SORTING

Michael Tsai

2017/3/28

Sorting

- Definition:
- Input: $\langle a_1, a_2, \dots, a_n \rangle$ a sequence of n numbers
- Output: $\langle a'_1, a'_2, \dots, a'_n \rangle$ is a permutation (reordering) of the original sequence, such that $a'_1 \le a'_2 \le \cdots \le a'_n$

- In reality, a_i is the key of a record (of multiple fields) (e.g., student ID)
- In a record, the data fields other than the key is called **satellite data**
- If satellite data is large in size, we will only sort the pointers pointing to the records. (avoiding moving the data)

Applications of Sorting

- Example 1: Looking for an item in a list

- Q: How do we look for an item in an **unsorted** list?
- A: We likely can only linearly traverse the list from the beginning.
- → $O(n)$

- Q: What if it is sorted?
- A: We can do binary search → $O(\log n)$

- But, how much time do we need for sorting? (pre-processing)

Applications of Sorting

- Example 2:
- Compare to see if two lists are identical (list all different items)
- The two lists are *n* and *m* in length

- Q: What if they are **unsorted**?
- Compare the 1st item in list 1 with (m-1) items in list 2
- Compare the 2nd item in list 1 with (m-1) items in list 2
- ...
- Compare the n-th item in list 1 with (m-1) items in list 2
- $O(nm)$ time is needed

- Q: What if they are sorted?
- A: $O(n+m)$

- Again, do not forget we also need time for sorting. But, how much?

Categories of Sorting Algo.

- Internal Sort:
  - Place all data in the memory

- External Sort:
  - The data is too large to fit it entirely in the memory.
  - Some need to be temporarily placed onto other (slower) storage, e.g., hard drive, flash disk, network storage, etc.

- In this lecture, we will only discuss **internal sort**.
- Storage is **cheap** nowadays. In most cases, only internal sort is needed.

Some terms related to sorting

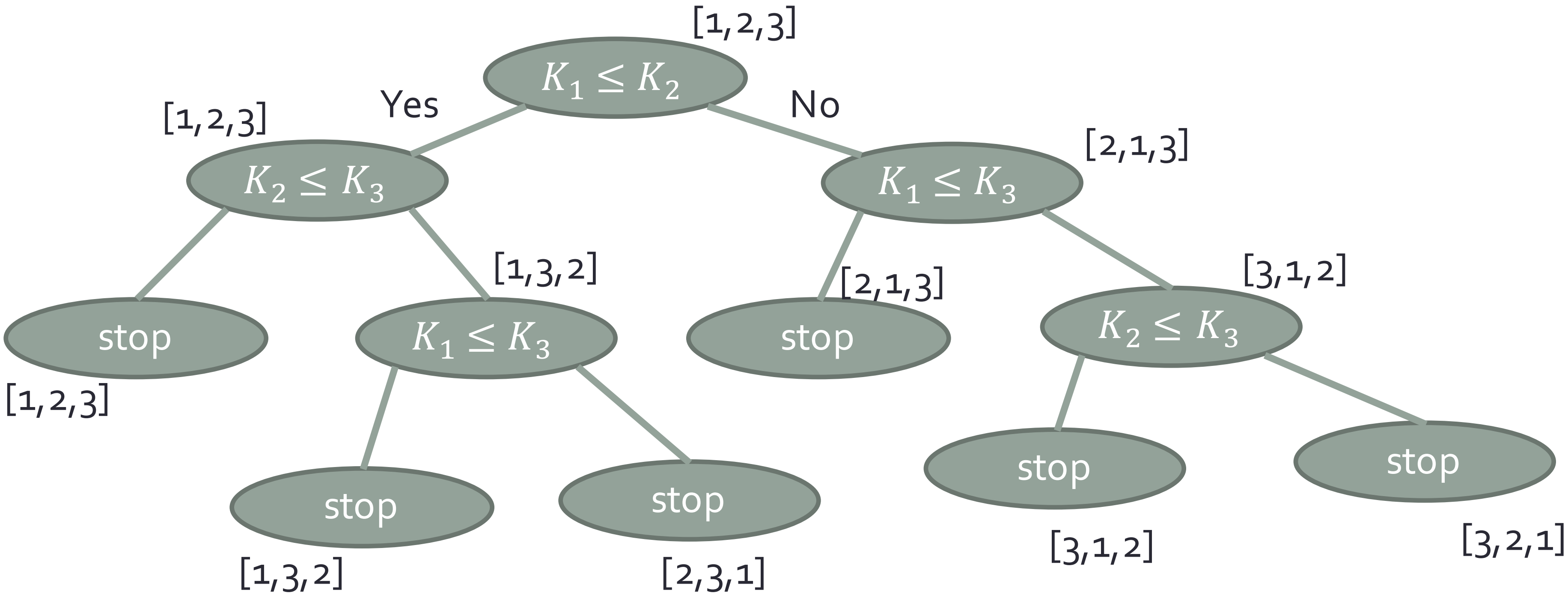
- **Stability:**
- If $a_i = a_j$ (equal key value) , then they maintain the same order before and after sorting.

