SC1007
Data Structures and Algorithms

Week 13: Revision

Dr Liu Siyuan (syliu@ntu.edu.sg)
N4-02C-117a

Office Hour: Mon & Wed 4-5pm

To Simplify……

- Given an algorithm
  - Derive a function f with respect to problem size
  - Compare against g
    - O(g(n)), Ω(g(n)), Θ(g(n))

g(n)
1
$\log_2 n$
n
$n\log_2 n$
n^2
n^3
2^n
n!

$\lim_{n\to\infty} \frac{f(n)}{g(n)}$	$f(n) \in O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \in \Theta(g(n))$
0	✓		
$0 < C < \infty$	✓	✓	✓
∞		✓	

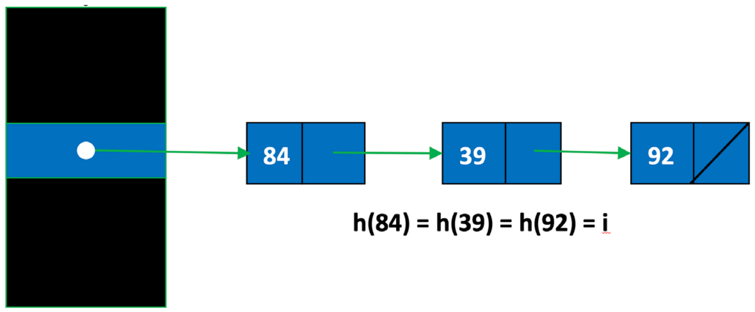
Hashing

- A typical space and time trade-off in algorithm
- To achieve search time in O(1), memory usage will be increased
- Each key is mapped to a unique index (hash value)

  hash function: {all possible keys} → {0, 1, 2, …, h-1}
- The array is called a hash table
- Each entry is called a hash slot
- When multiple keys are mapped to the same hash value, a collision occurs
- load factor $\alpha = \frac{n}{h}$

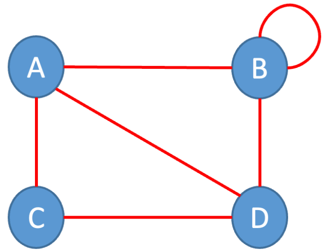
Collision Resolutions

- Closed Addressing Hashing

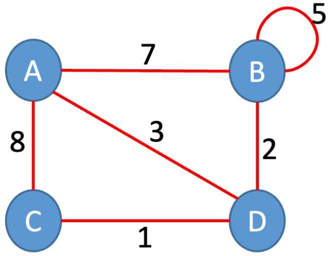


- Open Addressing Hashing
  - Linear Probing: $H(k,i) = (H(k) + i) \; mod \; h, where \; i \in [0, h-1]$
  - Quadratic Probing: $H(k,i) = (H(k) + c_1 i + c_2 i^2) \; mod \; h, where \; i \in [0, h-1]$
  - Double Hashing: $H(k,i) = (H(k) + iD(k)) \; mod \; h, where \; i \in [0, h-1]$

