11 Quick Sort

Key Ideas: Quicksort

- Select some element of the array, which we will call the *pivot*.
- Partition the array so that all items less then the pivot are to its left, all elements equal to the pivot are in the middle, and all elements greater than the pivot are to its right.
- Quicksort the left part of the array (elements less than the pivot).
- Quicksort the right part of the array (elements greater than the pivot).

The key ideas are very simple. The two major components to correctly (and efficiently) implementing Quicksort are:

1) - the partitioning algorithm (how does the algorithm move elements lesser/greater than the pivot to the left/right efficiently?)
2) - how does the algorithm select the pivot element?

1) The Partitioning Prolem:

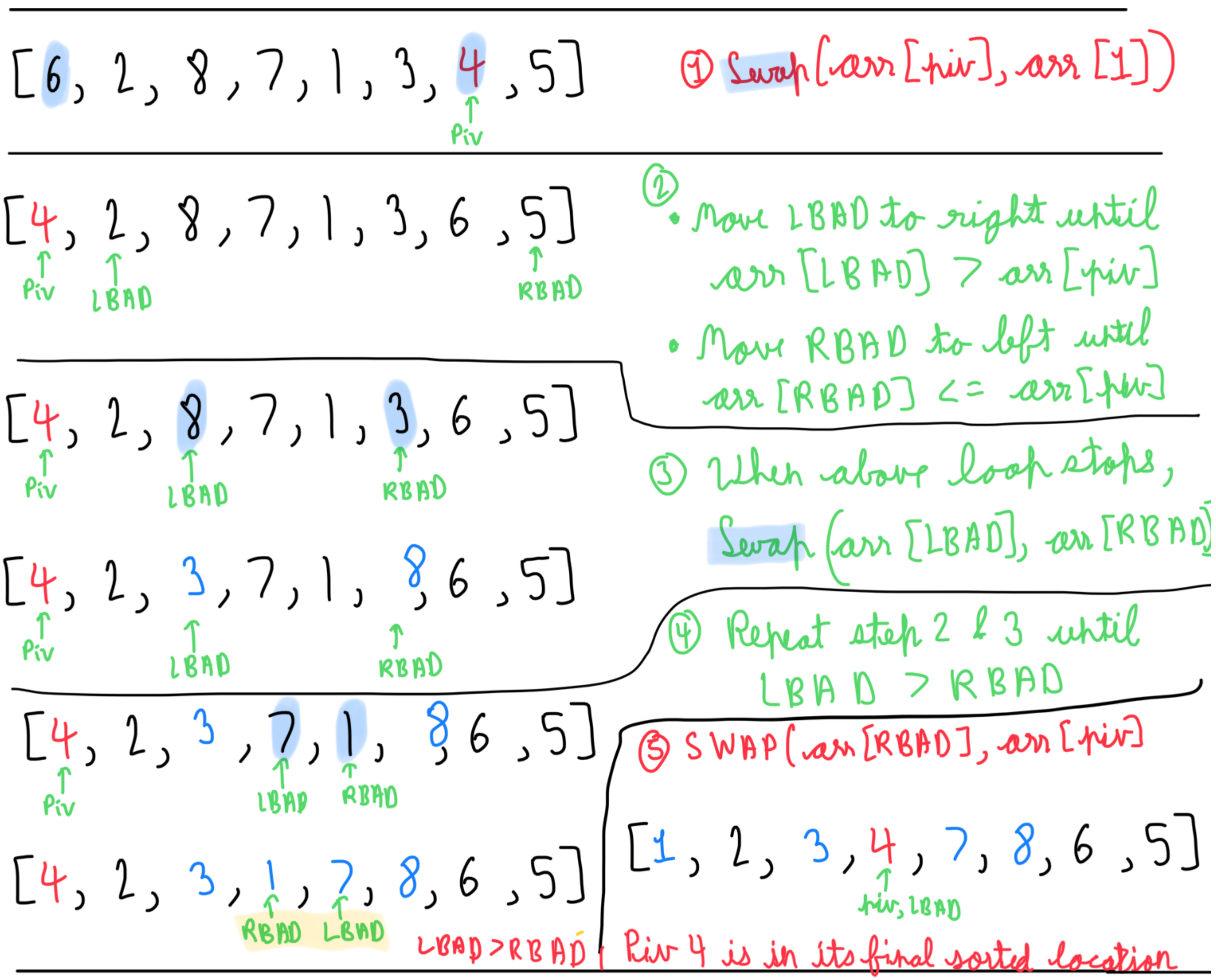
Choice of choosing a partioning algoritm affects whether Quick sort is In-place, Stable, and worst-case time complexity.

