https://www.cse.cuhk.edu.hk/irwin.king/teaching/csci2100b/2013

CSCI2100B 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

IRWIN KING @ WEB INTELLIGENCE & SOCIAL COMPUTING LAB

Trace: » 2013

You are here: home » teaching » csci2100b » 2013

NAVIGATION

- Home
- Profile
- Research Interests & Projects

ABOUT US

- Research Group | Presentations
- News | Newsletter
- Collaborators
- Contact Us

PUBLICATIONS

1. Conference Papers 2010-Now
2. Conference Papers 2005-2009
3. Journal Articles
4. Books, Edited Books & Proceedings
5. Book Chapters
6. Conference Papers 2000-2004
7. Conference Papers 1994-1999
8. Theses
9. Patents
10. Presentations
11. Datasets

PROFESSIONAL ACTIVITIES

1. Professional Achievements
2. Awards and Recognitions
3. Grants
4. Teaching
5. Education Excellence
6. Demos & Software
   I. Finding Experts Demo
   II. MEMPM Matlab Toolbox
   III. My iApps
7. Conference Activities

CSC2100B Data Structures

–Table of Contents

Breaking News

Edit

1. **11 January 2013**. Are you ready for the new semester? Welcome to CSC2100B Data Structures. This is an important fundamental course not only for Computer Scientists but for all engineering students. The knowledge learned here can be applied in any programming environment to help you write better programs and applications. Let's have fun in this course!

Edit

Spring 2013

	Lecture I	Lecture II	Tutorial I	Tutorial II
Time	M7-8, 2:30 pm - 4:15 pm	T2, 9:30 am - 10:15 am	T5, 12:30 pm - 1:15 pm	W8, 3:30 pm - 4:15 pm
Venue	LSB LT6	LSB LT6	ERB 404	ERB 404

The Golden Rule of CSC2100B: No member of the CSC2100B community shall take unfair advantage of any other member of the CSC2100B community.

Edit

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。

Personnel

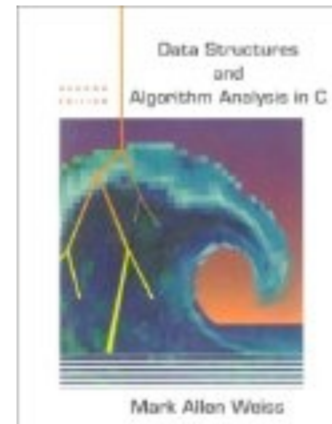
	Lecturer	Tutor	Tutor
Name	Irwin King	Yuanming Yu	Hongyi Zhang
Email	king AT cse.cuhk.edu.hk	ymyu AT cse.cuhk.edu.hk	hyzhang AT cse.cuhk.edu.hk
Office	Rm 908	Rm 606	Rm 1024
Telephone	3943 8398	6761 7170	5984 1406
Office Hour(s)	* M2, Monday 9:30 to 10:30 * M3, Monday 10:30 to 11:30 * By appointment.	NA	NA

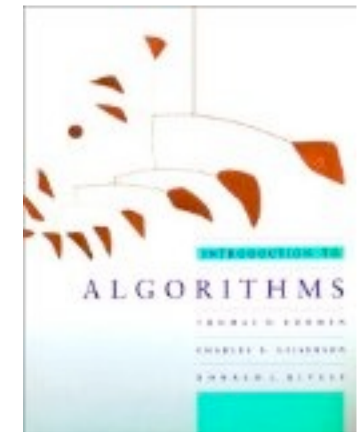
Note: This class will be taught in English. Homework assignments and examinations will be conducted in English.

