

소스코드 및 주석(코드 설명)

```

소스.cpp
6주차 실습 closest pair
1  #include <iostream>
2  #include <cmath>
3
4  typedef struct pair // std::pair를 쓰면 안될것 같아서 구조체로 했습니다.
5  {
6      int first;
7      int second;
8  };
9
10 void swap(pair& a, pair& b) // pair 자료형에 맞는 swap함수
11 {
12     pair temp = a;
13     a.first = b.first;
14     a.second = b.second;
15     b.first = temp.first;
16     b.second = temp.second;
17 }
18
19 int partition(pair* arr, int l, int r, int order) // quickSort할 때 구역을 나눌 기준을 생성하는 partition 함수.
20 {
21     pair pivot = arr[l]; // 왼쪽의 항을 피봇으로 선정한다.
22     int small = l + 1; // small을 pivot보다 작은 항을 두는 곳으로 한다.
23     if (order == 0) // x좌표를 기준으로 정렬
24     {
25         for (int i = l + 1; i <= r; i++)
26         {
27             if (arr[i].first < pivot.first)
28             {
29                 swap(arr[i], arr[small]); // small의 위치에 pivot.first 보다 작은 arr[i]와 바꿈.
30                 small++;
31             }
32         }
33         swap(arr[l], arr[small - 1]); // 이걸 해야 이제 pivot으로 선택된 배열의 크기상의 위치가 정해지는 것이다.
34         return small - 1; // 다음 피봇을 return해줌.
35     }
36     else // y좌표를 기준으로 정렬
37     {
38         for (int i = l + 1; i <= r; i++)
39         {
40             if (arr[i].second < pivot.second)
41             {
42                 swap(arr[i], arr[small]); // small의 위치에 pivot.second 보다 작은 arr[i]와 바꿈.
43                 small++;
44             }
45         }
46         swap(arr[l], arr[small - 1]); // 이걸 해야 이제 pivot으로 선택된 배열에서의 pivot의 위치가 정해지는 것이다.
47         return small - 1; // 다음 피봇을 return해줌.
48     }
49 }
50
51 void quickSort(pair* arr, int l, int r, int order)
52 {
53     if (l < r) // l < r이라는건 아직 정렬이 다 안됐다는것.
54     {
55         if (order == 0) // x좌표를 기준으로 정렬하겠다.
56         {
57             int next_pivot = partition(arr, l, r, 0);
58             quickSort(arr, l, next_pivot - 1, 0);
59             quickSort(arr, next_pivot + 1, r, 0);
60         }
61         else // y좌표를 기준으로 정렬할 것이다.
62         {
63             int next_pivot = partition(arr, l, r, 1);
64             quickSort(arr, l, next_pivot - 1, 1);
65             quickSort(arr, next_pivot + 1, r, 1);
66         }
67     }
68 }
69
70

```

```

70
71 // ex) arr[3] arr[4] arr[5] 일때 l = 3, r = 5;
72 double find_min(pair* arr, int l, int r) // 나누어진 구역의 점이 2개나 3개일 때 그 점들 사이의 거리 중 최솟값을 반환하는 함수.
73 {
74     if (r - l + 1 == 3) // 포함된 점이 3개일 때
75     {
76         double one = sqrt(pow(arr[l + 1].first - arr[l].first, 2) + pow(arr[l + 1].second - arr[l].second, 2)); // 첫번째, 두번째 점 사이의 거리.
77         double two = sqrt(pow(arr[l + 2].first - arr[l + 1].first, 2) + pow(arr[l + 2].second - arr[l + 1].second, 2)); // 두번째, 세번째 점 사이의 거리.
78         double three = sqrt(pow(arr[l + 2].first - arr[l].first, 2) + pow(arr[l + 2].second - arr[l].second, 2)); // 첫번째, 세번째 점 사이의 거리.
79         if (one <= two && one <= three) // 각각의 거리중 최소인 점을 반환함.
80             return one;
81         else if (two <= one && two <= three)
82             return two;
83         else
84             return three;
85     }
86     return sqrt(pow(arr[l + 1].first - arr[l].first, 2) + pow(arr[l + 1].second - arr[l].second, 2)); // 점이 두개라 그 거리가 최소임.
87 }
88

