

- Theory of Computation
 - Functions and Turing Machines
 - A non-computable function
 - Complexity of problems
 - Analysis of algorithms
 - Big-Oh (and its “relatives”)
 - Tractable vs intractable
 - Deterministic vs non-deterministic
 - P vs NP vs NP-complete

These slides were based on those of C Johnson, E McCreath and W Liang.

Theory of Computation

- What can computers do ? What can't they do ?
- Some problems are solvable by machines - some are not !
- There exist problems that no algorithmic system can solve
- There exist machine-solvable problems whose solutions are far from practical

Functions

- Mathematically speaking, a function is a correspondence between a collection of possible input values and a collection of output values such that each input is assigned one output
- e.g. conversion of Km to miles; addition; sorting
- Determining the output that a function assigns to a particular input is called *computing the function*
- A fundamental task of computer science is to find techniques for computing the functions that lie beneath a problem we want to solve

Computation

- Consider a system in which input and output can be pre-determined and recorded in a table
- To find the required output, look up the value in the table
- Computation is reduced to searching a table
- However, there is no limit on the number of possible inputs (and therefore the size of the table)

Computation

- A more powerful approach would be to provide an algebraic formula

$$y = ax^2 + bx + c$$

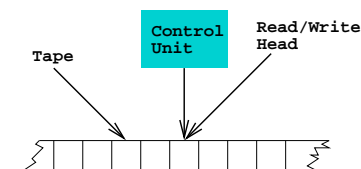
- This also has limitations
- There are functions whose input/output relationships are too complex to be described by algebraic manipulations e.g. $\sin(x)$, $\cos(x)$
- Can we always find a system for computing functions, regardless of their complexity ? (No)

Functions

- There exist functions that are so complex that there is no well-defined step-by-step process for determining output based on their input
- Their computation lies beyond the ability of any algorithmic system
- They are known as non-computable functions (as opposed to computable functions)
- The study of computable functions is the study of the ultimate capabilities of machines

Turing Machines

- Turing machines consist of a control unit that is able to read/write symbols from/to a tape
- The tape extends indefinitely at both ends
- The tape is divided into cells
- Each cell can contain any one of a finite set of symbols (language)

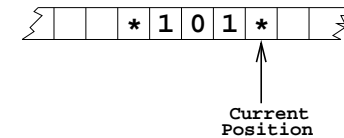


Turing Machines

- At any time the machine must be in any one of a finite number of conditions (states)
- The machine begins in the *start state*
- The machine ceases when it reaches the *halt state*
- The machine's computation consists of a sequence of steps that are executed by the control unit
- Each step consists of:
 - observing the symbol in the current cell
 - writing a symbol to that cell
 - possibly moving one cell to the left or right
 - changing state

A Specific Turing Machine

- The alphabet consists of 0, 1 and *
- The tape may appear as follows:



- A string of symbols on the tape represents binary numbers separated by asterisks
- The machine is designed to increment their values by 1

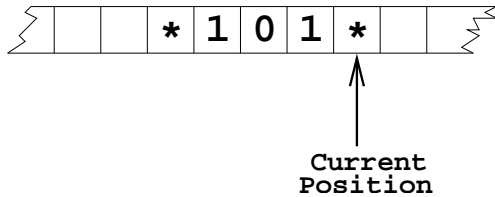
A Specific Turing Machine

- The states of the machine are
 - START
 - ADD
 - CARRY
 - RETURN
 - OVERFLOW
 - HALT

Current State	Current Content	Write Value	Direction to move	New State
START	*	*	L	ADD
ADD	0	1	R	RETURN
ADD	1	0	L	CARRY
ADD	*	*	R	HALT
CARRY	0	1	R	RETURN
CARRY	1	0	L	CARRY
CARRY	*	1	L	OVERFLOW
OVERFLOW	*	*	R	RETURN
RETURN	0	0	R	RETURN
RETURN	1	1	R	RETURN
RETURN	*	*	None	HALT

