Introduction to Computation and Programming

Program Efficiency, Binary Search, and Insertion Sort

Reading: [Guttag, Chapter 9], [CLRS, Chap 1, Sections 2.1, 3.1]

[CLRS] (AUB E-book link) : "Introduction to Algorithms", by T. H. **C**ormen, C. E. **L**eiserson, R. L. **R**ivest, and C. **S**tein, MIT press, third edition, 2009, MIT press.

Slides prepared for EECE 230C, Fall 2018-19, MSFEA, AUB

Updated with minor edits during the offering of EECE 230, Spring 2018-19, MSFEA, AUB

Material in these slides is based on [Guttag, Chapter 9], [CLRS, Chapters 1 and 2], and wiki.python.org

Outline

I

- Program efficiency, algorithmic complexity
- Asymptotic notations: Theta, Big O, little o
- Time of analysis of:
  - ➢Linear search
  - ➢Element distinctness
  - ➢Programming Assignment 2 algorithms

II

- Binary Search
- Insertion Sort

III

- Time analysis of some list operations and methods

I.1 Getting started: linear search

- Consider the linear search function (from PSS 4, while loops version):

Problem Solving Session

- If e is in L, function returns index of first occurrence returned. Otherwise, it returns -1

```python
def linearSearch(L,e):
    n = len(L)
    i = 0
    while i< n:
        if L[i]==e:
            return i
        i=i+1
    return -1
```

- Let $T(n)$ = worst case running time of **linearSearch** on a size- n list
- Worst case: Adversary chooses L and e
- Why worst case? It gives a guarantee

I.1 Getting started: linear search (Continued)

- Denote the cost, i.e., time, of Line i by c_i
- Worst case?

```
c1  1 def linearSearch(L,e):
c2  2     n = len(L)
c3  3     i = 0
c4  4     while i< n:
c5  5         if L[i]==e:
c6  6             return i
c7  7         i=i+1
c8  8     return -1
```

I.1 Getting started: linear search (Continued)

- Denote the cost, i.e., time, of Line i by c_i
- Worst case if e not in L
- Thus (worst case) time:

c_1	1	`def linearSearch(L,e):`
c_2	2	`    n = len(L)`
c_3	3	`    i = 0`
c_4	4	`    while i< n:`
c_5	5	`        if L[i]==e:`
c_6	6	`            return i`
c_7	7	`        i=i+1`
c_8	8	`    return -1`

I.1 Getting started: linear search (Continued)

- Denote the cost, i.e., time, of Line i by c_i
- Worst case if e not in L
- Thus (worst case) time:

```
c1   1 def linearSearch(L,e):
c2   2     n = len(L)
c3   3     i = 0
c4   4     while i< n:
c5   5         if L[i]==e:
c6   6             return i
c7   7         i=i+1
c8   8     return -1
```

When while breaks at i=n

$$T(n) = c_1 + c_2 + c_3 + (c_4 + c_5 + c_7) \times n + c_4 + c_8$$

$$= (c_4 + c_5 + c_7) \times n + (c_1 + c_2 + c_3 + c_4 + c_8)$$

$$= (\text{a constant}) \times n + (\text{a negligable term comapred to } n)$$

I.2 Asymptotic analysis

- We can't measure the running exactly as it depends on
  - ➢ Interpreter's implementation
  - ➢ Computer speed
- Solution: **asymptotic analysis**: look at growth of $T(n)$ as **the input size** $\boldsymbol{n} \rightarrow \infty$
- How does $T(n)$ **scale** as input size n doubles or gets multiplied by 10?
- Interested in the **complexity of the algorithm** and not its implementation using a particular programming language or its speed on a specific machine
- Key:
  - ➢ Ignore constants
  - ➢ Ignore low order terms

I.2 Asymptotic analysis (Continued)

- Examples:

Constant

  - $\boxed{5} \times n + \boxed{17}$

Low order terms

  - $\boxed{6} \times\ n^2 + \boxed{18 \times n + 5}$

Constant

- Theta notation:
  - $5 \times n + 17$
  - $6 \times\ n^2 + 18 \times n + 5$
  - $3 \times \log(n) + 7$
  - 10

1.2 Asymptotic analysis (Continued)

- Examples:

Constant

  - $\boxed{5} \times n + \boxed{17}$
  - $\boxed{6} \times n^2 + \boxed{18 \times n + 5}$

Low order terms

Constant

- Theta notation:
  - $5 \times n + 17 = \Theta(n)$
  - $6 \times n^2 + 18 \times n + 5 = \Theta(n^2)$
  - $3 \times \log n + 7 = \Theta(\log n)$
  - $10 = \Theta(1)$

1.3 Theta notation: formal definition

- Definition: Let $\mathrm{f}(n)$ and $\mathrm{g}(n)$ be functions defined on the nonnegative integers and taking real values.

  Assume that for n large enough, $\mathrm{f}(n) \geq 0$ and $\mathrm{g}(n) \geq 0$.

  We say that $\mathrm{f}(n) = \Theta\big(g(n)\big)$ if

  $$\lim_{n\to\infty} \frac{f(n)}{g(n)} \quad = \text{a positive constant}$$

  assuming that the limit exists.

1.3 Theta notation: formal definition (Continued)

- Check above examples:

$$\lim_{n\to\infty} \frac{5\times n+17}{n} = 5 > 0 \quad \Rightarrow \quad 5 \times n + 17 = \Theta(n)$$

$$\lim_{n\to\infty} \frac{6\times n^2+18\times n+5}{n^2} = 6 > 0 \quad \Rightarrow \quad 6 \times n^2 + 18 \times n + 5 = \Theta\left(n^2\right)$$

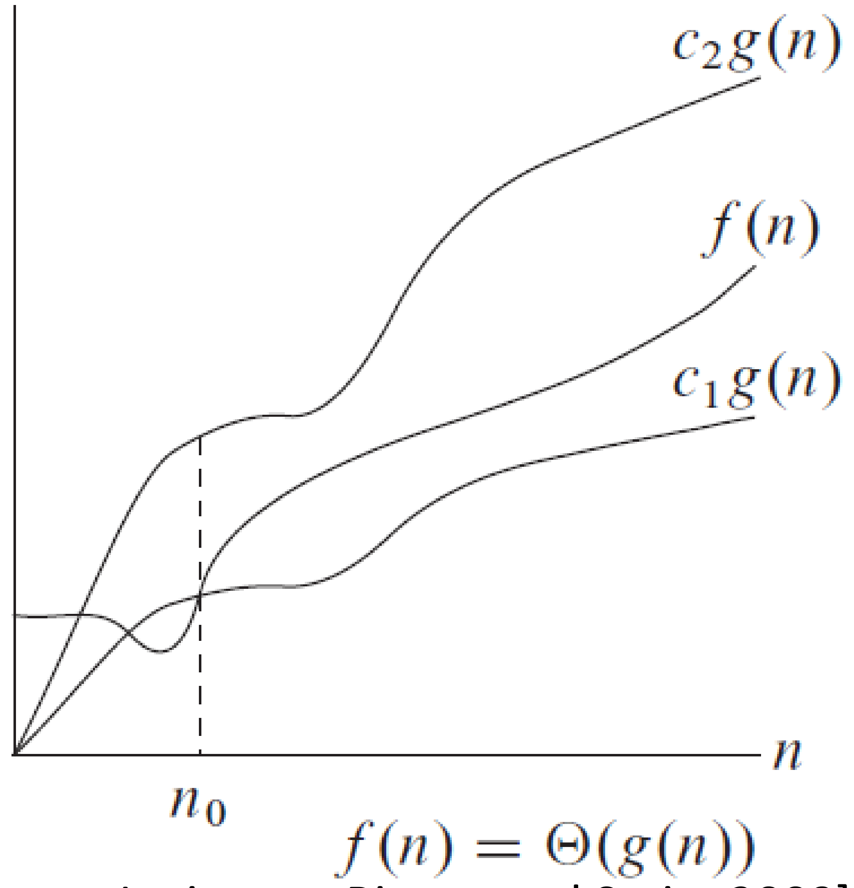
$$\lim_{n\to\infty} \frac{3\times \log n+7}{\log n} = 3 > 0 \quad \Rightarrow \quad 3 \times \log n + 7 = \Theta(\log n)$$

