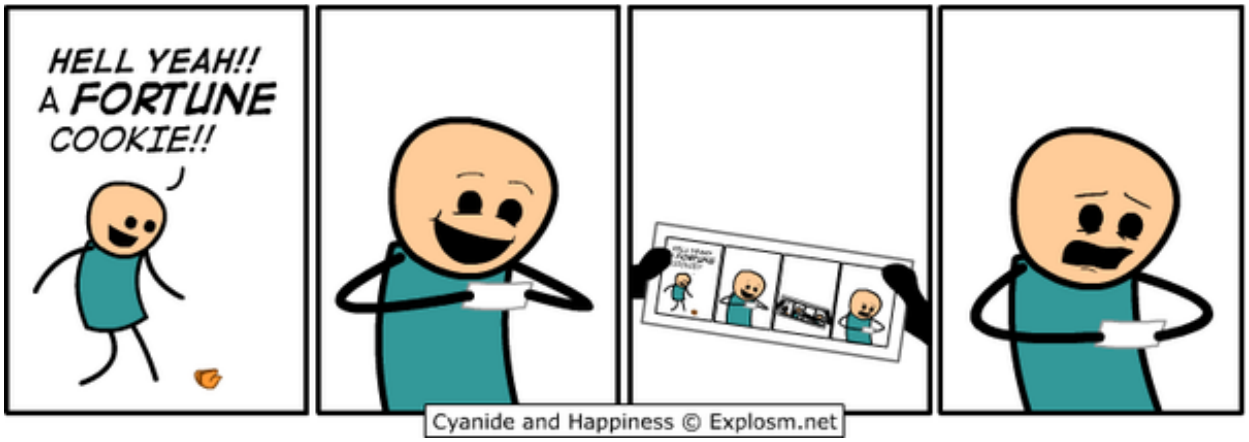
CS 106X, Summary
Introduction to Recursion

reading:

Programming Abstractions in C++, Chapter 7



Recursive Thinking

- **In code**, recursion is when a function in your program calls itself as part of its execution.
- **Conceptually**, a recursive problem is one that is *self-similar*; it can be solved via smaller occurrences of the same problem.

Note: Recursion: Describing a process/problem in terms of itself.
Code: write a function that calls itself.

Recursion works well with problems that are self-simlar
Example: look up a word in dictionary: look for wordA's definition -> look for wordB's definition -> .....

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **base case**: A simple occurrence that can be answered directly.
  - **recursive case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem.
  - *Key idea:* In a recursive piece of code, you handle a small part of the overall task yourself, then make a recursive call to handle the rest.
  - Ask yourself, "How is this task **self-similar**?"
    - "How can I describe this algorithm in terms of a smaller or simpler version of itself?"

"Recursion Zen"

- The real, even simpler, base case is an exp of 0, not 1:

```
int power(int base, int exp) {
    if (exp == 0) {
        // base case; base^0 = 1
        return 1;
    } else {
        // recursive case:  x^y = x * x^(y-1)
        return base * power(base, exp - 1);
    }
}
```

  - **Recursion Zen**: The art of properly identifying the best set of cases for a recursive algorithm and expressing them elegantly.

    (our informal term)

The Recursion Checklist

- ❑ **Find what information we need to keep track of.** What inputs/outputs are needed to solve the problem at each step? Do we need a wrapper function?
- ❑ **Find our base case(s).** What are the simplest (non-recursive) instance(s) of this problem?

building recursive step: find a way to divide the problem

- ❑ **Find our recursive step**. How can this problem be solved in terms of one or more simpler instances of the same problem that lead to a base case?
- ❑ **Ensure every input is handled.** Do we cover all possible cases? Do we need to handle errors?

Note: **check1**: understand what the function actually does. **check3**:How to devide the problem simpler in terms of problem itself; start from recursive case that is only a little harder than base case; recursive leap of faith. **check4**: precondition,throw.

Recursion in a nutshell

1. 函数的返回值和parameter是可以用来记录信息的（设计参数的时候考虑要记录哪些信息）搞清楚每一步(state)要记录的信息特征
2. base case：最简单的
3. recursive step： 思考如何以一种方式divide and conquer问题，把大问题细化；leap of faith
4. error handling：recursion的设计本身是一种test-case-driven approach：考虑各种case：base，recursive，error...

How Many Behind Me?

1. If there is no one behind me, I will answer **0**.
2. If there is someone behind me:
   - Ask them how many people are behind *them*
   - My answer is their answer plus 1

1. Base case: the simplest possible instance of this question. One that requires no additional recursion.

2. Recursive case: describe the problem using smaller occurrences of the same problem.

The "Recursive Leap of Faith"

- When writing a recursive function, pretend that someone has already written a function that can solve that problem for smaller inputs. How could you use it in your implementation?
- E.g. "what if we had a function that could print out a smaller file in reverse?"

