EL9343

Data Structure and Algorithm

Lecture 2: Mathematical Foundations, Solving Recurrences

Instructor: Pei Liu

Some Logistic Issues

- Homework #1 is due tonight
  - If you don’t follow the guideline to generate a clean scan of you answer, your submission will be returned. Please re-generate and re-submit by Friday.
  - Submission on Gradescope

We only accept a single PDF for homework. If it is handwritten, please make sure you use some APP to scan all the pages clearly with brightness/contrast adjusted. You can use Adobe Scan, iOS Notes App (https://www.macworld.com/article/232686/how-to-scan-documents-and-make-pdfs-using-notes-on-your-iphone-or-ipad.html) or some other third party software. Please don't just take pictures of your solution, as it is very difficult to read by the grader.

Once you upload the pdf, it prompts you to tag the pages for each questions. Please tag accurately. Otherwise, the grader may not be able to find where are your solutions.

Some Logistic Issues

- We use piazza.com for discussion of homework and Lecture
  - You should have received an announcement email
  - All students from three sessions are in the same discussion group
  - Me and TA are also there
  - If you ask a question that is specific about your session, please mention the session name in your question.

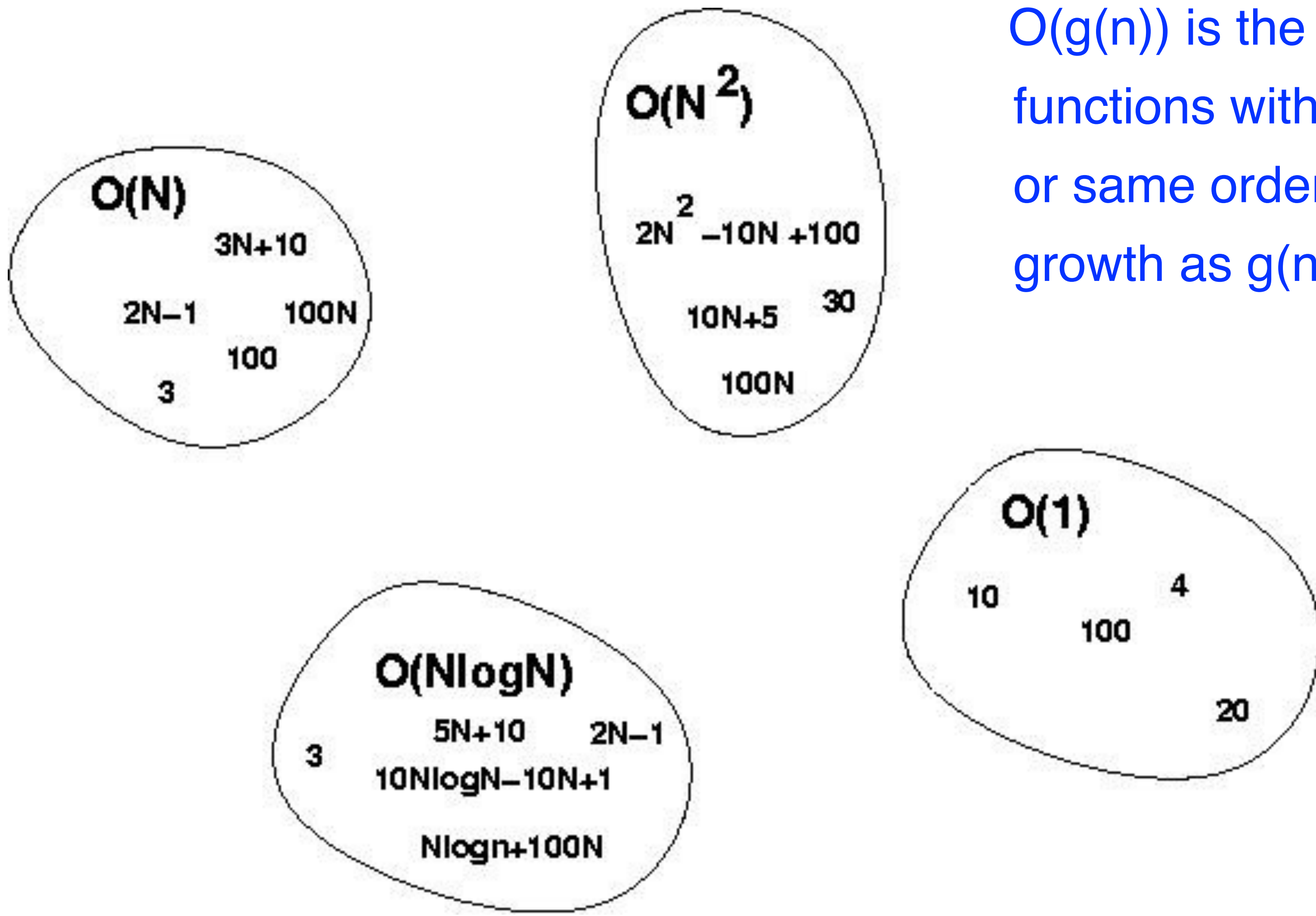
Last Lecture

- How to evaluate the efficiency of algorithm
  - By running time
  - As a function of the input size *n* (i.e., *f(n)*).
- Asymptotic analysis
  - Rough measure characterizes order of growth
  - Ignoring the constant factor
    - $347n$ is $\Theta(n)$
  - Constant factors of exponentials cannot be ignored
    - 2^{3n} is not $O(2^n)$
  - Concentrating on trends of the large value of n
    - $3n^2 \log n + 25n \log n$ is $\Theta(n^2 \log n)$

Asymptotic Notation

- O notation: asymptotic “less than”:
  - f(n)=O(g(n)) implies: f(n) “≤” g(n)
- Ω notation: asymptotic “greater than”:
