ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Module 10

Recursion

Dayu Wang

- **Recursive Algorithm**

  → A **recursive algorithm** solves a problem by breaking that problem into smaller subproblems, solving these subproblems, and combining the solutions.

  → An algorithm that is defined by repeated applications of the same algorithm on smaller problems is a recursive algorithm.

- **[Example 1] Exponential operation**

```
/** Calculates {x} ^ {y}.
    @param x: base
    @param y: exponent
    @return: result of {x} ^ {y}
*/
unsigned int power(unsigned int x, unsigned int y) {
    if (!y) { return 1; }
    else { return power(x, y - 1) * x; }
}
```

→ **Two parts** in a recursive algorithm:

1) Base case
2) Recurrence relation

- [Example 2] Binary Search

→ **Guess Number**

- The host writes down a number between 1 and 100 (inclusive). Initially, no one can see the number.
- The player guesses the number multiple times.

  In each guess, the host tells whether the secret number is greater than the secret number, less than the secret number, or bingo.
- In worst case, how many times the player needs to try to guarantee a bingo?

→ **Dichotomy**

Suppose there are 16 numbers sorted.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

First, we try '8'.

- If the secret number is less than 8, then the right half is eliminated.
- If the secret number is greater than 8, then the left half is eliminated.

1	2	3	4	5	6	7	8								

								9	10	11	12	13	14	15	16

								9	10	11	12	13	14	15	16

→ **Suppose the secret number is greater than 8, then what's the next step?**

- The next try should be "12", then another "half of the right half" will be eliminated.
- Repeat this step until the number is found.

→ **For 16 numbers, at most 5 guesses will guarantee the number to be found.**

For n numbers, at most $\lceil \log n \rceil + 1$ attempts will guarantee the number to be found.

→ **How to write the code of the algorithm above?**

- Re-define the problem

  Given a sorted vector of integers, find the index of a target number in the contiguous section from index `start` to index `end` (inclusive).

  If the target number does not appear in the section, return -1.

- **Write the recursive function.**

```
int search(const vector<int>& vec, int target, size_t i, size_t j) {
    if (i > j) { return -1; }  // Base case
    int mid = (i + j) / 2;  // Find the middle index.
    if (target < vec.at(mid)) {
        return search(vec, target, i, mid - 1);
    } else if (target > vec.at(mid)) {
        return search(vec, target, mid + 1, j);
    } else { return mid; }
}
```

The recursive function has an if statement that ends the recursion, called the base case.

After the base case, the rest contains recursive function calls.

→ **Four steps to create a recursive solution to solve problems:**

1) **Re-define the problem.**

   The problem you solve using recurrence relations may or may not be the same as the original problem.

2) **Write the base case(s).**
3) **Write the recurrence case(s).**
4) **Write a wrapper function.**


```
// Wrapper function
int search(const vector<int>& vec, int target) {
    return search(vec, target, 0, vec.size() - 1);
}
```

→ **Wrapper function converts the problem you re-defined back to the original problem.**

- **[Exercise] Base case and recurrence relation**

→ **Write down the base case(s) and recurrence relation(s) of the following algorithms:**

- **Calculate the factorial of a non-negative integer `n`.**
- **Reverse a string `s`.**
- **Test whether a string `s` contains lowercase English letters only.**

→ **Redefine the problem, write down the base case(s) and recurrence relation(s), and write a wrapper function for the following algorithms.**

- **Test whether a string `s` is a palindrome.**
- **Calculates the sum of all elements in a given vector of integers `vec`.**

- **Why people would like to write functions recursively?**

→ [Fact] All recursive algorithms can be rewritten iteratively.

→ Not all iterative algorithms can be rewritten recursively.

→ Advantage of recursive functions

- Neat code
- Less code (some loops become implicit)
- More understandable code

- **Fibonacci Numbers**

→ A Fibonacci sequence looks like the following:

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...]

- The zeroth number is 0, and the first number is 1.
- Starting from the second number, every number is the sum of the two previous numbers.

→ Suppose the `n`-th (zero-based) number in the Fibonacci sequence is `Fib(n)`, how to implement `Fib(n)`?

The general term of the Fibonacci sequence is $\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$.

We cannot use this to implement `Fib(n)`.

- **Iterative Algorithm**

→ Algorithm

1) Create an array of length `n + 1`.
2) From index `0` to `n`, fill cells with the `0`-th to `n`-th Fibonacci number.
3) Return the last number in the array.

→ [Example] Calculate `Fib(6)`.

