THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING & MATHEMATICAL SCIENCES

QUICKSORT

CS 124 / Department of Computer Science

Quicksort

Like merge sort, quicksort is a "divide-and-conquer" algorithm.

Divide-and-conquer algorithms break a problem down recursively and then combine the results of subproblems to produce the final result.

Recursion

- Recursive functions call themselves.
- Recursive functions require a base case or base cases to prevent infinite digress.
- Examples seen earlier in the course:
    - Merge sort
    - Fibonacci
    - Factorial
    - Shamos' subsequence sum

Quicksort

```
// pseudo-code (see: Stephens, p. 145)

quicksort(vector, start, end)
    pick an element to divide the array (a.k.a. "pivot")
    move items less than the pivot to the left of the pivot
    move items greater than or equal to the pivot to the right of the pivot
    let p be the index of the pivot
    quicksort(vector, start, p - 1)
    quicksort(vector, p + 1, end)
```

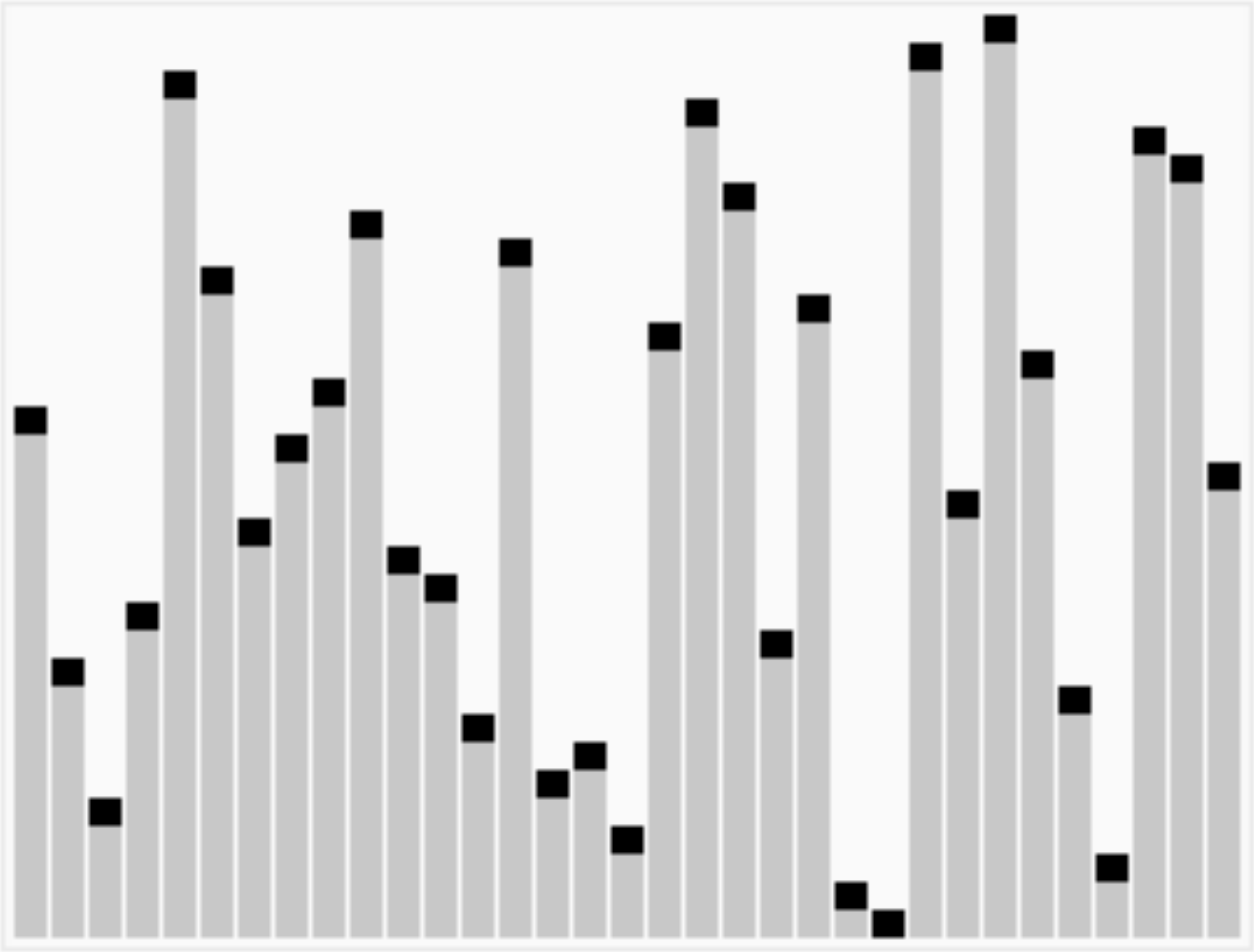
Quicksort

// pseudo-code (see: Stephens, p. 145)

```
quicksort(vector, start, end)
    pick an element to divide the array (a.k.a. "pivot")
    move items less than the pivot to the left of the pivot
    move items greater than or equal to the pivot to the right of the pivot
    let p be the index of the pivot
    quicksort(vector, start, p - 1)
    quicksort(vector, p + 1, end)
```

Quicksort

Animation



Animation source: Wikipedia

Quicksort

// pseudo-code (see: Stephens, p. 145)

```
quicksort(vector, start, end)
    pick an element to divide the array (a.k.a. "pivot")
    move items less than the pivot to the left of the pivot
    move items greater than or equal to the pivot to the right of the pivot
    let p be the index of the pivot
    quicksort(vector, start, p - 1)
    quicksort(vector, p + 1, end)
```

This is the basic idea, but there are lots of implementation details to consider!

Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

Quicksort

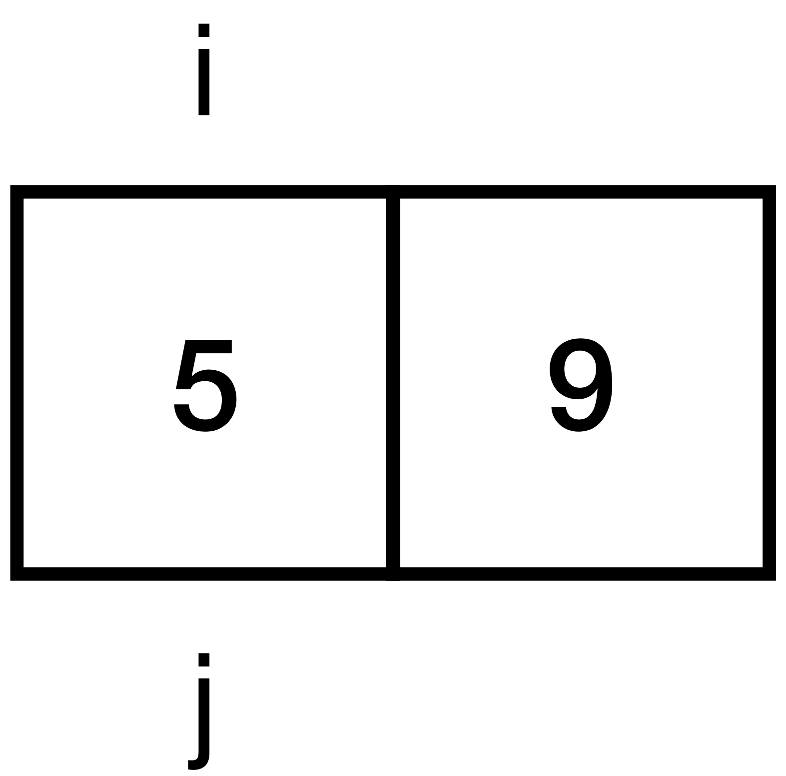
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

$\boxed{5}$

Quicksort

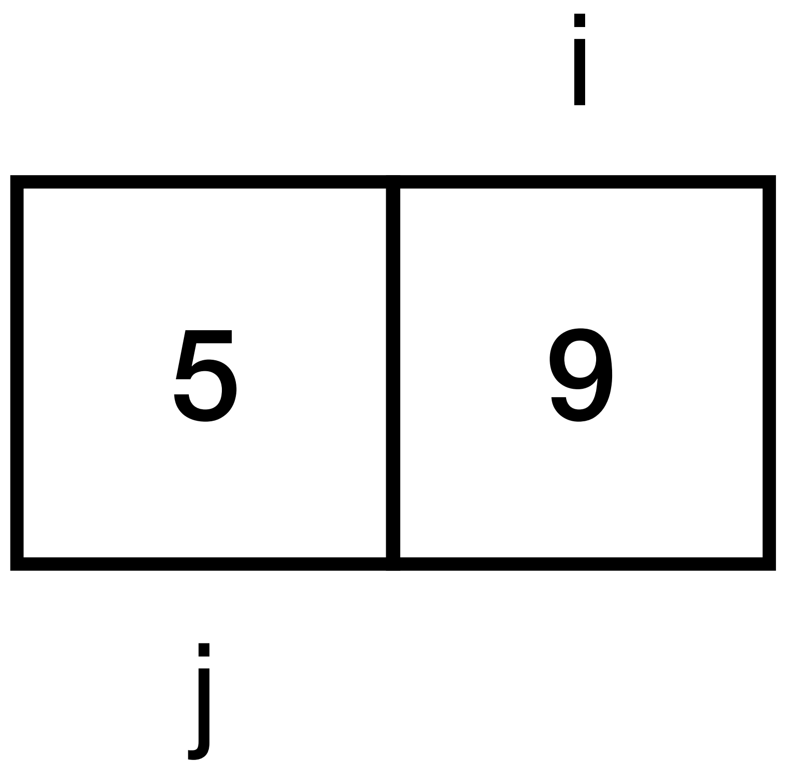
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

