Programming Abstractions

CS106B

Cynthia Bailey
Chris Gregg

pollev.com/cs106b

Today's Topics

- Recursion!
  - Code up the Binary Search algorithm from last time
    - New decomposition concept: recursive helper function
  - Rewrite our Fibonacci code
    - New algorithm concept: memo-ization to improve efficiency
  - Use recursion to enumerate all possible sequences
    - Assignment 3 hints!
- Next week:
  - More advanced recursion
  - "Backtracking" recursion for solving tricky problems like mazes

- **For important announcements, be sure to see the weekly announcements post on the Ed Q&A board! https://edstem.org**
- **Also on Ed: live lecture Q&A with Chris & Jonathan**

Binary Search Refresher

(RECALL FROM WEDNESDAY'S LECTURE)

Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- **Efficiency Hack: Jump to the middle of the Vector and look there to find:**
  - X (answer Yes)
  - A number greater than X (rule out entire second half of Vector)

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

  - A number less than X (rule out entire first half of Vector)

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- Key observation: with *one* comparison, you ruled out *N/2* of the N cells in the Vector!

Does this list of numbers contain X?

Context: we have a collection of numbers in a Vector, in sorted order.

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- **Extreme Efficiency Hack: Keep jumping to the middle!**
  - Let's say our first jump to the middle found a number less than X, so we ruled out the whole first half:

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

  - Now jump to the middle of the remaining second half:

0	1	2	3	4	5	6	7	8	9	10
2	7	8	13	25	29	33	51	89	90	95

- Key observation: we do one piece of work, then delegate the rest. **Recursion!!**

Binary Search Implementation

NOW WE UNDERSTAND THE APPROACH.

WHAT DOES THE CODE LOOK LIKE?

Binary Search pseudocode

From previous lecture

```
bool binarySearch(Vector<int>& data, int key)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key);
    } else {
        return binarySearch(data[second half only], key);
    }
}
```

Base cases

Recursive case

Binary Search pseudocode

```
bool binarySearch(Vector<int>& data, int key)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key);
    } else {
        return binarySearch(data[second half only], key);
    }
}
```

Issue: copying half the data into a new, smaller `Vector` each time would be very inefficient, and defeat the purpose of pass-by-reference.

Binary Search pseudocode

```
bool binarySearch(Vector<int>& data, int key, int start, int end)
{
    if (data.size() == 0) {
        return false;
    }
    if (key == data[midpoint]) {
        return true;
    } else if (key < data[midpoint]) {
        return binarySearch(data[first half only], key
    } else {
        return binarySearch(data[second half only], ke
    }
}
```

Idea: pass the whole `Vector` by reference, but add parameters that indicate the bounds on the portion of the `Vector` currently in use.

Design Tip: Recursive Helper Function

- Sometimes managing data within a recursive function requires extra parameters.
- We don't want to clutter up the parameter list that the outside world sees, so what should we do??

✓ Make the primary function basically just a quick pass-through to a recursive helper function that does the hard work (and takes some extra parameters to help it do that work).

✓ *In this class, unless otherwise stated, when you are asked to implement a function with a certain function signature, it's ok to add a recursive helper function behind the scenes.*

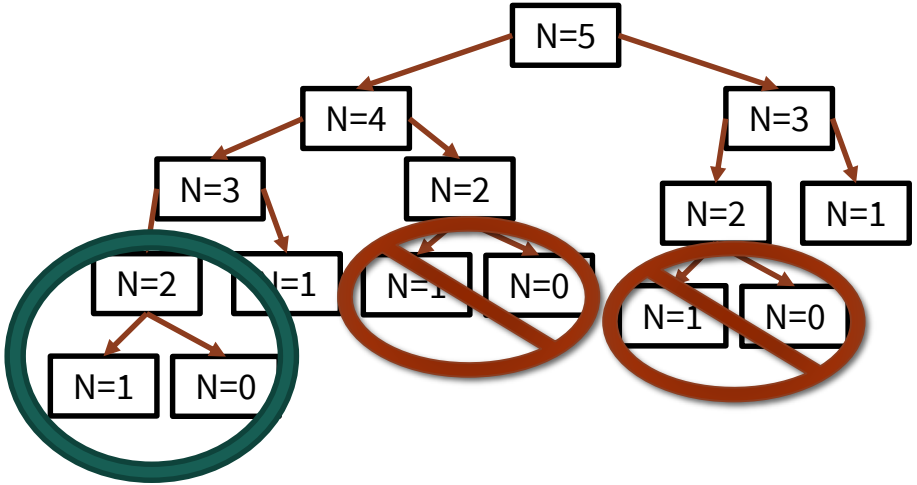
```
// this is the primary binary search function
bool binarySearch(const Vector<int>& data, int key) {
    return binarySearch(data, key, 0, data.size() - 1);
}

// this is a recursive helper function
bool binarySearch(Vector<int>& data, int key, int start, int end)
{
    if (start > end) {
        return false;
    }
    int mid = (start + end) / 2;
    if (key == data[mid]) {
        return true;
    } else if (key < data[mid]) {
        return binarySearch(data, key, start, mid - 1);
    } else {
        return binarySearch(data, key, mid + 1, end);
    }
}
```

