1 Complexity

1. What is the time complexity of the following sorting routine?

```
sort(array) {
  split = array.length - 2;
  left  = sort(array[1..split])
  right = sort(array[split+1..array.length])
  merge(left,right)
}
```

I will assume that the sorting algorithm terminates when an array of either length 0 or 1 is passed to it.

Sort calls itself on an input of size(n - 1) and itself on an input of size 1. Since we know that sort terminates on inputs of size 1; the call on an array of size 1 terminates in $O(1)$ time. Merge is also an $O(n)$ time operation.

Using this we can form a recurrence relation:

$$\begin{aligned} T(n) &= T(n-1) + T(1) + \Theta(n) \\ T(n) &= T(n-1) + \Theta(n) \end{aligned} \tag{1}$$

Substitute in $T(n) = k\frac{n(n+1)}{2}$.

$$\begin{aligned} T(n) &= k\frac{n(n-1)}{2} + kn \\ T(n) &= k\frac{n^2 - n + 2n}{2} \\ T(n) &= k\frac{n^2 + n}{2} \\ T(n) &= k\frac{n(n+1)}{2} \end{aligned} \tag{2}$$

So the solution is $T(n) = \frac{n(n+1)}{2}$ which implies that $T(n) \in \Theta(n^2)$.
So the sorting routine has complexity $\Theta(n^2)$

2. What is the time complexity if I replace line 2 as below?

```
split = round_down(log_base_2(array.length))
```

The sorting algorithm now calls itself twice on lists of length $n - \lg n$ and $\lg n$. Then does a $\Theta(n)$ merge.

So the recurrence relation is now:

$$T(n) = T(n - \lg(n)) + T(\lg n) + \Theta(n) \tag{3}$$

The time complexity is $\Theta\left(\frac{n^2}{\lg n}\right)$.

I will prove this by applying two comparison tests – one to give an upper bound and one to give a lower bound.

We recursively call the function on inputs of size $n - \lg n$ and $\lg n$. If the function were to have the recurrence relation $T(n) = T(n-k)+T(k)+\Theta(n)$ for some $\lg n \leq k \leq \frac{n}{2}$ we will always divide the sequence the same or better – which would give a better overall complexity. If we set $k = \lg i$ where i is the size of the array in the first function call then we would obtain the same or a better complexity. We must assume that the time taken to execute $T(n)$ for $n \leq 2\lg i$ is constant.

This means the new recurrence relation always divides the recurrence into more equally-sized groups than the original one so is guaranteed to have a complexity better than or equal to the original recurrence relation – so we can form a lower bound for the complexity of the recurrence relation by solving the new one.

The new series forms the recurrence relation:

$$\begin{aligned} S(n) &= S(n-k) + S(k) + \Theta(n) \\ S(n) &= S(n-k) + \Theta(n) \end{aligned} \tag{4}$$

Since we decrease the size of the original input by $\lg i$, there are $\frac{i}{\lg i}$ calls – each doing $\Theta(n)$ work.

$$\begin{aligned} S(n) &= \lg n \sum_{k=2}^{k=\frac{n}{\lg n}} k \\ S(n) &= \lg n \left(\frac{\frac{n}{\lg n}(\frac{n}{\lg n}+1)}{2} - 3 \right) \\ S(n) &= \frac{n^2 + n\lg n}{\lg n} - 3\lg n \\ S(n) &\in \Theta\left(\frac{n^2}{\lg n}\right) \end{aligned} \tag{5}$$

Since the complexity of $S(n)$ is better than or equal to the complexity for $T(n)$ and $S(n) \in \Theta\left(\frac{n^2}{\lg n}\right)$; this proves that $T(n) \in \Omega(\frac{n^2}{\lg n})$.

Now to prove an upper bound. For the purposes of this I will construct a series R which is marginally worse than T but still fairly tight. I will simplify R into a more manageable (but still tight) form and then make the recurrence a looser upper bound by assigning part of the expression a *higher* complexity than the other part. Since we are forming an upper bound, we are guaranteed that the new expression is still worse than the original one. This means that we can prove an upper bound via this method. Lemma 1 formalises this idea.