Key idea: You handle only small part of problem yourself, then make a recursive call-believe someone helps you-to handle the rest.

```
void reverseLines(ifstream& input) {
    string line;
    if (getline(input, line)) {
        reverseLines(input);      // "leap of faith"
        cout << line << endl;
    }
}
```

Wrapper Functions

```
Map<int, int> cache;
int sixthFibonacci = fibonacci(5, cache);  // 5
```

- The above function signature isn't ideal; it requires the client to know to pass in an (empty) map.
- In general, the parameters we need for our recursion will not always match those the client will want to pass.
- Is there a way we can remove that requirement, while still memoizing?
- **YES!** A "wrapper" function is a function that "wraps" around the first call to a recursive function to abstract away any additional parameters needed to perform the recursion.

Note: 包装函数，将recursive function包在内部，提供一个函数内的全局环境；recursive function的额外参数被包在/abstract away by wrapper function，一般可作为内部全局变量，用pass by reference.

That's a Wrap(per)!

```
// "Wrapper" function that returns the nth Fibonacci number.
// This version calls the recursive version with an empty cache.
int fibonacci(int i) {
    HashMap<int, int> cache;
    return fibonacci(i, cache);
}

// Recursive function that returns the nth Fibonacci number.
// This version uses memoization.
int fibonacci(int i, HashMap<int, int>& cache) {
    if (i < 0) {
        throw "Illegal negative index";
    } else if (i < 2) {
        return i;
    } else if (cache.containsKey(i)) {
        return cache[i];
    } else {
        int result = fibonacci(i-1, cache) + fibonacci(i-2, cache);
        cache[i] = result;
        return result;
    }
}
```

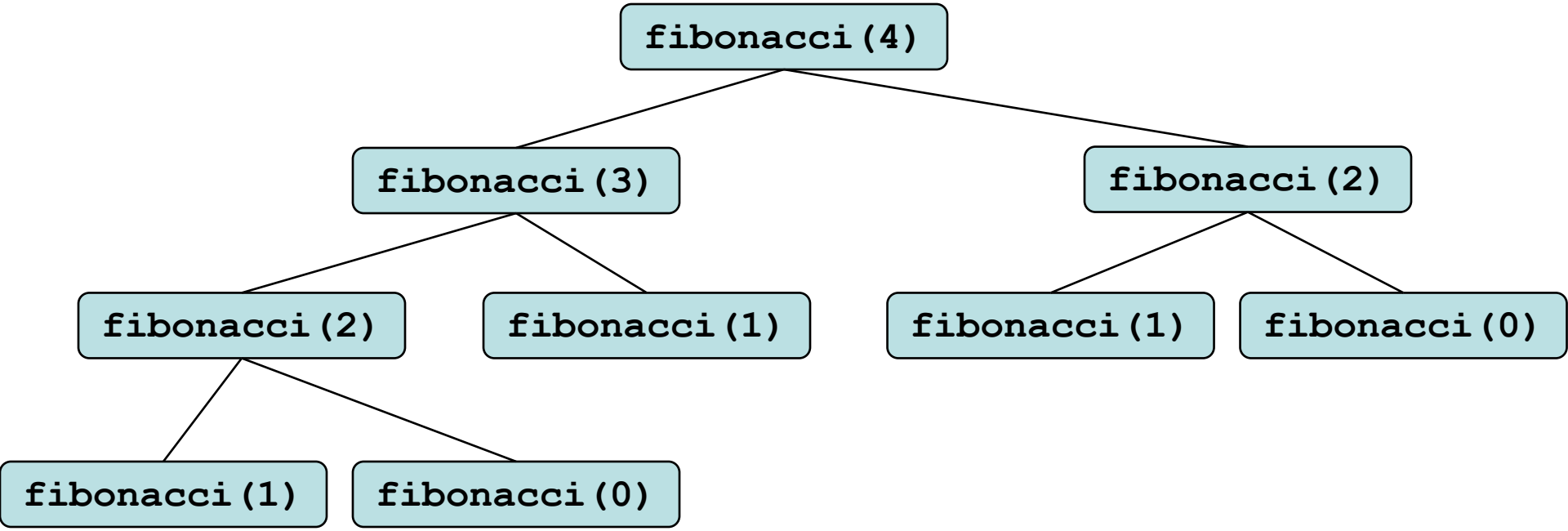
Tricky exercise!

evaluate exercise

- Write a recursive function **`evaluate`** that accepts a string representing a math expression and computes its value.
  - The expression will be "fully parenthesized" and will consist of + and * on single-digit integers only.

```
evaluate( "7" )                    => 7
evaluate( "(2+2)" )                => 4
evaluate( "(1+(2*4))" )            => 9
evaluate( "((1+3)+((1+2)*5))" )    => 19
```

Fibonacci: Big-O



- Each recursive call makes 2 *additional* recursive calls.
- The worst-case depth of the recursion is the index of the Fibonacci number we are trying to calculate (N).
- Therefore, the number of total calls is $O(2^N)$.
- Each individual function call does O(1) work. Therefore, the total runtime is $O(2^N) * O(1) = O(2^N)$.

