Complexity and Computability

Some fundamental things everyone should know

Pallab Dasgupta
Professor,
Dept. of Computer Sc & Engg

Transitive Closure

```
Transclosure ( int adjmat[ ][max], int path[ ][max] )
{
        for (i = 0; i < max; i++)
            for (j = 0; j < max; j++)
                path[i][j] = adjmat[i][j];

        for (k = 0; k < max; k++)
          for (i = 0; i < max; i++)
            for (j = 0; j < max; j++)
              if ((path[i][k] == 1)&&(path[k][j] == 1)) path[i][j] = 1;
}
```

How many operations are performed?

Merge-Sort

```
void mergesort ( int a[ ], int lo, int hi )          →  T(n)
{
        int m;
        if (lo<hi) {
                m=(lo+hi)/2;
                mergesort(a, lo, m);                 →  T(n/2)
                mergesort(a, m+1, hi);               →  T(n/2)
                merge(a, lo, m, hi);                 →  T(n)
        }
}
```

Complexity of Mergesort

$T(0) = 1$

$T(n) = T(n/2) + n + T(n/2)$

$= 2T(n/2) + n$

Rewrite n as 2^x:

$T(2^x) = 2T(2^{x-1}) + 2^x$

$= 2T(2T(2^{x-2}) + 2^{x-1}) + 2^x$

$= 2^2T(2^{x-2}) + 2^x + 2^x$

$= x2^x$

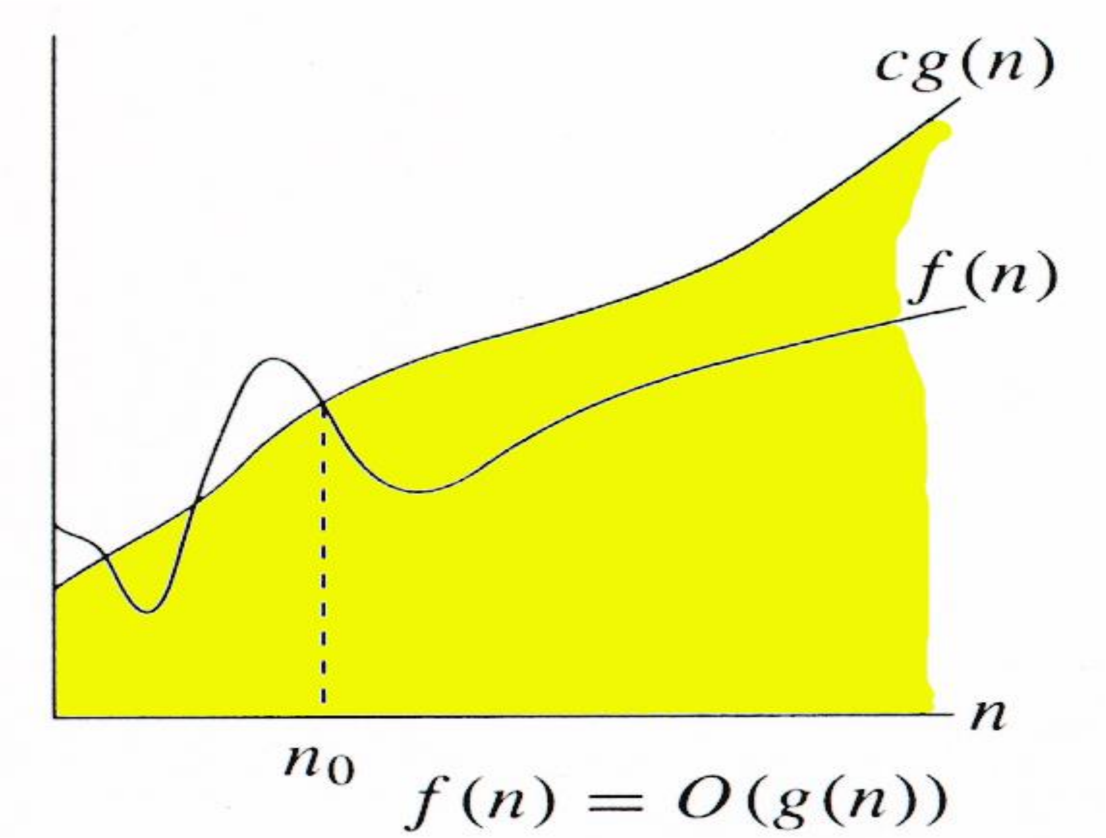
Therefore: $T(n) \cong n \log_2 n$

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$$O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0, \text{ such that } \forall n \geq n_0, \text{ we have } 0 \leq f(n) \leq cg(n) \}$$

Intuition:
Set of all functions whose rate of growth is the same as or lower than that of g(n).