Lemma 1:

$$\begin{aligned} f(n) \in O(g(n)) \implies \\ f(n) + g(\lg n) \in \max(O(f(n)), O(g(\lg n))) \end{aligned} \tag{6}$$

Since $\lg n \geq 1$ for all n, we know that $T(n) \in O(n^2)$ from part (1).

So using this lemma we can substitute $T(\lg n) = g(n)$ for any n such that $n^2 \in g(n)$ and we will not *decrease* the overall complexity. We can use this to find the upper bound without risk of decreasing the complexity.

If we start with $R(2n)$ and reduce it by $\lg n$ each time (rather than $\lg 2n$) – this expression is worse than $T(2n)$ since it decreases by a (marginally) smaller amount. In order to reduce $2n$ to n we will need to repeat this $\frac{n}{\lg n}$ times. This leads to the (somewhat) tight but slightly worse recurrence relation below:

$$R(2n) = R(n) + \frac{n}{\lg n} R(\lg n) + n \tag{7}$$

I will prove that $R(n) = \frac{cn^2}{\lg n}$ for an arbitrary constant c.

Substitute $R(n) = \frac{cn^2}{\lg n}$ and $R(\lg n) = \frac{4cn2^n}{n+1} - c2^n - n$. Note that $\frac{4cn2^n}{n+1} - c2^n - n \in \Theta(2^n)$ and $\frac{n^2}{\lg n} \in O(2^n)$ so Lemma 2 holds and we can perform the substitution.

$$
\begin{aligned}
R(2n) &= \frac{cn^2}{\lg n} + \frac{n}{\lg n}\left(\frac{4c(\lg n)2^{\lg n}}{\lg n + 1} - c2^{\lg n} - \lg n\right) + n \\
R(2n) &= \frac{cn^2}{\lg n} + \frac{n}{\lg n}\left(\frac{4cn\lg n}{(\lg n) + 1} - cn - \lg n\right) + n \\
R(2n) &= \frac{cn^2}{\lg n} + \frac{4cn^2}{(\lg n) + 1} - \frac{cn^2}{\lg n} - n + n \\
R(2n) &= \frac{c(4n^2)}{\lg 2n} \\
R(2n) &= \frac{c(2n)^2}{\lg(2n)} \\
R(2n) &\in O\left(\frac{n^2}{\lg n}\right)
\end{aligned}
\tag{8}
$$

Since this recurrence is guaranteed to have a complexity higher than ore equal to $T(n)$ we now know that $T(n) \in O(\frac{n^2}{\lg n})$.

Since we now have proven that the complexity of $T(n)$ is both $O(\frac{n^2}{\lg n})$ and $\Omega(\frac{n^2}{\lg n})$, this means that we have proven a tight bound and the complexity is $\Theta(\frac{n^2}{\lg n})$.

3. Suppose I have R sorted files of differing lengths that I concatenate end-to-end to get N entries in total. How would you sort the combined file into order and what is the worst case number of comparisons for your method?

   If we want to take advantage of the sorted files then we must make a **full** pass through the list (until we have found the start of the Rth file – in the case where $R = 1$ this would be without passing through the list at all) and then merging the files with files of a comparable length using a sorting algorithm. Failure to compare to files of a comparable length means that the merges can degrade into insertions – if you have one file of length n and $R - 1$ files of length 1 each element being larger than the previous and you merge the n length file with the length 1 files then you end up passing through the whole n length file for every merge. This degrades to $O(RN)$ – with the worst case number of comparisons being $\frac{N(N-1)}{2}$ in the merges and $\frac{N^2+N+2}{2}$ total including the initial pass through the list and occuring when the files are all unsorted and are concatenated in reverse order.

   A smarter strategy would be to analyse R and decide whether R is large enough to bother passing through the list at all! This removes the worst case complexity down to that of a normal mergesort – $\Theta(N \lg N)$. An even further optimisaion to this would be to decide whether to continue parsing through the list at all every time you find the end of one partition. This decreases the averge case complexity but has no effect on the worst case complexity.