Rewriting Fibonacci

USING **MEMOIZATION**
TO MAKE IT MUCH MORE
EFFICIENT

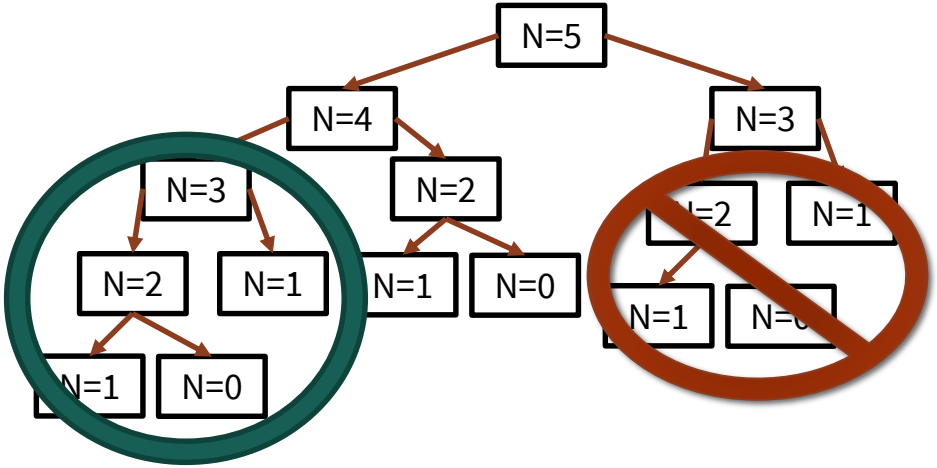
Rewriting Fibonacci

- Recall that the naïve recursive implementation involved a lot of wasteful duplicative work.
  - Once we calculate N=2 case the first time, let's just remember the answer is 1.
  - Now we don't have to do it again.



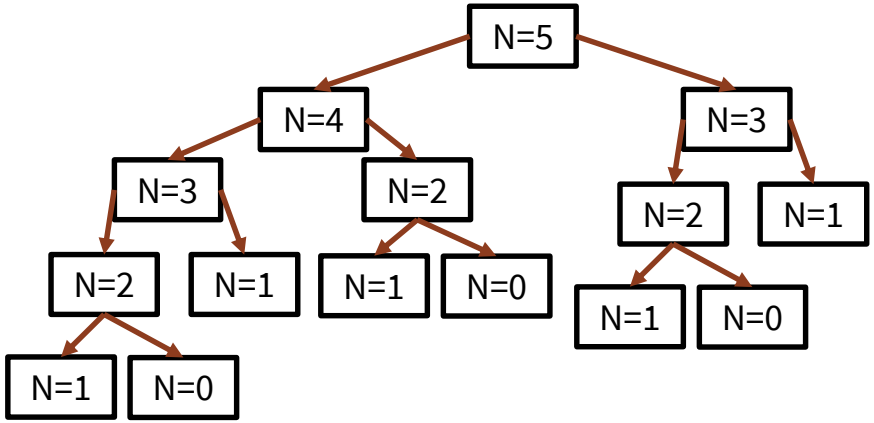
Rewriting Fibonacci

- Recall that the naïve recursive implementation involved a lot of wasteful duplicative work.
  - Once we calculate N=2 case the first time, let's just remember the answer is 1.
  - Now we don't have to do it again.
  - Once we calculate N=3 case the first time, let's just remember the answer is 3.
  - Now we don't have to do it again.

