

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 $\lceil \frac{n}{3} \rceil$)

groups of size 3, then at least half of the medians found are greater than the median-of-median x (in case of division by 3).

Thus, at least half of the $\lceil \frac{n}{3} \rceil$ groups contain two 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 at least $\frac{n}{3}$.

$$2 \left(\left[\frac{1}{2} \times \frac{n}{3} \right] - 2 \right)$$

\therefore at least $\frac{n}{3}$

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

at least $\frac{n}{3}$ elements are greater than x and at most $\frac{n}{3}$ elements are less than x (not included in the last group)

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

Thus, the recurrence relation that we get is.

$$T(n) \leq T\left(\frac{n}{3}\right) + T\left(\frac{2n+4}{3}\right) + o(n)$$

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

$T(n) \leq cn$ for some large constant c (we ignore terms of higher order, such as n^2 or n^3 for simplicity)
we also assume that, $o(n) = an$ for some constant a

P 192 making

- The recursive relation is given by,

also it means that partition size is multiplied by 3 at each step
 $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + C(2n/3) + \text{constant}$

, addition

+ 8 making

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

, so overall cost for partition is

so overall time complexity is $\Theta(n^2)$

\leq Constructing partition with partition size 3
 based on them get to find root in next 8 steps to compare

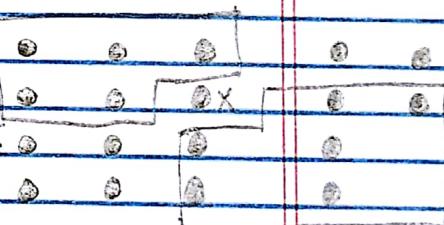
which is at most equality, so it is linear $\Theta(n)$ time complexity

4) The above inequality does not hold for every divide, do not get linear solution but for $n=9$, it will hold even though a partition each of 3 elements will result good enough

5) Therefore it is not possible to get $O(n)$ -time algorithm if we partition the elements with group size 3

(group size h :

$$S = \left\lceil \frac{n}{\lceil \frac{h}{2} \rceil} \right\rceil S$$



We assume that the median of 4 numbers is the second smallest one. When we partition the elements into groups of size h, then at least half of the medians

are greater than the median-of-medians and thus at least $\lceil \frac{n}{h} \rceil$ groups contribute 3 elements that are greater than x, except for the last group that have fewer than h elements. As n doesn't divide h exactly, and the group containing x itself. So, the number of elements greater than x is at least

$$(a)n + (\lceil \frac{n}{h} \rceil - 1)h + (\lceil \frac{n}{h} \rceil)h \geq (a)T$$

partitioning of $\frac{n}{h}$ is at least $\frac{3n}{8}$ but answers don't make any sense, so

Similarly the number of elements that must be less than x is at least $n - h$. In worst case, the select algorithm is recursive

so that $(a) \geq \frac{4}{3}n - 4n$ $a \geq (a)0$, with given value of

$$\text{called on at most } n - \left(\frac{n}{4} - 4\right) = \frac{3n}{4} + \frac{4c}{8}$$

Therefore the recurrence relation is:

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

As explained previously, for groups of 3, we get that

if the size of two recursive calls can be $\frac{n}{3}$ and $\frac{3n}{4}$

which sums up to n . (The running time of the algorithm is not linear but $T(n) = O(n \log n)$. We can prove this by substitution.

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

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

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

which is at most cn if, $4c + an \leq 0$

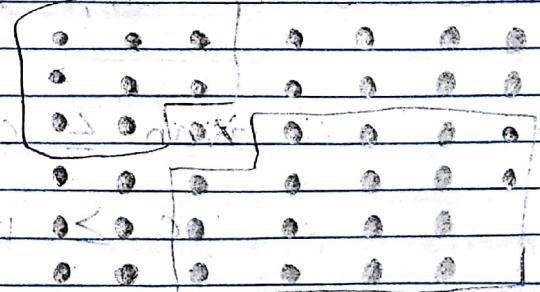
The above inequality does not hold true, and hence we do not get linear solution.

Therefore, if we divide the input elements to groups of 5 elements, the algorithm will not run in $O(n)$ time.

Group size 6: $n = 6k + r$, where r is remainder of division.
Here we assume the median to be the third smallest element.

Similarly, if our group size is 3, then the number of elements greater than x is at least $\frac{n}{3} - 1$.

