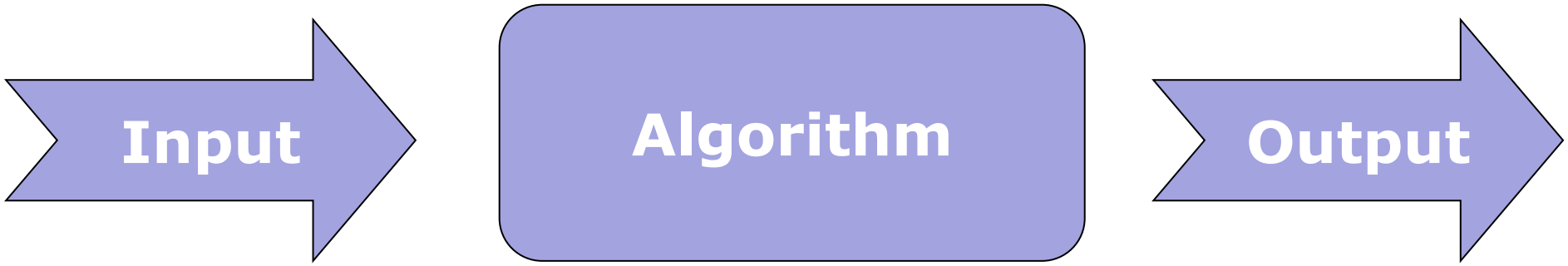
DTS203TC
Design and Analysis of Algorithms

Lecture 1: Getting Started

Dr. Qi Chen
School of AI and Advanced Computing

What is an algorithm?

- An algorithm is a sequence of computational steps that transform the input into the output.



- We can also view an algorithm as a tool for solving a well-specified computational problem.
- Daily life examples: cooking recipe

Algorithm vs. Program

- Algorithm
  - Design
  - Domain Knowledge
  - Any language
  - Hardware & OS
  - Analyze
- Program
  - Implementation
  - Programmer
  - Programming Language
  - Hardware & OS
  - Testing

Some Well-known Algorithms

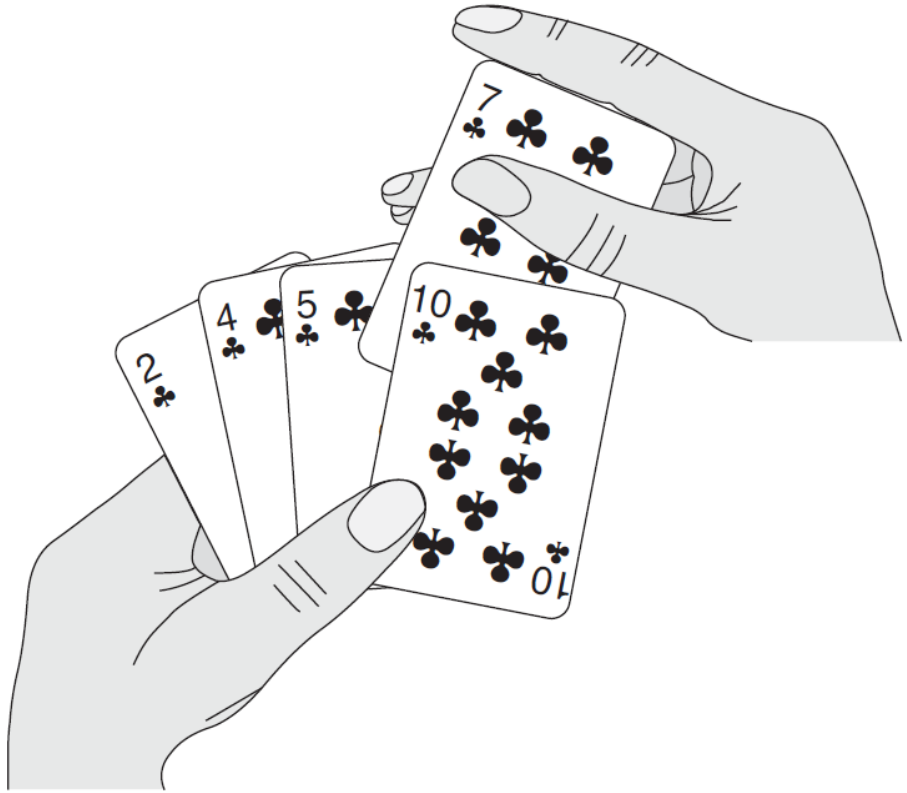
- Sorting
  - Insertion sort
  - Merge sort
- Searching
- Graph algorithms
  - Minimum Spanning Trees
  - Maximum Flow
- String matching
  - The Rabin-Karp algorithm
  - The Knuth-Morris-Pratt algorithm
- Number-Theoretic Algorithms
  - The RSA public-key cryptosystem

