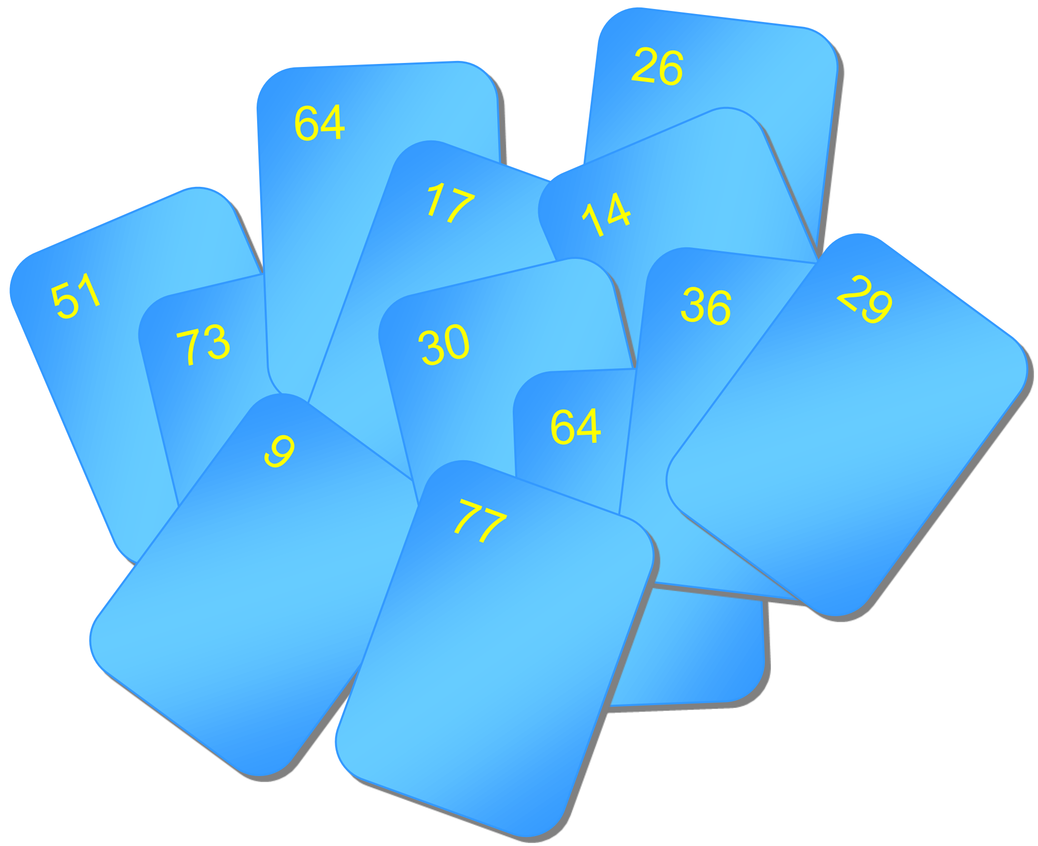
EE2331 Data Structures and Algorithms

Sorting

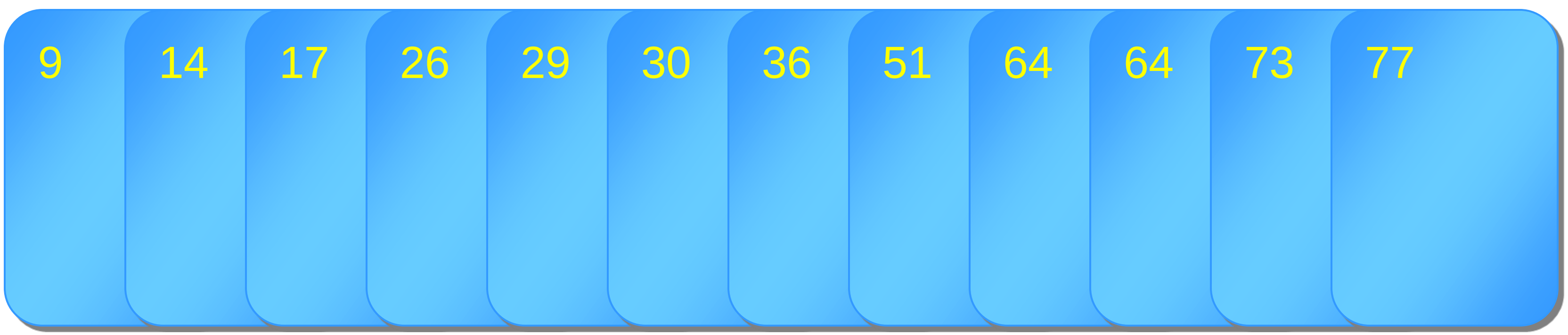
Given a List in Random Order

- How to find the largest number?
- How to find the smallest number?
- How to determine if an arbitrary number exists in the list?



Given a List in Ascending Order

- How to determine the largest / smallest / any arbitrary number now?
  - Remember binary search?



  - The numbers can be also in descending order
- Or at least in some proper order (such as BST and heap)

Sorting

- To rearrange the order (ascending or descending) of data for **ease of searching**
- In this notes, discuss the various ways to sort a large amount of data and compare them by time/space efficiency.
  - $O(n)$, $O(n\log n)$, $O(n^2)$ …
- Efficiency of a sorting method is usually measured by the number of comparisons and data movements required.

Outline

- Terminologies
- 6 sorting algorithms
  - Bubble Sort, Insertion Sort, Merge Sort
  - Heapsort, Quicksort, Radix Sort
- Sorting using Queues
- Sorting using Stacks
- Indirect Sorting

Terminologies

Stable vs. Unstable

Internal vs. External

Stable & Unstable Sort

- Sequence before sorting: 5, 3, $8^{\#}$, 6, 8*
- Sequence after sorting: 3, 5, 6, $8^{\#}$, 8*
  - Stable sort
- Sequence after sorting: 3, 5, 6, 8*, $8^{\#}$
  - Unstable sort

- Stable: if it always leaves elements with equal keys in their original order

Internal & External Sort

- Internal sort
  - Small data volume
  - Process in main memory
- External sort
  - Large amount of data
  - Need external or secondary storage in processing (e.g. disk storage)

Internal Sorting Algorithms

- In this course, we shall only discuss internal sorting algorithms. To simplify discussion, sorting of an integer array is used in our examples.
  1. Bubble Sort
  2. Insertion Sort
  3. Heap Sort
  4. Radix Sort
  5. Quick Sort
  6. Merge Sort (also good for external sort)

- How to choose the sorting algorithm?

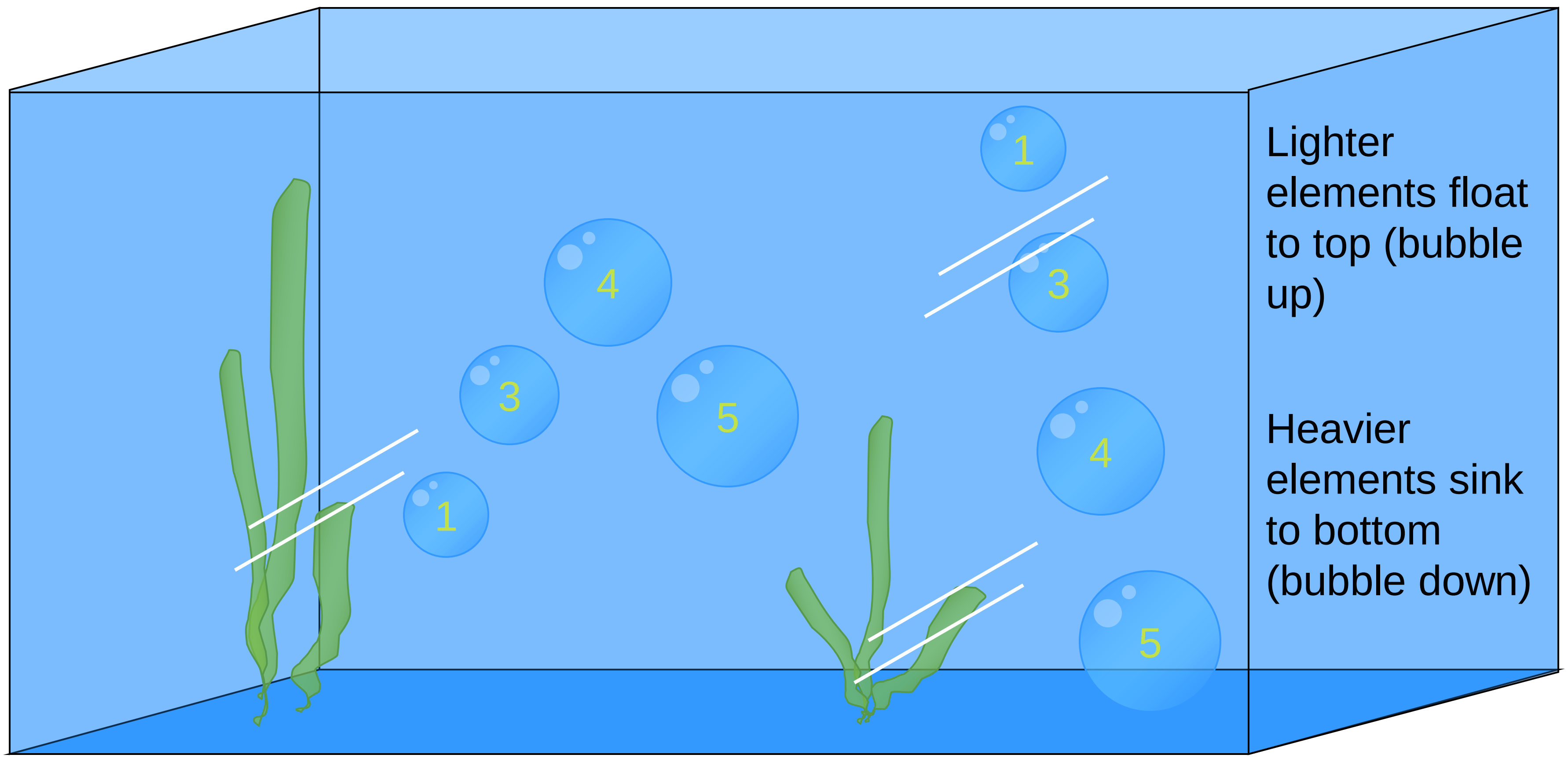
Bubble Sort

Time Complexity: O(n^2)

Space Complexity: O(1)

Daily Life Example

- Consider the goldfish bowl



Bubble Sort

- The easiest sorting algorithm
- The most time consuming algorithms
- Another name: **interchange sort**
- The idea:
  - Scanning the list from one end to the other
  - When a pair of adjacent keys is found to be out of order, swap those entries
  - In each pass, the largest key in the list will be bubbled to the end, but the earlier keys may still be out of order

Bubble Sort Example

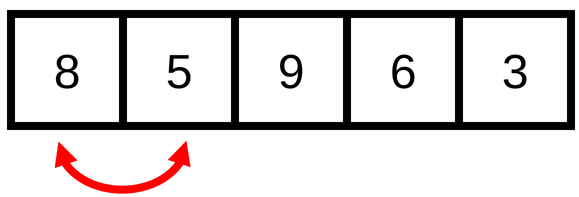
- Sort the sequence {8, 5, 9, 6, 3} in ascending order
- The final result should be {3, 5, 6, 8, 9}

8	5	9	6	3

unsorted list

Bubble Sort: 1st Pass

- 1st pass, 1st comparison



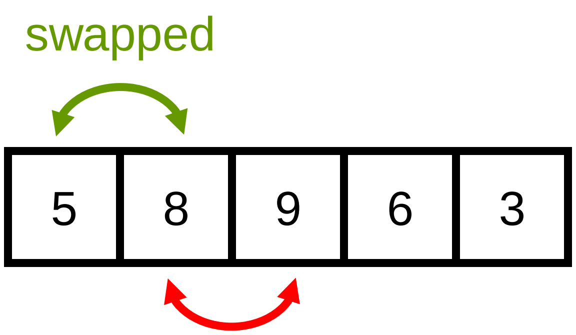
Compare 1st element with 2nd element

i.e. 8 vs. 5

if left hand side > right hand side, swap them!

Bubble Sort: 1st Pass

- 1st pass, 2nd comparison



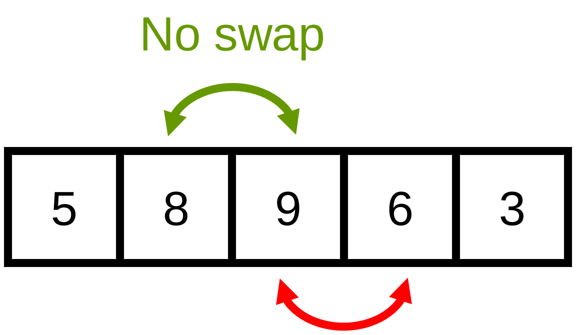
Compare 2nd with 3rd element

i.e. 8 vs. 9

Since left hand side < right hand side, do nothing!

Bubble Sort: 1^{st} Pass

- 1^{st} pass, 3^{rd} comparison



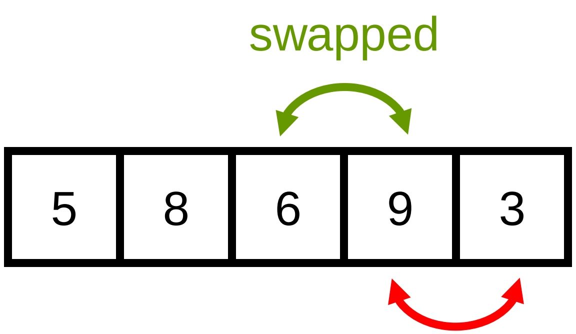
Compare 2^{nd} with 3^{rd} element

i.e. 9 vs. 6

Since left hand side < right hand side, swap them!

Bubble Sort: 1st Pass

- 1st pass, 4th comparison



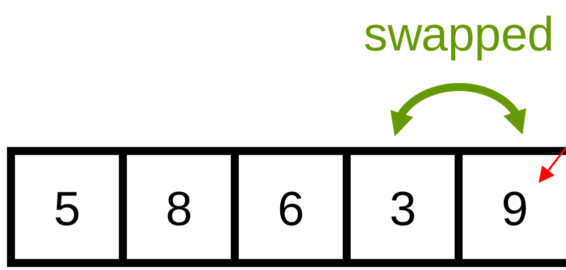
Compare 2nd with 3rd element

i.e. 9 vs. 3

Since left hand side < right hand side, swap them!

Bubble Sort: 1st Pass

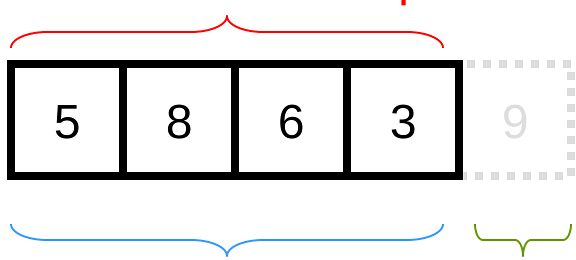
- After 1st pass



The largest element bubbled to bottom after running the 1st pass

Bubble Sort: 2nd Pass

- Start from 2nd pass, no need to consider the largest element (the last element)



Bubble Sort: 2nd Pass

2nd pass, 1st comparision

5	8	6	3	9

partially sorted list

2nd pass, 2nd comparision

5	8	6	3	9

2nd pass, 3rd comparision

5	6	8	3	9

The 2nd largest element fall to 2nd bottom after running the 2nd pass

After 2nd pass

5	6	3	8	9

unsorted list

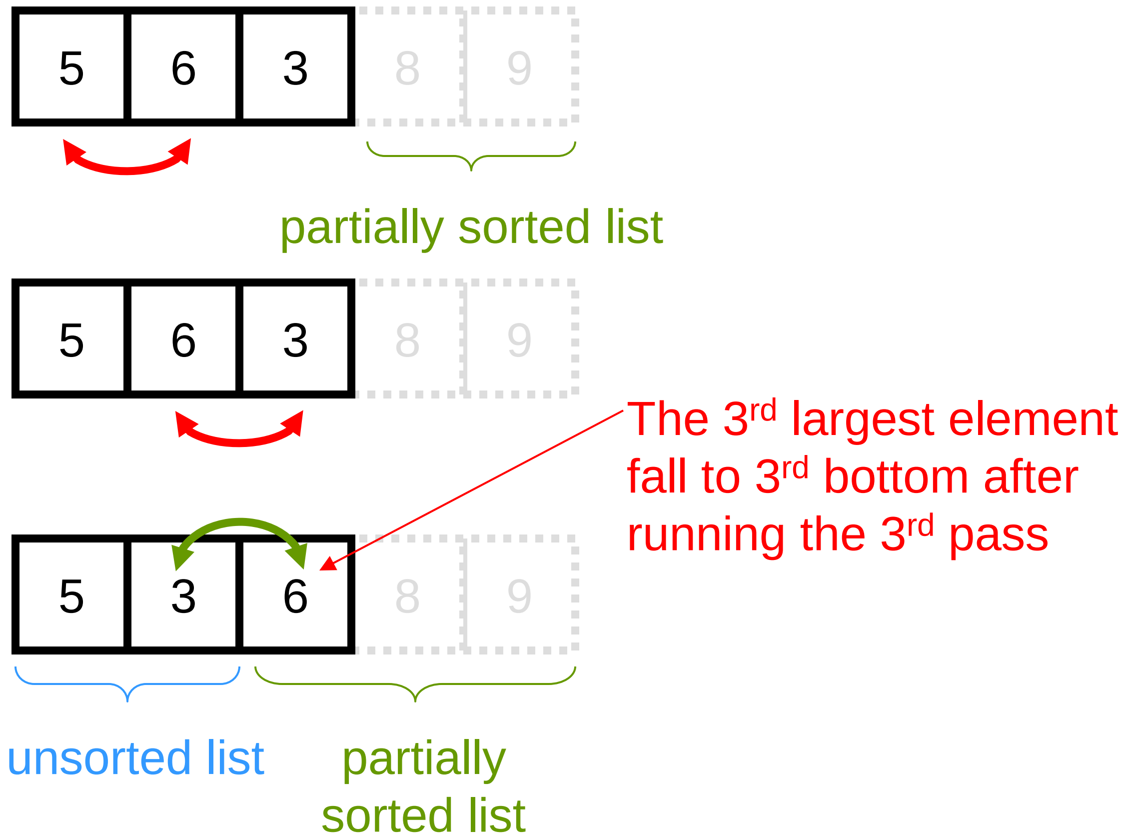
partially sorted list

Bubble Sort: 3rd Pass

3rd pass, 1st comparision

3rd pass, 2nd comparision

After 3rd pass

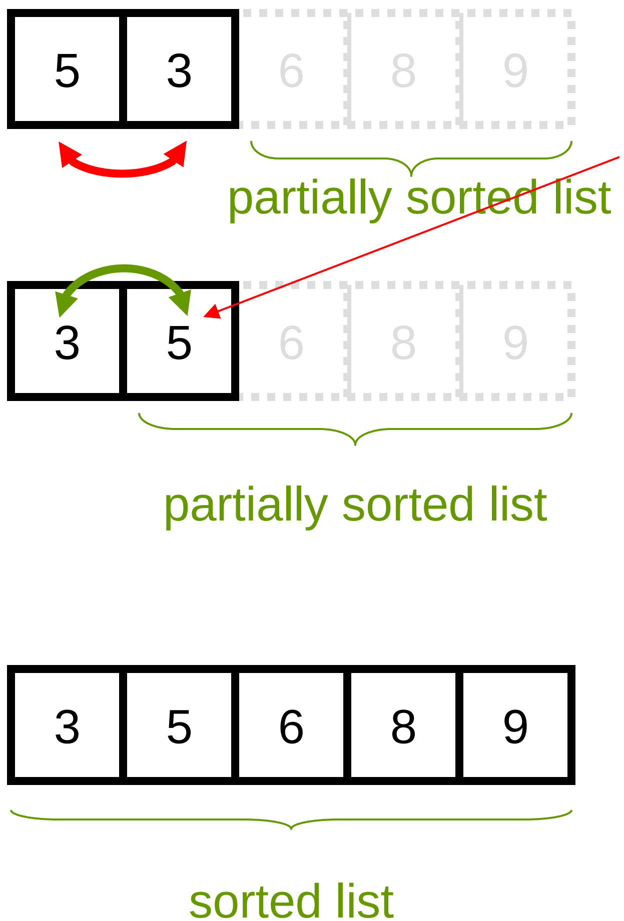


Bubble Sort: 4^{th} Pass

4^{th} pass, 1^{st} comparision

After 4^{th} pass

The final sequence



The 4^{th} largest element fall to 4^{th} bottom after running the 4^{th} pass

Not necessary to run the 5^{th} pass (why?)

Time Complexity

- The amount of time to compare two numbers is constant O(1)
- The amount of time to swap two numbers is also constant O(1)
- The amount of time require to sort the sequence is proportional to the **number of comparisons** (or **swaps**)

How Many Comparisons?

- If there are n elements in total
  - No. of passes?
    - $n - 1$
  - How many comparisons in each pass?
    - i^{th} pass: $n - i$ comparisons
  - How many comparisons in total?
    - $$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$
  - Therefore, the time complexity is
    - $O(n^2)$

How Many Swaps in Total?

