1. We saw how to compute the median of \( n \) elements in \( \Theta(n) \) time, using groups of 5. Show what happens if we form groups of 3 instead. You may assume that dividing by 3 still gives you an integer. Next, figure out what happens if we use groups of size \( k \), where \( k \) represents a constant greater than 5. Finally show what happens if we form 5 groups of size \( \frac{n}{5} \), or \( \sqrt{n} \) groups of size \( \sqrt{n} \).

**Answer:**

We get \( \frac{n}{3} \) groups of 3 elements. We will spend \( O(n) \) time forming all the groups and computing the median of each, by brute force. So, \( \frac{n}{3} \) elements will be group medians. Let’s call them “leaders” to avoid overusing the word “median”. Let \( x \) be the median of all leaders. We spend \( T(\frac{n}{3}) \) time computing \( x \).

Suppose, without loss of generality, that \( x \) has a higher rank than what we are looking for, in the original array. Then we will recurse on the set \( S \) of elements that are smaller than \( x \). What is the size of \( S \)? It will definitely contain half of the group leaders, i.e., the \( \frac{n}{6} \) elements that we visualize as being placed to the left of \( x \) in the middle row of the dot diagram. \( S \) will also definitely contain at least one more element from each of the groups (columns) corresponding to this subset of group leaders. Specifically, \( S \) will contain the elements visualized as being below and/or to the left of \( x \). \( S \) will definitely not contain any element above and/or to the right of \( x \). The rest of the elements in the diagram could go either way, so we will assume that the worst situation occurs, which is that every element in the leftmost \( \frac{n}{2} \) columns will be smaller than \( x \), and every element in the entire row below \( x \) will be smaller than \( x \) as well. Summing things up, we get at most \( \frac{2n}{3} \) elements that can be smaller than \( x \).

The recurrence, corresponding to obtaining the worst case split every time we compute a median, is \( T(n) \leq T(\frac{n}{3}) + T(\frac{4n}{3}) + \Theta(n) \). The first term is a recursive call to compute the median of group leaders, and the second term corresponds to the recursive call that we make after using \( x \) as a pivot, discovering that it is not the rank we wanted, and discarding some data. This recurrence solves to \( \Theta(n \log n) \). To see this, you can draw the recursion tree, and find that every level corresponds to \( \Theta(n) \) work. Then compare the tree to a shorter fully balanced tree and a larger fully balanced tree, respectively corresponding to the shortest and longest path in the actual tree. Both of the balanced trees will have logarithmic depth, just with a different log base. We have seen an example of this in class. All we care about is that this is \( \Omega(n \log n) \), so it’s worse than using groups of 5.

The interesting phenomenon here is that \( x \) is still partitioning the data quite evenly. We know that at least a third of the elements will be in the small group, and at least a third will be in the large group (always by comparing ranks against the rank of \( x \)). In other words, \( x \) is guaranteed to have a rank somewhere in the “middle third”. It’s a very good choice! In fact the worst case of this partition will not be as bad as the worst case when we used groups of 5. What hurts us is that to compute \( x \) we recursively call the median function on a large group; a third of the elements (instead of a fifth). To see an even more extreme effect of this, consider the unrealistic scenario of forming \( n \) groups of size 1. Finding the “leader” in each group is trivial (zero work required).
After computing the median of leaders, \( x \), we create a perfectly balanced partition (at least half of the data can be discarded). However the recursive call to find \( x \) involves doing exactly what we had to do in the beginning.

For groups of constant size \( k \): as \( k \) increases, the worst case partition quickly approaches a \( \frac{n}{2} \) vs \( \frac{3n}{4} \) split. Basically, for any \( k \), we know that we will discard all the data in one quadrant of the dot grid, including data in the middle row and column, if either dimension of the grid is odd. So for at least \( k/2 \) columns we throw out at least \( n/2k \) elements, meaning we discard at least \( n/4 \) overall. As mentioned above, for small \( k \), or very large \( k \), we discard even more than a quarter. So, compared to our analysis with groups of 5, things aren’t really any different, regarding the recursive call that is made when \( x \) doesn’t have the rank we are looking for. That is, we still have to pay \( T(\frac{3n}{4}) \) for this, in the worst case. One thing that does change is the parameter in the recursive call for finding the “median of leaders”. There are only \( \frac{n}{k} \) groups, so it becomes \( T(\frac{n}{k}) \). That’s good news; it’s getting smaller. We still have \( \Theta(n) \) work to do, to partition our array into groups of size \( k \), to partition all data with respect to the pivot, \( x \), and even to find the median in each group (column). This last task becomes increasingly costly as \( k \) grows, but as \( k \) remains constant, this still takes constant time per column, even by sorting or brute force. So our recurrence becomes \( T(n) = T(\frac{3n}{4}) + T(\frac{n}{k}) + \Theta(n) \). This still solves to \( \Theta(n) \) for \( k \geq 5 \). Either use substitution, or notice that for \( k > 5 \) things are strictly better than for \( k = 5 \). However, the non-recursive \( \Theta(n) \) term has a hidden constant, and in more detail is proportional to something like \( \Theta(n) + k \log k \cdot \frac{n}{k} \). We get the \( \Theta(n) \) because of partitioning, but then there is the influence of the task of computing the leader (median) of each of our groups of size \( k \). Doing this non-recursively, it takes \( O(k \log k) \) time (by sorting), and there are only \( n/k \) groups to deal with. So what’s really going on is:

\[
T(n) = T(\frac{3n}{4}) + T(\frac{n}{k}) + O(n \log k).
\]

That’s why we can’t allow \( k \) to grow to more than a constant.

Setting \( k = \frac{n}{5} \), i.e., forming only 5 groups, we get the recurrence \( T(n) = T(5) + T(\frac{3n}{4}) + \Theta(n) + f(n) \). The first term corresponds to recursively finding the median of leaders (i.e., \( x \)), and is just \( O(1) \) because it’s a base case. The \( \Theta(n) \) term corresponds to partitioning after finding \( x \), and the last term corresponds to the time it takes to find the leader (median) of each group of size \( n/5 \). Here we have a choice. We could sort, so \( f(n) = \Theta(n \log n) \), or we could recurse, so \( f(n) = 5T(\frac{n}{5}) \). In the former case, we’ve already done \( \Theta(n \log n) \) work without even recursing, so that’s bad news. On the other hand if we recurse for \( f(n) \), we get \( T(n) = \Theta(n) + T(\frac{3n}{4}) + 5T(\frac{n}{5}) \). Even if we ignore the \( T(\frac{3n}{4}) \) we get a standard divide-and-conquer which solves to \( \Theta(n \log n) \). Putting back the ignored component only makes things worse.

With \( \sqrt{n} \) groups, we have \( T(n) = \Theta(n) + T(\sqrt{n}) + T(\frac{3n}{4}) + \sqrt{n} \cdot f(n) \). We still perform linear work to partition after finding \( x \), and now we take \( T(\sqrt{n}) \) time to compute the median of group leaders (i.e., \( x \)), given that we already have the leaders in a new array. Here, \( f(n) \) corresponds to the time to compute one group leader. If we sort, that’s \( \Theta(\sqrt{n} \log n) \), but since we have to do this \( \sqrt{n} \) times, we have a bad overall time complexity. So we could try to get each leader recursively, which makes \( f(n) = T(\sqrt{n}) \). So \( T(n) \geq \Theta(n) + \sqrt{n} \cdot T(\sqrt{n}) + T(\frac{3n}{4}) \). If we ignore the last term and draw a recursion
tree, we see that we get a linear amount of work at every level. The number of levels is more than any constant, i.e., it is $\omega(1)$, because we cannot recursively take a constant number of square roots and get from $n$ down to a (constant) base case. So the total amount of work is super-linear ($\omega(n)$). In fact a brief calculation can show that there are $\log \log n$ levels. In any case, this analysis has already ignored the $T(3n^4)$. If we factor that in, we can use substitution to prove that $T(n) = \Omega(n \log n)$. The point is that we do not get a linear-time algorithm. $T(n) = \omega(n)$.

2. How fast can you partition a set of $2n$ elements into two groups of size $n$ so that the difference of the respective sums of elements is maximized? I’m looking for worst-case analysis, as usual.

**Answer:**
The difference of the sums is maximized by having the $n$ smallest elements in one group. So run the selection algorithm to find the element with rank $n$, then use it as a pivot to partition. This takes $O(n)$ time.

3. Let $A$ and $B$ be arrays, each respectively containing $n$ numbers in sorted order. Show how to compute the median of all the numbers in $A$ and $B$, in worst-case logarithmic time.