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



SUMMARY

$$\geq \frac{n-2}{3} \cdot \left(\frac{n-2}{3}\right) = n - 4n + 4 \text{ from formula for } T(n)$$

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

In worst case, the select algorithm is recursively called on at most $n - \frac{n-b}{4} = 3n + b$

total cost $\leq 2T(3n) + (3n+b)T + O(n) = O(nT)$

The recurrence relation is

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

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

$$cn + \left(\frac{n+b}{4}\right)c + (n+1)c \leq (c)T$$

therefore by induction,

$$T(n) \leq cn + nc\left(\frac{3n+b}{4} + 1\right) \leq cn + \frac{12ncn}{4} + nc + bn + c$$

$$\leq \frac{cn}{6} + 3cn + bn + cn$$

$$c \geq cn + bn + cn \text{ from the definition}$$

$$\therefore 12cn \leq cn + bn + cn \text{ which implies } 11cn \leq bn$$

$$\therefore 12an \leq cn + 72cn \text{ since } cn = cn + bn$$

$$\therefore c \geq 12a_n \text{ from the definition}$$

$$c = \frac{12a_n}{n-72}$$

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

Therefore, if we divide the input elements to groups of 6 elements, the select algorithm will run in $\Theta(n)$ -time.

Group size 7:

We assume the median of 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 n does not divide exactly.

The number of elements greater than x is at least

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

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

The number of elements less than x is at least $\frac{5n}{7} + 8$

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

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, $T(n) \leq cn \cdot n + O(n)$ where c and $O(n)$ are large constants.

SELECT NSOT

clear! $T(n) \leq c(n) + c\left(\frac{sn}{7} + 8\right)^2 + an$, sort partition
is constant. But each partition small enough that

to bound it $c n + 5 c n + 8 c + an$. In fact, $c n + 5 c n + 8 c + an$ is at least $c n + 5 c n + 8 c + an$.

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

which is at most cn . It, $\frac{cn}{7} + 8c + an \leq 0$. Then, $c n + 5 c n + 8 c + an \leq cn$.

$$\therefore cn + 56c \geq 7an.$$

$$\therefore c \geq 7af(n)$$

From above inequality, for any $n \geq 56$ 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 5th 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, at least 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

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

Similarly, the number of elements that must be less than x is at least $\frac{5n-10}{18}$. So, in worst case the select algorithm will be called recursively on at most $\frac{13n+10}{18}$ elements.

The recurrence relation is:

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

We assume that $T(n) \leq cn$ and so $c(n) = an$ for some large constant a .

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

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

Therefore by induction, we get a linear time algorithm for selecting x and y from n elements if $n \geq 60$.

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

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

$$\leq cn + 13cn + 10c + an$$

$$\leq 15cn + 10c + an$$

$$\begin{aligned} & \leq \left(\frac{15}{11} - \frac{10}{11}\right)n + \frac{180}{11} \\ & = cn + \left(-\frac{3}{18}cn + \frac{10}{11}c + \frac{180}{11}an\right) \end{aligned}$$

which is at most cn if $-\frac{3}{18}cn + \frac{10}{11}c + \frac{180}{11}an \leq 0$.

$$-\frac{3}{18}cn + \frac{10}{11}c + \frac{180}{11}an \leq 0$$

$$c(n)a + (c(n)a) \frac{10}{11} + (n) \frac{180}{11} \leq 0$$

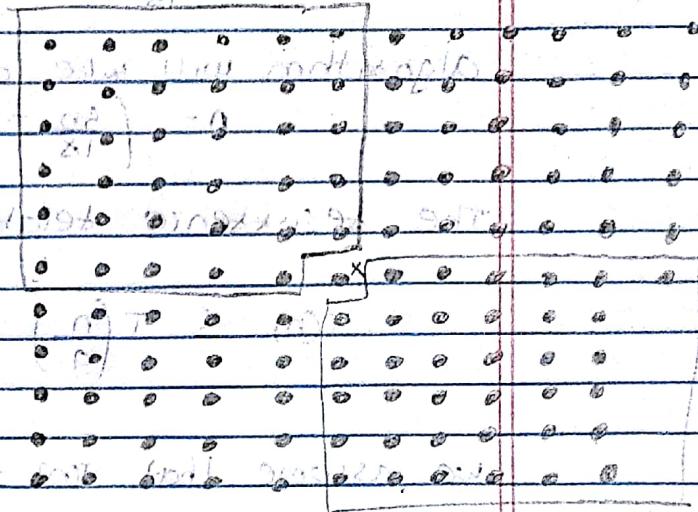
$$-cn + 6ac + 6an \leq 0$$

$$\frac{1}{n} + 6a \leq \frac{1}{n} + 6a \leq \frac{1}{n-60} + 6a \leq \frac{1}{n-60} + 6a$$

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

Therefore the select algorithm will run in $O(n)$ -time if we divide the elements into groups of 11 as follows.

