CSE 551 Homework 3 Solutions

October 2021

1 Question 1

Adapt the algorithm in class that finds closest points on a plane to work for points in 3-dimensions. Assume each point is represented by a tuple (x, y, z). What is the worst case complexity of your algorithm? Bonus: Can you generalize to n-dimensions?

Solution:

First sort the points by their x-components, y-components, and z-components and store in the arrays S_x, S_y, and S_z, respectively. For reasons we will get to soon, we must shift each data point by the smallest x, y, and z values (by adding) in S_x, S_y and S_z so that every point has only positive values in the coordinate systems. Observe this is an affine transformation so the distances between points are preserved under this transformation. Finally, sort the (newly translated) points based on the sum of their x and y components and store that result in S_{y+z}.

Then, create a plane perpendicular to the xz-plane the bisects the data. This divides the space into two halves H_L and H_R. Recursively solve the closest point problem for the H_L and H_R. This gives distances δ_L and δ_R and so, just as how δ is defined in the 2d case, let $\delta = \min\{\delta_L, \delta_R\}$.

It remains to show if there is a closer set of points where one point belongs to H_L and the other belongs to H_R. In this new merged list, we only want to consider points that lie δ away from the the dividing plane (the so-called δ-band). Eliminate those values in S_{y+z} whose x-component is more than δ away from band, call the resulting set S'_{y+z}. The following observation is made

Lemma 1.1. *Given a point in S'_{y+z} you only need to check a constant number more points later to find a closer point on the boundary.*

Proof. Given a point p in the band, it must be connected to some point that is less than δ away from p. Hence, it suffices to show there can be only a constant number of points to check in the $2\delta \times \delta \times 2\delta$ An upper bound on the number points in the cube can be 14 (the number corners of the the two $\delta \times \delta \times \delta$ cubes, removing the double count of the corners the two cubes share in common, and also two possible points in the middle of the cubes). □

Thus, given a point in the δ-band you only need to check a constant number of elements in S'_{x+y} after the index for a particular point p. Similar to the two dimensional case, you can do this for every point so iterate through each point and conduct do this for every p. Thus, really the only thing that changed from the 2 dimensional case is that we sorted by $y + z$ instead of by y. Sorting by the addition of the two does not change the complexity of mergesort. Thus, the resulting complexity is $O(n \log n)$.

In the case of n-dimensions, this problem can be solved using a similar procedure. Divide the data along a $n-1$ hyperplane along the y axis. Translate the data to a positive quadrant as done above. Instead of sorting by $y+z$ we need to take the sum over all axis but x. Thus, we can compute this at the beginning incurring a cost of $O(n \cdot N)$ where n (like mentioned above) is the number of dimensions and N is the number of points. The remaining follows similar to the 3 dimensional case. Note that by continuing the trend we need $2^{k+1} - 2^{k-1}$ points (both hypercubes minus the $k-1$-dimensional face they share. So when dimension is fixed, this algorithm has "great" assymptotic complexity, but when the dimension is not fixed, expect an exponential blow up.

2 Question 2

Tropical geometry is a sub-field of mathematics that has numerous applications including applications in computing the shortest path in a graph. The idea behind tropical geometry is instead of using the usual addition and multiplication, we take the minimum of two numbers for addition, and use the usual addition for multiplication.

Define $A \star B$, so that, given an $m \times n$ matrix A and $n \times p$ matrix B with entries $a_{ij} \in \mathbb{R}$ and $b_{ij} \in \mathbb{R}$ respectively, to be the matrix C that has entries $c_{ij} = \min_{k=1}^{n}\{a_{ik} + b_{kj}\}$. Similarly, define $A \oplus B = C$, where C has entries $c_{ij} = \min\{a_{ij}, b_{ij}\}$.

1. Prove $A \oplus B = B \oplus A$.
2. Prove $A \star (B \oplus C) = A \star B \oplus C \star D$.
3. Prove $(A \oplus B) \star C = A \star C \oplus B \star C$.
4. Prove for square matrices $A \star (B \star C) = (A \star B) \star C$.

It seems we can manipulate matrices similarly to how we do with the usual matrix multiplication and matrix addition. If we use $\oplus$ and $\star$ instead of the usual addition and multiplication for matrices, can we apply the naive divide and conquer algorithm to compute the product of $A \star B$? What about Strassen's algorithm?

