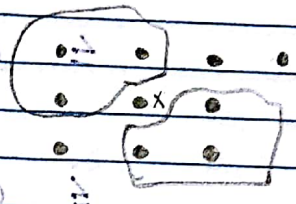
KAUTUK R .D FSAE
50247648

Problem Set 2

1) In Selection algorithm, if we partition the elements into groups of 3 each then the algorithm does not have linear solution.

Proof :-



we assume that the median of 3 numbers is the second smallest one.

when we partition the elements into groups of size 3, then at least half of the medians found are greater than the median-of-median x. This, at least half of the $\lceil n/3 \rceil$ groups contribute 2 elements that are greater than x, except for the last group that may have fewer than 3 elements if 3 does not divide n exactly, and the group containing x itself. So the number of elements greater than x is atleast

$$2\left(\left\lceil \frac{1}{2} \times \frac{n}{3} \right\rceil - 2\right)$$

$$\geq \frac{n}{3} - 4$$

Similarly, the number of elements that must be less than x is at least $\frac{n}{3} - 4$. So, In worst case the Select Algorithm will be recursively called on at most $n - \left(\frac{n}{3} - 4\right) = \frac{2n}{3} + 4$

Thus, the recurrence relation that we get is.

$$T(n) \leq T(n/3) + T(2n/3 + 4) + O(n)$$

We can show that running time is non-linear by substitution we assume that,

$T(n) \leq cn$ for some large constant c

we also assume that, $O(n) = an$ for some constant a

The recurrence relation is given by,

$$T(n) \leq c(n/5) + c\left(\frac{2n}{3} + 4\right) + an$$

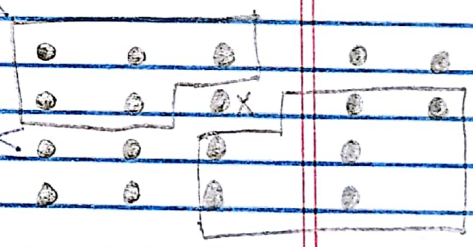
$$\leq \frac{cn}{3} + \frac{2cn}{3} + 4c + an.$$

$$\leq cn + 4c + an$$

which is at most cn if $-4c - an \geq 0$

The above inequality does not hold true, as we do not get linear solution. Therefore it is not possible to get $O(n)$ time algorithm if we partition the elements with group of size 3

Group size 4 :



We assume that the median of 4 numbers is the second smallest one. When we partition the elements into groups of size 4, then at least half of the medians are greater than the median of medians x. Thus at least half of $\lceil n/4 \rceil$ groups contribute 3 elements that are greater than x, except for the last group that have fewer than 4 elements as 4 does not divide n exactly, and the group containing x itself. So, the number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2} \times \frac{n}{4} \right\rceil - 2\right)$$

$$\geq \frac{3n}{8}$$

Similarly the number of elements that must be less than x is at least $\frac{n}{4} - 4$ In worst case, the select algorithm is recursive

called on at most $n - \left(\frac{n}{4} - 4\right) = \frac{3n}{4} + 4$

Therefore the recurrence relation is

$$T(n) \leq T(n/4) + T\left(\frac{3n}{4} + 4\right) + \Theta(n)$$

As explained previously, for groups of 3, we get that the size of two recursive calls can be $\frac{n}{4}$ and $\frac{3n}{4}$ which sums up to n. The running time of the algorithm is not linear but $T(n) = \Theta(n \log n)$. We can prove this by substitution

$$T(n) \leq c\left(\frac{n}{4}\right) + c\left(\frac{3n}{4} + 4\right) + an$$

$$\leq \frac{cn}{4} + \frac{3cn}{4} + 4c + an$$

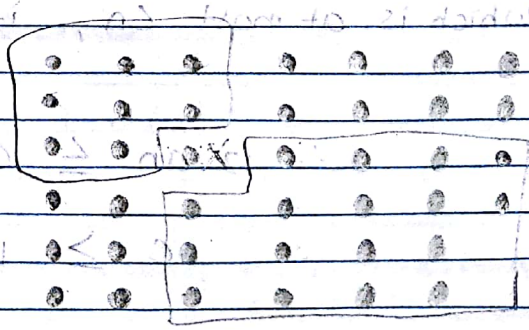
$$\leq cn + 4c + an$$

which is at most cn if, $4c + an \leq 0$
The above inequality does not hold true, and hence we do not get linear solution.