a) Naive Partioning:

```
def partition(arr[lo...hi], pivot):
    left, pivots, right = [], [], []
    for (i = lo to hi):
        if (arr[i] < pivot):    left.append(arr[i])
        elif (arr[i] == pivot): pivots.append(arr[i])
        else:                   right.append(arr[i])
    arr[lo...hi] = left + pivots + right
    return len(left) + ⌈len(pivots) / 2⌉ // retur idx of middle pivot
```

Time: $O(n)$ ✓
Is Stable ✓
NOT In-place ✗

b) Hoare partition algorithm:



① Swap(arr[piv], arr[1])

② • Move LBAD to right until arr[LBAD] > arr[piv]

• Move RBAD to left until arr[RBAD] <= arr[piv]

③ When above loop stops, Swap(arr[LBAD], arr[RBAD])

④ Repeat step 2 & 3 until LBAD > RBAD

⑤ SWAP(arr[RBAD], arr[piv])

LBAD > RBAD ; Piv 4 is in its final sorted location

```
def partition(arr[lo...hi], arr[p]):

    Swap(arr[lo], arr[p])
    i, j = lo+1, hi

  LI: arr[1...i-1] <= pivot & arr[j+1...N] > pivot
    while (i <= hi):
          LI: arr[1...i-1] <= pivot > arr[j+1...N]
          while (i <= j) and (arr[i] <= arr[lo]):  i += 1
          while (i <= j) and (arr[j] > arr[lo]):  j -= 1
```

Time: O(n) ✓
Not Stable ✗
Is In-Place ✓

```
        if (i <= j):   swap(arr[i], arr[j])
        LI:  arr[1... i-1] <= pivot > arr[j+1... N]

    swap( arr[lo] , arr[j])
        LI: arr[1... i-1] <= pivot   & arr[j+1] > pivot
    return j                         & arr[j] = pivot
```

Hoare Partion:

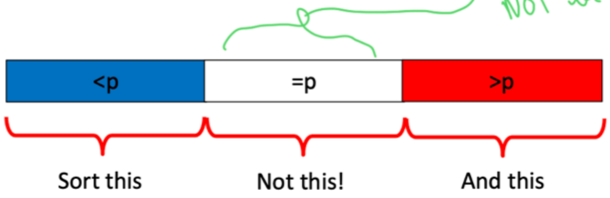
Pros: - each elem. only swapped once except pivot
- Simple idea
- Simple Invariant

Cons: - Not Stable
- Performs very badly when there are many elements that equal to the pivot, (
The partition is very unbalanced making Quick sort $O(n^2)$ when all items are same value.

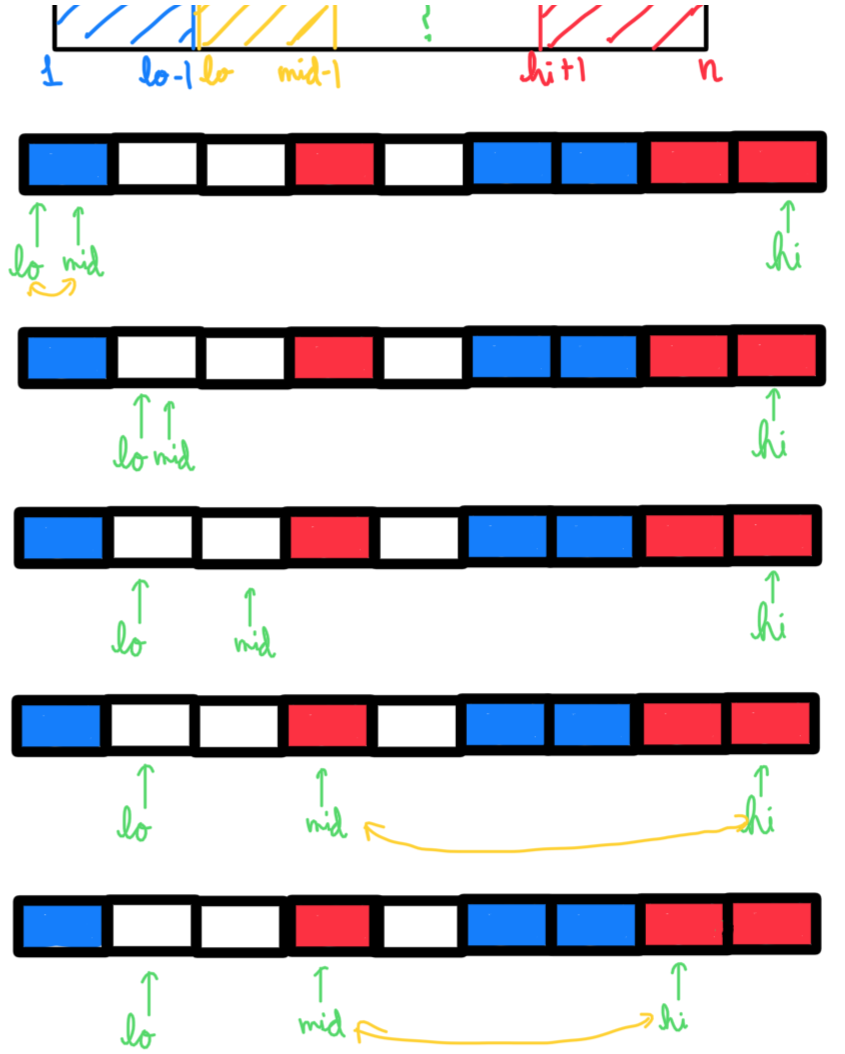
c) Dutch National Flag partion algorithm:

