Welcome to EE602

Introduction to Algorithms

Yingfei Dong

1

Course info

- Instructor: Yingfei Dong
  - *yingfei@hawaii.edu*
  - Office: Holmes Hall 442
  - Office hours: one hour before the class
- Course Website at Laulima, https://laulima.hawaii.edu/portal
  - Did you receive emails (e.g., Syllabus) from Laulima?
- Time and Location: H388, **Mon/Wen 3:30-4:45**
  - Homeworks and Problems
  - programming assignments
  - Exams
  - Presentations on advanced algorithms ****

2

Prerequisites

- Data Structure, EE367 or ICS 311, or similar
- Please read Mathematical background in
  - Appendix A: methods for evaluating and bounding summations
  - Appendix B: definitions and notations for sets, relations, functions, graphs, and trees
  - Appendix C: elementary principles of counting
  - Appendix D: matrix operations and basic properties
- If you cannot do the exercises yourself, you will have a hard time to take this class
  - Please take EE367 or ICS311 then
  - I have post slides on the appendixes in Laulima

3

Prerequisites

- programming experience
  - Install Java 1.8
  - Install Eclipse
- mathematical proofs:
  - proofs by mathematical induction
  - elementary calculus
- Discrete math

4

Reading Assignment

- Week 1 and Week 2
  - Chapter 1
  - Chapter 2
  - Please read Appendix A, B, C, and D
- **VideoLectures.net (recommended)**
  - http://videolectures.net/mit6046jf05_introduction_algorithms
  - + Powerpoint slides in parallel
  - + Topical index to jump to segments of video
- ***MIT Open Courseware Site***
  - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/video-lectures/

5

What is course about?

- "Before there were computers, there were algorithms; now there are more computers, there are even more algorithms. Algorithms lie at the heart of computing"
- What is an algorithm?
  - Please give me your definition

6

What is an Algorithm?

- An Algorithm → a well-defined computational procedure
  - take a set of values, as input
  - produce a set of values as output
- to solve a well-specified computational problem
- Here is how to formulate a sorting problem

  Input: A sequence of numbers $\{a_1, a_2, \ldots, a_n\}$

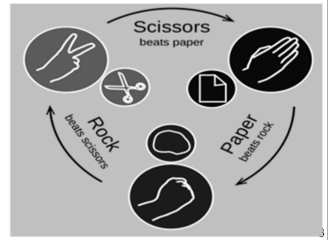
  Output: a permutation of input sequence such that $a_1 < a_2 < \ldots < a_n$

  Instance Input of a problem {2, 5, 9, 6, 4}

7

How to play rock-paper-scissor

- Design an algorithm for a player
- How to optimize it?



Course Focus

- The theoretical study of design and analysis of computer algorithms
  - Not about programming, not about math
  - Design: design correct algorithms which minimize cost
    - Efficiency is the design criterion
  - Analysis: predict the cost of an algorithm in terms of resource and performance

What are the main issues of algorithm design?

9

Course Overview

- Focus: solve application problems with algorithms
- What are the main differences between algorithms and data structures?
  - Algorithm: method for solving a problem
  - Data structure: method to store information
- Textbook: ***Introduction to Algorithms***, Cormen, Leiserson, Rivest, Stein (often referred as CLRS)
  - 3rd edition
  - *An excellent reference you should own*
  - *One of the most cited books in CS*

What are the Basic Goals of Designing Algorithms?

- Basic goals for an algorithm
  - always correct
  - always terminates
- More, we also care about *performance*
  - Tradeoffs between what is possible and what is impossible
  - We usually have a deadline
    - E.g., Computing 24-hour weather forecast within 20 hours

11

Coursework and Grading

- Homework and Programming Assignment: 40%
  - Will be posted in Laulima on-line
  - You are encouraged to discuss your homework with your partner, but you must write your answer alone.
  - Type and email me, or give me a printed copy
    - Prefer a **non-handwriting** copy
- Mid-term: 20%
- Final: 35%
  - *Closed book with one letter-size (8X11) cheatsheet*
- Participation in class discussion: 5%
- Presentation: (extra points 5%)

12

Attendance

- Attendance: Daily attendance is strongly encouraged
  - Any student missing a lesson is responsible for any material covered in class during his or her absence
  - I will help, but no makeup class
- Basic Requirements
  - Please be on time: being polite to me and everyone here
  - Turn off your laptop/teblet and silence your phone
    - Every occurrence → bring treats for everyone in the next class
    - Or, 1% of your final grade
- **Announcements will be sent to you via Laulima**
- Notes will be at Laulima after each classes

13

Main Topics in 17 weeks: a lot!