```

```

89
90 double find_middle(pair* arr, int l, int r, double mid_range) // 좌측 구역과 우측 구역, 가운데 구역으로 나뉘었을 때 가운데 구역의 점을 중 최솟값을 구하는 함수.
91 { // 이 함수를 재귀적으로 호출할 때, arr, l, r은 각각 구역의 범위, 좌측 구역의 범위, 우측 구역의 범위를 나타내며, mid_range는 현재 구간에서의 최솟값을 나타낸다.
92     double min = mid_range; // 최솟값에 func함수에서 나오는 좌측 구역, 우측 구역의 최솟값들 중 최솟값인 mid_range를 두고 mid구역에 있는 점들의 거리와 비교하여 점들의 거리가 더 작으면 min을 최솟값.
93
94     quickSort(arr, l, r, 0); // x좌표를 기준으로 정렬하고.
95     for (int i = l; i <= r - 1; i++)
96     {
97         for (int j = i + 1; j <= r; j++)
98         {
99             if (arr[j].second - arr[i].second >= mid_range)
100                 break;
101             if (sqrt(pow(arr[i].first - arr[j].first, 2) + pow(arr[i].second - arr[j].second, 2)) < min)
102                 min = sqrt(pow(arr[i].first - arr[j].first, 2) + pow(arr[i].second - arr[j].second, 2));
103         }
104     }
105     return min;
106 }
107
108

```

```

소스.cpp
6주차 실습 closest pair
(전역 범위)
find_middle(pair* arr, int l, int r, double mid_range)
106
107
108
109 double func(pair* arr, int l, int r) // l : 범위의 왼쪽 범위 번호, r : 범위의 오른쪽 범위 번호 arr[l]-arr[r]의 범위로 나뉘.
110 { // 이 함수를 재귀적으로 호출할 때, arr, l, r은 각각 구역의 범위, 좌측 구역의 범위, 우측 구역의 범위를 나타내며, mid_range는 현재 구간에서의 최솟값을 나타낸다.
111     int elements = r - l + 1; // 배열에 포함된 점의 개수가 3개 이하이면 거리의 최솟값을 구하는 함수.
112     if (elements <= 3)
113     {
114         return find_min(arr, l, r);
115     }
116
117     // 배열에 포함된 점들의 개수가 3보다 작다면 좌측, 우측, 중앙으로 나누어서 좌측, 우측은 각각에 대해서 이 함수를 재귀적으로 호출함.
118     int mid = (l + r) / 2;
119     double left = func(arr, l, mid);
120     double right = func(arr, mid + 1, r);
121
122     double mid_range; // mid_range는 left의 최솟값과 right의 최솟값 중 더 작은 수를 기준으로 중앙에서 찾을 x값의 범위를 정함.
123     if (left < right)
124         mid_range = left;
125     else
126         mid_range = right;
127
128     // mid_range를 통해서 좌측 구역의 가장 오른쪽인 arr[mid]부터 arr[mid+1], arr[mid+2]로 막 나뉘어지면 x좌표의 차이가 mid_range보다 커지는 걸 바로 전 점을 mid구역의 제일 좌측점.
129     // arr[mid+1]부터 arr[mid+2]로 막 나뉘어지면 x좌표의 차이가 mid_range보다 커지는 걸 바로 전 점을 mid구역의 제일 우측 점으로 해서 그 점들 사이의 거리 중 최솟값을 구한다.
130     int mid_left = mid - 1;
131     int mid_right = mid + 2;
132     while (arr[mid_left].first - arr[mid].first > mid_range && mid_left != 0) // mid 구역의 left값을 찾기 위한 loop
133         mid_left--;
134     while (arr[mid_right].first - arr[mid + 1].first > mid_range && mid_right != r) // mid 구역의 right값을 찾기 위한 loop
135         mid_right++;
136     double middle = find_middle(arr, mid_left, mid_right, mid_range); // 중앙의 점들을 비교할 알고리즘이 필요함. -> find_middle로
137     if (mid_range < middle) // 좌측, 우측 구역의 최솟값과 중앙의 최솟값을 비교하여 더 작은 값을 반환함.
138         return mid_range;
139     else
140         return middle;
141 }
142

