

# CSI310 Course Summary and Review

Goal: Learn to **think** and **evaluate**. Details in the text we didn't cover can be figured out or studied from other sources when needed.

## Choices—Illustrated by C vs STL string

- ▶ The partially-filled array plus used variable is one implementation choice. It is used by STL strings.
- ▶ The (null char. terminated) C-string is another implementation choice.
- ▶ Important goal: Learn to evaluate and compare multiple choices like these.

# Abstract Data Types VS. Implementations

- ▶ An ADT is an Interface Definition.
  - ▶ Abstract **Operation Names** (and parameters).
  - ▶ Data (such as sorted sequence) and descriptions of what each **Operation** does in terms of data.
  - ▶ Study “Operation Contracts” in Chapter 3 and beyond.
- ▶ **Implementations** are separated from **Interfaces** by “Walls”.
  - ▶ Separate `foo.h` and `foo.cpp` files; `public:` and `private:` class members are what we used in CSI310 for “Walls”.
  - ▶ **Different** implementation data structures for the **same** ADT give different **performance** characteristics.

## List ADT; four implementations

- ▶ **Sequence** of data items:  
**The order matters and repeats are OK.**
- ▶ Operations: INSERT, DELETE, RETRIEVE given a position.
- ▶ Maybe extend with a cursor.
- ▶ Fixed size partially filled array.
- ▶ Dynamically resizable partially filled array.
- ▶ Singly Linked list.
- ▶ Crazy: A binary tree where each node holds the number of nodes in its left child. Use binary search to access an element given the position!

### Study Tips:

Know how to declare and code each above.

For Binary Trees, Binary Search Trees, Binary Heap-ordered Trees: Recognize/declare (simple) classes for/create on paper/do on paper and write pseudo code for search, finding largest and smallest element, in-, pre- and post-order traversal.

# Asymptotic Performance Characteristics-Example

Abstract operation: Given a position number  $P$ , retrieve the item.

- ▶ Array implementations: Worst case time is  $O(1)$  (constant time).
- ▶ Linked List implementation: Worst case time is  $O(N)$ , where  $N$  is the number of items.

BUT for:

Abstract operation: Given a cursor, delete the item.

- ▶ Array implementations: Worst case time is  $O(N)$ .
- ▶ Linked List implementation: Worst case time is  $O(1)$ , (if you're careful to use a previous item pointer).

## Study Tip

What does the computer do and how do you program it? Why and how is the time constant sometimes and  $O(N)$  other times?

# Asymptotic Performance Characteristic-Meaning

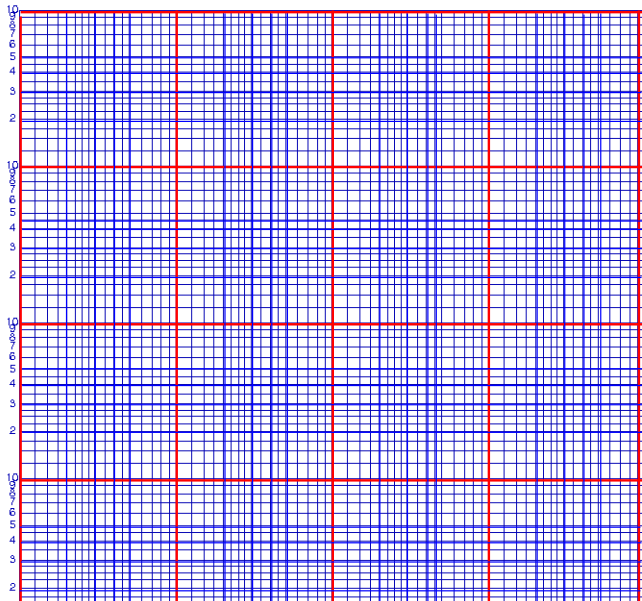
- ▶ We measured or figured out the running time  $T(N)$  as a **function of  $N$**
- ▶ We **plot** the graph of  $T()$ —get a curve as in calculus.
- ▶ We observe the **shape** of the curve: Oblique straight line? Horizontal straight line? Parabola? ( $y = Cx^2$ ) Cubic parabola? ( $y = Cx^3$ ) Logarithmic? ( $y = C \times \log(x)$ ) Log of  $N!$  approx? ( $y = x \times \log(x)$ )
- ▶ (Somewhat more exactly, two functions  $T()$  and  $f()$  have the same asymptotic shape when  $\lim_{N \rightarrow \infty} \frac{|T(N)|}{|f(N)|} \leq C \neq 0$ )
- ▶ The running time of mergesort on  $N$  elements is  $M(N)$ . We know  $M(N) = O(N \times \log(N))$ .
- ▶ The running time of selection on  $N$  elements is  $T(N)$ . We know  $S(N) = O(N^2)$ .

