Lecture 8: Unsolvability and nonfeasibility, halting problem

Conditional Execution Example

```
if (currentTemp > 80)
{
   alert(“Hot!”);
}else
{
   alert(“Okay”);
}
```

Loop at the machine level

Execution sequence: 0, 1, (2,3,4),(2,3,4)..., (2,3,4), 5

0: x = 0
1: y = 0
2: x = x + 2
3: y = y + 1
4: if (y – 10 < 0) then
jump to 2
5: halt

The task of above program is to compute 2*10 with using only the operations of additions.

- Program counter (PC) stores the **address of the instruction to be executed**.
- After an instruction has been executed, **the address in PC is increased by 1**.
- With the above settings, instructions are executed sequentially according to **their appearance ordering in the memory.**
- **Jump**: To **modify the address stored in PC**, so that change the instruction execution ordering
- **Loop:** Jump to **the address of previous executed instructions**, so that **a sequence of instructions will be executed again and again**

An Loop Example in JavaScript

```
x = 0;
finished_loop = 0; //To record the times of loop
while (finished_loop < 10) //continue the loop for 10 times
{
    x = x + 2;
    finished_loop = finished_loop + 1;  // increase the times of loop by 1
}
```

- After the loop
  - **finished_loop will be 10**, which corresponds to the times of loop
  - x will accumulate the summation of 2 10 times, namely x = 2 * 10.

Infinite loop

```
x = 0;
y = 0;
while (y < 10)
{
    x = x + 2;
    y = y + 1;
}
```

- Loop will continue with infinite many times, because
  - the loop condition control variable **y** keeps to be 0,
  - the loop condition test keeps to be true, therefore loop continue forever

Loop in a function

```
function sampleprogram(y)
{
   while (y > 0)
   {
      y = y + 1;
   }
}
```

- Will the loop in the function terminate? YES or NO, it depends on the input **y**:
  - With input y = 0, the loop terminates immediately
  - With input y > 0, the loop continues forever
  - With input y < 0, the loop terminates immediately

The Halting Problem

- Halting problem – given a **computer program** and **an input to the program**, will the **given program HALT** on the given input?
- Can we write a program to solve the halting problem? Namely, can we write a program **X** to tell whether a computer program will terminate given input **Y**?
  - **NO, WE CAN'T!**
- It is a problem **can not be solved** by computer! – It is a so-called **unsolvable problem**, or **non-computable** problem!

Why Halting Problem Unsolvable?

- Assume we can solve the halting problem, then we can write a function **will_halt**(program_X, input_Y), which
  - returns true, if program_X(input_Y) can terminate
  - return false, if program_X(input_Y) will run forever
- Some reminders about the inputs of **will_halt**
  - **program_X** is a sequence of instructions sitting in some area of the memory, essentially it is just **a sequence of 0s and 1s** just like the other data in memory**.**
  - **Input_Y** is the input data which program_X will work on**.** Since program_X is also data in memory, the setting allow the user use program_X as input to call program_X, namely the user can **set input_Y = program_X.**
- Can we have the function **will_halt**?
  - **NO!**

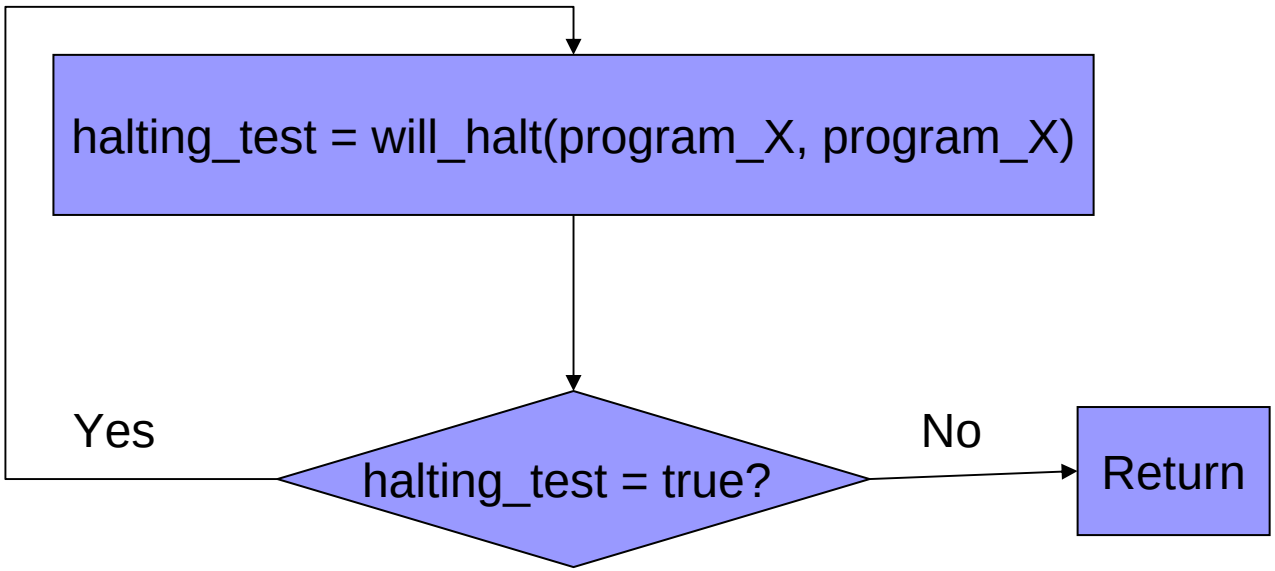
Solutions to halting problem cause contradiction!

```
function self_check(program_X)
{
   halting_test = true;
   while (halting_test == true)
   {
       halting_test = will_halt(program_X, program_X);
   }
}
```

