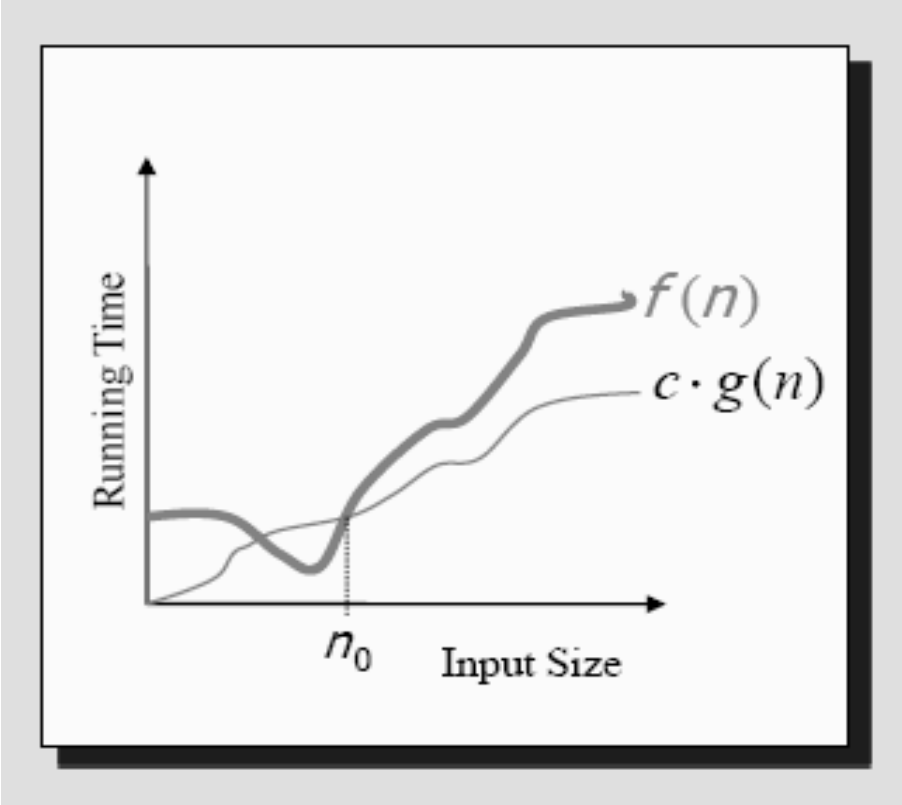
Data Structures

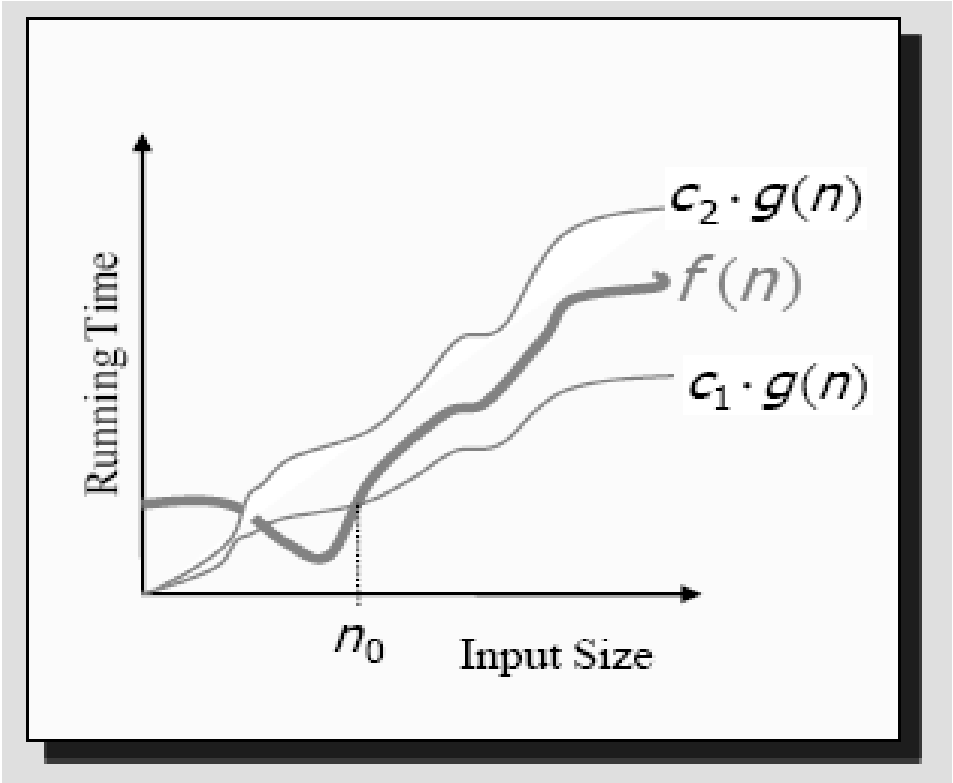
Asymptotic notation

- "Big-Omega" notation (Ω-notation)
  - Asymptotic lower bound on running time
  - f(n) is Ω(g(n)) if there exists constants "c > 0" and "$n_0 \geq 1$", s. t. c g(n) ≤ f(n), for all n ≥n_0.



