香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

- Introduction
- Asymptotic Analysis
  - Analyzing insertion sort
- Data Structure Recap

Yanlin Zhang & Shangqi Lu | DSAA 2043 Fall 2026

Introduction

Design and Analysis of Algorithms, Fall 2026

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Lecture Time: MoWe 10:30AM - 11:50AM, E1-101 (L01)
- Lab Session: Th 03:30PM - 04:20PM; Th 04:30PM - 05:20PM; Tu 03:00PM - 03:50PM (E1-227)
- Instructor:
  - Yanlin Zhang, yanlinzhang@hkust-gz.edu.cn
  - Office hour: Wed 13:00-14:00, E1-5F-511
- TAs
  - Houcheng Su, hsu638@connect.hkust-gz.edu.cn
  - Chuhan Hu, chu083@connect.hkust-gz.edu.cn
  - Hongru Chu, hchu853@connect.hkust-gz.edu.cn
  - Yucheng Xu, yxu662@connect.hkust-gz.edu.cn
  - Hongyi Duan, hduan469@connect.hkust-gz.edu.cn
  - Yusen Hou, yhou925@connect.hkust-gz.edu.cn
  - Xinjie Shi, xshi667@connect.hkust-gz.edu.cn

Design and Analysis of Algorithms, Fall 2026

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Lecture Time: TuTh 01:30PM - 02:50PM, E1-134 (L02)
- Lab Session: Th 03:30PM - 04:20PM; Th 04:30PM - 05:20PM; Tu 03:00PM - 03:50PM (E1-227)
- Instructor:
  - Shangqi Lu, shangqilu@hkust-gz.edu.cn
  - Office hour: Wed 9:00-10:00 am, E1-3F-308
- TAs
  - Houcheng Su, hsu638@connect.hkust-gz.edu.cn
  - Chuhan Hu, chu083@connect.hkust-gz.edu.cn
  - Hongru Chu, hchu853@connect.hkust-gz.edu.cn
  - Yucheng Xu, yxu662@connect.hkust-gz.edu.cn
  - Hongyi Duan, hduan469@connect.hkust-gz.edu.cn
  - Yusen Hou, yhou925@connect.hkust-gz.edu.cn
  - Xinjie Shi, xshi667@connect.hkust-gz.edu.cn

Course Goals

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- The design and analysis of algorithms
  - They usually appear together

- By taking this course, you will
  - Obtain a good understanding of various data structures and algorithms
  - Learn to think analytically about algorithms
  - Learn to design and apply algorithms to solve computational problems effectively
  - Learn to implement and evaluate algorithms and data structures

- Prerequisite: Students are assumed to have completed a Data Structures course.

Contents (Tentative)

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Week	Topic	Content (if any)
1	Introduction	Review of basic data structure and asymptotic complexity
2	Preliminaries	Loop invariant in algorithms (Insertion sort, bubble sort, KMP algorithm, etc.)
3	Sorting algorithms	Divide and conquer algorithms (Merge sort, quick sort)
4	Dynamic Programming I	Knapsack Problem, etc.
5	Dynamic Programming II	Longest Common Subsequence, backtracking, etc.
6	Greedy algorithms	Scheduling, MST
7	Advanced algorithms I and Midterm Exam	Algorithms on Strings
8	Advanced algorithms I	
9	Advanced algorithms II	Algorithms on Trees
10	Graph algorithms I	Graph representation, Graph traversal, Topological sorting, etc.
11	Graph algorithms II	Shortest path algorithms, etc.
12	Graph algorithms III	Network Flow, etc.
13	Complexity Theory	P/NP, NP-complete, etc.
-	Final exam	

Textbook and Materials

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Course website:

- Canvas and https://shangqilu.github.io/dsaa2043/

Textbook:

- Introduction to Algorithms. Cormen, Leiserson, Rivest, and Stein (CLRS)

Reference books:

- Algorithm Design. Kleinberg and Tardos
- The Algorithm Design Manual. Steven Skiena

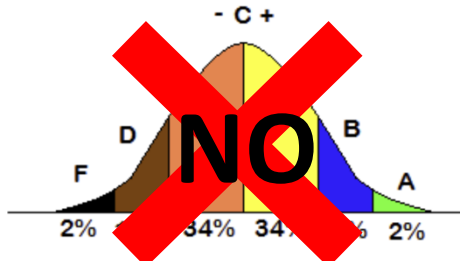
Online resources:

- MIT 6.006 - Introduction to Algorithms
- Stanford CS161 - Design and Analysis of Algorithms

Assessment and Grading (Tentative)

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Written assignment (20%):
  - Solving 60 OJ problems (15%) – Midterm style
  - Solving 1 set of algorithm design problems (5%) – Final exam style
- Individual Project Report (10%):
  algorithm design project (release after midterm)
- Mid-term exam (30%):
  closed book, on computer, week 7, Saturday, Oct. 24, 1:30pm to 3:00 pm
- Final exam (40%):
  closed book, paper-based
- We assess student performance using criterion-referencing approach. In addition to the criterion written in course syllabus, you can estimate your performance from your course work score:
  - A level: [85, 100]
  - B level: [70, 85)
  - C level: [55, 70]
  - D level: [40, 55)
  - F level: [0, 40)



More on OJ

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- OJ Problems (15%)
  - DDL: **2026/12/7 (Last day of Teaching)**
  - Release: week 3.
  - **Late submission is not allowed.**
  - Progress requirement: By **Sunday** of week x, you must have solved at least (x−2) OJ problems. Failure to meet this requirement will result in a 1-point penalty for that week. Penalties are cumulative!!!

Problems	Pts
60	15
55-59	14
50-54	13
45-49	12
40-44	11
35-39	10
30-34	9
25-29	8
20-24	7
15-19	6
10-14	5
5-9	4
0-4	0

Safe pattern 1: Solving 1 problem per week before week 13, then solving remaining problems.
Safe pattern 2: Solving 3-6 problems per week.

Late submission policy

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- OJ problems: Late submission will not be accepted.
- algorithm design problems (5%), and project report (10%):
  - Late penalty: **20% * n** deduction (*n* = number of late days)
  - Grace period:
    - 1 hour = 0 day
    - >1 hour = 1 day
  - Example: Peter submits his assignment 2 days after the deadline, and the original score is 9 → Due to the late submission, his final score will be 9*60%.

How to get the most out of this course

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Preview, Participate, and Review matters!

- **Before class:**
  - Prepare for the lecture
- **During class:**
  - Class participation: ask any questions anytime
  - Engage with in-class questions and exercises
- **After class:**
  - Review contents timely and ask questions☺ → Don't wait until the day before exam ☹
  - Do exercises
- **Generative AI:**
  - Using Generative AI to prepare and review course content is allowed.
  - Don't use it (brainlessly) to solve exercise.
    - Learning requires generation by you (not AI)
    - Learning algorithm do require learning abstraction, in-depth thinking, and asking critical questions!

Midterm Exam

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- **Time: 13:30–15:00, Sat, Oct. 24, 2026**
- Location: TBA (Lab room)
- Format: On computer
  - solve algorithmic problems by writing **correct** code
  - Pseudocode is accepted for partial credit, typically around **50–70%**, depending on its correctness and completeness.
- Preparation:
  - Having a good understanding of algorithms covered in class
  - Learn to write code without LLM (i.e., no copilot, no cursor, etc.)
  - Know basic python grammar such as use *self* in a python class.

Algorithm

Data Structure Recap

"A **data structure** is **a way to store and organize data** in order to facilitate **access** and **modifications**. Using the appropriate data structure or structures is an important part of algorithm design. *No single data structure works well for all purposes*, and so you should know the strengths and limitations of several of them."

- CLRS

Algorithm

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- "An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output** in a finite amount of time. An algorithm is thus **a sequence of computational steps that transform the input into the output**." (From *C.L.S.R)*
- All algorithms satisfy the following criteria:
  - **Input**: 0 or more quantities are supplied.
  - **Output**: At least one quantity is produced.
  - **Definiteness**: Each instruction is clear and unambiguous.
  - **Finiteness**: For all cases, the algorithm terminates after a finite number of steps.
  - **Effectiveness**: Every instruction must be basic enough (feasible).

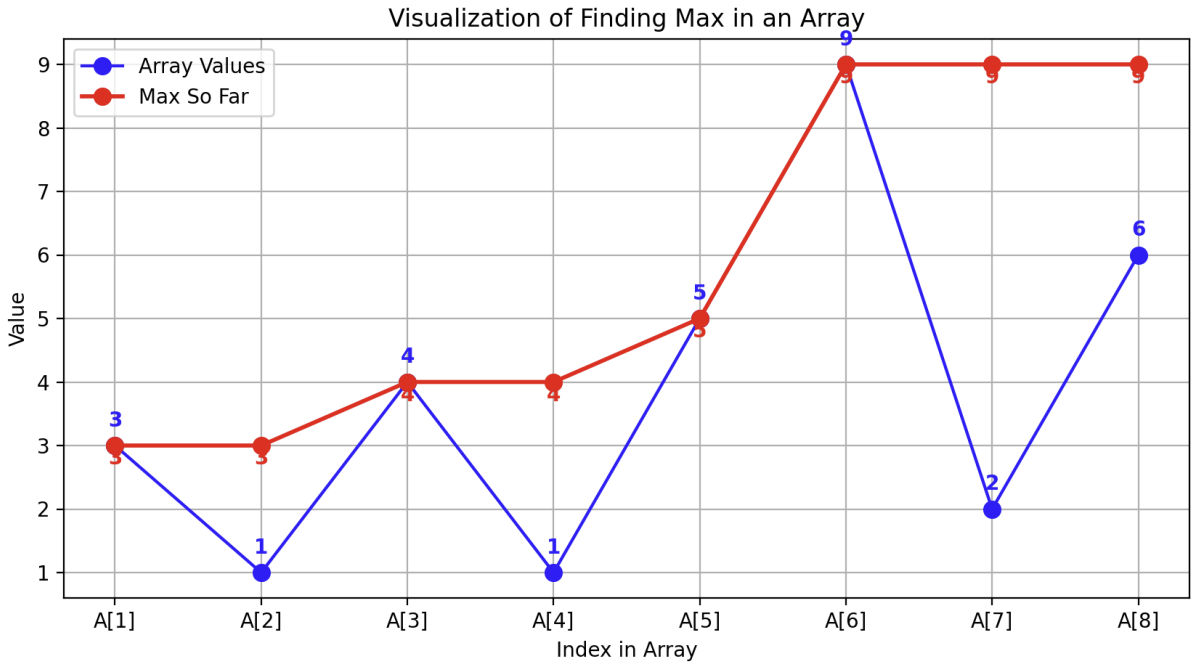
Data Structures and Algorithms

- **Algorithm:** Outline, the essence of a computational procedure, step-by-step instructions
- **Program:** an implementation of an algorithm in some programming language
- **Data structure:** Organization of data needed to solve the problem

Example: findMax pseudo-code

```
Algorithm FindMax(A)
Input: An array A with n elements
Output: The maximum value in A

1. max_so_far ← A[1]  // Initialize max with the first element
2. for i ← 2 to length(A) do
3.     if A[i] > max_so_far then
4.         max_so_far ← A[i]  // Update max if a larger value is found
5.     end if
6. end for
7. return max_so_far
```



Asymptotic Analysis

What is a Good Algorithm?

