CS213/CS213M DS&A
Sharat Chandran

www.cse.iitb.ac.in/~sharat/current/213m

(Many slides obtained from colleagues, and the Internet and gratefully acknowledged using the fair use copyright law)

Agenda

- Recursion (illustrated with examples)
  - Basic idea and basic principle
  - Parameterization
  - Computational complexity
  - Tail recursion
- The programmer's workbench
  - Some preliminary notes on Python

The Tower of Brahma

- The Tower of Hanoi puzzle
  - Three rods (A, B, & C) and disks of decreasing sizes that can slide on any rod assembled on A in a conical fashion
  - B and C are empty. Goal: Move disks to B
  - Only one disk can be moved at a time
  - No large disk may be placed on a smaller disk
- Programming: the number of disks = n
- Physical version presumably found in a temple in Kashi Vishwanath

Solution

- The most natural solution to this problem is to assume you have a solution S to a problem of size n-1
  - Bulk move applying S from A to C (using B)
  - Move the largest remaining disk from A to B
  - Bulk move applying S from C to B (using A)

Solution

- By the method of induction, the program is considered "proved" correct
- The number of moves M(n) satisfies the recurrence relation M(n) = 2M(n-1)+1 with M(1) = 1 as the base case
  - $M(n) = 2^n - 1$

```
void hanoi (int n,  char a, char b, char c){
  if (n == 1) cout << "move from " << a << " to " << b << endl;
  else {
    hanoi (n-1, a, c, b);
    cout << "move from " << a << " to " << b << endl;
    hanoi (n-1, c, b, a);
  }
}
```

Design Principle

1. Formulate the solution (to a problem of size n) in terms of the solution to the same problem of size less than n.
2. Determine a "base case" (at n = 0 or 1)
   - Solution is trivial and there is no recursive call
   - There should be at least one base case
   - All recursive call must eventually reach base case
3. Terminate recursion when this "base case" value is reached.

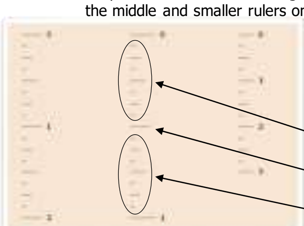
- Tip: While coding, do step 2 and 3 first and in a focused manner (very important)

Draw Ruler

- Print the ticks and numbers of a scale (or ruler)

Using Recursion

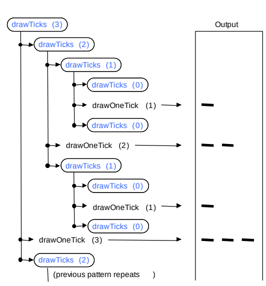
drawTicks (length)
Input: length of a 'tick'
Output: ruler with tick of the given length in the middle and smaller rulers on either side



```
drawTicks(length)
 if (length > 0 ) then
   drawTicks (length – 1 )
   draw tick of the given length
   drawTicks (length – 1 )
```


Recursive Drawing Method

- An interval with a central tick length L ≥1 consists of:
  - An interval with a central tick length L–1
  - An single tick of length L
  - An interval with a central tick length L–1



Ruler Code

```
def draw_line(tick_length, tick_label=''):
  """Draw one line with given tick length (followed by optional label)."""
  line = '-' * tick_length
  if tick_label:
    line += ' ' + tick_label
  print(line)

def draw_interval(center_length):
  """Draw tick interval based upon a central tick length."""
  if center_length > 0:                  # stop when length drops to 0
    draw_interval(center_length - 1)     # recursively draw top ticks
    draw_line(center_length)             # draw center tick
    draw_interval(center_length - 1)     # recursively draw bottom ticks

def draw_ruler(num_inches, major_length):
  """Draw English ruler with given number of inches, major tick length."""
  draw_line(major_length, '0')           # draw inch 0 line
  for j in range(1, 1 + num_inches):
    draw_interval(major_length - 1)      # draw interior ticks for inch
    draw_line(major_length, str(j))      # draw inch j line and label
```

