

Data Structures and Algorithms

Lecture 03 – Sum of two numbers (Hashing, Algo complexity analysis, Sorting)

Pengju Ren

Institute of Artificial Intelligence and Robotics

Xi'an Jiaotong University

<http://gr.xjtu.edu.cn/web/pengjuren>

Problem Solving



Which two suspects had a total of 12,000 in stolen money?

Solution 1

- Brute_force

```
def two_sum_brute_force(nums, target):  
    n = len(nums)  
    for i in range(n):  
        for j in range(i + 1, n):  
            if nums[i] + nums[j] == target:  
                return [i, j]  
    return [] # 无解
```

Time Complexity: ? , Space Complexity: ?

Solution 2

- Hash_table

```
def two_sum_hash_table(nums, target):  
    num_to_index = {} # 哈希表: 数值 -> 下标  
    for i, num in enumerate(nums):  
        complement = target - num  
        if complement in num_to_index:  
            return [num_to_index[complement], i]  
        num_to_index[num] = i // 哈希表建立的过程, 判断补数 complement 是否  
存在于之前的 Hashing 中;  
    return [] # 无解
```

Time Complexity: ?, Space Complexity: ?

Solution 3

- Sorting + Dual pointers

```
def two_sum_two_pointers(nums, target):  
    # 存储原始下标  
    indexed_nums = [(num, i) for i, num in enumerate(nums)]  
    indexed_nums.sort() # 按数值排序  
    left, right = 0, len(indexed_nums) - 1  
    while left < right:  
        num_left, idx_left = indexed_nums[left]  
        num_right, idx_right = indexed_nums[right]  
        current_sum = num_left + num_right  
        if current_sum == target:  
            return [idx_left, idx_right]  
        elif current_sum < target:  
            left += 1  
        else:  
            right -= 1  
    return [] # 无解
```

Time Complexity: ? , Space Complexity: ?

The Principal of Hash and its application

Pengju Ren 2026@YJTU IAIR

Python Dictionaries

A dictionary is a collection which is unordered, changeable and indexed

Key *Value*

myDict = {"James": "GOAT of NBA",
 "Curry" : "Best 3-point Shooter"}

myArray = ["James", "Curry"]

myArray[1]

Curry

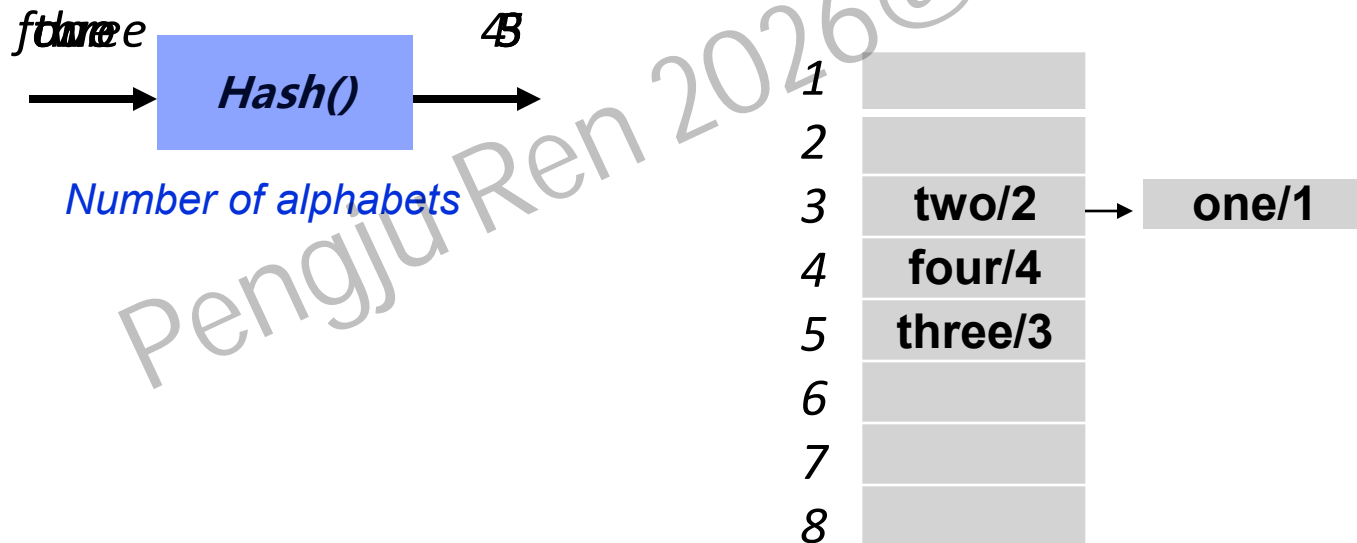
myDict['Curry']

Best 3-point Shooter

Hash table (Dictionary in Memory)

A *hash table* is a way of doing *Key-value lookups*, you can store the *values* in an array, and then use a *hash function* to find the index of the array cell that corresponds to the *key-value pair*.

Eng2Digit = ['four':4, 'three':3, 'two':2, 'one':1]



Bad hash() function, too many conflicts !!!

What is Hashing

Hashing is a method of allocating and indexing data. The idea behind hashing is to allow large amounts of data to be indexed using keys commonly created by formulas (hashing function)

Practical Use of Hashing

➤ Password verification

When you log into a website, how does the system confirm that your password is correct without storing your plaintext password?

➤ File system: File path is mapped to physical location on disk

When the number of files grows from dozens to billions, how do you quickly locate the corresponding file?

Hashing Terminology

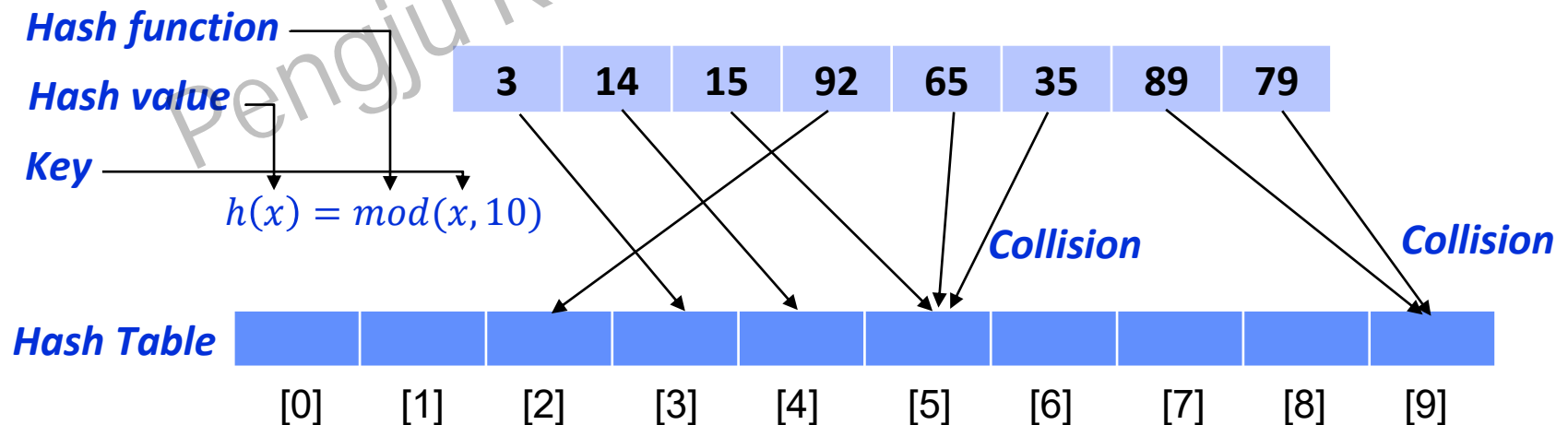
Hash function: It is a function that can be used to map of arbitrary size (n) to data of fixed size (m)