- # of swaps is at most # of comparisons
  - $\leq \frac{1}{2}n(n-1)$
- The worst case: the algorithms has to run all the *n* - 1 passes
- The best case: (already sorted list) the algorithms stops after running the 1st pass (i.e. O(*n*))

Drawback of Bubble Sort

- Slow
  - Worst case: O(n^2)
  - Average case: O(n^2)
    - Half the number of comparisons: $\sum_{i=1}^{n-1}(\frac{n-i}{2}) = \frac{n(n-1)}{4}$
  - Best case: O(n)

Simple Version

Mind the for-loop indexes here

i control the no. of passes

j control the no. of comparisons in each pass

```
void bubble(int data[], int n) {
    int i, j;
    //sort in ascending order
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - 1 - i; j++)
            if (data[j] > data[j+1])
                swap(&data[j], &data[j+1]);
}
```

1 pass

Each pass consists of comparing each element with its successor

Swap these two elements if they are not in proper order

After each pass *i*, the elements from ***data[n - i - 1]*** to ***data[n - 1]*** are sorted

Improved Version

```
void bubble(int data[], int n) {
    int i, j, no_swap;
    //sort in ascending order
    for (i = 0; i < n - 1; i++) {
        no_swap = true;
        for (j = 0; j < n – 1 - i; j++)
            if (data[j] > data[j+1]) {
                swap(&data[j], &data[j+1]);
                no_swap = false;
            }
        if (no_swap) break;
    }
}
```

1 pass

The algorithm breaks the for-loop and stops at once if there is no swap in one of the pass

Bubble Up and Down

- The previous algorithm bubble down the largest element in each pass
- The alternative way to implement bubble sort is:
  - Bubble up the smallest element to the front of the sublist in each pass
  - Their time and space complexities are the same

Bubble Up

Mind the changes in red

j starts from end of the list up to *i*

If the right element is smaller, bubble up to the left of the list

```
void bubble(int data[], int n) {
    int i, j, no_swap = 0;
    //move smallest element to front in each pass

}
```

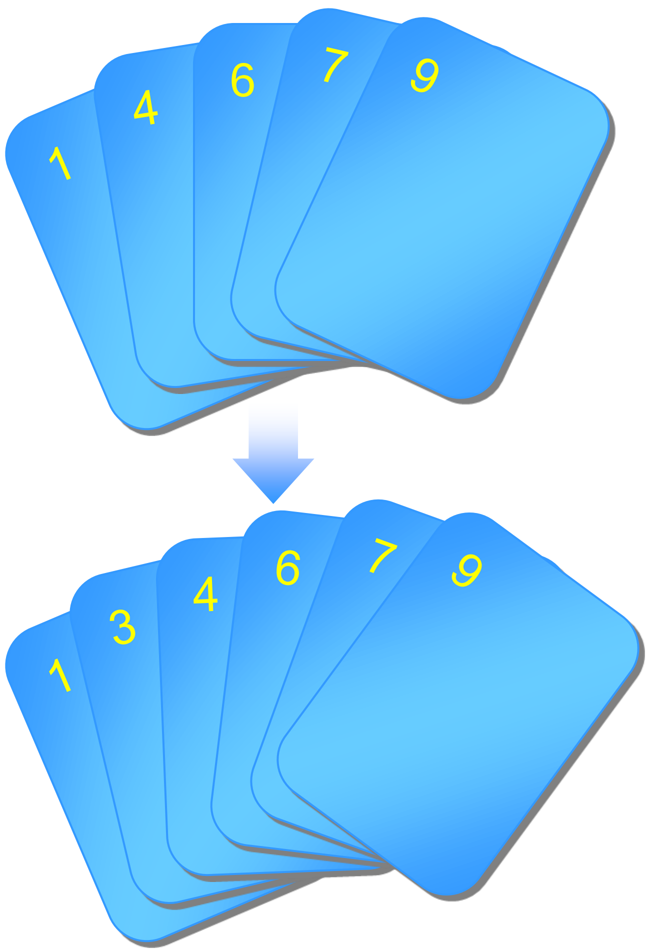
Insertion Sort

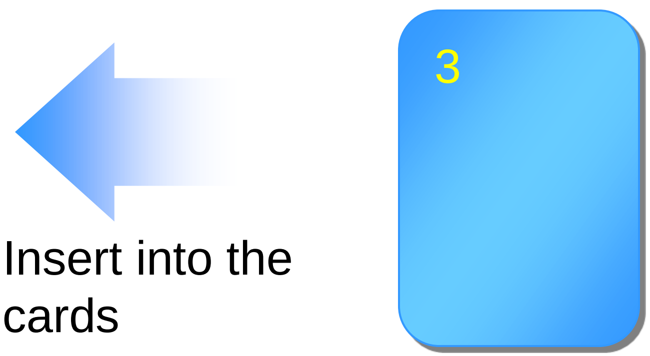
Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Daily Life Example

- The idea of insertion is like playing cards





Any pre-condition?

Insertion Sort

- Similar to bubble sort, consists of n - 1 passes
- Instead of bubbling the largest (or smallest) element, insertion sort successively inserts a new element into a (sorted) sublist in each pass
- Initially 1st element may be thought of as a sorted sublist of only one element
- After each sorted-insertion, the sorted sublist's length grows by 1.
- Insertion sort makes use of the fact that elements in the sublist are already known to be in sorted order.

Insertion Sort Example

The unsorted list:

The 1st pass

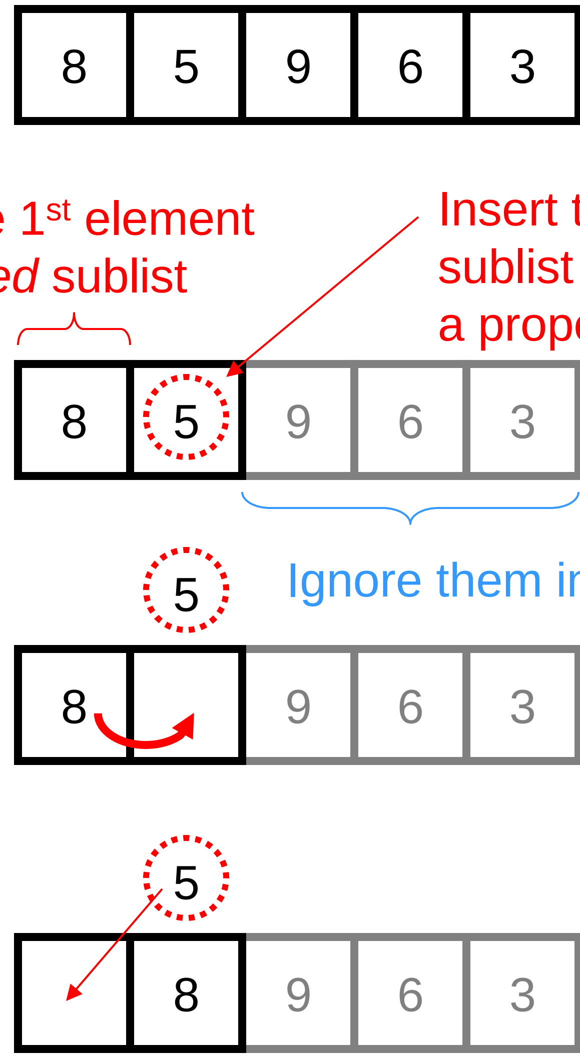
Consider the 1st element as a *sorted* sublist

Insert this element into the left sublist such that they maintain a proper order

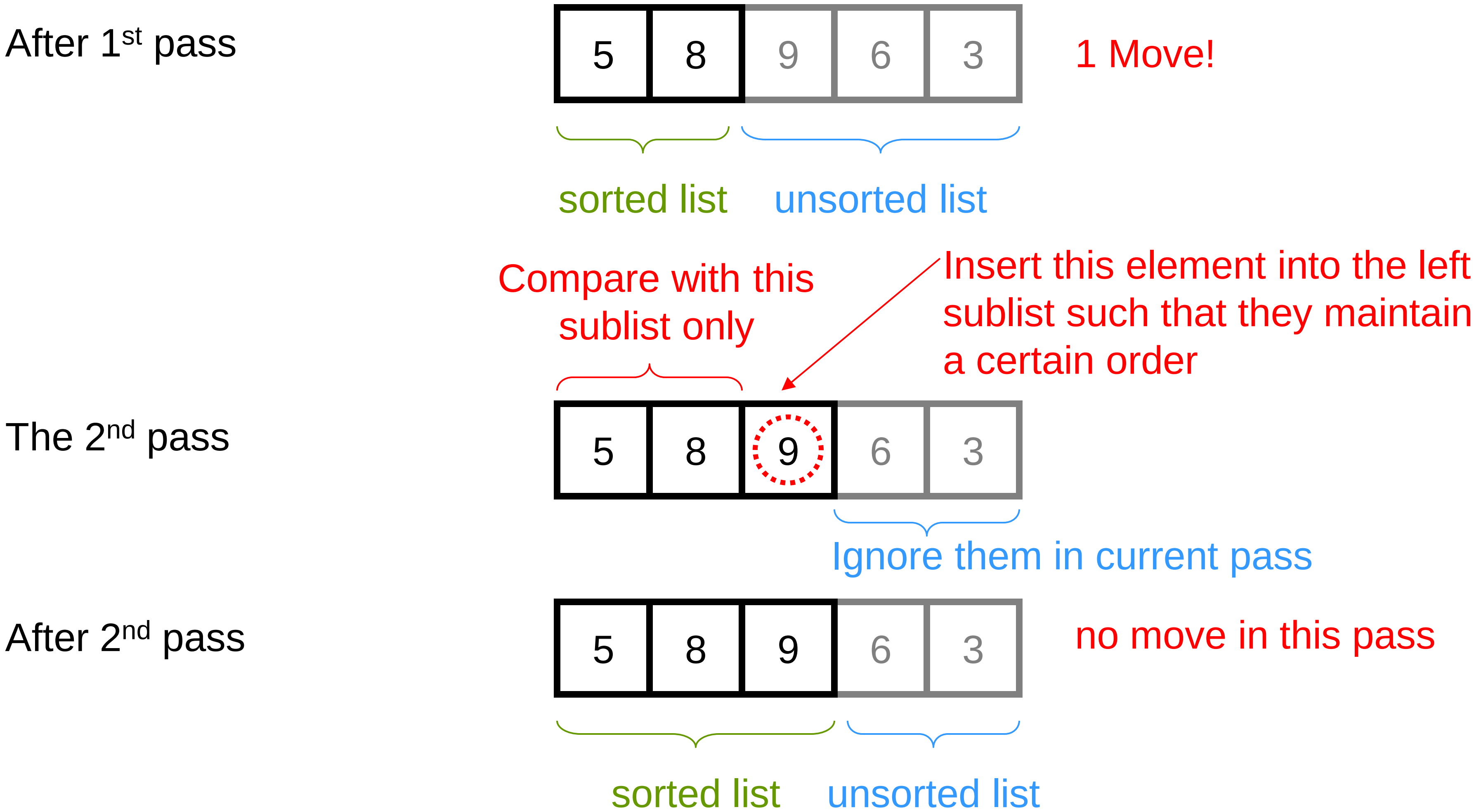
Ignore them in current pass

Pick up “5”. Move “8” to right

Insert “5” to the appropriate position



Insertion Sort Example



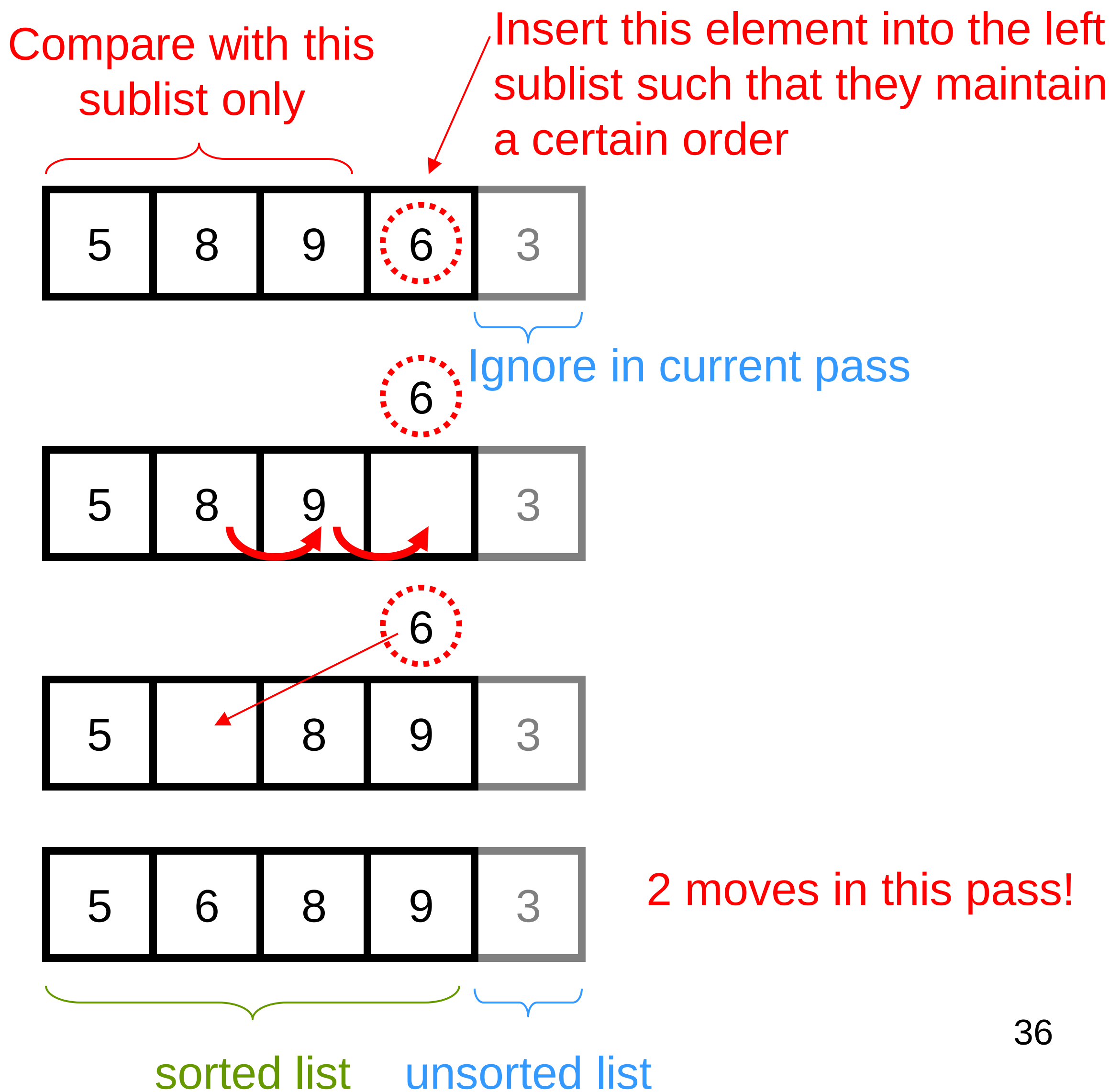
Insertion Sort Example

The 3rd pass

Pick up “6”. Move “9” and “8” to right

Insert “6” to the appropriate position

After 3rd pass



Insertion Sort Example

Compare with this sublist only

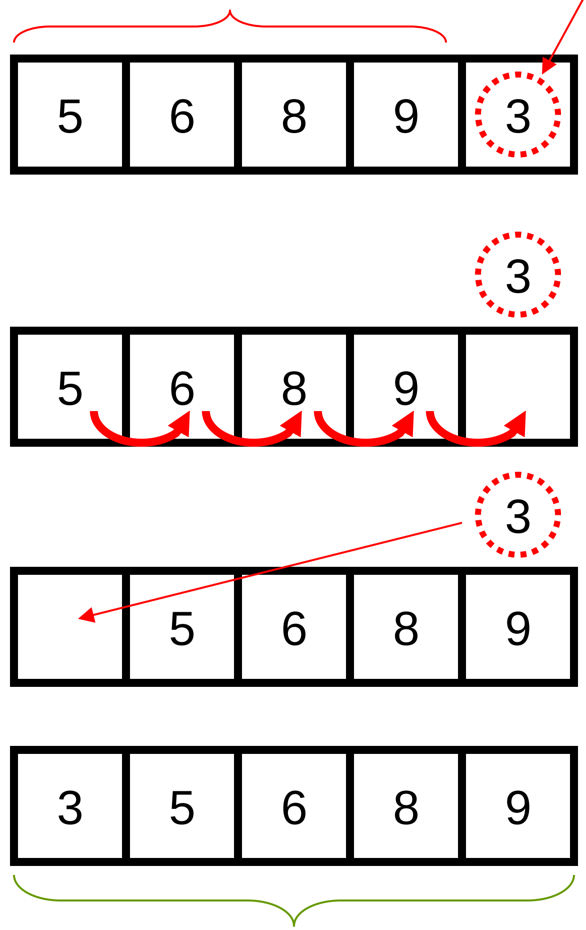
Insert this element into the left sublist such that they maintain a certain order

The 4th pass

Pick up “3”. Move “9”, “8”, “6” and “5” to right

Insert “3” to the appropriate position

After 4th pass



4 moves in this pass!

sorted list

Insertion Sort

```
void insertion(int data[], int n) {
    for (int i = 1; i < n; i++) {           // n -1 passes
        int temp = data[i];                 // element to be inserted
        // shift the elements in the sublist if they are not in order.
        // the sublist is from data[0] to data[i]
        int j;
        for (j = i-1; j >= 0 && data[j] > temp; j--)
            data[j+1] = data[j];

        data[j+1] = temp;     // j+1 is the location for insertion
    }
}
```

1 pass

Generic Version

- Generic sorting function for any data type

function pointer

```
template<class Type>
void insertionSort(Type *x, unsigned N,
                   int (*compare)(const Type&, const Type&)) {

  for (int i = 1; i < N; i++) {
    Type t = x[i];

    int j;
    for (j = i-1; j >= 0 && compare(x[j], t) > 0; j--)
      x[j+1] = x[j];

    x[j+1] = t;
  }
}
```

Complexity Analysis

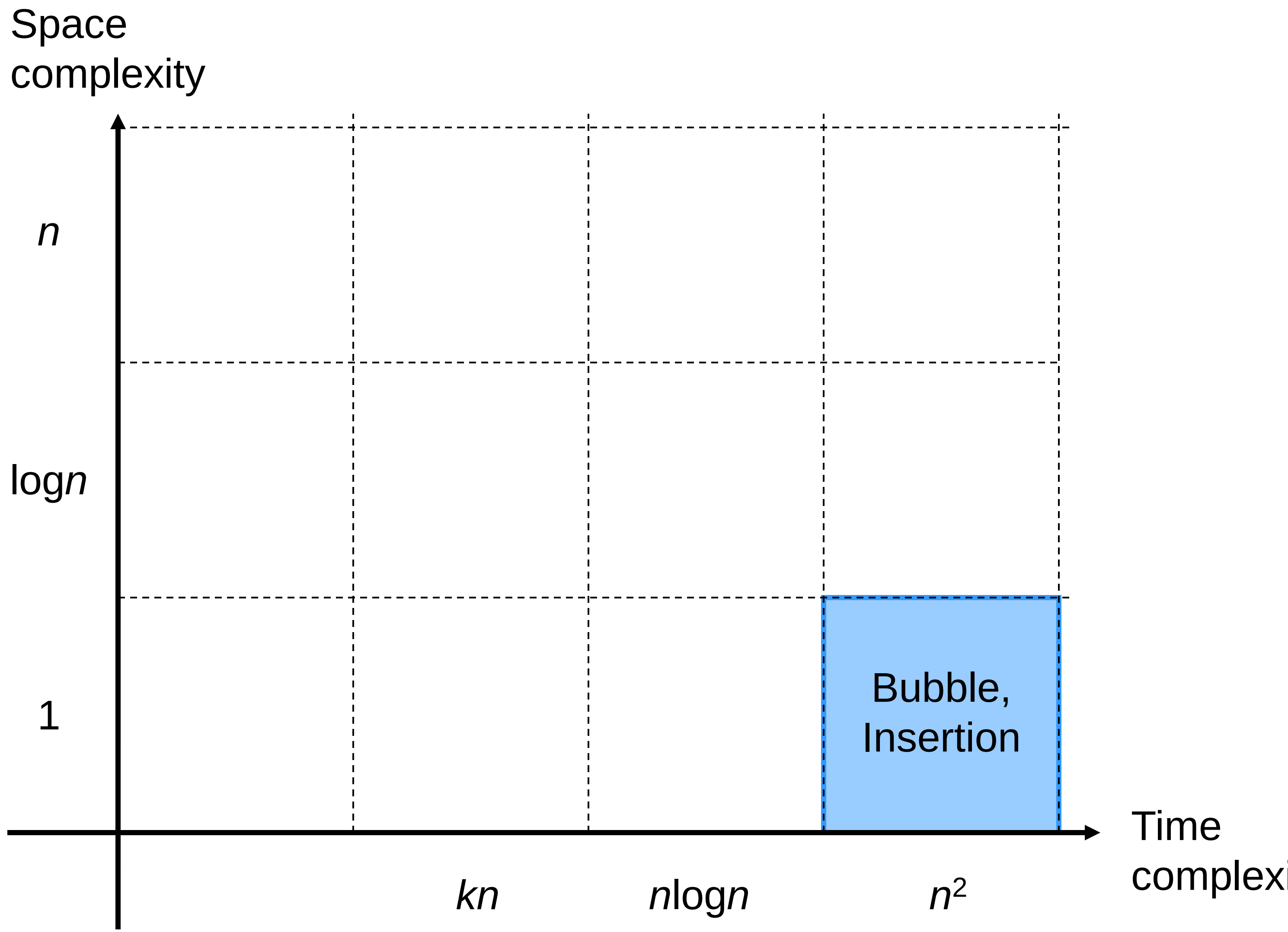
- Like bubble sort, need an extra temporary memory
  - Space complexity: O(1)
  - Bubble sort: the temp. variable is used for **swapping**
  - Insertion sort: the temp. variable is used to **hold** the element that going to be inserted into the sublist

Complexity Analysis

- The best case: O(n)
  - The list is already sorted; scan it once!
- The worst case: O(n^2)
  - *n-1* items to be inserted
  - At most *i* comparisons at *i-th* insertion
  - The total no. of comparisons = $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$
- The average case: O(n^2)
  - Half the number of comparisons

- Because of the simplicity of insertion sort, it is the fastest sorting method when the number of elements *N* is small, e.g. *N* < 10.

