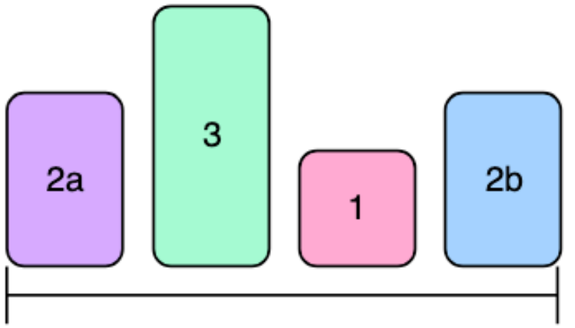
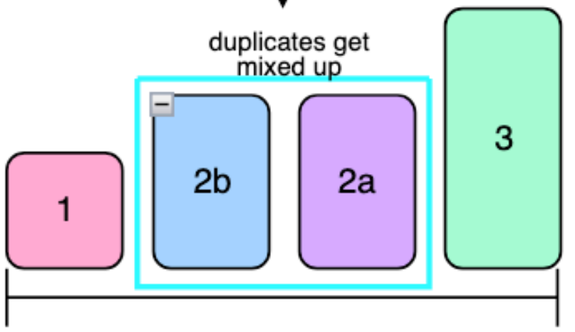
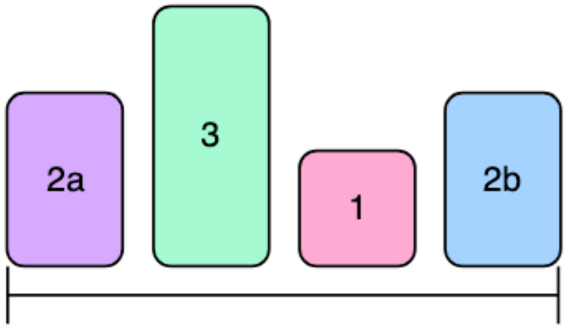
Stability



was a stable sort used?

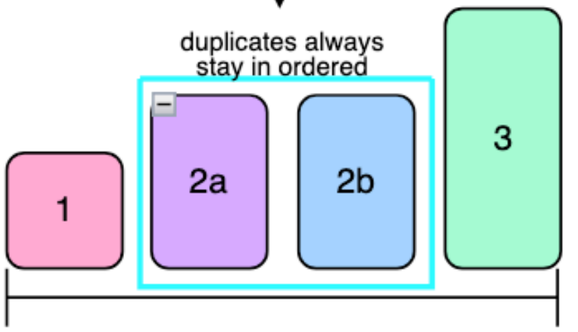
no___



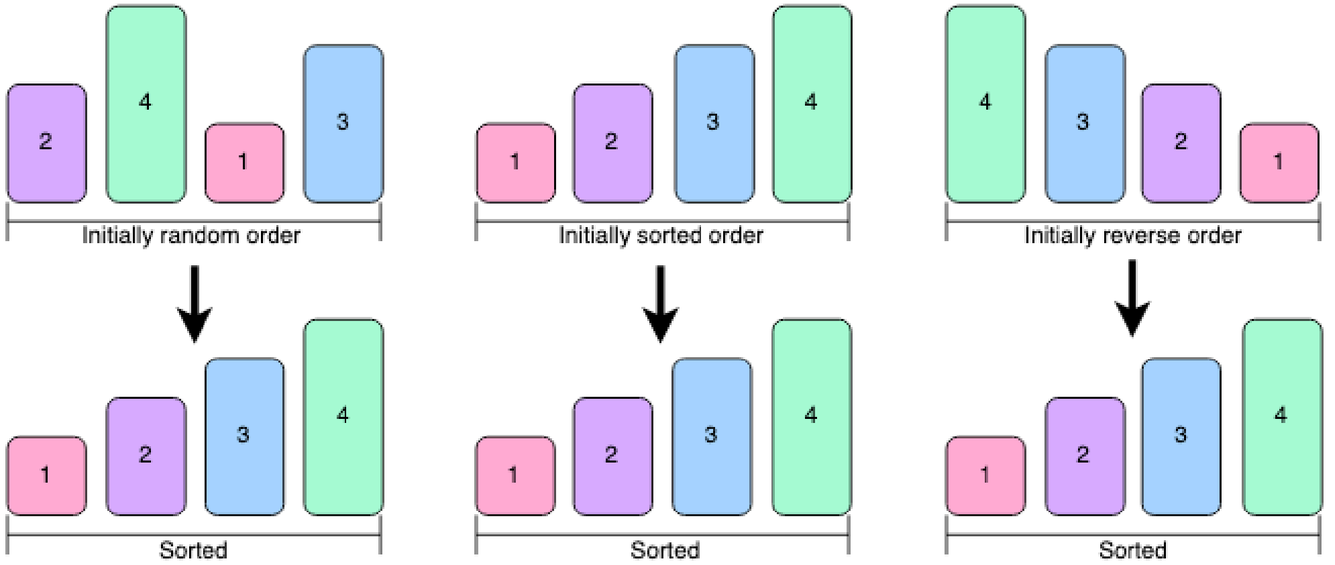


was a stable sort used?

maybe, maybe not



Adaptivity



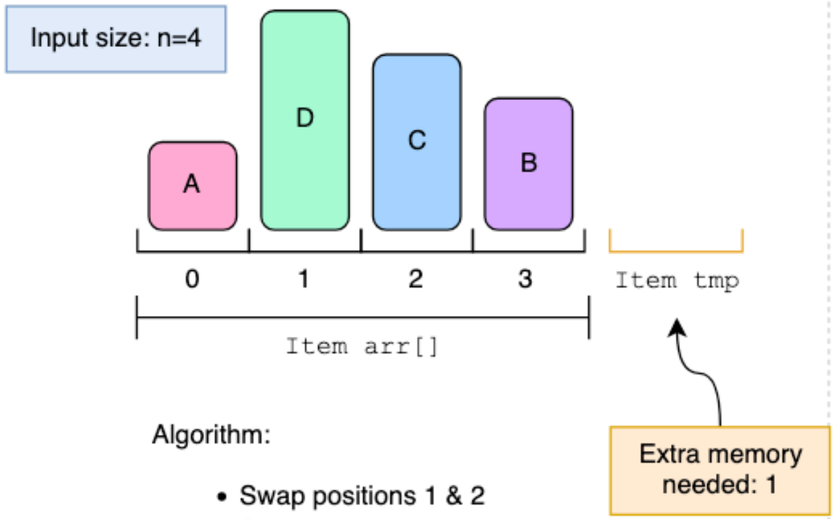
Suppose a sorting algorithm incurs a time complexity
- A to sort an initially random order sequence,
- B to sort an initially sorted order sequence,
- C to sort an initially reverse order sequence.