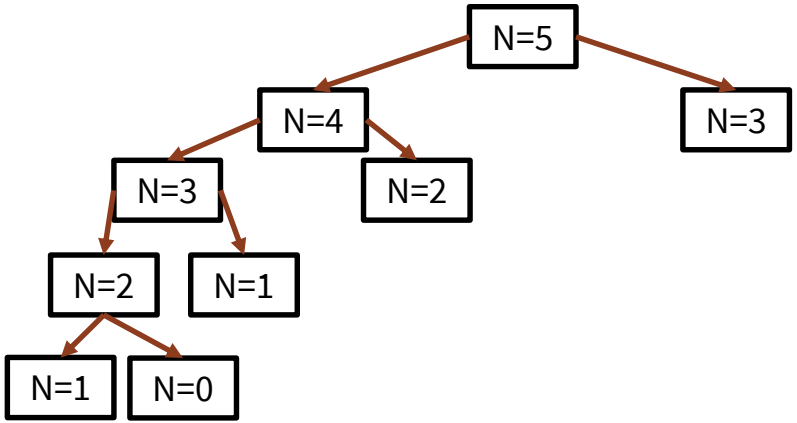


Rewriting Fibonacci

- Before



- After



Memoization implementation

```
// this is the primary function
int fasterFibonacci(int n)
{
    // this will hold our results as {n, Fib(n)}, starting with our base cases
    Map<int, int> memos = { {0, 0}, {1, 1} };
    return fasterFibonacci(n, memos);
}

// this is a recursive helper function
int fasterFibonacci(int n, Map<int, int>& memos)
{
    // if we already know the answer, just return it
    if (memos.containsKey(n)) {
        return memos[n];
    } else {
        // calculate new answer and save it for later
        int result = fasterFibonacci(n - 1, memos) + fasterFibonacci(n - 2, memos);
        memos[n] = result;
        return result;
    }
}
```

Another example of needing to make a recursive helper function that takes extra parameters.

Heads or Tails?

GENERATING SEQUENCES

Heads or Tails?

- You flip a coin 5 times
- What are all the possible heads/tails sequences you could observe?
  - TTTTT
  - HHHHH
  - THTHT
  - HHHHT
  - etc…
- We want to write a program to print each of the possible sequences.

Generating all possible coin flip sequences


```
void generateAllSequences(int length)
{
    string sequence;
    generateAllSequences(length, sequence);
}

void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

Your Turn: coin flip sequences


```
void generateAllSequences(int length)
{
    string sequence;
    generateAllSequences(length, sequence);
}

void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

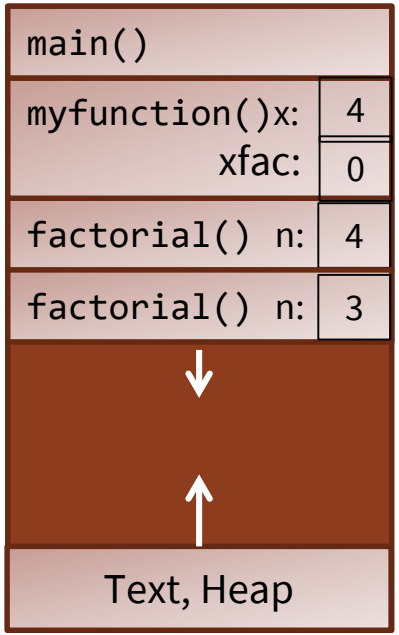
Q: Of these sequences (all of which should be printed), which sequence will be printed first? And which last?

