UCF

# 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:
      - $F_5 = F_4 + F_3$ $= 3 + 2 = 5$
      - $F_4 = F_3 + F_2$ $= 2 + 1 = 3$
      - $F_3 = F_2 + F_1$ $= 1 + 1 = 2$

# 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);
}
```

| | | |
|---|---|---|
| factorial(1) : | return 1; | **1** |
| factorial(2) : | return 2 * factorial(1); | **2 * 1 = 2** |
| factorial(3) : | return 3 * factorial(2); | **3 * 2 = 6** |
| factorial(4) : | return 4 * factorial(3); | **4 * 6 = 24** |

**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(…)'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.
    - I***terating***, 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):**
  - $T(n) = T(n-1) + 2 \qquad\qquad T(1) = 1$

- **Use the iteration technique to solve for the closed form solution of (Solved in class):**
  - $T(n) = T(n-1) + 2 \qquad\qquad T(1) = 1$

# 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 $T(1) = 1$

# Recursive Binary Search

- If we look at the Binary Search 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)**
  - $T(n) = T(n/2) + 1$ and $T(1) = 1$

```
int binsearch(int *values, int low, int high, int val) {
    int mid;
    if (low <= high){
       mid = (low+high)/2;
       if (val == values[mid])
           return 1;
       else if (val > values[mid])
           return binsearch(values, mid+1, high, val)
       else
           return binsearch(values, low, mid-1, val);
    }
    return 0;
}
```

- **Use the iteration technique to solve for the closed form solution of (Solved in class):**
  - **$T(n) = T(n/2) + 1$ and $T(1) = 1$**

- **Use the iteration technique to solve for the closed form solution of (Solved in class):**
  - **$T(n) = T(n/2) + 1$ and $T(1) = 1$**

# Exponentiation

- If we look at the Power recursive algorithm,
  - we can come up with the following recurrence relation for the # of operations:
    - **(where *T(exp)* is the number operations for an input size of *exp*)**
  - **T(exp) = T(exp - 1) + 1          and          T(1) = 1**

```
int Power(int base, int exp) {

      if (exp == 1)
          return base;
      else
          return (base*Power(base, exp – 1);
}
```

- **Use the iteration technique to solve for the closed form solution of (Solved in class):**
  - **T(exp) = T(exp - 1) + 1          and      T(1) = 1**

- **Use the iteration technique to solve for the closed form solution of (Solved in class):**
  - **T(exp) = T(exp - 1) + 1 and T(1) = 1**

# Fast Exponentiation

- If we look at the Fast Exponentiation recursive algorithm,
  - How do we come up with a recurrence relation for the # of operations?
    - **(where *T(exp)* is the number operations for an input size of *exp*)**
  - This one is a little more difficult because we do something different if exp is even, or exp is odd.

```
int PowerNew(int base, int exp) {
      if (exp == 0)
            return 1;
      else if (exp == 1)
            return base;
      else if (exp%2 == 0)
            return PowerNew(base*base, exp/2);
      else
            return base*PowerNew(base, exp-1);
}
```

# Fast Exponentiation

If we look at the Fast Exponentiation recursive algorithm,

- When exp is even we have:
  - T(exp) = T(exp/2) + 1
- When exp is odd
  - T(exp) = T(exp – 1) + 1

*And this step changes exp to be even!*

**So roughly speaking we have this:**

*T(exp) <= T(exp/2) + 2*

```
int PowerNew(int base, int exp) {
      if (exp == 0)
            return 1;
      else if (exp == 1)
            return base;
      else if (exp%2 == 0)
            return PowerNew(base*base, exp/2);
      else
            return base*PowerNew(base, exp-1);
}
```

- **Use the iteration technique to solve for the closed form solution of**
  - **T(exp) <= T(exp/2) + 2**
  - Hopefully we notice that this almost identical to the binary search recurrence relation:
    - **T(n) = T(n/2) + 1 (Except we would have an extra +1 at the end)**
  - So we would end up with:
    - **$T(n) = \log_2 n + 2$**
    - **O(log n)**
  - So if exp = $10^{20}$, we would do on the order of lg $10^{20}$ operations which is around 66.
  - As opposed to 100 billion billion operations.

# Pitfalls of Big-O Notation

1) Not useful for small input sizes
   - Because the constants and smaller terms will matter.
2) Omission of the constants can be misleading
   - For example, **2N log N** and **1000 N**
     - Even though its growth rate is larger, the 1st function is probably better. Because the 1000 constant could be memory accesses or disk accesses.
3) Assumes an infinite amount of memory
   - Not trivial when using large data sets.
4) Accurate analysis relies on clever observations to optimize the algorithm.

# Master Theorem

- There is a general plug n chug formula for recurrence relations as well
  - Good for checking your answers after using the iterative method (since you'll have to use the iterative method on the exam)
  - **If $T(n) = AT(n/B) + O(n^k)$, where $A, B, k$ are constants:**
  - **Then $T(n) =$**

$$T(n) = \begin{cases} O(n^{\log_B A}) & \text{if } A > B^k \\ O(n^k \log n) & \text{if } A = B^k \\ O(n^k) & \text{if } A < B^k \end{cases}$$

Is the Big-O run-time.

# Master Theorem

- $T(n) = AT(n/B) + O(n^k)$, where $A, B, k$ are constants:

- $T(n) = \begin{cases} O(n^{\log_B A}) & \text{if } A > B^k \\ O(n^k \log n) & \text{if } A = B^k \\ O(n^k) & \text{if } A < B^k \end{cases}$

- **Some examples:**

| Recurrence Rel. | Case | Answer |
|---|---|---|
| $T(n) = 3T(n/2) + O(n^2)$ | | |
| $T(n) = 4T(n/2) + O(n^2)$ | | |
| $T(n) = 9T(n/2) + O(n^3)$ | | |
| $T(n) = 6T(n/3) + O(n^2)$ | | |
| $T(n) = 5T(n/5) + O(n)$ | | |