   A further optimisation to help deal with the $\Theta(N^2)$ worst case would be when merging two lists, the longer having length l_1 and the shorter having length l_2, you should check whether $l_2 \lg(l_1) \leq l_1 + l_2$ and if so use a binary merge rather than a linear merge to eliminate the worst case. This should also be done every time prior to doing any comparison in a serial merge – else ie if all but a few elements in one list are smaller than every element in another list then the seemingly equal merge will degrade into a merge which would meet the $l_2 \lg(l_1) \leq l_1 + l_2$ criteria at the end.

   A binary merge starts with two sorted lists and binary inserts the first element in the shorter list into the longer list and repeats each time only considering the subset of the longer list which is greater than the previous element to be inserted.

   If $R = \frac{N}{2}$ then passing through the list once means we no longer have to do the first level of merges – and so save $\frac{N}{2}$ comparisons. So if $R \geq \frac{N}{2}$ we should simply mergesort.

If $R = \frac{N}{4}$ then passing through the list once means we no longer have to do $\frac{N}{2} + \frac{3N}{4}$ comparisons. So in this case passing through the list once is worth it.

If $\frac{N}{4} \leq R \leq \frac{N}{2}$ then the number of comparisons saved by passing through the list once is obtained by adding $\frac{N}{2}$ to the linear interpolation of R between $\frac{N}{2}$ and $\frac{N}{4}$

If R is somewhere between $\frac{N}{2}$ and $\frac{N}{4}$ then the number of comparisons C we save by passing through the list once is given by the formula:

$$\begin{aligned} C &= \frac{N}{2} + 4 \cdot (\frac{1}{2} - \frac{R}{N}) \cdot \frac{3N}{4} \\ C &= \frac{N}{2} + \frac{3N}{2} - 3R \\ C &= 2N - 3R \end{aligned} \tag{9}$$

If $C \geq N - 1$ then we should pass through the list and merge on the sorted files.

$$\begin{aligned} N - 1 &\leq 2N - 3R \\ N &< 2N - 3R \\ 3R &< N \\ R &< \frac{N}{3} \end{aligned} \tag{10}$$

So if $R < \frac{N}{3}$ then we should pass through the list to find the ends of the files and then merge them. Otherwise we perform a normal mergesort.

The worst case now is obviously deciding that R is large enough that the list is essentially totally unsorted and that the best strategy is a mergesort without prior knowledge – otherwise we would parse through the list to find out where the files were and choose a strategy with a lower worst case number of comparisons.

So the worst case of this algorithm is a normal mergesort. When merging two lists of length k, the worst case number of comparisons which must be made are $2 \cdot k - 1$. If we assume for simplicity that N is an exact power of 2 we can do a simple analysis. Let C be the worst case number of comparisons in a mergesort.

$$\begin{aligned} C &= \sum_{k=1}^{\lg N} \frac{(2^k - 1) \cdot N}{2^k} \\ C &= N \cdot \sum k = 1^{\lg N} 1 - 2^{-k} \\ C &= N \cdot (\lg N - \frac{\frac{1}{2}(1 - 2^{-\lg N})}{1 - \frac{1}{2}}) \\ C &= N \cdot (\lg N - 1 + 2^{-\lg N}) \\ C &= N \cdot (\lg N - 1 + \frac{1}{N}) \\ C &= N \lg N - N + 1 \end{aligned} \tag{11}$$

So the worst case number of comparisons that our algorithm takes is $N \lg N - N + 1$. However, this worst-case analysis assumes the worst case for R as well ($R = N$). If we know R then we can do an even tighter analysis. Assume that R is also an exact power of 2. Let C be the worst case number of comparisons if we pass through the list and then merge.