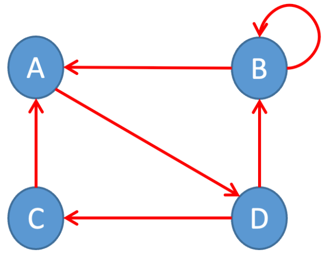
Graph Terminology



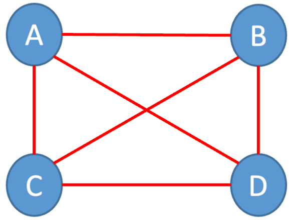
Undirected Graph



Weighted Graph



Directed Graph

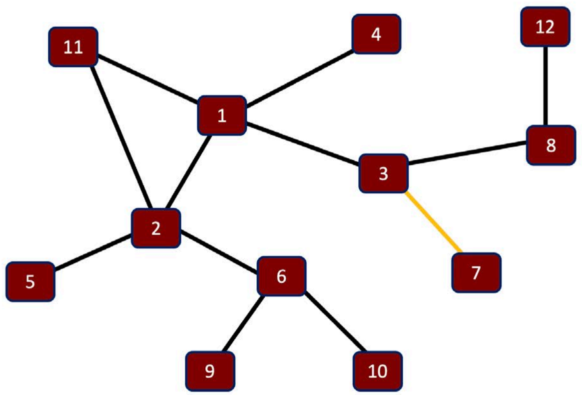


Complete Graph

- A graph $G = (V,\ E)$
  - A set V of vertices
  - |V| is the number of vertices
  - A set E of edges that connect the vertices
  - Degree of a vertex is the number of edges incident to it
- An undirected graph is connected if there is a path from any vertex to any other vertex.
- A directed graph is strongly connected if there is a path from any vertex to any other vertex. A path is a sequence of nodes connected by edges. A simple path is a path that does not repeat any nodes.
- A path is a cycle if it starts and ends in the same node. A simple cycle is one containing at least three vertices and repeats only the first and last nodes.

Graph Representation

- Adjacency Matrix



	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	1	1	0	0	0	0	0	0	1	0
2	1	0	0	0	1	1	0	0	0	0	1	0
3	1	0	0	0	0	0	1	1	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0	1	1	0	0
7	0	0	1	0	0	0	0	0	0	0	0	0
8	0	0	1	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	1	0	0	0	0	0	0
11	1	1	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	1	0	0	0	0

- Adjacency List

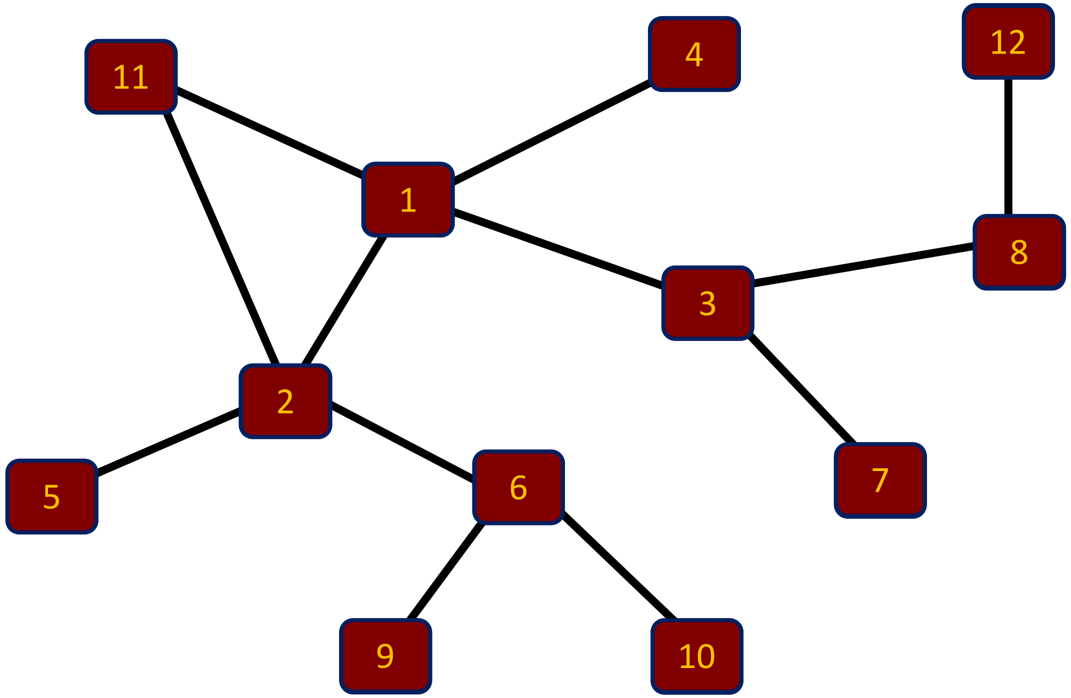
1	→ 2 → 3 → 4 → 11
2	→ 11 → 1 → 5 → 6
3	→ 1 → 8 → 7
4	→ 1
5	→ 2
6	→ 10 → 9 → 2
7	→ 3
8	→ 12 → 3
9	→ 6
10	→ 6
11	→ 2 → 1
12	→ 8

