香港中文大學(深圳)
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 11: QuickSort

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

QuickSort

- Sorting problem
  - Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - Output: a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of input such that $a'_1 <= a'_2 <= \ldots <= a'_n$

- Main features of QuickSort
  - Very fast known sorting algorithm in practice
  - Average running time is O (NlogN)
  - Worst case performance is $O(N^2)$ (but very unlikely)

Deterministic vs. randomized

- All previously learnt algorithms are deterministic
  - They do not involve any randomization
  - Given the same input, a deterministic algorithm always executes in the same way, no matter how many times we repeat it
    - The running cost therefore is also the same

- Randomized algorithms:
  - We include one more basic operation: $random(x, y)$
    - Generate an integer from $[x, y]$ uniformly at random

Randomized algorithms

- Randomized algorithms:
  - Given the same input, the algorithm may run in a different way since we bring randomization to our execution
  - The running cost, i.e., the number of basic operations, is also a random variable

- For example, every time when we execute the flipCoin algorithm, it may run in a different way
  - In the worst case, it may execute infinitely, even though the probability is close to zero
  - In randomized algorithms, we consider the expected running cost

Algorithm: *flipCoin()*

```
r ← RANDOM(0,1)
while r != 1
    r = RANDOM(0,1)
```

Expected running cost

- Let X be a random variable of the running cost, i.e., the number of basic operations of a randomized algorithm on an input. The expected running cost is then $E[X]$
  - We cannot consider worst case running time on random algorithms since it may run infinitely with very tiny chances

- Consider the expected running cost of flipCoin algorithm
  - Let X be the running cost (the number of basic operations) of flipCoin
  - $\Pr[X = 2] = \frac{1}{2}$
  - $\Pr[X = 4] = \frac{1}{4}$
  - …
  - $\Pr[X = 2i] = \frac{1}{2^i}$
  - $E[X] = 2 \cdot \sum_{i=1}^{+\infty} \frac{i}{2^i} = 4 = O(1)$

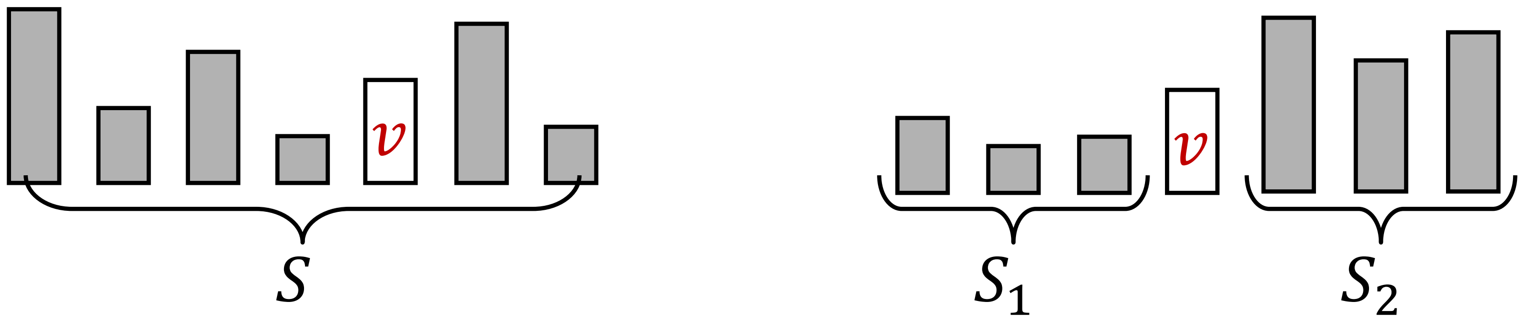
Algorithm: ***flipCoin()***

```
r ← RANDOM(0,1)
while r != 1
    r = RANDOM(0,1)
```

