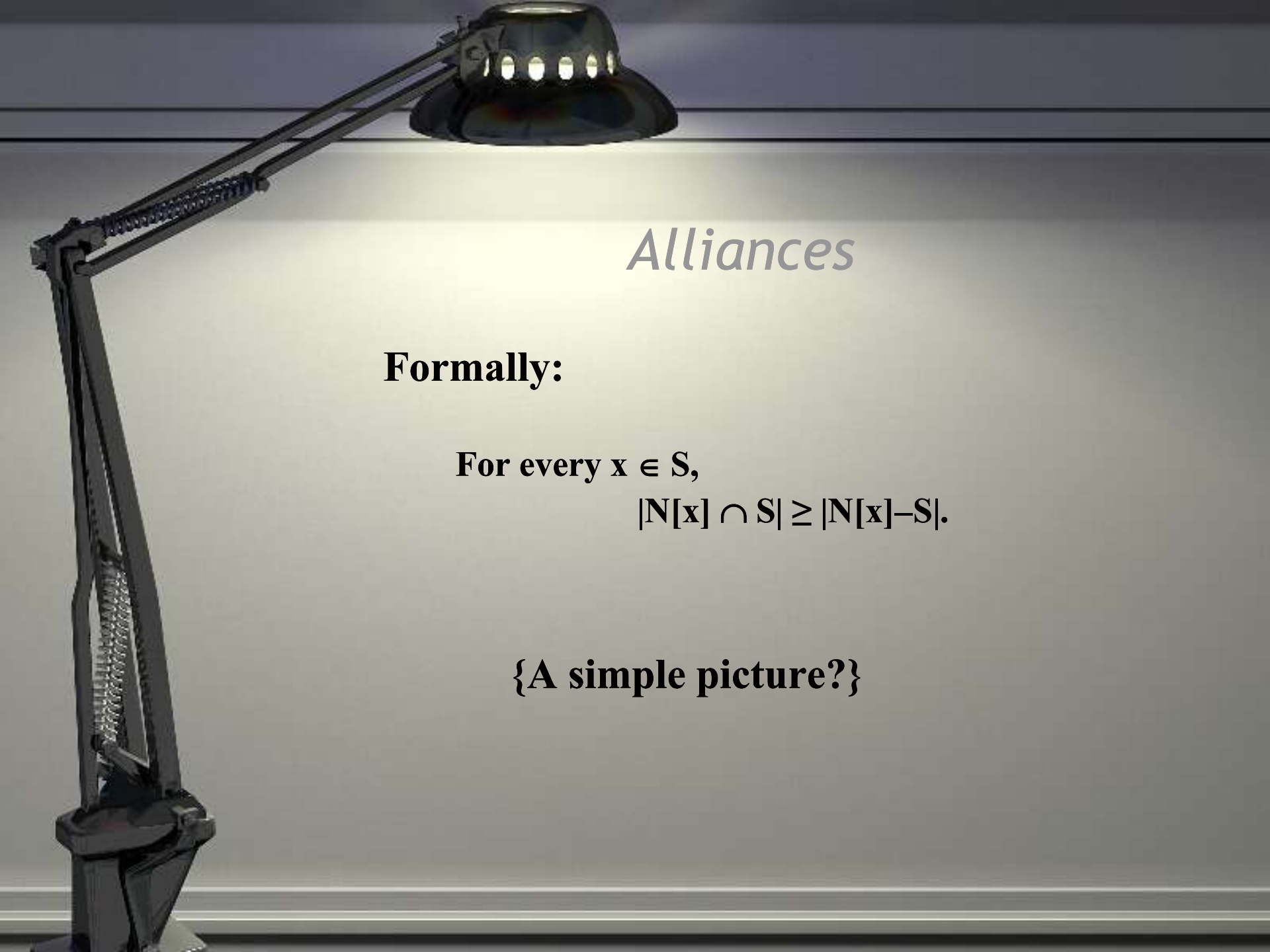


Alliances

A Graph Model:

For a graph G = (V, E),

S $\subseteq$ V(G) is a *defensive alliance* if for every vertex x in S, x plus its neighbors in S are, in number, at least as many as the number of neighbors of x that are not in S.



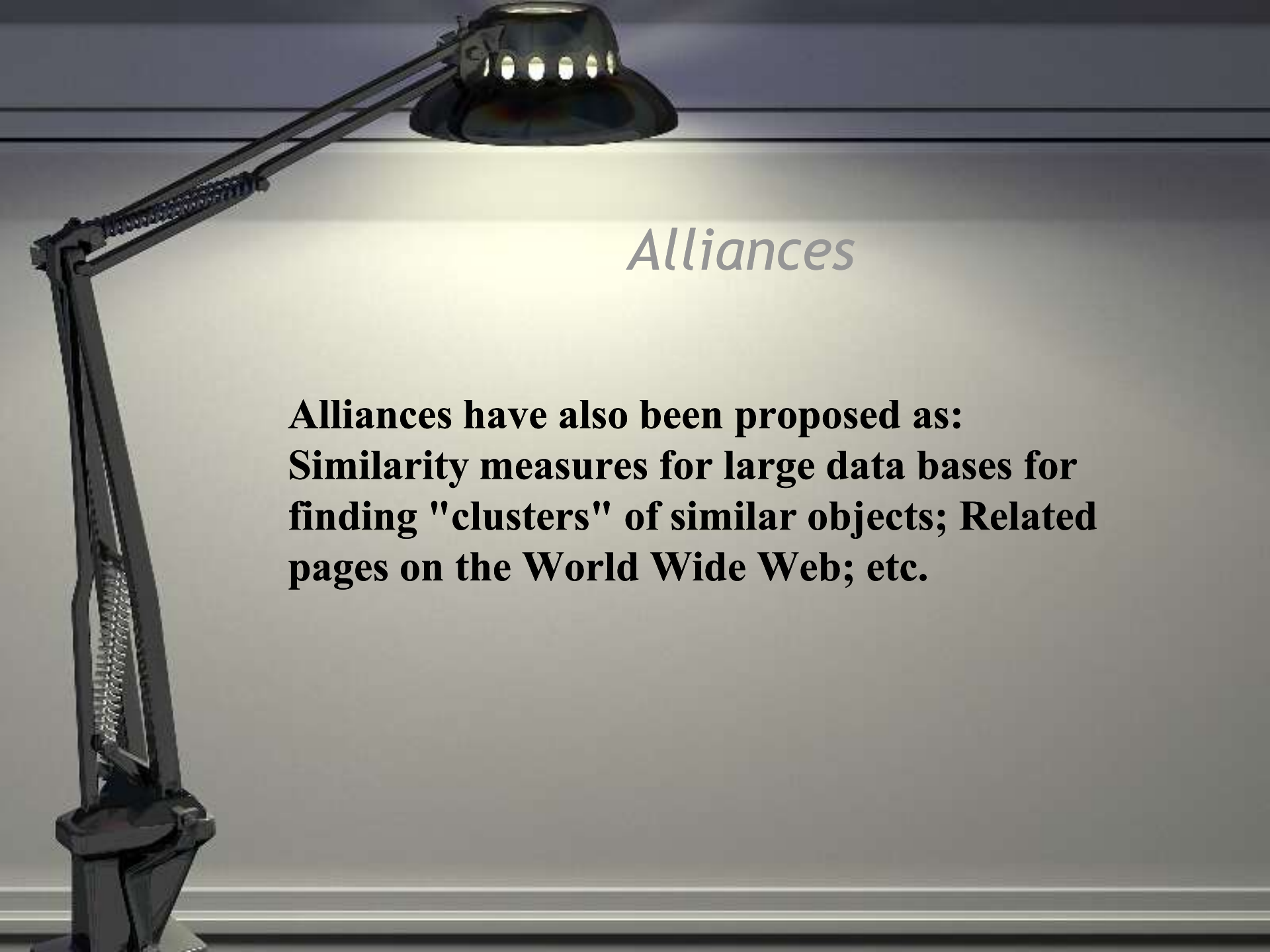
Alliances

Formally:

For every $x \in S$,

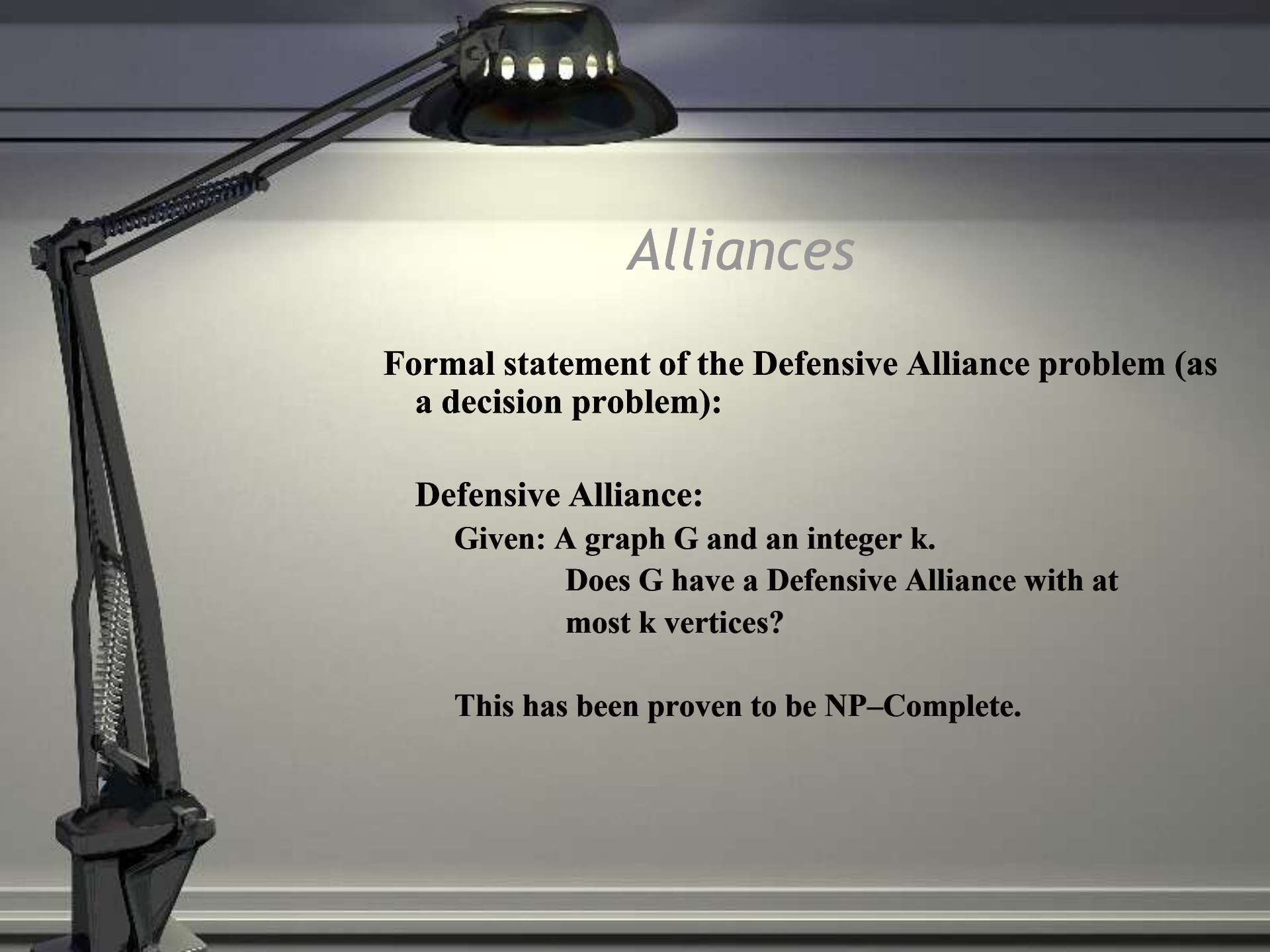
$$|N[x] \cap S| \geq |N[x]-S|.$$

{A simple picture?}



Alliances

Alliances have also been proposed as: Similarity measures for large data bases for finding "clusters" of similar objects; Related pages on the World Wide Web; etc.



Alliances

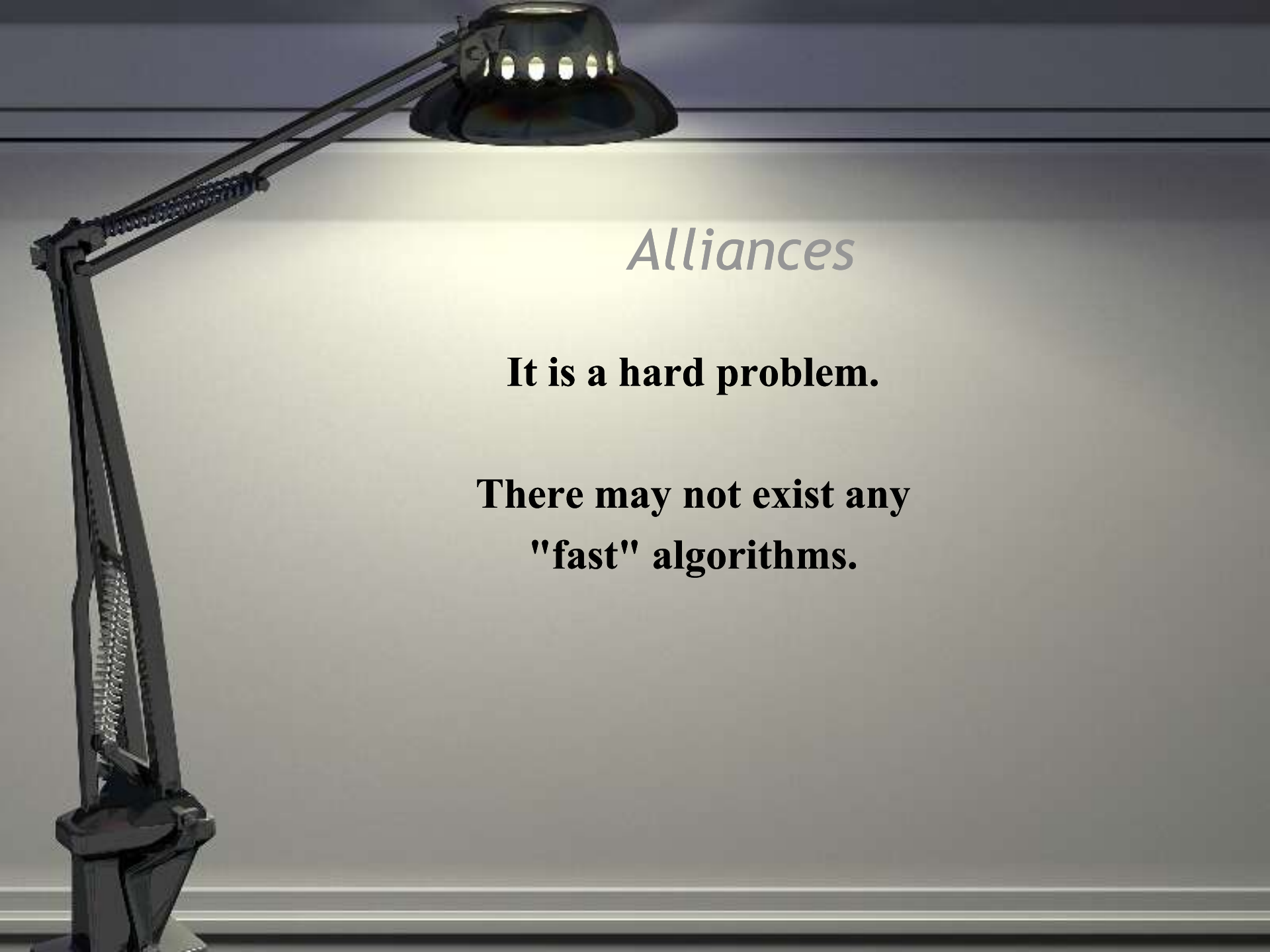
Formal statement of the Defensive Alliance problem (as a decision problem):

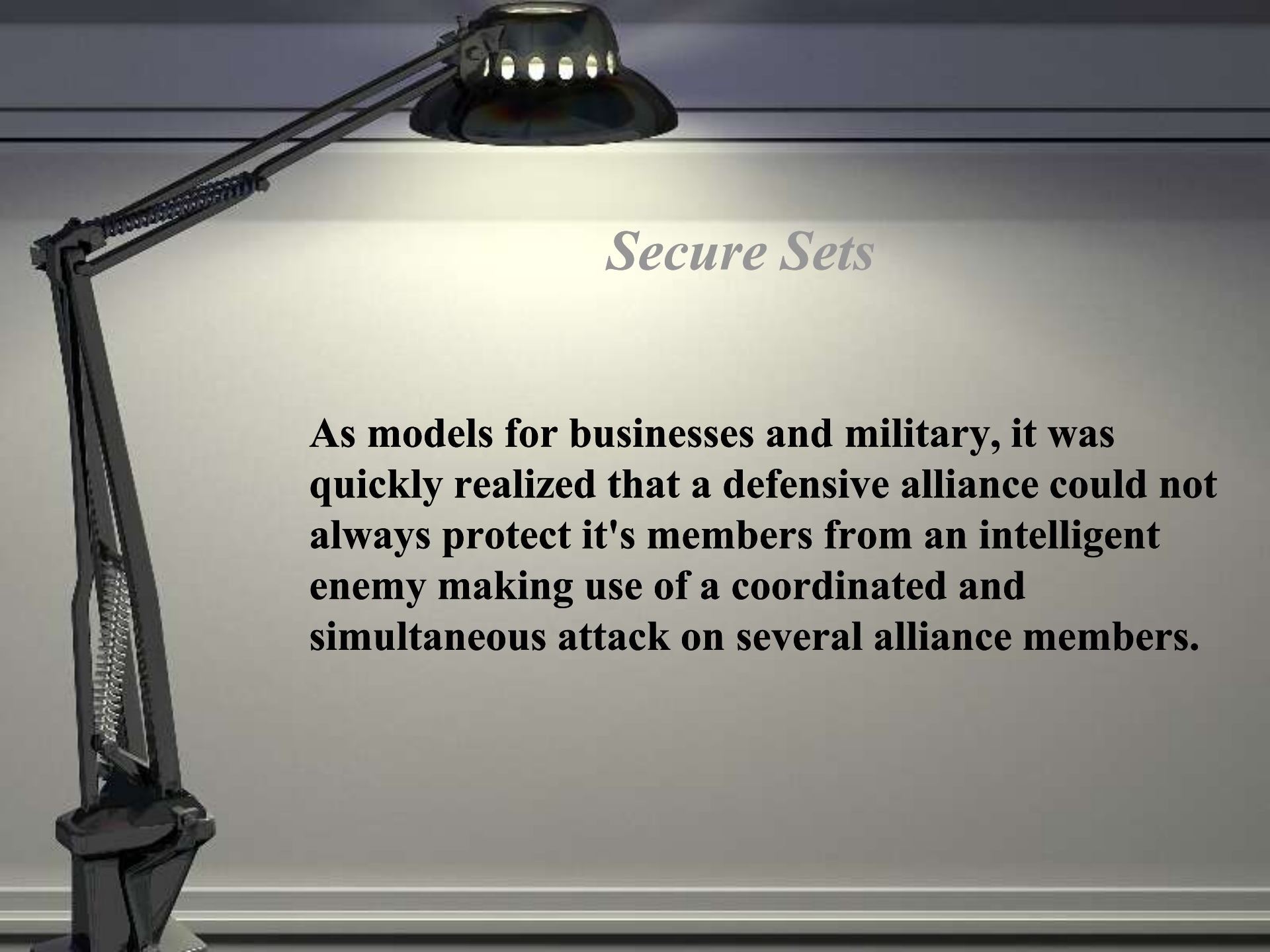
Defensive Alliance:

Given: A graph G and an integer k.

Does G have a Defensive Alliance with at most k vertices?

This has been proven to be NP–Complete.


Alliances
It is a hard problem.
There may not exist any
"fast" algorithms.

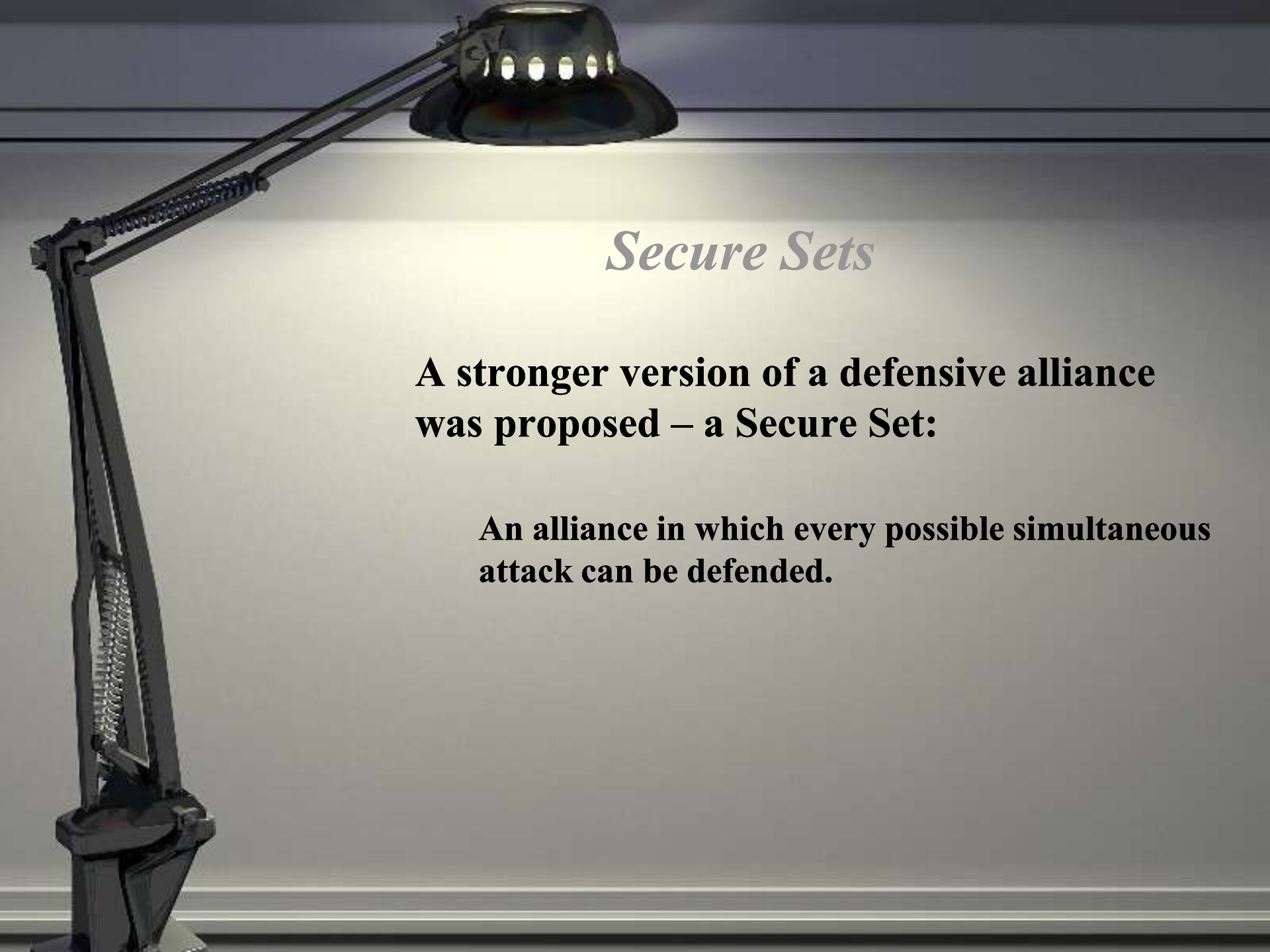


Secure Sets

As models for businesses and military, it was quickly realized that a defensive alliance could not always protect it's members from an intelligent enemy making use of a coordinated and simultaneous attack on several alliance members.

Secure Sets

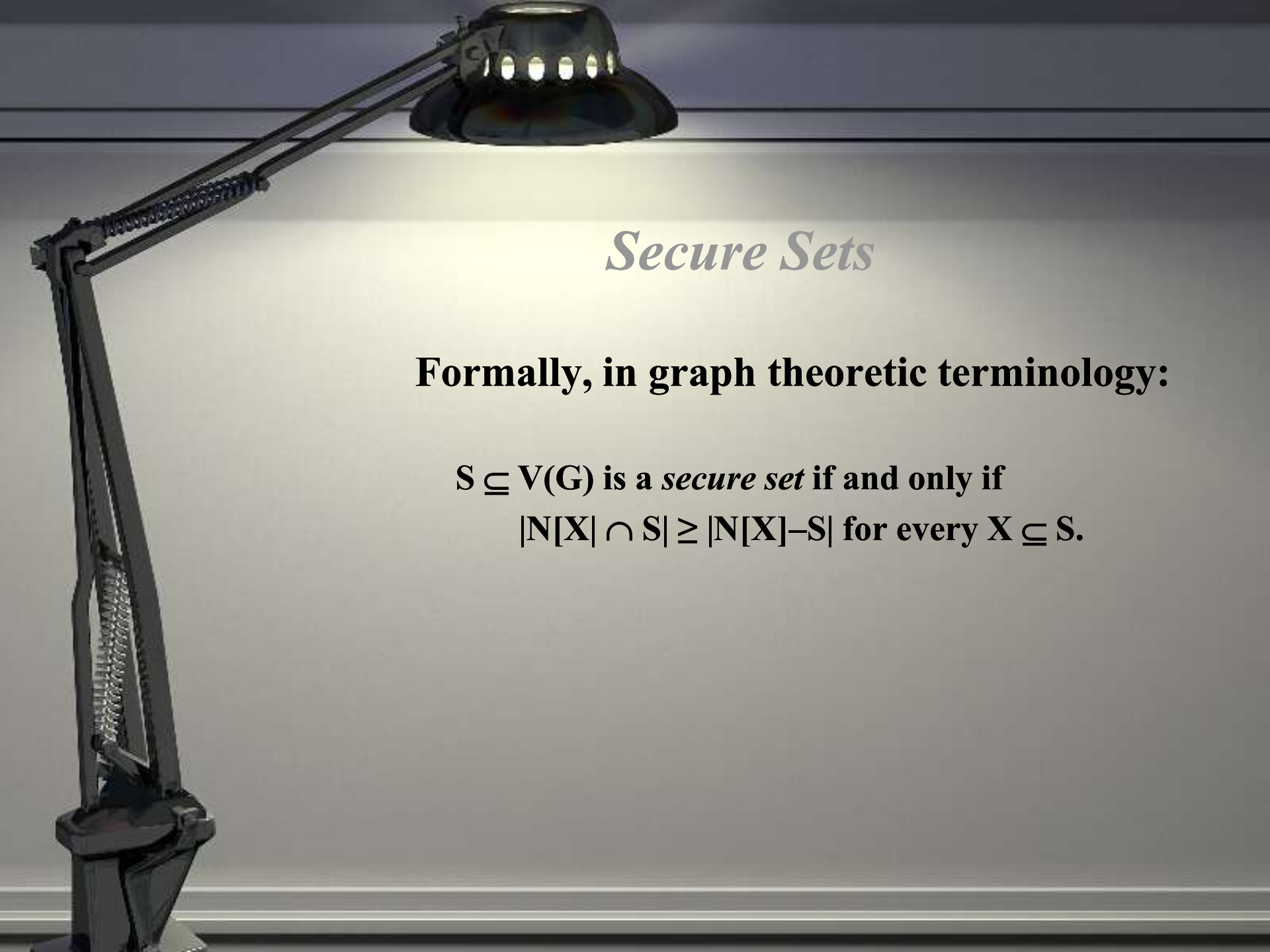
{Use the picture above as an example.}



Secure Sets

A stronger version of a defensive alliance was proposed – a Secure Set:

An alliance in which every possible simultaneous attack can be defended.

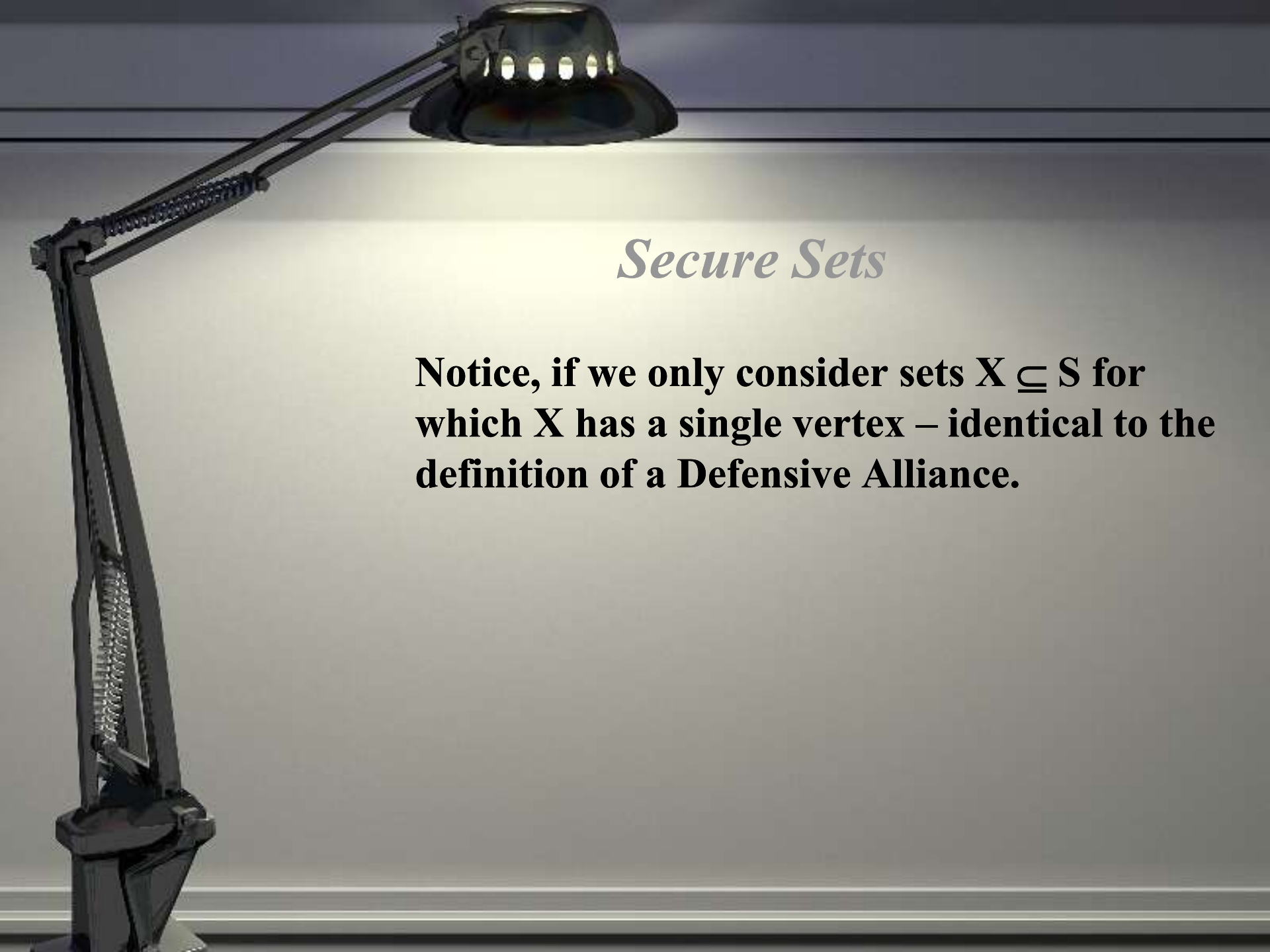


Secure Sets

Formally, in graph theoretic terminology:

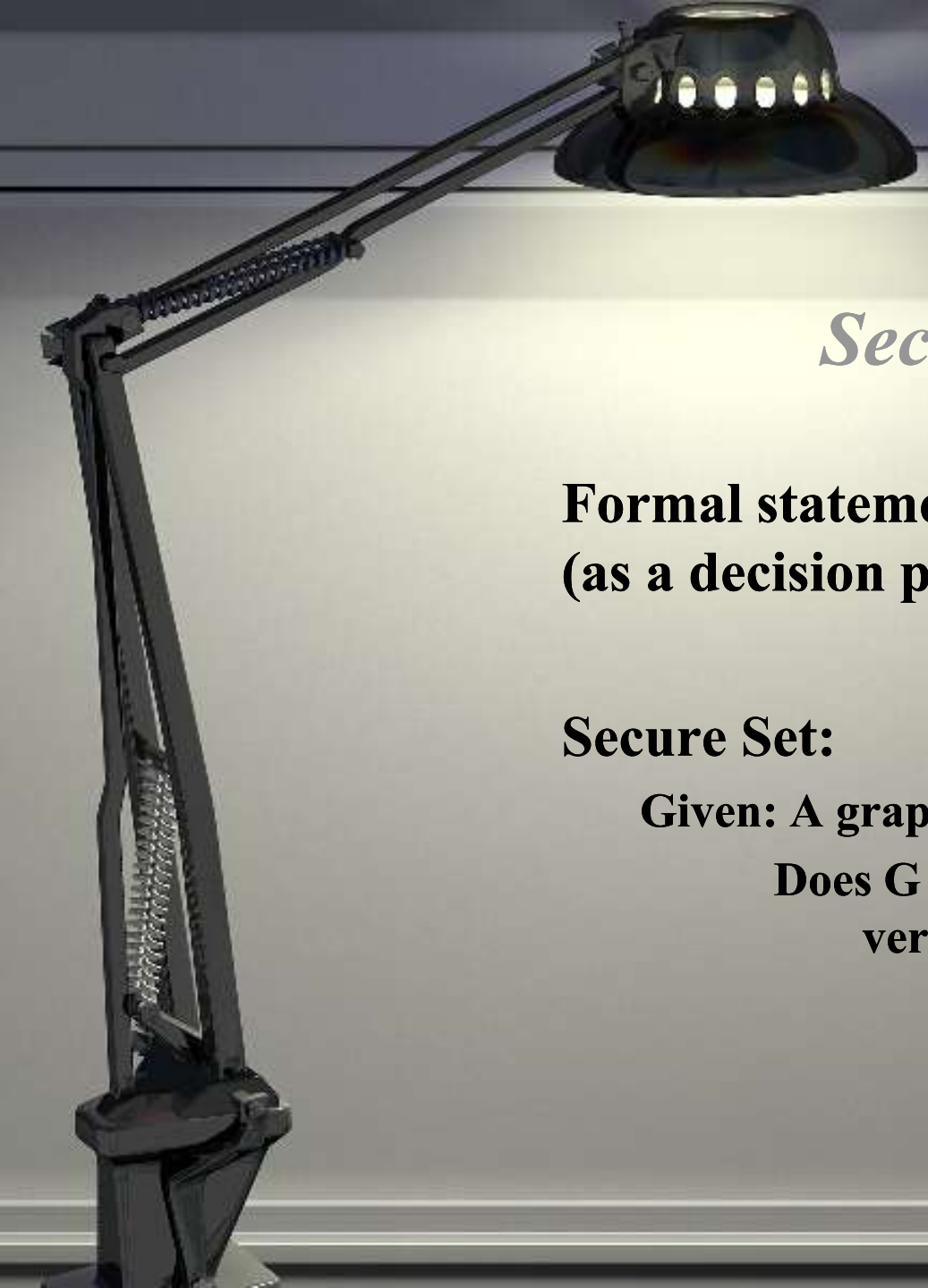
$S \subseteq V(G)$ is a *secure set* if and only if

$|N[X] \cap S| \geq |N[X]-S|$ for every $X \subseteq S$.



Secure Sets

Notice, if we only consider sets $X \subseteq S$ for which X has a single vertex – identical to the definition of a Defensive Alliance.



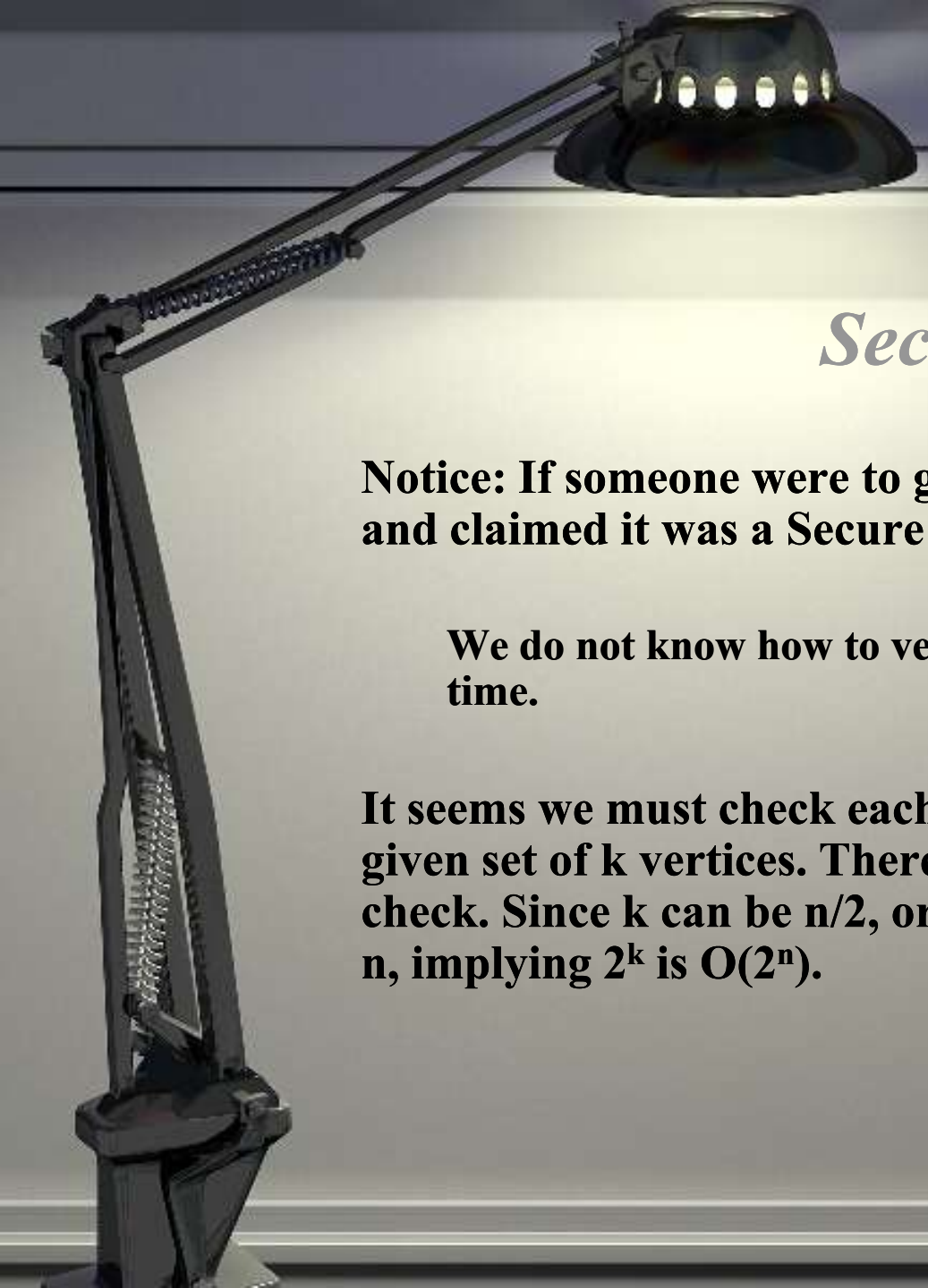
Secure Sets

Formal statement of the Secure Set problem (as a decision problem):

Secure Set:

Given: A graph G and an integer k.

Does G have a Secure Set with at most k vertices?

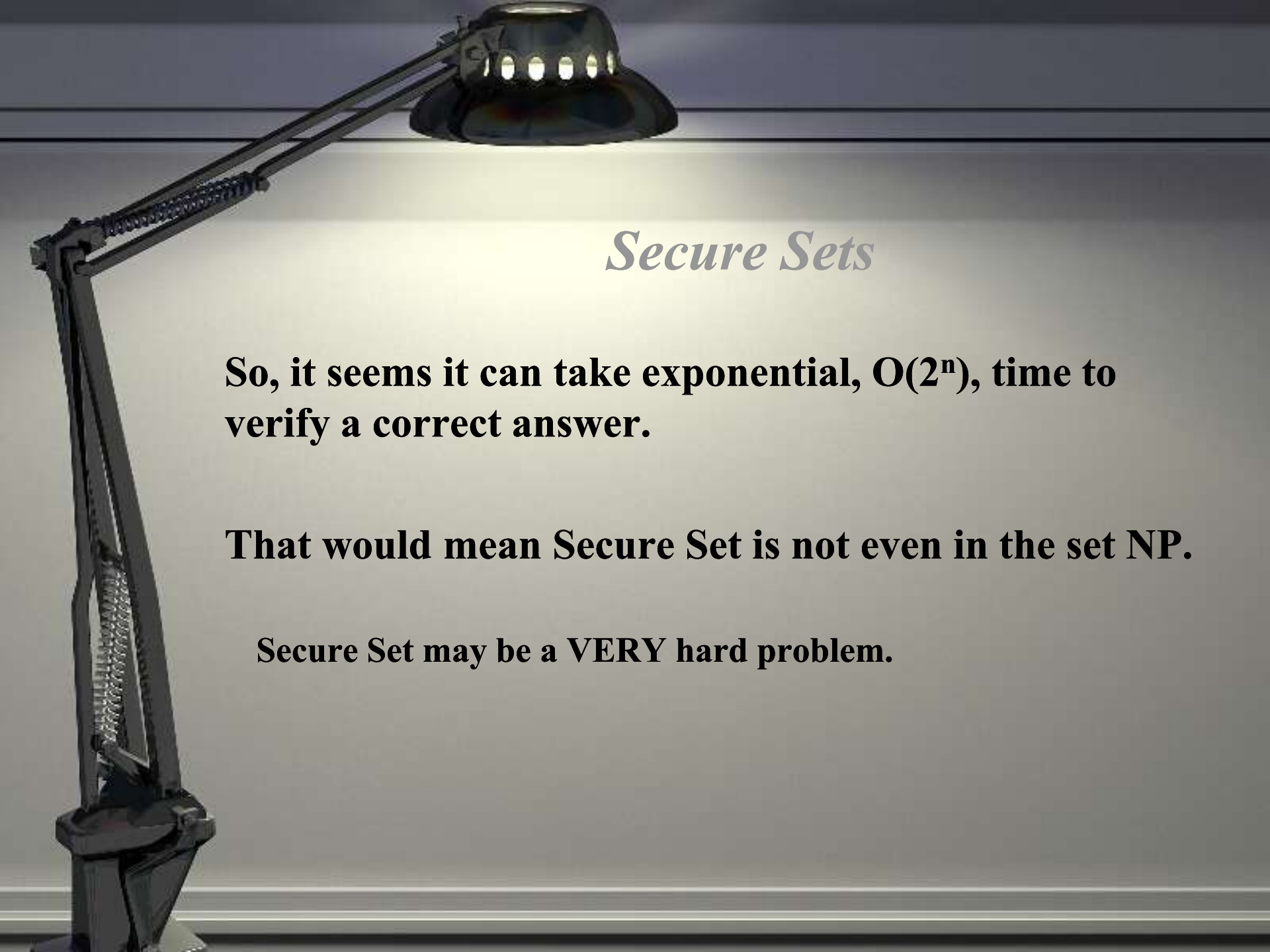


Secure Sets

Notice: If someone were to give us a set of k vertices and claimed it was a Secure Set:

> **We do not know how to verify the claim in polynomial time.**

It seems we must check each individual subset of the given set of k vertices. There are 2^k possible subsets to check. Since k can be n/2, or n/4, etc., k can be order n, implying 2^k is $O(2^n)$.

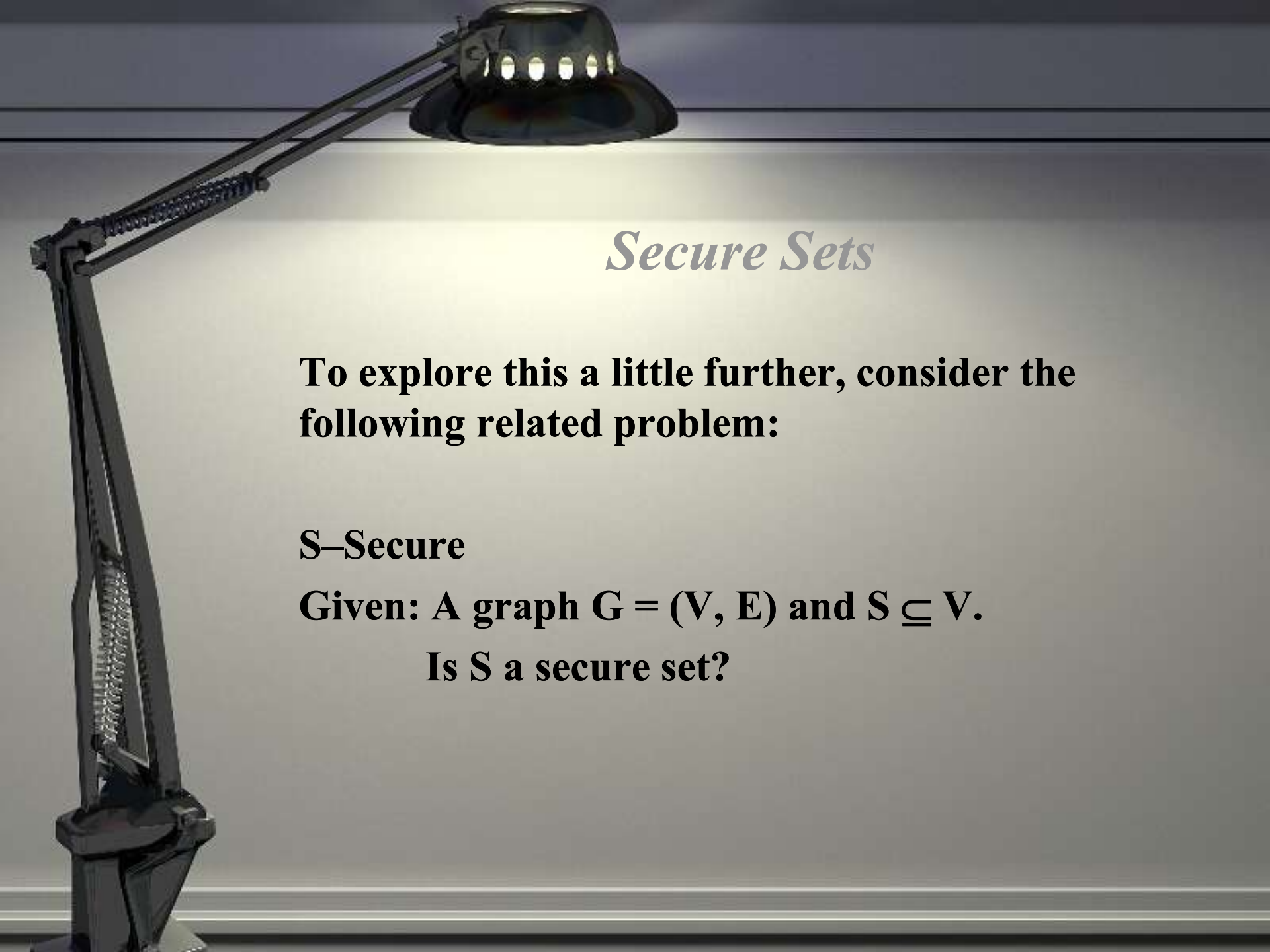


Secure Sets

So, it seems it can take exponential, $O(2^n)$, time to verify a correct answer.

That would mean Secure Set is not even in the set NP.

Secure Set may be a VERY hard problem.



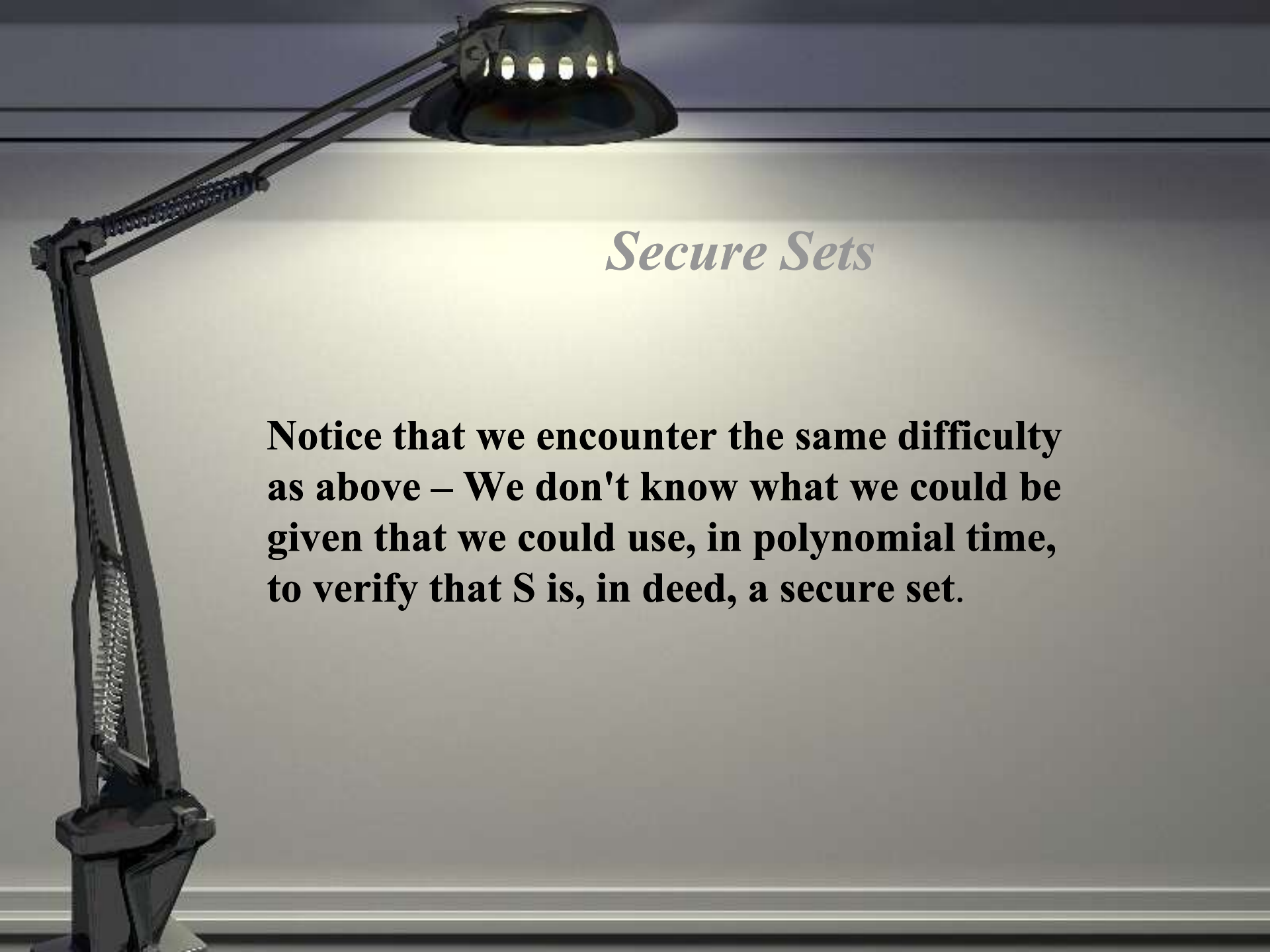
Secure Sets

To explore this a little further, consider the following related problem:

S–Secure

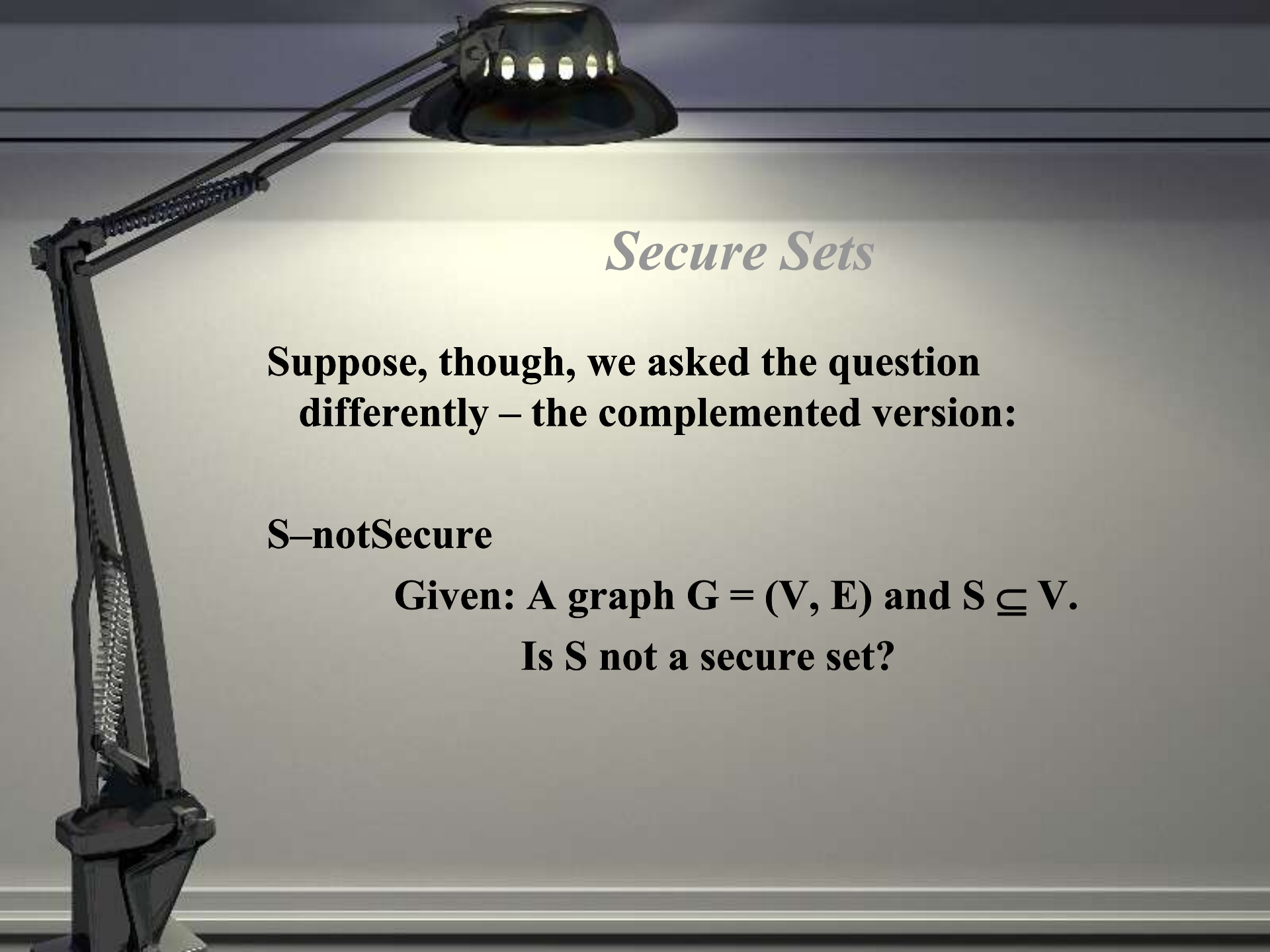
Given: A graph G = (V, E) and S $\subseteq$ V.

Is S a secure set?



Secure Sets

Notice that we encounter the same difficulty as above – We don't know what we could be given that we could use, in polynomial time, to verify that S is, in deed, a secure set.



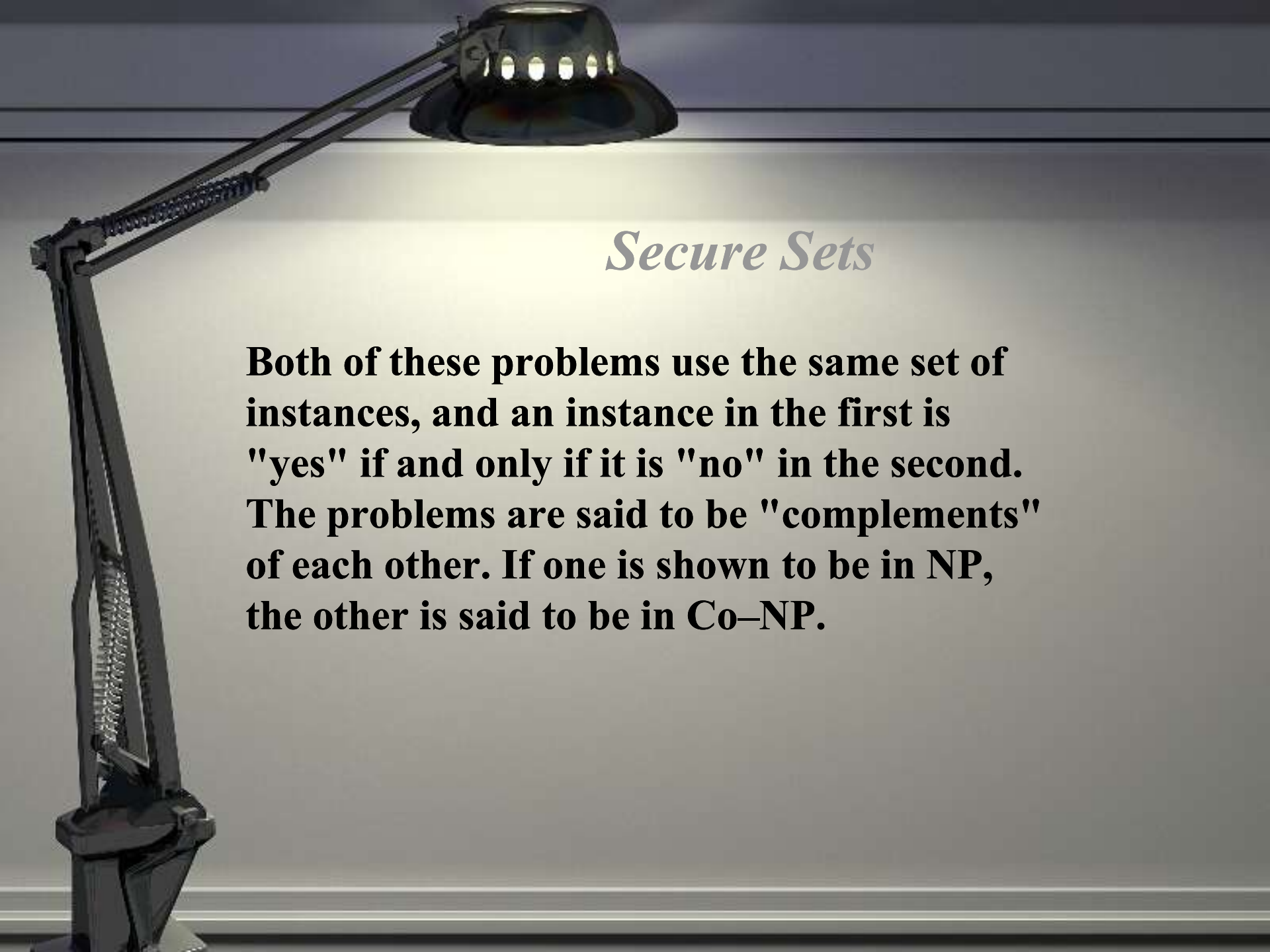
Secure Sets

Suppose, though, we asked the question differently – the complemented version:

S–notSecure

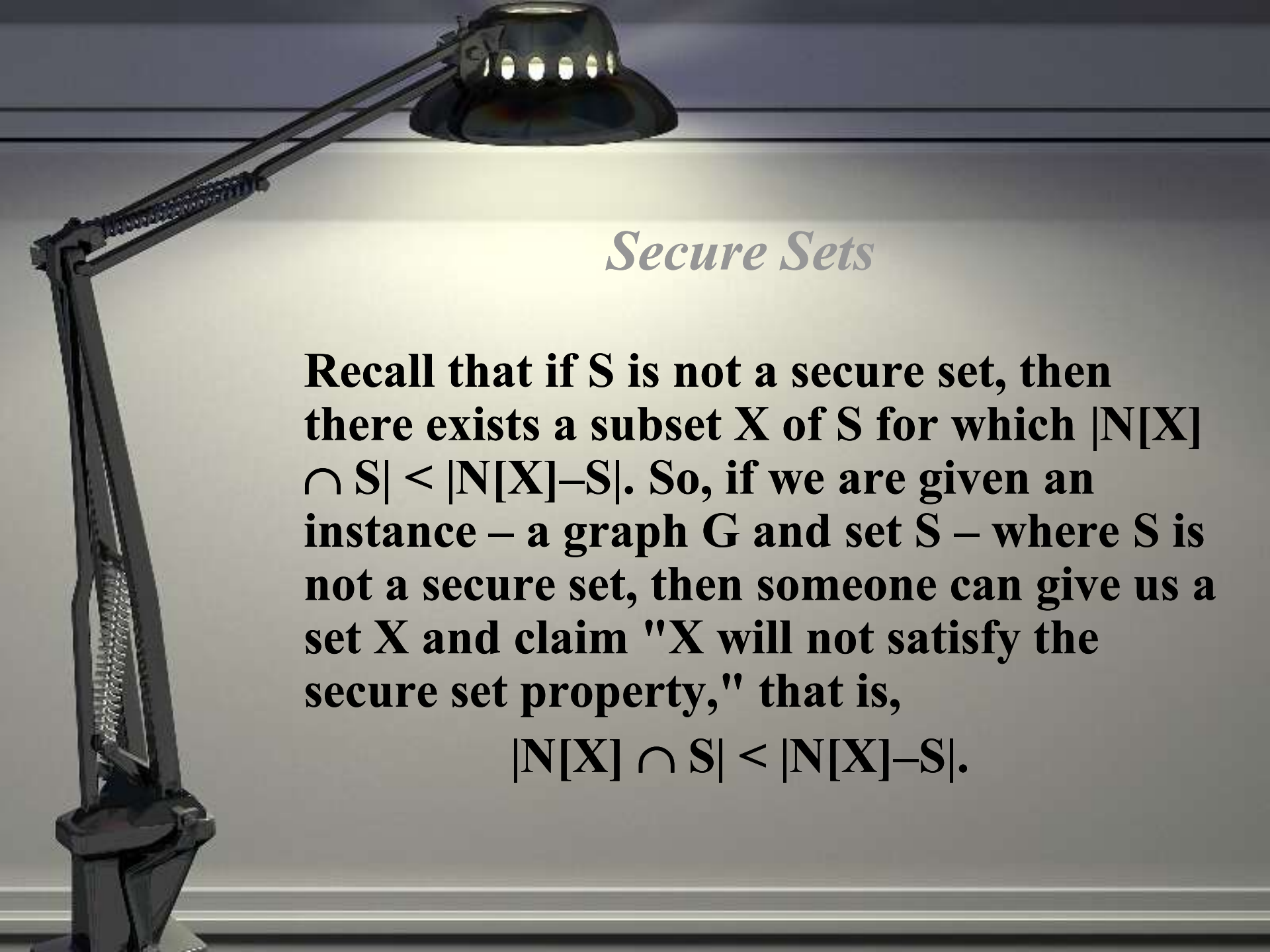
Given: A graph G = (V, E) and S $\subseteq$ V.

Is S not a secure set?



Secure Sets

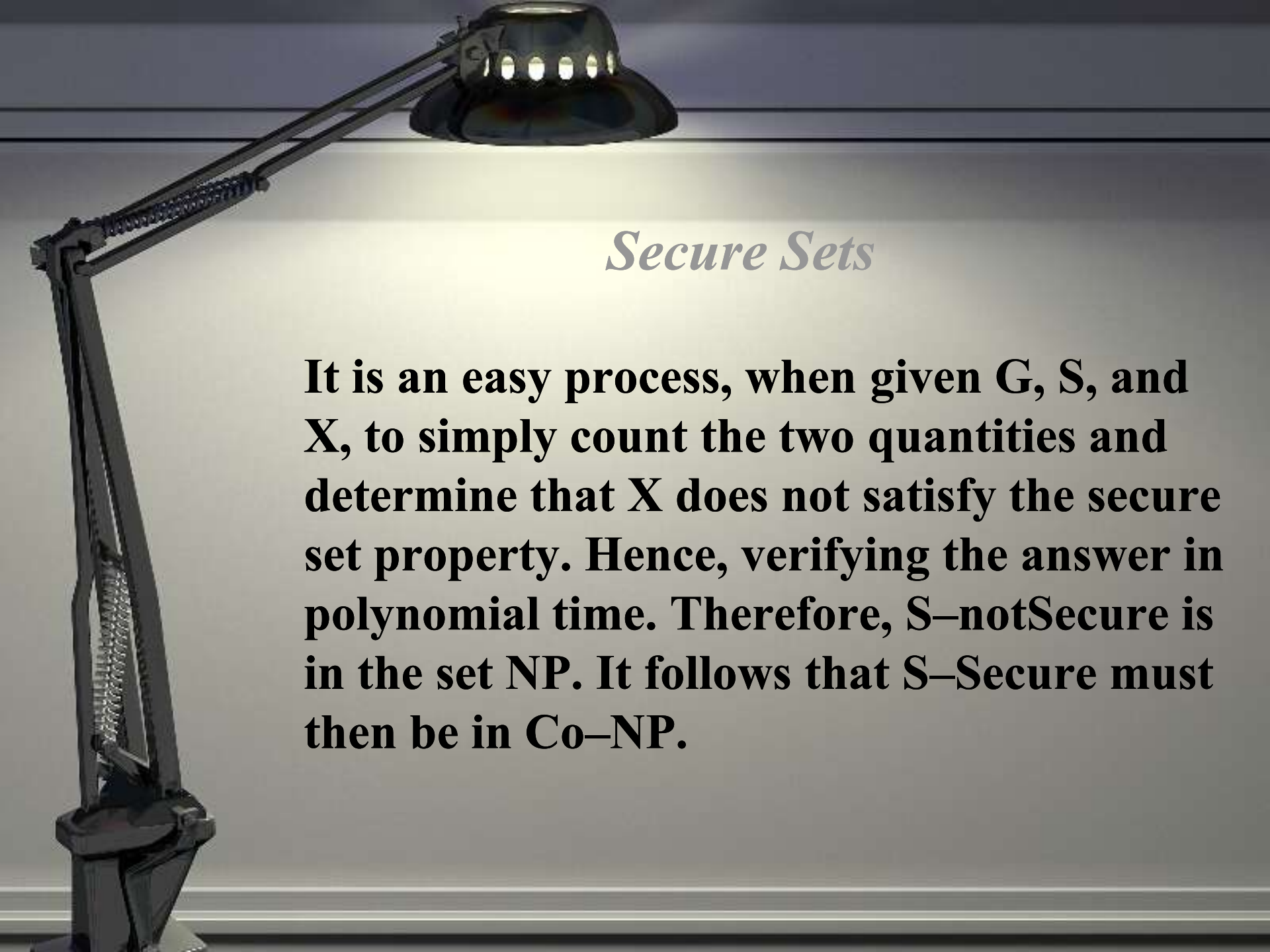
Both of these problems use the same set of instances, and an instance in the first is "yes" if and only if it is "no" in the second. The problems are said to be "complements" of each other. If one is shown to be in NP, the other is said to be in Co–NP.



Secure Sets

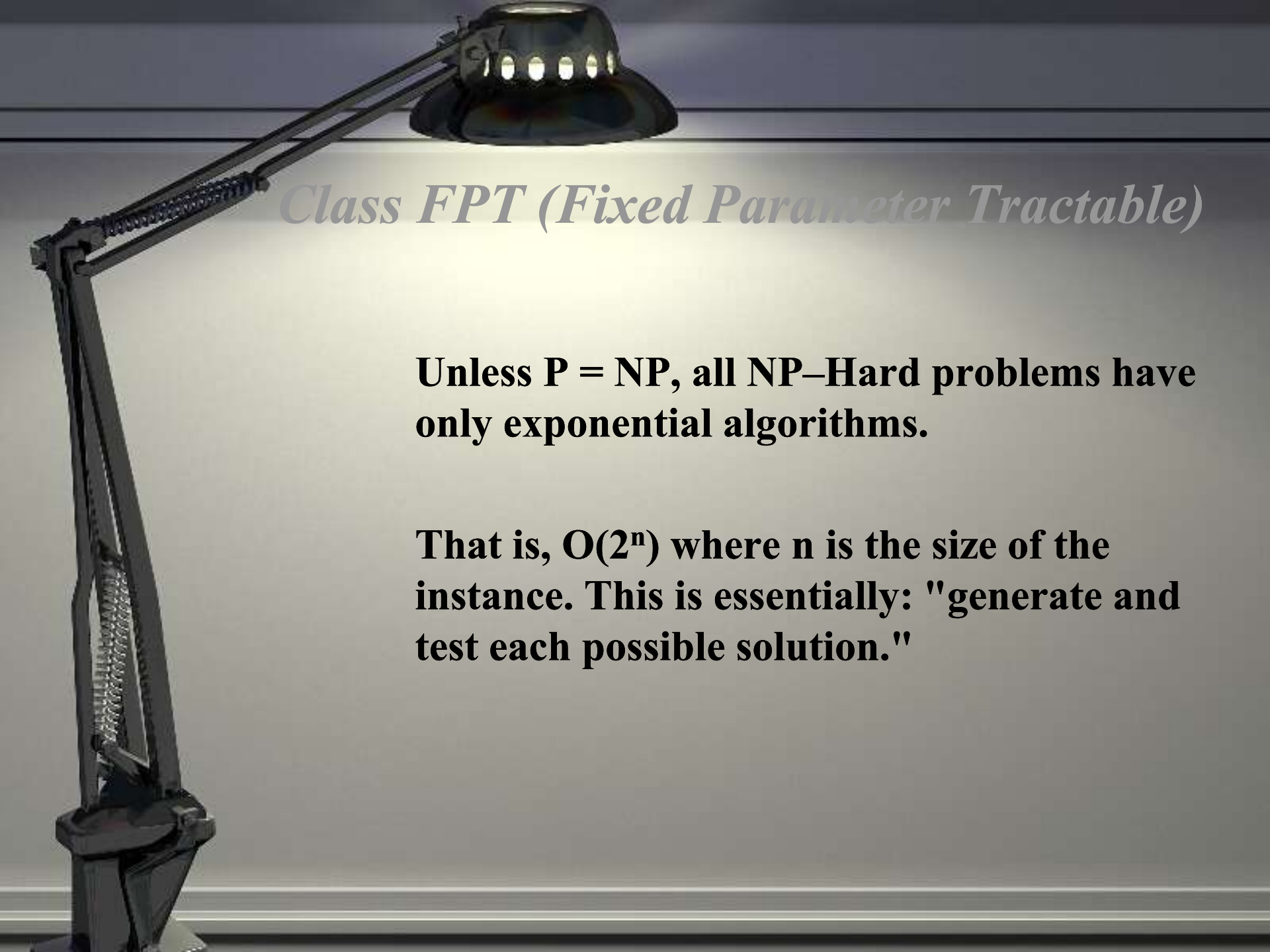
Recall that if S is not a secure set, then there exists a subset X of S for which $|N[X] \cap S| < |N[X]–S|$. So, if we are given an instance – a graph G and set S – where S is not a secure set, then someone can give us a set X and claim "X will not satisfy the secure set property," that is,

$$|N[X] \cap S| < |N[X]–S|.$$



Secure Sets

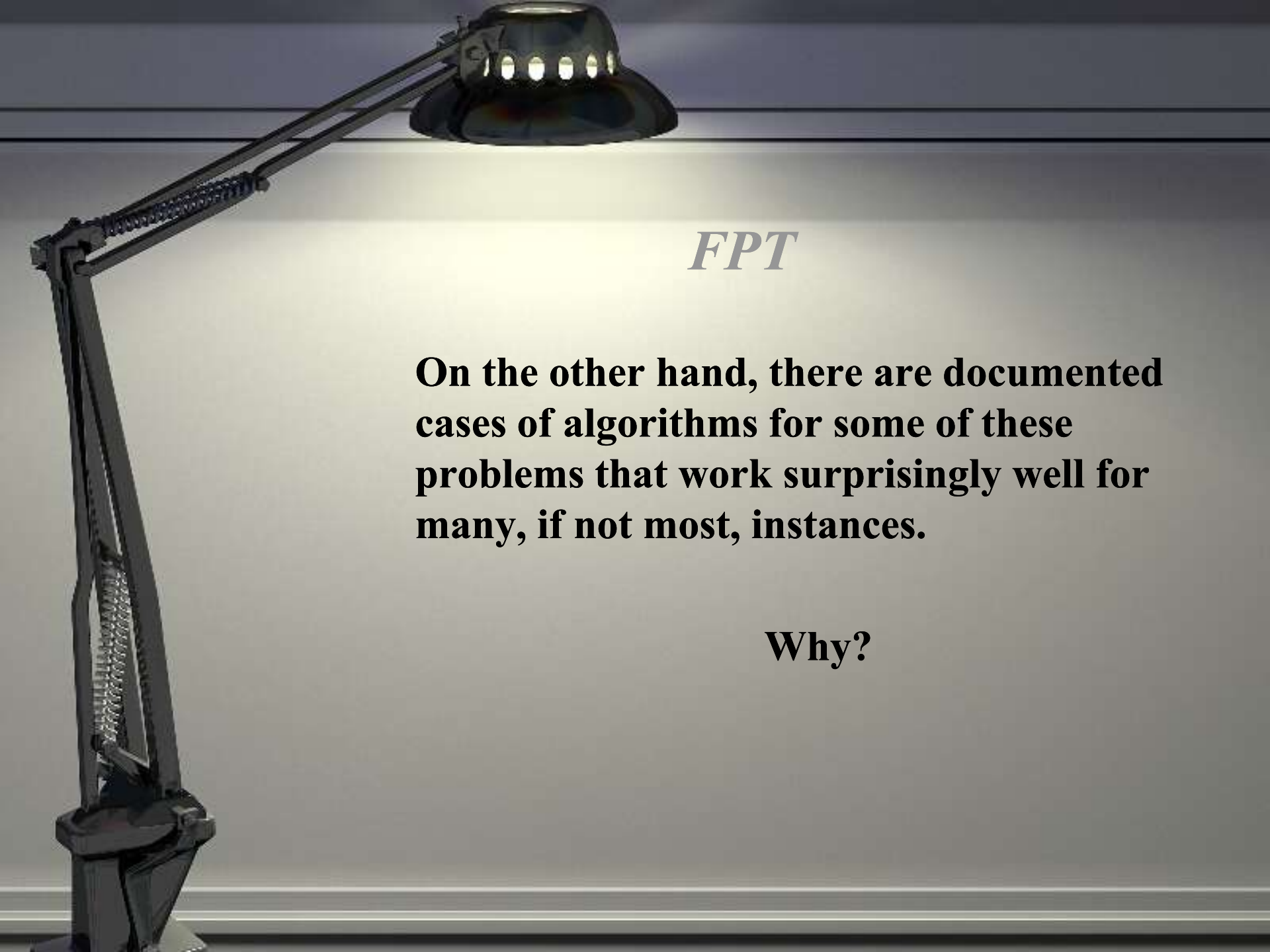
It is an easy process, when given G, S, and X, to simply count the two quantities and determine that X does not satisfy the secure set property. Hence, verifying the answer in polynomial time. Therefore, S–notSecure is in the set NP. It follows that S–Secure must then be in Co–NP.