(Actually,  $\Theta(N \log N)$  and  $\Theta(N^2)$  are more correct here.)

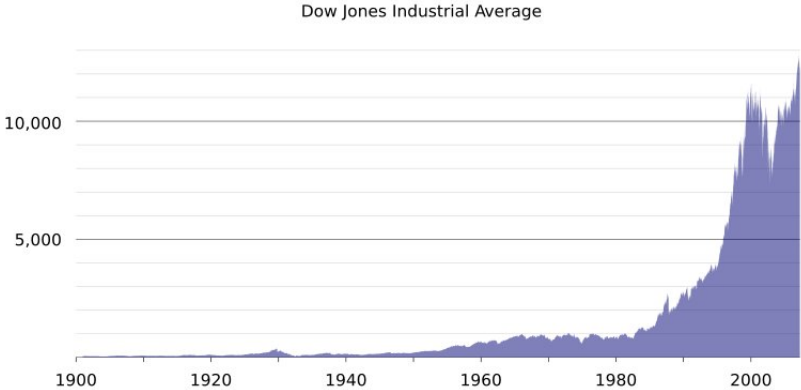
## Study Tips:

- ▶ Answer questions about asymptotics.
- ▶ Know what work the computer does and how to program it for the operation implementations covered in the course: selection and merge sort, binary search in a sorted array, search/insertion/deletion in an unordered array and linked list.
- ▶ Figure out how many of some steps are done for a given size data structure.