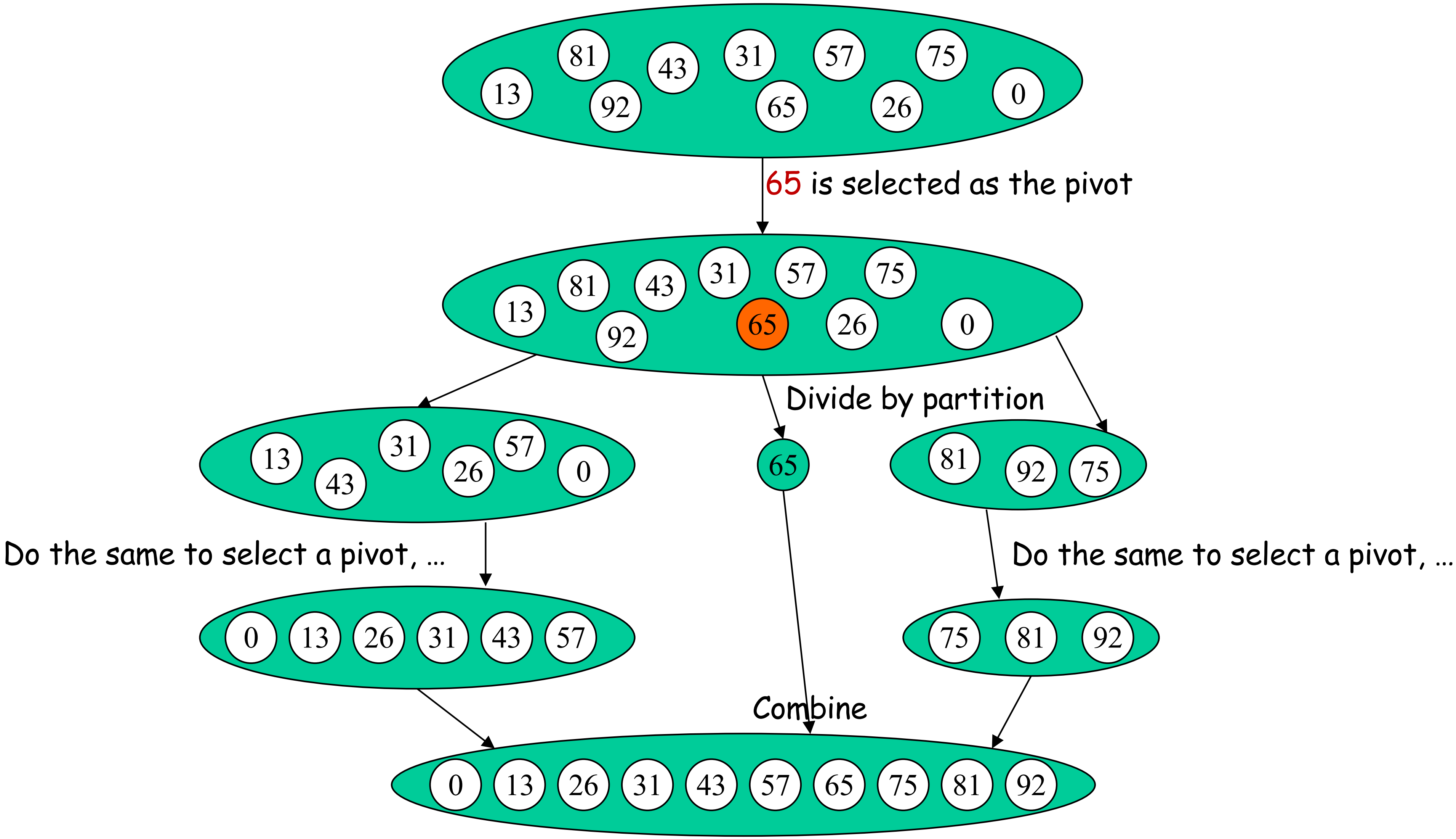
QuickSort (divide-and-conquer)

- A randomized algorithm using divide-and-conquer
  - Time complexity: $O(n \cdot \log n)$ expected running time

- High level idea: (assume that elements are distinct)
  - Randomly pick an element, denoted as the pivot, and partition the remaining elements to three parts
    - The pivot
    - The elements in the left part: smaller than the pivot
    - The elements in the right part: larger than the pivot
    - For the left part and right part: repeat the above process if the number of elements is larger than 1



