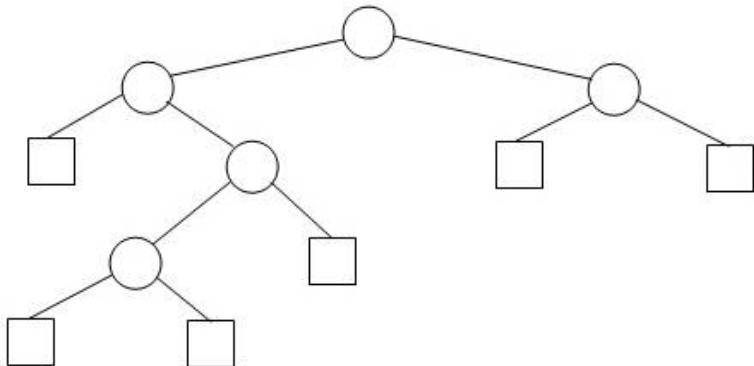


CSSE 230



Extended Binary Trees Recurrence relations

After today, you should be able to...
...explain what an extended binary tree is
...solve simple recurrences using patterns

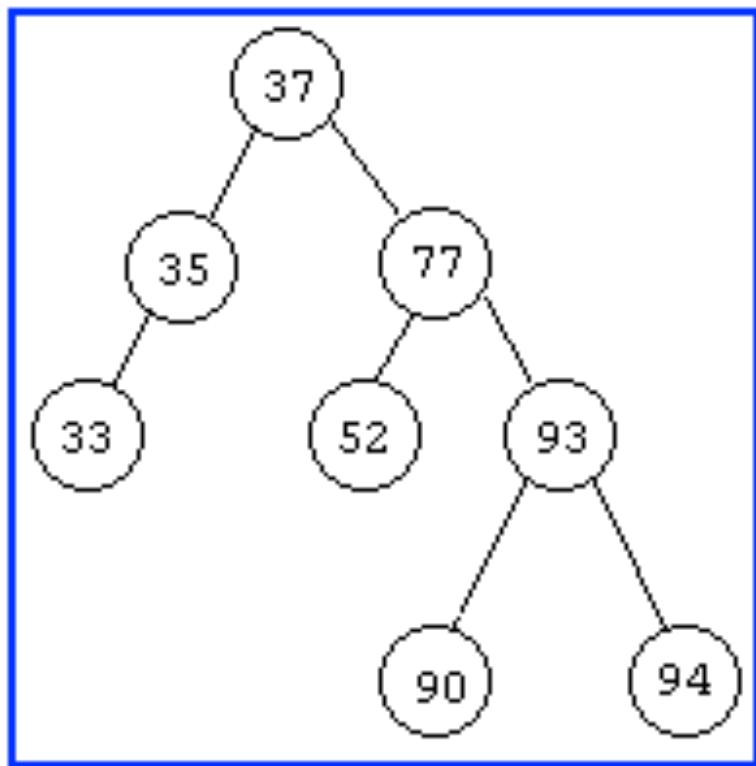
Reminders/Announcements

- ▶ Today:
 - Extended Binary Trees (on HW10)
 - Recurrence relations, part 1
- ▶ GraphSurfing Milestone 2
 - Permitted to work with a partner
 - Two additional methods: shortestPath(T start, T end) and stronglyConnectedComponent(T key)
 - Tests on Living People subgraph of Wikipedia hyperlinks graph
 - Bonus problem: find a “challenge pair”
 - Pair with as-long-as-possible shortest path