# Log-Log Graph Grid

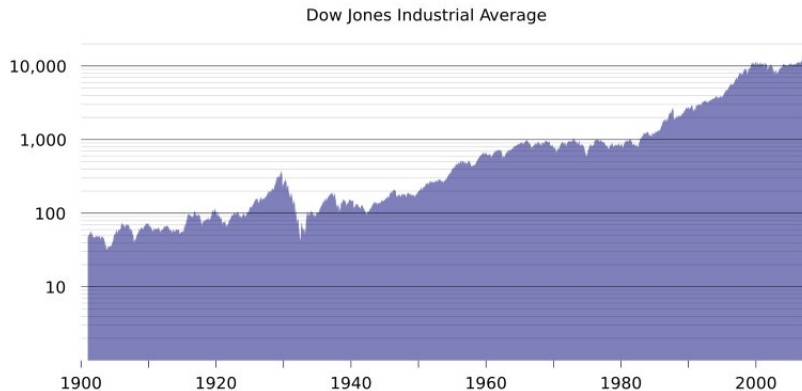


# Linear-Linear Plot



from Wikipedia

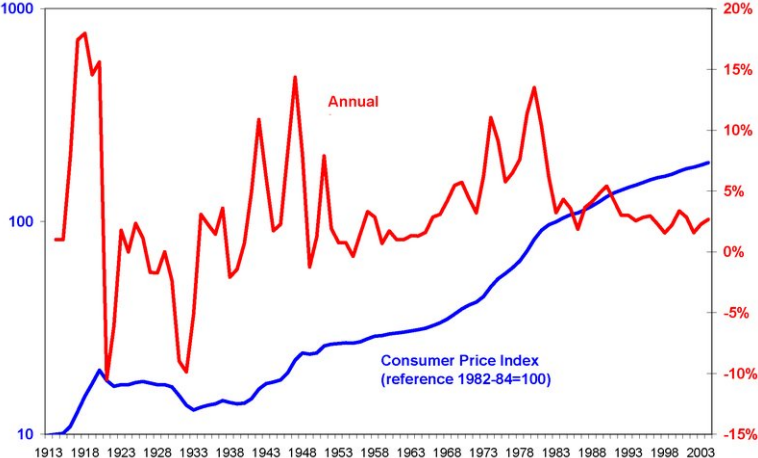
# Log-Linear Plot



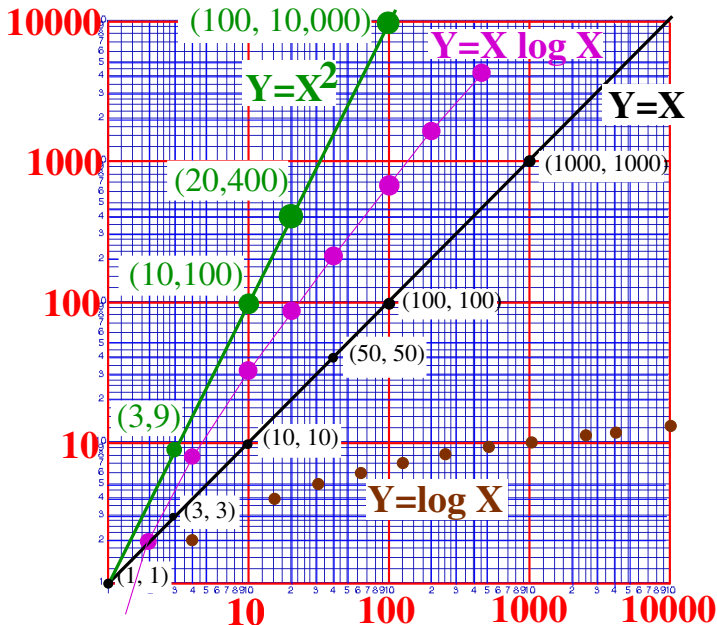
The Dow-Jones avg. grew at a roughly constant (exponential) rate throughout the last century.

from Wikipedia

# Consumer Price Index



from Wikipedia



For more..

## OOP

- ▶ Take CSI445 this summer.
- ▶ Read Head First Java.
- ▶ Read Head First Design Patterns.

## Algorithms and Data Structures

- ▶ Take CSI503 (discuss w/ Prof. Ravi) this fall or CSI403 next spring.
- ▶ Take CSI210 this summer or fall before that if necessary.

- ▶ Q/A session: Wed AM 9:00-11:00 in the Lab.
- ▶ Final exam in **LC-21**: Thu May 10 10:30-12:30(+15min)
- ▶ Final exam: Closed book/computer, like midterm, **except** one sheet of notes (crib-sheet) will be allowed.  
Topics for review guidance:

# Topic List and Textbook Sections I

- ▶ (lecture) Problems with large numbers of solutions. Intro to asymptotics. (lab) Plotting on log-log paper.
- ▶ (lecture) Characters, binary, (A.7) C-strings, what's in and what terminates a C-string, stack overflow.
- ▶ (5.1, 5 problem, 6.5) Backtracking, Path finding, Breadth-first labelling to find shortest maze distances, All-path enumeration.
- ▶ (7) Queues. What that ADT is, how can you implement it, what it can be used for. Role in Unix pipelines, terminal/network input, breadth-first labelling.
- ▶ (8.7, lecture) Using STL iterators (Prof. Doug Lea called them “pointers on steroids”.) We showed how print an STL vector with them.

## Topic List and Textbook Sections II

- ▶ (6.6) Relationships between recursion, stacks, activation trees.  
(5.2) Recursive definition of expressions. Infix, prefix, postfix expressions. Evaluation and conversions. Expression trees. Precedence. (6.2, 6.4) Use of stacks for expression conversion and evaluation.
- ▶ (10.1) Recursive definitions of trees and expressions. Root, parent, child, left/right subtrees, level of a node, height of a tree.
- ▶ (5.3) Recursion and mathematical induction. (Lectures) Analyses of recursion: Induction (“faith-based”) and (2.1) activation record diagrams and trees. “Base cases” and recursive cases; why recursive calls must apply to smaller problems.

