Analysis of Algorithms 1
Istanbul Technical University
Computer Eng. Dept.

Introduction

Course slides from Leiserson's MIT class have been used in preparation of these slides.

Analysis of Algorithms 1, Dr. Çataltepe & Dr. Ekenel, Dept. of Computer Engineering, ITU

Outline

- About this course

- Algorithms and data structures

- Insertion Sort

  – How to Analyze an Algorithm

    - Correctness (Loop Invariants)

    - Running Time for a Specific Input Size

      – Worst Case Analysis
      – Average Case Analysis

Course Information

Lecturers:

Dr. Zehra Çataltepe

Machine Learning

cataltepe@itu.edu.tr

http://web.itu.edu.tr/~cataltepe/resume.htm

Dr. Hazım Kemal Ekenel

Computer Vision

ekenel@itu.edu.tr

http://web.itu.edu.tr/ekenel/

Week 1

Course Information

Textbook: "Introduction to Algorithms", 2nd edition, Cormen et al, MIT Press.

Course webpage:

http://web.itu.edu.tr/~cataltepe/AoAI/

Ninova: http://ninova.itu.edu.tr/Ders/3960

- Course slides
- Homework assignments
- Announcements

  Make sure you are added to Ninova site and check your ITU e-mail once every week!

Lectures, Assignments, and Examinations

- Lectures to explain and to discuss concepts

- Assignments to reinforce hands-on applications

- Exams to check how well the material is learned

Some Tips

- Come to lecture

- Do the projects
  - You will have difficulty with exams without doing the projects

- Ask questions

- Start assignments early

Academic Honesty

- **Examinations**
  - No talking, i.e., "No cheating!"

- **Assignments**
  - May talk to each other about solutions but never copy code or algorithms from others' assignments
  - Best solution: Do not even look at others' codes!

Course Objectives

- To explain why main memory algorithms are not efficient in external memory

- To introduce advanced data structures which

  - are able to handle large amounts of data due their efficient use of secondary storage

  - can be used for fast solution of problems such as searching, sorting, etc.

- To introduce structures and techniques necessary for other courses such as Analysis of Algorithms, Databases, Computer Networks, and Artificial Intelligence

Week 1

Course Outline

- Introduction
  - *Growth of Functions*
  - *Recurrences*
  - *Probabilistic Analysis and Randomized Algorithms*
- Sorting and Order Statistics
  - Merge, Heap, QuickSort
  - Sorting in Linear Time
  - *Medians and Order Statistics*
- Data Structures, Advanced Analysis Techniques and Advanced Data Structures
  - Hash Tables
  - Binary Search Trees
  - Red-Black Trees
  - *Amortized Analysis*
  - B-Trees
  - Binomial and Fibonacci Heaps

Week 1

9

Where Does This Course Fit In?

DS
Pointers
Arrays
Queues
Lists
Dynamic and Static
   Memory
Sorting (Quick)
Searching
Trees

AOA 1
Secondary Storage Devices
Asymptotic Analysis
Recurrences
Sorting (Merge, Heap,Quick)
Sorting in Linear Time
Hashing
Binary Search Trees
Red-Black Trees
B-Trees
Amortized Analysis
Binomial Heaps
Fibonacci Heaps

AOA 2
Graphs
Greedy Algorithms
Divide and Conquer
Dynamic Programming
Network Flow
NP and Computational
   Intractability

10

Algorithm

- Historical aside:
  - Persian astronomer and mathematician Muhammad al-Khwarizmi (825 A.D.), in Latin Algoritmi (12th century)
  - First algorithm: Euclidean Algorithm, greatest common divisor (300 B.C.)

More on algorithms in **BLG 372E Analysis of Algorithms2** next term

picture: google images

Euclid's Algorithm

- Goal: Compute the Greatest Common Divisor of a and b (a,b positive integers)

- Example: What is the Greatest Common Divisor

  of 1071 and 462?

Euclid's Algorithm

- **Why:** "What do African rhythms, spallation neutron source (SNS) accelerators in nuclear physics, string theory (stringology) in computer science, and an ancient algorithm described by Euclid have in common? The short answer is: patterns distributed as evenly as possible."
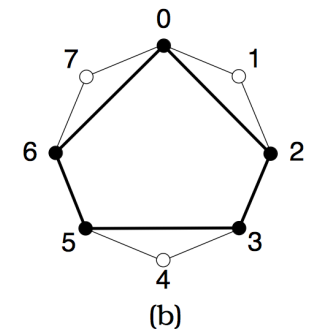
  Source: Toussaint, Godfried,

  "The Euclidean algorithm generates traditional musical rhythms", Proceedings of BRIDGES: Mathematical Connections in Art, Music, and Science (Banff, Alberta, Canada): 47–56

  Example rhythms: http://www.youtube.com/watch?v=YyyYW3bzzbQ
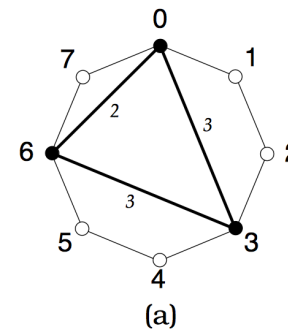
Cuban Tresillo   Cuban Cinquillo

- Brute force solution: try all possibilities!
- Don't, even if you have the strongest computers. Why? Will tell you later.