- TTTTT, HHHHH, THTHT, HHHHT

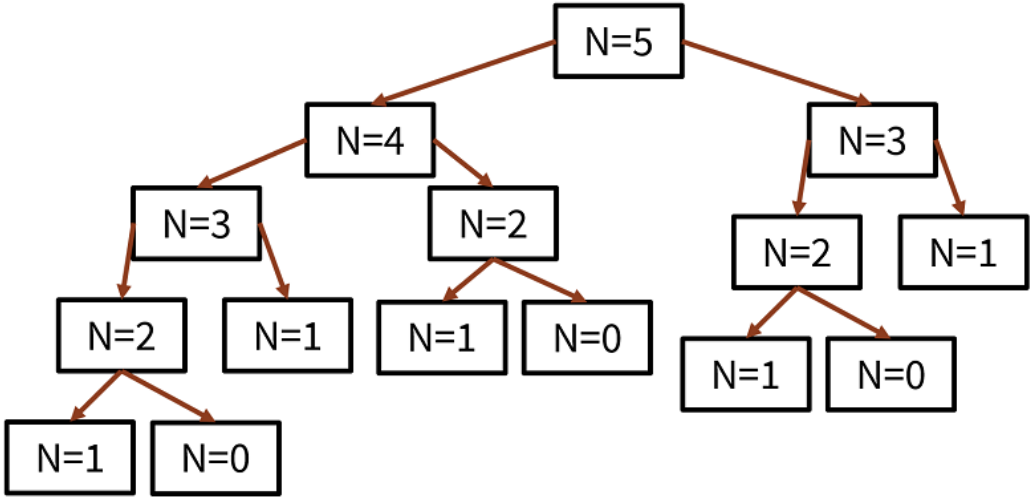
pollev.com/cs106b

Helpful mental models for recursion: the call stack, and the call tree

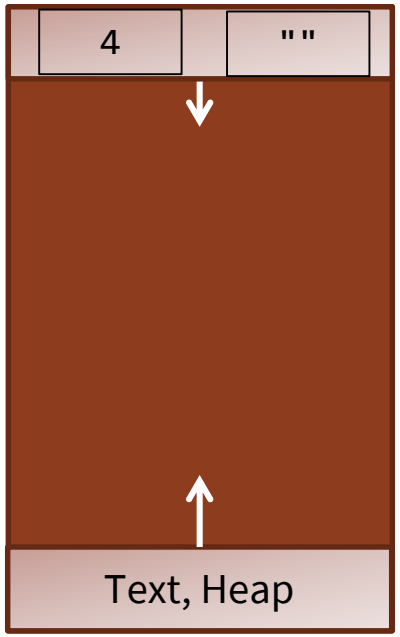
Remember we used this to help us understand **Factorial** recursion:



Remember we used this to help us understand **Fibonacci** recursion:



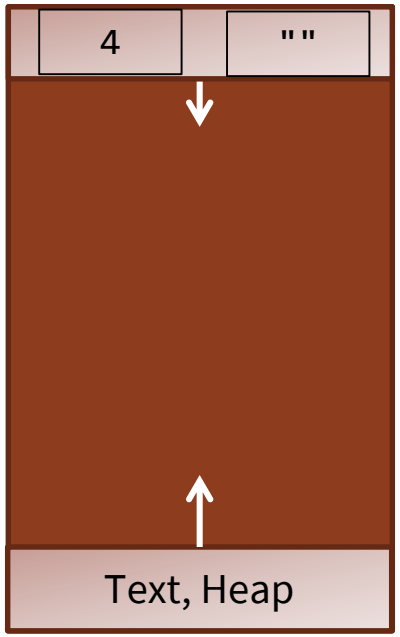
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

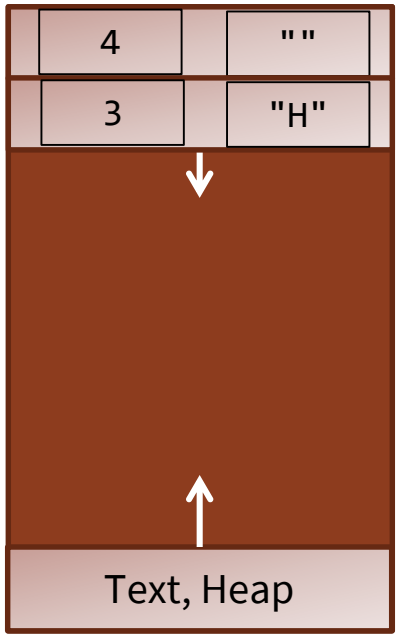
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

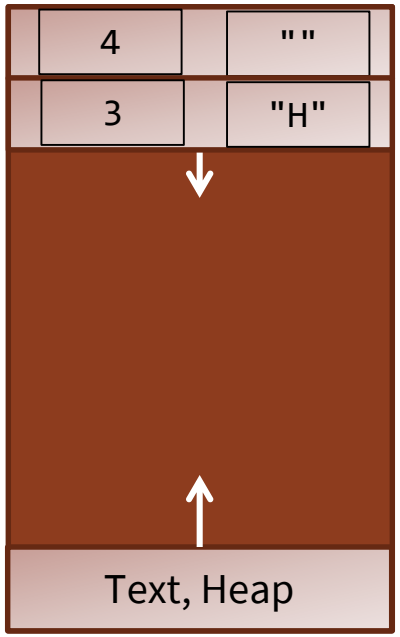
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

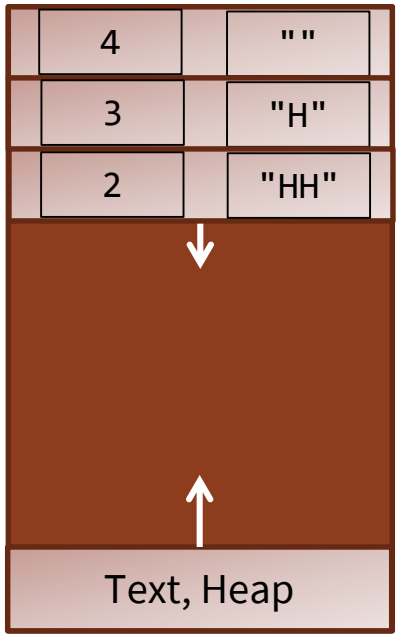
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