Solution:

First we address items 1, 2, and 3:

1. $A \oplus B = C$, with $B \oplus A = C'$. Observe that C has entries $c_{ij} = \min\{a_{ij}, b_{ij}\}$ and C' has entries $c'_{ij} = \min\{b_{ij}, a_{ij}\}$. But as $\min\{a, b\} = \min\{b, a\}$, it follows $\min\{a_{ij}, b_{ij}\} = \min\{b_{ij}, a_{ij}\}$. Thus, $c_{ij} = c'_{ij}$ for every i and j, thus $C = C'$ and so $A \oplus B = B \oplus A$.
2. $B \oplus C$ is a matrix with entries $\min\{b_{ij}, c_{ij}\}$. Thus, $A \star (B \oplus C)$ has entries (1) $\min_{k=1}^{n}\{a_{ik} + \min\{b_{kj}, c_{kj}\}\}$. Next observe that $A \star B \oplus A \star C$ has entries (2) $\min\{\min_{k=1}^{n}\{a_{ik} + b_{kj}\}_{ij}, \min_{k=1}^{n}\{a_{ik} + c_{kj}\}_{ij}\}$, but $\min\{a + b, a + c\} = a + \min\{b, c\}$, so (1) and (2) are in fact equal.
3. $A \oplus B$ is a matrix with entries $\min\{a_{ij}, b_{ij}\}$. Thus, $(A \oplus B) \star C$ has entries (1) $\min_{k=1}^{n}\{\min\{a_{kj}, b_{kj}\} + c_{ik}\}$. Next observe that $A \star C \oplus B \star C$ has entries (2) $\min\{\min_{k=1}^{n}\{a_{ik} + c_{kj}\}, \{\min_{k=1}^{n}\{b_{ik} + c_{kj}\}\}$, but $\min\{a + b, a + c\} = \min\{b, c\} + a$, so (1) and (2) are in fact equal.

Four the fourth property, we have already introduced the notation where $\oplus := \min$, let us also define the realted $\bigoplus_{k=1}^{n} := \min_{k=1}^{n}$

$$
\begin{aligned}
(A \star B)_{ij} &= \bigoplus_{p=1}^{n} a_{ip} \star b_{pj} \\
((A \star B) \star C)_{i'j'} &= \bigoplus_{p'=1}^{n} (A \star B)_{i'p'} \star c_{p'j'} \\
&= \bigoplus_{p'=1}^{n} \bigoplus_{p=1}^{n} a_{i'p} \star b_{pp'} \star c_{p'j}
\end{aligned}
$$

$$
\begin{aligned}
(B \star C)_{ij} &= \bigoplus_{p'=1}^{n} b_{ip'} \star c_{p'j} \\
(A \star (B \star C))_{i'j'} &= \bigoplus_{p=1}^{n} a_{i'p} (B \star C)_{pj'} \\
&= \bigoplus_{p'=1}^{n} a_{i'p} \star \bigoplus_{p=1}^{n} b_{pp'} \star c_{p'j'} \\
&= \bigoplus_{p=1}^{n} a_{i'p} \star \bigoplus_{p'=1}^{n} b_{pp'} \star c_{p'j'} \\
&= \bigoplus_{p=1}^{n} \bigoplus_{p'=1}^{n} a_{i'p} \star b_{pp'} \star c_{p'j'}
\end{aligned}
$$

The question now becomes can the inner and outer $\bigoplus$ be exchanged so the two equations become equal. Yes, min is a commutative operation so, no matter the order in which we minimize a finite number of elements the result stays the same.

Strassen's algorithm cannot be applied here because Strassen requires subtraction, but there does not always exist an additive inverse ($a + (-a) = 0$). -1, for instance, has no additive inverse as $\min\{-1, x\} \leq -1$. The naive algorithm, however, does not require any properties other than those mentioned above so it follows that the naive algorithm works.

3 Question 3

We are given two matrices of size $m \times n$ and $n \times p$ and the values m, n, and p (p is not necessarily a power of 2!). Can we apply the Strassen multiplication approach to multiply these two matrices? If so, what modifications have to be done? What is the running time of the algorithm in terms of m, n, and p?

Solution: Yes. Pad the matrices with zeros so the dimensions of the matrices $2^k \times 2^k$, such that k is the smallest k where $2^k > \max\{n, m, p\}$. Running time: $O(k^{2.81})$ where $k = \max\{n, m, p\}$

4 Question 4

Suppose you are given an $n \times n$ array where the elements are sorted in such a way that all elements in a row are in increasing order from left to right and all the elements in a column are in increasing order from top to bottom. Your goal is to search for a target value in this array. Give a divide and conquer algorithm to search for the target, and make the worst case as efficient as you can. Write the recurrence relation for the algorithm.

