CS1003:
Intro to CS, Summer 2008

Lecture #04
Data Representation, Algorithms

Instructor: Arezu Moghadam
arezuc@cs.columbia.edu

Agenda

- Finish up data representation
  - Two’s complement
- Algorithms
  - Iterative structures; ch 5.4
  - Recursive structures; ch 5.5
  - Algorithms efficiency; ch 5.6
- HW1 review
Storing Integers

- Binary system
  - Positive and negative numbers
  - **Two’s complement** the most popular!
- Positive: The usual binary representation
  - Iteratively divide by 2 until the quotient is less than 2
- Negative:
  - First calculate the positive representation
  - Flip all bits from 0 to 1 and 1 to 0 → one’s complement
  - Add 1 to the result → two’s complement
- Example:
  - +6 based on bit patterns of length 4: 0110
  - -6 based on bit patterns of length 4: 1001 + 1 = 1010

Two’s complement

- The left most bit is the sign bit
- The problem of overflow!
  - Limit to the size of the value that can be represented
  - 9 can’t be represented with 4 bits!

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Value represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>-1</td>
</tr>
<tr>
<td>110</td>
<td>-2</td>
</tr>
<tr>
<td>101</td>
<td>-3</td>
</tr>
<tr>
<td>100</td>
<td>-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Value represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1000</td>
<td>-8</td>
</tr>
</tbody>
</table>
Algorithms review

- Strategies of coming up with algorithms...
  - “Get foot in the door”: try to get an intuitive grasp on the problem first, conceptually
  - Stepwise refinement: take the big picture and break into smaller pieces
  - Determine if there are any iterative structures to be implemented
  - Keep boundary conditions in mind!

Iterative structures

- A collection of instructions is repeated in a looping manner
- Elements of an interactive structure
  - A loop; **while** or **for** loop
  - A condition determining the loop’s termination or continuation
Iterative structures, cont’d.

- Last time **while** loop
  - `while(condition == true) {....}`
- More loop constructs:
  - **for**: useful for situations where we’re doing a loop N times
    - `for(i=0; i < 10; i++) { ... }` runs exactly 10 times
    - Three parts: initialize, condition, increment
    - `for(; i < 10;) { ... } == while(i < 10) { ... }

---

while and **for** loops

- `int i;
  for (i=0; i<10; i++){
    ------
    ------
    ------
  }
  }
- Initialization, condition and incrementing all in the loop statement

- `int i=0;
  while (i < 10) {
    ------
    ------
    i++;
  }
  }
- Initialization before the loop statement
- Incrementing the loop counter inside the loop
Other forms of **while** loop

- **do-while:**
  - do {
    - ----
    - ----
    - ----
  } while(condition == true)
- **Example:**
  - do {
    - ----
    - ----
  } while (1<0) ?

- Using **break** keyword
  - while (true) {
    - ------
    - if (some condition)
      break;
    - ------
  }
Let’s revisit our examples

1. Print out the first $n$ numbers, and keep a running total... **using a for loop**
2. Print out the first $n$ Fibonacci numbers
3. Write a function that calculates $x^n$ (i.e., raise $x$ to the $n$ power)
4. Reverse a list (array) of numbers

---

Print out first $n$ numbers

- int $i$;
- int sum = 0;
- for ($i=0; \ i<=n; \ i++$) {
  - printf("%d \n", $i$);
  - sum = sum + $i$;
- }
- printf("total is: %d \n", sum);
Let’s revisit our examples

1. Print out the first \( n \) numbers, and keep a running total...
2. Print out the first \( n \) Fibonacci numbers
3. Write a function that calculates \( x^n \) (i.e., raise \( x \) to the \( n \) power)
4. Reverse a list (array) of numbers

Another way of looking at repetition

- Fibonacci numbers:
  - \( 0,1,1,2,3,5,8,13,21,34,55,89,... \)
- Pattern: \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \), right?
- We can actually encode that in a computer
  - **Recursion:** Define a solution in terms of a smaller version of itself
  - Must have *stopping* (base) case(s)
  - What’s the base case for the above recursion?
Recursive structures

- Another way of looking at repetition
- In iterative structures a set of instructions are completed and then repeated again
- Recursion involves repeating the set of instructions as a subtask of itself
- Instead of one-after-the-other, one is performed within the other
- Example: \( \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2); \)

Fibonacci code snippet

```c
fib (int n) {
    if (n == 1) return 0;
    if (n == 2) return 1;
    int sum=0;
    sum = fib(n-1) + fib(n-2);
    return sum;
}
```

- We need some base case(s) condition(s) to stop the recursion
- First, come up with the recursive statement
Let’s revisit our examples

1. Print out the first \( n \) numbers, and keep a running total...
2. Print out the first \( n \) Fibonacci numbers
3. Write a function that calculates \( x^n \) (i.e., raise \( x \) to the \( n \) power)
4. Reverse a list (array) of numbers

---

\( x^n \) using recursion

- Recursive statement:
  - \( \text{power}(x,n) = x \times \text{power}(x,n-1) \);
- Base or stopping condition:
  - \( \text{power}(x,0) = 1; \)
  - Translates to: if \( n==0 \) return 1;
- \( \text{power}(x,n) \) {
  - if \( n==0 \) return 1;
  - return \( x \times \text{power}(x,n-1) \);
- }
Iterative or recursive?