Euclid's Algorithm

- Euclid's Algorithm in English:

Starting with a and b, form a new pair consisting of the smaller number and the difference between the larger number and the smaller number. Repeat until the numbers in the new pair are equal to each other

Euclid's Algorithm

- Goal: Compute the Greatest Common Divisor of a and b (a,b positive integers)
- Algorithm: Starting with a and b, form a new pair consisting of the smaller number and the difference between the larger number and the smaller number. Repeat until the numbers in the new pair are equal to each other

- **Algorithm in Pseudocode:**

  function gcd(a, b)

      while a ≠ b

         if a > b

           a := a − b

         else

           b := b − a

      return a

  Try this: Compute gcd(1071, 462)

Week 1

Analysis of Algorithms 1

- Advanced data structures:
  - Heap
  - Hash table
  - Red-black tree
  - B-tree
  - Binomial and Fibonacci heaps
- Methods to analyze their performance:
  - Growth of functions
  - Recurrences
  - Probabilistic, amortized analysis

Algorithm and Data Structure

- Problem definition
- Algorithm
  - outline of computational procedure
  - step-by-step list of instructions
- Program
  - implementation of algorithm in programming language
- Data structure
  - organization and storage of data to facilitate access and modifications in order to solve the problem

Week 1

A Complete Program

- Investigating the problem
- Designing a program
  - Consider the computer architecture
  - Decide on / develop / design algorithm
- Writing the program (implementation)
- Testing the program (verification)
- Aiming at
  - Correctness
  - Efficiency
  - Robustness
  - Reusability
  - Adaptability

Why Do You Need AOA?

- Many experienced programmers were asked to code up binary search

- 80% got it wrong

- What did they lack?

  - Fundamental understanding of the algorithmic design techniques

  - Abstract thinking

  - Formal proof methods

Week 1

from slides by Jeff Edmonds, York University

Why Do You Need AoA?

- Boss assigns task:
- Given
  - prices of renting servers and
  - reaction time expected by customers
- Find the cheapest solution that tells which route to take when going from Beşiktaş to Bostancı

from slides by Jeff Edmonds, York University, pictures: google images

Why Do You Need AoA?

Your answer:

- "Um? Tell me what to code."
  - With more sophisticated software engineering systems, demand for mundane programmers will diminish

- "I learned this great algorithm that will work."
  - Soon all known algorithms will be available in libraries

from slides by Jeff Edmonds, York University

Why Do You Need AoA?

You need to understand

details of algorithms and data structures

in order to come up with

- a good solution for a new problem
  OR

- a better solution for an existing problem

from slides by Jeff Edmonds, York University
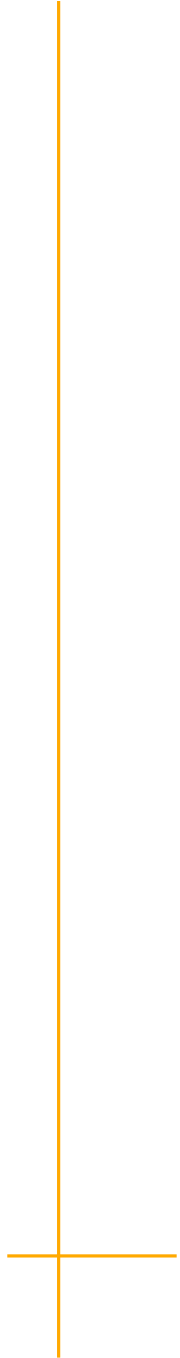
Week 1

Analysis of Algorithms 1
Istanbul Technical University
Computer Eng. Dept.

CHAPTER 1

Role of Algorithms in Computing

Course slides from Leiserson's MIT class have been used in preparation of these slides.

Analysis of Algorithms 1, Dr. Çataltepe & Dr. Ekenel, Dept. of Computer Engineering, ITU

Outline

- What is an Algorithm?
- Examples of Algorithms
- Correctness of Algorithms
- Specifying Algorithms
- Data Structures
- Algorithms as a Technology
- Efficiency of Algorithms

Algorithms

- Informally, an algorithm is any well-defined sequence of steps for solving well-specified computational problem

- Computational problem is specified as input / output relationship
  - An algorithm is thus sequence of steps that transforms input into output

Example: Sorting

- Sorting is a well-known computational problem

- Input: Sequence of n numbers $<a_1, a_2, \ldots, a_n>$

- Output: Permutation (reordering)

$<a'_1, a'_2, \ldots, a'_n>$ of input sequence such that

$a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- Example
  - Input: $<31, 41, 59, 26, 41, 58>$
  - Output: $<26, 31, 41, 41, 58, 59>$

Correctness of Algorithms

- Algorithm said to be **correct** if, for every input instance, it halts with correct output

- Correct algorithm **solves** given computational problem

- Incorrect algorithm might not halt at all on some input instances, or it might halt with answer other than desired one
  - Can sometimes be useful, if their error rate can be controlled

Specifying Algorithms

- Algorithm can be specified
  - in English, Turkish, Chinese, etc. (i.e., natural languages)
  - as a computer program
  - as a hardware design

- *Only requirement:* Specification must provide precise description of computational procedure to be followed

What Kinds of Problems Do Algorithms Solve?

