NTUCSIE ADA2015

Homework #1, Problem 1, 2, 3

B03902027 王冠鈞

Problem 1

(0) References:

[1] Slides of 3rd week for class 1

[2] http://stackoverflow.com/questions/7280224/solving-the-recurrence-relation-tn-%E2%88%9An-t%E2%88%9An-n

[3] http://stackoverflow.com/questions/25905118/finding-big-o-of-the-harmonic-series

[4] Observe with Geogebra

[5] Discussed with B03902028

(1)

(a)

Assume that there exists a set of solution n_0, c_1, c_2, such that when $n \geq n_0, f(n) = 16f\left(\frac{n}{4}\right) + 514n \leq c_1 n^2 - c_2 n$

Prove by mathematical induction:

(i) The inductive step:

Suppose $f(n) \leq c_1 n^2 - c_2 n, for\ n = m-1, m-2, \ldots$

when $n = m$,

$$f(n) = 16f\left(\frac{n}{4}\right) + 514n$$

$$\leq 16\left(c_1\left(\frac{n}{4}\right)^2 - c_2\left(\frac{n}{4}\right)\right) + 514n$$

$$= (c_1 n^2 - 4c_2 n) + 514n$$

$$= c_1 n^2 - c_2 n - (3c_2 - 514)n, \text{(take any } c_2 = \frac{514}{3}, \text{such that } (3c_2 - 514)n \geq 0)$$

$$\leq c_1 n^2 - c_2 n$$

(ii) The initial condition:

When $n = 1, f(n) = 16 \cdot 1 + 514 \cdot 1 = 530$

if $n_0 = 1, c_2 = \left\lceil \frac{514}{3} \right\rceil = 172, 530 \leq c_1 \cdot n_0 - c_2 \cdot n_0 = c_1 \cdot 1 - 172 \cdot 1,$

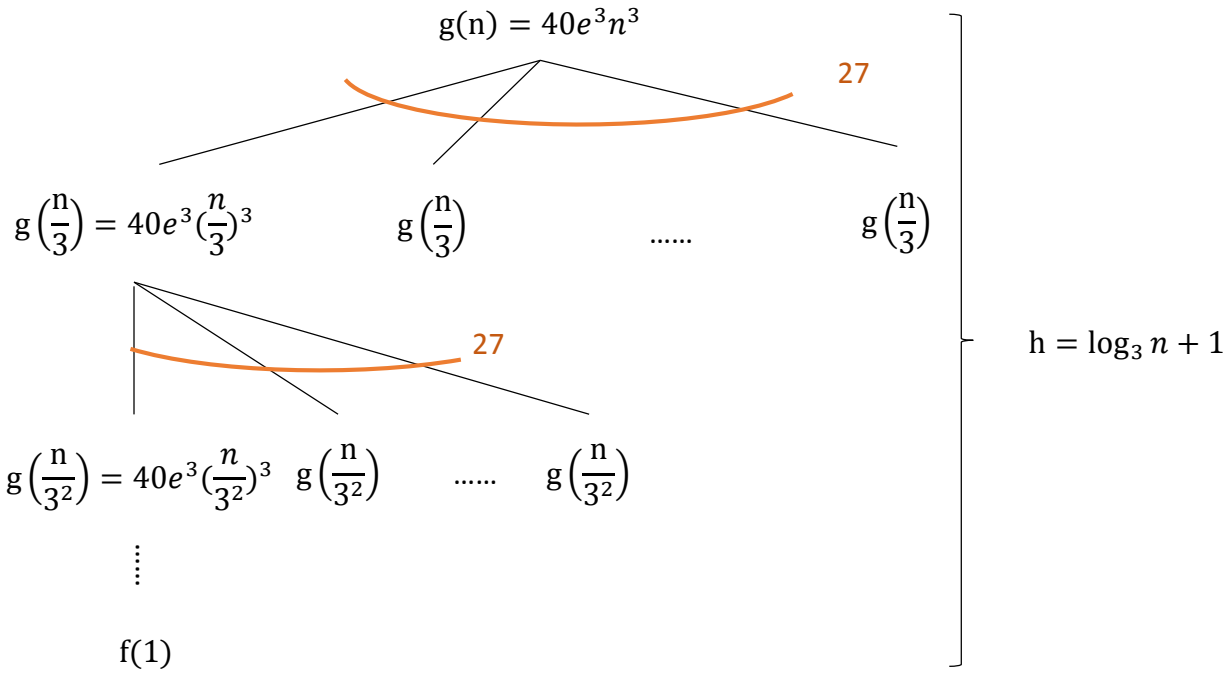
$\Rightarrow c_1 \geq 702$

Thus, we can take $\begin{cases} n_0 = 1 \\ c_1 = 702 \\ c_2 = 172 \end{cases}$ *as a solution.*

