C++ For C Coders 7

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

bubble sort
insertion sort
selection sort
quicksort

1. Bubble sort(거품 정렬)

- It sorts by repeatedly swapping the adjacent elements if they are in wrong order. It is a stable sort. For example,
- 1st Pass:
  - (**5 1** 4 2 8) –> (**1 5** 4 2 8), It compares the first two elements, and swaps since 5 > 1.
  - (1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
  - (1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
  - (1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), it does not swap them.
- 2nd Pass:
  - (**1 4** 2 5 8) –> (**1 4** 2 5 8)
  - (1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
  - (1 2 **4 5** 8) –> (1 2 **4 5** 8)
  - (1 2 4 **5 8**) –> (1 2 4 **5 8**)
  - Now, the sequence is already sorted, but the algorithm does not know if it is completed. It needs one **whole** pass without **any** swap to know it is sorted.

1: Bubble sort(거품 정렬)

- 3rd Pass:
  - (**1 2** 4 5 8) –> (**1 2** 4 5 8)
  - (1 **2 4** 5 8) –> (1 **2 4** 5 8)
  - (1 2 **4 5** 8) –> (1 2 **4 5** 8)
  - (1 2 4 **5 8**) –> (1 2 4 **5 8**)
  - Sorting is over since no element is swapped.

It compares every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

https://en.wikipedia.org/wiki/Bubble_sort

2: Insertion sort(삽입 정렬)

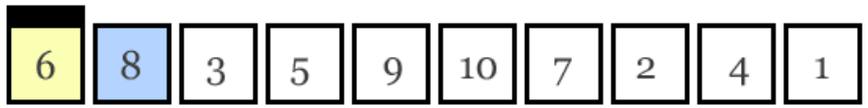
- It works the way we sort playing cards in our hands. It builds the final sorted array one item at a time.
- "Stable" does not change the relative order of elements with equal keys.
- "In-place" only requires a constant amount O(1) of additional memory space.
- "Online" can sort a list as it receives it.

The partial sorted list (black) initially contains only the first element in the list.
With each iteration one element (red) is removed from the "not yet checked for order" input data and inserted in-place into the sorted list.

https://en.wikipedia.org/wiki/Insertion_sort

3: Selection sort(선택 정렬)

- It selects the **smallest element** from an unsorted list in each iteration and places that element **at the beginning of the unsorted list.**
- It is unstable. Why?



Yellow is smallest number found
Blue is current item
Green is sorted list

It divides its list into a sorted and an unsorted section.
Then it swaps the smallest element it finds in each iteration, and add it to the sorted section of elements.

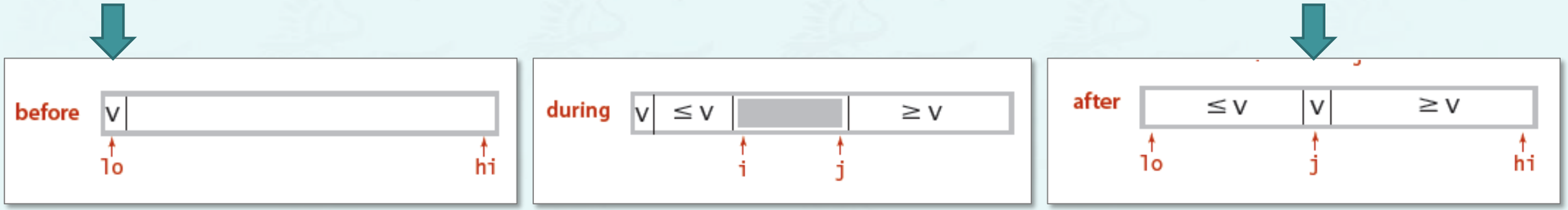
https://en.wikipedia.org/wiki/Selection_sort

3: Selection sort(선택 정렬)

- **Why is a selection sort algorithm unstable?**
  - It picks the minimum and swaps it with the element at current position.
  - Suppose the array is:
    5 2 9 5 4 3 1 6
  - Let's distinguish the two 5's as 5(a) and 5(b) .
    5(a) 2 9 5(b) 4 3 1 6
  - After the first iteration, will be swapped with the element in 1st position:
    So the array becomes:
    1 2 9 5(b) 4 3 5(a) 6
  - Now, we clearly see that **5(a)** and **5(b)** are swapped in the sorted array.
    Therefore, this algorithm is **unstable.**

4. quicksort(퀵 정렬)

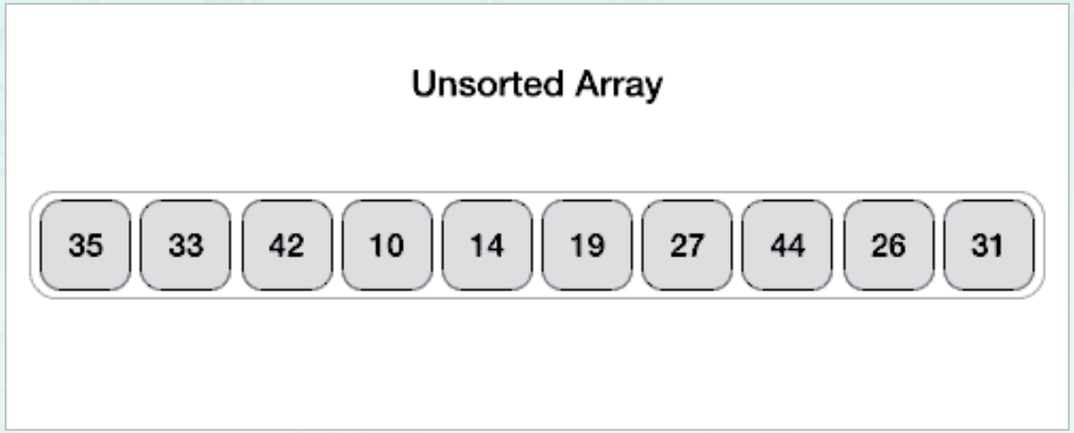
- Quicksort is a divide-and-conquer algorithm.
- It works by selecting a '**pivot**' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively.



