Problem 1

(10 pts) Suppose that we modify the `Partition` *algorithm in QuickSort in such a way that on on every third level of the recursion tree it chooses the worst possible pivot, and on all other levels of the recursion tree* `Partition` *chooses the best possible pivot. Write down a recurrence relation for this version of QuickSort and give its asymptotic solution. Then, give a verbal explanation of how this* `Partition` *algorithm changes the running time of QuickSort.*

I have no idea how to describe the recurrance relation.

Without the 'bad' choice of pivot every 3 recursions, what is described would be best-case behaviour of QuickSort - which means the recursion tree would have a depth of $log_2(n)$. With the addition of the extra layers with bad pivots, this depth increases to a maximum of $\frac{4}{3}log_2(n) = \Theta(lg(n))$ (actually each of the bad pivot layers takes away 1 from the problem size, so the amount of additional layers would be less than $\frac{1}{3}log_2(n)$). There is still $\Theta(n)$ work being done on each layer (regardless of a good or bad pivot), so the total running time remains $\mathcal{O}(nlg(n))$. The only change is a constant factor, which is irrelevant in the asymptotic analysis.

Problem 2

(10 pts) Mr. Ollivander, of Ollivanders wand shop, has hired you as his assistant, to find the most powerful wand in the store. You are given a magical scale which "weighs" wands by how powerful they are (the scale dips lower for the wand which is more powerful). You are given n wands $W_1, \ldots, W_n$, each having distinct levels of power (no two are exactly equal).

(a) *Consider the following algorithm to find the most powerful wand:*

i. *Divide the n wands into $\frac{n}{2}$ pairs of wands.*

ii. *Compare each wand with its pair, and retain the more powerful of the two.*

iii. *Repeat this process until just one wand remains.*

Illustrate the comparisons that the algorithm will do for the following $n = 8$ input:

$$W_1 : \frac{3}{2},\ W_2 : \frac{5}{2},\ W_3 : \frac{1}{2},\ W_4 : 1,\ W_5 : 2,\ W_6 : \frac{5}{4},\ W_7 : \frac{1}{4}\ , W_8 : \frac{9}{4}$$

iteration 1 (arrow direction indicates which wand is kept):
$W_1 \rightarrow W_5, W_2 \leftarrow W_6, W_3 \leftarrow W_7, W_4 \rightarrow W_8$

iteration 2:
$W_5 \rightarrow W_2, W_3 \rightarrow W_8$

iteration 3:
$W_2 \leftarrow W_8$

(b) *Show that for n wands, the algorithm (2a) uses at most n comparisons.*

The recurrance relation for the amount of comparisons (i.e. the running time) of the algorithm is given by:

$$\begin{aligned}
T(n) &= T(n/2) + n/2 && (1)\\
&= (T(n/4) + n/4) + n/2 && (2)\\
&= ((T(n/8) + n/8) + n/4) + n/2 && (3)\\
\dots\dots & && (4)\\
&= T(\frac{n}{2^i}) + \sum_{j=1}^{i} \frac{n}{2^i} && (5)\\
&= T(\frac{n}{2^i}) + n\sum_{j=1}^{i} \frac{1}{2^i} && (6)\\
&= T(\frac{n}{2^i}) + \frac{n}{2}(\frac{1-(\frac{1}{2})^i}{1-\frac{1}{2}}) && (7)
\end{aligned}$$

The base case here is when $\frac{n}{2^i} = 2$, as only one comparison can be made before the algorithm halts (i.e. $T(2) = 1$). The base case occurs when $i = lg(\frac{n}{2})$.

Thus the algorithm takes exactly:

$$\begin{aligned}
&T(\frac{n}{2^{lg(\frac{n}{2})}}) + \frac{n}{2}(\frac{1-(\frac{1}{2})^{lg(\frac{n}{2})}}{1-\frac{1}{2}}) && (8)\\
&= 1 + \frac{n}{2}(\frac{1-\frac{2}{n}}{1-\frac{1}{2}}) && (9)\\
&= 1 + \frac{n}{2}(2 - \frac{4}{n}) && (10)\\
&= n - 1 && (11)
\end{aligned}$$

comparisons.