QuickSort: example



QuickSort: implementation

Algorithm: *quicksort(arr, left, right)*

```
if left>=right
   return
pivot←RANDOM(left,right) // randomly select a pivot from [left,right]
pivotNewposition =partition(arr, left, right, pivot)
quicksort(arr, left, pivotNewposition-1)
quicksort(arr, pivotNewposition+1, right)
```

- A key step: partition
  - **Input**: array, left position, right position, randomly selected pivot position
  - **Goal**: divide the array into three parts: the left partition (smaller than pivot element), pivot position, and the right partition (larger than pivot element)
  - **Return**: the new pivot position which helps divide it into sub problems

pivot =3

4	2	3	6	9	5	7
left=[0]	[1]	[2]	[3]	[4]	[5]	[6]=right

pivotNewposition =4

4	2	3	5	6	7	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]

QuickSort: partition

Algorithm: ***partition(arr, left, right, pivot)***

```
pivotVal=arr[pivot] //record the pivot data
Swap(arr[right], arr[pivot]) //swap the pivot data and the last data
nextsmallpos=left//record the next position to put data smaller than pivotVal
for j from left to right-1
    if arr[j] < pivotVal
       swap(arr[nextsmallpos], arr[j])
       nextsmallpos++
Swap(arr[nextsmallpos], arr[right])
return nextsmallpos
```

pivot =3 left =0 right =6

4	2	3	6	9	5	7
[0]	[1]	[2]	[3]	[4]	[5]	[6]

pivotVal =6 nextsmallpos =0 j =0

4	2	3	7	9	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

pivotVal =6 nextsmallpos =3 j =3

4	2	3	7	9	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

pivotVal =6nextsmallpos =3 j =4

4	2	3	7	9	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

QuickSort: partition

Algorithm: ***partition(arr, left, right, pivot)***

```
pivotVal=arr[pivot] //record the pivot data
Swap(arr[right], arr[pivot]) //swap the pivot data and the last data
nextsmallpos=left//record the next position to put data smaller than pivotVal
for j from left to right-1
    if arr[j] < pivotVal
        swap(arr[nextsmallpos++], arr[j])
Swap(arr[nextsmallpos], arr[right])
return nextsmallpos
```

pivotVal =6 nextsmallpos =3 j =5

4	2	3	7	9	5	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

pivotVal =6 nextsmallpos =4 j =5

4	2	3	5	9	7	6
[0]	[1]	[2]	[3]	[4]	[5]	[6]

pivotVal =6 nextsmallpos =4 j =5

4	2	3	5	6	7	9
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Return nextsmallpos =4.
The position of the pivot

MergeSort v.s. Quicksort

- Both MergeSort and QuickSort use the Divide-and-Conquer paradigm

- When MergeSort executes the merge operation
  - Requires an additional array to do the merge operation
  - Needs to do additional data copy: copy to additional array and then copy back to the input array
- When QuickSort executes the partition operation
  - Operates on the same array
  - No additional space required

- Quicksort is typically 2-3 times faster than MergeSort even though they have the same (expected) time complexity $O(n \cdot \log n)$

QuickSort

- There are many ways for implementation

- In practice, a good way is:
  - Set the *pivot* to the median among the first, center and last elements
  - Exchange the second last element with the *pivot*
  - Set *i* at the second element
  - Set *j* at the third last element

QuickSort

- While *i* is on the left of *j*, move *i* right, skipping over elements that are smaller than the *pivot*
- Move *j* left, skipping over elements that are larger than the *pivot*
- When *i* and *j* have stopped, *i* is pointing at a large element and *j* at a small element

QuickSort

- If *i* is to the left of *j*, swap A [*i*] with A [*j*] and continue
- When *i* is larger than *j*, swap the pivot element with the element at *i*
- All elements to the left of pivot are smaller than pivot, and all elements to the right of pivot are larger than pivot
- What to do when some elements are equal to pivot?