## Topic List and Textbook Sections III

- ▶ (2.3) Binary search in an array. (10.3 Don't read all that code!) Binary Search in a Binary Search Tree. (Lectures, some of 11.2) Heap-ordered trees. What is a binary search tree? What is a heap ordered tree? What is the difference? Finding the maximum and minimum in a binary search tree and in a heap.
- ▶ (10 and lectures) Some basic tree lore, tree applications, linked tree representation (proj6), making decisions with decision trees (with non-leaf nodes labelled with questions), ideas of taxonomy, parse and file or domain name trees; pre, post, and in-order tree traversals.
- ▶ (9.2, lectures, project, lab) The mergesort algorithm, the merge operation; implementing both, activation trees for mergesort, running times of mergesort and merge.

## Topic List and Textbook Sections IV

- ▶ (lectures) Function calling, activation and return operations, local lifetime (local extant, automatic) variables and where they live (“box traces” of 2.1). Compare to dynamic variables.
- ▶ (2) Intro. to recursion.
- ▶ (9.2, project) Selection Sort, its running time.
- ▶ (4) Linked lists, new/delete, other applications of linked data structures: (10) trees, (6) stacks and (7) queues.
- ▶ (3, 4) Arrays, partially filled arrays, pointers, Dynamic Arrays
- ▶ (1, 3) Specification, Design, etc. Interface/implementation separation “Walls”: How in UA CSI310 and Why? Classes, Separate CPP/H Files, What `#include` means, Build scripts.

# Why do asymptotics?

## Invariance

When a different speed computer runs the **same algorithm**, the asymptotic characteristic will be the **same**

## What's different?

The constant in the running time functions will be different.

New 2.5GHz computer:  $M_{\text{new}}(N) = (0.0000001\text{sec}) \times N \log(N)$   
(0.0000001sec. = 100nanoseconds)

Very Old 25Mhz computer  $M_{\text{old}}(N) = (0.00001\text{sec}) \times N \log(N)$   
(0.00001sec. = 10,000nanoseconds)

# Some Themes and Variations

## Data Structures implementing Sequences

- ▶ Arrays: fixed logical length, fixed size but partially filled, dynamically allocated but fixed size, dynamically allocated resizable.
- ▶ Linked Lists: Usually dynamically allocated.

# Themes and Variations Cont'd

## Variations of Sequences

- ▶ **Stack**: Make/recognize palendromes (who cares??),  
**PARSE/EVALUATE NESTED EXPRESSIONS**  
**MANAGE ACTIVATION RECORDS** for **recursive** and other functions.  
**MANAGE PARTIAL SOLUTIONS** for path-finding problems.
- ▶ **Queue**: Recognize palendromes with a stack too (don't need to know the middle)  
**BUFFER**: When you use `cin >> ...;` UNIX pipeline command; all over the Internet.  
**BREADTH-FIRST LABELLING**: To find shortest-length maze paths.  
**SIMULATIONS**: Time step or event driven.

# Themes and Variations Cont'd

## Linked Data Structures

- ▶ “Non-linear” **Tree** implementations.
- ▶ Data structure in realistic software: Objects that “Own” or contain a variety of other objects. Bridge project 1: A Game has a Deck and 4 Hands.

# Function Activation, Execution Stack, Recursion, Trees

- ▶ Automatic aka local extant or lifetime variables (declared inside C++ blocks or non-reference function parameters): DIFFERENT VARIABLES for each activation.
- ▶ Basic recursion: Factorial, list-printing, tree-traversing functions.
- ▶ Please draw the TREE of ACTIVATIONS for one execution of mergesort.
- ▶ **Expression Trees**: For math. expressions and for html documents.
- ▶ Recursively defined structure for (1) Binary Trees and for (2) Expressions.
- ▶ Please draw the TREE of ACTIVATIONS for evaluating an expression tree. And/or the STACK used for evaluating a postfix expression and/or an expression tree. Maybe a new variation..

# Storage Management in Programming

## Local Lifetime/extent, automatic

- ▶ Allocated on stack. (one factor in of stack overflow attacks).
- ▶ Come and go with each **function activation** and **its return**.
- ▶ How to program and use them in C++: Declare (non-static) in a block or as a (non-reference) parameter.
- ▶ NEVER return `&LocExtVar`.

