Soham Chatterjee
Email: soham.chatterjee@tifr.res.in
Course: Algorithms

Assignment - 1
Dept: STCS
Date: September 10, 2024

Problem 1 P3 (15 marks)

Solve the recurrences:

(i) $T(n) = 2T(n/2) + n\log n$,

(ii) $T(n) = 7T(n/3) + n^2$,

(iii) $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

Solution:

(i) We have the recurrence relation $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$. So

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + n\log n\\ &= 4T\left(\frac{n}{4}\right) + \frac{n}{2}\log\frac{n}{2} + n\log n \le 2^2T\left(\frac{n}{2^2}\right) + 2n\log n\\ &= 2^3T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\log\frac{n}{2^2} + 2n\log n \le 2^3T\left(\frac{n}{2^3}\right) + 3n\log n\\ &\dots\\ &= 2^kT\left(\frac{n}{2^k}\right) + \frac{n}{2^k}\log\frac{n}{2^k} + (k-1)n\log n \le 2^kT\left(\frac{n}{2^k}\right) + kn\log n\\ &\dots\\ &\le 2^{\log n}T(1) + \log n(n\log n) \le nT(n) + n\log^2 n = n(T(1) + \log^2 n) = O(n\log^2 n)\end{aligned}$$

So we claim $T(n) \le cn(T(1)+\log^2 n)$ for all $n \ge n_0$ for some c which we will choose accordingly. Now $n_0 = 2$. So for $n = 2$ we have $T(2) = 2T(1) + 2\log 2 = 2T(1) + 2 = 2(T(1)+1) \le c2(T(1) + \log^2 2)$. Hence the base case follows. Now let $T(n) = cn\log^2 n$ is true for all $n = 2, \dots, k-1$. Now

$$T(k) = 2T\left(\frac{k}{2}\right) + k\log k \le 2c\frac{k}{2}\left(T(1) + \log^2\frac{k}{2}\right) + k\log k = ck\left(T(1) + \log^2\frac{k}{2}\right) + k\log k$$

Now $\log^2\frac{k}{2} = (\log k - 1)^2 = \log^2 k - 2\log k + 1$. So we have

$$ck\left(T(1) + \log^2\frac{k}{2}\right) + k\log k = ck(T(1)+\log^2 k) - 2ck\log k + ck + k\log k = ck(T(1)+\log^2 k) + (1-2c)k\log k + ck$$

If $c \ge 1$ we have $1 - 2c \le -1$. So we have

$$(1-2c)k\log k + ck \le ck - k\log k \le 0$$

Here the last inequality follows if $c \le \log k$. Since $k \ge 2$ we have $\log k \ge 1$. So take $c = 1$. Then $(1 - 2c)k\log k + ck \le 0$. Therefore

$$T(k) = k(T(1) + \log^2 k) + (1-2)k\log k + k \le k(T(1) + \log^2 k)$$

Hence by mathematical induction we have for all $n \ge 2$, $n \in \mathbb{N}$ we have $T(n) \le n(T(1) + \log^2 n)$. Now

$$n(T(1) + \log^2 n) = n(T(1)\log^2 n + \log^2 n) = (1 + T(1))n\log^2 n = O(n\log^2 n)$$

Hence we have $T(n) = O(n\log^2 n)$.

(ii) We have the recurrence relation $T(n) = 7T\left(\frac{n}{3}\right) + n^2$. So

$$\begin{aligned}
T(n) &= 7T\left(\frac{n}{3}\right) + n^2 \\
&= 7^2 T\left(\frac{n}{3^2}\right) + \frac{n^2}{9} + n^2 \\
&= 7^3 T\left(\frac{n}{3^3}\right) + \frac{n^2}{3^4} + \frac{n^2}{3^2} + n^2 = 7^3 T\left(\frac{n}{3^3}\right) + n^2 \sum_{i=1}^{3} \frac{1}{3^{2i}} \\
&\cdots \\
&= 7^k T\left(\frac{n}{3^k}\right) + n^2 \sum_{i=1}^{k} \frac{1}{9^i} \\
&\cdots \\
&= 7^{\log_3 n} T(1) + n^2 \sum_{i=1}^{\log_3 n} \frac{1}{9^i} \leq n^{\log 7_3} T(1) + \frac{9}{8} n^2 \leq T(1)n^2 + \frac{9}{8} n^2 = \left(T(1) + \frac{9}{8}\right) n^2
\end{aligned}$$

So we claim $T(n) = (T(1) + c)n^2$ for some $c \geq 2$ and $n \geq n_0$ where $n_0 \in \mathbb{N}$. So take $n_0 = 3$. Then $T(3) = 7T(1) + 9 \leq 9T(1) + 18 \times 9 = (T(1) + c)9$. Hence this follows for the base case. Now suppose $T(n) = (T(1) + c)n^2$ for all $n = 3, \ldots, k-1$. Then for $n = k$

$$T(k) = 7T\left(\frac{k}{3}\right) + k^2 \leq 7(T(1) + c)\frac{k^2}{3^2} + k^2 = k^2\left(\frac{7(T(1) + c)}{9} + 1\right)$$

We want

$$\frac{7(T(1) + c)}{9} + 1 \leq T(1) + c \iff 7(T(1) + c) + 1 \leq 9(T(1) + c) \iff 1 \leq 2(c + T(1))$$

this is indeed true since $c \geq 2$. Hence we have $T(k) \leq (c + T(1))k^2$. Hence by mathematical induction we have $T(n) \leq (c + T(1))n^2$ for all $n \geq 4$ with $n \in \mathbb{N}$. Now $(c + T(1))n^2 = O(n^2)$. Hence $T(n) = O(n^2)$.