***g*(*n*) is an *asymptotic upper bound* for *f*(*n*).**

Examples

$O(g(n)) = \{ f(n) : \exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq f(n) \leq cg(n) \}$

Any linear *function an* + *b* is in $O(n^2)$. How?

Show that $3n^3 = O(n^4)$ for appropriate c and n_0.

Some common notations

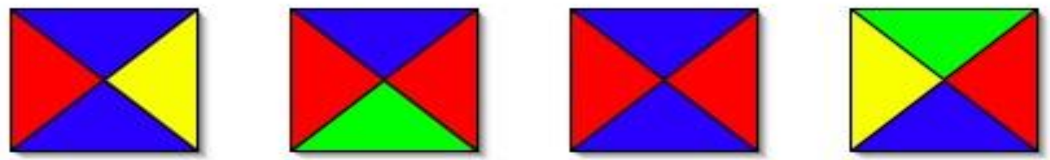
Notation	Name
O(1)	Constant
O(log n)	Logarithmic
O(n log n) = O(log n!)	Loglinear
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n)$	Exponential
O(n!)	Factorial

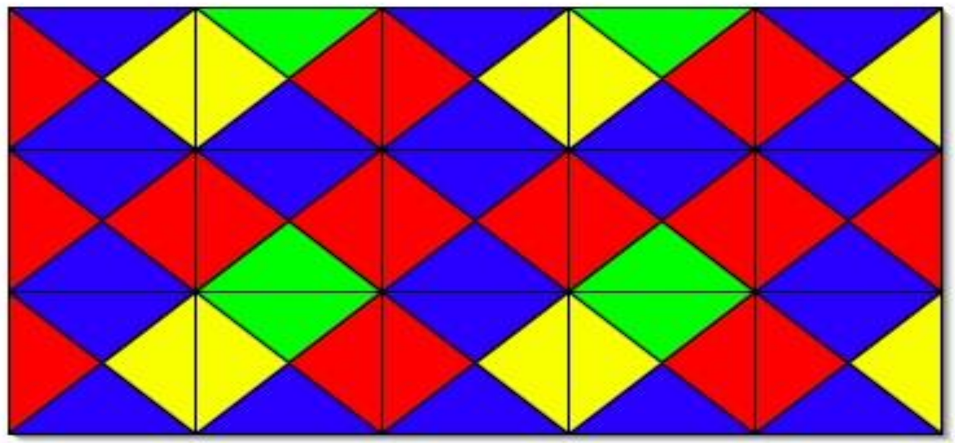
Computability

Unsolvable Problems ???

There are problems which are *unsolvable !!*

- Given a multivariate polynomial, determine whether it has an integer root
- Tiling problem:





How to *prove* that a problem is unsolvable?

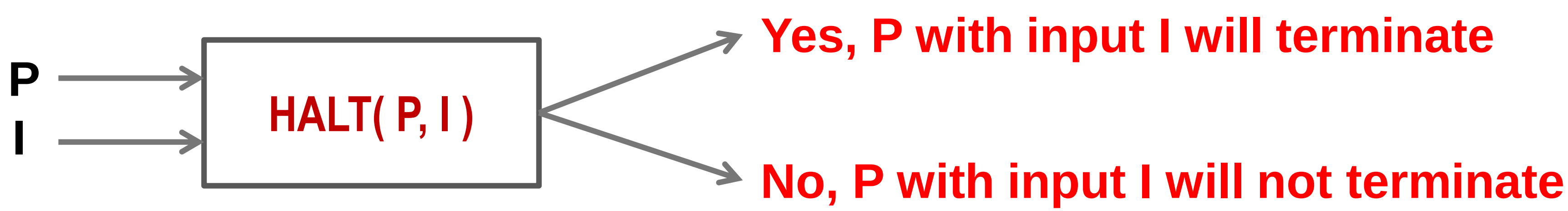
Lets start with a simple question

Have you heard of a program having another program as input?

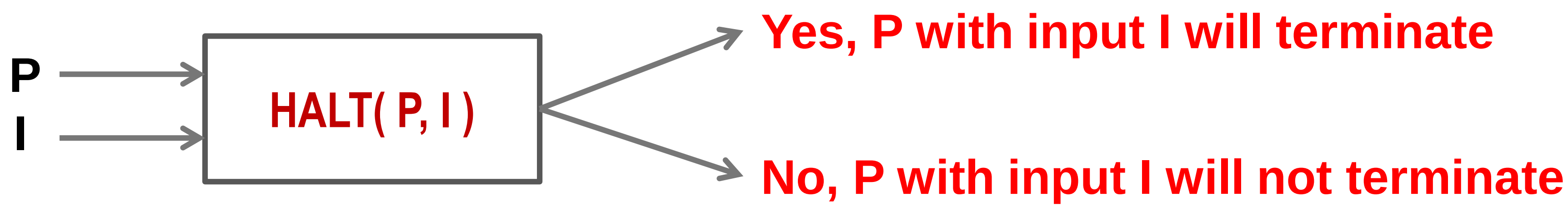
- Take for example the C compiler – it reads your program and creates the executable

You know that if you make a mistake in your program, it may not terminate.