(iii) From(i), (ii), $f(n) = O(n^2)$

(b)

Recursion tree of $f(n) = 27f\left(\frac{n}{3}\right) + 40e^3n^3$:



$$Total = 40e^3n^3 + 27 \cdot 40e^3\left(\frac{n}{3}\right)^3 + 27^2 \cdot 40e^3\left(\frac{n}{3^2}\right)^3 + \cdots + 27^{\log_3 n} \cdot 40e^3 f(1)$$
$$= (\log_3 n + 1) \cdot 40e^3n^3 = O(n^3 \log n)$$

(2)

#Classes appearing eariler have lower complexities

(a) $O(1)$

(i) $n^{\frac{1}{\lg n}}$

(ii) 2147483647

(iii) 2^{10000}

(b) $O(\log\log n)$

(i) $\lg \ln n$

(c) $O(\log n)$

(i) $f(n) = f(n-1) + \frac{1}{n}$

(ii) $\sum_{i=1}^{n} \frac{1}{i}$

The complexity of the harmonic series can be proven by Squeeze Theorem.

(d) $O(n^{\ln \sqrt{2}})$

(i) $\sqrt{2}^{\ln n}$

(e) $O(\frac{n}{\ln n})$

(i) $\frac{n}{\ln n}$

(f) $O(n)$

(i) $e^{\ln n}$

(ii) $\frac{10n}{e}$

(g) $O(n \log \log n)$

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

(ii) $en \lg (\ln n)$

(i) is known from reference [2].

(h) $O(n \log n)$

(i) $\ln n!$

(ii) $n \ln n$

(iii) $n \lg n$

(i) $O(n^{\log e})$

(i) $f(n) = e\, f(\frac{n}{2})$

(j) $O(n^{\frac{3}{2}})$

(i) $n^{\frac{3}{2}}$

(k) $O(n^3)$

(i) $e^5 n^3 - 10n^2 + e^{1000}$

(l) $O(n^k), 3 < k < 4$

(i) $f(n) = f(n-1) + n^e$

I guess that it's between $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = O(n^3)$ and $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} = O(n^4)$, because $2 < e < 3$

(m) $O(a^n), 1 < n < 2$

(i) $\mathrm{f(n)} = f(n-1) - f(n-2)$

The formula of finding the nth element in the fabonacci sequence is basically an exponential function with base 1.62... .

(n) $\mathrm{O}(\mathrm{n}^{\ln\log n})$

(i) $(\lg n)^{\ln n}$

(o) $\mathrm{O}(\mathrm{n}^{\log\ln n})$

(i) $\mathrm{n}^{\lg\ln n}$

(p) $\mathrm{O}(\mathrm{n}^{\log\log n})$

(i) $\mathrm{n}^{\lg\lg n}$

For (n), (o), (p), the result is because: $\lim_{n\to\infty} \log\log n > \lim_{n\to\infty} \log\ln n > \lim_{n\to\infty} \ln\log n$

(q) $\mathrm{O(n!)}$

(i) $\mathrm{n!}$

$\mathrm{n!}$ is sopposed to be the fastest-growing function among the 24 functions.

Most of the functions are abnormally complicated, so I tried to plot those functions with Geogebra, so most of the results (functions containing logarithms in its exponents in particular), are based on the observation.

Problem 2

(0) References:

[6] Hints for problem 2

[7] Discussed with B03902028, 030, 079

(1)

(a)

Description:

Spilt the array A into two subarrays of sizes $\left\lfloor \frac{\mathrm{A.size}}{2} \right\rfloor$ and $\mathrm{A.size} - \lfloor \frac{\mathrm{A.size}}{2} \rfloor$. Use recursive method and return the majority elements of subarrays. Only the majorities of the subarrays may become the majority of the original array since an element i appears p, q times respectively in the two subarrays, and $\mathrm{p} < \lfloor \frac{\lfloor \frac{\mathrm{A.size}}{2} \rfloor}{2} \rfloor, \mathrm{q} < \lfloor \frac{\mathrm{A.size} - \lfloor \frac{\mathrm{A.size}}{2} \rfloor}{2} \rfloor$, then $\mathrm{p} + \mathrm{q} < \left\lfloor \frac{\lfloor \frac{\mathrm{A.size}}{2} \rfloor}{2} \right\rfloor + \left\lfloor \frac{\mathrm{A.size} - \lfloor \frac{\mathrm{A.size}}{2} \rfloor}{2} \right\rfloor \leq \frac{\mathrm{A.size}}{4} + \frac{\mathrm{A.size}}{4} = \frac{\mathrm{A.size}}{2}$, which will not be an majority element.