pivot = 9



Quicksort

pivot = 9

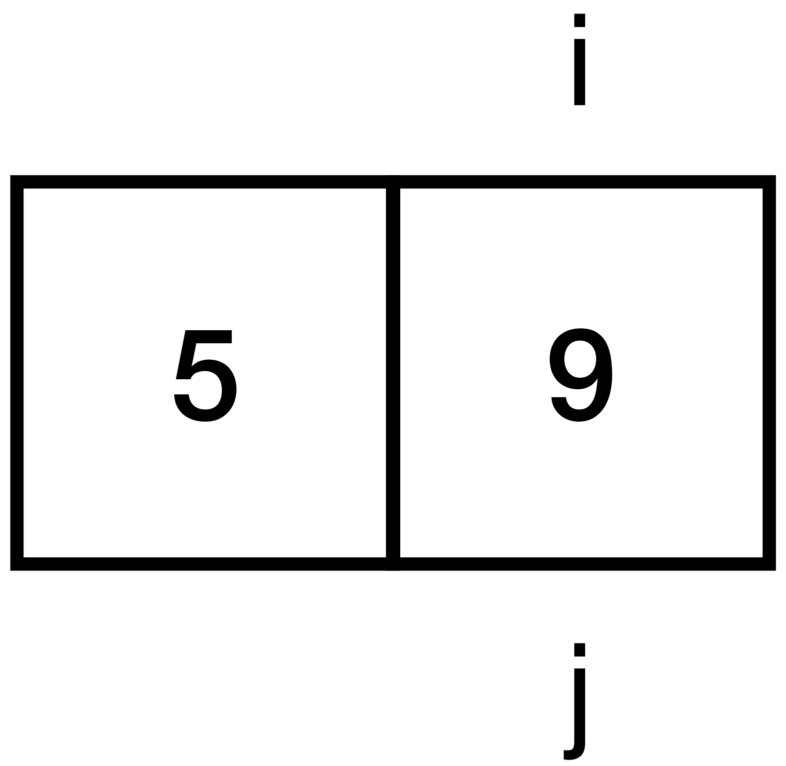



```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

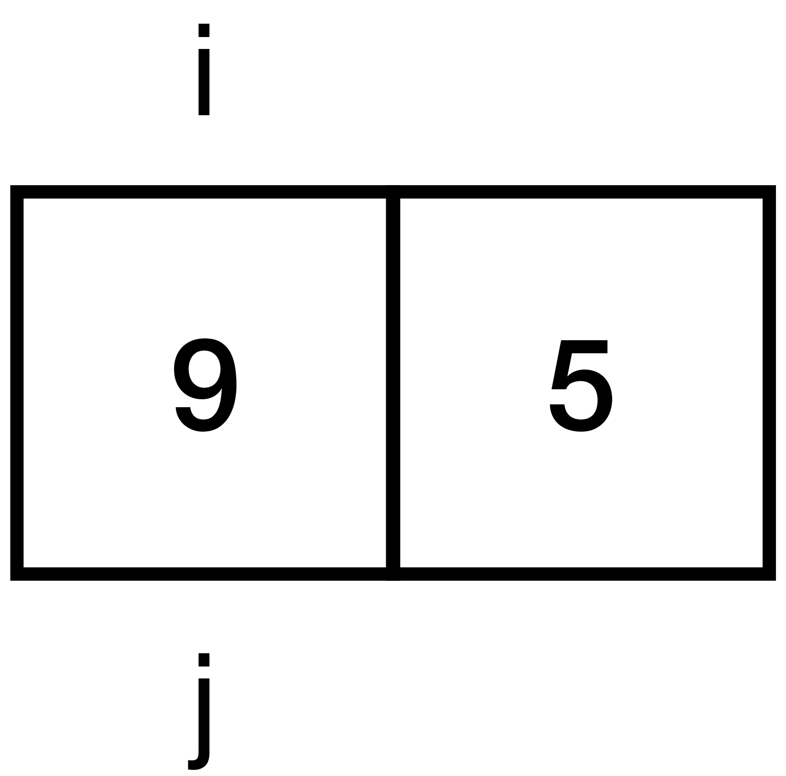
pivot = 9



Quicksort

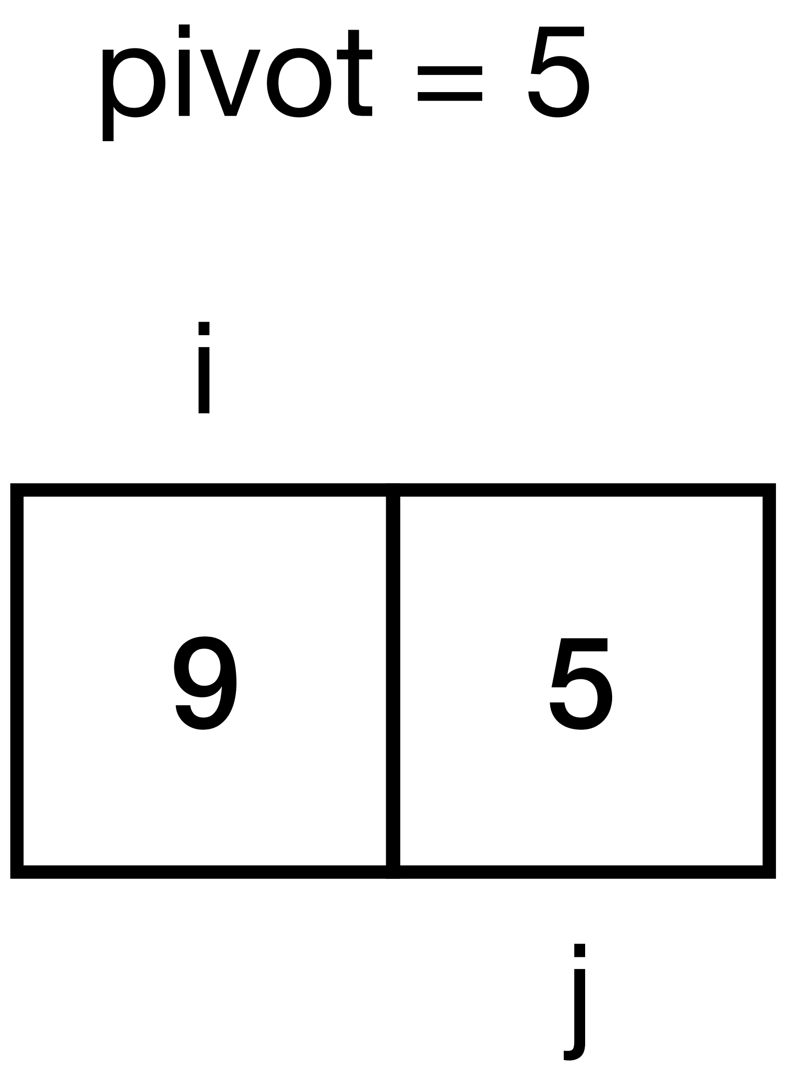
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

pivot = 5



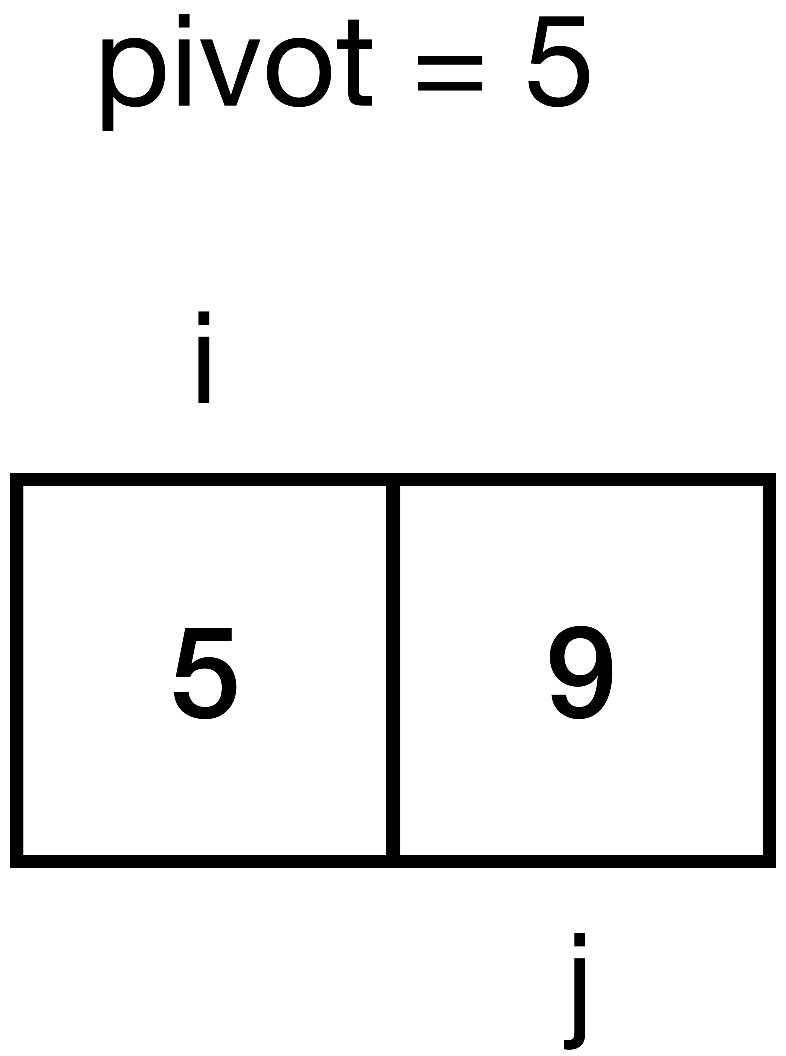
Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



Quicksort

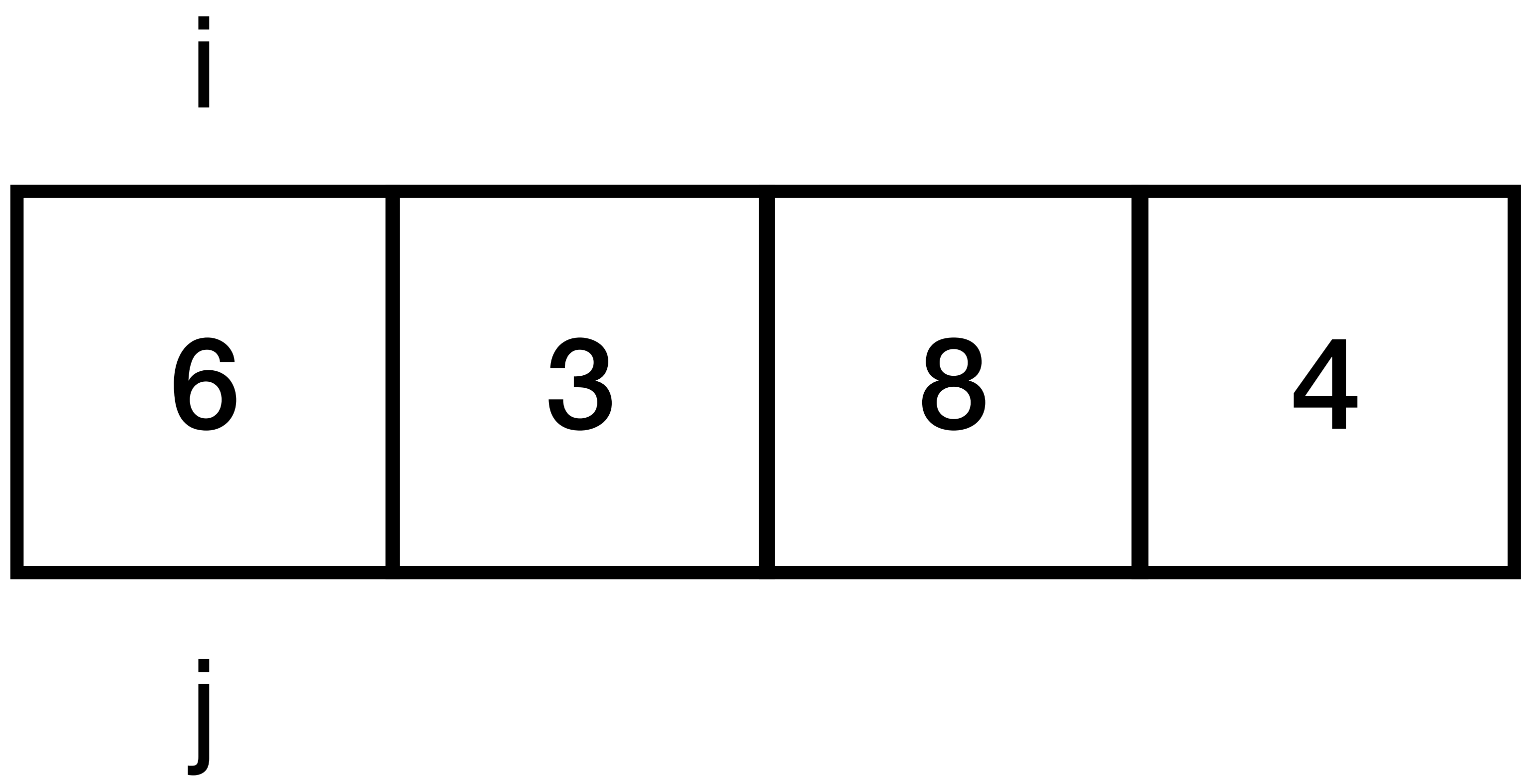