Group size 11: As shown in the following diagram, 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 the medians x . Thus, at least half of $[n/11]$ contribute 6 elements



that are greater than x , except for $n \equiv 0 \pmod{11}$ in which case there is no last group which may have fewer than 11 elements as n is not divided by 11 exactly, and the group containing x itself contains 6 elements.

$$n = (n + 11k) + (n) \leq (n)$$

$$6 \left(\frac{1}{2} \times \frac{n-8k-2}{11} \right) \geq \frac{3n}{11} - 12.$$

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 $n - \left(\frac{3n}{11} - 12\right)$

$$(n + 11k + 11) - \frac{3n}{11} + 12 =$$

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

As $n \geq n_0$, $n \geq 11$, $n \geq 11$ from the condition 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 c for all $n > 0$

so that $T(n) \leq Cn + c$ for all $n > 0$

$$\begin{aligned} T(n) &\leq C\left(\frac{n}{11}\right) + C\left(\frac{8n}{11} + 12\right) + an \quad \text{for } n \geq 11 \\ &\leq \frac{cn}{11} + \frac{8cn}{11} + 12c + an \\ &\leq \frac{cn}{11} + \left(-2cn + 12c + an\right) \end{aligned}$$

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

$$\therefore -2cn + 12 \times 11 \cdot cn + 11an \leq 0 \quad \text{so that is true}$$

$$\therefore cn - 66cn \geq 11an$$

$$c \geq \frac{11a}{2} \times \frac{n-66}{n-11}$$

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, A = 9

Summary:

Group size

$O(n)$ -time - 3

3

$O(n^2)$ condition

5

NO

7

$O(n^2)$ condition

9

$O(n^2)$ condition

11

$O(n^2)$ condition

13

$O(n^2)$ condition

6) let us assume (the two) submatrices to be

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

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, $A_1 = a$	$B_1 = (t - v) A_1 + u$
$A_2 = a + b$	$B_2 = v$
$A_3 = c + d$	$B_3 = s$
$A_4 = d$	$B_4 = (u - s) A_4 + u$
$A_5 = a + d$	$B_5 = s + v$
$A_6 = b - d$	$B_6 = u + v$
$A_7 = a - c$	$B_7 = s + t$

Now we compute the 7 multiplication required

$$\begin{aligned} R_1 &= A_1 \times B_1 = a \times (t - v) \\ R_2 &= A_2 \times B_2 = (a + b) \times v \\ R_3 &= A_3 \times B_3 = (c + d) \times s \\ R_4 &= A_4 \times B_4 = d \times (u - s) \\ R_5 &= A_5 \times B_5 = (a + d) \times s + v \\ R_6 &= A_6 \times B_6 = (b - d) \times (u + v) \\ R_7 &= A_7 \times B_7 = (a - c) \times (s + t) \end{aligned}$$

The resultant matrix C can be computed as follows.

$$e = a \cdot s + b \cdot v$$

$$f = a \cdot t + b \cdot v$$

$$g = c \cdot s + d \cdot v$$

$$h = c \cdot t + d \cdot v$$

From equations in step ③ and ②, we get that

$$as + bv = R_1 + R_5 + R_6 - R_2$$

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

$$R_1 = P_1 = a(t-v)$$

$$R_2 = P_5 = v(a+b)$$

$$R_3 = P_6 = s(c+d)$$

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

$$R_5 = P_3 = (a+d)(s+v)$$

$$R_6 = P_2 = (v-d)(u+v)$$

$$R_7 = P_1 = (a-c)(s+t)$$

$$\begin{aligned} \therefore as + bv - e &= R_1 + R_5 + R_6 - R_2 \\ &= P_7 + P_3 + P_2 - P_5 \end{aligned}$$

$$\begin{aligned} at + bv &= f = R_1 + R_2 \\ &= P_4 + P_5 \end{aligned}$$

$$\begin{aligned} cs + du &= g = R_3 + R_4 \\ &= P_6 + P_7 \end{aligned}$$

$$\begin{aligned} ct + dv &= h = R_5 + R_1 - R_3 - R_7 \\ &= P_3 + P_4 - P_6 - P_1 \end{aligned}$$

Solutions,

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

$$at + bv = P_4 + P_5$$

$$cs + du = P_6 + P_7$$

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

//

- 5) Given an array $A = \{a_1, a_2, a_3, \dots, a_n\}$ 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 pseudocode for the method is

```

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

```

Here, n is 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. A better option

Suppose we divide the array into two sub-arrays, $\{a_1, a_2, \dots, a_{n/2}\}$ and $\{a_{n/2+1}, a_{n/2+2}, \dots, a_n\}$. If we assume that we know the number of inversions 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 subarray inversion, right subarray inversion and the inversions in merge step.

