Design and Analysis of Algorithms
Lecture-1: Introduction

Prof. Eugene Chang

Class Web Site

http://class.svuca.edu/~eugene.chang/class/CS502_2015_Fall/

Instructor: eugene.chang@svuca.edu

Grader:

12:30 – 3:30 140301071@svuca.edu (Aishwarya Sukumaran)

4 – 7 130301044@svuca.edu (Srujana Ramanam)

Overview

- Class logistics and policies
- Introduction
  - Why should you study algorithms
  - What is an algorithm
  - Correctness and efficiency
  - Examples: Insertion sort
- Lecture materials are shared with Prof K. K. Low
- Part of the slides are based on material from Prof. Jianhua Ruan, The University of Texas at San Antonio, and Prof. Jennifer Welch, Texas A&M University

About Myself

- Me: Yuh-Lin Eugene Chang, originally from Taiwan
- MS from UCSB, PhD Computer Engineering from U of Texas Austin
- 22 years industry R&D (1993–now)
  - Large companies such as Panasonic, LG, and Intel
  - Small startups in SOC
  - Currently CTO of emReal Corp., working on e-commerce and mobile software
- Teaching
  - With SVU since 2008
  - Also teach Machine Learning and Data Mining

Class Structures

- Textbook
  - **Introduction to Algorithms, Third Edition** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Publisher: MIT Press (July 31, 2009). ISBN-10: 0262033844. ISBN-13: 978-0262033848
- Course focus
  - Algorithm analysis
  - Sorting algorithms
  - Data structures
  - Optimization
- Grades
  - Homework 30%, Midterm 25%, Final 35%, Others 10%
- Course website
  - http://class.svuca.edu/~eugene.chang/class/CS502_2015_Fall/
  - Lecture notes, homework assignments, and solutions will be posted

Teaching Plan (subject to change)

Week	Topic	Reading/Homework/Case Assignment
1 (Sep 12)	Introduction	Ch. 1, Ch. 2
2	Asymptotic Analysis	Ch. 3, (HW-1)
3	Divide-and-Conquer	Ch. 4
4	Heapsort	Ch. 6, (HW-2)
5	Quicksort	Ch. 7
6	Sorting in Linear Time	Ch. 8, (HW-3)
7	Midterm Review	Class Notes
8 (Oct 31)	Mid-Term Exam	
9	Data Structures, Hash Tables	Ch. 10, Ch. 11
10	Binary Search Tree	Ch. 12, (HW-4)
11	Dynamic Programming	Ch. 15
12	Greedy Algorithms	Ch. 16, (HW-5)
13	Greedy Algorithms	Ch. 16
14	Final Review	Class Notes
15 (Dec 19)	Final Exam	

Class Policies

- Homework submission
  - Submit in e-mail form to the grader on the due date
  - 25% penalty each additional day after the submission deadline
  - Submission will not be accepted once the solution is posted online
- Exams
  - Close book, close note
  - Cannot be made up, cannot be taken early, and must be taken in class at the scheduled time. Proofs are needed for exceptions or true emergencies
- Cheating
  - Will not be tolerated!
  - Cheating in an exam will result in failing the course
- Attendance will be taken every lecture and counted as part of the grades

What is an algorithm?

- An algorithm is a sequence of computational steps that transform the input into the output
- An algorithm is a step-by-step procedure to solve a problem
- Every program is the instantiation of some algorithms
  - Algorithm1 + algorithm2 + …, + algorithmN ➔ Program
- Algorithm is the thing that stays the same regardless of programming language and the computing hardware

What kinds of problems are solved by algorithms?

- The Human Genome Project
- The Internet
- Electronic commerce
- Optimization of resource allocation for manufacturing and other commercial enterprises
- Human face detection and recognition
- Stock prediction and trading
- Many others

Why Study Algorithms

- There are only a handful of classical problems
  - Nice algorithms have been designed for them
- If you know how to solve a classical problem (e.g., the shortest-path problem), you can use it to do a lot of different things
  - Abstract ideas from the classical problems
  - Map your requirement to a classical problem
  - Solve with classical algorithms
  - Modify it if needed
- Learn meta algorithms to design new algorithms
  - A meta algorithm is a class of algorithms for solving similar abstract problems
  - There are only a handful of them, e.g. divide and conquer, greedy algorithm, dynamic programming
  - Learn the ideas behind the meta algorithms to design new ones