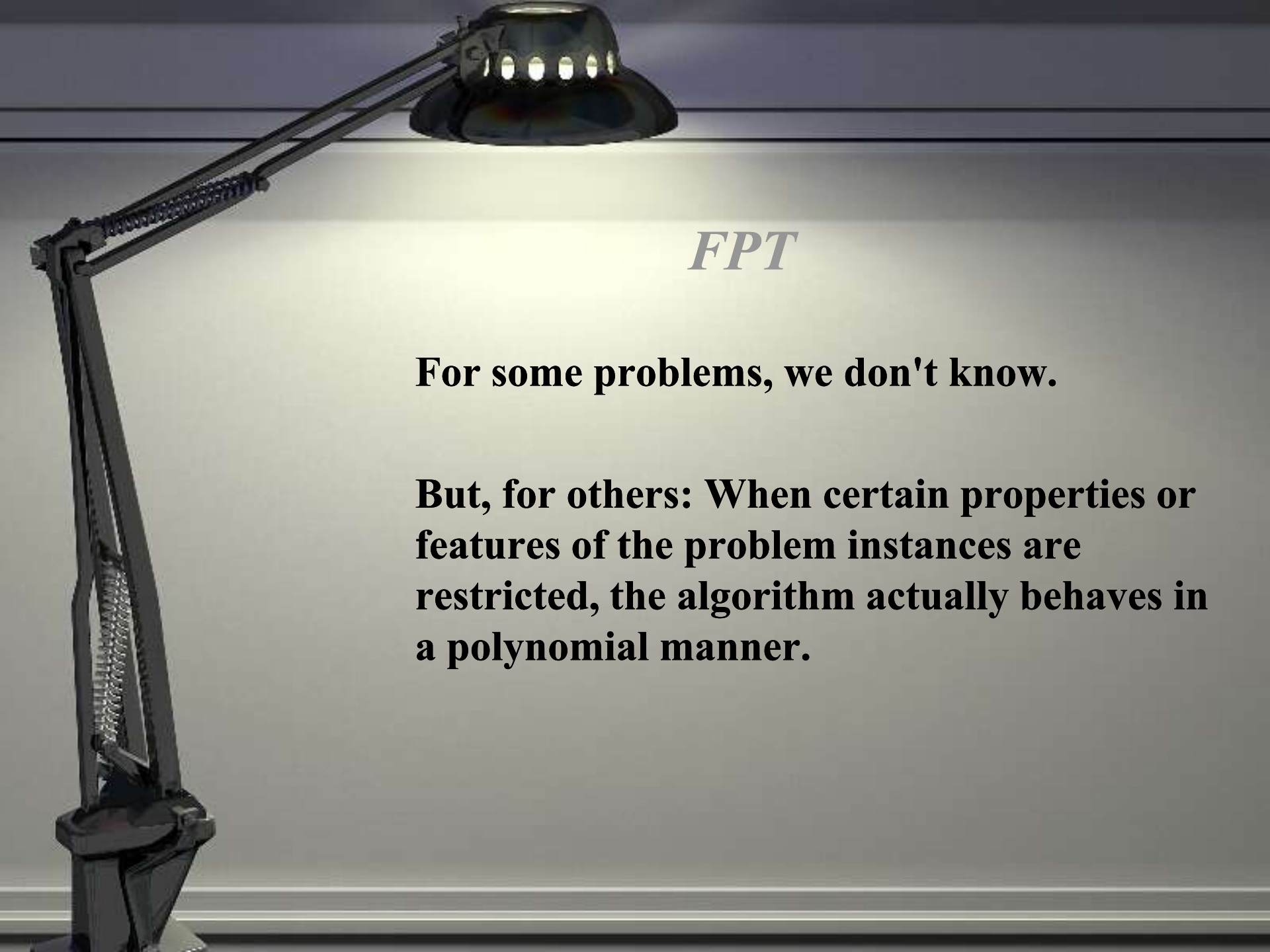


Class FPT (Fixed Parameter Tractable)

Unless P = NP, all NP–Hard problems have only exponential algorithms.

That is, $O(2^n)$ where n is the size of the instance. This is essentially: "generate and test each possible solution."

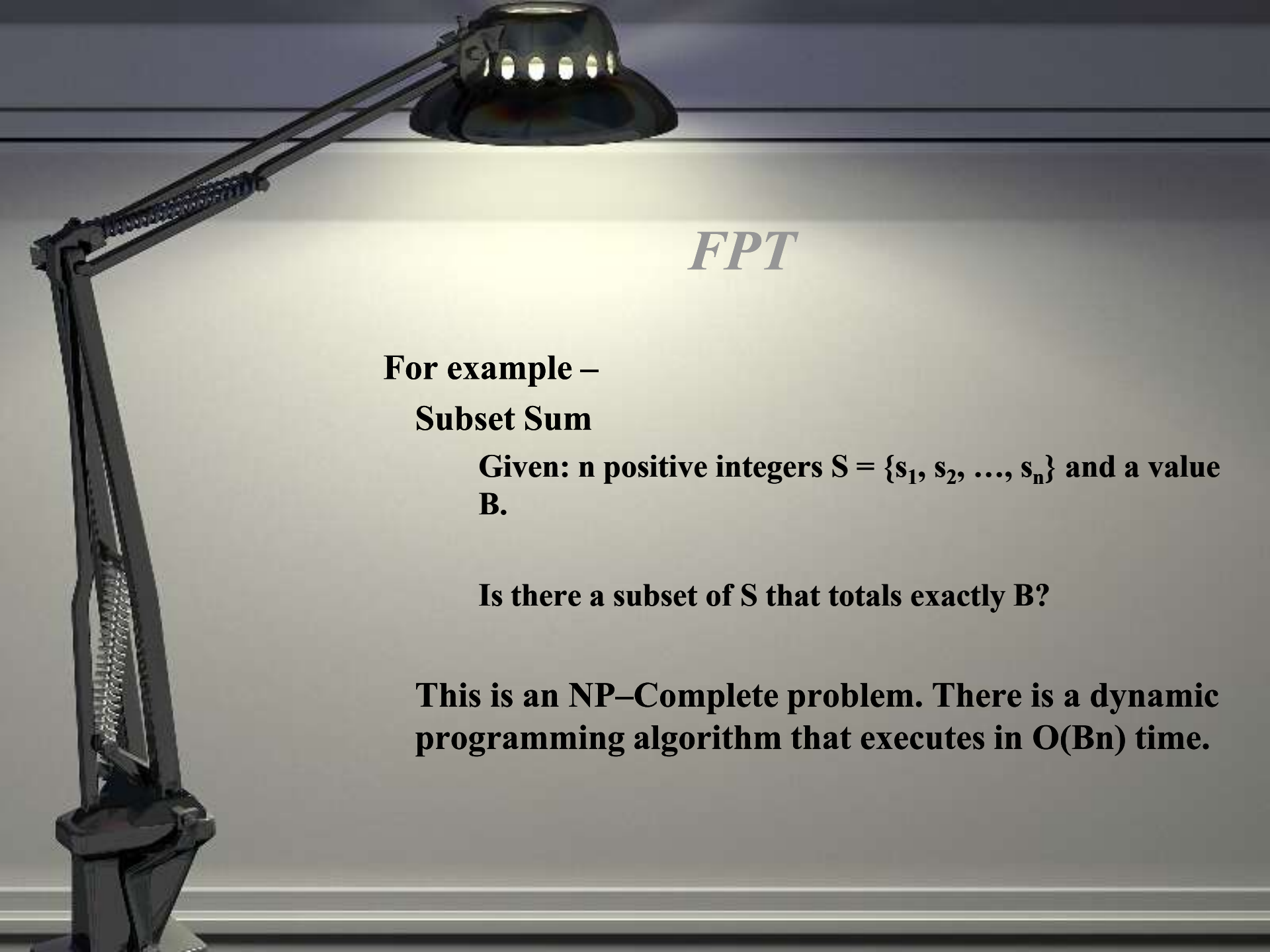

FPT
On the other hand, there are documented cases of algorithms for some of these problems that work surprisingly well for many, if not most, instances.
Why?



FPT

For some problems, we don't know.

But, for others: When certain properties or features of the problem instances are restricted, the algorithm actually behaves in a polynomial manner.



FPT

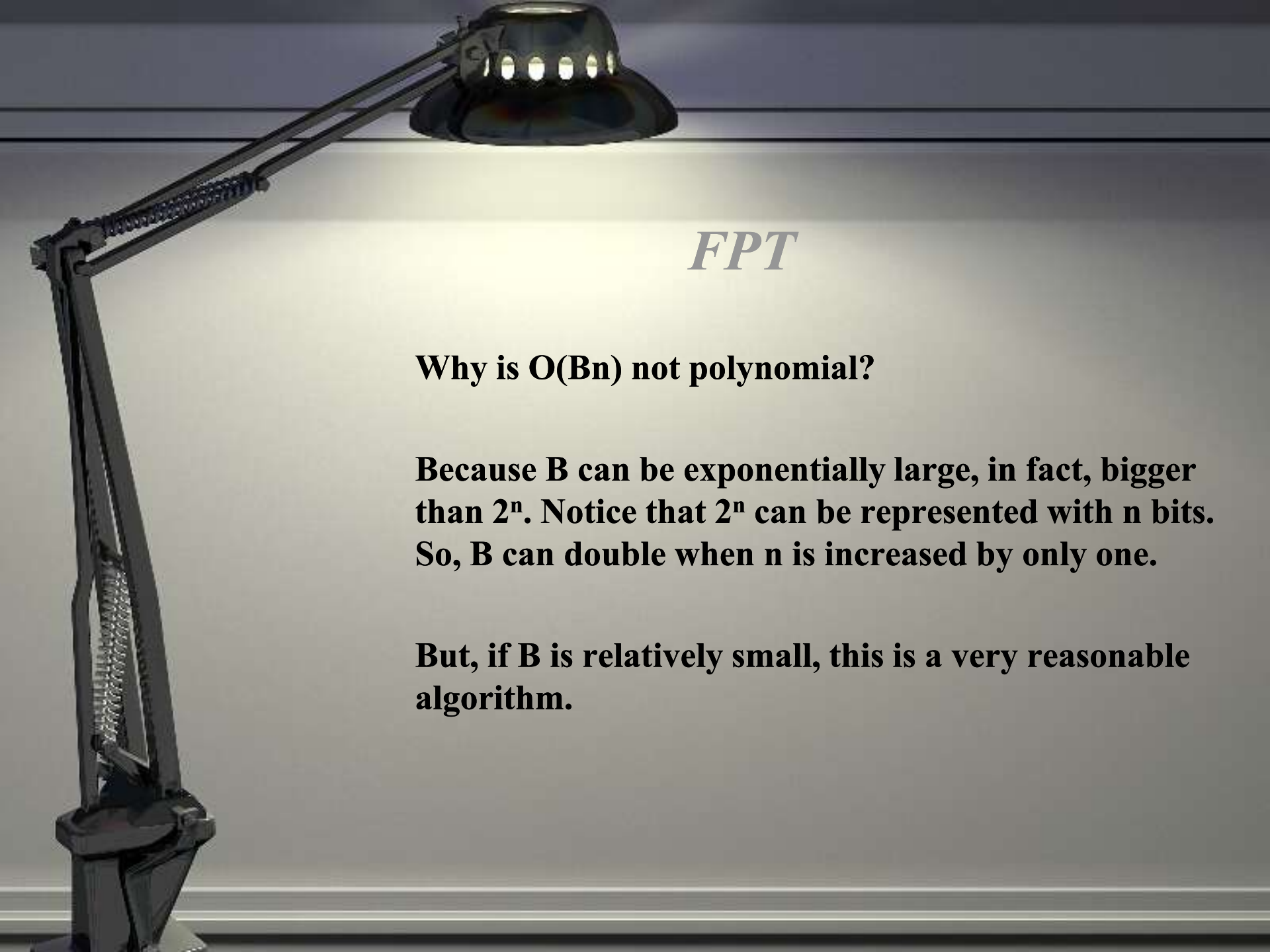
For example –

Subset Sum

Given: n positive integers S = {s_1, s_2, …, s_n} and a value B.

Is there a subset of S that totals exactly B?

This is an NP–Complete problem. There is a dynamic programming algorithm that executes in O(Bn) time.

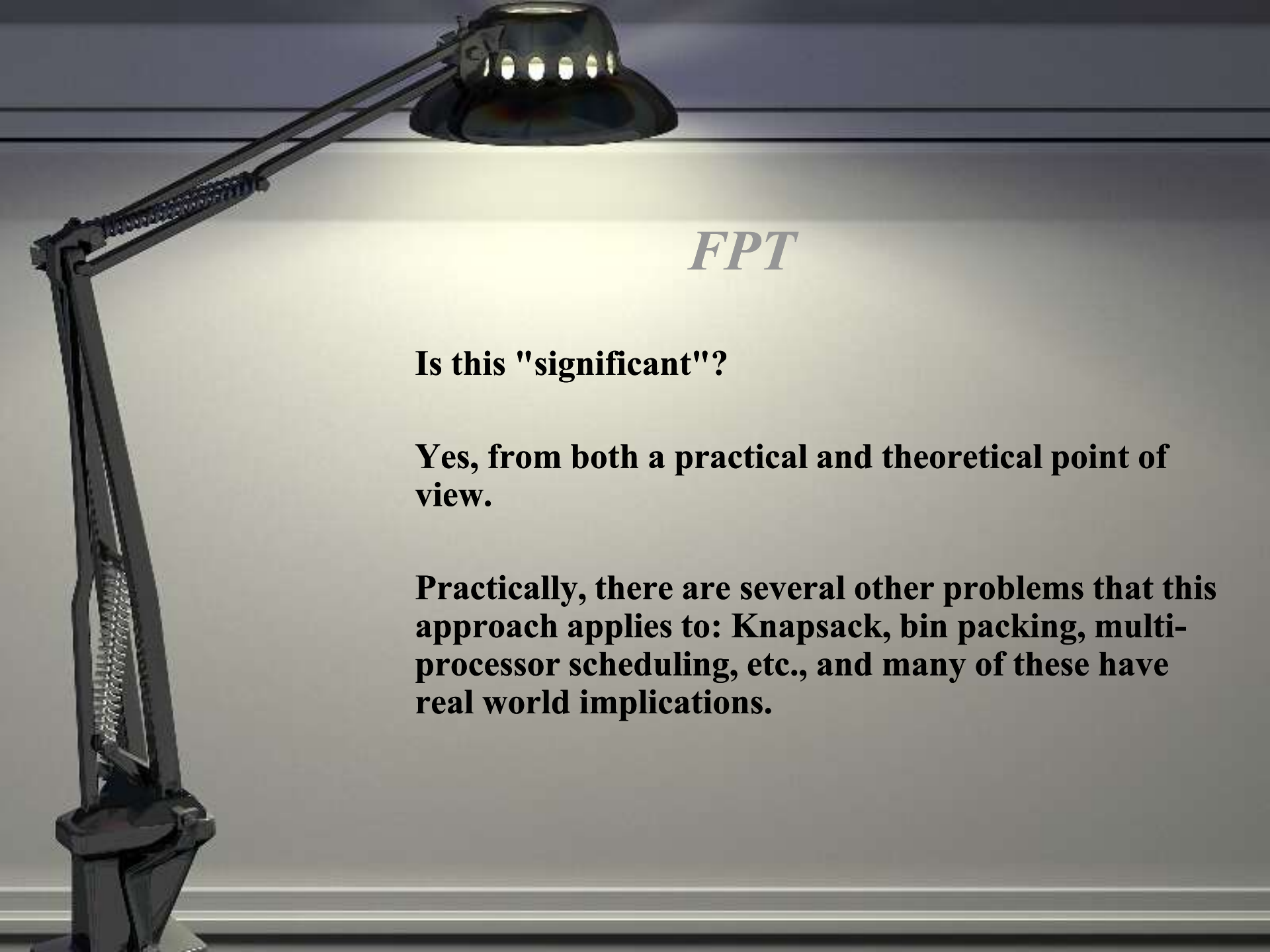


FPT

Why is O(Bn) not polynomial?

Because B can be exponentially large, in fact, bigger than 2^n. Notice that 2^n can be represented with n bits. So, B can double when n is increased by only one.

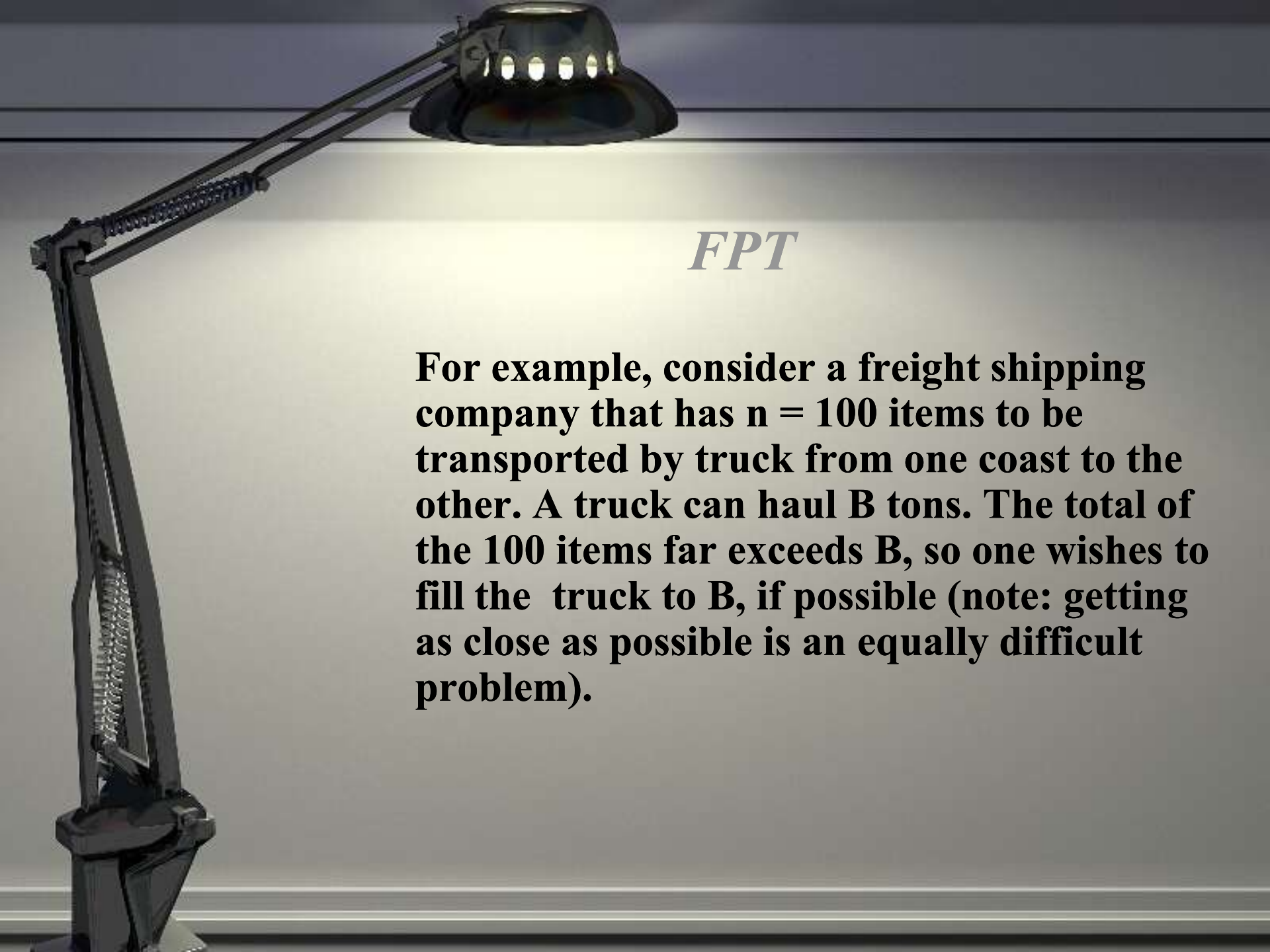
But, if B is relatively small, this is a very reasonable algorithm.



Is this "significant"?

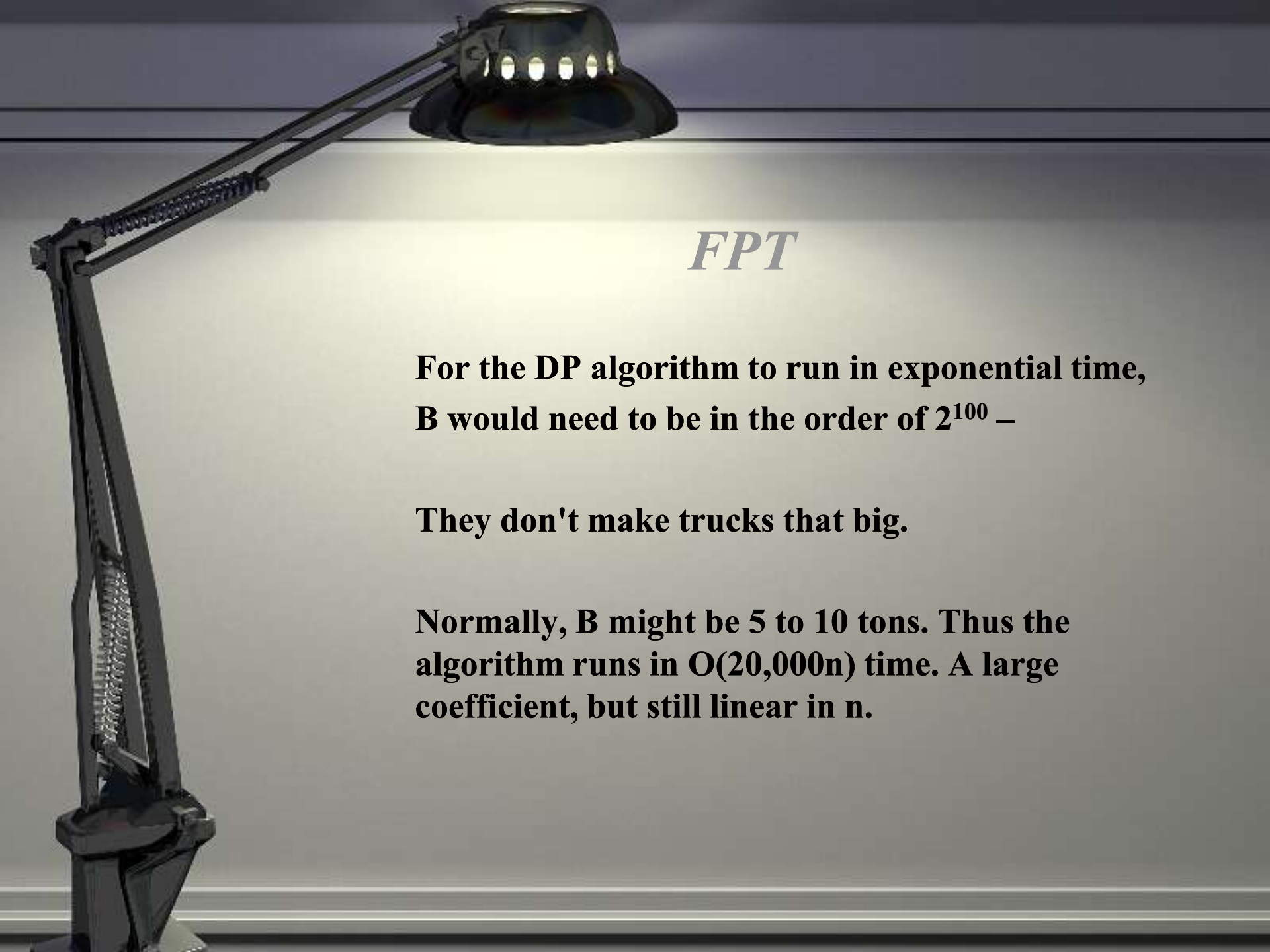
Yes, from both a practical and theoretical point of view.

Practically, there are several other problems that this approach applies to: Knapsack, bin packing, multi-processor scheduling, etc., and many of these have real world implications.



FPT

For example, consider a freight shipping company that has n = 100 items to be transported by truck from one coast to the other. A truck can haul B tons. The total of the 100 items far exceeds B, so one wishes to fill the truck to B, if possible (note: getting as close as possible is an equally difficult problem).

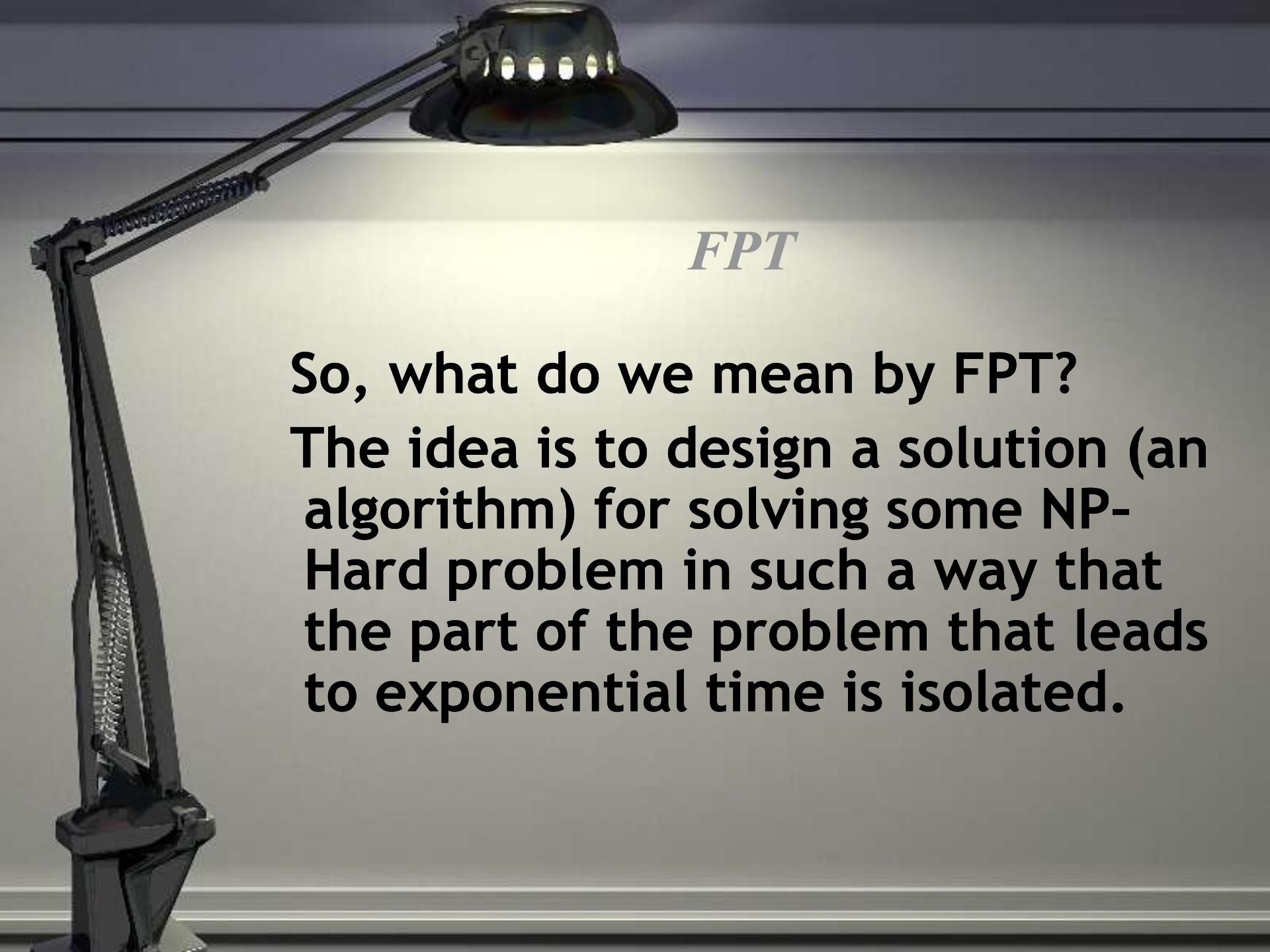


FPT

For the DP algorithm to run in exponential time, B would need to be in the order of 2^{100} –

They don't make trucks that big.

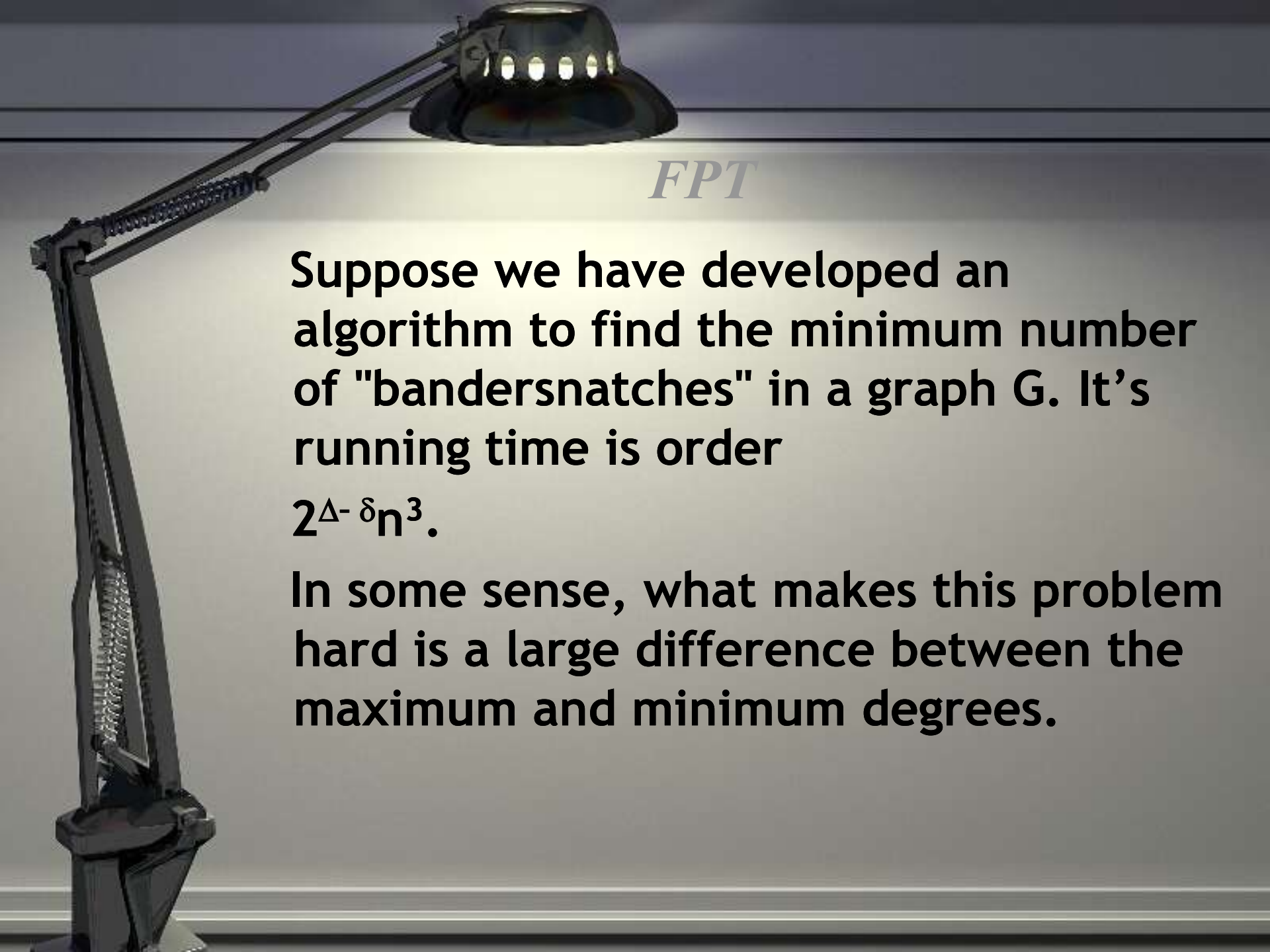
Normally, B might be 5 to 10 tons. Thus the algorithm runs in O(20,000n) time. A large coefficient, but still linear in n.



FPT

So, what do we mean by FPT?

The idea is to design a solution (an algorithm) for solving some NP-Hard problem in such a way that the part of the problem that leads to exponential time is isolated.

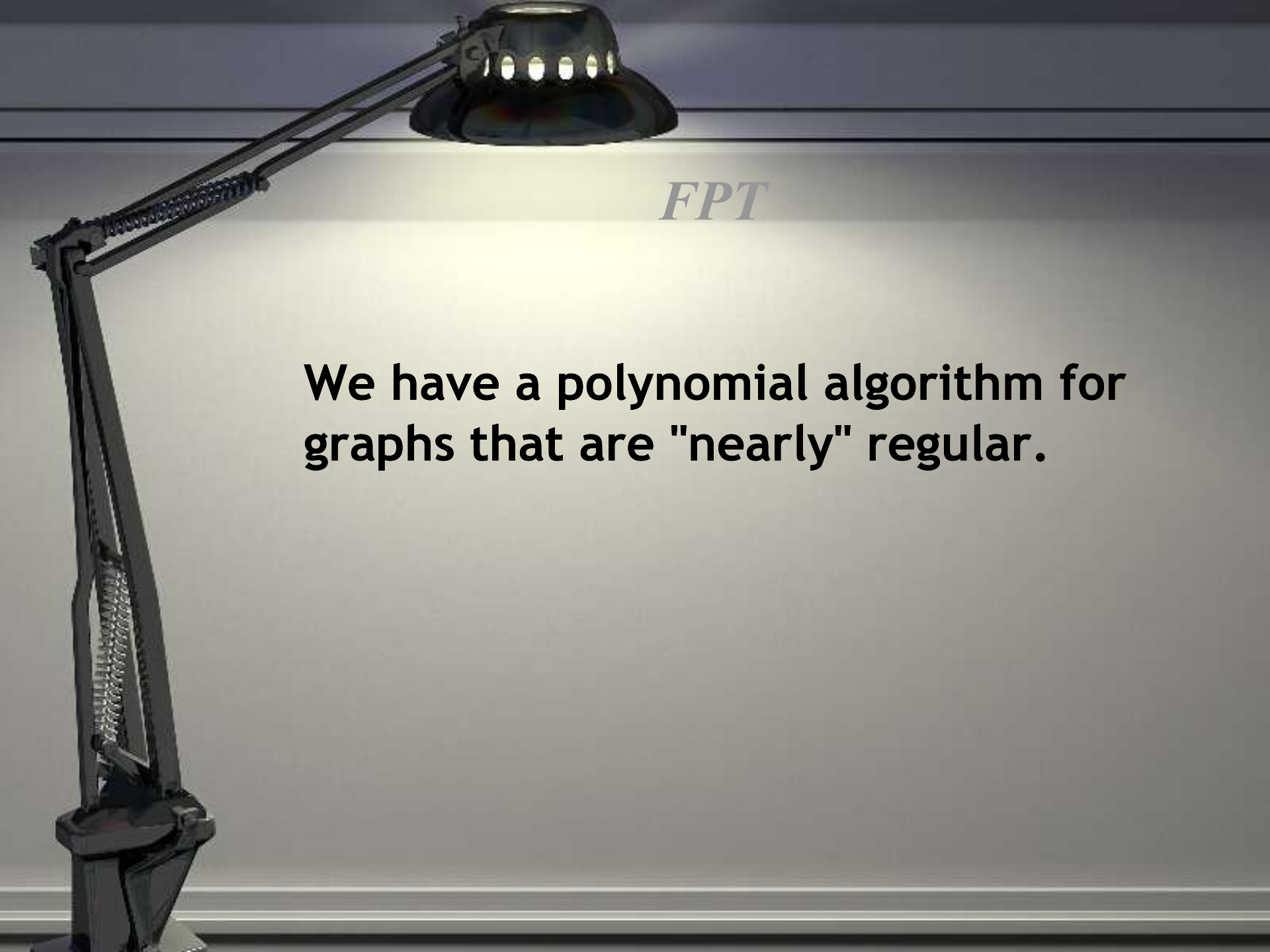


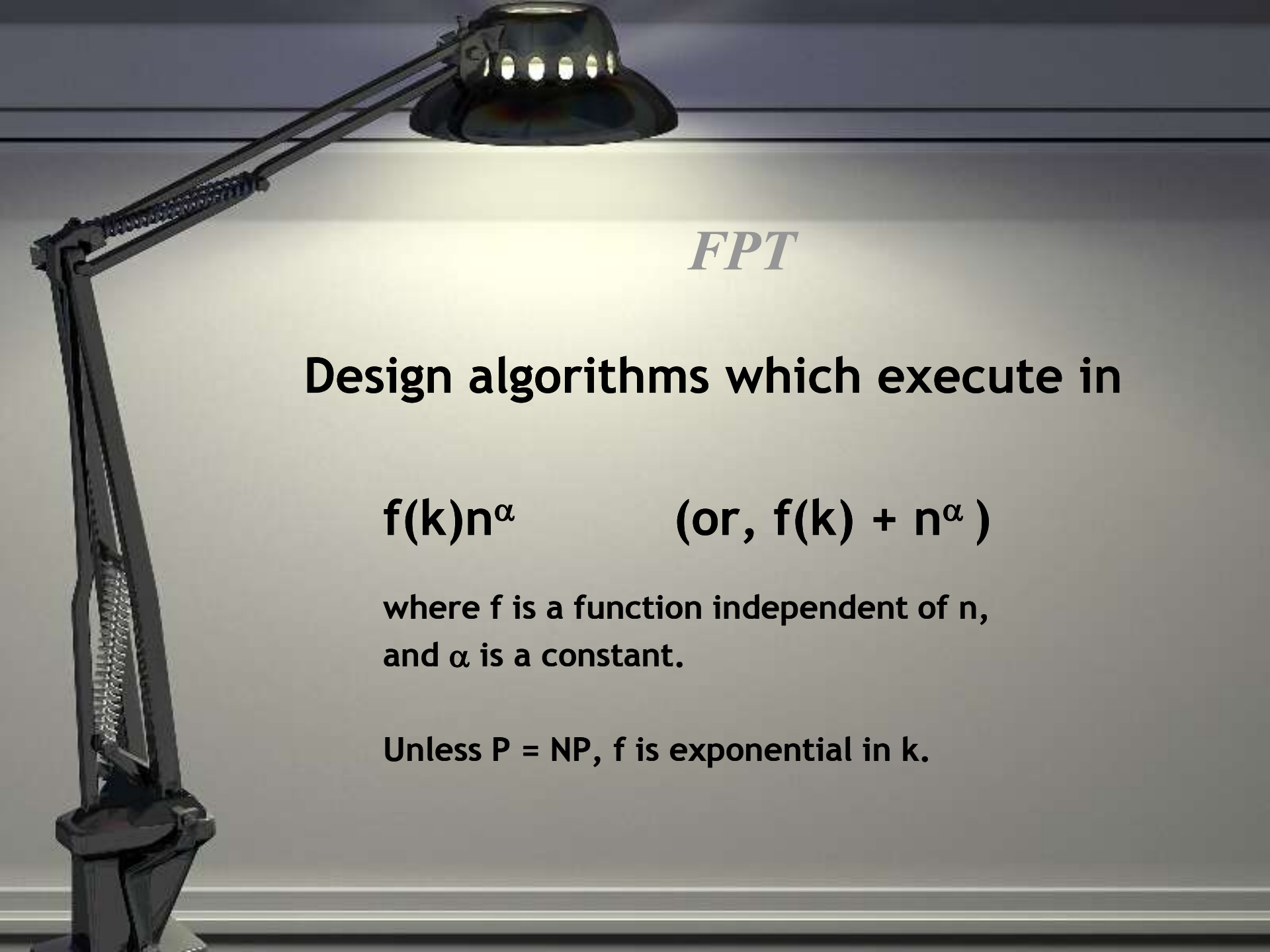
FPT

Suppose we have developed an algorithm to find the minimum number of "bandersnatches" in a graph G. It's running time is order

$2^{\Delta - \delta} n^3$.

In some sense, what makes this problem hard is a large difference between the maximum and minimum degrees.


FPT
We have a polynomial algorithm for graphs that are "nearly" regular.



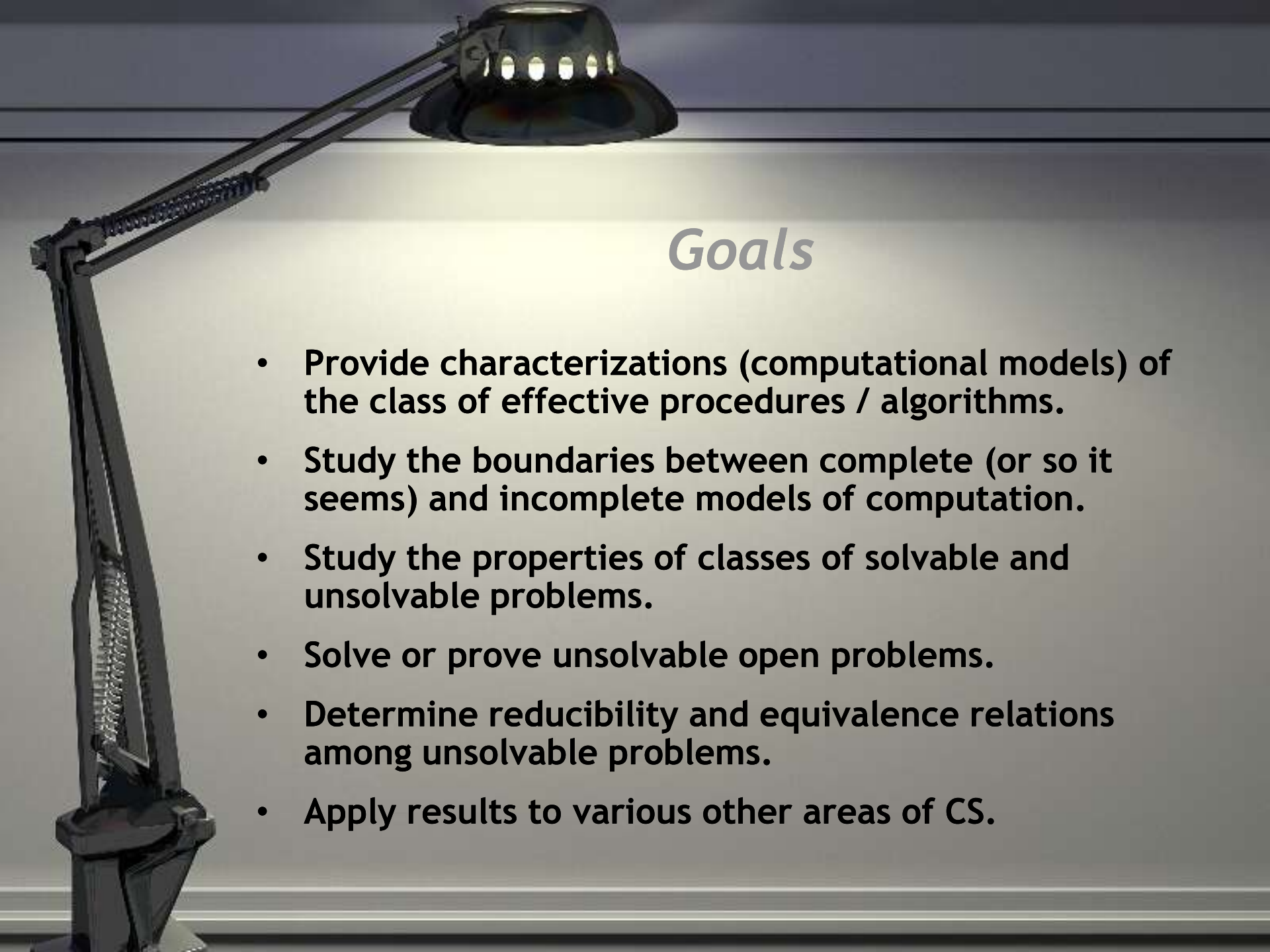
Design algorithms which execute in

$f(k)n^{\alpha}$ (or, $f(k) + n^{\alpha}$)

where f is a function independent of n, and α is a constant.

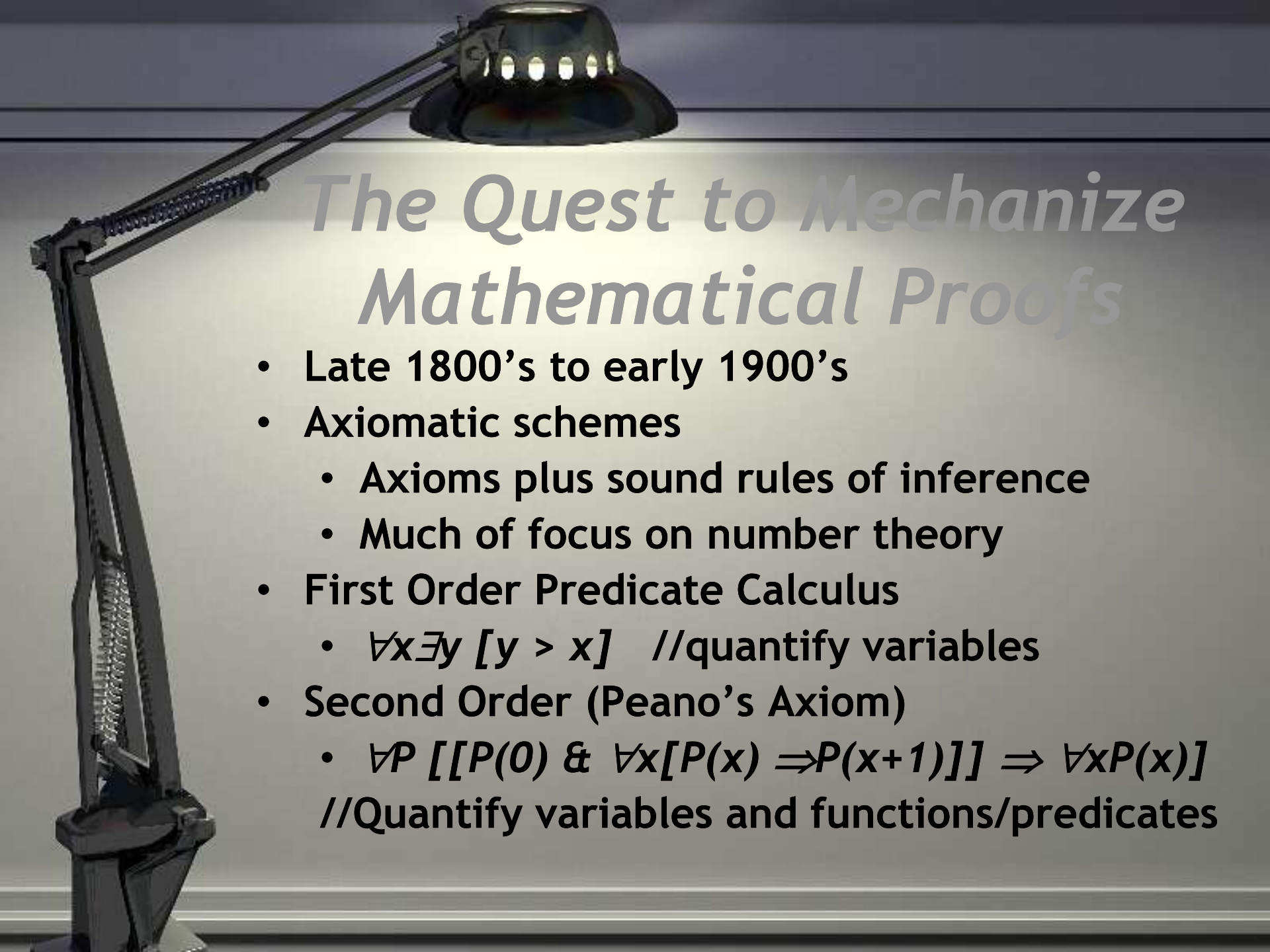
Unless P = NP, f is exponential in k.


Computability
Theory



Goals

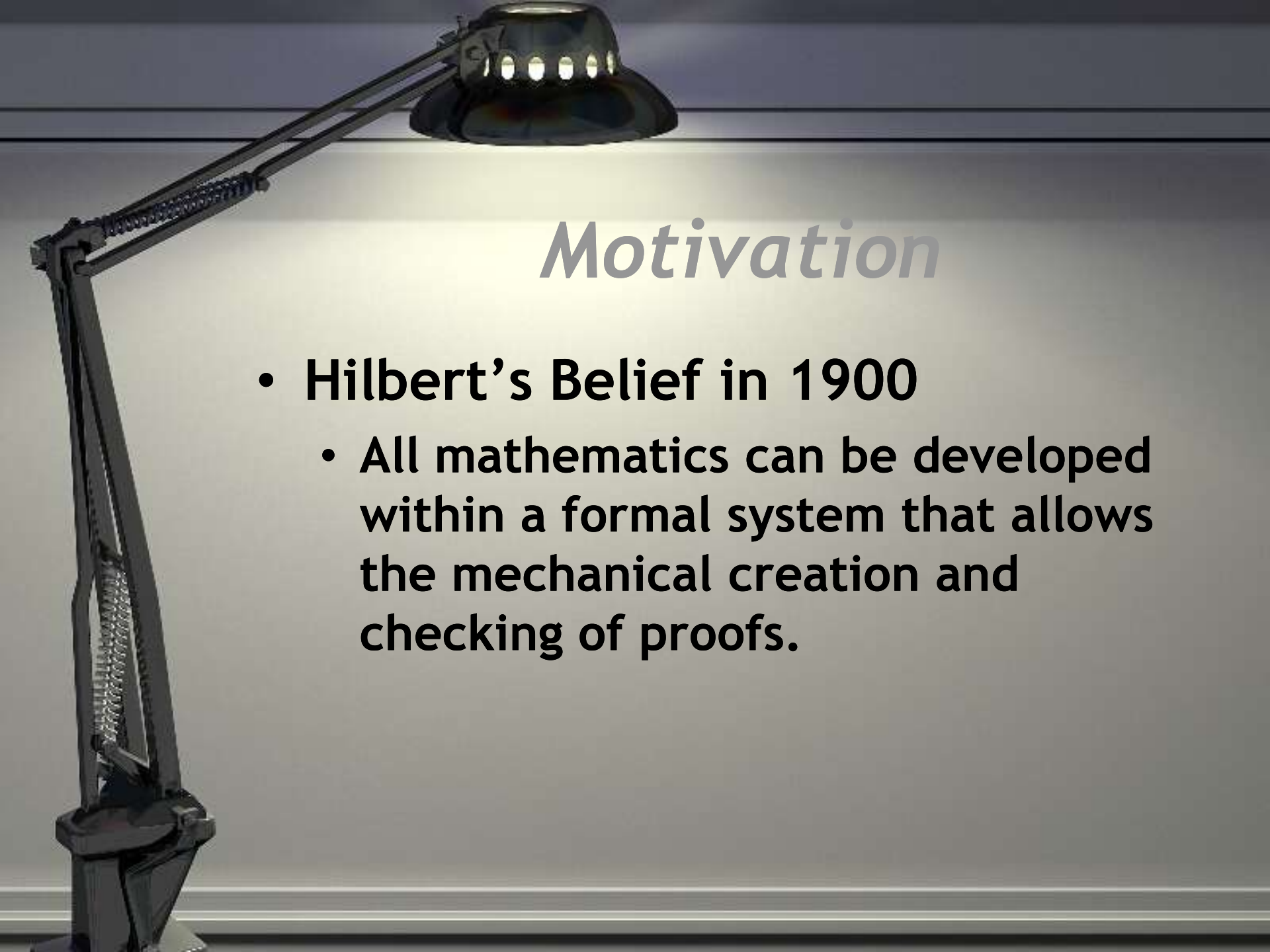
- Provide characterizations (computational models) of the class of effective procedures / algorithms.
- Study the boundaries between complete (or so it seems) and incomplete models of computation.
- Study the properties of classes of solvable and unsolvable problems.
- Solve or prove unsolvable open problems.
- Determine reducibility and equivalence relations among unsolvable problems.
- Apply results to various other areas of CS.



The Quest to Mechanize Mathematical Proofs

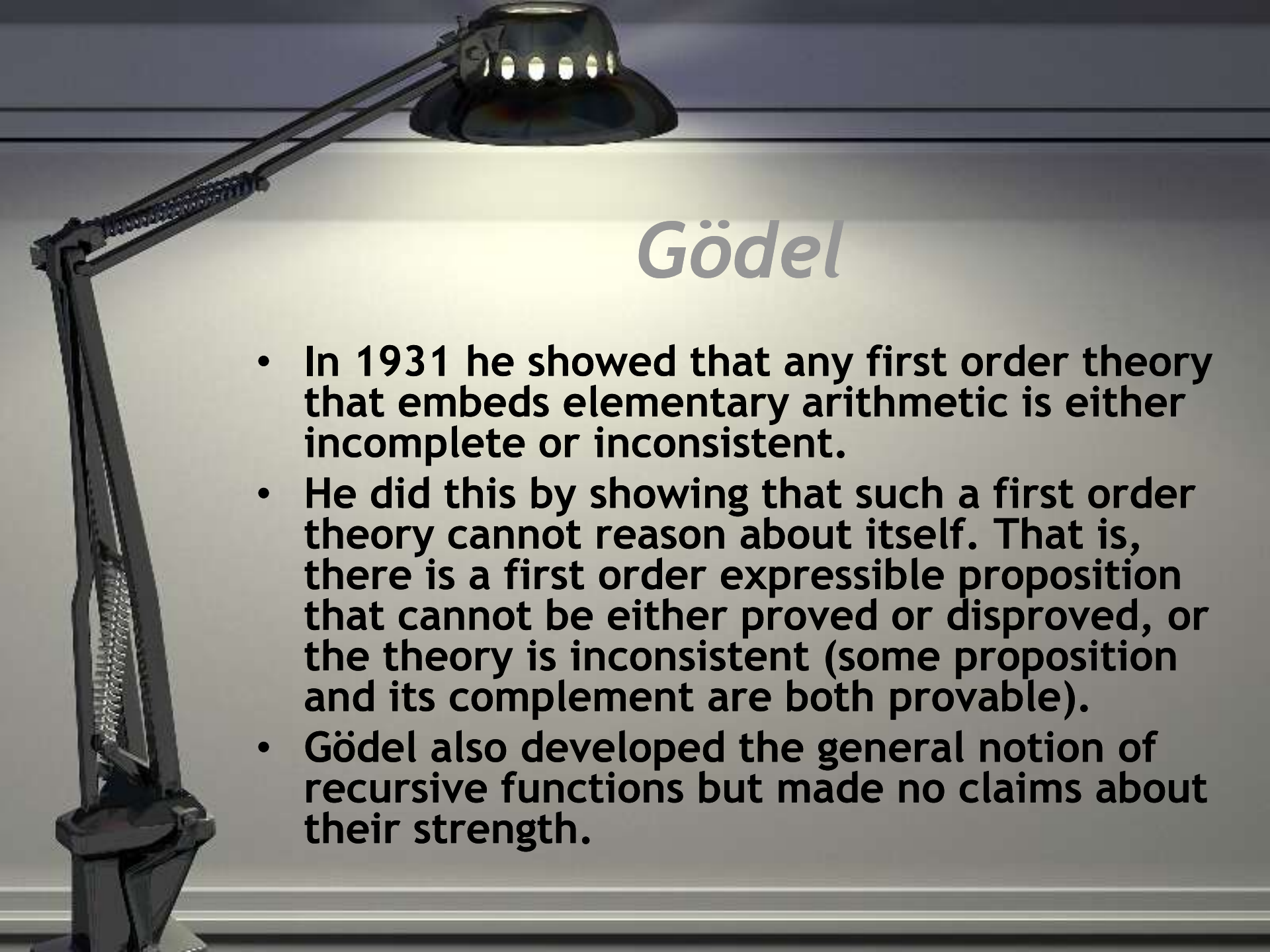
- Late 1800's to early 1900's
- Axiomatic schemes
  - Axioms plus sound rules of inference
  - Much of focus on number theory
- First Order Predicate Calculus
  - $\forall x \exists y\ [y > x]$ //quantify variables
- Second Order (Peano's Axiom)
  - $\forall P\ [[P(0)\ \&\ \forall x[P(x) \Rightarrow P(x+1)]] \Rightarrow \forall x P(x)]$

  //Quantify variables and functions/predicates



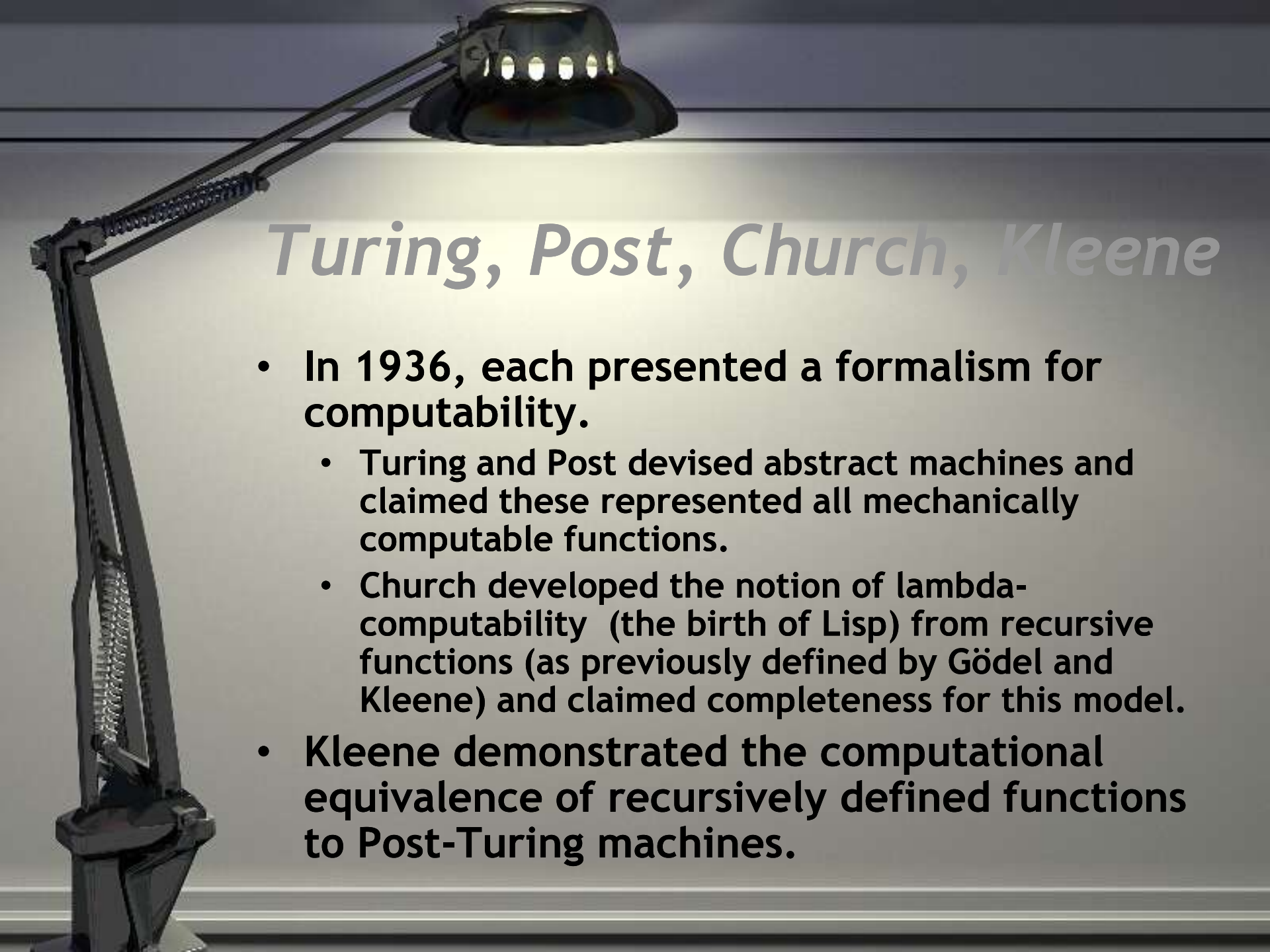
Motivation

- **Hilbert's Belief in 1900**
  - **All mathematics can be developed within a formal system that allows the mechanical creation and checking of proofs.**



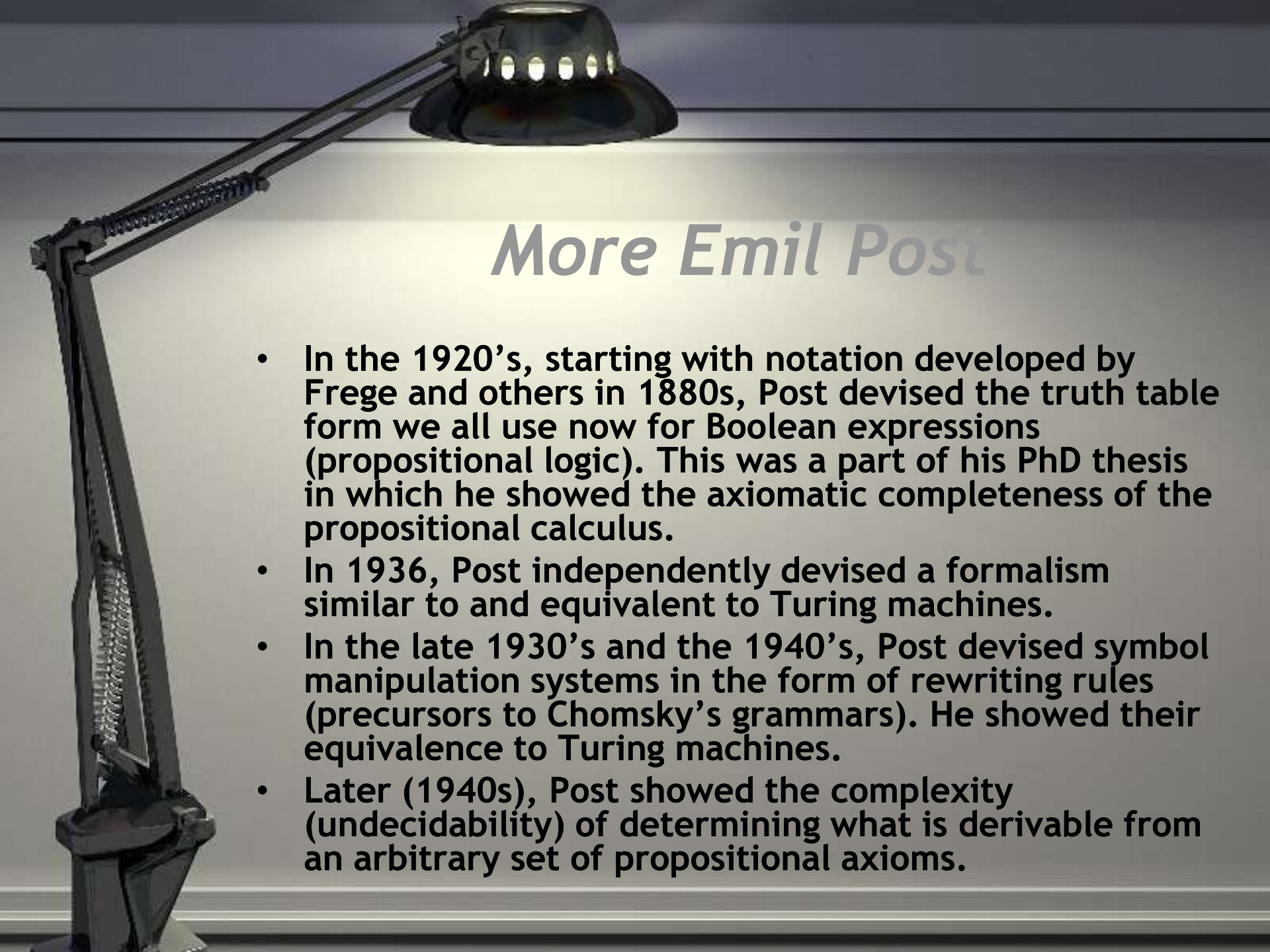
Gödel

- In 1931 he showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- He did this by showing that such a first order theory cannot reason about itself. That is, there is a first order expressible proposition that cannot be either proved or disproved, or the theory is inconsistent (some proposition and its complement are both provable).
- Gödel also developed the general notion of recursive functions but made no claims about their strength.



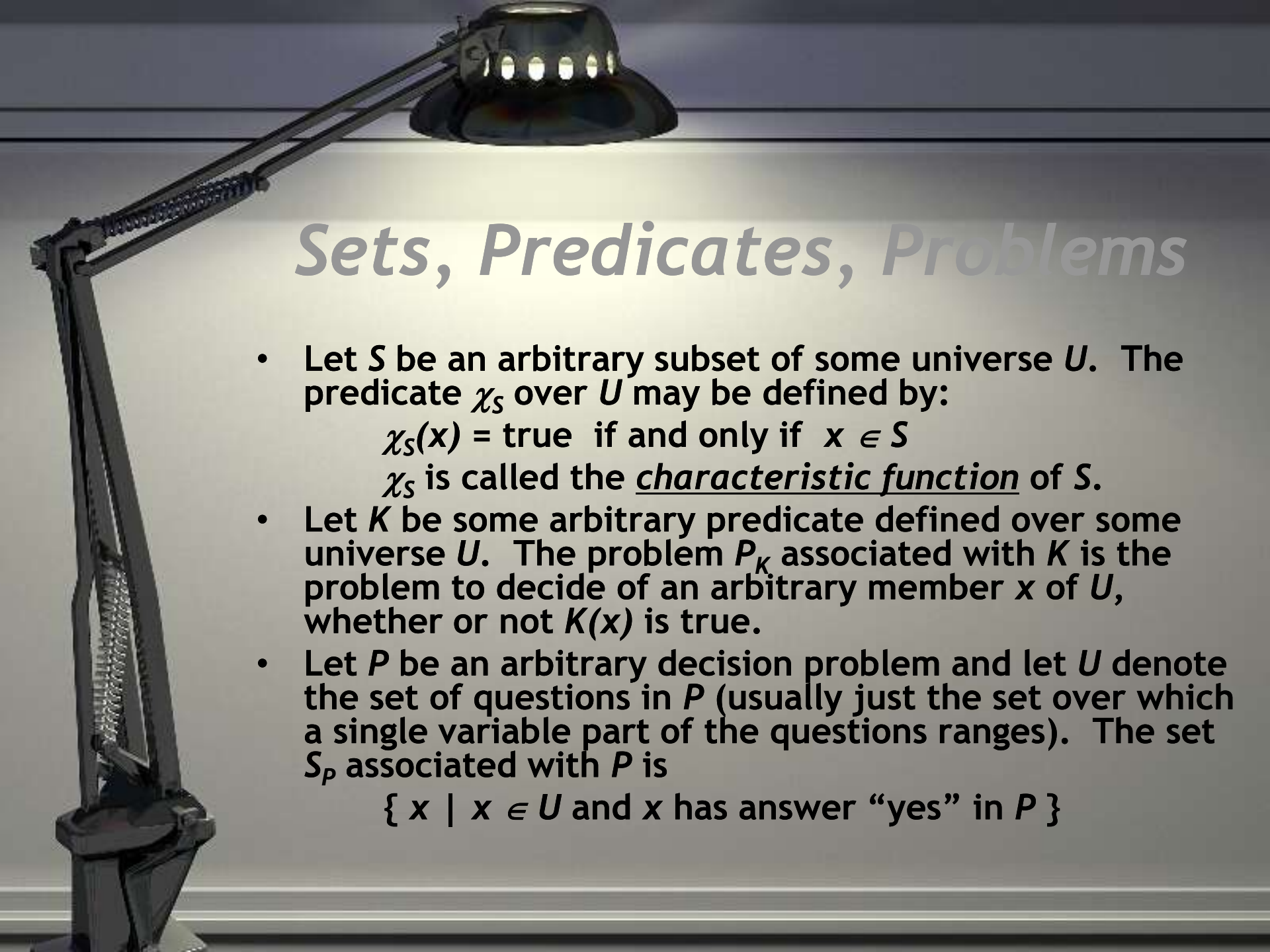
Turing, Post, Church, Kleene

- In 1936, each presented a formalism for computability.
  - Turing and Post devised abstract machines and claimed these represented all mechanically computable functions.
  - Church developed the notion of lambda-computability (the birth of Lisp) from recursive functions (as previously defined by Gödel and Kleene) and claimed completeness for this model.
- Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.



More Emil Post

- In the 1920's, starting with notation developed by Frege and others in 1880s, Post devised the truth table form we all use now for Boolean expressions (propositional logic). This was a part of his PhD thesis in which he showed the axiomatic completeness of the propositional calculus.
- In 1936, Post independently devised a formalism similar to and equivalent to Turing machines.
- In the late 1930's and the 1940's, Post devised symbol manipulation systems in the form of rewriting rules (precursors to Chomsky's grammars). He showed their equivalence to Turing machines.
- Later (1940s), Post showed the complexity (undecidability) of determining what is derivable from an arbitrary set of propositional axioms.



