CS106X Midterm Review

Jennie Yang and Jared Bitz

Based on slides from Nick Troccoli, Nolan Handali, Anton Apostolatos and Ashley Taylor


DON'T
PANIC

Topic List

- **C++ Fundamentals:** Console and file I/O, strings, pass by reference vs. pass by value
- **Using ADTS:** Vector, Grid, Stack, Queue, (Hash)Set, (Hash)Map, Lexicon and their tradeoffs/advantages
- **Algorithms Analysis and Big-O**
- **Recursion and Backtracking:** Tracing and writing recursive code
- **Unit Testing:** Writing unit tests for C++ functions

What's NOT on the Midterm

- Pointers
- Creating your own classes
- Operator overloading
- Arrays
- Anything not explicitly covered in lecture
- Anything only mentioned in the "overflow" part of Nick's slide decks
- Anything covered in class after unit testing in week five

Functions and Pass by Reference

pass by reference

cup = 

fillCup( )

pass by value

cup = 

fillCup( )

www.mathwarehouse.com

Functions and Pass by Reference

- In a function definition, by default all parameters are copies of their original.
- When you include an ‘&’ in the type of a parameter, this means that you are passing the original version of the parameter. This means any changes to it change the original!
- This is useful if you want to update multiple pieces of information, or save memory.

Functions and Pass by Reference

- Tips
  - Mark or otherwise remember when reading a program when something is pass by reference
  - Make sample mystery code and test yourself and your friends
  - Draw a box containing all the variables and parameters for each function you encounter. Cross out the box (tip: don't erase!) when the function is finished executing.

Practice: Trace

```
int main() {
    int frodo = 7;
    int sam = 5;
    int merry = 4;
    int sancho = dumas(frodo, sam, merry);
    int pippin = cervantes(sam, sancho);
    cout << sam << endl;
    cout << pippin << endl;
    cout << merry << endl;
    return 0;
}
```

Challenge: solve this before proceeding to solution!

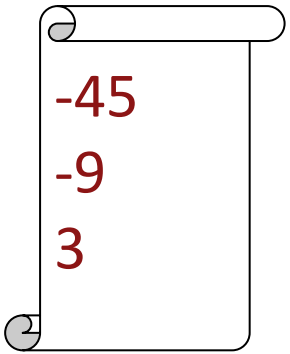
```
int cervantes(int &sancho, int quixote) {
    sancho *= quixote;
    return quixote--;
}


int dumas(int athos, int &aramis,
          int &porthos) {
    if (athos + aramis < porthos) {
        athos = aramis - porthos;
    } else {
        porthos--;
    }
    return aramis - (athos + 7);
}
```

Practice: Trace

```
int main() {
    int frodo = 7;
    int sam = 5;
    int merry = 4;
    int sancho = dumas(frodo, sam, merry);
    int pippin = cervantes(sam, sancho);
    cout << sam << endl;
    cout << pippin << endl;
    cout << merry << endl;
    return 0;
}
```

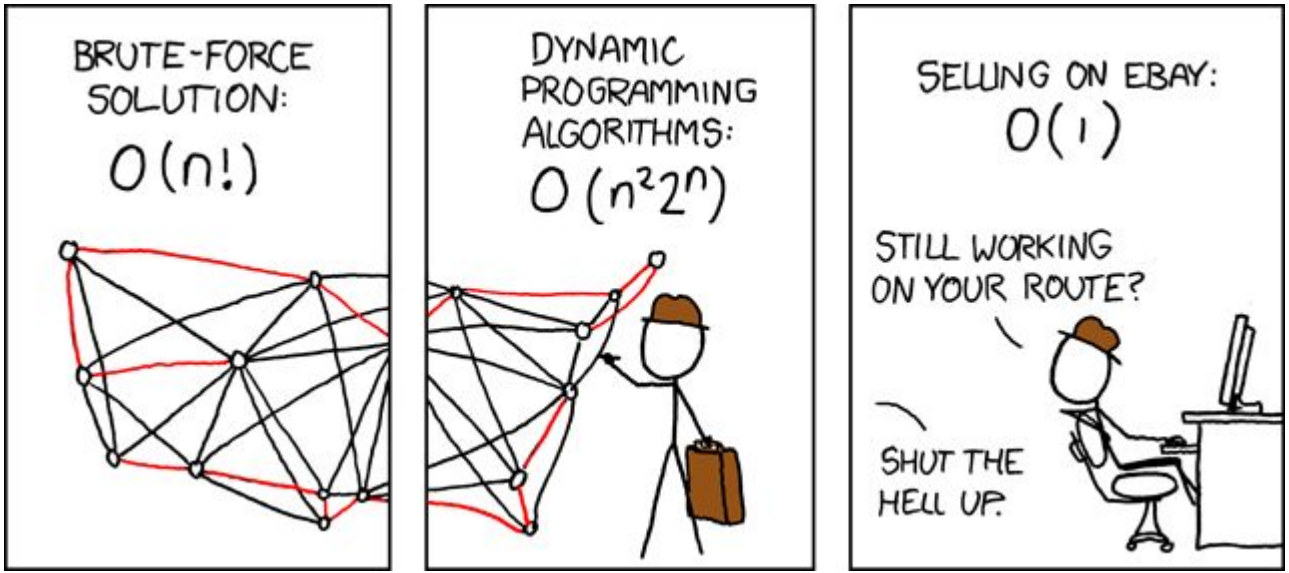
Console




```
int cervantes(int &sancho, int quixote) {
    sancho *= quixote;
    return quixote--;
}


int dumas(int athos, int &aramis,
          int &porthos) {
    if (athos + aramis < porthos) {
        athos = aramis - porthos;
    } else {
        porthos--;
    }
    return aramis - (athos + 7);
}
```

Big-O


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.

Big O Notation

- Big O Notation represents how the runtime of code changes as the amount of data it process increases, in the *worst case*.
- Big O is approximate – we are only interested in the largest-order terms that dominate the runtime.
- We provide the runtimes of collections functions on the reference sheet.

O(N)

```
for (int i = 0; i < N; i++) {
    cout << “Hello, world!” << endl;
}
```

O(1)

```
for (int i = 0; i < 1000000; i++) {
   cout << “Hello, world! “ << N << endl;
}
```

$O(N^2)$

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      cout << “Hello, world!” << endl;
   }
}
```

$O(N^2)$

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < i; j++) {
      cout << "Hello, world!" << endl;
   }
}
```

$O(N^3)$

```
Vector<int> v;
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      v.insert(0, i);
      v.insert(0, j);
   }
}
```

O(logN)

- The **logN** runtime appears when you continually divide the size of the data by some constant.
  - e.g. in binary search, we divide the data in half each time when searching for an element.
  - Some ADTs, like **Set** and **Map**, have common operations in O(logN) time because they are implemented in a way similar to binary search.

O(NlogN)

```
Set<int> v;
for (int i = 0; i < N; i++) {
   v.add(i);
}
```

Runtimes

$$O(1) < O(\log N) < O(N) < O(N^2) < O(N^3) < O(x^N)$$

Practice: Big O

```
Vector<int> v;
for (int i = 0; i < 3*N+1; i++) {
    v.add(i);
}

Set<int> s;
for (int i = v.size() - 1; i >= 0; i--) {
    s.add(v[i]);
}
```

Challenge: solve this before proceeding to solution!

Practice: Big O

```
Vector<int> v;
for (int i = 0; i < 3*N+1; i++) {
    v.add(i);
}

Set<int> s;
for (int i = v.size() - 1; i >= 0; i--) {
    s.add(v[i]);
}
```

O(NlogN)

Big O and Recursion

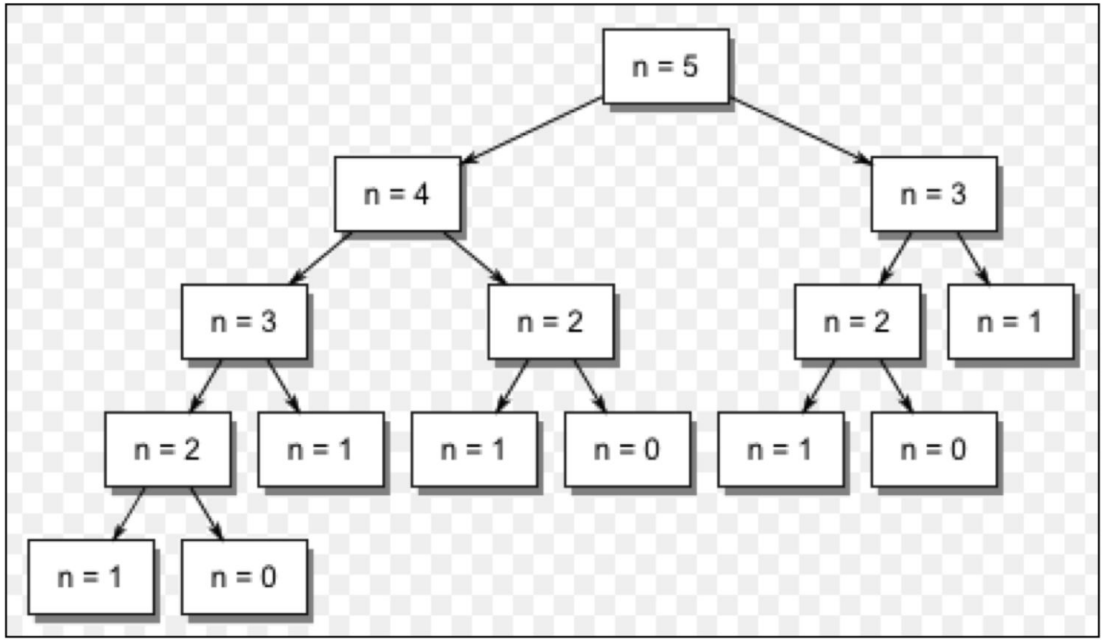
- To analyze the runtime of a recursive function, you need:
  - The number of recursive calls made
  - The runtime of a single recursive call

```
int fib(int n) {
  if(n <= 1) {
    // base case
    return 1;
  } else {
    // recursive case
    return fib(n – 1) + fib(n – 2);
  }
}
```

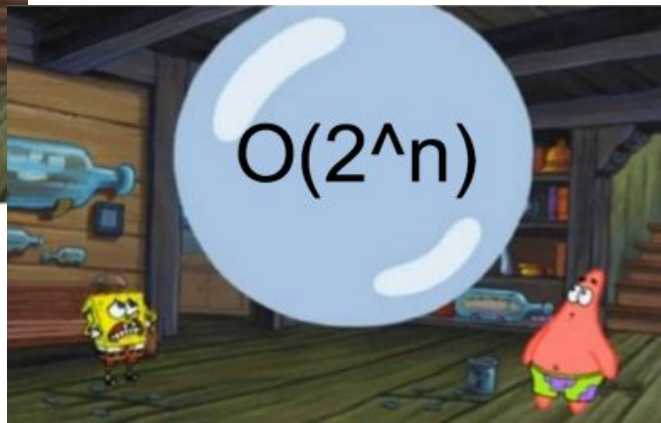
Big O and Recursion

- Example: Fibonacci
  - O(1) per recursive call
  - 2^N recursive calls

Here is the call tree of Fib(5);

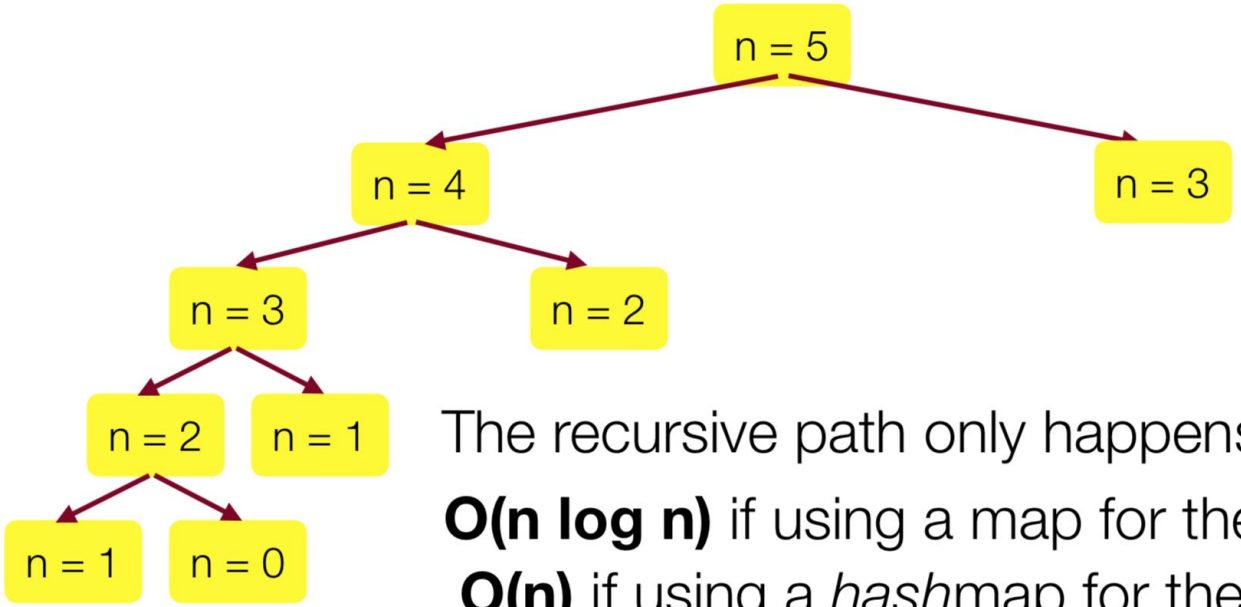


Big O and Recursion



Big O and Memoization

Complexity?



The recursive path only happens on the left...

O(n log n) if using a map for the cache

O(n) if using a *hash*map for the cache

Practice: Big O and Recursion

```
int recurse(Stack<int> &s) {
    if (!s.isEmpty()) {
        int x = s.pop();
        return x + recurse(s);
    }

    return 0;
}
```

Challenge: solve this before proceeding to solution!