Modeling the Real World

- Cast your application in terms of well-studied abstract data structures

Concrete	*Abstract*
arrangement, tour, ordering, sequence	permutation
cluster, collection, committee, group, packaging, selection	subsets
hierarchy, ancestor/descendants, taxonomy	trees
network, circuit, web, relationship	graph
sites, positions, locations	points
shapes, regions, boundaries	polygons
text, characters, patterns	strings

Some Important Problem Types

- Sorting
  - a set of items
- Searching
  - among a set of items
- String processing
  - text, bit strings, gene sequences
- Graphs
  - model objects and their relationships
- Combinatorial
  - find desired permutation, combination or subset
- Geometric
  - graphics, imaging, robotics
- Numerical
  - continuous math: solving equations, evaluating functions

Algorithm Design Techniques

- Brute Force & Exhaustive Search
  - follow definition / try all possibilities
- Divide & Conquer
  - break problem into distinct subproblems
- Transformation
  - convert problem to another one
- Dynamic Programming
  - break problem into overlapping subproblems
- Greedy
  - repeatedly do what is best now
- Iterative Improvement
  - repeatedly improve current solution
- Randomization
  - use random numbers

How to express algorithms?

Increasing precision

Nature language (e.g. English)

Pseudocode

Real programming languages

Ease of expression

Describe the *ideas* of an algorithm in nature language.
Use pseudocode to clarify sufficiently tricky details of the algorithm.

How to express algorithms?

Increasing precision

Nature language (e.g. English)

Pseudocode

Real programming languages

Ease of expression

To understand / describe an algorithm:

Get the big idea first.

Use pseudocode to clarify sufficiently tricky details

Example: sorting

- Input: A sequence of N numbers $a_1 \ldots a_n$
- Output: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \ldots \leq a_n$.
- Possible algorithms you've learned so far
  - Insertion, selection, bubble, quick, merge, …
  - More in this course
- We seek algorithms that are both *correct* and *efficient*

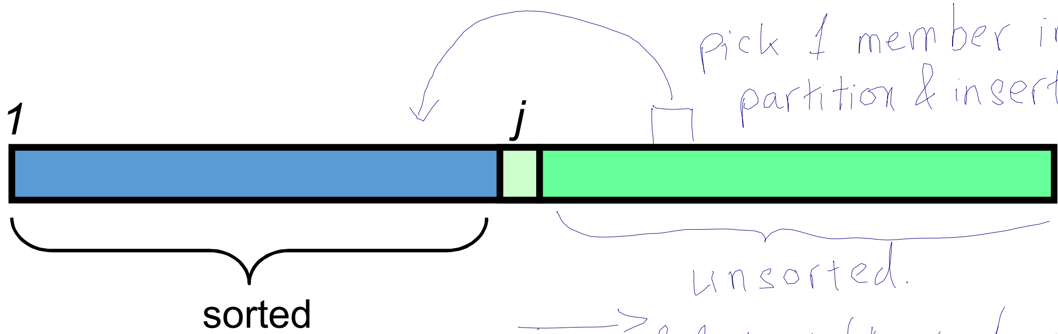
Insertion Sort

```
InsertionSort(A, n) {
 for j = 2 to n {

     ▷ Pre condition: A[1..j-1] is sorted

     1. Find position i in A[1..j-1] such that A[i] ≤ A[j] < A[i+1]
     2. Insert A[j] between A[i] and A[i+1]

 }   ▷ Post condition: A[1..j] is sorted
}
```

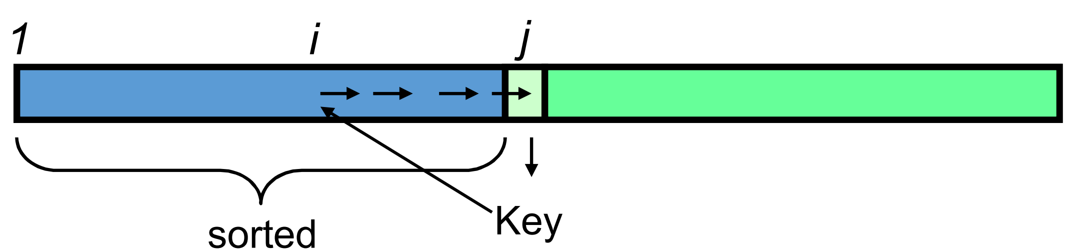


pick 1 member in the unsorted partition & insert into the sorted partition orderly

move left to right gradually.