QuickSort

```
private static int median3(int[] a, int left, int right) {
        // Ensure a[left] <= a[center] <= a[right]
        int center = (left + right) / 2;
        if (a[center] < a[left])
                swap (a, left, center);
        if (a[right] < a[left])
                swap (a, left, right);
        if (a[right] < a[center])
                swap (a, center, right);

        // Place pivot at position right - 1
        swap (a, center, right - 1);
        return a[right - 1];
}
```

QuickSort

```
/* Main quicksort routine */
private static void quicksort(int[] a, int left, int right) {
        if (left + CUTOFF <= right) {
                int pivot = median3(a, left, right);
                // Begin partitioning
                int i = left+1, j = right - 2;
                while (true) {
                        while (a[i] < pivot) {i++;}
                        while (a[j] > pivot) {j--;}
                        if (i >= j) break; // i meets j
                        swap (a, i, j);
                }
                swap (a, i, right - 1); // Restore pivot
                quicksort(a, left, i - 1); // Sort small elements
                quicksort(a, i + 1, right); // Sort large elements
        } else
                insertionSort(a, left, right);
}
```

QuickSort - median3 example

- Example: 8 1 4 9 6 3 5 2 7 0

  0 1 4 9 6 3 5 2 7 8

  Start: 0 1 4 9 7 3 5 2 6 8

  i if smaller if bigger *j*

  Move *i*: 0 1 4 9 7 3 5 2 6 8

  i *j*

  Move *j*: 0 1 4 9 7 3 5 2 6 8

  i *j*

QuickSort -median3 example

1st swap: 0 1 4 2 7 3 5 9 6 8

i *j*

Move *i*: 0 1 4 2 7 3 5 9 6 8

i *j*

Move *j*: 0 1 4 2 7 3 5 9 6 8

i *j*

QuickSort – median3 example

2nd swap: 0 1 4 2 5 3 7 9 6 8

i j

Move *i*: 0 1 4 2 5 3 7 9 6 8

i *meet j*

Move *j*: 0 1 4 2 5 3 7 9 6 8

(i & j crossed) *j* *i*

Swap element at *i* with pivot

$\boxed{0\ 1\ 4\ 2\ 5\ 3}$ 6 $\boxed{9\ 7\ 8}$

Exercise

- Use QuickSort to sort the following sequence of integer values

5,7,4,1,0,2,9

(1)Selecting pivot:

1,7,4,5,0,2,9

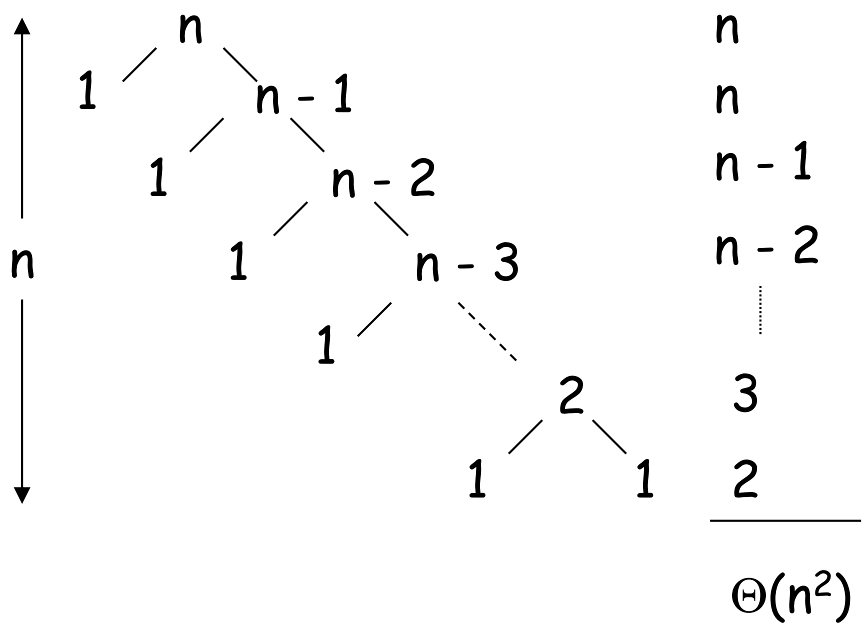
5 will be selected as the pivot

(2) Sorting:

Finish it by yourself…

Worst case partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced



- Recurrence: q=1

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

$T(1) = \Theta(1)$

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

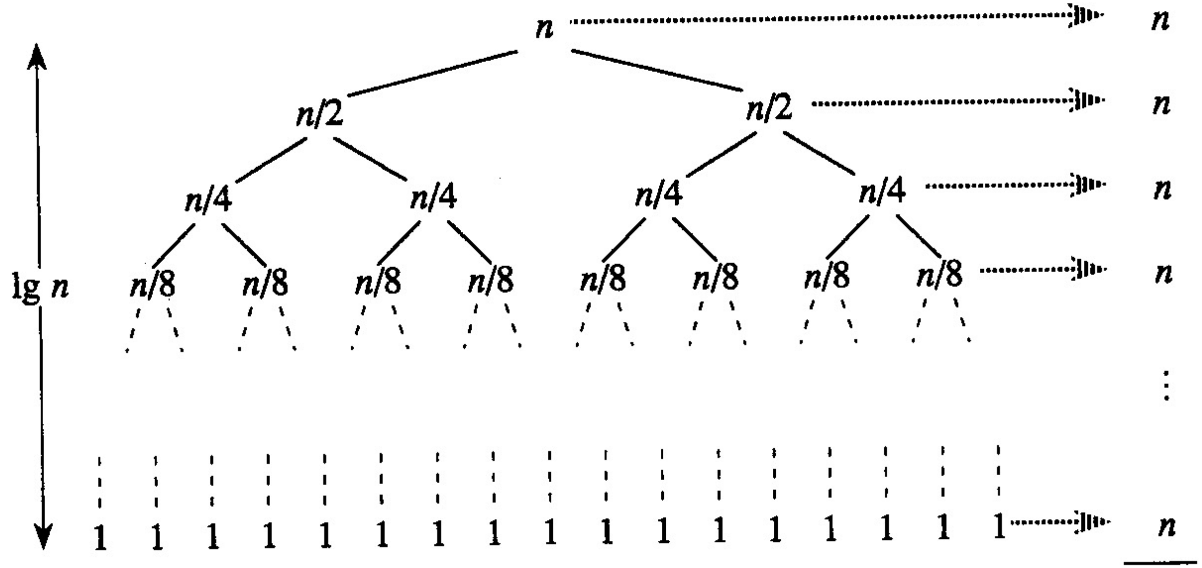
$$= n + \left(\sum_{k=1}^{n} k\right) - 1 = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Best case partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

  T(n) = 2T(n/2) + Θ(n)

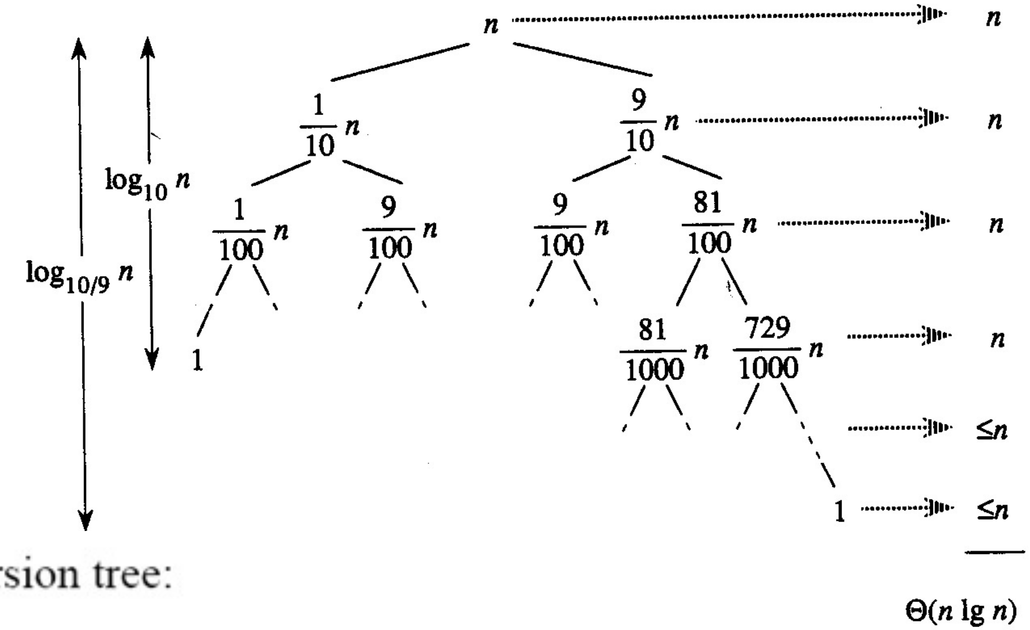
  T(n) = Θ(nlgn)



