Structure of this week's classes

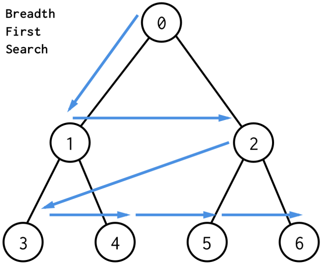
BFS vs. DFS

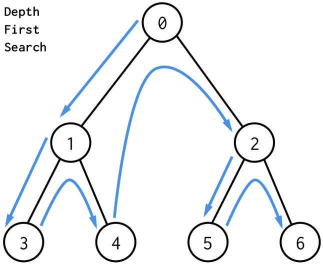
DFS - 4 Step Process

4-Step Process: Counting Isosceles Triangles in a Binary Tree

DFS BackTracing - N Queens

DFS vs. BFS

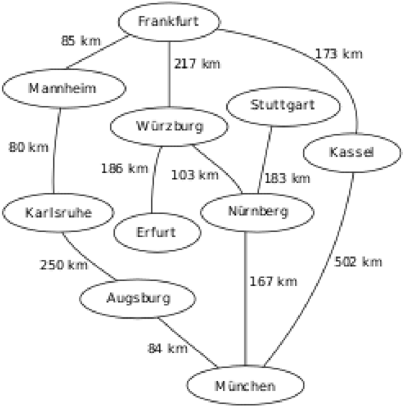




Difference DFS vs. BFS

- BFS is more suitable when the solution is near the root (more "optimistic"), DFS is more suitable when the solution can be anywhere in the tree (more "pessimistic")
- DFS is more suitable for game/puzzle problems, i.e., exploring all paths for finding the optimal/sum/all solution
- BFS is more suitable for shortest path problems
- Time complexity: $O(|V|+|E|)$, $O(|V| + |E|)$, space complexity: $O(W)$, $O(h)$
- For more of BFS/DFS difference please see `https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/`.

BFS for Shortest Path



- Finding the shortest path from Frankfurt to any cities
- Dijkstra's shortest path algorithm (using Greedy approach)
- A well explained tutorial for Dijkstra's algorithm: `https://www.youtube.com/watch?v=pVfj6mxhdMw` (starting 2:08)

The 4-Step Process for DFS in Binary Tree

- What information should the children return to the parent?
- What information should the parent pass on to the children?
- Handle the terminal nodes
- Update the optimal/complete/sum solution

Last Lecture: Valid BST

```
public boolean recursive_is_bst(Node<E> root, E lower_bound,
E upper_bound) {
    if (root == null) return true;

    if (root.value.compareTo(lower_bound) <= 0 ||
    root.value.compareTo(upper_bound) >= 0) return false;

    return recursive_is_bst(root.l_child, lower_bound,
    root.value) && recursive_is_bst(root.r_child, root.value,
    upper_bound);
}
```

- In last lecture, we talked about the above algorithm for checking whether a binary tree is a valid BST
- We can rewrite the above method as the method in the next page

The 4-Step Process: Valid BST

```
boolean is_valid_bst = true;

public void recursive_is_bst2(Node<E> root, E lower_bound
, E upper_bound) {

    if (root == null) return;

    if (root.value.compareTo(lower_bound) <= 0 ||
    root.value.compareTo(upper_bound) >= 0)
       is_valid_bst = false;

    recursive_is_bst2(root.l_child, lower_bound, root.value);
    recursive_is_bst2(root.r_child, root.value, upper_bound);
}
```

`recursive_is_bst` is more efficient than `recursive_is_bst2`, Why?

The 4-Step Process: Valid BST

- What information should the parent pass on to the children?
  - lower bound and upper bound
- Handle the terminal nodes

```
 if (root == null) return;
```

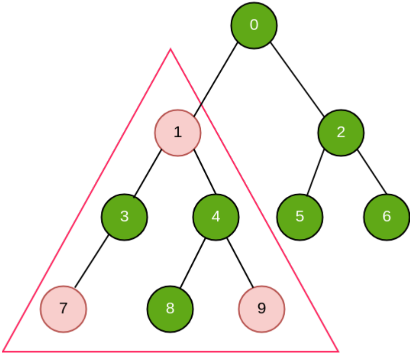
- Update the optimal/complete/sum solution

```
if (root.value.compareTo(lower_bound) <= 0 ||
root.value.compareTo(upper_bound) >= 0)
   is_valid_bst = false;
```

Counting Isosceles Triangles in a Binary Tree

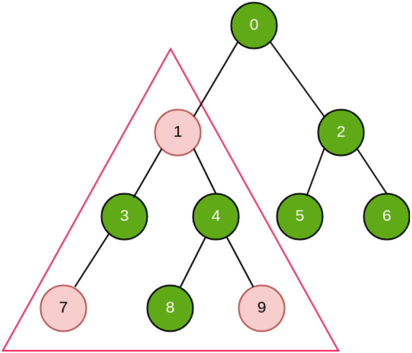
- A isosceles triangle contains three nodes
- Two nodes are on the same level
- The third node is the first two node's LCA (lowest common ancestor), and
- The three nodes must form a triangle

