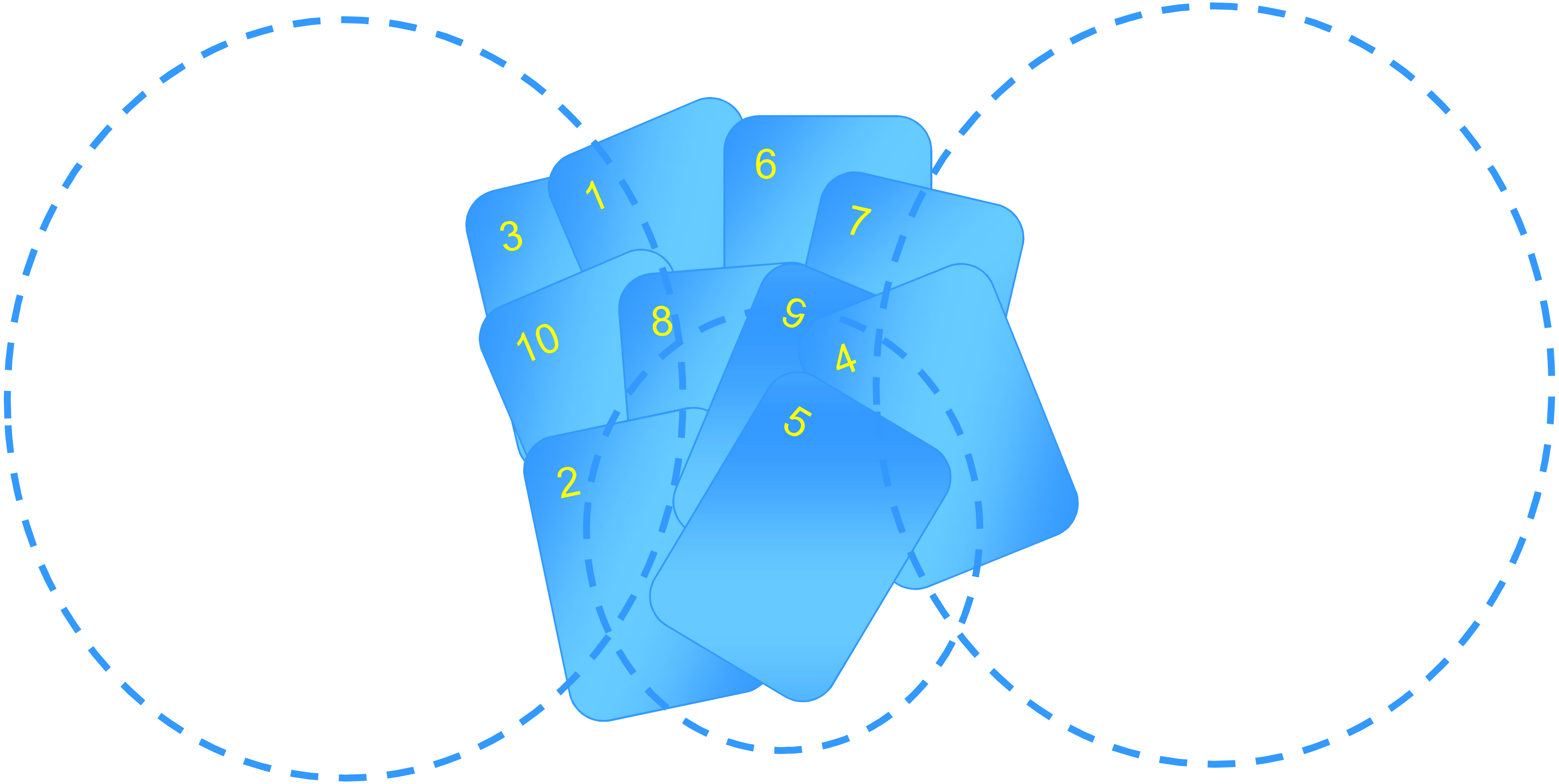
Quicksort

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

Space Complexity: $O(\log n)$

Quicksort



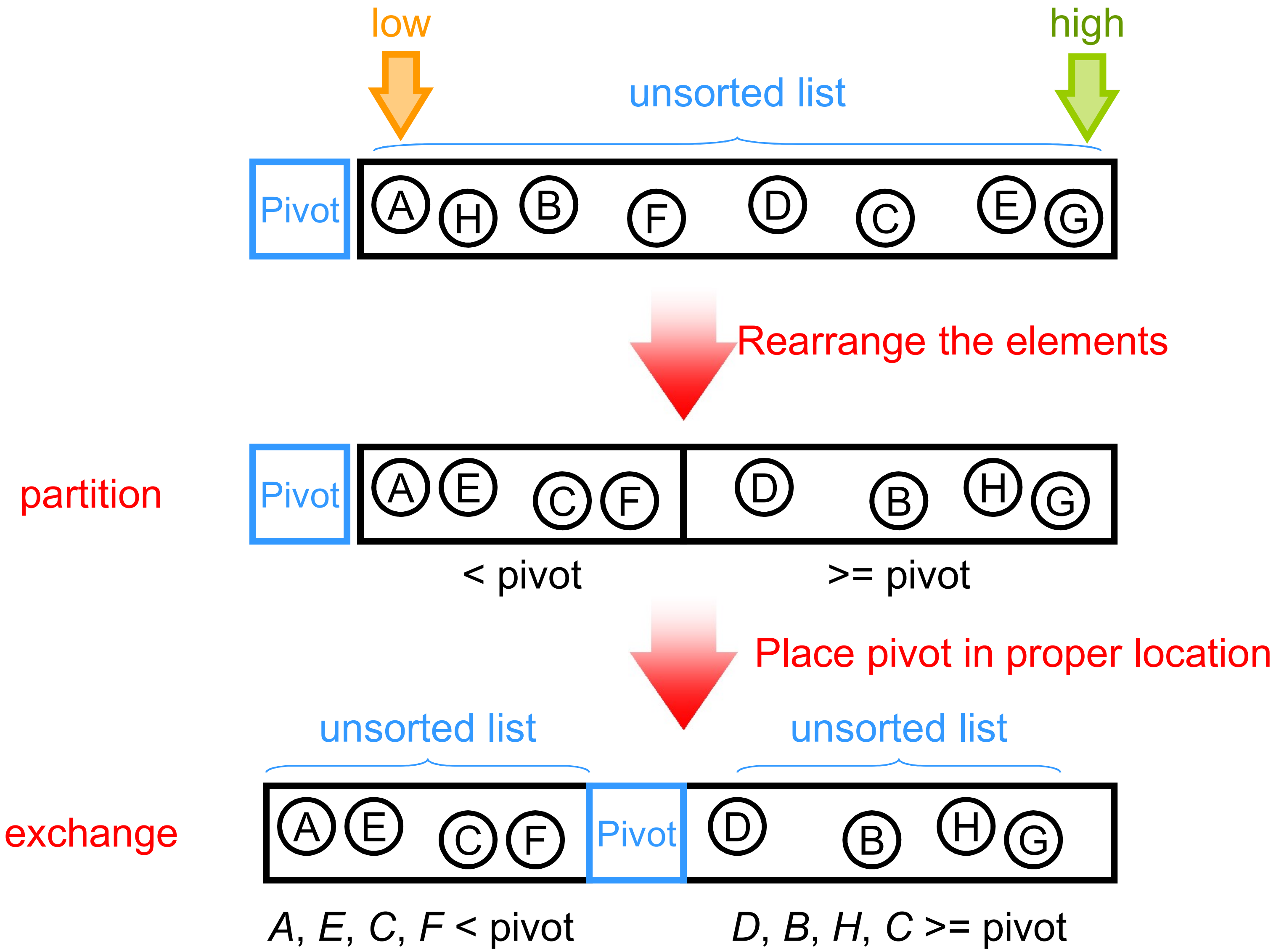
Left < pivot ≤ Right

(recursion) (recursion)