Note the two recursive calls

Summary

- Recursion (illustrated with examples)
  - Basic idea and basic principle (correctness, complexity, and running)
  - Parameterization
  - More on computational complexity
  - Tail recursion

Recursion: Parameterization

- Recursion (illustrated with examples)
  - Basic idea and basic principle (correctness, complexity, and running)
  - Parameterization
  - More on computational complexity
  - Tail recursion

Parameterization

- In creating recursive methods, it is useful to define the methods in ways that facilitate recursion
  - This sometimes requires we define additional parameters that are passed to the method
- Example: Reverse items in an array (use recursion)
  - ReverseArray (*S*) may be **awkward** (see excerpt)

```
def aReverse(S):
    """Assume we know how to
reverse a problem of size len(S)
-1."""
    if len(S) == 1:
        return S
    else:
        smaller = S[1:len(S)]
        Q = aReverse(smaller)
        Q.append(S[0])
        return Q
```

Pseudocode Example

Algorithm ReverseArray(*A, i, j*):

Input: An array *A* and nonnegative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at index *i* and ending at *j*

if *i* < *j* **then**

Swap *A*[*i*] and *A*[*j*]

ReverseArray(*A, i* + 1, *j* - 1)

return

Demo

Binary Search

- Problem: Find a target 'k' in a sorted array of known dimensions 'n'
- As you probably know, the operation can be done in time proportional to log(n)
- The obvious parameterization is to have a recursion prototype like search (data, target)
  - But the problem is that in the body of the function we would to copy half the data into a new data structure
  - This would be a linear time operation defeating the log(n) objective

Binary Search

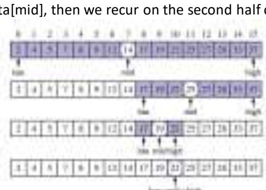
- Parameterized with two additional indices

```python
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                    # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:         # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid − 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Binary Search Example

- Searching for 22
- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Buggy Binary Search

- See this page for a 20 year bug

Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken

Friday, June 2, 2006

Posted by Joshua Bloch, Software Engineer

I remember vividly Jon Bentley's first Algorithms lecture at CMU, where he asked all of us incoming Ph.D. students to write a binary search, and then dissected one of our implementations in front of the class. Of course it was broken, as were most of our implementations. This made a real impression on me, as did the treatment of this material in his wonderful *Programming Pearls* (Addison-Wesley, 1986; Second Edition, 2000). The key lesson was to carefully consider the invariants in your programs.

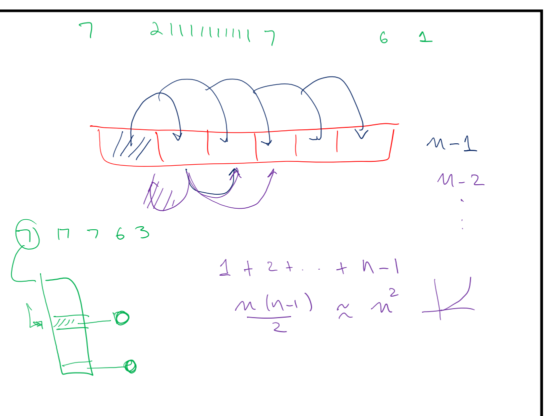
Fast forward to 2006. I was shocked to learn that the binary search program that Bentley proved correct and subsequently tested in Chapter 5 of *Programming Pearls* contains a bug. Once I tell you

Recursion: Computations

- Recursion (illustrated with examples)
  - Basic idea and basic principle (correctness, complexity, and running)
  - Parameterization
  - More on computational complexity
  - Tail recursion

Element Uniqueness

- Given an unsorted sequence S of n integers, is it a set? (Are elements unique)
- Straightforward algorithm that runs in time proportional to n^2



Element Uniqueness

- Given an unsorted sequence S of n integers, is it a set? (Are elements unique)
- Straightforward algorithm that runs in time proportional to n^2
  - We can also sort and do this asymptotically faster

Faster than $\approx n^2$