Practice: Big O and Recursion

```
int recurse(Stack<int> &s) {
    if (!s.isEmpty()) {
        int x = s.pop();
        return x + recurse(s);
    }

    return 0;
}
```

O(N)

Collections

Collections

- Vector
- Grid
- Stack
- Queue
- Map / HashMap
- Set / HashSet

Vector

- A Vector is a 1D list of elements.
- You can access elements at any index, and add/remove elements at any index.

`vec.size()`	Returns the number of elements in the vector.
`vec.isEmpty()`	Returns `true` if the vector is empty.
`vec[i]`	Selects the i^{th} element of the vector.
`vec.add(value)`	Adds a new element to the end of the vector.
`vec.insert(index, value)`	Inserts the value before the specified index position.
`vec.remove(index)`	Removes the element at the specified index.
`vec.clear()`	Removes all elements from the vector.

```
43   12   0   -20   32
```

Grid

- A Grid is a 2D list of elements.
- You can access elements at any index, resize, and check if in bounds.

`grid.numRows()`
Returns the number of rows in the grid.

`grid.numCols()`
Returns the number of columns in the grid.

`grid[i][j]`
Selects the element in the i^{th} row and j^{th} column.

`grid.resize(rows, cols)`
Changes the dimensions of the grid and clears any previous contents.

`grid.inBounds(row, col)`
Returns `true` if the specified row , column position is within the grid.

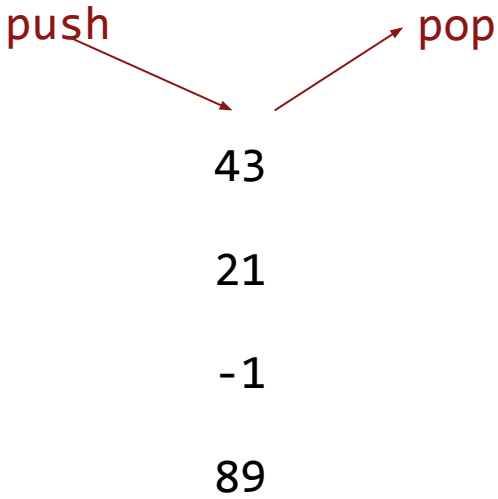
```
4   90   20

12   0   33
```

Stack

- A Stack is a 1D "LIFO" (last-in-first-out) list.
  - Like a stack of plates. The last one put on is the first off
- You can push an element on top or pop an element off the top of the stack.

`stack.size()` Returns the number of values pushed onto the stack.
`stack.isEmpty()` Returns `true` if the stack is empty.
`stack.push(value)` Pushes a new value onto the stack.
`stack.pop()` Removes and returns the top value from the stack.
`stack.peek()` Returns the top value from the stack without removing it.
`stack.clear()` Removes all values from the stack.



Queue

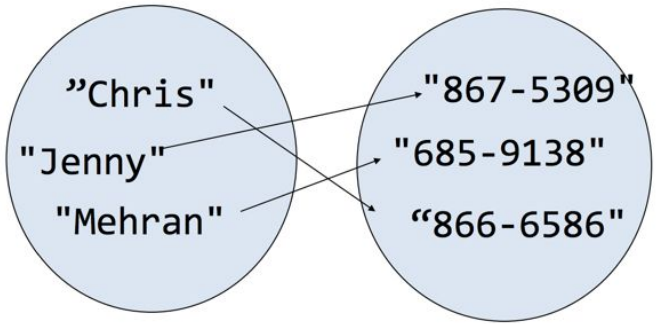
- A Queue is a 1D "FIFO" (first-in-first-out) list.
- You can enqueue onto the back or dequeue from the front.

`queue.size()` Returns the number of values in the queue.
`queue.isEmpty()` Returns `true` if the queue is empty.
`queue.enqueue(value)` Adds a new value to the end of the queue (which is often called its *tail*).
`queue.dequeue()` Removes and returns the value at the front of the queue (its *head*).
`queue.peek()` Returns the value at the head of the queue without removing it.
`queue.clear()` Removes all values from the queue.

Map

- A Map is a collection of key/value pairs.
- You can add/remove pairs, or check if something is in the map

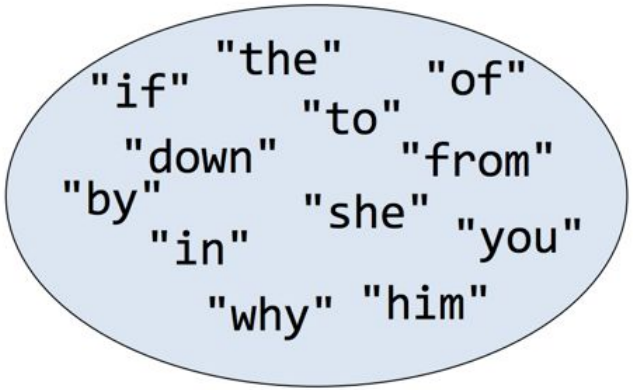
`m.clear();`	removes all key/value pairs from the map
`m.containsKey(key)`	returns `true` if the map contains a mapping for the given key
`m[key]` or `m.get(key)`	returns the value mapped to the given key; if key not found, **adds** it with a default value (e.g. 0, "")
`m.isEmpty()`	returns `true` if the map contains no k/v pairs (size 0)
`m.keys()`	returns a `Vector` copy of all keys in the map
`m[key] = value;` or `m.put(key, value);`	adds a mapping from the given key to the given value; if the key already exists, **replaces** its value with the given one
`m.remove(key);`	removes any existing mapping for the given key
`m.size()`	returns the number of key/value pairs in the map
`m.toString()`	returns a string such as `"{a:90, d:60, c:70}"`
`m.values()`	returns a `Vector` copy of all values in the map



Set

- A Set is a collection of unique elements.
- You can add/remove elements, or check if something is in the set

`set.size()` Returns the number of elements in the set.
`set.add(value)` Adds a new value to the set (ignores it if the value already was in the set).
`Set.contains(value)` Return true if the value is in the set.



Set/HashSet and Map/HashMap

- Remember that HashSet and HashMap are unordered, while Set and Map are ordered.
- Also remember that HashSet and HashMap can do common operations in O(1) time, while Set and Map can do common operations in O(logN) time.

Practice: Collections

At airports, there are usually different boarding lines for different passenger priorities. However, this time, somehow the boarding lines got mixed together!

Write a function **boardingLine** that takes in a list of passenger names, and a map from a passenger name to their priority, and returns a single Queue of passenger names in the order they should board.

Higher priorities should board first, and between two passengers with the same priority, the passenger who is first in the mixed-up line should board first. Assume a constant **NUM_LINES** number of lines.

Practice: Collections

Write a function **boardingLine** that takes in a list of passenger names, and a map from a passenger name to their priority, and returns a single Queue of passenger names in the order they should board.

Higher priorities should board first, and between two passengers with the same priority, the passenger who is first in the mixed-up line should board first. Assume a constant **NUM_LINES** number of lines.

```
Queue<string> boardingLine(const Vector<string>& passengers,
    const Map<string, int>& priorities);
```

Challenge: solve this before proceeding to solution!

Practice: Collections

1. Initialize a list of queues, 1 for each priority
2. Sort the passengers into the right queue
3. Merge the queues back together into a single queue

Practice: Collections

1. Initialize a list of queues, 1 for each priority
2. Sort the passengers into the right queue
3. Merge the queues back together into a single queue

Step 1: Initialize Queues

```
Vector<Queue<string>> queues;
for (int i = 0; i < NUM_QUEUES; i++) {
   Queue<string> q;
   queues.add(q);
}
...
```

Practice: Collections

1. Initialize a list of queues, 1 for each priority
2. Sort the passengers into the right queue
3. Merge the queues back together into a single queue

Step 2: Sort Passengers by Queue

```
...
for (string passenger : passengers) {
    int priority = priorities[passenger];
  queues[priority].enqueue(passenger);
}
...
```

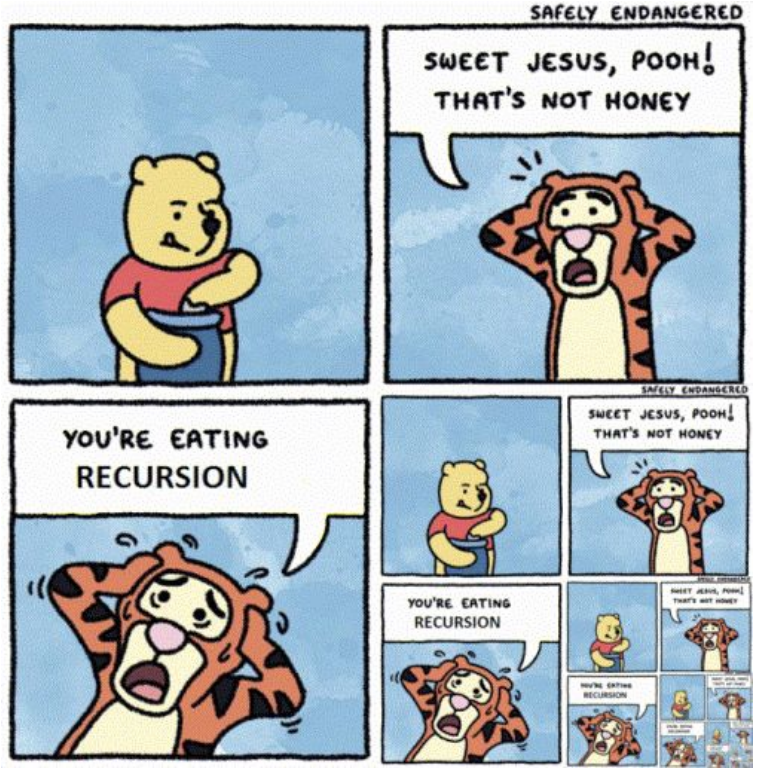
Practice: Collections

1. Initialize a list of queues, 1 for each priority
2. Sort the passengers into the right queue
3. Merge the queues back together into a single queue

Step 2: Sort Passengers by Queue

```
...
Queue<string> finalQueue;
for (int i = queues.size() - 1; i >= 0; i--) {
   while (!queues[i].isEmpty()) {
      finalQueue.enqueue(queues[i].dequeue());
   }
}
return finalQueue;
```

Recursion


SAFELY ENDANGERED
SWEET JESUS, POOH! THAT'S NOT HONEY
YOU'RE EATING RECURSION

Key Ideas

Your code must have a case for all valid inputs.

You must have a base case that makes no recursive calls.

When you make a recursive call, it should be to a simpler instance and make progress towards the base case.

Key Ideas

The base case represents the simplest possible instance of the problem you are solving.

How many people are behind you?	None!
What is the nth Fibonacci number?	F(0) = 1
Hailstone problem starting at N?	N = 1

Key Ideas

The recursive case represents how you can break down the problem into smaller instances of the same problem.

How many people are behind you? 1 + # behind-behind me

What is the nth Fibonacci number? F(N) = F(N-1) + F(N-2)

Hailstone problem starting at N? Hail(N/2) or Hail(3N+1)

Key Ideas

The recursive leap of faith is the idea that, after implementing the base case, assume that you already have a working version of the function you are implementing.

Eg. Fibonacci – we implement the base case, F(0) = 1. Now, when implementing how to calculate F(N), assume we have a working version of F. How could we use that?

Recursive 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 your base case(s).**

  What are the simplest (non-recursive) instance(s) of this problem?

- **Find your 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?

Practice: Recursion

Write a recursive function named **matchCount** that accepts two references to vectors of integers as parameters and that returns the number of elements that match between them. Two elements match if they occur at the same index in both vectors and have equal values. For example, given the two vectors shown below, the call of matchCount(v1, v2) would compare as follows:

v1: {2, 5, 0, 3, 8, 9, 1, 1, 0, 7}
| | | | | | | |
v2: {2, 5, 3, 0, 8, 4, 1}

Challenge: solve this before proceeding to solution!

The function should return 4 in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). If either vector is empty, by definition it has no matches with the other vector, so your function should return 0.

Practice: Recursion

- Base case: when either vector is empty, return 0
- Recursive step – let's look at one index at a time. We can compare elements at that index, and then recurse to get the rest of the matches.
- Should we have a helper function? What should it look like?

Practice: Recursion

- Helper function - what do we need to know when recursing?
  - Both of the vectors
  - Where in those vectors we are currently looking
    - Pretend that making additional vectors or copies is not allowed

```
int matchCountHelper(Vector<int>& v1,
                     Vector<int>& v2, int index);
```

Recursion: Practice

```
int matchCount(Vector<int>& v1, Vector<int>& v2) {
    return matchCountHelper(v1, v2, 0);
}

int matchCountHelper(Vector<int>& v1, Vector<int>& v2,
                     int index) {
    if (index == v1.size() || index == v2.size()) {
        return 0;
    }
    int count = matchCountHelper(v1, v2, index+1);
    if (v1[index] == v2[index]) {
        count++;
    }
    return count;
}
```

Recursion: Practice

```
int matchCount(Vector<int>& v1, Vector<int>& v2) {
    return matchCountHelper(v1, v2, 0);
}

int matchCountHelper(Vector<int>& v1, Vector<int>& v2,
                     int index) {
    if (index == v1.size() || index == v2.size()) {
        return 0;
    }
    int count = matchCountHelper(v1, v2, index+1);
    if (v1[index] == v2[index]) {
        count++;
    }
    return count;
}
```

Recursion: Practice

```
int matchCount(Vector<int>& v1, Vector<int>& v2) {
    return matchCountHelper(v1, v2, 0);
}

int matchCountHelper(Vector<int>& v1, Vector<int>& v2,
                     int index) {
    if (index == v1.size() || index == v2.size()) {
        return 0;
    }
    int count = matchCountHelper(v1, v2, index+1);
    if (v1[index] == v2[index]) {
        count++;
    }
    return count;
}
```

Recursion: Practice

```
int matchCount(Vector<int>& v1, Vector<int>& v2) {
    return matchCountHelper(v1, v2, 0);
}

int matchCountHelper(Vector<int>& v1, Vector<int>& v2,
                     int index) {
    if (index == v1.size() || index == v2.size()) {
        return 0;
    }
    int count = matchCountHelper(v1, v2, index+1);
    if (v1[index] == v2[index]) {
        count++;
    }
    return count;
}
```

Recursion: Practice

```
int matchCount(Vector<int>& v1, Vector<int>& v2) {
    return matchCountHelper(v1, v2, 0);
}

int matchCountHelper(Vector<int>& v1, Vector<int>& v2,
                     int index) {
    if (index == v1.size() || index == v2.size()) {
        return 0;
    }
    int count = matchCountHelper(v1, v2, index+1);
    if (v1[index] == v2[index]) {
        count++;
    }
    return count;
}
```

Recursion: Practice

```
int matchCount(Vector<int>& v1, Vector<int>& v2) {
    return matchCountHelper(v1, v2, 0);
}

int matchCountHelper(Vector<int>& v1, Vector<int>& v2,
                     int index) {
    if (index == v1.size() || index == v2.size()) {
        return 0;
    }
    int count = matchCountHelper(v1, v2, index+1);
    if (v1[index] == v2[index]) {
        count++;
    }
    return count;
}
```

Alternative solutions: make sub-vectors and chop off 1 element each time we recurse.

