CSI33 Data Structures

Sharon Persinger

Fall 2019

Day 11 October 2

General problem solving method: Divide-and-conquer

- Using Divide and conquer, you solve a problem by
  - Breaking it into some smaller problems.
  - Solving each of the smaller problems.
  - Reassembling the solutions of the smaller problems into a solution to the big problem.

Recursion, a specific type of divide-and-counquer

- n factorial or $n! = n(n-1)(n-2) \cdots (2)(1$
- A formal recursive definition of $n! = fact(n)$
  - If $n = 0$, $fact(n) = 1$
  - Else $fact(n) = n \cdot fact(n-1)$
- This is a recursive definition. The function is used in the definition of the function.
- How is is not a circular definition?
- Each successive function call has a smaller number as the argument to the function. Eventually we will get to $fact(0) = 1$

Write a recursive implementation of fact(n).

- Easy!

Defining functions recursively

- A properly defined recursive function must have one or more base cases where the function can be computed without a recursive call to the function.
- Every chain of recursive calls must eventually terminate at a base case.

Find all the anagrams (permutations) of a string of letters.

- Think recursively!
- To find all the permutations of the letters in ‘cat’, you could
  - First find all the permutations of the letters in ‘at’
  - Then insert the letter ‘c’ into every possible position in each of those strings.

Write a recursive definition of an anagrams function

- Think recursively!
- Specification:

def anagrams(word):
“pre: word is a string, possibly empty”
“post: return value is the list of all permutations of the characters in the string word”

- What to use as the base case?

Binary search is a divide-and-conquer algorithm.

- Compare the earlier version to a recursive implementation
- Chapter 1 bsearch
- Chapter 6 bsearch

Exponentiation

- Usual version of exponentiation uses a counting loop
- Write a function loopPower(a, n) that uses a counting loop to raise the base a to the integer power n.
- Easy, but time is $\Theta(n)$

Faster version of exponentiation using recursion

- Compute 2^16 = (2^8)^2
- 2^8 = (2^4)^2   and 2^4 = (2^2)^2
- So 2^2 = 4, 4^2 = 16, 16^2 = 256, 256^2 = 56,536
- 4 operations of repeated squaring, not 15 multiplications as with the loop method

Recursive version of exponentiation

- a^n =
  - a^(n//2) · a^(n//2) if n is even
  - a^(n//2) · a^(n//2) · a if n is odd
- Base case a^0 = 1
-
- Time analysis?

Fibonacci numbers Fib(n)

- Fib(1) = 1
- Fib(2) = 1
- Fib(n) = Fib(n-1) + Fib(n-2) for n > 2

- Calculate by hand
- Fib(3)
- Fib(4)
- Fib(6)

- Write a recursive function definition for Fib(n) and use it.
- Write a definition that uses a loop.
- Which is more efficient? Why?

Review the time analysis of recursive function examples

- Factorial
- Anagrams/permutations
- Binary search
- Exponentiation
- Fibonacci numbers

True/False assignment