- **In-place:**
- Directly sort the keys at **their current memory locations**. Therefore, only O(1) additional space is needed for sorting.

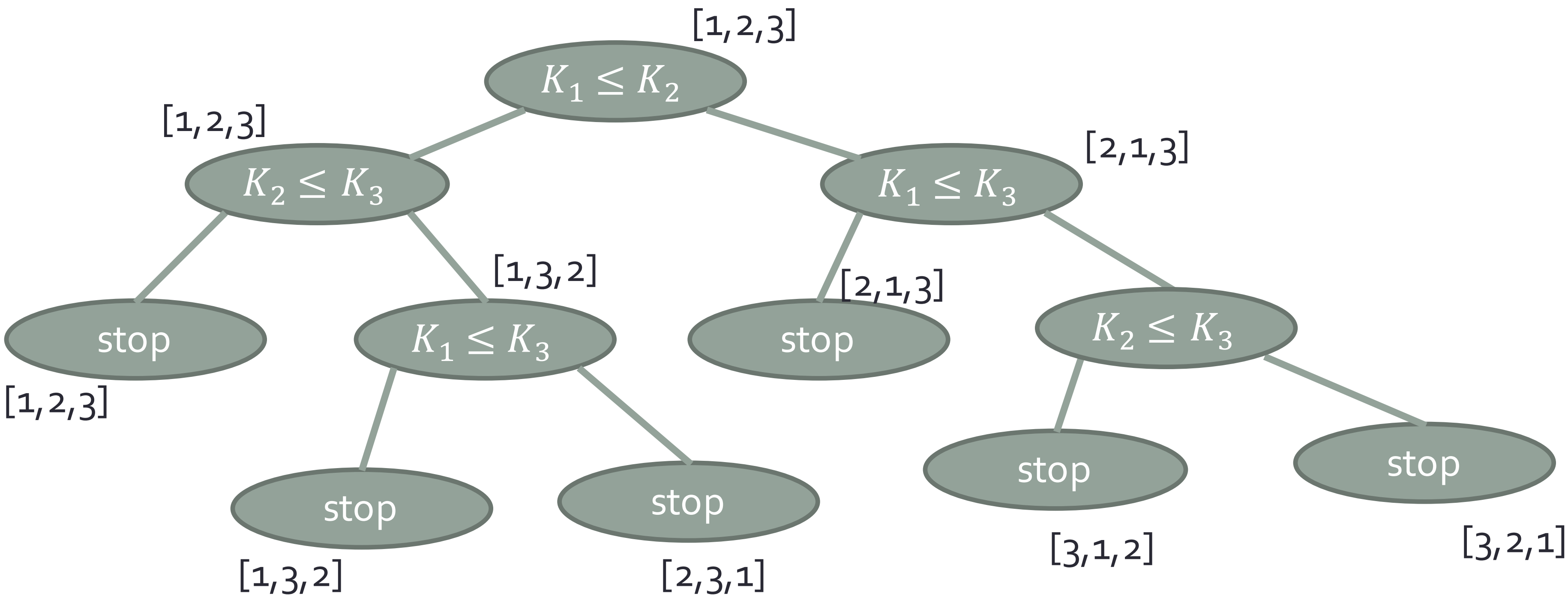
- **Adaptability:**
- If **part of the sequence is sorted**, then the time complexity of the sorting algorithm reduces.

How fast can we sort?

- Assumption: compare and swap
- Compare: compare two items in the list
- Swap: Swap the locations of these two items
- How much time do we need in the worst case?



