CS2040S Tutorial 3

Problem Set 3

PS3

Sorter D and E are unstable → Quick Sort or Selection Sort

Invalid Justification: Sorter E takes 5s and Sorter D takes 8s -> Sorter D is SelectionSort since SelectionSort is O(n^2) which is slower than QuickSort's O(nlogn)

- Big-O has hidden coefficients: note that c*n^2 could be smaller than d*nlogn

Asymptotic analysis is about how time varies with **input size**

PS3

Sorter D and E are unstable → Quick Sort or Selection Sort

Valid Justification: Sorter D takes 5s on array length 10000 and 20s on array length 20000; when n increases by 2, time increases by 4, so the asymptotic bound is O(n^2)

- When you take the ratio over different input sizes, you ‘cancel’ out the coefficient

Recap

Quick Sort

Following slides credit to Christian

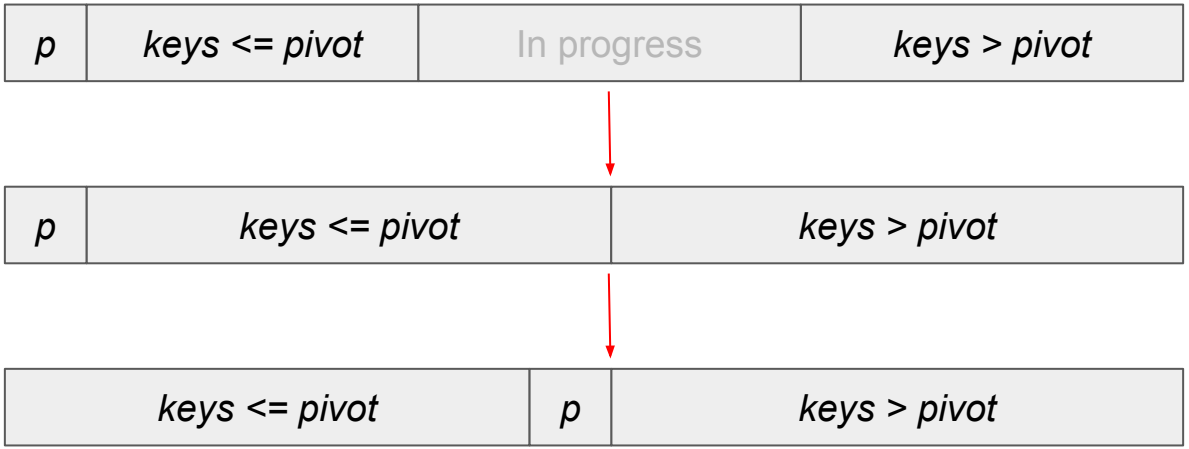
Quick Sort

The idea:

- Choose a pivot (and hope it’s not a bad one)
- Partition the array into regions of smaller and larger than the pivot
- Recursively sort the regions

Hoare’s Partition

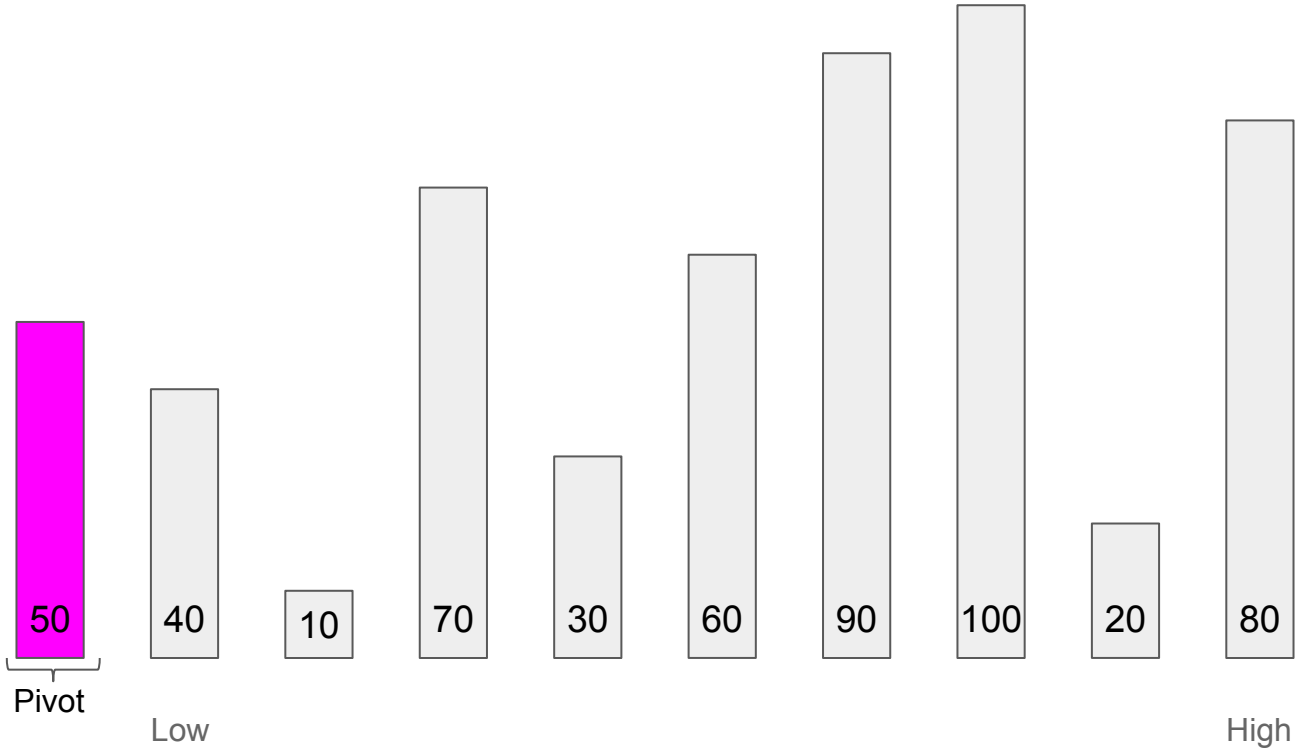
The idea:



Hoare's Partition

Initialisation

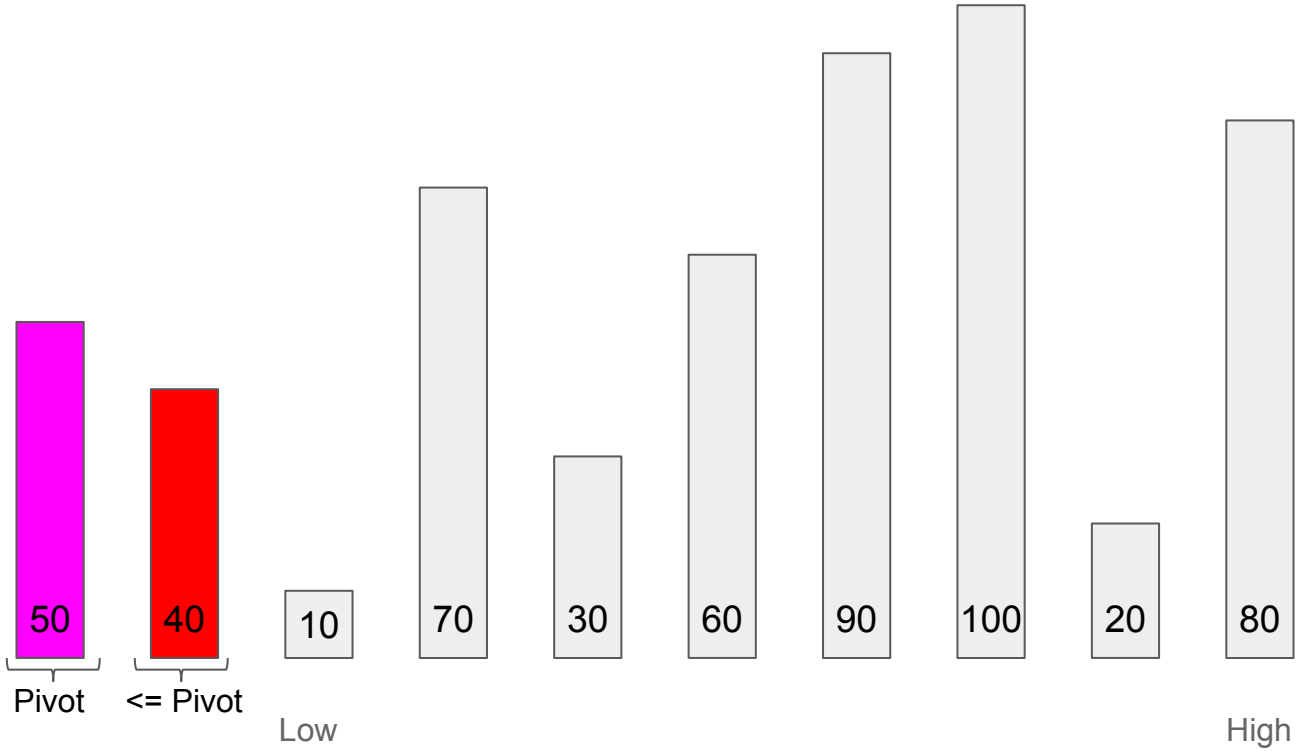
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

40 is less than pivot!

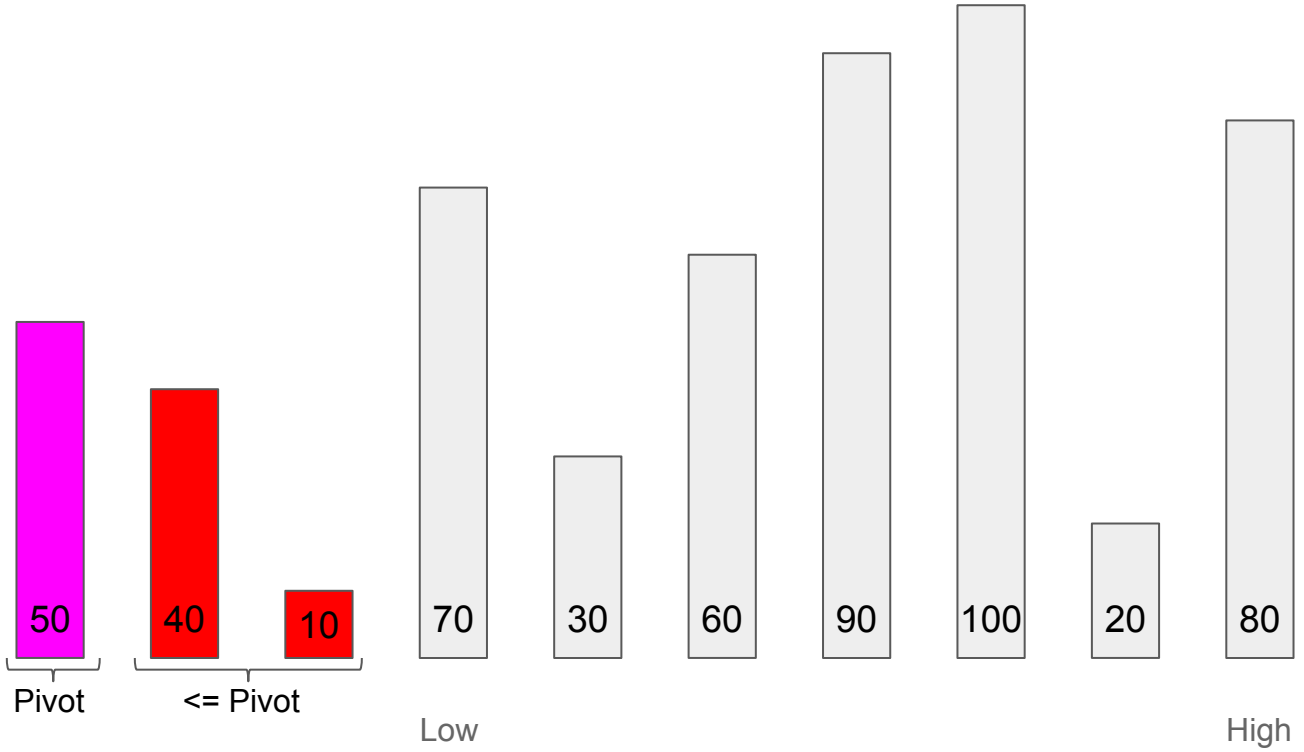
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

10 is less than pivot!

