CSOR 4231
Analysis of Algorithms I

Nakul Verma

Algorithms: what and why?

An algorithm is a set of instructions that is designed to accomplish a task

- Central to many topics in Computer Science:
  - Understand the complexity of task and how it will scale with larger inputs
  - Helps one find optimal solutions to problems, rather than relying on brute force or less efficient methods.
  - Critical in applications where performance and resource constraints are important.

- Heavily used in sciences and engineering to design efficient solutions

Algorithms in Practice

- **Sorting and Searching**: Efficient algorithms are fundamental for database management, information retrieval, and indexing
- **(Security) Cryptography**: Algorithms are crucial for securing data through encryption, hashing, and other cryptographic techniques
- **(AI) Neural Networks**: Algorithms for training and deploying AI models
- **(Finance) Trading Algorithms**: High-frequency trading and automated trading strategies
- **(Robotics) Path Planning**: Algorithms for navigating robots through environments
- **(Logistics) Optimization Algorithms**: Used for optimizing routes, schedules, and resource allocation

This course

We will learn:

- The theory and practice of algorithms
- Key principles and paradigms for designing successful algorithm design
- How to analyze the correctness and efficiency of an algorithm

Administrivia

Website: `http://www.cs.columbia.edu/~verma/classes/alg/`

The team:

Instructor: Nakul Verma (me)

TAs

Students: you!

Evaluation:

- Homeworks (0%)
  - For practice only – do as many or as few as you need to get good at problem solving!
- 3x Exams (33.3% each)

Discussion Board: Ed Stem (visit regularly!)

- Used for all class related announcements, questions and discussions
- Do not email the course staff directly!

Announcement!

- Visit the course website
- Review the basics (prerequisites)
- See all the announcements already posted on EdStem!

Let’s get started!

Acknowledgements first…

These slides are adapted from other excellent Algorithms courses offered from various sources.

Most notably:

- Algorithms course by Prof. Eleni Drinea (Columbia)
- Algorithms course by Prof. Christos Papadimitriou (Columbia)
- Algorithms course at UC San Diego

Algorithms: a formal definition

- An algorithm is a **well-defined** computational procedure that transforms the **input** (a set of values) into the **output** (a new set of values).

- The desired input/output relationship is specified by the statement of the **computational problem** for which the algorithm is designed.

- An algorithm is correct if, for *every input*, it **halts** with the correct output.

Efficient Algorithms

- In this course we are interested in algorithms that are correct and efficient.

- Efficiency is related to the resources an algorithm uses: time, space
  - How much time/space are used?
  - How do they scale as the input size grows?

We will primarily focus on efficiency in **running time**.

Running time

- Running time = number of primitive computational steps performed; typically these are

    - arithmetic operations: add, subtract, multiply, divide *fixed-size* integers
    - data movement operations: load, store, copy
    - control operations: branching, subroutine call and return

- We are typically interested in **worst case** running time, that is, on any input what is the maximum number of steps (ever) it will take to produce the correct solution by our algorithm
    - Other interesting cases: **best case** and **average case** running times.

Algorithm Presentation: Pseudocode

- We will use pseudocode for our algorithm descriptions.

  A high-level description of an algorithm that combines the structure of programming languages with the readability of natural language.

  - **Simplicity and Clarity**: focuses on the logic of the algorithm without getting bogged down by the syntax of a specific programming language. This makes it easier to understand and communicate complex ideas.

  - **Language-Agnostic**: can be understood by people with different programming backgrounds. This makes it versatile for explaining algorithms to a broader audience.

Good Pseudocode	Real Code (in Java)	Bad Pseudocode

method doMathHomework():
Get pencil
Open textbook and notebook
 	
Go through all the problems:
Complete problem
while the problem is wrong:
Try again
 	
Clean up your desk
Submit your homework
	public void doMathHomework(){
	this.getPencil();
	_textBk.open();
	_noteBk.open();
	
	for(int i = 0; i < _problems.length(); i++){
	_problems[i].solve();
	while(!_problems[i].isRight()){
	this.eraseSolution(i);
	_problems[i].solve();
	
	this.cleanDesk();
	this.submit();
	}
		method doMathHomework():
		Get things for homework
		
		Do the problems correctly
		
		Finish the homework

Our first algorithm: Insertion Sort

- Task: Sorting a list of integers

Input: A list A of n integers $x_1, \ldots, x_n$
Output: A permutation $x_1', x_2', \ldots, x_n'$ of the n integers where they are sorted in non-decreasing order, i.e., $x_1' \leq x_2' \leq \ldots \leq x_n'$