Decision tree for sorting



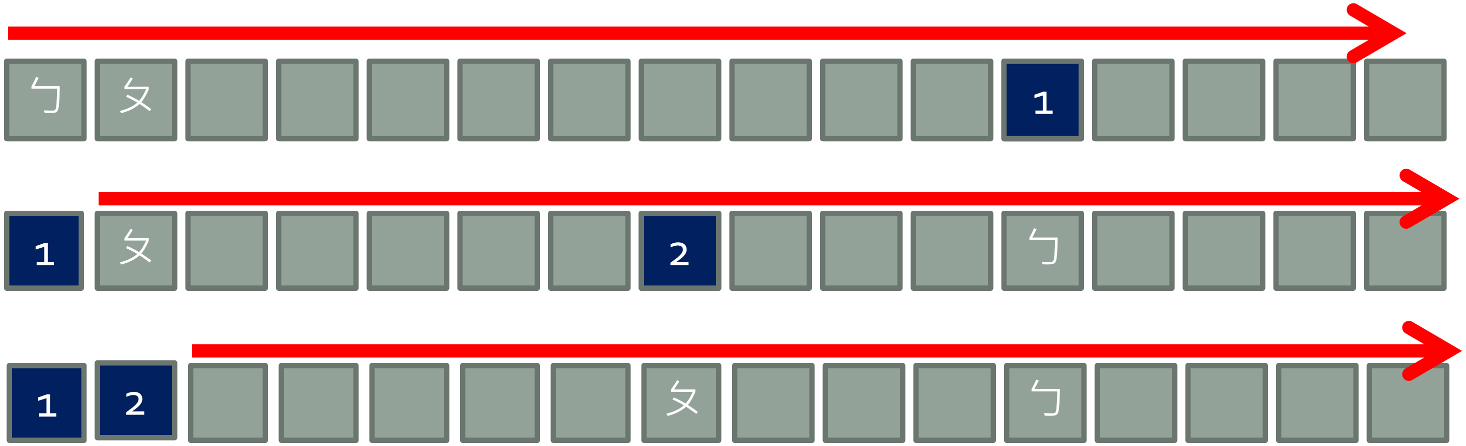
- Every node represents a comparison & swap
- Sorting is completed when reaching the leaf
- How many leaves?
- $n!$, since there are that many possible permutations

How fast can we sort?

- 所以, worst case所需要花的時間, 為此binary tree的height.
- 如果decision tree height為h, 有l個leaves
- $l \geq n!$, we have a least n! outcomes (leaves)
- $l \leq 2^h$, a binary tree (decision tree) of height h has at most 2^{h-1} leaves
- $2^h \geq l \geq n!$
- $h \geq \log_2 n!$
- $n! = n(n-1)(n-2)\dots 3 \cdot 2 \cdot 1 \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$
- $\log_2 n! \geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log_2 \frac{n}{2} = \Omega(n \log n)$
- Summary: Any "comparison-based" sorting algorithm has worst-case time complexity of $\Omega(n \log n)$.

Review: Selection Sort

- Select the smallest, move it to the first position.
- Select the second smallest, move it to the second position.
- ....
- The last item will automatically be placed at the last position.

