And More Recursion

Computer Science Department
University of Central Florida
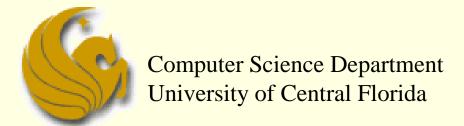
COP 3502 – Computer Science I

Announcements

- Contact method for instructor
  - Email: jcazalas@cs.ucf.edu
    - This is the email shown on the course website.
    - If you want a semi-quick reply, email this address
      - And NOT through Webcourses
- Next Quiz: possibly on Monday
- Questions on grading for program
  - Most likely the grade given is indeed accurate
  - Check the input and respective output files
  - And the solution, B4 asking "why this" or "why that"
- EXAM: START STUDYING NOW!!!!!!!!!!!!!!!!!!!!

Binary Search – **<u>A reminder</u>**

- ## Array Search
  - ### We are given the following **<u>sorted</u>** array:

index	0	1	2	3	4	5	6	7	8
value	2	6	19	27	33	37	38	41	118

  - ### We are searching for the value, 19 (for example)
  - ### Remember, we said that you search the middle element
    - If found, you are done
    - If the element in the middle is greater than 19
      - Search to the LEFT (cuz 19 MUST be to the left)
    - If the element in the middle is less than 19
      - Search to the RIGHT (cuz 19 MUST then be to the right)

Binary Search – <u>**A reminder**</u>

- **Array Search**
  - We are given the following <u>**sorted**</u> array:

index	0	1	2	3	4	5	6	7	8
value	2	6	19	27	33	37	38	41	118

  - We are searching for the value, 19
  - So, we MUST start the search in the middle INDEX of the array.
  - In this case:
    - The lowest index is 0
    - The highest index is 8
    - So the middle index is 4

Binary Search

- **Array Search**
  - Correct Strategy
    - We would ask, "is the number I am searching for, 19, greater or less than the number stored in index 4?
      - Index 4 stores 33
    - The answer would be "less than"
    - So we would modify our search range to in between index 0 and index 3
      - Note that index 4 is no longer in the search space
    - We then continue this process
      - The second index we'd look at is index 1, since (0+3)/2=1
      - Then we'd finally get to index 2, since (2+3)/2 = 2
      - And at index 2, we would find the value, 19, in the array

Binary Search

- Binary Search code:

```
int binsearch(int a[], int len, int value) {

        int low = 0, high = len-1;
        while (low <= high) {
                int mid = (low+high)/2;
                if (value < a[mid])
                        high = mid-1;
                else if (value > a[mid])
                        low = mid+1;
                else
                        return 1;
        }

        return 0;
}
```

Binary Search

- Binary Search code:
  - At the end of each array iteration, all we do is update either low or high
  - This modifies our search region
    - Essentially halving it
  - As we saw previously, this runs in **log n** time

  - But this iterative code isn't the easiest to read
  - We now look at the recursive code
    - MUCH easier to read and understand

Binary Search – Recursive

- Binary Search using recursion:
  - We need a stopping case:
    - We need to STOP the recursion at some point

  - So when do we stop:
  1) When the number is found!
  2) Or when the search range is nothing
     - huh?
     - The <u>search range is empty</u> when `(low > high)`

  - So how let us look at the code…

Binary Search – Recursive

- Binary Search Code (using recursion):
  - We see how this code follows from the explanation of binary search quite easily

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

Binary Search – Recursive

- Binary Search Code (using recursion):
  - So if the value is found
    - We `return 1`
  - Otherwise,
    - `if (searchval < values[mid])`
      - Then recursively call binSearch to the LEFT
    - `else if (searchval > values[mid])`
      - Then recursively call binSearch to the RIGHT
  - If `low` ever becomes greater than `high`
    - This means that `searchval` is NOT in the array

Brief Interlude: Human Stupidity

Recursive Exponentiation

- Example from Previous lecture
  - Our function:
    - Calculates b^e
      - Some base raised to a power, e
      - The input is the base, b, and the exponent, e
      - So if the input was 2 for the base and 4 for the exponent
        - The answer would be $2^4 = 16$
  - How do we do this recursively?
    - We need to solve this in such a way that part of the solution is a sub-problem of the EXACT same nature of the original problem.

Recursive Exponentiation

- Example from Previous lecture
    - Our function:
        - Using b and e as input, here is our function
            - $f(b,e) = b^e$
        - So to make this recursive, can we say:
            - $f(b,e) = b^e = b*b^{(e-1)}$
        - Does that "look" recursive?
        - YES it does!
        - Why?
        - Cuz the right side is indeed a sub-problem of the original
        - We want to evaluate b^e
        - And our right side evaluates $b^{(e-1)}$

Recursive Exponentiation

- **Example from Previous lecture**
  - Our function:
    - $f(b,e) = b*b^{(e-1)}$
    - So we need to determine the terminating condition!
    - We know that $f(b,0) = b^0 = 1$
      - So our terminating condition can be when e = 1
    - Additionally, our recursive calls need to return an expression for f(b,e) in terms of f(b,k)
      - for some k < e
    - We just found that $f(b,e) = b*b^{(e-1)}$
    - So now we can write our actual function…