Key: Input data by user

Hash value: A value that is returned by Hash Function

Hash Table: It is a data structure which implements an associative array abstract data type, a structure that can map keys to value

Collision: A collision occurs when two different keys to a hash function produce the same output(hash value)



Typical Hash Functions

1. Direct Addressing Method

$$\text{Hash}(\text{key}) = a * \text{key} + b \text{ (a, b are constants, } a \neq 0)$$

2. Modulo Method

$$\text{Hash}(\text{key}) = \text{key} \bmod m, \text{ m is often chosen as a prime number}$$

3. Universal hash function

$$\text{Hash}_{a,b}(\text{key}) = ((a * \text{key} + b) \bmod p) \bmod m$$

$$\Pr\{h(k_i) = h(k_j)\} = 1/m, \forall k_i \neq k_j \in \{0,1, \dots, n\}$$

Good Hash function: computes quickly, distributes keys uniformly, and minimizes collisions

Example: Hash for Password verification

Core Mechanism: One-Way Property and Avalanche Effect

Registration Phase: User enters PW \rightarrow *fingerprint* = Hash(PW) \rightarrow send *fingerprint* to the website's database

Login: User enters PW \rightarrow hash value = Hash(PW) \rightarrow Compare with stored *fingerprint* at the website side

One-way (irreversible): You cannot reconstruct the original meat(*password*) from the mince(*hash value*).

Avalanche Effect: If you change just one letter in the original password, the resulting hash value will change drastically.

Example: Hash for File Systems location

Core Mechanism: Uniform Distribution & Accelerated Access

Taking a file's unique identifier (like its filename), compute its hash value (e.g., `77e1ba46ee3a2b2d1558d7c5d07c4c0caa46c7bf`), and then use the first few characters of this hash as names for multi-level subdirectories

For example, the path `/77/e1/77e1ba46...` works as follows:

1. Take `77` of the hash as the name of the first-level subdirectory.
2. Take `e1` as the name of the second-level subdirectory.
3. Use the complete hash value as the filename, stored under the `/77/e1/` directory

Collision Resolution Techniques

Direct Chaining

- Hash table never gets full
- Huge linked list might causes performance leaks (Time complexity for search operation becomes $O(n)$.)

Open addressing

- Easy implementation
- **Linear probing:** place new key into closest following empty cell
- **Double hashing:** Interval between probes is computed by another hash function
- **When Hash table is full,** Create **2X** size of current Hash Table and recall hashing for current keys, creation of new Hash table affects performance (Time complexity for search operation becomes $O(n)$.)

If deletion operations are frequent, we prefer this

If the input size is known, we prefer this

Algorithm complexity analysis

Pengju Ren 2026@XJTU IAIR

What makes a good Algorithm?

Correctness & Efficiency (both in time and space)

$O(\cdot)$ 、 $\Omega(\cdot)$ 、 $\Theta(\cdot)$ are used to describe the efficiency of algorithms

Time complexity: A way of showing how the runtime of a function increases as the size of the input increases.

The function $f(n)=O(g(n))$, iff $\exists c$ and n_0 , such that $f(n)\leq cg(n)$, $\forall n \geq n_0$

The function $f(n)=\Omega(g(n))$, iff $\exists c$ and n_0 , such that $f(n)\geq cg(n)$, $\forall n \geq n_0$

The function $f(n)=\Theta(g(n))$, iff $\exists c_1, c_2$ and n_0 , such that $c_1g(n)\geq f(n)\geq c_2g(n)$, $\forall n \geq n_0$

What makes a good Algorithm?

Correctness & Efficiency (both in time and space)

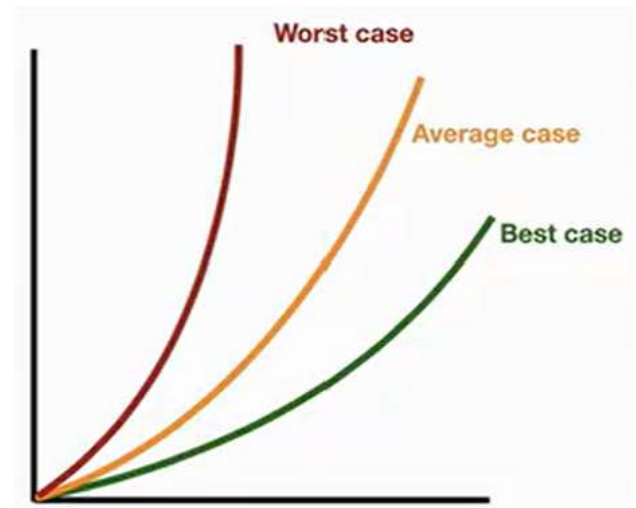
$O(\cdot)$ 、 $\Omega(\cdot)$ 、 $\Theta(\cdot)$ are used to describe the efficiency of algorithms

Time complexity: A way of showing how the runtime of a function increases as the size of the input increases.

