CSCI2100 Data Structures
Introduction

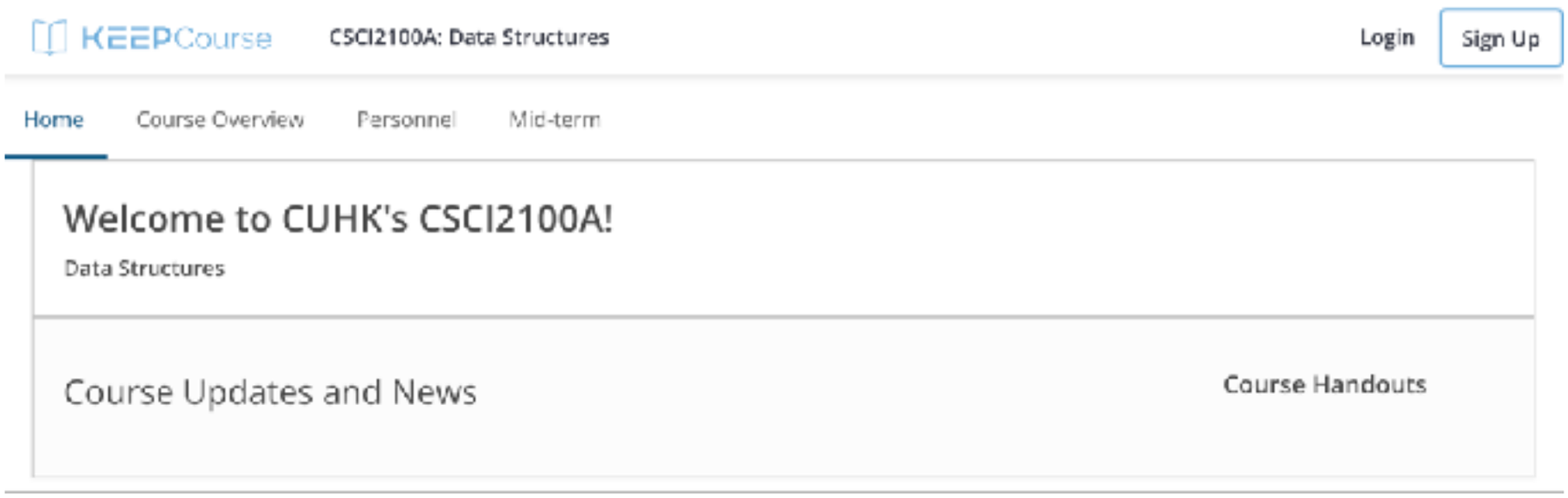
Irwin King

king@cse.cuhk.edu.hk
http://www.cse.cuhk.edu.hk/~king

Department of Computer Science & Engineering
The Chinese University of Hong Kong

Course Information

- CSCI2100A Data Structures in 2018-19 Term 2
- https://edx.keep.edu.hk/courses/course-v1:CUHK+CSCI2100A+2018_02/info



The Golden Rule of CSCI2100

No member of the CSCI2100 community shall take unfair advantage of any other member of the CSCI2100 community.

- Highly encourage you to **discuss in collaboration** with your classmates, but **do your own work!**

Expectations

- Read the document on **The Student/Faculty Expectations** on Teaching and Learning
  - http://www.cse.cuhk.edu.hk/irwin.king/_media/teaching/csci2100/staffstudentexpectations.pdf
- Read the page on **Honesty in Academic Work**
  - https://www.cuhk.edu.hk/policy/academichonesty/index.htm

Course Overview

	Lecture I	Lecture II	Tutorial I	Tutorial II
Time	Tue, 12:30 pm - 1:15 pm	Wed, 11:30 am - 1:15 pm	Wed, 5:30 pm - 6:15pm	Thur, 5:30 pm - 6:15 pm
Venue	ERB LT	ERB LT	ERB 407	LHC 103

- Course Description
  - The concept of abstract data types and the advantages of data abstraction are introduced. Various commonly used abstract data types including vector, list, stack, queue, tree, and set and their implementations using different data structures (array, pointer based structures, linked list, 2-3 tree, B-tree, etc.) will be discussed. Sample applications such as searching, sorting, etc. will also be used to illustrate the use of data abstraction in computer programming. Analysis of the performance of searching and sorting algorithms. Application of data structure principles.

- 本科介紹抽象數據類型之概念及數據抽象化的優點。並討論多種常用的抽象數據類型，包括向量、表格、堆棧、隊列、樹形；集(合)和利用不同的數據結構(例如：陣列、指示字為基的結構、連接表、2-3樹形、B樹形等)作出的實踐。更以實例(例如：檢索、排序等)來說明數據抽象化在計算機程序設計上的應用。並討論檢索與排序算法及數據結構之應用。

Course Overview

- Learning Objectives

1. To understand the **concepts** and **operations** of various data structures and their applications
2. To understand the concept of **abstract data types**
3. To have basic knowledge of **algorithms** and **complexity** of algorithms

- Learning Outcomes

1. To be able to **implement** the following **data structures** as abstract data types in a high level programming language: stack, queue, hash table, list, binary search tree (including AVL tree, red black tree and splay tree), B-tree, trie, disjoint set, graph (including minimum spanning tree and shortest path).
2. To be able to **use appropriate data structures** in different applications.
3. To be able to **implement** abstract data types.
4. To be able to **analyse** the complexity of simple algorithms (such as searching and sorting).