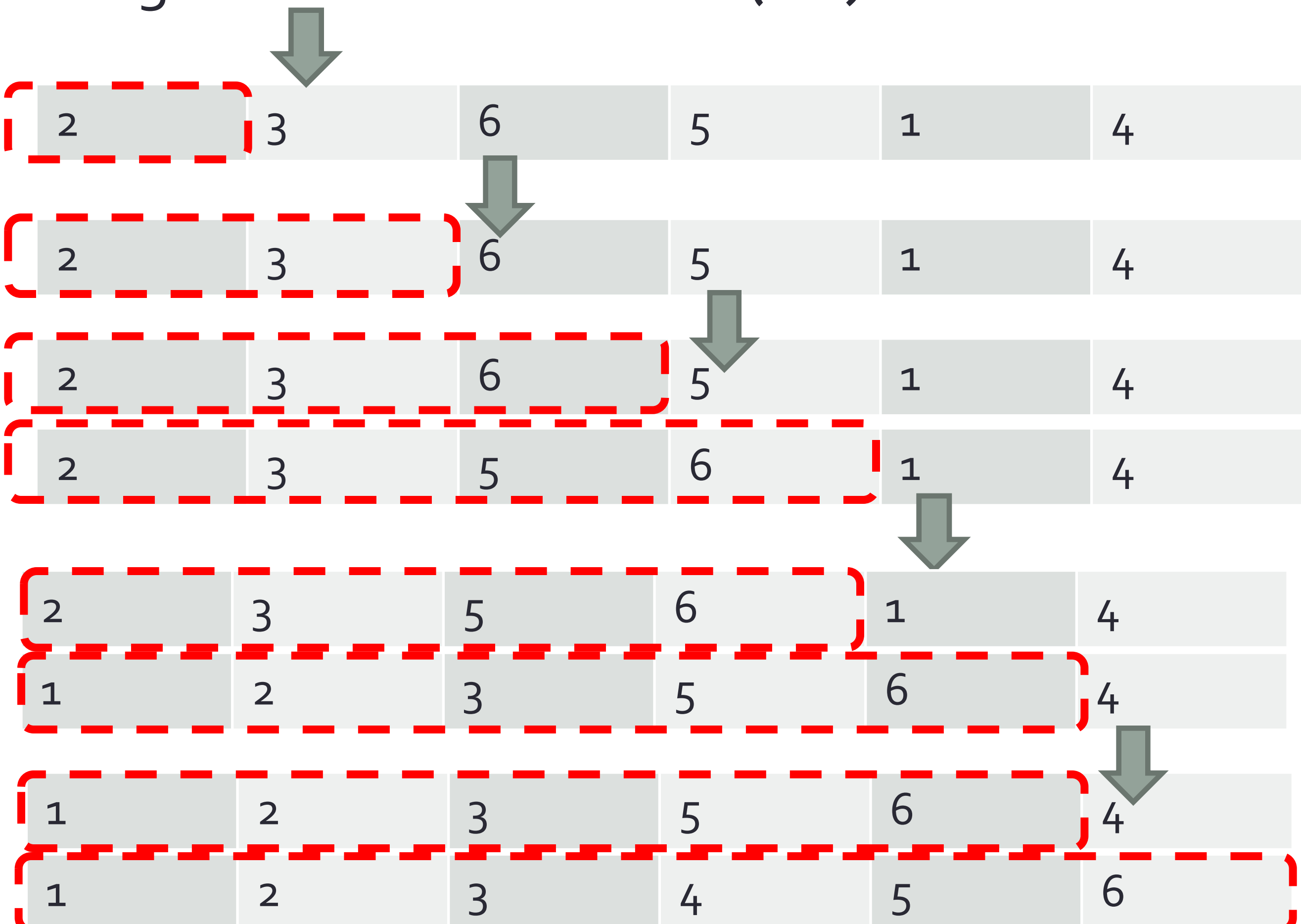


Review: Selection Sort

- Selection sort does not change the execution of the algorithm due to the current conditions.
- Always going through the entire array in each iteration.
- Therefore, its best-case, worst-case, average-case running time are all $O(n^2)$
- **Not adaptive!**
- **In-place**

Insertion Sort

- In each iteration, add one item to **a sorted list of *i* item**.
- Turning it into **a sorted list of *(i+1)* item**



Pseudo code

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Insertion Sort

- Q: How much time is needed?
- A: In the worst case, the item needs to be placed at the beginning for each and every iteration.
- (Spending time linear to the size of sorted part)
- $\sum_{1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
- Average-case complexity: $O(n^2)$. (Why?)

- Possible variation: (do those improve the time complexity?)
- 1. Use binary search to look for the location to insert.
- 2. Use linked list to store the items. Then moving takes only $O(1)$!

What's good about insertion sort

- Simple (small constant in time complexity representation)
  - Good choice when sorting a small list
- Stable
- In-place
- Adaptive
  - Example: In ⟨1,2,5,3,4⟩, only two inversions <5,3>, <5,4>.
  - The running time for insertion sort: O(n+d), d is the number of inversions
    Best case: O(n) (No inversion, **sorted**)
- Online:
  No need to know all the numbers to be sorted. Possible to sort and take input at the same time.

Merge Sort

- Use **Divide-and-Conquer** strategy
- Divide-and-Conquer:
  - Divide: Split the big problem into small problems
  - Conquer: Solve the small problems
  - Combine: Combine the solutions to the small problems into the solution of the big problems.

- Merge sort:
  - Divide: Split the n numbers into two sub-sequences of $n/2$ numbers
  - Conquer: Sort the two sub-sequences (use recursive calls to delegate to the clones)
  - Combine: Combine the two **sorted sub-sequences** into the one **sorted** sequence

Merge Sort

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Divide

Conquer x2

Combine

Merge Sort

A[]: the array to be sorted
temp: temporarily storage
left,right: the left & right indices of the range to be sorted.

