CS583 Lecture 01

Jana Kosecka

some materials here are based on Profs. E. Demaine , D. Luebke
A.Shehu, J-M. Lien and Prof. Wang's past lecture notes

Course Info

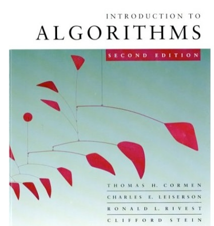
- course webpage:
  - from the syllabus on http://cs.gmu.edu/ or
  - http://cs.gmu.edu/~kosecka/cs583/
- http://mymason.gmu.edu//
- Information you will find
  - course syllabus, time table
  - office hours
  - .pdf copies of the lectures
  - handouts, practice problems

Prerequisite

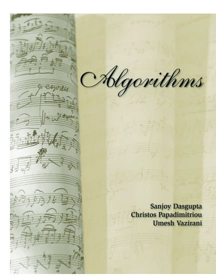
- Data structures and algorithms (CS 310)
- Formal methods and models (CS 330)
- Calculus (MATH 113, 114, 213)
- Discrete math (MATH 125)
- Ability to program in a high-level language that supports recursion

Textbook

- **Introduction to Algorithms** by T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, The McGraw-Hill Companies, 2nd Edition (2001)



- I also recommend you read the following book: **Algorithms**, by S. Dasgupta, C. Papadimitriou, and U. Vazirani, McGraw-Hill, 2006



- http://mitpress.mit.edu/algorithms/

Grades

- Short Quizes every 2 weeks (30%)
- Practice Problems
- Midterm Exam 30%
- Final Exam 40%
- Make-up tests will **NOT** be given for missed examinations

Other Important Info

- **Email**
  - make sure your gmu mail is activated
  - send only from your gmu account; mails might be filtered if you send from other accounts
  - when you send emails, put [CS583] in your subject header

Goal of the Course

- Design efficient algorithms and analyze their complexity
- **Analysis:** what are the computational resources needed ?
- time, storage, #processors, programs, communications

- **What is an algorithm:** Recipe to solve a problem
- Clear specification of the problem
- What is the input ? What is the output ?
- How long does it take, under particular circumstances ? (time)
- What are the memory requirements ? (space)

Examples of algorithms

- examples of algorithms
- sorting algorithms – everywhere
- routing, graph theoretic algorithms
- number theoretic algorithms, cryptography
- web search
- triangulation- graphics, optimization problems
- string matching (computational biology), cryptography - security

Shortest Paths

- Given a graph, find the shortest path in the graph connecting the start and goal vertices.
- What is a graph?
- How do you represent the graph?
- How do you formalize the problem?
- How do you solve the problem?

Shortest Paths

- What is the most naive way to solve the shortest path problem?
  - EX: a graph with only 4 nodes
  - How much time does your method take?
  - Can we do better?
  - How do we know our method is optimal? (i.e., no other methods can be more efficient.)

Shortest Paths

- Given a graph, find the shortest path in the graph that visits each vertex exactly once.
  - How do you formalize the problem?
  - How do you solve the problem?
  - How much time does your method take?
  - Can we do better?

Hard Problems

- We are able to solve many problems, but there are many other problems that we cannot solve efficiently
  - we can solve the shortest path between two vertices efficiently
  - but we cannot efficiently solve the shortest path problem that requires that path to visit each vertex exactly once

Course Topics

- Week 1: Algorithm Analysis (growth of functions)
- Week 2: Sorting & Order Statistics
- Week 3: Dynamic Programming
- Week 4: Greedy Algorithms
- Week 5: Graph Algorithms (basic graph search)
- Week 6: Minimum Spanning Tree
- Week 7: Single-Source Shortest Paths
- Week 8: All-Pairs Shortest Paths
- Week 9: Maximum Flow
- Week 10: Linear Programming
- Week 11: NP completeness

- **See updates on the course webpage**

Warning & Suggestions

- Please don't take this class if you
  - You do not have the mathematics and/or CS prerequisites
  - You are not able to make arrangements to come to GMU to take the exams on-site
  - You are working full-time and taking another graduate level computer science class
  - You are not able to spend a minimum of 9~12 hours a week outside of class reading the material and doing practice problem sets

Sorting

- Problem: Sort real numbers in **nondecreasing** order
- Input: A sequence of n numbers $\langle a_1, \ldots, a_n \rangle$
  Output:

A permutation $\langle a_1^{'}, \ldots, a_n^{'} \rangle$ s.t. $a_1^{'} \leq a_2^{'} \leq \ldots \leq a_n^{'}$

- Why do we need to sort?

