# Assignment 09

Alex Clemmer
Student number: u0458675

## 1

Our algorithm is going to have a recurrence relation of $2T(n/2) + O(n)$, because there will be $n/2$ steps ($n$ here being the size of the array), and each step will do work proportional to its size (here, confusingly, also called $n$). Also, the work for each step is done effectively twice, so $a = 2$.

The critical fact is essentially given to us by the assignment: If both halves of some array $A$ have dominant elements (call them $x$ and $y$, respectively), then according to the pigeonhole principle one or the other must be the dominant element of $A$, if there is one at all. Thus we need only tally how many times each occurs in the array and see if either is a majority.

On the other hand, if there are no dominating elements in either half, then none of the elements can comprise more than half the array. For instance, if we have an array $A = \{a, b, a, d\}$, where each half has no dominating term, even the most common term, $a$, can only comprise half $|A|$.

Thus, our algorithm begins by recursively splitting the array. This gives us log $n$ levels, such that each level is comprised of subarrays whose length is at most $n/2$. Do this until the maximum length of the subarray is 2. For example, the recursion pattern if you have an array:

$$A = \{a, b, c, d, e, f, g, h\} \Rightarrow \{\{a, b, c, d\}, \{e, f, g, h\}\} \Rightarrow \{\{a, b\}, \{c, d\}, \{e, f\}, \{g, h\}\}$$

For each of these resultant pairs, we check to see if there is a dominant term. If there is, then we step up one recursive level (*i.e.*, in this case the maximum subarray length would be 4, with each pair consisting of 2 pairs of subarrays with length 2 from the level below, which is where the $a = 2$ comes from), and we check to see if this element is the dominant element for this new, larger subarray.

For example, given an array $B = \{a, b, a, d, a\}$, we would start by noting that there is no dominant pair in $\{a, b\}$ or $\{a, d\}$, but that the final element, $\{a\}$ is indeed dominant. Thus when we merge $\{a, b\}$ and $\{a, d\}$, there is no dominant element, but when we merge them with $\{a\}$, we note that there is.

All this given, and considering the recurrence relation from before, we can see that $d = 2, a = 2$, and $1 = \log_2 2$, which means that our runtime is

$$O(n^1 \log\, n) \tag{1}$$

## 2

We can use the last problem to complete this problem. Again, since this is a fully-recursive divide-and-conquer algorithm, and because our base case is given to us, the recurrence so far can be described as $T(n) = aT(n/2) + f(x)$.

We recursively partition the array $A$ as we did above until the maximum subarray length is 2. At this point, we perform one comparison and act accordingly: we throw both away if they're the same, and we keep one if they're the same. If there's one element, that obviously just gets passed on.

This gets repeated at every level of recursion until we evaluate the top level, at which point we have either no elements or one element. There is no dominant term in the former, and the dominant term is the left-over in the latter.

Again, the amount of work done per step is best described as $T(n) = aT(n/2)$; here $a = 2$ because there are 2 partitions per step. The only remaining term in the recurrence is the amount of work it takes to join steps together, which is asymptotically constant. Thus, $T(n) = 2T(n/2) + O(1)$. Since here $d = 0$, the Master Theorem gives us $0 < \log\,_2 2$, which means our runtime is:

$$O(n^{\log_2 2}) = O(n) \tag{2}$$