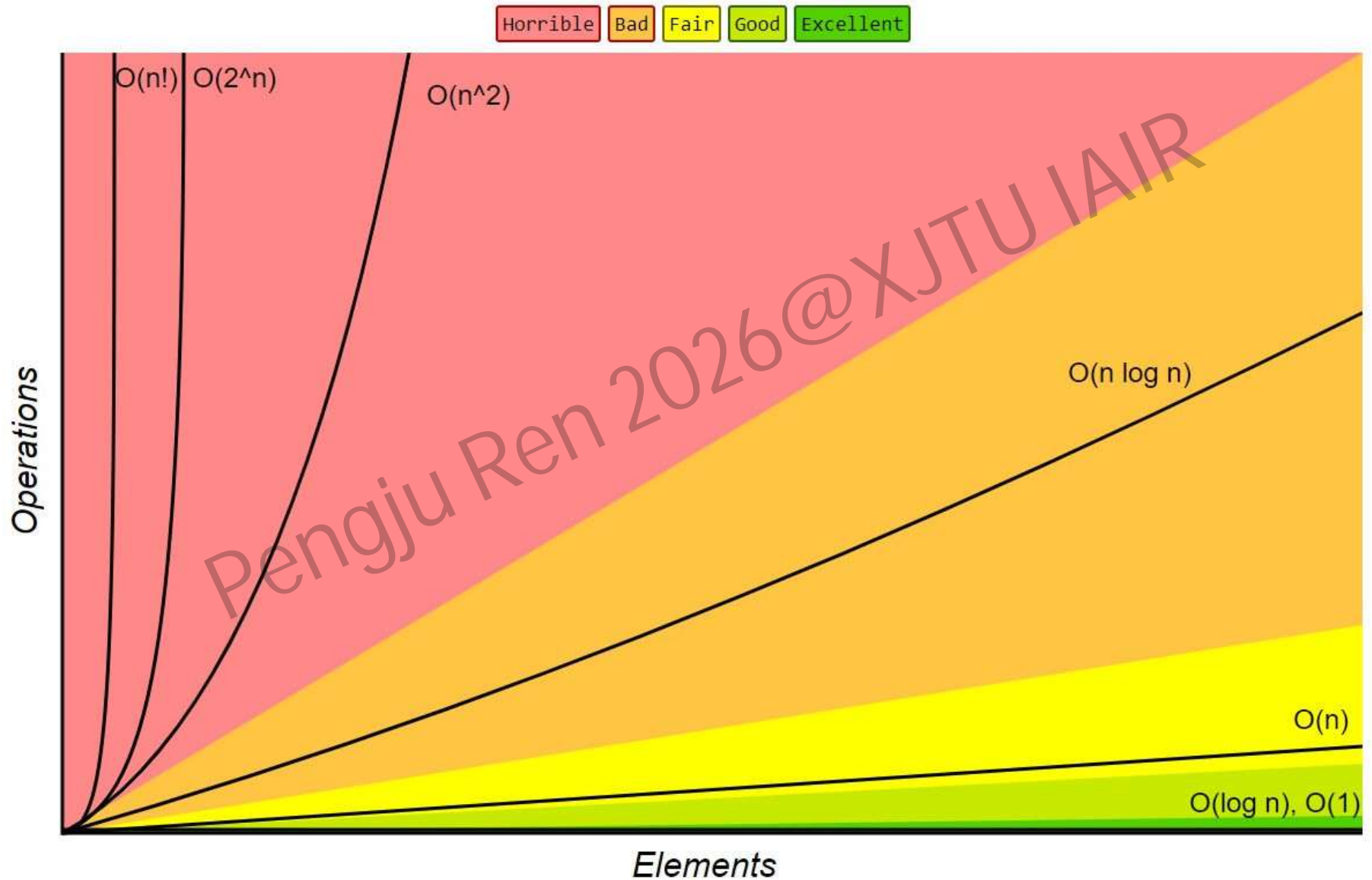
$O(\cdot)$: It is a complexity that is going to be less or equal to the worst case (Upper bound)

$\Omega(\cdot)$: It is a complexity that is going to be at least more than the best case (Lower bound)

$\Theta(\cdot)$: It is a complexity that is within bounds of the worst and the best case (Average bound)



Big-O Complexity Chart



Runtime Complexities (Non Dominant Terms)

Complexity	Name	Sample
$O(1)$	Constant	Excellent/Best Efficient. Independent on the input size (A simple add numbers function)
$O(\log n)$	Logarithmic	Good Every operation the problem size is reduced by half (Find an element in sorted array)
$O(n)$	Linear	Fair Go through numbers from 1 to n
$O(n \log n)$	N-log-n	Not Bad Sorting
$O(n^2)$	Quadratic	Bad Nested Loops
$O(n^3)$	Cubic	Bad Deeper nested Loops
$O(2^n)$	Exponential	Too Bad/Horrible Sub-problem in Fibonacci
$O(n!)$	Factorial	

```
def multi_numbers(n):
    return n*n
```

```
def print_items(n):
    for i in range(n):
        print i
```

```
def print_items(n):
    for i in range(n):
        for j in range(n):
            print (i,j)
```

```
def recursiveFib(n):
    if (n<2): return n;
    return recursiveFib(n-1)+recursiveFib(n-2)

print(recursiveFib(10))
```

Time Complexity: ?

Runtime Complexities (Non Dominant Terms)

Complexity	Name	Sample
$O(1)$	Constant	Calculation is not dependent on the input size (A simple add numbers function)
$O(\log n)$	Logarithmic	For each iteration the problem size is reduced by half (Find an element in sorted array)
$O(n)$	Linear	Loop through numbers from 1 to n
$O(n \log n)$	N-log-n	Sorting
$O(n^2)$	Quadratic	Nested Loops
$O(n^3)$	Cubic	Deeper nested Loops
$O(2^n)$	Exponential	Double recursion in Fibonacci
$O(n!)$	Factorial	

```
def print_items(a,b):  
    for i in range(a):  
        print i  
    for j in range(b):  
        print j
```

-If your algorithm is in the form “do this, then when you are all done, do that” then you **add** the runtimes

```
def print_items(a,b):  
    for i in range(a):  
        for j in range(b):  
            print (i,j)
```

-If your algorithm is in the form “do this for each time you do that” then you **multiply** the runtimes

Time Complexity: ?

Runtime Complexities (Non Dominant Terms)

Complexity	Name	Sample
$O(1)$	Constant	Calculation is not dependent on the input size (A simple add numbers function)
$O(\log n)$	Logarithmic	For each iteration the problem size is reduced by half (Find an element in sorted array)
$O(n)$	Linear	Loop through numbers from 1 to n
$O(n \log n)$	N-log-n	Sorting
$O(n^2)$	Quadratic	Nested Loops
$O(n^3)$	Cubic	Deeper nested Loops
$O(2^n)$	Exponential	Double recursion in Fibonacci
$O(n!)$	Factorial	

```
def findBiggestNumber(sampleArray):
    biggestNumber = sampleArray[0]
    for index in range(1, len(sampleArray)):
        if sampleArray[index] > biggestNumber:
            biggestNumber = sampleArray[index]
    print(biggestNumber)
```

Time Complexity: ?