A Specific Turing Machine - Example

Machine State = START

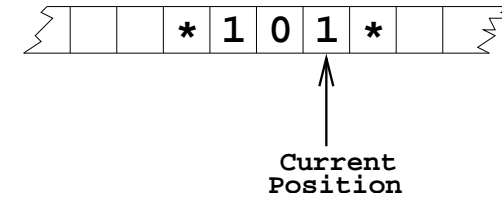


Current State	Current Content	Write Value	Direction to move	New State
---------------	-----------------	-------------	-------------------	-----------

START	*	*	L	ADD
-------	---	---	---	-----

A Specific Turing Machine - Example

Machine State = ADD

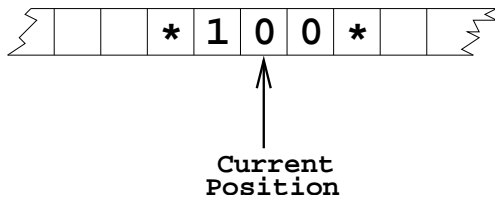


Current State	Current Content	Write Value	Direction to move	New State
---------------	-----------------	-------------	-------------------	-----------

ADD	1	0	L	CARRY
-----	---	---	---	-------

A Specific Turing Machine - Example

Machine State = CARRY

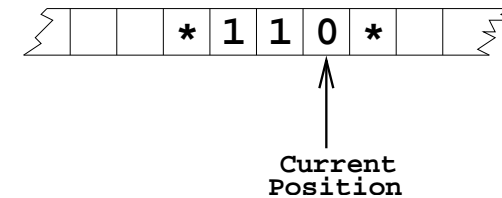


Current State	Current Content	Write Value	Direction to move	New State
---------------	-----------------	-------------	-------------------	-----------

CARRY	0	1	R	RETURN
-------	---	---	---	--------

A Specific Turing Machine - Example

Machine State = RETURN

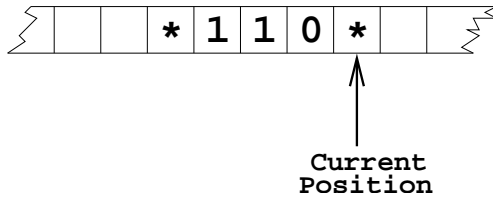


Current State	Current Content	Write Value	Direction to move	New State
---------------	-----------------	-------------	-------------------	-----------

RETURN	0	0	R	RETURN
--------	---	---	---	--------

A Specific Turing Machine - Example

Machine State = RETURN



Current State	Current Content	Write Value	Direction to move	New State
---------------	-----------------	-------------	-------------------	-----------

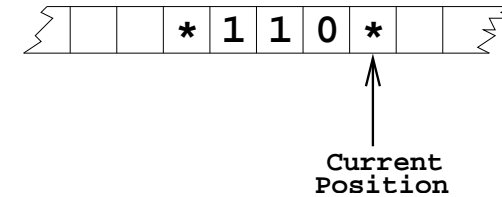
RETURN	*	*	None	HALT
--------	---	---	------	------

The Church-Turing Thesis

- The previous example can be used to compute the successor function
- Assign each non-negative input value n the output value $n + 1$
- A function that may be computed in this manner is said to be *Turing computable*
- Turing conjectured that computable and Turing computable were one and the same

A Specific Turing Machine - Example

Machine State = HALT



The Church-Turing Thesis

- The Turing machine concept provides a framework in which all computable functions can be expressed
- Referred to as the Church-Turing thesis (Alan Turing and Alonzo Church) and is widely accepted
- This is significant as it gives insight into the capabilities and the limitations of computing machinery

The Halting Problem

- Informally - The problem of predicting in advance whether a program will halt if started under certain conditions
- **while** ($x! = 0$) **do** ($x \leftarrow x + 1$)
- What if $x = 0$?
- What if $x > 0$?
- Whether a program ultimately halts depends on the initial value of its variables

