Homework 2 Solution

Yufan Zhou

1.

When partitioning elements into groups of size 6, 7, 9, 11, the $O(n)$-time algorithm can be achieved, while it can not be achieved when partitioning elements in to groups of size 3 and 4.

First, consider we divide the elements into groups that number of elements in each group is an odd number λ, the selection algorithm will be:

$Select(A, p, r, i)$:

(1). If $p = r$: return $A[p]$

(2). For each group G_i, find the median x_i

(3). For the collection of all the median numbers $M = [x_1, ..., x_{(n/\lambda)}]$, call $x = Select(M, 1, \frac{n}{\lambda}, \frac{n}{2\lambda})$, which means x is the median of M

(4). Swap($A[r]$, x)

(5). $q = Partition(A, p, r)$

(6). $k = q - p + 1$, which means that x is the k^{th} smallest number in $A[p : r]$

(7). If $k = i$: return $A[q]$

if $k > i$: return $Select(A, p, q - 1, i)$

if $k < i$: return $Select(A, q + 1, r, i - k)$

The running time for step (1), (2), (4), (6) are constant, and running time for (5) is $O(n)$, so we can use $c_0 n$ to denote their total time. Running time for (3) is $T(\frac{n}{\lambda})$, and what we really care about is running time for step (7).

If $\lambda = 3$:

After we found the median x, we know that there will be at least $\frac{n}{2 \cdot 3} \cdot 2 = \frac{n}{3}$ numbers in A are less than x, and at least $\frac{n}{3}$ numbers in A are larger than x, so step (7) will take at most $T(\frac{2n}{3})$ (some constant ignored here, but it won't influence the conclusion).

Thus we have $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + c_0 n$, if we assume that $T(n) = O(n)$, then we have $T(\frac{n}{3}) \leq \frac{cn}{3}$ and $T(\frac{2n}{3}) \leq \frac{2cn}{3}$, so $T(n) \leq c(\frac{n}{3}) + c(\frac{2n}{3}) + c_0 n = (c + c_0)n$, however we can not prove that $T(n) \leq cn$, so it's not a $O(n)$-time algorithm.

If $\lambda = 7$:

After we found the median x, we know that there will be at least $\frac{n}{2 \cdot 7} \cdot 4 = \frac{2n}{7}$ numbers in A are less than x, and at least $\frac{2n}{7}$ numbers in A are larger than x, so step (7) will take at most $T(\frac{5n}{7})$.

Thus we have $T(n) = T(\frac{n}{7}) + T(\frac{5n}{7}) + c_0 n$, if we assume that $T(n) = O(n)$, then we have $T(\frac{n}{7}) \leq \frac{cn}{7}$ and $T(\frac{5n}{7}) \leq \frac{5cn}{7}$, so $T(n) \leq c(\frac{n}{7}) + c(\frac{5n}{7}) + c_0 n = (\frac{6}{7}c + c_0)n$. If c is large enough to satisfy that $c \geq 7c_0$, we can prove that $T(n) \leq cn$, which means the $T(n) = O(n)$, because c_0 is a finite number.

Similarly, we can prove that when $\lambda = 9$, for $c \geq 6c_0$, we have $T(n) \leq cn$ which means $T(n) = O(n)$; when $\lambda = 11$, for $c \geq \frac{11}{2}c_0$, we have $T(n) \leq cn$ which means $T(n) = O(n)$.

Then consider the situation when λ is an even number.

If $\lambda = 4$:

we only change step (2) by finding the second large number x_i in each group instead of finding the median. Then there will be at least $\frac{n}{4}$ numbers in A are less than x, and at least $\frac{n}{4}$ numbers in A are larger than x, so we have $T(n) = T(\frac{n}{4}) + T(\frac{3n}{4}) + c_0 n$, this situation is the same as when $\lambda = 3$, it's not a O(n)-time algorithm. If we change step (2) by finding the third large number in every group, it's still the same situation: there will be at least $\frac{n}{4}$ numbers in A are less than x, and at least $\frac{n}{4}$ numbers in A are larger than x.

If $\lambda = 6$:

we only change step (2) by finding the third large number x_i in each group instead of finding the median. Then there will be at least $\frac{n}{4}$ numbers in A are less than x, and at least $\frac{n}{4}$ numbers in A are larger than x, so we have $T(n) = T(\frac{n}{6}) + T(\frac{3n}{4}) + c_0 n$, if we assume that $T(n) = O(n)$, then we have $T(\frac{n}{6}) \leq \frac{cn}{6}$ and $T(\frac{3n}{4}) \leq \frac{3cn}{4}$, so $T(n) \leq c(\frac{n}{6}) + c(\frac{3n}{4}) + c_0 n =$

$(\frac{11}{12}c + c_0)n$. If c satisfies that $c \geq 12c_0$, we can prove that $T(n) \leq cn$, which means the $T(n) = O(n)$.

