CSCI2100B Data Structures

Tutorial 07
Insertion Sort, Application and Analysis of Sorting Algorithms

LI Muzhi Raynor 李木之
mzli@cse.cuhk.edu.hk

Slides adapted from Professor Leung Ho-Fung’s Lecture notes, Mr. Chen Yixin’s Tutorial notes
Icons made by https://freepik.com

Insertion Sort (插入排序) – Pass 1 & 2

- Pass #1

Compare

A[0]	A[1]	A[2]	A[3]	A[4]
5	3	2	4	8

Swap

A[0]	A[1]	A[2]	A[3]	A[4]
3	5	2	4	8

- Pass #2

Compare

A[0]	A[1]	A[2]	A[3]	A[4]
3	5	2	4	8

Swap

A[0]	A[1]	A[2]	A[3]	A[4]
3	2	5	4	8

Compare

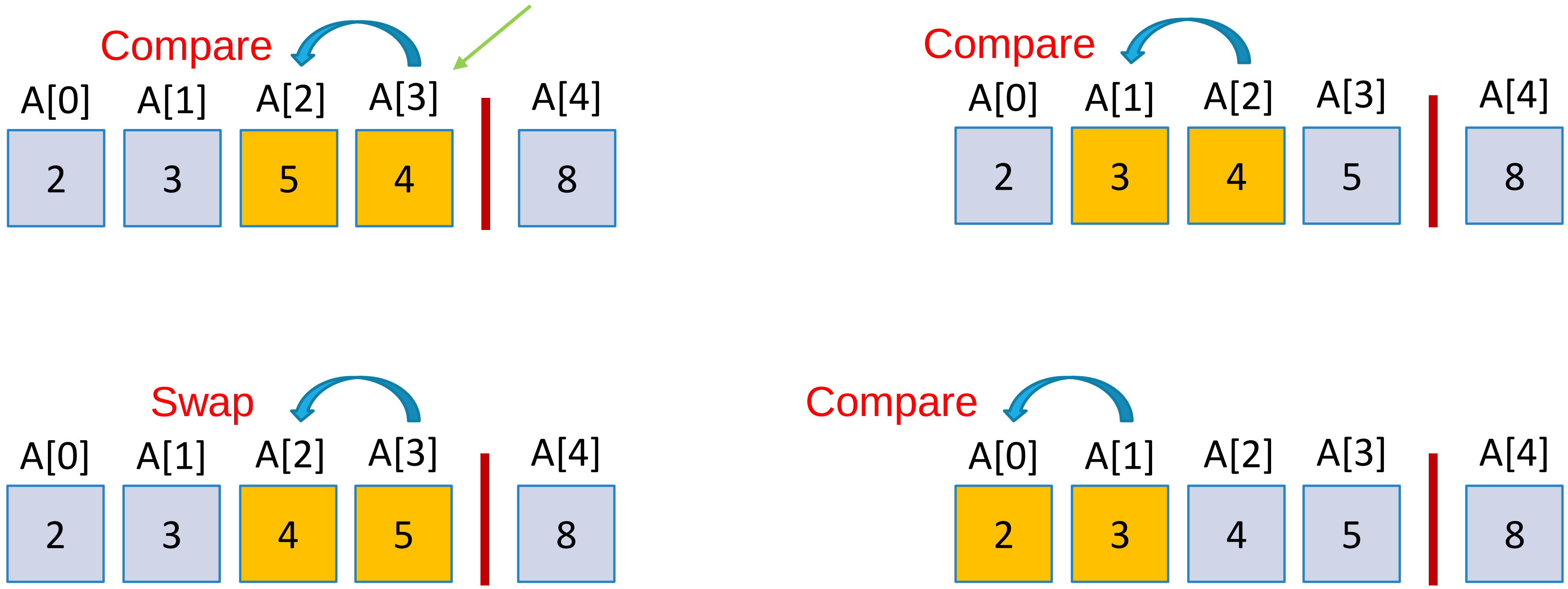
A[0]	A[1]	A[2]	A[3]	A[4]
3	2	5	4	8

Swap

A[0]	A[1]	A[2]	A[3]	A[4]