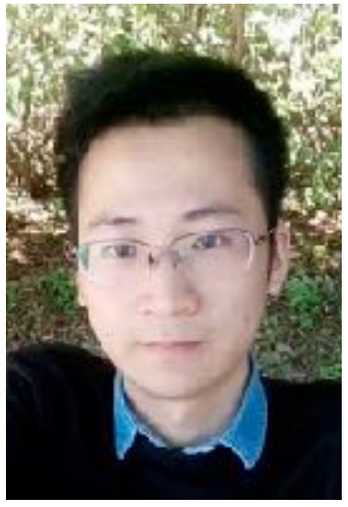
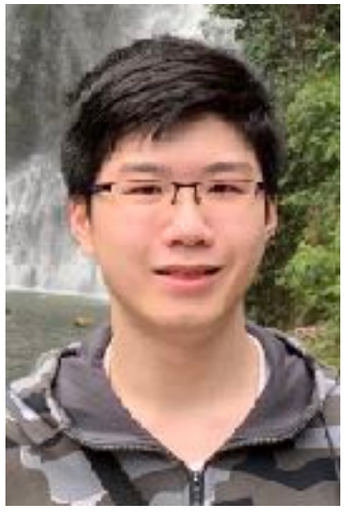
Course Syllabus

- Week 1—Introduction
- Week 2—Algorithm analysis
- Week 3—Lists, stacks, and queues
- Week 4—Lists, stacks, and queues; Tree data structures and algorithms I
- Week 5— Tree data structures and algorithms II
- Week 6—Heaps
- Week 7—Hash
- Week 8—Written midterm examination
- Week 9—Sorting algorithms I
- Week 10—Sorting algorithms II
- Week 11—Programming midterm examination
- Week 11—Graph data structures and algorithms I
- Week 12—Graph data structures and algorithms II
- Week 13—Course summary and wrap up

Grade Assessment Scheme

1. Assignments (25%)
    1. Written assignment (~5) (5%)
    2. Programming assignment (~5) (20%)
    3. Optional quizzes
    4. Late penalty: 10% for each day (no later than 3 days, only applicable for written assignment)
2. Midterms (55%)
    1. Written (20%)
    2. Programming (35%)
3. Final Examination (20%)
4. Extra Credit (There is no penalty for not doing the extra credit problems. Extra credit will only help you in borderline cases.)

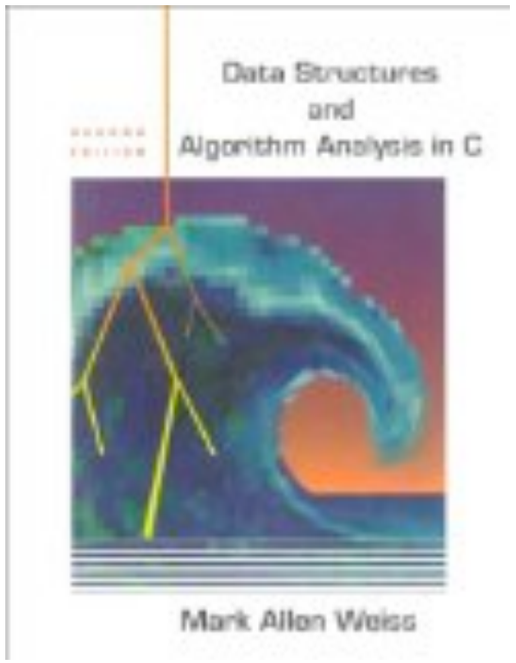
TAs for 2018-19 T2

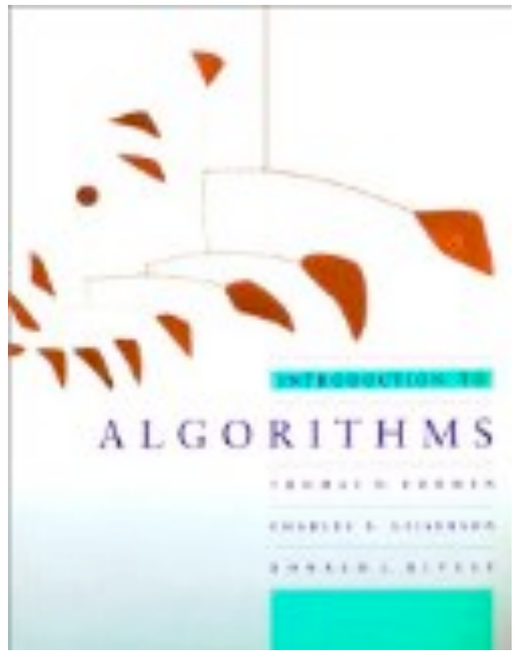




	Lecturer	Tutor 1	Tutor 2	Tutor 3	Tutor 4
Name	Irwin King	Jiani Zhang	Wang Chen	Xinyu Fu	TBD
Email	king AT cse.cuhk.edu.hk	jnzhang AT cse.cuhk.edu.hk	wchen AT cse.cuhk.edu.hk	xyfu AT cse.cuhk.edu.hk	TBD
Office	SHB 908	SHB 1024	SHB 1024	SHB 1024	TBD
Telephone	3943 8398				
Office Hour(s)	Mon, 9:30-10:30, and by appointment	Mon, 16:00-17:00	Tue, 17:00-18:00	Thu, 16:00-17:00	TBD

Administration

- Assignments (25%)
- Midterms (55%)
  - Written (20%)
  - Programming (35%)
- Final Examination (20%)





- Mark Weiss, Data Structures and Algorithm Analysis in C, Addison Wesley, 1997.
- Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, Introduction to Algorithms, The MIT Press, 1990.

A Few New Things

- Partially flipped-classroom
- Some video lectures may be available to view before coming to class
- More smaller homework assignments
- More programming tutorials and exercises
- Peer learning through teamwork
- Offering a **mirrored ESTR** session with more materials and work

If programming languages were vehicles

C was the great all-arounder: compact, powerful, goes everywhere, and reliable in situations where your life depends on it.

Java is another attempt to improve on C. It sort of gets the job done, but it's way slower, bulkier, spews pollution everywhere, and people will think you're a redneck.

C++ is the new C — twice the power, twice the size, works in hostile environments, and if you try to use it without care and special training you will probably crash.

Python is great for everyday tasks: easy to drive, versatile, comes with all the conveniences built in. It isn't fast or sexy, but neither are your errands.

C# is C++ with more safety features so that ordinary civilians can use it. It looks kind of silly but it has most of the same power so long as you stay near gas pumps and auto shops and the comforts of civilization. A well-known heavily muscular intimidator keeps touting it.