Summary (Average Performance)



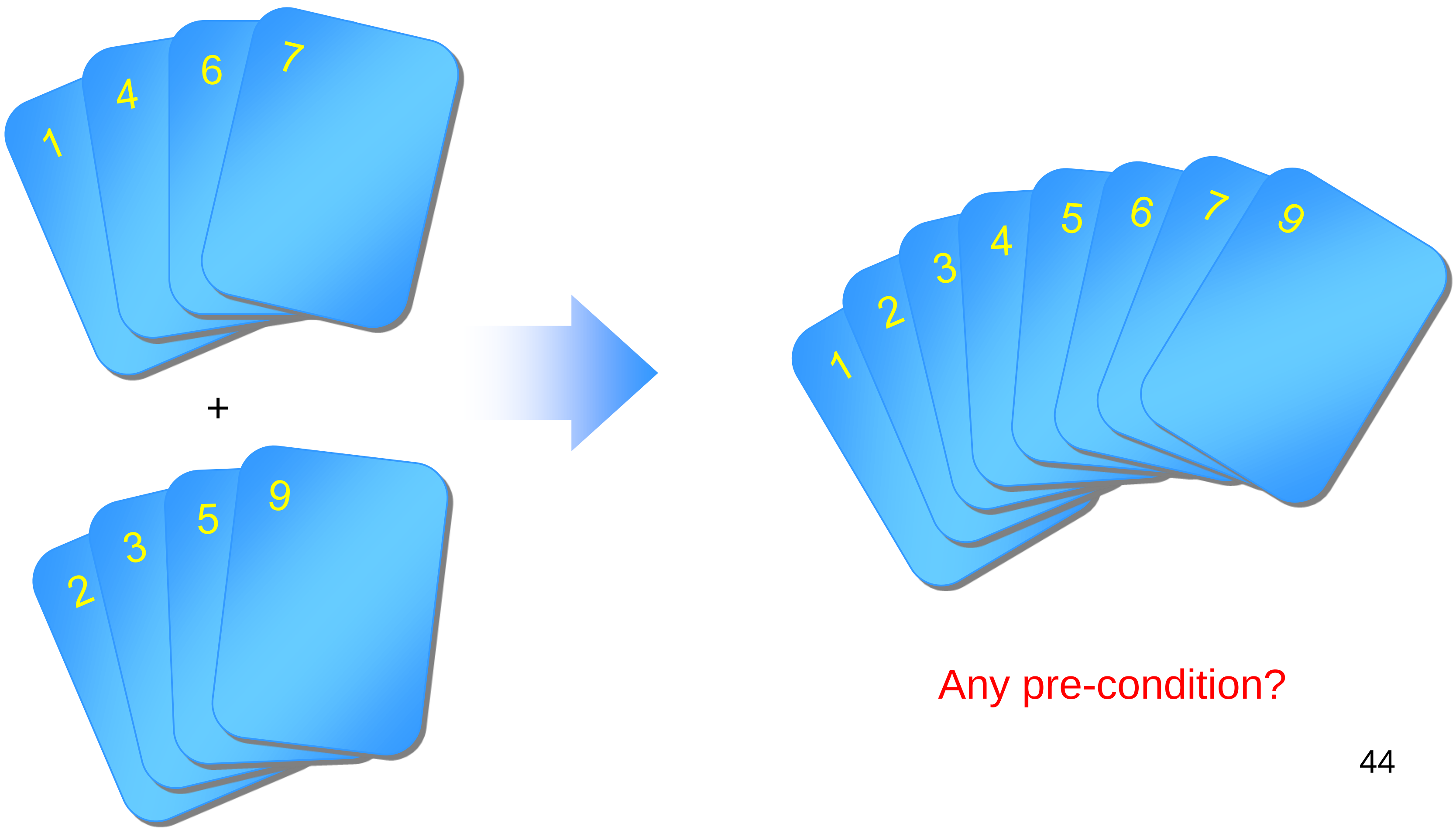
Merge Sort

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

Space Complexity: $O(n)$

Daily Life Example

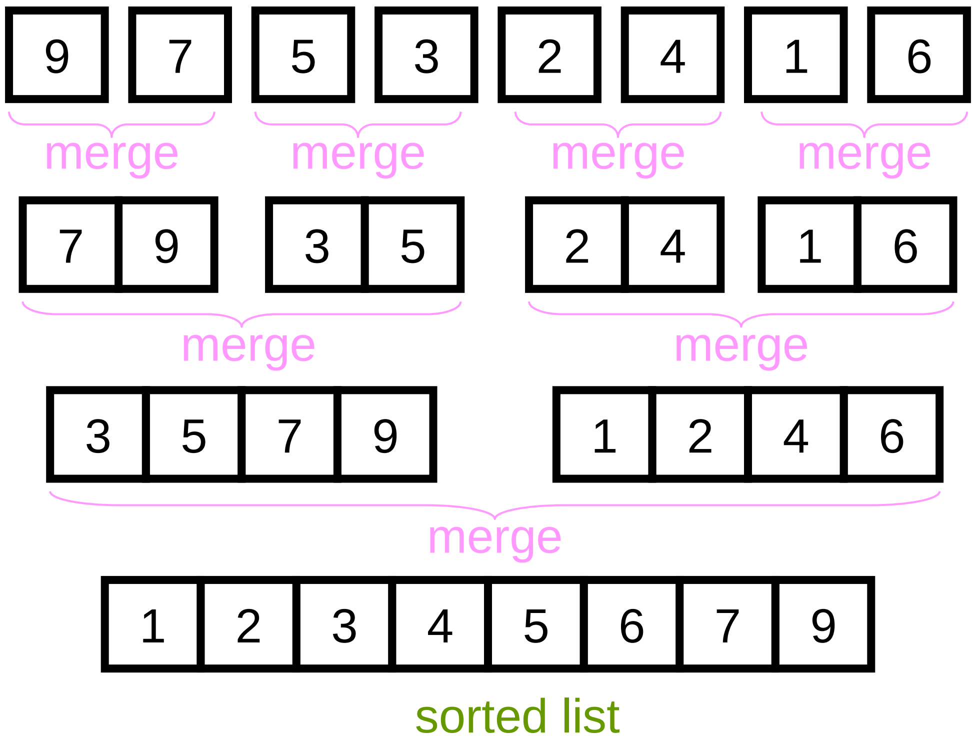
- The idea of merging



Any pre-condition?

The Algorithm

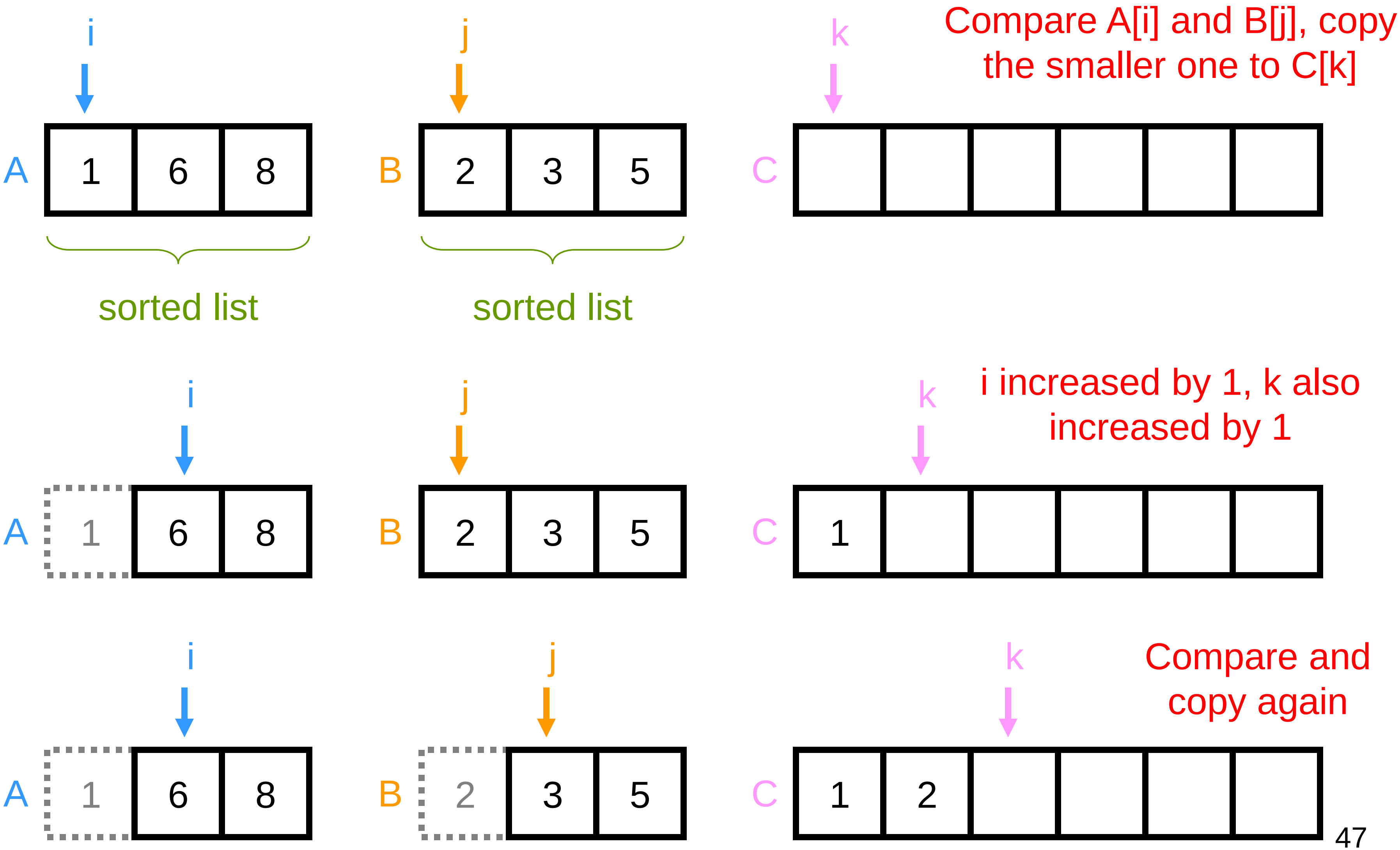
- Initially the input list is divided into *N* sublists of size 1
- Adjacent pairs of lists are merged to form larger sorted sublists
- The merging process is repeated until there is only one list



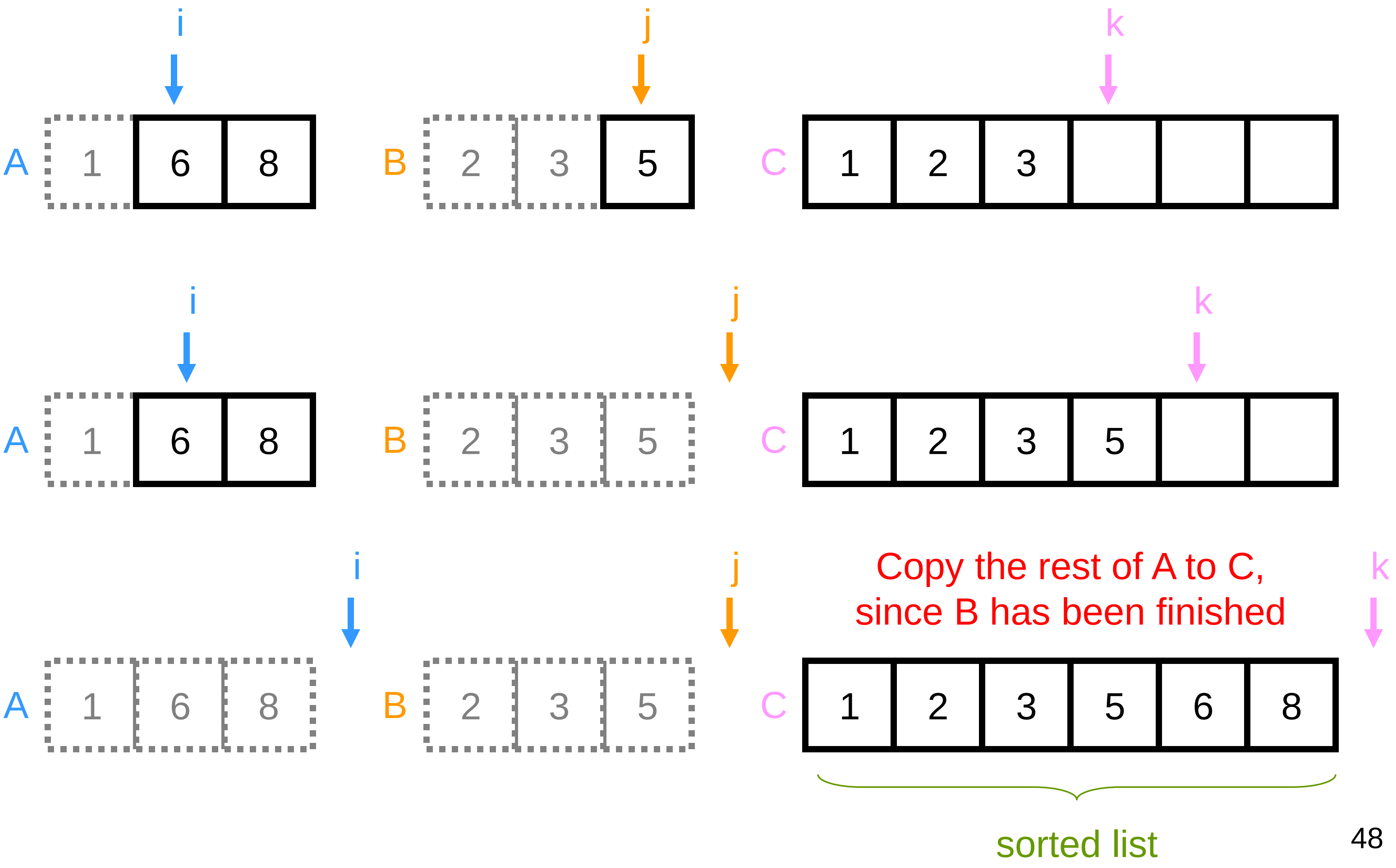
Merging

- To merge 2 **sorted** lists
- It takes 2 input arrays *A*[] & *B*[], 1 output array *C*[] and 3 counters (*i*, *j*, *k*) for the arrays respectively
- The smaller of *A*[*i*] and *B*[*j*] is copied to *C*[*k*], then the counters are advanced
- If either *A*[] or *B*[] finishes first, the reminder of the other array is copied to *C*[]

Merge Sort Example



Merge Sort Example



Merge Adjacent Lists

```
void merge(int data[], int first, int mid, int last) {
    int temp[SIZE], i = first, j = mid + 1, k = 0;
    while (i <= mid && j <= last) {
        if (data[i] <= data[j])
            temp[k++] = data[i++];
        else
            temp[k++] = data[j++];
    }
    while (i <= mid) temp[k++] = data[i++];
    while (j <= last) temp[k++] = data[j++];
    i = 0;
    while (i < k) data[first+i] = temp[i++];
}
```

Compare A[i] and B[j], copy the smaller one to temp[k]

A is data[first…mid]

B is data[mid+1…last]

C is temp[…]

The remaining A or B will be copied into temp

The sorted temp. array is copied back to data

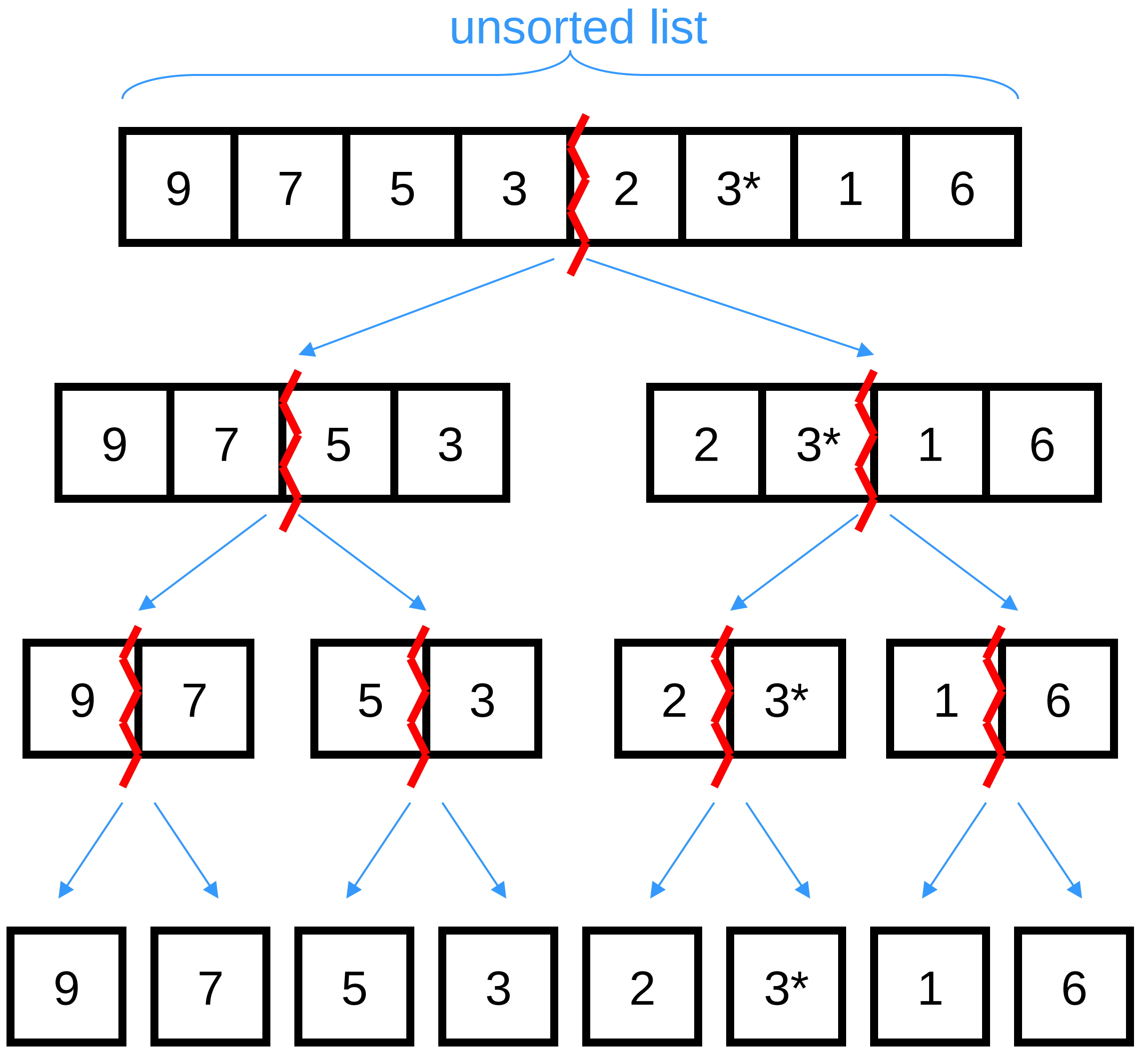
Divide-and-Conquer

- This algorithm is a classic divide-and-conquer strategy
- Very powerful use of recursion
- The problem is divided into smaller problems and solved independently and recursively
- The conquering phase consists of merging together the sorted lists

Dividing Phase

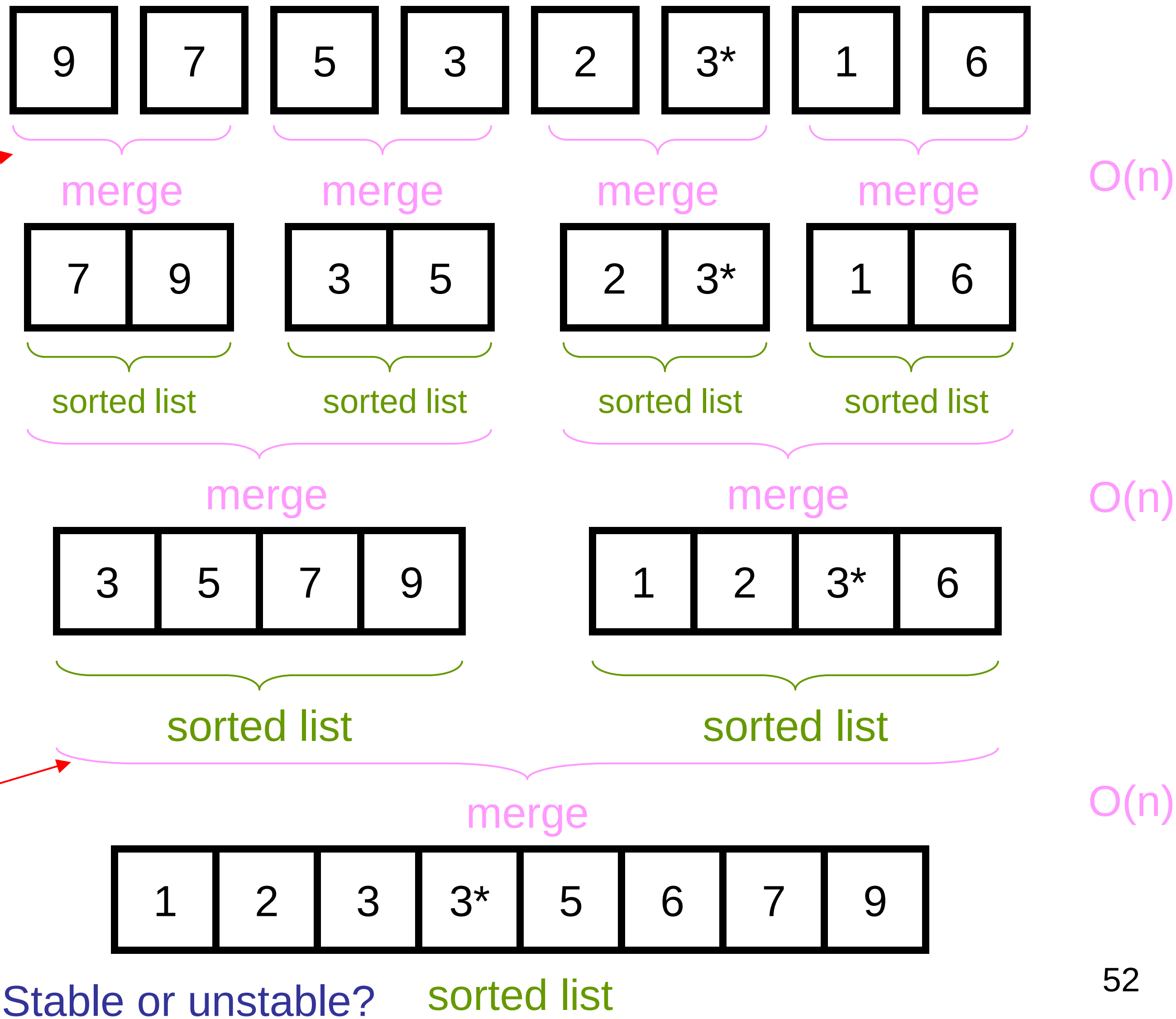
Divide the list into halves

Divide again and again until only one single element left in the list



Conquering Phase

Merge the small sorted lists into larger sorted lists



Merge again and again until one single list left

Stable or unstable?

Merge Sort (Using Recursion)

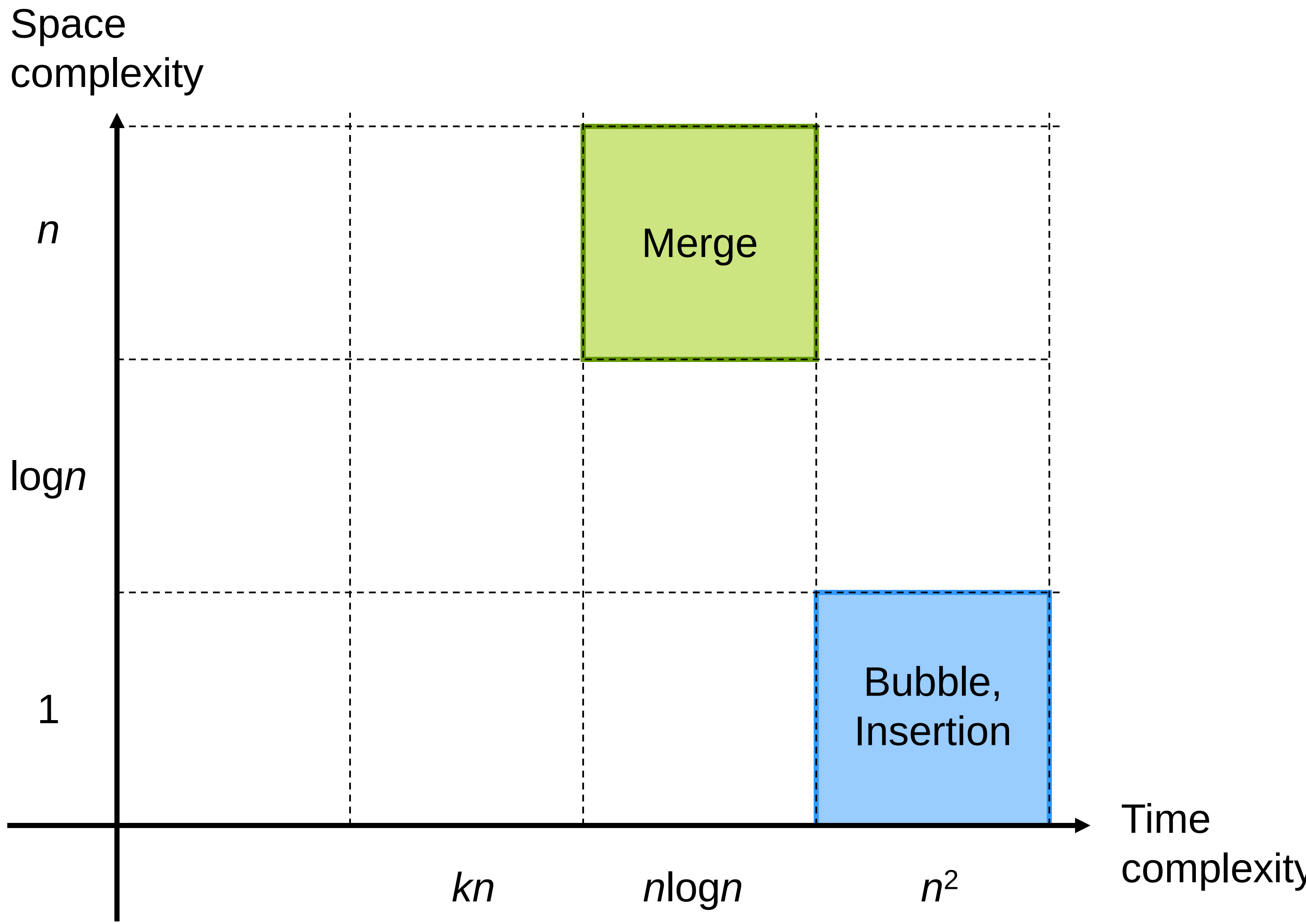
```
void mergesort(int data[], int first, int last) {
    int mid = (first + last ) / 2;
    if (first >= last) return;          //base case: size = 1
    mergesort(data, first, mid);    //recursion: divide the list into halves
    mergesort(data, mid+1, last);//recursion: divide the list into halves
    merge(data, first, mid, last);  //start merging the list: conquer
}
int main(…) {
    int data[] = {8, 5, 9, 6, 3};
    mergesort(data, 0, 4);
    return 0;
}
```

Complexity Analysis

- Merge sort goes through the same steps - independent of the data
  - Best case = Worst case = Average case
- For each runs, it requires O(n) time to finish
- There are $\log_2 n$ runs in total
- The time complexity is O($n\log n$)
- Faster than **bubble sort** and **insertion sort**!