Runtime Complexity (Quiz)

```
for(i=0; i<n; i+2){  
    Doing Something // O(1)  $O\left(\frac{n}{2}\right) = O(n)$   
}
```

```
for(i=0; i<n; i+k){  
    Doing Something // O(1)  $O\left(\frac{n}{k}\right) = O(n)$   
}
```

```
for(i=0; i<n; i++){  
    for(j=0; j<i; j++){  
        Doing Something // O(1)  
    }  
}  $\frac{n(n+1)}{2} \Rightarrow O(n^2)$ 
```

```
p=0  
for(i=0; p<n; i++){  
    p = p+i;  
}
```

$$\frac{k(k+1)}{2} \geq n \Rightarrow O(\sqrt{n})$$

```
p=0  
for(i=0; p<n; i++){  
    p = p+ixi; Doing Something // O(1)  
}
```

$$\frac{k(k+1)(2k+1)}{6} \geq n \Rightarrow O(\sqrt[3]{n})$$

```
p=0  
for(i=0; i<n; i=i*2){  
    p++;  
    for(j=0; j<p; j=j*2){  
        Doing Something // O(1)  
    }  
}
```

```
for(i=1; i<n; i=i*2){  
    Doing Something // O(1)  
}  $O(\log_2 n) = O(\log n)$ 
```

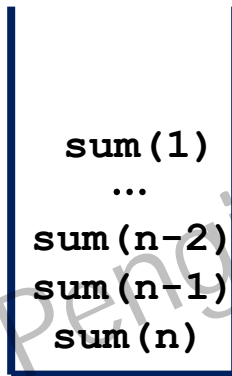
```
for(i=1; i<n; i=i*k){  
    Doing Something // O(1)  
}  $O(\log_k n) = O(\log n)$ 
```

```
for(i=0; i<n; i++){  
    for(j=1; j<n; j=j*2){  
        Doing Something // O(1)  
    }  
}  $O(n \log n)$ 
```

Space Complexity

How many bytes consumed during execution ?

```
def sum(n):  
    if n <=0:  
        return 0  
    return n+sum(n-1)
```



Stack

Space Complexity: $O(n)$

```
def pair_sum_sequence(n):  
    total = 0  
    for i in range(n):  
        total = total + pair_sum(i, i+1)  
    return total  
  
def pair_sum(a,b):  
    return a+b
```

Space Complexity: ?

Example of Algorithms (diff implementation diff complexity)

- Three-way Set Disjointness

```
def disjoint1(A, B, C):  
    """Return True if there is no element common to all three lists."""  
    for a in A:  
        for b in B:  
            for c in C:  
                if a == b == c: return False # we found a common value  
    return True # if we reach this, sets are disjoint
```

Time Complexity: $O(n^3)$

```
def disjoint2(A, B, C):  
    """Return True if there is no element common to all three lists."""  
    for a in A:  
        for b in B:  
            if a == b: # only check C if we found match from A and B  
                for c in C:  
                    if a == c: # (and thus a == b == c)  
                        return False # we found a common value  
    return True # if we reach this, sets are disjoint
```

Time Complexity: $O(n^2)$

At the worst case, disjoint2() is with same complexity as disjoint1()

Example of Algorithms (diff implementation diff complexity)

- Element Uniqueness

```
def unique1(S):  
    """Return True if there are no duplicate elements in sequence S."""  
    for j in range(len(S)):  
        for k in range(j+1, len(S)):  
            if S[j] == S[k]: return False # found duplicate pair  
    return True # if we reach this, elements were
```

Time Complexity: $O(n^2)$

```
def unique2(S):  
    """Return True if there are no duplicate elements in sequence S."""  
    temp = sorted(S) # create a sorted copy of S  
    for j in range(1, len(temp)):  
        if S[j-1] == S[j]: return False # found duplicate pair  
    return True # if we reach this, elements were
```

Time Complexity: $O(n \log n)$

The Motivation of Hashing

It is time efficient in case of **SEARCH** operation

Data Structure	Search	Insertion	Deletion
Array (Python List)	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(1)$	$O(1)$
Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hashing	$O(1)/O(n)$	$O(1)/O(n)$	$O(1)/O(n)$

- On average Insertion/Deletion/Search operations take **$O(1)$** time
- When hash function is not good enough, they takes **$O(n)$** time

Sorting Algorithms

Pengju Ren 2026@XJTU IAIR

What is Sorting ?

By definition sorting refers to arranging data in a particular format: either *ascending* or *descending*

- Bubble Sorting
- Selection Sorting
- Insertion Sorting
- Bucket Sorting
- Merge Sorting
- Quick Sorting
- Heap Sorting

Categories of Sorting ?

Space used

In place sorting: Sorting algorithms which does not require any extra space for sorting (e.g. *Bubble Sorting*)

Out place sorting: Sorting algorithms which requires an extra space for sorting (e.g. *Merge Sorting*)

Stability

Stable sorting: contents does not change the sequence of similar content in which they appear (e.g. *Insertion Sorting*)

Unstable sorting: contents might change the sequence of similar content in which they appear (e.g. *Quick Sorting*)

Stability

Unsorted data	
Name	Age
David	8
Green	8
Emma	7
Fendy	7
Bob	7
Candy	8
Helen	8
Andy	8



Sorted by Name	
Name	Age
Andy	8
Bob	7
Candy	8
David	8
Emma	7
Fendy	7
Green	8
Helen	8



Sorted by Age	
Name	Age
Bob	7
Emma	7
Fendy	7
Andy	8
Candy	8
David	8
Green	8
Helen	8

Stable sorting

Sorted by Age	
Name	Age
Emma	7
Bob	7
Fendy	7
David	8
Andy	8
Green	8
Candy	8
Helen	8

Unstable sorting

Bubble Soring

Repeatedly compare each pair of adjacent items and swap them if they are in the wrong order

