Quadratic-time sorting Overview		
Sorting		
Sorting Problem		
Input: An array of <i>comparable</i> elements		
<b>Output</b> : The same elements, sorted in ascending order		
• One of the most well-studied algorithmic problems		
• Use late of most well-studied algorithmic problems		
Has lots of practical applications		
You should already know a few algorithms		
SI 335 (USNA) Unit 2	Spring 2015	1 / 21
Quadratic-time sorting Overview		

## SelectionSort

```
def selectionSort(A):
    for i in range(0, len(A)-1):
        m = i
        for j in range(i+1, len(A)):
            if A[j] < A[m]:
                m = j
            swap(A, i, m)
</pre>
```

SI 335 (USNA)

Unit 2

Spring 2015 2 / 21

```
Quadratic-time sorting Overview

InsertionSort

def insertionSort(A):

    for i in range(1, len(A)):

        j = i - 1

    while j >= 0 and A[j] > A[j+1]:

        swap(A, j, j+1)

        j = j - 1

    Stas (USNA) Unic 2 Spring 2015 3/21
```

Common Features It's useful to look for larger patterns in <b>algorithm design</b> . Both InsertionSort and SelectionSort build up a sorted array one element at a time, in the following two steps:					
It's useful to look for larger patterns in <b>algorithm design</b> . Both InsertionSort and SelectionSort build up a sorted array one element at a time, in the following two steps:					
It's useful to look for larger patterns in <b>algorithm design</b> . Both InsertionSort and SelectionSort build up a sorted array one element at a time, in the following two steps:					
It's useful to look for larger patterns in <b>algorithm design</b> . Both InsertionSort and SelectionSort build up a sorted array one element at a time, in the following two steps:					
Both InsertionSort and SelectionSort build up a sorted array one element at a time, in the following two steps:					
• <b>Pick</b> : Pick an element in the unsorted part of the array					
• <b>Place</b> : Insert that element into the sorted part of the array					
For both algorithms, one of these is "easy" (constant time) and the other is "hard" $(O(n)$ time). Which ones?					
SL335 (USNA) Unit 2 Spring 2015 4 / 21					
Quadratic-time sorting Loop analysis with summations					
Analysis of SelectionSort					
Each loop has $O(n)$ iterations, so the total cost is $O(n^2)$ .					
What about a big- $\Theta$ bound?					
SI 335 (USNA) Unit 2 Spring 2015 5 / 21					
Quadratic-time sorting Loop analysis with summations					

Arithmetic Series An arithmetic series is one where consecutive terms differ by a constant. General formula:  $\sum_{i=0}^{m} (a + bi) = \frac{(m+1)(2a + bm)}{2}$ So the worst-case of SelectionSort is This is  $\Theta(n^2)$ , or **quadratic time**.

SI 335 (USNA)

Unit 2

	Quadratic-time sorting	Worst-case family of examples		
Worst-Case Family	V			
-				
Why can't we analyze	InsertionSort in	n the same way?		
We need a <b>family of e</b> the worst case.	examples, of a	rbitrarily large size, tha	at demonsti	rate
Worst-case for Insertio	nSort:			
Worst-case cost:				
SI 335 (USNA)	Un	it 2	Spring 2015	7 / 21
	Quadratic-time sorting	Worst-case family of examples		
SelectionSort (Recurs	sive Version)			
def selectionSort	tRec(A, sta	rt=0):		
if (start < 1	len(A) - 1)	:		
$m - minT_{1}$	adov (A ata	rt )		

```
m = minIndex(A, start)
```

```
swap(A, start, m)
selectionSortRec(A, start + 1)
```

minIndex

```
def minIndex(A, start=0):
    if start >= len(A) - 1:
        return start
    else:
        m = minIndex(A, start+1)
        if A[start] < A[m]:
            return start
        else:
            return m
SI 335 (USNA) Unit 2 Spring 2015 8/21
```

Quadratic-time sorting Recursive analysis

## Analysis of minIndex

Let T(n) be the worst-case number of operations for a size-*n* input array.

We need a **recurrence relation** to define T(n):

$$T(n) = \begin{cases} 1, & n \le 1 \\ 4 + T(n-1), & n \ge 2 \end{cases}$$

Solving the recurrence:

SI 335 (USNA)

Unit 2

	and a such sta	
Quadratic-time sorting rect	ursive analysis	
Analysis of recursive SelectionSo	ort	
Let $S(n)$ be the worst-case for SelectionS	Sort	
What is the recurrence?		
SI 335 (USNA) Unit 2	Spring 2015 10	/ 21
MergeSort Para	adigm	
Divide and Conquer		
A new Algorithm Design Paradigm: D	ivide and Conquer	
Works in three steps:		
<ol> <li>Break the problem into similar subpr</li> </ol>	roblems	
② Solve each of the subproblems recur	sively	
③ Combine the results to solve the orig	ginal problem.	
MergeSort and BinarySearch both follow (How do they approach each step?)	this paradigm.	
SI 335 (USNA) Unit 2	Spring 2015 11	/ 21
MergeSort Para	adigm	
MergeSort		
def mergeSort(A):		
if len(A) <= 1:		

mergeSort(A): if len(A) <= 1: return A else: m = len(A) // 2 B = A[0 : m] C = A[m : len(A)] mergeSort(B) mergeSort(C) A[:] = merge(B, C)

MergeSort Paradigm					
Morgo					
weige					
<pre>def merge(B, C): A = [] i, j = 0, 0 while i &lt; lex if B[i] A.ap i = i else: A.ap j = i while i &lt; lex A.append i = i + while j &lt; lex A.append j = j + return A</pre>	n(B) and j < <= C[j]: pend(B[i]) i + 1 pend(C[j]) j + 1 n(B): (B[i]) 1 n(C): (C[j]) 1	< len(C):			
SI 335 (USNA)	Unit	- 2	Spring 2015 13 / 21		
	MergeSort	Analysis			
Analysis of Merge Each while loop has constant cost.					
So we just need the total number of iterations through every loop.					
	Lower bound	Upper bound	Exact		
Loop 1	min( <i>a</i> , <i>b</i> )	a+b			
Loop 2	0	2			

	(u, ~)	u   2	
Loop 2	0	а	
Loop 3	0	b	
Total	min( <i>a</i> , <i>b</i> )	2(a+b)	

a is the size of A and b is the size of B.