Perl used to serve the same purpose as Python, but now only bearded ex-hippies use it.

http://s3.crashworks.org.s3-website-us-east-1.amazonaws.com/if-programming-languages-were-vehicles/

If programming languages were vehicles

LISP is programming stripped down to the bare essence. It's been around since forever. Using it makes you stronger, but only an athlete or a maniac can make a living with with it.

Go is a shiny new toy that tech nerds say will be the way of the future, but it's only practical if you limit everything you want to do to stay within its range.

PHP is this hand-me-down deathtrap that you only use because you're stuck with it, and when you hit a speed bump the wrong way it sets you and your passengers on fire.

COBOL probably seemed like a good idea at the time.

If programming languages were vehicles

MATLAB is what scientists use to do special scientist things.

R is what scientists use when they can't afford MATLAB.

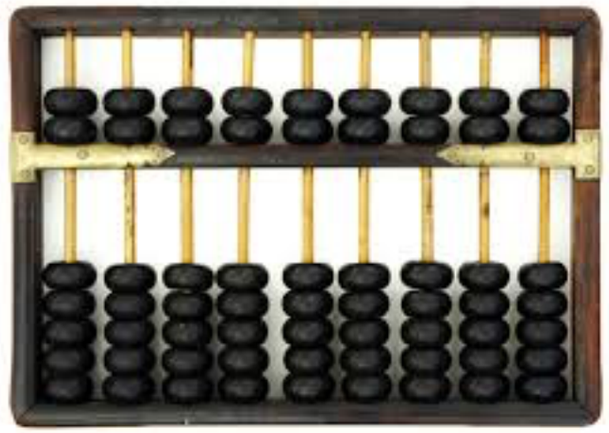
This is Javascript. If you put big wheels and a racing stripe on a golf cart, it's still a f***ing golf cart.

OCaml is this funny shaped thing that Europeans like for some reason.

Why C?

- Powerful, efficient, flexible, robust, etc.
- Simple low-level language without being object-oriented
- Comes with pointers for efficient data structures
- It's been around for a long time
- It is being used in system software, e.g., Unix