Sorting

Sorting is important, so
there are **many** sorting algorithms

- Selection sort
- Insertion sort
- Library sort
- Shell sort
- Gnome sort
- Bubble sort
- Comb sort
- Binary tree sort
- Topological sort

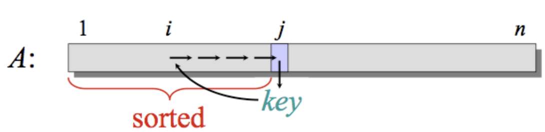
Sorting

- Algorithms in general
- We will be concerned with efficiency
- Memory requirements
- Independent of the computer speed

- How to design algorithms
- What is the easiest (or most naive) way to do sorting?
  - **EX**: sort 3,1,2,4
  - how efficient is your method?
  - We will look at two sorting algoritms

Insertion Sort

- If you ever sorted a deck of cards, you have done insertion sort
- If you don't remember, this is how you sort the cards:
  - you sort the card one by one
  - assuming the first i cards are sorted, now "sort" the $(i+1)$-th card
- **EX**: 4, 6, 1, 3, 7, 9, 2



Insertion Sort

```
for j ← 2 to n do
    Temp ← A[j]
    i ← j − 1
    while i > 0 and A[i] > Temp do
        A[i + 1] ← A[i]
        i ← i − 1
    end while
    A[i + 1] ← Temp
end for
```

- **EX**: 4, 6, 1, 3, 7, 9, 2

8 2 4 9 3 6

8 2 4 9 3 6

2 8 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

2 3 4 8 9 6

Analyze Insertion Sort

- Is it correct?
- How efficient/slow is insertion sort?
- Characterize running time as a function of input size
- Compute running time of each statement
- Sum up the running times

Insertion Sort

Cost times

```
for j ← 2 to n do
    Temp ← A[j]
    i ← j − 1
    while i > 0 and A[i] > Temp do
        A[i + 1] ← A[i]
        i ← i − 1
    end while
    A[i + 1] ← Temp
end for
```

- **EX**: 4, 6, 1, 3, 7, 9, 2

Insertion Sort

- Analysis

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 n$$

- Best case

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

- Worst case – input in the reverse order

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} j + c_5 \sum_{j=2}^{n} (j - 1) + c_6 \sum_{j=2}^{n} (j - 1) + c_7 n$$

$$T(n) = an^2 + bn + c$$

Algorithm analysis

- **Running time**
- depends on the size of the input (10 vs 100000)
- On the type of the input (sorted, partially sorted)
- Independent Speed of the computer
- **Kinds of analysis:**
- **Worst Case analysis** max time on any input
- **Average Case** T(n) = average time over all inputs
- of size n assuming some distribution
- **Best Case** T(n) = minimum time on some input
- can have bad algorithm which works only sometime it correct?

Algorithm analysis

- Use **pseudocode**
- description in the language independent way
- use proper indentation
- **Analysis of the running time of the algorithm**:
- How much time does it take ?
- (Cost per operation * number of operations)
- Choosing the basic operations and how long they take
- Too detailed, constant factors do not matter

Machine independent time

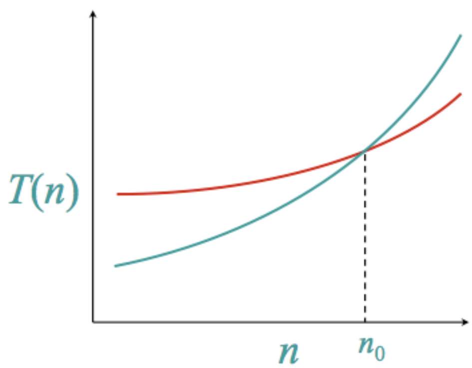
- We would like to ignore machine dependent constants
- characterize the running time T(n) as $n \to \infty$
- Asymptotic analysis - we introduce Big Theta Θ notation

Asymptotic Notation

- Big Θ

- **Definition**: $f(n)$ is in $\Theta(g(n))$ if $f(n)$ is bounded above and below by $g(n)$ (within constant multiple)
  - there exist positive constant c_1 and c_2 and non-negative integer n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for every $n \geq n_0$
- **Examples**:
  - $\frac{1}{2}n(n-1) \in \Theta(n^2)$
    - why?
  - $2n - 51 \in \Theta(n)$
    - why?