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

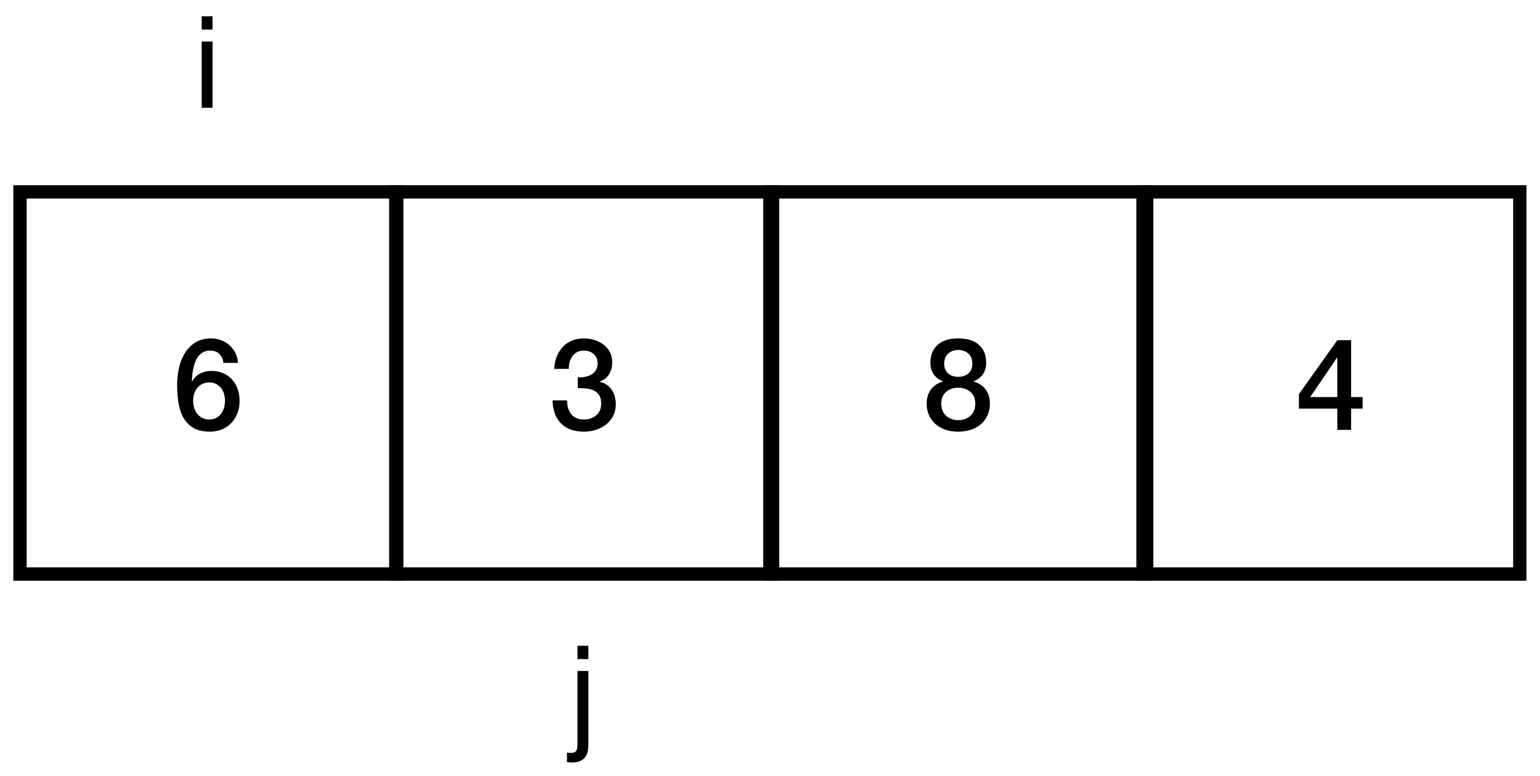
pivot = 4



Quicksort

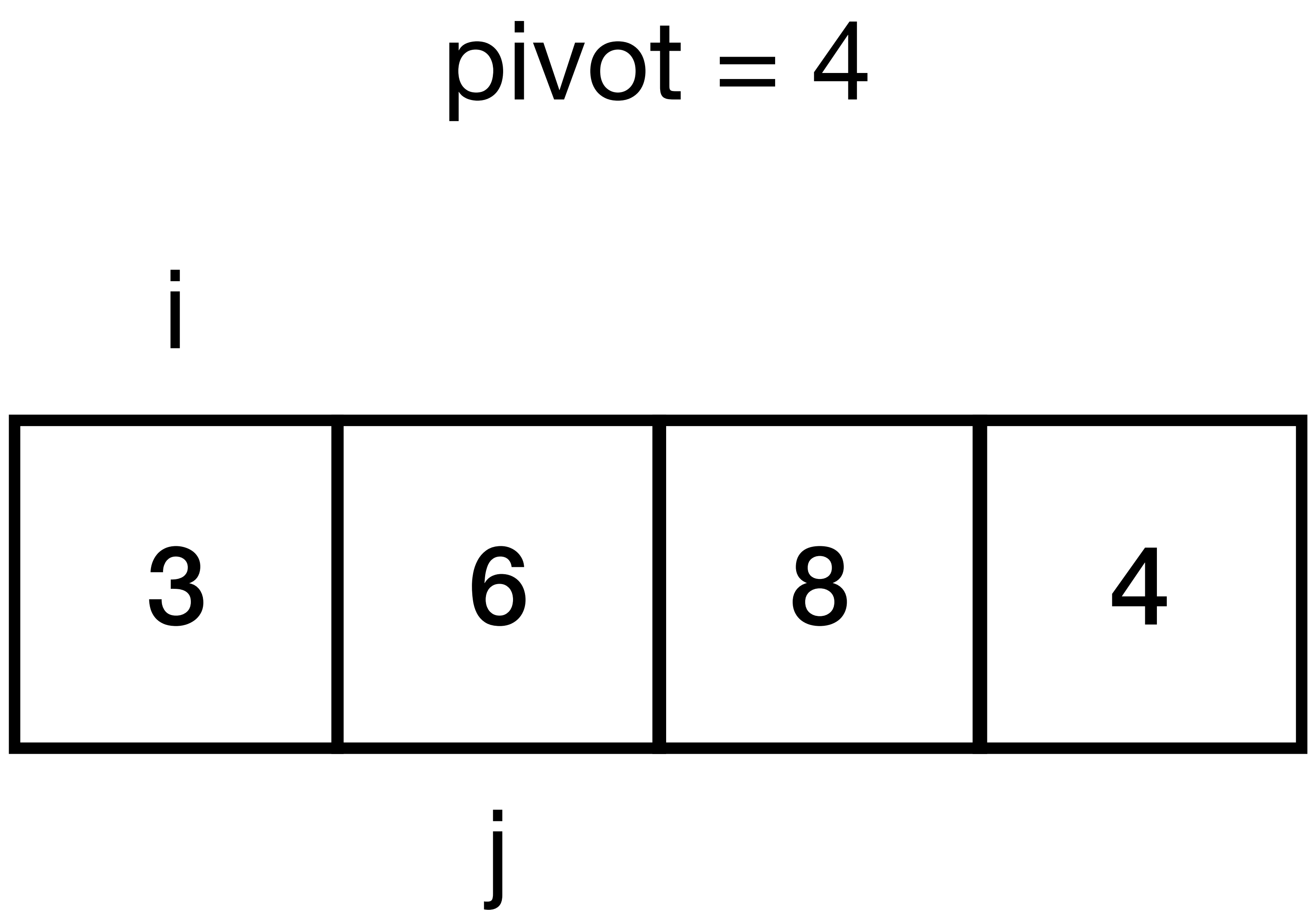
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

pivot = 4



Quicksort

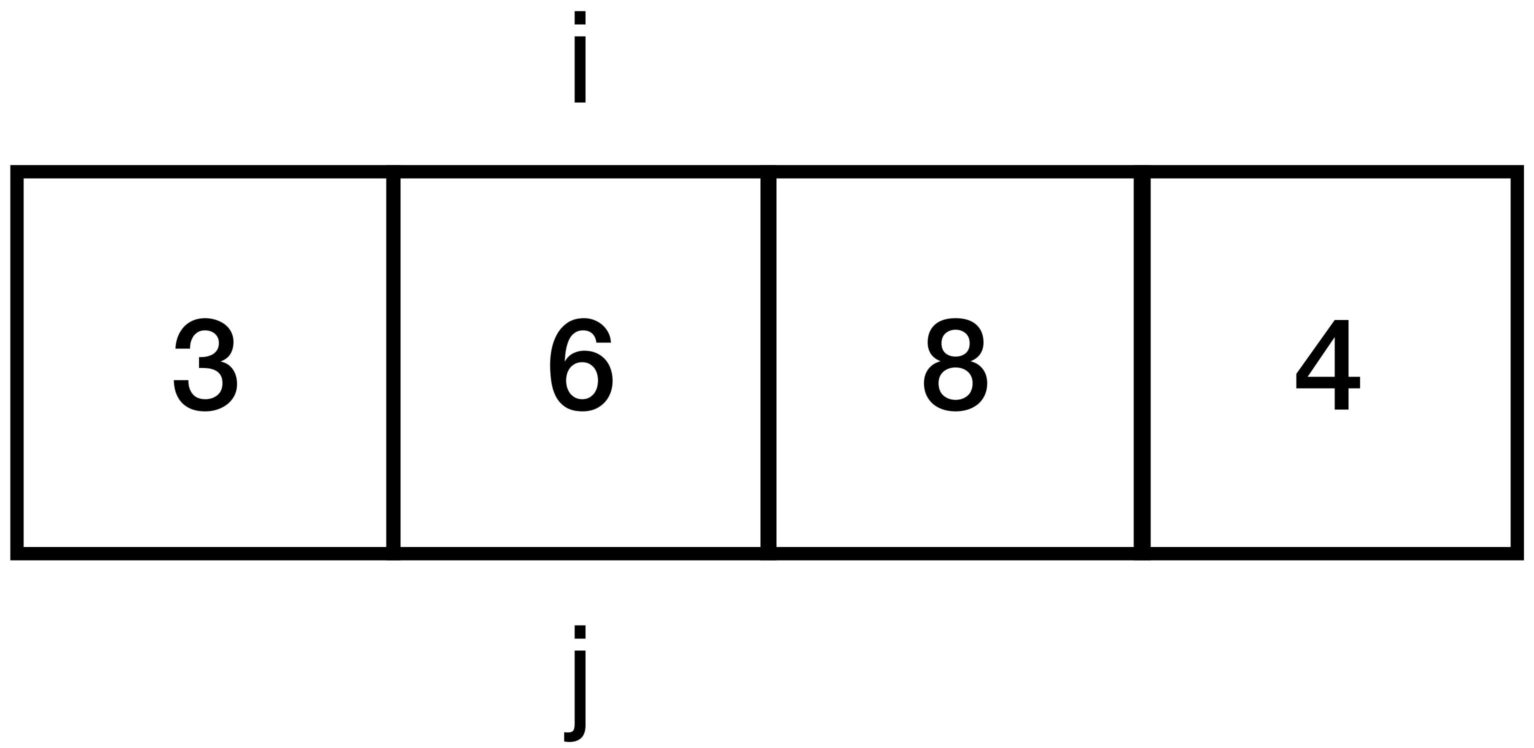
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



Quicksort

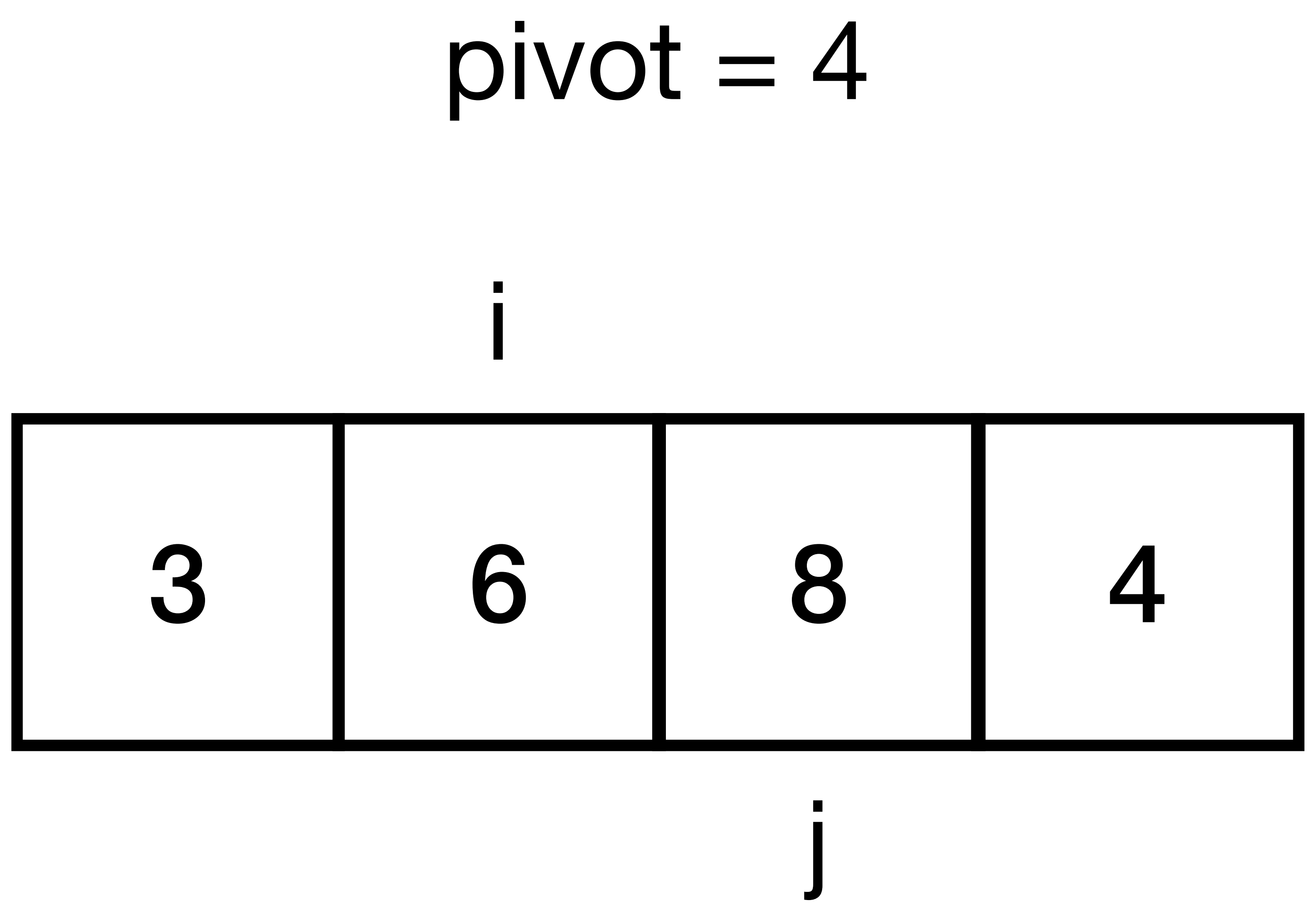
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