Recursive Exploration and Backtracking

Backtracking Checklist

- Find the choices you have at each step
- For each valid choice
  - Make it and explore recursively
  - Undo it after exploring
- Identify a base case

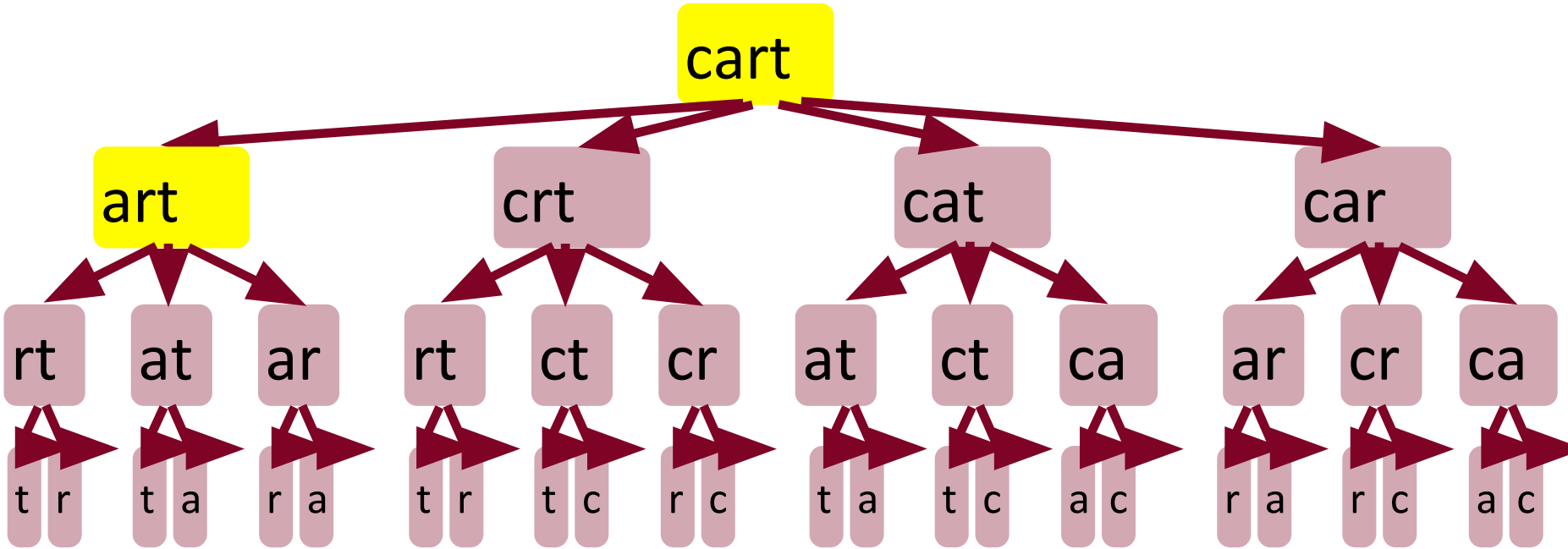
Exploration/Recursive Backtracking

- Exploration/Recursive Backtracking is a technique to explore all solutions to a problem in an effort to find some subset of those solutions.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Find/print all the solutions

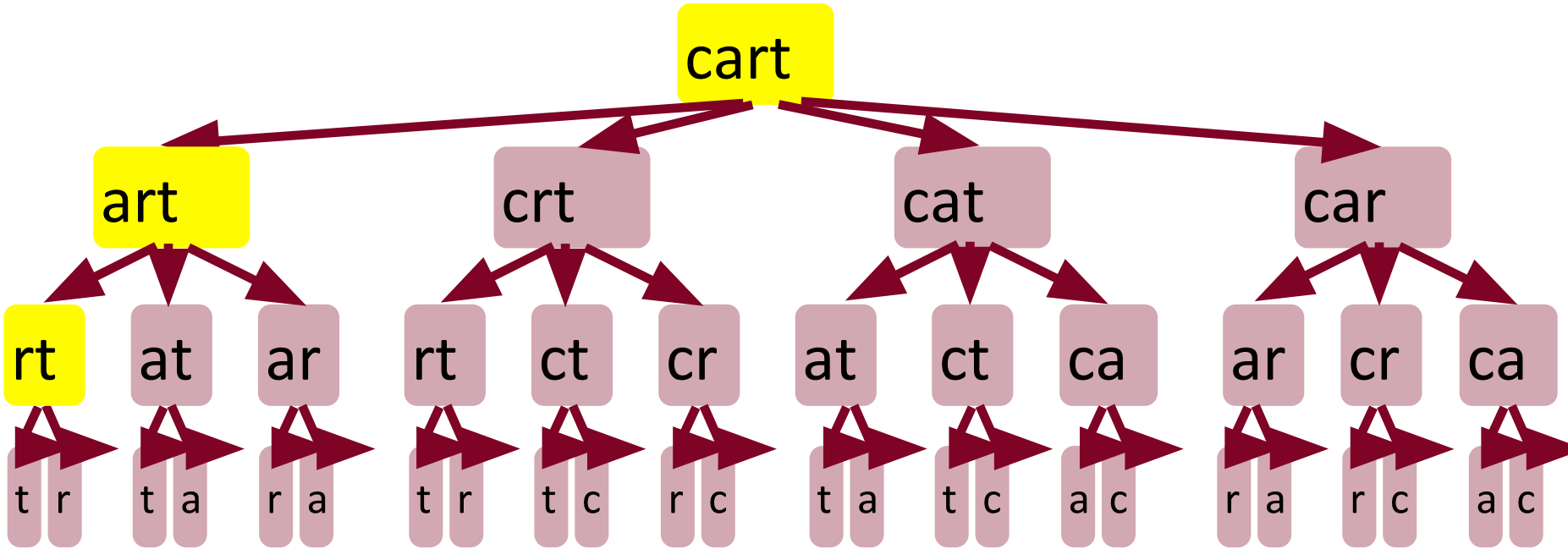
Let's define a reducible word as a word that can be reduced down to one letter by removing one character at a time, leaving a word at each step.

- Base case:
  - A one letter word in the dictionary.

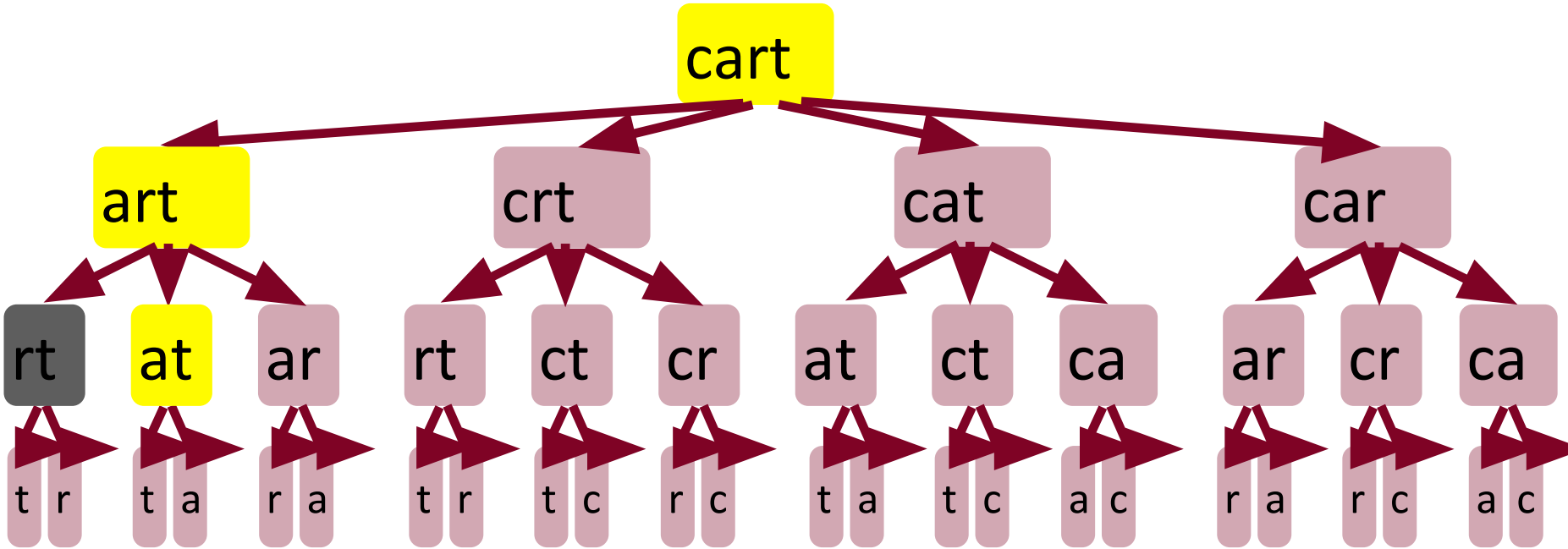
- Recursive Step:
  - Any multi-letter word is reducible if you can remove a letter (legal move) to form a shrinkable word.


cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

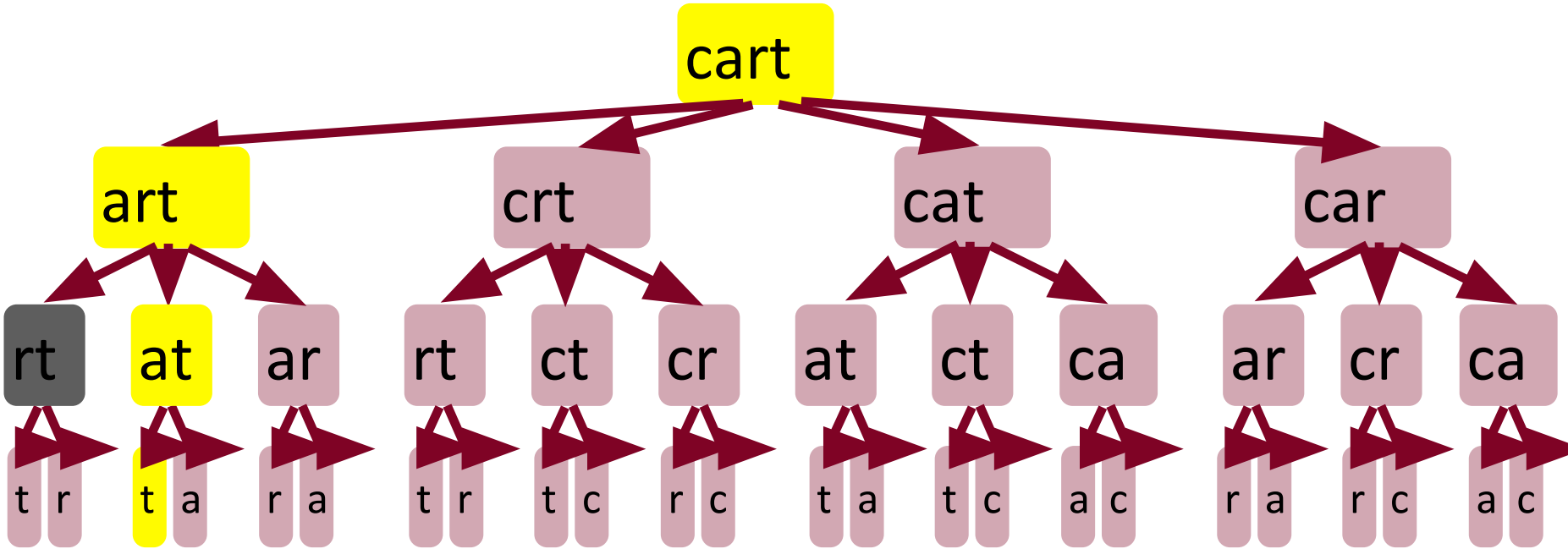
art: is a word


cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

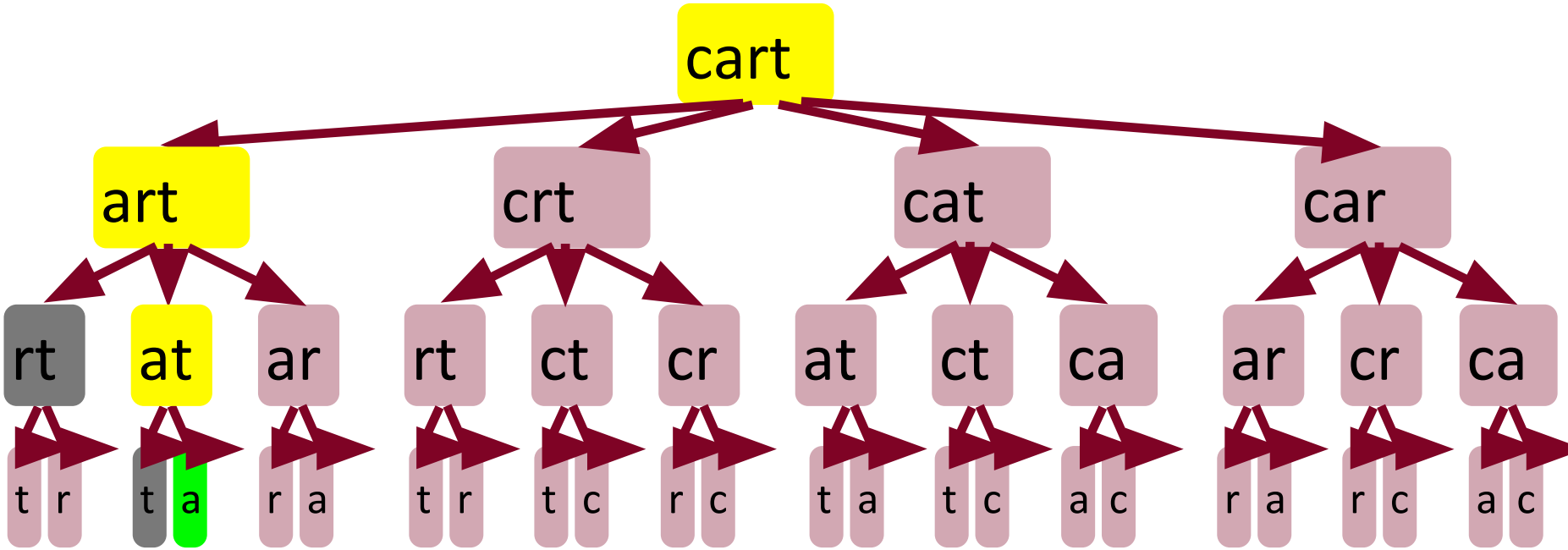
rt: not a word


cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

at: is a word

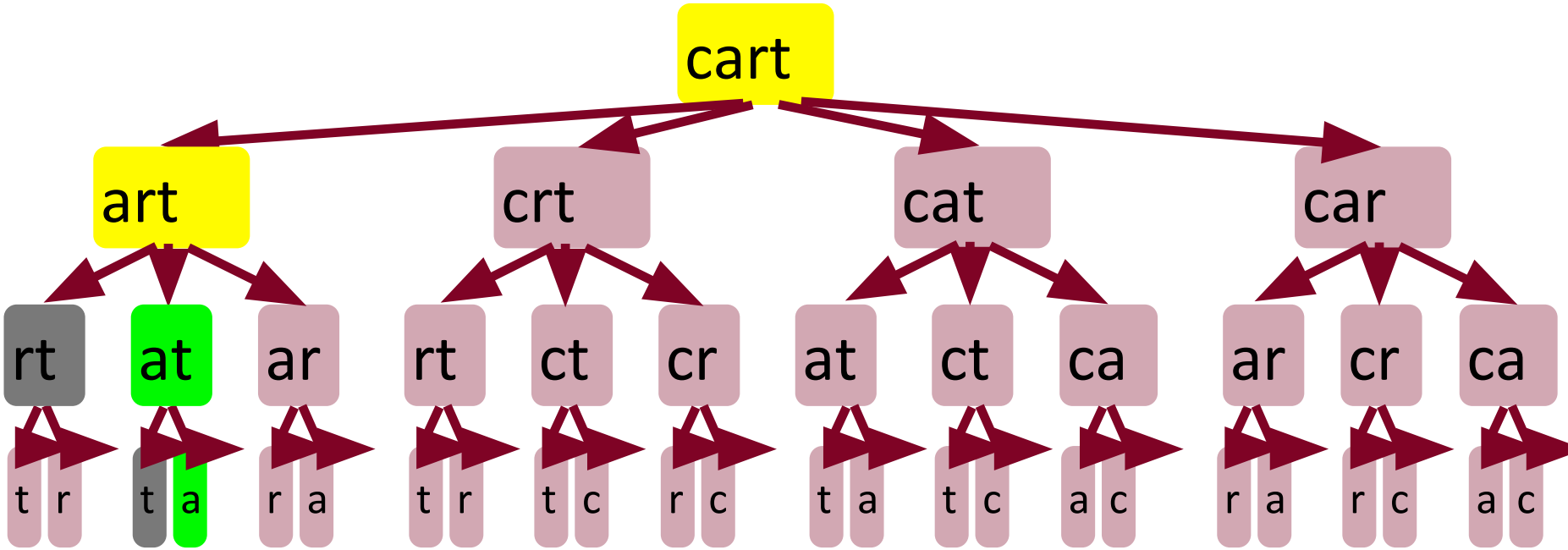

cart
art
crt
cat
car
rt
at
ar
rt
ct
cr
at
ct
ca
ar
cr
ca
t r
t a
r a
t r
t c
r c
t a
t c
a c
r a
r c
a c

t: not a word



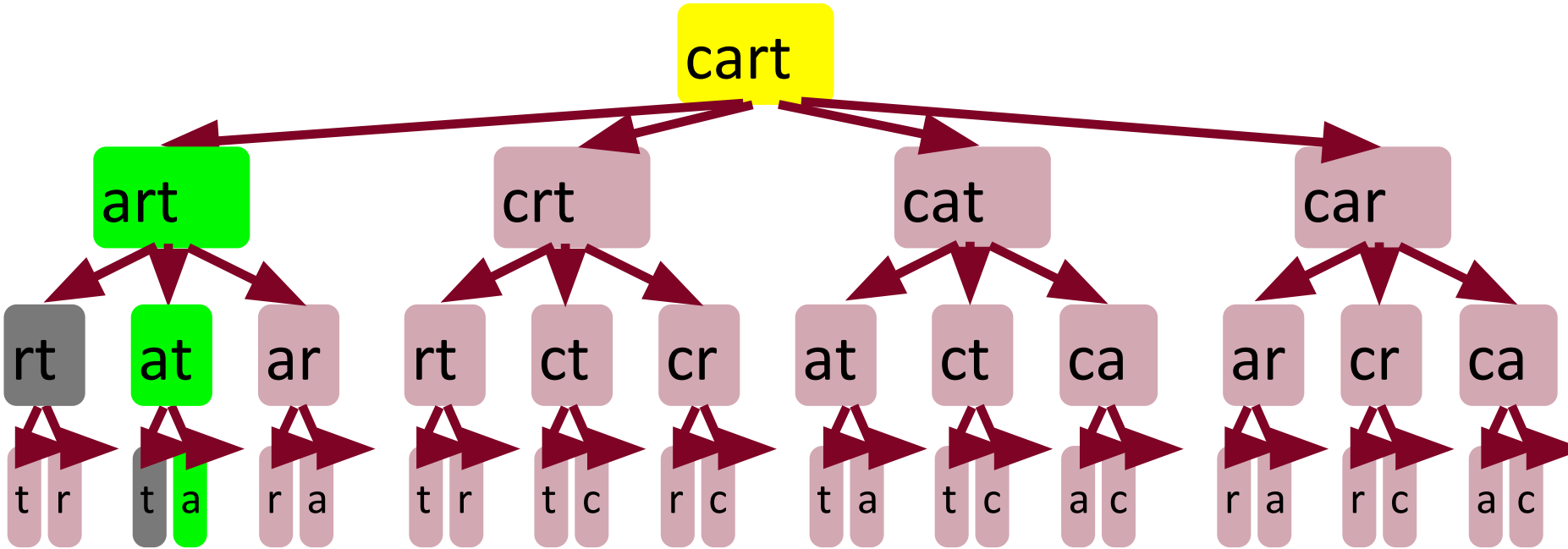
a: is a word

there is a solution!



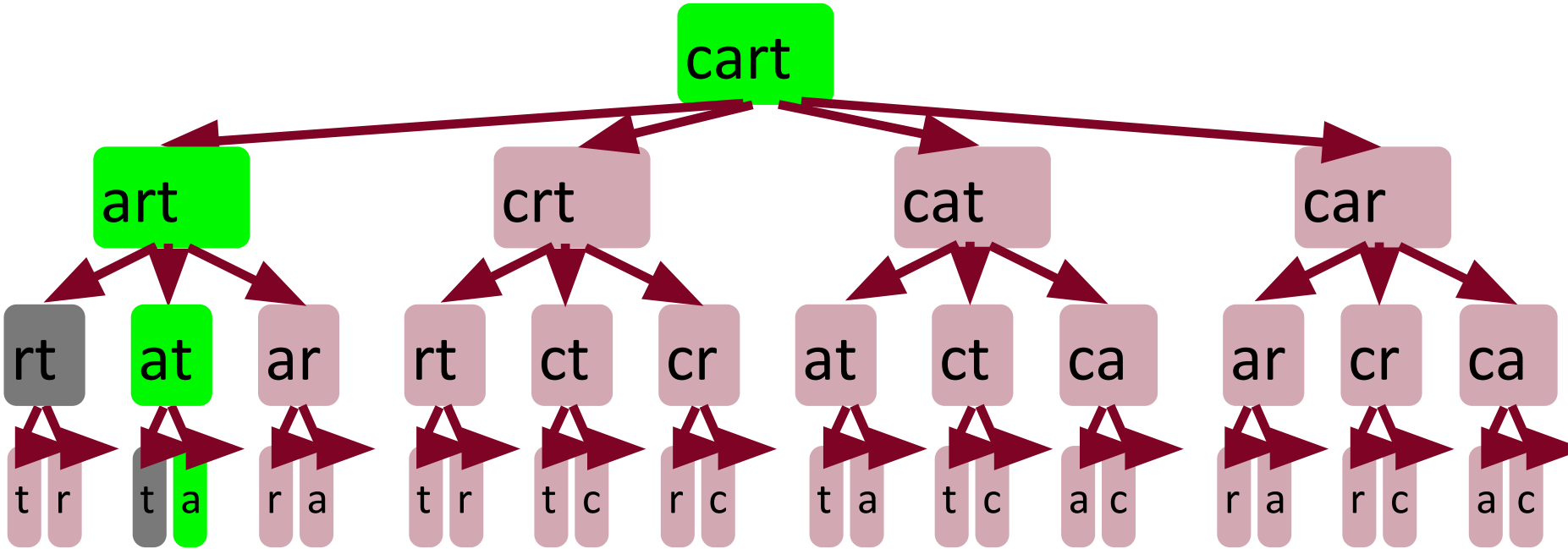
a: is a word

there is a solution!



a: is a word

there is a solution!



a: is a word

there is a solution!

General Exploration Structure

1. Identify the type of problem (all solutions? Best? Etc.)
2. Determine base case(s)
3. Identify what “choice” should be made at each step (e.g. on next character in a string, next index in a vector, etc.)
4. Identify how to undo the “choice” (e.g. return string to original form, add element back to the vector, etc.)

Exploration/Recursive Backtracking

- Exploration/Recursive Backtracking is a technique to explore all solutions to a problem in an effort to find some subset of those solutions.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Find/print all the solutions

Does a Solution Exist?

- Should return a boolean
- Base case: validate the solution so far; return **true** if valid
- Recursive step: check all potential choices. If one returns **true**, you return **true**. Otherwise, return **false**.

Example 1: Determine Existence

- You're playing a board game with your friends, and each one is very picky and will play only with certain pieces.
- Given a set of game pieces (strings), a list of friend names, and a map from friend names to acceptable pieces for that friend, can you assign each of your friends a unique piece?

```
bool gamePieces(Set<string>& unusedPieces,
                Vector<string>& friends,
                Map<string, Set<string>>& constraints);
```

Example 1: Determine Existence

```
bool gamePieces(Set<string>& unusedPieces,
                Vector<string>& friends,
                Map<string, Set<string>>& constraints) {
    if (friends.isEmpty()) {
        return true;
    }

    // let's just examine 1 of the players
    string currFriend = friends[0];
    friends.remove(0);

    ...
```

Example 1: Determine Existence

```
    for (string piece : constraints[currFriend]) {
        if (unusedPieces.contains(piece)) {
            unusedPieces.remove(piece);      // choose
            // explore
            if (gamePieces(unusedPieces, friends, constraints)) {
                // unchoose
                unusedPieces.add(piece);
                friends.insert(0, currFriend);
                return true;
            }
            unusedPieces.add(piece);         // unchoose
        }
    }
    friends.insert(0, currFriend);
    return false;
}
```

Exploration/Recursive Backtracking

- Exploration/Recursive Backtracking is a technique to explore all solutions to a problem in an effort to find some subset of those solutions.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Find/print all the solutions

Does a Solution Exist?

- Base case: validate the solution so far; return the solution if it's valid, or an empty solution otherwise.
- Recursive step: check all potential choices. If one returns a valid solution, return that. Otherwise, return an empty solution.

Example 2: Find a Solution

A Queen in chess can move diagonally, horizontally, and vertically (any straight line). Place N queens in an N*N grid such that none can attack another, or return an empty grid if we can't. A spot in the grid is true if a queen is placed there.

```
// Implement this
Grid<bool> placeQueens(int n);

// Provided with this
bool isSpotSafe(const Grid<bool>& board, int row, int col);
```

Example 2: Find a Solution

HINT: sometimes, it's easier to implement a recursive helper function that takes the solution by reference and populates it, and have the return type be a boolean indicating whether a solution was actually found. The boolean may not matter to the outer function, but it can make the recursion cleaner.

```
// Implement this
Grid<bool> placeQueens(int n);

// Provided with this
bool isSpotSafe(const Grid<bool>& board, int row, int col);
```

Example 2: Find a Solution

```
Grid<bool> placeQueens(int n) {
    Grid<bool> board(n, n);
    placeQueens(board, 0);
    return board;
}
```

Example 2: Find a Solution

```
bool placeQueens(Grid<bool> & board, int col) {
    if (col == board.numCols()) { //every col has a queen
        return true;
    }
    for (int row = 0; row < board.numRows(); row++) {
        if (isSpotSafe(board, row, col)) {//choose a safe row
            board[row][col] = true;
            if (placeQueens(board, col + 1)) { //explore
                return true; //solution found
            }
            board[row][col] = false; //remove the queen
        }
    }
    return false; //none of our choices worked, return false
}
```

Exploration/Recursive Backtracking

- Exploration/Recursive Backtracking is a technique to explore all solutions to a problem in an effort to find some subset of those solutions.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Find/print all the solutions

Find the Best Solution

- Base case: is it a valid solution? If so, return it. Otherwise, return a default/empty solution.
- Recursive step: check all potential choices, then output the "best" of all of them.