- If the list has many duplicates, then sometimes…
- One will be chosen as the pivot
- All the others **should** go next to the pivot (and therefore not need to be moved any more)
- But the algorithms we have seen would require them to be sorted in the recursive calls!
- We want a partition method that does this:

Elements = p should NOT be swapped.

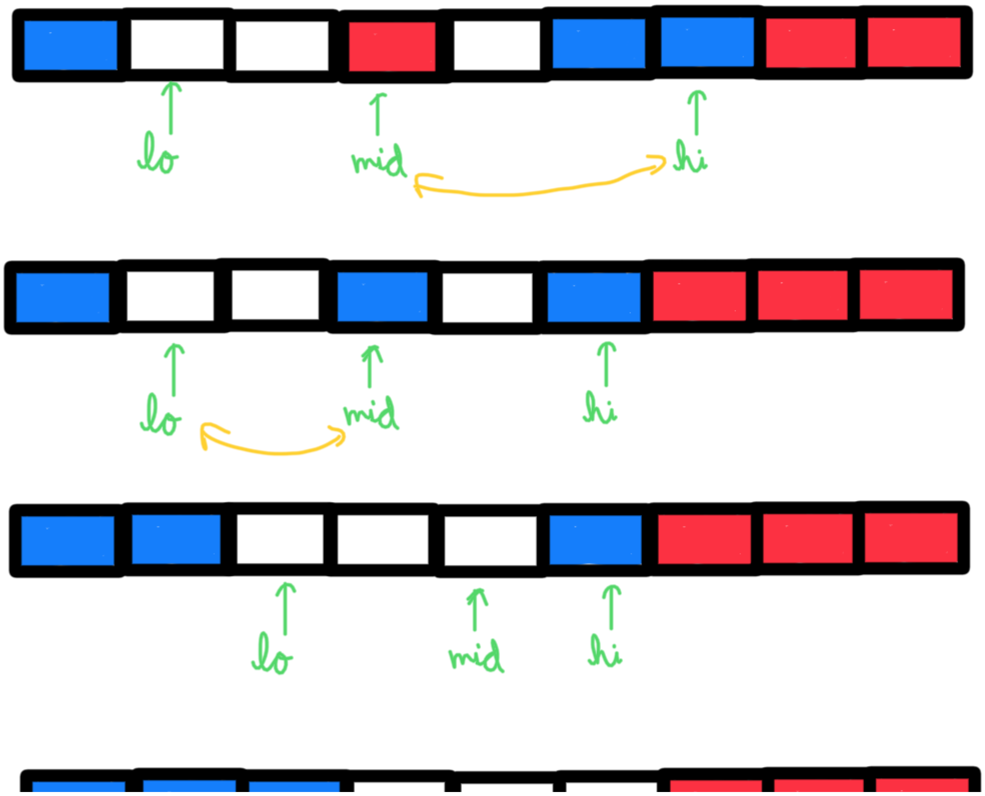


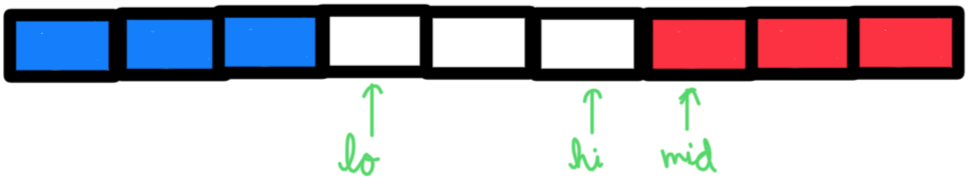
Time: $O(n)$ ✓
Not Stable ✓
In-Place ✗



```
① if (arr[mid] == red):
      swap(arr[mid], arr[hi])
      hi -= 1
② elif (arr[mid] == blue):
      swap(arr[mid], arr[lo])
      lo += 1
      mid += 1
③ else:
      mid += 1
```

Notice how arr[mid] is white OR = pivot, we do NOT SWAP!



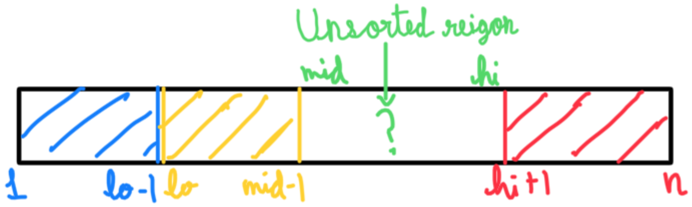


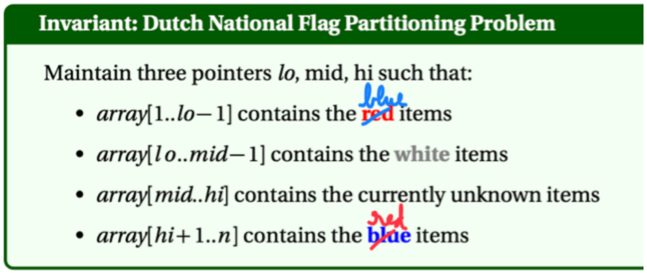
STOP : mid > hi

```
lo, mid, hi = 1, 1, N
LI: arr [1... lo-1] < pivot , arr [lo... mid-1] = pivot , arr [mid...hi] unsorted
                                                          , arr [hi+1...N] > pivot
while mid <= hi:

    if (arr[mid] == red):        > piv