$$\lim_{n\to\infty} \frac{10}{1} = 10 > 0 \quad \Rightarrow \quad 10 = \Theta(1)$$

I.4 Theta notation: more formal definition

More generally (even if limit doesn't exist), we say that $\mathrm{f}(n) = \Theta(g(n))$ if

for large values of n, $\mathrm{f}(n)$ can be sandwiched between two positive constant multiples of g(n), i.e.,



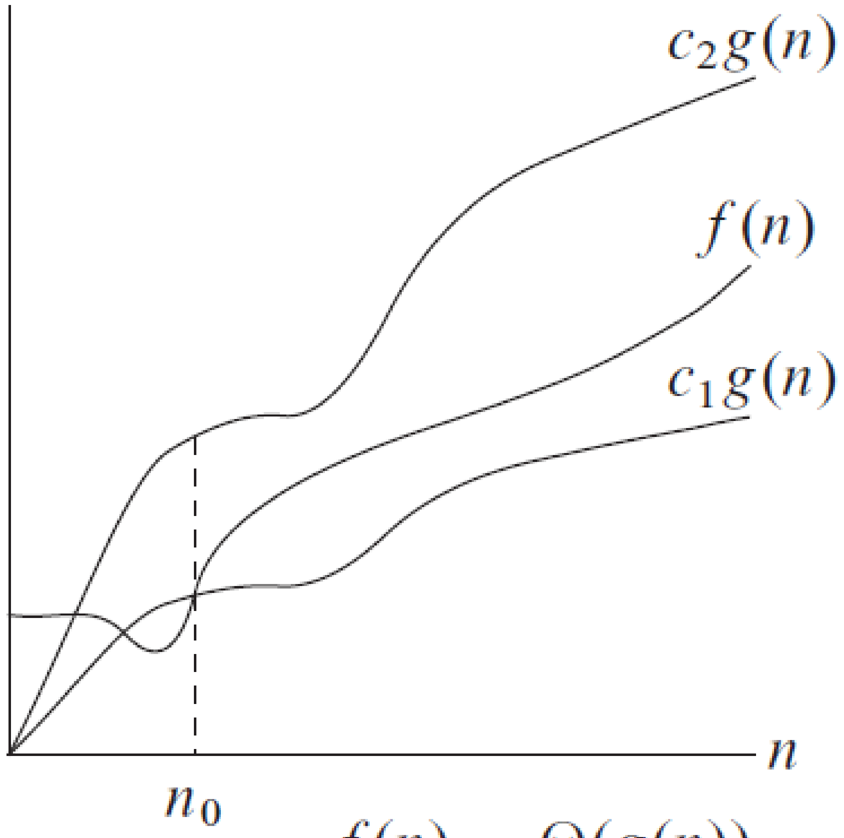
[Figure 3.1 in "Introduction to Algorithms", Cormen, Leriseron, Rivest, and Stein, 2009]

I.4 Theta notation: more formal definition

More generally (even if limit doesn't exist), we say that $\mathrm{f}(n) = \Theta\big(g(n)\big)$ if

for large values of n, $\mathrm{f}(n)$ can be sandwiched between two positive constant multiples of g(n), i.e., there exist $n_0 > 0$ and constants $\mathrm{c}_1, c_2 > 0$ such that for all $n > n_0$,

$$0 \leq c_1 \times g(n) \leq \mathrm{f}(n) \leq c_2 \times g(n)$$



$f(n) = \Theta(g(n))$

[Figure 3.1 in "Introduction to Algorithms", Cormen, Leriseron, Rivest, and Stein, 2009]

I.5 Working with Theta

Useful properties:

- f(n)=$\Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$

- $\boxed{\Theta(g_1(n)) + \Theta(g_2(n))} =$

means

$f_1(n) + f_2(n)$ for some
$f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$

I.5 Working with Theta (Continued)

Useful properties:

- f(n)=$\Theta(g(n)) \;\Rightarrow g(n) = \Theta(f(n))$

- $\Theta(g_1(n)) + \Theta(g_2(n)) = \Theta(g_1(n) + g_2(n))$

- $\Theta(g_1(n)) \times \; \Theta(g_2(n)) =$

I.5 Working with Theta (Continued)

Useful properties:

- f(n)=$\Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$
- $\Theta(g_1(n)) + \Theta(g_2(n)) = \Theta(g_1(n) + g_2(n))$
- $\Theta(g_1(n)) \times \Theta(g_2(n)) = \Theta(g_1(n) \times g_2(n))$

Examples:

- $\Theta(1) + \Theta(n)$ =
- $\Theta(n) + \Theta(n)$ =
- $\Theta(1) \times n =$
- $\Theta(n) \times n =$

I.5 Working with Theta (Continued)

Useful properties:

- f(n)=$\Theta(g(n)) \Rightarrow g(n) = \Theta(f(n))$
- $\Theta(g_1(n)) + \Theta(g_2(n)) = \Theta(g_1(n) + g_2(n))$
- $\Theta(g_1(n)) \times \Theta(g_2(n)) = \Theta(g_1(n) \times g_2(n))$

Examples:

- $\Theta(1) + \Theta(n) = \Theta(n)$
- $\Theta(n) + \Theta(n) = \Theta(n)$
- $\Theta(1) \times n = \Theta(n)$
- $\Theta(n) \times n = \Theta(n^2)$

I.5 Working with Theta: linear Search running time (Continued)

- Instead of using constants, use Θ notation
- Worst case if e not in L
- Worst case running time of **linearSearch**:

$$T(n) = \Theta(n) \text{ steps}$$

```python
def linearSearch(L,e):
    n = len(L)
    i = 0
    while i< n:
        if L[i]==e:
            return i
        i=i+1
    return -1
```

- Note that indexing operator L[i] takes $\Theta(1)$ time: recall for the lists lectures that they are implemented using contiguous memory cells
- Best case running time:

I.5 Working with Theta: linear Search running time (Continued)

- Instead of using constants, use Θ notation
- Worst case if e not in L
- Worst case running time of **linearSearch**:

$$T(n) = \Theta(n) \text{ steps}$$

```python
def linearSearch(L,e):
    n = len(L)
    i = 0
    while i< n:
        if L[i]==e:
            return i
        i=i+1
    return -1
```

- Note that indexing operator L[i] takes $\Theta(1)$ time: recall for the lists lectures that they are implemented using contiguous memory cells
- Best case running time: $\Theta(1)$ steps (if L[0] == e)