Is the sorting algorithm considered adaptive or not?

Time complexity A	Time complexity B	Time complexity C	Adaptive Sort?
O(n^2)	O(n * logn)	O(n * logn)	Yes
O(n^2)	O(n)	O(n^2)	Yes
O(n^2)	O(n^2)	O(n)	Yes
O(n * logn)	O(n^2)	O(n^2)	No
O(n^2)	O(n^2)	O(n^2)	No

In-place sorting

A sorting algorithm that **does not** use extra space proportional to the input size



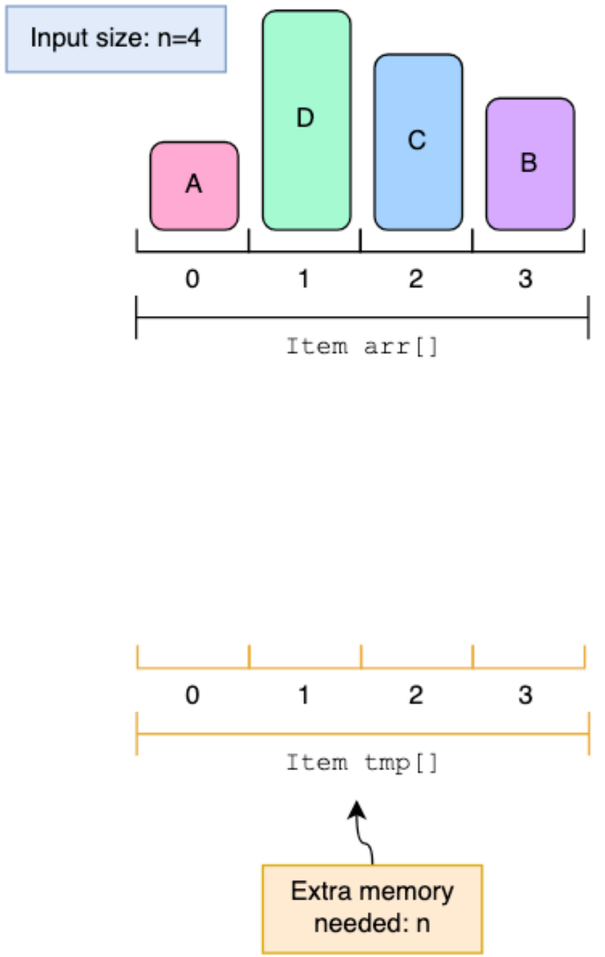
Algorithm:

- Swap positions 1 & 2
- Swap positions 2 & 3
- Swap positions 1 & 2

Space complexity of an in-place sorting algorithm: O(1)

Out-of-place sorting

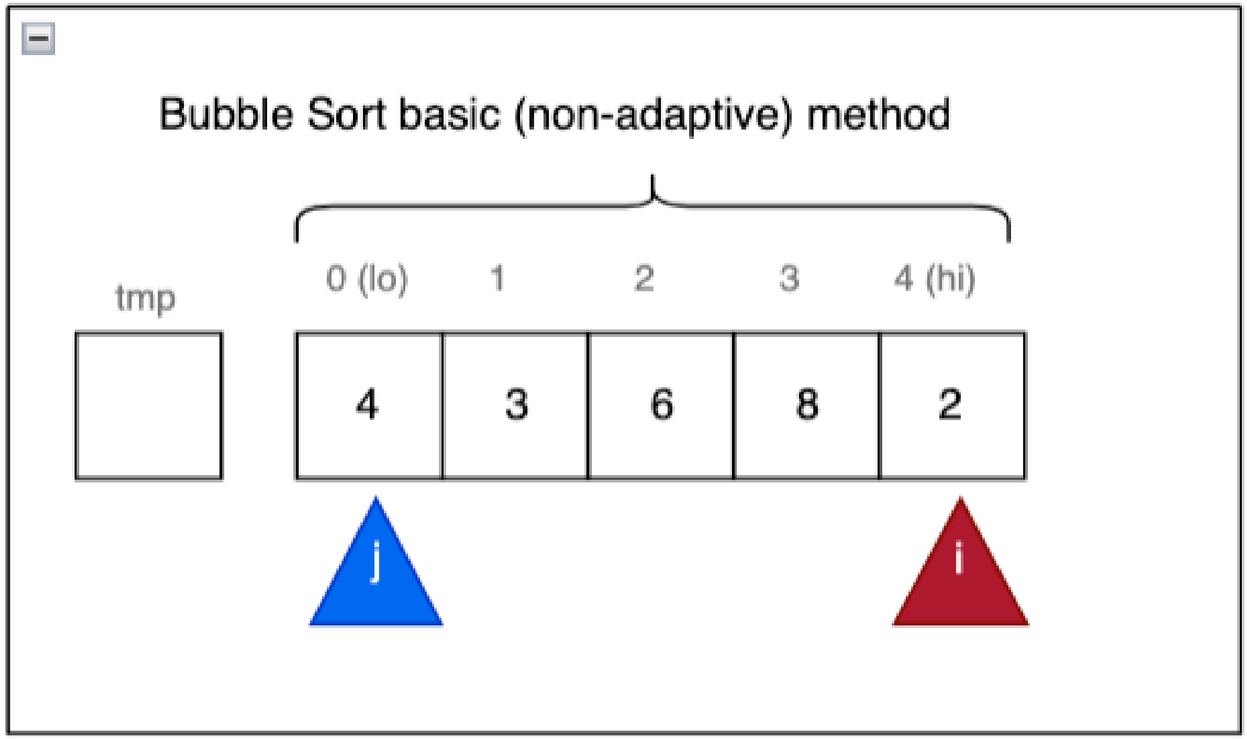
A sorting algorithm that requires extra space proportional to the input size



Algorithm:

- arr[0] -> tmp[0]
- arr[3] -> tmp[1]
- arr[2] -> tmp[2]
- arr[1] -> tmp[3]