Counting Isosceles Triangles in a Binary Tree



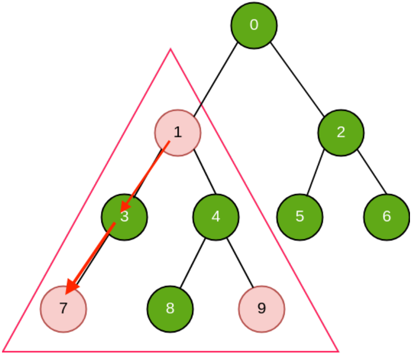
Counting Isosceles Triangles in a Binary Tree

$$count = \sum_{node\ n} count(n \text{ as root})$$



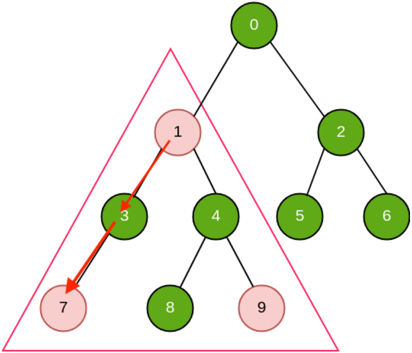
How to Count *count*(*n as root*)

- left_path_len: length of path that starts from the root and keeps going left;
- right_path_len: length of path that starts from the root and keeps going right;



How to Count *count*(*n as root*)

$$count(n \text{ as } root) = min(n.left\_path\_len, n.right\_path\_len)$$



Ideas - 4 Steps

- Step 1: What is the output of the recursive function?
  - i.e., after we are done with the left child, what information should it return to the parent?
- Step 2: What information should the parent pass to the children?
- Step 3: How to handle the terminal cases?
- Step 4: Updating the optimal solution at each node

Step 1: What to return to parent

How to update left_path_len and right_path_len?

- n.left_path_len = 1 + n.l_child.left_path_len
- Therefore, set left_path_len as the output

```
public Integer count_iso_triangle(parent)
    ...
    child_left_path_len = count_iso_triangle(parent.l_child);
    ...
    return child_left_path_len + 1;
}
```

Step 1: What to return to parent

How to update left_path_len and right_path_len?

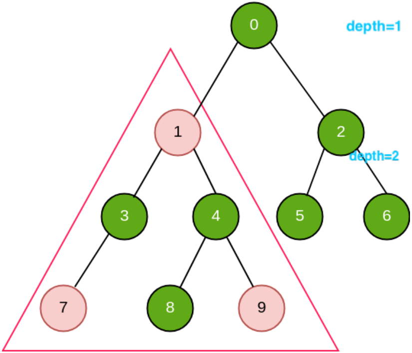
- set left_path_len and right_path_len as the output
- Java does not allow two outputs
- Return a Pair<Integer> object

```
protected class Pair<E>{
    E value1;
    E value2;

    protected Pair(E value1, E value2) {
        this.value1 = value1;
        this.value2 = value2;
    }
}
```

Step 2: What to pass to children?

Nothing, because *count*(n *as root*) does not depend on any recursive information above node n, e.g., depth of n




```java
public Pair<Integer> count_iso_triangle(Node<Integer> root) {
...
}
```

Step 3: Handling terminal cases

- If node is null, return 0, 0
- If node does not have left child, return 0 for left_path_len
- If node does not have right child, return 0 for right_path_len

Step 4: Updating the Optimal Solution

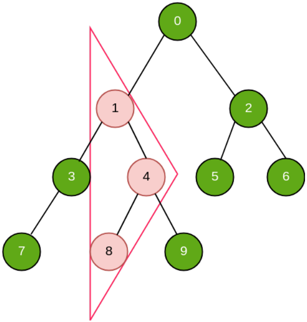
At each node, update $count(n \ as \ root)$ with $min(n.left\_path\_len, n.right\_path\_len)$

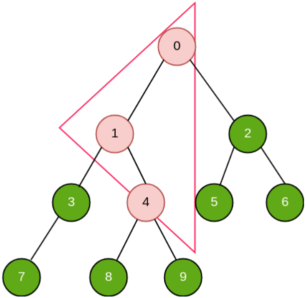
```
total_iso_triangle += Math.min(l_depth, r_depth);
```

Run test code: count_iso_triangle

HW4 Part 1: Iso Triangle 2

Count the number of second type of iso triangles:





BFS vs. DFS

DFS - 4 Step Process

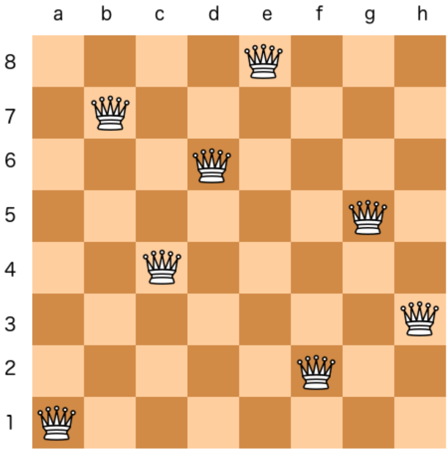
4-Step Process: Counting Isosceles Triangles in a Binary Tree

DFS BackTracing - N Queens

DFS BackTracing - N Queens

- DFS beyond binary tree
- So far we have been seeing examples where the solution is based on node values in the tree
- DFS can be used for playing games, where the solution is based on a series of decisions, where one decision can depends on another
- Example: N Queens

N Queens



- Chess, 8x8 matrix
- No two queens can be on the same row/column/diagonal.
- Print all the solutions

N Queens

```
1 5 8 6 3 7 2 4
1 6 8 3 7 4 2 5
1 7 4 6 8 2 5 3
1 7 5 8 2 4 6 3
2 4 6 8 3 1 7 5
2 5 7 1 3 8 6 4
2 5 7 4 1 8 6 3
2 6 1 7 4 8 3 5
2 6 8 3 1 4 7 5
2 7 3 6 8 5 1 4
2 7 5 8 1 4 6 3
2 8 6 1 3 5 7 4
3 1 7 5 8 2 4 6
3 5 2 8 1 7 4 6
3 5 2 8 6 4 7 1
3 5 7 1 4 2 8 6
3 5 8 4 1 7 2 6
3 6 2 5 8 1 7 4
3 6 2 7 1 4 8 5
3 6 2 7 5 1 8 4
3 6 4 1 8 5 7 2
3 6 4 2 8 5 7 1
3 6 8 1 4 7 5 2
```

- 92 solutions
- Every solution consists of 8 numbers
- 1586...: place the following 8 queens: (1, 1), (2, 5), (3, 8), ...

N Queens - DFS

```
/**
 * Recursive algorithm: for each column, try searching
 * to place the queen at each row
 * @param board
 * @param col
 */
public void try_place_queen(int board[][], int col) {
    // if reaching the terminal, it means no violation
    // therefore update the optimal solution
    if (col >= N) {
        printSolution(board);
        return;
    }
```

Checking validity of partial solution

```
    /* Search by col: try placing the queen at col
     * on row = i */
    for (int i = 0; i < N; i++) {
        /* check the validity of the partial solution
         * if it's safe, continue the search, otherwise,
         * prune the partial solution and search the next solution
         */
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            /* for the next col, enumerate the row number */
            try_place_queen(board, col + 1);
            board[i][col] = 0; // BACKTRACK
        }
    }
}
```

Checking validity of partial solution

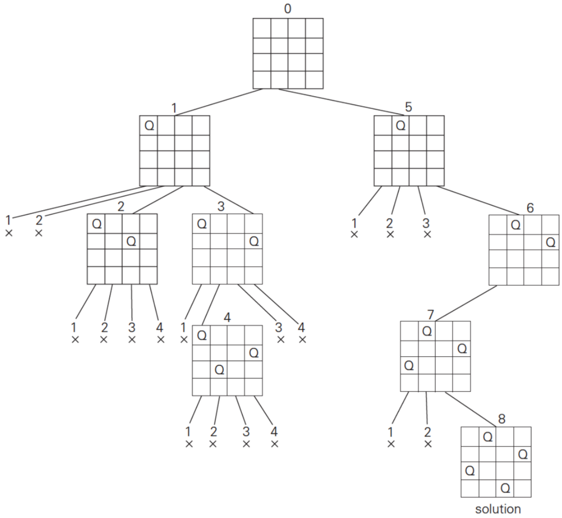
```java
/** check whether the existing partial solution allow us
 * place the queen at position (row, col)
 * @param board
 * @param row
 * @param col
 * @return
 */
public boolean isSafe(int board[][], int row, int col)
{
    int i, j;
    /* Check whether there are elements on the same row
     * There will not be elements on the same col,
     * because we are enumerating on the col
     */
    for (i = 0; i < col; i++)
        if (board[row][i] == 1)
            return false;
```

Checking validity of partial solution

```
    /* Check whether there are elements on the same
    upper diagonal */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1)
            return false;
    /* Check whether there are elements on the same
    ]lower diagonal */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j] == 1)
            return false;

    return true;
}
```

N Queens - DFS



DFS - Summarization

- Search within a *problem space* for the *optimal solution*: e.g., all iso triangles, all n-dimensional array that satisfy the NQueens definition

$$solution = argmax_{s' \in \mathcal{S}} score(s)$$

- Exhaustive search requires exponential time
- DFS saves time by *pruning*, e.g., rejecting partial solutions for NQueens that already violates the rule, do not proceed with deeper branches, instead backtrack

DFS - 3Sum

Given an array nums of n integers, are there elements a, b, c in nums such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero.

Solution: DFS