Example 3: Find Best Solution

Given a string, a **subsequence** is another string where all the characters of the subsequence appear in the string in the same relative order, but the not every character from the string needs to appear. E.g. “cef” is a subsequence of “abcdef”, but “db” is not.

Find the longest subsequence from a provided string such that all letters in the subsequence are strictly increasing. (e.g. A < B < C). Assume that the input string is in lowercase.

```
string longestIncreasingSubsequence(string input);
```

Example 3: Find Best Solution

```
string longestIncSubseq(string input, string subseq) {
    int length = subseq.size();
    if (length > 1 &&
        subseq[length - 1] <= subseq[length - 2]) {
        return “”; //not increasing subsequence
    }

    if (input == “”) {
        return subseq; //no more characters to process
    }

    ...
```

Example 3: Find Best Solution

```
    ...
    // leave to the next recursive call to check if this
    // new subsequence is actually valid.
    string withChar = longestIncSubseq(input.substr(1),
                                       subseq + input[0]);
    string withoutChar = longestIncSubseq(input.substr(1),
                                          subseq);
    //choose the "best" of the recursive calls
    if (withChar.size() > withoutChar.size()) {
        return withChar;
    }
    return withoutChar;
}
```

Exploration/Recursive Backtracking

- Exploration/Recursive Backtracking is a technique to explore all solutions to a problem in an effort to find some subset of those solutions.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Find/print all the solutions

Count the Number of Solutions

- Base case: is it a valid solution? If so, return 1. Otherwise, return 0.
- Recursive step: return the sum of all the recursive calls.

This approach is useful because sometimes we want to make sure that there is *exactly* one solution. For instance, a maze!

Example 4: Count Solutions

Given a maze represented as a Grid<bool> (true if you can go somewhere, false if it's a wall), and a start and end location, count the number of unique paths from the start to the end. We assume the maze is bordered by walls.

```
int mazeSolutions(Grid<bool>& maze, int startRow,
                  int startCol, int endRow,
                  int endCol);
```

Example 4: Count Solutions

```
int mazeSolutions(Grid<bool> & maze, int startRow,
                  int startCol, int endRow, int endCol) {
    if (!maze[startRow][startCol]) {
        // can't travel through walls
        return 0;
    }
    if (startRow == endRow && startCol == endCol) {
        return 1; //reached our goal
    }

    ...
```

Example 4: Count Solutions

```
    ...
    maze[startRow][startCol] = false; // mark our choice
    int numSolutions = mazeSolutions(maze, startRow + 1,
        startCol, endRow, endCol);
    numSolutions += mazeSolutions(maze, startRow - 1,
        startCol, endRow, endCol);
    numSolutions += mazeSolutions(maze, startRow,
        startCol + 1, endRow, endCol);
    numSolutions += mazeSolutions(maze, startRow,
        startCol - 1, endRow, endCol);
    maze[startRow][startCol] = true; // "undo" our choice
    return numSolutions;
}
```

Exploration/Recursive Backtracking

- Exploration/Recursive Backtracking is a technique to explore all solutions to a problem in an effort to find some subset of those solutions.
  - Determine whether a solution exists
  - Find a solution
  - Find the best solution
  - Count the number of solutions
  - Find/print all the solutions

Find/Print All Solutions

- Base case: is it a valid solution? If so, print it (or add to set of found solutions). If not valid, don't.
- Recursive step: Make all recursive calls. If you are returning a set, add each recursive result to the set.

Example 5: Find All Solutions

Write a recursive function named **listTwiddles** that accepts a string **str** and a reference to an English language Lexicon and uses exhaustive search and backtracking to print out all those English words that are **str's** twiddles. Two English words are considered twiddles if the letters at each position are either the same, neighboring letters, or next-to-neighboring letters. For instance, "sparks" and "snarls" are twiddles. Their second and second-to-last characters are different, but 'p' is two past 'n' in the alphabet, and 'k' comes just before 'l'.

Example 5: Find All Solutions

```
void listTwiddles(string str, const Lexicon &lex) {
    string prefix = "";
    listTwiddlesHelper(prefix, str, /* index */ 0, lex);
}
```

Example 5: Find All Solutions

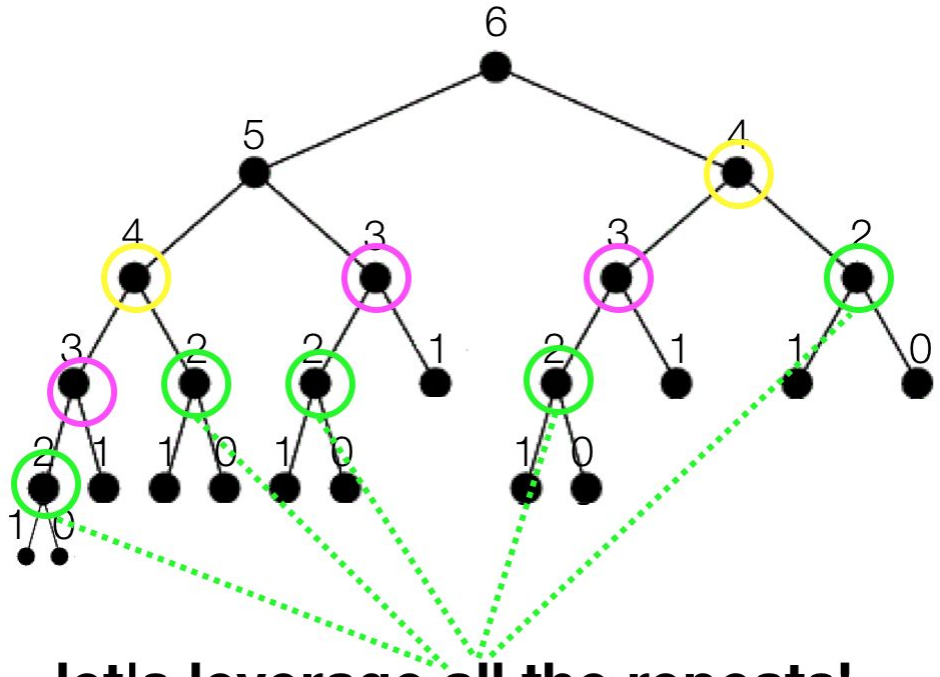
```
void helper(string prefix, string s, int i, const Lexicon &lex) {
    if (!lex.containsPrefix(prefix)) {
        return; // optimization; not necessary but good to do
    }
    if (index >= (int)str.size()) {
        if (lex.contains(prefix)) {
            cout << prefix << endl;
        }
    } else {
        for (char ch=str[index]-2; ch <= str[index]+2; ch++) {
            if (isalpha(ch)) {
                helper(prefix+ch, str, index+1, lex);
            }
        }
    }
}
```

Memoization

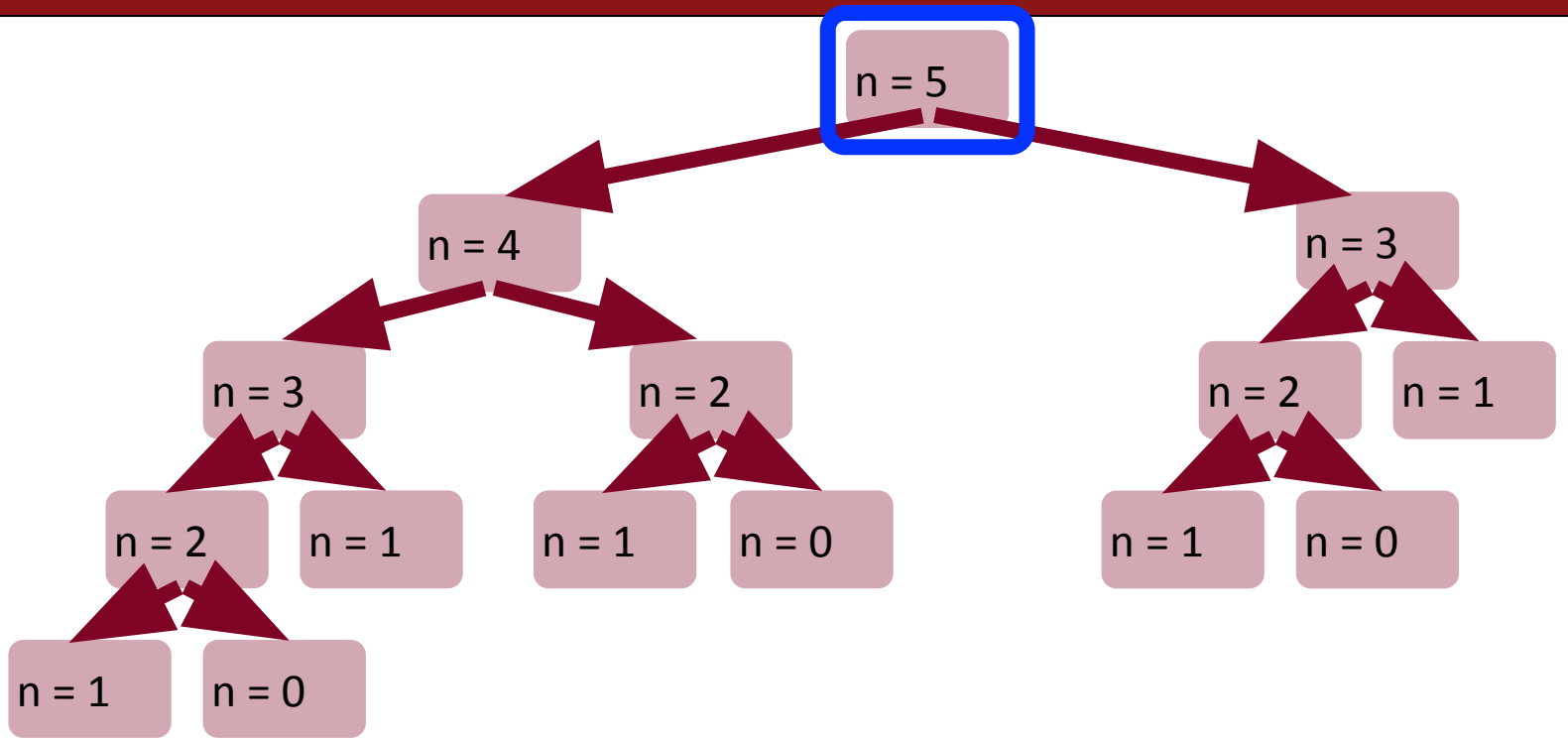
Memoization

- Recursion breaks down the problem into smaller identical sub-problems to solve. This means that sometimes we solve the same sub-problem multiple times!
- Instead, let's *remember* every subproblem we solve, and reuse past calculations.