2.

We can find a convex hull covering the given points, then the farthest points are among the vertices of the convex hull.

FarthestPoints(S):

(1). Sort the points in S by x coordinates, then denote the sorted points as $a_1, a_2...a_n$;

(2). Find convex hull covering $a_1...a_n$, and get vertices $b_1, b_2...b_m$ of the convex hull;

(3). Select one edge, b_1b_2 for example, draw its parallel lines on $b_3...b_m$, select the farthest one, b_p is the vertex on it, compute $|b_1b_p|$ and $|b_2b_p|$;

(4). Select next edge clock-wise, the farthest point to this edge should also be found in clock-wise direction (or still the same point). In total it will take $O(n)$ time to compute the farthest point for all the edges, then we can select the farthest points pair from all the computed pairs.

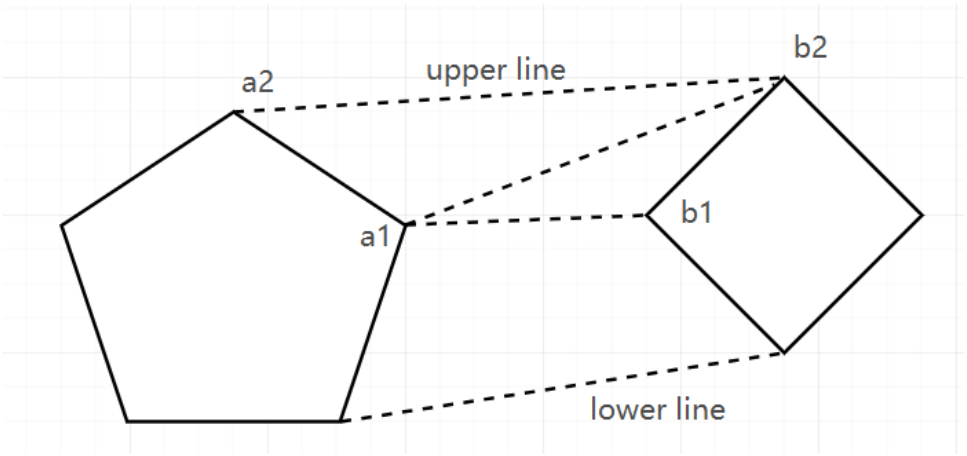
We use divide and conquer in step (2), it's easy because we have already sorted the points.

ConvexHull($a_1, a_2...a_n$):

(1). If $n = 3$, this can be directly done by drawing a triangle;

(2). Else call ConvexHull($a_1...a_{n/2}$)and CovexHull($a_{n/2}...a_n$);

(3). To combine two convex hulls together, we first connect the right-most point a_1 in the left convex hull and the left-most point b_1 in the right convex hull;



(4). Check if the line intersects the right convex hull, if it does, we choose clock-wise next point b_2 on right convex hull to replace b_1. Then check if the line passing through the left convex hull, if it does, we choose the anti clock-wise vertex a_2 on the left convex hull; Repeat until we get the line which doesn't intersects both convex hulls(denoted as upper line in the figure), this step will take $O(n)$;

(5). Similarly, we can find the lower line, then we have a new convex hull;

Easy to know we have $T(n) = 2T(\frac{n}{2}) + O(n)$ for this part.

The sort part will take $O(nlogn)$ time, the convex hull finding part will take $O(nlogn)$ time and finding the farthest points on the convex hull will take $O(n)$ time, so the running time for the whole algorithm is $O(nlogn)$,

3.

