Lecture 3: Algorithms and Complexity

Study Chapter 2.1-2.8
Topics

• Algorithms
  – Correctness
  – Complexity

• Some algorithm design strategies
  – Exhaustive
  – Greedy
  – Recursion

• Asymptotic complexity measures
What is an algorithm?

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

```
input
algorithm
output
```

**Problem: Complexity**

**Algorithm: Correctness**

**Complexity**
Problem: US Coin Change

- **Input**
  - an amount of money 
    \[ 0 \leq M < 100 \text{ in cents} \]

- **Output:**
  - \( M \) cents in US coins using the minimal number of coins

- **Example**
  - 72 cents
  - Two quarters
  - Two dimes
  - Two pennies

Is it correct?
Algorithm 1: Greedy strategy

Greedy coin alg

Algorithm description

\[ r \leftarrow M \]
\[ q \leftarrow \lfloor r / 25 \rfloor \]
\[ r \leftarrow r - 25 \cdot q \]
\[ d \leftarrow \lfloor r / 10 \rfloor \]
\[ r \leftarrow r - 10 \cdot d \]
\[ n \leftarrow \lfloor r / 5 \rfloor \]
\[ r \leftarrow r - 5 \cdot n \]
\[ p \leftarrow r \]

Use large denominations as long as possible

Two quarters, 22 cents left

Two dimes, 2 cents left

Two pennies
Algorithm 2: Exhaustive strategy

• Enumerate *all* combinations of coins. Record the combination totaling to $M$ with fewest coins
  – *All* is impossible. Limit the multiplicity of each coin!
  – First try (80,000 combinations)

<table>
<thead>
<tr>
<th>coin</th>
<th>Quarter</th>
<th>Dime</th>
<th>Nickel</th>
<th>Penny</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicity</td>
<td>0..3</td>
<td>0..9</td>
<td>0..19</td>
<td>0..99</td>
</tr>
</tbody>
</table>

– Better (200 combinations)

<table>
<thead>
<tr>
<th>coin</th>
<th>Quarter</th>
<th>Dime</th>
<th>Nickel</th>
<th>Penny</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicity</td>
<td>0 .. 3</td>
<td>0 .. 4</td>
<td>0 .. 1</td>
<td>0 .. 4</td>
</tr>
</tbody>
</table>

Is it correct?
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.

• US coin change problem
  – It is easy to show that the exhaustive algorithm is correct
  – The greedy algorithm is correct but we didn’t really show it
Observations

• Given a problem, there may be many correct algorithms.
  – They give identical outputs for the same inputs
  – They give the expected outputs for any valid input

• The costs to perform different algorithms may be different.

• US coin change problem
  – The exhaustive algorithm checks 200 combinations
  – The greedy algorithm performs just a few arithmetic operations
Change Problem: generalization

- **Input:**
  - an amount of money \( M \)
  - an array of denominations \( c = (c_1, c_2, \ldots, c_d) \) in order of decreasing value

- **Output:** the smallest number of coins

\[
M = 40 \\
c = (25, 20, 10, 5, 1)
\]

\[
\begin{align*}
M &\leftarrow M \\
n &\leftarrow 0 \\
\text{for } k &\leftarrow 1 \text{ to } d \\
i_k &\leftarrow \lfloor r / c_k \rfloor \\
n &\leftarrow n + i_k \\
r &\leftarrow r - i_k c_k \\
\text{return } n
\end{align*}
\]

Incorrect algorithm!

To show an algorithm was incorrect we showed an input for which it produced the wrong result. How do we show that an algorithm is correct?

The correct answer should be 2.
How to Compare Algorithms?

• **Complexity** — the cost of an algorithm can be measured in either time and space
  – Correct algorithms may have different complexities.

• How do we assign “cost” for time?
  – Roughly proportional to number of instructions performed by computer
  – Exact cost is difficult to determine and not very useful
    • Varies with computer, particular input, etc.

• How to analyze an algorithm’s complexity
  – Depends on algorithm design
Recursive Algorithms

- Recursion is an algorithm design technique for solving problems in terms of simpler subproblems
  - The simplest versions, called base cases, are merely declared.
    - Recursive definition: \( \text{factorial}(n) = n \times \text{factorial}(n - 1) \)
    - Base case: \( \text{factorial}(1) = 1 \)
  - Easy to analyze
- Thinking recursively...
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:
  - When a disk is moved from one peg it must be placed on another peg.
  - 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 1
Move 2
Move 3
Move 4

Diagram shows a sequence labeled 1, 2, 3 with an arrow indicating movement from 1 to 2 and then 2 to 3.
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
The problem for N disks becomes

- A base case of a one-disk move.
- A recursive step for moving n-1 disks.