Asymptotic analysis



- Sometimes asymptotically slower algorithms work well for small inputs
- Overall we are interested in running time as n gets large

Algorithm analysis

- Example $3n^3 + 90n^2 + 5 = \Theta(n^3)$
- We learn some **design principles**
- Every problem is different so it is hard to come up with the general theory of design (but there few hints this course can offer)
- E.g. Some problems can be described recursively – their Solution can be devised by solving smaller sub-problems
- **Divide and Conquer:** design methodology
- Yields the description of running time in terms or recurrences

Recurrences

- Reminder: recurrence – system of equations that describes
- The function in terms of it's values on smaller inputs
- e.g. factorial

  for k = 1 Fact(k) = 1
  else Fact(k) = kFact(k-1) else

- **Merge Sort** (divide and conquer approach)
- DIVIDE the original sequence to two sequences of n/2
- CONQUER sort the two sequences recursively
- COMBINE combine the two sequences
- Example: 7 3 2 8   6 1 5 4

  2 3 7 8
  1 4 5 6   => 1 2 3 ….

Merge Sort

$\mathrm{Mergesort}(A, p, r)$ Sorts elements in subarray p …r

1: **if** $p < r$ **then**
2: $\quad q \leftarrow (p + r)/2$
3: $\quad \mathrm{Mergesort}(A, p, q)$
4: $\quad \mathrm{Mergesort}(A, q + 1, r)$
5: $\quad \mathrm{Merge}(A, p, q, r)$
6: **end if**

Key is the merge procedure (textbook for pseudocode)

Merge

- Example

2	3	6	7

1	4	5	8

Analyze Merge Sort

- How efficient/slow is merge sort?

```
if p < r then
    q ← (p + r)/2
    Mergesort(A, p, q)
    Mergesort(A, q + 1, r)
    Merge(A, p, q, r)
end if
```

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

- Which algorithm would you prefer Insertion or Merge Sort and why?
- Which one is faster? by how much?
- Which one requires more space? by how much?

Analyze Merge Sort

- Running time for Merge Sort – solution to the recurrence equation

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n)$$

- Expand the recurrence
- Works correct for any n, analysis is simpler for $n = 2^k$
- Divide step $\Theta(1)$
- Conquer step $2T\left(\frac{n}{2}\right)$
- Combine step $\Theta(n)$

$$T(n) = c \text{ if } n = 1$$
$$T(n) = 2T(n/2) + cn \text{ if } n > 1$$

Analyze Merge Sort

- Solution to the recurrence

$$T(n) = c \text{ if } n = 1$$
$$T(n) = 2T(n/2) + cn \text{ if } n > 1$$

- By expansion

Analyze Merge Sort

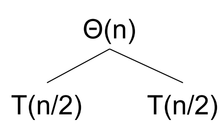
- Solution to the recurrence

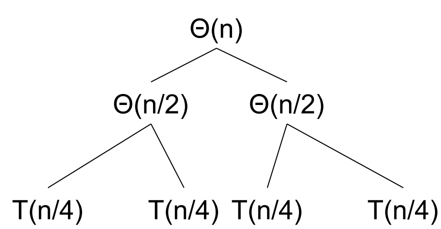
$$T(n) = c \text{ if } n = 1$$
$$T(n) = 2T(n/2) + cn \text{ if } n > 1$$

