It’s a poor of memory that only works backwards

Algorithms

Quiz next Tuesday
Need to cancel today’s office hours, but add 9-10am, and 3-4pm tomorrow
What is an Algorithm?

- An **algorithm** is a sequence of instructions that one must perform in order to solve a well-formulated **problem**.
**Algorithm vs. Program**

- An algorithm is an “abstract” description of a process that is precise, yet general
  - Algorithms are described as generally as possible, so they can be analyzed and proven correct
- Programs are often specific implementations of an algorithm
  - For a specific machine
  - In a specific language
An Example: Buying a CD

1. Go to Best Buy
2. Got to the correct music genre section
3. Search the racks for the artist’s name
4. Take a copy of the CD.
5. Go to the register.
6. Check out using credit card.
7. Rip it onto your laptop.

1. Sign into iTunes.com
2. Goto iTunes Store
3. Type CD title into search
4. Scroll through Album list to find CD cover
5. Click “Buy Album”.
6. Accept Credit Card charge
7. Agree to download
Two Observations

• Given a problem, there may be more than one correct algorithms.

• However, the costs to perform different algorithms may be different.

• We can measure costs in several ways
  – In terms of time
  – In terms of space
Correctness

• An algorithm is **correct** only if it produces correct result for all input instances.
  
  – If the algorithm gives an incorrect answer for one or more input instances, it is an **incorrect** algorithm.

• Coin change problem
  
  – Input: an amount of money $M$ in cents
  
  – Output: the smallest number of coins

• US coin change problem
US Coin Change

72 cents

Two quarters, 22 cents left

Two dimes, 2 cents left

Two pennies

\[ r \leftarrow \text{Amount} \]
\[ q \leftarrow r/25 \]
\[ r \leftarrow r - 25 \cdot q \]
\[ d \leftarrow r/10 \]
\[ r \leftarrow r - 10 \cdot d \]
\[ n \leftarrow r/5 \]
\[ r \leftarrow r - 5 \cdot n \]
\[ p \leftarrow r \]

Can we generalize it?

Is it correct?
Change Problem

• Input:
  – an amount of money “Amount”
  – an array of denominations $c = (c_1, c_2, \ldots, c_d)$ in decreasing values

• Output: the smallest number of coins

\[ \text{Amount} = 40 \]
\[ c = (25, 20, 10, 5, 1) \]

```
Amount
n  0
For k from 1 to d
  i_k = \lfloor r/c_k \rfloor
  n = n + i_k
  r = r \# c_k \cdot i_k
return n
```

Incorrect algorithm!

The correct answer should be 2.

Is it correct?
Complexity of an Algorithm?

- **Complexity** — the cost in time and space of an algorithm as a function of the input’s size
  - Correct algorithms may have different complexities.
- The cost to perform an instruction may vary dramatically.
  - An instruction may be an algorithm itself.
  - The complexity of an algorithm is **NOT** equivalent to the number of instructions.
- Thinking algorithmically...
Recursive Algorithms

• **Recursion** is technique for describing an algorithm in terms of itself.
  
  – These recursive calls are to simpler, or reduced, versions of the original calls.
  
  – The simplest versions, called “base cases”, are merely declared (because the answer is known).

**Recursive definition:** \( \text{factorial}(n) = n \times \text{factorial}(n - 1) \)

**Base case:** \( \text{factorial}(1) = 1 \)
Example of Recursion

```python
def factorial(n):
    if (n == 1):
        return 1
    else:
        return n*factorial(n-1)
```

- Recursion is a useful technique for specifying algorithms concisely
- Recursion can be used to decompose large problems into smaller simpler ones
- Recursion can illuminate the non-obvious
Towers of Hanoi

• There are three pegs and a number of disks with decreasing radii (smaller ones on top of larger ones) stacked on Peg 1.

• Goal: move all disks to Peg 3.

