CSC3100 Data Structures
Lecture 12: Sorting algorithms

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

Outline

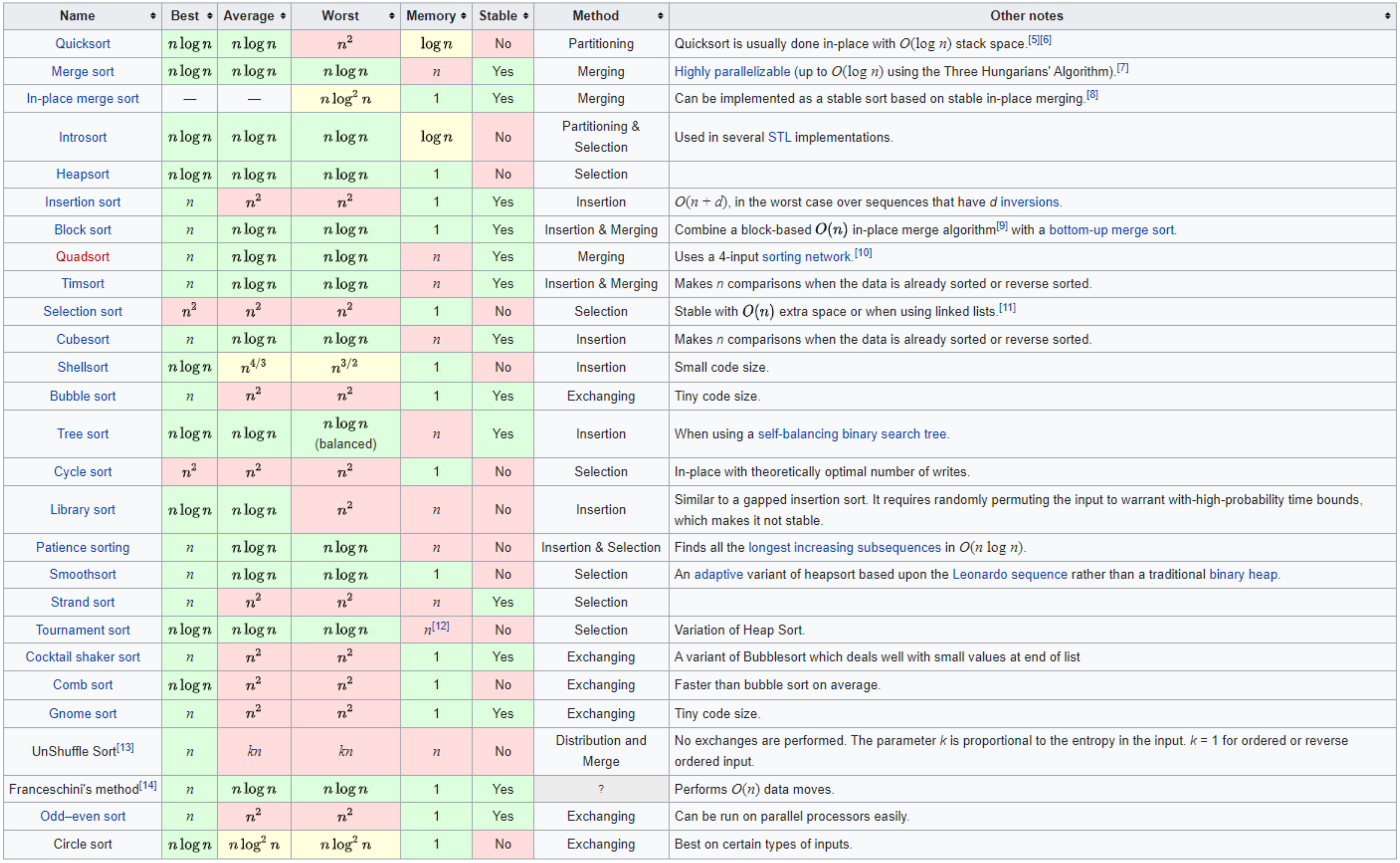
- Illustrating the midterm exam questions