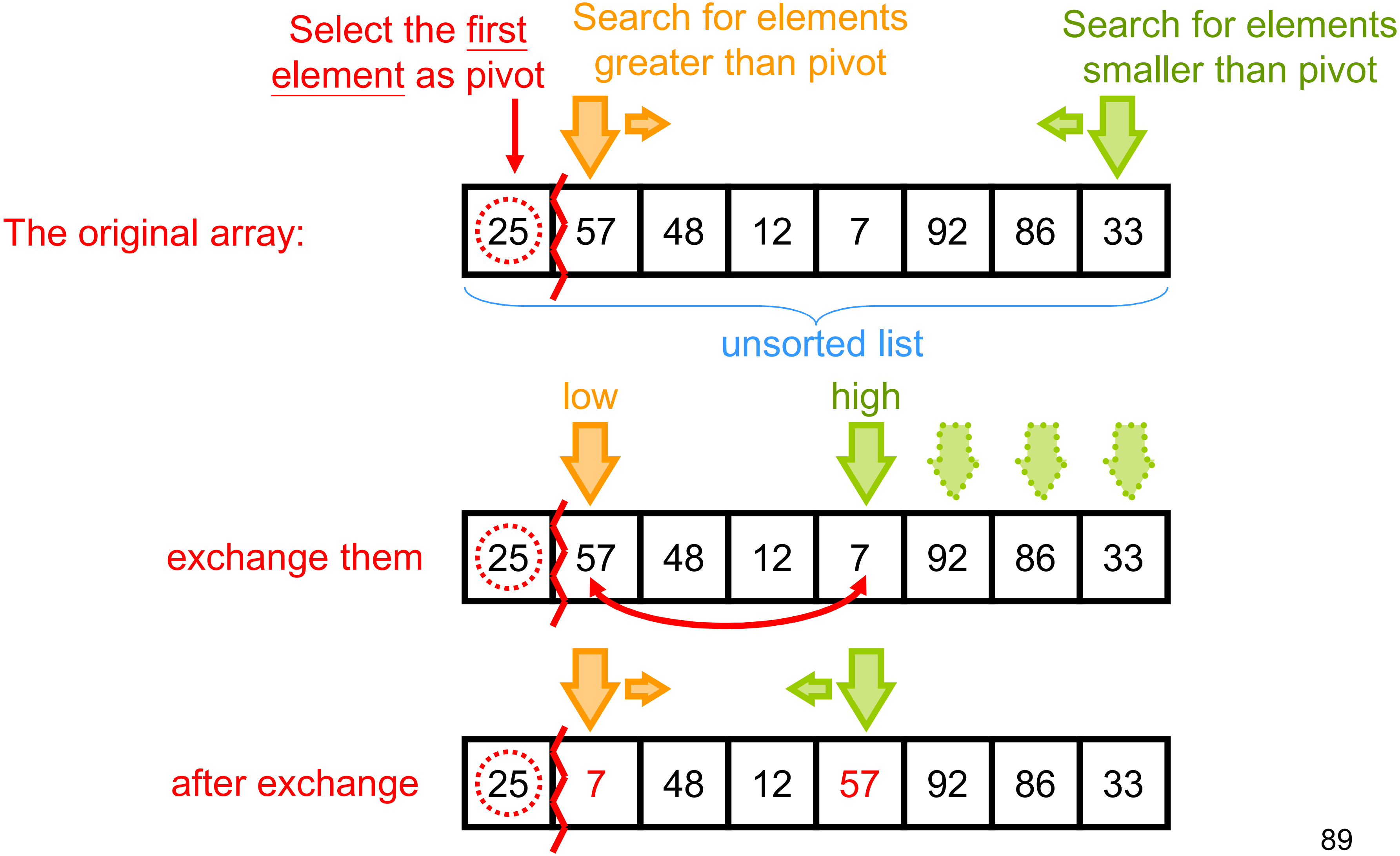
Exchange and Partition

- A.K.A. partition-exchange sort
  - Step 1) Exchange, then Step 2) Partition
- If the list has one or no elements (base case)
  - Do nothing (as already sorted)
- If the list has two or more elements
  - Pick an element as the **pivot**
  - Place the elements **smaller** than the pivot **before** it and the elements **larger** than or equal to the pivot **after** it (in any order) (by iteration)
  - Sort the sublist before the pivot (by recursion)
  - Sort the sublist after the pivot (by recursion)

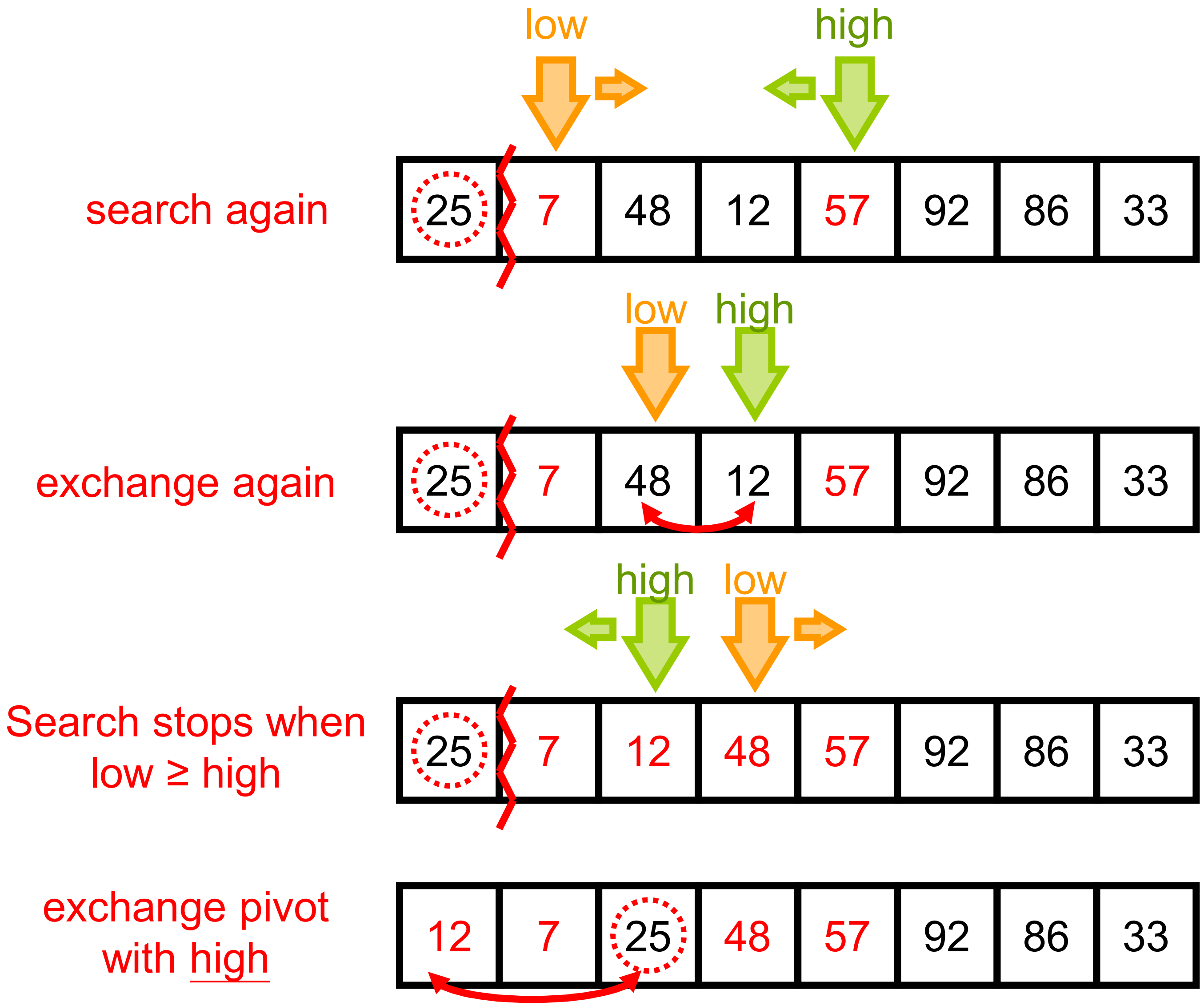
The General Concept



Quicksort Example

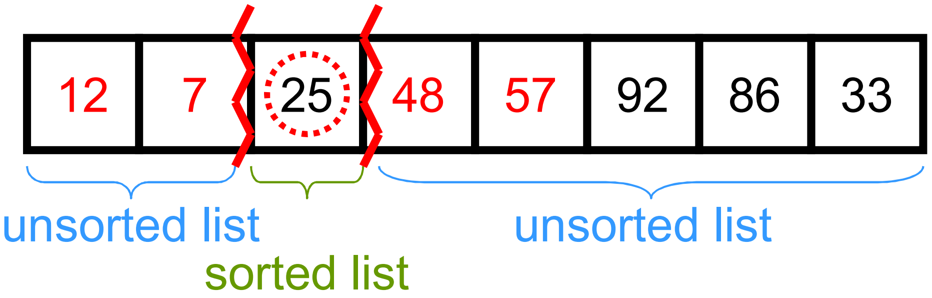


Quicksort Example



Quicksort Example

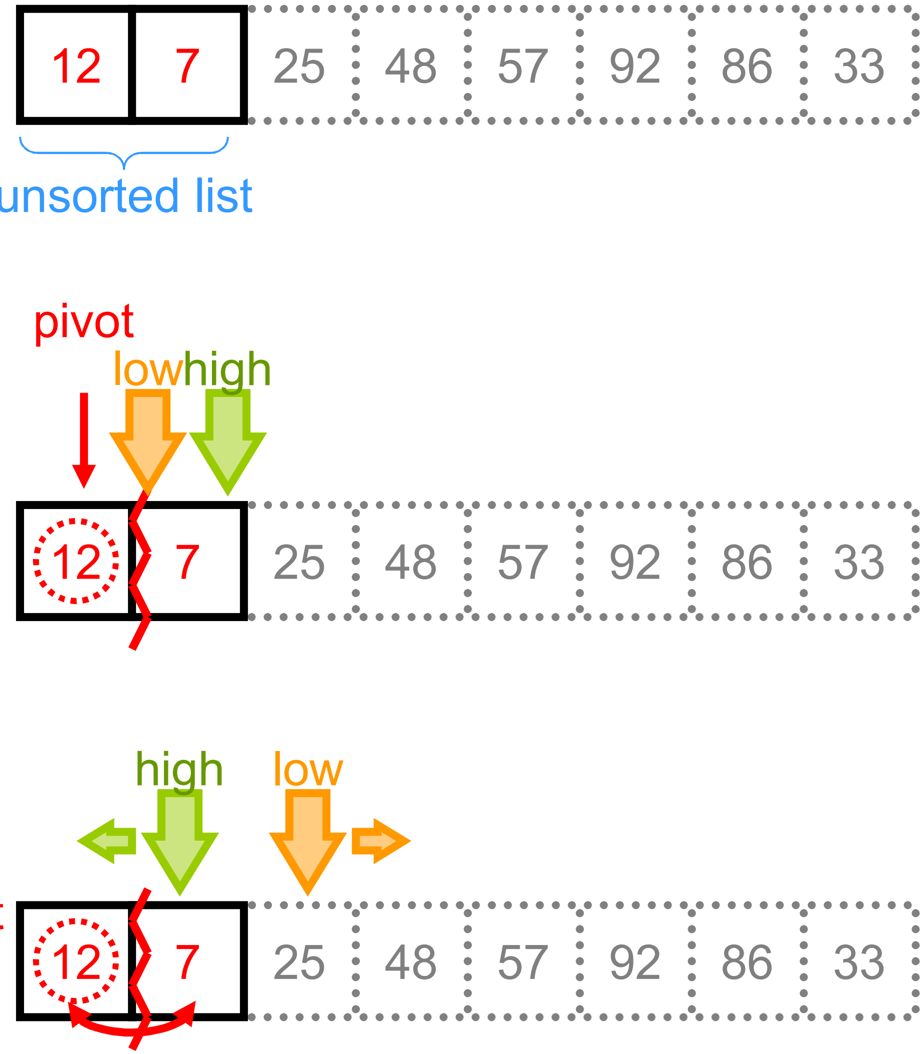
After 1st pass



$$\underbrace{\{12,\ 7\}}_{\text{sort recursively}} \quad < \quad 25 \quad \leq \quad \underbrace{\{48,\ 57,\ 92,\ 86,\ 33\}}_{\text{sort recursively}}$$

