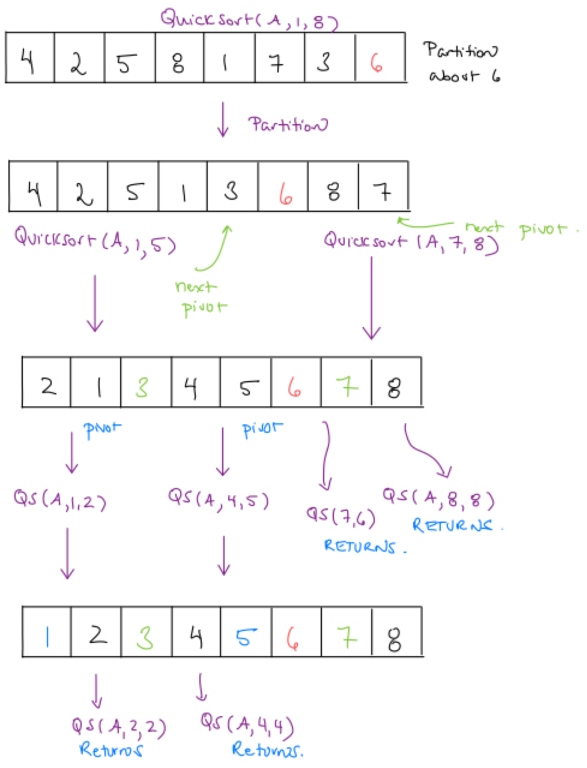
Practice Set 5: solutions

.

Problem 1



Problem 2

- The original Partition algorithm that we saw in Randomized-Select does not run in place. Furthermore, it returns the **rank** of the pivot. The Partition algorithm used by Quicksort, returns the **index** of the pivot. These are not always the same, in the start index of the array is not 1.
- If we use the simple version of Partition, then the algorithm no longer run in place, and each recursive call will use another array to perform the partition step. However, the overall run time is unaffected, since the simpler version of the partition algorithm is $\Theta(n)$. The recurrence for Quicksort remains the same, and therefore the runtime is still expected to be $O(n \log n)$ when using random pivots, and $O(n^2)$ in the worst-case.
- If we use use the in-place partitioning algorithm in Randomized-Select, the runtime of the partition step is unchanged. Therefore the worst-case runtime is still $O(n^2)$ and the best-case runtime is still $O(n)$.
- Both QuickSort and Randomized-Select pick a random pivot and perform a partition algorithm in $\Theta(n)$. The only difference is that QuickSort makes a recursive call to *both* the left and right subarrays, whereas Randomized-Select only makes a recursive call to *one* of the subarrays. The best-case recurrence for QuickSort is therefore $T(n) = 2T(n/2)+cn$, which has solution $O(n \log n)$ whereas that for Randomized-Select is $T(n) = T(n/2) + cn$, which has solution $O(n)$. The worst-case recurrence for both QuickSort and Randomized-Select are $T(n) = T(n-1) + cn$, which has solution $O(n^2)$.

Problem 3

This algorithm works just as Quicksort does, except that instead of a random pivot, the pivot selected is the one that splits the array into approximate sizes of one third and two thirds. The select algorithm from step 1 and the partitioning algorithm from set 3 both run in time $O(n)$. Steps 4 runs in time $T(n/3)$ and step 5 runs in time $T(n/3)$. Therefore the runtime recurrence is:

$$T(n) = T(2n/3) + T(n/3) + cn$$

We can show the runtime is $O(n \log n)$ using substitution:

Goal:. Show that $T(n) \leq dn \log n$:

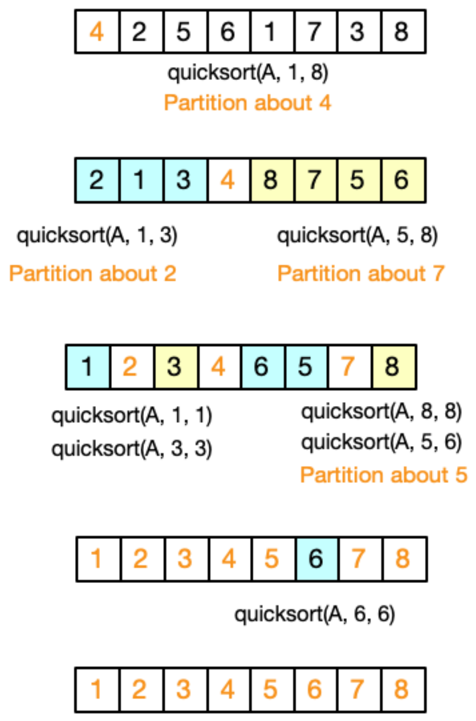
Assume:. $T(n/3) \leq d\frac{n}{3}\log(n/3)$ and $T(2n/3) \leq d\frac{2n}{3}\log(2n/3)$

Substitute:.

$$\begin{aligned}
T(n) &= T(2n/3) + T(n/3) + cn \\
&\leq d\frac{2n}{3}\log(2n/3) + d\frac{n}{3}\log(n/3) + cn \\
&=\leq d\frac{2n}{3}(\log(2) + \log n - \log 3) + d\frac{n}{3}(\log(n) - \log 3) + cn \\
&= dn\log n - (0.918)dn + cn \\
&\leq dn\log n
\end{aligned}$$

as long as $-0.198d + c < 0$.

Problem 4



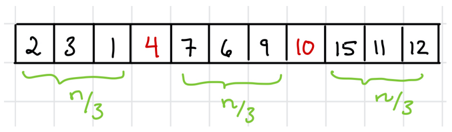
Best-case: It may be that the median of the array is actually the median of our set of three elements. If this happens very time, the runtime recurrence is $T(n) = 2T(n/2) + cn$, which we showed already has solution $O(n\log n)$.
Worst-case:
This approach may improve our chances of picking a good pivot. However, there is still a possibility that the array is split very unequally. If we happen to pick the three largest or the three smallest elements in the set of three, then the median will be either the second-largest or the second-smallest element. In this case, the recurrence for the runtime of quicksort is $T(n) = T(n-2) + cn$. This is similar to the original worst-case runtime, which is $T(n) = T(n-1) + cn$.
We can solve $T(n) = T(n-2) + cn$:

$$\begin{aligned}
T(n) &= T(n-2) + cn \\
&= T(n-4) + c(n-2) + cn \\
&= T(n-6) + c(n-4) + c(n-2) + cn \\
&= T(n-8) + c(n-6) + c(n-4) + c(n-2) + cn \\
&= \ldots \\
&= c(n + (n-2) + (n-4) + \ldots +) \\
&= \Theta(n^2)
\end{aligned}$$

Likelihood of the worst-case: In the original version of quicksort, the worst-case scenario occurs when the largest or smallest element is chosen as the pivot. The chance of this happening is $2/n$. In this new version, the worst-case occurs when the top *two* or bottom *two* elements are chosen. There is a smaller change of this happening. (The exact chance is $\frac{24}{n(n-1)(n-2)}$).

Footnote: A student asked how to explicitly show that the above sum is quadratic. Recall that $1 + 2 + 3 + \ldots + n = sum_{k=1}^{n} k = n(n+1)/2$. *Similarly,* $2 + 4 + 6 + \ldots + n = \sum_{k=1}^{n/2} 2k = n(n+2)/4$

Problem 5
The best-case runtime would be when the two pivots divided the elements into equal-sized sets. For example:

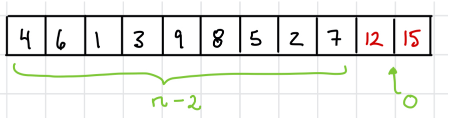


The partitioning into groups of 3 would also take $\Theta(n)$ time (a very easy approach would be an algorithm that is not in-place).
The recurrence for the best-case runtime would be:

$$T(n) = 3T(n/3) + \Theta(n)$$

which has solution $\Theta(n \log n)$ by the master method. So the best-case runtime of this algorithm is the same as the original Quicksort.