- You may have seen some before→ Now we need analyze them
- Part.1 Fundamentals of algorithms: growth functions, divide-and-conquer, merge-sort, recurrence.
- Part.2 Sorting and Order Statistics: heapsort, quicksort, order statistics.
- Part.3 Review of Data Structures: basic data structure, hash tables, binary search tree, augmenting data structure.
- Part.4 Advanced Design and Analysis Techniques: dynamic programming, greedy algorithms, amortized analysis.
- Part.5 Graph Algorithms. Breadth/depth-first search, minimum spanning tree, shortest path.
- Part.6 Multithreaded algorithms.
- Part.7 NP-complete problems and approximation algorithms.

14

Tentative Timeline for your reading assignment

- Week 1 , 2 and 3
  - The role of computing (Ch.1)
  - Analysis of Algorithms: Insertion Sort, Merge Sort (Ch.2)
  - Growth functions, Asymptotic Notations (Ch 3)
  - Divide and Conquer, Recurrences (Ch 4)
  - Probabilistic Analysis (Ch 5)
- Week 4 and 5
  - Heapsort (Chapter 6)
  - Quicksort (Chapter 7)
  - Linear-time Sorting, Lower Bounds, Counting Sort, Radix Sort. (Ch.8), Order Statistics (Ch.9)

15

Tentative Timeline

- Week 6, 7, and 8
  - Elementary data structure (Ch.10) (Self study)
  - Hash Tables (Ch.11)
  - Binary Search Trees (Ch.12)
  - Augmenting Data Structure (Ch.14)
  - Midterm →
- Week 9 and 10
  - Dynamic Programming(Ch.15)
  - Greedy Algorithms (Ch.16)
  - Amortized Analysis (Ch.17)
- Week 11 and 12
  - Elementary graph algorithms, Graph representation. DFS. BFS. Topological Sort, strongly connected components (Ch.22)
  - Minimum Spanning Tree. Greedy algorithms. Prim's algorithm, Kruskal algorithm. (Ch.23)

16

Tentative Timeline

- Week 13 and 14
  - Single source shortest path. Dijkstra's Algorithm, Bellman-Ford, Shortest Paths in Dags (Ch.24)
  - All-Pairs Shortest Paths. Dynamic programming, Matrix Multiplication. Floyd-Warshall Algorithm. Jonson's Algorithm. (Ch.25)
- Week 15, 16 and 17
  - Mutlithreaded algorithms (Ch.27)
  - NP-completeness and approximation algorithms (Ch.34 and 35)

17

Let me know a little about you

Your Name: ___________

- Something helps me pronounce your name correctly
- Let me know your experiences in algorithms design
  - EE367 Data Structrue, ICS 311 Algorithms
  - Other related classes
- Programming in C? Java? Other programming languages?
- Use of UNIX or Linux, Virtual Machine, Cloud Computing
  - How much experiences with Linux?

18

Most learning occurs outside classroom. Find yourself partners

- Introduce yourself
  - Your name and major
  - Your current research interests
  - Your experience with the course-related topic, e.g. previous classes, projects
  - Something you like to share:
    - Where you are from, your favorites (sports, drinks, foods, surfing spots, etc.)

19

Your suggestions are welcome any time

Please send me an email after the first class

- PLEASE put "EE602" or "ee602" in the subject line of your email. Thanks!
  - So your email will be put into proper folder
- Please briefly state your expectation of this course, suggestions, comments.
- Whenever you have any suggestions, please talk to me or send me an email.

20

Introduction to Algorithms (Ch.1)

- What is an algorithm?
  - the definition
- Why do we study algorithms?
  - Motivations?

21

What is an Algorithm?

- An Algorithm → a well-defined computational procedure
  - take a set of values, as input
  - produce a set of values as output
- to solve a well-specified computational problem
- Here is how to formulate a sorting problem

Input: A sequence of numbers $\{a_1, a_2, \ldots, a_n\}$
Output: a permutation of input sequence such that $a_1 < a_2 < \ldots < a_n$
Instance Input of a problem {2, 5, 9, 6, 4}
Instance Output of a problem {2, 4, 5, 6, 9}

22

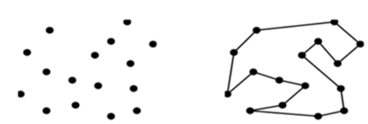
Basics of Algorithms

- An algorithm is said to be correct, if it halts with a correct output for every instance
  - Convergence → stop gradually
- An algorithm can be specified
  - In English → pseudo-code
  - as a computer program → word count program (wc)
  - a hardware design → TPM
  - The only requirement is that the specification must provide a precise description of the computational procedure to be followed

23

Hard Problems

- We focus on efficient ALs in this class
- But some problems which we do NOT know any efficient solutions → NP-complete problems
  - **NP: non-deterministic polynomial**
- E.g., Traveling-salesman problem, **knapsack**, ...
  - **Input:** Distance-weighted graph *G*
  - **Problem:** Find the shortest route to visit all of the vertices exactly once