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Efficient
  - Running time
  - Space used

- Efficiency as a function of input size
  - The number of bits in an input number
  - Number of data elements (numbers, points)

What is Analysis of Algorithms

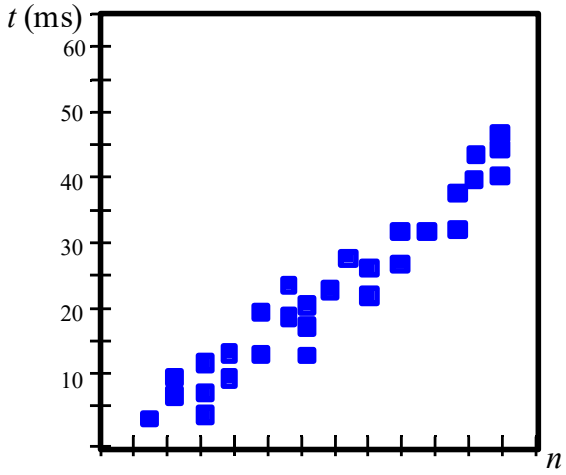
Estimate the running time.

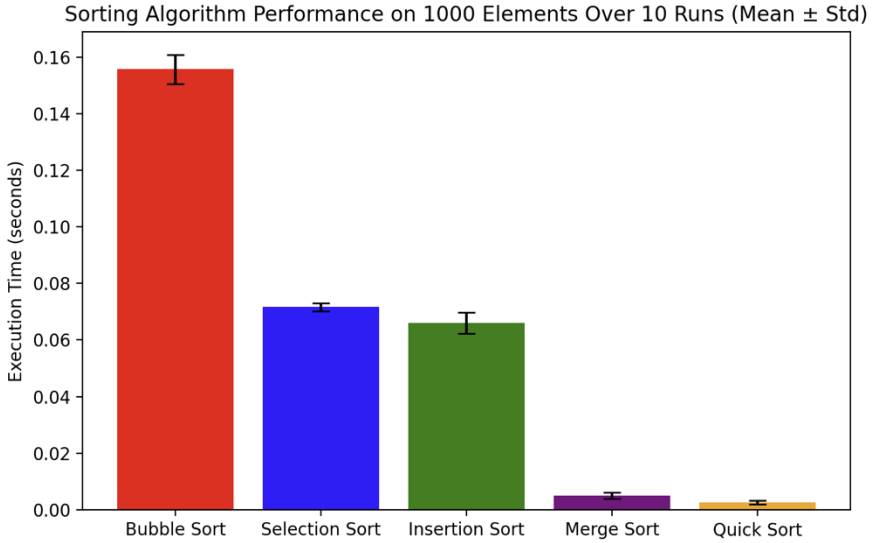
Estimate the memory space required.

Time and space depend on the input size.

Measuring the Running Time Experimentally

How should we measure the running time of an algorithm?





Experimental Study

- Write a program that implements the algorithm
- Run the program with data sets of varying size and composition
- Use a system call to get an accurate measure of the actual running time

Limitations of Experimental Studies

- Must implement and test the algorithm to determine its running time
- Experiments done only on a limited set of inputs
  - May not be indicative of the running time on other inputs not included in the experiment
- To compare two algorithms, the same hardware and software environments needed

Beyond Experimental Studies

- We will develop a general methodology for analyzing running time of algorithms. This approach
  - Uses a high-level description (pseudocode) of the algorithm instead of testing one of its implementations
  - Considers all possible inputs
  - Evaluates the efficiency of any algorithm being independent of the hardware and software environment
- To achieve that, we need to
  - Make simplifying assumptions about the running time of each basic (primitive) operations
  - Study how the number of primitive operations depends on the size of the problem solved

Primitive Operations

Simple computer operation that can be performed in time that is always the same, independent of the size of the bigger problem solved (we say: constant time)

- **Assigning a value to a variable:** x ←1 — T_{assign}
- **Calling a method:** Expos.addWin() — T_{call}
  - Note: doesn't include the time to execute the method
- **Returning from a method**: return x; — T_{return}
- **Arithmetic operations on primitive types** — T_{arith}

  x + y, r*3.1416, x/y, etc.
- **Comparisons on primitive types:** x==y — T_{comp}
- **Conditionals:** if (...) then.. else... — T_{cond}
- **Indexing into an array:** A[i] — T_{index}
- **Following object reference:** Expos.losses — T_{ref}

Note: Multiplying two Large Integers is *not* a primitive operation, because the running time depends on the size of the numbers multiplied.

More assumptions

- Counting each type of primitive operations is tedious
- The running time of each operation is roughly comparable:

$$T_{assign} \approx T_{comp} \approx T_{arith} \approx \ldots \approx T_{index} = 1 \text{ primitive operation}$$

- We are only interested in the **number** of primitive operations performed

Estimating Running Time for FindMin

```
Algorithm findMin(A, start, stop)
Input: Array A, index start & stop
Output: Index of the smallest element of A[start:stop]
minvalue ← A[start]
minindex ← start
index ← start + 1
while ( index <= stop ) do {
   if (A[index]<minvalue)
   then {
        minvalue ← A[index]
        minindex ← index
   }
   index = index + 1
}
return minindex
```

Running time	
$T_{index} + T_{assign}$	
T_{assign}	
$T_{arith} + T_{assign}$	
$T_{comp} + T_{cond}$	repeated stop-start times
$T_{index} + T_{comp} + T_{cond}$	
$T_{index} + T_{assign}$	
T_{assign}	
$T_{assign} + T_{arith}$	
$T_{comp} + T_{cond}$ (last check of loop)	
T_{return}	

Estimating Running Time for FindMin

- Running time depends on *n* = stop – start + 1
- $T(n) \leq 8 + 10 * (n-1)$ primitive operations
  - 8 primitive operations outside the loop
  - At most 10 primitive operations inside the loop
- How will the running time change for the following two input instances?

5	4	3	2	1	0

0	1	2	3	4	5

Picking and Placing Cards at Hand

Poker-Style Insertion Sort (magic power)

Table: 5 2 4 6 1 3
Hand:

Table: 2 4 6 1 3
Hand: 5

Table: 4 6 1 3
Hand: 2 5

Table: 6 1 3
Hand: 2 4 5

Table: 1 3
Hand: 2 4 5 6

Table: 3
Hand: 1 2 4 5 6

Table:
Hand: 1 2 3 4 5 6

Strategy

- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

Poker-Style Insertion Sort (no magic power)

Table: 5 2 4 6 1 3
Hand:

Table: 2 4 6 1 3
Hand: 5

able: 4 6 1 3
and: 2 5

able: 6 1 3
and: 2 4 5

able: 1 3
and: 2 4 5 6

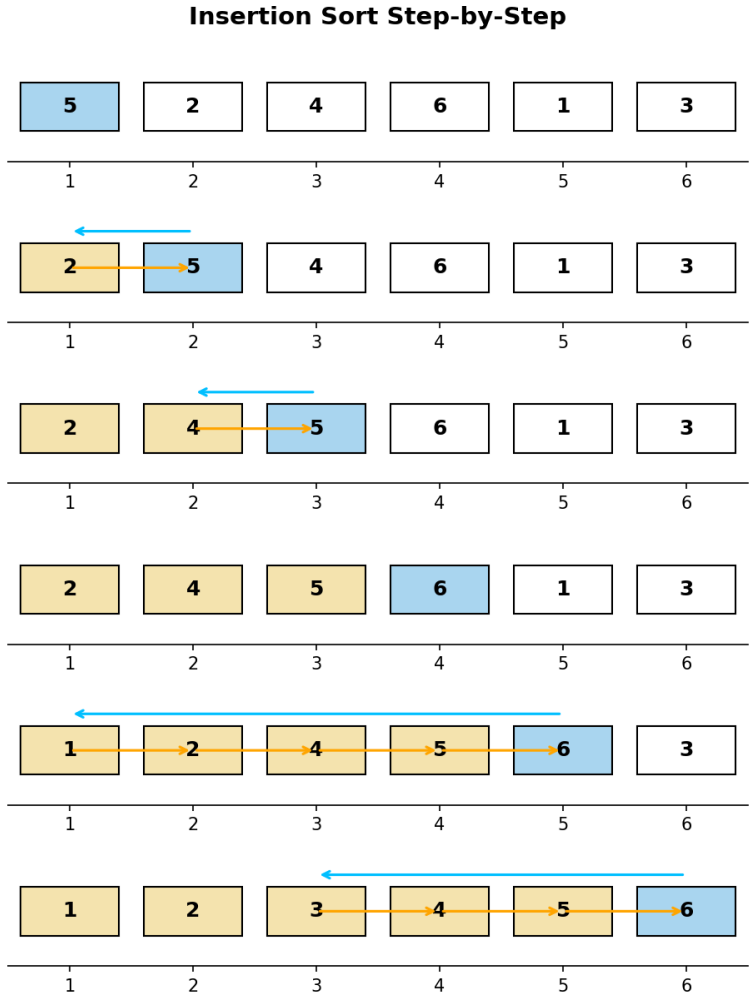
Table: 3
Hand: 1 2 4 5 6

Table:
Hand: 1 2 3 4 5 6

Insertion Sort (Recap.)

```
Algorithm InsertionSort(A)
Input: An array A with n elements
Output: A sorted in non-decreasing order

1. for i ← 2 to length(A) do
2.     key ← A[i]  // Current element to be inserted
3.     j ← i - 1   // Start comparing from the previous element
4.     while j > 0 and A[j] > key do
5.         A[j + 1] ← A[j]  // Shift element to the right
6.         j ← j - 1
7.     end while
8.     A[j + 1] ← key  // Place key in the correct position
9. end for
10. return A  // Sorted array
```