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
 }
}
```



Correctness

- What makes a sorting algorithm correct?
  - In the output sequence, the elements are ordered non-decreasingly
  - Each element in the input sequence has a unique appearance in the output sequence
    - [2 3 1] => [1 2 2]   X
    - [2 2 3 1] => [1 1 2 3]   X

Correctness

- For any algorithm, we must prove that it *always* returns the desired output for *all* legal instances of the problem.
- For sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Algorithm correctness is NOT obvious in some problems (e.g., optimization)

How to prove correctness?

- Given a concrete input, eg. <4,2,6,1,7>
  trace it and prove that it works.
- Given an abstract input, eg. <a_1, … a_n>
  trace it and prove that it works.
- Sometimes it is easier to find a counterexample to show that an algorithm does NOT work.
  - Think about all small examples
  - Think about examples with extremes of big and small
  - Think about examples with ties
  - Failure to find a counterexample does NOT mean that the algorithm is correct

Review: Induction

- Suppose
  - S(k) is true for fixed constant k
    - Often k = 0 or 1
  - S(n) → S(n+1) for all n >= k
- Then S(n) is true for all n >= k

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

Induction Example: Gaussian Closed Form

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

    $1 + 2 + \ldots + n + n+1 = (1 + 2 + \ldots + n) + (n+1)$

    $= n(n+1)/2 + n+1 = [n(n+1) + 2(n+1)]/2$

    $= (n+1)(n+2)/2 = (n+1)(n+1 + 1) / 2$

Induction Example: Geometric Closed Form

- Prove $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for all $a \neq 1$
  - Basis: show that $a^0 = (a^{0+1} - 1)/(a - 1)$

    $a^0 = 1 = (a^1 - 1)/(a - 1)$
  - Inductive hypothesis:
    - Assume $a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$
  - Step (show true for n+1):

    $a^0 + a^1 + \ldots + a^{n+1} = a^0 + a^1 + \ldots + a^n + a^{n+1}$

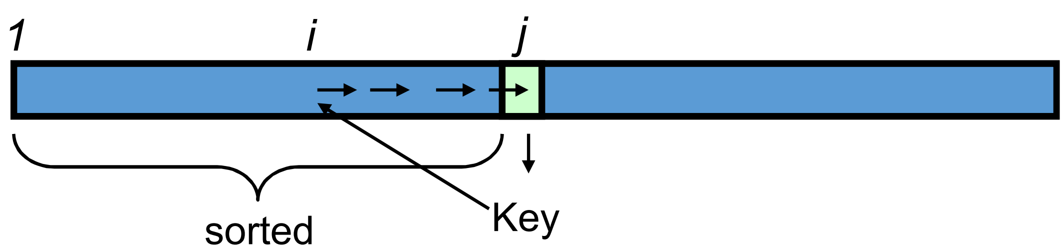
    $= (a^{n+1} - 1)/(a - 1) + a^{n+1} = (a^{n+1+1} - 1)/(a - 1)$

Induction

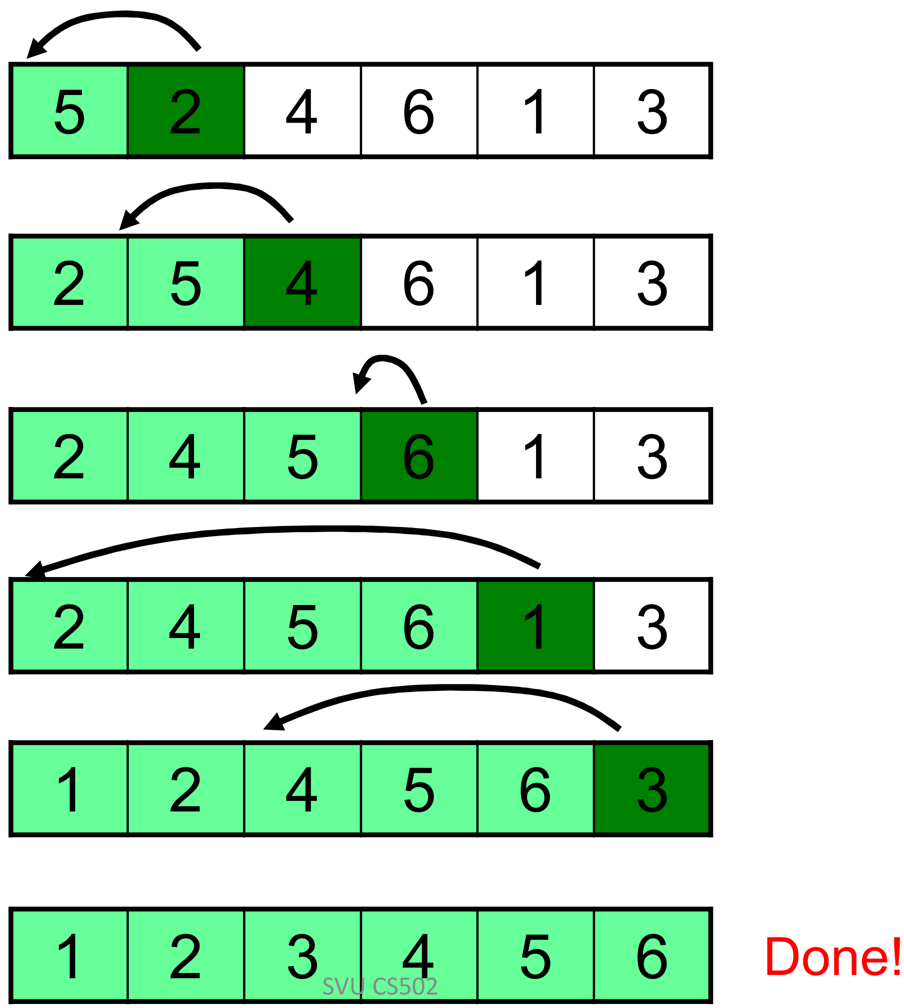
- We've been using *weak induction*
- *Strong induction* also holds
  - Basis: show S(0)
  - Hypothesis: assume S(k) holds for arbitrary k <= n
  - Step: Show S(n+1) follows
- Another variation:
  - Basis: show S(0), S(1)
  - Hypothesis: assume S(n) and S(n+1) are true
  - Step: show S(n+2) follows

An Example: Insertion Sort