24

NP-complete Problems

- Three interesting facts about NP-complete
  1. No existing efficient algorithms to solve them
  2. If we find one, then exist one for all of them
  3. Some of them similar to solved problems
- Under certain assumptions, efficient algorithms might give a near-optimum solution
  - these are called approximation algorithms

25

Parallelism improve performance, not solving NP complete problems

- Computer performance increase: two key methods
  - Hierarchy: Cache, memory, I/O system, etc.
  - Parallel
    - Co-processors, Multicore
    - Data Center, Cloud computing
    - → hardware improvement does not help with NP-complete
- Map/Reduce: Simplified Data Processing on Large Clusters
  - http://static.googleusercontent.com/external_content/untrusted_dlcp/labs.google.com/en/us/papers/mapreduce-osdi04.pdf
  - Large-Scale Data Processing
  - use 1000s of CPUs but don't want to ***manage*** things
  - 1st extra credit assignment → who like to make a 30-minute presentation on this?

26

Why do we study Algorithms

- Suppose computers were infinitely fast and computer memory was free
  - never true in reality
- Do we still have reasons to study algorithms?
- YES!!!
  - We still need to demonstrate our solution terminates with a correct answer
- Algorithms as technology
  - Resources are always limited → Efficiency is the center of algorithms

27

Why study algorithms? Tech. Com.

- Akamai, 1998 (old story: $300 per share in 1998)
  - Proxy and cache web contents, MIT Algorithm group
- Google, 1998
  - PageRank, Larry Page, Sergey Brin
  - MapReduce
- Baidu, 2000
  - RankDex site-scoring algorithm for search engines
  - NYU Buffalo, InfoSeek, Yanhong Li
- Cadence Design Systems, 1988
  - electronic design automation
  - Cadence claimed Avanti stole Cadence code !
  - Business Week: *"The Avanti case is probably the most dramatic tale of white-collar crime in the history of Silicon Valley"*
- Match.com, 1993; eharmony.com, chemistry.com
  - 22 dimension matching! Positive and Negative

28

The List goes on: Why study algorithms?

Their impact is broad and far-reaching

- Internet. Web search, packet routing, distri. file sharing
- Biology. Human genome project, protein folding.
- Computers. Circuit layout, file system, compilers.
- Computer graphics. Hollywood movies, video games, 3-D
- Security. Cell phones, e-commerce, voting machines.
- Multimedia. CD player, DVD, MP3/4, JPG, DivX, HDTV.
- Transportation. Airline crew scheduling, map routing.
- Physics. N-body simulation, particle collision simulation.

29

Course Goals

- **Learn critical thinking for problem solving**
  - **How to think?**
    - **Learn to design, using well known methods**
  - **What can we try (e.g., to find a min in a set)?**
    - **Basic technique: brute-force, divide-and-conquer, dynamic programming, greedy**
- **Implementing algorithms efficiently & correctly**
  - **Efficiency → Analyzing time complexity**
  - **Correctness → Arguing correctness**

30

Consider an algorithm, Main questions we have to answer

- **What is the <u>Problem</u>?**
  - **Find the position of key x in a <u>sorted</u> array**
- **What is our <u>Strategy</u>?**
  - **Binary search**
- **<u>Efficiency</u>: how to achieve?**
  - **Log(n)**
- **Analysis of <u>correctness</u>**
  - **<u>Prove it is correct</u>**

31

Importance of Algorithm Efficiency

- <u>Time → CPU time</u>
- <u>Storage → main memory</u>
- <u>I/O → new criterion in our current systems</u>
- Examples
  - Sequential search vs. Binary search
    Basic operation: <u>comparison</u>
    Number of comparisons is grown in different rate
  - n-th Fibonacci sequence
    <u>Recursive</u> versus <u>Iterative</u>

32

Example: search strategy

❑ <u>Sequential search vs. Binary search</u>

<u>Problem:</u> determine whether x is in the sorted array S of n keys

⦿ <u>Inputs:</u>
- A positive integer, n
- A sorted (*non-decreasing order*) array of keys, S, indexed from 1 to n
- A key, x

⦿ <u>Output:</u> the location of x in S (0 if x is not in S)

33

Example: <u>Sequential</u> search

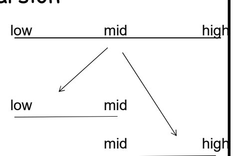
Basic operation: <u>comparison</u>

```
int Seq_search(int n, int S[], int x)
{
  int loc=0;
  while  ( loc<=n && S[loc] != x)    // comparsion
            loc++;  // due to the non-decreasing order
  if (loc > n)
      return 0;
  else
      return loc;
}
```

34

Example: Binary search

```
int Bin_search(int n, const int S[], int x)
{
  int low, high, mid, location=0;
  low = 1; high =n;
  while ( low <= high && location ==0) {
    mid = floor( (low+high)/2 );
    if (x == S[mid])       // comparsion
         location = mid;
    else   if (x < S[mid] )
                 high = mid -1;
           else   low = mid +1;
  }
  return location;
}
```