- Comparison-based sorting algorithms
  - QuickSort

The sorting problem

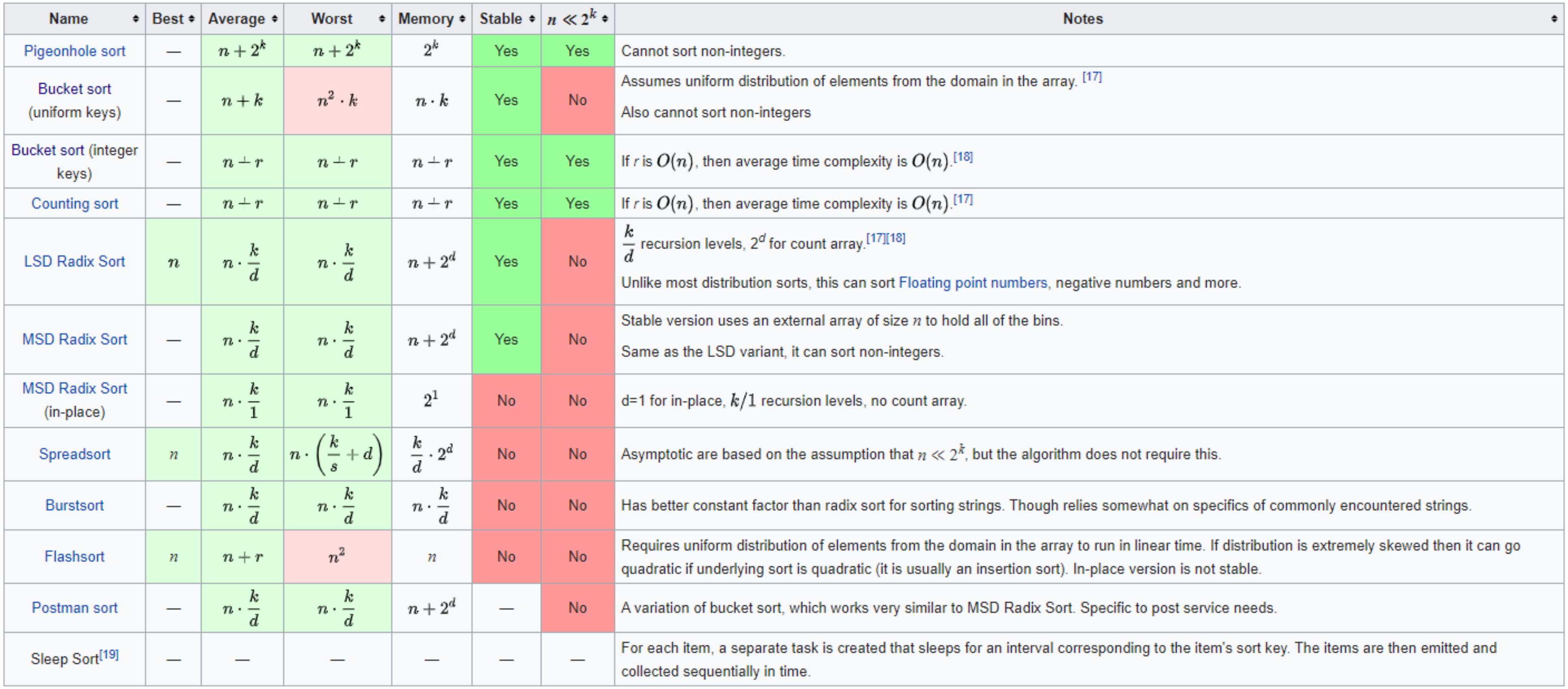
- Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- Output: a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of input such that $a'_1 <= a'_2 <= \ldots <= a'_n \rangle$
  - Stored in arrays
  - The numbers are referred as keys
  - Associated with additional information (satellite data)
  - Several ways to solve