        swap(arr[mid], arr[hi])
        hi -= 1

    elif (arr[mid] == blue):     < piv
        swap(arr[mid], arr[lo])
        lo += 1
        mid += 1
    else:                        = piv
        mid += 1
```





Invariant: Dutch National Flag Partitioning Problem

Maintain three pointers *lo*, mid, hi such that:

- *array*[1..*lo*−1] contains the ~~red~~ blue items
- *array*[*lo*..*mid*−1] contains the white items
- *array*[*mid*..*hi*] contains the currently unknown items
- *array*[*hi*+1..*n*] contains the ~~blue~~ red items

LI: arr [1... lo-1] < piv, arr [lo... mid-1] = piv
arr [mid... hi] unsorted, arr [hi-1...N] > piv

LI: arr [1... lo-1] < piv, arr [lo... mid-1] = piv
arr [mid ... hi] has no elements because mid > hi
arr [hi+1 ... N] > piv

```
def quick_sort(arr[lo... hi]): — T(n)
    if (hi > lo):
        pivot = arr[lo]
        mid = partition(arr[lo... hi, pivot) — O(n)
        quick_sort(arr[lo... mid-1])   } Best: 2T(n/2), if pivot = median
        quick_sort(arr[mid+1... hi])   } Worst: 2T(n-1), if pivot = min or max
```

