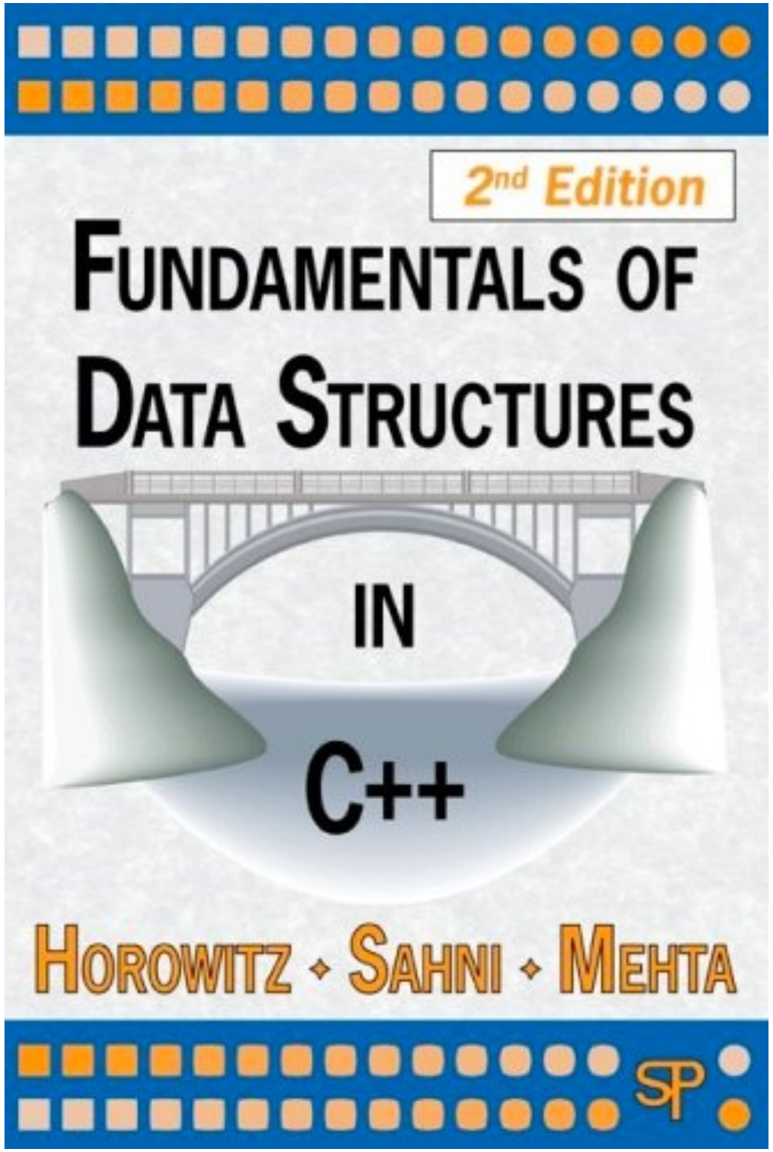
DS:
Algorithm Run Time Analysis

Liwei

Data Structure

and Algorithms


2nd Edition
FUNDAMENTALS OF
DATA STRUCTURES
IN
C++
HOROWITZ • SAHNI • MEHTA
SP

DS and Algorithms

- Algo Run time analysis
- Physical DS
  - Array
  - Linked List
- Logical DS
  - Stack
  - Queue
  - Tree
  - Hashing
  - Graph
- Miscellaneous Topics
  - Sorting
- Algorithm Techniques
  - Shortest Path
  - Divide and Conquer
  - Greedy Algorithm
  - Dynamic Proramming
  - Magic Framework

Algorithm Run Time Analysis

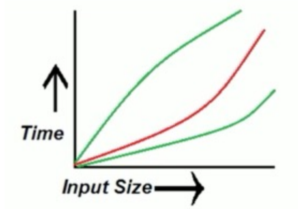
What and why

- What is “Algorithm Run Time Analysis”?
  - It is a study of a given algorithm’s running time, by identifying its behavior as the input size for the algorithm increases.
  - How much time will the given algorithm take to run
- Why should we learn this?
  - To measure efficiency of a given algorithm

Notation for Algorithm Run Time Analysis

- How much does this car runs on 1 liter of petrol?
  - In city traffic?
  - On highway?
  - Mixed environment?

Notation for Algorithm Run Time Analysis



- There are three notations for Run Time Analysis
- Omega(Ω)
  - This notation gives the tighter lower bound of a given algorithm
  - For any given input, running time of a given algorithm will not be “less than” given time.
- Big-o(O)
  - This notation gives the tighter upper bound of a given algorithm
  - For any given input, running time of a given algorithm will not be “more than” given time.
- Theta(θ)
  - This notation decides whether upper bound and lower bound of a given algorithms are same or not
  - For any given input, running time of a given algorithm will “on an average” be equal to given time.