I.5 Working with Theta: searching for two elements

```
def linearSearchForTwoElements(L,e1,e2):
    i1 = linearSearch(L,e1)
    i2 = linearSearch(L,e2)
    return (i1,i2)
```

- Worst case time:

I.5 Working with Theta: searching for two elements (Continued)

$\Theta(n)$
$\Theta(n)$

```
def linearSearchForTwoElements(L,e1,e2):
    i1 = linearSearch(L,e1)
    i2 = linearSearch(L,e2)
    return (i1,i2)
```

Passing parameters to function and return

- Worst case time: $\Theta(n)+\Theta(n)+\Theta(1) = \Theta(n)$ steps
- Two sequential loops: $\Theta(n)+\Theta(n) = \Theta(n)$
- Nesting loops costs more

I.6 Time analysis of element distinctness function

- From the lists lectures (function version): start with naive version
- Worst case ?

```
def naiveDistinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i!=j and L[i]==L[j]:
                return   False
    return True
```

I.6 Time analysis of element distinctness function (Continued)

- From the lists lectures (function version): start with naive version
- Worst case if all distinct
- Inner loop takes $\Theta(n)$ steps
- Thus total worst case time of naiveDistinctElements is

```python
def naiveDistinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i!=j and L[i]==L[j]:
                return False
    return True
```

$\Theta(n)$

$$\Theta(n) + \mathrm{n} \times \Theta(n) = \Theta(n^2) \quad \text{steps}$$

Control of outer for, passing parameters to function, final return

I.6 Time analysis of element distinctness function (Continued)

- From the lists lectures (function version): start with naive version
- Worst case if all distinct
- Inner loop takes $\Theta(n)$ steps

```python
def naiveDistinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i!=j and L[i]==L[j]:
                return False
    return True
```

$\Theta(n)$

- Thus total worst case time of naiveDistinctElements is

$$\Theta(n) + \mathrm{n} \times \Theta(n) = \Theta\left(n^2\right) \text{ steps}$$

- Nested loops
- Best case running time:

I.6 Time analysis of element distinctness function (Continued)

- From the lists lectures (function version): start with naive version
- Worst case if all distinct
- Inner loop takes $\Theta(n)$ steps
- Thus total worst case time of naiveDistinctElements is

```python
def naiveDistinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i!=j and L[i]==L[j]:
                return False
    return True
```

$\Theta(n)$

$$\Theta(n) + \mathrm{n} \times \Theta(n) = \Theta\left(n^2\right) \text{ steps}$$

- Nested loops
- Best case running time: $\Theta(1)$ (if L[0]==L[1])

I.6 Time analysis of element distinctness function (Continued)

- Now consider less naive function:
- Worst case?

```
def distinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(i+1,n):
            if L[i]==L[j]:
                return   False
    return True
```

I.6 Time analysis of element distinctness function (Continued)

- Now consider less naive function:
- Worst case if distinct, in which case inner loop takes $\Theta(n-i)$ steps

```
def distinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(i+1,n):
            if L[i]==L[j]:
                return   False
    return True
```

$\Theta(n-i)$

I.6 Time analysis of element distinctness function (Continued)

- Now consider less naive function:
- Worst case if distinct, in which case inner loop takes $\Theta(n-i)$ steps

```
def distinctElements(L):
    n = len(L)
    for i in range(n):
        for j in range(i+1,n):
            if L[i]==L[j]:
                return   False
    return True
```

$\Theta(n-i)$

- Therefore, worst case running time of distinctElements is $\boldsymbol{\Theta(n^2)}$:

$$\mathrm{T(n)} = \Theta(n) + \sum_{i=0}^{n-1} \Theta(n-i) = \Theta(\sum_{i=0}^{n-1}(n-i)) = \Theta(\mathrm{n}^2)$$

(since $\sum_{i=0}^{n-1}(n-i) = n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$)

- That is, the speedup trick ($j \geq i+1$) only changed T(n) by a constant

I.7 Other asymptotic notations

Theta: $f(n) = \Theta(g(n))$	f(n) is asymptotically like g(n)	
Big O*:* $f(n) = O(g(n))$	f(n) is asymptotically like g(n) or weaker than g(n)	
Little o*:* $f(n) = o(g(n))$	f(n) is asymptotically weaker than g(n)	

I.7 Other asymptotic notations (Continued)

Theta: f(n) = $\Theta(g(n))$	f(n) is asymptotically like g(n)	
Big O*:* f(n) = $O(g(n))$	f(n) is asymptotically like g(n) or weaker than g(n)	There exist c > 0 and $n_0 > 0$ such that for all $n > n_0$, $0 \leq f(n) \leq c \times g(n)$
Little o*:* f(n) = $o(g(n))$	f(n) is asymptotically weaker than g(n)	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

- Note: f(n) = $O(g(n))$ and g(n) = $O(f(n))$ $\Leftrightarrow$ f(n) = $\Theta(g(n))$
- Notational difference compared to [Guttag]:
  - $f = O(g)$ in [Guttag] means $f = \Theta(g)$ here
  - $f \in O(g)$ in [Gutta] means $f = O(g)$ here

I.7 Other asymptotic notations: examples

- $5 \times n^2 + 1000 \times n + 17 \quad \Theta(n^2)$
- $5 \times n^2 + 1000 \times n + 17 \quad \mathrm{O}(n^2)$
- ~~$5 \times n^2 + 1000 \times n + 17 \neq \mathrm{o}(n^2)$~~
- $5 \times n^2 + 1000 \times n + 17 \neq \Theta(n^3)$
- $5 \times n^2 + 1000 \times n + 17 \quad \mathrm{O}(\mathrm{n}^3)$
- ~~$5 \times n^2 + 1000 \times n + 17 \quad \mathrm{o}(\mathrm{n}^3)$~~
- $5 \times n^2 + 1000 \times n + 17 \quad \Theta(n)$
- $5 \times n^2 + 1000 \times n + 17 \quad \mathrm{O}(n)$
- $5 \times n^2 + 1000 \times n + 17 \quad \mathrm{o}(n)$

I.7 Other asymptotic notations: examples (Continued)

- $5 \times n^2 + 1000 \times n + 17 = \Theta(n^2)$
- $5 \times n^2 + 1000 \times n + 17 = \mathrm{O}(n^2)$
- $\cancel{5 \times n^2 + 1000 \times n + 17 \neq \mathrm{o}(n^2)}$
- $5 \times n^2 + 1000 \times n + 17 \neq \Theta(n^3)$
- $5 \times n^2 + 1000 \times n + 17 = \mathrm{O}(\mathrm{n}^3)$
- $\cancel{5 \times n^2 + 1000 \times n + 17 = \mathrm{o}(\mathrm{n}^3)}$
- $5 \times n^2 + 1000 \times n + 17 \neq \Theta(n)$
- $5 \times n^2 + 1000 \times n + 17 \neq \mathrm{O}(n)$
- $5 \times n^2 + 1000 \times n + 17 \neq \mathrm{o}(n)$

I.7 Other asymptotic notations (Continued)

- Say that you have an algorithm with worst case running time T(n)
- What does T(n) = $\Theta\big(g(n)\big)$ mean?
- What does T(n) = $O\big(g(n)\big)$ mean?
- What does T(n) = $o\big(g(n)\big)$ mean?