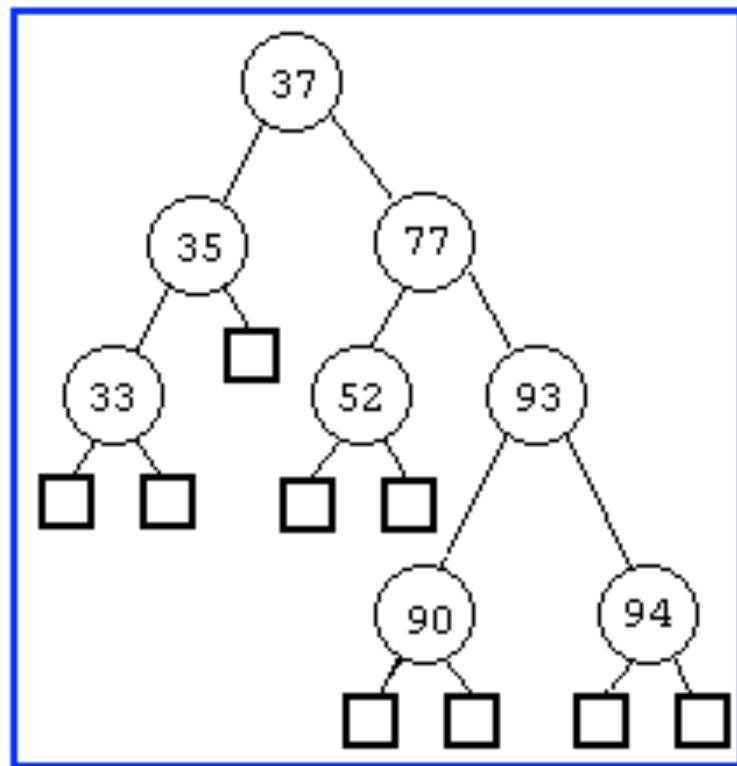
Extended Binary Trees (EBTs)

Bringing new life to Null
nodes!

Normal and Extended Binary Tree (EBT)



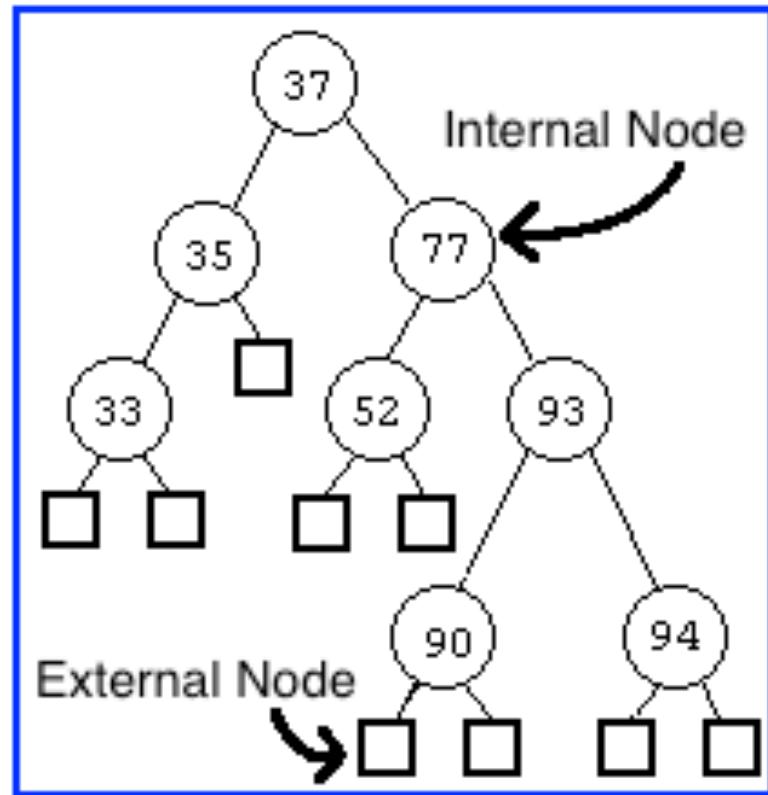
Normal Binary Tree



Extended Binary Tree

Extended Binary Tree (EBT)