Notation for Algorithm Run Time Analysis

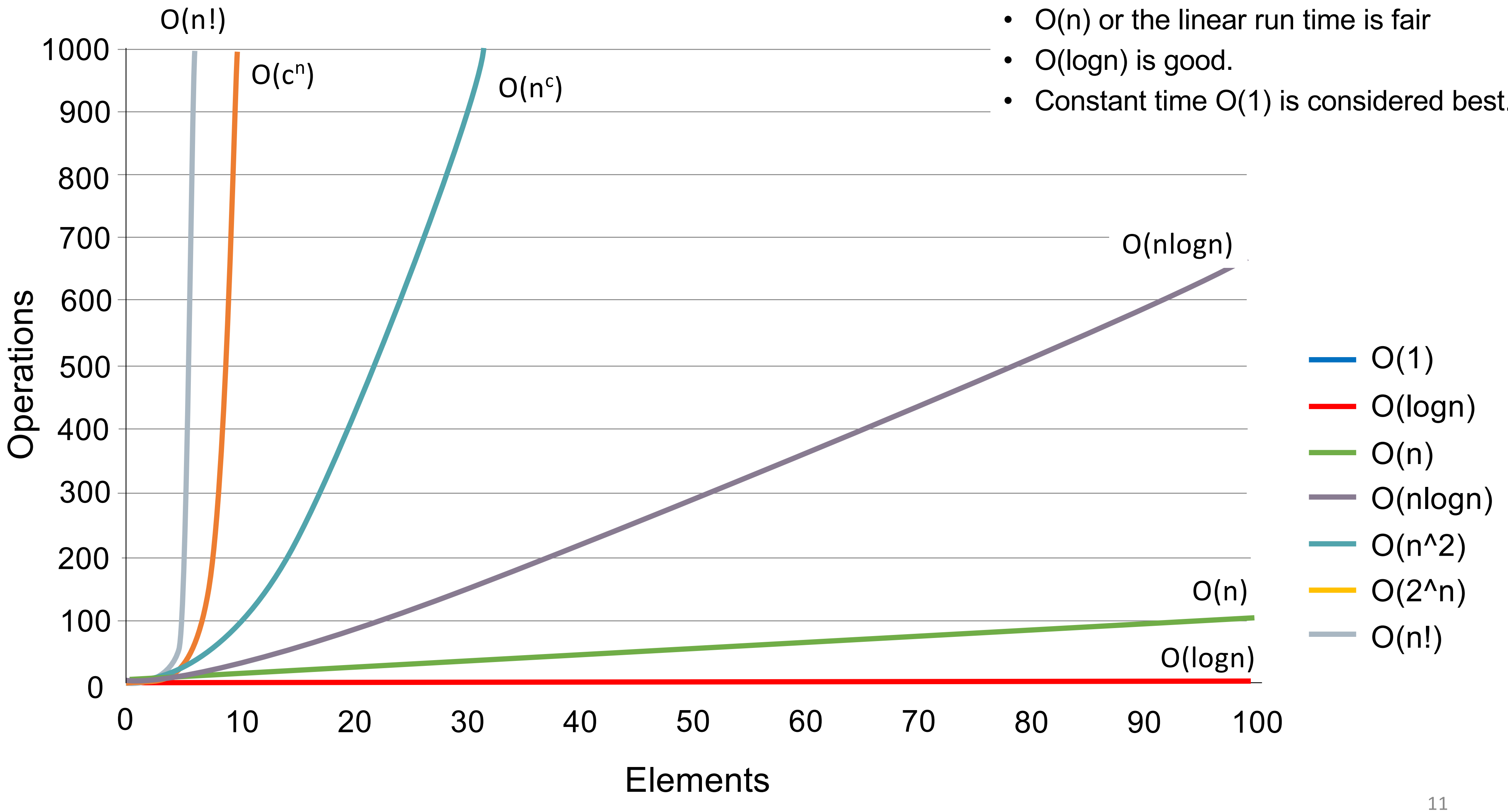
5	18	3	54	26	...	55	41	...	19	1	10

- Omega(Ω)
  - For any given input, running time of a given algorithm will not be “less than” given time.
  - $\Omega(1)$
- Big-o(O)
  - For any given input, running time of a given algorithm will not be “more than” given time.
  - $O(n)$
- Theta(Θ)
  - For any given input, running time of a given algorithm will “on an average” be equal to given time.
  - $\Theta(n/2)$

Examples

Time Complexity	Name	Example
$O(1)$	Constant	Add an element at front of linked list
$O(logn)$	Logarithmic	Finding an element in sorted array (**Binary Search)**
$O(n)$	Linear	Finding an element in unsorted array (**Linear Search)**
$O(nlogn)$	Linear Logarithmic	Merge sort
$O(n^2)$	Quadratic	Shortest path between 2 nodes in a graph
$O(n^3)$	Cubic	Matrix multiplication
$O(2^n)$	Exponential	Tower of Hanoi Problem

- $O(n!)$, $O(n^c)$, $O(c^n)$ are the worst.
- O(nlogn) is bad
- O(n) or the linear run time is fair
- O(logn) is good.
- Constant time O(1) is considered best.