Space complexity of an out-of-place sorting algorithm: O(n) or worse


Bubble Sort basic (non-adaptive) method
tmp
0 (lo)
1
2
3
4 (hi)
4
3
6
8
2
j
i

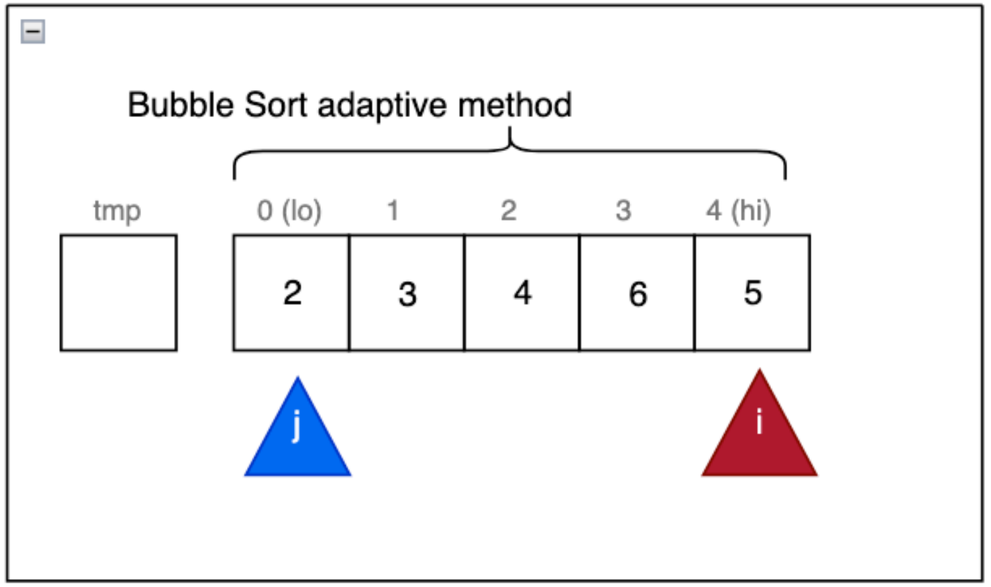
Bubble sort time complexity:

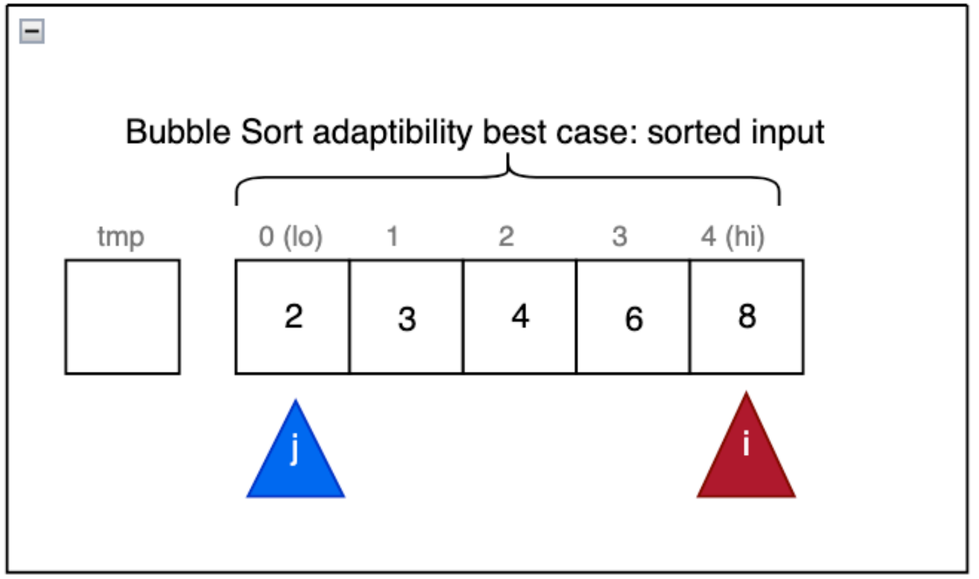
(n-1) + (n-1) + (n-1) + ... + (n-1) = O(n^2)

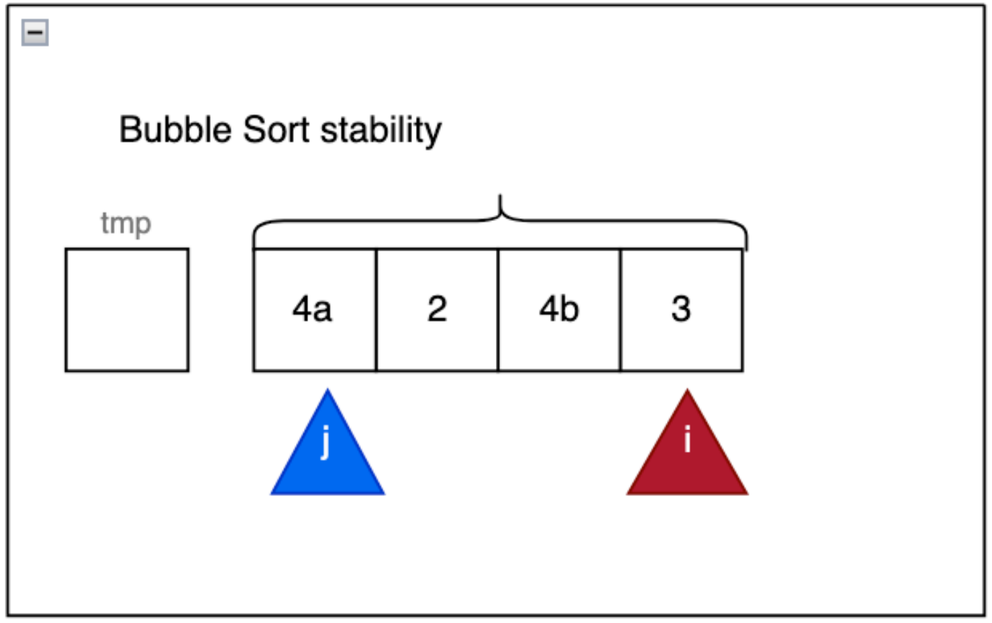
Minor optimisation for bubble sort:

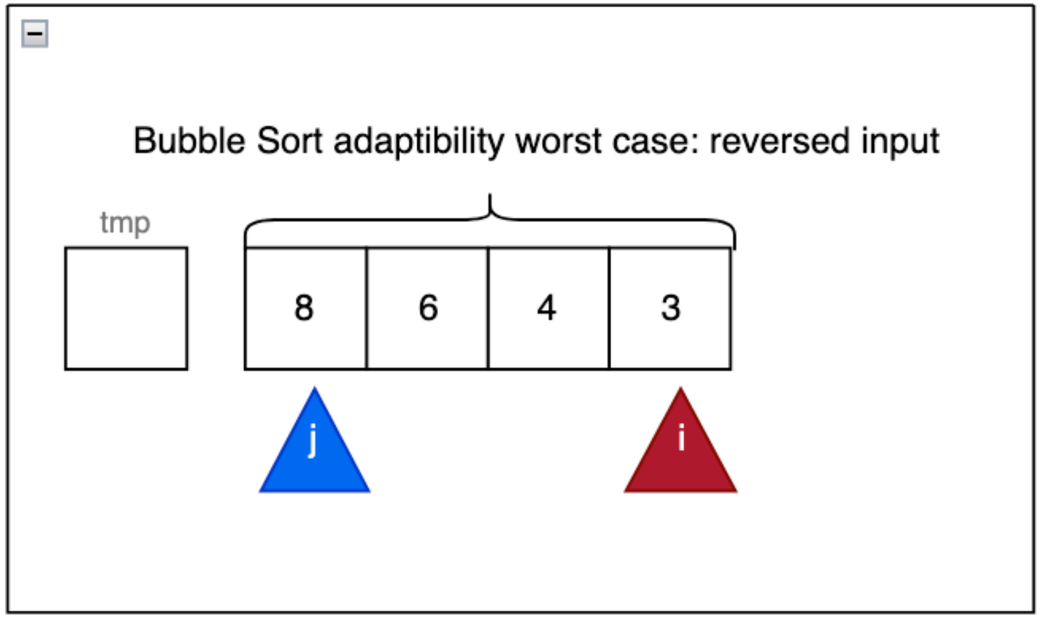
$$(n-1) + (n-2) + \ldots + 2 + 1$$
$$= \underbrace{n + n + \ldots + n}_{n/2}$$
$$= \frac{n}{2} \cdot n$$