Week	Date	Topics	Tutorials	Homework & Events	Resources
1	14/1, 15/1	**Introduction**	Introduction to C	TBD	TBD
2	21/1, 22/1	1. **Algorithm Analysis** 2. **Recurrence**	TBD	TBD	TBD
3	28/1, 29/1	**List, Stacks, and Queues**	TBD	TBD	TBD
4	4/2, 5/2	**List, Stacks, and Queues** **Tree Data Structures and Algorithms**	TBD	TBD	TBD
5	11/2, 12/2	**CHINESE NEW YEAR HOLIDAY**	**NO TUTORIALS**	TBD	TBD
6	18/2, 19/2	**Tree Data Structures and Algorithms**	TBD	TBD	TBD
7	25/2, 26/2	**Hash Functions**	TBD	TBD	TBD
8	4/3, 5/3	**Hash Functions**	TBD	TBD	TBD
9	11/3, 12/3	1. **Heaps** 2. **Sorting Algorithms**	TBD	TBD	TBD
10	18/3, 19/3	**Sorting Algorithms**	TBD	TBD	TBD
11	25/3, 26/3	**Sorting Algorithms**	TBD	TBD	TBD
12	1/4, 2/4	**EASTER HOLIDAY** **WRITTEN MIDTERM EXAMINATION**	**NO TUTORIALS**	TBD	TBD
13	8/4, 9/4	**PROGRAMMING MIDTERM EXAMINATION**	TBD	TBD	TBD
14	15/4, 16/4	**Graph Data Structures and Algorithms**	TBD	TBD	TBD
15	22/4, 23/4	**Course summary and final examination pointers**	TBD	TBD	TBD

Administration

- Assignment (20%)
- Midterms (30%)
  - Written (10%)
  - Programming (20%)
- Final Examination (50%)





- 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.

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 (no 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 form with out 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?

Notes

- What is the difference between an algorithm and a program?
- Algorithm
  - a process or set of rules used for calculation or problem-solving, esp. 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. [-concise OED '91]
  - 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?

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 $m=p\times g$ and $n=q\times g$ where p and q are coprime, from the definition of greatest common divisor. We claim that $gcd(m-n,n)=g$. Now $m-n=p\times g-q\times g=(p-q)g$. so we must show that $(p-q)$ and q are coprime. If not then $p-q=a\times c$ and $q=b\times c$ for some $a,b,c>1$. But then $p=q+a\times c=b\times c+a\times c=(a+b)\times c$ and because $q=b\times c$, 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 $m \geq n$.
    - (i) If $m \geq n$, 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;