I.7 Other asymptotic notations (Continued)

- Say that you have an algorithm with worst case running time T(n)
- What does T(n) = $\Theta\left(g(n)\right)$ mean? The worst case running time grows like g(n), i.e., g(n) is an asymptotic worst case guarantee which is attainable.
- What does T(n) = $O\left(g(n)\right)$ mean? The worst case running time grows like g(n) or is weaker than g(n), i.e., g(n) is an asymptotic worst case guarantee which may or may not be attainable.
- What does T(n) = $o\left(g(n)\right)$ mean? The algorithm is asymptotically much faster than g(n)

I.8 Common growth rates

- $\Theta(1)$ *is called* **constant** running time
- $\Theta(\log n)$ is called **logarithmic** running time
- $\Theta(n)$ is called **linear** running time
- $\Theta(n \log n)$ is called **log-linear** running time
- $\Theta\left(n^2\right)$ is called **quadratic** running time
- $\Theta(n^k)$,where $k > 0$ is a constant, is called **polynomial** running time
- $\Theta(c^n)$, where $c > 1$ is a constant, is called **exponential** running time

I.9 Comparison of common growth rates

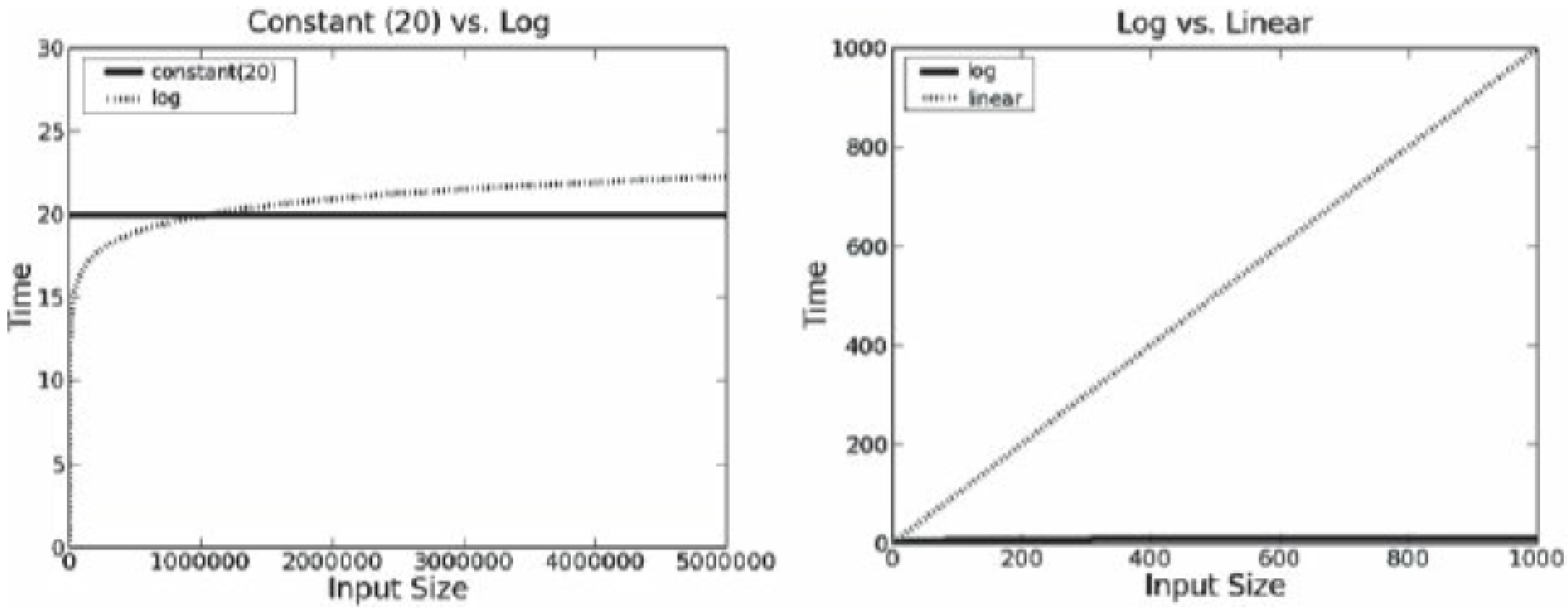


Figure 9.7 Constant, logarithmic, and linear growth

[Guttag, 2016, Chapter 9]

I.9 Comparison of common growth rates (Continued)

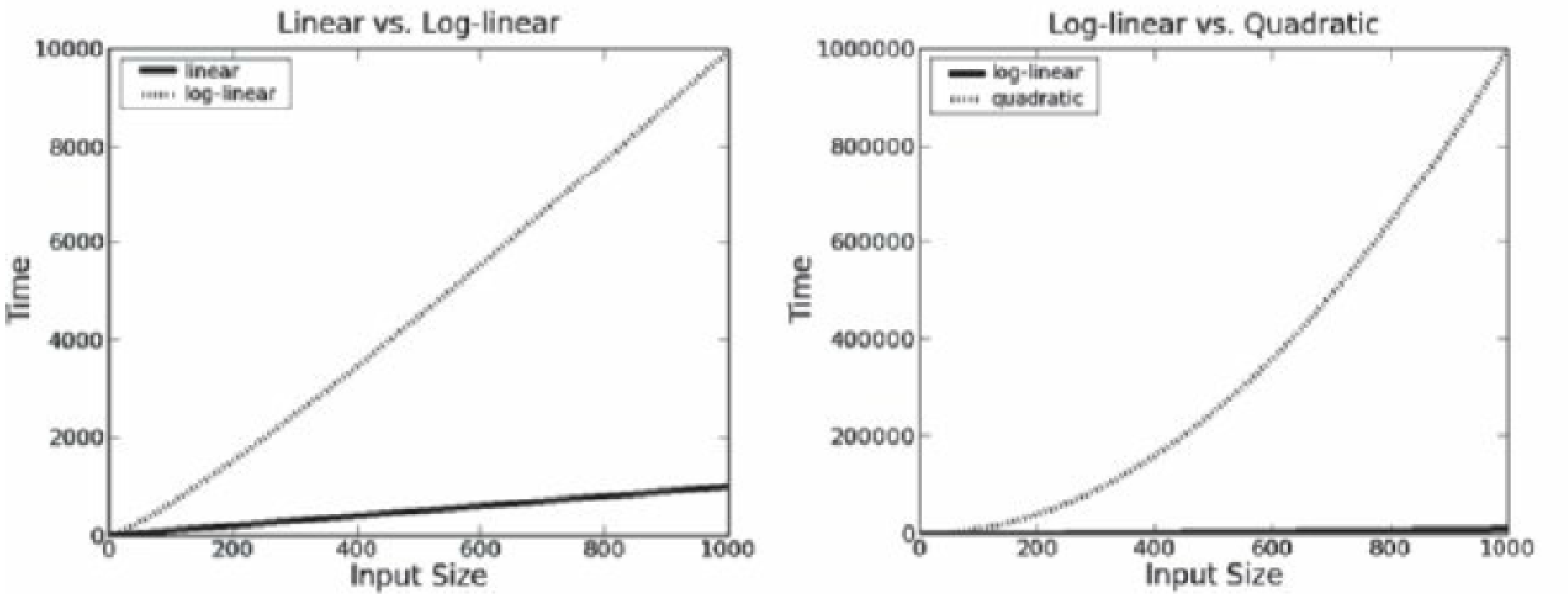


Figure 9.8 Linear, log-linear, and quadratic growth

[Guttag, 2016, Chapter 9]