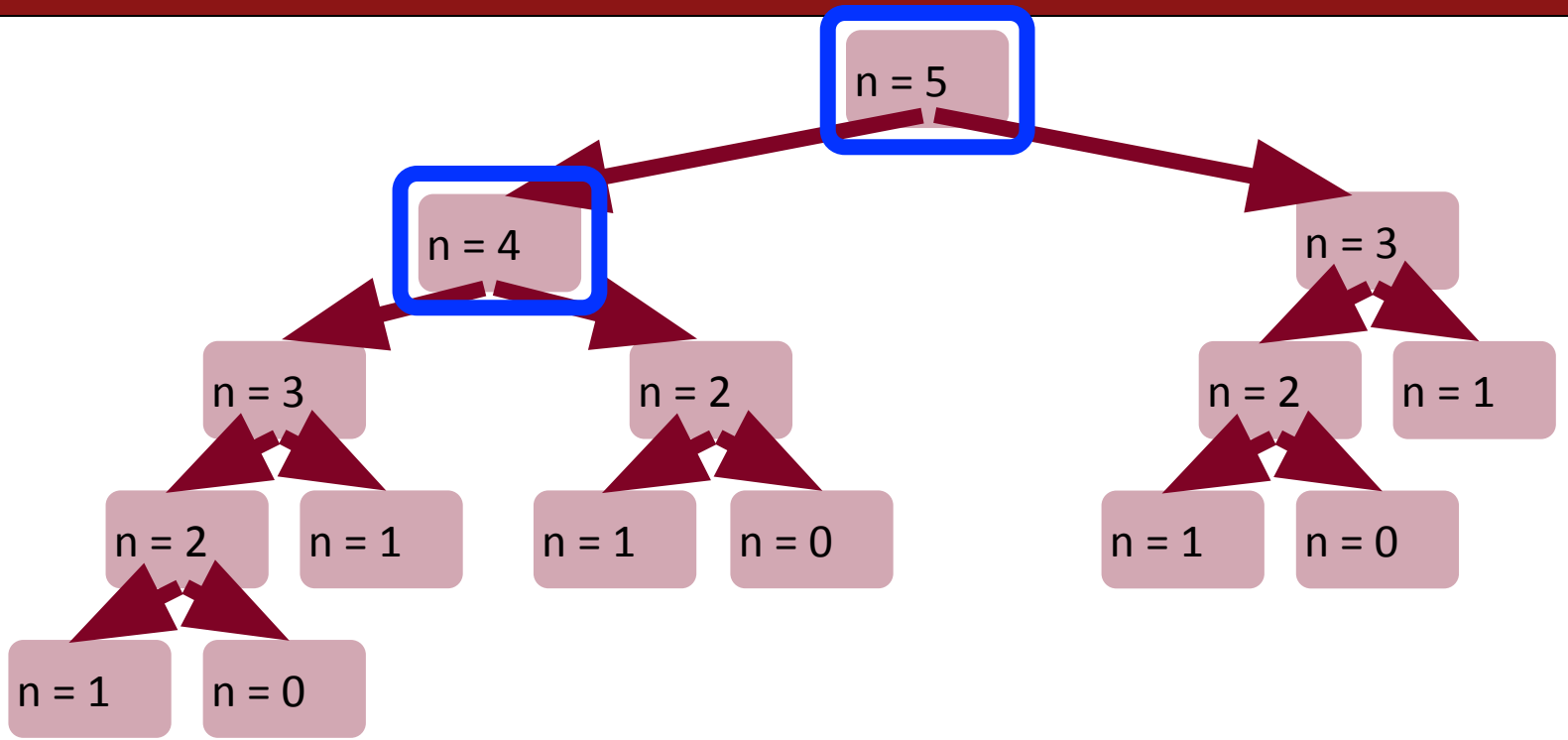
Memoization - Fibonacci



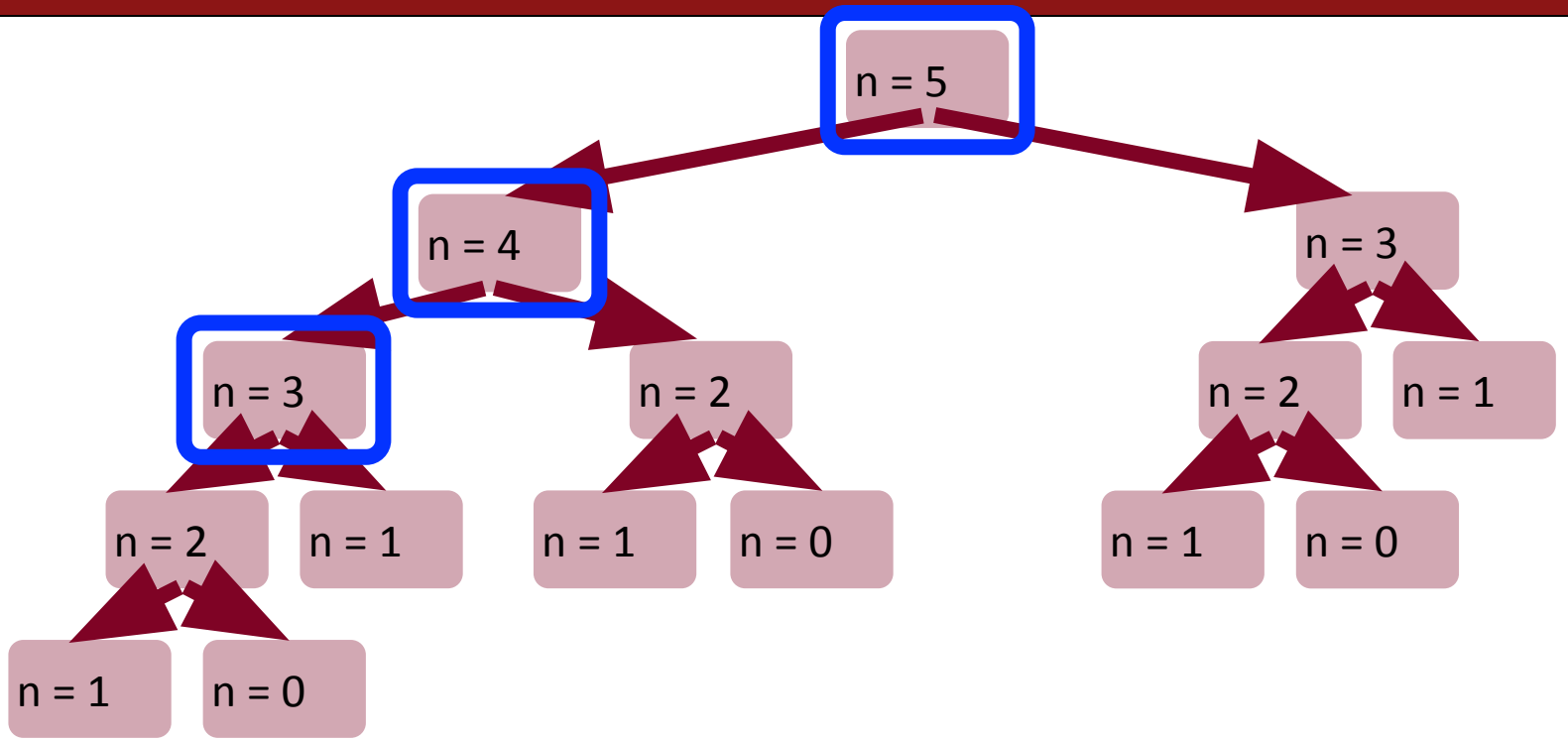
let's leverage all the repeats!