Therefore, if we divide the input elements to groups of 4 elements, the algorithm will not run in $O(n)$ time.

Group size 6:
Here we assume the median to be the third smallest element. Similarly to our group size 3 & 4, the number of elements greater than x is at least

$$4\left[\frac{1}{2}\left\lceil\frac{n}{6}\right\rceil - 2\right]$$



$\geq \frac{n}{3} - 8$

Similarly, the number of elements that must be less than x is at least $\frac{n}{4} - 6$

In worst case, the select algorithm is recursively called on at most

$$n - \left(\frac{n}{4} - 6\right) = \frac{3n}{4} + 6$$

The recurrence relation is

$$T(n) \leq T\left(\frac{n}{6}\right) + T\left(\frac{3n}{4} + 6\right) + O(n)$$

we assume, $T(n) \leq Cn$ for some large constant C

Then for by induction,

$$T(n) \leq \frac{cn}{6} + c\left(\frac{3n}{4} + 6\right) + an$$

$$\leq \frac{cn}{6} + \frac{3cn}{4} + 6c + an$$

$$\leq \frac{11cn}{12} + 6c + an$$

$$\leq cn + \left(-\frac{cn}{12} + bc + an\right)$$

which is at most cn, if $-\frac{cn}{12} + bc + an \leq 0$

$$\therefore 12an \leq cn - 72c$$

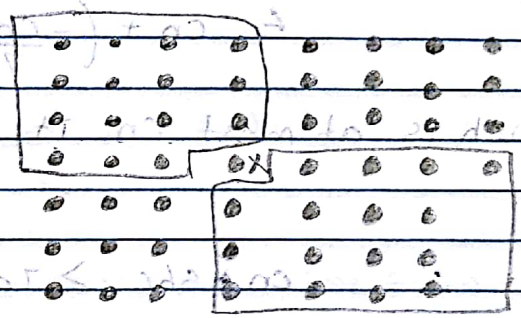
$$\therefore c \geq \frac{12an}{n - 72}$$

5024 7648

Therefore, for any $n \geq 72$ and choosing c accordingly will give linear solution for the recurrence.

Therefore, if we divide the input elements to groups of 5 elements, the select algorithm will run in $\Theta(n)$-time.

Group size 7:



We assume the median of 7 numbers to be fourth smallest number.
When we partition the elements into groups of size 7, then at least half of the medians are greater than median of medians x.
Thus at least half of $\lceil n/7 \rceil$ contribute to 4 elements that are greater than x, except the last group which may have fewer than 7 elements as 7 does not divid n exactly.
The number of elements greater than x is at least

$$4\left\lceil \frac{1}{2} \times \frac{n}{7} - 2 \right\rceil$$

$$\geq \frac{2n}{7} - 8$$

The number of elements less than x is at least $\frac{2n}{7} - 8$

So, In worst case the select algorithm will be called recursively for at most $n - \left(\frac{2n}{7} - 8\right) = \frac{5n}{7} + 8$

Therefore, the recurrence relation is

$$T(n) \leq T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7} + 8\right) + O(n)$$

we assume that, $T(n) \leq cn$ & $O(n) = an$ for large constants c and a

$$\therefore T(n) \leq c\left(\frac{n}{7}\right) + c\left(\frac{5n}{7} + 8\right) + an$$

$$\leq \frac{cn}{7} + \frac{5cn}{7} + 8c + an$$

$$\leq cn + \left(-\frac{cn}{7} + 8c + an\right)$$

which is at most cn if, $-\frac{cn}{7} + 8c + an \leq 0$

$$\therefore \quad cn - 56c \geq 7an.$$

$$\therefore \quad c \geq 7a\left(\frac{n}{n-56}\right)$$

From above inequality, for any $n > 56$ will give linear solution for the recurrence.