pivot = 4



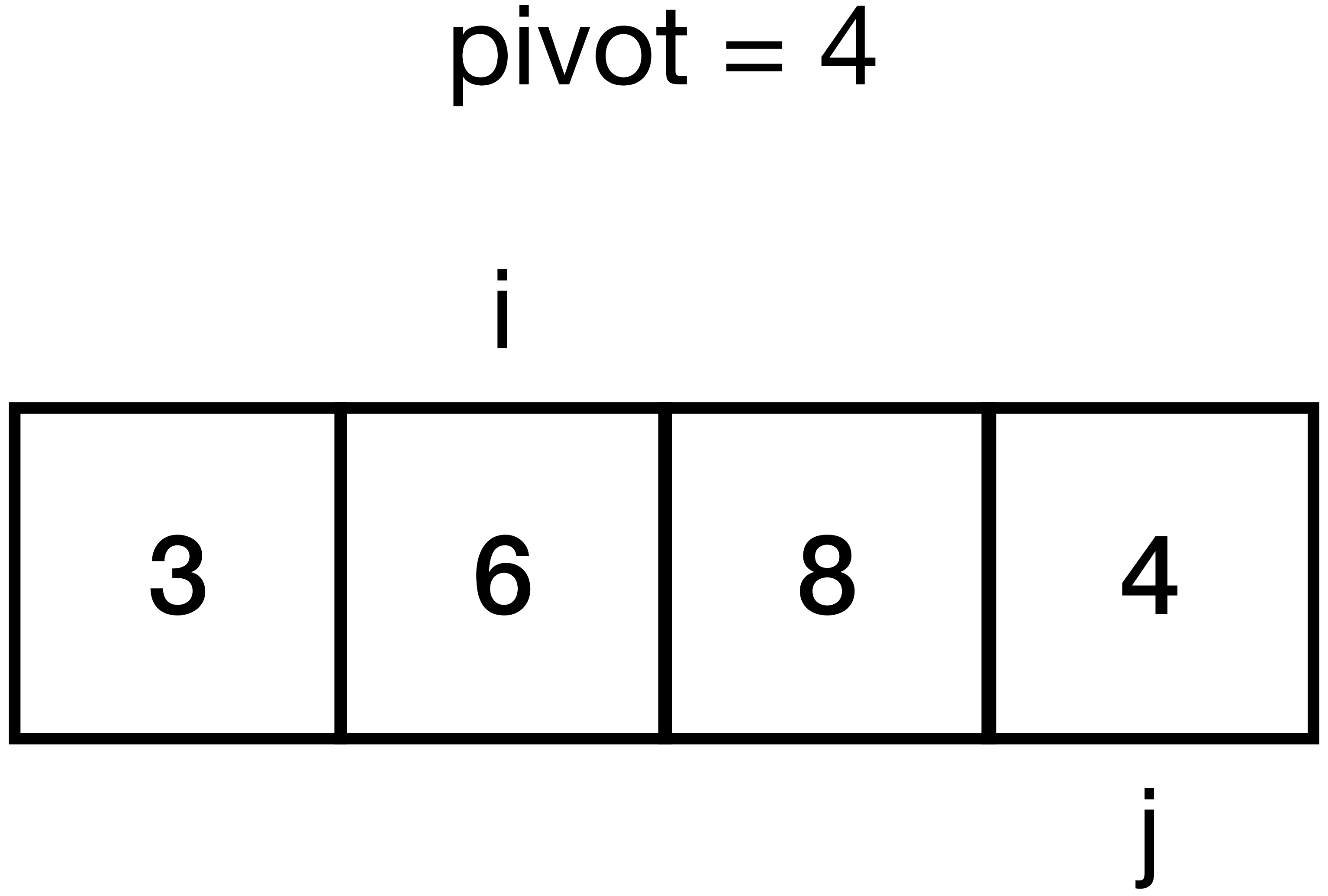
Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



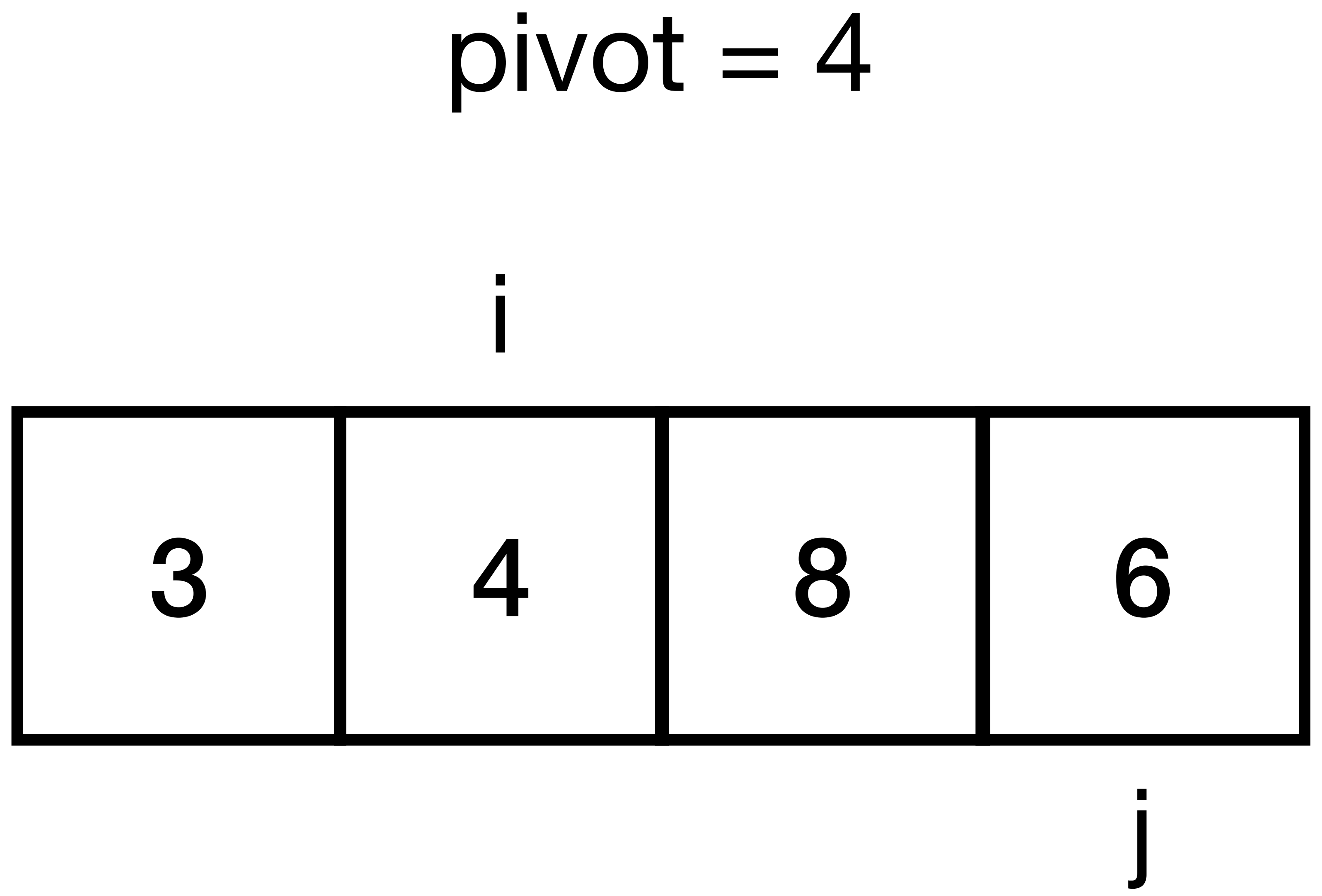
Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



Quicksort

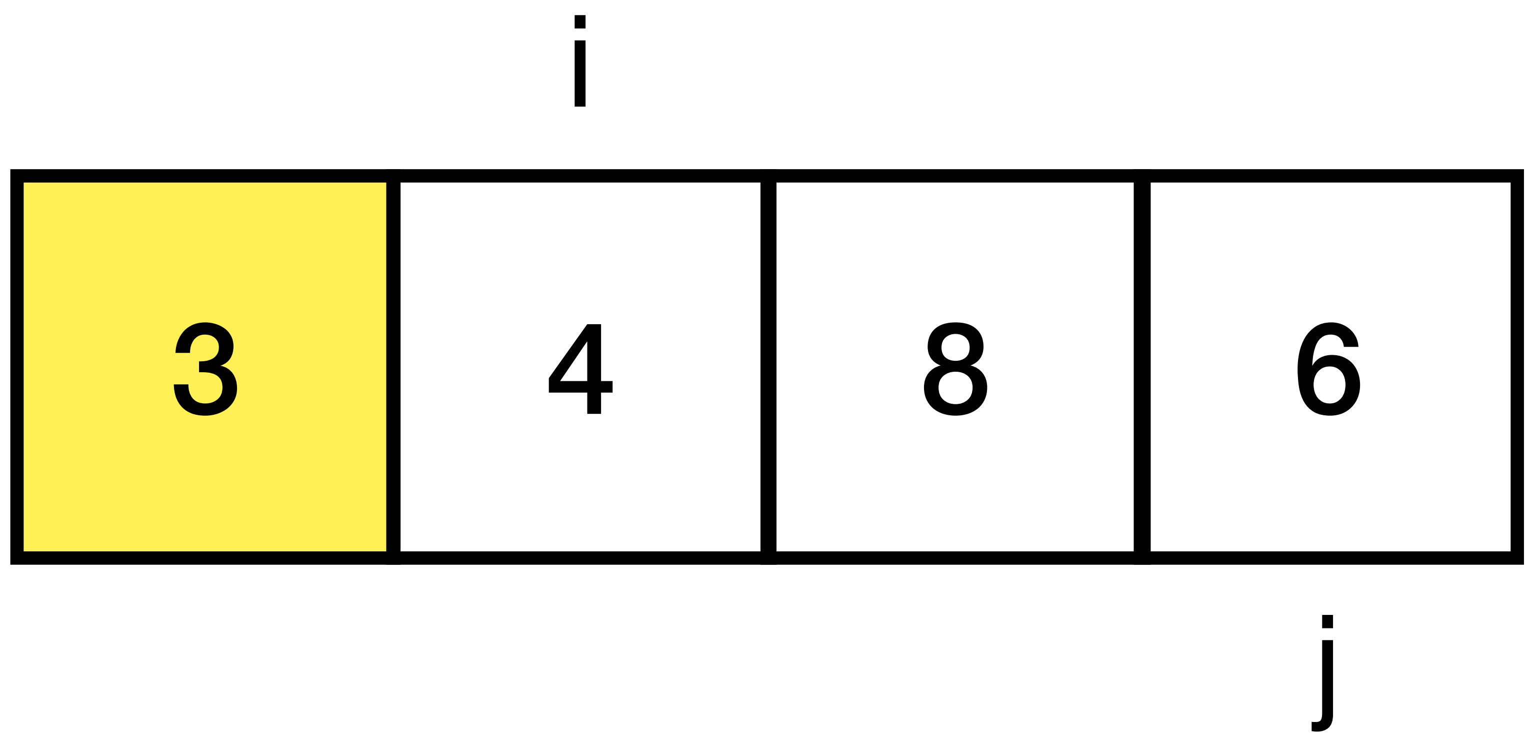
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



Quicksort

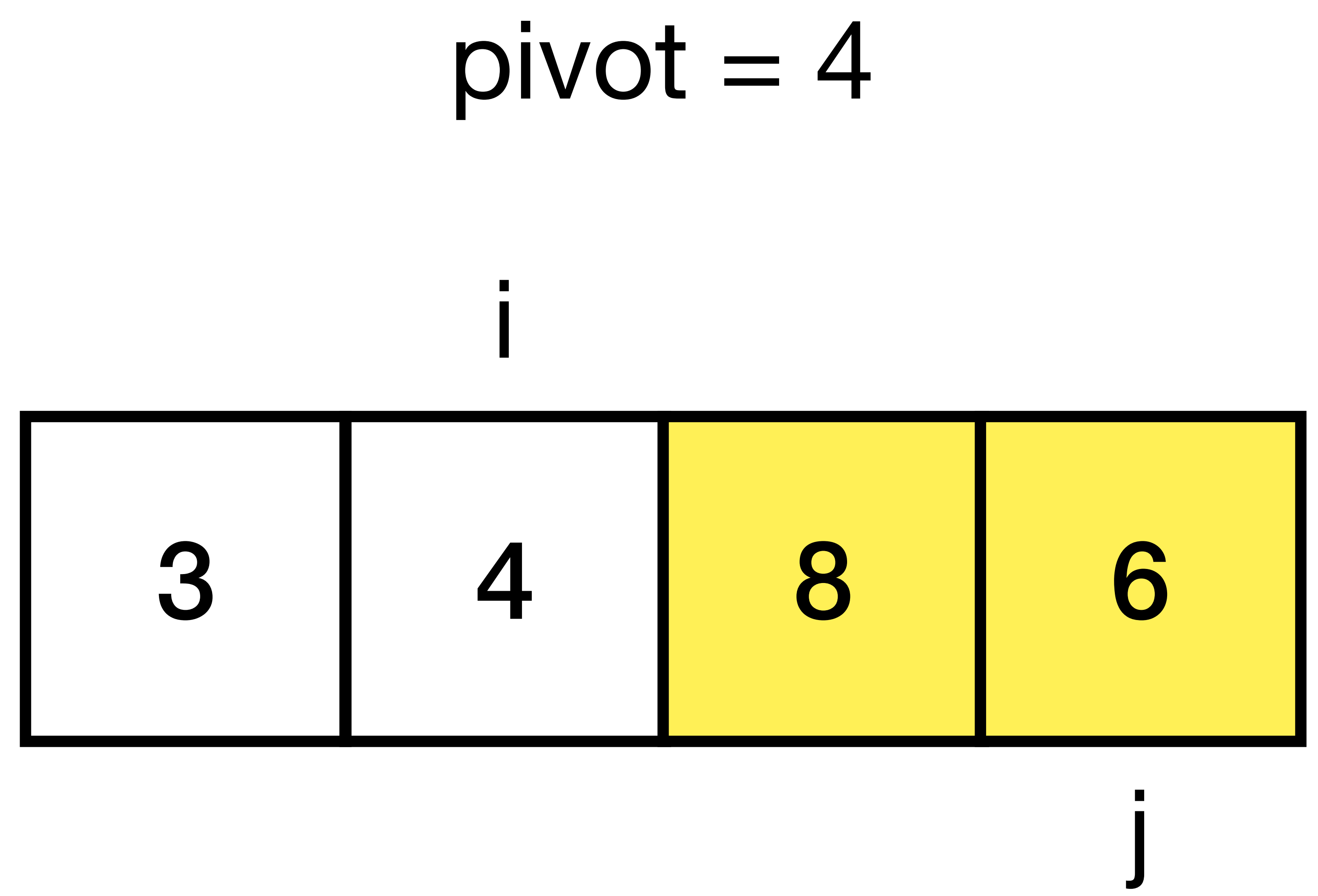
```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```