$$
\begin{aligned}
C &= N - 1 + N \cdot \sum_{k=\lg\left(\frac{N}{R}\right)+1}^{\lg N} \frac{(2^k - 1) \cdot N}{2^k} \\
C &= N - 1 + N \cdot \sum_{k=\lg\left(\frac{N}{R}\right)+1}^{\lg N} 1 - 2^{-k} \\
C &= N - 1 + N \cdot \left(\lg N - \lg\left(\frac{N}{R}\right) - \frac{\frac{1}{2}(1 - 2^{-\lg N})}{1 - \frac{1}{2}} + \frac{\frac{1}{2}(1 - 2^{-\lg \frac{N}{R}})}{1 - \frac{1}{2}} \right) \\
C &= N - 1 + N \cdot \left(\lg N - \lg\left(\frac{N}{R}\right) - (1 - 2^{-\lg N}) + (1 - 2^{-\lg \frac{N}{R}}) \right) \\
C &= N - 1 + N \cdot \left(\lg N - \lg\left(\frac{N}{R}\right) - 1 + \frac{1}{N} + 1 - \frac{R}{N} \right) \\
C &= N - 1 + N \cdot \left(\lg N - \lg\left(\frac{N}{R}\right) + \frac{1}{N} - \frac{R}{N} \right) \\
C &= N - 1 + N \cdot \lg N - N \cdot \lg\left(\frac{N}{R}\right) + 1 - R \\
C &= N \lg N - N \cdot \lg\left(\frac{N}{R}\right) + N - R \\
C &= N(\lg N - \lg N + \lg R) + N - R \\
C &= N \lg R + N - R
\end{aligned}
\tag{12}
$$

There is also an algorithm with the same complexity in which you extract the start of all the files, place them into a list, sort the list in $R \lg R - R + 1$ comparisons, then extract the minimum element from the list in 0 comparisons, and then binary-insert the next element from the file which the minimum was in. This insertion takes $\lg R$ comparisons.

Since there are $N - R$ elements not in the list we know that there will be $N - R$ insertions – and each insertion requires $\lg R$ comparisons, there are a further $(N - R) \lg R$ comparisons. The complexity of insertion can decrease if every element from some files are extracted.

We can ensure logarithmic insertion and no need to move every element in the list when inserting into the middle of the list by converting the list into a fully balanced tree (not a r-b tree or balanced tree – but a tree where after every removal the tree is rebalanced to ensure that it is always almost-full, this requires 0 comparisons on the elements but a lot of other computation). We would also keep a pointer to the smallest element in this tree to ensure $\Theta(1)$ removal of the minimum element.

This leads to the formula:

$$
\begin{aligned}
C &= (N - 1) + R \lg R + (N - R) \lg R \\
C &= N - 1 + R \lg R - R + 1 + N \lg R - R \lg R \\
C &= N \lg R + N - R
\end{aligned}
\tag{13}
$$

Which is the same number of comparisons as the previous algorithm.

This algorithm is better when the length of the lists is not uniformly distributed – since the size of R decreases as comparisons are made – however it requires much more additional space to store the elements after they are being merged than the other method. This means that I would in the general case prefer the original algorithm. However, both algorithms have the same worst-case complexity and mergesort is better than both at the same point.

So the worst case number of comparisons, C with either method is:

$$C = \begin{cases} N \lg N - N + 1 & \text{if } R \geq \frac{N}{3} \quad (14) \\ N \lg R + N - R & \text{else} \quad (15) \end{cases}$$

2 Mergesort

1. Mergesort uses divide and conquer to sort a vector (array). The same technique can be used to find the distance between the closest pair of points in a vector of points in a plane:

```
findclosest(2DPoint[] array) {
  mergesort_by_x_coord(array)
  return helper(array)
}

helper(2DPoint[] array) {
  left,right = split(array, array.length / 2)
  split_x = ( max_x_coord(left) + min_x_coord(right) ) / 2

  closest_left = findclosest(left)
  closest_right = findclosest(right)
  m = min(closest_left, closest_right)

  Points2D[] tmp = array.copy_subset(split_x-m <= x_coord <= split_x+m)
  mergesort_by_y_coord(tmp)
  for 2DPoint p in tmp:
    for 2DPoint p2 in tmp such that p2.y < p.y and p2.y+m > p.y:
      m = min(m ,distance(p,p2))

  return m;
}
```

Although there is a nested for loop in the algorith, optimisation can mean that the complexity of the inner for loop is only $\Theta(n)$ – since we are checking an area of side length $2m$ and we know that no two vectors in either the right half of the left half of the square are closer to each other than m, we can conclude that there can be no more than 9 elements in the square (tighter analysis is possible but more complicated and this analysis gives the same complexity).