```
bool unique(vector<int> &S) {
  for (int i=0; i<S.size(); i++)
    for (int j=i+1; j<S.size(); j++)
      if (S[j] == S[i]) return false;

  return true;
}
```

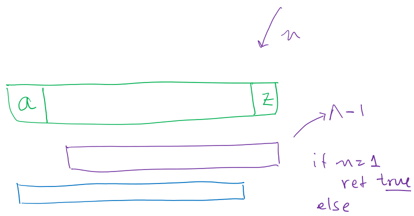
Advanced topics in red are mentioned out of interest and can be safely skipped

Element Uniqueness

- Given an unsorted sequence S of n integers, is it a set? (Are elements unique)
- Straightforward algorithm that runs in time proportional to n^2
- In a model of computation that allows only comparison, the problem cannot be solved in less than time proportional to n (log n) $\approx n^1$
  - Amazingly on a quantum model the number is $n^{2/3}$

Element Uniqueness

- If we try to write a recursive program (with no iterations inside), we can end up doing a terrible job (exponential complexity)



$a == z$

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

$\approx 2^n - 1$

Element Uniqueness

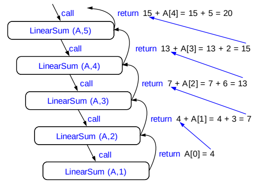
- If we try to write a recursive program (with no iterations inside), we can end up doing a terrible job (exponential complexity)

```python
def unique(S, start, stop):
    """Return True if there are no duplicate."""
    if stop - start <= 1:
        return True  # at most one item
    elif not unique(S, start, stop-1):
        return False  # first part
    elif not unique(S, start+1, stop):
        return False  # second part
    else:
        return S[start] != S[stop-1]#first and last differ?
```

Sum of items in an array

Algorithm LinearSum(*A, n*):
Input:
A integer array *A* and an integer *n* such that *A* has at least *n* elements
Output:
The sum of the first *n* integers in *A*
if *n* = 1 **then**
return *A*[0]
else
return LinearSum(*A, n* - 1) + *A*[*n* - 1]

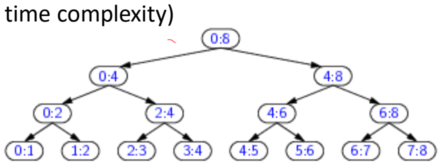
Example: LinearSum(S,5)
S = [4,3,6,2,8]



- Observe that we have to save the state and it grows in time proportional to n

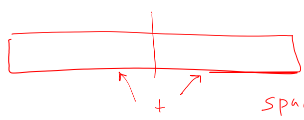
Linear Sum

- With two recursive calls instead of one, we can reduce the space complexity (but not time complexity)



Example: b_sum(0,8)

$T_{power}(n) = T_{power}(n/2) + 1$



$T(n) = 2T(\frac{n}{2}) + 1$

$\approx O(n)$

space complexity is $\approx O(\log n)$

Linear Sum

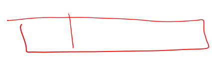
- With two recursive calls, we can enable parallel algorithm but not reduce the number of operations