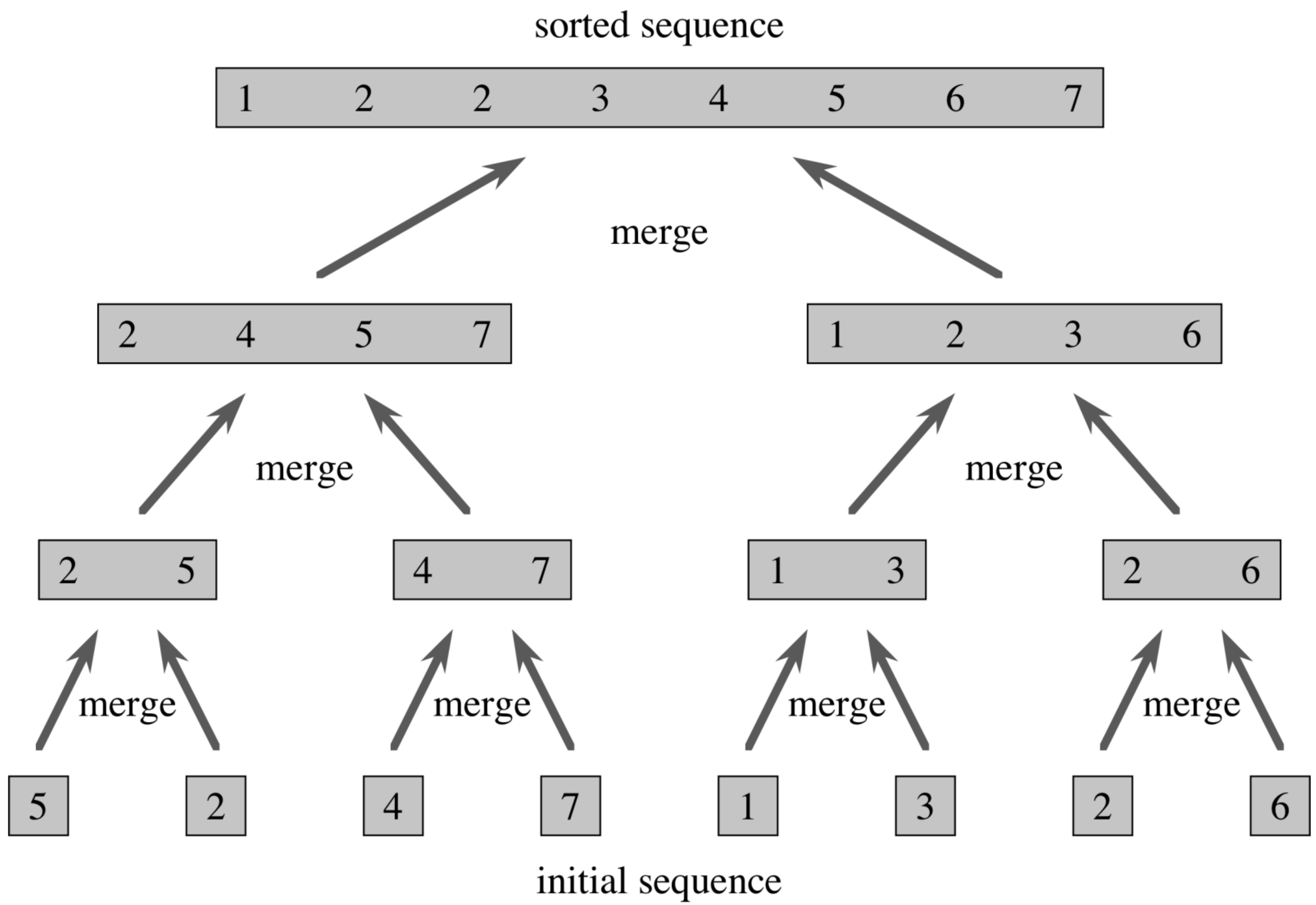
```
void Mergesort(int A[], int temp, int left, int right) {
        int mid;
        if (right > left) {
                mid=(right+left)/2;
                Mergesort(A,temp,left,mid);
                Mergesort(A,temp,mid+1,right);
                Merge(A,temp,left,mid+1,right);
        }
}
```

Divide

Conquer

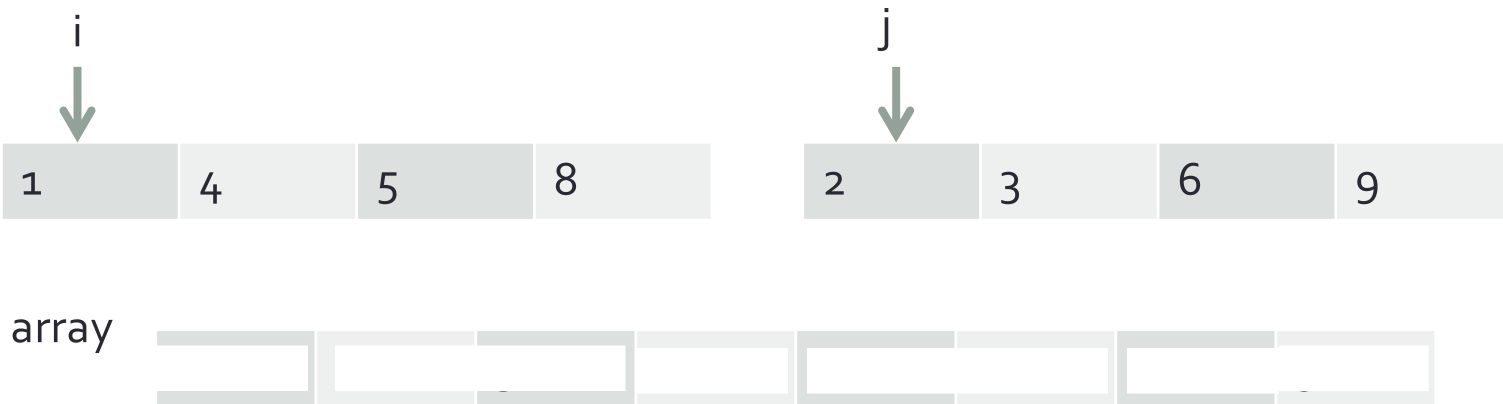
Combine

Merge Sort: Example



How to combine (merge)?