Index	0	1	2	3	4	5	6
Value	0	1	1	2	3	5	8

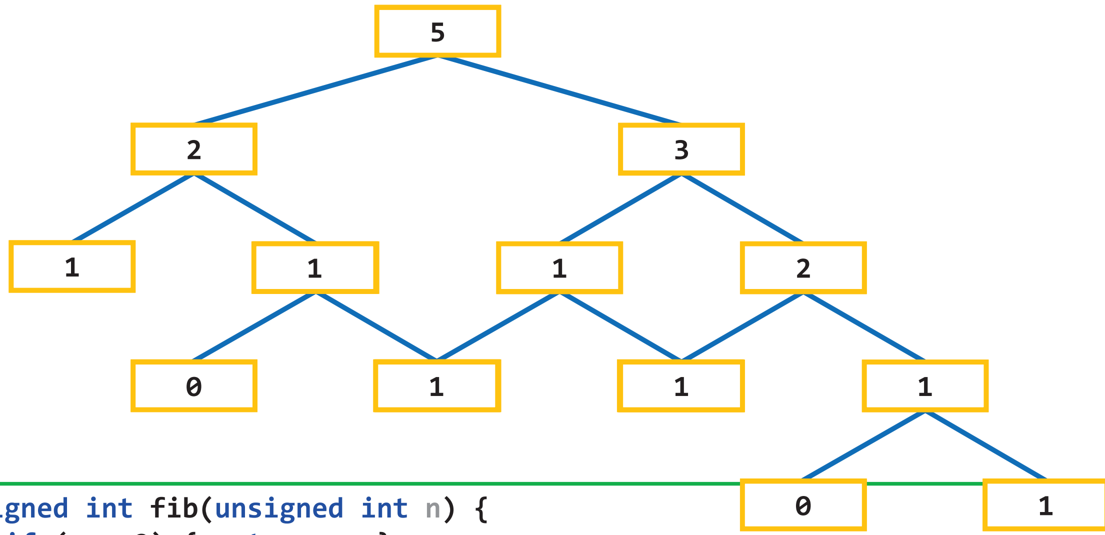
```
unsigned int fib(unsigned int n) {
    vector<unsigned int> fib_seq(n + 1);
    for (size_t i = 0; i < fib_seq.size(); i++) {
        if (i < 2) { fib_seq.at(i) = i; }
        else { fib_seq.at(i) = fib_seq.at(i - 2) + fib_seq.at(i - 1); }
    }
    return fib_seq.back();
}
```

- **Recursive Algorithm**

→ **Algorithm**

1) **If** `n == 0` **or** `n == 1`, **return** `n`.
2) **Otherwise, return** `Fib(n - 2) + Fib(n - 1)`.

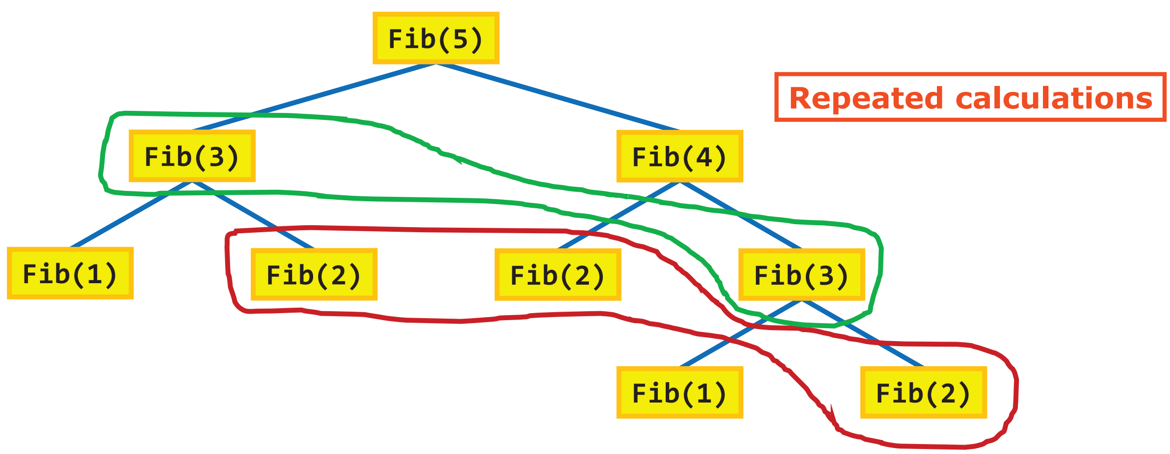
→ **[Example] Calculate** `Fib(5)`.




```
unsigned int fib(unsigned int n) {
    if (n < 2) { return n; }
    else { return fib(n - 2) + fib(n - 1); }
}
```


- **Disadvantage of Recursion**

→ **Recursive functions are often slower than iterative functions.**



- **Max (Min) Rewards**

Start

5	3	20	4
9	1	2	2
0	5	11	6
4	2	8	3

Stop

→ In an `n`-by-`n` grid, each cell contains a reward (non-negative integer).

→ You start from `grid[0][0]` (top-left corner).