Asymptotic notation

- "Big-Theta" notation (θ-notation)
  - Asymptotically tight bound
  - f(n) is θ(g(n)) if there exists constant "$c_1 > 0$", "$c_2 > 0$" and "$n_0 \geq 1$" s.t. $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$, for $n \geq n_0$



Asymptotic Notation

- Two more relatives of Big-Oh notation are:
  - “Little-Oh” notation: If f(n) is o(g(n)), then, for every c > 0, there should exist $n_0 > 0$, s.t. $f(n) \leq c\, g(n)$ for $n \geq n_0$
  - “Little-Omega” notation: If f(n) is ω(g(n)), then, for every c > 0, there should exist $n_0 > 0$, s.t. $c\, g(n) \leq f(n)$ for $n \geq n_0$

Importance of Asymptotics

Running Time	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

Stack operations: Running time

Algorithm push(o)

    if size() = N then
        indicate a stack-full error has occurred
    t ←t+1
    S[t] = o

Running time: O(1)

Algorithm pop()

    if(isEmpty()) then
        indicate a stack-empty error has occurred
    e ← S[t] {e is a temporary variable}
    S[t] ← null
    t ← t-1
    return e

Running time: O(1)

Queue Operations: Running time

Algorithm enqueue(e)

    if (size() == N-1)

        raise QueueFull exception

    Q[r] = e

    r ← (r+1) mod N

Running time: O(1)

Algorithm dequeue()

    if isEmpty() then

        raise QueueEmpty Exception

    temp ← Q[f]

    Q[f] = null

    f ← (f+1) mod N

    return temp

Running time: O(1)

List Operations: Running time

Algorithm insert(e, p)

    for i=n-1, n-2, . . ., p do

        S[i+1] ← S[i] {Make room for “e” to be inserted}

    S[p] ← e

    n ← n+1

Running time: O(n)

Algorithm removeAtPosition(p)

    e ← S[p] {e is a temporary variable}

    for i=p, p+1, . . ., n-2 do

        S[i] = S[i+1] {fill in for the removed element}

    n ←n-1

Running time: O(n)

Linked list: Running time

- Assume that a list is implemented using a singly linked list
- The list is maintained in sorted order
- We have reference to the head of the list
- The running time of insert(e) operation?
- The running time of delete(e) operation?

Sorting

- Storing data in an ordered (increasing or decreasing or alphabetical, etc.) manner
- Searching becomes efficient when data is maintained sorted order
- Computer graphics and computational geometry
- A large number of sorting algorithms representing different design techniques have been developed

Sorting Problem

Input: a sequence of n numbers $(a_1, a_2, \ldots, a_n)$

Output: A permutation or reordeing $(a_1', a_2', \ldots, a_n')$ of the input sequence such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