Temporary storage



Original array

- Running time: $O(n_1 + n_2) = O(n)$, n_1和n_2 are the lengths of the two sub-sequences.
- A temporary storage of size O(n) is needed during the merge process

Implementation: Merge

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

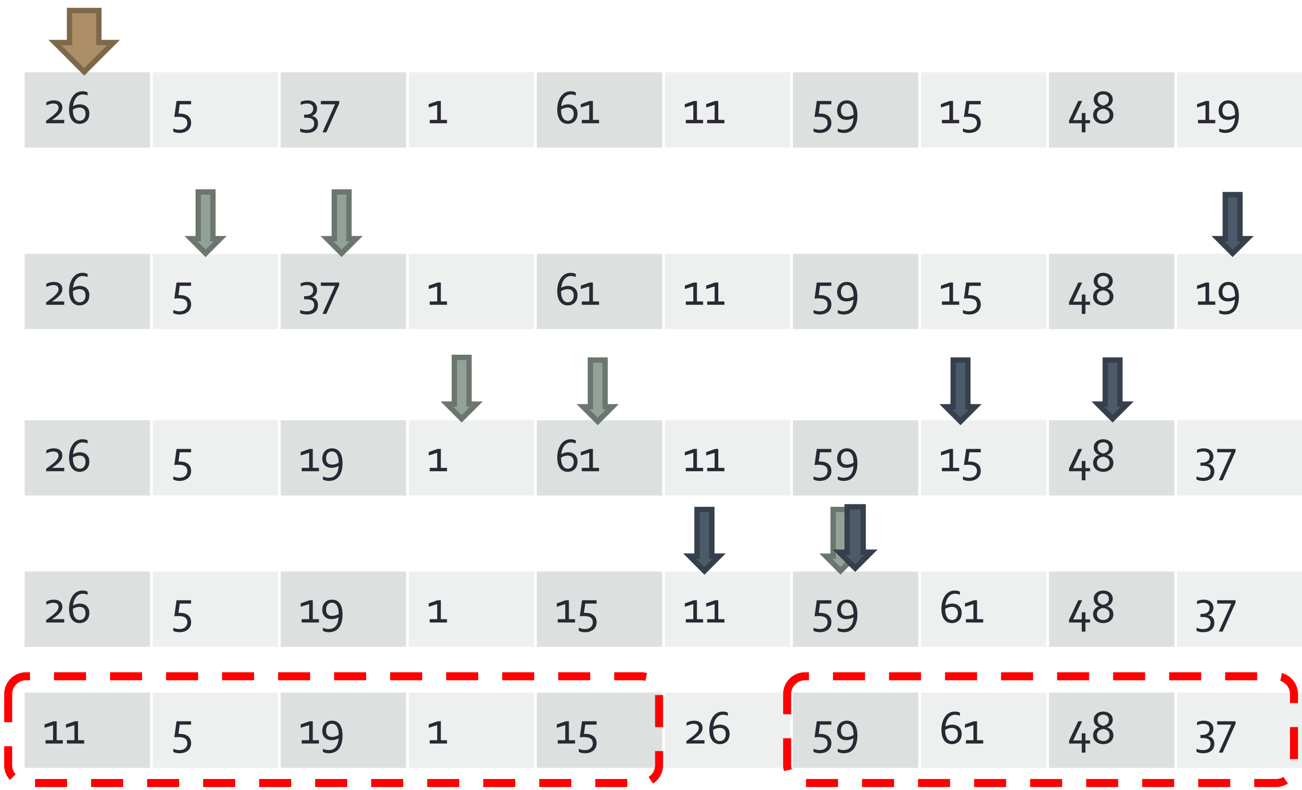
Merge sort

- Every item to be sorted is processed once per "pass"→$O(n)$
- How many passes is needed?
- The length of the sub-sequence **doubles** every pass, and finally it becomes the large sequence of n numbers
- Therefore, $\lceil \log_2 n \rceil$passes.
- Total running time: $O(n \log_2 n) = O(n \log n)$
- Worst-case, best-case, average-case: $O(n \log n)$ (**Not adaptive**)

- Not in-place: need additional storage for sorted sub-sequences
- Additional space: O(n)

Quick Sort

- Find a pivot(支點), manipulate the locations of the items so that:
  - (1) all items to its left is smaller or equal (**unsorted**),
  - (2) all items to its right is larger
- Recursively call itself to sort the left and right sub-sequences.