Self-Reference

- The idea of an object referring to itself
- If it does; then it doesn't!
- If it doesn't; then it does!
- Assign variables in a program an initial value that represents the program itself
- A program is an ASCII string which can be represented in binary form as a (very large) non-negative number
- This number is to be the initial value for the variables

Self-Reference

- **while** ($x! = 0$) **do** ($x \leftarrow x + 1$)
- What would happen if this program was started with x assigned to be the integer value of the program itself?
- $x \leftarrow 0$ **while** ($x! = 0$) **do** ($x \leftarrow x + 1$)
- What would happen if this program was started with x assigned to be the integer value of the program itself?

The Halting Problem

- A program is *self-terminating* if executing the program with all of its variables initialised to the program's own encoded representation leads to a terminating process
- Whether a program is self-terminating or not has nothing to do with why the program was written
- The halting problem is the problem of deciding whether or not a program is self-terminating

The Halting Problem

- There is no algorithm for answering this question in general
- The solution to the halting problem lies beyond the capabilities of computers
- The halting problem requests a single, generic algorithm that can be applied to any program to determine self-termination

Unsolvability of the Halting Problem

- Need to show that solving the halting problem would require solving a non-computable function
- Inputs to the function are encoded versions of programs
- Output of the function is either 0 (not self-terminating) or 1 (self-terminating)
- Refer to this function as the *halting function*

Unsolvability of the Halting Problem

- How do we show that the halting function is not computable?
- Use proof by contradiction
 - Prove something to be false by showing it cannot be true
- Try to show that the statement "*The halting function is computable*" cannot be true

Unsolvability of the Halting Problem

- If the halting function is computable, there must be a program that computes it
- i.e. There must exist a program that terminates with
 - output=1 if its input is the encoded version of a self-terminating program
 - output=0 if its input is the encoded version of a non self-terminating program

Unsolvability of the Halting Problem

- Assume the program's output is X
- Add the following code to the end of the program
 - **while** ($x! = 0$) **do** (;)
- This produces a new program
- This new program is either self-terminating or it isn't
- In fact it's neither

Unsolvability of the Halting Problem

1. Proposition

Given any encoded version of a program

proposed program

halts with $X=1$ if input represents a self-terminating program
 $X=0$ otherwise

Unsolvability of the Halting Problem

2. Modify

proposed program
<pre>while (X!=0) do (;)</pre>

Unsolvability of the Halting Problem

3. If new program was self-terminating

Start with its own encoding as input

proposed program

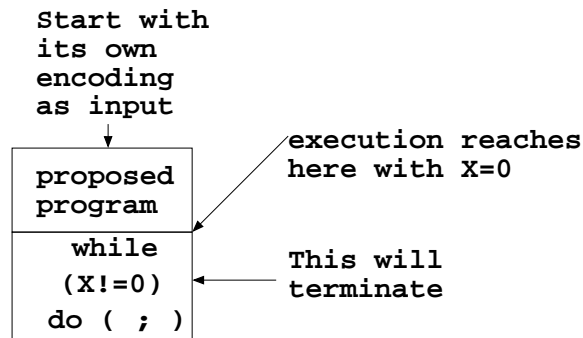
```
while (X!=0) do ( ; )
```

execution reaches here with $X=1$

This will not terminate

Unsolvability of the Halting Problem

4. If new program was not self-terminating

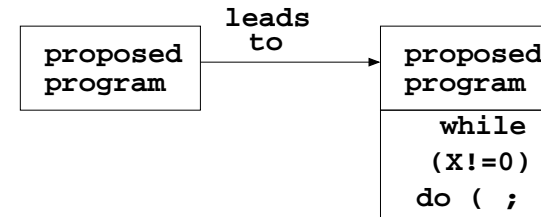


Unsolvability of the Halting Problem

5: Consequently

The existence of the proposed program

The existence of a new program



That is neither self-terminating or not self-terminating

so the existence of the proposed program is impossible

Unsolvability of the Halting Problem

- We have an impossible situation
- A program must be self-terminating or not
- Whereas our program is neither
- The assumption that lead to this must therefore be false
- The halting function is therefore not computable
- This problem lies beyond the capabilities of any algorithmic system
- An example of an *unsolvable problem*