- The trade-off is it needs extra memory to hold the temporary sorted result
- Space complexity = O(n)
- Improvement to the merge algorithm:
  - Instead of merging each set of lists from data[] to temp[] and then copy temp[] back to data[], alternate merge passes can be performed from data[] to temp[] and from temp[] to data[].

Summary


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

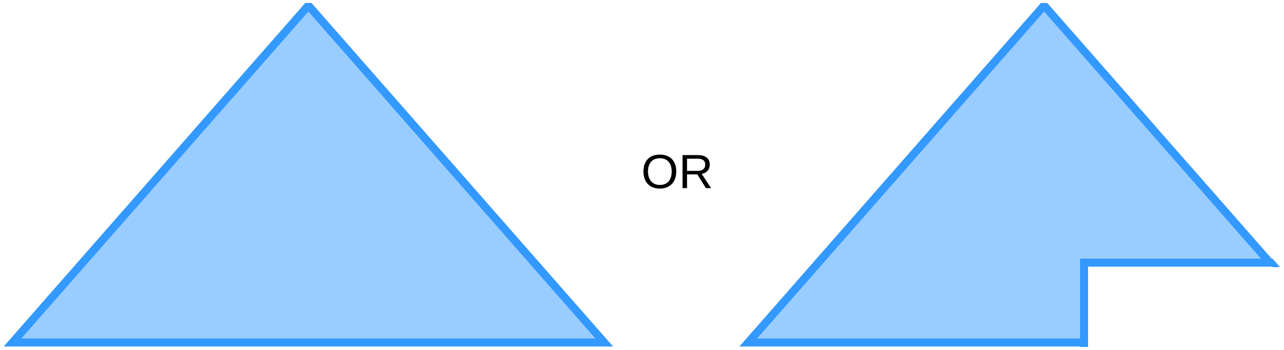
Heapsort

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

Space Complexity: $O(1)$

Heap Revision

- Max. heap tree is a binary tree with 2 properties
  - **Property 1**: The tree is complete
  - **Property 2**: The tree is descending



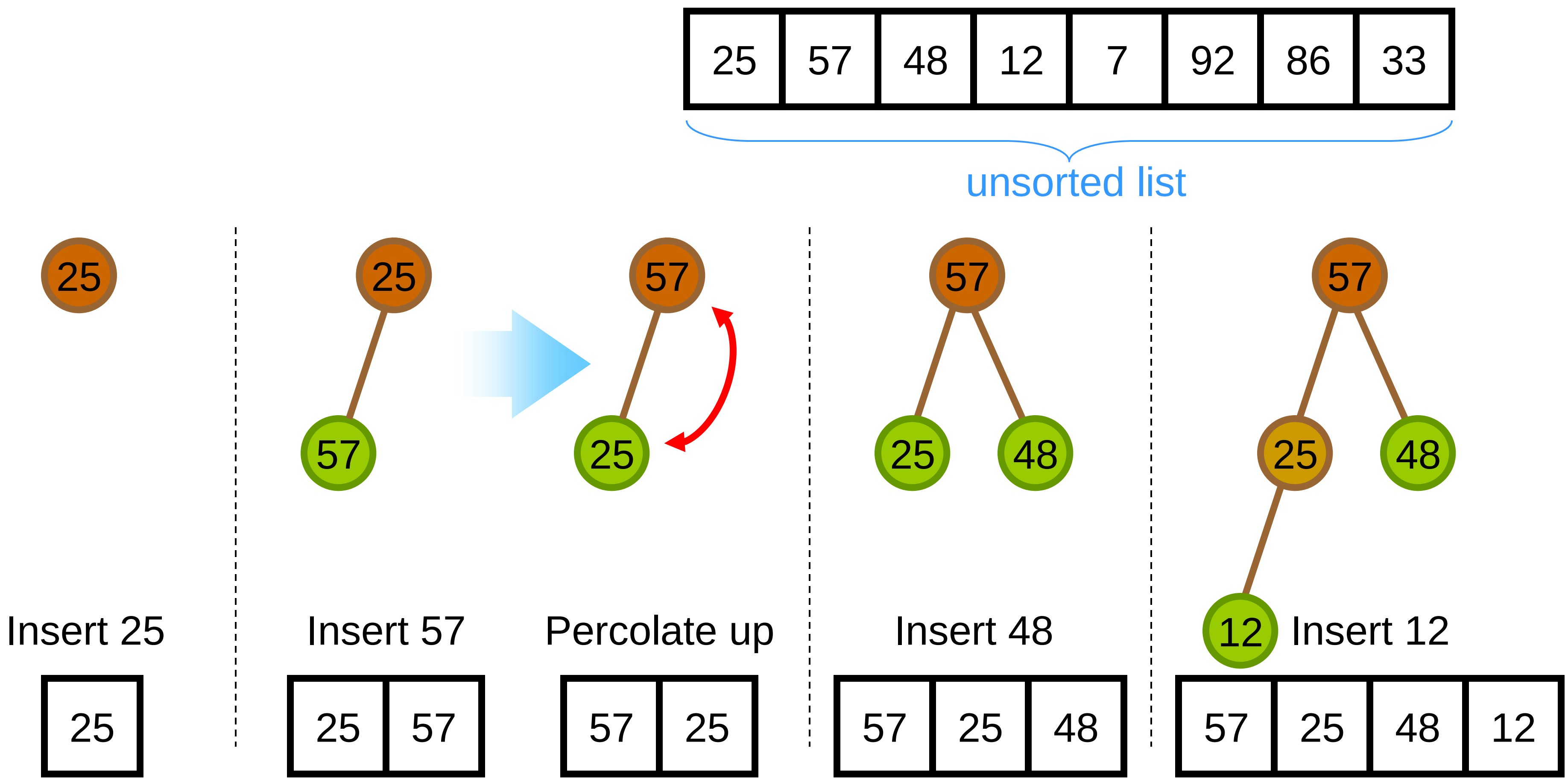
Complete binary tree

The Heapsort Algorithm

- Phase 1) Build Heap
  - Organize the input array as a max heap
- Phase 2) Swap Node
  - 1st pass
    - Swap the root (max node) with the last unsorted element
    - Now the original root (max node) has been sorted
    - Percolate down the new root if it is not the next-largest element
    - This puts the next-largest element into the root position
  - 2nd and the forth coming passes
    - Swap the next-largest element with the last unsorted element
    - Repeat until all nodes are sorted