(iii) We have the recurrence relation

$$T(n) = \sqrt{n}T(\sqrt{n}) + n \iff \frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 1$$

Now denote $F(n) = \frac{T(n)}{n}$. Then we have the new recurrence relation

$$f(n) = f(\sqrt{n}) + 1$$

Now suppose $n = 2^{2^k}$. Then

$$\begin{aligned}
f\left(2^{2^k}\right) &= f\left(\sqrt{2^{2^k}}\right) + 1 = f\left(2^{2^{k-1}}\right) + 1 \\
&= f\left(2^{2^{k-2}}\right) + 2 \\
&\cdots \\
&= f\left(2^{2^0}\right) + k \\
&= f(2) + k
\end{aligned}$$

Now $f(2) = \frac{T(2)}{2}$ which is a constant. So there exists $n_0 \in \mathbb{N}$ such that $f(2) \leq \log\log n$ for all $n \geq n_0$. So for large k we have

$$f\left(2^{2^k}\right) = f(2) + k \leq 2k$$

Hence we claim $f(n) = O(\log_2 \log_2 n)$. For $n = n_0$ we already have $f(n_0) \le 2\log_2 \log_2 n$. So let for $n = n_0, \dots, t-1$ we have $f(n) \le c \log_2 \log_2 n$ for some $c \in \mathbb{N}$. Certainly seeing the $n = n_0$ we have $c \ge 2$ but we will choose c appropriately later. Now for $n = t$

$$\begin{aligned} f(t) &= f(\sqrt{t}) + 1 \\ &\le c\log_2 \log_2(\sqrt{t}) + 1 \\ &= c\log_2\left(\frac{1}{2}\log_2 t\right) + 1 \\ &= c\log_2 \frac{1}{2} + c\log_2 \log_2 t + 1 \\ &= c\log_2 \log_2 t - c + 1 \le 2\log_2 \log_2 t \end{aligned}$$

So if we choose $c = 2$ then we are done. Hence by mathematical induction $f(n) = O(\log_2 \log_2 n)$ for all $n \ge n_0$. Now we have $f(n) = \frac{T(n)}{n}$ and $f(n) = O(\log_2 \log_2 n)$. Hence we have

$$T(n) = O(n \log_2 \log_2 n)$$

■

Problem 2 P4 (5 marks)

Give the best upper bounds you can on the nth Fibonacci number F_n, where $F_n = F_{n-1} + F_{n-2}$ and $F_1 = F_2 = 1$

Solution: We have the recurrence relation $F(n) = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. So we can represent this with matrices like following:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} = \cdots \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} F_2 \\ F_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Denote $\overline{F}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\overline{F}_k = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$ and $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. Therefore we have $\overline{F}_n = A^n \overline{F}_0$.

Now clearly A has full rank and $\forall\, k \in \mathbb{N}$, $\overline{F}_k \in \mathbb{R}^2$. So we will find the eigenvalues of A to find an eigenbasis.

$$\det(A - tI) = \det \begin{bmatrix} 1-t & 1 \\ 1 & -t \end{bmatrix} = -t(1-t) - 1 = t^2 - t - 1$$

So if $t^2 - t - 1 = 0$ then

$$t = \frac{1 \pm \sqrt{1+4}}{2} = \frac{1 \pm \sqrt{5}}{2}$$

So denote $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$. Now let $X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ be an eigenvector corresponding to φ. Then

$$AX = \begin{bmatrix} x_1 + x_2 \\ x_1 \end{bmatrix} = \varphi \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Therefore $x_1 = \varphi x_2$. Therefore take $x_2 = 1$ then we have $x_1 = \varphi$. So $X = \begin{bmatrix} \varphi \\ 1 \end{bmatrix}$. Similarly we have $Y = \begin{bmatrix} \psi \\ 1 \end{bmatrix}$ is an eigenvector of A corresponding to ψ.

Now we want to express $\overline{F}_0$ as a linear combination of X, Y. Notice

$$\frac{1}{\sqrt{5}}(X - Y) = \frac{1}{\sqrt{5}} \begin{bmatrix} \varphi - \psi \\ 0 \end{bmatrix} = \frac{1}{\sqrt{5}} \begin{bmatrix} \frac{1=\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} \\ 0 \end{bmatrix} = \frac{1}{\sqrt{5}} \begin{bmatrix} \sqrt{5} \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \overline{F}_0$$

Therefore

$$\overline{F}_n = A^n \overline{F}_0 = A^n \left(\frac{1}{\sqrt{5}}(X - Y) \right) = \frac{1}{\sqrt{5}}(AX - AY) = \frac{1}{\sqrt{5}}(\varphi^n X - \psi^n Y) = \frac{1}{\sqrt{5}} \varphi^n \begin{bmatrix} \varphi \\ 1 \end{bmatrix} - \frac{1}{\sqrt{5}} \psi^n \begin{bmatrix} \psi \\ 1 \end{bmatrix}$$

Therefore $F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$.