$$O\left(\frac{n}{2} \cdot n\right)$$
$$= O\left(\frac{1}{2} \cdot n^2\right)$$
$$= O(n^2)$$


Bubble Sort adaptive method
tmp
0 (lo)
1
2
3
4 (hi)
2
3
4
6
5
j
i


Bubble Sort adaptibility best case: sorted input
tmp
0 (lo)
1
2
3
4 (hi)
2
3
4
6
8
j
i


Bubble Sort stability
tmp
4a
2
4b
3
j
i

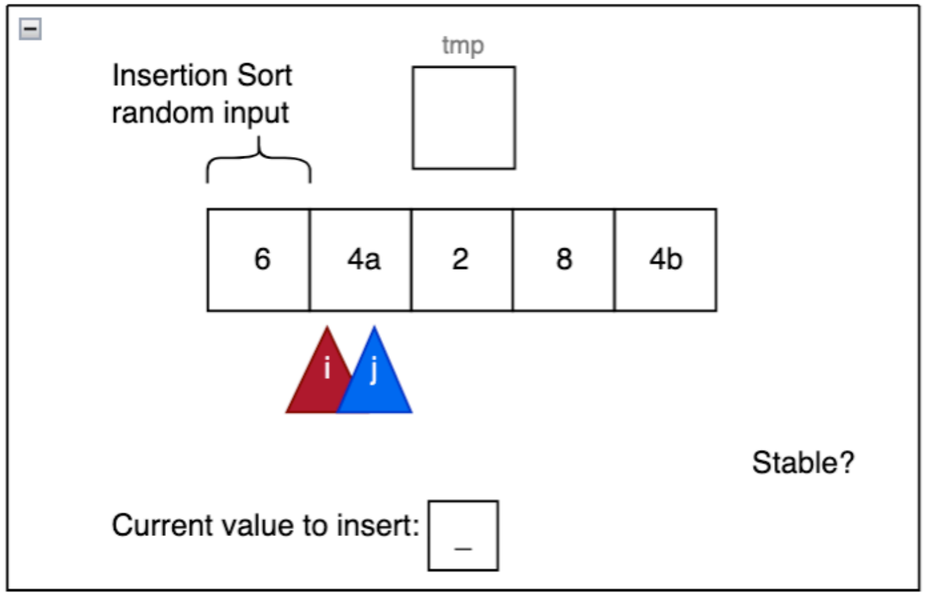

Bubble Sort adaptibility worst case: reversed input
tmp
8
6
4
3
j
i

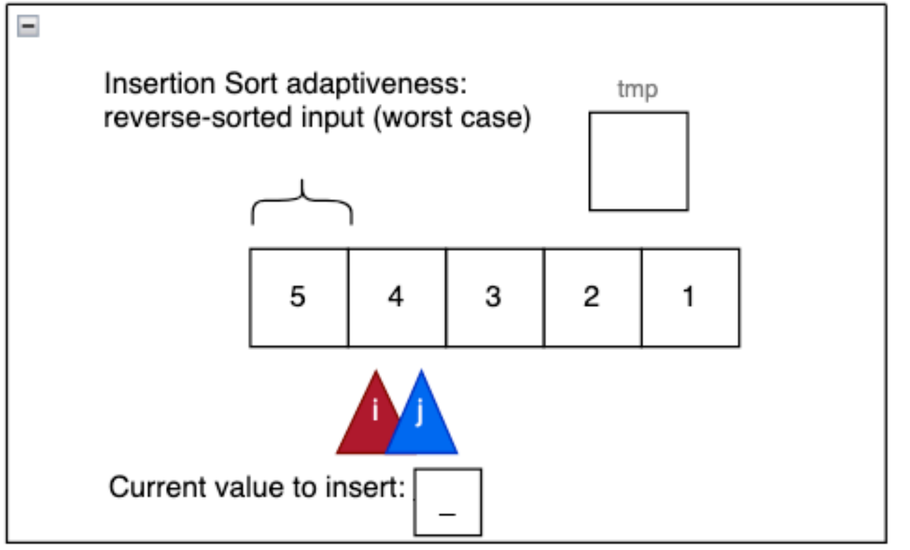
Bubble Sort	
Properties	
Stability	✅ / ❌
Adaptibility	✅ / ❌
In-place	in-place / out-of-place
Time complexity	
Best	O(__)
Worst	O(__)
Average	O(__)
Space complexity	
Average	O(__)