I.9 Comparison of common growth rates (Continued)

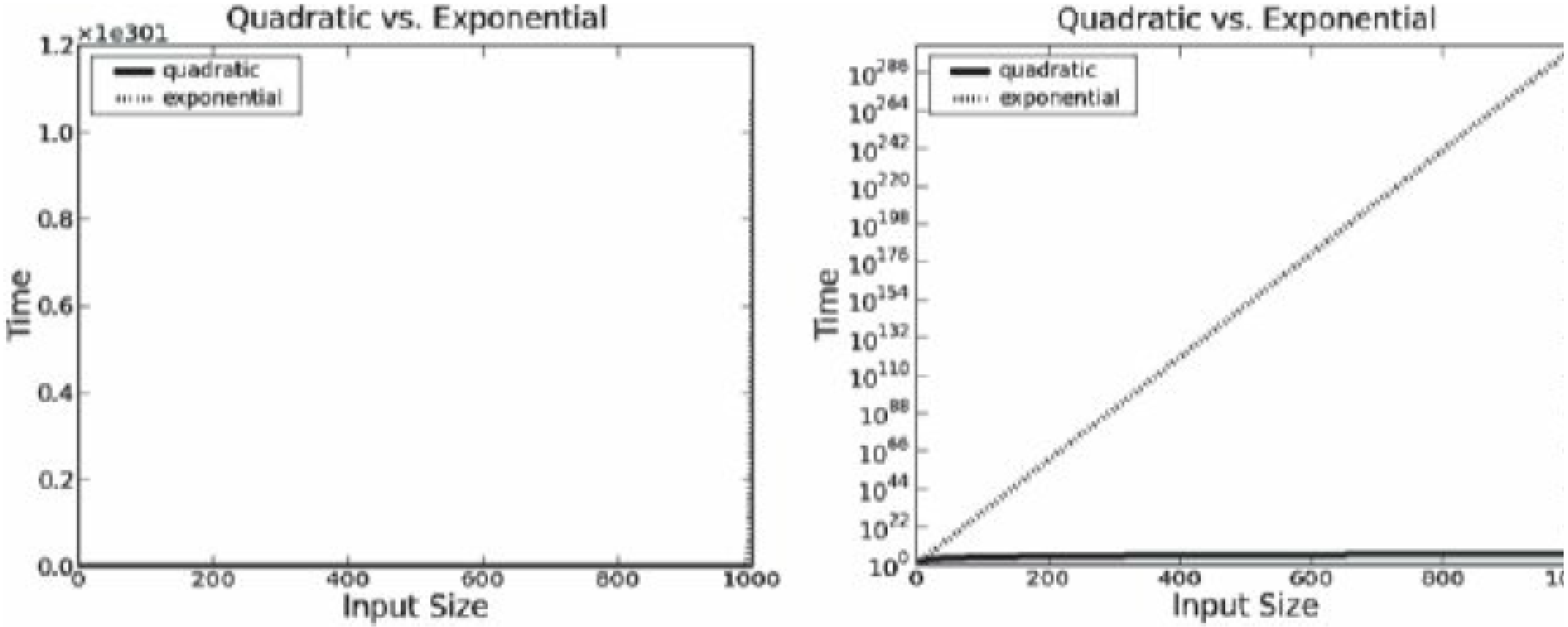


Figure 9.9 Quadratic and exponential growth

[Guttag, 2016, Chapter 9]

I.10 Examples from Programing Assignments (PA) 1 and 2

- [PA1.Problem 4] Quadratic equations solver:
- [PA2.Problem 1.a] Time to find the factorial of a given number n:
- [PA2.Problem 2] Time to find the max in a sequence of n number entered by user:
  - Space (memory):
- [PA2.Problem 3.a] Time to check if a given number n is prime:

I.10 Examples from Programing Assignments (PA) 1 and 2 (Continued)

- [PA1.Problem 4] Quadratic equations solver: $\Theta(1)$ time
- [PA2.Problem 1.a] Time to find the factorial of a given number n: $\Theta(\mathrm{n})$ **arithmetic operations** (for large n, multiplications and additions cost more than $\Theta(1)$ time)
- [PA2.Problem 2] Time to find the max in a sequence of n number entered by user: $\Theta(\mathrm{n})$ time
  - Space (memory): $\Theta(1)$
- [PA2.Problem 3.a] Time to check if a given number n is prime: $\Theta(\sqrt{\mathrm{n}})$ arithmetic operations
  (best known poly-log: $\Theta\left((\log\ n)^c\right)$, where $\mathrm{c} > 0$ is a constant)

I.10 Examples from Programing Assignment (PA) 2 (Continued)

- [PA2.Problem 4.a] Time to check if a given number n is square:
- [PA2.Problem 4.b] Time to check if a given number n is square using bisection method (function version):

```
def isSquareBisection(n):
    if n<0: return False
    elif n ==0:return True
    else:
        low = 1
        high = n
        while low<=high:
            mid = (low+high)//2
            if mid*mid ==n:
                return True
            elif mid*mid<n:
                low = mid+1
            else:
                high  = mid-1
        return False
```

I.10 Examples from Programing Assignment (PA) 2 (Continued)

- [PA2.Problem 4.a] Time to check if a given number n is square: $\Theta(\sqrt{n})$ arithmetic operations
- [PA2.Problem 4.b] Time to check if a given number n is square using bisection method (function version):
  - $\Theta(\log n)$ arithmetic operations
- Why?

```python
def isSquareBisection(n):
    if n<0: return False
    elif n ==0:return True
    else:
        low = 1
        high = n
        while low<=high:
            mid = (low+high)//2
            if mid*mid ==n:
                return True
            elif mid*mid<n:
                low = mid+1
            else:
                high  = mid-1
        return False
```

I.11 Time analysis of square-root test using bisection

- Initial search interval consists of the integers in $\mathbf{[1, n]}$
- After each iteration of the while loop, the length of the search interval is reduced by at least half
- Thus after $\mathbf{k}$ iterations, its length is at most $\mathbf{n/2^k}$
- Hence after at most $\mathbf{\log_2 n}$ iterations, its length is at most $\mathbf{1}$
- Moreover, it is reduced by at least one integer at each iteration (as either $\mathbf{low}$ is set to $\mathbf{mid+1}$ or $\mathbf{high}$ to $\mathbf{mid-1}$,if $\mathbf{mid \times mid \neq n}$)
- Hence after at most $\mathbf{\log_2 n + 1}$ iterations, its must be empty
- Thus, the worst case time: $\mathbf{O(\log_2 n + 1) = O(\log n)}$ arithmetic operation (Big O in Section I.7 above)
- It is actually $\mathbf{\Theta(\log n)}$: worst case when n is not square

||

- Binary Search
- Insertion Sort

II.1 Binary Search: the problem of searching sorted lists

- When we have many search queries, it is more efficient to first sort the list and implement the search queries using a searching algorithm smarter than linear search, which takes linear time

- *Given a list* **L[0…n-1]** of integers sorted in non-decreasing order and a number **x,** check if **x** is in **L:** if found return an index **i** such that **L[i]= x**, otherwise return **-1**