Sort the Left Sublist

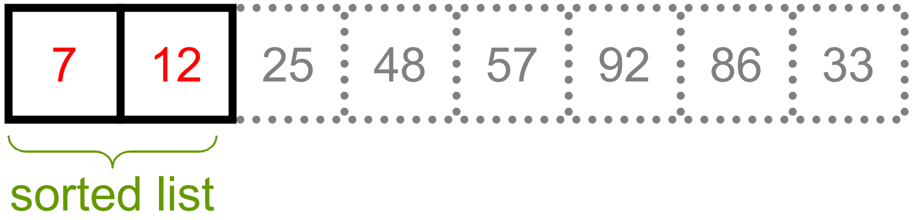
Sort the left sublist



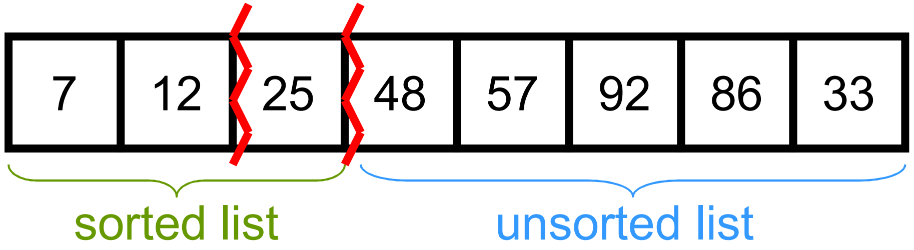
After searching, high will point to 7 (smaller than 12) and low will point out of the array

Sort the Left Sublist

Exchange pivot with high



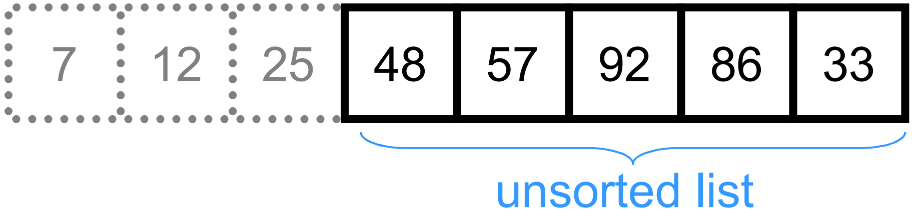
Combining the array



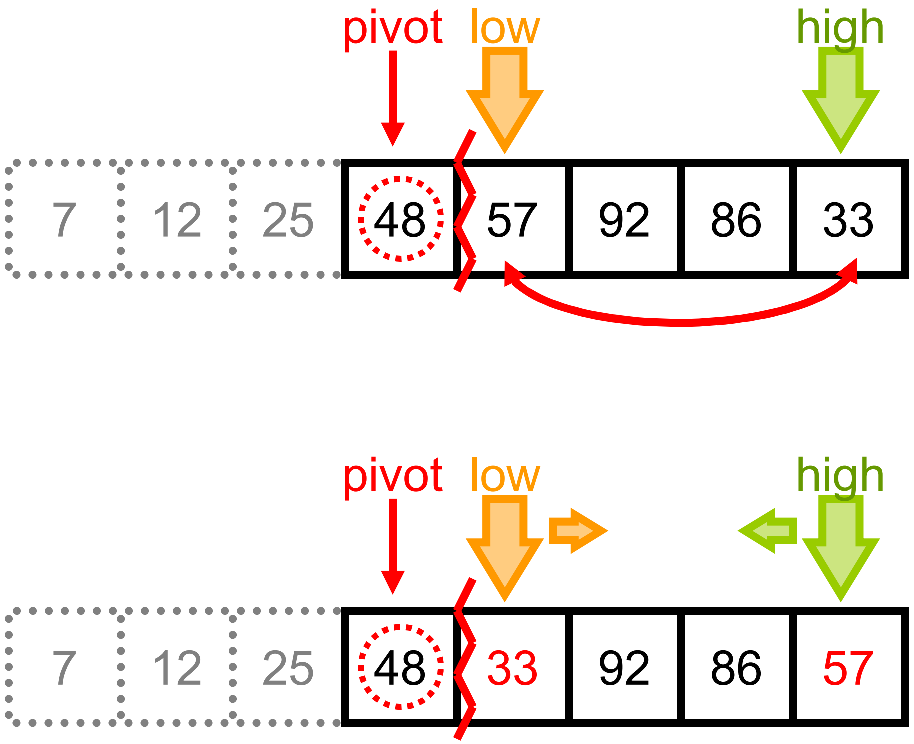
$$\{7, 12, 25\} \leq \underbrace{\{48, 57, 92, 86, 33\}}$$