Therefore, If we divide the number of elements in groups of 7, the select algorithm will run in $O(n)$ time

Group size 9.

We assume the median of 9 numbers to be the fifth smallest element when we partition the elements into groups of size 9, then at least half of the medians are greater than the median-of-medians x

Thus, atleast half of the $(n/9)$ groups contribute 5 elements that are greater than x, except for the last group which may have fewer than 9 elements, as n is not divided exactly by 9, and the group containing x itself. So the number of elements greater than x is

$$5\left(\frac{1}{2}\frac{n}{9} - 2\right) \geq \frac{5n}{18} - 10$$

Similarly, the number of elements that must be less than x is at least $\frac{5n}{18} - 10$. So, In worst case the select algorithm will be called recursively on at most

$$n - \left(\frac{5n}{18} - 10\right) = \frac{13n}{18} + 10$$

The recurrence relation is,

$$T(n) \leq T\left(\frac{n}{9}\right) + T\left(\frac{13n}{18} + 10\right) + O(n)$$

we assume that $T(n) \leq cn$ and $O(n) = an$ for some large constant c and a

Therefore by induction, we get

$$T(n) \leq c\left(\frac{n}{9}\right) + c\left(\frac{13n}{18} + 10\right) + an.$$

$$\leq \frac{cn}{9} + \frac{13cn}{18} + 10c + an.$$

$$\leq \frac{15cn}{18} + 10c + an$$

$$= cn + \left(\frac{-3cn}{18} + 10c + an\right)$$

which is at most cn if $\frac{-3cn}{18} + 10c + an \leq 0$.

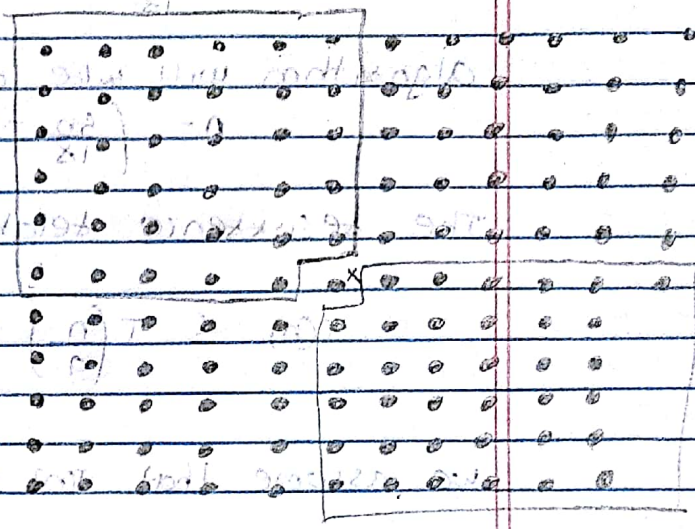
$$\therefore \quad -3cn + 180c + 18an \leq 0.$$

$$\therefore \quad -cn + 60c + 6an \leq 0.$$

$$c \geq 6a\left(\frac{n}{n-60}\right)$$

Hence, for any $n > 60$ the recurrence will give linear solution.

Therefore the select algorithm will run in $O(n)$ time if we divide the elements into groups of 9.