Example:

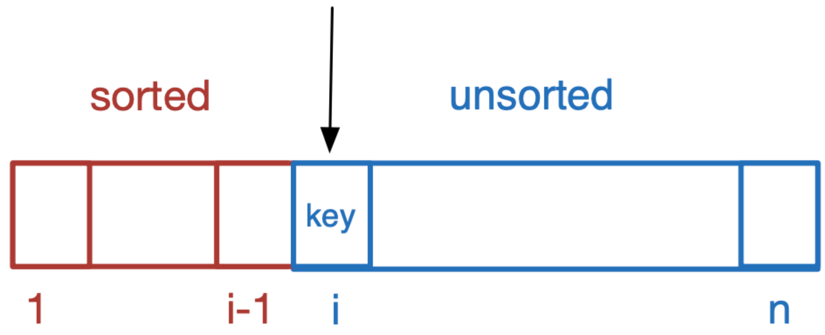
- Input: n = 6, A = (9,3,2,6,8,5)
- Output: A = (2,3,5,6,8,9)

What data structure should we use to represent the list?

Array: collection of items of the same data type

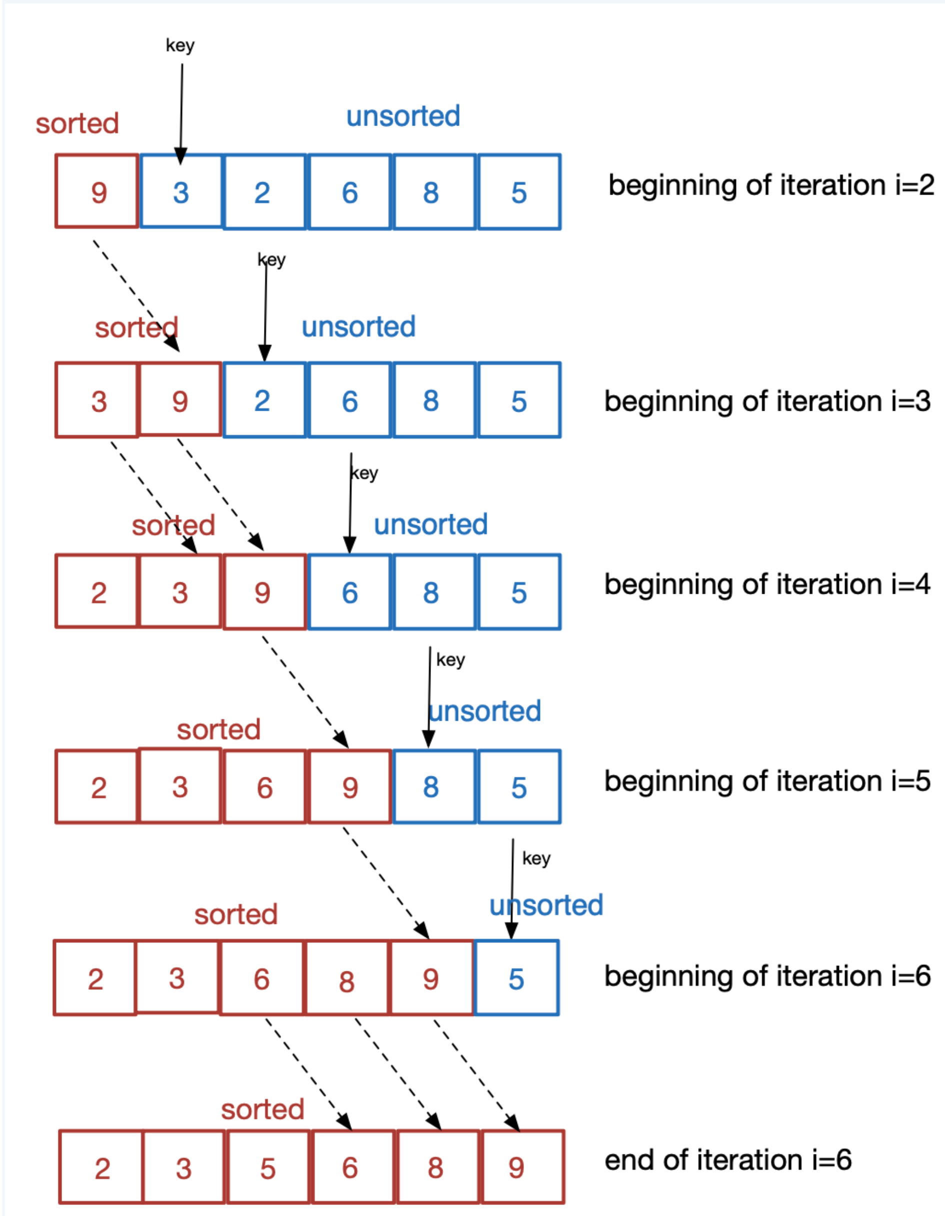
- allows for random access
- “zero” indexed in C++ and Java; in algorithms/pseudocode we usually do “one” index