35

Example: number of comparisons

❑ Sequential search: <u>worst case</u> → search all n keys

n = 32 128 1024 1,048,576

❑ Binary search: <u>worst case</u> → at most lg(n)+1

lg(n) + 1 6 8 11 21

Eg: when n = 32

S[1],…, S[16],…, S[24], S[28], S[30], S[31], S[32]
(1st) (2nd) (3rd) (4th) (5th) (6th)

36

Analysis of Algorithm Complexity

- Two main characters
  - Input size: n
  - Basic operation: e.g., comparison
- Time complexity
  - T(n) : Single-step time complexity of size n
  - W(n): Worst-case time complexity
  - A(n): Average-case time complexity
  - B(n): Best-case time complexity
- T(n) example
  - Add n array members together T(n) = n-1
  - Matrix multiplication n*n*n
  - Exchange sort n(n-1)/2

37

Math preparation

- Please read Appendix A, B, C, and D
  - You are supposed to know them
- Induction
- Logarithm
- Sets
- Permutation and combination
- Limits
- Series
- Asymptotic growth functions and recurrence
- Probability theory

38

Programming preparation

- Data structure
- C, Java
- Eclipse Develop Environment, http://www.eclipse.org/
  - For Java, C, and many other languages
  - **Install Eclipse on Linux**
  - **Install Eclipse on Windows**
    - **http://www.cs.dartmouth.edu/~cs5/install/eclipse-win/index.html**
  - http://www.java.com/en/download/win10.jsp
  - Eclipse And Java: Free Video Tutorials (16 flash videos)
    - **http://eclipsetutorial.sourceforge.net/totalbeginner.html**
    - **Companion Tutorial Document http://eclipsetutorial.sourceforge.net/Total_Beginner_Companion_Document.pdf**
- Java Code of this book is loaded in the "Resources" Section
  - You can create a Java project in Eclipse and import

39

What are Commonly used algorithms?

- **Search (sequential, binary)**
- **Sort (mergesort, heapsort, quicksort, etc.)**
- **Traversal algorithms (breadth, depth, etc.)**
- **Shortest path (Floyd, Dijkstra)**
- **Spanning tree (Prim, Kruskal)**
- **Knapsack**
- **Traveling salesman**
- **Bin packing**

40

Common Design Methods or Strategies

- **Divide-and-conquer**
- **Greedy**
- **Dynamic programming**
- **Backtracking**
  - **depth-first search with pruning**
  - **an improvement over brute-force**
- **Branch-and-bound**
  - breadth-first search with pruning

41

Using critical thinking for problem solving

- **Considering different approaches for solving a problem**
  - **dynamic programming or greedy algorithm**
  - **Analyzing the merits of each**
- **Considering different implementations for a chosen approach**
  - **E.g., Prim's algorithm**
  - **Analyzing the merits of different implementations**

42

Quick Review: Induction

- What is induction?
- Suppose
  - Statement S(k) is true, for fixed constant k
    - Often k = 0
  - If we have S(n) → S(n+1) , for all n ≥ k
  - Then, S(n) is true, for all n ≥ k

43

Proof By Induction

- Claim: formula S(n) is true, for all n ≥ k
- Basis:
  - Show formula S(n) is true when n = k
- Inductive hypothesis:
  - Assume formula S(n) is true, for an arbitrary n > k
- Step:
  - Show that formula S(n+1) is true, for all n > k

44

Induction Example: Gaussian Closed Form

- Prove 1 + 2 + 3 + … + n = n(n+1) / 2
  - Basis
    - If n = 0, then 0 = 0(0+1) / 2
  - Inductive hypothesis
    - Assume 1 + 2 + 3 + … + n = n(n+1) / 2
  - Step: show true for (n+1)
    1 + 2 + … + n + (n+1) = (1 + 2 + …+ n) + (n+1)
    = n(n+1)/2 + (n+1) = [n(n+1) + 2(n+1)]/2
    = (n+1)(n+2)/2
    = (n+1)(n+1 + 1) / 2

45

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + … + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: n=0, show that $a^0 = (a^{0+1} - 1)/(a - 1)$
    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis: S(n) is true
    - Assume $a^0 + a^1 + … + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show S(n+1) is true)
    $a^0 + a^1 + … + a^{n+1} = (a^0 + a^1 + … + a^n) + a^{n+1}$
    $= (a^{n+1} - 1)/(a - 1) + a^{n+1}$
    $= (a^{(n+1)+1} - 1)/(a - 1)$