II.1 Idea of Binary Search

- Same as the bisection method
- Compare x with the middle element of L
- If >, we can ignore the lower half of L since L is sorted
- If <, we can ignore the upper half of L since L is sorted
- If =, we are done (x is an element of L)
- Repeat

II.1 Try it on a example

Try it on:

a. L = [-3,-2,1,1,2,3, 5, 6, 8, 9,17] and x = 5
b. Same L with and x = 4

II.1.a Binary search for 5 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

II.1.a Binary search for 5 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

II.1.a Binary search for 5 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

6	7	8	9	10
5	6	8	9	17

[5, 6, 8, 9,17]

II.1.a Binary search for 5 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0 1 2 3 4 5 6 7 8 9 10

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

6 7 8 9 10

[5, 6, 8, 9,17]

6 7

[5, 6]

II.1.a Binary search for 5 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

6	7	8	9	10
5	6	8	9	17

[5, 6, 8, 9,17]

6	7
5	6

[5, 6]

Return the index 6 of 5

II.1.b Binary search for 4 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

II.1.b Binary search for 4 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

II.1.b Binary search for 4 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10
-3	-2	1	1	2	3	5	6	8	9	17

6	7	8	9	10
5	6	8	9	17

II.1.b Binary search for 4 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0	1	2	3	4	5	6	7	8	9	10

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

6	7	8	9	10

[5, 6, 8, 9,17]

6	7

[5, 6]

II.1.b Binary search for 4 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0 1 2 3 4 5 6 7 8 9 10

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

6 7 8 9 10

[5, 6, 8, 9,17]

6 7

[5, 6]

[]

II.1.b Binary search for 4 in [-3,-2,1,1,2,3, 5, 6, 8, 9,17]

0 1 2 3 4 5 6 7 8 9 10

[-3,-2,1,1,2,3, 5, 6, 8, 9,17]

6 7 8 9 10

[5, 6, 8, 9,17]

6 7

[5, 6]

[]

Empty search interval: return -1

II.1 Elaborate on idea

- We need 3 variables: **low**, **mid**, and **high**
- Initially: **low = 0** and **high = n-1**
- Compute: **mid = (n-1)//2**
  (floor of **(n-1)/2**, i.e., *largest integer less than or equal to* **(n-1)/2**)
- If **x==A[mid],** done: **return mid**
- If **L[mid]<x**, update **low = mid +1** and keep high the same
- If **L[mid]>x**, update **high = mid -1** and keep low the same
- Re-compute: **mid = (low+high)//2**
- Repeat this process until either **x** is found or **low > high**, in which case return **-1**

II.1 Binary Search function

```python
def binarySearch(L, x):
    n  = len(L)
    low = 0
    high = n-1
    while low<=high:
        mid = (low+high)//2
        if L[mid] == x:
            return mid
        elif L[mid]<x:
            low = mid+1
        else:
            high = mid-1
    return -1
```

II.1 Binary Search time analysis: same as square-root bisection

- Initially, list size is **n**
- After each iteration of the while loop, the length of the sub-list **L[start … end]** is reduced by at least half
- Thus after **k** iterations, its length is at most $\mathbf{n/2^k}$
- Hence after at most $\mathbf{\log_2 n}$ iterations, its length is at most $\mathbf{1}$
- Moreover, length is reduced by at least one at each iteration (as either $\mathbf{low}$ is set to $\mathbf{mid+1}$ or $\mathbf{high}$ to $\mathbf{mid-1}$, if $\mathbf{L}[\boldsymbol{mid}] \neq \boldsymbol{x}$)
- Hence after at most $\mathbf{\log_2 n + 1}$ iterations, its must be empty
- This shows that the worst case time = $\mathbf{O(\log_2 n + 1) = O(\log n)}$
- It is actually $\mathbf{\Theta(\log n)}$: worst case when x is not in the list
- Best case time = $\mathbf{\Theta(1)}$: if **x==** $\mathbf{L}[\boldsymbol{mid}]$ in the first iteration

II.2 Insertion Sort: the Porting Problem

- ***Input:*** list of n numbers L = [L[0],   L[1],· · ·, L[*n-1*]]
- ***Objective:*** permute the elements of L so that they are sorted in non-decreasing order, i.e., L[0] ≤ L[1] ≤ · · · ≤L[*n*-1]
- **Example:**
  - ***Input:*** L=[8, 2, 4, 9, 3, 2, 6]
  - ***Sorted:*** L=[2, 2, 3, 4, 6, 8, 9]
- In PA 4, you implemented the Selection Sort algorithm, which takes $\Theta\left(n^2\right)$ time
- Now: Insertion Sort, which also takes $\Theta\left(n^2\right)$ time

II.2 Idea of Insertion Sort

- Idea: sorting a hand of cards
- First card: ok
- Compare second card with the first and insert in its correct place
- Compare the third card with the first and second card and insert it in its correct place
- And so on until you reach the last card

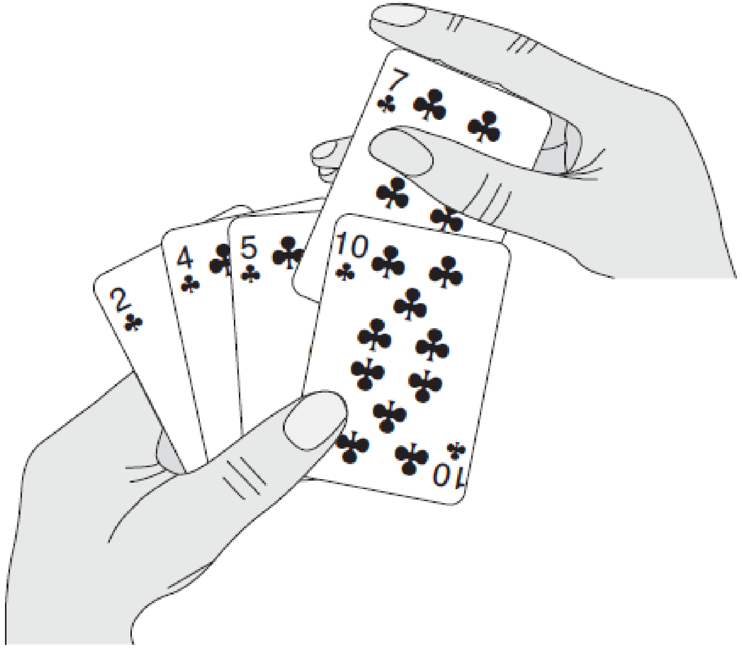


Figure 2.1 Sorting a hand of cards using insertion sort.