2	3	5	4	8

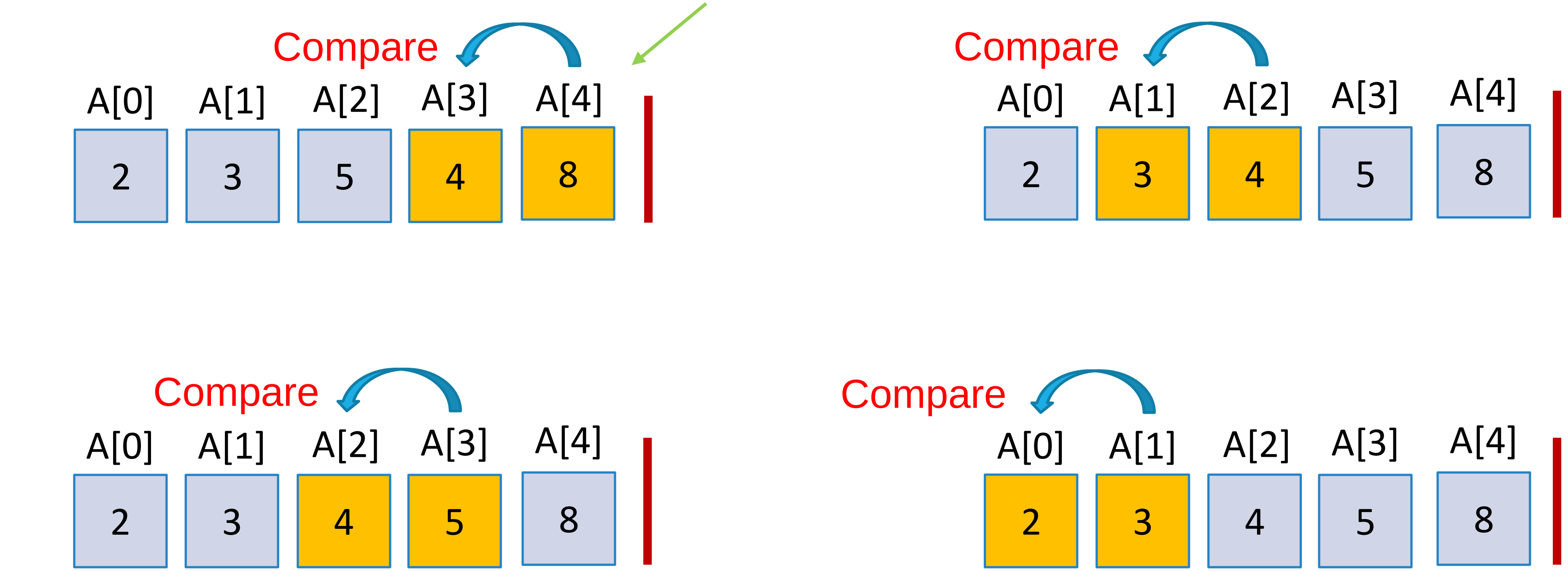
Insertion Sort (插入排序) – Pass 3

- Pass #3



Insertion Sort (插入排序) – Pass 4

- Pass #4



Insertion Sort – Observation

	A[0]	A[1]	A[2]	A[3]	A[4]
Initial Array	5	3	2	4	8
After Pass 1	3	5	2	4	8
After Pass 2	2	3	5	4	8
After Pass 3	2	3	4	5	8
After Pass 4	2	3	4	5	8

- After iteration i, the first $\underline{i+1}$ elements of the array are sorted.
- Therefore, use insertion sort to sort an array of 5 elements, $\underline{\ 4\ }$ iterations are required.
- Use insertion sort to sort an array of n elements, $\underline{n-1}$ passes are required.