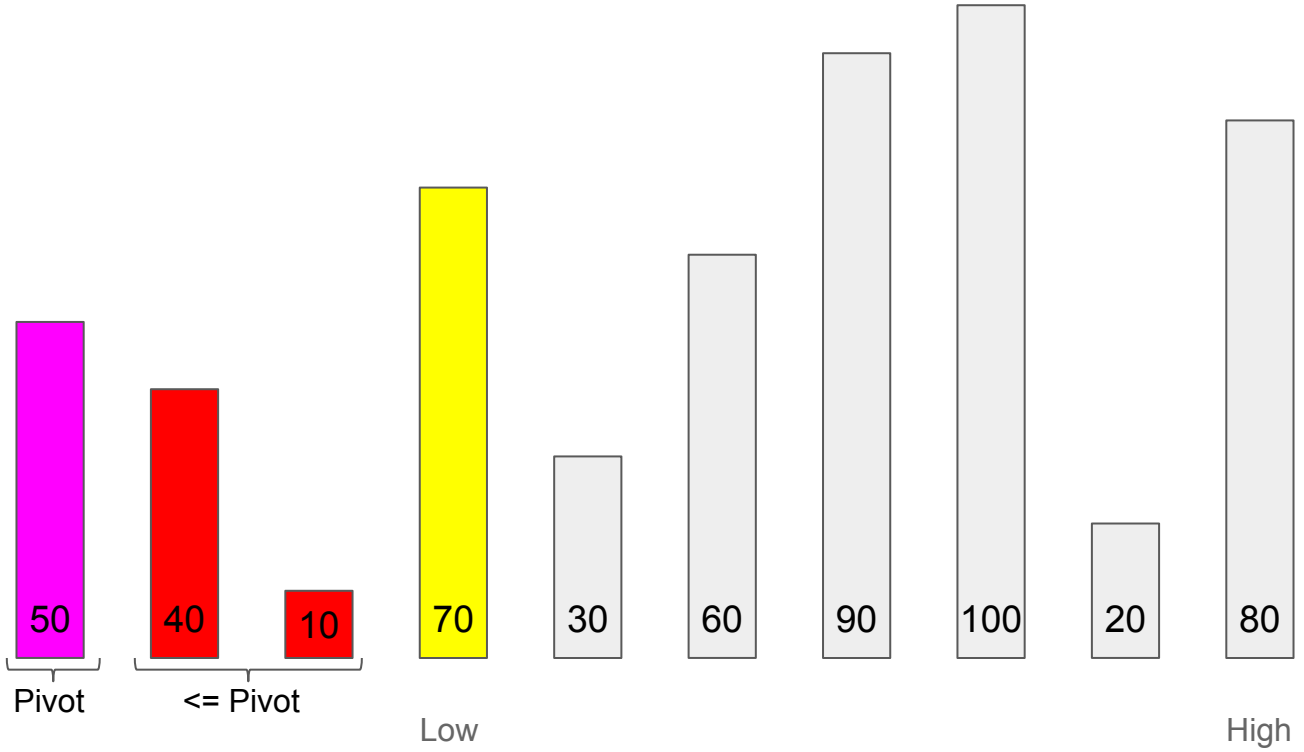
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

Uh oh! 70 is not less than equal to the pivot. So we stop "increasing" up from 'low', and we start decreasing from 'high' after this

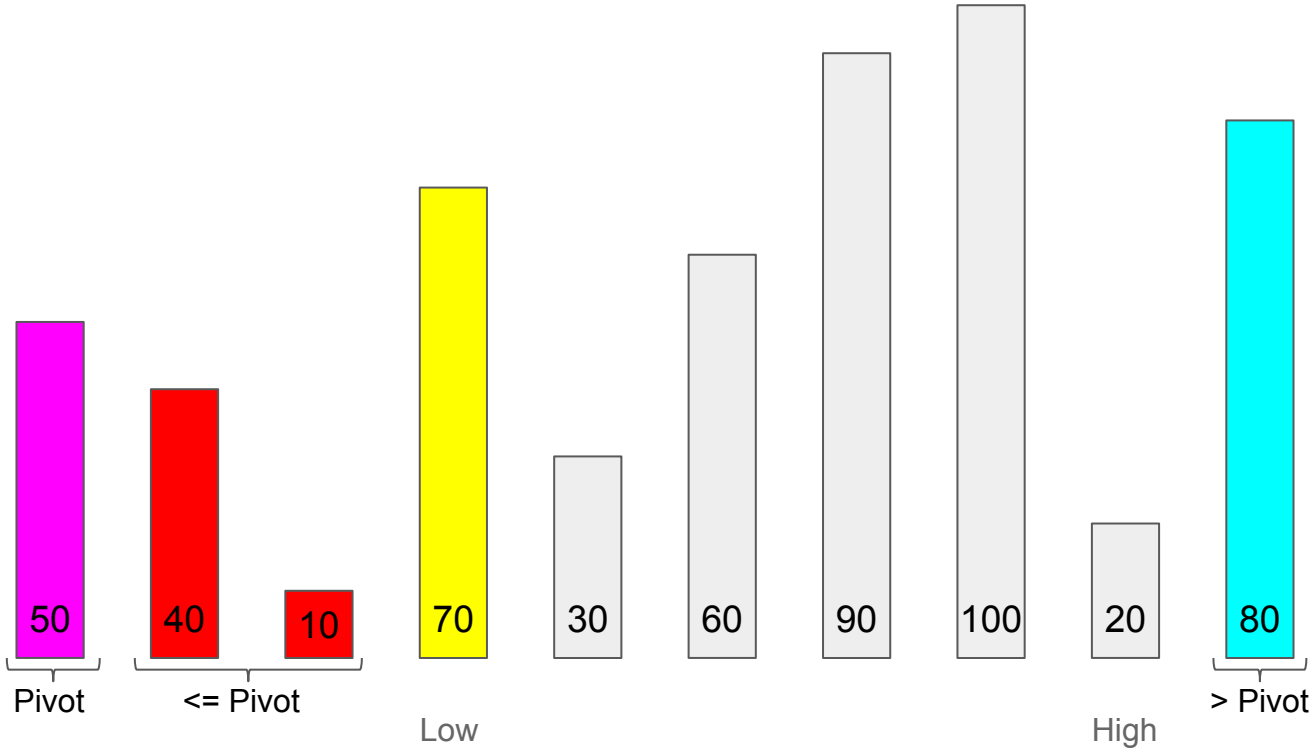
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

80 is greater than the pivot!

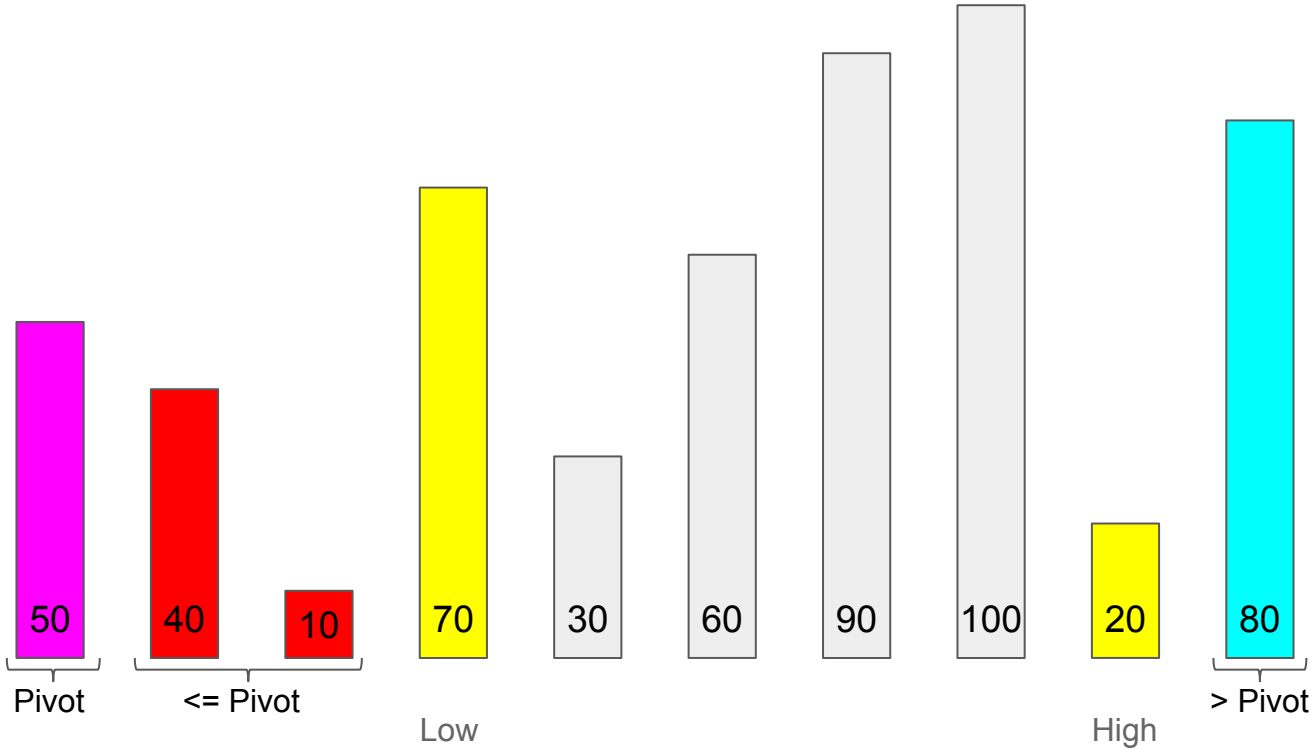
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

20 is less than the pivot! Now we have the two candidates to swap!

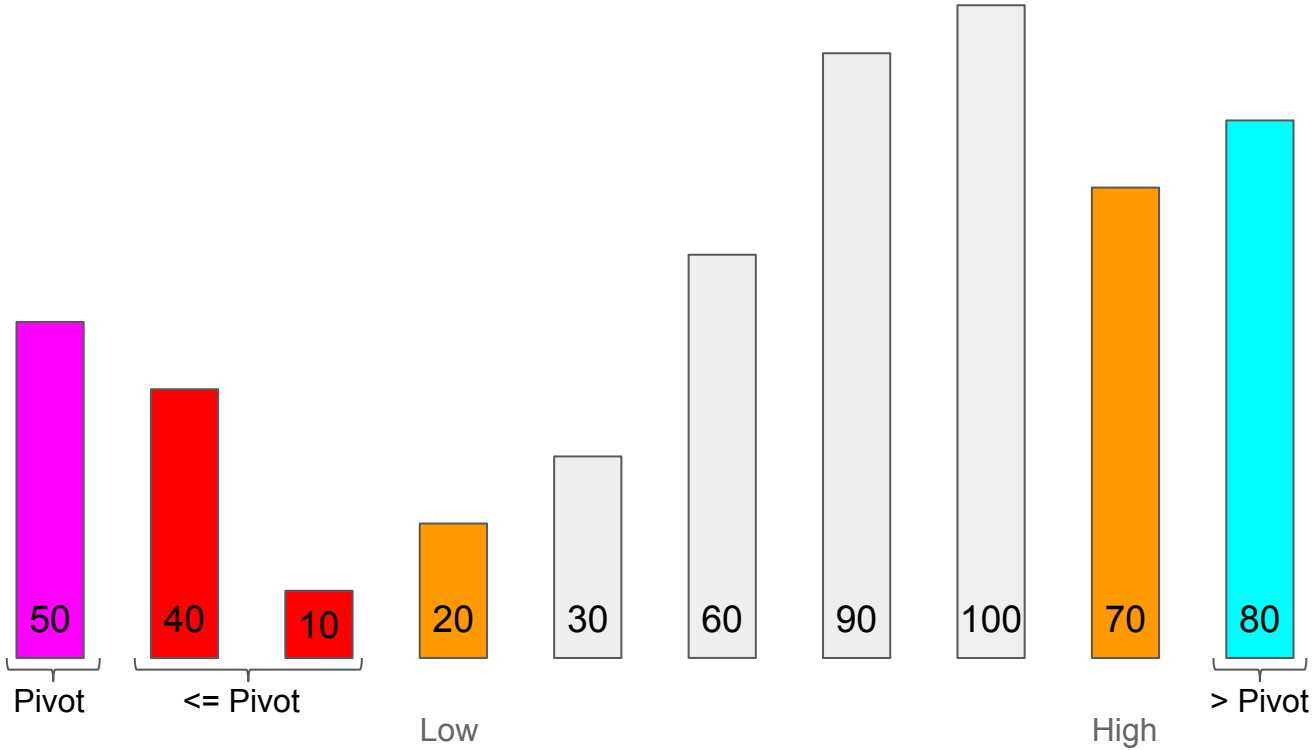
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

So we shall swap and...

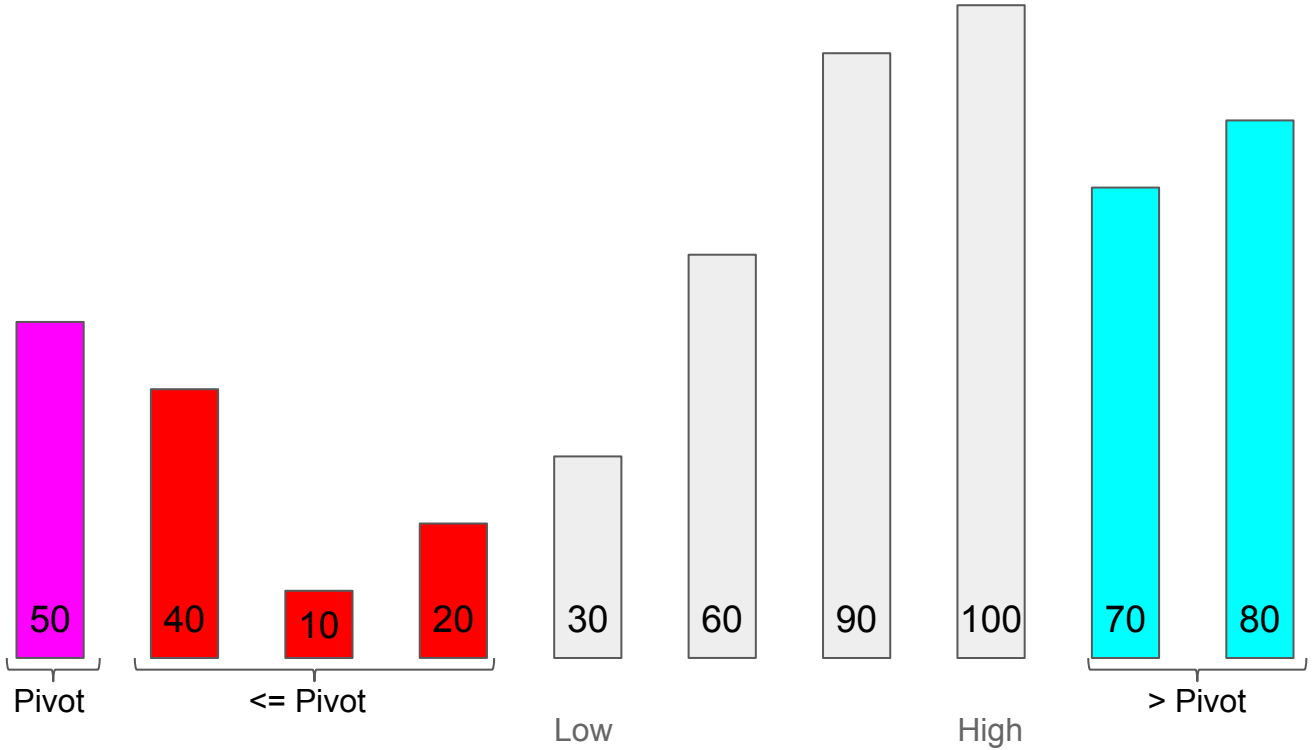
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

The two elements we just swapped now belong to the correct partition! Let's continue!