First sort the points by x coordinates, denote sorted points as $a_1, a_2...a_n$, then the problem can be solved by divided and conquer;

MaximumPoints($a_1, a_2...a_n$):

(1). Call MaximumPoints($a_1, a_2...a_{n/2}$),get maximum points of the left half: $p_1, p_2...p_s$;

(2). Call MaximumPoints($a_{n/2+1}, ...a_n$), get maximum points of the right half: $q_1, q_2...q_t$;

(3). We should notice that the maximum points of the right half are also maximum of the whole set, because we can't find points with larger x coordinates in the left half; What we need to do is looking for the maximum points in the left half dominated by some points in the right half; Easy to see that we only need to compare every p_i with q_1. Because q_1 has

largest y coordinates in the right half maximum points (or it will be dominated), and it has larger x coordinates than every p_i;

Sorting will take $O(nlogn)$ time, conquer part will take $O(n)$ time. Thus we have $T(n) = 2T(\frac{n}{2}) + O(n)$, it's a $O(nlogn)$ algorithm;

This problem can be easily extended to 3 dimensions; We still sort the points by x coordinates, then divide the space into 2 sub-spaces, A has smaller x coordinates, B has larger x coordinates; The maximum points in the group B will also be the maximum points of the whole set. The maximum points in the group A might be dominated. We can project the maximum points of A and B onto y-z plane. Then we need to solve a 2D problem to find which maximum points of A are dominated in the whole set.

(1). Sort the projected points by y coordinates, denote sorted points as $a_1, ...a_m$ by y coordinates increasing order. Set $z_{max} = -\infty$.

(2). a_i from a_m to a_1:

if $a_i \in B$ and a_i's z coordinate z_i is larger than z_{max}:

$z_{max} = z_i$

else if $a_i \in A$ and $z_i < z_{max}$:

mark a_i as dominated

At last we can get the maximum points in A that are also maximum points of the whole set.

The algorithm will take $T(nlog^2n)$ time, because of $T(n) = 2T(\frac{n}{2}) + O(nlogn)$.

4.

First sort by Merge Sort, S=MergeSort(S), this will take $O(nlogn)$ time.

FindNumber(S, B):

(1). If $length(S) < 3$: the numbers we want don't exist

(2). Else if $s_1 + s_2 + s_3 > B$: the numbers we want don't exist;

(3). Else if $s_n + s_{n-1} + s_{n-2} < B$: the numbers we want don't exist;

(4). Else: for i in range(1:n-2):

$j = i + 1,\ k = n$

while $(j < k)$:

if $s_i + s_j + s_k = B$ and $s_i \neq s_j \neq s_k$:

s_i, s_j, a_k are the numbers we are looking for;

else if $s_i + s_j + s_k < B$: $j = j + 1$;

else if $s_i + s_j + s_k > B$: $k = k - 1$;

(5). If we can not find numbers in step(3), then they don't exist.

Step(1),(2) and (3) in FindNumber(S, B) will take $O(1)$ time, and step(4) has 2 loops, which will at most take $O(n^2)$ time, and MergeSort(S) will take $O(nlogn)$ time, so the total running time is $O(n^2)$.

5.

If we divide the array into two parts, then the number of inversions in A is equal to the sum of number of inversions within the two parts and inversions between the two parts. The problem can be solved by divide and conquer. For each sub-problem, compute the number of inversions and sort them at the same time.

Inversion(A):

(1). sum=0, $n = length(A)$

(2). If $n = 1$: return 0

(3). Else:

$B = A[1 : \frac{n}{2}]$, $C = A[\frac{n}{2} : n]$

$sum = Inversion(B) + Inversion(C)$

$i = 1,\ j = 1,\ k = 1$

$while(i <= n)$:

if $j = length(B)$: $A[i] = C[k],\ k = k + 1,\ i = i + 1$

else if $k = length(C)$: $A[i] = B[j],\ j = j + 1,\ i = i + 1$

else if $B[j] < C[k]$: $A[i] = B[j],\ j = j + 1,\ i = i + 1$

else: $A[i] = C[k],\ k = k+1,\ i = i+1,\ sum = sum+length(B)-j+1$

(4). Return *sum*

Easy to know that $T(n) = 2T(\frac{n}{2}) + O(n)$, because the while loop will take $O(n)$ time, so running time is $T(n) = O(nlogn)$.

6.

Define a linear combination of p_i's:

$y = x_1p_1 + x_2p_2 + x_3p_3 + x_4p_4 + x_5p_5 + x_6p_6 + x_7p_7$

$y = (x_1 + x_3)as + (x_1 + x_4)at + (x_6 - x_1)cs - x_1ct + x_2bu + (x_2 + x_5)bv + (x_7 - x_2)du + (x_3 - x_2)dv + (x_3 + x_6 - x_7)ds + (x_3 + x_5 - x_4)av$

Easy to get:

$as + bu = p_2 + p_3 + p_7 - p_5$

$at + bv = p_4 + p_5$

$cs + du = p_6 + p_7$

$ct + dv = p_3 + p_4 - p_1 - p_6$