Heapsort Example: Phase 1

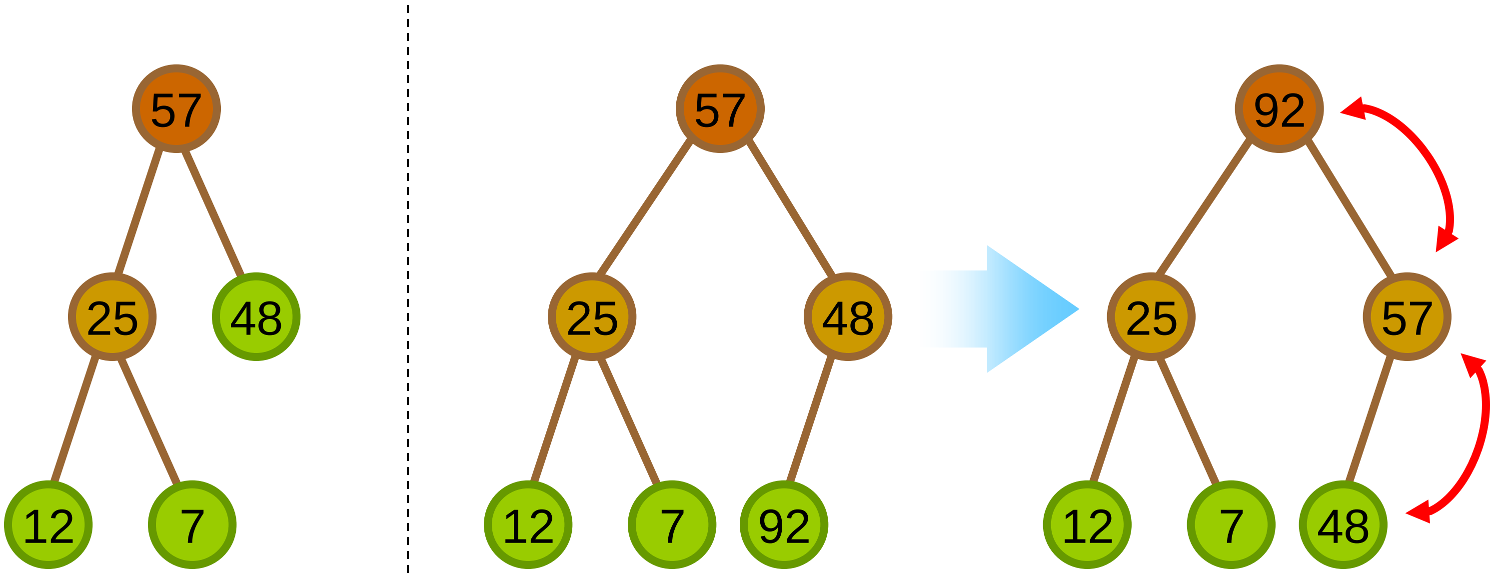
- Phase 1) Build the max. heap tree



Heapsort Example: Phase 1

25	57	48	12	7	92	86	33

unsorted list



Insert 7

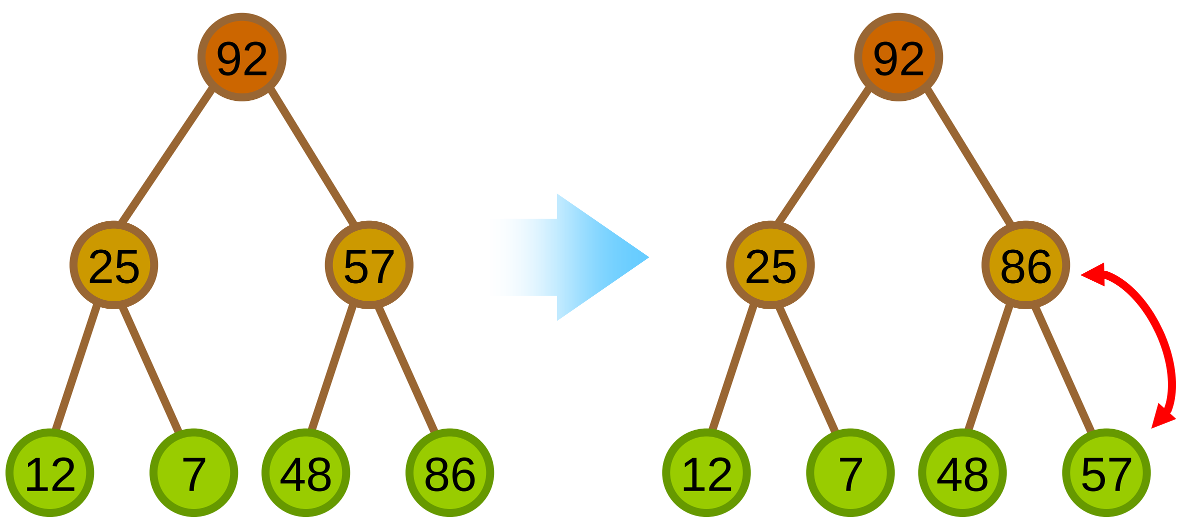
Insert 92

Percolate up twice

Heapsort Example: Phase 1

25	57	48	12	7	92	86	33

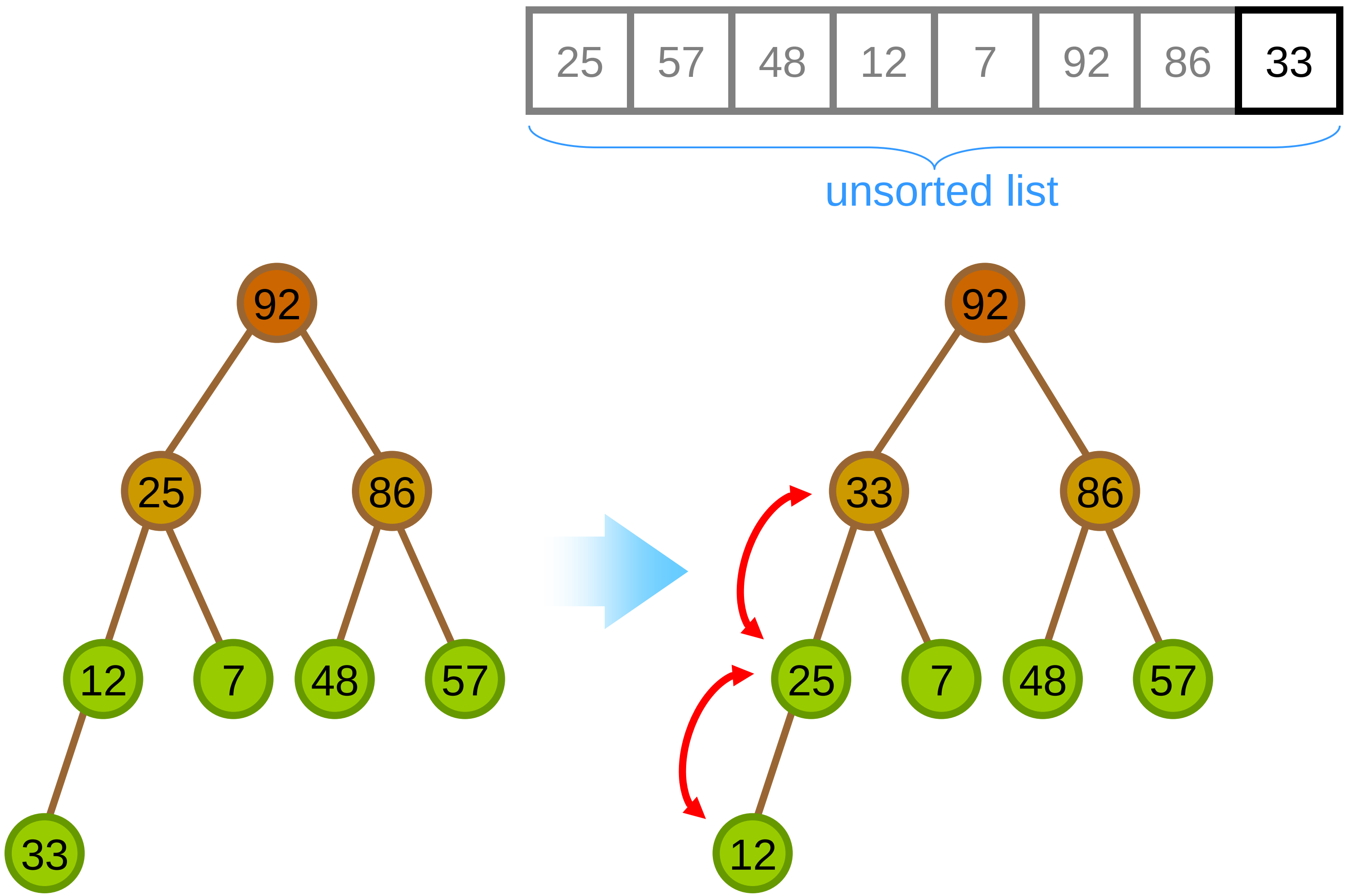
unsorted list



Insert 86

Percolate up

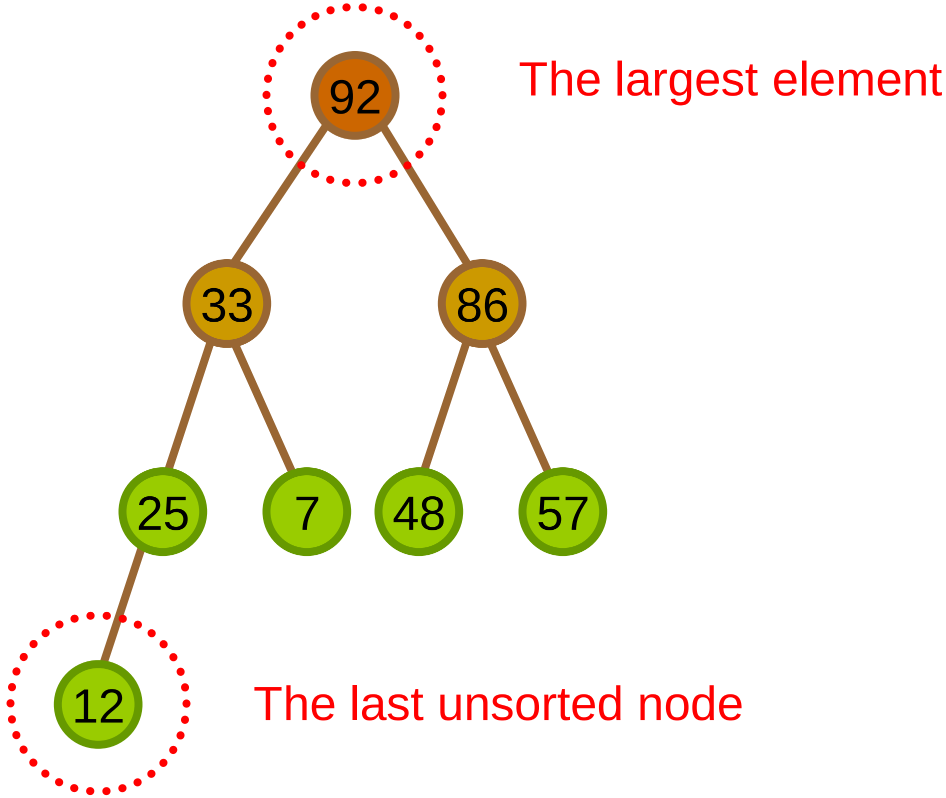
Heapsort Example: Phase 1



Insert 33

Percolate up twice

Heapsort Example: Phase 1

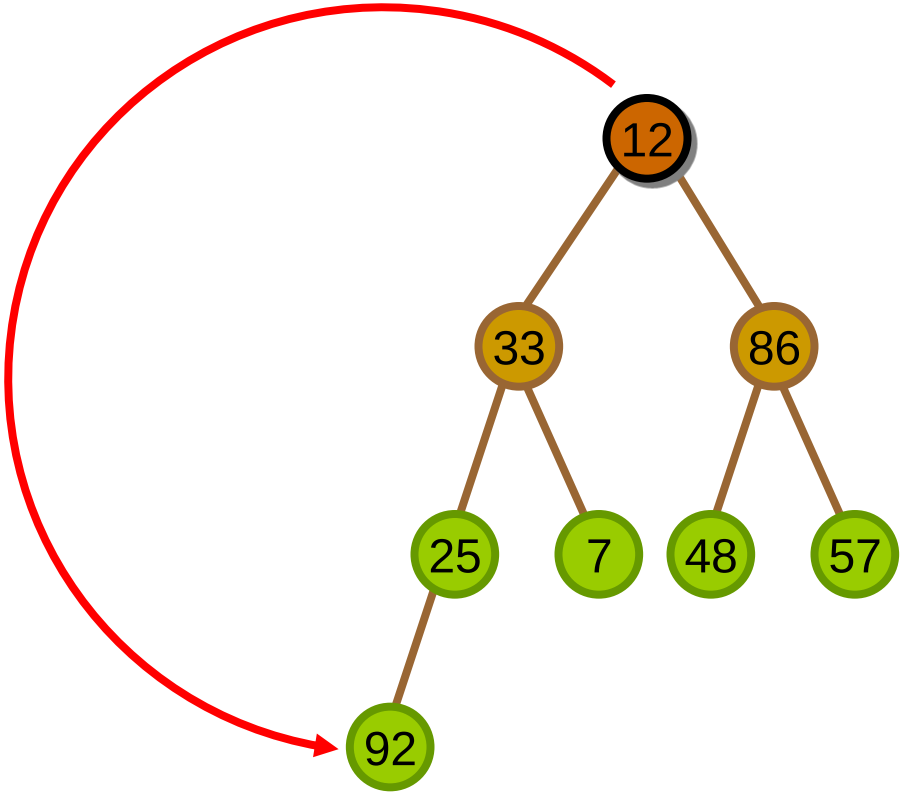


Heap array

92	33	86	25	7	48	57	12

unsorted list

Phase 2) 1st Pass



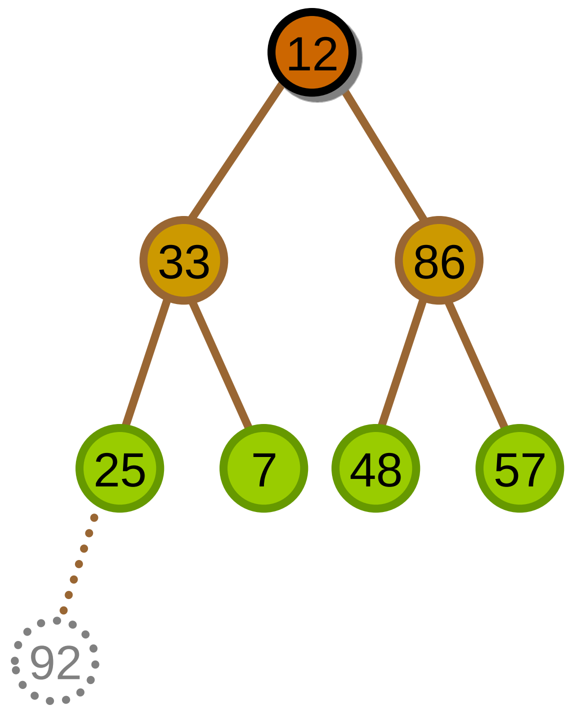
Swap root with the last unsorted node

Heap array

12	33	86	25	7	48	57	92

unsorted list   sorted list

Phase 2) 1st Pass



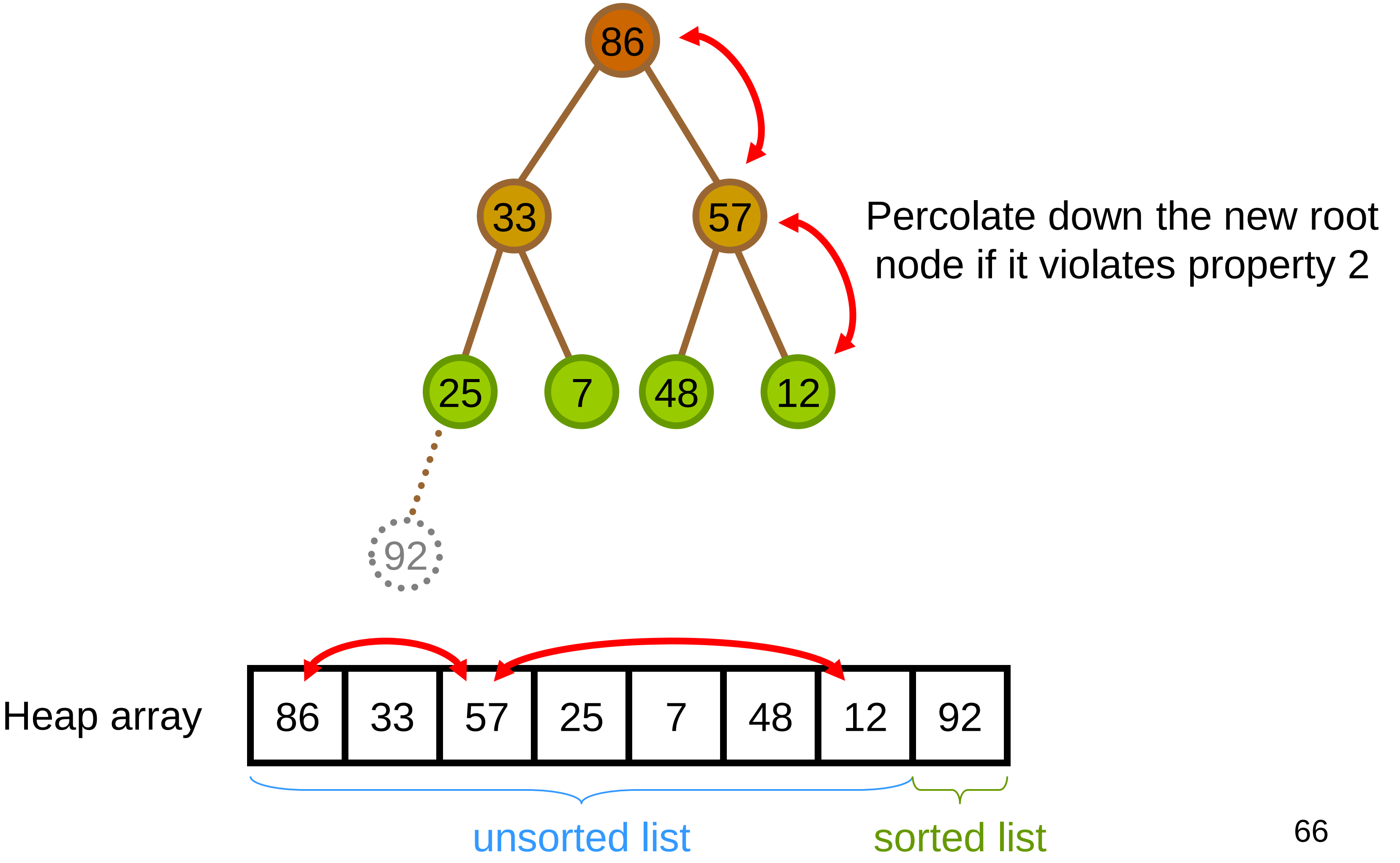
The original root (max node) can now be ignored

Heap array

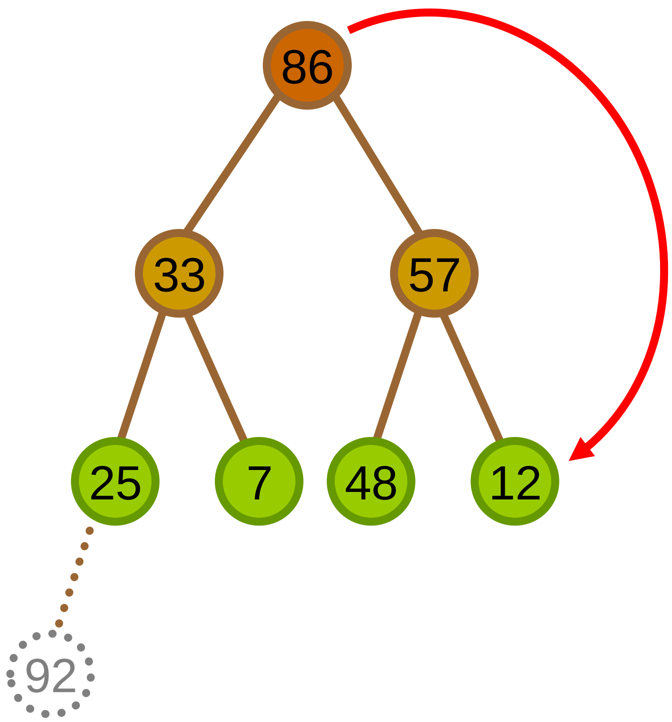
12	33	86	25	7	48	57	92

$\underbrace{12\ 33\ 86\ 25\ 7\ 48\ 57}_{\text{unsorted list}}\ \underbrace{92}_{\text{sorted list}}$

Phase 2) 1st Pass



Phase 2) 2nd Pass



Swap new root with the last unsorted node

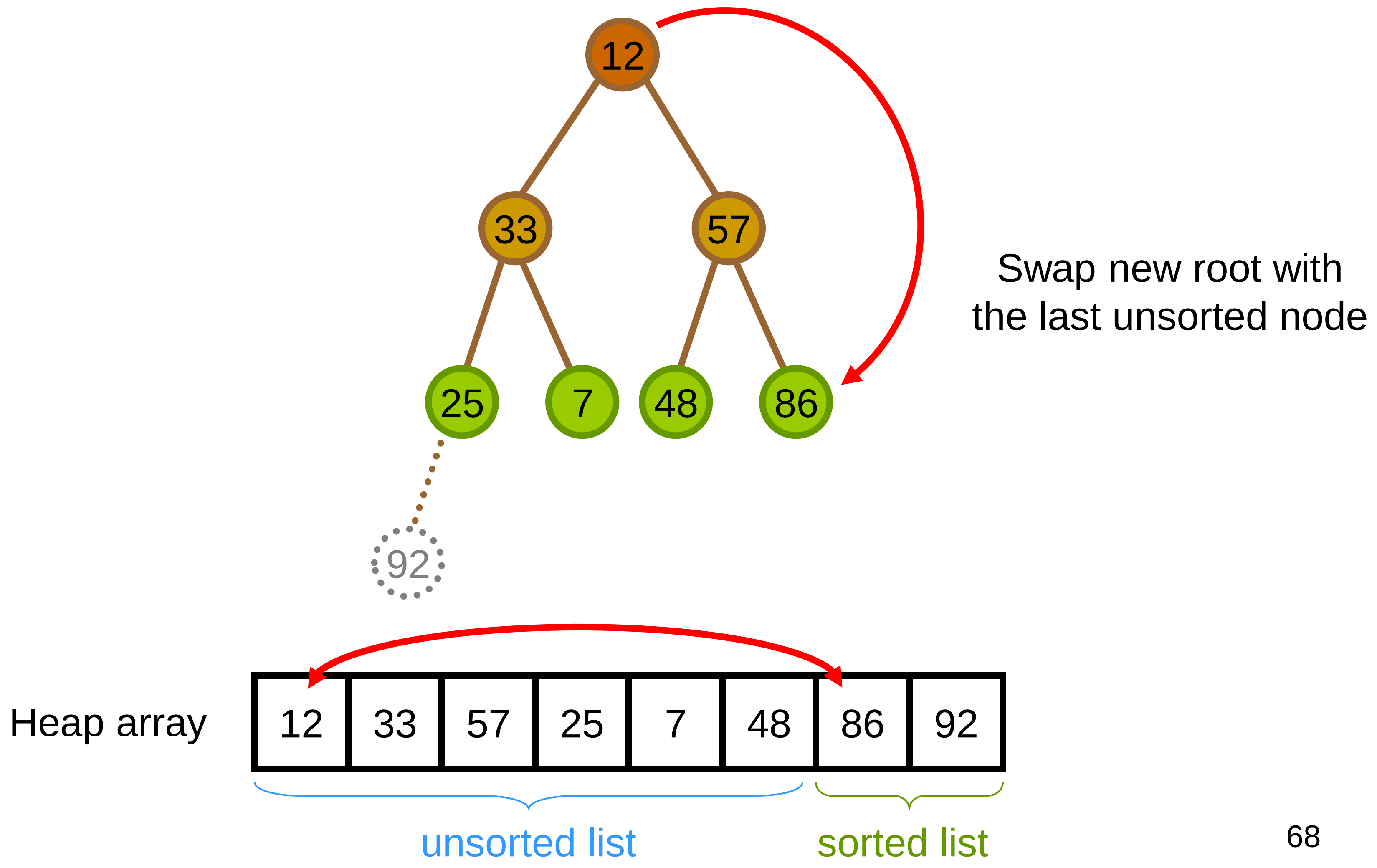
Heap array

86	33	57	25	7	48	12	92

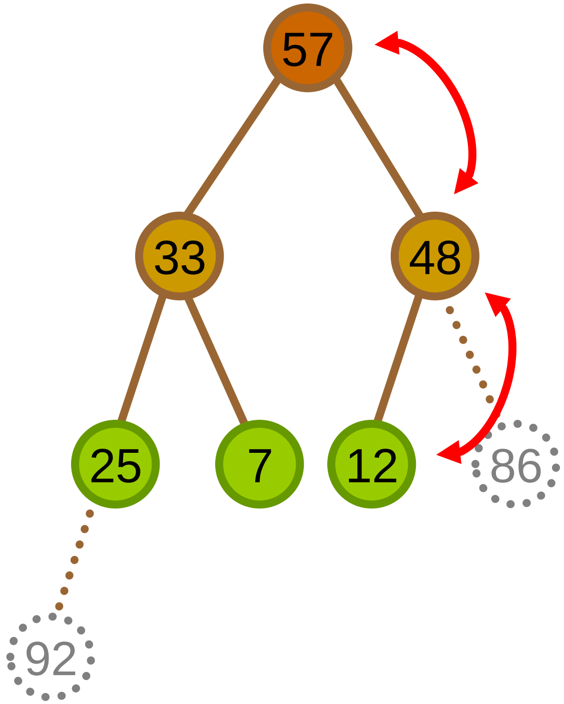
unsorted list

sorted list

Phase 2) 2^{nd} Pass

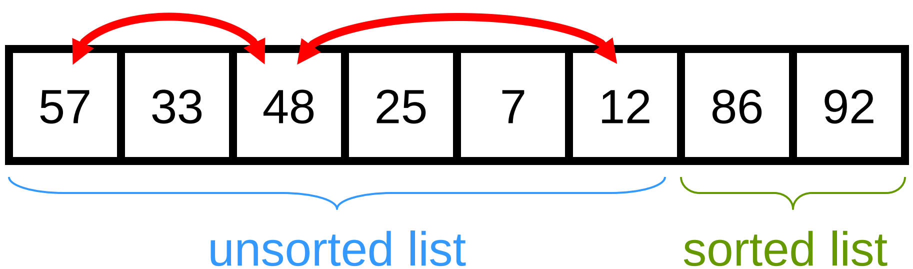


Phase 2) 2^{nd} Pass

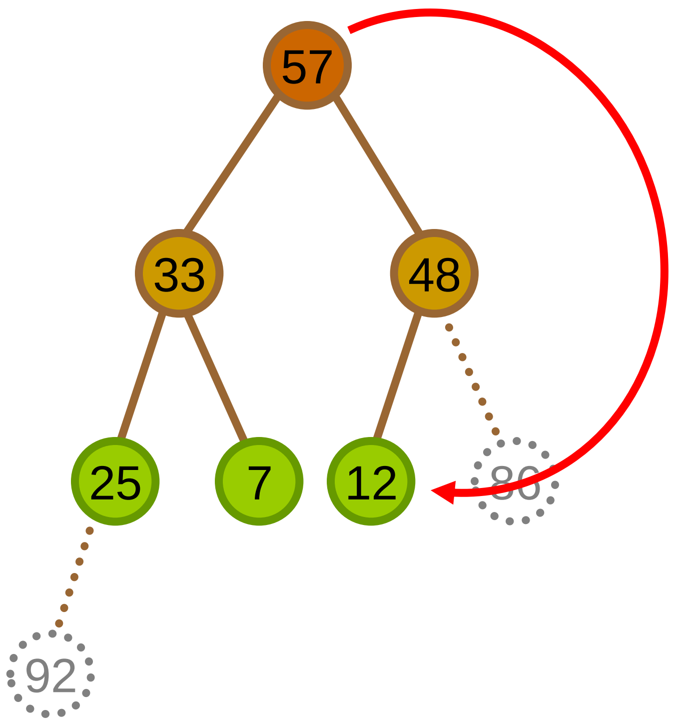


Percolate down the new root node

Heap array



Phase 2) 3rd Pass



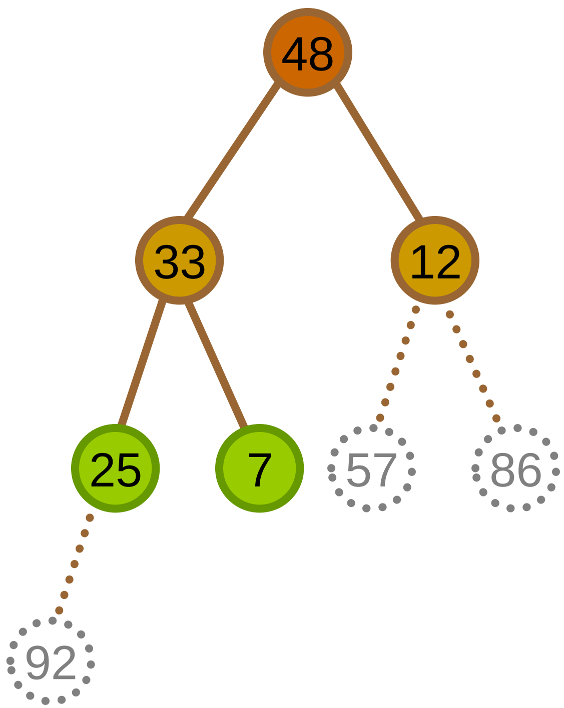
And so on…

Heap array

57	33	48	25	7	12	86	92

unsorted list sorted list

Phase 2) After 3rd Pass

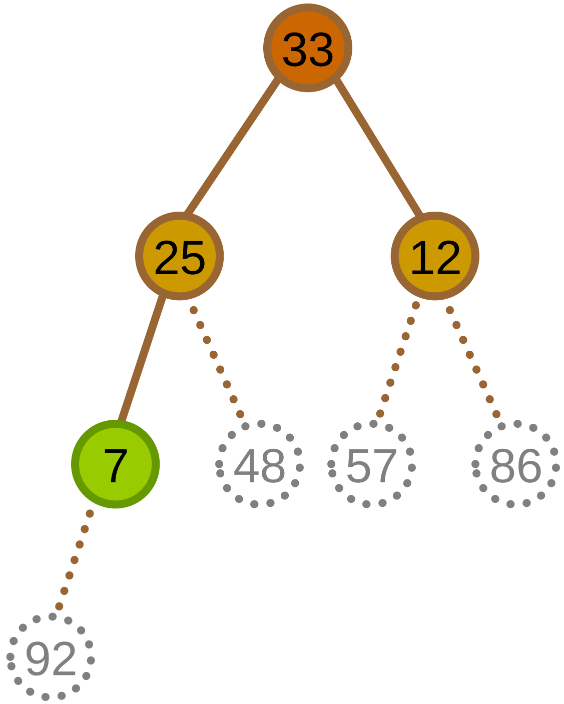


Heap array

48	33	12	25	7	57	86	92

unsorted list sorted list

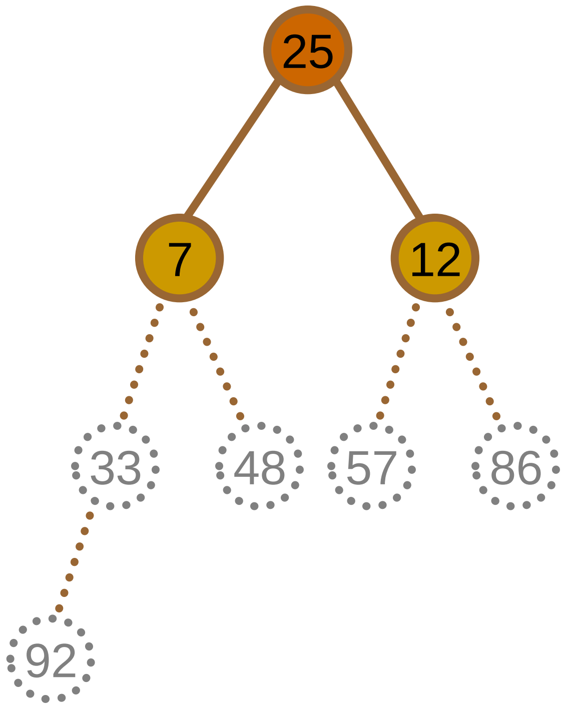
Phase 2) After 4th Pass



Heap array								
	33	25	12	7	48	57	86	92

unsorted list   sorted list

Phase 2) After 5th Pass

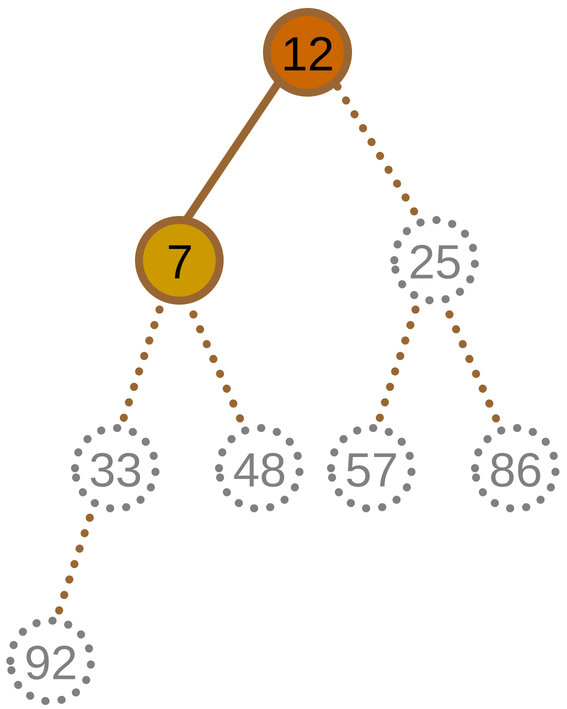


Heap array

25	7	12	33	48	57	86	92

unsorted list (25, 7, 12) — sorted list (33, 48, 57, 86, 92)

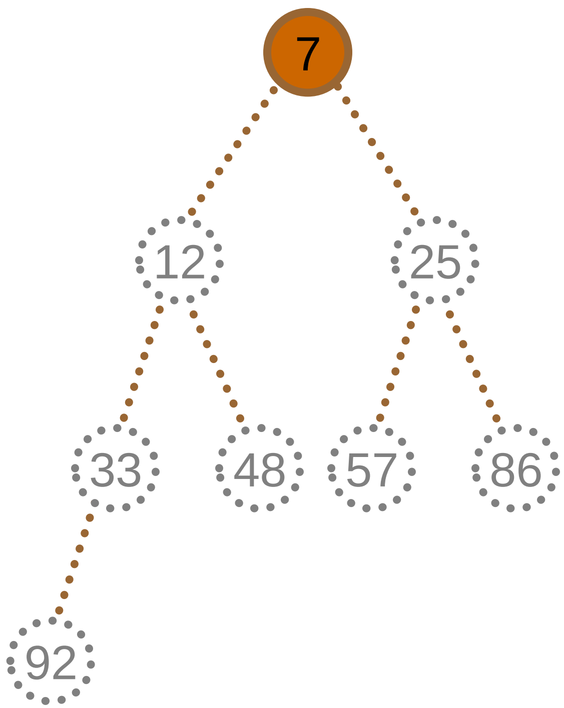
Phase 2) After 6th Pass



Heap array								
Heap array	12	7	25	33	48	57	86	92

unsorted list | sorted list

Phase 2) After 7th Pass



Until only one node left

Heap array

7	12	25	33	48	57	86	92

$\underbrace{\hspace{20em}}_{\text{sorted list}}$

Complexity Analysis

- Time to build the heap tree
  - Suppose there are n nodes
  - The depth of the tree is $\log_2 n$
  - So at most $\log_2 n$ comparison for each percolate up
  - Total $n \cdot \log_2 n$
- Time to sort the data
  - About $\log_2 n$ time for each percolate down process
  - Total $(n\text{-}1)\log_2 n$
- Time complexity: $O(n \cdot \log n)$
  - Go through the same steps in the second phase (percolate down)
  - Best case = Worst case = Average case
- Extra space is required for swapping the nodes
  - Space Complexity: O(1)

Heapsort (Recursive Version)

```
void percolateUp(int data[], int index) {
    int parent = (index - 1) / 2;

    if (parent < 0) return;                  //base case
    //note: if parent >= 0, index also >= 0

    if (data[index] > data[parent]) {        //general case
        swap(&data[index], &data[parent]);
        percolateUp(data, parent);
    }
}
```

Heapsort (Recursive Version)

```
void percolateDown(int data[], int n, int index) {
    int left, right, maxIndex;

    if (index < 0 || index >= n) return;            //base case 1

    left = 2 * index + 1;
    right = left + 1;
    if (left >= n) return;                          //base case 2

    maxIndex = right < n && data[left] < data[right] ? right : left;
    if (data[index] < data[maxIndex]) {             //general case
        swap(&data[index], &data[maxIndex]);
        percolateDown(data, n, maxIndex);
    }
}
```

Heapsort

```
void heapsort(int data[], int n) {
    int i, last;
    for (i = 1; i < n; i++) {              //start from index 1
        percolateUp(data, i);              //build the max. heap tree
    }
    for (last = n – 1; last > 0; last--) {
        swap(&data[0], &data[last]);       //sort the sequence
        percolateDown(data, last, 0);
    }
}
```