pivot = 4



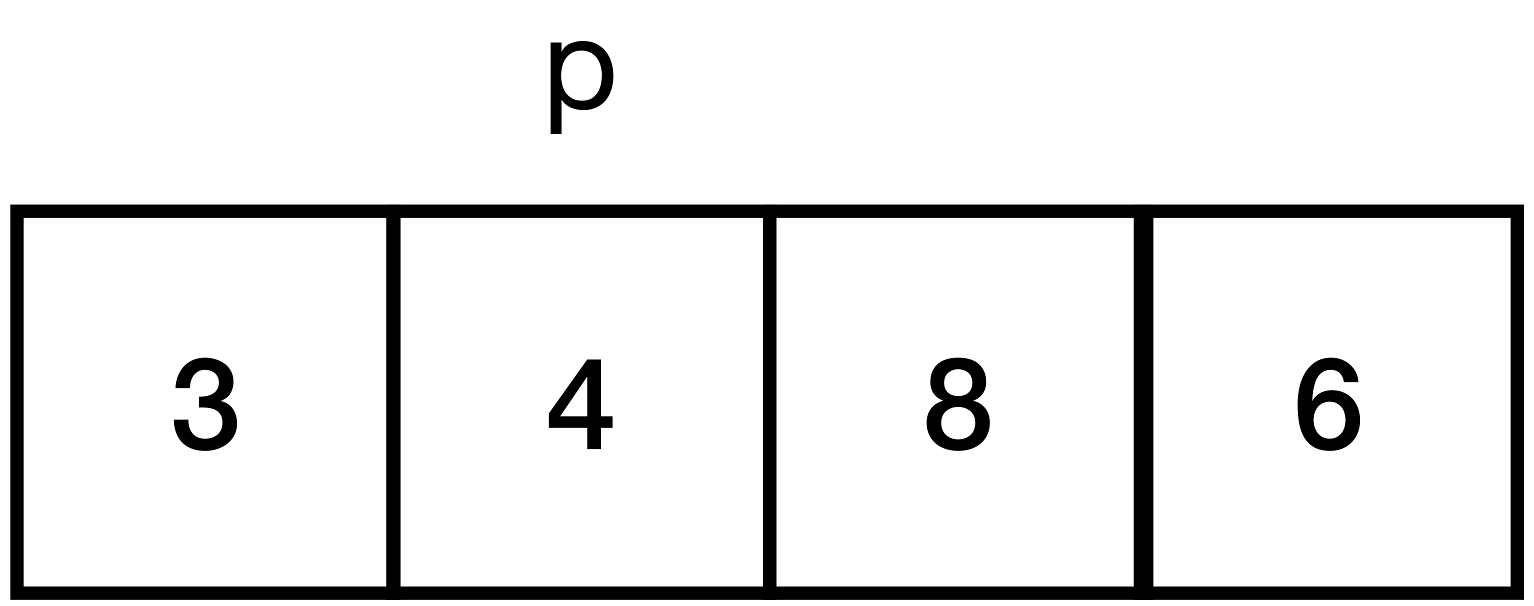
Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



