Recurrence Relations

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Outline

- Recursion
  - Simple warm up example (Factorial n)
- Recurrence Relations
  - Factorial N
  - Power N

Recursion

- What is Recursion?
  - Powerful, problem-solving strategy
  - Solves large problems by **reducing** them to **smaller** problems of the **<u>same form</u>**
- Example: Compute Factorial of a Number
  - 4! = 4 * 3 * 2 * 1 = 24
    - n! = n * (n-1) * (n-2) * … * 2 * 1
    - Also, 0! = 1
      - (just accept it!)

Recursion

- Example: Compute Factorial of a Number
  - Recursive Solution
    - Note that each factorial is related to a factorial of the next smaller integer
    - $n! = n * (n-1)!$
    - $4! = 4 * (4-1)! = 4 * (3!)$
    - But we need something else
      - We need a stopping case, or this will just go on and on and on
      - NOT good!
    - We let $0! = 1$
    - So in “math terms”, we say
      - $n! = 1$ if $n = 0$
      - $n! = n * (n-1)!$ if $n > 0$

Recursion

- Example: Compute Factorial of a Number
  - Recursive Solution --- in C code

```
int fact (int n) {
     if (n = 1 )
          return 1;
     else
          return (n * fact(n-1));
}
```

  - This is recursive. Why?
    - It defines the factorial of n in terms of the factorial of (n-1), thus reducing the problem

Recurrence Relations

- Today we go over Recurrence Relations
  - The Question: What is a recurrence relation?
    - an **equation** that defines a sequence recursively
      - each term of the sequence is defined as a function of the preceding term
  - What is the purpose?
    - In response, let us ask, what is the purpose using Summations in Big-O analysis?
    - Answer:
      - Summations are a tool to assist in measuring the running time of **iterative** algorithms

Recurrence Relations

- Today we go over Recurrence Relations
  - What is the purpose?
    - But can we use this same method of analysis, along with summations, to decipher the running time of recursive algorithms?
    - You cannot!
      - You cannot simply “eyeball” a recursive function for a minute or two, in the way you can an iterative function, and come up with a Big-O. Just doesn’t work.
    - So just like summations are a tool to help find the Big-O of <u>**iterative**</u> algorithms
    - **Recurrence Relations are a tool to help find the Big-O of <u>recursive</u> algorithms**

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
     if (n = 1)
          return 1;
     else
          return (n * fact(n-1));
}
```

- The GOAL:
  - We want to come up with an **equation** that properly expresses this `fact` function in a **recursive manner**.
  - Then we will need to **solve** this newly found equation.
    - We do so by putting it into its “closed form”.
  - Here’s the process…

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
     if (n = 1 )
          return 1;
     else
          return (n * fact(n-1));
}
```

  - What is happening in this problem?
    - At every step of the recursion,
      - meaning, each time the function is recursively called,
    - What happens? (i.e., what is going on with *n*)
      - We see that the input size (n) reduces by 1
      - So if n was 100, it is reduced to 99 when the function is called recursively for the first time.

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
      if (n = 1 )
            return 1;
      else
            return (n * fact(n-1));
}
```

  - What is happening in this problem?
    - Also, at every step of the recursion,
      - TWO mathematical operations are performed
        - The ‘*’ and the ‘-’ in `return (n * fact(n-1));`
    - So now we want to write an equation expressing these two facts.

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
     if (n = 1 )
          return 1;
     else
          return (n * fact(n-1));
}
```

  - What is happening in this problem?
    - We can say the following:
      - The total number of operations needed to execute this `fact` function for any given input, n, can be expressed as
      1) the sum of the 2 operations (the ‘*’ and the ‘-’)
      2) plus the number of operations needed to execute the function for n-1

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
     if (n = 1 )
          return 1;
     else
          return (n * fact(n-1));
}
```

- In techno talk:
  - Let T(n) represent the # of operations of this function,
  - T(n) can be expressed as a sum of:
    - T(n-1)
    - and the two arithmetic operations

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
     if (n = 1 )
          return 1;
     else
          return (n * fact(n-1));
}
```

  - In techno talk:
    - T(n) can be expressed as a sum of:
      - T(n-1)
      - and the two arithmetic operations

T(n) = T(n-1) + 2

T(1) = 1 Meaning, we it takes constant time to simply return.