https://en.wikipedia.org/wiki/Quicksort

4. quicksort(퀵 정렬)

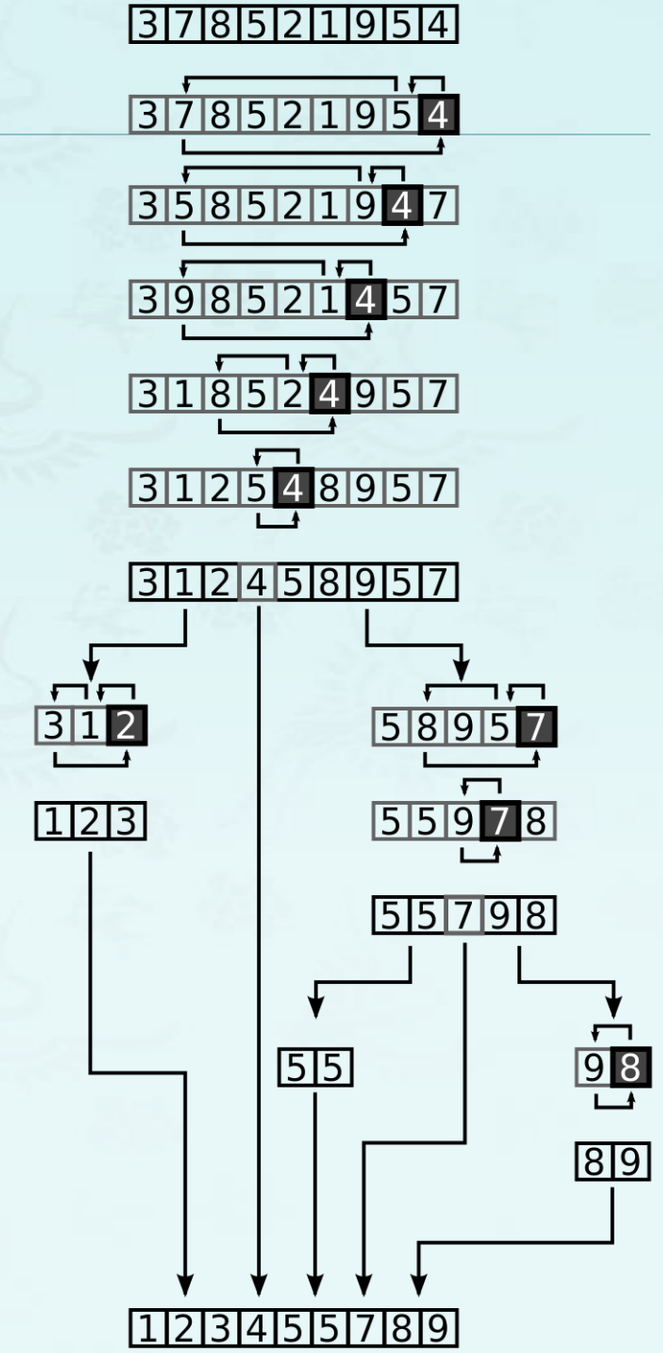
- Quicksort is a divide-and-conquer algorithm.
- It works by selecting a '**pivot**' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively.



https://en.wikipedia.org/wiki/Quicksort

4. quicksort(퀵 정렬)

- Quicksort is a divide-and-conquer algorithm.
- It works by selecting a '**pivot**' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively.

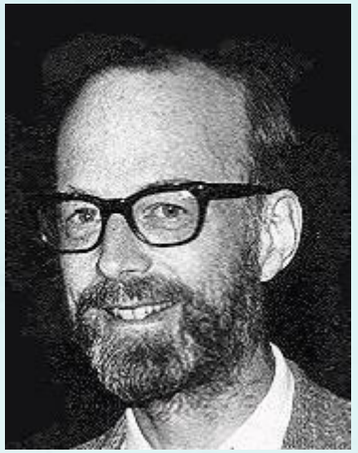


The shaded element is the **pivot**.
It is chosen as the last element of the partition here.

https://en.wikipedia.org/wiki/Quicksort

4. Quicksort - by Hoare in 1961

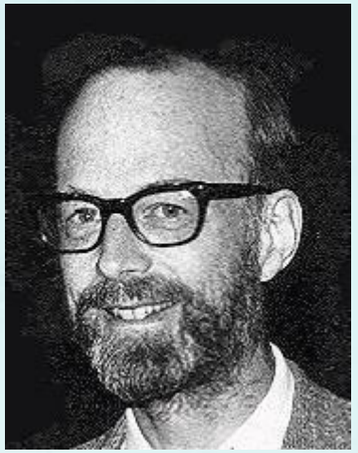
- Algorithm:
  - Shuffle the array.
  - Partition so that, for some j
    - entry a[j] is in place
    - no larger entry to the left of j
    - no smaller entry to the right of j
  - Sort each piece recursively.