Traversal of Graphs

- The traversal problem: check all nodes once and only once
- To traverse a graph, we can apply:
  - Breadth-first Search
  - Depth-first Search

BFS

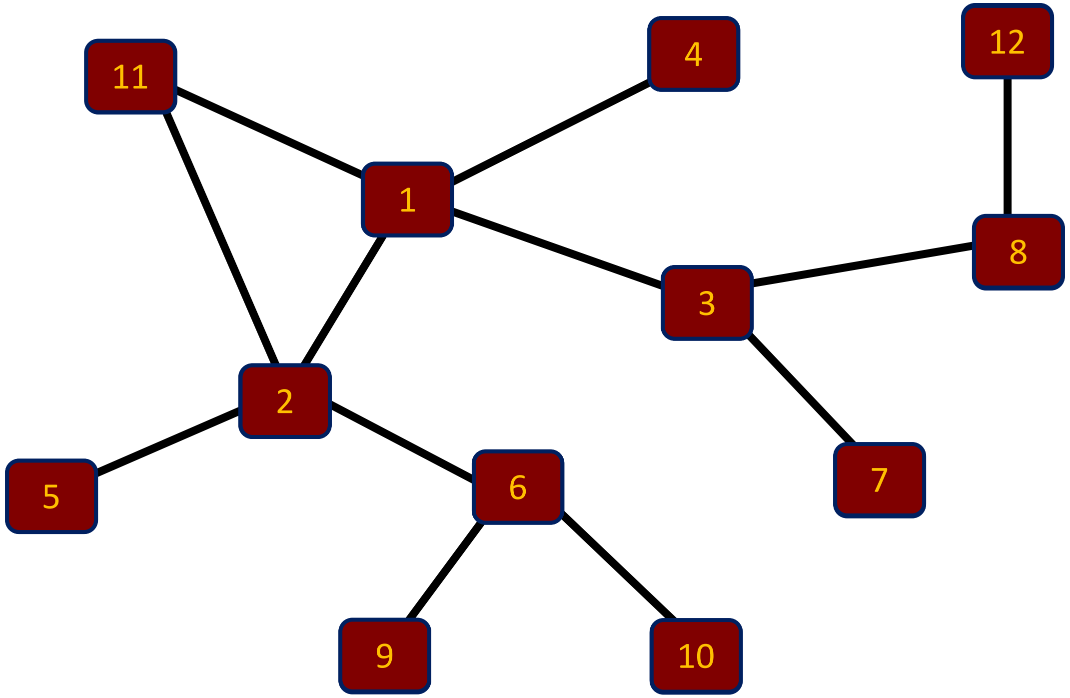
Explores the edges directly connected to a vertex before visiting vertices further away



function BFS(Graph G, Vertex v)
    create a Queue, Q
    enqueue v into Q
    mark v as visited
    while Q is not empty **do**
        dequeue a vertex denoted as w
        for each unvisited vertex u adjacent to w **do**
            mark u as visited
            enqueue u into Q
        end for
    end while
end function