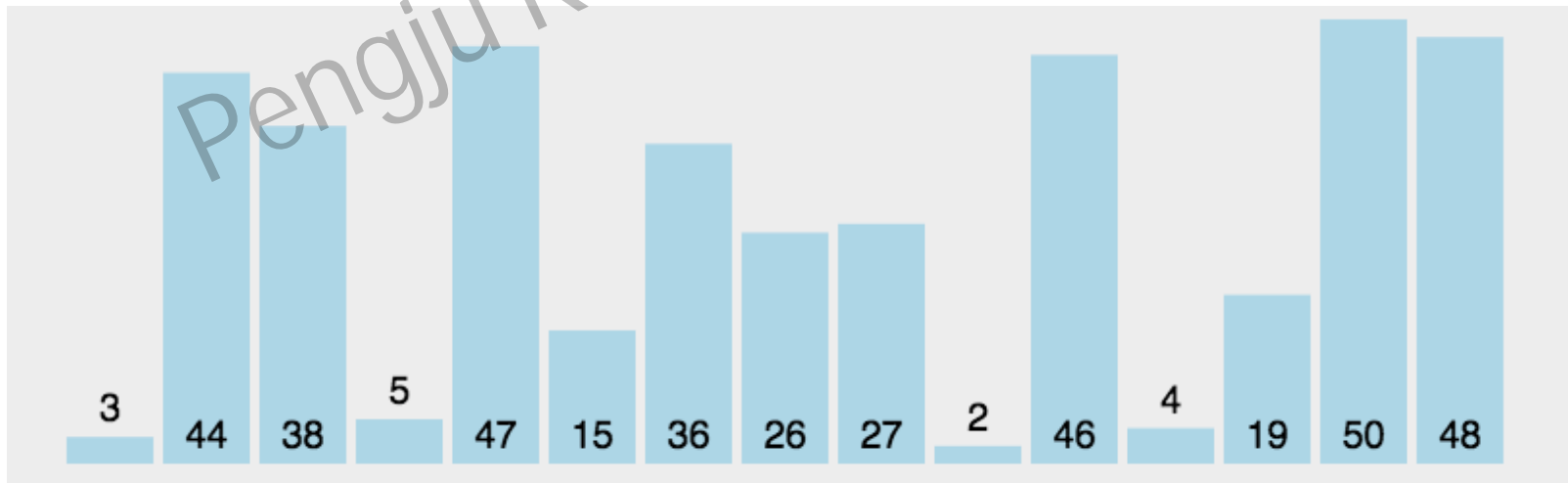
```
def bubble_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        for j in range(0, n-i-1):  
            if arr[j] > arr[j+1]:  
                arr[j], arr[j+1] = arr[j+1], arr[j]  
    return arr
```



Selection Soring

Repeatedly find the minimum element and move it to the sorted part of array to make unsorted part sorted

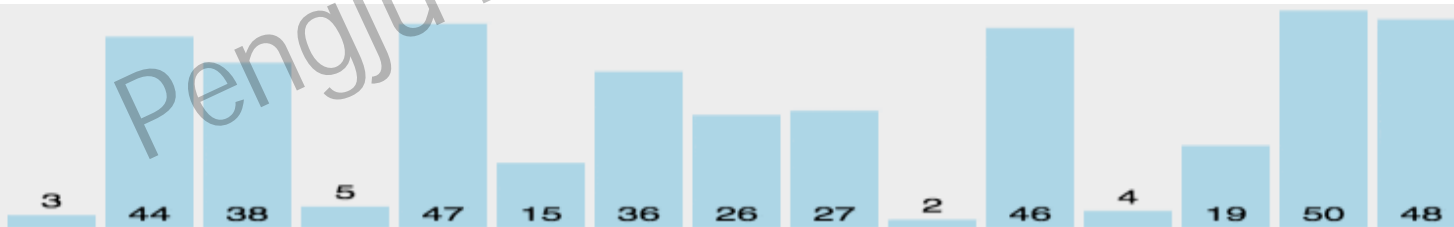
```
def selection_sort(arr):  
    n = len(arr)  
    for i in range(n):  
        min_idx = i  
        for j in range(i+1, n):  
            if arr[j] < arr[min_idx]:  
                min_idx = j  
        arr[i], arr[min_idx] = arr[min_idx], arr[i]  
    return arr
```



Insert Soring

Divide the array into two part, taking first element from unsorted array and find its correct position in sorted array, repeat until unsorted array is empty

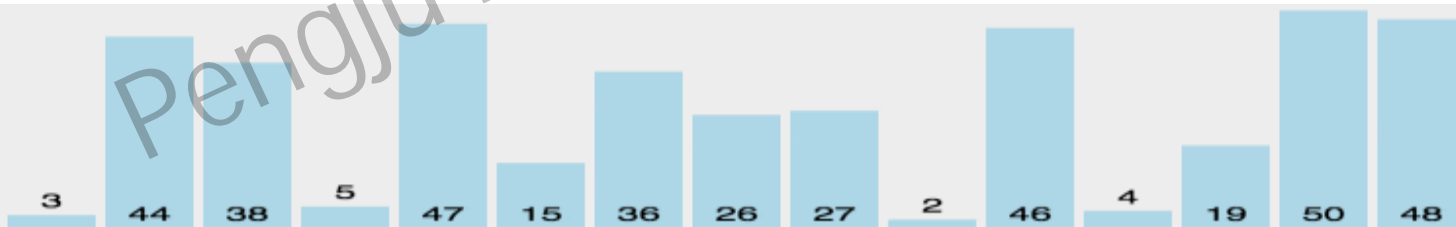
```
def insertion_sort(arr):  
    n = len(arr)  
    for i in range(1, n):  
        key = arr[i]  
        j = i-1  
        while j >= 0 and key < arr[j]:  
            arr[j+1] = arr[j]  
            j -= 1  
        arr[j+1] = key  
    return arr
```



Insert Soring

Divide the array into two part, taking first element from unsorted array and find its correct position in sorted array, repeat until unsorted array is empty

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



Bucket Sorting

Create buckets and distribute elements of array into buckets, sort buckets individually, then merge buckets after sorting

```
def bucket_sort(arr, bucket_size=4):  
    if len(arr) == 0:  
        return arr  
    min_val, max_val = min(arr), max(arr)  
  
    bucket_count = (max_val - min_val) // bucket_size  
    buckets = [[] for _ in range(bucket_count)]  
  
    for num in arr:  
        idx = (num - min_val) // bucket_size  
        buckets[idx].append(num)  
  
    sorted_arr = []  
    for bucket in buckets:  
        insertion_sort(bucket)  
        sorted_arr.extend(bucket)  
  
    return sorted_arr
```



Lets say, that this is the array of numbers that needs sorting

Merger Sorting

Divide the array in two halves and keep halving recursively until they become one element, merge halves by sorting them

```
def merge_sort(arr):  
    if len(arr) <= 1:  
        return arr  
  
    mid = len(arr) // 2  
    left = merge_sort(arr[:mid])  
    right = merge_sort(arr[mid:])  
  
    return merge(left, right)
```

```
def merge(left, right):  
    result = []  
    i = j = 0  
    while i < len(left) and j < len(right):  
        if left[i] <= right[j]:  
            result.append(left[i])  
            i += 1  
        else:  
            result.append(right[j])  
            j += 1  
    result.extend(left[i:])  
    result.extend(right[j:])  
    return result
```