→ In each move, you can either move to the right or move downward.

You cannot ~~move to the left~~ or ~~move upward~~.

→ When you reach a cell, you earn the reward stored in the cell.

→ You stop at `grid[n - 1][n - 1]` (bottom-right).

→ What is the max (min) reward you can get?

→ If each time you select the cell containing larger value…

5	3	20	4
9	1	2	2
0	5	11	6
4	2	8	3

Reward = 42

5	3	20	4
9	1	2	2
0	5	11	6
4	2	8	3

→ However,

Reward = 52

is the actual max.

Column (j)

Row (i)

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

- **Recursive Algorithm to Solve the "Max (Min) Reward Problem"**

→ Redefine the problem.

Let `MaxReward(i, j)` be the max reward from `grid[0][0]` to `grid[i][j]`.

→ Base case

If `i == 0` and `j == 0`, return `grid[i][j]`.

→ Recurrence relations

- If `i == 0` and `j > 0`, return `MaxReward[i][j - 1] + grid[i][j]`.
- If `i > 0` and `j == 0`, return `MaxReward[i - 1][j] + grid[i][j]`.
- If `i > 0` and `j > 0`,

  return `Max{MaxReward[i][j - 1], MaxReward[i - 1][j]} + grid[i][j]`.

```
unsigned int max_reward(const vector<vector<unsigned int>>& grid,
                        size_t i, size_t j) {
    if (!i && !j) { return grid.at(i).at(j); }  // Base case
    if (!i) { return max_reward(grid, i, j - 1) + grid.at(i).at(j); }
    if (!j) { return max_reward(grid, i - 1, j) + grid.at(i).at(j); }
    return max(max_reward(grid, i - 1, j), max_reward(grid, i, j - 1))
           + grid.at(i).at(j);
}
```



```
// Wrapper function
unsigned int max_reward(const vector<vector<unsigned int>>& grid) {
    return max_reward(grid, grid.size() - 1, grid.size() - 1);
}
```

```
int main() {
    vector<vector<unsigned int>> grid = {
        { 5, 3, 20, 4 },
        { 9, 1,  2, 2 },
        { 0, 5, 11, 6 },
        { 4, 2,  8, 3 }
    };
    cout << "Max Reward: " << max_reward(grid) << endl;
    system("pause");
    return 0;
}
```

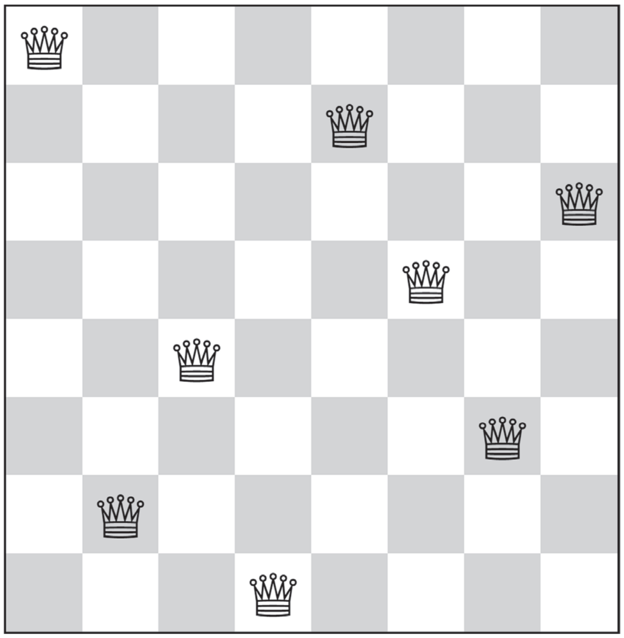
Console	52

- **The 8-Queen Puzzle**

→ **The 8-Queen Puzzle is the problem of placing 8 queens on an 8-by-8 chessboard and no two queens threaten each other.**

Thus, a solution requires that no two queens share the same row, column, or diagonal.

There are totally 92 different patterns (solutions) exist.



- **Data Fields in Class `Eight_Queen_Puzzle`**

  - **A 2-dimensional vector that represents the 8-by-8 chessboard**
  - **A character that represents a queen on the chessboard**
  - **A character that represents a blank on the chessboard**
  - **An integer that counts the number of solutions found**

```
typedef vector<vector<char>> Chessboard;

// Data fields

// An 8-by-8 chessboard
Chessboard board;
// Character to represent a queen on the chessboard
static const char QUEEN;
// Character to represent a blank on the chessboard
static const char BLANK;
// Stores the number of solutions found.
unsigned int num_of_solutions;
```

- **Easiest Idea to Solve an 8-Queen Puzzle**

→ **Since we can only place one queen in each row, our steps are:**

- **Place a queen in row 0.**
- **Place a queen in row 1.**

...