[I knew about How to solve Linear Recurrences using Matrices] ■

Problem 3 P5 (10 marks)

Consider two sets A and B, each having n integers in the range from 0 to $10n$. We wish to compute the Cartesian sum of A and B, defined by

$$C = \{x + y : x \in A, y \in B\}$$

Note that the integers in C are in the range 0 to $20n$. We want to find the elements in C and the number of times each element of C is realized as a sum of elements in A and B. Give an algorithm that solves the problem in $O(n \log n)$ time, and prove correctness.

Solution: Given A, B we create two polynomials $p_A(x) = \sum\limits_{k \in A} x^k$ and $p_B(x) = \sum\limits_{k \in B} x^k$. Since all entries of A and B ranges form 0 to $10n$. We have $\deg p_A \le 10n$ and $\deg p_B \le 10n$. Hence now we can use the algorithm for polynomial multiplication to calculate $p = p_A \cdot p_B$. Now $\deg p \le 20n$. For any term x^k in p, $\exists\ a \in A$ and $b \in B$ such that $a + b = c$ since p is the product of p_A and p_B. Let $S_k := \{(a, b) \in A \times B$ such that $a + b = k\}$. Then the coefficient of x^k in p is $|S_k|$ since

$$\text{Coeff}(x^k) = \sum_{i=0}^{k} \text{Coeff}_A(x^i) \cdot \text{Coeff}_B(x^{k-i})$$

where $\text{Coeff}_A(x^i)$ is the coefficient of x^k in p_A and $\text{Coeff}_B(x^{k-i})$ is the coefficient of x^{k-i} in p_B. Hence for any $i \in \{0, \dots, k\}$, $\text{Coeff}_A(x^i) \cdot \text{Coeff}_B(x^{k-i})$ will contribute to $\text{Coeff}(x^k)$ iff both of them are 1 iff $(i, k - i) \in A \times B$. So

$$\text{Coeff}(x^k) = |\{(a, b) \in A \times B \mid a + b = k\}$$

So now we will describe the algorithm. We denote the polynomial multiplication algorithm of two polynomials S, T by Polynomial-Multiplication(S, T). So the algorithm will be:

Algorithm 1: Cartesian-Sum

```
Input: $A = \{a_i \mid i \in [n],\ a_i \in \mathbb{Z},\ 0 \le a_i \le 10n\}$, $B = \{b_i \mid i \in [n],\ b_i \in \mathbb{Z},\ 0 \le b_i \le 10n\}$
Output: $C = \left\{(c, k_c) : \exists\ a \in A,\ b \in B \text{ st } c = a + b,\ k_c = \left|\{(a, b) \in A \times B \mid a + b = c\}\right|\right\}$
1 begin
2     Create two arrays $S_A$ and $S_B$ of length $10n + 1$ with all elements 0
3     for $i = 1, \dots, n$ do
4         $S_A[A[i]] \longleftarrow 1$          // Creates the polynomial $p_A$
5         $S_B[B[i]] \longleftarrow 1$          // Creates the polynomial $p_B$
6     $S \longleftarrow$ Polynomial-Multiplication$(S_A, S_B)$
7     $C \longleftarrow \{(k, S[k]) : S[k] \ne 0\}$
8     return $C$
```

Time Complexity: To create S_A and S_B it takes $O(n)$ time. Also for the for loop in each iteration it takes constant time and the loop runs for n iterations so the for loop in total takes $O(n)$ time.

Now in class we did the Polynomial-Multiplication algorithm which multiplies two degree $n - 1 = O(n)$ polynomial in $O(n \log n)$ time. Since the arrays S_A and S_B represents the polynomials p_A and p_B respectively and $\deg p_A, \deg p_B \le 10n = O(n)$ the Polynomial-Multiplication algorithm takes $O(n \log n)$ time to multiply them. Therefore in Line 6 it takes $O(n \log n)$ time.

Now $p = p_A \cdot p_B$. So $\deg p \le 20n$. So S is of length at most $20n = O(n)$. Therefore to create C it takes $O(n)$ time.

Therefore total time taken by the algorithm is $O(n) + O(n) + O(n \log n) + O(n) = O(n \log n)$. Hence time complexity of the algorithm is $O(n \log n)$. ∎

Problem 4 P6 (20 marks)

Define $[n] := \{1, 2, \ldots, n\}$. You are given n, and oracle access to a function $f : [n] \times [n] \to [n] \times [n]$ that takes as input two positive integers of value at most n, and returns two positive integers of value at most n. Let $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$ be the first and second coordinates of $f(x_1, x_2)$, respectively. You are also told that f_i is monotone nondecreasing in coordinate i when coordinate $3-i$ is kept fixed, and monotone nonincreasing in coordinate $3-i$ when coordinate i is kept fixed. That is, given $x_1 \le x_1' \in [n]$ and $x_2 \le x_2' \in [n]$, $f_1(x_1, x_2) \le f_1(x_1', x_2)$, and $f_1(x_1, x_2) \ge f_1(x_1, x_2')$. Similarly, $f_2(x_1, x_2) \ge f_2(x_1', x_2)$, and $f_2(x_1, x_2) \le f_2(x_1, x_2')$.