(σ, φ, A) antidiagonal terms + main diagonal off-diagonal
 $\rightarrow [a_{11}, \dots, a_{1n/2}]$ as terms + $[a_{n/2+1, n}]$ as terms
 (σ, φ, A) first column terms + main diagonal - antidiagonal
 (inversion 1) (inversion 2)

inversions in each half

(x,y,z,A) (anagram string array)
we will now understand, the calculation of inversion pairs in
merge step.

left half $\{a_1, a_2, \dots, a_{n/2}\}$ Right half $\{a_{n/2+1}, a_{n/2+2}, \dots, a_n\}$

i^{th} element of left half a_i > a_j then (a_i, a_j) is inversion pair

i^{th} element of right half a_j > a_i then (a_j, a_i) is inversion pair

i^{th} element of left half a_i > a_j then (a_i, a_j) is inversion pair

i^{th} element of right half a_j > a_i then (a_j, a_i) is inversion pair

In merge process, let i denotes 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 $(n/2 - i)$ elements $- i$ in inversions 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.

Now think at how many inversion pair is formed

Therefore the total inversion pairs will be $\text{inversion}_1 + \text{inversion}_2 + \text{merge step inversions}$

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:

1. $\text{count_inversion}(A, p, x)$ starts with p as left most index and x as right most index to count inversions

2. $\text{count_inversion}(A, p, x)$ ends with p as left most index and x as right most index to count inversions

3. $\text{inversion} = 0$ (initial value of inversion) to maintain count of inversions

4. $i = (p + x)/2$ (initial value of internal index for comparison)

5. $i = \lceil (p + x)/2 \rceil$; (final value of internal index for comparison)

6. $\text{inversion} = \text{inversion} + \text{count_inversion}(A, p, q);$

7. $\text{inversion} = \text{inversion} + \text{count_inversion}(A, q + 1, x);$

8. $\text{inversion} = \text{inversion} + \text{count_merge_inversions}(A, p, q, x);$

9. $\text{return } \text{inversion};$

$\text{count_merge_inversions}(A, p, q, x)$

10. $\text{if } \text{left_array_length} = q - p + 1; \text{ then } \text{break}; \text{ as we have seen}$

11. $\text{array_length} = x - q;$

12. $\text{left_array} = [\text{array_length} + 1];$

```

right_array = [array_length + 1]; // for inversion of i-1
for (int i=0; i < array_length; i++) {
    left_array[i] = A[p+i];
}
for (int j=0; j < array_length; j++) {
    right_array[j] = A[q+j];
}
int k=p, l=j+1;
for (int k=p; k < r; k++) {
    if (left_array[k] > right_array[l]) {
        inversion = inversion + array_length - k + 1;
        l++;
    } else {
        l++;
    }
}
return inversion;

```

main part of merge sort, merge arrays and take care of inversions
 (array A to array B) and divide from middle
 if left part need swap then swap and traverse and go to
 left original part it remains and right part half part one and
 $\lceil \frac{r-l}{2} \rceil + \lceil \frac{l-p}{2} \rceil$ most probably has passed

if largest of A and B is less than or equal to $\frac{p+r}{2}$
 then swap it back at same position of S-merge (first part of array)
 traverse to all greatest part S-merge (second part of array)
 $\frac{p+r}{2} - \frac{p+l}{2} + \frac{q+r}{2}$ length of last element from left to right
 if largest of largest then left $\lceil \frac{r-l}{2} \rceil + \lceil \frac{l-p}{2} \rceil$ part of traversal part of array

part of array and right $\lceil \frac{r-l}{2} \rceil + \lceil \frac{l-p}{2} \rceil$ part of array
 if both largest are same $\lceil \frac{r-l}{2} \rceil + \lceil \frac{l-p}{2} \rceil$ part of array
 if left part larger or largest of both part of array then swap the
 $\lceil \frac{r-l}{2} \rceil + \lceil \frac{l-p}{2} \rceil$ part of array part of array