46

Various Induction Methods

- We've been using weak induction
- Strong induction also holds
  - Basis: show S(0)
  - Hypothesis: assume S(k) holds for arbitrary k ≤ n
    - makes the inductive step easier to prove by using a stronger hypothesis
  - Step: Show S(n+1) follows
- Another variation:
  - Basis: show S(0), S(1)
  - Hypothesis: assume S(n) and S(n+1) are true
  - Step: show S(n+2) follows

47

Analysis of Algorithms (Sec 2.2)

- We perform analysis with respect to a computational model → a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - In reality, they may take different time
    - Except function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

48

Input Size: first parameter

- To find Time- and space-complexity
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends on
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - …

49

Running Time of basic operations

- Number of primitive steps that are executed
  - **Except for time of executing a function call**
  - **most statements roughly require the same amount of time**
    - **y = m * x + b**
    - **c = 5 / 9 * (t - 32)**
    - **z = f(x) + g(y)**
- We can be more exact if need

50

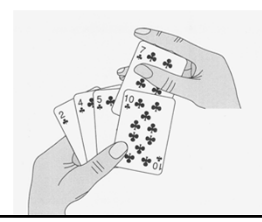
Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is "average"?
    - Random (equally likely) inputs
    - Real-life inputs

51

An Example: Insertion Sort (Sec. 2.1)

```
InsertionSort(A) {
  for j = 2 to n {  // n → A.size
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]  // shift one position
          i = i - 1
     }
     A[i+1] = key
  }
}
```



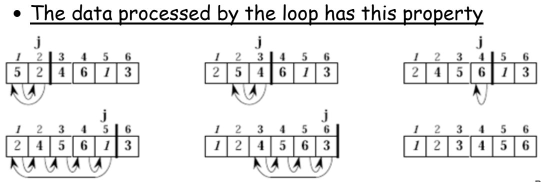
What is Sorting "in place"?

52

Correctness: Loop Invariants

- At the start of each iteration of the 'for' loop, subarray A[1 .. (j-1)] consists of the elements originally in A[1..j-1] but in sorted order: (j-1) cards already sorted; insert a new card
- We must show three things about loop invariants
  - Initialization
  - Maintenance
  - Termination

  use "induction" to prove the property
  -Base case
  -Induction step
  - The data processed by the loop has this property



53

An Example: Insertion Sort

30	10	40	20
1	2	3	4

j = ∅   i = ∅   key = ∅
A[i] = ∅   A[i+1] = ∅

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
  }
}
```

54

An Example: Insertion Sort

30	10	40	20
1	2	3	4

j = 2 i = 1 key = 10
A[i] = 30 A[i+1] = 10

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

55

An Example: Insertion Sort

30	30	40	20
1	2	3	4

j = 2 i = 1 key = 10
A[i] = 30 A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

56

An Example: Insertion Sort

30	30	40	20
1	2	3	4

j = 2 i = 0 key = 10
A[i] = 30 A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

57

An Example: Insertion Sort

30	30	40	20
1	2	3	4

j = 2 i = 0 key = 10
A[i] = ∅ A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

58

An Example: Insertion Sort

30	30	40	20
1	2	3	4

j = 2 i = 0 key = 10
A[i] = ∅ A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

59

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 2 i= 0 key = 10
A[i] = ∅ A[i+1] = 10

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

60

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 3 i = 0 key = 10
A[i] = ∅ A[i+1] = 10

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
     }
     A[i+1] = key
  }
}
```

61

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 3 i = 0 key = 40
A[i] = ∅ A[i+1] = 10

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
     }
     A[i+1] = key
  }
}
```

62

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 3 i = 0 key = 40
A[i] = ∅ A[i+1] = 10

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
     }
     A[i+1] = key
  }
}
```

63

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 3 i = 2 key = 40
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
     }
     A[i+1] = key
  }
}
```

64

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 3 i = 2 key = 40
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
     }
     A[i+1] = key
  }
}
```

65

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 3 i = 2 key = 40
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
     }
     A[i+1] = key
  }
}
```

66

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 4 i = 2 key = 40
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
   for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
   }
}
```

67

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 4 i = 2 key = 20
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
   for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
   }
}
```

68

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 4 i = 2 key = 20
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
   for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
   }
}
```

69

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 4 i = 3 key = 20
A[i] = 40 A[i+1] = 20

```
InsertionSort(A, n) {
   for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
   }
}
```

70

An Example: Insertion Sort

10	30	40	20
1	2	3	4

j = 4 i = 3 key = 20
A[i] = 40 A[i+1] = 20

```
InsertionSort(A, n) {
   for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
   }
}
```

71

An Example: Insertion Sort

10	30	40	40
1	2	3	4

j = 4 i = 3 key = 20
A[i] = 40 A[i+1] = 40

```
InsertionSort(A, n) {
   for j = 2 to n {
      key = A[j]
      i = j - 1
      while (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
      }
      A[i+1] = key
   }
}
```

72

An Example: Insertion Sort

10	30	40	40
1	2	3	4

j = 4 i = 3 key = 20
A[i] = 40 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