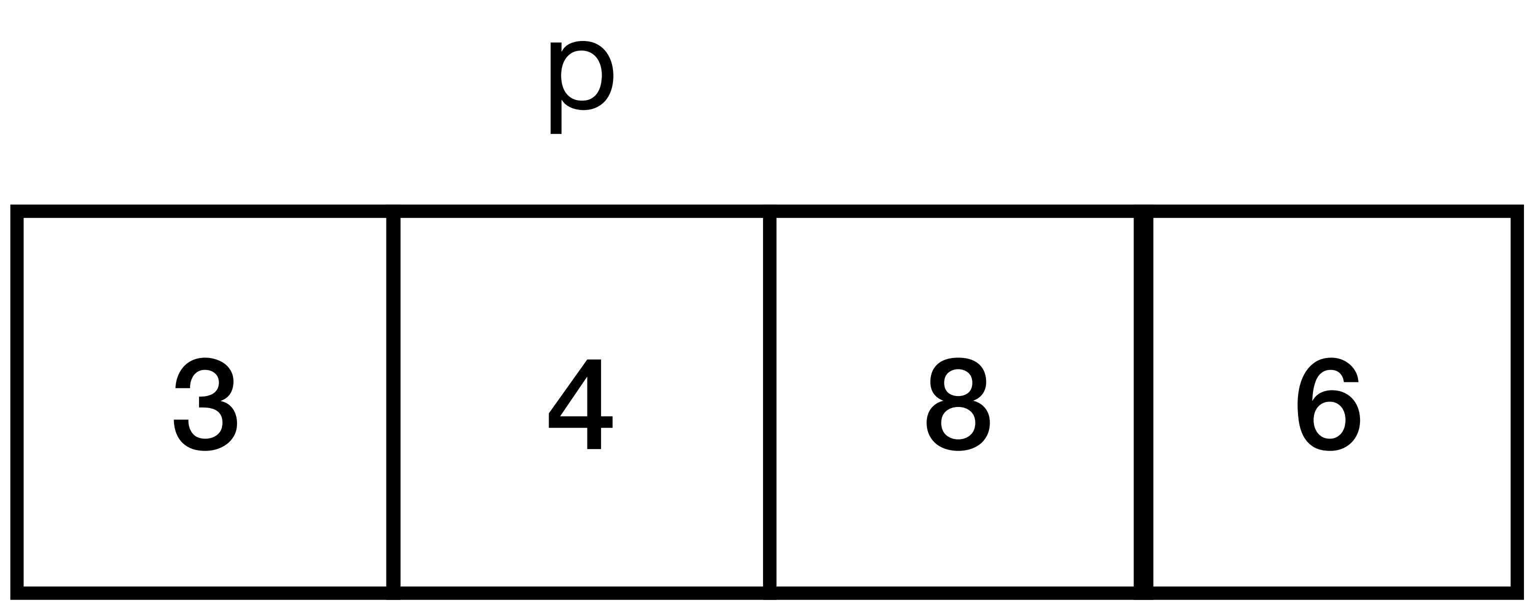
Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



Quicksort

```
partition(vector, start, end)
    pivot = vector[end]
    i = start
    for j from start to end
        if vector[j] < pivot then
            swap vector[i] and vector[j]
            i = i + 1
    swap vector[i] and vector[end]
    return i

quicksort(vector, start, end)
    if start < end then
        p = partition(vector, start, end)
        quicksort(vector, start, p - 1)
        quicksort(vector, p + 1, end)
```



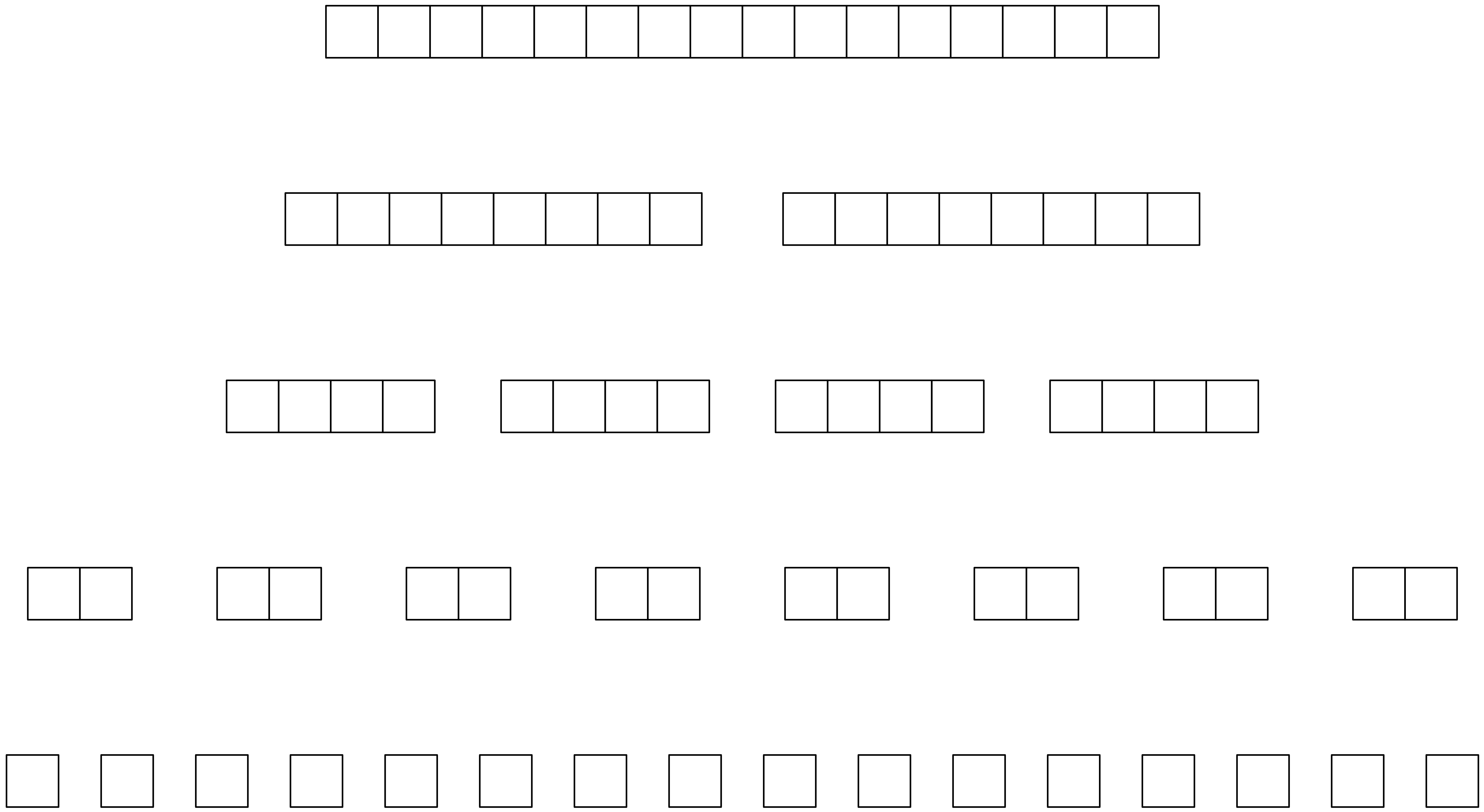
Quicksort

Quicksort has many different implementations and performance varies significantly with implementation.

- How do we choose a pivot (divider)? There are many schemes.
- Quicksort doesn't do well on sorted lists or when there are many repeated values. Quicksort can have worst-case complexity of $O(n^2)$.
- In the best cases, with a good implementation, quicksort outperforms merge sort (and heap sort, which we'll see a little later).
- Sometimes it is implemented in a hybrid form, with insertion sort used when subproblems become small.

Merge sort

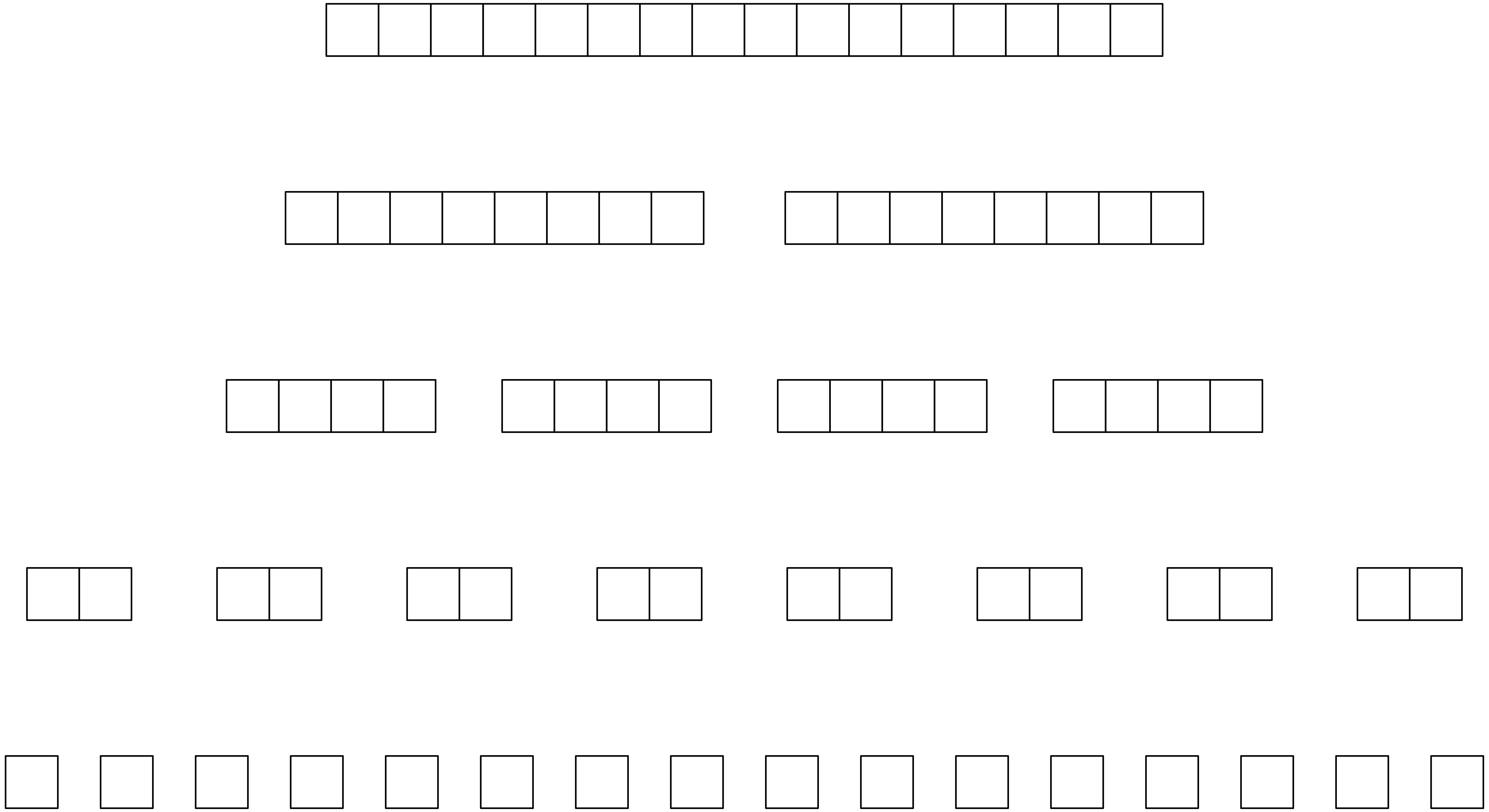
Recursion depth = log (n)