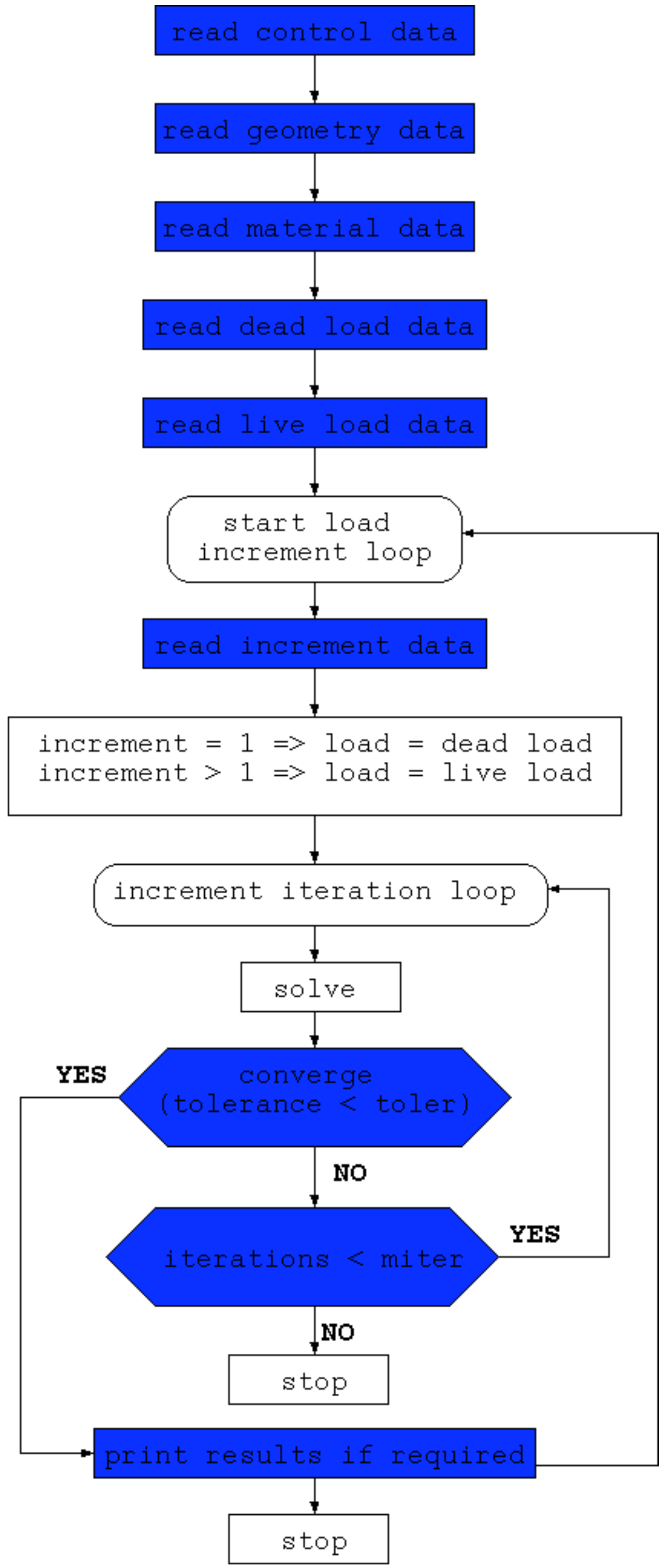
Definition

- 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

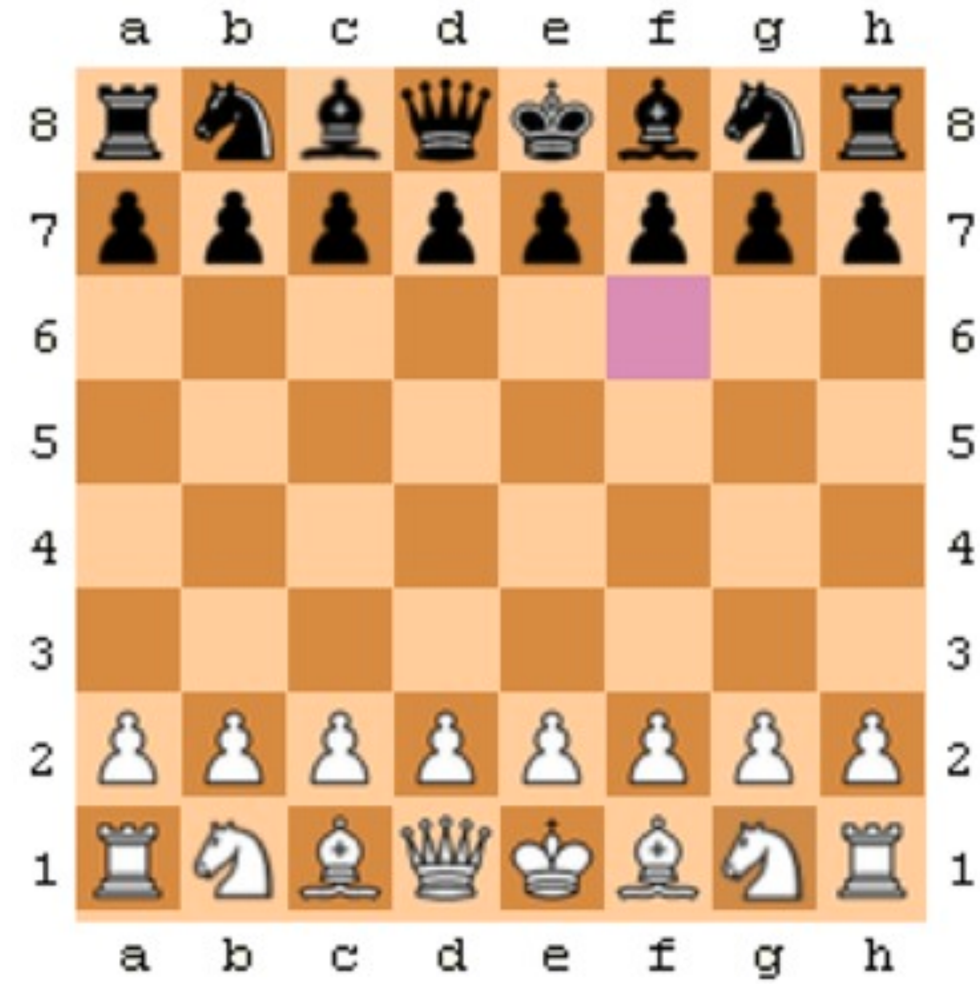
- Machines that hold data
- Languages for describing data manipulation
- Foundations which describe what kinds of refined data can be produced from raw data
- Structures for representing data