The problem is to find a fixed point of the function, i.e., values $x_1, x_2 \in [n]$ so that $f(x_1, x_2) = (x_1, x_2)$. Give an algorithm that given n and oracle access to such a function f, finds a fixed point of f in time $O(\text{poly}(\log n))$. You must also give a proof of correctness, and running time analysis.

Solution: We will solve this algorithm for any $n \times m$ where $n, m \in \mathbb{N}$ block where where we are given the oracle of a function $f : [n] \times [m] \to [n] \times [n]$ with properties same as given in question. First we will prove that for such function there is a fixed point.

Lemma 1. *Suppose $g : [k] \to [l]$ is an non-decreasing function for $k, l \in \mathbb{N}$ with $l \le k$ then there is a fixed point of g.*

Proof: Suppose there is no fixed point. Then $g(1) \ne 1 \implies g(1) \ge 2$. Then $g(2) \ge 2$ and $g(2) \ne 2 \implies g(2) \ge 3$. Similarly we get $g(3) \ge 4$. Continuing like this we get $g(k) \ge k+1$ which is not possible as for all $i \in [k]$, $g(i) \le l \le k$. Hence contradiction. Therefore g has a fixed point. $\square$

Lemma 2. *For the function $f : [n]\times[m] \to [n]\times[m]$ where f can be written as $f = (f_1, f_2)$ where $f_1 : [n]\times[m] \to [n]$ and $f_2 : [n] \times [m] \to [m]$. Then for $x_1 \le x_1' \in [n]$ and $y_1 \le y_1' \in [m]$*

- $f_1(x_1, y_1) \le f_1(x_1', y_1)$ *and* $f_1(x_1, y_1) \ge f_1(x_1, y_1')$
- $f_2(x_1, y_1) \ge f_2(x_1', y_1)$ *and* $f_2(x_1, y_1) \le f_2(x_1, y_1')$

Then f has a fixed point i.e. $\exists\ p \in [n], q \in [m]$ such that $f(p, q) = (p, q)$

Proof: Consider every vertical line i.e. $L_i := [n] \times \{i\}$ for all $i \in [m]$. For each of the line we claim there is a fixed point for f_1. For all $i \in [m]$ we can think of the new function $g_i : [n] \to [n]$ where $g_i(k) = f_1(k, i)$. Then by Lemma 1 there is a fixed point of g_i.

Let for each $i \in [m]$ let the $p_i \in [n]$ such that $g_i(p_i) = p_i$. Hence $f_1(p_i, i) = p_i$. Since there are m such points we claim that we can get such m points with the property that $p_1 \ge p_2 \ge \cdots \ge p_m$. Suppose for any $i \in [m]$ we have $f_1(p_i, i) = p_i$. Then consider the point $(p_i, i-1)$. Then we have $f_1(p_i, i-1) \ge f_1(p_i, i) = p_i$. Hence for the line segment $L_{i-1}^{p_i} = \{(x, i-1) \mid p_i \le x \le n\}$. the value of the function f_1 on $L_{i-1}^{p_i}$ is lower bounded by p_i and upper bounded by n. So we can think

$$h_{i,i-1} : [n - p_i + 1] \to [n - p_i + 1] \text{ where } f_1(x, i-1) = h_{i,i-1}(x - p_i) + p_i$$

Now by Lemma 1 $h_{i,i-1}$ has a fixed point. Therefore there is $k \in [n - p_i + 1]$ such that $h(k) = k$. Then $f_1(k + p_i) = h_{i,i-1}(k) + p_i = k + p_i$. So take $p_{i-1} = k + p_i$. Hence $p_{i-1} \ge p_i$. Therefore with this we get m fixed points $p_1, \ldots, p_m$ wrt f_1 on each horizontal lines L_i such that $p_i \ge p_{i+1}$ for all $i \in [m-1]$.

Now suppose we have the m points (p_i, i) for all $i \in [m]$ such that $p_i \ge p_{i+1}$ for all $i \in [m-1]$ such that $f(p_i, i) = p_i$ for all $i \in [m]$. Now we will show there is fixed point wrt f_2 among these m points. If we can show that we are done. Now for any $i \in [m-1]$ we have $f_2(p_i, i) \le f_2(p_i, i+1) \le f_2(p_{i+1}, i+1)$. Now consider the function $w : [n] \to [n]$ where $w(i) = f_2(p_i, i)$. Then since we have for all $i \in [m-1]$, $f_2(p_i, i) \le f_2(p_{i+1}, i+1)$, w is nondecreasing. Therefore by Lemma 1 there is a fixed point. Hence $\exists\ k \in [n]$ such that $w(k) = k$. Hence $f_2(p_k, k) = k$. Hence we get a point (p_k, k) such that $f_1(p_k, k) = p_k$ and $f_2(p_k, k) = k$ or $f(p_k, k) = (p_k, k)$. $\square$

Now we will describe the algorithm. We will make our algorithm work for any $n \times m$ board. First we will consider the line $L_{\lfloor \frac{m}{2} \rfloor} = \left\{\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) : k \in [n]\right\}$. By Lemma 1 there is a fixed point wrt f_1 in $L_{\lfloor \frac{m}{2} \rfloor}$. Let the point is $\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right)$. So $f_1\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) = k$.