We want to write a program that can read a program *P* and determine whether it will terminate for a given input *I*



Such a program cannot be written !!



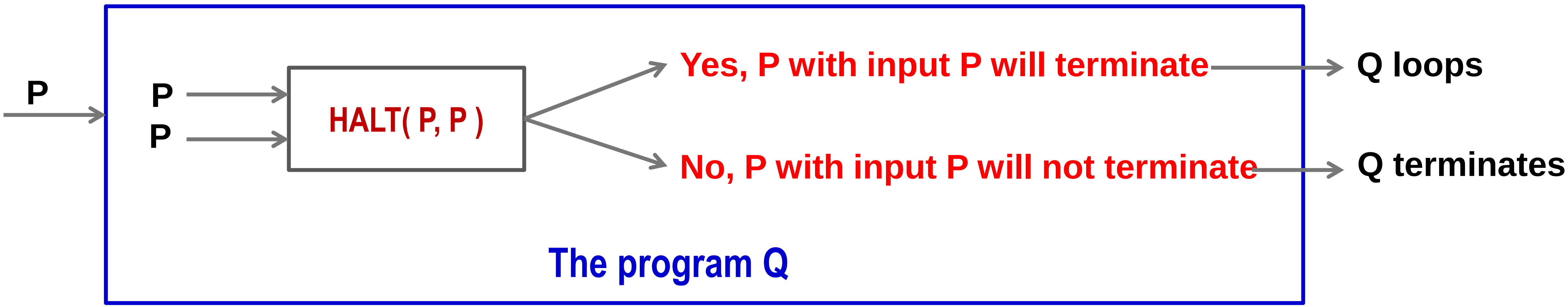
Theorem (Turing circa 1940): There is no program to solve the Halting Problem.

Proof

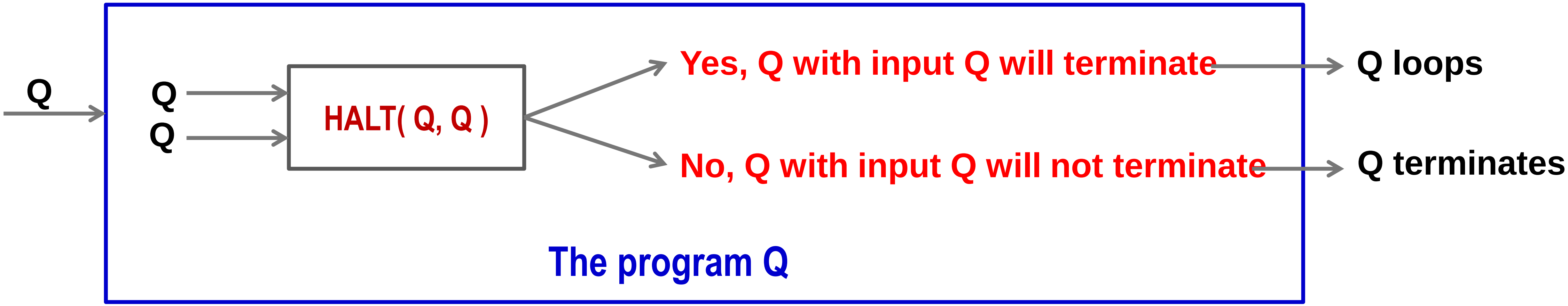
Suppose there exists a program Halt(*P*, *I*).

Then we can write a program Q that does the following:

- It runs Halt(*P, P*)
- If Halt(*P, P*) returns Yes, then it goes into an infinite loop
- If Halt (*P, P*) returns No, then it terminates.



What happens if we give Q as an input to Q?



Can you see the contradictions?

- **Q with input Q loops if HALT(Q, Q) returns Yes**
  - **but HALT(Q, Q) returns Yes only when *Q with input Q terminates* !!**
- **Q with input Q terminates if HALT(Q, Q) returns No**
  - **but HALT(Q, Q) returns No only when *Q with input Q will <u>not</u> terminate* !!**

Since HALT(Q, Q) always returns Yes/No, the contradiction is unavoidable.

Hence HALT(Q,Q) cannot exist.

How to *prove* another problem to be unsolvable?

The notion of reduction

Suppose a problem P is known to be unsolvable.

We are given a problem Q.

- Suppose we find a mechanism to transform instances of P to instances of Q
- Then we can conclude that Q too is unsolvable
  - Why?
  - Because if we had a method for solving Q, then we could solve P by converting its instances to instances of Q.

- [Example]: A *dead code* in a complex program is a line of code that is never executed. Can you show that finding dead code in a program is unsolvable in general?