Sorting

- Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- Output: A reordering $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Example:
  - Input: <8,2,4,9,3,6>
  - Output: <2,3,4,6,8,9>

Insertion sort

- Sorting a hand of cards using insertion sort

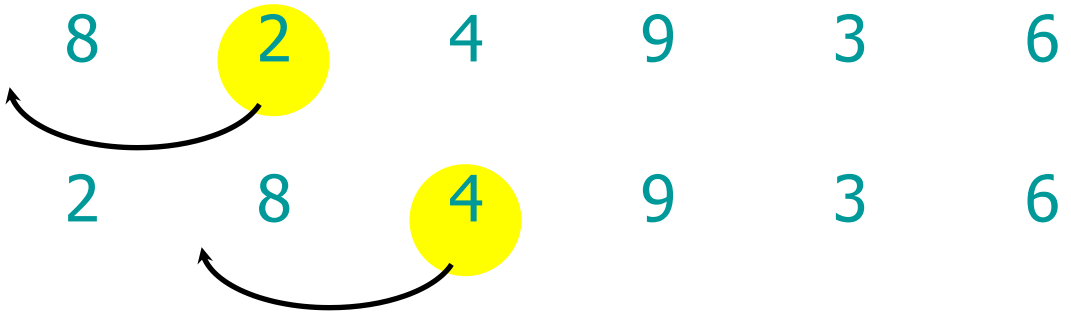