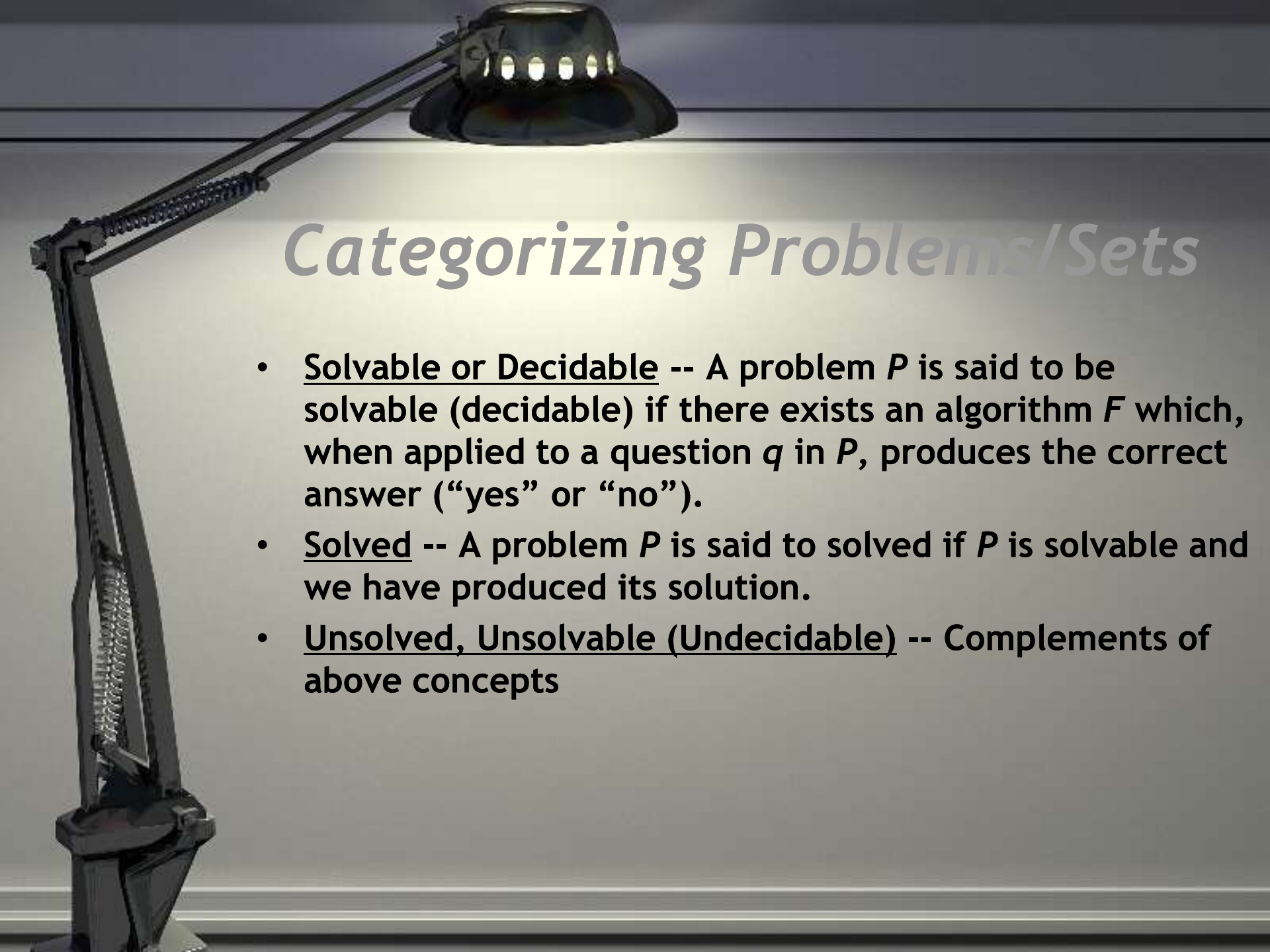
Sets, Predicates, Problems

- Let S be an arbitrary subset of some universe U. The predicate χ_S over U may be defined by:

  $\chi_S(x)$ = true if and only if $x \in S$

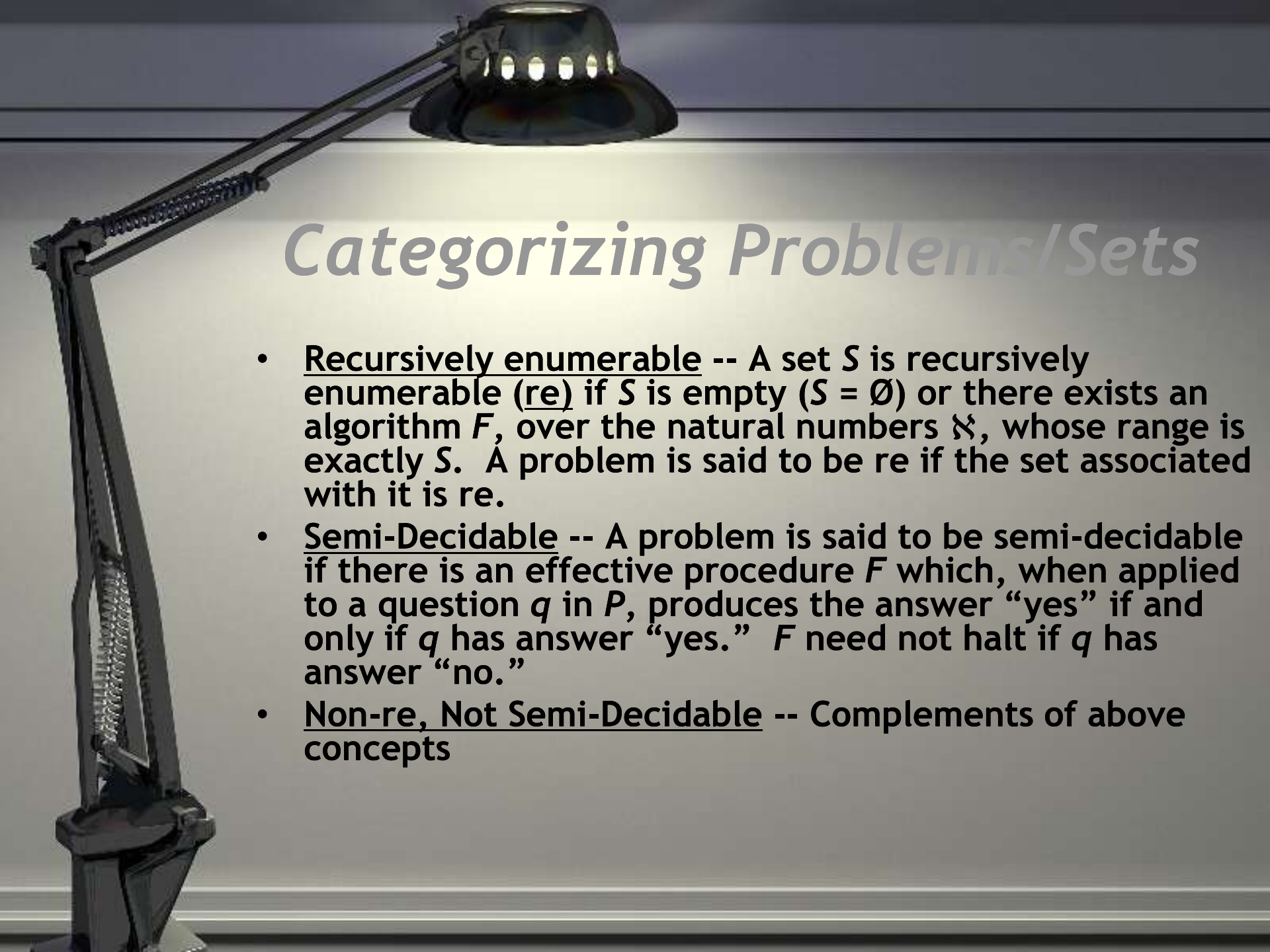
  χ_S is called the *characteristic function* of S.
- Let K be some arbitrary predicate defined over some universe U. The problem P_K associated with K is the problem to decide of an arbitrary member x of U, whether or not $K(x)$ is true.
- Let P be an arbitrary decision problem and let U denote the set of questions in P (usually just the set over which a single variable part of the questions ranges). The set S_P associated with P is

  $\{ x \mid x \in U$ and x has answer "yes" in $P \}$



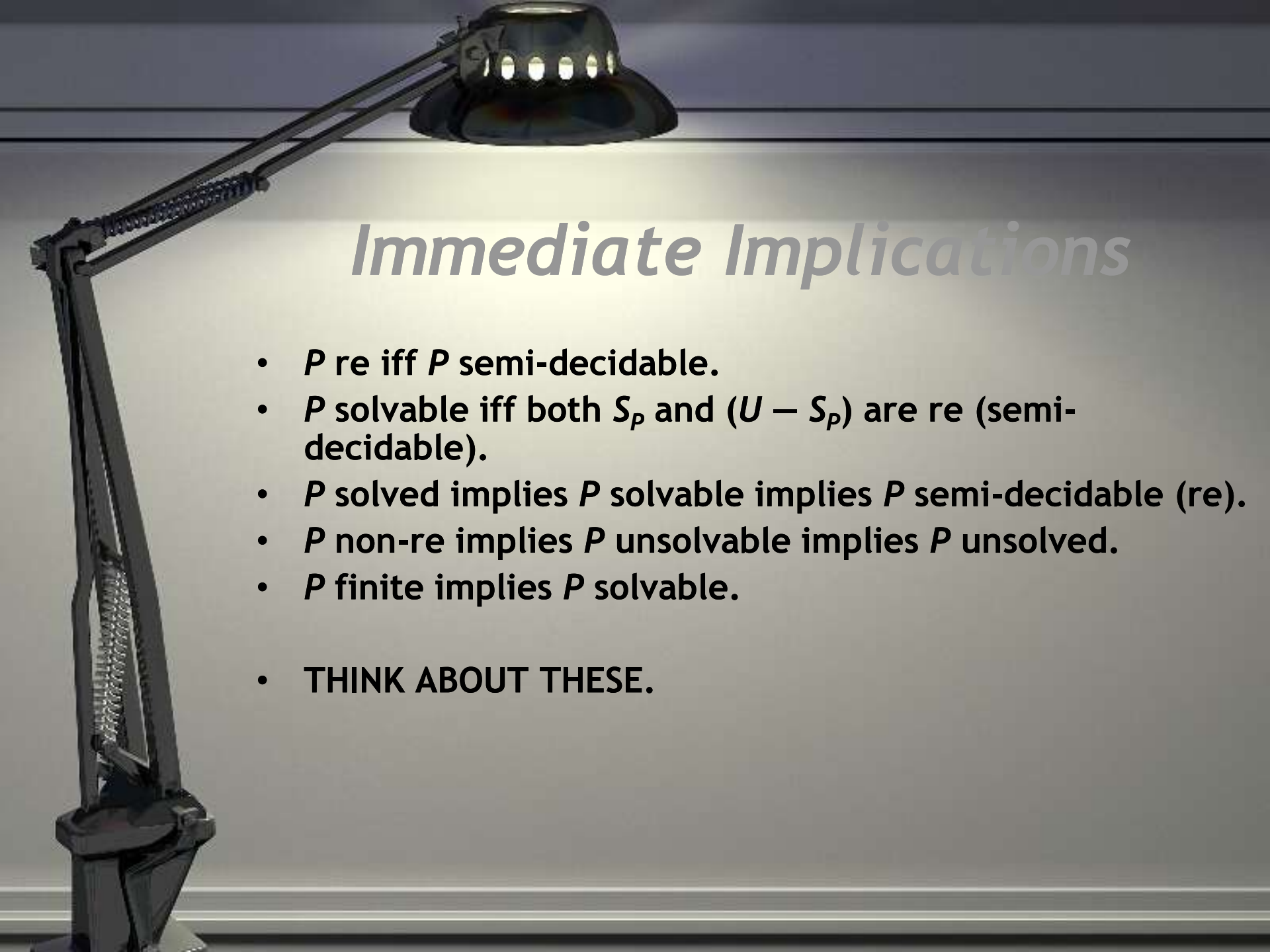
Categorizing Problems/Sets

- Solvable or Decidable -- A problem *P* is said to be solvable (decidable) if there exists an algorithm *F* which, when applied to a question *q* in *P*, produces the correct answer ("yes" or "no").
- Solved -- A problem *P* is said to solved if *P* is solvable and we have produced its solution.
- Unsolved, Unsolvable (Undecidable) -- Complements of above concepts



Categorizing Problems/Sets

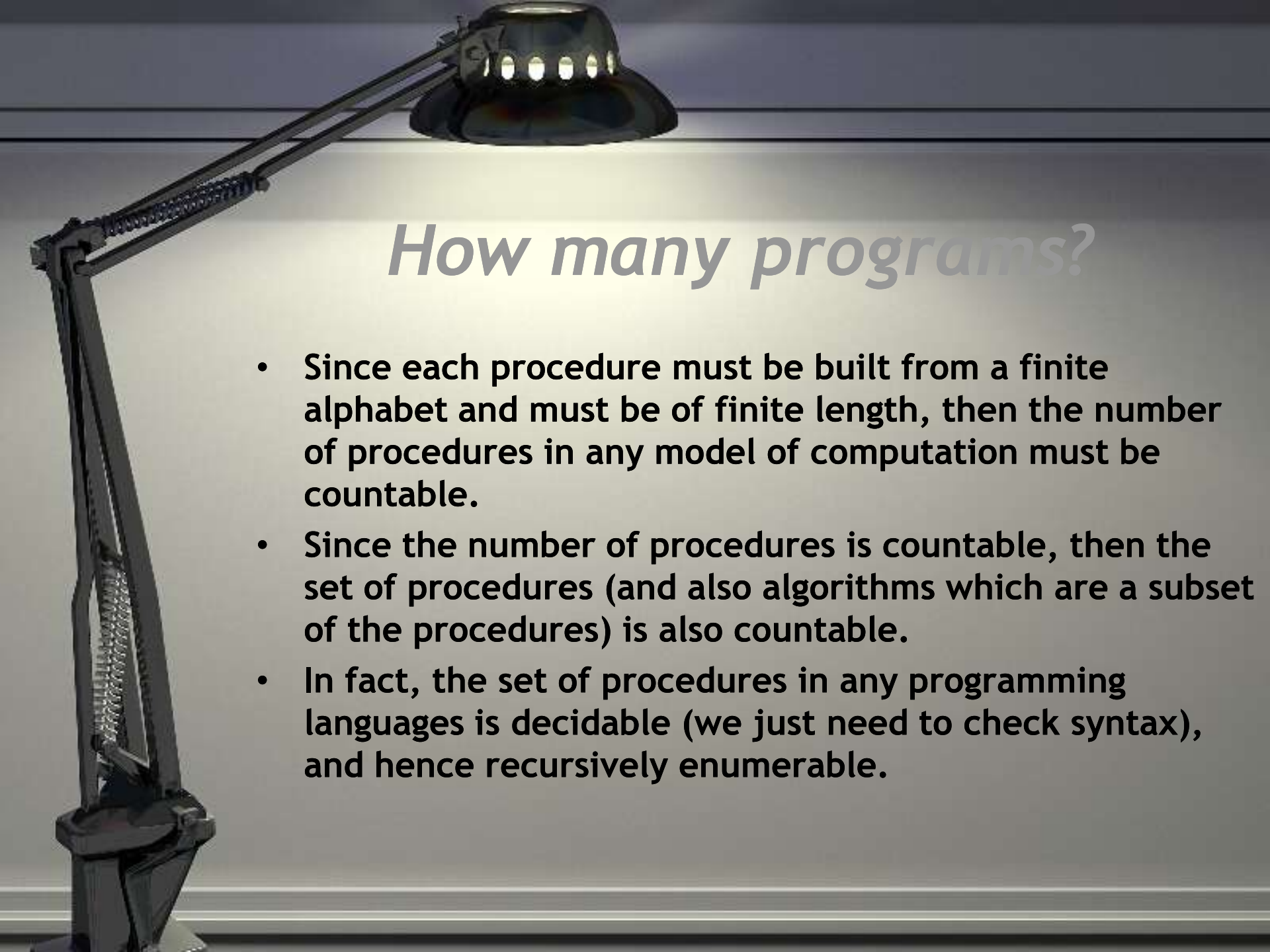
- Recursively enumerable -- A set *S* is recursively enumerable (re) if *S* is empty (*S* = Ø) or there exists an algorithm *F*, over the natural numbers ℵ, whose range is exactly *S*. A problem is said to be re if the set associated with it is re.
- Semi-Decidable -- A problem is said to be semi-decidable if there is an effective procedure *F* which, when applied to a question *q* in *P*, produces the answer "yes" if and only if *q* has answer "yes." *F* need not halt if *q* has answer "no."
- Non-re, Not Semi-Decidable -- Complements of above concepts



Immediate Implications

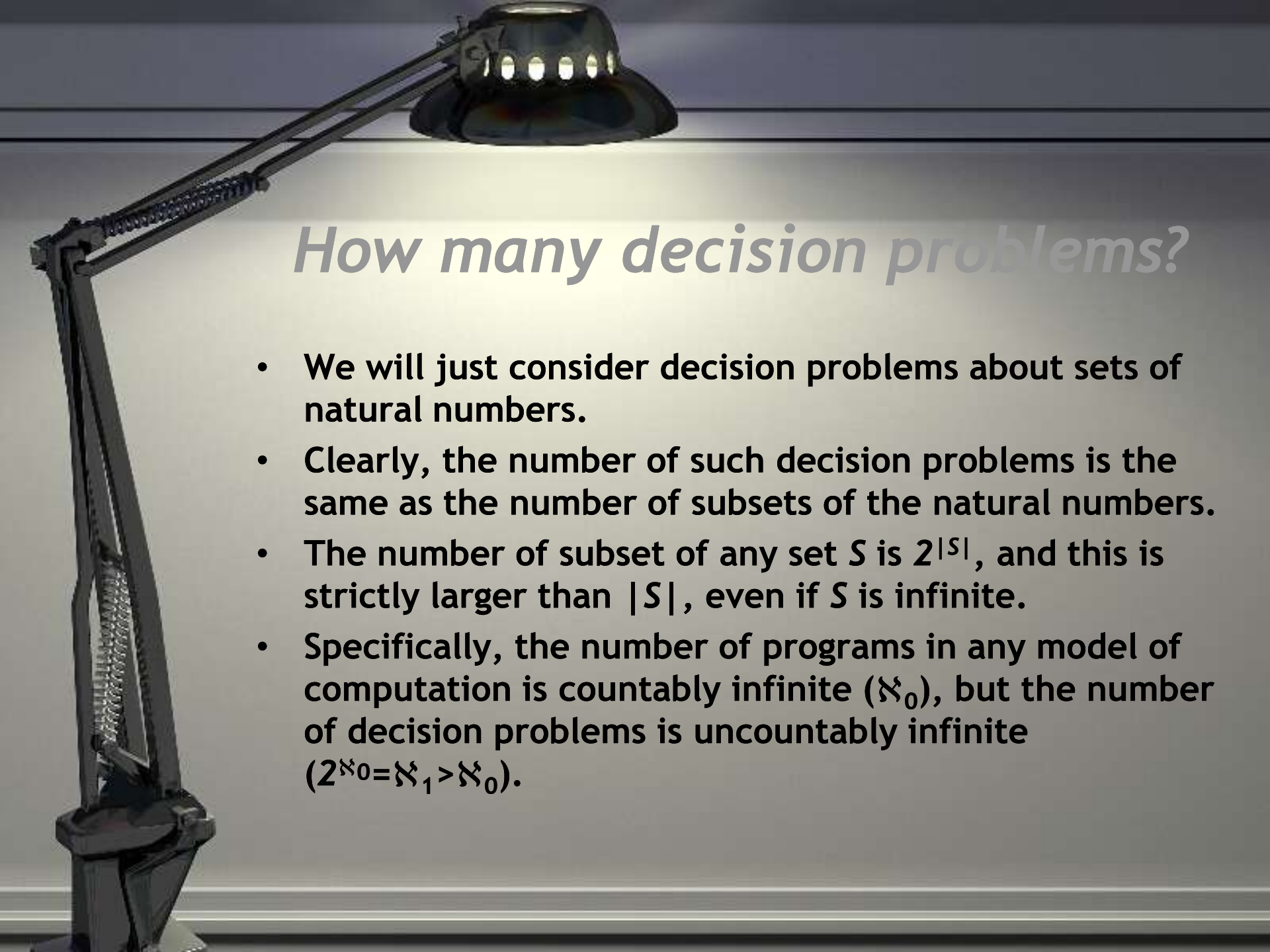
- *P* re iff *P* semi-decidable.
- *P* solvable iff both S_P and ($U - S_P$) are re (semi-decidable).
- *P* solved implies *P* solvable implies *P* semi-decidable (re).
- *P* non-re implies *P* unsolvable implies *P* unsolved.
- *P* finite implies *P* solvable.

- THINK ABOUT THESE.



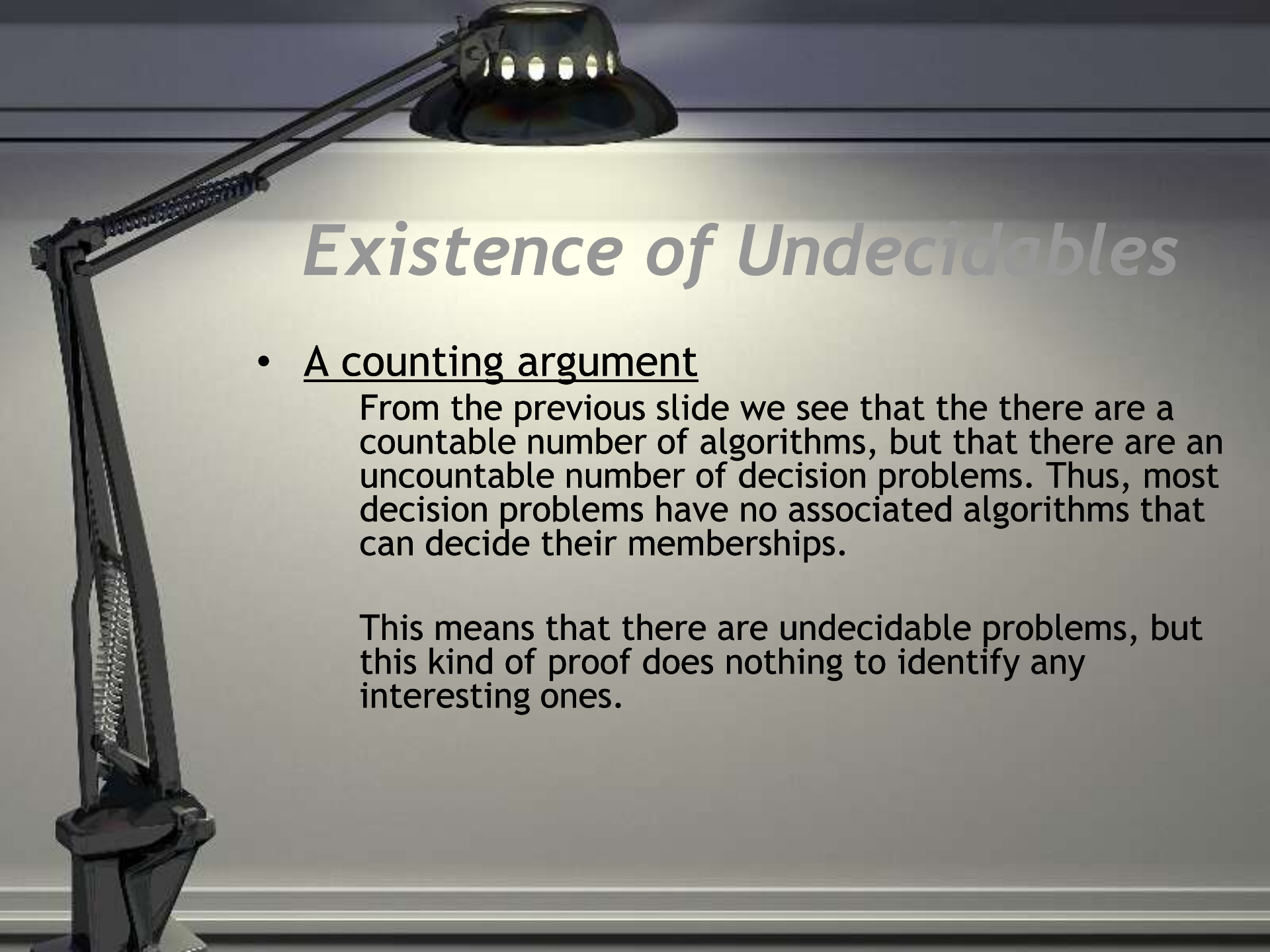
How many programs?

- Since each procedure must be built from a finite alphabet and must be of finite length, then the number of procedures in any model of computation must be countable.
- Since the number of procedures is countable, then the set of procedures (and also algorithms which are a subset of the procedures) is also countable.
- In fact, the set of procedures in any programming languages is decidable (we just need to check syntax), and hence recursively enumerable.



How many decision problems?

- We will just consider decision problems about sets of natural numbers.
- Clearly, the number of such decision problems is the same as the number of subsets of the natural numbers.
- The number of subset of any set S is $2^{|S|}$, and this is strictly larger than $|S|$, even if S is infinite.
- Specifically, the number of programs in any model of computation is countably infinite ($\aleph_0$), but the number of decision problems is uncountably infinite ($2^{\aleph_0}=\aleph_1>\aleph_0$).

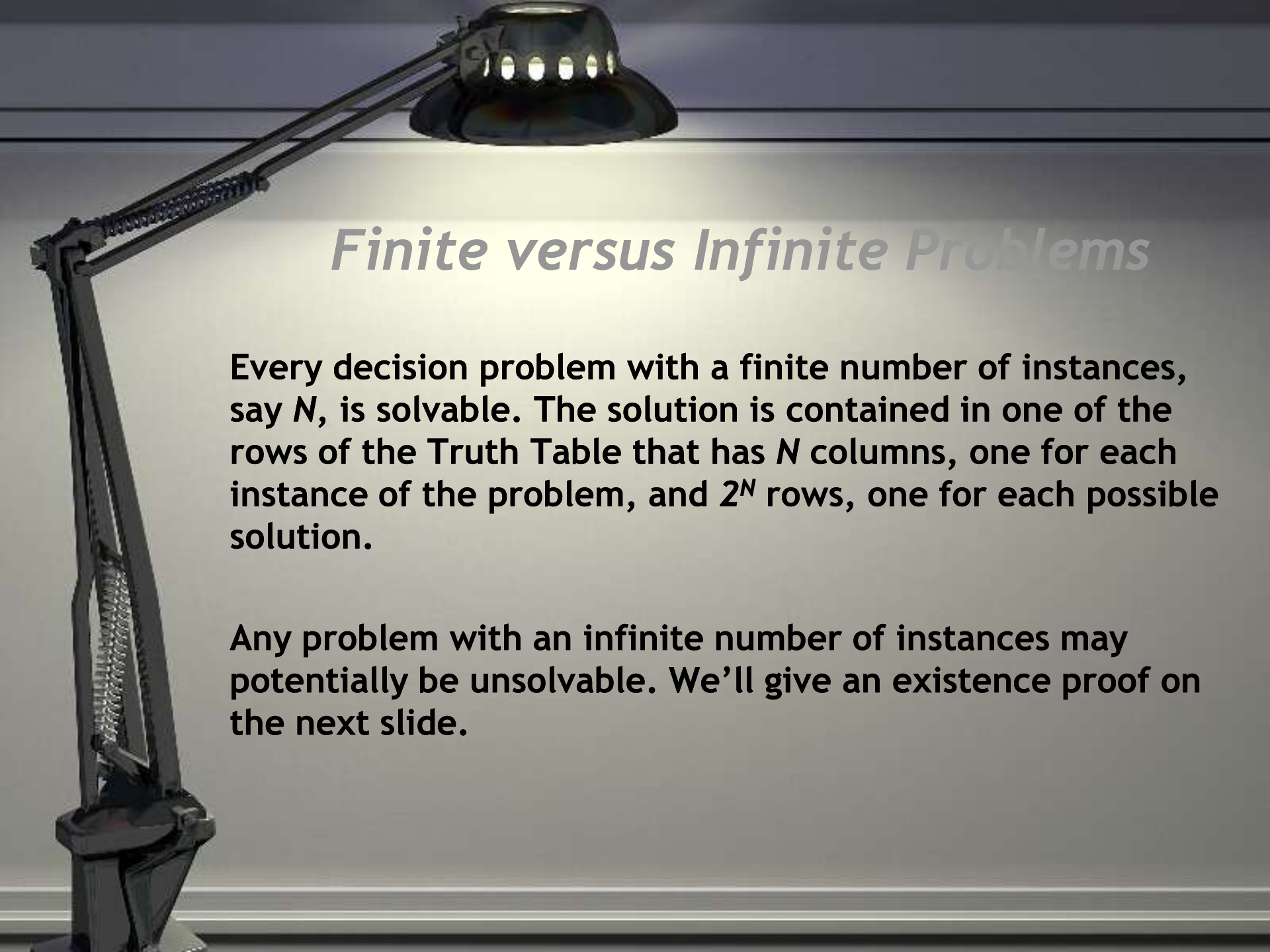


Existence of Undecidables

- A counting argument

  From the previous slide we see that the there are a countable number of algorithms, but that there are an uncountable number of decision problems. Thus, most decision problems have no associated algorithms that can decide their memberships.

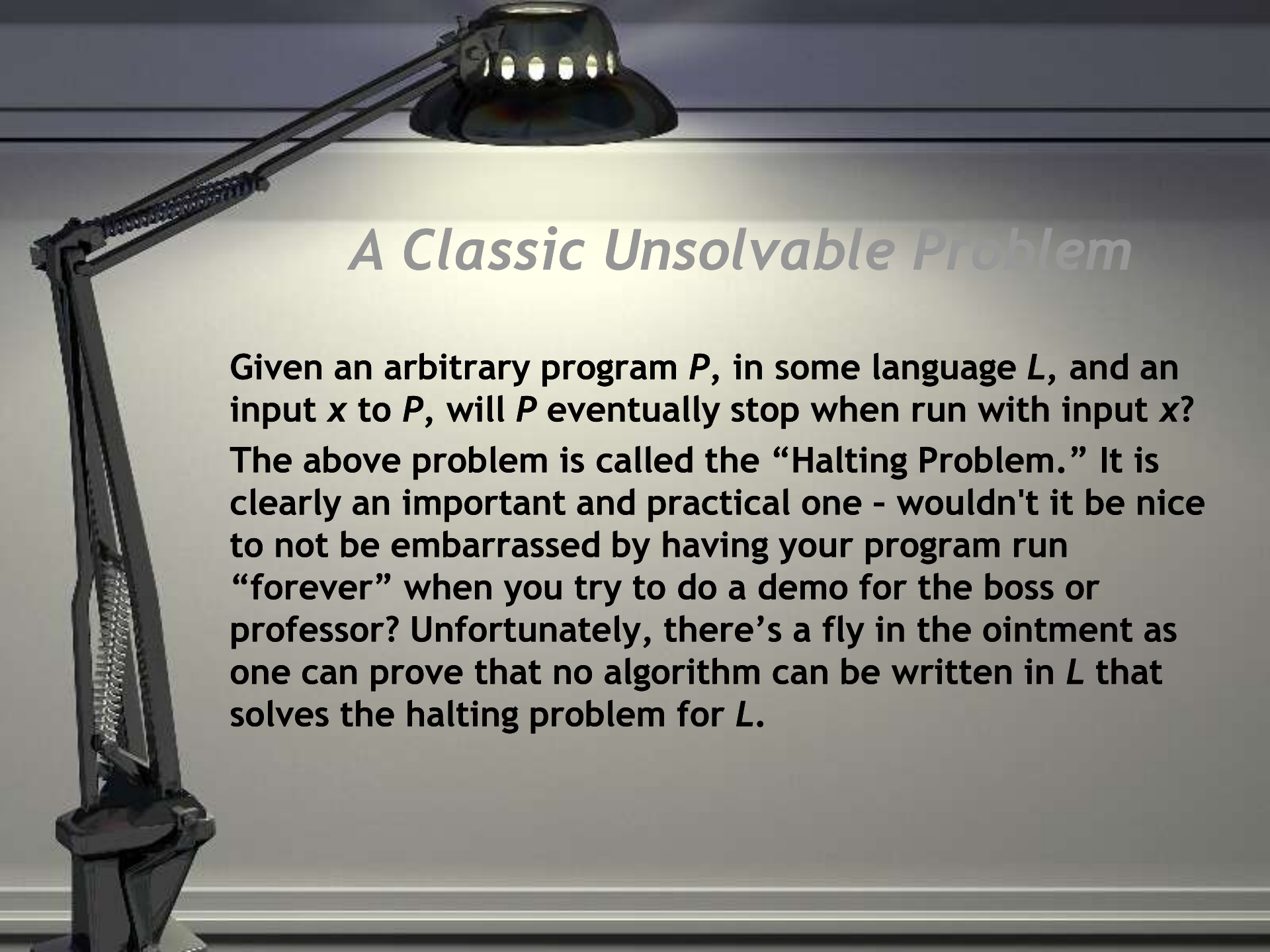
  This means that there are undecidable problems, but this kind of proof does nothing to identify any interesting ones.



Finite versus Infinite Problems

Every decision problem with a finite number of instances, say *N*, is solvable. The solution is contained in one of the rows of the Truth Table that has *N* columns, one for each instance of the problem, and 2^N rows, one for each possible solution.

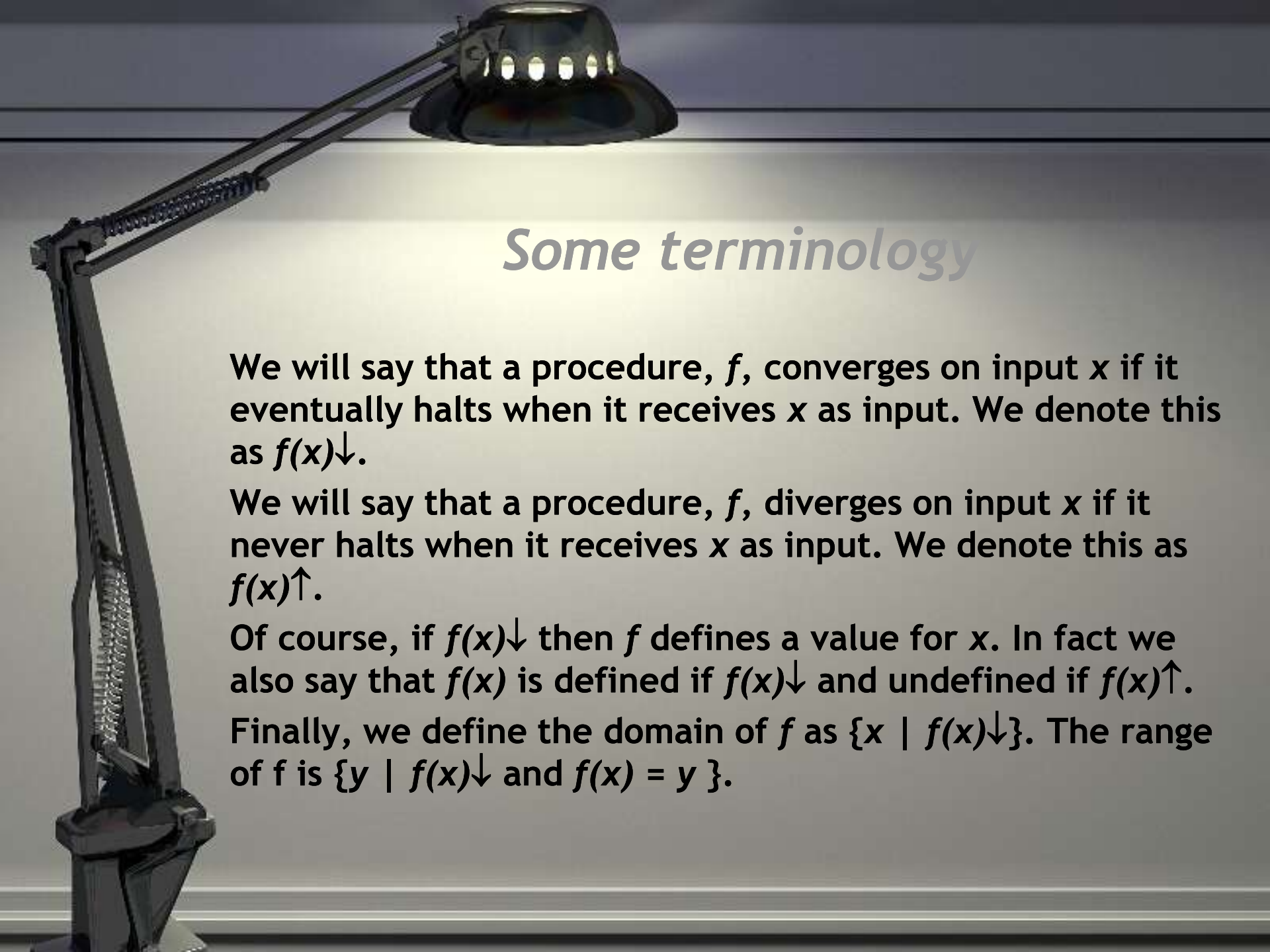
Any problem with an infinite number of instances may potentially be unsolvable. We'll give an existence proof on the next slide.



A Classic Unsolvable Problem

Given an arbitrary program *P*, in some language *L*, and an input *x* to *P*, will *P* eventually stop when run with input *x*?

The above problem is called the “Halting Problem.” It is clearly an important and practical one – wouldn't it be nice to not be embarrassed by having your program run “forever” when you try to do a demo for the boss or professor? Unfortunately, there’s a fly in the ointment as one can prove that no algorithm can be written in *L* that solves the halting problem for *L*.



Some terminology

We will say that a procedure, *f*, converges on input *x* if it eventually halts when it receives *x* as input. We denote this as $f(x)\downarrow$.

We will say that a procedure, *f*, diverges on input *x* if it never halts when it receives *x* as input. We denote this as $f(x)\uparrow$.

Of course, if $f(x)\downarrow$ then *f* defines a value for *x*. In fact we also say that *f(x)* is defined if $f(x)\downarrow$ and undefined if $f(x)\uparrow$.

Finally, we define the domain of *f* as $\{x \mid f(x)\downarrow\}$. The range of f is $\{y \mid f(x)\downarrow \text{ and } f(x) = y\}$.

Halting Problem

Assume we can decide the Halting Problem. Then there exists some total function *Halt* such that

$$Halt(x,y) = \begin{cases} 1 & \text{if } [x](y)\downarrow \\ 0 & \text{if } [x](y)\uparrow \end{cases}$$

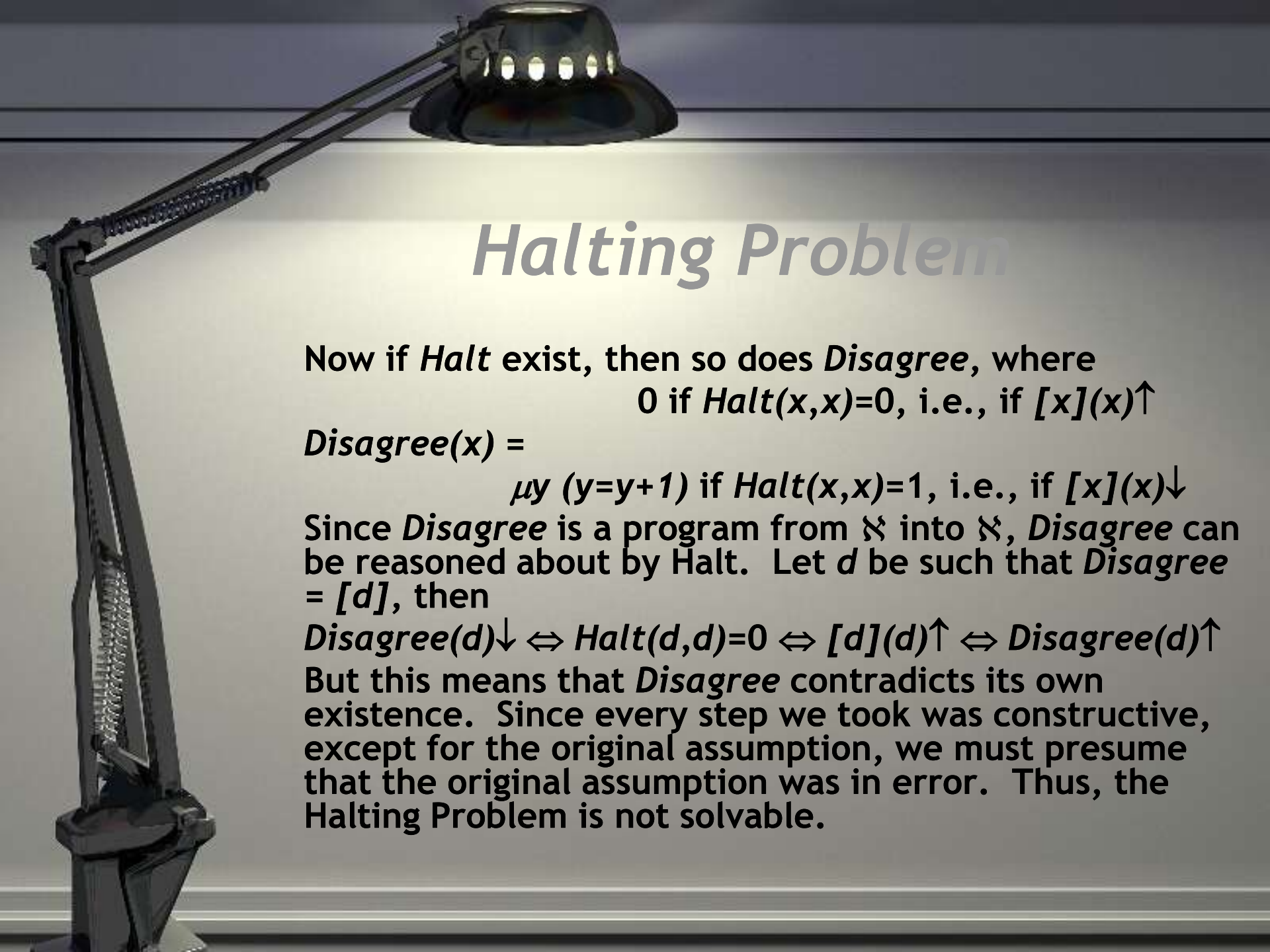
Here, we have numbered all programs and [*x*] refers to the *x*-th program in this ordering. Now we can view Halt as a mapping from $\aleph$ into {0,1} by treating its input as a single number representing the pairing of two numbers via the one-one onto function

$pair(x,y) = \langle x,y \rangle = 2^x\ (2y + 1) - 1$

with inverses

$\langle z \rangle_1 = exp(z+1,1)$

$\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$



Halting Problem

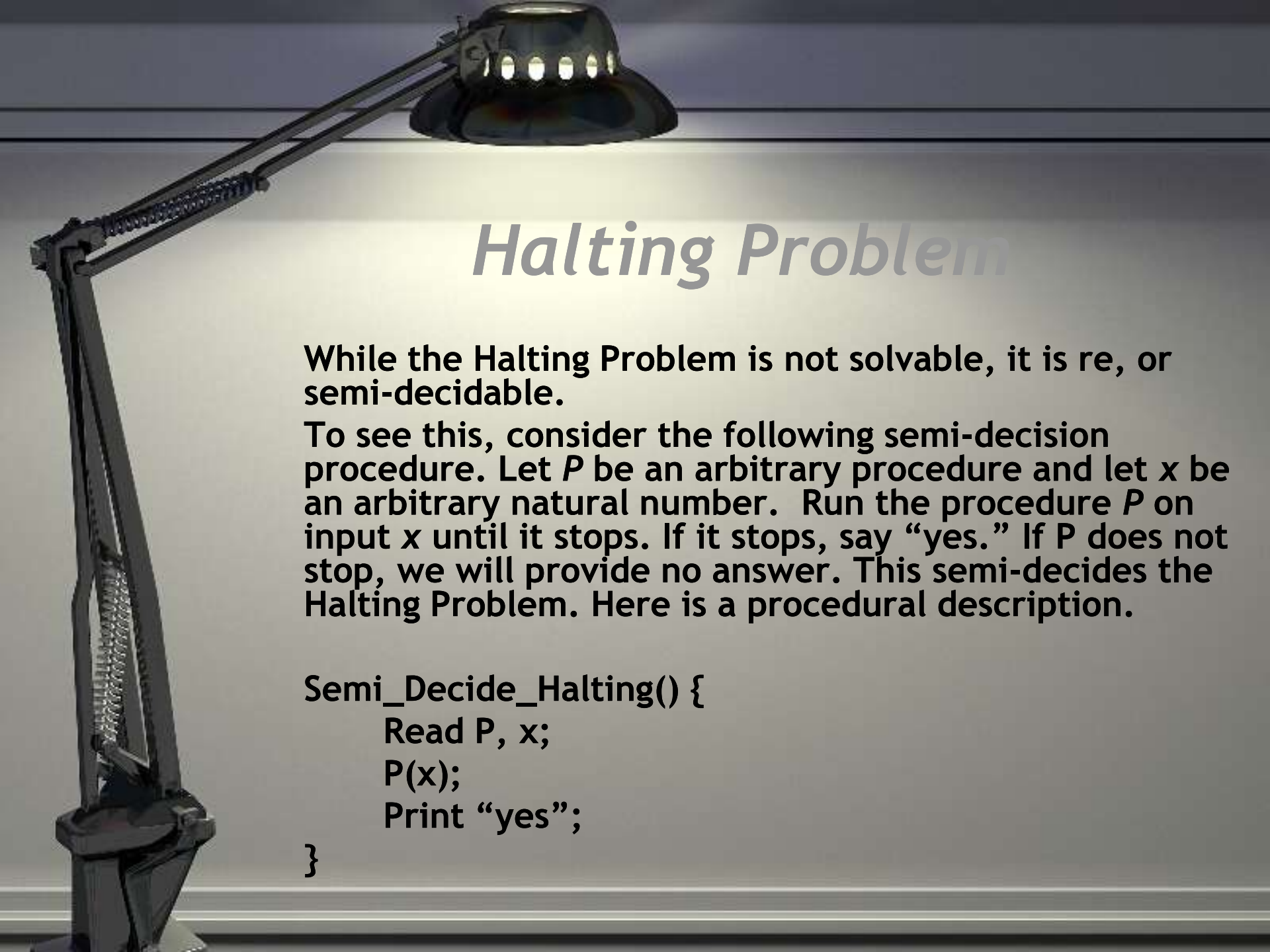
Now if *Halt* exist, then so does *Disagree*, where

$$Disagree(x) = \begin{cases} 0 \text{ if } Halt(x,x)=0, \text{ i.e., if } [x](x)\uparrow \\ \mu y\ (y=y+1) \text{ if } Halt(x,x)=1, \text{ i.e., if } [x](x)\downarrow \end{cases}$$

Since *Disagree* is a program from $\aleph$ into $\aleph$, *Disagree* can be reasoned about by Halt. Let *d* be such that *Disagree* = *[d]*, then

$$Disagree(d)\downarrow \Leftrightarrow Halt(d,d)=0 \Leftrightarrow [d](d)\uparrow \Leftrightarrow Disagree(d)\uparrow$$

But this means that *Disagree* contradicts its own existence. Since every step we took was constructive, except for the original assumption, we must presume that the original assumption was in error. Thus, the Halting Problem is not solvable.

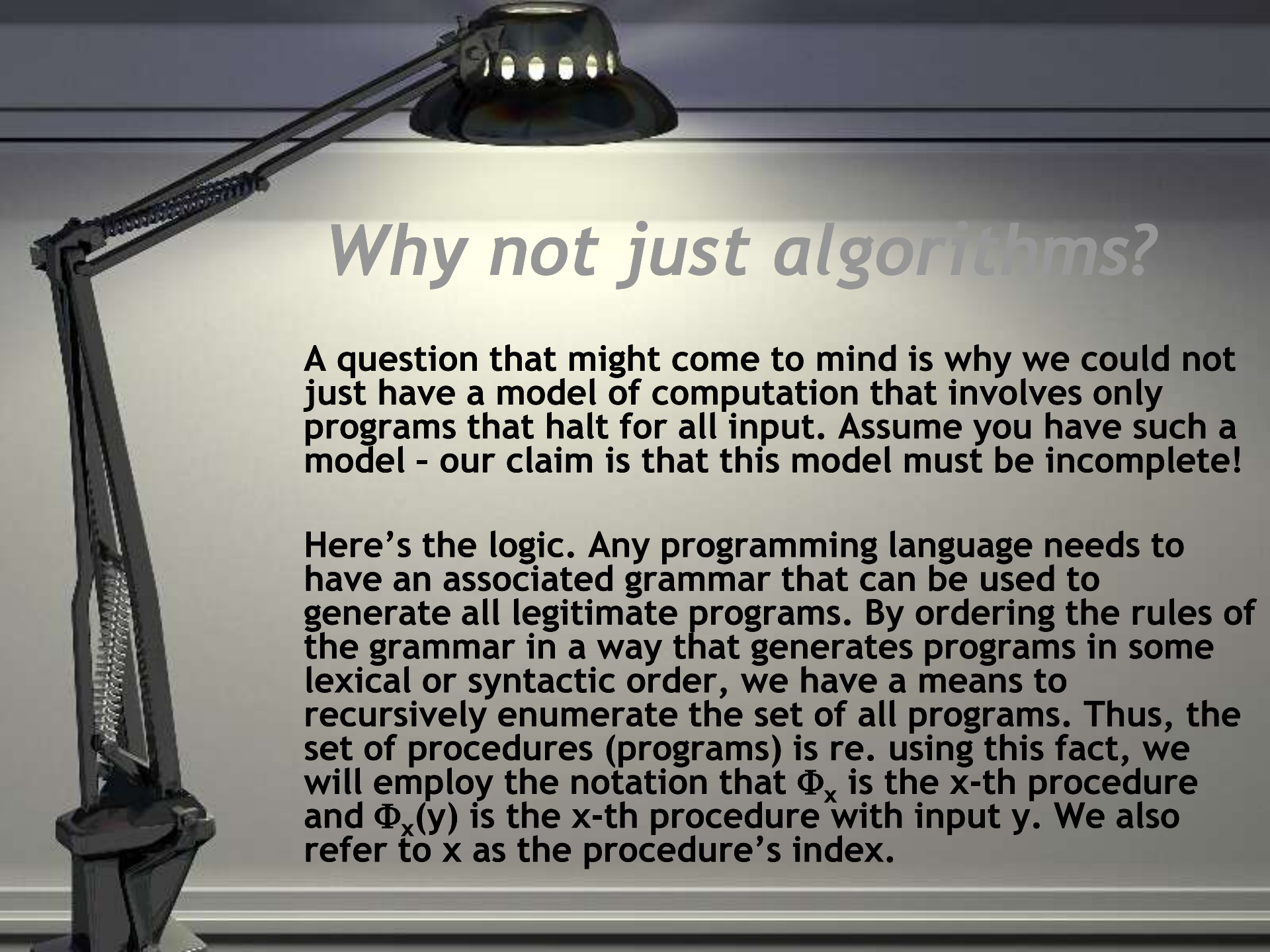


Halting Problem

While the Halting Problem is not solvable, it is re, or semi-decidable.

To see this, consider the following semi-decision procedure. Let *P* be an arbitrary procedure and let *x* be an arbitrary natural number. Run the procedure *P* on input *x* until it stops. If it stops, say "yes." If P does not stop, we will provide no answer. This semi-decides the Halting Problem. Here is a procedural description.

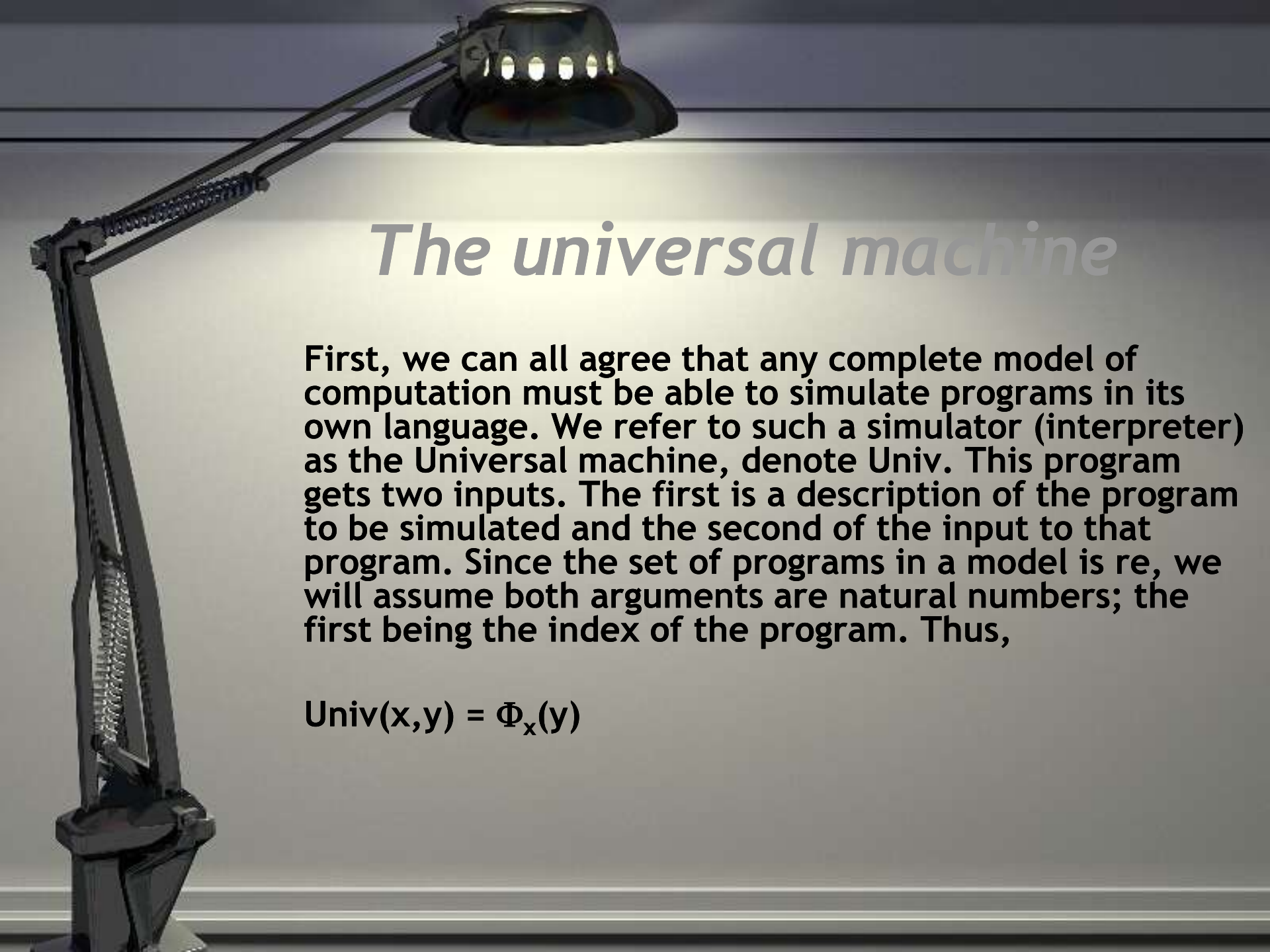
```
Semi_Decide_Halting() {
        Read P, x;
        P(x);
        Print "yes";
}
```



Why not just algorithms?

A question that might come to mind is why we could not just have a model of computation that involves only programs that halt for all input. Assume you have such a model – our claim is that this model must be incomplete!

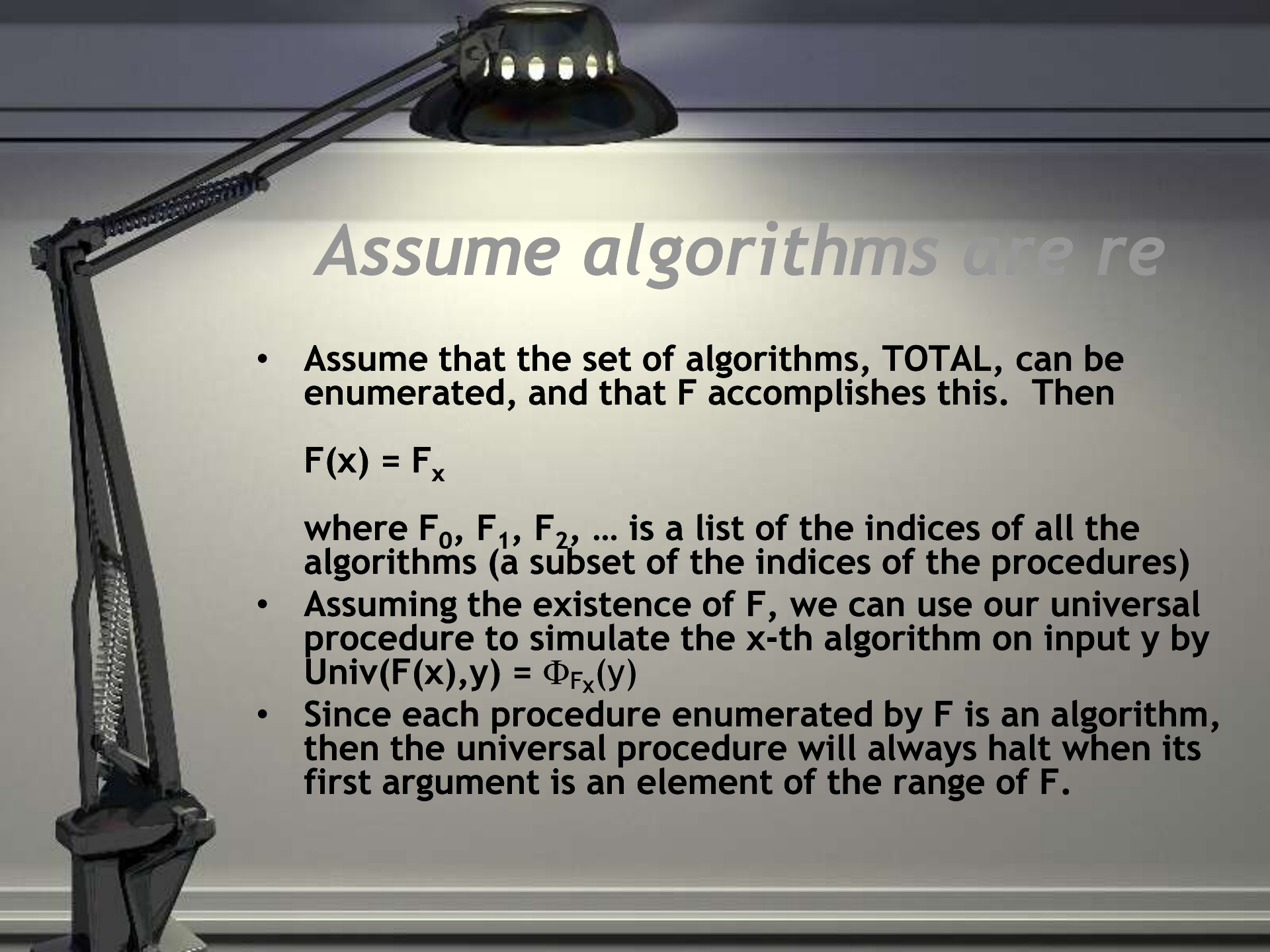
Here's the logic. Any programming language needs to have an associated grammar that can be used to generate all legitimate programs. By ordering the rules of the grammar in a way that generates programs in some lexical or syntactic order, we have a means to recursively enumerate the set of all programs. Thus, the set of procedures (programs) is re. using this fact, we will employ the notation that Φ_x is the x-th procedure and $\Phi_x(y)$ is the x-th procedure with input y. We also refer to x as the procedure's index.



The universal machine

First, we can all agree that any complete model of computation must be able to simulate programs in its own language. We refer to such a simulator (interpreter) as the Universal machine, denote Univ. This program gets two inputs. The first is a description of the program to be simulated and the second of the input to that program. Since the set of programs in a model is re, we will assume both arguments are natural numbers; the first being the index of the program. Thus,

$\text{Univ}(x,y) = \Phi_x(y)$

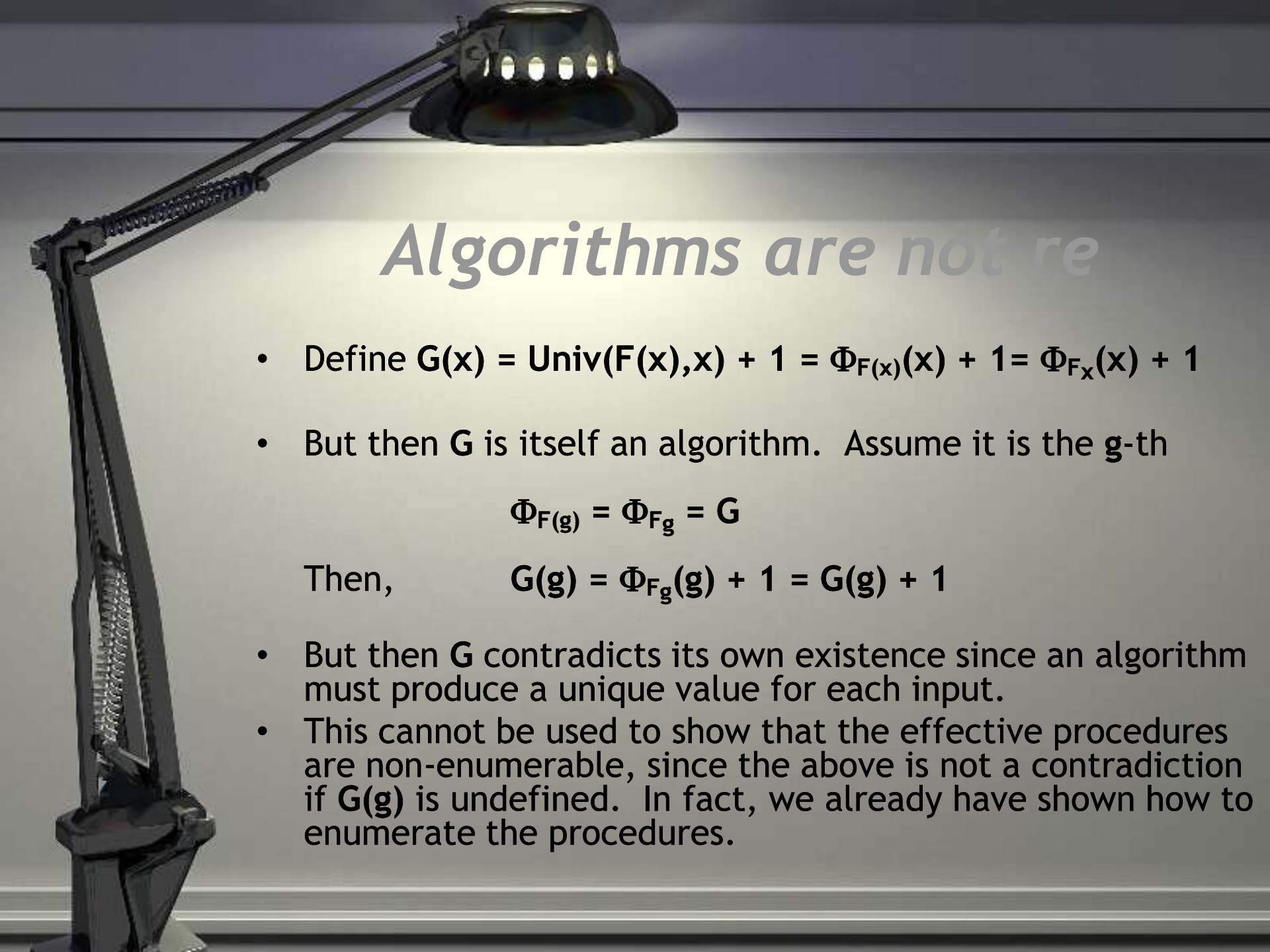


Assume algorithms are re

- Assume that the set of algorithms, TOTAL, can be enumerated, and that F accomplishes this. Then

  $F(x) = F_x$

  where F_0, F_1, F_2, … is a list of the indices of all the algorithms (a subset of the indices of the procedures)
- Assuming the existence of F, we can use our universal procedure to simulate the x-th algorithm on input y by $Univ(F(x),y) = \Phi_{F_x}(y)$
- Since each procedure enumerated by F is an algorithm, then the universal procedure will always halt when its first argument is an element of the range of F.



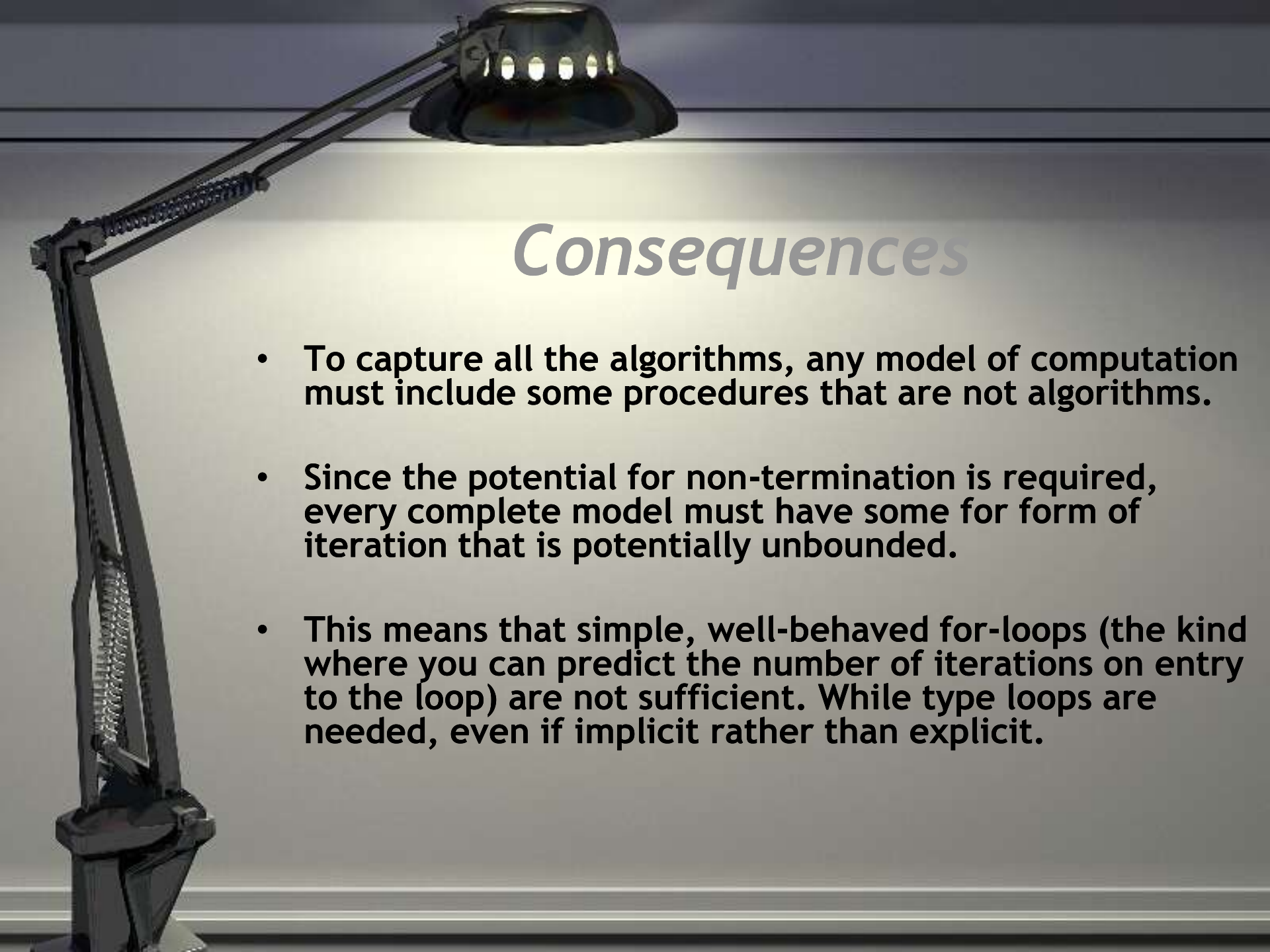
Algorithms are not re

- Define $G(x) = Univ(F(x),x) + 1 = \Phi_{F(x)}(x) + 1 = \Phi_{F_x}(x) + 1$
- But then **G** is itself an algorithm. Assume it is the **g**-th

$$\Phi_{F(g)} = \Phi_{F_g} = G$$

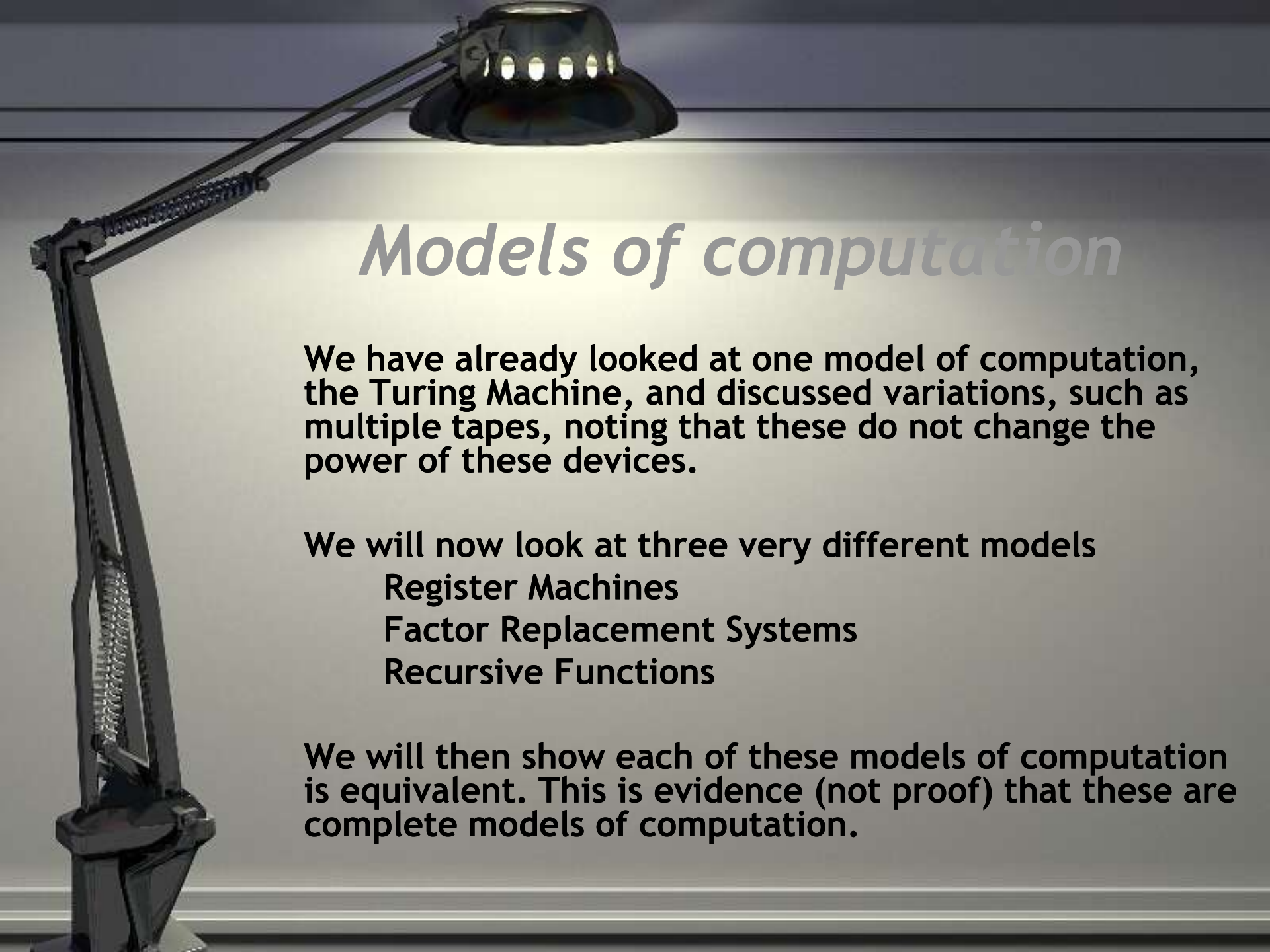
Then, $G(g) = \Phi_{F_g}(g) + 1 = G(g) + 1$

- But then **G** contradicts its own existence since an algorithm must produce a unique value for each input.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction if **G(g)** is undefined. In fact, we already have shown how to enumerate the procedures.



Consequences

- To capture all the algorithms, any model of computation must include some procedures that are not algorithms.
- Since the potential for non-termination is required, every complete model must have some for form of iteration that is potentially unbounded.
- This means that simple, well-behaved for-loops (the kind where you can predict the number of iterations on entry to the loop) are not sufficient. While type loops are needed, even if implicit rather than explicit.



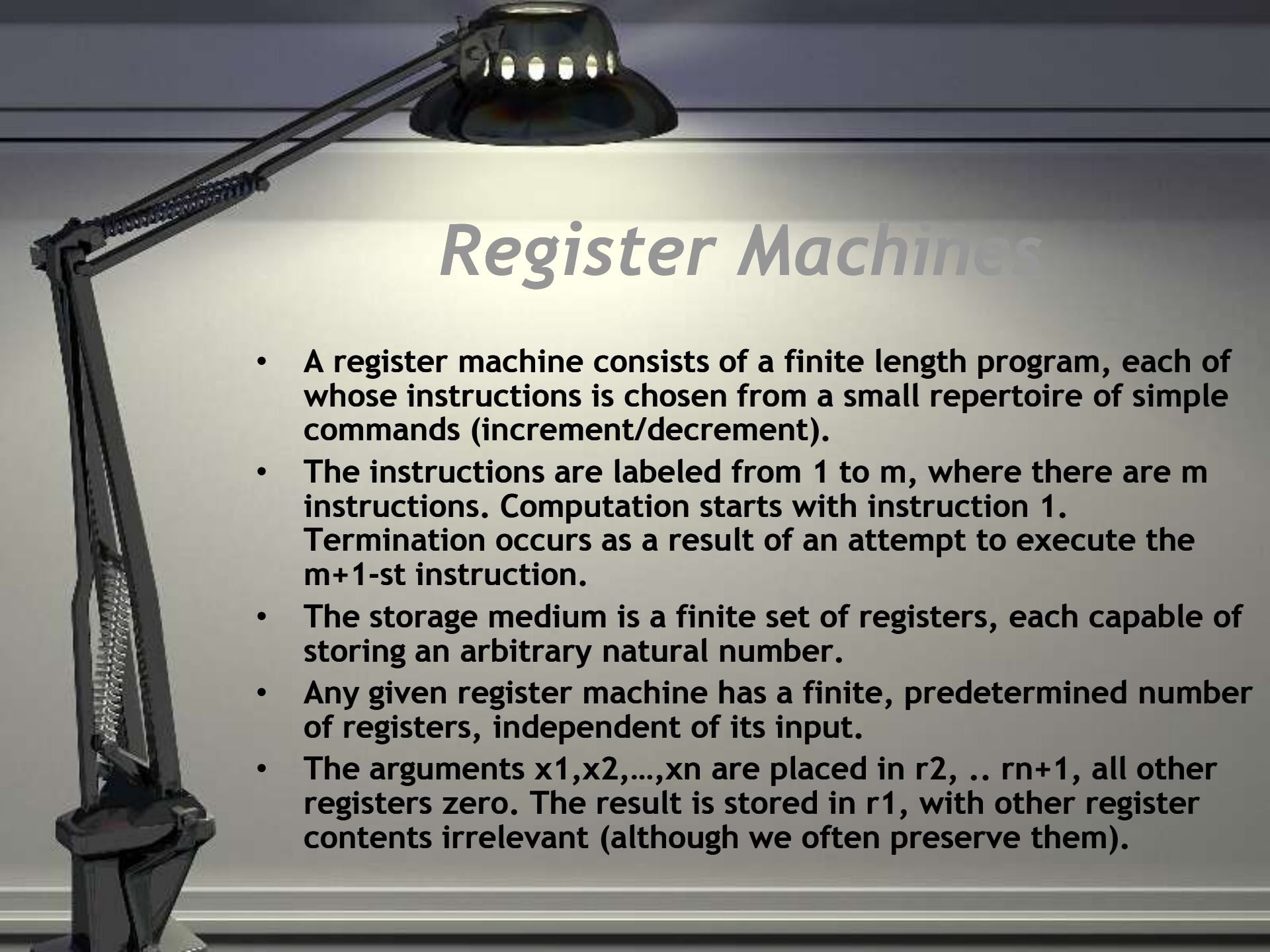
Models of computation

We have already looked at one model of computation, the Turing Machine, and discussed variations, such as multiple tapes, noting that these do not change the power of these devices.