Pseudocode:

```
#"NONE" isn't equal to any other elements in the array
Majority-Element(array A)
if A.size = 1
    return A[0]
Array B[⌊A.size/ 2⌋]
for i = 0 to ⌊A.size/ 2⌋- 1
    B[i] = A[i]
Array C[A.size - ⌊A.size/ 2⌋]
    for i = ⌊A.size /2⌋ to A.size – 1
        C[i - A.size/ 2⌋] = A[i]
left-maj = Majority-Element(B)
right-maj = Majority-Element(C)
count-left = 0
count-right = 0
for i = 0 to A.size
    if A[i] = left-maj
        count-left++
        if count-left > ⌊A.size/ 2⌋
            return left-maj
    else if A[i] = right-maj
        count-right++
        if count-right > ⌊A.size/ 2⌋
            return right-maj
return NONE
```

(b)

Description:

Apply a ***two-pass examination***. Pair up the elements in array A into $\lfloor n / 2 \rfloor$ pairs, where $n = A.\text{size}$ (and of course, if n is odd, there will be an ***isolated element***(e_i), which will be mentioned later). Assume there is a majority element a_M, and for convenience, we call an element a_p a ***most-paired element*** if a_p is the only element which has more *majority pairs* (a ***majority pair of k*** is a pair whose two elements are both k) than any other element. Then, the relations between e_i, a_M, a_p, n are:

$$\begin{cases} a_M = a_p, \ if\ n \equiv 0\ (mod\ 2) \\ a_M = a_p, \ if\ n \equiv 1\ (mod\ 2) \wedge (Quantity(a_M) > \left\lceil \frac{n}{2} \right\rceil \vee (Quantity(a_M) = \left\lceil \frac{n}{2} \right\rceil \wedge a_M \neq e_i)) \\ a_M = a_p \vee a_p = NONE, \ if\ n \equiv 1\ (mod\ 2) \wedge Quantity(a_M) = \left\lceil \frac{n}{2} \right\rceil \wedge a_M = e_i \end{cases}$$

We can easily show that when $e_i \neq a_M$, a_M is always the most-paired element. Consider a majority element p, and another non-majority element q. Assume that p has the same number majority pairs as q, and let it be m. Then,

$$Quantity(p) \leq 2m + \left(\left\lfloor \frac{n}{2} \right\rfloor - m - m\right) = \left\lfloor \frac{n}{2} \right\rfloor, if\ e_i \neq p$$

, which becomes a contradiction.

After the observations, we can pair up the array and apply the *first-pass examination* in order to find a_p, which will be the only *"majority element candidate"*, in other words, the only possible element to become the majority. However, there is a special exception if $a_p = \text{NONE} \wedge n \equiv 1\ (mod\ 2)$. In this case, we will apply the *isolated element*, e_i, as the majority element candidate. The complete algorithm will be shown below.

After finding the candidate, we need a *second-pass examination* to figure out whether it is the genuine majority element. If not, the result will be NONE.

The total elements examined are 2n, which is $O(n)$ time.

Pseudocode:

```
Majority-Element(array A)
candidate = NONE, count = 0
#First-pass examination
for i = 0 to ⌊A.size/ 2⌋ - 1
    if A[2*i] = A[2*i + 1]
        if candidate = A[2*i]
            count++
        else if candidate = NONE
            candidate = A[2*i]
            count++
        else
            count−-
            if count = 0
                candidate = NONE
#Exceptation check
if A mod 2 ≠ 0 and candidate = NONE
    candidate = A[A.size - 1]
```

```
#If A.size is even and candidate is NONE, there is no
#majority.
if candidate = NONE
    return NONE
#Second-pass examination
count = 0
for i = 0 to A.size – 1
    if A[i] = candidate
        count++
if count > ⌊A.size/ 2⌋
    return candidate
else
    return NONE
```

(2)

(a)

Ans: $O(k^2n)$

Description:

We need to merge $(k-1)$ times. For the ith time to merge, there are $in + n = (i+1)n$ elements. Since the arrays are sorted, we only need $O(n)$ time to merge two arrays with a total of n elements. After $(k-1)$ times of merger, the aggregated time is,

$$2n + 3n + 4n + \cdots + kn$$

$$= \sum_{i=1}^{k} (i+1)n = \frac{(k+2)(k-1)}{2} n = \frac{n}{2}k^2 + \frac{n}{2}k - n = O(k^2 n)$$

(b)

Description:

Spilt the arrays into two groups of half sizes recursively until one array, and then merge back, like the *mergesort*.