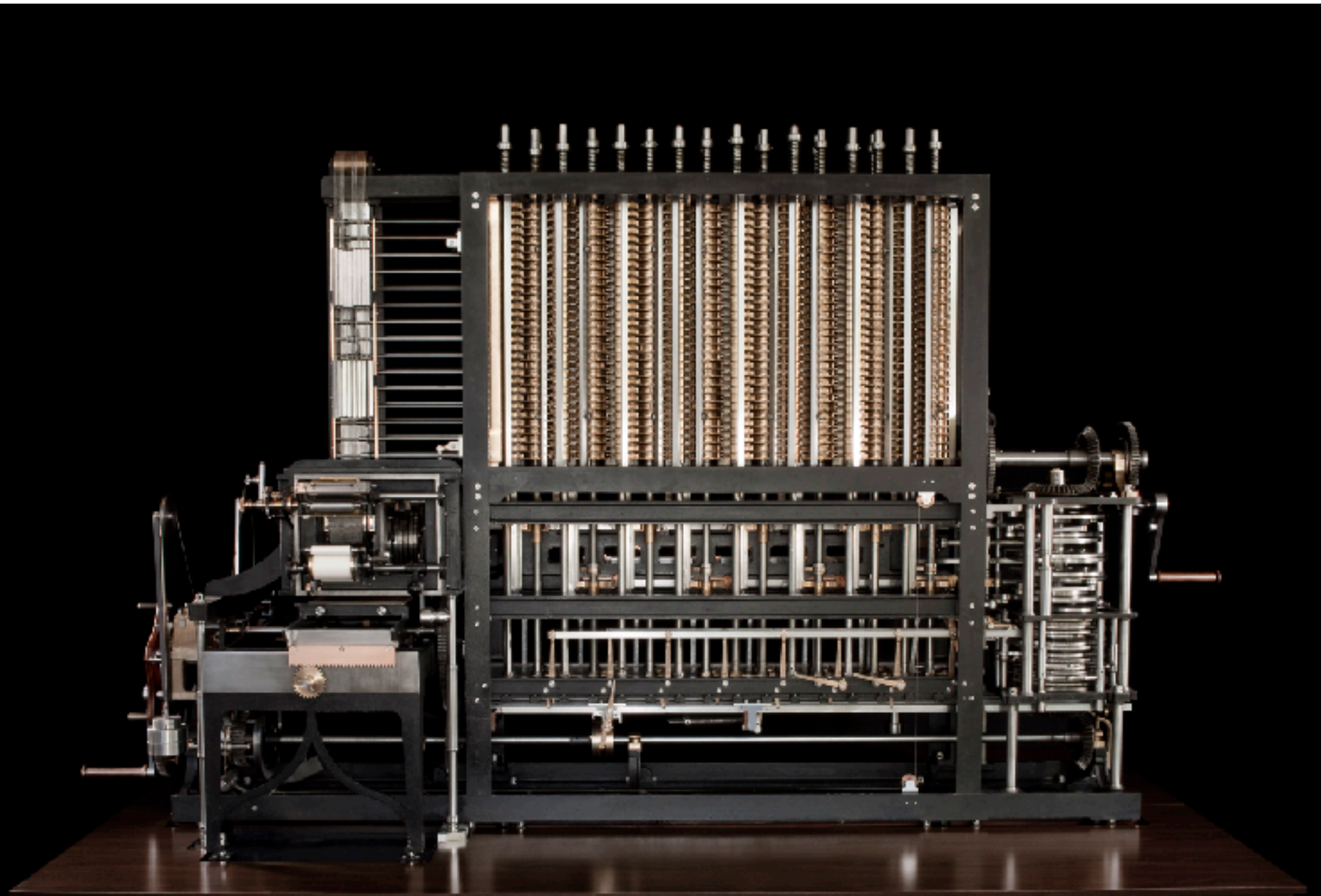
Ancient Computing Tools



Abacus

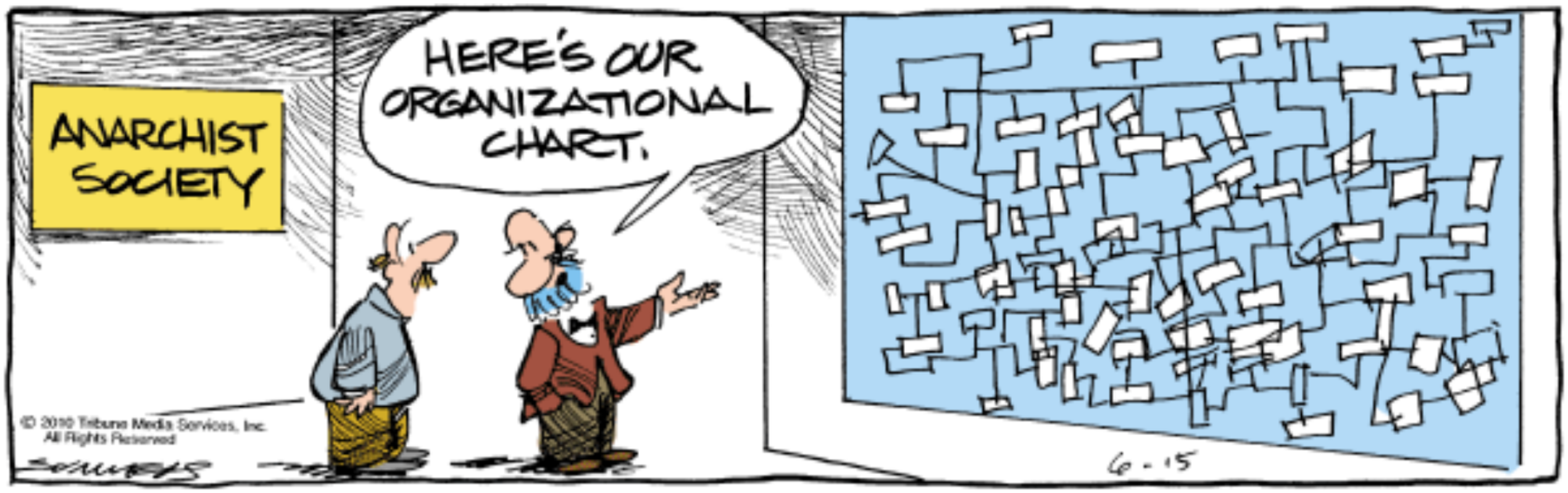
Slide Rules

Babbage Machine



http://www.computerhistory.org/babbage/

Data Structures = Organization



http://bilbosrandomthoughts.blogspot.hk/2012/04/cartoon-saturday_28.html

http://diymusician.cdbaby.com/musician-tips/expect-the-unexpected/

Some Interview Problems

- Write a function that counts the number of primes in the range [1-N]. Write the test cases for this function.
- Write a function that inserts an integer into a linked list in ascending order. Write the test cases for this function.
- Write the InStr function. Write the test cases for this function.
- Write a function that will return the number of days in a month (not using System.DateTime).
- Write a program to output all elements of a binary tree while doing a Breadth First traversal through it.
- Write a method to combine two sorted linked lists into one sorted list without using temporary nodes.

Example

- Given 10 numbers, find the maximum value in the list.
- Given 10 numbers, find the 3rd highest number in the list.
- Given 1,000 numbers, find the 500th highest number in the list.
- Given 1,000 numbers, find the *k*-th highest number in the list.

Example

- A *k* selection problem—given a set of numbers, select the *k*-th highest number in the list.

- How do you perform the above task?

Solution #1

- Read the *N* numbers into an array
- Sort the array in decreasing order by some simple algorithm such as bubblesort
- Return the element in position *k*

Solution #2

- Read the first *k* elements into an array and sort them (in decreasing order).
- Next, each remaining element is read one by one.
- As a new element arrives, it is ignored if it is smaller than the *k*-th element in the array.
- Otherwise, it is placed in its correct spot in the array, bumping one element out of the array.
- When the algorithm ends, the element in the *k*-th position is returned as the answer.

Notes

- How many different algorithms can solve the *k* selection problem?
- How many different programs can solve the *k* selection problem?

The Magic Square

?	?	?
?	?	?
?	?	?

- Place 1, 2, …, 9 into the 3x3 matrix so that the summation of the row, column, and diagonal is the same.
- Brute force would take 9!
- What is the procedure to generate the 3x3 matrix? How about an n x n matrix?
- How many solutions are there?
- Can you find a solution or all solutions?

Notes

- What is the difference between an algorithm and a program?
- Algorithm
  - a process or set of rules used for calculation or problem-solving, especially with a computer
    - algorithm is a step by step outline or flowchart how to solve a problem
- Program
  - a series of coded instructions to control the operation of a computer or other machine
  - program is an implemented coding of a solution to a problem based on the algorithm

Example

- Let's come up with a data structure for storing the date
- "01/01/08" (8 bytes) as string is good
- "2001/01/08" (10 bytes) as string is better
- "20010108" (8 bytes) as string is even better

Example

- However, we need not use string
- We can store it using bytes and bits
  - Year: 2 bytes (65K)
  - Month: 4 bits (16)
  - Day: 5 bits (32)
  - Total: 25 bits.
- What are the algorithms to process these data structures?

How About Storing Numbers?

- Given a series of 0's and 1's, how can we represent them?

The GCD Problem

- Problem: Find the greatest common divisor (GCD) of two integers, *m* and *n*.
    - Note: two positive integers that have greatest common divisor 1 are said to be relative prime to one another.