- All iterative structures can be implemented recursively and vice-versa
- Going back to our very first example
  - Total of the first n numbers
  - Implemented in an iterative fashion using `for` loop
  - Can we implement it using recursive structure?
  - Statement: \( \text{sum}(n) = \text{sum}(n-1) + n; \)
  - Base condition: if \( n=1 \) return 0;

Another recursive example

- Binary search: works for a sorted list of information
- Basic idea: pick the middle element
  - If that’s what we’re looking for, done
  - If it’s larger, recursively search the “top half”
  - Otherwise, recursively search the “bottom half”
  - If we’re stuck with an empty list, we failed
Sequential or binary search?

<table>
<thead>
<tr>
<th>Original list</th>
<th>First sublist</th>
<th>Second sublist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice Bob Carol David Elaine Fred George Harry Irene John Kelly Larry Mary Nancy Oliver</td>
<td>Irene John Kelly</td>
<td>Irene John Kelly</td>
</tr>
</tbody>
</table>

Recursion, redux

- Idea: instead of using explicit loops, cast problem in terms of itself
- *Base case(s) and recursive case*
- How can we compute n! recursively?
n!

- An algorithm to compute n! – recursively
- Base case:
  - 0! = 1
- Recursive case:
  - n! = n * (n-1)!
  - factorial(n) = n * factorial(n-1)
- Let’s write the code snippet!

Algorithm efficiency

- Often, there’s multiple ways to implement an algorithm
- How to characterize if one’s better or not?
- Two primary considerations:
  - How fast does an algorithm run?
  - How much memory does an algorithm take?
- Let’s focus on the first one for now
Our multiple Fibonacci algorithms

- Do they run at the same speed?
- Let’s try fib(10)... then 20... then 40
- Hmm, why do they differ?
- And can we classify this difference

How fast does an algorithm run?

- Let’s first think of it in the context of steps
- How long might a linear search take through a list of N elements?
- Canonical way to characterize this is to use “big-theta” notation $\Theta$
  - Key insight: we’re interested in orders of magnitude, not constants
Big-Theta notation

- Basic intuition:
  - Find the number of steps in terms of $n$ or other variables
  - Drop any constants or additive lower-order terms
  - Put a $\Theta( )$ around the result
  - Common: $\Theta(1), \Theta(\log N), \Theta(N), \Theta(N^2), \Theta(2^n)$

Big-theta notation

- Basic intuition:
  - Find the number of steps in terms of $n$ or other variables
  - Drop any constants or additive lower-order terms
  - Put a $\Theta( )$ around the result
- Let’s look at some of algorithms we discussed previously and see what their big-theta complexity is...
$x^n$ where $n$ is an Integer

- What is the time complexity of the recursive algorithm we discussed before?
- $x^n = x \times x^{n-1}$
- Idea: take the $x$ and multiply it by $x$, $(n-1)$ times
- $\Theta(n)$
- Can we do better?

$x^n$, revised

- What about this recursive statement?
  - $x^n = (x^{n/2})^2$
- $\Theta(\lg n)$
Binary search

<table>
<thead>
<tr>
<th>Original list</th>
<th>First sublist</th>
<th>Second sublist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td></td>
<td>Irene</td>
</tr>
<tr>
<td>Bob</td>
<td></td>
<td>John</td>
</tr>
<tr>
<td>Carol</td>
<td></td>
<td>Kelly</td>
</tr>
<tr>
<td>David</td>
<td></td>
<td>Larry</td>
</tr>
<tr>
<td>Elaine</td>
<td></td>
<td>Mary</td>
</tr>
<tr>
<td>Fred</td>
<td></td>
<td>Nancy</td>
</tr>
<tr>
<td>George</td>
<td></td>
<td>Oliver</td>
</tr>
<tr>
<td>Harry</td>
<td>Irene</td>
<td>John</td>
</tr>
<tr>
<td>Irene</td>
<td>John</td>
<td>Kelly</td>
</tr>
<tr>
<td>John</td>
<td>Kelly</td>
<td>John</td>
</tr>
<tr>
<td>Kelly</td>
<td>Larry</td>
<td>Kelly</td>
</tr>
<tr>
<td>Larry</td>
<td>Mary</td>
<td>Nancy</td>
</tr>
<tr>
<td>Mary</td>
<td>Nancy</td>
<td>Oliver</td>
</tr>
<tr>
<td>Nancy</td>
<td>Oliver</td>
<td></td>
</tr>
</tbody>
</table>

At each step a subset of \( n/2^i \) has to be searched.

The last step we are left with one object to compare.

Suppose total number of steps: \( h \)

Then \( n/2^h = 1 \) \( \Rightarrow h = \lg n \)

Running time asymptotically: \( \Theta(\lg n) \)
Other algorithms?

- Sort the contents of an array; ex. sorting a list of names
  - Insertion sort
  - Bubble sort
- We’ll continue to do more “interesting” algorithms as the semester proceeds

Sorting

- Common problem: given data, sort it in some fashion
- Most common-type is *comparison-based sort*
- Can you come up with way to sort information?
- Many different kinds; we’ll look at two today
  - Insertion sort
  - Bubble sort
Insertion sort

- Sort the list within itself
- No temporary location
- Iteratively!
- Analyze the running time complexity

Sorting the list Fred, Alex, Diana, Byron, and Carol alphabetically
Running time of the insertion sort

- Worst case scenario
- At each iteration the pivot is compared with all previous entries
  - \[ 1 + 2 + 3 + \ldots + (n-1) = \left(\frac{1}{2}\right)(n^2-n) = \Theta(n^2) \]

Next time

- Finish up the intro to algorithms
- Start data structures