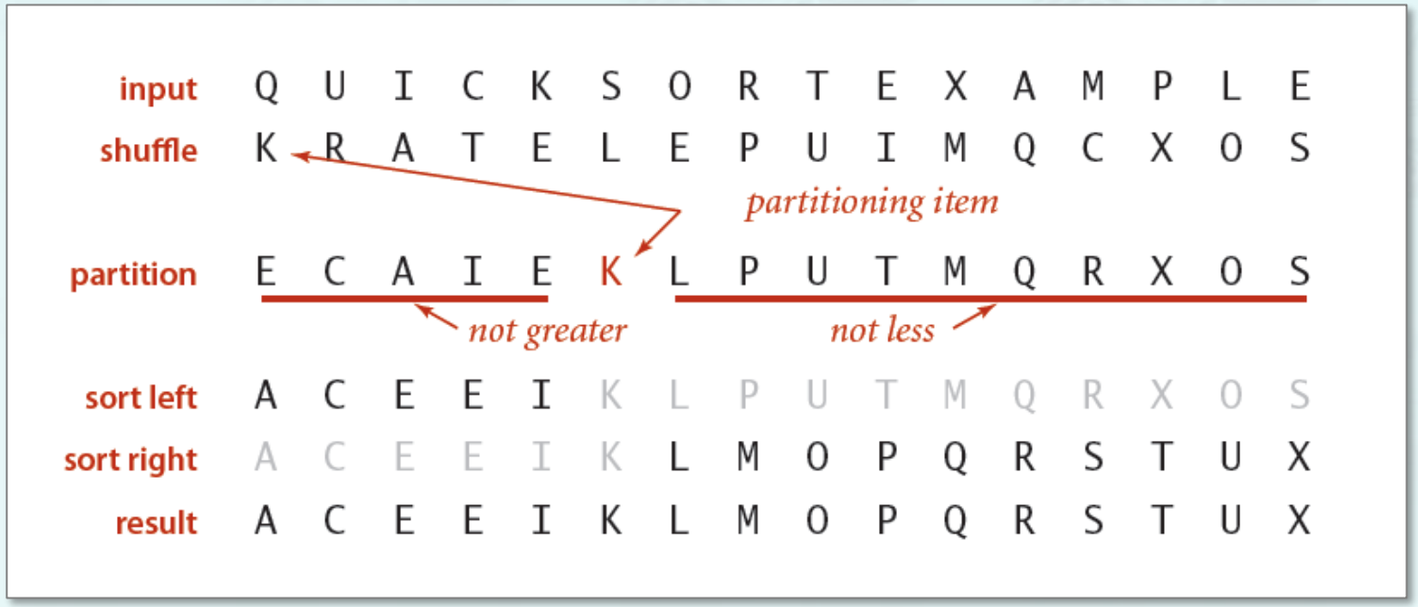
Sir Charles Antony Richard Hoare
1980 Turing award

4. Quicksort - by Hoare in 1961

- Algorithm:
  - Shuffle the array.
  - Partition so that, for some j
    - entry a[j] is in place
    - no larger entry to the left of j
    - no smaller entry to the right of j
  - Sort each piece recursively.



Sir Charles Antony Richard Hoare
1980 Turing award



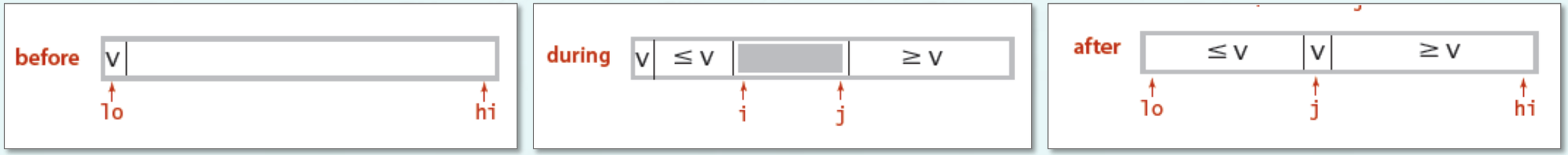
https://algs4.cs.princeton.edu/23quicksort/

Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.