- Euclid's Algorithm:

```
while m is greater than zero:
    If n is greater than m, swap m and n.
    Subtract n from m.
n is the GCD
```

http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/

The GCD Problem

- Program (in C):

```
int gcd(int m, int n)
/* precondition: m>0 and n>0.  Let g=gcd(m,n). */
 { while( m > 0 )
     { /* invariant: gcd(m,n)=g */
      if( n > m )
        { int t = m; m = n; n = t; } /* swap */
       /* m >= n > 0 */
       m -= n;
     }
   return n;
}
```


The GCD Problem

- Correctness: Why do we believe that this algorithm devised thousands of years ago, is correct?
  - Given m>0 and n>0, let g = gcd(m,n).
    - (i) If m=n then m=n=gcd(m,n) and the algorithm sets m to zero and returns n, obviously correct.
    - (ii) Otherwise, suppose m>n. Then m=p×g and n=q×g where p and q are coprime, from the definition of greatest common divisor. We claim that gcd(m-n,n)=g.
      Now m-n=p×g-q×g=(p-q)g. so we must show that (p-q) and q are coprime. If not then p-q=a×c and q=b×c for some a,b,c>1. But then p=q+a×c=b×c+a×c=(a+b)×c and because q=b×c, p and q would not have been coprime ... contradiction. Hence (p-q) and q are coprime, and gcd(m-n,n)=gcd(m,n).
    - (iii) If m<n, the algorithm swaps them so that m>n and that case has been covered.
- So the algorithm is correct, provided that it terminates.

The GCD Problem

- Termination
  - At the start of each iteration of the loop, either n>m or m≥n.
    - (i) If m≥n, then m is replaced by m-n which is smaller than the previous value of m, and still non-negative.
    - (ii) If n>m, m and n are exchanged, and at the next iteration case (i) will apply.
    - So at each iteration, max(m,n) either remains unchanged (for just one iteration) or it decreases.
      This cannot go on for ever because m and n are integers (this fact is important), and eventually a lower limit is reached, when m=0 and n=g.
- So the algorithm does terminate.

The GCD Problem

- Testing: Having proved the algorithm to be correct, one might argue that there is no need to test it. But there might be an error in the proof or maybe the program has been coded wrongly.
- Good test values would include:
  - special cases where m or n equals 1, or
  - m, or n, or both equal small primes 2, 3, 5, ..., or
  - products of two small primes such as p1×p2 and p3×p2,
  - some larger values, but ones where you know the answers,
  - swapped values, (x,y) and (y,x), because gcd(m,n)=gcd(n,m).
- The objective in testing is to "exercise" all paths through the code, in different combinations.
- Debugging code be inserted to print the values of *m* and *n* at the end of each iteration to confirm that they behave as expected.

The GCD Problem

- Complexity
  - We are interested in how much ***time*** and ***space*** (computer memory) a computer algorithm uses; i.e. how efficient it is.
  - This is called *time-* and *space-complexity*.
  - Typically the complexity is a function of the values of the inputs and we would like to know what function.
  - We can also consider the *best-*, *average-*, and *worst-cases*.

The GCD Problem

- Time
  - The time to execute one iteration of the loop depends on whether m>n or not, but both cases take constant time: one test, a subtraction and 4 assignments vs. one test, a subtraction and one assignment. So the time taken for one iteration of the loop is bounded by a constant. The real question then is, how many iterations take place? The answer depends on m and n.
  - If m=n, there is just one iteration; this is the best-case.
    If n=1, there are m iterations; this is the worst-case (equivalently, if m=1 there are n iterations).
    The average-case time-complexity of this algorithm is difficult to analyze.

The GCD Problem

- Space
  - The space-complexity of Euclid's algorithm is a constant, just space for three integers: *m*, *n*, and *t*.
  - We shall see later that this is `O(1)'.
- Exercises
  - Devise a quicker version of Euclid's algorithm that does not sit in the loop subtracting individual copies of n from m when *m*>>*n*.
  - Devise a GCD function that works for three or more positive integers as the largest divisor shared by all of them.

Study of Algorithms I

- Machines for executing algorithms
  - What is the processing speed?
  - How large is the processing space (memory)?
  - What is the organization of the processors?
- Languages for describing algorithms
  - Language design and translation
  - Syntax specification and semantics

Study of Algorithms II

- Foundations of algorithms
  - What is the minimum number of operations necessary for any algorithm?
    - What is the algorithm which performs the function?
- Analysis of Algorithms
  - What is the performance profile of the algorithm?

Study of Algorithms III

- How to devise algorithms
- How to express algorithms
- How to validate algorithms
- How to analyze algorithms
- How to test a program

Definition

- An algorithm is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:
  - Input
  - Output
  - Definiteness
  - Finiteness
  - Effectiveness

Another Definition

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into the output.

Definition

- We can also view an algorithm as a tool for solving a well-specified computational problem.
- The statement of the problem specifies in general terms the desired input/output relationship.
- The algorithm describes a specific computational procedure for achieving that input/output relationship.

Definition

- input: there are zero or more quantities which are externally supplied;
- output: at least one quantity is produced;
- definiteness: each instruction must be clear and unambiguous;
- finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

