

## Memory Functions

- Generic strategy for dynamic programming
- Recursive functions
  - simple code
  - no need for tables or auxiliary variables
  - slow because of duplication of effort
- Bottom-up dynamic programming
  - faster
  - more complex code
  - may compute intermediate values that are never needed
- Use a memory function
  - same interface as regular recursive function
  - stores all values in table
  - does recursive call only if value not known yet
  - top down
  - best of both worlds! No unnecessary computations!

© 2005-06. 324-1-100

## Knapsack with Memory Function

```
/** Return the value of the best possible packing.*/
int knapsack( int[] maxKnown, Item[] items, int capacity ) {
    int maxValue = 0;
    if( maxKnown[ capacity ] != UNKNOWN ) {
        return maxKnown[ capacity ] ;
    }
    for( int i = 0; i < capacity; i++ ) {
        int space = capacity - items[ i ]. size;
        if( space >= 0 ) {
            int value = knapsack( maxKnown, items, space )
                + items[ i ]. value;
            if( value > maxValue ) {
                maxValue = value;
            }
        }
    }
    maxKnown[ capacity ] = maxValue;
    return maxValue;
}
// after [Sedgewick 2003]
```

© 2005-06. 324-1-101

## Game Graphs

- Known from Artificial Intelligence
- Present all steps in a game
- Who-takes-the-last-coin wins
- Wolf, sheep, cabbage

© 2005-06. 324-1-102

## Traversing Trees

- Preorder
- Postorder
- Inorder
  
- Runtime:  $O(n)$
- Easily done recursively

© 2005-06. 324-1-103

## Traversing Graphs

- Depth-first
  - go to neighbor
  - go to neighbor of neighbor
  - recursively
  - need to remember which nodes visited already
- Breadth-first
  - put all neighbors in a queue
  - process each node
  - put its neighbors in queue
  - need to remember which nodes visited already
- Runtime
  - $O(\max(e, n))$
- Both can be used to create a spanning tree

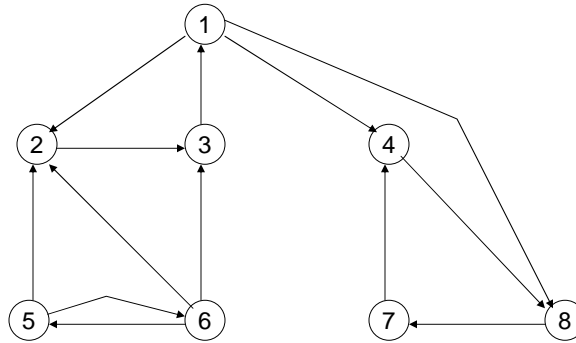
© 2005-06. 324-1-104

## Breadth-First Search

```
bfs( Node n ) {
    Queue< Node > q = new Queue< Node > ;
    Set< Node > marked = new Set< Node > ;
    marked. add( n );
    q. enqueue( n );
    while( ! q. isEmpty() ) {
        Node u = q. getFirst();
        // do something with u;
        q. dequeue();
        for( Node w: u. getNeighbors() ) {
            if( ! marked. contains( w ) ) {
                marked. add( w );
                q. enqueue( w );
            }
        }
    }
}
```

© 2005-06. 324-1-105

## Graph Traversal Example



© 2005-06. 324-1-106

## Infinite Graphs

- Graph with infinitely many nodes and edges
- Cannot be searched completely
- But: can be searched heuristically
  - success depends on search strategy used
  - breadth-first often better!
  - why?

© 2005-06. 324-1-107

## **Implicit Graphs**

- Graph is too large or too difficult to build
  - we build only that part of the graph we are working on
  - and shift it as needed
- Backtracking
  - finding a solution through depth-first search
  - if a branch doesn't work, back up
  - each path is considered a possible solution
  - partial parts may be discarded

© 2005-06. 324-1-108