  - f(n)= Ω (g(n)) implies: f(n) “≥” g(n)
- Θ notation: asymptotic “equality”:
  - f(n)= Θ (g(n)) implies: f(n) “=” g(n)

Big-O Visualization



O(g(n)) is the set of functions with smaller or same order of growth as g(n)

Exercise

For a pair of functions f(n) and g(n), their relative complexity relation can be:

A. f(n) is O(g(n))
B. f(n) is Ω(g(n))
C. f(n) = Θ(g(n))

For the following pairs, please write the corresponding relation (A, B or C) in the box next to each pair:

(1) $f(n)=\log n^2; g(n)=\log n+5$
(2) $f(n) = 2^n$; $g(n) = 10n^2$
(3) $f(n) = \log\log n$; $g(n) = \log n$

Today

- Some mathematical foundations
  - Review of some mathematical functions
  - Mathematical Induction
- Solving recurrence
  - Iteration method: recursion tree
  - Substitution method
  - Master method

Common Functions

- Polynomial: Powers of n, such as n^4.
- Exponential: A constant (not 1) raised to the power n, such as 3^n.
- Polylogarithmic: Powers of log n, such as $(\log n)^7$. We will usually write this as $\log^7 n$.

Logarithmic Functions

- In algorithm analysis we often use the notation "log n" without specifying the base
  - Binary logarithm: $\lg n = \log_2 n$
  - Natural logarithm: $\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

Blank page

Common Summations

- Arithmetic series: $$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$
- Geometric series: $$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1}-1}{x-1} \left(x \neq 1\right)$$
  - Special case: |x| < 1: $$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$
- Harmonic series: $$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$
- Other important formulas: $$\sum_{k=1}^{n} \lg k \approx n \lg n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \ldots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Blank page

Mathematical Induction

- A powerful, rigorous technique for proving that a statement S(*n*) is true for ***every*** natural number *n*, no matter how large n is.