DFS

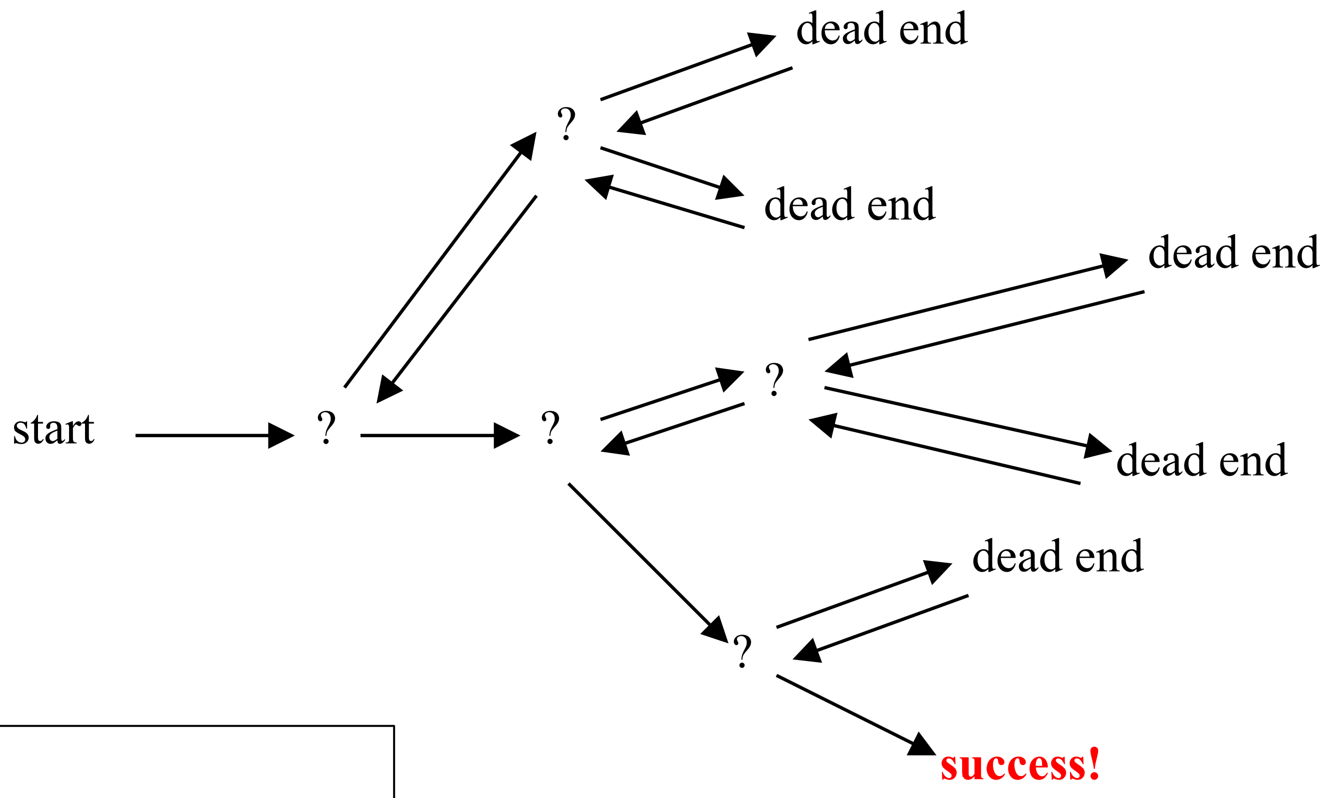
Explores along a path from vertex v as deeply into the graph as possible before backing up




```
function DFS(Graph G, Vertex v)
    create a Stack, S
    push v into S
    mark v as visited
    while S is not empty do
        peek the stack and denote the vertex as w
        if no unvisited vertices are adjacent to w then
            pop a vertex from S
        else
            push an unvisited vertex u adjacent to w
            mark u as visited
        end if
    end while
end function
```

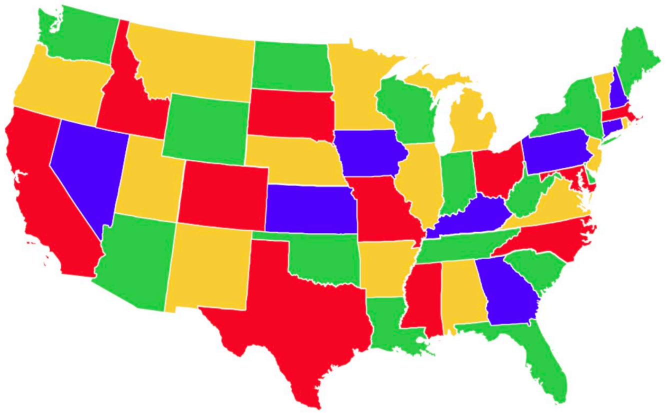
Backtracking

- A methodical way of trying out various sequences of decisions, until you find one that “works”