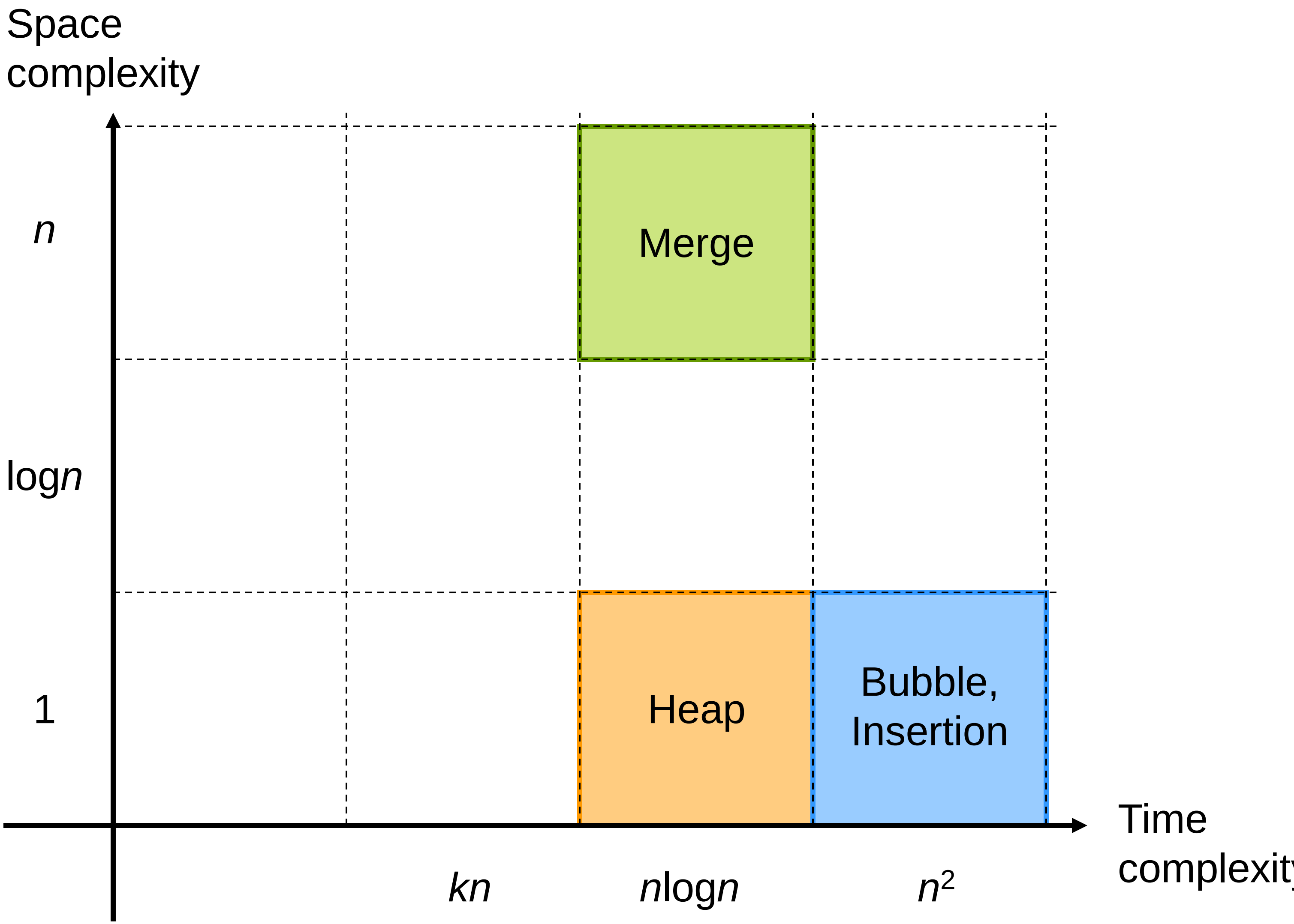
Heapsort (Iterative Version)

```
void percolateUp(int data[], int index) {

}
```

Heapsort (Iterative Version)

```
void percolateDown(int data[], int n, int index) {

}
```

Summary

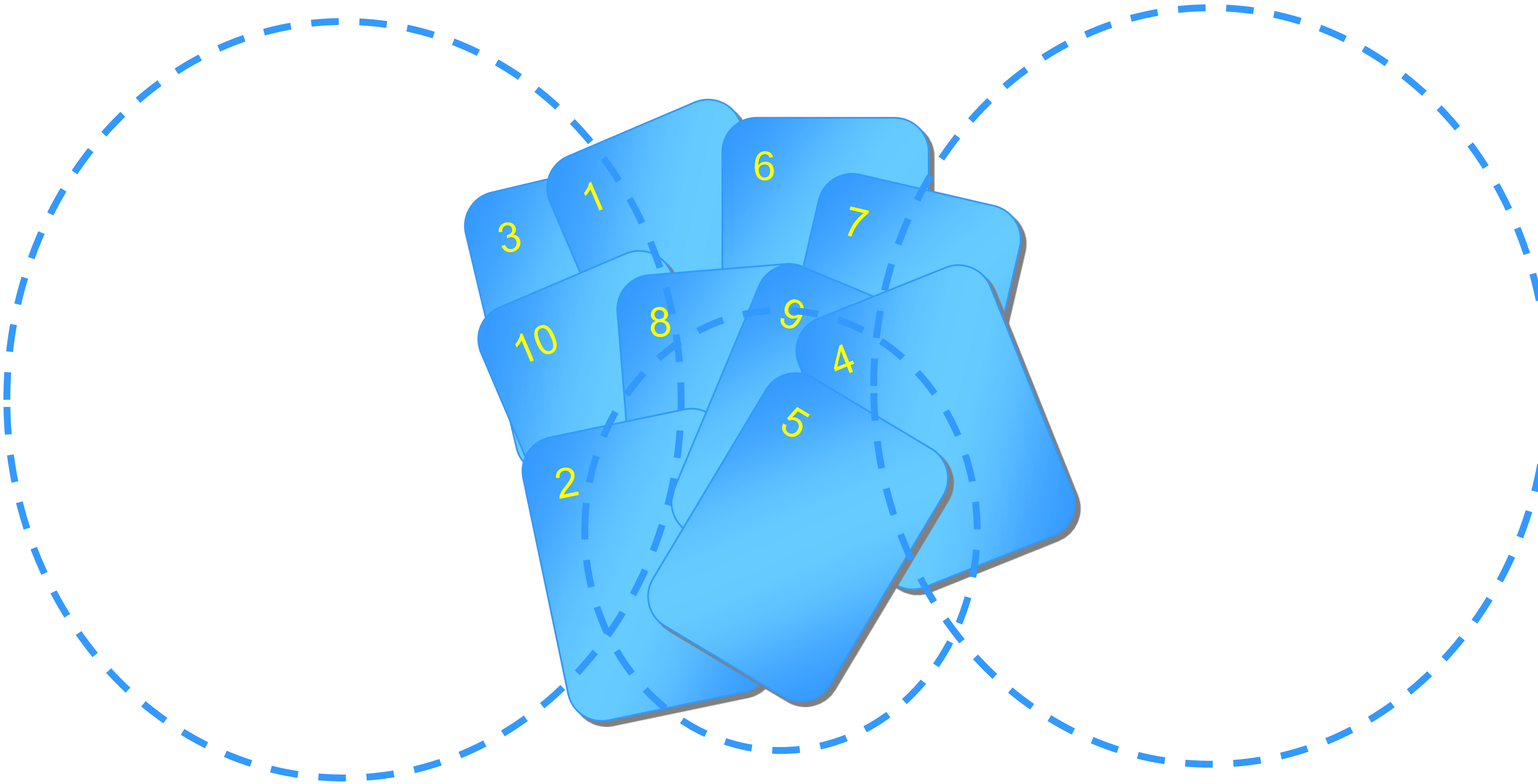

Space
complexity
n
logn
1
Merge
Heap
Bubble,
Insertion
kn
nlogn
n2
Time
complexity

Quicksort

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

Space Complexity: O(log$n$)

Quicksort



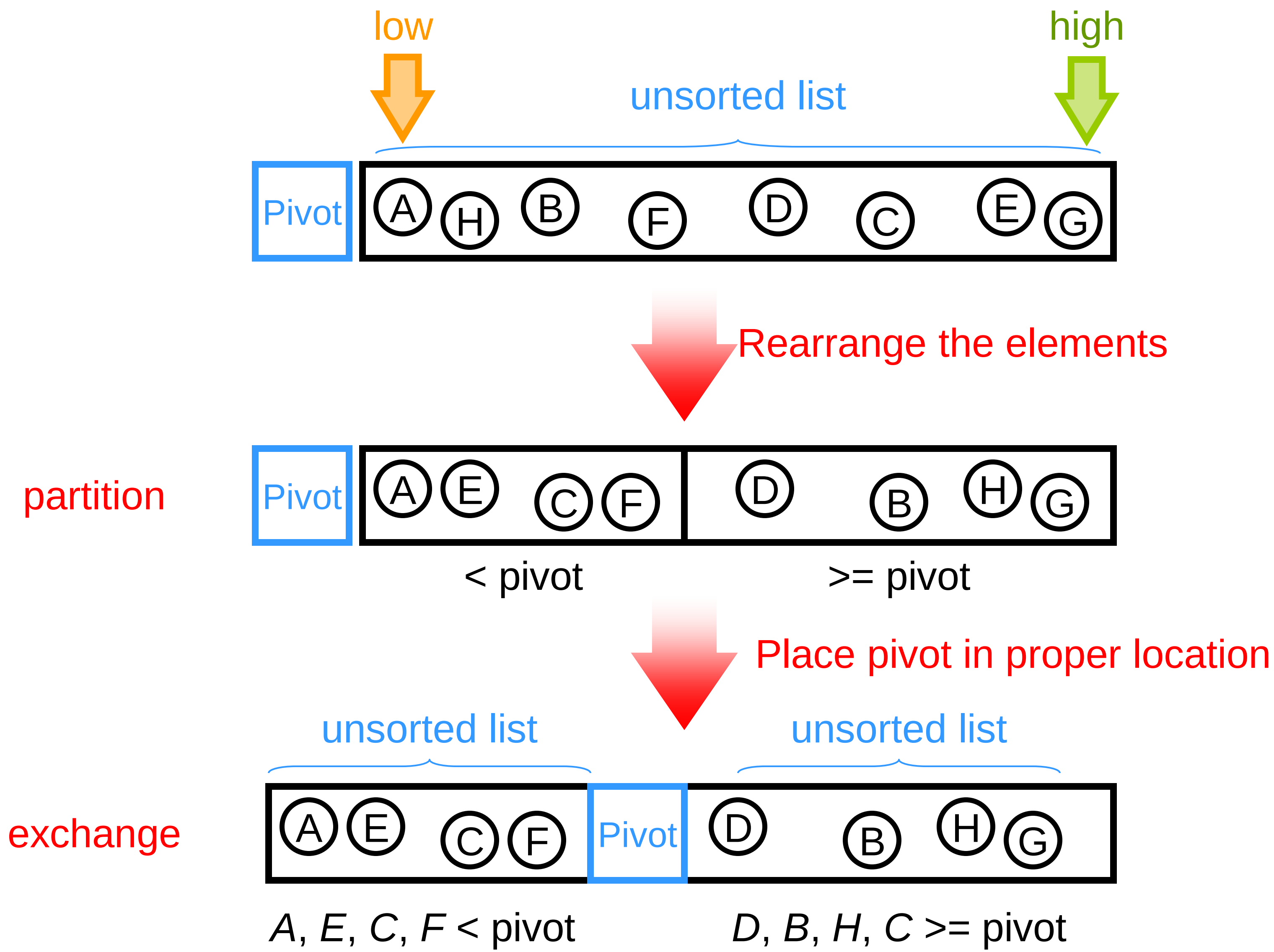
Left < pivot ≤ Right

(recursion) (recursion)

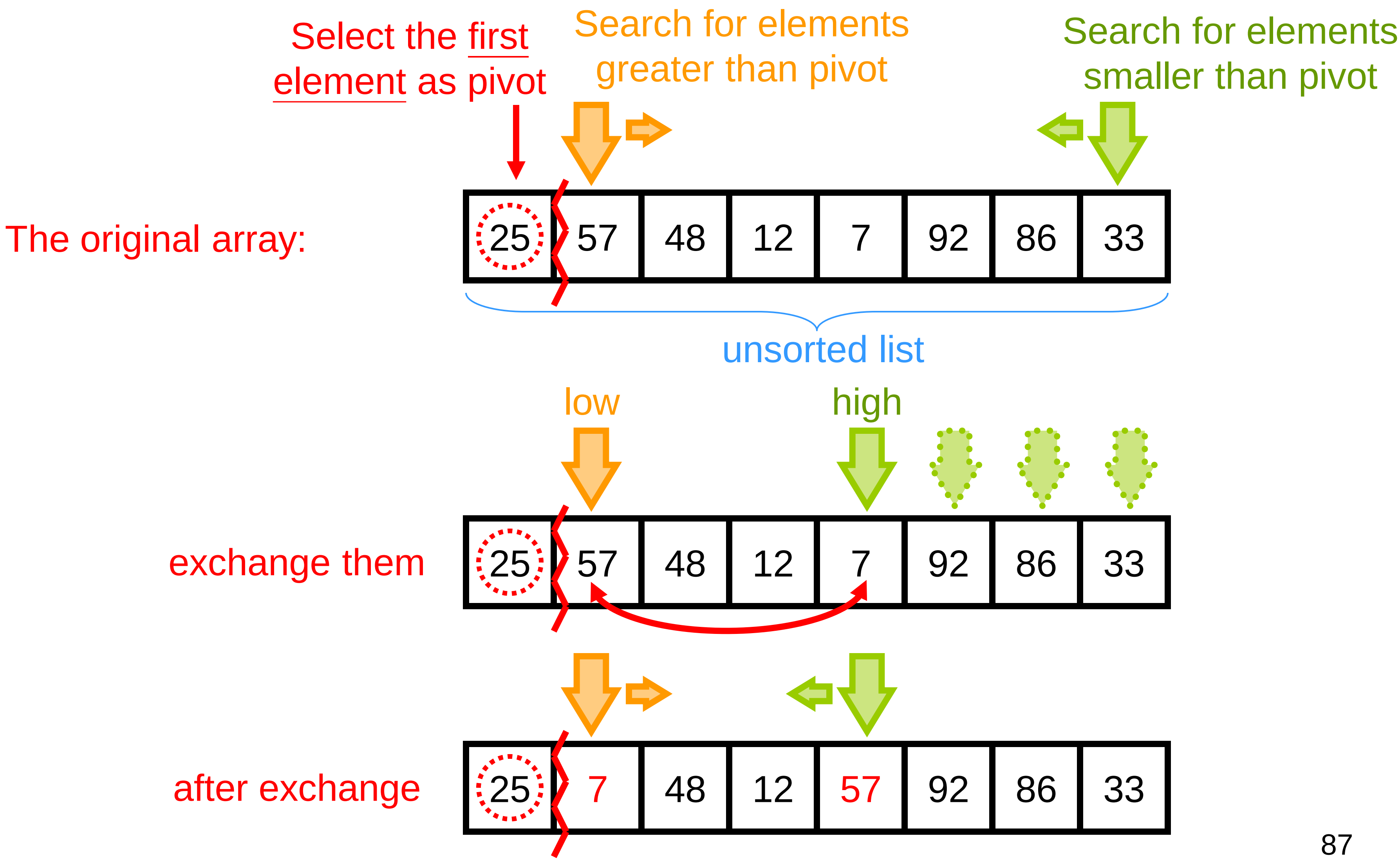
Exchange and Partition

- 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)

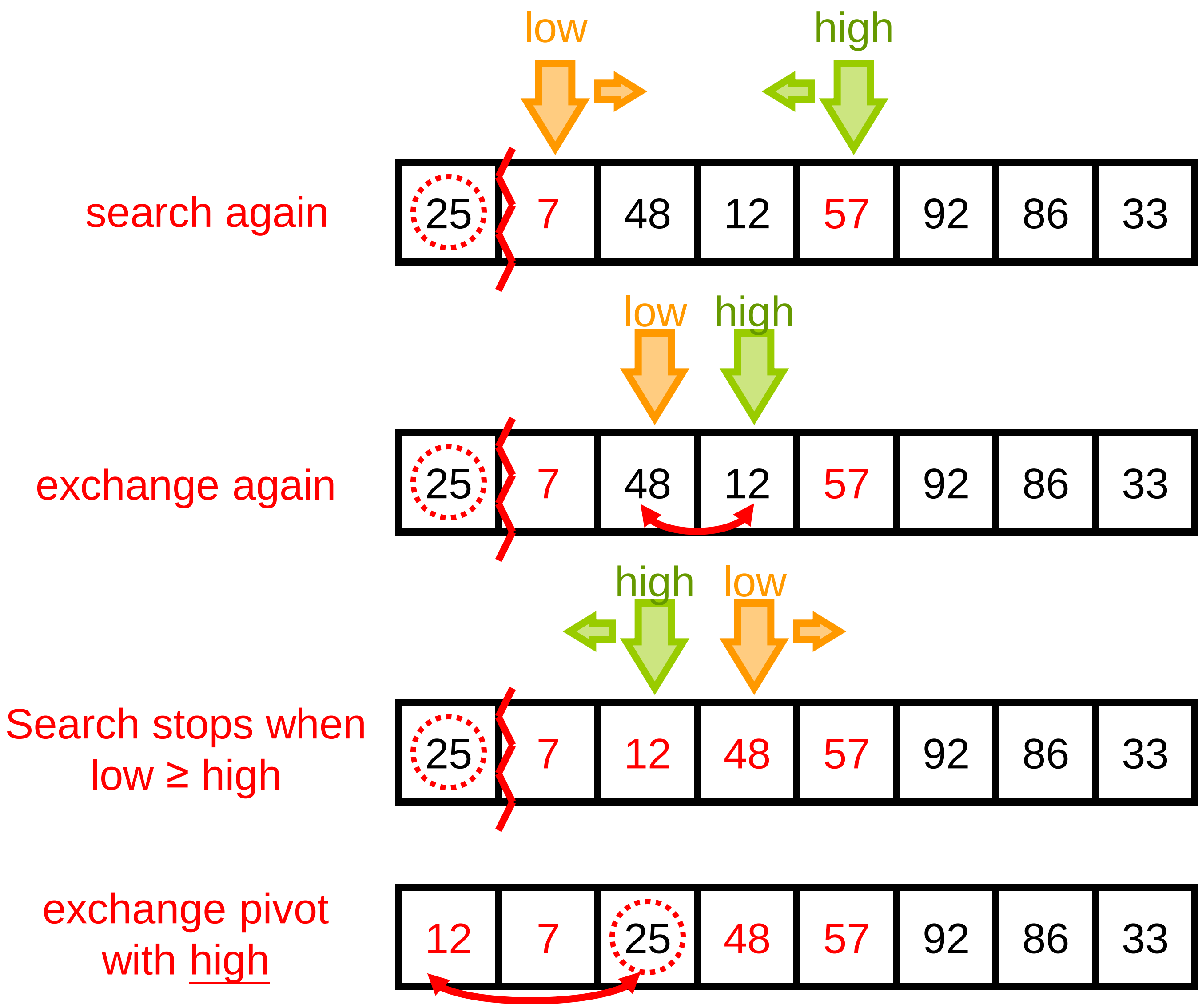
The General Concept



Quicksort Example

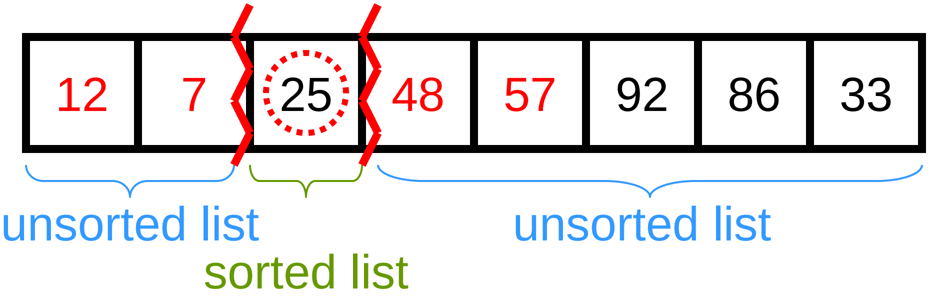


Quicksort Example



Quicksort Example

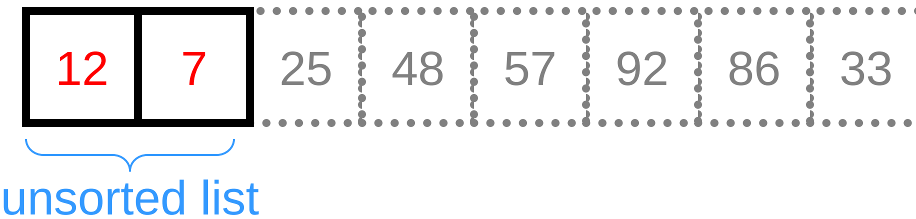
After 1st pass

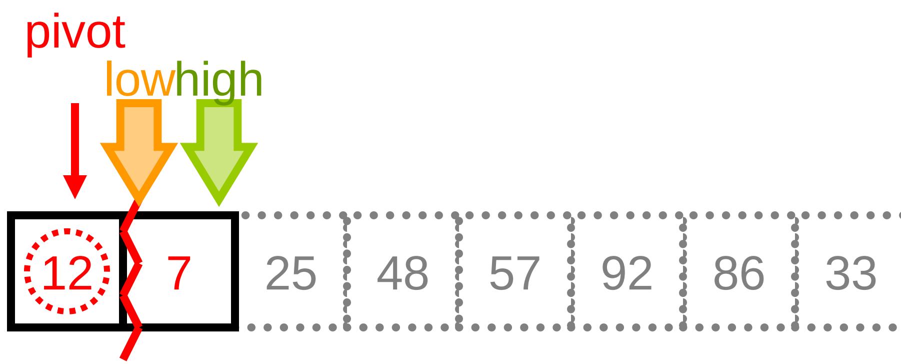


$$\underbrace{\{12, 7\}}_{\text{sort recursively}} \quad < \quad 25 \quad \leq \quad \underbrace{\{48, 57, 92, 86, 33\}}_{\text{sort recursively}}$$

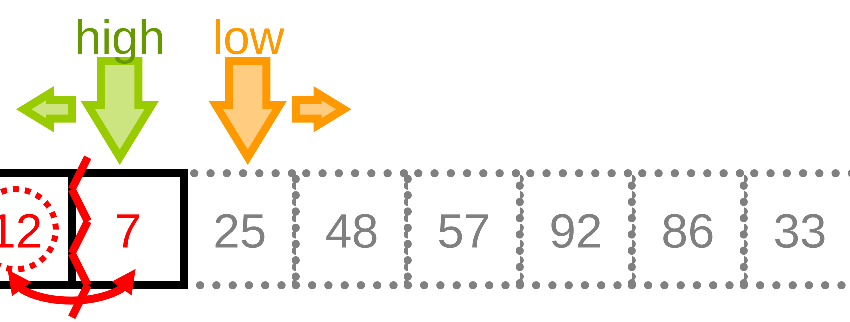
Sort the Left Sublist

Sort the left sublist



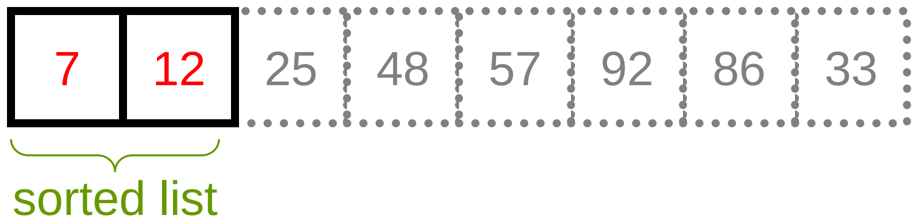


After searching, high will point to 7 (smaller than 12) and low will point out of the array