- Draw recurrence tree

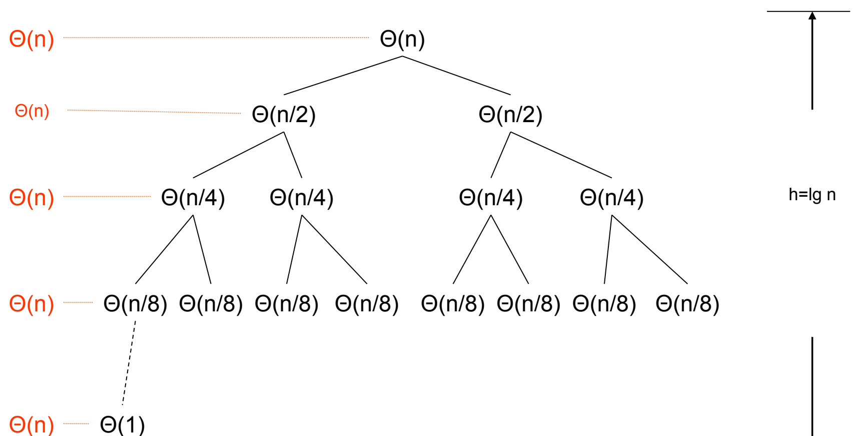
Recursion Tree

- T(n)=2T(n/2)+Θ(n)





Recursion Tree (cont)



(lg n)Θ(n)=Θ(n lg n). Θ(n lg n) grows more slowly than Θ(n^2). Mergesort asymtotically beats insertion sort in the worst case.

Divide and Conquer

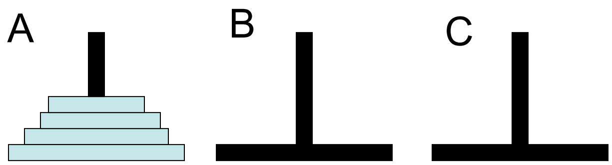
- Looking at the recursion tree you can compute the running time (solve the recurrence)
- DIVIDE and CONQUER in general

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{if } n > 1 \end{cases}$$

- Merge beats Insertion sort $\Theta(n \log n)$ grows more slowly then $\Theta(n^2)$

Towers of Hanoi

- Moves circles from A to B such that at no instances larger rings is atop smaller one

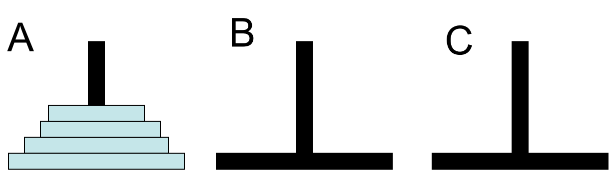


Recursive description of the problem:
1. Move n-1 rings from A-> C
2. Move largest ring from A-> B
3. Move all n-1 rings from C-> B

Towers of Hanoi

$$T(n) = 1 \text{ for } n = 1$$

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



Recursive description of the problem:
1. Move n-1 rings from A-> C
2. Move largest ring from A-> B
3. Move all n-1 rings from C-> B

Solution

- Solution to Tower of Hanoi by expansion

$$T(n) = 2T(n-1) + 1 = 2(2T(n-2) + 1) + 1$$
$$= 2^{n-1} + 2^{n-2} + ... + 2 + 1 = 2^n - 1$$

- Running time exponential in the size of input
- With 64 rings, if rings can be moved one ring per second. It would take 500 000 years to finish the task
- How to compare running time of different algorithms ? we need how to compute the running time within a constant factor

Order of growth of functions

- Enables asymptotic analysis
- How the algorithm behaves for large n
- Simple characterization of algorithm efficiency
- Enables comparative analysis of algorithms
- E.g.

$$3n^3 + 90n^2 + 5 = \Theta(n^3)$$

Order of Growth

- Theoretical analysis focuses on ``order of growth'' of an algorithm
- How the algorithm behaves as $n \to \infty$
- Some common order of growth

$$n, n^2, n^3, n^d, \log n, \log^* n, \log\log n, n\log n, n!, 2^n, 3^n, n^n, \sqrt{n}$$

Asymptotic Notation

- Big $O, \Omega . \Theta$
- upper, lower, tight bound (when input is sufficiently large and remain true when input is infinitely large)
- defines a set of similar functions

Big O

- **Definition**: $f(n)$ is in $O(g(n))$ if "order of growth of $f(n)$" $\leq$ "order of growth of $g(n)$" (within constant multiple)
  - there exist positive constant c and non-negative integer n_0 such that $f(n) \leq cg(n)$ for every $n \geq n_0$
- **Examples**:
  - $10n \in O(n^2)$
    * why?
  - $5n + 20 \in O(n)$
    * why?
  - $2n + 6 \notin O(\log n)$
    * why?
- g(n) is an upper bound