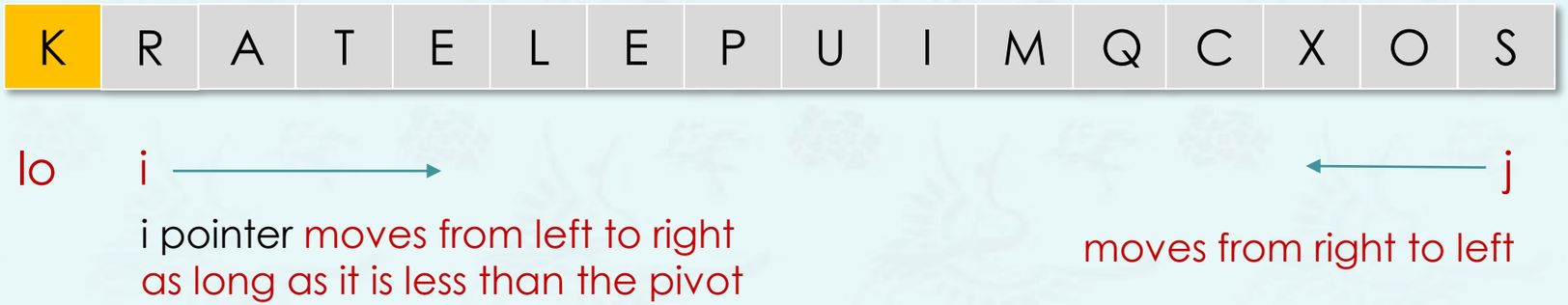
pivot partitioning element

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
lo	i														j



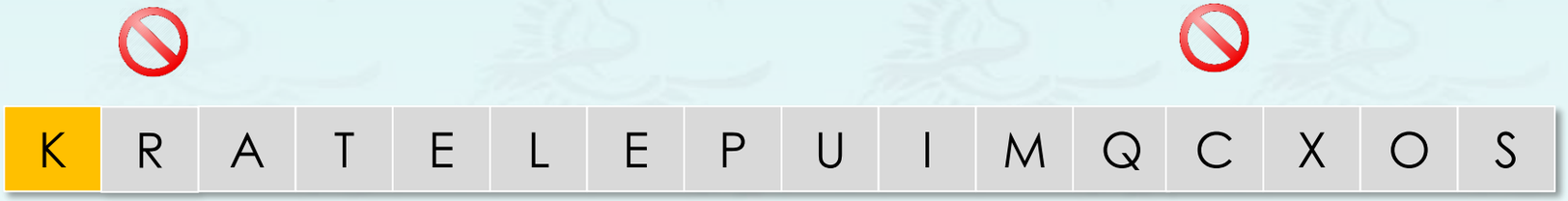
Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as $(a[i] < a[lo])$
  - Scan j from right to left so long as $(a[j] > a[lo])$.
  - Exchange $a[i]$ with $a[j]$.



lo i

i pointer moves from left to right
as long as it is less than the pivot

i pointer stops immediately since ….

j ← j

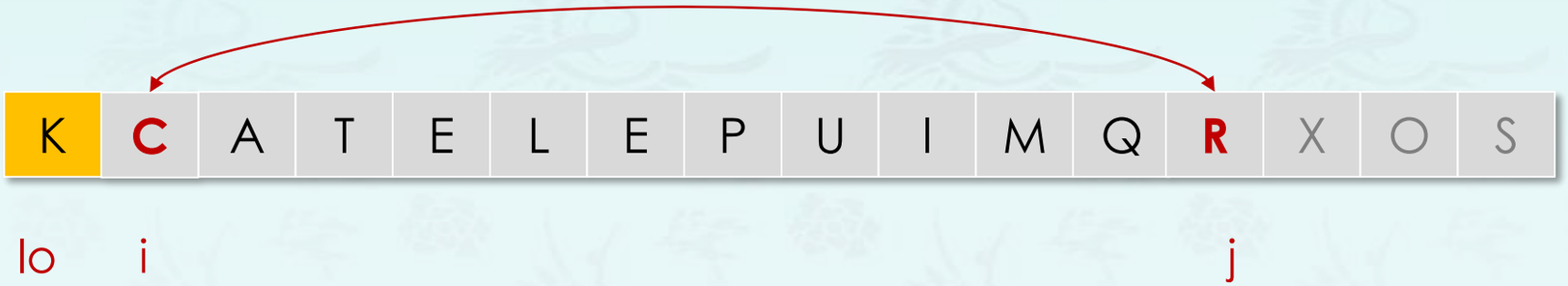
moves from right to left

now decrement j until ….

j pointer moves and stops at "C"

Quicksort partitioning demo

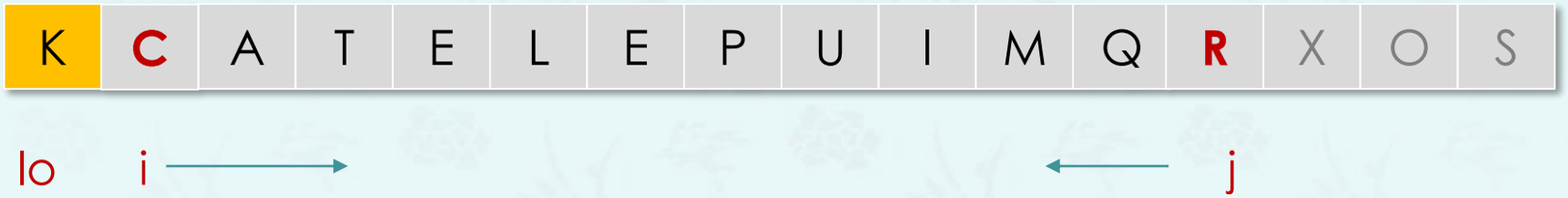
- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as $(a[i] < a[lo])$
  - Scan j from right to left so long as $(a[j] > a[lo])$.
  - Exchange $a[i]$ with $a[j]$.



stop scan and exchange a[i] with a[j]

Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



now increment i until ….

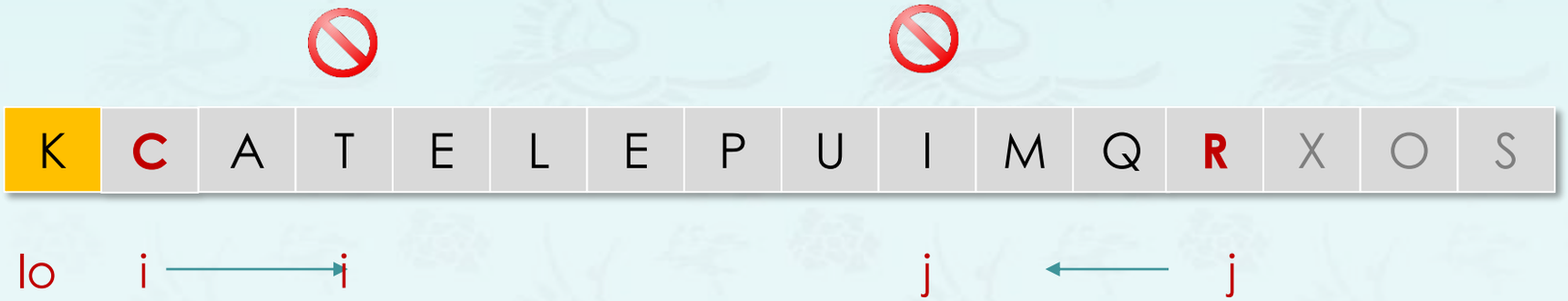
stop i scan because a[i] >= a[lo]

pivot

now decrement j until ….

Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



now increment i until ….

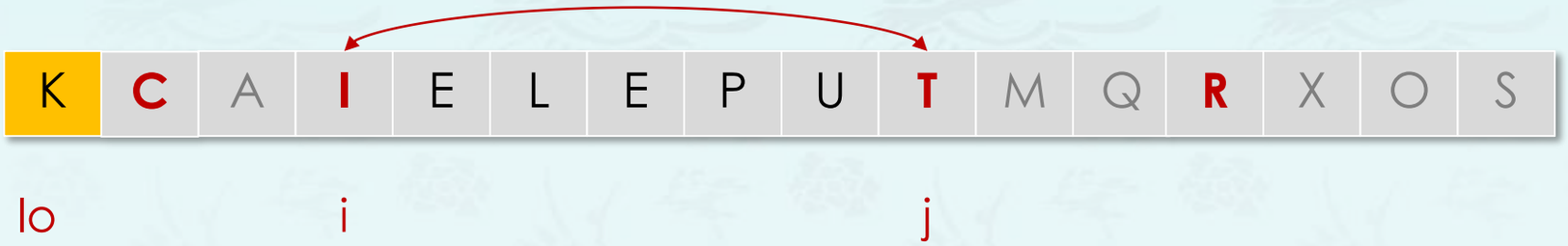
stop i scan because a[i] >= a[lo]

pivot

now decrement j until ….

Quicksort partitioning demo

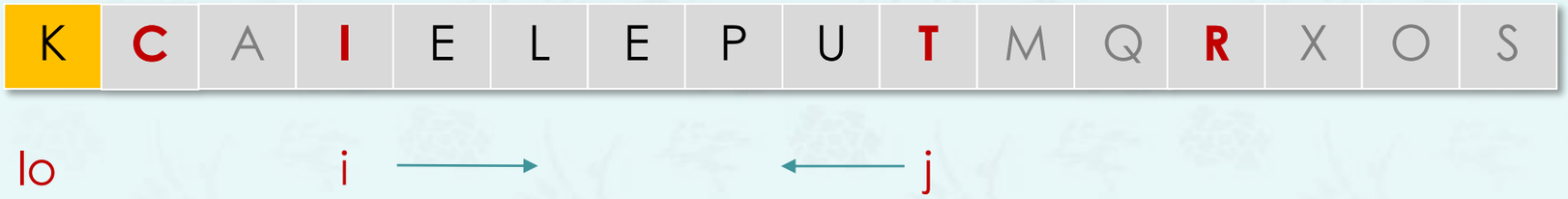
- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



stop scan and exchange a[i] with a[j]

Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



now increment i until ….

now decrement j until ….

Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.

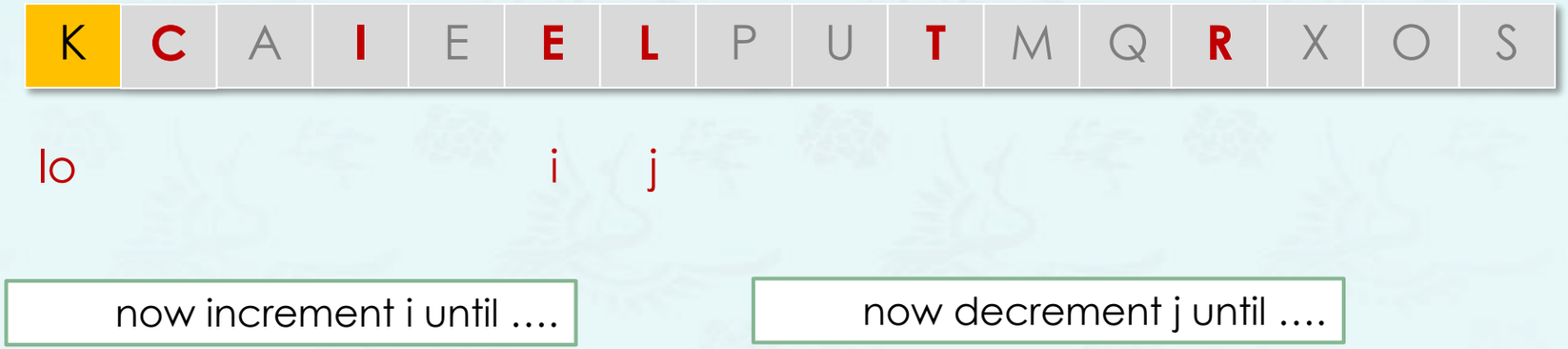
Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.

stop scan and exchange a[i] with a[j]