sort recursively

Sort the Right Sublist

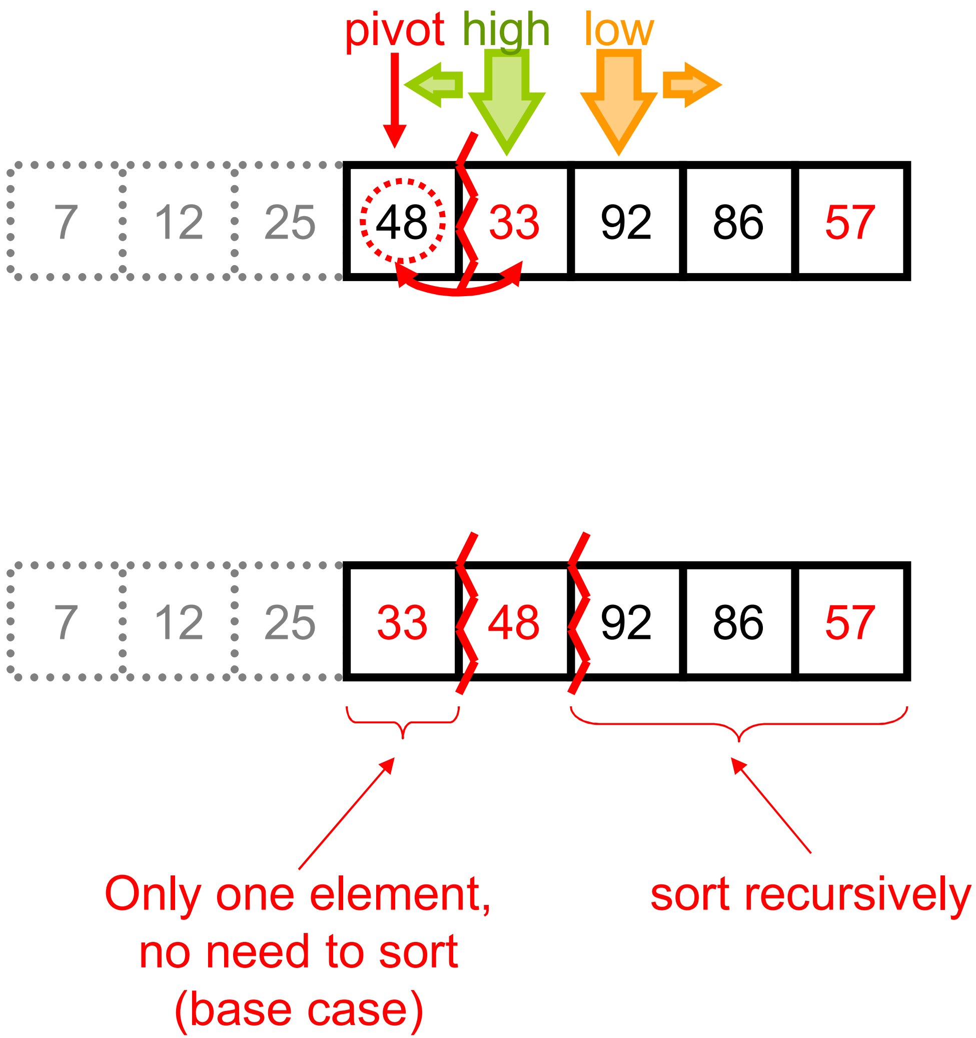


Search and exchange

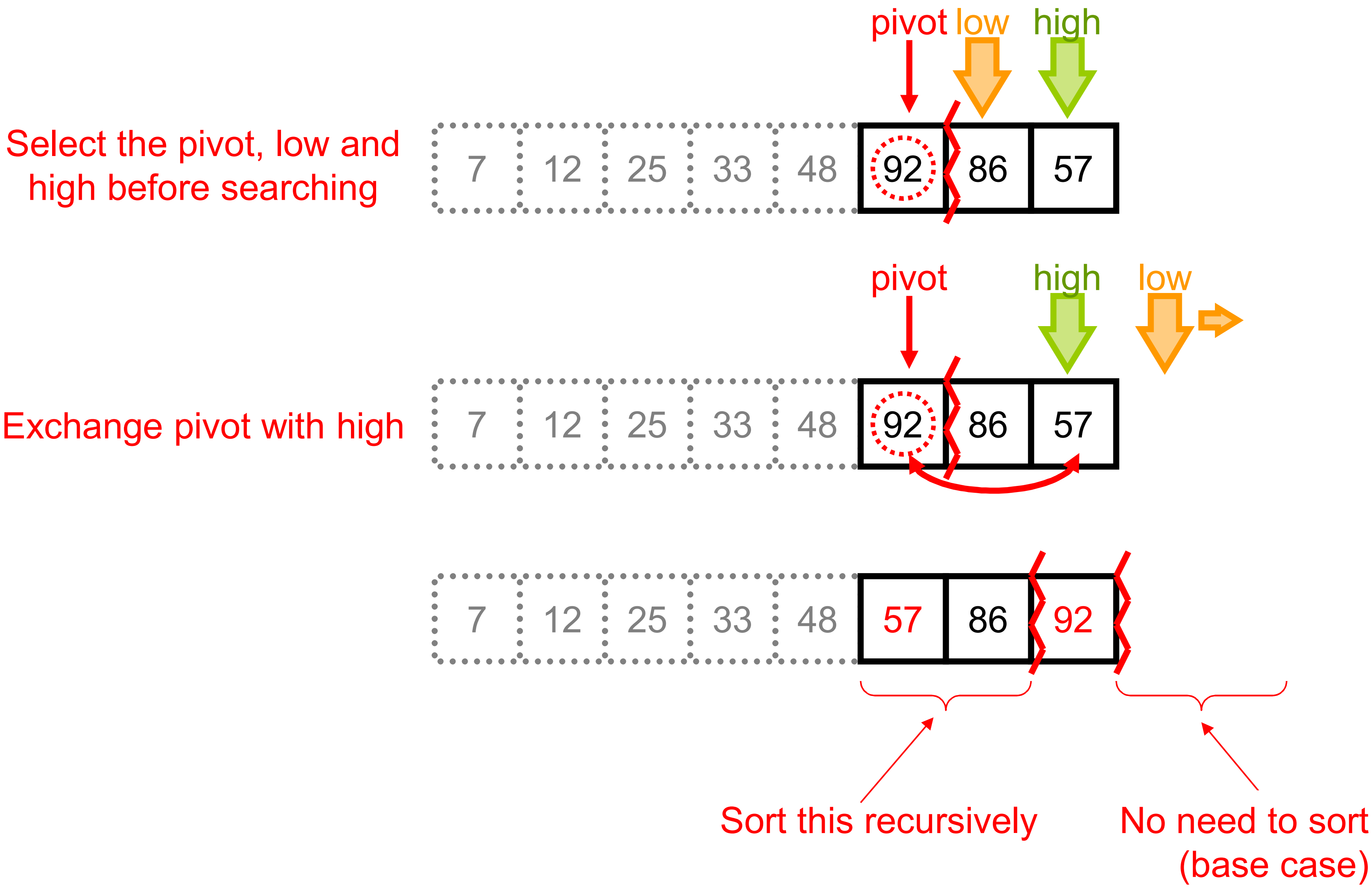


Sort the Right Sublist

Exchange pivot with high

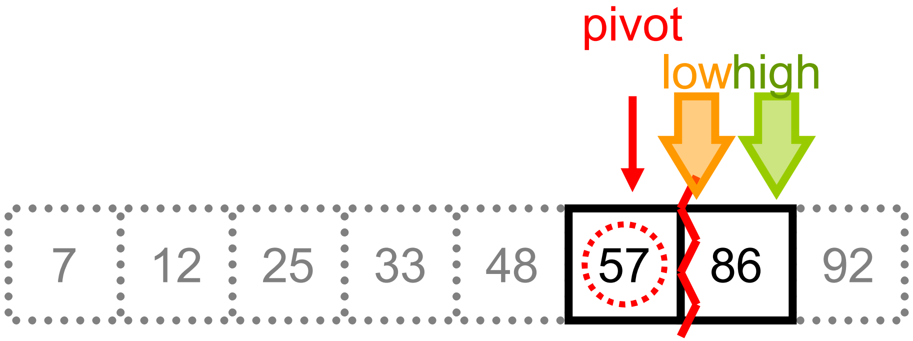


Sort Another Right Sublist

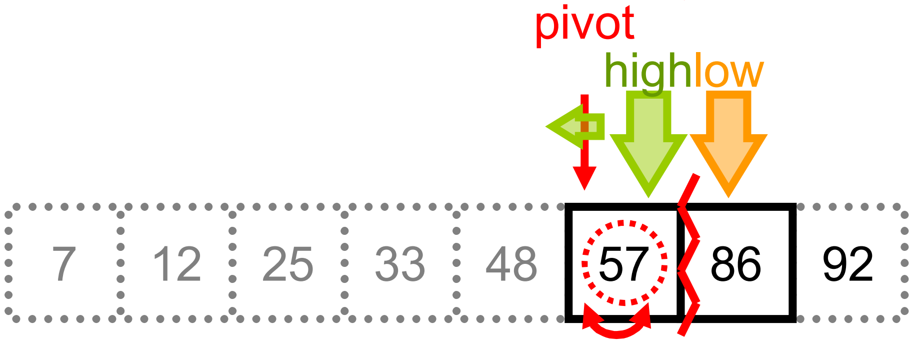


Sort the Last Sublist

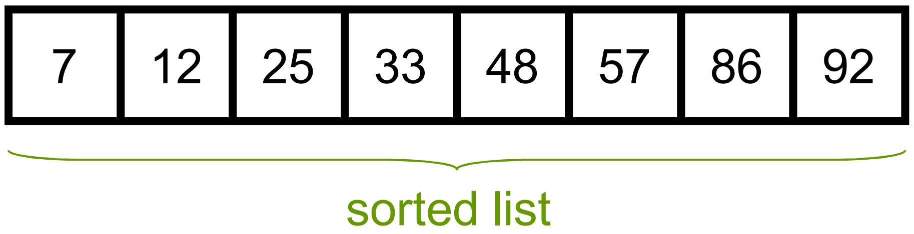
Select the pivot, low and high before searching



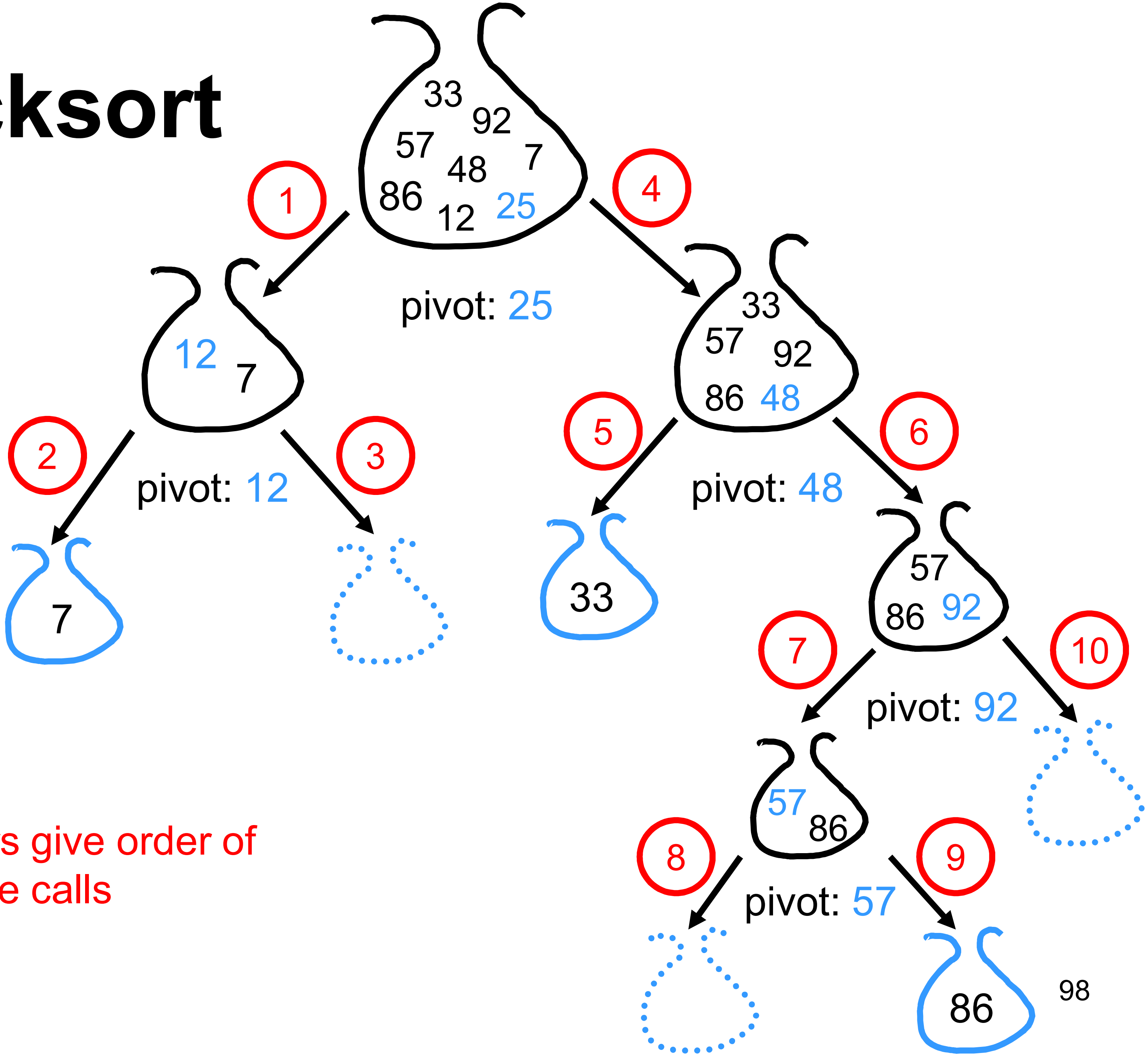
Exchange pivot with high (exchange with itself)



Finally, the list is sorted correctly



Quicksort



Numbers give order of recursive calls

Quicksort

- Divide-and-conquer sorting algorithm
- e.g. the unsorted array is data[*p*…*r*]
- Divide Stage
  - **Exchange** and **partition** the array data[] into **three** sub-arrays: data[*p*…*q*-1], data[*q*] and data[*q*+1…*r*] such that
  - All element in data[*p*…*q*-1] is less than data[q], and
  - All element in data[*q*+1…*r*] is greater than or equal to data[*q*]

Quicksort

- Conquer Stage
  - The two sub-arrays data[*p*…*q*-1] and data[*q*+1…*r*] are sorted recursively
- Combine Stage
  - The sub-arrays are sorted in place
  - No extra memory needed (except swapping)
  - No work is need to combine them