Insertion Sort

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

t_i : the number of while-loops used for inserting A[i]

Total time $T(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{i=2}^{n} t_i(c_5 + c_6 + c_7)$

$$-(c_2 + c_4 - c_6 - c_7 + c_8)$$

Best/Worst/Average Case

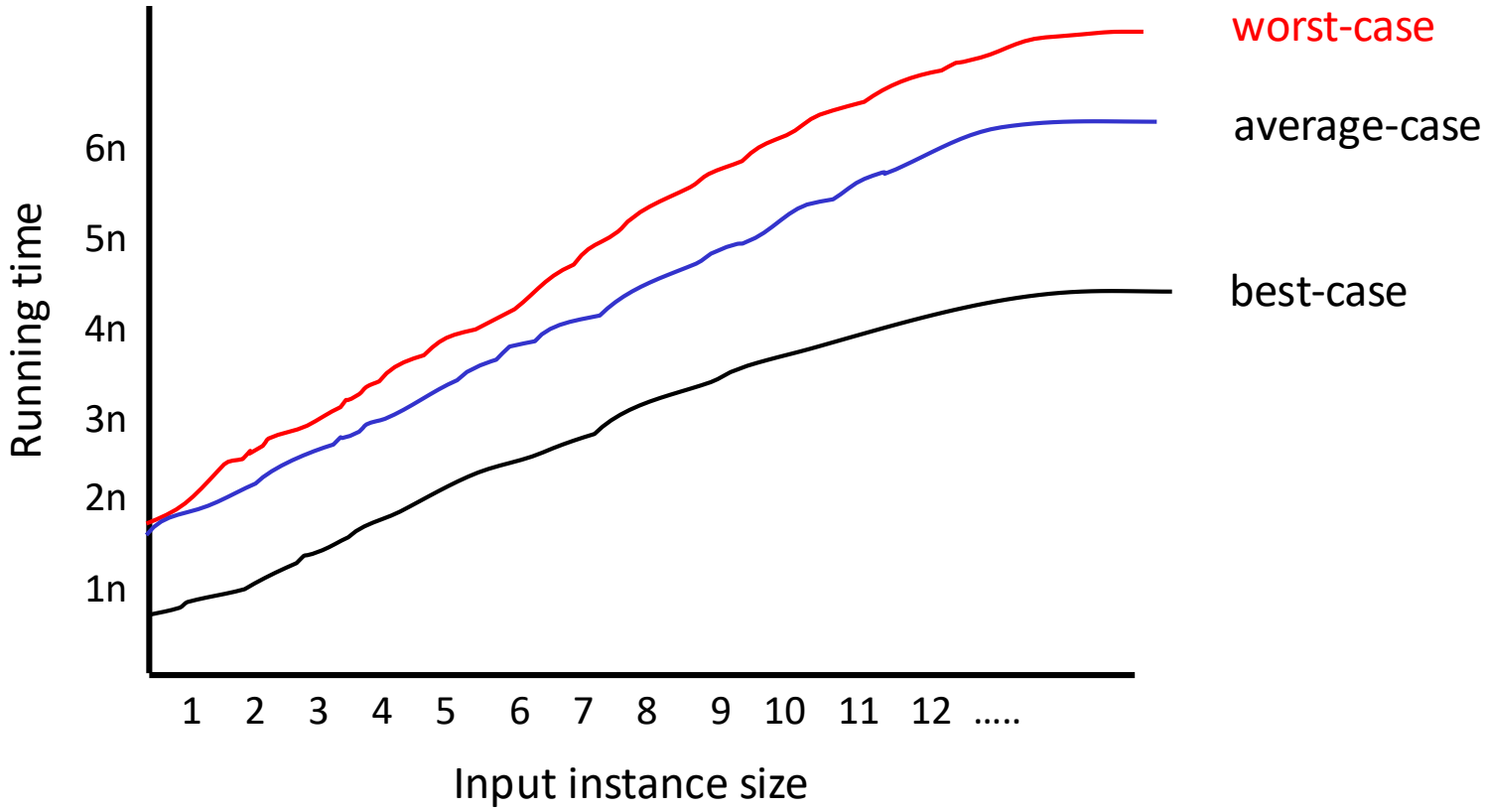
$$\text{Total time } T(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{i=2}^{n} t_i(c_5 + c_6 + c_7)$$

$$-(c_2 + c_4 - c_6 - c_7 + c_8)$$

- Best case:
  - elements are already sorted; Each element only does one comparison (t_i=1), running time = f(n), i.e., linear time
- Worst case:
  - elements are sorted in reverse order; each element shifts i-1 times (t_i=i-1), running time = f(n^2), i.e., quadratic time
- Average case:
  - Each element moves above half of its maximum shifts (t_i=(i-1)/2), running time = f(n^2), i.e., quadratic time

Best/Worst/Average Case

- For inputs of all sizes:



Best/Worst/Average Case

- Worst case is usually used
  - It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the worst-case time complexity is of crucial importance
  - For some algorithms worst case occurs fairly often
  - Average case is often as bad as worst case
- Finding average case can be very difficult
- The best (fastest) case is seldom of interest

Asymptotic Analysis

- Consider an arbitrary problem. Suppose that Algorithm 1 runs in
  - $1000n \cdot c_{multi} + 10n \cdot c_{mem}$ time,
- where c_{multi} is the time of one multiplication and c_{mem} the time of one memory access; Algorithm 2 runs in
  - $10n \cdot c_{multi} + 100n \cdot c_{mem}$ time.

- Which one is better? **It depends** on the specific values of c_{multi} and c_{mem}, which vary from machine to machine.

Why Not Constants?

- Consider an arbitrary problem. Suppose that Algorithm 1 runs in
  - $1000n \cdot c_{multi} + 10n \cdot c_{mem}$ time,
- where c_{multi} is the time of one multiplication and c_{mem} the time of one memory access; Algorithm 2 runs in
  - $10n \cdot c_{multi} + 100n \cdot c_{mem}$ time.

- In mathematics, we want to make a universally correct conclusion, which holds on all machines:
  - Regardless of the machine, their costs differ by at most a constant factor.

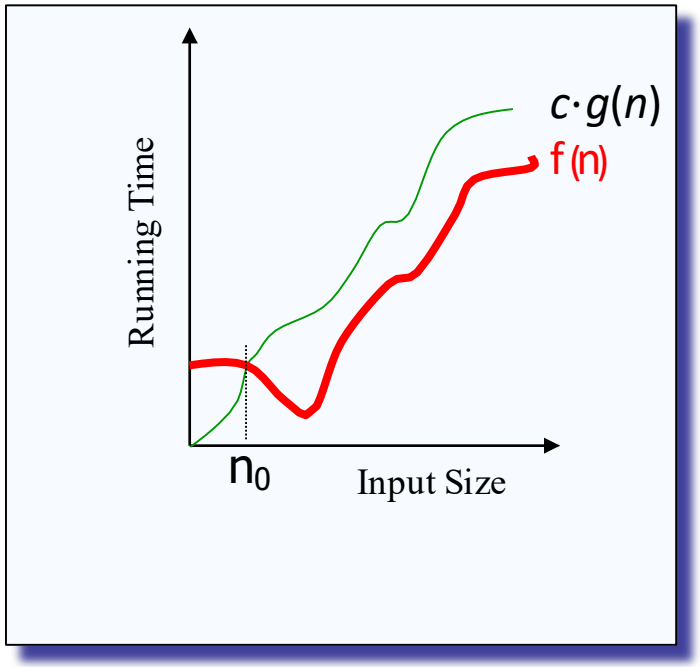
Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
  - like "rounding": 1,000,001 ≈ 1,000,000
  - $3n^2 \approx n^2$
- Capturing the essence:
  - The growth of the running time with the problem size n
  - We care about the efficiency of the problem when n is large (think: why?)

Asymptotic Notation

- The "big-Oh" O-Notation
  - asymptotic upper bound
  - let $f\ (n)$ and $g(n)$ be two functions of n.
  - we say that $f(n)$ grows asymptotically no faster than $g(n)$ if there exists positive constants c and n_0, s.t.

    $f(n) \leq\ c \cdot g(n)$ for all $n\ \geq\ n_0$.
  - we can denote this by $f(n)\ =\ O(g(n))$.

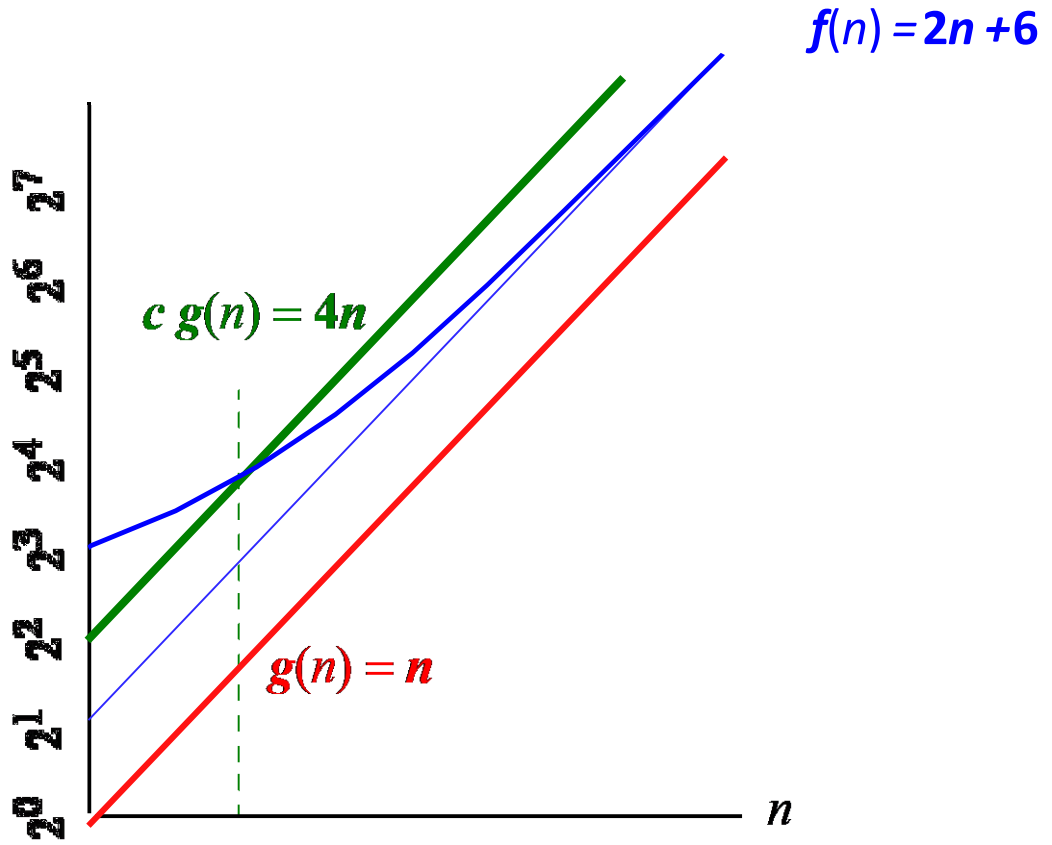


- Used for worst-case analysis
- Art of Computer Science:
  - Minimize the growth of running time in solving a problem