Insertion Sort: Main Idea



1. Start with a (trivially) sorted subarray of size 1 consisting of A[1].
2. Increase the size of the sorted subarray by 1, by inserting the next element of A, call it key, in the **correct** position in the **sorted** subarray to its left. *How?*
   - Compare key with every element x in the sorted subarray to the left of key, starting from the right.
     - If x > key, move x one position to the right.
     - If x ≤ key, insert key after x.
3. Repeat Step 2. until the sorted subarray has size n.

Insertion Sort on example input



Insertion Sort: Pseudocode

Let A be an array of n integers.

```
insertion-sort(A)
  for i = 2 to n do
      key = A[i]
      //Insert A[i] into the sorted subarray A[1, i − 1]
      j = i − 1
      while j > 0 and A[j] > key do
          A[j + 1] = A[j]
          j = j − 1
      end while
      A[j + 1] = key
  end for
```

Insertion Sort: Analysis

- Correctness

  formal proof often by induction

- Running time

  number of primitive computational steps

  - Not the same as **time** it takes to execute the algorithm
  - We want a measure that is independent of hardware
  - We want to know how running time scales with the size of the input.

- Space requirements:

  how much space is required by the algorithm

Recall: Induction

Fact: For all $n \geq 1$, we have that: $1+2+ \ldots + n = n\,(n+1)\,/\,2$

Proof: (by Induction)

Base case: $n = 1$

Inductive Hypothesis: Assume that the statement is true for $n \geq 1$, ie,

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

Inductive Step: Show that assuming the inductive hypothesis, the statement is true for the case $n+1$, i.e., need to show:

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

Conclusion: It follows that the statement is thus true for all $n \geq 1$!

Insertion Sort: Correctness

Notation: Let A[i,j] be the subarray of A that starts at position i and ends at position j.

Minor change in the pseudocode: in line 1, start from i=1 (rather than i=2)

How does this change affect the algorithm?

Claim: Let $n \geq 1$ be a positive integer. For all $1 \leq i \leq n$, after the i-th loop iteration, the subarray A[1, i] is sorted.

The correctness of insertion sort follows from the claim!

Proof of Claim

By induction on i.

- **Base case:** i = 1, trivial.
- **Induction hypothesis:** assume the statement is true for some $1 \le i < n$.

  ie, assume for some i, A[1,i] is sorted (after the i-th iteration)
- **Inductive step:** Show it is true for i + 1.