Group size 11:
we assume the median of 11 elements to be the sixth term. when we partition the elements in groups of 11, then at least half of the medians are greater than the median-of-medians x. Thus, at least half of $\lceil n/11 \rceil$ contribute 6 elements that are greater x, except for the last group which may have fewer than 11 elements as n is not divided by 11 exactly, and the group containing x itself.
So the number of elements that are greater than x is.

$$6\left(\frac{1}{2}\times\frac{n}{11}-2\right) \geq \frac{3n}{11} - 12.$$

Similarly, the number of elements that are less than x is $\frac{3n}{11} - 12$. So, In worst case the select algorithm will be recursively called at most $n - \left(\frac{3n}{11} - 12\right)$

$$= \frac{8n}{11} + 12$$

Therefore, the recurrence relation is

$$T(n) \leq T\left(\frac{n}{11}\right) + T\left(\frac{8n}{11} + 12\right) + O(n)$$

we assume that $T(n) \leq Cn$ for some large constant C and $O(n)$ is bound by an for all $n > 0$

$$\therefore T(n) \leq c\left(\frac{n}{11}\right) + c\left(\frac{8n}{11} + 12\right) + an$$

$$\leq \frac{cn}{11} + \frac{8cn}{11} + 12c + an$$

$$\leq cn + \left(\frac{-2cn}{11} + 12c + an\right)$$

which is at most cn if, $\frac{-2cn}{11} + 12c + an \leq 0$

$$\therefore -2cn + 12 \times 11.c + 11an \leq 0$$

$$\therefore cn - 66c \geq \frac{11}{2} an$$

$$\therefore c \geq \frac{11}{2} a \times \left(\frac{n}{n-66}\right)$$

Therefore for all $n > 66$, the recurrence will give linear solution.

Hence, the select algorithm will run in $O(n)$-time, if we divide the elements in groups of 11.

Summary:

Group size	$O(n)$-time ?	condition
3	NO	—
5	NO	—
6	Yes	$n \geq 72$
7	yes	$n > 56$
9	Yes	$n > 60$
11	yes	$n > 66$

6) let us assume the two sub matrices to be

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad B = \begin{bmatrix} s & t \\ u & v \end{bmatrix}$$

and the resultant matrix, $C = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$

According to strassen's matrix multiplication algorithm, we reduce the number of recursive calls to 7 and instead use additions/ subtractions

let, $A_1 = a$ $\qquad B_1 = t - v$

$A_2 = a + b$ $\qquad B_2 = v$

$A_3 = c + d$ $\qquad B_3 = s$

$A_4 = d$ $\qquad B_4 = u - s$

$A_5 = a + d$ $\qquad B_5 = s + v$

$A_6 = b - d$ $\qquad B_6 = u + v$

$A_7 = a - c$ $\qquad B_7 = s + t$

Now we compute the 7 multiplication required - (2)

$R_1 = A_1 \times B_1 = a \times (t - v)$

$R_2 = A_2 \times B_2 = (a + b) \times v$

$R_3 = A_3 \times B_3 = (c + d) \times s$

$R_4 = A_4 \times B_4 = d \times (u - s)$

$R_5 = A_5 \times B_5 = (a + d)(s + v)$

$R_6 = A_6 \times B_6 = (b - d)(u + v)$

$R_7 = A_7 \times B_7 = (a - c)(s + t)$

The resultant matrix C can be computed as follows - (3)

$e = R_4 + R_5 + R_6 - R_2 = a.s + bu$

$f = R_1 + R_2 = a.t + bv$

$g = R_3 + R_4 = c.s + du$

$h = R_5 + R_1 - R_3 - R_7 = ct + dv$

From equations in step ③ and ②, we get that.

$as + bu = R_4 + R_5 + R_6 - R_2$

comparing P terms with R terms in eqn ②, we get.