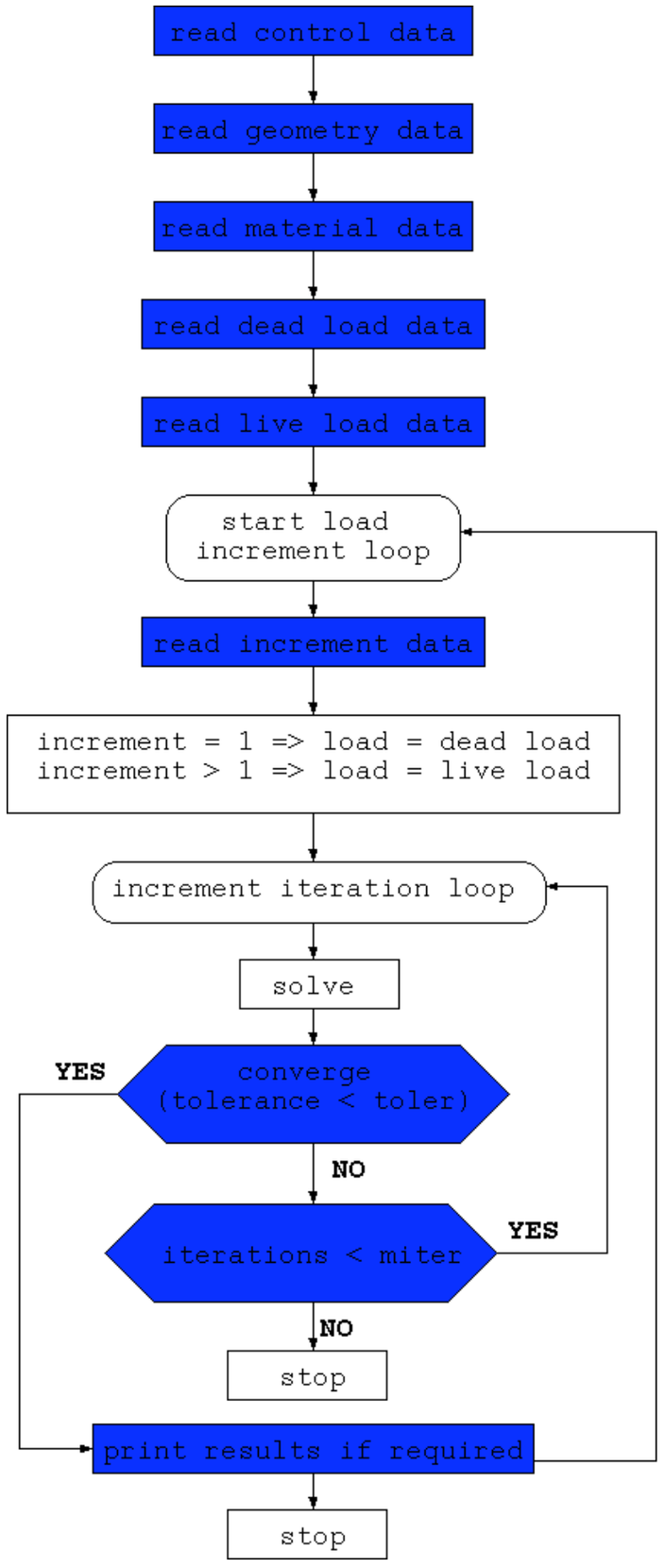
Definition

- effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper.
- It is not enough that each operation be definite as in (3), but it must also be feasible.

Notes on Algorithm

- An algorithm is said to be correct if, for every input instance, it halts with the correct output.
- We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired answer.
- Contrary to what one might expect, incorrect algorithms can sometimes be useful, if their error rate can be controlled.

A Flowchart



http://www.swan.ac.uk/civeng/Research/masonry/flowch.html

Study of Data

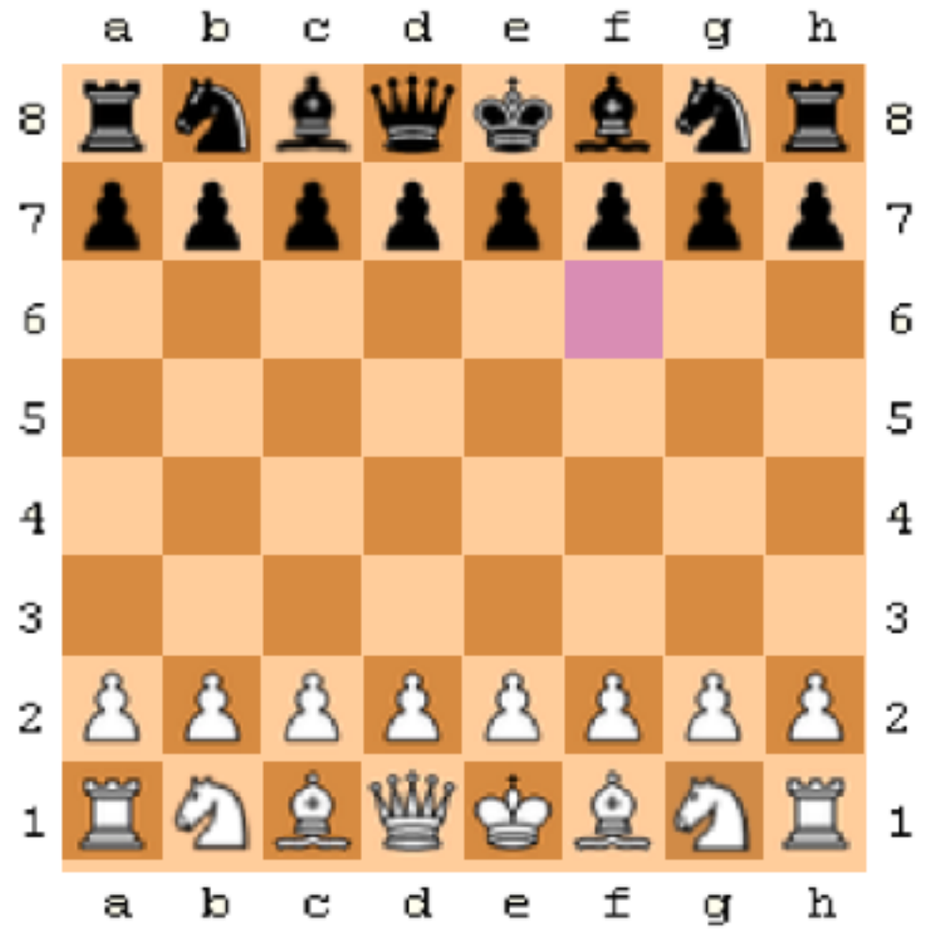
- Machines that hold data
- Languages for describing data manipulation
- Foundations which describe what kinds of refined data can be produced from raw data
- Structures for representing data

- A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently

Definition

- A data structure is a set of domains *D*, a designated domain *d* in *D*, a set of functions *F* and a set of axioms *A*.
- d = natno
- *D* = {boolean, integer, float}
- *F* = {ZERO, SUCC, ADD}
- *A* = {line 7 to 10 of the structure NATNO}
- The set of axioms describes the semantics of the operations.
- An implementation of a data structure *d* is a mapping from *d* to a set of other data structures e.

Chessboard Representation



- Requirements
  - Location of each piece on the board
  - Whose turn it is to move
  - Status of the 50-move draw rule
  - Whether a player is disqualified to castle
  - If an en passant capture is possible

Types of Representation

- Piece lists--16 black and white pieces
- Array-based--an 8x8 two-dimensional array
- 0x88 method
- Bitboard--use 64-bits to represent one piece
- Stream-based
- Huffman encodings

Huffman Encodings

```
Empty square = 0
Pawn         = 10c
Bishop       = 1100c
Knight       = 1101c
Rook         = 1110c
Queen        = 11110c
King         = 11111c
```

where c is a bit representing the color of the piece (1 = LIGHT, 0 = DARK).