4) $S = \{a_1, \dots, a_n\}$ $B \Rightarrow$ real value
 we need to check whether there 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=i+1; j < array.length - 1; j++) {
        for (int k=j+1; 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 $i \rightarrow$ 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 there do not exist
 3 distinct numbers that satisfies $a_i + a_j + a_k = B$
- let $j \rightarrow i + 1 \rightarrow$ the next element in array.
 $k \rightarrow$ 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 current j^{th} index to move closer to B (or if value of $a_i + a_j + a_k$ is less than B then we repeat the above 3 steps) until $j \leq K$ and if we do not find the 3 numbers then we conclude that there do not exist 3 numbers such in set $S = \{a_1, \dots, a_n\}$ 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;
```

```
        if (array[i] + array[j] + array[k] == B)
            return true;
```

```
        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.

```

for i in range(0, n):
    sum = 0
    for j in range(i, n):
        sum += w[j]
        if sum >= n/2:
            break
    median = i-1

```

median = $i-1$ element

$\sum_{j=i}^n w_j \geq \frac{n}{2}$

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

$$\sum_{j=1}^{k-1} w_j \leq \frac{n}{2}$$

we can prove this by induction. The initial case is the first iteration of the loop when $\sum = 0$. Since the elements are sorted ($x_1 \leq x_2 \leq \dots \leq x_n$) which means that in every iteration of the loop the sum increases by the weight of the element.

we know that $\sum w_i = n$, so the loop will terminate when the sum of all elements is n . 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

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

(since the sum with the last element does not meet the condition, we know that)

$$\text{sum}' + w_{k-1} < \frac{n}{2}$$

The above equation means,

$$w_{k-1} < \frac{n}{2}$$

If the loop has 2.0 elements, then also the above inequality holds true as the sum = 0 is $\leq \frac{1}{2}$.

Hence we proved that first condition of weighted median

$$\sum w_i^o \leq \frac{1}{2}. \quad (2)$$

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

$$\text{for } i < k, \sum w_i^o + w_k + \sum_{i>k} w_i^o = 1; \text{ other strategy.} \quad (3)$$

$$\sum_{i>k} w_i^o = 1 - \left\{ \begin{array}{l} \sum_{i>k} w_i^o - w_k \\ \sum_{i>k} w_i^o \end{array} \right\} \quad \begin{array}{l} (\text{eqn A}) \text{ no harm holding} \\ (\text{eqn B}) \text{ no harm} \end{array}$$

$$= 1 - \sum_{i>k} w_i^o - w_k \quad (A) \text{ no harm} \rightarrow \text{eqn 1}$$

$$\text{no harm} \leq \sum_{i>k} w_i^o = \text{last line} \quad (B)$$

\Rightarrow no harm line

\Rightarrow $a \leq b$

\Rightarrow eqn A no harm

From (3), we get that

$$\sum_{i>k} w_i^o + w_k \geq y_2.$$

$$\therefore \sum_{i>k} w_i^o + w_k \geq y_2 - w_k \quad \begin{array}{l} (\text{eqn A} \rightarrow \text{last line}) \\ \text{no harm} \leq \sum_{i>k} w_i^o \end{array}$$

\Rightarrow eqn A = last line

From eq (3) & (6), we get

$$1 - \sum_{i>k} w_i^o - w_k \leq \frac{1}{2} \quad \begin{array}{l} \text{no harm} + \text{no harm line} \rightarrow \\ \text{no harm} \leq \frac{1}{2} \end{array}$$

$$\therefore \sum_{i>k} w_i^o + w_k \leq \frac{1}{2} \quad \begin{array}{l} (\text{eqn A} \rightarrow \text{last line}) \\ \text{no harm} \leq \sum_{i>k} w_i^o \end{array}$$

In a similar fashion we proved the second condition of weighted median, therefore we can say that x_k exists.

3) v) 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 $\frac{1}{2}$, each then the weighted median is x_k .
- Else, the weighted median will be in half whose sum is $> \frac{1}{2}$.
- The half whose sum is $< \frac{1}{2}$ is lumped into the weight of x_k .
- continue searching in the half whose sum $> \frac{1}{2}$.