Pseudo Code

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q - 1)
      QUICKSORT(A, q + 1, r)
```

Divide

Conquer x2

No Combine!

Quick Sort

11	5	19	1	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	19	15	26	59	61	48	37
1	5	11	15	19	26	59	61	48	37
1	5	11	15	19	26	59	61	48	37
1	5	11	15	19	26	48	37	59	61
1	5	11	15	19	26	37	48	59	61
1	5	11	15	19	26	37	48	59	61

Quick Sort: Worst & Best case

- But worst case running time is still $O(n^2)$
- Q: Give an example which produces worst-case running time for the quick sort algorithm.
- In this case: running time is $O(n^2)$

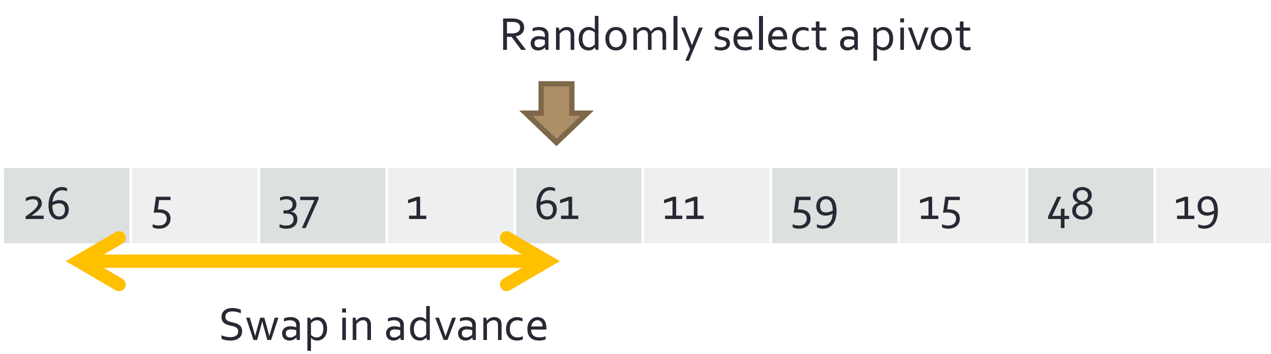
- Best case?
- Pivot can split the sequence into two sub-sequences of equal size.
- Therefore, T(n)=2T(n/2)+$O(n)$
- T(n)=$O(n\log n)$

Randomized Quick Sort

- Avoid worst case to happen frequently

- Randomly select a pivot (not always the leftmost key)

- Reduce the probability of the worst case
- However, worst case running time is still $O(n^2)$



Average running time

- Better if the selection of pivot can evenly split the sequence into two sub-sequences of equal size

- Why the average running time is close to the best-case one?
- 假設很糟的一個狀況: 每次都分成1:9

Time needed for the "9/10 subsequence"

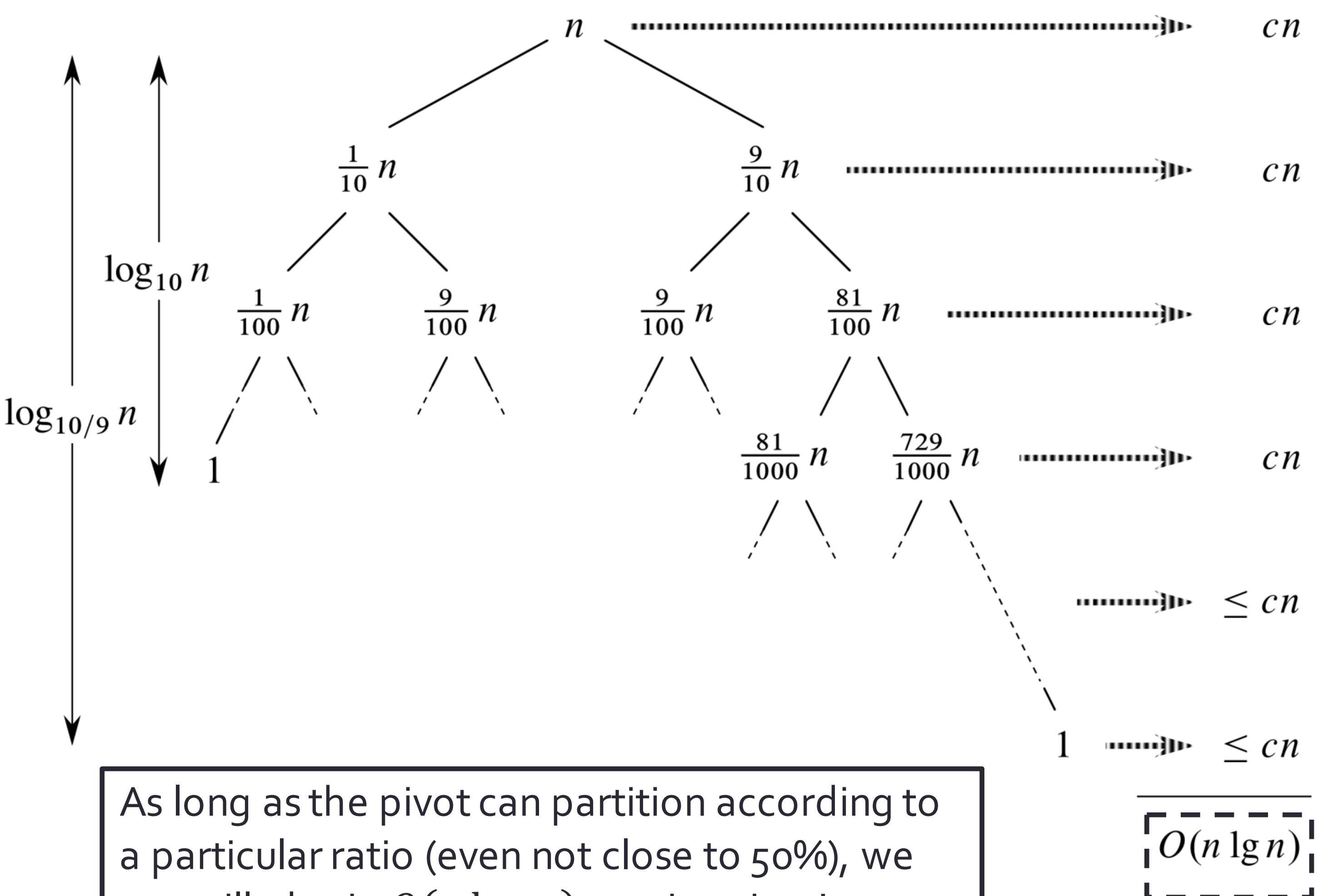
Time needed for partitioning

- $T(n) = T(9n/10) + T(n/10) + cn$

Time needed for the "1/10 subsequence"