Recurrence Relations

- Back to Factorial N…

```
int fact (int n)
{
     if (n = 1 )
          return 1;
     else
          return (n * fact(n-1));
}
```

  - So what did we just do?
    - We came up with an equation that properly expresses this fact function in a recursive manner.

      $T(n) = T(n-1) + 2$

      $T(1) = 1$
    - This equation is our Recurrence Relation

Recurrence Relations

- Back to Factorial N…
  - From this recurrence relation, T(n), we can come up with a Big-O
    - Great, so we solved it, so let's move on!
    - **Not so fast.**
  - As it is, the recurrence relation,

    $T(n) = T(n-1) + 2$

    $T(1) = 1$
  - doesn't tell us about the # of operations of T(n)
    - Does anyone know how many operations are in T(n-1)?
    - Is it 487 operations? Perhaps 515,243 operations?
    - We DON'T know!

Recurrence Relations

- Back to Factorial N…
  - The problem is only "**solved**" once we **remove all T(…)'s from the right side of the equation**
  - Again, here's the equation:

    $T(n) = T(n-1) + 2$
  - So T(n-1) needs to go bye-bye
  - Then the problem is in its "**closed form**" and is solved.
  - So how do we make this happen?

  - BUCKLE UP and HOLD ON.

Recurrence Relations

- Back to Factorial N
  - We need to solve T(n) in terms of n
  - For the recurrence relation,
    - $T(n) = T(n-1) + 2$
  - Do we know what T(n-1) equals?
    - Does it equal 8,572 operations?
  - Who knows? We surely don't know!
  - So we want to REDUCE the right side
    - specifically, the T(n-1)
  - UNTIL we get to that which we do know!
    - Meaning, something we KNOW to be a FACT

Recurrence Relations

- Back to Factorial N
  - We need to solve T(n) in terms of n
  - Starting from this equation:

    $T(n) = T(n-1) + 2$
  - We reduce the right side until we get to T(1).
  - Why?
    - CUZ we know T(1).
    - What is T(1)?
      - It is 1! …this was from our Recurrence Relation earlier.
    - So then we can put 1 in the place of T(1)
      - Effectively eliminating all T(…)s from the right side of eqn!

Recurrence Relations

- Back to Factorial N
  - We need to solve T(n) in terms of n

    $T(n) = T(n-1) + 2$
  - We reduce the right side until we get to T(1).
  - Here's the idea:

T(n-1)	if we assume that n = 100, we have…	T(100-1)
T(n-2)		T(100-2)
T(n-3)		T(100-3)
…		…
T(n-something) = T(1)		T(100-99) = T(1)

Recurrence Relations

- Back to Factorial N
  - We need to solve T(n) in terms of n

    $T(n) = T(n-1) + 2$
  - We reduce the right side until we get to T(1).
  - So, we do this in steps