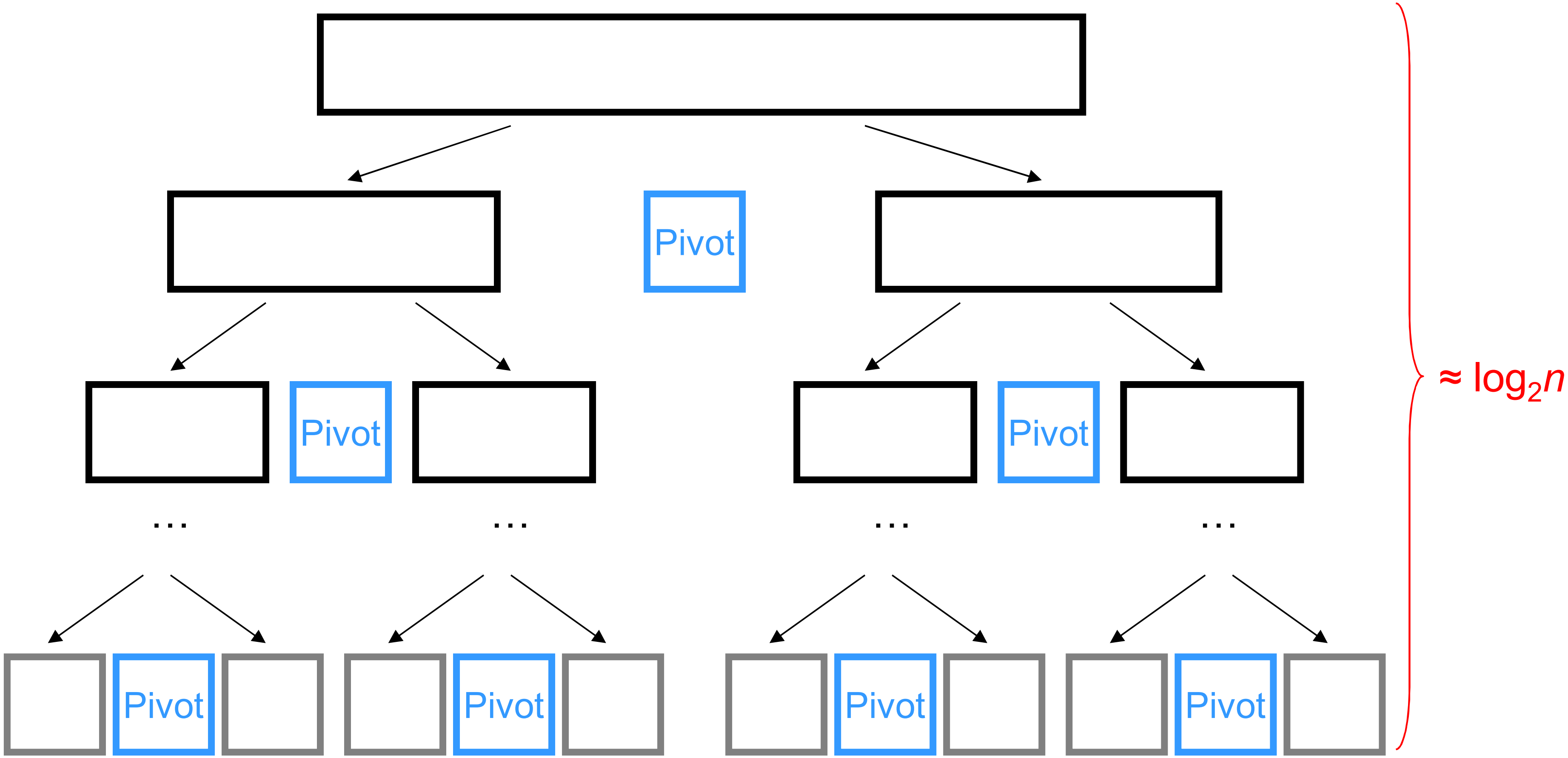
The Procedure

```
void quicksort(int data[], int p, int r) {  // p: start, r: end index
    int pivot, low, high, q;

    if (p >= r) return;  //base case

    pivot = p;              //set first element as pivot
    low = p + 1;
    high = r;

    while (low < high) {
        while(data[low] <= data[pivot] && low < r) low++;
        while(data[high] > data[pivot] && high > p) high--;
        if (low < high) swap(&data[low], &data[high]);
    }
    if (data[pivot] > data[high]) //swap pivot with high
        swap(&data[pivot], &data[high]);

    q = high;
    quicksort(data, p, q-1);
    quicksort(data, q+1, r);
}
```

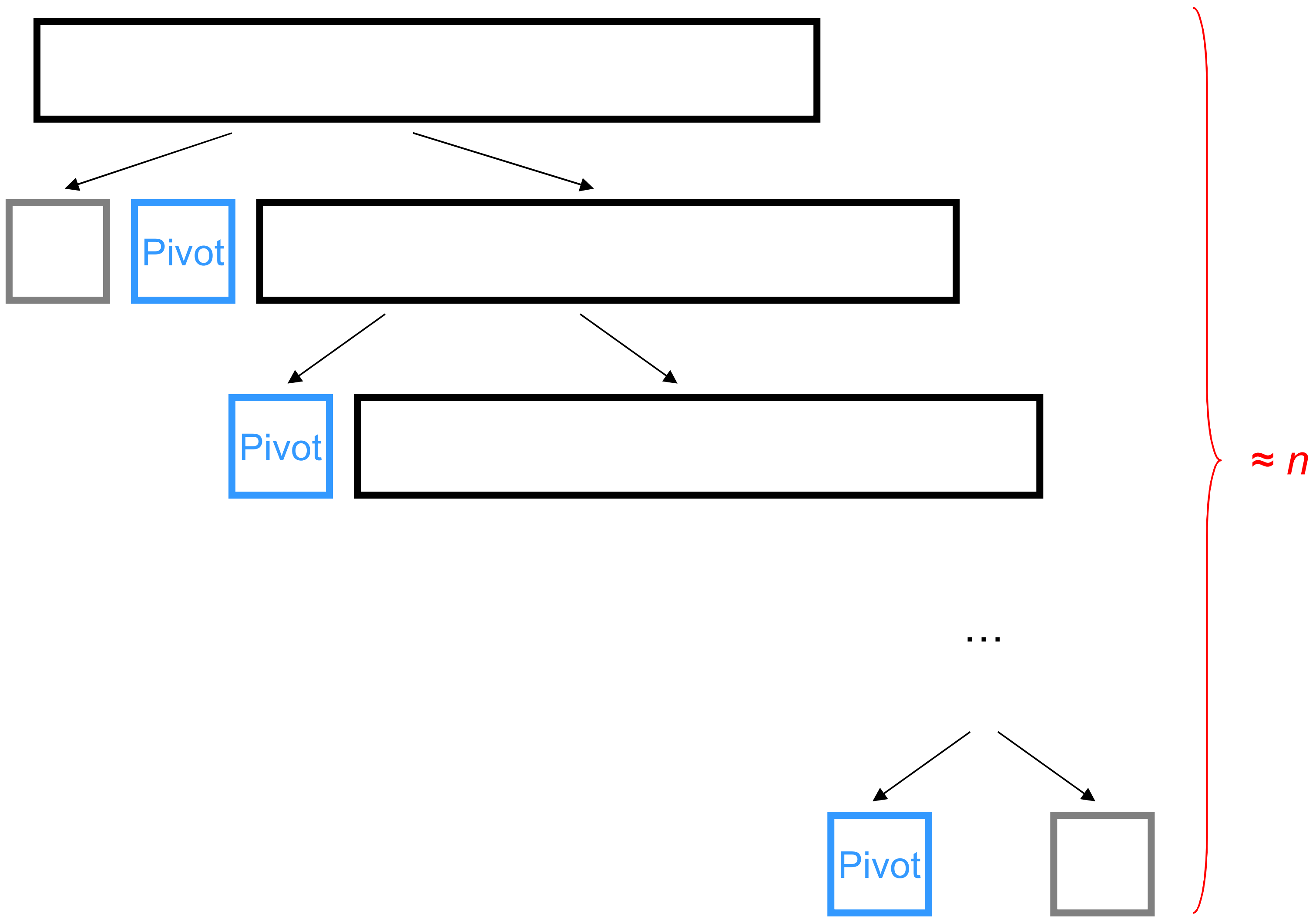
divide (exchange & partition) (iteration)

conquer (recursion)