The worst-case runtime is when the two pivots selected are either the two *largest* or the two *smallest*. In this case, the array would be partitioned as:



The recurrence for the worst-case runtime is:

$$T(n) = T(n-2) + \Theta(n)$$

which can be expanded out as:

$$\begin{aligned} T(n) &= T(n-2) + cn \\ &= (T(n-4) + c(n-2)) + cn \\ &= T(n-6) + c(n-4) + c(n-2) + cn \\ &= \ldots \\ &= \Theta(n^2) \end{aligned}$$

Footnote: A student asked how to explicitly show that the above sum is quadratic. Recall that $1+2+3+\ldots+n = sum_{k=1}^{n} k = n(n+1)/2$*. Similarly,* $2+4+6+\ldots+n = \sum_{k=1}^{n/2} 2k = n(n+2)/4$

Problem 6

Suppose we use **Select** to find the median of the array. We saw that this algorithm takes time $\Theta(n)$. We could then use the median as the pivot and continue with Quicksort as usual, where the recursive calls will each be on arrays of size approximately $n/2$. The steps of the algorithm work just like the original Quicksort steps:

if $s < f$:

- Let p = Select(A, $\lfloor n/2 \rfloor$), where p is now the actual value of the *median*. **Runtime:** $\Theta(n)$
- Loop through the array to find the index of p. Store this value in r. **Runtime:** $\Theta(n)$
- Partition A between s and f using pivot at index r. **Runtime:** $\Theta(n)$
- Recursively sort the left and right subarrays: Quicksort(A,s,r-1), Quicksort(A,r+1,f). Note that each subarray will have size approximately $n/2$. **Runtime:** $2T(n/2)$

TOTAL RUNTIME: $T(n) = \Theta(n) + \Theta(n) + 2T(n/2) = \Theta(n) + 2T(n/2)$, which has solution $\Theta(n \log n)$, since it is the same recurrence as Mergesort. This new version of Quicksort *guarantees* a worst-case runtime of $O(n \log n)$, whereas the original Quicksort has a worst-case runtime of $O(n^2)$. The drawback of this technique is that the implementation of this algorithm is very complex, and the constant behind the runtime of $\Theta(n)$ for Select is quite high. Furthermore, it does not run in-place, and uses a huge amount of external storage. It achieves a runtime that is the same as the average case of Quicksort, and although it is better than the worst-case of Quicksort, recall that the worst-case of Quicksort is very rare.

Problem 7

Included in the solution (for comparison sake) is Heapsort.

Algorithm	**Sorted**	**Reverse Sorted**	**Randomly Sorted**	**Worst case**
QuickSort with random pivot	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Note that Selection sort and Mergesort are *not* affected by the input type. They execute the same number of steps regardless of input. Note that Insertion Sort and Bubble sort on a random array both have a runtime that is just as bad as their worst-case. Quicksort on the other hand, has a runtime of $O(n \log n)$ when the input is random, much better than its worst case!