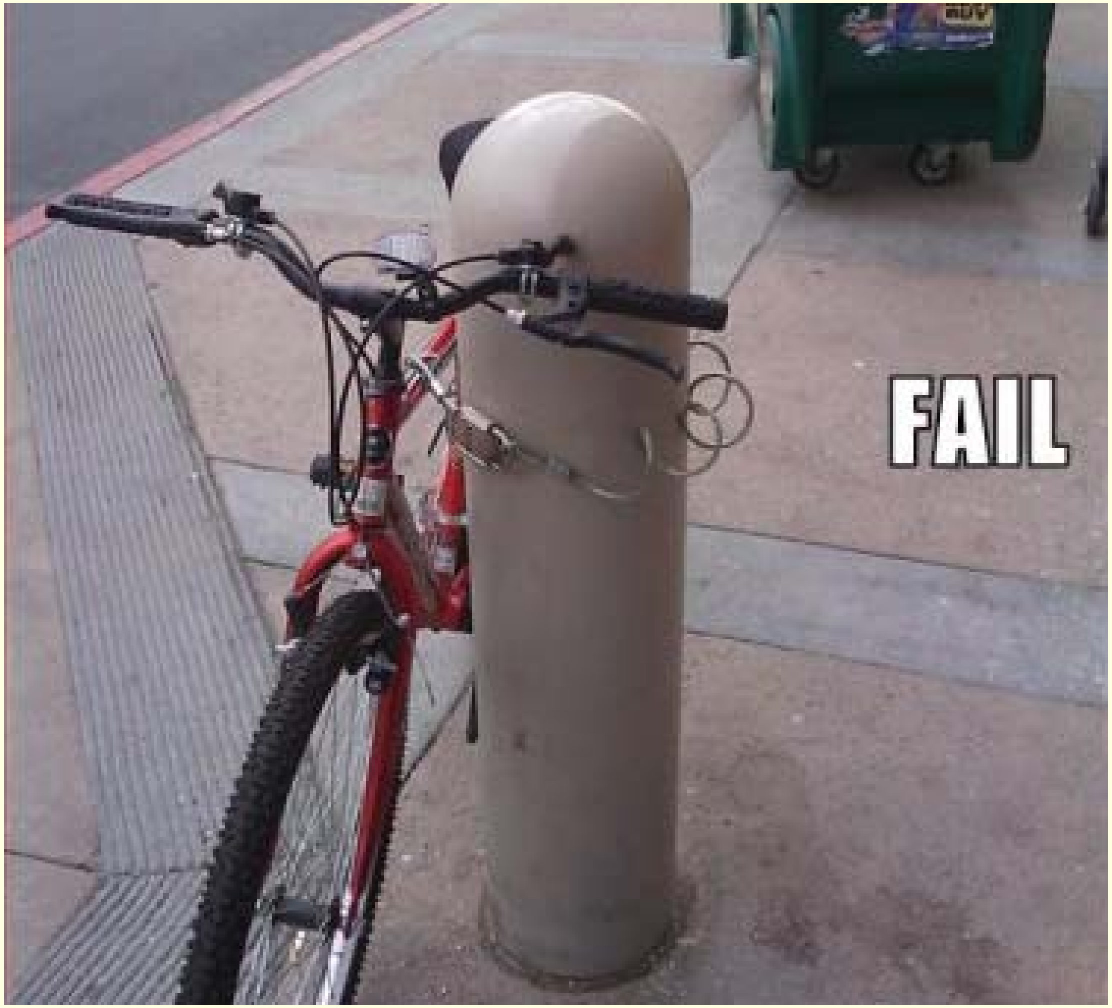
1) We **replace n with n-1** on both sides of the equation
2) We plug the result back in
3) And then we do it again

   and again and again and again…

   till a “light goes off” and we see something

Recurrence Relations

Or you're like this guy, whose lights never turned on.



Recurrence Relations

- Back to Factorial N
  - T(n) = T(n-1) + 2 ----- call this Eq. 1
    - Replace n with n-1

<u>DON'T overcomplicate this step.</u>

It is REALLY this SIMPLE.

Wherever you see an n in Eq. 1, simply replace with n-1.

So if you have T(n-1) and you replace that n with an n-1, you will get T((n-1)-1), which equates to T(n-2).

Simple right?

Right.

Recurrence Relations

- Back to Factorial N
  - T(n) = T(n-1) + 2 ----- call this Eq. 1
    - Replace n with n-1
    - T(n-1) = T(n-2) + 2 ----- call this Eq. 2
  - Now substitute the result of Eq. 2 into Eq. 1
    - T(n) = T(n-2) + 2 + 2

      Wait? How'd we get this?

      T(n) = T(n-1) + 2 ----- Eq. 1

      And from Eq. 2, we also have, T(n-1) = T(n-2) + 2

      So we simply plug in the result (the right side) of the Eq. 2 into Eq. 1 where we see T(n-1)

      T(n) = T(n-1) + 2

      T(n) = (T(n-2) + 2) + 2 removing parantheses, we get

      T(n) = T(n-2) + 2 + 2

Recurrence Relations

- Back to Factorial N
  - T(n) = T(n-1) + 2 ----- call this Eq. 1
    - Replace n with n-1
    - T(n-1) = T(n-2) + 2 ----- call this Eq. 2
  - Now substitute the result of Eq. 2 into Eq. 1
    - T(n) = T(n-2) + 2 + 2
      - We can look at 2 + 2 as 2*2 ….you'll see why we do this shortly
    - T(n) = T(n-2) + 2* 2 ----- call this Eq. 3
  - So what did we do:
    - We made ANOTHER equation for T(n)
    - **<u>But this one is in terms of T(n-2)</u>**
    - REDUCED from being in terms of T(n-1)

Recurrence Relations

- Back to Factorial N
  - So we now have this new equation for T(n):
    - T(n) = T(n-2) + 2*2
  - Are we done?
    - NO! Cuz we still have T(…)s on the right
  - And do we know how many operations are performed by T(n-2)?
    - Perhaps 5,219 operations? We don't know!
  - So we now need to REDUCE this equation further
  - We have T(n) in terms of T(n-2)
  - We want to get T(n) in terms of T(n-3)