73

An Example: Insertion Sort

10	30	40	40
1	2	3	4

j = 4 i = 3 key = 20
A[i] = 40 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

74

An Example: Insertion Sort

10	30	40	40
1	2	3	4

j = 4 i = 2 key = 20
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

75

An Example: Insertion Sort

10	30	40	40
1	2	3	4

j = 4 i = 2 key = 20
A[i] = 30 A[i+1] = 40

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

76

An Example: Insertion Sort

10	30	30	40
1	2	3	4

j = 4 i = 2 key = 20
A[i] = 30 A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

77

An Example: Insertion Sort

10	30	30	40
1	2	3	4

j = 4 i = 2 key = 20
A[i] = 30 A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
           A[i+1] = A[i]
           i = i - 1
     }
     A[i+1] = key
  }
}
```

78

An Example: Insertion Sort

10	30	30	40
1	2	3	4

j = 4 i = 1 key = 20
A[i] = 10 A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
     }
     A[i+1] = key
  }
}
```

79

An Example: Insertion Sort

10	30	30	40
1	2	3	4

j = 4 i = 1 key = 20
A[i] = 10 A[i+1] = 30

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
     }
     A[i+1] = key
  }
}
```

80

An Example: Insertion Sort

10	20	30	40
1	2	3	4

j = 4 i = 1 key = 20
A[i] = 10 A[i+1] = 20

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
     }
     A[i+1] = key
  }
}
```

81

An Example: Insertion Sort

10	20	30	40
1	2	3	4

j = 4 i = 1 key = 20
A[i] = 10 A[i+1] = 20

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
     }
     A[i+1] = key
  }
}
```

Done!

82

Sorting Algorithm Animations

- http://www.sorting-algorithms.com/
- Algorithm: Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3
- Initial Condition: Random · Nearly Sorted · Reversed · Few Unique

83

Again: Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
     }
     A[i+1] = key
  }
}
```

What is the precondition for this loop?