**Answer:**
First I go over one solution and some of it’s variants. At the end there is a somewhat different solution.

The first and probably most intuitive solution is simple, at least at a high level. There are some annoying techniquealities that are mentioned further down. The main idea is to compare the median of $A$ and the median of $B$, both found by indexing in constant time because the arrays are sorted. If the two medians match, i.e., $\text{med}(A)=\text{med}(B)$, then they simultaneously split both arrays so we already have the median of the union. Otherwise wlog\footnote{without loss of generality} assume that $\text{med}(A) > \text{med}(B)$. In other words if we were to place $\text{med}(A)$ in $B$ it would land in the right half of $B$. Thus, in $A \cup B$, fewer than $n$ elements are larger than $\text{med}(A)$, which means it and all elements in $A$ that are to its right are not candidates. Symmetrically all elements to the left of $\text{med}(B)$ in $B$ are not candidates. So we remove the smallest $n/2$ elements from $B$ and largest $n/2$ elements from $A$. We are left with two sorted arrays of size $n/2$, and recurse. Everything so far took constant time, so we will get $O(\log n)$ time overall, by geometric series.

Now for the technicalities. If $n$ is even, each median is in fact formed by a pair of elements. To deal with this, we can look at all four median elements (2 from $A$, 2 from
B), and figure out what their order is. Define \( \text{med1}_A \leq \text{med2}_A \) and \( \text{med1}_B \leq \text{med2}_B \). Assume \text{wlog} that \( \text{med1}_A < \text{med1}_B \). Then we have one of the following situations:
1) \( \text{med1}_A \leq \text{med2}_A \leq \text{med1}_B \leq \text{med2}_B \)
2) \( \text{med1}_A \leq \text{med1}_B \leq \text{med2}_A \leq \text{med2}_B \)
3) \( \text{med1}_A \leq \text{med1}_B \leq \text{med2}_B \leq \text{med2}_A \)
Determining which case we have takes \( O(1) \) time.
In case 3, we know that the average of \( \text{med1}_B \) and \( \text{med2}_B \) splits all input from A and B, so it’s the answer. In case 2, we know that the average of \( \text{med1}_B \) and \( \text{med2}_A \) splits all input from A and B, so this is the answer. In case 1 we don’t get an immediate answer. But we can discard \( \text{med1}_A \) and everything smaller in A, as well as \( \text{med2}_B \) and everything larger in B. So we discard half of the input.
For odd \( n \), as mentioned, if \( \text{med}(A) = \text{med}(B) \), we’re done. So, \text{wlog} suppose that \( \text{med}(A) < \text{med}(B) \). The situation is similar to the case where \( n \) is even. Let \( X_A \) be the element that ranks just above \( \text{med}(A) \) in A. Let \( X_B \) be the element that ranks just below \( \text{med}(B) \) in B. If \( X_B \leq \text{med}(A) \), then \( \text{med}(A) \) forms part of the median pair, and the other part is the min of \( X_A \) and \( \text{med}(B) \). Otherwise, \( \text{med}(A) < X_B \). In this case as long as \( X_B \leq X_A \), then \( X_B \) forms one part of the answer, along with the min of \( X_A \) and \( \text{med}(B) \). The only case requiring recursion is when \( X_A < X_B \). Then we must recurse, because we don’t know if other elements from A or B fall between these two elements. In other words we can’t guarantee that they form the median pair. However we can discard \( \text{med}(A) \) and all elements in A that are smaller, as well as \( \text{med}(B) \) and all elements in B that are larger.

The point is that you just need to figure out a subset of the data that is safe to discard and has \( \Theta(n) \) size. So in fact you could not deal with any of the case analysis above, play it safe and just discard a quarter of the data from each array each time.

Note that this algorithm also works (with minor modifications), if we assume that the median of \( 2n \) elements is the element with rank \( n \) (i.e., not an average of two numbers).

To use recursion, you have to make sure that every time you call your function, the input consists of two arrays of \textit{equal} size. Otherwise you are relying on solving a more general problem and you need to discuss this.

\textbf{Another nice solution} (explained at high level) is to start by testing \( \text{med}(A) \), and, as above, conclude that if it has the same value as \( \text{med}(B) \), we’re done. Otherwise if \text{wlog}, \( \text{med}(A) > \text{med}(B) \), then \( \text{med}(A) \) is not the correct value, nor is anything larger in A. So test the element with rank \( n/4 \) in A. Let it have value \( x \). If it were the median then there would have to be \( 3n/4 \) elements smaller than \( x \) in B. Meaning, we check if \( B[3n/4] = x \) to confirm. If the two are not equal, and \( x \) is larger, then again we can discard \( A[n/4] \) and higher values. But if \( x \) is smaller, then we discard \( A[n/4] \) and smaller values. Thus we binary search on A, trying to find an index \( y \) in A that has a value \( x = A[y] \) such that \( B[n - y] = x \) as well.