We will now look at three very different models

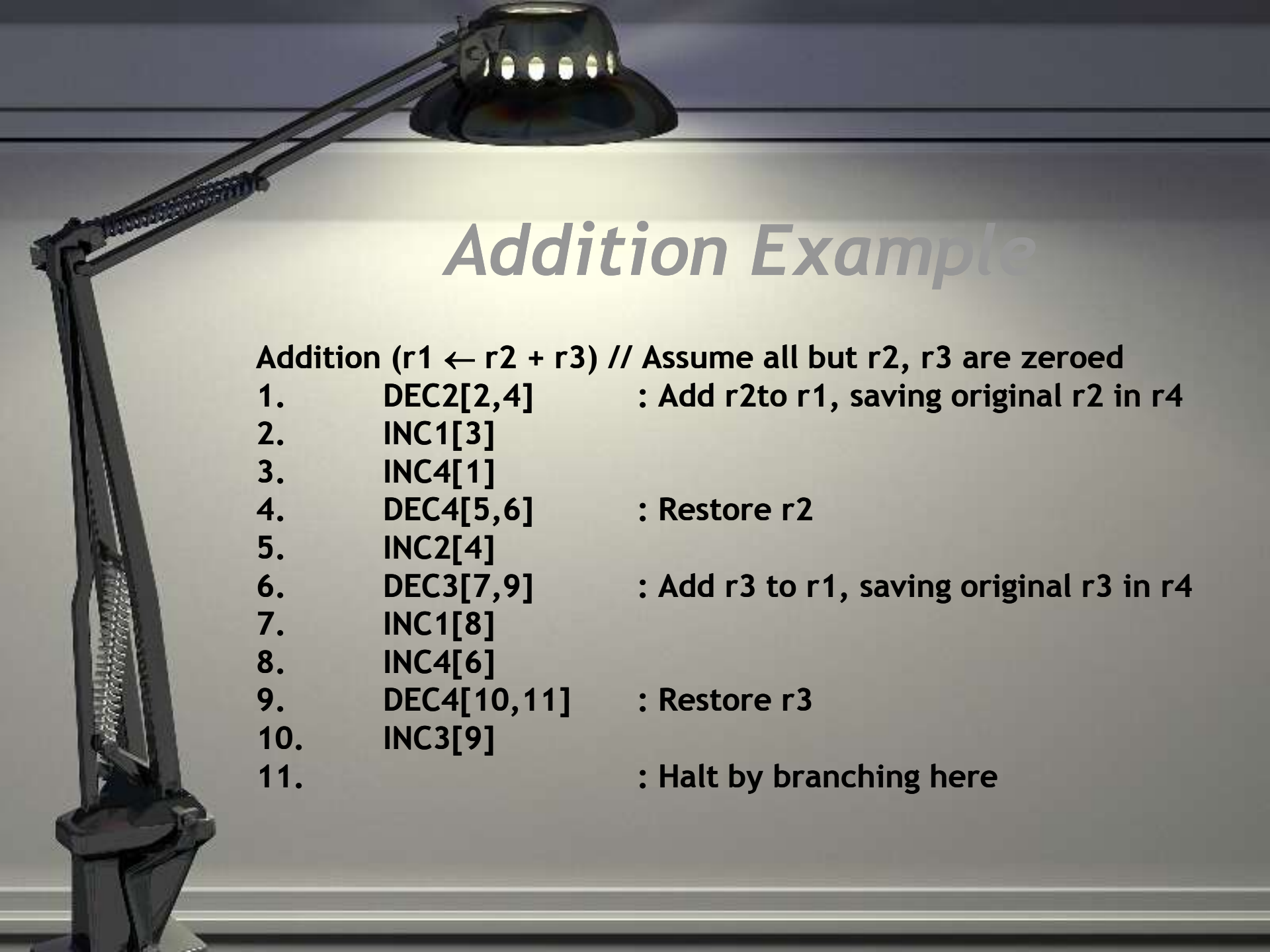
- Register Machines
- Factor Replacement Systems
- Recursive Functions

We will then show each of these models of computation is equivalent. This is evidence (not proof) that these are complete models of computation.



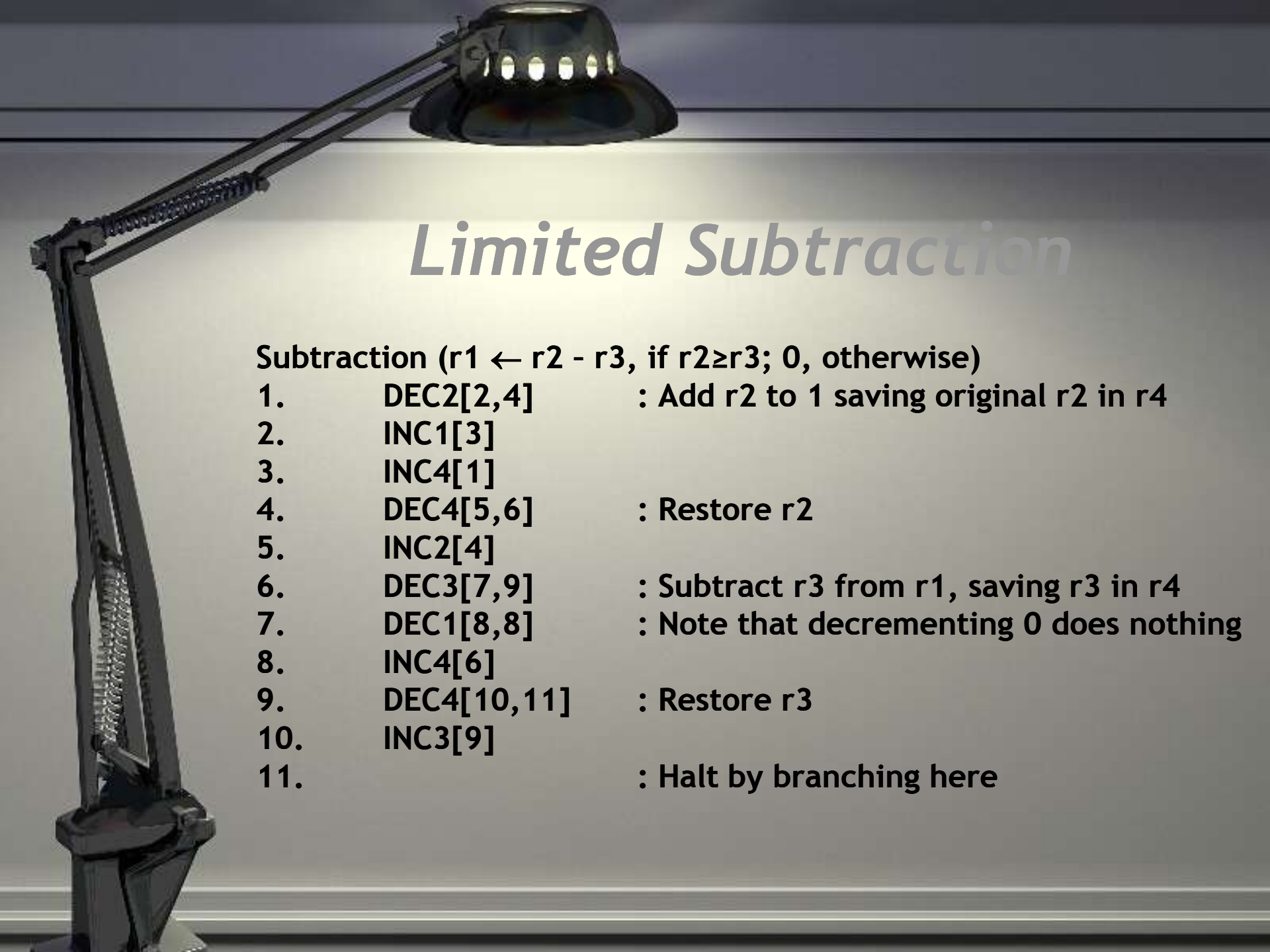
Register Machines

- A register machine consists of a finite length program, each of whose instructions is chosen from a small repertoire of simple commands (increment/decrement).
- The instructions are labeled from 1 to m, where there are m instructions. Computation starts with instruction 1. Termination occurs as a result of an attempt to execute the m+1-st instruction.
- The storage medium is a finite set of registers, each capable of storing an arbitrary natural number.
- Any given register machine has a finite, predetermined number of registers, independent of its input.
- The arguments x1,x2,…,xn are placed in r2, .. rn+1, all other registers zero. The result is stored in r1, with other register contents irrelevant (although we often preserve them).



Addition Example

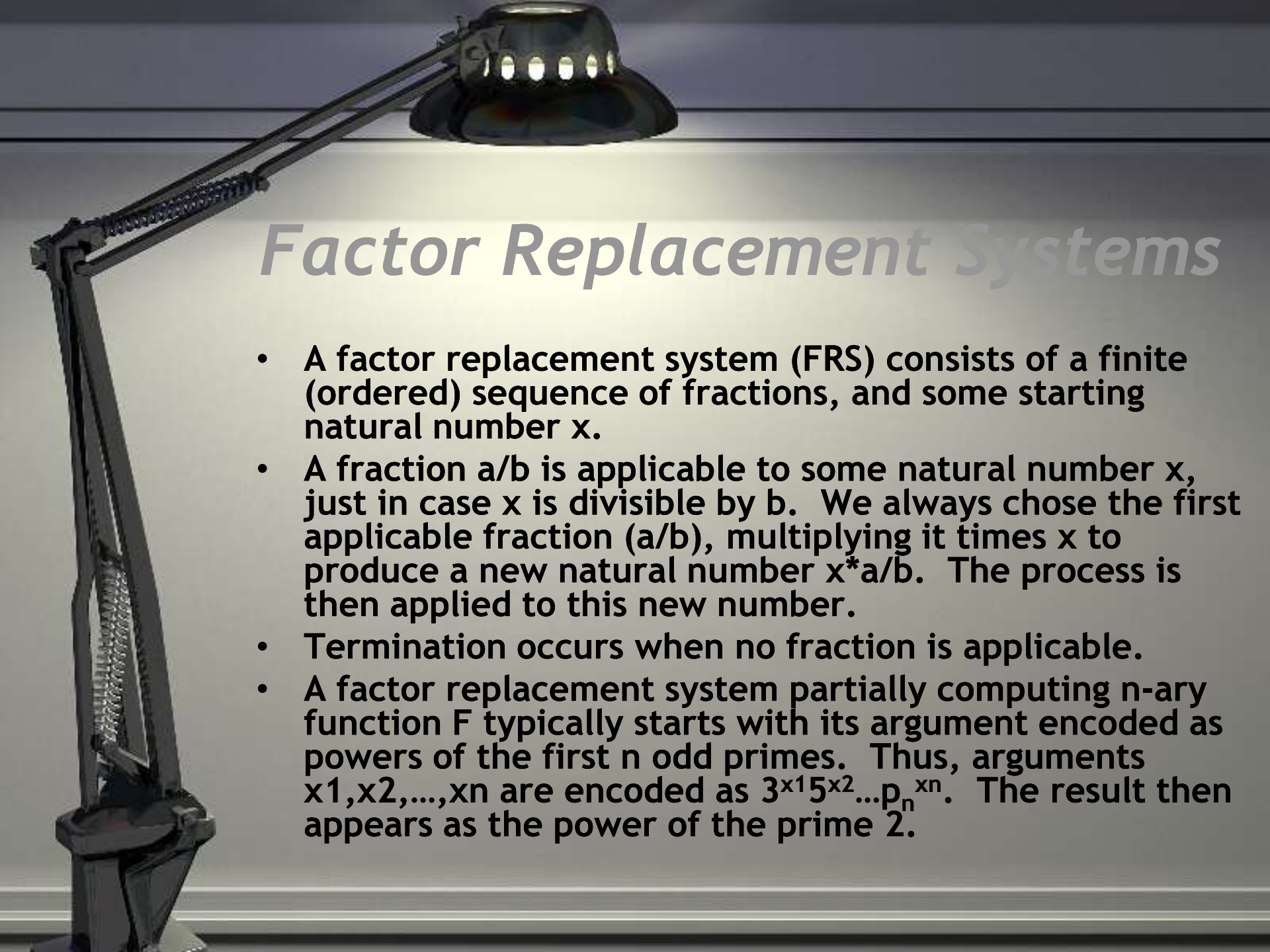
```
Addition (r1 ← r2 + r3) // Assume all but r2, r3 are zeroed
1.      DEC2[2,4]       : Add r2to r1, saving original r2 in r4
2.      INC1[3]
3.      INC4[1]
4.      DEC4[5,6]       : Restore r2
5.      INC2[4]
6.      DEC3[7,9]       : Add r3 to r1, saving original r3 in r4
7.      INC1[8]
8.      INC4[6]
9.      DEC4[10,11]     : Restore r3
10.     INC3[9]
11.                     : Halt by branching here
```



Limited Subtraction

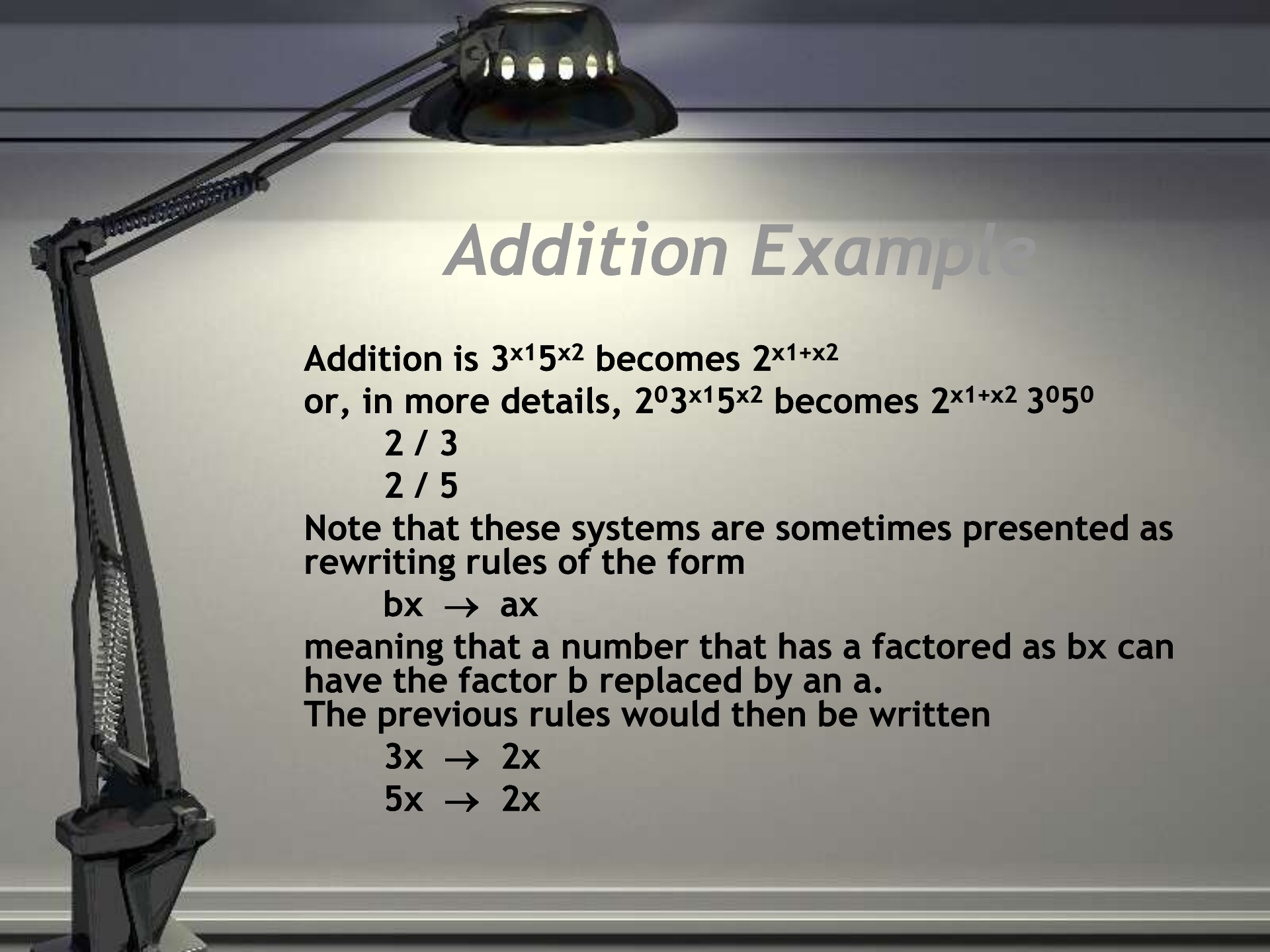
Subtraction ($r1 \leftarrow r2 - r3$, if $r2 \geq r3$; 0, otherwise)

```
1.    DEC2[2,4]     : Add r2 to 1 saving original r2 in r4
2.    INC1[3]
3.    INC4[1]
4.    DEC4[5,6]     : Restore r2
5.    INC2[4]
6.    DEC3[7,9]     : Subtract r3 from r1, saving r3 in r4
7.    DEC1[8,8]     : Note that decrementing 0 does nothing
8.    INC4[6]
9.    DEC4[10,11]   : Restore r3
10.   INC3[9]
11.                 : Halt by branching here
```



Factor Replacement Systems

- A factor replacement system (FRS) consists of a finite (ordered) sequence of fractions, and some starting natural number x.
- A fraction a/b is applicable to some natural number x, just in case x is divisible by b. We always chose the first applicable fraction (a/b), multiplying it times x to produce a new natural number x*a/b. The process is then applied to this new number.
- Termination occurs when no fraction is applicable.
- A factor replacement system partially computing n-ary function F typically starts with its argument encoded as powers of the first n odd primes. Thus, arguments x1,x2,…,xn are encoded as $3^{x1}5^{x2}\ldots p_n^{xn}$. The result then appears as the power of the prime 2.



Addition Example

Addition is $3^{x1}5^{x2}$ becomes 2^{x1+x2}

or, in more details, $2^{0}3^{x1}5^{x2}$ becomes $2^{x1+x2}\ 3^{0}5^{0}$

2 / 3

2 / 5

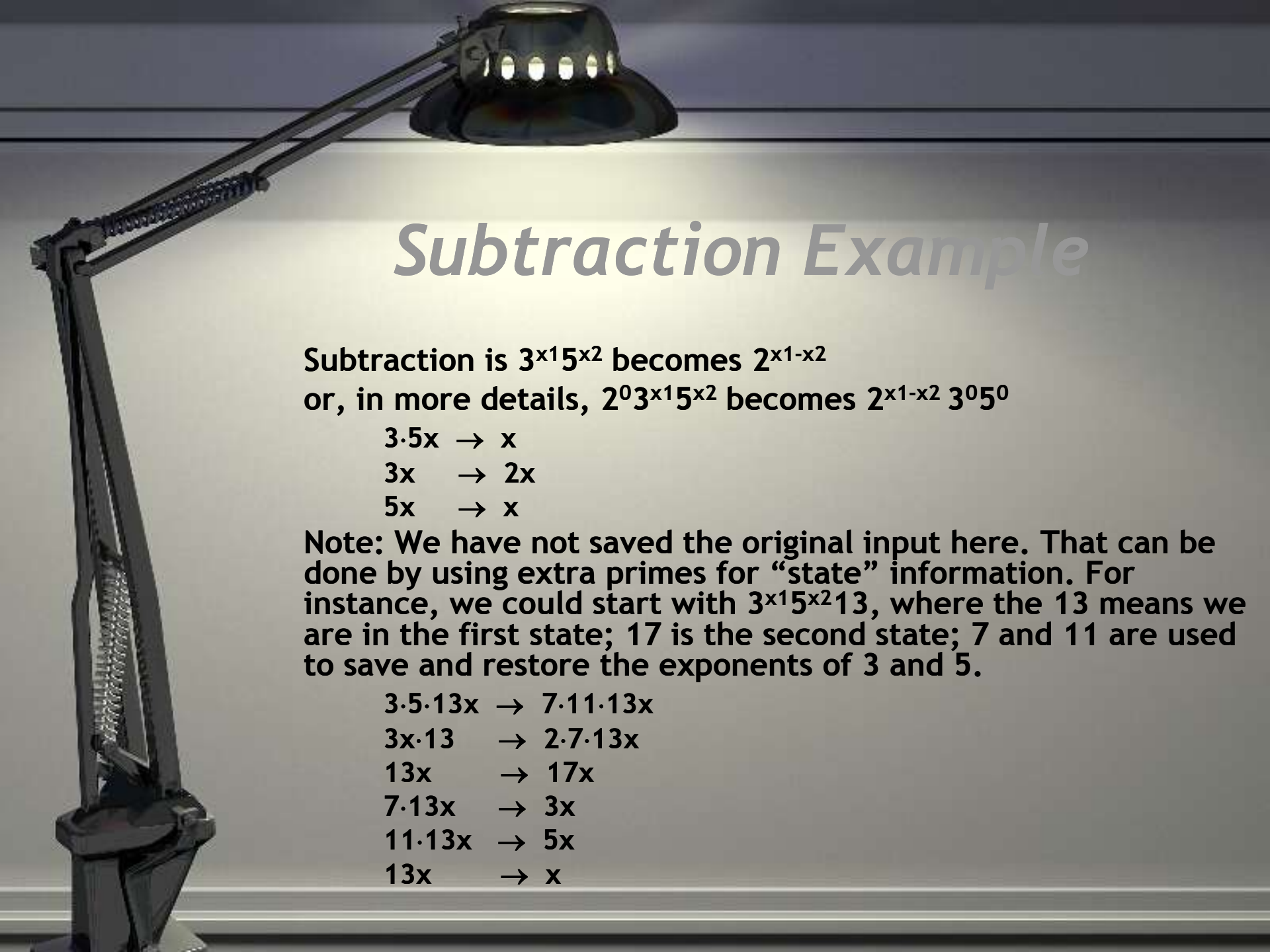
Note that these systems are sometimes presented as rewriting rules of the form

$bx \rightarrow ax$

meaning that a number that has a factored as bx can have the factor b replaced by an a.
The previous rules would then be written

$3x \rightarrow 2x$

$5x \rightarrow 2x$



Subtraction Example

Subtraction is $3^{x1}5^{x2}$ becomes 2^{x1-x2}

or, in more details, $2^0 3^{x1} 5^{x2}$ becomes $2^{x1-x2}\, 3^0 5^0$

$3 \cdot 5x \rightarrow x$

$3x \rightarrow 2x$

$5x \rightarrow x$

Note: We have not saved the original input here. That can be done by using extra primes for "state" information. For instance, we could start with $3^{x1}5^{x2}13$, where the 13 means we are in the first state; 17 is the second state; 7 and 11 are used to save and restore the exponents of 3 and 5.

$3 \cdot 5 \cdot 13x \rightarrow 7 \cdot 11 \cdot 13x$

$3x \cdot 13 \rightarrow 2 \cdot 7 \cdot 13x$

$13x \rightarrow 17x$

$7 \cdot 13x \rightarrow 3x$

$11 \cdot 13x \rightarrow 5x$

$13x \rightarrow x$

Importance of order

To see why determinism makes a difference, consider

$3 \cdot 5x \rightarrow x$

$3x \rightarrow 2x$

$5x \rightarrow x$

Starting with $135 = 3^3 5^1$, deterministically we get

$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$

Non-deterministically we get a larger, less selective set.

$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$

$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$

$135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1$

$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0$

$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0$

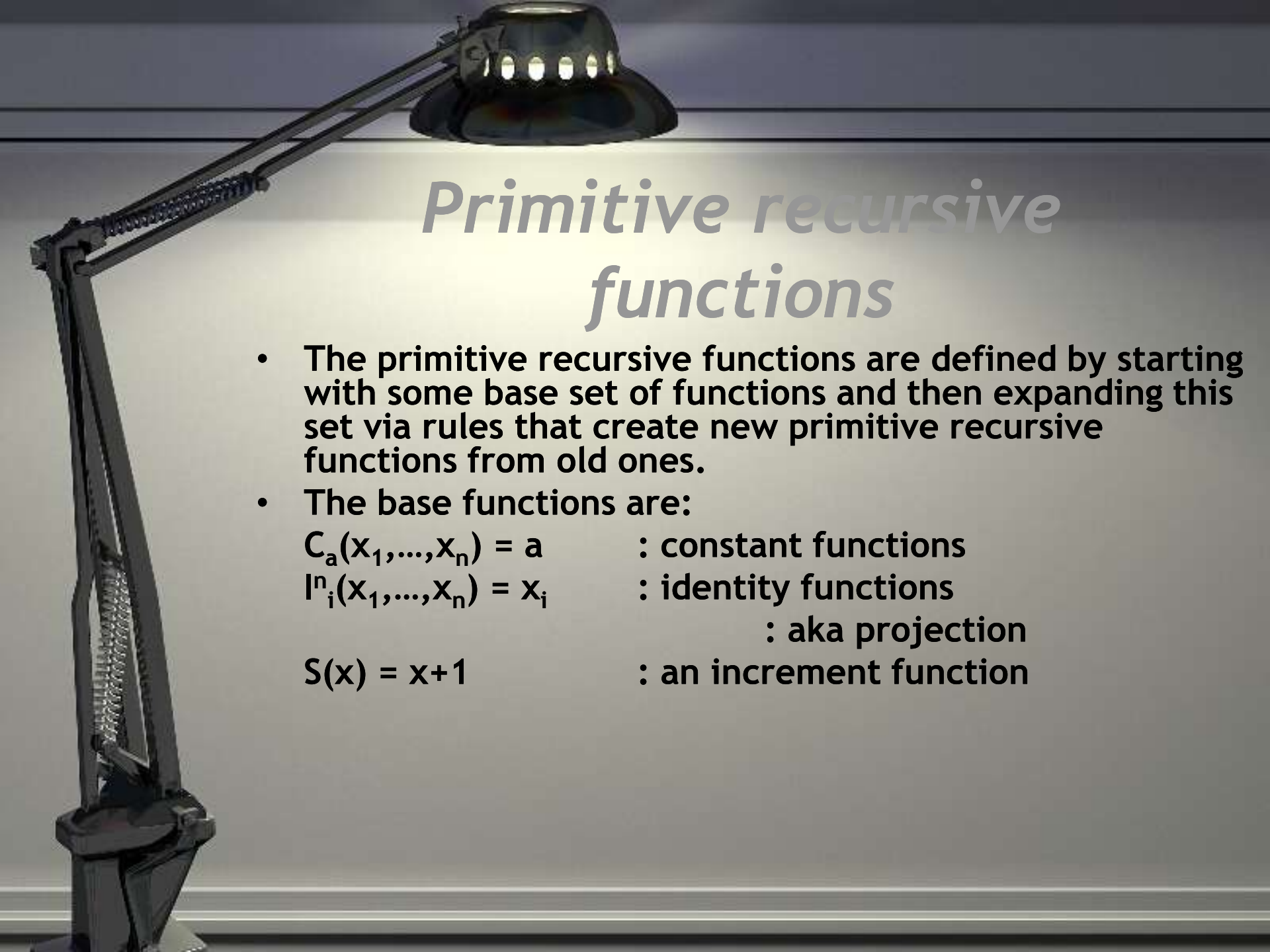
$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1$

$135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$

$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$

…

This computes 2^z where $0 \leq z \leq x_1$. Think about it.



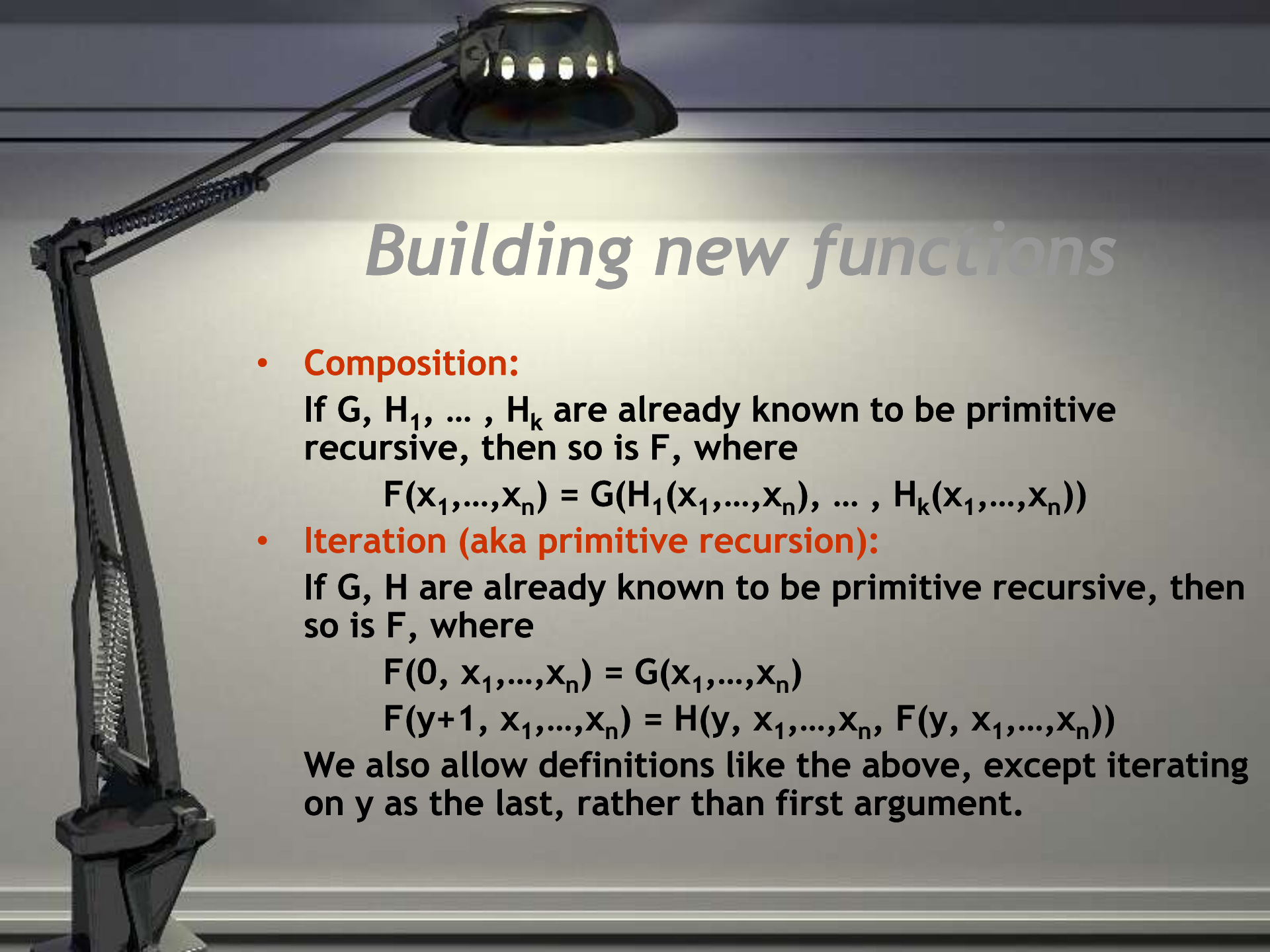
Primitive recursive functions

- The primitive recursive functions are defined by starting with some base set of functions and then expanding this set via rules that create new primitive recursive functions from old ones.
- The base functions are:

  $C_a(x_1,\ldots,x_n) = a$: constant functions

  $I^n_i(x_1,\ldots,x_n) = x_i$: identity functions
  : aka projection

  $S(x) = x+1$: an increment function



Building new functions

- **Composition:**

  If G, H_1, … , H_k are already known to be primitive recursive, then so is F, where

  $$F(x_1,\ldots,x_n) = G(H_1(x_1,\ldots,x_n), \ldots , H_k(x_1,\ldots,x_n))$$

- **Iteration (aka primitive recursion):**

  If G, H are already known to be primitive recursive, then so is F, where

  $$F(0, x_1,\ldots,x_n) = G(x_1,\ldots,x_n)$$

  $$F(y+1, x_1,\ldots,x_n) = H(y, x_1,\ldots,x_n, F(y, x_1,\ldots,x_n))$$

  We also allow definitions like the above, except iterating on y as the last, rather than first argument.

Addition and Multiplication

Example: Addition

$+(0,y) = I^1_1(y)$

$+(x+1,y) = H(x,y,+(x,y))$

where $H(a,b,c) = S(I^3_3(a,b,c))$

$= S(c) = +(x,y) + 1 = (x+y) + 1$

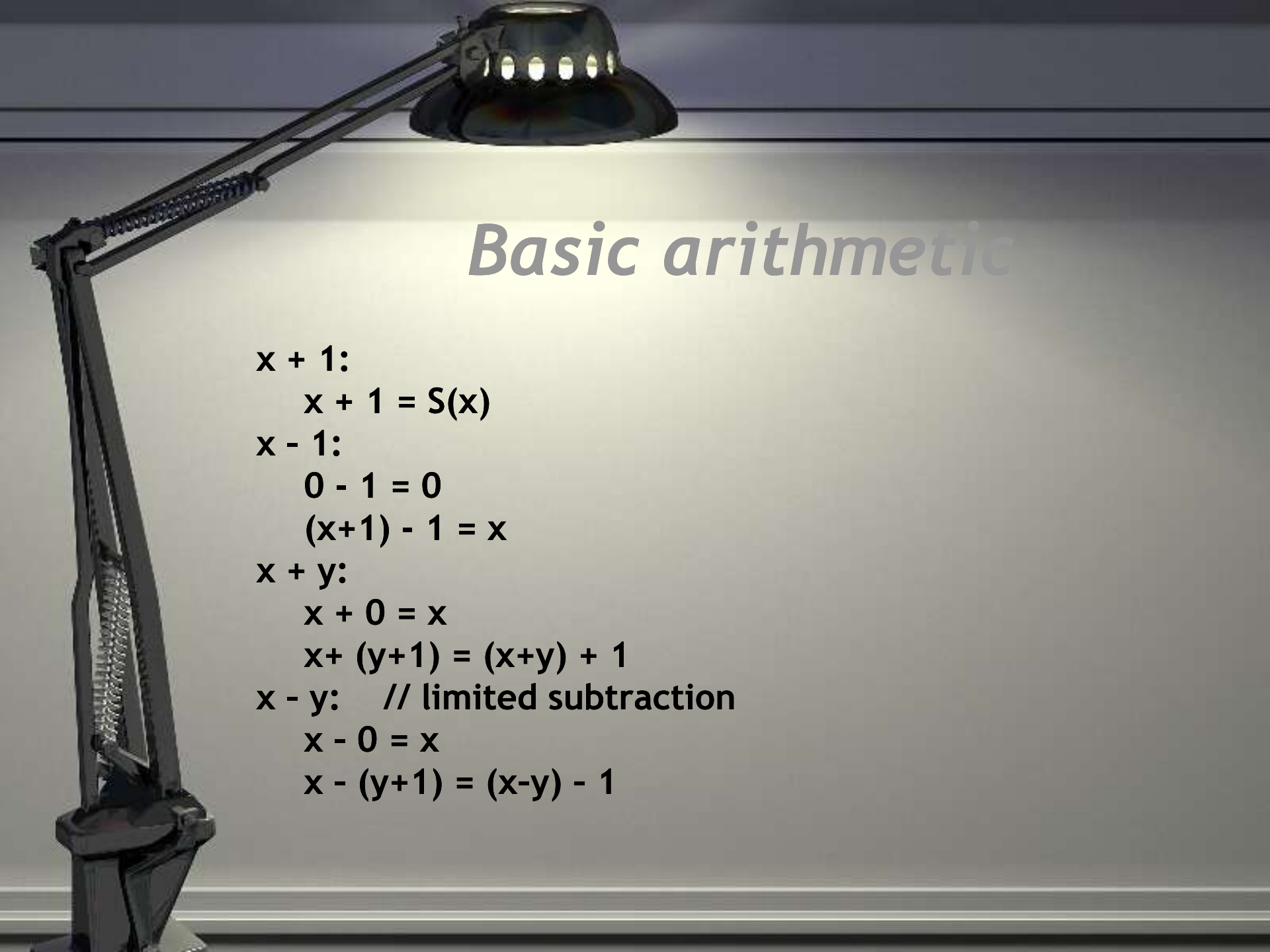
Example: Multiplication

$*(0,y) = C_0(y)$

$*(x+1,y) = H(x,y,*(x,y))$

where $H(a,b,c) = +(I^3_2(a,b,c), I^3_3(a,b,c))$

$= b+c = y + *(x,y) = (x+1)*y$



Basic arithmetic

x + 1:

    x + 1 = S(x)

x - 1:

    0 - 1 = 0

    (x+1) - 1 = x

x + y:

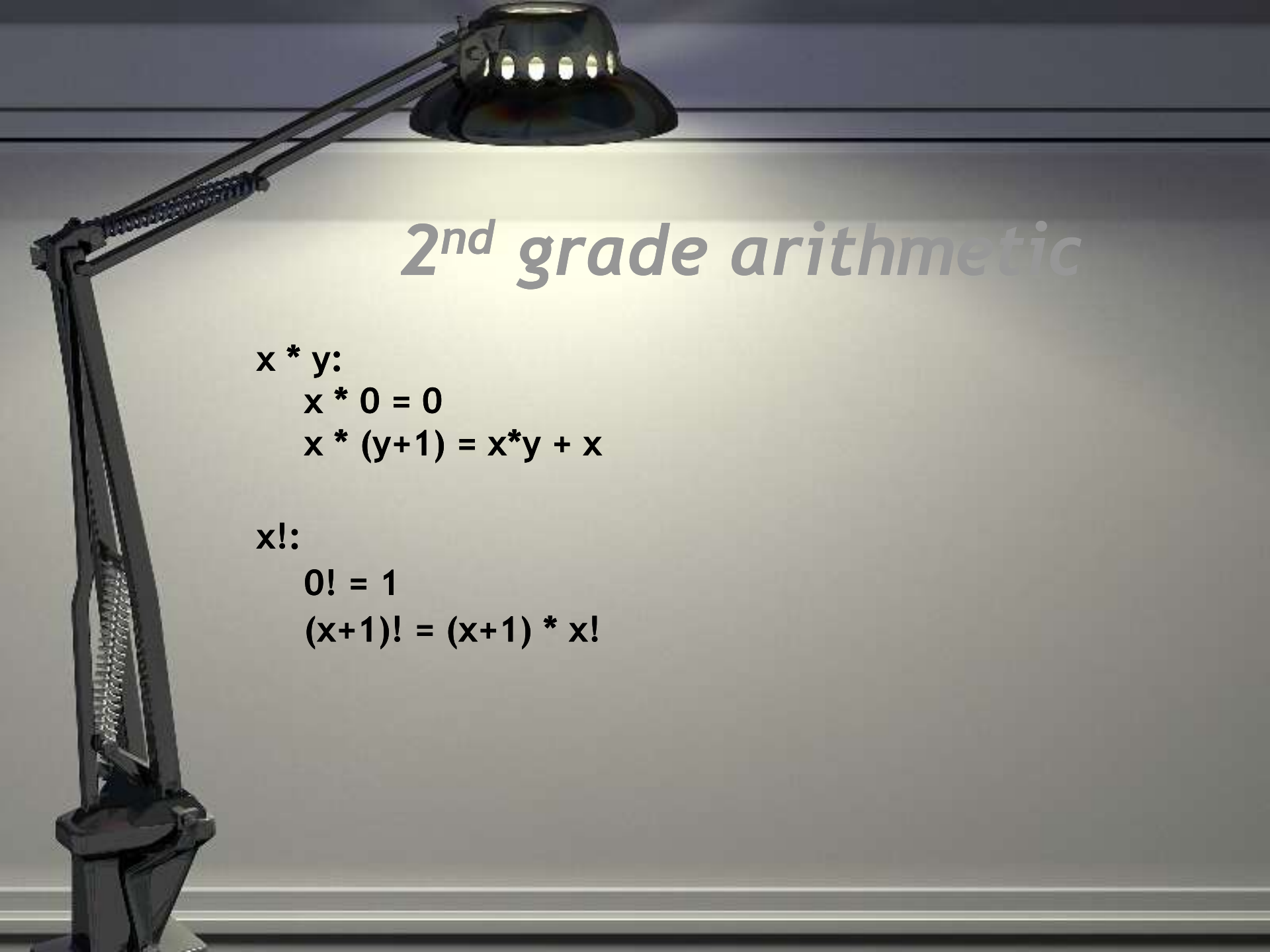
    x + 0 = x

    x+ (y+1) = (x+y) + 1

x - y: // limited subtraction

    x - 0 = x

    x - (y+1) = (x-y) - 1



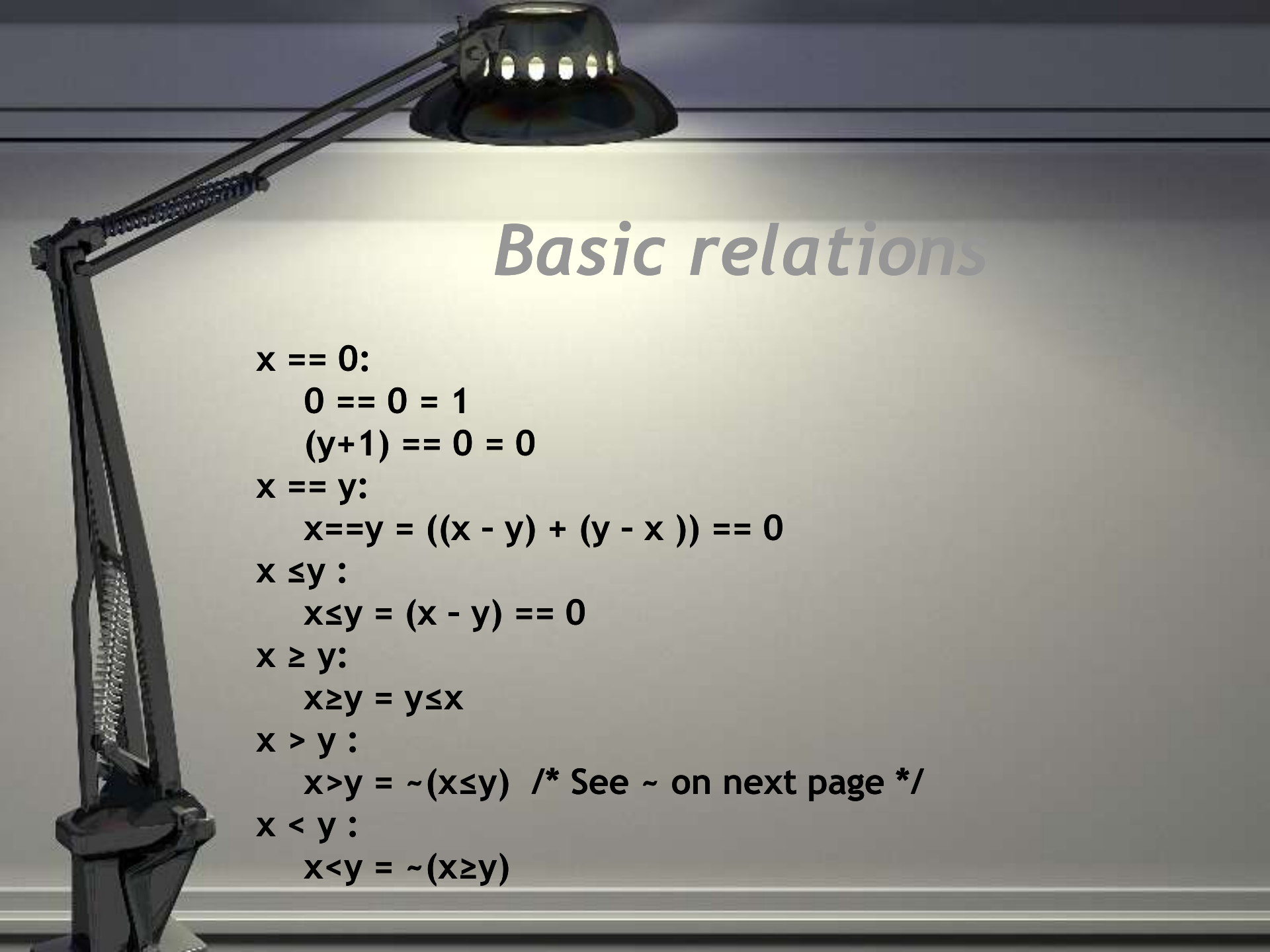
2^{nd} grade arithmetic

x * y:

- **x * 0 = 0**
- **x * (y+1) = x*y + x**

x!:

- **0! = 1**
- **(x+1)! = (x+1) * x!**



Basic relations

x == 0:

    0 == 0 = 1

    (y+1) == 0 = 0

x == y:

    x==y = ((x - y) + (y - x)) == 0

x ≤y :

    x≤y = (x - y) == 0

x ≥ y:

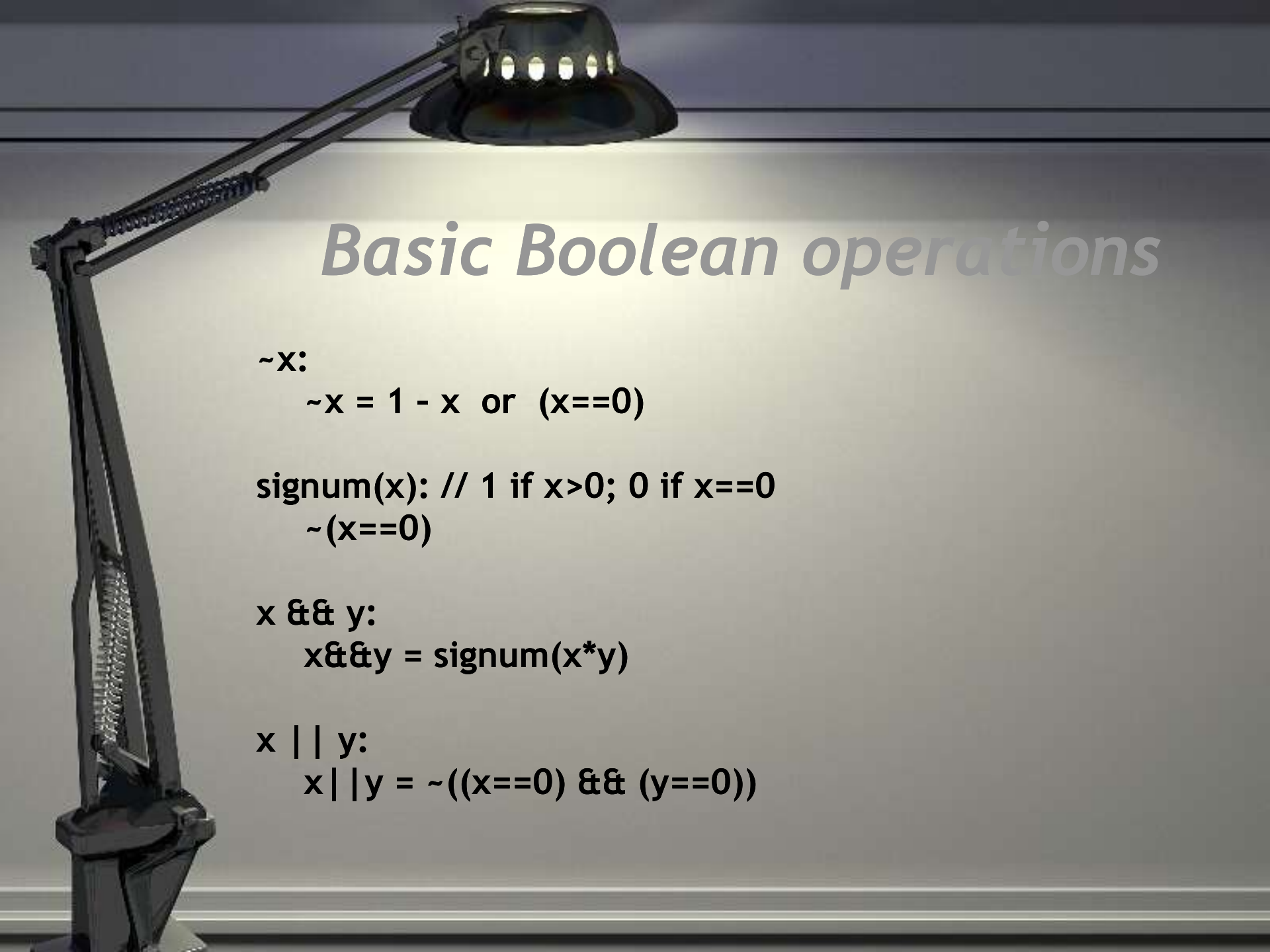
    x≥y = y≤x

x > y :

    x>y = ~(x≤y) /* See ~ on next page */

x < y :

    x<y = ~(x≥y)



Basic Boolean operations

~x:

~x = 1 - x or (x==0)

signum(x): // 1 if x>0; 0 if x==0

~(x==0)

x && y:

x&&y = signum(x*y)

x || y:

x||y = ~((x==0) && (y==0))

Definition by cases

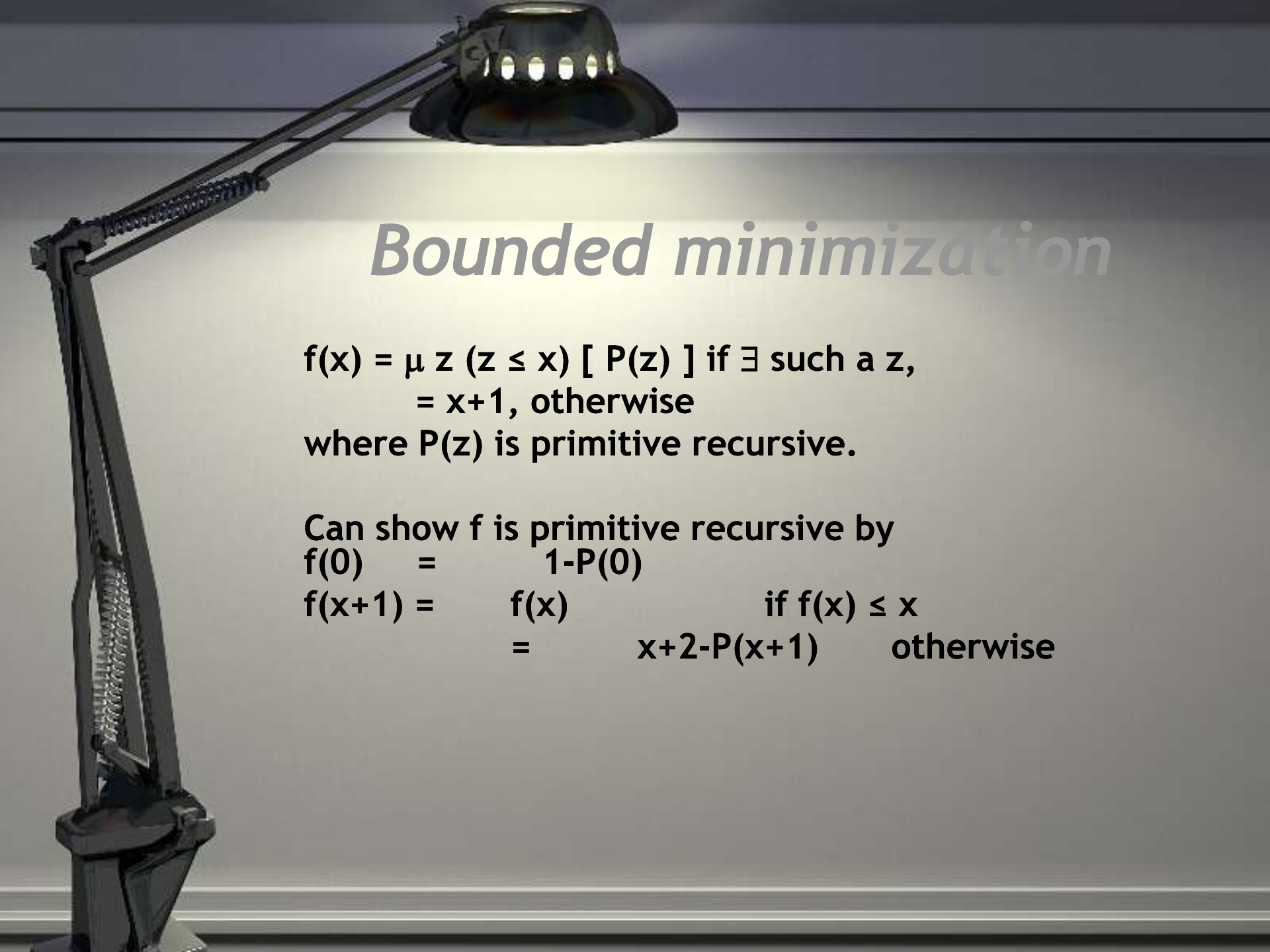
One case

$$f(x) = \begin{cases} g(x) & \text{if } P(x) \\ h(x) & \text{otherwise} \end{cases}$$

$$f(x) = P(x) * g(x) + (1-P(x)) * h(x)$$

Can use induction to prove this is true for all k>0, where

$$f(x) = \begin{cases} g_1(x) & \text{if } P_1(x) \\ g_2(x) & \text{if } P_2(x) \;\&\&\; \sim P_1(x) \\ \ldots & \\ g_k(x) & \text{if } P_k(x) \;\&\&\; \sim(P_1(x) \;||\; \ldots \;||\; \sim P_{k-1}(x)) \\ h(x) & \text{otherwise} \end{cases}$$



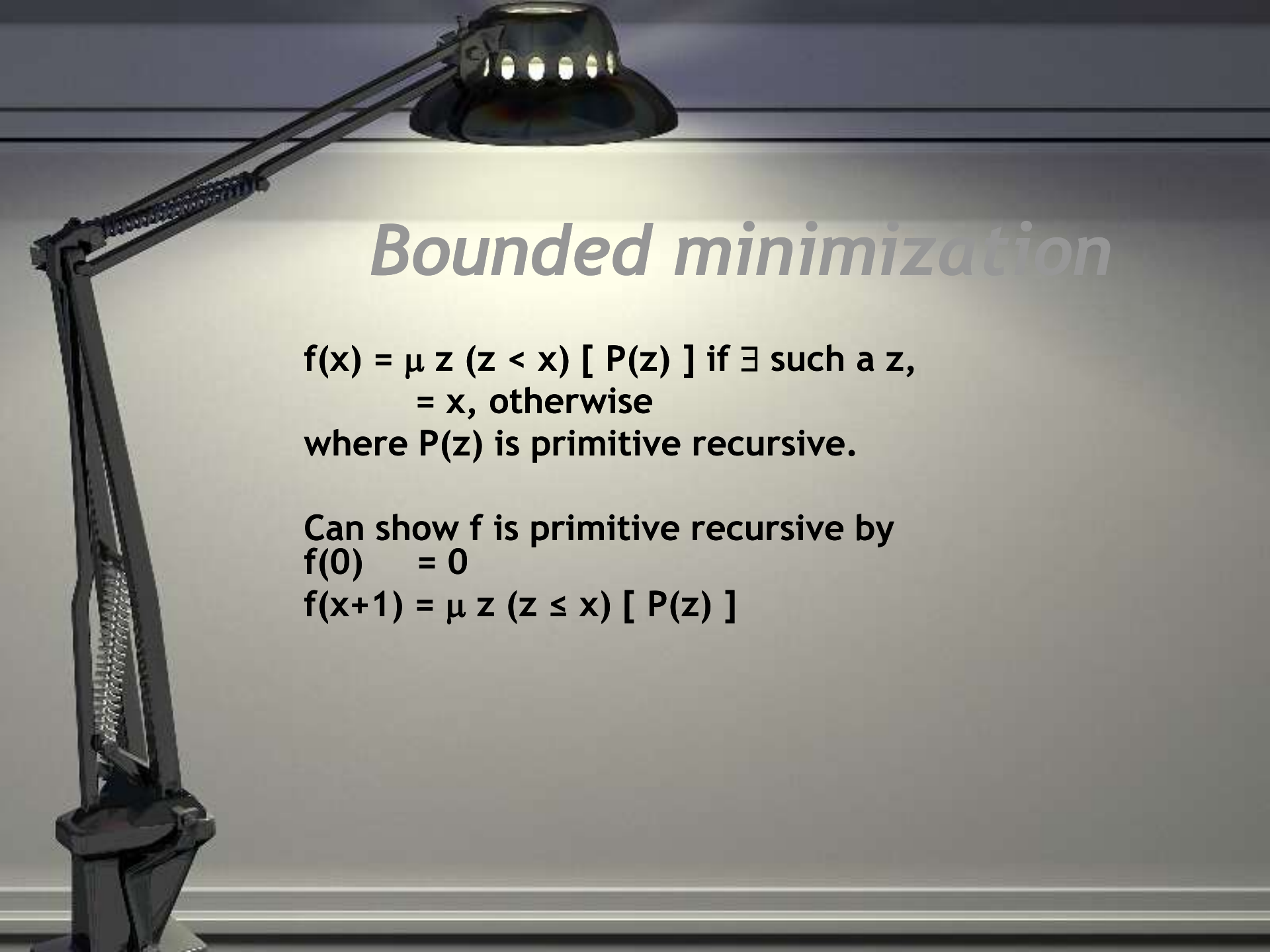
Bounded minimization

$f(x) = \mu z\ (z \le x)\ [\ P(z)\ ]$ if $\exists$ such a z,
$= x+1$, otherwise
where P(z) is primitive recursive.

Can show f is primitive recursive by

$f(0) = 1-P(0)$

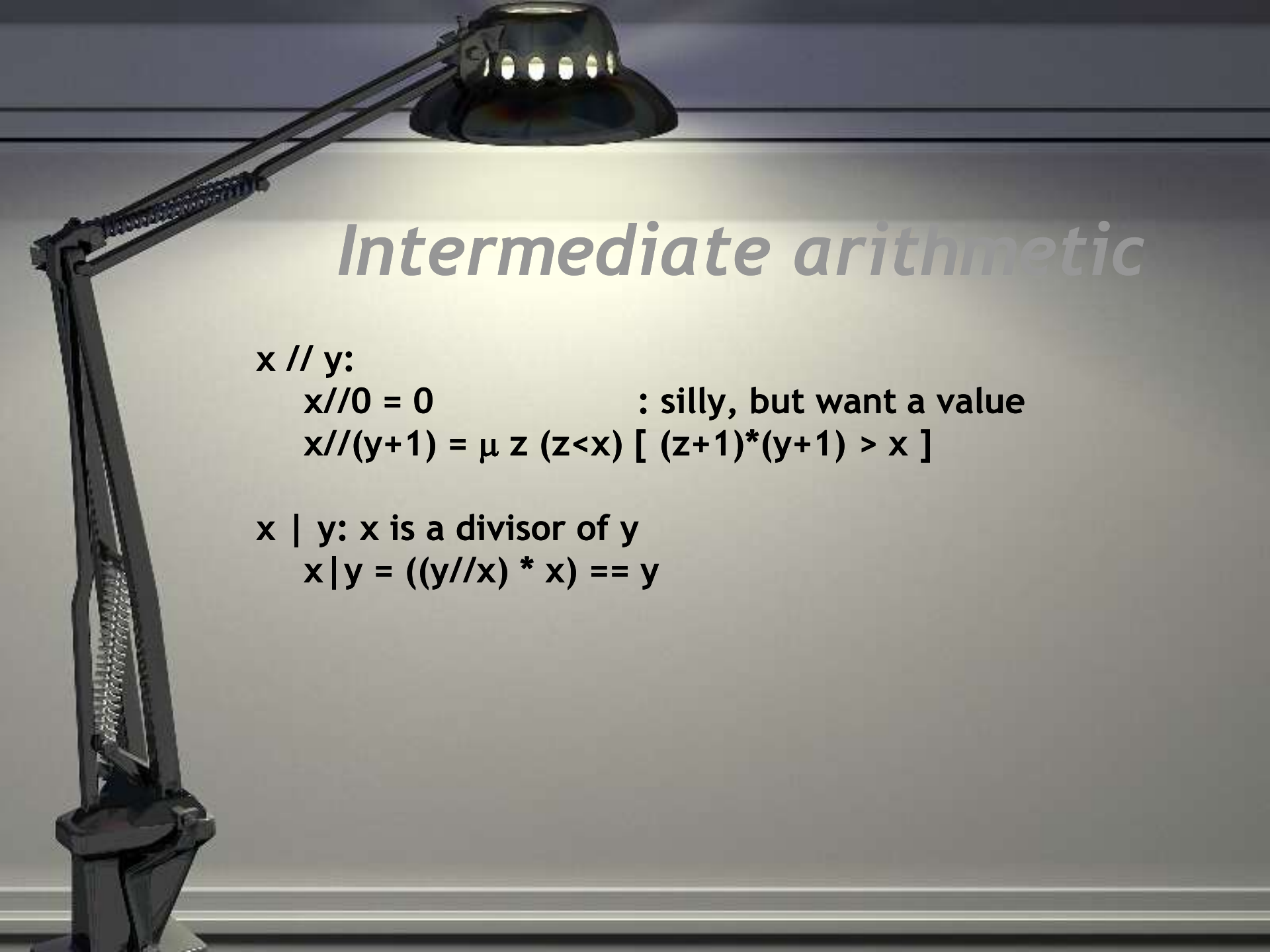
$f(x+1) = f(x)$ if $f(x) \le x$
$= x+2-P(x+1)$ otherwise



Bounded minimization

$f(x) = \mu z\ (z < x)\ [\ P(z)\ ]$ if $\exists$ such a z,
$= x$, otherwise
where P(z) is primitive recursive.

Can show f is primitive recursive by
$f(0) = 0$
$f(x+1) = \mu z\ (z \leq x)\ [\ P(z)\ ]$



Intermediate arithmetic

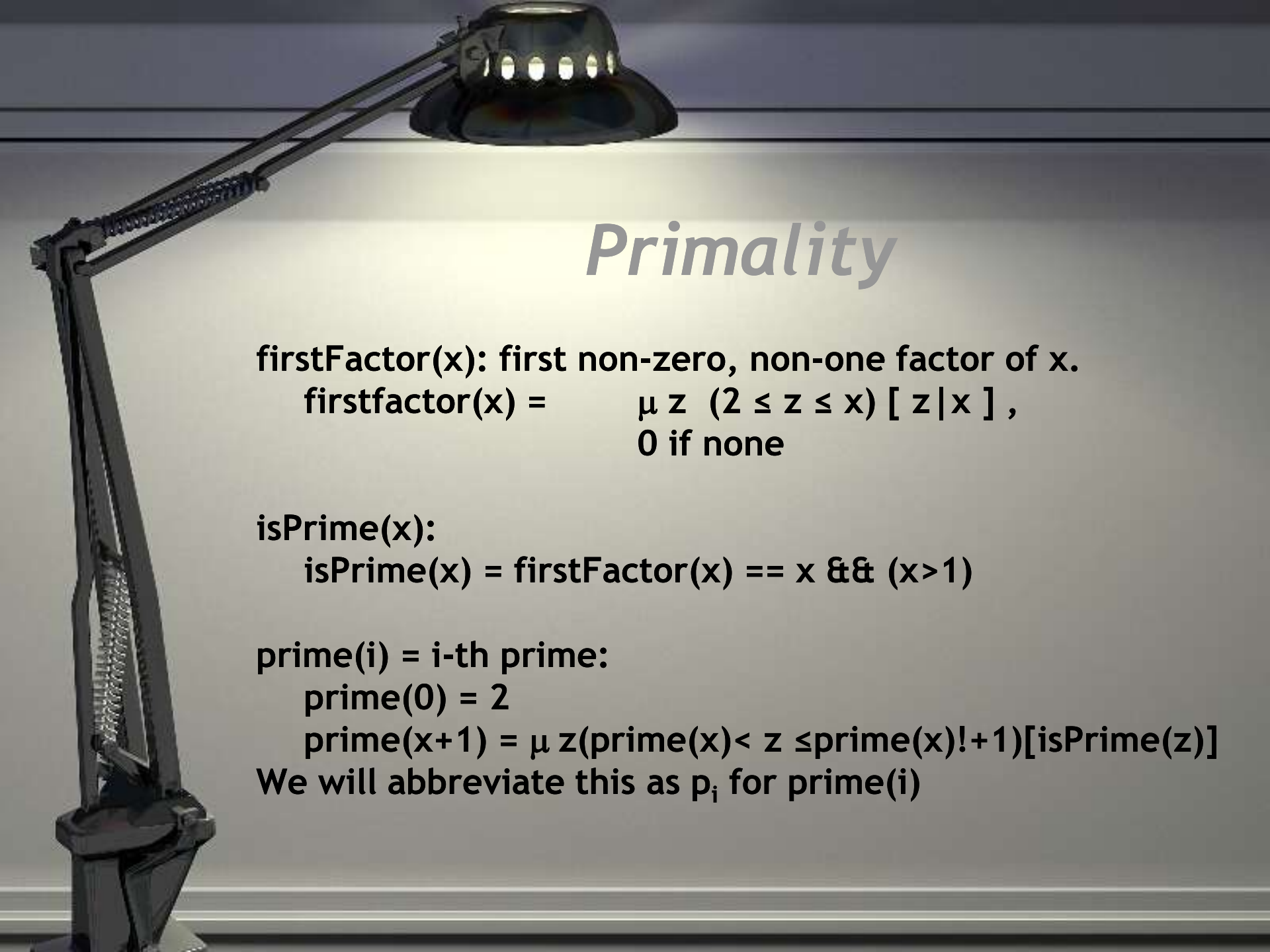
x // y:

x//0 = 0 : silly, but want a value

x//(y+1) = μ z (z<x) [(z+1)*(y+1) > x]

x | y: x is a divisor of y

x|y = ((y//x) * x) == y



Primality

firstFactor(x): first non-zero, non-one factor of x.

firstfactor(x) = $\mu z \ (2 \le z \le x) [z|x]$,
0 if none

isPrime(x):

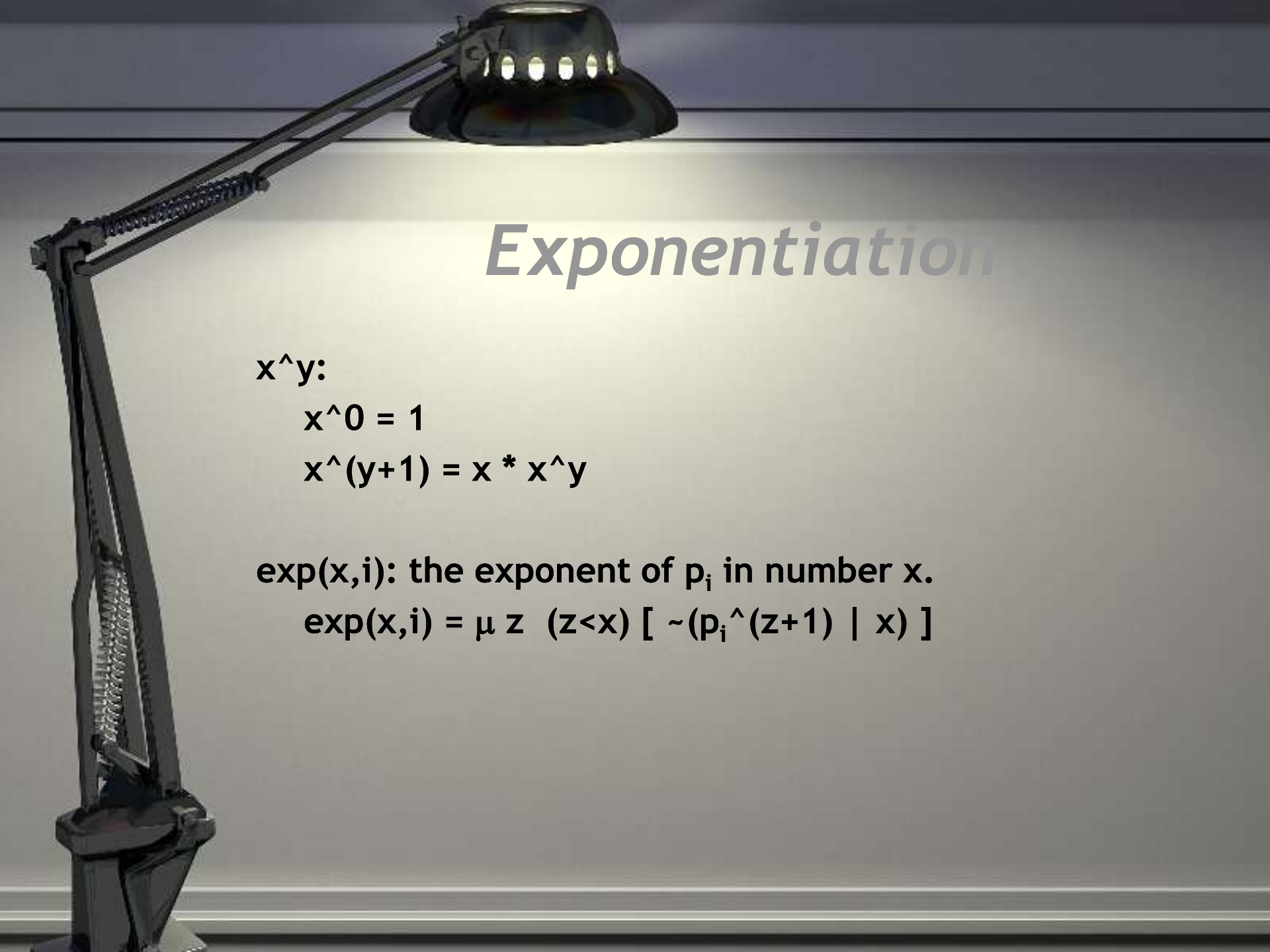
isPrime(x) = firstFactor(x) == x && (x>1)

prime(i) = i-th prime:

prime(0) = 2

prime(x+1) = $\mu z(\text{prime}(x) < z \le \text{prime}(x)!+1)[\text{isPrime}(z)]$

We will abbreviate this as p_i for prime(i)



Exponentiation

x^y:

x^0 = 1

x^(y+1) = x * x^y

exp(x,i): the exponent of p_i in number x.

exp(x,i) = μ z (z<x) [~(p_i^(z+1) | x)]

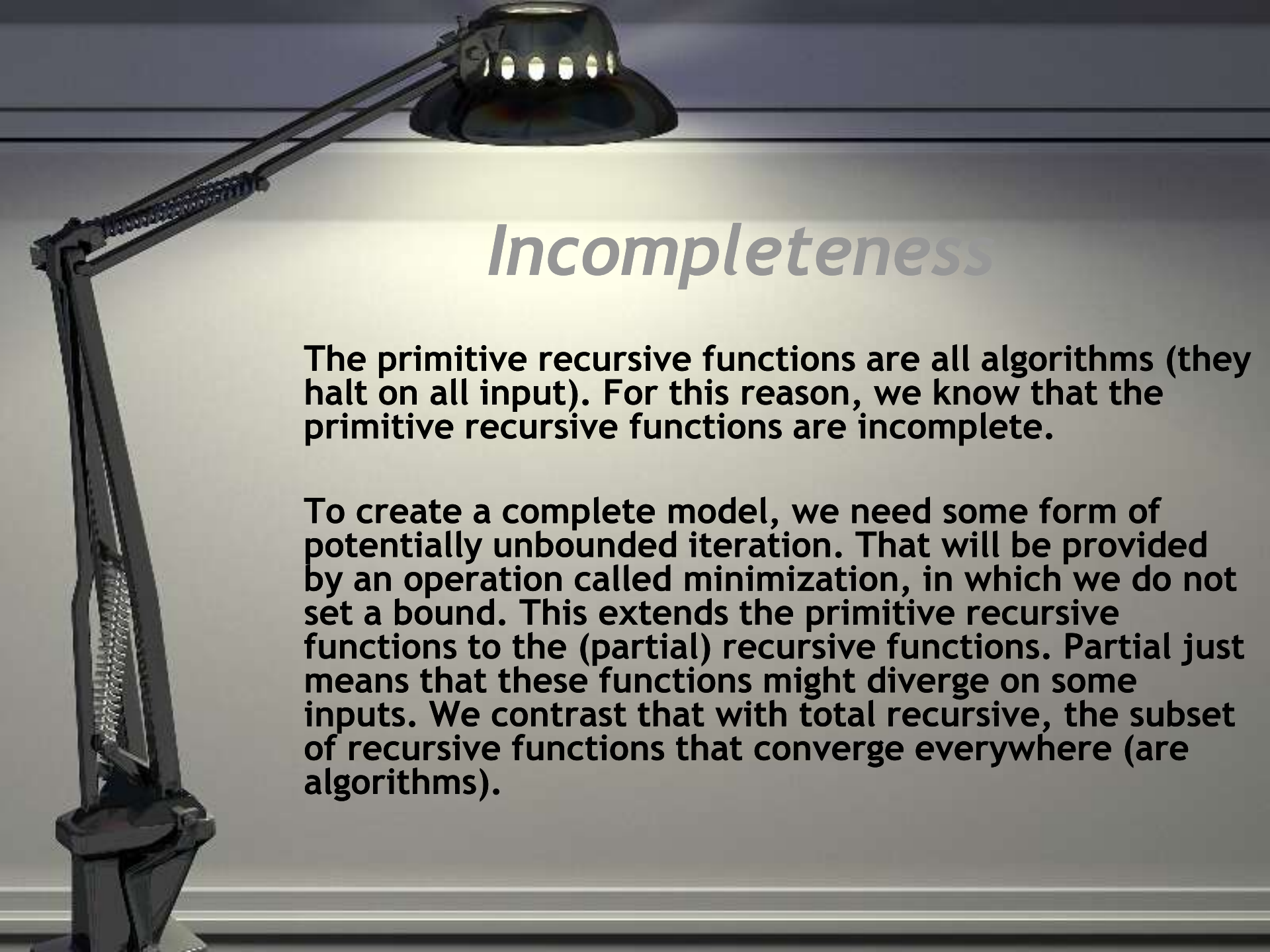
Pairing function

- pair(x,y) = $\langle x,y \rangle = 2^x (2y + 1) - 1$
- with inverses

  $\langle z \rangle_1 = \exp(z+1,0)$

  $\langle z \rangle_2 = (((z + 1) // 2^{\langle z \rangle_1}) - 1) // 2$
- These are very useful and can be extended to encode n-tuples

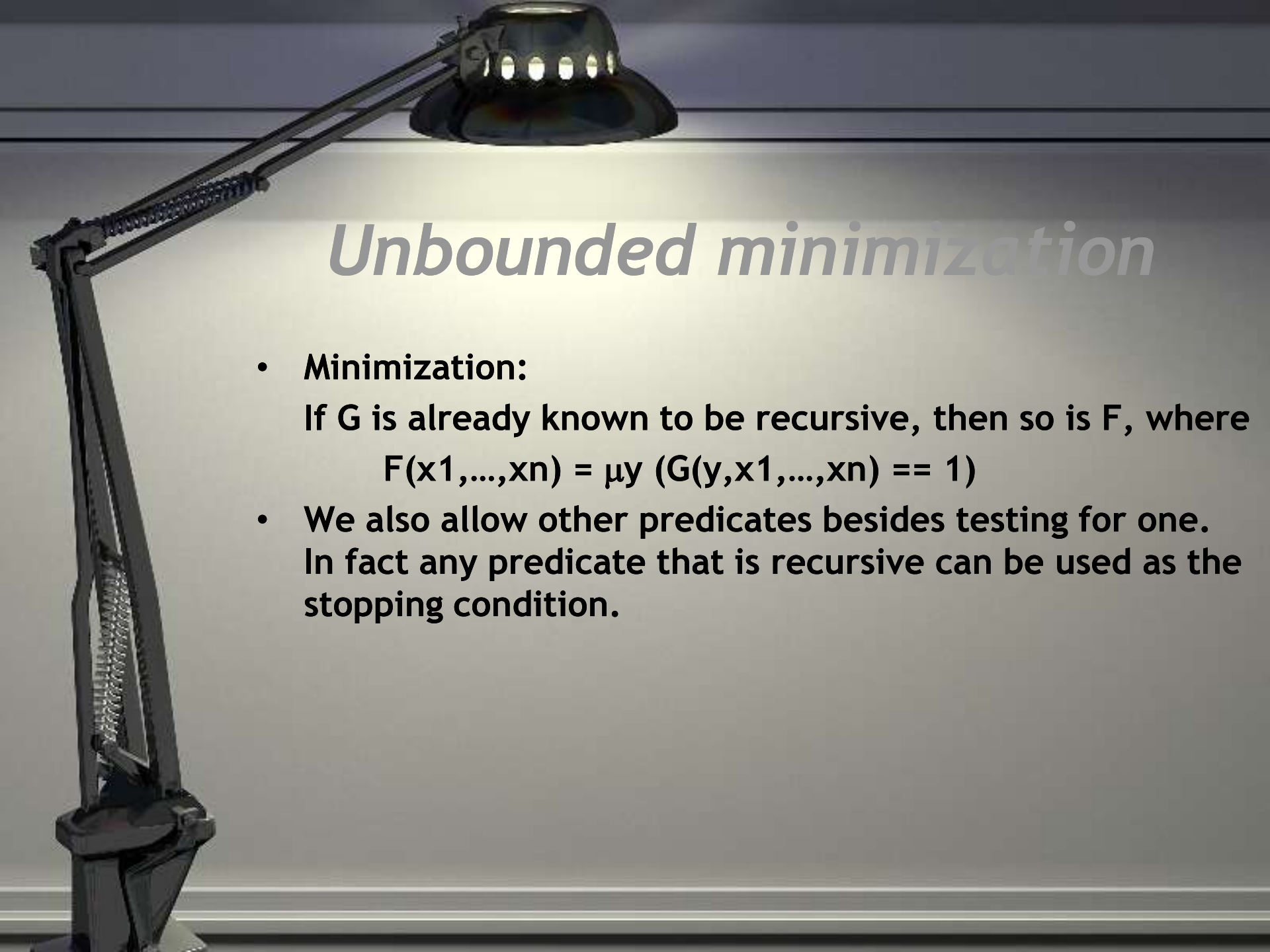
  $\langle x,y,z \rangle = \langle x, \langle y,z \rangle \rangle$ (note: stack analogy)



Incompleteness

The primitive recursive functions are all algorithms (they halt on all input). For this reason, we know that the primitive recursive functions are incomplete.