Solution:

The high-level idea behind this algorithm is (1) if the value is found, stop. (2) If the value is not found, divide the matrix into 4 sub-matrices of roughly the same size. If the value being searched for is greater than the value in the center of the matrix, then you do not need to consider the top left sub-matrix, only the other 3. Similarly, if the value being searched for is less than the value in the center of the matrix, then you do not need to consider the bottom right sub-matrix. Then, you recursively call the remaining 3 matrices.

Algorithm 1 presents this argument but with additional details filled in when the matrix cannot be evenly divided into.

Algorithm 1 Divide Conquer Search

```
1: procedure MAT-SEARCH(M, v, m, n)    ▷ M is an m × n matrix, v is the value we are searching for.
2:     cm := ⌊m/2⌋
3:     cn := ⌊n/2⌋
4:     mid ← M[cm][cn]
5:     if mid ≠ v and m, n = 0 then
6:         return Not found!
7:     if mid = v then
8:         return Found!
9:     else if mid < v then
10:        MAT-SEARCH(M[0 ... cm][0 ... cn], v, cm, cn)
11:        MAT-SEARCH(M[0 ... cm][cn ... n], v, cm, cn)
12:        MAT-SEARCH(M[cm ... m][cn ... n], v, cm, cn)
13:    else                                                    ▷ mid > v
14:        MAT-SEARCH(M[cm ... m][cn ... n], v, cm, cn)
15:        MAT-SEARCH(M[0 ... cm][cn ... n], v, cm, cn)
16:        MAT-SEARCH(M[cm ... m][cn ... n], v, cm, cn)
```

As for the recurrence relation, denote the recurrent relation as $T(n)$. In the recursive call, the problem is reduced into 3 sub-problems with half the dimension of the original problem. Each sub-problem only incurs an additional constant time penalty for basic arithmetic on top of the recursive call. Hence, the recurrence relation is

$$T(n) = 3T(n/2) + O(1)$$

.

5 Question 5

You are given n keys and n locks. Each key can unlock precisely one lock. The difference in size between the keys is too small for you to notice if you try comparing two keys directly. Similarly, upon visual inspection, you cannot compare two locks and determine which keyhole is larger or smaller. When you use insert a key in a lock, if the key is too small, it will go into the keyhole, but the lock will not unlock. If the key is too large for the lock, it will not fit into the keyhole. A key will only unlock a lock if it is the correct size. Give the fastest algorithm that you can (in the worst case) to match all keys to their corresponding locks.

Solution: The naive approach for this problem (check every key to every to every lock until it fits) will give us a $\Theta(n^2)$ algorithm.

Improving on the worst case $O(n^2)$ bound is difficult, however the average case for this problem can be improved significantly utilizing an algorithm similar to quicksort. Before presenting the algorithm, recall in the problem we cannot compare locks to locks and keys to keys, but we may compare keys to locks. Therefore, it is perfectly reasonable to say

1. key $<$ lock (or lock $>$ key) to mean the key can be inserted into the lock but not unlock the lock,
2. key $=$ lock (or lock $=$ key) to mean the key will unlock the lock,
3. and key $>$ lock (or lock $<$ key) to mean the key is too large for the lock.

The resulting algorithms (Algorithm 2 and Algorithm 3) sort the keys and locks by first arranging locks based on a pivot key, then arranging the keys based on the lock that matched the pivot key.

Algorithm 2 Match keys to locks

```
1: procedure QUICK-UNLOCK(keys[], locks[], low, high, pivot)
2:     if low < high then:
3:         p ← QUICK-UNLOCK-PARTITION(keys[], locks[] low, high)
4:         QUICK-UNLOCK(keys[], locks[], low, p-1)
5:         QUICK-UNLOCK(keys[], locks[], p+1, high)
```

Algorithm 3 Unlock Locks Partition Subroutine

```
1: procedure QUICK-UNLOCK-PARITION(keys[], locks[], low, high)
2:     pivot ← keys[high]
3:     low_index ← low
4:     for i = 0, ... n − 1 do
5:         if locks[i] < pivot then
6:             Swap lock[i] with lock[low_index]
7:             low_index = low_index + 1
8:         else
9:             pivot_lock ← locks[i]
10:    low_index ← low
11:    for i = 0, ... n − 1 do
12:        if key[i] < pivot_lock then
13:            Swap key[i] with key[low_index]
14:            low_index = low_index + 1
15:    Swap key[low_index] with key[high]
16:    Swap lock[low_index] with lock[high]
17:    return low_index
```