→ **There are 8 cells in a row, so which cell to place the queen?**

We need to use a `for` loop to try all the 8 cells in the row.

For each cell, how to tell whether it is a good position to place a queen?

1) There is no queen in the same column.

2) There is no queen in diagonal directions.

4 diagonal directions: topleft, topright, bottomright, bottomleft

If these two conditions are satisfied, then we can place a queen in the cell.

→ **Therefore, we need to define two functions:**

```cpp
// Tests whether there is already a queen in a given column.
bool queen_in_column(int) const;

// Tests whether there is already a queen in diagonal directions.
bool queen_in_diagonal(int, int) const;
```



```cpp
/** Tests whether there is already a queen in a given column.
    @param col: index of the column to test
    @return: {true} if there is already a queen in the column;
             {false} otherwise
*/
bool Eight_Queen_Puzzle::queen_in_column(int col) const {
    for (size_t row = 0; row < 8; row++) {
        if (board.at(row).at(col) == QUEEN) { return true; }
    }
    return false;
}
```

```
/** Tests whether there is already a queen in diagonal directions.
    @param row: row index of the cell to test
    @param col: column index of the cell to test
    @return: {true} if there is already a queen in diagonal
             directions; {false} otherwise
*/
bool Eight_Queen_Puzzle::queen_in_diagonal(int row, int col) const {
    // Test the topleft direction.
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board.at(i).at(j) == QUEEN) { return true; }
    }
    // Test the topright direction.
    for (int i = row, j = col; i >= 0 && j < 8; i--, j++) {
        if (board.at(i).at(j) == QUEEN) { return true; }
    }
    // Test the bottomright direction.
    for (int i = row, j = col; i < 8 && j < 8; i++, j++) {
        if (board.at(i).at(j) == QUEEN) { return true; }
    }
    // Test the bottomleft direction.
    for (int i = row, j = col; i < 8 && j >= 0; i++, j--) {
        if (board.at(i).at(j) == QUEEN) { return true; }
    }
    return false;  // No queen in either diagonal direction
}
```


- **When we can tell that a solution is found?**

→ **We start placing queens in row 0.**

After we successfully placed a queen in row 7 (last row), a solution is found.

→ **After a solution is found, we need to output that solution.**

```
// Writes a solution to an output stream.
void print_solution(ostream&);
```

```
/** Writes a solution to an output stream.
    @param out: output stream to write the found solution
*/
void Eight_Queen_Puzzle::print_solution(ostream& out) {
    // Write the solution number.
    out << "Solution: " << ++num_of_solutions << endl << endl;
    // Write the chessboard.
    for (size_t row = 0; row < 8; row++) {
        for (size_t col = 0; col < 8; col++) {
            out << board.at(row).at(col);
        }
        out << endl;
    }
    out << endl;
}
```

- **How to recursively find all the solutions?**

→ **Redefined problem:**

Let `print_solutions(row, out)` writes all the solutions to the output stream `out`, when we start placing queens from row index `row`.

→ **Base case:**

If `row == 8`, then a solution is found. Call `print_solution(out)`.

→ **Recurrence relation:**

After successfully placed a queen in this row, go to the next row (`row + 1`) and call `print_solutions(row + 1, out)` to generate all the solutions.

→ **How to find all the solutions?**

After placing a queen and all the solutions are generated, remove the queen from current position and try the next position (and so on), in order to find more solutions.

→ **Wrapper function:**

`print_solutions(0, out)` is the original problem.


```
// Writes all the solutions from a given row to an output stream.
void print_solutions(int, ostream&);
```

```
/** Writes all the solutions from a given row to an output stream.
    @param row: index of the row to start placing queens
    @param out: output stream to write the solutions
*/
void Eight_Queen_Puzzle::print_solutions(int row, ostream& out) {
    // Base case
    if (row == 8) { print_solution(out); }  // A solution is found.
    else {   // Place a queen in this row.
        for (int col = 0; col < 8; col++) {
            if (queen_in_column(col)) { continue; }
            if (queen_in_diagonal(row, col)) { continue; }
            // Place a queen.
            board.at(row).at(col) = QUEEN;
            // [Recursion] Go to the next row to place queens.
            print_solutions(row + 1, out);
            // Remove the queen from this cell and place it in other
            // cells to find more solutions.
            board.at(row).at(col) = BLANK;
        }
    }
}
```

```
void print_solutions(ostream&);  // Wrapper function
```

```
// Wrapper function
void Eight_Queen_Puzzle::print_solutions(ostream& out) {
    print_solutions(0, out);
}
```

```
int main() {
    // Open the output file.
    ofstream fout("solutions.txt");

    // Create a puzzle.
    Eight_Queen_Puzzle puzzle;

    // Solve the puzzle.
    puzzle.print_solutions(fout);

    // Close the output file.
    fout.close();

    return 0;
}
```