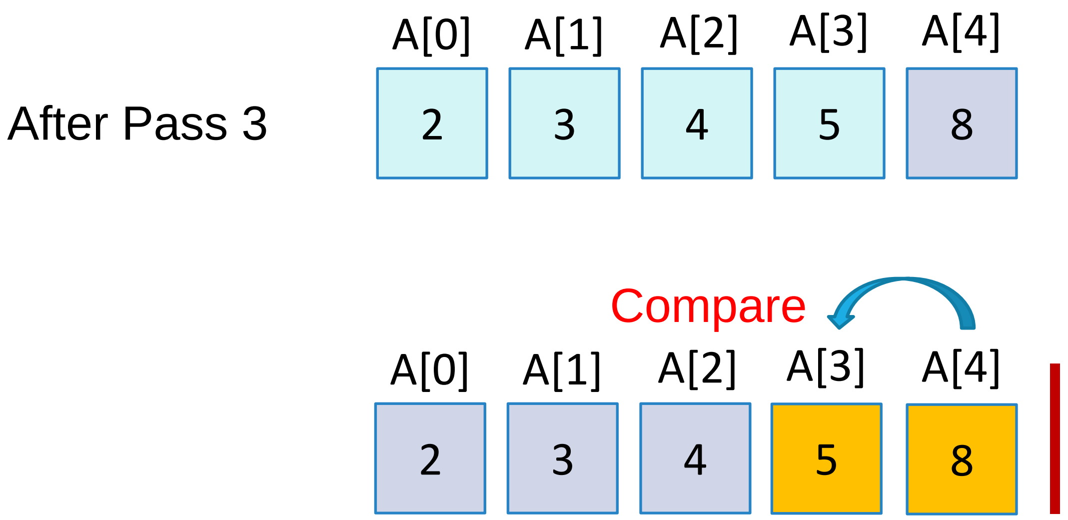
Insertion Sort Ver. 1.0 – Implementation

```
void insertion_sort(int arr[], int len) {
    for (int i = 1; i < len; i++) {
        for (int j = i; j > 0; j--) {
            if (arr[j] < arr[j-1]){
                swap(&arr[j], &arr[j-1]);
            }
        }
    }
}
```

```
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

Insertion Sort – Early Stop (for each pass)

- Pass #4



- At pass 4, the first 4 elements are sorted, then the 4^{th} element is the largest element among first 4 elements.
- As the 5^{th} element $8 > 5$, we know that the 5^{th} element is larger than the largest element of first 4 elements. Then we can stop this pass and enter the next one.
- More general, if one single comparison shows that no swapping is required, one can stop the current pass and enter the next one. Why?

Insertion Sort Ver. 1.1 – Implementation

```
void insertion_sort(int arr[], int len) {
   for (int i = 1; i < len; i++) {
      for (int j = i; j > 0 && arr[j] < arr[j-1]; j--) {
         swap(&arr[j], &arr[j-1]);
      }
   }
}
```

- At Pass i, the first $\underline{\ i\ }$ elements are sorted, then the $\underline{i^{th}}$ element is the largest element among first $\underline{\ i\ }$ elements.
- if one single comparison shows that no swapping is required, one can stop the current pass and enter the next one.

Insertion Sort – Analysis

```
void insertion_sort(int arr[], int len) {
    for (int i = 1; i < len; i++) {
        for (int j = i; j > 0 && arr[j] < arr[j-1]; j--) {
            swap(&arr[j], &arr[j-1]);
        }
    }
}
```

- At pass #1 ($i = 1$), we perform at most ___1___ comparisons.
- At pass #2 ($i = 2$), we perform at most ___2___ comparisons.
- At pass #m ($i = m$), we perform ___m or i___ item comparisons
- Therefore, the total number of running time at the worst case to sort a n element array is roughly proportional to ___$1 + 2 + \cdots + (n - 1) = \frac{n^2 - n}{2}$ or $O(n^2)$___

Interview Question

- At the worst case, which of the following sorting algorithm has the minimum time complexity?

  A. Bubble Sort   B. Quick Sort

  C. Insertion Sort   D. Merge Sort

Analysis of Sorting Algorithms

- Consider sorting algorithms based on compare and swapping only,
  e.g. Selection Sort, Bubble Sort, and Insertion Sort
  the worst time-complexity cannot below $\underline{\quad O(n^2) \quad}$.

- Consider sorting algorithms based on divide and conquer,
  e.g. Merge Sort and Quick Sort,
  the worst time complexity cannot below $\underline{O(n \log n)\ }$.

Programming Problem – Majority Element

- Given an array $nums$ of size n, return the majority Element.
- The majority element is the element that appears strictly more than $\lfloor n/2 \rfloor$ times.

  You may assume that the majority element always exists in the array.

Input: $nums = [3, 2, 3]$
Output: 3

Example 1

Input: $nums = [2, 2, 1, 1, 1, 2, 2]$
Output: 2

Example 2

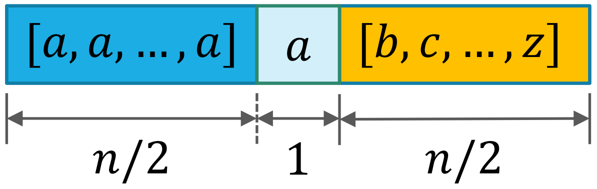
```
int majorityElement(int* nums, int numsSize) {
    // Your Task: finish this function (You can write other auxiliary functions)