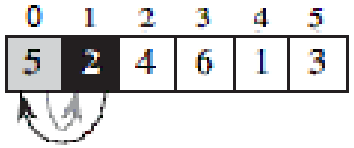
Figure 2.1 in [CLRS, page 17]

II.2 Try it on an example

L = [5, 2, 4, 6, 1, 3]

II.2 Try it on an example (Continued)

L = [5, 2, 4, 6, 1, 3]

(a)


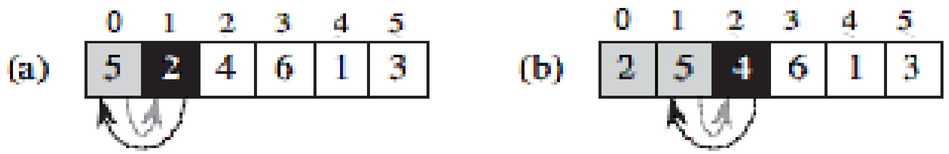
Edited version of Figure 2.2 in [CLRS, page 18]

- Black cell: value under consideration, called key
- Shaded cell: values compared to the key
- Shaded arrows: values moved the right
- Black arrow: where the key is inserted

II.2 Try it on an example (Continued)

L = [5, 2, 4, 6, 1, 3]

L = [2, 5, 4, 6, 1, 3]



Edited version of Figure 2.2 in [CLRS, page 18]

- Black cell: value under consideration, called key
- Shaded cell: values compared to the key
- Shaded arrows: values moved the right
- Black arrow: where the key is inserted

II.2 Try it on an example (Continued)

L = [5, 2, 4, 6, 1, 3]

L = [2, 5, 4, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]

Edited version of Figure 2.2 in [CLRS, page 18]

- Black cell: value under consideration, called key
- Shaded cell: values compared to the key
- Shaded arrows: values moved the right
- Black arrow: where the key is inserted

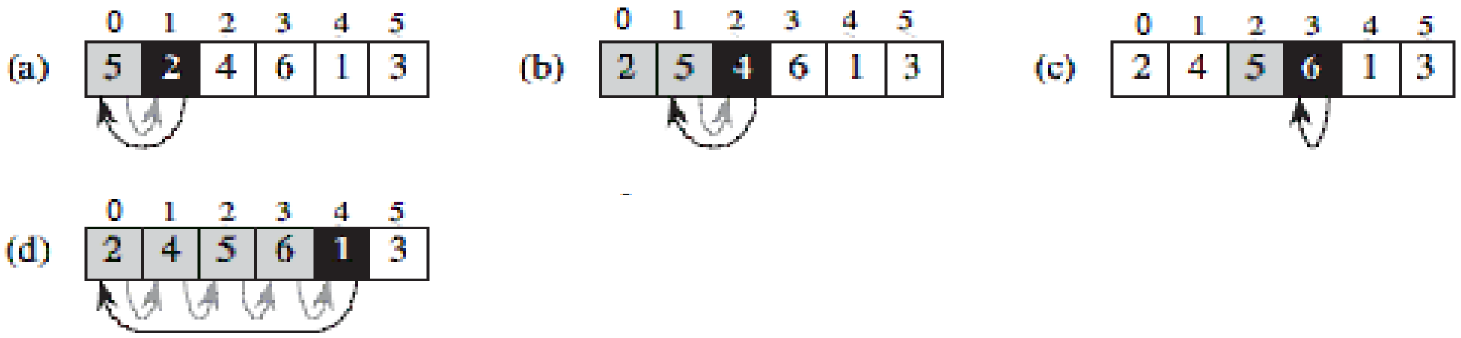
II.2 Try it on an example (Continued)

L = [5, 2, 4, 6, 1, 3]

L = [2, 5, 4, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]



Edited version of Figure 2.2 in [CLRS, page 18]

- Black cell: value under consideration, called key
- Shaded cell: values compared to the key
- Shaded arrows: values moved the right
- Black arrow: where the key is inserted

II.2 Try it on an example (Continued)

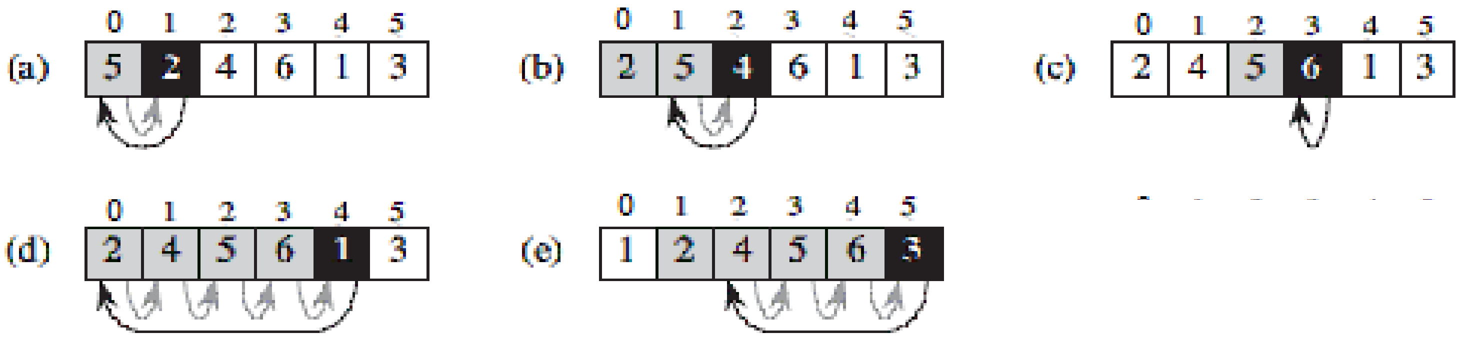
L = [5, 2, 4, 6, 1, 3]

L = [2, 5, 4, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]

L = [1, 2, 4, 5, 6, 3]



Edited version of Figure 2.2 in [CLRS, page 18]

- Black cell: value under consideration, called key
- Shaded cell: values compared to the key
- Shaded arrows: values moved the right
- Black arrow: where the key is inserted

II.2 Try it on an example (Continued)

L = [5, 2, 4, 6, 1, 3]

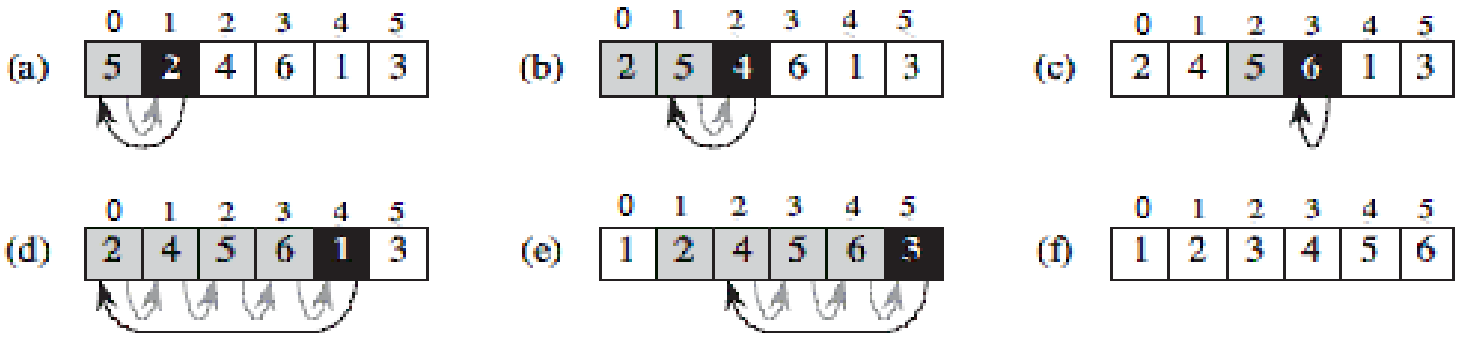
L = [2, 5, 4, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]

L = [2, 4, 5, 6, 1, 3]

L = [1, 2, 4, 5, 6, 3]

L = [1, 2, 3, 4, 5, 6]



Edited version of Figure 2.2 in [CLRS, page 18]