Pseudocode:

```
Spilt-and-Merge(k array(s))
if k = 1
    return array
array1 = Spilt-and-Merge(the first ⌊k⌋ arrays)
array2 = Spilt-and-Merge(the remaining (k - ⌊k⌋) arrays)
merged-array = SequentialMerge(array1, array2)
return merged-array
```

Time Complexity: $O(nk\ log\ k)$

The algorithm basically satisfies the recurrence:

$$T(n,k) = T\left(n,\frac{k}{2}\right) \times 2 + nk$$

Thus the complexity is $O(nk\ log\ k)$, according to the master theorem.

Problem 3

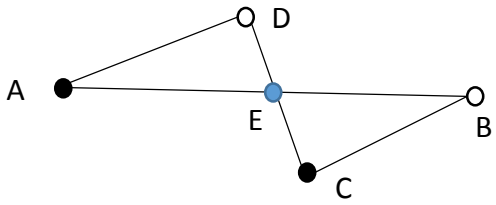
(0) References:

[8] Hints for problem 3

[9] http://mks.mff.cuni.cz/kalva/putnam/psoln/psol794.html

[10] Discussed with B03902028, 079

(1)



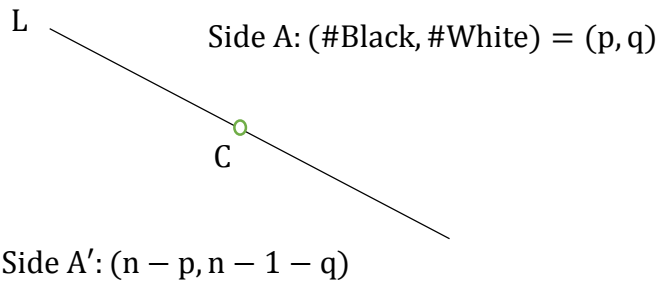
Consider a set of finite points satisfing the problem's condition. Since the number of the points is finite, there must exist a match which makes the total length of line segments a minimum. Now consider a minimum match. Assume segment $\overline{\mathrm{AB}}, \overline{CD}$ are two of the components in a minimum match, then $\overline{\mathrm{AB}} + \overline{CD} > \overline{AD} + \overline{CB}$, since the sum of two sides in a triangle is larger than the third. Thus, segment $\overline{AD}, \overline{CB}$, which don't intersect each other, can make the total length even smaller, which contradicts with the previous assumption. As a result, a minimum match, which always exists, is also a good-word match, which implies a good-word match also always exists.

(2)