Comparison sorts



Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n\log n$	$n\log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n\log n$	$n\log n$	$n\log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
In-place merge sort	—	—	$n\log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Introsort	$n\log n$	$n\log n$	$n\log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n\log n$	$n\log n$	$n\log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Block sort	n	$n\log n$	$n\log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort.
Quadsort	n	$n\log n$	$n\log n$	n	Yes	Merging	Uses a 4-input sorting network.[10]
Timsort	n	$n\log n$	$n\log n$	n	Yes	Insertion & Merging	Makes n comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[11]
Cubesort	n	$n\log n$	$n\log n$	n	Yes	Insertion	Makes n comparisons when the data is already sorted or reverse sorted.
Shellsort	$n\log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Tree sort	$n\log n$	$n\log n$	$n\log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Selection	In-place with theoretically optimal number of writes.
Library sort	$n\log n$	$n\log n$	n^2	n	No	Insertion	Similar to a gapped insertion sort. It requires randomly permuting the input to warrant with-high-probability time bounds, which makes it not stable.
Patience sorting	n	$n\log n$	$n\log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n\log n$	$n\log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n\log n$	$n\log n$	$n\log n$	n[12]	No	Selection	Variation of Heap Sort.
Cocktail shaker sort	n	n^2	n^2	1	Yes	Exchanging	A variant of Bubblesort which deals well with small values at end of list
Comb sort	$n\log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
UnShuffle Sort[13]	n	kn	kn	n	No	Distribution and Merge	No exchanges are performed. The parameter k is proportional to the entropy in the input. k = 1 for ordered or reverse ordered input.
Franceschini's method[14]	n	$n\log n$	$n\log n$	1	Yes	?	Performs $O(n)$ data moves.
Odd–even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.
Circle sort	$n\log n$	$n\log^2 n$	$n\log^2 n$	1	No	Exchanging	Best on certain types of inputs.

Non-comparison sorts



Name	Best	Average	Worst	Memory	Stable	$n \ll 2^k$	Notes
Pigeonhole sort	—	$n + 2^k$	$n + 2^k$	2^k	Yes	Yes	Cannot sort non-integers.
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No	Assumes uniform distribution of elements from the domain in the array. [17] Also cannot sort non-integers
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes	If r is $O(n)$, then average time complexity is $O(n)$.[18]
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes	If r is $O(n)$, then average time complexity is $O(n)$.[17]
LSD Radix Sort	n	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	Yes	No	$\frac{k}{d}$ recursion levels, 2^d for count array.[17][18] Unlike most distribution sorts, this can sort Floating point numbers, negative numbers and more.
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	Yes	No	Stable version uses an external array of size n to hold all of the bins. Same as the LSD variant, it can sort non-integers.
MSD Radix Sort (in-place)	—	$n \cdot \frac{k}{1}$	$n \cdot \frac{k}{1}$	2^1	No	No	d=1 for in-place, $k/1$ recursion levels, no count array.
Spreadsort	n	$n \cdot \frac{k}{d}$	$n \cdot \left(\frac{k}{s} + d\right)$	$\frac{k}{d} \cdot 2^d$	No	No	Asymptotic are based on the assumption that $n \ll 2^k$, but the algorithm does not require this.
Burstsort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	No	No	Has better constant factor than radix sort for sorting strings. Though relies somewhat on specifics of commonly encountered strings.
Flashsort	n	$n + r$	n^2	n	No	No	Requires uniform distribution of elements from the domain in the array to run in linear time. If distribution is extremely skewed then it can go quadratic if underlying sort is quadratic (it is usually an insertion sort). In-place version is not stable.
Postman sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	—	No	A variation of bucket sort, which works very similar to MSD Radix Sort. Specific to post service needs.
Sleep Sort[19]	—	—	—	—	—	—	For each item, a separate task is created that sleeps for an interval corresponding to the item's sort key. The items are then emitted and collected sequentially in time.