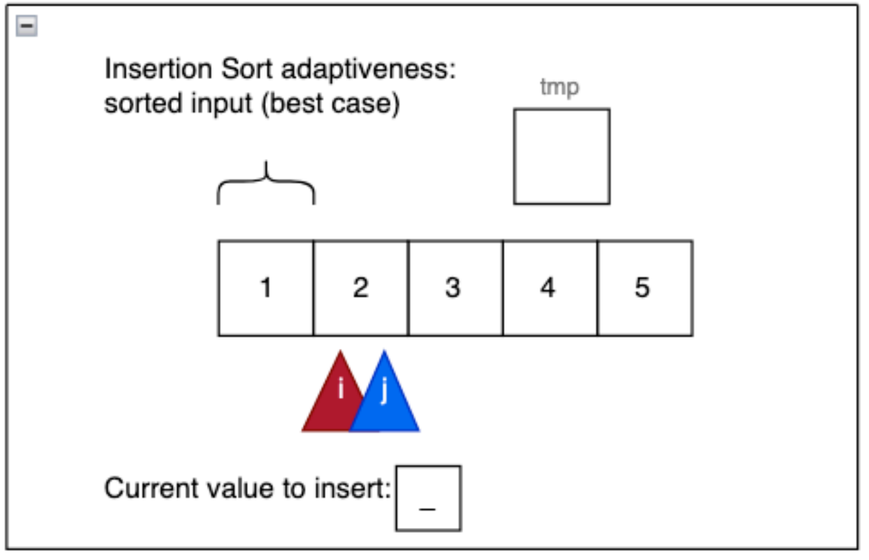
Bubble Sort

C Implementation

```
void bubbleSort(Item items[], int lo, int hi) {
    for (int i = hi; i > lo; i--) {
        bool swapped = false;
        for (int j = lo; j < i; j++) {
            if (gt(items[j], items[j + 1])) {
                swap(items, j, j + 1);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```


Insertion Sort
random input
tmp
6
4a
2
8
4b
i
j
Stable?
Current value to insert:
_


Insertion Sort adaptiveness:
reverse-sorted input (worst case)
tmp
5
4
3
2
1
i
j
Current value to insert:
_


Insertion Sort adaptiveness:
sorted input (best case)
tmp
1
2
3
4
5
i
j
Current value to insert:
_

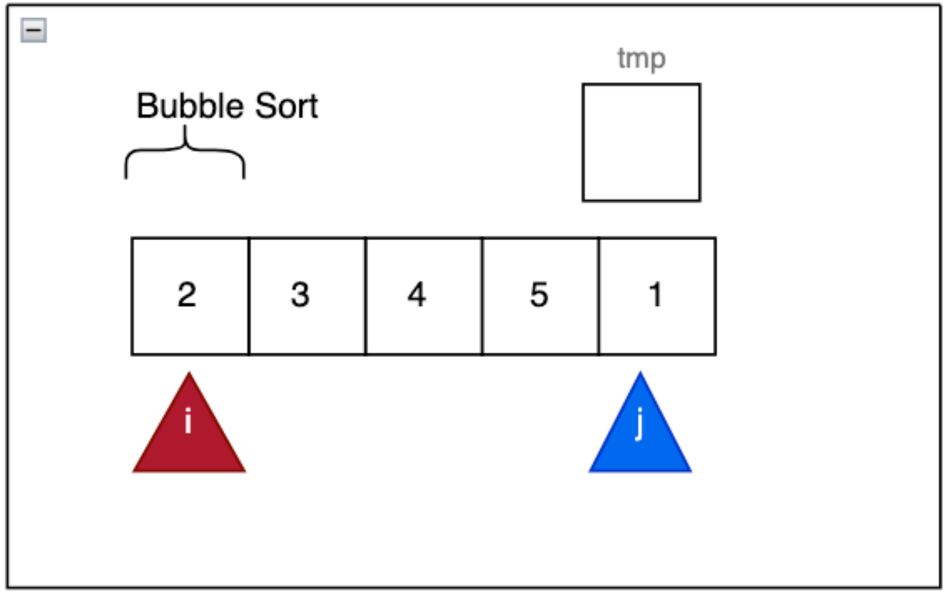
Insertion Sort	
Properties	
Stability	✅ / ❌
Adaptibility	✅ / ❌
In-place	in-place / out-of-place
Time complexity	
Best	O(__)
Worst	O(__)
Average	O(__)
Space complexity	
Average	O(__)

Insertion Sort

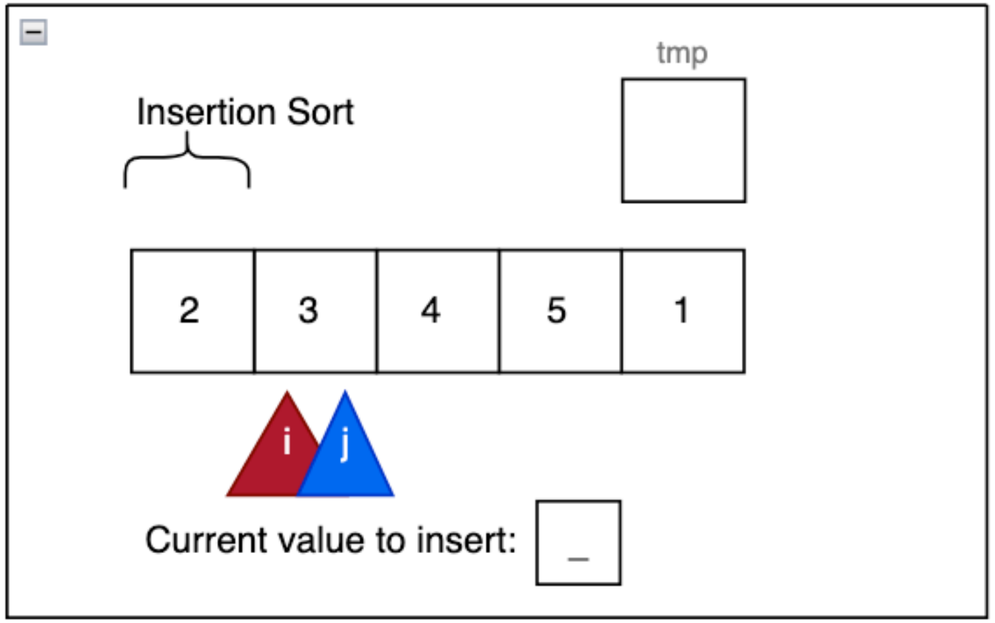
C Implementation