84

Insertion Sort

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j]
     i = j - 1
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
     }
     A[i+1] = key
  }
}
```

How many times will this loop execute?

What is insertion sort's worst-case time?

85

Statement	**Effort**
`InsertionSort(A, n) {`	
`for j = 2 to n {`	$c_1 n$
`key = A[j]`	$c_2(n-1)$
`i = j - 1`	$c_3(n-1)$
`while (i>0) and (A[i] > key) {`	$c_4 T$
`A[i+1] = A[i]`	$c_5(T-(n-1))$
`i = i - 1`	$c_6(T-(n-1))$
`}`	0
`A[i+1] = key`	$c_7(n-1)$
`}`	0
`}`	

$T = (t_2 + t_3 + \dots + t_n)$, where t_j is the number of 'while' expression evaluations for the **j^{th}** for loop iteration

86

Running time

- The running time depends on the input:
  - an already sorted sequence is easier to sort
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
  - $T_A(n)$ = time of *A* on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

87

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$
  $= c_8 T + c_9 n + c_{10}$
- What can T(n) be?
  - Best case -- inner loop body never executed
    - $t_i = 1 \rightarrow T(n) = cn + b$, a linear function
  - Worst case -- inner loop body executed for all previous elements
    - $t_i = i \rightarrow T(n) = c_{11}n^2 + c_{13}n + c_{12}$, a quadratic function
  - Average case
    - ???

88

Insertion sort analysis

Worst case: Input reverse sorted.

$$T(n) = \sum_{i=2}^{n} \Theta(i) = \Theta(n^2)$$ [arithmetic series]

Average case: All permutations equally likely.

$$T(n) = \sum_{i=2}^{n} \Theta(i/2) = \Theta(n^2)$$

-- *Is insertion sort a fast sorting algorithm?*
- Moderately so, for small *n*.
- Not at all, for large *n*.

89

Example 2: Integer Multiplication

- Let X = A B and Y = C D
  - where A,B,C and D are n/2 bit integers
- Simple Method: $X*Y = (2^{n/2}A + B)(2^{n/2}C + D)$
  - **→ four n/2-bit multiplications**
- Running Time Recurrence

  $T(n) < 4T(n/2) + 100n$
- Solution $T(n) = \theta(n^2)$

90

Better Integer Multiplication

- Let X = | A | B | and Y = | C | D |
  - where A, B, C and D are n/2 bit integers
- Karatsuba **algorithm**
  - → **three n/2-bit multiplications**
  - $X*Y = (2^{n/2}+2^n)AC+2^{n/2}(A-B)(C-D) + (2^{n/2}+1) BD$
- Running Time Recurrence
  - $T(n) < 3T(n/2) + 100n$
- Solution: $\theta(n) = O(n^{\log 3})$

91

Analysis → *Order of growth*

- Simplifications
  - Ignore actual and abstract statement costs
  - *Order of growth* is the interesting measure:
    - Highest-order term is what counts
      - **→ we are doing asymptotic analysis**
      - **As the input size grows larger, it is the high order term that dominates**
      - **Look at *growth* of *T*(*n*) as $n \to \infty$**

92

Asymptotic Performance (Ch.3)

- In this course, we care most about asymptotic performance
  - How does the algorithm behave as the problem size gets very large?
    - Running time
    - Memory, storage requirements
    - Bandwidth, **power requirements, logic gates,** etc.
    - **Power consumption:** CPU frequency, data center, etc
      - **2% of world power consumption in 2010 !!!**

93

Asymptotic Notation (Sec 3.1)

- You should have known (big-O) notation:
  - *What does O(n) running time mean? $O(n^2)$? $O(n \lg n)$?*
- Let us define this notation more formally and completely

94

Upper Bound Notation (Sec 3.1)

- We say InsertionSort's run time is $O(n^2)$
  - Properly we should say run time is ***in* O(n²)**
  - Read O as "Big-O" (you'll also hear it as "order")
- In general, a function f(n) is O(g(n))
  - if there exist positive constants c and n_0
  - such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$
- Formally
  - $O(g(n)) = \{ f(n): \exists$ positive constants c and n_0 such that $f(n) \le c \cdot g(n), \forall\ n \ge n_0 \}$

95

Insertion Sort is O(n²)

- Proof
  - Suppose runtime is $an^2 + bn + c$
    - If any of a, b, and c are less than 0 replace the constant with its absolute value
  - $an^2 + bn + c \le (a+b+c)n^2 + (a+b+c)n + (a+b+c)$
  - $\le 3(a+b+c)n^2$ for $n \ge 1$
  - Let $c' = 3(a+b+c)$, and let $n_0 = 1$
- Question
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $O(n)$?

96

Big-O Fact