Big Θ

- **Definition**: $f(n)$ is in $\Theta(g(n))$ if $f(n)$ is bounded above and below by $g(n)$ (within constant multiple)
  - there exist positive constant c_1 and c_2 and non-negative integer n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for every $n \geq n_0$
- **Examples**:
  - $\frac{1}{2}n(n-1) \in \Theta(n^2)$
    * why?
  - $2n - 51 \in \Theta(n)$
    * why?
- g(n) is a tight bound

Big Ω

For a given function g(n) $\Omega(g(n)) = f(n)$
There exist constant c and n_0 such that:

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0$$

f(n) **grows at least as fast as** g(n); g(n) is asymptotically lower bound.

Example:

$$\sqrt{n} = \Omega(\log n); c = 1, n_0 = 16$$

Asymptotic Notation

- Asymptotic notation has been developed to provide a tool for studying order of growth
  - $O(g(n))$: a set of functions with the same or smaller order of growth as $g(n)$
    - $2n^2 - 5n + 1 \in O(n^2)$
    - $2^n + n^{100} - 2 \in O(n!)$
    - $2n + 6 \notin O(\log n)$
  - $\Omega(g(n))$: a set of functions with the same or larger order of growth as $g(n)$
    - $2n^2 - 5n + 1 \in \Omega(n^2)$
    - $2^n + n^{100} - 2 \notin \Omega(n!)$
    - $2n + 6 \in \Omega(\log n)$
  - $\Theta(g(n))$: a set of functions with the same order of growth as $g(n)$
    - $2n^2 - 5n + 1 \in \Theta(n^2)$
    - $2^n + n^{100} - 2 \notin \Theta(n!)$
    - $2n + 6 \notin \Theta(\log n)$

- Useful relationships:
  - Symmetry

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

- Transpose Symmetry

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

- Transitivity

$$\text{if } f(n) = O(g(n)) \text{ and } g(n) = O(h(n))$$
$$\text{then } f(n) = O(h(n))$$

Useful conventions

- Set in a formula represents anonymous function in the set

$$n^2 + O(n) = O(n^2)$$

$$f(n) = n^3 + O(n^2)$$

How functions grow

Input size	$33n$	$46\, n \lg n$	$13\, n^2$	$3.4\, n^3$	2^n
10	0.00033s	0.0015s	0.0013s	0.0034s	0.001s
100	0.003s	0.03s	0.13s	3.4s	$4*10^6$ s
1,000	0.033s	0.45s	13s	0.94 hr	
10,000	0.33s	6.1s	22min	39days	
100,000	3.3s	1.3min	1.5day	108 yr.	

Function Comparison

- Verify the notation by compare the order of growth

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & t(n) \text{ has the same order of growth as } g(n) \\ \infty & t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

- useful tools for computing limits
  - L'Hôpital's rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

  - Stirling's formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Bounding Functions

- non-recursive algorithms
- set up a sum for the number of times the basic operation is executed simplify the sum
- determine the order of growth (using asymptotic notation)
- Textbook appendix - basic formulas

1. $\sum_{1=1}^{n} 1 = 1 + 1 + \cdots + 1 = n \in \Theta(n)$
2. $\sum_{1=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{n^2}{2} \in \Theta(n^2)$
3. $\sum_{1=1}^{n} i^2 = 1 + 4 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3} \in \Theta(n^3)$
4. $\sum_{1=0}^{n} a^i = 1 + a^1 + \cdots + a^n = \frac{a^{n+1}-1}{a-1}, \forall a \neq 1, \in \Theta(a^n)$
5. $\sum a_i + b_i = \sum a_i + \sum b_i$
6. $\sum ca_i = c \sum a_i$
7. $\sum_{1=0}^{n} a_i = \sum_{1=0}^{m} a_i + \sum_{1=m+1}^{n} a_i$

Bounding Recursions

- Next: Techniques for Bounding Recurrences
  - Expansion
  - Recursion-tree
  - Substitution
  - Master Theorem