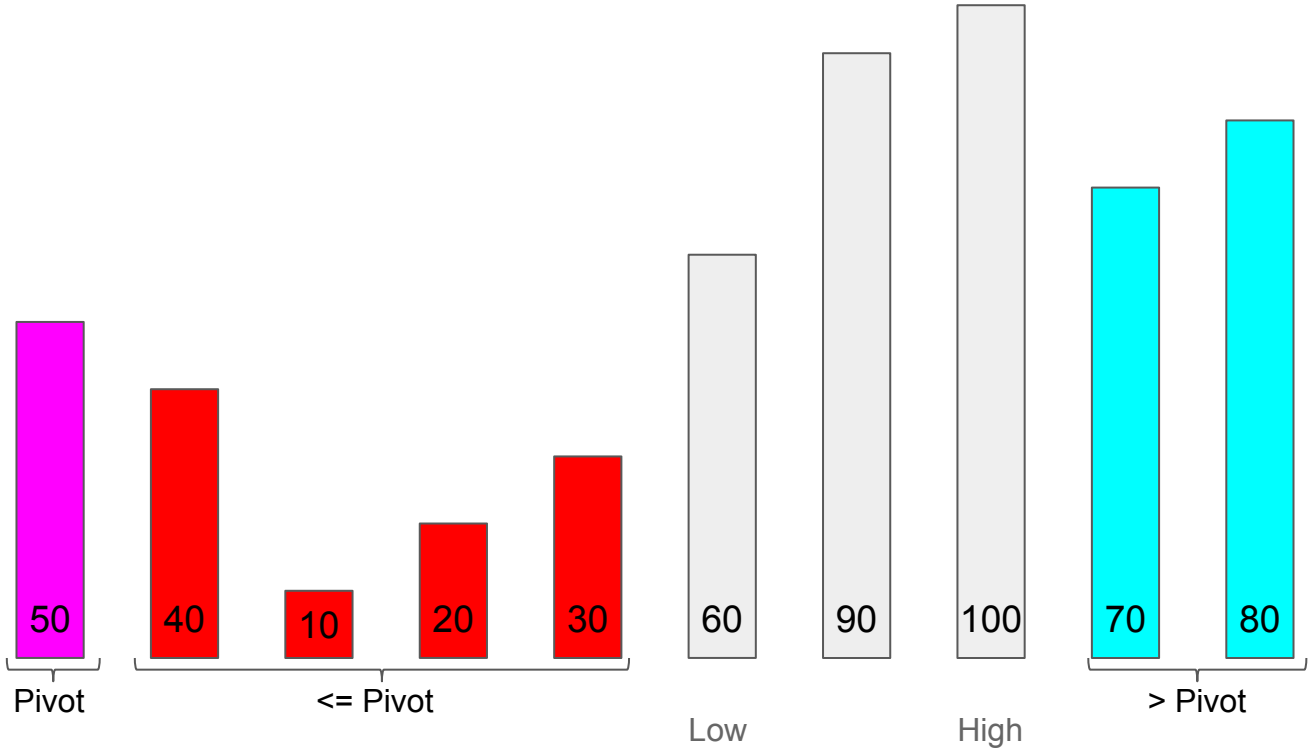
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

30 is less than the pivot

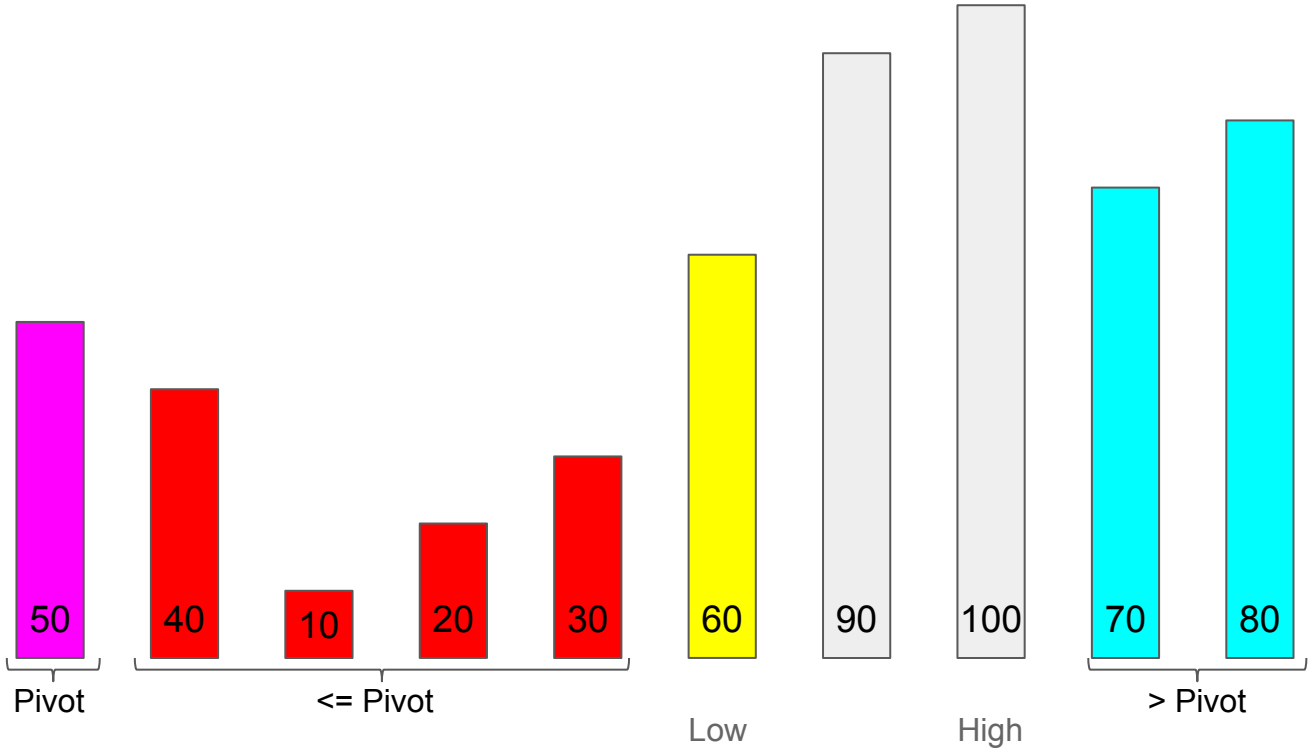
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

60 is greater than the pivot! So again, we stop incrementing the 'low'

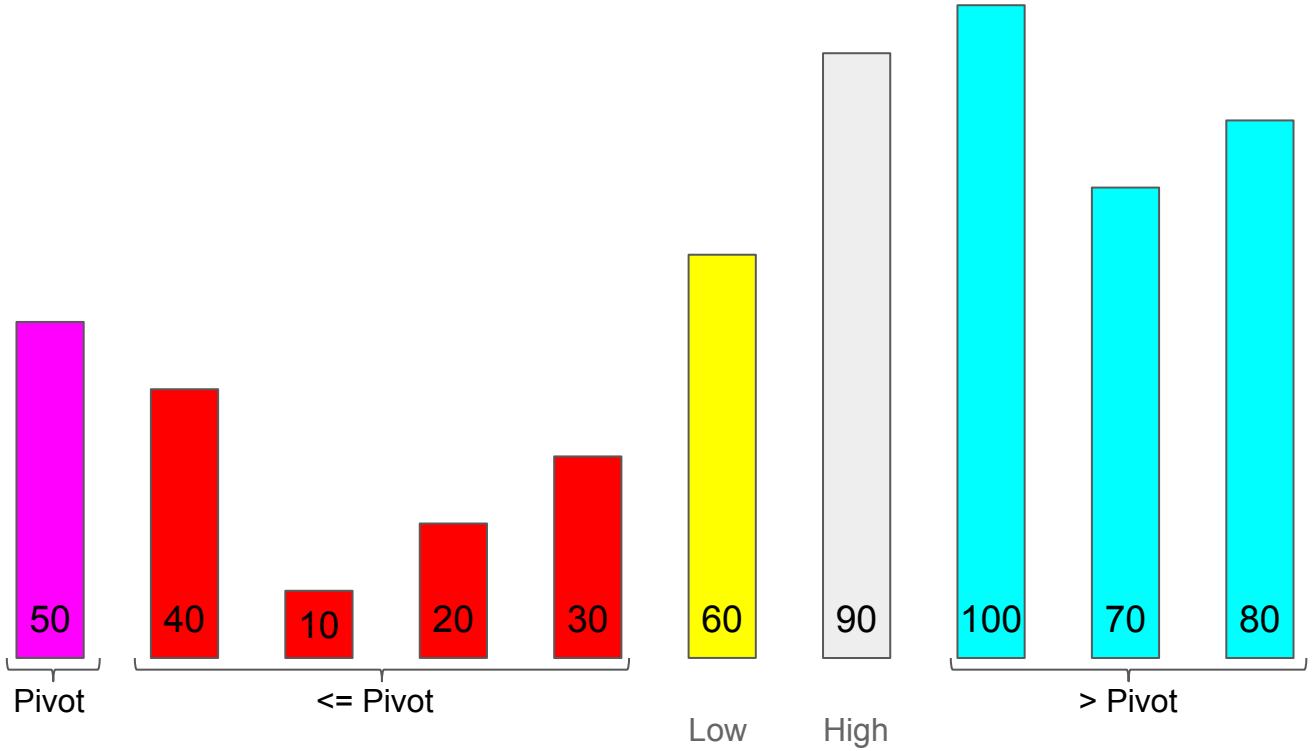
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

100 is greater than the pivot

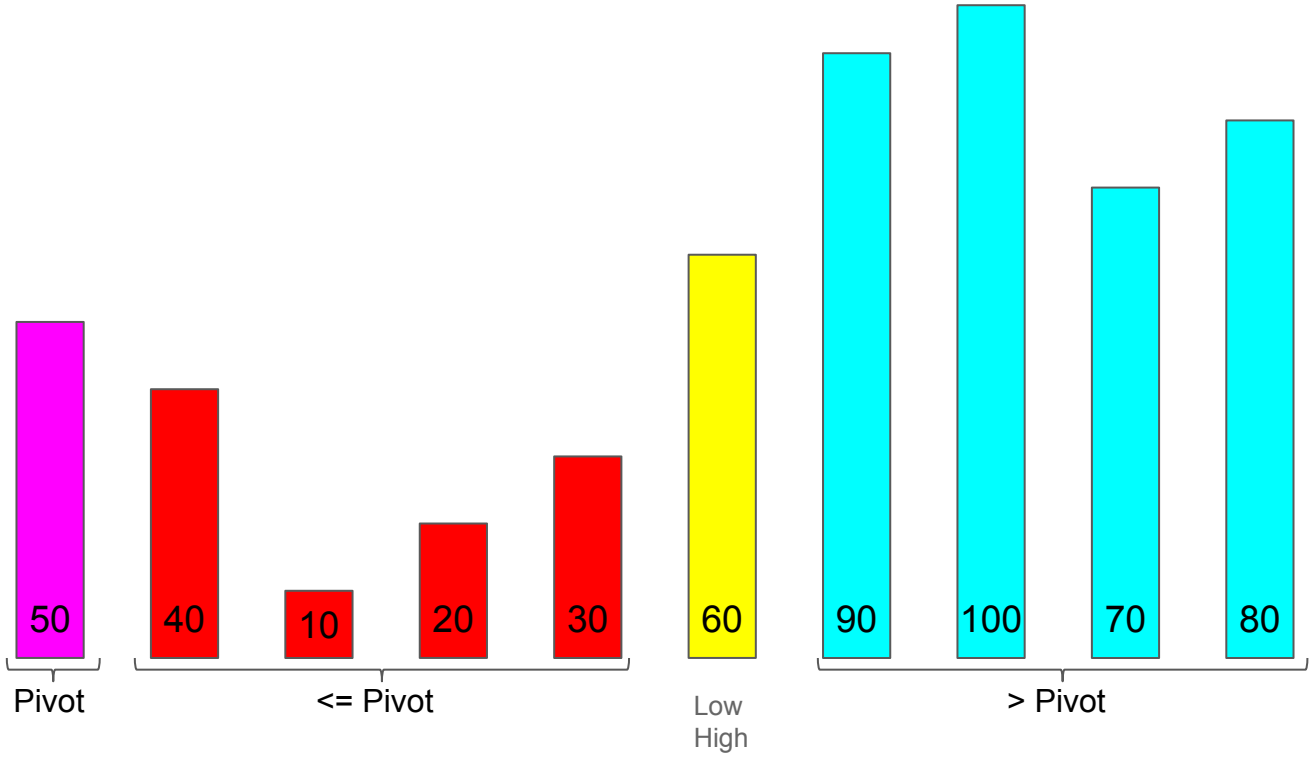
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