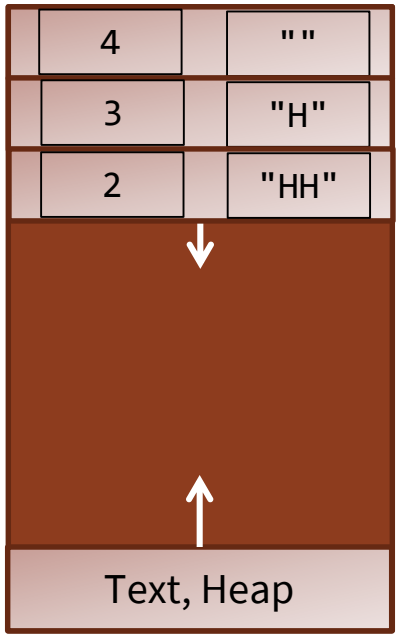
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

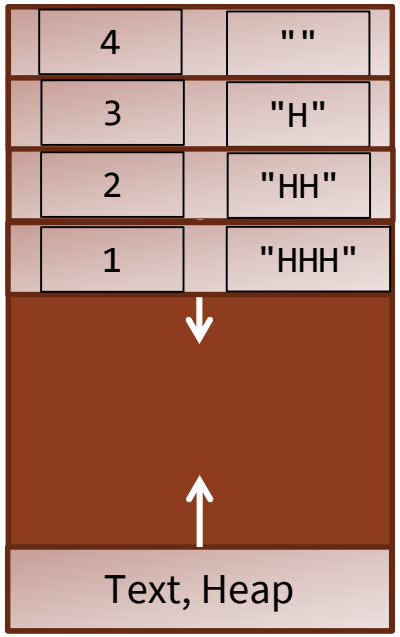
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

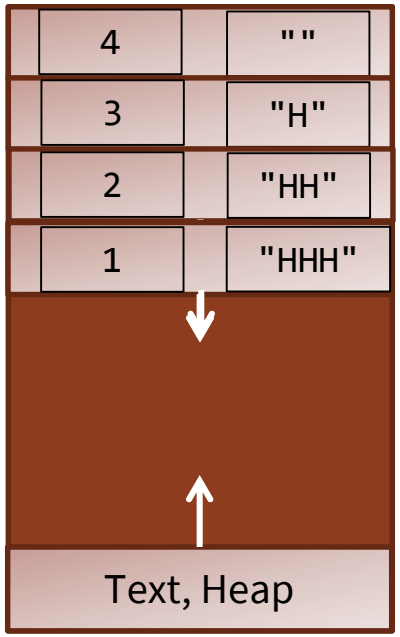
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

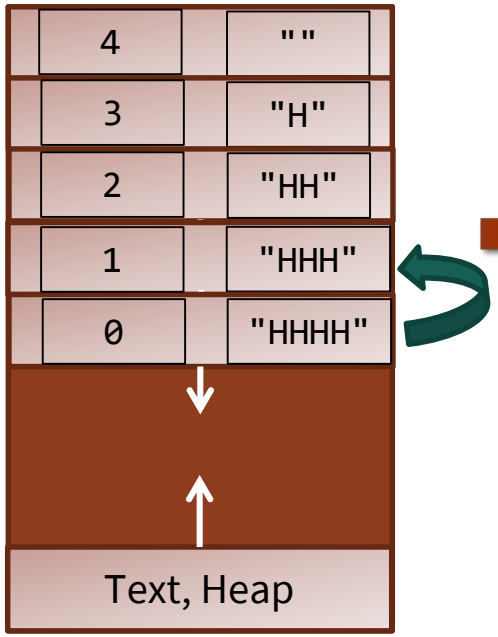
Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