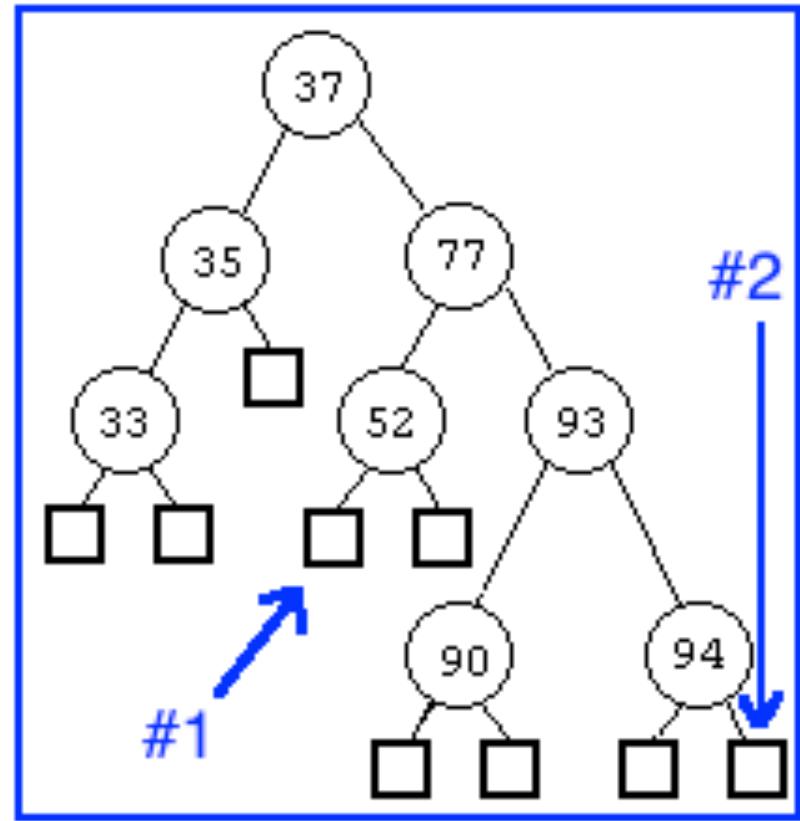
- ▶ Each node has either 0 or 2 children
- ▶ The *internal* nodes (2 children) of the EBT are the same as the original nodes in the normal binary tree
- ▶ The *external* nodes (0 children) are added to normal binary tree where NULL_NODES would normally have appeared



By convention
Internal nodes are circles
External nodes are squares

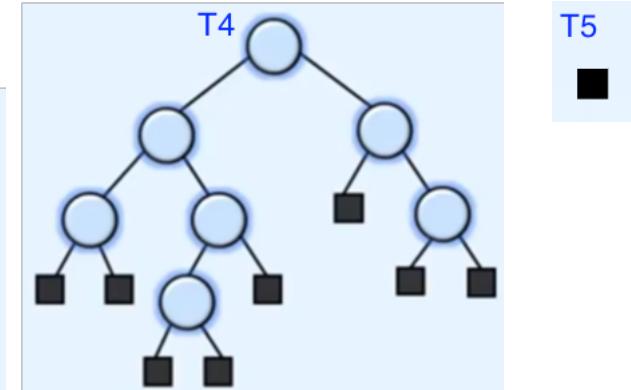
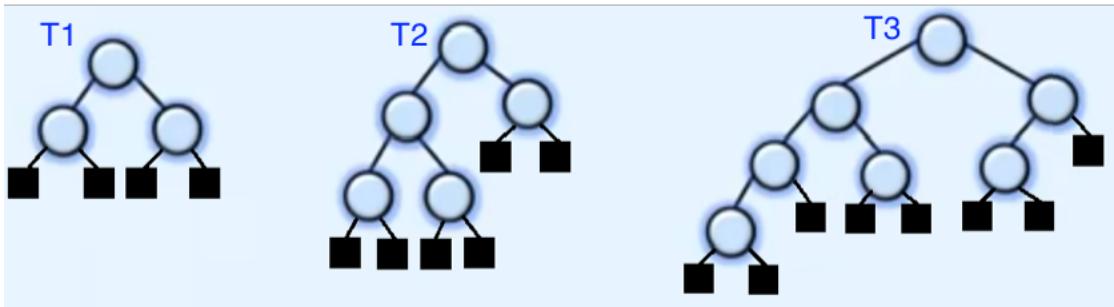
Extended Binary Tree (EBT)

- ▶ This is simply an alternative way of viewing binary trees: external nodes are “places” where a search can end or a new element can be inserted
- ▶ Question: For this EBT what are all the possible legal integer values that could end up at:
 - External node #1?
 - External node #2?



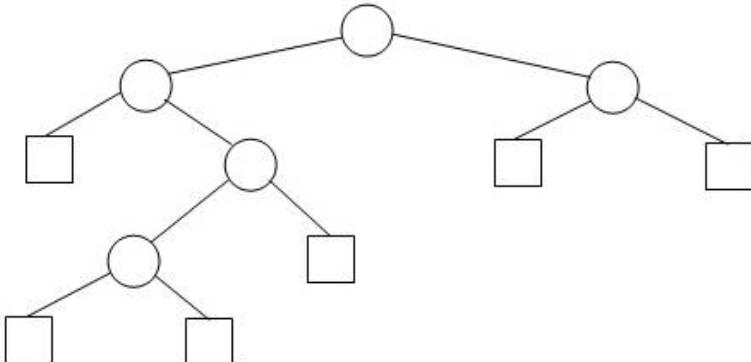
A property of EBTs

- ▶ **Property P(N):** For any $N \geq 0$, any EBT with N internal nodes has _____ external nodes.
(4) Write an equation for $\text{EN}(T)$ in terms of $\text{IN}(T)$
- ▶ Use example EBTs T_1 - T_4 below to come up with the equation, let:
 - $\text{EN}(T)$ = external nodes
 - $\text{IN}(T)$ = internal nodes