Insertion sort – cont’d

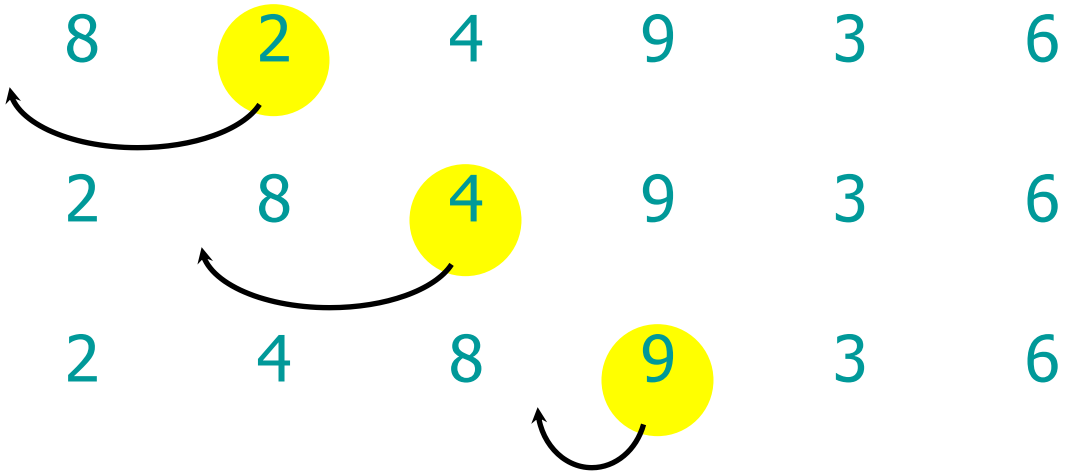
8 2 4 9 3 6

Insertion sort – cont’d

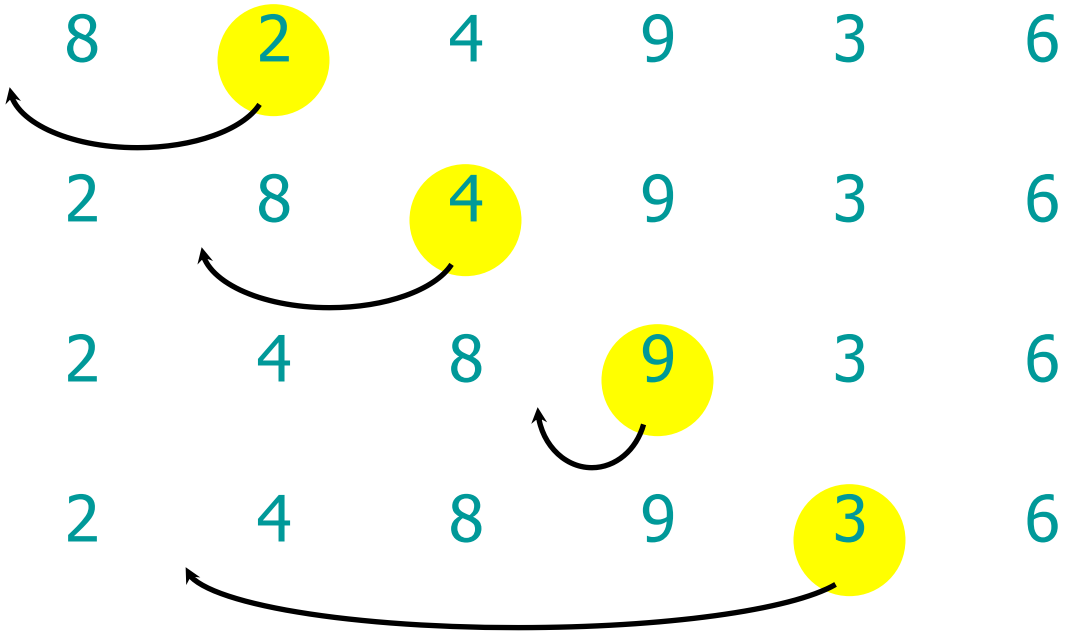
Insertion sort – cont'd



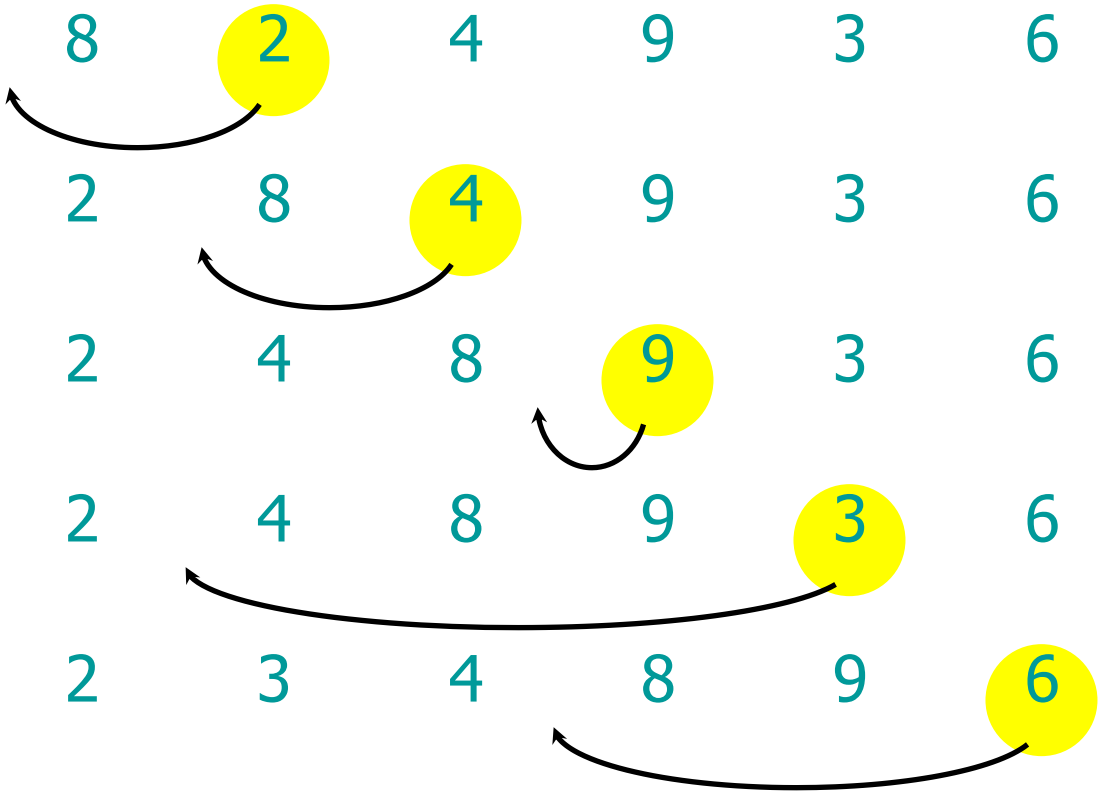
Insertion sort – cont'd



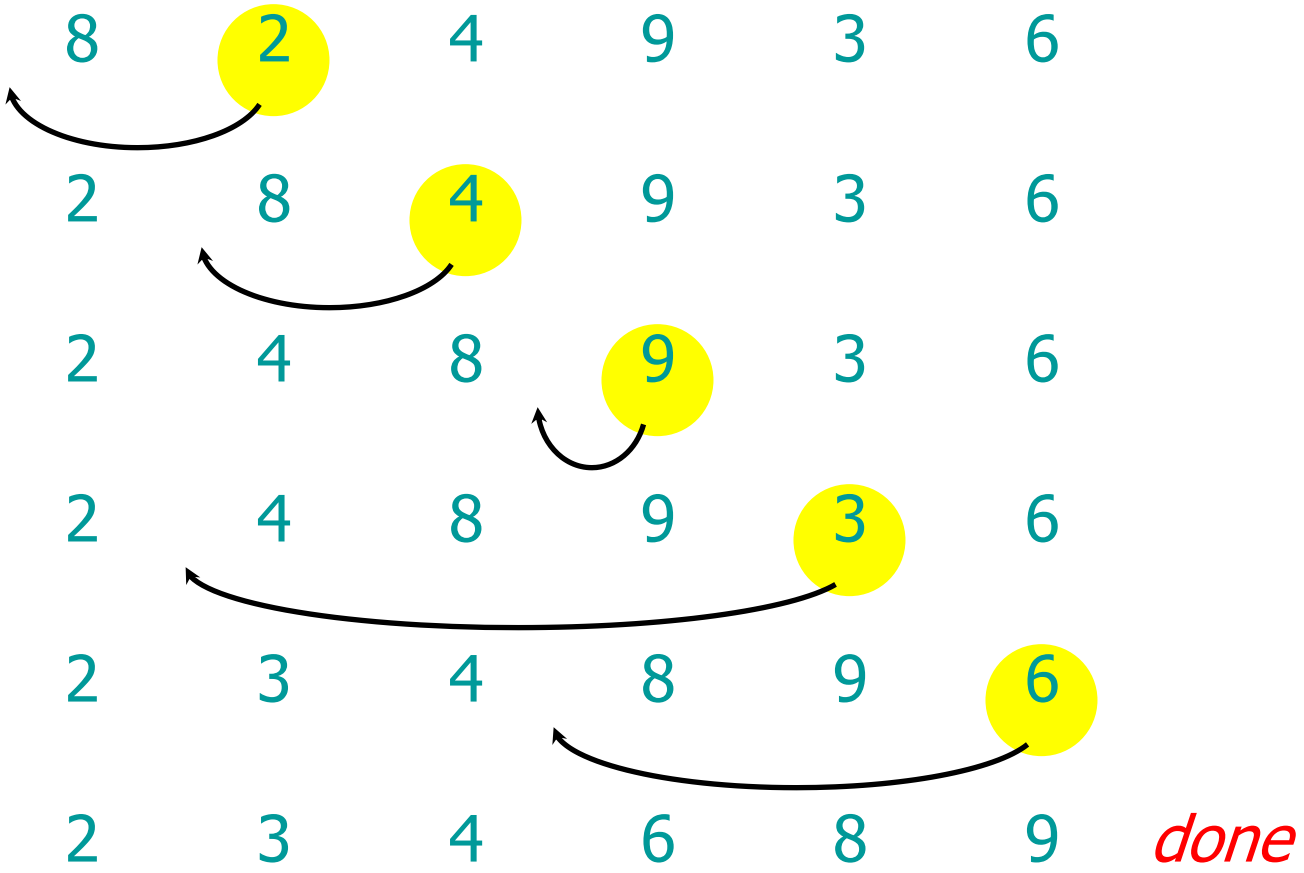
