Artificial Intelligence (AI)

AI – The History
- AI is as old as computing, whose theory started in the 1930 with Alan Turing, Alonzo Church, and others
- 1941 Konrad Zuse, Germany, general purpose computer
- 1943 Britain (Turing and others) Colossus, for decoding
- 1945 ENIAC, US. John von Neumann a consultant
- 1956 Dartmouth Conference organized by John McCarthy (inventor of LISP)
- The term Artificial Intelligence was coined at Dartmouth, which was intended as a two month study.

AI – The Achievements
- Computers land 200 ton jumbo jets unaided every few minutes.
- Search systems like Google are not perfect but provide very effective information retrieval.
- Robots cut slots for hip joints better than surgeons.
- The chess program Deep Blue beat world champion Kasparov in 1997.
- Medical expert systems can outperform doctors in many areas of diagnosis
- Self-driving cars are beginning to enter the market.
- IBM’s Watson beats humans at Jeopardy.
- Programs such as Siri communicate via natural language.

Artificial vs. Human Intelligence
- Today’s computers can do many well-defined tasks (for example, arithmetic operations), much faster and more accurate than human beings.
- However, the computers’ interaction with their environment is not very sophisticated yet.
- How can we test whether a computer has reached the general intelligence level of a human being?

Turing Test: Can a computer convince a human interrogator that it is a human?

But before thinking of such advanced kinds of machines, we will start developing our own extremely simple “intelligent” machines.

Why AI?
- One of major divisions in AI (and you can see it in the definitions on the previous slide) is between
- Those who think AI is the only serious way of finding out how we work (since opening heads does not yet give much insight into this) and
- Those who want computers to do very smart things, independently of how we work.

This is the important distinction between Cognitive Scientists vs. Engineers.

Symbolism vs. Connectionism
- There is another major division in the field of Artificial Intelligence:
  - **Symbolic AI** represents information through symbols and their relationships. Specific Algorithms are used to process these symbols to solve problems or deduce new knowledge.
  - **Connectionist AI** represents information in a distributed, less explicit form within a network. Biological processes underlying learning, task performance, and problem solving are imitated.
Paradigms of Computation

You all know the Turing machine, conceived by Alan Turing as a theoretical Model of automatic computation. It uses a tape head that reads and writes symbols on an infinite tape. Based on the currently read symbol and the machine’s current state, the head moves to the left or right or writes a new symbol, and the state is updated. These state transition rules constitute the program. It is believed (but has not been proven) that this machine can compute all functions that can be computed in principle.

Turing Machines

Turing machines inspired the construction of the first computers, which were based on the von-Neumann architecture. Here, digital memory stores the program and data, including the machine state. A Central Processing Unit (CPU) sequentially executes individual instructions in the program through memory read and write operations. This fundamental architecture is still shared by most of today’s computers.

Imperative Programming

This architecture is also reflected in most modern programming languages such as Java, C, C++, C#, Python, or Matlab. Their programs consist of sequences of instructions, each of which changes the system’s state, such as the values of variables or other memory content. Such languages are called imperative languages. Object-oriented programming provides mechanisms for encapsulation of functional program and data units but is still based on the imperative paradigm.

Lambda (λ) Calculus

Roughly at the same time when Turing developed his Turing machine, Alonzo Church devised a different paradigm of computation, called lambda calculus. It is based on anonymous functions described by lambda expressions. By mechanisms such as composition and recursion, lambda expressions can represent complex computations. It can be shown that Turing machines and lambda calculus have identical computational power, which is believed to be universal (Church-Turing thesis, 1937).

Functional Programming

The most striking feature of purely functional programming is that there is no state. This means that our variables are not variable, i.e., cannot change their values! In other words, they are immutable and only represent some constant value. The execution of a program only involves the evaluation of functions. This sounds weird – what are the advantages and disadvantages of functional programming?
Functional Programming
The advantage of having no state is that functions have no side effects.
Therefore, we can be sure that whenever we evaluate a function with the same inputs, we will get the same output, and nothing in our system changed due to this evaluation.
This prevents most of the bugs that commonly occur in imperative programming.
You will learn about other advantages during the next few lectures…

Haskell
In this course, we will use Haskell, because its purity forces you to use functional programming principles.
Get the latest Haskell Platform at Haskell.org.
Free Haskell tutorials:
http://learnyouahaskell.com/
http://book.realworldhaskell.org/
I recommend that you read Chapters 1 and 2 of “Learn you a Haskell” and experiment with the language a bit.

Haskell
Download the Haskell Platform at:
https://www.haskell.org/
There are no perfect Haskell IDEs, but some good plugins for editors such as Atom, Sublime Text, or IntelliJ.
For our purposes, a simple text editor plus console is good enough.
For Windows, Notepad++ with the NppExec plugin gives you syntax highlighting and integrated coding, testing, and profiling.

Example scripts for Notepad++:
Compiling:
ghc -O -o main.exe $(FULL_CURRENT_PATH)
cmd /C main.exe
Interactive testing/debugging:
ghci $(FULL_CURRENT_PATH)
Profiling:
ghc -O -prof -auto-all $(FULL_CURRENT_PATH)
$(NAME_PART) +RTS -p -RTS
cmd /C more $(NAME_PART).prof
Demo Session (I)

Here is the protocol of our demo session with GHCi:

*Main> 3 + 4
7
*Main> 2^1000
1071508607186267320948425049060001810561404811705533 607443750388370351051124931224931983788156958581275 9467291755314682518714528569231404359845775746985748 03934567774824230985421074609506237114187795418215304 647498358194126739876755916543946077062914571196477 686542167660429831652624386837205668069376
*Main> "hello"
"hello"

Demo Session (II)

*Main> :t "hello"
"hello" :: [Char]
*Main> :t 3
3 :: Num a => a
*Main> :t 3.5
3.5 :: Fractional a => a
*Main> [3, 2, 1]
[3,2,1]
*Main> 4:3:2:1
[4,3,2,1]
*Main> 4:3:5:[]
[4,3,5]

Demo Session (III)

*Main> (2, 3)
(2,3)
*Main> (2, 3) == (3, 2)
False
*Main> [2, 3] == [3, 2]
False
*Main> (5, 'a')
(5,'a')
*Main> :t head
head :: [a] -> a
*Main> head [4, 6, 1]
4

Demo Session (IV)

*Main> :t tail
tail :: [a] -> [a]
*Main> tail [4, 6, 1]
[6,1]
*Main> let mult a b = a*b
*Main> mult 6 7
42
*Main> :t map
map :: (a -> b) -> [a] -> [b]
*Main> map even [1..10]
[False,True,False,True,False,True,False,True,False,True]

Demo Session (V)

*Main> (mult 6) 8
48
*Main> let supermult = mult 6
*Main> supermult 5
30
*Main> :t supermult
supermult :: Num a => a -> a
*Main> even 7
False
*Main> :t filter
filter :: (a -> Bool) -> [a] -> [a]

Demo Session (VI)

*Main> :t even
even :: Integral a => a -> Bool
*Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
*Main> filter even [1..10]
[2,4,6,8,10]
*Main> :t map
map :: (a -> b) -> [a] -> [b]
*Main> map even [1..10]
[False,True,False,True,False,True,False,True,False,True]