- Computational biology: analyzing three billion chemical base pairs that make up human DNA

- Search engines: quickly finding pages that store relevant information

- Computer networks: efficient routing

- Electronic commerce: public-key cryptography and digital signatures

- Manufacturing/commercial settings: assigning crew to flights in least expensive way

Data Structures

- **Data structure:** way to store and organize data in order to facilitate access and modifications

- No single data structure works well for all purposes
  - Important to know strengths and limitations of several of them

Motivation for Data Structures

- Similar to algorithms: "What is fastest way to solve problem X?"

- Difference: Focuses on ways to organize typically long-lived corpus of data so that queries can be answered really fast

- Typical example: web search engines
  - Store sophisticated indexes that support a variety of queries ("What documents contain word X?") quickly
  - "Linear time" too slow: cannot afford to scan through entire web to answer a single query
  - Goal: typically logarithmic or even constant time per query
  - Space tends to become a more important issue because data corpus is usually big and we do not want a data structure that is larger (or even as large as) the data itself

from Prof. Erik Demaine's MIT 6.897 ADS course

Motivation for Data Structures

- Almost every algorithmic problem has at least a small data structural component and knowledge about data structures and the literature has made faster algorithms possible

from Prof. Erik Demaine's MIT 6.897 ADS course

Examples of Recent and Cool Data Structures

- Text indexing and web searching

  – Preprocess text in linear time so that you can search for an arbitrary substring in time proportional to length of that substring

  – Searching for one word is really really fast

  – Recent improvement: Data structure can be made small, proportional to text size and if text is compressible, data structure can be equally compressed

- Sorting

  – Sorting is $\Theta(n \lg n)$, but that is just in comparison model

  – More typical example in ADS world is sorting integers: best known sorting algorithm runs in $O(n\sqrt{\lg \lg n})$ time, and it is even conceivable that sorting runs in $O(n)$ time

from Prof. Erik Demaine's MIT 6.897 ADS course

Examples of Recent and Cool Data Structures

- Sorting and priority queues

  - Any priority queue data structure can be turned into a sorting algorithm

  - Reverse is also true: Any sorting algorithm that runs in $\mathrm{O}(n \cdot f(n))$ time can be converted into a priority queue data structure with $\mathrm{O}(f(n))$ time per operation

- Cache oblivious data structures

  - Any problem (sorting, priority queues, etc.) is different when you are dealing with disk instead of main memory or you care about cache performance

  - New wave of algorithms and data structures seems to capture most of these issues in theoretically clean way, and there have been several optimal results along these lines

from Prof. Erik Demaine's MIT 6.897 ADS course

Technique

- You can use your textbook as a "cookbook" for algorithms

- You may someday encounter a problem for which you cannot readily find a published algorithm

- We will learn techniques of algorithm design and analysis so that we can
  - develop algorithms
  - show that they give correct answer
  - understand their efficiency

Algorithms as a Technology

- Algorithms are a technology, just as are
  - Fast hardware
  - Graphical user interfaces
  - Object-oriented systems
  - Networks

Algorithms as a Technology

- **Suppose:** Computers were infinitely fast and computer memory was free

- Would you have any reason to study algorithms?

- Yes, you would still like to demonstrate that your solution method terminates and does so with correct answer

Algorithms as a Technology

- If computers were infinitely fast, any correct method for solving a problem would do

- You would probably want your implementation to be within bounds of good software engineering practice (well designed and documented), but you would most often use whichever method was easiest to implement

Algorithms as a Technology

- Computers may be fast, but they are not infinitely fast

- Memory may be cheap, but it is not free

- Computing time is a bounded resource and so is space in memory

- These resources should be used wisely
  - Algorithms that are efficient in terms of time or space will help you do so

Efficiency

- Algorithms devised to solve same problem often differ dramatically in their efficiency