```
void insertionSort(Item items[], int lo, int hi) {
    for (int i = lo + 1; i <= hi; i++) {
        Item item = items[i];
        int j = i;
        for (; j > lo && lt(item, items[j - 1]); j--) {
            items[j] = items[j - 1];
        }
        items[j] = item;
    }
}
```

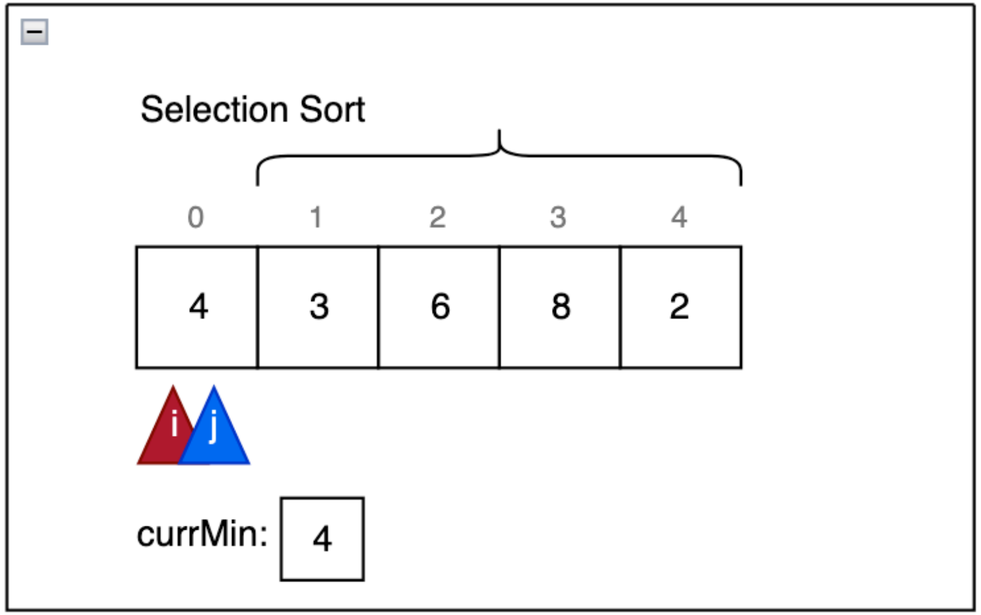
Test case to distinguish between bubble and insertion sort

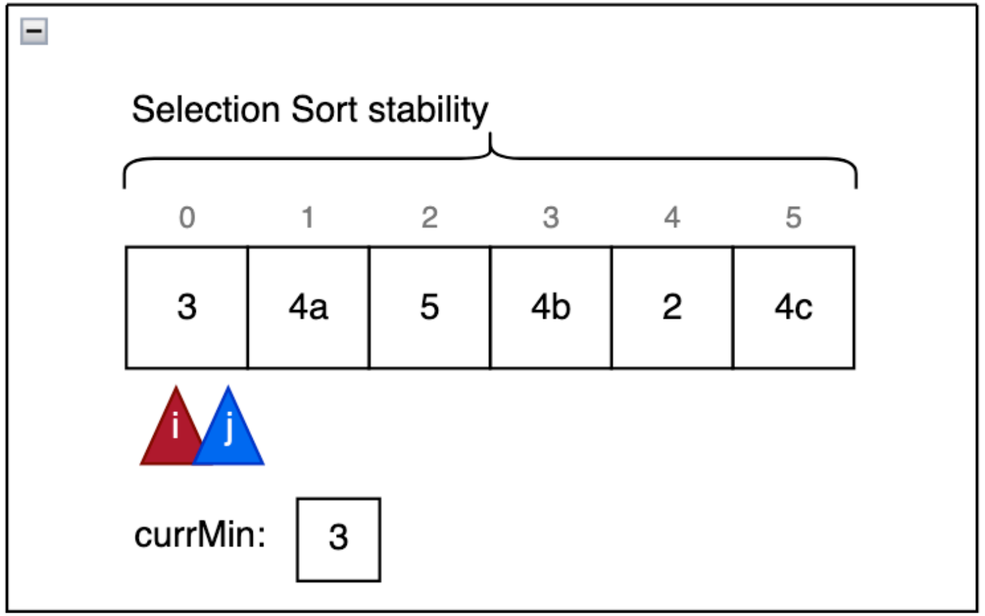


O(n^2)



O(n)


Selection Sort
0
1
2
3
4
4
3
6
8
2
i
j
currMin:
4


Selection Sort stability
0
1
2
3
4
5
3
4a
5
4b
2
4c
i
j
currMin:
3

Selection Sort	
Properties	
Stability	✅ / ❌
Adaptibility	✅ / ❌
In-place	in-place / out-of-place
Time complexity	
Best	O(__)
Worst	O(__)
Average	O(__)
Space complexity	
Average	O(__)

Selection Sort

C Implementation

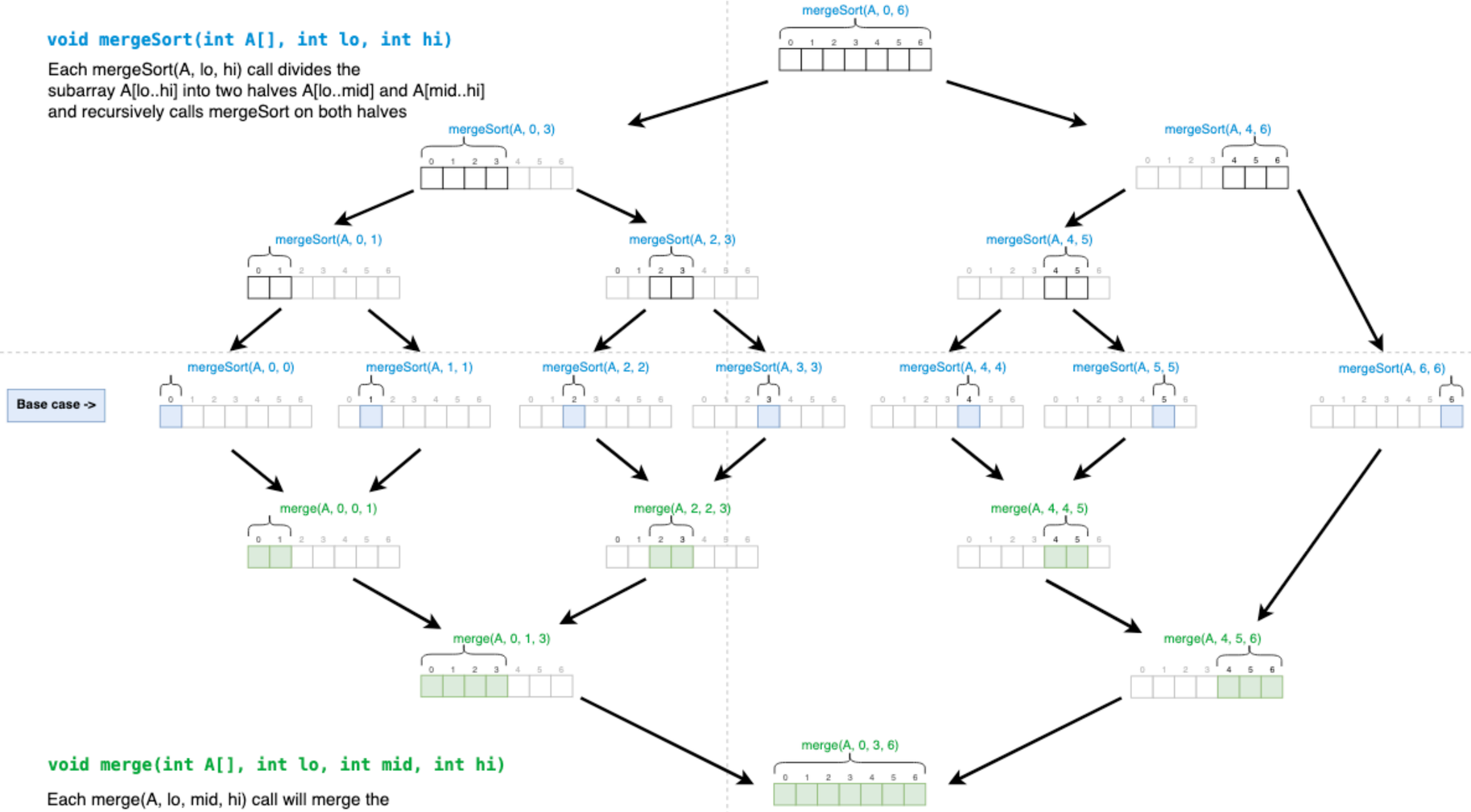