}
```

Hint: Sorting Algorithm and / or Divide and Conquer

Solution #1: Use Sorting Algorithm

Claim: The median of the sorted array is the majority element of the original array

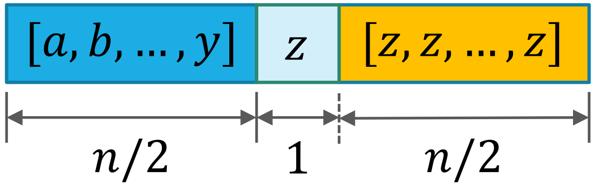
- Case 1a: n is odd



Note that $^5/_2 = 2$ for integer division

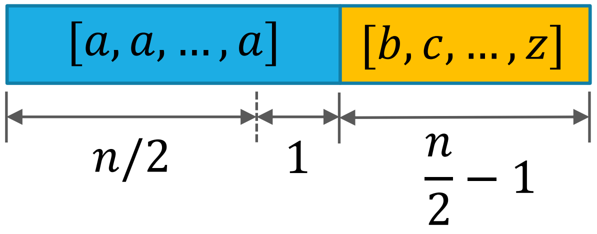
$$nums[n/2] = a$$

- Case 1b: n is odd



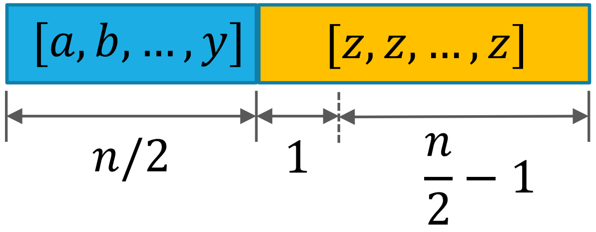
$$nums[n/2] = z$$

- Case 2a: n is even



$$nums[n/2] = a$$

- Case 2b: n is even



$$nums[n/2] = z$$

Solution #1: Use Sorting Algorithm

```c
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int findpivot(int* arr, int i, int j) {
    return i;
}

int majorityElement(int* nums, int numsSize){
    quick_sort(nums, 0, numsSize-1);
    return (nums[numsSize / 2]);
}
```

Success Details ›

Obviously, the problem has better solution

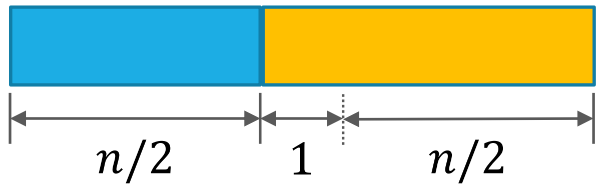
Runtime: 24 ms, faster than $\boxed{34.40\%}$ of C online submissions for Majority Element.

Memory Usage: 7.6 MB, less than 91.15% of C online submissions for Majority Element.