Quicksort

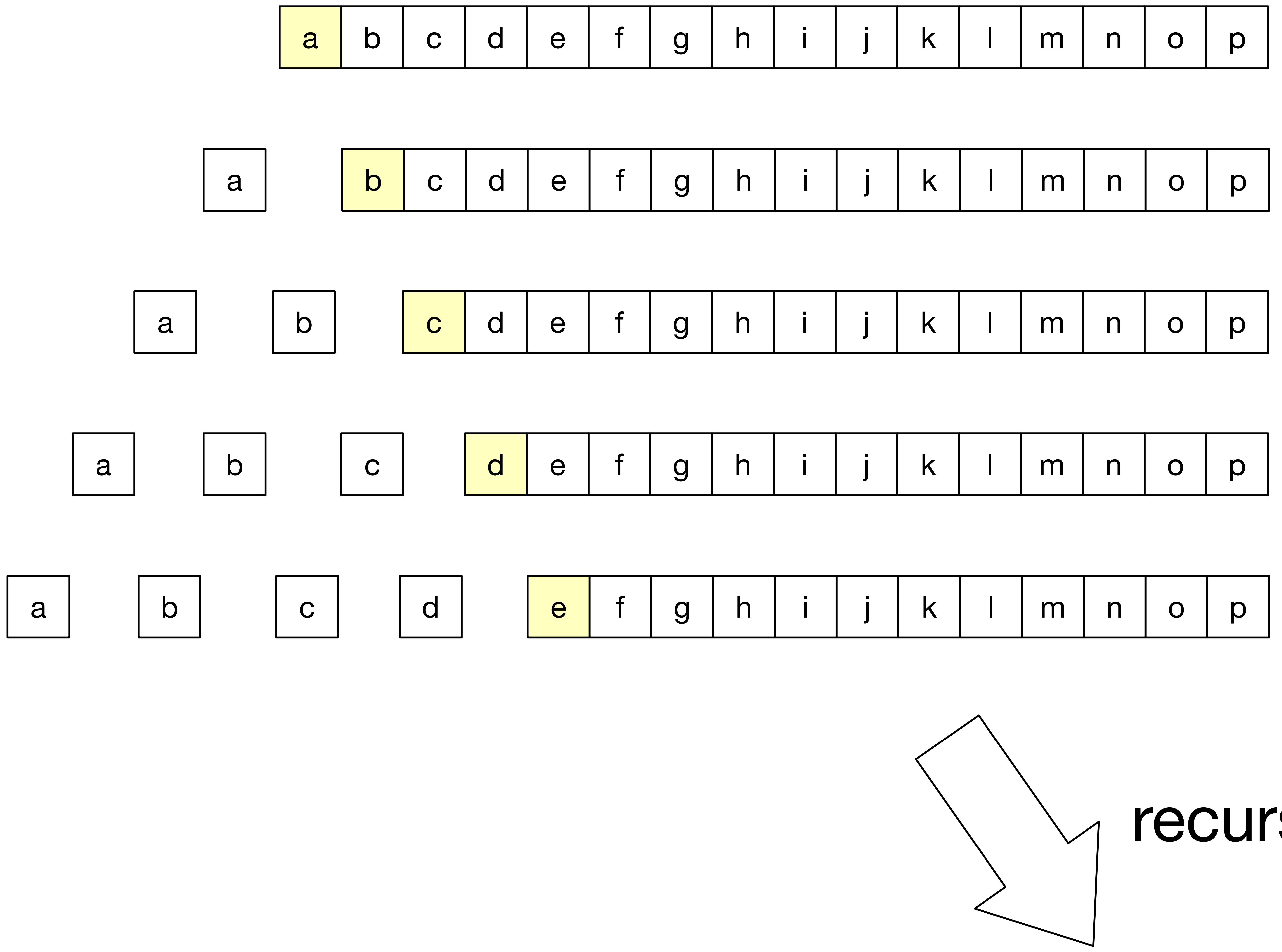
Typical case

Recursion depth $\cong \log(n)$



Quicksort

Worst case



recursion depth = n

Quicksort

Choosing a pivot

Method	Pro	Con
Choose element at one end of the vector	Works pretty well in most cases; super easy implementation	Can be *horrible* with sorted or partially sorted list
Pick at random	Works pretty well in most cases	Cost of choosing at random; not really worth the trouble
Median-of-three	Works pretty well in most cases and not too costly; reduces likelihood of worst-case performance	A little overhead (three extra comparisons and up to three extra swaps)

Quicksort

Median-of-three

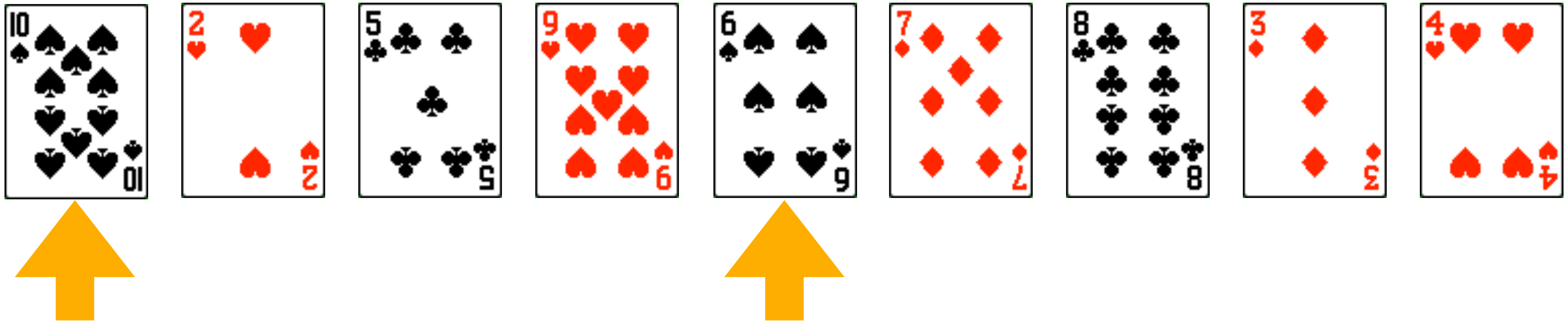
```
// pseudo-code, given some vector v, and start and end indices

middle = (start + end) / 2;
if element indexed by middle is less than element indexed by start
    swap middle and start
if element indexed by end is less than element indexed by start
    swap end and start
if element indexed by middle is less then element indexed by end
    swap middle and end
let the pivot be the element now at the end position
```