## Dynamically Allocated

- ▶ Allocated in the free store, sometimes called “heap”.
- ▶ Program controlled by executing `new ...`.
- ▶ Why/how pointers are needed for this in C++.
- ▶ Deallocated by `delete ...` in C++. Memory leak issues.

# Two more kinds of storage lifetime..

## Static Lifetime

- ▶ Storage allocated when the **process** (one run of the program) begins and continues until it dies.
- ▶ You get it in C++ by declaring the variable **OUTSIDE** of any block or class, by declaring it **INSIDE** a block with the `static` qualifier, or by declaring it `static` within a class.

## Persistent Storage—very new as a language feature

- ▶ Persists between **different** program runs or processes.
- ▶ Usually it is a disk file, a file shared over a network, or a database server.
- ▶ Some programming languages or libraries provide persistently stored objects that programmers access just like any other objects.

## Observation:

Technologies and practices (like files) known and used informally by programmers for a long time are becoming formalized and explicitly embedded in programming languages and systems (like persistent objects).

# Object Oriented Programming

## Data Abstraction

- ▶ Object Oriented Languages help (and sometimes force) people express interface-implementation separation **explicitly** in the code they write.
- ▶ A (non-static) method (abstract operation) is **always** activated ON or FOR a particular object. So such operations are always tied to data.
- ▶ The object for each method activation is pointed to by the `this` pointer in code within the body of the method. Useful for debugging.
- ▶ For a data member `D` or function member `F()`, it can be referred to or invoked with `this->D`, just `D`; or `this->F()`, just `F()` within the method body.

OOP = Data abstraction plus **polymorphism** (like overloading of `<<` in `cout << ...`, which method used to format output **DEPENDS** on the data type of `...`) plus **inheritance**.

# Evaluating and Choosing

## Abstract Data Type TABLE or DICTIONARY

- ▶ Stores a collection of data items.
- ▶ Each data item has a (usually) unique **key** value plus (usually) other data.
- ▶ ADT operations include DELETE or RETRIEVE the entire data item given the key value (or fail when it is not found.) Also traverse all the items in sorted search key order.

## Simple facts

- ▶ Very useful and popular. Chief abstraction for databases (take CSI410-411 this summer for more info!).
- ▶ Easy to implement with any list implementation if you don't care about performance.

## Implementation choices and analyses

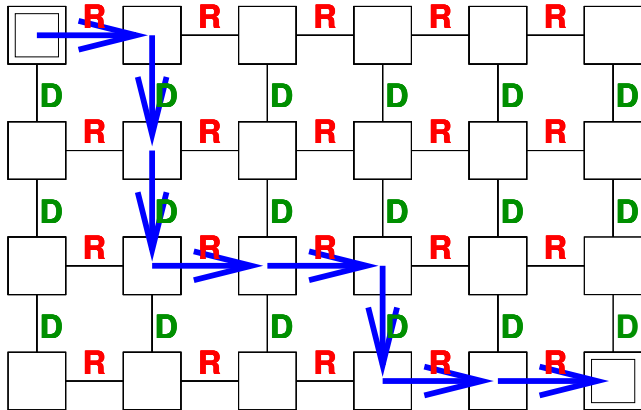
	<u>Insertion</u>	<u>Deletion</u>	<u>Retrieval</u>	<u>Traversal</u>
Unsorted array	$O(1)$	$O(N)$	$O(N)$	$O(N)$
Unsorted links	$O(1)$	$O(N)$	$O(N)$	$O(N)$
Sorted array	$O(N)$	$O(N)$	$O(\log N)$	$O(N)$
Sorted links	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Binary Search Tree	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(N)$

(from Carrano p. 602. Notes: Assume arrays are partially filled and dynamically expandable. Assume the Binary Search Tree is kept balanced.)

Hashing  $O(1)$   $O(1)$   $O(1)$   $O(N)$

(These figures are for **Average** performance assuming the table is not mostly full.)