Analysis of Algorithms

- Experimental analysis is valuable, but has limitations
- We could sum up the time it takes for each instruction executed but this would be complicated
- We restrict ourselves to primitive operations, such as, assigning a value to a variable or calling a function
- We then simply count how many primitive operations are performed and use this as a high-level estimate of the running time of the algorithm

ListMax

```
procedure ListMax(List)
(CurrentMax ← first element in List
NextItem ← second element in List
while (there are items still to be examined in List)
do [ if (NextItem > CurrentMax)
      then ( CurrentMax ← NextItem; )
      NextItem ← next item in List;
    ]
)
```

Average-Case and Worst-Case Analysis

- An algorithm may run faster on some inputs rather than others
- We may wish to express the running time of such an algorithm as an average taken over all possible inputs
- This is typically quite challenging
- Need to define a probability distribution on the set of inputs
- Calculate expected running time based on this distribution
- Requires heavy mathematics and probability theory

Why Does it Matter ?

Running times on 10^9 instructions/second computer

n	n	$n \log n$	n^2	n^3	2^n
10	1×10^{-8} s	3×10^{-8} s	1×10^{-7} s	1×10^{-6} s	1×10^{-6} s
50	5×10^{-8} s	3×10^{-7} s	3×10^{-6} s	1×10^{-4} s	13 days
100	1×10^{-7} s	7×10^{-5} s	1×10^{-5} s	1×10^{-3} s	1013 yrs
10^3	1×10^{-6} s	1×10^{-5} s	1×10^{-3} s	1 s	10283 yrs
10^4	1×10^{-5} s	1×10^{-4} s	0.1 s	17 min	??
10^5	1×10^{-4} s	2×10^{-3} s	10 s	12 days	??
10^6	1×10^{-3} s	2×10^{-2} s	17 min	32 yrs	??

Why Does it Matter ?

- Consider the following computer:
 - It is a cube with side H containing many cubic cells or processors with side h
 - These processors work in parallel and exchange their result after each step

Why Does it Matter ?

Computer's Size:

$$H = 1\text{km} = 10^3\text{m}$$

Cell's Size:

$$h = 10^{-8}\text{cm} = 10^{-10}\text{m}$$

Time per Operation:

$$t = 0\text{s}$$

Exchange Speed:

$$V = c = 300000\text{km/s} = 3 \times 10^8\text{m/s}$$

Number of Cells (Processors):

$$N = (H/h)^3 = 10^{39}$$

Why Does it Matter ?

Time per Step:

$$\begin{aligned} T &= \text{Time per operation} + \text{Exchange time} \\ &= 0 + H/V \sim 0.33 \times 10^{-5}\text{s} \end{aligned}$$

Number of Operations:

$$M = N/T = 3 \times 10^{44}\text{per Second}$$

(currently we have about 10^9)

Now consider an $n \times n$ -matrix A with determinant $\det(A)$

Such a determinant contains $n!$ different terms

So, computing $\det(A)$ "by the definition" means we must perform at least $n!$ operations

Why Does it Matter ?