Now we check the value of f_2 at $\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right)$.

- **If $f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) > \left\lfloor \frac{m}{2} \right\rfloor$:** Then for all points (x, y) with $x \le k$, $y \ge \left\lfloor \frac{m}{2} \right\rfloor$ we have

$$\left\lfloor \frac{m}{2} \right\rfloor < f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) \le f_2\left(x, \left\lfloor \frac{m}{2} \right\rfloor\right) \le f_2(x, y) \le n$$

  Hence in the $[k] \times \left\{\left\lfloor \frac{m}{2} \right\rfloor, \dots n\right\}$. If we restrict f then we have $f : [k] \times \left\{\left\lfloor \frac{m}{2} \right\rfloor, \dots n\right\} \to [k] \times \left\{\left\lfloor \frac{m}{2} \right\rfloor, \dots n\right\}$. Therefore by Lemma 2 there is a fixed point in the region $[k] \times \left\{\left\lfloor \frac{m}{2} \right\rfloor, \dots n\right\}$. So we can just search for the fixed point in that region.

- **If $f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) < \left\lfloor \frac{m}{2} \right\rfloor$:** Then for all points (x, y) with $x \ge k$, $y \le \left\lfloor \frac{m}{2} \right\rfloor$ we have

$$\left\lfloor \frac{m}{2} \right\rfloor > f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) \ge f_2\left(x, \left\lfloor \frac{m}{2} \right\rfloor\right) \ge f_2(x, y) \ge 1$$

  Hence in the $\{k, \dots, n\} \times \left[\left\lfloor \frac{m}{2} \right\rfloor\right]$ if we restrict f then we have $f : \{k, \dots, n\} \times \left[\left\lfloor \frac{m}{2} \right\rfloor\right] \to \{k, \dots, n\} \times \left[\left\lfloor \frac{m}{2} \right\rfloor\right]$. Therefore by Lemma 2 there is a fixed point in the region $\{k, \dots, n\} \times \left[\left\lfloor \frac{m}{2} \right\rfloor\right]$. So we can just search for that fixed point.

- **If $f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) = \left\lfloor \frac{m}{2} \right\rfloor$:** Then we already have a fixed point so we are done.

Here in order to give the sets like $\{k, \dots, l\}$ where $k, l \in \mathbb{N}$ we can just input the ordered pair (k, l) to denote the set. So we don't have to input all the $l - k + 1$ many points into the algorithm. So in the following algorithm by $\{k, \dots, l\}$ as input to the algorithm we mean the ordered pair (k, l). So now we have the following algorithm for a general $[n] \times [m]$ block:

Algorithm 2: FIXED-POINT($[n]$, $[m]$)

```
1  begin
2      if $|A| \times |B| \le 4$ then
3          Then find fixed point using brute force (Check value of $f$ at all points)
4      $k \longleftarrow$ Find fixed point of $g(x) = f_1\left(x, \left\lfloor \frac{m}{2} \right\rfloor\right)$ by Binary Search
5      if $f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) == \left\lfloor \frac{m}{2} \right\rfloor$ then
6          return $\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right)$
7      if $f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) > \left\lfloor \frac{m}{2} \right\rfloor$ then
8          return FIXED-POINT$\left([k], \left\{\left\lfloor \frac{m}{2} \right\rfloor, \dots m\right\}\right)$
9      if $f_2\left(k, \left\lfloor \frac{m}{2} \right\rfloor\right) < \left\lfloor \frac{m}{2} \right\rfloor$ then
10         return FIXED-POINT$\left(\{k, \dots, n\}, \left[\left\lfloor \frac{m}{2} \right\rfloor\right]\right)$
```

Time Complexity: At first in line 2,6 it takes constant time to check. In line 4 it takes $O(\log n)$ time to find the fixed point for $g(x) = f_1\left(x, \left\lfloor \frac{m}{2} \right\rfloor\right)$. Now in line 7-8 or 9-10 depending on k can be at worst 1 or n. In that case we have the recursion relation $T(nm) \le T\left(\frac{nm}{2}\right) + O(\log n) \le T\left(\frac{nm}{2}\right) + O(\log n) + O(\log m) = T\left(\frac{nm}{2}\right) + O(\log nm)$. Hence we have $T(nm) = O(\log^2 nm)$.

Hence if we run our algorithm for $m = n$ we have the algorithm for the given problem. So FIXED-POINT($[n]$, $[n]$) takes time $O(\log^2 n^2) = O(\log^2 n)$ time. Hence we have an algorithm to find the fixed point of the $[n] \times [n]$ with the oracle for the given function in $O(\log^2 n)$ time. ■

[Me and Soumyadeep came up with the solution together. I discussed with Shubham]

Problem 5 P7 (15 marks)

A palindrome is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia`. Give an efficient algorithm, with proof of correctness and run-time analysis, to find the longest palindrome that is a subsequence of a given input string. For example, given the input string `character`, your algorithm should return `carac`.