**R D D R R D R R** String of 3 Ds and 5 Rs  
 that codes one shortest path.



**How many shortest paths? Answer: the number of ways of picking the 3 positions for Ds out of 8 total.**

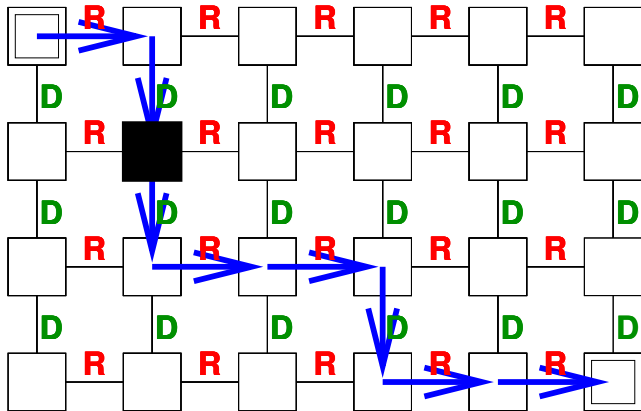
That is  $\binom{8}{3} = \frac{8 \times 7 \times 6}{3 \times 2 \times 1} = 56$

## Sometimes, you can figure out a count

- ▶ Number of shortest paths in that 6x4 maze =  $\binom{8}{3}$
- ▶ What if one square is forbidden? Answer =

$$\binom{8}{3} - (\text{\#Shortest paths through the forbidden square})$$

**R D D R R D R R** String of 3 Ds and 5 Rs that codes one shortest path.



Number of forbidden paths =  $\binom{2}{1} * \binom{6}{2} = 2 * 15 = 30$

Number of OK paths =  $56 - 30 = 26$

# Two forbidden squares?

## Two cases

- ▶ The two squares S1, S2 are never in the same shortest path.

$$\binom{8}{3} - (\# \text{ through S1}) - (\# \text{ through S2})$$

- ▶ The two squares can be in the same shortest path.

$$\binom{8}{3} - (\# \text{ through S1}) - (\# \text{ through S2}) + (\# \text{ through BOTH})$$

- ▶ This is an example of the “principle of inclusion-exclusion” (from CSI210 & MAT367).

# Variants with Different Complexities

## Four Separate Problems:

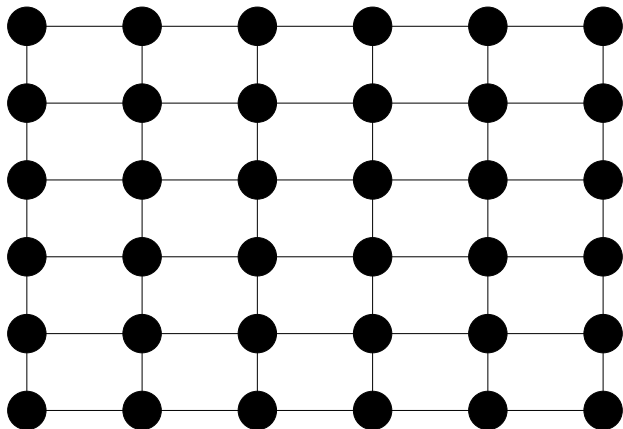
1. Given a maze, find/list/count ALL simple paths from start to goal.
2. Given a maze, find ONE simple path from start to goal, or report that there isn't any.
3. Given a maze, find ONE simple path THAT VISITS ALL THE VERTICES, or report there isn't any. (#3 is the "Hamiltonian Path Problem")
4. Given a maze, find ONE simple path of shortest length, or report there isn't any.

How many large can the count of #1 be?

Is it easier to solve #2 than by trying to solve #1?

A million dollar question: "Is #3 harder than #2?" (More formally, "Does  $P=NP$ ???" If you can answer definitively, you win a Clay Institute of Mathematics Prize.)

What about #4?