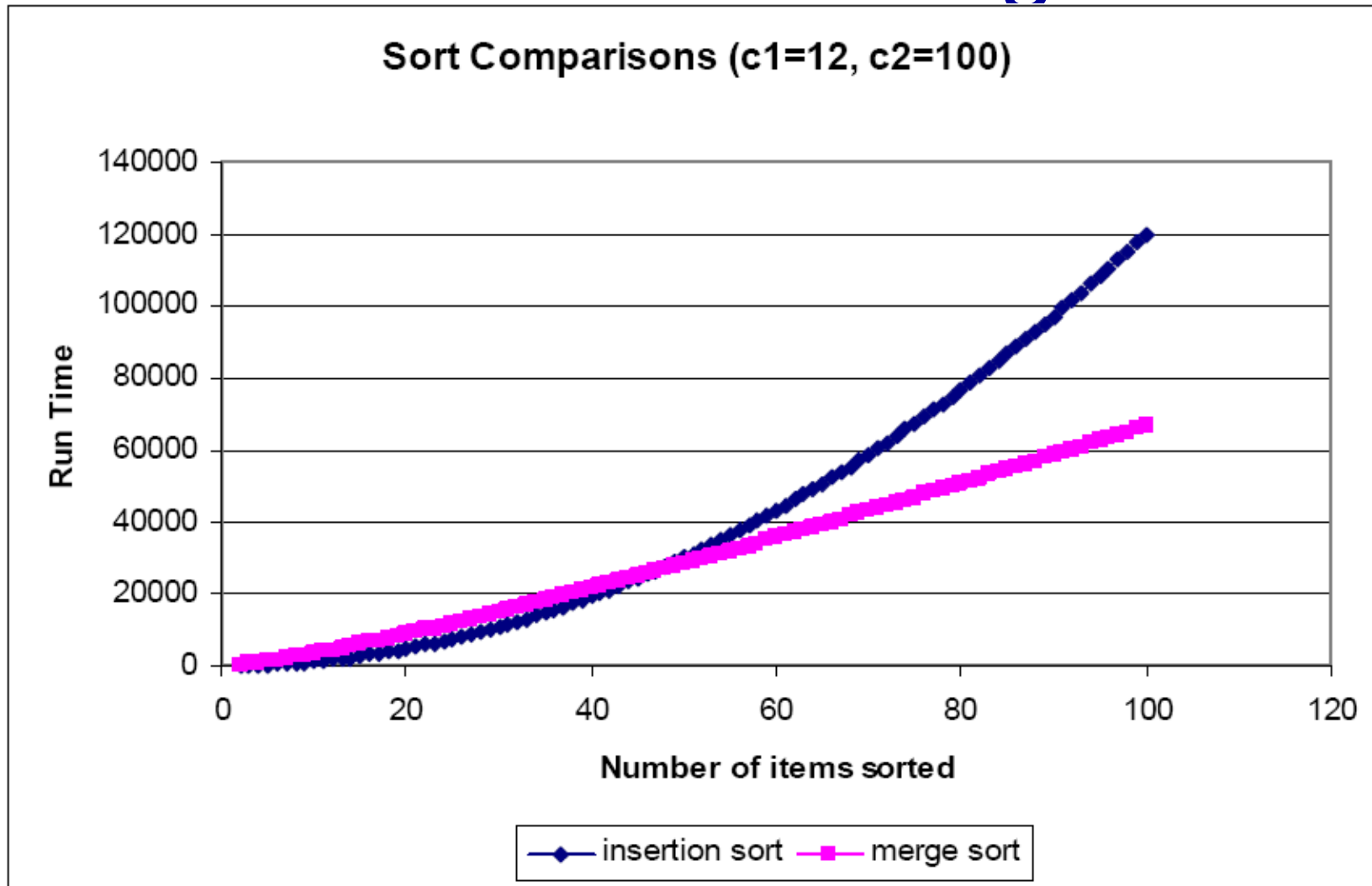- These differences can be much more significant than differences due to hardware and software

Efficiency: Two Algorithms for Sorting

- ## Insertion sort
  - c_1n^2 to sort n items, where constant c_1 does not depend on n (takes time roughly proportional to n^2)

- ## Merge sort
  - Takes time roughly proportional to $c_2n\lg n$,

  where $\lg n$ stands for $\log_2 n$ and

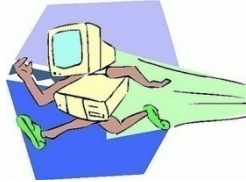 c_2 is another constant that does not depend on n

Insertion Sort vs. Merge Sort

- Insertion sort usually has a smaller constant factor than merge sort, so that $c_1 < c_2$

- Constant factors can be far less significant in running time than dependence on input size n

- Where merge sort has a factor of lg$n$ in its running time, insertion sort has a factor of n, which is much larger

- Although insertion sort is usually faster than merge sort for small input sizes, once input size n becomes large enough, merge sort's advantage of lg$n$ vs. n will more than compensate for difference in constant factors

- No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster

Insertion Sort vs. Merge Sort

Sort Comparisons (c1=12, c2=100)

Run Time vs. Number of items sorted

Legend: insertion sort, merge sort

from lecture notes by Prof. John Boon, Jr., Hood College

Efficiency: Concrete Example

Computer A

- Insertion sort
- One billion instructions per second
- World's craftiest programmer, machine language
- $2n^2$ instructions to sort n numbers
- Sorting one million numbers takes

$$\frac{2(10^6)^2 \text{ instructions}}{10^9 \text{ instructions/sec}} = 2000 \text{ sec}$$