How to calculate algorithm time complexity?

- Iterative Algorithm
- Recursive Algorithm

Example 1: Time Complexity of iterative algo

```
FindBiggestNumber (int arr[])
        biggestNumber = arr[0]
        for i=1 to length(arr) -1
                if arr[i] > biggestNumber
                biggestNumber = arr[i]
        return biggestNumber
```

Example 1: Time Complexity of iterative algo

```
FindBiggestNumber (int arr[])
        biggestNumber = arr[0]                                      O(1)
        for i=1 to length(arr) -1                         O(n)      O(n)
                if arr[i] > biggestNumber       O(1)      O(1)
                biggestNumber = arr[i]          O(1)
        return biggestNumber                                        O(1)
```

Time Complexity = O(1) + O(n) + O(1)
= O(n)

O(n-1)=O(n)
O(2n)=O(n)
O(10)=O(1)
O(1000)=O(1)

Example 2: Time Complexity of recursive algo

```
FindBiggestNumber (A, n)
        static highest = integer.Min
        if n equals -1
                return highest
        else
                if A[n] > highest
                update highest
       return FindBiggestNumber(A, n-1)
```

5	18	3	54	26	…	55	41	…	19	1	10

Example 2: Time Complexity of recursive algo

FindBiggestNumber (A, n)	T(n)
static highest = integer.Min	O(1)
if n equals -1	O(1)
return highest	O(1)
else	O(1)
if A[n] > highest	O(1)
update highest	O(1)
return FindBiggestNumber(A, n-1)	T(n-1)

Example 2: Time Complexity of recursive algo

```
FindBiggestNumber (A, n) ------------------------------ T(n)
    static highest = integer.Min --------------------- O(1)
    if n equals -1 ----------------------------------- O(1)
        return highest ------------------------------- O(1)
    else --------------------------------------------- O(1)
        if A[n] > highest ---------------------------- O(1)
        update highest ------------------------------- O(1)
    return FindBiggestNumber(A, n-1) ----------------- T(n-1)
```

Back substitution:

T(n) = O(1) + T(n-1)

T(-1) = O(1)   (base condition)

T(n-1)=O(1)+T((n-1)-1)

T(n-2)=O(1)+T((n-2)-1)

Back substitution:

$T(n) = O(1) + T(n-1)$

$T(-1) = O(1)$

$T(n-1)=O(1)+T((n-1)-1)$

$T(n-2)=O(1)+T((n-2)-1)$

$$
\begin{aligned}
T(n) &= 1 + T(n-1) \\
&= 1 + (1+T(n-2)) \\
&= 2 + T(n-2) \\
&= 2 + (1 +T(n-3)) \\
&= 3 + T(n-3) \\
&= k + T(n-k) \\
&= (n+1) + T(n-(n+1)) \\
&= n+1 + T(-1) \\
&= n + 1 + 1 \\
&= O(n)
\end{aligned}
$$

Example 3: Time Complexity of recursive algo

```
BinarySearch (int findNumber, int arr[], start, end)
        if (start equals end)
                if(arr[start] equals findNumber)
                        return start
                else return error message (number not exists)

        mid = findMid (arr[], start, end)
        if (mid > findNumber)
                BinarySearch(findNumber, arr, start, mid)
        else if (mid < findNumber)
                BinarySearch(findNumber, arr, mid, end)
        else if (mid == findNumber)
                return mid
```

10	20	30	40	50	60	70	80	90	100	110

Example 3: Time Complexity of recursive algo

```
BinarySearch (int findNumber, int arr[], start, end) ----------------  T(n)
        if (start equals end) ------------------------------------------  O(1)
                if(arr[start] equals findNumber) -----------------------  O(1)
                        return start -----------------------------------  O(1)
                else return error message (number not exists) ---------  O(1)

        mid = findMid (arr[], start, end) ------------------------------  O(1)
        if (mid > findNumber) ------------------------------------------  O(1)
                BinarySearch(findNumber, arr, start, mid) --------------  T(n/2)
        else if (mid < findNumber) -------------------------------------  O(1)
                BinarySearch(findNumber, arr, mid, end) ----------------  T(n/2)
        else if (mid == findNumber) ------------------------------------  O(1)
                return mid ---------------------------------------------  O(1)
```

Time Complexity = $T(n) = O(1) + T(n/2)$

Back substitution:

$T(n) = T(n/2) + 1$

$T(1) = 1$

$T(n/2)=T(n/4)+1$

$T(n/4)=T(n/8)+1$

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= (T(n/4)+1) +1 \\
&= T(n/4) + 2 \\
&= (T(n/8)+1) +2 \\
&= T(n/8) + 3 \\
&= T(n/2^k) + k \\
&= T(1) + \log n \\
&= 1 + \log n \\
&= \log n
\end{aligned}
$$