Recursion & Big-O

- The runtime of a recursive function is the number of function calls times the work done in each function call.
- The number of calls for a branching recursive function is usually

$$O(b^d)$$

where

  - **b** is the worst-case branching factor (# recursive calls per function execution)
  - **d** is the worst-case depth of the recursion (the longest path from the top of the recursive call tree to a base case).

Memoization

- **memoization**: Caching results of previous expensive function calls for speed so that they do not need to be re-computed.
  - Often implemented by storing call results in a collection.

- Pseudocode template:

```
cache = {}.          // initially empty

function f(args):
    if I have computed f(args) before:
        Look up f(args) result in cache.
    else:
        Actually compute f(args) result.
        Store result in cache.
    Return result.
```

Note about Memoizaiton:

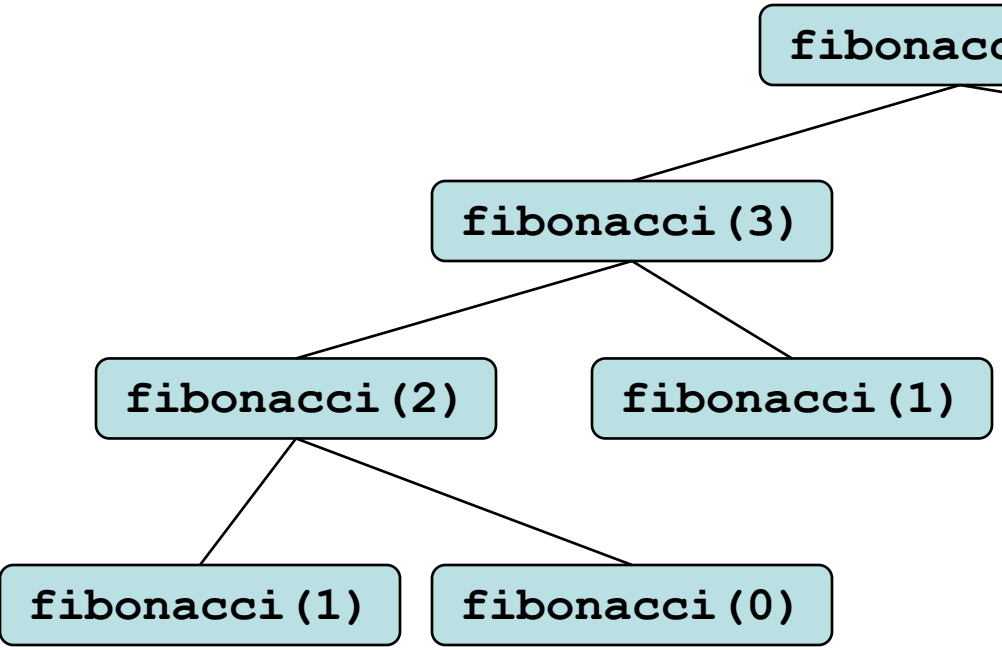
1. Clearly understand the meaning of return model

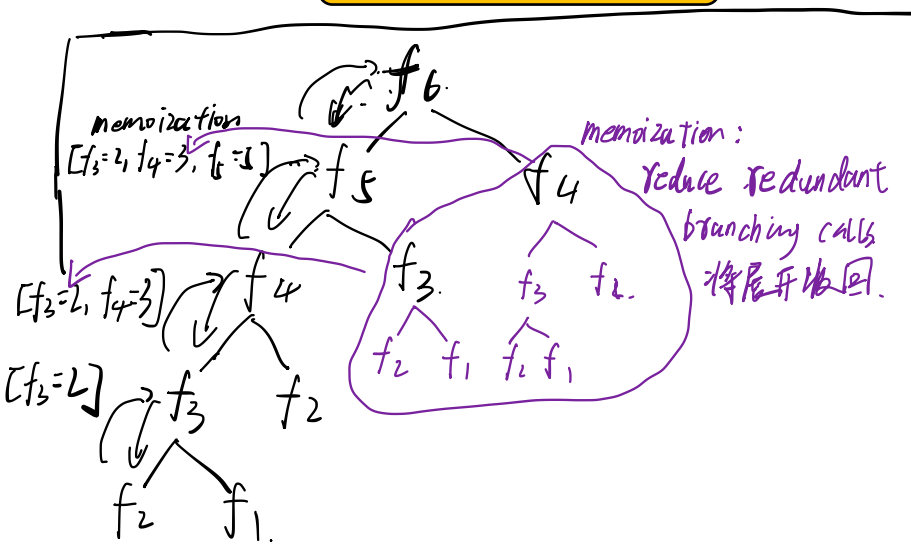
2. Classify input args: some args are key args while others are helper args, e.g. storage variable. Only key args should be key of catche.

3. We don't have to store result in base case.

4. Checked wheather f(key args) was computed or not first, then base case, finally recursive case.

Recursive Tree





memoization optimization

Is there a way to remember what we already computed?