```
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
 }
}
```



Example of insertion sort



Use loop invariants to prove the correctness of loops

- A loop invariant (LI) is a formal statement about the variables in your program which holds true throughout the loop
- 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
  - Initialization: the LI is true prior to the 1^{st} iteration
  - Maintenance: if the LI is true before the j^{th} iteration, it remains true before the $(j+1)^{th}$ iteration
  - Termination: when the loop terminates, the LI 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 A[1..j-1] 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.

Loop invariant: 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.

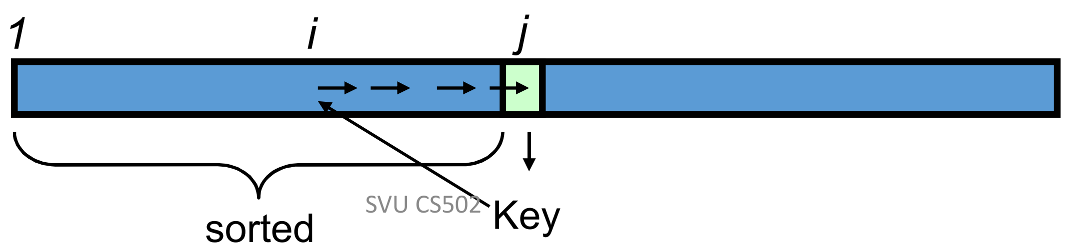
Maintenance

Loop invariant: 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.

