Example: reverse a string

```c
void reverse(char *str, int len) {
    if (len > 1) {
        // swap characters
        char temp = str[len-1];
        str[len-1] = str[0];
        str[0] = temp;
        // recursive call
        reverse(str+1, len-2);
    }
}
```
Binary Search

1 2 5 10 15 20 22 30 35 40 43 48 51

low high

k = 48?

Binary Search

1 2 5 10 15 20 22 30 35 40 43 48 51

low mid high

k = 48?

Binary Search

1 2 5 10 15 20 22 30 35 40 43 48 51

low mid high

k = 48?

k = 22?

k = 0?

k = 51?

k = 55?
```c
#define NOT_FOUND -12345
int binsearch(int *A, int lo, int hi, int k) {
    // base case
    if (hi < lo) {
        return NOT_FOUND;
    }
    // calculate midpoint index
    int mid = (lo + hi) / 2;
    // key found?
    if (A[mid] == k) return mid;
    // key in upper subarray?
    if (A[mid] < k) return binsearch(A, mid+1, hi, k);
    // key is in lower subarray?
    return binsearch(A, lo, mid-1, k);
}
```

**Analysis of Binary Search**

**Base Case**

\[ T(1) = c_0 \]

**Recursive Case**

\[ T(n) = T(n/2) + c_1 \]

“A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.” (CLRS)
Recurrence relations

• By itself, a recurrence does not describe the running time of an algorithm
  ✓ need a closed-form solution (non-recursive description)
  ✓ exact closed-form solution may not exist, or may be too difficult to find

• For most recurrences, an asymptotic solution of the form $\Theta()$ is acceptable
  ✓ … in the context of analysis of algorithms

How to solve recurrences?

• By unrolling (expanding) the recurrence
  ✓ a.k.a. iteration method or repeated substitution

• By guessing the answer and proving it correct by induction

• By using a Recursion Tree

• By applying the Master Theorem

Unrolling a Recurrence

• Keep unrolling the recurrence until you identify a general case
  ✓ then use the base case

• Not trivial in all cases but it is helpful to build an intuition
  ✓ may need induction to prove correctness

Unrolling a Recurrence

\[
T(1) = c_0 \\
T(n) = T(n/2) + c_1
\]

\[= T(n/2^k) + kc_1\]
Applying the base case

We already know $T(1)$ is equal to a constant $c_0$:

$T(n) = T(n/2^k) + kc_1$

$= c_0 + c_1 \log n$

$= O(\log n)$

Recursion Tree (binary search)

$T(1) = c_0$

$T(n) = T(n/2) + c_1$

$= c_0 + c_1 \log n$

$= O(\log n)$

Example

$T(1) = a$

$T(n) = 2T(n/2) + n$
Example

\[ T(1) = 1 \]
\[ T(n) = T(n - 1) + c \]

Example

\[ T(0) = 0 \]
\[ T(n) = 2T(n - 1) + 1 \]

Example

```c
1 int power(int b, int n) {
2   // base case
3     if (n == 0) {
4       return 1;
5     }
6     // recursive call
7     return b * power(b, n-1);
8 }
```

Unimodal arrays

- An array is (strongly) **unimodal** if it can be split into an increasing part followed by a decreasing part
  
  <table>
  | 1 | 2 | 5 | 8 | 15 | 20 | 22 | 20 | 15 | 12 | 10 | 8 | 5 |
  </table>

- An array is (weakly) **unimodal** if it can be split into a nondecreasing part followed by a nonincreasing part
  
  <table>
  | 1 | 2 | 5 | 5 | 15 | 20 | 22 | 22 | 35 | 38 | 13 | 8 | 5 |
  </table>
Unimodal arrays

- An array is (strongly) unimodal if it can be split into an increasing part followed by a decreasing part
  
  1 2 5 8 15 20 22 20 15 10 8 5

- An array is (weakly) unimodal if it can be split into a nondecreasing part followed by a nonincreasing part
  
  1 2 5 5 15 20 22 22 35 20 13 8 5

Find the mode (strongly unimodal)

- Algorithm?

- Running Time?

Find the mode (weakly unimodal)

Two recursive calls
Find the mode (strongly unimodal)

- Algorithm?

- Running Time?

- Recursion Tree?