- ▶ The start and goal vertices are the upper left and lower right ones.
- ▶ Let  $N$  be the number of “down” steps needed to go from the top to the bottom row. So, in terms of Project 7,  $N = n - 1$ . The figure shows the graph with  $N = 5$ .
- ▶ Some but not all of the solutions are formed by  $N$  right and  $N$  down steps. ONLY. We illustrated the first such solution: 5 down steps followed by 5 right steps.
- ▶ We write the binary string 00101 down the left side. The following rule specifies how a length 5 string like this determines one solution:
- ▶ When we are at each row, take one step down if the bit is 0; and take one step left followed by one step down if the bit is 1. When we reach the bottom row, take right only steps to reach the goal.
- ▶ So string 00101 corresponds to DDRDDRDRRR. String 01110 corresponds to DRDRDRDDRR.

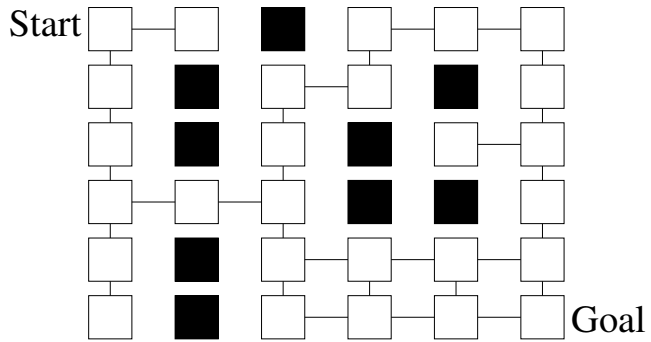
- ▶ This rule provides a 1-1 function from length 5 (generally  $N$ ) binary strings to solution paths. It not an **onto** function; there are (many) more solutions than the ones from this rule.
- ▶ The number of length  $N$  0-1 strings is  $2^N$ . For our  $N = 5$  example, using this rule demonstrates that there are over 32 different solutions.
- ▶ Consider  $N = 1000$ . The description of the maze can fit on a floppy disk. However, the number of solutions, more than  $2^{1000}$ , the computer must print is way too big to be computed for mortals! (more than the number of protons that can be packed into the known universe, etc.)

## Traverse and Search

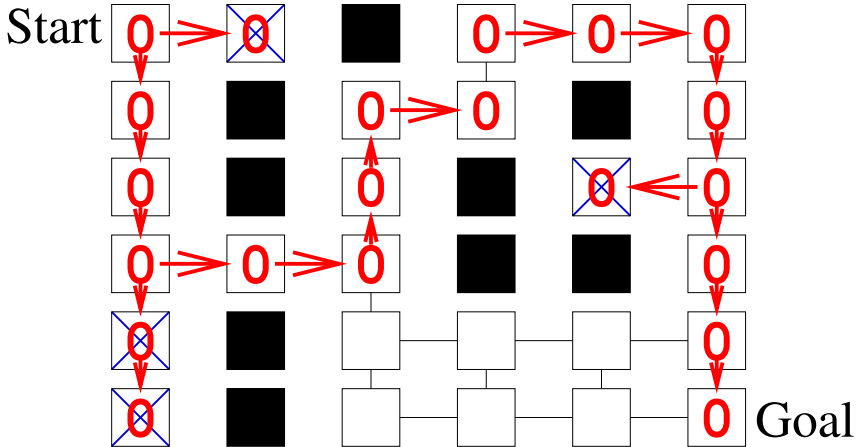
Let us compare problems #1 and #2. Graph traverse and search algorithms can solve problem #2, to find one solution path if any and print “none” if there are none, doing less work than the backtracking algorithm from the project.

## Labelling Search

- ▶ We sketched the operation of a “labelling” type search algorithm. The main idea is to put and retain a “mark” on each vertex as soon as we determine that it can be reached from the start vertex.
- ▶ This way, the algorithm to find one path or determine that there is none, would take a number of steps proportional to  $N^2$  instead of getting finding as many as  $2^N$  paths.



# Depth-first Labelling Search



here are the squares in the order they are inserted in the QUEUE  
 0,0 0,1 1,0 2,0 3,0 3,1 4,0 3,2 5,0  
 2,2 4,2 1,2 4,3 5,2 1,3 4,4 5,3 0,3 4,5  
 5,4 0,4 3,5 5,5 DONE!

## Breadth-first Labelling Search