To create a complete model, we need some form of potentially unbounded iteration. That will be provided by an operation called minimization, in which we do not set a bound. This extends the primitive recursive functions to the (partial) recursive functions. Partial just means that these functions might diverge on some inputs. We contrast that with total recursive, the subset of recursive functions that converge everywhere (are algorithms).



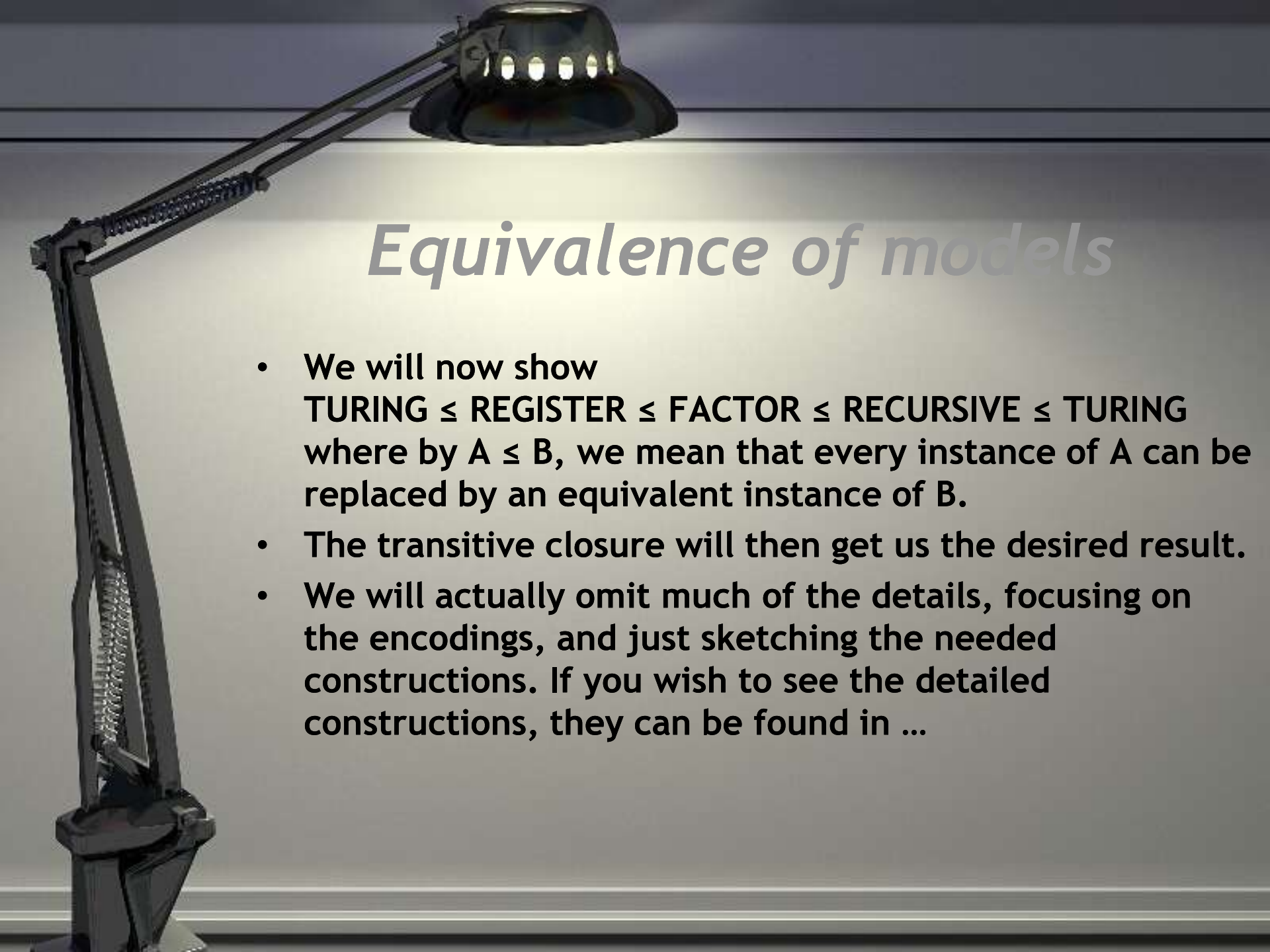
Unbounded minimization

- **Minimization:**

  If G is already known to be recursive, then so is F, where

  $F(x1,\ldots,xn) = \mu y\ (G(y,x1,\ldots,xn) == 1)$

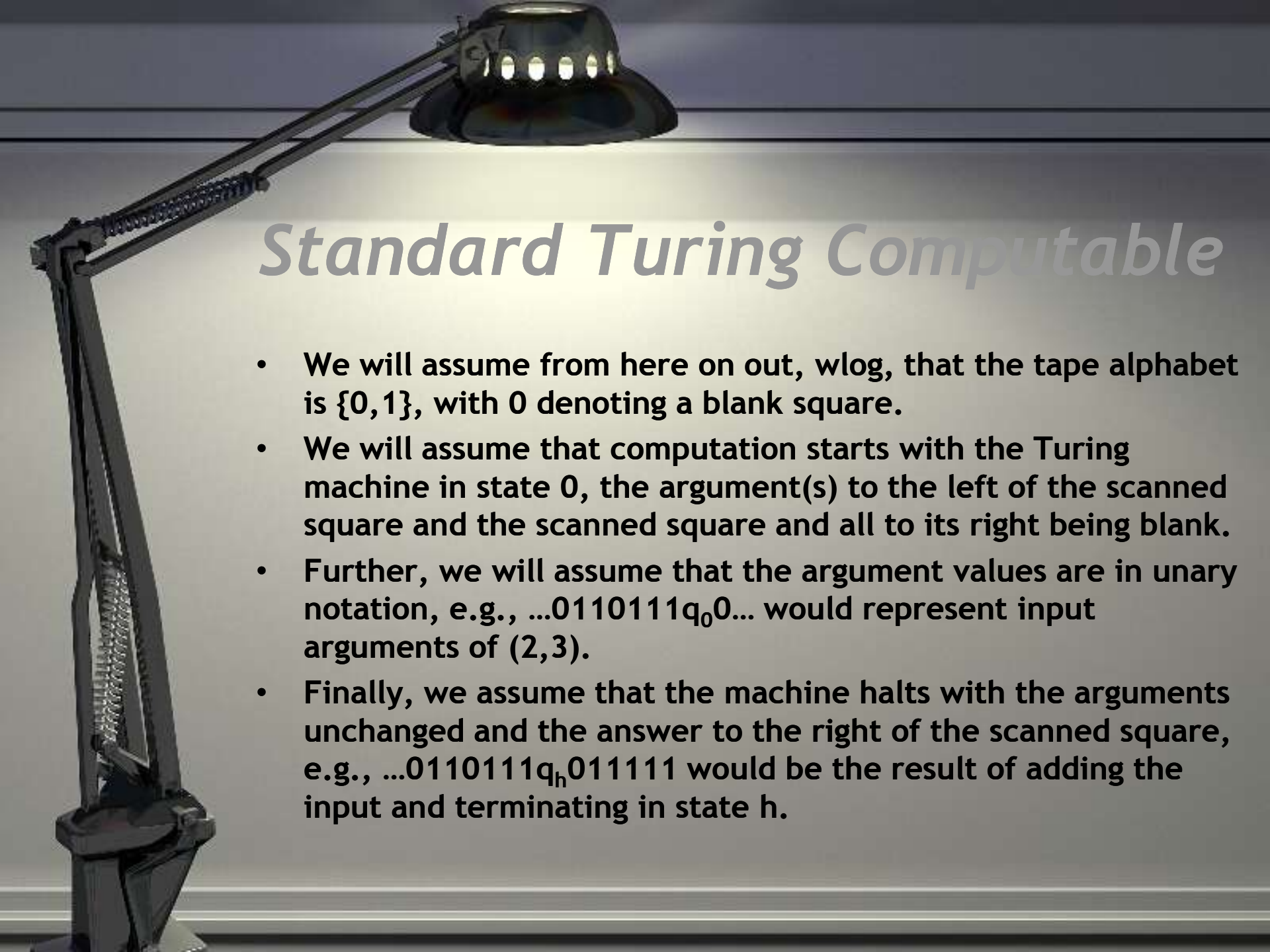
- **We also allow other predicates besides testing for one. In fact any predicate that is recursive can be used as the stopping condition.**



Equivalence of models

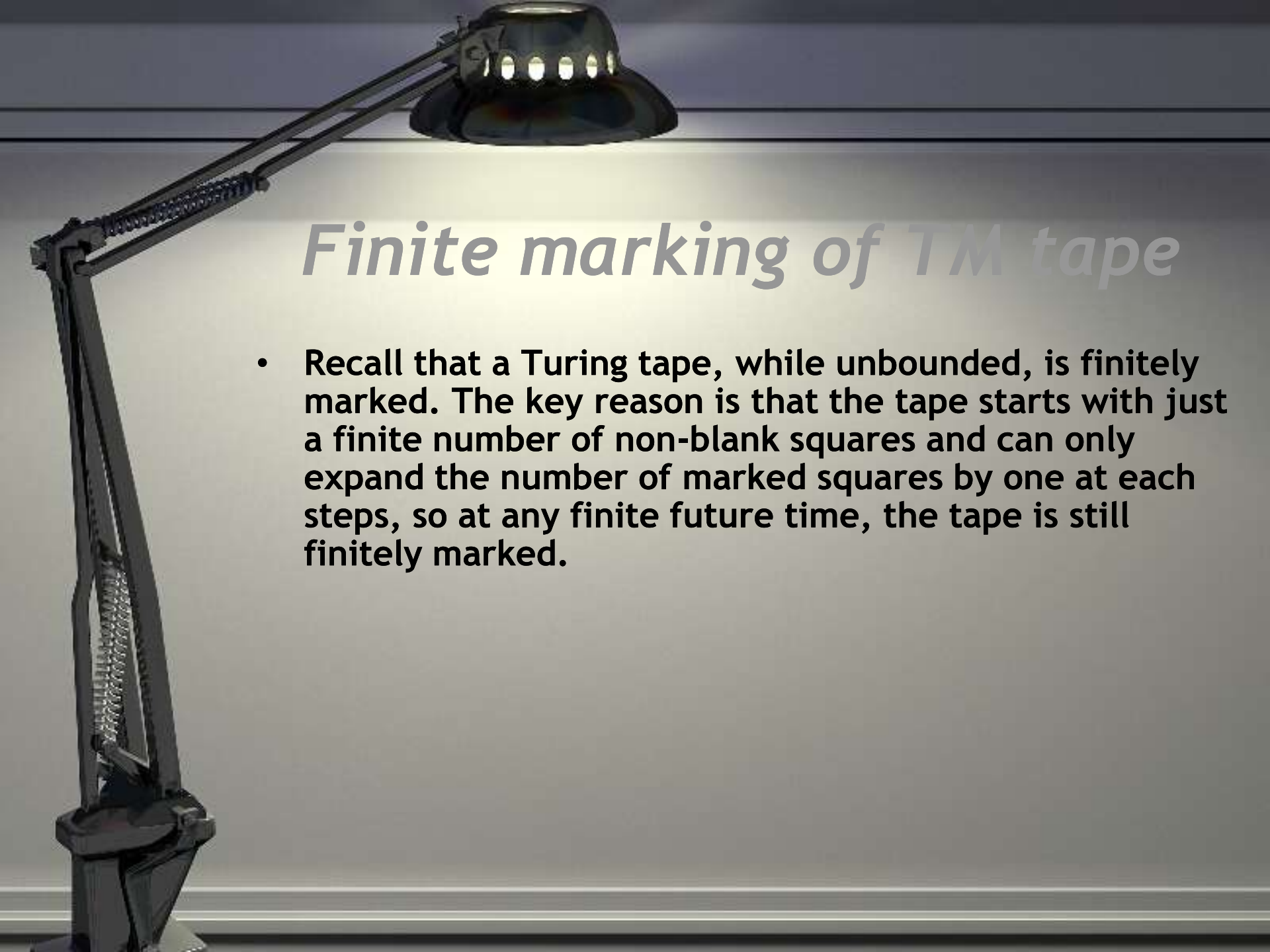
- We will now show
  TURING ≤ REGISTER ≤ FACTOR ≤ RECURSIVE ≤ TURING
  where by A ≤ B, we mean that every instance of A can be replaced by an equivalent instance of B.
- The transitive closure will then get us the desired result.
- We will actually omit much of the details, focusing on the encodings, and just sketching the needed constructions. If you wish to see the detailed constructions, they can be found in …


TURING ≤ REGISTER



Standard Turing Computable

- We will assume from here on out, wlog, that the tape alphabet is {0,1}, with 0 denoting a blank square.
- We will assume that computation starts with the Turing machine in state 0, the argument(s) to the left of the scanned square and the scanned square and all to its right being blank.
- Further, we will assume that the argument values are in unary notation, e.g., …0110111$q_0$0… would represent input arguments of (2,3).
- Finally, we assume that the machine halts with the arguments unchanged and the answer to the right of the scanned square, e.g., …0110111q_h011111 would be the result of adding the input and terminating in state h.



Finite marking of TM tape

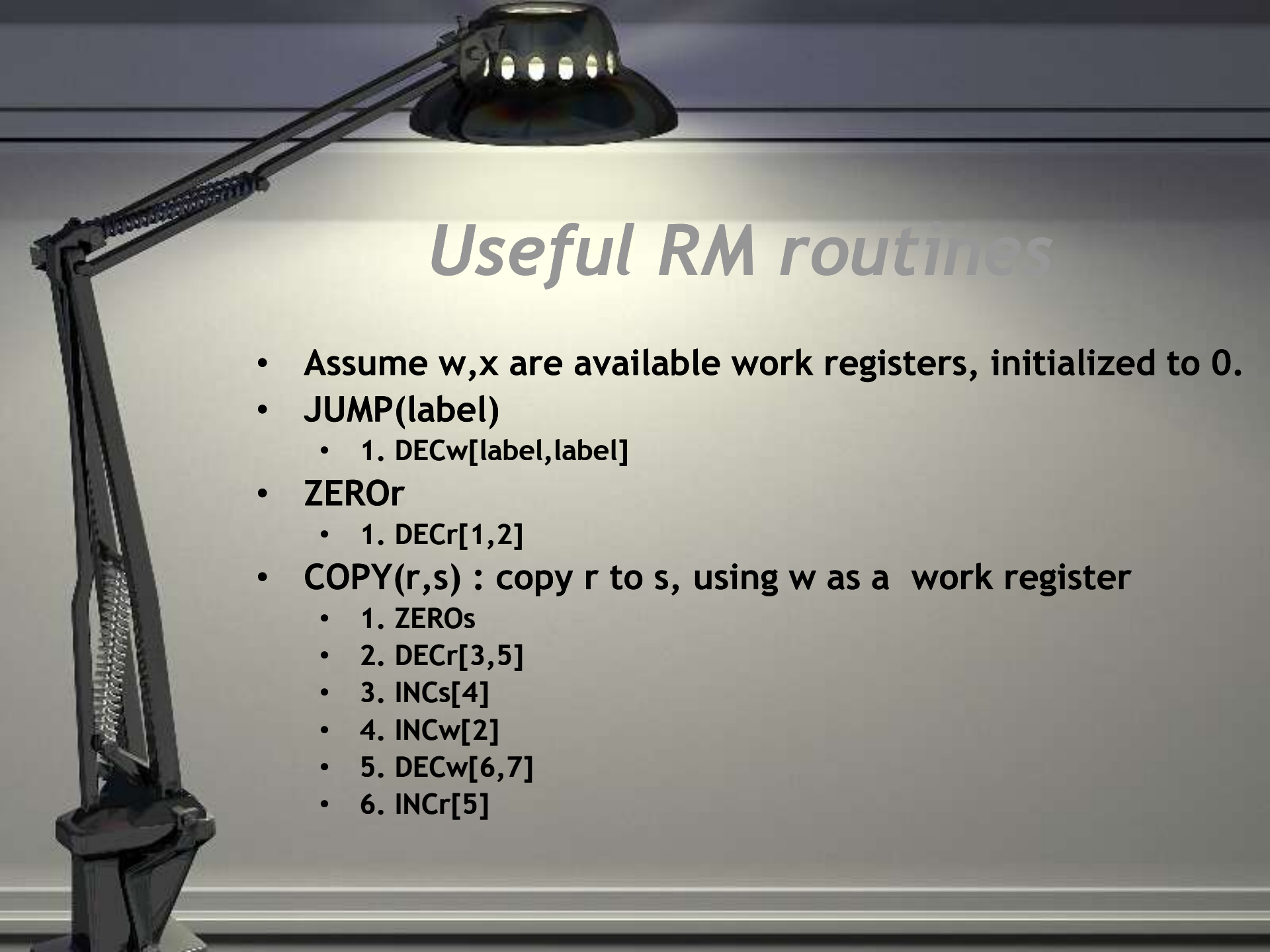
- Recall that a Turing tape, while unbounded, is finitely marked. The key reason is that the tape starts with just a finite number of non-blank squares and can only expand the number of marked squares by one at each steps, so at any finite future time, the tape is still finitely marked.

Encoding a Turing Machine

- For any model of computation, we require a finite representation of the machine's current status, called an instantaneous description (id). For a Turing Machine, we need to represent the squares to the left of the scanned square; the scanned square and all those to its right; and the current state.
- To see how this can be done, consider a machine that is in state 7, with its tape containing
  … 0 0 1 0 1 0 0 1 1 q_7 $\underline{0}$ 1 0 …
- The underscore indicates the square being read. We denote this by the finite id
  1 0 1 0 0 1 1 q_7 $\underline{0}$ 1
- In this notation, we always write down the scanned square, even if it and all symbols to its right are blank.

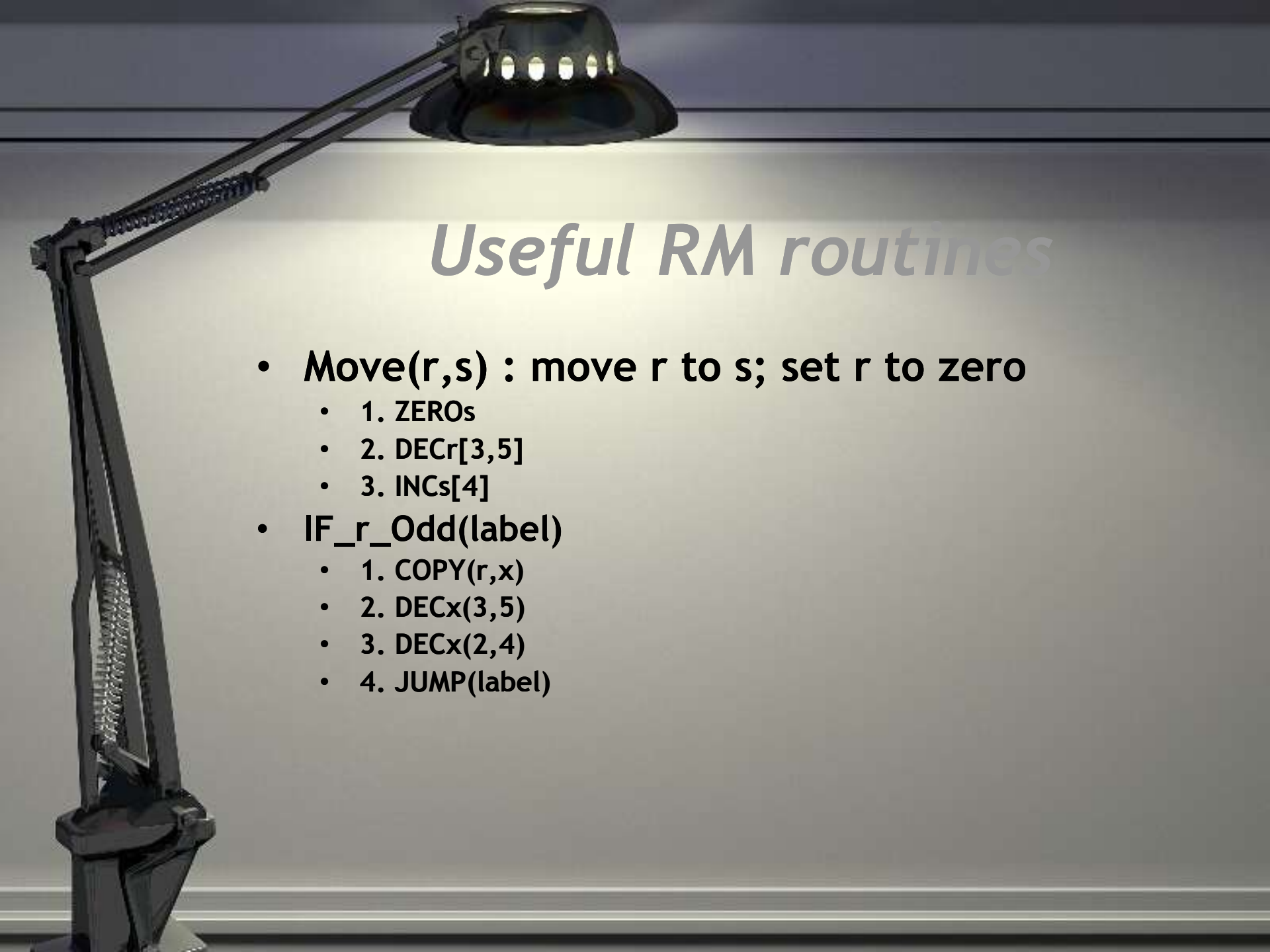
Encoding a Turing Machine

- An id can be represented by a triple of natural numbers, (L,R,i), where L is the number denoted by the binary sequence to the left, R is the number denoted by the reversal of the binary sequence to the right of the qi, and i is the state index (assume n states, 0..n-1).
- So,
  … 0 0 1 0 1 0 0 1 1 q_7 $\underline{0}$ 0 0 …
  is just (83, 0, 7).
  … 0 0 1 0 q_5 $\underline{1}$ 0 1 1 0 0 …
  is represented as (2, 13, 5).
- We can store the R part in register 1, the L part in register 2, and the state index in register 3 of a Register Machine.



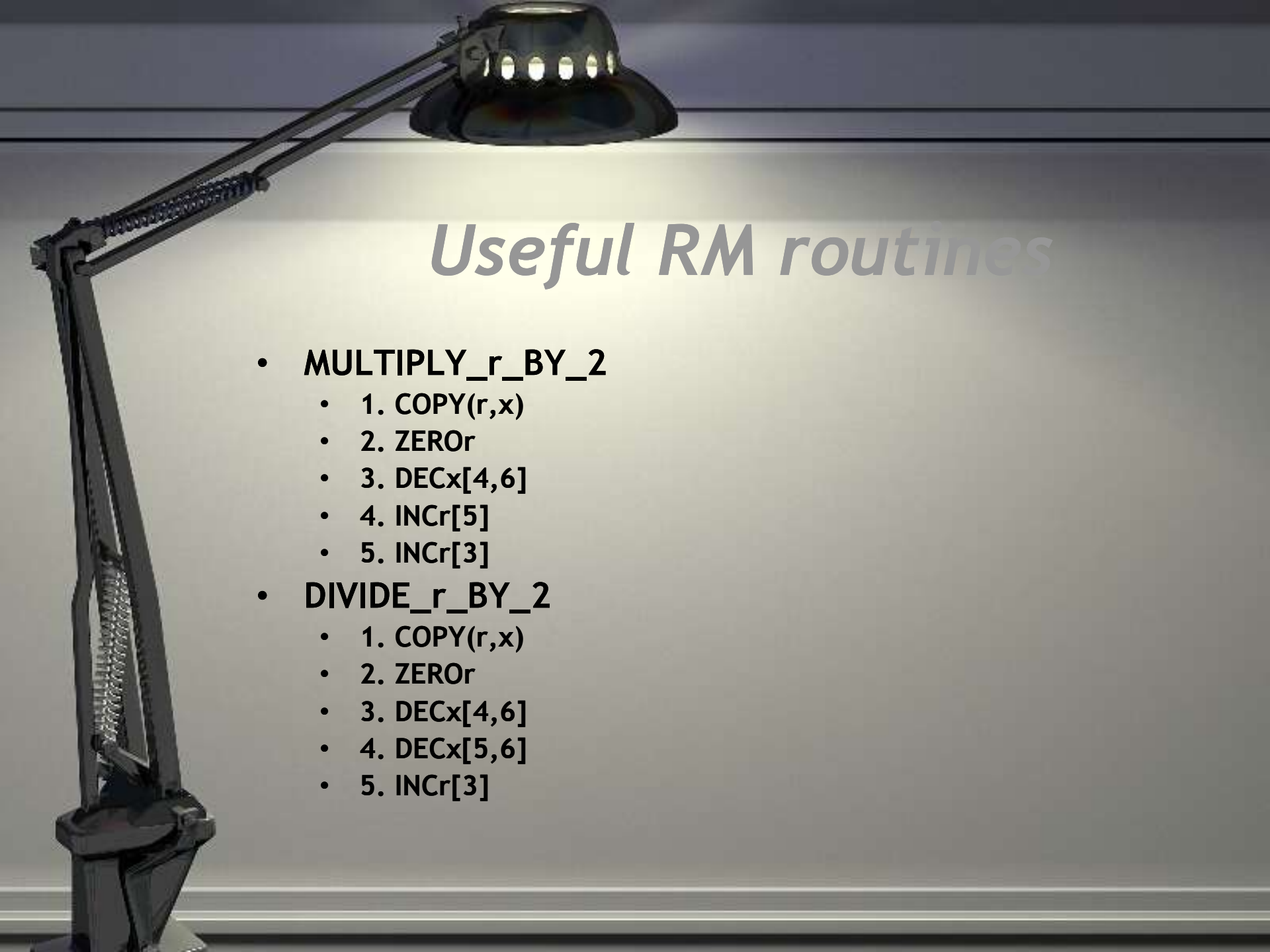
Useful RM routines

- Assume w,x are available work registers, initialized to 0.
- JUMP(label)
  - 1. DECw[label,label]
- ZEROr
  - 1. DECr[1,2]
- COPY(r,s) : copy r to s, using w as a work register
  - 1. ZEROs
  - 2. DECr[3,5]
  - 3. INCs[4]
  - 4. INCw[2]
  - 5. DECw[6,7]
  - 6. INCr[5]



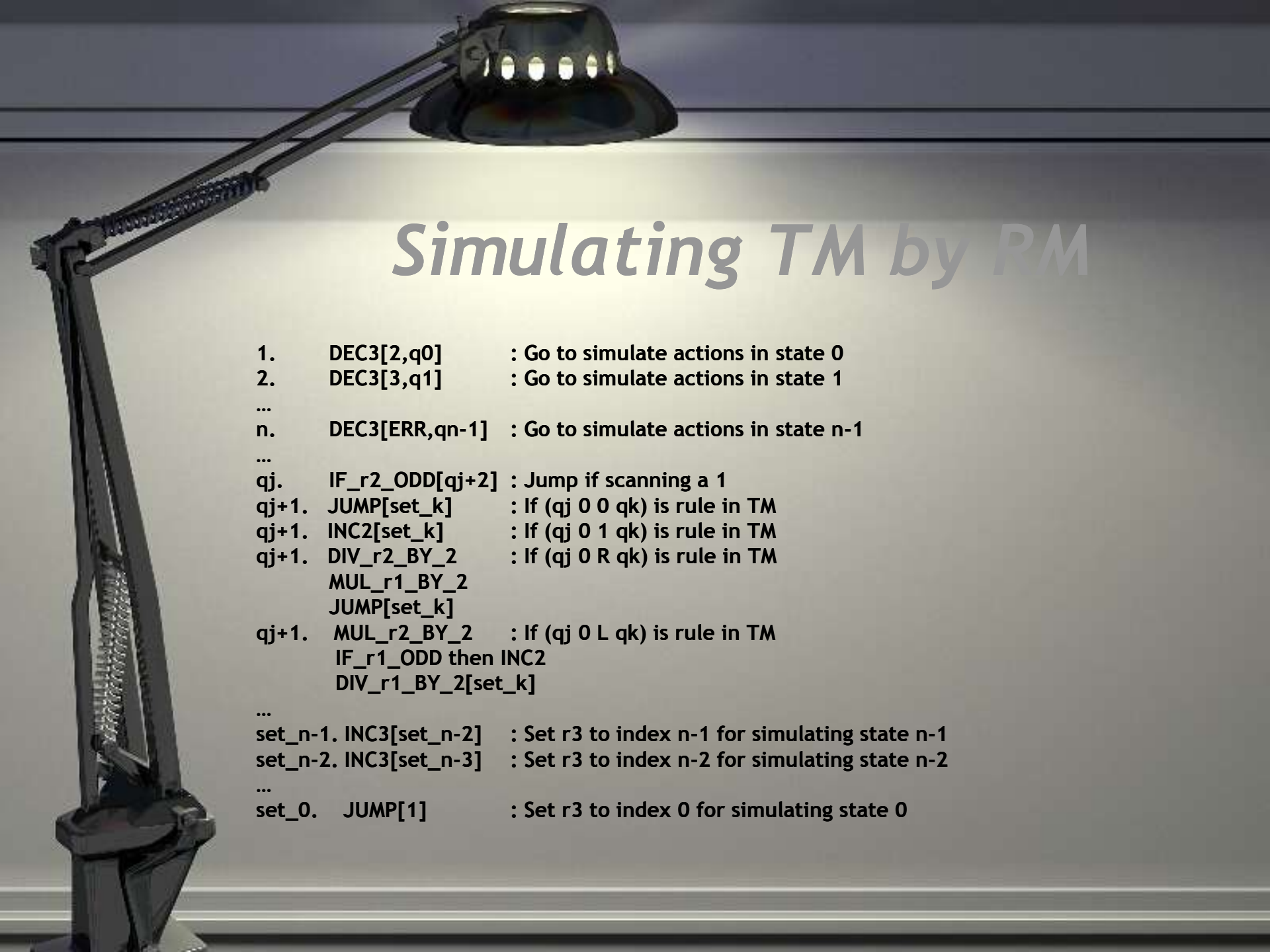
Useful RM routines

- Move(r,s) : move r to s; set r to zero
  - 1. ZEROs
  - 2. DECr[3,5]
  - 3. INCs[4]
- IF_r_Odd(label)
  - 1. COPY(r,x)
  - 2. DECx(3,5)
  - 3. DECx(2,4)
  - 4. JUMP(label)



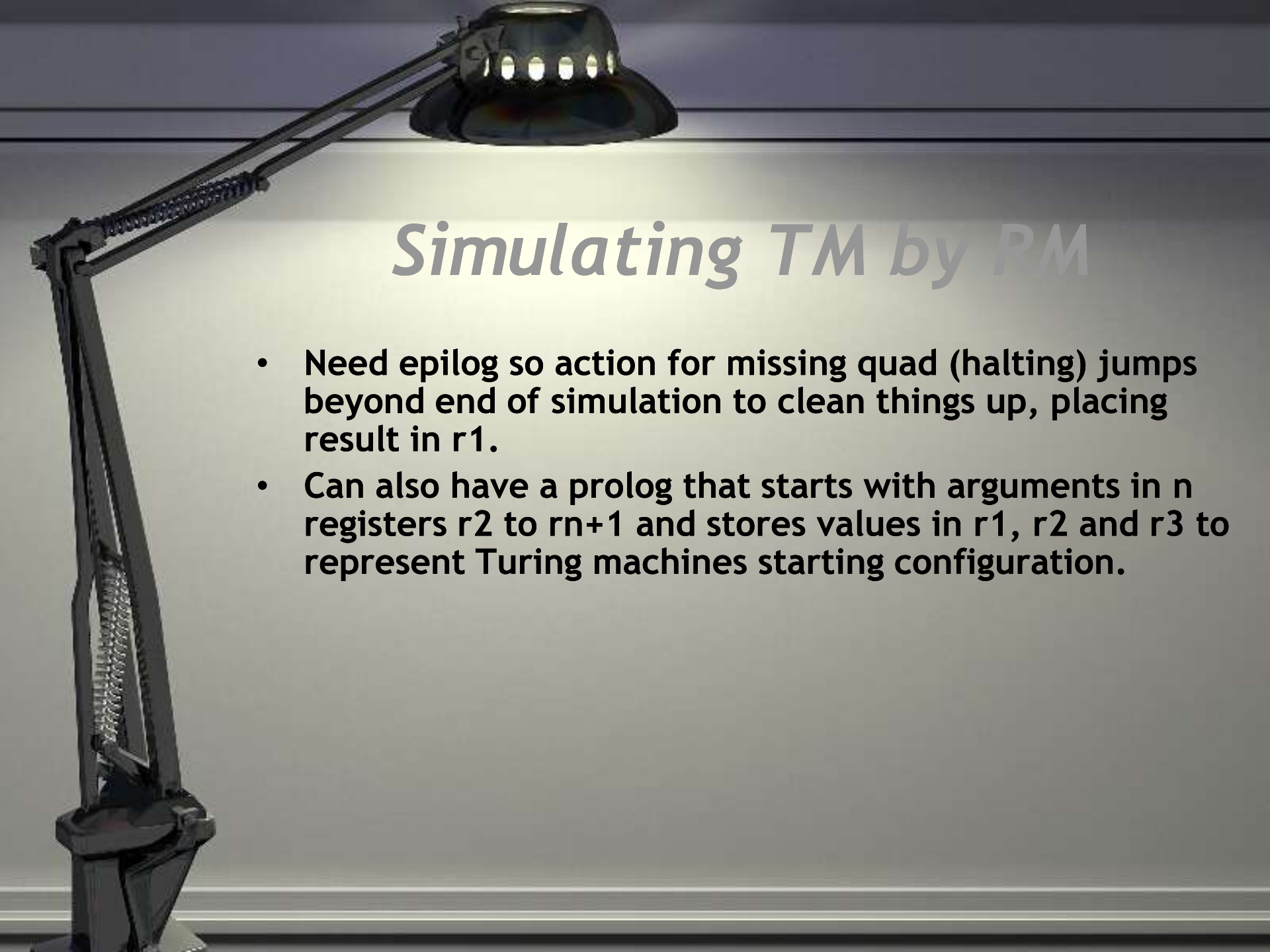
Useful RM routines

- MULTIPLY_r_BY_2
  - 1. COPY(r,x)
  - 2. ZEROr
  - 3. DECx[4,6]
  - 4. INCr[5]
  - 5. INCr[3]
- DIVIDE_r_BY_2
  - 1. COPY(r,x)
  - 2. ZEROr
  - 3. DECx[4,6]
  - 4. DECx[5,6]
  - 5. INCr[3]



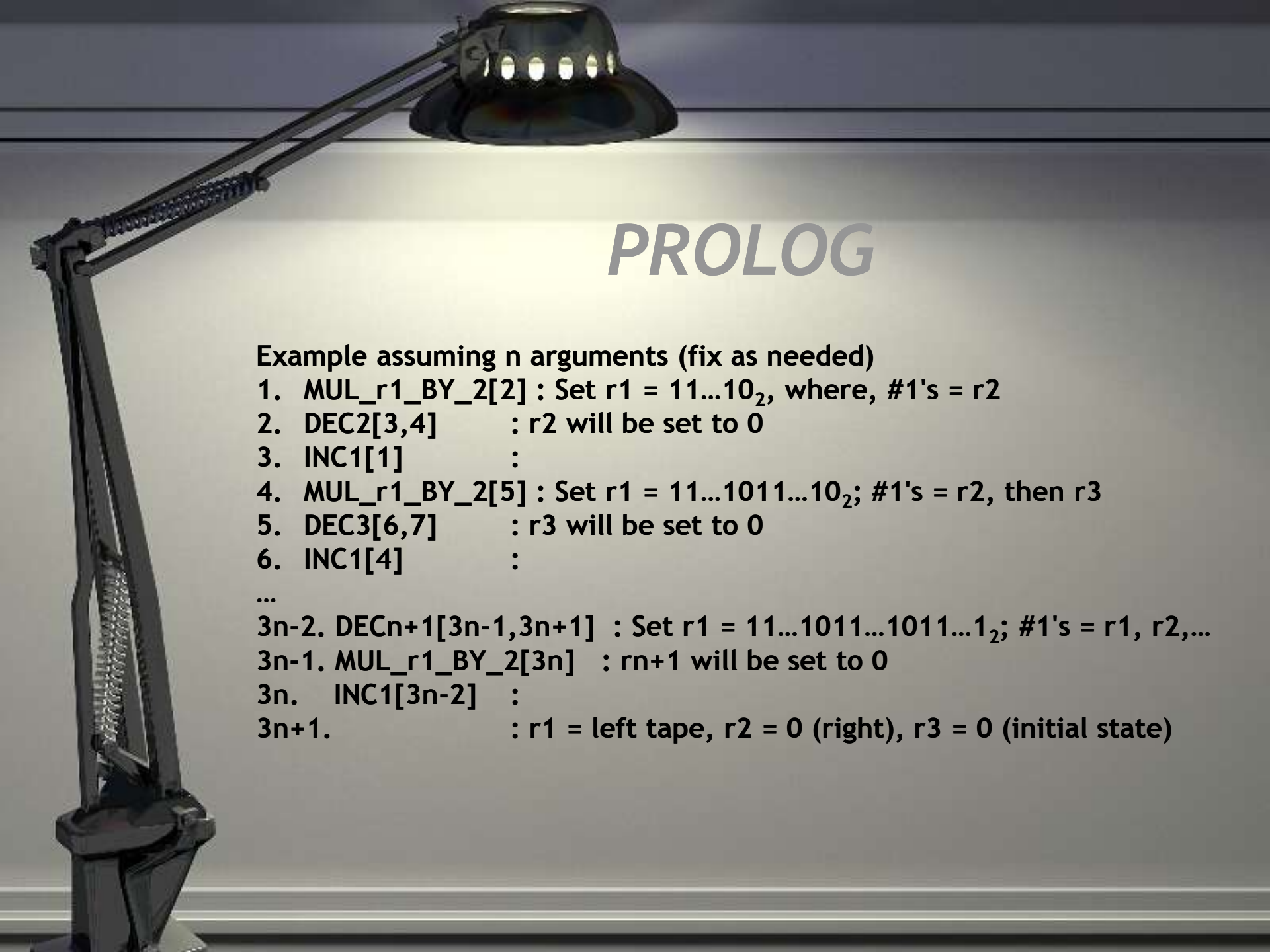
Simulating TM by RM

```
1.      DEC3[2,q0]          : Go to simulate actions in state 0
2.      DEC3[3,q1]          : Go to simulate actions in state 1
…
n.      DEC3[ERR,qn-1]      : Go to simulate actions in state n-1
…
qj.     IF_r2_ODD[qj+2]     : Jump if scanning a 1
qj+1.   JUMP[set_k]         : If (qj 0 0 qk) is rule in TM
qj+1.   INC2[set_k]         : If (qj 0 1 qk) is rule in TM
qj+1.   DIV_r2_BY_2         : If (qj 0 R qk) is rule in TM
        MUL_r1_BY_2
        JUMP[set_k]
qj+1.    MUL_r2_BY_2        : If (qj 0 L qk) is rule in TM
         IF_r1_ODD then INC2
         DIV_r1_BY_2[set_k]
…
set_n-1. INC3[set_n-2]      : Set r3 to index n-1 for simulating state n-1
set_n-2. INC3[set_n-3]      : Set r3 to index n-2 for simulating state n-2
…
set_0.   JUMP[1]            : Set r3 to index 0 for simulating state 0
```



Simulating TM by RM

- Need epilog so action for missing quad (halting) jumps beyond end of simulation to clean things up, placing result in r1.
- Can also have a prolog that starts with arguments in n registers r2 to rn+1 and stores values in r1, r2 and r3 to represent Turing machines starting configuration.



PROLOG

Example assuming n arguments (fix as needed)

1. MUL_r1_BY_2[2] : Set r1 = 11…10_2, where, #1's = r2
2. DEC2[3,4] : r2 will be set to 0
3. INC1[1] :
4. MUL_r1_BY_2[5] : Set r1 = 11…1011…10_2; #1's = r2, then r3
5. DEC3[6,7] : r3 will be set to 0
6. INC1[4] :

…

3n-2. DECn+1[3n-1,3n+1] : Set r1 = 11…1011…1011…1_2; #1's = r1, r2,…
3n-1. MUL_r1_BY_2[3n] : rn+1 will be set to 0
3n. INC1[3n-2] :
3n+1. : r1 = left tape, r2 = 0 (right), r3 = 0 (initial state)


REGISTER ≤ FACTOR

Encoding an RM's id

- This is a really easy one based on the fact that every member of Z^+ (the positive integers) has a unique prime factorization. Thus all such numbers can be uniquely written in the form

  where the p_i's are distinct primes and the k_i's are non-zero values, except that the number 1 would be represented by 2^0.

- Let R be an arbitrary n-register machine, having m instructions.

  Encode the contents of registers r1,…,rn by the powers of $p_1,…p_n$.

  Encode rule number's 1…m by primes p_{n+1} ,…, p_{n+m}

  Use pn+m+1 as prime factor that indicates simulation is done.

- This is in essence the Gödel number of the RM's state.

Simulation by FRS

- Now, the j-th instruction ($1 \le j \le m$) of R has associated factor replacement rules as follows:

  j. INCr[i]

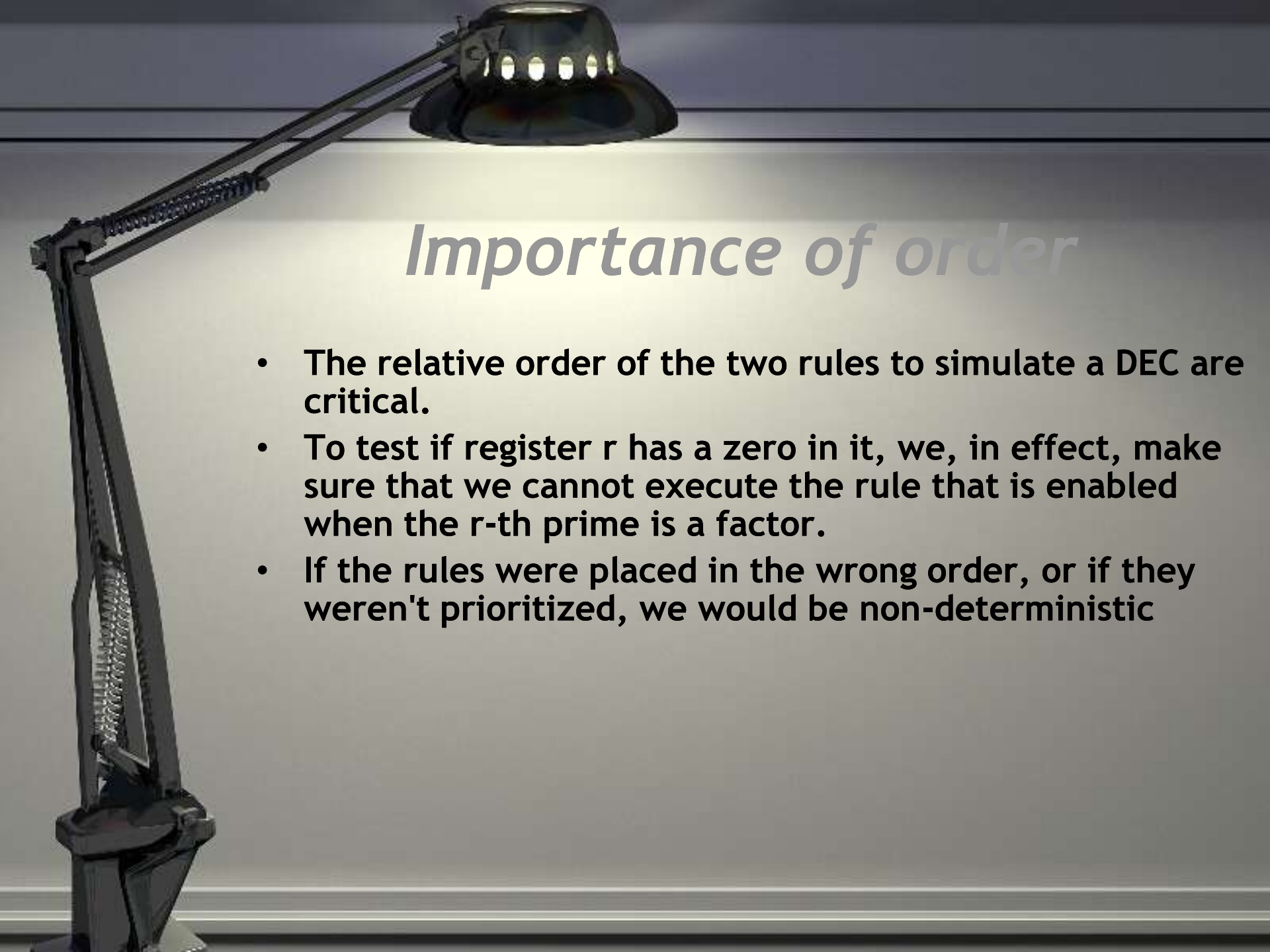
  $$p_{n+j}x \rightarrow p_{n+i}p_r x$$

  j. DECr[s, f]

  $$p_{n+j}p_r x \rightarrow p_{n+s}x$$
  $$p_{n+j}x \rightarrow p_{n+f}x$$

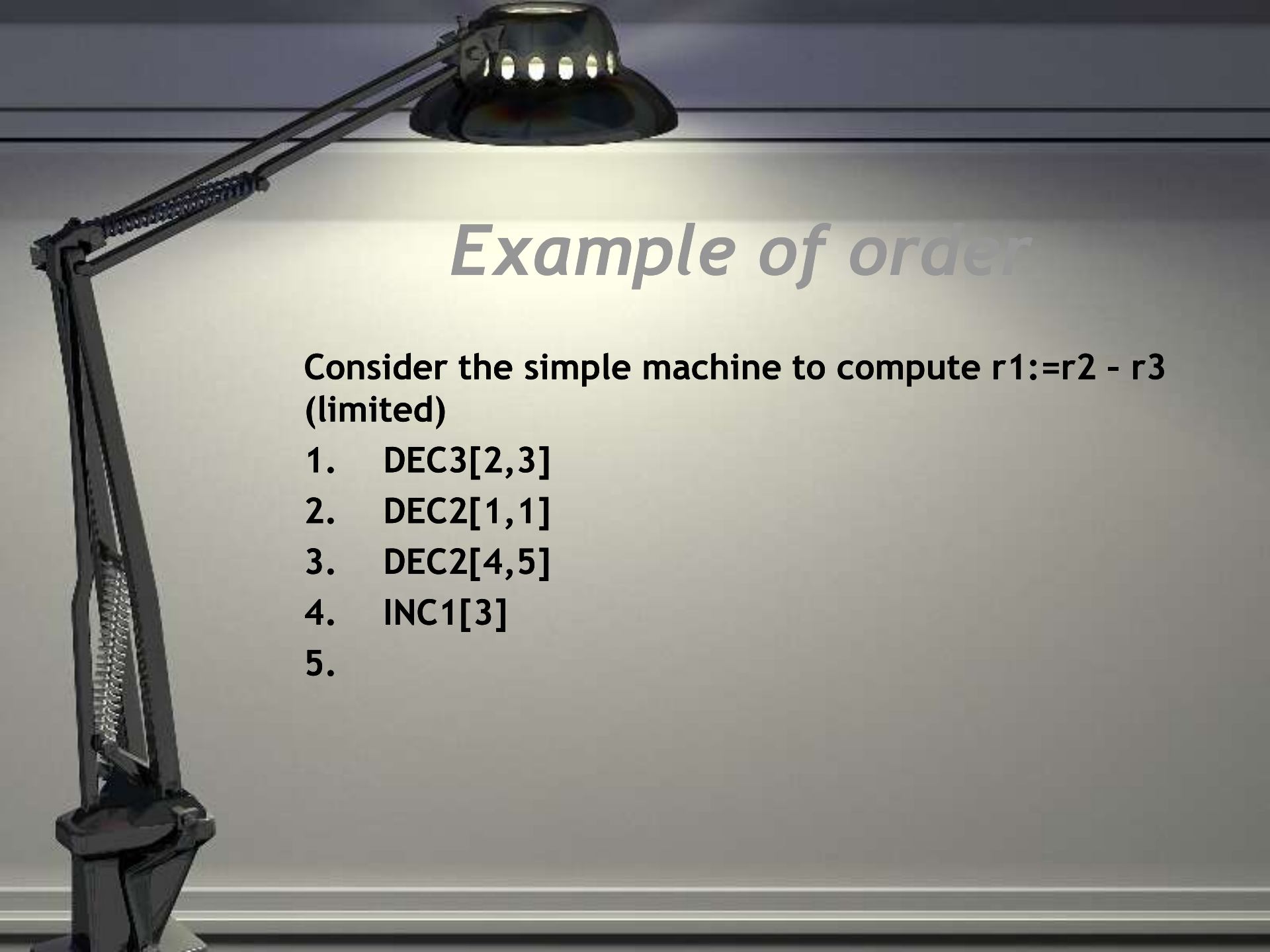
- We also add the halting rule associated with m+1 of

  $$p_{n+m+1}x \rightarrow x$$



Importance of order

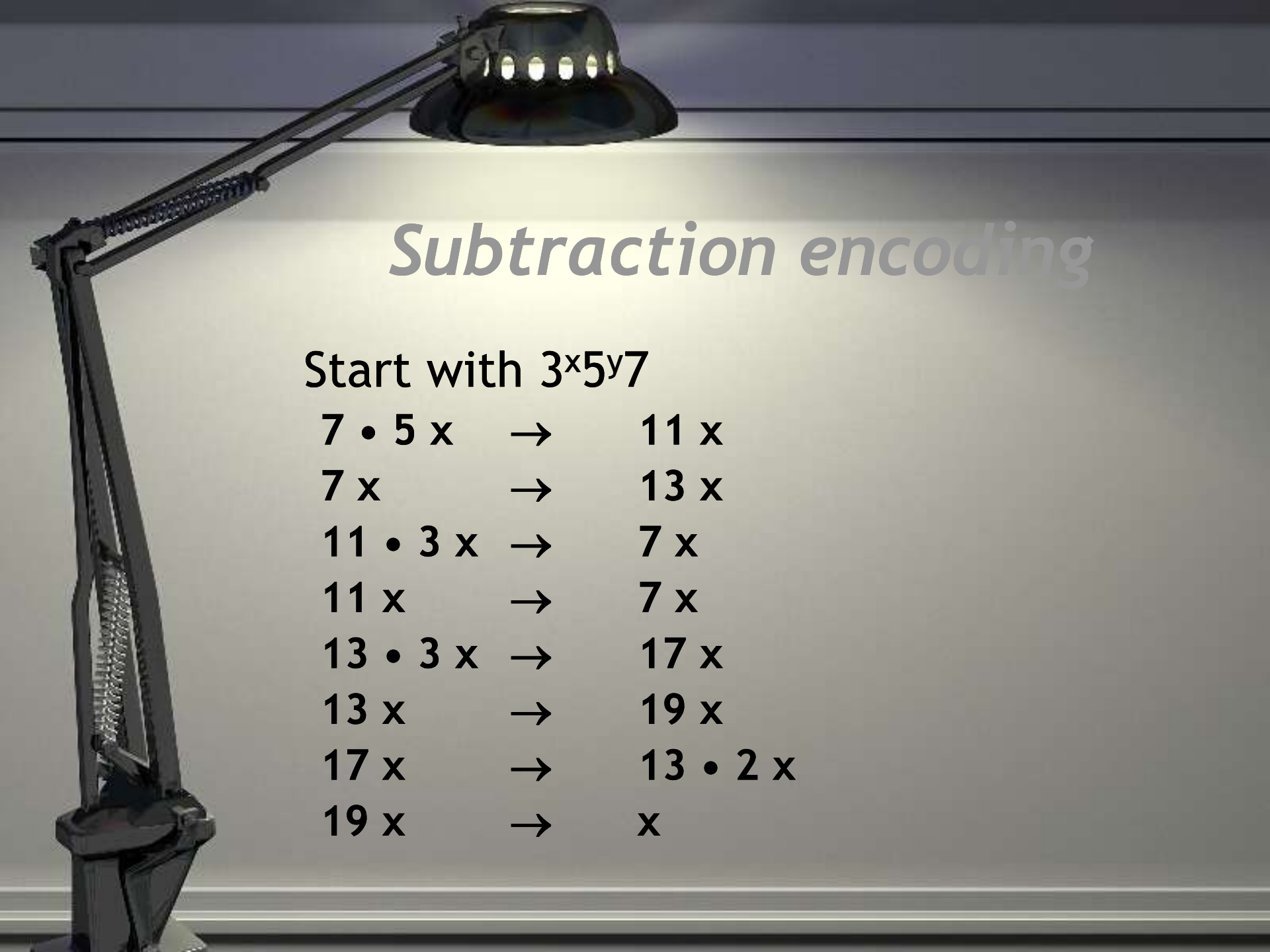
- The relative order of the two rules to simulate a DEC are critical.
- To test if register r has a zero in it, we, in effect, make sure that we cannot execute the rule that is enabled when the r-th prime is a factor.
- If the rules were placed in the wrong order, or if they weren't prioritized, we would be non-deterministic



Example of order

Consider the simple machine to compute r1:=r2 - r3 (limited)

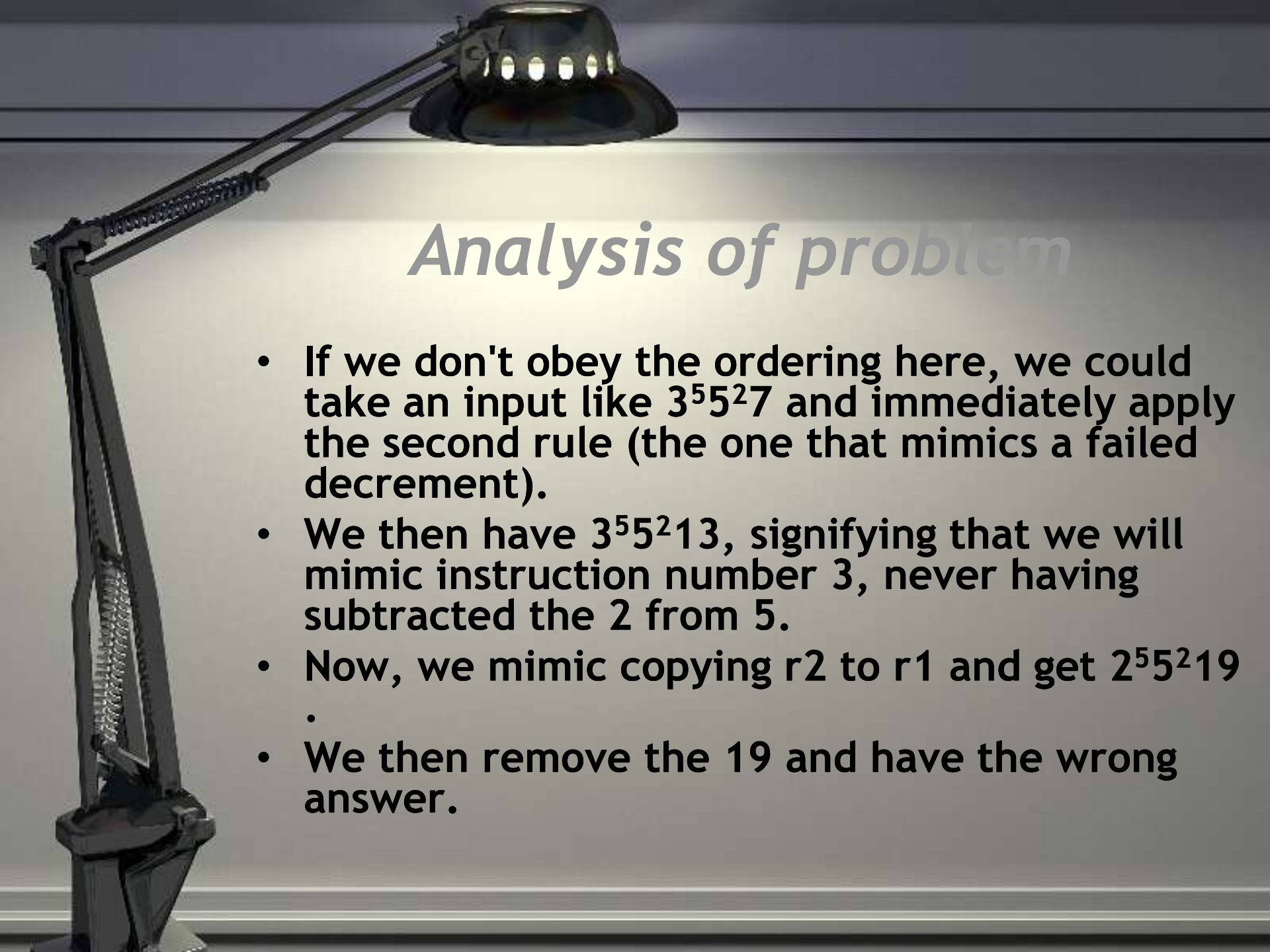
1. DEC3[2,3]
2. DEC2[1,1]
3. DEC2[4,5]
4. INC1[3]
5.



Subtraction encoding

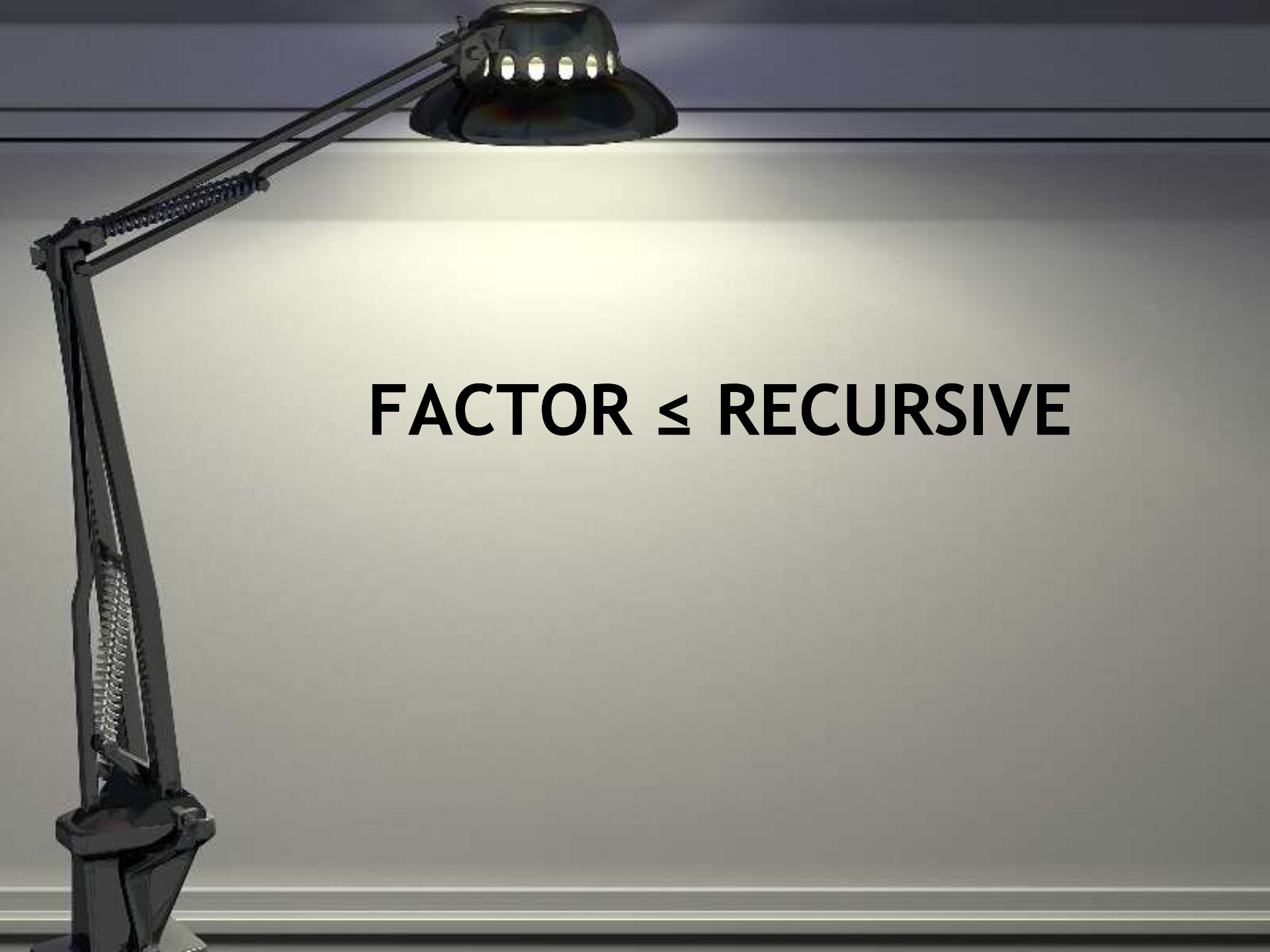
Start with 3^x5^y7

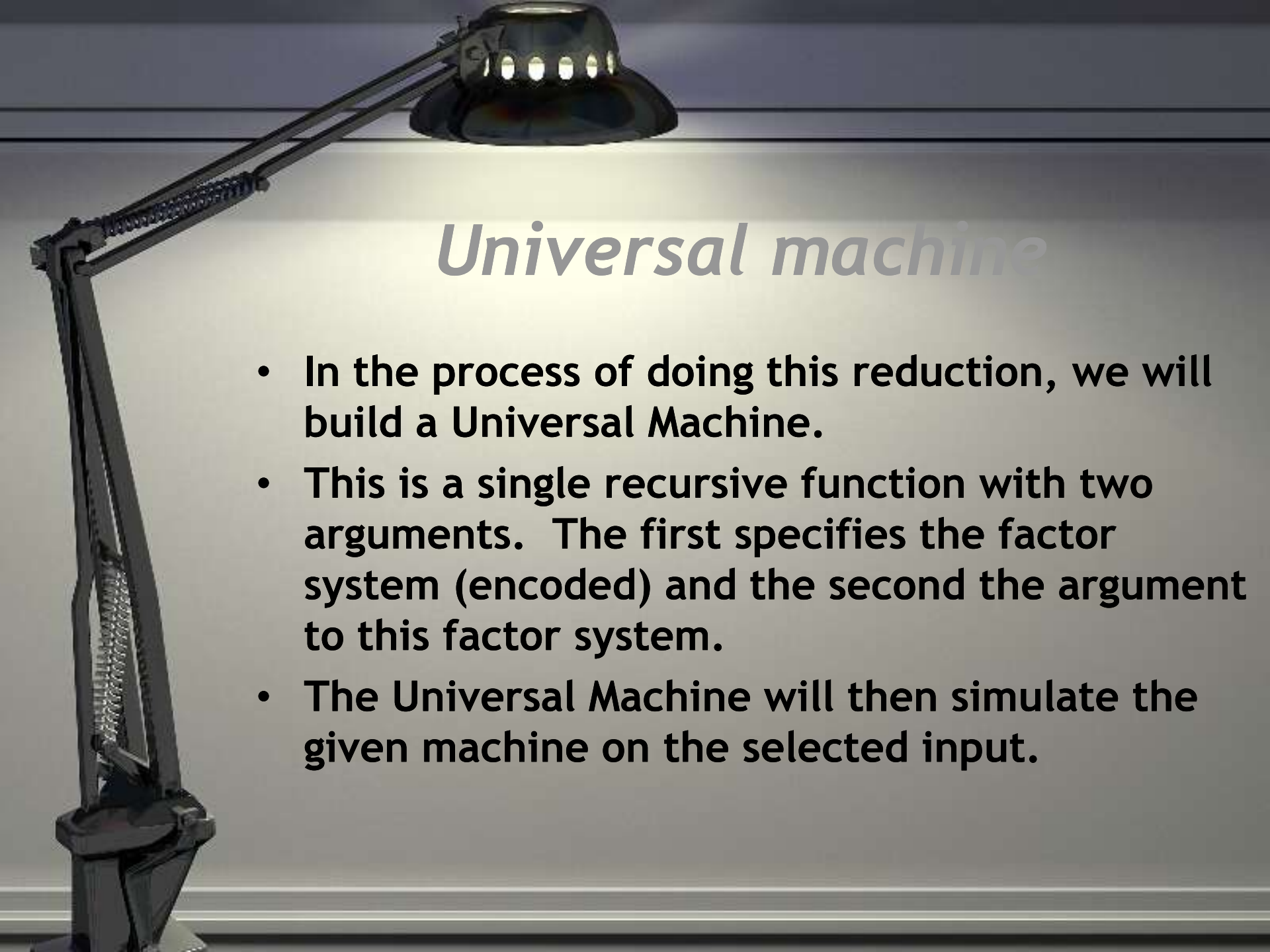
7 • 5 x	→	11 x
7 x	→	13 x
11 • 3 x	→	7 x
11 x	→	7 x
13 • 3 x	→	17 x
13 x	→	19 x
17 x	→	13 • 2 x
19 x	→	x



Analysis of problem

- If we don't obey the ordering here, we could take an input like $3^5 5^2 7$ and immediately apply the second rule (the one that mimics a failed decrement).
- We then have $3^5 5^2 13$, signifying that we will mimic instruction number 3, never having subtracted the 2 from 5.
- Now, we mimic copying r2 to r1 and get $2^5 5^2 19$.
- We then remove the 19 and have the wrong answer.


FACTOR ≤ RECURSIVE



Universal machine

- In the process of doing this reduction, we will build a Universal Machine.
- This is a single recursive function with two arguments. The first specifies the factor system (encoded) and the second the argument to this factor system.
- The Universal Machine will then simulate the given machine on the selected input.

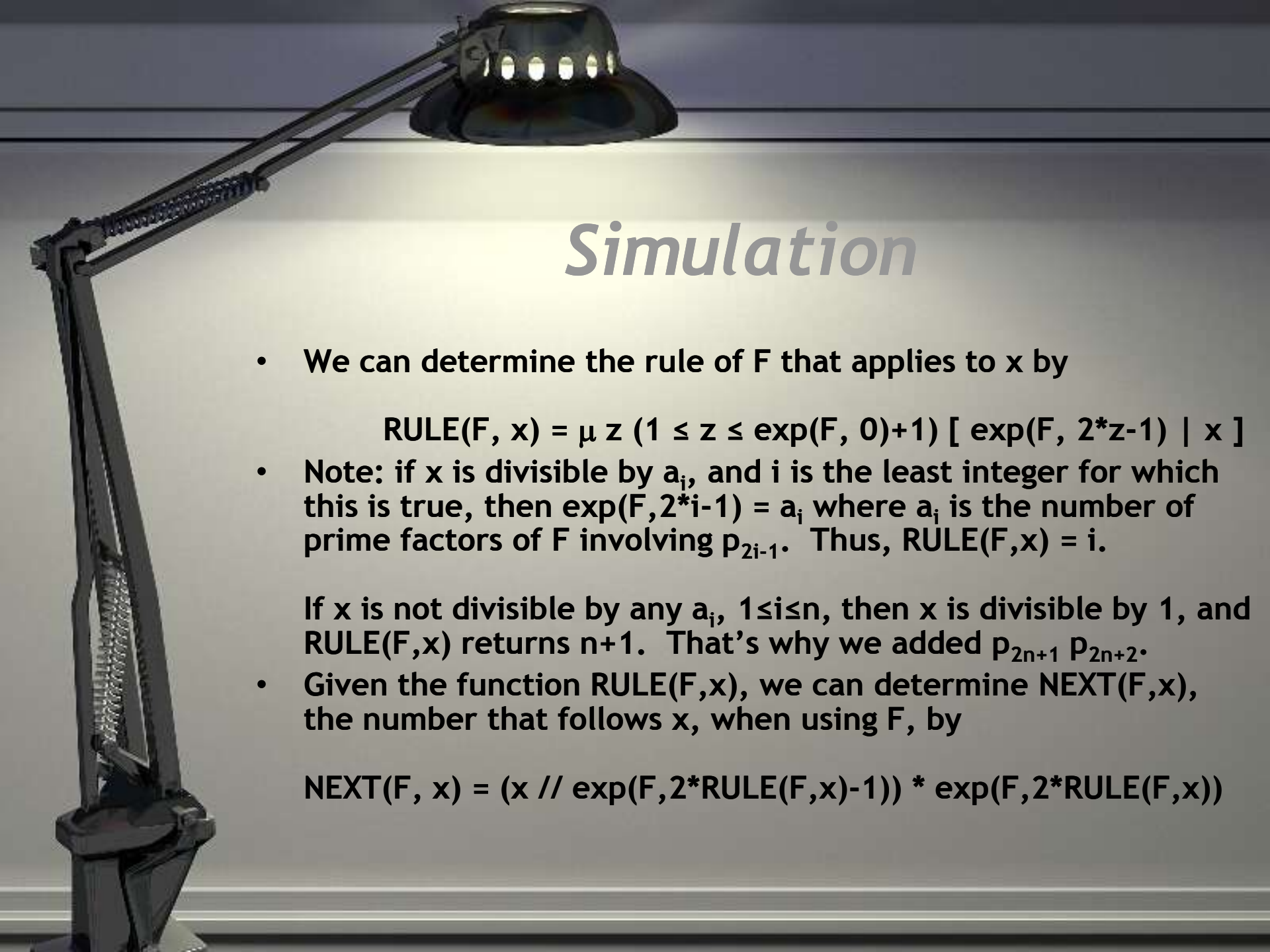
Encoding FRS

- Let (n, $((a_1,b_1), (a_2,b_2), \ldots, (a_n,b_n))$ be some factor replacement system, where (a_i,b_i) means that the i-th rule is

  $a_i x \quad \rightarrow \quad b_i x$

- Encode this machine by the number F,

$$2^n 3^{a_1} 5^{b_1} 7^{a_2} 11^{b_2} \cdots p_{2n-1}^{a_n} p_{2n}^{b_n} p_{2n+1} p_{2n+2}$$



Simulation

- We can determine the rule of F that applies to x by

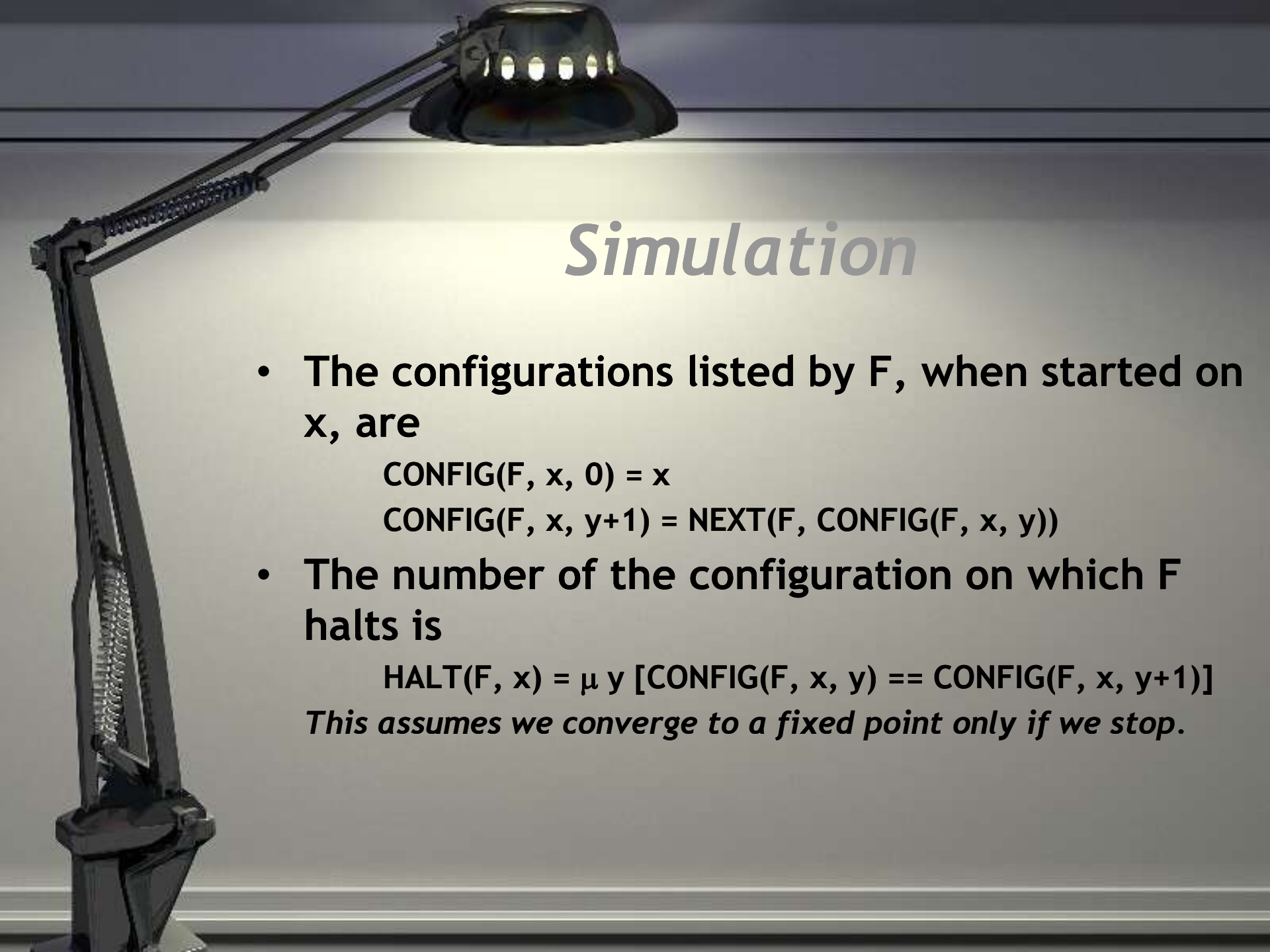
  $$RULE(F, x) = \mu z\ (1 \le z \le exp(F, 0)+1)\ [\ exp(F, 2*z-1)\ |\ x\ ]$$

- Note: if x is divisible by a_i, and i is the least integer for which this is true, then $exp(F,2*i-1) = a_i$ where a_i is the number of prime factors of F involving p_{2i-1}. Thus, RULE(F,x) = i.