```
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



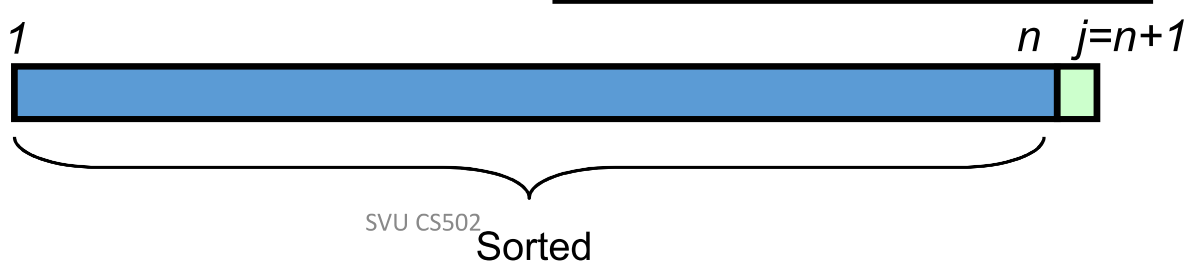
Termination

Loop invariant: 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.

```
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
 }
}
```

The algorithm is correct!

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



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?

Analysis of Algorithms

- Analysis is performed with respect to a computational model
- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size
    - Unless we are explicitly manipulating bits

How to measure complexity?

- Raw 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

n (list size)	Computer A run-time	Computer B run-time
15	7	100,000
65	32	150,000
250	125	200,000
1,000	500	250,000
…	…	…
1,000,000	500,000	500,000
4,000,000	2,000,000	550,000
16,000,000	8,000,000	600,000

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

Input Size

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

Running Time

- 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(x)
- We can be more exact if need be

Asymptotic Analysis

- Running time depends on the size of the input
  - Larger array takes more time to sort
  - T(n): the time taken on input with size n
  - Look at ***growth*** of $T(n)$ as $n \to \infty$.

    “Asymptotic Analysis”
- Size of input is generally defined as the number of input elements
  - In some cases may be tricky

Running time of insertion sort

- The running time depends on the input: an already sorted sequence is easier to sort.
- Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of analyses

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Best case – not very useful
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs

Analysis of 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 line execute?

Analysis of 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 line execute?

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	
`}`		

$S = t_2 + t_3 + \ldots + t_n$ where t_j is number of while expression evaluations for the j^{th} for loop iteration

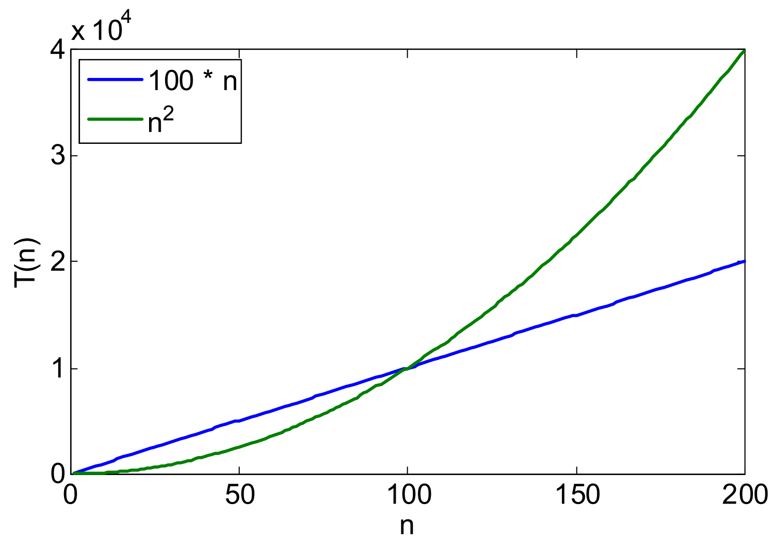
Analyzing Insertion Sort

- $T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4S + c_5(S - (n-1)) + c_6(S - (n-1)) + c_7(n-1)$

  $= c_8S + c_9n + c_{10}$
- 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 on 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

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



Comparison of functions

	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	10^2	10^3	10^3	10^6
10^2	6.6	10^2	660	10^4	10^6	10^{30}	10^{158}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	10^5	10^8	10^{12}		
10^5	17	10^5	10^6	10^{10}	10^{15}		
10^6	20	10^6	10^7	10^{12}	10^{18}		

For a super computer that does 1 trillion operations per second, it will be longer than 1 billion years

Order of growth

$1 << \log_2 n << n << n\log_2 n << n^2 << n^3 << 2^n << n!$

(We are slightly abusing of the “<<“ sign. It means a smaller order of growth).