• Rules:
  - At each move a disk is moved from one peg to another.
  - Only one disk may be moved at a time, and it must be the top disk on a tower.
  - A larger disk may never be placed upon a smaller disk.
A single disk tower
A single disk tower
A two disk tower
Move 1
Move 2
Move 3
A three disk tower
Move 2
Move 3

1

2

3
Move 4
Move 5
Move 6
Move 7
Simplifying the algorithm for 3 disks

- Step 1. Move the top 2 disks from 1 to 2 using 3 as intermediate
Simplifying the algorithm for 3 disks

- Step 2. Move the remaining disk from 1 to 3
Simplifying the algorithm for 3 disks

- Step 3. Move 2 disks from 2 to 3 using 1 as intermediate.
Simplifying the algorithm for 3 disks
Recursive Towers of Hanoi

- At first glance, the recursive nature of the towers of Hanoi problem may not be obvious.
- Consider, that the 3 disk problem must be solved as part of the 4 disk problem.
- In fact it must be solved twice! Moving the bottom disk once in-between.
The problem for 3 disks becomes

• A base case of a one-disk move from 1 to 3.
• A recursive step for moving 2 or more disks.

• To move \( n \) disks from Peg 1 to Peg 3, we need to
  – Move \( (n-1) \) disks from Peg 1 to Peg 2
    (Note: Peg 2 is the “unused” extra peg)
  – Move the \( n^{th} \) “bottom” disk from Peg 1 to Peg 3
  – Move \( (n-1) \) disks from Peg 2 to Peg 3
Towers of Hanoi Algorithm

def towersOfHanoi(n, fromPeg, toPeg):
    if (n == 1):
        print "Move disk from peg", fromPeg, "to peg", toPeg
        return
    unusedPeg = 6 - fromPeg - toPeg
    towersOfHanoi(n-1, fromPeg, unusedPeg)
    print "Move disk from peg", fromPeg, "to peg", toPeg
    towersOfHanoi(n-1, unusedPeg, toPeg)
    return

The number of disk moves is:

\[ T(1) = 1 \]

\[ T(n) = 2T(n-1) + 1 = 2^n - 1 \]

Exponential algorithm
Towers of Hanoi

• If you call towerOfHanoi with ____ it takes ____

- 1 disk ... 1 move
- 2 disks ... 3 moves
- 3 disks ... 7 moves
- 4 disks ... 15 moves
- 5 disks ... 31 moves
- ...
- ...
- 20 disks ... 1,048,575 moves
- 32 disks ... 4,294,967,295 moves
Another Algorithm: Sorting

• A very common problem is to arrange data into either ascending or descending order
  – Viewing, printing
  – Faster to search, find min/max, compute median/mode, etc.

• Lots of different sorting algorithms
  – From the simple to very complex
  – Some optimized for certain situations (lots of duplicates, almost sorted, etc.)
Class Exercise

• You are given a list of 10 numbers
  \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}\}
• Write down precise detailed instructions for sorting them in ascending order
Next Time

• We’ll look at your sorting algorithms more closely
• Are they correct?
• How many steps are used to sort $N$ items?
How to Sort?

• How would you describe the task of sorting a list of numbers to a 5-year old, who knows only basic arithmetic operations?

• Goal 1: A correct algorithm

• There are many possible approaches

• Each requires the atomic operation of comparing two numbers

• Are all sorting approaches equal?

• What qualities distinguish “good” approaches from those less good?
  – Speed? Space required?
**Method #1**  

Selection Sort

Find the smallest element and swap it with the first:

Find the next smallest element and swap it with the second:

Do the same for the third element:

And the fourth:

Finally, the fifth:

Completely sorted:
def selectionSort(list):
    first = 0
    while (first < len(list)):
        index = findMin(list, first)
        temp = list[index]
        list[index] = list[first]
        list[first] = temp
        first = first+1

def findMin(list,first):
    index = first
    for i in xrange(first+1, len(list)):
        if (list[i] < list[index]):
            index = i
    return index

\[ \frac{n(n - 1)}{2} \text{ comparisons} \]

(n - 1) swaps
Other Ways to Sort?

• Would you use this algorithm yourself?
  - Progress is slow, (i.e. moving one value to the front of the list after comparing to all others)

• Any Ideas?

• An Insertion Sort
Other Ways to Sort?