```c
int partition(int* arr, int i, int j, int p) {
    int pivot = arr[p];
    int L = i - 1, R = j + 1;
    do {
        do R--; while (arr[R] > pivot);
        do L++; while (arr[L] < pivot);
        if (L < R)
            swap(&arr[L], &arr[R]);
    } while (L < R);
    return R;
}

void quick_sort(int* arr, int i, int j) {
    if (i < j) {
        int p = findpivot(arr, i, j);
        int k = partition(arr, i, j, p);
        quick_sort(arr, i, k);
        quick_sort(arr, k + 1, j);
    }
}
```

Solution #2: Divide and Conquer

- **Claim**: If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.



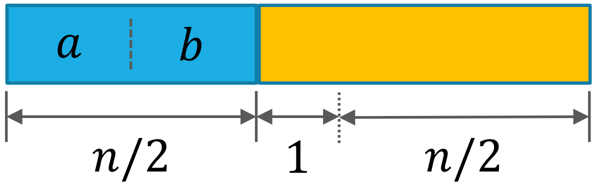
Solution #2: Divide and Conquer

- **Base Case:** If the (sub)array has only 1 elements, the majority elements is the only element.

- **Case 2:** If the majority element of the left sub-array agree with the majority element of the right sub-array, return this majority element.

- **Case 3:** If not, check the number of occurrence of the majority element in each sub-array
  3a: if one majority element occur more times than the other, return the one with higher occurrence

  3b: if not? Randomly return one



n is odd

```
int majorityElementRec(int* nums, int lo, int hi) {

}
```

Solution #2: Divide and Conquer

```c
int majorityElementRec(int* nums, int lo, int hi) {
    if (lo == hi) {
        return nums[lo];
    }

    int mid = (hi-lo)/2 + lo;
    int left = majorityElementRec(nums, lo, mid);
    int right = majorityElementRec(nums, mid+1, hi);

    if (left == right) {
        return left;
    }

    int leftCount = countInRange(nums, left, lo, hi);
    int rightCount = countInRange(nums, right, lo, hi);

    return leftCount > rightCount ? left : right;
}
```

```c
int countInRange(int* nums, int num, int lo, int hi) {
    int count = 0;
    for (int i = lo; i <= hi; i++) {
        if (nums[i] == num) {
            count++;
        }
    }
    return count;
}