Asymptotic Performance

- 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.

Asymptotic Notation

- By now you should have an intuitive feel for asymptotic (big-O) notation:
  - *What does O(n) running time mean? $O(n^2)$? O(n lg n)?*
  - *How does asymptotic running time relate to asymptotic memory usage?*
- Our task is to define this notation more formally and completely
  - Job for the next lecture

Practice: Analyze this…

```
AllUnique(A[1..n])
for I = 1 to n
        for j = 1 to n
                if A[i] = A[j] and i ≠ j return false
return true
```

Best case?
Worst case?
Average case?

Analyze this…

```
AllUnique(A[1..n])
for I = 1 to n
        for j = 1 to n
                if A[i] = A[j] and i ≠ j return false
return true
```

Average case?
Quit halfway through, $O(n*n/2)$
Still $O(n^2)$
Often, Average case = Worst case.

Analyze this…

```
AllUnique(A[1..n])
for I = 1 to n
        for j = i to n
                if A[i] = A[j] return false
return true
```

Analyze this…

```
AllUnique(A[1..n])
for I = 1 to n
        for j = i to n
                if A[i] = A[j] return false
return true
```

$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1 = n(n+1)/2 \Rightarrow$ Still $O(n^2)$

Extra Material

An Example: Insertion Sort

30	*10*	*40*	*20*
1	2	3	4

$i = \varnothing \quad j = \varnothing \quad key = \varnothing$

$A[j] = \varnothing \qquad A[j+1] = \varnothing$

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

An Example: Insertion Sort

30	10	40	20
1	2	3	4

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

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

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2 \quad j = 1 \quad key = 10$

$A[j] = 30 \quad A[j+1] = 30$

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

An Example: Insertion Sort

30	30	40	20
1	2	3	4

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

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

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

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

An Example: Insertion Sort

30	30	40	20
1	2	3	4

$i = 2 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 30$

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

i = 2 j = 0 key = 10

A[j] = ∅ A[j+1] = 10

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

An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

$i = 3 \quad j = 0 \quad key = 10$

$A[j] = \varnothing \quad A[j+1] = 10$

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 0 \quad key = 40$

$A[j] = \varnothing \quad A[j+1] = 10$

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

An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

$i = 3 \quad j = 0 \quad key = 40$

$A[j] = \varnothing \quad A[j+1] = 10$

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

$i = 3 \quad j = 2 \quad key = 40$

$A[j] = 30 \quad A[j+1] = 40$

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

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

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

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

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

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

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

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

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

An Example: Insertion Sort

10	*30*	*40*	*20*
1	2	3	4

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

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

An Example: Insertion Sort

10	30	40	20
1	2	3	4

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

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

An Example: Insertion Sort

1	2	3	4
10	*30*	*40*	*20*

$i = 4 \quad j = 3 \quad key = 20$

$A[j] = 40 \quad A[j+1] = 20$

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

An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4 \quad j = 3 \quad key = 20$

$A[j] = 40 \quad A[j+1] = 40$

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

An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4 \quad j = 3 \quad key = 20$

$A[j] = 40 \quad A[j+1] = 40$

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

An Example: Insertion Sort

10	30	40	40
1	2	3	4

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

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

An Example: Insertion Sort

10	*30*	*40*	*40*
1	2	3	4

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

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

An Example: Insertion Sort

10	30	40	40
1	2	3	4

$i = 4 \quad j = 2 \quad key = 20$

$A[j] = 30 \quad A[j+1] = 40$

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

An Example: Insertion Sort

1	2	3	4
10	*30*	*30*	*40*

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

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

An Example: Insertion Sort

10	30	30	40
1	2	3	4

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

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

An Example: Insertion Sort

10	30	30	40
1	2	3	4

$i = 4 \quad j = 1 \quad key = 20$

$A[j] = 10 \quad A[j+1] = 30$

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

An Example: Insertion Sort

1	2	3	4
10	*30*	*30*	*40*

$i = 4 \quad j = 1 \quad key = 20$

$A[j] = 10 \quad A[j+1] = 30$

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

An Example: Insertion Sort

10	20	30	40
1	2	3	4

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

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

An Example: Insertion Sort

10	20	30	40
1	2	3	4

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

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

Done!