$R_1 = P_4 = a(t-v)$ $\quad R_5 = P_3 = (a+d)(s+v)$

$R_2 = P_5 = v(a+b)$ $\quad R_6 = P_2 = (b-d)(u+v)$

$R_3 = P_6 = s.(c+d)$ $\quad R_7 = P_1 = (a-c)(s+t)$

$R_4 = P_7 = d.(u-s)$

$\therefore \quad as + bu - e = R_4 + R_5 + R_6 - R_2$

$= P_7 + P_3 + P_2 - P_5$

$\therefore$

$at + bv = f = R_1 + R_2$

$= P_4 + P_5$

$cs + du = g = R_3 + R_4$

$= P_6 + P_7$

$ct + dv = h = R_5 + R_1 - R_3 - R_7$

$= P_3 + P_4 - P_6 - P_1$

Solutions,

$as + bu = P_2 + P_3 + P_7 - P_5$

$at + bv = P_4 + P_5$

$cs + du = P_6 + P_7$

$ct + dv = P_3 + P_4 - P_1 - P_6$ //

5) Given an array $A = \{a_1, a_2, a_3, \ldots, a_n\}$ of n unsorted numbers we can find the total number of inversion pair by a naive method which has $O(n^2)$ running time. The pseudo code for the method is

```
for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
        if (A[i] > A[j])
            inversion_count++;
```

Here, n = length of the array A. For each element in array A we count the number of elements which are on right side of the current element and are smaller than it.

Method 2: The problem can be solved by divide and conquer paradigm. We will modify the merge sort to count the number of inversions.

Suppose we divide the array in two sub-arrays, $\{a_1, a_2, \ldots a_{n/2}\}$ $\{a_{n/2+1}, a_{n/2+2}, \ldots, a_n\}$. We assume that we know the number of inversion in the left half of array and right half of the array let them be 'inversion1' and 'inversion2'.

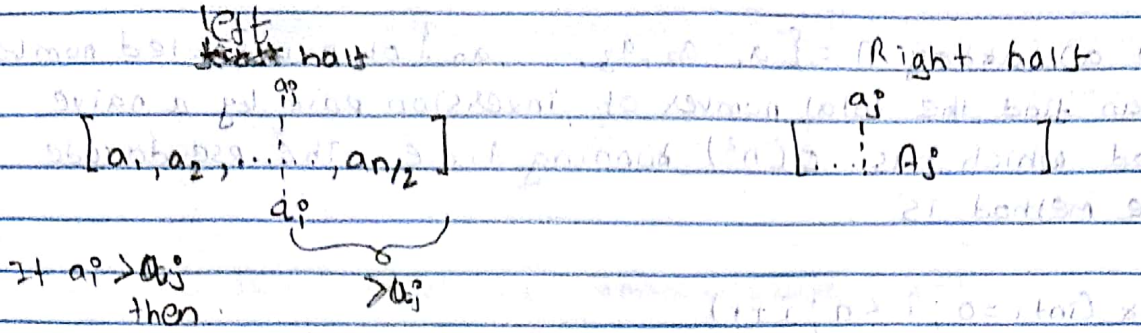
Now, the kinds of inversion pairs that are left out can be found in the merge step. Therefore the total count of inversions is the summation of, left sub-array inversion, right sub-array inversion and the inversions in merge step.

$[a_1, \ldots, a_{n/2}]$ $[a_{n/2+1}, \ldots, a_n]$

(inversion 1) (inversion 2)

inversions in each half

we will now understand, the calculation of inversion pairs in merge step.



In merge process, let i denote the indexing in left subarray and j for right subarray. At a particular step in merge process, if $A[i] > A[j]$ then there are $(n/2$ elements $- i)$ inversion pairs. This is because, if any term $A[i] > A[j]$ then all remaining term from $A[i+1]$ to $A[n/2]$ will be greater than $A[j]$, as the two array are sorted.