90 is greater than the pivot

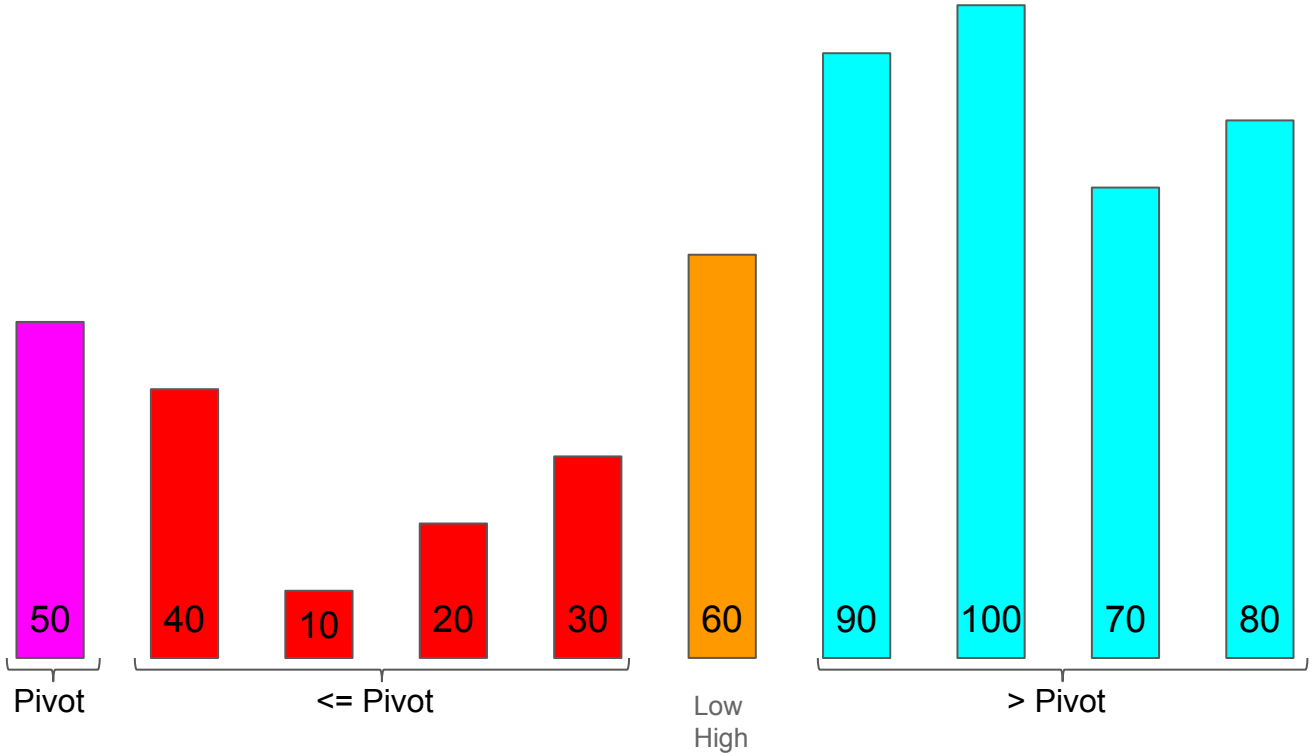
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

The 'low' and 'high' has met! So we can stop doing the iterations

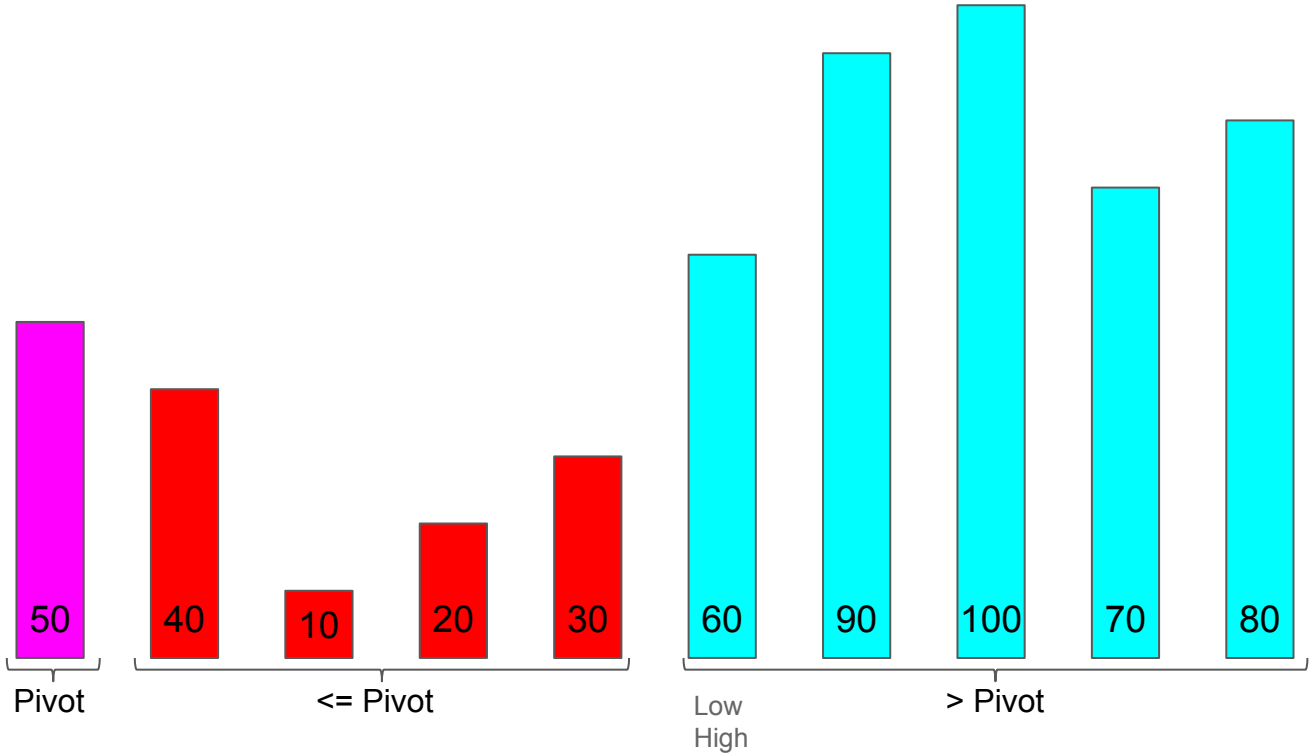
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

As it turns out, 60 belonged to the > pivot partition

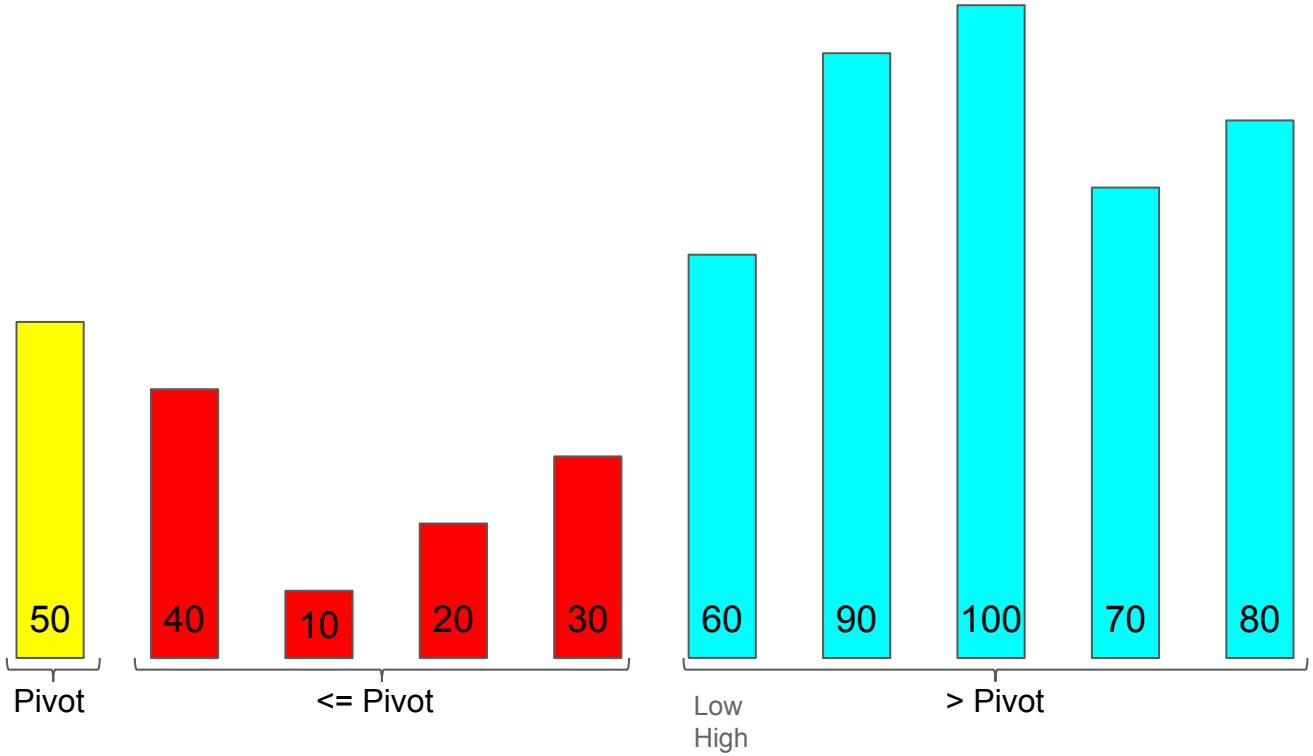
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

Remember: our goal is to have the partition in the center! Which element should I swap with?

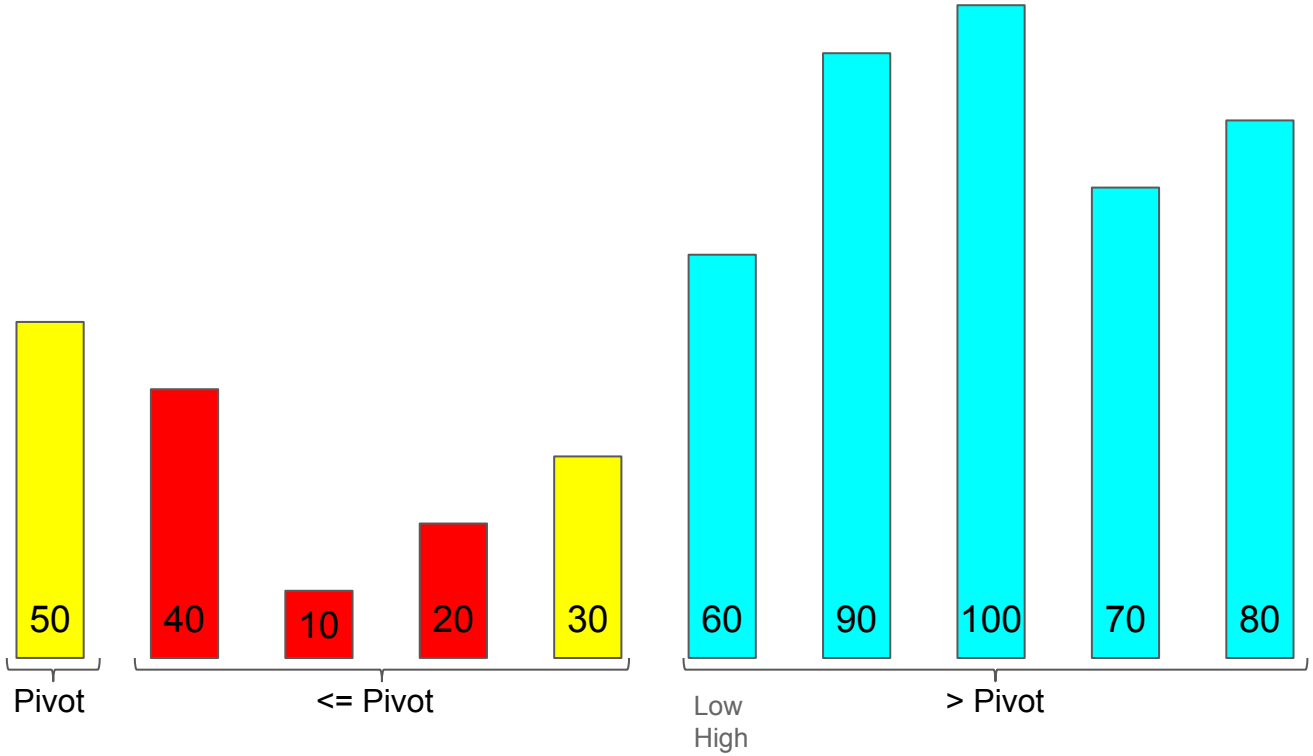
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