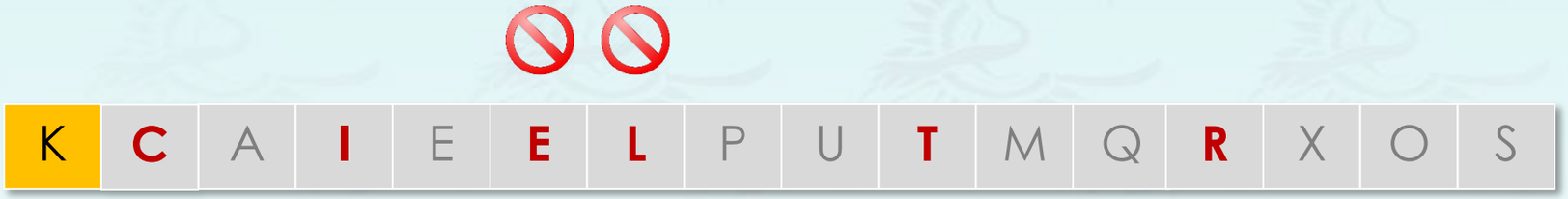
Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.



stop j scan because a[j] <= a[lo]

now j points the last element of left subarray

at this point, partitioning process is complete!

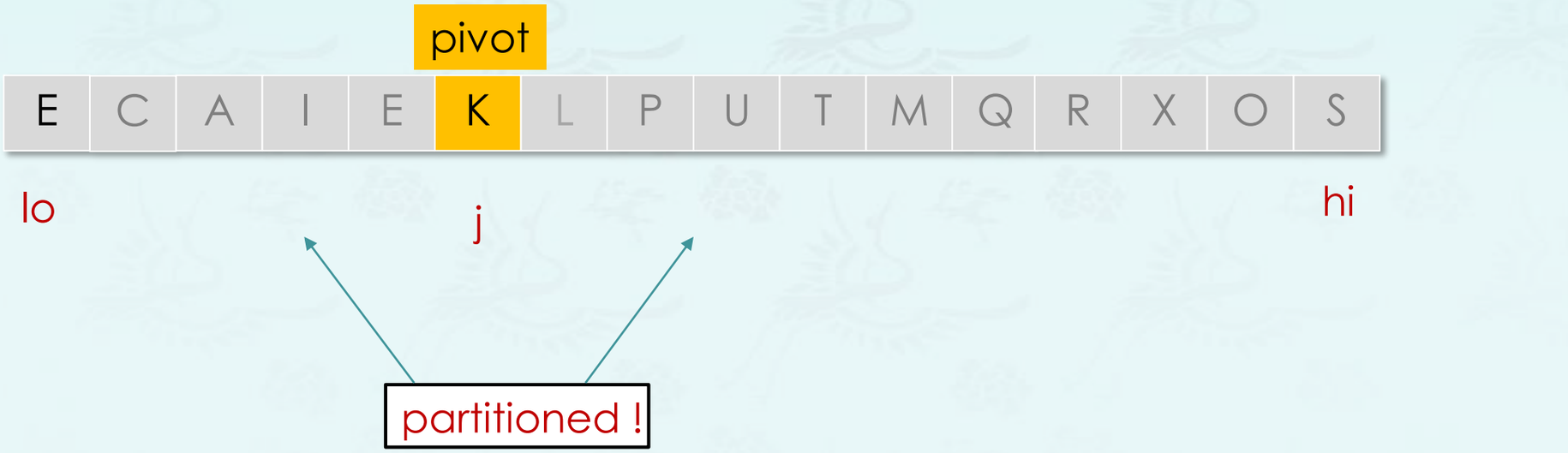
Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.
- **Phase II. When pointers cross.**
  - Exchange a[lo] with a[j].

at this point, partitioning process is complete!

Quicksort partitioning demo

- Phase I. Repeat until ***i*** and ***j*** pointers cross:
  - Scan i from left to right so long as ($a[i] < a[lo]$)
  - Scan j from right to left so long as ($a[j] > a[lo]$).
  - Exchange $a[i]$ with $a[j]$.
- **Phase II. When pointers cross.**
  - Exchange a[lo] with a[j].



at this point, partitioning process is complete!

Quicksort implementation

```
bool less(char a, char b)  { return a < b; }
void swap(char *a, int i, int j)  { char t = a[i];  a[i] = a[j];   a[j] = t; }

int partition(char *a, int lo, int hi) {
    int i = lo; int j = hi + 1;
    while (1) {
        while (less(a[++i], a[lo]))
            if (i == hi) break;

        while (less(a[lo], a[--j]))
            if (j == lo) break;

        if (i >= j) break;
        swap(a, i, j);
    }
    swap(a, lo, j);
    return j;
}
```