- Sorting ten million numbers takes 2.3 days

Computer B

- Merge sort
- Ten million instructions per second
- Average programmer, high-level language with inefficient compiler
- $50n\lg n$ instructions to sort n numbers
- Sorting one million numbers takes

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instructions}}{10^7 \text{ instructions/sec}} \approx 100 \text{ sec}$$

- Sorting ten million numbers takes under 20 minutes

In general as problem size increases, so does relative advantage of merge sort

Comparison of Running Times

| Running Time in µs	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2500	150000	9000000
$20n \log n$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

- For each function $f(n)$ and time t, largest size n of a problem that can be solved in time t, assuming that algorithm to solve problem takes $f(n)$ microseconds

Algorithms and Other Technologies

- Example showed that algorithms, like computer hardware, are a **technology**

- Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware

- Although there are some applications that do not explicitly require algorithmic content at the application level (e.g., some simple web-based applications), most also require a degree of algorithmic content on their own

Algorithms and Other Technologies

- Even an application that does not require algorithmic content at the application level relies heavily upon algorithms

  - GUI, networking, use of language other than machine code (processed by compiler, interpreter, or assembler)

- With ever-increasing capacities of computers, we use them to solve larger problems than ever before

- At larger problem sizes, differences in efficiencies in algorithms become particularly prominent

- Having a solid base of algorithmic knowledge and technique is one characteristic that separates truly skilled programmers from novices
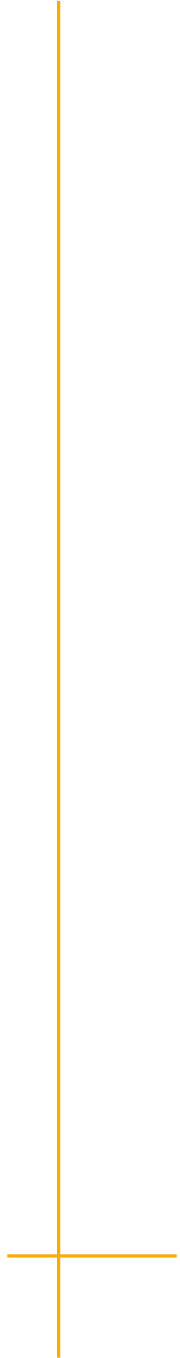
Week 1

Analysis of Algorithms 2 (Fall 2013) Istanbul Technical University Computer Eng. Dept.

Chapter 2: Getting Started

Course slides from
Leiserson's @MIT
Edmonds@York Un.
Ruan @UTSA
have been used in
preparation of these slides.

Last updated: Sept. 18, 2013

Outline

- **Insertion Sort**
  - Pseudocode Conventions
  - Analysis of Insertion Sort
  - Loop Invariants and Correctness

- Merge Sort
  - Divide and Conquer
  - Analysis of Merge Sort

- Growth of Functions
  - Asymptotic notation
  - Comparison of functions
  - Standard notations and common functions

Week 1

51

Sorting Problem

Input: A sequence of n numbers

$$\langle a_1, a_2, ..., a_n \rangle$$

Output: A permutation (reordering)

$$\langle a_1', a_2', ..., a_n' \rangle$$

such that

$$a_1' <= a_2' <= ... <= a_n'$$

Sorting

- Numbers to be sorted (a_i) also known as *keys*
  - for example, studentID, checkNumber, etc.
- In real implementations of sorting algorithms records need to also be moved with keys
- To save time, pointers may be copied instead of actual records
- Algorithm does not concern itself with these details, but it will be your job when you actually do the programming

Insertion Sort

- Simple algorithm

- Basic idea:

  - Assume initial j-1 elements are sorted

  - Until you find place to insert j^{th} element, move array elements to right

  - Copy j^{th} element into its place

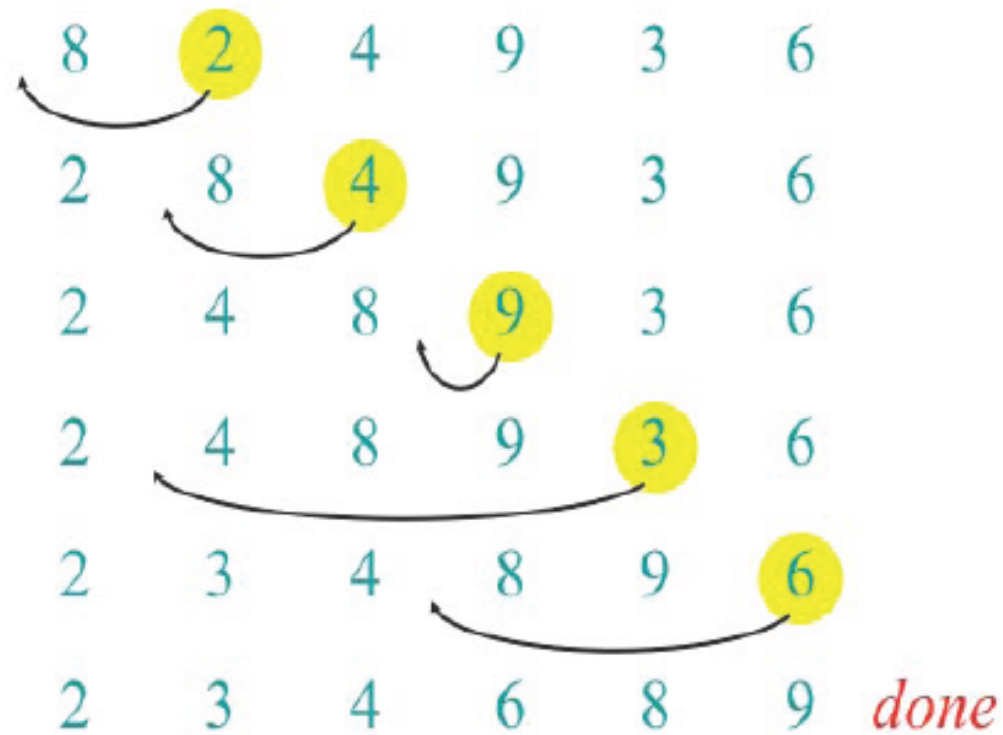- Insertion Sort is an "in place" sorting algorithm. No extra storage is required.

Insertion Sort Example

8   2   4   9   3   6
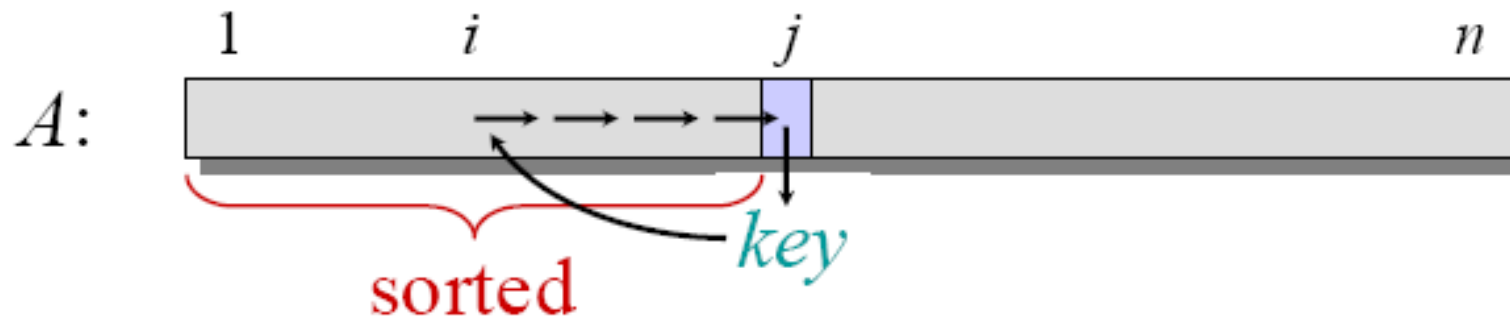
Week 1

Insertion Sort Example

8   2   4   9   3   6

Insertion Sort Example

Insertion Sort

INSERTION-SORT (A, n) $\quad \triangleright A[1 \ldots n]$

$\quad$ **for** $j \leftarrow 2$ **to** n

$\quad\quad$ **do** $key \leftarrow A[j]$

$\quad\quad\quad$ $i \leftarrow j - 1$

$\quad\quad\quad$ **while** $i > 0$ and $A[i] > key$

$\quad\quad\quad\quad$ **do** $A[i+1] \leftarrow A[i]$

$\quad\quad\quad\quad\quad$ $i \leftarrow i - 1$

$\quad\quad\quad$ $A[i+1] = key$

"pseudocode"

A:

1 $\quad$ i $\quad$ j $\quad$ n

key

sorted

Pseudocode Conventions

- Indentation
  - indicates block structure
  - saves space and writing time
- Looping constructs (**while, for, repeat**) and conditional constructs (**if, then, else**)
  - like in C, C++, and Java
  - we assume that loop variable in a **for** loop is still defined when loop exits
- Symbol "$\triangleright$" indicates that remainder of line is comment
- Multiple assignment $i \leftarrow j \leftarrow e$ assigns to both variables i and j value of e (== $j \leftarrow e, i \leftarrow j$)
- Variables are local, unless otherwise specified

Pseudocode Conventions

- Array elements are accessed by specifying array name followed by index in square brackets
  - A[i] indicates ith element of array A
  - Notation ".." is used to indicate a range of values within an array (A[i..j] = A[1], A[2],…, A[j])
- We often use **objects**, which have **attributes** (equivalently, **fields**)
  - For an attribute *attr* of object x, we write *attr*[*x*]
  - Equivalent of *x.attr* in Java or *x-> attr* in C++
- Objects are treated as references, like in Java
  - If *x* and *y* denote objects, then assignment *y* ← *x* makes *x* and *y* reference same object
  - It does not cause attributes of one object to be copied to another