A Good Pivot



A Bad Pivot



Complexity Analysis

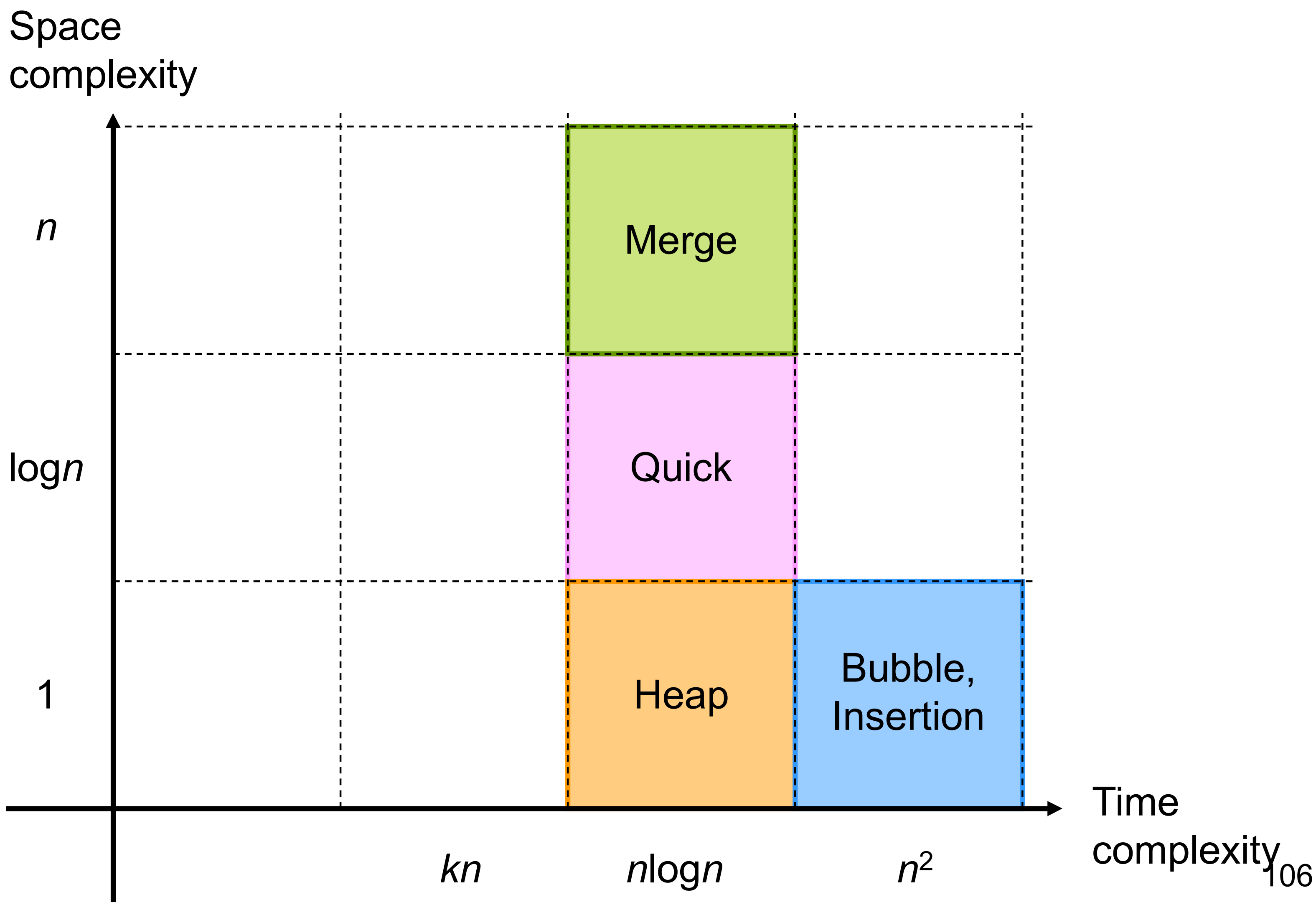
- Partition
  - Low pointer moves to right, while high pointer moves to left
  - Total $n - 1$ comparisons
  - $O(n)$: linear time
- Exchange
  - Swapping nodes: O(1)
- How many passes in total?
  - The best case
    - Ideally, the two sub-lists will be of equal size if the median is chosen as pivot in each pass
    - There will be about $\log_2 n$ passes
    - So total time complexity is $O(n \cdot \log n)$
  - The worst case
    - If one of the sub-arrays is always empty, or has only one element
    - Total no. of passes is about n
    - Then quicksort takes $O(n^2)$ time

Choosing a Good Pivot

- By choosing the pivot carefully, we can obtain $O(n \cdot \log n)$ time in the average case
- The simplest (poor) version
  - Choose the first element as pivot
  - If the list is already sorted, the time complexity would be $O(n^2)$
- Two better versions
  - Choose the pivot randomly in each pass, or
  - Select the median between first, last and middle element as pivot
  - These two solutions cannot completely avoid the worst case
  - It can also be shown that the average cast complexity of quicksort is approximately equal to $1.38\, n \log_2 n$

- If the size of the array is large, quicksort is the fastest sorting method known today.

Summary


Space complexity
n
logn
1
Merge
Quick
Heap
Bubble,
Insertion
Time complexity
kn
nlogn
n2

Radix Sort

Time Complexity: O($k \cdot n$)

Space Complexity: O(n)

Sorting Model

- The sorting algorithms introduced so far are based on a comparison model where elements are compared to determine their relative order.
- It has been proven that this kind of algorithms require at least O($n$log$n$)
- Can we sort better without doing comparison?

Radix Sort

- What if every element can be represented by *k* digits with positional notation?
  - Consider one digit at a time, LSD first (the right most digit)
  - Divide the list into *r* sublists based on the digit, where *r* is the radix of a digit
    - 10 for decimal number; 2 for binary number
  - Consider another digit in the next pass until finally the list is completely sorted with totally *k* passes

  - Another name: bucket sort
  - A very great algorithm! Can sort data in almost <u>linear time</u>