- Contraction arises when calling **self_check(self_check) (namely, prograx_X=self_chck):**
  - If **self_check(self_check) halts**, **then** halting_test = will_halt(self_check, self_check) will be true forever, which means that the **self_check(self_check) will run forever**. **A contradiction!**
  - If self_check(self_check) doesn't halt, then halting_test = will_halt(self_check, self_check), which means that the **self_check(self_check) will terminate. Another contradiction!**
- **This means a paradox**
  - **If we can have a computer program to check whether another program halt or not, then such a halting checking program will always cause contradictory situations!**
  - **But contradictory situations can not happen in the real world, therefore we can not have a computer program to check whether another computer program halt or not!**

Diagram of contradictory situations caused by the halting problem

function **self_check**(program_X)



- self_check(self_check) halts -> (“Yes” path) continue the loop -> means run forever! **(contradiction !)**
- self_check(self_check) runs forever -> (“No” path) stop and return -> terminate!! **(contradiction!)**

What is the Capability of Computer?

- **Computability:** Is every task computable? – Is there an algorithm (series of steps) to solve a given task?
  - **NO.**
- **Complexity:** Is every computable task feasible?
  - **NO. (Too many steps or too much memory are needed!)**

Turing-Church Thesis

- *Thesis: "Every 'function which would naturally be regarded as computable' can be computed by a Turing machine."*
- The **Church–Turing thesis** (also known as the **Church's thesis**, **Church's conjecture** and **Turing's thesis**) is a hypothesis about the **nature of computers**, such as a **digital computer** or **a human** with a pencil and paper following **a set of rules**.

Computability

- A problem is **computable** if it is possible to write a computer program to solve it.
- A problem is **noncomputable** if it is impossible to write a computer program to solve it. Such a problem is also called **unsolvable**.

Nonfeasibility and Running Time

- **Non-feasible** – a computable problem that takes too long to solve (with the fastest algorithm).

Examples of feasible problems

- Adding to two integer numbers
  - several computer instructions
- Searching a list of n items
  - Need to do about **n** operations (to check the n items in the list one by one)
- Sorting a list of n items, one of many solutions:
  - Search for the smallest element, put it in the beginning (cost about **n** operations)
  - Do this n times for each i^{th} smallest element
  - Overall: about n^2 operations

Examples of non-feasible problems

- List all the possible score permutation of a class
  - Given the class size n, score are between 0-100
  - Need to do at least 100^n operations to output the result (assuming 1 operation for 1 permutation for simplicity)
  - For a class of 30 students, we at least need to do $\mathbf{10^{60}}$ operations
    - Modern computer can perform about 10^9 instructions in a second
    - There is roughly $3.15 * 10^7$ seconds per year
    - $\mathbf{10^{60}} / (3.15 * 10^7 * 10^9) > \mathbf{3 * 10^{10}}$ years!!!
- Consider all the possible moves and outcome of chess playing
- Consider every possible seating arrangement for the class
- In general, any solution that considers every possible subset of a set requires exponential steps in term of the size of the set!

A list of steps in term of input size

Given a problem's input of size n

- Requiring n (linear) operations is feasible
- Requiring n^2 (quadratic) operations is feasible
- Requiring n^c (polynomial) operations is feasible, where c is constant in spite of n
- In general, requiring less than n^c operations is feasible
- Requiring c^n (exponential: 2^n, 10^n,…) operations is non-feasible when n becomes large, because the requiring amount of operations increases rapidly as n increases.

Analysis of algorithms

- Analysis of algorithms – analyzing the running times of algorithms (or programs). This field of computer science studies algorithms and their efficiency, in terms of the amount of work (and space) needed to execute an algorithm.

Summary

- Loop
- Halting problem
- Computable and non-computable
- Feasible and non-feasible