- Black cell: value under consideration, called key
- Shaded cell: values compared to the key
- Shaded arrows: values moved the right
- Black arrow: where the key is inserted

II.2 Insertion Sort function (Continued)

```
def insertionSort(L):
    n = len(L)
    for j in range(1,n):
        # Insert L[j] into the sorted sequence  L[0···j-1]
        key = L[j] # Save L[j] in key to avoid loosing it
```

II.2 Insertion Sort function (Continued)

```python
def insertionSort(L):
    n = len(L)
    for j in range(1,n):
        # Insert L[j] into the sorted sequence  L[0···j-
        key = L[j] # Save L[j] in key to avoid loosing i
        i = j-1
        while  i>=0 and L[i] > key:
            L[i+1] = L[i]  # move L[i] forward
            i = i -1       # and go one step back
        L[i+1] = key
```

- Function modifies input list L: it has no return value

II.2 Insertion Sort function (Continued)

```python
def insertionSort(L):
    n = len(L)
    for j in range(1,n):
        # Insert L[j] into the sorted sequence  L[0···j-1]
        key = L[j] # Save L[j] in key to avoid loosing it
        i = j-1
        while  i>=0 and L[i] > key:
            L[i+1] = L[i]  # move L[i] forward
            i = i -1       # and go one step back
        L[i+1] = key

L = [1, 5,17, 3 -1, 0,17.3, 105, 56.9]
insertionSort(L)
```

- Function modifies input list L: it has no return value

II.2 Time Analysis Insertion Sort

- We have two nested loops each running for at most n steps
- Thus the worst case time is $\mathrm{O}\left(\mathrm{n}^2\right)$ steps
- By more carful analysis, we will show below that it is $\Theta\left(n^2\right)$
- Analysis similar to element distinctness algorithm

II.2 Time Analysis Insertion Sort (Continued)

- Worst case when array in reverse order: inner while loop will always go back $i = 0$ and stop at $i = -1$
- Thus at the j'th iteration of the outer loop, the inner loop takes Θ (j) steps. Therefore, the j'th iteration of the outer loop takes Θ (j) steps
- Hence total worst case running time:

$$\Theta(n) + \sum_{j=1}^{n-1} \Theta(j) = \Theta\left(\sum_{j=1}^{n-1} j\right) = \Theta\left(n^2\right)$$

```
def insertionSot(L):
    n = len(L)
    for j in range(1,n):
        key = L[j]
        i = j-1
        while i>=0 and L[i] > key:
            L[i+1] = L[i]
            i = i -1
        L[i+1] = key
```

Θ (1)

Θ (j)

Θ (1)

Θ (j) steps

II.2 Selection Sort versus Insertion Sort

	Selection Sort	Insertion Sort
Worst case running time		
Best case running time		

II.2 Selection Sort versus Insertion Sort (Continued)

	Selection Sort	Insertion Sort
Worst case running time	$\Theta(n^2)$	$\Theta(n^2)$
Best case running time	$\Theta(n^2)$	$\Theta(n)$
Number of write operations on list		

II.2 Selection Sort versus Insertion Sort (Continued)

	Selection Sort	Insertion Sort
Worst case running time	$\Theta(n^2)$	$\Theta(n^2)$
Best case running time	$\Theta(n^2)$	$\Theta(n)$
Number of write operations on list	$\Theta(n)$	$\Theta(n^2)$ Worst case

III. Time analysis of some list operations and methods

III.1 Time analysis of basic list operations and methods

Assume below that objects in lists are $\Theta(1)$-size scalars (i.e., integers of size $\Theta(1)$ or objects of type float, bool, or None)

Equality check: L1==L2		
Concatenation: L = L1+L2		
Membership test: e in L		
Slicing: L[i:j+1]		
L.count(e)		
L.index(e)		
L.reverse(e)		

III.1 Time analysis of basic list operations and methods (Continued)

Assume below that objects in lists are Θ(1)-size scalars (i.e., integers of size Θ(1) or objects of type float, bool, or None)

Equality check: L1==L2	Θ(min(len(L1), len(L2))	PA 3.Problem 2.b
Concatenation: L = L1+L2	Θ(len(L1)+len(L2))	
Membership test: e in L	Θ(len(L1)) if e is a scalar	PSS 3.Prolem 1.b
Slicing: L[i:j+1]	Θ(j-i)	
L.count(e)	Θ(len(L))	PSS 4.Problem 1.a
L.index(e)	Θ(len(L))	PSS 4.Problem 1.b
L.reverse(e)	Θ(len(L))	PSS 4.Problem 1.c

III.1 Time analysis of basic list operations and methods (Continued)

Assume below that objects in lists are Θ(1)-size scalars (i.e., integers of size Θ(1) or objects of type float, bool, or None)

same complexities for strings

Equality check: L1==L2	Θ(min(len(L1), len(L2))	PA 3.Problem 3.b
Concatenation: L = L1+L2	Θ(len(L1)+len(L2))	
Membership test: e in L	Θ(len(L1)) if e is a scalar	PSS 3.Prolem 1.b
Slicing: L[i:j+1]	Θ(j-i)	
L.count(e)	Θ(len(L))	PSS 4.Problem 1.a
L.index(e)	Θ(len(L))	PSS 4.Problem 1.b
L.reverse(e)	Θ(len(L))	PSS 4.Problem 1.c

III.2 List.append method

- Recall from [Functions III.3] that in the worst case, a single L.append(e) operations takes Θ(len(L)) time: if not enough contiguous cells are available, the whole list is copied to new place in memory and resized
- But the overhead on a long sequence of append operation is not substantial
- Why? The implementation of append in Python is something like the this: when append makes the list size a power of 2, the list is doubled, i.e., it is copied to new place in memory and resized to twice its size

III.3 List.append method: $\boxed{\text{amortize}}$d analysis

reduce or pay off (a debt) with regular payments [Oxford Dictionaries]

- Consider the following sequence of append operations:

```
L = []
for i in range(n):
    # get e from somewhere, e.g., user input
    L.append(e)
```

- Let k be the largest power of 2 less than n, i.e., $2^k < n$
- Then for $\mathrm{i} = 1,2,2^2, 2^3, \dots, 2^{\mathrm{k}}$, the cost of append is $\Theta(2i) = \Theta(\mathrm{i})$
- For all other values of i,the cost is $\Theta(1)$
- **Thus total cost** : $\Theta(\mathrm{n}) + \Theta\left(\sum_{t=0}^{k} 2^t\right) = \boldsymbol{\Theta(n)}$

  since $\sum_{t=0}^{k} 2^t = 2^{\mathrm{k}+1} - 1 < 2\mathrm{n} - 1$

III.3 List.append method: amortized analysis (Continued)

- Compare with L=L+[e]:

```python
L = []
for i in range(n):
    # get e from somewhere, e.g., user input
    L=L+[e]
```

- For each i, the cost of L=L+[e] is $\Theta(i)$ (a new list is created)
- Thus total cost : $\Theta\left(\sum_{i=0}^{n-1} i\right) = \Theta\left(n^2\right)$

III.4 List.sort method

- List.sort takes $\Theta(n \log n)$ time to sort a size-n list
- Much faster than Selection Sort and Insertion Sort, which take $\Theta(n^2)$ time each
- Next topic is recursion
- Among other things, we will see how recursion can be used to sort in $\Theta(n \log n)$ time