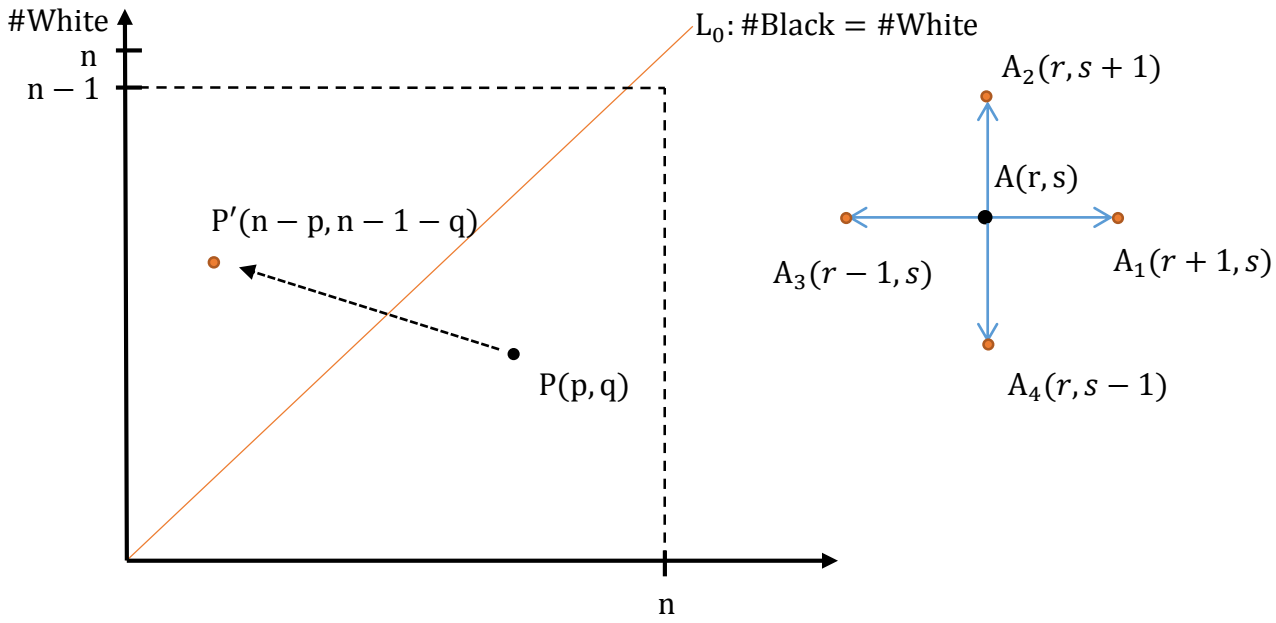
Description:

First, I'd like to show that whichever point we choose as the pivot, there is at least one valid solution. Without loss of generality, let the pivot be an arbitrary white point, so among the remaining points, black points have one point more than white ones.

Select an arbitrary white point C, and draw a line L without passing through any points other than C. Define one of the side to be side A, which initially



has p black points and q white points, where $0 \leq p \leq n, 0 \leq q \leq n-1$, as the figure shown. Then, mark $P(p,q), P'(n-p, n-1-q)$ on a cartesian coordinate. Without loss of generality, let $p > q$, then $n-1-q \geq n-p$, and they're equal only if $p-q=1$. Thus P' must be on the 45-degree line(L_0) or on the opposite side from P.

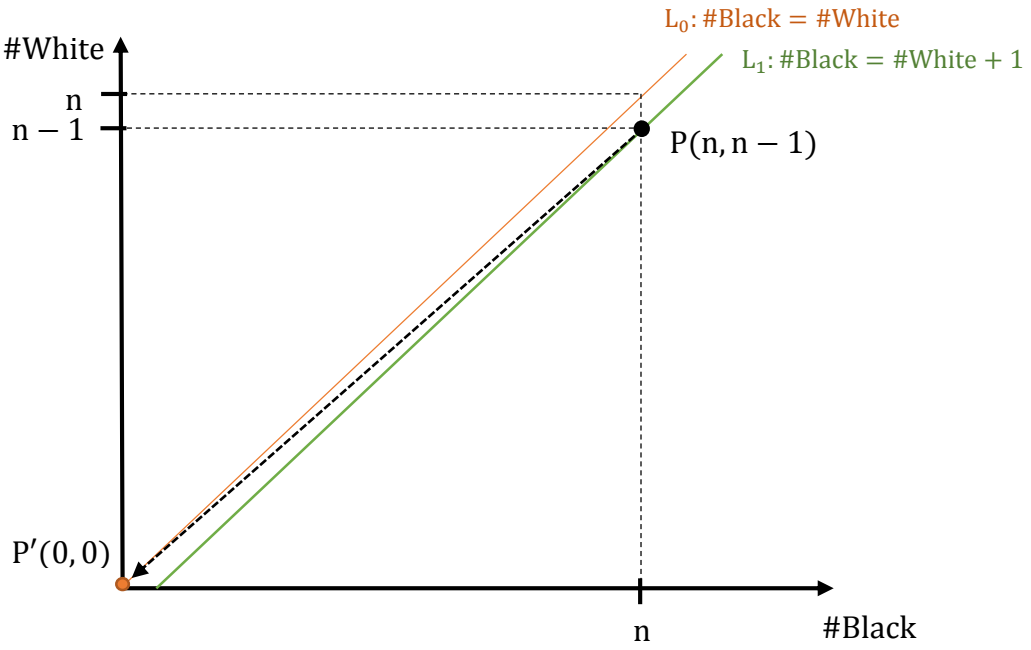


If we rotate line L for $180°$, the number of points on side A will become P' on the coordinate above. During the process of rotating, the number of the points will vary, but it will eventually fall on P' at the end, which means that if we record the number of points on the coordinate system every time the number varies, and connect the vertices, it will become a path from P to P'. Moreover, according to the axiom that "*no three points on the same line*", for every changes, it is only possible to add or subtract $\#White$ or $\#Black$ by 1, as point A above. In other words, the point will only move horizontally or vertically for 1 unit. Combining the results above, the path from P to P' must intersects with the 45-degree line on a vertex which denotes one of the states during the rotation. In other words, whichever point we choose, we can always find a line such that the number of the black points is equal to the number of the white on one of the sides.