```
void selectionSort(Item items[], int lo, int hi) {
    for (int i = lo; i < hi; i++) {
        int min = i;
        for (int j = i + 1; j <= hi; j++) {
            if (lt(items[j], items[min])) {
                min = j;
            }
        }
        swap(items, i, min);
    }
}
```

Merge Sort

Divide and Conquer

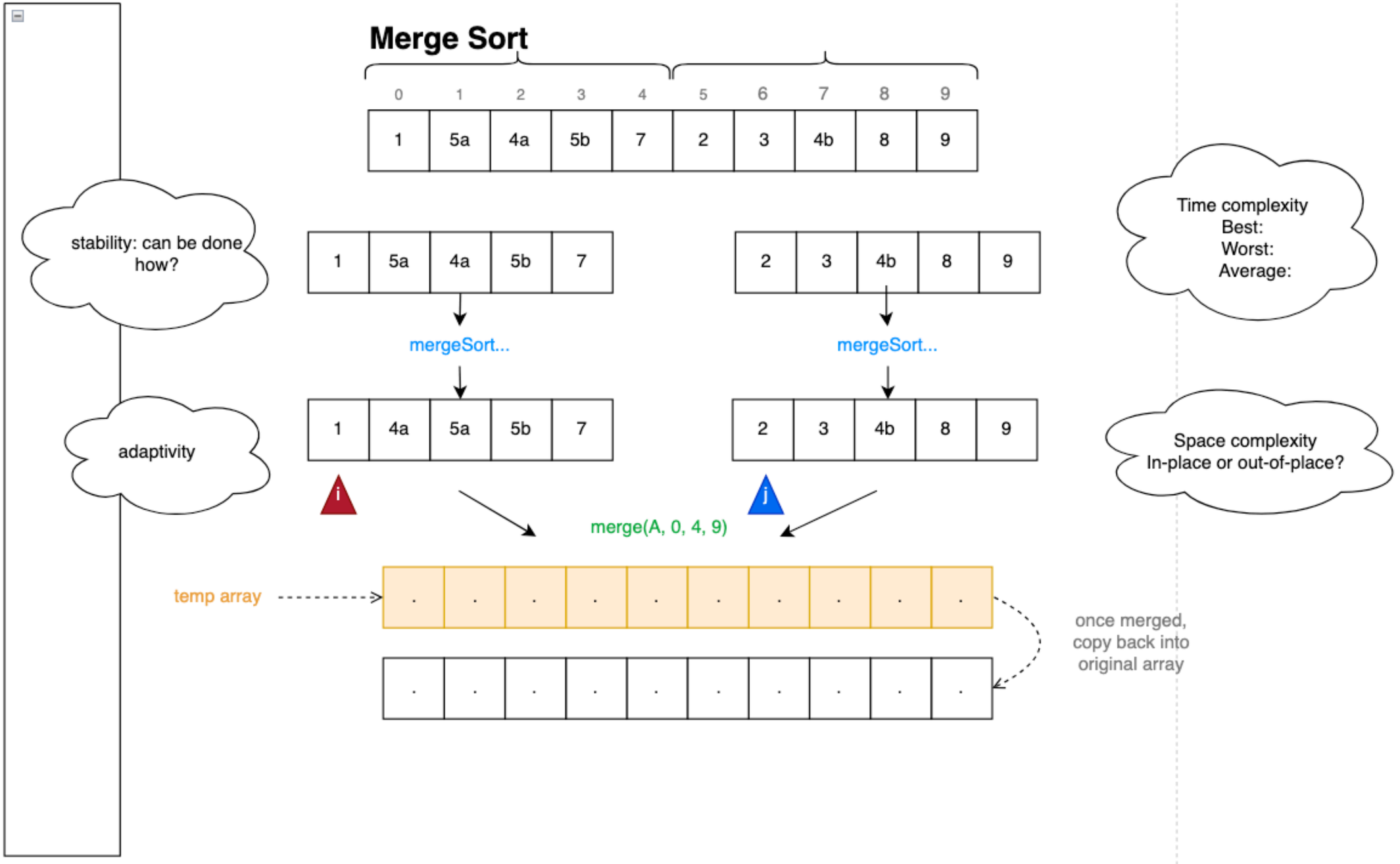
`void mergeSort(int A[], int lo, int hi)`

Each mergeSort(A, lo, hi) call divides the subarray A[lo..hi] into two halves A[lo..mid] and A[mid..hi] and recursively calls mergeSort on both halves



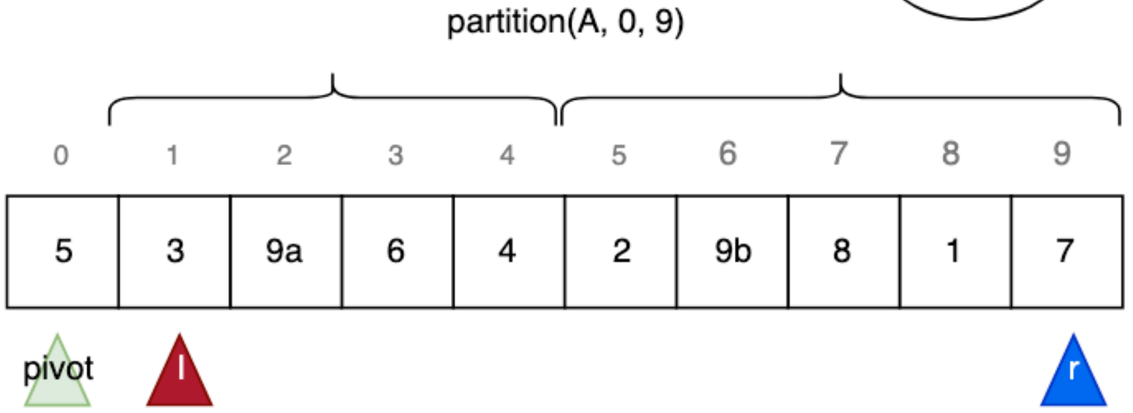
`void merge(int A[], int lo, int mid, int hi)`

Each merge(A, lo, mid, hi) call will merge the two subarrays A[lo..mid] and A[mid..hi] back into the unified subarray A[lo..hi] in sorted order.


Merge Sort
0 1 2 3 4 5 6 7 8 9
1 5a 4a 5b 7 2 3 4b 8 9
stability: can be done how?
adaptivity
1 5a 4a 5b 7
2 3 4b 8 9
mergeSort...
mergeSort...
1 4a 5a 5b 7
2 3 4b 8 9
i
j
merge(A, 0, 4, 9)
temp array
once merged, copy back into original array
Time complexity
Best:
Worst:
Average:
Space complexity
In-place or out-of-place?

Merge Sort	
Properties	
Stability	✅ / ❌
Adaptibility	✅ / ❌