Example

For functions $f(n)$ and $g(n)$ there are positive constants c and n_0 such that: $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

conclusion: 2n+6 is O(n)



Another Example

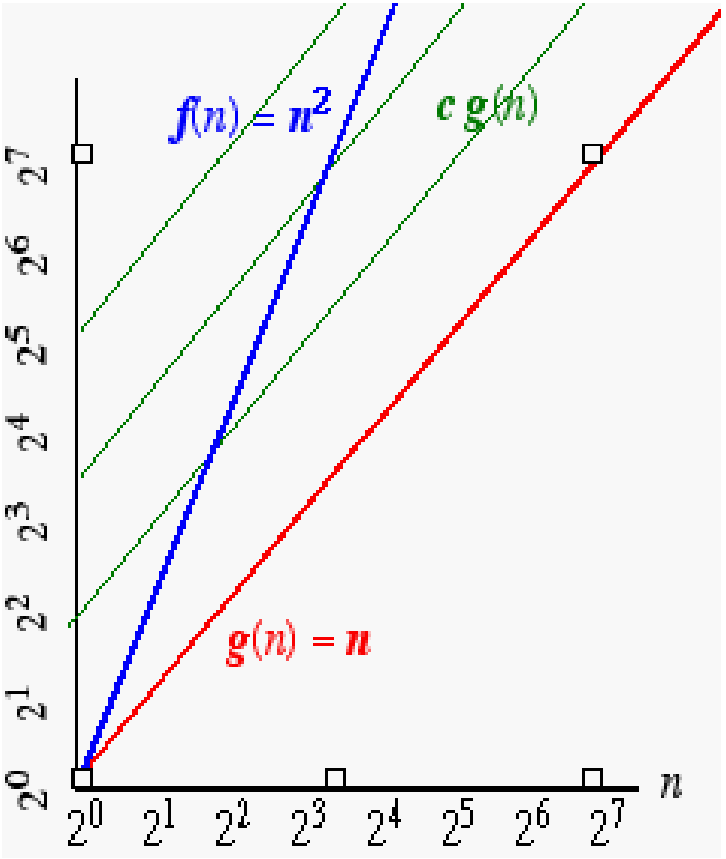
香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

On the other hand…

n^2 is not $O(n)$ because there is no c and n_0 such that:

$$n^2 \leq cn \text{ for } n \geq n_0$$

The graph to the right illustrates that no matter how large a c is chosen there is an n big enough that $n^2 > cn$.



Asymptotic Notation

- **Simple Rule:** drop lower order terms and constant factors.
  - $50n\log n$ is $O(n\log n)$
  - $7n-3$ is $O(n)$
  - $8n^2\log n + 5n^2 + n$ is $O(n^2\log n)$

- **An interesting fact:**

  $\log_{b_1} n = O(\log_{b_2} n)$ for any constants $b_1 > 1$ and $b_2 > 1$.

- Because of the above, in computer science, we omit all the constant logarithm bases in big-O. For example, instead of $O(\log_2 n)$, we will simply write $O(\log n)$.

Asymptotic Analysis of Running Time

- Use O-notation to express number of primitive operations executed as function of input size.
- Comparing asymptotic running times
  - an algorithm that runs in $O(n)$ time is better than one that runs in $O(n^2)$ time
  - similarly, $O(\log n)$ is better than $O(n)$
  - hierarchy of functions: $\log n < n < n^2 < n^3 < 2^n$
- Caution! Beware of very large constant factors. An algorithm running in time 1,000,000 n is still $O(n)$ but might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis


```
Algorithm prefixAverages1(X)
```

Input: An n-element array X of numbers

Output: An n-element array A of numbers such that A[i] is the average of elements X[1], ..., X[i]

```
for i ← 1 to n do
    a ← 0
    for j ← 1 to i do
        a ← a+X[j]
    A[i] ← a/i
return array A
```

←1 step

i iterations with i=1,2,...,n

n iterations

Analysis: running time is $O(n^2)$

A Better Algorithm

Algorithm prefixAverages2(X)

Input: An *n*-element array X of numbers

Output*:* An *n*-element array A of numbers such that A[*i*] is the average of elements X[1], ... , X[*i*]

```
s ← 0
for i ← 1 to n do
   s ← s + X[i]
   A[i] ← s/i
return array A
```

Analysis: Running time is …

Asymptotic Notation (Terminology)

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Special classes of algorithms:
  - Logarithmic: $O(\log n)$
  - Linear: $O(n)$
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k), k \geq 1$
  - Exponential: $O(a^n), a > 1$
- "Relatives" of the Big-Oh
  - $\Omega(f(n))$: Big Omega -asymptotic lower bound
  - $\Theta(f(n))$: Big Theta -asymptotic tight bound

What does O(1) mean?

We say $t(n)$ is $O(1)$, if there exist two positive constants n_0 and c such that, for all $n \geq n_0$.

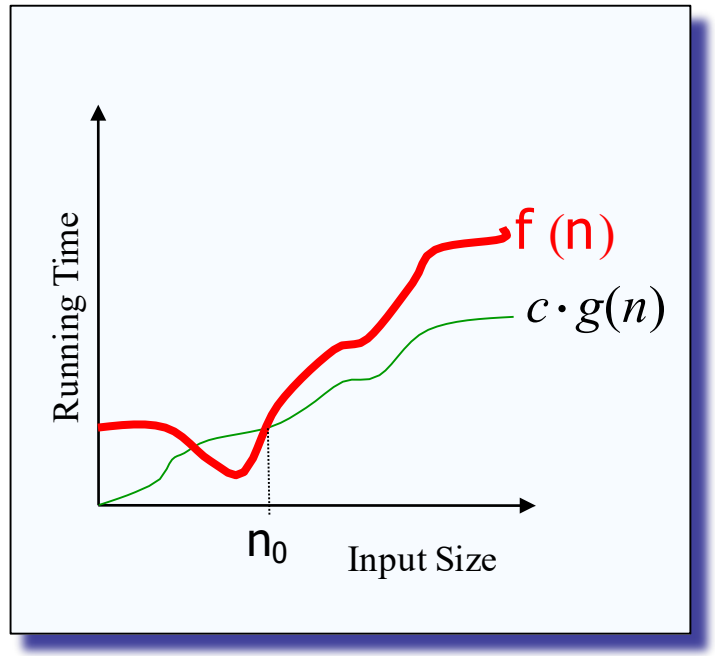
$$t(n) \leq c$$

So, it means that **$t(n)$ is bounded**.

Asymptotic Notation

- The "big-Omega" Ω−Notation
  - Asymptotic lower bound
  - Let $f\ (n)$ and $g(n)$ be two functions of n.
  - We say that $f(n)$ grows asymptotically no slower than $g(n)$ if there exists positive constants c and n_0, s.t.

    $f(n) \geq\ c \cdot g(n)$ for all $n\ \geq\ n_0$.
  - We can denote this by $f(n)\ = \Omega(g(n))$.

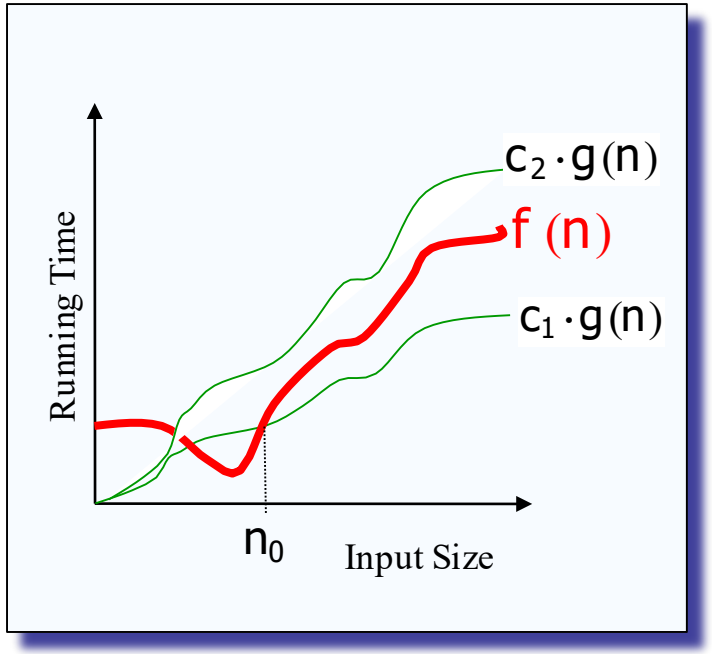


- Used to describe best-case running times or lower bounds for algorithmic problems
- E.g., lower-bound for searching in an unsorted array is $\Omega(n)$.

Asymptotic Notation

- The "big-Theta" Θ –Notation
  - asymptotically tight bound
  - let $f\ (n)$ and $g(n)$ be two functions of n.
  - If $f(n) = \mathrm{O}(g(n))$ and $f(n) = \Omega(g(n))$, then we define $f(n) = \Theta(g(n))$ to indicate that $f(n)$ grows asymptotically as fast as g(n).
  - $f(n) = \Theta(g(n))$ if and only if there exists positive constants c_1, c_2 and n_0, s.t.