Solution: We will solve this using dynamic programming. So we will create an array LPS of $n \times n$ size where for any $i, j \in [n]$ with $i \leq j$, $LPS[i, j]$ denotes the longest palindromic subsequence starting in the subarray starting at i and ending at j. Let the given string is S. Now for any $i, j \in [n]$ with $i \leq j$ we have:

- If $S[i] = S[j]$ then $LPS[i, j] = 2 + LPS[i+1, j-1]$
- If $S[i] \neq S[j]$ then $LPS[i, j] = \max\{LPS[i+1, j], LPS[i, j-1]\}$

This is the main recursion relation in order to find the longest palindromic subsequence. For edge cases we have

- If $i = j$ then $LPS[i, i] = 1$
- If $j = i+1$ and $S[i] = S[j]$ then $LPS[i, j] = 2$.

In order to have the algorithm by the recursion relation we have to move by finding all the entries of LPS in each diagonal parallel to principle diagonal and then move to the next diagonal parallel to principle diagonal. There are n diagonals parallel to principle diagonal. In each diagonal d there are $n+1-d$ many points. And for any diagonal d the i^{th} point on the diagonal has coordinates $(i, d+i-1)$. So with this we describe the algorithm:

Algorithm 3: LONGEST-PALINDROMIC-SUBSEQUENCE(S)

```
Input: A string $S$ of length $n$.
Output: $T$ where $T \subseteq S$ is the Longest Palindromic Subsequence of $S$
begin
    $n \longleftarrow$ LENGTH($S$)
    Create an array $LPS$ of size $n \times n$
    for $i \in [n]$ do
        $LPS[i, i] \longleftarrow (1, [S[i]])$
    for $i \in [n-1]$ do
        if $S[i] == S[i+1]$ then
            $LPS[i, i+1] \longleftarrow (2, [S[i], S[i+1]])$
        else
            $LPS[i, i+1] \longleftarrow (1, [S[i]])$
    for $d = 2, \ldots, n$ do
        for $i \in [n+1-d]$ do
            $j \longleftarrow d+i-1$ if $S[i] == S[j]$ then
                $LPS[i, j] \longleftarrow (2 + LPS[i-1, j-1][1], [S[i]] + +LPS[i-1, j-1][2] + +[S[d+i-1]])$
            else
                if $LPS[i+1, j][1] \geq LPS[i, j-1][1]$ then
                    $LPS[i, j] \longleftarrow LPS[i+1, j]$
                else
                    $LPS[i, j] \longleftarrow LPS[i, j-1]$
    return $LPS[1, n][2]$
```

Time Complexity: The first two for loops take $O(n)$ time each. In the second for loop in each iteration we are running over $n+1-d = O(n)$ many iterations each of which takes a constant time. Hence it takes $O(n^2)$ time. Therefore the above algorithm takes $O(n^2)$ time to find the longest palindromic sequence. ■

Problem 6 P8 (25 marks)

The purpose of this question is to extend the closest-points algorithm seen in the first lecture, to give an $O\left(n\log^2 n\right)$ algorithm for finding the closest pair of points in 3 dimensions. All points in this question are in $\mathbb{R}^3$.

(a) (5 marks) Prove that, if all points are at least distance δ apart, a cube with each dimension of size 2δ contains at most a constant (say k) number of points.

(b) (10 marks) You are now given 2 sets of points S_1 and S_2, each containing n points. The distance between any pair of points in S_1 is at least δ, and further, each point in S_1 has z-coordinate in $[0,\delta]$. Similarly, the distance between any pair of points in S_2 is at least δ, and each point in S_2 has z-coordinate in $[-\delta, 0]$.

Extend the algorithm discussed in class to give an $O(n\log n)$-time algorithm for finding the closest pair of points in $S_1 \cup S_2$. Note that, by the first part of the question, any cube with each dimension at most 2δ, contains at most $2k$ points from $S_1 \cup S_2$.

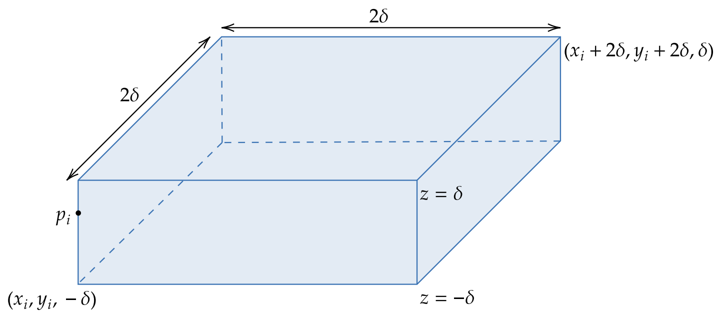
(c) (10 marks) Given a set S of n points in $\mathbb{R}^3$, now give an $O\left(n\log^2 n\right)$-time algorithm to find the closest pair of points.

Solution:

(a) Divide the cube into small cubes with each dimension $\frac{\delta}{4}$. There are total 64 such small cubes. So there can be at most 64 points inside the cube with each dimension 2δ. Cause if there are more than 64 points in side the cube then there exists at least one small cube containing at least 2 points.