int majorityElement(int* nums, int numsSize){
    return majorityElementRec(nums, 0, numsSize-1);
}
```

Success Details ›

Better than Solution #1

Runtime: 16 ms, faster than $\boxed{94.84\%}$ of C online submissions for Majority Element.

Memory Usage: 7.9 MB, less than 39.07% of C online submissions for Majority Element.

Question

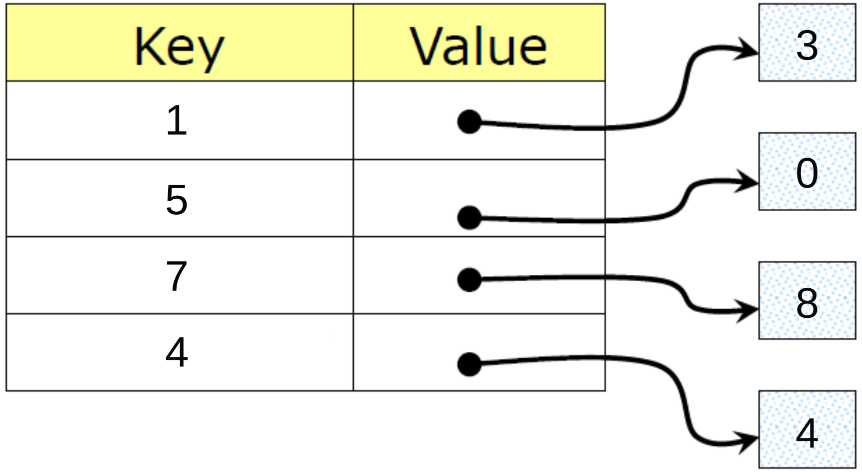
Do we have faster solution?

YES!

Solution #3: Hash Table

Claim: If an element occur more than half of the array length times, it is the majority element.

Example: $nums = [7, 1, 7, 7, 4, 1, 7, 7, 7, 7, 4, 7, 1, 4, 4]$



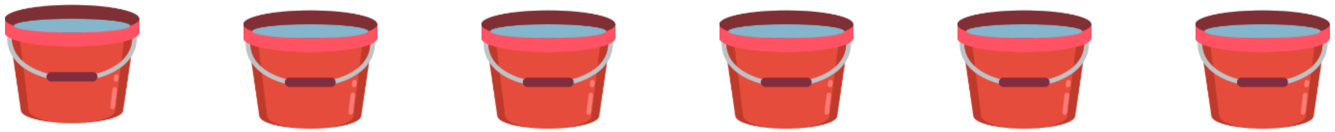
- Solution: Insert each element into Hash Table with sorted keys…
  (implementation leave as an exercise)
- It seems that we can utilize a hash table to sort an array?

Bucket Sort

- Bucket sort iteratively place each element into limited indexed buckets.
- Example: Sort array $[3, 5, 4, 3, 2, 4]$ to ascending order

Consider element: $3 \quad 5 \quad 4 \quad 3 \quad 2 \quad 4$

Count:	0	0	0	0	0	0



Index:	1	2	3	4	5	6

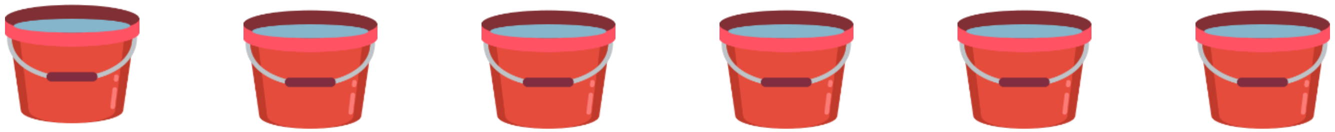
- Step 1: Initialization – create M buckets and initialize the count of elements for each bucket as 0
- Step 2: Insert each element into the corresponding bucket and update the count.

Bucket Sort

- Bucket sort iteratively place each element into limited indexed buckets.
- Example: Sort array $[3, 5, 4, 3, 2, 4]$ to ascending order

Consider element: $3 \quad 5 \quad 4 \quad 3 \quad 2 \quad 4$

Count:	0	1	2	2	1	0
Index:	1	2	3	4	5	6



- Step 3: Print the result by counting the number of the elements saved in each bucket

$[2, 3, 3, 4, 4, 5]$

Bucket Sort – Analysis

- Time Complexity:

  We only perform __1__ pass on the original array

  Therefore, use k buckets to sort an array with n elements, the time complexity is _$O(n+k)$_.

- Space Complexity: (Consider the base case similar as the example)

  To sort an array with n integers, at least _$\max(arr) - \min(arr)$_ buckets are required.

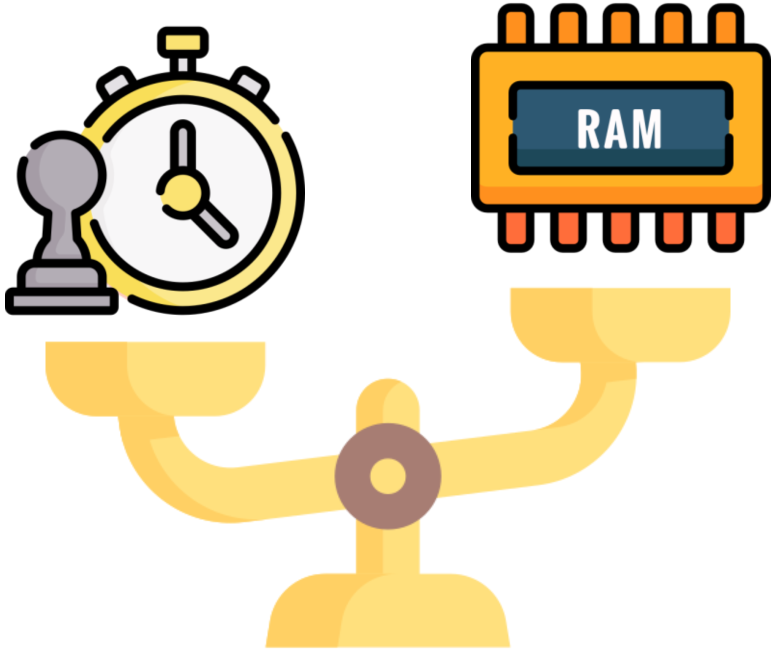
  Therefore, use bucket sort with k buckets to sort an array with n elements, the space complexity is _$O(k)$_.

- Note that $k \gg n$ may happen if the value of the array spread sparsely (稀疏分佈)