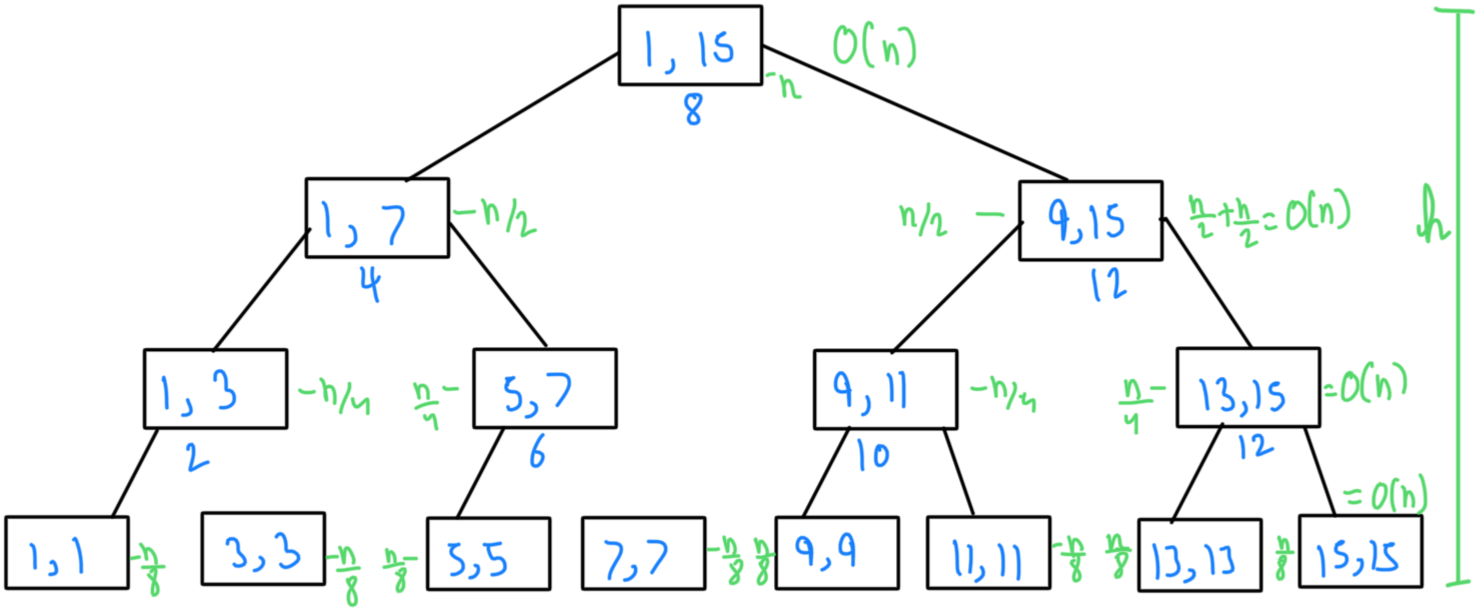
$$T(n) = \begin{cases} 2T(n/2)+n = O(n\log n) & \text{- Best} \\ 2T(n-1)+n = O(n^2) & \text{- Worst} \end{cases}$$

Time:

Best-Case: $O(n \log n)$, when selected pivot is always the median.

[1, 2, 3, - - . . . - - ——— - -, 14, 15]

1 - - - . . - - - 7 8 9 - . . . — 15



At each level O(n) work is done by partition algorithm.

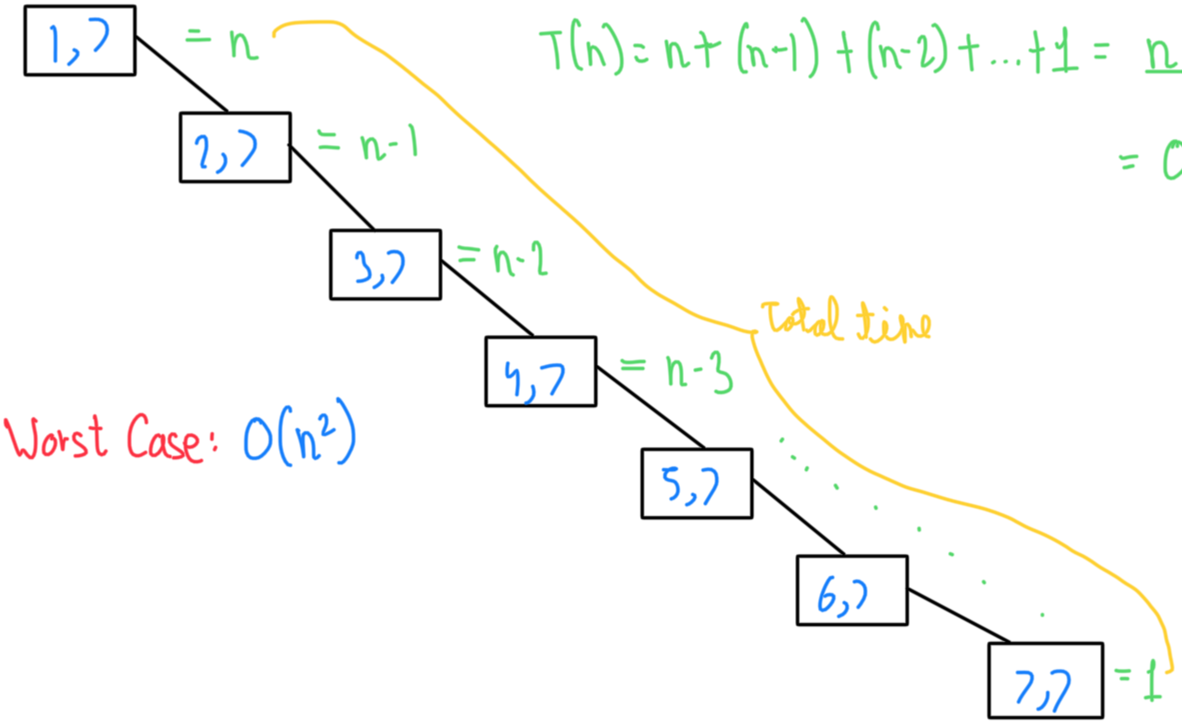
There are total of $\log_2(n)$ levels because the size of lis is continuosly halving until it reaches 1 and it took $\log_2(n)$ halving to get to size 1 for each sub-list:

$$n/2/2/2/\ldots/2 = 1 \rightarrow n/2^k = 1 \rightarrow 2^k = n \rightarrow h = \log_2(n)$$

Quick sort best-case time : tot.levels · partition = $\log_2(n) \cdot n$
: $O(n\log n)$

Worst-Case: $O(n^2)$, when pivot is either min. OR max. element.

[2, 4, 8, 10, 16, 18, 17]
1 2 3 4 5 6 7



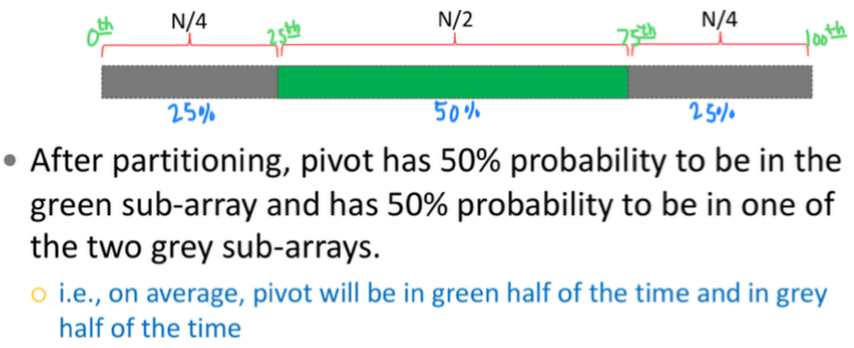
$$T(n) = n + (n-1) + (n-2) + \dots + 1 = \frac{n(n+1)}{2}$$
$$= O(n^2)$$