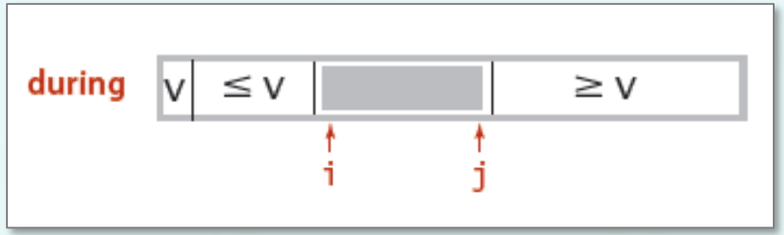
pivot

find item on left to swap

find item on right to swap

check if pointers cross
swap

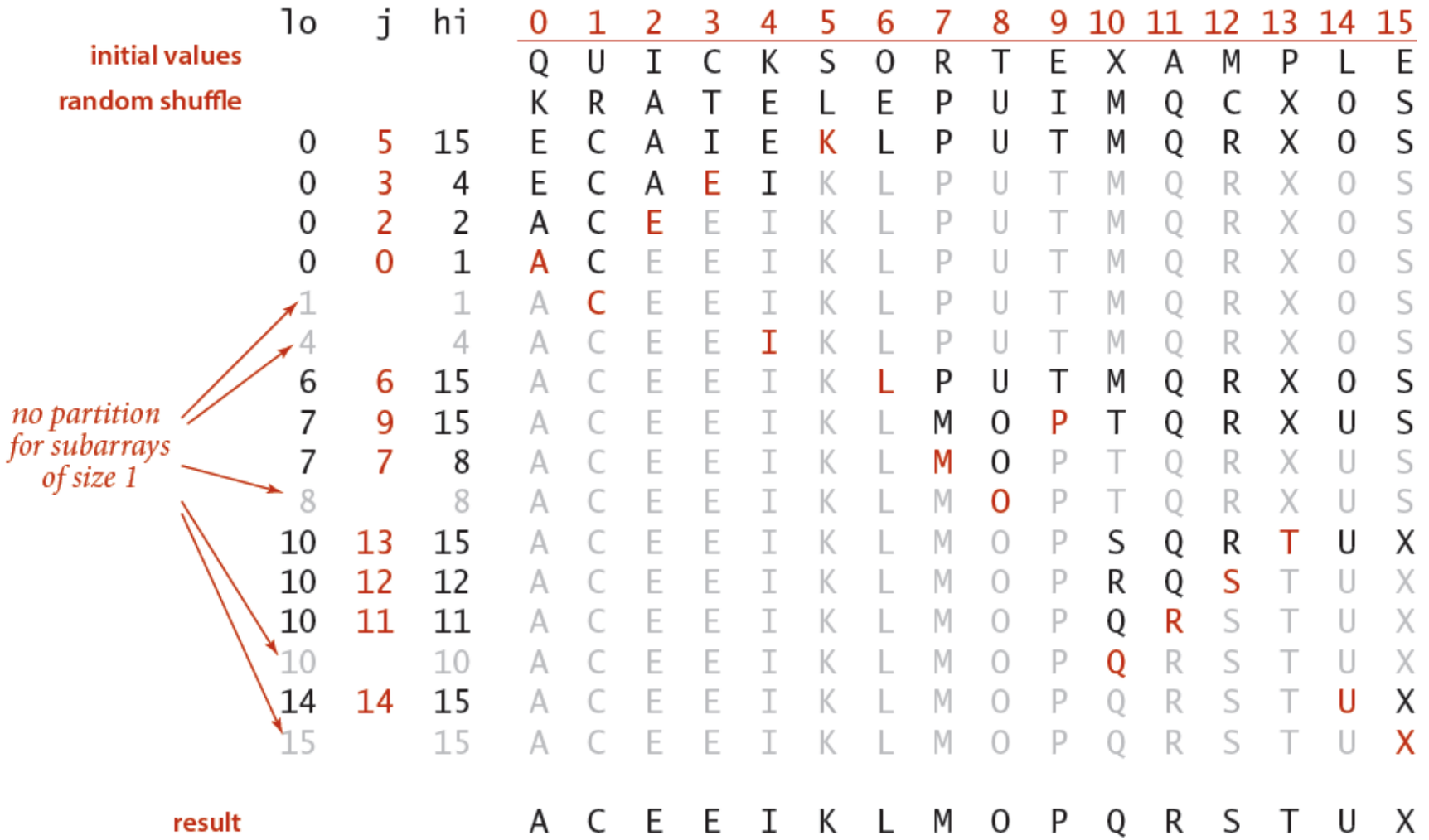
swap with pivot
return index of item now sorted



Quicksort implementation

```
void quicksort(char *a, int lo, int hi) {
    if (hi <= lo) return;

    int j = partition(a, lo, hi);

    quicksort(a, lo,    j - 1);
    quicksort(a, j + 1, hi   );
}

void main() {
  char a[] = {'Q', 'U', 'I', 'C', 'K', 'S', 'O', 'R', 'T', 'E', 'X', 'A', 'M', 'P', 'L', 'E'};
  int N = sizeof(a) / sizeof(a[0]);

  cout << "UNSORTED: \n";
  for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
  //  shuffle(a, N);

  quicksort(a, 0, N-1);
  cout << "SORTED: \n";
  for (int i = 0; i < N; i++) cout << a[i]; cout << endl;
}
```

Quicksort implementation



Quicksort trace (array contents after each partition)

https://algs4.cs.princeton.edu/23quicksort/

Quicksort: best-case analysis

- **Best case:** Number of compares is ~ $N \lg N$.

