COMP3027 ASM 1

490399992

March 2021

Task 1

Describe your algorithm

My algorithm uses a divide and conquer approach. The divide step involves dividing the array into two equal halves. We compare the first number in the array, $S[0]$, to the middle number in our array, $S[n/2]$. If $S[0] > S[n/2]$ we recursively search for the largest number in the subarray $S[0...n/2]$. Otherwise we recursively search using the subarray $S[n/2...n]$.

The base case of this algorithm is when $n = 1$ or $n = 2$. If $n = 1$, our merge step involves returning the only number in the array as the largest number. If $n = 2$, our merge step involves returning the largest out of the two numbers as the largest number.

Prove correctness

To prove that the algorithm always returns a potential answer, we can show that the algorithm will always arrive at one of the two base cases, as each base case cannot be further subdivided into smaller subarrays. When $n = 1$, the array cannot be divided any further as n must be an integer. When $n = 2$, because the algorithm always includes the middle number as well as the start or end number in the subarray, recursing on this array will still result in $n = 2$. For $n > 2$, we are able to split the problem into smaller subproblems. Therefore, as each base case returns an answer, and the algorithm will eventually reach a base case, the algorithm is guaranteed to return a possible answer.

We can prove the correctness of the base cases intuitively.

We can prove that the algorithm will always arrive at the correct answer through proof by cases. Take the index of the largest number as i_L.

Firstly, from the definition of the problem we have

$$(n-1) \bmod n = (x + i_L) \bmod n$$

From here, we can realise that

$$\begin{aligned}(x + (i_L + 1)) \bmod n &= (n - 1 + 1) \bmod n \\ &= n \bmod n \\ &= 0\end{aligned}$$

This shows that the index directly after that of the largest number, $i_L + 1$, is the smallest number in the array. In the case where $x = 0$, $i_L + 1 = n \bmod n = 0$. We can show in a similar fashion that all numbers at indices smaller than i_L are strictly decreasing, and all numbers at indices greater than $i_L + 1$ are strictly increasing.

We can also show that

$$(x + 0) \bmod n = x \bmod n$$

and

$$(x + (n-1)) \bmod n = (x-1) \bmod n$$

This means that the element at the start of the array, $S[0]$, will always be larger than the element at the end of the array, $S[n-1]$, unless $x = 0$, in which case $S[0]$ is the smallest number and $S[n-1]$ is the largest number.

The first case to consider is when $i_L < n/2$, meaning that the largest number is in the left half of the array. In this case, $x \neq 0$ as we know $S[n-1]$ is not the largest number. Hence, $S[n/2] < S[n-1]$, as all numbers after $S[i_L]$ are strictly increasing. As $S[0] > S[n-1] > S[n/2]$, we know that $S[0] < S[n/2]$ when $i_L < n/2$.

The second case is when $i_L > n/2$, meaning that the largest number is in the right half of the array. In this case, $S[n/2] > S[0]$ as all numbers to the left of $S[i_L]$ are strictly decreasing.

We can see that the algorithm satisfies both cases, as it will recurse on the left half when $S[0] > S[n/2]$ and the right half when $S[n/2] > S[0]$.

The final case is when $i_L = n/2$. In this case, as $S[0] < S[n/2]$, the algorithm will recurse on the left half. Because the algorithm always includes $S[n/2]$ in the subproblem, $S[i_L]$ will remain in the array.

As the algorithm will always select the correct subproblem for all cases of i_L in relation to $n/2$, we can prove its correctness.

Prove time complexity

The divide and merge steps of the algorithm are bounded by $O(1)$ time, as we do a constant number of operations each time. Similarly, the base cases also run in constant time. The size of each subproblem is $n/2$ as we split the array in half. As we only recurse on one half, we can write the recurrence relation as

$$T(n) = T(n/2) + O(1)$$

Using the master theorem, this evaluates to $O(logn)$, therefore the algorithm is bounded by $O(logn)$ time.

Task 2

Describe your algorithm

First, my algorithm combines the center points of all the squares, with the set of all points, and sorting them by their x coordinate. We then divide the problem by splitting the combined set so that there are an equal number of points on the left and right.

On each half, we recursively determine the square containing the largest number of points. To merge these two halves together, the algorithm looks for the square with the most points that is intersected by the line splitting the points in half. The algorithm then compares these three squares and returns the square with the most points as the solution. In the situation where there is no square within the left, right or middle, we can return a null or 0 value as appropriate to show that there is no square within this problem.

The base case for this algorithm is when there are three points or less in our subproblem. For the base case, the algorithm will split the set of combined points into a separate set of centers, $C\{c_1, c_2, ...\}$ and a set of points, $P\{p_1, p_2, ...\}$, and sort them by decreasing y coordinates through brute force. In the case where $|C| > 0$, we can also through brute force determine if any points in P lie within any of the squares represented by C. If we detemine a point lies within a square, we can increment the square's point counter, and delete the point from the set, to avoid recounting the point in the future. Afterwards, we can also determine which square (if any) currently has the most points.