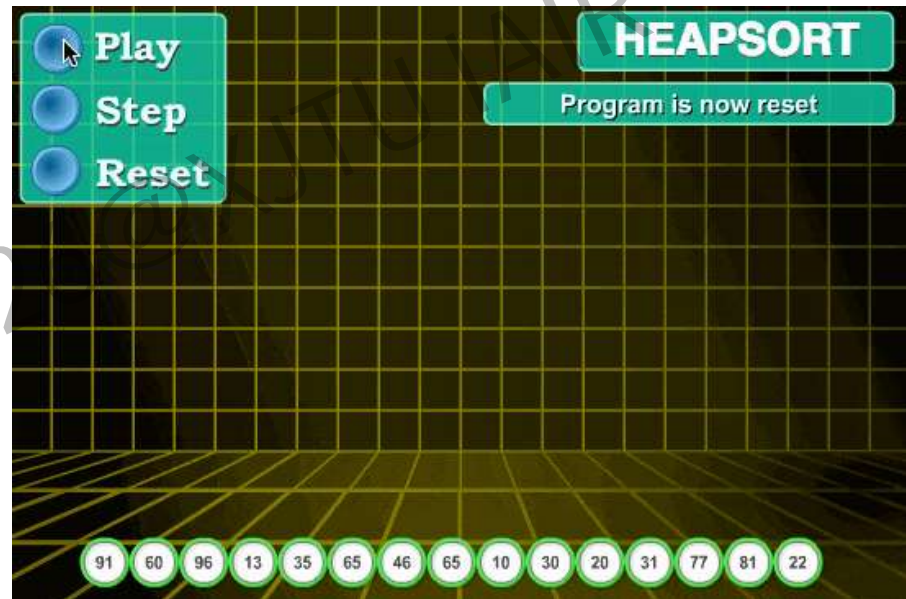
Quick Sorting

Selecting a *pivot element* from the array and partitioning the other elements into two sub-arrays, according to whether they are $<$ or $>$ than the pivot. Then sub-arrays are sorted recursively

```
def quick_sort(arr):  
    if len(arr) <= 1:  
        return arr  
    pivot = arr[len(arr)//2]  
    left = [x for x in arr if x < pivot]  
    middle = [x for x in arr if x == pivot]  
    right = [x for x in arr if x > pivot]  
    return quick_sort(left) + middle + quick_sort(right)
```

Heap Sorting (Suited for array, not linked-list)

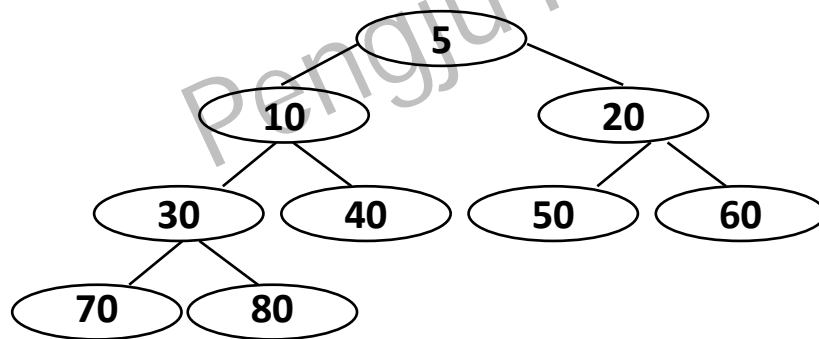
1. *Build_max_heap(A)* from unorder array
2. Find Maximum element $A[1]$
3. Swap element $A[n]$ and $A[1]$ (the max element is at the end of the array)
4. Discard node n from heap (heap size is decreased)
5. New root may violate max heap property, but its children are max heaps, *max_heapify(A)* to fix this.



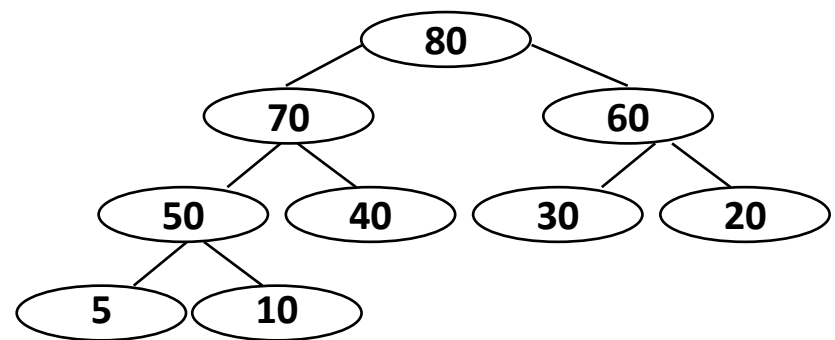
What is a Binary Heap

A **Binary Heap** is a *Binary Tree* with following properties.

- A Binary Heap is either Min Heap or Max Heap. In a Min/Max Binary Heap, the key at root must be minimum/maximum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary tree
- It's complete tree (all levels are completely filled except possibly the last level and the least level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array



Min Heap

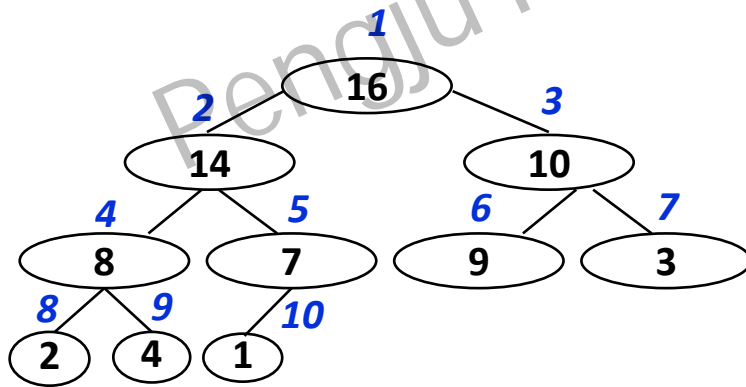


Max Heap

Heap as a Tree

- An array, visualized as a nearly complete *binary tree*
- *root* of tree: first element in the array (index $i=1$)
- $parents(i) = i/2$: return the index of node's *parent*
- $left(i) = i*2$: return the index of node's *left child*
- $right(i) = i*2+1$: return the index of the node's *right child*

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



No pointers required!
Height of a binary tree is $\log n$

Common Ops on Binary Heap

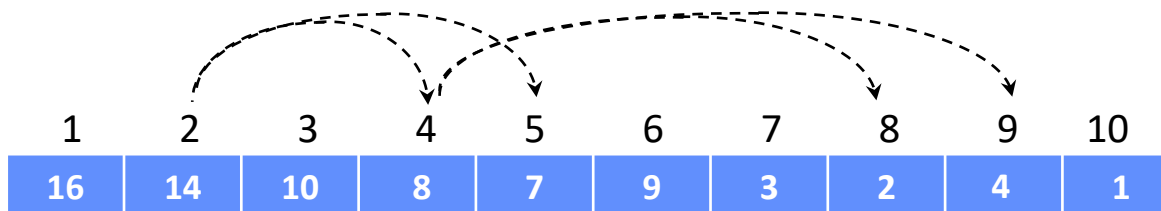
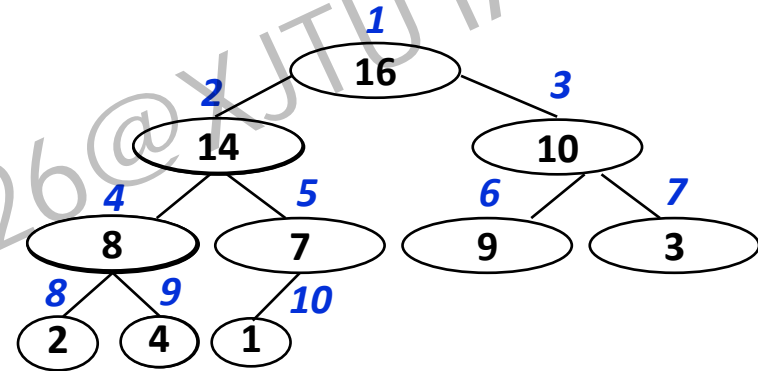
Bulid_max_heap: produce a max-heap from an unordered array

max_heapify: correct a **single** violation of the heap property in a subtree at its root