Since any two points are at least δ distance apart in each of this cube, if there are two points inside one small cube then their distance will be at most $\frac{\delta}{4}\sqrt{3} < \delta$. This is not possible. Hence contradiction. Therefore there can be at most 64 points inside a cube with each dimension 2δ. So a cube with each dimension of size 2δ contains at most a constant number of points.

(b) We assume the points are in general position. So first in $S_1 \cup S_2$ we sort the points with respect to their $x-$coordinates and $y-$coordinates. Let those new sorted sets are called T_x and T_y respectively. Each points in $S_1 \cup S_2$ has $z-$coodinate in $[-\delta, \delta]$. So now after sorting the points of $S_1 \cup S_2$ we start from left most point and for any point $p_i \in S_1 \cup S_2$ we take the cube with two diagonal points $(x_i, y_i, -\delta)$, $(x_i + 2\delta, y_i + 2\delta, \delta)$ which is a cube with each dimension 2δ (Like in the picture)



Now by part (a) we know there are at most 64 points in this cube. For that we have to check the next 64 points in both T_x and T_y since we have to ensure we have checked all the points inside the cube. Hence the algorithm is:

Algorithm 4: FIND-CLOSEST-2-SETS(S_1, S_2, δ)

```
Input: Pair of sets $S_1, S_2$ each with $n$ points with in each set any two points have distance at least $\delta$ and
       further $z-$coordinate of points in $S_1$ in $[0,\delta]$ and of points in $S_2$ in $[-\delta, 0]$.
Output: Closest pair of ponts, $(P_i, P_j, \delta_0)$ where $\delta_0 = d(P_i, P_j)$ with $P_i, P_j \in S_1 \cup S_2$
1  begin
2      $T_x \longleftarrow S_1 \cup S_2$ sorted by $x-$coordinate
3      $T_y \longleftarrow S_1 \cup S_2$ sorted by $y-$coordinate
4      for $i = 1, \ldots, 2n$ do
5          $U_x \longleftarrow$ Next 64 points in $T_x$
6          $U_y \longleftarrow$ Next 64 points in $T_y$
7          for $\hat{P} \in U_x \cup U_y$ do
8              if $d(P, \hat{P}) \leq \delta$ then
9                  $\delta \longleftarrow d(P, \hat{P})$
10                 $(P_1, P_2) \longleftarrow (P, \hat{P})$
11     return $(P_1, P_2, \delta)$
```

In the algorithm instead od checking for a point with distance less than δ we are checking pair of points with distance less than or equal to δ this is because we don't have any pair of points in S_1 or S_2 which are at least δ distance. So if there is no pair of points in $S_1 \cup S_2$ with distance less than δ we still get a pair of points with distance equal to δ and that pair points have the shortest distance.

Time Complexity: Here to sort the points in $S_1 \cup S_2$ it takes $O(n \log n)$ time since there are total $2n$ points in $S_1 \cup S_2$. Now the for loop runs for $2n$ iterations and in each iterations it is checking for constant number of points and for each point it is doing constant time operations. So each iteration of the loop takes $O(1)$ time and the for loop takes in total $2nO(1) = O(n)$ time. Hence the whole algorithm takes $O(n \log n)$ time.

(c) Like in the case of $\mathbb{R}^2$ algorithm we will first divide the set of all points into two roughly equal parts which we can do by sorting the points with respect to their $z-$coordinates. Let $\overline{z}$ be the $\left\lfloor \frac{n}{2} \right\rfloor^{th}$ highest $z-$coordinate. We we partition S into points with $z-$coordinate $< \overline{z}$ and $\geq z$. Call these sets S_U and S_D respectively. Then run the algorithm recursively on each of those sets. And thus we get a closest pair of points and their distance in each of the smaller sets. So we take the pair of points with the distance equal to the minimum of those two distances. Call this distance δ. So we only need to check the points from S with $z-$coordinate $\in [\overline{z} - \delta, \overline{z} + \delta]$. Call this set of points as $T = S_U^T \sqcup S_D^T$. So

$$S_U^T = \{P_i \in S_U : z_i \in [\overline{z}, \overline{z} + \delta]\} \qquad S_D^T = \{P_i \in S_D : z_i \in [\overline{z} - \delta, \overline{z}]\}$$

Now the situation is almost like the situation in part (b). So we have two sets of points S_U^T, S_D^T each set has at most $\frac{n}{2}$ points and any two points in any one of the sets has distance at least δ. Then we need to find the closest pair of points in the union of those two sets.

The only change from the situation in part (b) is the now the points in S_U^T have $z-$coordinate in $[\overline{z}, \overline{z}+\delta]$ instead of $[0, \delta]$ and similarly the points in S_D^T have $z-$coordinate in $[\overline{z} - \delta, \overline{z}]$ instead of $[-\delta, 0]$. Also like in the case of part (b) we don't have to worry about not having a pair of points with their distance $= \delta$. So we have to only look for points with distance less than δ. Therefore we need to do slight change in the algorithm above and then we are good to go.