Recurrence Relations

- Back to Factorial N
  - So we now need to REDUCE this equation further
  - We want to get T(n) in terms of T(n-3)
  - How are we going to do this?
    - We currently have T(n) = T(n-2) + 2*2
    - We want to develop an equation with T(n-2) on the **<u>left</u>**
    - and <u>in terms of T(n-3)</u>
  - So, in Eq. 2, once again, replace n with n-1
    - T(n-1) = T(n-2) + 2 ----- Eq. 2
    - <u>Replace n with n-1</u>
    - T(n-2) = T(n-3) + 2 ----- call this Eq. 4
  - Ah! So we now have our "T(n-2)" equation

Recurrence Relations

- Back to Factorial N
  - Now substitute the result of Eq. 4 into Eq. 3
    - T(n-2) = T(n-3) + 2 ----- Eq. 4
    - T(n) = T(n-2) + 2* 2 ----- Eq. 3
    - T(n) = T(n-3) + 2 + 2*2
      - 2 + 2*2 really is 2*3 …again, you'll see why we do this in a bit
    - T(n) = T(n-3) + 2*3
  - Again, what did we accomplish?
    - We made ANOTHER equation for T(n)
    - **<u>But this one is in terms of T(n-3)</u>**
    - REDUCED from being in terms of T(n-2)

Recurrence Relations

- Back to Factorial N
  - Thus far, we have three equations with T(n) on the left side
    - T(n) = T(n-1) + 2*1
      - Note that I added the *1 next to the 2
      - This doesn't change anything right?
      - 2*1 is the same as just plain 'ole 2
      - You'll see why we did this in a second.
    - T(n) = T(n-2) + 2*2
    - T(n) = T(n-3) + 2*3

Recurrence Relations

- Back to Factorial N
  - Is there a pattern developing? Perhaps some "light" going off?
    - 1st step of recursion, we have: $T(n) = T(n-1) + 2*1$
    - 2nd step of recursion, we have: $T(n) = T(n-2) + 2*2$
    - 3rd step of recursion, we have: $T(n) = T(n-3) + 2*3$
  - If we followed the process one more time, we get
    - $T(n) = T(n-4) + 2*4$ …for the 4th step of the recursion
  - So on the **kth step/stage of the recursion**, we get a **generalized recurrence relation**:
    - $T(n) = T(n-k) + 2*k$

Recurrence Relations

- Back to Factorial N
  - So on the **kth step/stage of the recursion**, we get a **generalized recurrence relation**:
    - T(n) = T(n-k) + 2*k
  - WHEW!
    - That was a lot!
    - But we're finally done! Right.?.
    - WRONG!!! Why aren't we done yet?
    - CUZ we still have T(…)s on the right side of the equation
  - **So now we need to actually solve this generalized recurrence relation**