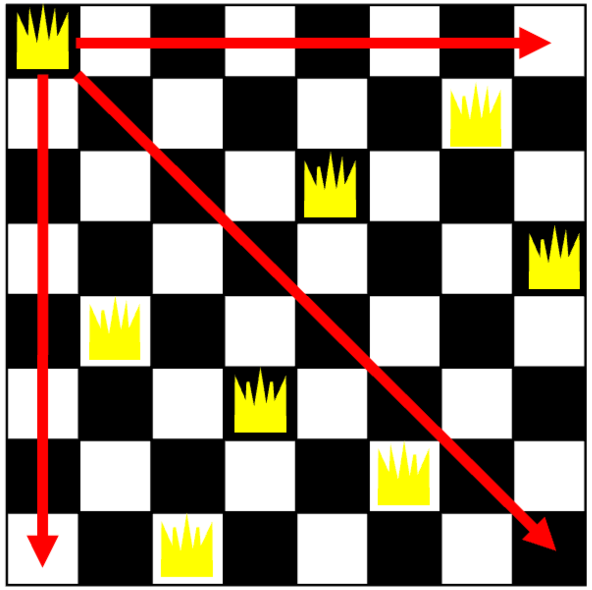

```
Backtracking(N)
        If N is a goal node, return “success”
        Else if N is a leaf node, return “failure”
        For each child C of N,
                If Backtracking(C) == “success”
                        Return “success”
        Return “failure”
```

Backtracking: Coloring problem



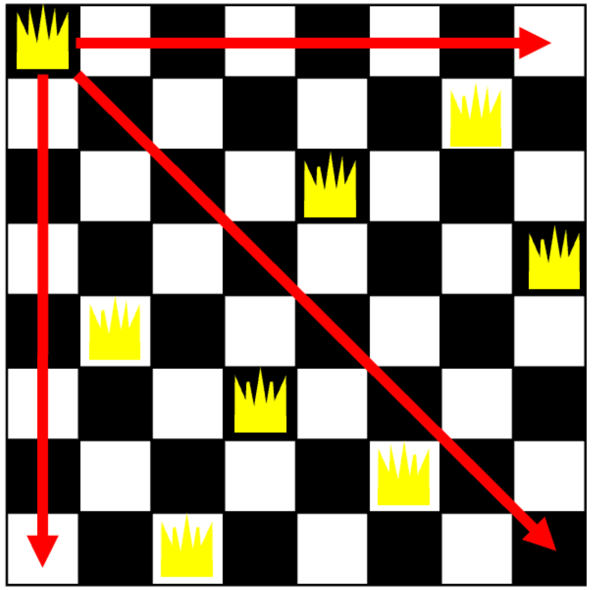
```
bool mColoring(int colors, int color[], int vertex){
   if (vertex == V) //when all vertices are considered
      return true;
   for (int col = 1; col <= colors; col++) {
      if (isValid(vertex,color, col)) { //check whether color col is valid or not
         color[vertex] = col;
         if (mColoring (colors, color, vertex+1) == true) //go for additional vertices
            return true;
         color[vertex] = 0;
      }
   }
   return false; //when no colors can be assigned
}

bool isValid(int v,int color[], int c){   //check whether putting a color valid for v
   for (int i = 0; i < V; i++)
      if (graph[v][i] && c == color[i])
         return false;
   return true;
}

int main(){
   int colors = 3; // Number of colors
   int color[V]; //make color matrix for each vertex
   for (int i = 0; i < V; i++)
      color[i] = 0; //initially set to 0
   if (mColoring(colors, color, 0) == false) { //for vertex 0 check graph coloring
      printf("Solution does not exist.");
   }
   printf("Assigned Colors are: \n");
   for (int i = 0; i < V; i++)
      printf("%d ", color[i]);
   return 0;
}
```

Backtracking: Eight Queens Problem



function NQUEENS($Board[N][N]$, $Column$)
    if Column >= N **then return** true ▷ Solution is found
    else
        for $i \leftarrow 1, N$ **do**
            if Board[i][Column] is safe to place **then**
                Place a queen in the square
                if NQueens($Board[N][N]$, $Column + 1$) **then return** true ▷ Solution is found
                end if
                Delete the queen
            end if
        end for
    end if
return false ▷ no solution is found
end function