Quicksort

Median-of-three

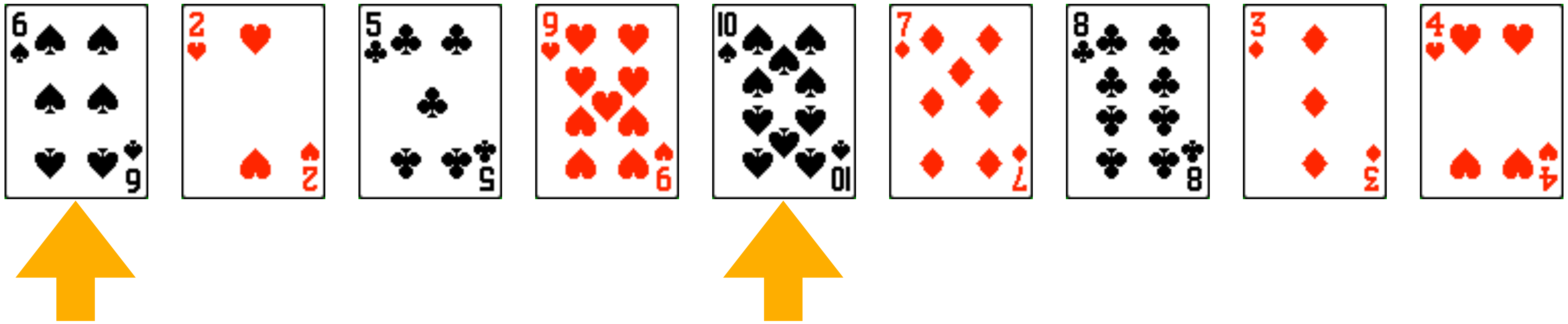
Quicksort

Median-of-three



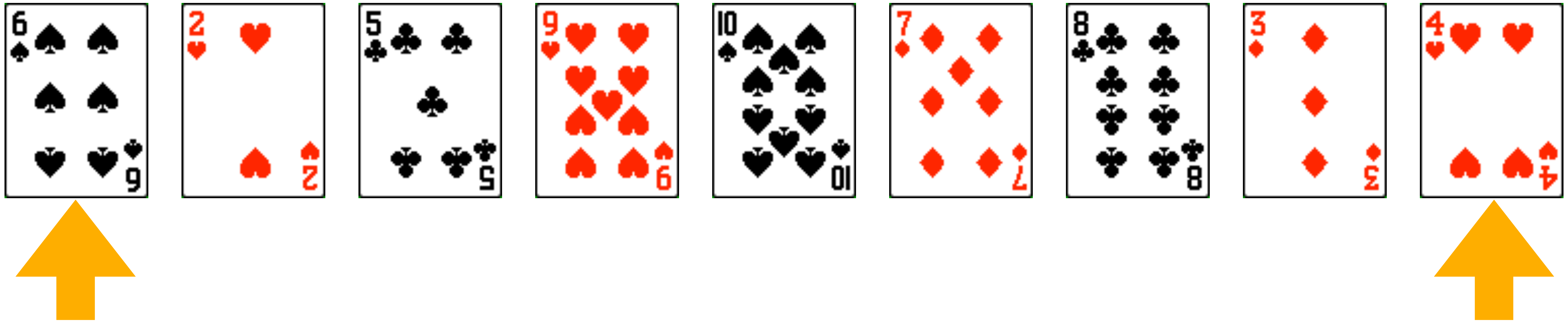
Quicksort

Median-of-three



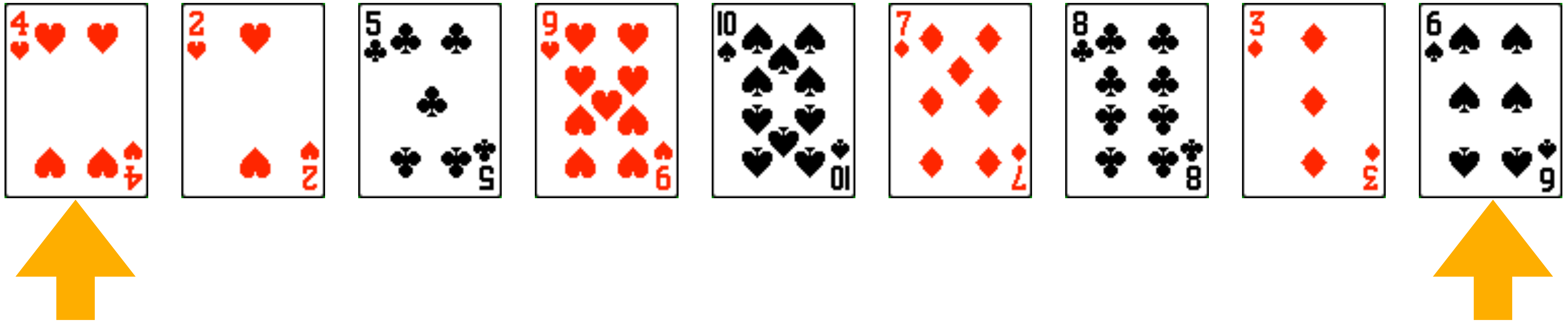
Quicksort

Median-of-three



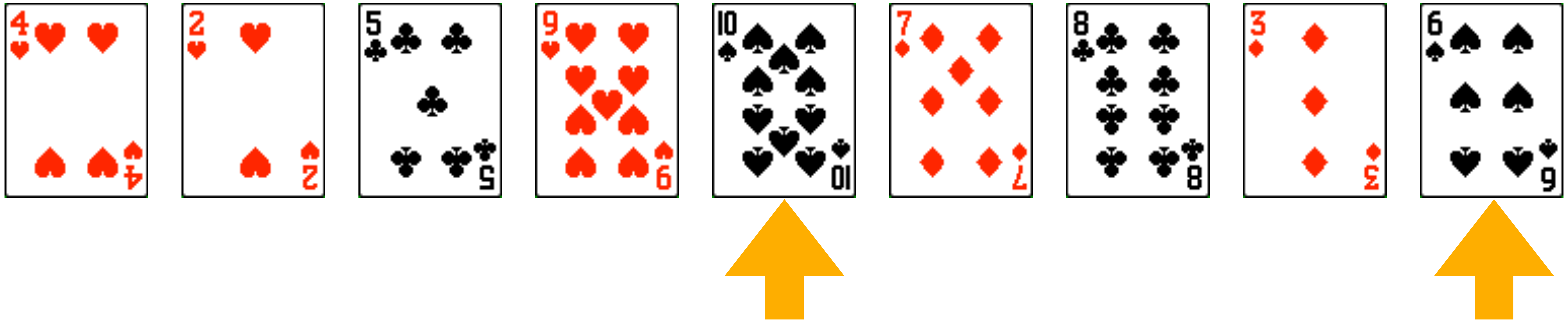
Quicksort

Median-of-three



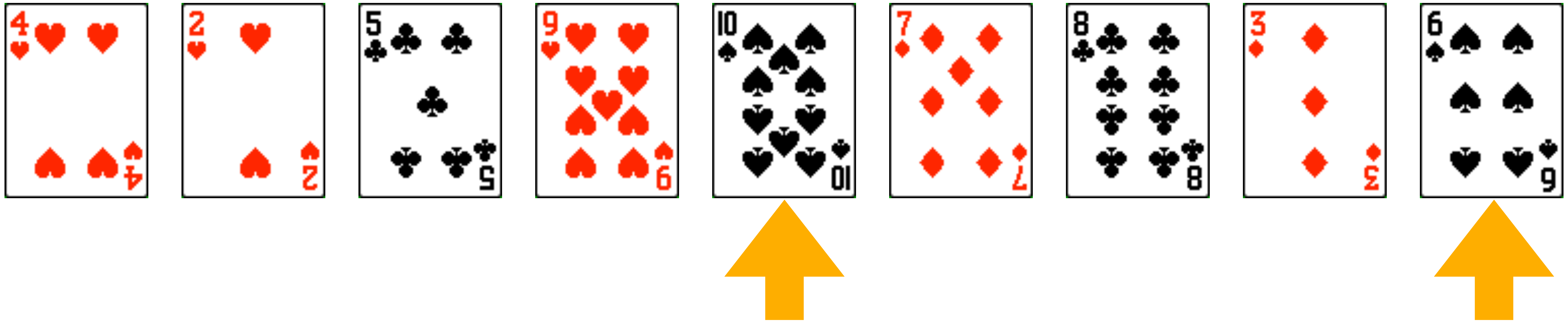
Quicksort

Median-of-three



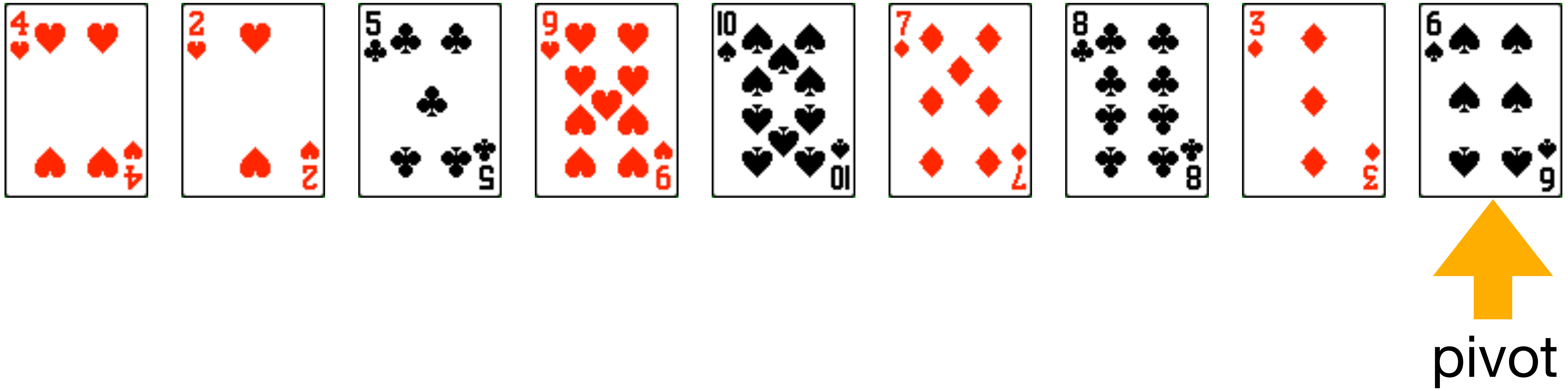
Quicksort

Median-of-three



Quicksort

Median-of-three



Quicksort Complexity

The partition function must process $O(n)$ elements, and the recursion depth is typically $O(\log n)$, hence $O(n \log n)$. This is the best case and average case.

The worst case is, as we have seen, when we have a linear chain of partitions, and a recursion depth of $O(n)$. Again, the partition function processes $O(n)$ elements so we have $O(n^2)$.

However, we can avoid worst-case performance in many situations by choosing our pivot well. This is why many implementations of quicksort use the median-of-three approach, or something similar.

Quicksort Stability

Is quicksort stable? MAYBE

Quicksort hybrid

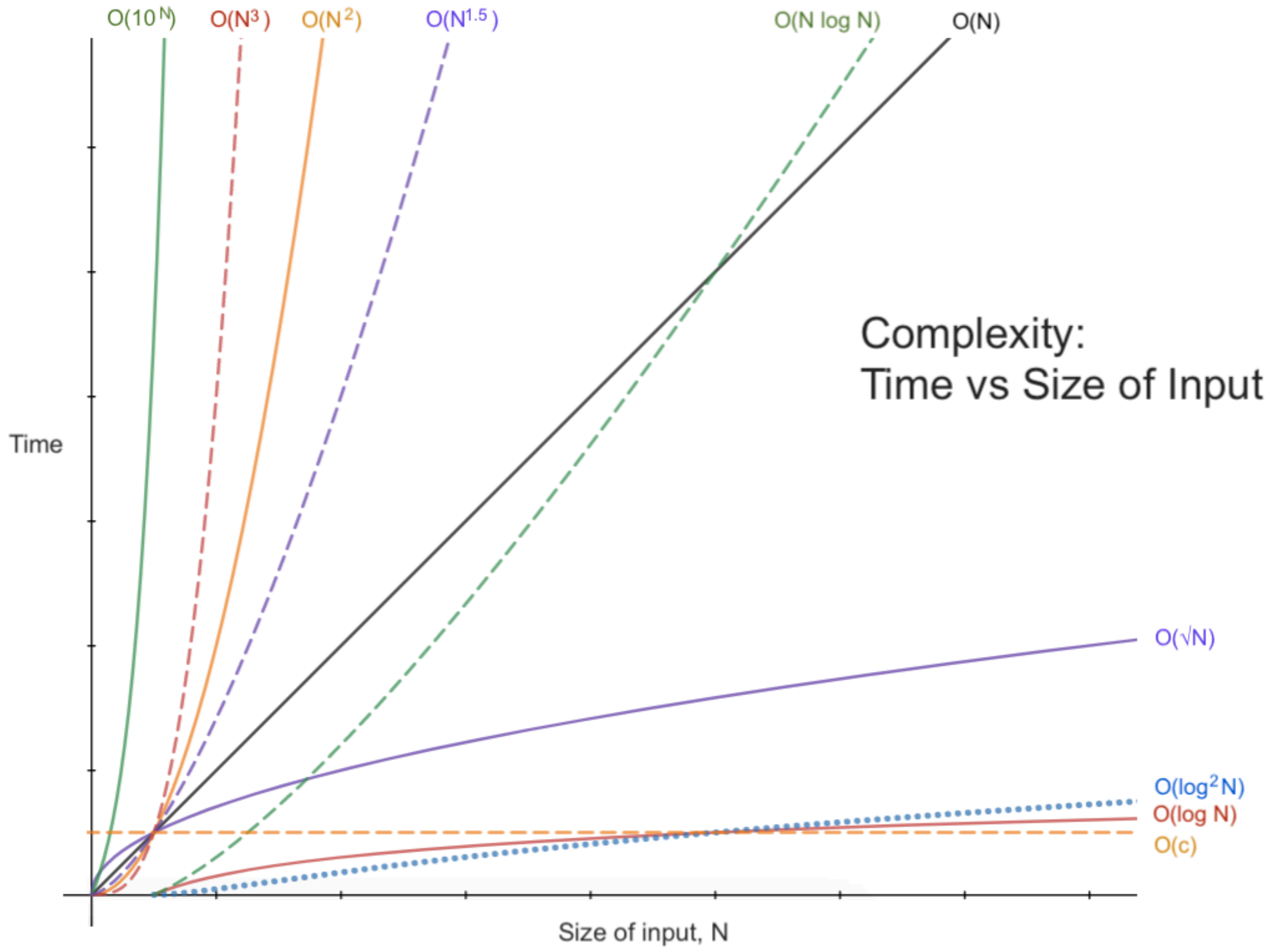
It is not uncommon that quicksort is hybridized with another algorithm -- typically insertion sort.

When we have small vectors, we've seen that swapping an element with itself is not uncommon. One way to address this is to use insertion sort for vectors below a certain size, say around ten elements. This gives quicksort a modest speedup in many cases.

Comparison

Algorithm	Time complexity	Space complexity	Stable	Comment
Bubble sort	$O(n^2)$	$O(1)$	yes	can tell if list is already sorted
Selection sort	$O(n^2)$	$O(1)$	no	performs fewest swaps
Insertion sort	$O(n^2)$	$O(1)$	yes	ignores unsorted portion of vector / can process data on-line
Merge sort	$O(n \log n)$	$O(n)$	yes	recursive divide-and-conquer
Quicksort	$O(n \log n)$	$O(1)$	no	recursive divide-and-conquer

Comparison


O(10^N)
O(N^3)
O(N^2)
O(N^1.5)
O(N log N)
O(N)
Complexity:
Time vs Size of Input
Time
O(√N)
O(log^2 N)
O(log N)
O(c)
Size of input, N

Summary

- Quicksort is a recursive divide-and-conquer algorithm.
- Quicksort has $O(n \log n)$ time complexity.
- Quicksort has $O(1)$ space complexity.
- Quicksort in its original design is not stable, but at the expense of increasing space complexity it can be implemented as a stable algorithm.