Insertion sort – cont'd



Insertion sort – cont'd



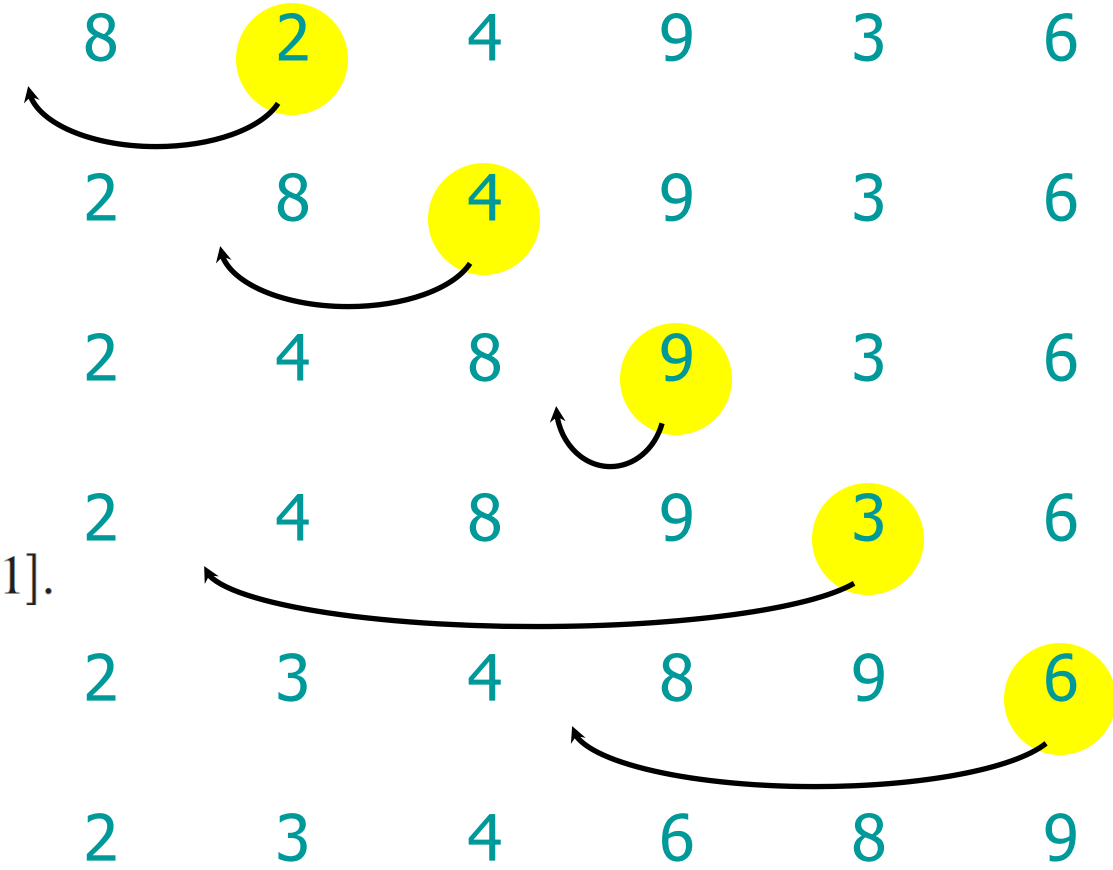
Insertion sort – cont'd

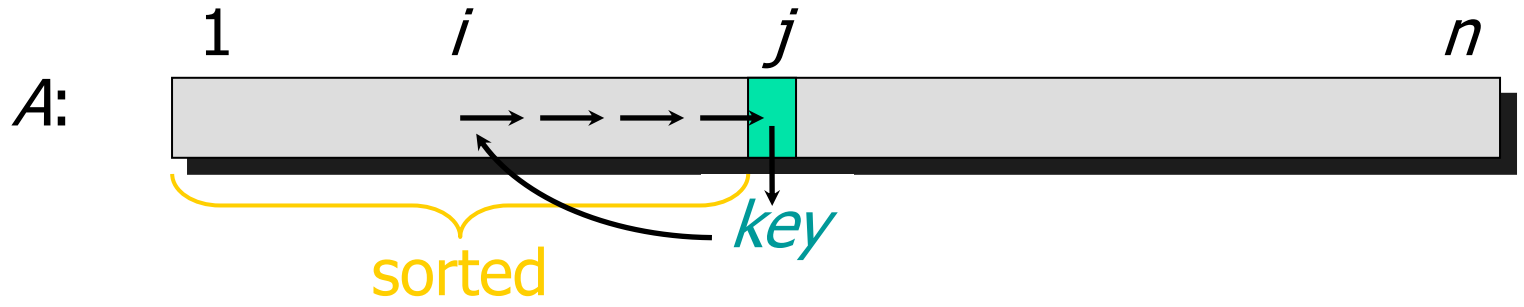


Insertion sort – cont'd

- pseudocode

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```





Analysis of Algorithms

- Proof of correctness: show that the algorithm gives the desired result
- Time complexity analysis: find out how fast the algorithm runs
- Space complexity analysis: decide how much memory space the algorithm requires
- Look for improvement: can we improve it to run faster or use less memory?