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst-case analysis

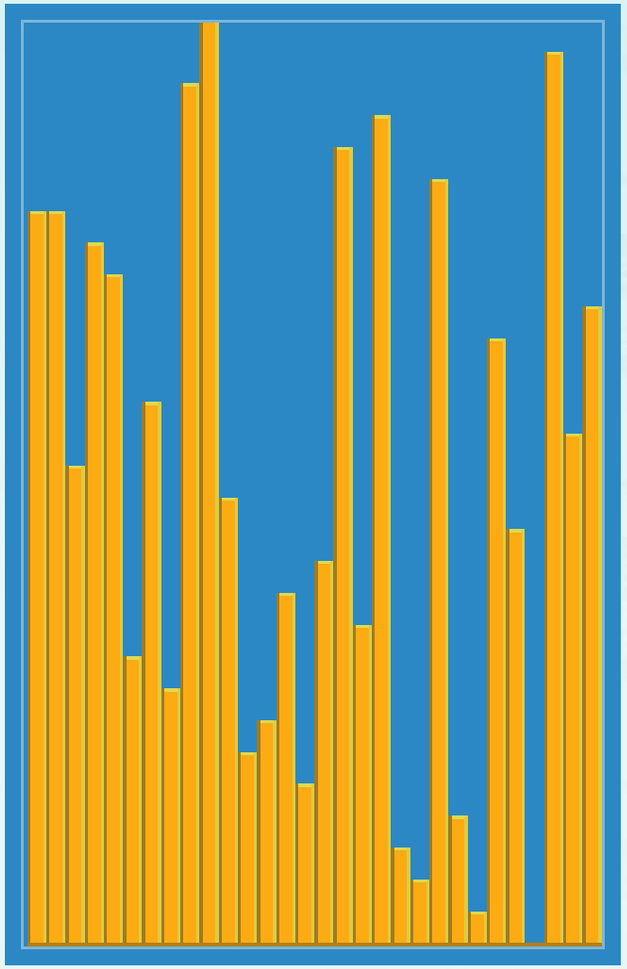
- **Worst case:** Number of compares is ~ ½ N^2 .

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

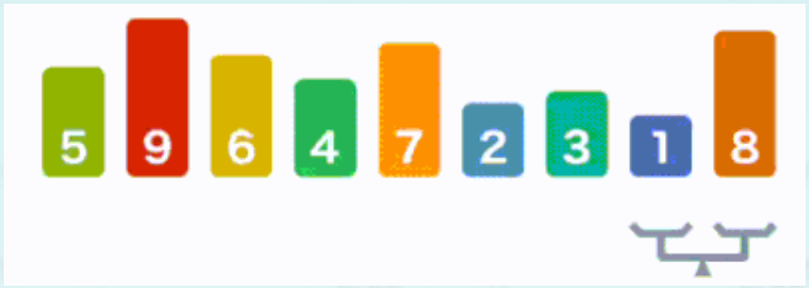
Quicksort: average-case analysis

- Worst case: Number of compares is quadratic.
  - $N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2$.
  - More likely that your computer is struck by lightning bolt
- **Average case:** Number of compares is ~ 1.39 N lg N.
  - 39% more compares than mergesort.
  - But faster than mergesort in practice because of less data movement.
- Random shuffle:
  - Probabilistic guarantee against worst case.
  - Basis for math model that can be validated with experiments.

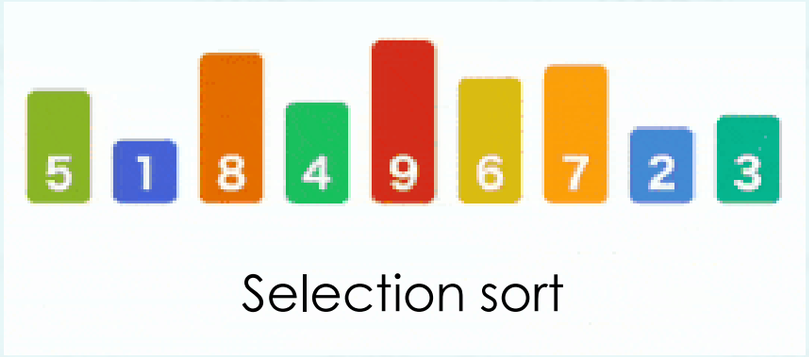
Sorting Algorithm Animation



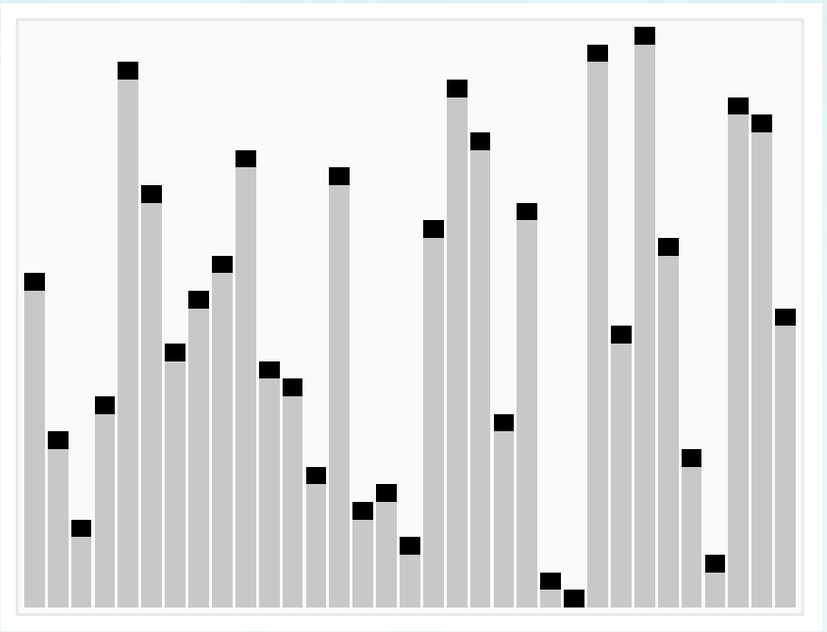
Insertion sort



Bubble sort



Selection sort



Quicksort

https://commons.wikimedia.org/wiki/File:Insertion-sort-example.gif
https://en.wikipedia.org/wiki/Insertion_sort

C++ For C Coders 7

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

bubble sort
insertion sort
selection sort
quicksort