- Would you use this algorithm yourself?
  - Progress is slow, (i.e. moving one value to the front of the list after comparing to all others)
- Perhaps we can exploit recursion for sorting...
- Better yet, we can divide and conquer!
Method #2

**Merge Sort**

Split the list in half forming 2 sublists

Continue splitting sublists until lists are a just one item

Then combine sorted sublists together, by selecting the smallest value from the front of each sublist
def mergeSort(list):
    if (len(list) == 1):
        return list
    half = len(list)/2
    left = mergeSort(list[:half])
    right = mergeSort(list[half:]))
    return combine(left,right)

def combine(listL,listR):
    mergedList = []
    while (len(listL) > 0 and len(listR) > 0):
        if (listL[0] < listR[0]):
            mergedList.append(listL.pop(0))
        else:
            mergedList.append(listR.pop(0))
    while (len(listL) > 0):
        mergedList.append(listL.pop(0))
    while (len(listR) > 0):
        mergedList.append(listR.pop(0))
    return mergedList
N(N-1)/2 vs N \log_2 N

• For small numbers, perhaps not
  – N = 4, N(N-1)/2 = 6, N \log_2 N = 8
  – N = 8, N(N-1)/2 = 28, N \log_2 N = 24
  – N = 16, N(N-1)/2 = 120, N \log_2 N = 64

• But the difference can be quite large for a large list of numbers
  – N = 1000, N(N-1)/2 = 499500, N \log_2 N = 9966
Is Recursion the Secret Sauce?

- A noticeable difference between selection sort and merge sort, is that merge sort was specified as a recursive algorithm.
- Does recursion always lead to fast algorithms?
- Previously, I offered recursion as a tool for specifying algorithms concisely, in terms of a common repeated “kernel”.
Year 1202: Leonardo Fibonacci:

- He asked the following question:
  - How many pairs of rabbits are produced from a single pair in one year if every month each pair of rabbits more than 1 month old produces a new pair?
  - Here we assume that each pair has one male and one female, the rabbits never die, initially we have one pair which is less than 1 month old
  - $f(n)$: the number of pairs present at the beginning of month $n$
Fibonacci Number
Fibonacci Number

• Clearly, we have:
  – \( f(1) = 1 \) (the first pair we have)
  – \( f(2) = 1 \) (still the first pair we have because they are just 1 month old. They need to be more than one month old to reproduce)
  – \( f(n) = f(n-1) + f(n-2) \) because \( f(n) \) is the sum of the old rabbits from last month \( (f(n-1)) \) and the new rabbits reproduced from those \( f(n-2) \) rabbits who are old enough to reproduce.
  – \( f: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots \)
  – The solution for this recurrence is:

\[
f(n) = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1 - \sqrt{5}}{2} \right)^n
\]
Fibonacci Number

Recursive Algorithm

Exponential in time!

def fibonacciRecursive(n):
    if (n <= 2):
        return 1
    else:
        a = fibonacciRecursive(n-1)
        b = fibonacciRecursive(n-2)
        return a+b
Fibonacci Number

Iterative Algorithm

def fibonacci(n):
    f = [1, 1]
    for i in xrange(2, n):
        f += [f[i-1]+f[i-2]]
    return f[n-1]

Linear in time!
Is there a “Real difference”? 

- $10^1$
- $10^2$ Number of students in a department
- $10^3$ Number of students in the college of art and science
- $10^4$ Number of students enrolled at UNC
- ...
- ...
- $10^{10}$ Number of stars in the galaxy
- $10^{20}$ Total number of all stars in the universe
- $10^{80}$ Total number of particles in the universe
- $10^{100} <<$ Number of moves needed for 400 disks in the Towers of Hanoi puzzle

- Towers of Hanoi puzzle is *computable* but it is NOT feasible.
Is there a “Real” Difference?

- Growth of functions

<table>
<thead>
<tr>
<th>n</th>
<th>lgn</th>
<th>n</th>
<th>nlgn</th>
<th>n²</th>
<th>n³</th>
<th>2ⁿ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3.32</td>
<td>10</td>
<td>33</td>
<td>100</td>
<td>1,000</td>
<td>1024</td>
</tr>
<tr>
<td>100</td>
<td>6.64</td>
<td>100</td>
<td>664</td>
<td>10,000</td>
<td>1,000,000</td>
<td>1.2 x 10²⁰</td>
</tr>
<tr>
<td>1000</td>
<td>9.97</td>
<td>1000</td>
<td>9970</td>
<td>1,000,000</td>
<td>10⁹</td>
<td>1.1 x 10³⁰</td>
</tr>
</tbody>
</table>
Asymptotic Notation

• **Order of growth** is the interesting measure:
  - Highest-order term is what counts
  - As the input size grows larger it is the high order term that dominates

• Θ notation: \( \Theta(n^2) = \text{“this function grows similarly to } n^2 \text{”} \).