Backtracking: Eight Queens Problem



```
bool solveNQ(int board[N][N], int col)
{
    // Base case: If all queens are placed
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQ(board, col + 1))
                return true;

            board[i][col] = 0; // BACKTRACK
        }
    }
    return false;
}
```

```
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}
```

Dynamic Programming

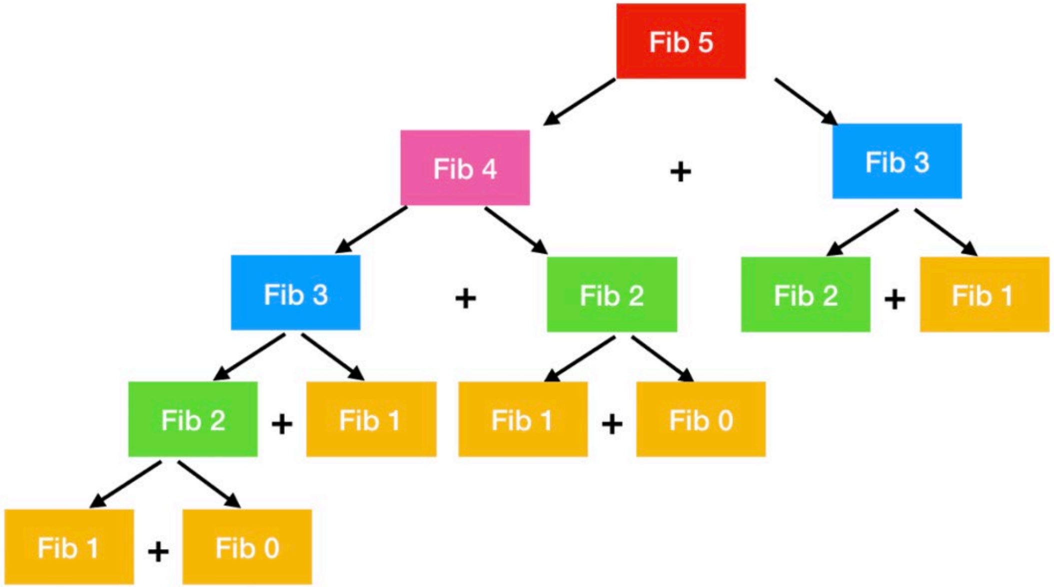
$\boxed{\text{Dynamic Programming} = \text{Recursion} + \text{Memoization}}$

- Recursion: problem can be solved recursively
- Memoization: Store optimal solutions to sub-problems in table (or memory or cache)

Fibonacci

$F(0) = 0, F(1) = 1$

$F(n) = F(n-1) + F(n-2)$ *with* $n>=2$




```
Fib(n)
{
   if (n == 0)
      return 0;

   if (n == 1)
      return 1;

  return Fib(n-1) + Fib(n-2);
}
```

Fibonacci: DP Top-down approach

```
Fib(n)
{
   if (n == 0)
          M[0] = 0; return 0;
   if (n == 1)
          M[1] = 1; return 1;

   if (M[n-1] == -1)                //F(n-1) was not calculated
          M[n-1] = Fib(n-1)         //calculate F(n-1) and store in M

   if (M[n-2] == -1)                //F(n-2) was not calculated
          M[n-2] = Fib(n-2)         //calculate F(n-2) and store in M

  M[n] = M[n-1] + M[n-2]
  return M[n];
}
```

Fibonacci: DP Bottom-up approach

```
Fib(n)
{
   M[0] = 0;
   M[1] = 1;

   int i = 0;
   for (i = 2; i<=n; i++)
         M[i] = M[i-1] + M[i-2];
   return M[n];
}
```

Other examples of DP

- Rod Cutting
  - Time complexity: $\Theta(2^n)$-> $\Theta(n^2)$
- 0/1 Knapsack
  - Time complexity: $\Theta(2^n)$-> $\Theta(nC)$