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 execute 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
  - $\Pr[X = 2] = \frac{1}{2}$
  - $\Pr[X = 4] = \frac{1}{4}$
  - …
  - $\Pr[X = 2i] = \frac{1}{2^i}$
  - $E[X] = 2 \cdot \sum_{i=1}^{+\infty} \frac{i}{2^i} = 4 = O(1)$

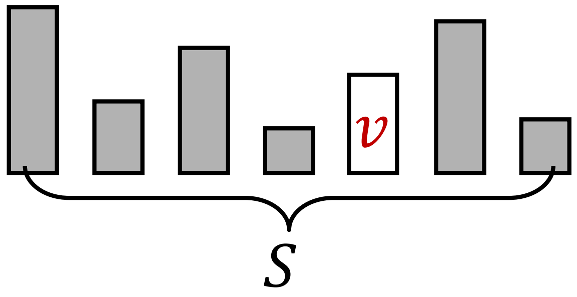
Algorithm: ***flipCoin()***

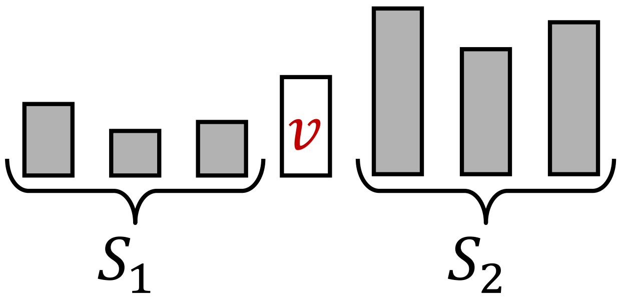
```
r ← RANDOM(0,1)
while r != 1
    r = RANDOM(0,1)
```

QuickSort (Divide-and-Conquer)

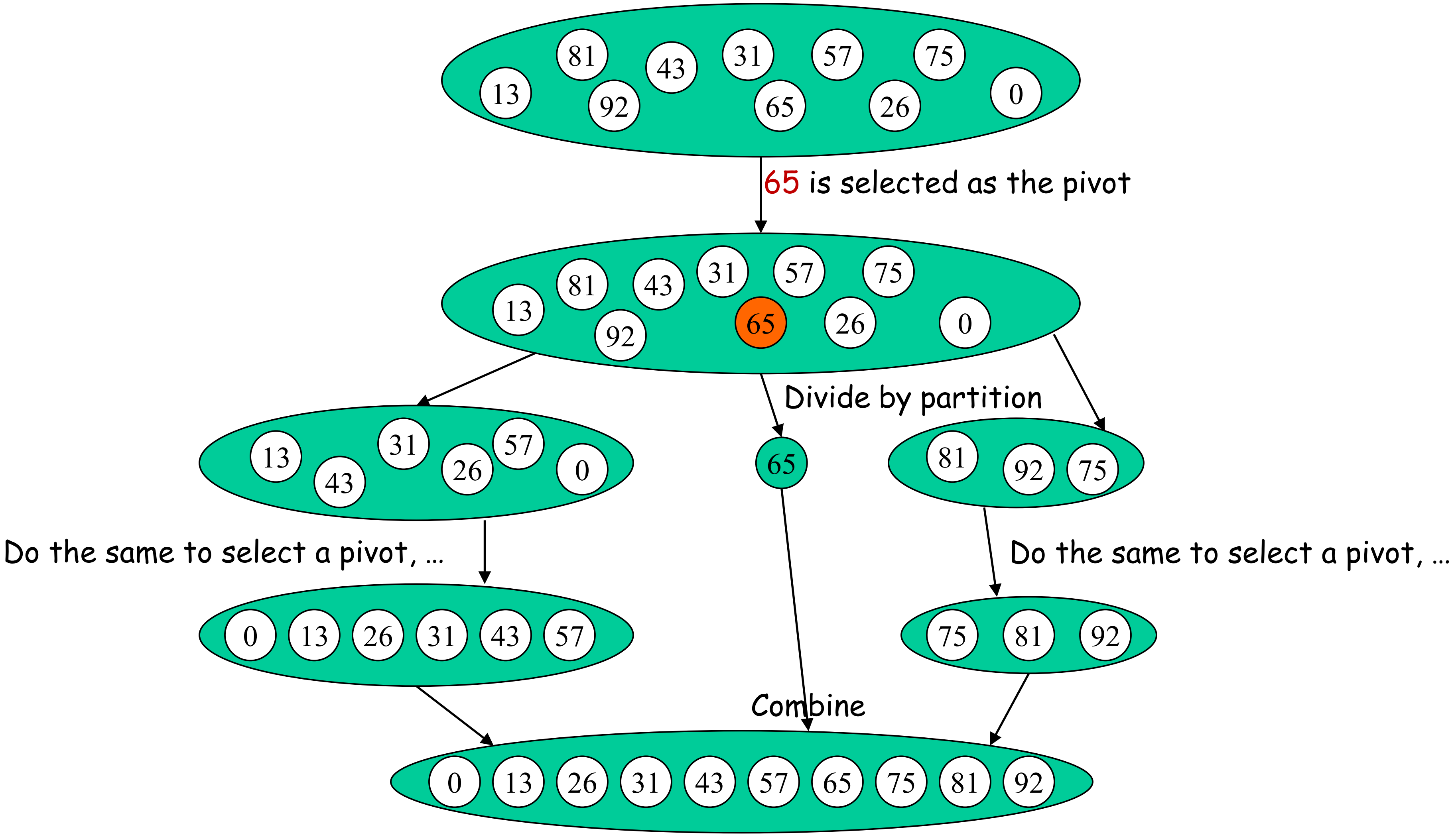
- A randomized algorithm using divide-and-conquer
  - Time complexity: $O(n \cdot \log n)$ expected running time