Case between worst and best

- 9-to-1 proportional split

$$Q(n) = Q(9n/10) + Q(n/10) + n$$



- Using the recursion tree:

longest path: $Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 nlgn$

shortest path: $Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n(\log_{10} n + 1) = c_1 nlgn$

Thus, $Q(n) = \Theta(nlgn)$

QuickSort: complexity analysis

- Expected running time:
  - $O(n \cdot \log n)$
  - We need to count the number of comparisons in QuikSort
  - How many times will an element get selected as a pivot in quicksort?
    - At most once

- Let e_x denote the x-th smallest element. When will two element e_i and e_j get compared such that $i < j$?
    - e_i and e_j are not compared, if any element between them gets selected as a pivot before them

Complexity analysis (optional)

- Observation: e_i and e_j are compared if and only if either one is the first among $e_i, e_{i+1,\ldots}, e_j$ picked as a pivot
  - What is $\Pr[X_{i,j} = 1]$?

- Define an indicator random variable $X_{i,j}$ to be 1 if e_i and e_j are compared. Otherwise $X_{i,j} = 0$
  - Then, we know $\Pr[X_{i,j} = 1] = \frac{2}{j-i+1}$
  - Accordingly, $E[X_{i,j}] = 1 \cdot \Pr[X_{i,j} = 1] + 0 \cdot \Pr[X_{i,j} = 0] = \frac{2}{j-i+1}$

Complexity analysis (optional)

- The total number of comparisons is:
  - $E\left[\sum_{1 \le i < j \le n} X_{i,j}\right]$
  - is equal to $\sum_{1 \le i < j \le n} E[X_{i,j}]$ by linearity of expectation

- Let X be a random variable to denote the total number of comparisons in QuickSort
  - Then, $X = \sum_{1 \le i < j \le n} X_{i,j}$
  - Thus, $E[X] = \mathrm{E}[\sum_{1 \le i < j \le n} X_{i,j}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$
  - We prove that $E[X] = O(n \cdot \log n)$

Complexity analysis (optional)

- $E[X] = E[\sum_{1 \le i < j \le n} X_{i,j}] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$, then $E[X] = O(n \cdot \log n)$

Proof. Let $j - i = x$, when $x = 1$, we have $i = 1, j = 2$, or $i = 2, j = 3$, or $i = 3, j = 4$, …, or $i = n-1, j = n$ options. Similarly, we can derive for $j - i = x$, we have $n - x$ options.

Therefore the above equation can be rewritten as:

$$E[X] = 2\sum_{x=1}^{n-1} \frac{n-x}{x+1} = 2\sum_{x=1}^{n-1} \frac{n+1-x-1}{x+1} = 2(n+1) \cdot \sum_{x=1}^{n-1} \frac{1}{x+1} - 2n$$

Now, we use the fact that $1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n} = O(\log n)$, which is called the harmonic series, and is frequently encountered in complexity analysis.

Hence, $E[X] = O(n \cdot \log n)$ and we prove that the expected running time of QuickSort is $O(n \cdot \log n)$.

Comparison of sorting algorithms

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	√	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	×	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge sort	√	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Heap sort	×	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$
Quick sort	×	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(1)$

** A sorting algorithm is said to be stable, if two objects with equal keys appear in the same order in sorted output, as they appear in the input array

Selection Sort: 5，1，2，1* => 1*，1，2，5

How fast can we sort?

- In terms of the worst case analysis, we have seen algorithms with either $O(n \log n)$ or $O(n^2)$

- Is there any hope that we can do better than $O(n \log n)$, for example $O(n)$? In other words, what is the best we can achieve?