30! Or in particular: one index less than 'low' or 'high'

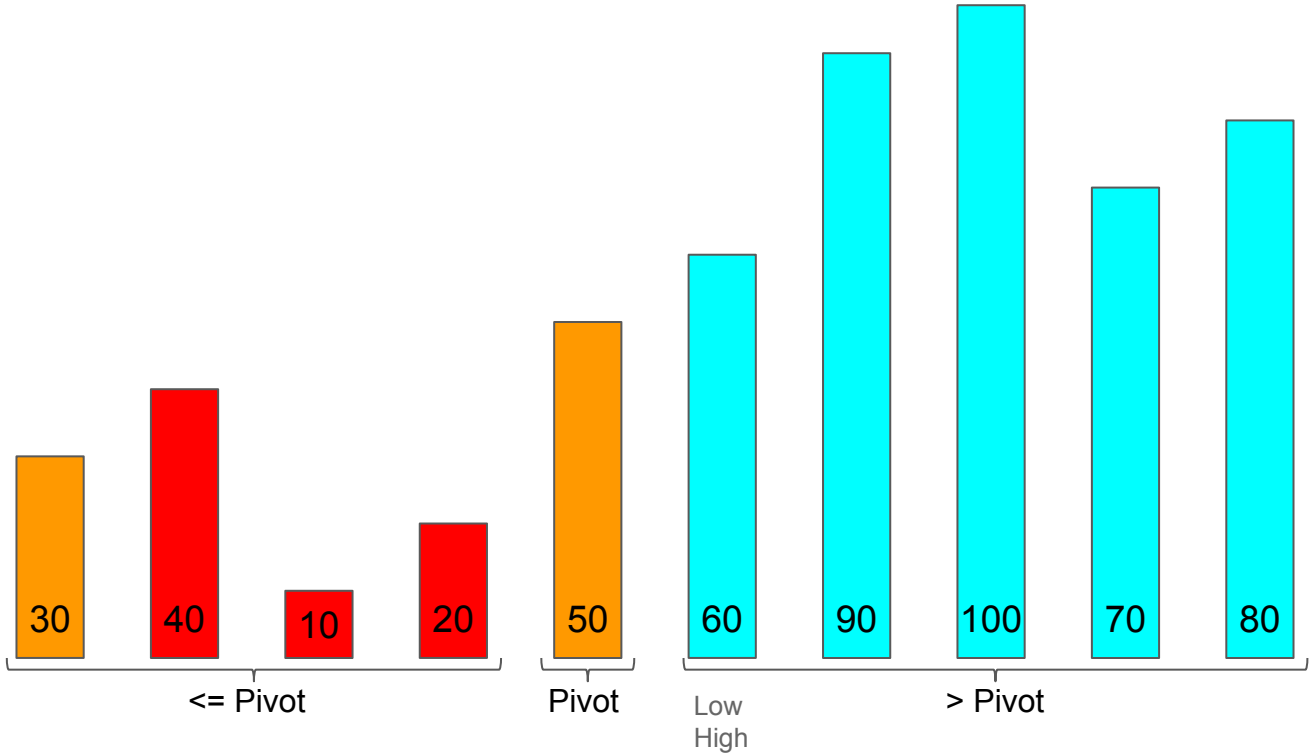
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

And swap! (This is to satisfy the postconditions)

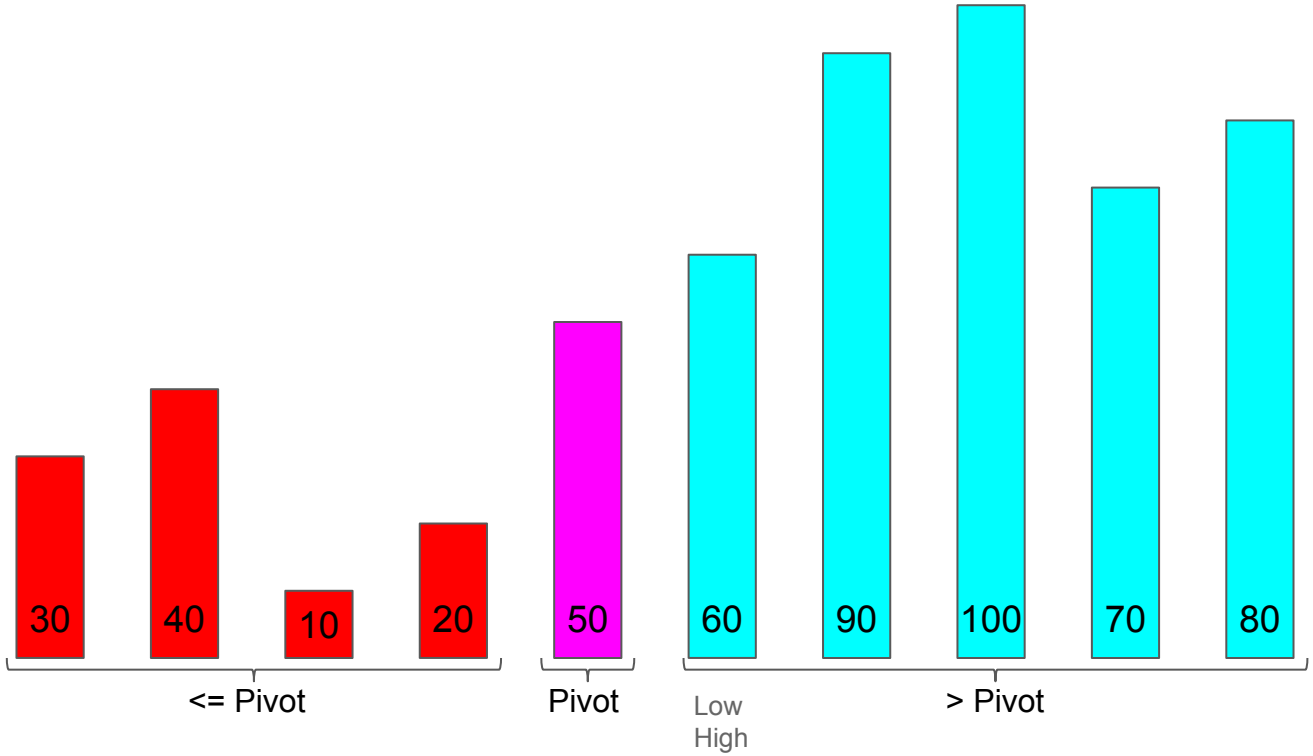
Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Hoare's Partition

Now we have successfully partitioned the array :D

Legend
Pivot
<= Pivot
> Pivot
About to swap
Post-swapping



Analysis of Quicksort

(Refer to lecture slides for more details)

If we randomly choose the pivot:

- Average Case: $O(nlogn)$ time
- Worst-Case: $O(n^2)$ time

Space: $O(1)$ space because only swappings are required

Randomization

- Why?

- Guarding against worst case
- Better probability of success

Tutorial Time

Qn 1a

Suppose that the pivot choice is the median of the first, middle and last keys, can you find a bad input for QuickSort?

Qn 1 Answer

- Yes! As long as we have a fixed pivot choice, the time complexity would remain at O(n^2) as it is always possible to find a bad input for the algorithm.

Qn 1 Answer

- Yes! As long as we have a fixed pivot choice, the time complexity would remain at O(n^2) as it is always possible to find a bad input for the algorithm.
  - Construct an array where median of the first, middle, last keys is always the 2nd largest/smallest element.
  - $T(n) = T(n - 2) + O(n) \rightarrow$ Total time complexity is $O(n^2)$

Qn 1 Answer

- For example (the underlined section indicates the subarray that is currently being recursed on, bolded are the first, middle and last keys):

1st Partitioning : [**8**, 3, 2, 1, **5**, 4, 6, 7, **9**] (8 will be selected as the pivot)

2nd Partitioning : [**7**, 3, 2, **1**, 5, 4, **6**, 8, 9] (6 will be selected as the pivot)

3rd Partitioning : [**4**, 3, **2**, 1, **5**, 6, 7, 8, 9] (4 will be selected as the pivot)

4th Partitioning : [**1**, **3**, **2**, 4, 5, 6, 7, 8, 9] (2 will be selected as the pivot)

Qn 1b

Are any of the partitioning algorithms we have seen for QuickSort stable? Can you design a stable partitioning algorithm? Would it be efficient?

Qn 1b Answer

- No, all partitioning algorithms are not stable.(at least the ones you have seen)

- But we can make them stable! How?

Qn 1b Answer

- Create a new array with the original indices of the key

Original Array : [1, **2**, 5, 3, 5, 3, 8, 7, **2**]

New Array : [0, **1**, 2, 3, 4, 5, 6, 7, **8**]

When comparing elements, the new array would be used to disambiguate elements with equal keys, creating a “total ordering” between every key.

- When comparing the 2s, sorting algorithm will check which value came first using the new array of original indices

Qn 1b Answer

*** Note: The partitioning algorithm will no longer be in place and require an extra O(n) space

Qn 1c

Consider a QuickSort implementation that uses the 3-way partitioning scheme (i.e. elements equal to the pivot are partitioned into their own segment).

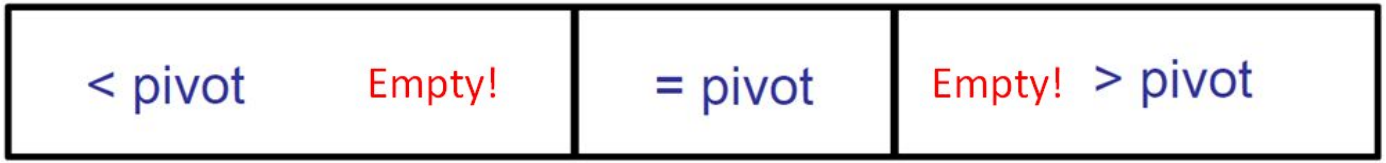
Qn 1c

Consider a QuickSort implementation that uses the 3-way partitioning scheme (i.e. elements equal to the pivot are partitioned into their own segment).

i) If an input array of size n contains all identical keys, what is the asymptotic bound for QuickSort?

Qn 1c (i) Answer

- O(n)
- After first pass, no more unsorted elements



Qn 1c

Consider a QuickSort implementation that uses the 3-way partitioning scheme (i.e. elements equal to the pivot are partitioned into their own segment).

(ii) If an input array of size n contains $k < n$ distinct keys, what is the asymptotic bound for QuickSort?

Qn 1c (ii) Answer

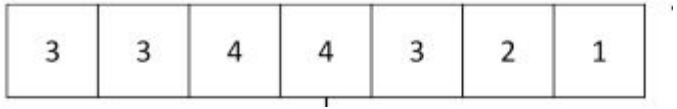
- Each branch in the recursion tree of QuickSort is the result of 1 partition
- Each partition selects 1 pivot
- Since we are using 3-way partitioning, we can have up to k pivots as there are k distinct keys
- Thus, there are $O(k)$ branches in our recursion tree!
- In the worst case, the height of our recursion tree is also $O(k)$
    - When the tree is unbalanced

Qn 1c (ii) Answer

- Now that we have established an upper bound for the height of our recursion tree, we need to figure out the amount of work done at each level
- Partitioning an array takes $O(n)$ time
- As such, the total amount of work done at each level of the recursion tree is $O(n)$.
- Thus, we arrive at an asymptotic bound of $O(nk)$ by multiplying the height of the tree by the amount of work done at each level!

Qn 1c (ii) Answer

Example:

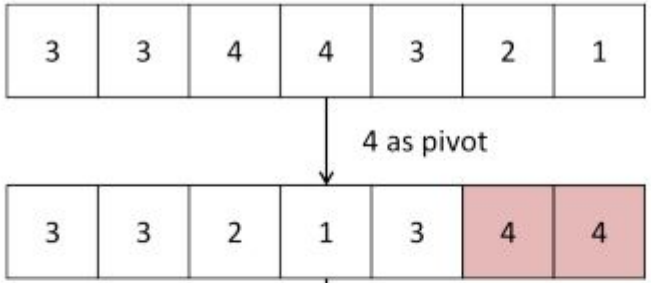


k = 4

Qn 1c (ii) Answer

Example:

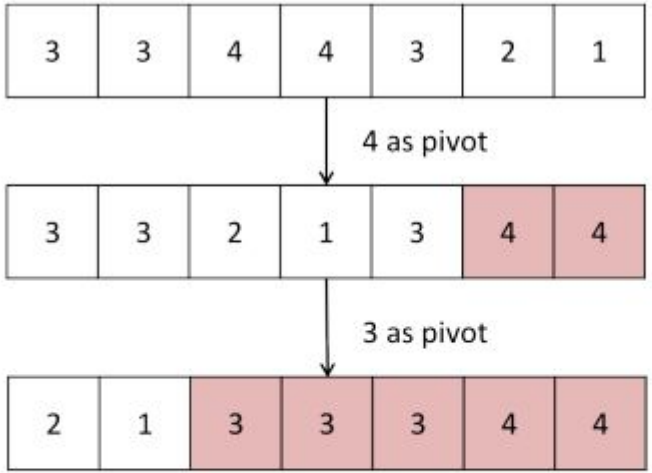
k = 4



Qn 1c (ii) Answer

Example:

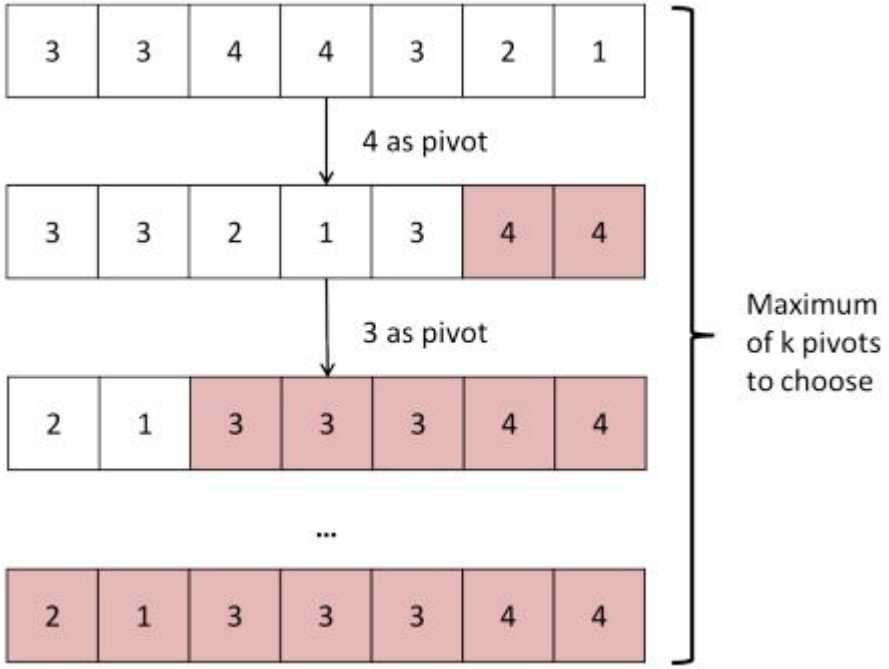
k = 4



Qn 1c (ii) Answer

Example:

k = 4



Qn 1c (ii) Answer

What if our pivot selection is guaranteed to be balanced?