```

```

142
143 int main()
144 {
145     int n; // 전체 점의 개수.
146     std::cin >> n;
147
148     pair* arr = new pair[n]; // x좌표와 y좌표를 같이 보관할 구조체 pair를 만들고, pair동적 배열을 n개 만들.
149
150     int i = 0; // 입력받는 점들의 상을 i = 0으로 함으로써 첫번째부터 받게 함.
151     char comma; // 입력에 ,가 있는걸 받는 용도.
152
153     int repeat = n; // while문을 n번 반복
154     while (repeat--)
155     {
156         std::cin >> arr[i].first >> comma >> arr[i].second;
157         i++;
158     }
159
160     quickSort(arr, 0, n-1, 0); // x좌표를 기준으로 오름차순 정렬하기.
161
162     printf("%.6lf", func(arr, 0, n-1)); // 소수점 6번째까지 받기 위해서 printf로 출력함.
163     // 구역을 arr[0]-arr[n/2]까지와 arr[n/2 + 1]-arr[n-1]까지로 나눈다.
164     // 구역에 남아있는 점의 개수가 3개 이하가 될 때까지 나누고 그것들의 거리들을 구해서 그중 최솟값을 얻어낸다.
165
166     delete[] arr; // 동적배열 해제 해주고.
167 }

```

시간 복잡도 계산.

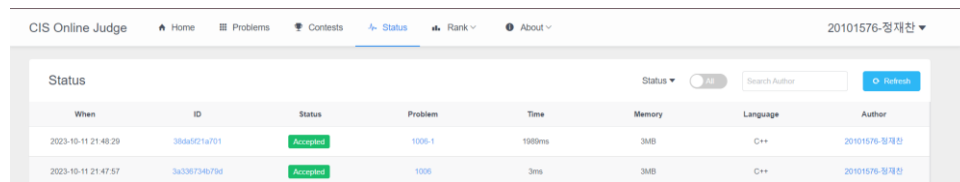
쿼정렬의 시간 복잡도 : $O(n \log n)$

한 구역의 최솟값을 구하하려고 func함수를 실행 할 때마다 3개의 부분문제로 쪼개지고, 좌측, 우측 구역의 2개의 부분문제는 문제의 크기가 $n/2$ 가 되기 때문에 좌측 + 우측의 시간복잡도는 $2 \times O(\log n)$ 이다.

마지막으로 중앙 구역의 최솟값을 구하는 함수의 시간복잡도는 함수가 한번 실행될 때마다 y축을 기준으로 배열을 정렬하는 데 $O(n \log n)$ 이 걸리고, 점들 사이의 거리를 비교하는 데 걸리는 시간은 범위에 따라 다르므로 일반적으로 $O(1)$ 이 걸린다고 하면 시간복잡도는 $O(n \times \log n)$ 이 된다.

마지막으로 func함수는 한번 실행될 때마다 문제의 크기가 $n / 2$ 로 줄어들으므로 func함수는 $\log n$ 번 실행되게 된다. 그리고 func함수가 실행될 때마다 좌측 : $O(\log n)$, 우측 : $O(\log n)$, 중앙 : $O(n \log n)$ 의 실행을 하므로 총 func함수에 의한 시간복잡도는 $O(n (\log n)^2)$ 이 된다.

정답 인증 사진.



The screenshot shows the 'Status' page of the CIS Online Judge. The page has a navigation bar with links for Home, Problems, Contests, Status (active), Rank, and About. The user is logged in as '20101576-정재찬'. The Status page includes a search bar for the author and a table of submissions.

When	ID	Status	Problem	Time	Memory	Language	Author
2023-10-11 21:48:29	3bda9021a701	Accepted	1006-1	1989ms	3MB	C++	20101576-정재찬
2023-10-11 21:47:57	3a338734b79d	Accepted	1006	3ms	3MB	C++	20101576-정재찬