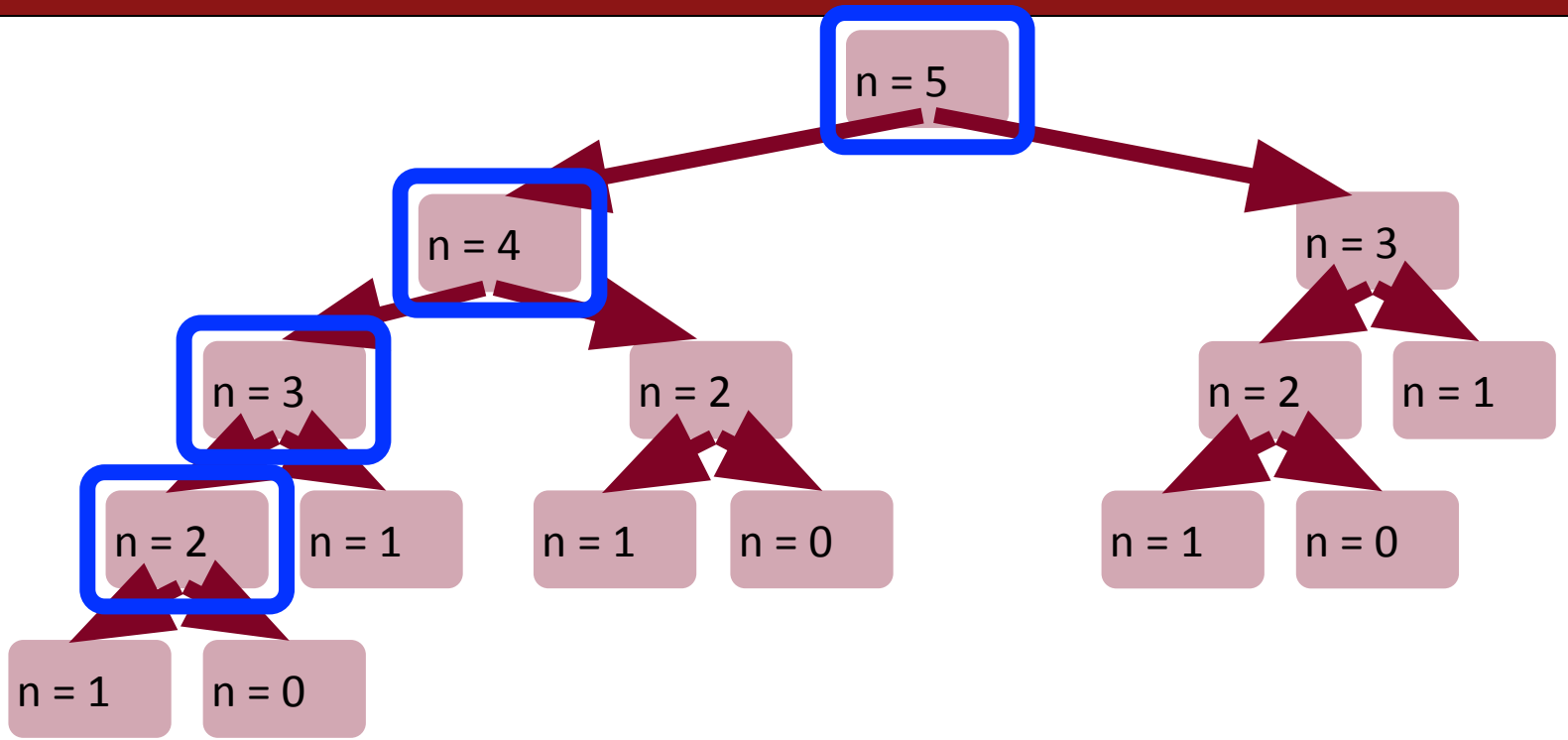
Cache: <empty>



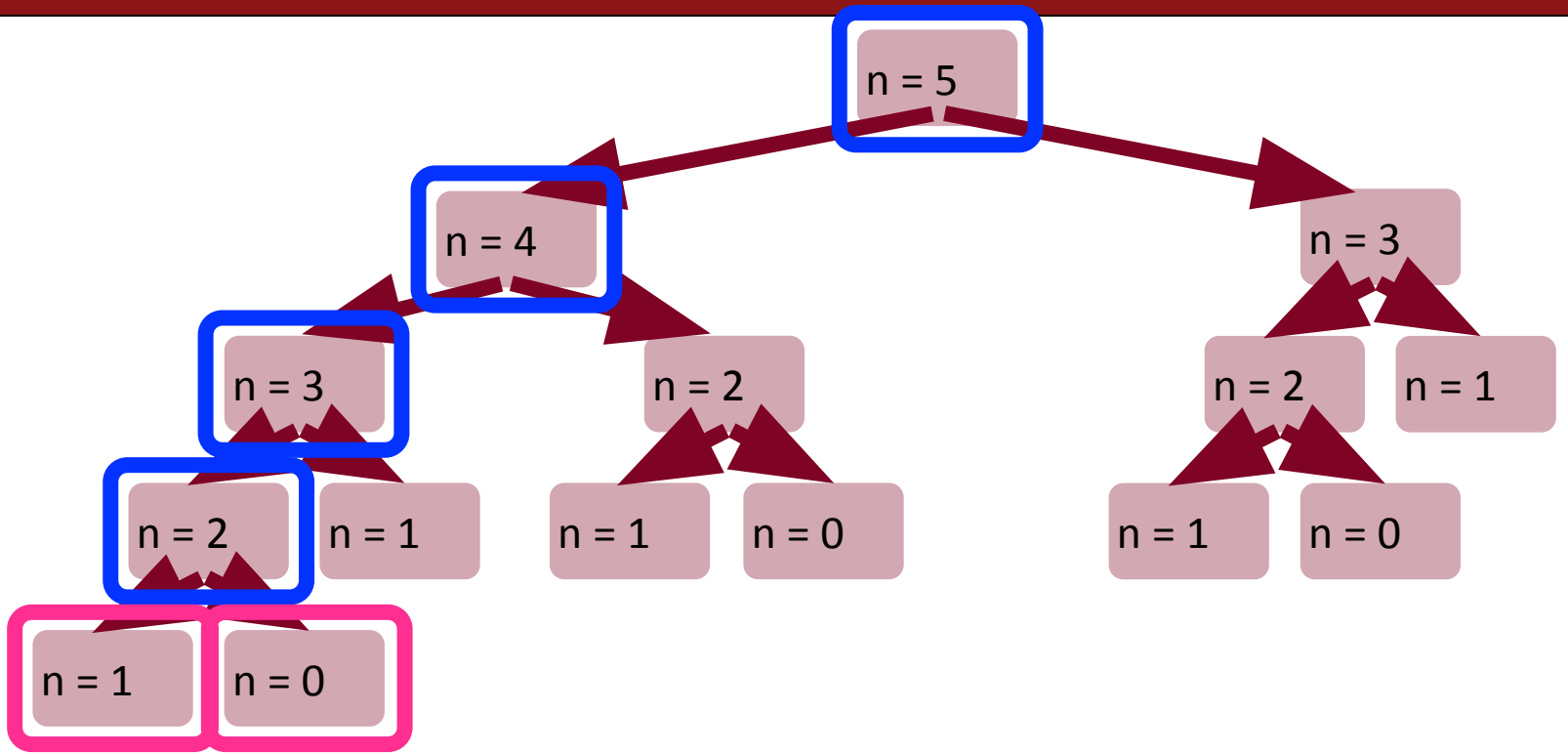
Cache: <empty>



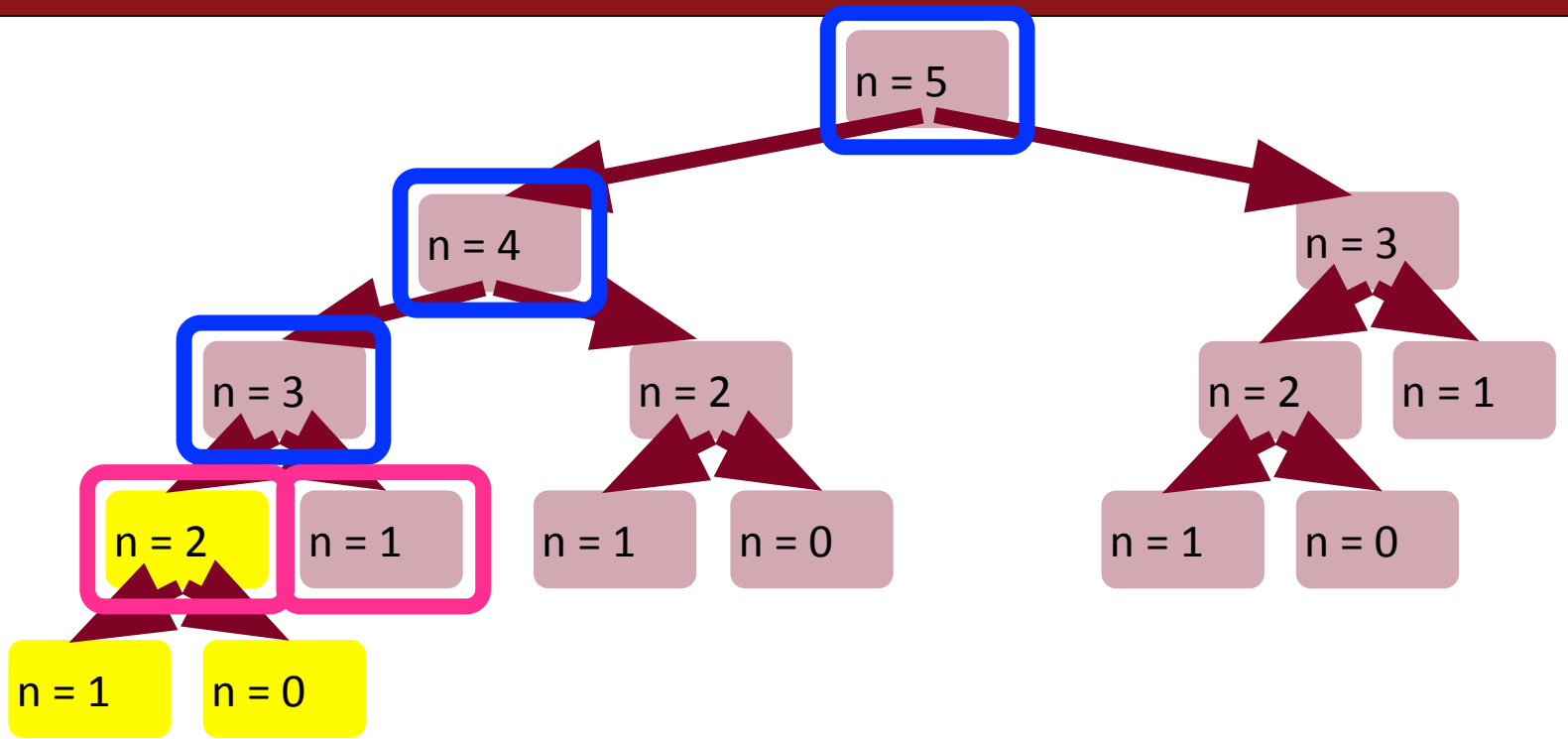
Cache: <empty>



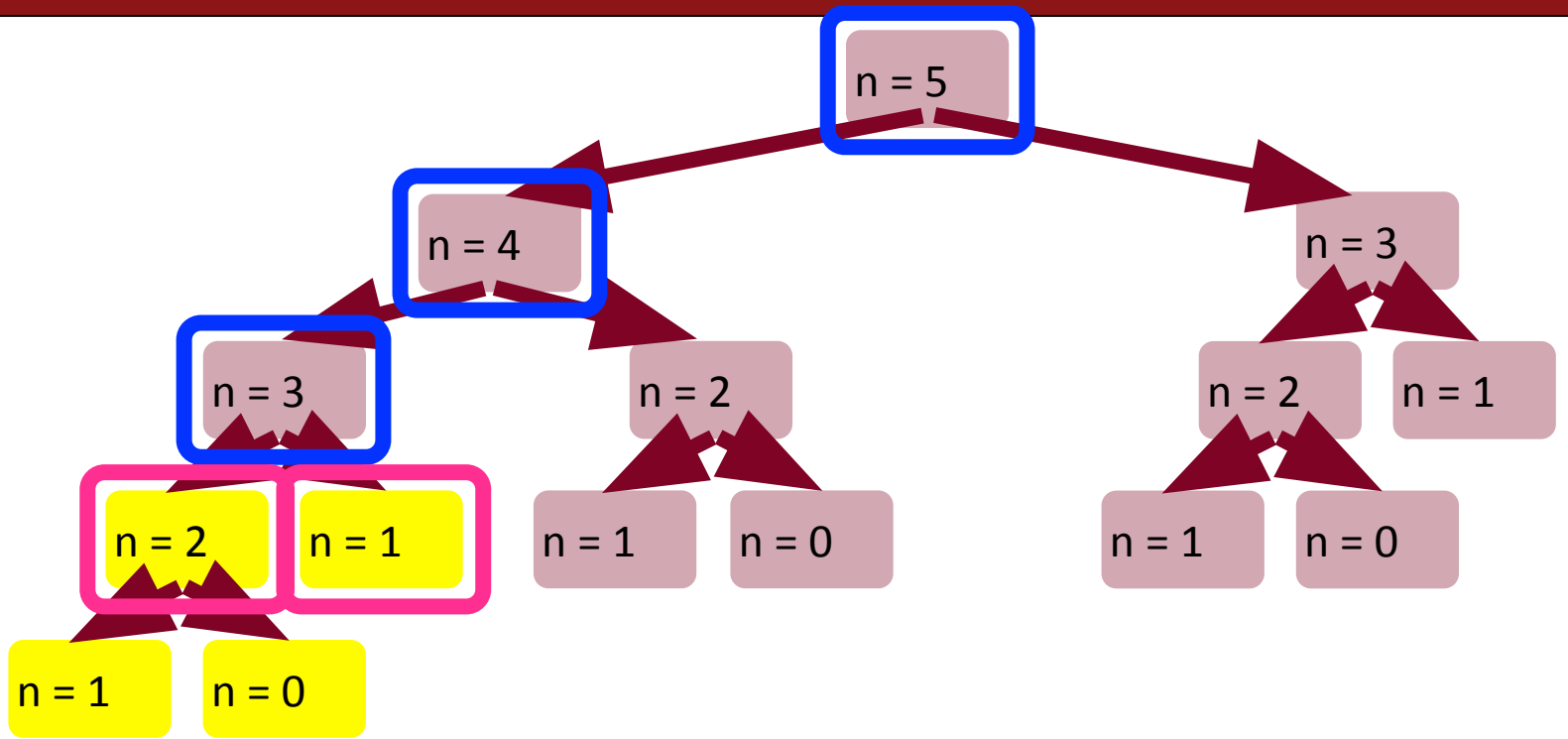
Cache: <empty>



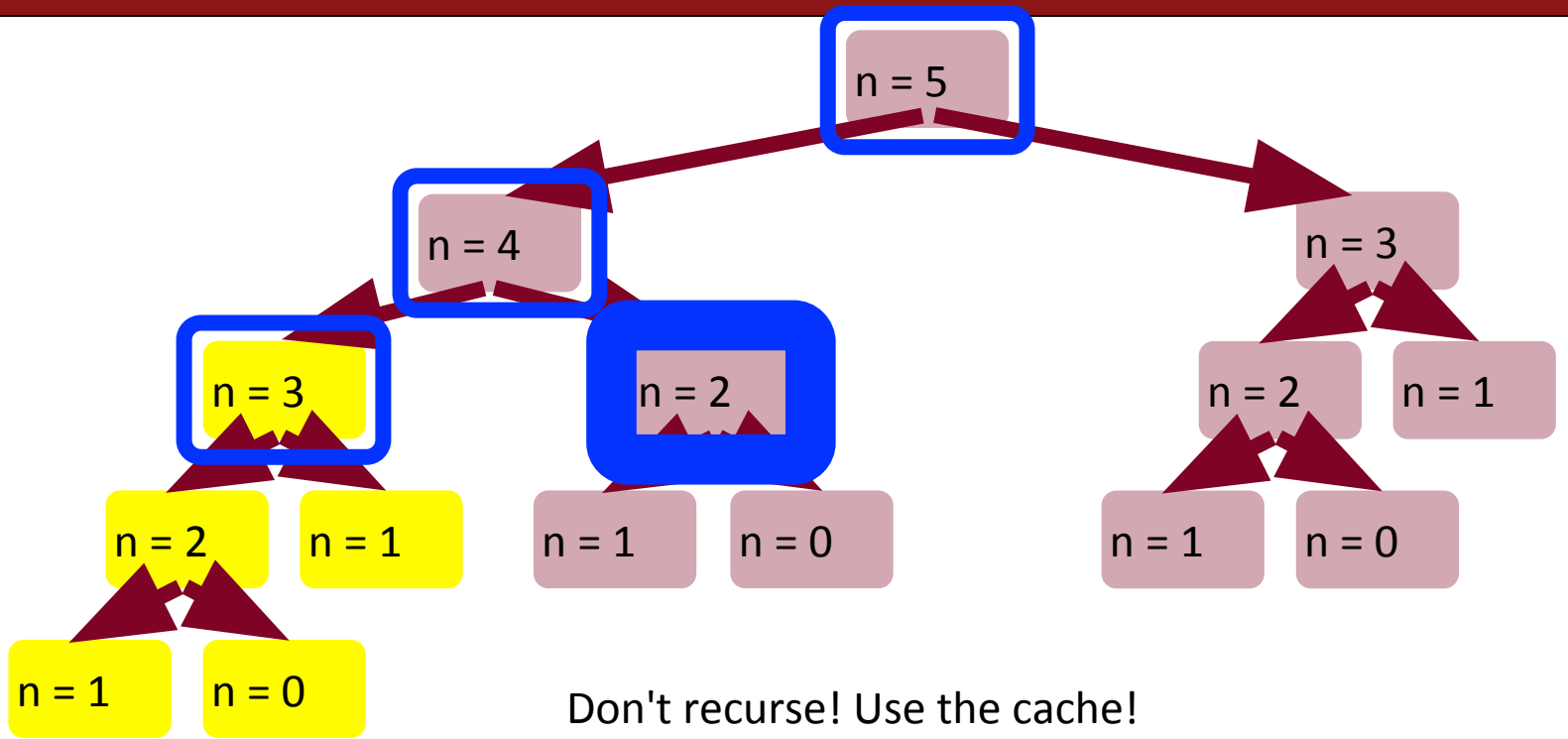
Cache: <empty>



Cache: fib(2) = 1

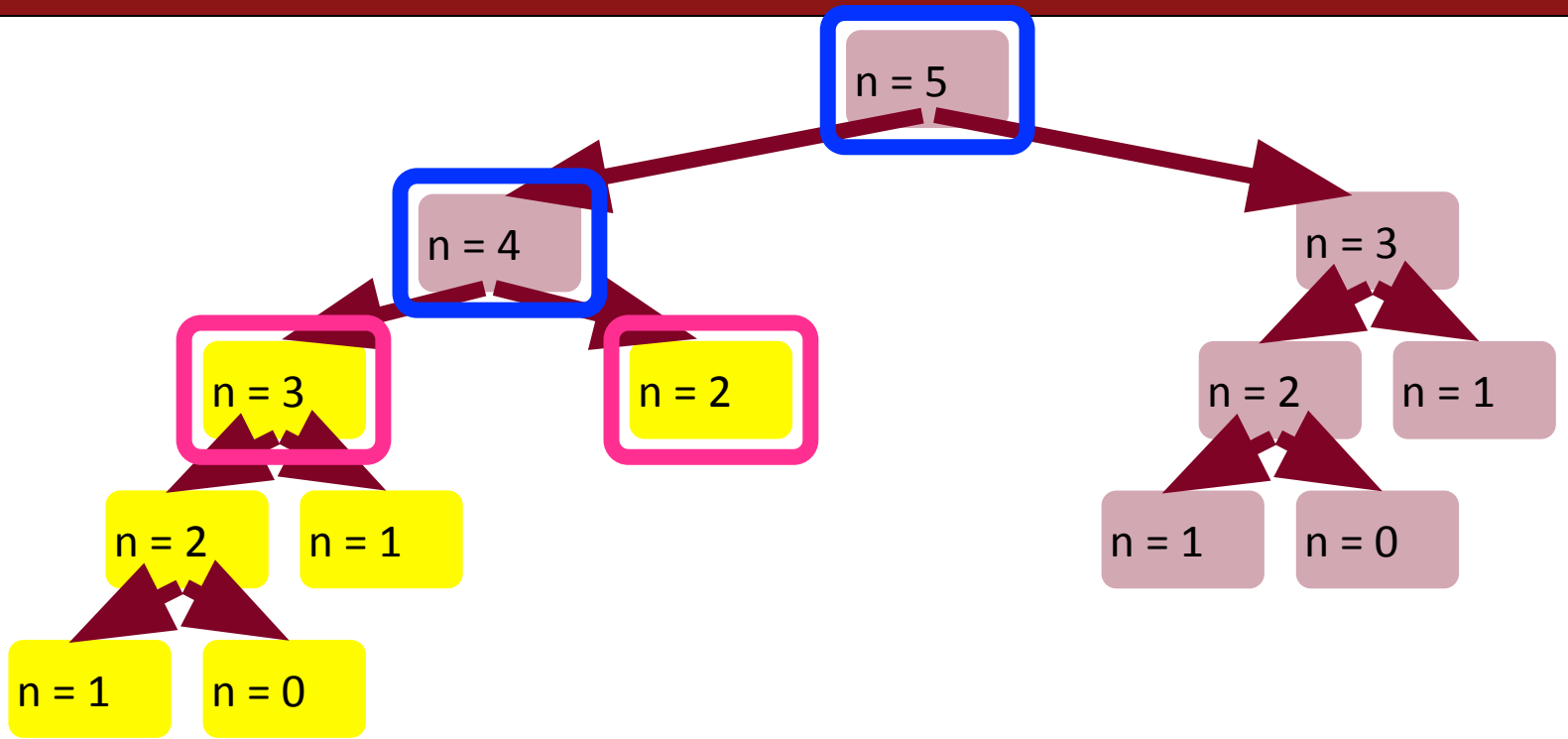


Cache: fib(2) = 1, fib(3) = 2

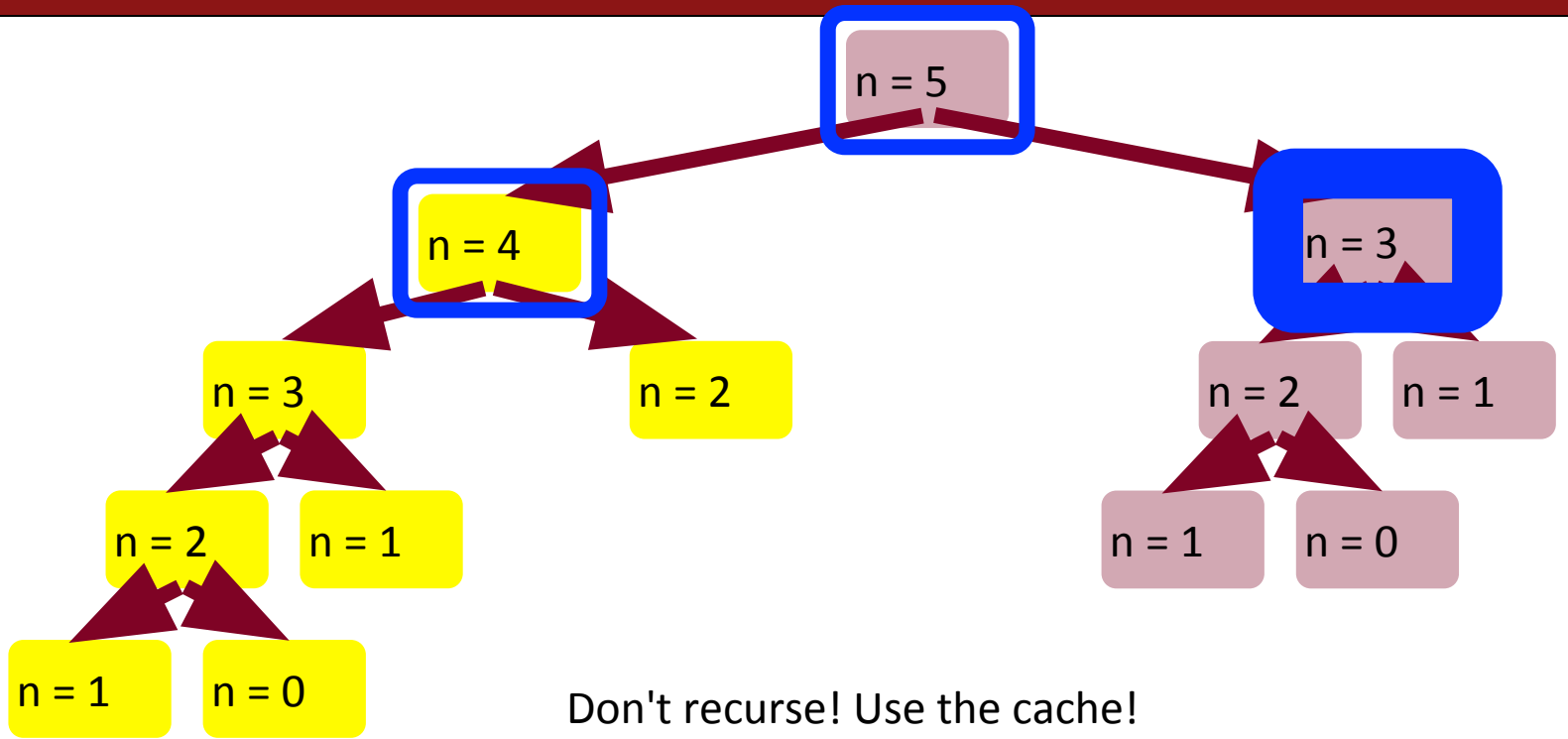


Don't recurse! Use the cache!