- High level idea: (we 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





A 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**: the input array, the left position and right position of the array, and the randomly selected pivot position
  - **Goal**: divide the array into three parts: the left partition (smaller than the pivot element), pivot position, and the right partition (larger than the 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

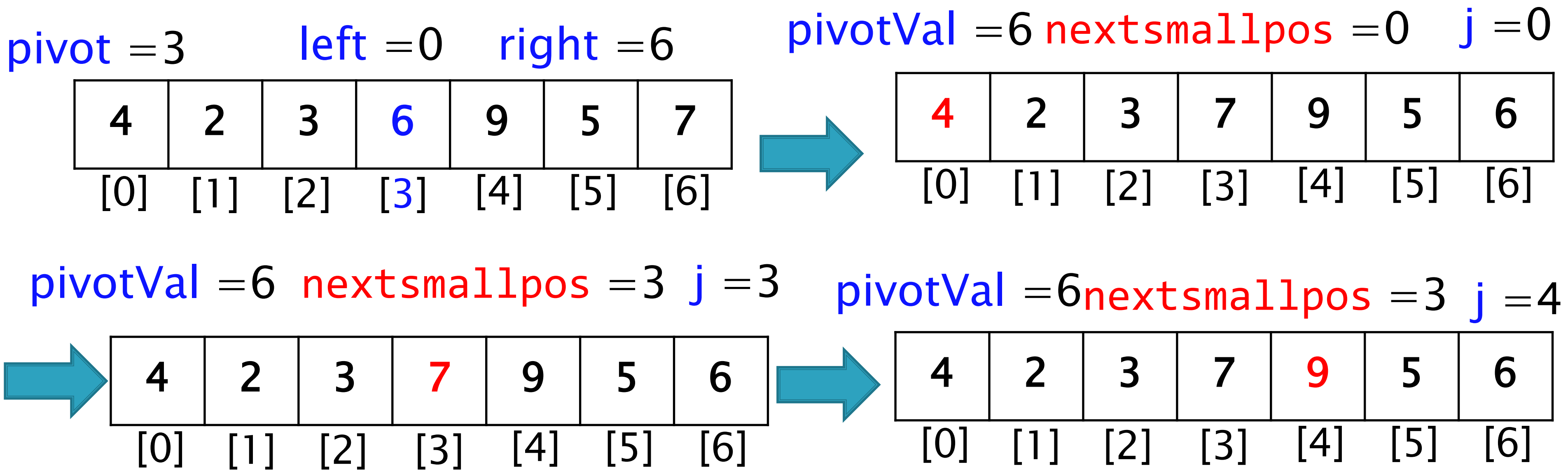
pivotNewposition =4

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

QuickSort: Partition

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

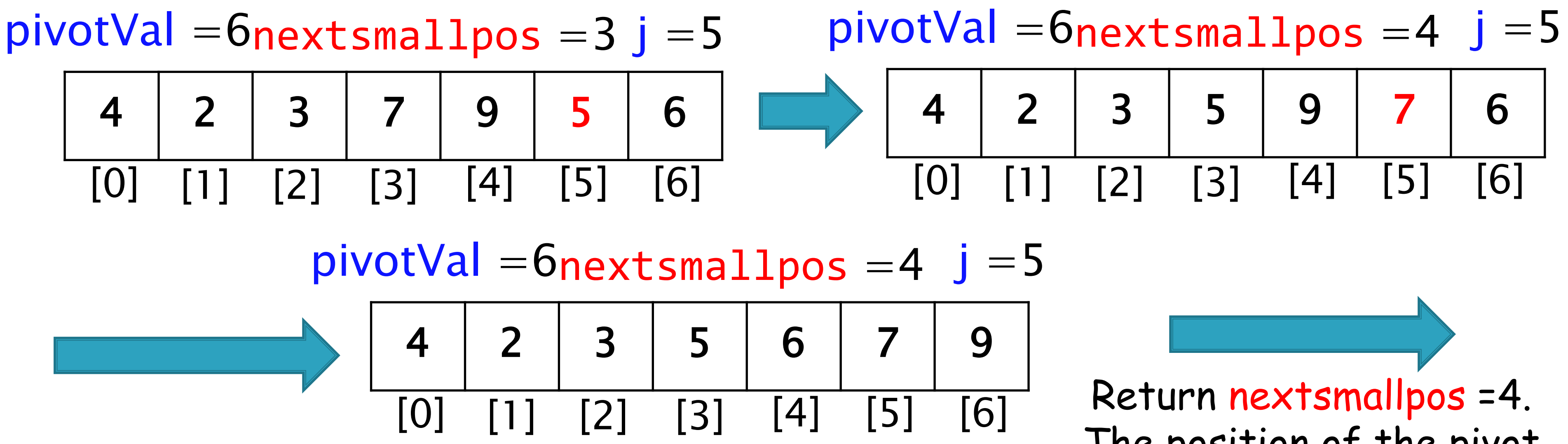
```
pivotVal=arr[pivot] //record the pivot data
arr[right] ↔ arr[pivot] //swap the pivot data and the last data
nextsmallpos=left//record the next position to put data smaller than pivotVal
for j from left to right-1
    if arr[j] < pivotVal
       arr[nextsmallpos] ↔ arr[j]
       nextsmallpos++
arr[nextsmallpos]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



QuickSort: Partition

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

