Some Basic divide & Conquer scenario.

→ I hope you know merge sort + quick sort + binary/ ternary search by now.

1 **Closest pair of points:** You are trying to find the smallest distance between two points in a given array.

Lets say we have total of 10 points, (p) each having (x,y) co-ordinate value.

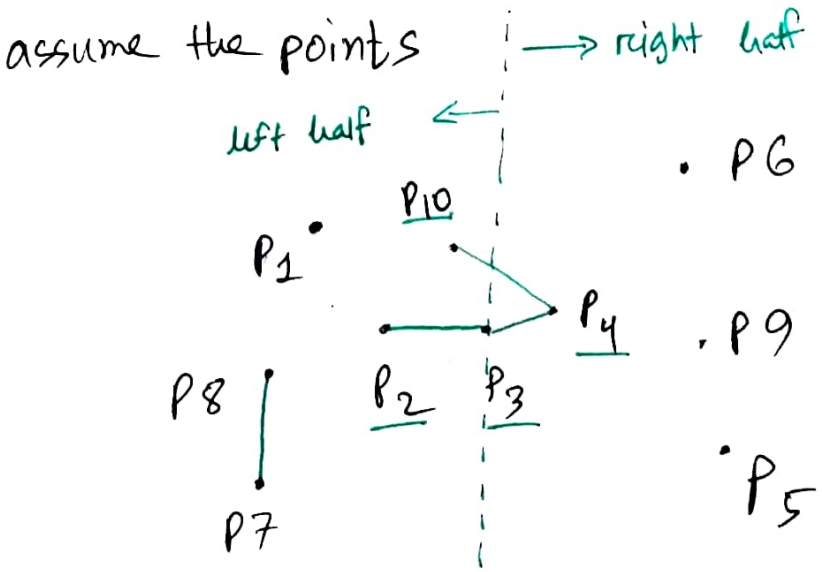
$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

we need to find the smallest $d(P_i, P_j)$

#approach 1: brute-force: use nested loop for two point and check all the combination ⇒ $(O(n^2))$

ideas → you can sort the array with x-dimention & check the distance of y

→ can use divide & conquer.



using divide & conquer if we find the closest pair in the right half of the plain and similarly closest pair in the left half, it may solve the problem. for an example; from right half ⇒ (P_4, P_3) and left half ⇒ (P_8, P_7). now the lesser of these too will be the answer & we can just divide it recursively and add all the answer to get the minimum.

However!! If you take a look at the pair (P_4, P_{10}) which has one point in both halves; that's the issue we are ignoring. So the steps will be:

Step 1: L → left half of P (in the example, 1,2,8,7,10)
R → right half of P (" " " , 3,4,5,9,6)

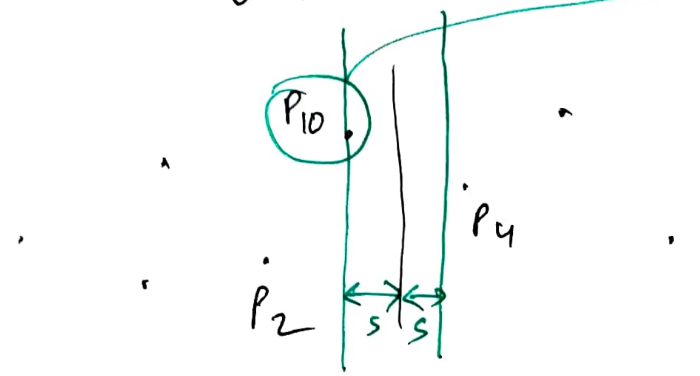
Step 2: find (L_1, L_2) → closest pair in all points of L
find (R_1, R_2) → " " " " " " R

Step 3: $S = \min \{ d(L_1, L_2), d(R_1, R_2) \}$

Step 4: find (M_1, M_2) → closest split pair who intersected in both side (P_x, P_y, S) ⇒ the range for off-set is the S we calculated earlier, since >S will not be closest anyways.

Step 5: return $\min\{\underbrace{d(L_1,L_2), d(R_1,R_2)}_{S}, d(M_1,M_2)\}$

Elaborating step 4, find any terminal point closest to the boundary.



here from left side, biggest x value holder $P_{10}(\bar{X})$ will be selected. not with the value of S in step 3 the range of step 4's search will be $\{(\bar{X}-S, \bar{X}+S), Y\} = S_Y$ where y is sorted. [this sort can be bypassed if it's sorted at the beginning]. now find the pair with the smallest distance (P_L, P_R) where, P_L belongs from the left side, P_R belongs from the Right side.