Mathematical Induction

- Mathematical induction: a mathematical technique to prove that a statement holds for every natural number n=0,1,2,….

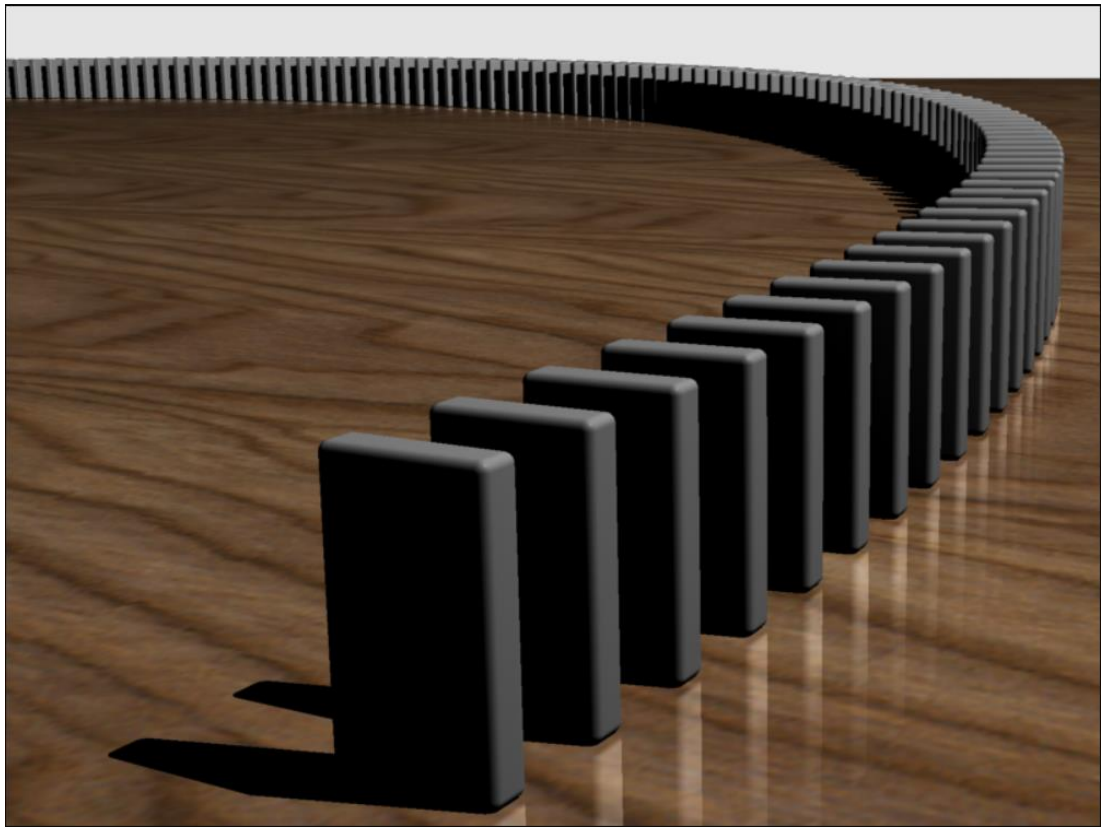
- For example, to prove $1+2+\ldots+n = n(n+1)/2$ $\forall$ integers $n \geq 1$
  - when n is 1, L.H.S = 1, R.H.S = 1*2/2 = 1 **OK!**
  - when n is 2, L.H.S = 1+2 = 3, R.H.S = 2*3/2 = 3 **OK!**
  - when n is 3, L.H.S = 1+2+3 = 6, R.H.S = 3*4/2 = 6 **OK!**

However, none of these constitute a proof and we cannot enumerate over all possible numbers.

=> Mathematical Induction

Mathematical induction – cont'd

- Mathematical induction can be informally illustrated by reference to the sequential effect of falling dominoes.
- If the first domino falls, then the second domino falls. If the second domino falls, then the third domino will fall too. And so on.
- Conclusion: If the first domino falls, then any n, nth domino falls.