SI 335 (USNA)

Unit 2

Spring 2015 14 / 21

	MergeSort	Analysis		
Analysis of MergeSort				
Analysis of Mergesolt				
SI 335 (USNA)	Un	it 2	Spring 2015	15 / 21

	Lower Bound for Sortin	e		]
Complexity of C	ortin~	•		
Complexity of 2	orting			
Algorithms we have	e seen so far:			
	Sort	Worst-case cost		
	SelectionSort	$\Theta(n^2)$		
	InsertionSort	$\Theta(n^2)$		
	HeapSort	$\Theta(n \log n)$ $\Theta(n \log n)$		
Million dollar que	<b>stion</b> : Can we d	o better than $\Theta(n)$	og n)?	
			Carrier 2015	16 / 21
51 335 (USINA)		Unit 2	Spring 2015	10 / 21
	Lower Bound for Sortin	g		
Comparison Mc	del			
Elements in the inp	out array can only	y be accessed in tw	o ways:	
<ul> <li>Moving them (</li> </ul>	(swap, copy, etc.	)		
<ul> <li>Comparing two</li> </ul>	o of them (<, >	, =, etc.)		
Every sorting algor	ithm we have se	en uses this model.		
It is a very <b>general</b> anything else.	model for sortir	ng strings or integer	s or floats or	
What operations ar	e <i>not</i> allowed in	this model?		
·				
SI 335 (USNA)		Unit 2	Spring 2015	17 / 21
				1
	Lower Bound for Sortin	g		
Permutations	Lower Bound for Sortin	g		
Permutations	Lower Bound for Sortin	g		
Permutations	Lower Bound for Sortin	g		
Permutations How many ordering	Lower Bound for Sortin	sions) are there of n	elements?	
Permutations How many ordering <i>n</i> factorial, written	Lower Bound for Sortin is (aka $permutat$ n! = n  imes (n-1)	ions) are there of $n$ ) $ imes$ ( $n-2$ ) $ imes$ $\cdots$ $ imes$	elements? $2 \times 1.$	
Permutations How many ordering <i>n</i> factorial, written <b>Observation</b> : A co <i>A</i> , not the actual co	Lower Bound for Sortin n! = n  imes (n-1) omparison-based ontents.	ions) are there of $n)  imes (n-2)  imes \cdots  imessort is only sensitive$	elements? 2  imes 1. e to the <b>orderin</b>	ng of
Permutations How many ordering <i>n</i> factorial, written <b>Observation</b> : A co <i>A</i> , not the actual co For example, Merge [1,2,4,3], [34,35]	Lower Bound for Sortin as (aka <i>permutat</i> $n! = n \times (n - 1)$ apparison-based ontents. eSort will do the ,37,36], or [10,	$\vec{s}$ $\vec{s}$ ions) are there of $n$ $) \times (n - 2) \times \cdots \times$ sort is only sensitive same things on 20,200,99].	elements? $2 \times 1$ . e to the <b>orderir</b>	<b>ıg</b> of
Permutations How many ordering <i>n</i> factorial, written <b>Observation</b> : A co <i>A</i> , not the actual co For example, Merge [1,2,4,3], [34,35]	Lower Bound for Sortin $n! = n \times (n - 1)$ comparison-based ontents. eSort will do the ,37,36], or [10,	Final set $(n-2) \times \cdots \times$ sort is only sensitive same things on 20,200,99].	elements? 2  imes 1. e to the <b>orderir</b>	ıg of

Unit 2

Lower B	ound for Sorting					
Logarithms						
Recall some useful facts about logarithms:						
• $\log_b b = 1$						
• $\log_b ac = \log_b a + \log_b a$	ь С					
• $\log_b a^c = c \log_b a$	-					
• $\log_b a = (\log_c a)/(\log_c a)$	<i>b</i> )					
Now how about a lower bo	und on lg <i>n</i> !?					
SI 335 (USNA)	Unit 2	Spring 2015 19 / 21				
Lower B	ound for Sorting					
Lower Bound on Sort	Lower Bound on Sorting					
A correct algorithm m possible input permut.	ust take different action ations.	s for each of the				
② The choice of actions is determined only by comparisons.						
③ Each comparison has two outcomes.						
④ An algorithm that performs c comparisons can only take 2 <sup>c</sup> different actions.						
S The algorithm must perform at least lg n! comparisons.						
Therefore <b>ANY comparison-based sort is</b> $\Omega(n \log n)$						
SI 335 (USNA)	Unit 2	Spring 2015 20 / 21				
Conclusions						
Any sorting algorithm that only uses comparisons must take at least $\Omega(n \log n)$ steps in the worst case.						

- This means that sorts like MergeSort and HeapSort couldn't be much better they are **asymptotically optimal**.
- What if I claimed to have a O(n) sorting algorithm?
   What would that tell you about my algorithm (or about me)?
- Remember what we learned about summations, recursive algorithm analysis, and logarithms.