Then, let's consider a special case. If one of the states is $P(n, n-1), P'(0,0)$, although vertex P' is on the 45-degree line, it violates the restriction that *"Both sides of L should be nonempty"*, so it cannot be a solution. We must show that even such circumstances happen, there still always exists a valid solution.

Consider a line $L_1: \#Black = \#White + 1$ on the coordinate system. Since every point on this line always satisfies $p-q=1$, so if a vertex V is on L_1, there must exist another vertex V' denoting the number of the points on side A' that is on L_0.

Thus, when we find a path from P to P', the path must pass through L_0 and L_1, meaning that there must exist a valid solution at somewhere on the path. Thus, there must contain at least one valid solution for any point in the set.



After confirming the existence of the solution, we can design an algorithm of $O(n \log n)$ with the following steps:

(i) Choose an arbitrary point. $\longrightarrow O(1)$

(ii) Calculate the directions from the chosen point C to all the other points, and save the directions (same as "θ" in a polar coordinate) in degree in an array. $\longrightarrow O(n)$

(iii) Sort the array. $\longrightarrow O(n \log n)$

(iv) Set the *begin* and *end* iterators such that *begin* points to the beginning place of the array and the *end* points to the first index whose element is not smaller than *begin + 180*, and if it doesn't exist, return the end. $\longrightarrow O(\log n)$, with binary search

(v) Record the *"#Black - #White"* values of two sides. $\longrightarrow O(n)$

(vi) Move the iterators and modify the *"#Black - #White"* values, until one reaches 0 and the elements are not empty. $\longrightarrow O(n)$

(vii) Return the result.

The total time complexity is decided by the *sorting* step, which is $O(n \log n)$.

Pseudocode:

```
#Return a pair(pivot, line angle(0 <= angle < 180))
#Structure PointData consists of: color, coord, angle
```

```
Find-Line(PointData Array S)
PointData pivot = random-point(S)
array[S.size - 1]
for i = 0 to S.size – 2
    if array[i] ≠ pivot
        array[i].coord = S[i].coord
        array[i].color = S[i].color
        array[i].angle = find-angle(pivot, array[i].coord)
sort(array)
#find-end: using binary search (O(logn))
#count-diff: Count the #Black - #White value in O(n)
begin = 0
end = find-end(array)
diff1 = count-diff(array, begin, end)
diff2 = 1 – diff1
current = array[0].angle
#Let array[array.size].angle = infinity
while diff1 ≠ 0 and diff2 ≠ 0
    if array[begin + 1] – current < array[end] – 180 – current
        if array[begin].color = white
            diff1++
            diff2--
        else
            diff1--
            diff2++
        begin++
        current = begin.angle
    else
        if array[end].color = black
            diff1++
            diff2--
        else
            diff1--
            diff2++
        end++
        current = end.angle - 180
    if (diff1 = 0 and end – begin ≠ 1) or (diff2 = 0 and end –
    begin ≠ array.size)
```

```
#array-copy: copy the angle-sorted array to S sequentially and
#put the pivot at the end(optional), taking O(n) time
       array-copy(array, pivot, S)
       return (pivot, (current + min(array[begin + 1] – current,
       array[end] – 180 – current)) / 2)
```

(3)

Description:

Use divide-and conquer to divide the set of points by using line-finding algorithm recursively until the subset reaches 2. The worst case occurs when the line divide 2 elements apart from the others for every set-spiltting process, which will become $O(n^2 \log n)$.

Pseudocode:

```
Find-GWM(array S)
if S.size = 2
    output(S)
    return
pair result = Find-Line(S)
#divide-set: return the subset divided by line L, and
#without pivot Since S is sorted, it takes O(n) time
#get-remain: return the other subset containing the pivot
subarray1 = divide-set(S, result)
subarray2 = get-remain(S, result)
Find-GWM(subarray1)
Find-GWM(subarray2)
```