Call stack for our Heads/Tails code

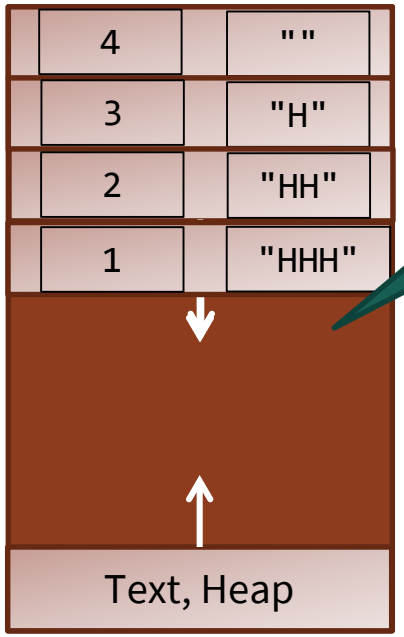


Recursive code

```
void generateAllSequences(int length
{
    // base case: no more flips to
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

Finally hit base case! Print HHHH and return.

Call stack for our Heads/Tails code



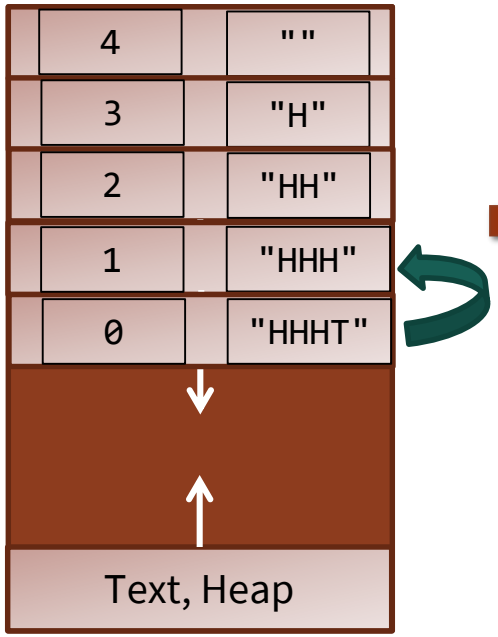
Most recent stack frame "popped" off the stack when we returned.

Recursive code

```
eAllSequences(int length, string sequence)

ase: no more flips to perform
if (length <= 0) {
    cout << sequence << endl;
    return;
}
// recursive cases: add H or T ar
generateAllSequences(length - 1, s uence + "H");
generateAllSequences(length - 1, sequence + "T");
}
```

We come back to this next line so it's time to add a T.

Call stack for our Heads/Tails code

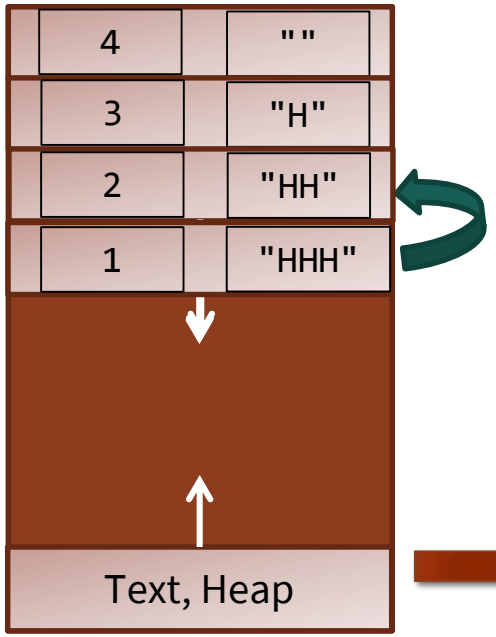


Recursive code

```
void generateAllSequences(int length
{
    // base case: no more flips to
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

At the base case again, we print HHHT and return.

Call stack for our Heads/Tails code

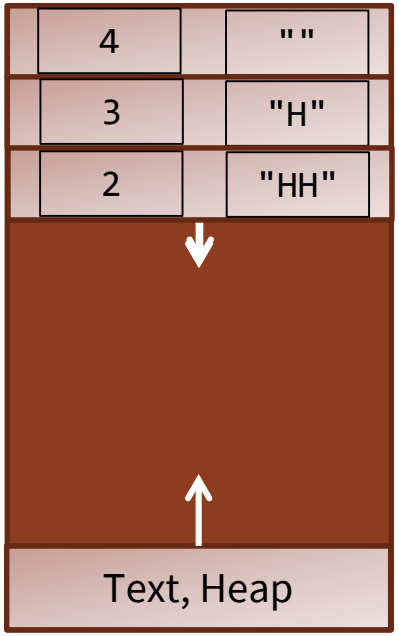


Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

This function call has reached the end (did both recursive calls), so it is done and it returns.