If there were more than 9 points in the square then m would be smaller and the square would be smaller – which leads to a contradiction. So each point in the inner loop must check no more than 8 other points. Meaning that the inner for loop (if optimised correctly – which I will implicitly assume it is) will do a constant number of comparisons. And so the two for loops have a complexity of $\Theta(n)$.

The recursive call, however does have a mergesort which has a $\Theta(n \lg n)$ complexity. So the complexity of a recursive call is $\Theta(n \lg n)$.

The algorithm calls itself twice on inputs of half the size. This leads to the following recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \lg n) \quad (16)$$

Which has the solution $T(n) \in O(n(\lg n)^2)$.

2. What change would make this $O(n \lg n)$.

   I have two solutions to this – one more serious and one was my first thought and I just wrote it up properly.

   (a) In order to achieve $\Theta(n \lg n)$ practically we have to form the recurrence relation:

   $$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \tag{17}$$

   The algorithm supplied already has two recursive calls on lists of size $\frac{n}{2}$ – so what we must do is convert each recursive call to a $\Theta(n)$ operation. The most obvious unneccessary thing in the algorithm is mergesorting by x at each call – after the first mergesort; the list is already sorted and so there is no requirement to sort by x on each recursive call.

   However, now we still are sorting by y at each call. This makes the complexity beyond $O(n \lg n)$. Note that if we sort by y before calling the algorithm and then pass this array sorted by y then we can create `tmp` without sorting the array in each recursive call. However, this requires finding every element in the array sorted by y in linear time. This can be achieved by using a set – since checking whether a set contains an element is $O(1)$ time. The creation of which is also linear.

   I would like to reiterate that we must also ensure that the way in which we select the points such that "p2.y < p.y and p2.y + m > p.y" is also optimised – however this is easy to do (by starting at the point at position i and moving towards the start of the array until this predicate is no longer met – I assume that this is done.

   Since we now do not need to sort the array at each call, the recursive calls should now be to `helper` rather than `findclosest`.

   So we now have the below algorithm:

```
findclosest(2DPoint[] array) {
  mergesort_by_x_coord(array)
  yarray = array.copy()
  mergesort_by_y_coord(yarray)
  return helper(array, yarray)
}

helper(2DPoint[] xsorted, 2DPoint[] ysorted) {
  left,right = split(array, array.length / 2)
  split_x = ( max_x_coord(left) + min_x_coord(right) ) / 2

  Set<2DPoint> leftset = set(left)
  2DPoint[] yleft = 2DPoint[left.size]
  2DPoint[] yright = 2DPoint[right.size]

  for 2DPoint p in ysorted:
    if p in leftset:
        yleft.append(p)
      else:
        yright.append(p)

  closest_left = helper(left, yleft)
  closest_right = helper(right, yright)
 m = min(closest_left, closest_right)

  Set<2DPoint> xset = set(array.subset(split_x-m <= x_coord <= split_x+m))
```

```
  2DPoint[] tmp = 2DPoint[xset.size]
  for 2DPoint p in ysorted:
        if p in xset:
          tmp.append(p);

  for 2DPoint p in tmp:
    for 2DPoint p2 in tmp such that p2.y < p.y and p2.y+m > p.y:
      m = min(m ,distance(p,p2));

  return m;
}
```

Which has the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \tag{18}$$

Which has the solution $T(n) \in O(n \lg n)$ – and so the algorithm is now $\Theta(n \lg n)$ as required.

(b) My second "solution":

Remove all optimisation and make it $\Theta(\lg n)$ by fully parallelising it.