- Proof:
  - **Basis step**: prove that the statement is true for base case (e.g., n = 1)
  - **Inductive step:** assume that statement is true for n (or all integers <=n), and then prove that statement is true for n+1

Mathematical Induction

- Prove that: $2n + 1 \le 2^n$ for all $n \ge 3$
- **Basis step:**
  - $n = 3$: $2 * 3 + 1 \le 2^3 \Leftrightarrow 7 \le 8$ TRUE
- **Inductive step:**
  - Assume inequality is true for n, and prove it for (n+1):
  - $2n + 1 \le 2^n$ must prove: $2(n + 1) + 1 \le 2^{n+1}$
  - $2(n + 1) + 1 = (2n + 1) + 2 \le 2^n + 2 \le$

    $\le 2^n + 2^n = 2^{n+1}$, since $2 \le 2^n$ for $n \ge 1$

Recursive Algorithms

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input

Example 1:

- Factorial: n!

$$\operatorname{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \operatorname{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Pseudocode (recursive):

```
function factorial is:
input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × … × 1]

    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]

end factorial
```

Recursive Algorithms

Example 2: binary search:
searching a sorted array for a single element by cutting the array in half with each recursive pass.

```
/*
  Binary Search Algorithm.

  INPUT:
       data is a array of integers SORTED in ASCENDING order,
       toFind is the integer to search for,
       start is the minimum array index,
       end is the maximum array index
  OUTPUT:
       position of the integer toFind within array data,
       -1 if not found
*/
int binary_search(int *data, int toFind, int start, int end)
{
   //Get the midpoint.
   int mid = start + (end - start)/2;   //Integer division

   //Stop condition.
   if (start > end)
      return -1;
   else if (data[mid] == toFind)        //Found?
      return mid;
   else if (data[mid] > toFind)         //Data is greater than toFind, search lower half
      return binary_search(data, toFind, start, mid-1);
   else                                 //Data is less than toFind, search upper half
      return binary_search(data, toFind, mid+1, end);
}
```

Recurrences and Running Time

- An equation or inequality that describes a function in terms of its value on smaller inputs.

$$T(n) = 2T(n/2) + n$$

$$T(1) = C$$

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
  - Find an explicit formula of the expression
  - Bound the recurrence by an expression that involves n

Example Recurrences

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$ $\Theta(\log n)$
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$ $\Theta(n)$
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

Methods for Solving Recurrences

- Iteration method
  - Usually solved by expanding recursion tree
- Substitution method
- Master method

The Recursion-tree Method

Convert the recurrence into a tree:

- Each node represents the cost incurred at various levels of recursion
- Sum up the costs of all levels

Example 1

- $T(n) = 2T(n/2) + \Theta(n)$
- How to solve this recurrence by the recursion-tree method?

Blank page

The Substitution Method

- Guess a solution
- Use induction to prove that the solution works

The Substitution Method

- Guess a solution
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation
    - $T(n) \le c\, g(n)$, for some $c > 0$ and $n \ge n_0$
  - Induction hypothesis: $T(k) \le c\, g(k)$ for all $k < n$
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants c and n_0 for which the **induction goal** holds

Back to Example 1

- $T(n) = 2T(n/2) + \Theta(n)$

- How to solve this recurrence by the substitution method?

Example 2

- $T(n) = 4T(n/2) + n$

- How to solve this recurrence by the recursion-tree method?
  - For "guessing" the solution for substitution method

- How to solve this recurrence by the substitution method
  - For verifying the answer got from recursion tree

Blank page

Example 3

- $T(n) = T(n/4) + T(n/2) + n^2$
- How to solve this recurrence by the recursion-tree method?
- Verify it by substitution method

Blank page

Master's Method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Idea: compare $f(n)$ with $n^{\log_b a}$

- $f(n)$ is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^{ε}
- $f(n)$ is asymptotically equal with $n^{\log_b a}$

Master's Method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, a ≥ 1, b > 1, and f(n) > 0

- **Case 1:** if $f(n) = O(n^{\log_b a-\epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$;
- **Case 2:** if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$;
- **Case 3:** if $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Back to Example 2

- $T(n) = 4T(n/2) + n$

- How to solve this recurrence by the master method?

What's next...

- Divide-and-Conquer algorithms (Read CLRS Chapter 4)
- Some basic sorting algorithms (Read CLRS Chapter 2)
  - Insertion sort
  - Bubble sort
  - Mergesort
  - ...