Requirements for the output: output should be a permutation of the input

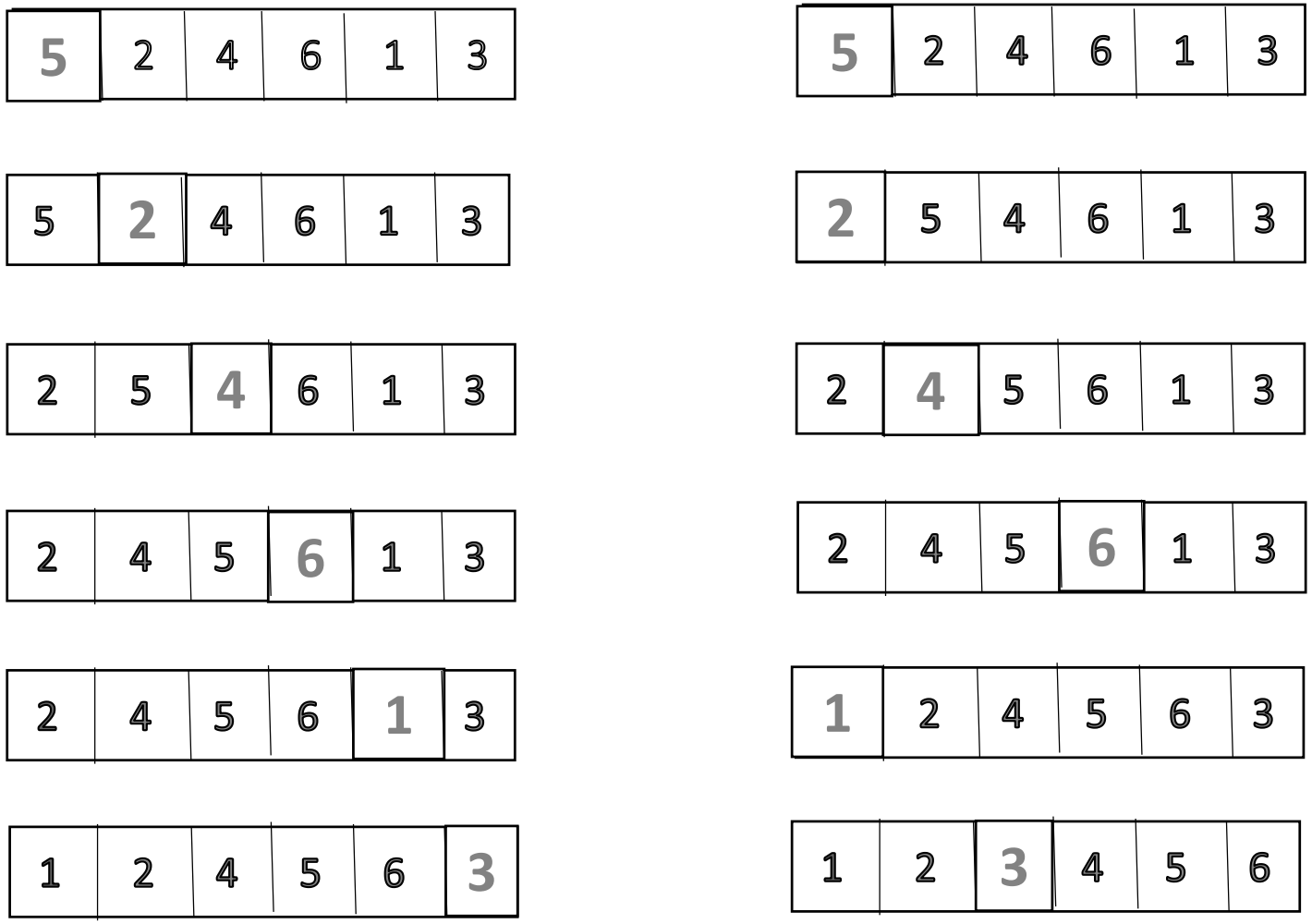
Factors affecting the running time: The input size, how sorted the input sequence is, and the algorithm used

Insertion Sort

- At any point of time, the given sequence can be divided into two parts:
  - Sorted part
  - Unsorted part
- Initially, the sorted part is empty
- Pick the first element from the unsorted part
- Place the picked element in the correct position in the sorted part
- Repeat the same process with the remaining elements