    $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.



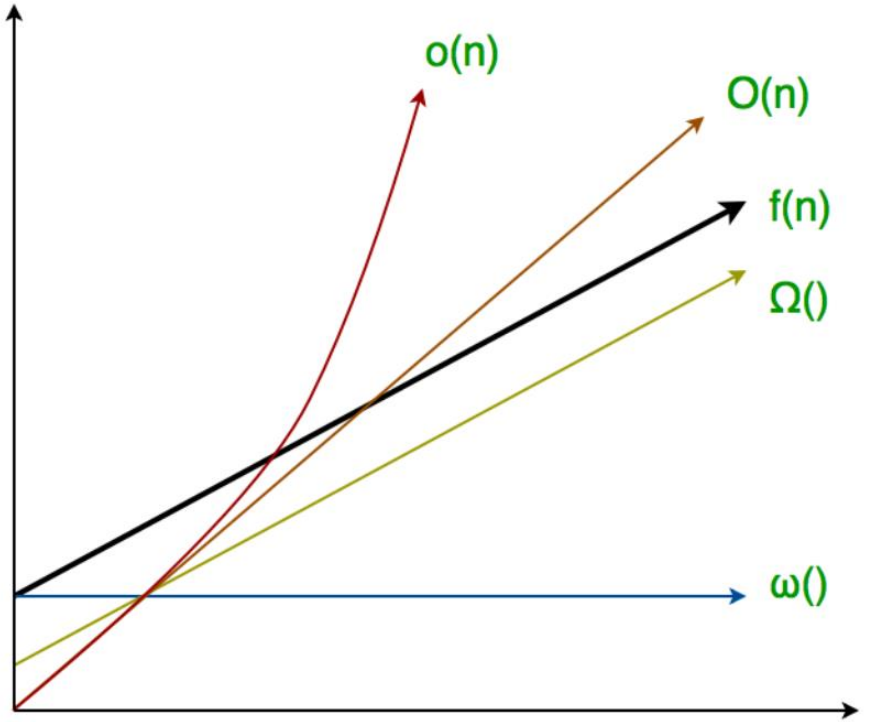
Asymptotic Notation

Two more asymptotic notations

- "Little-Oh" notation $f(n)$ is $o(g(n))$ non-tight analogue of Big-Oh
  - For every $c > 0$, there should exist n_0 , s.t. $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
  - Used for **comparisons** of running times. If $f(n)$ is o(g(n)), it is said that $g(n)$ dominates $f(n)$.
- "Little-omega" notation $f(n)$ is $\omega(g(n))$ non-tight analogue of Big-Omega

Asymptotic Notation

- Analogy with real numbers
  - $f(n)$ is $O(g(n))$ ≅ $f \leq g$
  - $f(n)$ is $\Omega(g(n))$ ≅ $f \geq g$
  - $f(n)$ is $\Theta(g(n))$ ≅ $f = g$
  - $f(n)$ is $o(g(n))$ ≅ $f < g$
  - $f(n)$ is $\omega(g(n))$ ≅ $f > g$

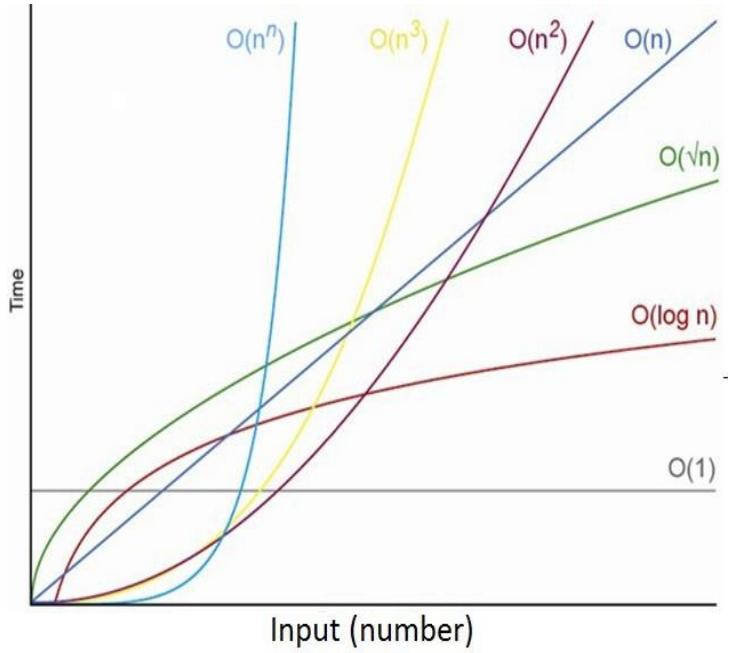


- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$

The big O (resp. big Ω) denotes a tight upper (resp. lower) bounds, while the little o (resp. little ω) denotes a loose upper (resp. lower) bounds.

Practical meaning of big O…

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}



If the unit is in seconds, this would make ~10^{11} years…

Constant Factor Rule

Suppose f(n) is O(g(n)) and a is a positive constant.
Then, $a \cdot f(n)$ is also $O(g(n))$

Proof: By definition, if f(n) is O(g(n)) then there exists two positive constants n_0 and c such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Thus, $$a \cdot f(n) \leq a \cdot c \cdot g(n)$$

We use the constant $a \cdot c$ to show that $a \cdot f(n)$ is $O(g(n))$.

Multiplying a function by a constant does not change its Big O upper bound since these upper bounds ignore constants

Sum Rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) + f_2(n)$ *is* $O(g(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g(n), \text{ for all } n \geq n_2$$

So, $f_1(n) + f_2(n) \leq (c_1+c_2)g(n), \text{ for all } n \geq \max(n_1, n_2)$.

We can use the constants $c_1 + c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

A sum of two functions is inferior to a sum of two greater functions

Product Rule

Suppose $f_1(n)$ *is* $O(g(n))$ and $f_2(n)$ *is* $O(g(n))$.

Then, $f_1(n) \cdot f_2(n)$ *is* $O(g_1(n) \cdot g_2(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f_1(n) \leq c_1 g_1(n), \text{ for all } n \geq n_1$$

$$f_2(n) \leq c_2 g_2(n), \text{ for all } n \geq n_2$$

So, $f_1(n) \cdot f_2(n) \leq (c_1 \cdot c_2) \cdot (g_1(n) \cdot g_2(n))$, for all $n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

A product of two functions is less than a product of two greater functions

Transitivity Rule

Suppose $f(n)$ *is* $O(g(n))$ and $g(n)$ *is* $O(h(n))$.

Then, $f(n)$*is* $O(h(n))$.

Proof: Let n_1, c_1 and n_2, c_2 be constants such that

$$f(n) \leq c_1 g(n), \text{ for all } n \geq n_1$$

$$g(n) \leq c_2 h(n), \text{ for all } n \geq n_2$$

So, $f(n) \leq (c_1 \cdot c_2) h(n), \text{ for all } n \geq \max(n_1, n_2)$.

We can use the constants $c_1 \cdot c_2$ and $\max(n_1, n_2)$ to satisfy the definition.

If a function A is greater than function B, and function B is greater than function C, then function A is greater than function C

Analyzing insertion sort

$O(n^2)$ sorting algorithm: Insertion Sort

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

Best case: A is sorted, while loop does not execute.

$$\mathrm{T}(n) = n(c_1 + c_2 + c_4 + c_5 + c_8) - (c_2 + c_4 + c_5 + c_8) = O(n)$$

$O(n^2)$ sorting algorithm: Insertion Sort

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

Worse case: A is reverse-ordered. The while loop execute i-1 times for each i

$$\mathrm{T}(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{\mathrm{i}=2}^{n}(i-1)(c_5 + c_6 + c_7) - (c_2 + c_4 - \mathrm{c}_6 - c_7 + c_8)$$

$$= (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + \frac{n(n-1)(c_5 + c_6 + c_7)}{2} - (c_2 + c_4 - \mathrm{c}_6 - c_7 + c_8) = O(n^2)$$

$O(n^2)$ sorting algorithm: Insertion Sort

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

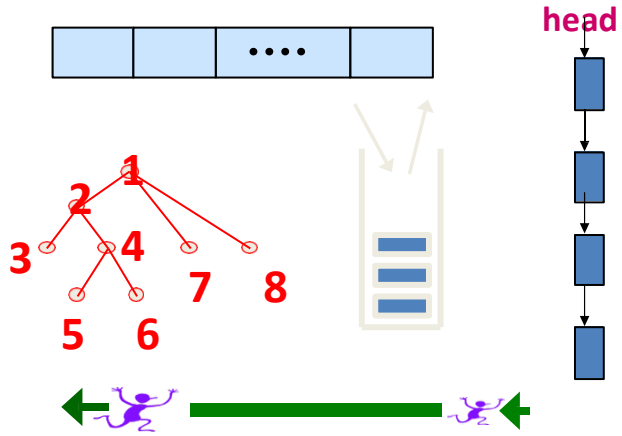
Average case: The while loop is expected to execute (i-1)/2 times for each i

$$\mathrm{T}(n) = n(c_1 + c_2 + c_4 + c_8 - c_6 - c_7) + \sum_{\mathrm{i}=2}^{n} \frac{i-1}{2}(c_5 + c_6 + c_7) \; -(c_2 + c_4 + \mathrm{c}_6 - c_7 - c_8)$$

$$= (c_1 + c_2 + c_4 + c_8 - c_6 - c_7)n + \frac{n(n-1)(c_5 + c_6 + c_7)}{4} - (c_2 + c_4 - \mathrm{c}_6 - c_7 + c_8) \; = O(n^2)$$

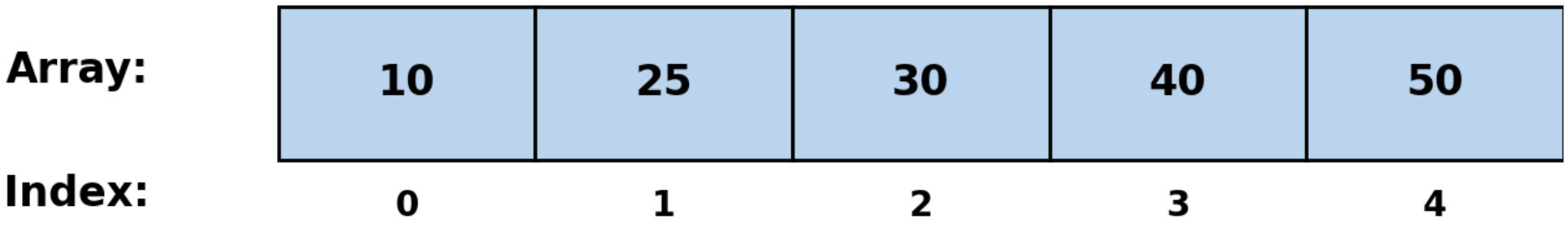
Data Structures (Prerequisite Recap)