• Big-O notation: \( O(n^2) = \text{“this function grows at least as slowly as } n^2 \text{”} \).
  - Describes an upper bound.
Big-O Notation

\[ f(n) = O(g(n)) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \]
\[ 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \]

- What does it mean?
  - If \( f(n) = O(n^2) \), then:
    - \( f(n) \) can be larger than \( n^2 \) sometimes, but...
    - We can choose some constant \( c \) and some value \( n_0 \) such that for every value of \( n \) larger than \( n_0 \): \( f(n) < cn^2 \)
    - That is, for values larger than \( n_0 \), \( f(n) \) is never more than a constant multiplier greater than \( n^2 \)
    - Or, in other words, \( f(n) \) does not grow more than a constant factor faster than \( n^2 \).
Visualization of $O(g(n))$
Big-O Notation

\[ 2n^2 = O(n^2) \]
\[ 1,000,000n^2 + 150,000 = O(n^2) \]
\[ 5n^2 - 7n + 20 = O(n^2) \]
\[ 2n^3 + 2 \neq O(n^2) \]
\[ n^{2.1} \neq O(n^2) \]
Big-O Notation

• Prove that: \(20n^2 + 2n + 5 = O(n^2)\)
• Let \(c = 21\) and \(n_0 = 4\)
• \(21n^2 > 20n^2 + 2n + 5\) for all \(n > 4\)
  \[n^2 > 2n + 5\] for all \(n > 4\)
  TRUE
\(\Theta\)-Notation

- Big-O is not a tight upper bound. In other words \(n = O(n^2)\)
- \(\Theta\) provides a tight bound

\[ f(n) = \Theta(g(n)) \]: there exist positive constants \(c_1, c_2,\) and \(n_0\) such that

\[ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \] for all \(n \geq n_0\)

- \(n = O(n^2) \neq \Theta(n^2)\)
- \(200n^2 = O(n^2) = \Theta(n^2)\)
- \(n^{2.5} \neq O(n^2) \neq \Theta(n^2)\)
Visualization of $\Theta(g(n))$

$c_2g(n)$

$f(n)$

$c_1g(n)$

$n_0$
Some Other Asymptotic Functions

• Little $o$ – A non-tight asymptotic upper bound
  - $n = o(n^2)$, $n = O(n^2)$
  - $\exists n^2 \neq o(n^2)$, $\exists n^2 = O(n^2)$

• $\Omega$ – A lower bound

\[ f(n) = \Omega(g(n)) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \]
\[ f(n) \geq cg(n) \text{ for all } n \geq n_0 \]

- $n^2 = \Omega(n)$

• $\omega$ – A non-tight asymptotic lower bound

• $f(n) = \Theta(n) \iff f(n) = O(n) \text{ and } f(n) = \Omega(n)$
Visualization of Asymptotic Growth

\( \Omega(f(n)) \)

\( \Theta(f(n)) \)

\( O(f(n)) \)

\( o(f(n)) \)

\( \omega(f(n)) \)

\( n_0 \)
Analogy to Arithmetic Operators

\[ f(n) = O(g(n)) \quad \approx \quad a \leq b \]
\[ f(n) = \Omega(g(n)) \quad \approx \quad a \geq b \]
\[ f(n) = \Theta(g(n)) \quad \approx \quad a = b \]
\[ f(n) = o(g(n)) \quad \approx \quad a < b \]
\[ f(n) = \omega(g(n)) \quad \approx \quad a > b \]
Measures of Complexity

• Best case
  – Super-fast in some limited situation is not very valuable information

• Worst case
  – Good upper-bound on behavior
  – Never get worse than this

• Average case
  – Averaged over all possible inputs
  – Most useful information about overall performance
  – Can be hard to compute precisely
Complexity

• Time complexity is not necessarily the same as the space complexity

• Space Complexity: how much space an algorithm needs (as a function of $n$)

• Time vs. space
Next Time

• Algorithm design techniques
  – Exhaustive search
  – Greedy algorithms
  – Branch and bound algorithms
  – Dynamic programming
  – Divide and conquer algorithms
  – Randomized algorithms

• Tractable vs intractable algorithms