Traversal of Binary Heap

Insert value in Binary Heap

Extract Min/Max



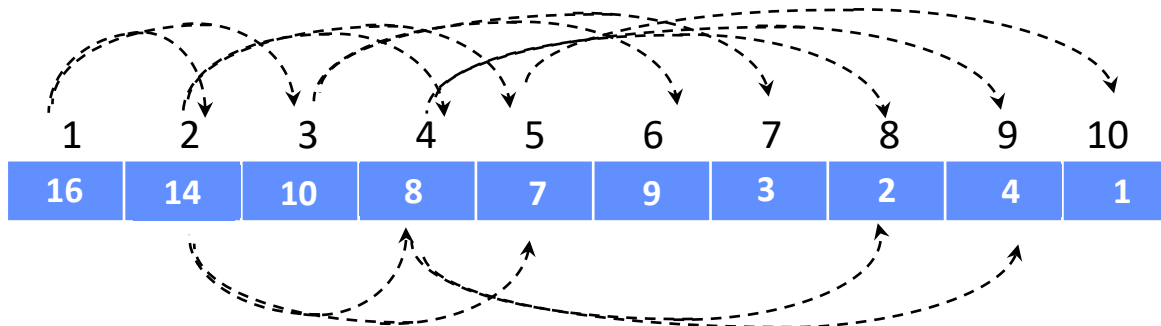
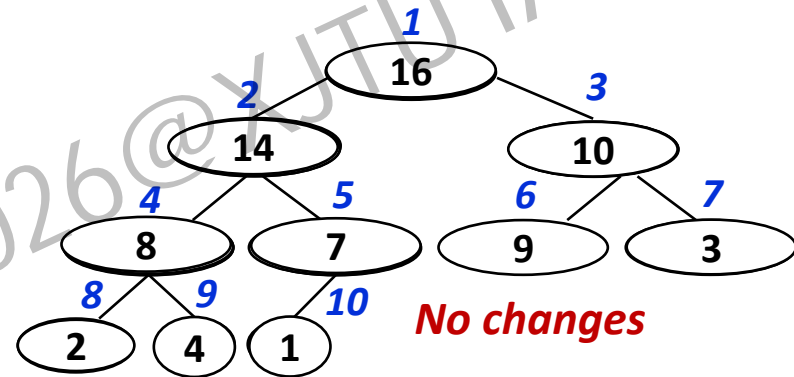
Common Ops on Binary Heap

Build_max_heap: produce a max-heap from an *unordered* array

Build_max_heap(A):
for $i = n/2$ downto 1
do max_heapify

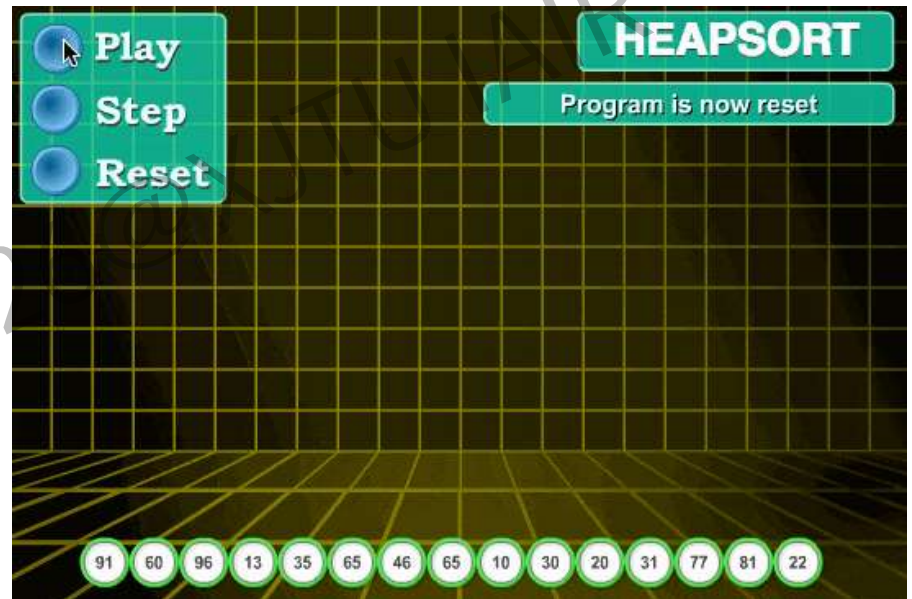
Why start at $n/2$?

Because elements $A[n/2+1 \dots n]$
are all leaves of the tree



Recap: Heap Sorting

1. *Build_max_heap(A)* from unordered array
2. Find Maximum element $A[1]$
3. Swap element $A[n]$ and $A[1]$ (the max element is at the end of the array)
4. Discard node n from heap (heap size is decreased)
5. New root may violate max heap property, but its children are max heaps, *max_heapify(A)* to fix this.