- Let's consider the scenario where the operations allowed on keys are only comparisons, e.g., $<$, $>$, $=$, …

Theorem: Any comparison-based sorting algorithm will take $\Omega(n \cdot \log n)$ time

Recommended reading

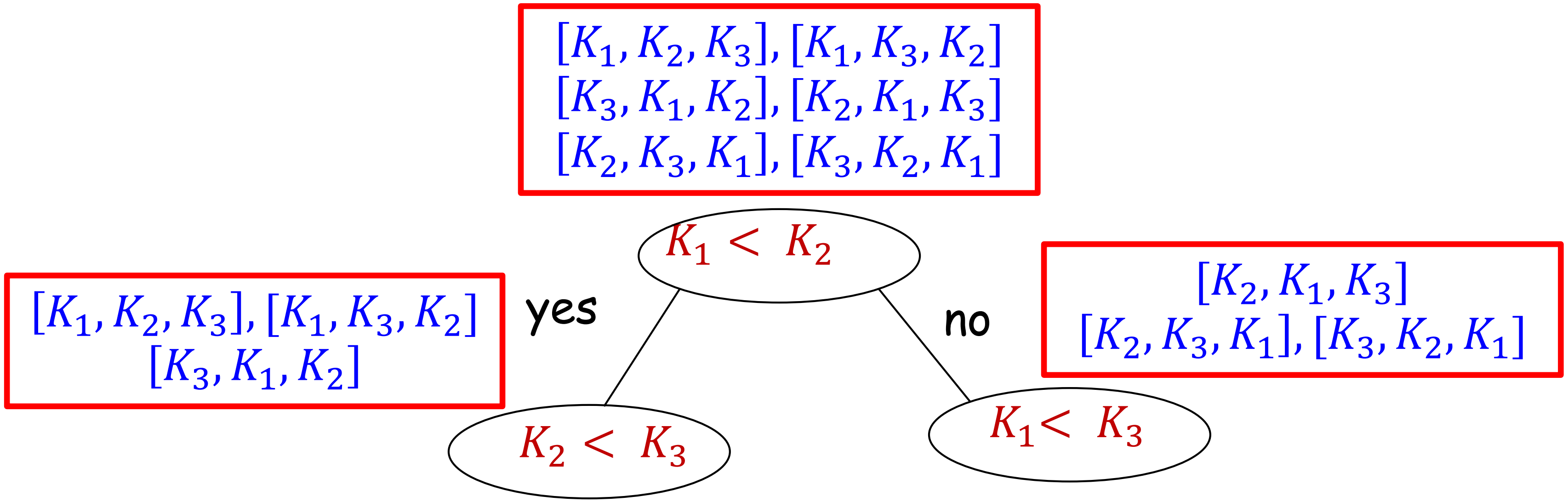
- Reading this week
  - Chapter 7, textbook
- Next lecture
  - More sorting algorithms: chapter 8, textbook

How fast can we sort?

- Given an array A with length n, there are $n!$ different permutations of the elements therein
  - If n = 3, then there are 6 permutations:
    - A[1], A[2], A[3]
    - A[1], A[3], A[2]
    - A[2], A[1], A[3]
    - A[2], A[3], A[1]
    - A[3], A[1], A[2]
    - A[3], A[2], A[1]
  - The goal of the sorting problem is essentially to decide which of the $n!$ permutations corresponds to the final sorted order

How fast can we sort?