  If x is not divisible by any a_i, $1 \le i \le n$, then x is divisible by 1, and RULE(F,x) returns n+1. That's why we added $p_{2n+1}\ p_{2n+2}$.

- Given the function RULE(F,x), we can determine NEXT(F,x), the number that follows x, when using F, by

  NEXT(F, x) = (x // exp(F,2*RULE(F,x)-1)) * exp(F,2*RULE(F,x))



Simulation

- The configurations listed by F, when started on x, are

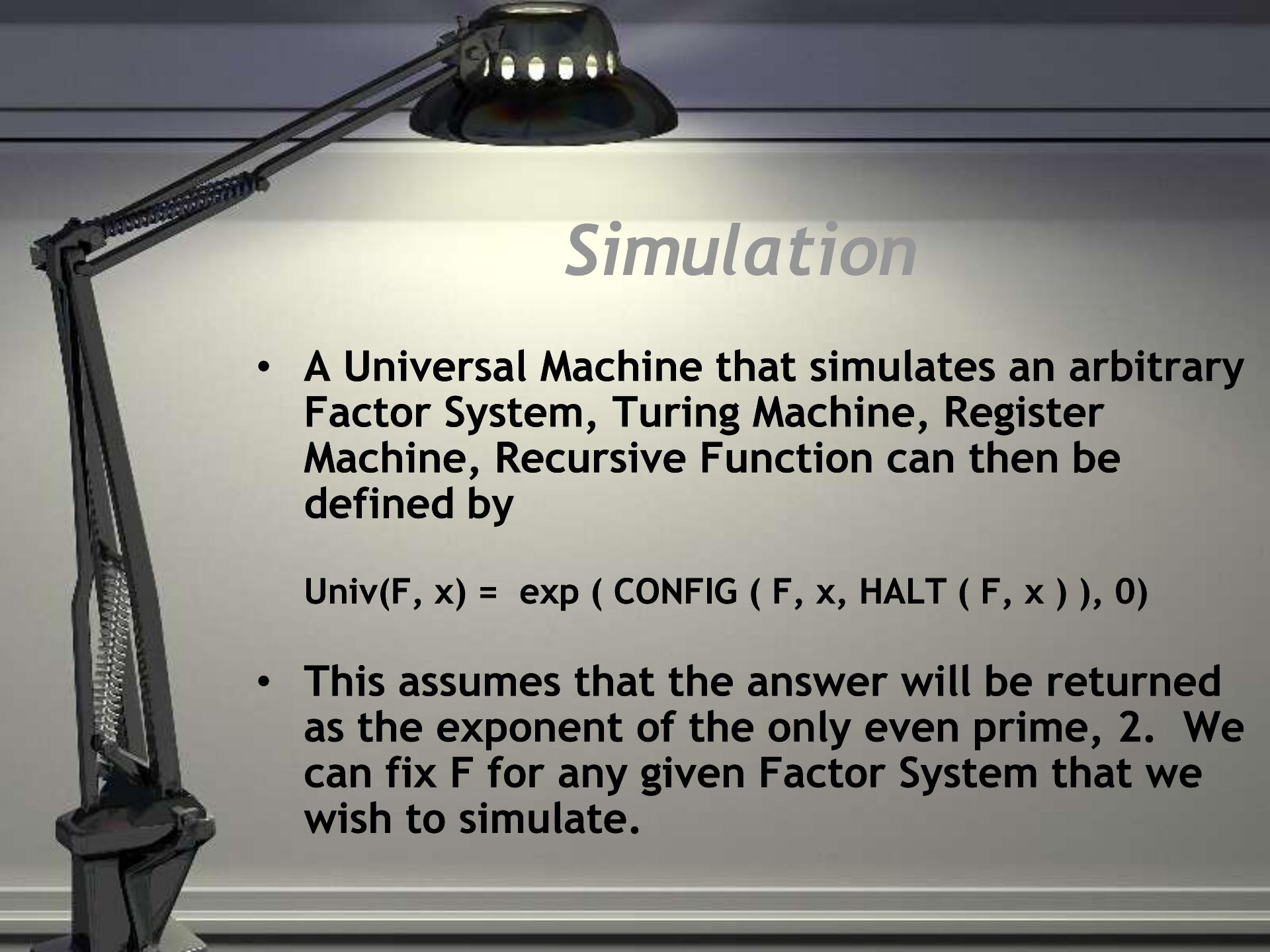
  CONFIG(F, x, 0) = x

  CONFIG(F, x, y+1) = NEXT(F, CONFIG(F, x, y))

- The number of the configuration on which F halts is

  HALT(F, x) = μ y [CONFIG(F, x, y) == CONFIG(F, x, y+1)]

  This assumes we converge to a fixed point only if we stop.

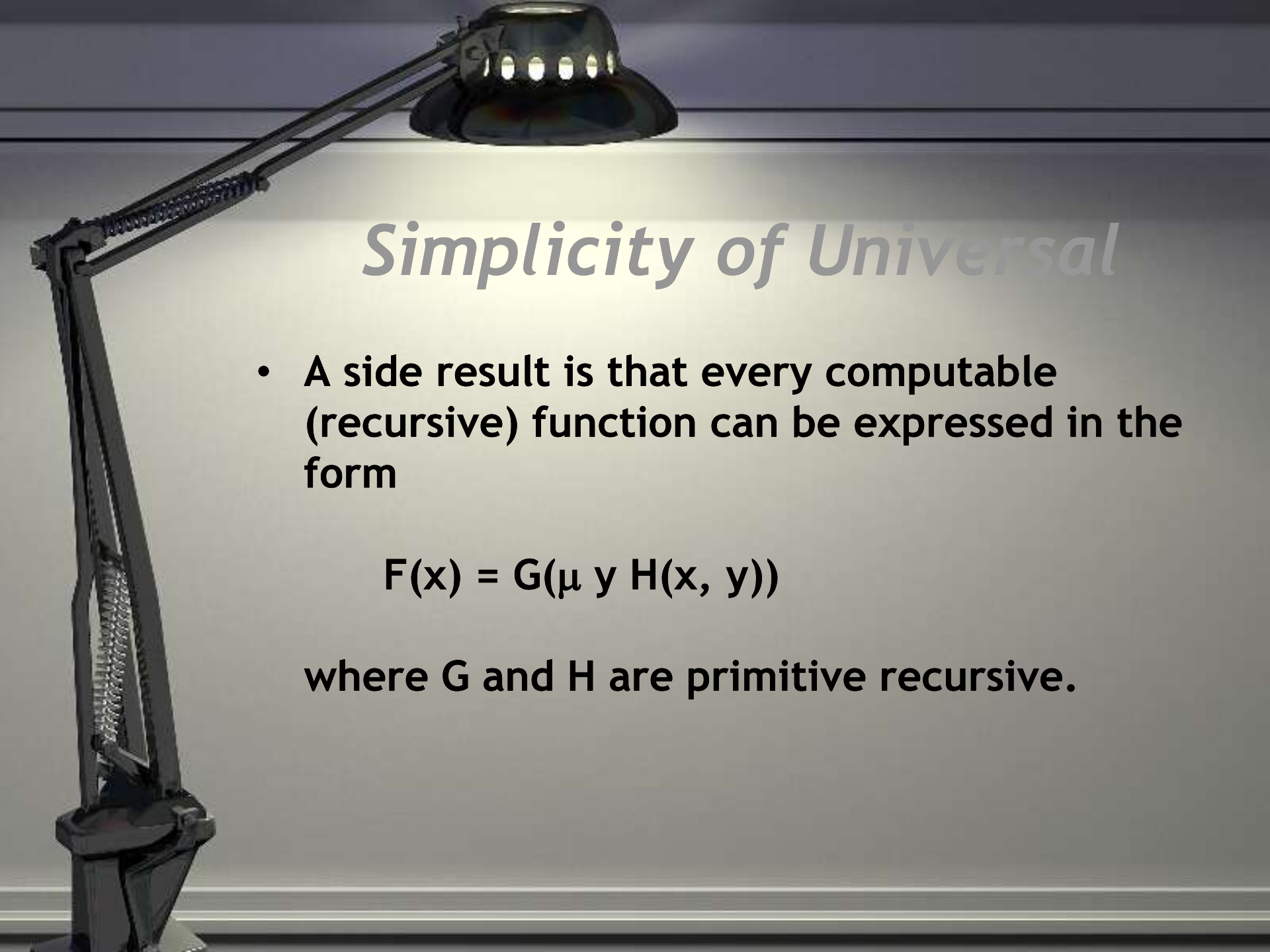


Simulation

- A Universal Machine that simulates an arbitrary Factor System, Turing Machine, Register Machine, Recursive Function can then be defined by

  Univ(F, x) = exp (CONFIG (F, x, HALT (F, x)), 0)

- This assumes that the answer will be returned as the exponent of the only even prime, 2. We can fix F for any given Factor System that we wish to simulate.



Simplicity of Universal

- A side result is that every computable (recursive) function can be expressed in the form

$$F(x) = G(\mu\, y\, H(x, y))$$

where G and H are primitive recursive.

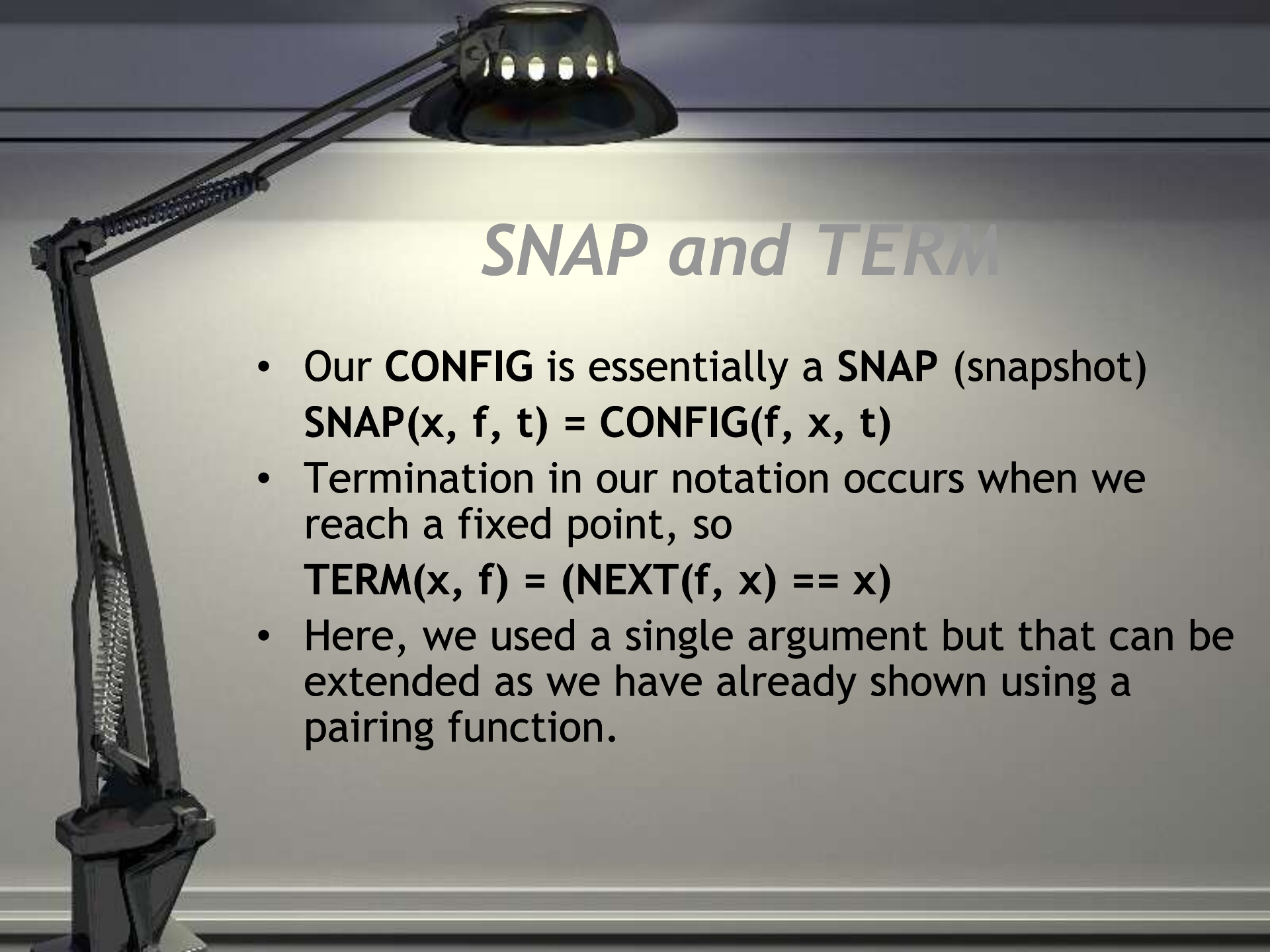
Universal Machine Notation

- $\Phi^{(n)}(x_1,\ldots,x_n, f) = \text{Univ}\,(f, \prod_{i=1}^{n} p_i^{x_i})$
- We will sometimes adopt the above and also its common shorthand

  $\Phi_f{}^{(n)}(x_1,\ldots,x_n) = \Phi^{(n)}(x_1,\ldots,x_n, f)$

  and the even shorter version

  $\Phi_f(x_1,\ldots,x_n) = \Phi^{(n)}(x_1,\ldots,x_n, f)$
- We even omit the (n) when n=1, as in $\Phi_f(x) = \Phi(x, f)$

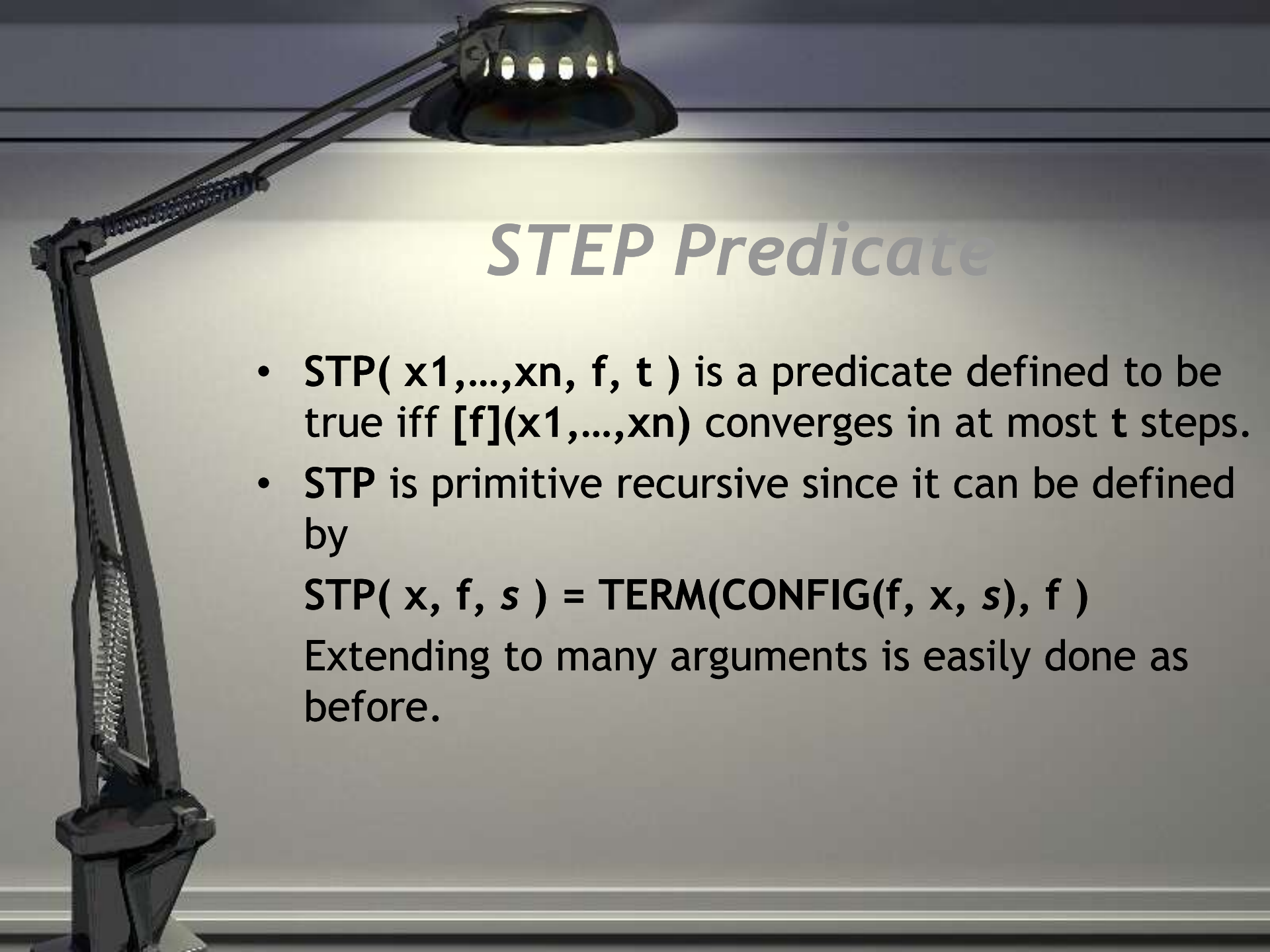


SNAP and TERM

- Our **CONFIG** is essentially a **SNAP** (snapshot)

  SNAP(x, f, t) = CONFIG(f, x, t)
- Termination in our notation occurs when we reach a fixed point, so

  TERM(x, f) = (NEXT(f, x) == x)
- Here, we used a single argument but that can be extended as we have already shown using a pairing function.

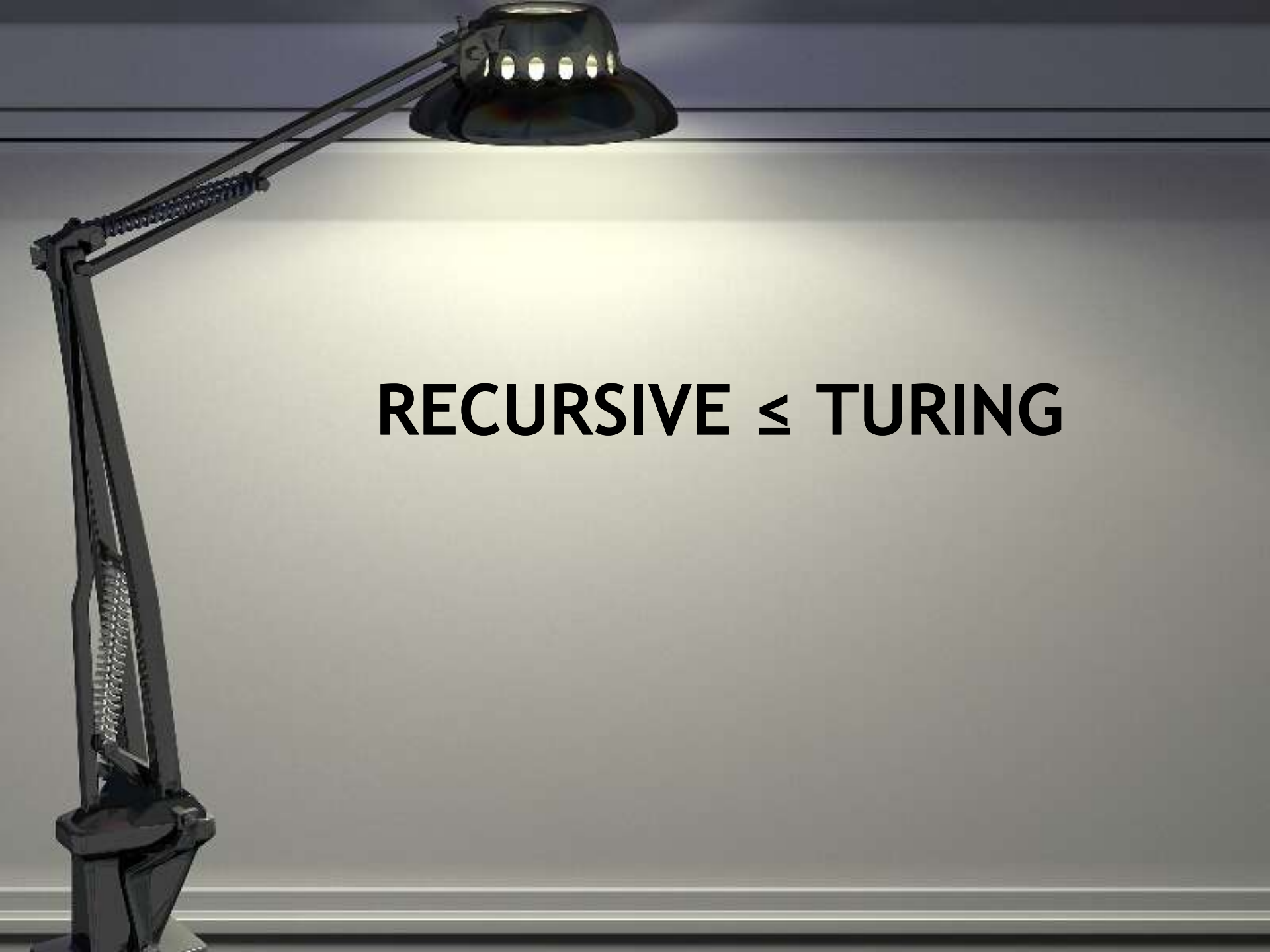


STEP Predicate

- **STP(x1,…,xn, f, t)** is a predicate defined to be true iff **[f](x1,…,xn)** converges in at most **t** steps.
- **STP** is primitive recursive since it can be defined by

  STP(x, f, *s*) = TERM(CONFIG(f, x, *s*), f)

  Extending to many arguments is easily done as before.


RECURSIVE ≤ TURING

Recall standard Turing

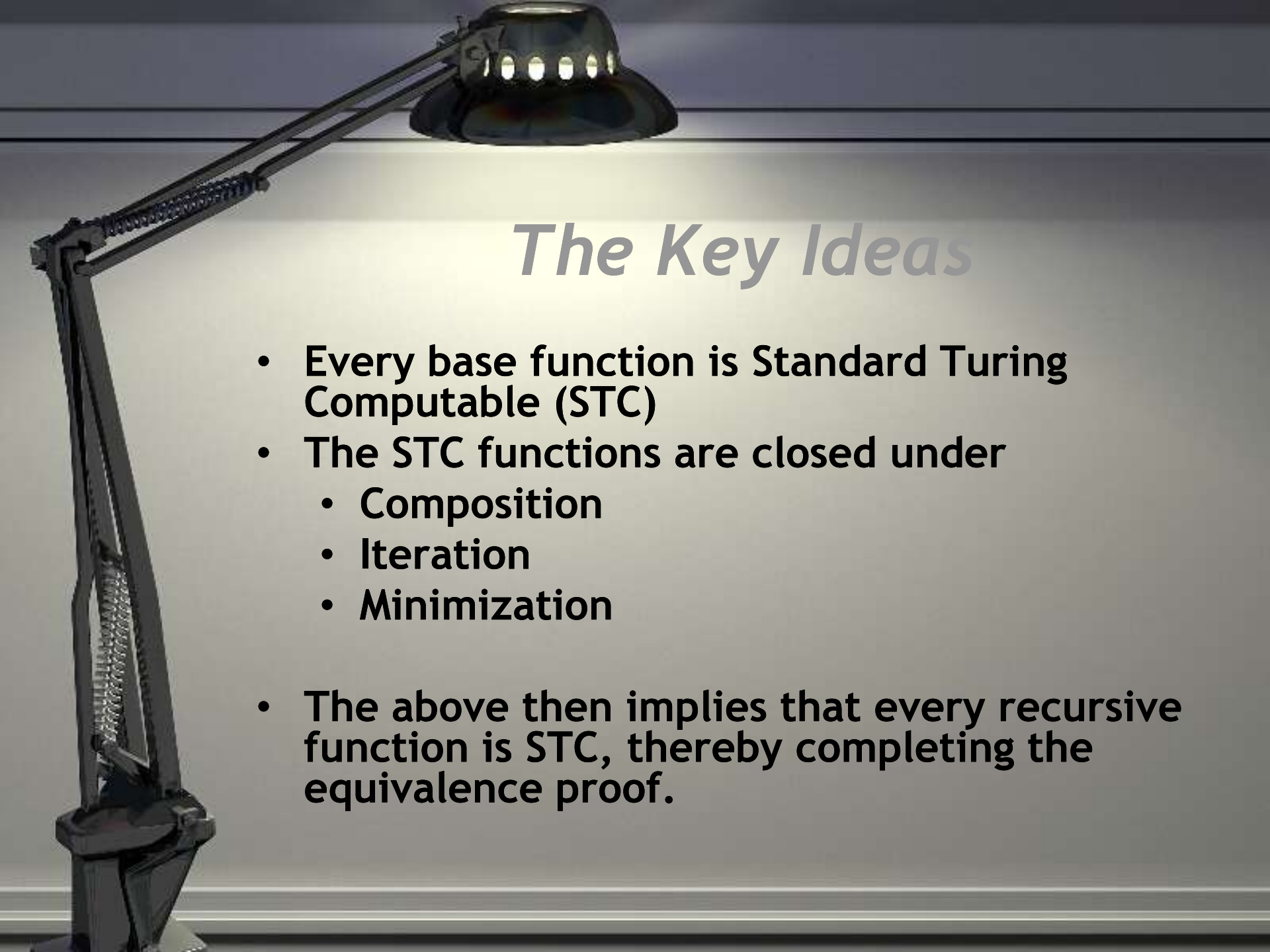
- Our notion of standard Turing computability of some n-ary function F assumes that the machine starts with a tape containing the n inputs, x1, … , xn in the form (underscore is scanned symbol)

  …$01^{x1}01^{x2}0…01^{xn}\underline{0}$…

  and ends with

  …$01^{x1}01^{x2}0…01^{xn}01^{y}\underline{0}$…

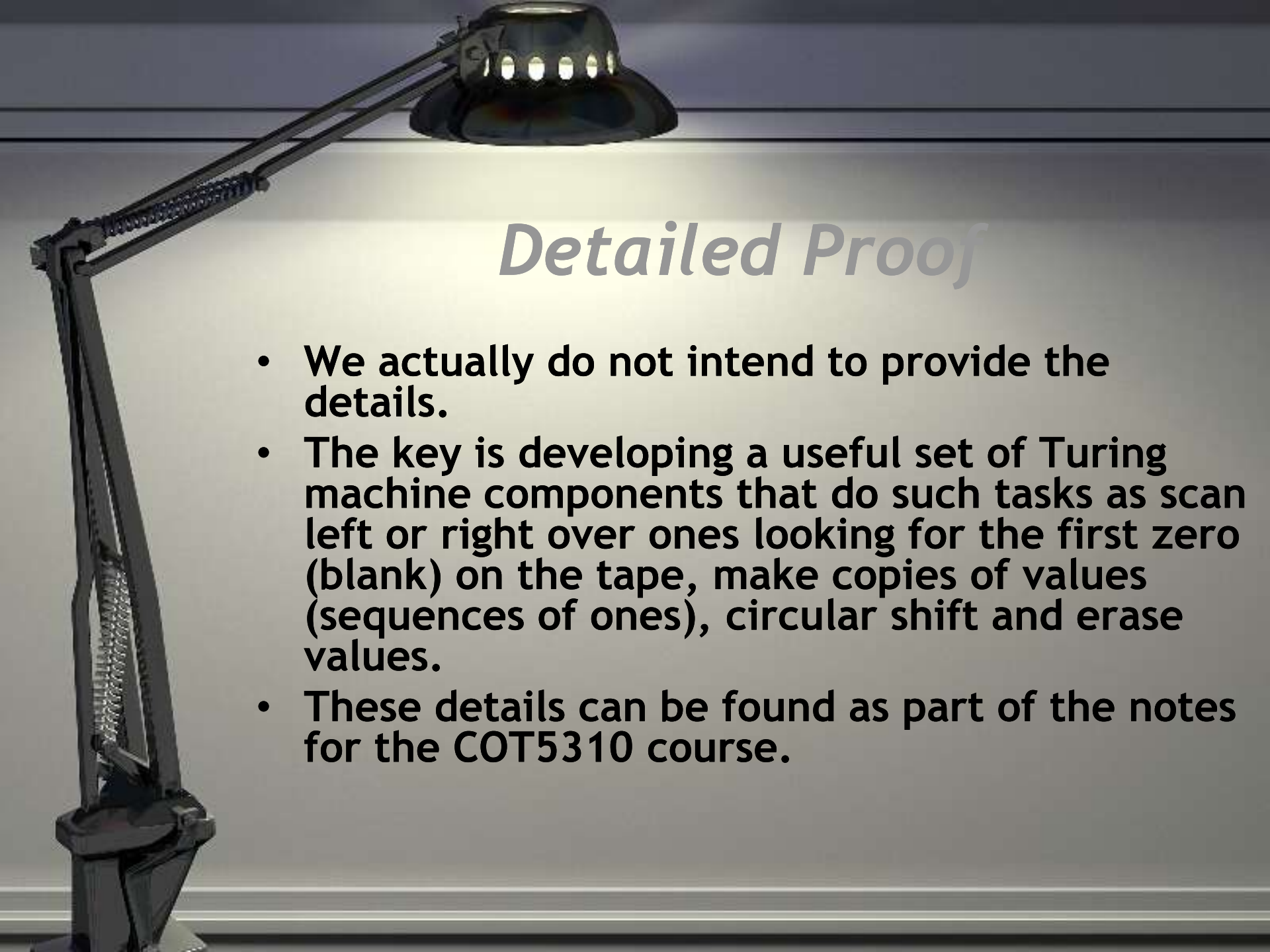
  where y = F(x1, … , xn).



The Key Ideas

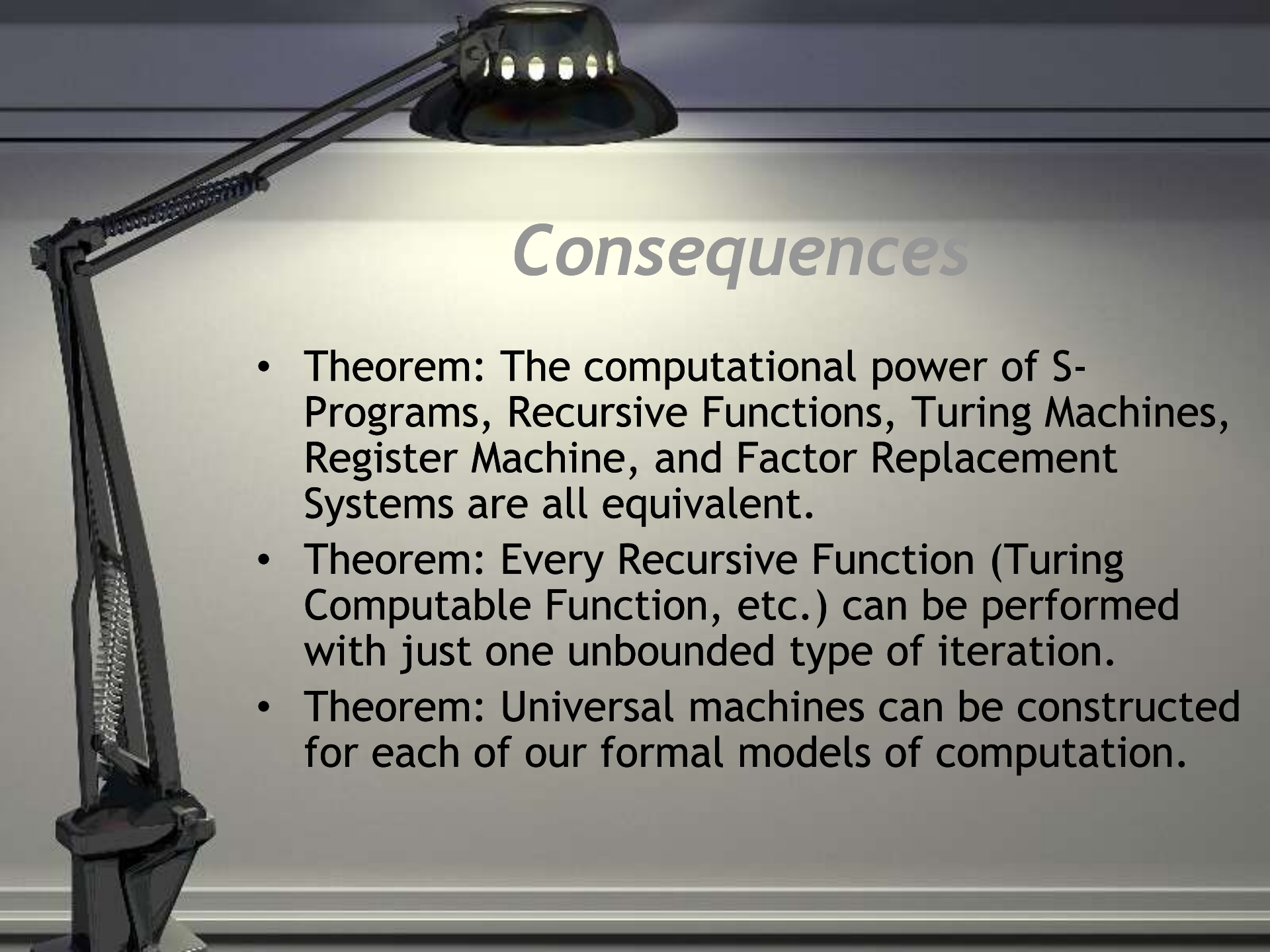
- Every base function is Standard Turing Computable (STC)
- The STC functions are closed under
  - Composition
  - Iteration
  - Minimization

- The above then implies that every recursive function is STC, thereby completing the equivalence proof.



Detailed Proof

- We actually do not intend to provide the details.
- The key is developing a useful set of Turing machine components that do such tasks as scan left or right over ones looking for the first zero (blank) on the tape, make copies of values (sequences of ones), circular shift and erase values.
- These details can be found as part of the notes for the COT5310 course.



Consequences

- Theorem: The computational power of S-Programs, Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.
- Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.
- Theorem: Universal machines can be constructed for each of our formal models of computation.


UNDECIDABILITY

Halting Problem (again)

Assume we can decide the **Halting Problem**. Then there exists some total function **Halt** such that

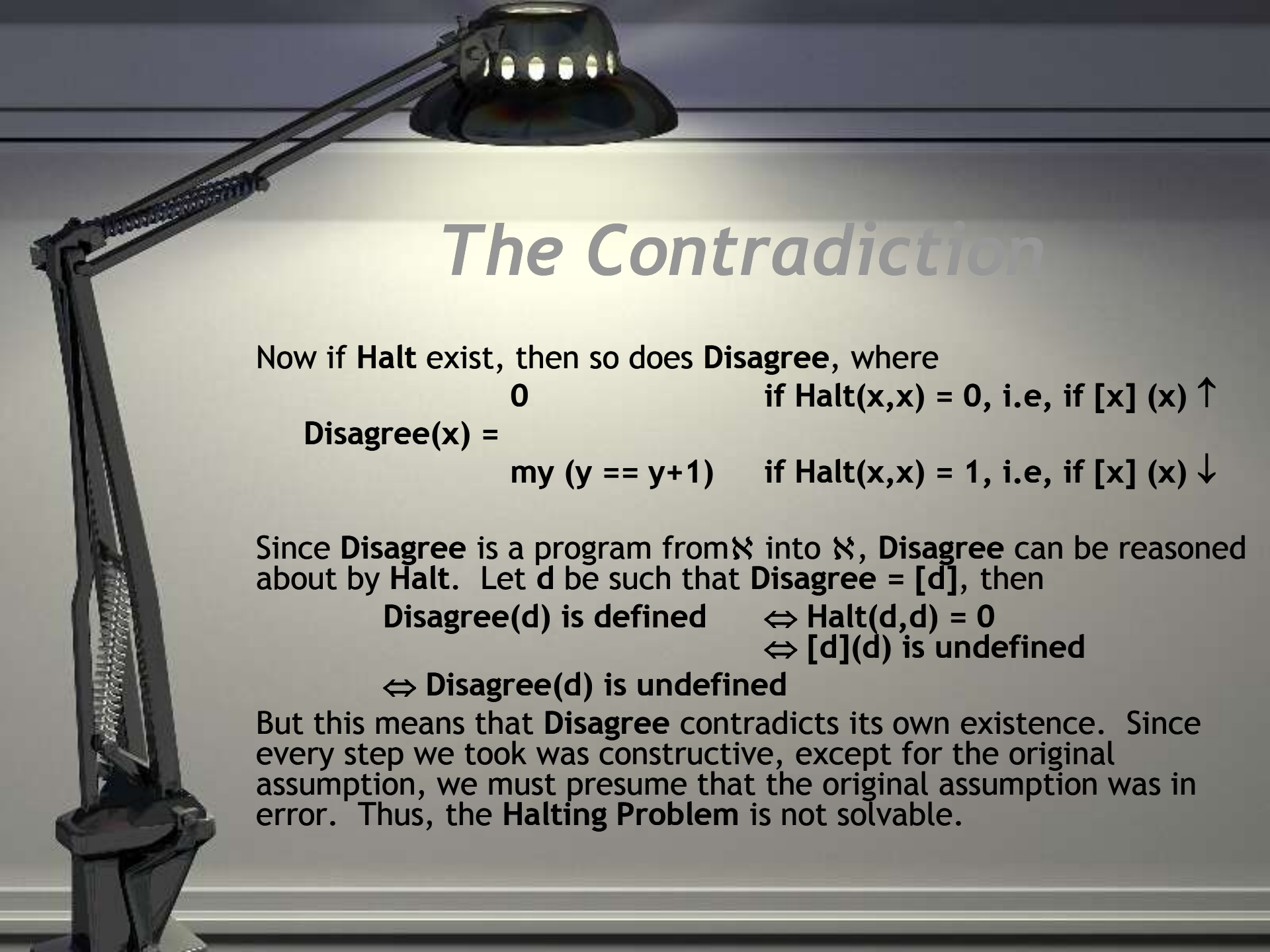
$$\mathbf{Halt}(x,y) = \begin{cases} 1 & \text{if } [x](y)\downarrow \\ 0 & \text{if } [x](y)\uparrow \end{cases}$$

Here, we have numbered all programs and **[x]** refers to the **x**-th program in this ordering. Now we can view **Halt** as a mapping from $\aleph$ into $\aleph$ by treating its input as a single number representing the pairing of two numbers via the one-one onto function

$$\mathbf{pair}(x,y) = \langle x,y \rangle = 2^x (2y + 1) - 1, \text{ with inverses}$$

$$x = \langle z \rangle_1 = \mathbf{exp}(z+1,1)$$

$$y = \langle z \rangle_2 = (((\, z + 1\,) \,//\, 2^{\langle z \rangle_1}\,) - 1\,) \,//\, 2$$



The Contradiction

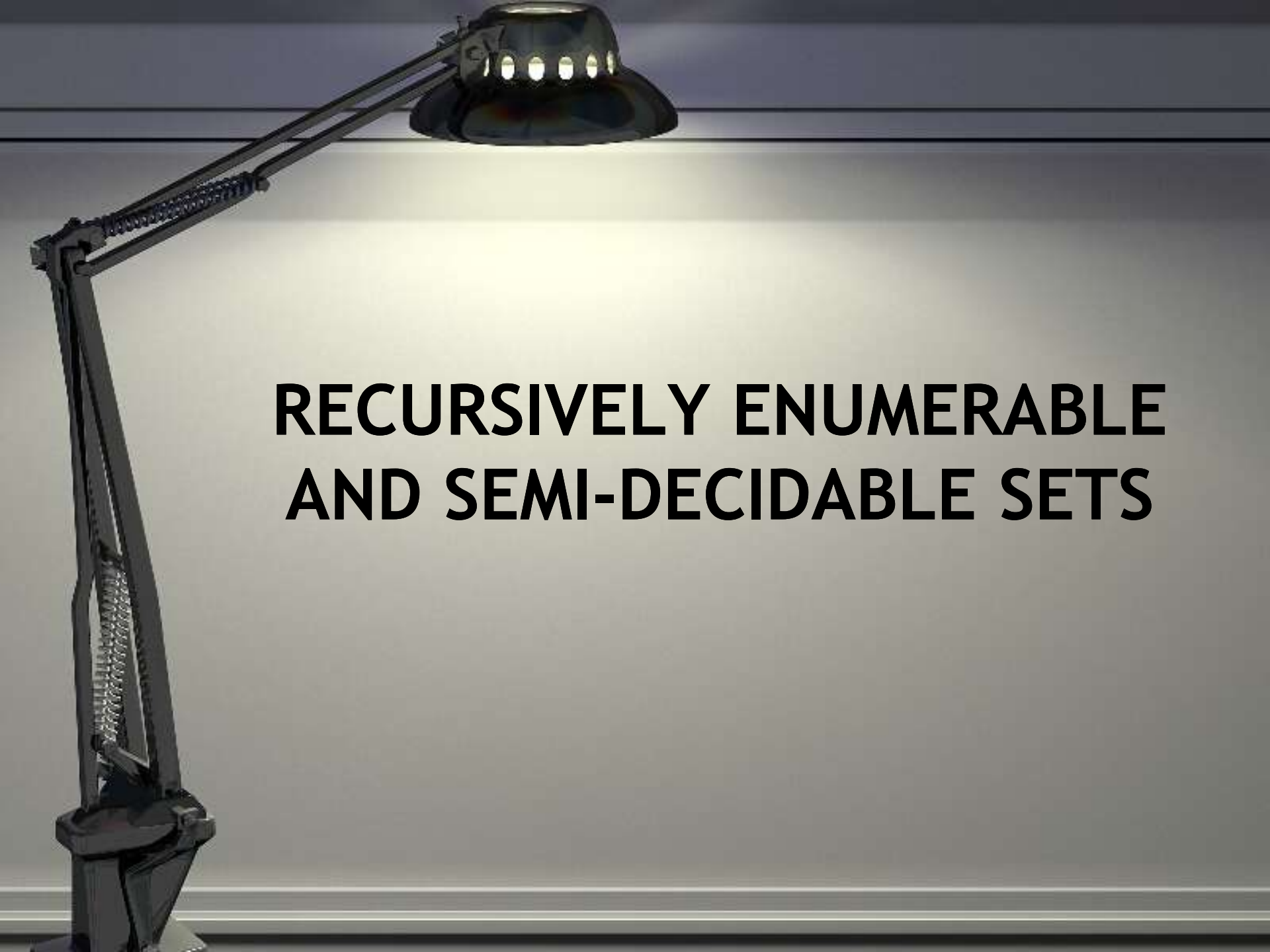
Now if **Halt** exist, then so does **Disagree**, where

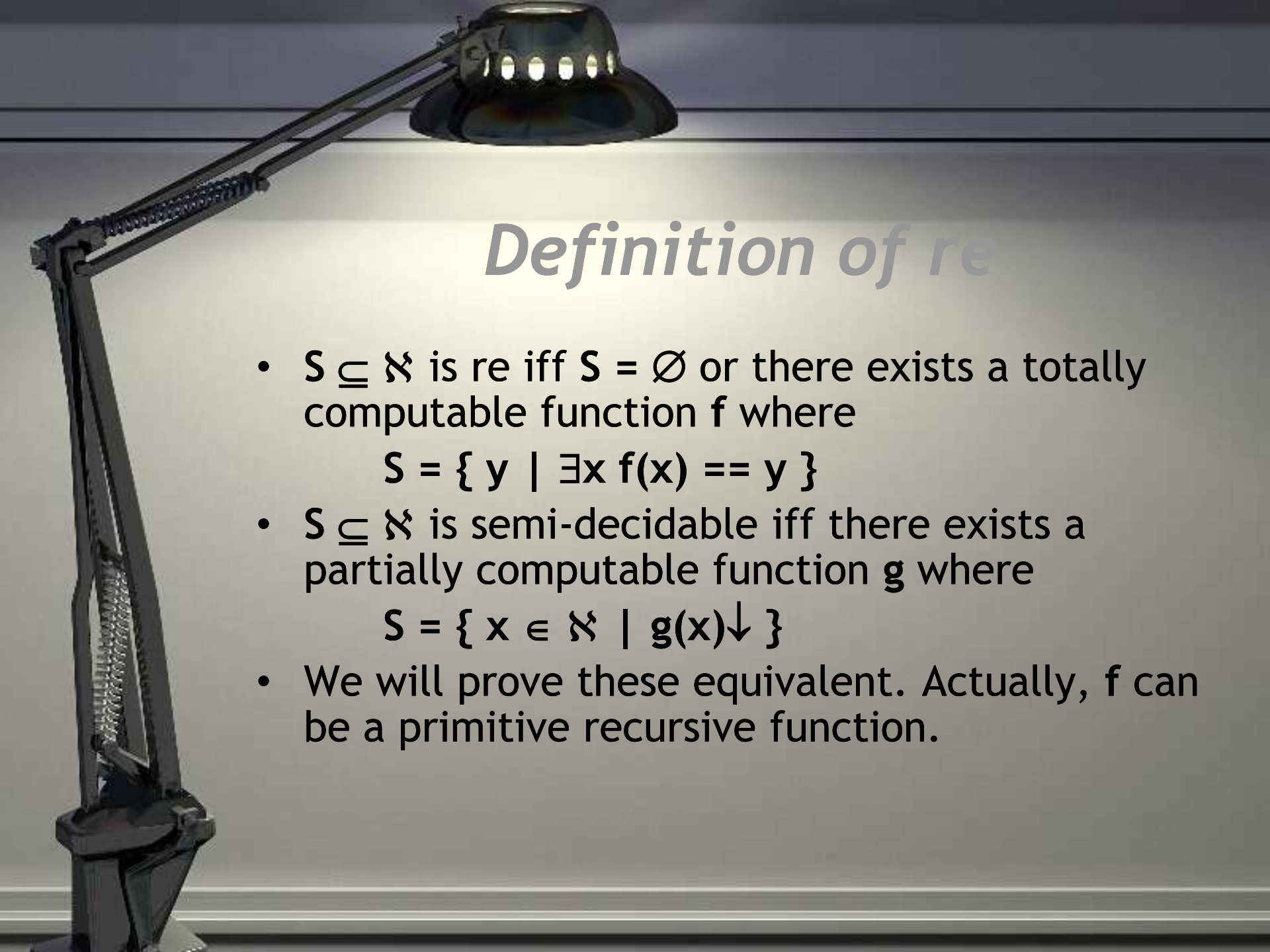
$$\textbf{Disagree}(x) = \begin{cases} 0 & \text{if } \textbf{Halt}(x,x) = 0, \text{ i.e, if } [x](x)\uparrow \\ \textbf{my } (y == y+1) & \text{if } \textbf{Halt}(x,x) = 1, \text{ i.e, if } [x](x)\downarrow \end{cases}$$

Since **Disagree** is a program from $\aleph$ into $\aleph$, **Disagree** can be reasoned about by **Halt**. Let **d** be such that **Disagree** = **[d]**, then

Disagree(d) is defined $\Leftrightarrow$ **Halt(d,d) = 0**
$\Leftrightarrow$ **[d](d) is undefined**
$\Leftrightarrow$ **Disagree(d) is undefined**

But this means that **Disagree** contradicts its own existence. Since every step we took was constructive, except for the original assumption, we must presume that the original assumption was in error. Thus, the **Halting Problem** is not solvable.


RECURSIVELY ENUMERABLE
AND SEMI-DECIDABLE SETS



Definition of re

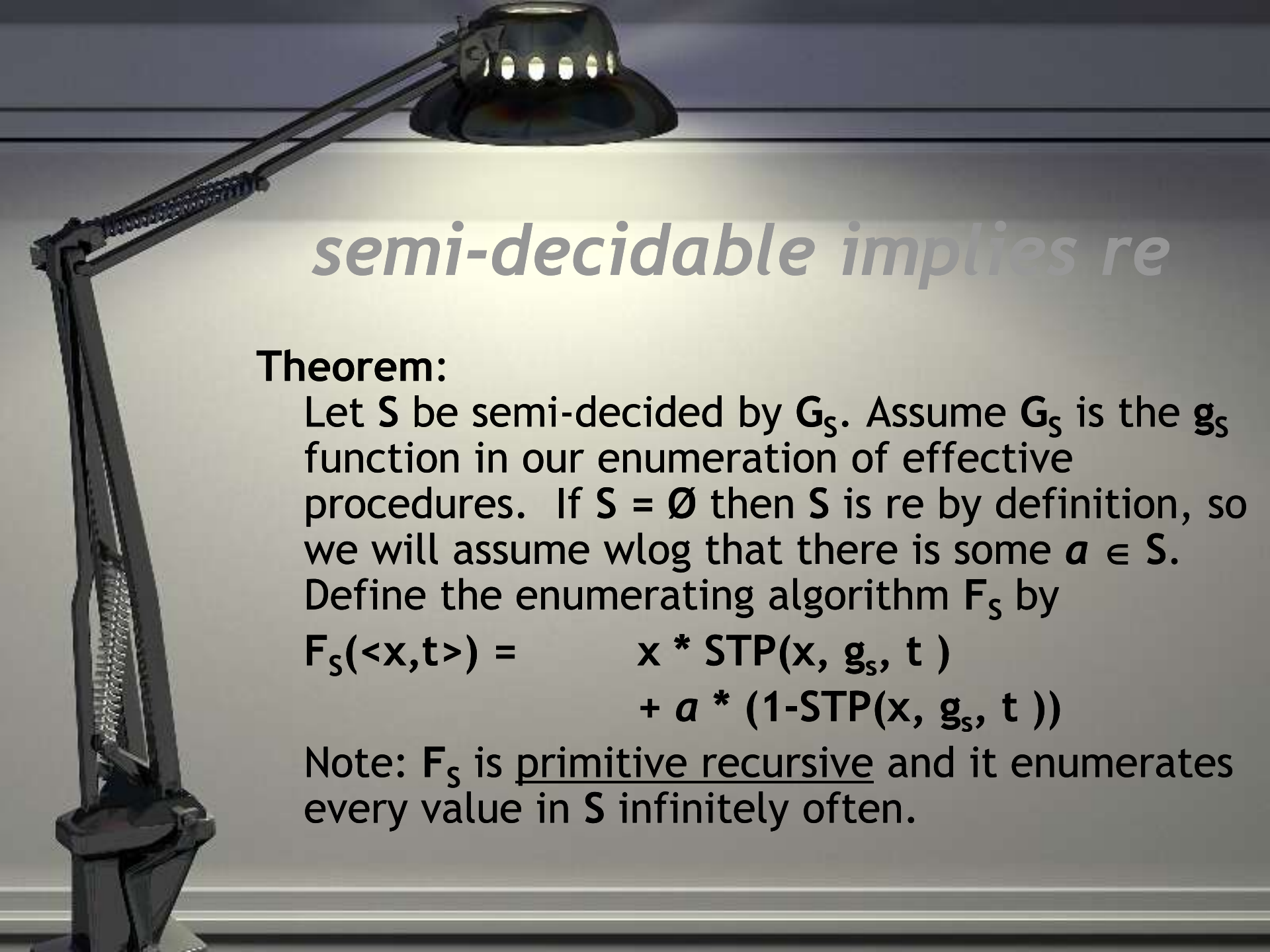
- $S \subseteq \aleph$ is re iff $S = \varnothing$ or there exists a totally computable function **f** where

  $$S = \{ y \mid \exists x\ f(x) == y \}$$

- $S \subseteq \aleph$ is semi-decidable iff there exists a partially computable function **g** where

  $$S = \{ x \in \aleph \mid g(x)\downarrow \}$$

- We will prove these equivalent. Actually, **f** can be a primitive recursive function.



semi-decidable implies re

Theorem:

Let **S** be semi-decided by G_S. Assume G_S is the g_S function in our enumeration of effective procedures. If **S** = **Ø** then **S** is re by definition, so we will assume wlog that there is some $a \in S$. Define the enumerating algorithm F_S by

$$F_S(\langle x,t \rangle) = x * STP(x, g_s, t) + a * (1-STP(x, g_s, t))$$

Note: F_S is primitive recursive and it enumerates every value in **S** infinitely often.

re implies semi-decidable

Theorem:

By definition, **S** is re iff **S == Ø** or there exists an algorithm F_S, over the natural numbers $\aleph$, whose range is exactly **S**. Define

$$\psi_S(x) = \begin{cases} \mu y\,[y == y+1] & \text{if } S == \emptyset \\ \text{signum}((\mu y[F_S(y)==x])+1), & \text{otherwise} \end{cases}$$

This achieves our result as the domain of ψ_S is the range of F_S, or empty if **S == Ø**.

Domain of a procedure

Corollary: **S** is re/semi-decidable iff **S** is the domain / range of a partial recursive predicate $\mathbf{F_S}$.

Proof: The predicate ψ_S we defined earlier to semi-decide **S**, given its enumerating function, can be easily adapted to have this property.

$$\psi_S(x) = \begin{cases} \mu y\,[y == y+1] & \text{if } S == \emptyset \\ x * \text{signum}((\mu y[F_S(y)==x])+1) & \text{, otherwise} \end{cases}$$

Recursive implies re

Theorem: Recursive implies re.

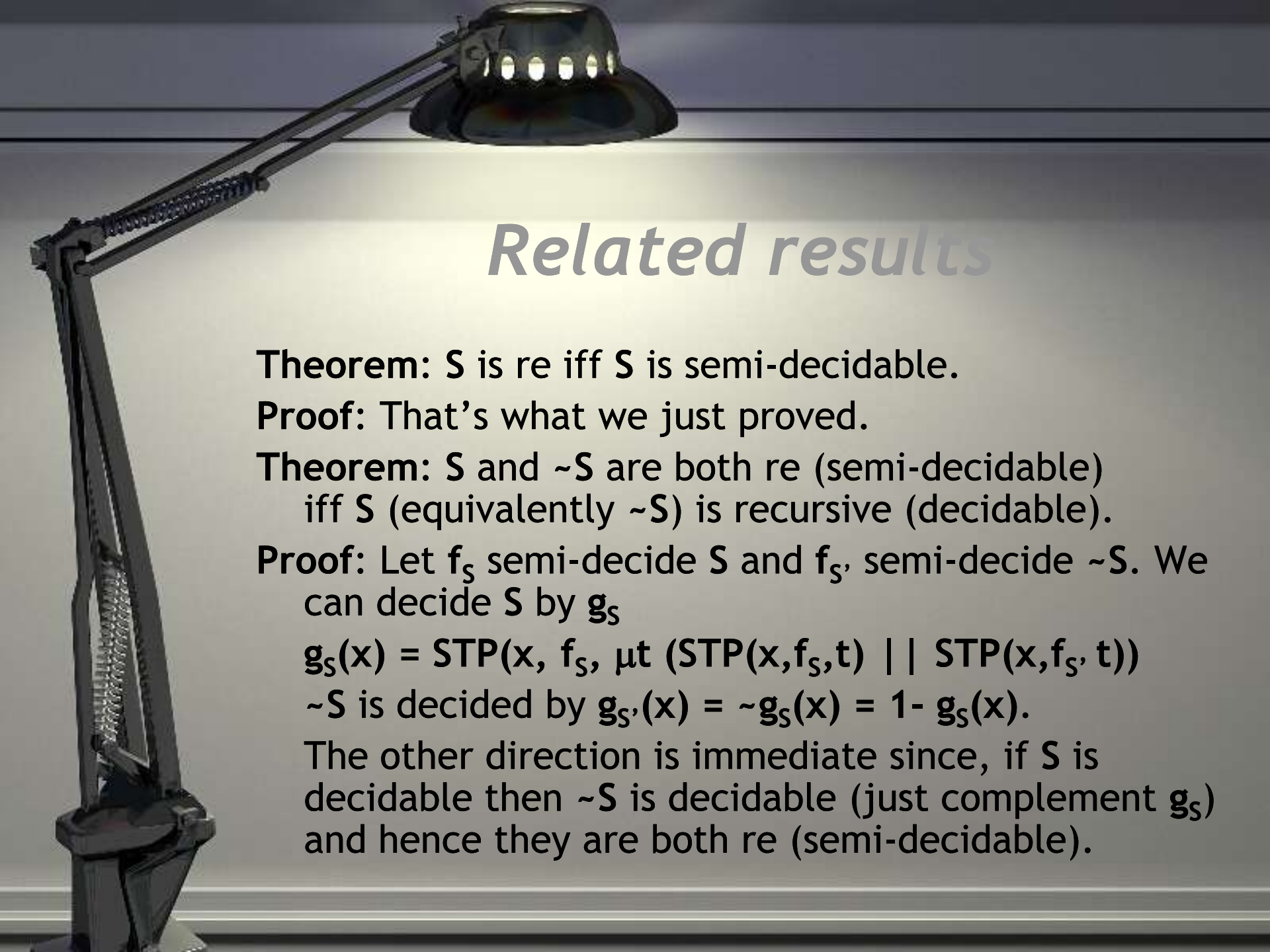
Proof: S is recursive implies there is a total recursive function f_S such that

$$S = \{ x \in \aleph \mid f_s(x) == 1 \}$$

Define $g_s(x) = \mu y\ (f_s(x) == 1)$

Clearly

$$\begin{aligned} dom(g_s) &= \{x \in \aleph \mid g_s(x)\downarrow\} \\ &= \{ x \in \aleph \mid f_s(x) == 1 \} \\ &= S \end{aligned}$$



Related results

Theorem: **S** is re iff **S** is semi-decidable.

Proof: That's what we just proved.

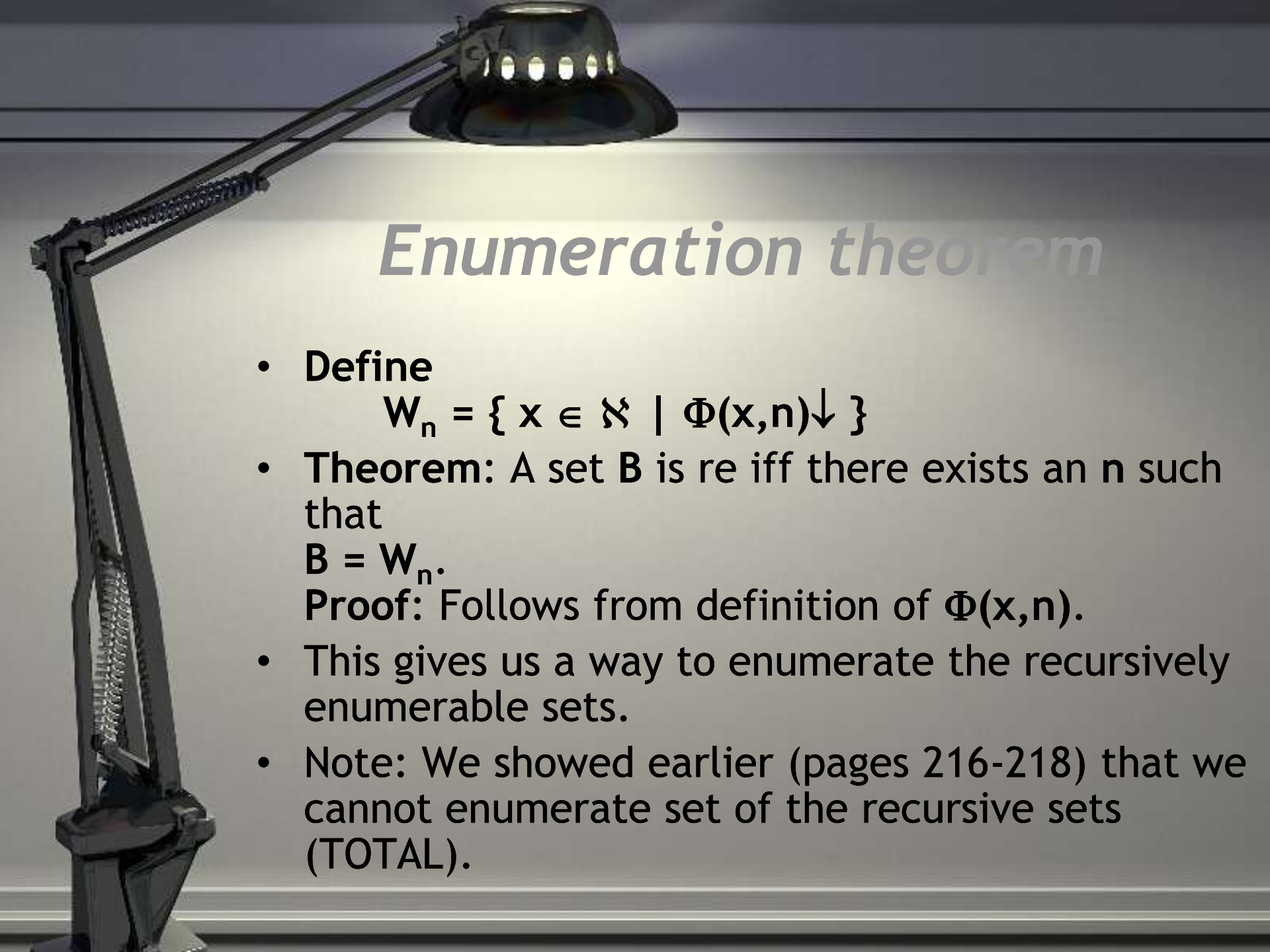
Theorem: **S** and **~S** are both re (semi-decidable) iff **S** (equivalently **~S**) is recursive (decidable).

Proof: Let f_S semi-decide **S** and $f_{S'}$ semi-decide **~S**. We can decide **S** by g_S

$g_S(x) = STP(x, f_S, \mu t\ (STP(x,f_S,t)\ ||\ STP(x,f_{S'}, t))$

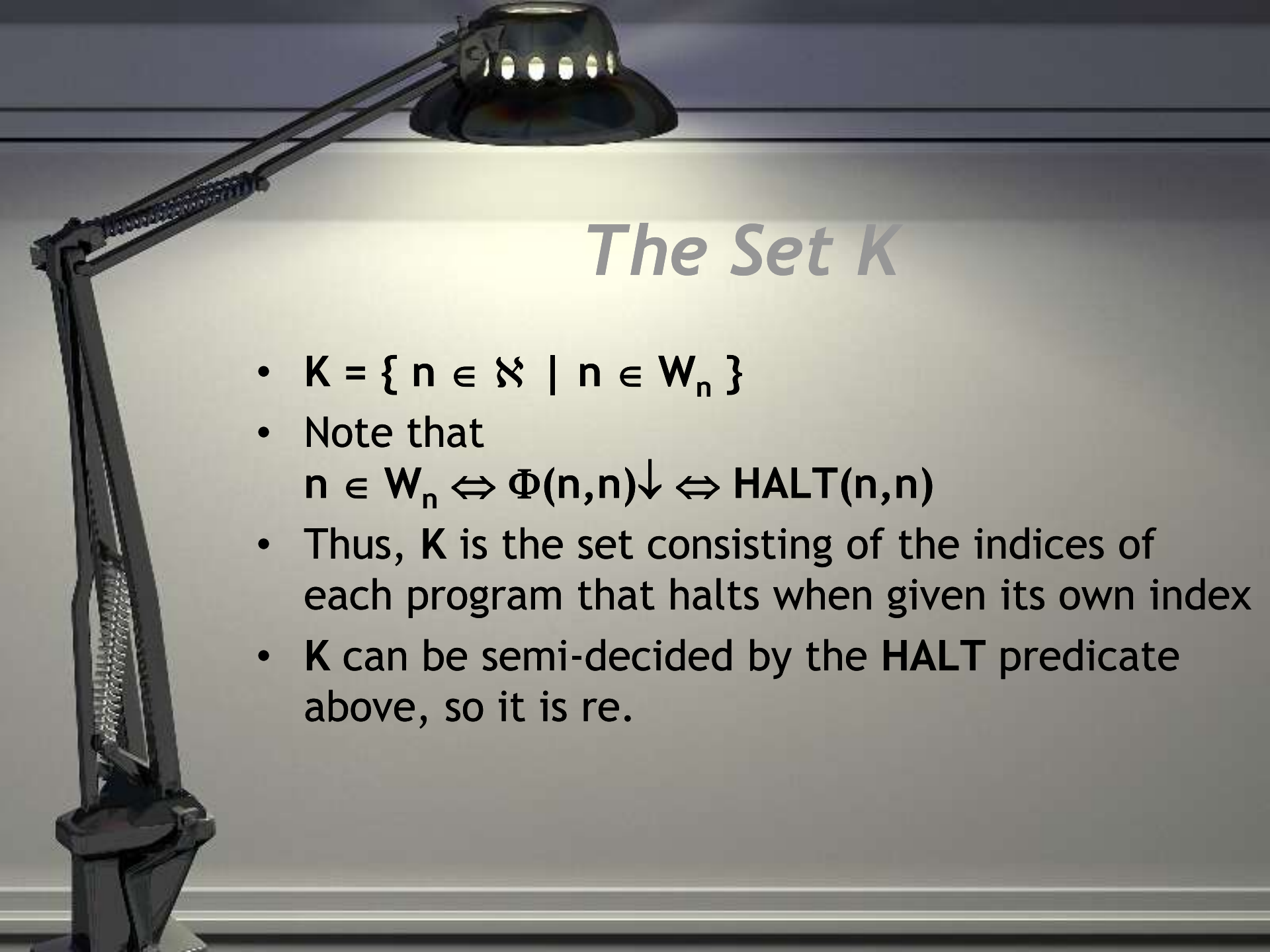
~S is decided by $g_{S'}(x) = \sim g_S(x) = 1 - g_S(x)$.

The other direction is immediate since, if **S** is decidable then **~S** is decidable (just complement g_S) and hence they are both re (semi-decidable).



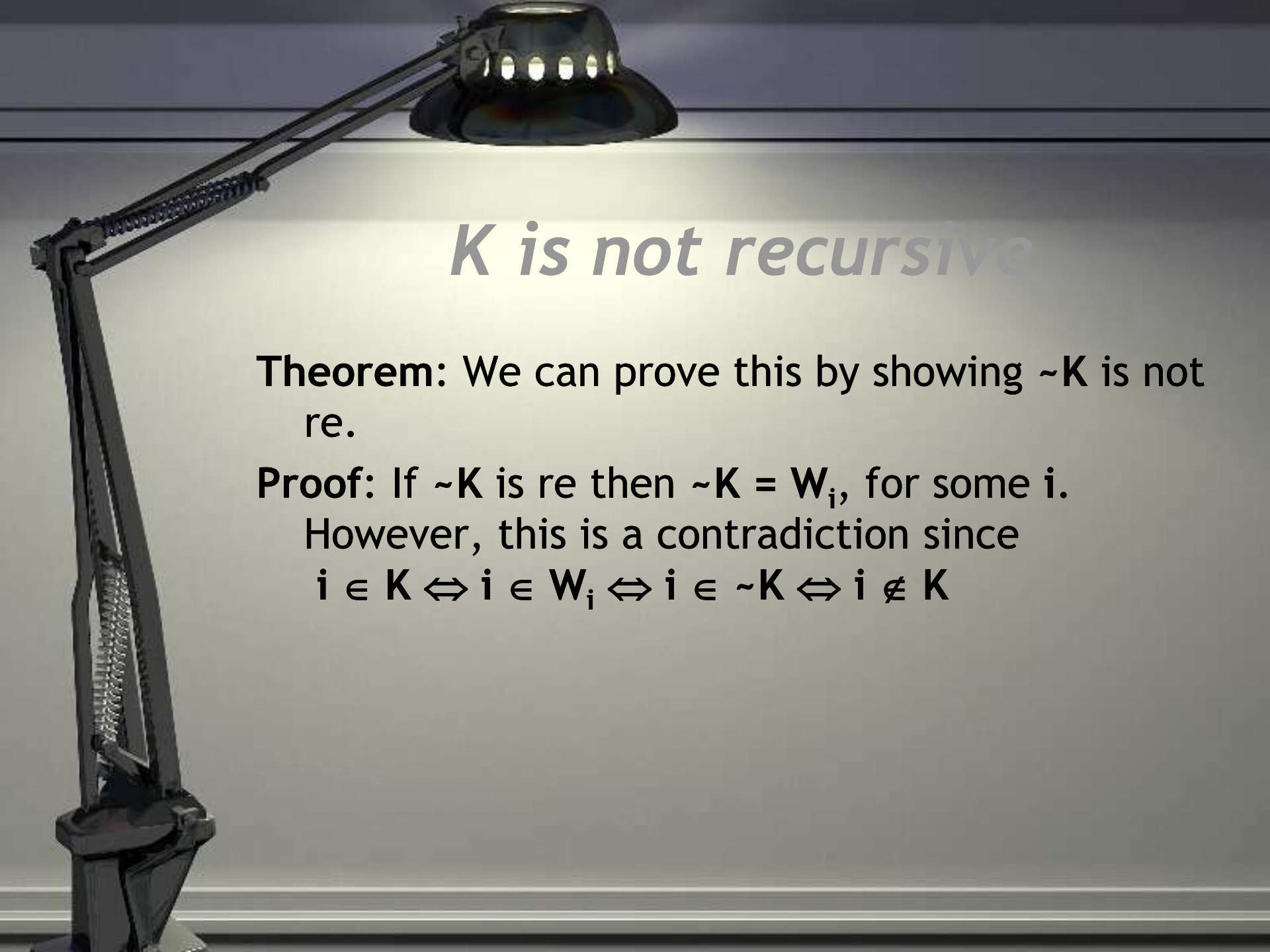
Enumeration theorem

- **Define**
  $W_n = \{ x \in \aleph \mid \Phi(x,n)\downarrow \}$
- **Theorem**: A set **B** is re iff there exists an **n** such that
  $B = W_n$.
  Proof: Follows from definition of $\Phi(x,n)$.
- This gives us a way to enumerate the recursively enumerable sets.
- Note: We showed earlier (pages 216-218) that we cannot enumerate set of the recursive sets (TOTAL).



The Set K

- $K = \{ n \in \aleph \mid n \in W_n \}$
- Note that
  $n \in W_n \Leftrightarrow \Phi(n,n)\downarrow \Leftrightarrow \mathbf{HALT}(n,n)$
- Thus, **K** is the set consisting of the indices of each program that halts when given its own index
- **K** can be semi-decided by the **HALT** predicate above, so it is re.