After n iteration the heap is empty, every iteration involves a swap and a *max_heapify* operation $O(\log n)$, Overall is $O(n \log n)$

More about Heap Sorting

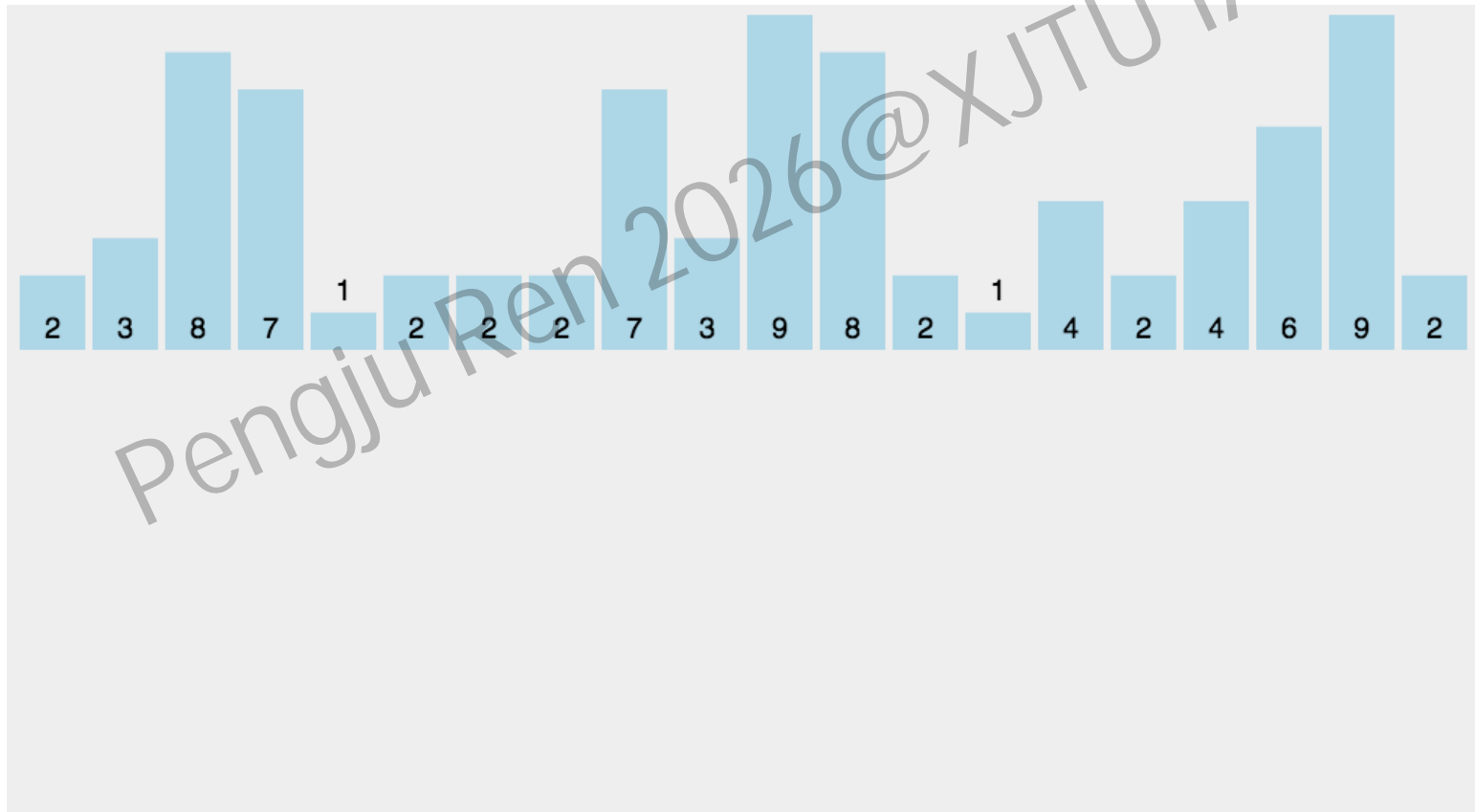
- **Top-K** sorting is achieved by maintaining a fixed-size min-heap (**v**) or max-heap () ?
- It is particularly effective for sorting *streaming data*:

```
def top_k(stream, k):  
    min_heap = []  
    for x in stream:  
        if len(min_heap) < k: #堆未满, 直接插入堆中;  
            heapq.heappush(min_heap, x)  
        else:  
            if x > min_heap[0]: #如果x比目前top-k中最小的元素大, 则替换之前堆顶元素  
                Bulid_min_heap([x, min_heap[1:]) # 弹出堆顶并插入 x  
    return min_heap
```

Count Sorting

Step1: Find the maximum and minimum elements in the array to be sorted;

Step2: Count the occurrences of each element with value i in the array and store the counts in the i -th element of array C



Radix Sorting

- **Step1: Determine the Maximum Digits:** Identify the maximum number of digits in the largest number in the list.
- **Sort by Each Digit:** Starting from the least significant digit, sort all numbers using a stable sorting algorithm like Counting Sort.
- **Repeat for Higher Digits:** Continue sorting for each digit until all place values are processed.

	7	10	5	6	15	11	2	9	14	4	0	13	3	12	1	8
8		■			■	■		■	■			■		■		■
4	■		■	■	■				■	■		■		■		
2	■	■		■	■	■	■		■				■			
1	■		■		■	■		■				■	■		■	

Sorting Algorithms

Name	Best Case	Time Complexity	Worst Case	Space Complexity	Stable
Bubble	$O(n)$	$O(n^2)$		$O(1)$	Y
Selection		$O(n^2)$		$O(1)$	N
Insertion	$O(n)$	$O(n^2)$		$O(1)$	Y
Bucket	$O(n)$	$O(n \log n)$		$O(1)$	Y
Merge		$O(n \log n)$		$O(n)$	Y
Quick		$O(n \log n)$	$O(n^2)$	$O(n)$	N
Heap		$O(n \log n)$		$O(1)$	N
Count		$O(n)$		$O(n)$	Y
Radix		$O(n)$		$O(n)$	Y

Summary

- **Hashing:** It is a function that can be used to map of arbitrary size to data of fixed size
- **Algorithm Complexity :** Running time and Space usage are used to analysis the efficiency of particular algorithm (**Time/Space Complexity**)
- **Sorting:** Master various sorting algorithms and perform comparisons from different perspectives.

Homework 2

1: Given a linked list, sort the list in ascending order and return the sorted list.

– Example:

Input list: 4 -> 2 -> 1 -> 3 -> 0 -> -1

Sorted list: -1 -> 0 -> 1 -> 2 -> 3 -> 4

• **Test data:**

– Small-scale: Randomly generate a linked list with 100 nodes

– Large-scale: Randomly generate a linked list with 10,000 nodes

Pengju Ren 2026@XJTU IAIR

Homework 2

2: Which sorting algorithms support parallel acceleration?

From the fundamental thinking behind algorithm design, whether a sorting algorithm can be accelerated in parallel depends on whether *its computational task can be broken down into multiple independent subtasks*. If the subtasks are highly dependent on each other, requiring the previous step to complete before the next can proceed, then parallelization is difficult; conversely, if the data can be partitioned, or the comparison operations themselves are independent, then parallelization is very suitable.

- **In exploring parallel sorting algorithms, what other algorithms did you learn that weren't covered in the course? (e.g, *bitonic sort, Odd-even sort* ?)**

Submission requirements:

- 1: Description of the implementation approach
- 2: Performance test results and analysis
- 3: Difficulties encountered and solutions

Homework naming: HW2-Class X-stu ID-name, e.g. HW2-Class01-12345678-张三

Submission email: chengyuma@stu.xjtu.edu.cn

Submission time: 3.22 24:00