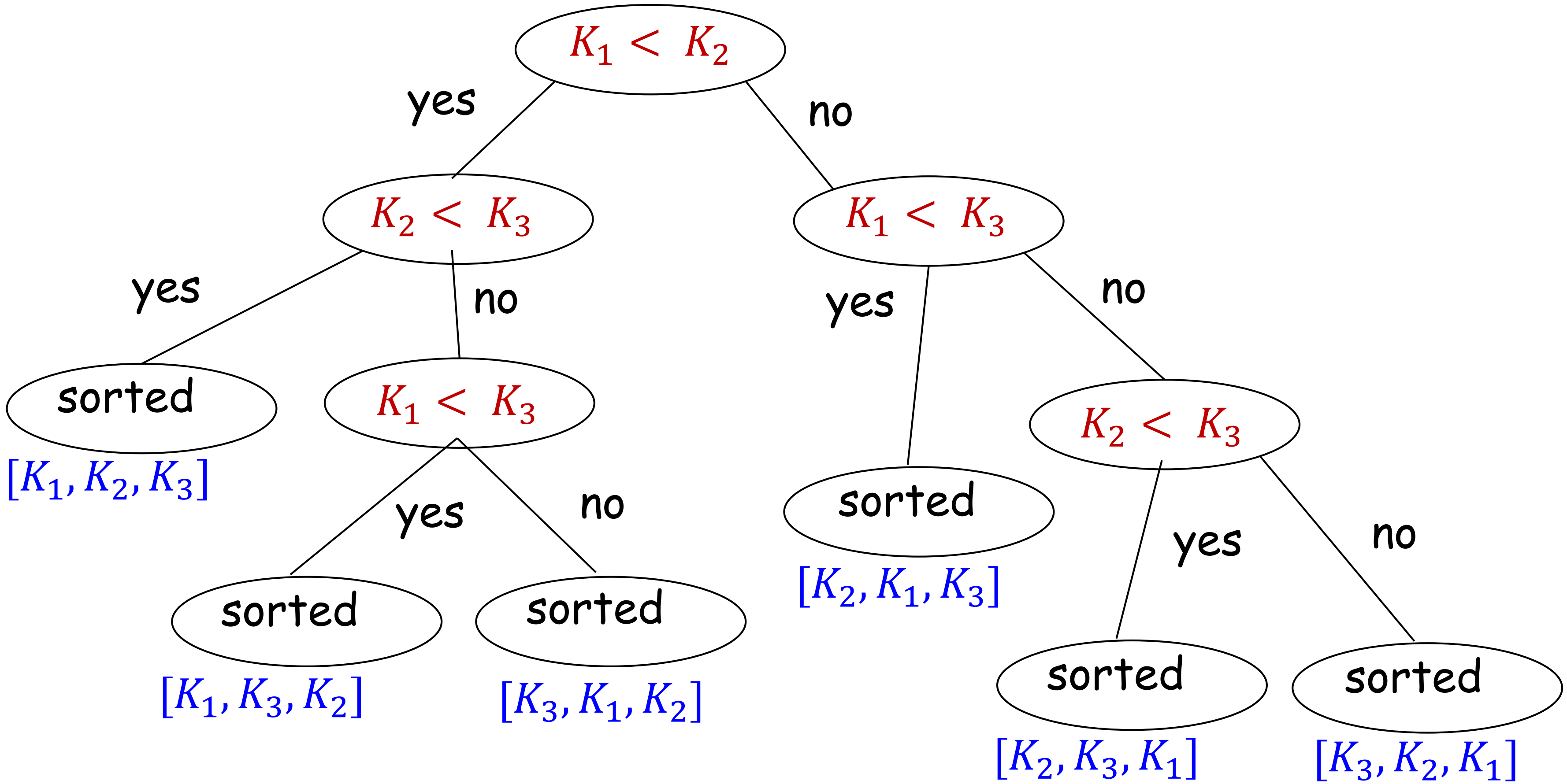
- Consider a decision tree that describes the sorting :
  - A node represents a key comparison
  - An edge indicates the result of the comparison (yes or no). We assume that all keys are distinct



The result of a comparison, e.g., $K_1 < K_2$, makes the possible number of permutation satisfying the constraint ($\text{e.g.}, K_1 < K_2$) become smaller and smaller

How fast can we sort?

- Consider a decision tree that describes the sorting:
  - A node represents a key comparison
  - An edge indicates the result of the comparison (yes or no). We assume that all keys are distinct



How fast can we sort?

- Theorem: Any decision tree that sorts n distinct keys has a height of at least $\log_2 n! + 1$.

Proof: When sorting n keys, there are $n!$ different possible results. Thus, every decision tree for sorting must have at least $n!$ leaves.

Note a decision tree is a binary tree, which has at most 2^{k-1} leaves if its height is k. Therefore, $2^{k-1} \geq n!$, the height must be at least $\mathrm{k} \geq \log_2 n! + 1$.

Notice: $\log_2 n! = \sum_{i=1}^{n} \log i \geq \sum_{i=\frac{n}{2}}^{n} \log i \geq \frac{n}{2} \cdot \log \frac{n}{2} = \Omega(n \cdot \log n)$

Therefore, the comparison based sorting algorithms needs $\Omega(n \cdot \log n)$ comparisons in the worst case