- A polynomial of degree k is $O(n^k)$
- Proof:
  - Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \dots + b_1 n + b_0$
    - Let $a_i = | b_i |$
  - $f(n) \le a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$

$$\le n^k \sum a_i \frac{n^i}{n^k} \le n^k \sum a_i \le cn^k$$

97

Lower Bound Notation

- We say InsertionSort's run time is $\Omega(n)$, Omega
- a function f(n) is $\Omega(g(n))$
  - if $\exists$ positive constants c and n_0
  - such that $0 \le c \cdot g(n) \le f(n)$, $\forall$ $n \ge n_0$
- Proof:
  - Suppose run time is (an + b)
    - Assume a and b are positive (what if b is negative?)
  - an $\le$ an + b

98

Asymptotic Tight Bound, Θ Theta

- A function f(n) is $\Theta(g(n))$
  - if $\exists$ positive constants c_1, c_2, and n_0
  - such that $c_1\, g(n) \le f(n) \le c_2\, g(n)$, $\forall$ $n \ge n_0$
- Theorem
  - f(n) is $\Theta(g(n))$, i.f.f. f(n) is both $O(g(n))$ and $\Omega(g(n))$

99

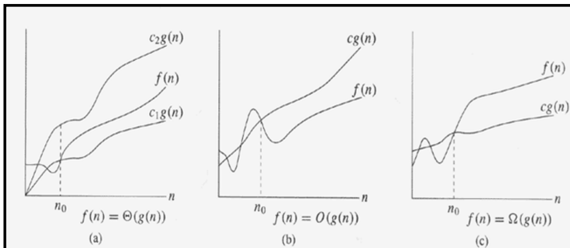


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

100

Reading Assignment and Homework

- Week 1
  - Read Ch.1, Ch.2, Ch3
  - Appendix A, B, and C
- Week 2
  - Read Ch.4
- Homework 1, due in the first class in Week 3
  - Type and email me before the class

101

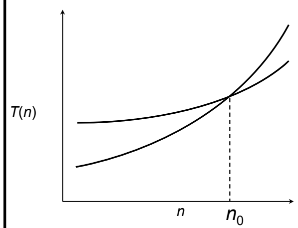
3.2 Standard Notations & Common Functions

- Monotonicity
  - Monotonically increase: if $m \le n$, then $f(m) \le f(n)$
  - Monotonically decrease: if $m \le n$, then $f(m) \ge f(n)$
  - Strictly increase: if $m < n$, then $f(m) < f(n)$
  - Strictly decrease: if $m < n$, then $f(m) > f(n)$
- Floors and Ceilings
  - $X-1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x+1$
- Modular arithmetic: $(a \bmod n) = a - n\lfloor a/n \rfloor$
- Polynomials: $p(n) = \Sigma\, a_i\, n^i$
- Exponentials: $e(n)= n^i$
- Logarithms: log(n)
- Factorials: n!
- Functional Iteration: $f^{(i)}(n)=n$, if $i=0$; $f^{(i)}(n)= f(f^{(i-1)}(n))$, if $i>0$
- Fibonacci Numbers: F(0)=0, F(1)=1, F(i)=F(i-1)+F(i-2)

102

Asymptotic performance

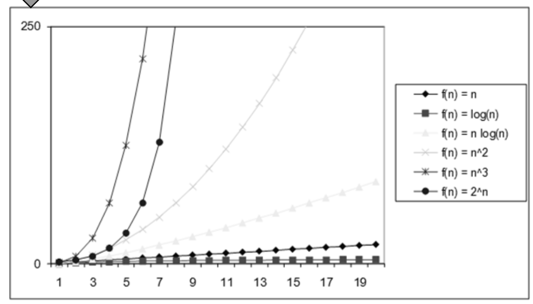
When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- Asymptotic analysis is a useful tool to help to structure our thinking toward better algorithm
- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing

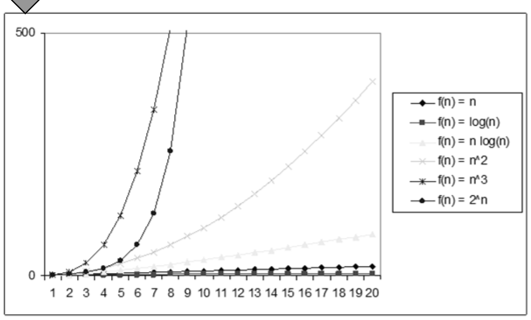
103

Practical Complexity: in this range < 250



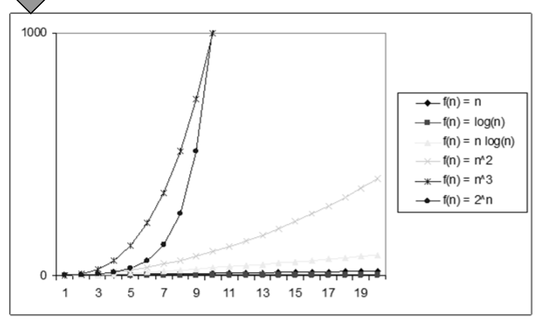
104

Practical Complexity: in this range < 500



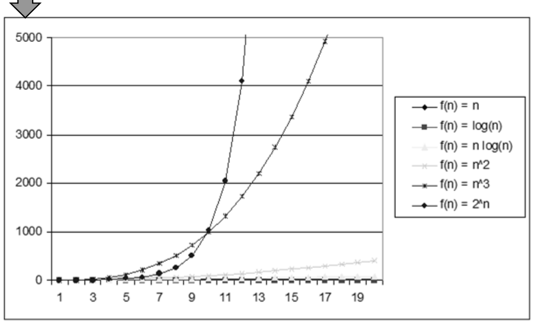
105

Practical Complexity: in this range <1000



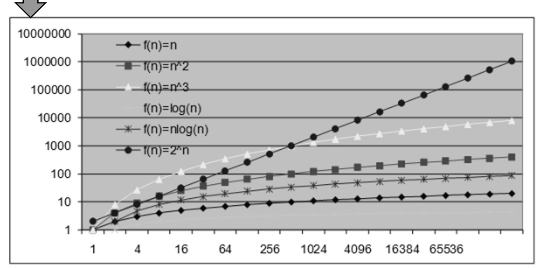
106

Practical Complexity: in this range <5000



107

Practical Complexity, 10^8



108

Other Asymptotic Notations

- A function f(n) is o(g(n))
  - if ∃ positive constants *c* and n_0
  - such that f(n) < c g(n), ∀ n ≥ n_0
- A function f(n) is ω(g(n))
  - if ∃ positive constants *c* and n_0
  - such that c g(n) < f(n), ∀ n ≥ n_0
- Intuitively,
  - o() is like <
  - ω() is like >
  - Θ() is like =
  - O() is like ≤
  - Ω() is like ≥

109

Up Next

- Divide-and-Conquer
- Solving recurrences
  - Substitution method
  - Iteration method
  - Master theorem

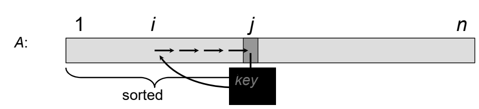
110

111

Insertion sort

"pseudocode"

```
INSERTION-SORT (A, n)    ⊳ A[1 . . n]
  for j ← 2 to n
      do key ← A[ j]
         i ← j – 1
         while i > 0 and A[i] > key
             do A[i+1] ← A[i]
                i ← i – 1
         A[i+1] = key
```



L1.112

Example of insertion sort

8 2 4 9 3 6

L1.113

Example of insertion sort

8 2 4 9 3 6

L1.114

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

L1.115

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

L1.116

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

L1.117

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

L1.118

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

L1.119

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

L1.120

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

L1.121

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

L1.122

Example of insertion sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

L1.123