- $= \left(T\left(\frac{81n}{100}\right) + T\left(\frac{9n}{100}\right) + \frac{9cn}{10}\right) + \left(T\left(\frac{9n}{100}\right) + T\left(\frac{1n}{100}\right) + \frac{cn}{10}\right) + cn$
- $= \cdots$

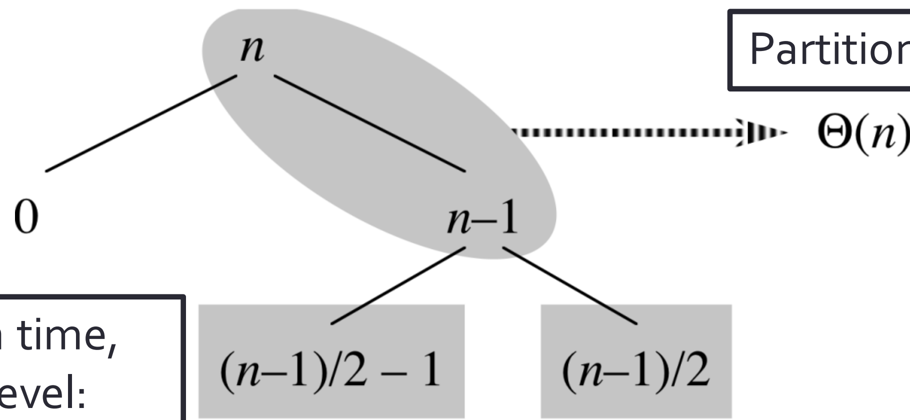
Average running time



As long as the pivot can partition according to a particular ratio (even not close to 50%), we can still obtain $O(n \log n)$ running time!

Average running time

Case 1:
Worst case for the first level partition,
but best-case for second level.

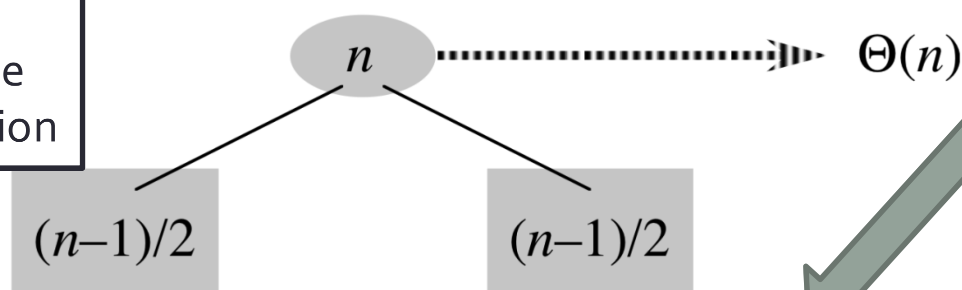


Partition time for the first level: $\Theta(n)$

Partition time, second level: $\Theta(n-1)$

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

Case 2:
Best case for the first level partition



Partition time for the first level: $\Theta(n)$

Same!
(Case 1 has larger constant)
The better-partitioned level would "absorb" the extra running time for worse-partitioned level.

比較四大金剛

	Worst	Average	Additional Space?
Insertion sort	$O(n^2)$	$O(n^2)$	O(1)
Merge sort	$O(n \log n)$	$O(n \log n)$	O(n)
Quick sort	$O(n^2)$	$O(n \log n)$	O(1)
Heap sort	$O(n \log n)$	–	O(1)

Not covered today!

- Insertion sort: quick with small input size n. (small constant)
- Quick sort: Best average performance (fairly small constant)
- Merge sort: Best worst-case performance
- Heap sort: Good worst-case performance, no additional space needed.
- Real-world strategy: **a hybrid of insertion sort** + others. Use **input size** *n* to determine the algorithm to use.