Cache: fib(2) = 1, fib(3) = 2

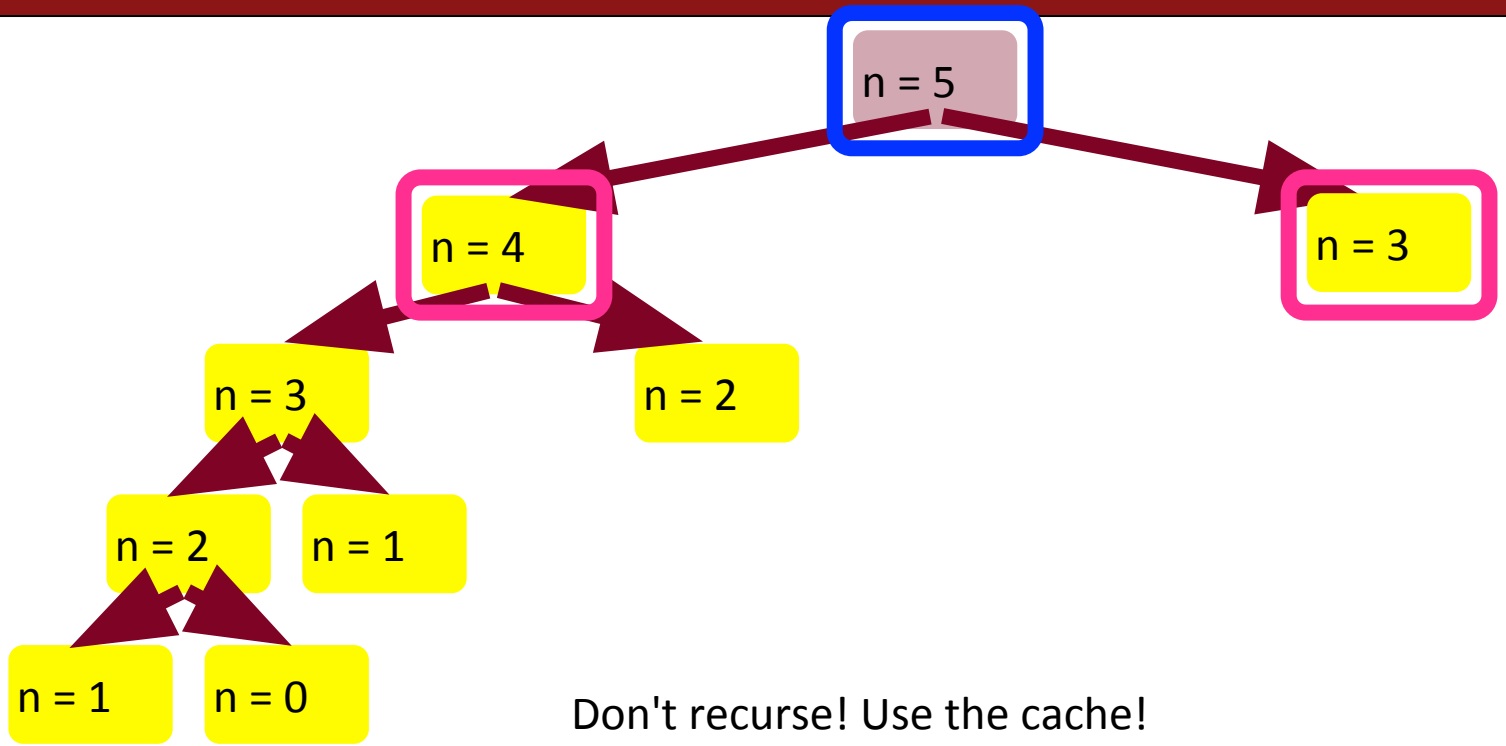


Cache: fib(2) = 1, fib(3) = 2



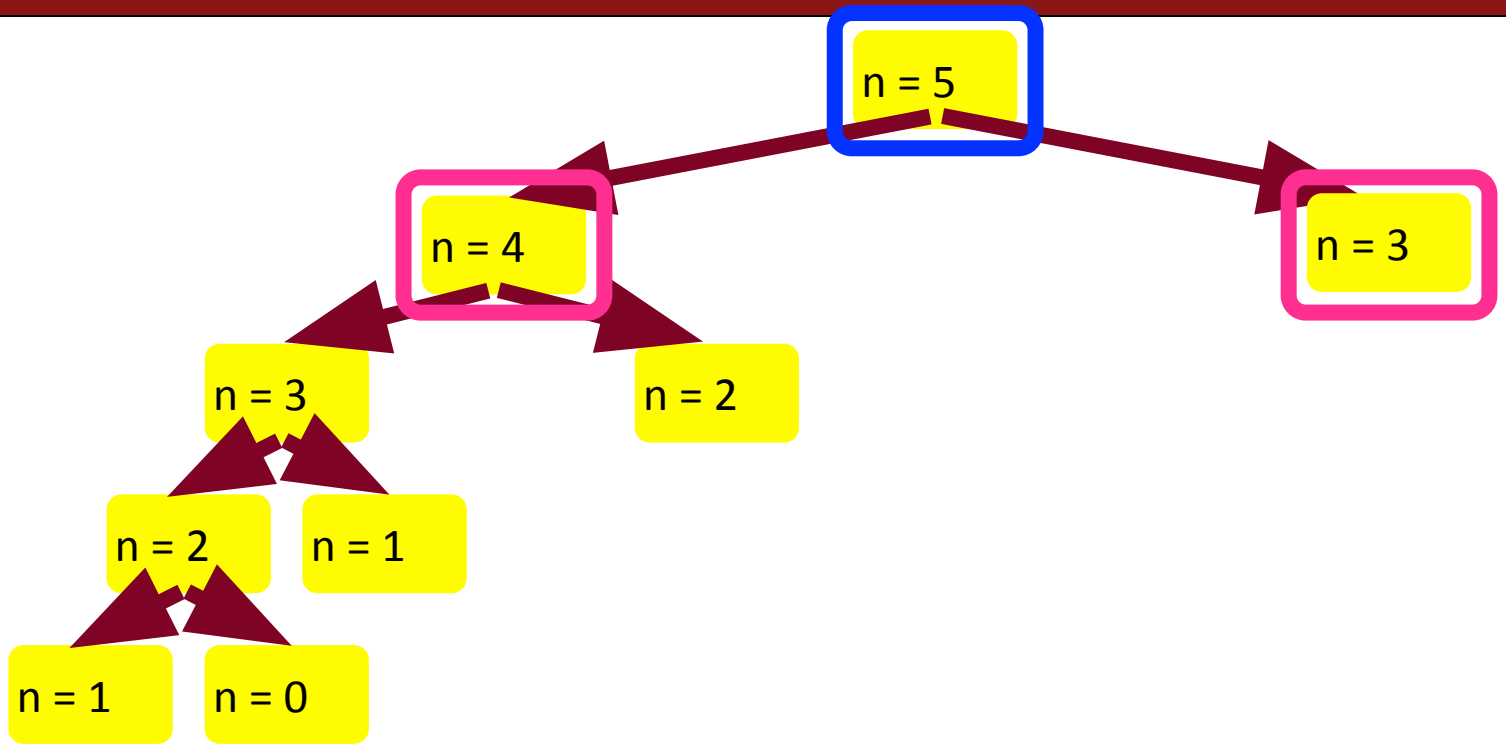
Don't recurse! Use the cache!

Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3

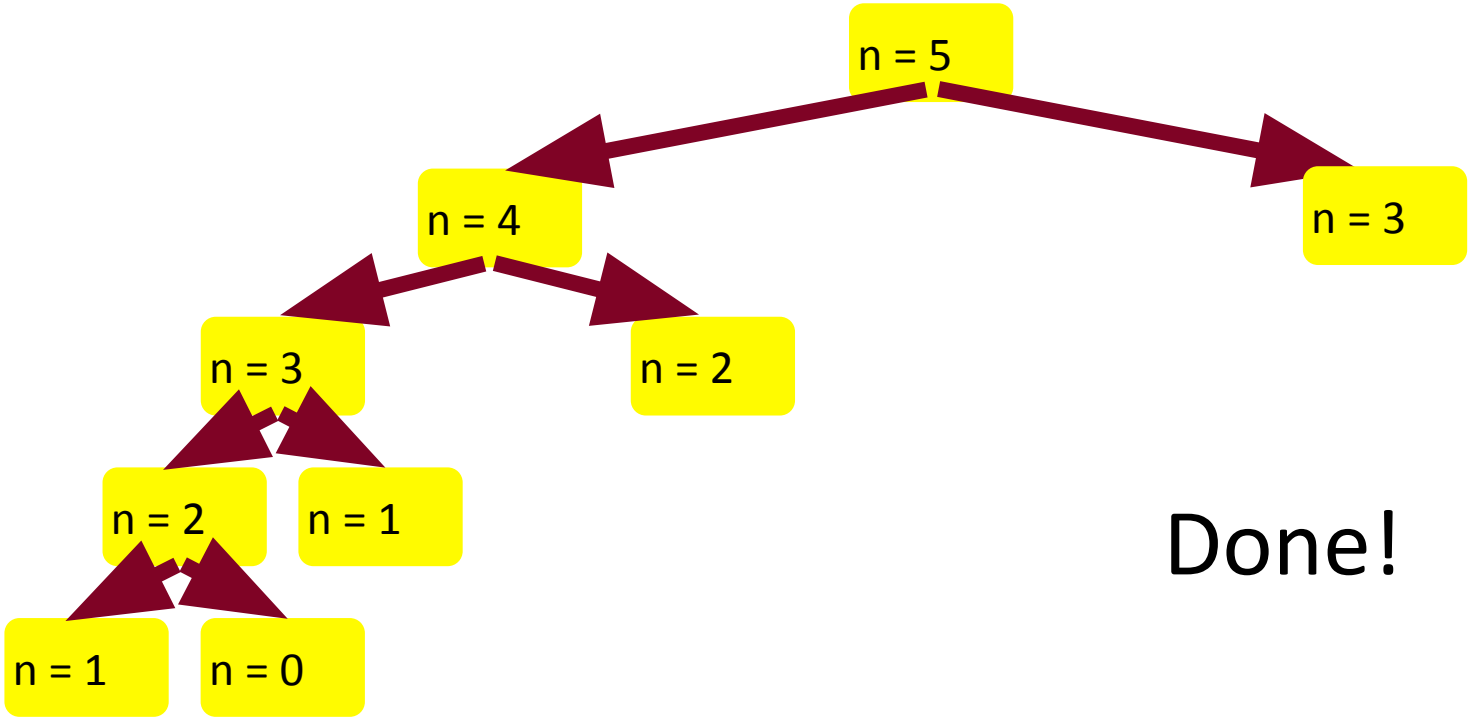


Don't recurse! Use the cache!

Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3



Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3, fib(5) = 5



Done!

Cache: fib(2) = 1, fib(3) = 2, fib(4) = 3, fib(5) = 5

Memoization

- Pass the cache as a parameter
- Before making a recursive call, check if the answer is in the cache
  - If so, return it.
  - If it's not, make the recursive call, and then the save the value in the cache for the future!

Fibonacci - Memoization

```
long memoizationFib(int n) {
    Map<int, long> cache;
    return memoizationFib(cache, n);
}
...
```

Fibonacci - Memoization

```
long memoizationFib(Map<int, long>&cache, int n) {
    if(n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if(cache.containsKey(n)) {
        return cache[n];
    }
    // recursive case
    long result = memoizationFib(cache, n-1)
                + memoizationFib(cache, n-2);
    cache[n] = result;
    return result;
}
```

Unit Testing

We have no idea how this will be tested...but here's a little review of it, just in case

¯_(ツ)_/¯

Unit Testing

- Unit Testing is a method for testing small pieces or ”units” of source code in a larger piece of software.
- Each test is usually represented as a single function.
- Key idea: each test should examine one portion of functionality that is as narrow and isolated as possible.
- Each test has a way of indicating pass or failure.
- Benefits:
  - Limits your code to only what is necessary
  - Finds bugs early
  - Preserves functionality when code is changed

Unit Testing

```
/* Info about the test */
ADD_TEST("Description of the test") {
  /* body of the test */
  expect(/* some condition you want */);
}
```


YOU CAN DO IT
I BELIEVE IN YOU
quickmeme.com