(#) base case can be if the total number of points are a constant small enough, just calculate it bruteforce. (points number <=3)

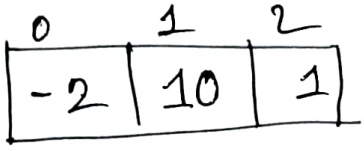
optimizing step 4 is the key for better performance and (nlogn) solution. It's easier said than done.

#2 Maximum Sub-array Sum:

Sub-array means one continuous part of an array which is >=1 of length. & <= length(array)

to visualize the problem:

consider an array of length 3:



⇒ sub arrays are → {[-2], [10], [1], [-2,10], [10,1], [-2,10,1]}

max{subarray} = 11 ⇒ [10, 1]

Sol^N ⇒ bruteforce

```
ans = -∞
for (subsize = 1; subsize <= n, subsize++) {
    for (ind = 0; ind < n; ind++) {
        if (ind + sbsize > n) break
        sum = 0
        for (i = ind; i < (subsize + ind); i++) {
            sum += arr[i]
        }
        ans = max(ans, sum)
    }
}
```

⇒ $T(n) = O(N^3)$

SolN^2 → optimized.

You can bypass the inner loop because you dont need to calculate from scratch for each subarray. example:

[1|2|3|4] → subarray sum = 10

↓

[1|2|3] + 4 → sum of subarray 3 + [4]

↓

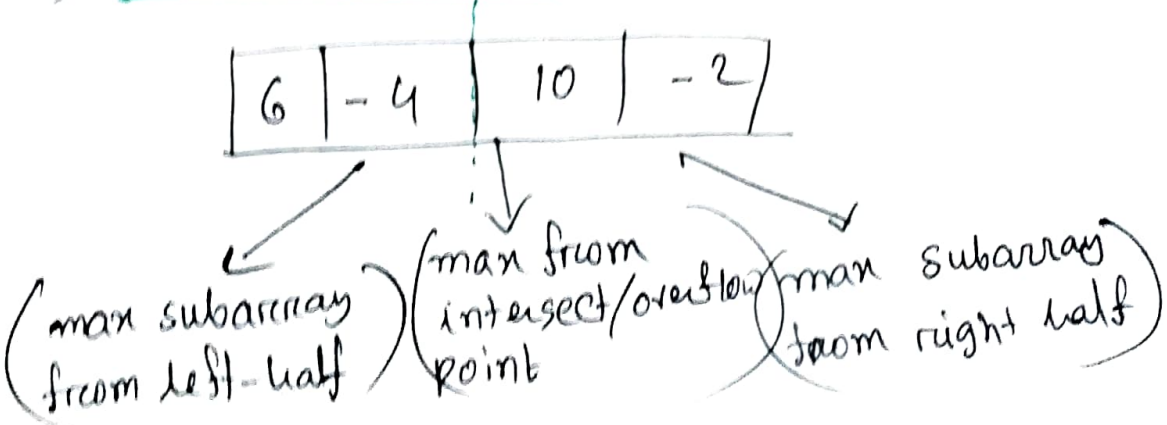
[1|2] + 3 → sum of subarray 2 + [3]