These are the total inversion pairs will be inversion1 + inversion2 + merge step inversions

Since counting inversions takes only constant amount of time within the merge sort. The running time of the algorithm is same as that of the merge sort i.e $\theta(n \log n)$

Pseudo code for the above algorithm:

```
count-inversion (A, p, r)
inversion = 0;
if (p < r)
    q = [(p + r)/2];
    inversion = inversion + count-inversion (A, p, q);
    inversion = inversion + count-inversion (A, q+1, r);
    inversion = inversion + count_merge inversions (A, p, q, r);

return inversion;


count_merge_inversions (A, p, q, r)
larray_length = q - p + 1;
rarray_length = r - q;
left_array = [larray_length + 1];
```

```
right_array = [x array_length + 1];

for (int i=0; i < left_array_length; i++)
    left_array[i] = A[p+i-1];

for (int j=0; j < right_array_length; j++)
    right_array[j] = A[q+j];
i=1; j=1;
for (int k=p; k < r; k++)
    if (left_array[i] > right_array[j])
        inversion = inversion + left_array_length - i + 1;

    if (left_array[i] ≤ right_array[j])
        i = i+1;

    else
        j = j+1;

return inversion;
```

4) $S = \{a_1, \ldots, a_n\}$ $B \Rightarrow$ real value

we need to check whether these exists 3 distinct numbers such that $a_i + a_j + a_k = B$

Method 1 : we can loop over to generate all possible a_i, a_j, a_k and compare the sum with B. This method will need 3 different loops nested in each other to generate the 3 numbers.
Hence the running time of this method is $\theta(n^3)$

```
for (int i = 0; i < array-length - 2; i++)
    for (int j = 0; j < array-length - 1; j++)
        for (int k = 0; k < array-length; k++)
            if (S[i] + S[j] + S[k] == B)
                return true;

return false;
```

For large n ; this method is not feasible.

Method 2 : we first sort the given array, this can be done using merge sort which has a running time of $\theta(n \log n)$
Then we select one element from the array and fix it, we can then find the other two elements by looping the array and checking from S[i+1] to S[length - 1]

- let i → be the index of element selected from 0 to length -2
Here we loop until length-2 is because we want to find 3 elements and if we do not find it until length -2 then these do not exist and distinct numbers that satisfies $a_i + a_j + a_k = B$
- let j → be i+1 → the next element in array
k → be the last element in array

- If S[i] + S[j] + S[k] = B then we return true
- If S[i] + S[j] + S[k] > B then we decrease the K because we know that the array is sorted in ascending order and the k^{th} term is taking the sum > B

- If $S[i] + S[j] + S[k] < B$ then we increase the j^{th} index to move closer to B.

we repeat the above 3 steps until $j < k$ and if we donot find the 3 numbers then we conclude that these do not exist 3 numbers such in set $S = \{a_1, \ldots, a_n\}$ such that.

$a_i + a_j + a_k = B$.

Pseudo code:

```
find_triplets (Array, B)
array_length = array.length;
i = 0; j = 0; k = 0;

while ( i <= array_length - 2)
    j = i + 1;
    k = array_length - i + 1;

    while (j < k)
        if ( Array[i] + Array[j] + Array[k] == B)
            return true;

        if (Array[i] + Array[j] + Array[k] > B)
            k = k - 1;
        else
            j = j + 1;

    return false;
```

The running time of the above algorithm is $\theta(n^2)$ as we only have 2 nested loops.

3) Algorithm for weighted median.

- In order to compute the weighted median of n elements, we first sort the elements and sum up the weights of the elements until we find the median.

```
sum = 0, int i
for (int i = 0; i < n; i++)
    sum = sum + w[i]
    if (sum > 1/2)
        break.
```

median = i-1 element.

∴ we say that, sum of all weights of all elements less than x_k is

$$Sum = \sum_{x_i < x_k} w_i \quad \text{--- (1)}$$

we can prove this by induction. The initial case is the first iteration of the loop when sum = 0. Since the elements are sorted, $x_i < x_k$. which means that in every iteration of the loop the sum increases by the weight of the element.

we know that $\sum w_i = 1$, so the loop will terminate when the sum of all elements is 1. when the loop terminates the (i-1) element is the weighted median.

let sum' be the value of sum at the start of the next to last iteration of the loop

$$\therefore \quad sum = sum' + w_{k-1}$$

Since the sum with the last element does not meet the condition, we know that.

$$sum' + w_{k-1} < 1/2$$

The above equation means, $\sum_{x_i < x_k} w_i < 1/2$

If the loop has 0 elements then also the above inequality holds true as sum = 0 is $< 1/2$.