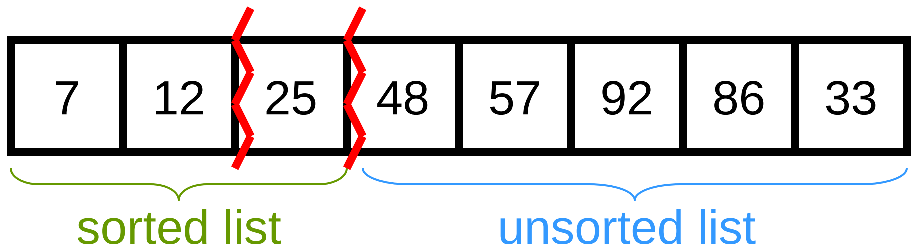


Sort the Left Sublist

Exchange pivot with high



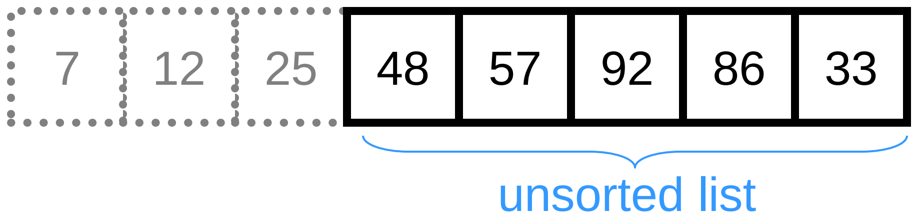
Combining the array



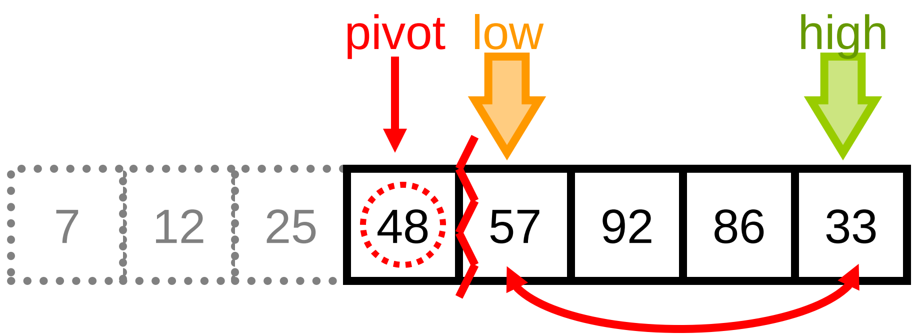
{7, 12, 25} ≤ {48, 57, 92, 86, 33}

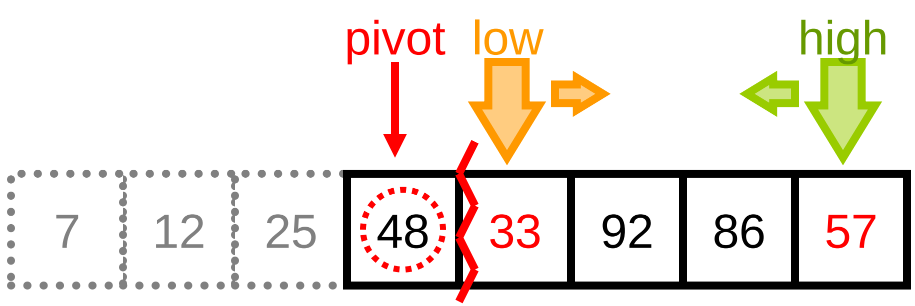
sort recursively

Sort the Right Sublist

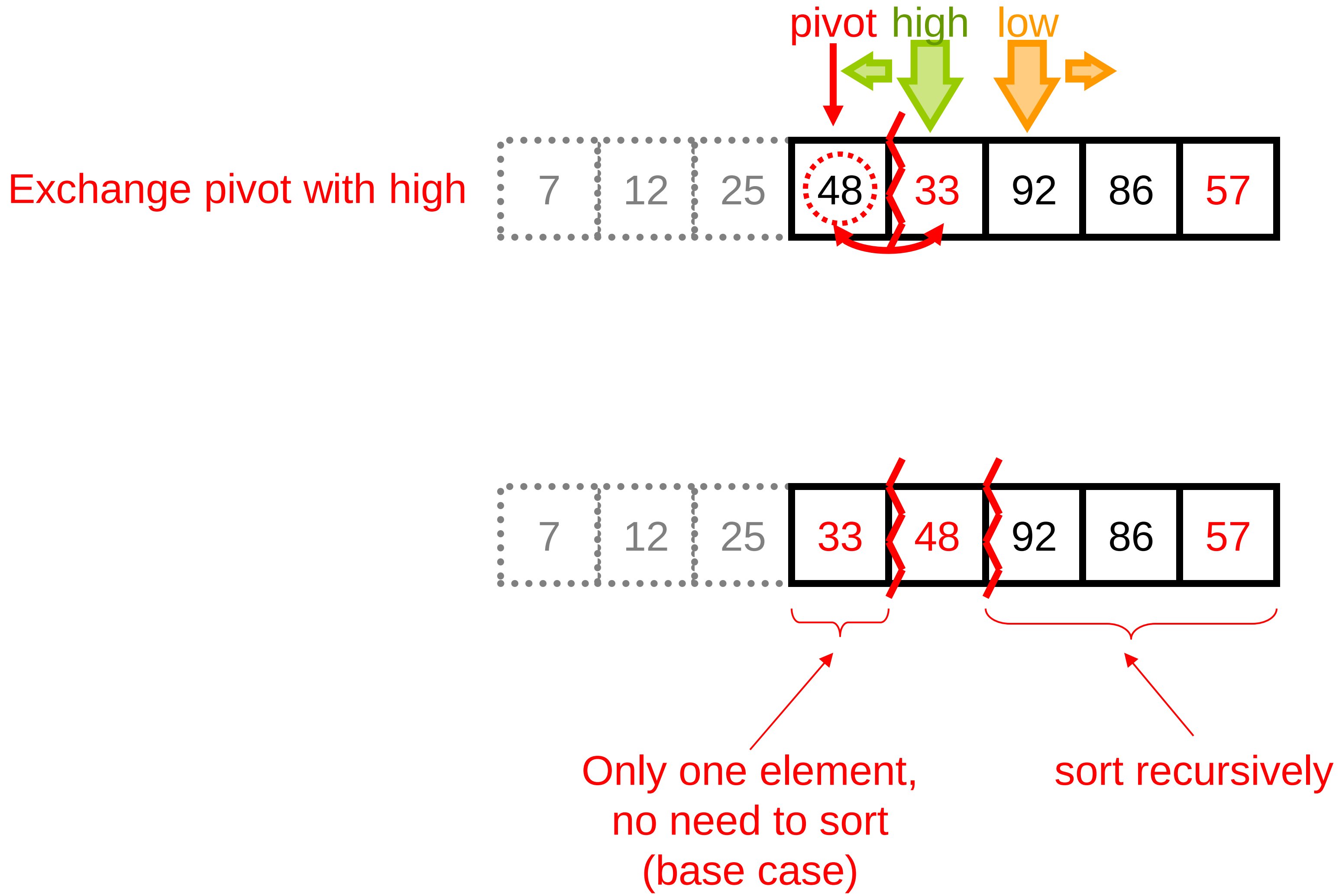


Search and exchange

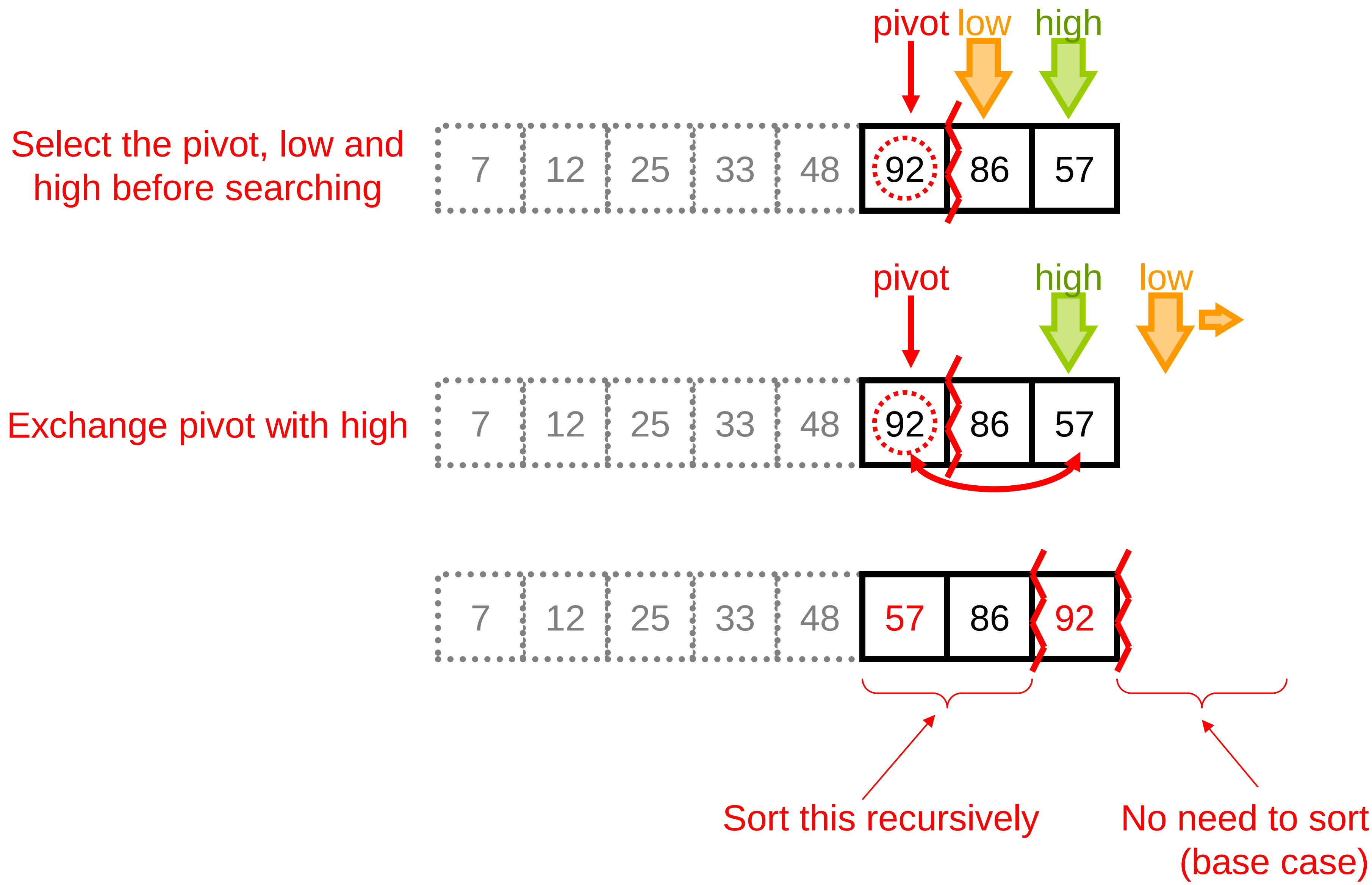




Sort the Right Sublist

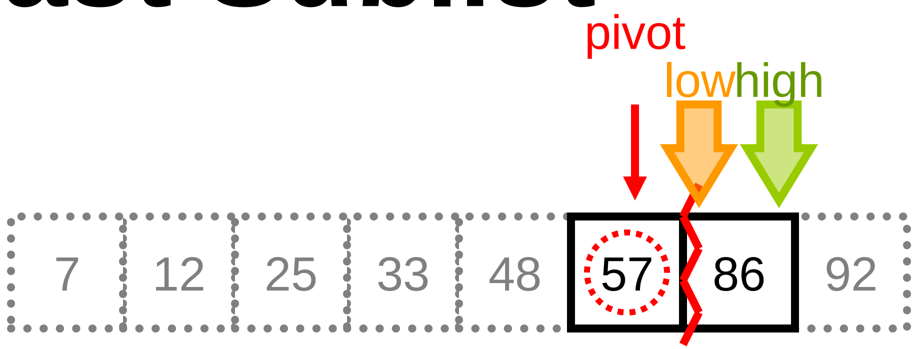


Sort Another Right Sublist

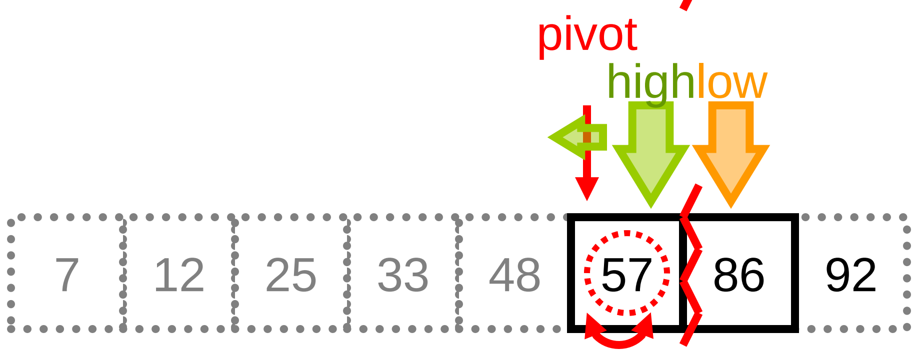


Sort the Last Sublist

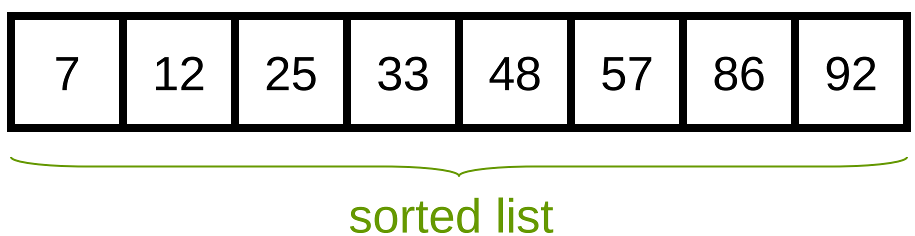
Select the pivot, low and high before searching



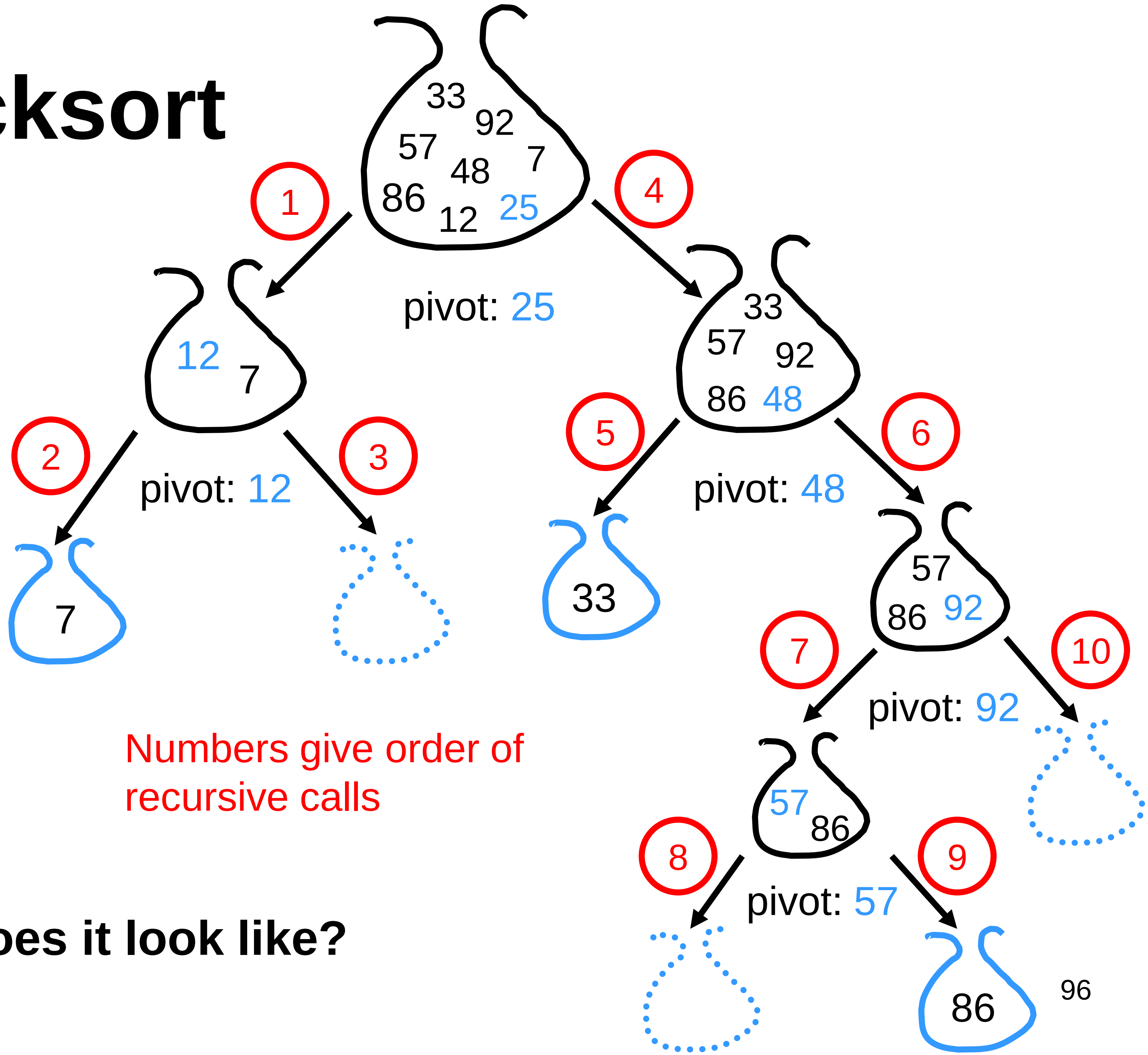
Exchange pivot with high (exchange with itself)



Finally, the list is sorted correctly



Quicksort



Numbers give order of recursive calls

What does it look like?

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

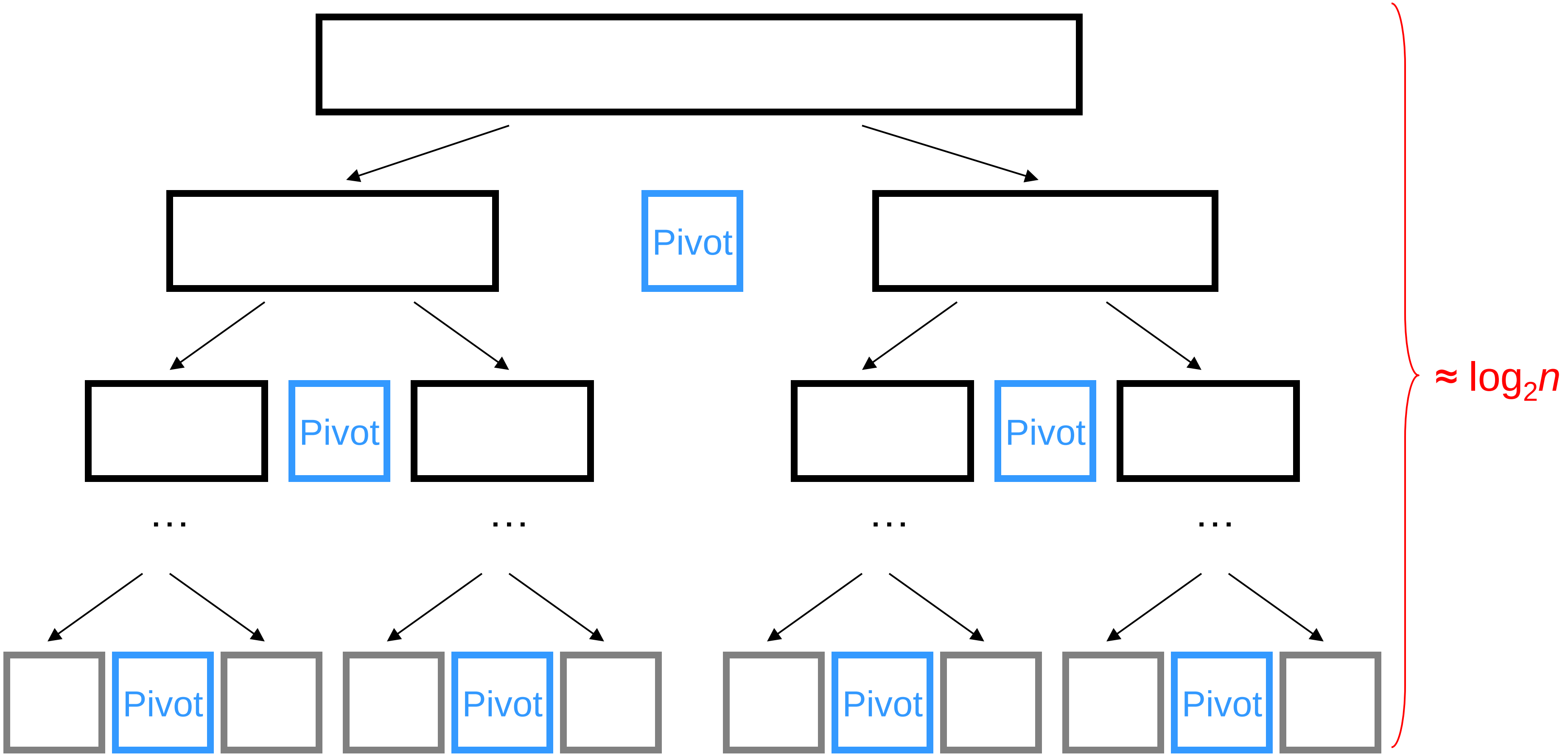
The Procedure

```
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);
}
```