(c) *Describe an algorithm that uses the results of (2a) to find the second most powerful wand, using at most log_2n additional comparisons. There is no need for pseudocode; just write out the steps of the algorithm like we have written in (2a). Hint: if you follow sports, especially wrestling, read about the* ***repechage****.*

From the results of the algorithm in 2a, we know the following information:

- $W_1 < W_5 < W_2$
- $W_6 < W_2$
- $W_7 < W_3 < W_8 < W_2$
- $W_4 < W_8 < W_2$

From this we can reason that wands 1, 7, 3, and 4 will not be in the running for second place as there are other wands more powerful than them which are not the most powerful. These wands are W_5, W_6, and W_8 - we need to compare these wands to each other to find the second most powerful.

Assuming the algorithm from 2a also kept track of which wands were compared with which wands, an algorithm for (a) finding these 'secondary' wands and (b) finding the most powerful among them is:

i. look at the comparison in the last iteration. Let $p =$ the most powerful wand $=$ the greater wand in this comparison. Let $s =$ the other wand.
ii. go up through each iteration i until i=0, compare each wand that was previously compared to p to s. Set $s =$ the greater of the two.
iii. when i=0 (i.e. past the beginning of the result set) $s =$ the second most powerful wand.

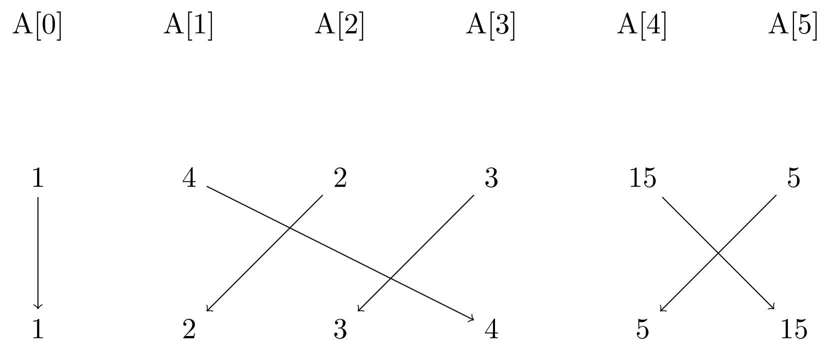
Since there are at most log_2n iterations, with one comparison to p in each iteration - there are at most log_2n additional comparisons. In reality this is $log_2n - 1$ as the last iteration gives us a starting value for s with no comparison needed.

(d) *Show the additional comparisons that your algorithm in (2c) will perform for the input given in (2a).*

$W_8 \leftarrow W_5 \implies s = W_8$
$W_8 \leftarrow W_6 \implies s = W_8$

Problem 3

(20 pts) For obtuse historical reasons, Prof. Dumbledore asks his students to line up in ascending order by height in a very tight room with little extra space. Similar to Alex the African Grey parrot (look it up!), the students, being bored, decided to play a little trick on Prof. Dumbledore. They lined up in order by height – almost. They made sure that each person was no more than k positions away from where they were supposed to be (in ascending order), but this allowed them to significantly mess up the precise ordering. Here is an example of an array with this property when $k = 2$:



(a) *Write down pseudocode for an algorithm that would sort such an array in place–so it fits in the tight room–in time $\mathcal{O}(nk \log k)$. Your algorithm can use a function sort(A, l, r) that sorts the subarray $A[l], \ldots, A[r]$ in place in $\mathcal{O}((r - l)\log(r - l))$ steps (assuming r > l).*

Assuming 0-based indexing:

```
def studentsort(A, k):
    for n in [0 .. A.length]
        if (n+k < A.length):
            sort(A, n, n+k)
        else: sort(A, n, A.length-1)
```

This algorithm runs in $O(nk \log k)$ as it simply loops through n elements of A, and calls sort() on the next k elements. Because sort() runs in $O(n \log n)$ time, the entire algorithm runs in $O(nk \log k)$ time.