Mathematical Induction Examples

- To prove: $1+2+\ldots+n = n(n+1)/2$ $\forall$ integers $n \geq 1$
- Base case: When n=1, L.H.S = 1, R.H.S = 1*2/2=1. Therefore, the statement is true for n=1.
- Induction hypothesis: Assume that statement is true when n=k for some integer $k \geq 1$.
  - i.e., assume that $1+2+\ldots+k = k(k+1)/2$
- Induction step: When n=k+1,
  - L.H.S = 1+2+...+k+(k+1) = (k^2+3k+2)/2
  - R.H.S = (k+1)((k+1)+1)/2 = (k^2+3k+2)/2 = L.H.S

Mathematical Induction Examples – Cont'd

- We have proved
  - statement true for n=1
  - If statement is true for n=k, then also true for n=k+1
- In other words,
  - true for n=1 implies true for n=2 (induction step)
  - true for n=2 implies true for n=3 (induction step)
  - true for n=3 implies true for n=4 (induction step)
  - and so on ……
- Conclusion: true for all integers n

Question

n! = n(n-1)(n-2) … *2*1

- Use Mathematical Induction to prove $2^n < n!$ ∀ integers $n \geq 4$.

Mathematical Induction Examples – Cont'd

- To prove $2^n <$ n! ∀ integers n ≥ 4.
- Base case: When n=4, L.H.S = 16, R.H.S = 4! = 4*3*2*1 = 24, L.H.S < R.H.S. So, statement true for n=4
- Induction hypothesis: Assume that statement is true for some integer k ≥ 4, i.e., assume $2^k <$ k!
- Induction step: When n=k+1
  - L.H.S = 2^{k+1} = 2*2k < 2*k! <- by hypothesis
  - R.H.S = (k+1)! = (k+1)*k! > 2*k! > L.H.S <-because k+1>2
  - So, statement true for k+1
- Conclusion: statement true ∀ integers n ≥ 4.

Loop invariants and the correctness of insertion sort

- Back to Insertion Sort, We use loop invariants (Similar to Mathematical Induction) to help us understand why an algorithm is correct.

- loop invariant:
  - Initialization: It is true prior to the first iteration of the loop.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Loop invariants and the correctness of insertion sort – cont'd

- Initialization: When j = 2, the subarray A[1..j-1] consists of just the single element A[1], which shows that the loop invariant holds prior to the first iteration of the loop.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop invariants and the correctness of insertion sort – cont'd

- Maintenance: The body of the for loop works by moving A[j-1], A[j-2], A[j-3] and so on by one position to the right until it finds the proper position for A[j] (lines 4–7), at which point it inserts the value of A[j] (line 8). The subarray A[1…j] then consists of the elements in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Loop invariants and the correctness of insertion sort – cont'd

- Termination: The condition causing the for loop to terminate is that =j =n+1. We have the entire array A[1…n] consists of the elements in sorted order. Hence, the algorithm is correct.

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Complexity Measures

- Why we need to analyze algorithm complexity?
  - Computing time is a bounded resource, and so is space in memory.

- What we need to analyze?
  - Analyzing an algorithm has come to mean predicting the resources that the algorithm requires (computational time, space, power consumption, number of exchanged messages, and so on).

Running Time

- The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed.
- It is convenient to define the notion of step so that it is as machine-independent as possible.
- 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

Running Time – cont'd

- The running time of the algorithm is the sum of running times for each statement executed; a statement that takes time c_i to execute and executes n times will contribute $c_i n$ to the total running time.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Running Time – cont'd

- The total running time T(n) for INSERTION-SORT is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Running Time – cont'd

- The running time also depends on the input: an already sorted sequence (Best-case) is easier to sort.

- Generally, we seek upper bounds (Worst-case) on the running time, to have a guarantee of performance.

best-case for Insertion Sort

- The best case occurs if the array is already sorted, which means $t_j = 1$.

$$T(n) = (c_1+c_2 + c_4 + c_5 + c_8)n - (c_2+c_4 + c_5 + c_8)$$

- It's a linear function of n.

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

Worst-case for Insertion Sort

- The worst case occurs if the array is reverse sorted.

$$T(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

- It is a quadratic function of n.

INSERTION-SORT(A)		*cost*	*times*
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

Worst-case and average-case analysis

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input.

- The average-case running time is the amount of time used by the algorithm, averaged over all possible inputs. The average-case is often roughly as bad as the worst case.