Let $n = 100$

Then the number of operations is:

$$Q = 100! > 30^{100} > 3 \times 10^{100}$$

Here we use the known inequality:

$$n! > (n/e)^n > (n/3)^n$$

So, our computer will need more than:

$$Q/M \geq 3 \times 10^{100} / 3 \times 10^{44} = 10^{56}\text{s} > 10^{48}\text{years}$$

The "Big-Oh" Notation

- Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers
- We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n' \geq 1$, such that $f(n) \leq c \cdot g(n)$ for every integer $n \geq n'$
- We say $f(n)$ is big-Oh of $g(n)$
- Some people say $f(n)$ is order $g(n)$

The “Big-Oh” Notation

- If $f(n)$ is a function of n then we write

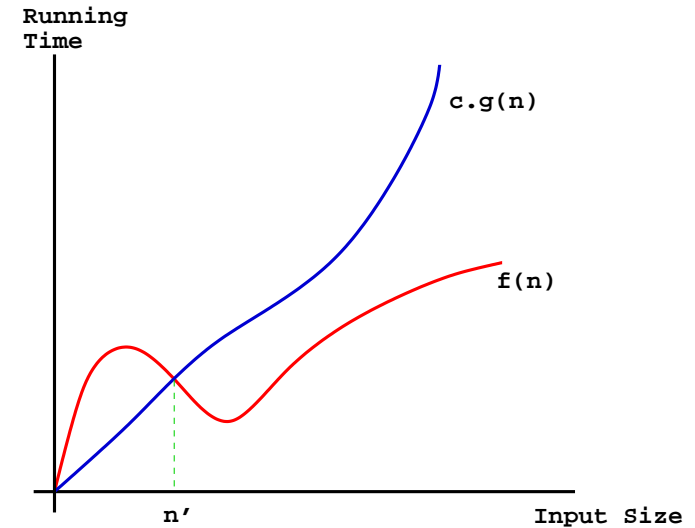
$$f(n) \text{ is } O(g(n))$$

- with the intention that, for some constant, c ,

$$|f(n)| \leq c \cdot |g(n)|$$

- as $n \rightarrow \infty$
- So, $O(g(n))$ is an asymptotic upper bound

The “Big-Oh” Notation



The “Big-Oh” Notation

- $7n - 2$ is $O(n)$
- We need to find a real constant $c > 0$ and an integer constant $n' \geq 1$, such that $7n - 2 \leq c \cdot n$ for every integer $n \geq n'$
- Choose $c = 7$ and $n' = 1$
- Any real number ≥ 7 will work for c here
- and any integer ≥ 1 will work for n'

The “Big-Oh” Notation

- $20n^3 + 10n \log(n) + 5$ is $O(n^3)$

$$20n^3 + 10n \log(n) + 5 \leq 35n^3 \text{ for } n \geq 1$$

- $5/n$ is $O(1/n)$

$$5/n \leq 5(1/n), \text{ for } n \geq 1$$

- 2^{100} is $O(1)$

$$2^{100} \leq 2^{100} \cdot 1, \text{ for } n \geq 1$$

The “Big-Omega” Notation

- Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers
- We say that $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$;
- i.e., there is a real constant $c > 0$ and an integer constant $n' \geq 1$, such that $f(n) \geq c \cdot g(n)$ for every integer $n \geq n'$
- We say $f(n)$ is big-Omega of $g(n)$

The “Big-Omega” Notation

- Sometimes we need lower bounds, so we may write

$$f(n) \text{ is } \Omega(g(n))$$

- when there is a constant, c , such that

$$|f(n)| \geq c \cdot |g(n)|$$

- as $n \rightarrow \infty$
- So, $\Omega(g(n))$ is an asymptotic lower bound

The “Big-Theta” Notation

- Let $f(n)$ and $g(n)$ be functions mapping non-negative integers to real numbers
- We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$;
- i.e., there are real constants $c' > 0$ and $c'' > 0$ and integer constant $n' \geq 1$, such that

$$c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$$

- for every integer $n \geq n'$
- We say $f(n)$ is big-Theta of $g(n)$

An Example

- Let $f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \dots + a_mn^m$, then

$$f(n) = O(n^m)$$

$$f(n) = \Omega(n^m)$$

$$f(n) = \Theta(n^m)$$

- It would have also been correct to say

$$f(n) = O(n^{m+17})$$

$$f(n) = \Omega(1)$$

Manipulations

- Some useful formulae are

$$O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$$

$$c \cdot O(f(n)) = O(f(n))$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