```
def b_sum(S, start, stop):
    """Return the sum of numbers in S"""
    if start >= stop: # zero elements
        return 0
    elif start == stop-1: # one element
        return S[start]
    else:   # two or more elements in slice
        mid = (start + stop) // 2
        return b_sum(S, start, mid) + b_sum(S, mid, stop)
```

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$



if $n=0$ ret 1

$x \; (x^{n-1})$

$x \cdot r \uparrow$

x^n — $x \cdot x^{n-1}$

$x^n \rightarrow x^{n/2} \, x^{n/2}$

$y + y$

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in time proportional to n time (for we make n recursive calls)
- We can do better than this, however

Computing Power

Algorithm Power(*x*, *n*):
Input: A number *x* and integer *n* = 0
Output: The value x^n
if *n* = 0 **then**
return 1
if *n* is odd **then**
y = Power(*x*, (*n* - 1)/ 2)
return *x* · *y* · *y*
else
y = Power(*x*, *n*/ 2)
return *y* · *y*

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

Recursive Squaring

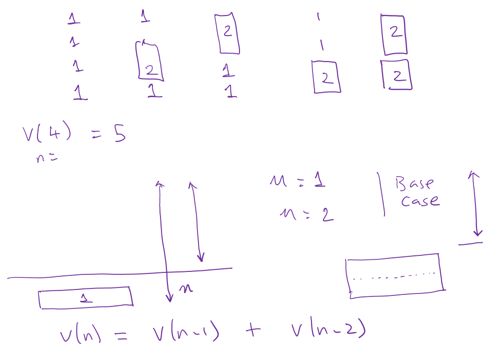
- A more efficient recursive algorithm

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

```
def power(x, n):
    """Compute the value x**n for integer n."""
    if n == 0:
        return 1
    else:
        partial = power(x, n // 2)
        result = partial * partial
        if n % 2 == 1:
            result *= x
        return result
```

Virahanka Numbers

- You have bricks of height 1 and height 2.
- How many ways can you create a tower of height 4?
  - V(4) = 5
  - V(1) = 1, V(2) = 2, V(3) = 3
- Sequence
  - prime numbers: 2, 3, 5, 7 …
  - Virahanka numbers: 0, 1, 1, 1, 2, 3, 5, …
- Interested in computing V(n)



Virahanka Numbers

- You have bricks of height 1 and height 2.
- How many ways can you create a tower of height n?

Number of ways to build a tower of height n with bottom brick of height 1

Number of ways to build a tower of height n with bottom brick of height 2

Virahanka Numbers

- You have bricks of height 1 and height 2.
- How many ways can you create a tower of height n?

Number of ways to build a tower of height n with bottom brick of height 1

Number of ways to build a tower of height n with bottom brick of height 2

- We see V(n) = V(n-1) + V(n-2)
- Easy to write an iterative algorithm for V(n)

Recursive Fibonacci

- Let n_k be the number of recursive calls by Fib(k)
  - $n_0 = 1$, $n_1 = 1$, $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$, $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$, $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

```python
def fib(n):
    """Return the nth Fibonacci number."""
    if n <= 1:
        return n
    else:
        return fib(n-2) + fib(n-1)
```

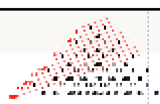
A Better Fibonacci Algorithm

- Reparameterization

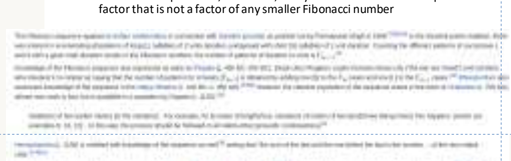
Algorithm LinearFibonacci(k):
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers (F_k, F_{k-1})
if $k = 1$ **then**
    return (k, 0)
else
    (i, j) = LinearFibonacci($k - 1$)
    return (i +j, i)

- LinearFibonacci makes k−1 recursive calls

Fibonacci Numbers



- Also known as Hemachandra numbers
- Why do we care about such numbers (factorial, prime, Hemachandra?)
- Appear in mysterious fashions in computer science, financial markets, biology, optics, and mathematics
  - Every third number is even
  - It is not known whether there are infinitely many Fibonacci primes
  - With the exception of 1, 8 and 144, every Fibonacci number has a prime factor that is not a factor of any smaller Fibonacci number



Summary

- Element uniqueness: Easy to write sloppy program
- Sum of items in an array: We can save space
- Computing powers: We can save time
- Fibonacci: Reparameterization let's save lots of time
- Quest: Design nice data structures and algorithms to solve more complicated problems