- Bucket sort prefers large array with narrow scope of data

There is no “perfect” sorting algorithm

Time Complexity

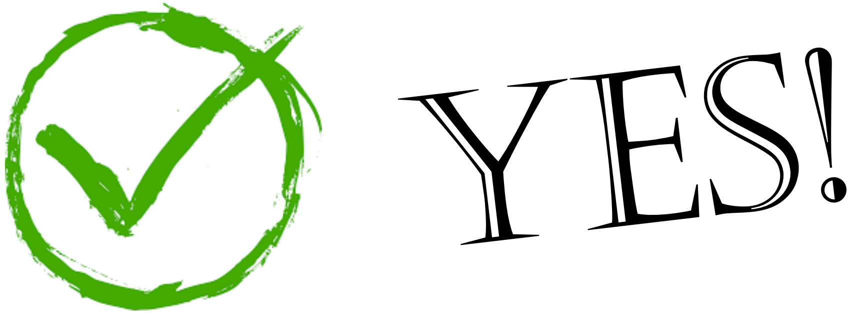


Space Complexity

- It is impractical to design sorting algorithm with both minimal time complexity and minimal space complexity
- “In practice”, we still consider Quick Sort as the “best” or “fastest” algorithm.
- We can only select the best sorting algorithm for specific task …

Back to our Programming Problem

Do we still have better solution?



Solution #4: Boyer-Moore Voting Algorithm

- YES, we do have better algorithm to solve this problem.
- This is NOT an algorithm design course, so I won't go through this method in detail...

- Example: Sort array $[7, 7, 5, 7, 5, 1, 5, 7, 5, 5, 7, 7, 7, 7, 7, 7]$ to ascending order

  Initially, set variable count = 0, and majority = NIL

Array	**7**	**7**	**5**	**7**	**5**	**1**	**5**	**7**	**5**	**5**	**7**	**7**	**7**	**7**	**7**	**7**
Count	1	2	1	2	1	0	1	0	1	2	1	0	1	2	3	4
majority	7	7	7	7	7	NIL	5	NIL	5	5	5	NIL	7	7	7	7

- If count = 0, set majority = element, count = 1, iterate to the next element;
- If current element = majority, add count by 1; else, minus count by 1.

  If count = 0, set majority = NIL;

  iterate to the next element;

https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm

Solution #4: Boyer-Moore Voting Algorithm

```c
int majorityElement(int* nums, int numsSize){
    int count = 0;
    int majority = -1;

    for (int i = 0; i < numsSize; i++) {
        if (count == 0) {
            majority = nums[i];
        }
        count += (nums[i] == majority) ? 1 : -1;
    }
    return majority;
}
```

Success Details >

Runtime: 12 ms, faster than 99.51% of C online submissions for Majority Element.

Memory Usage: 7.8 MB, less than 56.27% of C online submissions for Majority Element.

Question: Time Complexity = __$O(n)$__, Space Complexity = __$O(1)$__

Homework

Please **re-do** the programming problem
by yourself without the help of my slides.

English-Chinese Correspondence

English	Traditional Chinese	Simplified Chinese
Bubble Sort	泡沫排序	泡沫排序
Insertion Sort	插入排序	插入排序
Quick Sort	快速排序	快速排序
Merge Sort	合併排序	合并排序
Bucket Sort	桶排序	桶排序
Hash Table	哈希表	哈希表
Queue	隊列	队列
Stack	棧 / 堆棧	栈 / 堆栈
Linked List	鏈表	链表
Binary Search Tree	二叉搜尋樹	二叉搜索树