Recursive Exponentiation

- Example from Previous lecture
  - Code:

```
// Pre-conditions: e is greater than or equal to 0.
// Post-conditions: returns b^e.
int Power(int base, int exponent) {

        if ( exponent == 0 )
                return 1;
        else
                return (base*Power(base, exponent-1));
}
```

Recursive Exponentiation

- Example from Previous lecture
  - Say we initially call the function with 2 as our base and 8 as the exponent
  - The final return will be
    - return 2*2*2*2*2*2*2*2
    - Which equals 256
  - You notice we have 7 multiplications (exp was 8)
  - The number of multiplications needed is <u>one less than the exponent value</u>
  - So if `n` was the exponent
    - The # of multiplications needed would be `n-1`

Fast Exponentiation

- **Example from Previous lecture**
  - This works just fine
  - BUT, it becomes VERY slow for large exponents
    - If the exponent was 10,000, that would be 9,999 mults!
  - How can we do better?

  - One key idea:
    - Remembering the laws of exponents
      - Yeah, algebra…the thing you forgot about two years ago
    - So using the laws of exponents
      - We remember that $2^8 = 2^4 * 2^4$

Fast Exponentiation

■ **Example from Previous lecture**

■ One key idea:

■ Remembering the laws of exponents

■ $2^8 = 2^4 * 2^4$

■ Now, if we know 2^4

■ we can calculate 2^8 with one multiplication

■ What is 2^4?

■ $2^4 = 2^2 * 2^2$

■ and $2^2 = 2*(2)$

■ So… $2*(2) = 4$, $4*(4) = 16$, $16*(16) = 256 = 2^8$

■ So we've calculated 2^8 using on three multiplications

■ MUCH better than 7 multiplications

Fast Exponentiation

- **Example of Fast Exponentiation**
  - So, in general, we can say:
  - $b^n = b^{n/2} * b^{n/2}$
  - So to find b^n, we find $b^{n/2}$
    - HALF of the original amount
  - And to find $b^{n/2}$, we find $b^{n/4}$
    - Again, HALF of $b^{n/2}$
  - This smells like a `log n` running time
    - log `n` number of multiplications
    - Much better than `n` multiplications
  - But as of now, this only works if `n` is even

Fast Exponentiation

- Example of Fast Exponentiation
    - So, in general, we can say:
    - $b^n = b^{n/2}*b^{n/2}$
    - This works when n is even
    - But what if n is odd?
    - Notice that $2^9 = 2^4*2^4*2$
    - So, in general, we can say:

$$a^n = \begin{cases} a^{n/2}(a^{n/2}) & \text{if n is even} \\ a^{n/2}(a^{n/2})(a) & \text{if n is odd} \end{cases}$$

Fast Exponentiation

■ Example of Fast Exponentiation

  ■ Also, this method relies on "integer division"

    ■ We've briefly discussed this

    ■ Basically if n is 9, then n/2 = 4

      ■ Integer division

      ■ Think of it as dividing

      ■ AND then rounding down, if needed, to the nearest integer

    ■ So if n is 121, then n/2 = 60

    ■ Finally, if n is 57, then n/2 = 28

  ■ Using the same base case as the previous power function, here is the code…

Fast Exponentiation

■ Example of Fast Exponentiation

■ Code:

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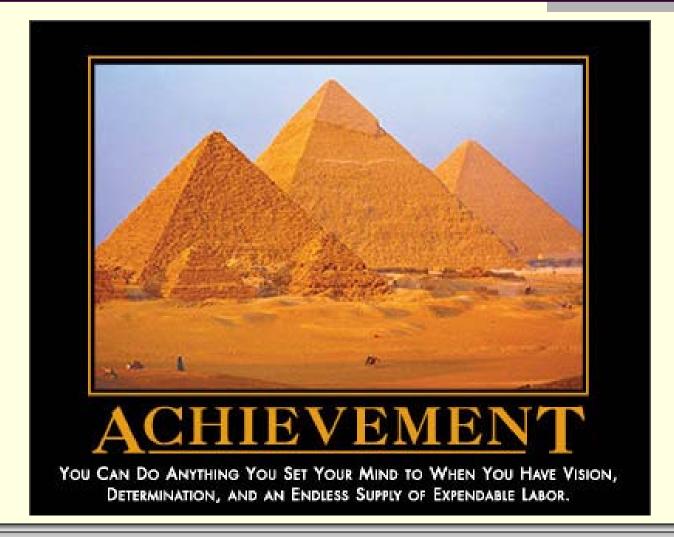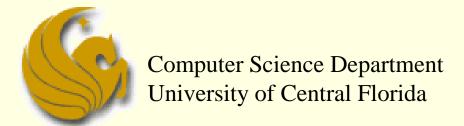
Recursion

WASN'T THAT BODACIOUS!

Daily Demotivator

ACHIEVEMENT

YOU CAN DO ANYTHING YOU SET YOUR MIND TO WHEN YOU HAVE VISION, DETERMINATION, AND AN ENDLESS SUPPLY OF EXPENDABLE LABOR.

And More Recursion

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I