On a fully parallelised machine we could use almost any algorithm and achieve $O(n \lg n)$. I would like to emphasise that the question asks us to make this $O(n \lg n)$ and not $\Theta(n \lg n)$. $\Theta(\lg n) \in O(n \lg n)$. The only complexity that is mentioned is *time* complexity – no mention is made of space complexity or *core* complexity – so an algorithm having $\Theta(n^2)$ space complexity and requiring n^2 cores is allowed by the question.

```
findclosest(2DPoint[] array) {
  mindistances = float[array.size];
  // do not initialise floats in the array
  parallel for i = 0 to array.size:
        distancefromp = float[array.size];
        // do not initialise floats in the array
        parallel for j = 0 to array.size:
          if i != j:
                distancefromp[j] = distance(array[i], array[j]);
          else:
                distancefromp[j] = inf;
        mindistances[i] = parallel min(distancefromp);
  mindistance = parallel min(mindistances);
  return mindistance
}
```

This algorithm ensures that the mindistances array is read only and the arrays which are written to are never written to the same place by any two cores – or read from until all cores have finished their parallel for loop. This means that we do not have to handle race conditions and so we can have fully parallel execution on an ideal fully parallelised machine.

Parallel for executes each part of the loop on a different core.

Creation of an array of size n could be $\Theta(n)$ if we initialise values serially – however if we do not initialise the values in the array (since we know we will overwrite every one of them without reading), we can create the arrays in $\Theta(1)$ time. We can execute the nested for loop on n^2 cores in $\Theta(1)$ time. However, finding the

minimum element in an array is $\Theta(\lg n)$. A parallel min recursively does tournament selection – you compare every even numbered element to the odd numbered element to its right and disregard the larger element in that comparison (since the larger element is larger than at least one other element it cannot possibly be the minimum) – at every iteration we halve the size of the array allowing us to find the minimum in $\Theta(\lg n)$.

This solution performs a series of $\Theta(1)$ operations, followed by two $\Theta(\lg n)$ operations and so has an overall time complexity of $\Theta(\lg n)$.

However, this solution (while asymptotically optimal and fun) is not practical for the overwhelming majority of real-world applications.

3 Quicksort

1. Quicksort must select an item to use as the pivot. If the first item in the input array of length n is used as the pivot, describe 4 input orderings that lead to $O(n^2)$ time complexity (e.g. "sorted order" is one, find 4 others).

   There is a generalisation to this. The complexity of quicksort is $O(n^2)$ in all cases where the distance of the pivot from either end is constant (or within a constant bound).

   Some examples are:

   - The first element in the array is always the largest element in the array.
   - The first element in the array is always the 3^{rd} smallest element.
   - The first element in the array is always within the 9 largest elements in the array. element in the array.
   - The first element in the array is always within the 10000 smallest elements in the array (this is true in the general case and for large arrays).

2. If all orderings of the input array are equally likely, is there any benefit to choosing the pivot randomly?

   No.

   For obvious reasons if any order is equally likely then any element in the array is equally likely to be in any location in the array – and so selection of any arbitrary index (1st element, 8th element, last element) is a suitable choice for the pivot. Any randomly selected element is no more or less likely to be a good choice for the pivot than any non-random selection.

3. It is suggested that quicksort should split the array into 3 regions: items strictly less than the chosen pivot value, items equal to the pivot value, and items strictly greater than the pivot value. Is there any merit to this idea?

   There are both merits and drawbacks.

   The main advantage is that this quicksort deals far better in the case where the number of distinct items is significantly smaller than the number of items.

   Take the example where you are sorting everyone on the earth by age. There are 8b people but only 120 different ages. If you have a third class which includes "equal to" the pivot then the maximum number of passes that you need to do is 120. However if you do not have a separate class for "elements equal to the pivot" then you will have to make billions of passes. So in this example including a class which is equal to the pivot makes the sort significantly better.

   In some situations, for example if you were sorting floating point numbers, then you would almost never come across an element which was equal to the pivot – and so

engineering your algorithm to enable this option is wasteful and only slows it down since we must have additional cases and a further array/method of holding the values equal to the pivot.

Also; the requirement to have three classes means that the most efficient in-place algorithms now no longer work and you will now require many more memory accesses to place all the positions correctly.