- Arrays
- Lists
- Stacks
- Queues
- Trees



Array

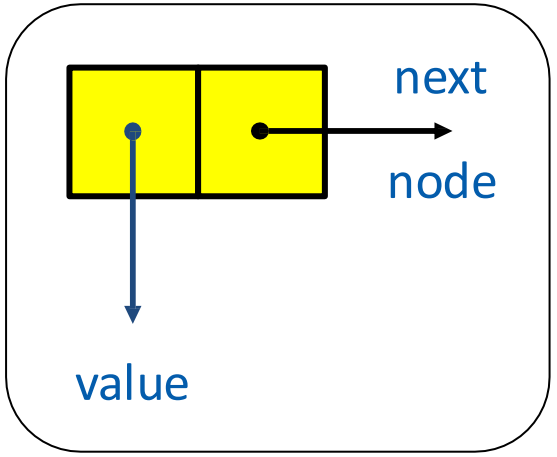
- An array is a linear data structure that stores a fixed-size collection of elements of the same type in continuous memory locations.
  - **Fixed Size** – Its size is declared at initialization and cannot be changed.
  - **Contiguous Memory Allocation** – Elements are stored sequentially in memory, making access fast (O(1) for direct access by index).
  - **Homogeneous** – All elements in an array must be of the same data type.
  - **Index-Based Access** – Elements are accessed using an index, starting from 1.

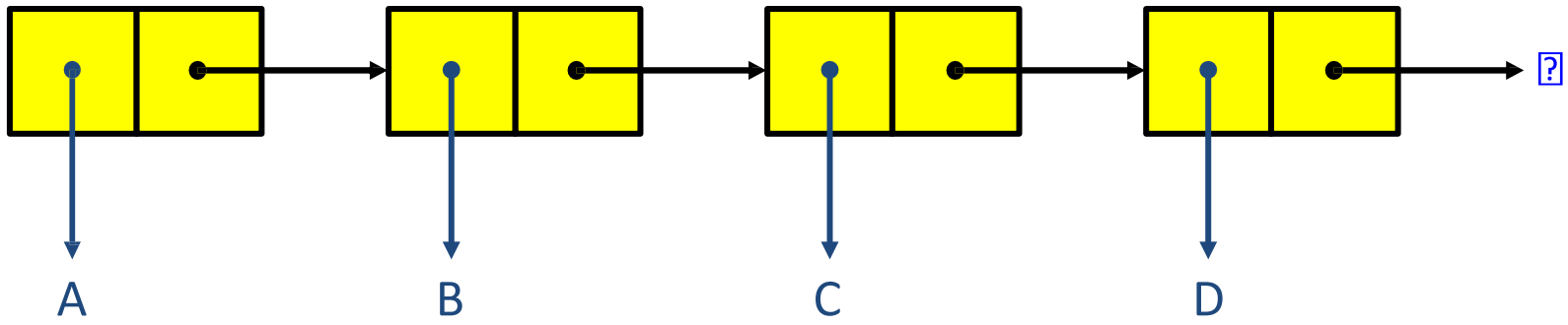


Recap: What if the array is full when appending a new element?

Singly Linked List

- A singly linked list is a dynamic data structure consisting of a sequence of nodes
- Each node contains two parts:
  - Data – Stores the actual value.
  - Next Pointer – Points to the next node in the list.
  - The last node's next pointer is NULL, indicating the end of the list.

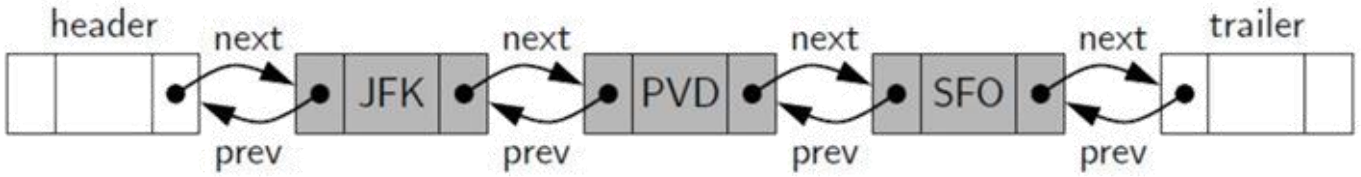




Recap: How to insert or remove items?

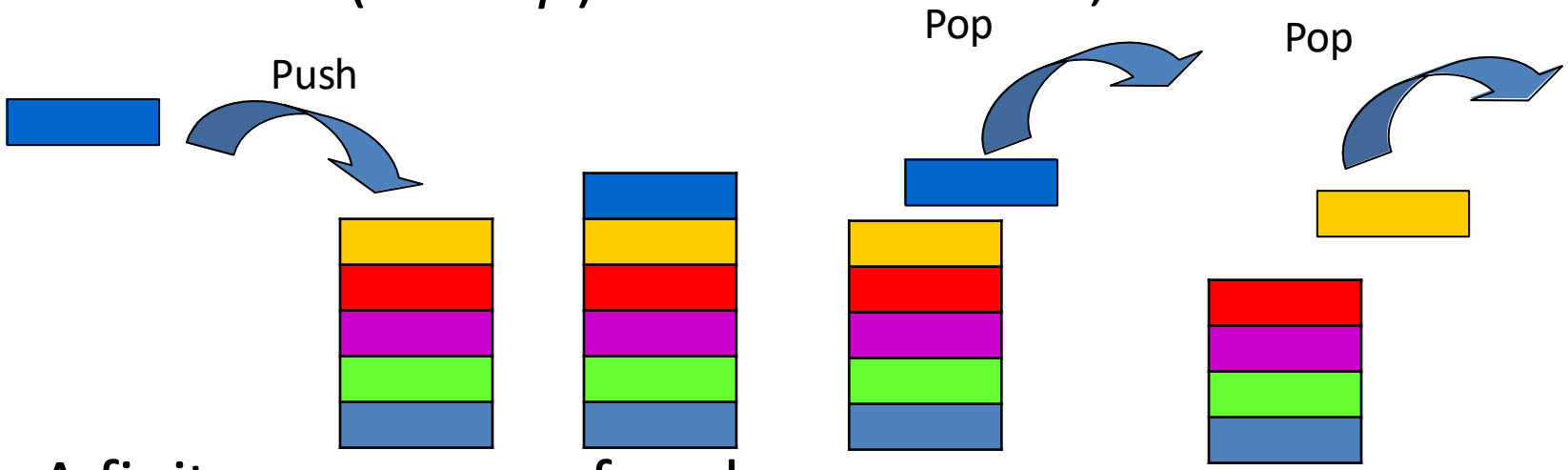
Doubly Linked List

- **Structure**: Each node has links to both the next and previous nodes.
- **Advantages**:
  - Quick insertions and deletions anywhere in the list (O(1) time).
- **Sentinel Nodes**:
  - Use special "header" and "trailer" nodes at the start and end.
  - These dummy nodes make handling edge cases easier.
  - In an empty list, the header points to the trailer, and the trailer points back to the header.

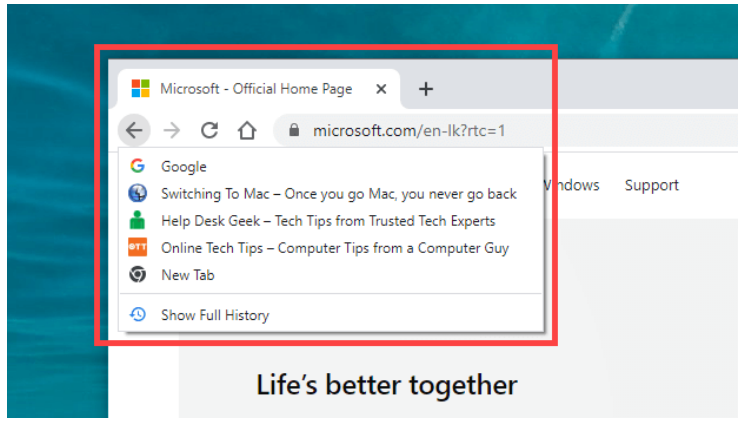


Stack

- **def.** A list for which Insert (push) and Delete (pop) are allowed only at one end of the list (the *top*) → **LIFO** – Last in, First out

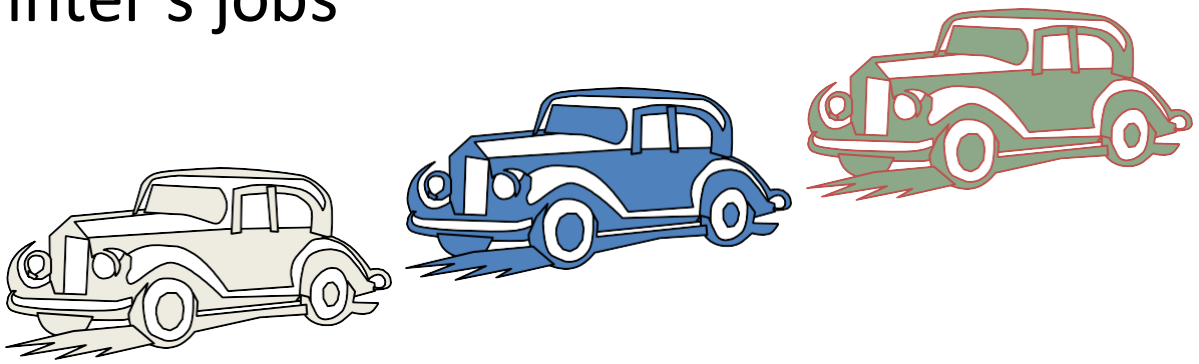


- **Objects:** A finite sequence of nodes
- **Operations:**
  - Push: Insert a new element
  - Pop: Remove and return top element
    - (the most recently inserted element)
- **Applications:** undo operations