Hence we proved that first condition of weighted median

$$\sum_{x_i < x_k} w_i^* < 1/2 \quad \text{--- (2)}$$

Now, we know that $\sum w_i^* = 1$

$$\therefore \sum_{x_i^* < x_k} w_i^* + w_k + \sum_{x_i^* > x_k} w_i^* = 1$$

$$\therefore \sum_{x_i^* > x_k} w_i^* = 1 - \left[\sum_{x_i^* < x_k} w_i^* - w_k \right]$$

$$= 1 - \text{Sum} + w_k \quad \text{--- from (1)} \quad \text{--- (3)}$$

From (2), we get that

$$\sum_{x_i^* < x_k} w_i^* + w_k \geq 1/2$$

$$\therefore \sum_{x_i^* < x_k} w_i^* \geq 1/2 - w_k \quad \text{--- (4)}$$

From eq (3) & (4), we get.

$$1 - \text{Sum} - w_k \leq 1/2$$

$$\therefore \sum_{x_i^* > x_k} w_i^* \leq 1/2.$$

Hence we proved the second condition of weighted median,

Therefore we can say that x_k exists

3) b) Weighted median algorithm :-

- find the median x_k of the n elements and then partition around it.
- compute the total weights of each half, if the weights of the two halves are less than $1/2$ each then the weighted median is x_k.
- Else, the weighted median will be in half whose sum is $> 1/2$
- The half whose sum is $< 1/2$ is lumped into the weight of x_k.
- Continue searching in the half whose sum $> 1/2$.

The pseudo code is given below for $A = \{x_1, x_2, \ldots, x_n\}$

```
weighted_median (A, sum)
n = length (A)
sum = 0
median = Median (A)
left_half = A[i] < median
right_half = A[i] >= median
right_median = 0
If n = 1
    return A[i]

for (int i = 0; i < n; i++)
    If A[i] < median
        right_median = right_median + wi
        left_half = A[i]
    else
        right_half = A[i]

If right_median + sum > 1/2
    weighted_median (left_half, sum)
else
    weighted_median (right_half, right_median)
```

we initially call the procedure weighted median (A, 0) where A is the sorted list of elements and sum = 0.

Since in each recursive call, we compute on at the most half of the elements + median element, the recurrence of the algorithm is given by

$$T(n) = T(n/2) + \Theta(n)$$

The running time of recursive call is $\Theta(n)$. therefore the worst case running time of weighted median algorithm is $T(n) = \Theta(n)$ //

2) Closest point pair problem - 3D

In order to derive an algorithm, we will simply extend the 2D algorithm. The pointset P is represented by $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, y_3, \ldots, y_n\}$ and $Z = \{z_1, z_2, \ldots, z_n\}$

- We first sort the point by X, Y and Z coordinate. This takes $O(n \log n)$ time i.e by merge sort

- If n is small, ≤ 4 then find shortest pair directly. This will be done in $O(1)$ time

- Divide the point set P into two parts by a vertical plane such that the plane divides P into P_L (points on the left) and P_R (points on the right). Therefore, $P_L = \lceil n/2 \rceil$ and $P_R = \lceil n/2 \rceil$

- We know that X is sorted, we can draw plane L between $x_{(n/2)}$ and $x_{(n/2)+1}$. We can now scan X and collect points into P_L and P_R. This takes $O(1)$ time