Order of growth

- For Insertion Sort, we expressed the worst-case running time as
  - $T(n) = an^2 + bn + c$
- We consider only the leading term of a formula (e.g., an^2).
- We write that insertion sort has a worst-case time complexity of

$$O(n^2)$$

Time complexity

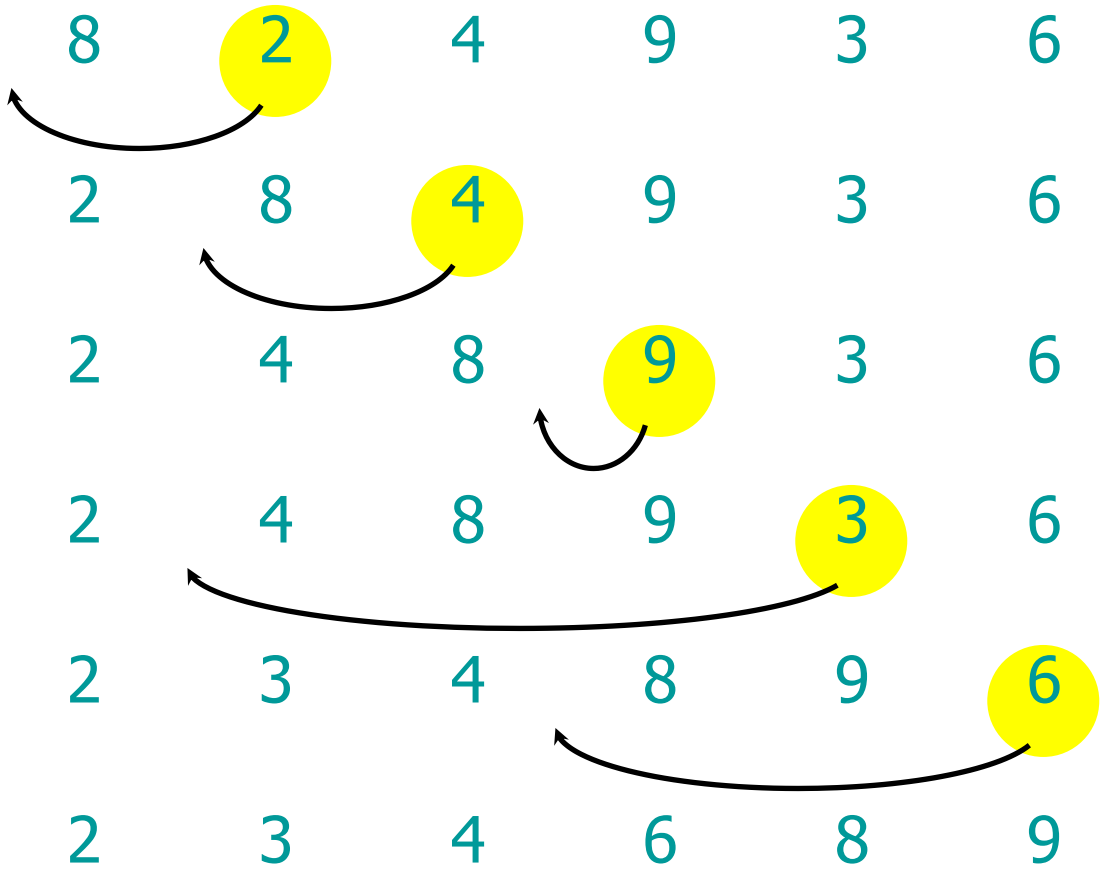
- Time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm.
- We commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size.
- The time complexity is commonly expressed using big O notation
- Insertion sort has a time complexity of $O(n^2)$

Space complexity

- The space complexity of an algorithm is the amount of memory space required to solve an instance of the computational problem.
- Space complexity is often expressed in big O notation.
- Auxiliary space refers to space other than that consumed by the input
- We commonly considers the auxiliary space complexity

What is the space complexity of Insertion sort?

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



Space complexity of Insertion sort

- In-place: An in-place algorithm updates its input sequence only through replacement or swapping of elements
- Only requires a constant amount $O(1)$ of additional memory space
- (Auxiliary) space complexity: $O(1)$

Selection sort

- sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

Selection sort

```
for i = 1 to n-1:
    min = i
    for j = i+1 to n do
        if a[j] < a[min]
            min = j
    swap a[i] and a[min]
```

Selection sort

- Worst-case Time complexity?
- Best-case time complexity?
- Average-case time complexity?
- (Auxiliary) space complexity?

Selection sort

- Worst-case Time complexity: $O(n^2)$
- Best-case time complexity: $O(n^2)$
- Average-case time complexity: $O(n^2)$
- (Auxiliary) space complexity: $O(1)$

Learning outcomes

- Algorithm definition
- Examples of algorithmic problems
- Insertion sort
- Analysis of algorithms
- Mathematical Induction
- Worst-case and average-case time complexity
- Space complexity