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

weighted median(A, sum)

$n = \text{length}(A)$

$\text{sum} = 0$

$\text{median} = \text{Median}(A)$

$\text{left_half} = A[i] < \text{median} : i = 1$

$\text{right_half} = A[i] \geq \text{median}$

$\text{right_median} = 0$

If $n = 1$

return $A[1]$

for (int $i=0$; $i < n$; $i++$)

If $A[i] < \text{median}$

$\text{right_median} = \text{right_median} + w_i$

$\text{left_half} = A[i]$

else

$\text{right_half} = A[i]$

If $\text{right_median} + \text{sum} > \frac{1}{2}$

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 $\text{sum} = 0$.

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

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

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

running time is $\Theta(n)$

total complexity of merging also $\Theta(n)$ because, after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

so total complexity is $\Theta(n)$ because after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

so total complexity is $\Theta(n)$ because after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

so total complexity is $\Theta(n)$ because after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

so total complexity is $\Theta(n)$ because after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

so total complexity is $\Theta(n)$ because after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

so total complexity is $\Theta(n)$ because after partitioning into two halves, it takes $\Theta(n)$ time to merge them. so total complexity is $\Theta(n) + \Theta(n) = \Theta(n)$

2) Closest point pair problem - 3D

In order to derive an algorithm, we will simply extend the 2D algorithm. The point set P is represented by $X = \{x_1, x_2, \dots, x_n\}$ and $Y = \{y_1, y_2, y_3, \dots, y_n\}$ and $Z = \{z_1, z_2, \dots, z_n\} \in \mathbb{R}^{3n}$.

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

- If n is small, or the ≤ 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 $P_R = \lfloor n/2 \rfloor$

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

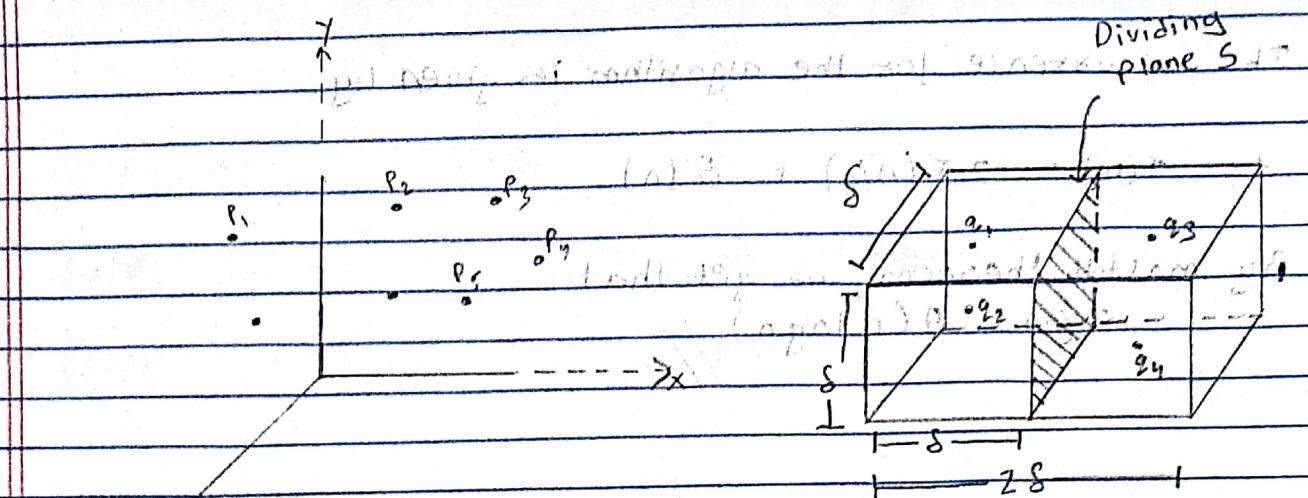
- Now, find the closest point pair in P_L such that $\delta_L = \text{dist}(P_L^1, P_L^2)$ where P_L^1, P_L^2 be the point pair with the smallest distance in P_L

- Similarly we find the closest point pair in P_R such that $\delta_R = \text{dist}(P_R^1, P_R^2)$ where P_R^1, P_R^2 be the point pair with smallest distance in P_R .

- Then find the minimum of two smallest distances found above. $S = \min(\delta_L, \delta_R)$. 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 $2s$ and height s and depth of s .

- let P' be $\{q_1, q_2, \dots, q_n\}$ 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 $\leq s$. Therefore we compute the distance between point pairs 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 $\delta' \leq s$, the distance computed in the above step is the shortest distance and hence we find the closest pair.

- If $\delta' > s$, the the minimum distance found in P_L or P_R will be $\delta + \text{the shortest distance}$

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

8/18/2022

The recurrence for the algorithm is given by,

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

By master theorem, we get that

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