A property of EBTs

- ▶ Prove the equation from the previous slide using strong induction, based on the recursive definition.
 - A notation for this problem: IN(T), EN(T)



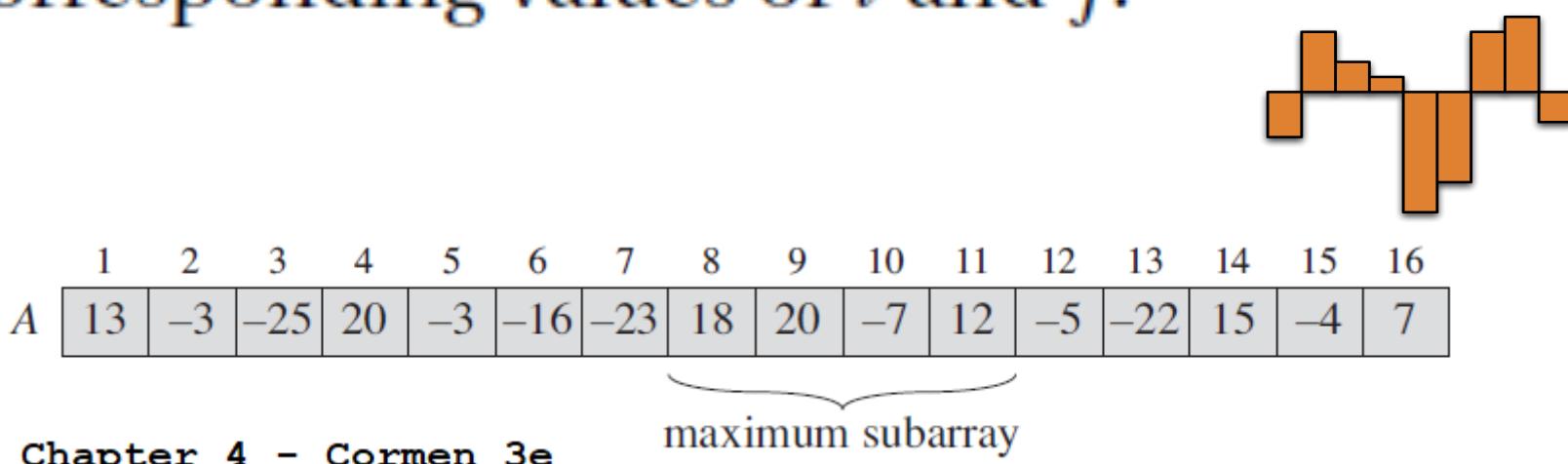
Hint (reminder): Find a way to relate the properties for larger trees to the property for smaller trees.

Introduction to Recurrence Relations

A technique for analyzing
recursive algorithms

Recap: Maximum Contiguous Subsequence Sum problem

Problem definition: Given a non-empty sequence of n (possibly negative) integers A_1, A_2, \dots, A_n , find the maximum consecutive subsequence $S_{i,j} = \sum_{k=i}^j A_k$, and the corresponding values of i and j .

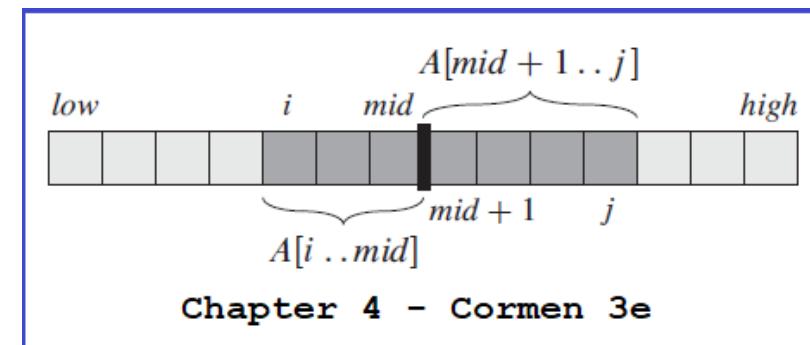
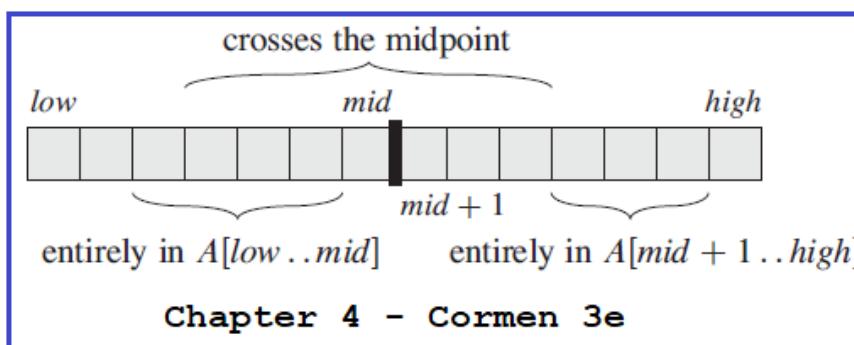