We can define line m as the vertical line midway between the rightmost point on the left half and the leftmost point on the right half. As our goal with the merge step is to find the square with the most points intersected by m, our first step is to delete any points in P_{left} and P_{right} whose x coordinates are further than L from m. We also delete any points in C_{left} and C_{right} that are further than $L/2$ from m. This is to guarantee that all squares that we consider are intersected by m.

The next step is to merge P_{left} with P_{right} and C_{left} with C_{right} into P and C, ensuring that they remain sorted by y coordinate. This can be done in a similar fashion to the merge step in merge sort. From here, using two pointers we iterate through P and C.

For each pair of point, p, and center, c, we first check if $y_p > y_c + L/2$. If so, we know that the point is too far up to be contained in the square represented by c, and so we move to the next point. If $y_p < y_c - L/2$, we know that the point is too far down to be contained in the current square, and so we move to the next center. Otherwise, we check if the p is within the square represented by c, and increment the square's point counter and delete the point if so, afterwards moving to the next point.

After iterating through P and C fully, we take the square with the largest number of points, and compare it to the square returned by the left subproblem, and the right subproblem. We return the largest out of these three squares as the square with the most points within the current problem.

Prove correctness

We can prove that the algorithm always returns a valid answer as the recursive step will always either return the largest left square, the largest right square, the largest middle square, or no square (in the case where there are no squares in the problem). As each of these are a valid result, the final recursive step will always return a valid result.

We can show that the algorithm always selects the square with the most points through an inductive proof.

Firstly, we can prove the correctness of the base cases. The base case will always return the square with the most points as through the brute force method it will compare all combinations of points, and will account for all center points. It will also avoid points being recounted later on through deleting any points that are shown to be inside a square. In the case where there is a tie between squares (for example, if the base case contains three center points), it does not matter which square you return as the largest as it will not affect the final result.

Second, we assume that the algorithm works for the set of size $n/2$. Using this, we can prove that the algorithm works for a set of size n. We know from our inductive hypothesis that the square returned from the left subproblem is the square with the most points on the left of m, and similarly for the square returned by the right subproblem.

We are able to prove the correctness of finding the square with the most points that is intersected by m by proving that the iteration through P and C assigns all points that are contained in a square to their correct squares. Once we do so, finding the square with the most points in C is trivial.

We can show that all center points in C represent squares that are intersected by m, as the worst case scenario is where a square borders m, with $|x_c - x_m| = L/2$, and the algorithm will remove all points further away than $L/2$. Therefore, we will not be able to allocate points to squares not intersected by m, as that could potentially change the outcome of the left and right subproblems.

To prove that all points are assigned correctly, we can do proof by cases. The first case is where point p is outside of the range of the square represented by c. If $y_p > y_c + L/2$, p is located above the square, so we can guarantee that the point will not be contained in the square represented by c. We can also guarantee that it will not be contained in any squares after c, as C is ordered by decreasing y coordinate. Therefore, we are able to move to the next point in P. Similarly, if $y_p < y_c - L/2$, p is located below the square, guaranteeing that there are no more points in P contained within the square, as P is sorted by decreasing y coordinate. Therefore, we are able to move to the next center point in C.

The second case to consider is when p is within the range of the square represented by c. In this case, we can check if p is contained within the square. If it is, we are able to remove it as a point cannot be located within more than one square. If it is not in the square, we can guarantee that it will not be located within another square in C, because squares cannot overlap and each square shares an intersecting line on the y axis. Therefore, we are able to concretely confirm whether or not the point lies within any of the squares in C, and are able move to the next point in P.

As all cases of p in relation to c are considered, we can prove that the iteration through P and C performed by the algorithm will correctly assign all points in P to squares in C. Therefore, we can prove that the merge step will produce a correct value for the square with the most points intersected by m.

As the algorithm selects the square with the most points out of our three options, we can prove that the algorithm will function correctly for a set of size n under the inductive hypothesis. Therefore we can prove using induction that the algorithm is correct for all sets of points and squares.

Prove time complexity

The initial sorting of the combined point set by x coordinate will take $O(nlogn)$ time, using a method such as merge sort.

For each subproblem, we can show that the divide and merge steps are bounded by $O(n)$. The base cases will run in $O(1)$ time. The deletion of points further than L from the middle line takes $O(n)$ time as it is a single iteration through the set of combined points. As $\{C_{left}, P_{left}\}$ and $\{C_{right}, P_{right}\}$ are already sorted by y coordinate, merging them will take $O(n)$ time as similar to merging in merge sort it involves a single iteration through each of the sets. As we only perform a single iteration through C and P to allocate points to squares, this step will also run in $O(n)$ time. Additionally, finding the square with the most points in the middle section involves a single iteration through C and also takes $O(n)$ time. Overall, the divide and merge steps is bounded by $O(n)$ time.

As we divide our problem into two equal subproblems, and recurse on both, the recurrence relation can be written as

$$T(n) = 2T(n/2) + O(n)$$

Using the master theorem, this evaluates to $O(nlogn)$, therefore overall the algorithm is bounded by $O(nlogn)$ time.