- Asymptotically all logarithmic behaviour is the same

$$\log_2(n) = O(\ln(n))$$

$$\ln(n) = O(\log_2(n))$$

- So we may as well just write “log”
- Note $\log(n^c) = c \cdot \log(n)$ so $\log(n^c) = O(\log(n))$

An Example

- Consider the following:

```
i ← 0;
while ( i < n ) do (
    j ← i;
    while ( j < n )
    do (
        j ← j + 1;
    )
    i ← i + 1;
)
```

- How many times does the $j \leftarrow j + 1$ statement get executed?

An Example

- How many times does the $j \leftarrow j + 1$ statement get executed?

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1$$

- What is the order of growth of this sum ?

$$\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-1} (n - i) = \sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$$

Comparison of Time Efficiency of Algorithms

- Evaluate $(-1)^n$ where $n \geq 0$

```
procedure f(n)
[
    result ← -1;
    i ← 1;
    while ( i < n ) do ( result = result * -1;
        i ← i + 1;)
]
```

Comparison of Time Efficiency of Algorithms

- Evaluate $(-1)^n$ where $n \geq 0$

```
procedure f(n)
[
  if ( n mod 2 is 0 ) result  $\leftarrow$  1;)
  else           result  $\leftarrow$  -1;)
]
```

Comparison of Time Efficiency

- To compare efficiency of 2 algorithms independent of implementation
- To predict the growth in run-time as n increases
- How many times does most-nested statement iterate (as function of n)
- Say we have 2 algorithms A and B for solving a problem

```
Algorithm A           Algorithm B
  i  $\leftarrow$  1         i  $\leftarrow$  1
  while ( i < n ) do  while ( i < n ) do
    ( ... )             ( j  $\leftarrow$  1; while ( j < n ) do
                       ( ... ) )
```

Examples

```
i  $\leftarrow$  1
while ( i < n )
do [ j  $\leftarrow$  1;
     while ( j < n )
     do ( ...
         j  $\leftarrow$  j + 1
       )
     ...
     i  $\leftarrow$  i + 1
  ]
```

$$T(n) = O(n^2)$$

Examples

```
i  $\leftarrow$  1
while ( i < n )
do [ ...
     i  $\leftarrow$  i * 2
  ]
```

$$T(n) = O(\log n)$$

Examples

```
i ← n
while ( i > 0 )
do [ j ← n;
    while ( j > 1 )
    do ( ...
        j ← j - 1;
    )
    ...
    i ← i/2
]
```

$$T(n) = O(n \log n)$$

Examples

```
procedure f(n)
(
  if (...)
  then var ← ...;
  else var ← c · f(n - 1);
)
```

$$T(n) = O(n)$$

Examples

```
procedure f(List)
(
  if (...)
  then var ← ...;
  else if var ← f(some sublist of List);
  else var ← f(some other sublist of List);
)
```

$$T(n) = O(\log n)$$

Examples

```
procedure f(List)
(
  if (...)
  then var ← ...;
  else (
    List1 ← f(half of List);
    List2 ← f(other half of List);
    combine List1 and List2;
  )
)
```

$$T(n) = O(n \log n) \text{ if combining is } O(n)$$

Examples

```
procedure f(n)
(
  if (...)
  then var = ...;
  else var = f(n/2);
)
```

$$T(n) = O(\log n)$$

Examples

```
procedure f(List)
(
  if (...)
  then var ← ...;
  else var ← f(some sublist of List) ...
           f(some other sublist of List) ...
           ...
           f(some other sublist of List);
)
```

$$T(n) = O(2^n)$$