Additional bits are needed for:

50-move rule (7 bits)

en-passant column (3 bits)

color to move (1 bit)

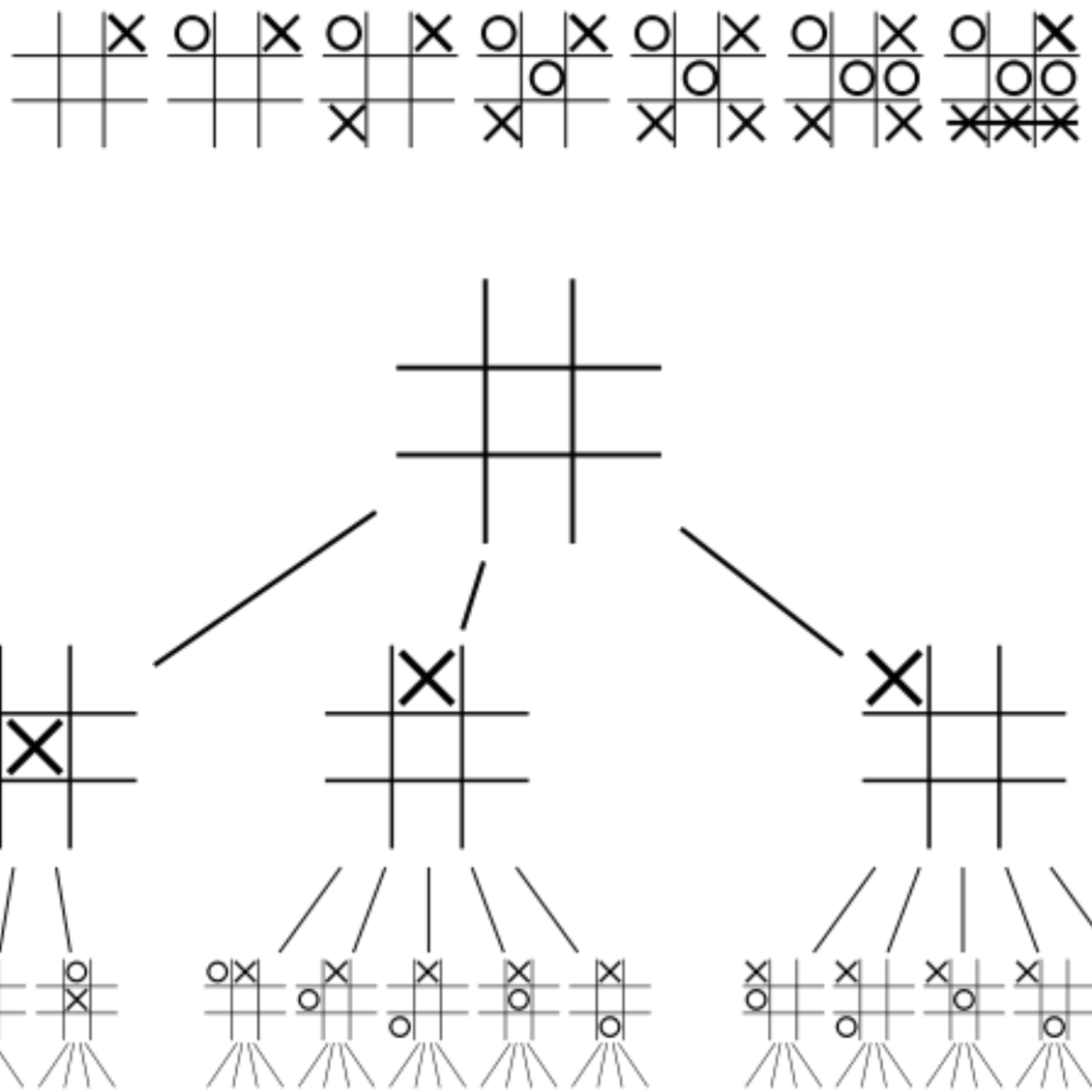
castling rights (4 bits).

Notes

- Row = (int) (position / 8)
- Column = position % 8
- internal int Score;
- internal bool BlackCheck;
  internal bool BlackMate;
  internal bool WhiteCheck;
  internal bool WhiteMate;
  internal bool StaleMate;
- internal byte FiftyMove;
  internal byte RepeatedMove;
- internal bool BlackCastled;
  internal bool WhiteCastled;

```
0  1  2  3  4  5  6  7
8  9  10 11 12 13 14 15
16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55
56 57 58 59 60 61 62 63
```

A Simple Tic Tac Toe Game



```
This is the game of Tic Tac Toe.
You will be playing against the computer.
   |   |
---|---|---
   |   |
---|---|---
   |   |
Enter X,Y coordinates for your move: 1,1
 X | O |
---|---|---
   |   |
---|---|---
   |   |
Enter X,Y coordinates for your move: 2,2
 X | O | O
---|---|---
   | X |
---|---|---
   |   |
Enter X,Y coordinates for your move: 3,3
You won!
 X | O | O
---|---|---
   | X |
---|---|---
   |   | X
```


http://www.java2s.com/Code/C/Data-Type/AsimpleTicTacToegame.htm

Complexity of the Problem

- How many possible board layouts are there?
- How many different sequences for placing the X's and O's on the board?
- How many possible games, assuming X makes the first move every time?
- What about a 3-dimensional tic-tac-toe on a 3x3x3 board? or an n x n x n game?