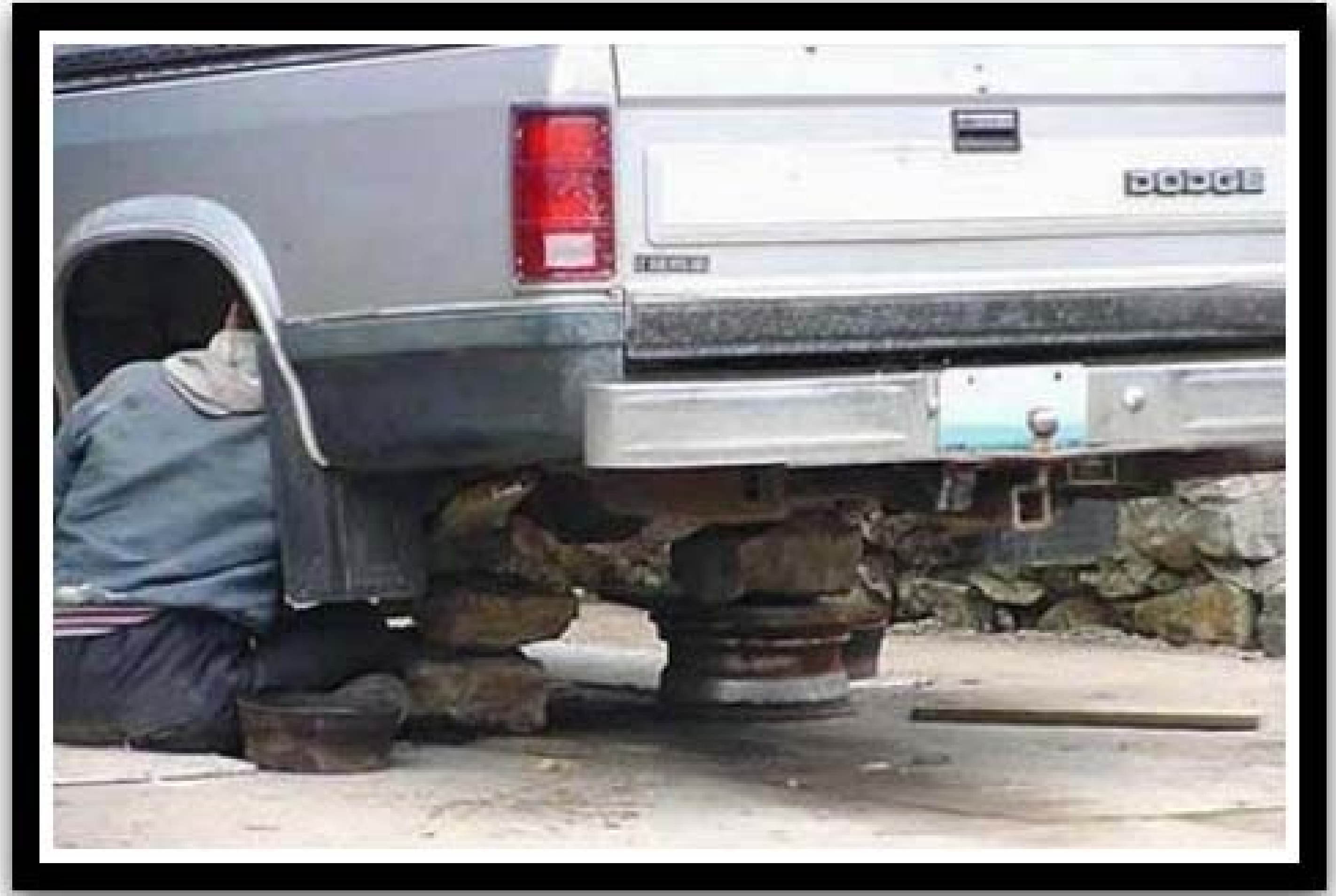
Recurrence Relations

- Back to Factorial N
  - We need to solve this generalized rec. relation
    - $T(n) = T(n-k) + 2*k$
  - How?
    - Remember we said we wanted to reduce the right side of the equation to T(1)
    - Again, why?
      - Because we know what T(1) equals…it equals 1!
    - So we have T(n-k) and we want T(1)
    - What can we do?
      - Simple! Let $n - k = 1$
      - Solve for k leaving $k = n – 1$ (then plug back into equation)

Recurrence Relations

- Back to Factorial N
  - We need to solve this generalized rec. relation
    - $T(n) = T(n-k) + 2*k$
    - $k = n – 1$
      - Plug into above equation
    - $T(n) = T(n-(n-1)) + 2(n-1) \ = T(1) + 2(n-1)$
      - And we know that $T(1) = 1$
    - Therefore….
    - $T(n) = 2(n-1) + 1 = 2n – 1$
    - And we are done!
  - Right side does not have any T(…)'s
  - This rec. relation is now solved!
  - This algorithm runs in O(n), or LINEAR time.

Brief Interlude: Human Stupidity



Recurrence Relations

- Let's look at a function that calculates powers

```
int power (int x, int n) {      // calculates the value of x^n
     if (n == 0)
          return 1;
     if (n == 1)
          return x;
     if (n is even)
          return power(x*x, n/2);
     else // if n is odd
          return power(x*x, n/2)*x;
}
```

- What's going on in this problem?
  - At every step, the problem size is reduced by **half**
  - If n is even, 2 arithmetic operations are computed
  - If n is odd, 3 arithmetic operations are computed

Recurrence Relations

- Power Function
  - What's going on in this problem?
    - At every step, the problem size is reduced by **half**
    - If n is even, 2 arithmetic operations are computed
    - If n is odd, 3 arithmetic operations are computed
  - When computing time complexity, **we assume the worst case**
    - We assume n is odd at each step
      - So 3 operations are assumed to be always needed
  - Thus, T(n) can be expressed as the sum of T(n/2) and the 3 operations needed at each step

    $T(n) = T(n/2) + 3$

    $T(1) = 1$

Recurrence Relations

- Power Function
  - So here's our recurrence relation:

    $T(n) = T(n/2) + 3$

    $T(1) = 1$
  - We need to solve this by removing all T(…)'s from the right side.
    - T(n/2) needs to hit the road
  - Then the problem is in its "closed form" and is solved.