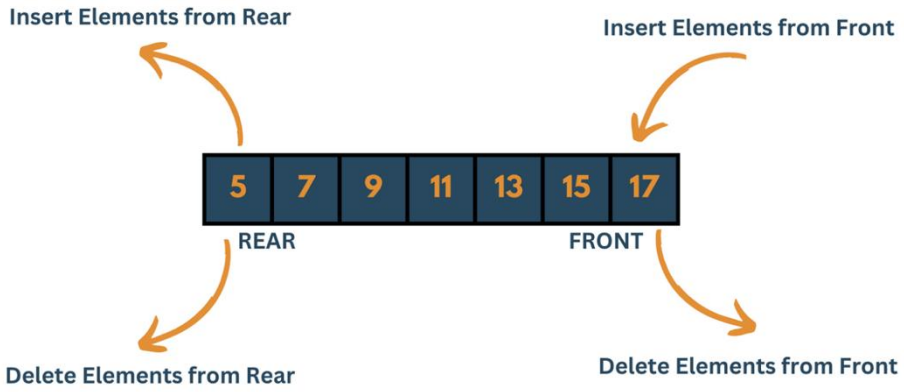
Queues

- **def.:** A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. This means that elements are added at the rear (enqueue) and removed from the **front** (dequeue).
- **Operations:**
  - Enqueue(x) → Adds x to the rear.
  - Dequeue() → Removes and returns the front (least recently inserted) element.
- **Applications:** printer's jobs



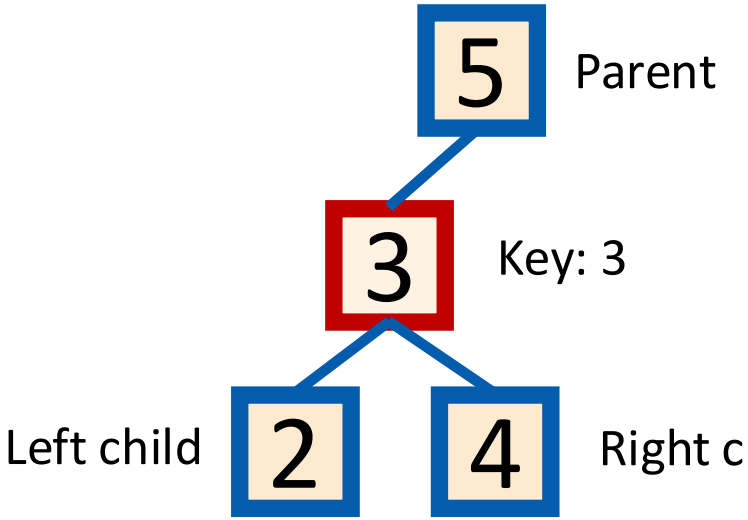
Deque: double-ended queue

- **Def.** A Deque (Pronounced ‘deck’) is a linear data structure that allows insertion and deletion from both ends (front and rear).
- Supports both FIFO and LIFO operations.
  - Insert and delete from both front and rear.
- More flexible than a normal queue.
  - Efficient O(1) insertion & deletion at both ends



Binary Tree Terminology

Every node stores a distinct element as its key and has at most two children.

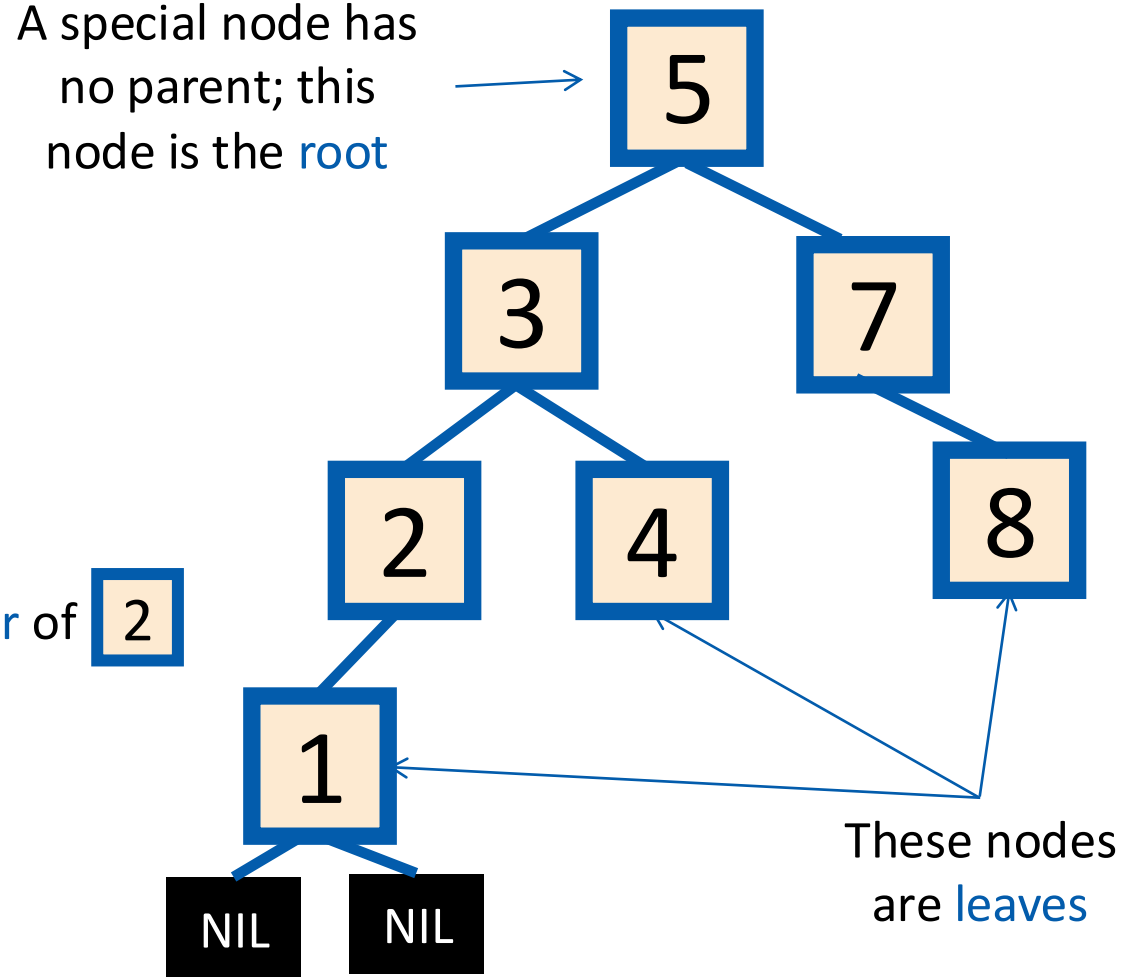


2 is a descendant of 5

5 is an ancestor of 2

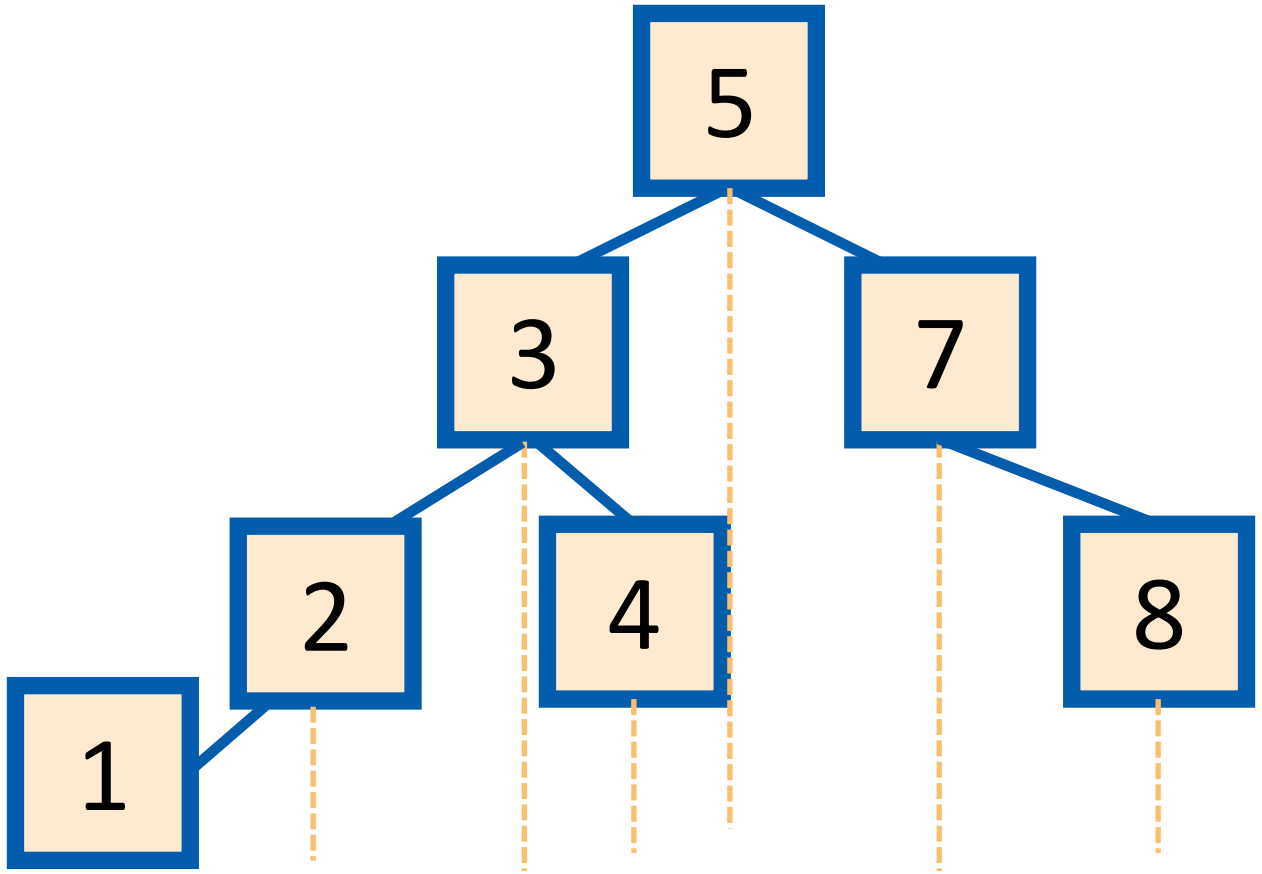
Both children of 1 are NIL (usually not drawn)

The height of this tree is 3



Binary Search Tree (BST)

- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.