10/09/14
Week 1

Pseudocode Conventions

- Parameters are passed **by value**, as in Java and C (and the default mechanism in C++).
  - When an object is passed by value, it is actually a reference (or pointer) that is passed
  - Changes to the reference itself are not seen by caller, but changes to the object's attributes are
- Boolean operators "and" and "or" are **short-circuiting**
  - If after evaluating left-hand operand, we know result of expression, then we do not evaluate right-hand operand
  - If x is FALSE in "x and y", then we do not evaluate y
  - If x is TRUE in "x or y", then we do not evaluate y

10/09/14
Week 1

Efficiency

- Correctness alone is not sufficient
- **Brute-force** algorithms exist for most problems
- To sort n numbers, we can enumerate all permutations of these numbers and test which permutation has the correct order
  - Why cannot we do this?
  - Too slow!
  - By what standard?

How to measure complexity?

- Accurate running time is not a good measure

- It depends on **input**

- It depends on the **machine** you used and who implemented the algorithm

- We would like to have an analysis that **does not depend** on those factors

Machine-independent

- A generic uniprocessor random-access machine (RAM) model
  - No concurrent operations
  - Each simple operation (e.g. +, -, =, *, if, for) takes 1 step.
    - Loops and subroutine calls are *not* simple operations.
  - All memory equally expensive to access
    - Constant word size
    - Unless we are explicitly manipulating bits
    - No memory hierarch (caches, virtual mem) is modeled

Running Time

- **Running Time:T(n):** Number of primitive operations or steps executed for an input of size n.

- Running time depends on input
  - already sorted sequence is easier to sort

- Parameterize running time by size of input
  - short sequences are easier to sort than long ones

- Generally, we seek upper bounds on running time
  - everybody likes a guarantee

Kinds of Analysis

- **Worst-case:** (usually)
  - T(n) = maximum time of algorithm on any input of size n

- **Average-case:** (sometimes)
  - T(n) = expected time of algorithm over all inputs of size n

  - Need assumption about statistical distribution of inputs

- **Best-case:** (bogus)
  - Cheat with a slow algorithm that works fast on some input

Analysis of insertion Sort

Statement	cost	time
`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	$\sum_{j=2..n} t_j$
`      A[i+1] = A[i]`	c_5	$\sum_{j=2..n} t_j -1$
`      i = i - 1`	c_6	$\sum_{j=2..n} t_j -1$
`    }`	0	
`    A[i+1] = key`	c_7	(n-1)
`  }`	0	
`}`		

t_j : number of times while loop test is executed for j^{th} **`for`** loop iteration

Analysis of insertion Sort

Statement	cost	time
`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	S
`            A[i+1] = A[i]`	c_5	$S - (n-1)$
`            i = i - 1`	c_6	$S - (n-1)$
`        }`	0	
`        A[i+1] = key`	c_7	(n-1)
`    }`	0	
`}`		

t_j : number of times while loop test is executed for j^{th} **for** loop iteration

Analyzing Insertion Sort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 S + c_5(S - (n-1)) + c_6(S - (n-1)) + c_7(n-1)$

    $= c_8 S + c_9 n + c_{10}$

- What can S be?
    - Best case -- inner loop body never executed
        - $t_j = 1 \rightarrow S = n - 1$
        - $T(n) = an + b$ is a linear function
    - Worst case -- inner loop body executed for all previous elements
        - $t_j = j \rightarrow S = 2 + 3 + \ldots + n = n(n+1)/2 - 1$
        - $T(n) = an^2 + bn + c$ is a quadratic function
    - Average case
        - Can assume that in average, we have to insert A[j] into the middle of A[1..j-1], so $t_j = j/2$
        - $S \approx n(n+1)/4$
        - $T(n)$ is still a quadratic function

Insertion Sort Running Time

Theta Notation, see next week.

- **Best-case:**
  - $\Theta(n)$, inner loop not executed at all
- **Worst-case:** Input reverse sorted
  - $T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$ [Arithmetic series]

- **Average-case:** All permutations equally likely
  - $T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$

Is Insertion Sort a fast sorting algorithm?
- Moderately so, for small n
- Not at all, for large n