Recurrence Relations

- Power Function
  - We need to solve T(n) in terms of n
  - Starting from this equation

    $T(n) = T(n/2) + 3$

    We reduce the right side until we get to T(1).
  - Why?
    - T(1) is known to us (it equals 1)
  - We do this in steps
    - We replace n with **n/2** on both sides of the equation
    - We plug the result back in
    - And then we do it again…till a "light goes off" and we see something

Recurrence Relations

- Power Function
  - This time we'll do a slightly different order of things…just so you see two different ways
    - Start with the base recurrence relation
    - $T(n) = T(n/2) + 3$ ----- call this Eq. 1
    - Replace n with n/2, and go ahead and do this several times
    - $T(n/2) = T(n/4) + 3$ ----- call this Eq. 2
    - $T(n/4) = T(n/8) + 3$ ----- call this Eq. 3
    - $T(n/8) = T(n/16) + 3$ ----- call this Eq. 4
  - Now we substitute each one of these back into Eq.1 and hopefully see a pattern

Recurrence Relations

- Power Function
  - Here's the four current equations we have:
    - $T(n) = T(n/2) + 3$ ----- Eq. 1
    - $T(n/2) = T(n/4) + 3$ ----- Eq. 2
    - $T(n/4) = T(n/8) + 3$ ----- Eq. 3
    - $T(n/8) = T(n/16) + 3$ ----- Eq. 4
  - Now substitute the result of Eq. 2 into Eq. 1
    - $T(n) = T(n/4) + 3 + 3$
      - We can look at 3 + 3 as 3*2 ….you remember why…right.?.
    - $T(n) = T(n/4) + 3*2$ ----- call this Eq. 5

Recurrence Relations

- Power Function
  - Here's the four current equations we have:
    - $T(n) = T(n/2) + 3$ ----- Eq. 1
    - $T(n/2) = T(n/4) + 3$ ----- Eq. 2
    - $T(n/4) = T(n/8) + 3$ ----- Eq. 3
    - $T(n/8) = T(n/16) + 3$ ----- Eq. 4
  - Now substitute the result of Eq. 3 into Eq. 5
    - $T(n) = T(n/8) + 3 + 3*2$
    - $T(n) = T(n/8) + 3*3$ ----- call this Eq. 6
  - One more substitution of Eq. 4 into Eq. 6:
    - $T(n) = T(n/16) + 3*4$ ----- call this Eq. 7

Recurrence Relations

- Power Function
  - Now show all the equations we developed with T(n) on the left…is there a pattern developing?
    - $T(n) = T(n/2) + 3*1 \qquad = T(n/2^1) + 3*1$
    - $T(n) = T(n/4) + 3*2 \qquad = T(n/2^2) + 3*2$
    - $T(n) = T(n/8) + 3*3 \qquad = T(n/2^3) + 3*3$
    - $T(n) = T(n/16) + 3*4 \qquad = T(n/2^4) + 3*4$
  - So on the kth step/stage of the recursion, we get a generalized recurrence relation:
    - $T(n) = T(n/2^k) + 3*k$
  - We're not done yet right.
  - Cuz we need to get rid of the $T(n/2^k)$

Recurrence Relations

- Power Function
  - We need to solve this generalized rec. relation
    - $T(n) = T(n/2^k) + 3*k$
  - How?
    - Remember we said we wanted to reduce the right side of the equation to T(1)
    - Again, why?
      - Because we know what T(1) equals…it equals 1!
    - So we have $T(n/2^k)$ and we want T(1)
    - Simple! Let $n = 2^k$
    - Solve for k
      - Take log base 2 of both sides
      - $k = \log n$

Plug back into equation

Recurrence Relations

- Power Function
  - We need to solve this generalized rec. relation
    - $T(n) = T(n/2^k) + 3*k$
    - So $n = 2^k$ and $k = \log n$
      - Plug into above equation
    - $T(n) = T(1) + 3(\log n)$
      - And we know that $T(1) = 1$
    - Therefore….
    - $T(n) = 1 + 3\log(n)$
    - And we are done! This algorithm runs in logarithmic time.
  - Right side does not have any T(…)'s
  - This rec. relation is now solved!