Worst Case: $O(n^2)$

Average Case: $O(n \log n)$

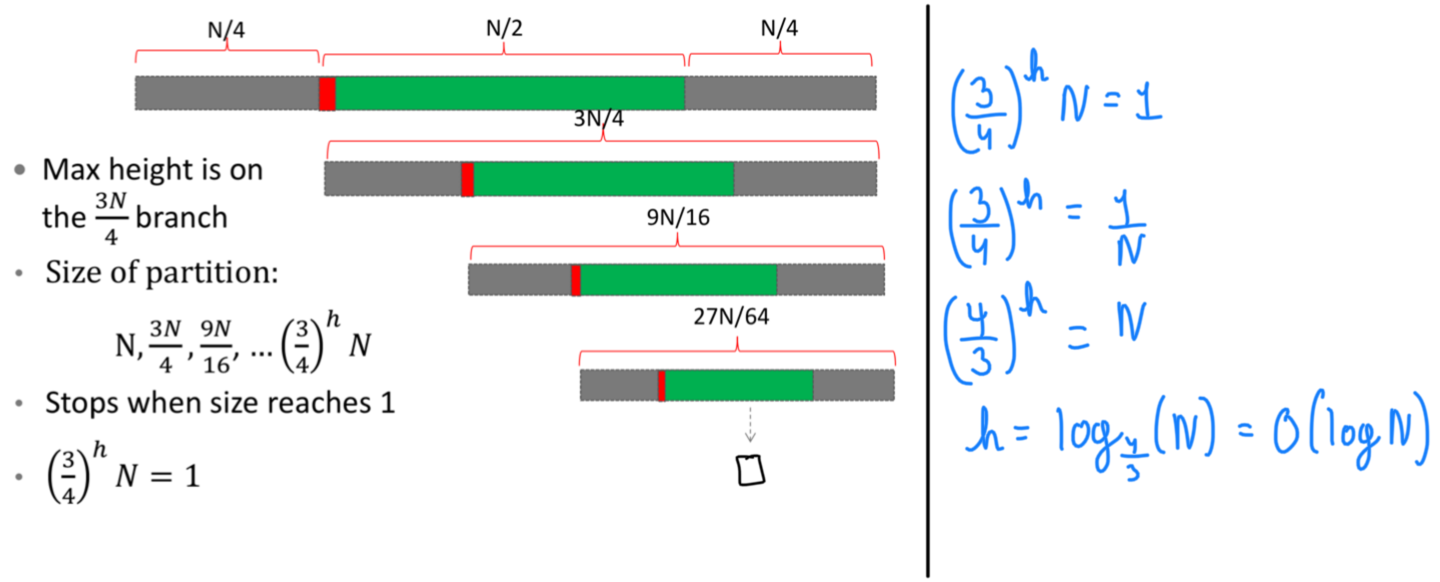
Proof: Using a coin-flip arguement

Consider an arbitrary array of *n* elements. Ideally, we would like to select a pivot that partitions the array fairly evenly. Let's define a "good pivot" to be one that lies in the middle 50% of the array, i.e. a good pivot is one such that at least 25% of the array is less than it, and 25% of the array is greater than it.



- After partitioning, pivot has 50% probability to be in the green sub-array and has 50% probability to be in one of the two grey sub-arrays.
    - i.e., on average, pivot will be in green half of the time and in grey half of the time

Suppose we are lucky and we select a good pivot every single time. The worst thing that can happen is that the pivot might be on the edge of the good range, i.e. we will select a pivot right on the 25th or 75th percentile. In this case, our recursive calls will have arrays of size 0.25*n* and 0.75*n* to deal with. The longest branch of the recursion tree will therefore consist of the later calls, of sizes 0.75n



- Max height is on the $\frac{3N}{4}$ branch
- Size of partition:

$$N, \frac{3N}{4}, \frac{9N}{16}, \dots \left(\frac{3}{4}\right)^h N$$

- Stops when size reaches 1
- $\left(\frac{3}{4}\right)^h N = 1$

$$\left(\frac{3}{4}\right)^h N = 1$$

$$\left(\frac{3}{4}\right)^h = \frac{1}{N}$$

$$\left(\frac{4}{3}\right)^h = N$$

$$h = \log_{\frac{4}{3}}(N) = O(\log N)$$

At each level of recursion we perform *O* (*n*) work for the partitioning, and hence the time taken will be *O*(*n* log(*n*)).

We know that we can not expect to select a good pivot every time, but since the good pivots make up 50% of the array, we have a 50% probability of selecting a good pivot. Therefore we should expect that on average, every second pivot that we select will be good.

In other words, we expect to need two flips of an unbiased coin before seeing heads. This means that even if every other pivot is bad and barely improves the sub-problem size, we expect that after 2 × log[4/3](*N*) levels of recursion to hit the base case.

This means that the expected amount of work to Quicksort a random array is just twice the amount

of work required in the best case, which was $O(n \log(n))$.
Double of $O(n \log(n))$ is still $O(n \log(n))$, hence from this we can conclude that the **average case complexity of Quicksort is $O(n \log(n))$**.

Space: partition aux + call stack

Using the other definition, if we use one of the in-place parti- tioning schemes (Hoare or DNF) then Quicksort will also be in-place, but not stable. If we use a stable partitioning scheme (such as the naive partitioning scheme), then Quicksort will be stable, but not in-place.

Best-Case: $O(\log n)$, balanced partition, so $(\log n)$ recursive calls.

Average-Case: $O(\log n)$

Worst-Case: $O(n)$, unbalanced partition, so (n) recursive calls.

Since, Quick sort does not have $O(1)$ aux space, it is best to choose a stable partioning algorithm and not worry about in-place partitioning.