- Then, we know that the height of our recursion tree is bounded by $O(\log k)$, giving us an overall time complexity of $O(n \log k)$ instead

Qn 2a

Given an array A, decide if there are any duplicated elements in the array.

Qn 2a Answer

Given an array A, decide if there are any duplicated elements in the array.

- Sort → O(n log n)
- Traverse array, check if i is same as i + 1 → O(n)

Overall: O(n log n)

** Possible to do in O(n) with hashing which will be covered later in this module

Qn 2b

Given an array A, output another array B with all the duplicates removed. Note the order of the elements in B does not need to follow the same order in A. That means if array A is {3, 2, 1, 3, 2, 1}, then your algorithm can output {1, 2, 3}.

Qn 2b Answer

Given an array A, output another array B with all the duplicates removed. Note the order of the elements in B does not need to follow the same order in A. That means if array A is {3, 2, 1, 3, 2, 1}, then your algorithm can output {1, 2, 3}.

- Similar to 2a
- Sort → O(n log n)
- Variable k to keep track of largest element encountered → O(n)
  - Remove if element at index i is identical to k
  - Update the value of k if not duplicate

Overall: O(n log n)

** Also possible to do in O(n) with hashing

Qn 2c

Given arrays A and B, output a new array C containing all the distinct items in both A and B. You are given that array A and array B already have their duplicates removed.

Qn 2c Answer

Given arrays A and B, output a new array C containing all the distinct items in both A and B. You are given that array A and array B already have their duplicates removed.

- Merge sort!
- Sort both arrays in ascending order and do the merge step of merge sort
- If element has already been added to array C, discard it

Qn 2c Answer

- Sort both arrays → O(n log n)
- Merge both arrays → O(n)

$$
\begin{aligned}
& n_A \log n_A + n_B \log n_B = \log n_A^{n_A} + \log n_B^{n_B} \\
& = \log(n_A^{n_A} \times n_B^{n_B}) \leq \log(\max\{n_A, n_B\}^{n_A+n_B}) \\
& \leq \log\left((n_A + n_B)^{n_A+n_B}\right) = \log n^n = n \log n
\end{aligned}
$$

Overall: O(n log n)

** Also possible (again) to do in O(n) with hashing

Qn 2d

Given array A and a target value, output two elements x and y in A where (x+y) equals the target value.

Qn 2d Answer

- Sort (again!)
- Use two pointers, low and high
- If the target is less than the required value, low++
- If the target is greater than the required value, high--

Overall: O(n log n)

Qn 3

Quicksort is pretty fast. But that was with one pivot.

Can we improve it by using two pivots?

What about k pivots? What would the asymptotic running time be?

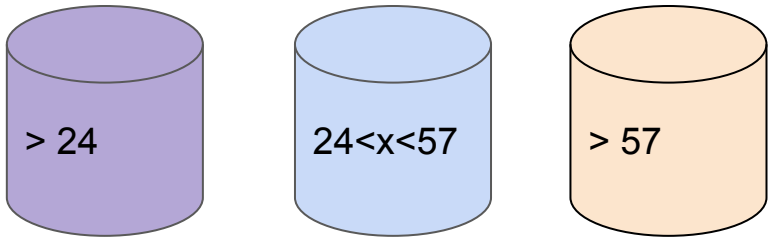
Qn 3 Answer

- Sort the pivots?
- Placing the item into the correct basket?

Example with 2 pivots:

Before partition: [**57** 8 42 75 29 77 38 **24**]

After partition: [8 **24** 42 38 29 **57** 75 77]



Qn 3 Answer

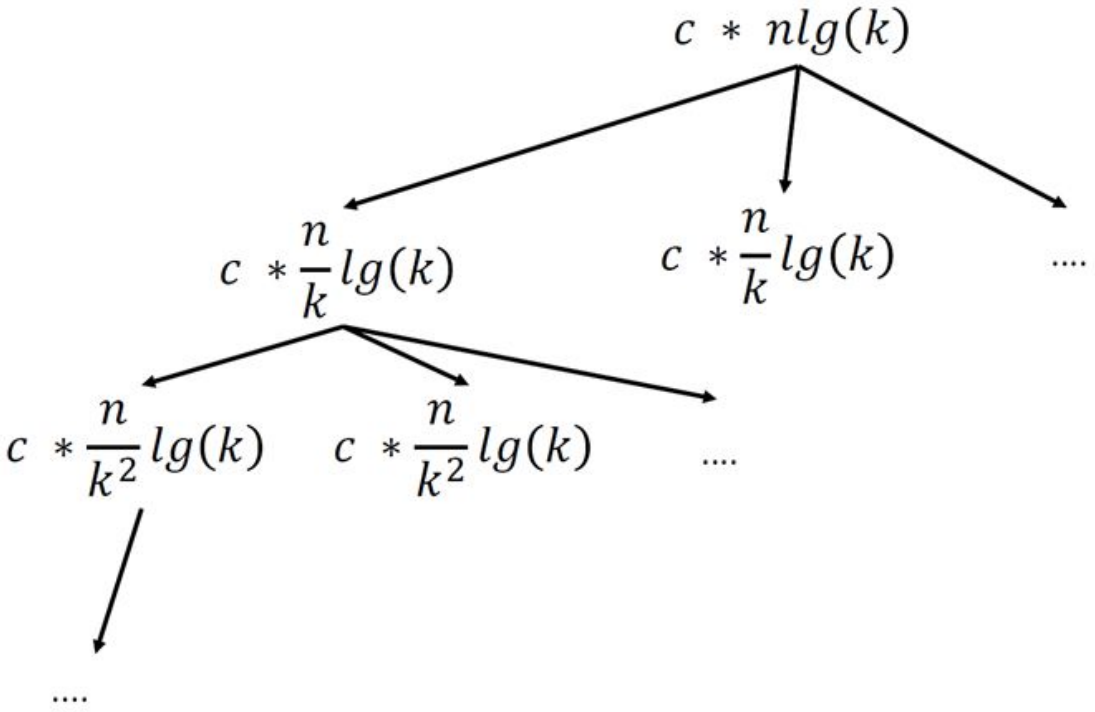
Analysis of one partition operation

- Sort k pivots – $O(k \log k)$
- Place each element in the right bucket – $O(n \log k)$
  - Binary search k pivots to find the correct bucket – $O(\log k)$
  - Do it $n-k$ times for each element – $O(n)$

In each partition operation, $O(n \log k)$ is the dominant term because $n > k$

Resulting recurrence relation: $T(n) = kT\left(\frac{n}{k}\right) + O(n \log k)$

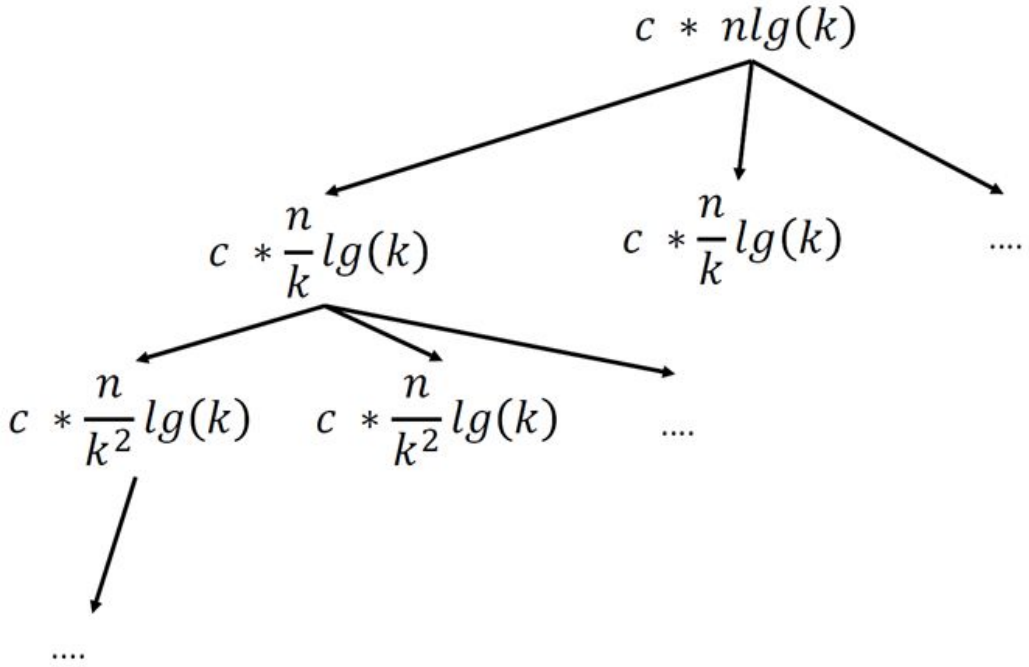
Qn 3 Answer



** c is some constant

$$T(n) = kT\left(\frac{n}{k}\right) + O(n \log k)$$

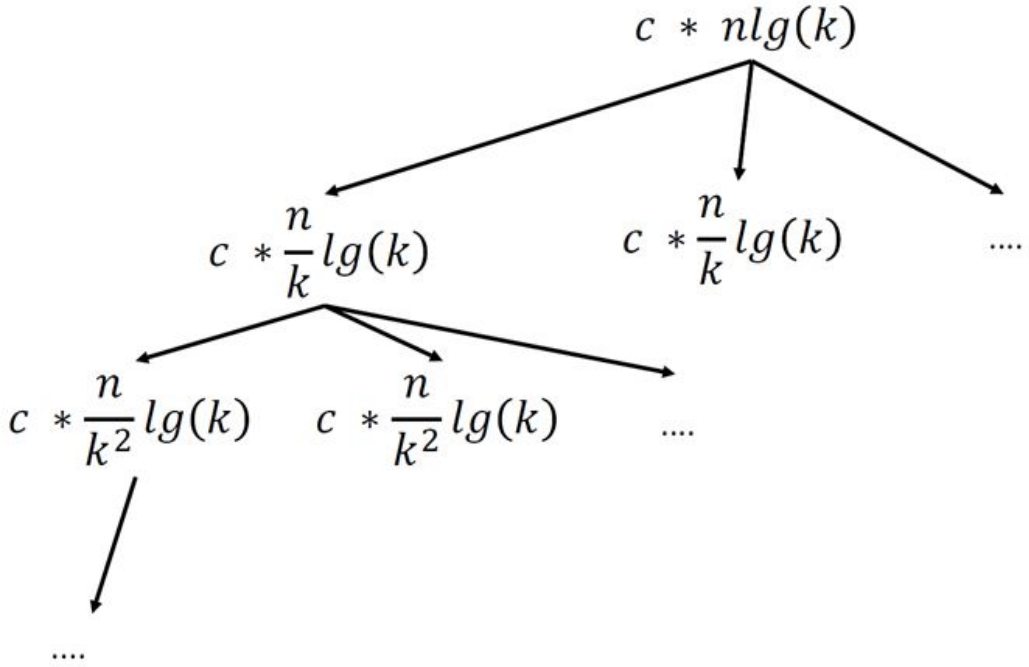
Qn 3 Answer



- We can see that the height of the recursion tree is $\log_k n$
- At each level of the tree, we have k^{depth} nodes, with each node doing $c(n / k^{depth}) \log k$ amount of work
- Hence total amount of work done at each level is