This algorithm is correct as an unsorted element in the array cannot be k places away from its correct position, thus by sorting blocks of k elements at a time we can sort the whole array. Consider the first iteraton of the loop: the minimum element in the array has to be within k positions of the first position, and since we are sorting only the first k elements, we know that the minimum element will be sorted into the correct (first) position. Now the loop moves to the second position in the array where the same logic follows (i.e. the second least element in the array has to be within the *next* k elements being sorted). Essentially the left region of the array (A[0 .. n-1]) is always sorted, and when $n = A.length$ the loop exits, implying the whole array is sorted.

(b) *Suppose you are given to an auxiliary room which can fit k + 1 students. Modify your previous algorithm to sort the given array in time $\mathcal{O}(nk)$.*

Assuming 0-based indexing:

```
def studentsortv2(A, k):
    tmp = 0
    for n in [0 .. A.length-1]:
        for j in [1 .. k-1]:
            if (n+j < A.length and a[n] > a[n+j]):
                tmp = a[n]
                a[n] = a[n+j]
                a[n+j] = tmp
```

In the worst case, this algorithm runs in (5 x O(1)) [the if conditions, plus the swapping assignments] x O(k-1) [the inner loop] x O(n) [the outer loop] = O(nk).

Essentially this algorithm is doing the same thing as the previous algorithm - looking ahead the next k elements and putting the correct element in the first 'unknown' slot. However, in this algorithm we don't waste time sorting all k elements. The only element we can be sure about sorting into the correct final position is the element going into slot n (i.e. the element at n+1 may need to be put at position k+1 in the next iteration, so it is wasted computation to sort it in this iteration). This algorithm simply looks at each next element up til n+k and swaps the element at n with the lowest - preserving the property that the leftmost region of A is sorted.

(c) *With the same extra room as in the previous part, modify heap sort using a binary min heap of size $k+1$ to sort the given array in time $\mathcal{O}(n \log k)$.*

Assuming 0-based indexing, heapify(A, i, j) is a function that performs the necessary operations on A to maintain the min-heap property from starting index i to ending index j.

```
def studentsortv3(A, k):
    heapify(A, 0, A.length-1)
    for n in [0 .. A.length-1]:
        if (n+k >= A.length):
          k = k - (A.length-n)
        heapify(A, n, n+k)
```

This algorithm runs in $\mathcal{O}(n \log k)$ as heapify(A,i,j) is called exactly n times, and takes $\mathcal{O}(\log(j-i))$ per the definition of heapsort. Since (j-i) = (n+k-n) = k, heapify will take $\mathcal{O}(\log k)$

Since we are moving the front of the heap forward instead of the end of the heap backward, there is no need to swap elements. This uses the same principle as our last two examples (that is, having the minimum element sorted to the leftmost 'unknown' position), however it does the sorting more efficiently.

Problem 4

(20 pts) Consider the following strategy for choosing a pivot element for the Partition subroutine of QuickSort, applied to an array A.

- *Let n be the number of elements of the array A.*
- *If $n \leq 24$, perform an Insertion Sort of A and return..*
- *Otherwise:*
  - *Choose $2\lfloor n^{1/3} \rfloor$ elements at random from n; let S be the new list with the chosen elements.*
  - *Sort the list S using Insertion Sort and use the median m of S as a pivot element.*
  - *Partition using m as a pivot.*
  - *Carry out QuickSort recursively on the two parts.*

(a) *How much time does it take to sort S and find its median? Give a Θ bound.*

Insertion sort runs in $O(n^2)$, and finding the median once S is sorted is an atomic operation which runs in $\Theta(1)$. Therefore the bound for these operations is $\Theta(n^2)$

(b) *If the element m obtained as the median of S is used as the pivot, what can we say about the sizes of the two partitions of the array A?*

They should be roughly equal, as the median of S should be roughly equal to the median of (sorted) A because S is a random sample.

(c) *Write a recurrence relation for the worst case running time of QuickSort with this pivoting strategy.*

$T(n) = T(n-1) + O(n^2)$
$T(24) = O(n^2)$

References

1. CLRS
2. `https://en.wikipedia.org/wiki/Geometric_series`
3. `https://en.wikipedia.org/wiki/Heapsort`