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

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)

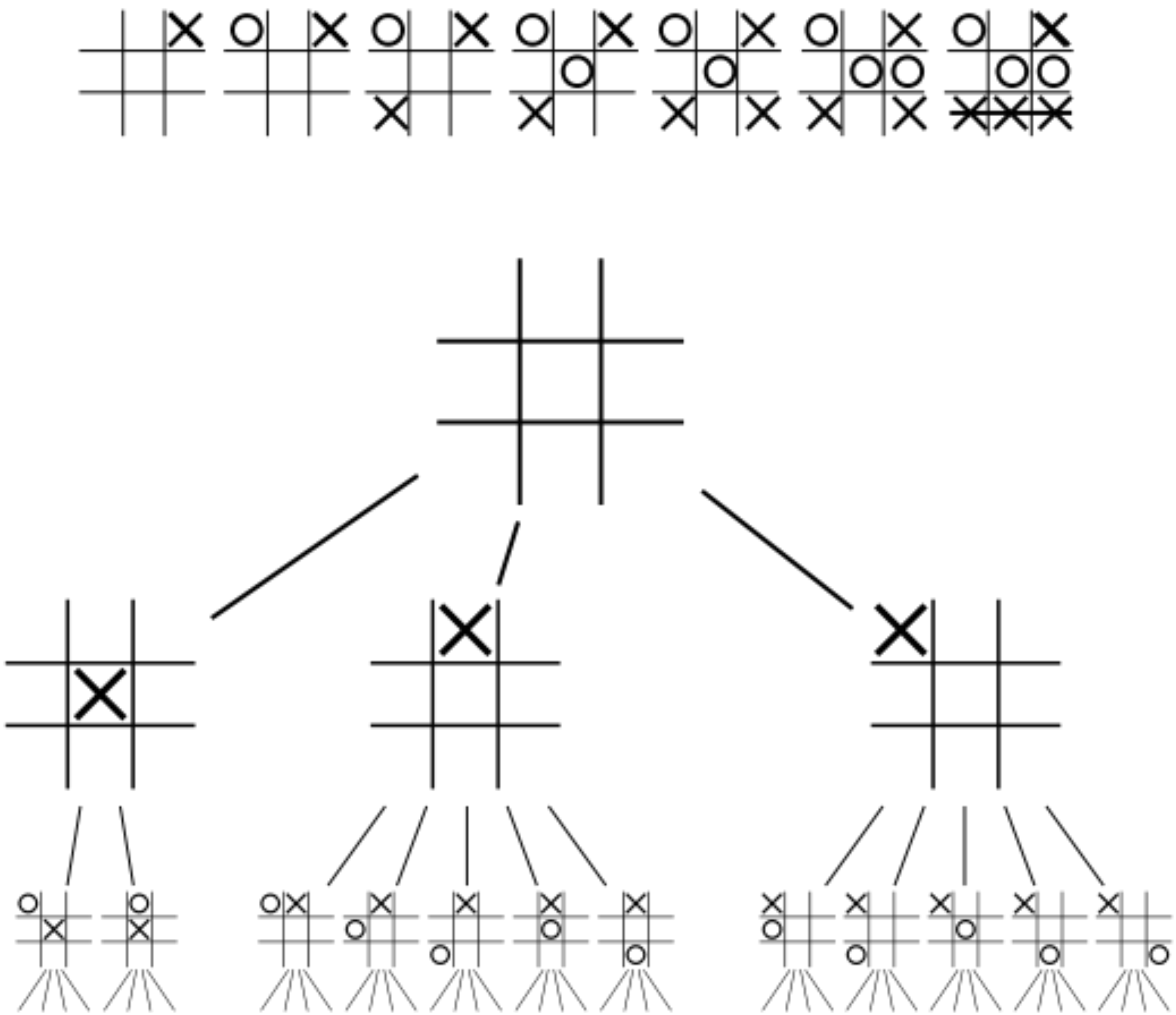
castling rights (4 bits).