```
pivotVal=arr[pivot] //record the pivot data
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
        arr[nextsmallpos++] ↔ arr[j]
arr[nextsmallpos] ↔ arr[right]
return nextsmallpos
```



Return nextsmallpos =4.
The position of the pivot

Practice

- Let $A[0 \dots 5] = [6,3,5,2,7,1]$. Assume that we call $quicksort(A, 0,5)$ and the pivot position we randomly choose is 2

- Show how the partition works and indicate the value of nextsmallpos during the invocation of $partition(A, 0, 5, 2)$ step by step

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.

- 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>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) {
        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: $\underline{8}$ 1 4 9 $\underline{6}$ 3 5 2 7 $\underline{0}$

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

  Start: 0 1 4 9 7 3 5 2 $\textcircled{\underline{6}}$ 8

  i $\rightarrow$ if smaller   if bigger $\leftarrow$ *j*

  Move *i*: 0 1 4 9 7 3 5 2 $\textcircled{\underline{6}}$ 8

  i under 9, *j* under 2

  Move *j*: 0 1 4 9 7 3 5 2 $\textcircled{\underline{6}}$ 8

  i under 9, *j* under 2

QuickSort -median3 example

1st swap:	0	1	4	2	7	3	5	9	$\underline{6}$	8
				i				*j*		
Move *i*:	0	1	4	2	7	3	5	9	$\underline{6}$	8
					i			*j*		
Move *j*:	0	1	4	2	7	3	5	9	$\underline{6}$	8
					i		*j*			

QuickSort – median3 example

2nd swap: 0 1 4 2 5 3 7 9 $\underline{6}$ 8

i *j*

Move *i*: 0 1 4 2 5 3 7 9 $\underline{6}$ 8

i *meet j*

Move *j*: 0 1 4 2 5 3 7 9 $\underline{6}$ 8