In-place	in-place / out-of-place
Time complexity	
Best	O(___)
Worst	O(___)
Average	O(___)
Space complexity	
Average	O(___)

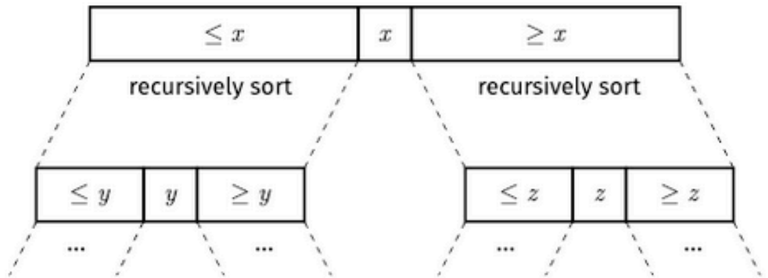
Naive Quick Sort



Quick Sort
Analysis

Best case: $O(n \log n)$

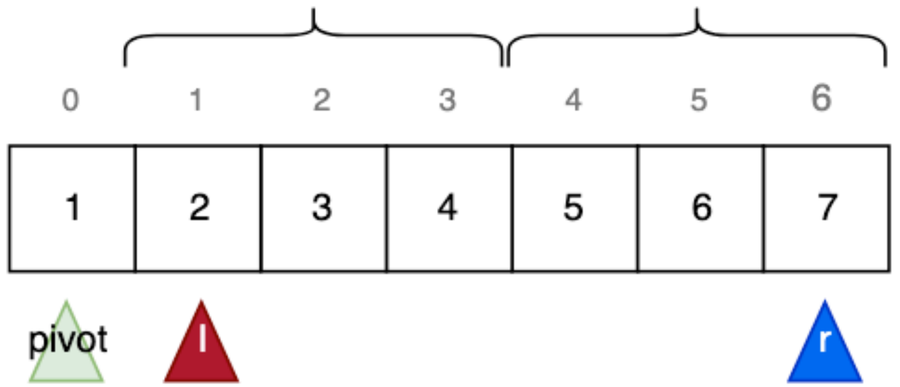
- Choice of pivot gives two equal-sized partitions
- Same happens at every recursive call
  - Resulting in $\log_2 n$ recursive levels
- Each "level" requires approximately n comparisons



Naive Quick Sort

adaptivity?

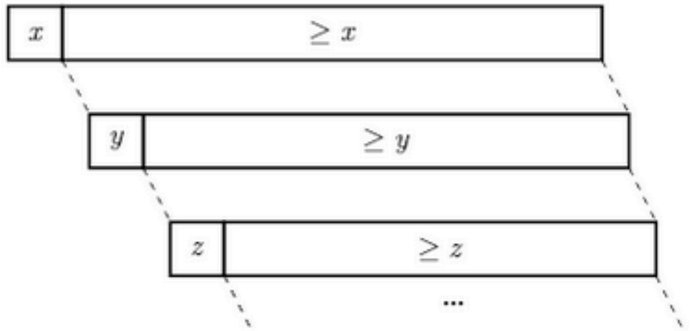
Sorted input (worst case)



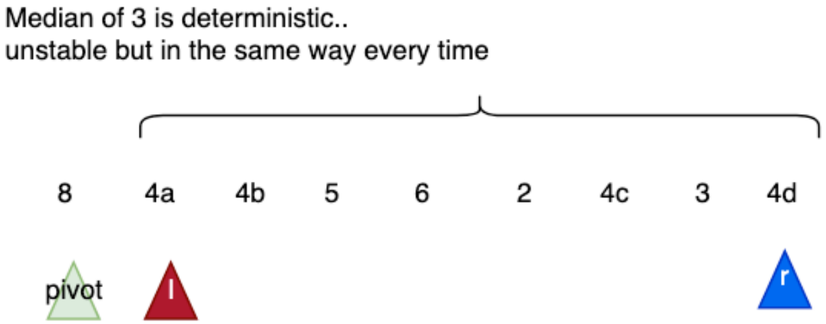
Quick Sort
Analysis

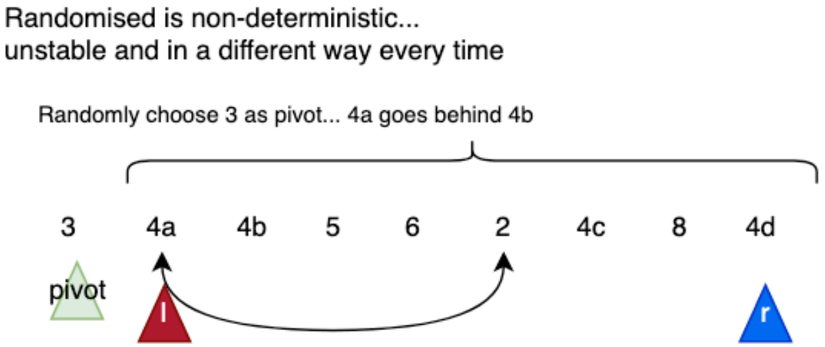
Worst case: $O(n^2)$

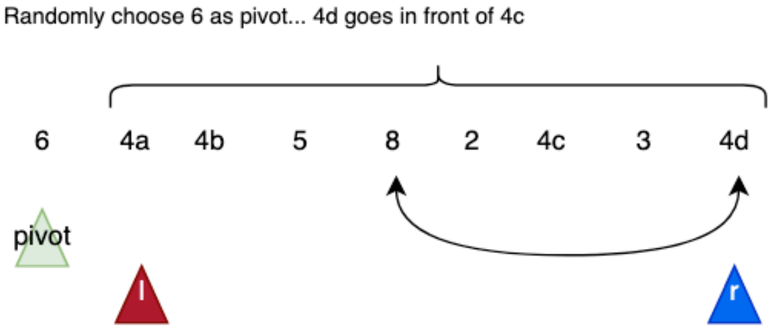
- Always choose lowest/highest value for pivot
  - Resulting in partitions of size 0 and $n-1$
  - Resulting in n recursive levels
- Each "level" requires one less comparison than the level above



Quick Sort	
Properties	
Stability	✅ / ❌
Adaptibility	✅ / ❌
In-place	in-place / out-of-place
Time complexity	
Best	O(__)
Worst	O(__)
Average	O(__)
Space complexity	
Average	O(__)







- How to distinguish between Median-of-3 and Randomised quicksort?