To move n disks from Peg 1 to Peg 3, we need to
- Move (n-1) disks from Peg 1 to Peg 2
- Move the n\textsuperscript{th} disk from Peg 1 to Peg 3
- Move (n-1) disks from Peg 2 to Peg 3

The number of disk moves is

\[ T(1) = 1 \]

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

- If you play HanoiTowers 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
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 sorting algorithms
  – From the simple to very complex
  – Some optimized for certain situations (lots of duplicates, almost sorted, etc.)
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:

“In-place” sort
Selection sort

```python
def selectionSortRecursive(a, first, last):
    if (first < last):
        index = indexOfMin(a, first, last)
        temp = a[index]
        a[index] = a[first]
        a[first] = temp
        a = selectionSortRecursive(a, first+1, last)
    return a
```

```python
def indexOfMin(arr, first, last):
    index = first
    for k in xrange(index+1, last):
        if (arr[k] < arr[index]):
            index = k
    return index
```

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

\[n - 1\] swaps

Quadratic in time
Year 1202: Leonardo Fibonacci

- He asked the following question:
  - How many pairs of rabbits are produced from a single pair in \( n \) months if every month each pair of rabbits more than 1 month old produces a new pair?
  - Here we assume that each pair born has one male and one female and breeds indefinitely
  - The initial pair at month 0 are newborns
  - Let \( f(n) \) be the number of rabbit pairs present at the beginning of month \( n \)
Fibonacci Number

<table>
<thead>
<tr>
<th>month</th>
<th>rabbit pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 (newborn)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Fibonacci Number

• Clearly, we have:
  – \( f(0) = 1 \) (the original pair, as newborns)
  – \( f(1) = 1 \) (still the original pair because newborns need to mature a month before they reproduce)
  – \( f(n) = f(n-1) + f(n-2) \) in month \( n \) we have
    • the \( f(n-1) \) rabbit pairs present in the previous month, and
    • newborns from the \( f(n-2) \) rabbit pairs present 2 months earlier
  – \( f: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots \)
  – The solution for this recurrence is \( (n > 0): \)

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

Recursive Algorithm

Exponential time!

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

def fibonacciIterative(n):
    f0, f1 = 1, 1
    for i in xrange(0, n):
        f0, f1 = f1, f0 + f1
    return f0

Iterative Algorithm

Linear time!
Orders of magnitude

- $10^1$
- $10^2$ Number of students in computer science 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></th>
<th>1</th>
<th>lgn</th>
<th></th>
<th></th>
<th>n^2</th>
<th>n^3</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<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>1</td>
<td>3.32</td>
<td>10</td>
<td>33</td>
<td>100</td>
<td>1,000</td>
<td>1,024</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>6.64</td>
<td>100</td>
<td>664</td>
<td>10,000</td>
<td>1,000,000</td>
<td>1.2 x 10^30</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>9.97</td>
<td>1000</td>
<td>9970</td>
<td>1,000,000</td>
<td>10^9</td>
<td>1.1 x 10^301</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

• $\Theta$ notation: $\Theta(n^2) =$ “this function grows similarly to $n^2$”.

• Big-O notation: $O(n^2) =$ “this function grows no faster than $n^2$”.
  – 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

\[ 2n^2 = O(n^2) \]
\[ 1,000,000n^2 + 150,000 = O(n^2) \]
\[ n^2 + 1,000,000n + 20 = O(n^2) \]
\[ 3n + 4 = O(n^2) \]
\[ 2n^3 + 2 \neq O(n^2) \]
\[ n^{2.1} \neq O(n^2) \]
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
**Θ-Notation**

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

\[
f(n) = Θ(g(n)): \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}
\]
\[
0 ≤ c_1 g(n) ≤ f(n) ≤ c_2 g(n) \text{ for all } n ≥ n_0
\]

- \(n = O(n^2) \neq Θ(n^2)\)
- \(200n^2 = O(n^2) = Θ(n^2)\)
- \(n^{2.5} \neq O(n^2) \neq Θ(n^2)\)
Visualization of $\Theta(g(n))$
Some Other Asymptotic Functions

• Little $o$ – A **non-tight** asymptotic upper bound
  
  – $n = o(n^2), \ n = O(n^2)$
  
  – $3n^2 \neq o(n^2), \ 3n^2 = O(n^2)$

• $\Omega$ – 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

\[ o(f(n)) \]

\[ O(f(n)) \]

\[ \Theta(f(n)) \]

\[ \Omega(f(n)) \]

\[ \omega(f(n)) \]

\[ o(f(n)) \]

\[ O(f(n)) \]

\[ \Theta(f(n)) \]

\[ \Omega(f(n)) \]

\[ \omega(f(n)) \]

\[ n_0 \]
Analogy to Arithmetic Operators

\[ f(n) = O(g(n)) \sim f \leq g \]
\[ f(n) = \Omega(g(n)) \sim f \geq g \]
\[ f(n) = \Theta(g(n)) \sim f = g \]
\[ f(n) = o(g(n)) \sim f < g \]
\[ f(n) = \omega(g(n)) \sim f > g \]
Measures of complexity

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

- **Worst case**
  - Good upper-bound on behavior
  - Never gets worse than this

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

- Space Complexity $Sp(n)$: how much memory an algorithm needs (as a function of $n$)
- Space complexity $Sp(n)$ is not necessarily the same as the time complexity $T(n)$
  - $T(n) \geq Sp(n)$
Next Time

• Our first “bio” algorithm
• Read book 4.1 – 4.3