  $c(n/k^{depth}) \log k \times k^{depth} = cn \log k$

Qn 3 Answer

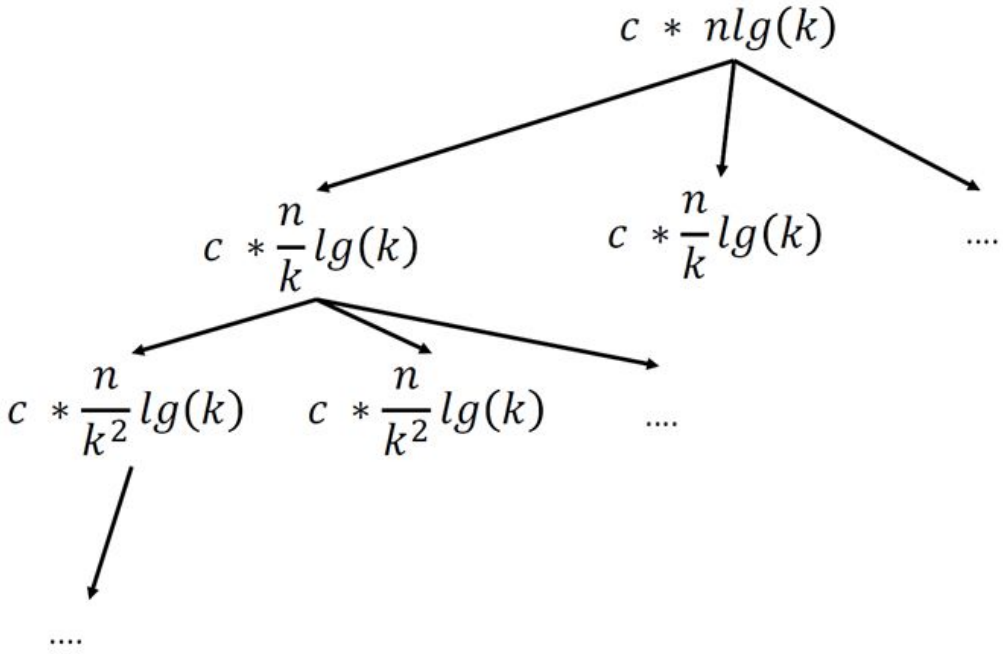


- By multiplying the work done at each level by the height of the tree, we get

$$
\begin{aligned}
& cn \log k \times \log_k n \\
&= cn \left(\frac{\log n}{\log k} \right) \log k \\
&= cn \log n \\
&= O(n \log n)
\end{aligned}
$$

Qn 3 Answer

$$T(n) = kT\left(\frac{n}{k}\right) + O(n \log k)$$



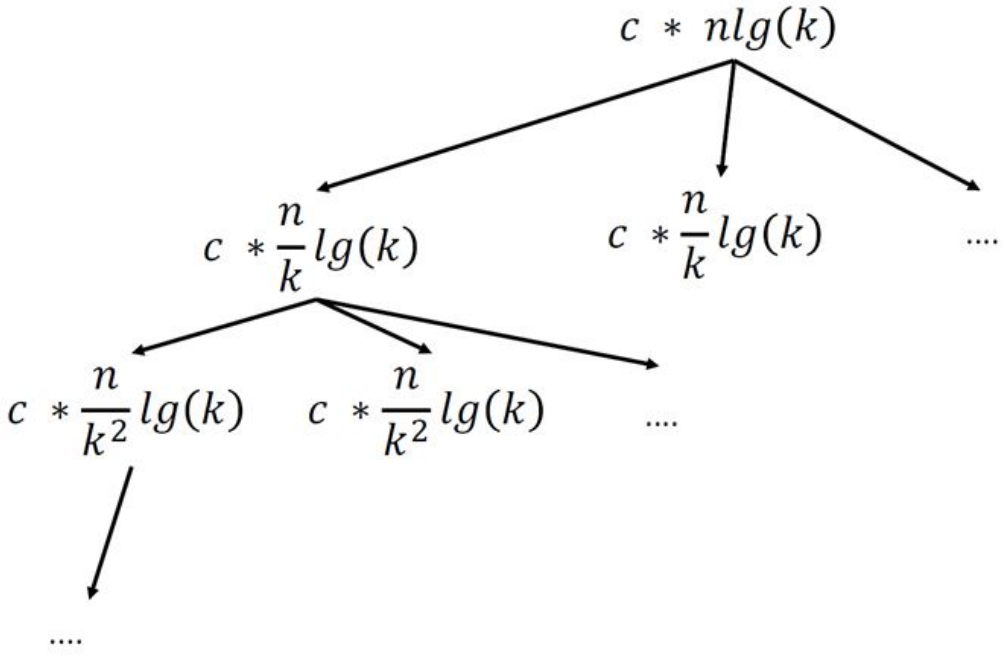
- By multiplying the work done at each level by the height of the tree, we get

$$\begin{aligned} & cn \log k \times \log_k n \\ &= cn\left(\frac{\log n}{\log k}\right) \log k \\ &= cn \log n \\ &= O(n \log n) \end{aligned}$$

no asymptotic improvement!!

Qn 3 Answer

$$T(n) = kT\left(\frac{n}{k}\right) + O(n \log k)$$



- By multiplying the work done at each level by the height of the tree, we get

$$\begin{aligned} & cn \log k \times \log_k n \\ &= cn\left(\frac{\log n}{\log k}\right) \log k \\ &= cn \log n \\ &= O(n \log n) \end{aligned}$$

no asymptotic improvement!!

need to do partition in place = more complicated and more cost

Qn 3 Answer

It turns out 2 and 3 pivot QuickSort is actually faster than regular QuickSort!

- Takes benefits of modern computer architecture and has reduced cache misses
- Memory speed not keeping up with processing speed
  - See https://arxiv.org/pdf/1511.01138.pdf
- In fact, Java's default sorting algorithm is a dual-pivot quicksort
  - https://github.com/openjdk/jdk/blob/151963259772 54ee73d96bf99be62c520de59af9/src/java.base/share/classes/java/util/DualPivotQuicksort.java#L2186

Qn 3 (2 pivot Quicksort Example)

[**57** 8 42 75 29 77 38 **24**]

- 57 and 24 are chosen as pivots
- Sort them (so that other elements can binary search the segments to be placed in

Qn 3 (2 pivot Quicksort Example)

[**57** 8 42 75 29 77 38 **24**]

- 57 and 24 are chosen as pivots
- Sort them (so that other elements can binary search the segments to be placed in

After first partitioning:

[8 **24** 42 38 29 **57** 75 77]

Qn 3 (2 pivot Quicksort Example)

[**57** 8 42 75 29 77 38 **24**]

- 57 and 24 are chosen as pivots
- Sort them (so that other elements can binary search the segments to be placed in

After first partitioning:

[8 **24** 42 38 29 **57** 75 77]

Next: recurse in the left segment (8), middle segment (42 38 29), and right segment (75 77).

Qn 4 Child Jumble

Come up with an efficient algorithm to match each child to their shoes.

Luckily, their feet (and shoes) are all of slightly different sizes. Unfortunately, they are all very similar, and it is very hard to compare two pairs of shoes or two pairs of feet to decide which is bigger. As such, you cannot compare shoes to shoes or feet to feet.

The only thing you can do is to have a toddler try on a pair of shoes.

Qn 4 Child Jumble

What are the constraints in this problem?

How should we compare children? (What is a feasible criteria)?

What algorithm would you use?

How fast is your algorithm?

Qn 4 Child Jumble

We can use quicksort!

The basic solution is to choose a random pair of shoes (e.g., the orange converse), and use it to partition the kids into “bigger” and “smaller” groups.

Larger feet

Smaller feet

pivot

Qn 4 Child Jumble

Along the way, you find one kid (“Alex”) for whom the orange Converse fit.

Smaller feet

pivot

Alex

Larger feet

Qn 4 Child Jumble

Then, Alex partitions the shoes.

Larger shoes

Smaller shoes

pivot
“Alex”

To summarize…

1. Choose a random shoe. 
2. Partition the children based on this shoe.

3. Along the way we will find child A that fits this random shoe.

4. Now we ask this child A to try all the other shoes.

Now we successfully partition **both** the shoes and children into 2 groups. So we can recurse on the “too big” side and “too small” side.

Qn 4 Child Jumble

If you choose the “pivot” shoes at random, you will get exactly the QuickSort recurrence, which results in a runtime of O(n log n) where n is the number of children.

Qn 5 Integer Sort (a)

How to sort 0s and 1s efficiently?

Could it be in-place? Is it stable?

0	1	0	1	1	1	0	0	0	1	1	0	1	0

Qn 5 Integer Sort (a)

How to sort 0s and 1s efficiently?

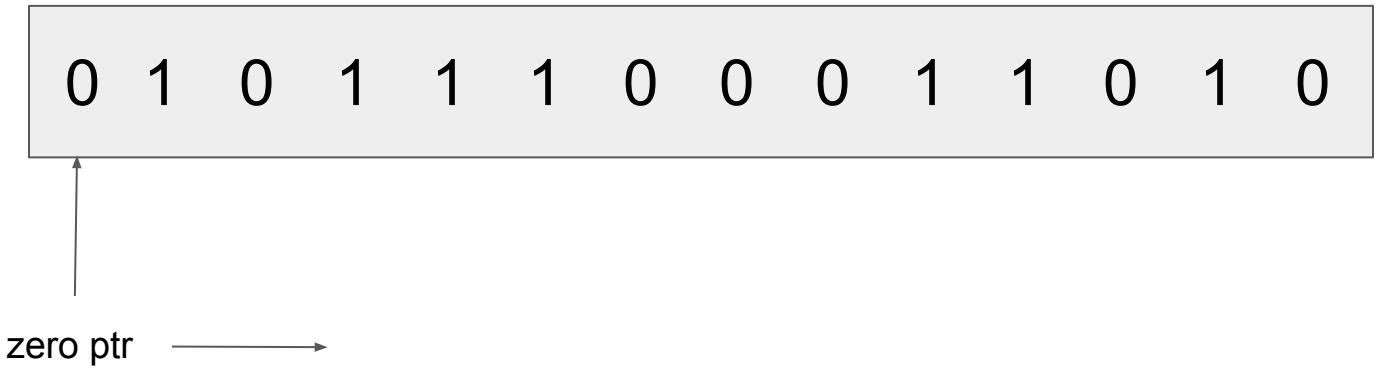
Could it be in-place? Is it stable?

Since there are only two "types" of elements in the array now, how could you simplify the problem?

0	1	0	1	1	1	0	0	0	1	1	0	1	0

Qn 5 Integer Sort (a)

Use QuickSort partitioning with two pointers, one at each side.



Qn 5 Integer Sort (a)

Use QuickSort partitioning with two pointers, one at each side.

Qn 5 Integer Sort (a)

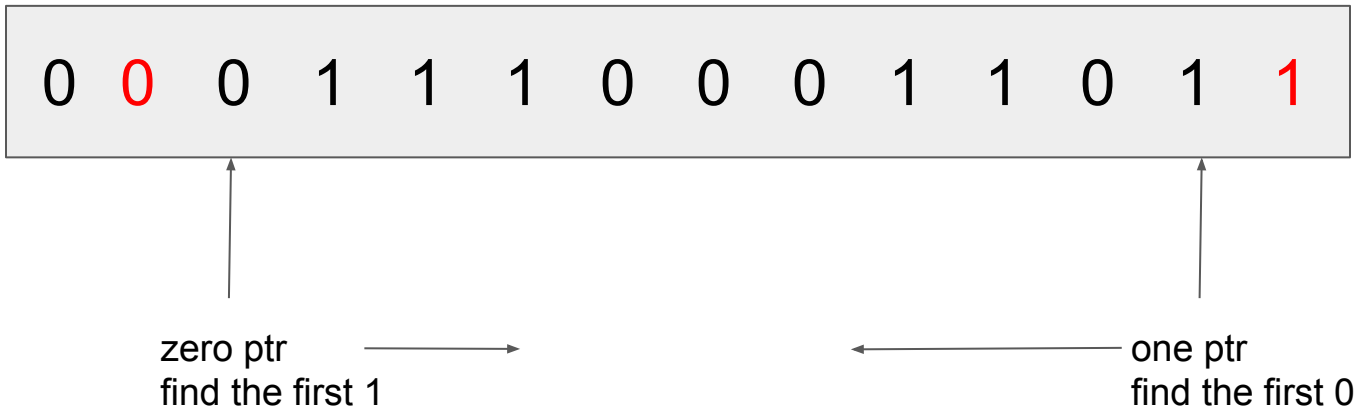
Use QuickSort partitioning with two pointers, one at each side.

Qn 5 Integer Sort (a)

Swap!

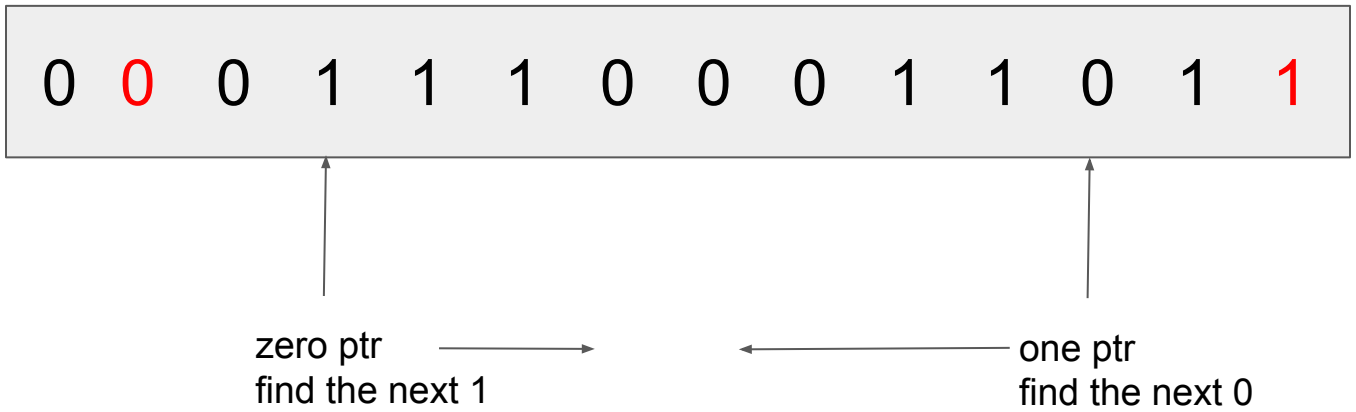
Qn 5 Integer Sort (a)

Find the next pair that could be switched.