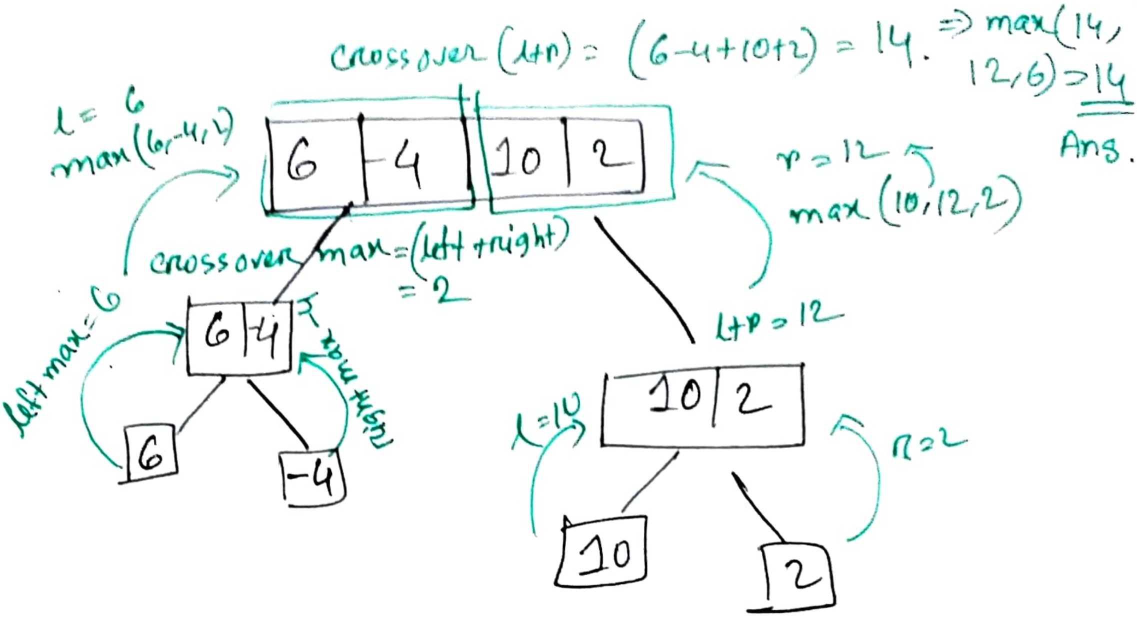
↓

[1] + 2 → " " " 1 + [2]

so, we can go from the bottom to the top and use the smaller subarray sum to calculate the bigger one. $T(n) = O(N^2)$