Overall I would recommend having a separate class for elements equal to the pivot only on larger lists when you know the number of distinct elements is very small compared to the total number of elements (however in many cases like that could argue that a bucketsort or radixsort would be more suitable).

4. We have an array of distinct items (no duplicates) and our pivot is to be the median of 3 items chosen randomly from the array. Does this improve the worst case time complexity, in asymptotic or real terms?

   This does not decrease the worst case complexity. It is still possible for the pivot to be the second smallest element in the list every time. This means that the pivot would decrease the size of the list linearly with each pass leading to the recurrence relation:

$$T(n) = T(n-2) + k \cdot n \tag{19}$$

   The solution to this recurrence relation is $T(n) \in \Theta(n^2)$. So this strategy does not reduce the asymptotic worst case complexity.

   However, in reality the liklihood of the worst case complexity has been decreased – and now know that the pivot will always decrease the list by at least two rather than at least one. So this halves the maximum number of iterations which will have to be done. However: we now have to make two additional comparisons to find the median element as well as randomly generating three numbers.

   The number of comparisons to pivot on a list of length n is $n-1$. So the total number of comparisons C which we have to make in the worst case for quicksort is:

$$\begin{aligned} C &= \sum_{k=1}^{n} k - 1 \\ C &= n \cdot \frac{n-1}{2} \end{aligned} \tag{20}$$

   However under the new scheme the worst case number of comparisons C we will have to make is given by:

$$\begin{aligned} C &= \sum_{k=1}^{\frac{n}{2}} k - 1 + 3 \\ C &= \sum_{k=1}^{\frac{n}{2}} k + 2 \\ C &= \frac{n}{2} \cdot \frac{n+2}{4} + 2n \\ C &= \frac{n^2}{8} + \frac{9n}{4} \\ C &= n \cdot \frac{n+9}{8} \end{aligned} \tag{21}$$

   This is better than the normal worst-case number of comparisons. So althought the asymptotic worst-case complexity has decreased: both the probability of achieving a near-$O(n^2)$ complexity and the number of comparisons made in the case that such a complexity is achieved are decreased. Given that the worst-case of quicksort is already

unlikely with randomised pivot selection – one may ask why bother with this additional optimisation.

5. We have an array of distinct items (no duplicates) and, to choose our pivot, we find the median of items 1–5, 6–10, 11–15, etc, and then use the median of those medians as our pivot. Does this improve the asymptotic worst case time complexity? (You may assume that the median of medians of n items can be found in $O(n)$ time.)

   The pivot we select is the median of medians. So there must be at least half of the medians which are smaller than the pivot. For simplicity of analysis assume that n is an exact multiple of 5. In each of the sets which those medians were selected from: 3 of the elements were less than or equal to the median. So we have guarantees that $\frac{3}{5} \cdot \frac{n}{2} = \frac{3n}{10}$ elements are less than the pivot. So the worst-case pivot is now that we divide the list into two sublists of size $\frac{3n}{10}$ and $\frac{7n}{10}$. This leads to the worst-case recurrence relation:

   $$T(n) = T\left(\frac{3n}{10}\right) + T\left(\frac{7n}{10}\right) + k \cdot n \tag{22}$$

   Substitute $T(n) = cn \lg(n)$.

   $$\begin{aligned}
   T(n) &= T\left(\frac{3n}{10}\right) + T\left(\frac{7n}{10}\right) + k \cdot n \\
   cn \lg n &= \frac{3cn}{10} \lg\left(\frac{3n}{10}\right) + \frac{7cn}{10} \lg\left(\frac{7n}{10}\right) + k \cdot n \\
   cn \lg n &= \frac{3cn}{10} \lg n + \frac{3cn}{10} \lg 3 - \frac{3cn}{10} \lg 10 + \frac{7cn}{10} \lg n + \frac{7cn}{10} \lg 7 - \frac{7cn}{10} \lg 10 + k \cdot n \\
   cn \lg n &= \frac{10cn}{10} \lg n + cn\left(\frac{3}{10} \lg 3 + \frac{7}{10} \lg 7 - \lg 10\right) + k \cdot n \\
   0 &= cn\left(\frac{3}{10} \lg 3 + \frac{7}{10} \lg 7 - \lg 10\right) + k \cdot n \\
   0 &= -c + \frac{k}{\lg 10 - \frac{3}{10} \lg 3 - \frac{7}{10} \lg 7} \\
   c &= \frac{k}{\lg 10 - \frac{3}{10} \lg 3 - \frac{7}{10} \lg 7}
   \end{aligned} \tag{23}$$