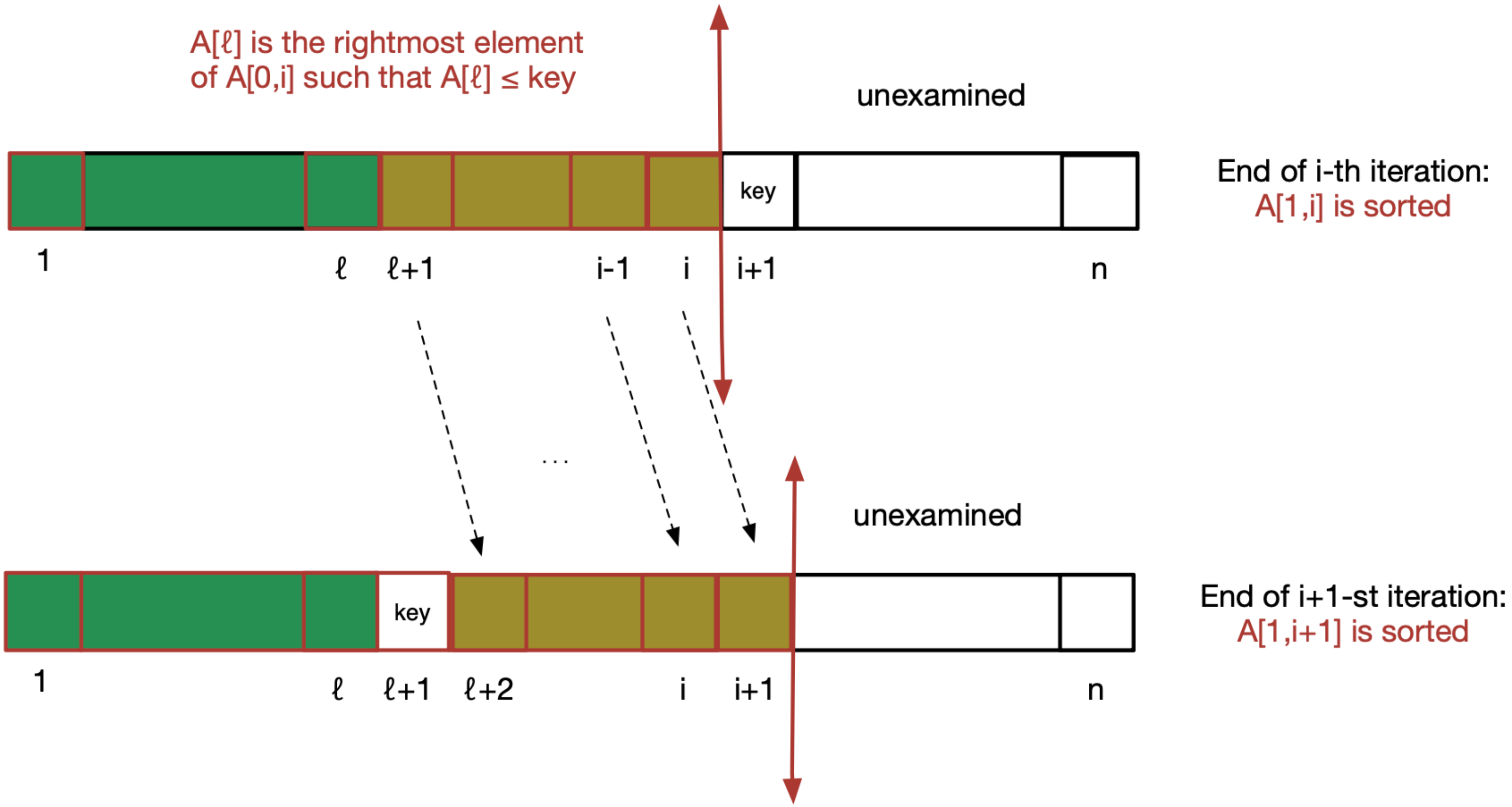
  In loop i + 1, element key = A[i + 1] is inserted into A[1, i].

  By the induction hypothesis, A[1, i] is sorted. Since

  1. key is inserted after the last element $A[\ell]$ such that $0 \le \ell \le i$ and $A[\ell] \le$ key;
  2. all elements in $A[\ell + 1, j]$ are pushed one position to the right with their order preserved,

  Hence, the statement is true for i + 1.

Proof of Claim



Insertion Sort: Runtime

```
for i = 2 to n do
    key = A[i]
    //Insert A[i] into the sorted subarray A[1, i − 1]
    j = i − 1
    while j > 0 and A[j] > key do
        A[j + 1] = A[j]
        j = j − 1
    end while
    A[j + 1] = key
end for
```

Let T(n) be the running time of the algorithm on an input of size n.

- How many primitive computational steps are executed by the algorithm?
- Equivalently, what is the running time T(n)? Bounds on T(n)?

Insertion Sort: Runtime

for $i = 2$ to n **do**	line 1
$\quad$ key $= A[i]$	line 2
$\quad$ //Insert $A[i]$ into the sorted subarray $A[1, i-1]$	
$\quad$ $j = i - 1$	line 3
$\quad$ **while** $j > 0$ and $A[j] >$ key **do**	line 4
$\quad\quad$ $A[j+1] = A[j]$	line 5
$\quad\quad$ $j = j - 1$	line 6
$\quad$ **end while**	
$\quad$ $A[j+1] =$ key	line 7
end for	

For $2 \leq i \leq n$, let t_i = #times line 4 is executed. Then

$$T(n) = n + 3(n-1) + \sum_{i=2}^{n} t_i + 2\sum_{i=2}^{n}(t_i - 1) = 3\sum_{i=2}^{n} t_i + 2n - 1$$

- Which (size n) input yields the smallest (best-case) running time?
- Which (size n) input yields the largest (worst-case) running time?

Worst case analysis

Worst-case running time: largest possible running time of the algorithm over all inputs of a given size n.

Why worst-case analysis?

- It gives well-defined computable bounds.
- Average-case analysis can be tricky: how do we generate a “random” instance?

The worst-case running time of insertion-sort is quadratic.

So… is insertion-sort efficient?