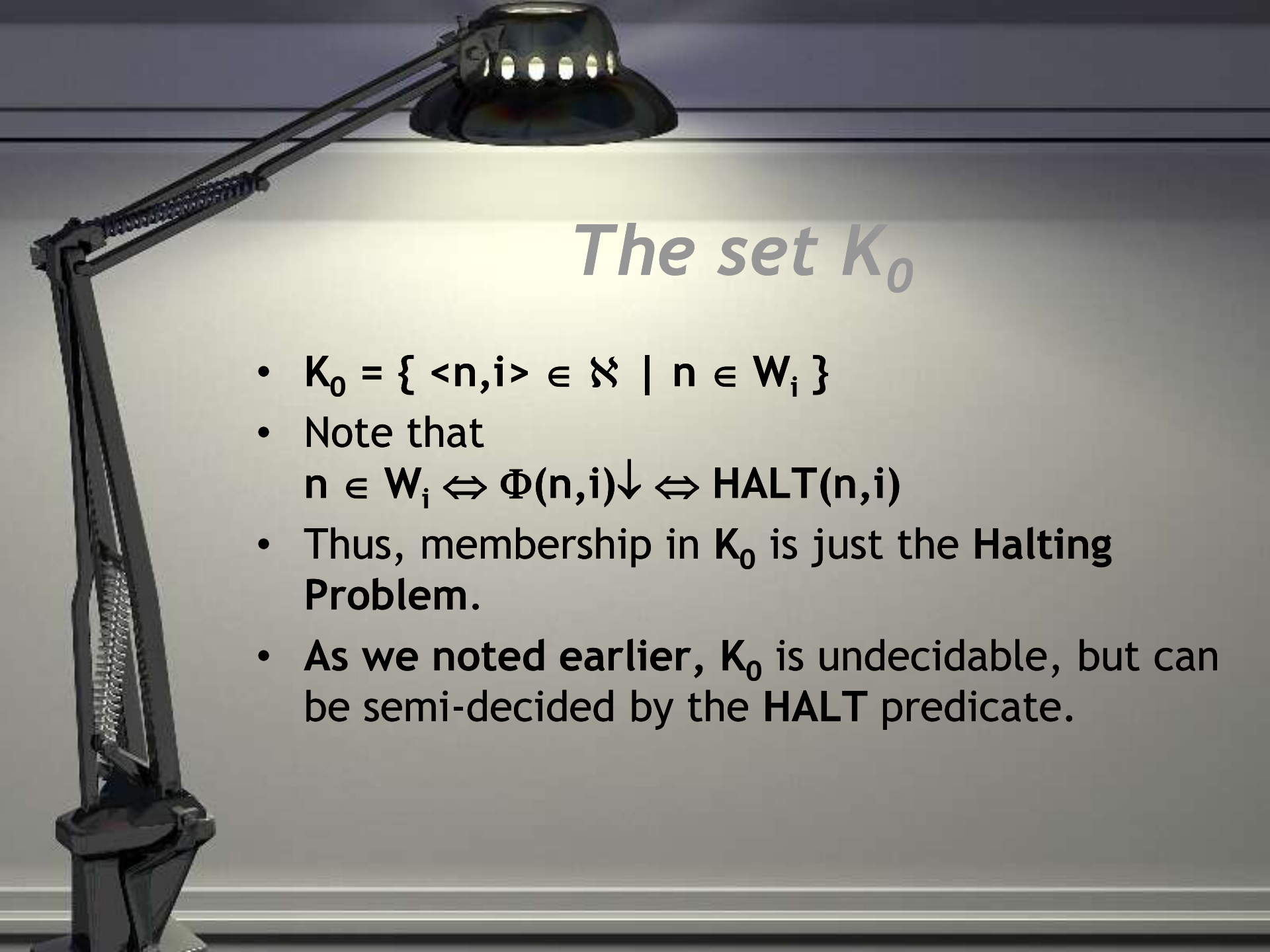


K is not recursive

Theorem: We can prove this by showing **~K** is not re.

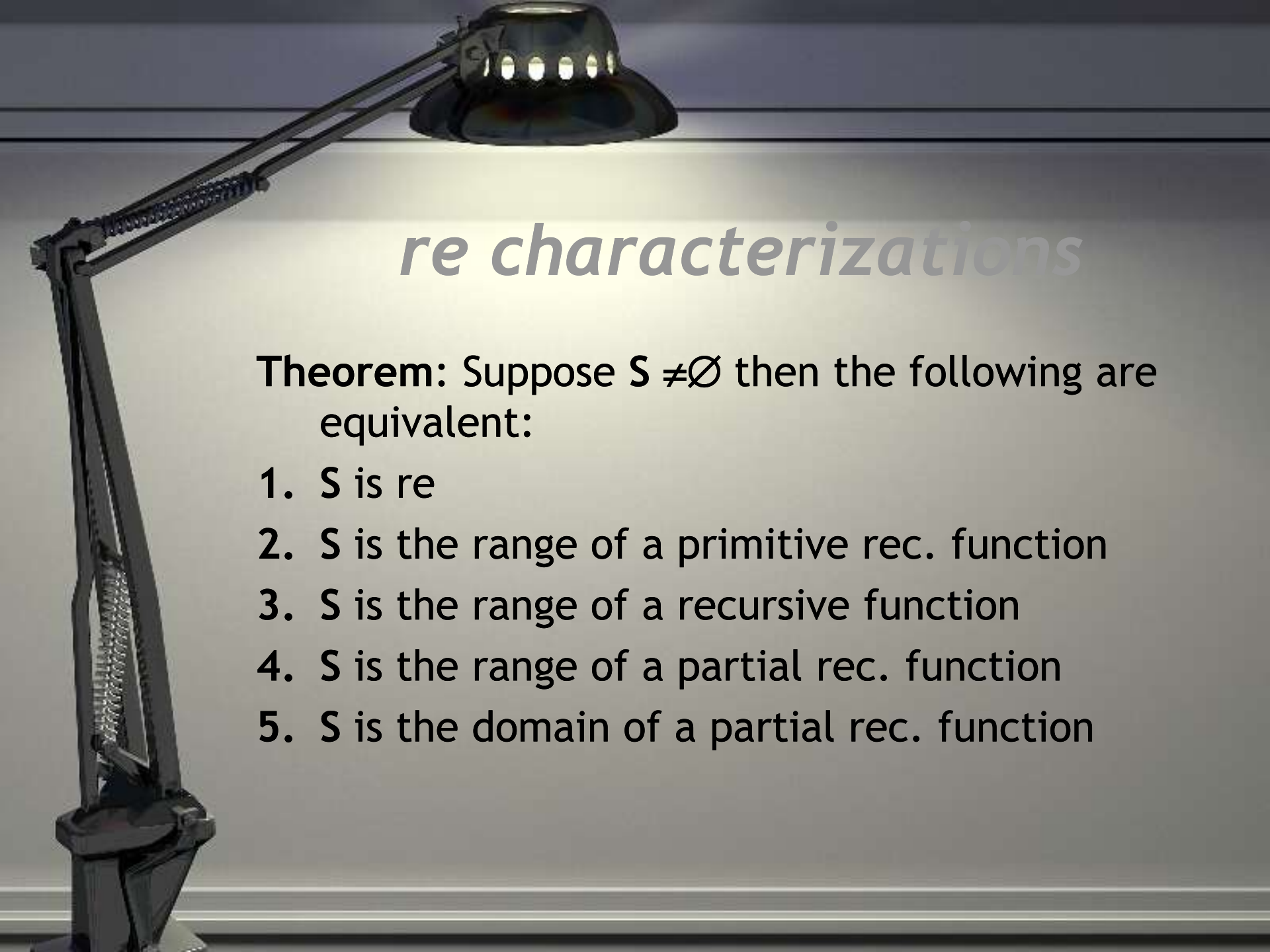
Proof: If **~K** is re then **~K = W_i**, for some **i**. However, this is a contradiction since

$$i \in K \Leftrightarrow i \in W_i \Leftrightarrow i \in \sim K \Leftrightarrow i \notin K$$



The set K_0

- $K_0 = \{ \langle n,i \rangle \in \aleph \mid n \in W_i \}$
- Note that
  $n \in W_i \Leftrightarrow \Phi(n,i)\downarrow \Leftrightarrow \mathbf{HALT}(n,i)$
- Thus, membership in K_0 is just the **Halting Problem**.
- **As we noted earlier,** K_0 is undecidable, but can be semi-decided by the **HALT** predicate.

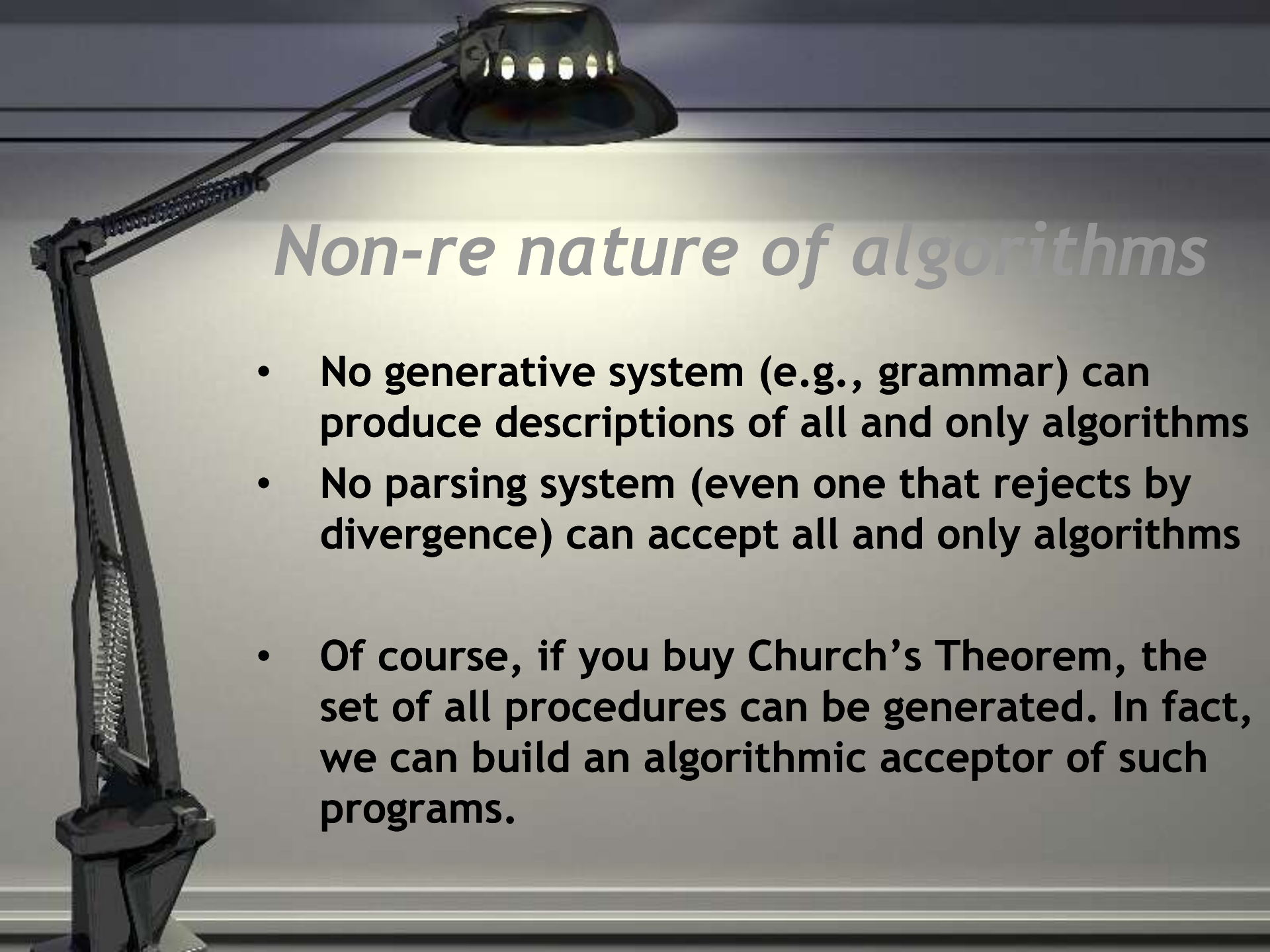


re characterizations

Theorem: Suppose **S** ≠∅ then the following are equivalent:

1. **S** is re
2. **S** is the range of a primitive rec. function
3. **S** is the range of a recursive function
4. **S** is the range of a partial rec. function
5. **S** is the domain of a partial rec. function

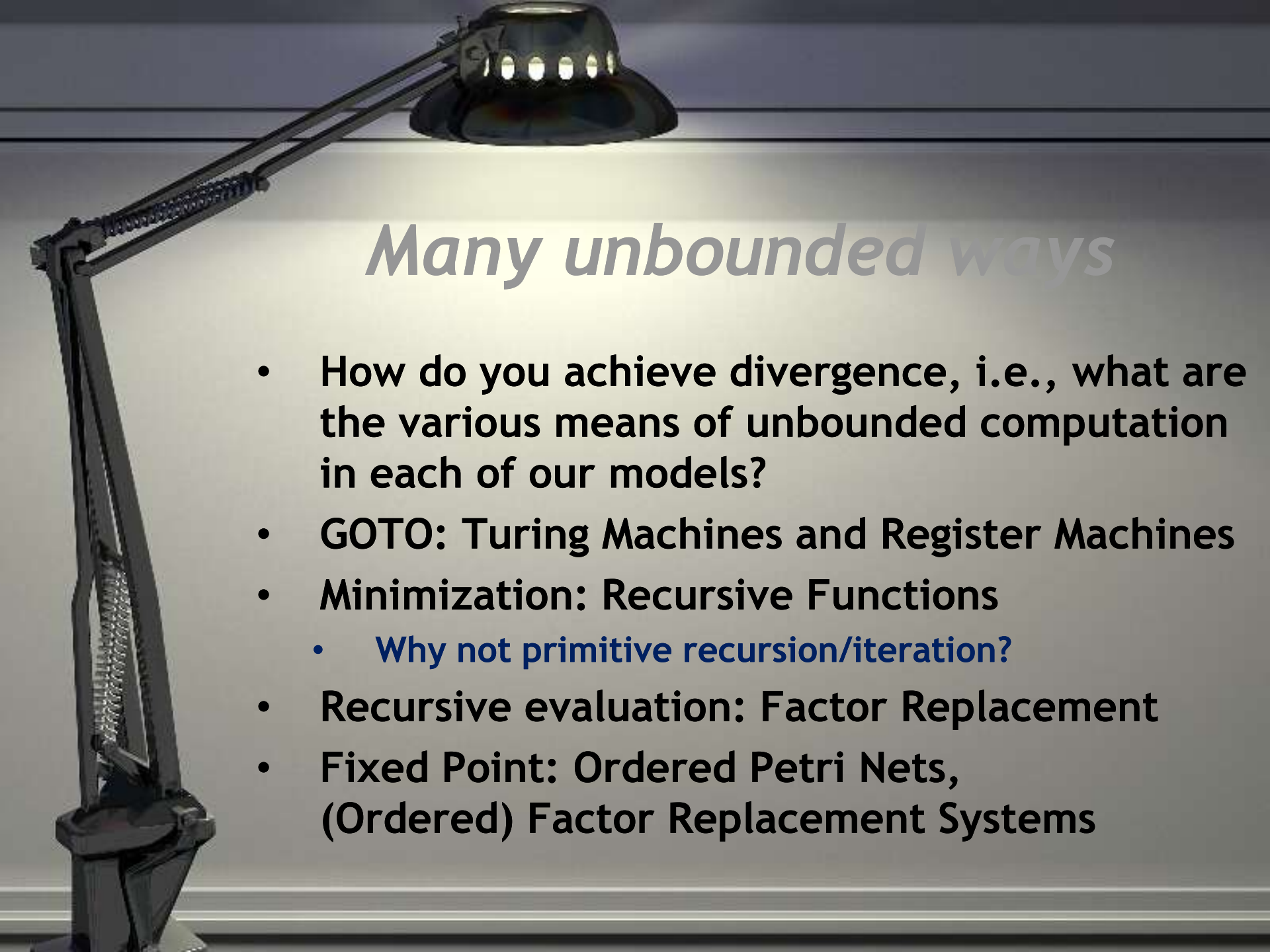

INSIGHTS



Non-re nature of algorithms

- No generative system (e.g., grammar) can produce descriptions of all and only algorithms
- No parsing system (even one that rejects by divergence) can accept all and only algorithms

- Of course, if you buy Church's Theorem, the set of all procedures can be generated. In fact, we can build an algorithmic acceptor of such programs.

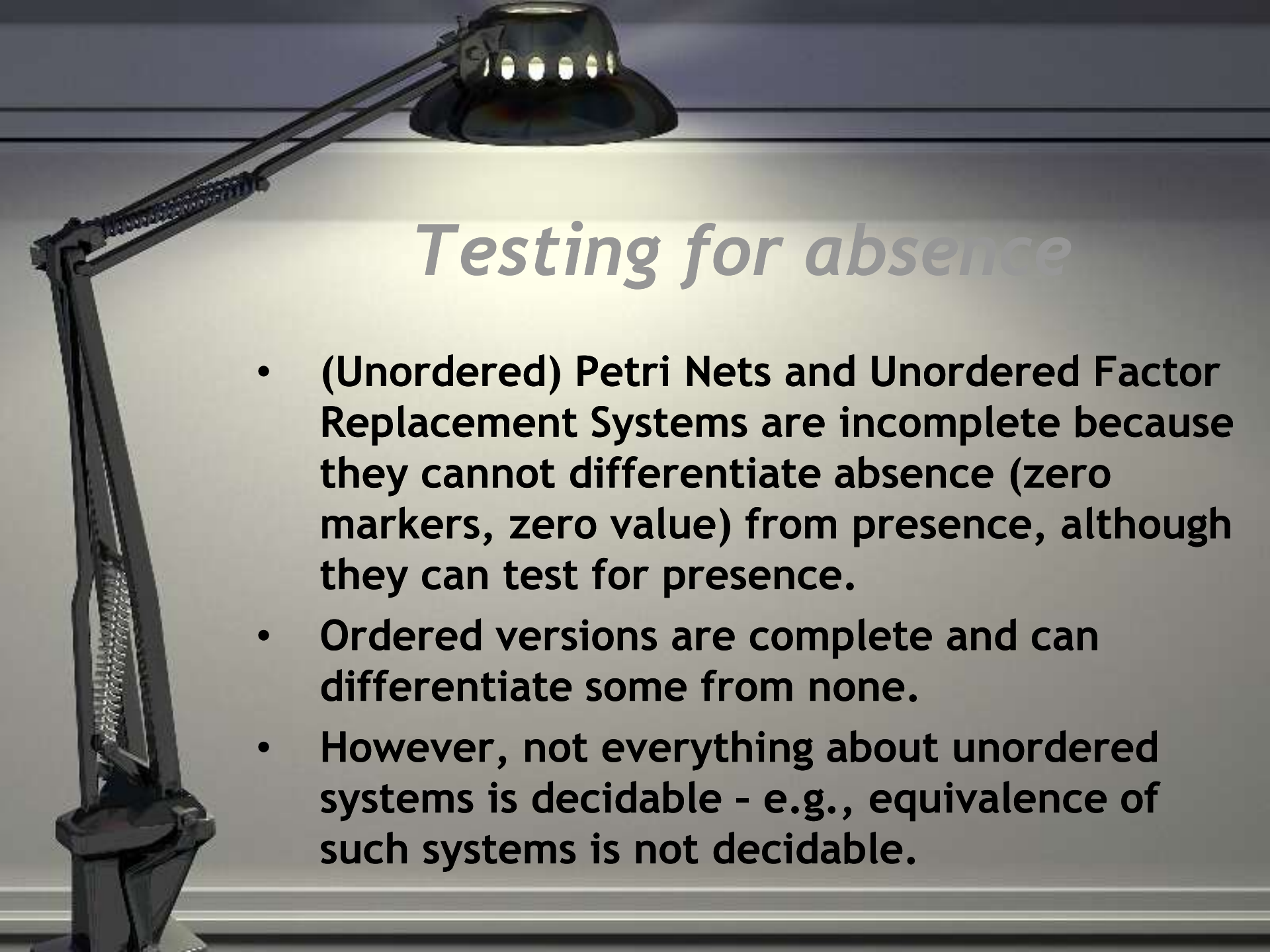


Many unbounded ways

- How do you achieve divergence, i.e., what are the various means of unbounded computation in each of our models?
- GOTO: Turing Machines and Register Machines
- Minimization: Recursive Functions
  - Why not primitive recursion/iteration?
- Recursive evaluation: Factor Replacement
- Fixed Point: Ordered Petri Nets, (Ordered) Factor Replacement Systems

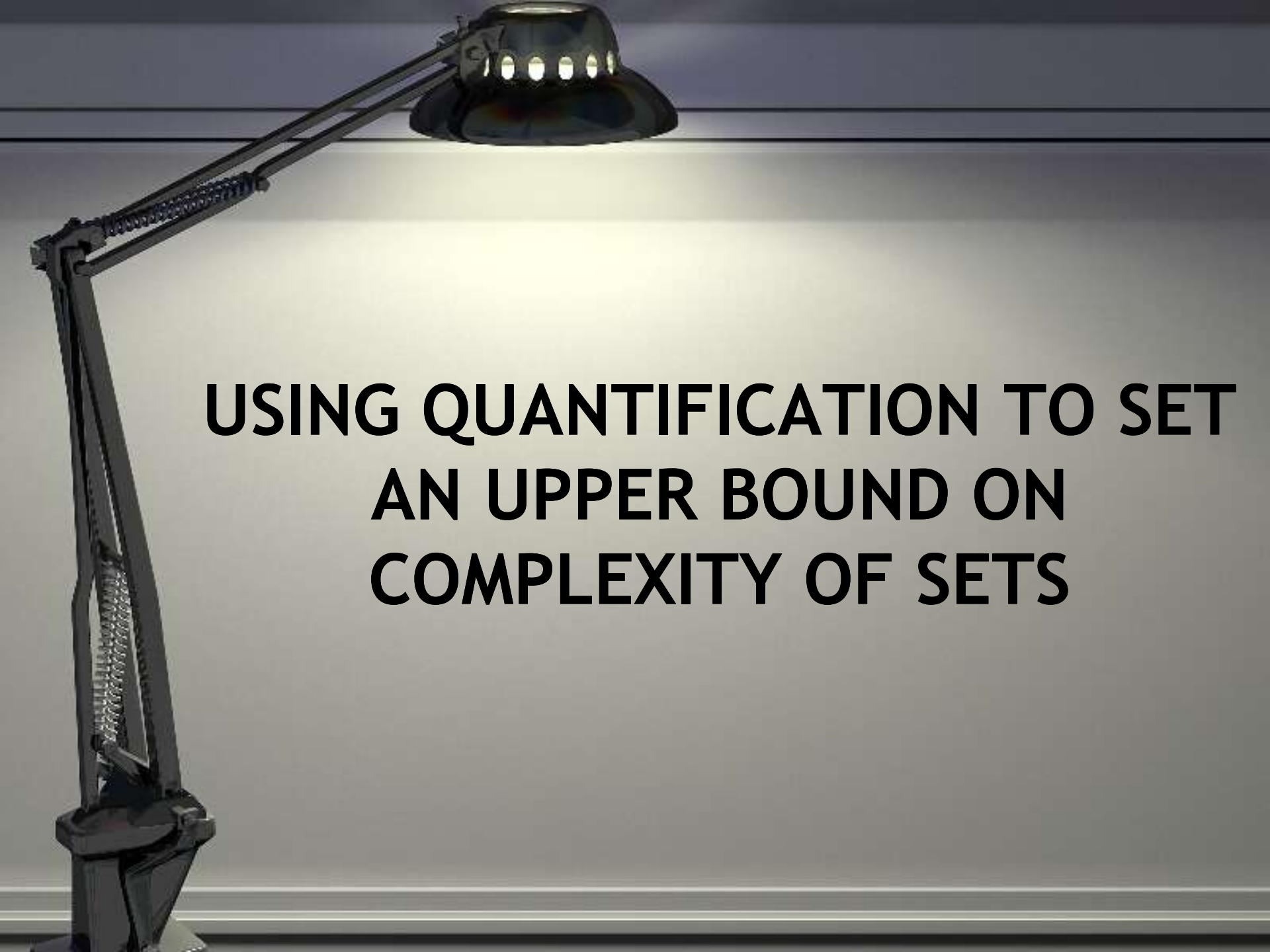
Non-determinism

- It sometimes doesn't matter
  - Turing Machines, Finite State Automata, Linear Bounded Automata
- It sometimes helps
  - Push Down Automata
- It sometimes hinders
  - Factor Replacement Systems, Petri Nets



Testing for absence

- (Unordered) Petri Nets and Unordered Factor Replacement Systems are incomplete because they cannot differentiate absence (zero markers, zero value) from presence, although they can test for presence.
- Ordered versions are complete and can differentiate some from none.
- However, not everything about unordered systems is decidable – e.g., equivalence of such systems is not decidable.

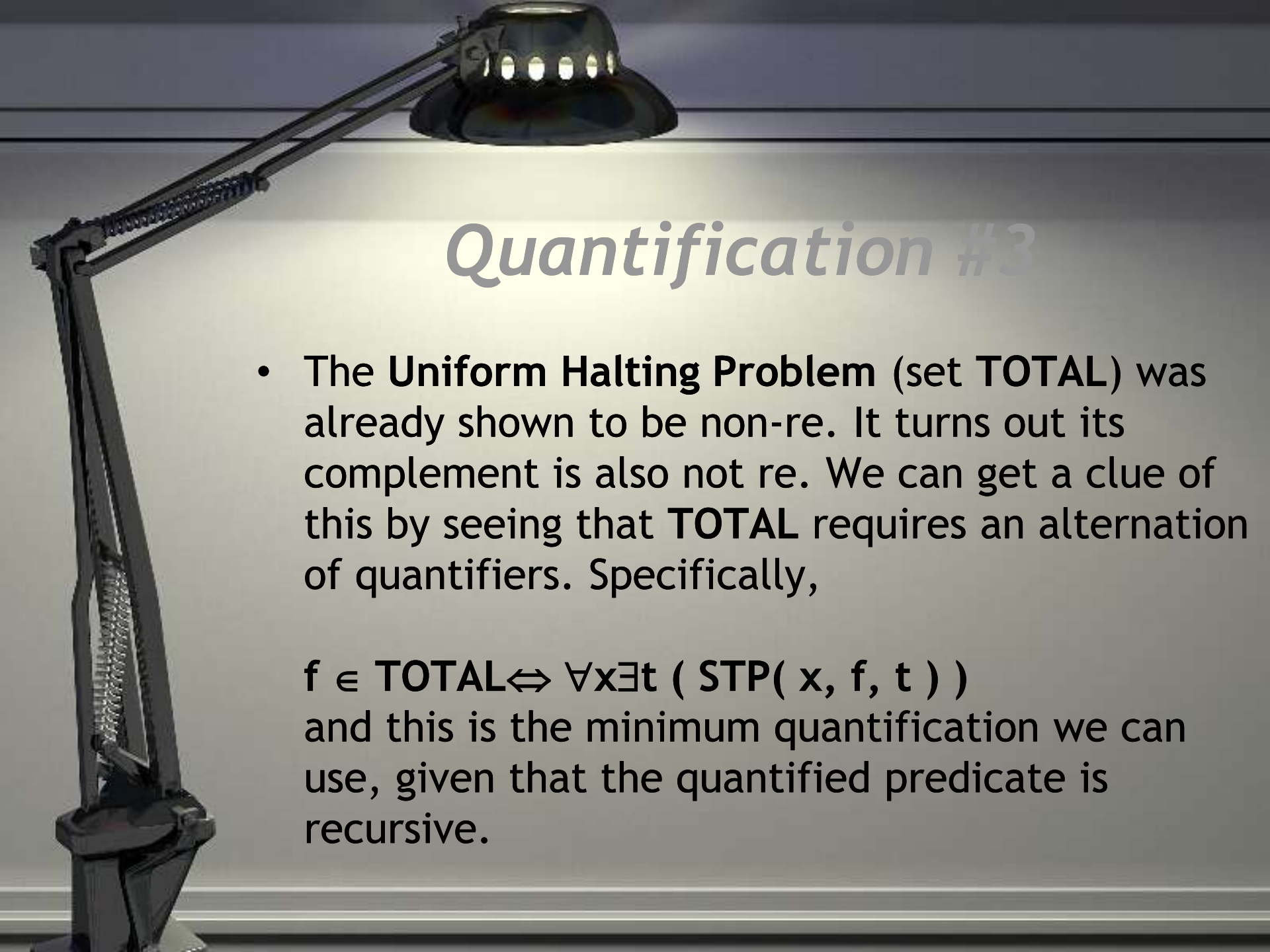

USING QUANTIFICATION TO SET
AN UPPER BOUND ON
COMPLEXITY OF SETS

Quantification #1

- **S** is decidable iff there exists an algorithm χ_S (called **S**'s characteristic function) such that
  $x \in S \Leftrightarrow \chi_S(x)$
  This is just the definition of decidable.
- **S** is re iff there exists an algorithm A_S where
  $x \in S \Leftrightarrow \exists t\ A_S(x,t)$
  This is clear since, if g_S is the index of procedure ψ_S defined earlier that semi-decides **S** then
  $x \in S \Leftrightarrow \exists t\ STP(x, g_S, t)$
  So, $A_S(x,t) = STP_{g_S}(x, t)$, where STP_{g_S} is the **STP** function with its second argument fixed.

Quantification #2

- **S** is re iff there exists an algorithm A_S such that
  $x \notin S \Leftrightarrow \forall t\ A_S(x,t)$
  This is clear since, if g_S is the index of procedure ψ_S that semi-decides **S**, then
  $x \notin S \Leftrightarrow \sim\exists t\ STP(x, g_S, t) \Leftrightarrow \forall t\ \sim STP(x, g_S, t)$
  So, $A_S(x,t) = \sim STP_{g_S}(x, t)$, where STP_{g_S} is the **STP** function with its second argument fixed.
- Note that this works even if **S** is recursive (decidable). The important thing there is that if **S** is recursive then it may be viewed in two normal forms, one with existential quantification and the other with universal quantification.
- The complement of an re set is co-re. A set is recursive (decidable) iff it is both re and co-re.



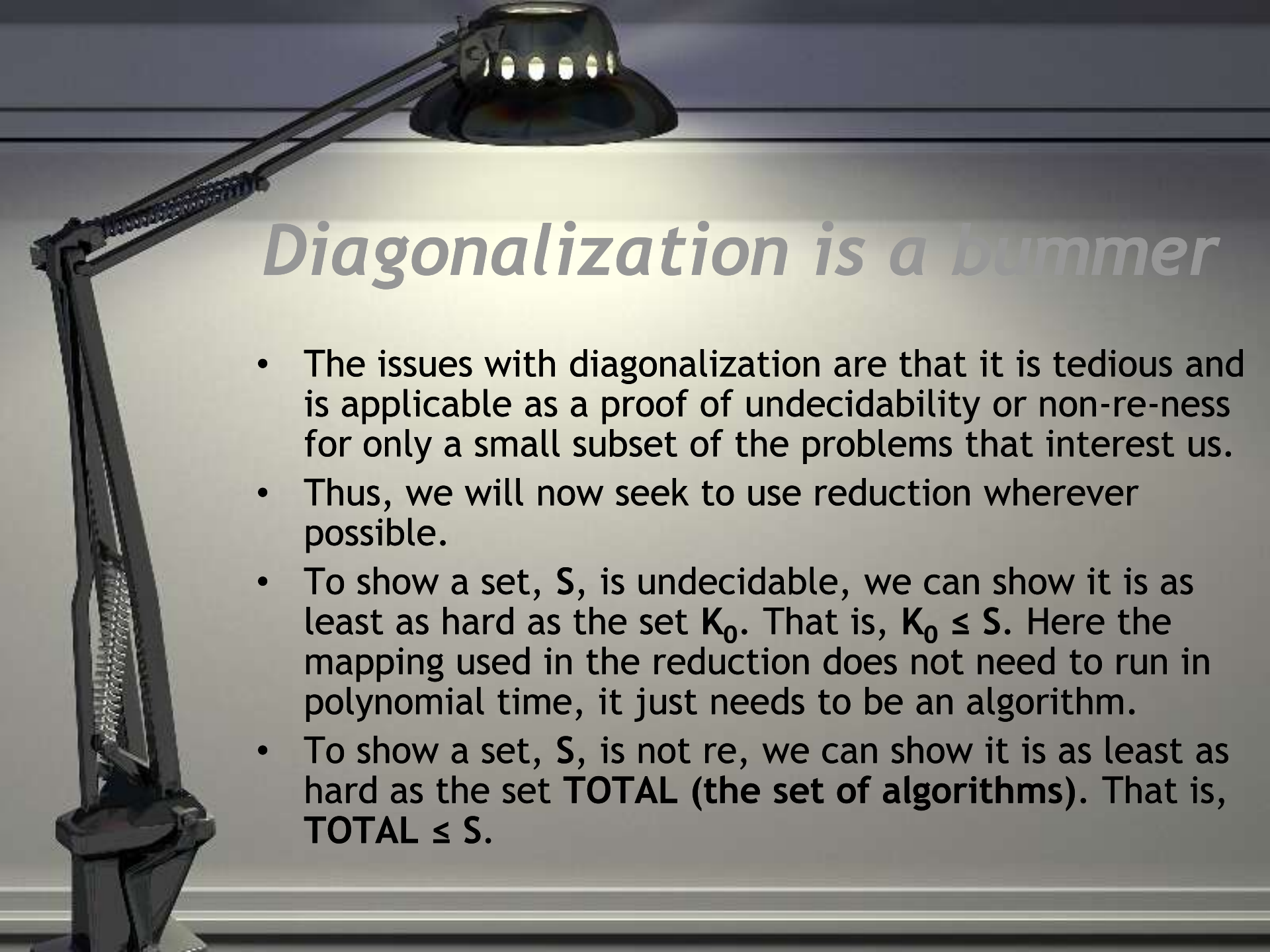
Quantification #3

- The **Uniform Halting Problem** (set **TOTAL**) was already shown to be non-re. It turns out its complement is also not re. We can get a clue of this by seeing that **TOTAL** requires an alternation of quantifiers. Specifically,

  $\mathbf{f} \in \mathbf{TOTAL} \Leftrightarrow \forall \mathbf{x} \exists \mathbf{t}\ (\ \mathbf{STP}(\ \mathbf{x}, \mathbf{f}, \mathbf{t}\ )\ )$

  and this is the minimum quantification we can use, given that the quantified predicate is recursive.


REDUCIBILITY

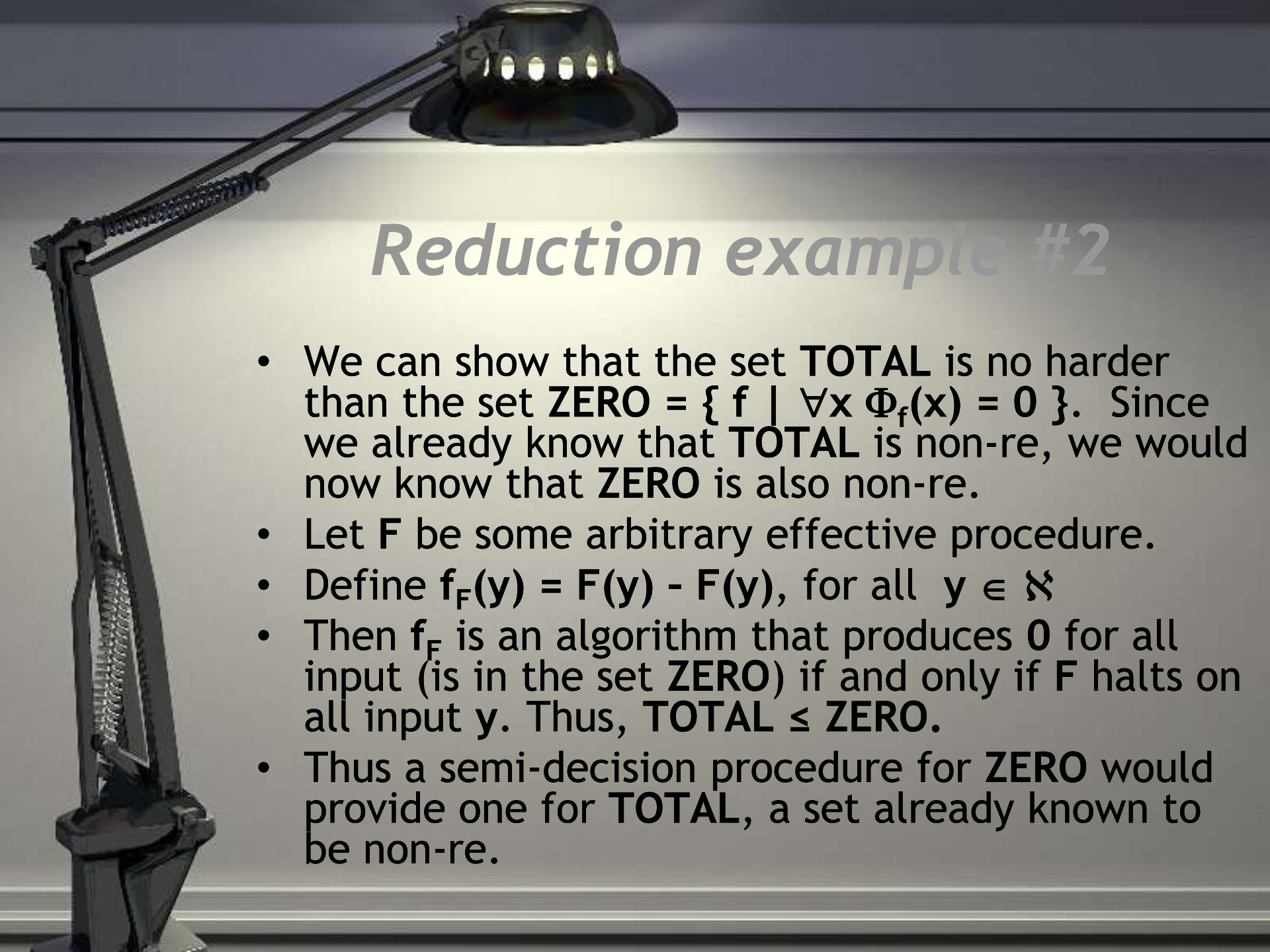


Diagonalization is a bummer

- The issues with diagonalization are that it is tedious and is applicable as a proof of undecidability or non-re-ness for only a small subset of the problems that interest us.
- Thus, we will now seek to use reduction wherever possible.
- To show a set, **S**, is undecidable, we can show it is as least as hard as the set $\mathbf{K_0}$. That is, $\mathbf{K_0 \leq S}$. Here the mapping used in the reduction does not need to run in polynomial time, it just needs to be an algorithm.
- To show a set, **S**, is not re, we can show it is as least as hard as the set **TOTAL (the set of algorithms)**. That is, **TOTAL ≤ S**.

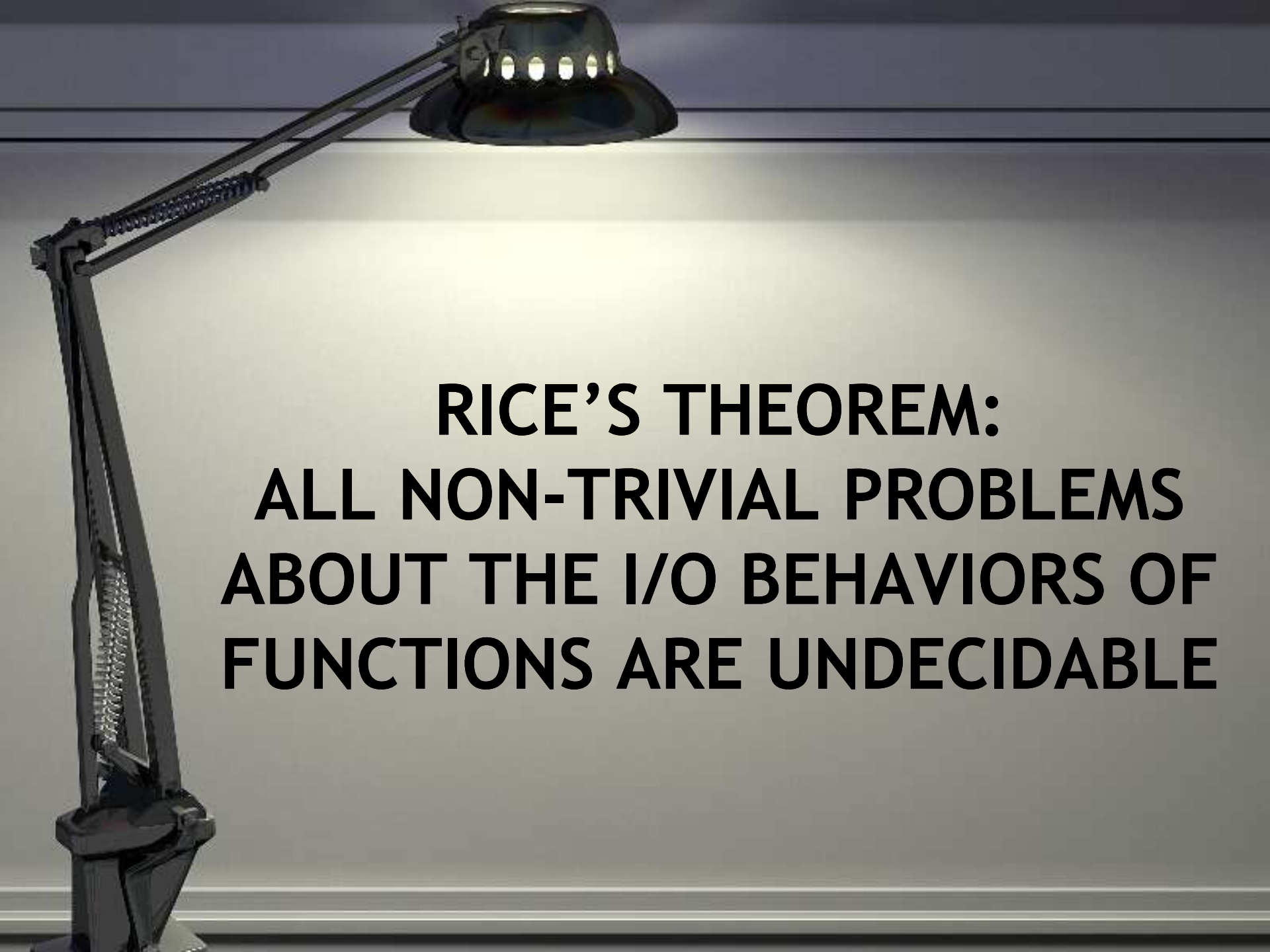
Reduction example #1

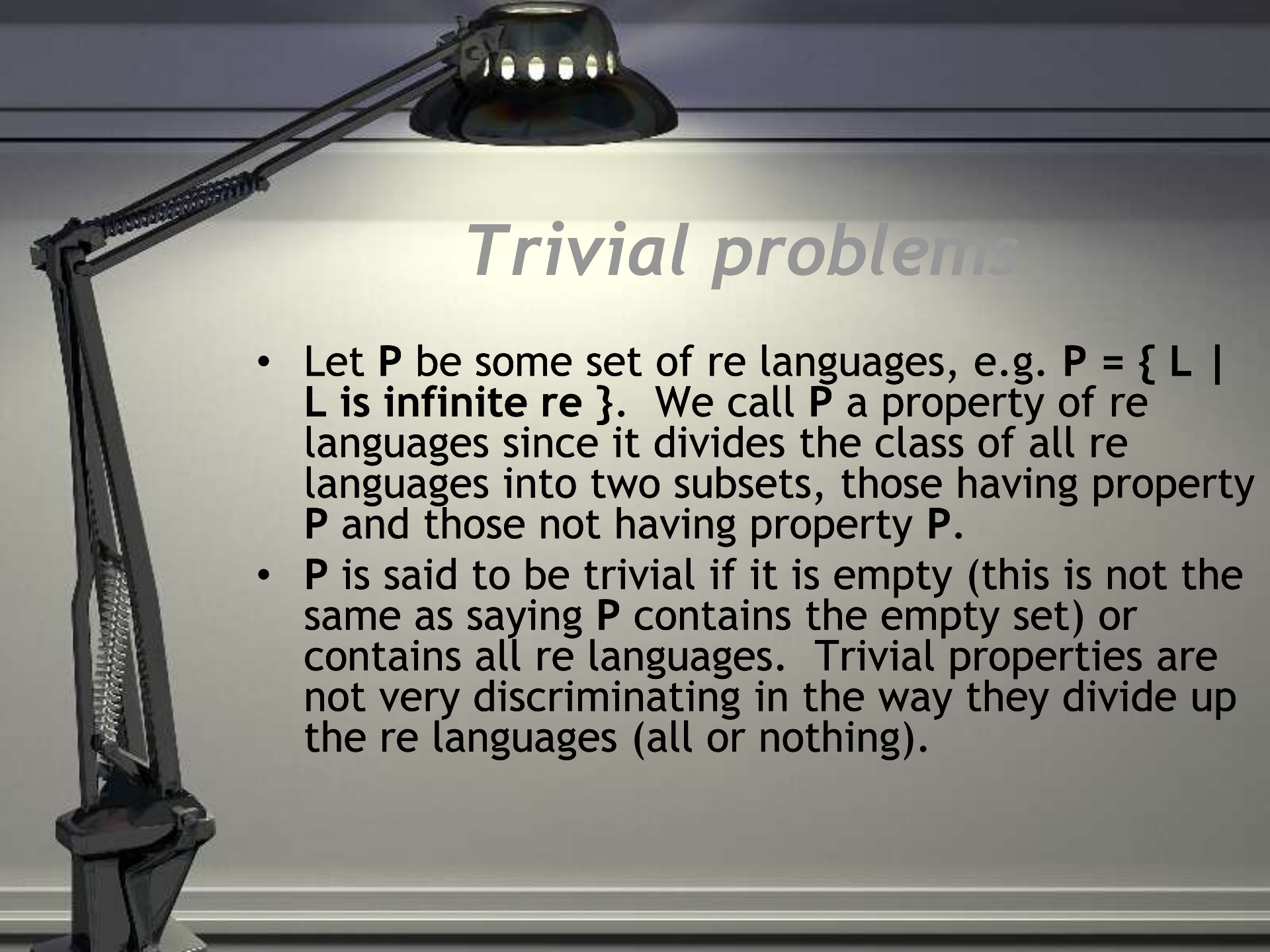
- We can show that the set K_0 is no harder than the set **TOTAL**. Since we already know that K_0 is unsolvable, we would now know that **TOTAL** is also unsolvable. We cannot reduce in the other direction since **TOTAL** is in fact harder than K_0.
- Let **F** be some arbitrary effective procedure and let x be some arbitrary natural number.
- Define $F_x(y) = F(x)$, for all $y \in \aleph$
- Then F_x is an algorithm if and only if **F** halts on **x**.
- Thus, $K_0 \leq$ **TOTAL**, and so a solution to membership in **TOTAL** would provide a solution to K_0, which we know is not possible.



Reduction example #2

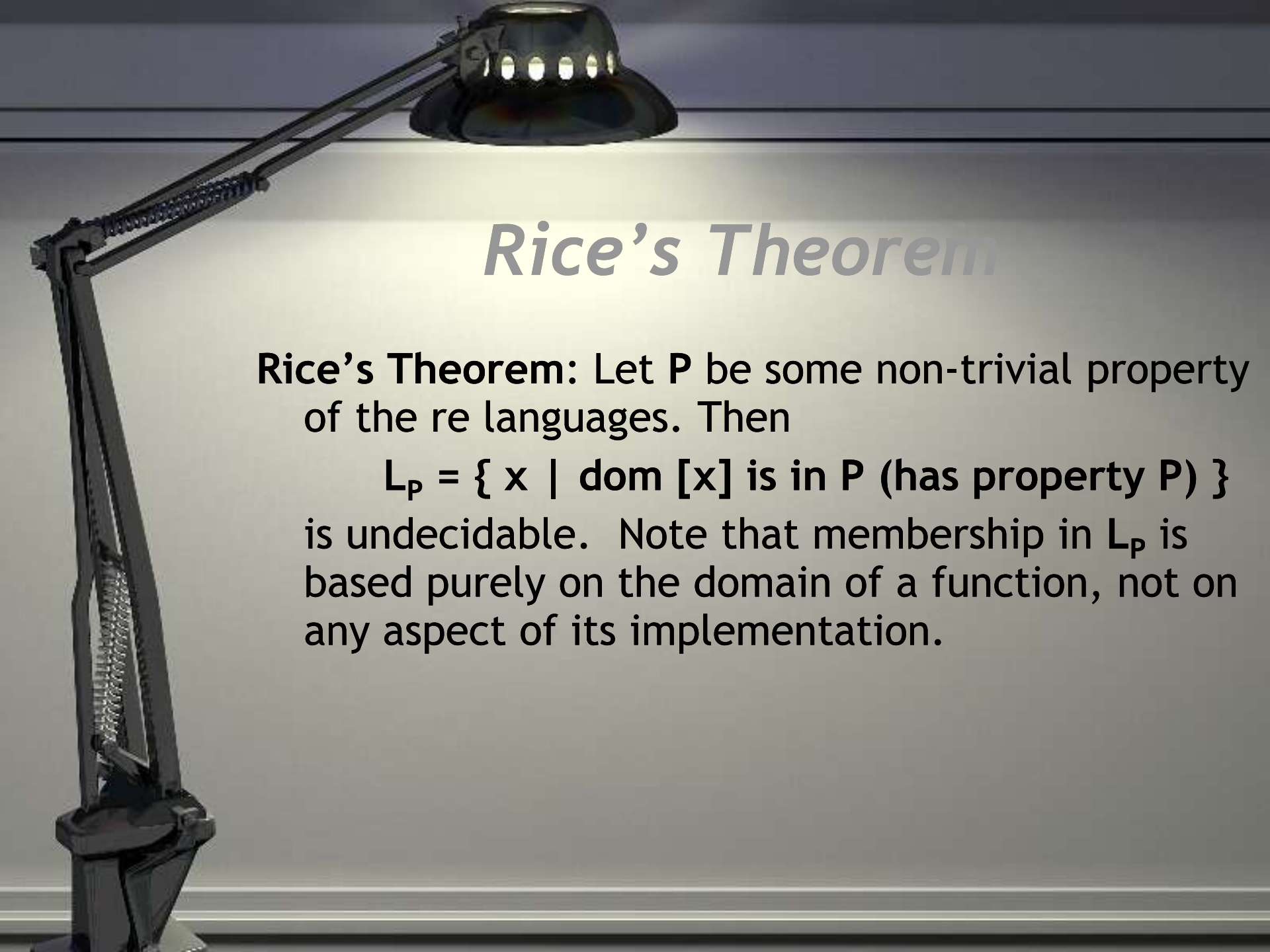
- We can show that the set **TOTAL** is no harder than the set **ZERO** = { **f** | $\forall x\ \Phi_f(x) = 0$ }. Since we already know that **TOTAL** is non-re, we would now know that **ZERO** is also non-re.
- Let **F** be some arbitrary effective procedure.
- Define $f_F(y) = F(y) - F(y)$, for all $y \in \aleph$
- Then f_F is an algorithm that produces **0** for all input (is in the set **ZERO**) if and only if **F** halts on all input **y**. Thus, **TOTAL ≤ ZERO.**
- Thus a semi-decision procedure for **ZERO** would provide one for **TOTAL**, a set already known to be non-re.


RICE'S THEOREM:
ALL NON-TRIVIAL PROBLEMS
ABOUT THE I/O BEHAVIORS OF
FUNCTIONS ARE UNDECIDABLE



Trivial problems

- Let **P** be some set of re languages, e.g. **P = { L | L is infinite re }**. We call **P** a property of re languages since it divides the class of all re languages into two subsets, those having property **P** and those not having property **P**.
- **P** is said to be trivial if it is empty (this is not the same as saying **P** contains the empty set) or contains all re languages. Trivial properties are not very discriminating in the way they divide up the re languages (all or nothing).

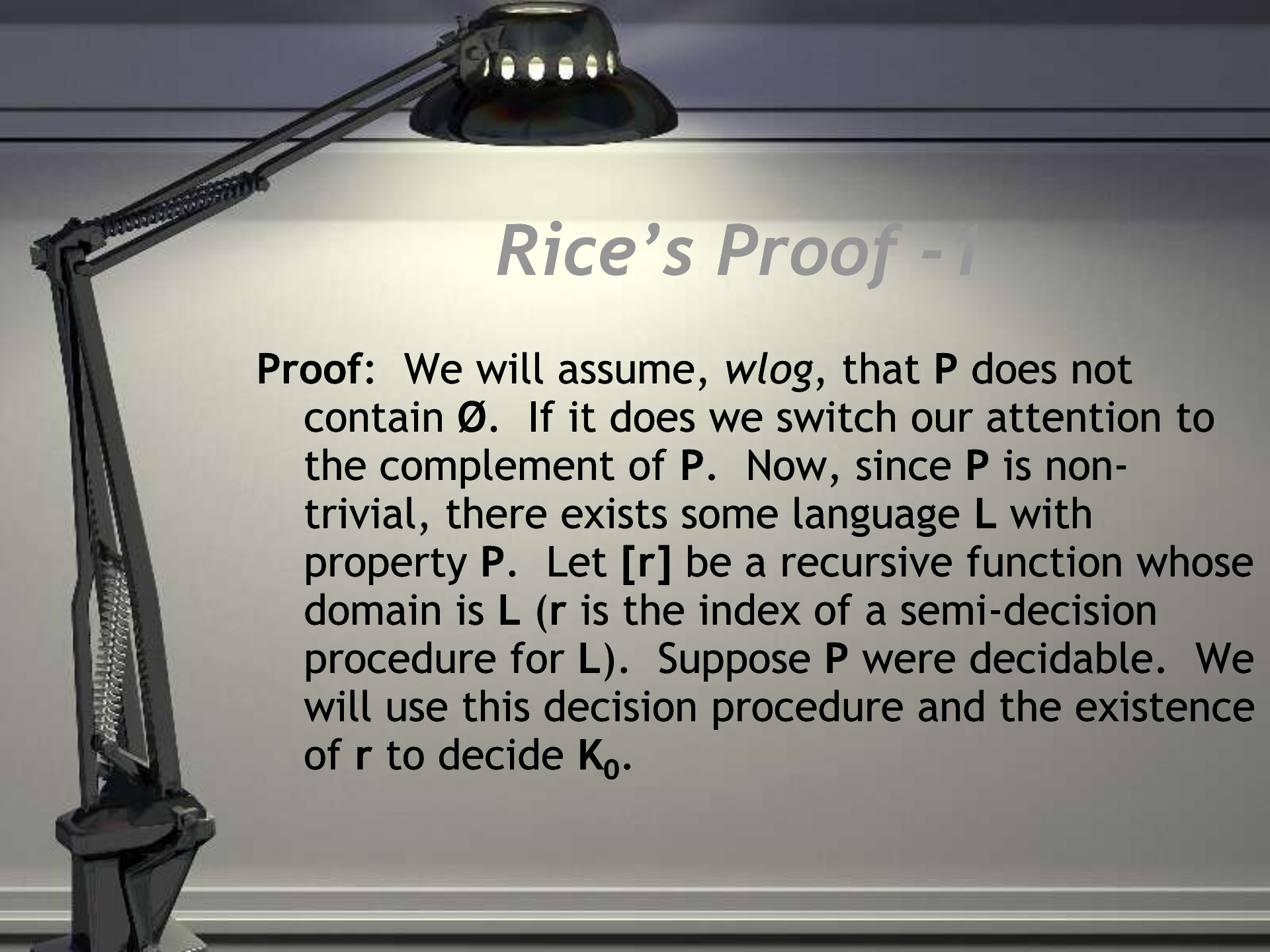


Rice's Theorem

Rice's Theorem: Let **P** be some non-trivial property of the re languages. Then

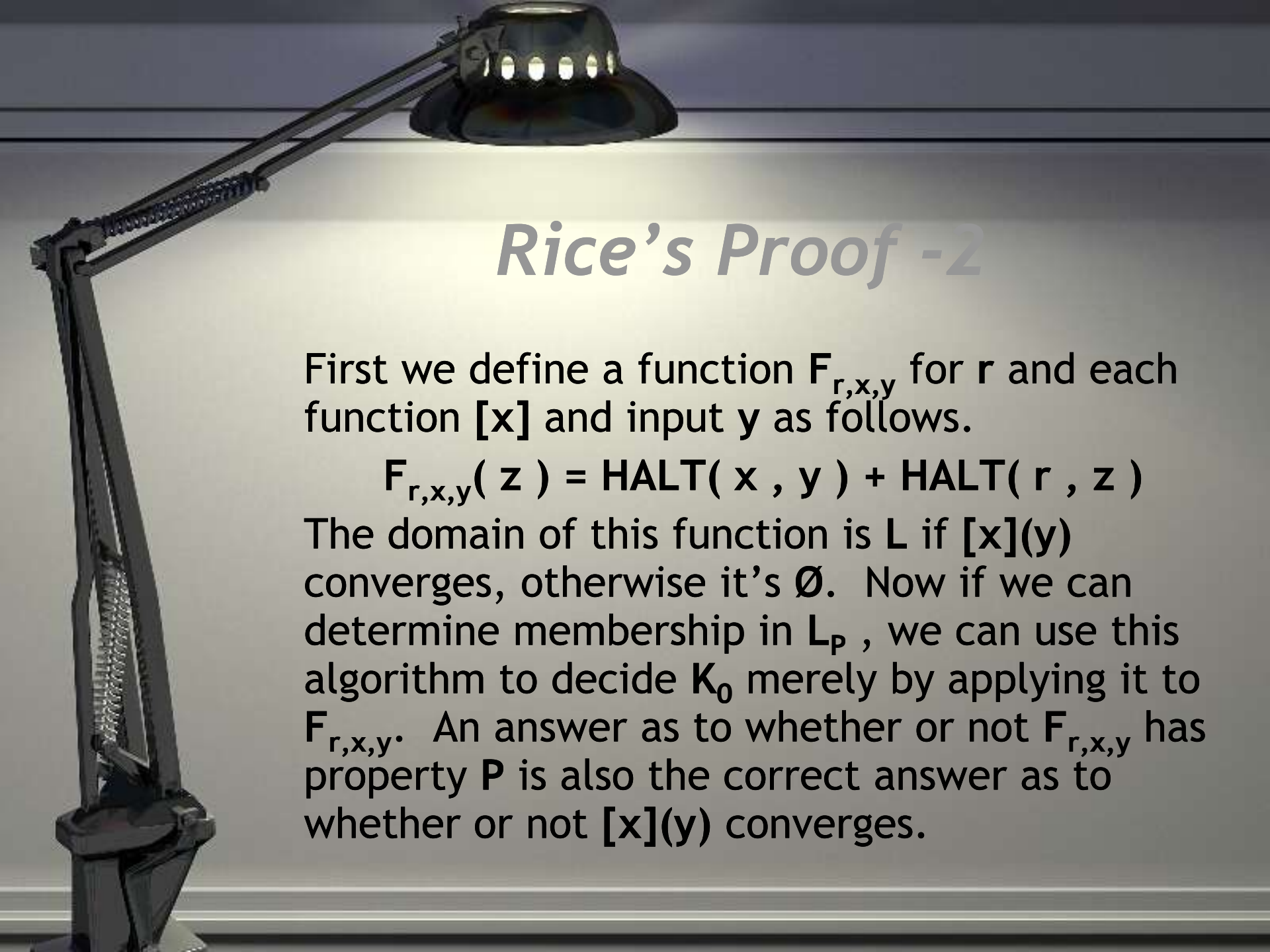
$$L_P = \{ x \mid \text{dom } [x] \text{ is in P (has property P)} \}$$

is undecidable. Note that membership in L_P is based purely on the domain of a function, not on any aspect of its implementation.



Rice's Proof - 1

Proof: We will assume, *wlog*, that **P** does not contain **Ø**. If it does we switch our attention to the complement of **P**. Now, since **P** is non-trivial, there exists some language **L** with property **P**. Let **[r]** be a recursive function whose domain is **L** (**r** is the index of a semi-decision procedure for **L**). Suppose **P** were decidable. We will use this decision procedure and the existence of **r** to decide $\mathbf{K_0}$.

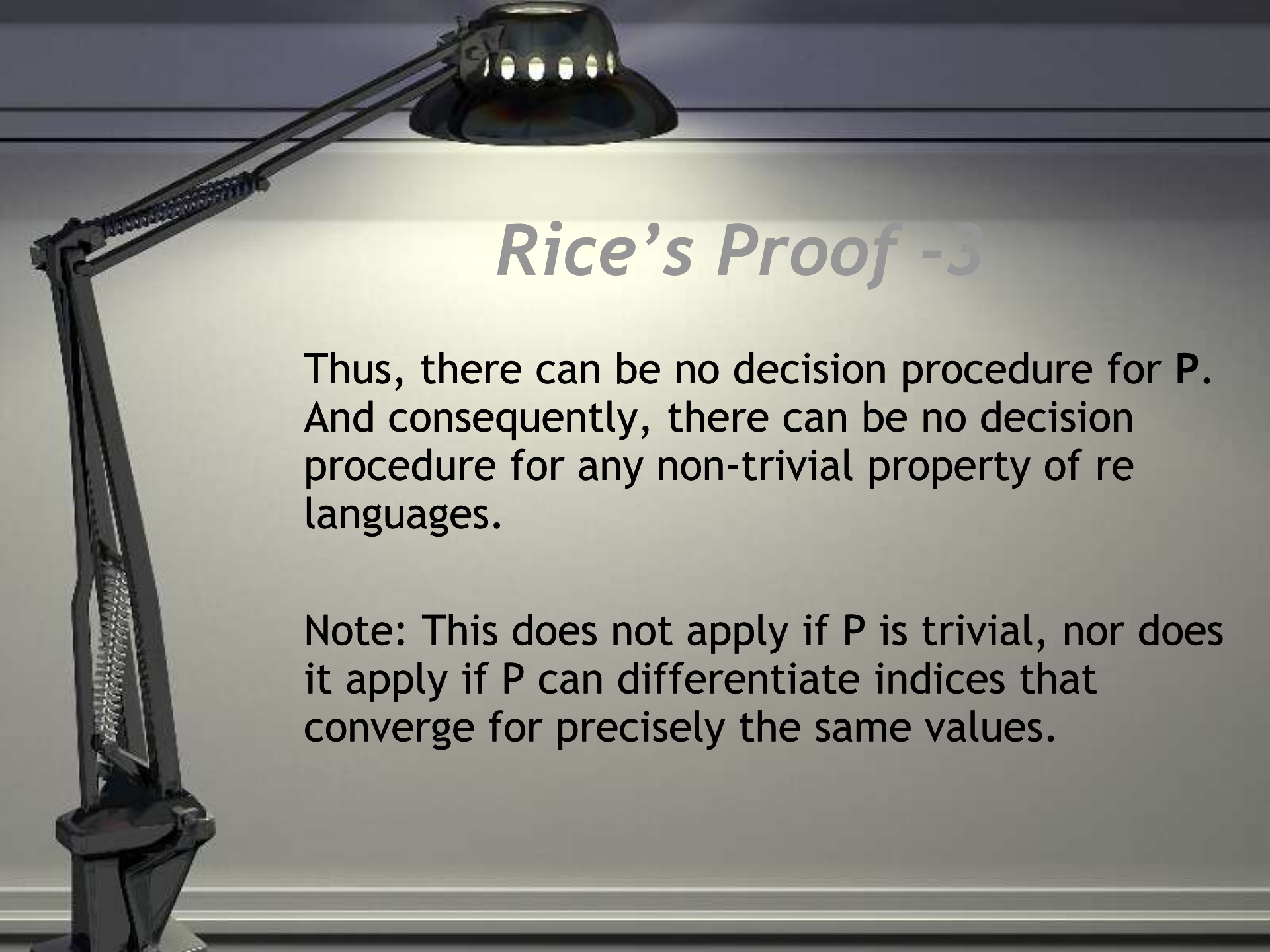


Rice's Proof -2

First we define a function $F_{r,x,y}$ for **r** and each function **[x]** and input **y** as follows.

$$F_{r,x,y}(z) = HALT(x, y) + HALT(r, z)$$

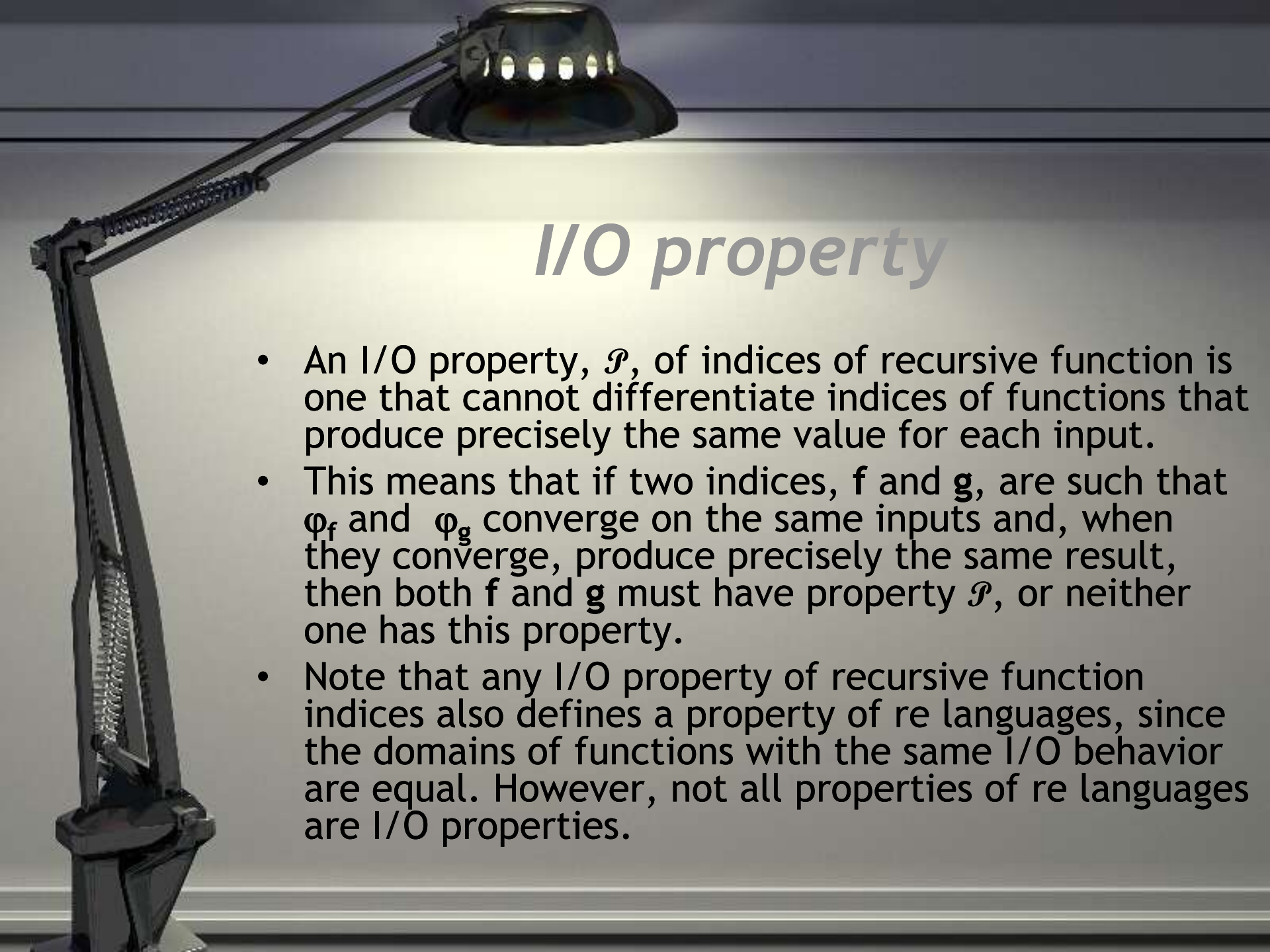
The domain of this function is **L** if **[x](y)** converges, otherwise it's **Ø**. Now if we can determine membership in L_P , we can use this algorithm to decide K_0 merely by applying it to $F_{r,x,y}$. An answer as to whether or not $F_{r,x,y}$ has property **P** is also the correct answer as to whether or not **[x](y)** converges.



Rice's Proof -3

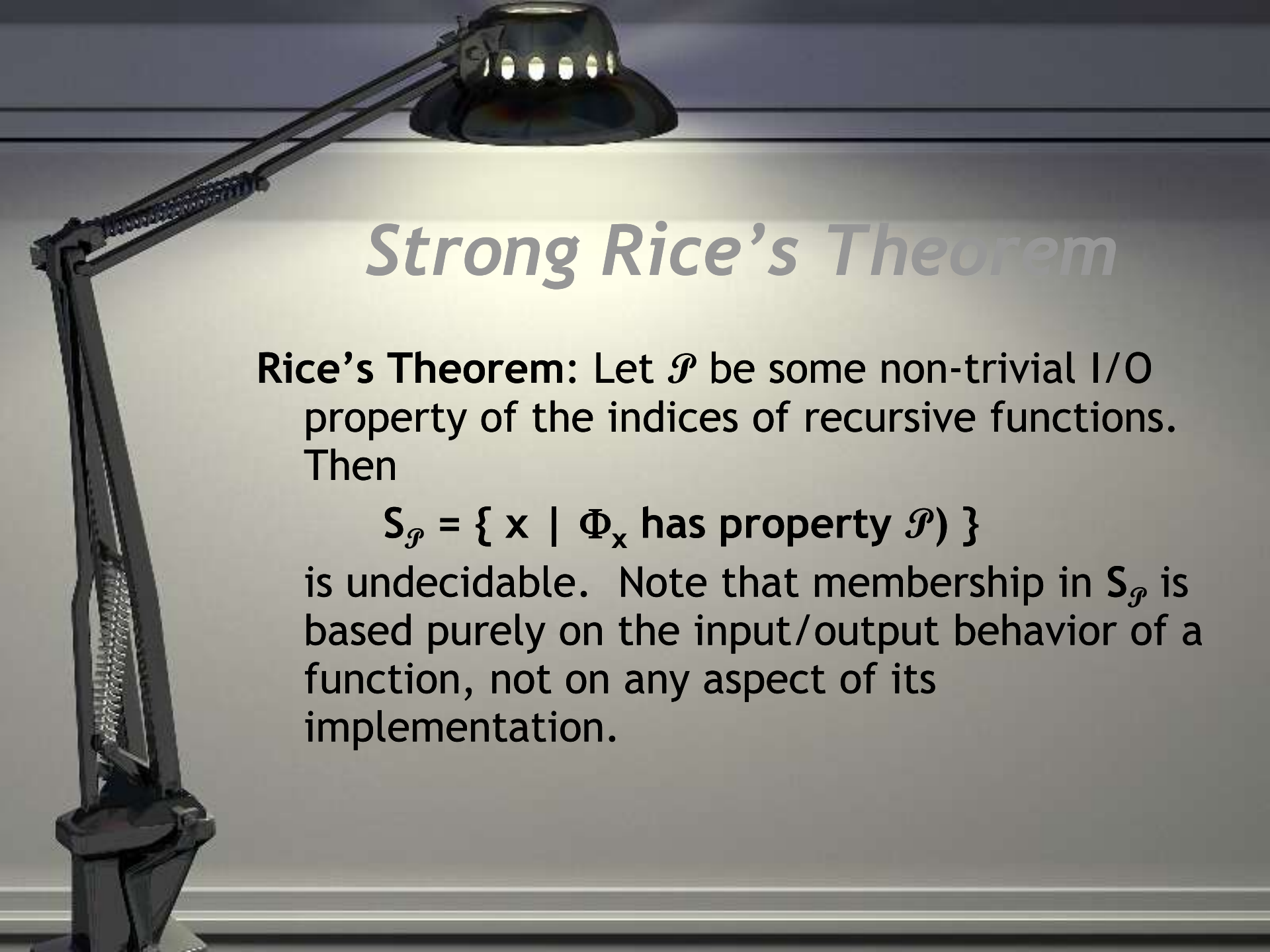
Thus, there can be no decision procedure for **P**. And consequently, there can be no decision procedure for any non-trivial property of re languages.

Note: This does not apply if P is trivial, nor does it apply if P can differentiate indices that converge for precisely the same values.



I/O property

- An I/O property, $\mathcal{P}$, of indices of recursive function is one that cannot differentiate indices of functions that produce precisely the same value for each input.
- This means that if two indices, **f** and **g**, are such that φ_f and φ_g converge on the same inputs and, when they converge, produce precisely the same result, then both **f** and **g** must have property $\mathcal{P}$, or neither one has this property.
- Note that any I/O property of recursive function indices also defines a property of re languages, since the domains of functions with the same I/O behavior are equal. However, not all properties of re languages are I/O properties.

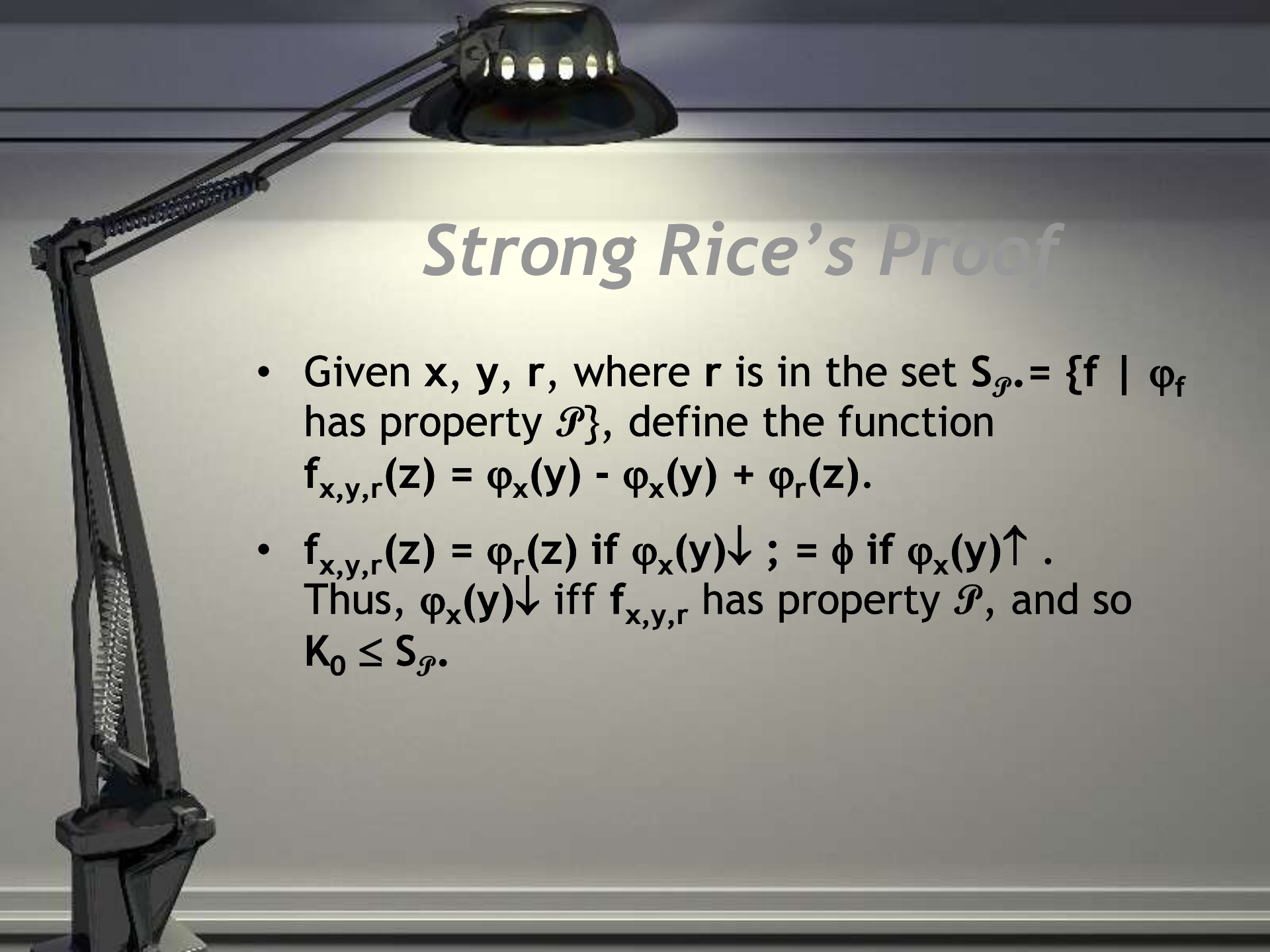


Strong Rice's Theorem

Rice's Theorem: Let $\mathcal{P}$ be some non-trivial I/O property of the indices of recursive functions. Then

$$S_{\mathcal{P}} = \{ x \mid \Phi_x \text{ has property } \mathcal{P}) \}$$

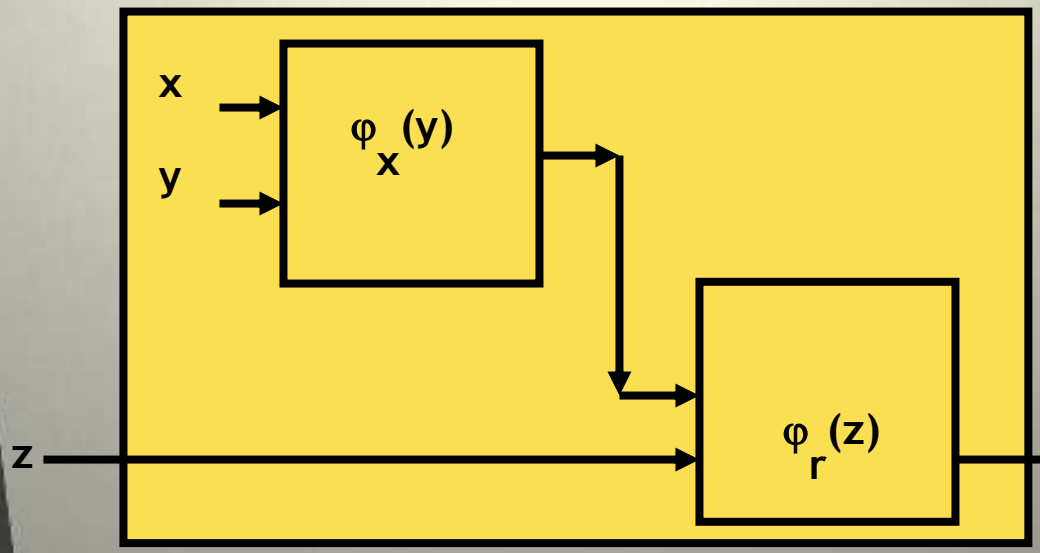
is undecidable. Note that membership in $S_{\mathcal{P}}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.