Preamble

```
/*
C: The Complete Reference, 4th Ed. (Paperback)
by Herbert Schildt

ISBN: 0072121246
Publisher: McGraw-Hill Osborne Media; 4 edition (April 26, 2000)
*/
```

```
#include <stdio.h>
#include <stdlib.h>

char matrix[3][3];  /* the tic tac toe matrix */

char check(void);
void init_matrix(void);
void get_player_move(void);
void get_computer_move(void);
void disp_matrix(void);
```


Main

```
int main(void)
{
  char done;

  printf("This is the game of Tic Tac Toe.\n");
  printf("You will be playing against the computer.\n");

  done =  ' ';
  init_matrix();

  do {
    disp_matrix();
    get_player_move();
    done = check(); /* see if winner */
    if(done!= ' ') break; /* winner!*/
    get_computer_move();
    done = check(); /* see if winner */
  } while(done== ' ');

  if(done=='X') printf("You won!\n");
  else printf("I won!!!!\n");
  disp_matrix(); /* show final positions */

  return 0;
}
```


Matrix Initialization

```
/* Initialize the matrix. */
void init_matrix(void)
{
  int i, j;

  for(i=0; i<3; i++)
    for(j=0; j<3; j++) matrix[i][j] =  ' ';
}
```


Player's Move

```
/* Get a player's move. */
void get_player_move(void)
{
  int x, y;

  printf("Enter X,Y coordinates for your move: ");
  scanf("%d%*c%d", &x, &y);

  x--; y--;

  if(matrix[x][y]!= ' '){
    printf("Invalid move, try again.\n");
    get_player_move();
  }
  else matrix[x][y] = 'X';
}
```


Computer's Move

```
/* Get a move from the computer. */
void get_computer_move(void)
{
  int i, j;
  for(i=0; i<3; i++){
    for(j=0; j<3; j++)
      if(matrix[i][j]==' ') break;
  }

  if(i*j==9)  {
    printf("draw\n");
    exit(0);
  }
  else
    matrix[i][j] = 'O';
}
```


Display Matrix

```
/* Display the matrix on the screen. */
void disp_matrix(void)
{
  int t;

  for(t=0; t<3; t++) {
    printf(" %c | %c | %c ",matrix[t][0],
            matrix[t][1], matrix [t][2]);
    if(t!=2) printf("\n---|---|---\n");
  }
  printf("\n");
}
```


Check For Winner

```
/* See if there is a winner. */
char check(void)
{
  int i;

  for(i=0; i<3; i++)  /* check rows */
    if(matrix[i][0]==matrix[i][1] &&
       matrix[i][0]==matrix[i][2]) return matrix[i][0];

  for(i=0; i<3; i++)  /* check columns */
    if(matrix[0][i]==matrix[1][i] &&
       matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* test diagonals */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matrix[2][2])
       return matrix[0][0];

  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matrix[2][0])
       return matrix[0][2];

  return ' ';
}
```


Chinese Checker

- What is the best way to represent the board?
- How to check for valid moves?
- How to generate new moves?

http://pobox.upenn.edu/~davidtoc/images/chinesecheckers/ohio2.html

Five Phases of a Program

1. Requirement
2. Design
3. Analysis
4. Coding
5. Verification
   - Program proving
   - Program testing
   - Program debugging

Methodology

- Top-down approach
- Bottom-up approach

Mathematics Review

- Exponents
- Logarithms
- Series
- Modular Arithmetic
- Proofs

Exponents

$$
\begin{aligned}
X^A X^B &= X^{A+B} \\
\frac{X^A}{X^B} &= X^{A-B} \\
(X^A)^B &= X^{AB} \\
X^N + X^N &= 2X^N \neq X^{2N} \\
2^N + 2^N &= 2^{N+1}
\end{aligned}
$$

Logarithms

$$
\begin{aligned}
X^A &= B \text{ iff } \log_X B = A \\
\log_A B &= \frac{\log_C B}{\log_C A}; A, B, C > 0, A \neq 1 \\
\log AB &= \log A + \log B; A, B > 0 \\
\log \frac{A}{B} &= \log A - \log B \\
\log(A^B) &= B \log A \\
\log X &< X, \text{ for all } X > 0 \\
\log 1 &= 0 \\
\log 2 &= 1 \\
\log 1,024 &= 10
\end{aligned}
$$

All logarithms are to the base 2 unless specified otherwise.

Series

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=0}^{N} A^i \leq \frac{1}{A-1}, 0 < A < 1, N \to \infty$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=1}^{N} \approx \frac{N^{k+1}}{|K+1|}, k \neq -1$$

Modular Arithmetic

- *A* is congruent to *B* modulo *N*, written $A \equiv B \pmod{N}$, if *N* divides *A* - *B*.
- This means that the remainder is the same when either *A* or *B* is divided by *N*.
- $38 \equiv 14 \pmod{12}$.
- As with equality, if $A \equiv B \pmod{N}$, then $A + C = B + C \pmod{N}$ and $AD = BD \pmod{N}$.

Proofs

- Proofs vs. Approximate Engineering Solutions (Good Enough Solutions)
- Types of Proofs
  - Proof by Induction
  - Proof by Counterexample
  - Proof by Contradiction

Proof by Induction

- There are two steps.
  - Prove a base case to establish that a theorem is true for some small value(s).
  - Next an inductive hypothesis is assumed to be true for all cases up to some limit k.
  - Using this assumption, the theorem is then shown to be true for the next value, which is typically *k*+1.

Example

- Prove that

$$f(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

For $f(1) = 1$, it is true.
Now $f(n) - f(n-1) = (n^2 + n - n^2 + n)/2 = n$ so it is true.

Proof by Counterexample

- The statement $F_k \leq k^2$ is false.
- F_k is the k-th Fibonacci number.
- The easiest way to prove this is to compute $F_{11} = 144 > 112$ where the example fails.

Proof by Contradiction

- Proof by Contradiction proceeds by
  - assuming that the theorem is false and
  - showing that this assumption implies that some known property is false, and
  - hence the original assumption was erroneous.

Example

- Proof that there is an infinite number of primes.
- Assume that the theorem is false, so that there is some largest prime P_k.
- Let $P_1, P_2, \ldots, P_k$ be all the primes in order and consider $N = P_1 \, P_2 \, P_3 \, \ldots \, P_k + 1$.
- Clearly, N is larger than P_k, so by assumption N is not prime so this is a contradiction.

Summary

- Data structures is the way we organize our data for the efficient computation of algorithm
- Data structures and algorithms go hand in hand as each component can affect the design of the other
- Designing algorithm is an art and also a science