Call stack for our Heads/Tails code

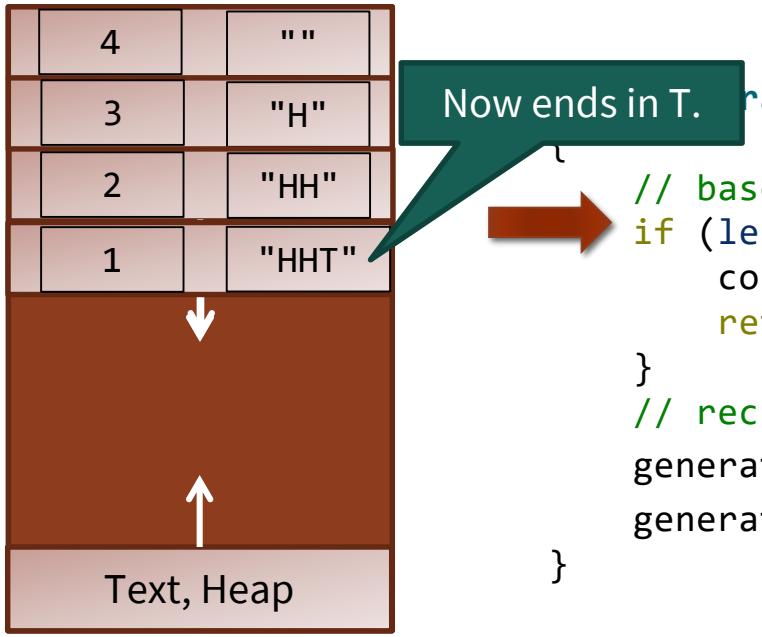


Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T ar
    generateAllSequences(length - 1, s      uence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

We come back to this next line so it's time to add a T.

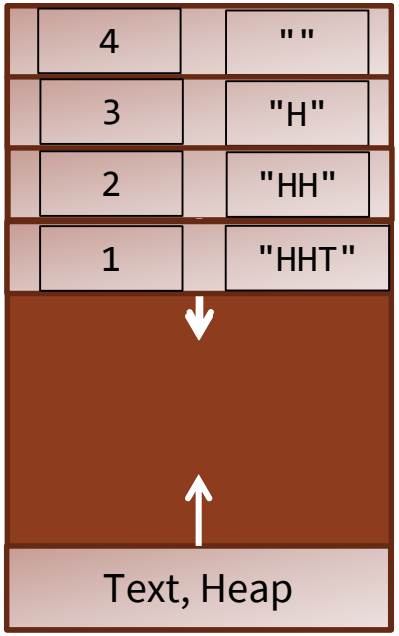
Call stack for our Heads/Tails code



Recursive code

```
rateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

Call stack for our Heads/Tails code

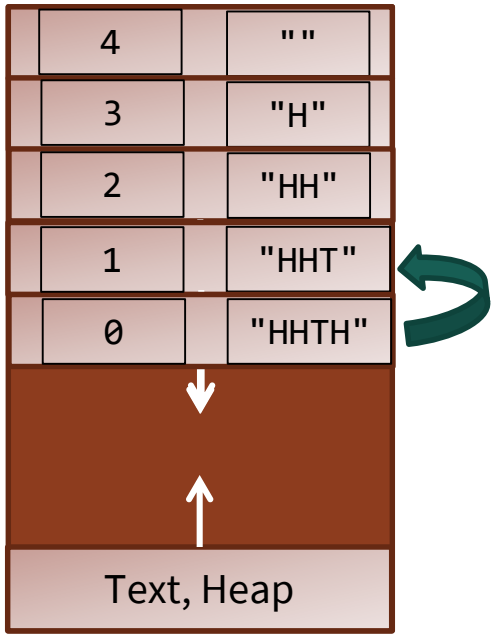


Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

Add an H after the T

Call stack for our Heads/Tails code

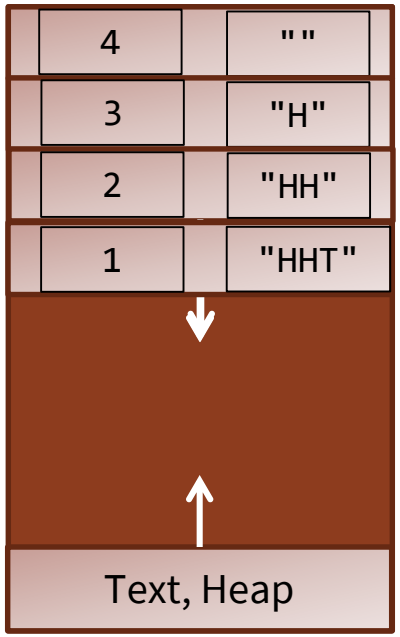


Recursive code

```
void generateAllSequences(int leng
{
    // base case: no more flips to pe
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

At the base case again, we print HHTH and return.