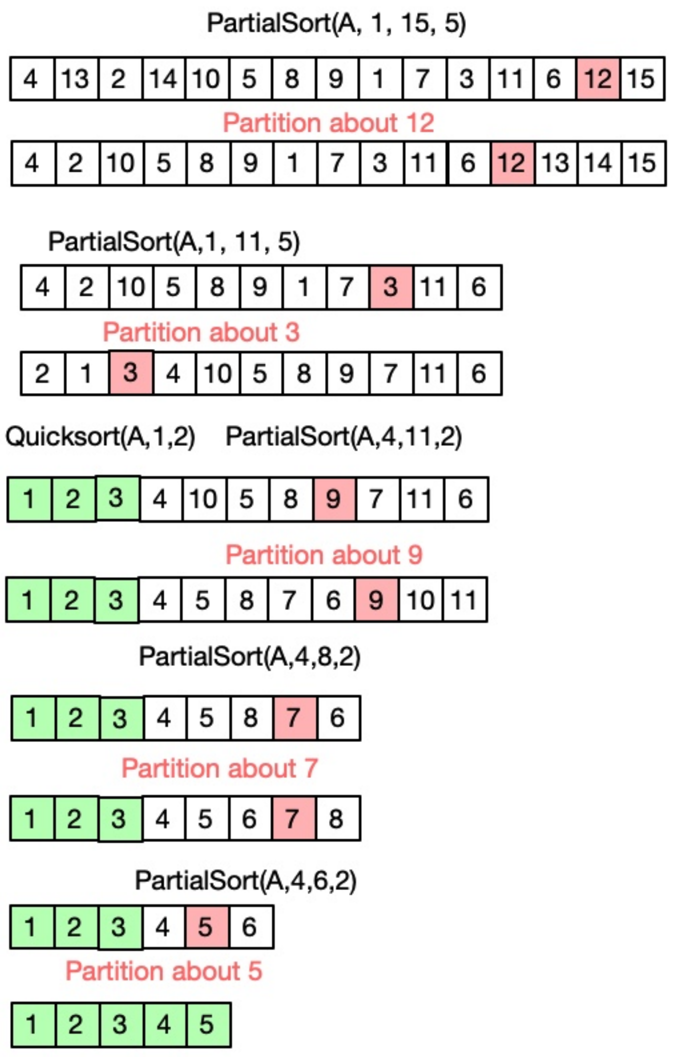
Problem 8

Quicksort certainly runs in place when using the in-place partitioning algorithm. Insertion sort, Bubble sort and Selection sort also run in place: no external arrays are used. Merge sort however uses an external array during the merge step, and therefore doesn't run in place.

Problem 9

The algorithm is a combination of randomized-select and quicksort. Notice that the recursive calls are designed to sort all elements *below* the rank k. When the element of rank k is to the left of the pivot, we make a recursive call to the left. When the element of rank k is to the right of the pivot, we we call quicksort to the left subarray and make a recursive call to the right. The base cases of the recursion as just like in randomzied-select. The algorithm terminates when the item of rank k is found.

Below is the execution on the given array when $k = 5$. Note that the result is that the first 5 elements of the array are sorted.



We could get the same result with the following steps:

1. m = Select(k)
2. Partition the array about m
3. Call Quicksort on subarray A from s to $s + k - 2$.

The runtime of the above 3 steps is $\Theta(n)$ for step 1, $\Theta(n)$ for step 2, and an expected runtime of $O(k \log k)$. Since $k \leq n$, the expected runtime is $O(n \log n)$ and the worst-case runtime is $O(n^2)$.

Hands on application:

Test the following procedures with different sized inputs:

```
def insertionSort(arr, s, f):
    # Traverse through 1 to len(arr)
    for i in range(s+1, f+1):
        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= s and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key

def partition(l, r, nums):
    # Last element will be the pivot and the first element the pointer
    pivot, ptr = nums[r], l
    for i in range(l, r):
        if nums[i] <= pivot:
            # Swapping values smaller than the pivot to the front
            nums[i], nums[ptr] = nums[ptr], nums[i]
            ptr += 1
    # Finally swapping the last element with the pointer indexed number
    nums[ptr], nums[r] = nums[r], nums[ptr]
    return ptr

def quicksort(l, r, nums):
    if l >= r:  # Terminating Condition
        return nums
    elif l < r:
        pi = partition(l, r, nums)
        quicksort(l, pi-1, nums)  # Recursively sorting the left values
        quicksort(pi+1, r, nums)  # Recursively sorting the right values
    return nums
```

Here is the new version of Quicksort which uses insertion sort when the input size is small:

```
def quicksort_update(l, r, nums):
    if l >= r:
        return nums
    if r - l <= 5:
        insertionSort(nums, l, r)
        return nums
    elif l < r:
        pi = partition(l, r, nums)
        quicksort_update(l, pi-1, nums)  # Recursively sorting the left values
        quicksort_update(pi+1, r, nums)  # Recursively sorting the right values
    return nums
```

You can create random input and time a function like this:

```
i = 1
seed(i)
N = 1000
values = randint(0, N*N, N)
import time
a = time.time()
quicksort(0, len(values)-1, values)
Total = time.time() - a
```