Sorting using Queues

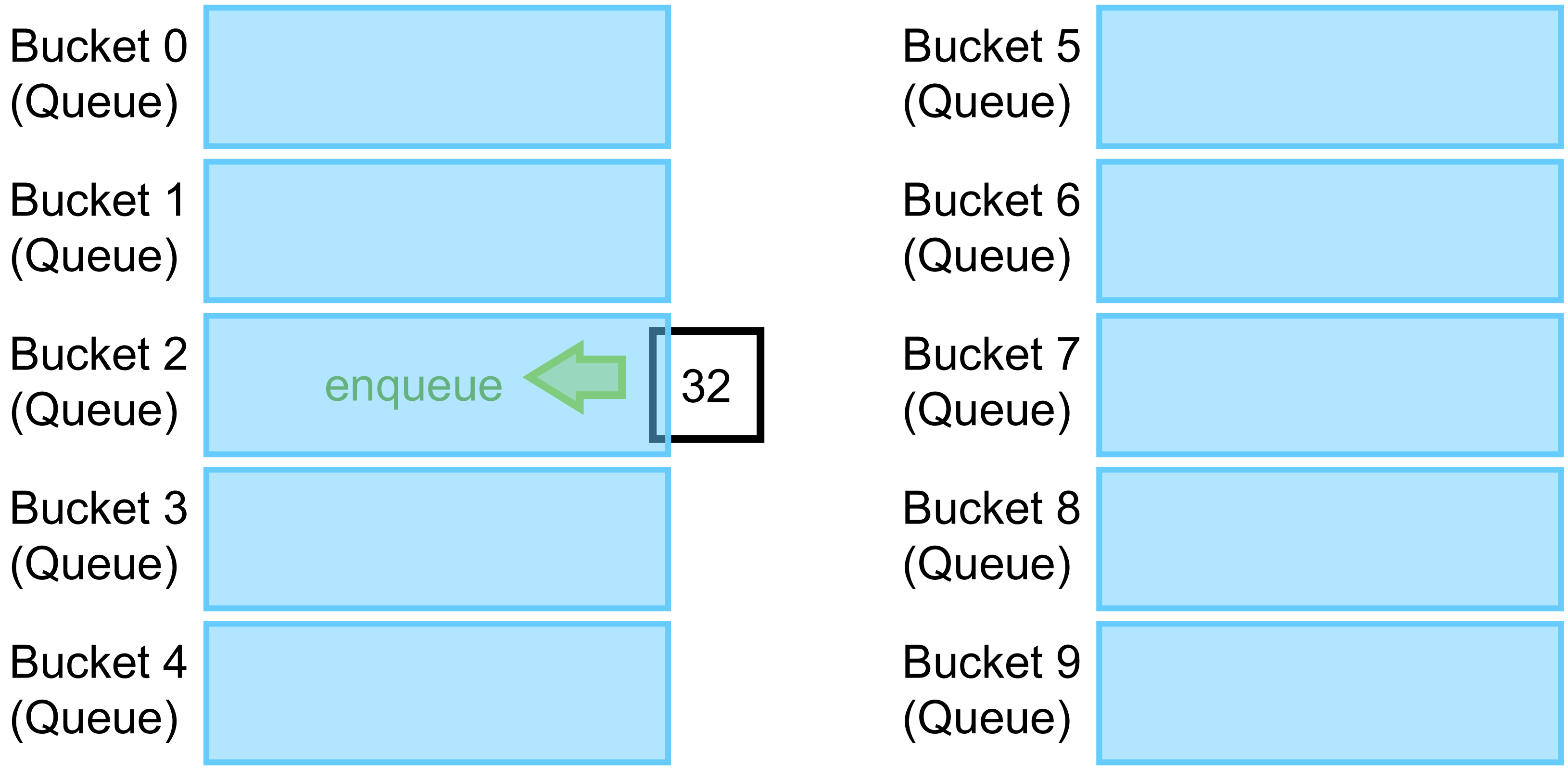
Radix sort

Implement Radix Sort Using Queues

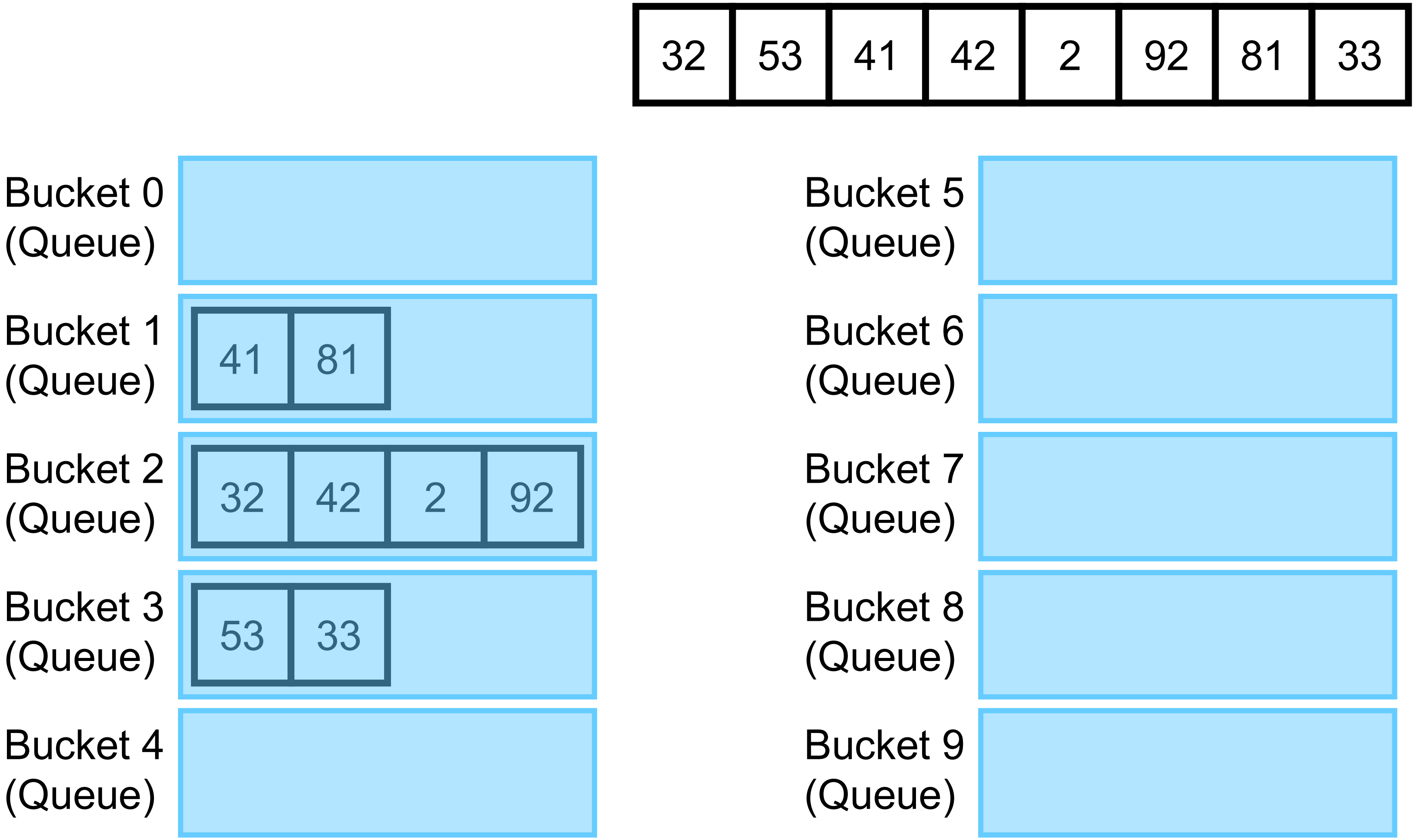
unsorted list

32	53	41	42	2	92	81	33

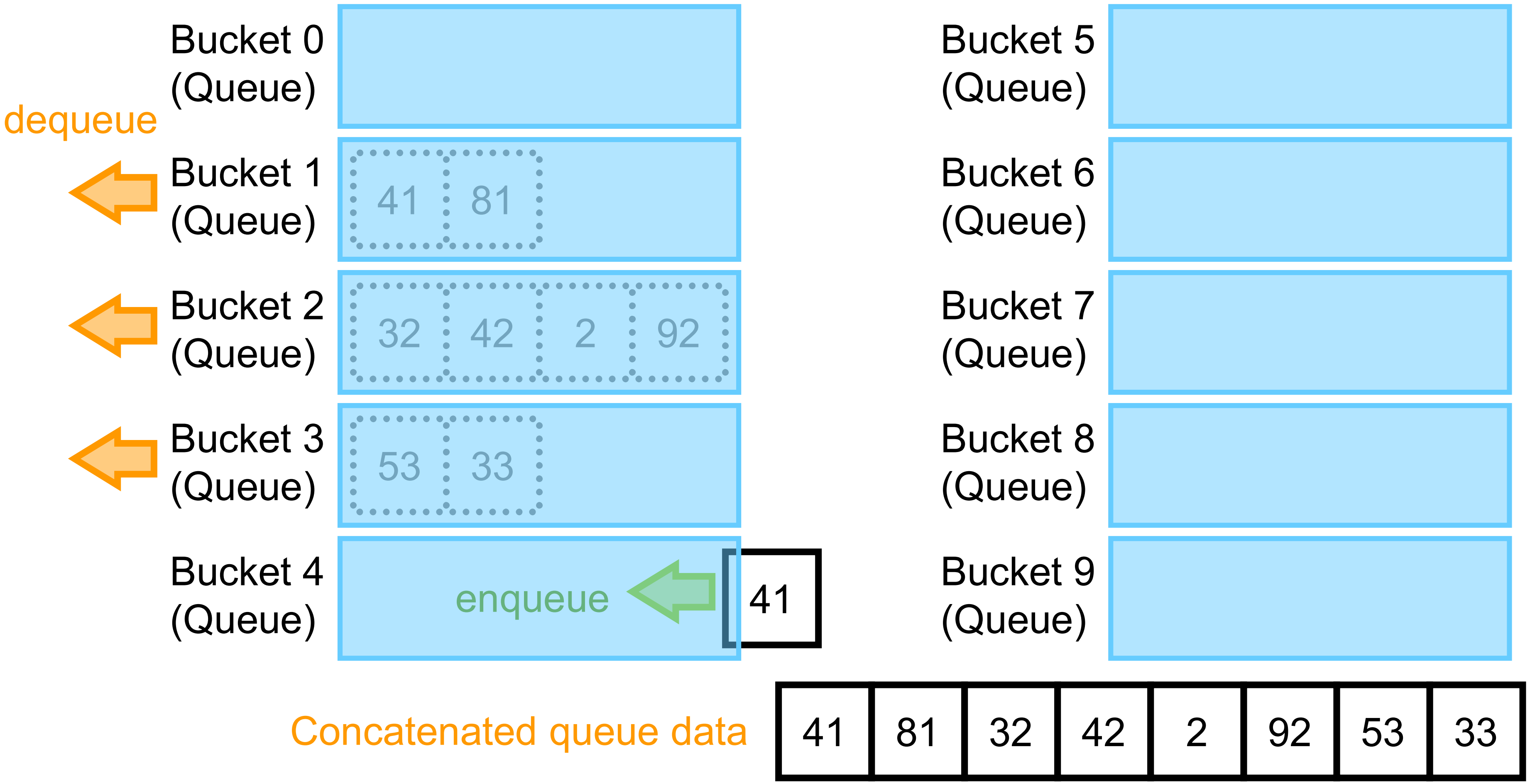
Enqueue the element into the queues (buckets) one by one (by the LSD)



After 1st Pass

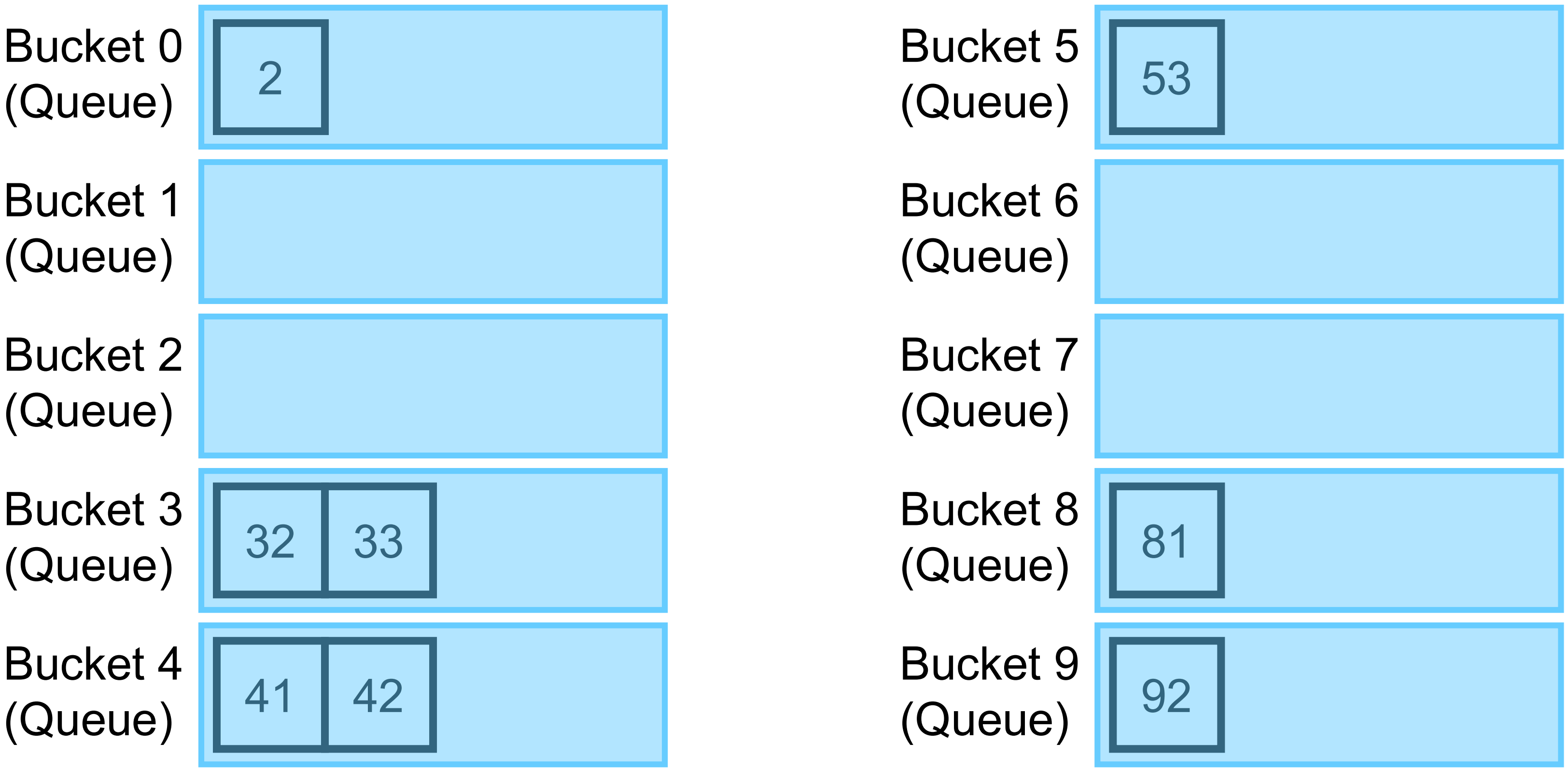


Dequeue All, then Enqueue One by One Again



After 2nd Pass

Using queues to maintain the stability
(equal keys remain the same order)



Concatenated queue data

2	32	33	41	42	53	81	92

How to Obtain the Digits?