Strong Rice's Proof

- Given **x**, **y**, **r**, where **r** is in the set $S_{\mathcal{P}} = \{f \mid \varphi_f$ has property $\mathcal{P}\}$, define the function $f_{x,y,r}(z) = \varphi_x(y) - \varphi_x(y) + \varphi_r(z)$.
- $f_{x,y,r}(z) = \varphi_r(z)$ if $\varphi_x(y)\downarrow$; $= \phi$ if $\varphi_x(y)\uparrow$. Thus, $\varphi_x(y)\downarrow$ iff $f_{x,y,r}$ has property $\mathcal{P}$, and so $K_0 \leq S_{\mathcal{P}}$.

Picture Proofs



$\forall z\ f_{x,y,r}(z)=\varphi_r(z)$ If $\varphi_x(y)\downarrow$

$rng(f_{x,y,r})=rng(\varphi_r)$ If $\varphi_x(y)\downarrow$

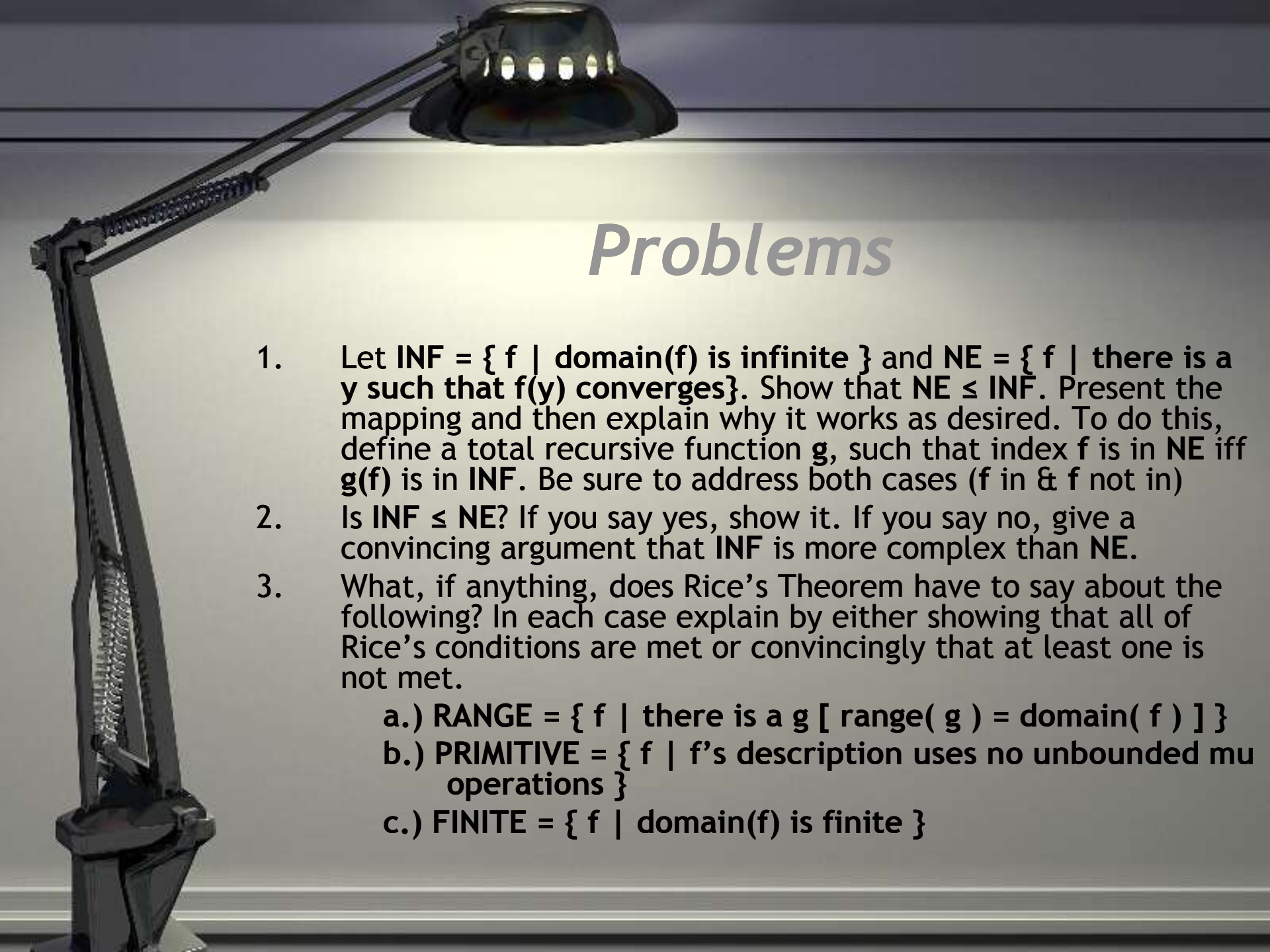
$dom(f_{x,y,r})=dom(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$rng(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$\exists z\ f_{x,y,r}(z)\neq\varphi_r(z)$ If $\varphi_x(y)\uparrow$

Black is for standard Rice's Theorem;
Black and Red are needed for Strong Version
Blue is just another version based on range



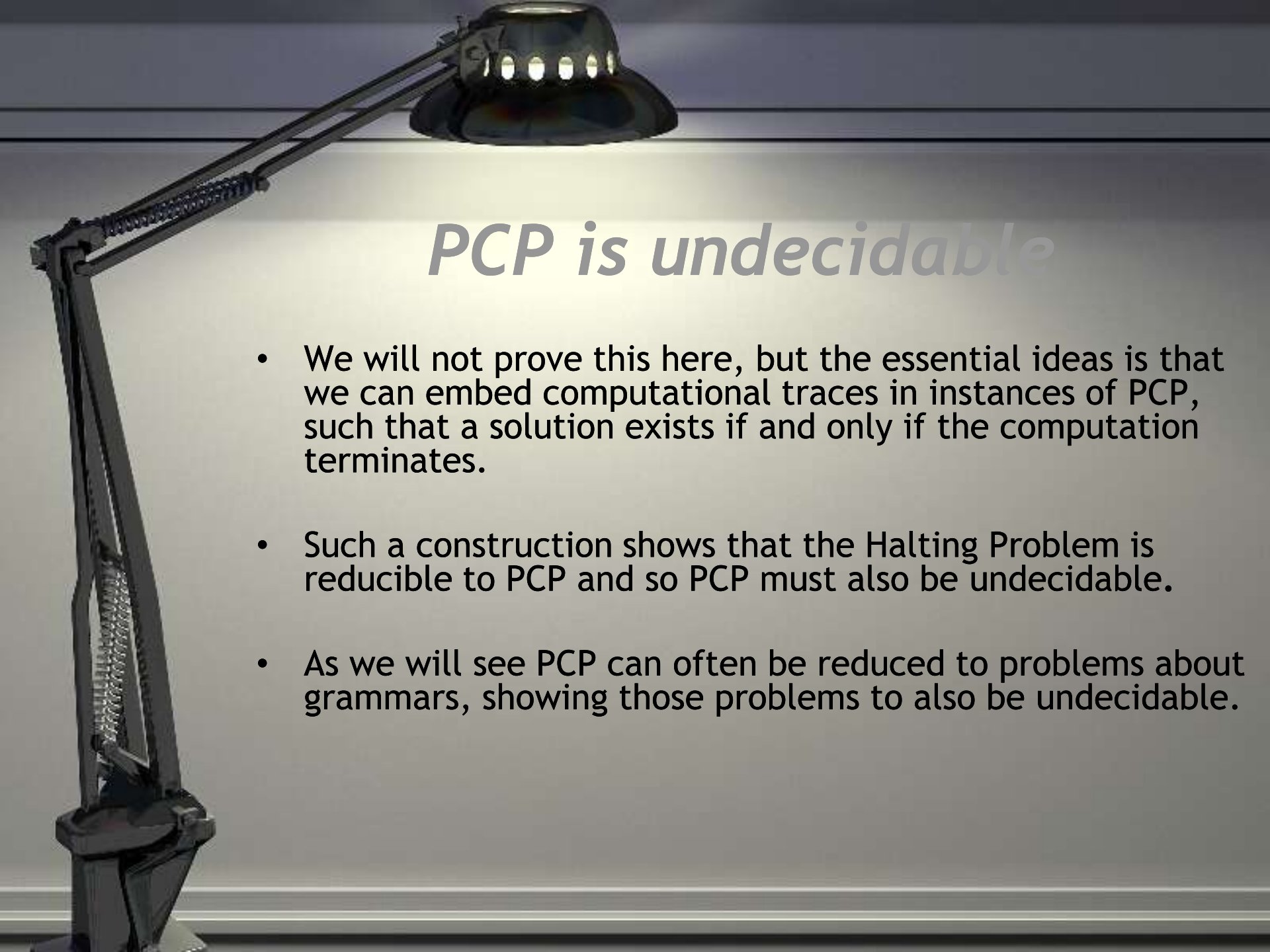
Problems

1. Let **INF = { f | domain(f) is infinite }** and **NE = { f | there is a y such that f(y) converges}**. Show that **NE ≤ INF**. Present the mapping and then explain why it works as desired. To do this, define a total recursive function **g**, such that index **f** is in **NE** iff **g(f)** is in **INF**. Be sure to address both cases (**f** in & **f** not in)
2. Is **INF ≤ NE**? If you say yes, show it. If you say no, give a convincing argument that **INF** is more complex than **NE**.
3. What, if anything, does Rice's Theorem have to say about the following? In each case explain by either showing that all of Rice's conditions are met or convincingly that at least one is not met.
   - **a.) RANGE = { f | there is a g [range(g) = domain(f)] }**
   - **b.) PRIMITIVE = { f | f's description uses no unbounded mu operations }**
   - **c.) FINITE = { f | domain(f) is finite }**


GRAMMARS

Post Correspondence

- Many problems related to grammars can be shown to be no more complex than the Post Correspondence Problem (PCP).
- Each instance of PCP is denoted: Given **n>0**, Σ a finite alphabet, and two n-tuples of words
  $(x_1, \ldots , x_n)$, $(y_1, \ldots , y_n)$ over Σ,
  does there exist a sequence $i_1, \ldots , i_k$, $k>0$, $1 \le i_j \le n$, such that $x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$?
- Example of PCP:
  n = 3, Σ = { **a , b** },
  x = (a b a , b b , a), y = (b a b , b , b a a).
  Solution 2 , 3, 1 , 2
  b b a a b a b b = b b a a b a b b



PCP is undecidable

- We will not prove this here, but the essential ideas is that we can embed computational traces in instances of PCP, such that a solution exists if and only if the computation terminates.
- Such a construction shows that the Halting Problem is reducible to PCP and so PCP must also be undecidable.
- As we will see PCP can often be reduced to problems about grammars, showing those problems to also be undecidable.

Ambiguity of CFG

- Problem to determine if an arbitrary CFG is ambiguous

  $S \rightarrow A \mid B$

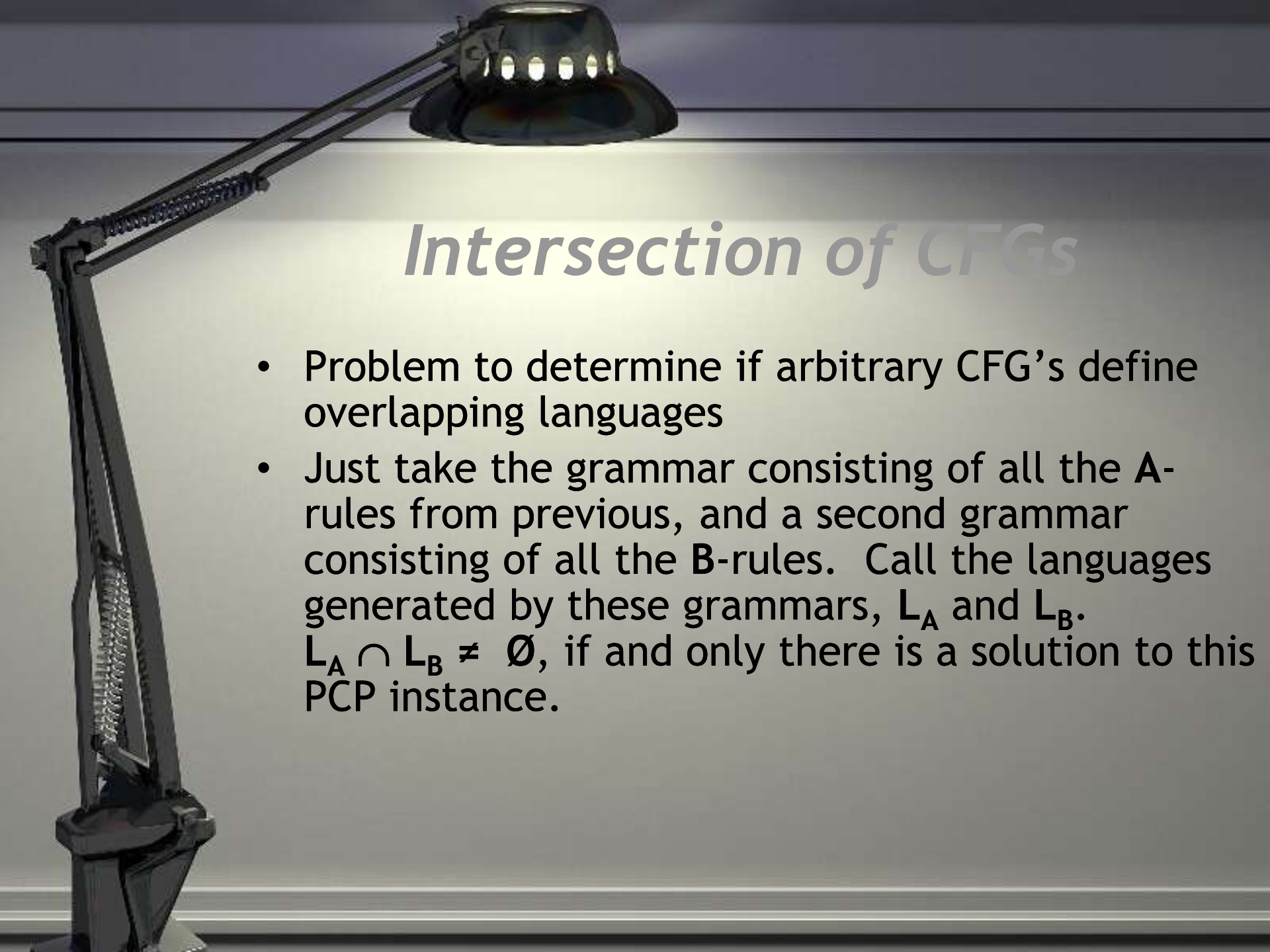
  $A \rightarrow x_i\ A\ [i] \mid x_i\ [i]$ $\quad 1 \le i \le n$

  $B \rightarrow y_i\ B\ [i] \mid y_i\ [i]$ $\quad 1 \le i \le n$

  $A \Rightarrow^* x_{i_1} \ldots x_{i_k}\ [i_k] \ldots [i_1]$ $\quad k > 0$

  $B \Rightarrow^* y_{i_1} \ldots y_{i_k}\ [i_k] \ldots [i_1]$ $\quad k > 0$

- Ambiguous if and only if there is a solution to this PCP instance.



Intersection of CFGs

- Problem to determine if arbitrary CFG's define overlapping languages
- Just take the grammar consisting of all the **A**-rules from previous, and a second grammar consisting of all the **B**-rules. Call the languages generated by these grammars, L_A and L_B. $L_A \cap L_B \neq \emptyset$, if and only there is a solution to this PCP instance.

Non-emptiness of CSL

$S \rightarrow x_i\ S\ y_i^R \mid x_i\ T\ y_i^R \quad 1 \le i \le n$

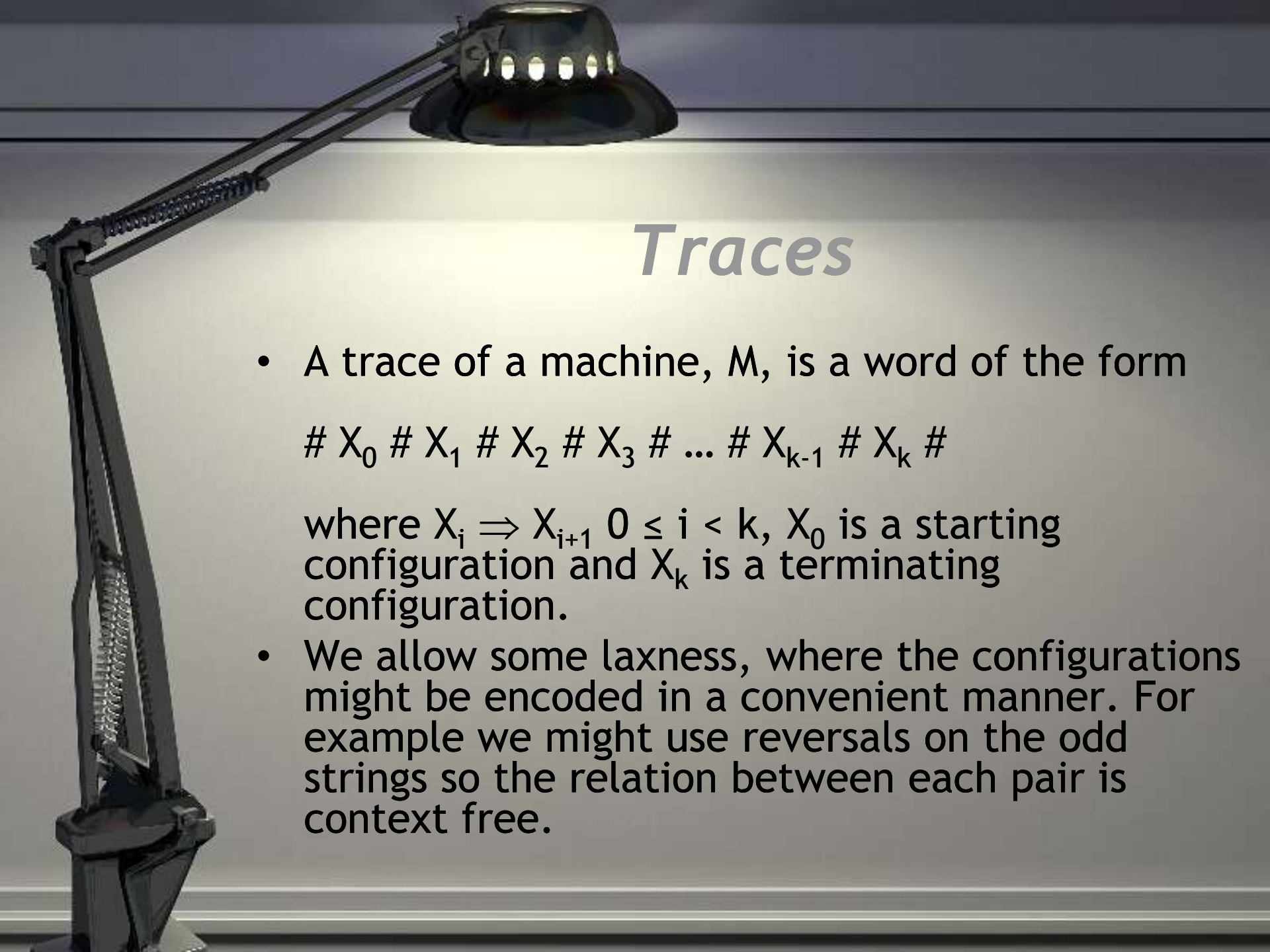
$a\ T\ a \rightarrow *\ T\ *$

$*\ a \rightarrow a\ *$

$a\ * \rightarrow *\ a$

$T \rightarrow *$

- Our only terminal is *. We get strings of form $*^{2j+1}$, for some j's if and only if there is a solution to this PCP instance.

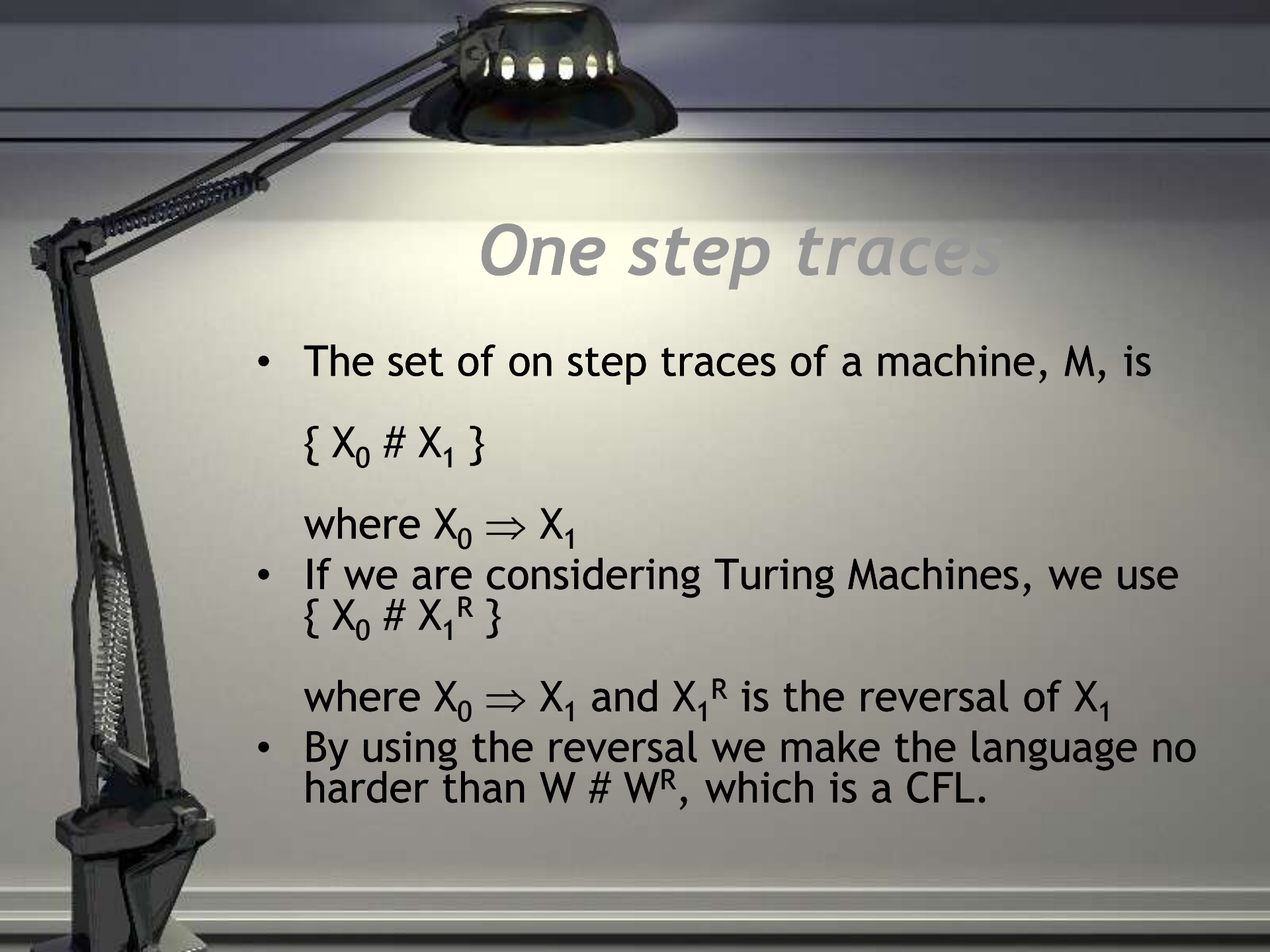


Traces

- A trace of a machine, M, is a word of the form

  $\# X_0 \# X_1 \# X_2 \# X_3 \# \ldots \# X_{k-1} \# X_k \#$

  where $X_i \Rightarrow X_{i+1}$ $0 \leq i < k$, X_0 is a starting configuration and X_k is a terminating configuration.
- We allow some laxness, where the configurations might be encoded in a convenient manner. For example we might use reversals on the odd strings so the relation between each pair is context free.



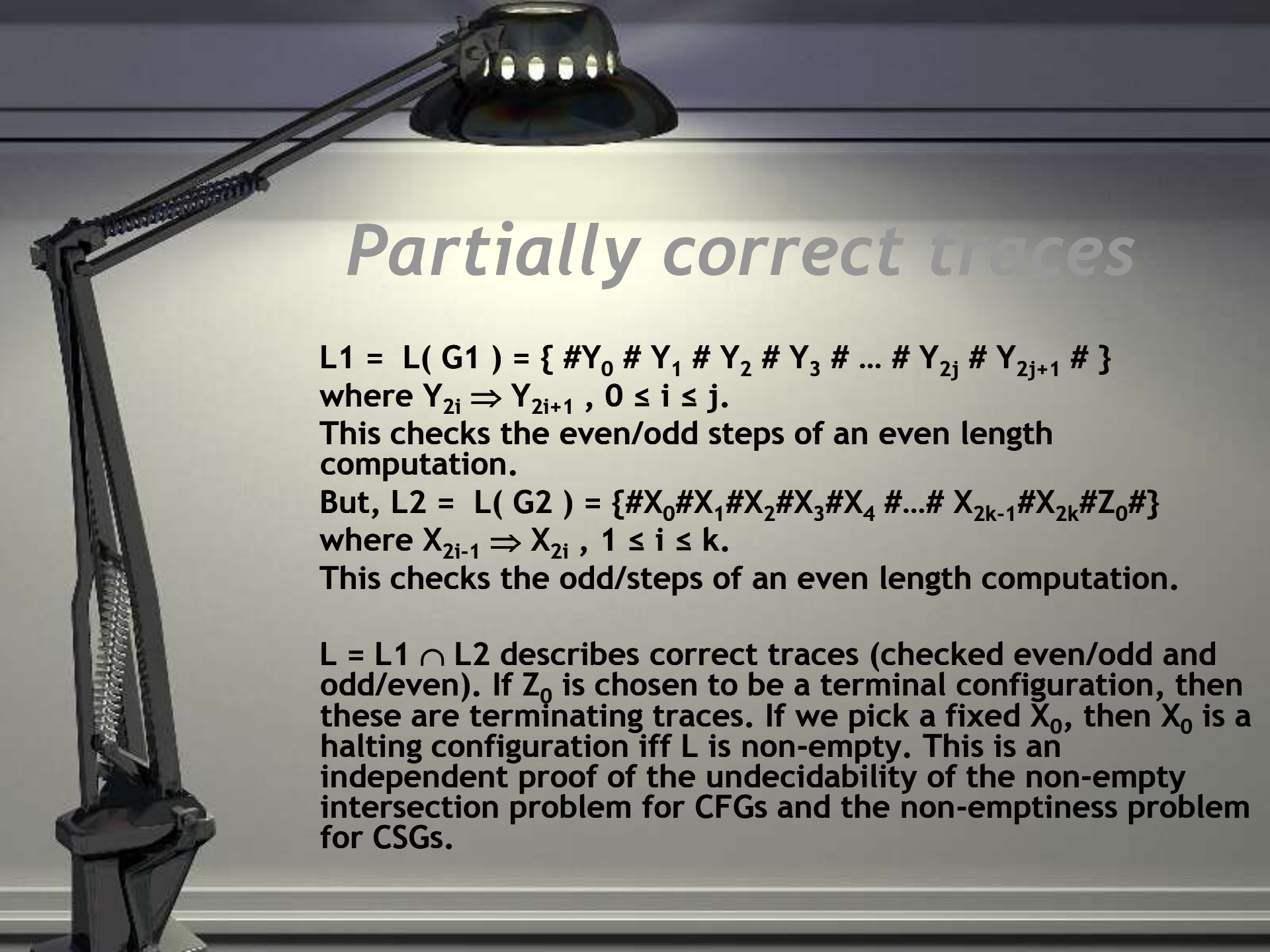
One step traces

- The set of on step traces of a machine, M, is

  $\{ X_0 \# X_1 \}$

  where $X_0 \Rightarrow X_1$
- If we are considering Turing Machines, we use $\{ X_0 \# X_1^R \}$

  where $X_0 \Rightarrow X_1$ and X_1^R is the reversal of X_1
- By using the reversal we make the language no harder than $W \# W^R$, which is a CFL.



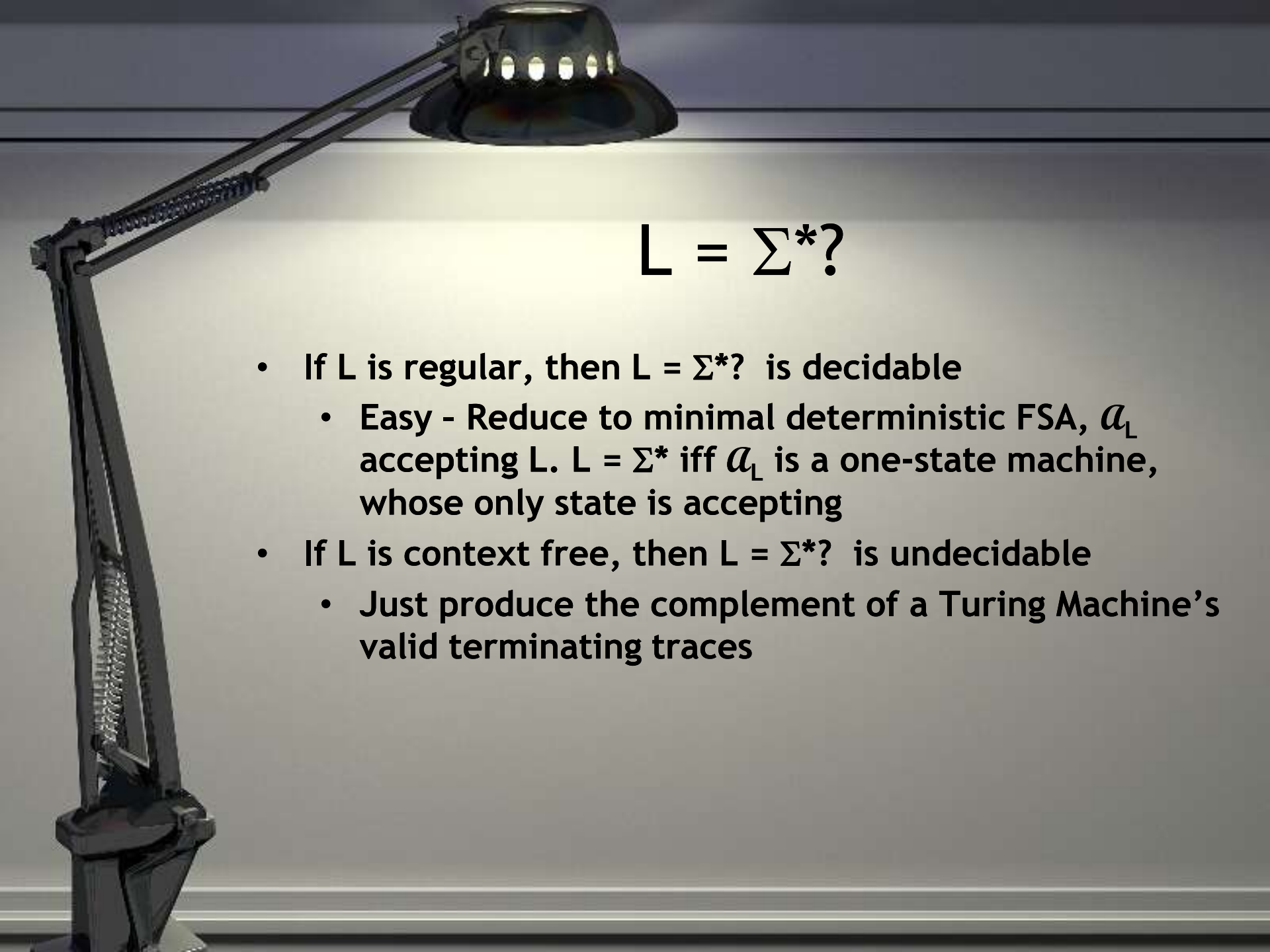
Partially correct traces

L1 = L(G1) = { $\#Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \#$ }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.
This checks the even/odd steps of an even length computation.
But, L2 = L(G2) = {$\#X_0\#X_1\#X_2\#X_3\#X_4 \#\ldots\# X_{2k-1}\#X_{2k}\#Z_0\#$}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$.
This checks the odd/steps of an even length computation.

L = L1 $\cap$ L2 describes correct traces (checked even/odd and odd/even). If Z_0 is chosen to be a terminal configuration, then these are terminating traces. If we pick a fixed X_0, then X_0 is a halting configuration iff L is non-empty. This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.

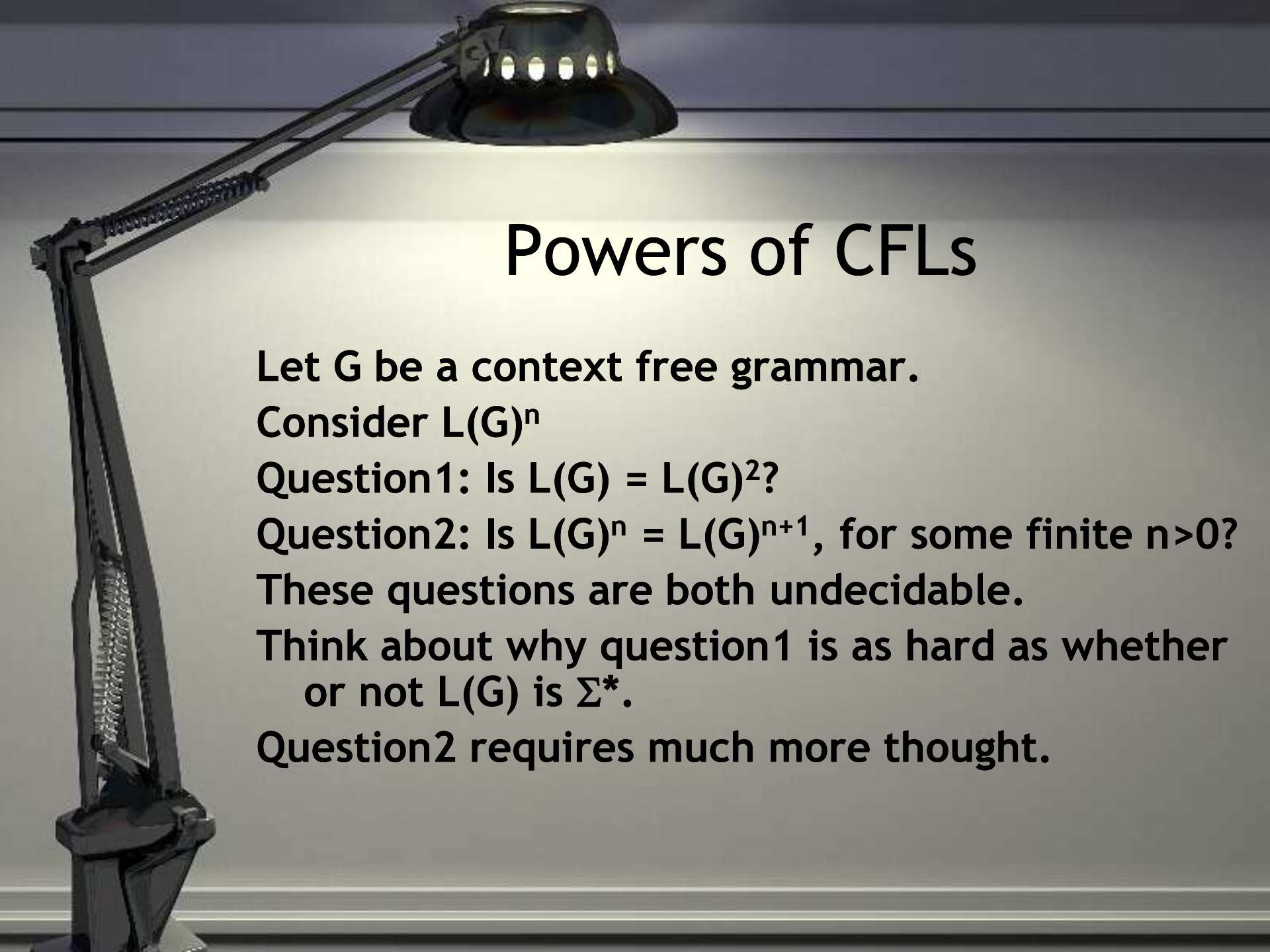
Quotients of CFLs

L1 = L(G1) = { $ #Y_0 # Y_1 # Y_2 # Y_3 # … # Y_{2j} # Y_{2j+1} # }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
This checks the even/odd steps of an even length computation.
But, L2 = L(G2) = {X_0 $ #X_0 # X_1 # X_2 # X_3 # X_4 # … # X_{2k-1} # X_{2k}# Z_0 #}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ and Z is a unique halting configuration.
This checks the odd/steps of an even length computation, and includes an extra copy of the starting number prior to its $.
Now, consider the quotient of L2 / L1 . The only ways a member of L1 can match a final substring in L2 is to line up the $ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting point (the one on which the machine halts.) Thus,
L2 / L1 = { X_0 | the system halts}.
Since deciding the members of an re set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.



L = Σ*?

- If L is regular, then L = Σ*? is decidable
  - Easy – Reduce to minimal deterministic FSA, a_L accepting L. L = Σ* iff a_L is a one-state machine, whose only state is accepting
- If L is context free, then L = Σ*? is undecidable
  - Just produce the complement of a Turing Machine's valid terminating traces



Powers of CFLs

Let G be a context free grammar.

Consider $L(G)^n$

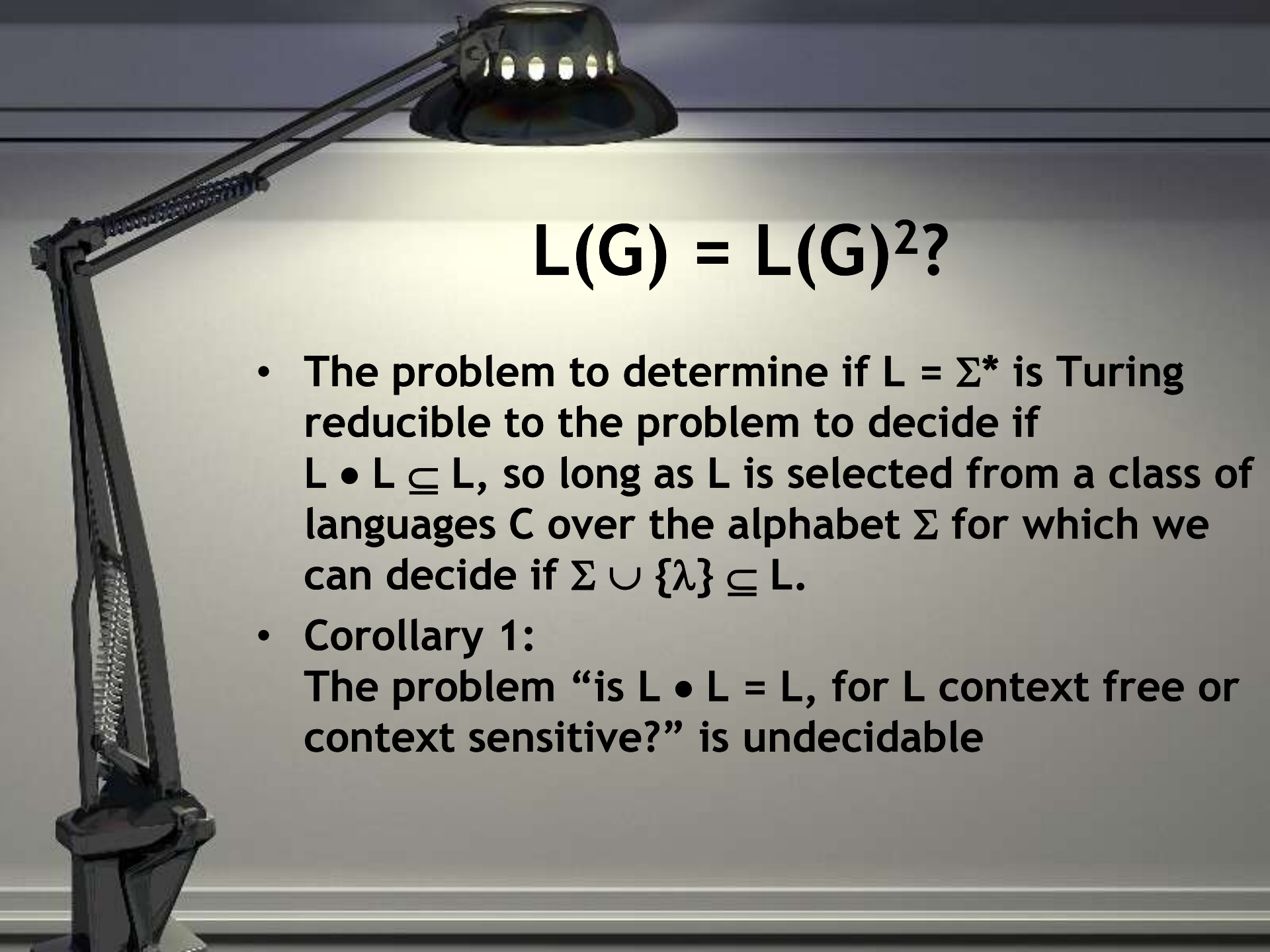
Question1: Is $L(G) = L(G)^2$?

Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n>0$?

These questions are both undecidable.

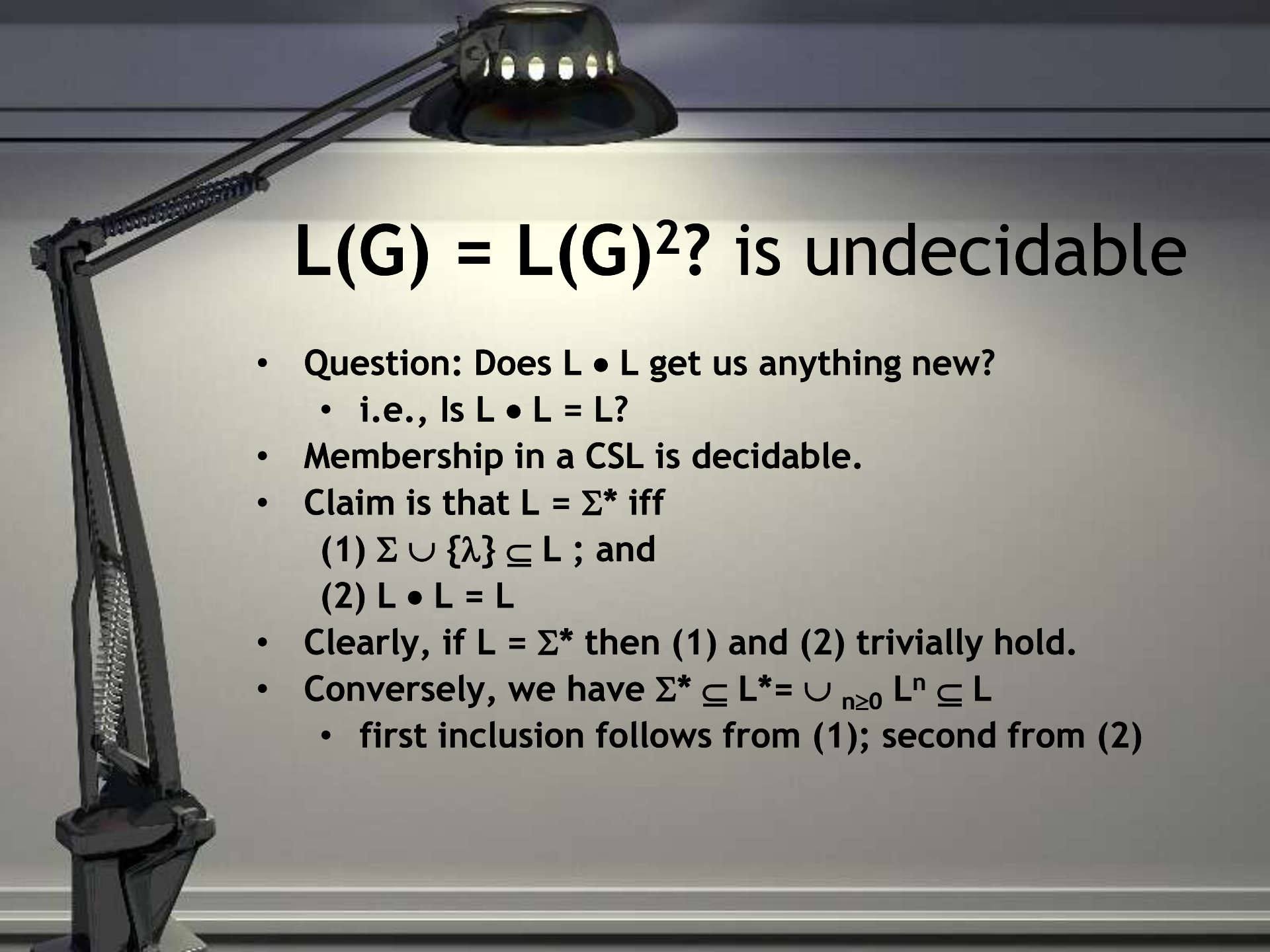
Think about why question1 is as hard as whether or not L(G) is Σ^*.

Question2 requires much more thought.



$L(G) = L(G)^2$?

- The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \bullet L \subseteq L$, so long as L is selected from a class of languages C over the alphabet Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.
- Corollary 1:
  The problem "is $L \bullet L = L$, for L context free or context sensitive?" is undecidable

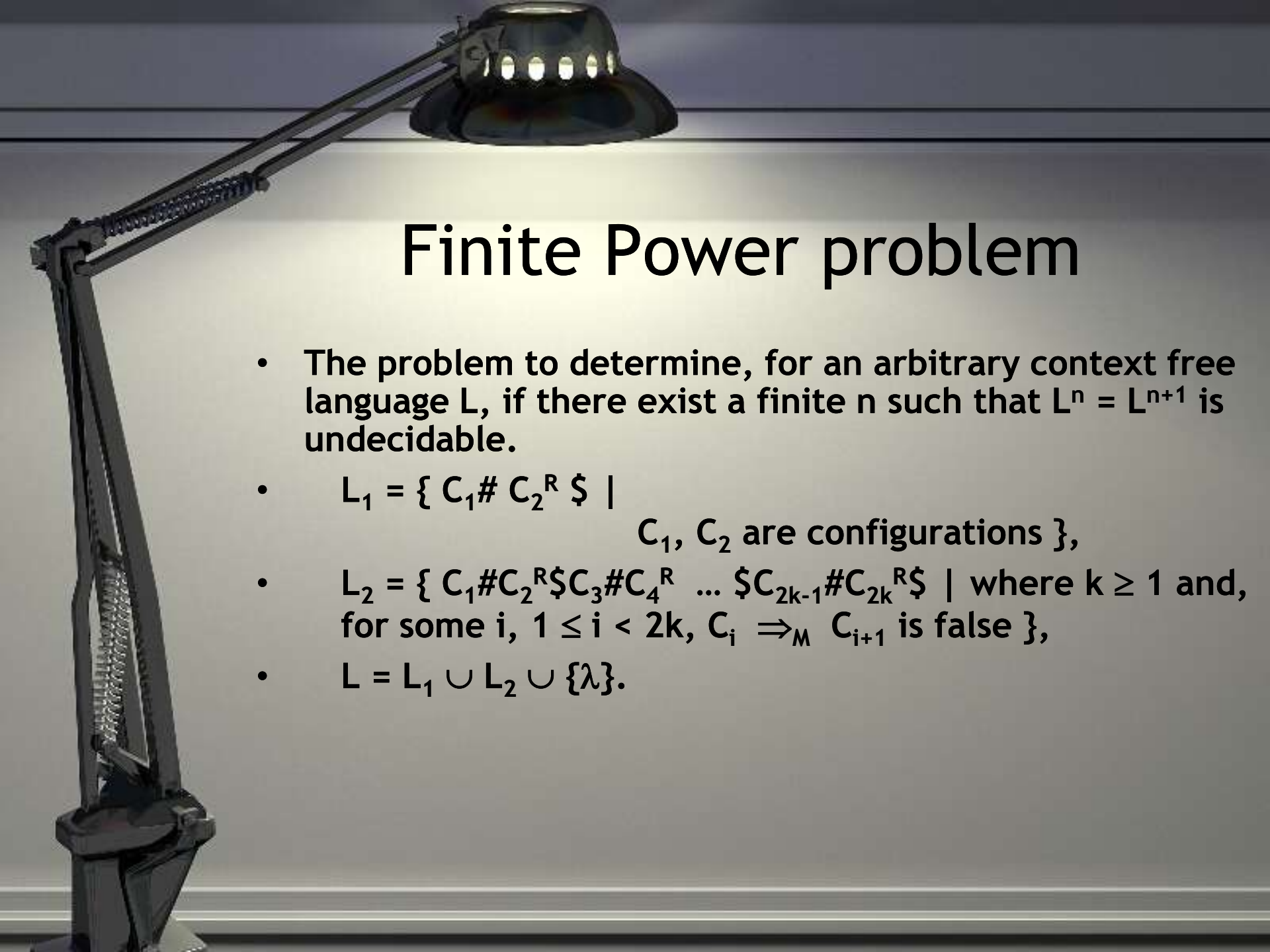


$L(G) = L(G)^2$? is undecidable

- Question: Does $L \bullet L$ get us anything new?
  - i.e., Is $L \bullet L = L$?
- Membership in a CSL is decidable.
- Claim is that $L = \Sigma^*$ iff

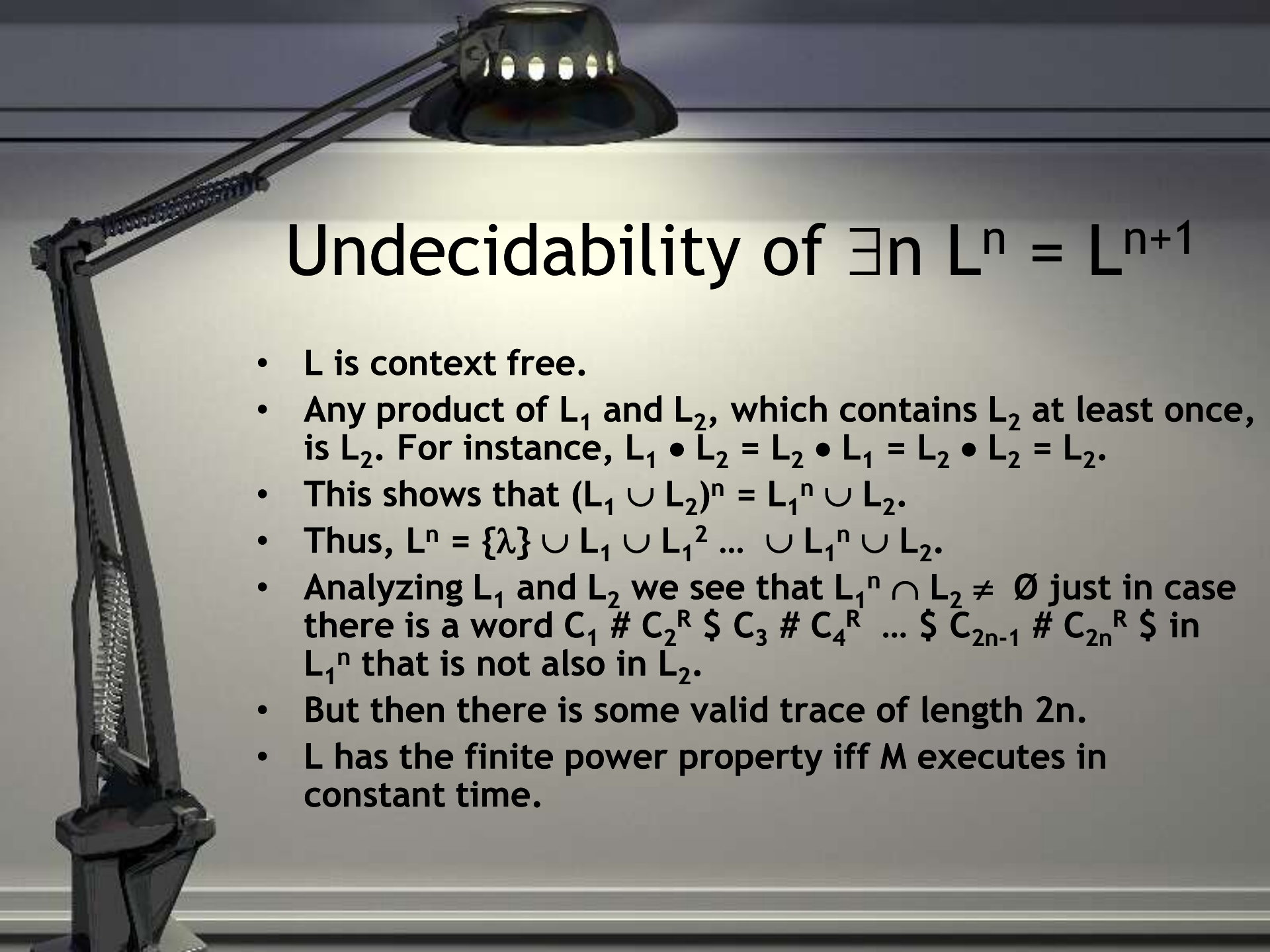
  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) $L \bullet L = L$
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n \geq 0} L^n \subseteq L$
  - first inclusion follows from (1); second from (2)



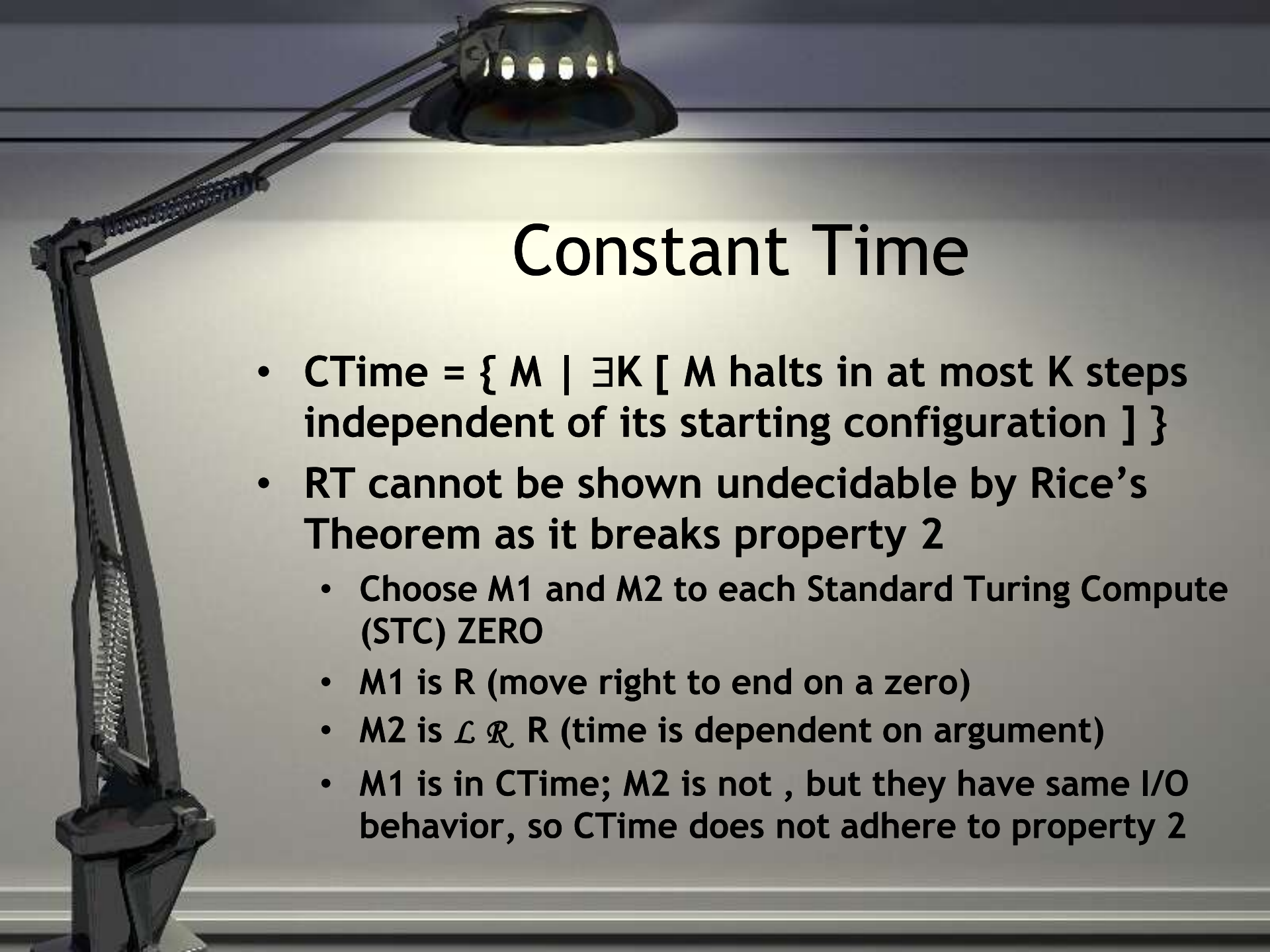
Finite Power problem

- The problem to determine, for an arbitrary context free language L, if there exist a finite n such that $L^n = L^{n+1}$ is undecidable.
- $L_1 = \{ C_1 \# C_2^R \$ \mid C_1, C_2$ are configurations $\}$,
- $L_2 = \{ C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2k-1} \# C_{2k}^R \$ \mid$ where $k \geq 1$ and, for some i, $1 \leq i < 2k$, $C_i \Rightarrow_M C_{i+1}$ is false $\}$,
- $L = L_1 \cup L_2 \cup \{\lambda\}$.



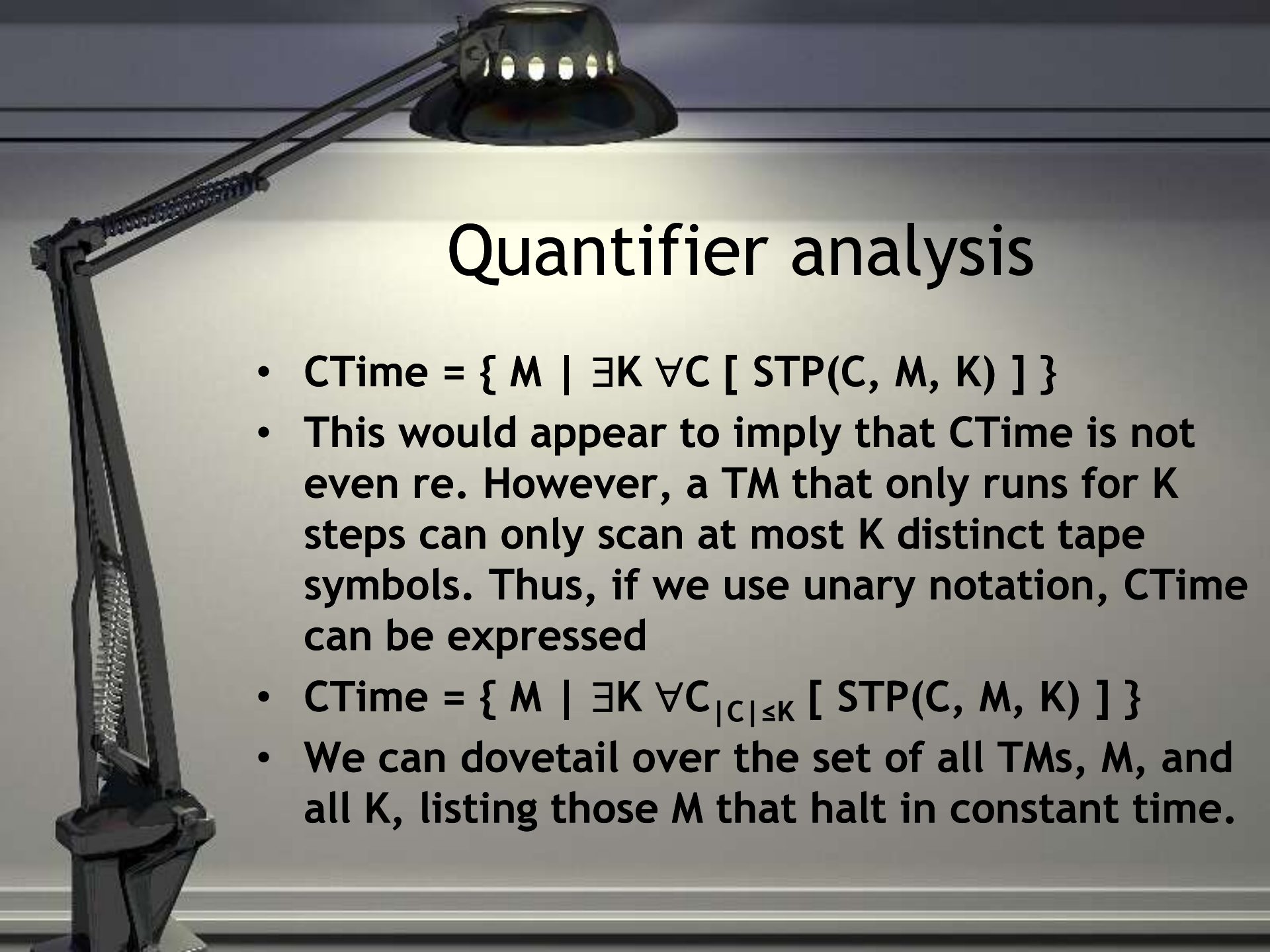
Undecidability of $\exists n\ L^n = L^{n+1}$

- L is context free.
- Any product of L_1 and L_2, which contains L_2 at least once, is L_2. For instance, $L_1 \bullet L_2 = L_2 \bullet L_1 = L_2 \bullet L_2 = L_2$.
- This shows that $(L_1 \cup L_2)^n = L_1^n \cup L_2$.
- Thus, $L^n = \{\lambda\} \cup L_1 \cup L_1^2 \ldots \cup L_1^n \cup L_2$.
- Analyzing L_1 and L_2 we see that $L_1^n \cap L_2 \neq \emptyset$ just in case there is a word $C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2n-1} \# C_{2n}^R \$$ in L_1^n that is not also in L_2.
- But then there is some valid trace of length 2n.
- L has the finite power property iff M executes in constant time.



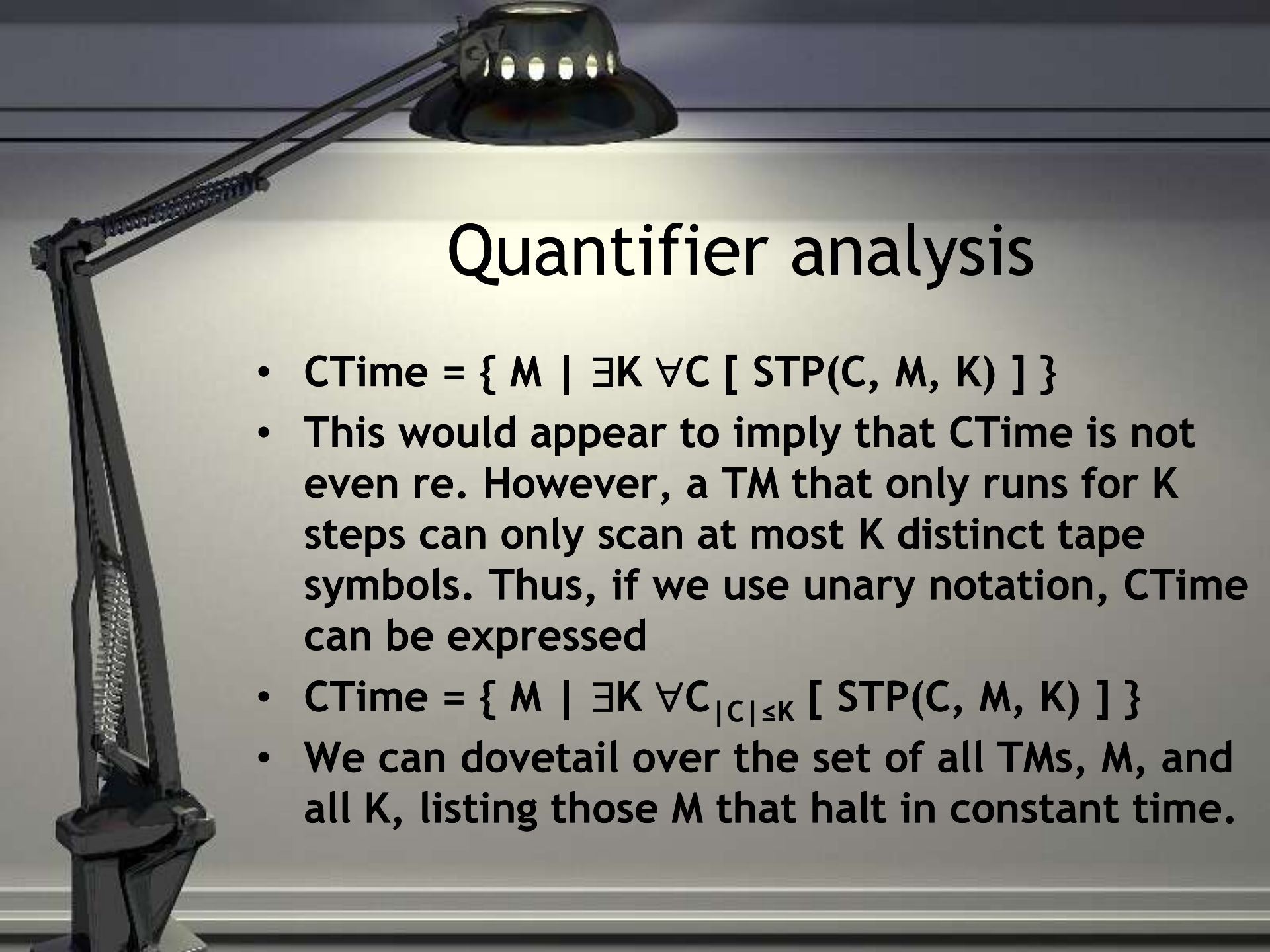
Constant Time

- CTime = { M | ∃K [M halts in at most K steps independent of its starting configuration] }
- RT cannot be shown undecidable by Rice's Theorem as it breaks property 2
  - Choose M1 and M2 to each Standard Turing Compute (STC) ZERO
  - M1 is R (move right to end on a zero)
  - M2 is $\mathcal{L}$ $\mathcal{R}$ R (time is dependent on argument)
  - M1 is in CTime; M2 is not , but they have same I/O behavior, so CTime does not adhere to property 2



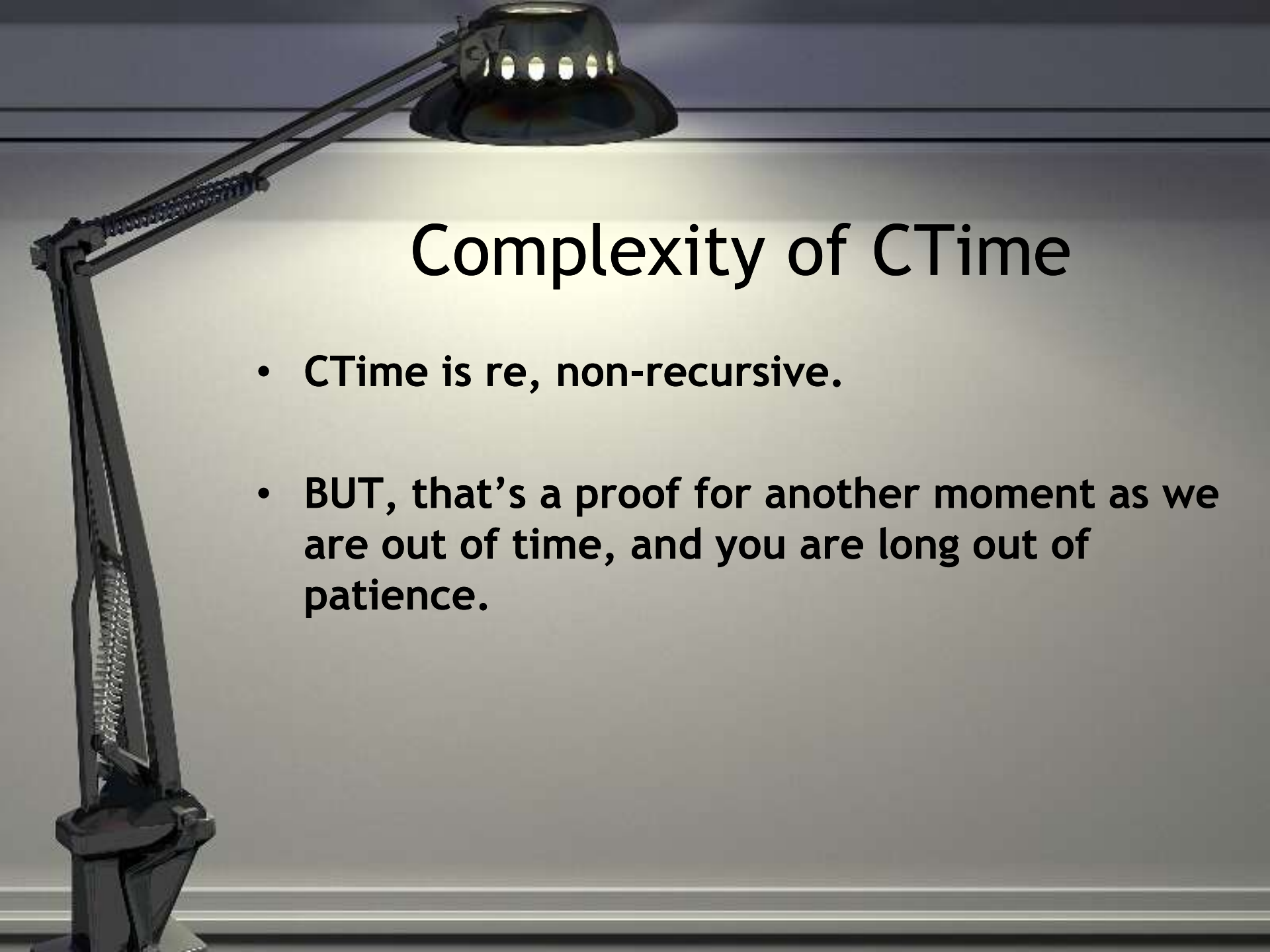
Quantifier analysis

- CTime = { M | $\exists K \, \forall C$ [STP(C, M, K)] }
- This would appear to imply that CTime is not even re. However, a TM that only runs for K steps can only scan at most K distinct tape symbols. Thus, if we use unary notation, CTime can be expressed
- CTime = { M | $\exists K \, \forall C_{|C| \leq K}$ [STP(C, M, K)] }
- We can dovetail over the set of all TMs, M, and all K, listing those M that halt in constant time.



Quantifier analysis

- CTime = { M | $\exists K \, \forall C$ [STP(C, M, K)] }
- This would appear to imply that CTime is not even re. However, a TM that only runs for K steps can only scan at most K distinct tape symbols. Thus, if we use unary notation, CTime can be expressed
- CTime = { M | $\exists K \, \forall C_{|C| \leq K}$ [STP(C, M, K)] }
- We can dovetail over the set of all TMs, M, and all K, listing those M that halt in constant time.



Complexity of CTime

- **CTime is re, non-recursive.**
- **BUT, that's a proof for another moment as we are out of time, and you are long out of patience.**