   Since $\lg 10 - \frac{3}{10} \lg 3 - \frac{7}{10} \lg 7 = 0.88\cdots > 0$, we can conclude that there exists a positive real constant c such that $T(n) = c \cdot n \cdot \lg n$. This implies that $T(n) \in O(n \lg n)$.

   So the asymptotic worst case has now been improved to $O(n \lg n)$.

6. Show how find this "median of medians" of n items in $O(n)$ time.

   Obviously the first step to doing this is to find the medians. This requires $O(n)$ operations since it involves splitting the list into subarrays of length 5 and selecting the middle element in them. I will assume the use of any reasonable strategy to doing this – it will take constant time to sort each subarray since their length (5) is independent of n. We do this $\frac{n}{5}$ times. So working out the medians takes $O(n)$ operations.

   The next (and more complicated) step is to find the median of these medians. The simplest algorithm to doing this is quickselect – an adaptation of quicksort where an element is selected and pivoted around – except only the side which contains the i^{th} element is traversed (when searching for the i^{th} element). This has an average case of $O(n)$. However, like quicksort; quickselect also has a worst case complexity of $O(n^2)$. We can employ the strategy used in the previous question to remove this though – selecting a "median of medians" to pivot on.

Using the results from the last question: the worst case here is where the element we are searching for is always in the larger subdivision. This means that in the worst case the recurrence relation is:

$$T(n) = T\left(\frac{7n}{10}\right) + k \cdot n \tag{24}$$

Using the master method we can see that $\log_{\frac{10}{7}} 1 = 0$ and so the term is dominated by the $k \cdot n$ constant. This means that quickselect is $O(n)$.

So we recursively call the algorithm to find the median of medians and then find a list of medians which we need to select the median from. We repeatedly call this until the length of the list of medians is less than 5. At which point we use quickselect to unwind the stack and pivot around finding the median element in guaranteed $O(n)$ time.

The recursive calls to the yields the recurrence relation:

$$T(n) = T\left(\frac{n}{5}\right) + k \cdot n \tag{25}$$

Again using the master method we see that $\log_5 1 = 0 < 1$ and so the recurrence relation is dominated by $k \cdot n$. So $T(n) \in \Theta(n)$. This proves that you can select the "median of medians" in guaranteed $O(n)$ time by using the quickselect algorithm using the median of medians as the pivot.

Here is a python implementation of a $\Theta(n)$ median of medians selection algorithm and a supporting $\Theta(n)$ quickselect algorithm.

```
def median_of_medians(arr: list):
    if len(arr) <= 5:
        return sorted(arr)[len(arr) // 2]
    else:
        medians = []
        for i in range((len(arr) + 4) // 5):
            sublist = arr[5 * i: 5 * (i + 1)]
            medians.append(sorted(sublist)[len(sublist) // 2])
    if len(medians) <= 5:
        return sorted(medians)[len(medians) // 2]
    else:
        return quickselect(medians, len(medians) // 2)

def quickselect(arr: list, target: int):
    left = []
    right = []
    pivot = median_of_medians(arr)
    looking_for_pivot = True
    for i in range(len(arr)):
        v = arr[i]
        if looking_for_pivot and v == pivot:
            looking_for_pivot = False
        else:
            if v <= pivot:
                left.append(v)
            else:
                right.append(v)
    if len(left) > target:
        return quickselect(left, target)
    elif len(left) == target:
        return pivot
    else:
        return quickselect(right, target - len(left) - 1)
```