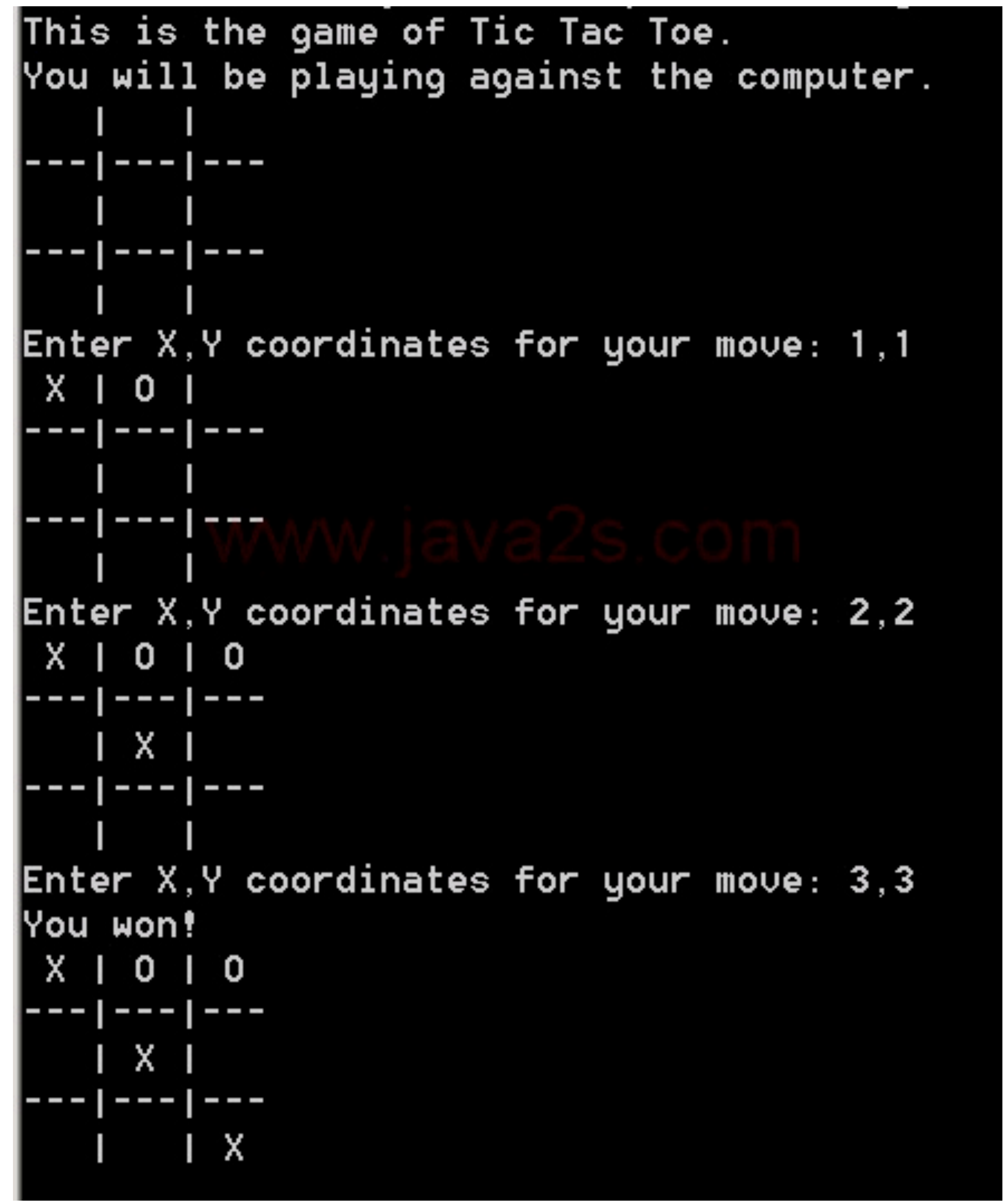
Notes

- `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





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
/*
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

```c
  /* 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(matrix[i][j]==' ') break;
  }

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

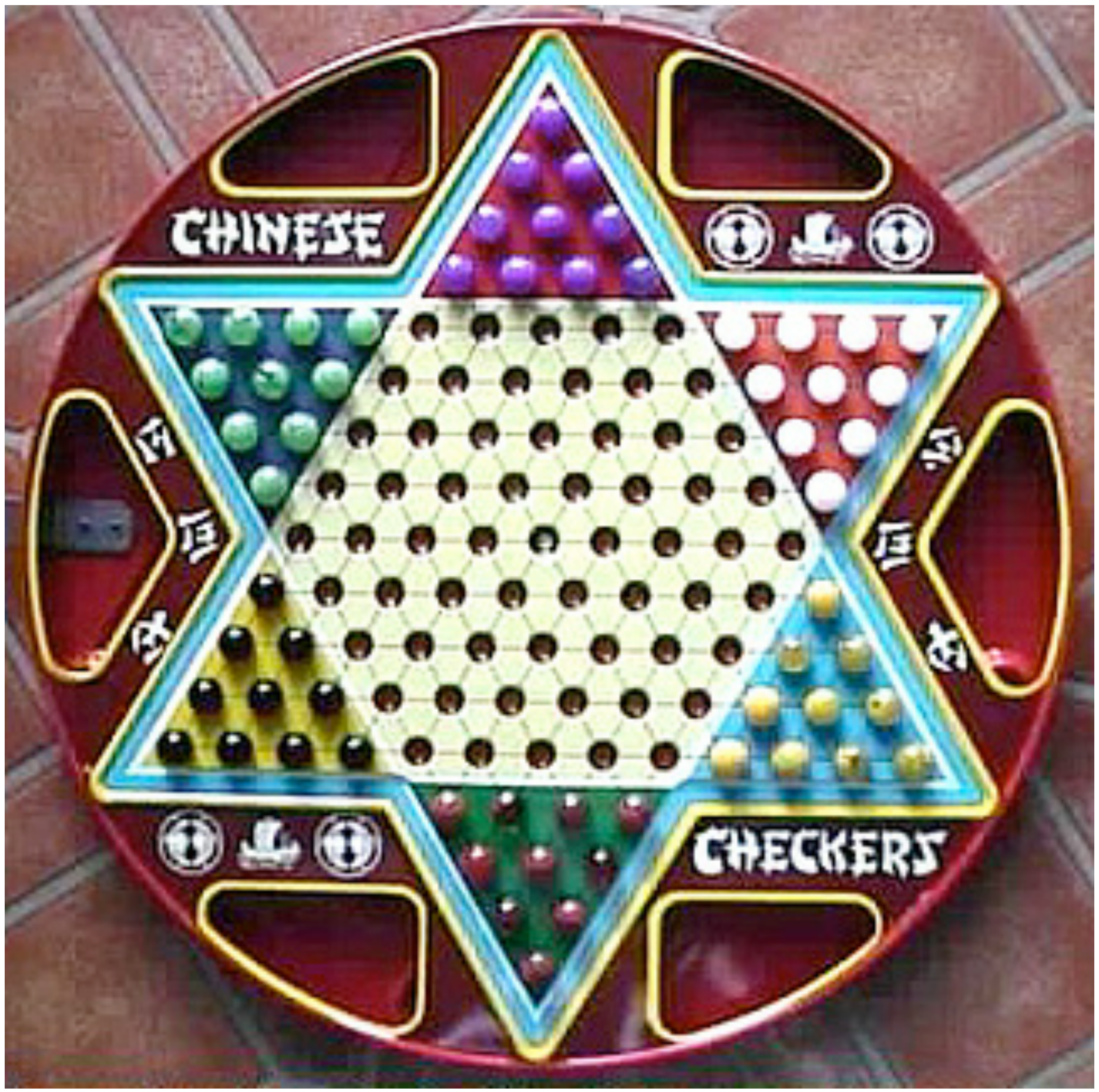

Display Matrix

```c
   /* 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

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

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

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

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

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

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

Modular Arithmetic

- A is congruent to B modulo N, written A B (mod N), if N divides A - B.
- This means that the remainder is the same when either A or B is divided by N.
- 81 = 61 = 1 (mod 10).
- As with equality, if A = B (mod N), then A + C = B + C (mod N) and AD = BD (mod 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 Pk.
- Let P1, P2, …, Pk be all the primes in order and consider N = P1 P2 P3 … Pk + 1.
- Clearly, N is larger than Pk, so by assumption N is not prime so this is a contradiction.