Recurrence Relations

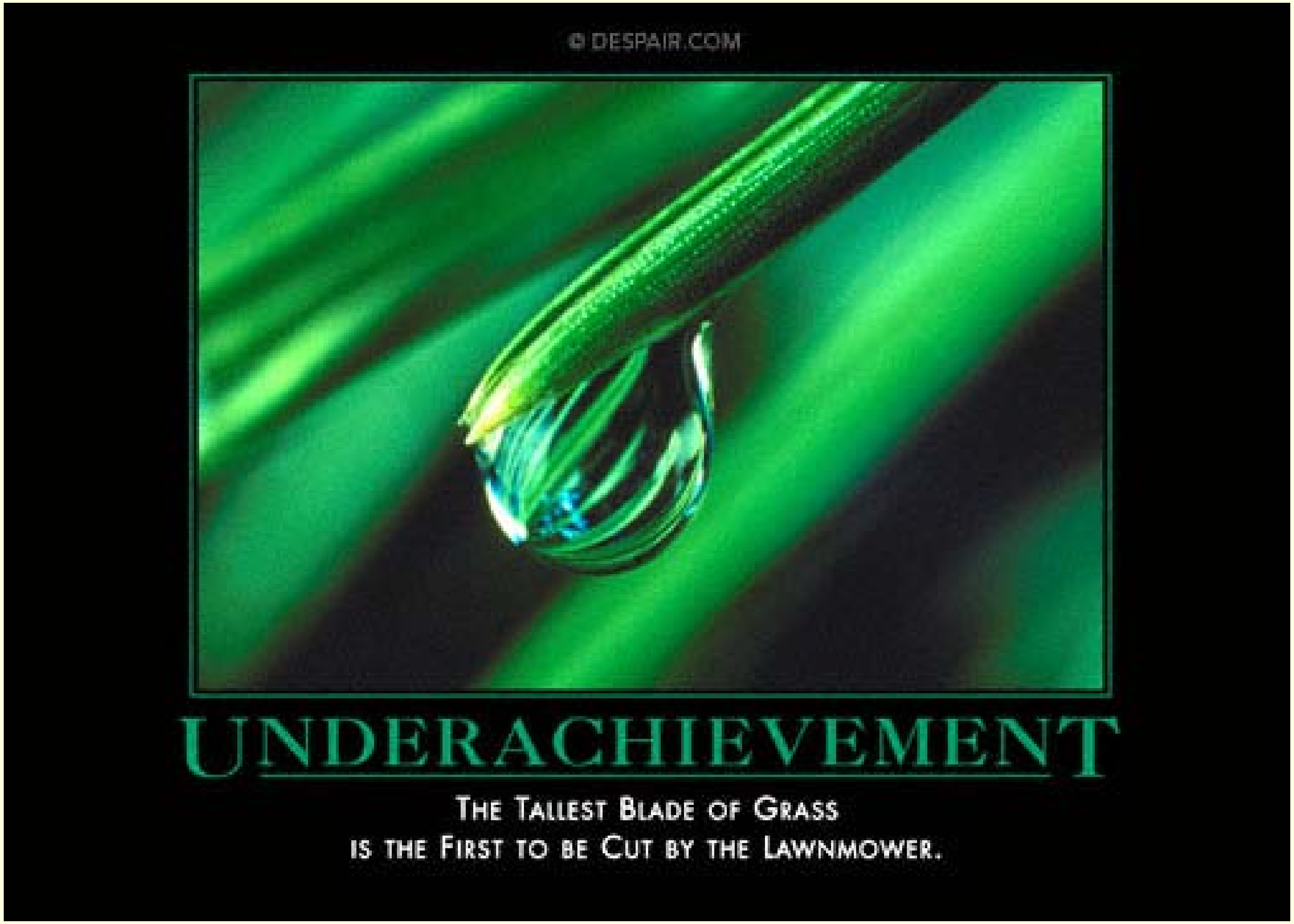
WASN'T

THAT

(Let's admit it:

that sucked!)

Daily Demotivator



Recurrence Relations

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I