Divide and Conquer Approach

- ▶ Split the sequence in half
- ▶ Where can the maximum subsequence appear?

- ▶ Three possibilities :

 - entirely in the first half,
 - entirely in the second half, or
 - begins in the first half and ends in the second half



This leads to a recursive algorithm

1. Using recursion, find the maximum sum of first half of sequence
2. Using recursion, find the maximum sum of second half of sequence
3. Compute the max of all sums that begin in the first half and end in the second half
 - (Use a couple of loops for this)
4. Choose the largest of these three numbers

```
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right ) // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum );
}
```

N = array size

What's the run-time?

```
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right ) // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum );
}
```

Runtime =
Recursive part +
non-recursive part

Analysis?

- ▶ Write a **Recurrence Relation**

- $T(N)$ gives the run-time as a function of N
 - $T(N)$ has a two (or more) part definition:

- *Base case:*

- For example: $T(1) = c$

- Base case performance is constant, because base case is when problem is so simple, there's little work remaining to be done

- *Recursive case:*

- For example: $T(N) = T(N/2) + 1$

So, what's the recurrence relation for the recursive MCSS algorithm?

General Form – Recurrence

$$T(n) = aT(n/b) + f(n)$$

- ▶ a = # of subproblems
- ▶ n/b = size of subproblem
- ▶ $f(n) = D(n) + C(n)$
- ▶ $D(n)$ = time to divide problem before recursion
- ▶ $C(n)$ = time to combine after recursion

```

private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right ) // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum );
}

```

How many recursive calls?

Size of problem handed to recursive call(s)?

```
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right )
        return a[ left ];

    int maxLeftSum = 0;
    int maxRightSum = 0;

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                maxLeftBorderSum + maxRightBorderSum );
}
```

Big-O cost for $D(N) + C(N)$?
 $D = \text{divide}$
 $C = \text{combine}$

General Form – Recurrence

$$T(n) = aT(n/b) + f(n)$$

- ▶ a = # of subproblems
- ▶ n/b = size of subproblem
- ▶ $f(n) = D(n) + C(n)$
- ▶ $D(n)$ = time to divide problem before recursion
- ▶ $C(n)$ = time to combine after recursion

- ▶ For recursive MCSS:
- ▶ $T(1) = 1$
- ▶ $T(N) = 2T(N/2) + O(N)$

Recurrence Relation, Formally

- ▶ An equation (or inequality) that relates the n^{th} element of a sequence to certain of its predecessors (recursive case)
 - ▶ Includes an initial condition (base case)
 - ▶ **Solution:** A function of n .
-
- ▶ Similar to differential equation, but discrete instead of continuous
 - ▶ Some solution techniques are similar to diff. eq. solution techniques

Solve Simple Recurrence Relations

- ▶ One strategy: look for patterns

- Forward substitution
 - Backward substitution

- ▶ Examples:

As class:

1. $T(0) = 0, T(N) = 2 + T(N-1)$
2. $T(0) = 1, T(N) = 2 T(N-1)$
3. $T(0) = T(1) = 1, T(N) = T(N-2) + T(N-1)$

On quiz:

1. $T(0) = 1, T(N) = N T(N-1)$
2. $T(0) = 0, T(N) = T(N - 1) + N$
3. $T(1) = 1, T(N) = 2 T(N/2) + N$
(just consider the cases where $N=2^k$)

Next time: More solution strategies for recurrence relations

- ▶ Find patterns
- ▶ Telescoping
- ▶ Recurrence trees
- ▶ The master theorem

GraphSurfing Work Time