Sol^N 3 ⇒ divide and conquer

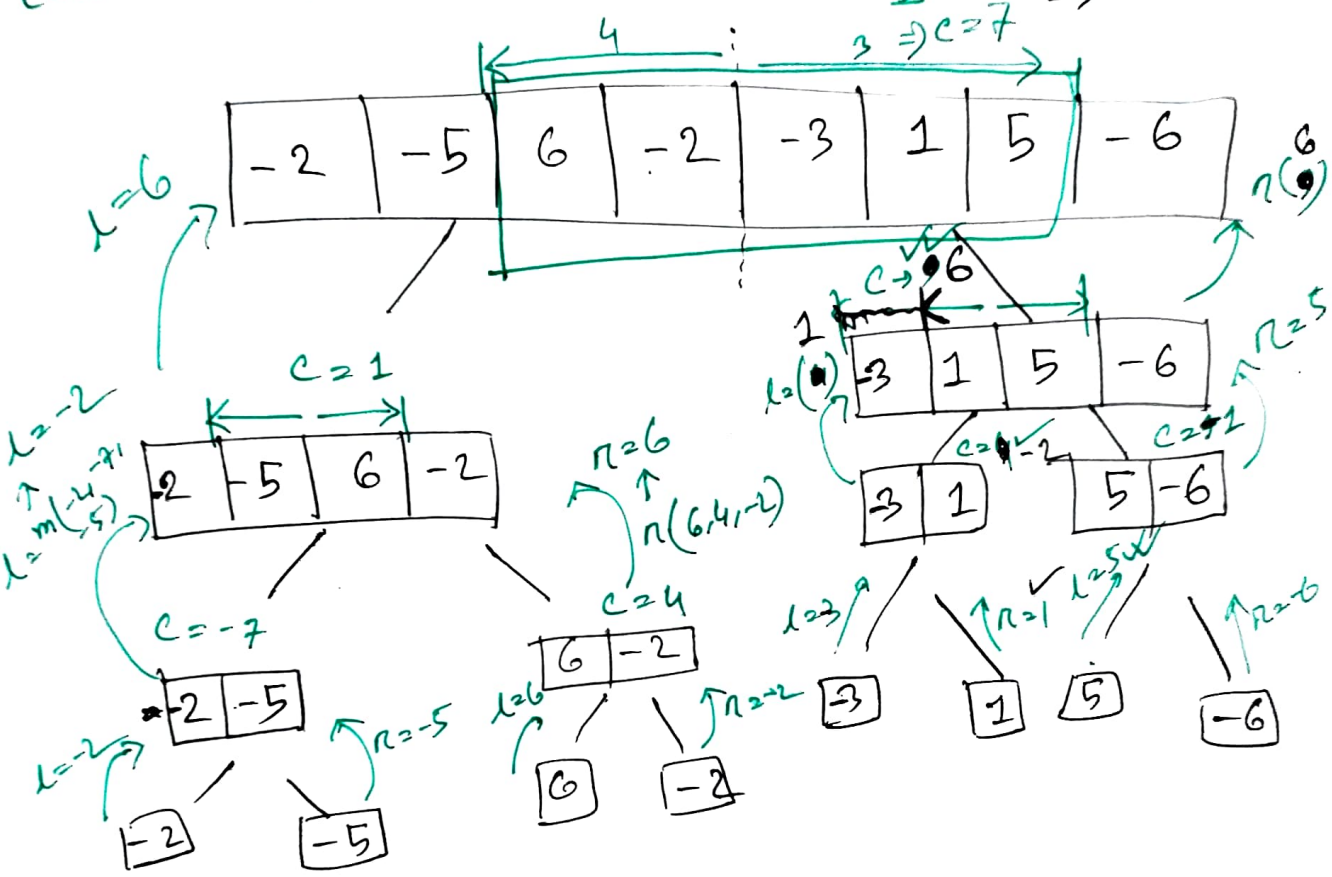




Pseudocode:

```
maxsub (arr): /(arr, start, end)

    if len(arr) == 1:
        return arr[0]

    l_max = maxsub (arr, 0, len(arr)//2)
    r_max = maxsub (arr, len(arr//2), len(arr))

    for (i = len/2; i <= len ; i++){        ⊛
        sum += arr[i]
        r_sum = max(r_sum, sum)
    }

    for (i = len/2 ; i >= 0 , i--) {        ⊛
        sum += arr[i]
        l_sum = max(l_sum, sum)
    }
return max (l_max, r_max, (r_sum + l_sum))
```

* keep in mind, the crossover sum only stores the max sum it can obtain by going from one half to another. It must cross over the mid line! another example:
[you can call it as maximum continuous sum]



Integer multiplication:

Let's see the traditional approach:

```
      4 5 6 7
    x 1 2 3 4
  ------------
      1 8 2 6 8   -->  (~ 2n operation for each digit
    1 3 7 0 1 -   -->  of input n; depending how many
    9 1 3 4 - -   -->  carry bits are adding)
  4 5 6 7 - - -   -->
  ------------
  5 6 3 5 6 7 8
```

n rows

[each row → 2n ~]
total → $2n^2$ ~ (approx.)

Summing all values from top to bottom also takes ~ $2n^2$

given for each digit n additions are performed and maximum n carry bits can be there; for n digit → ~ $2n^2$

So, the total time cost/run time for a multiplication of 2 n digit number : $\theta(n) = \sim 4n^2$

$$\Rightarrow \boxed{T(n) = O(N^2)}$$

Divide & Conquer Approach: [Karatsuba Multiplication]

Observation:

$x =$ [45 (a)] [67 (b)]

$y =$ [12 (c)] [34 (d)]

So, $x = 4567 = a \times 100 + b \Rightarrow \underset{a}{45} \times 10^2 + \underset{b}{67}$

$$\Rightarrow \boxed{x = a * 10^{n/2} + b}$$

Similarly, $y = c * 10^{n/2} + d$

So, $x * y \Rightarrow (a * 10^{n/2} + b) \times (c * 10^{n/2} + d)$

$\Rightarrow (a * 10^{n/2}) \times (c * 10^{n/2}) + (ad * 10^{n/2}) + (bc * 10^{n/2}) + bd$

$\Rightarrow \boxed{ac} * 10^{(n/2)\cdot 2} + \boxed{(ad+bc)} * 10^{n/2} + \boxed{bd}$ → Karatsuba Equation!

now, if we can calculate these 4 multiplications it can be done in divide & conquer since its getting $10^{n/2}$ each time, thus breaking it further will divide itself by another 2, almost like "merge-sort" fashion. but still, 4 multiplication each level is not the efficient approach.

The generalized version of Karatsuba has 3 steps.

(i) find $\boxed{ac}$

(ii) find $\boxed{bd}$

(iii) find $\boxed{(a+b)(c+d) - ac - bd}$

↳ deriving; $ac + ad + bc + bd - ac - bd$

$\Rightarrow \boxed{ad+bc}$; now making it an only 3 multiplication task instead of 4.

If we look at the code,

```
karatsuba(x, y):
    if x<10 or y<10:
        return x* y
    else:
        n = max_digit(x, y)
        h = n//2
        a = x// (10^h)
        b = x% (10^h)
        c = y// (10^h)
        d = y% (10^h)
        ac = karatsuba(a, c)
        bd = karatsuba(b, d)
        ad_bc = karatsuba(a+b, c+d) - ac - bd
        return ac*(10^(2*h)) + ad-bc * (10^h) + bd
```

Pseudocode

$$T(n) = 3T(n/2) + \Theta(n)$$

$$\Rightarrow T(n) = \Theta(n \log_2^3) \Rightarrow \underline{\Theta(n^{1.58})}$$

its a considerable jump_up from the traditional one. in values where $n \to \infty$, this will be more efficient.