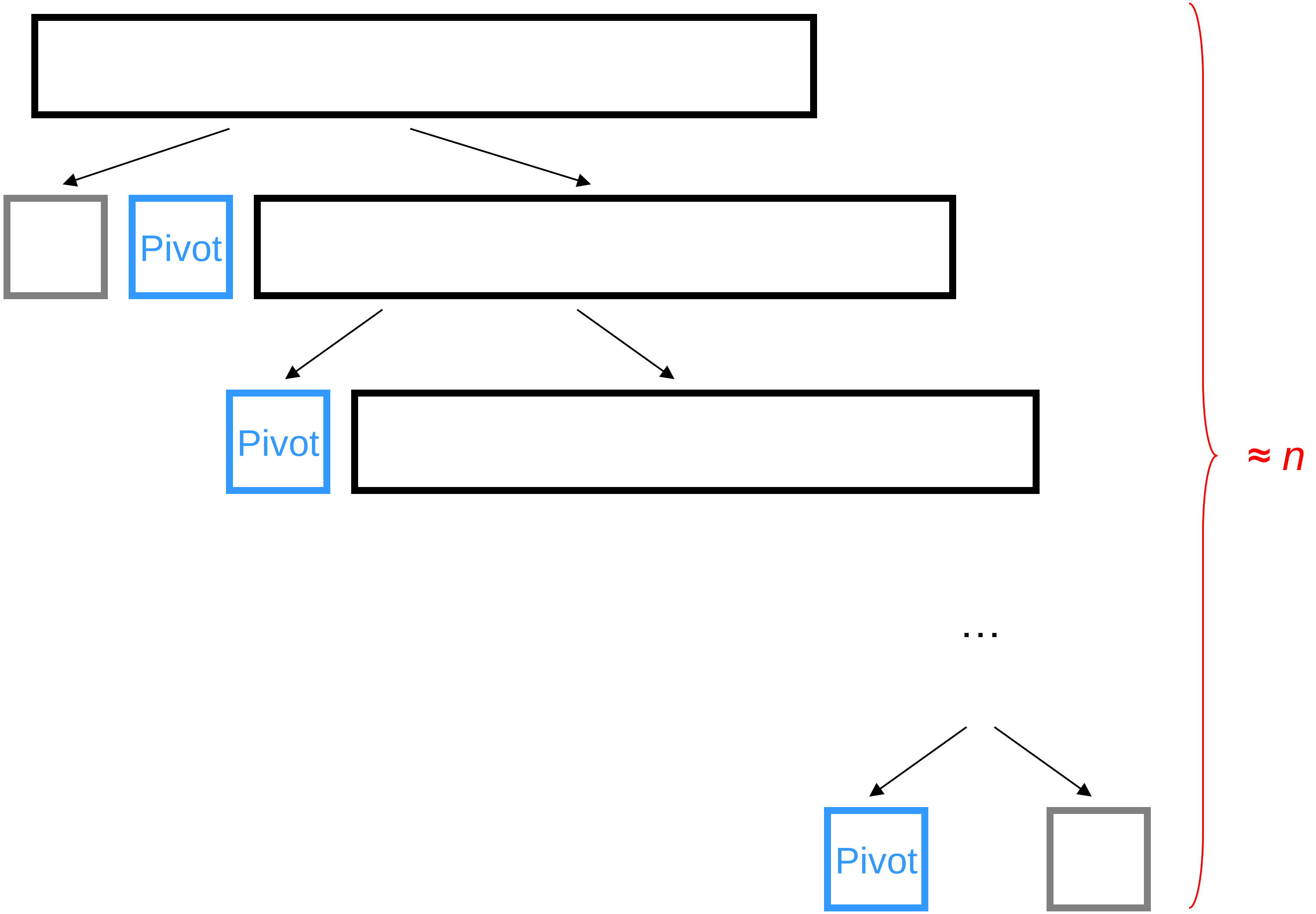
divide (exchange & partition) (iteration)

conquer (recursion)

A Good Pivot



A Bad Pivot



Complexity Analysis

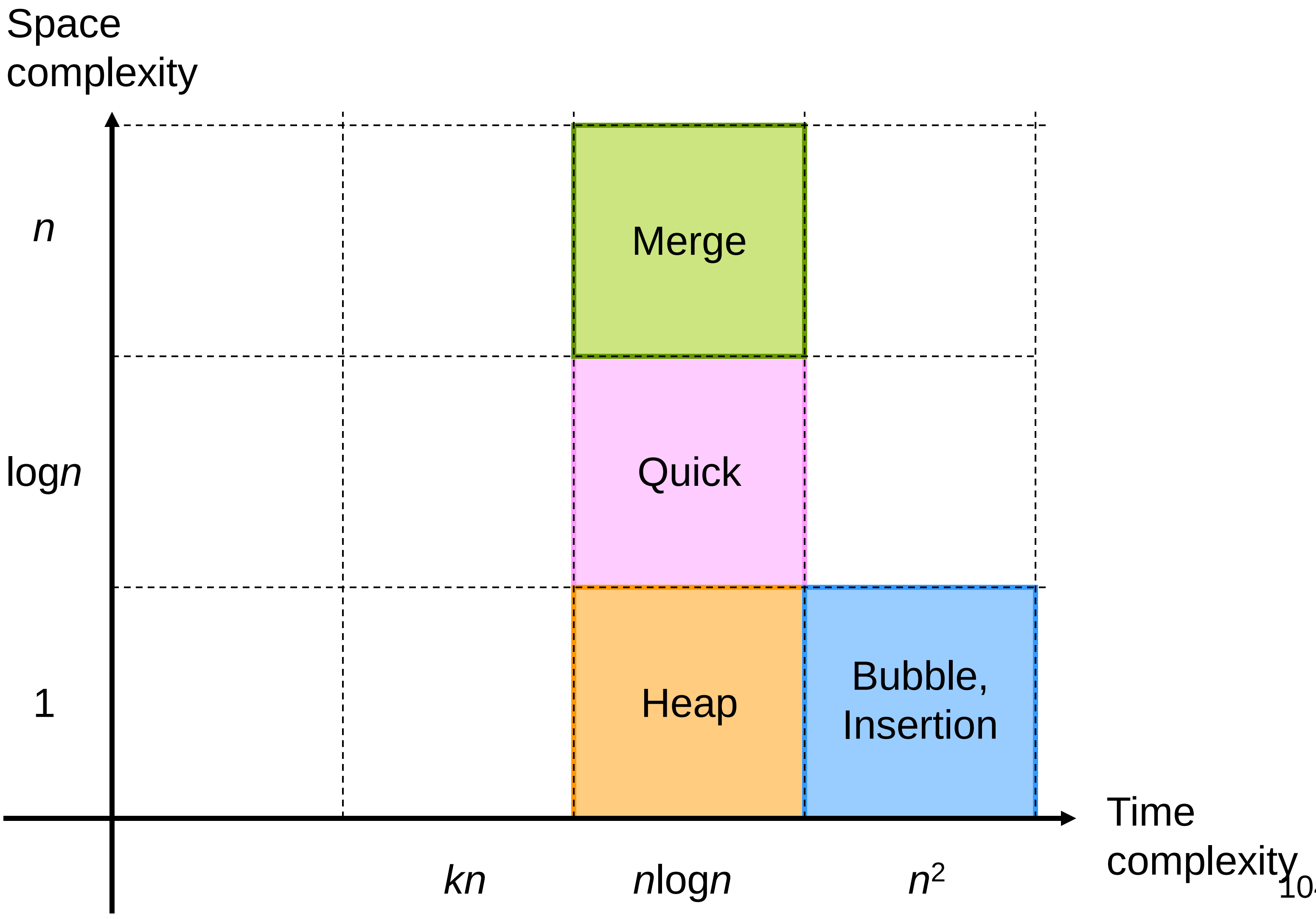
- Partition
  - Low pointer moves to right, while high pointer moves to left
  - Total $n - 1$ comparisons
  - $O(n)$: linear time
- Exchange
  - Swapping nodes: O(1)
- How many passes in total?
  - The best case
    - Ideally, the two sub-lists will be of equal size if the median is chosen as pivot 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

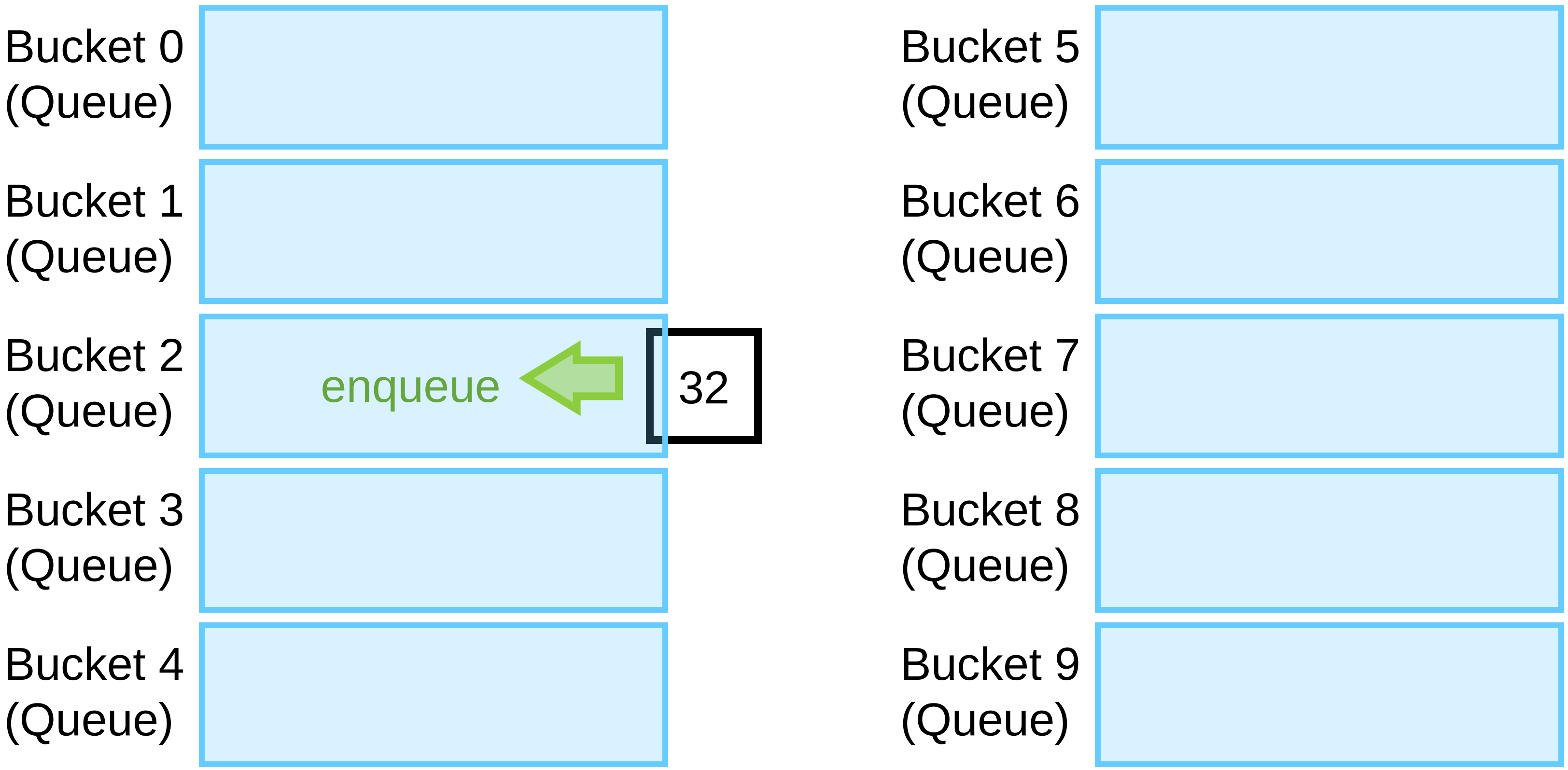
Radix sort

Implement Radix Sort Using Queues

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

unsorted list

32	53	41	42	2	92	81	33

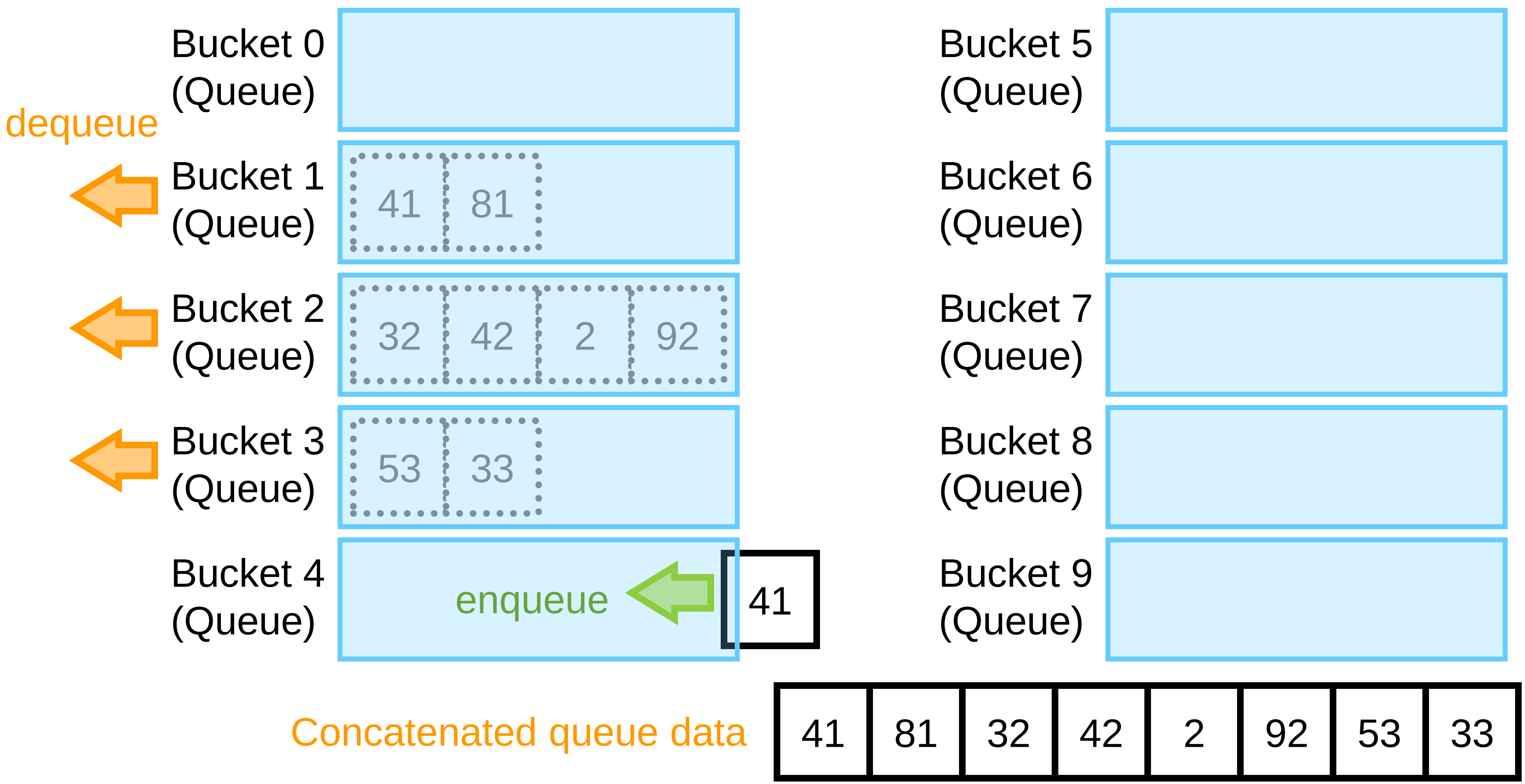


After 1st Pass

32	53	41	42	2	92	81	33

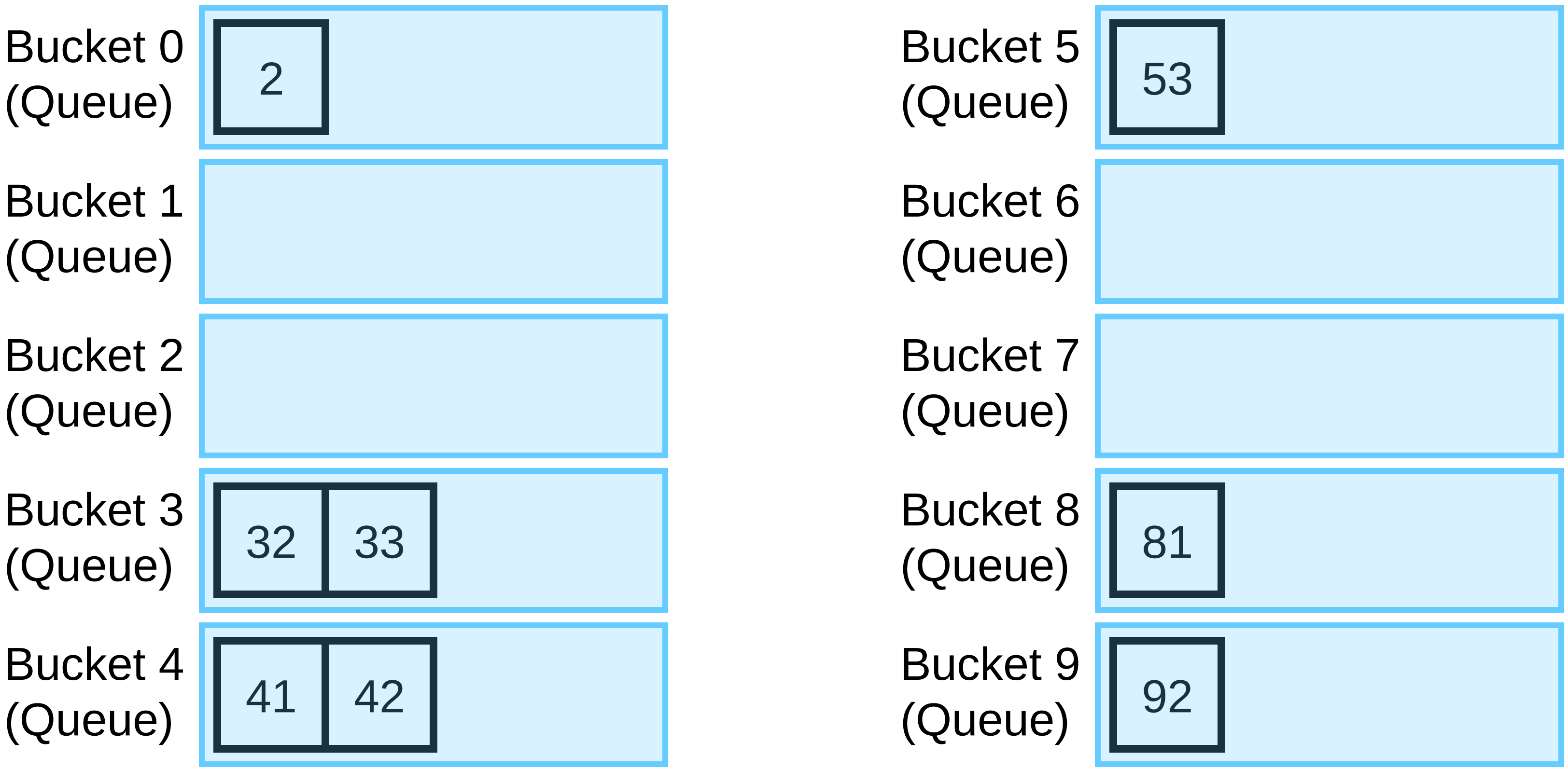
Bucket 0 (Queue)	
Bucket 1 (Queue)	41, 81
Bucket 2 (Queue)	32, 42, 2, 92
Bucket 3 (Queue)	53, 33
Bucket 4 (Queue)	
Bucket 5 (Queue)	
Bucket 6 (Queue)	
Bucket 7 (Queue)	
Bucket 8 (Queue)	
Bucket 9 (Queue)	

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

- Radix sort can have many variations
- Sorting strings
- Use 26 buckets (a to z)?
  - **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

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

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 6th pass

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

After 7th pass

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

After 8th 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

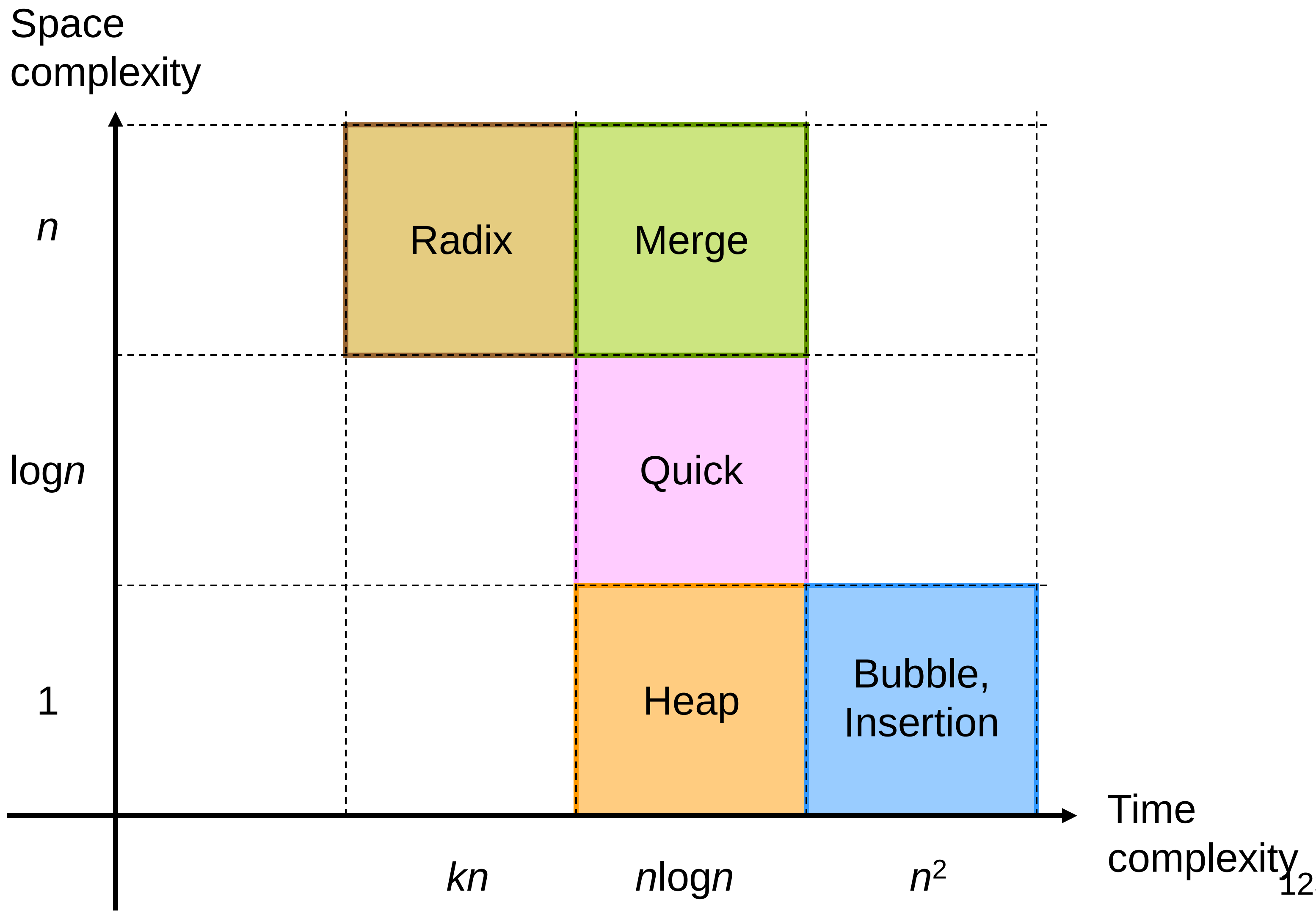
- To obtain 2nd last bit
  - bit = (c >> 1) & 0x01;
  - // >> 1: shift the bits one step to right. The original right most bit is discarded
  - //c >> 1: 0010 1001
  - //0010 1001 AND 0000 0001 = 1

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;

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/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);
}
```