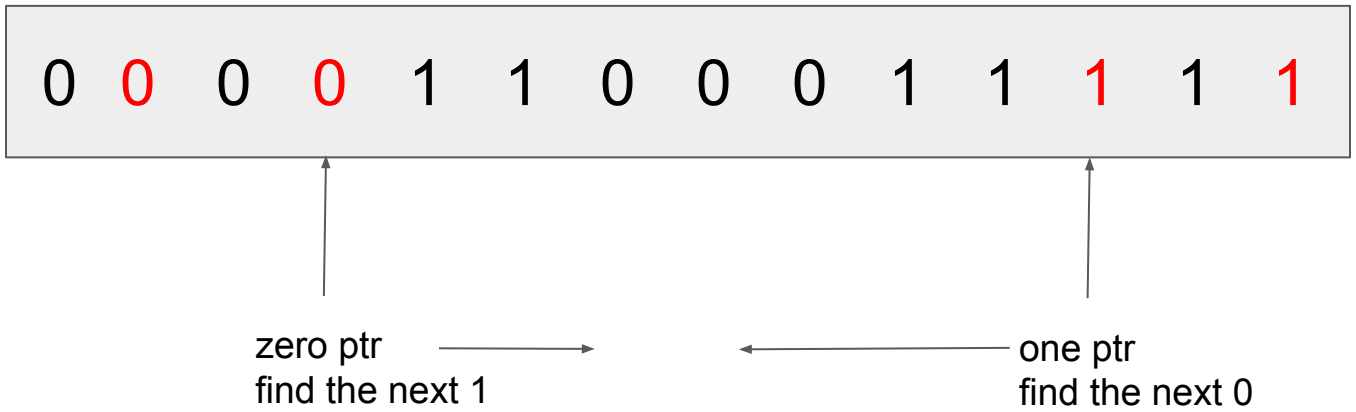
Qn 5 Integer Sort (a)

Find the next pair that could be switched.



Qn 5 Integer Sort (a)

Swap! (And repeat)



Runtime: O(n) - only going through the array once.

Qn 5 Integer Sort (b)

Now consider sorting an array consisting of integers between 0 and M, where M is a small integer.

(This time, do not try to do it in-place; you can use extra space to record information about the input array and you can use an additional array to place the output in.)

0	1	2	2	1	4	3	4	3	3	1	3	2	0

Qn 5 Integer Sort (b)

(1) First, go through the array once counting how many of each element you have.

• Then, go through the array and compute where the first element for each value should go in the output array. For example, to find out where the first ‘3’ in the array goes, sum up how many 0’s, 1’s, and 2’s there are in the input array: if there are 5 ‘0’s, 3 ‘’1’s, and 4 ‘’2’s, then the very first 3 is going to go in slot A[12] of the output array A (counting from zero). This can easily be computed from the array computed in the previous step.

• Treat this new array as a set of M pointers that point to the beginning of each block for each value. Now iterate through the input array and copy each item into the proper place indicated by the pointer. Then advance the pointer.

Qn 5 Integer Sort (b)

(1) First, go through the array once counting how many of each element you have.

0	1	2	2	1	4	3	4	3	3	1	3	2	0

Element Value	Count
0	2
1	3
2	3
3	4
4	2

Qn 5 Integer Sort (b)

(2) Go through the array and compute where the first element for each value should go in the output array.

0	1	2	2	1	4	3	4	3	3	1	3	2	0

Element Value	Count	Where the first index should be in output array
0	2	Index 0
1	3	Index 2 (=0+2)
2	3	Index 5 (=2+3)
3	4	Index 8 (=5+3)
4	2	Index 12(=8+4)

(3) Now iterate through the input array and copy each item into the proper place indicated by the pointer. Then advance the pointer.

0	1	2	2	1	4	3	4	3	3	1	3	2	0

Element Value	Count	pointers that point to the beginning of each block for each value
0	2	Index 0
1	3	Index 2 (=0+2)
2	3	Index 5 (=2+3)
3	4	Index 8 (=5+3)
4	2	Index 12(=8+4)

(3) Now iterate through the input array and copy each item into the proper place indicated by the pointer. Then advance the pointer.

0	1	2	2	1	4	3	4	3	3	1	3	2	0

Element Value	Count	Pointers that point to the beginning of each block for each value
0	2	Index 0
1	3	Index 2 (=0+2)
2	3	Index 5 (=2+3)
3	4	Index 8 (=5+3)
4	2	Index 12(=8+4)

This takes M space to keep track of how many of each item there are, and M space to keep track of the pointers to each region of the array.

Qn 5 Integer Sort (c)

Consider the following sorting algorithm for sorting integers represent in binary:

First, use the in-place algorithm from part (a) to sort by the first (high-order) bit.

First bit	Second bit	Third bits…
0	2	xxxxxxxx
0	3	xxxxxxxx
1	3	xxxxxxxx
1	4	xxxxxxxx
1	2	xxxxxxxx

Qn 5 Integer Sort (c)

Now, sort the two parts using the same algorithm, but using the second bit instead of the first. And then, sort each of those parts using the 3rd bit, etc.

First bit	Second bit	Third bits…
0	0	xxxxxxxx
0	1	xxxxxxxx
1	0	xxxxxxxx
1	0	xxxxxxxx
1	1	xxxxxxxx

Qn 5 Integer Sort (c)

Assuming that each integer is 64 bits, what is the running time of this algorithm?

First bit	Second bit	Third bits…
0	1	xxxxxxxx
0	0	xxxxxxxx
1	0	xxxxxxxx
1	1	xxxxxxxx
1	0	xxxxxxxx

Qn 5 Integer Sort (c)

Assuming that each integer is 64 bits, what is the running time of this algorithm?

Each level: O(n) time to sort

Repeat this process for 64 times (64 bits)

So it takes about 64n steps.

It is in-place, and just involves scanning the array 64 times, so it is fairly efficient in a lot of way.

Qn 5 Integer Sort (c)

When do you think this sorting algorithm would be faster than QuickSort? If you want to, write some code and test it out.

Qn 5 Integer Sort (c)

When do you think this sorting algorithm would be faster than QuickSort? If you want to, write some code and test it out.

64 n < n log n

64 < log n

n > 2^64

Unfortunately, since QuickSort runs in approximately Θ(nlogn) time, this will only likely beat QuickSort when n > 2^64 , which is a bit large!

Qn 5 Integer Sort (d)

Can you improve on this by using the algorithm from part (b) instead to do the partial sorting? What are the trade-offs involved?

Qn 5 Integer Sort (d)

Can you improve on this by using the algorithm from part (b) instead to do the partial sorting? What are the trade-offs involved?

For example, you might divide each integer up into 8 chunks of 8 bits each.

This will still take O(n) time for each partial sort. Now the “recursion” only goes 8 levels deep.

Since the algorithm is not in-place, the trade-off we make is space: It will take 256 integers worth of space to do the sorting using the part (b) algorithm.

Qn 6 Optional Question

What solutions did you find for Contest 1 (Treasure Island)?

Qn 6 Optional Question

Here's one approach for doing better:

- Choose a set of n/k keys that are still unexamined and put them in set S.
- Check if there is at least one correct key in S by using all the keys not in S to unlock the chest.
- If there is at least one correct key in S, then repeatedly use binary search to find the correct keys in S. Each correct key you find here will take O(log(n/k)) time.
- Mark the keys in set S as examined, and repeat with the remaining keys.

Qn 6 Optional Question

Since each correct key you find takes O(log(n/k)) time, you will spend O(k log(n/k)) time doing binary searches for correct keys.

How many times will you query a set S and discover no correct keys?

There are only k different sets with n/k keys, so this will take at most k queries. So the total number of queries is k log(n/k) + k. This is optimal, since Ω(k log(n/k)) is the best you can do.

Bonus Qn

Problem 3. **(Chicken Rice)**
Imagine you are the judge of a chicken rice competition. You have in front of you n plates of chicken rice. Your goal is to identify which plate of chicken rice is best.

Problem 3.a. A simple algorithm:

- Put the first plate on your table.
- Go through all the remaining plates. For each plate, taste the chicken rice on the plate, taste the chicken rice on the table, decide which is better. If the new plate is better than the one on your table, replace the plate on your table with the new plate.
- When you are done, the plate on your table is the winner!

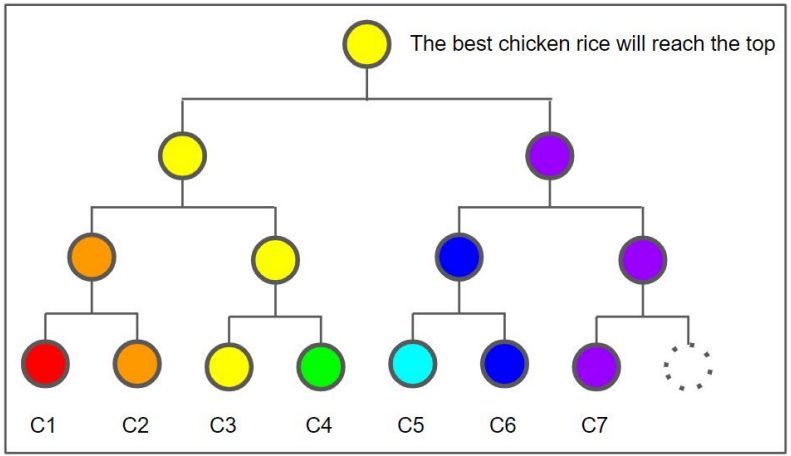
Assume each plate begins containing n bites of chicken rice. When you are done, in the worst-case, how much chicken rice is left on the winning plate?

- Use quickselect!
- Analysis
  - In one step of quickselect (with an array of size n)
    - The pivot plate has $n - 1$ bites eaten
    - All other plates have 1 bite eaten
  - If the pivot plate is chosen at random, the median plate has an expected cost of $\frac{1}{n}(n-1) + \left(1 - \frac{1}{n}\right)1 = 2\left(1 - \frac{1}{n}\right) = 2 - \frac{2}{n} \leq 2$ bites eaten
    - In other words, at each level of recursion, there are at most 2 bites eaten from the median plate in expectation

Bonus Qn

Problem 3.b. Oh no! We want to make sure that there is as much chicken rice left on the winning plate as possible (so you can take it home and give it to all your friends). Design an algorithm to maximize the amount of remaining chicken rice on the winning plate, once you have completed the testing/tasting process. How much chicken rice is left on the winning plate? How much chicken rice have you had to consume in total? (Give a tight asymptotic bound.)

- Use a tournament tree!
  - Group the plates of chicken rice into pairs and compare within each pair to get a winner
  - Group the winners of the previous round into pairs and compare within each pair to get a winner
  - Repeat until we have only 1 plate left
- In the worst case, we consume
  - $O(\log n)$ bites from the winning plate
    - Height of the tree
  - $O(n)$ bites overall
    - Each comparison takes 2 bites
    - Each comparison removes 1 plate



Bonus Qn

Problem 3.c. Now I do not want to find the best chicken rice, but (for some perverse reason) I want to find the median chicken rice. Again, design an algorithm to maximize the amount of remaining chicken rice on the median plate, once you have completed the testing/tasting process. How much chicken rice is left on the median plate? How much chicken rice have you had to consume in total? (Give a tight asymptotic bound. If your algorithm is randomized, give your answers in expectation.)

- Use quickselect!
- Analysis
  - In one step of quickselect (with an array of size n)
    - The pivot plate has $n - 1$ bites eaten
    - All other plates have 1 bite eaten
  - If the pivot plate is chosen at random, the median plate has an expected cost of $\frac{1}{n}(n-1) + \left(1 - \frac{1}{n}\right)1 = 2\left(1 - \frac{1}{n}\right) = 2 - \frac{2}{n} \leq 2$ bites eaten
    - In other words, at each level of recursion, there are at most 2 bites eaten from the median plate in expectation

- Analysis
  - With high probability and in expectation, the recursion will terminate in $O(\log n)$ levels
  - In the average case, we consume
    - $O(\log n)$ bites from the median plate
    - $O(n)$ bites overall
      - If we select pivots randomly, we should get a good split most of the time

- In the solution sheet, an alternative mentioned is to use an AVL tree
- However, since this solution is deterministic (not randomised), it is possible for there to be the following bad input:
  - Insert the median plate into the AVL tree
    - By default, the first plate inserted will be the root of the AVL tree
  - Insert all other plates in an order such that no rotations ever occur
    - This means that the median plate stays at the root throughout
- In the worst case, we consume $O(n)$ bites from the median plate!