Algorithm 5: FIND-CLOSEST(S)

```
    Input: Set of $n$ points, $S = \{P_i(x_i, y_i, z_i) \mid x_i, y_i, z_i \in \mathbb{R},\ \forall\ i \in [n]\}$.
    Output: Closest pair of ponts, $(P_i, P_j, \delta)$ where $\delta = d(P_i, P_j)$
 1  begin
 2      if $|S| \leq 130$ then
 3          Solve by Brute Force (Consider every pair of points)
 4      $S^z \longleftarrow S$ sorted by $z$−coordinate
 5      $\bar{z} \longleftarrow \lfloor \frac{n}{2} \rfloor$ highest $z$−coordinate
 6      $S_U \longleftarrow \{P_i \mid z_i > \bar{z},\ \forall\ i \in [n]\}$, $S_D \longleftarrow \{P_i \mid z_i \leq \bar{z},\ \forall\ i \in [n]\}$
 7      $(P_1^L, P_2^L, \delta^L) \longleftarrow$ FIND-CLOSEST($S_U$), $(P_1^R, P_2^R, \delta^R) \longleftarrow$ FIND-CLOSEST($S_D$)
 8      $\delta_{min} \longleftarrow \min\{\delta^L, \delta^R\}$
 9      if $\delta_{min} < \delta^L$ then
10          $P_1 \longleftarrow P_1^R$, $P_2 \longleftarrow P_2^R$
11      else
12          $P_1 \longleftarrow P_1^L$, $P_2 \longleftarrow P_2^L$
13      $S_U^T \longleftarrow \{P_i \mid z_i - \bar{z} \leq \delta_{min}\}$, $S_D^T \longleftarrow \{P_i \mid \bar{z} - z_i \leq \delta_{min}\}$
14      $(P_1', P_2', \delta) \longleftarrow$ FIND-CLOSEST-2-SETS-MOD$\left(S_U^T, S_D^T, \bar{z}, \delta_{min}\right)$
15      if $\delta < \delta_{min}$ then
16          return $(P_1', P_2', \delta)$
17      else
18          return $(P_1, P_2, \delta_{min})$
```

Where we describe he FIND-CLOSEST-2-SETS-MOD algorithm:

Algorithm 6: FIND-CLOSEST-2-SETS-MOD(S_1, S_2, z, δ)

```
    Input: Pair of sets $S_1, S_2$ each with $n$ points with in each set any two points have distance at least $\delta$ and
           further $z$−coordinate of points in $S_1$ in $[z, z + \delta]$ and of points in $S_2$ in $[z - \delta, z]$.
    Output: Closest pair of ponts, $(P_i, P_j, \delta_0)$ where $\delta_0 = d(P_i, P_j)$ with $P_i, P_j \in S_1 \cup S_2$
 1  begin
 2      $T_x \longleftarrow S_1 \cup S_2$ sorted by $x$−coordinate, $T_y \longleftarrow S_1 \cup S_2$ sorted by $y$−coordinate
 3      for $i = 1, \ldots, 2n$ do
 4          $U_x \longleftarrow$ Next 64 points in $T_x$, $U_y \longleftarrow$ Next 64 points in $T_y$
 5          for $\hat{P} \in U_x x \cup U_y$ do
 6              if $d(P, \hat{P}) < \delta$ then
 7                  $\delta \longleftarrow d(P, \hat{P})$
 8                  $(P_1, P_2) \longleftarrow (P, \hat{P})$
 9      return $(P_1, P_2, \delta)$
```

Time Complexity: Let the algorithm takes $T(n)$ to find closest pair of points from n points. To sort the set of points with respect to their z−coordinate takes $O(n \log n)$ time. Now forming the sets S_U and S_D takes $O(n)$ time. Now $|S_U|, |S_D| \leq \frac{n}{2}$. Hence in lines 8,9 it takes $T\left(\frac{n}{2}\right)$ time. Now line 11-14 it takes constant time. Then again forming the sets S_U^T, S_D^T takes linear time. Then to run the algorithm FIND-CLOSEST-2-SETS-MOD takes $O(n \log n)$ time. Therefore total time taken is

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

So by Problem 1: P3 (i) $T(n) = O(n \log^2 n)$. Hence it takes $O(n \log^2 n)$ to find closest pair of points in $\mathbb{R}^3$.

■

[I discussed the solution with Shubham]

Problem 7 P9 (10 marks)

This problem relates to one of the questions asked in class. For any $p, q \geq 1$, and any points x, y, and $z \in \mathbb{R}^2$, prove or disprove the following:

$$\|x-y\|_p \leq \|x-z\|_p \Leftrightarrow \|x-y\|_q \leq \|x-z\|_q$$

That is, prove or disprove that y is closer to x than z in the L_p distance metric if and only if it is closer to x in the L_q distance metric As usual, $\|x-y\|_p = ((x_1-y_1)^p + (x_2-y_2)^p)^{1/p}$.

Solution: This question reduces to showing for any two points $x, y \in \mathbb{R}^2$ if the following is true of not

$$\|x\|_p \leq \|y\|_p \iff \|x\|_q \leq \|y\|_q$$

This is not true. Take $p = 1$, $q = 2$. And $x = (1, 0)$ and $y = (0.55, 0.55)$. Then $\|x\|_1 = \|x\|_2 = 1$. But $\|y\| = 1.1$, $\|y\| = 0.55\sqrt{2} < 1$. Hence $\|x\|_1 < \|y\|_1$ but $\|x\|_2 > \|y\|_2$. Hence the above claim is not true. ■

[I discussed the counter example with Soumyadeep]