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

Swap element at *i* with pivot

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

Exercise

- Use QuickSort to sort the following sequence of integer values

  5,7,3,6,4,1,0,2,9,8

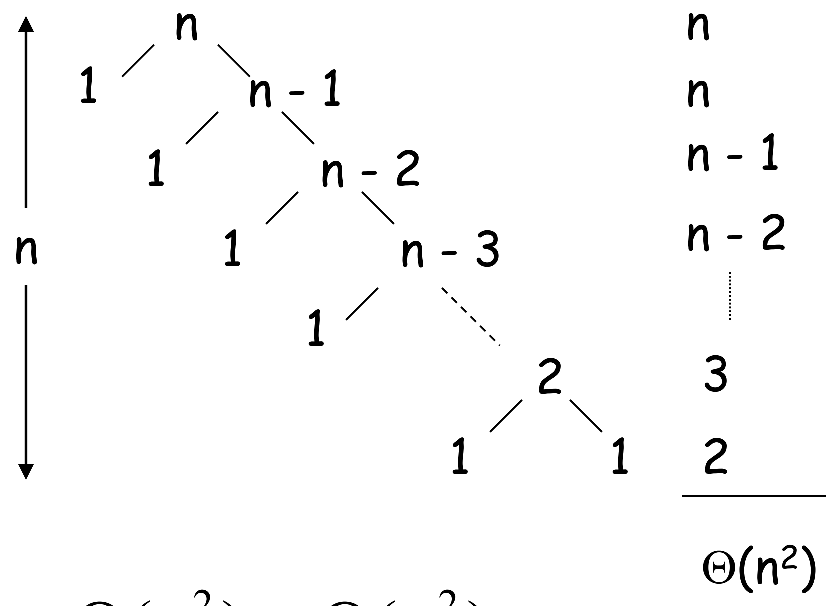
Worst Case Partitioning

- 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) = Θ(1)

  T(n) = T(n – 1) + n



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

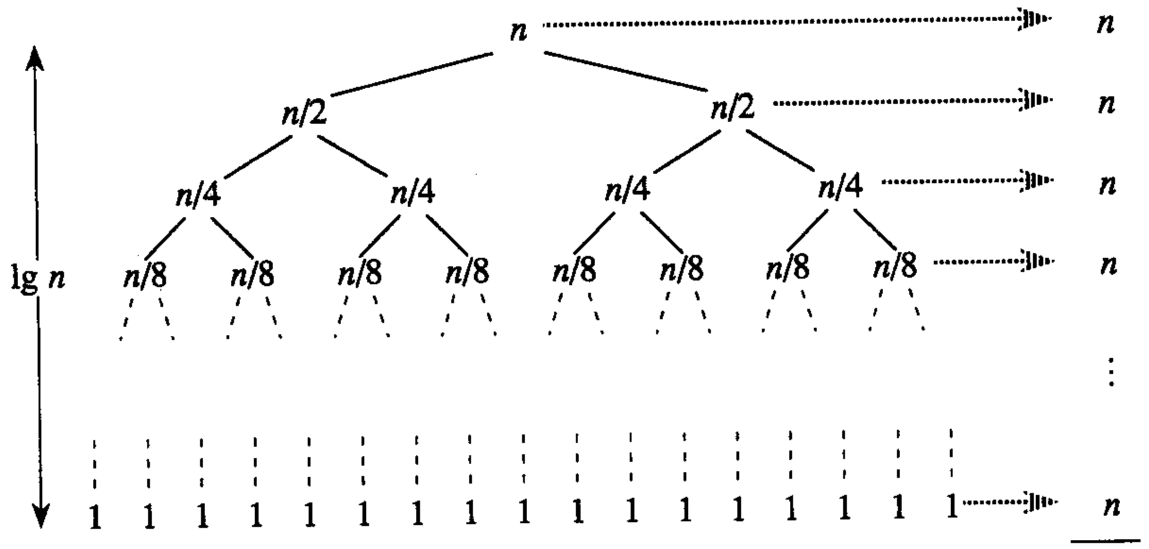
When does the worst case happen?

Best Case Partitioning

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

  $T(n) = 2T(n/2) + \Theta(n)$

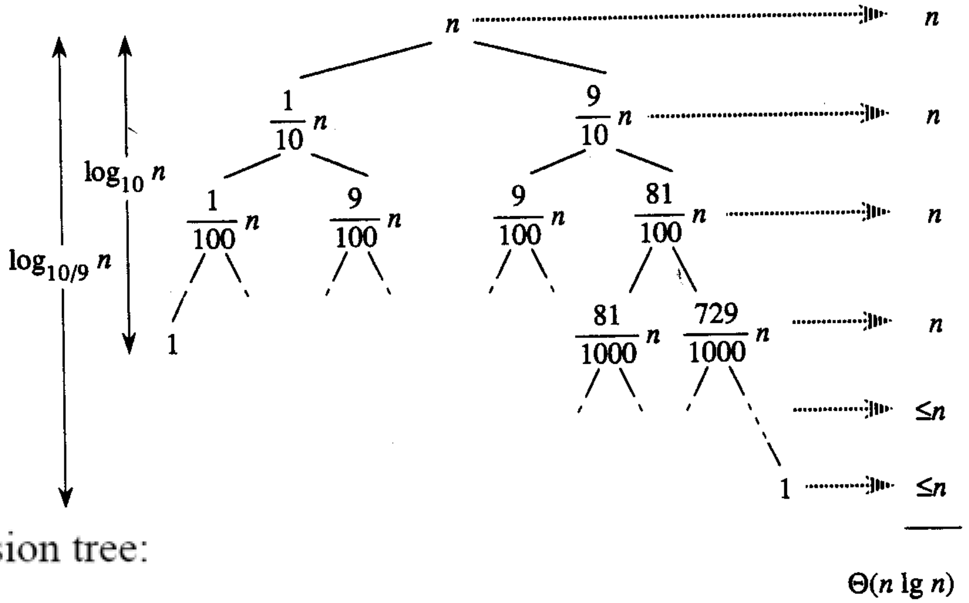
  $T(n) = \Theta(n\lg n)$



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: $Q(n) \leq n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 n lgn$

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

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

QuickSort

- Worst case of QuickSort

$$T(N) = T(1) + c\sum_{i=2}^{N} i = O\left(N^2\right)$$

- Best case

$$T(N) = c\,N \log N + N = O\,(N \log N)$$

- Average casae

$$T(N) = O\,(N \log N)$$

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 $\Pr[X_{i,j} = 1]$?
    - $\Pr[X_{i,j} = 1] = \frac{2}{j-i+1}$

- 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 number of comparisons in quicksort. Then, $X = \sum_{1\le i<j\le n} X_{i,j}$.
  - Therefore $E[X] = \mathrm{E}[\sum_{1\le i<j\le n} X_{i,j}] = \sum_{1\le i<j\le n} \frac{2}{j-i+1}$
  - 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 $O(n \cdot \log n)$.