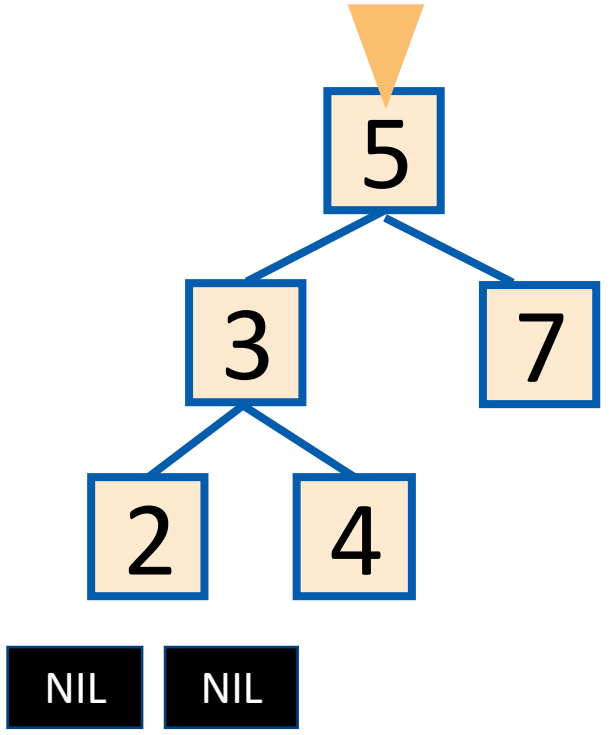
Q: Is this the only binary search tree I could possibly build with these values?

Traversal

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)

Pre-order / post-order traversal?



Sorted!

2 3 4 5 7

AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a binary search tree that also meets the following rule

AVL condition: For every node, the height of its left subtree and right subtree differ by at most 1.

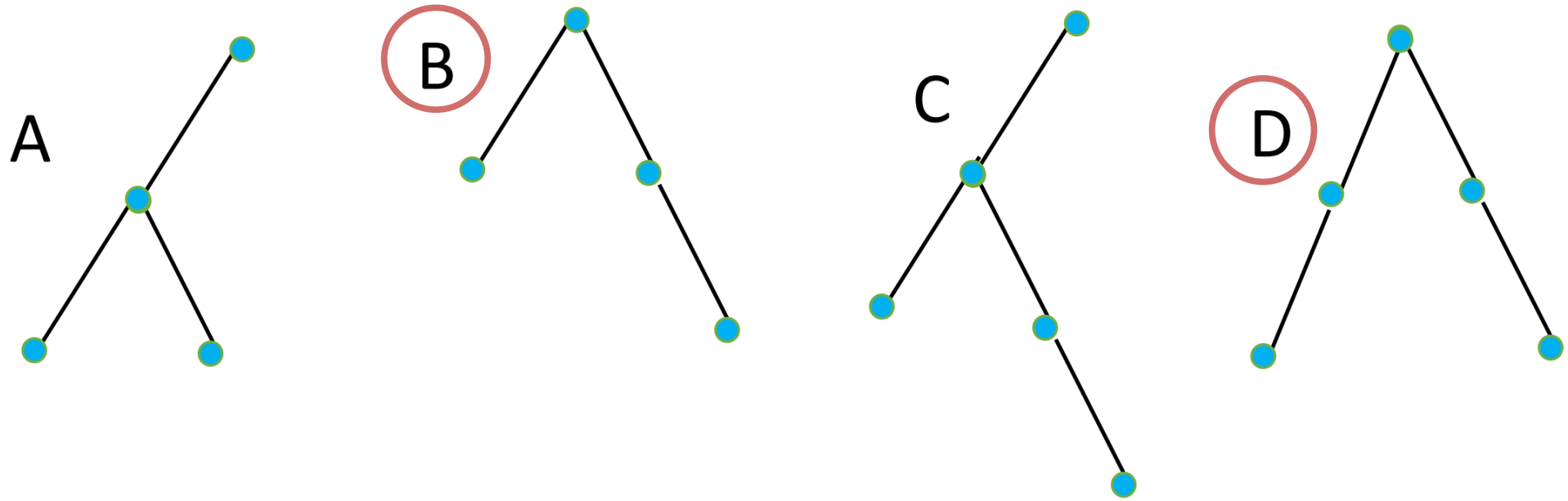
Height of a tree:
Maximum number of edges on a path from the root to a leaf.

A tree with one node has height 0.
A null tree (no nodes) has height -1.

Each node in the AVL tree stores the heights of its left and right subtrees.

AVL Trees

Which one(s) is balanced according to AVL's definition?

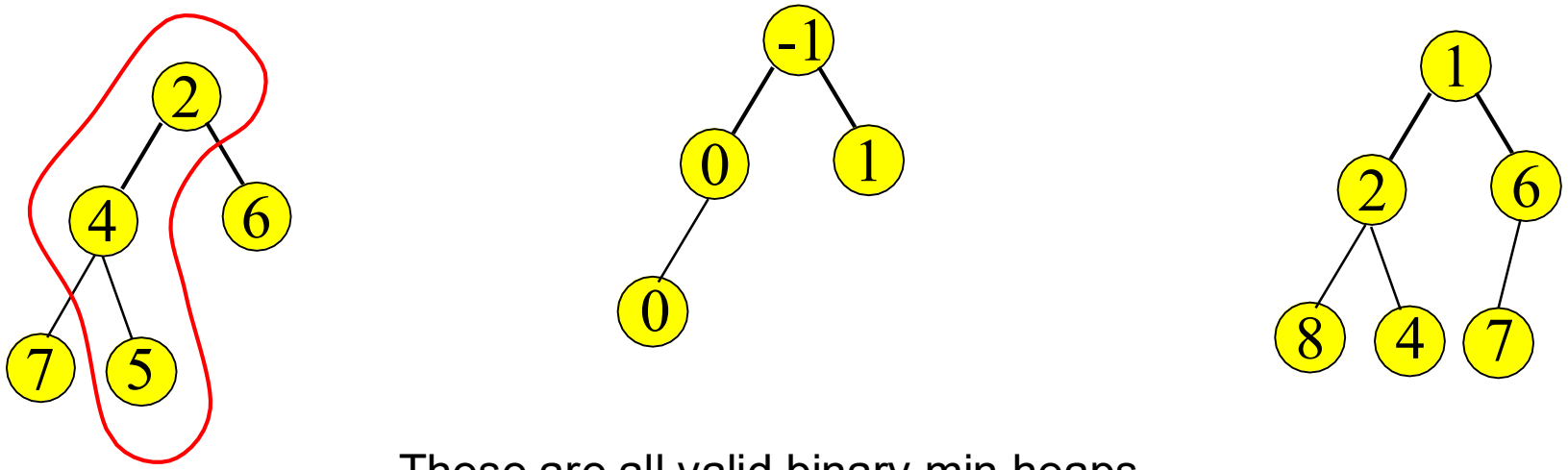


Binary Heaps

- A binary heap is a binary tree (NOT a B**S**T) that is:
  - Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  - Satisfies the **heap order property**
    - every node is less than or equal to its children (MinHeap, the default)
    - or every node is greater than or equal to its children (for MaxHeap)
- The root node is always the smallest node
  - or the largest, depending on the heap order (for MaxHeap)

Heap order property

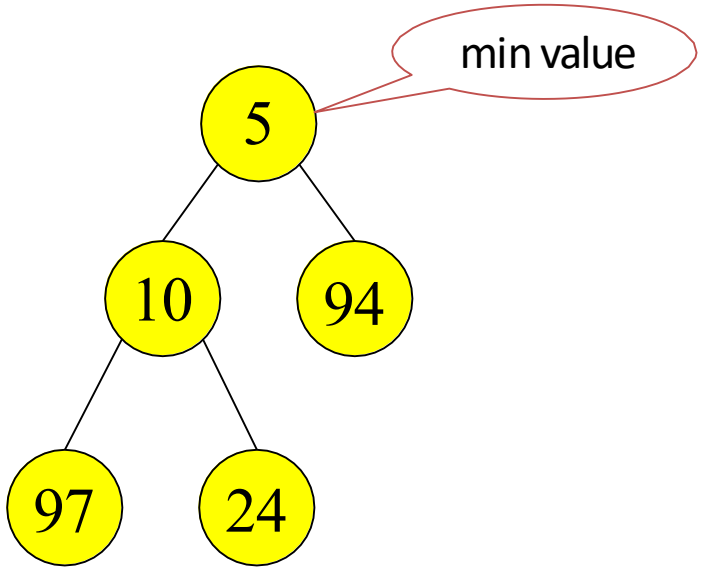
- A heap provides limited ordering information
- Each *path* is sorted, but the subtrees are not sorted relative to each other
  - A binary heap is NOT a binary search tree



These are all valid binary min heaps

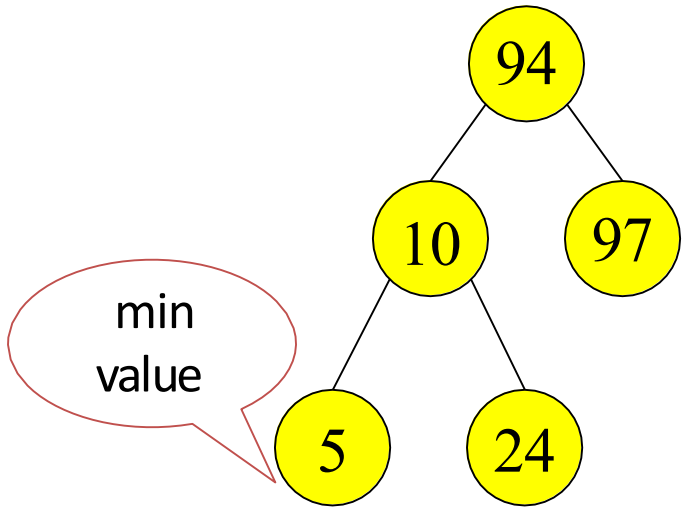
Binary Heap vs Binary Search Tree

Binary Heap



Parent is less than both left and right children

Binary Search Tree



Parent is greater than left child, less than right child

Structure Property

- A binary heap is a complete tree
  - All nodes are in use except for possibly the right end of the bottom row