Insertion Sort

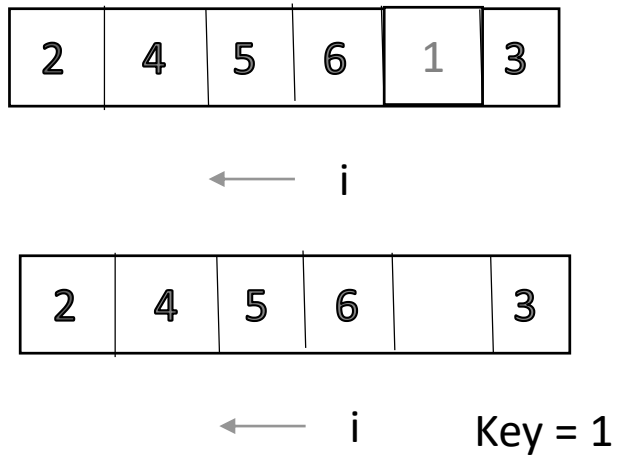
- Strategy



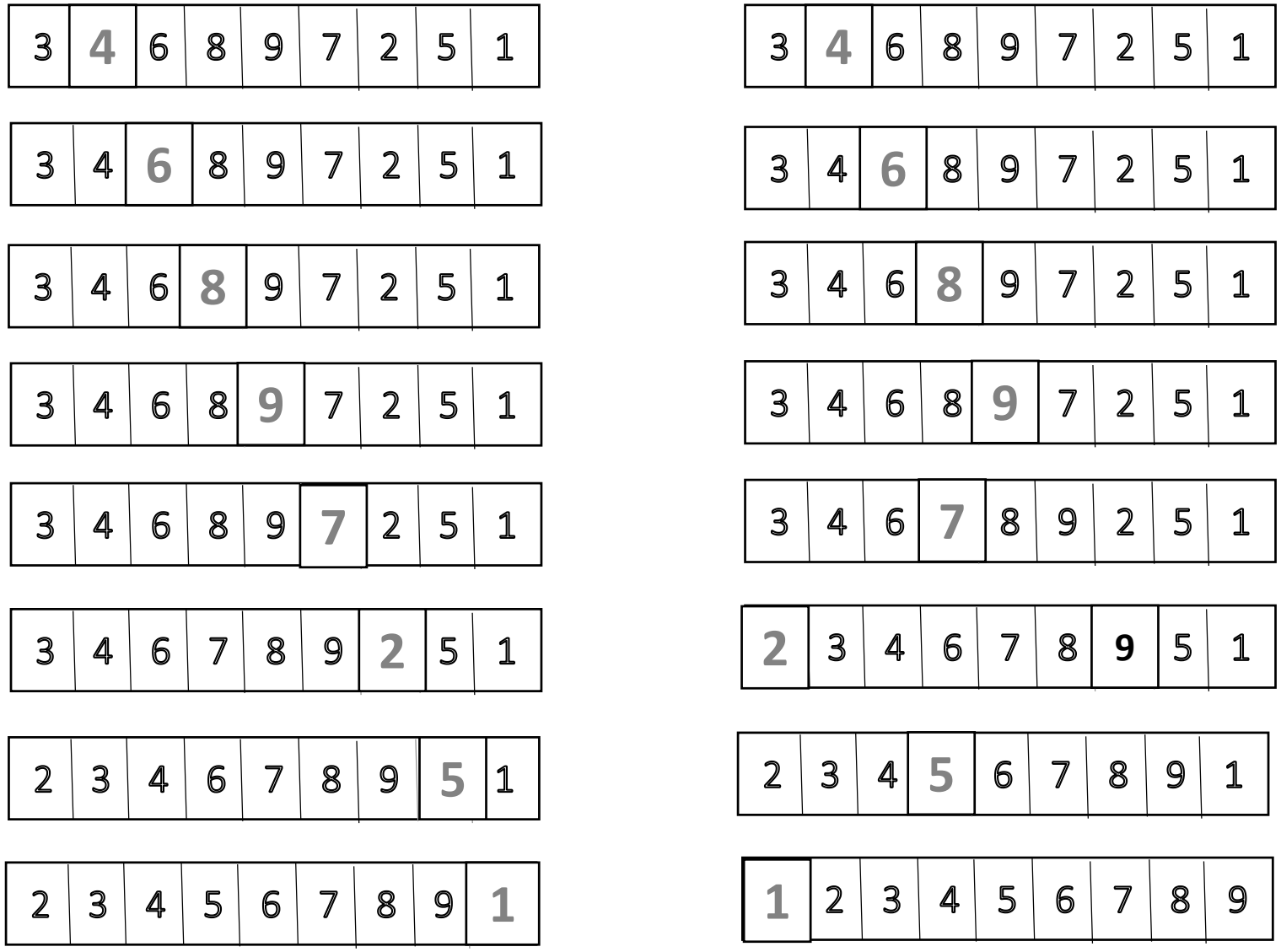
Insertion sort algorithm

Algorithm Insertion_Sort(A)
Input: A[0. .n-1] – an array of integers
Output: a permutation of A such that A[0] ≤ A[1] ≤ . . . ≤ A[n-1]

```
for j ← 1 to n-1 do
        key ← A[j]
        {insert A[j] into the sorted
                sequence A[0. .j-1]}
        i ← j-1
        while i ≥ 0 and A[i] > key do
                A[i+1] ← A[i]
                i - -
        A[i+1] ← key
```



Insertion Sort: Example



Analysis of insertion sort

for j ← 1 to n-1 do	n
key ← A[j]	n-1
{insert A[j] into the sorted sequence A[0. .j-1]}	
i ← j-1	n-1
while i ≥ 0 and A[i] > key do	$\sum_{j=1}^{n-1} t_j$
A[i+1] ← A[i]	$\sum_{j=1}^{n-1} (t_j - 1)$
i - -	$\sum_{j=1}^{n-1} (tj - 1)$
A[i+1] ← key	n-1

Analysis of insertion sort

- Best-case: $O(n)$
- Worst-case: $O(n^2)$
- Average case: $O(n^2)$