- To obtain the least significant digit
  - bucket # = e % 10
- To obtain the 2nd least significant digit
  - bucket # = e / 10 % 10
- To obtain the 3rd least significant digit
  - bucket # = e / 100 % 10
- To obtain the k^{th} least significant digit
  - bucket # = e / pow(10, k - 1) % 10

Complexity Analysis

- Time to enqueue and dequeue the elements in each pass is O(n)
- There are k passes
  - k is the no. of digits of the elements
- The time complexity is O($k \cdot n$)
- Radix sort's complexity depends directly on the length of elements
  - Other sorting methods depends on n only

Complexity Analysis

- If *k* is large and *n* relatively small, radix sort is not a good choice, e.g. to sort 5 and 100,000,000,000,000,000
  - k = 18 and n = 2
  - Use comparison sorts
- But if *k* is small and *n* is large, then radix sort will be **faster** (linear time) than any other method we have studied, e.g. to sort #0 ~ #99 (uniformly distributed)
  - k = 2 and n = 100
  - Time complexity is O(n)
- Other drawbacks
  - Memory overhead: additional memory for queues
  - Space complexity: O(*n*)

Advanced Example

- Sorting characters
- **Two** buckets are enough
- "Convert" characters into binary bits first
- Compare the bits one by one

0100 0001	A
0100 0010	B
0100 0011	C
…	
0101 1010	Z

Sorting Characters

The unsorted string is “SORTING”, sort the characters by ASCII code in ascending order

The original data

0101 0011	S
0100 1111	O
0101 0010	R
0101 0100	T
0100 1001	I
0100 1110	N
0100 0111	G

After 1st pass

0101 0010	R
0101 0100	T
0100 1110	N
0101 0011	S
0100 1111	O
0100 1001	I
0100 0111	G

After 2nd pass

0101 0100	T
0100 1001	I
0101 0010	R
0100 1110	N
0101 0011	S
0100 1111	O
0100 0111	G

Sorting Characters

After 3rd pass

0100 1001	I
0101 0010	R
0101 0011	S
0101 0100	T
0100 1110	N
0100 1111	O
0100 0111	G

After 4th pass

0101 0010	R
0101 0011	S
0101 0100	T
0100 0111	G
0100 1001	I
0100 1110	N
0100 1111	O

After 5th pass

0100 0111	G
0100 1001	I
0100 1110	N
0100 1111	O
0101 0010	R
0101 0011	S
0101 0100	T

Sorting Characters

The sorted string is "GINORST"

After 6^{th} pass

0100 0111	G
0100 1001	I
0100 1110	N
0100 1111	O
0101 0010	R
0101 0011	S
0101 0100	T

After 7^{th} pass

0100 0111	G
0100 1001	I
0100 1110	N
0100 1111	O
0101 0010	R
0101 0011	S
0101 0100	T

After 8^{th} pass

0100 0111	G
0100 1001	I
0100 1110	N
0100 1111	O
0101 0010	R
0101 0011	S
0101 0100	T

How to Obtain the Bits?

- To obtain the last bit, use the bit-wise operator
  - int bit; char c = ‘S’; //0101 0011 (binary)
  - bit = c & 0x01; //0x01 (hex) = 0000 0001 (binary)
  - //0101 0011 AND 0000 0001 = 0000 0001 = 1

- To obtain 2nd last bit
  - bit = (c >> 1) & 0x01;
  - // >> 1: shift the bits one step to right. The original right most bit is discarded
  - //c >> 1: 0010 1001
  - //0010 1001 AND 0000 0001 = 1

How to Obtain the Bits?

- To obtain 3rd last bit
  - (c >> 2) & 0x01;
  - //c >> 2: 0001 0100
  - //0001 0100 AND 0000 0001 = 0

- To obtain the k^{th} bit
  - (c >> (k – 1)) & 0x01;

An Easier Method

- If you can't understand the bit operation, you may use a **slower** method to get the bits

```
int bit; char c = 'S';
bit = c % 2;  // to get the 1st bit
bit = c / 2 % 2;  // to get the 2nd bit
bit = c / 4 % 2;  // to get the 3rd bit
```

- Actually shifting the bits means dividing the number by the power of 2
- The general equation to obtain the k^{th} base b digit of symbol c
  - digit = c / pow(b, k – 1) % b;

Advanced Example

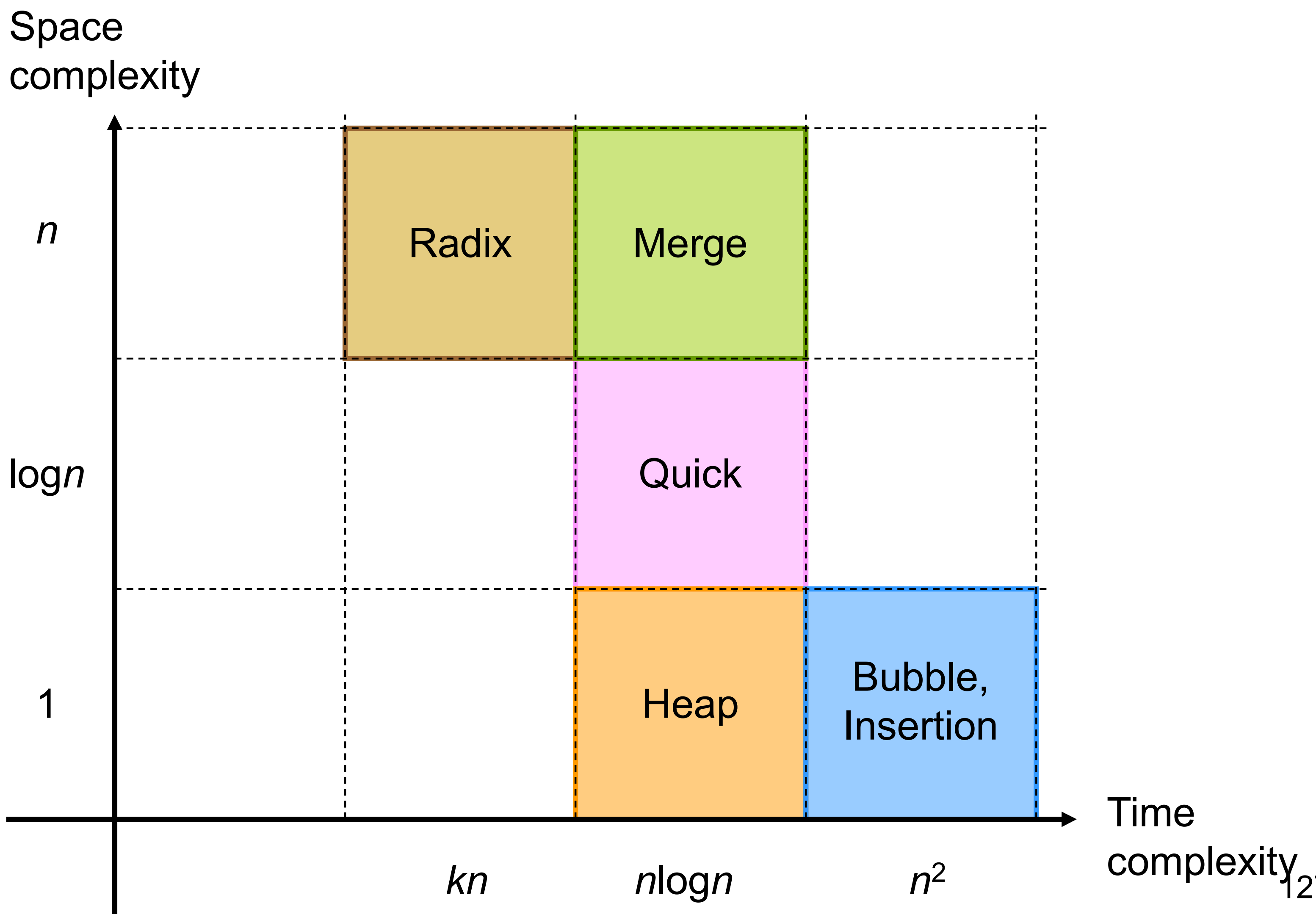
- Radix sort can have many variations

- e.g. Sorting strings
  - Each “digit” is a character
  - Need 26 buckets (since there are 26 characters)
  - Sort with the least significant character first

Example: sorting strings

Original data	After 1st pass	After 2nd pass	After 3rd pass
now	sob	tag	ace
for	nob	ace	bet
tip	ace	bet	dim
ilk	tag	dim	for
dim	ilk	tip	hut
tag	dim	sky	ilk
jot	tip	ilk	jot
sob	for	sob	nob
nob	jot	nob	now
sky	hut	for	sky
hut	bet	jot	sob
ace	now	now	tag
bet	sky	hut	tip

Summary



Space complexity
n
logn
1
Radix
Merge
Quick
Heap
Bubble, Insertion
kn
nlogn
n2
Time complexity

Built-in Sort Function

Built-in Sort Function in C

- C standard library function that implements a polymorphic sorting algorithm for arrays of arbitrary objects according to a user-provided comparison function.
- Include <cstdlib>

```
void qsort(void *base, unsigned num, unsigned objSize,
              int (*compare)(const void *, const void *));
```

```
1.  base        void pointer, the base address of input array
2.  num         number of elements in array
3.  objSize     size in bytes of each element in the array.
4.  compare     function pointer, for ordering two elements
```

Example 1 of qsort()

```
// Use qsort()to sort an array of fraction

int compareFraction(const void *a, const void *b) {
         fraction *f1 = (fraction *)a;  //type cast the pointer
         fraction *f2 = (fraction *)b;  //before using it to refer to an object

         if (*f1 == *f2)             return 0;
         else if (*f1 < *f2)         return -1;
         else                        return 1;
}

int main() {
   int len = 100;
   fraction *list = new fraction[len];

   // codes to assign values to list[] ……
   qsort(list, len, sizeof(fraction), compareFraction);
}
```

Example 2 of qsort()

```
// Use the qsort function to sort a list of names (cstring, char [])

// the void pointer arguments point to cstring (char*)
// i.e. (char**), which is pointer-to-(char*)
int compareString(const void *a, const void *b) {
   char **c1 = (char **)a;
   char **c2 = (char **)b;

   // dereferencing once becomes cstring (char *)
   return strcmp(*c1, *c2);  //compare cstring
}

int main() {
   char *name[] = {“Wong Chi Ming”,
                   “Chan Tai Man”,
                   “Ho Pui Shan”,
                   “Au Pui Ki”,
                   “Cheung Ka Man”};

    qsort(name, 5, sizeof(char *), compareString);
}
```