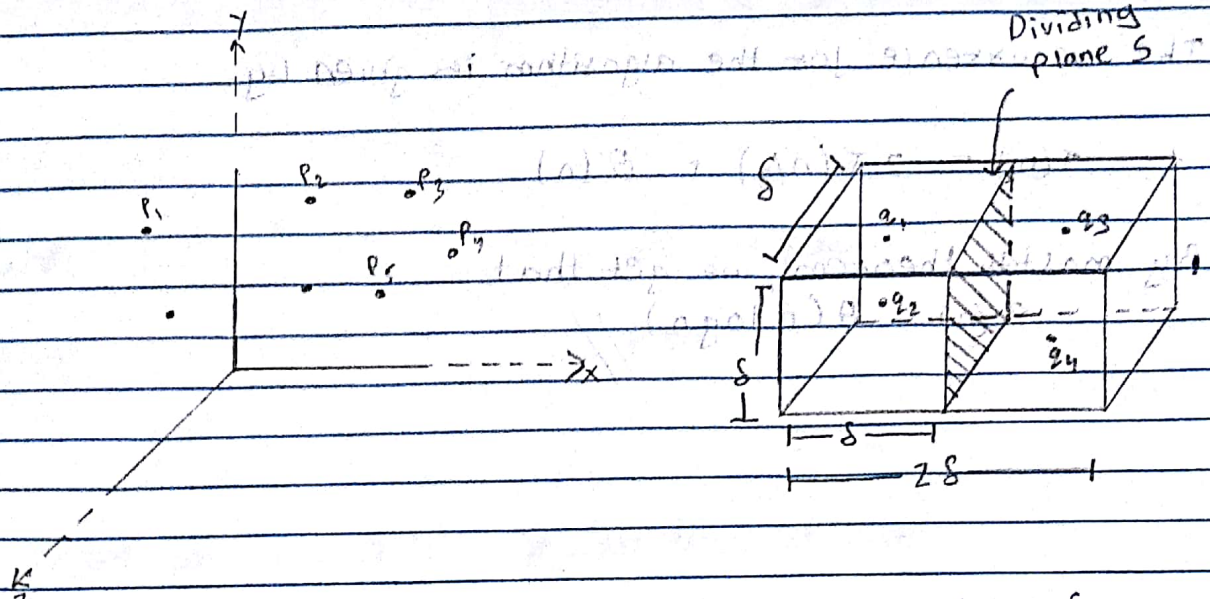
- Now, find the closest point pair in P_L such that $\delta_L = dist(P_{Li}, P_{Lj})$ where P_{Li}, P_{Lj} be the point pair with the smallest distance in P_L

- Similarly we find the closest point pair in P_R such that $\delta_R = dist(P_{Ri}, P_{Rj})$ where P_{Ri}, P_{Rj} be the point pair with smallest distance in P_R.

- Then find the minimum of two smallest distances found above. $\delta = \min(\delta_L, \delta_R)$. This also takes $O(1)$ time.

- Now, the solution of the given problem must be in one of the three cases mentioned below

1) The closest pair are both in P_L
2) The closest pair are both in P_R.
3) The only condition for which we need to find the closest pair is that, when one point is in P_L and the other in P_R.



From the above diagram, let the width of cuboid be 2δ and height & δ and depth of δ

- let $P' = \{q_1, q_2, \ldots, q_n\}$ be the points in cuboid. let Y' be y-coordinates of points in P' in increasing order. Similarly let Z' be z-coordinates of points in P'.

Since Y' & Z' are already sorted we scan Y and Z and only include points that are in cuboid. This will take $O(n)$ time for each process.

- From the diagram, we say that there can be at the most 16 points which could have dist $< \delta$. Therefore we compute the distance between point pair in P' such that 1 point is compared to 15 other points in the cuboid. let the minimum distance from this step be δ'.

- If $\delta' < \delta$, the distance computed in the above step is the shortest distance and hence we find the closest pair.

- If $\delta' > \delta$, the the minimum distance found in P_L or P_R will be of the shortest distance

- Since this algorithm divide the problem into half and calls recursively with all other operation taking $O(n)$ running time

The recurrence for the algorithm is given by,

$$T(n) = 2T(n/2) + \theta(n)$$

By master theorem, we get that

$$T(n) = \theta(n \log n)$$ //