



RECURRENCE RELATIONS

COP 3502

Recurrence Relation

- In mathematics, a recurrence relation is an equation that recursively defines a sequence.
 - For example, a mathematical recurrence relation for the Fibonacci Numbers is:
 - $F_n = F_{n-1} + F_{n-2}$
 - With base cases:
 - $F_2 = 1$
 - $F_1 = 1$
 - With that we can determine the 5th Fibonacci number:
 - $F_5 = F_4 + F_3 = \underline{3 + 2} = 5$
 - $F_4 = F_3 + F_2 = \underline{2 + 1} = 3$
 - $F_3 = F_2 + F_1 = \underline{1 + 1} = 2$



Recurrence Relations

- What we are going to use Recurrence Relations for in this class is to solve for the run-time of a recursive algorithm.
 - Notice we haven't looked at the run-time of any recursive algorithms yet,
 - We have only analyzed iterative algorithms,
 - Where we can either approximate the runtime just by looking at it,
 - or by using summations as a tool to solve for the run-time.
 - Recurrence relations will be the mathematical tool that allows us to analyze recursive algorithms.



Recursion Review

- What is Recursion?
 - A problem-solving strategy that solves large problems by reducing them to smaller problems of the same form.



Recursion Review

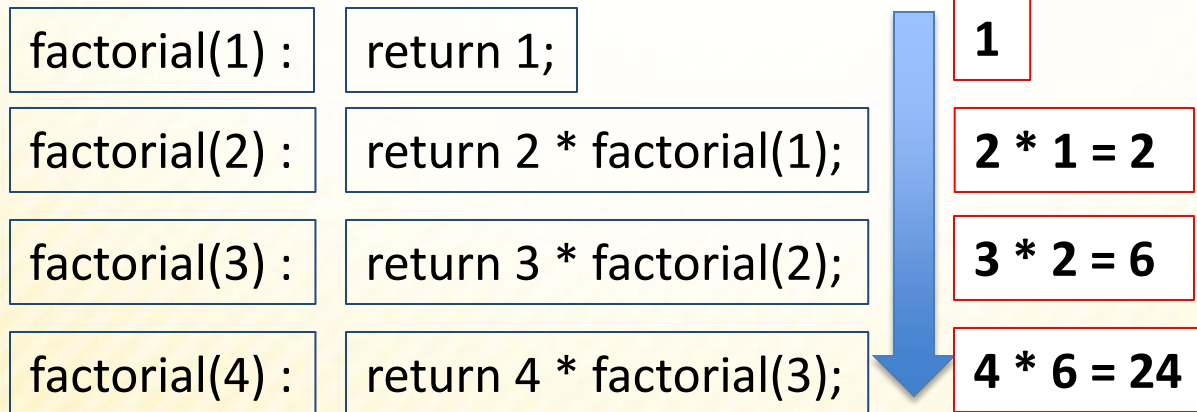
- An example is the recursive algorithm for finding the factorial of an input number n .
 - Where $4!$
 - $= 4 * 3 * 2 * 1 = 24$
 - Note that each factorial is related to the factorial of the next smaller integer:
 - $n! = n * (n-1)!$
 - So, $4! = 4 * (3-1)! = 4 * 3!$
 - We stop at $1! = 1$
 - In mathematics, we would define:
 - $n! = n * (n-1)!$ if $n > 1$
 - $n! = 1$ if $n = 1$



Recursion Review

- The recursive algorithm for finding the factorial of an input number n .
 - Where $4!$
 - $= 4 * 3 * 2 * 1 = 24$

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



Stack



Recurrence Relations

- Let's determine the run-time of factorial,
 - Using Recurrence Relations
- We can see that the total number of operations to execute factorial for input size n
 - 1) The sum of the 2 operations (the '*' and the '-')
 - 2) Plus the number of operations needed to execute the function for $n-1$.
- OR if it's the base case just one operation to return.

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



Recurrence Relations

- We will define $T(n)$ as the number of operations executed in the algorithm for input size n .
 - So $T(n)$ can be expressed as the sum of:
 - $T(n-1)$
 - plus the 2 arithmetic operations
 - This gives us the following Recurrence Relation:
 - $T(n) = T(n-1) + 2$
 - $T(1) = 1$

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



Recurrence Relations

- So we've come up with a Recurrence Relation, that defines the number of operations in factorial:
 - $T(n) = T(n-1) + 2$
 - $T(1) = 1$
- BUT this isn't in terms of n , it's in terms of $T(n-1)$,
 - So what we want to do is remove all of the $T(\dots)$'s from the right side of the equation.
 - This will give us the **"closed form"** and we will have solved for the number of operations in terms of n
 - AND THEN, we can determine the Big-O Run-Time!

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



Recurrence Relations

- Solve for the closed form solution of:
 - $T(n) = T(n-1) + 2$
 - $T(1) = 1$
- We are going to use the iteration technique.
 - First, we will recursively solve $T(n-1)$ and plug that back into the equation,
 - And we will continue doing this until we see a pattern.
 - *Iterating*, which is why this is called the iteration technique.

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



- Use the iteration technique to solve for the closed form solution of (Solved in class):

➤ $T(n) = T(n-1) + 2$ $T(1) = 1$



- Use the iteration technique to solve for the closed form solution of (Solved in class):

➤ $T(n) = T(n-1) + 2$ $T(1) = 1$



Towers of Hanoi

- If we look at the Towers of Hanoi recursive algorithm,
 - we can come up with the following recurrence relation for the # of operations:
 - (where again $T(n)$ is the number operations for an input size of n)
 - $T(n) = T(n-1) + 1 + T(n-1)$ and $T(1) = 1$
 - Simplifying: $T(n) = 2T(n-1) + 1$ and $T(1) = 1$

```
void doHanoi(int n, char start, char finish, char temp) {
    if (n==1) {
        printf("Move Disk from %c to %c\n", start,
finish);
    }
    else {
        doHanoi(n-1, start, temp, finish);
        printf("Move Disk from %c to %c\n", start finish);
        doHanoi(n-1, temp, finish, start);
    }
}
```

- Use the iteration technique to solve for the closed form solution of (Solved in class):

➤ $T(n) = 2T(n-1) + 1$ and $T(1) = 1$



- Use the iteration technique to solve for the closed form solution of (Solved in class):

➤ $T(n) = 2T(n-1) + 1$ and $T(1) = 1$