70

Week 1

Asymptotic Analysis

- Ignore actual and abstract statement costs
- *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
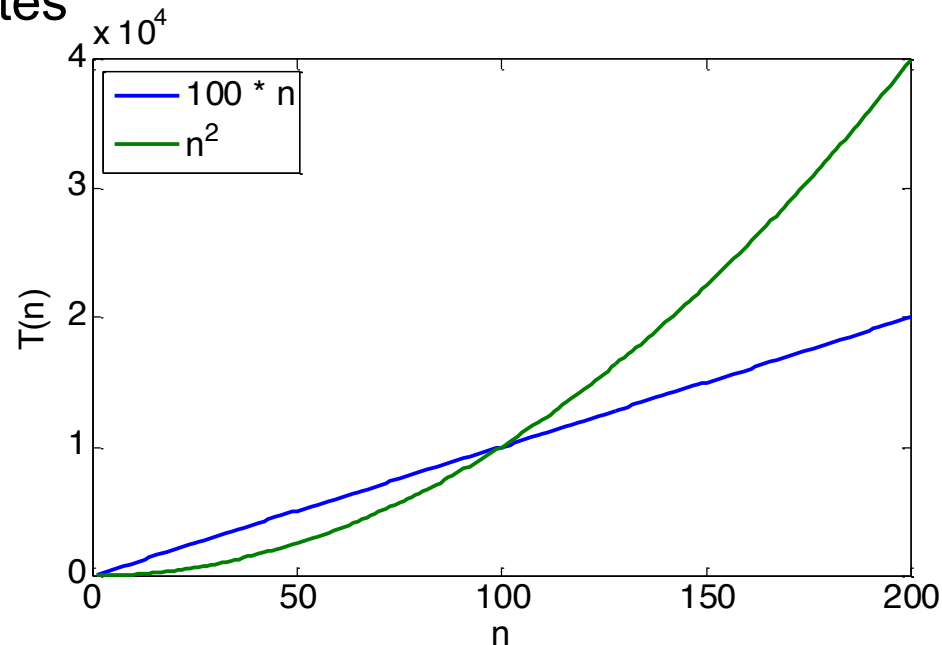
Loop invariants and correctness of insertion sort

- Claim: at the start of each iteration of the for loop, the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

- Proof: by induction

Review: Proof By Induction

- Claim:S(n) is true for all n >= k

- Basis:
  - Show formula is true when n = k

- Inductive hypothesis:
  - Assume formula is true for an arbitrary n

- Step:
  - Show that formula is then true for n+1

Prove correctness using loop invariants

- **Initialization (basis):** the loop invariant is true prior to the first iteration of the loop

- **Maintenance:**
  - Assume that it is true before an iteration of the loop (Inductive hypothesis)
  - Show that it remains true before the next iteration (Step)

- **Termination:** show that when the loop terminates, the loop invariant gives us a useful property to show that the algorithm is correct

Prove correctness using loop invariants

```
InsertionSort(A, n) {
  for j = 2 to n {
     key = A[j];
     i = j - 1;
     ▷Insert A[j] into the sorted sequence A[1..j-1]
     while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;

     }
     A[i+1] = key
  }
}
```

Loop invariant: at the start of each iteration of the for loop, the subarray consists of the elements originally in A[1..j-1] but in sorted order.

Initialization

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j];
      i = j - 1;
      ▷ Insert A[j] into the sorted sequence A[1..j-1]

      while (i > 0) and (A[i] > key) {
          A[i+1] = A[i];
          i = i - 1;

      }
      A[i+1] = key
  }
}
```
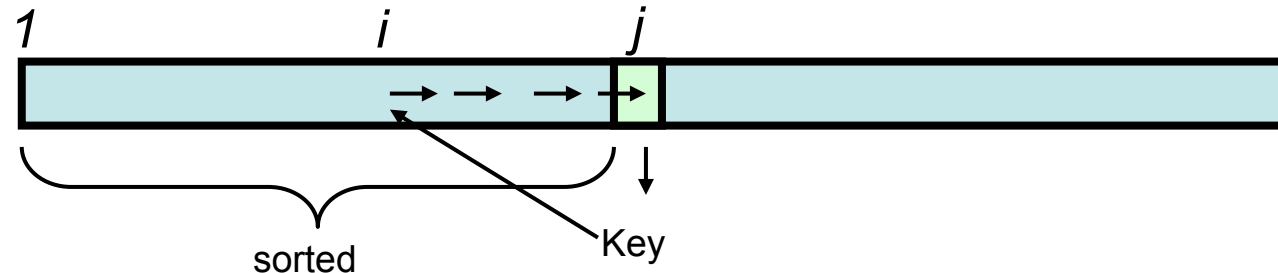
Subarray A[1] is sorted. So loop invariant is true before the loop starts.

Maintenance

```
InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];
    i = j - 1;
    ▷ Insert A[j] into the sorted sequence A[1..j-1]
    while (i > 0) and (A[i] > key) {
      A[i+1] = A[i];
      i = i - 1;
    }
    A[i+1] = key
  }
}
```

Assume loop variant is true prior to iteration j

Loop variant will be true before iteration j+1

1   i   j

sorted   Key

Termination

```
InsertionSort(A, n) {
  for j = 2 to n {
    key = A[j];      The algorithm is correct!
    i = j - 1;
    ▷Insert A[j] into the sorted sequence A[1..j-1]

    while (i > 0) and (A[i] > key) {
        A[i+1] = A[i];
        i = i - 1;

    }
    A[i+1] = key
  }
}
```

Upon termination, A[1..n] contains all the original elements of A in sorted order.

1   *n j=n+1*

Sorted