Call stack for our Heads/Tails code

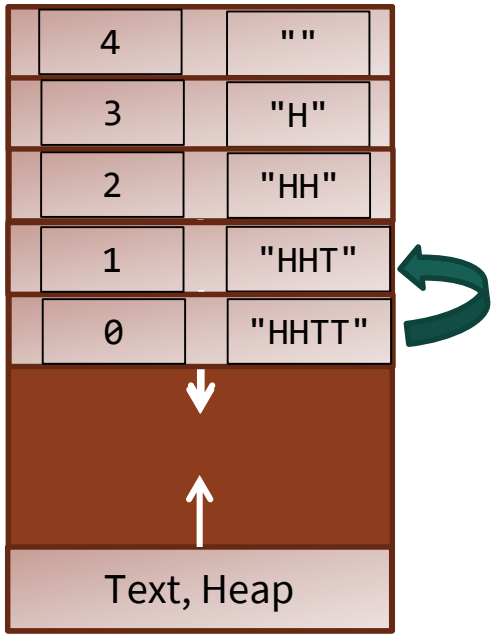


Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

Came back here, time to do the second recursive call.

Call stack for our Heads/Tails code

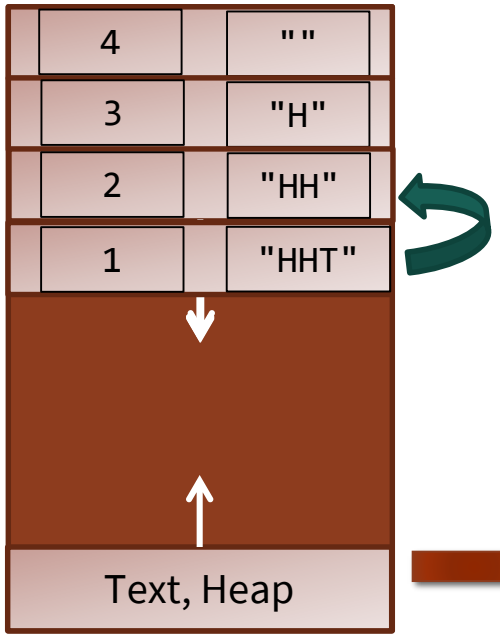


Recursive cod

```
void generateAllSequences(int len
{
    // base case: no more flips to p
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

At the base case again, we print HHTT as the fourth completed sequence, and return.

Call stack for our Heads/Tails code

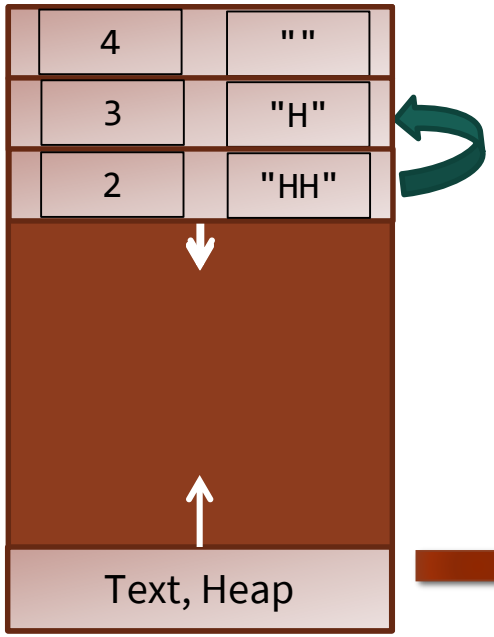


Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

This function call has reached the end (did both recursive calls), so it is done and it returns.

Call stack for our Heads/Tails code

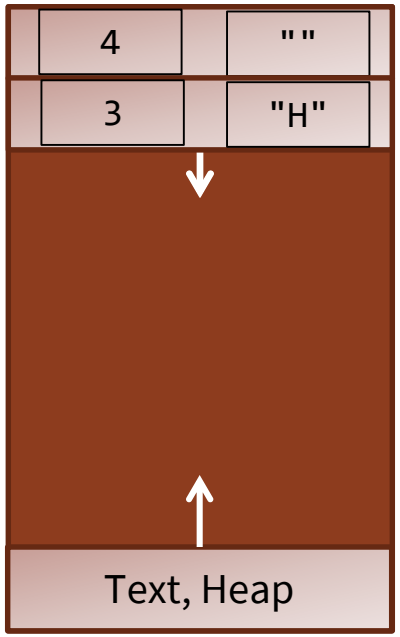


Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips to perform
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or T and continue
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

This function call has also reached the end (did both recursive calls), so it is also done and returns.

Call stack for our Heads/Tails code



Recursive code

```
void generateAllSequences(int length, string sequence)
{
    // base case: no more flips t
    if (length <= 0) {
        cout << sequence << endl;
        return;
    }
    // recursive cases: add H or
    generateAllSequences(length - 1, sequence + "H");
    generateAllSequences(length - 1, sequence + "T");
}
```

This function still needs to do its second recursive call with T, but we'll end our animation here. ☺

Call tree for Heads/Tails code

Labels are based on the value of the **sequence** parameter at the time of the function call (without add/erase/add edits we make inside the function).